Trabajo de Fin de Grado

Simulador didáctico de arquitectura de computadores

Didactic simulator for Computer Architecture .

Adrián Abreu González

La Laguna, 13 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 13 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 repasar fundamentos teóricos que tienen una base histórica y que resultan difíciles de comprender debido a que a pasar de que son las bases de los sistemas modernos, conllevan una gran abstracción respecto a las soluciones que se implementan en la actualidad.

En este trabajo, se hace hincapié en el aspecto del paralelismo a nivel de instrucción, un aparte fundamental de las computadoras modernas y que permitió una mejora exponencial en el rendimiento.

Sin embargo, a pesar de que han pasado ya más de veinte años de los diseños iniciales, este y muchos otros conceptos no resultan fáciles de enseñar en la docencia. A pesar de que todo es abstraíble a una serie de fundamentos simples, resulta muy difícil visualizar el paso de la *segmentación* al paralelismo a nivel de instrucción.

En este contexto, un simulador permite ayudar a comprender, de manera visual y esquematizada como funcionan estos mecanismos y actuar como un potente catalizador para la asimilación de los conceptos.

1.2. Paralelismo a nivel de instrucción

Con el objetivo de mejorar el rendimiento de una computadora es necesario superar la barrera de 1 CPI *(ciclo por instrucción)*, para ello, se debe aumentar el grado de emisión de instrucciones.

Para alcanzar este propósito se deben considerar varios factores:

1. Se debe proveer de una estructura de emisión.
2. Se deben detectar los posibles riesgos asociados.

3. Se debe hacer un *scheduling*.

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

1. **Planificación dinámica**
2. **Planificación estática**

1.2.1. Superescalar

Las máquinas Superescalares hacen uso de la planificación dinámica, es decir, el hardware en si es el encargado de mantener la emisión de múltiples instrucciones, decodificarlas y ejecutarlas.

Este tipo de parlaleismo conlleva un enorme grado de complejidad, ya que se deben abordar todos estos problemas:

1. Capturar y decodificar más instrucciones en un ciclo.
2. Detectar dependencias entre las instrucciones capturas y las que se ejecutan.

Para resolver los problemas de dependencias se utiliza el algoritmo de Tomasulo.
CITAR ALGORITMO TOMASULO

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 conjunto 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 Arquitecutra de Computadores resultaba un campo realmente interesante. De hecho, esta herramienta ya existe. El actual 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 más de la docencia, 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 hecha utilizando C++98 y C++ Builder y necesita un nuevo enfoque.

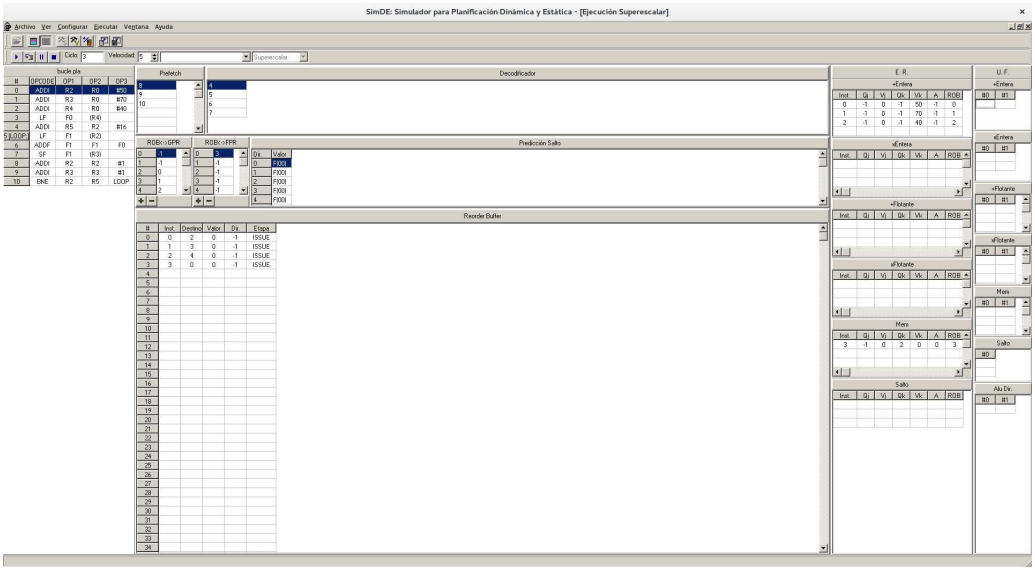

Figura 2.1: Emulador original de SIMDE

A día de hoy, una de las mejores soluciones para implementar este tipo de

simulador es la web. Permitiendo la accesibilidad por un mayor número de usuarios, centralizando el acceso a la aplicación y sobre todo, explotando todas las nuevas herramientas que se han ido generando con el paso del tiempo.

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.

2.3. Evolución del mundo web

A modo de curiosidad, en el proyecto original, se consideraba poco factible la realización de este trabajo de fin de grado en versión web debido al escaso rendimiento que ofrecían las soluciones disponibles (tanto el uso de Applets como Javascript).

Hoy en día, disponemos de aplicaciones realmente complejas, con una elaborada funcionalidad gracias a la evolución tanto de los estándares

Capítulo 3

Tecnologías

3.1. 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.

3.1.1. Coffescript

3.1.2. 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, Dart ha quedado descartado por diversas razones:

1. no va

3.1.3. Typescript

Typescript es un lenguaje libre y de código abierto desarrollado por Microsoft que actúa como un superconjunto de Javascript, es decir incorpora los distin-

Figura 3.1: Ejemplo

tos 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.

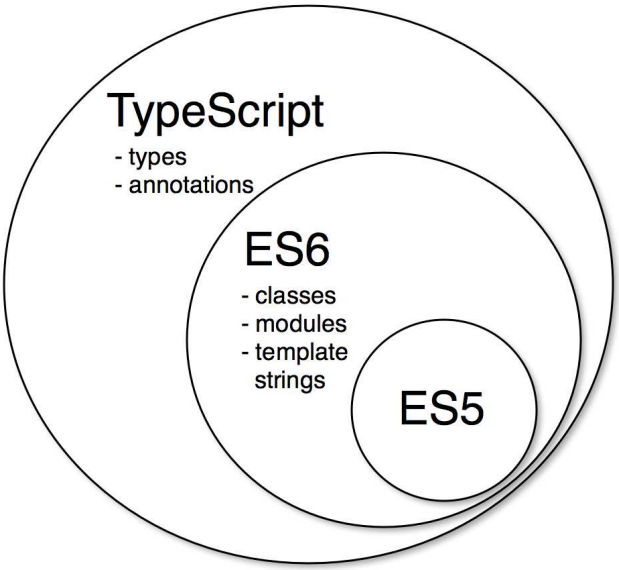

Figura 3.2: Typescript como superconunto de Javascript

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.

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 experiencia con Typescript.

3.2. Tecnología para la integración modelo vista

3.2.1. Webcomponents

Desde el inicio del proyecto, 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 servirme en esta tarea, pero muchos de ellos (como por ejemplo Angular), no son lo suficientemente flexibles y acaban condicionando el desarrollo de la aplicación.

Lo que SIMDE necesitaba era un proyecto que hiciera uso de los Web components, una serie de apis que permiten crear etiquetas html personalizadas y reutilizables. Los webs componentes consisten en 4 características principales que pueden usarse por separado o todos juntos:

1. **Elementos personalizados.**

2. **Shadow DOM.**

3. **HTML Imports.**

4. **HTML Templates.**

3.2.2. Polymer

Probablemente la librería basada en Web Componentes más conocida es Polymer, una librería desarrollada por Google y anunciada en 2013.

Pero a pesar de que las múltiples ventajas de Polymer existe una librería cuya comunidad es espléndida, y no es otra que React.

3.2.3. React

Existen una gran cantidad de motivos para escoger React sobre Polymer: Como por ejemplo, que ahora mismo se utiliza en decenas de aplicaciones importantes, como netflix, airbnb, Wallmart... (TODO INCLUIR CITA)

Que la comunidad es impresionantemente activa y cuenta con una gran cantidad de usuarios dispuestos a ayudar, así como cuenta con muchísima documentación y muchísimas implementaciones de librerías de terceros.

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".

Ademas React implementa operaciones sobre el DOM virtual de tal forma que las operaciones sobre el verdadero DOM sean eficientes.

3.3. Tecnología para hacer la build

Aunque este proceso pueda parecer trivial, para poder construir el empaquetado final de esta aplicación deben ejecutarse múltiples tareas:

1. Compilar el código typescript a javascript.
2. Compilar el código .tsx a .jsx.
3. Resolver los imports de los múltiples componentes.
4. Procesar el código sass y convertirlo en css.

3.3.1. Gulp/Grunt

Con la aparición de NodeJS, se crearon

3.3.2. Webpack

Una de las grandes herramientas de 2016 que acabó por cambiar el flujo de muchos desarrolladores web y desbancó a tasks runners como Grunt y Gulp fue webpack.

Para poder integrar todo este código y resolver el problema de los múltiples imports era necesario utilizar algún tipo de herramienta de gestión de paquetes, como por ejemplo commonjs, o requirejs. Sin embargo, webpack se encarga de resolver todas estas dependencias y crear statics assets para el navegador.

Uno de los mayores puntos a favor es que es altamente configurable, existen muchísimos plugins de webpack que permiten hacer preprocesamiento de css, tratamiento de imágenes, minimización codigo, étctera..

Actualmente en la aplicación de SIMDE webpack se encarga de compilar el código de typescript, el código de react y de generar un bundle

3.4. 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 cms.

Dado que la documentación es bastante extensa pero que en realidad, no es mas 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.

3.4.1. 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 mantneer 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.

3.4.2. Hexo

Hexo es un generador de contenido estático basado en NodeJS. No posee demasiadas diferencias destacables sobre el resto y ha sido escogido para este trabajo de fin de grado básicamente por estar basado también en Javascript, de tal forma que todo quede enfocado hacia javascript.

¿Comparativa hexo vs jekyll vs hugo?

Capítulo 4

Objetivos y fases

4.1. 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.

4.2. 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 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:

// INSERTAR CODIGO original

Se ha convertido en:

// CODIGO FLEX

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.
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 nos otorga cierto grado de

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 una de los ejercicios que se propone en SIMDE requiere realizar mejoras en el rendimiento de código haciendo uso de técnicas como el desenrollado de bucles, que dan lugar a códigos de considerable longitud.

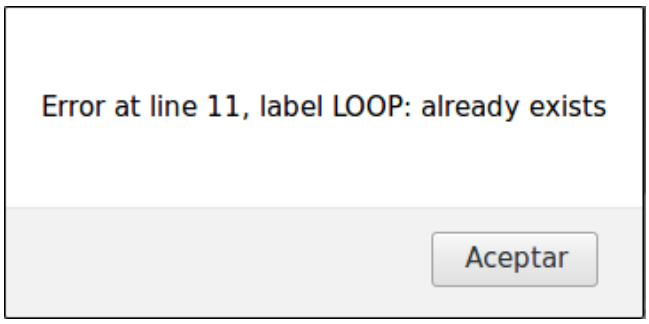

Figura 6.2: Ejemplos de errores en la nueva versión de simde

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] ACM LaTeX Style. http://www.acm.org/publications/latex_style/.

[2] FACOM OS IV SSL II USER'S GUIDE, 99SP0050E5. Technical report, 1990.

[3] D. H. Bailey and P. Swarztrauber. The fractional Fourier transform and applications. *SIAM Rev.*, 33(3):389–404, 1991.

[4] A. Bayliss, C. I. Goldstein, and E. Turkel. An iterative method for the Helmholtz equation. *J. Comp. Phys.*, 49:443–457, 1983.

[5] C. Darwin. *The Origin Of Species*. November 1859.

[6] C. Goldstein. Multigrid methods for elliptic problems in unbounded domains. *SIAM J. Numer. Anal.*, 30:159–183, 1993.

[7] P. Swarztrauber. *Vectorizing the FFTs*. Academic Press, New York, 1982.

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