Trabajo de Fin de Grado

Simulador didáctico de arquitectura de computadores

Didactic simulator for Computer Architecture .

Adrián Abreu González

La Laguna, 2 de junio de 2017

D. **Iván Castilla Rodríguez**, con N.I.F. 78.565.451-G profesor adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

C E R T I F I C A

Que la presente memoria titulada:

"*Simulador didáctico de arquitectura de computadores.*"

ha sido realizada bajo su dirección por D. **Adrián Abreu González**, con N.I.F. 54.111.250-R.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 2 de junio de 2017

Agradecimientos

A Víctor, por aquellas largas noches programando cuando los dos aún teníamos mucho que aprender.

A mi tutor, Iván Castilla, por su paciencia y buena voluntad para echarme una mano siempre que me hiciera falta.

A Diana, por ser capaz de darme siempre otro punto de vista, fuera cual fuera el problema.

Licencia

Resumen

En el campo de arquitectura de computadores resulta difícil transmitir muchos de los fundamentos teóricos mediante los medios de enseñanza tradicionales.

Con el objetivo de servir de apoyo a la docencia, concretamente en el apartado de máquinas superescalares, se ha querido desarrollar un simulador del funcionamiento de las mismas.

Para ello se han utilizado las últimas tecnologías modernas del mundo web tales como Typescript, React, Webpack y Sass, permitiendo aprovechar muchas de las mejoras que se han incorporado a este campo en los últimos años.

El resultado es una aplicación web que representa de forma visual e interactiva el comportamiento de las máquinas superescalares, facilitando así la comprensión de sus fundamentos por parte del usuario.

Palabras clave: Arquitectura de Computadores, Superescalar, Aplicación Web, Typescript, React, Webpack.

Abstract

In Computer Architecture, sometimes is really hard to teach much of the theorical fundaments using the usual teaching ways.

En el campo de arquitectura de computadores resulta difícil transmitir muchos de los fundamentos teóricos mediante los medios de enseñanza tradicionales.

With the purpose of helping teaching, focusing in how superescalar machines works,

Con el objetivo de servir de apoyo a la docencia, concretamente en el apartado de máquinas superescalares, se ha querido desarrollar un simulador del funcionamiento de las mismas.

In order to achieve this goal, it has been used the latest technologies in web development: Typescript, React, Webpack and Sass. Allowing the use of many of the improvements that have been introduced to this field in the recent years.

Para ello se han utilizado las últimas tecnologías modernas del mundo web tales como Typescript, React, Webpack y Sass, permitiendo aprovechar muchas de las mejoras que se han incorporado a este campo en los últimos años.

As result, we have a web application that represent in a visual and interactive way the behavior of superscalar machines, making easier to the users to comprehend the fundaments of those machines.

El resultado es una aplicación web que representa de forma visual e interactiva el comportamiento de las máquinas superescalares, facilitando así la comprensión de sus fundamentos por parte del usuario.

Keywords: *Computer Architecture, Superescalar, Wep Application, React, Typescript, Webpack.*

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Introducción

El peso de la tecnología en la eficiencia de los computadores actuales es innegable. Sin embargo, los conceptos básicos que definen la arquitectura de estos computadores se basan en ideas de hace varias décadas. En el ámbito docente resulta difícil transmitir estos fundamentos, ya que han quedado superpuestos por el aumento de la complejidad de la arquitectura, haciendo que sea imprescindible una gran abstracción por parte del alumno al no poder disponer de las máquinas originales en los que se implementaron.

Las herramientas docentes típicas, como pueden ser una pizarra, un libro de texto o diapositivas, tienen una capacidad limitada para representar los fundamentos ya expuestos.

Este trabajo se centra en el uso de simuladores como medio de apoyo docente para el tema del paralelismo a nivel de instrucción, una parte fundamental en el incremento de rendimiento de las computadoras.

En este contexto, los simuladores juegan una pieza clave en el campo de la Arquitectura de Computadores, permitiendo asociar fundamentos y teorías, simplificando abstracciones y facilitando la labor docente.

1.2. Paralelismo a nivel de instrucción

Para aumentar el rendimiento de los computadores es necesario que se puedan ejecutar varias instrucciones en paralelo.

Desde 1985 la mayoría de computadoras han explotado el paralelismo a nivel de instruccion implementando la técnica conocida como segmentación -una técnica que promueve una disposición concreta de los recursos de la máquina y una divsión de la ejecución de las instrucciones en etapas para conseguir un funcionamiento similar al de una cadena de montaje-. Esta técnica tiene como

límite máximo el rendimiento de 1 ciclo por instrucción (CPI). Para poder superar este límite, es necesario emitir varias instrucciones diferentes de forma simultánea [1].

Para alcanzar este propósito se deben considerar los posibles riesgos que ello implica, al ejecutar multiples instrucciones aparecen diversos riesgos estructurales y dependencias de distintos tipos: control, datos o recursos. Estas dependencias deben ser detectadas y resuelas correctamente.

Existen dos grandes técnicas para conseguir explotar el paralelismo a nivel de instrucción con emisión múltiple:

1. **Planificación dinámica**: La responsabilidad de detectar y resolver los problemas recae en el propio hardware.
2. **Planificación estática**: Delega las decisiones en el software. Manteniendo un hardware relativamente simple.

1.2.1. Máquinas Superscalares

Las máquinas superescalares son aquellas máquinas que cumplen las siguientes características:

- Emiten instrucciones desde un único flujo secuencial de instrucciones.
- Poseen emisión múltiple.
- Hacen uso de la planificación dinámica.

Para resolver los problemas de dependencias se utiliza una modificación del algoritmo de Tomasulo. Este algoritmo nació originnalmente con el objetivo de controlar las dependencias en la unidad funcional de punto flotante de la máquina 360/91 de IBM.

Mediante un seguimiento de los operandos y renombrado de registros se eliminaron los tres tipos de riesgos de datos: *Read After Write*, *Write After Read* y *Write After Write*.

Además, las máquinas superescalares incorporan nuevas estructuras de hardware como el ReorderBuffer para permitir la ejecución fuera de orden.

1.3. Motivación para el trabajo

Como se ha mencionado, el uso de un simulador para apoyar la docencia de esta área de Arquitectura de Computadores resultaba un campo realmente interesante y relevante. De hecho, un simulador que cumpla estas características, ya existe. El profesor de la Universidad de La Laguna, Iván Castilla desarrolló un simulador en C++ con este propósito.

Este simulador se ha estado utilizando como un complemento fundamental en la docencia de Arquitectura de Computadores, pero con el paso del tiempo, ha quedado obsoleto. No tanto por su funcionalidad, puesto que los fundamentos teóricos sobre los que se basa no han cambiado con el tiempo, como por su aspecto visual y su accesibilidad.

Es por esto se ha querido recuperar esta herramienta para continuar con su desarrollo y ampliación y este trabajo de fin de grado se centra en migrar esta aplicación a versión web de tal forma que sirva como base para los futuros proyectos.

Capítulo 2

Antecedentes

2.1. SIMDE

En el año dos mil cuatro, el por aquel entonces estudiante de esta universidad, Iván Castilla Rodríguez - y ahora tutor de este trabajo de fin de grado-, desarrolló como proyecto final de carrera un Simulador didáctico para la enseñanza de arquitectura de computadores, el cual fue bautizado como Simde [2].

Este simulador como se ha comentado en el apartado 1.4 cumple con las características deseadas y esperadas de un simulador para la docencia de este ámbito.

Sin embargo, esta herramienta ya se encuentra desfasada. No ha sido un proyecto en constante evolución, fue diseñada utilizando C++98 y C++ Builder y el código ahora mismo no resultaría fácil de adaptar y mantener.

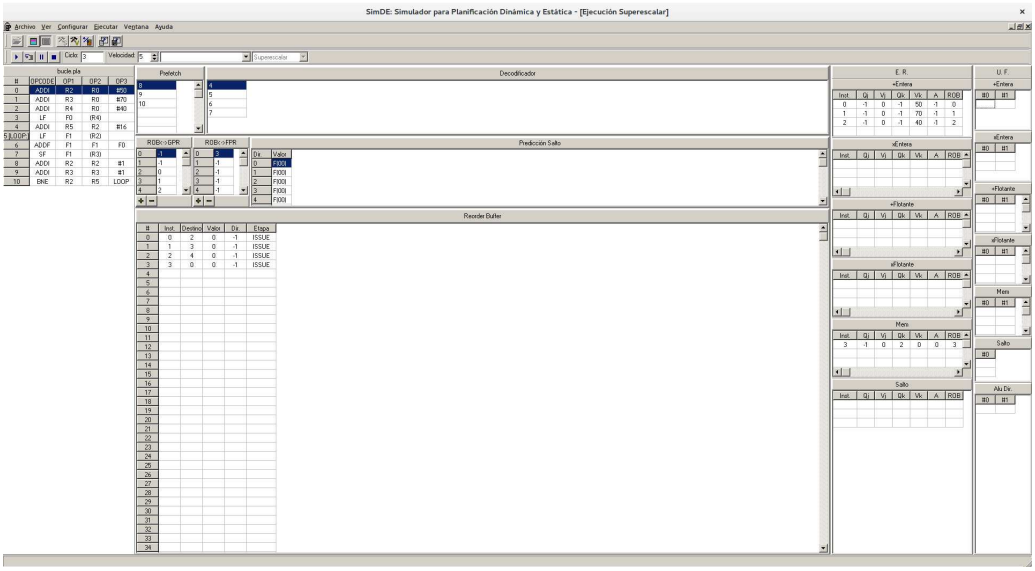

Figura 2.1: Simulador original SIMDE

2.2. Otros simuladores

Sería irrazonable embarcarse en la tarea de migrar SIMDE sin comprobar antes si ya existe alguna herramienta que se adapte a estas necesidades.

Simulador	Descripción
SESC [3]	Simulador de máquinas Superescalares desarrollado en la Universidad de Illinois, posee características tan interesantes como el SMT, el CMP... Pero carece de interfaz gráfica.
ReSIM [4]	Este simulador de máquinas Superescalares está basado en la ejecución por trazas. Se encuentra disponible para múltiples dispositivos Xilink.
SATSim [5]	Simulador de máquinas superescalares con traja que utiliza una simulación interactiva.
VLIW-DLX [6]	Basado en WinDLX y desarrollado en la Universidad de praga, intenta aportar un conocimiento detallado de las máquinas VLIW.

Tabla 2.1: Tabla comparativa de simuladores

Como vemos, aunque existen múltiples simuladores dentro de esta sección, ninguno cumple con el objetivo inicial, muchos se centran más en estudiar los efectos de utilizar un tipo concreto de arquitectura implementando máquinas específicas -incluso careciendo de interfaz-. Y el qué más se asemejaría a nuestro propósito, el SATSim, tiene más de una década.

2.3. Evolución del mundo web

A modo de curiosidad, en el proyecto original, se consideraba poco factible la realización de un simulador de estas características en el entorno debido al escaso rendimiento. Cito textualmente:

> Utilizar programación web. Pese a las muchas ventajas de la programación web en aspectos como la distribución y accesibilidad del programa, la realidad es que el uso que se iba a dar a esta herramienta no invitaba a la utilización de esta opción. El simulador no estaba planteado como una herramienta con arquitectura cliente-servidor, ni tampoco se pretendía una ejecución distribuida. La idea era que se procesara en la máquina del alumno y no en un servidor. El uso de applets de java tampoco era una opción atractiva debido a la lentitud de ejecución de java por ser interpretado [2].

Este argumento totalmente justificado y válido en el momento de su enunciado resulta difícil de defender en un contexto actual. Hoy en día, el rendimiento del lenguaje Javascript en el navegador es increíblemente alto. Y si valoramos que la diferencia de edad entre el documento citado y éste mismo es de unos escasos trece años -aunque trece años en el mundo de la tecnología no es un lapso de tiempo despreciable- se puede observar que la diferencia es asombrosa.

Si intentamos encontrar un origen para esta diferencia tan increíble de rendimiento, sin duda es necesario remontarse a la aparición de las primeras grandes aplicaciones que hacen uso de la tecnología *Ajax* como por ejemplo Gmail [7].

La tendencia a diseñar más y más aplicaciones utilizando esta tecnología resultó en una guerra por aumentar el rendimiento por parte de los distintos navegadores. Que ha acabado con el escenario actual, donde no sólo cada vez más aplicaciones de escritorio tienen una versión web disponible (véase las herramientas de Microsoft Office), si no que además existen muchos frameworks para desarrollar aplicaciones de escritorio a partir de aplicaciones web como *Electron* [8].

Capítulo 3

Objetivos y fases

3.1. Objetivos

Al comienzo de este trabajo fin de grado se tenía una visión abstracta de lo que se quería conseguir. En las primeras reuniones con el director, se concentraron los objetivos del proyecto.

1. Migrar la lógica de la aplicación a la web.
2. Proveer de una nueva interfaz a la aplicación de SIMDE.
3. Recuperar la documentación de SIMDE.

3.2. Fases

El desarrollo del trabajo se ha divido en cuatro fases principales desde un primer momento.

1. **Migración del "núcleo de la aplicación"**: Es decir las estructuras básicas comunes a las máquinas y el punto de entrada para la generación de las estructuras correspondientes, es decir, el analizador léxico de MIPS.
2. **Migración de la máquina superescalar**: El proceso completo de reescribir todas las estructuras de la máquina y su correcto funcionamiento en javascript.
3. **Desarrollo de la interfaz**: El desarrollo tanto de un diseño remodelado con las tecnologías web, como de componentes que gestionen correctamente su propio estado.
4. **Integración interfaz-máquina**: La realización de las conexiones necesarias entre el código desarrollado y la interfaz que permitan permitir le uso del simulador por parte del usuario.

De forma paralela mientras se hacia el desarrollo de la máquina superescalar se creaba un pequeño *wireframe* de la interfaz original.

Y también de forma paralela, tras terminar el *wireframe*, se realizaba la migración de la documentación original.

Capítulo 4

Tecnologías

Uno de los puntos más importantes de este trabajo era determinar las diferentes tecnologías a utilizar. El mundo web ha sufrido una gran revolución en los últimos años y ahora se disfruta de una gran variedad de propuestas para cada tipo de tarea.

Es por ello, que ha habido que hacer una gran valoración del estado del arte tecnológico para determinar qué curso debería seguir la aplicación.

4.1. Lenguaje para la lógica de la aplicación

Por normal general cuando se habla de programación web en el lado del cliente, se tiende a pensar de forma inmediata en Javascript, y en general, este razonamiento es indudablemente válido. Pero dado que SIMDE es una aplicación fuertemente orientada a objetos y con una gran base de código, se han valorado múltiples alternativas con el objetivo de agilizar la realización de este proyecto.

4.1.1. Coffescript

Coffescript es un pequeño lenguaje que se compila en Javascript, su objetivo era mejorar la legibilidad y concisión de Javascript añadiendo varios *syntactic sugars* inspirados en otros lenguajes como *Ruby* o *Python* [9].

Coffescript es un lenguaje con un largo recorrido, apareciendo a finales del año 2009. Y tiene soporte por parte de *Ruby on Rails* y *Play framework* [10].

Coffescript podría ser la opción ideal para agilizar le desarrollo debido a la similitud de sintaxis con Ruby.

```
class Animal
  constructor: (@name) ->

  alive: ->
```

```
    false

class Parrot extends Animal
  constructor: ->
    super("Parrot")

  dead: ->
    not @alive()
```

Sin embargo, la opción de Coffescript ha sido desestimada en gran medida por su decreciente popularidad y la poca certeza del futuro que tomará el lenguaje.

4.1.2. Dart

Dart es un lenguaje de código abierto desarrollado por Google que permite desarrollador aplicaciones web, móvil, de servidor y también se puede utilizar en el *Internet of Things* [11].

Se ha considerado en el desarrollo de esta aplicación porque es un lenguaje orientado a objetos que utiliza una sintaxis similar a C#. Además, aunque Google Chrome tiene una máquina virtual nativa para este lenguaje, es posible transpilar el código a Javascript para los navegadores que no tiene este soporte nativo.

Tras razonar detenidamente, a pesar de que Dart es una opción tremendamente atractiva ha quedado descartado tanto porque se trata de un lenguaje totalmente diferente a nivel sintáctico de lenguajes como Javascript como porque su uso es mayoritariamente por parte de Google y de una reducida comunidad.

4.1.3. Typescript

Typescript es un lenguaje libre y de código abierto desarrollado por Microsoft que actúa como un superconjunto de Javascript, es decir incorpora los distintos estándares: ECMA5, ECMA6, ECMA7... Y además, como característica destacable, añade comprobación de tipos en tiempo de compilación [12].

Este tipado no se refleja en el código final, de hecho una interfaz, por ejemplo, añade 0 sobrecarga en el código final. Pero si que es interesante por las capacidades de autocompletado (a través de Microsoft Intellisense) que añade.

Al final, en este proyecto Typescript ha sido la tecnología seleccionada, y existen múltiples razones:

1. Typescript tiene bastante apoyo por parte de la comunidad y por parte de la propia Microsoft. La documentación es extensa y efectiva.

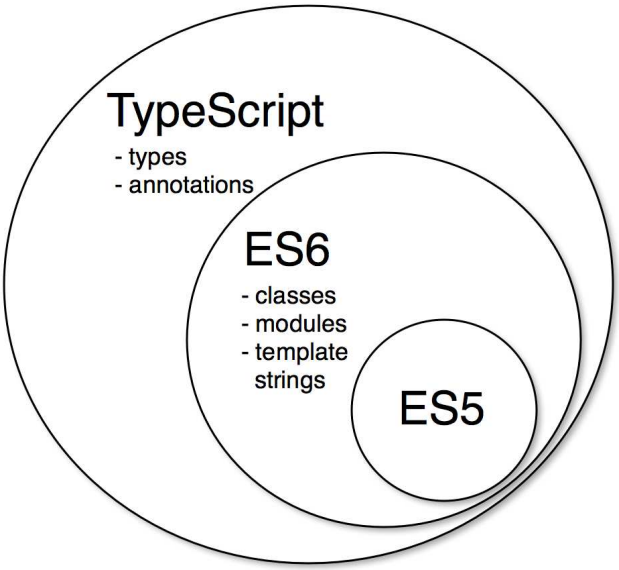

Figura 4.1: Typescript como superconjunto de Javascript

2. Typescript está alineada en cierta forma con el futuro de Javascript. Microsoft es uno de los muchos que forman parate del concenso de estándar de ECMASCRIPT.

3. Typescript no establece ningún límite en la posibilidad de usar Javascript, todo código Javascript es código Typescript válido.

4.2. Tecnología para la integración modelo vista

Desde el inicio del proyecto, se tenía claro que alguna librería se encargaría de gestionar el tedioso proceso de manipular el DOM. Actualmente existen múltiples librerías y frameworks que podían servir para realizar esta tarea, pero muchos de ellos (como por ejemplo Angular), son demasiado *rígidos* y acaban condicionando la forma de desarrollar la aplicación, lo cual resulta ser contraproducente.

4.2.1. Webcomponents

Una de las características más deseadas para el nuevo diseño de la interfaz de SIMDE era contar con un diseño basado en componentes. Siendo la caracteŕistica más deseada de todo el incorporar las nuevas ventajas que ofrecen los Web Components. Los Web Components son un conjunto de características que se están añadiendo a las especificaciones W3C de Html y del DOM [13].

El objetivo de estas características es permitir crear componentes personalizados, reusables y con su propia encapsulación. Esto se consigue a través de cuatro

características principales:

1. **Elementos personalizados**: Esta característica permite diseñar y utilizar nuevos tipos de elementos del DOM.

2. **Shadow DOM**: Esta característica permite al navegador incluir un subárbol de elementos del DOM en el renderizado del documento pero **NO** se incluyen el DOM principal.

3. **HTML Imports**: Esta característica permite incluir y reutilizar documentos HTML en otros documentos HTML.

4. **Plantillas HTML**: Esta característica permite declarar fragmentos de código de marcas que no se utilizan en el carga de la página pero que se pueden instanciar en tiempo de ejecución.

4.2.2. Polymer

Esta librería fue la primera que hizo uso de los Web Components.Desarrollada por Google y anunciada en el año 2013, Polymer permite aprovechar las características de los Web Components [14] a través de los polyfills -códigos que implementan características en los navegadores que no soportan las mismas de forma nativa-. *(Comúnmente se conoce como polyfill a la librería que implementa el estándar de HTML5)* [15].

A pesar de la revolución que marcó, Polymer no fue ampliamente acogida por la comunidad de desarrolladores -quizás por ser una librería adelantada a su tiempo-. Y hoy en día nos encontramos con otro intento por parte de ganar peso en la comunidad: Polymer 2.0. Esta nueva versión incorpora el soporte de las clases de ES6 y además permite utilizar el método de la especificación de custom elements v1 para definir elementos.

Aunque las mejoras que incorpora Polymer 2.0 la hacen una opción totalmente válida y viable, aún no tiene una comunidad lo suficientemente sólida con lo que esto acaba traduciéndose en una menor cantidad de recursos disponibles.

4.2.3. React

La empresa autora de esta librería de código abierto, Facebook, la define como: “It’s a Javascript library for building UI’s” [16].

Pero realmente aunque esta declaración es totalmente cierta, React no es tan algo tan simple. Para resolver el problema de la modificación del DOM de una forma eficiente: Simulándolo esta estructura en memoria y aplicando diversos

algoritmos para calcular cuales serían los mínimos cambios necesarios a realizar sobre el *DOM verdadero* para representar los diversos cambios de estado.

React se basa en el uso de componentes, no en el sentido explícito de los *Web Components* tal como los define el éstandar de HTML5, sino como pequeños bloques reusables que incorporan cierta funcionalidad. Sin embargo, a pesar de que React se puede integrar con la api de *Web Components*, el uso de esta api incrementa de forma exponencial la complejidad de la aplicación, con lo cual se ha pospuesto el uso de esta característica para versiones futuras.

React utiliza un híbrido entre html y javascript denominado jsx, como también tiene soporte para Typescript, en este caso utilizamos tsx.

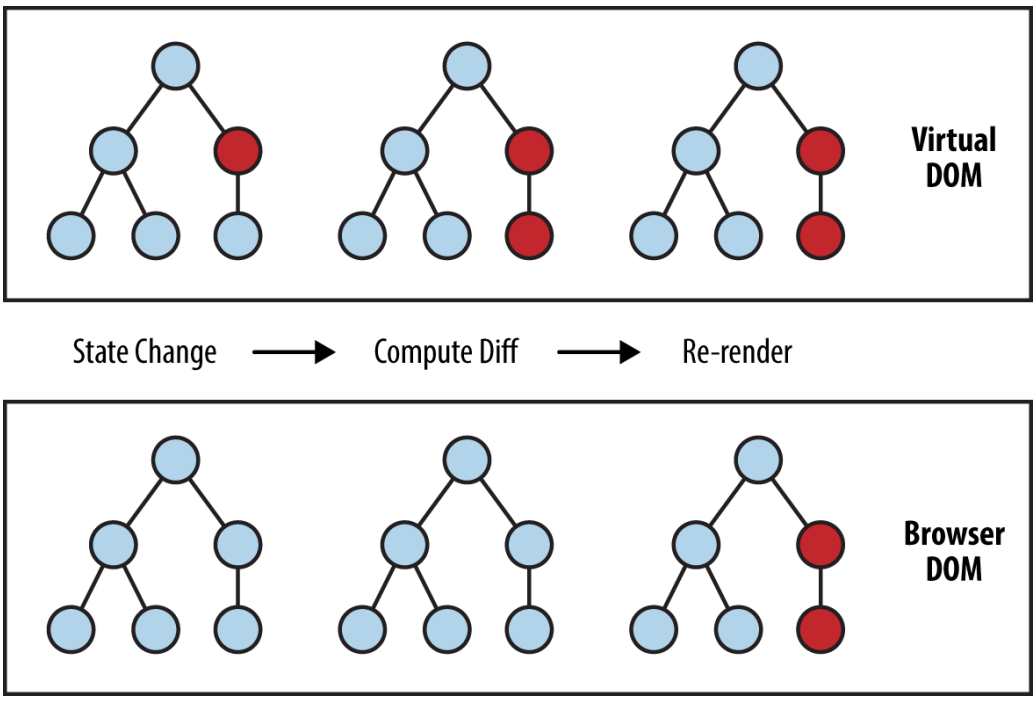

Figura 4.2: Funcionamiento del DOM virtual de React [17]

React es ampliamente utilizada por muchísimas empresas gracias a su capacidad de integrarse con otras librerías. Por ejemplo Microsoft mantuvo parte de la página en jQuery mientras iba integrando React.

Otro ejemplo de grandes empresas que hagan uso de esta librería son: AirBnB, Netflix, Wallmart... [18]

Y muchas de ellas han contribuido al ecosistema de React, ya sea mediante guías de estilo, conjunto de componentes, patrones... [19]

Además,la comunidad de usuarios es increíblemente activa, por ejemplo es común ver a Dan Abramov resolviendo dudas en distintos sitios como *Github* o

Reddit. Dan Abramov es el creador de Redux (una implementación de gestión de estados), desarrollador de la nueva implementación de React *(react-fiber)*, empleado de Facebook, participante en muchas conferencias y además autor de múltiples herramientas como *react-hot-loader*.

Debido a todo lo anterior y a que además, React tiene una excelente integración con Typescript utilizando el formato .tsx, queda claro que es la mejor opción posible para esta aplicación. React es una librería desarrollada por Facebook para construir interfaces.

4.3. Tecnología para el ensamblaje

Debido a la complejidad de las aplicaciones web modernas, es necesario realizar una serie de pasos intermedios entre el código original y el resultado final de la apliación. Para el caso de este proyecto, se debe:

- Compilar el código typescript a javascript.
- Compilar el código .tsx a .jsx.
- Resolver las importaciones de dependencias, tanto de la lógica como de los componentes.
- Procesar el código sass y convertirlo en css.

4.3.1. Gulp/Grunt

La primera tendencia -debido a su gran extensión- sería utilizar lo que se conoce como un *task runner*. Actualmente, dos de los más conocidos son **Grunt** [20] y **Gulp** [21].

Ambos están basados en NodeJs y son compatibles entre sí en gran medida. Su funcionamiento es sencillo, en un gruntfile o gulpfile se definen las tareas a ejecutar, seleccionando los ficheros de fuente sobre los que actuar -si cabe- y la tarea a realizar.

Existen muchisimos plugins desarrollados que permiten hacer todo tipo de tareas, desde traducir markdown hasta minimizar el contenido de los ficheros de estilos y de javascript.

Sin embargo, a pesar de que esta opción era altamente atractiva debido a su robustez, se ha optado por probar una solución aún más moderna, **webpack**.

4.3.2. Webpack

Webpack es un *empaquetador de módulos* para aplicaciones de Javascript modernas. Cuando webpack procesa la aplicación, construye un grafo de dependencias incluyendo todos los módulos y luego lo empaqueta en orden [22].

Webpack incorpora de serie diversas características interesantes tales como el poder ejecutar un servidor de desarrollo que aplica *hot reloading* sobre el código sin requerir refrescar la página o el poder separar el código css/js de terceros para el entorno de producción.

El funcionamiento de webpack puede ser extremadamente resumido y simplificado en:

- Partiendo de un punto de entrada, una serie de reglas sobre los distintos tipos de ficheros activan una serie de *loaders* correspondientes para procesarlos.
- Estos loaders pueden ser concatenados entre sí para obtener el resultado deseado, por ejemplo podemos traducir el código typescript a es6 para luego traducir este código junto a otro a es5 mediante babel.
- Se aplican, si fuera necesario, el uso de plugins para tareas más complejas que se quieren aplicar sobre todos los paquetes.

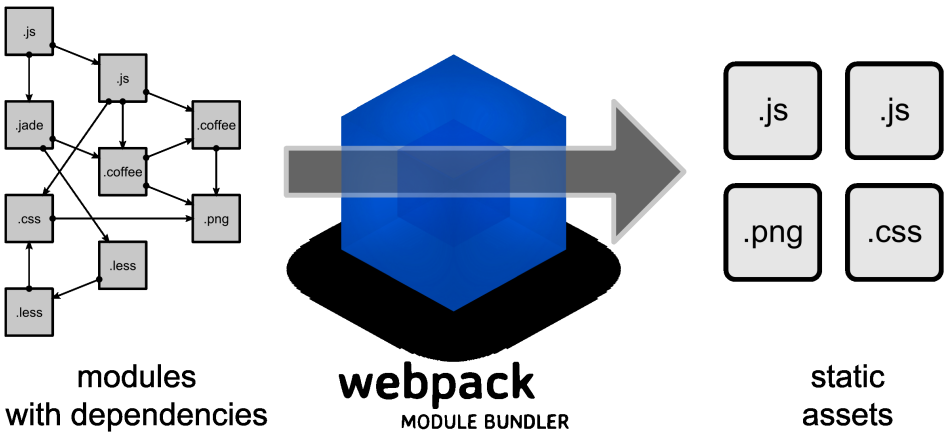

Figura 4.3: Imagen descriptiva de Webpack

Como resultado final se obtiene una serie de paquetes que contienen todas las dependencias resueltas.

4.4. Tecnología para la documentación

Para integrar la documentación en la nueva aplicación web de SIMDE resultaba obvio que esta documentación estuviera también en formato web. Para esto existían muchas alternativas, desde un conjunto de ficheros html hasta un pequeño sistema de gestión de contenidos.

Dado que la documentación es bastante extensa pero que en realidad, no es más que un documento que se redactará en una ocasión y se le irán realizando pequeñas ampliaciones y/o correcciones se optó por una solución diferente, los generadores de contenido estático.

4.4.1. Generadores de contenido estático

Los generadores de contenido estático se encargan -resumido de forma breve- de generar un conjunto de ficheros *HTML* y *CSS* a partir de una plantilla y una serie de ficheros de fuente, básando normalmente en un formato de entrada común. Actualmente este formato es el *Markdown* [23].

Este tipo de generadores estáticos tienen un gran auge entre los desarrolladores que desean mantener un blog, sin tener que estar desplegando bases de datos o teniendo que mantene múltiples ficheros html.

Existen múltiples ventajas de utilizar este tipo de tecnologías, una de las más importantes es que se alimentan de un formato como es el markdown. El cual es realmente intuitivo y tiene una amplia acogida más allá de este tipo de tecnologías.

4.4.2. Hexo

Hexo es un generador de contenido estático basado en NodeJS [24]. No posee demasiadas diferencias destacables sobre el resto de alternativas, y ha sido escogido para este proyecto principalmente porque al estar basado en Javascript todo queda enfocado hacia un mismo ecosistema dando una sensación de homogeneidad con el resto del proyecto.

Capítulo 5

Desarrollo del proyecto

5.1. Proyecto base

Para llevar a cabo este proyecto se ha decidido trabajar con un proyecto de NodeJS que gestionara las dependencias. Se ha utilizado como base el repositorio de github *typescript-starter* [25]. Este proyecto incluía las dependencias necesarías para empezar a trabajar en la lógica, como el compilador de Typescript y el *test runner* Ava.

Para gestionar las dependencias se ha utilizado Yarn[26]. Este gestor de dependencias surgió como alternativa a npm[27]. Al igual que todos los proyectos basados en NodeJS, el fichero más importante es el *package.json*, donde se definen las dependencias y las acciones.

Para instalar las dependencias basta con ubicarse en directorio raíz del proyecto y ejecutar el siguiente comando.

```
yarn install
```

Si se quiere continuar desarrollando el proyecto, debe utilizarse el comando:

```
yarn start
```

Este comando arrancará un servidor embebido en webpack con BrowserSync, de tal forma que se la página se recargará automáticamente cada vez que se modifican los ficheros, permitiendo ver los cambios realizados.

Si se quiere publicar la aplicación debe ejecutarse el comando:

```
yarn run webpack:prod
```

De esta forma se activa el perfil de producción de webpack, se minimiza el código y se generan los ficheros estáticos a publicar en la página web en la carpeta *dist*.

5.2. Migración del núcleo de la aplicación

5.2.1. Analizadores léxicos

El núcleo de la aplicación se basa en el uso del generador de analizadores léxicos FLEX para parsear un conjunto de instrucciones similar a las del MIPS IV. Para realizar el proceso de migración ha sido necesario comprender primero el funcionamiento de los analizadores léxicos.

Un analizador léxico es un programa que recibe como entrada el código fuente de otro programa y produce una salida compuesta de tokens o símbolos que alimentarán a un analizador sintáctico.

Para proceder con esta tarea se ha aislado la implementación original y se han realizado pequeñas pruebas concretas.

5.2.2. Lex

A pesar de las multiples librerías que hay disponibles se decidió que la más adecuada para este proyecto era Lex [28].

El funcionamiento de este paquete es realmente sencillo, partiendo de un objeto Lexer, se definen las reglas y el token a devolver. De esta forma el código original que alimentaba Flex se ha convertido en:

```
export class Parser {

    private LEX: LEX;

    private _lexer;

    constructor() {

        this._lexer = new Lexer();

        this._lexer.addRule(/^[0-9]+/i, function (lexeme) {
            this.yytext = lexeme;
            return LEX.LINESNUMBER;
        })
        // Version acortada
        .addRule(/\/\/.*/, function (lexeme) {
            return;
        });
    }

    setInput(input: string) {
```

```
        this._lexer.input = input;
    }

    lex(): Lexema {
        let value = this._lexer.lex();
        return {
            value: value,
            yytext: this._lexer.yytext
        };
    }
}
```

Como vemos, se ha hecho una integración con Typescript creando una interfaz propia como es el Lexema y un enumerado para los distintos tipos. Estos enumerados se utilizan en la integración del código y en los tests, demostrando las ventajas de utilizar esta tecnología.

5.3. Migración de la máquina superescalar

En un principio se analizó el código de la versión original de SIMDE[2] y, utlizando como referencia la memoria del mismo, donde se incluían las decisiones de diseño tomadas para la realización del simulador, se concluyó que era una buena base para la realización de este simulador.

Aún así, en la migración de C++ a Typescript se echaron en falta múltiples características del primer lenguaje, como por ejemplo: las estructuras, la sobrecarga de operadores y el uso de iteradores sobre colecciones.

La mayor dificultad en esta parte del proceso era ser capaz de seguir el flujo de un volumen tan grande de código. Por suerte, la *linterna* de Typescript y el *intellisense* ayudaron mucho a reducir la cantidad de errores sintácticos.

Aún así era notorio el hecho de que el código tenía más de una década. Al usarse C++98 muchás de las librerías no estaban incluidas en el éstandar, lo cual dificultaba entender este proceso. Por ejemplo, la siguiente porción de código dió bastantes problemas en un primer momento.

```
TStringList *etiquetas; // Lista de etiquetas
if ((ind = etiquetas->IndexOf(str + AnsiString(":"))) != -1)
   bbas = (bBasico *)etiquetas->Objects[ind];
```

Este bloque resultó especialmente confuso al castear un elemento de una lista de strings a una estructura del tipo bloque básico. Por suerte, se pudo encontrar una referencia a la clase *TStringList* en internet, explicando que cada uno de

los elementos de la lista eran de la clase *TString* la cual, tiene una propiedad que representa un vecotr de objetos asociado [29] .

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.

Se aplicaron algunos tests para la clase de *CircularBuffer*, que representaba una parte importante del funcionamiento del prefetch y el decoder y cuyo mal uso acarreó varios errores durante el desarrollo.

5.4. Desarrollo de la interfaz

Esta tarea, aunque en un principio pudiera parecer mucho menos intensa que la migración del código superescalar, también ha tenido una carga de trabajo. Todo ello porque se ha intentado por encima de todo, conseguir un alto grado de modularización y reutilización.

5.4.1. Análisis de la interfaz original

Si analizamos la interfaz original de SIMDE nos encontramos con esto:
A grosso modo podríamos diferenciar 5 *"zonas"* principales.:

- La barra de herramientas.
- La barra de accesos.
- La zona del código.
- La zona de ejecución.
- La zona de memoria / registros.

Debido a la gran cantidad de información que se debe mostrar al usuario se ha decidido que lo mejor es agrupar parte de estas zonas en pestañas. Dado que el usuario en principio querrá ver lo que es la ejecución en sí, la zona de código y de ejecución permanecerá en la primera pestaña.

De esta forma, la segunda pestaña agrupará los componentes de memoria y registros.

5.4.2. El nuevo diseño web

Tras realizar algunas pruebas, se ha mantenido un diseño similar a la versión original. En este punto se plantearon algunas cuestiones. Se utilizaría la librería

de iconos de *fontawesome* [30] y *bootstrap* [31]. Además de aplicarse la identidad visual de la Universidad de La Laguna.

La mayoría de problemas estuvieron relacionados con la distribución de las alturas y las distintas pantallas. Al ser una aplicación atípica, resultaba difícil aprovechar las distintas características de los frameworks, que se centran más en la distribución del ancho que del alto.

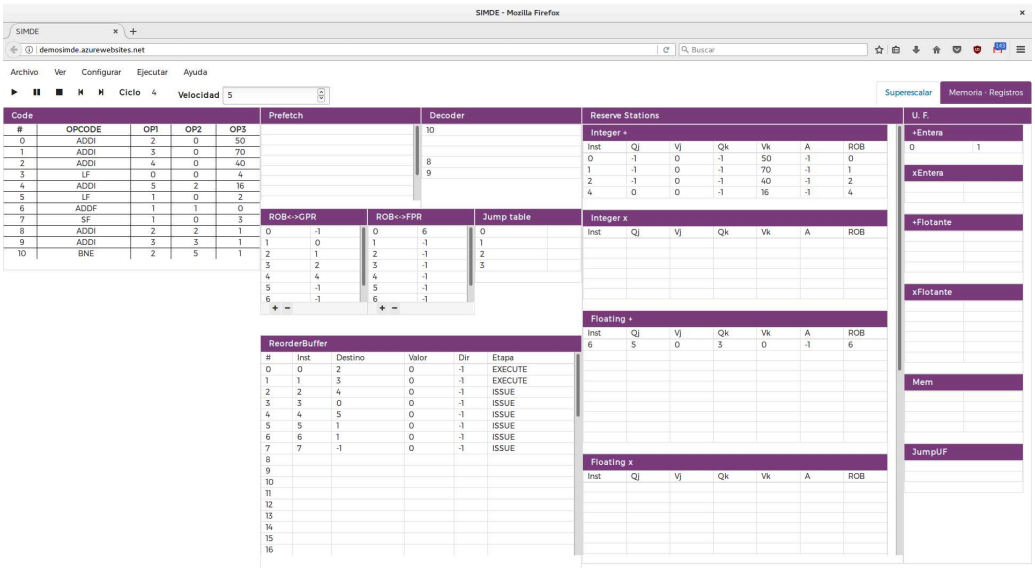

Figura 5.1: Nueva interfaz de Simde, parte uno

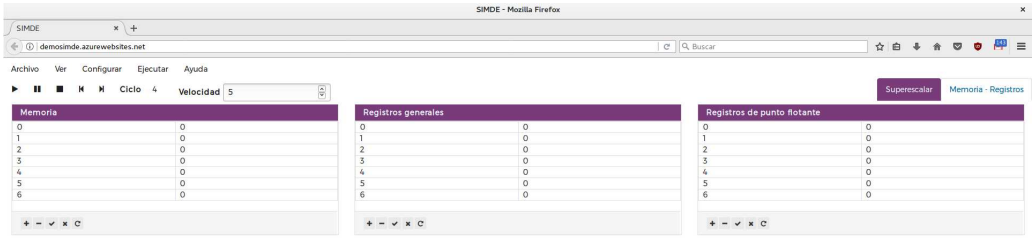

Figura 5.2: Nueva interfaz de Simde, parte dos

5.4.3. El nuevo diseño por componentes

Cada sección de la interfaz es realmente un componente independiente. Para modelar estas secciones adecuadamente se decidió importar los estilos de cada uno.

Sin embargo, esto se convirtió en un problema.Cualquier modificación de cáracter general requería repetirlo a lo largo de los distintos ficheros. Es por eso, que se decidió utilizar la tecnología Sass [32].

Esta tecnología se interpreta como CSS y añade entre múltiples características, el uso de variables. Con lo cual se pueden centralizar los estilos comunes como serían la fuente, el color.

Las características de React, permitieron hacer de esta una tarea mucho más sencilla de lo esperado en un principio.

5.5. Integración interfaz - lógica desarrollada

Una de las mayores ventajas que aporta la flexibilidad de React es que la única limitación real para el correcto funcionamiento de la librería consiste en que el código javascript en sí no debe realizar modificaciones sobre los elementos propios de los componentes de React.

Para realizar la integración entre la lógica y la interfaz se ha recurrido al uso de los callbacks. La idea es utilizar un sistema de gestión de estados que cuelgue del objeto del navegador *window*. Cuando un componente se renderiza, se suscribe a un objeto general que concentra todos los componentes y los pasos o "tics"se invocan mediante el uso de los callbacks.

Dado que todos los componentes creados para la interfaz hacían uso de estos callback se ha aplicado la herencia. Todos los componentes extienden la clase *BaseComponent* en vez de la clase *React.Component*.

Además, utilizando el mismo *título* que tenga ese componente, se decide mediante un simple switch cual es el contenido que lleva asociado. Esta abstracción permitirá extender este modelo en un futuro para la máquina VLIW.

5.6. Migración de la documentación

La primer ampliación ha sido la incorporación de documentación del programa. Aunque el término de ampliación no es del todo correcto, puesto que en el proyecto original el autor elaboró una extensa documentación, esta quedó inaccesible.

La documentación fue realizada en formato .HLP, un formato de ayuda de Windows que quedó en desuso en Windows Vista. Y esta documentación era realmente interesante, pues no sólo contenía datos sobre la aplicación, sino que incluía consejos para el desarrollo de aplicaciones para las distintas máquinas y además explicaba el funcionamiento de las máquinas.

Para recuperar esta documentación se ha utilizado una herramientas de extracción denominada Help Decompiler [33]. Esta herramienta de línea de comandos procesa los ficheros de ayuda de Windows .HLP y genera un fichero de texto

enriquecido con la documentación y en una carpeta externa el contenido multimedia que incluye la misma.

Para poder llevar a cabo la tarea de la documentación de forma paralela se consideró que lo mejor era hacer un proyecto aparte. Resultaba evidente que la documentación de una aplicación web debía de estar en la web.

Tras barajar algunas opciones, se optó por mover la documentación a un formato mantenible como es markdown. Y partiendo de esto se utilizó un generador de contenido estático basado en NodeJS (Hexo) para convertir este markdown en web. Se desarrollo un tema simple y personalizado para la ayuda y se añadió la capacidad de cambiar entre inglés y español.

Además, la documentación original también tenía su versión inglesa. Por desgracia el soporte de i18n no es tan óptimo como se podría desear, con lo cual se optó por migrar las dos versiones y habilitar un botón que permitiera seleccionar entre las dos versiones.

Al tratarse de un proyecto independiente, si se desea continuar con el desarrollo es necesario instalar las dependencias aparte, ya sea con *npm* o *Yarn*. El comando:

```
hexo server
```

Nos permite ver los cambios en local realizados sobre la documentación. Mientras que el comando:

```
hexo
```

Se encarga de generar los ficheros a publicar en la carpeta *public*. Es importante notar que para que las rutas se resuelvan correctamente se debe configurar los siguientes párametors del fichero *_config.yml*:

```
url: http://adrianabreu.com/SIMDE_DOCS/
root: /SIMDE_DOCS/
```

Este es el estado actual de la aplicación de la documentación.

Se puede acceder a ella desde el menú *Ayuda - Documentación* en la nueva aplicación del SIMDE.

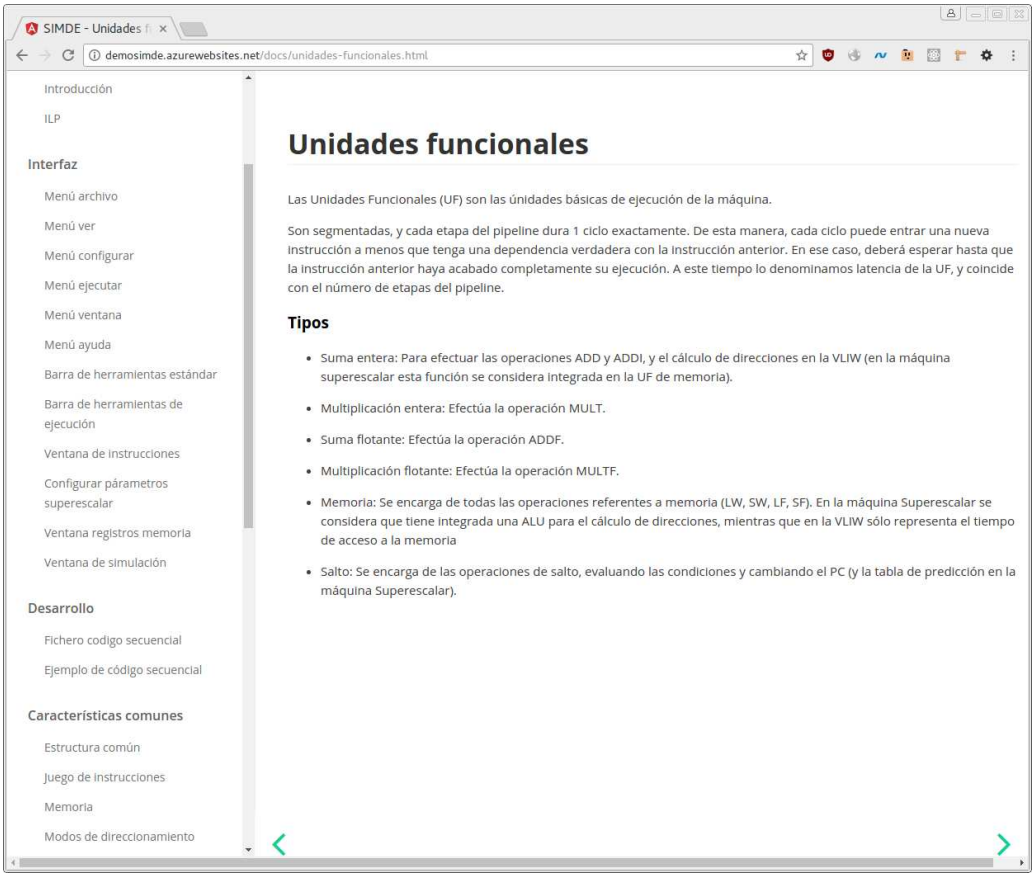

Figura 5.3: Nueva versión de la documentación

Capítulo 6

Ampliación de funcionalidades

En este proyecto de fin de grado se han realizado tres mejoras sobre las funcionalidades originales del simulador. Todas ellas han sido consecuencia de haber utilizado la aplicación original.

6.1. Parseador

Una de las características más deseadas por parte de los usuarios de SIMDE era un sistema de errores más descriptivo.

Por desgracia, en la versión original de SIMDE sólo se mostraba una notificación que indicaba que el código cargado contenía errores.

Figura 6.1: Notificación de error original SIMDE

Ahora, tras una serie de modificaciones en el parseador de código, se muestran los siguientes errores:

1. **Operando erróneo**.
2. **Opcode desconocido**.

3. **Etiqueta repetida**.

Además, se muestra la línea del error. Esto resulta tremendamente importante, ya que uno de los ejercicios que se propone en SIMDE consiste en realizar mejoras en el rendimiento de código haciendo uso de técnicas como el *desenrollado de bucles*, que dan lugar a códigos de considerable longitud.

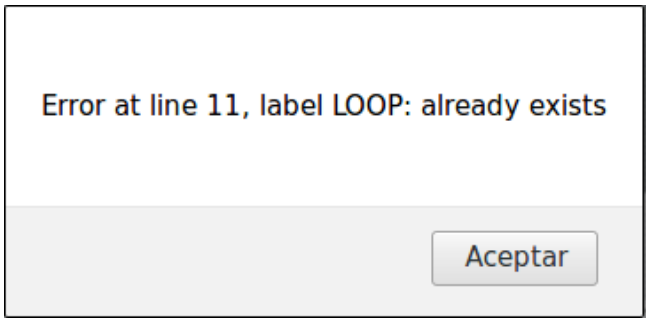

Figura 6.2: Ejemplos de errores en la nueva versión de simde

6.2. Modo de ejecución en lotes

Otra característica deseable por parte de los usuarios es no tener el límite de velocidad de simulación que tiene la versión original.

Es común proponer ejercicios a los alumnos en los que se apliquen algoritmos de ordenación. Estos ejercicios tienen una cantidad media de ciclos que pueden incluso superar 500 ciclos, requiriendo un tiempo medio de ejecución de 2 a 3 minutos.

Este tiempo sumado a la necesidad de hacer medias, a las múltiples pruebas, a la depuración de errores, y a las distintas optimizaciones, hacen que se desperdicie una considerable cantidad de tiempo de forma innecesaria en tareas secundarias, distrayendo al alumno del objetivo original.

Es por eso que ahora, cuando el campo de velocidad está en 0 se ejecuta la simulación a velocidad máxima, actualizando la interfaz gráfica solamente cuando termina la ejecución.

6.3. Histórico

Otra característica que resultaba necesaria en el simulador y que se añoraba sobre todo en el primer contacto era la posibilidad de ir hacia atrás.

En un principio, se esperaba utilizar algún sistema gestor de estados y permitir el time traveling. Por desgracia debido al volumen de este trabajo de fin de grado decidió no utilizarse este tipo de soluciones.

Sin embargo, la idea del *time traveling* resultaba más que deseable, por lo que se implementó una versión más simple de la misma pero con la misma funcionalidad. Para permitir emular este comportamiento y mantener la fidelidad de la ejecución (es importante recordar que existen una serie de operaciones que están sujetas a un porcentaje de fallos aleatorios), lo que se hizo fue acumular el estado visual de la máquina, el contenido en sí que muestran los componentes.

De esta forma cuando un usuario entra en este modo de *time traveling* lo que hace es recorrer un array de estados de la interfaz, imprimiendo la información almacenada sin afectar al comportamiento de la máquina, pero representando esa sensación de retroceder y avanzar en la ejecución de cara al usuario.

Por tanto, se considera que un usuario que retroceda X pasos, deberá avanzar esos X pasos para continuar la ejecución (o en su defecto, pulsar el botón **Play**).

Capítulo 7

Conclusiones y líneas futuras

7.1. Conclusiones

Con el desarrollo de este trabajo se ha conseguido disponer de una nueva versión del simulador de paralelismo a nivel de instrucción SIMDE [34].

Se ha visto además que las múltiples herramientas y tecnologías disponibles hoy día (los nuevos lenguajes transpilables a javascript, los web components, las herramientas para la distribución), permiten elaborar y diseñar con relativa facilidad aplicaciones que se salen de la norma.

Es necesario recordar que, como en todo proceso de software, el desarrollo de una aplicación está vivo, sujeto a cambios y dado el impresionante ritmo al que evoluciona el mundo web, es posible que en un futuro aparezcan alternativas que permitan mejorar la experiencia de este simulador.

7.2. Líneas futuras

Tras el desarrollo de este trabajo se abren varias líneas futuras:

- Implementación de la máquina VLIW: Con el desarrollo de este trabajo de fin de grado también se han implementado las estructuras básicas que se comparte con la máquina VLIW. Esta línea de trabajo tiene la mayor prioridad, pues equipara la funcionalidad de la aplicación web de SIMDE a la aplicación original.

- Realizar una mayor cantidad de test: En el mundo web no resulta sencillo realizar test para los distintos casos, sin embargo, la lógica que acompaña al simulador es un gran candidato a ser testeado. Con las bases asentadas en los tests realizados para la estructura de la cola y del parseador del código, se podría extender este funcionamiento a pequeñas simulaciones.

- Implementar un sistema de gestión de estados: En la aplicación actual se ha hecho un sistema de estados simple debido al volumen de trabajo que requiere incorporar tecnologías como Redux y a las dificultades que entrañan los Observables que vienen en un sistema como Mobx. En líneas futuras este sistema podría sustituirse por un sistema más robusto desarrollado por terceros.

- Intregar tutoriales de funcionamiento: Ahora que SIMDE es una aplicación con un gran grado de accesibilidad, la única barrera a la que se enfrentan sus usuarios es a la dificultad de comprender lo que están visualizando y el objetivo en sí de la máquina. A pesar de que esto se explica en la documentación la integración de pequeños tutoriales de funcionamiento acabaría con esta barrera inicial y fomentaría el uso a gente con un menor conocimiento específico del campo de arquitectura de computadores.

- Automatizar el sistema de ejercicios: También con el objetivo de fomentar la autonomía podría resultar interesante automatizar el sistema de ejercicios, de esta forma, mediante el uso de alguna tecnología en *backend* se podría no solo entregar al alumno un problema a resolver sino comparar la solución que ha propuesto con algunas soluciones propuestas por los profesores de antemano de tal forma que el alumno sea capaz de recibir una retroalimentación instántanea sobre su solución.

- Incluir gamificación: Mediante la incorporación de dinámicas de juego, se podría fomentar la competitividad entre los usuarios, premiando la creatividad para la resolución de problemas e instando a los alumnos a comprender mejor las arquitecturas de las máquinas y sus ventajas y limitaciones para obtener códigos que requieran menor tiempo de ejecución.

- Permitir el desarrollo colaborativo: Dado que uno de los requisitos principales de un ingeniero informático es la capacidad de trabajar en equipo, una línea de desarrollo interesante en este sentido sería sincronizar la ejecución de las máquinas entre varios miembros de un grupo mediante el uso de websockets. Así pues, si además se incluyera alguna herramienta de comunicación como un chat, se lograría fomentar una buena practica entre los alumnos y se dotaría de cierto dinamismo al desarrollo.

- Desarrollar más simuladores: Con el desarrollo de este simulador se asientan las bases para el futuro desarrollo de múltiples simuladores que permitan enseñar de forma interactiva diversos fundamentos de la arquitectura de computadores como por ejemplo la el paralelismo a nivel de hilo o la coherencia a nivel de cache en sistemas multiproceso.

Capítulo 8

Summary and Conclusions

8.1. Summary

With the development of this work we have now available a new versión of the Instruction Level Paralelism's simulator SIMDE.

It has been demonstrated that the diversity of tools and techologies available today (such as the new languages for the web, the webcomponents and the building tools), make possible to develop and design with relative easiness complex applications.

It is important to remember that as in every software development, the develop of a program is a process that is alive, being able to suffer changes, and since the exhausting rythm that the web world is following, is possible that in a close future there may be new choices that improve the user experience of this simulator.

8.2. Future work lines

After the development of this work, there are many future lines open:

- Implementation of the Very Long Instruction Word machine: Based on the initial design, the application is ready for evolve and get the same capabilities that original simulator. This line has the highest priority of all.
- Make more test: In the web world is not easy to test for the multiple cases that can be presented, but this application is really focused on its core logic. Making it a great candidate for being tested. There are a few test donde based on Ava, the futuristic test runner for typescript. Making more and more concrete test would make of SIMDE a great and robust software.
- Integrate exercises with the application: SIMDE is an educational tool.

And since it is web based, it would be possible to design an environment where the application could interact with another app, making it easier for autonomous students.

- Implement a standard state management system: Right now, the application uses a handmade tool for managing the states and allowing time travelling. There are many options available but all of them require a deep knowledge of the tool, which surpases the amount of work of this project. In my opinion, the best tool for this job would be Mobx, instead of Redux. Since it is object oriented and requires less boilerplate code.

Capítulo 9

Presupuesto

Descripción	**Coste**
200 Horas de trabajo	8000 €
Ordenador para desarrollo	1300 €
Total	9300 €

Tabla 9.1: Presupuesto

Bibliografía

[1] David Patterson and John L. Hennessy. *Computer Architecture - A Quantitative Approach*. September 2011.

[2] Iván Castilla Rodriguez. *Simulador didáctico de arquitectura de computadores*. PhD thesis, Universidad de La Laguna, 2004.

[3] SESC: SuperESCalar Simulator. http://iacoma.cs.uiuc.edu/~paulsack/sescdoc/.

[4] ReSim, a Trace-Driven, Reconfigurable ILP Processor Simulator . https://infoscience.epfl.ch/record/138644/files/resim_date09.pdf.

[5] SATSim: A Superscalar Architecture Trace Simulator Using Interactive Animation. https://www.ncsu.edu/wcae/ISCA2000/submissions/wolff.pdf.

[6] VLIW-DLX Simulator for Educational Purposes. https://www.ncsu.edu/wcae/ISCA2007/p8-becvar.pdf.

[7] Javascript Perfomance Evolution. https://en.wikipedia.org/wiki/JavaScript_engine#Performance_evolution.

[8] Electron. https://github.com/electron/electron.

[9] Coffescript. http://coffeescript.org/.

[10] Página oficial de React. https://facebook.github.io/react/.

[11] Dart. https://www.dartlang.org/.

[12] Typescript - Javascript that Scales. https://www.typescriptlang.org/.

[13] Web Components. https://www.webcomponents.org/.

[14] Polymer. https://www.polymer-project.org/.

[15] Polyfill - Wikipedia. https://en.wikipedia.org/wiki/Polyfill.

[16] Página oficial de React. https://facebook.github.io/react/.

[17] BUILDING ISOMORPHIC WEB APPLICATIONS - USING REACT AND NODE.
http://donaldwhyte.co.uk/isomorphic-react-workshop/#/4/6.

[18] Sitios que utilizan React.
https://github.com/facebook/react/wiki/sites-using-react.

[19] React's Style Guide.
https://github.com/airbnb/javascript/tree/master/react.

[20] Grunt The Javascript Runner. https://gruntjs.com/.

[21] gulpjs. http://gulpjs.com/.

[22] webpack module bundler. https://webpack.github.io/.

[23] An Introduction to Static Site Generators.
https://davidwalsh.name/introduction-static-site-generators.

[24] Hexo - A fast, simple and porwerful blog framework. https://hexo.io/.

[25] typescript-starter. https://github.com/bitjson/typescript-starter.

[26] Yarn - Fast, reliable, and secure dependency management.
https://yarnpkg.com/en/.

[27] Npm. https://www.npmjs.com/.

[28] Lex. https://www.npmjs.com/package/lex.

[29] TStrings.Objects Property. http://docs.embarcadero.com/products/rad_studio/delphiAndcpp2009/HelpUpdate2/EN/html/delphivclwin32/Classes_TStrings_Objects.html.

[30] Font Awesome, the iconic font and CSS toolkit. http://fontawesome.io/.

[31] Bootstrap · The world's most popular mobile-first and responsive front-end framework. http://getbootstrap.com/.

[32] Sass: Syntactically Awesome Style Sheets. http://sass-lang.com/.

[33] Decompile HLP Files with the WinHelp Decompiler.
https://www.helpscribble.com/decompiler.html.

[34] Simde. adrianabreu.com/SIMDE-Simulator.

[35] Web Components - React.
https://facebook.github.io/react/docs/web-components.html.

[36] William M. Johnson. Super-Scalar Processor Design. Technical report, Stanford University, 1989.

[37] Documentacion - SIMDE. http://adrianabreu.com/SIMDE-Docs/.