Trabajo de Fin de Grado

Simulador didáctico de arquitectura de computadores

Didactic simulator for Computer Architecture .

Adrián Abreu González

La Laguna, 22 de mayo de 2017

D. **Iván Castilla Rodríguez**, con N.I.F. 78.565.451-G profesor Titular de Universidad adscrito al Departamento de Nombre del Departamento 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 22 de mayo de 2017

Agradecimientos

XXX
XXX
XXX
XXX

Licencia

Resumen

El objetivo de este trabajo ha sido .... bla, bla, bla bla, bla, bla bla, bla, bla

La competencia [E6], que figura en la guía docente, indica que en la memoria del trabajo se ha de incluir: antecedentes, problemática o estado del arte, objetivos, fases y desarrollo del proyecto, conclusiones, y líneas futuras.

Se ha incluido el apartado de 'Licencia' con todas las posibles licencias abiertas (Creative Commons). En el caso en que se decida hacer público el contenido de la memoria, habrá que elegir una de ellas (y borrar las demás). La decisión de hacer pública o no la memoria se indica en el momento de subir la memoria a la Sede Electrónica de la ULL, paso necesario en el proceso de presentación del TFG.

El documento de memoria debe tener un máximo de 50 páginas.

No se deben dejar páginas en blanco al comenzar un capítulo, ya que el documento no está pensado para se impreso sino visionado con un lector de PDFs.

También es recomendable márgenes pequeños ya que, al firmar digitalmente por la Sede, se coloca un marco alrededor del texto original.

El tipo de letra base ha de ser de 14ptos.

Palabras clave: Palabra reservada1, Palabra reservada2, ...

Abstract

Here should be the abstract in a foreing language...

Keywords: *Keyword1, Keyword2, Keyword3, ...*

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Introducción

En el área de Arquitectura de Computadores, es común trabajar sobre fundamentos teóricos que tienen una base histórica y a pesar de todo el tiempo que ha pasado desde que aparecieron estos conceptos y de la abundante bibliografía de la que se dispone, resultan conceptos difíciles de asimilar debido en parte, al desfase que existe con las implementaciones de los sistemas modernos. Es decir, estos fundamenos conllevan una gran abstracción para el alumno al no disponer de las máquinas originales en las que se implementaron.

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.

Sin embargo, a pesar de que han pasado ya más de cuarentas años de los diseños iniciales de esta técnica, aún no resulta sencillo explicar en detalle el funcionamiento de la misma.

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

Con el objetivo de mejorar el rendimiento de una computadora es necesario superar la barrera de 1 ciclo por instrucción -el límite máximo que se puede conseguir con la técnica de segmentación-, para ello, se debe conseguir ejecutar varias instrucciones diferentes de forma paralela.

Para alcanzar este propósito se deben considerar varios factores:

- Se debe proveer de una estructura capaz de emitir múltiples instrucciones.

- Se deben detectar los posibles riesgos asociados (tanto entre las instrucciones que se van a emitir entre sí como entre las que se van a emitir y las que ya están ejecutándose).
- Se debe hacer una planificación para evitar que sucedan los posibles riesgos de datos o de estructuras, y también optimizar el funcionando de la máquina.

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

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. Superescalar

La mayoría de máquinas Superescalares hacen uso de la planificación dinámica, es decir, el hardware en sí mismo es el encargado de mantener la emisión de múltiples instrucciones, decodificarlas y ejecutarlas.

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 dependencias *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.

1.2.2. 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 optimizacions 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.

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.

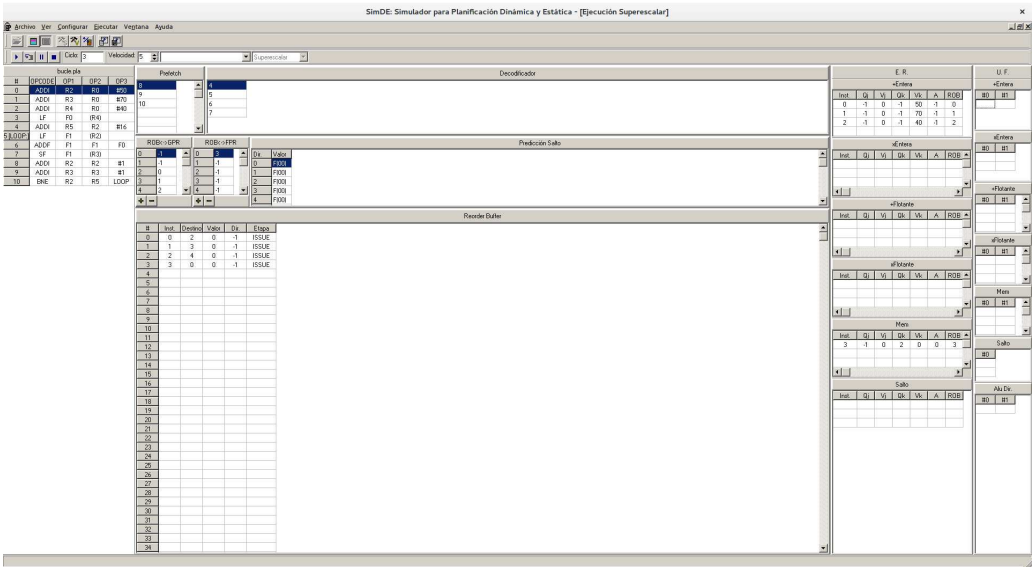

Figura 2.1: Simulador original SIMDE

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	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	Este simulador de máquinas Superescalares está basado en la ejecución por trazas. Se encuentra disponible para múltiples dispositivos Xilink.
VLIW-DLX	Basado en WinDLX y desarrollado en la Universidad de praga, intenta aportar un conocimiento detallado de las máquinas VLIW.

Como vemos, aunque existen múltiples simuladores dentro de esta sección, ninguno cumple con el objetivo inicial, es más, lamentablemente, la mayoría de los simuladores tienen más de una década y NINGUNO de los mencionados se encuentra disponible en versión web.

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.

Resulta difícil plantear la validez de este argumento en un contexto actual. Hoy en día, el rendimiento del lenguaje Javascript en el navegador es increíblemente alto. Y si vemos que la diferencia de este texto a ahora, es de unos escasos trece años -aunque ciertamenta nada despreciables en el mundo tecnológico, la diferencia resulta asombrosa.

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

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**: Realizar 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*.

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*.

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*.

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 por una razón de peso y es que se trata de un lenguaje totalmente diferente y su uso es mayoritariamente por parte de Google.

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.

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 ganadora, 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.

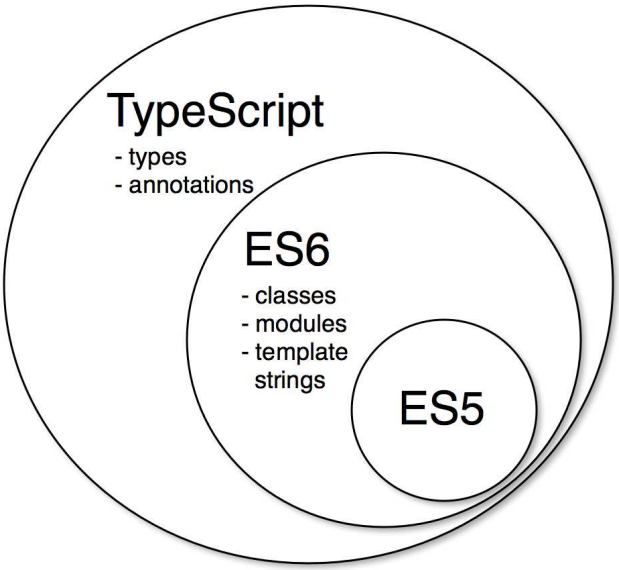

Figura 4.1: Typescript como superconjunto de Javascript

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 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 realizar por mí 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

Lo que la nueva versión de SIMDE necesitaba era aprovechar las características que ofrecen los Web Components. Los Web Components son un conjunto de caracteŕisticas que se están añadiendo a las especificaciones W3C de Html y del DOM.

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 subarbol 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 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).*

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”.

Pero realmente aunque esta declaración es válida, esto no es tan sencillo, React intenta resolver el problema del refresco del DOM de forma novedosa:

Simulandolo completamente y solo modificando el DOM *"verdadero"* cuando sea necesario.

React utiliza un híbrido entre html y javascript denominado jsx, como también tiene soporte para Typescript, en este caso utilizamos tsx, y se basa en un "unidirectonial data-flow".

Los componentes de React tienen un método render, que devuelve una estructura de DOM virtual, la cual, mediante cambios de estado se muestra en el DOM real. Y mediante la aplicación de teoría de grafos y uso de árboles, solo se realizarán las manipulaciones mínimas del DOM.

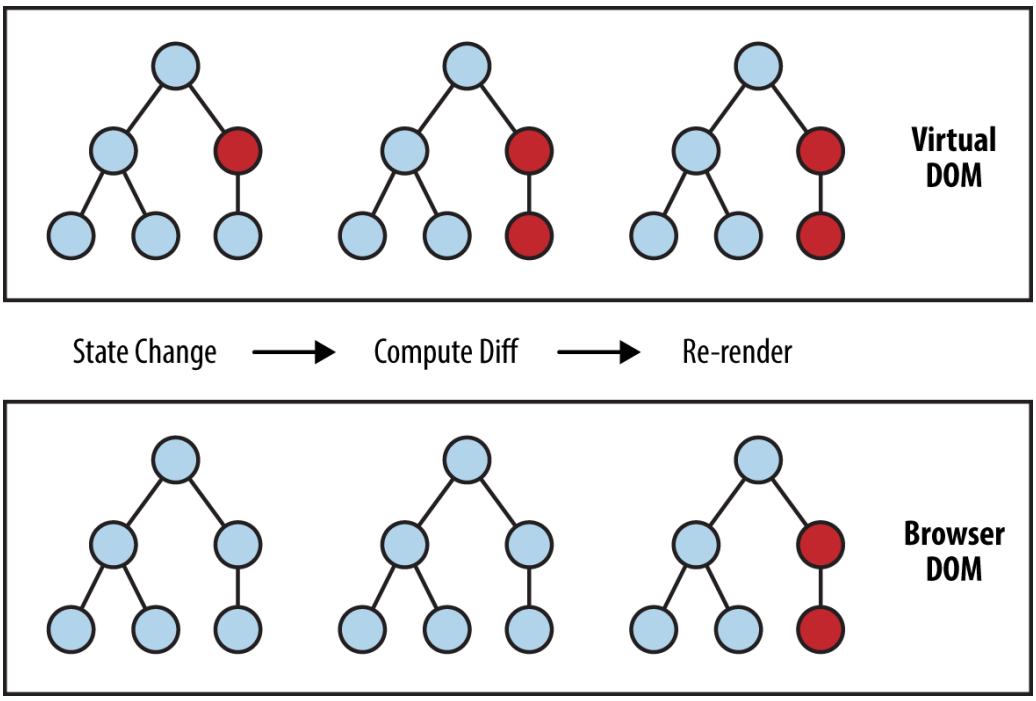

Figura 4.2: Funcionamiento del DOM virtual de React

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...

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

Además,la comunidad de usuarios es increíblemente activa, por ejemplo el mayor representante es Dan Abramov, 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 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 **Gulp** y **Grunt**.

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.

Webpack posee de forma intrínseca diversas características interensantes tales como el poder ejecutar un servidor de desarrollo que aplica *hot reloading* sobr 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.

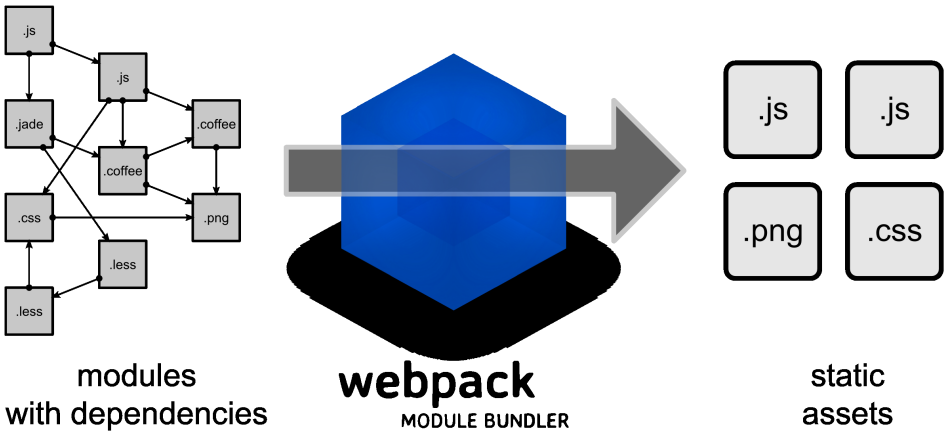

Figura 4.3: Imagen descriptiva de Webpack

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 tosca y breve- de generar un conjunto de htmls y css a partir de una plantilla y una serie de ficheros fuentes.

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, pero sin duda para mi la más importante, es que se alimentan de un formato como es el markdown. El cual es muy intuitivo de usar y tiene soporte más allá de este tipo de tecnologías.

4.4.2. Hexo

Hexo es un generador de contenido estático basado en NodeJS. 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. Migración del núcleo de la aplicación

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

5.1.2. Lex

A pesar de las multiples librerías, que hay disponibles, se decidió que la más adecuada para este proyecto era Lex https://www.npmjs.com/package/lex.

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:

```
H                       [A-Fa-f0-9]
D                       [0-9]
E                       [Ee][+-]?{D}+
id                      [A-Za-z_][A-Za-z0-9_]*
espacio         [ \t\v\f]
direccion       [+-]?{D}*"("[Rr]{D}+")"

%%
```

```
^{D}+                   {       /* Esto es el numero de lineas d
                                return LEXNLINEAS;
                        }

#[+-]?{D}+              { return LEXINMEDIATO; }
[Ff]{D}+                { return LEXREGFP; }
[Rr]{D}+                { return LEXREGGP; }
{id}                    { return LEXID; }
{id}":"                 { return LEXETIQUETA; }
{direccion}             { return LEXDIRECCION; }

"//".*                  { /* Comentario */ }
{espacio}+              { /* Espacio en blanco */ }
(.|\n)                  { /* Cosas extranas y retornos de carro

%%

int yywrap() { return 1; }

int setYyin(char *nombre) {
        FILE *aux = fopen(nombre, "r");
        if (aux == NULL)
                return -1;
        yyrestart(aux);
        return 0;
}

int unsetYyin() {
        int res = fclose(yyin);
        yyin = NULL;
        return res;
}

char *getYytext() {
        return yytext;
}
```

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;
        }).addRule(/[Ff][0-9]+/i, function (lexeme) {
            this.yytext = lexeme;
            return LEX.REGFP;
        }).addRule(/[Rr][0-9]+/i, function (lexeme) {
            this.yytext = lexeme;
            return LEX.REGGP;
        }).addRule(/#[+-]?[0-9]+/i, function (lexeme) {
            this.yytext = lexeme;
            return LEX.INMEDIATE;
        }).addRule(/[A-Za-z][A-Za-z0-9]*\:/i, function (lexeme) {
            this.yytext = lexeme;
            return LEX.LABEL;
        }).addRule(/[A-Za-z][A-Za-z0-9]*/i, function (lexeme) {
            this.yytext = lexeme;
            return LEX.ID;
        }).addRule(/[+-]?[0-9]*\([Rr][0-9]+\)/i, function (lexeme)
            this.yytext = lexeme;
            return LEX.ADDRESS;
        }).addRule(/^[0-9]+/i, function (lexeme) {
            return;
        }).addRule(/[ \t\v\f]+/i, function (lexeme) {
            return;
        }).addRule(/(.|\n)/i, function (lexeme) {
            return;
        }).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
        };
    }
}
```

5.2. Migración de la máquina superescalar

Una de las cosas que más puedo agradecer, es que el diseño del autor original de SIMDE era bastante bueno. Además, en la memoria del proyecto **CITAR MEMORIA** se incluían las decisiones de diseño tomadas para la realización del simulador. Dando sentido.

Aún así, en la migración de C++ a Typescript se echaron 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 encontré además, algunos problemas a la hora de migrar el código debido al uso de librerías que no estaban en el estándar.
CITAR FAMOSO CASO BBAS.

Por comodidad durante el desarrollo, se convirtieron las estructuras de C++ en clases, de tal forma que la gestión de las mismas fuero más sencilla.

5.3. 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.

5.3.1. 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.

Se ha considerado que lo más sensato es agrupar las ventanas
Ahora analizaremos el funcionamiento de los componentes, en sí, con el objetivo de aislar conceptos /funcionalidades.

5.3.2. El nuevo diseño web

A día de hoy

Ahora surge un problema, ¿merece el esfuerzo aplicar el diseño de las ventanas redimensionables y arrastables? La conclusión es que no. SIMDE muestra la información que necesitamos. En un principio nos interesa ver la ejecución en sí. Luego, en su realización, nos interesa ver simplemente el resultado de la ejecución.

Por eso, se ha decidido separar la ejecución de los registros / memoria mediante el uso de pestañas.
a) Pestaña 1.
b) Pestaña 2.

5.3.3. El nuevo diseño por componentes

5.4. Integración interfaz - lógica desarrollada

5.4.1. Realización de los bindings

Uno de los puntos a favor de haber utilizado este tipo de librerías es que su elevada flexibilidad. Mientras que el código javascript en sí no altere los elementos propios de los componentes de React, no hay problema.

Para realizar la integración entre la lógica y la interfaz se ha recurrido al uso de los callbacks. 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.

5.5. 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 palicación, 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. 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.

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

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: Notificacion 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.

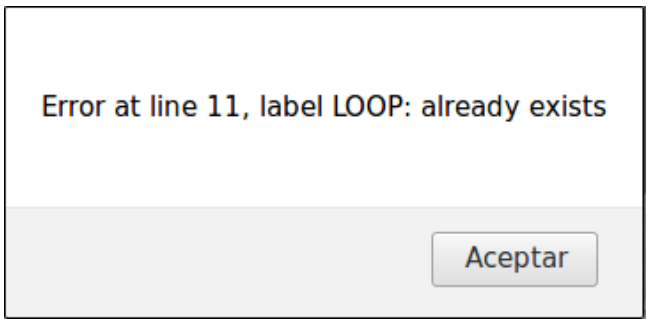

Figura 6.2: Ejemplos de errores en la nueva versión de simde

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 limite 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 media de pruebas, a la depuración de errores, y a las distintas optimizaciones, alargaba 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 SIMDE.

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 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.

- 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

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

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.

8.2. Future work lines

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 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.

- 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 9

Presupuesto

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

Tabla 9.1: Presupuesto

Apéndice A

Título del Apéndice 1

A.1. Algoritmo XXX

```
*****************************************************************************
*
* Fichero .h
*
*****************************************************************************
*
* AUTORES
*
*
* FECHA
*
*
* DESCRIPCION
*
*
*****************************************************************************/
```

A.2. Algoritmo YYY

```
/*****************************************************************************
 *
 * Fichero .h
 *
 *****************************************************************************
 *
 * AUTORES
 *
 * FECHA
 *
 * DESCRIPCION
 *
 *
 *****************************************************************************/
```

Bibliografía

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

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

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

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

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

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

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

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

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

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

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

[12] S. Taásan. *Multigrid Methods for Highly Oscillatory Problems*. PhD thesis, Weizmann Institute of Science, Rehovot, Israel, 1984.