Escuela Politécnica Superior de Jaén

Grado en Ingeniería Informática

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior de Jaén

Trabajo Fin de Grado

Desarrollo de interfaz web para un simulador de procesador DLX

Alumno: Antonio Mudarra Machuca

Tutor: Francisco Charte Ojeda

Dpto: Informática

UNIVERSIDAD DE JAÉN

D. Francisco Charte Ojeda, tutor del Trabajo Fin de Grado titulado: **Desarrollo de interfaz web para un simulador de procesador DLX**, que presenta Antonio Mudarra Machuca, autoriza su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, junio de 2022

El estudiante	El tutor
Firmado digitalmente por MUDARRA MACHUCA ANTONIO - 76665902W	CHARTE OJEDA FRANCISCO - 25996290B Firmado digitalmente por CHARTE OJEDA FRANCISCO - 25996290B Fecha: 2022.07.04 07:26:24 +02'00'
Antonio Mudarra Machuca	Francisco Charte Ojeda

Agradecimientos

Quiero expresar mi agradecimiento a todas esas personas que de una forma u otra me han apoyado en mis estudios y en este proyecto.

En primer lugar, a mi tutor Francisco Charte, que me motivó a estudiar código ensamblador e hizo que amara esta rama de la informática.

A mi familia por apoyarme desde pequeño y motivarme a estudiar lo que realmente deseaba. En especial a mi madre y mi padre, gracias por vuestro apoyo incondicional y por mostrarme el camino cuando no he podido encontrarlo. A mis dos primas que han sido mis hermanas a lo largo de mi vida y a mis dos queridas abuelas, quienes se fueron antes de ver terminados mis estudios.

A mis amigas Ana y Julia, que me animaron a continuar y a disfrutar de los pequeños momentos de la vida.

Por último, a todos los magníficos compañeros que he tenido en estos años y con los que he compartido buenos y malos momentos.

Muchas gracias.

Antonio Mudarra Machuca.

Tabla de contenidos

Lista de figuras

Lista de tablas

Capítulo 1

Introducción y motivación

En este capítulo se especificará una introducción y una estructura de contenidos inspirados en los criterios y recomendaciones que recogen en libros como “Criterios generales para elaboración de proyectos de Sistemas de información **UNE 157801; 2007**” [1], así como la guía “**PMBOK**” [2] estos libros y guías nos ayudaran mucho en la planificación del proyecto.

En este documento se utilizarán acrónimos y citas, estos estarán descritos en los apartados de “Glosario” y “Bibliografía” al final del documento.

1.1. Introducción

El principal propósito de este proyecto ha sido la creación de un simulador moderno, pensado para una enseñanza de la arquitectura de computadores DLX [3] (pronunciada “Deluxe”), con el fin de mostrar el funcionamiento interno de un microprocesador y cómo funciona mediante instrucciones máquina.

Los simuladores DLX que actualmente se usaban estaban desfasados o tenían grandes carencias, además presentaban incompatibilidad incluso con los sistemas operativos modernos por temas de retro compatibilidad, imposibilitando su ejecución.

Esto se produjo en el año 2007 con el lanzamiento de Windows Vista, por el avance tecnológico se perdió compatibilidad con programas desarrollados para sistemas operativos antiguos, esta es la razón por la que usamos máquinas virtuales para

ejecutar sistemas operativos antiguos y poder usar el simulador **WinDLX** desde la máquina virtual en un ordenador actual.

Es por ello que ha nacido este proyecto, pensado para las nuevas tecnologías y herramientas, diseñando e implementando un simulador web *online* y multiplataforma que pueda ejecutarse en distintos sistemas operativos.

Este *software* integrará de una forma clara y simple todas las herramientas necesarias, servirá para comprender el funcionamiento de un microprocesador y que procesos se llevan dentro, y de qué forma ejecutan las instrucciones programadas.

1.2. Motivación

La principal motivación es la de facilitar el aprendizaje de los distintos tipos de arquitecturas de computadores, comprender el funcionamiento interno de un microprocesador, así como el aprendizaje de lenguajes de bajo nivel (ensamblador) como es el lenguaje DLX [3].

Además, aunque existen herramientas (simuladores) no todas tienen todas las utilidades que deseamos usar, es por ello que debíamos usar herramientas de terceros para conversiones, documentación o desarrollar el código en un IDE externo.

Aprender a programar con instrucciones máquina permite crear programas y código mucho más eficiente, pues se tiene un conocimiento real de cómo estas instrucciones se ejecutan en el procesador y podemos ver de una forma más intuitiva cómo se ejecuta nuestro código.

Este conocimiento es de vital importancia si lo que se desea desarrollar es software embebido, el cual se ejecuta en máquinas con muy pocos recursos, así como otros proyectos que requieran programas de respuesta real.

1.3. Descripción

THUMDER es una aplicación multiplataforma pensada para ejecutarse de dos maneras, mediante una interfaz web o mediante una aplicación de escritorio.

Entre otras funcionalidades, se ofrecen las siguientes:

- Tabla y editor de Memoria.
- Tabla y editor de Registros.
- Tabla del código con representación de las instrucciones.
- Visualización del estado del cauce de ejecución (*pipeline*).
- Visualización de las etapas.
- Gestor de ficheros.
- Editor de código DLX.
- Representar estadísticas.
- Calculadora y otras utilidades.

1.4. Objetivos principales

Este software debe permitir el procesamiento de programas escritos en lenguaje ensamblador DLX [3] mostrando la información y datos relevantes que se producen en la CPU, este software emula el comportamiento real de las partes del cauce de ejecución (*Pipeline*), el banco de registros, la memoria, los datos de entrada salida y estadísticas.

Además, debe ser modificable el tamaño de la memoria de la máquina y el número de los núcleos de cálculo para suma, multiplicación y división de números en punto flotante (`fadd`, `fmult`, `fdiv`) y su latencia.

Se deberán incluir ejemplos de código DLX funcionales que permitan su estudio.

1.5. Objetivos secundarios

Como objetivos secundarios, se deberá implementar un gestor de ficheros para almacenar el código que se desarrolla de una forma cómoda e intuitiva, así como un servicio de autenticación para gestionar el sistema de ficheros.

Otro de los objetivos secundarios es que el editor resalte las palabras clave como instrucciones, incluya una breve descripción de la operación y otro objetivo que permita autocompletado.

Otro de los objetivos secundarios sería la sección de estadísticas, ya que **WinDLX** [4] nos permite visualizar todos los datos relevantes de la ejecución y métricas para saber rápidamente el número de riesgos tales como leer después de escribir (RAW), escribir después de leer (WAR), escritura tras escritura (WAW) o los stalls que se producen, así como otros tipos de datos útiles.

Por último, debemos tener una sección con las distintas utilidades que nos permitan representar un dato en distintos formatos.

- Sistema gestor de ficheros.
- Sistema editor de ficheros con documentación, resaltado de palabras clave, autocompletado y utilidades.
- Documentación DLX.
- Calculadora de datos binarios, hexadecimales, octales, etc.
- Estadísticas.

1.6. Estructura de la memoria

La estructura de esta memoria se ha inspirado en el estándar UNE 157801; 2007 [5], aunque esta norma es para proyectos de tipo “Ingeniería civil”, por lo que solo ha servido de guía.

- **Capítulo 1**: describimos el problema que debemos abordar y dar solución, así como las propuestas para solucionar dicho problema.

- **Capítulo 2**: explicación detallada de la planificación y desarrollo de las soluciones que debemos abordar, para ello se explican y plantean las distintas técnicas de Ingeniería del software.

- **Capítulo 3**: en este capítulo se presentan tanto los requisitos funcionales como los no funcionales del proyecto.

- **Capítulo 4**: el objetivo del capítulo es documentar el diseño del proyecto, el diseño se ha creado a partir de los requisitos del proyecto, esto son diagramas que abstraen la lógica de la implementación.

- **Capítulo 5**: en el quinto capítulo se plantean las distintas alternativas de *frameworks* y librerías, se elegirá la que mejor se adapte a nuestro proyecto en base, requisitos, y las necesidades planteadas en los diagramas. En este capítulo explicamos las tecnologías, los ciclos de vida y de ejecución de los *frameworks* y las arquitecturas que sé usan.

- **Capítulo 6**: en el sexto capítulo se desarrolla el proyecto, pues ya hemos definido la arquitectura y los diagramas de clases, en función de esta arquitectura, se desarrolla el proyecto usando las herramientas y *frameworks*. En este capítulo se explican las tecnologías, arquitecturas y decisiones que se han tomado en el desarrollo.

- **Capítulo 7**: en el séptimo capítulo se incluye y explican las pruebas que se han creado para el proyecto, estas pruebas mantienen la compatibilidad frente a modificaciones que causen respuestas erróneas, así como comprobar que el comportamiento de la aplicación es el esperado.

- **Capítulo 8**: en el último capítulo se incluye una conclusión del proyecto y las posibles líneas de investigación o mejoras que pueden surgir del desarrollo de este proyecto.

- **(Capítulo 8 manuales, apéndices, definiciones y abreviaturas)**: se incluyen los apéndices para la comprensión de la memoria, los manuales para el uso e instalación de la aplicación en sus distintas versiones, las definiciones y abreviaturas para aclarar los distintos términos.

Capítulo 2

Especificación del trabajo

2.1. Descripción de la situación de partida

2.1.1. Descripción del entorno actual

En la actualidad existen múltiples herramientas que nos permiten estudiar el funcionamiento interno de un microprocesador y de la arquitectura DLX, pero estas herramientas presentan ciertas carencias o incompatibilidades con los sistemas operativos modernos. Lo que nos obliga a usar incómodas herramientas de simulación de sistemas operativos antiguos.

2.1.2. Resumen de las deficiencias y carencias identificadas

Podemos encontrar ciertas deficiencias en los otros simuladores, los que he estudiado son **WinDLX** [4], *OpenDLX* [6] y *DLX* [7]

Muchos de estos simuladores no cuentan con editor de código propio, sino que cargan un fichero "`.s`" e interpretan este código. Otros simuladores sí cuentan con editor, pero son muy simples y no cuenta con auto completado o documentación, sino que solo es un editor de texto plano, ese es el caso de *OpenDLX* [6] o DLX [7].

Desarrollar código ensamblador es una tarea muy costosa y complicada, por lo que comprender qué instrucciones son las más adecuadas y qué hace cada una es muy

útil, esta tarea no se encuentra en ningún simulador por lo que podríamos considerarla como una carencia.

Muchas de estas herramientas no cuentan con documentación propia de cómo funciona la arquitectura o sus instrucciones, por lo que consideramos una carencia.

Otra de las carencias más significativas es un sistema de gestión de ficheros, normalmente los simuladores al ser solo aplicaciones de escritorio gestionas tú mismo los ficheros con el sistema operativo, pero en nuestro caso al ser una aplicación web lo más cómodo será que proporcionemos un sistema propio de gestión de ficheros.

2.2. Restricciones

El TFG es una asignatura que tiene asignados 12 créditos, cada crédito equivale a 25 horas, por lo que el total de horas del desarrollo del proyecto será de 300 horas, incluyendo todas las etapas del ciclo de desarrollo, exceptuando el mantenimiento, este será mínimo y no supondrá nuevas mejoras en un futuro, por lo que tenemos una restricción de tiempo.

Aunque esta restricción no se aplicará a este proyecto, pues se ha alargado el desarrollo por mucho tiempo por pequeñas mejoras de estabilidad y usabilidad.

Contamos con distintos equipos y sistemas operativos, entre ellos Windows 10, Ubuntu 20.04 y macOS (BigSur y Monterrey), por lo que no consideramos una restricción el equipamiento, aunque sí consideraremos una restricción el desarrollo para sistemas operativos distintos a los mencionados o versiones anteriores, es decir, no realizaremos un control exhaustivo en múltiples máquinas con distintas configuraciones por ser muchas las posibilidades y combinaciones que puedan desencadenar fallos.

2.3. Metodología de desarrollo software

En la actualidad la forma de desarrollar software ha cambiado mucho, podemos clasificar las distintas metodologías en dos grandes grupos, en **metodologías tradicionales** o **metodologías ágiles** [8].

2.3.1. Metodologías - Tradicionales

- **Cascada - Ventajas**
  Es fácil de implementar, solo hay que seguir los pasos, recursos mínimos.

- **Cascada - Desventajas**
  No puede volver atrás, por lo que un error de diseño es difícil de resolver.
  Además, si el cliente no tiene una idea clara puede complicar mucho el desarrollo, así como si es una aplicación que puede crecer con el tiempo.

- **Prototipado - Ventajas**
  Los usuarios tienen un modelo que pueden analizar y dar *feedback* a los desarrolladores.
  Permite el desarrollo de una idea cambiante o poco clara.
  El usuario suele participar de forma activa en el desarrollo lo que reduce la incertidumbre del producto deseado.

- **Prototipado - Desventajas**
  Se suele confundir el prototipo con el producto por lo que los clientes creen que ya está finalizado.
  Se puede perder la idea y el tiempo en aspectos no funcionales en una demostración de un prototipo.
  Requiere la participación activa del cliente.

- **Espiral - Ventajas**
  Mezcla los dos modelos (cascada y prototipado), añadiendo un análisis de riesgos.
  Divide el proceso en cuatro etapas (planificación, análisis de riesgo, desarrollo de prototipo y evaluación del cliente).
  Permite comprobaciones periódicas enfocadas al riesgo.
  Conocimiento de los costes, recursos y calidad del producto.

- **Espiral - Desventajas**
  Gran esfuerzo de gestión, la toma continua de decisiones alarga el desarrollo, es costoso para proyectos pequeños

- **Incremental - Ventajas**
  Es como el modelo en cascada, pero reduciendo el riesgo en cada incremento,

esto permite limitar el coste en caso de un error en la planificación.
Permite dar entregas parciales al cliente correspondientes al incremento.

- **Incremental - Desventajas**
  Requiere una planificación cuidadosa del incremento y conocer las partes anteriores que pueden dar conflictos.
  Requiere metas claras para conocer el estado del proyecto.
  Este modelo no se recomienda para sistemas de respuesta en tiempo real, que requieran niveles de seguridad altos, etc.

- **Diseño rápido de aplicaciones (RAD) - Ventajas**
  Recomendable para pequeños equipos. Se entregan las partes que mayor prioridad tienen para el cliente.
  Al basarse en modelado se reduce el riesgo de defectos.

- **Diseño rápido de aplicaciones (RAD) - Desventajas**
  No todos los desarrollos son compatibles con RAD.
  No recomendable para pequeños proyectos, ni para proyectos con una dificultad técnica alta.
  Alta exigencia de tiempos para los desarrolladores que puede ralentizar el proceso en otras etapas.

2.3.2. Metodologías - Ágiles

- **Kanban - Ventajas**
  Tiempos de entrega cortos, tareas simples y cortas, no se necesita una excesiva planificación para implementarlo, aumenta la productividad, ya que cada miembro del desarrollo se encarga de una parte.
  Se crea un flujo de trabajo visual para las distintas tareas.

- **Kanban - Desventajas**
  No es bueno para tiempos de desarrollo muy largos, si no se gestiona bien la división de tareas se puede dar el caso de periodos de trabajo con mucha carga y otros con muy poca o ninguna para ciertas partes del equipo.
  Exige que el desarrollo de las tareas sea repetitivo.

- **Scrum - Ventajas**
  Los usuarios pueden participar en cada una de las etapas del proceso y proponer soluciones, todo se debe evaluar de forma conjunta.
  Flexibilidad y adaptación a los cambios, gestión de los riesgos.
  Divide el proyecto en metas que mejoran el rendimiento del personal al ver cambios significativos en cada sprint.
  Al obtener resultados periódicos se pueden detectar fallos y realizar pruebas reales sobre usuario finales.
  Permite tener una visión global del proyecto.

- **Scrum - Desventajas**
  Requiere un equipo cualificado que conozca la metodología.
  Las tareas pendientes no finalizadas en un *sprint*, ralentizan el proceso de otras tareas más importantes en el siguiente *sprint*. Scrum funciona muy mal cuando los tiempos de entrega no están bien definidos o son cambiantes.

- **Lean - Ventajas**
  Estructura limpia de los datos importantes y críticos del proyecto, entrega rápida de las primeras versiones, motivación del equipo por la toma de decisiones.

- **Lean - Desventajas**
  Alta dependencia del equipo por los compromisos del personal, toma de decisiones tardías, se deben tomar las decisiones críticas en el momento adecuado, es por ello que se debe tener al personal correcto en la tarea adecuada.

- **Programación extrema (XP) - Ventajas**
  Cercanía con el cliente, ausencia de código innecesario, pruebas y CI para desarrollar un código estable, el *pair programming* permite detectar fallos.

- **Programación extrema (XP) - Desventajas**
  Mayor trabajo por posibles cambios debido a posibles cambios del cliente, requiere más tiempo.

> *Scrum* es un marco que permite el trabajo colaborativo entre equipos. Al igual que un equipo de rugby (de donde proviene su nombre) cuando entrena para un gran partido, *scrum* anima a los equipos a aprender a través de las experiencias, a auto organizarse mientras aborda un problema y a reflexionar sobre sus victorias y derrotas para mejorar continuamente.
>
> Atlassian - ¿Qué es scrum? [9]

Para nuestro proyecto se ha decantado por una metodología híbrida entre varias metodologías, esto permite más flexibilidad.

Se ha seguido una metodología híbrida entre *Scrum* y *Lean*, permitiendo hacer entregas de las distintas partes del proyecto y pudiendo redimensionar las tareas en función de los problemas e incompatibilidades que surjan en el camino, con la metodología *Lean* se ha usado dentro del proyecto para reducir el número de ficheros de documentación y organización del proyecto.

Podemos plantear, en función de estas metodologías, las distintas formas de organizar y medir las tareas que deberemos realizar en nuestro proyecto.

De esta forma se han planteado el coste de las tareas usando una comparación de tallas de ropa [10], esto pertenece a *Scrum*, es una recomendación para la gestión de proyectos en equipos, aunque este proyecto se ha realizado de forma individual, esta gestión y división de tareas sirve para saber el coste del proyecto, planificar metas, etc.

Scrum: se quiere aplicar *Scrum* para desarrollar las distintas vistas y componentes en cada *sprint*, se desarrollará un componente vista o componente, el objetivo de desarrollar de esta forma será priorizar los elementos y características más relevantes, de forma que no se produzcan impedimentos o tareas bloqueantes [11].

Se organizarán las tareas más prioritarias y bloqueantes al principio del desarrollo, pues ciertos componentes se acoplan a estas tareas como una dependencia.

Lean: se quiere aplicar *Lean* directamente escribiendo los requisitos, *mocks* y *tests*. Esto pertenece a la técnica “LEAN Amplify Learning” que nos permite reducir la documentación de nuestro proyecto y mantener actualizado el conocimiento común.

2.4. Organización y gestión

Debemos estimar el coste de desarrollo de la aplicación; para ello, se debe analizar cada tarea de forma individual, en especial la prioridad de las distintas tareas. Podemos usar distintas estrategias para estimar el coste de las tareas, entre ellas las estrategias ágiles más famosas son *Dot Voting*, *T-Shirt Size*, *Planning Poker*, *The Bucket System* o *Large / Uncertain / Small* [12], las técnicas mencionadas permiten reducir el error y el esfuerzo en el desarrollo.

Estas técnicas nos permitirán estimar los siguientes costes (dinero, esfuerzo, recursos y tiempo) para poder organizar de la mejor forma el proceso de desarrollo. Podemos ver la estimación de nuestra aplicación en las Tablas 2.1 y 2.2 mediante la técnica *T-Shirt Size*.

Tareas agrupadas	Tareas	Estimación Tallas	Peso
Gestión cuentas	Crear una cuenta	XL	8
	Recuperar contraseña	M	5
	Iniciar y cerrar sesión	L	7
Gestión ficheros	Crear ficheros	L	7
	Modificar ficheros (nombre del fichero)	L	7
	Eliminar ficheros	L	7
Editor de código	Mostrar fichero	M	5
	Editar fichero	M	5
	Guardar fichero	M	5
	Añadir puntos de parada (*breakpoints*) a un fichero	L	7
	Resaltar palabras clave del lenguaje DLX	L	7
	Resaltar una simulación en el código	XL	8
Simulación	Simular una ejecución de código DLX	XL	8
	Simular etapas de la ejecución por pasos	XL	8
	Actualizar los registros	M	5
	Actualizar la memoria	M	5
	Actualizar el segmento de datos (Estadísticas)	M	5
	Actualizar el pipeline	L	7
	Actualizar el cycle-clock	L	7
Configuración	Modificar el autoguardado de la aplicación	M	5
	Modificar los adelantamientos de la simulación	S	3
	Modificar el *delay* de la simulación	S	3
	Modificar el servidor de comunicación	S	3
	Modificar el idioma de la aplicación	S	3
	Modificar el *delay* de las unidades de cálculo de DLX	S	3
	Modificar el número de unidades de cálculo de DLX	S	3
	Modificar el tamaño de la memoria de DLX.	S	3
	Modificar la vista múltiple de la aplicación.	L	7
	Reiniciar la máquina con la nueva configuración	XL	8
		Total:	164

Tabla. 2.1: Tareas - Parte 1

Tareas agrupadas	Tareas	Tallas	Peso
Calculadora	Representar datos en distintos formatos y estándares.	M	5
Componente Memoria	Representar memoria en formato *byte, halfword, word, float, double*	M	5
	Modificar memoria en formato *byte, halfword, word, float, double*	XXL	10
	Simular etapas de la ejecución por pasos	-	
Componente Código	Representar las etapas de ejecución en la memoria y su instrucción	L	7
	Simular etapas de la ejecución por pasos	-	
Componente Registros	Representar registros de propósito general	S	2
	Representar registros de tipo entero	S	2
	Representar registros de simple precisión	M	5
	Representar registros de doble precisión	L	7
	Modificar registros de propósito general	M	5
	Modificar registros de tipo entero	M	5
	Modificar registros de simple precisión	L	7
	Modificar registros de doble precisión	XL	8
	Simular etapas de la ejecución por pasos	-	
Componente Cauce de ejecución	Representar el pipeline en el estado de la simulación	XXL	10
	Simular etapas de la ejecución por pasos	-	
Componente Diagrama de ciclos	Representar un diagrama de ciclos en el estado de la simulación	XXL	10
	Añadir instrucciones	L	7
	Añadir etapas	L	7
	Simular etapas de la ejecución por pasos	-	
Servidor de pruebas	Administrar conexión WebSocket	L	7
	Administrar máquinas	M	5
	Administrar los registros de una máquina	L	7
	Administrar la memoria de una máquina	L	7
	Administrar simulación de una máquina	L	7
	Administrar ficheros	XL	8
	Administrar *mocks* y *stubs*	L	7
		Total:	150

Tabla. 2.2: Tareas - Parte 2

2.5. Estimación del tamaño y esfuerzo del proyecto

Este proyecto se encuentra limitado por dos factores, el personal a cargo de él, un solo desarrollador que es el estudiante y el tiempo asignado para la asignatura en cuestión.

Para calcular el esfuerzo podemos usar la estimación por Puntos de caso de uso (UCP) [13]. Se usan los casos de uso que se han descrito en las tablas 2.1 y 2.1, además del personal a cargo del desarrollo, estos son los puntos principales para el cálculo del esfuerzo, este método analiza los puntos de caso de uso sin ajustar, ajustados y esfuerzo del personal por tiempo y dentro de los puntos de caso de uso sus factores asociados.

Por ello dividimos el esquema para el cálculo de la siguiente manera:

1. Puntos de caso de uso sin ajustar (UUCP)
   1.1 Factor de peso de los actores sin ajustar (UAW)
   1.2 Factor de peso de los casos de uso sin ajustar (UUCW)
2. Puntos de caso de uso ajustados (UCP)
   2.1 Factores de complejidad técnica (TCF)
   2.2 Factores ambientales (EF)
3. Esfuerzo del personal por tiempo dedicado

Para el cálculo de los puntos de uso sin ajustar se usa la siguiente fórmula:

$$UAW = \sum_{i=0}^{n} (cantidadDeUnTipoDeActor_i * Factor_i) \tag{2.1}$$

$$UUCW = \sum_{i=0}^{n} (CantidadDeUnTipoDeCasoUso_i * Factor_i) \tag{2.2}$$

$$UUCP = UAW + UUCW \tag{2.3}$$

$$\tag{2.4}$$

Para el cálculo de los puntos de casos de uso ajustados, usamos estas otras fórmulas:

$$TFactor = \sum_{i=0}^{n}(Valor_i \cdot Peso_i) \tag{2.5}$$

$$TCF = 0{,}6 + (0{,}01 \cdot TFactor) \tag{2.6}$$

$$EFactor = \sum_{i=0}^{n}(Valor_i \cdot Peso_i) \tag{2.7}$$

$$EF = 1{,}4 + (-0{,}03 \cdot EFactor) \tag{2.8}$$

$$UCP = UUCP \cdot TCF \cdot EF \tag{2.9}$$

$$\tag{2.10}$$

Para el cálculo del esfuerzo usamos la siguiente fórmula:

$$E = UCP \cdot CF \tag{2.11}$$

El método cuenta con cuatro etapas, cálculo de los pesos de los actores sin ajustar, cálculo de los casos de uso sin ajustar, cálculo de los puntos de casos de uso ajustados y por último calculo esfuerzo horas-personal.

2.5.1. Puntos de caso de uso sin ajustar (*UUCP*)

Para calcular nuestro factor de peso de los actores sin ajustar (UAW) usamos la fórmula vista anteriormente 2.1, en nuestro caso es simple, pues nuestra aplicación gráfica solo cuenta con un actor, que es el usuario, no debemos tener en cuenta el servidor, pero sí las transacciones y operaciones que se realizan entre ambos. Podemos obtener el factor de peso de los casos de uso sin ajustar (UUCW) de las tablas 2.1 y 2.2, nos da un valor de 314 puntos.

Tipo de actor	Descripción	Factor	Proyecto
Simple	Otro sistema que interactúa con el sistema a desarrollar mediante una interfaz de programación de aplicaciones (API).	1	
Medio	Otro sistema interactuando a través de un protocolo (ej. TCP/IP) o una persona interactuando a través de una interfaz en modo texto.	2	
Complejo	Una persona que interactúa con el sistema mediante una interfaz gráfica (GUI).	3	✓
		Total:	3

Tabla. 2.3: Peso de los actores sin ajustar

Como resultados tenemos $UAW = 3$ y $UUCW = 314$ usamos la fórmula 2.3, dando como resultado final del cálculo los puntos de caso de uso sin ajustar $UUPC = 317$.

2.5.2. Puntos de caso de uso ajustados (*UCP*)

Para calcular los puntos de casos de uso ajustados (UCP) debemos calcular primero los factores de complejidad técnica (TCF) y los factores ambientales (EF). Para ello usamos las fórmulas y tablas que se describen a continuación.

En la complejidad técnica y ambiental, hablamos de tres tramos en la escala que propone el método (**Irrelevante** con un valor del 0 al 2, **Medio** de 3 al 4, **Esencial** 5)

Realizamos los cálculos sobre las siguientes tablas usando las fórmulas $TFactor$ 2.5 y $EFactor$ 2.7 propuestas por el método, asignamos los valores para nuestro proyecto, multiplicamos por su peso y sumamos.

Factor Técnico	Descripción	Peso	Valor	Proyecto $\sum_{i=0}^{n} Peso_i \cdot Valor_i$
T1	Sistema distribuido	2.0	2	4
T2	Objetivos de rendimiento temporales	1.0	4	4
T3	Eficiencia del usuario final	1.0	4	4
T4	Procesamiento interno complejo	1.0	4	4
T5	El código debe ser reutilizable	1.0	4	4
T6	Facilidad de instalación	0.5	4	2
T7	Facilidad de uso	0.5	4	2
T8	Portabilidad	2.0	4	8
T9	Facilidad de cambio	1.0	1?	1
T10	Concurrencia	1.0	2	2
T11	Incluye objetivos especiales de seguridad	1.0	3	3
T12	Provee acceso directo a terceras partes	1.0	0	0
T13	Provee entrenamiento para usuarios	1.0	3	3
			Total:	41

Tabla. 2.4: Factores técnicos

Factor Ambiental	Descripción	Peso	Valor	Filtro	Proyecto $\sum_{i=0}^{n} Peso_i \cdot Valor_i$
E1	Familiaridad con proyectos similares	1.5	2	✓	3
E2	Experiencia en la aplicación	0.5	3	✓	1.5
E3	Experiencia en programación orientada a objetos	1.0	5		5
E4	Capacidad del analista líder	0.5	3	✓	1.5
E5	Motivación	1.0	5		5
E6	Estabilidad de los requerimientos	2.0	5		10
E7	Personal a tiempo parcial	-1.0	3	✓	-3
E8	Dificultad del lenguaje de programación	-1.0	0		0
				Total:	23

Tabla. 2.5: Factores ambientales

De resultado final en los puntos de caso de uso ajustados (UCP) tenemos de valores de los factores técnicos $TFactor = 41$ y los valores en factores ambientales $EFactor = 23$, aplicamos las fórmulas para calcular y obtenemos los fac-

tores técnicos $TCF = 0,6 + (0,01 \cdot TFactor) = 1,01$ y los factores ambientales $EF = 1,4 + (-0,03 \cdot EFactor) = 0,77$.

2.5.3. Esfuerzo horas por personal (*E*)

Factor	Filtro	Proyecto
De E1 a E6	Factor <3	3
De E7 a E8	Factor >3	1

Tabla. 2.6: Factor del esfuerzo horas-persona

Horas-persona (*CF*)	Descripción	Proyecto
20	Si el valor es <= 2	
28	Si el valor es <= 4	✓
36	Si el valor es >= 5	

Tabla. 2.7: Cantidad de horas por persona según el valor EF

Podemos ver los factores ambientales que se han seleccionado en la columna de "filtros" de la tabla 2.5, en total son 4, por lo que nos da $CF = 28$ horas por persona.

Por último, debemos aplicar la fórmula del cálculo del esfuerzo (E) 2.11, dando el siguiente resultado $E = UCP \cdot CF = 246,5 \cdot 28 = 6902 \approx 6900$.

Este dato se debe aplicar a cada actividad en el desarrollo de la aplicación.

Actividad	Porcentaje	Proyecto
Análisis	10 %	690
Diseño	20 %	1380
Programación	40 %	2760
Pruebas	15 %	1035
Sobrecarga	15 %	1035
	Total:	6900

Tabla. 2.8: Esfuerzo total para el desarrollo del proyecto

2.6. Planificación

Como se ha explicado anteriormente, el desarrollo del proyecto se realizará siguiendo una metodología *Scrum* y *Lean*, por lo que se han fijado algunos objetivos, estos objetivos se explican en los siguientes capítulos y apartados, para comprender rápidamente esta sección lo primero que debemos que tener en cuenta en nuestro proyecto es la planificación, ya que es primer paso, a continuación, debemos desarrollar un prototipo, plantilla o marco de partida, debemos empezar la construcción de nuestro proyecto, en los siguientes *sprints* debemos ir añadiendo las características más importantes sin que estás dependan de otras secciones del proyecto.

Como se muestra a continuación, en los siguientes diagramas de *Gantt*, se puede apreciar que se ha tenido en consideración la fecha de matriculación en la asignatura, las fechas de los festivos nacionales más importantes, así también los tiempos de desarrollo de cada una de las tareas, aunque no se ha tenido en cuenta los posibles arreglos que puedan surgir al implementar nuevas características, está planificación es una idea simple de cómo se van a desarrollar las tareas y los objetivos que se han fijado.

2.6.1. Diagrama de Gantt - Septiembre 2021

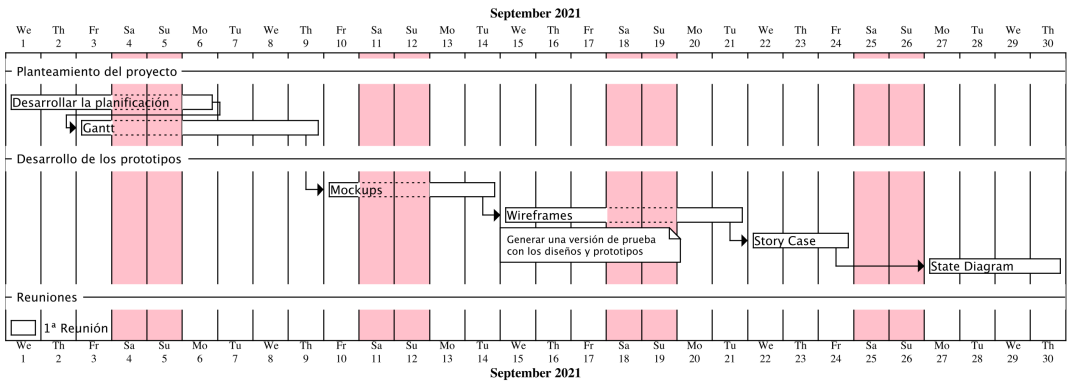


Figura 2.1: Diagrama de Gantt - Septiembre 2021

2.6.2. Diagrama de Gantt - Octubre 2021

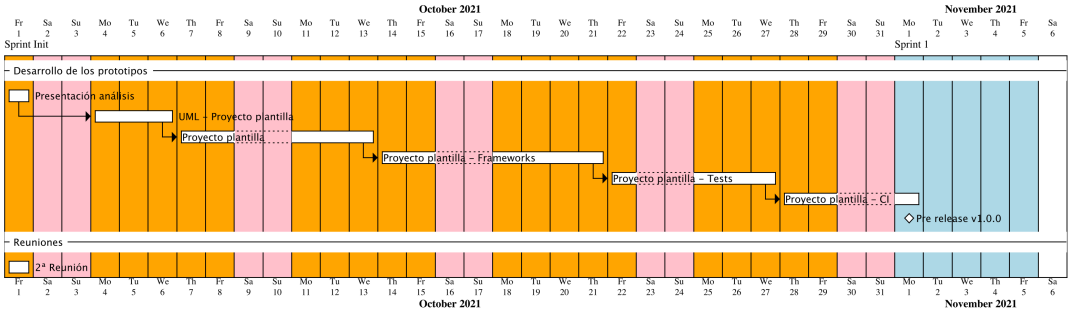


Figura 2.2: Diagrama de Gantt - Octubre 2021

2.6.3. Diagrama de Gantt - Noviembre 2021

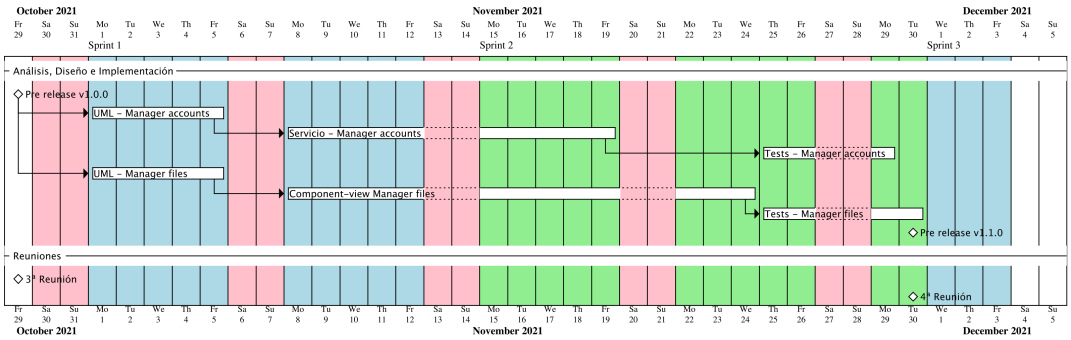


Figura 2.3: Diagrama de Gantt - Noviembre 2021

2.6.4. Diagrama de Gantt - Diciembre 2021

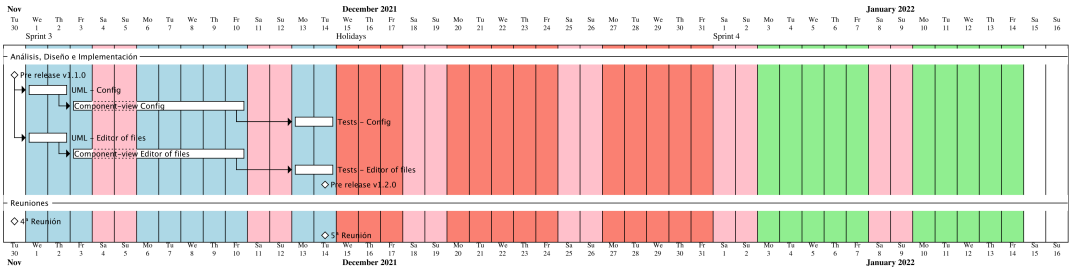


Figura 2.4: Diagrama de Gantt - Diciembre 2021

2.6.5. Diagrama de Gantt - Enero 2022

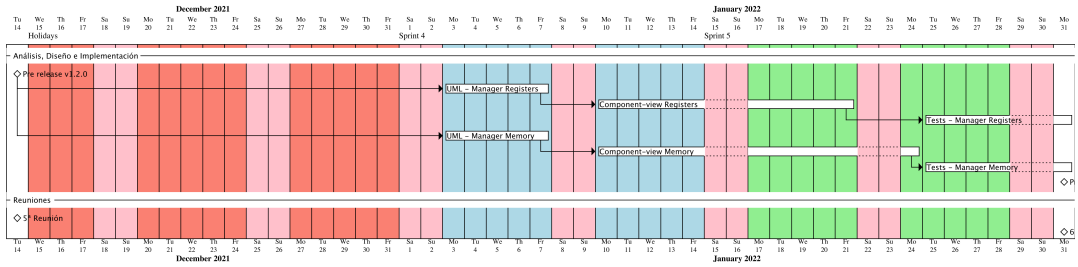


Figura 2.5: Diagrama de Gantt - Enero 2022

2.6.6. Diagrama de Gantt - Febrero 2022

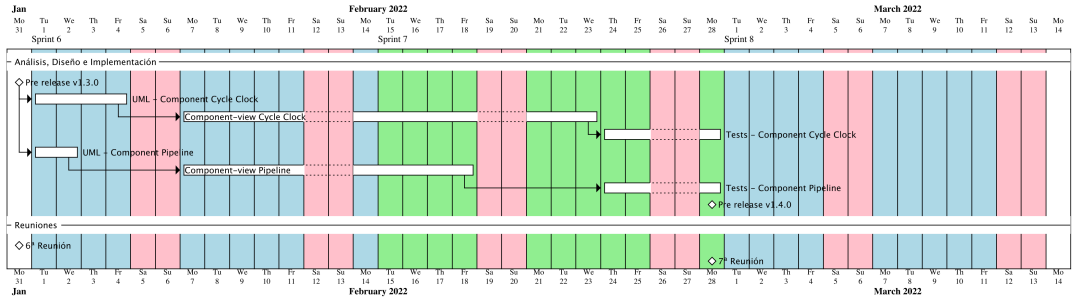


Figura 2.6: Diagrama de Gantt - Febrero 2022

2.6.7. Diagrama de Gantt - Marzo 2022

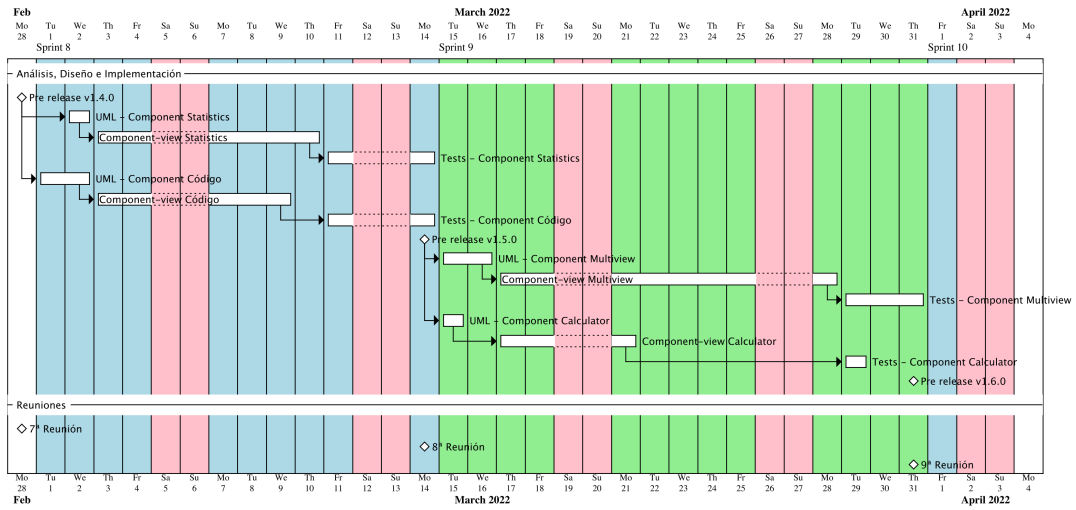


Figura 2.7: Diagrama de Gantt - Marzo 2022

2.6.8. Diagrama de Gantt - Abril 2022

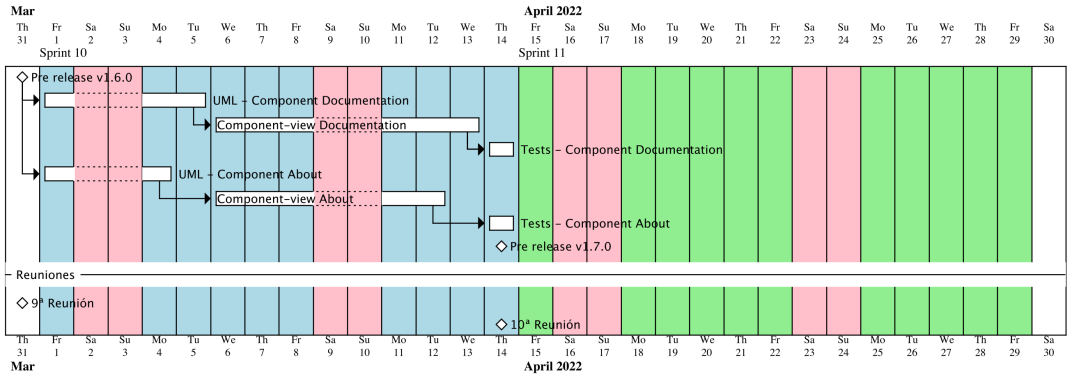


Figura 2.8: Diagrama de Gantt - Abril 2022

2.7. Presupuesto

Para estimar el coste se han estudiado los costes tanto de personal, como de equipo, materiales y servicios.

Se ha tenido en cuenta la utilización de software libre y gratuito para la reducción de costes.

2.7.1. Estimación de recursos

Para la realización de este proyecto contamos con los siguientes equipos, programas y servicios.

- Equipo informático y software
  - MacBook Pro (13-inch, 2018)
    - Procesador Intel Core i5 de 4 núcleos 2,3 GHz
    - RAM 8 GB 2133 MHz LPDDR3
    - Gráficos Intel Iris Plus Graphics 655 1536 MB
    - Disco SSD 256 GB
  - Torre personal (Windows, Ubuntu 20.04)
    - Procesador AMD FX-6300
    - RAM 8 GB 2133 MHz
    - Gráficos Radeon R9

        - Disco SSD 256 GB (Windows)
        - Disco SSD 256 GB (Ubuntu)
        - Disco HDD 2048 GB
- Software
    - GIMP: Edición de imágenes.
    - TexMaker: Editor de documentos.
    - Oveleaf: Editor de documentos.
    - Visual Studio Code: Editor de texto (código).
- Servicios:
    - Alquiler oficina
    - Agua
    - Electricidad
    - Internet
    - Seguro
    - IDE
        - WebStorm
            - Versión 2021
            - Suscripción anual 156.09€ con IVA
- Salario:
    - Horas diarias: 5
    - Días laborales: 6 meses, 120 días.

2.7.2. Estimación de costes

Como ya hemos indicado, el proceso de desarrollo del proyecto será de 6 meses, 120 días, es decir, 600 horas totales.

En general, un ingeniero informático puede trabajar en una amplia variedad de empresas, dentro de ellas, en muchos puestos diferentes. Aunque para las empresas de desarrollo de software se usa el “Convenio colectivo estatal de empresas de consultoría y estudios de mercado y de la opinión pública” [14], por lo que para nuestro proyecto se nos aplica el siguiente apartado, “Área 3. Consultoría, desarrollo y sistemas”, con el nivel 2 salariales del grupo C, los datos se pueden ver actualizados en la siguiente tabla citada [15].

Área 3. Consultoría, desarrollo y sistemas. Diseño, implantación y gestión de nuevas infraestructuras de las tecnologías de la información y comunicaciones (TIC). Agrupa las actividades de programación y codificación de software y pruebas unitarias.

BOE - Convenio colectivo estatal de empresas de consultoría, y estudios de mercados y de la opinión pública

Concepto	**Importe Unitario**	**Duración**	**Total**
Suscripción WebStorm	13€/ mes	6 meses	78€
Seguro multi riesgo	300€/ año	6 meses	150€
Alquiler oficina	400€/ mes	6 meses	2.400€
Salario	19.866,84€/ año	6 meses	9.933€
		Total	12.561€

Tabla. 2.9: Presupuesto

Capítulo 3

Análisis de requisitos

Para el diseño de nuestro software debemos entender los requisitos y aplicar los patrones de diseño, asegurando así la calidad del software y de la solución final.

El sistema permitirá la simulación a partir del uso de una cuenta registrada. Para realizar una simulación el sistema deberá ser capaz de interpretar un fichero de código y poder ejecutarlo, es por ello que el sistema deberá gestionar ficheros. El sistema también será capaz de realizar ejecuciones, instrucción a instrucción, así como añadir o quitar puntos de parada (*breakpoints*), alterar el estado de la máquina, etc.

3.1. Captura de requisitos

Los requisitos se han descrito a partir de la Figura 3.1. Estos diagramas muestran la interacción entre el usuario y el sistema desde la perspectiva de un agente externo.

Los casos de uso definen el comportamiento del sistema, es decir, qué puede realizar.

En la fase inicial del desarrollo hay que tener en cuenta los casos de uso de la Figura 3.1 y a partir de cada uno de estos casos de uso guiar el diseño.

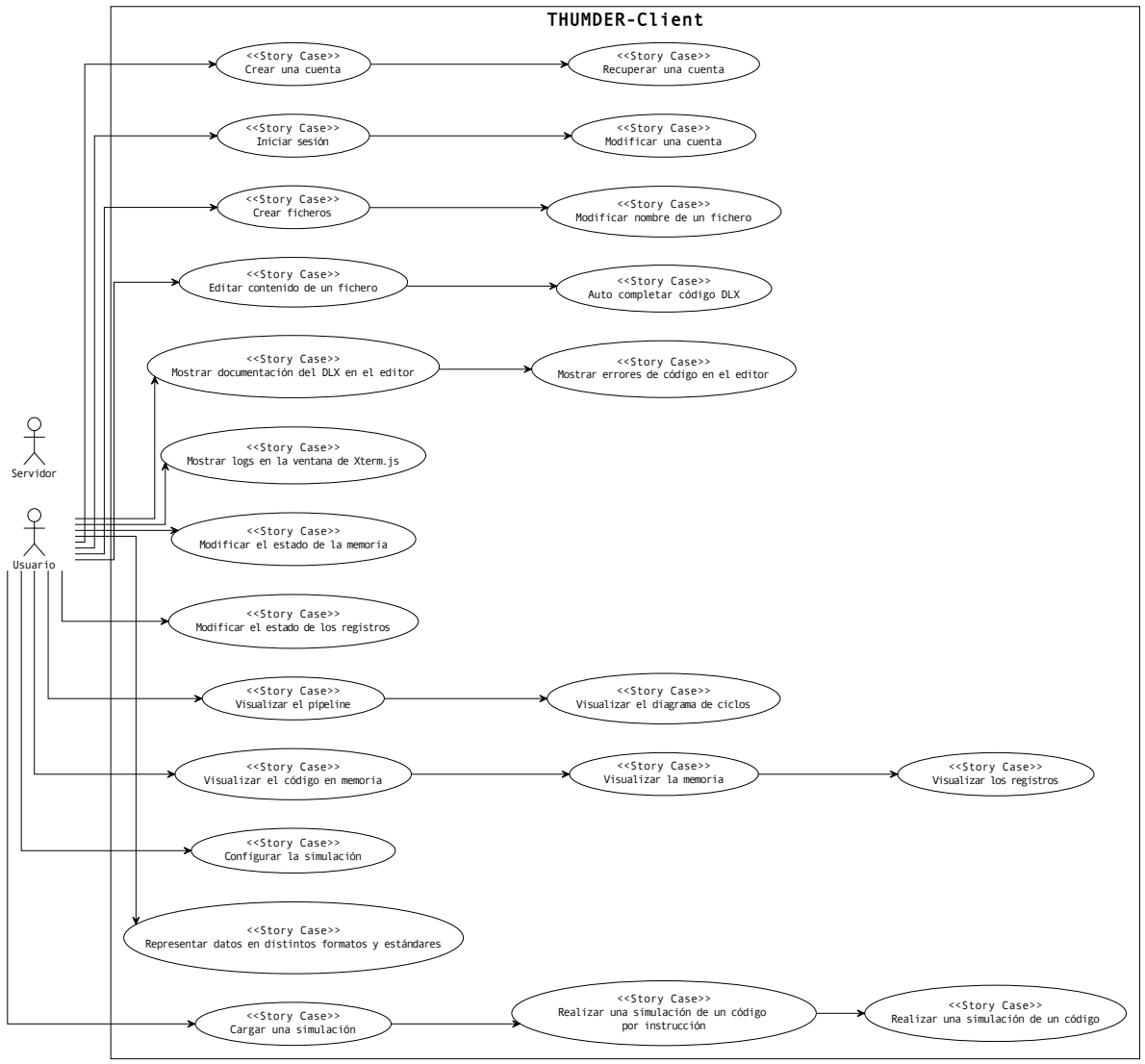


Figura 3.1: Diagrama de casos de uso del sistema

3.2. Funcionales

Los requisitos funcionales son en ingeniería de sistemas y de software lo que especifica que es lo qué se puede hacer con el proyecto, estos requisitos se especifican entre el equipo de desarrollo y el cliente, deben ser claros y definir claramente qué hace el sistema.

Estos requisitos funcionales especifican el comportamiento de la aplicación.

Para ello se han definido en la tabla 3.1 los siguientes requisitos funcionales:

Código	Casos de uso
CU-01	Crear una cuenta
CU-02	Iniciar una sesión
CU-03	Recuperar una cuenta
CU-04	Modificar una cuenta
CU-05	Mostrar información de la aplicación
CU-06	Crear un fichero
CU-07	Eliminar un fichero
CU-08	Modificar nombre de un fichero
CU-09	Editar contenido de un fichero
CU-10	Auto completar código DLX
CU-11	Mostrar documentación del DLX en el editor
CU-12	Mostrar errores de código en el editor
CU-13	Mostrar información en la consola de información
CU-14	Modificar el estado de la memoria
CU-15	Modificar el estado de los registros
CU-16	Visualizar el cauce de ejecución (*Pipeline*)
CU-17	Visualizar el diagrama de ciclos
CU-18	Visualizar el código en memoria
CU-19	Visualizar la memoria
CU-20	Visualizar los registros
CU-21	Configurar la simulación
CU-22	Representar datos en distintos formatos y estándares
CU-23	Cargar una simulación
CU-24	Realizar una simulación de un código por instrucción
CU-25	Realizar una simulación de un código

Tabla. 3.1: Requerimientos funcionales

3.2.1. CU-01 Crear una cuenta

En el primer caso de uso se define como un usuario puede crear una cuenta para interaccionar con el sistema. El sistema deberá notificar si la acción es permitida o no.

CU-01	**Crear una cuenta**
Actores	Usuario, Sistema
Pre condiciones	- No tener una sesión iniciada
Pos condiciones	- Se elimina el acceso a las secciones restringidas por el inicio de sesión
Escenario principal	1. El usuario selecciona crear una cuenta 2. El usuario introduce los datos de entrada 3. El sistema acepta los datos de sesión 4. El sistema notifica que se ha creado la cuenta 5. Fin del escenario con éxito
Escenarios alternativos	3.a.1. El sistema rechaza los datos de sesión 3.a.2. El sistema informa de que los datos no son válidos 3.a.3. Volver al paso 2 del escenario principal
Excepciones	3.x.1. El sistema no acepta peticiones 3.x.2. El sistema informa de que los datos no sé pueden validar por un fallo 3.x.3. Fin del escenario sin éxito

Tabla. 3.2: CU-01 Crear una cuenta

3.2.2. CU-02 Iniciar una sesión

En el segundo caso de uso se describe cómo un usuario accede a su cuenta y como sé válida y notifica la acción. El sistema debe notificar si los datos son válidos y el estado de la operación, por si se ha podido iniciar sesión o se ha producido un error.

CU-02	**Iniciar una sesión**
Actores	Usuario, Sistema
Pre condiciones	- No tener una sesión iniciada
Pos condiciones	- Sesión iniciada
Escenario principal	1. El usuario selecciona la opción de iniciar sesión 2. El usuario introduce los datos de inicio de su perfil 3. El sistema notifica si los datos son válidos 4. Fin del escenario con éxito
Escenarios alternativos	3.a.1. El sistema no acepta los datos de inicio de sesión 3.a.2. Volver al paso 2
Excepciones	3.x.1. El sistema no puede comunicarse con el servidor 3.x.2. Fin del escenario sin éxito

Tabla. 3.3: CU-02 Iniciar una sesión

3.2.3. CU-03 Recuperar una cuenta

En el tercer caso de uso se define la acción del usuario para recuperar el acceso a su cuenta en caso de pérdida de la contraseña y cómo restablecer esta. El sistema debe notificar si los datos son válidos y del estado de la operación.

CU-03	**Recuperar una cuenta**
Actores	Usuario, Sistema
Pre condiciones	- No tener una sesión iniciada
Pos condiciones	- Ninguna
Escenario principal	1. El usuario selecciona la opción de recuperar contraseña 2. El sistema envía un correo electrónico con el enlace 3. El usuario introduce los nuevos datos en el sistema 4. El sistema notifica si los datos son válidos en el sistema 5. Fin del escenario con éxito
Escenarios alternativos	3.a.1. El sistema no acepta los datos del sistema 3.a.2. Volver al paso 2
Excepciones	3.x.1. El sistema no puede comunicarse con el servidor 3.x.2. Fin del escenario sin éxito

Tabla. 3.4: CU-03 Recuperar una cuenta

3.2.4. CU-04 Modificar una cuenta

En el cuarto caso de uso se define la acción de modificar una cuenta para personalizar la cuenta. El sistema debe notificar si los datos son válidos y el estado de la operación.

CU-04	**Modificar una cuenta**
Actores	Usuario, Sistema
Pre condiciones	- Tener una sesión iniciada
Pos condiciones	- Los cambios se guardan
Escenario principal	1. El usuario accede a la sección de perfil 2. El usuario selecciona la opción de modificar contraseña 3. El usuario introduce la nueva contraseña 4. El sistema notifica si los datos son válidos 5. Fin del escenario con éxito
Escenarios alternativos	4.a.1. El sistema no acepta los datos del usuario 4.a.2. Volver al paso 3
Excepciones	4.x.1 El sistema no puede comunicarse con el servidor 4.x.2. Fin del escenario sin éxito

Tabla. 3.5: CU-04 Modificar una cuenta

3.2.5. CU-05 Mostrar información de la aplicación

En el quinto caso de uso se define el proceso para acceder a la información de la aplicación.

CU-05	**Mostrar información de la aplicación**
Actores	Usuario, Sistema
Pre condiciones	- Tener una sesión iniciada
Pos condiciones	- Ninguna
Escenario principal	1. El usuario accede a la sección del Sobre nosotros 2. El usuario selecciona la opción a mostrar 3. El sistema despliega la información 4. El sistema oculta el resto de información 5. Fin del escenario con éxito
Escenarios alternativos	- Ninguno
Excepciones	- Ninguna

Tabla. 3.6: CU-05 Mostrar información de la aplicación

3.2.6. CU-06 Crear un fichero

En el sexto caso de uso se define el proceso para crear un nuevo fichero y los escenarios alternativos. El sistema debe notificar el estado de la operación.

CU-06	**Crear un fichero**
Actores	Usuario, Sistema
Pre condiciones	- Tener una sesión iniciada
Pos condiciones	- Ninguna
Escenario principal	1. El usuario accede a la sección del gestor de ficheros 2. El usuario selecciona la opción de crear un nuevo fichero 3. El sistema notifica si se ha creado el fichero 4. Fin del escenario con éxito
Escenarios alternativos	3.a.1. El sistema no permite crear un nuevo fichero 3.a.2. Volver al paso 2
Excepciones	3.x.1 El sistema no puede comunicarse con el servidor 3.x.2. Fin del escenario sin éxito

Tabla. 3.7: CU-06 Crear un fichero

3.2.7. CU-07 Eliminar ficheros

En el séptimo caso de uso se define el proceso para eliminar un fichero y los escenarios alternativos. El sistema debe notificar el estado de la operación, si se ha eliminado el fichero o se ha producido un error.

CU-06	**Eliminar un fichero**
Actores	Usuario, Sistema
Pre condiciones	- Tener una sesión iniciada
Pos condiciones	- Ninguna
Escenario principal	1. El usuario accede a la sección del gestor de ficheros 2. El usuario selecciona un fichero 3. El usuario selecciona la opción de eliminar el fichero 3. El sistema notifica si se ha eliminado el fichero 4. Fin del escenario con éxito
Escenarios alternativos	3.a.1. El sistema no permite eliminar el fichero 3.a.2. Volver al paso 2
Excepciones	3.x.1 El sistema no puede comunicarse con el servidor 3.x.2. Fin del escenario sin éxito

Tabla. 3.8: CU-07 Eliminar un fichero

3.2.8. CU-08 Modificar nombre de un fichero

En el octavo caso de uso se define el proceso para modificar el nombre de un fichero y sus escenarios alternativos. El sistema debe notificar el estado de la operación, si se ha podido modificar el nombre del fichero o se ha producido un error.

CU-08	**Modificar nombre de un fichero**
Actores	Usuario, Sistema
Pre condiciones	- Tener una sesión iniciada
Pos condiciones	- Ninguna
Escenario principal	1. El usuario accede a la sección del gestor de ficheros 2. El usuario selecciona un fichero 3. El usuario selecciona la opción de modificar el nombre de un fichero 4. El usuario introduce el nuevo nombre del fichero 5. El sistema notifica si se ha procesado la operación con éxito 6. Fin del escenario con éxito
Escenarios alternativos	5.a.1. El sistema notifica de que ya existe el fichero con ese nombre 5.a.2. Volver al paso 1
	5.b.1. El sistema no permite la operación 5.b.2. Volver al paso 2
Excepciones	5.x.1 El sistema no puede comunicarse con el servidor 5.x.2. Fin del escenario sin éxito

Tabla. 3.9: CU-08 Modificar nombre de un fichero

3.2.9. CU-09 Editar contenido de un fichero

En el noveno caso de uso se define el proceso para abrir y editar un fichero y sus escenarios alternativos. El sistema debe notificar el estado de la operación, el sistema notificará con un mensaje descriptivo en caso de error.

CU-09	**Editar contenido de un fichero**
Actores	Usuario, Sistema
Pre condiciones	- Tener una sesión iniciada
Pos condiciones	- Ninguna
Escenario principal	1. El usuario accede a la sección del gestor de ficheros 2. El usuario selecciona un fichero 3. El sistema solicita al servidor el fichero 4. El sistema abre el fichero en el editor 5. El sistema comprueba si se ha procesado la operación con éxito 6. Fin del escenario con éxito
Escenarios alternativos	- Ninguno
Excepciones	5.x.1. El sistema no puede comunicarse con el servidor 5.x.2. Fin del escenario sin éxito

Tabla. 3.10: CU-09 Editar contenido de un fichero

3.2.10. CU-10 Auto completar código DLX

En el décimo caso de uso se define el proceso que se sigue cuando se está escribiendo el código en el editor para auto completar código en función de los parámetros preestablecidos.

CU-10	**Auto completar código DLX**
Actores	Usuario, Sistema
Pre condiciones	- Tener una sesión iniciada - Tener un fichero cargado en el editor
Pos condiciones	- Autocompleta la instrucción
Escenario principal	1. El usuario accede a la sección del editor
	2. El usuario escribe el principio de una palabra clave 3. El sistema muestra en un desplegable las opciones de la palabra clave 4. El usuario se desplaza entre las opciones del desplegable 5. El usuario selecciona una instrucción del desplegable 6. Fin del escenario con éxito
Escenarios alternativos	- Ninguno
Excepciones	- Ninguno

Tabla. 3.11: CU-10 Auto completar código DLX

3.2.11. CU-11 Mostrar documentación del DLX en el editor

En el undécimo caso de uso se define el proceso que se sigue cuando se está escribiendo el código para que el editor nos muestre la documentación de la instrucción escrita.

CU-11	**Mostrar documentación del DLX en el editor**
Actores	Usuario, Sistema
Pre condiciones	- Tener un fichero cargado en el editor - Tener una palabra clave escrita en el editor
Pos condiciones	- Se muestra la documentación en el editor
Escenario principal	1. El usuario accede a la sección del editor 2. El usuario coloca encima de la palabra clave el ratón 3. El sistema muestra en una ventana emergente la documentación a la que corresponde la palabra clave 4. Fin del escenario con éxito
Escenarios alternativos	- Ninguno
Excepciones	- Ninguno

Tabla. 3.12: CU-11 Mostrar documentación del DLX en el editor

3.2.12. CU-12 Mostrar errores de código en el editor

En el duodécimo caso de uso se define el proceso que se sigue cuando se está escribiendo el código en el editor y el sistema detecta un posible fallo.

CU-12	**Mostrar errores de código en el editor**
Actores	Usuario, Sistema
Pre condiciones	- Tener una sesión iniciada - Tener un fichero cargado en el editor
Pos condiciones	- Se muestran los errores en el código
Escenario principal	1. El usuario accede a la sección del editor 2. El usuario escribe una palabra clave en el editor 3. El usuario escribe erróneamente los registros en la instrucción 4. El sistema detecta un patrón que no es válido 5. El sistema informa del error en una ventana emergente con la información 6. Fin del escenario con éxito
Escenarios alternativos	- Ninguno
Excepciones	5.x.1. El sistema detecta el fallo, pero no notifica la característica del error 5.x.2. Fin del escenario sin éxito

Tabla. 3.13: CU-12 Mostrar errores de código en el editor

3.2.13. CU-13 Mostrar información en la consola de información

En el décimo tercer caso de uso se define el proceso que se sigue cuando el usuario quiere consultar el estado de las operaciones realizadas y registradas.

CU-13	**Mostrar información en la consola de información**
Actores	Usuario, Sistema
Pre condiciones	- Tener una sesión iniciada - Tener un fichero cargado en el editor - Cargar el código - Realizar una simulación
Pos condiciones	- Se muestran información registrada durante la simulación
Escenario principal	1. El usuario accede a la sección de la consola de información (*logger*) 2. El sistema muestra los informes 3. Fin del escenario con éxito
Escenarios alternativos	- Ninguno
Excepciones	- Ninguna

Tabla. 3.14: CU-13 Mostrar información en la consola de información

3.2.14. CU-14 Modificar el estado de la memoria

En el décimo cuarto caso de uso se define el proceso que se sigue cuando el usuario quiere modificar el estado de la memoria y su alteración en el servidor, así como sus escenarios alternativos y excepciones.

CU-14	**Modificar el estado de la memoria**
Actores	Usuario, Sistema
Pre condiciones	- Tener una sesión iniciada
Pos condiciones	- Ninguna
Escenario principal	1. El usuario accede a la sección de la memoria 2. El usuario habilita el editor de la memoria 3. El usuario selecciona el tipo de dato que va a editar 4. El usuario selecciona la dirección de memoria a editar 5. El sistema verifica la dirección de memoria 6. El usuario introduce el dato 7. El sistema verifica que el dato es válido 8. Fin del escenario con éxito
Escenarios alternativos	5.a.1. La dirección de memoria no es válida 5.a.2. El sistema bloquea el editor 5.a.3. Volver al paso 4
	7.b.1. El dato no es válido 7.b.2. El sistema pone a 0 los bits de la dirección de memoria 7.b.3. Volver al paso 6
Excepciones	6.x.1. El sistema no puede comunicarse con el servidor 6.x.2. Fin del escenario sin éxito

Tabla. 3.15: CU-14 Modificar el estado de la memoria

3.2.15. CU-15 Modificar el estado de los registros

En el décimo quinto caso de uso se define el proceso que se sigue cuando el usuario quiere modificar el estado de los registros y su alteración en el servidor, así como sus escenarios alternativos y excepciones.

CU-15	**Modificar el estado de los registros**
Actores	Usuario, Sistema
Pre condiciones	- Tener una sesión iniciada
Pos condiciones	- Ninguna
Escenario principal	1. El usuario accede a la sección de los registros 2. El usuario habilita el editor de los registros 3. El usuario selecciona el tipo de registro que va a editar 4. El usuario selecciona el registro que va a editar 5. El usuario introduce el dato 6. El sistema verifica que el dato es válido 7. Fin del escenario con éxito
Escenarios alternativos	6.a.1. El tipo de dato no es válido en el registro 6.a.2. Volver al paso 5
Excepciones	- Ninguna

Tabla. 3.16: CU-15 Modificar el estado de los registros

3.2.16. CU-16 Visualizar el cauce de ejecución (*Pipeline*)

En el décimo sexto caso de uso se define el proceso que se sigue cuando el usuario quiere visualizar el estado del cauce de ejecución en la simulación.

CU-16	**Visualizar el cauce de ejecución**
Actores	Usuario, Sistema
Pre condiciones	- Tener una sesión iniciada - Tener un fichero cargado en el editor - Cargar el código - Realizar una simulación
Pos condiciones	- Ninguna
Escenario principal	1. El usuario accede a la sección del cauce de ejecución 2. El sistema muestra el estado del cauce de ejecución 3. Fin del escenario con éxito
Escenarios alternativos	- Ninguno
Excepciones	- Ninguna

Tabla. 3.17: CU-16 Visualizar el cauce de ejecución

3.2.17. CU-17 Visualizar el diagrama de ciclos

En el décimo séptimo caso de uso se define el proceso que se sigue cuando el usuario quiere visualizar el estado del diagrama de ciclos en la simulación.

CU-17	**Visualizar el diagrama de ciclos**
Actores	Usuario, Sistema
Pre condiciones	- Tener una sesión iniciada - Tener un fichero cargado en el editor - Cargar el código - Realizar una simulación
Pos condiciones	- Ninguna
Escenario principal	1. El usuario accede a la sección del diagrama de ciclos 2. El sistema muestra el estado del diagrama de ciclos 3. Fin del escenario con éxito
Escenarios alternativos	- Ninguno
Excepciones	- Ninguna

Tabla. 3.18: CU-17 Visualizar el diagrama de ciclos

3.2.18. CU-18 Visualizar el código en memoria

En el décimo octavo caso de uso se define el proceso que se sigue cuando el usuario quiere visualizar la correlación entre la memoria en la maquina y su representación como instrucción y los registros que opera.

CU-18	**Visualizar el código en memoria**
Actores	Usuario, Sistema
Pre condiciones	- Tener una sesión iniciada - Tener un fichero cargado en el editor - Cargar el código
Pos condiciones	- Ninguna
Escenario principal	1. El usuario accede a la sección del visualizador del código 2. El sistema muestra el código en la memoria 3. Fin del escenario con éxito
Escenarios alternativos	- Ninguno
Excepciones	- Ninguna

Tabla. 3.19: CU-18 Visualizar el código en memoria

3.2.19. CU-19 Visualizar la memoria

En el décimo noveno caso de uso se define el proceso que se sigue cuando el usuario quiere visualizar la memoria en la máquina y sus distintas representaciones.

CU-19	**Visualizar la memoria**
Actores	Usuario, Sistema
Pre condiciones	- Tener una sesión iniciada
Pos condiciones	- Ninguna
Escenario principal	1. El usuario accede a la sección del visualizador de memoria 2. El usuario selecciona el tipo de representación 3. El sistema muestra el código máquina en la memoria 4. Fin del escenario con éxito
Escenarios alternativos	- Ninguno
Excepciones	- Ninguna

Tabla. 3.20: CU-19 Visualizar la memoria

3.2.20. CU-20 Visualizar los registros

En el vigésimo caso de uso se define el proceso que se sigue cuando el usuario quiere visualizar el estado de los registros.

CU-20	**Visualizar los registros**
Actores	Usuario, Sistema
Pre condiciones	- Tener una sesión iniciada
Pos condiciones	- Ninguna
Escenario principal	1. El usuario accede a la sección del visualizador de registros 2. El usuario selecciona el tipo de registro 3. El usuario selecciona el tipo de representación 4. El sistema muestra el valor de los registros 5. Fin del escenario con éxito
Escenarios alternativos	- Ninguno
Excepciones	- Ninguna

Tabla. 3.21: CU-20 Visualizar los registros

3.2.21. CU-21 Configurar la simulación

En el vigésimo primer caso de uso se define el proceso que se sigue cuando el usuario debe cambiar la configuración de la máquina y cómo esto se registra, así como las excepciones que puede causar. El sistema debe notificar y registrar los cambios en el servidor que realiza la simulación.

CU-21	**Configurar la simulación**
Actores	Usuario, Sistema
Pre condiciones	- Tener una sesión iniciada
Pos condiciones	- Ninguna
Escenario principal	1. El usuario accede a la sección de la configuración 2. El usuario selecciona el número de núcleos de cálculo de sumas de flotantes 3. El usuario selecciona el número de núcleos de cálculo de multiplicación de flotantes 4. El usuario selecciona el número de núcleos de cálculo de división de flotantes 5. El usuario selecciona el tiempo de espera (*delay*) en los núcleos de cálculo de sumas de flotantes 6. El usuario selecciona el tiempo de espera (*delay*) en los núcleos de cálculo de multiplicación de flotantes 7. El usuario selecciona el tiempo de espera (*delay*) en los núcleos de cálculo de división de flotantes 8. El usuario selecciona el tamaño de la memoria en bytes 9. El sistema valida los datos 10. El sistema muestra y registra si los datos son válidos 11. Fin del escenario con éxito
Escenarios alternativos	- Ninguno
Excepciones	- Ninguna

Tabla. 3.22: CU-21 Configurar la simulación

3.2.22. CU-22 Representar datos en distintos formatos y estándares

En el vigésimo segundo caso de uso se define el proceso que se sigue cuando el usuario quiere representar los datos en sus distintas representaciones.

CU-22	**Representar datos en distintos formatos y estándares**
Actores	Usuario, Sistema
Pre condiciones	- Tener una sesión iniciada - Tener un fichero cargado en el editor - Cargar el código
Pos condiciones	- Ninguna
Escenario principal	1. El usuario accede a la sección de la calculadora 2. El usuario selecciona el tipo de dato a representar 3. El usuario introduce el dato a transformar en su estándar 4. El sistema transforma el dato en sus distintas representaciones 5. Fin del escenario con éxito
Escenarios alternativos	- Ninguno
Excepciones	- Ninguna

Tabla. 3.23: CU-22 Representar datos en distintos formatos y estándares

3.2.23. CU-23 Cargar una simulación

En el vigésimo tercer caso de uso se define el proceso que se sigue cuando el usuario quiere cargar un código en el servidor y que este le responda con su simulación.

CU-23	**Cargar una simulación**
Actores	Usuario, Sistema
Pre condiciones	- Tener una sesión iniciada
Pos condiciones	- Cargar el código
Escenario principal	1. El usuario accede a la sección del editor 2. El usuario carga el código del editor en el servidor 3. El sistema notifica si la carga es válida 4. Fin del escenario con éxito
Escenarios alternativos	3.a.1. El sistema no válida el código en el servidor 3.a.2. Ir al escenario Editar contenido de un fichero
Excepciones	2.x.1. El sistema no válida el servidor 2.x.2. El sistema bloquea el simulador

Tabla. 3.24: CU-23 Cargar una simulación

3.2.24. CU-24 Realizar una simulación de un código por instrucción

En el vigésimo cuarto caso de uso se define el proceso que se sigue cuando el usuario desea realizar una simulación instrucción por instrucción cargando el código y sus escenarios alternativos y excepciones.

CU-24	**Realizar una simulación de un código por instrucción**
Actores	Usuario, Sistema
Pre condiciones	- Tener una sesión iniciada
Pos condiciones	- Ninguna
Escenario principal	1. El usuario carga el código 2. El sistema válida el código de la simulación 3. El sistema habilita los instrumentos de la simulación 4. El usuario ejecuta la simulación de una sola instrucción 5. El sistema realiza la simulación de una sola instrucción 6. Fin del escenario con éxito
Escenarios alternativos	2.a.1. El sistema no válida el código de la simulación 2.a.2. Ir al escenario Editar contenido de un fichero
Excepciones	2.x.1. El sistema no puede comunicarse con el servidor 2.x.2. Fin del escenario sin éxito

Tabla. 3.25: CU-24 Realizar una simulación de un código por instrucción

3.2.25. CU-25 Realizar una simulación de un código

En el vigésimo quinto caso de uso se define el proceso que se sigue cuando el usuario desea realizar una simulación, instrucción por instrucción, pero continuada, así como sus distintos escenarios alternativos y excepciones.

CU-25	**Realizar una simulación de un código**
Actores	Usuario, Sistema
Pre condiciones	- Tener una sesión iniciada
Pos condiciones	- Ninguna
Escenario principal	1. El usuario carga el código 2. El sistema válida el código de la simulación 3. El sistema habilita los instrumentos de la simulación 4. El usuario ejecuta el *runner* de la simulación 5. El sistema realiza el *runner* de la simulación 6. Fin del escenario con éxito
Escenarios alternativos	2.a.1. El sistema no válido el código de la simulación 2.a.2. Ir al escenario Editar contenido de un fichero
Excepciones	2.x.1. El sistema no puede comunicarse con el servidor 2.x.2. Fin del escenario sin éxito

Tabla. 3.26: CU-25 Realizar una simulación de un código

3.3. No funcionales

Un requisito no funcional se define en ingeniería de sistemas y de software de la siguiente forma, especifica criterios que pueden usarse para juzgar un sistema en lugar de su comportamiento, modela el diseño y la implementación.

No describen una acción, sino una característica del software, son los denominados atributos de calidad de un sistema, suelen ser restricciones o condiciones que impone el cliente al programa que necesita.

Estos suelen ser requisitos de rendimiento, estabilidad, accesibilidad, capacidad, privacidad, robustez, estabilidad, garantía, etc. [16]

Algunos de estos requisitos pueden clasificarse en las siguientes categorías:

- Requerimientos del producto.
- Requerimientos de la organización.
- Requerimientos externos.

Se han definido los siguientes requisitos no funcionales:

1. La aplicación debe ser multiplataforma.
2. La aplicación debe seguir un patrón de diseño MVVM híbrido o también llamado MVW.
3. La aplicación debe tener una interfaz adaptable (responsive).
4. La aplicación debe ser mantenible.
5. La aplicación debe seguir el protocolo de comunicación con el servidor.
6. La aplicación debe ser tolerante a fallos y las operaciones deben ser transaccionales, deben finalizar de forma segura sin dañar la estabilidad del sistema.
7. La aplicación debe tener pruebas unitarias y pruebas de interfaz.
8. La aplicación debe desplegarse de forma automática descargando y gestionando las dependencias mediante un sistema de compilación automática.
9. La aplicación debe tener documentación.
10. El sistema de ficheros y editor de la aplicación debe estar protegido y notificar si no se puede editar.

3.4. Especificación de requisitos - Modelo de dominio

Como se ha especificado en los casos de uso, podemos modelar nuestro modelo de dominio a partir de los casos de uso.

En nuestro caso, el modelo de dominio es grande en atributos, por lo que solo se representan las entidades conceptuales más representativas del proyecto, además en un futuro estos atributos pueden cambiar.

Este modelo no representa el objeto software, sino una representación visual de las relaciones más importantes y su interacción con el usuario [1].

[1] En este modelo de dominio solo nos referimos para la parte de ficheros y documentos que se almacenan en la base de datos de nuestro servidor de Firebase

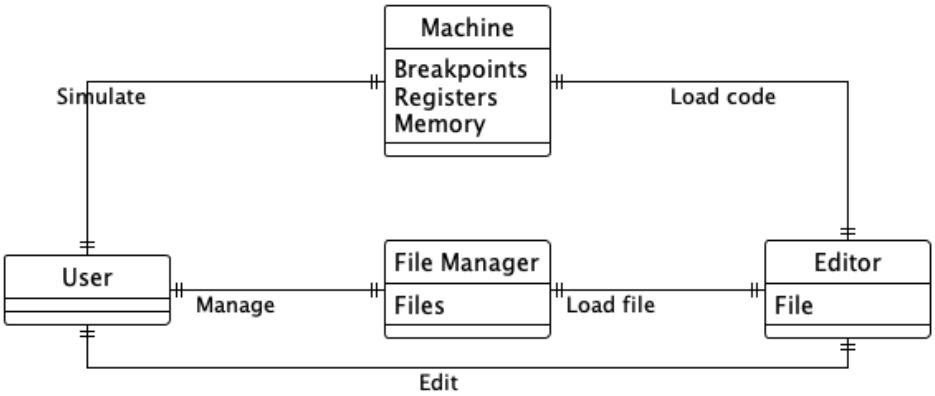


Figura 3.2: Modelo de dominio

Capítulo 4

Diseño

El diseño del software es una de las etapas cruciales en nuestro desarrollo, pues es en este momento donde se debe analizar y guiar el proyecto para que la solución que se proponga sea estable y sólida, de forma que el proyecto pueda ser desarrollado en diferentes tecnologías y pueda crecer de forma segura siguiendo las guías para desarrollar código de cálida.

4.1. Diagrama de sistema

Para comprender la forma en la que se ha de organizar el proyecto, debemos ir desglosando las partes más grandes e importantes, ir poco a poco a las secciones más pequeñas.

Lo primero es entender el diagrama del sistema, es decir, entender la organización principal de la aplicación.

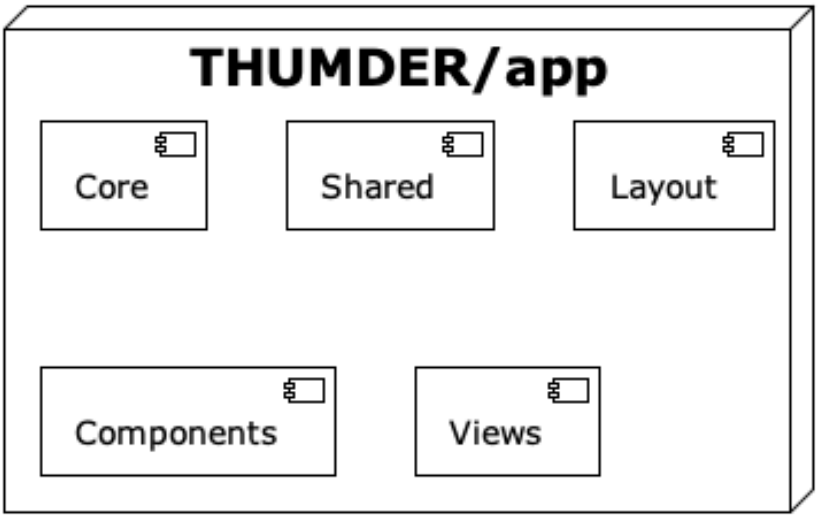


Figura 4.1: Diagrama del sistema

4.2. Diagrama de paquetes

Diferenciamos las tres grandes entidades que existen en el proyecto (usuario, el sistema y el servidor), deberemos analizar y moldear el problema en función de estas entidades, nuestro proyecto es una interfaz que representa los distintos estados que va pasando una simulación de una arquitectura DLX, por lo que debemos tener en cuenta la comunicación con el servidor como una de las secciones más críticas.

Una vez que sabemos cómo se organiza la aplicación en su jerarquía superior, lo mejor es entender la división de los paquetes, desde dónde se organiza la información hasta dónde se envía para representarla.

Los paquetes son los siguientes:

1. Núcleo - *Core* 4.2.1
2. Distribución - *Layout* 4.2.2
3. Útiles - *Shared* 4.2.3
4. Componentes - *Components* 4.2.4
5. Vistas - *Views* 4.2.5

4.2.1. Diagrama del paquete del núcleo (`core`)

El paquete del núcleo o (*core*) es el encargado de gestionar y encapsular toda la lógica de la simulación, así como la comunicación con el servidor.

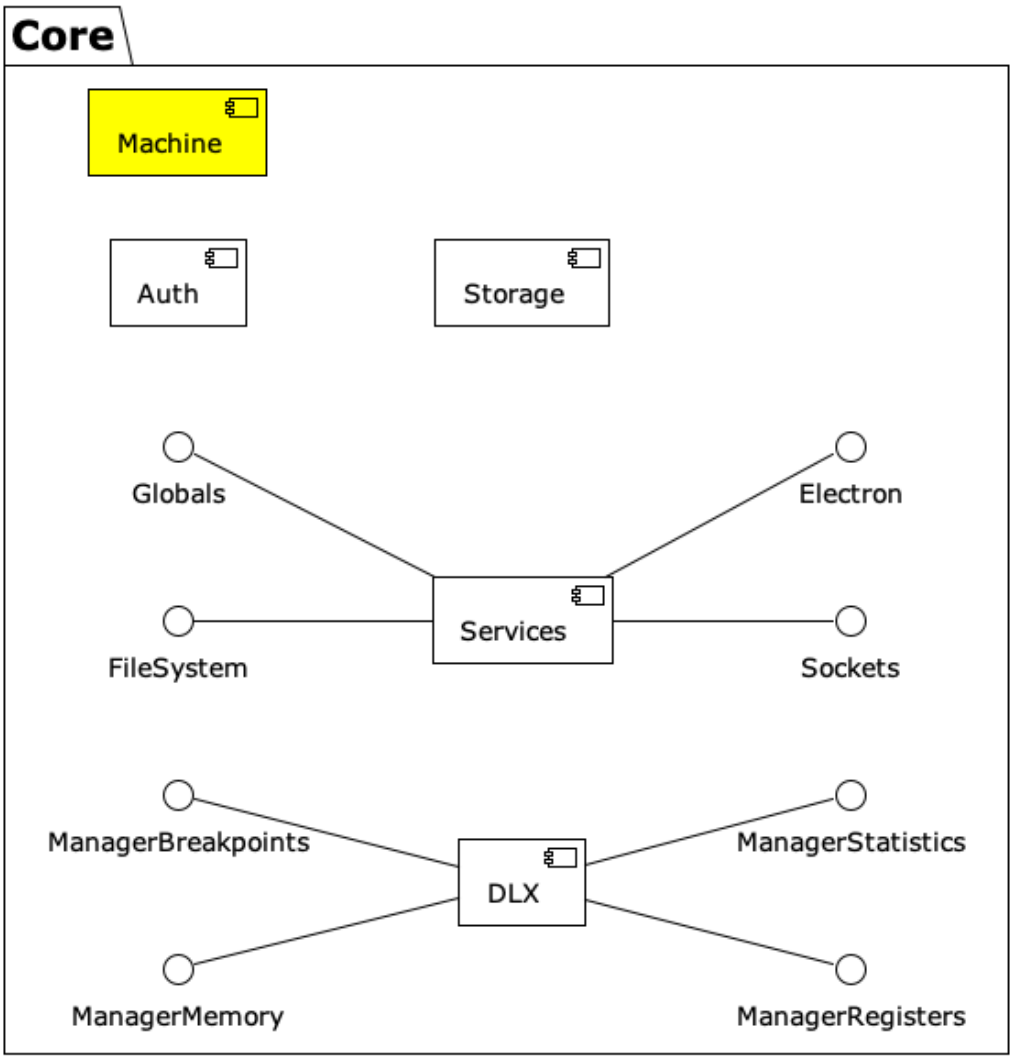


Figura 4.2: Diagrama del paquete del núcleo

4.2.2. Diagrama del paquete de disposición (`layout`)

El paquete de disposición o (*layout*) es el encargado de gestionar la interfaz de la aplicación en función del usuario y los permisos que este posea en función de la sesión.

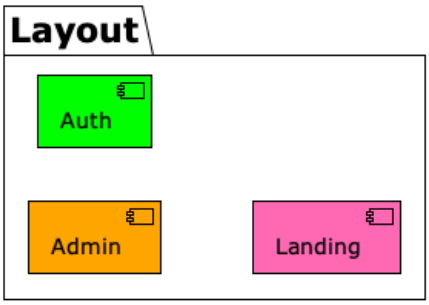


Figura 4.3: Diagrama del paquete de disposición

4.2.3. Diagrama del paquete de utilidades (`shared`)

El paquete de utilidades o (*shared*) es el encargado de transformar datos, comunicar información entre distintos componentes y de presentar errores internos de la aplicación.

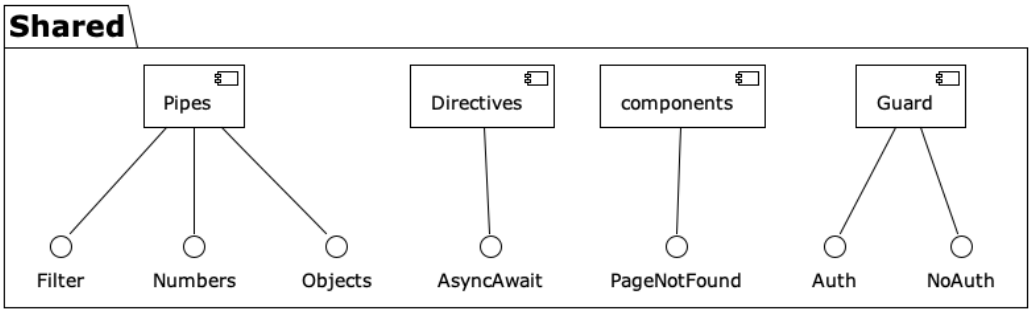


Figura 4.4: Diagrama del paquete de utilidades

4.2.4. Diagrama del paquete de componentes (`components`)

El paquete de componentes o (*components*) es el encargado de la lógica de cada componente por separado, encapsula la información que recibe de la capa superior para reutilizar cada componente.

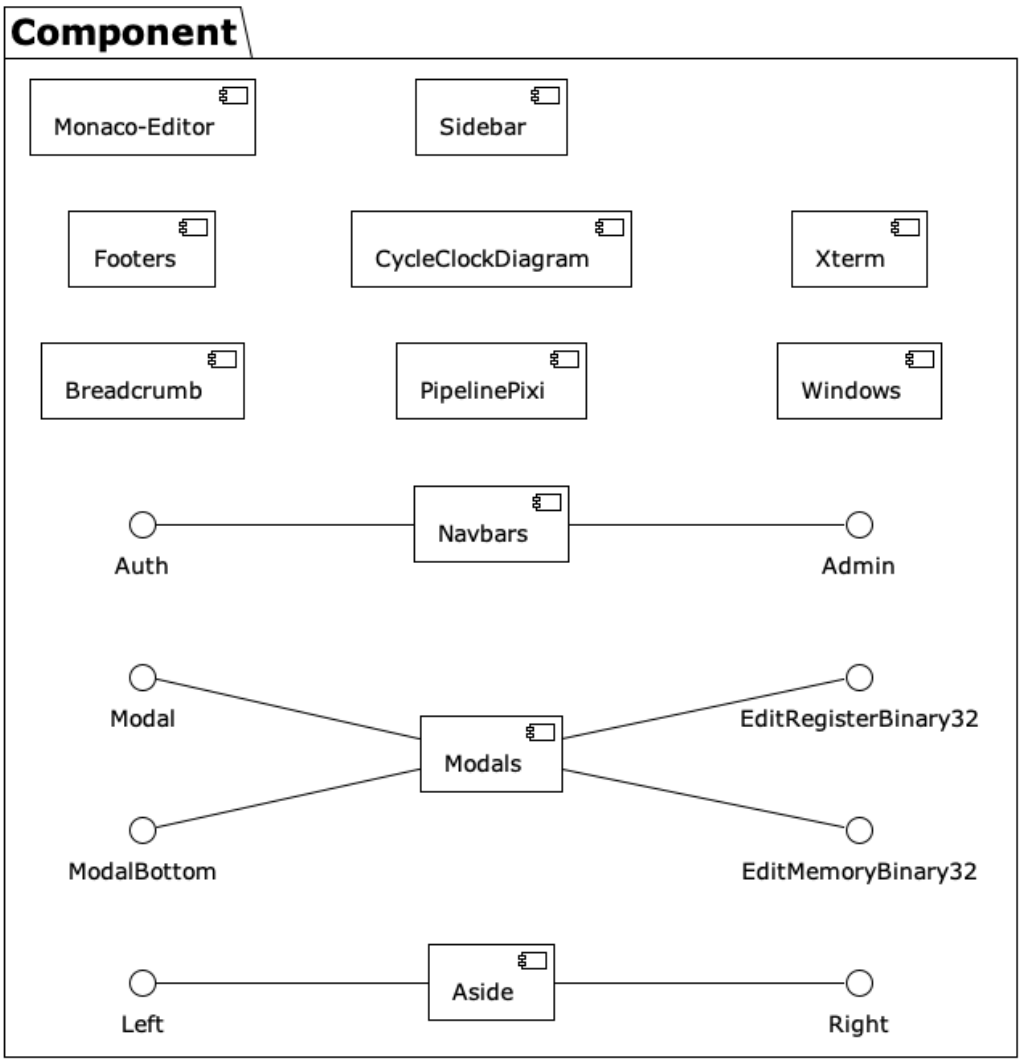


Figura 4.5: Diagrama del paquete de componentes

4.2.5. Diagrama del paquete de vistas (`views`)

El paquete de vistas o (*views*) es el encargado de presentar la interfaz adecuada, ajustar las dimensiones, tratar los estilos de la aplicación y comunicar la información entre el núcleo y los componentes.

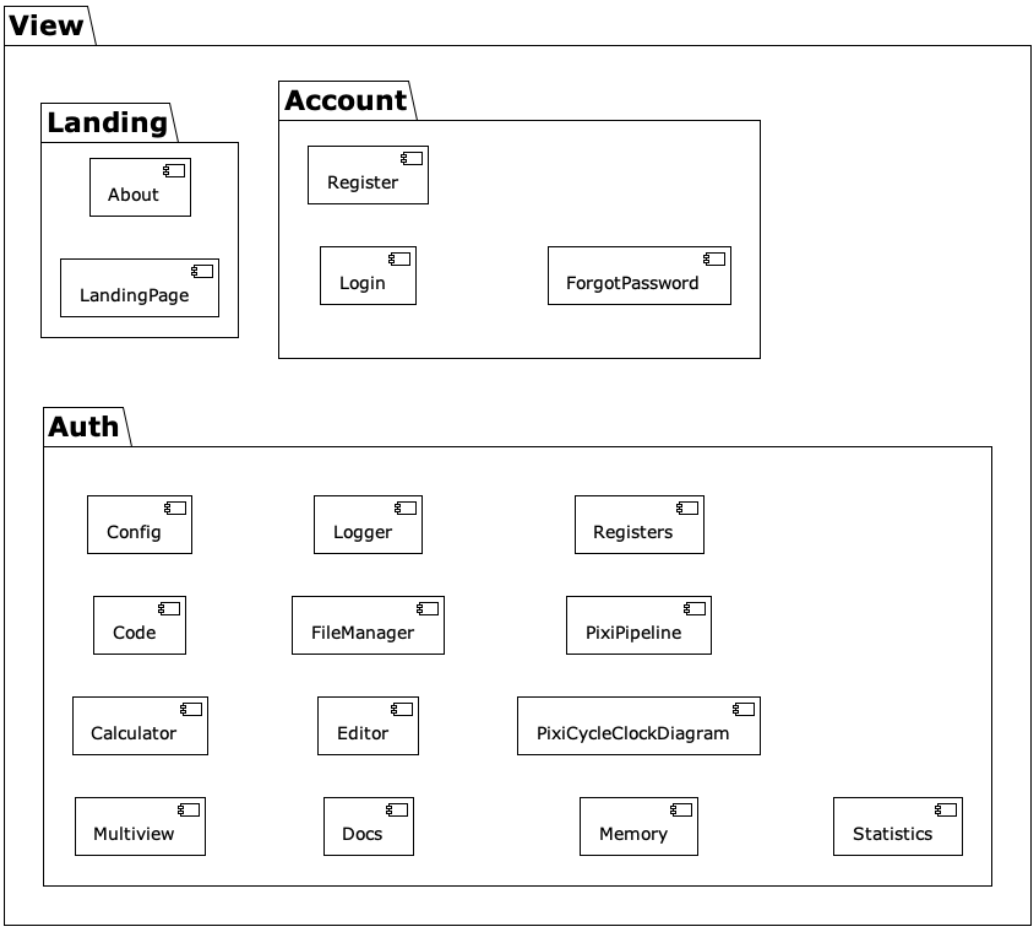


Figura 4.6: Diagrama del paquete de vistas

4.3. Diagramas de clase

Una vez que se ha explicado el diseño de los paquetes, debemos ir a las divisiones que se han realizado en los diagramas de clase, las clases del *núcleo*, de los **componentes** y de las **vistas**.

En los siguientes diagramas vamos a representar las distintas clases y métodos del proyecto, para ello hay que entender la división que se ha seguido, dicha división o arquitectura que usamos es la que proponen ciertas tecnologías webs, dicha arquitectura se basa y usa la programación orientada a eventos para compartir información entre distintas partes del proyecto.

1. En primer lugar, se ha definido los `components`, estas serían partes de diseño y lógica de pequeños elementos en la interfaz no relacionados con la lógica del DLX, son pequeños componentes típicos (*aside*, *navbar*, *footer*, *cards*, etc.), estos componentes son necesarios para casi todos los proyectos webs que uno se enfrenta, aunque en este caso se han adaptado ligeramente a las necesidades del proyecto.

2. A continuación, también se ha definido el módulo *views*, en este módulo ya sí entramos en vistas que representan cómo funciona una simulación de DLX, estas vistas representan la memoria, los registros, el cauce de ejecución (*pipeline*) y otras vistas de utilidades como la calculadora, el editor, etc.

3. Por otra parte, se ha definido el módulo *layout* que nos permite representar las vistas y componentes, organizando la forma y disposición en la que se agrupan las vistas y componentes, pudiendo diferenciar los distintos usuarios que pueden usar la aplicación (usuario identificado o usuario no identificado), esto nos permitirá si en un futuro queremos añadir roles como un administrador poder gestionar estas herramientas y desarrollarlas.

4. Para controlar ciertos aspectos de la web, representación de datos, control de rutas, funcionalidades añadidas a los componentes y representación de códigos de error como el 404 o el 500 lo gestionamos desde *shared*, este módulo gestiona los métodos que se usan para transformar datos, validar secciones y extender características, *shared* es una sección conocida como útiles.

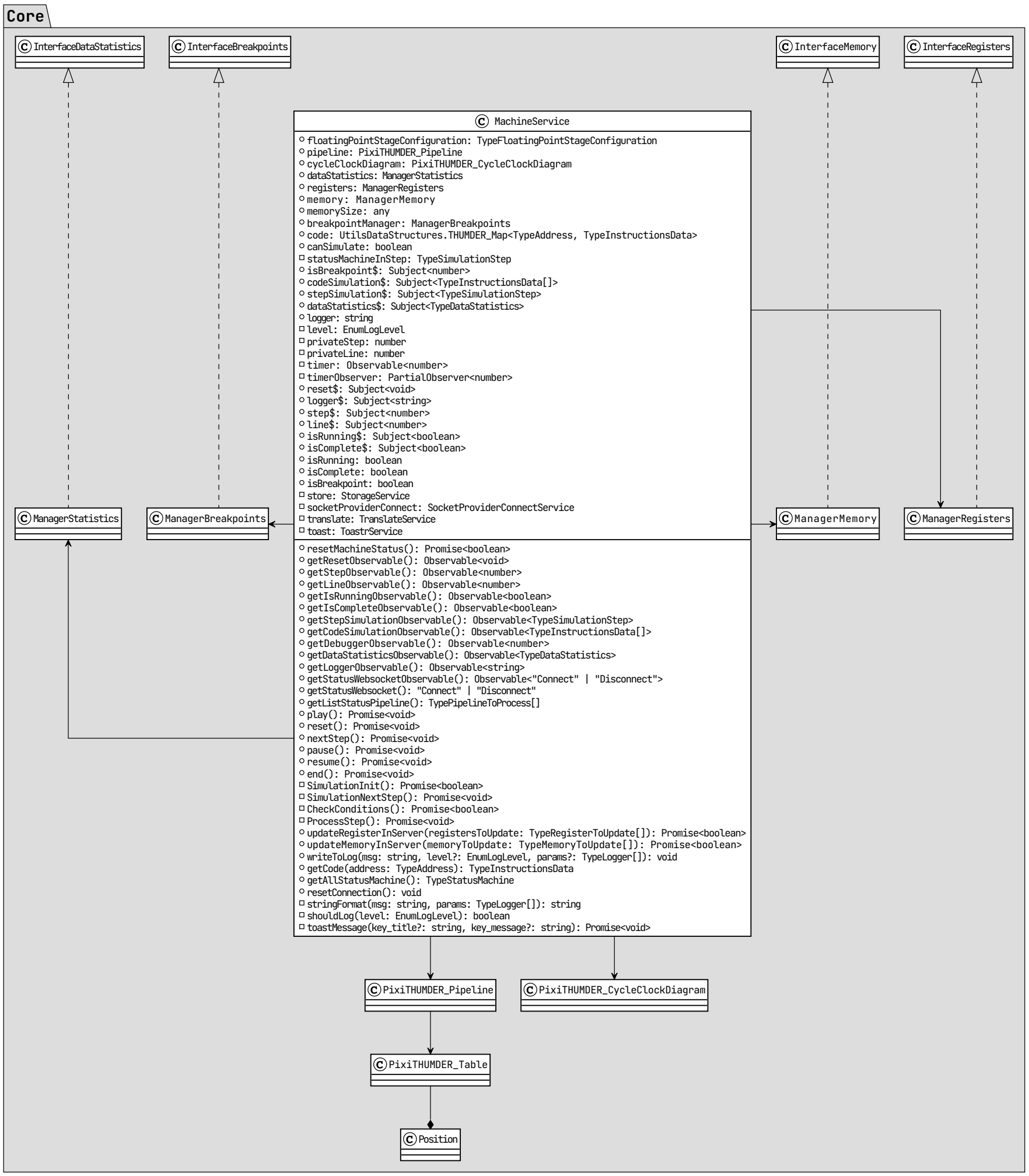
5. Por último, tenemos el módulo *core*, este es el encargado de la lógica de nuestro proyecto, la comunicación con el servidor, la autenticación y el guardado de datos con nuestro modelo de datos (base de datos) y la tarea más importante, el control del estado de la máquina, por lo que envía información sobre la memoria, registros, instrucciones, estadísticas y el estado de la simulación.

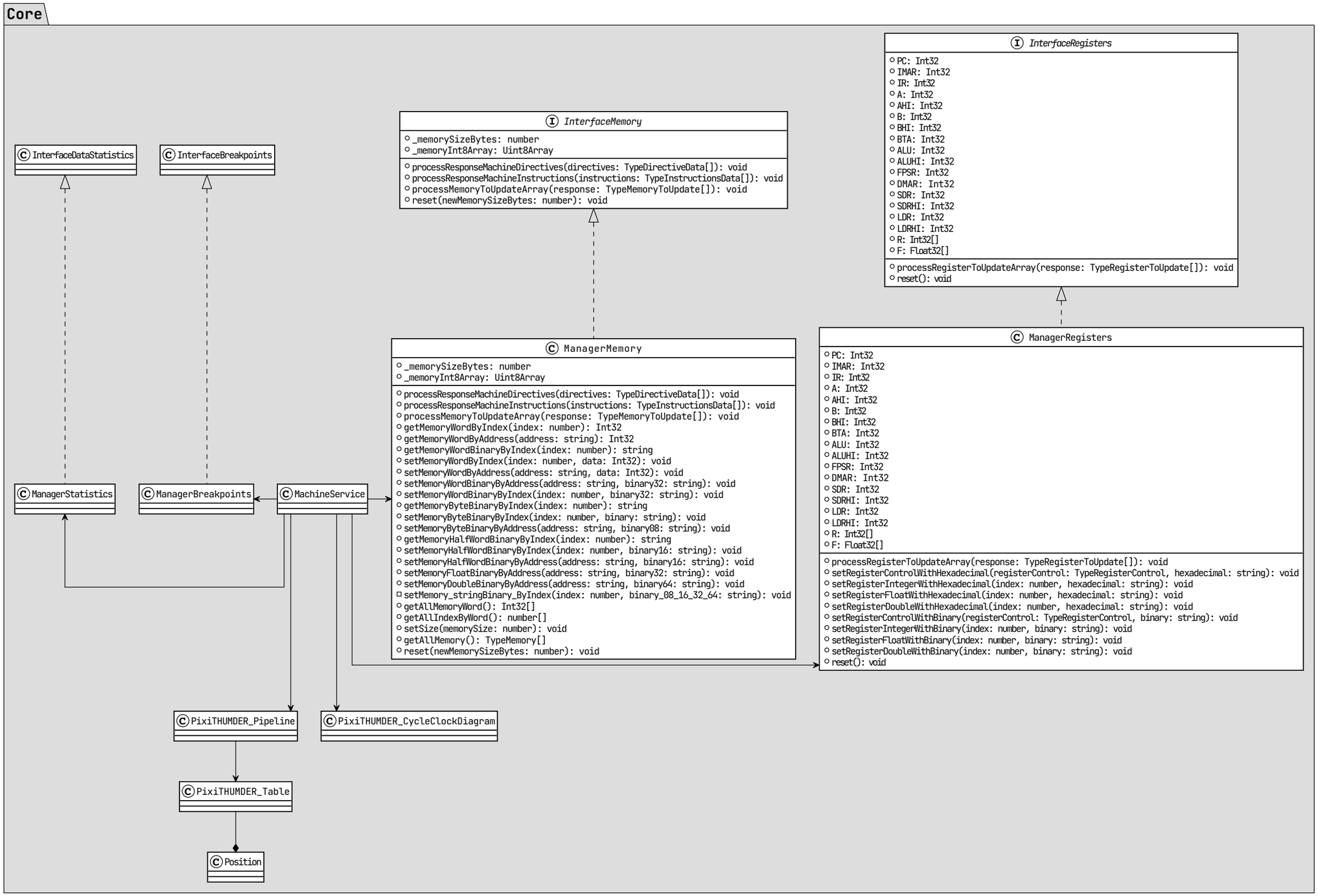
4.3.1. Diagramas de clases del núcleo

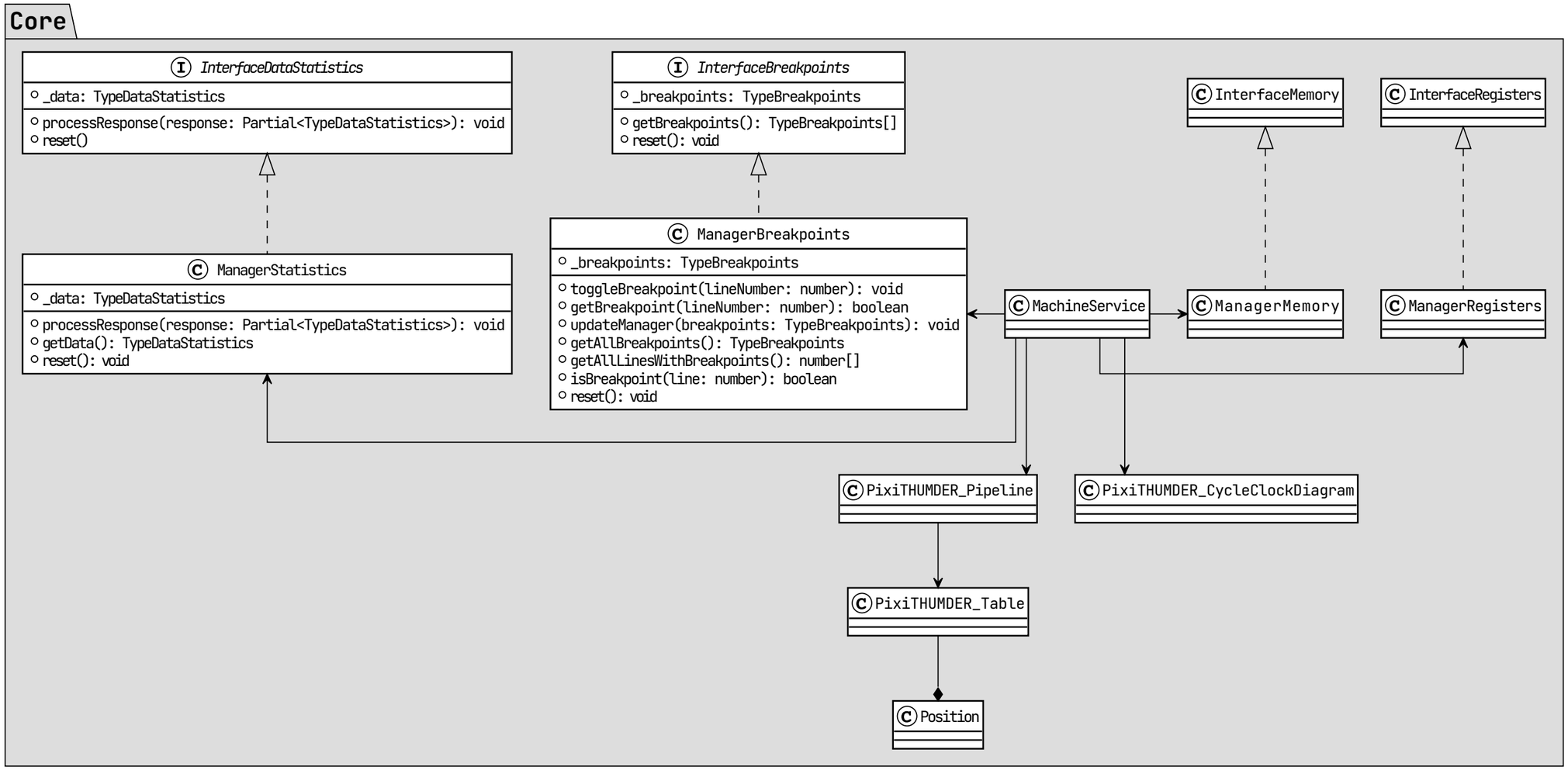
A continuación, se muestran los diagramas de clases del núcleo (*core*) por completo, estos se han dividido en varias páginas para que se pueda leer correctamente, pues son muy grandes para ser representados en una sola página.

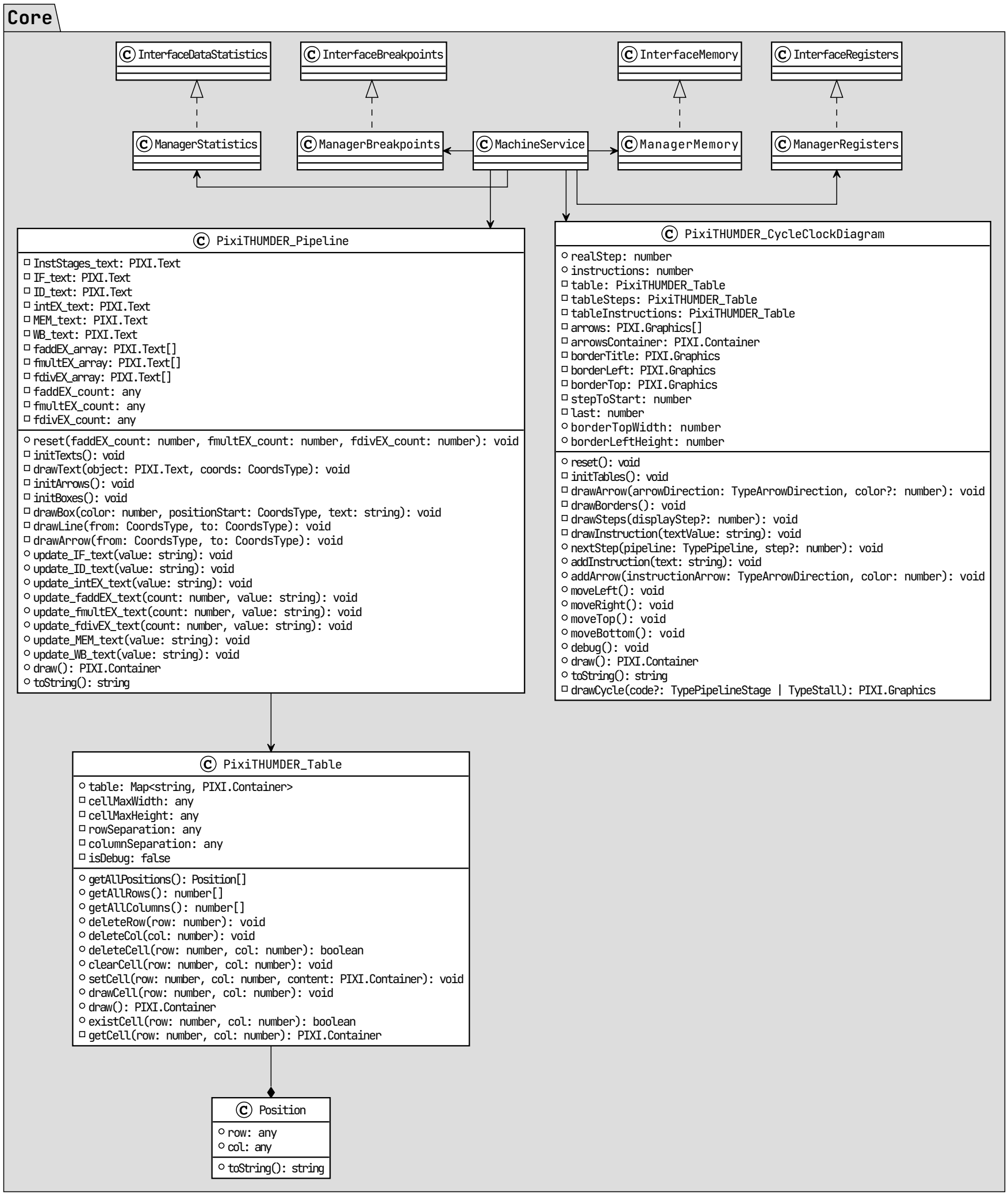
Además, también se muestran los diagramas de clases de la gestión de los archivos (`THUMDER_FILE`), junto a sus interfaces y los demás servicios del núcleo.

Como se aprecia, se han representado las distintas abstracciones de la simulación (memoria, registros, puntos de ruptura, estadísticas, cauce de ejecución y el diagrama de ciclos), se han indicado también las relaciones que existen con la clase `MachineService`.

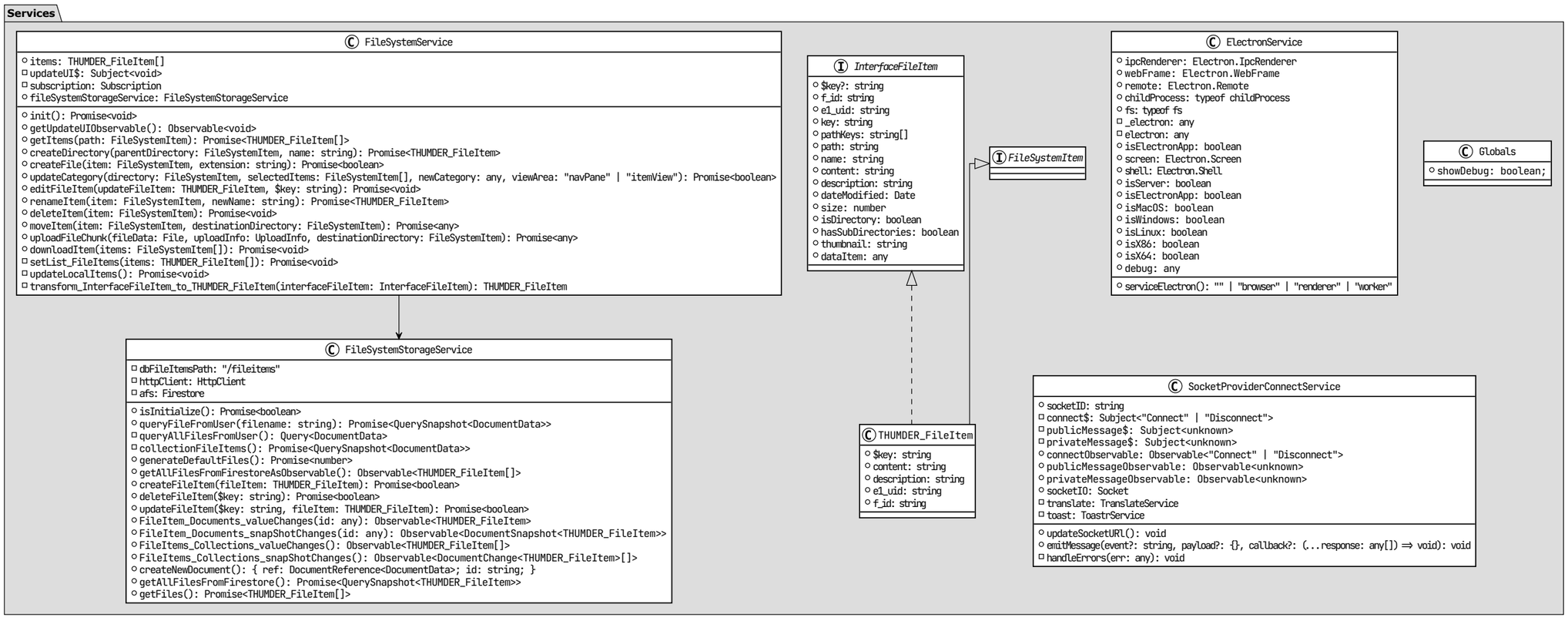

Core
InterfaceDataStatistics
InterfaceBreakpoints
InterfaceMemory
InterfaceRegisters
MachineService
floatingPointStageConfiguration: TypeFloatingPointStageConfiguration
pipeline: PixiTHUMDER_Pipeline
cycleClockDiagram: PixiTHUMDER_CycleClockDiagram
dataStatistics: ManagerStatistics
registers: ManagerRegisters
memory: ManagerMemory
memorySize: any
breakpointManager: ManagerBreakpoints
code: UtilsDataStructures.THUMDER_Map<TypeAddress, TypeInstructionsData>
canSimulate: boolean
statusMachineInStep: TypeSimulationStep
isBreakpoint$: Subject<number>
codeSimulation$: Subject<TypeInstructionsData[]>
stepSimulation$: Subject<TypeSimulationStep>
dataStatistics$: Subject<TypeDataStatistics>
logger: string
level: EnumLogLevel
privateStep: number
privateLine: number
timer: Observable<number>
timerObserver: PartialObserver<number>
reset$: Subject<void>
logger$: Subject<string>
step$: Subject<number>
line$: Subject<number>
isRunning$: Subject<boolean>
isComplete$: Subject<boolean>
isRunning: boolean
isComplete: boolean
isBreakpoint: boolean
store: StorageService
socketProviderConnect: SocketProviderConnectService
translate: TranslateService
toast: ToastrService
resetMachineStatus(): Promise<boolean>
getResetObservable(): Observable<void>
getStepObservable(): Observable<number>
getLineObservable(): Observable<number>
getIsRunningObservable(): Observable<boolean>
getIsCompleteObservable(): Observable<boolean>
getStepSimulationObservable(): Observable<TypeSimulationStep>
getCodeSimulationObservable(): Observable<TypeInstructionsData[]>
getDebuggerObservable(): Observable<number>
getDataStatisticsObservable(): Observable<TypeDataStatistics>
getLoggerObservable(): Observable<string>
getStatusWebsocketObservable(): Observable<"Connect" | "Disconnect">
getStatusWebsocket(): "Connect" | "Disconnect"
getListStatusPipeline(): TypePipelineToProcess[]
play(): Promise<void>
reset(): Promise<void>
nextStep(): Promise<void>
pause(): Promise<void>
resume(): Promise<void>
end(): Promise<void>
SimulationInit(): Promise<boolean>
SimulationNextStep(): Promise<void>
CheckConditions(): Promise<boolean>
ProcessStep(): Promise<void>
updateRegisterInServer(registersToUpdate: TypeRegisterToUpdate[]): Promise<boolean>
updateMemoryInServer(memoryToUpdate: TypeMemoryToUpdate[]): Promise<boolean>
writeToLog(msg: string, level?: EnumLogLevel, params?: TypeLogger[]): void
getCode(address: TypeAddress): TypeInstructionsData
getAllStatusMachine(): TypeStatusMachine
resetConnection(): void
stringFormat(msg: string, params: TypeLogger[]): string
shouldLog(level: EnumLogLevel): boolean
toastMessage(key_title?: string, key_message?: string): Promise<void>
ManagerStatistics
ManagerBreakpoints
ManagerMemory
ManagerRegisters
PixiTHUMDER_Pipeline
PixiTHUMDER_CycleClockDiagram
PixiTHUMDER_Table
Position


Core
InterfaceDataStatistics
InterfaceBreakpoints
InterfaceMemory
_memorySizeBytes: number
_memoryInt8Array: Uint8Array
processResponseMachineDirectives(directives: TypeDirectiveData[]): void
processResponseMachineInstructions(instructions: TypeInstructionsData[]): void
processMemoryToUpdateArray(response: TypeMemoryToUpdate[]): void
reset(newMemorySizeBytes: number): void
InterfaceRegisters
PC: Int32
IMAR: Int32
IR: Int32
A: Int32
AHI: Int32
B: Int32
BHI: Int32
BTA: Int32
ALU: Int32
ALUHI: Int32
FPSR: Int32
DMAR: Int32
SDR: Int32
SDRHI: Int32
LDR: Int32
LDRHI: Int32
R: Int32[]
F: Float32[]
processRegisterToUpdateArray(response: TypeRegisterToUpdate[]): void
reset(): void
ManagerMemory
_memorySizeBytes: number
_memoryInt8Array: Uint8Array
processResponseMachineDirectives(directives: TypeDirectiveData[]): void
processResponseMachineInstructions(instructions: TypeInstructionsData[]): void
processMemoryToUpdateArray(response: TypeMemoryToUpdate[]): void
getMemoryWordByIndex(index: number): Int32
getMemoryWordByAddress(address: string): Int32
getMemoryWordBinaryByIndex(index: number): string
setMemoryWordByIndex(index: number, data: Int32): void
setMemoryWordByAddress(address: string, data: Int32): void
setMemoryWordBinaryByAddress(address: string, binary32: string): void
setMemoryWordBinaryByIndex(index: number, binary32: string): void
getMemoryByteBinaryByIndex(index: number): string
setMemoryByteBinaryByIndex(index: number, binary: string): void
setMemoryByteBinaryByAddress(address: string, binary08: string): void
getMemoryHalfWordBinaryByIndex(index: number): string
setMemoryHalfWordBinaryByIndex(index: number, binary16: string): void
setMemoryHalfWordBinaryByAddress(address: string, binary16: string): void
setMemoryFloatBinaryByAddress(address: string, binary32: string): void
setMemoryDoubleBinaryByAddress(address: string, binary64: string): void
setMemory_stringBinary_ByIndex(index: number, binary_08_16_32_64: string): void
getAllMemoryWord(): Int32[]
getAllIndexByWord(): number[]
setSize(memorySize: number): void
getAllMemory(): TypeMemory[]
reset(newMemorySizeBytes: number): void
ManagerRegisters
PC: Int32
IMAR: Int32
IR: Int32
A: Int32
AHI: Int32
B: Int32
BHI: Int32
BTA: Int32
ALU: Int32
ALUHI: Int32
FPSR: Int32
DMAR: Int32
SDR: Int32
SDRHI: Int32
LDR: Int32
LDRHI: Int32
R: Int32[]
F: Float32[]
processRegisterToUpdateArray(response: TypeRegisterToUpdate[]): void
setRegisterControlWithHexadecimal(registerControl: TypeRegisterControl, hexadecimal: string): void
setRegisterIntegerWithHexadecimal(index: number, hexadecimal: string): void
setRegisterFloatWithHexadecimal(index: number, hexadecimal: string): void
setRegisterDoubleWithHexadecimal(index: number, hexadecimal: string): void
setRegisterControlWithBinary(registerControl: TypeRegisterControl, binary: string): void
setRegisterIntegerWithBinary(index: number, binary: string): void
setRegisterFloatWithBinary(index: number, binary: string): void
setRegisterDoubleWithBinary(index: number, binary: string): void
reset(): void
ManagerStatistics
ManagerBreakpoints
MachineService
PixiTHUMDER_Pipeline
PixiTHUMDER_CycleClockDiagram
PixiTHUMDER_Table
Position


Core
I InterfaceDataStatistics
_data: TypeDataStatistics
processResponse(response: Partial<TypeDataStatistics>): void
reset()
I InterfaceBreakpoints
_breakpoints: TypeBreakpoints
getBreakpoints(): TypeBreakpoints[]
reset(): void
C InterfaceMemory
C InterfaceRegisters
C ManagerStatistics
_data: TypeDataStatistics
processResponse(response: Partial<TypeDataStatistics>): void
getData(): TypeDataStatistics
reset(): void
C ManagerBreakpoints
_breakpoints: TypeBreakpoints
toggleBreakpoint(lineNumber: number): void
getBreakpoint(lineNumber: number): boolean
updateManager(breakpoints: TypeBreakpoints): void
getAllBreakpoints(): TypeBreakpoints
getAllLinesWithBreakpoints(): number[]
isBreakpoint(line: number): boolean
reset(): void
C MachineService
C ManagerMemory
C ManagerRegisters
C PixiTHUMDER_Pipeline
C PixiTHUMDER_CycleClockDiagram
C PixiTHUMDER_Table
C Position


Core
InterfaceDataStatistics
InterfaceBreakpoints
InterfaceMemory
InterfaceRegisters
ManagerStatistics
ManagerBreakpoints
MachineService
ManagerMemory
ManagerRegisters
PixiTHUMDER_Pipeline
InstStages_text: PIXI.Text
IF_text: PIXI.Text
ID_text: PIXI.Text
intEX_text: PIXI.Text
MEM_text: PIXI.Text
WB_text: PIXI.Text
faddEX_array: PIXI.Text[]
fmultEX_array: PIXI.Text[]
fdivEX_array: PIXI.Text[]
faddEX_count: any
fmultEX_count: any
fdivEX_count: any
reset(faddEX_count: number, fmultEX_count: number, fdivEX_count: number): void
initTexts(): void
drawText(object: PIXI.Text, coords: CoordsType): void
initArrows(): void
initBoxes(): void
drawBox(color: number, positionStart: CoordsType, text: string): void
drawLine(from: CoordsType, to: CoordsType): void
drawArrow(from: CoordsType, to: CoordsType): void
update_IF_text(value: string): void
update_ID_text(value: string): void
update_intEX_text(value: string): void
update_faddEX_text(count: number, value: string): void
update_fmultEX_text(count: number, value: string): void
update_fdivEX_text(count: number, value: string): void
update_MEM_text(value: string): void
update_WB_text(value: string): void
draw(): PIXI.Container
toString(): string
PixiTHUMDER_CycleClockDiagram
realStep: number
instructions: number
table: PixiTHUMDER_Table
tableSteps: PixiTHUMDER_Table
tableInstructions: PixiTHUMDER_Table
arrows: PIXI.Graphics[]
arrowsContainer: PIXI.Container
borderTitle: PIXI.Graphics
borderLeft: PIXI.Graphics
borderTop: PIXI.Graphics
stepToStart: number
last: number
borderTopWidth: number
borderLeftHeight: number
reset(): void
initTables(): void
drawArrow(arrowDirection: TypeArrowDirection, color?: number): void
drawBorders(): void
drawSteps(displayStep?: number): void
drawInstruction(textValue: string): void
nextStep(pipeline: TypePipeline, step?: number): void
addInstruction(text: string): void
addArrow(instructionArrow: TypeArrowDirection, color: number): void
moveLeft(): void
moveRight(): void
moveTop(): void
moveBottom(): void
debug(): void
draw(): PIXI.Container
toString(): string
drawCycle(code?: TypePipelineStage | TypeStall): PIXI.Graphics
PixiTHUMDER_Table
table: Map<string, PIXI.Container>
cellMaxWidth: any
cellMaxHeight: any
rowSeparation: any
columnSeparation: any
isDebug: false
getAllPositions(): Position[]
getAllRows(): number[]
getAllColumns(): number[]
deleteRow(row: number): void
deleteCol(col: number): void
deleteCell(row: number, col: number): boolean
clearCell(row: number, col: number): void
setCell(row: number, col: number, content: PIXI.Container): void
drawCell(row: number, col: number): void
draw(): PIXI.Container
existCell(row: number, col: number): boolean
getCell(row: number, col: number): PIXI.Container
Position
row: any
col: any
toString(): string

Services


FileSystemService
items: THUMDER_FileItem[]
updateUI$: Subject<void>
subscription: Subscription
fileSystemStorageService: FileSystemStorageService
init(): Promise<void>
getUpdateUIObservable(): Observable<void>
getItems(path: FileSystemItem): Promise<THUMDER_FileItem[]>
createDirectory(parentDirectory: FileSystemItem, name: string): Promise<THUMDER_FileItem>
createFile(item: FileSystemItem, extension: string): Promise<boolean>
updateCategory(directory: FileSystemItem, selectedItems: FileSystemItem[], newCategory: any, viewArea: "navPane" | "itemView"): Promise<boolean>
editFileItem(updateFileItem: THUMDER_FileItem, $key: string): Promise<void>
renameItem(item: FileSystemItem, newName: string): Promise<THUMDER_FileItem>
deleteItem(item: FileSystemItem): Promise<void>
moveItem(item: FileSystemItem, destinationDirectory: FileSystemItem): Promise<any>
uploadFileChunk(fileData: File, uploadInfo: UploadInfo, destinationDirectory: FileSystemItem): Promise<any>
downloadItem(items: FileSystemItem[]): Promise<void>
setList_FileItems(items: THUMDER_FileItem[]): Promise<void>
updateLocalItems(): Promise<void>
transform_InterfaceFileItem_to_THUMDER_FileItem(interfaceFileItem: InterfaceFileItem): THUMDER_FileItem
InterfaceFileItem
$key?: string
f_id: string
e1_uid: string
key: string
pathKeys: string[]
path: string
name: string
content: string
description: string
dateModified: Date
size: number
isDirectory: boolean
hasSubDirectories: boolean
thumbnail: string
dataItem: any
FileSystemItem
ElectronService
ipcRenderer: Electron.IpcRenderer
webFrame: Electron.WebFrame
remote: Electron.Remote
childProcess: typeof childProcess
fs: typeof fs
_electron: any
electron: any
isElectronApp: boolean
screen: Electron.Screen
shell: Electron.Shell
isServer: boolean
isElectronApp: boolean
isMacOS: boolean
isWindows: boolean
isLinux: boolean
isX86: boolean
isX64: boolean
debug: any
serviceElectron(): "" | "browser" | "renderer" | "worker"
Globals
showDebug: boolean;
FileSystemStorageService
dbFileItemsPath: "/fileitems"
httpClient: HttpClient
afs: Firestore
isInitialize(): Promise<boolean>
queryFileFromUser(filename: string): Promise<QuerySnapshot<DocumentData>>
queryAllFilesFromUser(): Query<DocumentData>
collectionFileItems(): Promise<QuerySnapshot<DocumentData>>
generateDefaultFiles(): Promise<number>
getAllFilesFromFirestoreAsObservable(): Observable<THUMDER_FileItem[]>
createFileItem(fileItem: THUMDER_FileItem): Promise<boolean>
deleteFileItem($key: string): Promise<boolean>
updateFileItem($key: string, fileItem: THUMDER_FileItem): Promise<boolean>
FileItem_Documents_valueChanges(id: any): Observable<THUMDER_FileItem>
FileItem_Documents_snapShotChanges(id: any): Observable<DocumentSnapshot<THUMDER_FileItem>>
FileItems_Collections_valueChanges(): Observable<THUMDER_FileItem[]>
FileItems_Collections_snapShotChanges(): Observable<DocumentChange<THUMDER_FileItem>[]>
createNewDocument(): { ref: DocumentReference<DocumentData>; id: string; }
getAllFilesFromFirestore(): Promise<QuerySnapshot<THUMDER_FileItem>>
getFiles(): Promise<THUMDER_FileItem[]>
THUMDER_FileItem
$key: string
content: string
description: string
e1_uid: string
f_id: string
SocketProviderConnectService
socketID: string
connect$: Subject<"Connect" | "Disconnect">
publicMessage$: Subject<unknown>
privateMessage$: Subject<unknown>
connectObservable: Observable<"Connect" | "Disconnect">
publicMessageObservable: Observable<unknown>
privateMessageObservable: Observable<unknown>
socketIO: Socket
translate: TranslateService
toast: ToastrService
updateSocketURl(): void
emitMessage(event?: string, payload?: {}, callback?: (...response: any[]) => void): void
handleErrors(err: any): void

4.3.2. Diagrama de clases de los componentes

Los componentes se han definido como las partes más simples de la interfaz, estos componentes son usados por las vistas.

Cada uno de los diagramas de los componentes y vistas cuenta con los siguientes métodos `ngOnInit(): void`, encargado de inicializar atributos y suscripciones, `ngAfterViewInit(): void`, encargado de actualizar la información de los atributos una vez se ha cargado la interfaz, `ngOnDestroy(): void` encargado de liberar la memoria y las suscripciones.

Se han organizado los diagramas de forma que se muestren primero los atributos, luego los métodos que controlan la interfaz, métodos de retrollamada (*callback*), suscripciones y por último los métodos propios y exclusivos de cada clase, aislando el comportamiento de cada parte.

4.3.2.1. Diagrama de clase del componente - Editor de memoria

Figura 4.7: Diagrama de clase del componente - Editor de memoria

4.3.2.2. Diagrama de clase del componente - Editor de registros

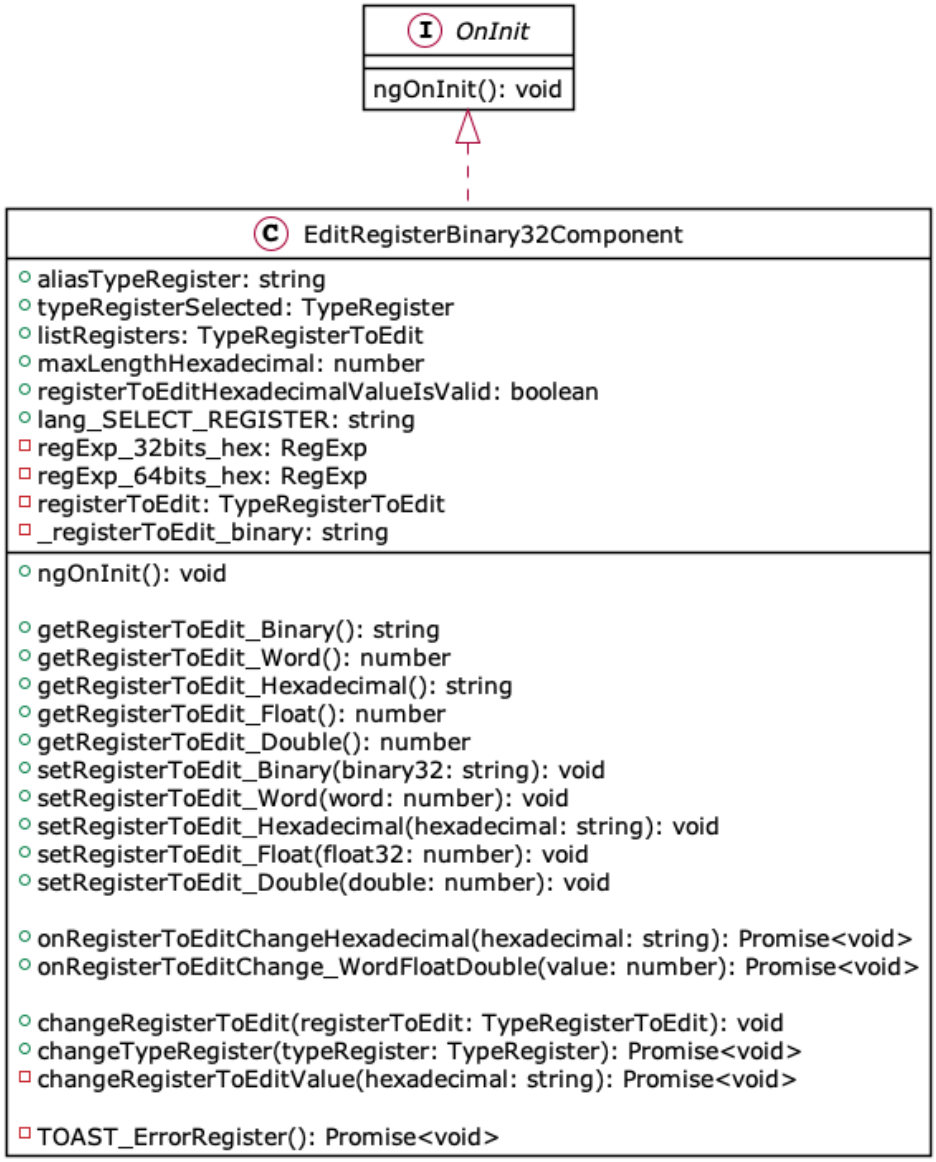


Figura 4.8: Diagrama de clase del componente - Editor de registros

4.3.2.3. Diagrama de clase del componente - Editor de ficheros

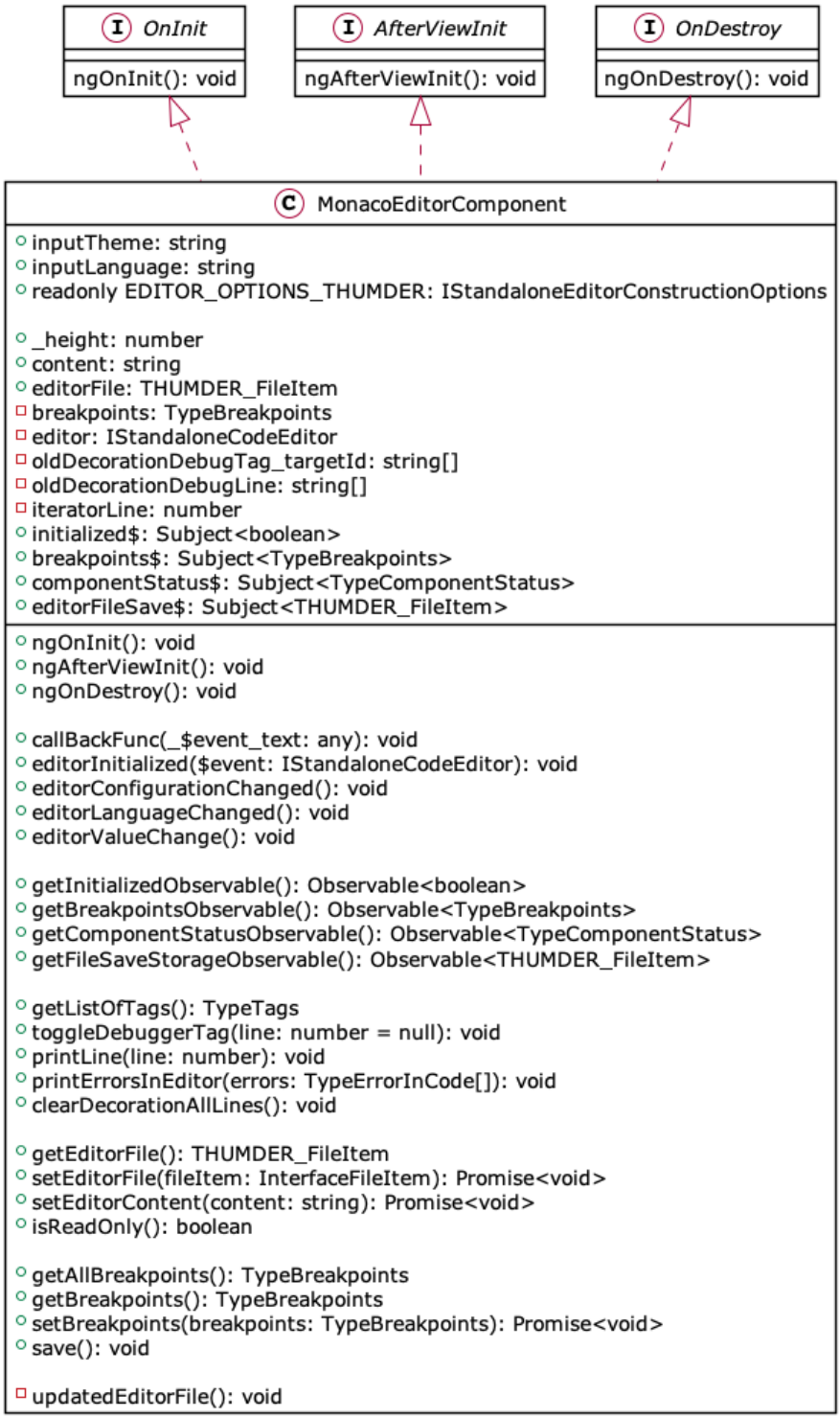


Figura 4.9: Diagrama de clase del componente - Editor de ficheros

4.3.2.4. Diagrama de clase del componente - Cauce de ejecución

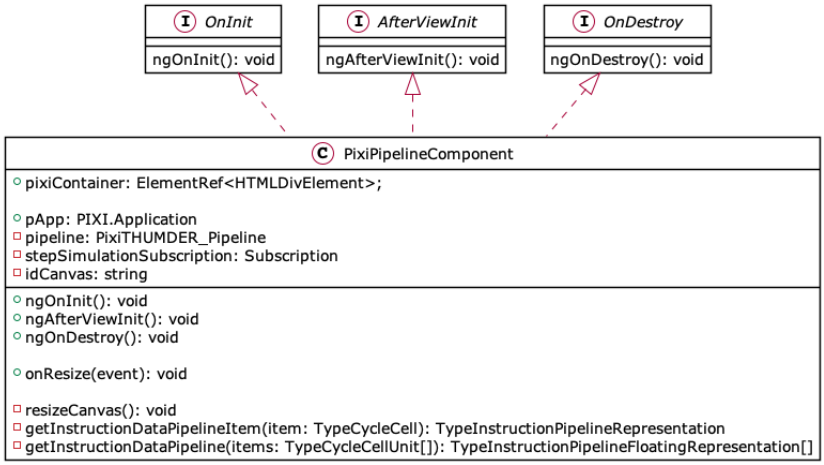


Figura 4.10: Diagrama de clase del componente - Cauce de ejecución

4.3.2.5. Diagrama de clase del componente - Ciclos de reloj

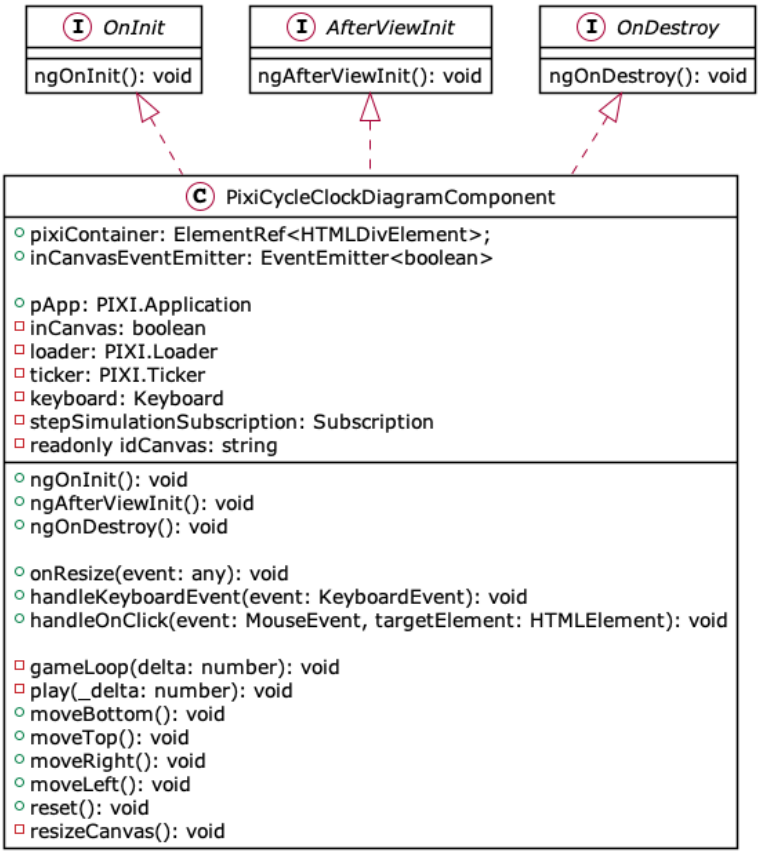


Figura 4.11: Diagrama de clase del componente - Ciclos de reloj

4.3.3. Diagrama de clases de las vistas

Las vistas son las distintas secciones que tiene nuestra aplicación, estas secciones utilizan los componentes y los servicios en el núcleo para representar la información.

4.3.3.1. Diagrama de clase de la vista - Calculadora

Figura 4.12: Diagrama de clase de la vista - Calculadora

4.3.3.2. Diagrama de clase de la vista - Código

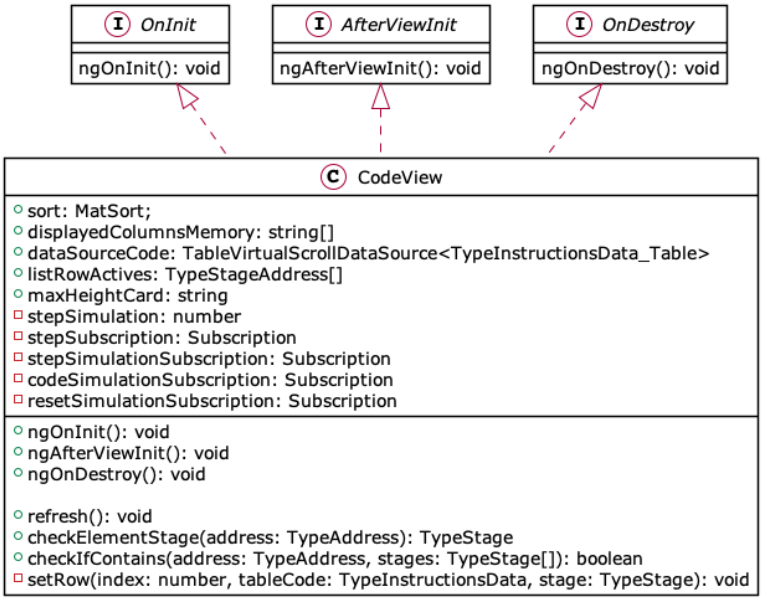


Figura 4.13: Diagrama de clase de la vista - Código

4.3.3.3. Diagrama de clase de la vista - Configuración

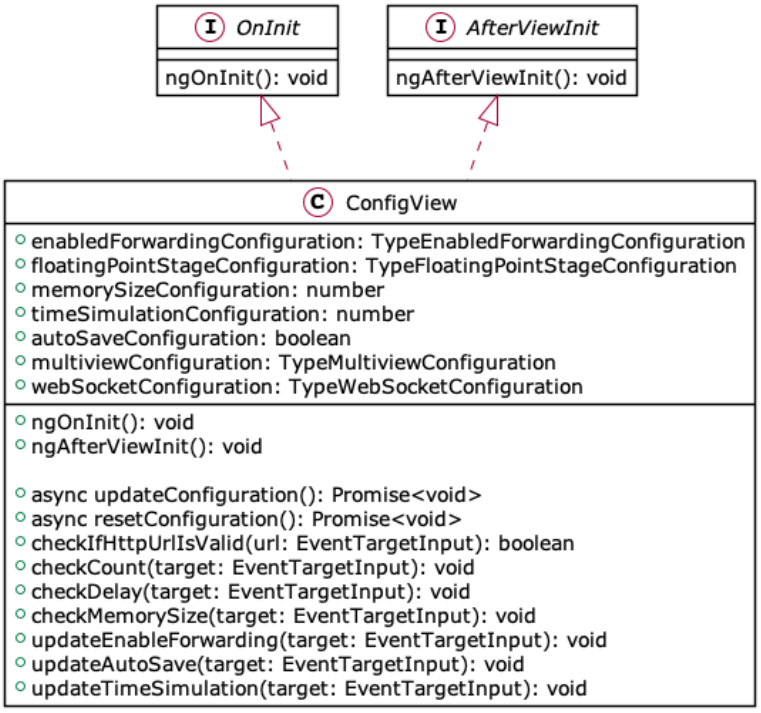


Figura 4.14: Diagrama de clase de la vista - Configuración

4.3.3.4. Diagrama de clase de la vista - Cauce de ejecución

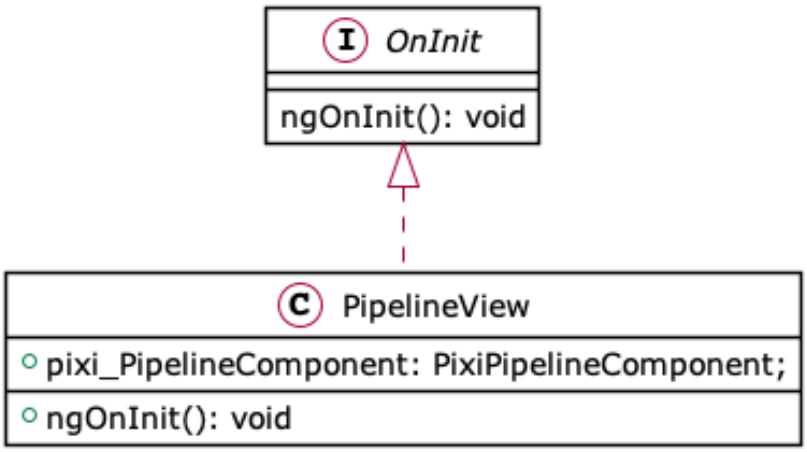


Figura 4.15: Diagrama de clase de la vista - Cauce de ejecución

4.3.3.5. Diagrama de clase de la vista - Ciclos de reloj

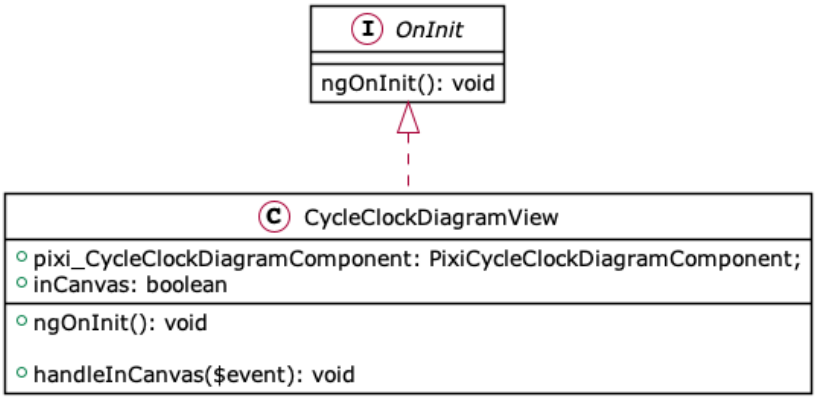


Figura 4.16: Diagrama de clase de la vista - Ciclos de reloj

4.3.3.6. Diagrama de clase de la vista - Documentación

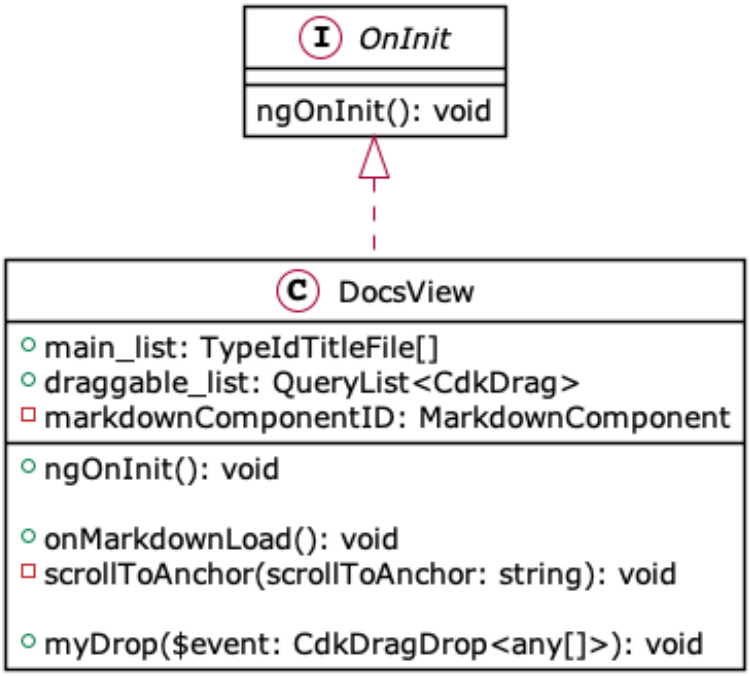


Figura 4.17: Diagrama de clase de la vista - Documentación

4.3.3.7. Diagrama de clase de la vista - Editor de ficheros

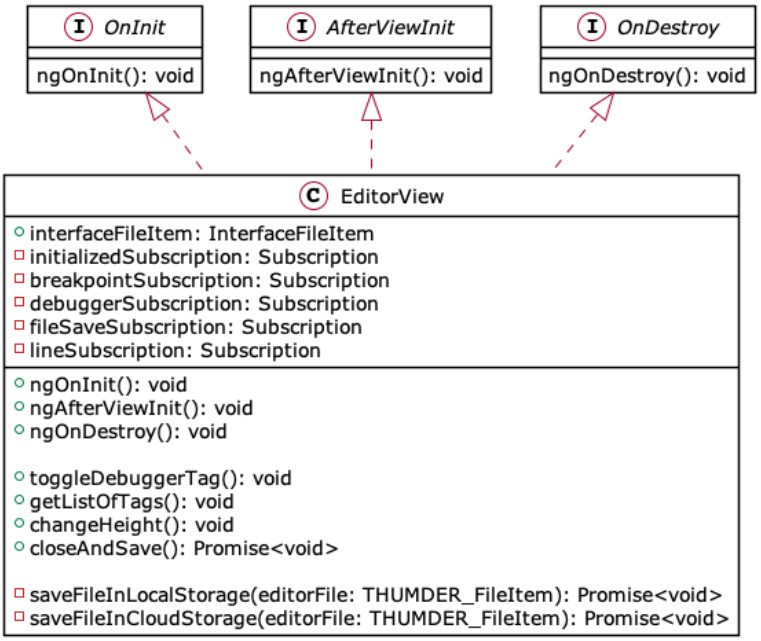


Figura 4.18: Diagrama de clase de la vista - Editor de ficheros

4.3.3.8. Diagrama de clase de la vista - Gestor de ficheros

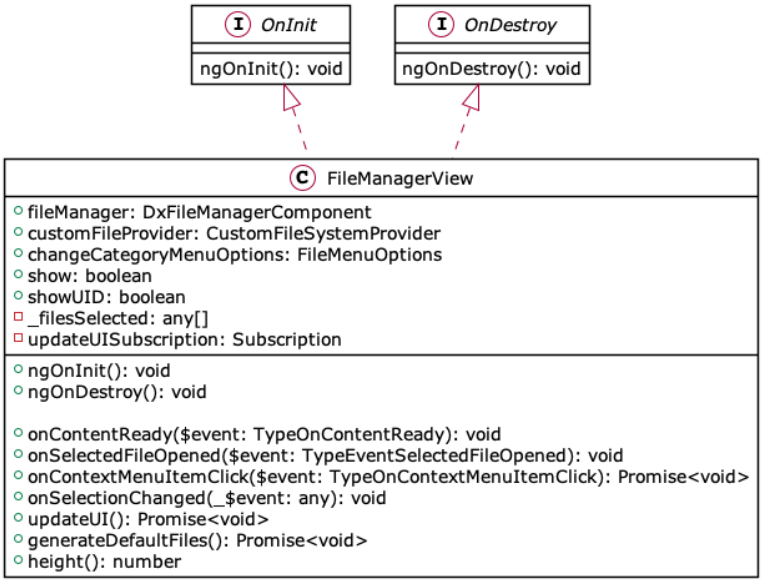


Figura 4.19: Diagrama de clase de la vista - Gestor de ficheros

4.3.3.9. Diagrama de clase de la vista - Memoria

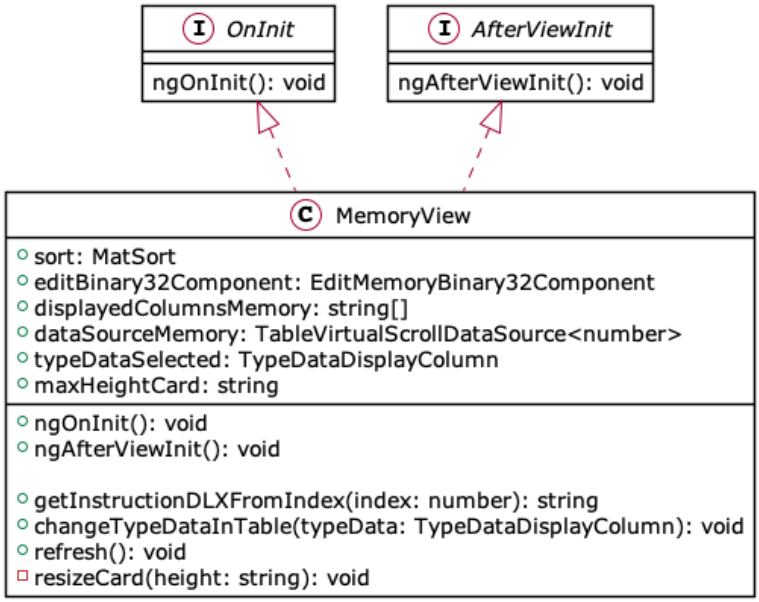


Figura 4.20: Diagrama de clase de la vista - Memoria

4.3.3.10. Diagrama de clase de la vista - Registros

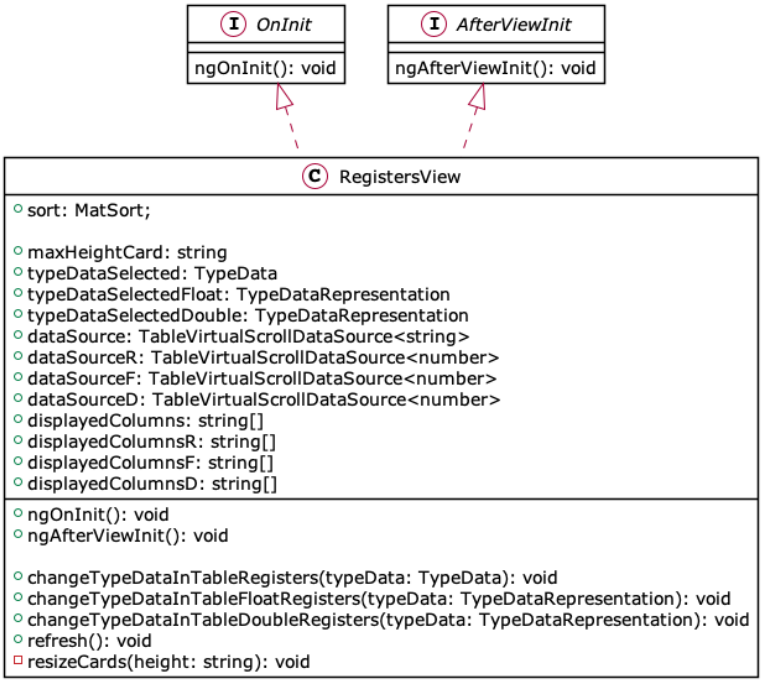


Figura 4.21: Diagrama de clase de la vista - Registros

4.3.3.11. Diagrama de clase de la vista - Estadísticas

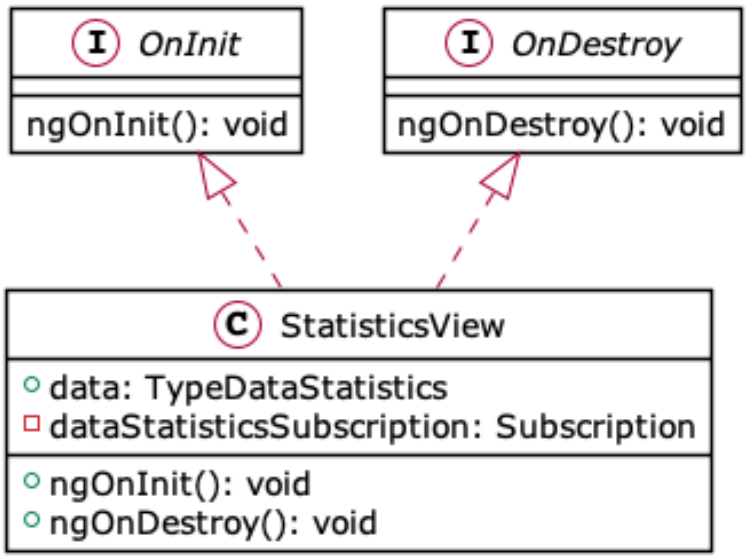


Figura 4.22: Diagrama de clase de la vista - Estadísticas

4.4. Diagramas de secuencia

En el diagrama de secuencia se presentan las tres entidades que hablamos al inicio del capítulo (usuario, sistema y servidor), ahora añadimos dos más (interfaz y proveedor). En total cinco entidades el usuario, la interfaz, el sistema, el servidor y el proveedor.

Para este proyecto se debe cumplir estrictamente un protocolo de comunicación, ya que el cliente y el servidor deben poder entenderse y saber qué tareas deben realizar cada uno, para ello, cada tarea que necesite de la respuesta o procesamiento del servidor deberá seguir ciertas reglas y pasos de comunicación.

4.4.1. Crear cuenta

Para acceder al servicio debemos registrarnos y para ello podremos usar distintos proveedores de servicios de autenticación (*OAuth*) o una forma más tradicional con el correo electrónico (*email*) y contraseña. Una vez seleccionada una forma podemos acceder al sistema mediante la cuenta. En caso de que se detecte un error enviará un mensaje con una descripción y un código del error, este se mostrará en pantalla con una notificación.

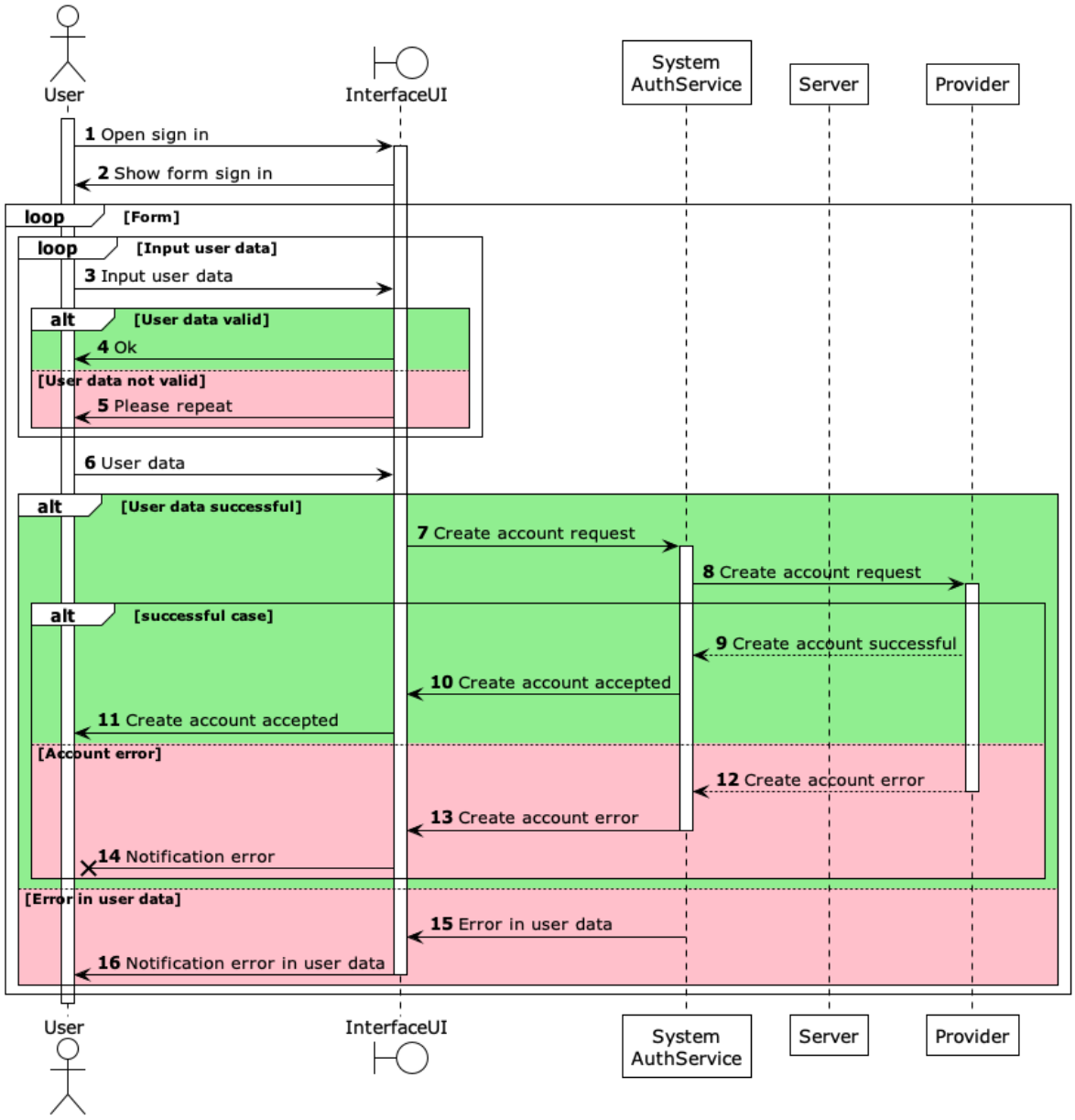


Figura 4.23: Diagrama de secuencia - Crear cuenta

4.4.2. Iniciar sesión

Una vez creada una cuenta de usuario, interesa poder acceder a la aplicación con la cuenta creada, como muestra el diagrama de secuencia para iniciar sesión en la aplicación con el usuario y el rol adecuado. Al usar un proveedor en caso de que se produzca algún error, se notificará con un mensaje descriptivo y un código que mostraremos en pantalla con una notificación.

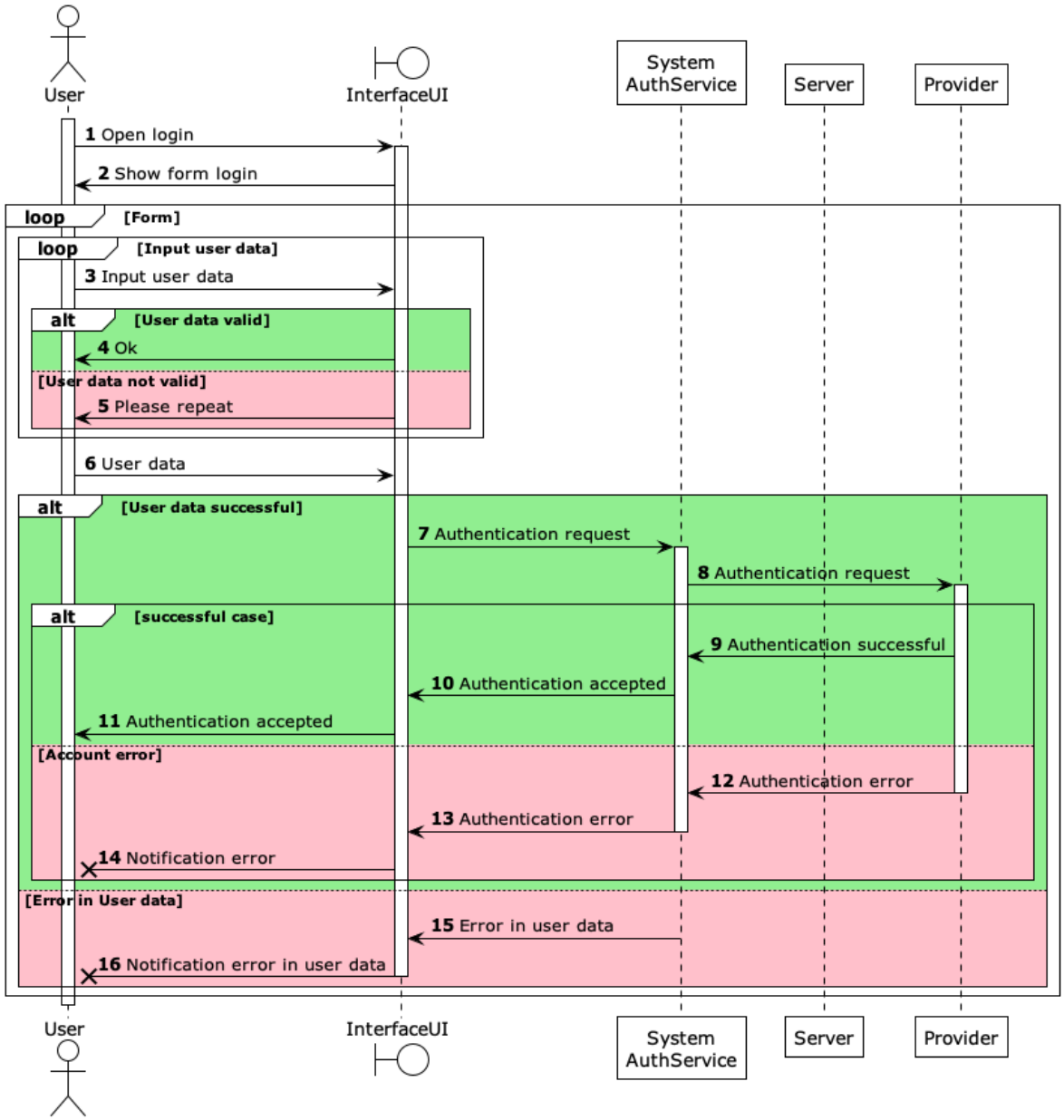


Figura 4.24: Diagrama de secuencia - Iniciar sesión

4.4.3. Mostrar información aplicación

En esta sección se puede consultar las distintas licencias, el `README` del proyecto, la política de *cookies* en la web y el historial de cambios por las versiones.

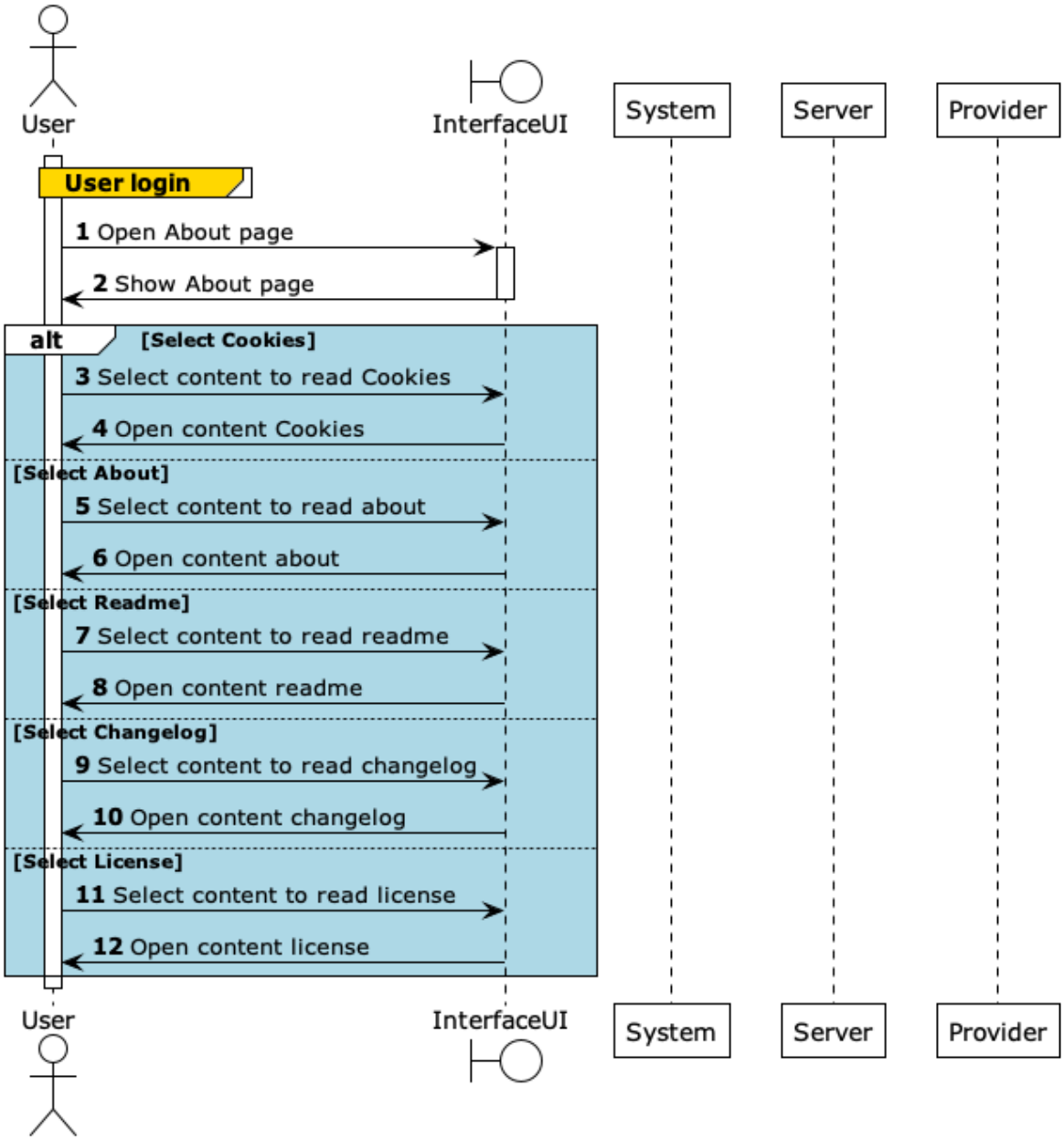


Figura 4.25: Diagrama de secuencia - Mostrar información aplicación

4.4.4. Crear archivo

Una vez que se ha iniciado la sesión se puede gestionar los archivos con el código que vamos desarrollando, para ello debemos poder crear archivos.

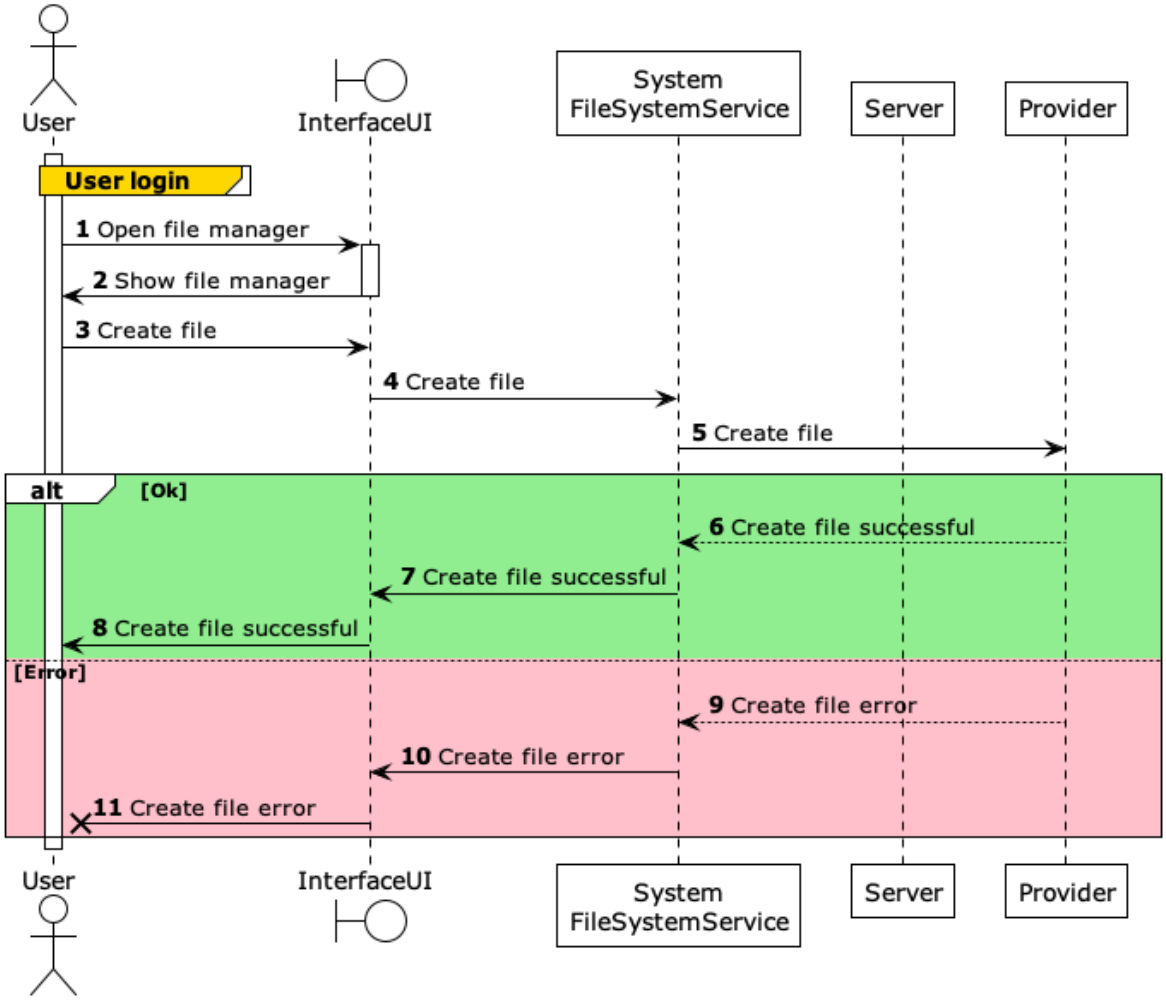


Figura 4.26: Diagrama de secuencia - Crear archivo

Se envía una petición al servidor con la petición de crear un archivo, este responderá con la confirmación positiva o negativa notificando si se ha producido algún error.

4.4.5. Borrar archivo

Al eliminar un archivo, el sistema advertirá de que es una acción destructiva y pedirá una confirmación.

Para esta tarea se selecciona un archivo y envía una petición al servidor con el identificador del archivo a borrar.

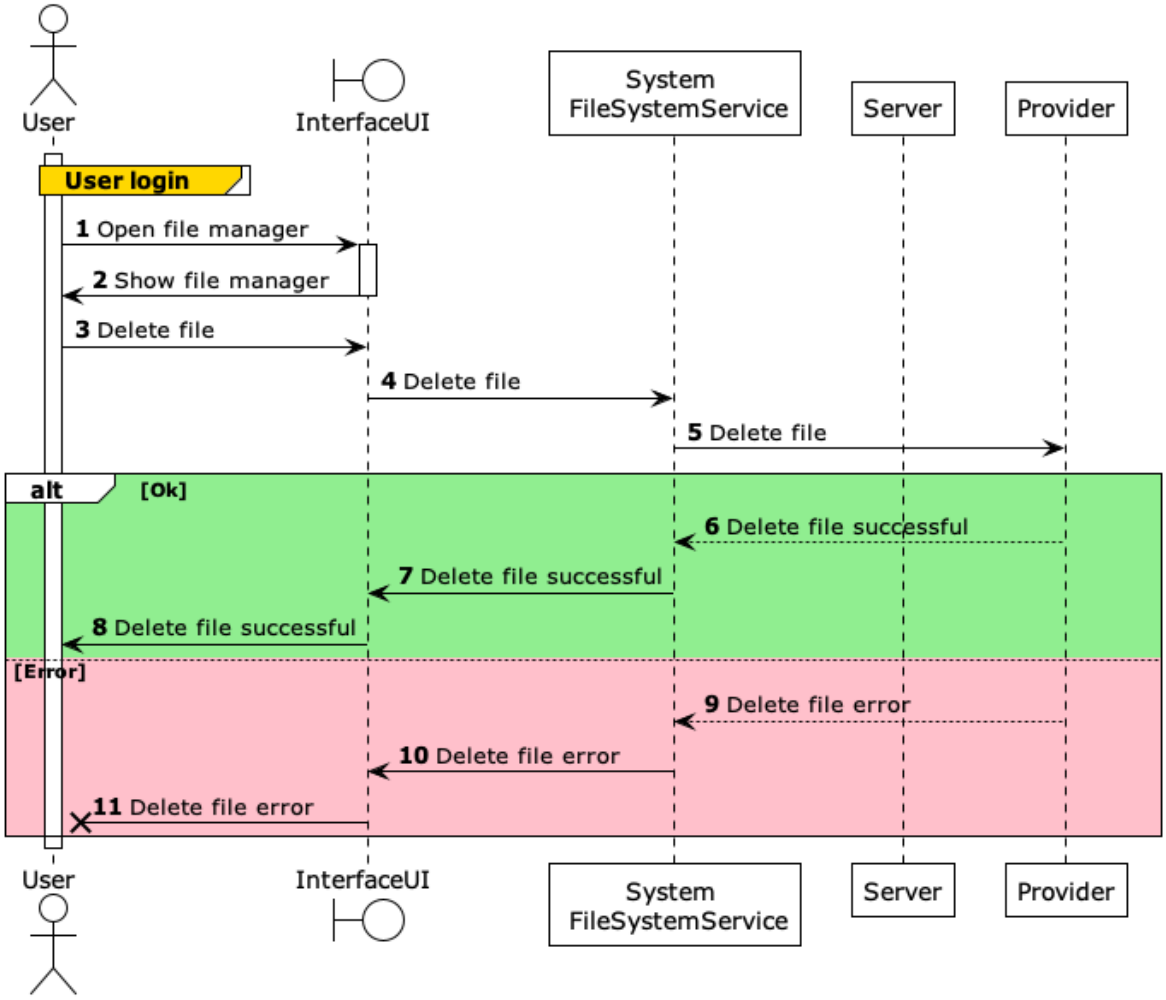


Figura 4.27: Diagrama de secuencia - Borrar archivo

El servidor responderá con la confirmación de la eliminación, en caso positivo se actualiza la interfaz y en caso negativo se notifica el error.

4.4.6. Renombrar archivo

Se puede modificar el nombre de los archivos sin que estos cambien el contenido.

Para esta tarea se selecciona un archivo y la opción de actualizar el nombre, a continuación, el sistema pedirá mediante una ventana modal el nuevo nombre del archivo, y el usuario confirma los cambios, el sistema envía una petición al servidor con la información del archivo a renombrar.

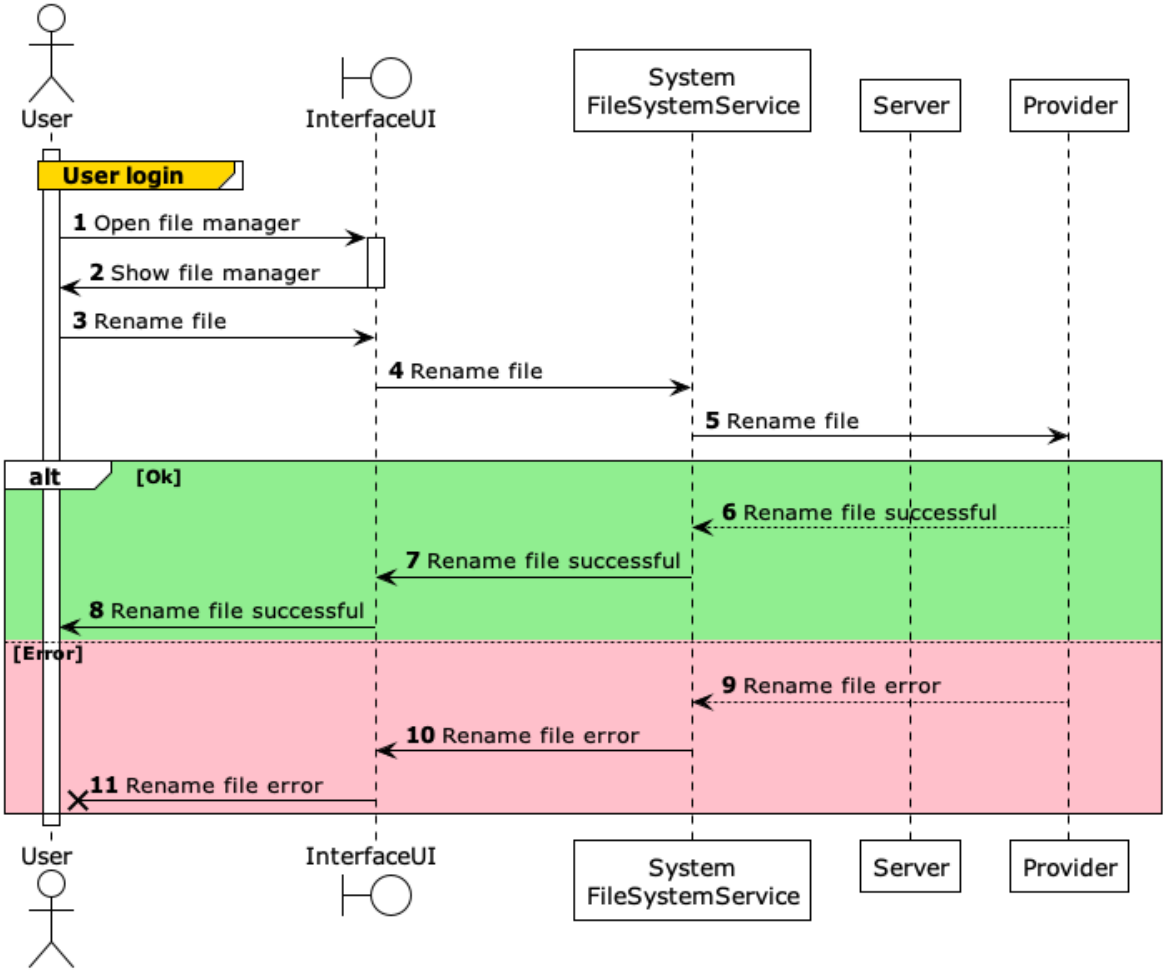


Figura 4.28: Diagrama de secuencia - Renombrar archivo

El servidor responde con dos posibles estados, en caso positivo la interfaz se actualiza mostrando el nuevo nombre del archivo, en caso negativo se notifica el error que se ha producido.

4.4.7. Editar archivo

El gestor de archivos permite seleccionar un archivo y abrirlo y editar el contenido.

El editor permite escribir y guardar un documento, también debe mostrar los metadatos del archivo.

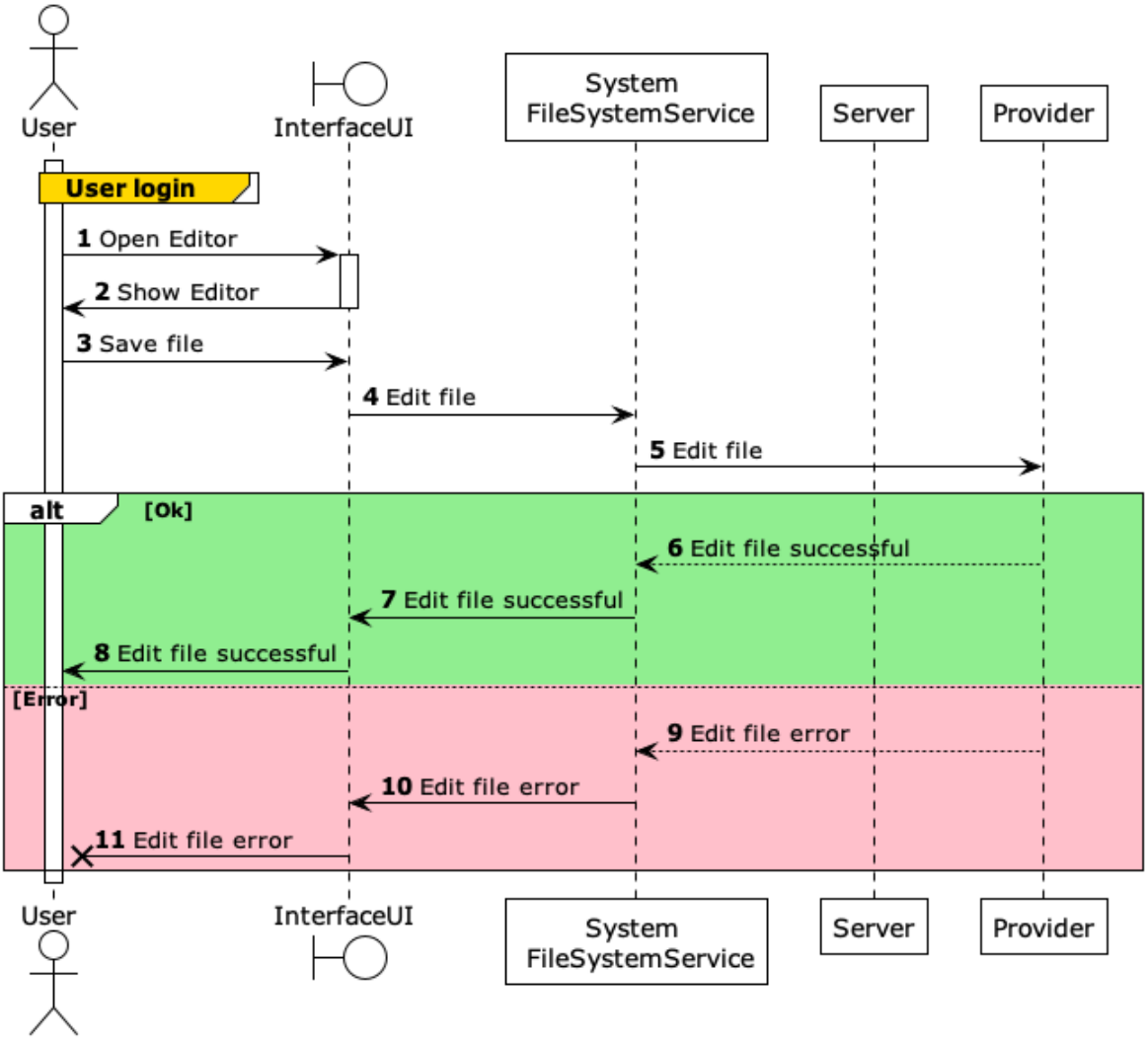


Figura 4.29: Diagrama de secuencia - Editar archivo

El archivo se guarda automáticamente si la opción de guardado está activa en la configuración.

Este guardado automático se hace tanto en local como en la nube, una vez que la base de datos NoSQL procesa el guardado, será la encargada de notificar al sistema si se ha procesado correctamente con dos estados.

En caso positivo el sistema lanza una notificación informando al usuario, y en caso negativo

4.4.8. Autocompletar código

El editor muestra al escribir las palabras clave las distintas opciones de las instrucciones a auto completar.

Si el usuario desea ver todas las posibilidades de autocompletado, podemos ver la documentación del DLX en la vista Documentación, así como una breve descripción de la instrucción.

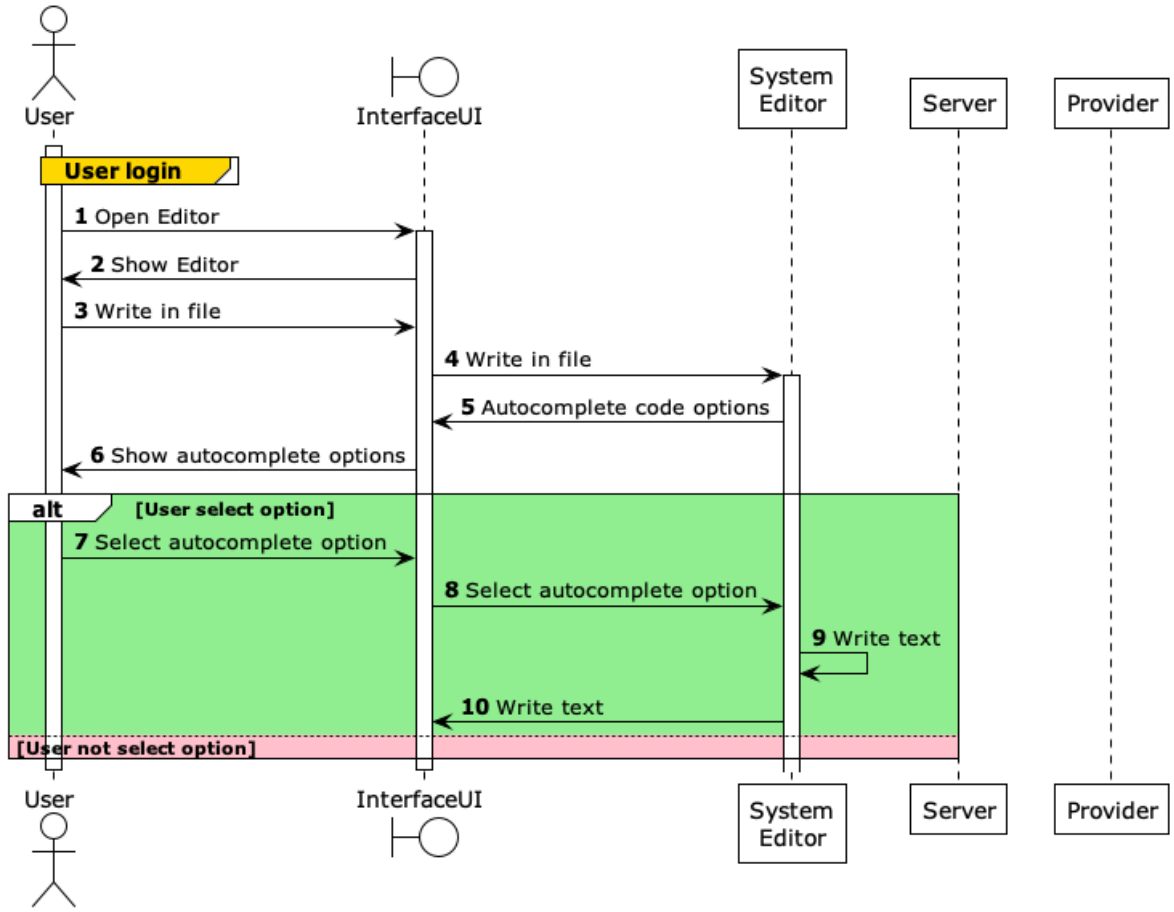


Figura 4.30: Diagrama de secuencia - Autocompletar código

4.4.9. Mostrar documentación

En el editor, al pasar el ratón por encima de una palabra clave, se mostrará un breve resumen de lo que realiza esa instrucción.

Esta sección se corresponde con la información de la instrucción del código ensamblador.

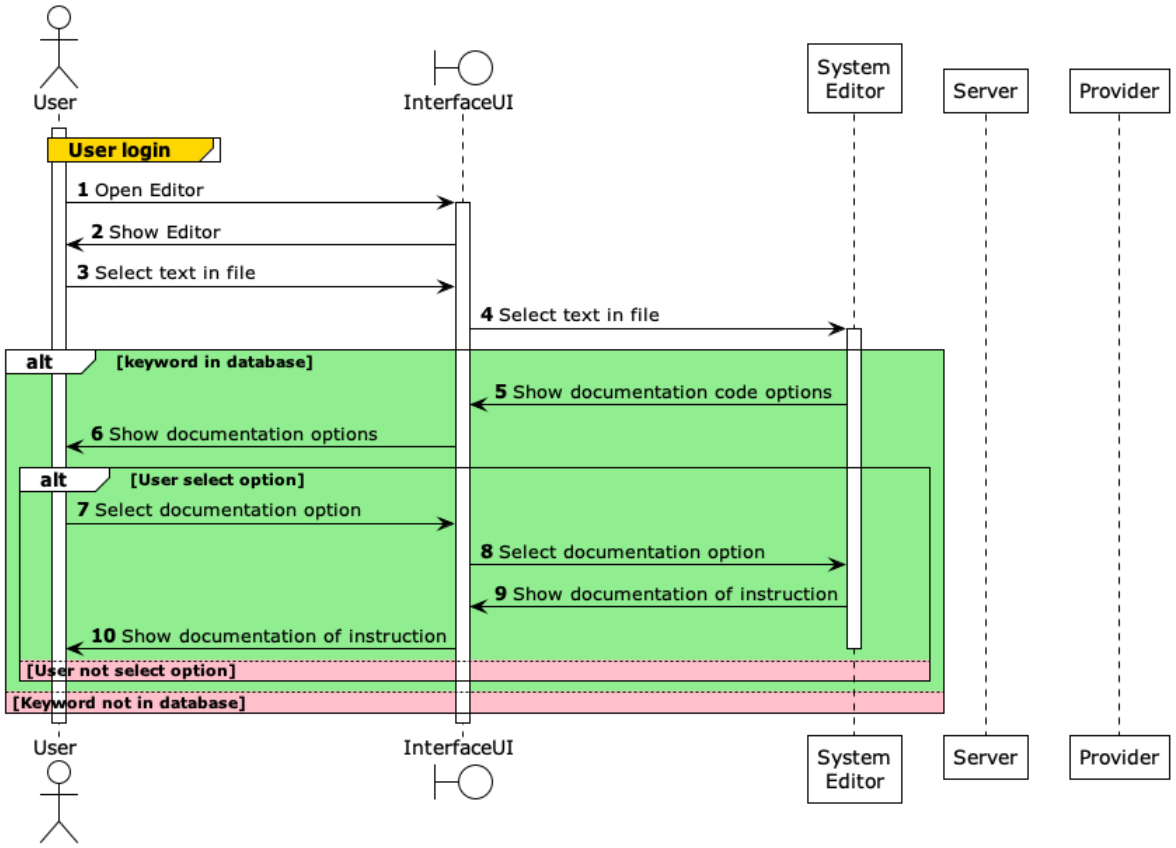


Figura 4.31: Diagrama de secuencia - Mostrar documentación

4.4.10. Mostrar errores en el código

El editor muestra los errores que tiene el código, una vez se ha analizado el código en el servidor y este ha respondido.

Podemos diferenciar dos tipos de errores, los que se hacen de forma local y los que requieren un análisis del servidor, ambos se muestran en el editor de la misma forma.

Estos errores o advertencias aparecerán subrayadas en rojo o en naranja con una descripción.

Es por ello que se tratan de dos tipos de errores, errores al escribir una instrucción en un formato no válido o errores de compilación.

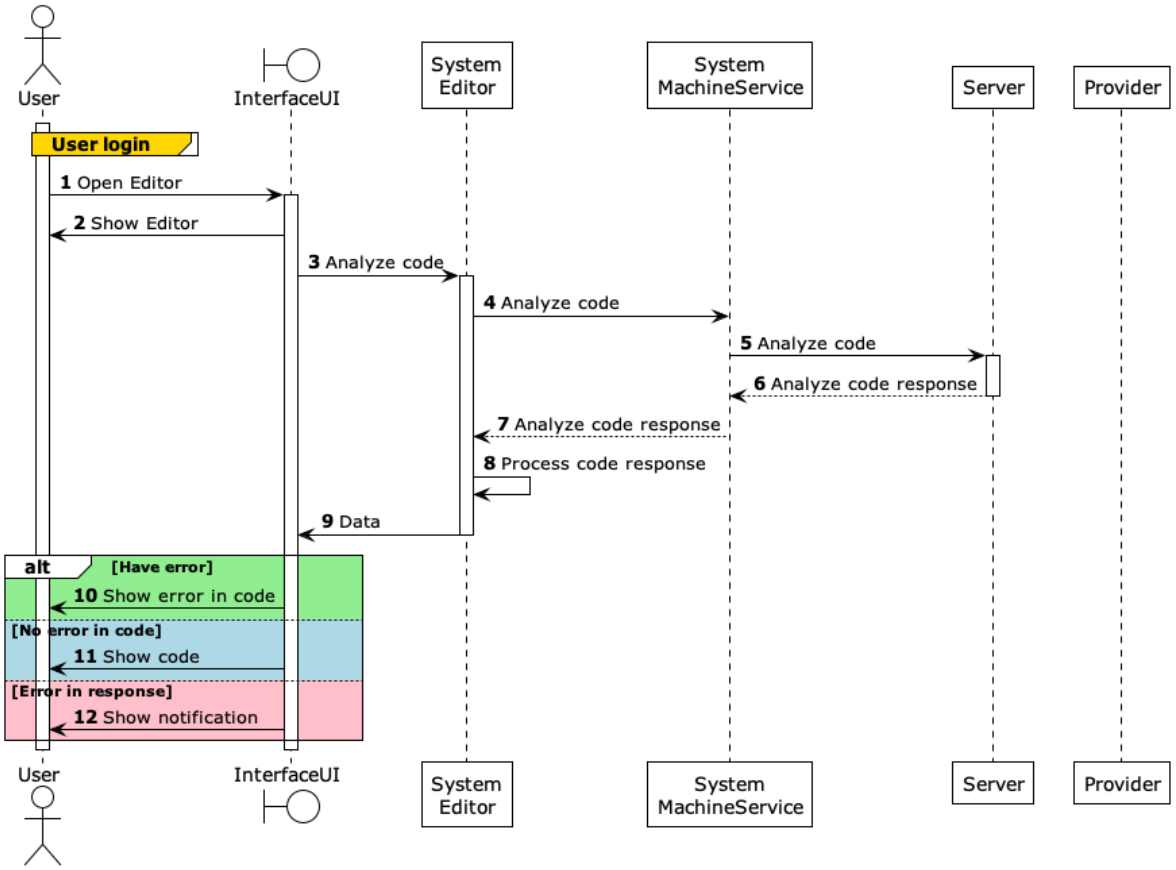


Figura 4.32: Diagrama de secuencia - Mostrar errores en el código

Para ello el usuario debe pedir una solicitud de análisis de código, esto se realiza al sincronizar el código en la aplicación y en el servidor.

Se edita el código desde el editor y se sincroniza el código, esto envía una solicitud nueva al servidor.

4.4.11. Modificar registros

La vista de los registros muestra las cuatro opciones para modificar un registro, seleccionar el registro de control, de tipo entero, flotantes de simple precisión o los flotantes de doble precisión.

Una vez seleccionado el tipo de registro se actualiza la lista de los posibles registros a modificar y el campo del valor del registro a modificar, se selecciona el registro a modificar y modificamos el valor del registro.

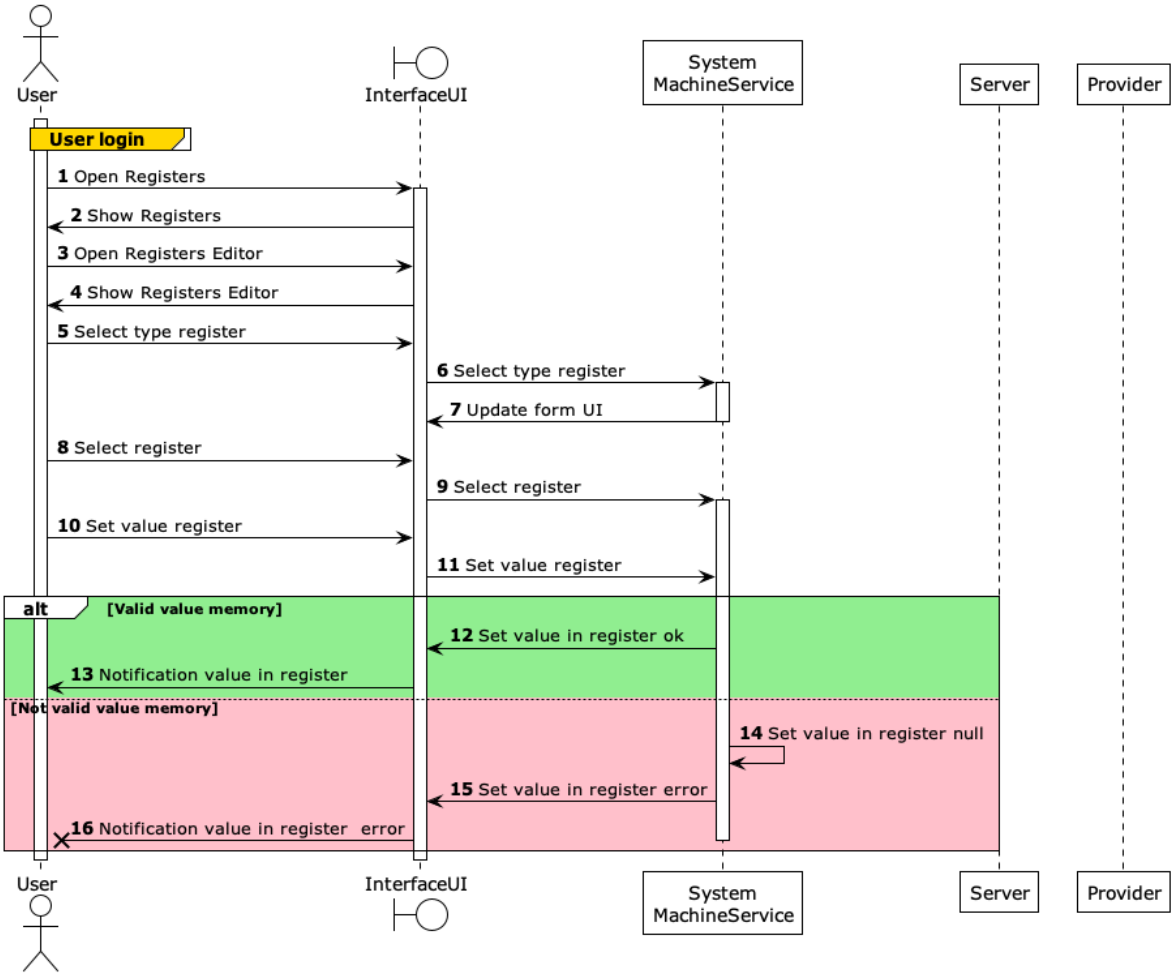


Figura 4.33: Diagrama de secuencia - Modificar registros

Una vez editado el registro, el sistema envía una petición al servidor, el servidor responderá afirmativamente confirmando el cambio en el estado de la simulación o negativamente notificando el error.

4.4.12. Modificar memoria

En la vista de la memoria, el usuario puede editar la memoria de la máquina en la simulación, para ello debe indicar el tipo de dato, la dirección de memoria y el valor a actualizar, en la misma vista se muestran las distintas representaciones de los datos.

Una vez que se han definido el tipo de dato, la dirección de memoria y el valor, el sistema envía una petición de actualización de la memoria al servidor.

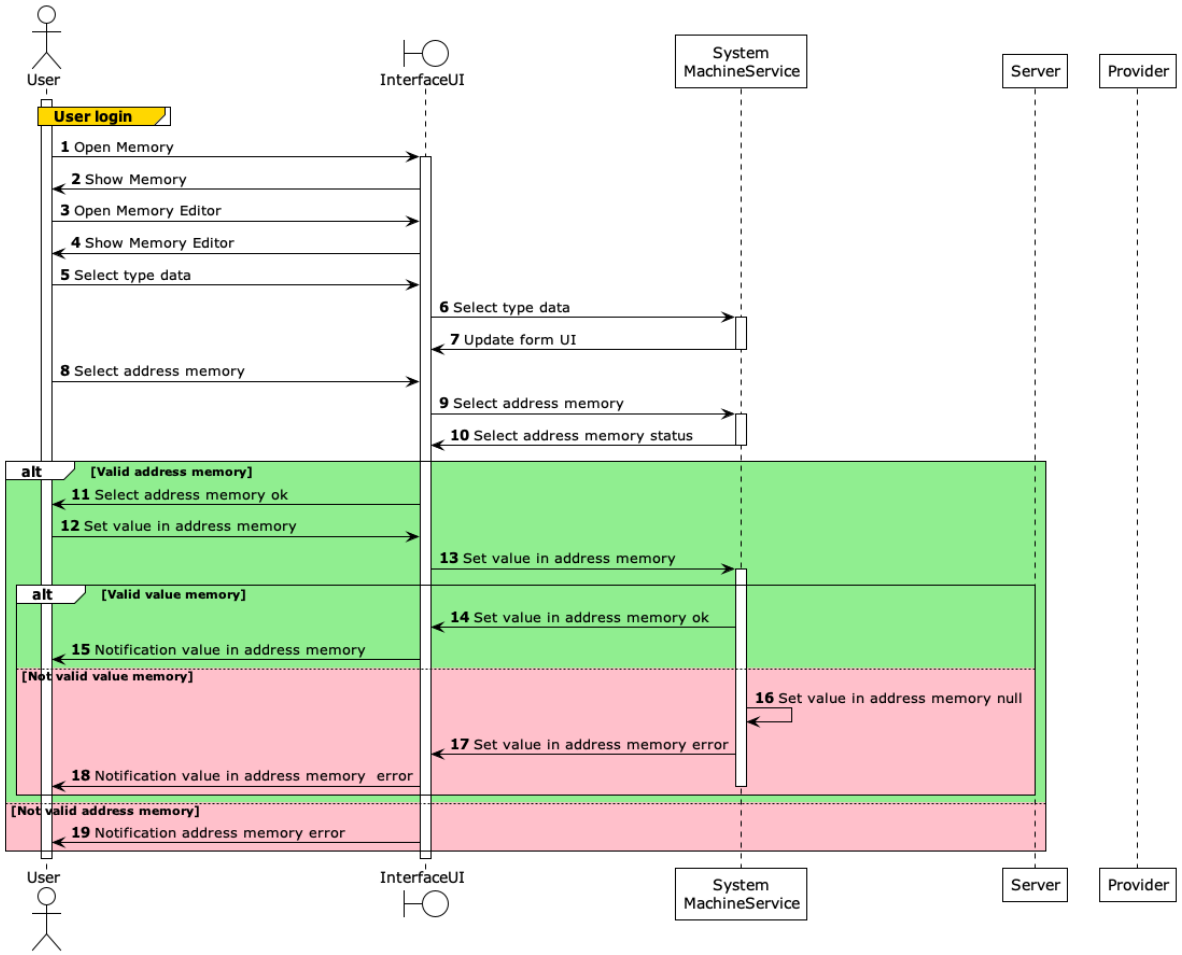


Figura 4.34: Diagrama de secuencia - Modificar memoria

El servidor responderá con una confirmación de la actualización o con un error mostrando un mensaje informativo.

4.4.13. Mostrar el cauce de ejecución

Una vez creada y cargada una simulación, el usuario puede ver el estado de la máquina en el cauce de ejecución (*pipeline*), analizando cómo se trasladan las instrucciones por las distintas etapas (`IF`, `ID`, `EX`, `MEM`, `WB`) en cada etapa mostrará la dirección en memoria y la instrucción.

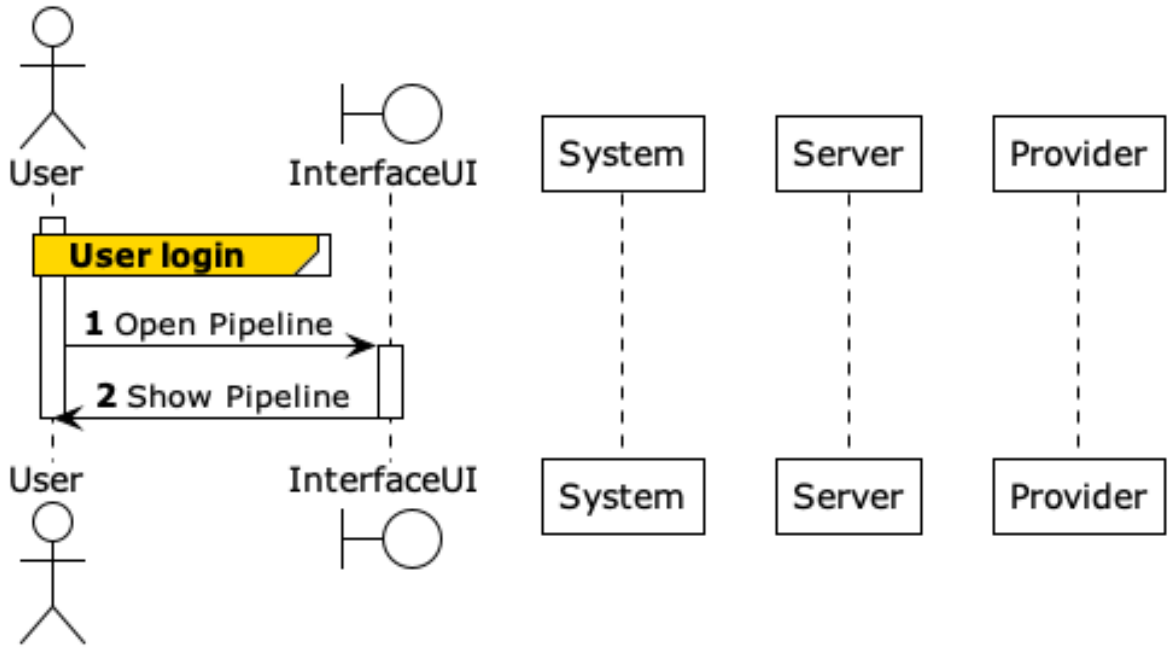


Figura 4.35: Diagrama de secuencia - Mostrar el cauce de ejecución

4.4.14. Mostrar diagrama de ciclos

Al igual que en el cauce de ejecución (*pipeline*) se ve el historial de la simulación, en el diagrama de ciclos del reloj también se muestra el estado de la simulación en los distintos pasos e iteraciones de la simulación.

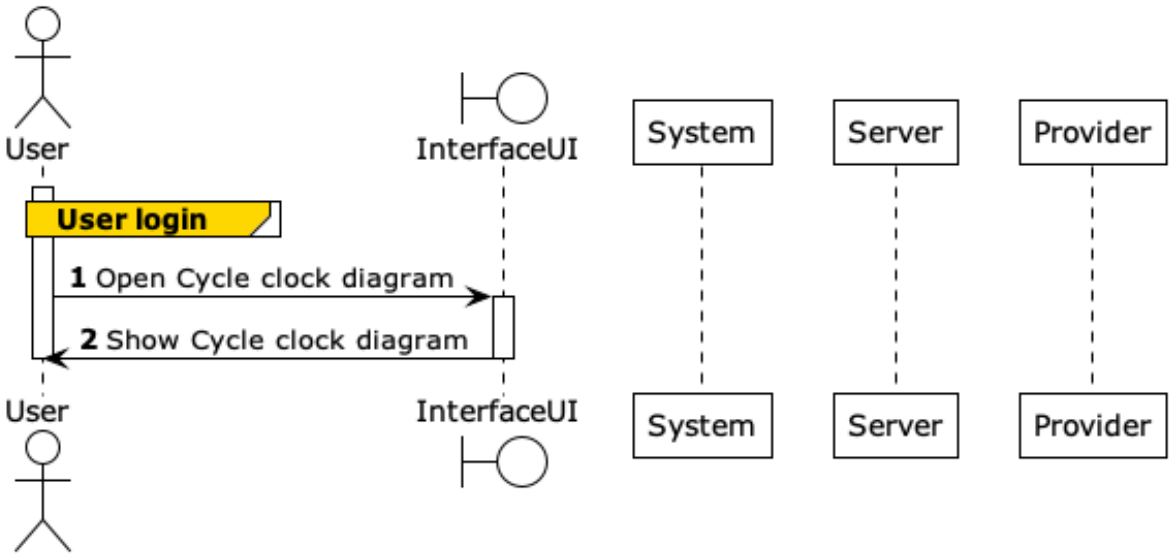


Figura 4.36: Diagrama de secuencia - Mostrar diagrama de ciclos

En esta sección no se indican las posibilidades de moverse dentro del diagrama de ciclos de reloj, en las filas se indican las instrucciones y en las columnas el paso, por el que va la simulación, también se indican en distintos colores las flechas, indicando los adelantamientos.

4.4.15. Mostrar instrucciones en memoria

El usuario puede acceder a la sección de código (*code*) sin que exista una simulación en curso.

Una vez creada y cargada una simulación, el sistema puede representar el código que se almacena en la máquina, junto a las distintas etiquetas, la instrucción y el estado en cada paso de la simulación.

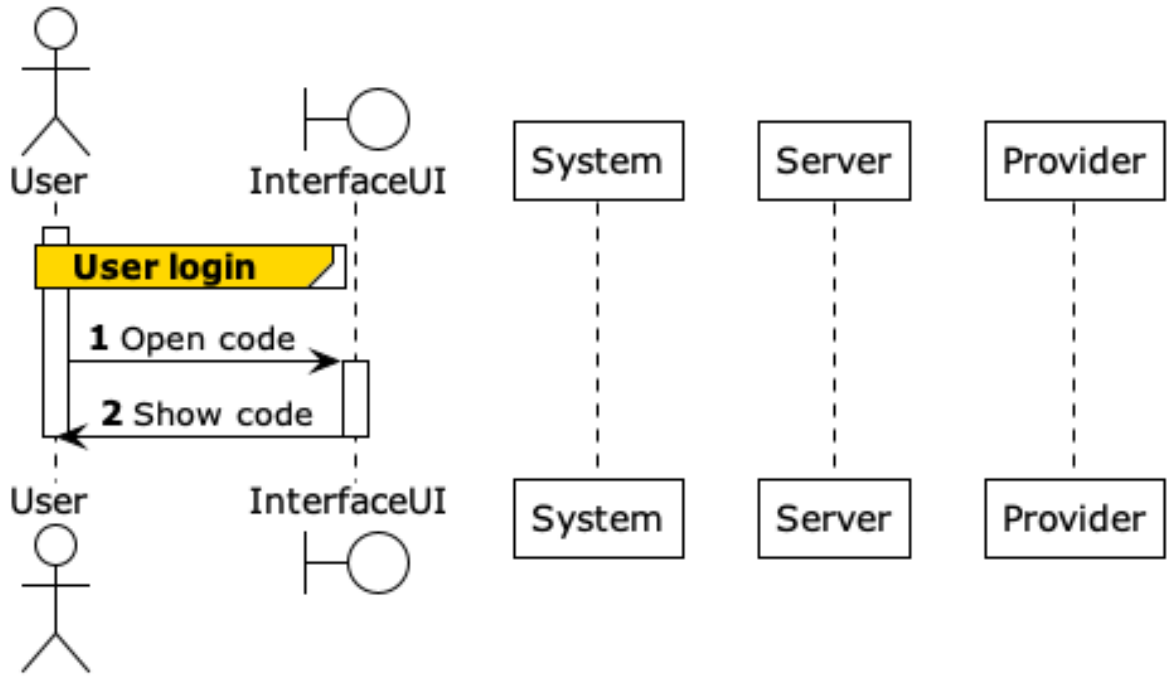


Figura 4.37: Diagrama de secuencia - Mostrar instrucciones en memoria

4.4.16. Mostrar memoria

En la vista de la memoria, el usuario puede seleccionar uno de los distintos modos para que el sistema presente los datos en el formato correspondiente, mediante

los botones *bytes*, *halfwords*, *words*, flotantes de simple precisión (32 bits) o doble precisión (64 bits) en formato IEEE754.

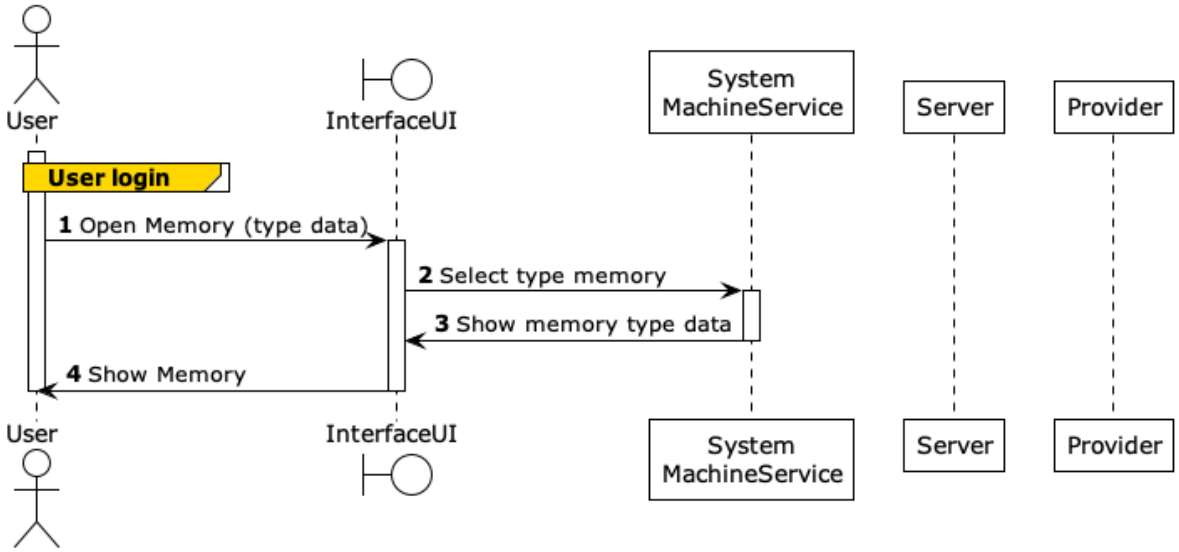


Figura 4.38: Diagrama de secuencia - Mostrar memoria

4.4.17. Mostrar registros

Al igual que en la vista de la memoria se visualizan los registros en los distintos tipos de datos y sus representaciones, en esta sección solo se puede ver los datos de tipo entero o flotantes de simple precisión y doble precisión en representación IEEE754.

Para ello el usuario pide a la interfaz que muestre los registros, la interfaz pide al sistema que muestre los registros y esta le da a la interfaz los datos, por último, la interfaz transforma los datos para que el usuario los visualice en el formato seleccionado.

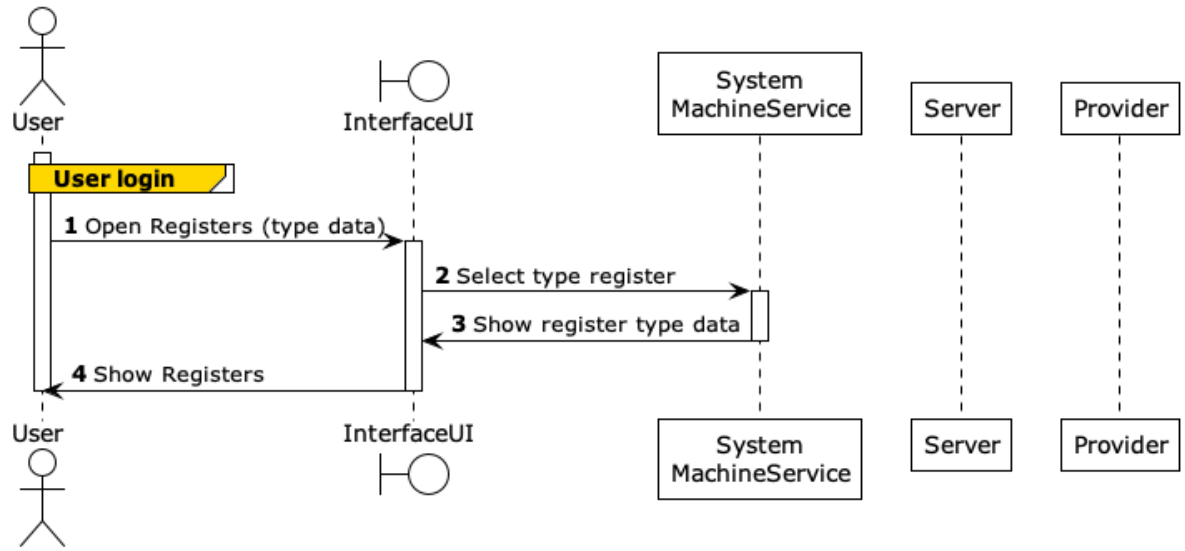


Figura 4.39: Diagrama de secuencia - Mostrar registros

4.4.18. Editar la configuración del IDE

En esta sección se agrupan los distintos tipos de configuraciones tanto del editor, las vistas, la máquina o la simulación.

El usuario debe definir las unidades de punto flotante, el tiempo que tarda en procesar una instrucción en cada unidad, el tamaño de la memoria de la máquina, el tiempo de ejecución por pasos, la configuración de la vista múltiple y el orden en el que se muestra.

Por último, el usuario selecciona la opción de guardar o reiniciar la configuración.

Al finalizar la tarea se notifica si sucede algún error o si todo se ha procesado correctamente.

No se han añadido todas las configuraciones posibles al diagrama, ya que no es necesario al tratarse de objetivos secundarios.

Por defecto se ha dejado una configuración estándar predefinida.

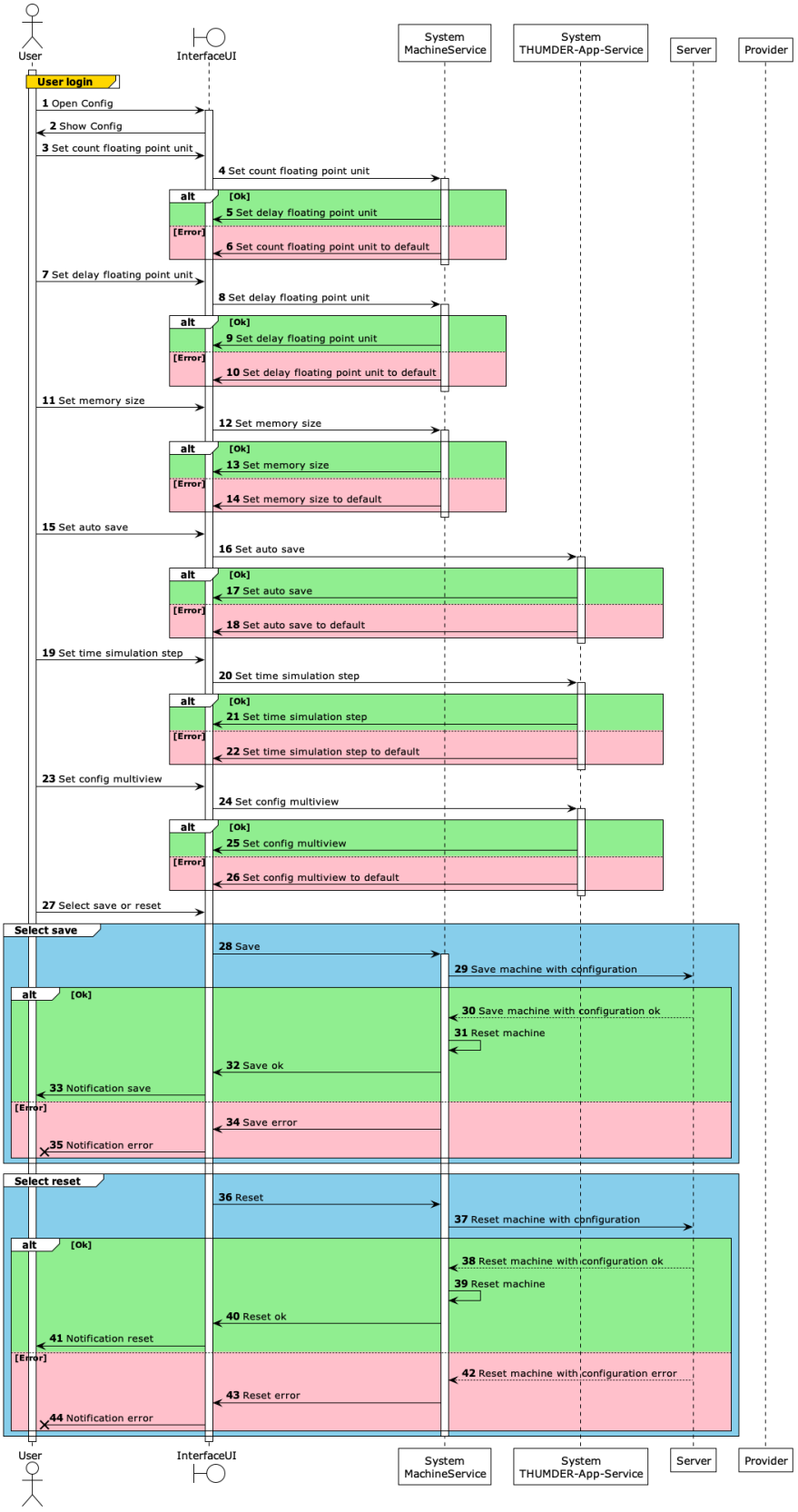


Figura 4.40: Diagrama de secuencia - Editar la configuración del IDE

4.4.19. Calculadora

El usuario puede usar la calculadora como una herramienta de utilidades para representar los datos en los distintos formatos. La herramienta permite representar datos de tipo binario, hexadecimal o decimal en los distintos formatos (*byte*, *halfword*, *word*, IEEE754 de 32 o 64 bits).

El usuario debe abrir la calculadora y escribir en los campos de la interfaz el valor que desea transformar, la interfaz envía el dato al sistema y este devuelve el dato transformado en los distintos formatos que se mostrarán en la interfaz.

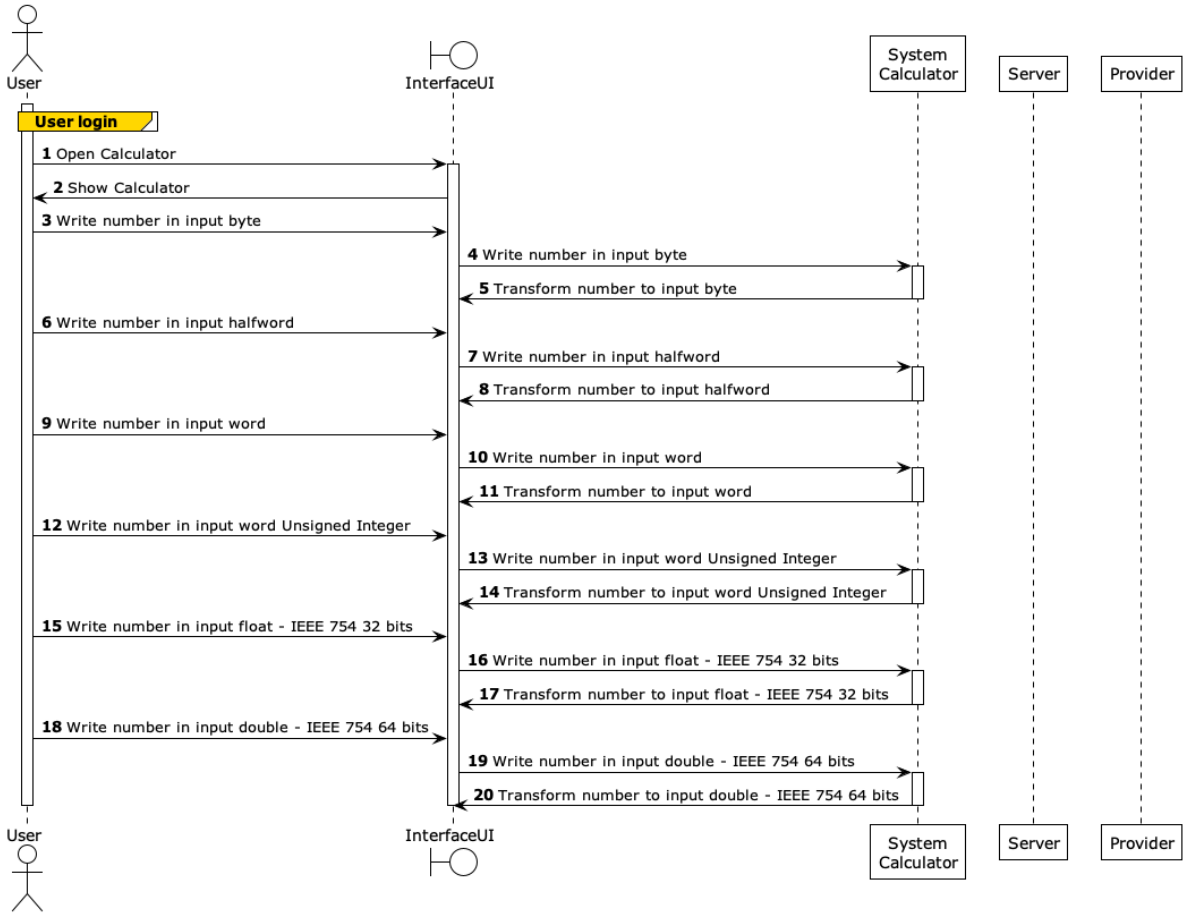


Figura 4.41: Diagrama de secuencia - Calculadora

4.4.20. Cargar simulación

Una vez realizada la conexión con el servidor, la aplicación envía el texto en el editor (el código) de nuestro programa para que el servidor responda con el código binario a representar en la simulación.

Para cargar una simulación el usuario debe tener abierto un archivo en el editor, a continuación, el usuario indica que desea sincronizar el archivo con el servidor, el servidor responderá con las instrucciones máquina y código binario necesario de forma que el sistema cargue las directivas y el código en la memoria de la máquina.

El sistema también ejecuta una pequeña simulación con ciertos pasos ya predefinidos que inicializa la simulación y la deja preparada.

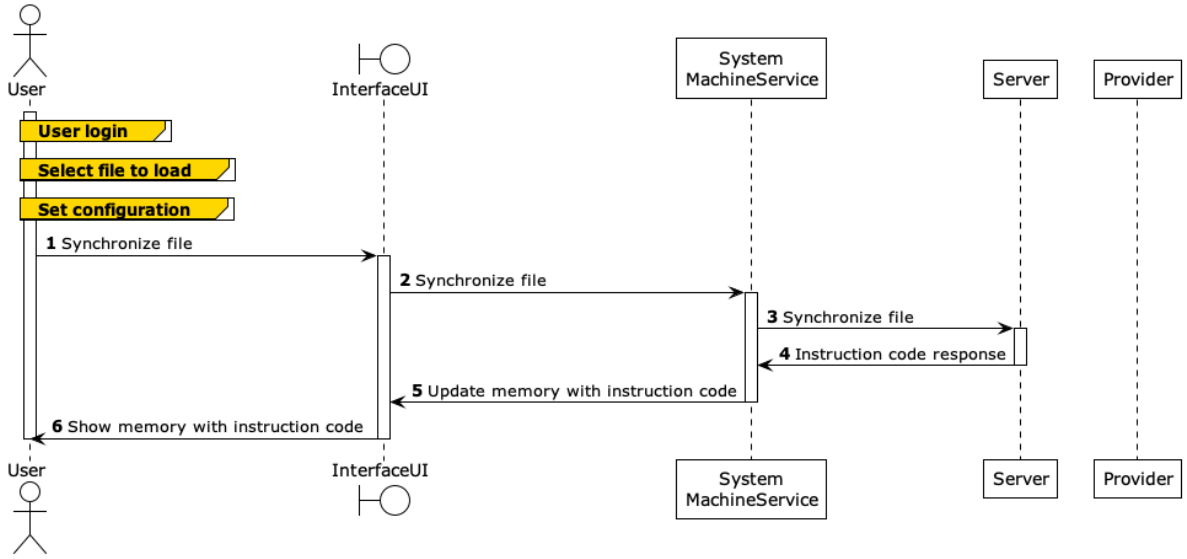


Figura 4.42: Diagrama de secuencia - Cargar simulación

4.4.21. Iterar en la simulación

La simulación se realiza por pasos, por lo que se puede pasar de un estado al siguiente, en cada paso se actualiza la memoria, registros, diagrama de ciclos, cauce de ejecución (*pipeline*) y las estadísticas.

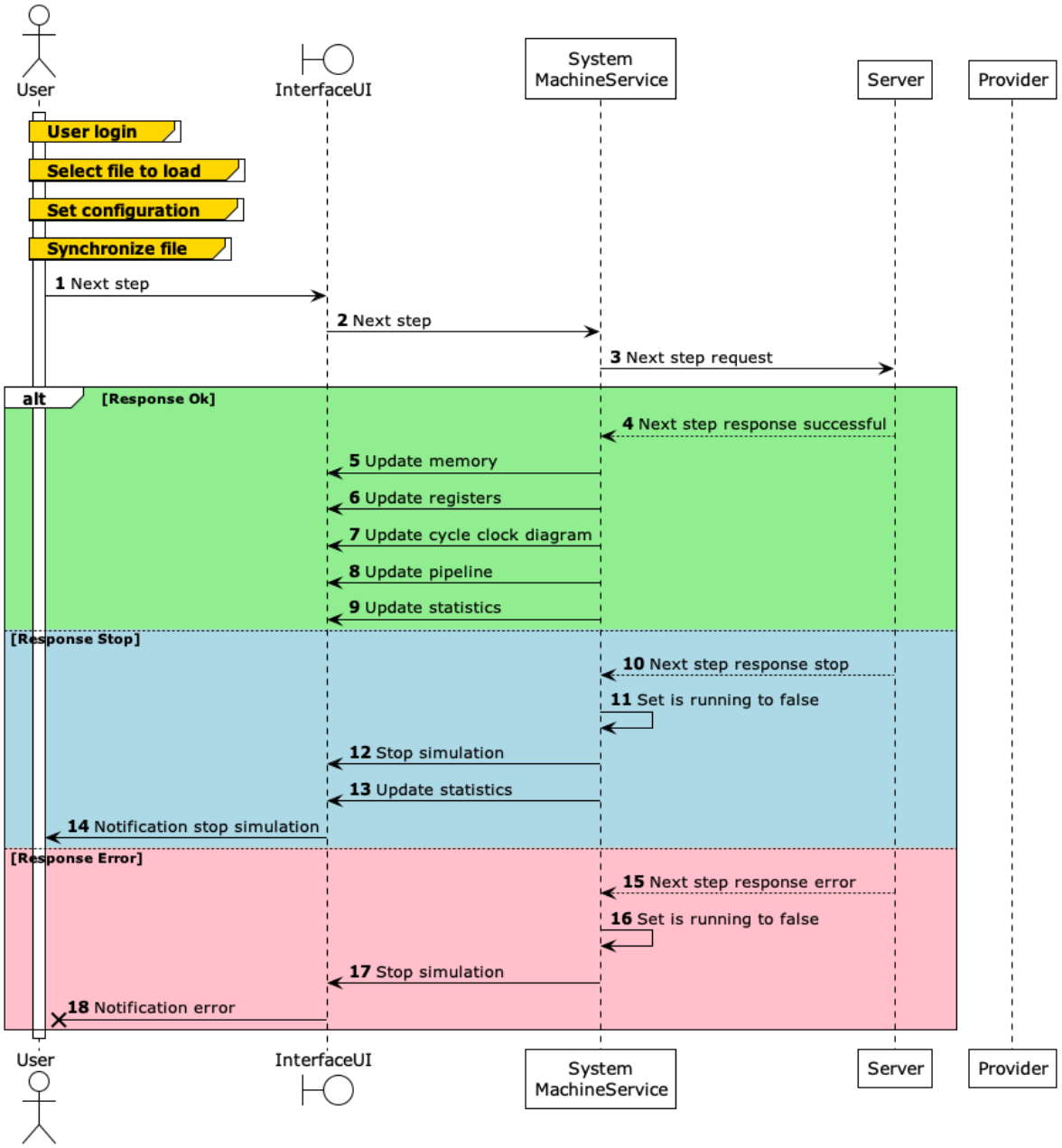


Figura 4.43: Diagrama de secuencia - Iterar en la simulación

Esto permite visualizar cómo evoluciona el estado de la máquina en cada paso, además en cada paso se da un contexto de cómo transcurre cada iteración, indicando si se puede seguir avanzando en la simulación, si ha habido algún punto de parada (*breakpoints*) o si se ha detenido la simulación, ya sea por alguna interrupción del sistema o por finalización la simulación.

4.4.22. Ejecutar simulación de forma continua

Para ejecutar la simulación, el sistema debe tener en cuenta los pasos en una simulación, así como las variantes que pueden suceder en esta simulación con los puntos de parada (*breakpoints*). Todas las simulaciones se realizan por pasos, hay dos modalidades, la primera ejecuta todo el código mediante un *runner* hasta que llegue al final de la simulación o hasta que encuentre uno de los puntos de parada (*breakpoints*), el otro método es ejecutar la simulación paso por paso.

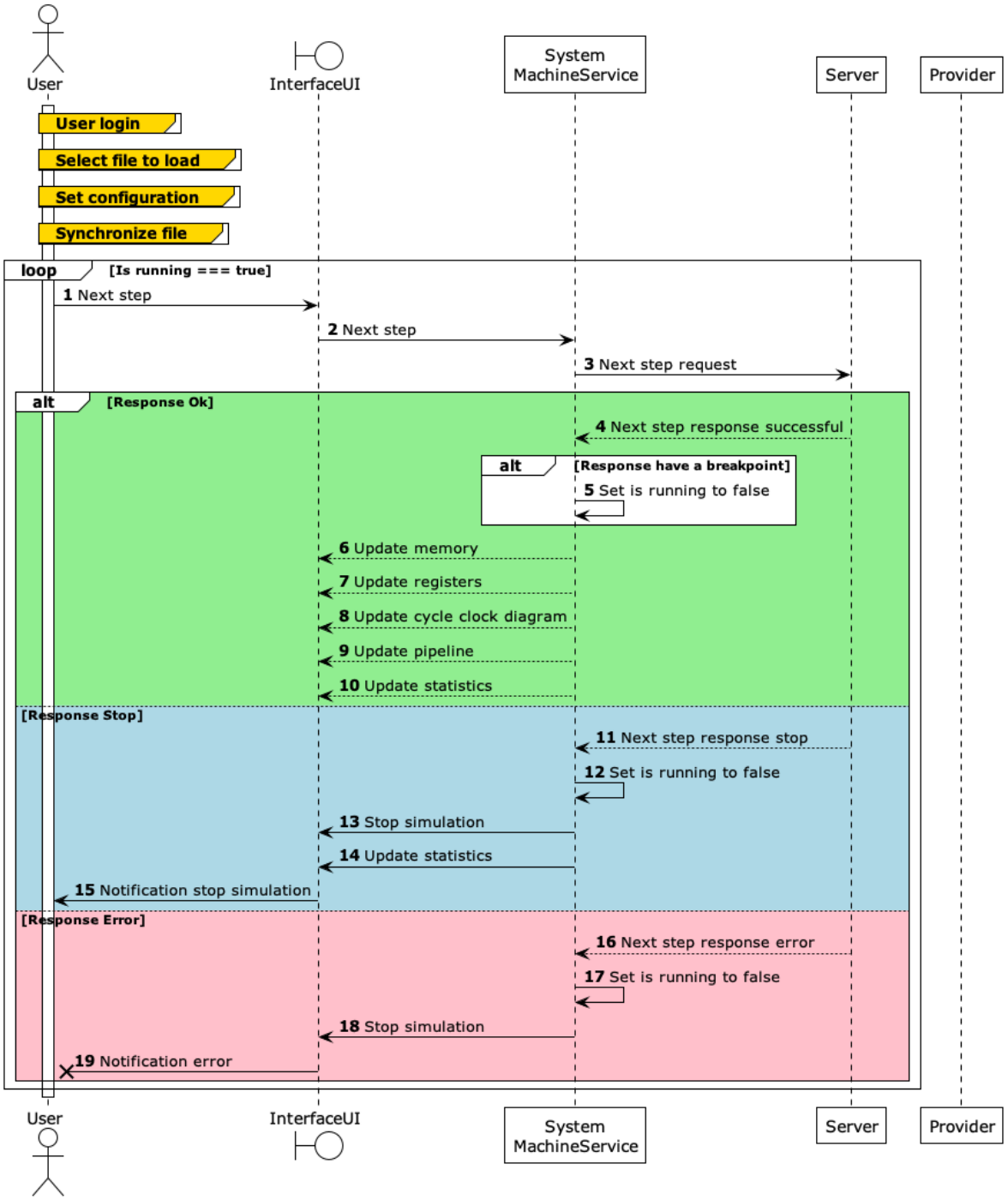


Figura 4.44: Diagrama de secuencia - Iterar en la simulación

Este diagrama usa el diagrama de simulación por pasos, el sistema usa el contexto que proporciona la respuesta del servidor para pasar al siguiente estado de la simulación.

4.5. Prototipos y Wireframes

Se han desarrollado varios *wireframes*, estos *wireframe* representan una vista genérica que pueda incluir los distintos componentes, es una vista maestra de ejemplo de una sección.

Esto se ha hecho para simplificar los distintos diseños, pues muchas de estas secciones son tablas o componentes muy similares.

Esta distribución es la planteada para el proyecto, es un diseño híbrido entre una aplicación web moderna y el diseño que usó la aplicación `WinDLX`, algunos cambios están sujetos a cambios estéticos, de márgenes, pero cuenta con la distribución (*layout*) planeada para el proyecto.

Antes de este *wireframe* se realizaron varios bosquejos (*Mockup*).

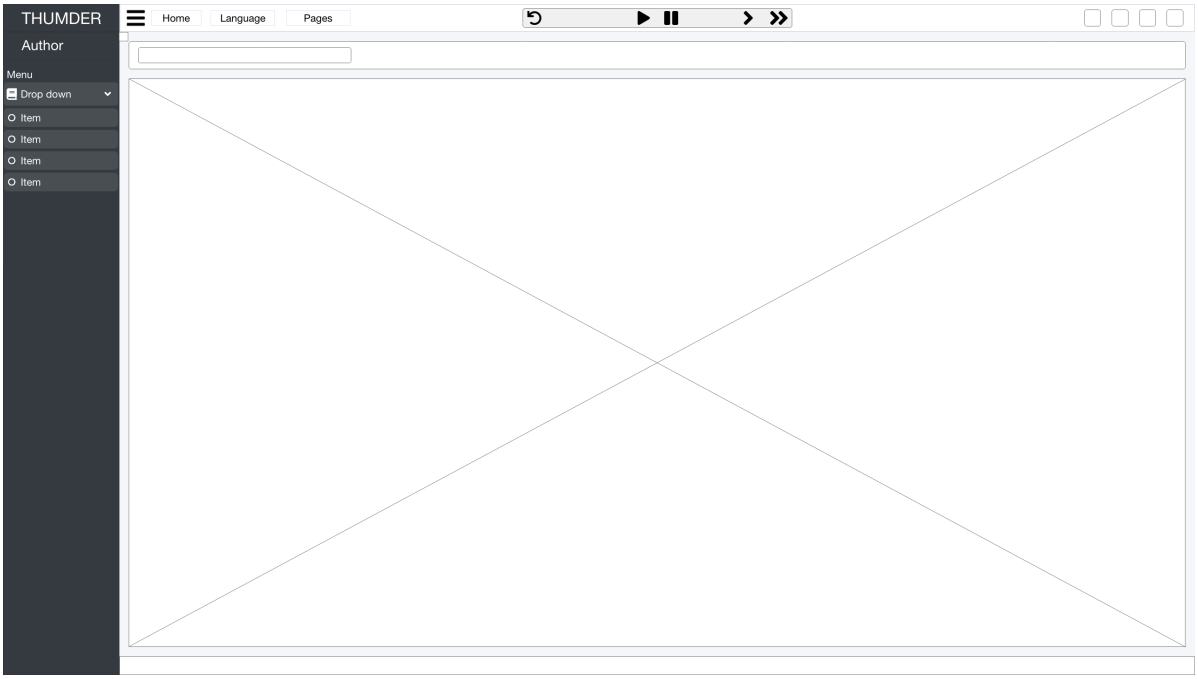


Figura 4.45: Wireframe layout

Wireframe de ejemplo de la sección del cauce de ejecución (*Pipeline*) de la máquina.

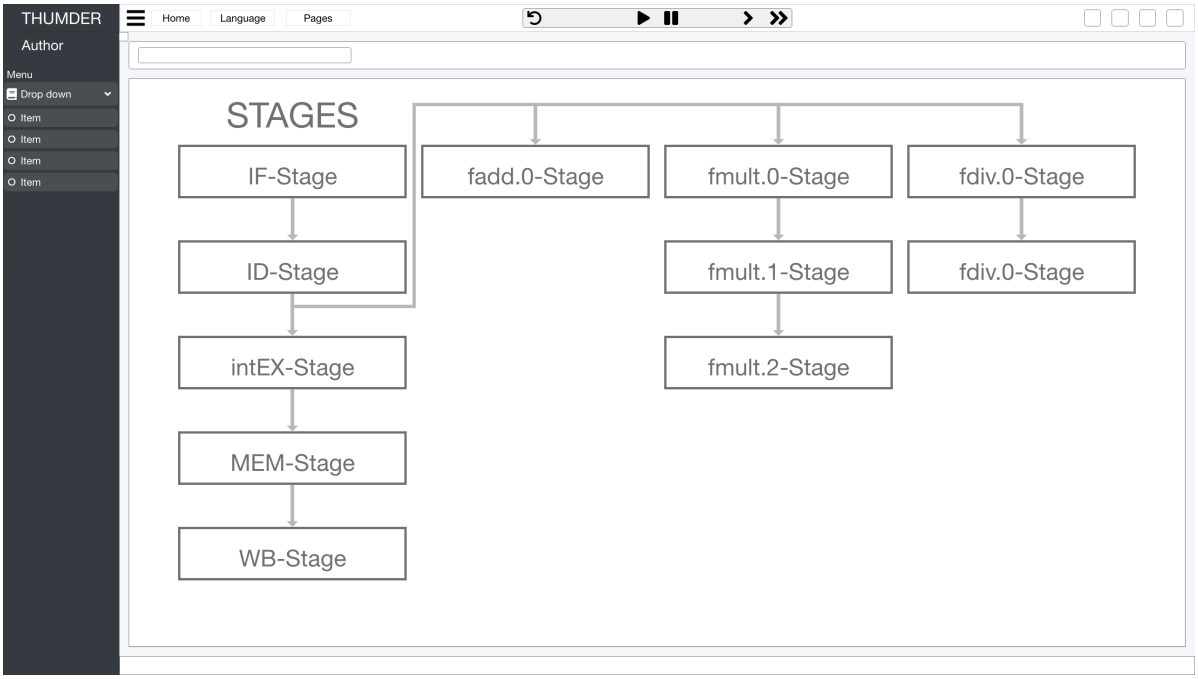


Figura 4.46: Wireframe - cauce de ejecución (*Pipeline*)

Wireframe de ejemplo de la sección del diagrama de ciclos de reloj (*Cycle Clock Diagram*) de la máquina.

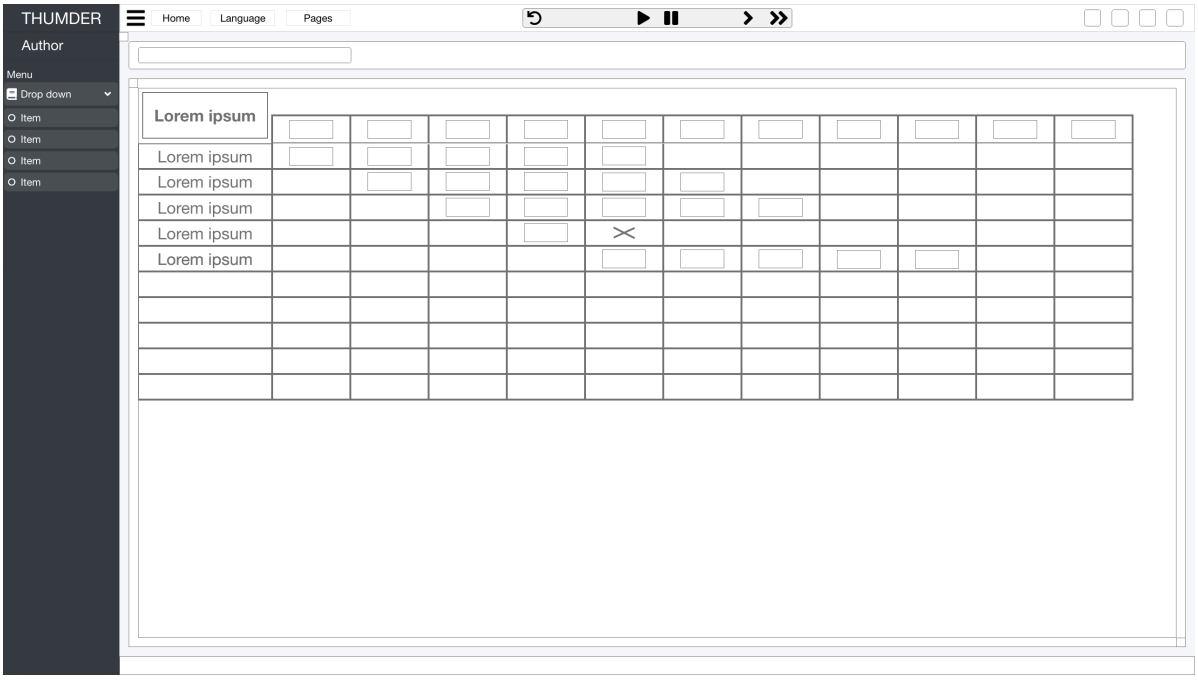


Figura 4.47: Wireframe - diagrama de ciclos de reloj (*Cycle Clock Diagram*)

Capítulo 5

Planificación del desarrollo

5.1. Herramientas de desarrollo

Primero para el desarrollo de nuestro proyecto debemos elegir qué tecnologías vamos a usar, estas tecnologías son muchas y muy variadas, no todas tienen un propósito específico, algunas de estas abarcan varios campos y usos, es por eso que esta sección se describen brevemente las herramientas software previstas a usar para nuestro proyecto, a continuación, se indica la elección y qué ventajas nos proporciona cierta tecnología.

Algunas de estas herramientas pueden ser .NET, Node, NPM, Kotlin, TypeScript, Babel, Esbuild, Angular, React, Electron, etc.

Podemos agrupar las herramientas en las siguientes grandes categorías.

1. Lenguajes de programación 5.1.1
2. *Frameworks* - *Frontend* 5.1.2
3. *Frameworks* - *Backend* 5.1.3
4. Aplicaciones de escritorio 5.1.4

Al tratar con estas tecnologías un equipo de desarrollo se decanta por el uso de los *frameworks*, estos son esquemas de organización, librerías, proporcionan una base de la que empezar un proyecto.

Hay muchos *frameworks* de *frontend* o *backend* escritos en decenas de lenguajes (PHP, Kotlin, Java, JavaScript, C-Sharp, CPP, etc.)

Estos *frameworks*, suelen proponer una estructura para organizar el código.

5.1.1. Lenguajes de programación

Podemos elegir un lenguaje de programación frente a otro en función de las ventajas que ofrezca, el problema a resolver, las características del problema, ventajas y desventajas del rendimiento y el nivel de abstracción del lenguaje, el enfoque del lenguaje, orientado a objetos o de programación funcional y muchas más características.

Estos lenguajes pueden estar orientados al entorno frontend o backend, aunque en nuestro caso, las derivaciones de estos lenguajes terminan compilando en el lenguaje de JavaScript.

5.1.1.1. JavaScript

JavaScript es un lenguaje de programación o de secuencias de comandos que nos permite implementar funciones complejas en la web, este lenguaje se ha desarrollado mucho a lo largo de los años, se puede clasificar en función de sus versiones (ECMAScript [17]), aunque su principal desarrollo ha sido por el uso de Node.js [18], permitiendo que el lenguaje no se quedase únicamente en un lenguaje solo para páginas web, sino que se ha extendido y permitido crear entornos *backend* y múltiples herramientas de escritorio.

5.1.1.2. TypeScript

TypeScript es un superconjunto de herramientas que permiten escalar una aplicación web, permite aumentar la legibilidad del código y ofrece muchas posibilidades de gestión de tipos de datos, mejoras para organizar y compilar el código, las posibles alternativas de esta herramienta son CoffeeScript, Babel, Kotlin, etc.

5.1.1.3. Kotlin

Kotlin ha evolucionado mucho desde su creación en 2011, este lenguaje se ha adaptado a múltiples plataformas y entornos (Kotlin/Native, Kotlin/JS, Kotlin/JVM), creado para sustituir a Java en Android, se ha extendido su uso y ahora permite manipular el DOM de una página web y compilar en JavaScript, esto nos proporciona múltiples opciones para desarrollar un servidor *backend* o un cliente reactivo en el *frontend*.

5.1.2. Librerías y *frameworks - frontend*

Se considera frontend a la parte que tiene acceso el cliente de forma que interactúa directamente con ella mediante interfaces.

Muchos de estos *frameworks* de páginas web son de tipo SPA, es decir, proporcionan una abstracción de la programación del código JavaScript con respecto al DOM, frente a esto existen otros *frameworks frontend*, son los llamados *frameworks* de interfaz, CSS o de estilos, estos solo se centran en el diseño web y en la programación del código con respecto al DOM.

5.1.2.1. React

React es una librería de código abierto creada por Facebook, esto escrita en JavaScript pensada para crear interfaces de usuario, esta librería se utiliza para crear una aplicación en una web sin recargas, está especializada en la creación de componentes aislados.

Está pensada para la reutilización de sus componentes en una sola dirección, es decir, un componente solo puede heredar el comportamiento de su padre, es lo que denominan enlace de datos unidireccional.

React es muy ligero y rápido, cuenta con una curva de aprendizaje sencilla, ya que la librería cuenta con pocas funcionalidades, aunque se puede extender su uso con otras herramientas.

Por último, React no cuenta con CLI propio, por lo que ciertas tareas deberemos hacerlas nosotros mismos o usar algunas herramientas creadas por la comunidad.

5.1.2.2. Angular

Angular es un framework de código abierto creado por Google, está basado en código TypeScript y permite crear aplicaciones web tipo SPA, también se basa en la creación de componentes reutilizables, este proyecto es una derivación de lo que fue AngularJS.

Angular tiene un sistema de datos bidireccional (Código - Vista), es decir, si se actualiza el valor en la vista se actualiza el valor de la variable en el código y viceversa, así son datos reactivos.

Angular es muy ligero una vez compilado, es rápido en ejecución y tiene una curva de aprendizaje media, ya que tiene muchas funcionalidades, todas estas utilidades hacen que su ecosistema sea grande y requiera tiempo aprenderlo.

Angular al ser un *framework* propone ciertas formas de crear el código y los componentes.

Angular se gestiona mediante su propio CLI que proporciona tareas automáticas, compilación, publicaciones, estadísticas, etc.

5.1.2.3. Vue

Vue es un *framework* de código abierto creado por Evan You, aunque es mantenido por empresas tecnológicas tales como Alibaba o Xiaomi, está escrito en JavaScript, permite crear aplicaciones web tipo SPA mediante componentes.

Vue cuenta con un sistema de componentes en un solo archivo, uso reactivo de datos de una forma diferente al resto de tecnologías, ya que de lo contrario puede fallar en su actualización, al ser flexible es fácil de usar y de añadir otros componentes, pero ha de gestionarse bien o el código puede ser confuso.

Además de los componentes en archivos, también podemos crear componentes globales que se reutilizan mediante referencias.

Vue también cuenta con un CLI que proporciona tareas automáticas para el control del proyecto.

5.1.2.4. Otros

Hay decenas y decenas de *frameworks* de desarrollo *frontend*, se podría decir que cada semana hay uno nuevo y popular, es por ello que solo nos vamos a limitar a nombrar los *frameworks* más usados en los últimos años y que nos den ciertas características útiles para nuestro proyecto.

- Bootstrap [19]: es uno de los *frameworks* de estilos CSS más famosos y usados, cuenta con muchos componentes, entre ellos formularios, botones, fragmentos de código y con una gran comunidad que ha dado soporte a muchos complementos, está desarrollado por la empresa Twitter.
- Semantic UI [20]: al igual que Bootstrap este *framework* proporciona una serie de componentes, entre ellos formularios, botones, títulos y formas de distribuir la aplicación.
- AdminLTE [21]: es una derivación de Bootstrap, a diferencia de Bootstrap sí presenta una serie de plantillas y tiene integradas librerías que extienden su uso.
- Foundation [22]: muy similar a Bootstrap ha sido el gran competidor de este, aunque no ha tenido la aceptación y el apoyo de muchos complementos de la comunidad de desarrolladores.

5.1.3. Librerías y *frameworks - Backend*

Se considera backend a la parte que no tiene acceso el cliente de forma directa, interactúa directamente con una interfaz (*frontend*) e indirectamente con el *backend* mediante peticiones, frente a esta situación se han creado un conjunto de herramientas para desarrollar aplicaciones y estructurar la aplicación de una forma simple y rápida.

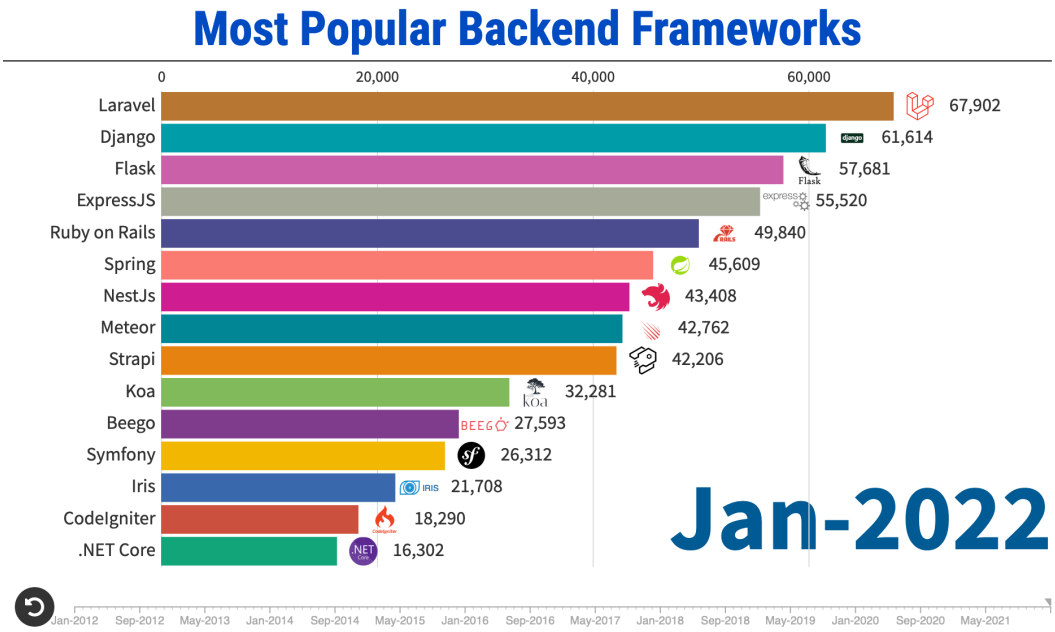


Figura 5.1: Lista de *frameworks* famosos en *backend*

La clasificación anterior 5.1 no indica que sean los mejores *frameworks*, solamente muestra una clasificación de *frameworks* famosos, la información sobre la clasificación se ha extraído de la web statisticsanddata.org [23].

5.1.3.1. Laravel

Laravel es un framework escrito sobre PHP creado por Taylor Otwell, sigue la arquitectura MVC (*Model-View-Controller*), cuenta con múltiples características muy útiles y sencillas, entre ellas el motor de edición Blade para editar el HTML, ORM para escribir consultas sobre objetos, cuenta con clases de seguridad y cifrado, cuenta con un gestor de bases de datos y control de migraciones, también cuenta con la herramienta de línea de comandos Artisan para el control del proyecto.

5.1.3.2. .NET

.NET es una plataforma de desarrollo mantenida por Microsoft creada para competir directamente contra JAVA, se desarrolló en el lenguaje de programación C-Sharp, contiene un conjunto muy grande de librerías y utilidades que permite abordar múltiples problemas.

Microsoft en sus últimos años se ha enfocado mucho en este framework para mejorar su ecosistema Azure.

Antes solo estaba orientado a Windows, pero con el desarrollo del proyecto Mono [24] ahora también se pueden ejecutar en Linux y macOS.

.NET es un framework que incluye múltiples soluciones, por lo que se ha dividido en distintas funcionalidades

1. **.NET Framework** proporciona servicios web y aplicaciones de escritorio en Windows.
2. **NET Core** es una implementación multiplataforma orientada a la web, servidores y aplicaciones de línea de comandos para sistemas operativos Windows, Linux y macOS.
3. **Xamarin** / **Mono** es la implementación para ejecutar aplicaciones desarrolladas bajo .NET en los principales sistemas operativos de escritorio y sistemas operativos móviles (iOS, Android, Linux, UbuntuTouch, etc.).

5.1.3.3. Otros

Hay muchos *frameworks* de desarrollo *backend*, este tema daría para varios estudios de investigación, algunos de los *frameworks* más famosos son los siguientes.

Estos son algunos de los *frameworks* más conocidos en la industria.

- Codeigniter y Laravel escritos en PHP.
- Django y Flask escritos en Python.
- Ruby on Rails escrito en Ruby.
- Spring escrito en Java.
- Express escrito en JavaScript.
- ASP.NET escrito en C-Sharp.

5.1.4. Aplicaciones de escritorio

Hay ciertas tecnologías que nos permiten el desarrollo de aplicaciones multiplataforma sin la necesidad de un servidor, estas herramientas suelen estar organizadas de forma que agrupan el motor de navegador Chromium [25] y la aplicación, no entramos en detalle de las decenas de opciones si hablamos de aplicaciones multiplata-

formas, en nuestro caso, el TFG nos limita al desarrollo de una interfaz web, por lo que debemos agrupar una web y un posible entorno

5.1.4.1. Electron

Electron es un proyecto de código abierto creado por Cheng Zhao en 2013 [26], actualmente desarrollado por GitHub.

Pensado para la ejecución de aplicaciones gráficas en un entorno de escritorio, usa Node.js en el lado del servidor y Chromium para la interfaz.

Electron funciona para las plataformas Windows, Linux y macOS.

Electron es el motor de muchos proyectos modernos, entre ellos el editor de código de Visual Studio Code, servicios de mensajería tales como Discord, Skype, Wire, Signal, etc.

5.1.4.2. NW.js

NW.js o también llamado Node-Webkit es un proyecto de código abierto creado por la empresa Intel, se distribuye actualmente bajo una licencia MIT [27].

Es un entorno de ejecución de aplicaciones gráficas desarrolladas para la web, está basado en Node.js y el proyecto Chromium.

Permite usar los módulos (`modules`) de Node.js desde el mismo DOM, por lo que el desarrollo de una aplicación puede ser distinto de otros proyectos.

NW.js funciona para las plataformas Windows, Linux y macOS.

5.1.4.3. Otros

Hay más alternativas que las mencionadas anteriormente, algunas de las siguientes opciones son bastante usadas actualmente, aunque algunas están obsoletas.

- Meteor [28].
- AppJS [28].
- Proton-native [29].

5.1.5. Librería de pruebas

Un proyecto de este tipo exige un sistema de pruebas y validación, en el desarrollo web se han creado decenas de *frameworks* y librerías que permiten probar nuestro código y por suerte estas librerías han terminado siendo muy similares unas a otras en sintaxis, por lo que en muchos casos cambiar de una *suite* de pruebas a otra suele ser sencillo, aunque la orientación de cada herramienta es distinta, algunas solo prueban nuestra lógica, mientras otras herramientas prueban la funcionalidad en distintos navegadores.

Cada conjunto de herramientas de pruebas está orientada a una función más concreta y no realizan la misma función.

Muchas de estas librerías analizan la cobertura del código [1].

5.1.5.1. Jest

Jest es una herramienta de pruebas creada por Meta (Facebook) [30], es de código abierto y desarrollada en JavaScript, permite crear pruebas para todo tipo de librerías desarrolladas en JavaScript.

Jest permite el desarrollo bajo TypeScript, aumentando mucho su legibilidad, también permite el análisis de cobertura de código creando informes de que líneas se ejecutan en las pruebas.

5.1.5.2. Jasmine y Karma

Jasmine es una *suite* de pruebas basada en el comportamiento para probar código JavaScript. No depende de ninguna otra dependencia, tampoco requiere el uso del DOM, tiene una sintaxis limpia y clara para que pueda escribir pruebas fácilmente y ser reutilizables.

Karma no es una librería de pruebas, es una herramienta lanzamiento de pruebas y de generación de texto que genera el fichero del análisis de pruebas realizadas.

[1] Descripción tomada de Wikipedia: La cobertura de código es una medida (porcentual) en las pruebas de software que mide el grado en que el código fuente de un programa ha sido comprobado con pruebas de software. Sirve para determinar la calidad del test que se lleve a cabo y para determinar las partes críticas del código que no han sido comprobadas y las partes que ya lo fueron, además se puede utilizar como técnica de optimización dentro de un compilador optimizador para llevar a cabo una eliminación de código muerto, más específicamente sirve para detectar código inalcanzable

5.1.5.3. Cypress

A diferencia de las tecnologías anteriores, Cypress se enfoca en las pruebas de interfaz, estas se centran en el rendimiento, tiempos de respuesta de la aplicación, validación de componentes y validación de comportamientos deterministas.

Cypress surgió como alternativa a Selenium, Cypress se posicionó mejor ante las altas demandas de las nuevas aplicaciones web que ofrecían novedades muy rápido, es por ello que Cypress se centra en la validación de cualquier aplicación que se ejecute en un navegador.

5.1.5.4. Otros

Hay muchos *frameworks* de pruebas (*testing*) entre ellos también destacan los siguientes.

- Selenium IDE [31].
- WebDriverIO [32].
- Serenity [33].
- Mocha [34].

5.2. Elección para el proyecto

Para el desarrollo de este proyecto se han analizado muchas y distintas herramientas, al tratarse de un proyecto de interfaz web se han analizado proyectos creados con Node y NPM (*Node Package Manager*) pues es la tecnología por excelencia para cualquier de desarrollo web moderno.

La decisión final de usar Angular y Electron frente a React y React-native ha sido simple, usando este conjunto de herramientas ha sido más simple añadir ciertas dependencias necesarias tales como el editor de texto Monaco o el gestor de ficheros DevExpress.

Estos son los proyectos principales desarrollados y sus ficheros `package.json`, en cada uno de estos ficheros se especifican las dependencias finales seleccionadas para cada uno.

La decisión final para el desarrollo del proyecto del servidor de pruebas se decantó por usar TypeScript y la librería Socket-io principalmente, ya que ambas tecnologías combinan y se dan soporte mediante los `@types`, esto permite un desarrollo mucho más sencillo al estar todo en un mismo contexto y lenguaje de programación, otra de las principales ventajas al elegir estas dos tecnologías es que ambas tienen una comunidad de desarrolladores muy grande que puede ayudarnos en caso de dudas de implementación.

- `THUMDER` [35] THUMDER package.json
- `THUMDER-server` [36] THUMDER-server package.json
- `THUMDER-pixi` [37] THUMDER-pixi package.json

A continuación, se explican en detalle las tecnologías que se van a usar.

5.2.1. Node.js

Node.js es un entorno de ejecución multiplataforma, de código abierto, para la capa del servidor basado en el lenguaje de programación JavaScript, se ejecuta, por tanto, como un script, pero permite tareas asíncronas, usa una arquitectura orientada a eventos para la entrada y salida de datos y basado en el motor V8 de Google [38].

Usamos Node.js para la ejecución del código JavaScript, esto lo podemos usar para ejecutar el código generado mediante TypeScript, esta herramienta se usa para muchas y distintas tareas, desde la construcción de la aplicación de Angular, el servidor de desarrollo de Angular o incluso el proyecto completo del servidor o la creación del backend.

Nos planteamos el uso de Node.js para el desarrollo de un posible servidor, la gestión de conexiones entre la aplicación del cliente y el backend.

Node.js nos permite editar el tamaño de su cola de procesos y de hilos, aunque esto es algo interno de su código fuente y de su `EventLoop` [39].

Las principales características de Node.js es que nos permite escalar horizontalmente, es de ejecución rápida, tiene muchas librerías mediante paquetes, tiene una ejecución sólida, es multiplataforma y es mantenible.

5.2.1.1. NPM - Node Package Manager

Node Package Manager (NPM) es un gestor de paquetes desarrollado usando JavaScript, mantenido por la comunidad y por Isaac Schlueter, usa el motor de V8 de Google, este gestor nos permite obtener librerías, agregar dependencias, distribuir paquetes y administrar módulos o el proyecto en general.

Todos los proyectos bajo **NPM** usan el fichero `package.json`, en este fichero se definen los datos más relevantes para el proyecto, el identificador único, la versión, así como el autor o equipo del proyecto, la URL del repositorio, una descripción del proyecto, palabras clave del proyecto, una serie de scripts o rutinas importantes para el proyecto en cuestión y lo que es más importante el listado de paquetes y dependencias que usa el proyecto.

Estos listados de dependencias es lo que usa NPM para descargarse las dependencias y usarlas en el proyecto.

5.2.2. TypeScript

TypeScript es un lenguaje de programación de código abierto desarrollado y mantenido por Microsoft, es un superconjunto de JavaScript que permite definir tipos de datos y documentar el código mediante tipos de datos.

El uso de TypeScript por parte de la comunidad ha sido uno de los principales motivos por lo que el desarrollo de nuevas características de JavaScript se ha acelerado, se podría decir que el TypeScript ha sido el promotor de nuevas características en JavaScript, tanto en tecnologías en el ámbito de cliente (navegador) o en el ámbito de servidor (Node.js).

El código JavaScript funciona sobre TypeScript, pues TypeScript extiende la sintaxis.

Lo usamos en nuestro proyecto, ya que nos permite definir los datos y escalar nuestro código de una forma mucho más mantenible, además mantiene un código más limpio y claro.

En nuestro proyecto se usará la versión 3.7.9 de TypeScript.

5.2.3. Angular

Angular es un framework de código abierto desarrollado y mantenido por Google, nació en 2009 con el nombre AngularJS, su desarrollo cambio en 2016 con la implementación de TypeScript, fue pensado por y para realizar un desarrollo Web en el cliente (navegador) y tener una alta eficiencia.

Pensado para ejecutarse solamente en el cliente sin recargas, ya que tan solo realizan las comunicaciones con el servidor que sean necesarias (obtener los datos necesarios mínimos mediante distintas APIs), todas sus vistas y componentes se cargan de forma dinámica.

Implementa muchas herramientas tales como `services`, `pipes`, `templates`, `containers`, `forms`, `WebWorkers`, `routing`, `navigations`, seguridad y utilidades de accesibilidad. Así como herramientas de desarrollo rápido.

Angular proporciona una herramienta para el control del proyecto `angular-cli`, la herramienta de consola se instala usando `npm` al igual que el proyecto.

Angular CLI es el programa de líneas de comandos que permite activar o crear tareas automáticas, entre ellas de generar código como módulos, vistas, componentes y directivas "`ng generate ...`" y otras más interesantes como compilar la aplicación mediante su propio framework o añadir nuestro framework de compilación "`ng build ...`", cuenta con un conjunto de soluciones muy amplio.

Este framework proporciona una solución completa a la mayoría de problemas que requieren una solución en el desarrollo web.

5.2.3.1. Arquitectura Modelo-Vista-Modelo *MVVM*

Angular se basa en la arquitectura *MVVM* que implementa el enlace bidireccional de datos, esto significa que si se cambia un dato en nuestro modelo se actualiza de forma inmediata el dato en concreto en la vista.

Esto permite tener un bajo acoplamiento, reutilización de las vistas, desarrollo separado del dominio del diseño, por lo que pueden ser probadas por separado la lógica del proyecto de la capa del diseño UI.

El patrón *MVVM* en Angular se divide en cuatro partes, Vista, Modelo-Vista, Modelo y Controlador o servicio.

5.2.4. Electron

Electrón es un *framework* de código abierto mantenido actualmente por GitHub que permite el desarrollo de aplicaciones web embebidas en un navegador y pudiendo usar tecnologías de aplicaciones de escritorio a la misma vez, originalmente desarrollado para aplicaciones web que se ejecutan solo en una máquina sin necesidad de un servidor.

Utiliza Node.js [18] para el motor de ejecución y Chromium para la interfaz.

Esta herramienta la utilizamos para realizar compilaciones de nuestro proyecto, de forma que creemos un ejecutable que pueda usarse en Windows, Linux o macOS, aunque el proyecto se ha desarrollado para una versión de navegador o página web.

5.2.5. Dependencias externas

Para el desarrollo de este proyecto se han utilizado múltiples librerías de diversas funcionalidades, además de programas de limpieza de código o de pruebas, para ello se han utilizado las siguientes librerías a las que se hace mención por ser más relevantes en el proyecto.

1. `ngx-translate`, `ngx-toastr`, `ngx-socket-io`, `ngx-cookie-service`: traducciones, mensajes, notificaciones, comunicación, gestión de *cookies*.

2. `Firebase`: base de datos, gestión de usuarios, inicio de sesión, registro, cambio de contraseña, etc.

3. `Monaco-editor`: este es un proyecto de un editor de código mantenido por Microsoft creado para la edición de los ficheros en una interfaz web, este proyecto surgió del IDE Visual Studio Code.

4. `Pixi.js`: se trata de un motor de videojuegos pensado para web, con él haremos las animaciones e interfaces gráficas que requieran representaciones difíciles de implementar en HTML usando un *canvas* HTML 5 y el motor.

5. `DevExtreme`: esta es una *suite* de componentes UI mantenida por la compañía DevExtreme [40], esta *suite* se centra en interfaces de múltiples tipos, en nuestro proyecto nos resulta muy interesante y útil, ya que debemos implementar el gestor de ficheros y carpetas para representar los distintos programas que estamos desarrollando.

6. `Jest`: este es un *framework* de JavaScript pensado para pruebas (*tests*), antiguamente mantenido por Meta [30], se centra especialmente en la simplicidad y la claridad de los test, estos lo usaremos para validar la lógica de nuestras funciones y en especial para nuestro servidor de pruebas.

7. `Cypress`: *framework* de JavaScript y `Electron`, mantenido por la compañía Cypress creada por *Brian Mann*, este proyecto se caracteriza por ser un entorno completo de pruebas unitarias y de interfaz que permite visualizar todas las pruebas que se desarrollen, crear un informe completo y mostrar los datos y fallos que se produzcan. Este framework lo usaremos para el cliente, pues se integra muy bien junto a Angular.

En caso de desear consultar alguna de estas tecnologías, lo más sencillo es leer el documento `README` que hay en su correspondiente repositorio, todo esto puede ser encontrado en https://www.npmjs.com/.

Capítulo 6

Desarrollo

6.1. Introducción

Para el desarrollo del proyecto debemos seguir unos pasos para facilitar el proceso, primero debemos definir las capas más externas de nuestra interfaz, el `layout`, a continuación, los `components`, después los `shared` y por último, las `views`, el proceso de desarrollo del **core** depende de los componentes y vistas.

Cada una de estas secciones están estructuradas de forma jerárquica, lo que hace que el proyecto cargue y descargue de la memoria las secciones en función de la vista activa.

Por otra parte, se encuentra nuestro núcleo de ejecución, el núcleo estará siempre cargado en memoria y en ejecución, pues se trata de un servicio, por tanto, serán las vistas las que se subscriban a los flujos de datos correspondientes.

Esto se complementa con la gestión de la sesión y de los ficheros mediante el SDK de Firebase

6.2. Rutas, componentes y vistas

El proyecto se ha dividido en secciones llamadas vistas usando rutas, de esta forma podemos mejorar el rendimiento de la aplicación y mostrar solo las secciones que estemos interesados, esto lo hacemos, ya que podemos aprovecharnos de una de

las características internas de Angular, si usamos los métodos internos de Angular como `[RouterLink]` la página no se recarga, solamente cambiamos la sección de visualización, de esta forma el estado de la máquina y la simulación no se ve alterado ni reiniciado.

Código de ejemplo 6.1 app-routing.module.ts

```
const routes: Routes = [
  {
    path: "", component: LayoutAuthComponent, canActivate: [AuthGuard],
    children: [ {path: "", component: IndexView, pathMatch: "full"} ],
  },
  {
    path: "auth", component: LayoutAuthComponent, canActivate: [AuthGuard],
    data: { breadcrumb: "Home" },
    children: [
      { path: "", redirectTo: "login", pathMatch: "full" },
      { path: "calculator", component: CalculatorView, data: { breadcrumb: "Calculator" } },
      { path: "code", component: CodeView, data: { breadcrumb: "Code" } },
      // ...
    ],
  },
  { path: "login", component: LoginView, canActivate: [NoAuthGuard] },
  { path: "register", component: RegisterView, canActivate: [NoAuthGuard] },
  { path: "forgot-password", component: ForgotPasswordView, canActivate: [NoAuthGuard] },
  { path: "**", component: PageNotFoundComponent },
  // ...
];
```

6.3. Ciclo de ejecución de las tecnologías

6.3.1. Ciclo de vida y de ejecución de la aplicación (Angular)

El ciclo de compilación y luego de ejecución de Angular como servidor es mucho más complejo, ya que ahí, es donde está toda la lógica del proyecto, este proceso pasa por distintas etapas, pero podemos agruparlas en dos, definición de módulos y de componentes.

Los módulos agrupan componentes y los componentes encapsulan la lógica de las distintas partes del proyecto, además para que un componente y módulo tengan acceso a otros componentes y módulos, estos deben definirse como exportables e importarlos en el módulo que vamos a usar.

Dentro de THUMDER hay 4 módulos, `core`, `Components`, `shared` y `app`, esto se hace por eficiencia en los tiempos de compilación de la aplicación por la recarga activa y por encapsular la lógica.

El proceso de Angular es el siguiente.

1. Angular inicia desde un `main.ts` con el `Web-Component` definido en el fichero `index.html`, en este caso y por defecto el nodo raíz es `<app-root>`, este es el `Web-Component` encargado de iniciar la aplicación.

2. A continuación, se carga el `app.module.ts` que se encarga de las definiciones, exportaciones e importaciones de los distintos componentes y módulos, para ello hay que importarlos y añadirlos a las distintas estructuras del `NgModule`. Aquí hay dos componentes muy importantes, el editor `Monaco` de Microsoft [41], así como el servicio `MachineService` que se ha creado para el proyecto, usamos una característica especial de Angular para cargar esta parte con una clase tipo `Factory`, esto lo hacemos para que el estado de la máquina este en ejecución siempre y no se descargue, esto lo hemos definido en `Utils` como `Utils.initServicesFactory()`.

3. Por último, se carga el sistema de rutas (`app-routing.module.ts`) que carga las distintas partes del proyecto, así como los `layouts`, `views` y los `guards`.

   3.1 De forma inmediata carga, el `app.component.ts` esta es la parte encargada de gestionar el sistema de traducciones y de la comunicación con el servicio `ElectronService`, de esta forma siempre es accesible el sistema de traducciones y cambiarlo en cualquier momento de la ejecución de la aplicación, además también tenemos acceso a la tecnología de Electron.

Por último, indicar el orden de las etapas por las que pasa un componente, el comportamiento es interno de toda aplicación de Angular, pero es relevante para nuestro proyecto, pues lo usamos para activar las subscripciones, liberar memoria, realizar peticiones, etc.

1. `ngOnChanges`	2. `ngOnInit`	3. `ngDoCheck`
4. `ngAfterContentInit`	5. `ngAfterContentChecked`	6. `ngAfterViewInit`
7. `ngAfterViewChecked`	8. `ngOnDestroy`	

Tabla. 6.1: Angular - Ciclo de vida de un componente

Los métodos más relevantes son `ngOnInit`, `ngAfterViewInit` y `ngOnDestroy`.

- `ngOnInit`: permite inicializar el componente una vez que ha recibido las propiedades de entrada.
- `ngAfterViewInit`: se ejecuta una vez que la vista del componente y sub componentes ha finalizado de cargar.
- `ngOnDestroy`: se ejecuta una sola vez, antes de que Angular destruya el componente, se usa para liberar memoria, subscripciones y eventos de escucha.

En grandes rasgos, una aplicación Angular como es nuestro proyecto se organiza como en el siguiente ejemplo.

Código de ejemplo 6.2 app.module.ts

```
@NgModule({
  declarations: [
// AppComponent
// Vistas y componentes
  ],
  imports:      [
// Módulos Core, Components y Shared
// Módulos externos
  ],
  providers:    [
// AppComponent
// Guards Auth y NoAuth
// Servicios internos
  ],
  exports:      [
// Al tratarse del main no exportamos nada, pero en los módulos sí.
  ],
  bootstrap:    [
// AppComponent
  ]
})
export class AppModule {
}
```

6.3.2. Ciclo de vida y de ejecución de la aplicación (Electron)

El ciclo de compilación de Electron es más sencillo de lo que puede parecer, ya que tenemos dos instancias, una para la versión web (Angular) y otra para la compilada junto con Electron, toda la complejidad del sistema de Electron y que inicia el programa está en el fichero `main.ts` en la raíz del proyecto, que crea una instancia de Node con Angular en caso de que se ejecute como una aplicación de escritorio.

Código de ejemplo 6.3 main-Electron.ts

```
// Create the browser window.
win = new BrowserWindow({
    x: 0,
    y: 0,
    width: size.width,
    height: size.height,
    webPreferences: {
        nodeIntegration: true,
        nativeWindowOpen: true,
        allowRunningInsecureContent: (isServe),
        contextIsolation: false,
        enableRemoteModule: true
    },
});

if (isServe) {
    require('electron-reload')(__dirname, {
        electron: require(`${__dirname}/node_modules/electron`)
    });
    win.loadURL('http://localhost:4200');
} else {
    win.loadURL(url.format({
        pathname: path.join(__dirname, 'dist/index.html'),
        protocol: 'file:',
        slashes: true
    }));
}
```

6.4. Núcleo (*Core*)

Aquí se recoge a grandes rasgos la lógica de nuestro proyecto, los servicios y suscripciones que comunican la información, esto se representa mediante las clases y los flujos de información.

1. `AuthService`: gestiona los usuarios, registros, inicios de sesión y la base de datos de Firebase.
2. `MachineService`: gestiona el estado de la máquina y la comunicación entre los componentes y el servidor.
   - 2.1 `ManagerBreakpoints`: gestiona los puntos de parada (breakpoints) de nuestro fichero a simular.
   - 2.2 `ManagerMemory`: gestiona la memoria de la máquina a simular.
   - 2.3 `ManagerRegisters`: gestiona los registros de la máquina a simular.
   - 2.4 `ManagerStatistics`: gestiona las estadísticas de la simulación.
3. `ElectronService`: gestiona el estado de `Electron` y la interfaz Chromium, notificaciones y datos locales.

4. `FileSystemService`: gestiona el sistema de ficheros local y la base de datos de Firebase mediante un *CRUD* y suscripciones.
5. `StorageService`: gestiona la base de datos local y los datos por defecto.

6.4.1. AuthService

Este es el servicio encargado de la comunicación con Firebase, permite a nuestra aplicación usar una serie de herramientas desde autenticación usando OAuth2 o bases de datos NoSQL, bases de datos en tiempo real, almacenamiento, aprendizaje automático, análisis, pruebas, etc.

Esta herramienta cuenta con muchas utilidades, aunque en nuestro proyecto solo la vamos a usar para la autenticación de los usuarios, la autenticación podrá hacerse de tres modos, se han activado y configurado los proveedores de Google, de GitHub e inicio de sesión mediante correo electrónico y contraseña.

Código de ejemplo 6.4 AuthService.ts

```
1  public async SignIn(email, password): Promise<boolean>;
2  public async SignOut(): Promise<void>;
3  public async SignUp(email, password): Promise<UserCredential | void>;
4  public async SendVerificationMail(userCredential: UserCredential): Promise<void>;
5  public async ForgotPassword(passwordResetEmail): Promise<void>;

7  public async GoogleAuth(): Promise<void>;
8  public async GithubAuth(): Promise<void>;

10 private async AuthLoginAnonymously(): Promise<void>;
11 private async AuthLogin(provider): Promise<void>
```

6.4.2. MachineService

La emulación de la máquina se realiza mediante los servicios con subscripciones y observables de Angular, usando la librería `rxjs`, es aquí donde está definido el componente de comunicación con el servidor, este es el servicio encargado de interpretar y gestionar las respuestas del servidor.

MachineService es el servicio con la lógica de representación del estado de la simulación, se han definido distintas clases y gestores (`Managers`), para representar, controlar y usar los datos en cada componente, se hace uso de distintas herramientas y estructuras de datos, se han definido los registros de DLX [3]:

Tipo de dato	**Nombre del registro**				
Int32	PC	IMAR	IR	A	AHI
	B	BHI	BTA	ALU	ALUHI
	FPSR	DMAR	SDR	SDRHI	LDR
	LDRHI	16 registros especiales			
Array<Int32>	R[0-31]	32 registros de enteros			
Array<Float32>	F[0-31]	32 registros de coma flotante de simple precisión.			
Array<Double64>	D[0-15]	16 registros de coma flotante de doble precisión.			

Tabla. 6.2: Tabla de registros

Las clases y tipos de datos definidas como `Int32`, `Float32` y `Double64` corresponden a las limitaciones técnicas de un registro de su tipo. [1]

Estas estructuras son útiles, ya que nos permiten gestionar, detectar y notificar errores en rangos de datos no permitidos.

Este servicio también se encarga de la gestión de la simulación, aunque esto se explica en más profundidad en la sección del intérprete (*runner*) 6.10.

Este es el servicio encargado de proporcionar las estructuras y métodos necesarios para que cada componente pueda adecuarse a los datos en cada etapa, esto lo hacemos mediante `Observable<Type>`, `Subscription` y `Subject<Type>`, también es el servicio encargado de gestionar la ejecución mediante métodos asíncronos como son `play, pause, resume, end, reset o nextStep`, por último, también es el método encargado de comunicarse con el servidor y notificar si debe actualizar los registros o la memoria.

[1] Los límites de los números enteros al ser almacenados en conjuntos de 32 bits [42].

Código de ejemplo 6.5 MachineService.ts

```
1  public resetMachineStatus(): Promise<boolean>;
2  public getResetObservable: Observable<void>;
3  public getStepObservable: Observable<number>;
4  public getLineObservable: Observable<number>;
5  public getIsRunningObservable: Observable<boolean>;
6  public getIsCompleteObservable: Observable<boolean>;
7  public getStepSimulationObservable: Observable<TypeSimulationStep>;
8  public getCodeSimulationObservable: Observable<TypeInstructionsData[]>;
9  public getDebuggerObservable: Observable<number>;
10 public getDataStatisticsObservable: Observable<TypeDataStatistics>;
11 public getLoggerObservable: Observable<string>;
12 public getStatusWebsocketObservable: Observable<"Connect" | "Disconnect">;
13 public getErrorsInCodeObservable: Observable<TypeErrorInCode[]>;

15 public async play(): Promise<void>;
16 public async reset(): Promise<void>;
17 public async nextStep(): Promise<void>;
18 public async pause(): Promise<void>;
19 public async resume(): Promise<void>;
20 public async end(): Promise<void>;

22 private async SimulationInit(): Promise<boolean>;
23 private async SimulationNextStep(): Promise<void>;
24 private async CheckConditions(): Promise<boolean>;
25 private async ProcessStep(): Promise<void>;

27 public async updateRegisterInServer(registersToUpdate: TypeRegisterToUpdate[]): Promise<boolean>;
28 public async updateMemoryInServer(memoryToUpdate: TypeMemoryToUpdate[]): Promise<boolean>
```

6.4.3. ElectronService

El proyecto al tratarse de una aplicación multiplataforma que se ejecuta en navegador y en escritorio, hay ciertas funciones que son del sistema operativo desde el que se ejecuta, el servicio `ElectronService` se encarga de ello.

- Detectar el sistema operativo.
- Detectar la arquitectura del ordenador (x86, x64).
- Detectar la versión de Node.
- Enviar mensajes a `Electron`.

El proyecto Angular no puede ejecutar todo el código de `Electron`, pero sí podemos enviar mensajes con argumentos a `Electron`, estos mensajes corresponden a funciones con sus correspondientes argumentos.

Esta característica la usamos para crear notificaciones nativas del sistema operativo, abrir enlaces en el navegador predeterminado, ajustar parámetros, recargar la aplicación, etc.

6.4.4. FileSystemService y FileSystemStorageService

La gestión de los ficheros la realizamos mediante dos servicios (`FileSystemService` y `FileSystemStorageService`), una parte se encarga de la lógica interna del gestor de ficheros, esta será la de obtener los ficheros, crear ficheros, editar un fichero o borrar un fichero y el otro servicio `FileSystemStorageService` será el encargado de la comunicación y la lógica de la base de datos.

La división servirá como interfaz entre el gestor y los ficheros (`FileSystemService`) y entre los ficheros y la base de datos (`FileSystemStorageService`), esto lo hacemos para separar la lógica de ficheros y la base de datos por si en un futuro requiera cambiar de base de datos no tener un código acoplado.

Como podemos ver en el siguiente código hemos usado el sistema de promesas de JavaScript mediante `Async / Await`, esto lo usamos para desarrollar un código mucho más simplificado tratando con elementos asíncronos como lo son una base de datos de `firestore` [2].

Código de ejemplo 6.6 file-system.service.ts

```
1 public async init(): Promise<void> {...}
2 public getUpdateUIObservable(): Observable<void> {...}
3 public async getItems(path: FileSystemItem): Promise<Array<THUMDER_FileItem» {...}
4 public async createFile(item: FileSystemItem, extension: string): Promise<boolean> {...}
5 public async editFileItem(updateFileItem: THUMDER_FileItem, $key: string): Promise<void> {...}
6 public async renameItem(item: FileSystemItem, newName: string): Promise<THUMDER_FileItem> {...}
7 public async deleteItem(item: FileSystemItem): Promise<void> {...}
```

[2] *firestore* es una herramienta de Firebase.

Código de ejemplo 6.7 file-system-storage.service.ts

```
1  public async isInitialize(): Promise<boolean> {...}
2  public async generateDefaultFiles(): Promise<number> {...}

4  public queryFileFromUser(filename: string): Promise<QuerySnapshot<DocumentData» {...}
5  private queryAllFilesFromUser(): Query<DocumentData> {...}
6  private collectionFileItems(): Promise<QuerySnapshot<DocumentData» {...}

8  // Observable<InterfaceFileItem[]>
9  public getAllFilesFromFirestoreAsObservable(): Observable<THUMDER_FileItem[]> {...}
10 public async createFileItem(fileItem: THUMDER_FileItem): Promise<boolean> {...}
11 public async deleteFileItem ($key: string): Promise<boolean> {...}
12 public async updateFileItem ($key: string, fileItem: THUMDER_FileItem): Promise<boolean> {...}

14 // Documents
15 public FileItem_Documents_valueChanges(id): Observable<THUMDER_FileItem> {...}
16 public FileItem_Documents_snapShotChanges(id): Observable<DocumentSnapshot<THUMDER_FileItem» {...}

18 // Collections
19 public FileItems_Collections_valueChanges(): Observable<THUMDER_FileItem[]> {...}
20 public FileItems_Collections_snapShotChanges(): Observable<DocumentChange<THUMDER_FileItem>[]> {...}
21 public async getAllFilesFromFirestore(): Promise<QuerySnapshot<THUMDER_FileItem» {...}
22 public async getFiles(): Promise<THUMDER_FileItem[]> {...}
23 public createNewDocument() {...}
```

6.5. Código de los componentes (*Shared*)

Es la capa encargada de la lógica del proyecto que no está ligada a partes concretas del proyecto, sino que puede ser usada en cualquier parte y momento requerido, como botones que requieran de eventos (`async`, `await`), lógica de sesión, lógica para transformar datos o funciones, etc.

1. `Components`: son elementos visuales compartidos, se pueden describir como secciones de una interfaz o vista, estos se usan para mostrar información estática, como errores del servidor, accesos indebidos, etc.

2. `Directives`: son clases que agregan comportamientos adicionales a elementos, tal que botones, formularios, etc.

3. `Guards`: son elementos lógicos que se añaden a las rutas para analizar y permitir o denegar el acceso a la misma, además de poder modificar sus datos.

4. `Pipes`: son una herramienta de Angular que nos permite transformar visualmente la información.

6.6. Distribución de la interfaz (*Layout*)

La visualización y distribución de la aplicación cambia dependiendo de si el usuario ha iniciado sesión o no. Para controlar esta distribución se han creado distintas formas de organizar y distribuir la interfaz.

Una de las buenas prácticas en el momento de diseñar e implementar, es dejar la posibilidad de ampliación, para ello se plantea que se pueden aumentar el número de actores, por ello en el desarrollo se deja la posibilidad de cambiar la distribución de la interfaz en función del actor que interactúa.

Como ya se ha hablado anteriormente la aplicación depende del estado de la sesión del usuario, es mediante esta gestión que el sistema muestra u oculta ciertos elementos y secciones, esta gestión se realiza mediante los `Guards`, es en los `Guards` donde se definen las reglas de validación que usamos para cada sección, es con ello que se definen sus permisos.

En el siguiente código podemos observar la forma en la que se ha definido la validación para una sección, en este caso casi toda la responsabilidad del método recae en el servicio `AuthService`, aunque dependiendo de la aplicación, las condiciones que se pueden dar para activar una sección pueden ser distintas.

Código de ejemplo 6.8 NoAuthGuard.ts

```
1  // GUARDS ->NoAuthGuard
2  export class NoAuthGuard implements CanActivate {
3    constructor(public authService: AuthService, public router: Router) {
4    }
5    public async canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Promise<boolean> {
6      if (this.authService.isLoggedIn) {
7        await this.router.navigateByUrl("/");
8      }
9      return Promise.resolve(!this.authService.isLoggedIn);
10   }
11 }

13 // app-routing.module.ts
14 const routes: Routes = [
15   // ...
16   // no auth views
17   {
18     path: "login",
19     component: LoginView,
20     canActivate: [ NoAuthGuard ] // <- NoAuthGuard
21   },
22   // ...
23 ];
```

Los componentes y secciones de la aplicación se organizan de forma jerárquica, esto lo hacemos mediante la plantilla de `AdminLTE` y el uso de los componentes de *Angular*.

```
<div class="wrapper">
    <THUMDER-auth-navbar></THUMDER-auth-navbar>
    <THUMDER-aside-left></THUMDER-aside-left>

    <div class="content-wrapper">
        <!-- ... -->
        <THUMDER-breadcrumb></THUMDER-breadcrumb>
        <!-- ... -->
        <main class="content">
            <router-outlet></router-outlet>
        </main>
    </div>

    <THUMDER-aside-right></THUMDER-aside-right>
    <THUMDER-footer></THUMDER-footer>
</div>
```

Código de ejemplo 6.9 layout-auth.component.html

Por último, aunque nuestra aplicación solo usa `Auth` y `Landing`, se ha creado la distribución `Admin` por si en un futuro la aplicación requiera la gestión de un administrador con control en la base de datos o ciertos permisos superiores.

6.7. Componentes (*Components*)

Este es el listado completo de los componentes de la aplicación, estos componentes se han desarrollado y adaptado para las distintas vistas.

- `aside-left & aside-right`
- `breadcrumb`
- `footer`
- `edit-memory-binary32`
- `edit-register-binary32`
- `modal & modal-bottom`
- `monaco-editor`
- `navbars-auth & navbars-admin`
- `pixi-cycle-clock-diagram`
- `pixi-pipeline`
- `sidebar`
- `docs-item`

Gran parte de estos componentes se han extraído de la librería **AdminLTE** [21] transformando todas las secciones visibles en componentes de `Angular` como son el `Aside`, `Navbar`, `Card`, `Breadcrumb`, `Footers`, `Sidebar`, etc.

Mientras que otros componentes como `xterm` o `monaco-editor` [41] se han usado librerías externas para añadirlos, modificando y adaptando la librería a nuestro proyecto de Angular-Electron.

Para los componentes de renderizado, es decir `pixi-cycle-clock-diagram` [37] o `pixi-pipeline` [37] estamos usando **PixiJS** que es un motor de videojuegos para la web, nosotros hemos creado dos clases que extienden de la clase `PIXI.Container` del paquete de npm `pixi.js`, representamos ciertos gráficos, como tablas, listas o flechas que hemos tenido que calcular y representar matemáticamente [43] usando este motor.

Otro componente relevante es el editor de código `Monaco` desarrollado por Microsoft para el editor de código *Visual Studio Code*, con este editor web podemos resaltar el código ensamblador en nuestro proyecto definiendo una platilla de colores para marcar palabras reservadas del lenguaje usando expresiones regulares.

6.8. Vistas (*Views*)

Este es el listado completo de todas las secciones que tiene nuestra aplicación, muchas de ellas son derivaciones de otras, como es el caso de crear cuenta o iniciar sesión, memoria, registros y código máquina, etc.

1. `_index`
2. `_account`
   - 2.1 `forgot-password`
   - 2.2 `login`
   - 2.3 `register`
3. `_landing`
   - 3.1 `about`
   - 3.2 `landing`
4. `_auth`
   - 4.1 `_views`
   - 4.2 `calculator`
   - 4.3 `code`
   - 4.4 `config`
   - 4.5 `docs`
   - 4.6 `editor`
   - 4.7 `file-manager`
4. `_auth`
   - 4.8 `logger`
   - 4.9 `memory`
   - 4.10 `pixi-cycle-clock-diagram`
   - 4.11 `pixi-pipeline`
   - 4.12 `registers`
   - 4.13 `statistics`

Ciertas vistas usan las directivas propias de la API de Angular y de la API de Angular Material [44], estas son `<ng-content>`, `cdkDropList` o `cdkDrag`, esto lo usamos para

extender el comportamiento de un componente, añadir la propiedad de arrastrar y soltar (*drag and drop*) a algunos elementos, pudiendo modificar la disposición de los elementos.

6.8.1. Configuración

La finalidad de esta sección es la de poder modificar la configuración inicial de la máquina, ciertos parámetros del editor de código, la sección de la vista múltiple o la URL del servidor.

Para ello se han usado los métodos para formularios HTML de la API de `Angular`, tales como `[ngModelOptions]`, `[(ngModel)]` o los eventos `(change)`, permitiendo hacer que los formularios sean reactivos.

Se han creado algunos métodos de validación propios, estos corresponden a la cantidad de unidades de coma flotante, `checkCount(...)`, a sus tiempos de cálculo `checkDelay(...)`, además de la opción de activar o desactivar los adelantamientos, `updateEnableForwarding(...)`, etc. Estos métodos validan que los datos de entrada de la máquina estén dentro de los rangos válidos.

Cuando el usuario introduce la nueva configuración para el simulador y pulsa el botón de guardar (*save*) o ejecuta el reinicio del simulador (*reset*), se actualizan las estructuras internas del proyecto (*Pipeline*, *CycleClockDiagram*, *Code*, *Memory*, etc.). El sistema envía una petición al servidor y el servidor debe responder cumpliendo el protocolo, al final de la acción se guarda la nueva configuración en el `LocalStorage` del navegador.

Debemos recordar que al cambiar la configuración ya sea para actualizar o reiniciar la simulación de la máquina, la información en la memoria de la máquina se reinicia, dejando toda la memoria a 0.

Multiview:

Para la gestión de la vista múltiple (*multiview*), usamos dos listas de elementos, cada lista corresponde a una columna de la vista múltiple, estos son elementos de texto (`string`), es decir tenemos dos listas de texto con los identificadores de los componentes. Estos elementos se usan activando y desactivando elementos de tipo `input checkbox`, los elementos del formulario se combinan con la propiedad de poder arras-

trar y soltar (*drag and drop*) de la API de Angular-Material, de esta forma organizamos la distribución de la vista múltiple, arrastrando, soltando, activando y desactivando los elementos de la vista.

Para la propiedad de arrastrar y soltar usamos las directivas `cdkDrag` que habilita el desplazamiento de elementos en la interfaz, `cdkDragHandle` da la propiedad de gancho para los elementos de arrastrar y soltar, `cdkDropList` actúa como enlazador para vincular los elementos `cdkDrag` que estén dentro de sus elementos HTML hijos y `cdkDropListGroup` que agrupa las listas en grupos creando zonas donde soltar el elemento.

Por último están las directivas que habilitan eventos y enlazan las listas, estas son `[cdkDropList]` qué habilita la transferencia de elementos *drag and drop* entre zonas de colocación conectadas. Puede conectar una o más instancias de `cdkDropList` juntas, configurando la propiedad `[cdkDropListConnectedTo]` o envolviendo los elementos en un elemento con el atributo `cdkDropListGroup`.

6.8.2. Gestor de ficheros

Tal y como expresamos en el Apartado 6.4.4, el gestor de ficheros es el encargado de representar estos ficheros y de ser una interfaz ante los datos (metadatos) de los ficheros y de la base de datos.

Como ya hemos indicado anteriormente al desarrollar una aplicación de este tipo, web y multiplataforma, debemos tener muchos más factores y posibles errores en cuenta, ya que la aplicación se ejecuta de forma multiplataforma y al gestionar los ficheros mediante una sesión o cuenta, en una nube, puede suceder que un recurso esté siendo compartido en varios sistemas operativos, varios navegadores o varios sistemas operativos y navegadores de forma simultánea, para solucionar este problema se ha decidido desarrollar un código de suscripciones y observables, esto hace que una actualización en el recurso compartido sea notificada a los subscriptores, siendo estos actualizados.

Código de ejemplo 6.10 FileManager.ts

```
this.fileSystemService.getUpdateUIObservable().subscribe(async () => {
  await this.updateUI();
});

this.fileSystemService.init().then(() => {
  this.customFileProvider = new CustomFileSystemProvider({
    getItems: async (parentDirectory: FileSystemItem) => {
      return await this.fileSystemService.getItems(parentDirectory);
    },
    createDirectory: async (parentDirectory: FileSystemItem, name: string) => {
      return await this.fileSystemService.createDirectory(parentDirectory, name);
    },
    renameItem: async (item: FileSystemItem, name: string) => {
      return await this.fileSystemService.renameItem(item, name);
    },
    deleteItem: async (item: FileSystemItem) => {
      return await this.fileSystemService.deleteItem(item);
    },
    moveItem: async (item: FileSystemItem, destinationDirectory: FileSystemItem) => {
      return await this.fileSystemService.moveItem(item, destinationDirectory);
    },
    uploadFileChunk: async (fileData, uploadInfo, destinationDirectory: FileSystemItem) => {
      return await this.fileSystemService.uploadFileChunk(fileData, uploadInfo, destinationDirectory);
    },
    downloadItems: async (items: FileSystemItem[]) => {
      return await this.fileSystemService.downloadItem(items);
    }
  });
});
```

6.8.3. Editor de ficheros

El editor se ha dividido en dos partes (`views` y `components`), una abstracción del editor `component`, que sirve como una implementación genérica para cualquier tipo de situación en la que se requiera editar un fichero usando las características de un *IDE* [45], por otra parte, la implementación para nuestro proyecto se encuentra en `view`, es ahí donde definimos el `language` y el `theme` que se aplican para el lenguaje DLX.

Posibles características de un *IDE*:

1. Editor de texto.
2. Intérprete y compilador.
3. Herramientas para la automatización de tareas.
4. Depurador de código.
5. Sistema de control de versiones (la mayoría).

6. Sistema de diseño GUI o interfaz gráfica de usuario (botones, *debugger*, listas desplegables, etc.).

Para tratar el editor como un IDE, debe tener ciertas características, las más relevantes para nosotros son las de auto completado, documentación, detectar y resaltar palabras clave, para ello usamos expresiones regulares, aquí hay un ejemplo no implementado de cómo se ha realizado para nuestro proyecto, solo explicamos cómo se haría mediante `monaco` sin entrar en detalles técnicos.

Código de ejemplo 6.11 monaco-config.ts

```
// Registra el lenguaje para DLX
monaco.languages.register({id: 'thumderLanguage'});
// DETECTAR PALABRAS CLAVE mediante expresiones regulares
monaco.languages.setMonarchTokensProvider('thumderLanguage', {
  ignoreCase: true, /* <-- la expresión regular no permite /.../i    */
  tokenizer:  {
    root: [ /*EXPRESIONES REGULARES --> TOKEN*/ ]
  }
});
// MARCADO DE PALABRAS CLAVE en su color correspondiente a partir de las expresiones regulares anteriores
monaco.editor.defineTheme('thumderTheme', {
    // ...
    rules:                [ /* REGLAS - {color, token} */ ]
});
// AUTO COMPLETADO a partir de palabras clave en un ejemplo de la instrucción
monaco.languages.registerCompletionItemProvider('thumderLanguage', {
  provideCompletionItems: (model, position, context, token) => {
    // ...
  }
});
// MUESTRA DOCUMENTACIÓN de la instrucción en la que se posa el ratón, se muestra un dialogo con el texto
monaco.languages.registerHoverProvider('thumderLanguage', {
  provideHover(model: monaco.editor.ITextModel, position: monaco.Position, token: monaco.CancellationToken)
              : monaco.languages.ProviderResult<monaco.languages.Hover> {
    // ....
  }
});
```

6.8.4. Memoria

La vista de la memoria está compuesta por dos estructuras principales, la tabla de visualización y por el editor, la tabla que usamos para mostrar los datos es generada mediante la clase `TableVirtualScrollDataSource` de la librería `ng-table-virtual-scroll`, esta tabla nos permite usar tablas dinámicas que mejoran el rendimiento de nuestra aplicación al mostrar únicamente los elementos que se requieren, esto se complementa con ciertos cambios en la representación de los datos con el uso de `<ng-container>` para las condiciones y `Pipes` de *THUMDER*, esto nos permite mostrar los datos en el formato requerido.

Esto se hace en repetidas ocasiones para cambiar el formato de representación.

```
1  <!-- IF () -->
2  <ng-container matColumnDef="Binary">
3    <th mat-header-cell *matHeaderCellDef class="width-40">{{'MACHINE.BINARY' |
         translate}}</th>
4    <td mat-cell *matCellDef="let␣index" class="width-40">
5      <p>{{ (machine.memory.getMemoryWordBinaryByIndex(index)) }}</p>
6    </td>
7  </ng-container>
8  <!-- ELSE IF () -->
9  <ng-container matColumnDef="BinaryFloat">
10   <th mat-header-cell *matHeaderCellDef class="width-40">{{'MACHINE.BINARY_FLOAT' |
         translate}}</th>
11   <td mat-cell *matCellDef="let␣index" class="width-40">
12     <p>{{ (machine.memory.getMemoryWordBinaryByIndex(index)) }}</p>
13   </td>
14 </ng-container>
15 <!-- ELSE IF () -->

17 ...
```

Código de ejemplo 6.12 memory.component.html

Por último, tenemos la sección del editor de la memoria, este componente es un elemento modal adaptado a la tarea de actualizar la memoria de la máquina, este componente cuenta con dos campos editables, el primero selecciona la dirección de memoria, el segundo el valor.

Internamente, usamos los métodos *Getter* y *Setter* [46] para controlar y gestionar la actualización, ya que a la hora de actualizar solo hay definido un atributo privado tipo `string` que representa una cadena binaria.

Los métodos encargados de actualizar los valores internos serán los métodos *Getter* y *Setter*, los métodos `changeAddressMemoryToEdit` y `changeMemory` son asíncronos, ya que son los encargados de actualizar en el servidor.

Por último, para mostrar la cadena binaria y sus segmentos se ha creado el método `drawDigitsToChangeByTypeData`, esto nos permite actualizar un componente dinámicamente e incrustarlo en una vista `html`.

`valueInSection_Byte`	Modifica de forma interna la memoria en formato *Byte*.
`valueInSection_HalfWord`	Modifica de forma interna la memoria en formato *HalfWord*.
`valueInSection_Word`	Modifica de forma interna la memoria en formato *Word*.
`valueInSection_Float_Binary32_IEEE754`	Modifica de forma interna la memoria en formato *IEEE754 32 bits*.
`valueInSection_Double_Binary64_IEEE754`	Modifica de forma interna la memoria en formato *IEEE754 64 bits*.
`changeTypeData(...)`	Cambiar el tipo formato de dato a actualizar.
`changeAddressMemoryToEdit(...)`	Cambiar la dirección de memoria.
`changeMemory(...)`	Cambia y actualiza en el servidor la memoria.

Tabla. 6.3: Métodos para actualizar la memoria

Para ello se usa la estructura de `Angular SafeHtml`, esto nos permite mezclar estructuras de `Angular` con nuestra programación y mantenerla unidas mediante la propiedad de enlace (*binding* [47]).

6.8.5. Registros

La vista de los registros no es simple, pues depende de la simulación, esta sección del proyecto es crucial, es por ello que está fuertemente ligado al estado de la máquina y de la simulación.

Para su implementación se han agrupado los registros en cuatro tipos de categorías, las que se definen en DLX, estos son los registros de control, enteros y decimales de punto flotante de simple precisión (32 bits) y de doble precisión (64 bits).

Es por ello que la vista de registros tiene cuatro secciones que usan los componentes (`cards`), uno para cada grupo de registros, usamos las mismas estructuras de tablas dinámicas que hemos usado anteriormente en la vista de la memoria, (`TableVirtualScrollDataSource`), también se usan los mismos `pipes` para dar formato a los datos, así como `pipes` para las traducciones de cada cabecera de la tabla.

Código de ejemplo 6.13 registers.component.ts

```
1  public typeDataSelected: TypeData = "Byte";
2  public typeDataSelectedFloat: "Binary" | "Uint8Array" = "Binary";
3  public typeDataSelectedDouble: "Binary" | "Uint8Array" = "Binary";
4  public displayedColumns = [ "Register", "Hexadecimal", "Binary", "Byte" ];
5  public dataSource = new TableVirtualScrollDataSource<string>(MACHINE_REGISTERS_C as string[]);
6  public dataSourceR = new TableVirtualScrollDataSource<number>(MACHINE_REGISTERS_R as number[]);
7  public dataSourceF = new TableVirtualScrollDataSource<number>(MACHINE_REGISTERS_F as number[]);
8  public dataSourceD = new TableVirtualScrollDataSource<number>(MACHINE_REGISTERS_D as number[]);
9  public displayedColumnsR = [ "Register", "Hexadecimal", "Binary", "Integer" ];
10 public displayedColumnsF = [ "Register", "Hexadecimal", "Binary" /*Binary or Uint8Array*/, "Float" ];
11 public displayedColumnsD = [ "Register", "Hexadecimal", "Binary" /*Binary or Uint8Array*/, "Double" ];
```

Por último, explicar el editor de registros, al igual que la vista de memoria, esta sección es un formulario *modal*, esta es una de las características más importantes y complejas, pues requiere poder editar estructuras de tipo binario en múltiples formatos, entre estos formatos, el formato IEEE 754 [48] en 32 o 64 bits, debemos poder dar valor a un registro anteriormente seleccionado.

Los métodos para actualizar los registros son los siguientes.

`registerToEdit_Binary`	Método `Get` y `Set` del registro.
`registerToEdit_Word`	Método `Get` y `Set` del registro.
`registerToEdit_Hexadecimal`	Método `Get` y `Set` del registro.
`registerToEdit_Float`	Método `Get` y `Set` del registro.
`registerToEdit_Double`	Método `Get` y `Set` del registro.
`changeTypeRegister(...)`	Cambia el tipo de registro a editar y actualiza la lista de registros editables.
`changeRegisterToEdit(...)`	Selección del de registro a editar.
`changeRegisterToEditValue(...)`	Cambia el valor del registro y actualiza el registro en el servidor.

Tabla. 6.4: Métodos para actualizar los registros

6.8.6. Código

La vista de código máquina, es la que nos permite interpretar las instrucciones que están en la memoria, además muestran las instrucciones que están en el cauce de ejecución (*Pipeline*), esta sección se puede confundir con la vista de la memoria, pero es distinta, pues incluye dos intérpretes, uno propio dentro del proyecto y otro que es el intérprete del servidor.

La tabla de la sección muestra el estado del cauce mediante el uso de clases de HTML y CSS, para ello se selecciona la fila de la instrucción mediante una subscripción que notifica que dirección de memoria está en ejecución.

El funcionamiento de la sección es simple, se muestra una lista de direcciones activas, si en la tabla dinámica (`TableVirtualScrollDataSource`) hay alguna de estas direcciones de memoria, entonces se activa la clase CSS que representa la etapa en el cauce.

```
<tr mat-row *matRowDef="let␣row;␣columns:␣displayedColumnsMemory;"
    [class.stage-IF]="checkIfContains(row.address,␣['IF'])"
    [class.stage-ID]="checkIfContains(row.address,␣['ID'])"
    [class.stage-intEX]="checkIfContains(row.address,␣['intEX',␣'faddEX',␣'fmultEX',
        ␣'fdivEX'])"
    [class.stage-MEM]="checkIfContains(row.address,␣['MEM'])"
    [class.stage-WB]="checkIfContains(row.address,␣['WB'])"></tr>
```

Código de ejemplo 6.14 code.view.html

Código de ejemplo 6.15 code.view.ts

```
private setRow(index: number, tableCode: TypeInstructionsData, stage: TypeStage = ""): void {
  this.dataSourceCode.data[index] = {
    address:     tableCode.address,
    instruction: tableCode.instruction ?? Utils.convertHexCodeToTextMachineInstructionDLX(tableCode.code),
    // ...
  };
  this.refresh();
}
```

Además, como en el resto de representaciones de la memoria se ha añadido un intérprete propio que lee la memoria y genera la instrucción, este intérprete no está completo, ya que no analiza las etiquetas (*tags*) para los saltos, por lo que deja la instrucción con una etiqueta (*tag*) predeterminada, el correcto funcionamiento corresponde al servidor.

El método *convertHexCodeToTextMachineInstructionDLX(...)* es el encargado de representar una instrucción en caso de que no se reciba una instrucción del servidor.

Código de ejemplo 6.16 code.view.ts

```
this.codeSimulationSubscription = this.machine.getCodeSimulationObservable().subscribe((typeTableCode) => {
  for (const data_code of typeTableCode) {
    const index = Math.round(Utils.hexadecimalToDecimal(data_code.address) / 4);
    this.setRow(index, data_code);
  }
});
```

6.8.7. Diagrama del reloj (Cycle clock diagram)

Para el desarrollo de esta sección se ha usado un elemento *Canvas* de HTML 5 y el motor de gráficos PixiJS, se ha desarrollado la clase controladora de la tabla (`PixiTHUMDER_CycleClockDiagram.ts`), esta clase nos permite dibujar y representar el historial del cauce de ejecución, esto lo hacemos mediante el uso de tres tablas (`PixiTHUMDER_Table.ts`), la tabla de pasos (eje horizontal), la tabla de instrucciones (eje vertical) y la tabla del cauce (representación), por último, se ha creado la clase `PixiTHUMDER_CycleClockDiagram.ts` esta es la encargada de crear, editar y eliminar celdas, además de gestionar el desplazamiento de la cámara, fijando los limites para su correcta visualización y desplazamiento.

Internamente, se tiene un registro del número de instrucciones, el número de flechas y el ciclo en ejecución, esto es así para controlar la liberación de memoria y lo que se muestra en el elemento *Canvas*.

La simulación en el diagrama de ciclos se ha organizado de una forma simple, por filas las instrucciones y por columnas los pasos, pues en cada ciclo de ejecución el sistema obtiene del servidor el cauce de ejecución, por lo solo hace falta representar el cauce en la tabla usando su dirección de memoria y paso correspondiente.

Cada etapa en el cauce de ciclos se representa mediante un color representativo, por lo que el diagrama de ciclos se mantiene esta representación.

Usamos el `loop` del motor de PixiJS para el desplazamiento de la tabla, actualizamos los elementos y su posición siempre con ciertas condiciones.

Código de ejemplo 6.17 pixi-cycle-clock-diagram.component.ts

```
ngOnInit(): void {
  this.stepSimulationSubscription = this.machine.getStepSimulationObservable().subscribe((stepSimulation) => {
    if (stepSimulation.isNewInstruction === true) {
      this.machine.cycleClockDiagram.addInstruction(
          this.machine.code.getOrDefaultValue(stepSimulation.pipeline.IF.address).instruction
      );
    }
    for (const arrow of stepSimulation.pipeline.arrows) {
      // ...
      const color = parseInt(String(arrow.color), 16);
      this.machine.cycleClockDiagram.addArrow(arrowDraw, color);
    }
    this.machine.cycleClockDiagram.nextStep(stepSimulation.pipeline, stepSimulation.step);
  });
}

private gameLoop(delta: number): void {
  this.play(delta);
  this.keyboard.update();
}

private play(_delta: number): void {
  if (this.keyboard.isKeyDown("ArrowLeft", "KeyJ")) {
    this.machine.cycleClockDiagram.moveRight();
  }
  // ...
}

// ...
```

Se han implementado muchas características adicionales para que la aplicación funcione correctamente, entre ellas redimensionar el elemento *Canvas* y mantener las proporciones, liberar memoria cuando no está en uso, optimización de los fotogramas por segundo (*FPS*) en el *Canvas*.

6.8.8. Cauce de ejecución (Pipeline)

Al igual que la sección del diagrama de ciclos, esta sección se ha realizado mediante un elemento *Canvas* de HTML 5 y el motor gráfico PixiJS [49], en esta sección se ha representado de la forma más clara el cauce de ejecución y cómo esté evoluciona en cada paso, para ello se han creado una caja por cada etapa en el cauce y tres grupos de cajas que representan las unidades de cálculo, en estas cajas se representa la dirección de memoria como su instrucción.

La clase gestora de este elemento es `PixiTHUMDER_Pipeline.ts`, aunque a diferencia `PixiTHUMDER_CycleClockDiagram.ts` la simulación del cauce de ejecución (*Pipeline*) no se controla completamente internamente, ya que en cada paso se debe

recalcular la instrucción que está en el cauce, esto sería sencillo si se mostrara la dirección de memoria, pero debemos mostrar la instrucción que se ejecuta.

Para procesar esta información en cada etapa de forma eficiente se ha creado el tipo de dato `TypePipelineInstructions`, este tipo de estructuras nos permite separar la lógica de representación de los datos del intérprete que nos envía el servidor.

Código de ejemplo 6.18 pixi-pipeline.component.ts

```
export type TypePipelineInstructions = {
  IF:       { text: string; draw: boolean | TypeStall; };
  // ...
  faddEX:   { unit: number; text: string; draw: boolean | TypeStall; }[]
  // ...
};

this.stepSimulationSubscription = this.machine.getStepSimulationObservable().subscribe((stepSimulation) => {

  const instructions: TypePipelineInstructions = {
    IF:       this.getInstructionItem(stepSimulation.pipeline.IF),
    // ...
    faddEX:   this.getInstruction(stepSimulation.pipeline.faddEX),
    // ...
  };

  this.pipeline.processStep(instructions);
});
// ...
private getInstructionItem(item: TypeCycleCell) {
  return {
    text: this.machine.getCode(item.address).instruction,
    draw: item.draw
  };
}
private getInstruction(item: TypeCycleCellUnit[]) {
  return item.map((v) => {
    return {
      unit: v.unit,
      text: this.machine.getCode(v.address).instruction,
      draw: v.draw
    };
  });
}
```

Para la creación de esta interfaz se han creado distintos contenedores que representan las etapas, estos contenedores están enlazados mediante flechas que representan el cauce.

6.8.9. Documentación

La documentación dentro del proyecto se gestiona mediante los documentos *markdown*, aunque estos documentos no son propios del proyecto, sino que son documentos que están en un submódulo de `git`, para usarlos dentro del proyecto debemos

hacer unos movimientos de ficheros, para no tener estos ficheros repetidos y tener que actualizarlos usamos las herramientas de Angular para copiar ficheros según ciertos patrones.

Esto crear una entrada virtual de los ficheros, que al construir el proyecto se copian en la carpeta indicada y podemos acceder a ellos en función de su ruta relativa.

Código de ejemplo 6.19 angular.json

```
"assets": [
  "src/assets",
  {
    "glob": "*.md",
    "input": "./wiki",
    "output": "./assets/wiki/"
  },

// ...
]
```

Código de ejemplo 6.20 docs.view.ts

```
public main_list: TypeIdTitleFile[] = [{
    id:    "README",
    title: "README",
    file:  "assets/docs.md"
  }, {
    id:    "Tabla",
    title: "Tabla DLX - OPCODES",
    file:  "assets/md/DLX-TABLE-Instructions.md"
  },
  // ...
];
```

El siguiente código es muy simple, en primer lugar, tenemos un `*ngFor` que realiza un bucle de `main_list`, `cdkDrag` le da la propiedad para ser movido el elemento en el navegador.

Por último, se hace una llamada, una plantilla HTML de Angular, esto se hace mediante `<ng-template #AccordionCard let-item='item'>`, donde `<ng-template>` es la definición de la plantilla, `#AccordionCard` es el nombre de la plantilla y `let-item='item'` son los parámetros.

```
<div *ngFor="let␣document␣of␣main_list" cdkDrag>
  <div class="example-custom-placeholder" *cdkDragPlaceholder></div>
  <ng-container [ngTemplateOutlet]="AccordionCard"
                [ngTemplateOutletContext]="{item:␣document}"></ng-container>
</div>


<ng-template #AccordionCard let-item='item'>
  <div class="card␣card-primary␣card-outline">

    <div class="card-header">
      <h4 class="card-title">
        <a class="d-block␣w-100" data-toggle="collapse" href="#collapse_{{item.id}}"
           >
          {{ item.title }}
        </a>
      </h4>
      <div class="card-tools">
        <button type="button" class="btn␣btn-tool" cdkDragHandle>
          <i class="fas␣fa-arrows-alt"></i>
        </button>
      </div>
    </div>

    <div id="collapse_{{item.id}}" class="collapse␣" data-parent="#accordion">
      <div class="card-body">
        <markdown [src]="item.file" (load)="onMarkdownLoad()"></markdown>
      </div>
    </div>

  </div>
</ng-template>
```

Código de ejemplo 6.21 docs.view.html

6.8.10. Información aplicación

En esta sección se han añadido cinco elementos con la información más relevante de la aplicación, los documentos `markdown` que tienen la documentación sobre el desarrollo, las políticas de las *cookies*, licencias, información de la app, el historial de cambios (*changelog*) y el listado de dependencias.

La representación de los documentos `markdown` la hacemos mediante la librería `ngx-markdown`, aunque hemos modificado ciertos comportamientos para adaptarnos al entorno de `Electron`, de esta forma abrimos los enlaces a web externas en el navegador predeterminado en caso de estar en la aplicación de escritorio.

```
<markdown [src]="'assets/md/README.md'"
    (load)="onMarkdownLoad('markdownComponentID_README')"
    #markdownComponentID_README></markdown>
```

Código de ejemplo 6.22 about.view.html 1

Por último, las dependencias no son un documento `markdown`, para su visualización se ha usado una `API` externa (`unpkg.com`), mediante el uso de nuestro `package.json` leemos las dependencias de nuestro proyecto y las dependencias de desarrollo y realizamos la consulta para obtener los datos.

Usando el sistema de promesas asíncronas de `JavaScript` (`async / await`) creamos una lista de promesas (consultas a la API) que se resuelven de forma paralela para mayor eficiencia, esto se logra mediante el método `Promise.all(...)`.

Código de ejemplo 6.23 about.view.ts 2

```
private async prepareData() {
  let dependenciesData_Promises: Promise<IPackageJson>[] = [];
  for (const dependency of this.dependencies) {
    dependenciesData_Promises.push(this.queryNPMPackage(dependency[0], dependency[1]));
  }
  this.dependenciesData = await Promise.all(dependenciesData_Promises);
  // ...
}
private async queryNPMPackage(package_name: string, version: string): Promise<IPackageJson> {
  return new Promise((resolve, reject) => {
    const QUERY = this.SERVER_API + package_name + "@" + version + "/package.json";
    this.httpClient.get<IPackageJson>(QUERY).toPromise().then((response) => {
      resolve(JSON.parse(JSON.stringify(response)));
    })
  })
}
```

La `API` nos proporciona directamente los ficheros `package.json` de las dependencias, nosotros simplemente procesamos los datos necesarios.

```
<ng-container *ngFor="let data of this.dependenciesData">
  <tr>
    <td>
        <ng-container *ngIf="data?.homepage">
          <a (click)="openInExternal(data?.homepage)">{{data?.name}}</a>
        </ng-container>
        <ng-container *ngIf="!data?.homepage">
          {{data?.name}}
        </ng-container>
    </td>
    <td>{{data?.version}}</td>
    <td>{{data?.license}}</td>
    <td>{{data?.author?.name}}</td>
    <td>{{data?.description}}</td>
  </tr>
</ng-container>
```

Código de ejemplo 6.24 about.view.html 2

6.8.11. Calculadora

La calculadora se gestiona mediante un formulario reactivo, cada representación del dato está vinculado a su tipo de dato, por lo que al actualizar el valor se notifica al resto de campos mediante las herramientas de los formularios reactivos de Angular, como son `[ngModelOptions]`. `[(ngModel)]`.

Se han usado los métodos *Getter* y *Setter* de JavaScript para simplificar las actualizaciones.

Código de ejemplo 6.25 calculator.view.ts

```
// Byte
private _valueByte: Uint8Array = new Uint8Array([ 0 ]);
get binary08_Value(): number
set binary08_Value(value: number)
get binary08_Hexadecimal(): string
set binary08_Hexadecimal(value: string)
get binary08_Binary(): string
set binary08_Binary(binary: string)

// HalfWord, Word Uint, Word int, IEEE754 32 bits IEEE754 64 bits
// ...
```

El formulario se distribuye en filas los tipos de datos y en columnas los distintos tipos de representaciones (decimal, hexadecimal y binario).

```
<div class="col-3">
  <div class="form-group">
    <label for="byte_value">{{ 'TYPE_DATA.BYTE' | translate }}</label>
    <input type="number" class="form-control" id="byte_value" step="1"
         [ngModelOptions]="{standalone: true}"
         [(ngModel)]="binary08_Value">
  </div>
</div>


<!-- ... -->

<div class="col-3">
  <div class="form-group">
    <label for="ieee754_number">{{ 'TYPE_DATA.IEEE754' | translate }}</label>
    <input type="number" class="form-control" id="ieee754_number" step="0.1"
         [ngModelOptions]="{standalone: true}"
         [(ngModel)]="binary32_IEEE754_Value">
  </div>
</div>
```

Código de ejemplo 6.26 calculator.view.html

6.9. Dependencias para la aplicación

Para el desarrollo del proyecto se han desarrollado dos sub proyectos de forma paralela, el primero fue un proyecto web simple con la interfaz del diagrama de ciclos de reloj y el cauce de ejecución del DLX, el segundo sub proyecto fue el servidor como intérprete de la máquina.

6.9.1. Cliente

Para el desarrollo del proyecto se han creado otros sub proyectos para hacer pruebas y comprobar la compatibilidad con los distintos navegadores, no todos estos proyectos se han usado al final, algunos se descartaron por motivos de rendimiento o comportamientos anómalos, ese fue el caso de las pruebas con la librería P5.js o Three.js.

Hay muchas librerías de gráficos 2D como P5.js, PixiJS, Three.js, se usó finalmente el proyecto PixiJS principalmente porque era una librería que ya conocía y porque

implementarla junto a Angular es mucho más sencillo y existen muchos proyectos de referencia que usan ambas tecnologías.

6.9.1.1. Sub proyecto - THUMDER-pixi

Este sub proyecto se ha desarrollado en un repositorio separado de Angular para que su desarrollo fuera más rápido y luego integrarlo dentro del proyecto principal. https://github.com/nonodev96/THUMDER-pixi [37]

En el desarrollado del sub proyecto hemos usado WebPack como servidor con recarga caliente (*hot reload*), usando PixiJS se han implementado las funcionalidades y vistas de las distintas partes del DLX como es el cauce de ejecución o el diagrama de ciclos de reloj.

Parece un sub proyecto simple, aunque su desarrollo ha sido complejo, ya que para una buena interpretación de las vistas hay que distribuir la interfaz de forma correcta, hacer cálculos exactos con la separación en píxeles de cada elemento, usar ecuaciones para dibujar flechas de distintos tamaños y ángulos y que estas flechas se representen de forma adecuada.

Todo esto ha llevado mucho más tiempo del que se tenía pensado en su planificación.

6.9.2. Servidor

El servidor es el agente encargado de interpretar un fichero (programa DLX), gestionar una máquina virtual, gestionar la memoria y los registros de la máquina y realizar simulaciones de dicha máquina.

Para poder comunicar nuestro proyecto con el servidor se ha de desarrollar un protocolo simple que nos permita comunicarnos entre el cliente y el servidor, una vez que ya se comunican podemos realizar ciertas pruebas.

1. Enviar, interpretar, simular y ejecutar un fichero de DLX
2. Dividir una ejecución de DLX
3. Actualizar registros y memoria
4. Crear y destruir una máquina

6.9.2.1. Sub proyecto - THUMDER-server

Para realizar el desarrollo del proyecto de la forma más fiable se optó por realizar una implementación de las secciones más cruciales del servidor, aunque el servidor de pruebas desarrollado no está completo y no realiza simulaciones de todos los casos que se pueden dar en un programa DLX, si es lo suficientemente bueno para ciertos casos y pequeñas simulaciones controladas, estas simulaciones comprenden gran parte del conjunto de instrucciones.

La comunicación se realiza mediante un protocolo, este estará definido mediante los `@Types` de TypeScript a modo de objetos que envían y responden con información sobre la simulación.

La idea de este servidor de pruebas es usarlo como un servidor para hacer pruebas con `mocks`.

Para este sub proyecto se ha implementado un servidor de pruebas y se han desarrollado una serie de pruebas de integridad que en función de sus *mocks* y *stubs* [50] debe devolver exactamente un objeto tipo `JSON` con los datos (memoria, registros, código y cauce de ejecución) correspondientes a su programa DLX.

https://github.com/nonodev96/THUMDER/tree/dev/UML [36]

6.10. Simulador

El *runner* se le ha considerado a una serie de clases, métodos y estructuras de datos que permiten gestionar la ejecución del código ensamblador del DLX en el simulador, el proceso que se sigue la simulación es simple:

1. Primero se debe cargar un fichero en el editor de código `monaco` [41].
2. Segundo se escribe y se modifica el código en el editor.
3. En la barra de navegación (*navbar*) se encuentran las herramientas de simulación, al darle a *play*, *next* o *execute* se lanza la siguiente petición:
   - 3.1 Se envía al servidor el código ensamblador cumpliendo el protocolo. [51]
   - 3.2 Se procesa el código en el servidor y se envía una respuesta de vuelta cumpliendo el protocolo. [51]

3.3 Se recibe la respuesta en el cliente mediante el *Socket* y se procesa mediante la clase de `MachineService`, esta proporciona los métodos y estructuras de datos e interfaces para la lectura e interpretación del código.

4. La respuesta se procesa y se carga en un objeto JSON que tiene una lista con las distintas etapas y estados que debe representar.

En esta respuesta se encuentra el estado de los registros, la memoria, las instrucciones a representar en el `cycle-clock-diagram` y para el `pipeline`, así como los adelantamientos en cada etapa.

Código de ejemplo 6.27 MachineService.ts

```ts
// ...
this.timerObserver = {
  next: async (_: number): Promise<void> => {
    if (this.isRunning === true) {
      await this.SimulationNextStep();
    }
    return Promise.resolve();
  }
};
// ...

private async SimulationNextStep(): Promise<void> {
  try {
    const payload = JSON.stringify({
      step: this.privateStep + 1,
      // ...
    });
    // ...
    this.socketProviderConnect.emitMessage("SimulationNextStepRequest", payload, async (response) => {
      this.statusMachineInStep = JSON.parse(response ?? DEFAULT_STEP_SIMULATION) as TypeSimulationStep;
      const canNextInstruction = await this.CheckConditions();
      if (canNextInstruction) {
        // ...
        await this.ProcessStep();
        // ...
      }
    });
    return Promise.resolve();
  } catch (error) {
    console.error(error);
    return Promise.reject(error.message);
  }
}
```

6.11. Publicación y compilación

Una vez desarrollado el proyecto debemos poder publicarlo, para esta tarea usamos GitHub Actions que nos permite compilar un proyecto multiplataforma como es Electron mediante NPM [52], para esto hemos creado el script `create-release.yaml` definido en la carpeta `.github/`.

Hacer un despliegue en nuestro ordenador de desarrollo es distinto de usar GitHub Actions, ya que estamos limitados al sistema operativo de nuestro ordenador.

A continuación, se explica el *script* que se ha creado para el despliegue (`create-release.yaml`).

Esta tecnología funciona en modo *script*, por lo que para usar ciertas herramientas debemos importarlas, para ello usamos la palabra reservada `uses: <name>`, con esto vamos cargando los distintos módulos.

Para iniciar esta tarea debemos subir un *commit* con una etiqueta en cierto formato, **v<A.B.C>**, por ejemplo, **v1.1.5**, de esta forma podemos publicar una versión para Windows, Linux y Mac.

El *script* creado para esta tarea sigue la siguiente estructura y pasos.

1. Se instala Node.js y NPM.
2. Se lee el fichero `package.json`.
3. Se crea la entrada de publicación (*release*) con la etiqueta correspondiente.
4. Por cada sistema operativo (Windows, Linux, Mac) creamos una máquina virtual que creará el fichero ejecutable para su sistema operativo correspondiente.
   4.1 Se clonan los sub módulos de Git del proyecto.
   4.2 Instalamos Node.js y NPM.
   4.3 Se lee el fichero `package.json`.
   4.4 Se escribe en el proyecto las variables de entorno definidas como secretos en GitHub.
   4.5 Se instalan las dependencias del proyecto.
   4.6 Se construye el fichero binario del proyecto (Electron).
   4.7 Se publica el fichero construido (*release*).

Por último, se crea una versión de lanzamiento (*release*) en GitHub con la etiqueta de versión que hemos definido en el *commit* y esta se publica con las distintas versiones generadas para cada sistema operativo.

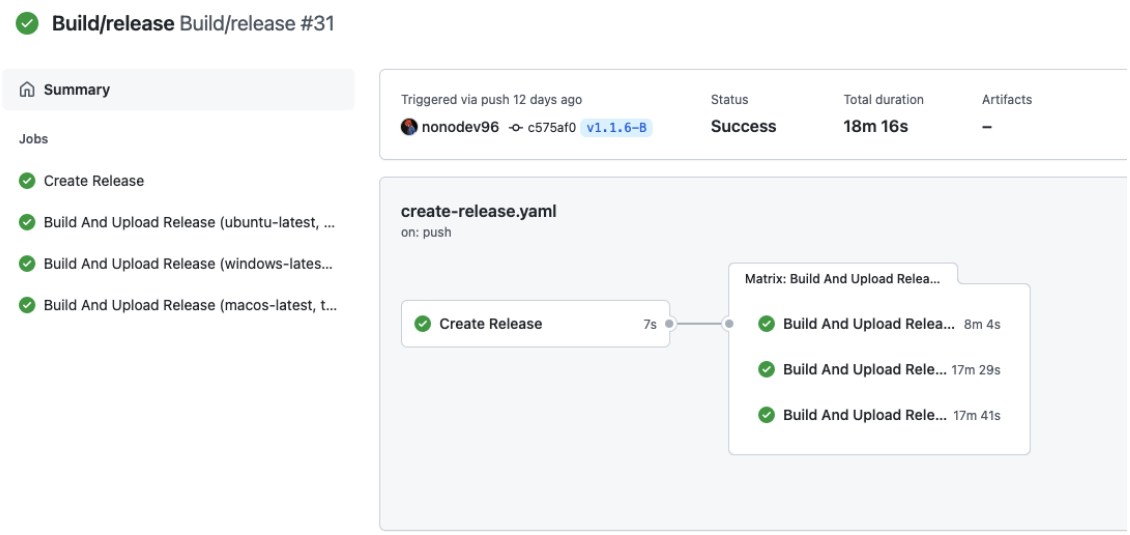


Figura 6.1: Creación de la versión final mediante GitHub Actions

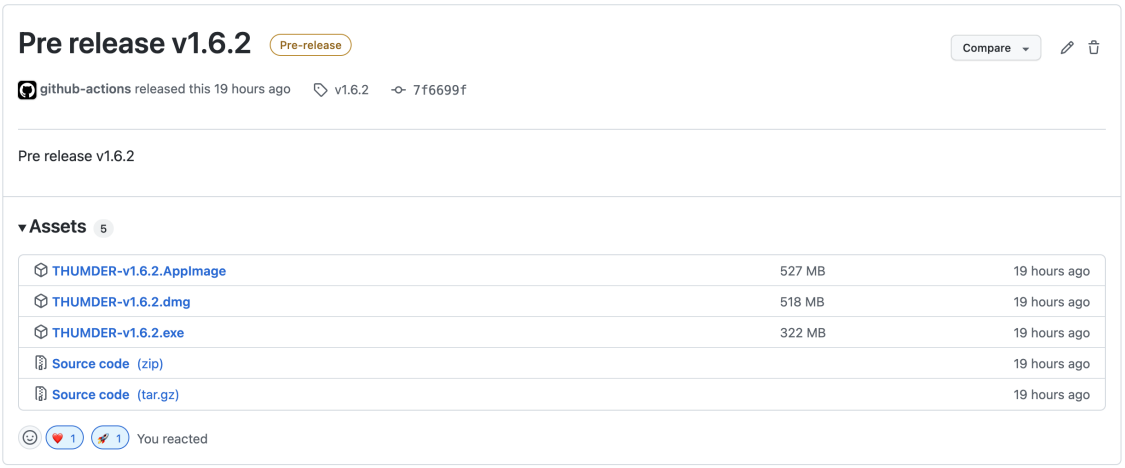


Figura 6.2: Publicación de la versión final mediante GitHub Actions

Capítulo 7

Pruebas

7.1. Pruebas unitarias y pruebas de interfaz

Para la validación de los datos se han desarrollado distintos tipos de pruebas en los dos proyectos.

En el proyecto principal se han desarrollado las pruebas con Cypress de forma que podamos visualizar las pruebas, ver el flujo de ejecución, ver los fallos en la interfaz.

En la parte del servidor del proyecto se ha usado Jest y el desarrollo se ha guiado mediante TDD de TypeScript, se han implementado las pruebas más relevantes para validar los datos del intérprete del servidor.

7.1.1. THUMDER

Para las pruebas en el proyecto se han creado los ficheros `cypress.json` donde se definen las variables del navegador y rutas de carpetas del proyecto de pruebas y el fichero `cypress.env.json` donde se definen las variables globales del proyecto de pruebas.

Una vez definidas estas dos secciones debemos desarrollar las pruebas que el proyecto requiere, para ello se han implementado distintas funciones auxiliares que automatizan ciertas pruebas, esto se realiza siguiendo el patrón que sugiere Cypress, en el fichero `support/index.ts` se han definido los comandos de Cypress, estas fun-

ciones están implementadas dentro de `support/commands/*`, para esto usamos el `namespace` de Cypress.

Se han creado comandos especiales para iniciar sesión, cerrar sesión, cambiar el idioma, moverse a ciertas rutas, etc.

Listado de ficheros con pruebas.

1. `app.spec.ts`: comprueba que la aplicación es alcanzable y que tiene el título esperado.
2. `auth.spec.ts`: comprueba que el usuario puede iniciar sesión en la aplicación.
3. `calculator.spec.ts`: comprueba que la lógica de la calculadora es válida, se realizan conversiones y validaciones.
4. `cookies.spec.ts`: comprueba que las *cookies* de la aplicación funcionan correctamente.
5. `lang.spec.ts`: comprueba que las traducciones son válidas y se aplican correctamente.
6. `cycle-clock.spec.ts`: comprueba el estado del *Canvas* y que esté mostrándose correctamente, comprueba la resolución y los objetos en el diagrama de ciclos.
7. `pipeline.spec.ts`: comprueba el estado del *Canvas* y que esté mostrándose correctamente, comprueba la resolución y los objetos en el cauce de ejecución.
8. `memory.spec.ts`: comprueba la memoria en la tabla y la actualización de la memoria en función del formulario.
9. `registers.spec.ts`: comprueba los registros en las tablas y la actualización de estos en función del formulario.
10. `navigate.spec.ts`: comprueba que las rutas son alcanzables si el usuario ha iniciado sesión.

Se han implementado un total de 37 pruebas para validar el funcionamiento de la aplicación.

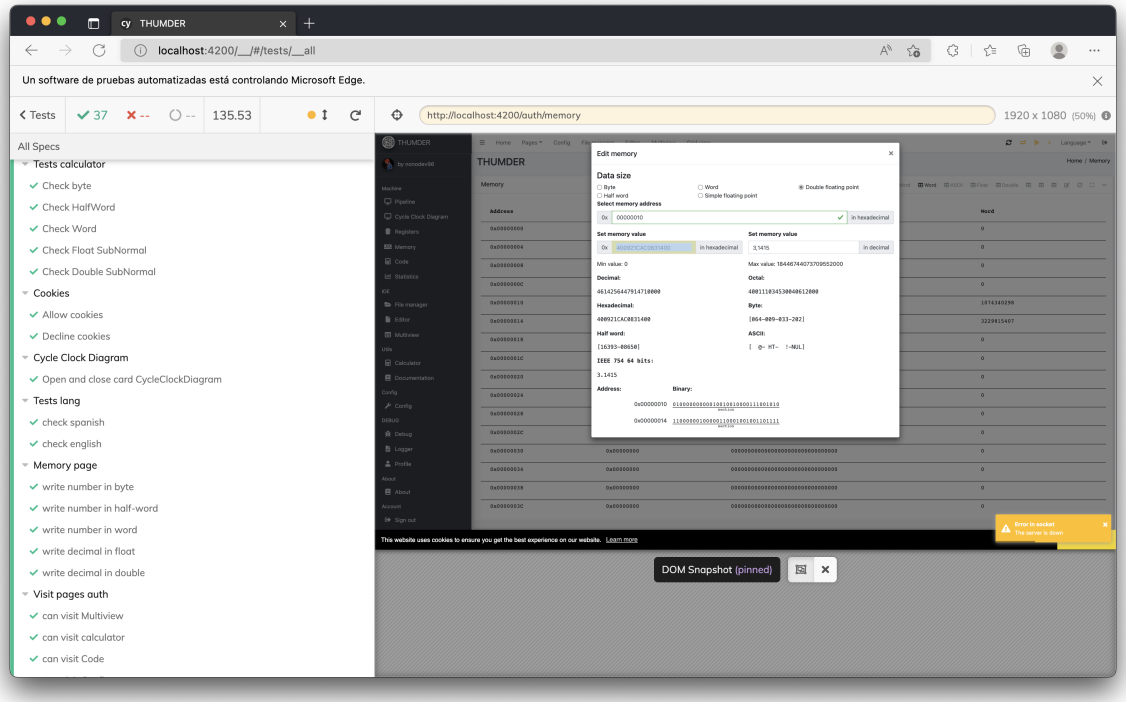


Figura 7.1: Pruebas de interfaz *Cypress*

7.1.2. THUMDER-server

Las pruebas del servidor se realizan con Jest, esta librería posee un superconjunto de utilidades que nos permite validar decenas de casos y tipos abstractos, también se ha usado Socket.io para la simulación de las conexiones con los clientes mediante las conexiones de tipo canal abierto (`WebSocket`).

A grandes rasgos, cada vez que se ejecuta una prueba se crea y destruye la conexión con el servidor, eliminando en el proceso la simulación y la máquina creada.

Los datos de validación se han realizado ejecutando simulaciones en **WinDLX** [4] y creando pruebas con mocks y stubs [50] a partir de los datos de la simulación, esto nos permiten validar la simulación que realiza nuestro servidor, otra de las pruebas más relevantes del servidor han sido los métodos para actualizar la memoria, los registros o la configuración de la máquina, por otra parte, se validan los métodos que pide al servidor la memoria, los registros o las estadísticas.

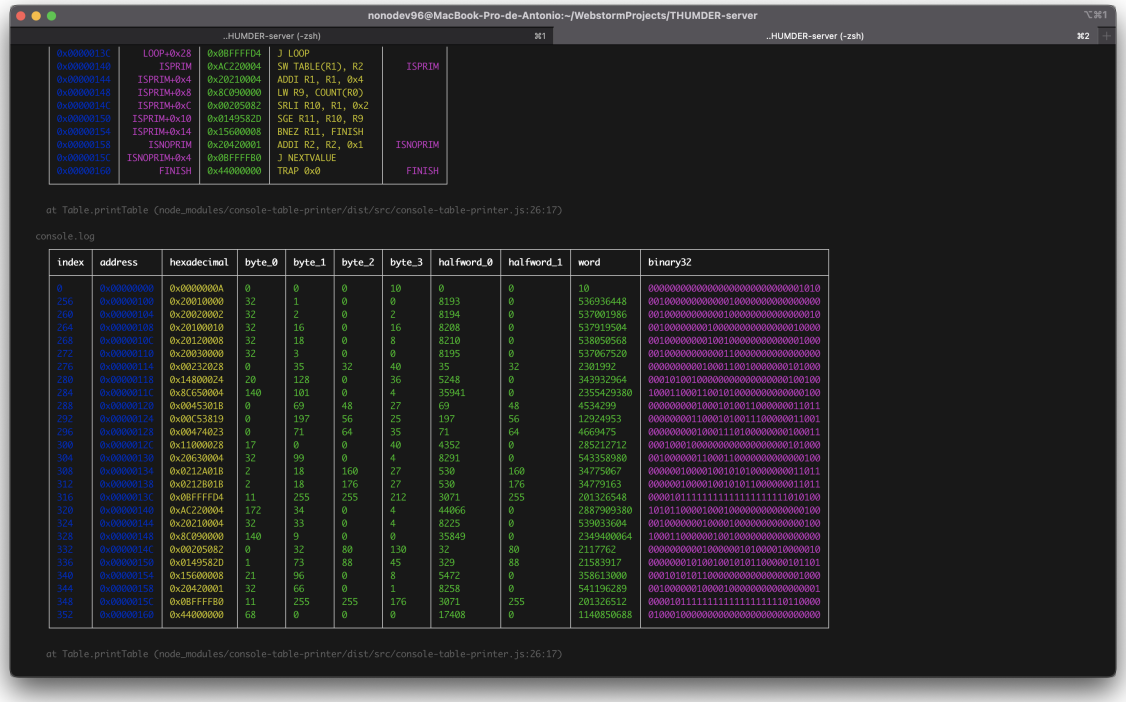


Figura 7.2: Pruebas del servidor de pruebas (*Tests*)

La cobertura de estas pruebas alcanza el 78 %, analizando los datos se ve como parte del código que no se valida por nuestras pruebas, es el que notifica los errores o fallos del intérprete.

1. *Stmts* las declaraciones del programa

2. *Branch* se refiere a cada rama (también denominada ruta DD) de cada estructura de control (`if`, `switch`, `breaks`, etc.). Es una forma de medir el análisis de los límites

3. *Funcs* llamado de las funciones o subrutinas

4. *Lines* se refiere a cada línea ejecutable en el archivo fuente

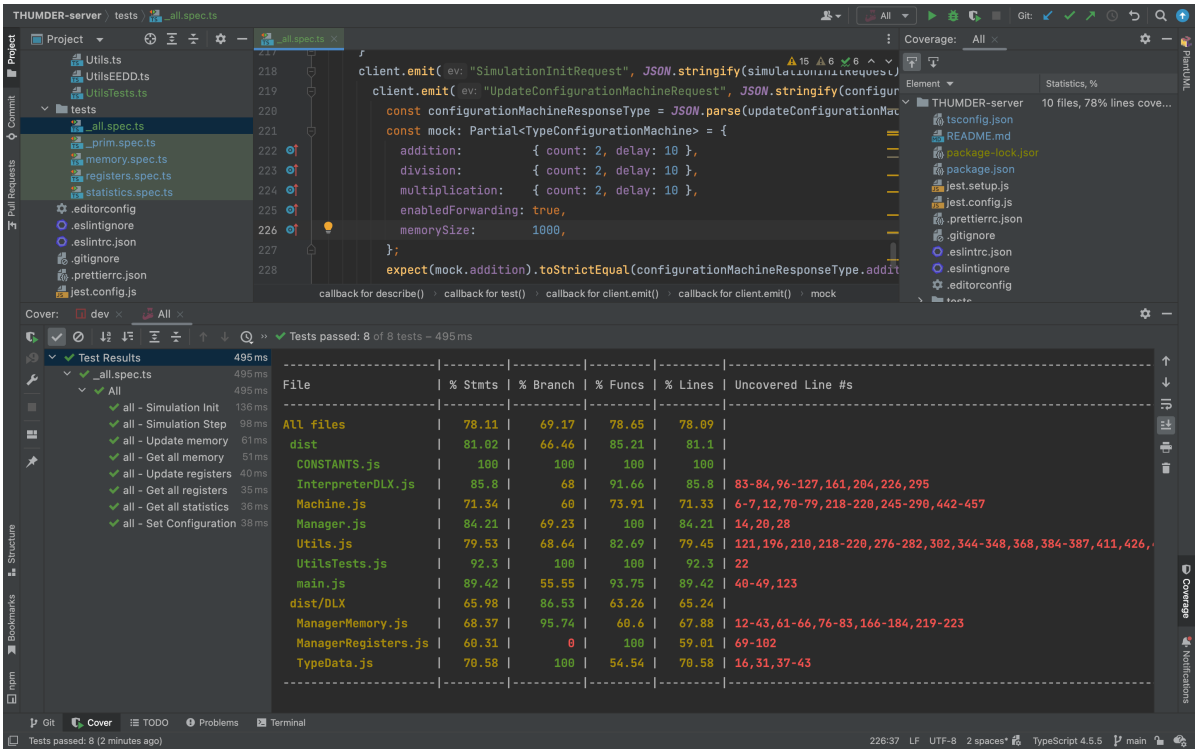


Figura 7.3: Pruebas unitarias en Jest con WebStorm

Capítulo 8

Conclusiones y trabajos futuros

Como conclusión final para este proyecto he de mencionar qué es lo que más he aprendido y en qué he mejorado.

Principalmente, este proyecto me ha servido para aprender nuevas arquitecturas para organizar el código del proyecto, metodologías para el desarrollo *software* y procesos para redactar documentación de una forma eficiente.

También he aprendido mucho sobre la forma correcta de organizar proyectos con Node y NPM. También he aprendido sobre pruebas unitarias y pruebas de interfaz para este tipo de proyectos, así como la importancia de estas pruebas. He mejorado mucho en el manejo de lenguajes de programación como TypeScript o en programación funcional.

Por último, mencionar las posibles mejoras del proyecto que por motivos de tiempo no se han podido desarrollar.

1. Actualizador automático.
2. Vistas desplegables en Electron o en el navegador.
3. Múltiples tipos de simuladores para no estar limitados al DLX.
4. Editor de código en tiempo real colaborativo (*Live share*).
5. Verificación mediante un *hash* para validar la información de la simulación.

6. Barra de progreso de la ejecución (*Timeline*), para simulaciones finitas.

7. Mejoras en la ventana I/O

Capítulo 9

Manuales

9.1. Arquitectura DLX

Para comprender este proyecto es necesario entender los conceptos generales de la arquitectura DLX, es por ello que definiremos el funcionamiento de dicha arquitectura, cuáles son sus propiedades y cuáles los elementos más importantes.

Este apartado es un resumen muy breve de los apartados de DLX del libro “Arquitectura de computadores - Un enfoque cuantitativo” [3] y del manual DLX de la Universidad de Valencia [53].

El DLX es un microprocesador de tipo RISC desarrollado por John L. Hennessy.

La arquitectura DLX solo cuenta con las primitivas más frecuentes utilizadas en los programas.

- Cuenta con un repertorio de instrucciones para cargar datos y guardar datos en la memoria.
- Su segmentación está pensada y diseñada de forma eficiente.
- Cuenta con repertorio de instrucciones que son simples de decodificar por su formato.
- Los compiladores para la arquitectura DLX son eficientes.

Pensado y diseñado como un buen modelo arquitectónico para su análisis y estudio, es fácil de comprender, se utiliza como recurso didáctico para comenzar a entender los principios de la arquitectura de computadores.

9.1.1. Componentes de la arquitectura DLX

Componentes de la arquitectura DLX [3]:

- DLX cuenta 32 registros de propósito general (**GPRs**), 32 registros de punto flotante (**FPRs**) de simple precisión, 32 *bits* cada uno, que pueden usarse como 16 registros de doble precisión (**DPR**) si se usan una pareja de FPR. Se pueden realizar operaciones de simple o doble precisión, hay un conjunto de registros especiales utilizados para acceder a la información sobre el estado del registro de estado **FP** (Punto flotante) se utiliza para comparaciones y excepciones. Todas las transferencias al y del registro de estado se realizan a través de los registros de propósito general (GPR), hay un salto que examina el bit de comparación del registro de estado de FP.

- La memoria es direccionable por *bytes* en el modo "Big Endian" con una dirección de 32 *bits*. Los accesos que involucran a los GPR pueden realizarse a un byte, media palabra a una palabra. Los FPR se pueden cargar y almacenar con palabras en simple o doble precisión. Todos los accesos a memoria deben estar alineados.

- Hay instrucciones para transferencia entre datos, el tipo FPR y GPR.

- Todas las instrucciones son de 32 *bits* y deben estar alineadas.

- Hay también unos pocos registros especiales que pueden transferir a y desde registros enteros. Entre ellos, el registro de estado de punto flotante que se utiliza para almacenar la información sobre los resultados de las operaciones en punto flotante.

9.1.2. Operaciones de la arquitectura DLX

DLX cuenta con cuatro tipos de instrucciones:

1. Instrucciones de tipo cargas y almacenamientos.

2. Instrucciones con operaciones de tipo ALU.

3. Instrucciones para saltos y bifurcaciones.

4. Instrucciones con operaciones en coma flotante [48].

Abreviatura	Representación	Traducción
L	Load	Cargar
S	Save	Almacenar
B	Byte	Byte
H	Half Word	Media palabra
W	Word	Palabra
F	Float	Flotante
D	Double	Doble
U	Unsigned	Sin signo
SP	Simple precision	Simple precisión
DP	Double precision	Doble precisión

Tabla. 9.1: Alias DLX

9.1.2.1. Transferencia de datos, carga y almacenamiento

Todos los registros de propósito general o de coma flotante se pueden usar para cargar o almacenar datos, excepto R0 que siempre estará a cero.

Solamente hay un modo único de direccionamiento, registro base sumando el desplazamiento de 16 bits con su signo.

Las cargas de media palabra (*halfword*) y byte colocan el valor en la parte inferior del registro y su mitad superior se rellena con el signo del valor o ceros.

Los números en coma flotante de simple precisión ocupan (32 bits) mientras que los números decimales de doble precisión ocupan un par (64 bits). Las conversiones entre simple y doble precisión deben realizarse aplicando la norma o estándar técnico IEEE754 [48].

Las instrucciones que operan con la ALU son instrucciones que realizan operaciones registro a registro. Esto significa que realizan operaciones aritméticas sencillas (sumar, restar, multiplicar o dividir) y operaciones lógicas (AND, OR, XOR) o desplazamiento de bits SHIFTS.

La operación LHI o carga superior inmediata, carga la mitad superior de los bits de un registro, a continuación, pone a cero (**0**) la mitad de los bits inferiores. Con dos instrucciones, se crea una constante del tamaño de una palabra (32 bits).

Cuenta con operaciones de transferencia de datos entre registros y la memoria, entre registros enteros, registros de coma flotante o registros especiales.

Instrucciones Código OP	**Significado**
LB, LBU, SB	Carga byte, carga byte sin signo, almacena byte
LH, LHU, SH	Carga media palabra, carga media palabra sin signo, almacena media palabra
LW, SW	Carga palabra, almacena palabra (a/desde registros enteros)
LF, LD, SF, SD	Carga coma flotante simple precisión, carga coma flotante doble precisión, almacena coma flotante simple precisión, almacena coma flotante doble precisión
MOVI2S, MOVS21	Transfiere desde/a GPR a/desde un registro especial
MOVF, MOVD	Copia un registro de coma flotante o un par en doble precisión en otro registro o par de registros
MOVFP21, MOVILFP	Transfiere 32 bits desde los registros FP al desde registros enteros

Tabla. 9.2: Transferencia de datos

9.1.2.2. Aritmético y lógicas

Las operaciones que realiza la ALU son del tipo registro-registro. Estas operaciones suelen ser operaciones aritméticas tales como sumas, restas u operaciones lógicas tales como *and*, *or*, *xor*, desplazamientos, etc.

Se incluyen instrucciones de tipo carga y almacenamiento de tipo inmediato de media palabra.

También contamos con operaciones de comparación, estas instrucciones comparan dos registros, son los operadores lógicos (!=, ==, <, >, >=, <=), si los registros cumplen la condición entonces pone un uno (**1**) sobre el registro de destino.

Lista de las instrucciones que realizan operaciones aritméticas, lógicas, cargas, etc.

Instrucciones Código OP	Significado
ADD, ADDI, ADDU, ADDUI	Suma, suma inmediato, todas las instrucciones inmediatas son de 16 bits con signo y sin signo
SUB, SUBI, SUBU, SUBUI	Resta, resta inmediata con signo y sin signo
MULT, MULTU, DIV, DIVU	Multiplica y divide, con signo y sin signo; los operandos deben ser registro de tipo entero; todas las operaciones tienen valores de 32 bits
AND, ANDI	And, and inmediato
OR, ORI, XOR, XORI	Or, or inmediato, or exclusiva, or exclusiva inmediata
LHI	Carga del dato inmediato superior
SLL, SRL, SRA, SLLI, SRLI, SRAI	Desplazamientos: inmediatos (S-I) y forma variable (S-), son desplazamientos lógicos a la izquierda, lógicos a la derecha, aritméticos a la derecha, etc.
SLT, SGT, SLE, SGE, SEQ, SNE	Inicialización condicional
SLTI, SGTI, SLEI, SGEI, SEQI, SNEI	Inicialización condicional inmediata

Tabla. 9.3: Aritmética y Lógica

9.1.2.3. Control de flujo, saltos y bifurcaciones

Hay cuatro instrucciones de salto, se diferencian por la forma de especificar la dirección destino y si se hace o no se hace un enlace. Dos instrucciones utilizan el contador de programa para indicar de forma relativa el desplazamiento, eso utiliza 26 bits; las otras dos instrucciones de salto especifican un registro que contiene la dirección de salto. Hay dos formas de salto: salto plano y salto con enlace (llamadas a procedimientos). El último tipo de salto almacena la dirección de retorno en el registro *R31*.

Las bifurcaciones son operaciones condicionales. La operación condicional se especifica en la instrucción, la cual puede comprobar si el registro fuente es o no es **0**; el registro fuente puede ser el resultado de una comparación.

Lista de instrucciones con saltos y bifurcaciones condicionales; relativos al PC o mediante registros.

Instrucciones Código OP	Significado
BEQZ, BNEZ	Salto GPR igual/no igual a cero; desplazamiento de 16 bits desde PC+4
BFPT, BFPF	Prueba de bit de comparación en el registro FP y el desplazamiento de 16 bits desde el PC más 4
J, JR	Bifurcaciones: desplazamiento de 26 bits desde el PC (J) o destino en registro (JR)
JAL, JALR	Bifurcación y enlace: guarda el PC más 4 en el registro R31, el destino es relativo al PC (JAL) o un registro (JALR)
TRAP	Transfiere a sistema operativo a una dirección vectorizada
RFE	Volver al código del usuario desde una excepción, restaurar modo de usuario

Tabla. 9.4: Control

9.1.2.4. Coma flotante

Operaciones con valores numéricos decimales usan las unidades de tipo flotante, estas operaciones pueden ser de simple (32 bits) o doble precisión (64 bits). Debemos recordar que las operaciones de simple precisión usan cualquier registro de tipo flotante, mientras que las de doble precisión debían usar un par (ejemplo: F2-F3). Las operaciones de carga y almacenamiento funcionan de una forma similar, simplemente copian el dato desde la memoria al registro, tanto para simple como para doble precisión.

Se incluyen instrucciones que realizan operaciones de multiplicación, divisiones enteras de que operan sobre registros de coma flotante, además de otras instrucciones de conversión entre tipos de datos.

Lista de instrucciones con operaciones en coma flotante en formatos de simple precisión y doble precisión.

Instrucciones Código OP	Significado
ADDD, ADDF	Suma números DP, SP
SUBD, SUBF	Resta números DP, SP
MULTD, MULTF	Multiplicación coma flotante DP, SP
DIVD, DIVF	Divide coma flotante DP, SP
CVTFZD, CVTFZI, CVTDZF, CVTDZI, CVTIZF, CVTIZD	Transforma instrucciones: CVT**xZy** convierte de tipo x a y. **X** e **Y**: (1: Entero, F: Simple precisión o D: Doble precisión). Ambos operandos están en los registros de coma flotante
LTF, GTF, LEF, EQF, NEF	Compara SP, pone bit de comparación en registro de estado FP
LTD, GTD, LED, EQD, NED	Compara DP, pone bit de comparación en registro de estado FP

Tabla. 9.5: Coma Flotante

9.1.3. Estructura de las instrucciones

Todas las instrucciones se almacenan en la memoria en grupos de 32 *bits*, de esos 32 *bits*, 6 *bits* están dedicados al código de operación, estos códigos de operación se organizan en tres tipos de formatos. Los formatos I y R son muy similares

1. Instrucción tipo I 9.1.3.1.
2. Instrucción tipo R 9.1.3.2.
3. Instrucción tipo J 9.1.3.3.

9.1.3.1. Instrucción tipo I

Orden de los *bits*:

6 bits	5 bits	5 bits	16 bits
Código OP	RS	RT	Inmediato

Figura 9.1: Representación de las instrucciones de tipo I

1. 6 *bits* indican el código de operación OP.
2. 5 *bits* indican el registro índice RS.
3. 5 *bits* indican el registro de destino u origen de la información RT.

4. 16 *bits* indican el dato inmediato o dirección Dirección.

9.1.3.2. Instrucción tipo R

Orden de los *bits*:

6 bits	5 bits	5 bits	5 bits	11 bits
Código OP	RS	RT	RD	Func

Figura 9.2: Representación de las instrucciones de tipo R

1. 6 *bits* indican el código de operación OP.
2. 5 *bits* indican el primer registro fuente RS.
3. 5 *bits* indican el segundo registro fuente RT.
4. 5 *bits* indican el registro destino RD.
5. 11 *bits* indican el desplazamiento (5 *bits*) Shamt y la operación (6 *bits*) Funct.

9.1.3.3. Instrucción tipo J

Orden de los *bits*:

6 bits	26 bits
Código OP	Desplazamiento añadido al PC

Figura 9.3: Representación de las instrucciones de tipo J

1. 6 *bits* indican el código de operación OP.
2. 26 *bits* indican el desplazamiento añadido al PC Dirección.

9.1.4. Tabla de referencias de operaciones máquina en DLX

Para identificar una instrucción en función del estado de la máquina debemos seguir las siguientes referencias.

Todas las instrucciones en la tabla 9.7 y 9.8 son instrucciones tipo R. En la tabla 9.6 las instrucciones “J, JAL, REF, TRAP” son de tipo J el resto son tipo I.

(bit3-5) / (bit0-2)	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
000	(rr alu)	(float)	J	JAL	BEQZ	BNEZ	BFPT	BFPF
001	ADDI	ADDUI	SUBI	SUBUI	ANDI	ORI	XORI	LHI
010	RFE	TRAP	JR	JALR	SLLI	-	SRLI	SRAI
011	SEQI	SNEI	SLTI	SGTI	SLEI	SGEI	-	-
100	LB	LH	-	LW	LBU	LHU	LF	LD
101	SB	SH	-	SW	-	-	SF	SD
110	-	-	-	-	-	-	-	-
111	-	-	-	-	-	-	-	-

Tabla. 9.6: Tabla DLX Operaciones tipo I y J

En DLX operaciones ALU tipo R con tipo R con (opcode = 0): Solo los 6 bits menos significativos en el campo “func” son usado

(bit29-31) / (bit26-28)	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
000	-	-	-	-	SLL	-	SRL	SRA
001	-	-	-	-	-	-	-	-
010	-	-	SLTU	SGTU	SLEU	SGEU	-	-
011	MULT	MULTU	DIV	DIVU	-	-	-	-
100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	-
101	SEQ	SNE	SLT	SGT	SLE	SGE	-	-
110	MOVI2S	MOVS2I	MOVF	MOVD	MOVFP2I	MOVI2FP	-	-
111	-	-	-	-	-	-	-	-

Tabla. 9.7: Tabla DLX Operaciones tipo R con opcode = 0

En DLX instrucciones de punto flotante con (opcode = 1): Solo los 6 bits menos significativos en el campo “func” son usados.

(bit29-31) / (bit26-28)	000	001	010	011	100	101	110	111
000	ADDF	SUBF	MULTF	DIVF	ADDD	SUBD	MULTD	DIVD
001	CVTF2D	CVTF2I	CVTD2F	CVTD2I	CVTI2F	CVTI2D	-	-
010	EQF	NEF	LTF	GTF	LEF	GEF	-	-
011	EQD	NED	LTD	GTD	LED	GED	-	-
100	-	-	-	-	-	-	-	-
101	-	-	-	-	-	-	-	-
110	-	-	-	-	-	-	-	-
111	-	-	-	-	-	-	-	-

Tabla. 9.8: Tabla DLX Operaciones tipo R con opcode = 1

9.1.5. Modos de direccionamiento

9.1.5.1. Direccionamiento inmediato

El valor a operar está almacenado directamente en la instrucción.

Figura 9.4: Direccionamiento inmediato

9.1.5.2. Direccionamiento por registro

El valor a operar se almacena en memoria y usamos el registro RS para conocer a esa dirección y el valor respectivamente.

Figura 9.5: Direccionamiento por registro

9.1.5.3. Direccionamiento relativo al registro

Este direccionamiento usa la suma de un registro y la dirección (desplazamiento) almacenada en la instrucción para calcular la dirección de memoria donde está el operando.

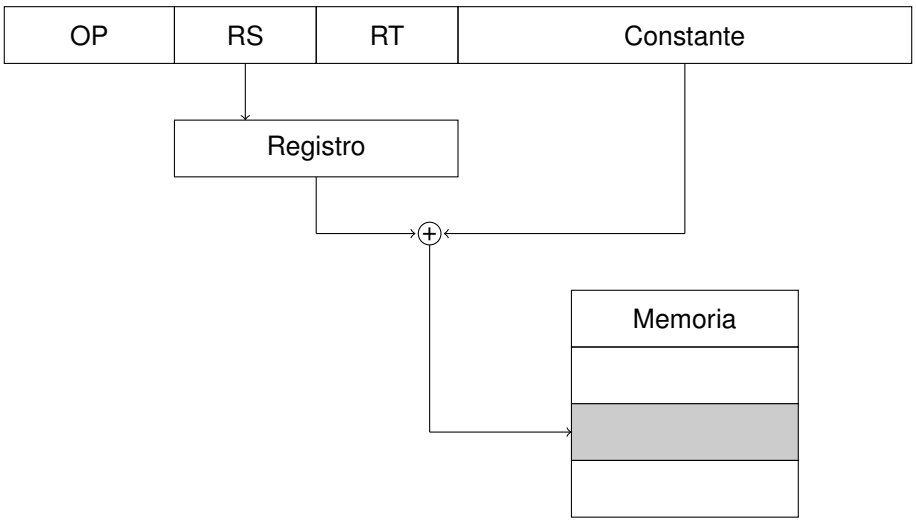


Figura 9.6: Direccionamiento relativo al registro

9.1.5.4. Direccionamiento relativo al PC

La dirección es la suma del Contador de programa (PC) y la constante que está almacenada en la instrucción. El compilador añade 4 al Contador de programa (PC) para traducir la dirección.

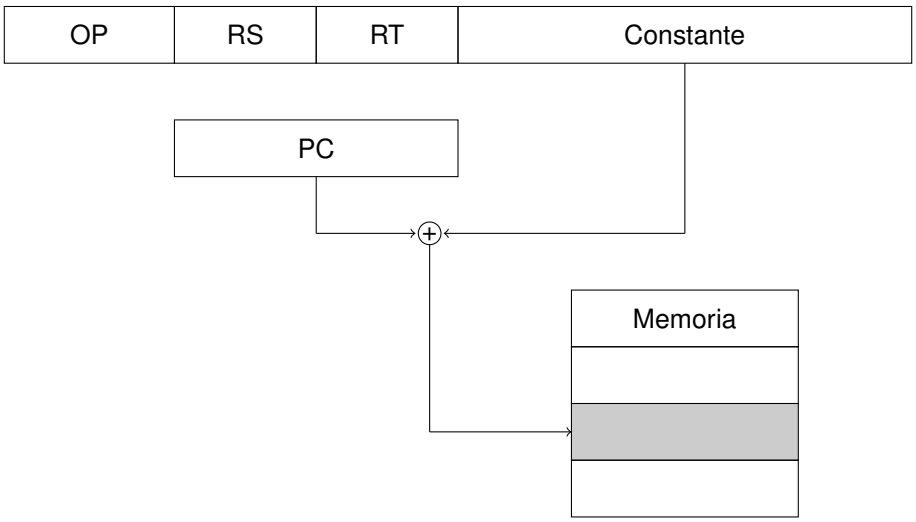


Figura 9.7: Direccionamiento relativo al PC

9.1.6. Programación en ensamblador

Diferenciamos dos tipos claros de división cuando hablamos de programación en ensamblador, el lenguaje ensamblador es el que utiliza símbolos (instrucciones),

mientras que el lenguaje máquina es el equivalente numérico de las instrucciones máquina, para ello usamos los programas denominados como “ensambladores”.

Los programas como WinDLX [4] leen los documentos que terminen en `.s` y si cumplen las normas y formatos correctos, entonces creara un código binario que podrá ser ejecutado.

Algunos de los formatos son los siguientes.

1. Los comentarios son por línea y comienzan con el carácter `;`.
2. Las constantes pueden ser definidas con signo o sin signo.
3. Las etiquetas (*tag*) tienen un patrón o formato definido, este patrón especifica cuáles son las normas de escritura, en el caso de la etiqueta define que deben empezar por un símbolo alfabético, guion bajo o el símbolo dólar, puede contener números, pero no debe ser el primer carácter y siempre ha de terminar con `:`

   3.1 Las etiquetas pueden ser llamadas desde cualquier punto del fichero, y desde otros ficheros cargados posteriormente, siempre que dicha etiqueta se definiera como global (directiva `.global`).

9.1.6.1. Directivas

Las directivas en los lenguajes ensambladores son las encargadas de organizar los datos de inicialización y carga de datos en la memoria, estas suelen estar junto al código del programa.

.align	`n`	Indica que los datos que se indican a continuación se almacenen en la parte más alta de la memoria, con los `n` bits puestos a 0
.ascii	“string1”, “string2”, ...	Cadenas de texto almacenadas, son una lista de caracteres en formato ASCII, las cadenas no terminan en el *byte* `NULL`
.asciiiz	“string1”, “string2”, ...	Actúa de la misma forma que `.ascii`, pero esta directiva sí introduce el *byte* `NULL`
.byte	“byte1”, “byte2”, ...	Introduce en la memoria de forma secuencial los *bytes* definidos en la directiva
.data	`[dirección]`	Indica que los siguientes datos e instrucciones se deben almacenar en la sección `DATA` de la memoria
.double	número 1, número 2, ...	Almacena de forma secuencial la lista de números en la memoria en formato coma flotante de doble precisión
.float	número 1, número 2, ...	Almacena de forma secuencial la lista de números en la memoria en formato coma flotante de simple precisión
.global	`etiqueta`	Hace que la etiqueta sea accesible y referenciada desde otros archivos que se carguen después
.space	*n*	Desplaza el puntero hacia delante tantos *bytes* como indique n
.text	*[dirección]*	Define que el código a continuación ha de ser almacenado en la sección de código desde la dirección indicada
.word	palabra 1, palabra 2, ...	Almacena de forma secuencial las palabras que se indican en la directiva

Tabla. 9.9: Transferencia de datos

9.1.6.2. Llamadas al sistema

DLX emula las llamadas al sistema mediante la instrucción `trap <num>`, las llamadas al sistema usan los registros `R14` y `R1`, en el registro `R14` se almacena la dirección de memoria donde están los argumentos, y en el registro `R1` se almacena el resultado de la operación.

En el caso de error, el código del error (motivo) se almacena en la posición 0 de la memoria.

Código	Error	Descripción
2	ENOFILE, ENOENT	Archivo no encontrado
3	ENOPATH	Ruta no encontrada
4	EMFILE	Demasiados archivos abiertos
5	EACCESS	Acceso denegado
6	EBADF	Manejador de archivo incorrecto
11	EINVFMT	Formato no válido
12	EINVACC	Código de acceso no válido
15	EINVDRV	Unidad de disco incorrecta
18	ENMFILE	No se puede crear o abrir más archivos
19	EINVAL	Argumento no válido
35	EEXIT	El archivo especificado aún existe

Tabla. 9.10: Códigos de error al tratar con archivos

1. **Trap 0**: finaliza el programa sin finalizar las operaciones pendientes.

2. **Trap 1**: abrir el fichero, debe mirar la ruta (cadena de texto `string`) que empieza en la dirección de memoria almacenada en el registro `R14`, los parámetros de acceso son los especificados en la tabla 9.11, mientras que los modos de acceso están especificados en la tabla 9.12. En caso de que la operación se realice de forma exitosa, el manejador se almacena en `R1`, si ha habido un error se almacena -1 en `R1`, el código especificando el error estará en la dirección 0 de la memoria.

   - `manejador = open(camino, acceso [, modo])`

3. **Trap 2**: cierra un archivo asociado a su manejador, en caso de éxito se almacena 0 en `R1`, en caso de error se almacena -1 en `R1` y su código de error en la dirección 0 de memoria.

   - `resultado = close(manejador)`

4. **Trap 3**: lee del archivo asociado al manejador los *bytes* indicados en “número” y los almacena en “destino”. Si el manejador es 0, entonces la entrada es la entrada estándar (I/O). En caso de que no haya errores, en `R1` se almacenara

0 si se ha leído todo el fichero o el número de *bytes* leídos, en caso de error se almacena -1 en `R1` y el código de error en la dirección 0 de la memoria.

- `leídos = read(manejador, destino, número)`

5. **Trap 4**: escribe sobre un archivo asociado a su manejador, escribe en el archivo tantos *bytes* "número" desde la dirección de memoria descrita en "destino". Si hay fallos se almacena -1 en `R1` y el código de error asociado en la dirección 0 de la memoria, en caso contrario se almacena el número de *bytes* escritos.

   - `escritos = write(manejador, destino, número)`

6. **Trap 5**: esta llamada al sistema funciona como la función `printf()` de C99 [54]. Envía la salida formateada a la salida estándar. Si todo es exitoso, se almacena en `R1` el número de *bytes* enviados a la salida estándar, si se produce algún fallo se almacena -1 en `R1` y el código de error en la dirección 0 de la memoria.

   - `resultado = printf(formato [, argumentos, ...])`

7. **Trap 6**: finaliza la ejecución del programa y sus operaciones pendientes.

Los *flags* definidos en las tablas 9.11 y 9.12 están definidos por MS-DOS y no son válidos en UNIX.

Código	Parámetro	Descripción
1	O_RONLY	Abre archivo en modo solo lectura
2	O_WRONLY	Abre archivo en modo solo escritura
4	O_RDWR	Abre archivo en modo lectura y escritura
0x0100	O_CREAT	Crea y abre un archivo
0x0200	O_TRUNC	Abre archivo truncándolo
0x0400	O_EXCL	Abre archivo de forma exclusiva
0x0800	O_APPEND	Añade al final del archivo
0x4000	O_TEXT	Abre archivo en modo texto
0x8000	O_BINARY	Abre archivo en modo binario

Tabla. 9.11: Parámetros de acceso a los archivos

Código	Modo	Descripción
0x0000	S_IFREG	Archivo normal
0X0100	S_IREAD	Propietario puede leer
0X0080	S_IWRITE	Propietario puede escribir
0X0040	S_IEXEC	Permiso de ejecución

Tabla. 9.12: Parámetros modo de acceso a los archivos

9.1.7. Subrutinas

Las subrutinas o etiquetas es una de las formas en las que se organiza el código para poder reutilizarlo, comprender y escalar el código.

DLX cuenta con un conjunto de instrucciones para bifurcar un proceso y volver a su punto anterior, son las instrucciones `jal <tag>` y `jr R31`, estás dos instrucciones funcionan almacenando en `R31` la dirección de memoria por la que estaba el proceso antes de ejecutar `jal`, en caso de querer guardar la dirección de memoria por una interrupción de otro proceso lo correcto es guardar el valor de `R31` en memoria.

DLX tiene como estándar usar los registros de `R4` a `R7` para los argumentos.

Manual proyecto

Este manual es una referencia a la documentación interna del proyecto, si se produce algún cambio en un futuro se notificará en la wiki de github del proyecto [55].

Apéndice A

Instalación

La instalación de un fichero binario de Electron es muy simple, pues se compila como un fichero portable.

Este fichero portable simplemente debemos dejarlo en una carpeta que deseemos y ejecutarlo cuando lo requiramos.

A.1. Windows

La instalación de Windows es simple, pues solo debemos descargar el fichero binario, abrir el fichero y listo, no requiere ningún tipo de configuración previa.

A.2. Linux

La instalación en una distribución Linux requiere que anteriormente le demos permisos al fichero binario, bien mediante la terminal de comandos o por la interfaz de usuario.

```
chmod a+x THUMDER.AppImage
```

A.3. Mac

El fichero en el que se distribuye el binario en macOS es en formato ‘.dmg‘, este formato es similar a un fichero comprimido ‘.zip‘, por lo que debemos extraerlo, macOS por defecto no realiza instalaciones como Windows, sino que estos programas son portables que se dejan en la carpeta ‘~/Applications/‘.

Una vez en esta carpeta, si se ejecuta nos pedirá permisos que deberemos darle desde las preferencias del sistema.

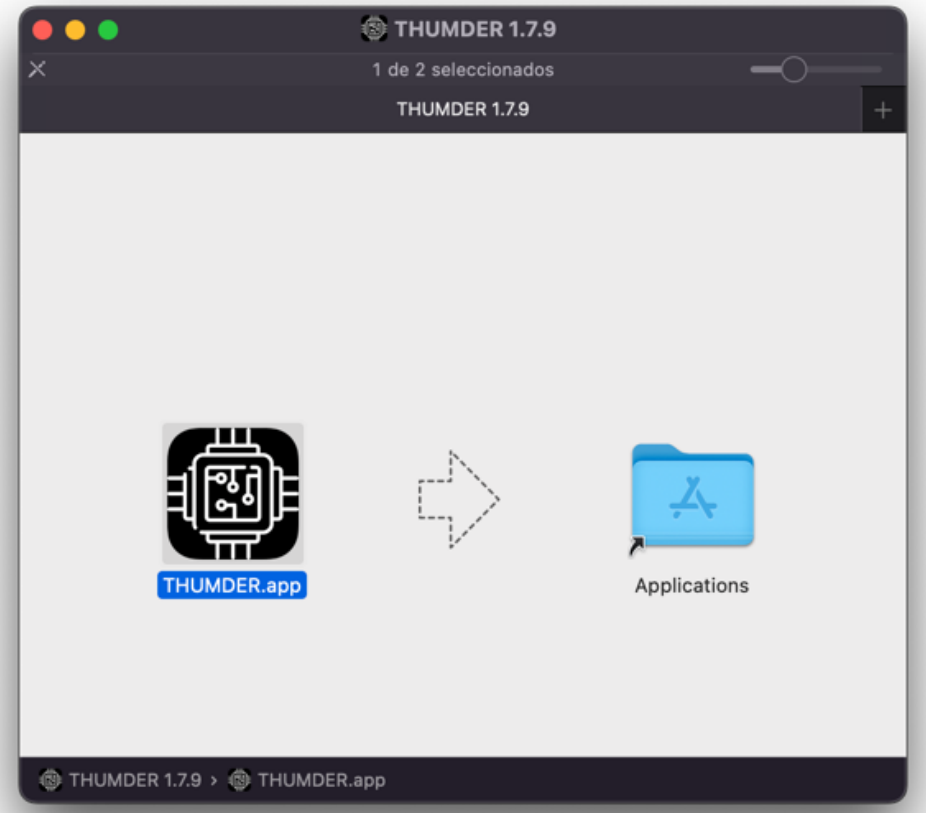


Figura A.1: Instalación fichero DMG

Apéndice B

La ventana de los registros

Esta es una ventana compuesta, es decir una sección que tiene 4 tablas cada una de ellas con conjunto de registros.

Aunque los flotantes de doble precisión usan los de simple precisión se han dividido para que sea más sencilla su lectura.

B.1. Tipos de registros

- Tabla de registros especiales o de propósito general.
- Tabla de registros enteros.
- Tabla de registros decimales, flotantes de simple precisión.
- Tabla de registros decimales, flotantes de doble precisión.

B.2. Representación

En esta ventana se representan los distintos registros agrupados en 4 tablas.

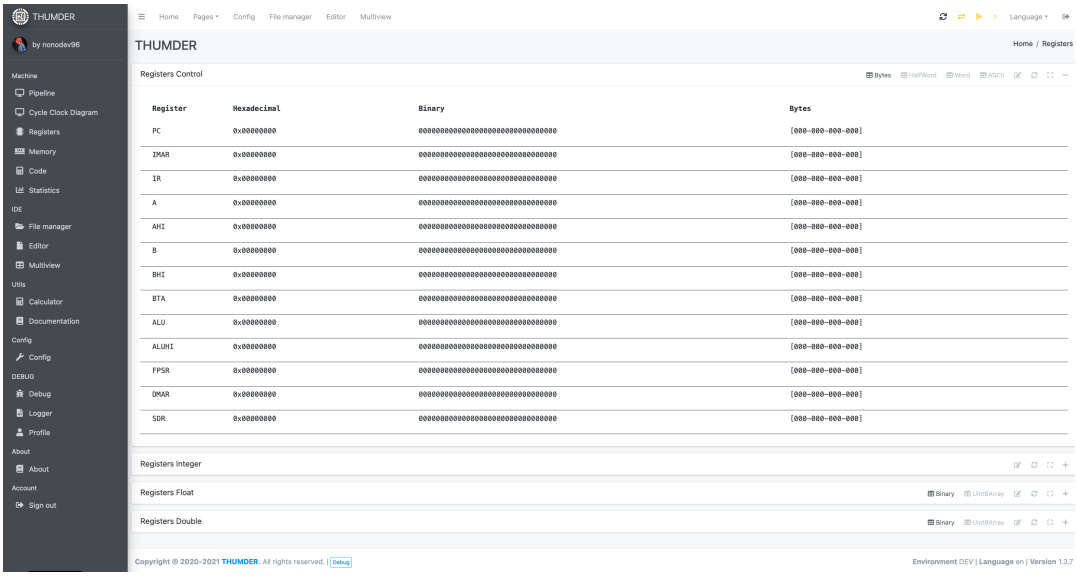


Figura B.1: Representar registros

B.3. Simulación

Durante la simulación se van cambiando los distintos valores de los registros en función de la simulación.

Podemos ver los valores de los registros como el “Program Counter” o “Instruction Memory Address Register” van cambiando en función de como avanza la simulación.

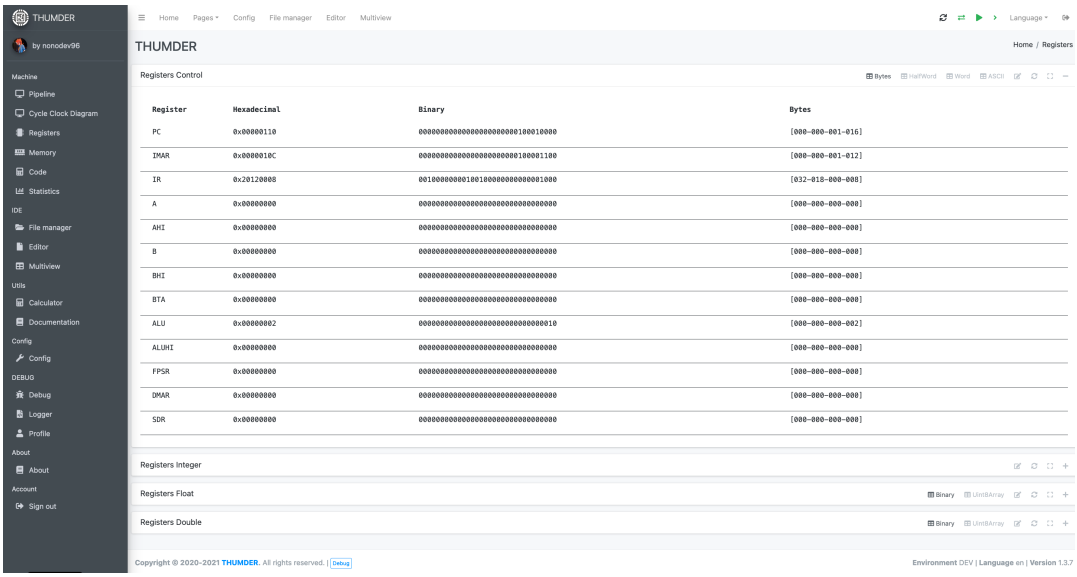


Figura B.2: Representar registros en una simulación

B.4. Editar registros

Para editar los registros abrimos el formulario de edición.

- Seleccionamos de los cuatro tipos el tipo de registro a editar.
- (Se actualiza la lista de registros disponibles)
- Seleccionamos el registro.
- Editamos el valor en formato hexadecimal o en formato decimal.

Esta ventana modal nos permite visualizar de una forma cómoda las distintas representaciones del valor del registro (binario, decimal, octal, hexadecimal, byte, halfword, ASCII o representación IEEE 754 de 32 bits)

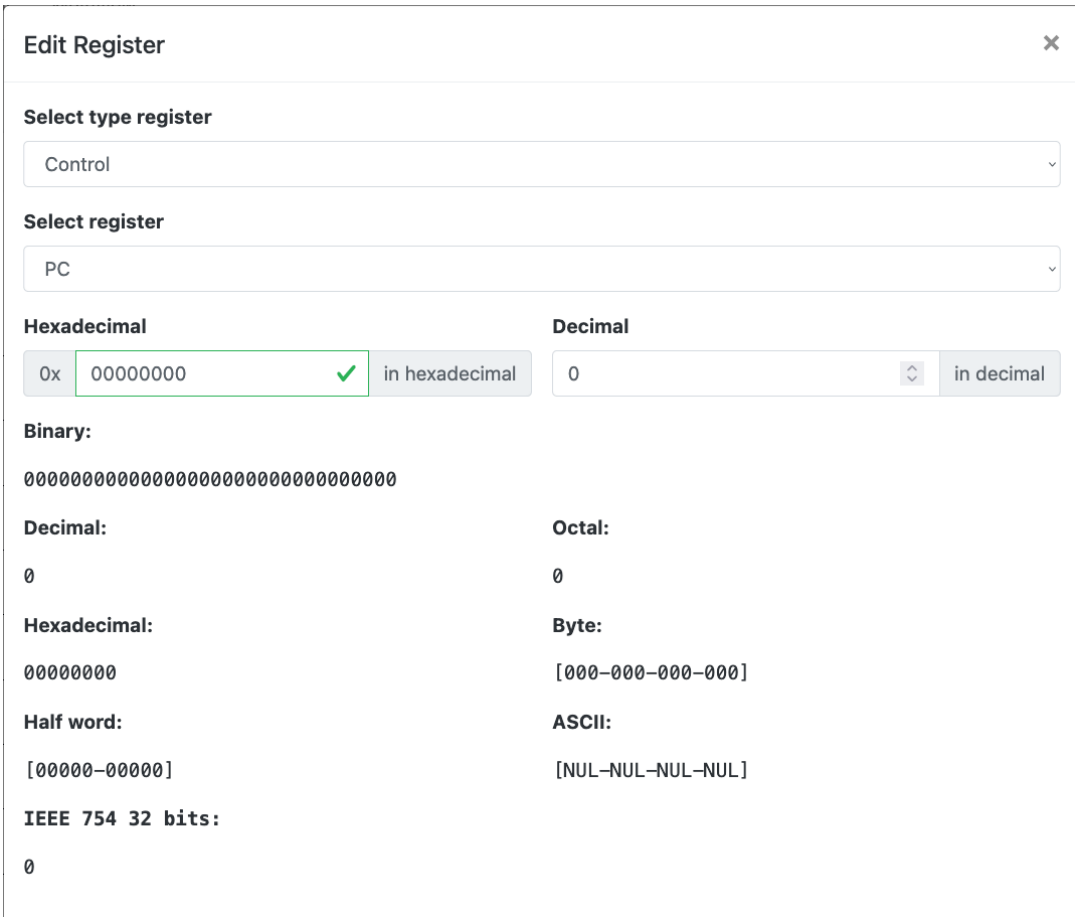


Figura B.3: Editar registros

B.5. Manual DLX

Resumen de los distintos registros de propósito general

- **PSR (Floating-Point Status Register).** Es un registro de estado de 1 bit de longitud, utilizado para comparaciones y excepciones de coma flotante. Todos los movimientos desde y hacia este registro se realizan a través de los registros de propósito general. Las comparaciones en punto flotante asignan el bit de este registro, estando disponibles instrucciones de salto que basan su resultado en el valor del bit (1 true, 0 false).
- **PC (Program Counter).** Siempre contiene la dirección de la próxima instrucción que va a ser ejecutada. Los saltos y las bifurcaciones pueden cambiar el contenido del mismo.
- **IMAR (Instruction Memory Address Register).** Este registro es inicializado con el contenido del contador de programa en la etapa IF a causa de que está conectado con el sistema de memoria, mientras que el PC no.
- **IR (Instruction Register).** En la etapa IF es cargado con la próxima instrucción a ejecutarse.
- **A, B.** Son cargados en la etapa ID y sus valores son enviados a los operandos de la unidad aritmética lógica en la siguiente etapa, la EX. En WinDLX, además existen los pseudo-registros AHI y BHI que contienen los 32 bits superiores para valores en coma flotante de doble precisión.
- **BTA (Branch Target Address).** En la etapa ID, la dirección de salto/bifurcación es calculada y escrita en este registro (ver página 292 del texto base de la asignatura).
- **ALU (Aritmethic Logical Unit).** El resultado de una operación en la ALU es transferido a este registro. En WinDLX existe un pseudo-registro llamado ALUHI que contiene los 32 bits superiores para valores en coma flotante de doble precisión.
- **DMAR (Data Memory Address Register).** La dirección de memoria a la que se va a acceder es transferida a este registro en la etapa EX. En la etapa MEM, el acceso a la memoria para lectura o escritura es efectuado con el valor almacenado en este registro.
- **SDR (Store Data Register).** El dato que se va a escribir en memoria por medio de una instrucción es almacenado previamente en este registro. En WinDLX

existe un pseudo-registro llamado SDRHI que contiene los 32 bits superiores para valores en coma flotante de doble precisión.

- **LDR (Load Data Register).** El dato que es leído de memoria se almacena en este registro. En WinDLX existe un pseudo-registro llamado LDRHI que contiene los 32 bits superiores para valores en coma flotante de doble precisión.

B.6. Desarrollo

Para actualizar el sistema de registros se hace una petición con un vector de objetos de tipo 'TypeRegisterToUpdate', indicando el tipo de registro, el registro en sí y el valor hexadecimal, el servidor responderá con el mensaje 'TypeRegisterToUpdateResponse'.

```
export type TypeRegisterToUpdate = {
  typeRegister:     TypeRegister;
  register:         string; // TypeRegisterControl & number [0-31]
  hexadecimalValue: string; // hexadecimal 0x00000000
};

export type TypeRegisterToUpdateResponse = {
  allOK: boolean,
  registerToUpdates: TypeRegisterToUpdate[]
};
```

Apéndice C

La memoria

Esta es la representación de la memoria de nuestra máquina, es una de las más importantes, ya que contiene todos los datos y todo nuestro código, por ello incluye mucha información y vistas para representar los datos de la forma más sencilla.

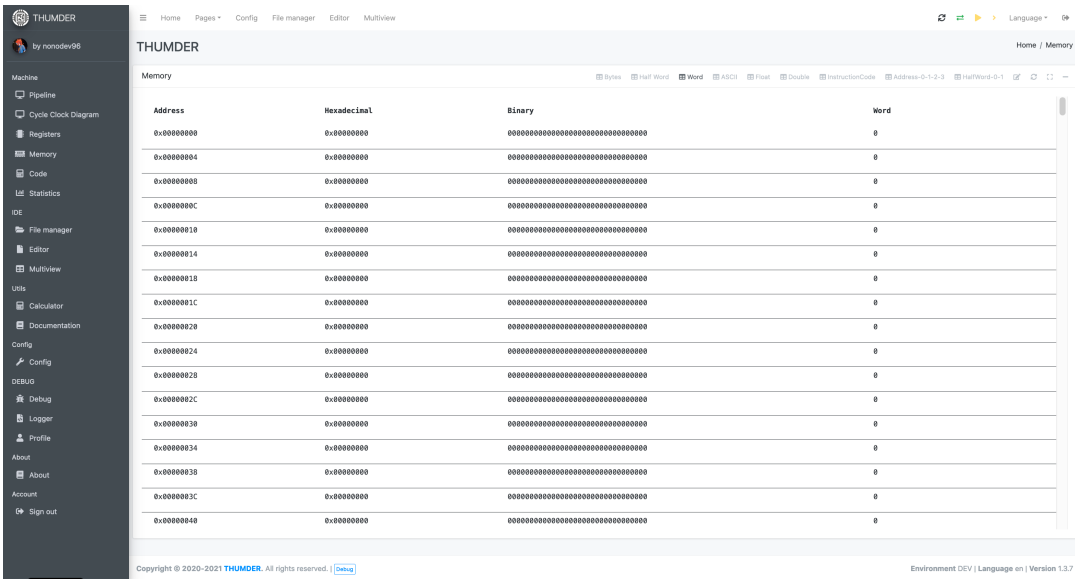


Figura C.1: Ventana de la memoria

C.1. Tipos de datos

Dentro de DLX se ajustan estos cinco tipos de datos (Byte, Half-word, Word, Float y Double), ASCII lo consideramos como especial, ya que es la representación de un Byte.

Type	Bits
Byte	8 bits
Half-word	16 bits
Word	32 bits
Single Floating point	32 bits
Double Floating point	64 bits
ASCII	8 bits

C.2. Editar

Para editar debemos fijar primero el tipo de dato que queremos introducir, por ello se incluye arriba una serie de selectores excluyentes, una vez seleccionado el tipo de dato que deseamos introducir seleccionamos la dirección de memoria para ello introducimos su valor Hexadecimal, se modificarán los valores.

Al cambiar el valor, si el tipo de dato y el valor es válido, entonces se envía este cambio al servidor que actualiza el estado de la máquina en el servidor.

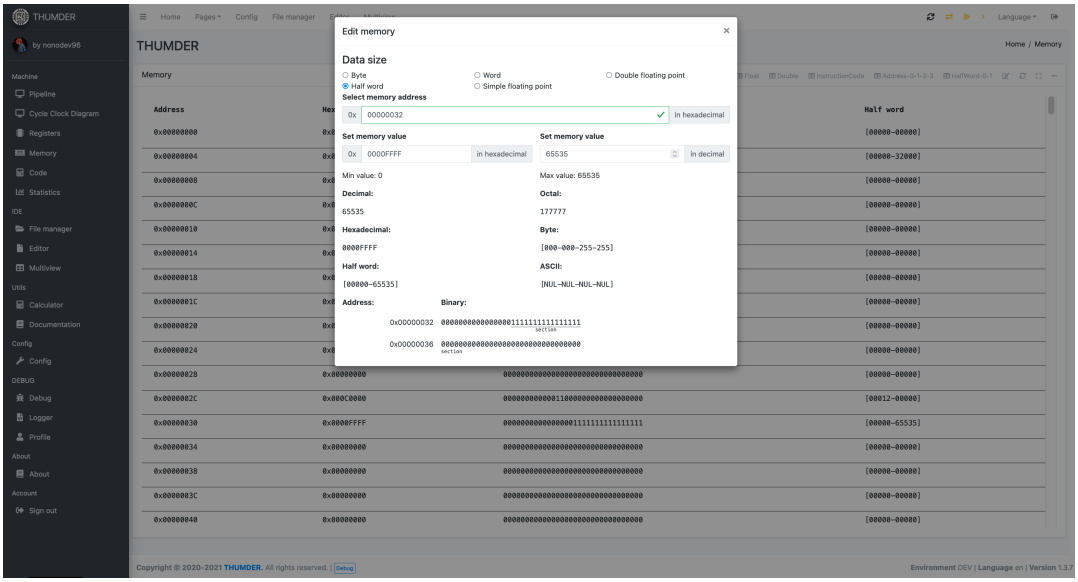


Figura C.2: Editar memoria

C.3. Ejecución

Durante la ejecución podemos ver como los valores de la memoria cambian, también las instrucciones que están en la memoria, los valores en sus distintas representaciones, etc.

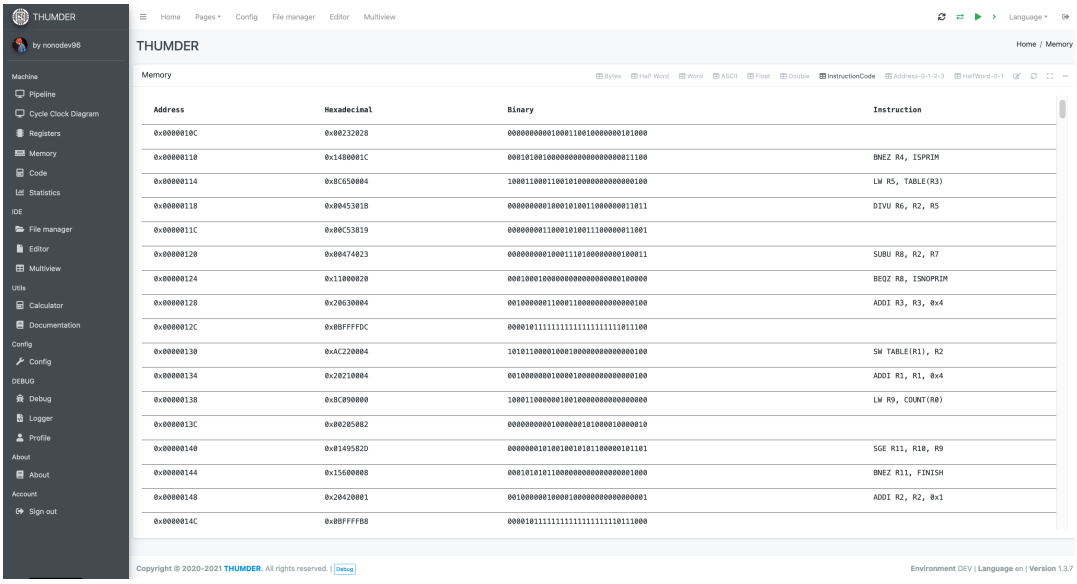


Figura C.3: Representación de la memoria en simulación

En esta sección no podremos ver el flujo del pipeline, esa tarea corresponde a la sección de código.

C.4. Desarrollo

Para actualizar la memoria hay varios métodos, podemos enviar el código que va a ejecutar, el sistema se encarga de escribir en la memoria o escribir directamente en la memoria el valor, para ello se define una serie de direcciones a actualizar, indicar el tipo de dato y el valor hexadecimal, para ello usamos el tipo de dato 'TypeMemoryToUpdateResponse'.

```
export type TypeMemoryToUpdate = {
  typeData: TypeData;
  address: string;
  value: string;
};

export type TypeMemoryToUpdateResponse = {
  allOK: boolean,
  memoryToUpdates: TypeMemoryToUpdate[]
};
```

```
{
  "memory": [ {
   "address": "0x00000100",
   "typeData": "Word",
   "value": "0x00000A00"
  } ]
}
```

Apéndice D

La ventana de código

En esta sección se muestra el proceso para almacenar el código a ejecutar que se encuentra en la memoria, esto nos permite visualizar las instrucciones, los registros y los datos que se insertan en la memoria.

En esta ventana se nos muestra la dirección de memoria en formato hexadecimal, el texto que representa al *label* junto al desplazamiento, así como el binario de su representación, el formato hexadecimal, el *stage* que se encuentra y la instrucción en sí.

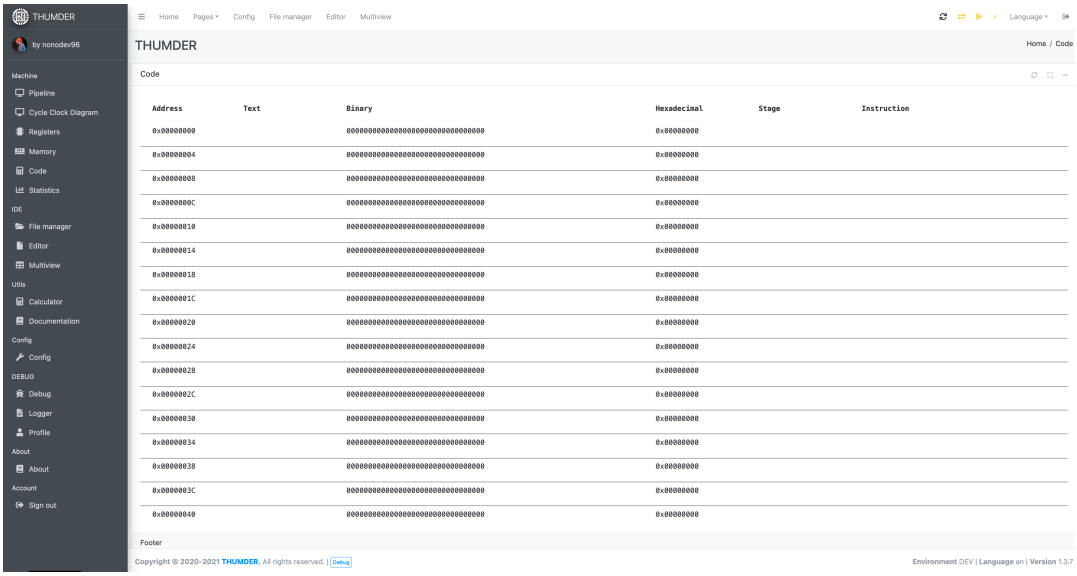


Figura D.1: Ventana de código

Además, nos permite una representación sencilla de la ejecución en el cauce (*Pipeline*).

Al realizar una simulación, esta ventana nos muestra un color representativo de la etapa en la que se encuentra cada instrucción en el pipeline.

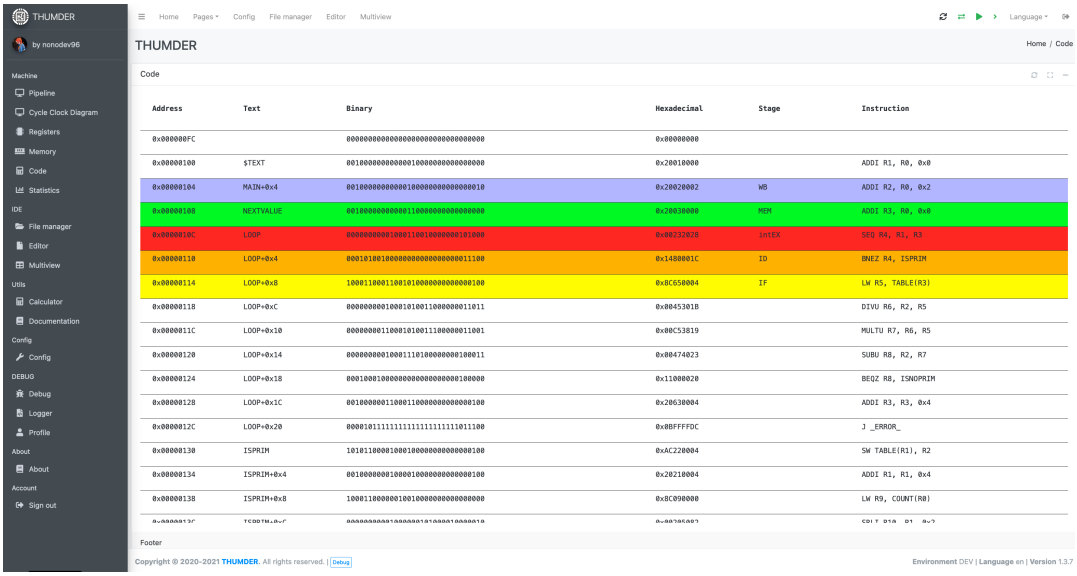


Figura D.2: Ventana de código en ejecución

D.1. Desarrollo

Podemos pedir directamente al servidor que nos genere el código máquina de nuestro código, para ello usamos los eventos **CodeRequest** enviando el mensaje con el tipo de dato ‘TypeFile‘, el servidor nos responderá por el evento **CodeResponse** con las instrucciones máquina en una lista de objetos de tipo ‘TypeCodeResponse‘.

```
export type TypeFile = {
  content: string;
  // ...
}

export type TypeDirectiveData = {
  address:      TypeAddress;
  hexValue:     string;
  text:         string;
```

```
  directive:    TypeDirective;
};

export type TypeInstructionsData = {
  address:      TypeAddress;
  code:         string;
  text?:        string;
  instruction:  string;
};

export type TypeCodeResponse = {
  machineDirectives:    TypeDirectiveData[],
  machineInstructions:  TypeInstructionsData[],
};

{
  "id":                 "",
  "filename":           "prim.s",
  "date":               "2021-08-28T10:25:00.000Z",
  "canSimulate":        true,
  "lines":              68,
  "machineDirectives":  [ {
    "address":  "0x00000000",   "text":         "COUNT",
    "hexValue": "0x0000000A",   "directive":    "WORD"
  } ],
  "machineInstructions": [ {
    "address":  "0x00000100",   "text":         "$TEXT",
    "code":     "0x20010000",   "instruction":  "ADDI R1, R0, 0x0"
  }, {
    "address":  "0x00000104",   "text":         "MAIN+0x4",
    "code":     "0x20020002",   "instruction":  "ADDI R2, R0, 0x2"
  } ],
  "runner": [ {
    "step":             0,
    "line":             0,
    "isNewInstruction": false,
```

```
    "pipeline": {
      "IF": {
        "address": "", "addressRow": 0, "draw": false
      },
      "ID": {
        "address": "", "addressRow": 0, "draw": false
      },
      "intEX": {
        "address": "", "addressRow": 0, "draw": false
      },
      "MEM": {
        "address": "", "addressRow": 0, "draw": false
      },
      "WB": {
        "address": "", "addressRow": 0, "draw": false
      },
      "faddEX":          [],
      "fmultEX":         [],
      "fdivEX":          [],
      "arrows":          []
    },
    "registers":         [],
    "memory":            [],
    "statistics":        {}
  } ]
}
```

Apéndice E

La ventana del diagrama de ciclos de reloj

En esta ventana se muestra la tabla con el historial de la ejecución de la simulación.

Esta tabla representa las instrucciones en las distintas etapas, siendo la instrucción la fila, y la etapa la columna.

Esto nos permite representar las instrucciones y los datos que se leen, calculan o escriben en cada momento.

Esta tabla está hecha con un canvas de HTML 5 para representar flechas de los adelantamientos, movimientos, los colores de las distintas etapas, etc.

E.1. Detenciones y adelantamientos

Las paradas se representan en cajas, que se superponen con un color correspondiente al escenario parado. El contenido de la caja da más información sobre el tipo de puesto:

Tipo	Descripción
R-Stall	RAW-Stall. (Read After Write Stall) Una flecha roja muestra (si está habilitada) la instrucción que provoca la parada.
W-Stall	WAW-Stalls. (Write After Write Stall) Una flecha roja muestra (si está habilitada) la instrucción que provoca el bloqueo.
T-Stall	Trap-Stall. Esta pérdida ocurre solo por una instrucción *trap*, permanece en la etapa IF, hasta que no haya otras instrucciones en el cauce de ejecución (*pipeline*).
S-Stall	Puesto estructural. No hay suficientes recursos para atender la instrucción.
Stall	Detención. Si una instrucción de punto flotante está en la etapa MEM, la siguiente instrucción se detendrá en la etapa intEX, etiquetada con "Stall".

Los adelantamientos se representan con una flecha verde que muestra el origen y el destino del adelantamiento.

E.2. Tabla

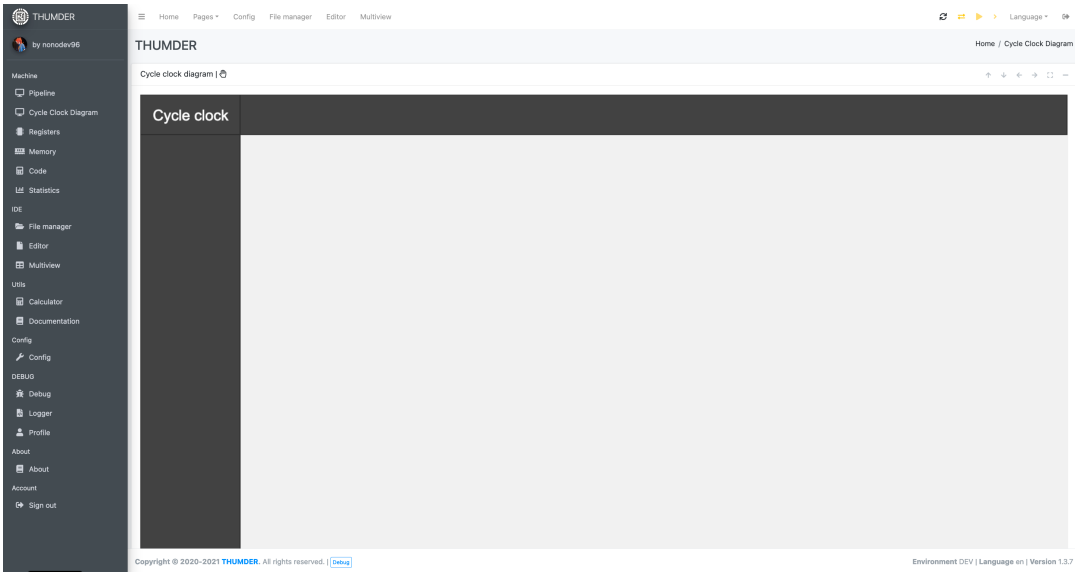


Figura E.1: Diagrama de ciclos del reloj

E.3. Ejecución

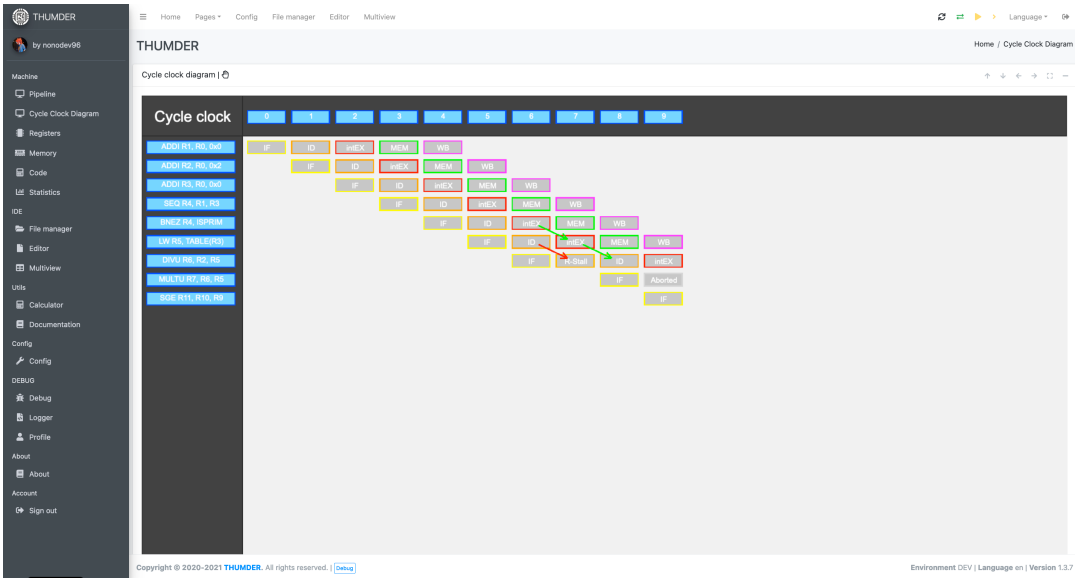


Figura E.2: Diagrama de ciclos del reloj en ejecución

Como podemos ver en la imagen anterior se muestran los distintos adelantamientos en distintos colores (Rojo o verde)

E.4. Desplazamientos

Como podemos ver se han incluido cuatro botones de desplazamiento, representados con flechas (Arriba, Abajo, Izquierda, Derecha).

Estos botones de desplazamiento se han enlazado a los eventos del teclado para poder mover la ventana de forma cómoda, rápida y fluida.

- Desplazamiento:
- (**i key up**): Desplazamiento hacia arriba.
- (**k key down**): Desplazamiento hacia abajo.
- (**j key left**): Desplazamiento hacia la izquierda.
- (**l key right**): Desplazamiento hacia la derecha.

Además, se ha añadido un icono de selección (mano) este indica cuando hemos seleccionado la tabla y vamos a desplazarnos dentro de ella, ya que las teclas de dirección del teclado se usan para moverse en el navegador.

De esta forma hemos indicado cuando bloqueamos el desplazamiento del navegador frente al de la tabla, para que no se produzca un efecto *parallax*.

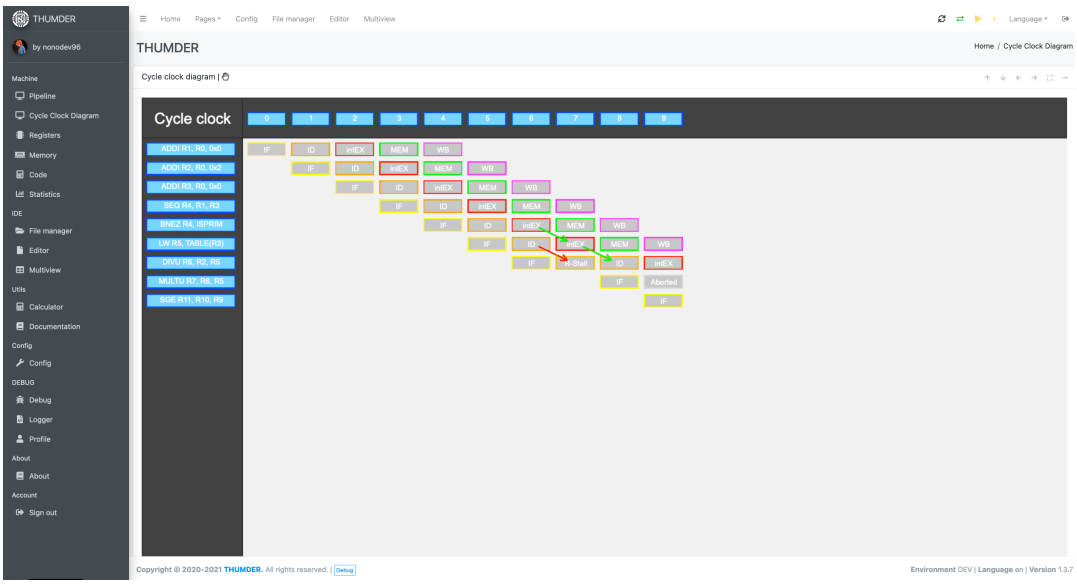


Figura E.3: Desplazamiento de la tabla

E.5. Desarrollo

Para la representación del diagrama de ciclos mediante tablas se usan los datos del cauce de ejecución con sus correspondientes adelantamientos, solo se debe organizar el orden de ejecución por columnas los pasos y por filas las nuevas instrucciones.

Apéndice F

La ventana del cauce de ejecución (Pipeline)

Como se ha explicado en el manual del DLX, esta ventana representa las cinco etapas que puede encontrarse una instrucción en el microprocesador.

Esto nos permite visualizar que dato se está calculando en cada momento, ver cuánto tiempo va a estar en cada etapa, etc.

Etapa	Descripción
IF	Unidad de captación de instrucción. Típicamente referida como la “unidad de carga” en terminología moderna.
ID	Unidad de descodificación de instrucción. Esta unidad toma la instrucción del IF, y extrae el opcode y los operandos. También obtiene los valores en registros si es necesario.
EX	Unidad de ejecución. Ejecuta la instrucción, llamada como ALU en terminología moderna.
MEM	Unidad de acceso a memoria. Obtiene los datos o los escribe para las instrucciones de almacenamiento. Controlada desde las etapas de ID y EX.
WB	WriteBack unit. Llamada como unidad de almacenamiento en terminología moderna. Es la encargada de escribir los resultados en los registros de destino.

Hay unas etapas intermedias en la etapa de ejecución (**EX**), representan el cálculo de datos decimales como divisiones, cálculos de multiplicaciones o sumas, estas etapas son las operaciones en coma flotante IEEE754.

Debemos tener en cuenta la configuración de la máquina, ya que en función de esta configuración podremos tener una unidad o varias.

F.1. Inicialización

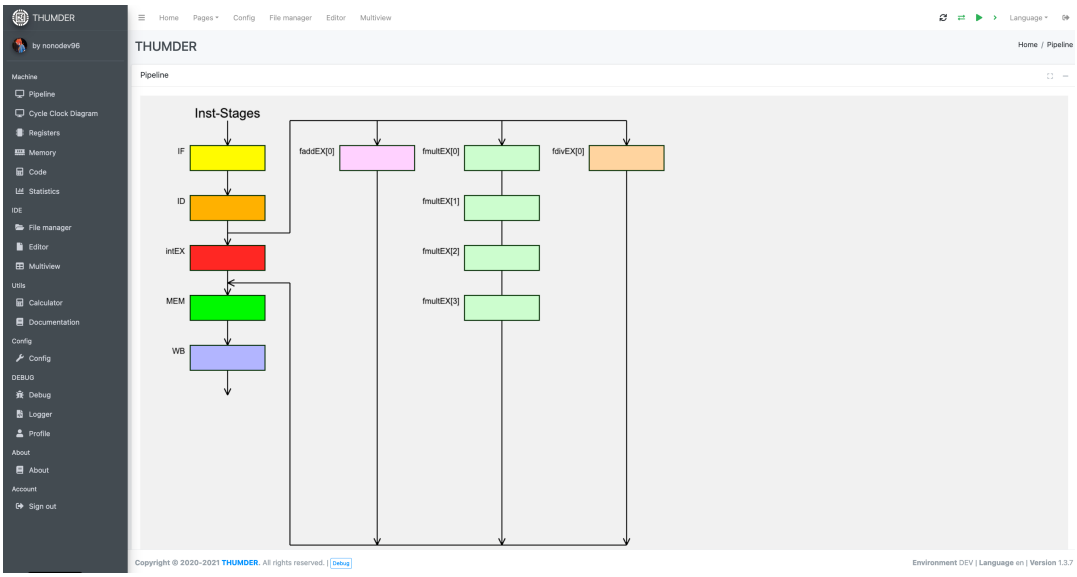


Figura F.1: Ventana del cauce

F.2. Ejecución

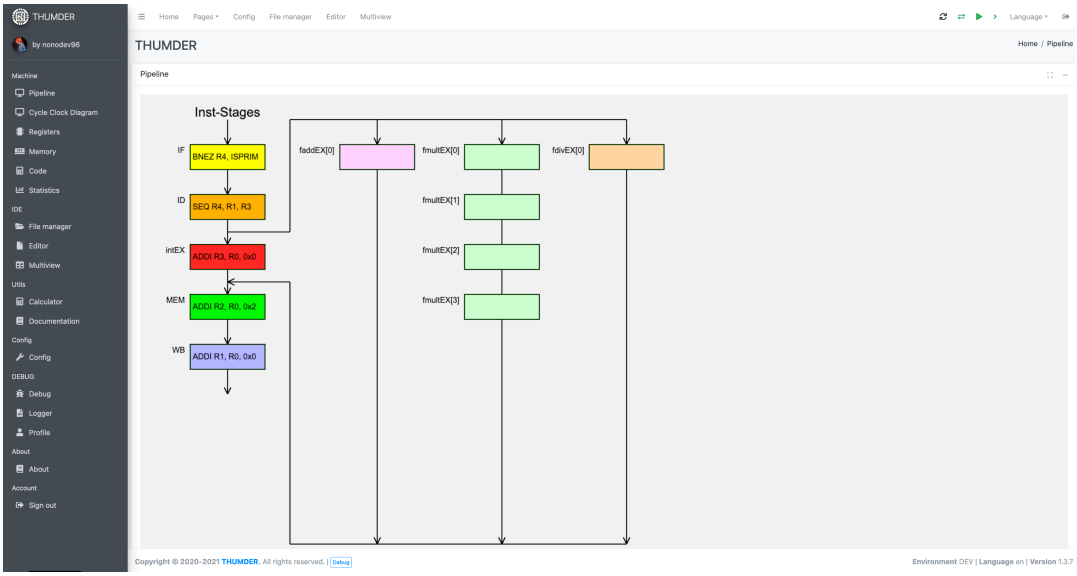


Figura F.2: Ejecución en el cauce

F.3. Desarrollo

Ejemplo parcial de los datos que responde el servidor para representar el cauce (*Pipeline*) en una de sus etapas.

```
export type TypeStall = "Aborted" | "Stall"
                      | "R-Stall" | "T-Stall" | "W-Stall" | "S-Stall";

export type TypeCycleCell = {
  address:      string;
  addressRow:   number;
  draw:         boolean | TypeStall;
};

export type TypeCycleCellUnit = TypeCycleCell & {
  unit?: number
};

export type TypeArrowCycle = {
```

```
  fromAddressRow:   number;
  fromStep:         number;
  toAddressRow:     number;
  toStep:           number;
  color:            number; // hexadecimal
};

export type TypePipeline = {
  IF:       TypeCycleCell;
  ID:       TypeCycleCell;
  intEX:    TypeCycleCell;
  MEM:      TypeCycleCell;
  WB:       TypeCycleCell;
  faddEX:   TypeCycleCellUnit[];
  fmultEX:  TypeCycleCellUnit[];
  fdivEX:   TypeCycleCellUnit[];
  arrows:   TypeArrowCycle[];
};
```

Apéndice G

Estadísticas

En la sección de estadísticas mostramos el listado de datos representativos de una simulación, estos datos indican los ciclos ejecutados, decodificaciones, instrucciones en el pipeline, saltos, operaciones en coma flotante, detenciones, adelantamientos, etc. Esto nos sirve para analizar el rendimiento de nuestro programa desarrollado en DLX.

Esta sección se encuentra con traducciones en inglés y en español.

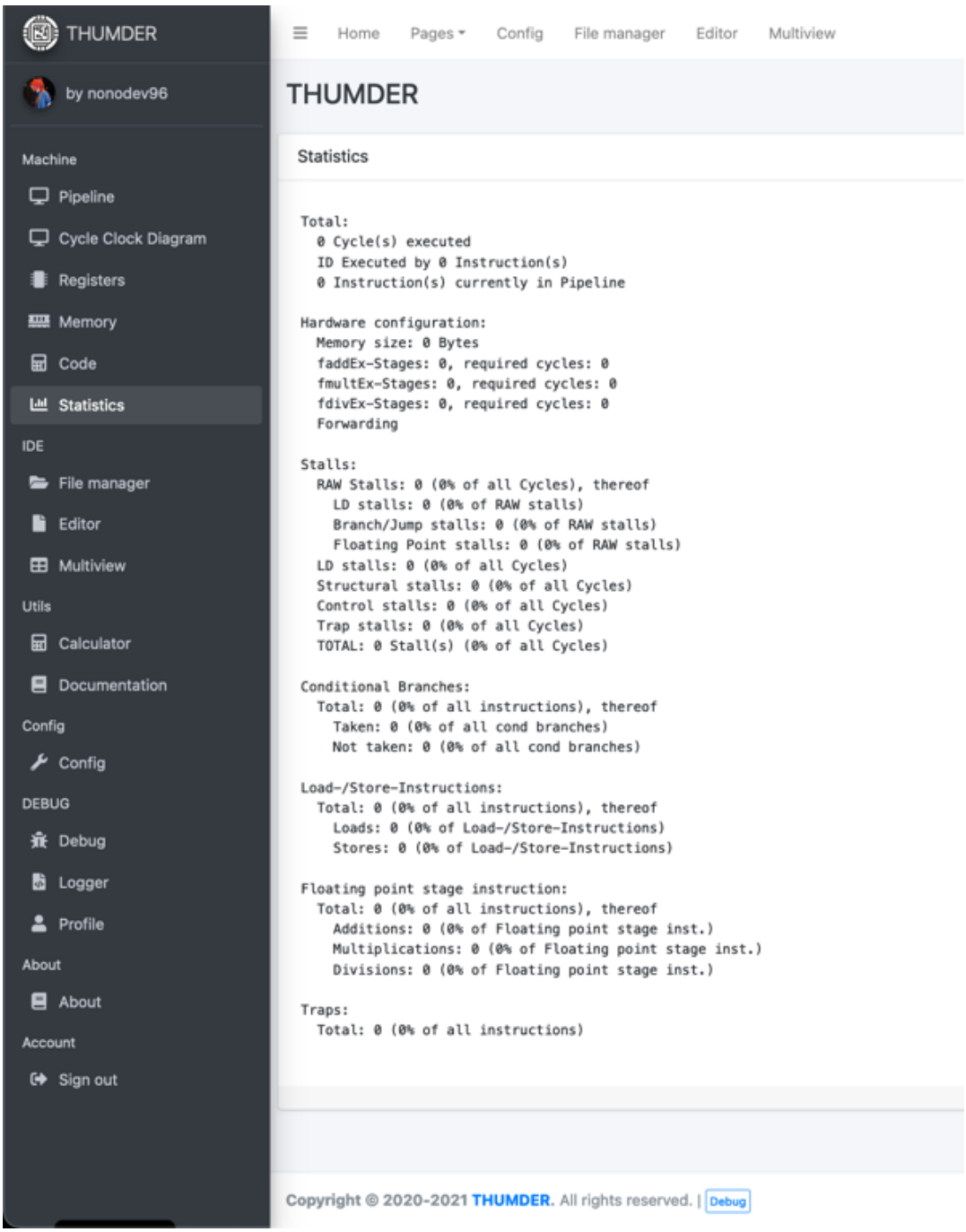


Figura G.1: Estadísticas

G.1. Desarrollo

Para el envío de datos con el servidor se establece el siguiente protocolo, no tenemos que enviar todo el objeto JSON en cada paso, solo la sección que se actualiza en

cada paso (‘Partial<TypeDataStatistics>‘), esto nos permite solo actualizar las secciones que nos interese.

```
export type TypeDataStatistics = {
  TOTAL: {
    CYCLES_EXECUTED:            { cycles: number };
    ID_EXECUTED:                { instructions: number };
    INSTRUCTIONS_IN_PIPELINE:   { instructions_in_pipeline: number };
  };
  HARDWARE: {
    MEMORY_SIZE:                { size: number };
    FADD_EX_STAGES:             { num: number; cycles: number };
    FMULT_EX_STAGES:            { num: number; cycles: number };
    FDIV_EX_STAGES:             { num: number; cycles: number };
    FORWARDING:                 { enabled: boolean };
  };
  STALLS: {
    RAW_STALLS:                 { num: number; per: number };
    LD_STALLS:                  { num: number; per: number };
    BRANCH_STALLS:              { num: number; per: number };
    FLOATING_POINT_STALLS:      { num: number; per: number };
    WAW_STALLS:                 { num: number; per: number };
    STRUCTURAL_STALLS:          { num: number; per: number };
    CONTROL_STALLS:             { num: number; per: number };
    TRAP_STALLS:                { num: number; per: number };
    TOTAL:                      { num: number; per: number };
  };
  CONDITIONAL: {
    TOTAL:                      { num: number; per: number };
    TAKEN:                      { num: number; per: number };
    NOT_TAKEN:                  { num: number; per: number };
  };
  LOAD_STORE: {
    TOTAL:                      { num: number; per: number };
    LOADS:                      { num: number; per: number };
    STORES:                     { num: number; per: number };
```

```
  };
  FLOATING: {
    TOTAL:                      { num: number; per: number };
    ADDITIONS:                  { num: number; per: number };
    MULTIPLICATIONS:            { num: number; per: number };
    DIVISIONS:                  { num: number; per: number };
  };
  TRAPS: {
    TOTAL:                      { num: number; per: number };
  }
};
```

Apéndice H

Puntos de ruptura (breakpoints)

Para poder entender esta sección es recomendable ir al gestor de ficheros, pues es esta sección la que se encarga de mostrar los puntos de parada (*breakpoints*).

Debemos tener en cuenta que el listado de puntos de parada (*breakpoints*) lo gestiona el fichero.

H.1. Desarrollo

A la hora de pedir al servidor una simulación de nuestro código debemos indicarle unos datos de partida, un identificador, nombre del fichero, fecha de inicio de la simulación, el contenido del fichero (código de nuestro programa), los tipos de datos de la memoria y los registros que no son nulos en nuestro proyecto, así como los puntos de parada 'breakpoints', las líneas de nuestro código que queremos parar en una simulación.

```
export type TypeSimulationInitRequest = {
  id:           string;                 filename:   string;
  date:         string;                 content:    string;
  registers:    TypeRegisterToUpdate[];
  memory:       TypeMemoryToUpdate[];
  breakpoints:  TypeLine[]; // <- Líneas con breakpoints
};
```

Apéndice I

El proceso de simulación

Hemos llamado proceso de simulación a los pasos necesarios para preparar una simulación.

1. Configurar la máquina (aunque tiene una configuración por defecto recomendamos modificarla a nuestro gusto)
2. Desde el gestor de ficheros crear nuestro fichero de prueba.
3. Abrir o crear el fichero a simular.
   - Una vez en el editor será este el fichero que enviemos al servidor a simular.
4. Sincronizar con el servidor o reiniciar simulación.
   - Limpia la memoria local de la simulación y pide al servidor que intérprete el fichero de nuevo.
5. Simular (dos formas):
   - Simular paso por paso.
   - Simular por reloj.
6. Ir a una sección de visualización
   - **Multiview**
   - Pipeline
   - Cycle Clock Diagram
   - Registers
   - Memory
   - Code
   - Statistics

I.1. Desarrollo

Para iniciar el proceso se debe sincronizar el código de nuestro editor, para ello usamos el mensaje 'TypeSimulationInitRequest' en formato JSON para iniciar una simulación, el servidor responderá con el objeto JSON que corresponde al formato de mensaje 'TypeSimulationInitResponse', por último el servidor nos irá respondiendo paso a paso con el mensaje 'TypeSimulationStep' en formato JSON.

```
export type TypeSimulationInitRequest = {
  id:           string;
  filename:     string;
  date:         string;
  content:      string;
  breakpoints:  TypeLine[];
  registers:    TypeRegisterToUpdate[];
  memory:       TypeMemoryToUpdate[];
};

export type TypeSimulationInitResponse = {
  filename:             string;
  id:                   string;
  date:                 string;
  lines:                number;
  canSimulate:          boolean;
  errors:               TypeErrorInCode[];
  machineDirectives:    TypeDirectiveData[];
  machineInstructions:  TypeInstructionsData[];
  runner:               TypeSimulationStep[];
};

export type TypeSimulationStep = {
  isComplete?:      boolean;
  isBreakpoint?:    boolean;
  step:             number;
  line:             number;
  isNewInstruction: boolean;
```

```
  pipeline:          TypePipeline;
  registers:         TypeRegisterToUpdate[];
  memory:            TypeMemoryToUpdate[];
  statistics:        Partial<TypeDataStatistics>;
};
```

I.2. Control de la simulación

Figura I.1: Reiniciar socket

Figura I.2: Sincronizar con el servidor

Figura I.3: Simular por reloj

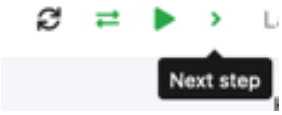


Figura I.4: Simular por pasos

Apéndice J

Editar configuración

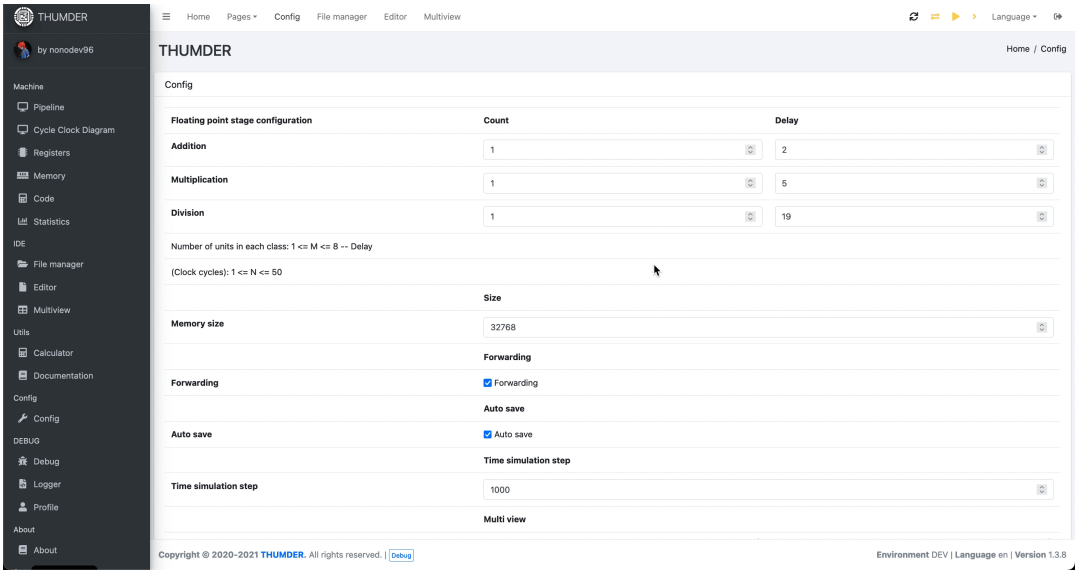


Figura J.1: Editar la configuración del IDE

J.1. Núcleos de operaciones de punto flotante.

DLX tiene una estructura con 3 tipos de núcleos especializados en operaciones de punto flotante (suma, división y multiplicación), estos núcleos se pueden configurar para tener entre 1 y 8 núcleos y que cada uno de estos tenga un tiempo de cálculo de operación, nosotros podemos editar está configuración.

J.2. Tamaño de la memoria

Podemos modificar el tamaño de la memoria en ‘Bytes‘, por defecto tiene ‘32768 bytes‘.

J.3. Adelantamientos

Podemos activar o desactivar los adelantamientos.

J.4. Tiempo de simulación

Podemos definir el tiempo de simulación de cada paso para que sea más sencillo ejecutar y ver como se mueven los datos.

Como mínimo tenemos ‘100 ms‘ en cada paso.

J.5. Autoguardado

Para que sea más sencillo se incluye la opción de autoguardado para ejecutar y guardar en la nube.

J.6. Multiview

Podemos indicar que ver y en qué orden ver las distintas secciones en la vista múltiple.

J.7. Desarrollo

Para actualizar la configuración de la máquina se envía un JSON de tipo ‘TypeConfigurationMachine‘ por el socket ‘UpdateConfigurationMachineRequest‘, se puede tratar la respuesta por ‘UpdateConfigurationMachineRequest‘ o por ‘UpdateConfigurationMachineResponse‘, ya que ambos eventos usan el socket.

Este tipo de envío nos permite configurar los núcleos y el delay de cada núcleo, así como el tamaño de la memoria y si se encuentra activo los adelantamientos.

```
export type TypeConfigurationMachine = {
  addition: {
    count: number;
    delay: number;
  },
  multiplication: {
    count: number;
    delay: number;
  },
  division: {
    count: number;
    delay: number;
  },
  memorySize: number;
  enabledForwarding: boolean;
};
```

Apéndice K

Gestor de ficheros

El gestor de ficheros es una de las secciones más importantes, ya que es la encargada de almacenar, cargar, crear o eliminar los ficheros de cada usuario.

Esta herramienta usa Firebase Storage y Firebase Database, por una parte, almacenamos los ficheros que vienen por defecto y en la base de datos las referencias a los ficheros de los usuarios, así como pequeños metadatos de cada fichero.

- Nombre del fichero
- Ruta del fichero
- Fecha de creación
- Usuario al que pertenece (**$Key**)
- UUID

Apéndice L

Editor de ficheros

El Editor de ficheros es una de las secciones más importantes y complejas de nuestro proyecto, se trata de un editor de ‘Monaco‘ (un proyecto de Microsoft) modificado y con una extensión creada por nosotros que nos permita analizar, **interpretar**, **autocompletar**, **marcar** y **mostrar documentación**.

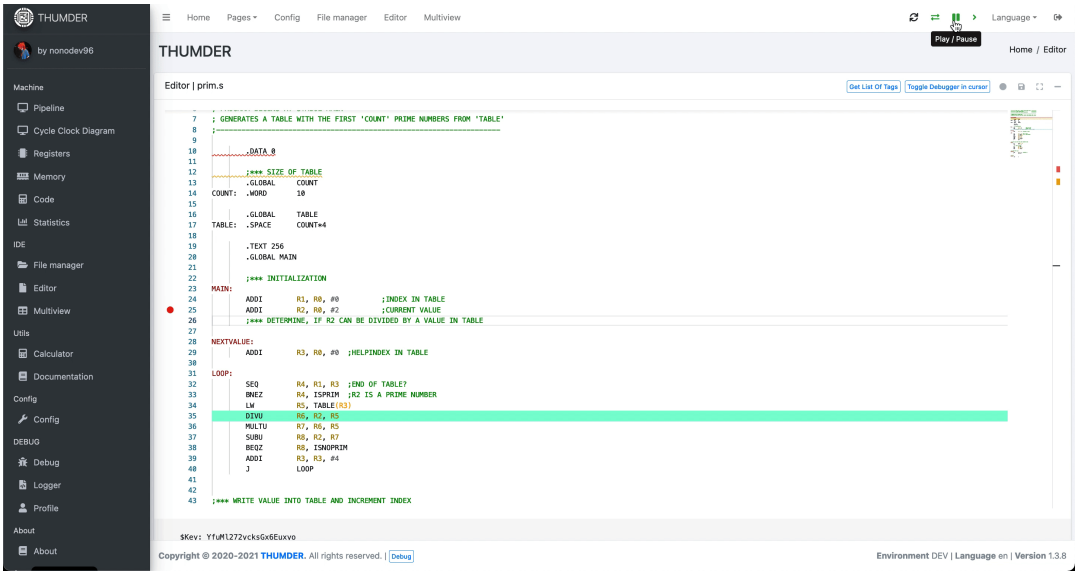


Figura L.1: Ejemplo de file editor THUMDER

En este ejemplo se ve el proceso para abrir un fichero, su ejecución y varias pruebas, entre ellas mostrar errores (no hay en el código, solo es un ejemplo del mensaje y

la gravedad del posible error), añadir un ‘breakpoint‘, ejecutar un paso o ejecutar la simulación.

Como podemos observar, la línea de ejecución en color azul verdoso corresponde a la etapa **IF**.

L.1. Autocompletado

Como podemos ver en el ejemplo del inicio se puede autocompletar las instrucciones, para ello se han puesto una serie de ejemplos que nos añaden los registros y desplazamientos de ejemplo.

L.2. Documentación

Para mostrar la documentación solo debemos pasar el ratón por encima de la palabra (instrucción) y se nos mostrará los datos de la instrucción (En inglés).

Todos estos ejemplos se encuentran definidos en ‘The DLX Instruction Set, BYU Edition‘.

L.3. Errores

Como hemos visto en el ejemplo, el proyecto nos muestra errores simples, pero en caso de tener errores más complejos, estos se verán reflejados una vez que sincronicemos nuestro código con el servidor y esté intérprete el fichero.

Estos errores se muestran en función de la gravedad, con un mensaje descriptivo.

```
export type TypeErrorInCode = {
  line:     number;
  message:  string;
  severity: EnumSeverity;
}
```

L.4. Desarrollo

Leer la sección ‘06.TheBreakpoints.md‘ para comprender como se ajustan los puntos de ruptura (*breakpoints*) en el editor.

Después de pedir una simulación, si todo va correctamente (‘canSimulate‘) nos permitirá ejecutar nuestro código, en caso fallo, el campo de ‘errors‘ nos reportará los distintos fallos puede ocurrir en nuestro código y que impidan su ejecución.

```
export type TypeSimulationInitRequest = {
  id:           string;
  filename:     string;
  date:         string;
  content:      string;

  breakpoints:  TypeLine[];
  registers:    TypeRegisterToUpdate[];
  memory:       TypeMemoryToUpdate[];
};

export type TypeSimulationInitResponse = {
  filename:             string;
  id:                   string;
  date:                 string;
  lines:                number;
  canSimulate:          boolean;
  errors:               TypeErrorInCode[];

  machineDirectives:    TypeDirectiveData[];
  machineInstructions:  TypeInstructionsData[];
  runner:               TypeSimulationStep[];
};
```

Apéndice M

Vista múltiple

La ventana de vista múltiple es una de las herramientas más sencillas de usar, agrupa los distintos componentes que representan el estado del procesador, la memoria, el diagrama de ciclos o las estadísticas, es la forma más sencilla de visualizar todos los datos en conjunto y poder editar algunos datos en ejecución.

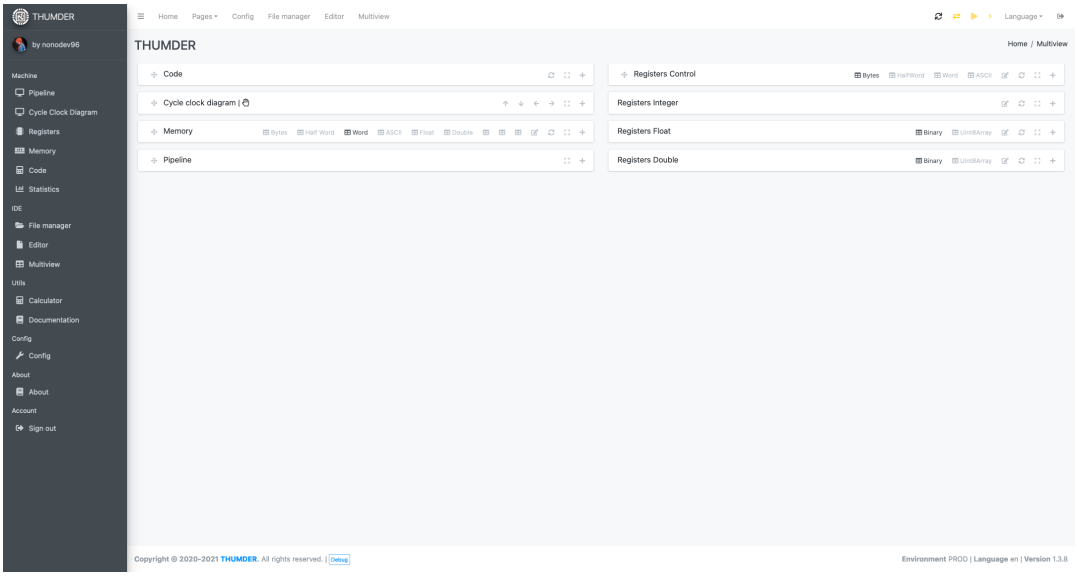


Figura M.1: Ventana de vista múltiple

Se ha dividido en dos columnas, estás dos columnas, se pueden reorganizar y dividir para colocar la sección de la forma que nos resulte más cómoda.

Estás secciones se pueden mover, pero al cambiar de ventana se vuelve al estado inicial, si se quiere cambiar el estado inicial se debe cambiar el orden desde la configuración.

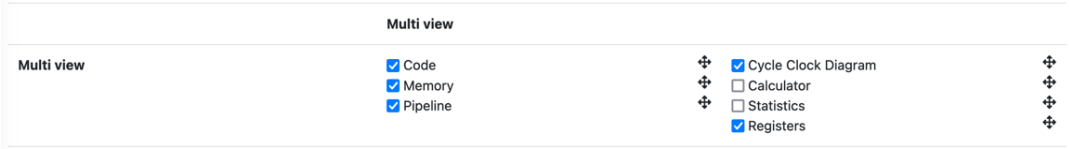


Figura M.2: Configuración

Nota: Se ha añadido este icono para representar formas que tienen una funcionalidad **Drag and drop**, es decir se pueden desplazar y acomodar al gusto.

Figura M.3: Icono - drag and drop

Apéndice N

Protocolo de comunicación

El protocolo de comunicación usa el canal que se crean en una conexión tipo túnel del WebSocket, usamos esto y los eventos de escucha de *socket.io* para diferenciar los mensajes con peticiones de los de respuesta, estos mensajes siempre son encapsulados en formato JSON.

El estado de la simulación es dependiente de la conexión con el servidor, pues usamos el identificador de la conexión como identificador de la máquina, de tal forma que al enviar una petición al servidor siempre operemos con la misma máquina.

Evento escucha	**Mensaje recibido**
UpdateConfigurationMachineRequest	`TypeConfigurationMachine`
CodeRequest	`TypeFile`
UpdateRegisterRequest	`TypeRegisterToUpdate[]`
UpdateMemoryRequest	`TypeMemoryToUpdate[]`
GetAllRegistersRequest	-
GetAllMemoryRequest	-
GetAllStatisticsRequest	-
SimulationNextStepRequest	-
SimulationInitRequest	`TypeSimulationInitRequest`

Tabla. N.1: Protocolo de comunicación peticiones

Evento respuesta	**Mensaje enviado**
UpdateConfigurationMachineResponse	`TypeConfigurationMachine`
CodeResponse	`TypeCodeResponse`
UpdateRegisterResponse	`TypeRegisterToUpdateResponse`
UpdateMemoryResponse	`TypeMemoryToUpdateResponse`
GetAllRegistersResponse	`TypeAllRegisters`
GetAllMemoryResponse	`TypeAllMemory`
GetAllStatisticsResponse	`TypeDataStatistics`
SimulationNextStepResponse	`TypeSimulationStep`
SimulationInitResponse	`TypeSimulationInitResponse`

Tabla. N.2: Protocolo de comunicación respuestas

Glosario

ALU ALU (Arithmetic Logic Unit), tambien llamado circuito digital que se implementan en microprocesadorees con operaciones aritméticas y operaciones lógicas entre distintos valores. 155

backend El backend es la parte del desarrollo web que se encarga de que toda la lógica operacional de un proyecto. Contiene la capa de acceso a los datos, la lógica tecnológica que hace de núcleo o motor del proyecto. 98, 101, 107

breakpoints Marca que ponemos en una linea de nuestro código fuente, de tal forma que cuando la ejecución llegue a ese punto el proceso de nuestra aplicación se detendrá. 26, 92, 93, 116

CF Horas-Persona. 16

CLI Command Line Interface (*CLI*) o en español interfaz de línea de comandos, son programas de consola que nos permiten crear y gestionar los distintos aspectos del proyecto. 99, 100

Contador de programa (PC) Program Counter o contado de programa, es un registro del procesador que indica la posición donde está el procesador en su secuencia de instrucciones. 163

CPU Central Processing Unit (*CPU*) en español Unidad Central de Procesamiento, es la encargada de procesar las instrucciones ejecutando operaciones aritmeticas, lógicas y de entrada y salida de dispositivos externos. 3

CSS CSS, en español hojas de estilo en cascada, es un lenguaje de diseño gráfico para definir y crear la presentación de un documento estructurado escrito en un lenguaje de marcado. 99

Dirección Dirección base de la memoria. 160

DOM Document Object Model (*DOM*) en español Modelo de Objetos del Documento es una interfaz que proporciona un conjunto estándar de objetos para representar documentos HTML, XHTML y XML, es un modelo estándar sobre cómo pueden combinarse dichos objetos, y una interfaz estándar para acceder a ellos y manipularlos. 99, 104, 105

E Esfuerzo estimado en horas-persona. 16, 19

EF Factores ambientales. 15–17

frontend El frontend es la parte del desarrollo web que se dedica a la interfaz de un sitio web, en pocas palabras del diseño de un sitio web, desde la estructura del sitio hasta los estilos como colores, fondos, tamaños hasta llegar a las animaciones y efectos. 98, 99

Funct Selecciona la variante de la operación del campo op. 160

I/O I/O son las siglas en inglés de Input/Output, cuyo significado en español es Entrada/Salida, se usa para describir cualquier programa o dispositivo que transfiere datos desde un ordenador hacia un dispositivo periférico o viceversa. 152

IDE Es un programa informático que proporciona servicios y herramientas al programador, para facilitar el desarrollo de software. 2

LEAN Amplify Learning Se trata de usar cualquier método al alcance para poder aumentar el conocimiento sobre el producto y los usuarios o el mercado que lo va a usar. 11

mocks los *Mocks*, “son objetos pre programados con expectativas que conforman la especificación de lo que se espera que reciban las llamadas”, es decir, son

objetos que se usan para probar que se realizan correctamente llamadas a otros métodos, por ejemplo, a una web API, por lo que se utilizan para verificar el comportamiento de los objetos. 148

MVVM MVVM abreviatura de Model-View-ViewMode (modelo-vista-modelo de vista) es un tipo de arquitectura que usa la tecnología de enlace bidireccional para unir la vista al modelo. 45, 109

OP Operación de la instrucción. 159, 160

ORM Modelo de programación que permite mapear las estructuras de una base de datos relacional. 102

RAW También se conoce como Dependencia verdadera o Dependencia de flujo. Ocurre cuando el valor producido por una instrucción es requerido por una instrucción posterior. Por ejemplo:

```
ADD R1, -, -;
SUB -, R1, -; .4
```

RD Registro del operando de destino, resultado de la operación. 160

responsive La interfaz se adapta a la pantalla o dispositivo. 45

RISC Reduced Instruction Set Computer, en español Computador con conjunto de instrucciones reducido. 153

RS Primer registro, operando fuente. 159, 160, 162

RT Segundo registro, operando fuente. 159, 160

Shamt Desplazamiento. 160

SPA Una web SPA o Single page application se define como una forma de desarrollo web en la que la página web está contenida en un único archivo, tanto el CSS, código JavaScript y el HTML con el objetivo de optimizar la aplicación. 99, 100

stalls Una detencion o *stall* es cuando el procesamiento de una instrucción es pausado en una de sus etapa hasta que se termine de calcular un dato, esto suele ocurrir en la etapa MEM del cauce de ejecución. 4

stubs Los *Stubs*, "proporcionan respuestas predefinidas a ciertas llamadas durante las pruebas, sin responder a otra cosa para la que no hayan sido programados", es decir, los *stubs* son configurados para que devuelvan valores que se ajusten a lo que la prueba unitaria quiere probar, por lo que se utilizan para verificar el estado de los objetos. 148

TCF Factores técnicos. 15–17

TDD Desarrollo guiado por pruebas. 146

TFG Trabajo final de grado. 7, 104

UAW Factor de peso de los actores sin ajustar. 15, 16

UCP Puntos de casos de uso ajustados. 15–18

URL URL significa *Uniform Resource Locator* y es la dirección única y específica que se asigna a cada uno de los recursos disponibles de la *World Wide Web* para que puedan ser localizados por el navegador y visitados por los usuarios. 108, 125

UUCP Puntos de casos de uso sin ajustar. 15, 16

UUCW Factor de peso de los casos de uso sin ajustar. 15, 16

WAR También se conoce como anti dependencia. Estos peligros ocurren cuando el registro de salida de una instrucción se usa justo después de leer una instrucción anterior. Por ejemplo:

```
ADD -, R1, -;
SUB R1, -, -; .
```
4

WAW También se conoce como dependencia de salida. Estos peligros suceden cuando el registro de salida de una instrucción se usa después de que lo haya escrito una instrucción anterior. Por ejemplo:

```
ADD R1, -, -;
SUB R1, -, -; .4
```

Bibliografía

[1] AENOR. *Criterios generales para la elaboración de proyectos de sistemas de información UNE 157801:2007*. AENOR, 2007. doi: M43360:2007.

[2] Norma Nacional Americana. *Guía de los Fundamentos de la Dirección de Proyectos*. Norma Nacional Americana, 2004. doi: ANSI/PMI99-001-2004.

[3] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Amsterdam, 5 edition, 2012. ISBN 978-0-12-383872-8.

[4] Windlx, 1995. URL http://electro.fisica.unlp.edu.ar/arq/downloads/Software/WinDLX/windlx.html.

[5] UNE. Une 157801:2007, 2007. URL https://www.une.org/encuentra-tu-norma/busca-tu-norma/norma/?c=N0039577. Disponible en https://www.une.org/encuentra-tu-norma/busca-tu-norma/norma/?c=N0039577.

[6] opendlx, 2011. URL https://github.com/smetzlaff/openDLX.

[7] Dlx electron simulator, 2018. URL https://github.com/fwcd/dlx.

[8] Modelos de desarrollo de software tradicional / Ágil, 2020. URL https://www.becas-santander.com/es/blog/metodologias-desarrollo-software.html. Disponible en https://www.becas-santander.com/es/blog/metodologias-desarrollo-software.html.

[9] Atlassian. ¿qué es scrum?, 2017. Disponible en https://www.atlassian.com/es/agile/scrum.

[10] Julia Martins. Mucho más que solo tallas de camisetas: Descubre cómo las tallas de camisetas pueden ayudarte con la estimación de los proyectos,

2021. URL https://asana.com/es/resources/t-shirt-sizing. Disponible en https://asana.com/es/resources/t-shirt-sizing.

[11] Carlton. Impedimentos vs. bloqueadores: ¿por qué hacer la distinción?, 2016. URL https://lookforwardconsulting.com/2016/04/14/impedimentos-vs-bloqueadores-por-que-hacer-la-distincion/. Disponible en https://lookforwardconsulting.com/2016/04/14/impedimentos-vs-bloqueadores-por-que-hacer-la-distincion/.

[12] Geeks For Geeks. Estimation technique in agile, 2022. URL https://www.geeksforgeeks.org/estimation-technique-in-agile/. Disponible en https://www.geeksforgeeks.org/estimation-technique-in-agile/.

[13] Bautam Banerjee. Use case points, 2001. URL http://www2.fiit.stuba.sk/~bielik/courses/msi-slov/reporty/use_case_points.pdf. Disponible en http://www2.fiit.stuba.sk.

[14] BOE. Convenio colectivo estatal de empresas de consultoría, y estudios de mercados y de la opinión pública, 2018. URL https://www.boe.es/boe/dias/2018/03/06/pdfs/BOE-A-2018-3156.pdf. Disponible en https://www.boe.es/boe/dias/2018/03/06/pdfs/BOE-A-2018-3156.pdf.

[15] BOE. Convenio colectivo estatal del sector de empresas de publicidad, 2020. URL https://www.boe.es/diario_boe/txt.php?id=BOE-A-2020-13574. Disponible en https://www.boe.es/diario_boe/txt.php?id=BOE-A-2020-13574.

[16] Metodología gestión de requerimientos, 2012. URL https://sites.google.com/site/metodologiareq/.

[17] ECMAScript. Ecmascript, 2022. URL https://www.ecma-international.org/. Disponible en https://www.ecma-international.org/.

[18] OpenJS. Node.js, 2022. URL https://nodejs.org/es/. Disponible en https://nodejs.org/es/.

[19] Twitter. Bootstrap, 2022. URL https://getbootstrap.com/. Disponible en https://getbootstrap.com/.

[20] Semantic Org. Semantic ui, 2022. URL https://github.com/semantic-org/semantic-ui. Disponible en https://github.com/semantic-org/semantic-ui.

[21] Admin LTE. Github @adminlte, 2021. Disponible en https://adminlte.io/.

[22] Semantic Org. Foundation, 2022. URL https://get.foundation. Disponible en https://get.foundation.

[23] statisticsanddata. Most popular backend frameworks 2022, 2001. URL https://statisticsanddata.org/data/most-popular-backend-frameworks-2012-2022/. Disponible en https://statisticsanddata.org/data/most-popular-backend-frameworks-2012-2022/.

[24] Microsoft. Mono, 2022. URL https://www.mono-project.com/. Disponible en https://www.mono-project.com/.

[25] Google. The chromium projects, 2022. URL https://www.chromium.org. Disponible en https://www.chromium.org.

[26] Cheng Zhao. Electron, 2013. URL https://en.wikipedia.org/wiki/Electron_(software_framework). Disponible en https://en.wikipedia.org/wiki/Electron_(software_framework).

[27] Instituto Tecnológico de Massachusetts. Licencia mit, 2022. URL https://opensource.org/licenses/MIT. Disponible en https://opensource.org/licenses/MIT.

[28] meteor. Meteor, 2022. URL https://www.meteor.com. Disponible en https://www.meteor.com.

[29] elisiri_. Proton native, 2022. URL https://proton-native.js.org. Disponible en https://proton-native.js.org.

[30] Facebook. Meta, 2022. URL https://about.facebook.com/meta. Disponible en https://about.facebook.com/meta.

[31] Google. Selenium ide, 2022. URL https://www.selenium.dev/selenium-ide/. Disponible en https://www.selenium.dev/selenium-ide/.

[32] OpenJS. Webdriverio, 2022. URL https://webdriver.io. Disponible en https://webdriver.io.

[33] Jan Molak. Serenity js, 2022. URL https://github.com/serenity-js. Disponible en https://github.com/serenity-js.

[34] OpenJS. Mocha, 2022. URL https://mochajs.org/. Disponible en https://mochajs.org/.

[35] Antonio Mudarra Machuca. Github @nonodev96 thumder, 2022. URL https://github.com/nonodev96/THUMDER. Disponible en https://github.com/nonodev96/THUMDER.

[36] Antonio Mudarra Machuca. Github @nonodev96 thumder-server, 2022. URL https://github.com/nonodev96/THUMDER-server. Disponible en https://github.com/nonodev96/THUMDER-server.

[37] Antonio Mudarra Machuca. Github @nonodev96 thumder-pixi, 2022. URL https://github.com/nonodev96/THUMDER-pixi. Disponible en https://github.com/nonodev96/THUMDER-pixi.

[38] statisticsanddata. Descripción node.js, 2022. URL https://www.krama.es/tecnologias.html. Disponible en https://www.krama.es/tecnologias.html.

[39] Node. Nodejs - event loop, 2016. URL https://nodejs.dev/learn/the-nodejs-event-loop. Disponible en https://nodejs.dev/learn/the-nodejs-event-loop.

[40] DevExpress. Devextremecompany, 2022. URL https://www.devexpress.com/aboutus/. Disponible en https://www.devexpress.com/aboutus/.

[41] Microsoft. Github @microsoft, 2018. Disponible en https://github.com/microsoft/monaco-editor.

[42] Microsoft. Documentación @microsoft, 2018. Disponible en https://docs.microsoft.com/es-es/cpp/cpp/integer-limits?view=msvc-160.

[43] Math @stackexchange. Drawing an arrow, 2016. Disponible en https://math.stackexchange.com/questions/1314006/drawing-an-arrow.

[44] Google LLC. Angular material - material design components for angular, 2022. URL https://material.angular.io/. Disponible en https://material.angular.io/.

[45] Wikipedia. Ide - entorno de desarrollo integrado, 2022. URL https://es.wikipedia.org/wiki/Entorno_de_desarrollo_integrado. Disponible en https://es.wikipedia.org/wiki/Entorno_de_desarrollo_integrado.

[46] Mozilla. Get & set, 2015. URL https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Functions/get. Disponible en https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Functions/get.

[47] Angular. Property binding, 2022. URL https://angular.io/guide/property-binding. Disponible en https://angular.io/guide/property-binding.

[48] Estandar ieee 754, 2021. URL https://es.wikipedia.org/wiki/IEEE_754.

[49] PixiJS Team. Pixijs, 2022. URL https://pixijs.com/. Disponible en https://pixijs.com/.

[50] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Signature Series. Addison-Wesley, Upper Saddle River, NJ, 2007. ISBN 978-0-13-149505-0. URL https://www.safaribooksonline.com/library/view/xunit-test-patterns/9780131495050/.

[51] Antonio Mudarra Machuca. Github @nonodev96 thumder-ontology, 2022. Disponible en https://github.com/nonodev96/THUMDER-ontology.

[52] GitHub @npm. Proyecto node package manager, 2012. Disponible en https://www.npmjs.com/ o https://github.com/npm.

[53] Universidad de Valencia. Manual dlx, 2022. URL https://www.uv.es/varnau/ManualDLX.pdf. Disponible en https://www.uv.es/varnau/ManualDLX.pdf.

[54] open std. C programming language standard, 2022. URL https://www.open-std.org/jtc1/sc22/wg14/. Disponible en https://www.open-std.org/jtc1/sc22/wg14/.

[55] Thumder - wiki, 2022. URL https://github.com/nonodev96/THUMDER/wiki. Disponible en https://github.com/nonodev96/THUMDER/wiki.

Generado con la plantilla LaTeX *TFG Dpto. Informática - EPSJ - v1.0*