UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Desarrollo de una interfaz web para un simulador de procesadores superescalares

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Rafael Rodríguez Real

Tutor: Pedro López Rodríguez
Sergio Sáez Barona
Salvador Petit Martí

Curso 2019/2020

Resumen

En este Trabajo Fin de Grado se diseña y desarrolla una interfaz web de una sóla página (SPA), destinada a ser la parte visual de un simulador de procesadores superescalares para uso docente.

Mediante la interfaz, el usuario puede modificar los parámetros de ejecución de dicho simulador, además de cargar y ejecutar sus propios ficheros de código ensamblador, para, una vez ejecutado el código, visualizar los diferentes datos proporcionados por la salida del simulador. Estos datos son procesados e incluidos en diferentes tablas que representan el estado del procesador en cada ciclo. El diseño de la interfaz es extensible para facilitar que, en un futuro, se añadan más tablas con mínimas modificaciones en el código. La implementación de la interfaz se ha realizado utilizando exclusivamente tecnologías front-end, concretamente JavaScript, HTML y CSS. También se han utilizado algunas funcionalidades del framework Material Design, Materialize CSS y el editor de código embebido CodeMirror. Para la implantación de interfaz y simulador se ha utilizado la tecnología Emscripten, que permite compilar el código en C del simulador y traducirlo a JavaScript. Mediante esta tecnología, tanto interfaz como simulador pueden ejecutarse sin necesidad de soporte dinámico en el back-end, lo que facilita la implantación en plataformas como PoliformaT.

Palabras clave: Front-end, SPA, JavaScript, HTML, CSS, Materialize CSS, CodeMirror, Emscripten, Interfaz, Simulador de procesadores superescalares

Abstract

In this Final Degree Project a single page web interface (SPA) is designed and developed, aimed to be the visual part of a superscalar processors simulator for teaching purposes.

With the application, the user can modify the simulator execution parameters, in addition to loading and executing assembly code files. Once the code has been executed the user can visualize the different data provided by the simulator output. Output data is processed and included in different tables that represent the state of the processor in each execution cycle. The interface design is extensible to facilitate adding more data tables with minimal source code modifications. The interface implementation has been done using specific front-end technologies, specifically JavaScript, HTML and CSS. Some functionalities of the Material Design framework, Materialize CSS, and the CodeMirror embedded code editor have also been leveraged. Emscripten technology has been used too, which allows the original simulator's C code to be compiled and translated into JavaScript. Using this technology, both the interface and the simulator can be run without the need of dynamic support in the back-end, which makes it easier the deployment of the application on platforms such as PoliformaT.

Keywords: Front-end, SPA, JavaScript, HTML, CSS, Materialize CSS, CodeMirror, Emscripten, Interface, Superscalar processors simulator

Tabla de contenidos

1) Introducción 8
1.1) Motivación 9
1.2) Objetivos 10
Impacto esperado 10
1.3) Convenciones 11
2) Estado del arte 12
2.1) WeMIPS 13
2.2) MIPS Interpreter 14
2.3) Visual MIPS 15
2.4) Problemática 16
2.5) El simulador actual de AIC 17
2.6) Problemática del actual simulador de AIC 21
2.7) Tabla comparativa de simuladores 22
3) Análisis del problema y propuesta 23
3.1) Análisis del problema 24
3.2) Propuesta 25
4) Diseño e implementación 27
4.1) Ficheros principales 28
4.2) Tecnologías de desarrollo 28
4.3) Interfaz de entrada 30
4.3.1) La barra de navegación 30
4.3.2) Espacio de configuraciones y el editor 33
4.3.2.1) Espacio de configuraciones 34
4.3.2.2) Editor de código 39
4.4) Interfaz de visualización de datos 43
4.4.1) La barra de navegación 43
4.4.2) Espacio para las tablas y el cronograma 46
4.5) Algoritmos para dotar de funcionalidad a la aplicación 52
4.5.1) Algoritmo para la creación de tablas 53
4.5.2) La estructura de datos del cronograma 58
4.5.3) Algoritmos necesarios para cambiar los datos 60
4.5.3.1) El algoritmo rellena_inicio() 61
4.5.3.2) El algoritmo hacia_adelante() 65
4.5.3.3) El algoritmo calculaPrev() 65

4.5.3.4) El algoritmo hacia_atras() ... 67
4.5.3.5) El algoritmo dibujaCrono() ... 70
4.5.4) El algoritmo de control de ciclo ... 76
4.5.5) Manejo de eventos y algoritmos adicionales ... 79
4.5.5.1) Botones de cambio de ciclo ... 80
4.5.5.2) Cambiar el tamaño del cronograma ... 80
4.5.5.3) El desplegable contenedor de las tablas ... 81
4.5.5.4) Botones de maximización y minimización ... 83
4.5.5.4.1) Maximizar las tablas ... 84
4.5.5.4.2) Maximizar el cronograma ... 85
4.5.5.4.3) Los botones de minimización ... 87
4.5.5.5) Volver a la pantalla de configuración ... 90
4.5.5.6) Funciones para leer y ejecutar ficheros ... 91
4.5.5.6.1) Función leerFichero() ... 91
4.5.5.6.2) Función ejecutarFichero() ... 93
4.5.5.6.3) El fichero ejecutar-ensamblador.js ... 95
5) Interfaz, implantación, y pruebas ... 99
5.1) Interfaz de carga y ejecución de ficheros ... 100
5.2) Interfaz de visualización de datos. ... 102
5.2.1) Funciones involucradas ... 102
5.2.2) Cambio de ciclo de simulación ... 103
5.2.3) Cambio de ciclo de simulación ... 106
5.2.4) Maximización y minimización de tablas y cronograma ... 107
5.5) Unificación de ficheros: Implantación final ... 108
5.6) Pruebas ... 110
6) Conclusiones ... 112
7) Relación con los estudios cursados ... 114
8) Trabajo futuro ... 115
9) Bibliografía ... 116
Glosario de términos ... 121
Anexo 1: Dependencias del proyecto y estructura ... 123
Anexo 2: El grid en Materialize CSS ... 125
Anexo 3: Las clases y reglas CSS utilizadas ... 127

Tabla de figuras

Figura 1: Interfaz de la web-app WeMIPS....13
Figura 2: Interfaz MIPS Interpreter y tabla de registros 14
Figura 3: Direcciones de memoria de MIPS Interpreter....14
Figura 4: La interfaz de Visual MIPS, su editor y las tablas separadas por pestañas 15
Figura 5: La interfaz de configuraciones, carga y ejecución....17
Figura 6: El <textarea> de estadísticas de salida....18
Figura 7: Página “INICIO” de la interfaz del simulador....19
Figura 8: Página "FINAL" de la interfaz del simulador 19
Figura 9: Página "Estado" de la interfaz del simulador 20
Figura 10: El cronograma de la interfaz del simulador.... 20
Figura 11: El apartado "BTB" de la interfaz del simulador....21
Figura 12: Diseño de la barra de carga y ejecución de archivos 30
Figura 13: Diseño del espacio de configuraciones y el editor.... 34
Figura 14: Diseño del espacio de configuraciones.... 35
Figura 15: Diseño del editor CodeMirror 39
Figura 16: El editor CodeMirror con el código "daxpy" 41
Figura 17: La interfaz de carga y ejecución de archivos, con las configuraciones y el editor (volteado).... 42
Figura 18: Diseño de la barra de navegación de visualización de datos.... 43
Figura 19: Diseño del espacio de visualización de tablas y cronograma 46
Figura 20: Diseño del contenedor de tablas horizontales....47
Figura 21: Diseño del contenedor de tablas verticales 47
Figura 22: Diseño del contenedor del cronograma.... 48
Figura 23: La interfaz de visualización completa (volteado) 50
Figura 24: El cronograma maximizado (volteado) 51
Figura 25: La interfaz de configuraciones, carga y ejecución de archivos 100
Figura 26: Código iterador de prueba cargado.... 101
Figura 27: Ejecución del código iterador, el botón "volver a carga" 101
Figura 28: Interfaz de visualización de datos....102
Figura 29: Tablas y cronograma con los valores iniciales....103
Figura 30: Tablas y cronograma en el ciclo 1, con los botones de cambio de ciclo y tamaño....104
Figura 31: Tablas y cronograma en el ciclo 22 104
Figura 32: Ciclo 50 con tamaño de cronograma 50 106
Figura 33: Ciclo 50 con tamaño de cronograma 5 106
Figura 34: Los botones de maximización....107
Figura 35: Tabla maximizada con botón de minimización 107
Figura 36: El directorio raíz 108
Figura 37: Ejecución del fichero HTML fuera de su directorio raíz....109
Figura 38: El código ensamblador iterador usado para las pruebas.... 110
Figura 39: Configuración empleada para las pruebas en ambos simuladores 110
Figura 40: Comparación de tiempos de ejecución entre simuladores....111
Figura 41: Árbol de directorios del proyecto 124

Índice de tablas

Tabla 1: Comparativa de simuladores 22
Tabla 2: Comparativa de simuladores final..... 113

1) Introducción

Son varias las asignaturas de la Universidad relacionadas con el aprendizaje del funcionamiento de los procesadores. En ellas, a la hora de enseñar a los alumnos se suelen utilizar simuladores de estos en las sesiones de prácticas, en vez de procesadores reales. Esto se debe a que los simuladores son realmente útiles a la hora de emular sistemas complejos o costosos de implementar físicamente en la vida real. Más aún, en el contexto de una asignatura con las limitaciones de tiempo y recursos disponibles.

Otro de los motivos por los que en la docencia se suele utilizar simuladores en vez de procesadores reales es que los primeros son mucho más flexibles, ya que, en un simulador, se pueden modificar los parámetros de configuración y aspectos de diseño del procesador fácilmente, sin necesidad de pasar por todo el proceso que que implica la implementación de un procesador real.

Por otro lado, en la industria, cuando, se pretenden evaluar nuevas arquitecturas de procesador o simplemente realizar modificaciones en el diseño de las arquitecturas existentes, muchas veces es preferible adaptar o construir un simulador de la nueva arquitectura para observar su comportamiento, ahorrando costes de fabricación y acelerando el proceso de diseño [WP|13].

En la asignatura de Arquitectura e Ingeniería de Computadores (AIC), actualmente se cuenta con dos simuladores desarrollados en lenguaje C por los profesores de la asignatura. Ambos simuladores modelan procesadores con la arquitectura MIPS64. Uno de los simuladores modela un procesador segmentado básico, mientras que el otro modela un procesador superescalar con soporte a gestión dinámica de instrucciones y ejecución especulativa.

El simulador del procesador superescalar mencionado tiene actualmente una interfaz web con la cual los alumnos pueden interactuar, pero su implementación no ofrece una interfaz moderna y presenta problemas de rendimiento y consumo de memoria. El objetivo de este proyecto es resolver estos problemas y actualizar la interfaz del simulador. Por ello, el proyecto se enfoca la implementación eficiente de una interfaz web de un simulador de procesador superescalar, es decir, la parte visible y manipulable para el usuario de dicho simulador de procesador, que cumpla los requisitos expresados por los docentes de la asignatura.

1.1) Motivación

Inicialmente, la asignatura AIC contaba con un simulador escrito en C, preparado para ejecutarse en un entorno GNU/Linux. Con el propósito de facilitar que el alumnado pueda ejecutar el simulador bien on-line o bien sin disponer de ese sistema operativo, los profesores de la asignatura compilaron el simulador a JavaScript mediante la tecnología Emscripten [EM|17], la cual permite embeber el simulador en cualquier interfaz web. En este sentido, los profesores de la asignatura desarrollaron una interfaz web básica con objetivos docentes.

Aunque la implementación actual de la interfaz dota de unas funcionalidades básicas, presenta algunos inconvenientes:

- ➔ El entorno de desarrollo no aprovecha todas las posibilidades que ofrece JavaScript, limitándose a generar, un gran array JavaScript que contiene páginas HTML, donde cada una de las cuales incluye datos correspondientes a un ciclo de ejecución del simulador.
- ➔ La ocupación en memoria de este array es considerable, debido a la cantidad de ficheros HTML que se generan, por lo que el consumo de recursos en el navegador limita la cantidad de ciclos que se pueden simular.

Así pues, la principal motivación de este Trabajo de Final de Grado es resolver estos problemas, desarrollando una interfaz alrededor del simulador compilado a JavaScript que se adapte a este y los solvente.

Uno de los requisitos de la asignatura es que el simulador esté contenido en una sola página, de modo que el usuario tenga, desde un sólo fichero HTML, acceso a toda la información devuelta por el simulador. Además, la aplicación se diseñará de manera que sea extensible, lo que permitirá añadir nuevas tablas de datos de simulación en un futuro sin tener que modificar, o modificando el mínimo posible el código fuente de la aplicación desarrollada en este TFG. También se desea que la aplicación web desarrollada sea independiente de ningún back-end, permitiéndonos la posibilidad de subir el proyecto a cualquier plataforma, por ejemplo, PoliformaT. En resumen, los alumnos podrán simular código ensamblador en cualquier lugar, conectado o no a internet y en cualquier tipo de sistema operativo, simplemente teniendo el acceso a un navegador.

Con este principal objetivo se utilizarán tecnologías orientadas al desarrollo web: el lenguaje de programación JavaScript para implementar la funcionalidad de la página, el lenguaje de marcado HTML para dotarlo de una estructura y un marco de trabajo estandarizado dedicado al desarrollo de la parte visual de la aplicación web combinado con reglas de hoja de estilos en cascada, CSS.

1.2) Objetivos

Actualmente, el simulador dispone de una interfaz web HTML puro que se genera cuando se realiza una simulación. El objetivo principal es diseñar una interfaz completamente nueva que solvente los problemas disctuidos, para ello se perseguirán los siguientes subobjetivos:

- ➔ Diseñar la interfaz de carga y de ejecución de archivos con sus menús correspondientes.
- ➔ Diseñar una interfaz de edición de código.
- ➔ Diseñar los menús de los parámetros de configuración del simulador.
- ➔ Diseñar la disposición de los datos, menús y tablas que se van a observar.
- ➔ Implementar mecanismos de entrada de los parámetros de configuración y del código ensamblador al simulador.
- ➔ Implementar mecanismos de interpretación de la salida de los datos del simulador.
- ➔ Implementar la lógica de la creación de tablas de datos de modo flexible para en un futuro poder modificar el número de tablas.
- ➔ Implementar la lógica de creación de un cronograma donde se vean los diferentes pasos que realizan las instrucciones en cada ciclo.
- ➔ Implementar la lógica para que el usuario pueda navegar entre los diferentes datos de salida, en forma de ciclos, que proporciona el simulador.
- ➔ Implementar un sistema que permita ampliar la visualización de los datos mostrados en algún punto.
- ➔ Implantar la aplicación de modo que no sea dependiente de ningún back-end o sistema operativo, respetando el funcionamiento del simulador original.

Impacto esperado

El impacto esperado de la aplicación es sustituir la interfaz web ya existente para dicho simulador, y que esta sea portable y ejecutable en diferentes dispositivos.

Se desea que pueda utilizarse en las clases de la Universidad, sobre todo en Arquitectura e Ingeniería de Computadores, para ayudar a los alumnos a comprender mejor el funcionamiento de un procesador.

1.3) Convenciones

Para facilitar la lectura y comprensión del documento, se han hecho uso de diferentes convenciones:

- ➔ El código fuente se mostrará en color AÑIL
- ➔ Las variables aparecerán en color MARRÓN
- ➔ Las menciones a funciones serán de color PÚRPURA
- ➔ Las referencias a elementos HTML y sus atributos se harán en *cursiva*
- ➔ Se entrecomillarán los identificadores, nombres de atributos, textos de menús o palabras extrañas cuando sea necesario.
- ➔ Se añadirán referencias bibliográficas entre corchetes: [REF].

2) Estado del arte

En este capítulo se discute el estado actual de los simuladores de procesadores, en concreto, aquellos que están disponibles para el usuario en Internet, en un entorno de aplicación web descargable y utilizable offline, tal como el que se pretende desarrollar en este TFG. Los simuladores tienen características similares a las del simulador objetivo de este TFG. Es decir, son simuladores de procesadores MIPS, accesibles online y orientados a la docencia de estructura y arquitectura de computadores. Los simuladores que se discuten son:

- ➔ WeMIPS
- ➔ MIPS Interpreter
- ➔ Visual MIPS

Posteriormente se comentará el simulador del cual partimos, el implementado por los profesores de Arquitectura e Ingeniería de Computadores de la Universitat Politècnica de València. Se presentarán sus características y se pasará a analizar aquellos inconvenientes que presenta y que se pretenden solventar.

2.1) WeMIPS

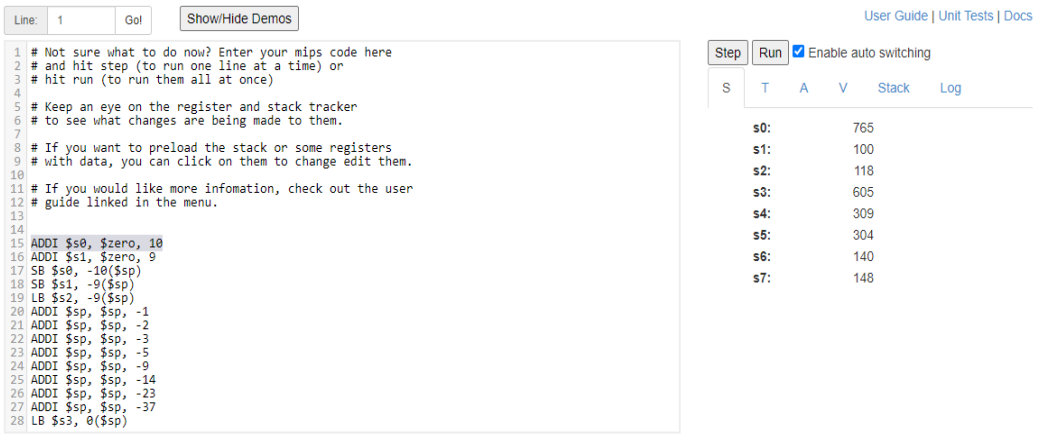


Figura 1: Interfaz de la web-app WeMIPS

WeMIPS es una web-app descargable y utilizable offline desde cualquier computador, por ello, no necesita ningún back-end para funcionar [EO|14]. Está implementada con varios ficheros JavaScript y un fichero HTML, los cuales trabajan conjuntamente. Su interfaz visual se ha realizado con Bootstrap [BO|20].

Es una aplicación que no simula un procesador superescalar, sino sólo un subconjunto de instrucciones MIPS. Este simulador no proporciona ningún tipo de cronograma, indispensable para observar el estado de la ejecución de las instrucciones en cada momento. Sin embargo, si que se aprecian tablas separadas en pestañas destinadas a mostrar el valor de los registros, la pila de memoria y algunos logs.

Además, dispone de varios códigos de demostración consistentes en diferentes programas preparados para cargar en el editor y ejecutarlos al pulsar el botón “Show/Hide Demos”.

La Figura 1 muestra la apariencia del simulador. Como se puede apreciar, presenta una interfaz minimalista con pocos colores, un editor CodeMirror donde aparecen marcadas las instrucciones del programa a ejecutar, la instrucción en curso y los botones “Step” para avanzar y “Run” para ejecutar el programa completo.

2.2) MIPS Interpreter

MIPS Interpreter también es una web-app descargable y utilizable offline sin necesidad de back-end [DY|18]. Al igual que la anterior, se presenta en una sola página que actualiza su contenido durante el transcurso de la ejecución mediante JavaScript.

Esta aplicación está orientada a la interpretación y ejecución de código ensamblador. Como la anterior, simula también un subconjunto de instrucciones MIPS.

Tiene un editor CodeMirror, como se puede apreciar en la Figura 2, donde el usuario puede insertar su código a ejecutar y tres botones de menú principales: “Reset”, “Step” para realizar un paso en la ejecución y “Run” para ejecutar el programa completo. Cuenta con una opción para escoger la frecuencia de la CPU a simular.

Como WeMIPS, Está implementada con varios ficheros JavaScript y HTML. Además, su interfaz visual también se ha realizado con Bootstrap [BO|20].

En esta aplicación solo se aprecian dos tablas y carece de cronograma. En la Figura 2 observamos la tabla de registros y en la Figura 3 la de direcciones de memoria. Están colocadas a modo que, si el usuario quiere ver la tabla de direcciones de memoria, debe hacer scroll hacia abajo lo cual es un poco incómodo. Una característica resaltable es que dispone de un menú implementado con el que se puede descargar el contenido de la memoria en formato txt.

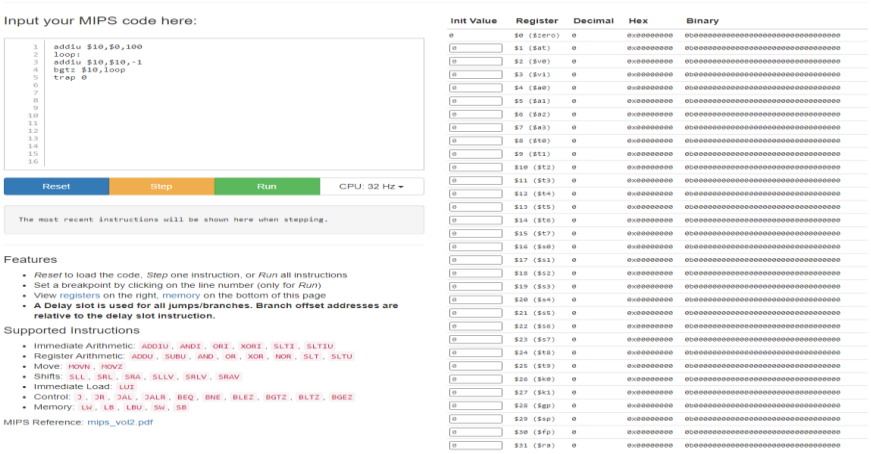


Figura 2: Interfaz MIPS Interpreter y tabla de registros

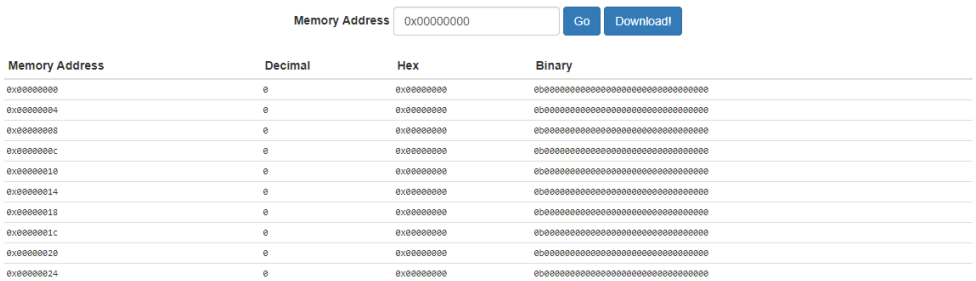


Figura 3: Direcciones de memoria de MIPS Interpreter

2.3) Visual MIPS

Visual MIPS (ver Figura 4) es otro ejemplo de una web-app que se puede descargar y utilizar offline sin necesidad de back-end, como las mencionadas anteriormente [WB|18]. También se presenta como una página que va cambiando durante el transcurso de la ejecución. En la página, se comenta que se ha desarrollado con el lenguaje F# y se ha compilado a JavaScript mediante el compilador FABLE. Su interfaz visual, se ha desarrollado utilizando el framework Materialize CSS [MC|14-20].

Como las anteriores aplicaciones, Visual MIPS está orientado a la interpretación y ejecución de código ensamblador, simulando también un subconjunto de las instrucciones MIPS.

También dispone de un editor CodeMirror, en el cual se sitúan las instrucciones MIPS a simular. La salida del simulador se representa a través de tablas. Bajo el editor de texto, hay un espacio reservado para mostrar un log de errores.

Visual MIPS cuenta con cuatro pestañas que separan las tablas de resultados. Las dos primeras, "Set 1" y "Set 2" muestran los registros. En la siguiente pestaña se muestra el PC actual y próximo y, en la última, las direcciones de memoria.

El usuario tiene la opción de mostrar los valores de las tablas en decimal, hexadecimal o en binario mediante unos botones incrustados en ellas. Al igual que las anteriores aplicaciones, carece de cronograma.

En cuanto a sus menús de ejecución, se puede simular todo el código mediante el botón "Execute", se puede reiniciar la ejecución actual mediante el botón "Reset", y, además, nos permite realizar un paso hacia adelante o hacia atrás mediante los botones "Step fordwards" y "Step backwards".

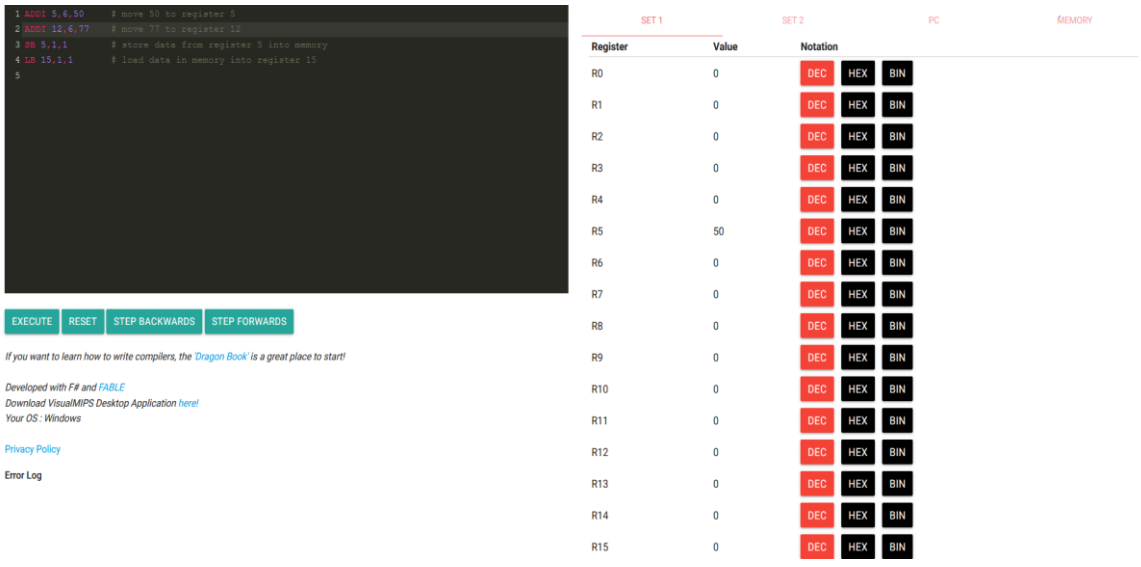


Figura 4: La interfaz de Visual MIPS, su editor y las tablas separadas por pestañas

2.4) Problemática

Los simuladores mencionados anteriormente son válidos para la docencia del ensamblador del MIPS. Sin embargo, no simulan la arquitectura del procesador. Es decir, únicamente simulan la funcionalidad de un subconjunto de las instrucciones ensamblador del MIPS y no detallan como se realiza esta ejecución con una microarquitectura o *pipeline* específico.

Para la docencia de Arquitectura e Ingeniería de Computadores, se necesita un simulador más completo que muestre el detalle del procesador en cada ciclo de ejecución, es decir, que muestre el estado de las estructuras del procesador relacionadas con las instrucciones que se están ejecutando, teniendo en cuenta, por ejemplo, la segmentación en etapas por las que pasan las instrucciones durante su ejecución. Además, el simulador debe ofrecer opciones que afecten al diseño de estas estructuras y de la arquitectura del procesador en general.

Sin embargo, de los simuladores comentados, se pueden extraer características deseables para la nueva interfaz del simulador de AIC, que son:

- ➔ Que se trate de una aplicación de una sóla página.
- ➔ Que no dependa de ningún back-end para poder publicar el simulador en PoliformaT para que pueda ser descargado y utilizado off-line por los alumnos.

Estas características, se convierten en requisitos que se tendrán en cuenta a la hora de crear la nueva interfaz para el simulador actual del que dispone la asignatura.

2.5) El simulador actual de AIC

Actualmente, la asignatura de Arquitectura e Ingeniería de Computadores utiliza un simulador MIPS que cumple con los requisitos educativos necesarios para explicar la arquitectura del procesador. En este apartado se discute la interfaz web de dicho simulador y su funcionamiento.

La interfaz, construida con HTML y CSS, está dividida en dos partes principalmente:

- ➔ La interfaz de configuraciones y de carga y ejecución de archivos
- ➔ La interfaz de visualización de datos.

La interfaz de configuraciones y de carga y ejecución se muestra en la Figura 5:

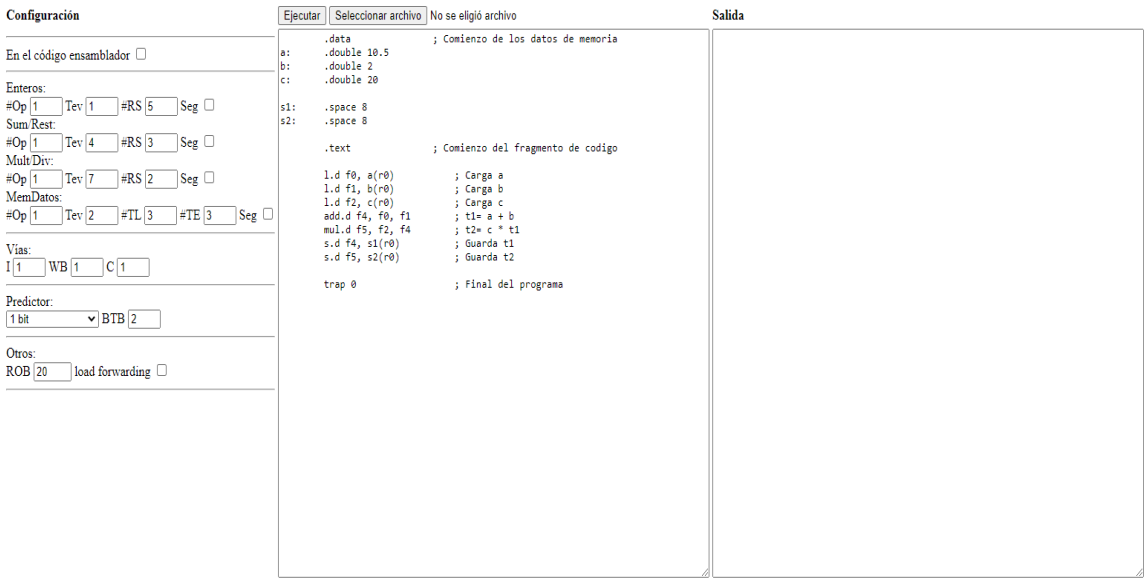


Figura 5: La interfaz de configuraciones, carga y ejecución

En ella, se observan a la izquierda los diferentes controles destinados a que el usuario pueda configurar diferentes aspectos de la arquitectura del procesador a simular. Entre ellos, las configuraciones de los diferentes operadores del procesador, como pueden ser el operador de enteros, el de suma y resta en coma flotante, los operadores de multiplicación y división, etc. Además de esto, se puede configurar la unidad de memoria y el número de vías que determinan el grado superescalar del procesador simulado. También es posible escoger el tipo de predictor de saltos y el número de entradas del *ReOrder Buffer* (ROB).

En el centro de la Figura 5 se sitúan, arriba, los botones de carga y ejecución de ficheros y justo debajo, en un <*textarea*> donde se puede editar el código, el código ensamblador a ejecutar. Inicialmente, este *textarea* contiene un código ejemplo disponible por defecto.

Por último, a la derecha del editor de código se encuentra otro <*textarea*> *(ver* Figura 6*)* que proporciona las estadísticas de ejecución y que también actúa como log de la ejecución una vez el usuario pulsa el botón "Ejecutar".

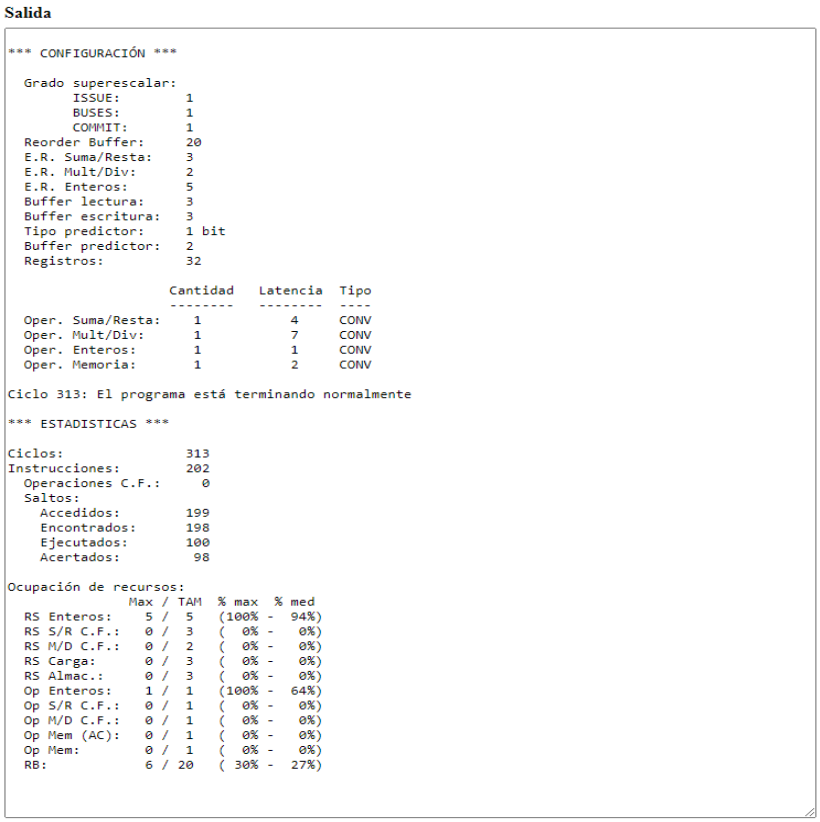
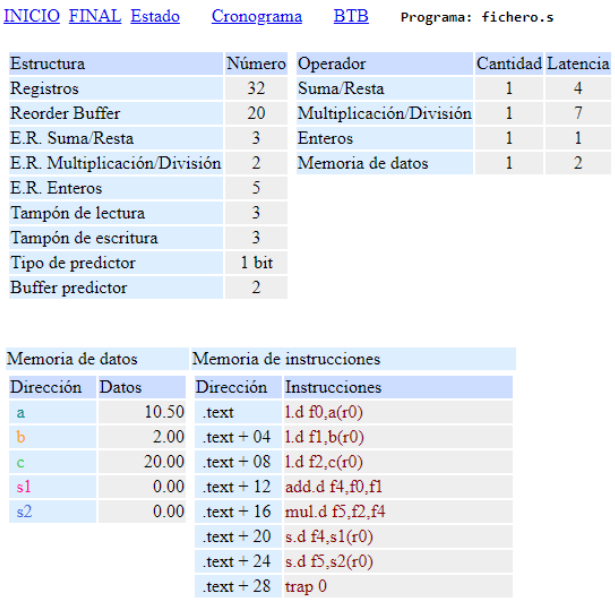


Figura 6: El <textarea> de estadísticas de salida

Cuando se pulsa "Ejecutar" se abre una ventana nueva que contiene la interfaz de visualización de datos. Dicha ventana tiene un menú con enlaces en la parte superior mediante el cual se puede navegar dentro de la visualización.

Mediante el enlace "INICIO" se accede a la ventana que se abre por defecto y se muestra en la Figura 7. En esta ventana, el usuario puede observar unas tablas que contienen información del inicio de la ejecución del simulador. En ellas se muestran las estructuras y operadores disponibles en el procesador simulado y el contenido inicial de la memoria de datos y la memoria de instrucciones.



INICIO FINAL Estado Cronograma BTB Programa: fichero.s

Estructura	Número
Registros	32
Reorder Buffer	20
E.R. Suma/Resta	3
E.R. Multiplicación/División	2
E.R. Enteros	5
Tampón de lectura	3
Tampón de escritura	3
Tipo de predictor	1 bit
Buffer predictor	2

Operador	Cantidad	Latencia
Suma/Resta	1	4
Multiplicación/División	1	7
Enteros	1	1
Memoria de datos	1	2

Memoria de datos

Dirección	Datos
a	10.50
b	2.00
c	20.00
s1	0.00
s2	0.00

Memoria de instrucciones

Dirección	Instrucciones
.text	l.d f0,a(r0)
.text + 04	l.d f1,b(r0)
.text + 08	l.d f2,c(r0)
.text + 12	add.d f4,f0,f1
.text + 16	mul.d f5,f2,f4
.text + 20	s.d f4,s1(r0)
.text + 24	s.d f5,s2(r0)
.text + 28	trap 0

Figura 7: Página "INICIO" de la interfaz del simulador

Mediante el enlace "FINAL" se accede a la página mostrada en la Figura 8 , donde el usuario puede observar el contenido de la memoria de datos e instrucciones al final de la ejecución, información sobre estructuras y operadores y dos tablas en las que aparece información estadística de la simulación y la ocupación de recursos.

INICIO FINAL Estado Cronograma BTB Programa: fichero.s

Ciclos	Instrucciones	CPI	Op. CF	Op. CF/ciclo	Saltos buscados	Saltos en BTB	Saltos ejecutados (C)	Saltos acertados
27	8	3.38	2	0.07	0	0 (-nan%)	0 (0.0%)	0 (-nan%)

Ocupacion de recursos

RS Ent.	RS S/R	RS M/D	RS Carga	RS Almac.	Op Ent.	Op S/R	Op M/D	Op AC	Op Mem.	RB
0 (0%)	1 (33%)	1 (50%)	3 (100%)	2 (66%)	0 (0%)	1 (100%)	1 (100%)	1 (100%)	1 (100%)	6 (30%)

Estructura	Número
Registros	32
Reorder Buffer	20
E.R. Suma/Resta	3
E.R. Multiplicación/División	2
E.R. Enteros	5
Tampón de lectura	3
Tampón de escritura	3
Tipo de predictor	1 bit
Buffer predictor	2

Operador	Cantidad	Latencia
Suma/Resta	1	4
Multiplicación/División	1	7
Enteros	1	1
Memoria de datos	1	2

Memoria de datos

Dir	+0	+1	+2	+3
a	10.50			
a + 04				
b	2.00			
b + 04				
c	20.00			
c + 04				
s1	12.50			
s1 + 04				
s2	250.00			
s2 + 04				

Memoria de instrucciones

Dirección	Instrucciones
.text	l.d f0,a(r0)
.text + 04	l.d f1,b(r0)
.text + 08	l.d f2,c(r0)
.text + 12	add.d f4,f0,f1
.text + 16	mul.d f5,f2,f4
.text + 20	s.d f4,s1(r0)
.text + 24	s.d f5,s2(r0)
.text + 28	trap 0

Figura 8: Página "FINAL" de la interfaz del simulador

En “Estado”, como su nombre indica, se muestran mediante tablas el contenido de las estructuras del procesador y la memoria. Un ejemplo de esta página se muestra en la Figura 9. Hay una página “Estado” para cada ciclo de ejecución, comenzando por el primero. En particular, se observa el ciclo indicado en la parte superior, los registros, la memoria y el contenido de estructuras como el ROB. En el menú superior se dispone de unos enlaces que permiten al usuario moverse en pasos de uno o en bloques de cinco ciclos hacia adelante o hacia atrás del estado actual.

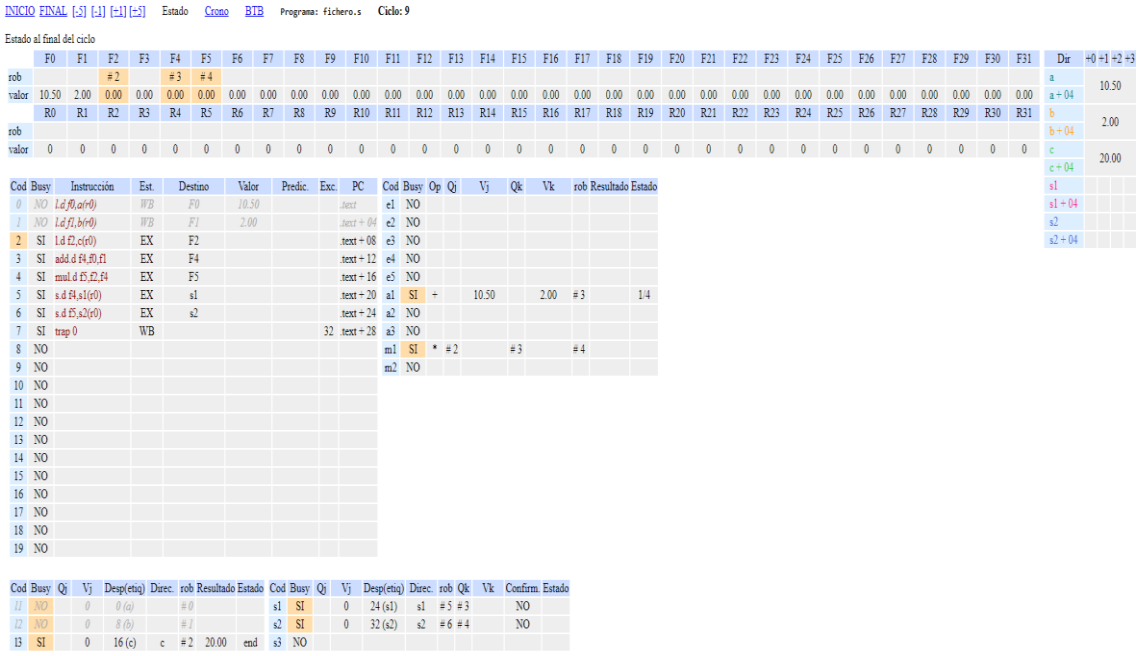


Figura 9: Página "Estado" de la interfaz del simulador

Seguidamente, en la página “Crono” (visible en la Figura 10) se visualiza el cronograma hasta un ciclo determinado de la ejecución de las instrucciones. Si en la primera línea del cronograma pulsamos sobre sobre un ciclo diferente, se carga la página correspondiente hasta ese ciclo en concreto. Como en el caso de la página “Estado” hay tantas páginas “Crono” como ciclos de ejecución.

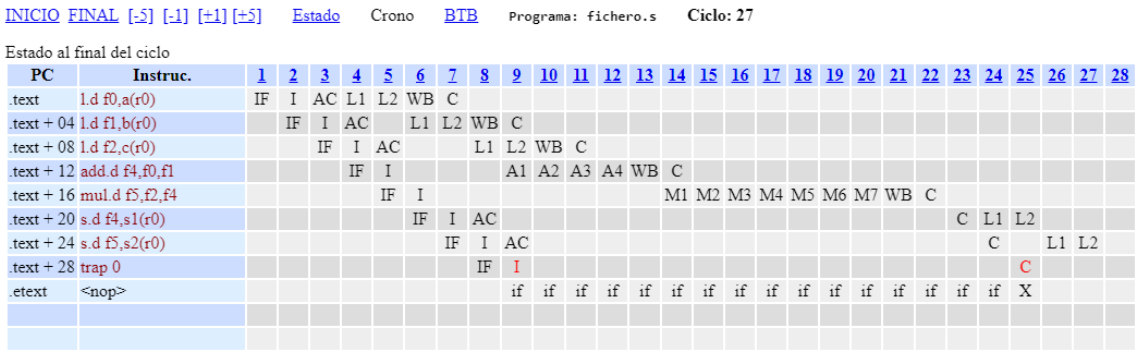


Figura 10: El cronograma de la interfaz del simulador

Por último, en la página “BTB”, representada en la Figura 11, se observa el estado del predictor de saltos. Como en el caso de las páginas “Estado” y “Crono”, el simulador genera una página “BTB” para cada ciclo.

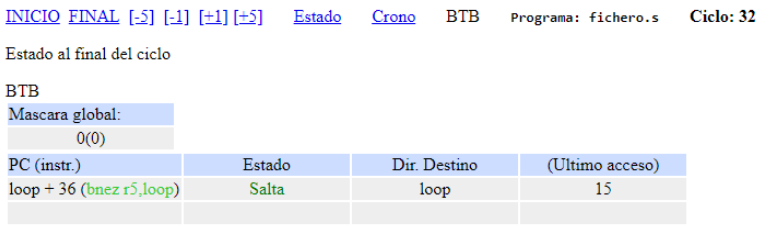


Figura 11: El apartado "BTB" de la interfaz del simulador

2.6) Problemática del actual simulador de AIC

El simulador cuenta actualmente con una interfaz de visualización que se genera una vez ejecuta el simulador. Como se ha comentado, en ella hay diferentes secciones, separadas en diferentes páginas, en las cuales el usuario puede observar el estado de la simulación y el cronograma de ejecución en cada ciclo. También puede avanzar o retroceder cambiando el ciclo de visualización y observar los datos iniciales o finales de la ejecución.

La interfaz actual cumple con los requisitos de simulador web docente que necesita la asignatura, no obstante, presenta un principal inconveniente: el simulador genera como salida una gran cantidad de páginas HTML (varias por ciclo de simulación), contenidas en un array JavaScript. Debido a este funcionamiento, cuando el número de ciclos es muy alto, el tamaño del array causa un desbordamiento de la memoria que los navegadores destinan a las aplicaciones web. Esto limita el tipo y complejidad de los programas que se pueden utilizar en el simulador.

Además de resolver este problema, se desea actualizar la interfaz del simulador para que sea más atractiva y que presente un diseño más actual y amigable.

Finalmente, se desea que la aplicación desarrollada en este TFG muestre toda la información en una sola página y que además esté contenida en un solo fichero HTML publicable fácilmente en plataformas como PoliformaT.

2.7) Tabla comparativa de simuladores

En la Tabla 1 se comparan los simuladores analizados con el objetivo de visualizar de manera directa las diferencias entre ellos.

Tabla 1: Comparativa de simuladores

Prestación	**WeMIPS**	**MIPS Interp.**	**Visual MIPS**	**Sim. AIC**
Cronograma	NO	NO	NO	SÍ
Tamaño de cronograma	-	-	-	NO
Reset	NO	SÍ	SÍ	SÍ (1*)
Paso atrás	NO	NO	SÍ	SÍ
Paso adelante	SÍ	SÍ	SÍ	SÍ
Múltiples pasos	NO	NO	NO	SÍ (2*)
Editor CodeMiror	SÍ	SÍ	SÍ	NO
Tablas de estado	NO	NO	NO	SÍ
Tablas de registros	SÍ	SÍ	SÍ	SÍ
Tabla dir. Memoria	SÍ	SÍ	SÍ	SÍ
Una sola página	SÍ	SÍ	SÍ	NO
Código de muestra	SÍ	NO	SÍ	SÍ
Configuración del procesador simulado	NO	SÍ (3*)	NO	SÍ
Un sólo archivo	NO	NO	NO	SÍ

(1*) Es posible volver al ciclo inicial pulsando sobre él en el cronograma

(2*) Hasta en bloques de ± 5 pasos

(3*) Únicamente cambiar la frecuencia de la CPU a simular

3) Análisis del problema y propuesta

En este capítulo se analiza el funcionamiento de la interfaz del simulador de la asignatura y se propone una alternativa a la interfaz actual.

3.1) Análisis del problema

El problema de la generación de ficheros HTML puede solventarse si se analiza previamente el funcionamiento de la interfaz del simulador:

Se dispone de un simulador de procesador MIPS implementado en C y compilado a JavaScript mediante Emscripten, al que hay que enviar una entrada y recoger e interpretar su salida tras la ejecución de un código ensamblador.

- ➔ La entrada de la ejecución está formada, por una parte, por el código que se quiere simular y, por otra parte, por los parámetros que el usuario desde el menú de la interfaz decide escoger para realizar la ejecución en el simulador.
- ➔ Se simulará el código teniendo en cuenta la configuración escogida en la interfaz.
- ➔ En la salida el simulador proporcionará los datos obtenidos de dicha ejecución los cuales se interpretan y procesan para mostrar al usuario.
- ➔ Se muestran los datos iniciales al usuario (ciclo 1).
- ➔ Una vez mostrados los datos iniciales de la salida, se puede navegar por los diferentes ciclos de ejecución del simulador, mostrando en cada uno de ellos los datos pertenecientes a dicho ciclo.

Dado el funcionamiento de la aplicación, se debe decidir una forma alternativa a la generación de un array de ficheros HTML a la salida de la simulación, justo antes de la interpretación de los datos para mostrar al usuario. Esto se debe a que actualmente con esta aproximación existe el inconveniente del desbordamiento de la memoria, comentado anteriormente.

La alternativa consiste en cambiar el modo en que el simulador retorna los datos obtenidos después de la ejecución. Para ello, se va a utilizar como salida una representación en JSON, que tiene la ventaja de ocupar mucho menos espacio. De este modo, el simulador retornará a la salida una variable JSON con toda la información referente a la ejecución, incluyendo una descripción de las tablas de datos a mostrar y el contenido de estas para cada uno de los ciclos de simulación.

En resumen, la aplicación, compuesta por un único fichero HTML mostrará toda la información de la simulación, evitando la generación y almacenamiento de una página HTML para cada uno de los apartados y cada uno de los ciclos de ejecución.

3.2) Propuesta

Una vez analizado el estado del arte y el simulador actual de AIC, la propuesta de este proyecto es la de construir con los principales lenguajes de programación orientados a la web, una interfaz que facilite la comprensión de cómo funciona la arquitectura de un procesador MIPS. La aplicación debe presentar un diseño amigable para que pueda ser utilizada fácilmente por los alumnos de la asignatura para observar, dinámicamente, mediante un cronograma y varias tablas de estado, cómo responde dicho procesador al ejecutar instrucciones en cada ciclo.

Las tablas en las cuales se observarán los datos a mostrar estarán programadas de modo genérico, para que los docentes puedan decidir si quieren añadir más tablas en la salida de datos de la ejecución del simulador, sin tener que abordar grandes modificaciones en el código de la aplicación. Esta salida, será una variable JSON que será procesada e interpretada por la interfaz web, organizando los datos en las tablas especificadas en la misma variable.

Como se ha explicado anteriormente, en el análisis de la interfaz que se usa actualmente, la ejecución de la nueva aplicación también tendrá ciertas partes principales que deberemos tener en cuenta a la hora de implementarla. Cada una de ellas será independiente del resto, pero deberán trabajar en conjunto para proporcionar el funcionamiento deseado:

- ➔ Interfaz de entrada:

Esta interfaz contendrá los menús necesarios para seleccionar las diferentes configuraciones de ejecución del simulador del procesador. Será la parte de entrada de la aplicación y únicamente se saldrá de ella una vez hayamos seleccionado una configuración y escogido un programa a ejecutar. Tendrá unas opciones por defecto, así como un programa cargado previamente para, en caso de no subir ningún fichero, poder realizar una ejecución de ejemplo predeterminada.

- ➔ Ejecución del simulador:

En esta parte, el simulador recibe como entrada los parámetros de configuración y el código escogidos por el usuario y realiza la simulación. El simulador ya está implementado y se ha compilado a Javascript para poder embeberse en una aplicación web, por lo que sólo hay que preocuparse de que los parámetros y el código lleguen como entrada en el formato correcto.

- ➔ Datos de salida:

Una vez finalizada la ejecución, el simulador retornará una variable JSON, como se ha descrito anteriormente, con los datos y resultados de la simulación. En esta parte para

su correcta interpretación y posterior visualización en la interfaz web, hay que identificar y separar los distintos datos de salida.

➔ Visualización de los datos:

Con la salida obtenida, se crearán dinámicante las tablas y el cronograma necesario para la visualización. Se mostrarán los datos iniciales y se deberá ser capaz de poder navegar por los diferentes ciclos de ejecución, para mostrar al usuario los distintos estados por los que ha pasado el simulador al ejecutar el código que se ha decidido simular.

Se requiere que el código para las tablas esté implementado a modo que este sea agnóstico al contenido y formato de esta. De este modo, desde la variable JSON se indicará la información para la creación de la tabla y el código de la aplicación la interpretará y la generará independientemente del resto o de su contenido. Así, si en un futuro el docente decide modificar el número de tablas a visualizar, podrá hacerlo y el código de la interfaz se adaptará a esta decisión sin necesidad de modificar la implementación de la aplicación.

En cuanto a la aproximación de la implementación de la web-app, se ha decidido, como en los tres primeros simuladores analizados en el estado del arte, que se trate de una SPA o *Single Page Aplication*: Una SPA, como su nombre indica, consiste en una aplicación web dinámica de una sóla página, cuyo contenido o funcionalidad cambia según el usuario interactúe con ella [DJ|18] [KP|19].

Además, está SPA estará contenida en un único fichero HTML De este modo, cargando un único archivo HTML, todas las funcionalidades del simulador estarán disponibles para el usuario. Además, la aplicación no será dependiente de ningún back-end, permitiendo mayor velocidad, disponibilidad de la aplicación, y una interacción más fluida.

4) Diseño e implementación

Para llevar a cabo la implementación habrá que seguir ciertos pasos indispensables para la estructuración y desarrollo de la aplicación según los criterios mencionados anteriormente. En los siguientes apartados se irán desarrollando cada uno de los puntos necesarios para abarcar la construcción de la aplicación.

4.1) Ficheros principales

En los siguientes apartados se definen más detalladamente cada una de las acciones que hay que realizar para la implementación del proyecto. Estas acciones se desarrollan en cuatro ficheros principales:

- ➔ *index.html*: Este fichero contiene casi todo el texto HTML necesario para construir la estructura de la aplicación.
- ➔ *app.js*: Fichero JavaScript que se encarga de toda la lógica de la página y de modificar el código HTML cuando sea necesario.
- ➔ *style.css*: Fichero con la hoja de estilo en el en el cual se añadirá dinámicamente código de hoja de estilos adicional para adaptar el apartado visual cuando sea necesario.
- ➔ *ejecutar-ensamblador.js*: Fichero con el código JavaScript correspondiente al simulador transcompilado desde C.

Estos ficheros serán unificados en un único archivo HTML, mediante un shell script que compila en un sólo fichero toda la funcionalidad de estos. El objetivo es disponer de toda la aplicación en un solo archivo publicable en cualquier plataforma sin soporte de back-end, como PoliformaT, desde donde los alumnos puedan utilizarlo.

4.2) Tecnologías de desarrollo

Elección de tecnologías de desarrollo

Para este proyecto era necesario escoger entre las diferentes opciones de desarrollo front-end, puesto que lo que nos atañe es el funcionamiento y la apariencia de la aplicación por el lado del cliente, ya que carece de back-end [AT|06]. Las tecnologías escogidas han sido las siguientes:

- ➔ HTML: Se ha escogido ya que es el lenguaje de marcado estándar para la elaboración de páginas web, mediante el cual vamos a ir construyendo la estructura interna de la aplicación. Una vez esté construido nos ayudaremos de algunas funciones de manipulado del DOM (Document Object Model) para ir variando la página según nuestras necesidades.

- ➔ JavaScript: Escogido por ser un lenguaje esencial para la programación web del lado del cliente y ser soportado por todos los navegadores modernos más usados. Es un lenguaje interpretado, débilmente tipado y orientado a objetos. Unido a JQuery, proporcionan todo lo necesario para implementar las diferentes

funcionalidades de la lógica de la aplicación, así como la interacción con los elementos del DOM.

➔ JQuery: Se ha escogido por ser una biblioteca JavaScript que tiene diversas funcionalidades como manejar eventos, simplificar la manipulación código HTML y la forma de interacción con el DOM. Se puede extender con nuevas funcionalidades y es altamente usado hoy día en el desarrollo de páginas web, hasta el punto de que actualmente se ha convertido en prácticamente esencial. En el proyecto se le ha dado uso sobre todo a la hora de manejar algunos eventos de la aplicación. Se ha indexado su código fuente vía CDN para reducir el espacio en disco ocupado por el proyecto.

➔ Materialize CSS: A la hora de dar estilo a la parte visual de la página, se ha deicidido utilizar Materialize CSS [MC|14-20]. Es un framework de CSS basado en Material Design, una normativa de diseño desarrollada por Google enfocada en la visualización para diferentes tipos de dispositivo. Su elección proporciona una manera intuitiva y simple de crear layouts para la aplicación, así como una hoja de estilos CSS predefinida con diferentes opciones de personalización y varios tipos de componentes. Se considera que tiene todo lo necesario para llevar a cabo la parte visual de la aplicación obteniéndose, además, una presentación atractiva para el usuario.

➔ CSS: adicional y complementariamente al framework Materialize CSS, se va a utilizar el popular lenguaje de estilos en cascada para cosas más concretas como son el estilo de las tablas, añadir clases a algunos elementos del DOM para que contrasten con otros y aspectos menores en cuanto al diseño de la parte visual.

➔ CodeMirror: es un editor de texto implementado en JavaScript, preparado para ser incrustado en una página web con diferentes temas y modos de visualización en forma de diferentes estilos CSS. Tiene su propia API [CM|20] con la cual se puede interactuar con el editor para realizar diferentes funciones. Tiene soporte para una gran cantidad de lenguajes de programación. A pesar de que el lenguaje ensamblador del simulador no está soportado, aun así, se considera una buena elección a la hora de editar y manipular texto debido a su atractiva e intuitiva interfaz para el usuario.

➔ Emscripten: Es un transcompilador. En este caso, se encarga de procesar el código intermedio creado al compilar código C [EMS|15]. Como salida, devuelve un fichero JavaScript ya procesable por los navegadores web. Ha sido utilizado para portar el simulador, inicialmente implementado en C a JavaScript, permitiendo ejecutarlo desde el lado del cliente y obtener su resultado.

4.3) Interfaz de entrada

Esta interfaz será lo primero que se muestre una vez se acceda a la aplicación. Toda su construcción será realizada en el fichero Index.html. Deberá tener varios menús implementados para cambiar las diferentes configuraciones con las que se permite ejecutar el simulador. Tendrá que ser capaz de cargar ficheros de código ya escritos y mostrarlos en pantalla, editarlos y llamar al simulador para su ejecución.

La interfaz estará dividida en 2 partes:

- ➔ Una barra de navegación fija con los botones principales.
- ➔ Un espacio dónde colocar las diferentes configuraciones y el editor.

4.3.1) La barra de navegación

La barra de navegación de la interfaz de entrada (ver Figura 12) tendrá que ser capaz de proporcionar al usuario la capacidad de cargar ficheros de código para su posterior ejecución y visualización de los datos. Se ha decidido un modelo minimalista con dos botones destinados a dotar a la aplicación con estas dos funcionalidades, uno para ejecutar el código introducido o bien el código que el simulador tiene de prueba y otro para que el usuario pueda cargar sus propios ficheros. Además, se ha añadido un *checkbox* para indicar al usuario que se ha cargado un archivo, en el caso de haberlo hecho. La barra, deberá ser fija para que el usuario tenga visible sus opciones en todo momento. En este apartado se redacta cómo se ha implementado esta parte de la interfaz:

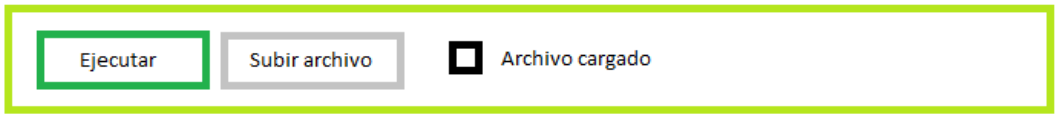


Figura 12: Diseño de la barra de carga y ejecución de archivos

Lo primero que hará será una barra de navegación fija que se mantenga en su posición, aunque el usuario haga scroll hacia abajo. De este modo los botones principales del menú siempre estarán visibles para este.

Para ello se crea una etiqueta <*div*> HTML con identificador, *id='navCarga'*, al cual se le agregan las clases de Materialize CSS correspondientes para que tenga una respuesta como la anteriormente comentada:

```
<div class='navbar-fixed' id='navCarga' style="display: block;">
    <nav class= "nav-extended teal lighten-4">
      <div class="nav-wrapper">
      </div>
    </nav>
  </div>
```

De este modo el <*div*> exterior tendrá la clase *"navbar-fixed"* es decir, navegador fijo. Contendrá un campo con la etiqueta <*nav*> el cual será la propia barra de navegación, cuya clase de Materialize CSS será *"nav-extended"*. Con esto le indicamos que se trata de una barra de navegación con menús, las clases "teal lighten-4" indican el color y el brillo del color de la barra. Dentro de esta, tendremos ubicado el <*div*> interior con la clase *"nav-wrapper"* es decir, este <*div*> será la "envoltura" del contenido de la barra de navegación.

El atributo *style='display: block'* indica que la barra de navegación será visible desde el primer momento que accedamos a la aplicación.

Una vez hecho esto, se incluirán dentro del <*div*> interior, aquellos componentes necesarios para crear el menú.

```
<div class="nav-content row">

      <div class="col s12"></div>

</div>
```

Estas etiquetas, que irán en el interior del <*div*> *"nav-wrapper",* se explican a continuación:

La etiqueta <*div*> exterior tiene varias clases, con *"nav-content"*, le indicamos al framework MaterializeCSS que el interior será el contenido de una barra de navegación. A continuación de *"nav-content"* aparece en el mismo *classList* la cadena *"row"*. Con *"row"* lo que se hace es indicarle al framework que el layout interior del navegador tendrá forma de fila.

La fila mencionada se va a dividir en doce subdivisiones, pero a modo de columna con la siguiente etiqueta <*div*> con la clase *"col s12"*. Así se le indica a nuestro framework que nos divida esa fila en una columna con doce subdivisiones y que el navegador ocupará esas subdivisiones.

Por último, se introduce en la etiqueta interior, una lista desordenada, <*ul*> con los elementos a mostrar dentro de nuestra barra de navegación:

```
<label class="waves-effect btn col s1" id="ejecutar"
onclick="ejecutar_ensamblador()">Ejecutar</label>
```

Se utiliza una etiqueta HTML <*label*> para representar una etiqueta para el botón *"Ejecutar"*.

Su atributo *class* contiene una lista de clases. La clase *"waves-effect"* provee el tipo de efecto que se quiere que Materialize aplique cuando se pulse en el *label*. Con *"btn"* Materialize entiende que dicho *label* es un botón, y le pondrá el estilo correspondiente. Se opta por el color turquesa por defecto de Materialize CSS en este botón para diferenciarlo del resto, ya que va a ser con el cual se va a ejecutar la llamada al simulador. Con *"col s1"* se le indica que es un elemento columna el cual va a ocupar una posición de las doce en las cuales habíamos dividido la etiqueta contenedora <*div*> anterior, aquella que tiene como atributo *class "col s12"*. Por último, este *label* llamará la función *ejecutar_ensamblador()* cuando se dispare un evento *onclick* sobre él, es decir, cuando el usuario clique el botón. El *label* tendrá como identificador *id="ejecutar"* con el cual se diferencia del resto a la hora de trabajar en JavaScript o en JQuery con él.

Por otra parte, se dispone de otro botón de funcionalidad "*Subir Archivo",* implementado también como un *label*, con la diferencia de que cuando se clique sobre él, se despegará un menú de carga de archivos:

```
<label class="waves-effect grey btn col s1" for="examinar">
Subir Archivo</label>
```

La clase de dicho *label* es igual a la del anterior, ocupando también un espacio del <*div*> superior con clase *"col s12"*, con la única diferencia de que en este se ha optado por un tono grisáceo, el cual será el tono de color predeterminado para el resto de los botones a partir de ahora.

En el *label*, el atributo *for="examinar"* indica que dicho *label* es para el siguiente elemento con *id="examinar"*, es decir, para el elemento con la etiqueta <*input*> que va a continuación de él.

```
<input type="file" id="examinar" onchange="leerFichero(this)"
style="display: none;"></input>
```

Este elemento <*input*> será el encargado de recoger el fichero de código que se quiera cargar. Para ello, se ha indicado en su atributo tipo, *"type"* el valor *"file"* de modo que HTML lo identifique como un elemento de carga de ficheros. Como se ha comentado anteriormente, se le ha añadido el identificador *"examinar",* esto es lo que consigue que el *label* anteriormente mencionado funcione como botón para este elemento. Se ha colocado su atributo *"style"* como *style="display: none;"* que lo que hace es que oculta este elemento, dejando visible únicamente el *label*. Así cuando el usuario clique en este, será cuando el *"input file"* se abra y permita cargar el archivo que se decida.

El elemento tiene un atributo *onchange* el cual es un evento que lanzará la función *leerFichero()* y le proveerá el valor *this*. Es decir, cuando el elemento cargue un archivo, se disparará el evento *onchange* y se ejecutará la función mencionada, inyectándole como valor el archivo que se ha decidido cargar.

Para finalizar, habrá un último <*label*> que contendrá dos elementos:

Un elemento <*input*> de tipo *checkbox*, que por defecto estará desactivado y sólo se activará cuando se haya cargado un fichero, y una etiqueta <*span*> que contendrá un pequeño texto informativo para el *checkbox*.

```
<label>
      <input type="checkbox" disabled="disabled" id="checkfile"/>
      <span style="color:black">Archivo cargado</span>
</label>
```

El atributo *style* de la etiqueta <*span*> únicamente indica el color del texto.

4.3.2) Espacio de configuraciones y el editor

Este espacio aparecerá justo debajo de la barra de navegación antes mencionada y será dividido en dos bloques (ver Figura 13). Su propósito es que el usuario pueda acceder a las diferentes opciones de configuración que tiene el simulador, para decidir algunos parámetros relativos a su ejecución.

Deberá ser capaz de visualizar un espacio de edición de código, donde se podrá observar y modificar un archivo que previamente haya cargado o bien editar el que se le proporciona por defecto.

Los bloques en los que va a estar dividido este espacio son justo estos: el espacio de configuraciones y el editor de código.

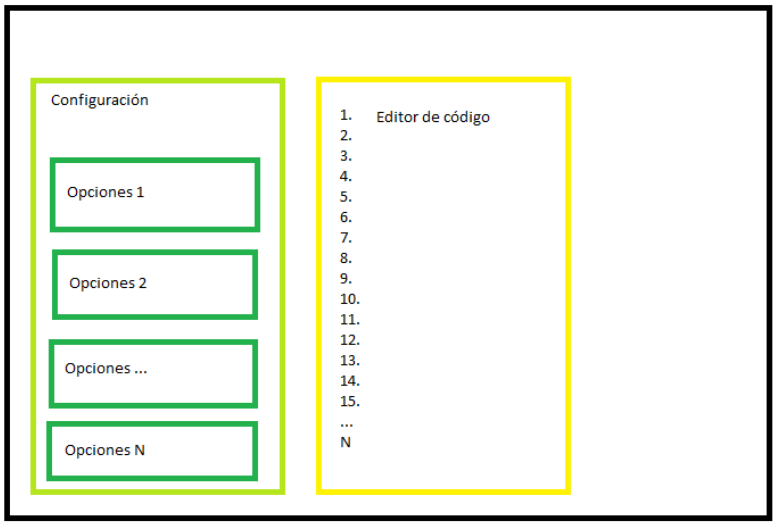


Figura 13: Diseño del espacio de configuraciones y el editor

El espacio principal será una etiqueta HTML <*div*> inmediatamente precedido de otro <*div*> con la clase *"row"* de Materialize CSS, de ese modo, conseguimos que todo el contenido aparezca alineado en una sola fila:

```
<div id='cargaDatos' style='display: block;'>

      <div class="row">Aquí irá el contenido</div>

</div>
```

Dentro del <*div class= "row"*> se proporciona el siguiente contenido:

4.3.2.1) Espacio de configuraciones

Este espacio (ver Figura 14), queda reservado para que el usuario escoja entre las diferentes opciones posibles de configuración a la hora de ejecutar el simulador. En el habrá diferentes elementos con los que el usuario podrá interactuar. Estos elementos serán de diferentes tipos según el tipo de configuración que representan.

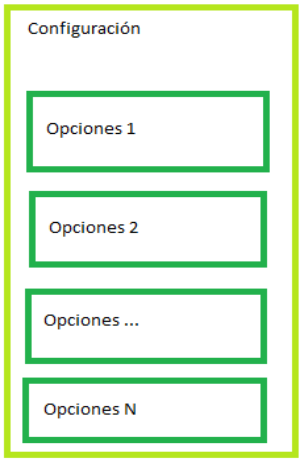


Figura 14: Diseño del espacio de configuraciones

Para este espacio de configuraciones con *id='configuración'* se ha decidido establecer una etiqueta <*div*> con la clase *"col s4"* dónde *s4* indica el tamaño en ancho de la clase columna:

```
<div class="col s4" id='configuracion'>
      <div class="card teal lighten-5">
      <div class="card-content black-text" ></div></div>
</div>
```

Dentro de dicho <*div*> se han incluido otros dos con las clases *"card"* y *"card-content"* los cuales indican a Materialize CSS que se trata de un elemento de tipo *card*, un tipo de contenedor que este framework ofrece. Dentro de *"card-content"* como es obvio, irá el contenido de la *"card"*.

El resto de las subclases que pueden apreciarse son únicamente para dar estilo al contenedor, con *"teal lighten-5"* se le indica el color y la luminosidad con el cual representar el elemento, y con *"black-text"* se le indica que el texto interior será de color negro.

Dentro de la etiqueta <*div*> de contenido será donde se van a disponer los elementos para conformar los menús:

```
<h class="brand-logo"><i class='material-icons'>filter_drama</i>
<b style="color: black;"> Configuración</b></h>
```

Este elemento es simplemente decorativo, es la cabecera de la carta, a la cual se le ha añadido un icono de la librería *material-icons* y el título en letra de color negro.

Seguidamente aparecen las opciones de configuración, cada una de ellas tendrá un identificador asociado mediante el cual se obtendrán los valores de los parámetros para configurar la ejecución del simulador. Estas opciones estarán separadas por una etiqueta <*hr*> o separador.

```
<hr>
<label style='float: right;'>
      <input type="checkbox" id='config_en_s' />
      <span style='color: black;'>En el código ensamblador</span>
</label>
```

El primer *checkbox* de configuración del simulador con su id y su etiqueta <*span*> de texto de acompañamiento tiene el estilo *style='float: right'* lo que indicará que este *checkbox* aparecerá a la derecha de la carta.

Los siguientes cuatro bloques de parámetros son prácticamente idénticos, diferenciándose únicamente en el texto que contienen y su identificador, ya que cada uno maneja una configuración específica para el simulador. En el caso de la configuración para los números enteros:

```
<hr>
<b>Enteros:</b>
<br>
#OP:
<div class="input-field inline row s1">
      <input id="NUM_INICIO_ENTEROS_p" value="1" size="1"
      style="text-align: center; background-color: white; border-
radius: 10px;">
</div>

         Tev:
```

```
<div class="input-field inline row s1">

      <input id="TEVAL_ENTEROS_p" value="1" size="1"

      style="text-align: center; background-color: white; border-
radius: 10px;">

</div>


         #RS:

<div class="input-field inline row s1">

      <input id="TAM_RS_ENTEROS_p" value="5" size="1"

      style="text-align: center; background-color: white; border-
radius: 10px;">

</div>


          &nbs
p;       

<label>

      <input type="checkbox" id="SEGMENTADO_ENTEROS_p">

      <span style='color: black;'>Seg</span>

</label>
```

Se incluye cada etiqueta <*input*> dentro de una etiqueta <*div*> con la clase *"input-field inline"* la cual le indica al framework que se trata de un campo de entrada y que debe estar alineado con el resto de los elementos. Esto también se fuerza al incluir como subclase *"row s1"* lo que le indica que es un elemento fila de tamaño mínimo. De este modo quedan alineados los tres <*div*> a los cuales les precede el nombre de los parámetros de configuración que se van a modificar, en este caso serán *#OP, Tev* y *#RS*. En cada <*input*> se incluye también el tamaño de estos y el valor por defecto a mostrar, además de algunos parámetros dentro del atributo *style*, el cual comparten algunos elementos.

En el caso de los <*input*> para números, *style="text-align: center; background-color: white; border-radius: 10px;*" está indicando que el texto de dentro del <*input*> estará alineado en el centro, tendrá un color de fondo blanco y el redondeado de su borde será de unos diez píxeles de radio.

Se han añadido algunas entidades de espacios en blanco HTML, * ,* para separar aún más los menús.

Como puede observarse, cada <*input*> tendrá su propio y único identificador, al igual que el resto de los parámetros configurables del simulador.

Por último, aparece una etiqueta <*label*> del tipo *checkbox* para la opción de activar o desactivar la opción de segmentado. Los próximos cuatro menús de configuración, como ya he comentado, son similares, cambiando sus identificadores.

El siguiente menú que difiere de los anteriores es un selector, el cual se incluirá dentro de una etiqueta HTML <*div*> con la clase anteriormente mencionada *"input-field inline"*, sólo que esta vez daremos un tamaño más grande al elemento mediante la subclase de *row, "s2"*.

```
<div class="input-field inline row s2">
<select id="tipo_predictor_p" class='browser-default'>
      <option value="" disabled selected>Predictor</option>
      <option value="0">1 bit</option>
      <option value="1">2 bit hist</option>
      <option value="2">2 bit sat</option>
      <option value="3">perfecto</option>
      <option value="4">clarividente</option>
      <option value="5">predict-not-taken</option>
</select>
</div>
```

Dentro de él, se ha incluido una etiqueta <*select*> que proporciona un desplegable con varias opciones en forma de etiquetas <*option*>, en este caso numeradas por medio de su atributo *value*, desde: "" (cadena vacía), la cual será por la que venga por defecto (opción *selected*) y no podrá ser seleccionada (opción *disabled*), a la opción cinco. La clase de la etiqueta <*select*>, *"browser-default"* le indica a Materialize CSS que debe usar el elemento selector por defecto del cual nos provee el navegador, ya que utilizar uno de su abanico de componentes no funcionaba correctamente en el layout establecido. Por último, se ha añadido una última sección, llamada "Otros" en la cual se incluye algo de configuración extra, en concreto la configuración para el tamaño del *ROB* o *reorder buffer* y la opción *load forwarding*.

4.3.2.2) Editor de código

El usuario debe ser capaz de cargar y editar código ensamblador en la aplicación para su posterior ejecución. La funcionalidad de carga y ejecución de datos le será dada por la barra de navegación. En este apartado se describirá el espacio reservado para insertar el editor de código proporcionado por CodeMirror, que será donde aparezca al inicio el código de prueba y donde, al cargar un fichero, se mostrará el código para que pueda ser editado previamente a su ejecución.

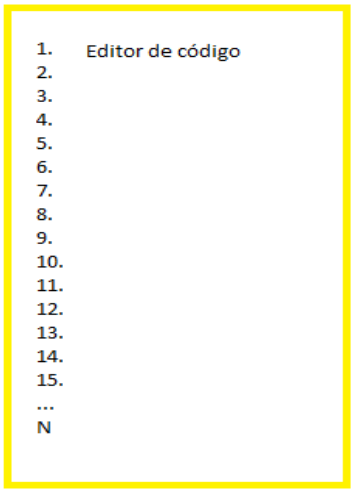


Figura 15: Diseño del editor CodeMirror

Este espacio para el editor de código aparecerá justo al lado de las configuraciones, esto se le indica a Materialize de la siguiente manera:

```
<div id='edit' class="col s8">
<br>
      <textarea id="ensamblador">
      </textarea>
</div>
```

Con la clase *col* y la subclase *s8*, se indica que este espacio con identificador *id="edit"* va a ocupar el resto de espacio que queda, ya que el <*div*> con *id="configuracion"* reservado para el espacio de configuración tiene la clase y subclase *"col s4"* por lo que se rellena el espacio completo, *"s12"*.

Dentro de *"edit"* se incluye una etiqueta <*textarea*> a la cual se le ha dado el identificador *"ensamblador",* con el que indicarle a la librería CodeMirror que ese es el espacio de texto

donde se quiere que se cargue su editor de código. Además, en el fichero app.js se inicializa una variable CodeMirror [CM|20] y se le indica que el <*textarea*> sobre el que tiene que actuar es aquél con dicho identificador. Se selecciona el tema de visualización que tiene que cargar y se activa la opción de que muestre los números de línea:

```
var mirror = CodeMirror.fromTextArea(
      document.getElementById("ensamblador"), {
      theme: "duotone-light",
      lineNumbers: true,
});
```

Se establece el tamaño del editor de CodeMirror, a modo de filas y columnas, como si de un <*textarea*> común se tratase:

```
mirror.setSize(700, 950);
mirror.setValue(fichero_default);
mirror.save();
```

Con setValue(string) se le indica cuál va a ser el valor de inicio a mostrar, en este caso el código "daxpy", visible en la Figura 16, guardado en la variable documento_default proporcionado por los docentes como prueba por defecto y por último, con save() se actualiza el texto interior del <*textarea*> "ensamblador".

```
    .data
    ; Vector x
x:  .double 1, 2, 3, 4, 5, 6, 7, 8
    ; Vector y
y:  .double 100, 100, 100, 100, 100, 100, 100, 100
    ; Vector z
z:  .space 64
fin_z:
a:  .double 2
    .text
start:
dadd r1, r0, x
dadd r2, r0, y
dadd r3, r0, z
dadd r4, r0, fin_z
l.d f0, a(r0)
loop:
l.d f1, 0(r1); Load X
l.d f2, 0(r2); Load Y
mul.d f3, f1, f0; a * X
add.d f4, f2, f3; a * X + Y
s.d f4, 0(r3); Store Z
dadd r1, r1, 8
dadd r2, r2, 8
dadd r3, r3, 8
dsub r5, r4, r3
bnez r5, loop
end:
trap 0

```

Figura 16: El editor CodeMirror con el código "daxpy"

En la Figura 17 se observa el resultado del diseño e implementación de la interfaz de entrada a la aplicación, donde se puede escoger configuraciones, cargar código y editarlo.

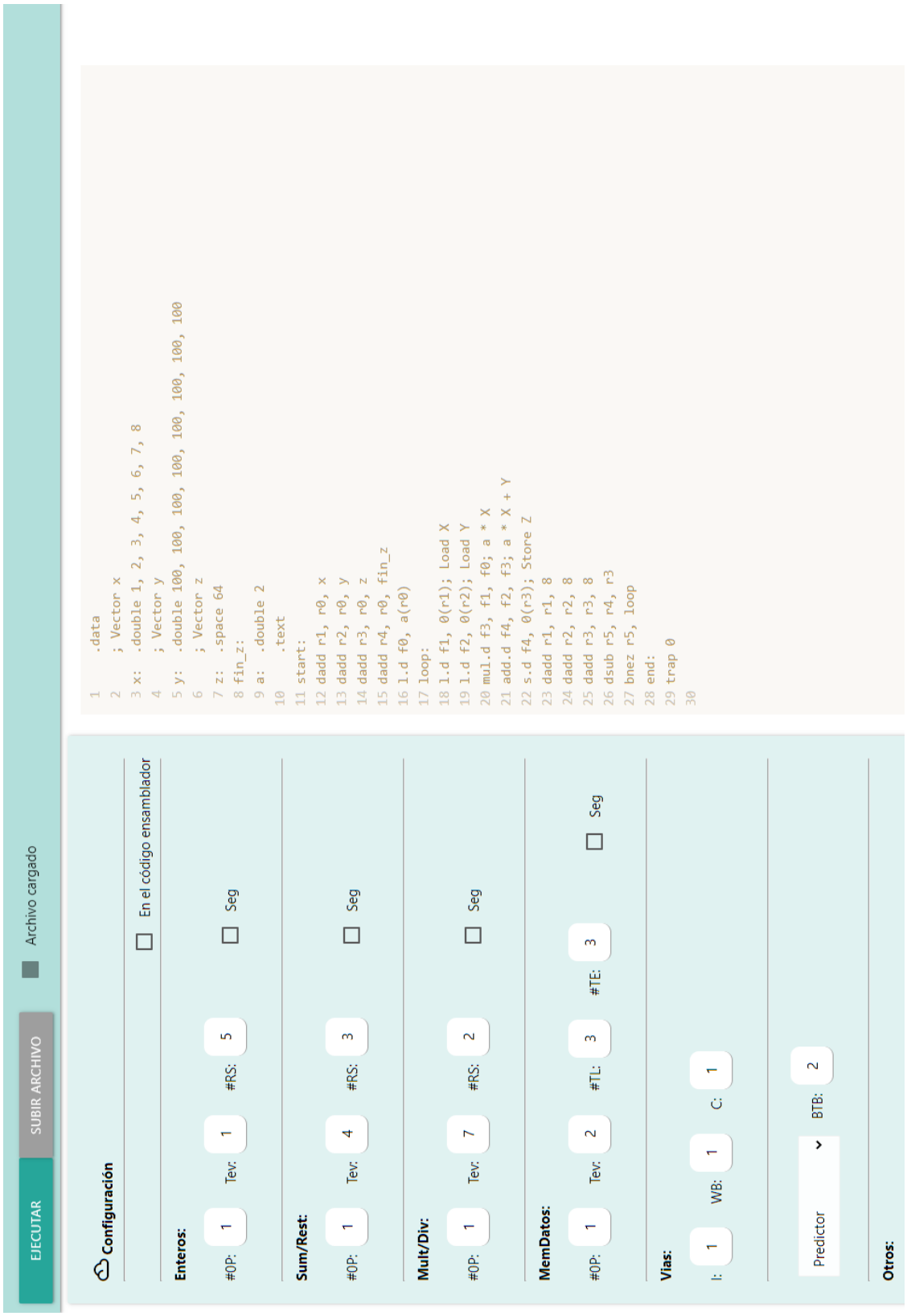


Figura 17: La interfaz de carga y ejecución de archivos, con las configuraciones y el editor (volteado)

4.4) Interfaz de visualización de datos

Esta interfaz sólo será accesible una vez hemos cargado y ejecutado un programa y estará dividida también en dos bloques:

Una barra de navegación mediante la cual el usuario podrá realizar diferentes acciones como avanzar de ciclo, cambiar el tamaño del cronograma en ciclos y volver, si lo desea, a la interfaz inicial.

Por otra parte, bajo la barra de navegación, se dispondrá un espacio donde visualizar los datos según el usuario vaya variando de ciclo. Este espacio se actualizará cada vez que el usuario realice alguna acción, y deberá mostrar siempre los datos de simulación correctos.

En resumen, la interfaz se divide en dos secciones:

- ➔ La barra de navegación
- ➔ Espacio para las tablas y el cronograma

4.4.1) La barra de navegación

Esta barra de navegación, visible en la Figura 18, será algo más compleja que la explicada en el apartado 4.3, ya que contendrá más acciones posibles.

En ella, el usuario deberá ser capaz de avanzar o retroceder entre los diferentes ciclos de la ejecución del simulador, para poder visualizar los datos de cada ciclo en concreto. Además, se le va a dar la posibilidad de modificar el tamaño del cronograma para observar más o menos ciclos de los que se proporcionan por defecto. También será posible que el usuario vuelva a la interfaz de configuración y carga de código de la sección anterior mediante el botón “Volver a carga”.

Figura 18: Diseño de la barra de navegación de visualización de datos

Su metodología de construcción es parecida a la anterior, se trata de una barra de navegación fija, para que sus menús queden siempre visibles:

```
<div class='navbar-fixed' id='navApp' style='display: none;'>
<nav class="nav-extended teal lighten-4">
      <div class="nav-wrapper">
      <div class="nav-content row">
            <div class="col s12"></div>
      </div></div>
</nav></div>
```

Se mantiene el <*div*> externo con la clase *"navbar-fixed"*. Lo único que ha cambiado es que desde el inicio esta barra de navegación no será visible debido a su atributo *style="display : none"*. Sólo cuando pulsemos ejecutar y obtengamos el resultado del simulador será cuando la barra y la interfaz de carga se ocultarán y se mostrará la barra de navegación de visualización de datos, junto a la interfaz reservada para las tablas y el cronograma, debido a que se producirá un cambio de los valores de sus atributos *style* que pasarán a ser *style='display : block'*.

Se ha mantenido la misma subclase de color y brillo para mantener un aspecto homogéneo en ambas barras de navegación de la aplicación, así como la manera de estructurar el contenido como una fila que pasa a ser una columna de tamaño s12.

Las diferencias de esta barra respecto a la anterior serán, esencialmente, los menús que contienen.

Dentro de la etiqueta interior <*div class="col s12"*> se va a introducir una lista desordenada <*ul*> como en el apartado anterior, solo que esta vez se añaden varios botones que permitirán, más adelante, navegar entre los diferentes ciclos a mostrar del simulador:

```
<a class="waves-effect grey btn col s1" id="menosDiez">-10</a>
<a class="waves-effect grey btn col s1" id="menosCinco">-5</a>
<a class="waves-effect grey btn col s1" id="menosUno">-1</a>
<a class="waves-effect grey btn col s1" id="masUno">+1</a>
<a class="waves-effect grey btn col s1" id="masCinco">+5</a>
<a class="waves-effect grey btn col s1" id="masDiez">+10</a>
```

Se han mantenido las mismas clases de animación *"waves-effect"* y la subclase *"btn"* para indicarle a Materialize CSS que el elemento en cuestión debe ser un botón, además, se ha indicado mediante las subclases *"col s1"* que el tamaño de cada botón va a ser el mínimo, es decir, *s1*.

Por último, a cada botón se le ha añadido un identificador diferente de modo que cuando el usuario clique alguno de ellos, se pueda manejar el evento correspondiente independientemente del resto.

En el texto interior del botón se indica la cantidad en ciclos en que se podrá desplazar hacia atrás o hacia adelante para visualizar cualquier ciclo concreto.

Seguidamente se ha introducido un párrafo *<p>* con tamaño *s1*, dentro del cual se le proporciona al usuario la información de en qué ciclo se encuentra.

```
<p class="col s1" id="actual" style='color:black'></p>
```

El siguiente elemento añade un botón que al ser pulsado lanza un evento que dispara una función para cambiar el tamaño del cronograma:

```
<a class="waves-effect grey btn col s1" id="tamCrono">Tamaño Crono</a>
```

A continuación, se coloca un *<input>* de tamaño *s1* en el cual el usuario introducirá el número del tamaño deseado (en ciclos) para el cronograma:

```
<a><input class="redondeado col s1" type="text col s1"
id="tamDeseado"></a>
```

Mediante los identificadores que aparecen en los elementos, se trabajará con ellos en JQuery o JavaScript. El funcionamiento de los eventos que dispara cada botón será explicado más adelante en su apartado correspondiente.

Se ha añadido también otro botón, a la derecha del todo de la barra de navegación (*style='float: right'*), con el cual el usuario podrá volver de nuevo a la interfaz de carga de ficheros y configuraciones:

```
<a class="waves-effect btn"
style="float:right;" id="pantalla_carga">Volver a carga</a>
```

4.4.2) Espacio para las tablas y el cronograma

Este espacio, mostrado en la Figura 19, albergará las diferentes tablas de datos disponibles y el cronograma para la visualización. Ocupará el resto del espacio disponible tras colocar la barra de navegación y estará dividido en varias partes.

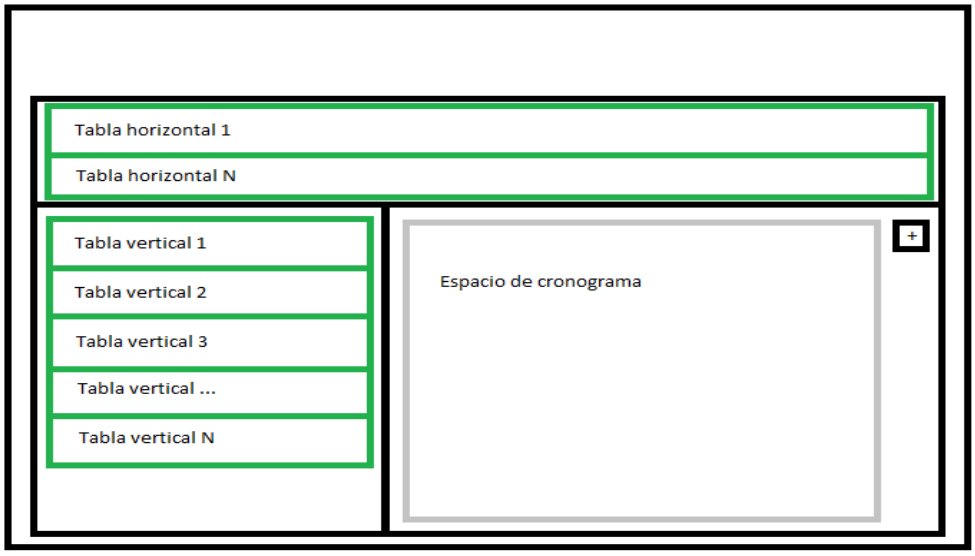


Figura 19: Diseño del espacio de visualización de tablas y cronograma

Su construcción se detalla a continuación:

```
<div id='mainApp' style='display: none;'></div>
```

Este será el elemento <*div*> externo dentro del cual estarán integradas las diferentes partes de la escena de visualización.

Por defecto, el elemento, igual que la barra de navegación de este punto, estarán ocultos al inicio (*style='display : none;'*) para que posteriormente, al cargar un archivo de código, ejecutarlo y obtener un resultado, se mostrarán (*style='display : block'*).

En el interior del <*div id='mainApp'*> se colocan el resto de los elementos necesarios para la composición de la escena.

El primero de ellos será:

```
<div id="horizontales" style="width: 100%; height: 10%;

position: relative; border: 1px solid black"></div>
```

El cual contendrá un elemento desplegable donde se contendrán las tablas cuyo *"layout"* sea horizontal (ver Figura 20).

Se ha decidido que ocupe el 100% de el ancho del espacio de visualización y cuando esté plegado, ocupará un 10% del alto del espacio.

Figura 20: Diseño del contenedor de tablas horizontales

Para las tablas verticales (ver Figura 21), tendremos otro del mismo modo:

```
<div id="verticales" style="width: 30%; overflow-y: scroll; height:
80vh; position: relative; float: left; border: 1px solid black;"></div>
```

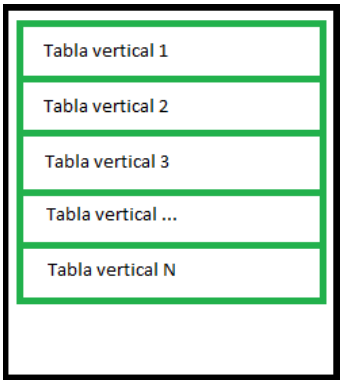


Figura 21: Diseño del contenedor de tablas verticales

En este caso, las tablas compartirán anchura con el cronograma. El * * para las tablas verticales ocupará el 30% del ancho disponible. Estará a la parte izquierda de la visualización y ocupará *80vh*, es decir, un 80% de la altura del viewport. Tendrá scroll vertical (propiedad *overflow-y: scroll*;) para cuando se amplíe el elemento desplegable, el usuario pueda moverse por el elemento y ver los valores de la tabla que le intersen.

El cronograma, ocupará el 70% restante del ancho de la visualización quedando a la derecha del espacio del espacio de las tablas verticales. También se le ha dado una altura de *80vh*. En el interior del * * se han colocado otros dos elementos * *. El primero, con *id= "crono_btns"* irá a la parte derecha de id="cronograma" y será el encargado de contener el botón de maximización del mismo. El segundo, como su identificador indica, *id= "crono_space"* será el encargado de contener la tabla del cronograma. Se le ha dado a este último la propiedad *white-space: nowrap*, para eliminar huecos en blanco y saltos de línea, y se le ha dotado de *overflow* en los dos ejes, para que, dado el caso, aparezcan las barras de scroll vertical y horizontal.

```
<div id="cronograma" style="width: 70%; height: 80vh; float: left;
border: 1px solid black">
            <div id="crono_btns" style="float: right;"></div>
            <div id="crono_space" style="overflow-x: auto;
            overflow-y: auto; height: 79vh; white-space:nowrap;"></div>
</div>
```

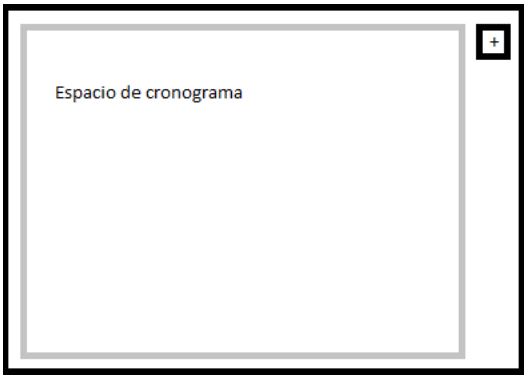


Figura 22: Diseño del contenedor del cronograma

El último elemento definido en *"mainApp"* será el dedicado a la maximización de tablas. Por el momento no será visible. Este elemento servirá como un contenedor para inyectar el contenido de las tablas cuando queramos maximizar la visualización de alguna de ellas. Sólo será visible en ese momento y se mostrará hasta que el usuario clique de nuevo sobre un botón de minimización.

```
<div id="maximizar" style="width: 100%; height: 100% ;display: none;">
            <div id="max_btns" style="float: right;"></div>
            <div id="max_space" style="overflow-x: auto;
            overflow-y: auto; width: 100%; height: 850px"></div>
</div>
```

Cuando el elemento con *id='maximizar'* sea visible, sólo mostrará una tabla y el resto de los elementos interiores del * * se ocultarán. Al maximizar, se tendrán también dos elementos * *: El primero, que irá a la derecha, servirá para albergar los botones de minimización una vez hayamos maximizado una tabla, y el segundo con *id="max_space"* será el que contendrá dicha tabla.

En las figuras de las siguientes páginas, Figura 23 y Figura 24, se observa el resultado final de la implementación de la interfaz completa, junto con la barra de navegación de visualización de datos.

Figura 23: La interfaz de visualización completa (volteado)

Figura 24: El cronograma maximizado (volteado)

4.5) Algoritmos para dotar de funcionalidad a la aplicación

En este apartado se detallarán los algoritmos que van a ser utilizados para que la aplicación tenga la funcionalidad que se espera. Para ello se explicarán tanto las variables utilizadas y de dónde proceden sus valores, como los procedimientos que realiza cada uno de los algoritmos para conseguir su finalidad. Todos los algoritmos aquí comentados, trabajarán en conjunto para hacer realidad el objetivo de que el usuario pueda interactuar con la aplicación y pueda observar en cada momento los datos de los que precise. Cada uno de ellos tendrá una funcionalidad en concreto.

Las funciones de tablas, cronograma y cambios de ciclo trabajarán en base a la información extraída tras la ejecución del simulador, la cual estará en una variable global en formato JSON a la que se ha llamado sim_data.

Dentro de esta variable estará toda la información necesaria, el número de tablas, los diferentes valores de cada una de ellas, así como los cambios que se deberán aplicar a estas en cada ciclo en concreto.

Los campos más importantes de sim_data, serán guardarlos en variables globales por separado, y son:

```
var info = sim_data["info"];
```

Contiene el nombre del archivo ejecutado e información miscelánea.

```
var estructuras = sim_data["structures"];
```

Contiene las tablas e información sobre ellas: identificador, label, tipo de estructura, tamaño de esta y los diferentes campos que contiene.

```
var cMax = sim_data["cycles"].length - 1;
```

En esta variable se guarda el valor numérico del ciclo final.

```
var cMin = 1;
```

El ciclo inicial de cualquier programa ejecutado siempre será el ciclo 1.

```
var cicloActual = cMin;
```

Por tanto, habrá una variable cicloActual la cual indicará en que ciclo se está en cada momento y que al inicio tomará el valor cMin.

```
var ciclos = sim_data["cycles"];
```

Por último, en esta variable estará la información de los cambios que aplicar en cada ciclo a cada tabla.

4.5.1) Algoritmo para la creación de tablas

Es necesario disponer de un algoritmo que sea capaz de crear tablas HTML e indexarlas en el código de modo agnóstico, es decir, siendo independiente su programación a los datos o el contenido de la tabla que se va a crear.

De este modo, se podrá crear las tablas independientemente de su función y, además, será posible que en un futuro el simulador decida mostrar más estructuras de contenido que las que se dispone actualmente. Será posible la creación y disposición de nuevas tablas en la interfaz sin tener que modificar el código del algoritmo.

Para ello, se trabajará acorde a la información que se obtenga de la variable JSON entregada tras la ejecución del simulador, y se construirán las tablas sin tener en cuenta, a la hora de programar, el contenido que estas puedan tener.

Para la creación de este algoritmo, se tendrá en cuenta la siguiente variable procedente de la salida JSON de la ejecución:

```
var estructuras = sim_data["structures"];
```

En ella, en cada posición, se tendrá la información de una tabla y se trabajará en base a la información extraída de los campos que contiene de cada una de ellas.

Cada tabla, en su representación JSON tendrá asociados varios atributos:

- ➔ Id: el indentificador de la tabla.
- ➔ Label: la etiqueta correspondiente al nombre de la tabla.
- ➔ Type: el tipo siempre será tipo “tabla”, con excepción del cronograma.
- ➔ Layout: indicará si se dispondrá de forma vertical u horizontal.
- ➔ Size: el número de filas que tendrá su contenido.
- ➔ Fields: las columnas que tendrá cada tabla con su respectivo encabezamiento.

Según su atributo *layout*, las tablas pueden ser horizontales o verticales. Lo primero que se hará será crear un *bucle for* que recorra las diferentes posiciones de la variable estructuras, extraiga sus características y distinga si se trata de una tabla vertical u horizontal para tratarlas por separado. Con esta finalidad hará falta crear dos variables, tablas_horizontales y tablas_verticales, que serán arrays donde se guardarán las tablas ya tratadas y separadas junto con sus datos característicos. También se crearán dos listas desordenadas <*ul*> HTML, donde se inyectarán las tablas una vez se hayan creado, y se les irá dando su espacio en la aplicación.

```
function tablas(){
	tablas_verticales = [];
	tablas_horizontales = [];
	document.getElementById('verticales').innerHTML = "";
	// Se limpia el HTML
	document.getElementById('horizontales').innerHTML = "";
	var collection_verticales = "<ul>"; //Listas desordenadas
	var collection_horizontales = "<ul>";

	for( i=0; i<estructuras.length; i++){ // Para cada estructura
		var t_id= estructuras[i].id; // ID de la tabla
		var t_label = estructuras[i].label; // Label de la tabla
		var t_layout = estructuras[i].layout;
		// Layout, horizontal o vertical
		var t_size = estructuras[i].size; // Tamaño de la tabla
		var t_fields = estructuras[i].fields;
		// Campos/cabeceras de la tabla

		if(t_layout=="vertical"){
			…
		}else { if(t_id!='crono'){
			// la estructura cronograma se tratará aparte
			…
		}}	}//end for  }
```

Una vez hecho esto ya se puede tratar cada tipo de tabla por separado: las verticales (sentencia *if*) y las horizontales (sentencia *else*) y en una función aparte el cronograma, ya que es un caso de tabla especial.

Dentro del *if* de tablas verticales, se creará una tabla nueva por cada estructura con *layout == "vertical"*:

```
var nuevo_elemento = "<li>" + "<button type='button' class='collaps'>"
      + "<b>" + t_label + "</b></button>"
      + "<div class='content' style='display: none;'>"
      + "<button style='float : right' class='max'>+</button>"
      + "<div id='"+ t_id + "_" + " '></div>"
      + "</div></li>";
```

La lista desordenada collection_verticales será realmente el bloque HTML que contenga todas y cada una de las tablas verticales, y mediante las variables nuevo_elemento que se generen, se rellenará la lista con elementos independientes.

Lo mismo ocurrirá para las tablas horizontales con collection_horizontales.

Cada elemento de la lista tendrá un botón con la clase *"collaps"* que se utilizará para desplegar o guardar el espacio reservado a la tabla según se quiera verla u ocultarla y dentro de este se dispondrá un botón de *clase="max"* que dará la funcionalidad necesaria para llevar el contenido de la tabla al * *. Dicho funcionamiento quedará explicado del todo finalmente en el apartado dedicado a eventos. La tabla, por defecto, irá dentro de un espacio * * cuyo identificador será el de la propia tabla + un guión bajo.

Acto seguido se inicializará una variable t_html en la cual se irá agregando el código HTML de la tabla.

```
var t_html = "<table id='" + t_id + "' class='claseTabla2'>";
```

La tabla tendrá su propio identificador, como una clase CSS creada a propósito para ellas.

Se irá construyendo una tabla vertical vacía. Primero se creará el header:

```
t_html += "<thead>";

for(y=0; y<t_fields.length; y++){

      t_html = t_html + "<th>" + t_fields[y].label + "</th>";     }

t_html += "</thead>";
```

Para cada y en t_fields (cabeceras) añadimos su atributo *label* como texto del elemento *<th>*.

En el cuerpo de la tabla, tendremos en cuenta la columna de encabezamiento en la que estará dispuesta la celda, y el tamaño de la tabla:

Para cada j en el rango de tamaño de la tabla, crearemos un número de celdas t_fields.length, que irán en un mismo *table row* ,*<tr>*, de la tabla. Cada celda tendrá como id, la id de la columna (cabecera) más "-" y el número de fila j para poder en un futuro cambiar sus valores más cómodamente.

```
for (j = 0; j < t_size; j++) {

      t_html += "<tr>";

      for (z = 0; z < t_fields.length; z++) {

            t_html = t_html +

            "<td id='" + t_id + "-" + t_fields[z].id + "-" + j + "'>"

            + "</td>";

}

      t_html+= "</tr>"  }
```

Se finaliza el contenido de la variable t_html de la tabla y se añade a su correspondiente *collection*, en este caso collection_verticales. También se guardan los datos de la tabla en el array tablas_verticales para darles uso más adelante. De modo que cada posición de este array contendrá diferentes campos de información sobre cada tabla en cuestión.

```
t_html += "</tbody>";

t_html += "</table>";

tablas_verticales.push([t_id,t_fields,t_html,t_layout,t_label,t_size])

collection_verticales += nuevo_elemento;
```

En el caso del procedimiento para crear las tablas horizontales, excluyendo al cronograma, es parecido, sólo que en este caso se intercambian filas por columnas:

Para cada fila en t_fields.length , se inserta un <tr> con las celdas correspondientes a cada columna t_size, incluyendo en el identificador la id de la cabecera + "-" + el número de columna z.

```
for (j = 0; j < t_fields.length; j++) {

      t_html += "<tr>";

      for (z = 0; z < t_size; z++) {

            t_html = t_html +

            "<td id='" + t_id + "-" + t_fields[j].id + "-" + z + "'>" +
            "</td>";    }

t_html += "</tr>"

}
```

Y finalmente se realiza el mismo procedimiento que con las tablas verticales:

```
t_html += "</tbody>";

t_html += "</table>";

tablas_horizontales.push([t_id,t_fields,t_html,t_layout,t_label,
t_size]);

collection_horizontales += nuevo_elemento;
```

Una vez ha terminado el bucle for(i=0; i<estructuras.length; i++) y ya están creadas y separadas las tablas HTML con sus respectivas filas y columnas, se finalizan ambas listas desordenadas *collection* y se asigna a su espacio HTML <*div*> correspondiente, cada bloque de elementos desplegables con tablas:

```
collection_verticales += "</ul>";

collection_horizontales += "</ul>";

document.getElementById('verticales').innerHTML =
collection_verticales;

document.getElementById('horizontales').innerHTML                    =
collection_horizontales;
```

4.5.2) La estructura de datos del cronograma

Se necesita una estructura de datos para separar y albergar cada uno de los elementos a mostrar en el cronograma durante la visualización por parte del usuario. Para ello, se ha decidido implementar la función que se describe en este apartado.

Con esta finalidad, la función cronograma() se encargará de filtrar y ordenar los datos de la variable global: var ciclos = sim_data["cycles"]; obtenida tras la ejecución del simulador.

Para ello se han creado las siguientes variables globales, las cuales se irán rellenando durante la ejecución de la función:

```
var datos_crono = new Array(cMax);

var crono_pc = []

var crono_instruc = [];

var crono_values = new Array(cMax *
document.getElementById("NUM_VIAS_ISSUE_p").value);
```

En las variables crono_pc y crono_instruc se irán guardando por orden de aparición los datos que deberán mostrarse en las columnas "*PC*" e "*Instrucción*" del cronograma, que también conformarán el eje vertical de este. Datos_crono será una variable auxiliar para filtrar los datos.

Por otro lado, al final de la ejecución de la función, crono_values será un array de arrays, en el cual en cada posición *"x" "y"* almacenará el dato en concreto a mostrar en el cronograma para cada *pc/instrucción* "x" y para cada ciclo "y" en el momento que deban aparecer. Se ha decido que para las posición *[0,0]* crono_values esté vacía para indexar a partir de la posición 1, así, la primera posición real coincide con el número de ciclo inicial cMin = 1. El largo de crono_values, vendrá dado por cMax y por el *número de vías issue* que el usuario haya decidido en la etapa de configuración, que será el valor introducido en el elemento con *id= "NUM_VIAS_ISSUE_P"*.

Al comienzo, la función cronograma() leerá cada dato de la variable ciclos, y si se trata de un dato perteneciente al cronograma lo filtrará en datos_crono de la siguiente manera:

```
for(i = 0; i< ciclos.length; i++){

datos_crono[i] = new Array(); // crea un nuevo array en la posición i

      for(j=0; j< ciclos[i].values.length; j++){

            // Si el valor j del ciclo i, forma parte del cronograma,
            lo indexa en datos_crono[i]

            if (ciclos[i].values[j].id.startsWith('crono')) {
            datos_crono[i].push(ciclos[i].values[j])       }

      }

}
```

Seguidamente, lo que se hace es transformar crono_values en un array de arrays. Esta será la estructura de datos que se utilizará para almacenar en las posiciones que correspondan, cada uno de los datos:

```
for(c=1; c<crono_values.length; c++){

      crono_values[c] = new Array(cMax);

      }
```

Y después se recorre datos_crono extrayendo del identificador de cada dato la posición que le corresponde en crono_pc, crono_instruc y crono_values:

```
for (i = 0; i < datos_crono.length; i++) {

      for(j=0; j<datos_crono[i].length; j++){

            c_v = datos_crono[i][j];


            if(c_v.id.startsWith("crono-pc-")) {

                  // si se trata de un dato con id "crono-pc"

                  pos = parseInt(c_v.id.replace('crono-pc-', ''))

                  crono_pc[pos] = c_v.value;

                  //se inserta en la posición dada por el identificador
```

```
        }else if(c_v.id.startsWith("crono-instruc-")){
        // si se trata de un dato con id "crono-instruc"

              pos = parseInt(c_v.id.replace('crono-instruc-', ''))
              crono_instruc[pos] = c_v.value;
              // se inserta en la posición dada por el identificador

        }else {
              eje = c_v.id.split("-");
              // se calcula el eje "x, y" dado por el identificador
              // y se inserta en la estructura
              y = parseInt(eje[1]);
              x = parseInt(eje[2]);
              crono_values[x][y] = c_v.value;}
        }
}
```

Así, indexando como *[pos]* y el eje *[x][y],* donde *"x"* será la instrucción e *"y"* el ciclo, se coloca cada dato en la posición que le corresponde en su estructura correspondiente. De este modo, cuando se represente el cronograma, será mucho más sencillo disponer los datos en sus celdas.

4.5.3) Algoritmos necesarios para cambiar los datos

Una vez ya están decididos los algoritmos para la creación de las tablas e implementada la estructura de datos del cronograma, es el momento de detallar los algoritmos que se van a utilizar para actualizar los datos a mostrar cuando se cambia de ciclo de simulación. Nótese que estos datos puede que tengan asociado un atributo de clase CSS, para cambiar su visualización en las celdas.

Para el caso de las tablas, hay cuatro algoritmos, dependiendo del ciclo y de la dirección en que el usuario se quiera mover.

- ➔ rellena_inicio() coloca en las tablas los datos iniciales y hace una llamada dibujaCrono(cMin) para inicializar y rellenar el cronograma con los datos de inicio.

- ➔ hacia_adelante(nuevoCiclo) se encargará de, dado un ciclo N, colocar los datos de dicho ciclo en las tablas. También actualizará las clases CSS de las celdas en caso de ser necesario.

- ➔ calculaPrev(cicloAct) se encargará de guardar en una variable global aquellos datos que en un ciclo determinado tienen un campo de valor previo, de modo que, al viajar a ciclos anteriores, se puedan recuperar los valores previos al ciclo en el que se esté.

- ➔ hacia_atras(nuevoCiclo) se ocupará de, dado un ciclo N, colocar los datos de dicho ciclo, teniendo en cuenta los valores previos registrados por la función calculaPrev(). También actualizará las clases CSS de las celdas en caso de ser necesario.

En el caso del cronograma, una vez creada e indexados los valores en crono_values, crono_pc y crono_instruc mediante la función cronograma(), sólo habrá que llamar a la función dibujaCrono(cicloAct), donde cicloAct será el ciclo cuyos valores se quieran mostrar.

4.5.3.1) El algoritmo rellena_inicio()

El propósito de este algoritmo es rellenar los valores iniciales (ciclo 1) de las tablas de manera que estas no estén vacías al inicio e incrustarlas en su espacio HTML correspondiente. Al final de él, se realiza una llamada a la función dibuja_crono(cMin) para que el cronograma muestre también sus valores de inicio.

- ➔ La función utiliza valores tomados de la variable global antes detallada, ciclos = sim_data["cycles"]; concretamente los valores de la posición inicial, por ello, se ha decidido separarlos dentro de la función como la variable cicloInicio = sim_data["cycles"][0].values;

➔ También se nutre de la información recogida por la función tablas(), en los vectores tablas_verticales y tablas_horizontales, cuya longitud será el número de tablas de cada tipo que se tiene ahora mismo en la aplicación.

De este modo, el algoritmo realizará cálculos para cada tabla horizontal y vertical:

```
function rellenaInicio(){

      var cicloInicio = sim_data["cycles"][0].values;

      for (i = 0; i < tablas_verticales.length; i++) { ...}

      for (i = 0; i < tablas_horizontales.length; i++) {... }
      }
```

Lo primero que hará será sacar algunos valores de cada posición i de los vectores. Si recordamos, en la función tablas(), se habían guardado en cada posición los siguientes datos:

```
tablas_horizontales.push([t_id, t_fields, t_html, t_layout, t_label,
t_size]);
```

Así, se toman las posiciones *[0]* y *[2]* para cada elemento, donde tenemos el identificador, y la tabla HTML creada para la estructura i en concreto.

```
var t_id = tablas_verticales[i][0];

var t_data = tablas_verticales[i][2];
```

También, volviendo al algoritmo tablas(), al crear cada uno de los elementos de la lista desordenada donde se debían mostrar, se había creado un elemento <*div*> con el propio identificador de la tabla + un guion bajo, se aprovecha esto para insertar la tabla HTML en dicho elemento:

```
if (document.getElementById(t_id +"_") != undefined) {

      document.getElementById(t_id+ "_").innerHTML = t_data;

}
```

Llegado a este punto ya se tiene colocada la tabla en el código HTML y sólo queda modificar los valores de cada celda por los iniciales, teniendo en cuenta que cada celda tendrá su propio identificador.

Los datos extraídos del array cicloInicio pueden tener varios campos, pero aquellos que no variarán será su identificador, *"id"*, indicando en que columna de que tabla tiene que estar dicho dato y su campo *"value"* o valor de este. Opcionalmente, pueden tener un campo *"class"* dentro de *"value"* el cual indicará que clase CSS hay que aplicar a la celda a la hora de mostrarla en la tabla. Así pues, habrá que recorrer en un bucle cada dato de cicloInicio, extrayendo de cada uno sus atributos *"id"*, *"value"* y *"class":*

```
for(z=0; z <cicloInicio.length; z++){
      dato_id = cicloInicio[z].id;
      dato_valor = cicloInicio[z].value;
      dato_class = cicloInicio[z].value.class;
      ….    }
```

Dentro de este mismo bucle se realizarán diferentes comprobaciones para ver si el "id" del dato coincide con alguno de los identificadores de la tabla:

```
if(document.getElementById(dato_id) != undefined){
      // si alguna celda coincide con id
      if (typeof(dato_valor) == 'object'){ // si el dato es un objeto
            if(Array.isArray(dato_valor)){ // y es un array
            document.getElementById(dato_id).innerHTML =
            dato_valor[0].text;
                  // se coloca el primer elemento del array
      } else {
      // en caso de no sea un array pero siga siendo un objeto
            //se coloca el dato.text en la celda
                  document.getElementById(dato_id).innerHTML =
                  dato_valor.text;

            // adicionalmente, si la clase del dato es compuesta
```

```
            if (dato_valor.class.includes(" ")) {
            // se obtiene un array con las diferentes clases CSS
                  var clases = dato_valor.class.split(" ");
                  for (c = 0; c < clases.length; c++) {
                   // y se aplica a la celda en cuestión
                   document.getElementById(dato_id)
                   .classList.add(clases[c]); }

            // en caso de que no sea compuesta,
            // se aplica la única clase a la celda
            } else {
            document.getElementById(dato_id)
            .setAttribute('class',dato_valor.class); }
            }
      // por último, en caso de que el dato no sea un objeto o un
      //array, se aplica solamente el dato a la celda
} else { document.getElementById(dato_id).innerHTML = dato_valor; }
} // fin if dato_id != undefined
```

De este modo, se coloca cada dato inicial disponible en cada celda cuyos identificadores coincidan. Los identificadores de las celdas se habían calculado previamente en la función tablas().

En el caso de las tablas horizontales, se tiene un bucle

for(i = 0; i < tablas_horizontales.length; i++) cuyo contenido será idéntico.

Al final de la ejecución de los bucles, se realiza una llamada dibujaCrono(cMin) para rellenar el espacio del cronograma.

4.5.3.2) El algoritmo hacia_adelante()

Este algoritmo tiene como propósito que, dado un ciclo N, cambiar los datos de las tablas de las celdas ya creadas, sean los que sean, por los datos obtenidos por el simulador para ese ciclo en concreto. Su funcionamiento es parecido al algoritmo anterior, pero este, recibe como parámetro el ciclo cuyos datos se quieren mostrar. De este modo hacia_adelante(ciclo) extraerá los datos del JSON de dicho ciclo y los colocará en la tabla. Para que se reflejen los cambios para un ciclo N correctamente, se deberán realizar llamadas sucesivas a esta función de modo que, si se está en el ciclo 1 y se quiere mostrar el ciclo N+5, habrá que llamar a la función tantas veces como ciclos se quiera avanzar. De estas ejecuciones se encargará la función de control de ciclo de la aplicación.

Así pues, la función hacia_adelante(nuevoCiclo), tendrá como parámetro el ciclo cuyos datos se quieran aplicar a la tabla. Se extraen esos datos como la variable:

```
var cicloN = sim_data["cycles"][nuevoCiclo].values;
```

Como en el anterior algoritmo, se recorre el array de datos del ciclo, extrayendo sus atributos:

```
for(i =0; i<cicloN.length; i++){
      dato_id = cicloN[i].id;
      dato_value = cicloN[i].value;
      dato_class = cicloN[i].class;
      …. }
```

Seguidamente se comprueba si el identificador del dato aparece en alguna tabla, y si lo hace, se realizan las comprobaciones de tipo de dato y se comprueba si tiene clase CSS asociada. En caso de tenerla, se aplica la clase, como en el algoritmo anterior.

4.5.3.3) El algoritmo calculaPrev()

Previamente a detallar el algoritmo hacia_atras(), hay que documentar otro algoritmo cuya funcionalidad se aprovechará en hacia_atras(). Este es el algoritmo calculaPrev(cicloAct), el cual dado un ciclo de entrada N, recogerá en una variable global

aquellos datos que en el ciclo N tienen un campo de valor previo, es decir, el campo que permite viajar desde ese mismo ciclo al N-1 y que permitirá, una vez estén esos datos guardados, viajar con hacia_atras() desde cualquier ciclo a su inmediato anterior, mostrando correctamente los datos en las tablas.

La variable global que esta función actualizará será la variable:

```
var prevs = new Array(cMax);
```

donde en cada posición i se almacenarán los valores previos que tiene ese mismo ciclo para ir a su predecesor.

El algoritmo buscará dado un ciclo N, aquellos valores los cuales tienen algún campo para dato previo:

```
var cicloN = sim_data["cycles"][cicloAct].values;
```

Y se recorren los valores de dicho ciclo y si tiene campo previo se guarda en un array llamado previos:

```
 var previos = [];
```

Por último, se indexarán en prevs[cicloActual], aquellos valores recogidos en previos:

```
function calculaPrev(cicloAct){
      var cicloN = sim_data["cycles"][cicloAct].values;
      var previos = [];
      for(i=0; i< cicloN.length; i++){
            dato_i = cicloN[i];
            if(dato_i.prev != undefined){
            // si el dato actual tiene un campo .prev
                  previos.push(dato_i); //se indexa en previos    }
                              }
            prevs[cicloActual] = previos;       }
```

Este algoritmo se llamará cada vez que se realice un avance de ciclo hacia adelante, para que, de este modo, cuando el usuario quiera volver hacia atrás, los datos previos ya estén calculados y solo haya que aplicar cambios.

4.5.3.4) El algoritmo hacia_atras()

Este algoritmo se encargará de, dado un ciclo N, hacer los cambios necesarios para volver hacia atrás. Es decir, una llamada hacia_atras(cicloAct), eliminará de las tablas los valores que se han introducido en el ciclo N y en el caso de que para ese ciclo N haya valores almacenados en la variable previos, los colocará en las tablas, dejando sus celdas como si se tratasen de las del ciclo N-1. Para llegar a ciclos menores a de N-1, se realizarán sucesivas llamadas en la función de control de ciclo.

Para ello, se utilizan dos vectores, colocadosAct y previos. En el primero se almacenan los datos que ya se han colocado en las tablas para el ciclo N para, posteriormente, eliminarlos. En el segundo de ellos, se tendrán los valores de la posición de la variable global prevs para el ciclo N, que serán aquellos datos colocados que tenían un valor previo a este ciclo. En base a estos datos se realizarán diferentes comprobaciones para decidir si hay que eliminar y cambiar algún dato de las tablas.

```
var previos = prevs[cicloActual];

var colocadosAct = ciclos[cicloAct].values;
```

Lo primero que se hará es eliminar aquellos datos que se han colocado en el ciclo que recibimos como parámetro, para ello se recorrerán los datos en colocadosAct eliminando aquellos datos de las tablas que se han colocado en dicho ciclo:

```
for(i=0; i< colocadosAct.length; i++){ // para cada dato

      if(!colocadosAct[i].id.startsWith("crono")){

      // si no es un dato del cronograma

            if (document.getElementById(colocadosAct[i].id)){ /

            // si existe la celda
            document.getElementById(colocadosAct[i].id).innerHTML = "";

            // se elimina el dato de la celda
```

```
            document.getElementById(colocadosAct[i].id)
            .removeAttribute("class");
            // se elimina su clase css    }     }     }
```

Una vez eliminados los datos colocados en el ciclo N, se recorrerá el array de valores previos, se comprobará su tipo de valor y se añadirá la clase CSS asociada a dicho valor previo:

```
for (i = 0; i < previos.length; i++) { // para cada dato en previos
      dato_i = previos[i]; // el dato previo en cuestión
      dato_id = previos[i].id; // su identificador
      dato_prev = previos[i].prev; // su valor
      dato_class = previos[i].prev.class; // su clase css asociada

      if (document.getElementById(dato_id) != undefined) {
            // si la celda con el id existe
            if (Array.isArray(dato_prev)) {
                  // si se trata de un dato de tipo array
                  document.getElementById(dato_id).innerHTML =
                  dato_prev[0].text;
                  //se coloca el primer elemento

            }else if(dato_prev.text != undefined){ // si es un objeto
            // si es un dato objeto con campo de valor.text
            //se coloca el nuevo dato en la celda en cuestión

            document.getElementById(dato_id).innerHTML =
            dato_prev.text;
                  if (previos[i].prev.class){
                  // si el dato tiene clase css asociada
                  // se elimina la clase actual y se reemplaza por la
                  clase del previo
```

```
                    document.getElementById(dato_id)
                    .removeAttribute('class');
                    document.getElementById(dato_id)
                    .setAttribute('class', previos[i].prev.class);
                    }else{
                    //en caso de que no tenga
                    // se elimina la clase css de la celda.
                    document.getElementById(dato_id)
                    .removeAttribute('class');


      } else { // en caso de que el dato no sea un objeto
            //se coloca el dato
            document.getElementById(dato_id).innerHTML = dato_prev;
            // y se realizan las comprobaciones de clase css
            if (previos[i].class) {
            document.getElementById(dato_id).removeAttribute('class');
            document.getElementById(dato_id)
            .setAttribute('class', previos[i].class);
            }else {
                  document.getElementById(dato_id)
                  .removeAttribute('class');
                  }
            }
} // end if dato_id  != undefined
} // end for previos
```

4.5.3.5) El algoritmo dibujaCrono()

Este algoritmo será el encargado de, una vez rellenada la estructura de datos del cronograma, es decir los arrays crono_pc, crono_instruc y crono_values mediante la función cronograma(), crear y rellenar la tabla dedicada al cronograma para un ciclo en concreto N, que recibirá como parámetro en su llamada: dibujaCrono(cicloAct).

Dependiendo del tamaño en número de columnas (ciclos) con el que se le haya dotado, la tabla cronograma mostrará más o menos información, para ello se ha creado una variable llamada criba_abajo, la cual calculará a partir de qué número en ciclos se mostrará en la tabla, del modo:

```
criba_abajo = cicloAct - num_columnas;
```

Donde cicloAct será el ciclo N que recibe como parámetro la función y num_columnas el número de columnas por defecto a mostrar, inicializado al inicio a 25, pero pudiendo cambiar su valor más adelante mediante la función cambiaTam(nuevoTam) que se explica en el apartado de algoritmos adicionales y manejo de eventos. De este modo, se mostrarán los ciclos desde criba_abajo +1 hasta cicloAct.

Se crearán en la función dos variables adicionales, las cuales almacenarán el valor HTML del cronograma, separando la cabecera <*thead*> y el cuerpo <*tbody*> de la tabla para procesarlas por separado y luego unirlas:

```
crono_html_head = "<table id='cronoTable' style='table-layout: fixed;
border-collapse: collapse;'>
<thead><td class='instruc-td' style='width: 200px; padding: 0px;'>
<div style='float: left; width: 50px'>PC</div>
<div style='left: 50px; width: 200px'>Instrucciones</div></td>";
//se inicia la cabecera de la tabla
crono_html_body = "<tbody id='cronoBody'>";
//se inicia el cuerpo de la tabla
```

En crono_html_head también se indican algunas de las opciones de estilo CSS para la tabla: La propiedad *table-layout: fixed* indica un tamaño fijo para las celdas de las tablas, de modo que, indicando el tamaño de la primera fila, se mantendrá el tamaño para el resto. Con *border-collapse: collapse* se indica que los bordes de la tabla permanecerán unidos. Las dos primeras celdas de la cabecera se han incluido en un único <td> con la clase CSS "*instruc-td*". En este elemento se incluyen dos elementos <*div*>, uno para la

celda “PC” colocado a la izquierda, con un ancho de 50px y otro para la celda “Instrucciones”, que deja esos 50 px a la izquierda de la celda anterior y ocupará 200 px de ancho.

Una vez hecho esto, se crearán las celdas de la cabecera de la tabla, para cada ciclo c desde cMin hasta cicloAct teniendo en cuenta que si c es menor o igual que el valor de criba_abajo, dicho ciclo no se mostrará. Estas celdas tendrán su tamaño fijado a 50px de ancho:

```
for(c=cMin; c<= cicloAct; c++){

      if (c > criba_abajo) { // si c mayor que criba_abajo

                  crono_html_head += '<th style="width: 50px; padding:
                  0px" class="ciclo">' + c + '</th>';

                  // se añade una cabecera para c

                  }

}

crono_html_head += "</thead>"; // se finaliza la cabecera de la tabla
```

Para la construcción del cuerpo del cronograma, se tendrá en cuenta que cada elemento en crono_pc simboliza una fila y cada fila tendrá su campo *PC*, su campo de *instrucción* y tantos elementos como valores disponibles haya para ese ciclo y esa instrucción en crono_values. Teniendo en cuenta también que solo se van a mostrar los ciclos desde criba_abajo+1 hasta cicloAct, si una fila no tiene datos a mostrar en ese intervalo, no se mostrará. Para ello, se ha utilizado una variable de tipo booleano llamada encontrado, cuyo valor se inicializará a *falso* y sólo se cambiará a *verdadero* si hay algún dato disponible a mostrar.

Así pues, se crea un bucle externo que recorrerá el número de filas a mostrar y uno interno que hará lo propio con el número de columnas. La variable para filas crono_html_tr se irá rellenando conforme la iteración de estos bucles, que finalmente se decidirá si incluir en la tabla si se ha encontrado, como mínimo, un elemento a colocar:

```
for(x=1; x<crono_pc.length; x++){ // para cada fila posible x

      var encontrado = false;

      // al inicio de iteración no se ha encontrado ningún dato

      crono_html_tr = '';

      crono_html_tr += '<tr>'; // se crea el elemento fila actual
```

```
if (crono_instruc[x] && crono_pc[x] !== undefined){

      //Si existe valor de pc e instrucc para x

            crono_html_tr += "<th class='pc-td'

                        style='padding: 0px;' >" +

                        crono_pc[x] + "</th>";


            crono_html_tr += "<th class='" +
            crono_instruc[x].class +

            "'style='white-space: nowrap; padding: 0px'>" +
            crono_instruc[x].text + "</th>"

            //Se colocan en la fila      }


for(y=1; y <=cicloAct; y++){ // para cada columna y

      if(y> criba_abajo && y <=cMax){

      // si el valor del ciclo "y" es mayor que criba_abajo

      //se tiene en cuenta


      if(crono_values[x][y] !== undefined){

            // si existe valor actual para fila y columna

                  //encontrado será verdadero con sólo encontrar
                  // un valor, la fila se añadirá

                  encontrado = true;


                        if (crono_values[x][y].text !== undefined){

                          //Si se trata de un objeto, se coloca su
                          //valor .text y su clase css

                          crono_html_tr += "<td class='" +
                          crono_values[x][y].class  +

                          "' style='padding: 0px'><b>" +
                          crono_values[x][y].text + "</b></td>";
```

```
                        } else {
                            //Si se nos da el valor directamente:
                        //Se coloca únicamente el valor.
                        crono_html_tr += '<td style="padding: 0px">'
                        + crono_values[x][y] + '</td>
                } else {
                //En caso de que no exista valor para [x][y]
                //se coloca un espacio en blanco
                crono_html_tr += '<td style="padding: 0px;">'
                + ' ' + '</td>';}
        } //end if y > criba_abajo
        } //end for y
        crono_html_tr +='</tr>'; //Se finaliza la fila
        // si se ha encontrado como mínimo un dato para la [x] actual
        // se añade la fila al <tbody>
        if(encontrado){crono_html_body += crono_html_tr;}
} //end for x
```

Después de la ejecución del doble bucle, ya tenemos las filas que contienen algún dato a mostrar colocadas en el cuerpo de la tabla del cronograma.

A partir de aquí ya sólo queda concatenar crono_html_head y crono_html_body para tener el cronograma ya construido para el ciclo N dado e insertarlo en el elemento <*div*> con *id=”crono_space”* para su visualización, o bien, en caso de que estemos en el modo maximizado de las tablas, colocarlo en el elemento con *id=”max_space”*.

- ➔ En el caso de insertarlo en *id=”crono_space”*, se colocará junto al cronograma, un botón de maximización en el espacio con *id=”crono_btns”* para poder acceder al modo maximizado de la interfaz.
- ➔ En el caso de colocarlo en *id=”max_space”*, se colocará un botón de minimización en el espacio con *id=”max_btns”* para volver a la interfaz original.

El botón de maximización se ha guardado en la variable crono_html_button, para poder incluirlo o no, según sea el caso:

```
var crono_html_button = "<button style='float : right;' id='maxcrono'
class='max2'>+</button>";
```

En cuanto al botón de minimización, se tendrá en la variable:

```
var min_crono_btn = "<button style='float : right;' id='mincrono'
class='max2'> - </button>"
```

El modo en el que se muestre (maximizado o normal) vendrá dado por las variables globales booleanas max_tablas y max_crono, que, por defecto, desde el inicio estarán inicializadas a *"false"*.

Dependiendo del valor de estas variables, se colocará el cronograma en un lugar u otro del código HTML siendo tres los posibles casos:

```
if (max_crono == false && max_tablas == false){

      // caso en el cual no se está maximizando nada

      //se coloca el botón y la tabla del cronograma en "crono_btns" y
      // "crono_space"

          document.getElementById('crono_btns').innerHTML =
          crono_html_button;

          document.getElementById('crono_space').innerHTML =
          crono_html_head + crono_html_body;


            $('#maxcrono').click(function () { ...});

            //se le da funcionalidad al botón de maximización


} else if (max_tablas== false && max_crono == true ){

      // caso en el cual se está maximizando el cronograma

      //Se coloca el botón de minimizado y el cronograma en "max_btns"
      y "max_space"

            document.getElementById('max_btns').innerHTML =
            min_crono_btn;

            document.getElementById('max_space').innerHTML =
            crono_html_head + crono_html_body;
```

```
            $('#mincrono').click(function () { … })

            // Se le da funcionalidad al botón de minimización



} else { //cualquier otro caso

//Se coloca el botón y el cronograma en "crono_btns" y "crono_space"

          document.getElementById('crono_btns').innerHTML =
          crono_html_button;

          document.getElementById('crono_space').innerHTML =
          crono_html_head + crono_html_body;


                  $('#maxcrono').click(function () { ... });

                  //se le da funcionalidad al botón de maximización

}
```

Una vez colocado el cronograma en el lugar que le corresponde, se establece el scroll del espacio de la tabla lo mas a la derecha y hacia abajo posible, para que siempre se muestre el último dato indexado:

```
   //se colocal el valor del scrollTop y scrollLeft

   //en el valor final de scroll

    var scrollH = document.getElementById('crono_space').scrollHeight;

    document.getElementById('crono_space').scrollTop = scrollH;

    var scrollW = document.getElementById('crono_space').scrollWidth;

    document.getElementById('crono_space').scrollLeft = scrollW;
```

En el caso de que estemos maximizando el cronograma, se colocan los scrolls horizontal y vertical para que apunten lo más hacia abajo a la derecha posible:

```
if(max_crono){
  var scrollH = document.getElementById('max_space').scrollHeight;
  document.getElementById('maximizar').scrollTop = scrollH;
  var scrollW = document.getElementById('max_space').scrollWidth;
  document.getElementById('maximizar').scrollLeft = scrollW; }
```

4.5.4) El algoritmo de control de ciclo

Una vez ya definidas las funciones para mostrar y cambiar los datos en las tablas y el cronograma, habrá que establecer un mecanismo para que el usuario pueda cambiar el ciclo a mostrar y con ello, se ejecuten las funciones anteriormente detalladas para visualizar los datos.

Para este fin, se ha implementado la función cambiaCiclo(cicloAct , adyacente). Como se puede observar, en su llamada recibirá dos valores como parámetros: el primero corresponde al ciclo actual que se está mostrando y el segundo, adyacente, será un número positivo o negativo que nos indicará la distancia en ciclos que se quiere recorrer respecto del actual. Con estos valores, la función se encargará de calcular el ciclo que se quiere mostrar, e irá recorriendo de manera ascendente o descendente los mismos hasta llegar al ciclo objetivo.

Para ello, se calculará la suma de los dos valores recibidos. Este cálculo nos dará el nuevo ciclo, nc, al que hay que dirigirse:

```
var nc = cicloAct + (adyacente);
```

Como los valores permitidos en adyacente pueden ser positivos o negativos, se dan varios casos, y dentro de ellos algunos subcasos.

Los casos principales serán:

- ➔ La suma nc es menor o igual a cero.
- ➔ La suma nc es mayor que cero.

En el primero de los casos, cuando nc <= 0, como no es posible mostrar un ciclo igual a cero o menor, se regresará a cMin.

```
} else {// la suma es menor o igual a cero
                for (cicloActual; cicloActual > cMin; cicloActual--) {
                  //Mientras cicloActual > cMin
                     haciaAtras(cicloActual);
                  //Se cambian los datos del cicloActual
                }
                dibujaCrono(cicloActual);
                  // Se dibuja el cronograma para cMin
                visualizaCiclo(cicloActual);
                  // Se muestra el ciclo al usuario   }
```

En el segundo de los casos, cuando nc > 0, pueden darse varios subcasos, dependiendo del valor de nc:

- Si nc es mayor que cicloActual y es menor o igual que cMax:

En este caso, se recorrerán los ciclos desde cicloActual hasta el ciclo objetivo, modificando los valores de las tablas para cada uno de los ciclos intermedios y actualizando cicloActual:

```
if (nc > cicloAct && nc <= cMax) {
      for (cicloActual; cicloActual < nc; cicloActual++){
            calculaPrev(cicloActual);
            // Se calculan los valores previos para cada ciclo
            haciaAdelante(cicloActual);
      // Se modifican los datos de las tablas para cada ciclo    }
      // cuando cicloActual == nc:
      calculaPrev(cicloActual);
      // se calculan los datos previos del ciclo objetivo
      haciaAdelante(cicloActual); // se colocan los datos del ciclo
      dibujaCrono(cicloActual) // se modifica el cronograma
      visualizaCiclo(cicloActual); //se muestra el ciclo al usuario
      }
```

- Si nc es menor que el cicloActual:

En este caso estamos retrocediendo, y se volverá desde cicloActual hasta nc.

```
else if (nc < cicloAct) {
      for (cicloActual; cicloActual > nc; cicloActual--) {
      //Desde cicloActual hasta nc
            haciaAtras(cicloActual); //Se vuelve hacia atrás
                        }
            dibujaCrono(cicloActual) //Se dibuja el cronograma para nc
            visualizaCiclo(cicloActual); //Se muestra el ciclo
}
```

- Si nc es mayor que cMax:

En este caso, se recorren hacia adelante los ciclos aplicando sus cambios hasta llegar a cMax:

```
else if (nc > cMax) { // si se supera , se va a cMax
      for (cicloActual; cicloActual < cMax; cicloActual++){
      calculaPrev(cicloActual);
      // Se calculan los valores previos para cada ciclo
            haciaAdelante(cicloActual);
            // Se modifican los valores de las tablas
      }
            calculaPrev(cicloActual);
            // Se calculan los valores previos para cMax
            haciaAdelante(cicloActual); // Se colocan sus valores
            dibujaCrono(cicloActual); //Se muestra el crono en cMax
            visualizaCiclo(cicloActual); // Se muestra el ciclo actual
}
```

Para que el usuario sepa en que ciclo se encuentra, la función visualizaCiclo(cicloActual) únicamente recibe como parámetro el ciclo actual y actualiza un elemento HTML con *id="actual"*:

```
function visualizaCiclo(cicloActual){
            document.getElementById('actual').innerHTML =
                  "<b>Ciclo actual: " + cicloActual + "</b>";
      }
```

4.5.5) Manejo de eventos y algoritmos adicionales

Este apartado se ha reservado para explicar cómo se gestionan los diferentes eventos que se generan cuando el usuario interactúa con la aplicación y algunas funciones adicionales que también forman parte del transcurso de ejecución de estos eventos.

La mayoría de los eventos de la aplicación se gestionan usando JQuery, ya que proveen una mayor transparencia al programador a la hora de ser implementados y ejecutados. Por otra parte, también se han incluido otras maneras de manejar eventos como puede ser mediante el método addEventListener() de JavaScript.

Estos eventos se dispararán cuando su listener asociado detecte que el usuario ha interactuado con algún elemento en concreto y ejecutarán alguna acción que afectará al desarrollo de la interactuación con la aplicación.

Los eventos con los que la aplicación deberá trabajar son:

- ➔ Botones de cambio de ciclo
- ➔ Cambiar el tamaño del cronograma
- ➔ El desplegable contenedor de las tablas
- ➔ Botones de maximización de las tablas y el cronograma.
- ➔ Volver a la pantalla de configuración
- ➔ Funciones para leer y ejecutar archivos de código.

Seguidamente se detallarán uno por uno cómo se activan estos eventos y que acciones realizan.

4.5.5.1) Botones de cambio de ciclo

Una vez ya estamos en la parte de la interfaz dedicada a la visualización de tablas de datos y el cronograma, estos botones estarán situados en la barra de navegación.

Se ha decidido gestionar su uso con JQuery, mediante la llamada: $("#elemento").click().

Asociado a cada botón de cambio de ciclo se tendrá un listener JQuery [JQF2|20], de modo que cuando el usuario clique alguno de ellos, se dispare el contenido de su evento.

Cada gestor de eventos contiene una función de callback que lo que realizará será una llamada a la función de cambio de ciclo cambiaCiclo() , pasando como parámetros el cicloActual en el cual el usuario se encuentra y el número en ciclos en el que se quiere desplazar, que será el número que se indica en el botón.

De este modo, el código asociado al manejo del evento *"click"* de estos botones quedaría como sigue:

```
$("#menosDiez").click(function () { cambiaCiclo(cicloActual, -10); });
$("#menosCinco").click(function () { cambiaCiclo(cicloActual, -5); });
$("#menosUno").click(function () { cambiaCiclo(cicloActual, -1); });
$("#masUno").click(function () { cambiaCiclo(cicloActual, +1); });
$("#masCinco").click(function () { cambiaCiclo(cicloActual, +5); });
$("#masDiez").click(function () { cambiaCiclo(cicloActual, +10); });
```

Donde "#id" es el identificador del botón en cuestión y el callback llama a la función de cambio de ciclo.

4.5.5.2) Cambiar el tamaño del cronograma

El evento asociado a este elemento funciona de una manera similar a la anterior, cambiando la función a la que llama.

También habrá un listener JQuery asociado al evento click, cuyo contenido se disparará cuando el usuario clique en el botón con *id="tamCrono":*

```
$("#tamCrono").click(function () {

      var nuevo = document.getElementById('tamDeseado').value;

      cambiaTam(nuevo)});
```

Se creará una variable llamada nuevo donde se guardará el valor que el usuario ha insertado en el *<input id="tamDeseado">*. Con este valor se llamará a la función cambiaTam() que será la encargada de cambiar el tamaño actual del cronograma. La función cambiaTam() se ejecuta dándole como parámetro un valor, que será analizado para ver si es posible o no realizar el cambio de tamaño, ya que sólo se admitirán valores numéricos entre 1 y 100.

4.5.5.3) El desplegable contenedor de las tablas

En la sección dedicada a detallar la función tablas(), se crearon dos listas de elementos HTML asociadas a la creación de una lista de elementos contenedores del espacio reservado para la visualización de las tablas.

Estos elementos *<li>* se concatenaban en las variables collection_verticales y collection_horizontales, unas listas que, posteriormente, se indexaban en la página:

```
var nuevo_elemento = "<li>" + "<button type='button' class='collaps'>

   <i class='material-icons'>filter_drama</i><b>" + t_id
   +"</b></button>"

   + "<div class='content' style='display: none;'>"

   + "<button style='float: right' class='max'>+</button>"

   + "<div id='"+ t_id + "_" +"'></div>"

   + "</div></li>";
```

Seguidamente se va a explicar cómo se ha hecho posible que estos elementos conformen un menú desplegable [RD4|99-20]. Para ello se han asociado eventos JavaScript mediante la función addEventListener() [MZ3|19].

Se observa en nuevo_elemento hay dos elementos al mismo nivel (hermanos) dentro de la etiqueta *<li>*. Un botón con la clase CSS asociada *"colaps"* y un elemento * * cuya clase asociada es *"content"* con *style="display : none;"* es decir, oculto.

Se utilizarán algunas propiedades HTML y de las clases asociadas a estos elementos para crear y manejar el evento asociado a mostrar y ocultar el elemento *"content"* cuando el usuario clique en el botón *"collaps"*.

```
var col = document.getElementsByClassName("collaps");
```

Lo primero que se hace es guardar en la variable col, aquellos elementos con la clase *"collaps"* usando document.getElementsByClassName("collaps"). De este modo se tendrá una lista de elementos.

Una vez hecho esto, mediante un bucle *for* se le asocia a cada uno de estos elementos un listener para cuando el usuario clique sobre él, usando: addEventListener("click" , function() {....}).

```
for (i = 0; i < col.length; i++) {
      //Para cada elemento en la lista col
        col[i].addEventListener("click", function () {
            //Se añade un listener de evento "click"
            var content = this.nextElementSibling;
            //Se obtiene el elemento hermano al botón "collaps"
            if (content.style.display === "block") {
                  //Si es visible, se oculta
                content.style.display = "none";
            } else { // En caso contrario, se muestra
                content.style.display = "block"; }
        });
  }
```

De este modo, cuando el usuario clique sobre uno de los botones *"collaps"*, se obtendrá su elemento hermano, en este caso el <*div*> cuya clase es *"content"*. Si su atributo *display* indica que ha de observarse, se ocultará y viceversa.

4.5.5.4) Botones de maximización y minimización

Con estos botones se le dotará a la aplicación de la posibilidad de esconder algunos elementos para únicamente observar una de las tablas. Se entrará en el modo de maximización, es decir, se reservará todo el espacio de visualización disponible para la tabla que se pretende maximizar y el resto de los elementos permanecerán ocultos.

En el modo de visualización normal de la interfaz, cada tabla dispondrá de un botón en su esquina superior derecha para maximizarla.

En el caso de las tablas de datos comunes se tratará del botón:

```
"<button style='float: right' class='max'>+</button>"
```

generado junto al espacio contenedor de las tablas cuando se ejecuta la función tablas().

En el caso del cronograma, habrá un botón de idéntico estilo, pero con un identificador diferente. Es el que se añade a la hora de dibujar el cronograma con la función dibujaCrono(), con *id="maxcrono"*:

```
var crono_html_button = "<div><button style='float : right;'
id='maxcrono' class='max2'><b>+</b></button></div>";
```

Una vez se esté en el modo de maximización, se colocarán en lugar de los botones de maximización, unos botones de similar estilo mediante los cuales se podrá *"minimizar"* la tabla que se esté visualizando en ese momento:

```
var min_button = "<button style='float : right;' id='min' class='max2'>-
</button>"

var min_crono_btn = "<button style='float : right;' id='mincrono'

class='max2'>-</button>"
```

Estos botones, tendrán cada uno un listener asociado y una vez el usuario los pulse, se captará el evento y se realizará una acción determinada.

Para dotar a la aplicación de esta funcionalidad, se ha decidido crear como variables globales dos booleanos: var max_tablas = false y var max_crono = false que al inicio tendrán asignado el valor *"falso"* y cambiarán a verdadero cuando el usuario decida maximizar una de las tablas o bien el cronograma.

4.5.5.4.1) Maximizar las tablas

En el caso de las tablas comunes se ha decidido registrar sus listeners con la función addEventListener() de JavaScript, de manera parecida al elemento desplegable de las tablas.

En primer lugar, se ha decidido guardar en una variable, todos los elementos cuya clase asociada sea *"max"*, de manera similar a la mencionada anteriormente con los elementos cuya clase era *"colaps"*:

```
var max = document.getElementsByClassName("max");
```

Así, se puede recorrer la variable agregando un listener y una función callback a cada elemento del vector:

```
for(i= 0; i< max.length; i++){
            max[i].addEventListener("click",function(){ … });   }
```

Una vez pulsado uno de los botones (evento "click") se disparará la función de callback que ejecutará las siguientes líneas de código:

```
hermano = this.nextElementSibling;
tabla_max = hermano.innerHTML;
hermano.innerHTML = "";
max_tablas = true;
```

Primero se guarda el elemento hermano del botón pulsado (this) que será la propia tabla que se quiere maximizar. Su contenido quedará guardado en la variable tabla_max. Se elimina el innerHTML del elemento después de guardarlo en la variable y se cambia el valor de max_tablas a *verdadero*, ya que se va a maximizar una tabla.

También se esconden aquellos elementos que se quieren ocultar de la visualización:

```
document.getElementById('cronograma').setAttribute('style',
'display: none;');
```

```
document.getElementById('verticales').setAttribute('style',
'display: none;');

document.getElementById('horizontales').setAttribute('style',
'display: none;');
```

Por último, se coloca el contenido de la tabla en el espacio reservado para la maximización, y el botón de minimización en el espacio de botones. También se cambia el atributo *display* del espacio con *id="maximizar"* para que este sea visible:

```
document.getElementById('max_btns').innerHTML = min_button;

document.getElementById('max_space').innerHTML = tabla_max;

document.getElementById('maximizar').style.display="block";
```

De este modo se ocultará el resto de los elementos y se dejará disponible todo el espacio de la interfaz para la visualización de una única tabla.

4.5.5.4.2) Maximizar el cronograma

Para dejar el espacio disponible para el cronograma, dotamos de funcionalidad al botón de su esquina superior derecha:

```
var crono_html_button = "<div><button style='float : right;'
id='maxcrono' class='max2'><b>+</b></button></div>";
```

En este caso se ha decidido capturar y gestionar su evento "click" mediante JQuery, como ocurría con los botones de cambio de ciclo, del modo:

```
$('#maxcrono').click(function(){ ... })
```

Una vez el usuario pulse el botón, se capturará el evento con JQuery y se ejecutará el código introducido en la función callback.

Lo primero que realizará esta función es darle el valor booleano *"verdadero"* a la variable max_crono, para indicarle a la aplicación que la tabla que va a maximizarse será la del cronograma.

```
max_crono = true;
```

Seguidamente, se eliminará el contenido del elemento donde aparecía el cronograma inicialmente, así como el botón de maximizado.

```
document.getElementById('crono_btns').innerHTML = "";

document.getElementById('crono_space').innerHTML = "";
```

Se cambiarán los atributos de los elementos de visualización de modo que sólo quede visible el elemento con *id="maximizar"*.

```
document.getElementById('maximizar').setAttribute('style',
'display: block;');

document.getElementById('cronograma').setAttribute('style',
'display: none;');

document.getElementById('verticales').setAttribute('style' ,
'display: none;');

document.getElementById('horizontales').setAttribute('style' ,
'display: none;');
```

Por último, se llamará a la función dibujaCrono(), la cual al tener max_crono = true , colocará el cronograma y su botón asociado los * * y * * respectivamente.

```
dibujaCrono(cicloActual);
```

De esta manera, el usuario tendrá todo el espacio disponible para la visualización del cronograma.

4.5.5.4.3) Los botones de minimización

Para volver a la interfaz de visualización inicial, bastará con pulsar el botón de minimización correspondiente. Cada uno de ellos tendrá un listener JQuery asociado de modo que cuando el usuario lo pulse JQuery captará el evento *"click"* y se ejecutará una función callback: $('#button').click(function(){ … }

Dependiendo si la tabla mostrada en el * * es una de las comunes o es el cronograma, en el momento de pulsar el botón de minimización, se ejecutará una sección de código u otra.

➔ Estando el cronograma maximizado, si el usuario decide minimizarlo, los pasos que la aplicación seguirá serán los siguientes:

El botón asociado a la acción de minimizar será:

```
var min_crono_btn = "<button style='float: right;' id='mincrono'
class='max2'> - </button>"
```

Y al pulsarlo se manejará su evento con el siguiente handler:

```
$('#mincrono').click(function () { … });
```

Lo primero que se hará será eliminar el botón de minimización que se ha pulsado con el método jQuery remove():

```
$('#mincrono').remove();
```

Seguidamente, se colocará un botón de maximización asociado al cronograma, para que el usuario pueda volver a maximizarlo cuando él decida y se colocará el cronograma en el espacio de la interfaz en el cual estaba inicialmente. También se eliminará el contenido del espacio de maximización *"max_space"*.

```
document.getElementById('crono_btns').innerHTML = crono_html_button;
document.getElementById('crono_space').innerHTML =
                        document.getElementById('max_space').innerHTML;
```

```
document.getElementById('max_space').innerHTML = "";
```

Una vez hecho esto, se reestablecerán los atributos *display* de los asociados a la visualización, ocultando el <*div id=”maximizar”*>.

```
document.getElementById('maximizar').style.display = "none";
document.getElementById('verticales').style.display = "block";
document.getElementById('horizontales').style.display = "block";
document.getElementById('cronograma').style.display = "block";
```

Se reestablece el valor del booleano max_crono a falso, ya que vamos a salir de la maximización:

```
max_crono = false;
```

En el caso de que el tamaño del cronograma sea mayor al máximo estipulado para el tamaño normal de visualización, se volverá a colocar num_columnas al máximo de 100 para el modo normal.

```
if (num_columnas >= 100) { num_columnas = 100; dibujaCrono(cicloActual)}
```

Ahora, se colocarán los valores de scroll de “crono_space” a su valor máximo, tanto para el scroll vertical como para el horizontal:

```
var scrollH = document.getElementById('crono_space').scrollHeight;
document.getElementById('crono_space').scrollTop = scrollH;
var scrollW = document.getElementById('crono_space').scrollWidth;
document.getElementById('crono_space').scrollLeft = scrollW;
```

Y, por último, se vuelve a dar funcionalidad al nuevo botón de maximización colocado:

```
$('#maxcrono').click(function () {… });
```

➔ En el caso de haber maximizado una tabla común y se pretenda minimizarla:

El botón asociado a esta acción será:

```
var min_button = "<button style='float: right;' id='min'
class='max2'>-</button>"
```

Al pulsarlo, se ejecutará la función incluida en el siguiente handler:

```
$('#min').click(function () {.. });
```

Lo primero, como en el caso anterior, será eliminar el botón de minimización.

```
$('#min').remove();
```

Una vez hecho esto, se copiará el contenido de *"max_space"* a la posición original de la tabla, guardada en la variable hermano. También se eliminará el contenido del espacio de maximización.

```
hermano.innerHTML = document.getElementById('max_space').innerHTML;
document.getElementById('max_space').innerHTML = "";
```

Para finalizar, se reestablecerán los atributos *display* de los espacios de visualización a su valor inicial, cambiando también el valor del booleano max_tablas a falso:

```
document.getElementById('maximizar').style.display = "none";
document.getElementById('verticales').style.display = "block";
document.getElementById('horizontales').style.display = "block";
document.getElementById('cronograma').style.display = "block";
max_tablas = false;
```

4.5.5.5) Volver a la pantalla de configuración

Una vez el usuario quiera volver a cargar otro fichero, modificar algo del código que está ejecutando o cambiar alguna configuración del simulador, deberá ser capaz de volver a la interfaz inicial de configuración y edición.

Con este fin, se ha colocado un botón en la barra de navegación de la interfaz de visualización de datos. Dicho elemento es:

```
<a class="waves-effect btn offset-s1"
id="pantalla_carga">Volver a carga</a>
```

Al pulsar este botón con *id="pantalla_carga"* , se controlará su evento "click" mediante JQuery, como en casos anteriores:

```
$('#pantalla_carga').click(function(){…});
```

El listener lanzará una función callback que realizará las operaciones necesarias para devolver al usuario a la pantalla inicial con los siguientes pasos:

Se colocará el valor *"falso"* de nuevo en las variables max_crono y max_tablas, ya que por defecto el usuario volverá de nuevo a la pantalla de visión de datos con el modo de maximización desactivado:

```
max_crono = false;
max_tablas = false;
```

En consecuencia, se eliminará cualquier rastro de código HTML que pudiera permanecer en el <*div id="maximizar"*>:

```
document.getElementById('max_btns').innerHTML= "";
        document.getElementById('max_space').innerHTML = "";
```

Y por último se harán visibles la barra de navegación inicial y el espacio de configuraciones y del editor. La interfaz de visualización de datos se ocultará y se dejarán las configuraciones de visualización de cada uno de sus elementos como estaban por defecto:

```
document.getElementById('maximizar').style.display="none";

document.getElementById('cronograma').style.display="block";

document.getElementById('verticales').style.display="block";

document.getElementById('horizontales').style.display="block";

document.getElementById('navCarga').style.display="block";

document.getElementById('cargaDatos').style.display="block";

document.getElementById('navApp').style.display="none";

document.getElementById('mainApp').style.display="none";
```

4.5.5.6) Funciones para leer y ejecutar ficheros

Este apartado queda dedicado a explicar cómo la aplicación ofrece la posibilidad al usuario de subir ficheros de código, que se muestren en la instancia de CodeMirror y puedan editarse y ser ejecutados.

Se han definido dos funciones para gestionar estas operaciones:

- ➔ Función leerFichero()
- ➔ Función ejecutar_fichero()

4.5.5.6.1) Función leerFichero()

La función leerFichero() es la que se va a encargar de, dado un fichero de código que el usuario pretenda subir, leerlo y realizar las operaciones pertinentes para que su contenido se muestre en la instancia de CodeMirror proporcionada por la interfaz.

El usuario será capaz de subir un archivo gracias al elemento *<input type=" file" id="examinar">* perteneciente a la interfaz de configuración y editor de texto:

```
<label class="waves-effect grey btn col s1" for="examinar">Subir
Archivo</label>

<input type="file" id="examinar" onchange="leerFichero(this)"
style="display: none;"></input>
```

Una vez el Usuario haga "click" en el botón de "Subir Archivo", se desplegará el menú de carga de ficheros asociado al <input type="file">. Cuando el usuario haya seleccionado el fichero que desea cargar, se disparará el evento "onchange": onchange="leerFichero(this)" , donde this.files, será un objeto *FileList* que contiene un conjunto de objetos tipo *File*. En este caso, el fichero cargado estará en la primera posición de *files*, *files[0]* ya que únicamente se carga un fichero cada vez [WD|10].

Ya entregado el objeto como parámetro a la función leerFichero(fichero_ensamblador), esta creará una instancia del objeto *FileReader* de JavaScript:

```
var reader = new FileReader();
```

Y se comenzará la lectura del fichero en cuestión:

```
reader.readAsText(fichero_ensamblador.files[0]);
```

Seguidamente, su contenido quedará guardado en el atributo *"result"* del *FileReader*, por lo que mediante un listener en reader se gestionará lo que hacer con su contenido:

```
    reader.addEventListener("load", function () {
        if(reader.result){
            document.getElementById('checkfile')
            .setAttribute('checked', 'true')}
                  mirror.setValue(reader.result);
                  mirror.save();
    }, false);
```

De este modo, cuando reader termine de leer el fichero se manejará el evento *"load"*, el cual ejecutará una función que **a)** modificará el valor del atributo *"checked"* del *checkbox* utilizado para indicar al usuario que el fichero se ha cargado correctamente, **b)** cambiará el valor del contenido de la instancia de CodeMirror por el valor del fichero cargado y **c)**

actualizará con save() el valor del contenido del <*textarea id=”ensamblador”*>, asociado al editor CodeMirror.

4.5.5.6.2) Función ejecutarFichero()

La función ejecutarFichero() se encargará de, una vez se tenga cargado un fichero de código o bien el usuario decida ejecutar el programa de prueba, enviar el código al simulador, recibir su ejecución y preparar todo lo necesario para pasar a la visualización de datos.

Para este proceso, se recibirá el código a ejecutar desde la instancia de CodeMirror y, mediante una llamada a save(), se guardará en su <*textarea id=”ensamblador”*> asociado.

```
mirror.save();

//Se guarda el código de “mirror” en el <textarea id=”ensamblador>
```

Estando ya el código en el <*textarea*> se llamará a la función ejecutar_ensamblador() del simulador. El resultado de esta llamada, es decir, el resultado de la ejecución de la emulación, se guardará en la variable global sim_data con formato JSON, actualizando su valor.

```
sim_data = JSON.parse(ejecutar_ensamblador(devuelve_json = 1));
```

Hecho esto, se tendrá que actualizar el resto de las variables globales de la aplicación, acorde al resultado obtenido de la ejecución:

```
info = sim_data["info"];

estructuras = sim_data["structures"]; // Información sobre tablas a
mostrar

cMax = sim_data["cycles"].length - 1; //Número de ciclo máximo

cMin = 1;  //Número de ciclo mínimo

cicloActual = cMin; //El ciclo en el cual se encuentra actualmente
```

```
ciclos = sim_data["cycles"]; //Datos de ejecución en cada ciclo

prevs = new Array(cMax); // Se inicializa el array de valores previos

datos_crono = new Array(cMax); //Se inicializa el array de valores del
cronograma

crono_values = new Array(cMax *

                  document.getElementById("NUM_VIAS_ISSUE_p").value);

crono_pc = [];

crono_instruc = [];  //Se inicializan las estructuras de datos del
cronograma
```

También se inicializarán los valores globales auxiliares para el funcionamiento de la aplicación:

```
max_tablas = false; //Por defecto, no se estarán maximizando tablas

max_crono = false; //Por defecto, no se estará maximizando el cronograma

num_columnas = 25; //Por defecto, tamaño inicial del cronograma será 25

tablas_verticales = []; //Se inicializan los arrays de tablas

tablas_horizontales = [];

hermano = "";

// Se inicializan las variables auxiliares hermano y tabla_max

tabla_max = "";
```

Después de estas actualizaciones de variables, se prosigue ejecutando las funciones descritas en apartados anteriores para crear las nuevas tablas y rellenarlas con los valores iniciales obtenidos (ciclo 1):

```
tablas();

cronograma();

rellenaInicio();

visualizaCiclo(cicloActual);
```

Ahora se inicializarán los listeners asociados a las tablas y al elemento desplegable. El listener de maximización de cronograma se inicializa en la llamada a rellenaInicio(), la cual hace una llamada a dibujaCrono(cMin).

```
var i;
var col = document.getElementsByClassName("collaps");
var max = document.getElementsByClassName("max");

for (i = 0; i < col.length; i++) {
        col[i].addEventListener("click", function () { … })  }
for (i = 0; i < max.length; i++) {
        max[i].addEventListener("click", function () { … }) }
```

Por último, ya realizadas todas estas acciones, se ocultará la pantalla de carga y se mostrará al usuario el espacio de visualización de datos con las nuevas tablas creadas:

```
document.getElementById('navCarga').style.display="none";
document.getElementById('cargaDatos').style.display="none";
document.getElementById('navApp').style.display="block";
document.getElementById('mainApp').style.display="block";
```

Ahora el usuario ya podrá navegar por las tablas y por los diferentes ciclos de ejecución, pudiendo volver a la pantalla de carga cuando él decida.

4.5.5.6.3) El fichero ejecutar-ensamblador.js

El fichero *ejecutar-ensamblador.js* es el propio simulador transpilado de C a JavaScript mediante el uso de Emscripten. Una vez tenemos portado el simulador, podemos llamar a su función ejecutar_ensamblador() desde ejecutarFichero() y que esta nos devuelva el resultado de la ejecución.

Para ello, se necesita en *ejecutar-ensamblador.js* una función que obtenga los parámetros de configuración escogidos en la interfaz de carga y nos devuelva un resultado.

Esto se hace mediante una llamada a Module.ccall de Emscripten, que es una función que permite llamar a una función exportada de una librería en C (el simulador):

```
var result_ptr = Module.ccall('libmipsooo_main',
     // libmipsooo_main returns a pointer to a char* string
    'number'
      ,
    // Tipos de los argumentos
    [
      //char* ensamblador_s,
      'number',
      //int config_en_s,
      'number',
      //int devuelve_json,
      'number',
      //int NUM_INICIO_ENTEROS_p, int TEVAL_ENTEROS_p,
      //int TAM_RS_ENTEROS_p, int SEGMENTADO_ENTEROS_p,
      'number', 'number', 'number', 'number',
      //int NUM_INICIO_SUMREST_p, int TEVAL_SUMREST_p,
      //int TAM_RS_SUMREST_p, int SEGMENTADO_SUMREST_p,
      'number', 'number', 'number', 'number',
      //int NUM_INICIO_MULTDIV_p, int TEVAL_MULTDIV_p,
      //int TAM_RS_MULTDIV_p, int SEGMENTADO_MULTDIV_p,
      'number', 'number', 'number', 'number',
      //int NUM_INICIO_MEMDATOS_p, int TEVAL_MEMDATOS_p,
      //int TAM_TAMPON_LECT_p, int TAM_TAMPON_ESCR_p,
      //int SEGMENTADO_MEMDATOS_p,
      'number', 'number', 'number', 'number', 'number',
```

```
      //int NUM_VIAS_ISSUE_p, int NUM_VIAS_BUS_p,

      //int NUM_VIAS_COMMIT_p,

      'number', 'number', 'number',

      //int tipo_predictor_p, int TAM_BUFFER_PREDIC_p,

      'number', 'number',

      //int TAM_REORDER_p,

      'number',

      //int load_forwarding_p

      'number'

    ]

, // Argumentos

[     ensamblador_s_ptr,

      arg('config_en_s'),

      devuelve_json,

      arg('NUM_INICIO_ENTEROS_p'), arg('TEVAL_ENTEROS_p'),
      arg('TAM_RS_ENTEROS_p'), arg('SEGMENTADO_ENTEROS_p'),

      arg('NUM_INICIO_SUMREST_p'), arg('TEVAL_SUMREST_p'),
      arg('TAM_RS_SUMREST_p'), arg('SEGMENTADO_SUMREST_p'),

      arg('NUM_INICIO_MULTDIV_p'), arg('TEVAL_MULTDIV_p'),
      arg('TAM_RS_MULTDIV_p'), arg('SEGMENTADO_MULTDIV_p'),

      arg('NUM_INICIO_MEMDATOS_p'), arg('TEVAL_MEMDATOS_p'),
      arg('TAM_TAMPON_LECT_p'), arg('TAM_TAMPON_ESCR_p'),
      arg('SEGMENTADO_MEMDATOS_p'),

      arg('NUM_VIAS_ISSUE_p'), arg('NUM_VIAS_BUS_p'),
      arg('NUM_VIAS_COMMIT_p'),

      arg('tipo_predictor_p'), arg('TAM_BUFFER_PREDIC_p'),

      arg('TAM_REORDER_p'),

      arg('load_forwarding_p')           ] );
```

La llamada a Module.ccall, consta de tres argumentos. El primero de ellos es la función que se exporta, en este caso la función: 'libmipsooo_main'. Seguidamente, para el segundo, aparece una lista con los tipos de los argumentos, todos de tipo “number”. El tercer argumento, son los datos de la llamada, que se detallan a continuación:

- ➔ ensamblador_s (char*): Cadena con programa ensamblador a simular
- ➔ config_en_s (int): Lee la configuración de la primera línea de ensamblador_s
- ➔ devuelve_json (int): Devolver el resultado de la simulación representado en json
- ➔ NUM_INICIO_ENTEROS_p, TEVAL_ENTEROS_p, TAM_RS_ENTEROS_p, SEGMENTADO_ENTEROS_p (int): Configuración de operador enteros
- ➔ NUM_INICIO_SUMREST_p, TEVAL_SUMREST_p, TAM_RS_SUMREST_p, SEGMENTADO_SUMREST_p (int): Configuración operador suma/resta c.f.
- ➔ NUM_INICIO_MULTDIV_p, TEVAL_MULTDIV_p, TAM_RS_MULTDIV_p, SEGMENTADO_MULTDIV_p (int): Configuración operador mult/div
- ➔ NUM_INICIO_MEMDATOS_p,TEVAL_MEMDATOS_p, TAM_TAMPON_LECT_p,TAM_TAMPON_ESCR_p, SEGMENTADO_MEMDATOS_p (int): Configuración de unidad de memoria
- ➔ NUM_VIAS_ISSUE_p, NUM_VIAS_BUS_p, NUM_VIAS_COMMIT_p (int): Configuración vías superescalar
- ➔ tipo_predictor_p, TAM_BUFFER_PREDIC_p (int): Configuración del predictor de saltos
- ➔ TAM_REORDER_p (int): Número de entradas del ROB
- ➔ load_forwarding_p (int): 0: Load bypassing, !0: Load forwarding

Estos parámetros se extraen directamente de la interfaz de configuración de la aplicación para después ser utilizados por la *Module.ccall*.

La función retorna un puntero a cadena de caracteres, que luego se convierte a una String de Python. La cadena de caracteres se libera con *Module._free*. También se libera con *Module._free* la cadena de caracteres que se crea en el mismo código para almacenar el programa en ensamblador ensamblador_s_ptr:

```
Module._free(ensamblador_s_ptr);

var result = Pointer_stringify(result_ptr);

Module._free(result_ptr);

return result; // El resultado de la simulación
```

5) Interfaz, implantación, y pruebas

En este capítulo se presentan, una vez detalladas las diferentes partes de la de la solución al problema, la interfaz del simulador, el Shell script utilizado para su implantación y finalmente, algunas pruebas de rendimiento para comprobar si este mejora con respecto al simulador original utilizado en AIC.

5.1) Interfaz de carga y ejecución de ficheros

La interfaz de carga y ejecución de ficheros se observa en la Figura 25. Las funciones implementadas en el código utilizadas por esta interfaz son las siguientes:

- La función leerFichero() mediante la cual cargamos un fichero de código, y se muestra en la instancia de CodeMirror (ver Figura 26).
- La función ejecutar_fichero() que será la encargada de enviar el fichero de código al simulador, recoger su salida, actualizar las variables globales según los datos obtenidos y llevar al usuario a la interfaz de visualización de datos.

Esta última función llama al simulador, ya portado de C a JavaScript en el fichero *ejecutar-ensamblador.js*, a través del cual se realiza la ejecución del código cargado y la obtención de resultados. El simulador nos devuelve una variable JSON cada vez que se realiza una ejecución, con sus parámetros y resultados particulares de cada una de ellas.

En la interfaz de visualización de datos, se ha añadido el botón "volver a carga" que lleva al usuario de nuevo a la interfaz de carga y configuración, observable en la Figura 27, permitiéndole volver de nuevo cuando lo necesite.

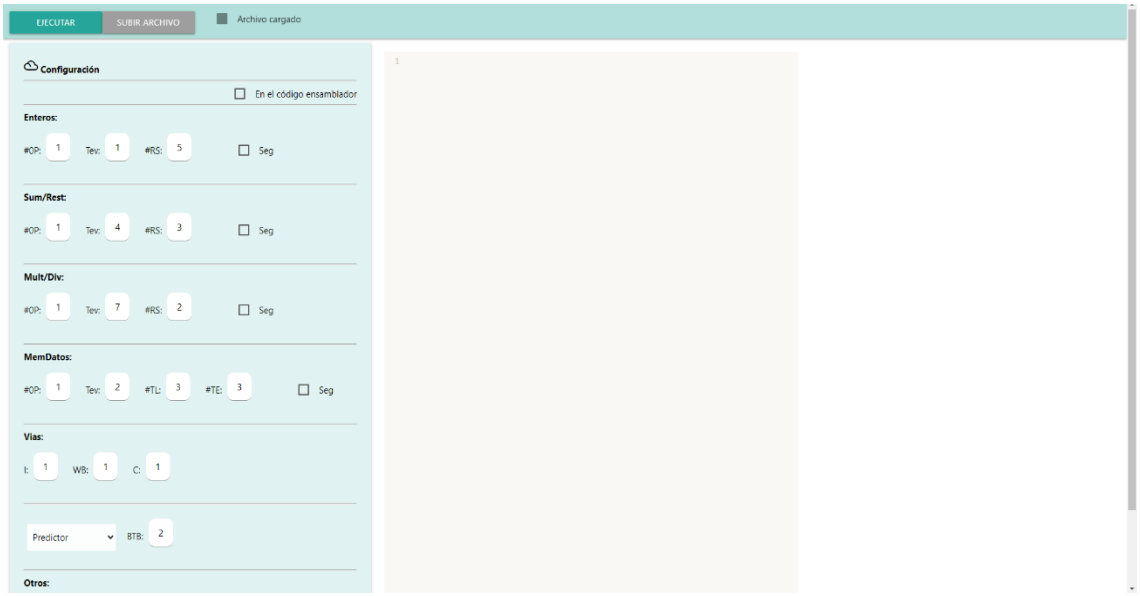


Figura 25: La interfaz de configuraciones, carga y ejecución de archivos

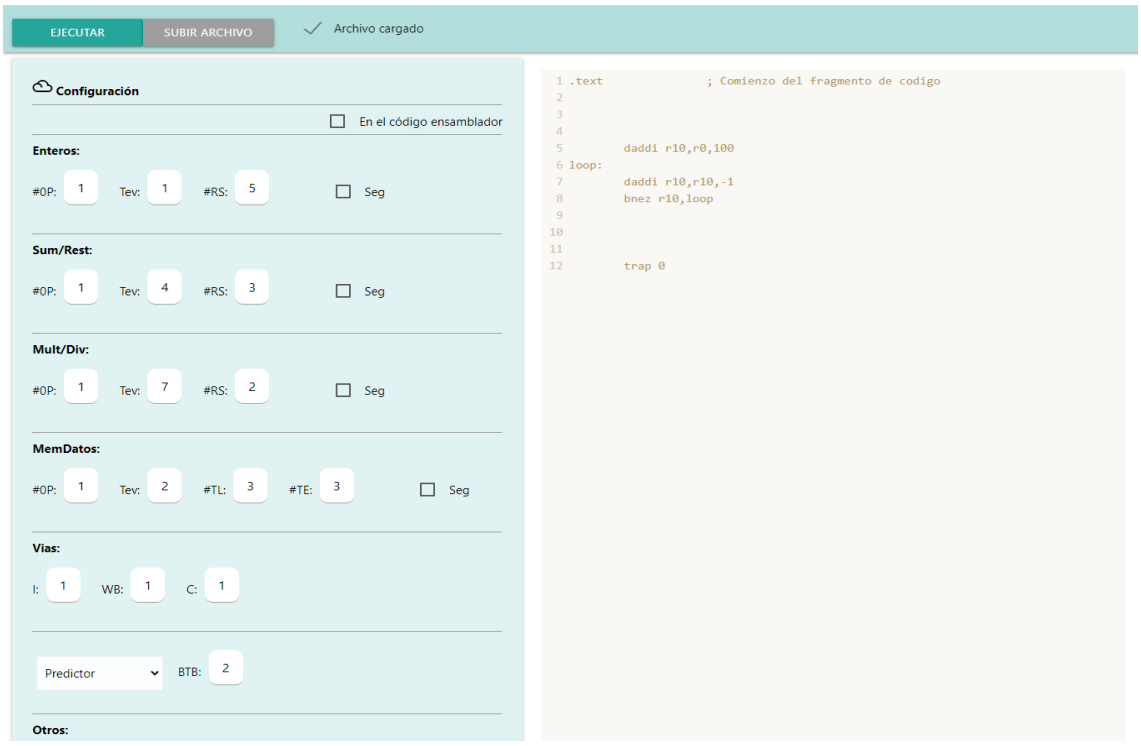


Figura 26: Código iterador de prueba cargado

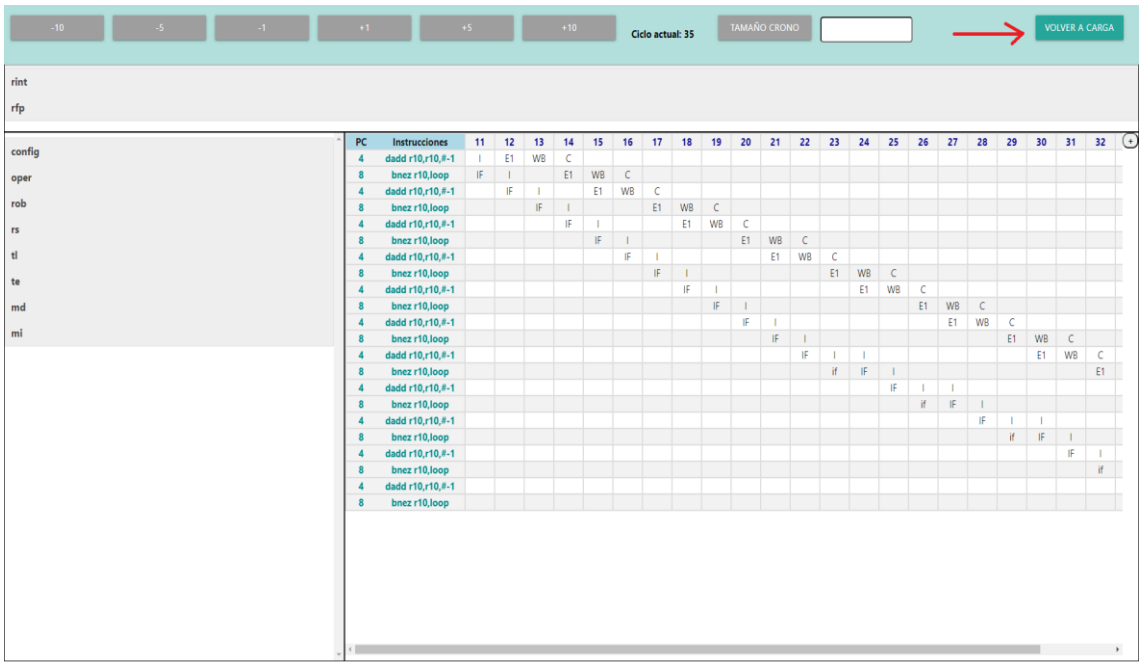


Figura 27: Ejecución del código iterador, el botón "volver a carga"

5.2) Interfaz de visualización de datos.

La interfaz de visualización de datos se muestra en la Figura 28. En las siguientes subsecciones se explican los aspectos principales de esta interfaz.

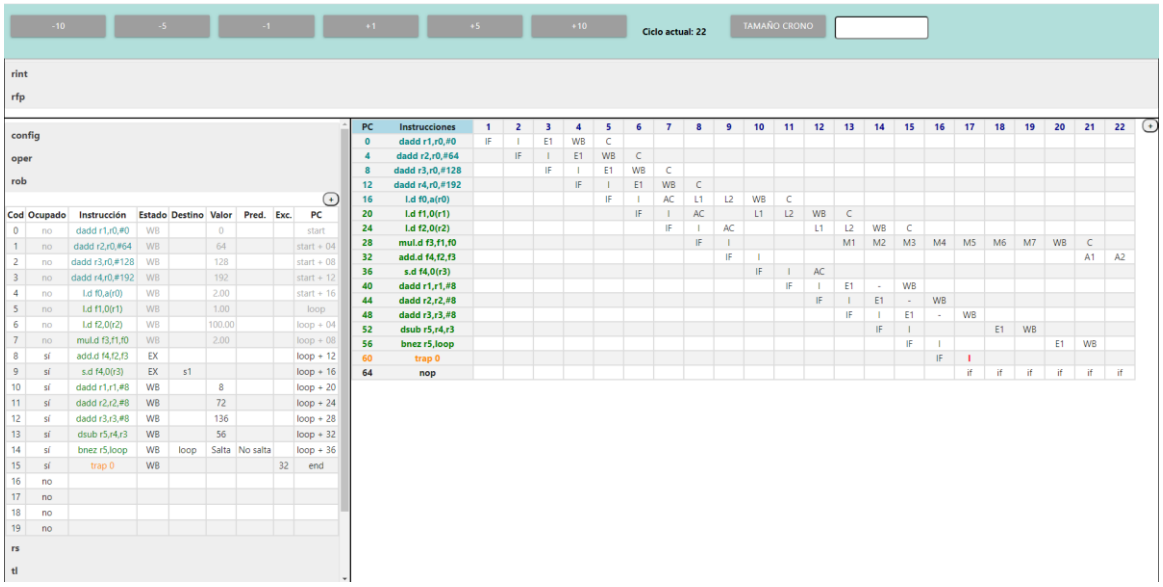


Figura 28: Interfaz de visualización de datos

5.2.1) Funciones involucradas

La interfaz de visualización de datos involucra a las siguientes fuciones:

- ➔ tablas(): para la creación de las estructuras de datos extraídas de la información devuelta por el simulador.
- ➔ cronograma(): para la creación de una estructura de datos en la cual organizar los datos del cronograma devuelto por el simulador.
- ➔ rellenaInicio(): para posicionar y rellenar las tablas construidas con los valores iniciales de la ejecución (ver Figura 29).
- ➔ dibujaCrono(): para rellenar los primeros datos de la tabla del cronograma.

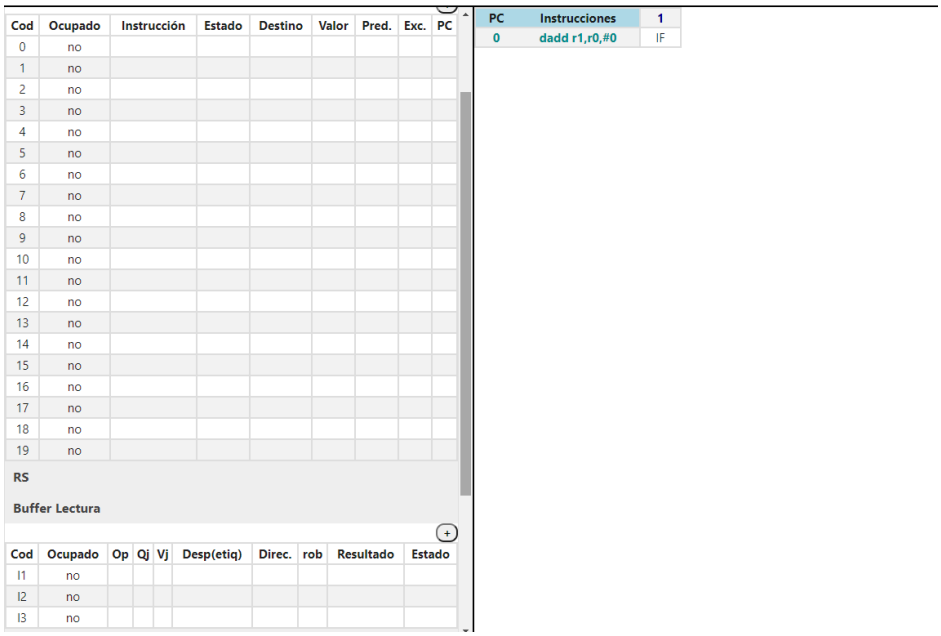


Figura 29: Tablas y cronograma con los valores iniciales

5.2.2) Cambio de ciclo de simulación

Las siguientes funciones se encargan de la funcionalidad correspondiente a cambiar el ciclo de simulación y actualizar correspondienteme los datos en las tablas de la interfaz de visualización:

- ➔ haciaAdelante(): con el objetivo de avanzar un ciclo y cambiar los datos de las tablas de visualización.
- ➔ calculaPrev(): para guardar los campos previos de cada celda en un ciclo determinado, permitiéndonos volver hacia atrás sin realizar algoritmos de búsqueda por cada una de las celdas de las tablas.
- ➔ haciaAtrás(): con la cual se retrocede un ciclo, actualizando los datos de las tablas.
- ➔ dibujaCrono(): actualiza la visualización del cronograma para el ciclo actual.
- ➔ cambiaCiclo(): destinada a llamar al resto de funciones cuando el usuario modifique el ciclo de simulación a visualizar.

Estas funciones se llaman como respuesta a eventos producidos por el usuario. Los botones de cambio de ciclo, visibles en la Figura 30, permiten al usuario avanzar o retroceder un número de ciclos determinado realizando llamadas a cambiaCiclo() (ver Figura 31).

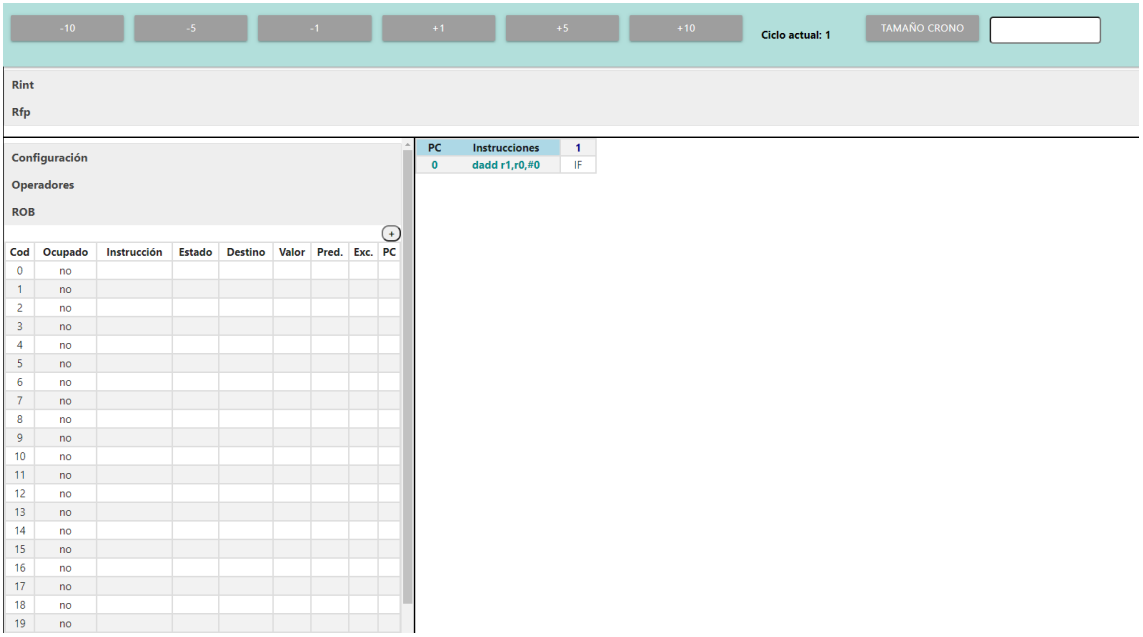


Figura 30: Tablas y cronograma en el ciclo 1, con los botones de cambio de ciclo y tamaño

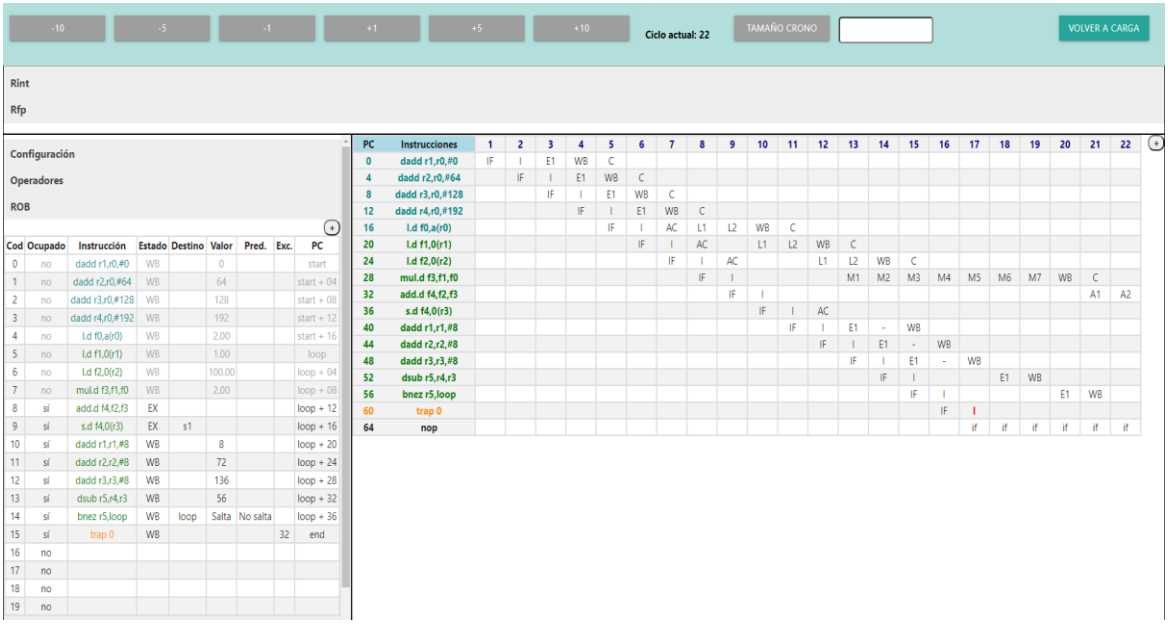


Figura 31: Tablas y cronograma en el ciclo 22

Además de los botones de cambio de ciclo, el usuario puede cambiar a cualquier ciclo visible dentro del cronograma. La función que provee de esta funcionalidad se denomina saltaCiclo() y se lista a continuación:

```
    saltar_a = ciclo;
    for(cicloActual; cicloActual>ciclo; cicloActual-- ){
            haciaAtras(cicloActual);
    }
        dibujaCrono(cicloActual)
        visualizaCiclo(cicloActual);

}
```

Para invocar a esta función, cada una de las celdas de cronograma que indica el ciclo de ejecución (primera fila del cronograma) incluye un listener. De este modo, cuando el usuario pulsa sobre una de ellas, se dispara esta función, actualizando todas las tablas con la información del ciclo correspondiente.

El código que añade los lístener es el siguiente:

```
var cyc = document.getElementsByClassName("ciclo");
for (i = 0; i < cyc.length; i++) {
                cyc[i].addEventListener("click", function () {
                    var salto_a = parseInt(this.innerHTML);
                    saltaCiclo(salto_a);
                                        });
            }
```

5.2.3) Cambio de ciclo de simulación

Para mejorar la visualización de la ejecución, se permite al usuario aumentar el tamaño en ciclos del cronograma, mediante la función cambiaTam(). Su llamada y parámetros se gestionan mediante un botón y un *<input type="text">* correspondiente para que el usuario pueda indicar los ciclos. Esto puede observarse en la Figura 32 y en la Figura 33.

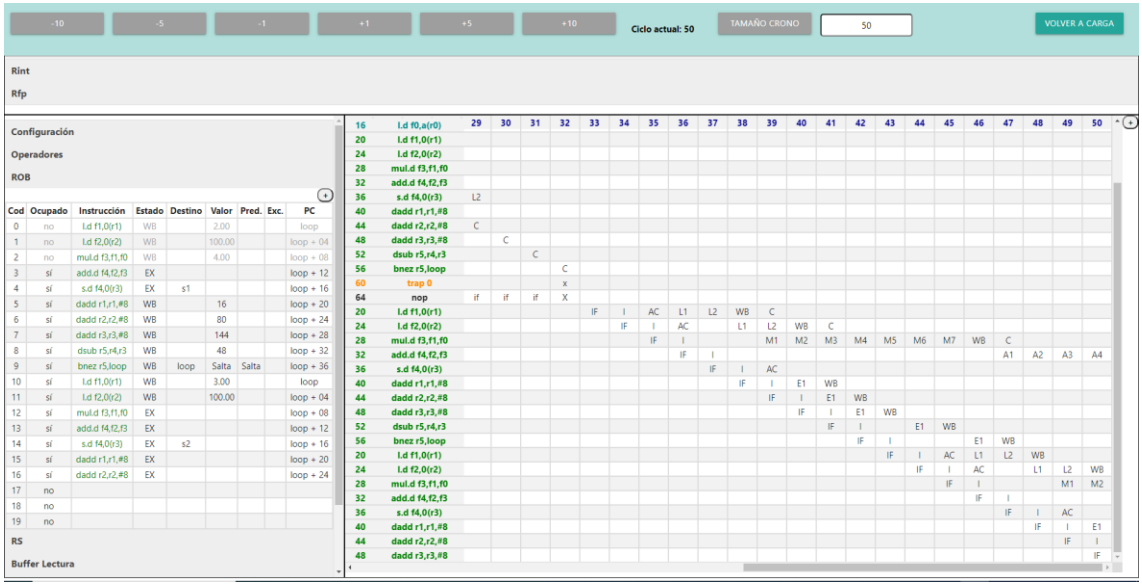


Figura 32: Ciclo 50 con tamaño de cronograma 50

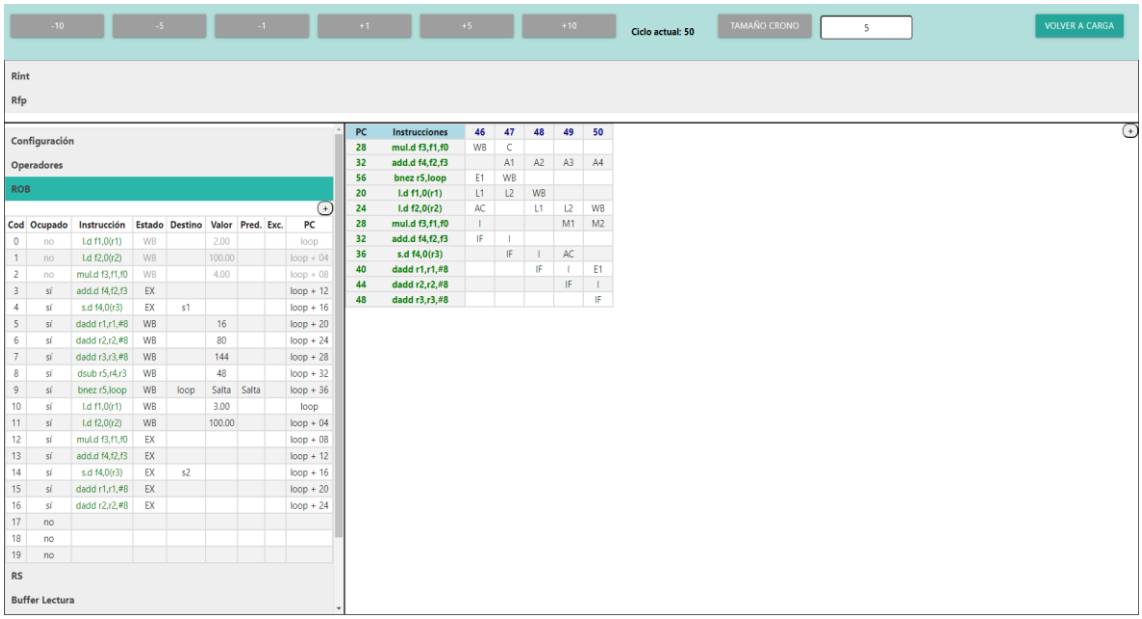


Figura 33: Ciclo 50 con tamaño de cronograma 5

5.2.4) Maximización y minimización de tablas y cronograma

Por motivos docentes, se ha añadido la funcionalidad de maximizar cualquier tabla ofreciéndoles todo el espacio disponible en la aplicación.

Para ello, existen botones y manejadores de eventos correspondientes destinados a maximizar y minimizar las tablas, tal como se observa en la Figura 34 y la Figura 35.

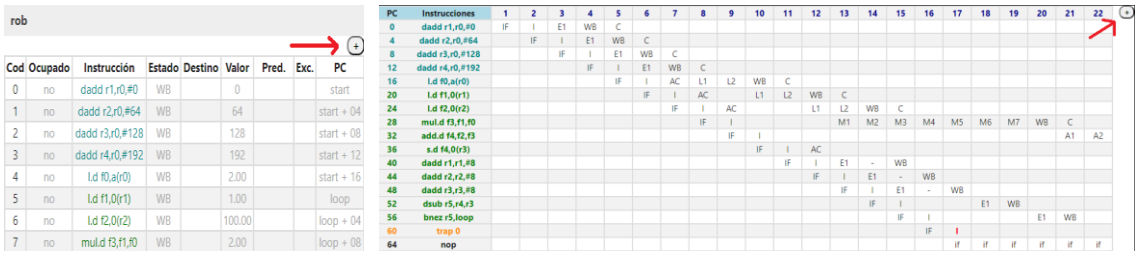


Figura 34: Los botones de maximización

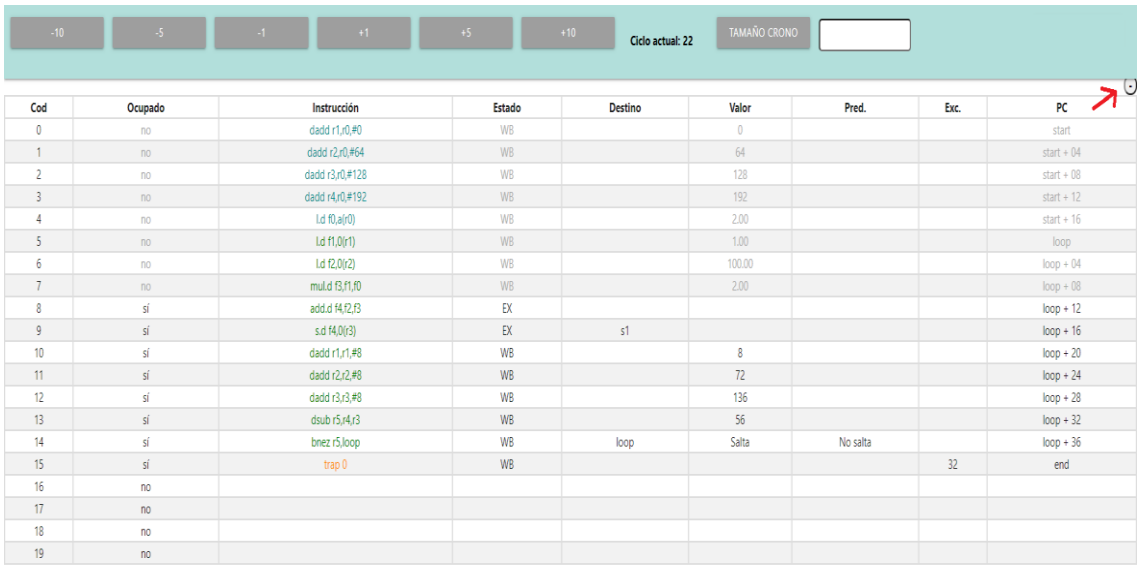


Figura 35: Tabla maximizada con botón de minimización

5.5) Unificación de ficheros: Implantación final

Con el propósito de contener toda la funcionalidad de la aplicación en un solo fichero y así dotarla de una fácil portabilidad e implantación, se ha desarrollado un Shell script que unifica todos los ficheros, proporcionando como salida, un único código HTML con todo lo necesario para la ejecución ya incluido.

El script, denominado *genera_index.sh*, está creado a modo que hay que incluirlo en la carpeta raíz del proyecto, tal como se muestra en la Figura 36.

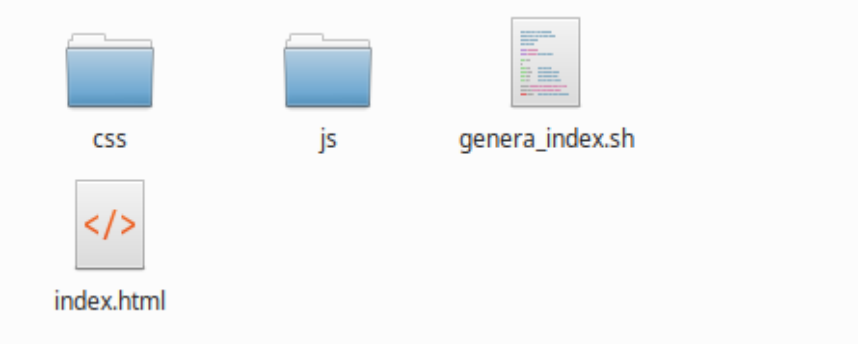


Figura 36: El directorio raíz

El procedimiento seguido se detalla a continuación:

➔ Se crea un archivo html nuevo desde terminal en el directorio raíz:

```
bire-One-753:/home/r/Escritorio/SimWeb#
bire-One-753:/home/r/Escritorio/SimWeb# touch generado.html
bire-One-753:/home/r/Escritorio/SimWeb#
```

➔ Seguidamente se ejecuta el script, el cual recorrerá cada elemento del directorio y lo incluirá en el nuevo HTML generado:

```
Aspire-One-753:/home/r/Escritorio/SimWeb#
Aspire-One-753:/home/r/Escritorio/SimWeb# ./genera_index.sh > generado.html
Aspire-One-753:/home/r/Escritorio/SimWeb#
```

El fichero html generadose puede publicar en cualquier plataforma para que los usuarios lo puedan utilizar desde su navegador, incluso localmente. Por ejemplo, se puede observar el proyecto funcionando fuera de su directorio raíz en la Figura 37.

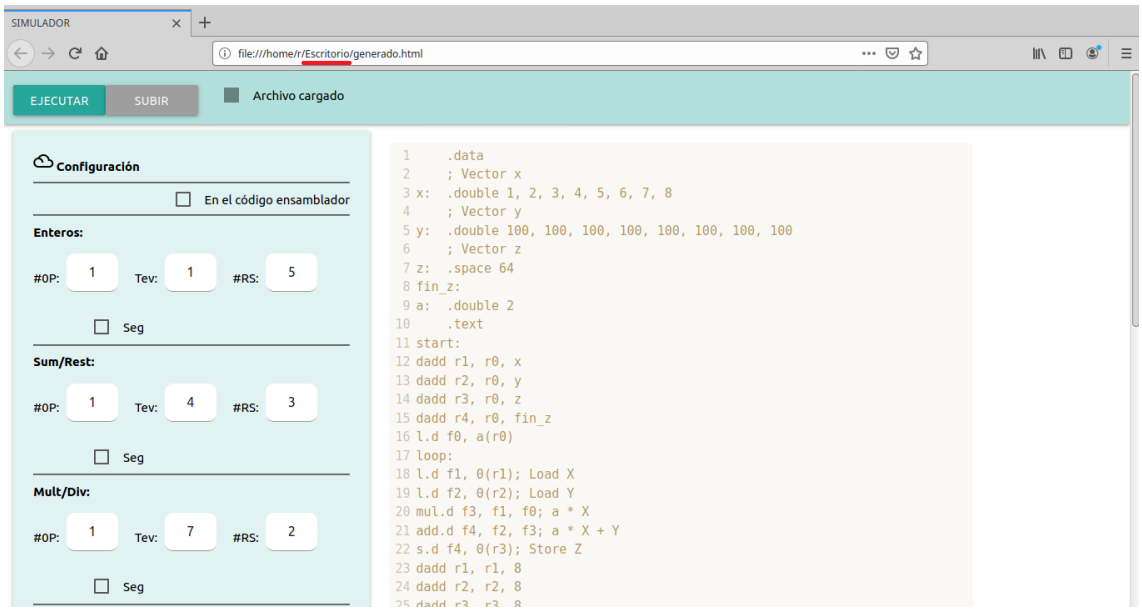


Figura 37: Ejecución del fichero HTML fuera de su directorio raíz

5.6) Pruebas

En esta sección se compara el funcionamiento del simulador de AIC con su interfaz utilizada hasta ahora con la implementación de la interfaz propuesta en este proyecto. Para ello se van a realizar distinas ejecuciones con las mismas configuraciones mediante en las que se observará los diferentes tiempos de respuesta para un mismo código de entrada.

El código que se va a utilizar (ver Figura 38) es un fichero ensamblador preparado para que el simulador realice un número de iteraciones determinado por el usuario. Las iteraciones irán aumentando en cada prueba con el objetivo de medir sus tiempos de ejecución. La configuración empleada será la configuración por defecto, que se muestra en Figura 39.

```
1 .text              ; Comienzo del fragmento de codigo
2
3
4
5         daddi r10,r0,100
6 loop:
7         daddi r10,r10,-1
8         bnez r10,loop
9
10
11
12        trap 0
```

Figura 38: El código ensamblador iterador usado para las pruebas

Configuración

En el código ensamblador ☐

Enteros:
#Op 1 Tev 1 #RS 5 Seg ☐
Sum/Rest:
#Op 1 Tev 4 #RS 3 Seg ☐
Mult/Div:
#Op 1 Tev 7 #RS 2 Seg ☐
MemDatos:
#Op 1 Tev 2 #TL 3 #TE 3 Seg ☐

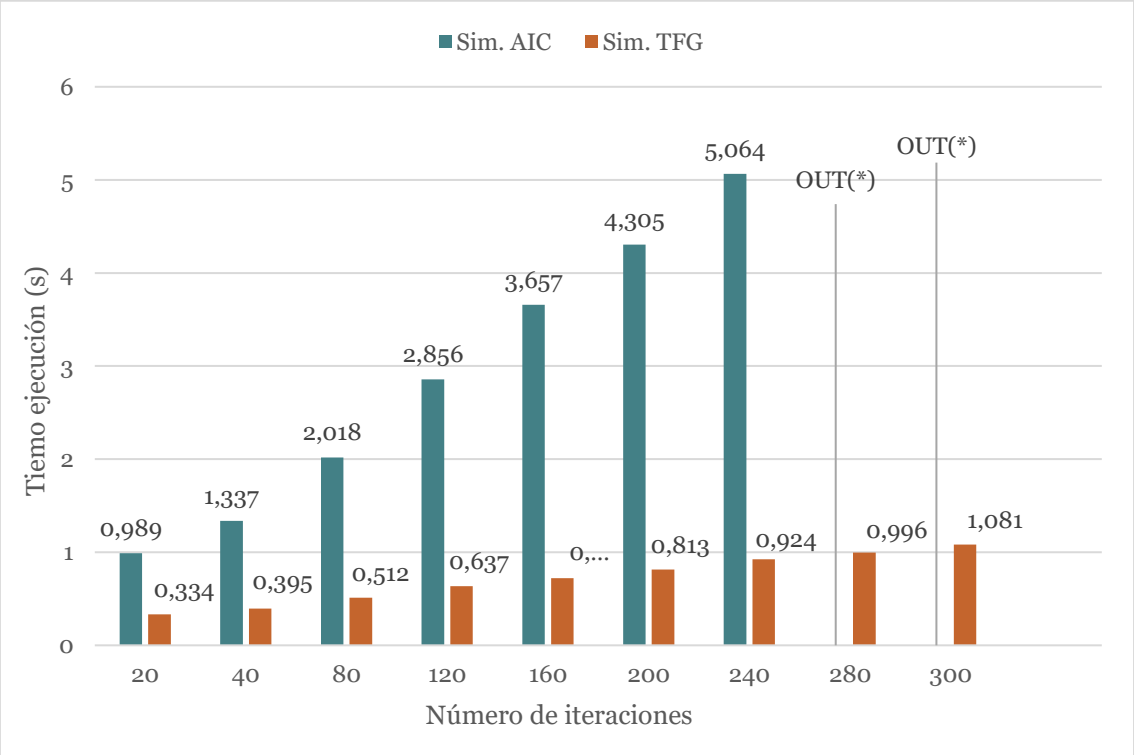
Vías:
I 1 WB 1 C 1

Predictor:
1 bit BTB 2

Otros:
ROB 20 load forwarding ☐

Figura 39: Configuración empleada para las pruebas en ambos simuladores

En la Figura 40, se observan los resultados de cada una de las ejecuciones, teniendo en cuenta el número de iteraciones que realiza el código ensamblador y el tiempo de ejecución.



(*) OUT OF MEMORY: Memoria del navegador desbordada

Figura 40: Comparación de tiempos de ejecución entre simuladores

Como puede observarse, la aproximación con la que se devuelven los datos generados del simulador mediante una variable JSON tal y como se desarrolla en este proyecto, es mucho más eficiente que la generación de ficheros HTML que realiza en el simulador de AIC original. Se consigue hacer un mayor número de iteraciones en mucho menos tiempo e incluso llegando a valores en los que el simulador con la implentación original no es capaz de soportar debido al desbordamiento de la memoria del navegador.

6) Conclusiones

El objetivo principal del proyecto era realizar una interfaz web para un simulador de procesador MIPS. Después de escoger las tecnologías para hacerlo y desarrollar la solución, se llegan a varias conclusiones:

- ➔ Se ha conseguido lo propuesto. El simulador actualmente está integrado completamente en la interfaz. Después de la portabilidad de este a JavaScript, se ha conseguido incluirlo en un front-end sin necesidad de realizar llamadas a ningún tipo de servidor.

- ➔ El resultado obtenido es satisfactorio, puesto que se ha podido dotar a la interfaz de todas las funcionalidades esperadas e incluir alguna más como el salto a ciclos concretos o cambiar el tamaño del cronograma. La aplicación es capaz de cargar y ejecutar ficheros de código ensamblador, pudiendo modificarlas en editor al gusto del usuario, permitiendo también cambiar las configuraciones según el tipo de ejecución que se le quiere pedir al simulador.

- ➔ La aplicación ahora es más liviana, con el simulador integrado, que la anterior, con el simulador y la generación de interfaces que había antes de esta, cuando se generaba un fichero HTML por cada sección de la interfaz que se quería visualizar. Actualmente la aplicación con el simulador incluido en el front-end, ocupa aproximadamente 2,2MB frente a los 54MB que ocupaba una ejecución del código “daxpy” con todos sus ficheros HTML generados. También se ha reducido en gran medida su tiempo de ejecución con la nueva interfaz, mediante la modificación de la salida del simulador a modo de una variable JSON, llegando a poder realizar ejecuciones que la implementación anterior no soportaba por desbordamiento de memoria.

- ➔ Ha resultado acertada la elección de HTML, JavaScript y Materialize CSS. Con estas herramientas se pueden realizar aplicaciones potentes y livianas. Se cargan y ejecutan con rapidez además de tener la opción de incluir plugins o librerías externas, las cuales abundan por la red, permitiendo añadir nuevas funcionalidades o directamente, agregar módulos de código con una función concreta para nuestra aplicación.

En la Tabla 2 se aprecian las prestaciones conseguidas con la nueva implementación de la interfaz para el simulador, en comparación con el resto de los simuladores analizados en el estado del arte.

Tabla 2: Comparativa de simuladores final

Prestación	**WeMIPS**	**MIPS Interp.**	**Visual MIPS**	**Sim. AIC**	**Sim. TFG**
Cronograma	NO	NO	NO	SÍ	SÍ
Tamaño de cronograma	-	-	-	NO	SÍ
Reset	NO	SÍ	SÍ	SÍ (1*)	SÍ (1*)
Paso atrás	NO	NO	SÍ	SÍ	SÍ
Paso adelante	SÍ	SÍ	SÍ	SÍ	SÍ
Múltiples pasos	NO	NO	NO	SÍ (2*)	SÍ (4*)
Editor CodeMiror	SÍ	SÍ	SÍ	NO	SÍ
Tablas de estado	NO	NO	NO	SÍ	SÍ
Tablas de registros	SÍ	SÍ	SÍ	SÍ	SÍ
Tabla dir. Memoria	SÍ	SÍ	SÍ	SÍ	SÍ
Una sola página	SÍ	SÍ	SÍ	NO	SÍ
Código de muestra	SÍ	NO	SÍ	SÍ	SÍ
Configuración del procesador simulado	NO	SÍ (3*)	NO	SÍ	SÍ
Un sólo archivo	NO	NO	NO	SÍ	SÍ

(1*) Es posible volver al ciclo inicial pulsando sobre él en el cronograma.

(2*) Hasta en bloques de ± 5 pasos.

(3*) Únicamente cambiar la frecuencia de la CPU a simular.

(4*) Hasta bloques de ± 10 pasos.

7) Relación con los estudios cursados

En el desarrollo y la implantación del proyecto, se han utilizado diferentes conocimientos relacionados con el Grado en Ingeniería Informática impartido por la Universidad Politécnica de Valencia:

- ➔ Para el diseño de la interfaz gráfica de la aplicación, se han utilizado conocimientos de diseño de interfaces orientadas al usuario, impartidas en las clases de Interfaces Persona Computador.

- ➔ En la implementación de la lógica de la aplicación, se ha utilizado el lenguaje de scripting para el lado del cliente, JavaScript, cuyos conocimientos del lenguaje se han impartido en la asignatura de Tecnologías de Sistemas en Red. Además, se han obtenido conocimientos adicionales sobre el mismo en la plataforma orientada al desarrollo web W3Schools [RD2|99-20] entre otros [JC|17] [JD|12].

- ➔ JQuery: la librería complementaria a JavaScript para el desarrollo de front-end. Los conocimientos necesarios se han aprendido de forma autodidacta, tomando referencias [RM|11] [RD5|99-20], así como de la propia API de la librería [JQF|20]. Estas competencias también se imparten en la asignatura optativa de la Universidad, dedicada al desarrollo web.

- ➔ CSS: Los conocimientos adquiridos sobre el lenguaje de hoja de estilos en cascada CSS, se han obtenido mediante la búsqueda en la red, así como también consultando en la web de W3Schools [RD3|99-20] entre otros documentos y portales [JC|17] [JD|12] [CC|20]. Dichos conocimientos también se imparten en la asignatura de desarrollo web.

- ➔ El framework Materialize CSS, el cual es un marco de trabajo de diseño para la parte visual de la aplicación, se ha aprendido de forma autodidacta revisando su página oficial [MC|14-20] y revisando tutoriales on-line [TP|20].

- ➔ El módulo del editor de código de CodeMirror ha sido utilizado revisando su propia API en su web oficial [CM|20].

- ➔ Los conocimientos sobre la manera de trabajar con datos en formato JSON han sido adquiridos por la utilización de dicho formato en diferentes asignaturas de la Universidad Politécnica de Valencia.

8) Trabajo futuro

Como funcionalidades añadidas a la aplicación, en un futuro podrían realizarse diferentes implementaciones que la dotarían de características interesantes a la hora de proveerlas al usuario, entre ellas podrían darse:

- ➔ La posibilidad de añadir una nueva pestaña en la interfaz destinada a la observación de las estadísticas de ejecución, una vez se haya cargado y ejecutado un programa. Estos datos serían interesantes para el usuario a la hora de comparar diferentes maneras de ejecutar un mismo código o simplemente para observar los recursos que consume el simulador a la hora de ejecutar trozos de código con diferente carga computacional pero la misma configuración.

- ➔ Sería interesante la opción de poder exportar a un formato externo algunas tablas como las de operadores y configuración, junto al cronograma, el código ejecutado y la pestaña de configuración, para su posible impresión o almacenamiento de cada una de las ejecuciones realizadas. Para ello, se podría incluir algún plugin que se ocupase de extraer, tratar y colocar los datos mencionados en un archivo *.doc* o *pdf*, por ejemplo. Hay varias opciones para realizarlo con un pluggins para JQuery o JavaScript puro [JQS|15] [PM|20].

9) Bibliografía

[EO|14] Eric Wooley, Ortal Yahdav. Simulador “WeMIPS”:

https://github.com/ericwooley/WeMips

(02/2014)

[DY|18] Danny Qiu, Yiteng Guo. Simulador “MIPS Interpreter”:

https://github.com/dannyqiu/mips-interpreter

(11/2018)

[WB|18] Wei Jian Wong, Ben Withers. Simulador “Visual MIPS”:

https://github.com/VisualMIPS/visualmips.github.io

(09/2018)

[DJ|18] @Davidjguru. Single Page Application: Un viaje a las SPA a través de Angular y Javascript.

https://medium.com/@davidjguru/single-page-application-un-viaje-a-las-spa-a-trav%C3%A9s-de-angular-y-javascript-337a2d18532
(12/2018)

[JV|17] Jose Mª Baquero García. Single-Page Application, todo un website desde única página.

https://www.arsys.es/blog/programacion/diseno-web/spa-unica-pagina/

(04/2017)

[JC|17] Juilie C. Meloni. Programación HTML5, CSS3 y JavaScript. ANAYA

(05/2017)

[KP|19] Kiko Palomares. ¿QUÉ es una web SPA?

https://kikopalomares.com/que-es-una-web-spa-single-page-application/

(01/2019)

[RM|11] Rebecca Murphey. Fundamentos de JQuery.

http://ferko.byethost31.com/352da3_Fundamentos_de_jQuery.pdf
(2011)

[MC|14-20] Portal de Materialize CSS. Uso del framework y sus componentes

https://materializecss.com/

(2014-2020)

[JD|12] Juan Diego Gauchat. El gran libro de HTML5 CSS3 y JavaScript. MARCOMBO. (2012)

[AT|06] Aníbal de la Torre. Lenguajes del lado servidor o cliente.

http://www.adelat.org/media/docum/nuke_publico/lenguajes_del_lado_servidor_o_cliente.html

(2006)

[MZ1|20] Mozilla MDN web docs. Lista de Elementos HTML5.

https://developer.mozilla.org/es/docs/HTML/HTML5/HTML5_lista_elementos

(actualizado 01/2020)

[MR|19] MediaRoom solutions. 10 tendencias de desarrollo web indispensables para 2019

https://www.mediaroomsolutions.es/blog/tendencias-desarrollo-web-2019/

(2019)

[RD1|99-20] Refsnes Data. W3Schools HTML tutorial.

https://www.w3schools.com/html/

(1999)

[RD2|99-20] Refnes Data. W3Schools JavaScript tutorial.

https://www.w3schools.com/js/

(1999-2020)

[RD3|99-20] Refnes Data. W3Schools CSS tutorial.

https://www.w3schools.com/css

(1999-2020)

[MZ2|19] Mozilla MDN web docs. FileReader.

https://developer.mozilla.org/es/docs/Web/API/FileReader
(actualizado 03/2019)

[RD4|99-20] Refnes Data. W3schools Colapsible Element.

https://www.w3schools.com/howto/howto_js_collapsible.asp

(1999 -2020)

[RD5|99-20] Refnes Data. W3schools JQuery tutorial.

https://www.w3schools.com/jquery/

(1999-2020)

[CC|20] Chris Coyier. A Complete Guide to the Table Element.

https://css-tricks.com/complete-guide-table-element/

(actualizado 05/2020)

[JQS|15] JQueryScript.net Export Html to word document with images using jQuery Word Export plugin.

https://www.jqueryscript.net/other/Export-Html-To-Word-Document-With-Images-Using-jQuery-Word-Export-Plugin.html

(actualizado 02/2015)

[EM|17] Elia Maino. Webassembly: calling C functions from Javascript with emscripten.

https://medium.com/@eliamaino/calling-c-functions-from-javascript-with-emscripten-first-part-e99fb6eedb22

(07/2017)

[TP|20] TutorialsPoint. Materialize – Grids.

https://www.tutorialspoint.com/materialize/materialize_grids.htm

(actualizado 2020)

[CM|20] CodeMirror. Editor de texto implementado en JavaScript.

https://codemirror.net/doc/manual.html

(actualizado 2020)

[JQF1|20] JQuery Fundation. Librería JavaScript.

https://api.jquery.com/

(actualizado 2020)

[PM|20] Pdfmake. Plugin JavaScipt para exportar a pdf.

https://pdfmake.github.io/docs/getting-started/client-side/

(actualizado 2020)

[WP|13] Wikipedia. Microarchitecture simulation

https://en.wikipedia.org/wiki/Microarchitecture_simulation

(actualizado 07/2013)

[EMS|15] Emscripten.org. About Emscripten.
https://emscripten.org/docs/introducing_emscripten/about_emscripten.html

(actualizado 2015)

[MZ3|19] Mozilla MDN web docs. element.addEventListener()

https://developer.mozilla.org/es/docs/Web/API/EventTarget/addEventListener

(actualizado 2019)

[JQF2|20] JQuery Foundation. JQuery click event.

https://api.jquery.com/click/

(actualizado 2020)

[WD|10] Kayce Basques, Pete LePage. Read files in JavaScript

https://web.dev/read-files/

(2010, actualizado 2020)

[BO|20] Bootstrap, página oficial.

https://getbootstrap.com/docs/4.5/getting-started/introduction/

(actualizado 2020)

Glosario de términos

- **API**: conjunto de funciones, procedimientos y métodos que ofrece alguna librería con el objetivo de ofrecer una capa de abstracción al programador a la hora de ser utilizadas.

- **Aplicación web**: Conjunto de funcionalidades y herramientas que los usuarios pueden utilizar mediante el uso de un navegador. Es decir, son programas codificados en lenguajes que un navegador web puede interpretar y ejecutar.

- **Back-end:** parte trasera de una aplicación invisible para el usuario, encargada de gestionar los datos y la lógica necesarios para el intercambio de datos con un un servidor o una base de datos.

- **CDN**: red de computadoras que contienen copias de datos, colocados en varios puntos de una red con el fin de dar acceso a los datos a los usuarios. Un cliente accede a una copia de la información realizando una solicitud.

- **Ciclo**: Serie de estados o fases por las que pasa un determinado evento o fenómeno, en nuestro caso son cada uno de los pasos que realiza la simulación del procesador.

- **DOM**: API definida para representar e interactuar con cualquier documento HTML. Es un modelo del documento que lo representa como un objeto árbol de nodos, donde cada nodo representa un elemento de la página. Estos elementos pueden crearse, eliminarse, moverse o modificarse.

- **Emscripten**: Transcompilador, o compilador source-to-source, procesa código de bajo nivel generado al compilar código C o C++, como salida, devuelve un archivo en lenguaje JavaScript que puede se procesado por un navegador web.

- **Ensamblador**: Lenguaje de programación de bajo nivel, consiste en un conjunto de instrucciones que representan las instrucciones básicas para procesadores, microcontroladores y otros tipos de piezas hardware programables.

- **Evento**: un evento es una acción que es detectada por un programa, que puede provocar algún cambio o ser ignorado. Normalmente los eventos son manejados con el fin de realizar alguna acción en el programa en ejecución. En nuestro caso los eventos han sido de carga de ficheros y eventos de “click” sobre algún elemento de la parte visual.

- **Framework**: Es un conjunto estandarizado de herramientas, módulos de código y procedimientos establecidos para enfocar el desarrollo de alguna determinada tarea. En este proyecto se ha utilizado un framework CSS para diseñar la parte visual de la página.

- **Front-end**: consiste en la transformación de los datos de una aplicación de modo que sean utilizables en una interfaz gráfica para facilitar la interacción del usuario con ellos. Referido a una web, es la parte visual con la que interactúa el cliente.

- **Hoja de estilos**: conjuntos de instrucciones que se asocian a los archivos de texto y se ocupan de los aspectos de formato y de presentación de los contenidos.

- **Interfaz**: En nuestro caso, la propia parte visual de la aplicación, en ella se observan los diferentes elementos con los que se interactúa, actuando como la puerta de interacción con la capa lógica de la aplicación.

- **JSON**: Formato de texto sencillo para el intercambio de objetos y datos, es un subconjunto de la notación de objetos JavaScript.

- **Layout**: esquema que representa la distribución de los elementos de un diseño. A partir del layout se empiezan a desarrollar los elementos específicos.

- **Librería**: conjunto de funciones codificadas en un determinado lenguaje de programación, preparadas para ser importadas y utilizadas por el desarrollador.

- **Procesador**: Pieza de hardware de un dispositivo programable que interpreta las instrucciones de algún programa informático mediante la realización de las operaciones básicas aritméticas, lógicas y de entrada/salida del sistema.

- **Simulador**: Entorno que permite la reproducción total o parcial de un sistema. En este caso, se trata de una réplica implementada a código de un procesador superescalar.

- **SPA**: Siglas de Single Page Application. Se trata de una aproximación de desarrollo web mediante la cual en una sola página se puede implementar una aplicación funcional. La web varía junto a la interacción del usuario.

Anexo 1: Dependencias del proyecto y estructura

En este apartado se describen las dependencias de archivos que necesita la aplicación para poder ejecutarse y visualizarse correctamente. Estos se indican en el archivo *Index.html*, el cual es el fichero principal de la aplicación. Algunos dentro del campo <*head*> de la página, mientras que otras, se ha decidido añadirlas al final del campo <*body*> a modo de buenas prácticas para que esta cargue más rápidamente. Las hojas de estilo se han referenciado con la etiqueta <*link*> mientras que los archivos JavaScript que albergan la funcionalidad se referencian con la etiqueta <*script*>.

Los archivos que necesita este proyecto para su funcionamiento son:

- ➔ jQuery: la librería de JavaScript se ha incluido vía CDN:

```
<script type="text/javascript"

src="https://code.jquery.com/jquery-3.1.1.min.js"></script>
```

- ➔ CodeMirror: el editor de texto incrustable, consta de tres archivos, en uno se incluye el código JavaScript que provee de su funcionalidad, mientras que los dos últimos son la propia hoja de estilos del editor y el tema que he decidido cargar. También se incluyen vía CDN:

```
<script
src="https://cdn.jsdelivr.net/npm/codemirror@5.55.0/lib/codemirror.js"
>

</script>

<link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/codemirror@5.55.0/lib/codemirror.cs
s">

<link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/codemirror@5.55.0/theme/duotone-
light.css">
```

- ➔ Materialize CSS: El framework CSS se ha descargado y se ha colocado en los directorios */js* y */css* del proyecto, junto al código JavaScript y CSS de la aplicación. Se ha decidido hacerlo así para poder borrar de su archivo *.css* aquellos estilos que no interesaban:

```
<script src="./js/materialize.js"></script><link rel="stylesheet"
href="./css/materialize.css">
```

➔ Style.css: Este documento es el fichero de hoja de estilos de la aplicación, junto a Materialize CSS, definen cómo se van a observar los elementos de esta. Se sitúa en el directorio */css*.

```
<link rel="stylesheet" href="css/style.css">
```

➔ App.js: En este fichero queda implementada toda la lógica de la aplicación. Es el encargado de manejar los eventos, ejecutar funciones para modificar el *Index.html* cuando sea necesario, instanciar CodeMirror y llamar y obtener un resultado del fichero *ejecutar-ensamblador.js*. Es uno de los ficheros que se ha decidido cargar al final del <*body*>. Se sitúa en el directorio */js*.

```
<script type="text/javascript" src="js/app.js"></script>
```

➔ Ejecutar-ensamblador.js: En este fichero se recoge la funcionalidad para, dado un código ensamblador, ejecutarlo y devolvernos el resultado. Será llamado por *app.js* cuando el usuario decida ejecutar un código. También se carga al final del <*body*> para agilizar la carga de la página. Se sitúa en el directorio */js*. Es el propio simulador del procesador compilado en JavaScript mediante Emscripten.

```
<script type="text/javascript"
src="js/ejecutar-ensamblador.js"></script>
```

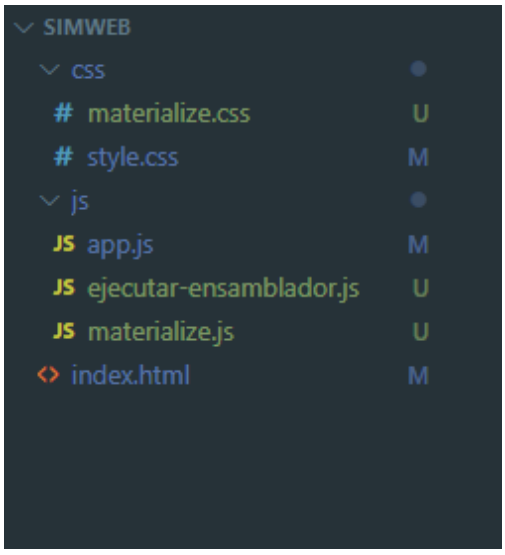


Figura 41: Árbol de directorios del proyecto

Anexo 2: El grid en Materialize CSS

El grid, o cuadrícula, es la manera que tiene Materialize CSS para organizar el espacio visual para los elementos de una página.

En este anexo se detalla brevemente cómo funciona el mismo, con el objetivo de reforzar las explicaciones de los apartados de diseño del documento.

Básicamente la idea del grid de Materialize CSS es una cuadrícula de filas y columnas que se reajustan de manera adaptativa al cambiar el tamaño de la página o al cambiar de tipo de dispositivo.

A la hora de definir una etiqueta <*div*> en HTML, o cualquier otro elemento, siendo estos los más comunes, se le pueden añadir ciertas cadenas a la lista de clases (atributo class HTML) de modo que Materialize CSS las detecte y las interprete para representar la página.

Los elementos más importantes de dicho sistema grid son:

- ➔ Row: especifica un contenedor fila sin padding, es decir, sin relleno en los bordes. Esta clase es obligatoria para conseguir que los elementos que haya en su interior sean completamente adaptables al entorno.

- ➔ Col: especifica un contenedor columna, suele aparecer con alguna subclase concatenada que indica el número de columnas que ocupará dicho contenedor.

Las subclases comunes a row y col y son:

Subclases de s1 a s12: Definen el tamaño que ocupará dicho elemento en dispositivos con pantallas pequeñas, típicamente dispositivos smartphone, siendo s1 el tamaño más pequeño y s12 todo el espacio disponible.

Subclases de m1 a m12: Definen igualmente el tamaño que ocupará el elemento con la diferencia de que esta vez se tratará de dispositivos con pantallas medianas. Se trata de dispositivos tipo tablet. Siendo m1 el tamaño más pequeño y m12 todo el espacio.

Subclases de l1 a l12: Definen el tamaño que ocupará el elemento en dispositivos con pantallas grandes, es decir, ordenadores y laptops.

Si solo se especifica un tipo de subclase, como es el caso de este proyecto, en el cual sólo se han utilizado sublases de tipo s1 a s12, Materialize CSS calculará el tamaño de esa subclase para el resto de los dispositivos.

- ➔ Container: La clase container no es una parte estricta de las clases del grid, se utiliza para obtener un espacio en cual el contenido se muestra en el centro de la página. La clase container establece el ancho del contenido en un 70% del ancho de la página.

Anexo 3: Las clases y reglas CSS utilizadas

A parte de los estilos y componentes que vienen incluidos en Materialize CSS, se han añadido algunas reglas y estilos a parte en el documento *style.css*. Este apartado queda para mencionar dichos estilos y explicar a qué elementos van asociados.

- ➔ El *input-text* para cambiar el tamaño del cronograma, tendrá la clase css *"redondeado"*:

```
.redondeado{
      height:  38px; //Altura fijada a 38 píxeles
      text-align: center; // El texto se alineará al centro
      border-radius: 5px; // Los bordes tendrán un radio de 5px
      }
```

- ➔ Los datos de las tablas. Según el tipo de dato o el estado de dicho dato a la hora de visualizarse en ellas, su texto aparecerá de un color u otro. Estas clases se leen del propio resultado obtenido del simulador, y se van aplicando sobre la marcha. Las clases mencionadas son:

```
.entry-old { color: darkgray;} // Clases y colores asignados a cada una
.instruc-2 {color: darkcyan}
.instruc-3 {color: green}
.instruc-4{ color: darkorange}
.entry-busy.instruc-2{ color: darkcyan}
.entry-busy.instruc-3 {color: green}
.entry-busy.instruc-4{ color: darkorange}
.exception-phase{color: red}
```

- ➔ Los botones de maximización y minimización. Estos tendrán una clase CSS asociada para darles un aspecto casi redondo.

```
.max{ border-radius: 10px;}  //Radio de 10px en los bordes para
redondearlos
.max2{ border-radius: 10px; }
```

➔ Los botones de los desplegables de las tablas tienen una clase CSS “collaps” asociada para darle el aspecto de elemento desplegable.

```
.collaps {
      background-color: #eee; //Color gris claro para el fondo del botón
      color: #444; //Color gris oscuro para el texto
      cursor: pointer; // Cuando el cursor esté encima del botón será tipo pointer
      padding: 10px; // El espacio alrededor del texto será de 10 px
      height: 100%; // El botón ocupará el 100% del alto y del ancho disponible
      width: 100%;
      border: none; // No tendrá bordes definidos
      text-align: left; //El texto se alineará a la izquierda
      outline: none; // El botón no tendrá borde outline
      font-size: 16px; //Tamaño del texto
      }
```

➔ Las celdas de PC, instrucción y las celdas de ciclo del cronograma tendrán sus propias clases CSS para diferenciarlas del resto:

```
.pc-td{ background-color: lightblue;} // Color de fondo
.instruc-td{ background-color: lightblue;} // Color de fondo
.ciclo{
      cursor: pointer; // El cursor encima será de tipo pointer
      color: darkblue; // Color del texto
      min-width: 50px; // Ancho mínimo de la celda
      }
```

➔ Las tablas normales (las tablas diferentes al cronograma) tendrán su propia clase CSS:

```
table.claseTabla2{
      table-layout: auto;
      //La anchura de las columnas depende de su contenido
      width: 100%; // Se utilizará el máximo posible del ancho disponible
      border-collapse: collapse; // Los bordes de las celdas estarán unidos
   }
```

➔ Para que las celdas de la cabecera del cronograma permanezcan fijas, se ha añadido la siguiente regla CSS, aplicándola sobre su ID: *cronoTable*

```
#cronoTable thead th {
      position: sticky; // Posición fija respecto del scroll
      top: 0;
      // La distancia a partir de la cual se fija respecto del "top"
      background-color: #f2f2f2; //El color de fondo de las celdas
      }
```

➔ Además de estos estilos, se han añadido reglas de estilo a las celdas de las tablas, para añadir algunas propiedades:

```
th {
      text-align: center; // El texto aparecerá en el centro
      height: 20px;     //La altura de la celda se fija a 20px
      }
td {text-align: center; } // El texto se alineará al centro
td, td {
            border: 2px solid #ddd; //Borde entre td y td de 2px de ancho
            border-collapse: collapse; //No habrá espacio entre bordes
      }
th, th {
      border: 2px solid #ddd; //Borde entre th y th de 2px de ancho
      border-collapse: collapse; //No habrá espacio entre bordes
      }
tr:nth-child(even) {
      background-color: #f2f2f2;
      // Se aplica el estilo zebra a las filas pares
      }
```