



# Trabajo de Fin de Grado

---

## Simulador didáctico de arquitectura de computadores

*Didactic simulator for Computer Architecture .*

Adrián Abreu González

---

La Laguna, 31 de mayo de 2017

D. **Iván Castilla Rodríguez**, con N.I.F. 78.565.451-G profesor Titular de Universidad 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 31 de mayo de 2017

# Agradecimientos

# Licencia



© Esta obra está bajo una licencia de Creative Commons Reconocimiento 4.0 Internacional.

## Resumen

*El objetivo de este trabajo ha sido desarrollar un simulador que sirva de ayuda a la docencia de arquitectura de computadores, en concreto, al funcionamiento de las máquinas superescalares.*

*A pesar de contar ya con una herramienta para tal fin, como es el simulador desarrollado por el profesor Ivan Castilla SIMDE. Esta herramienta ha quedado obsoleta, con lo cual se ha reescrito el mismo como una aplicación web.*

*Para poder llevar a cabo este rediseño se han utilizado tecnologías modernas tales como Typescript, React, Webpack y Sass, permitiendo aprovechar todas las mejoras que se han producido en el mundo de las aplicaciones web en los últimos años.*

**Palabras clave:** SIMDE, React, Superescalar, Typescript, Webpack, Arquitectura de Computadores.

## Abstract

The purpose of this work was to develop a simulator for helping in the teaching of Computer Architecture's fundamentals, principally, in how superescalar machines work.

Despite there exists a tool with that purpose like Ivan Castillas's SIMDE. This tool is now out-dated. Using it as the basis of this project, a new version of this tool has been made from scratch as a web application.

In order to rewrite and older application, this projects uses the latests technologies in the web world like Typescript, React, Webpack and Sass, allowing to make use of the last improvements in the world of the web applications.

**Keywords:** *SIMDE, React, Superescalar, Typescript, Webpack, Computer Architecture.*

# Índice general

<b>1. Introducción</b>	<b>1</b>
1.1. Introducción . . . . .	1
1.2. Paralelismo a nivel de instrucción . . . . .	1
1.2.1. VLIW . . . . .	2
1.3. Motivación para el trabajo . . . . .	3
<b>2. Antecedentes</b>	<b>4</b>
2.1. SIMD . . . . .	4
2.2. Otros simuladores . . . . .	5
2.3. Evolución del mundo web . . . . .	5
<b>3. Objetivos y fases</b>	<b>7</b>
3.1. Objetivos . . . . .	7
3.2. Fases . . . . .	7
<b>4. Tecnologías</b>	<b>9</b>
4.1. Lenguaje para la lógica de la aplicación . . . . .	9
4.1.1. Coffescript . . . . .	9
4.1.2. Dart . . . . .	10
4.1.3. Typescript . . . . .	10
4.2. Tecnología para la integración modelo vista . . . . .	11
4.2.1. Webcomponents . . . . .	11
4.2.2. Polymer . . . . .	12
4.2.3. React . . . . .	12
4.3. Tecnología para hacer la build . . . . .	14
4.3.1. Gulp/Grunt . . . . .	14
4.3.2. Webpack . . . . .	14
4.4. Tecnología para la documentación . . . . .	15
4.4.1. Generadores de contenido estático . . . . .	16
4.4.2. Hexo . . . . .	16
<b>5. Desarrollo del proyecto</b>	<b>17</b>
5.1. Proyecto base . . . . .	17
5.2. Migración del núcleo de la aplicación . . . . .	18

5.2.1. Analizadores léxicos . . . . .	18
5.2.2. Lex . . . . .	18
5.3. Migración de la máquina superescalar . . . . .	19
5.4. Desarrollo de la interfaz . . . . .	20
5.4.1. Análisis de la interfaz original . . . . .	20
5.4.2. El nuevo diseño web . . . . .	20
5.4.3. El nuevo diseño por componentes . . . . .	21
5.5. Integración interfaz - lógica desarrollada . . . . .	22
5.5.1. Realización de los bindings . . . . .	22
5.6. Migración de la documentación . . . . .	22
<b>6. Ampliación de funcionalidades</b>	<b>24</b>
6.1. Parseador . . . . .	24
6.2. Modo de ejecución en lotes . . . . .	25
6.3. Histórico . . . . .	25
<b>7. Conclusiones y líneas futuras</b>	<b>27</b>
7.1. Conclusiones . . . . .	27
7.2. Líneas futuras . . . . .	27
<b>8. Summary and Conclusions</b>	<b>29</b>
8.1. Summary . . . . .	29
8.2. Future work lines . . . . .	29
<b>9. Presupuesto</b>	<b>31</b>
<b>Bibliografía</b>	<b>31</b>



# Índice de figuras

2.1. Simulador original SIMDE . . . . .	4
4.1. Typescript como superconjunto de Javascript . . . . .	11
4.2. Funcionamiento del DOM virtual de React [1] . . . . .	13
4.3. Imagen descriptiva de Webpack . . . . .	15
5.1. Funcionamiento del DOM virtual de React [1] . . . . .	21
5.2. Nueva version de la documentacion . . . . .	23
6.1. Notificacion de error original SIMDE . . . . .	24
6.2. Ejemplos de errores en la nueva versión de simde . . . . .	25

# Índice de tablas

2.1. Tabla comparativa de simuladores . . . . .	5
9.1. Presupuesto . . . . .	31

# 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. Sin embargo, 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 instrucción implementando la técnica conocida como segmentación -una técnica que promueve una disposición concreta de los recursos de la máquina y una división 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.[2]

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 resueltas 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.

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 originalmente 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.

### 1.2.1. VLIW

A diferencia de las máquinas Superescalares, las máquinas *Very Long Instruction Word*, mantienen un hardware simple de emisión y todas las responsabilidades caen en el compilador.

Para mantener este hardware simple, estas máquinas trabajan como su propio nombre indica, con un tamaño de palabra muy grande, es decir, las instrucciones de la máquina son un "grupo" de instrucciones normales que pueden ejecutarse en paralelo.

Esto por supuesto, implica una serie de ventaja / inconvenientes. Por una parte, el compilador, debe tener una visión completa del programa, conlleva un nivel de complejidad elevado y requiere un alto grado de especialización por máquina para poder realizar las mayores optimizaciones posibles.

Por otra parte, sin embargo, el software es versátil y la circuitería al ser de una menor complejidad permite mayores velocidades de reloj.

### **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. [3]

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.

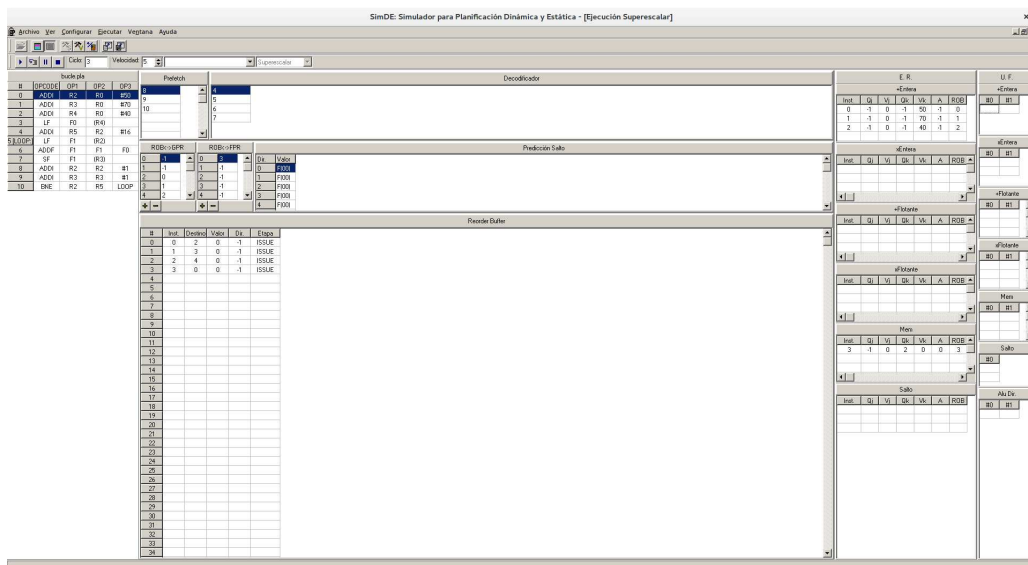


Figura 2.1: Simulador original SIMDE

## 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 [4]	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 [5]	Este simulador de máquinas Superescalares está basado en la ejecución por trazas. Se encuentra disponible para múltiples dispositivos Xilinx.
SATSim [6]	Simulador de máquinas superescalares con traja que utiliza una simulación interactiva.
VLIW-DLX [7]	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.

[3]

Por sorprendete que resulte, 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.[8]

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*. [9]



# 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 dividido 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 el 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*. [10]

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*. [11]

Coffescript podría ser la opción ideal para agilizar el 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*. [12]

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, ha pesar de lo atractivo del lenguaje, Dart 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. [13]

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.

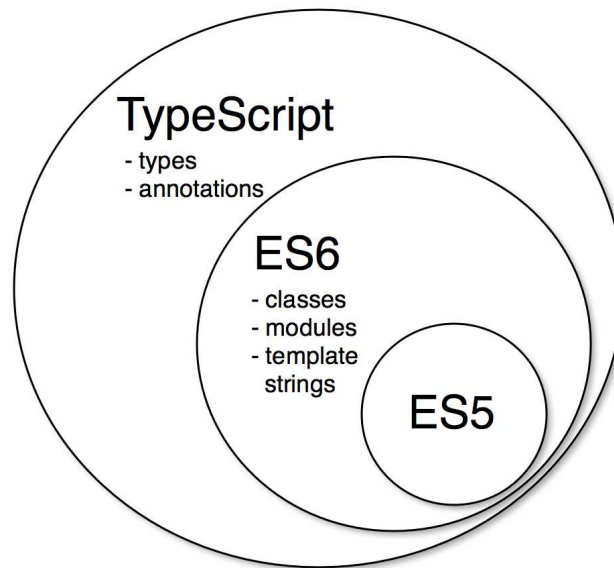


Figura 4.1: Typescript como superconjunto de Javascript

2. Typescript está alineada en cierta forma con el futuro de Javascript. Microsoft es uno de los muchos que forman parte del consenso de estándar de ECMAScript.
3. Typescript no me limita en la posibilidad de usar javascript, todo código javascript es código Typescript válido.
4. Por último y no menos importante: Tengo cierta experiencia con Typescript.

## 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 característica 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. [14]

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 [15] 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*). [16]

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”. [17]

Pero realmente aunque esta declaración es válida, 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 estándar de HTML5, sino como pequeños bloques reusables que incorporan cierta funcionalidad. Sin embargo,

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

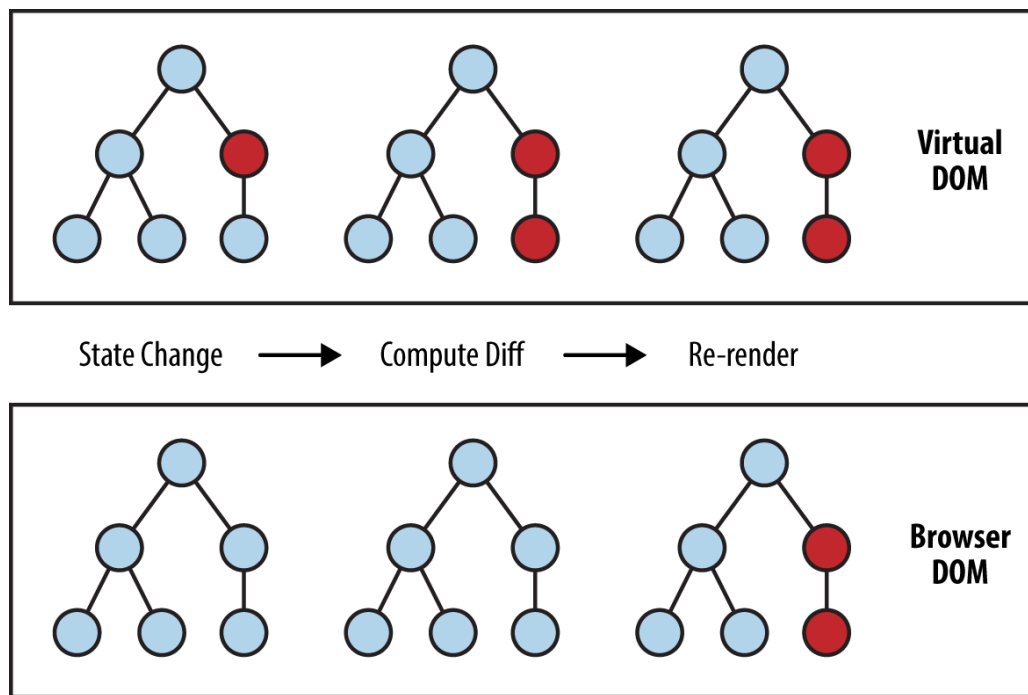


Figura 4.2: Funcionamiento del DOM virtual de React [1]

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, Walmart... [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 hacer la build

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 aplicación. 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 muchísimos 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 posee de forma intrínseca 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.

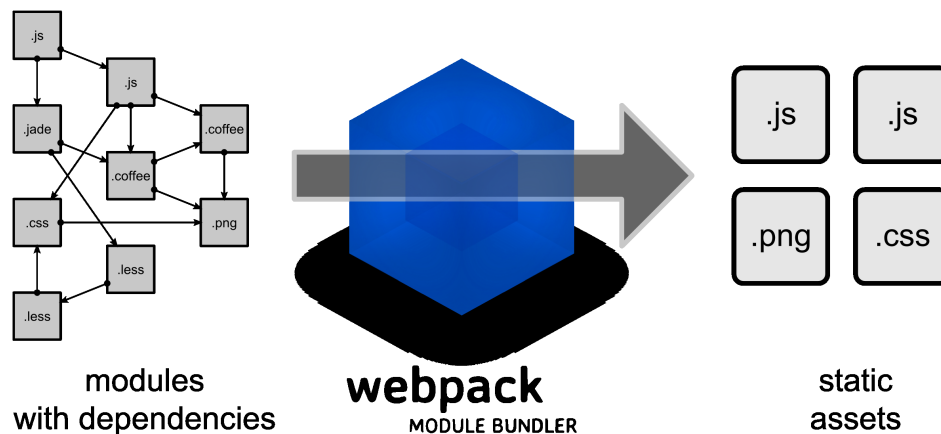


Figura 4.3: Imagen descriptiva de Webpack

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, formato legible . [23]

Este tipo de generadores estáticos tienen un gran auge entre los desarrolladores que desean mantener un blog -yo mismo por ejemplo, tengo uno hecho en *Hugo*-.

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 por dos motivos principales:

1. No me resulta desconocido, ya que lo he utilizado durante cierto tiempo.
2. Al estar basado en Javascript todo queda enfocado hacia un mismo ecosistema dando una sensación más uniforme respecto al 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 proyecto *typescript-starter*.<sup>[25]</sup> Este proyecto incluía las dependencias necesarias 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<sup>[26]</sup>. Este gestor de dependencias surgió como alternativa a npm<sup>[?]</sup>. 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 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 he tenido que 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 múltiples librerías, que hay disponibles, se decidió que la más adecuada para este proyecto era Lex[27].

El funcionamiento de este paquete es realmente sencillo, partiendo de un objeto Lexer, se definen las reglas y el token a retornar. 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.ytext = 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 SIMD[3] y, utilizando 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 pifias.

Aún así era notorio el hecho de que el código tenía más de una década. Al usarse C++98 muchas de las librerías no estaban incluidas en el estándar, 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 de strings tenía un vector de objetos asociado.

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* y *bootstrap*. 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.

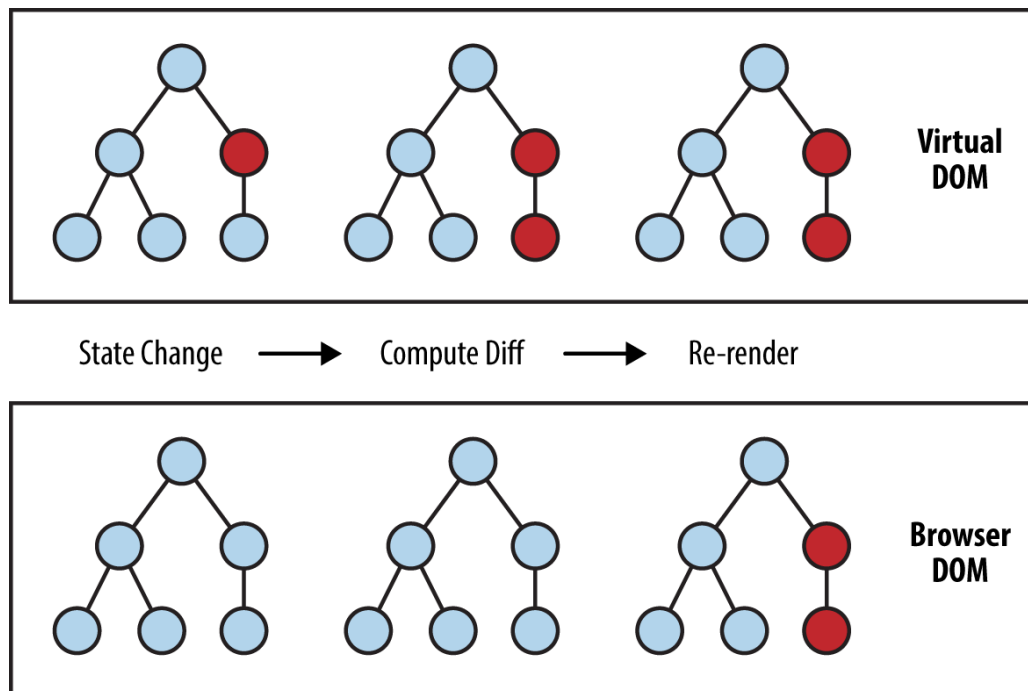


Figura 5.1: Funcionamiento del DOM virtual de React [1]

### 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 carácter general requería repetirlo a lo largo de los distintos ficheros. Es por eso, que se decidió utilizar la tecnología Sass [?].

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

### 5.5.1. Realización de los bindings

Uno de los puntos a favor de haber utilizado este tipo de librerías es que su elevada flexibilidad. La única limitación es 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*.

## 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 herramienta de extracción denominada Help Decompiler. 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 mark-



down en web. Se desarrollo un tema simple y personalizado para la ayuda y se añadió la capacidad de cambiar entre inglés y español.

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

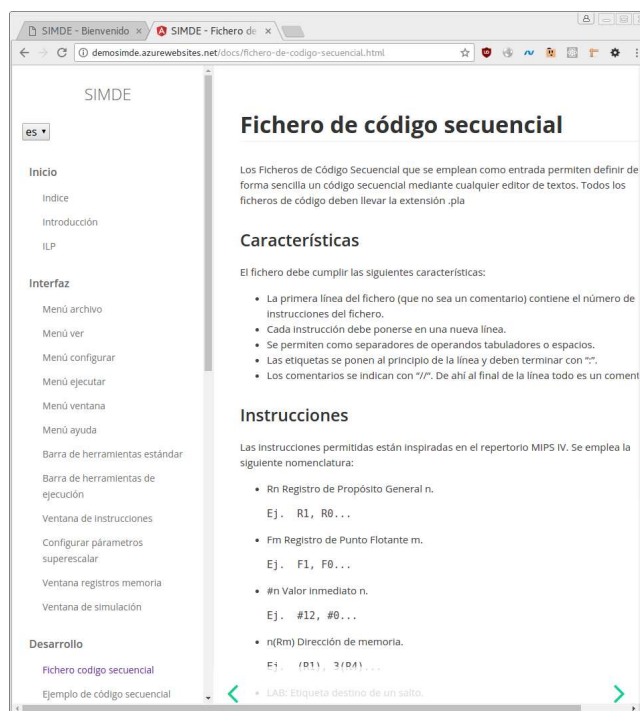


Figura 5.2: Nueva version de la documentacion

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

# 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 (entre los que yo mismo me puedo incluir), es 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.

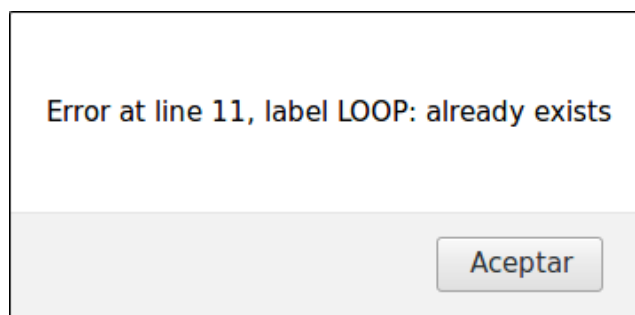


Figura 6.2: Ejemplos de errores en la nueva versión de simde

## 6.2. Modo de ejecución en lotes

Otra característica que habría sido resultado muy deseable por mi parte y por parte de mis compañeros, es no tener el límite de velocidad que tiene la versión original.

En mi caso, la cantidad media de ciclos de los ejercicios que se me propusieron superaba los 500 ciclos, haciendo un tiempo medio de ejecución de 2 - 3 minutos, lo cual sumado a la necesidad de hacer medias, a las pruebas, a la depuración de errores, y a las distintas optimizaciones, se perdía tiempo de forma innecesaria el desarrollo de los ejercicios.

Es por eso que ahora, cuando el campo de velocidad está en 0, se ejecuta la simulación a velocidad máxima, y solo se refresca la interfaz cuando termina la ejecución. Ahorrando así recursos.

## 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 de una forma un tanto rústica. Para permitir emular este comportamiento y mantener la fidelidad de la ejecución (recordemos que existen una serie de operaciones que son sujetas a un porcentaje de fallos aleatorios), lo que se hizo fue acumular el estado visual de la máquina, el modelo en sí con el que se dibujaban los componentes.

De tal forma, cuando un usuario entraba en este modo timetraveling recorre un array de estados de la interfaz, imprimiendo la información almacenada, sin afectar al comportamiento de la máquina, pero emulando ese *time traveling* 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**). Esto es así porque aunque como repito, se trata de una característica enormemente deseada en la primera toma de contacto del simulador, no se trata de una característica de la que se espere que el usuario abuse.

# Capítulo 7

## Conclusiones y líneas futuras

### 7.1. Conclusiones

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

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 recordad, que como en todo proceso de software, el desarrollo de una aplicación esta vivo, sujeto a cambios y que dado el impresionante ritmo al que evoluciona el mundo de la web, quizás este trabajo sea un pequeño anexo al pie de página de lo que esta aplicación pueda representar.

### 7.2. Líneas futuras

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

- Implementación de la máquina VLIW: Esta línea no resulta sorprendente. 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 SIMD 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 testado. 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.

- Realizar un tutorial de inicio: Ahora que SIMDE es una aplicación accesible para todo el mundo, sería deseable que no supusiera una barrera para cualquier usuario que sintiera la necesidad de probar
- Implementar un sistema de gestión de estados: En la aplicación actual se ha hecho un sistema de estados rudimentario debido al volumen de trabajo de sistemas como Redux y a las dificultades que entrañan los Observables que vienen en un sistema como Mobx. En líneas futuras este sistema rudimentario podría sustituirse por un sistema más robusto desarrollado por terceros.

# 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 technologies available today (such as the new languages for the web, the webcomponents and the building tools), make possible to develop and design with realitve 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, maybe this work will be no more that a little footn note at what it may be able to become.

### 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: This can not be a surprise. 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 management the states and allow time travelling. There are many options available but all of them requires a deep knowledge of the tool, which stays out of 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] BUILDING ISOMORPHIC WEB APPLICATIONS - USING REACT AND NODE. <http://donaldwhyte.co.uk/isomorphic-react-workshop/#/4/6>.
- [2] David Patterson and John L. Hennessy. *Computer Architecture - A Quantitative Approach*. September 2011.
- [3] Iván Castilla Rodriguez. *Simulador didáctico de arquitectura de computadores*. PhD thesis, Universidad de La Laguna, 2004.
- [4] SESC: SuperESCalar Simulator. <http://iacoma.cs.uiuc.edu/~paulsack/sescdo>
- [5] ReSim, a Trace-Driven, Reconfigurable ILP Processor Simulator . [https://infoscience.epfl.ch/record/138644/files/resim\\_date09.pdf](https://infoscience.epfl.ch/record/138644/files/resim_date09.pdf).
- [6] SATSim: A Superscalar Architecture Trace Simulator Using Interactive Animation. <https://www.ncsu.edu/wcae/ISCA2000/submissions/wolff.pdf>.
- [7] VLIW-DLX Simulator for Educational Purposes. <https://www.ncsu.edu/wcae/ISCA2007/p8-becvar.pdf>.
- [8] Javascript Perfomance Evolution. [https://en.wikipedia.org/wiki/JavaScript\\_](https://en.wikipedia.org/wiki/JavaScript_)
- [9] Electron. <https://github.com/electron/electron>.
- [10] Coffescript. <http://coffeescript.org/>.
- [11] Página oficial de React. <https://facebook.github.io/react/>.
- [12] Dart. <https://www.dartlang.org/>.
- [13] Typescript - Javascript that Scales. <https://www.typescriptlang.org/>.
- [14] Web Components. <https://www.webcomponents.org/>.
- [15] Polymer. <https://www.polymer-project.org/>.
- [16] Polyfill - Wikipedia. <https://en.wikipedia.org/wiki/Polyfill>.

- [17] Página oficial de React. <https://facebook.github.io/react/>.
- [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-style-guide>.
- [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 powerful 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] Lex. <https://www.npmjs.com/package/lex>.
- [28] Web Components - React. <https://facebook.github.io/react/docs/web-components.html>.
- [29] William M. Johnson. Super-Scalar Processor Design. Technical report, Stanford University, 1989.
- [30] Documentacion - SIMD. [http://adrianabreu.com/SIMD\\_DOCS/](http://adrianabreu.com/SIMD_DOCS/).