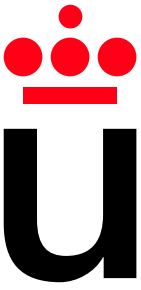


Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

**Grado en Ingeniería de Computadores**

**Curso 2023-2024**

**Trabajo Fin de Grado**

# DISEÑO E IMPLEMENTACIÓN DE UN SIMULADOR VISUAL DE UN PROCESADOR CON SEGMENTACIÓN

**Autor: Raúl Llona Cabeza**
**Tutor: Ángel Serrano Sánchez de León**

# Agradecimientos

A todas las personas que me han ayudado de cualquier forma durante estos meses de trabajo, en especial a:

Mi pareja, Lucía, por todos los momentos que me has apoyado incondicionalmente y en los que me has ayudado para que el trabajo salga adelante. Gracias por entrar en mi vida y estar a mi lado estos últimos años y ahora motivarme a terminar la carrera para graduarnos juntos.

Toda mi familia, especialmente a mis padres y mi hermano, por darme todo lo que necesito y he necesitado para progresar y formarme en estos estudios universitarios. Gracias por vuestro apoyo día tras día mientras estaba delante de un ordenador trabajando para conseguirlo.

# Resumen

En el ámbito de la ciencia de la computación es fundamental poseer conocimientos sobre la arquitectura de los procesadores, comprender su funcionamiento y estructura. Cualquier estudiante de este campo debe aprender sobre ello, otorgándole una comprensión más profunda sobre los fundamentos de la computación.

En este trabajo se ha desarrollado un simulador de un procesador que cuenta con una interfaz de usuario con el propósito de ayudar a estos estudiantes a asimilar el comportamiento de los procesadores. Se trata de una herramienta más para que tengan a su disposición y, a diferencia de otros recursos estáticos como libros y apuntes, se pueda interactuar con ella y se pueda visualizar de forma dinámica su funcionamiento.

Para el desarrollo de este simulador, se ha implementado un procesador MIPS multiciclo segmentado. La arquitectura MIPS cuenta con bastante popularidad y relevancia en la educación de la ingeniería de computadores, esto se debe principalmente a que proporciona un equilibrio adecuado entre simplicidad y funcionalidad. El diseño multiciclo permite que cada instrucción se divida en varios ciclos de reloj, mejorando la comprensión de cómo se desglosan y ejecutan las instrucciones en etapas. Además, la segmentación del procesador ofrece una visión detallada del funcionamiento del mismo, permitiendo a los estudiantes observar y analizar cada etapa de la ejecución de instrucciones. Este enfoque no solo facilita la comprensión teórica, sino que también proporciona una experiencia práctica interactiva y visual que enriquece el aprendizaje de la arquitectura de procesadores.

**Palabras clave**:

- Simulador
- Procesador MIPS
- Arquitectura de computadores
- Java

# Índice de contenidos

# Índice de figuras

# 1 Introducción

Este trabajo ha consistido en el desarrollo de una aplicación de escritorio que simule el funcionamiento de un procesador MIPS. El principal propósito del programa es ser una herramienta útil para estudiar la arquitectura y comportamientos de los sistemas de cómputo.

## 1.1. Contexto y alcance

La arquitectura de computadores forma parte del temario académico de los grados en ingeniería de computadores e informática y de otros estudios superiores. Este campo es esencial para entender como funcionan los computadores a bajo nivel, comprende al diseño conceptual y al sistema operacional fundamental de un ordenador. Incluye aspectos como la estructura de la unidad central de procesamiento (CPU), la organización y jerarquía de la memoria, la comunicación con los dispositivos de entrada y salida y la interconexión de los componentes mediante buses y direcciones. Tener una base sólida de conocimientos en arquitectura de computadores proporciona los fundamentos sobre los cuales se construyen la gran mayoría de aspectos de la ciencia de la computación. [1]

El funcionamiento de un procesador es sin duda la cuestión más esencial de la arquitectura de computadores. Los procesadores tienen un conjunto de instrucciones que pueden ejecutar y están compuestos por sub-componentes como la unidad de control, los registros o las unidades aritmético-lógicas. Los procesadores actuales son demasiado complejos como para utilizarlos para aprender sobre la arquitectura básica de una CPU, por este motivo, se emplean procesadores más simples como

los procesadores MIPS. Un procesador MIPS (Microprocessor without Interlocked Pipeline Stages) es un tipo de procesador basado en una arquitectura con un conjunto reducido de instrucciones, las cuales son simples y de longitud fija. [2]

En el caso concreto de la asignatura de Arquitectura de Computadores del grado en Ingeniería de computadores de la Universidad Rey Juan Carlos, el temario se imparte usando como ejemplo un procesador MIPS. Los alumnos de la asignatura aprenden sobre esta arquitectura tanto para los ejercicios como otras actividades prácticas. A partir de aquí es cómo surge este proyecto, desarrollar un simulador de un procesador MIPS que los estudiantes puedan usar como una herramienta de ayuda y para realizar las prácticas de dicha asignatura.

A grandes rasgos, la aplicación deseada debe ser capaz de simular el funcionamiento del procesador MIPS ajustándose a los contenidos específicos impartidos en la asignatura de Arquitectura de Computadores.

# 2

# Objetivos e investigación previa

## 2.1. Resumen de la arquitectura MIPS

Antes de definir todos los requisitos concretos que el simulador a construir debe cumplir, se dedicó tiempo a investigar sobre la arquitectura MIPS y sobre las características propias del modelo de procesador que se imparte en la asignatura de Arquitectura de Computadores. En las clases prácticas se estudia un modelo de procesador didáctico llamado coloquialmente "nanoMIPS", definido en el libro “Diseño y evaluación de arquitectura de computadoras”[3]. Esta arquitectura es una versión reducida del procesador MIPS que cuenta con varias mejoras.

Se trata de un procesador multiciclo segmentado, el cual divide cada instrucción de código ensamblador en distintas etapas o fases. Este tipo de procesadores poseen una arquitectura de carga-almacenamiento, los datos deben estar cargados en estructuras llamadas registros para poder utilizarse en las distintas operaciones del procesador. Existen dos bancos de registros diferenciados, uno para números enteros y otro para números en coma flotante. Por otro lado, el tamaño de palabra del procesador implementado en el simulador es de 32 bits.

La segmentación es una técnica que permite solapar la ejecución de distintas instrucciones al mismo tiempo. Esto se consigue dividiendo en secciones los recursos hardware por los que cada instrucción pasa en su camino de datos por el procesador. De esta manera, el trabajo que supone ejecutar una instrucción completa se divide en varias etapas a las cuales les corresponden dichas secciones hardware diferentes. La idea es que por cada ciclo de reloj del procesador, cada etapa ejecute una instrucción diferente, y cuando se pasa al siguiente ciclo, cada etapa pasa a

ejecutar la instrucción de la etapa anterior.

Las etapas de un procesador segmentado son:

- **Fetch (IF)**: Se busca la siguiente instrucción a ejecutar en la memoria de instrucciones según marque el contador de programa.
- **Decode (ID)**: Se decodifica la instrucción según el tipo de la misma, obteniendo los valores necesarios de los registros y preparando sus componentes para las unidades de ejecución.
- **Execute (EX)**: Se realiza el calculo aritmético de la operación indicada en la instrucción en la unidad aritmético-lógica (ALU).
- **Memory (MEM)**: Se accede a la memoria para obtener o guardar valores si la instrucción así lo indica.
- **Write Back (WB)**: Los resultados de la operación se escriben sobre los registros si la instrucción así lo indica.

Para garantizar que a lo largo de la ejecución de una instrucción no hay interferencias entre las etapas y no hay pérdidas de información se emplean los registros de segmentación. En estos registros se guardan los resultados de cada etapa al terminar un ciclo de reloj para que, en el siguiente ciclo, la siguiente etapa utilice como entrada dicha información almacenada.

En el nanoMIPS, estos registros de segmentación son 4 y se encuentran entre cada etapa: registro Fetch/Decode, registro Decode/Execute, registro Execute/Memory y registro Memory/WriteBack. El camino de datos que recorre cada instrucción durante su ejecución sería el siguiente:

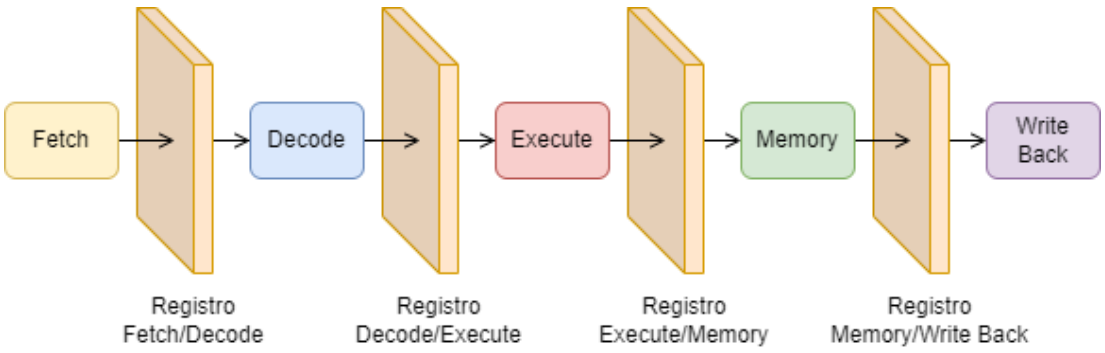



Figura 2.1: Esquema registros de segmentación (Imagen de elaboración propia)

Las instrucciones de ensamblador que se pueden ejecutar se dividen en 4 tipos:

- **Tipo Aritmético-lógicas**: Instrucciones que realizan operaciones matemáticas o lógicas sobre valores de los registros.
- **Tipo Memoria**: Instrucciones que acceden a la memoria del procesador.
- **Tipo Salto incondicional**: Modifican el valor del contador de programa para alterar la secuencia de instrucciones.
- **Tipo Salto condicional**: Igual que el tipo anterior, pero dependiendo si cumplen la condición indicada.

### 2.1.1. Operaciones con números en coma flotante

Las operaciones con números enteros se calculan en un único ciclo de reloj durante la etapa Execute. No obstante, este no es el caso para las operaciones con números decimales, los cuales se representan en la arquitectura MIPS mediante el sistema de representación numérica de coma flotante. Para realizar cálculos en dicho sistema, el procesador cuenta con estructuras hardware adicionales especializadas en cada tipo de operación, para las sumas y restas un sumador/restador, para las multiplicaciones un multiplicador y para las divisiones un divisor. Estas estructuras tardan más de un ciclo en realizar sus cálculos, esto supone que la etapa Execute de las instrucciones cuyos atributos manejan números de coma flotante tarden varios ciclos en ejecutarse.

### 2.1.2. Riesgos de datos

Para la correcta ejecución de las instrucciones, hay que tener en cuenta que las etapas Decode y Write Back ambas acceden a los registros, la primera para leer sus valores y la segunda para modificarlos. En el caso de que varias instrucciones consecutivas accedan al mismo registro, se pueden generar riesgos de datos. Existen cuatro tipos de dependencias de datos:

- **RAR (Read after Read)**: Ocurre cuando dos instrucciones consecutivas leen el valor de un mismo registro.
- **RAW (Read after Write)**: Ocurre cuando una instrucción lee el valor de un registro sobre el que otra instrucción ha escrito dicho valor con anterioridad.
- **WAR (Write after Read)**: Ocurre cuando una instrucción escribe el valor a un registro del que otra instrucción ha leído un valor anterior.

- **WAW (Write after Write)**: Ocurre cuando dos instrucciones consecutivas escriben sobre un mismo registro.

Para evitar riesgos de datos y que no se obtengan resultados equivocados, es imprescindible que el orden de acceso a los registros se preserve según indica el orden de instrucciones. En el caso del procesador nanoMIPS, sólo las dependencias RAW y WAW generan riesgos de datos. Los casos de riesgos WAW se pueden producir únicamente al ejecutar instrucciones con valores de coma flotante.

Los riesgos RAW ocurren cuando una instrucción trata de leer el valor de un registro sobre el que otra instrucción anterior aún no ha escrito su resultado. Por otro lado, los riesgos WAW ocurren cuando dos instrucciones consecutivas escriben sobre un mismo registro y la primera instrucción tarda más ciclos en ejecutarse que la segunda, de tal forma que el valor final que permanece en el registro es el de la primera instrucción, cuando lo correcto sería que fuese el valor de la segunda.

La forma mediante la cual se pueden solucionar estos riesgos es introduciendo ciclos de parada en el procesador, de tal manera que se espere a que la primera instrucción que provoca el riesgo acceda a los registros para que la segunda lo haga posteriormente. Sin embargo, la introducción de paradas no resulta óptimo ya que ralentiza la ejecución de instrucciones, perjudicando en la eficiencia del procesador.

Para el caso de las dependencias RAW, existe una técnica hardware que soluciona los riesgos que provocan y evita la necesidad de introducir paradas: los adelantamientos. Un adelantamiento consiste en pasar directamente el resultado obtenido de una instrucción a las instrucciones que lo necesitan como operando. De esta manera, la segunda instrucción implicada en un riesgo RAW no debe esperar a que la primera pase por su etapa Write Back y almacene su resultado en el registro indicado, sino que obtiene el resultado en el momento en el que el valor ha sido calculado en la etapa Execute o adquirido desde la memoria en la etapa Memory.

### 2.1.3. Riesgos estructurales

Durante la ejecución de un programa, puede darse el caso en el que varias instrucciones intentan acceder a la misma etapa o mismo componente hardware en el mismo ciclo de reloj. Cuando esto sucede, se produce un riesgo estructural, el cual se puede solucionar introduciendo paradas hasta que el componente quede liberado.

Para evitar que existan riesgos estructurales a raíz del hecho de que tanto en la etapa Decode como en la etapa Write Back se accede al banco de registros, en la primera mitad de la duración de un ciclo se realizan las escrituras mientras que en la segunda mitad se producen las lecturas.

Para el caso de las unidades de coma flotante, existe la posibilidad de segmentarlas internamente de tal forma que se puedan ejecutar varias operaciones simultáneamente, evitando tener que introducir paradas en el proceso.

### 2.1.4. Riesgos de control

Por último, existe otro tipo de riesgos que afectan a la ejecución de instrucciones, los riesgos de control. Las instrucciones de salto modifican el valor del contador de programa, este contador indica la dirección de la siguiente instrucción que se va a ejecutar. Las instrucciones de salto deben calcular el nuevo valor de dicha dirección durante su ejecución, por lo que el procesador no conoce inmediatamente qué instrucción siguiente debe ejecutar.

En el procesador nanoMIPS, el cálculo de la dirección destino del salto y, en el caso de instrucciones de salto condicional, la evaluación de la condición se realizan durante la etapa Decode. De esta forma el procesador sólo debe esperar un único ciclo extra para conocer la siguiente instrucción a ejecutar.

Con el objetivo de intentar omitir la necesidad de introducir dicha parada, existen varias formas de gestionar las instrucciones de salto condicional que tratan de predecir su resultado:

- **Predicción de salto no tomado**: Sin esperar a la resolución del salto, se comienza la ejecución de la siguiente instrucción siguiendo el orden del programa. Si la predicción acierta y el salto no hay que tomarlo, se habrá evitado introducir una parada. Por el contrario, si falla hay que cancelar la ejecución de dicha instrucción para comenzar al ejecución de la instrucción correcta.
- **Predicción de salto mediante estados**: Tienen en cuenta el historial de saltos anteriores para realizar su predicción. Si un salto se ha tomado con anterioridad, se predirá que se tome el salto y viceversa.

### 2.1.5. Diagrama de ciclos

Un diagrama de ciclos o diagrama de ejecución de instrucciones muestra el progreso de todas las instrucciones de un programa por sus etapas a lo largo del tiempo de ejecución de un programa.

En la siguiente página se muestra su estructura.

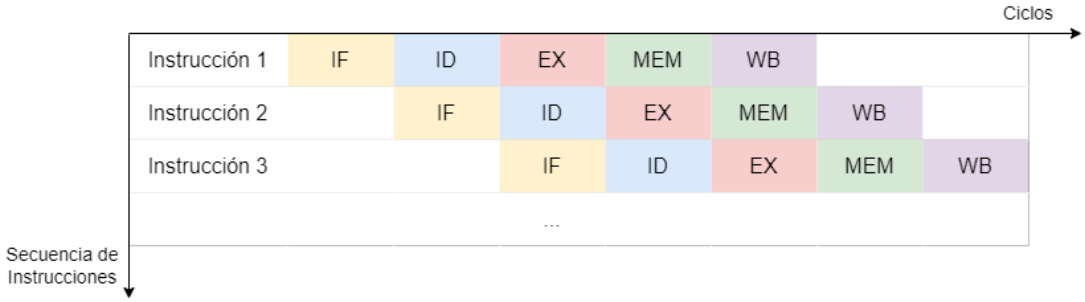



Figura 2.2: Estructura diagrama de ciclos (Imagen de elaboración propia)

El eje vertical contiene el orden de ejecución de las instrucciones del programa. Por otro lado, en el eje horizontal se muestra el tiempo de ejecución en ciclos. Cada casilla contiene la etapa de una instrucción ejecutada en un ciclo de reloj, de esta forma, todas las etapas de distintas instrucciones de una columna se ejecutan en el mismo ciclo.

## 2.2. Estado del arte

Actualmente, para las clases prácticas de la asignatura de Arquitectura de Computadores se utiliza el programa WinMIPS64, cuyas características se acercan considerablemente al temario de la asignatura pero cuenta con varias diferencias que provocan dificultades y confusiones en dichas clases prácticas. WinMIPS64 ha sido por lo tanto la referencia esencial para el desarrollo del proyecto, el objetivo principal es construir un simulador parecido que lo mejore y se ajuste fielmente al contenido de la asignatura.

### 2.2.1. WinMIPS64

WinMIPS64[4] es un simulador de un procesador MIPS de 64 bits segmentado que permite a los usuarios ejecutar programas escritos en lenguaje ensamblador y observar el recorrido de sus instrucciones por el camino de datos del procesador. Cuenta con una interfaz gráfica que muestra un diagrama de ciclos de las instrucciones, los estados de la memoria y los registros y estadísticas sobre la ejecución del programa.

Ciertas características del procesador son configurables dentro de la aplicación, además, cuenta también con un repertorio de instrucciones bastante amplio el cual puede ser consultado en un documento adjuntado al descargar el programa.

No obstante, cuenta con ciertas limitaciones que hacen que este simulador no sea óptimo para el desarrollo de la asignatura de Arquitectura de Computadores.

Las principales son:

- No permite exportar los diagramas generados a archivos de imagen.
- Los adelantamientos sólo se indican al ejecutar el programa ciclo a ciclo, y únicamente se señalan cambiando el color de los registros en las instrucciones involucradas. Resulta bastante confuso para el usuario apreciar cómo se producen.
- No permite activar y desactivar a elección del usuario la segmentación interna de las unidades de coma flotante. Posee por defecto dicha segmentación activada en el sumador/restador y en el multiplicador, mientras que el divisor no está segmentado. De esta manera no se pueden simular ejecuciones en las que dichas características varíen.
- No permite desactivar las predicciones de salto. Cuenta con dos sistemas entre los que elegir: la predicción de salto no tomado y la predicción de saltos mediante estados de 1 bit.

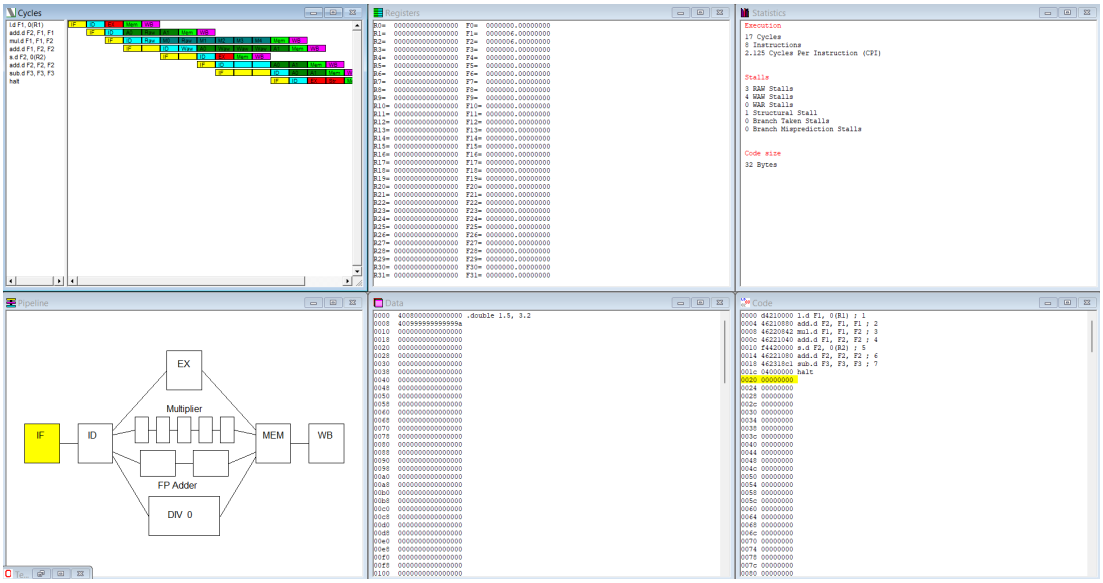



Figura 2.3: Interfaz de usuario de WinMIPS64
[4]

### 2.2.2. Otras aplicaciones

A parte de WinMIPS64, se han explorado más opciones existentes relacionadas con la arquitectura de computadores:

- **MARS MIPS simulator**[5]: Otro simulador de procesadores MIPS, en este caso de 32 bits, ejecuta código ensamblador y muestra los resultados por pantalla. Se muestra el camino de datos de las instrucciones en su versión uniciclo, es decir, cada instrucción se completa en un ciclo de reloj del procesador, no está segmentado en distintas etapas. En la aplicación se pueden observar los estados de los registros y de la memoria, también cuenta con opciones de depuración como una ejecución ciclo a ciclo y permite establecer puntos de ruptura. Sin embargo no genera ningún diagrama en el que se pueda observar el estado de las instrucciones en cada ciclo.

  Cuenta con una interfaz de usuario completa que permite a los usuarios escribir, editar, compilar y ejecutar programas en lenguaje ensamblador MIPS. Además, el editor de código reconoce y resalta la sintaxis y detecta posibles errores para su posterior compilación.

  Por otro lado, también soporta una variedad de llamadas de sistema que permiten realizar operaciones de entrada/salida, como leer y escribir en la consola, manipular archivos, y otros servicios del sistema que son útiles para escribir programas más complejos.

- **Simula3MS**[6]: Simulador de una arquitectura básica MIPS con varias versiones disponibles. Cuenta con tres opciones de simulación diferentes: entrada/-salida, técnicas de salto y camino de datos. Esta última opción permite escoger entre diferentes configuraciones del camino de datos: monociclo, multiciclo, segmentado básico, Marcador y algoritmo de Tomasulo. Implementa un subconjunto de instrucciones basadas en el repertorio de instrucciones del procesador MIPS R2000/R3000. Además, contiene un entorno de trabajo con interfaz gráfica que consta de un editor que permite analizar sintácticamente las instrucciones antes de su ejecución. También contiene una ventana de ejecución en la que puede seguir la evolución de las instrucciones por el camino de datos, así como de los registros y del segmento de datos.

- **Computer Science Field Guide MIPS Simulator**[7]: Simulador MIPS online, ejecuta código hexadecimal ensamblado por otra herramienta de la misma web. Contiene únicamente dos áreas de texto, una para introducir el código ensamblado, y otra que funciona como terminal, mostrando las salidas por pantalla y las trazas de los registros. Es útil para observar de forma exacta cómo al ejecutarse cada instrucción se vuelcan los resultados en hexadecimal al registro correspondiente. Esta herramienta forma parte de un proyecto de la Universidad de Canterbury, Nueva Zelanda.

### 2.2.3. Trabajos relacionados

Asimismo, se han investigado artículos y otros proyectos con un carácter similar a este, los cuales tienen como objetivo ayudar a estudiantes y usuarios a mejorar su

comprensión sobre la arquitectura de los procesadores.

- **Aprender Arquitectura de Computadores con la herramienta Simula3MS**[8]: Artículo de las universidades de Coruña y Santiago de Compostela que explica la metodología que emplean en las clases prácticas de las asignaturas de Arquitectura de Computadores al usar el simulador Simula3MS. Además, se realiza un análisis sobre el impacto que ha tenido el uso del simulador en la actividad docente.

  Al final concluye mostrando los beneficios que se obtienen al usar un simulador como Simula3MS en las aulas: Se permite a los estudiantes experimentar con instrucciones de carga, almacenamiento, y operaciones aritméticas y lógicas en un entorno controlado. Esto no solo hace las clases más dinámicas y atractivas, sino que también ayuda a construir una comprensión más sólida de los principios fundamentales de la arquitectura de computadores.

- **A Brick-by-Brick Approach to Learning MIPS Microarchitecture**[9]: En este trabajo se presenta un nuevo método de enseñanza con el objetivo de mejorar la comprensión conceptual de la arquitectura MIPS. Para demostrar su eficacia, dicho método se pone a prueba en varios casos de estudio descritos en el artículo. Se explica también los motivos de usar procesadores MIPS en el ámbito de la educación, siendo clave la simplicidad de su arquitectura para el entendimiento de los alumnos y su popularidad entre instituciones y universidades.

- **UCOMIPSIM 2.0: Pipelined MIPS Architecture Simulator**[10]: Este trabajo de la Universidad de Córdoba consiste en la creación de un simulador de la arquitectura MIPS que incluye un esquema de la circuitería del procesador. Al ejecutar un programa en este simulador, se puede observar cómo las instrucciones viajan por el camino de datos recorriendo el circuito. A través de esta visualización gráfica del circuito del procesador los alumnos pueden entender mejor su comportamiento y acercarse a los propios componentes hardware necesarios para su funcionamiento.

- **Pipelined MIPS Simulation: A plug-in to MARS simulator for supporting pipeline simulation and branch prediction**[11]: En este proyecto se ha desarrollado una extensión llamada “Pipelined Simulator and Branch Explainer” para la aplicación ya mencionada “MARS MIPS simulator”. Esta extensión permite a los usuarios de la aplicación simular la ejecución de programas en un procesador segmentado, algo que la versión original no contempla. De esta forma, el simulador MARS se siente más completo y con más utilidades para poder ser empleado en un mayor abanico de situaciones. Adicionalmente, la extensión incluye predictores de salto y la opción de generar el diagrama de ciclos que la aplicación base no realiza.

Todos estos ejemplos tienen en común el hecho de emplear MIPS con fines didácticos. Su uso en la educación proporciona una base clara y manejable para que los estudiantes comprendan los conceptos esenciales de la arquitectura de computadores, apoyándose en herramientas y simuladores diseñados específicamente para este fin. Esta metodología ha sido probada y adoptada en diversas instituciones académicas debido a su efectividad y a la simplicidad y claridad en la estructura de la arquitectura MIPS.

## 2.3. Descripción del problema

El propósito de este trabajo surge de las limitaciones observadas en la aplicación WinMIPS64. Aunque WinMIPS64 ofrece una herramienta útil para la simulación y el aprendizaje de la arquitectura MIPS, presenta diferencias y carencias en relación con el contenido específico que se aborda en la asignatura de Arquitectura de Computadores.

Para solucionar estas limitaciones, se ha propuesto desarrollar una nueva aplicación que se ajusta perfectamente al temario de la asignatura. Esta aplicación ha sido diseñada no solo para cubrir las diferencias entre WinMIPS64 y el material enseñado, sino también para maximizar el aprovechamiento de las clases prácticas.

La nueva herramienta desarrollada proporciona un entorno más alineado con las necesidades de la asignatura, permitiendo a los estudiantes una mejor comprensión y aplicación de los conceptos teóricos que se aprenden y se ponen en práctica durante su itinerario. Con esta aplicación, se espera mejorar significativamente la experiencia de aprendizaje y la adquisición de competencias en el ámbito de la arquitectura de computadores.

Tomando como inspiración las aplicaciones del estado del arte y teniendo en cuenta las características del procesador nanoMIPS, se han establecido una serie de requisitos que el producto final debe cumplir:

### 2.3.1. Requisitos funcionales

- La aplicación debe ser un simulador de un procesador MIPS.
- Debe ser capaz de reconocer y ejecutar código escrito en lenguaje ensamblador. En concreto, debe ser capaz de ejecutar un repertorio representativo del procesador MIPS, aunque no es necesario que el repertorio esté completo.
- Debe ajustarse a las características del nanoMIPS descritas en la asignatura de Arquitectura de Computadores mencionadas en el apartado 2.1.

- Debe mostrar la secuencia de instrucciones ejecutadas en un diagrama de ciclos.
- Debe mostrar el estado de los registros.
- Debe mostrar el estado de la memoria.
- Debe mostrar estadísticas sobre la ejecución.
- Debe incluir operaciones y registros de coma flotante.
- Debe incluir adelantamientos para solventar riesgos de datos y permitir activarse y desactivarse.
- Debe incluir distintas técnicas de resolución de instrucciones de salto.

### 2.3.2. Requisitos no funcionales

- La aplicación debe ser eficiente en términos de tiempo de ejecución y uso de recursos.
- La interfaz de usuario debe ser intuitiva y fácil de usar para facilitar la interacción con la aplicación.
- Debe ser compatible con los sistemas operativos y entornos de desarrollo utilizados en la asignatura.
- Debe garantizar que las simulaciones y las estadísticas de ejecución sean precisas y reflejen correctamente el comportamiento del procesador MIPS.

# 3

# Descripción informática

## 3.1. Planificación y metodología

Durante el desarrollo del proyecto se ha empleado la metodología de desarrollo software incremental. La aplicación ha sido construida mediante incrementos con los que se iban implementando distintas funcionalidades, permitiendo que la evolución del proyecto fuese escalonada. Cada vez que se implementaba una característica nueva a la aplicación se dedicaba tiempo a comprobar su funcionamiento y corregir los posibles errores encontrados.

A su vez, el total del trabajo realizado para la construcción del simulador fue planificado y divido en varias tareas clave:

- Investigación.
- Diseño de la aplicación.
- Programación.
- Testing.
- Lanzamiento de la aplicación.

Cada una de estas tareas clave fue a su vez dividida en otras tareas más concretas las cuales se fueron completando progresivamente a lo largo del desarrollo del proyecto. Estas tareas concretas eran apuntadas en una lista de pendientes por hacer, siendo marcadas una vez eran terminadas.

Durante el transcurso del trabajo, se realizaron varias sesiones de control junto al tutor y docente de la asignatura de Arquitectura de Computadores. En estas reuniones se mostraba el progreso realizado en el proyecto para poder recibir retroalimentación y se tomaron algunas decisiones de diseño. Los cambios sugeridos por el tutor se anotaban en la lista de tareas para su posterior implementación.

La duración total del proyecto es de unos 5 meses aproximadamente, se ha realizado un diagrama de Gantt para poder observar el progreso de las tareas claves a lo largo del tiempo:

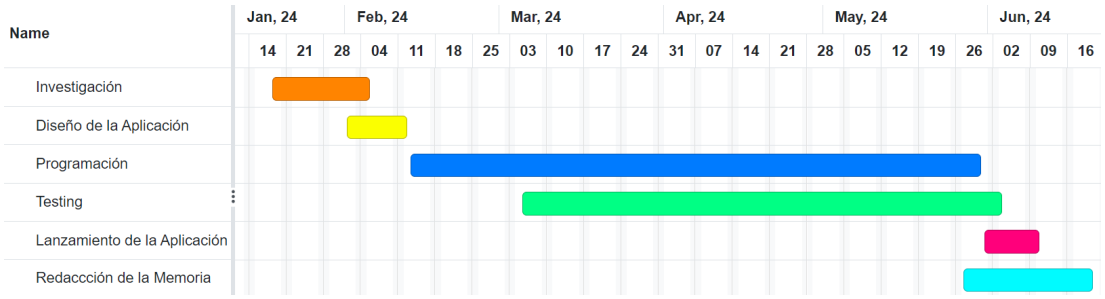



Figura 3.1: Diagrama de Gantt del proyecto (Imagen de elaboración propia)

## 3.2. Herramientas utilizadas

El lenguaje de programación seleccionado para desarrollar la aplicación ha sido Java[12]. Este lenguaje forma parte del paradigma de la programación orientada a objetos, el cual ha resultado esencial para construir la arquitectura del simulador. En cuanto al entorno de desarrollo escogido, este fue Eclipse IDE[13], ampliamente utilizado para el desarrollo de software en Java y particularmente con experiencia con el entorno.

Para la implementación de la interfaz de usuario de la aplicación se ha utilizado JavaFX[14], una plataforma de desarrollo de aplicaciones gráficas. JavaFX incluye una amplia gama de controladores y componentes ya definidos a disposición del desarrollador para crear sus propias interfaces gráficas. Estas interfaces se implementan en archivos FXML, un lenguaje basado en XML que permite definir la jerarquía de los elementos gráficos. Para diseñar de forma visual dichos ficheros FXML, se ha usado SceneBuilder[15], un editor de ficheros FXML para JavaFX.

Con el objetivo de mantener un registro del progreso del desarrollo de la aplicación y por cuestiones de seguridad, el proyecto se encuentra ubicado en un repositorio de GitHub[16]. GitHub ofrece la posibilidad de ir creando versiones a medida que el proyecto avanza, de tal manera que si una de estas versiones resulta malfuncionar siempre se puede regresar a la versión anterior. Además, es conveniente y altamente recomendable mantener el proyecto en un repositorio y no solo

en una carpeta local, en caso de cualquier fallo el repositorio sirve como copia de seguridad. Para realizar las subidas de nuevas versiones al repositorio desde el equipo personal se ha utilizado GitHub Desktop[17], una herramienta de escritorio que vincula el proyecto local con el repositorio de GitHub.

## 3.3. Diseño

El diseño de la interfaz gráfica de la aplicación está fuertemente influenciado por el de WinMIPS64 pero con ciertos cambios. Se quiso dar mucha más importancia al diagrama de ciclos ya que es la mejor herramienta para observar el progreso en la ejecución de las instrucciones. Además, desde el principio se decidió que la aplicación contase con un editor de texto incorporado en el que el usuario pueda escribir manualmente código en ensamblador, a diferencia de WinMIPS64, en el que solo se pueden cargar archivos de código para su ejecución sin poder editarlos.

Con estas premisas se realizó un boceto del concepto inicial de la aplicación:

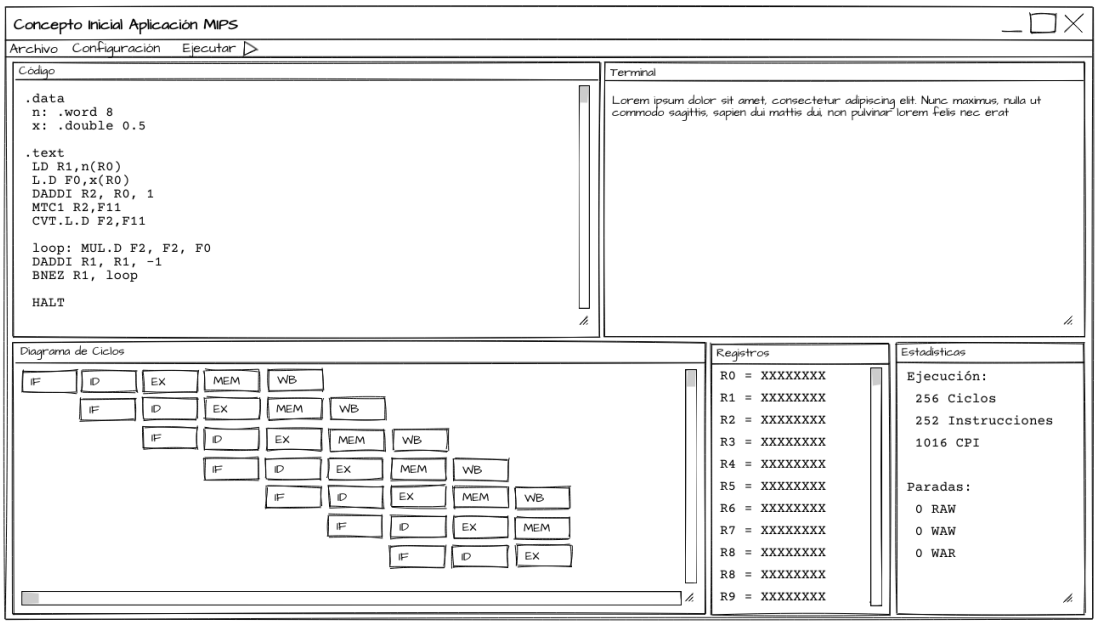



Figura 3.2: Concepto inicial de la aplicación (Imagen de elaboración propia)

Como se puede observar, el primer concepto contaba con 5 apartados diferenciados: el editor de código, una terminal, el diagrama de ciclos, los registros y las estadísticas de la ejecución. El principal cambio respecto a este concepto con la versión final de la aplicación fue desechar la terminal e incorporar una tabla que muestre el estado de la memoria. Se consideró que la relevancia de la terminal era muy escasa dado al público objetivo al que va dirigido el simulador, los estudiantes

de la asignatura de Arquitectura de Computadores. En las clases prácticas y los ejercicios apenas hay ejemplos de códigos en ensamblador que impriman resultados en la terminal y no forma parte del temario central de la asignatura. Por otro lado, poder comprobar el estado de la memoria sí resulta esencial para poder comprender el esquema total de la arquitectura del procesador y observar como las instrucciones que acceden a memoria funcionan.

Con dichos cambios implementados la interfaz de usuario final es la siguiente:

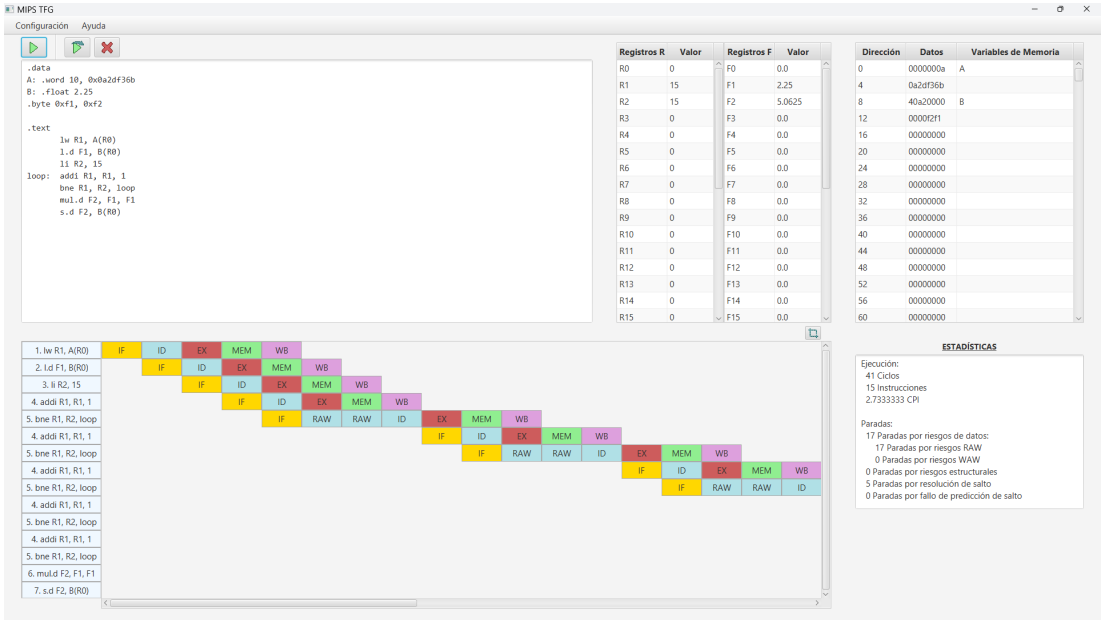



Figura 3.3: Interfaz gráfica final de la aplicación (Imagen de elaboración propia)

Se puede observar cómo los registros ahora están ubicados en la parte superior derecha junto a la tabla de la memoria. De esta forma, la parte inferior de la interfaz está principalmente ocupada por el diagrama de ciclos, acentuando aún más su importancia, manteniendo una pequeña sección para las estadísticas.

Entrando más en detalle en cada componente:

- Editor de código:

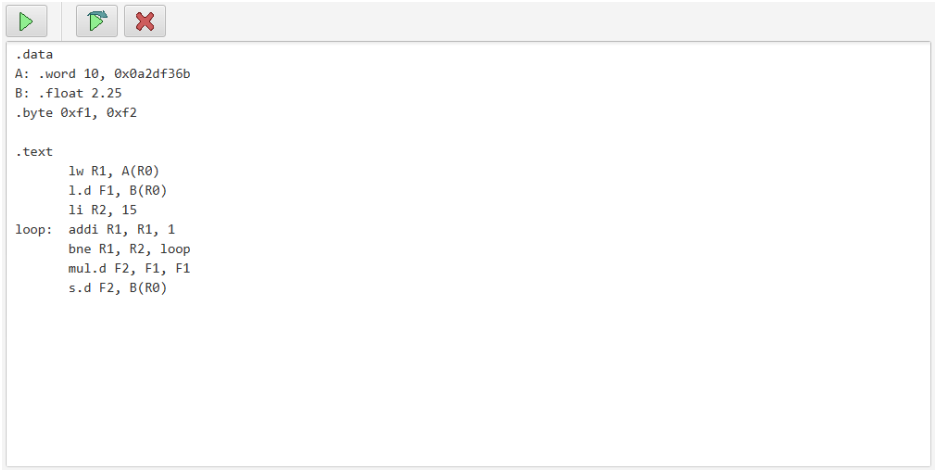
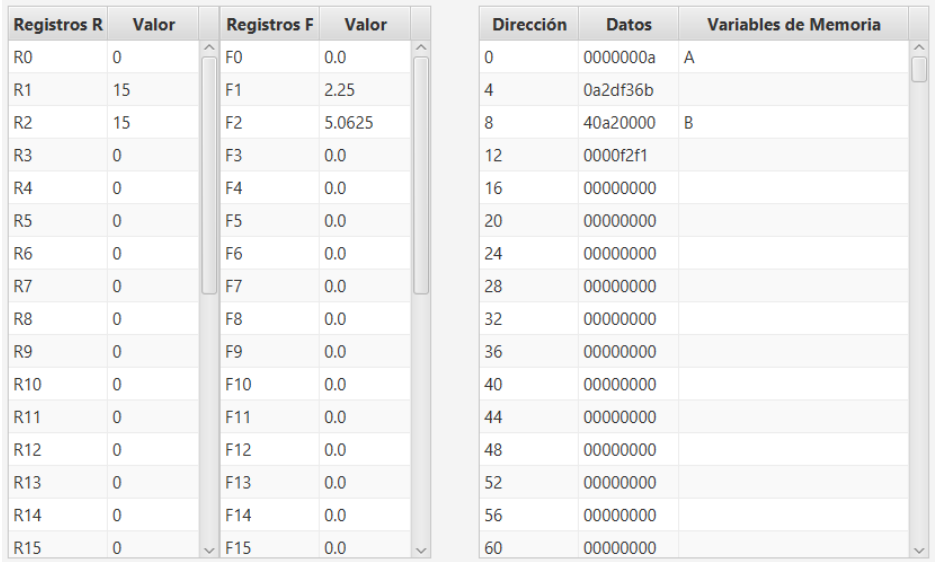



Figura 3.4: Interfaz del editor de código (Imagen de elaboración propia)

Cuenta con un cuadro de texto para el código en ensamblador, se puede escribir o pegar código directamente. Los botones de arriba sirven para ejecutar el código completo, ejecutarlo ciclo a ciclo y para cancelar la ejecución.

- Tablas de registros y memoria:



| Registros R | Valor |
|---|---|
| R0 | 0 |
| R1 | 15 |
| R2 | 15 |
| R3 | 0 |
| R4 | 0 |
| R5 | 0 |
| R6 | 0 |
| R7 | 0 |
| R8 | 0 |
| R9 | 0 |
| R10 | 0 |
| R11 | 0 |
| R12 | 0 |
| R13 | 0 |
| R14 | 0 |
| R15 | 0 |

| Registros F | Valor |
|---|---|
| F0 | 0.0 |
| F1 | 2.25 |
| F2 | 5.0625 |
| F3 | 0.0 |
| F4 | 0.0 |
| F5 | 0.0 |
| F6 | 0.0 |
| F7 | 0.0 |
| F8 | 0.0 |
| F9 | 0.0 |
| F10 | 0.0 |
| F11 | 0.0 |
| F12 | 0.0 |
| F13 | 0.0 |
| F14 | 0.0 |
| F15 | 0.0 |

| Dirección | Datos | Variables de Memoria |
|---|---|---|
| 0 | 0000000a | A |
| 4 | 0a2df36b | |
| 8 | 40a20000 | B |
| 12 | 0000f2f1 | |
| 16 | 00000000 | |
| 20 | 00000000 | |
| 24 | 00000000 | |
| 28 | 00000000 | |
| 32 | 00000000 | |
| 36 | 00000000 | |
| 40 | 00000000 | |
| 44 | 00000000 | |
| 48 | 00000000 | |
| 52 | 00000000 | |
| 56 | 00000000 | |
| 60 | 00000000 | |

Figura 3.5: Interfaz de los registros y la memoria (Imagen de elaboración propia)

Existen 2 tablas para los registros, una para los enteros y otra para los de coma flotante, muestran el valor en decimal de cada uno junto a su identificador. En la tabla de la memoria se observa la dirección de cada sección de 4 bytes de memoria, mostrando los datos en hexadecimal y con una columna dedicada a las variables de memoria que se pueden asignar a cada dirección.

- Diagrama de ciclos:

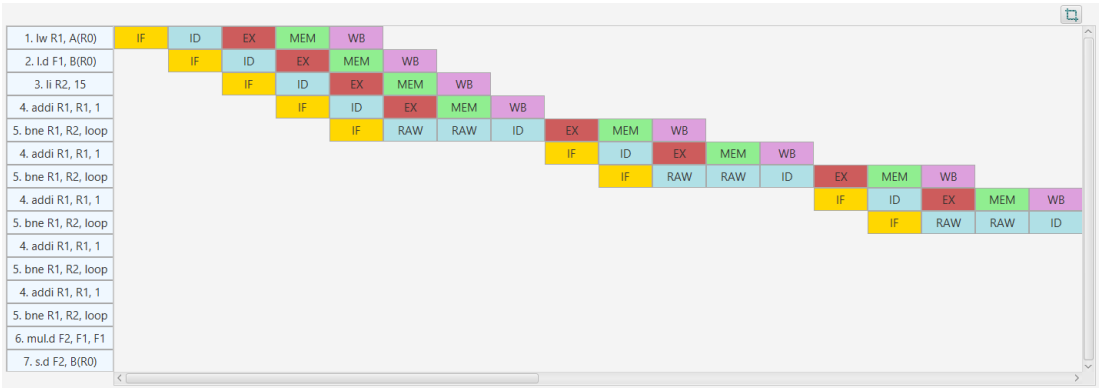



Figura 3.6: Interfaz del diagrama de ciclos (Imagen de elaboración propia)

La columna de la izquierda muestra en orden las instrucciones ejecutadas por el programa, con su índice y contenido. Las etapas de cada instrucción se generan en casillas, manteniendo el mismo código de colores de WinMIPS64, pero con tonos menos saturados que resultan más agradables visualmente. El pequeño botón situado en la parte superior derecha sirve para guardar la imagen del diagrama en el equipo en formato png.

- Estadísticas:

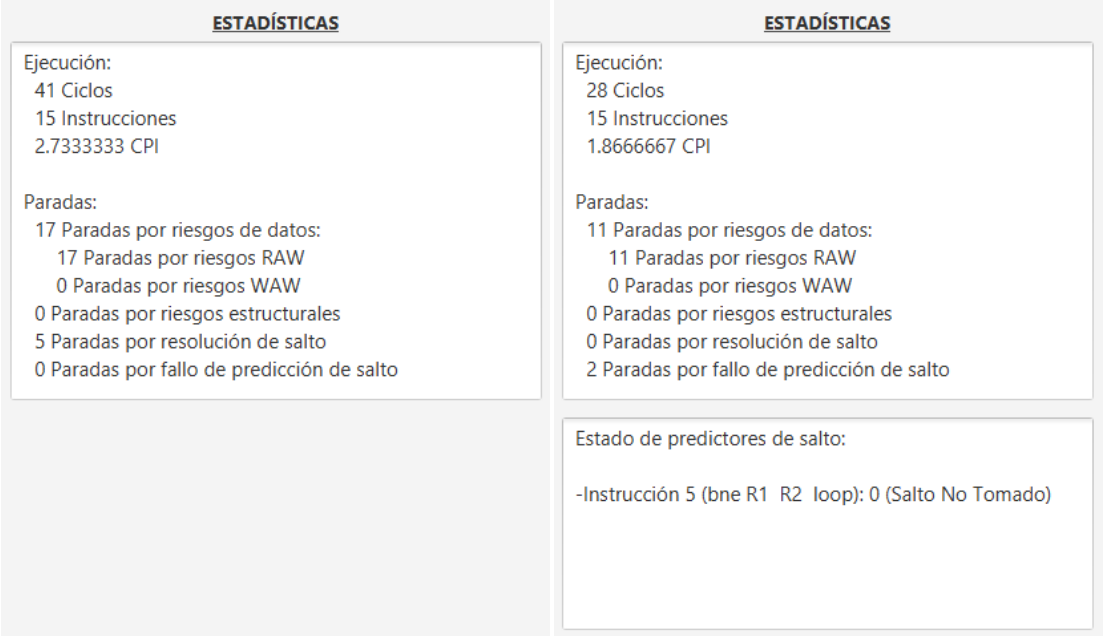



Figura 3.7: Interfaz de las estadísticas (Imágenes de elaboración propia)

Muestra datos relevantes a la ejecución del programa. El espacio vacío en la parte inferior se ocupa cuando el programa se ejecuta ciclo a ciclo con un predictor de saltos basado en estados activado.

- Barra de menús:

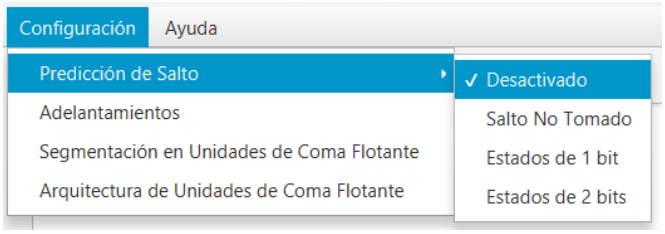



Figura 3.8: Interfaz de la barra de menús (Imagen de elaboración propia)

Situada en la parte superior de la aplicación, se puede acceder al menú de configuración, en el cual se pueden ajustar las características del procesador simulado, y al menú de ayuda desde el que el usuario puede consultar información importante sobre el uso de la aplicación.

La sección de “Predicción de Salto” del menú de configuración permite elegir al usuario entre las opciones de desactivar las predicciones, establecer la predicción de salto no tomado o implementar predicciones mediante estados de 1 o 2 bits.

La opción de “Adelantamientos” puede ser activada o desactivada al hacer clic sobre ella.

De igual manera, la sección de “Segmentación en Unidades de Coma Flotante” puede ser activada y desactivada según las preferencias del usuario.

Por último, al configurar la “Arquitectura de Unidades de Coma Flotante” se abre una ventana distinta en la que se puede modificar las duraciones en ciclos de reloj de dichas unidades, siendo el mínimo 2 ciclos y el máximo 10 para el sumador/restador, 20 para el multiplicador y 30 para el divisor:

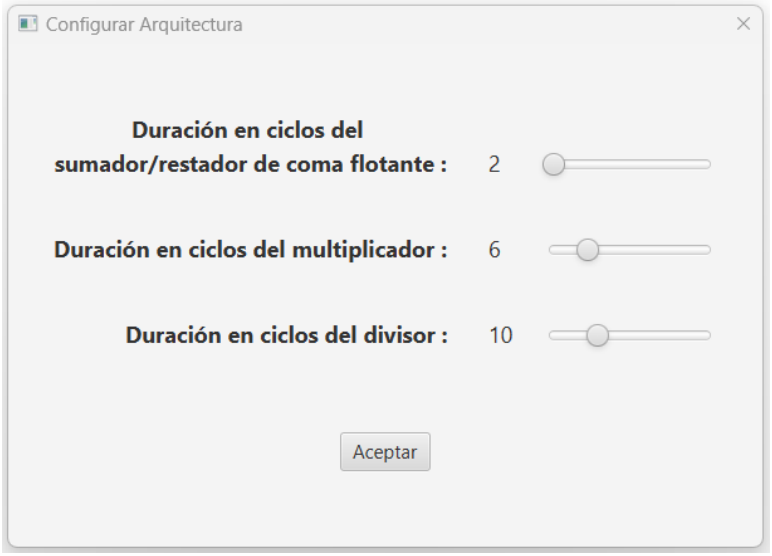



Figura 3.9: Interfaz de la configuración de las unidades de coma flotante
(Imagen de elaboración propia)

En el menú de ayuda se puede acceder a otra ventana que cuenta con un manual e información de la aplicación, el contenido completo de esta ventana se puede consultar en el apéndice A.


Figura 3.10: Interfaz de la ventana de ayuda (Imagen de elaboración propia)

## 3.4. Desarrollo

### 3.4.1. Arquitectura Modelo-Vista-Controlador

Para una correcta implementación de JavaFX en el proyecto, se ha empleado el patrón de diseño Modelo-Vista-Controlador (MVC, Model-View-Controller en inglés). Esta arquitectura es ampliamente utilizada en aplicaciones software que cuentan con una interfaz de usuario. El principal objetivo de este patrón es separar la lógica de la aplicación de su interfaz gráfica, para lograrlo, se divide en 3 componentes diferenciados:


Figura 3.11: Esquema arquitectura Modelo-Vista-Controlador (Imagen de elaboración propia)

- **Modelo**: Se encarga de ejecutar la lógica de la aplicación y encapsula la estructura de los datos.
- **Vista**: Representa la interfaz de usuario de la aplicación, presenta los datos al usuario y recibe sus interacciones.

- **Controlador**: Actúa de intermediario entre la vista y el modelo, se encarga de interpretar las interacciones del usuario que recibe desde la vista y consecuentemente ordena al modelo la lógica que debe ejecutar. También recoge el flujo de resultados desde el modelo para presentarlos en la vista.

Entrando en la propia arquitectura del simulador, los ficheros FXML de JavaFX componen la vista de la aplicación, solo definen la interfaz de usuario y no contienen lógica de la misma. Los controladores están implementados como clases de Java que importan los componentes necesarios de las librerías de JavaFX y están vinculados a los elementos de la vista. Y finalmente, el resto de clases de Java representan el modelo, estas no contienen ningún tipo de importación de JavaFX, la lógica que ejecutan es Java "puro".

Aplicar la arquitectura MVC separando la aplicación en los tres componentes consigue que dichos componentes tengan un propósito claro y diferenciado. Esto facilita el mantenimiento de la aplicación y la legibilidad del código, una persona que no tenga conocimientos de JavaFX pero sí de Java no tendrá ningún problema de entender la lógica del componente modelo. El software resultante es por tanto limpio y está modularizado.

### 3.4.2. JavaFX

Los archivos FXML que componen la interfaz de usuario son 3 para toda la aplicación, uno para la ventana principal que contiene todos los componentes mencionados en el apartado de diseño llamado "Main.fxml", y otros 2 para las ventanas auxiliares de ayuda y configuración de arquitectura, "HelpInfo.fxml" y "ArchitectureConfig.fxml" respectivamente.

Para hacer uso de las características de JavaFX, cada archivo FXML está vinculado a su propio controlador que ejecutan la lógica de la interfaz de usuario. No obstante, en el caso del fichero "HelpInfo.fxm" no es necesario que haya ningún controlador vinculado ya que este sólo contiene textos e imágenes, no cuenta con ningún elemento interactivo como un botón del que haya que gestionar la lógica que suceda al pulsarse. Por lo tanto, solo son necesarias las clases de java "MainController" y "ArchitectureConfigController".

### 3.4.3. Parseo del código ensamblador

Al pulsar uno de los botones que ejecutan el código introducido, el contenido del cuadro de texto se parsea línea a línea para gestionar los datos a inicializar en la memoria e introducir las instrucciones en un array de Strings. Para lograrlo, el texto se divide en las secciones de "data" y "text".

- En la sección “data” se introducen datos y variables en la memoria empleando determinadas directivas. Se parsean tanto números enteros como hexadecimales, los enteros son transformados a hexadecimal para introducirlos en la memoria.

- La sección “text”, también llamada “code”, contiene la secuencia de instrucciones con códigos de operación de ensamblador. Si el código de operación de una instrucción no se reconoce por el programa, se ignora dicha instrucción.

El repertorio de instrucciones incluidas en la aplicación es un subconjunto del total de las instrucciones MIPS existentes. Este limitado repertorio cuenta con las instrucciones más comúnmente usadas, asegurando que incluye las necesarias para el correcto desarrollo de las clases de Arquitectura de Computadores, permitiendo ejecutar la mayoría de ejemplos de códigos dados en la asignatura.

A continuación se muestran las instrucciones implementadas y su funcionamiento:

**Instrucciones aritmético-lógicas:**

- **add**: Suma de registros de enteros.

- **sub**: Resta de registros de enteros.

- **addi**: Suma de un registro entero y un inmediato.

- **li**: Carga de un inmediato a un registro entero (pseudo-instrucción que se transforma a addi del registro R0 y el inmediato).

- **and**: Operación AND lógica de registros enteros.

- **or**: Operación OR lógica de registros enteros.

- **xor**: Operación XOR lógica de registros enteros.

- **nor**: Operación NOR lógica de registros enteros.

- **add.d**: Suma de registros de coma flotante.

- **sub.d**: Resta de registros de coma flotante.

- **mul.d**: Multiplicación de registros de coma flotante.

- **div.d**: División de registros de coma flotante.

**Instrucciones de memoria:**

- **lw**: Carga de memoria de una palabra (32 bits) en un registro entero.
- **sw**: Guardado de un registro entero en memoria como una palabra (32 bits).
- **lb**: Carga de memoria de un byte (8 bits) en un registro entero.
- **sb**: Guardado de un registro entero en memoria como un byte (8 bits).
- **lh**: Carga de memoria de media palabra (16 bits) en un registro entero.
- **sh**: Guardado de un registro entero en memoria como media palabra (16 bits).
- **l.d**: Carga de memoria de una palabra (32 bits) en un registro de coma flotante.
- **s.d**: Guardado de un registro de coma flotante en memoria como una palabra (32 bits).

**Instrucciones de salto:**

- **j**: Salto incondicional a una instrucción marcada en el código ensamblador con una etiqueta.
- **beq**: Salto a una etiqueta bajo condición si los valores de los registros enteros son iguales.
- **bne**: Salto a una etiqueta bajo condición si los valores de los registros enteros no son iguales.

### 3.4.4. Funcionamiento del procesador simulado

La ejecución de la secuencia de instrucciones se produce ciclo a ciclo. En cada ciclo, las 5 etapas del procesador segmentado se ejecutan, recogiendo la información del registro de segmentación anterior a la etapa e insertando la nueva información procesada en el registro de segmentación siguiente. En el siguiente diagrama de flujo se puede visualizar el bucle que realiza el procesador por cada ciclo, se han nombrado a los registros de segmentación con las siglas de las dos etapas entre las que se encuentra, por ejemplo, el registro DE es sobre el que escribe la etapa Decode y de donde lee la etapa Execute:

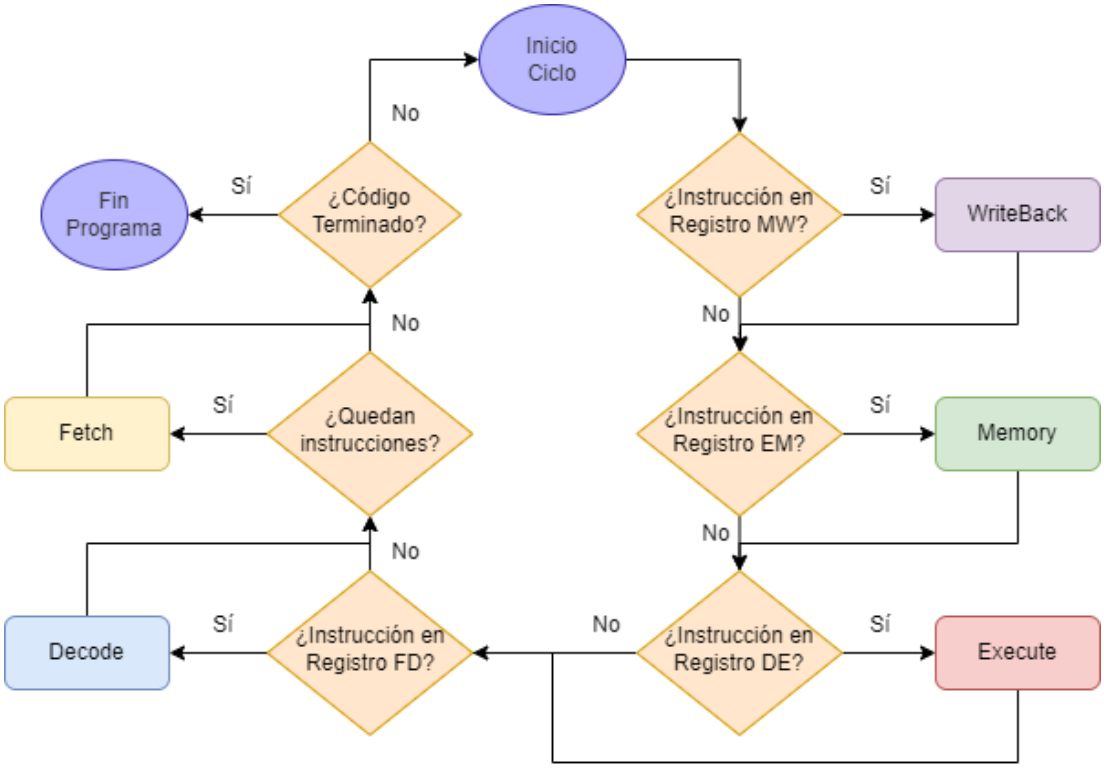



Figura 3.12: Diagrama de flujo de la ejecución por ciclos de un programa
(Imagen de elaboración propia)

Con este concepto, cada ciclo de reloj permite que una instrucción avance a la siguiente etapa. El orden de ejecución de las etapas en el bucle parece desafiar el sentido común, la etapa Fetch debería ser la primera ya que es la primera por la que pasa cada instrucción. Sin embargo, el orden inverso establecido es necesario para la correcta ejecución de cada ciclo, por cada etapa se comprueba si el registro anterior contiene una instrucción, y si es así, se ejecuta. De esta manera, en el primer ciclo de la ejecución de un programa, solo se ejecuta la etapa Fetch de la primera instrucción, y a partir de ahí, al escribir la instrucción en el registro FD, se irá propagando en cada repetición del bucle por el resto de etapas.

En el momento en el que la última instrucción del programa haya terminado su ejecución al pasar por la etapa Write Back el código habrá finalizado y el bucle se detiene.

### 3.4.5. Distribución de clases

El proyecto Java completo cuenta con alrededor de una veintena de clases, cada una cumple una función determinada. En el siguiente diagrama de clases se puede visualizar cómo están distribuidas (se han omitido varias clases y atributos en pos

de facilitar la comprensión y legibilidad del diagrama):

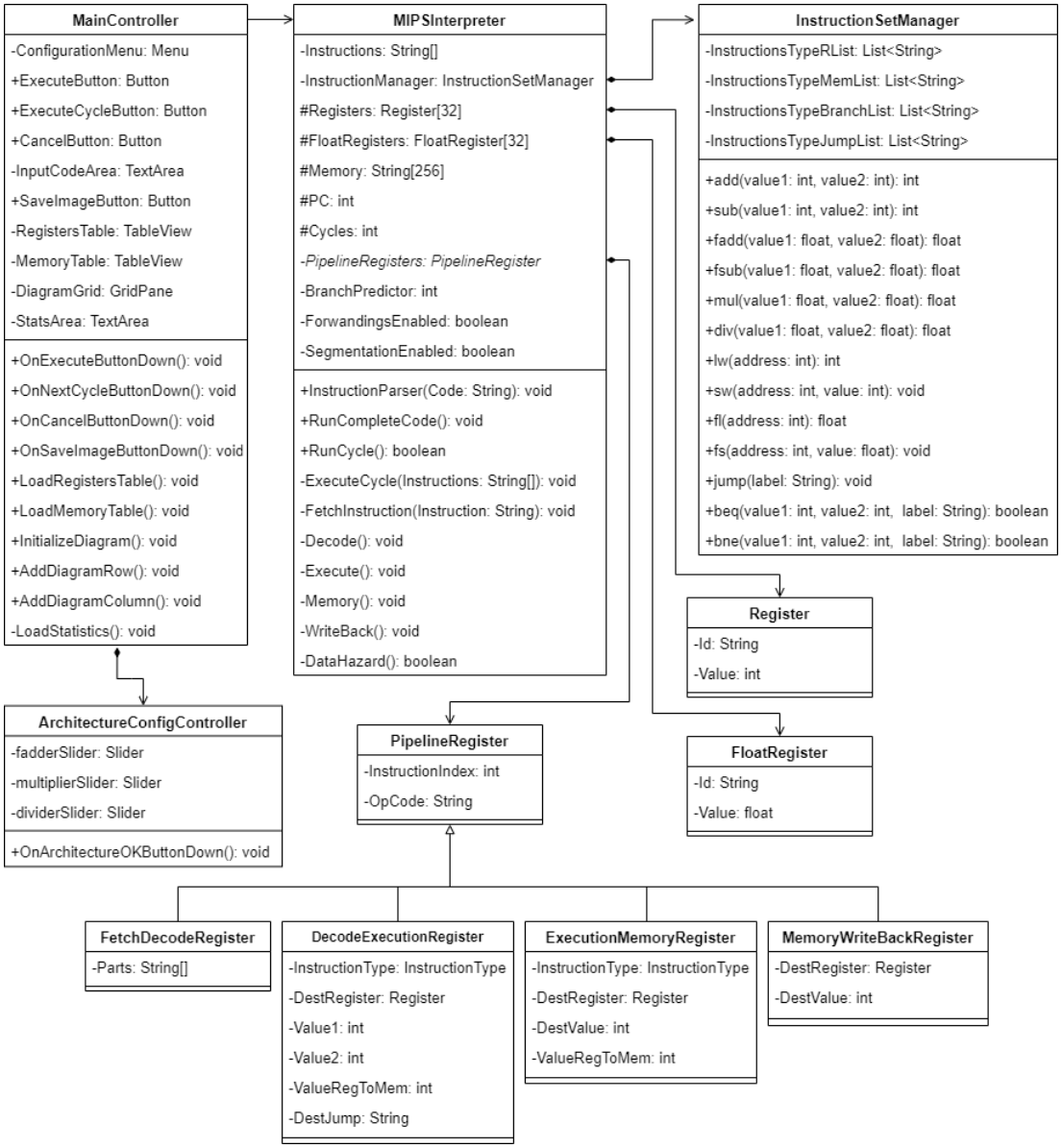



Figura 3.13: Diagrama de clases del proyecto (Imagen de elaboración propia)

### Clase MIPSInterpreter

Se trata de la clase principal que contiene el bucle de ejecución de cada ciclo y el método que parsea e interpreta las instrucciones en lenguaje ensamblador de MIPS. Esta clase se puede considerar como el cerebro de la aplicación, ya que es aquí donde realmente se simula el comportamiento del procesador MIPS.

A continuación se muestra la implementación del concepto del bucle de ciclos

en pseudocódigo que contiene esta clase:

```
mientras (codigoNoTerminado) {
    // Etapa WriteBack
    Si (registroMW.indiceInstruccion != -1):
        WriteBack()

    // Etapa Memory
    registroMW.indiceInstruccion = registroEM.indiceInstruccion
    Si (registroEM.indiceInstruccion != -1):
        Memory()

    // Etapa Execute
    registroEM.indiceInstruccion = registroDE.indiceInstruccion
    Si (registroDE.indiceInstruccion != -1):
        Execute()

    // Etapa Decode
    registroDE.indiceInstruccion = registroFD.indiceInstruccion
    Si (registroFD.indiceInstruccion != -1):
        Decode()

    // Etapa Fetch
    Si (pc < instrucciones.longitud):
        registroFD.indiceInstruccion = pc
        instruccion = instrucciones[pc]
        Fetch(instruccion)
        pc = pc + 1
    Sino:
        registroFD.indiceInstruccion = -1

    ciclos = ciclos + 1
}
```

Como se puede observar, por cada iteración del bucle se ejecuta un ciclo del procesador. Para comprobar si una etapa debe ejecutarse se comprueba si el índice de instrucción del registro de segmentación correspondiente es válido (distinto a -1).

La etapa Fetch va introduciendo en la cadena de ejecución las instrucciones que obtiene del array parseado según el contador de programa (pc, por sus siglas en inglés “Program Counter”), el cual también se va incrementando. En el caso de que no queden más instrucciones que ejecutar, introduce en el registro de segmentación FD el índice no válido -1.

#### Clase InstructionSetManager

La clase “MIPSInterpreter” contiene una instancia de esta clase, se encarga de ejecutar las operaciones de las instrucciones. Contiene 4 listas con los códigos de operación de cada tipo de instrucción y los métodos que ejecutan dichas instrucciones. La clase “MIPSInterpreter” llama a estos métodos en la etapa Execute para ejecutar la operación correspondiente a la instrucción que se encuentre en dicha etapa.

#### Clase PipelineRegister

Se trata de una clase abstracta de la que heredan todos los registros de segmentación del simulador. Contiene los campos imprescindibles que comparten todos estos registros: el índice de instrucción, que indica la posición en la secuencia de instrucciones del programa, y el código de operación de la instrucción.

El resto de campos de las clases hijos de “PipelineRegister” varían según la información necesaria para el funcionamiento de las etapas entre las que se encuentran:

- La clase “FetchDecodeRegister” contiene las partes de la instrucción que han sido generadas por la etapa Fetch. Por ejemplo, la instrucción “add R3, R1, R2” se divide en las cuatro partes [“add”, “R3”, “R1”, “R2”].
- La clase “DecodeExecutionRegister” almacena los valores que la etapa Decode decodifica, usando el mismo ejemplo: reconociendo el tipo al que la instrucción pertenece (Tipo R), el registro destino sobre el que se escribirá el resultado de la operación (R3) y obteniendo los valores a sumar de los registros R1 y R2. En el caso de una instrucción de guardado en memoria, se almacena el valor del registro que sé escribirá en la memoria. En el caso de una instrucción de salto, se guarda la etiqueta que contiene la dirección del salto.
- La clase “ExecutionMemoryRegister” guarda el valor destino resultante de haber sumado los valores de los registros R1 y R2, y propaga el resto de campos. En el caso de la instrucción de guardado en memoria se propaga el valor del registro para que se escriba en la etapa Memory.
- La clase “MemoryWriteBackRegister” propaga el registro destino y el valor que se va a escribir en él para que lo realice la etapa Write Back.

#### Clase MainController

Es el controlador principal que gestiona los componentes de JavaFX como ya se ha mencionado anteriormente. Su principal tarea es ejecutar la lógica de los eventos producidos a raíz de las interacciones del usuario.

Cada botón de la interfaz está vinculado a un método propio. El botón de ejecución llama al método “RunCompleteCode()” de la clase “MIPSInterpreter”, el cual ejecuta el bucle de ciclos del programa entero. Por otro lado, el botón de ejecución por ciclos llama al método “RunCycle()” cada vez que es pulsado, el cual ejecuta un único ciclo del bucle del programa. El botón de cancelar reinicia la ejecución del programa y limpia la interfaz de usuario (tablas, diagrama y estadísticas). El último botón, el que guarda una imagen del diagrama de ciclos, abre el explorador de archivos del equipo del usuario para que seleccione la ubicación y el nombre de la imagen a guardar.

En cuanto al diagrama de ciclos, es importante definir cómo se genera. En cada ciclo del programa se renderiza una columna del diagrama, mostrando las etapas que se hayan ejecutado. Por otro lado, las filas se generan cada vez que una instrucción nueva comienza a ejecutarse, esto no ocurre necesariamente en cada ciclo. Cuando se genera una fila nueva, la instrucción correspondiente se renderiza en la primera columna del diagrama que muestra la secuencia de instrucciones. El tamaño del contenedor del diagrama se va ajustando a medida que este crece, permitiendo al usuario hacer “scroll” por el contenido tanto vertical como horizontalmente.

### 3.4.6. Implementación de operaciones de coma flotante

En el procesador nanoMIPS las operaciones con números de coma flotante se realizan en unidades hardware específicas. Estas unidades hardware de coma flotante son un sumador/restador, un multiplicador y un divisor. Las instrucciones cuyo código de operación corresponde a una operación de coma flotante utilizan otros registros específicos también, que admiten valores decimales.

A diferencia de las operaciones enteras que se ejecutan en un ciclo de reloj en la etapa Execute, las operaciones de coma flotante se demoran más de un ciclo dependiendo de la arquitectura de cada unidad. En estos casos, la abreviatura de la etapa cambia según el tipo de operación a “A”, para el caso del sumador/restador, “MUL”, para el multiplicador, y “DIV” para el divisor. En el siguiente ejemplo de diagrama de ciclos se puede observar la ejecución de los 3 tipos de operaciones de coma flotante:

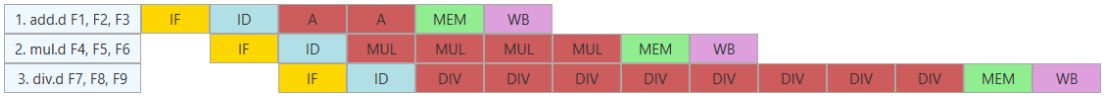



Figura 3.14: Diagrama con operaciones de coma flotante
(Imagen de elaboración propia)

En este caso la suma tarda 2 ciclos, la multiplicación 4 y la división 8. Pero como se mencionó en la sección de diseño, estas duraciones se pueden modificar a

preferencia del usuario 3.9.

El hecho de que estas operaciones tarden varios ciclos impone la necesidad de crear nuevos registros de segmentación para no perder la información de la instrucción a lo largo de su ejecución. Estos nuevos registros son propios de cada unidad de coma flotante, en el siguiente esquema se puede observar el camino de datos que recorre una instrucción de coma flotante, usando como ejemplo una multiplicación:

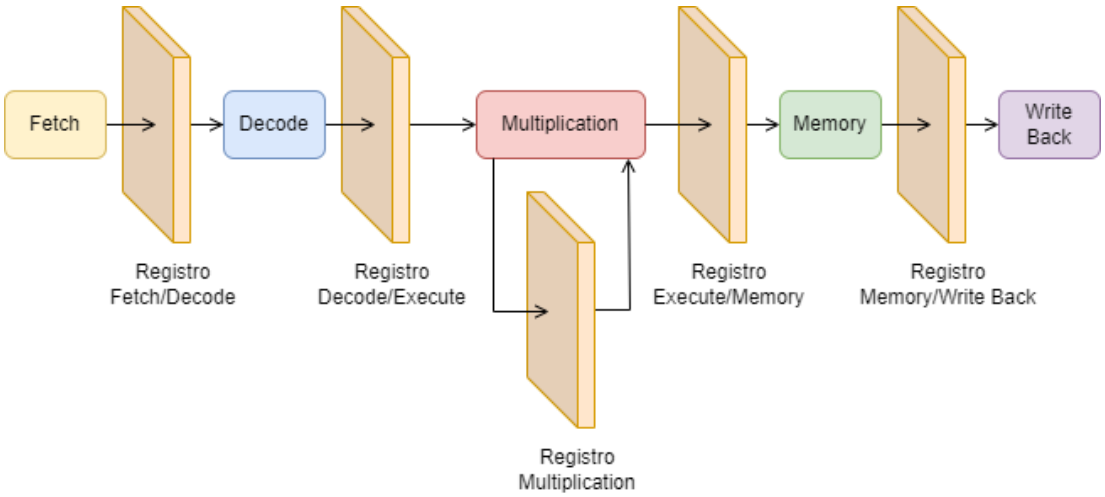



Figura 3.15: Esquema registros de segmentación para operadores de coma flotante (Imagen de elaboración propia)

La multiplicación accederá al registro de segmentación del multiplicador tantas veces como ciclos adicionales se necesiten para realizar dicho calculo. Si la multiplicación tarda 2 ciclos, se accederá una vez, si tarda 3, accederá 2 veces, etc. Esto es igual para el sumador/restador y el divisor, cada uno cuenta con su registro de segmentación adicional.

Estos registros están implementados como atributos en la clase "MIPSInterpreter" y heredan, al igual que el resto de registros de segmentación, de la clase abstracta "PipelineRegister". Su contenido es muy similar al registro Execute/Memory, pero sus valores son float en vez de int.

### 3.4.7. Implementación de detección de riesgos de datos

La detección de riesgos de datos se realiza durante la etapa Decode, hay 2 tipos de riesgos que pueden comprometer la correcta ejecución de programas en el procesador nanoMIPS:

- Riesgos **RAW** (Read After Write): Se producen cuando una instrucción trata de leer el valor de un registro sobre el que otra instrucción anterior aún no ha

escrito su resultado. Para solucionarlo se introducen paradas en el procesador entre las etapas Fetch y Decode hasta que la primera instrucción escriba el valor en el registro en la etapa Write Back. En el siguiente ejemplo se puede observar cómo la segunda instrucción debe esperar a que la primera escriba en el registro R3 para poder leerlo, introduciendo dos paradas antes de terminar la etapa Decode:

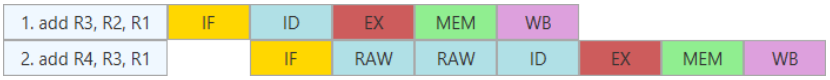



Figura 3.16: Diagrama de riesgo RAW (Imagen de elaboración propia)

- Riesgos **WAW** (Write After Write): Surgen a raíz de la incorporación de las operaciones de coma flotante por sus duraciones diversas. Se producen cuando una instrucción trata de escribir un valor en un registro sobre el que otra instrucción anterior aún no ha escrito su resultado. Para solucionarlo también se introducen paradas hasta que sea seguro que la segunda instrucción pase por la etapa Write Back después de la primera. Al igual que con los riesgos RAW, las paradas WAW se colocan entre las instrucciones Fetch y Decode. Por otro lado, en caso que se produzca a la vez un riesgo RAW y un riesgo WAW, se escribirá con prioridad en riesgo RAW. En este otro ejemplo se puede observar cómo la segunda instrucción espera a que la primera haya escrito en F3, introduciendo dos paradas antes de terminar la etapa Decode. Sin dichas paradas la segunda instrucción ejecutaría su etapa Write Back antes que la primera instrucción:

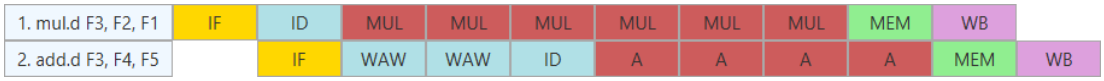



Figura 3.17: Diagrama de riesgo WAW (Imagen de elaboración propia)

Para detectar estos riesgos la etapa Decode llama al método “dataHazard()”, el cual realiza comparaciones entre los registros de la instrucción que se encuentra en la etapa Decode con los registros de las instrucciones que se encuentran en las etapas posteriores. En el caso que se detecte una coincidencia se produce una parada.

Las paradas bloquean el avance de las instrucciones de la etapa Decode y Fetch, dando tiempo a que las instrucciones del resto de etapas continúen su ejecución. Para introducir dichas paradas se emplea el índice de instrucción no válido -1.

### 3.4.8. Implementación de adelantamientos

Los adelantamientos ofrecen una solución para evitar las paradas en el procesador por riesgos RAW, consisten en, como su propio nombre indica, adelantar el

valor resultado de la primera instrucción implicada a la segunda sin la necesidad de haberse registrado en la etapa Write Back. El valor se obtiene del registro de segmentación correspondiente a la etapa en el que se calcula, pudiendo ser una operación aritmético-lógica calculada en la etapa Execute o una operación de carga de memoria de la etapa Memory. Dicho valor se asignará a la instrucción que se encuentra en la etapa Decode cuando se detecta que se ha producido un riesgo RAW.

Para poder visualizar los adelantamientos en el diagrama de ciclos, se genera una flecha que viaja desde la etapa en la que se obtiene el valor resultado hasta la etapa en la que dicho valor se aplica. El siguiente ejemplo se trata del mismo caso de riesgo RAW aplicando un adelantamiento para evitar las paradas:

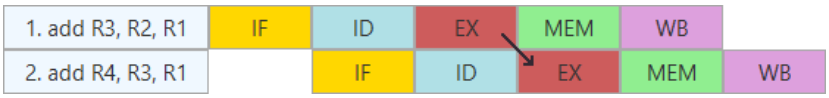



Figura 3.18: Diagrama de adelantamiento (Imagen de elaboración propia)

Cabe aclarar que el adelantamiento se debe producir entre dos ciclos consecutivos, pero no necesariamente entre dos instrucciones consecutivas en el código fuente. Como excepción, se puede hacer con una distancia de 2 ciclos si se hace desde MEM en vez de desde EX. En cuanto al destino del adelantamiento, es ID si la instrucción receptora es un salto condicional, EX si es una aritmético lógica entera o una de carga o almacenamiento en el caso del registro que se suma al inmediato, A/MUL/DIV si es una aritmético lógica de coma flotante, o MEM si es el registro que se escribe en memoria en un almacenamiento.

### 3.4.9. Implementación de detección de riesgos estructurales

Otro tipo de riesgos que pueden surgir durante la ejecución de un programa son los riesgos estructurales, estos se producen cuando dos o más instrucciones distintas tratan de acceder al mismo recurso hardware. Para solucionarlo también es necesario introducir paradas. En el siguiente ejemplo, ambas instrucciones necesitan acceder al sumador/restador de coma flotante que no está segmentado, por lo que la segunda instrucción debe esperar a que la primera deje el recurso libre:

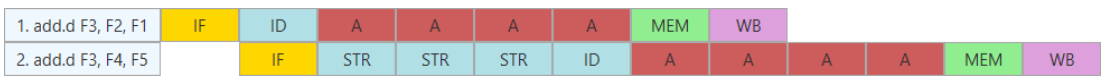



Figura 3.19: Diagrama de riesgo estructural (Imagen de elaboración propia)

### 3.4.10. Implementación de segmentación de unidades de coma flotante

Para el caso de los riesgos estructurales causados por las unidades de coma flotante existe una solución para no provocar paradas. Esta solución consiste en segmentar internamente dichas unidades, al igual que el procesador entero está segmentado, se pueden seccionar para permitir la ejecución simultánea de varias operaciones.

Con el objetivo de segmentar correctamente el sumador/restador, el multiplicador y el divisor es necesario que cuenten con más registros de segmentación adicionales, tantos como la duración en ciclos de la operación. De esta manera, las unidades pueden contener la información de todas las instrucciones que estén ejecutando. Para lograrlo, cuando la segmentación está activada se inicializan listas de registros de segmentación que heredan de la clase "PipelineRegister" para cada unidad de coma flotante con sus tamaños correspondientes.

El resultado permite ejecutar instrucciones de coma flotante simultáneamente:

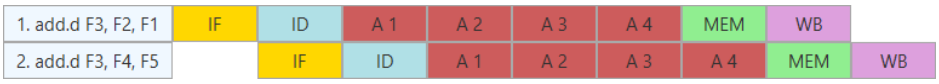



Figura 3.20: Diagrama de unidades de coma flotante segmentadas
(Imagen de elaboración propia)

### 3.4.11. Implementación de predictores de salto

Existen varios métodos de gestionar las instrucciones de salto condicional en procesadores MIPS, en el temario de Arquitectura de Computadores principalmente se imparten 4 tipos. Estos 4 tipos han sido implementados en el simulador para que el alumno pueda escoger cuál prefiere para la ejecución de un programa, de esta manera se cubre una mayor cantidad de casos de ejercicios prácticos.

Las instrucciones de salto modifican el valor del contador de programa (pc) para dictaminar la siguiente instrucción que debe comenzar a ejecutarse en su etapa Fetch.

#### Predicción de saltos desactivada

Bajo esta opción el procesador no trata de predecir el comportamiento del salto, espera a su resolución para decidir cual será el índice de la siguiente instrucción. Con el objetivo de no introducir demasiadas paradas, la decisión de la toma del salto en este tipo de instrucciones se realiza al final de la etapa Decode. De esta forma el procesador solo necesita introducir un ciclo de parada. El índice de la siguiente

instrucción será el que contiene la etiqueta indicada si se toma el salto o el de la siguiente instrucción por orden de secuencia si no se toma. En el siguiente ejemplo se puede observar como la instrucción toma el salto con un ciclo de retardo por la parada:

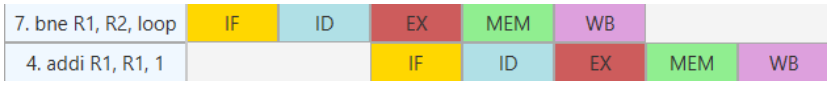



Figura 3.21: Diagrama de predicción de salto desactivada
(Imagen de elaboración propia)

### Predicción de salto no tomado

Esta opción establece que en todas las instrucciones de salto se prediga que no se va a tomar, comenzando inmediatamente la ejecución de la instrucción siguiente por orden de secuencia. El salto se resuelve realmente un ciclo más tarde al final de la etapa Decode y, en el caso que la predicción falle y haya que tomar el salto, la instrucción introducida será cancelada para comenzar a ejecutar la instrucción correcta. Para implementar dicha cancelación se vuelve a emplear el índice no válido -1 en el registro que contiene la instrucción a cancelar, de esta manera el resto de etapas no la ejecutarán. En el siguiente diagrama se puede observar el caso en el que la predicción falla y se cancela la ejecución de la instrucción:

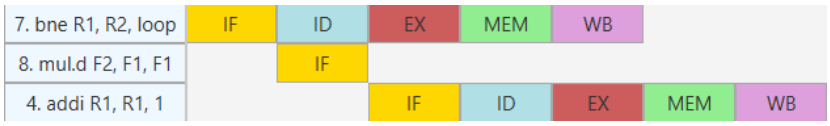



Figura 3.22: Diagrama de predicción de salto no tomado
(Imagen de elaboración propia)

**Predicción de salto mediante estados de 1 bit**

Con esta opción seleccionada se hace uso de una nueva estructura que guarda información de todos los saltos registrados de un programa. Esta estructura es un "buffer" que guarda para cada instrucción de salto su índice, el índice de la instrucción a la que se salta cuando el salto es tomado y el bit que marca la predicción de tomar o no el salto.

El valor de dicho bit se actualiza según el siguiente diagrama de estados:

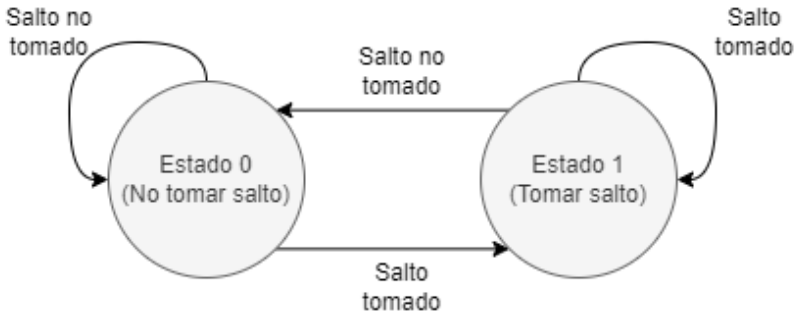



Figura 3.23: Diagrama de estados del predictor de salto de 1 bit
(Imagen de elaboración propia)

Este "buffer" está implementado en la clase "BranchTargetBuffer", se accede a él durante la etapa Fetch de las instrucciones de salto cuyo índice coincide con alguno presente en su estructura. En ese momento, se asigna al pc el índice de la instrucción siguiente a ejecutar según indique el bit del estado. El punto de partida de cualquier instrucción es el estado con valor inicial 0 (No tomar salto).

Cuando realmente se resuelve el salto al final de la etapa Decode, si el predictor ha fallado, se cancela la instrucción introducida al igual que en la opción anterior, comenzando la ejecución de la instrucción correcta.

**Predicción de salto mediante estados de 2 bits**

El funcionamiento es idéntico al caso anterior, pero con un sistema de estados de 2 bits:

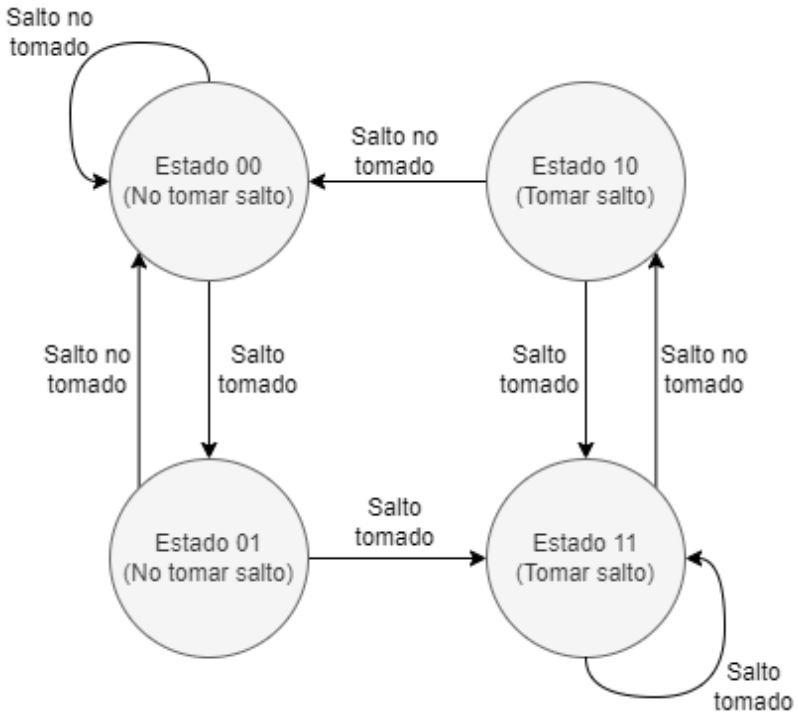



Figura 3.24: Diagrama de estados del predictor de salto de 2 bits
(Imagen de elaboración propia)

De esta manera, cuando en una instrucción se ha tomado 2 veces el salto con anterioridad, seguirá prediciendo tomar el salto aunque se haya equivocado en una ocasión. Es necesario que se vuelva a producir el mismo resultado 2 veces seguidas para cambiar la predicción. Por otro lado, el punto de partida es el estado con valor inicial 01 (No tomar salto).

## 3.5. Lanzamiento

Finalmente, con todas características implementadas en el proyecto, llegó el momento de exportar la aplicación completa a un ejecutable para que pueda ser distribuido en un futuro.

Esta tarea resultó ser mucho más compleja de lo que se esperaba por diversos factores. Lo primero de todo es que se trata de un programa desarrollado en Java con Eclipse que, aunque permite generar archivos ejecutables .jar de la aplicación, el objetivo inicial era conseguir un archivo .exe que pudiese ser ejecutado en cualquier ordenador sin la necesidad de que tengan Java instalado.

Además, surgía otro problema, el archivo .jar no abría la aplicación a no ser que se ejecutase mediante un comando para incluir las librerías de JavaFX. Con estos problemas identificados, se comenzó a investigar distintos métodos posibles para conseguir el objetivo final de crear un archivo .exe que contenga la aplicación y no sea necesario tener instalado java ni las librearías de JavaFX para ejecutarlo.

La solución paso a paso fue la siguiente:

1. Transformar el proyecto a Maven[18], con esta herramienta se ha conseguido comprimir el proyecto Java y gestionar las dependencias de las librerías de JavaFX. Para su correcto funcionamiento se tuvo que crear un proyecto desde cero e ir introduciendo clase a clase todo el contenido de la aplicación para que se ajustase a la estructura de Maven. Para gestionar las dependencias se hizo uso del archivo “pom.xml”, en el cual el software de Maven gestiona cómo incluir todas las librerías de JavaFX. Como resultado, construyendo el proyecto con Maven se genera un archivo .jar el cual ya incluye los componentes de JavaFX y es ejecutable teniendo únicamente la versión de Java compatible instalada.

2. El siguiente paso es la creación del .exe, para ello se ha utilizado Launch4j[19], una herramienta que convierte aplicaciones Java en archivos ejecutables.

3. Finalmente, para que el ejecutable funcione sin la necesidad de tener Java instalado, se ha comprimido en un .zip junto a una carpeta que contiene el runtime de Java de la versión correcta. Este archivo comprimido se puede distribuir a cualquier ordenador con sistema operativo Windows y, para usar la aplicación, el usuario solo debe descomprimir el fichero y ejecutar el archivo .exe.

# 4
# Resultados

Con el desarrollo de la aplicación finalizado, se han revisado los objetivos establecidos como requisitos al comienzo del proyecto. Se han valorado si han sido totalmente cumplidos indicando algunas apreciaciones:

- **La aplicación debe ser un simulador de un procesador MIPS**: Cumplido. El resultado final es un programa ejecutable que simula efectivamente el comportamiento del procesador segmentado nanoMIPS.
- **Debe ser capaz de reconocer y ejecutar código escrito en lenguaje ensamblador**: Medianamente cumplido. La aplicación parsea el código del editor de texto, sin embargo, hay que mencionar que no cuenta con la capacidad de detectar errores de sintaxis con anterioridad a la ejecución.
- **Debe ajustarse a las características del nanoMIPS descritas en la asignatura de Arquitectura de Computadores**: Cumplido. Durante todo el transcurso del proyecto se ha hecho hincapié en que se ajuste al procesador nanoMIPS.
- **Debe mostrar la secuencia de instrucciones ejecutadas en un diagrama de ciclos**: Cumplido. Incluso se puede exportar a archivos de imagen.
- **Debe mostrar el estado de los registros**: Cumplido. Se muestran en una tabla.
- **Debe mostrar el estado de la memoria**: Cumplido. Se muestra en una tabla.
- **Debe mostrar estadísticas sobre la ejecución**: Cumplido. En una sección de la aplicación se pueden comprobar distintos datos sobre la ejecución del programa.

- **Debe incluir operaciones y registros de coma flotante**: Cumplido. Se han creado las clases y estructuras necesarias para su correcta implementación.
- **Debe incluir adelantamientos para solventar riesgos de datos**: Cumplido. Son activables a voluntad del usuario.
- **Debe incluir distintas técnicas de resolución de instrucciones de salto**: Cumplido. Cuenta con 4 formas distintas de gestionar los saltos.
- **La aplicación debe ser eficiente en términos de tiempo de ejecución y uso de recursos**: Cumplido. Durante la ejecución de la aplicación no se han detectado ningún tipo de problemas de rendimiento ni errores que causen el cierre repentino de la misma. Además se ha probado la aplicación en varios ordenadores con características distintas, alguno de ellos con mayores limitaciones hardware y tampoco se han observado problemas de ejecución.
- **La interfaz de usuario debe ser intuitiva y fácil de usar para facilitar la interacción con la aplicación**: Cumplido. La interfaz de usuario es clara y concisa, presenta un conjunto de elementos y opciones ordenados y no está sobrecargada.
- **Debe ser compatible con los sistemas operativos y entornos de desarrollo utilizados en la asignatura**: Medianamente cumplido. Si bien la aplicación es ejecutable en sistemas operativos Windows, el cual está presente en los ordenadores de los laboratorios de la Universidad Rey Juan Carlos, no existen las versiones correspondientes para otros sistemas operativos ampliamente utilizados como Linux o macOS, los cuales pueden estar presentes en los ordenadores personales de los estudiantes.
- **Debe garantizar que las simulaciones y las estadísticas de ejecución sean precisas y reflejen correctamente el comportamiento del procesador MIPS**: Medianamente cumplido. Durante todo el desarrollo del simulador se han hecho pruebas para comprobar su correcto funcionamiento. Sin embargo, al día de entrega de esta memoria, se siguen detectando errores que se corregirán en el futuro.

Adicionalmente, se ha realizado un ejercicio de comparación respecto a la principal inspiración del proyecto: WinMIPS64. Al fin y al cabo el objetivo principal que cumple esta aplicación es proporcionar a los alumnos de la asignatura de Arquitectura de Computadores un simulador que se ajuste fielmente al temario impartido y que otorgue nuevas utilidades que WinMIPS64 no dispone.

En la tabla de las siguientes páginas se han reunido las principales características que se han considerado más importantes para su uso en una asignatura como la de Arquitectura de Computadores.

| Características | SimuladorMIPS | WinMIPS64 |
|---|---|---|
| Ejecuta programas de ensamblador MIPS | SÍ | SÍ |
| Repertorio de instrucciones más amplio | NO | SÍ |
| Permite ejecutar el programa ciclo a ciclo | SÍ | SÍ |
| Genera diagrama de ciclos | SÍ | SÍ |
| Permite guardar los diagramas generados como imágenes | SÍ | NO |
| Muestra estadísticas y el estado de los registros y la memoria | SÍ | SÍ |
| Contiene un editor de texto para el código | SÍ | NO |
| Permite introducir puntos de ruptura en el código | NO | SÍ |
| Contiene una terminal | NO | SÍ |
| Permite activar y desactivar los adelantamientos | SÍ | SÍ |

| Características | SimuladorMIPS | WinMIPS64 |
|---|---|---|
| Se indican los adelantamientos de forma precisa en el diagrama de ciclos | SÍ | NO |
| Permite activar y desactivar la segmentación de unidades de coma flotante | SÍ | NO |
| Permite modificar la duración de las operaciones de coma flotante | SÍ | SÍ |
| Permite desactivar las predicciones de salto | SÍ | NO |
| Tiene predicción de salto no tomado | SÍ | SÍ |
| Tiene predicción de salto mediante estados de 1 bit | SÍ | SÍ |
| Tiene predicción de salto mediante estados de 2 bit | SÍ | NO |
| Permite reorganizar los componentes de la interfaz de usuario de la aplicación | NO | SÍ |
| Número de características cumplidas | 14/18 | 12/18 |

Figura 4.1: Tabla Comparativa con WinMIPS64

# 5

# Conclusiones y trabajos futuros

Teniendo en cuenta todos los resultados anteriores, se puede concluir que el principal propósito de este trabajo se ha logrado. La gran mayoría de requisitos establecidos al comienzo del proyecto se han cumplido al completo. Además, observando el ejercicio de comparación, se puede afirmar que el simulador construido cumple más características útiles para el desarrollo de la asignatura que el programa WinMIPS64.

Como conclusión, se ha conseguido desarrollar un simulador de un procesador MIPS del que los estudiantes de computación podrán verse beneficiados, y que puede perfectamente ser usado como sustituto o complemento del programa WinMIPS64 que actualmente se utiliza en las clases de Arquitectura de Computadores.

A lo largo del proyecto, se ha investigado y profundizado en el ámbito de la arquitectura de computadores. Se ha analizado el funcionamiento interno de los procesadores para luego poder recrearlo y plasmarlo en código, tarea que ha resultado ser un desafío. Asimismo, se han obtenido nuevos conocimientos sobre el desarrollo de aplicaciones de escritorio, específicamente, aplicaciones Java.

No obstante, considerando el trabajo realizado completo y el resultado final, la aplicación no está exenta de ciertas limitaciones y se podría beneficiar de la incorporación de nuevas características.

## 5.1. Limitaciones

Como se ha mencionado brevemente en el apartado de resultados, el editor de texto no detecta errores de sintaxis en el código que los usuarios proporcionan. Esto puede afectar a la experiencia de usuario ya que no recibe ninguna indicación del motivo de una ejecución incorrecta debido a un código con algún error escrito. Además el editor de texto no cuenta con ninguna herramienta adicional para facilitar el desarrollo de código como un sistema de colores para cada campo de una instrucción o de la posibilidad de introducir puntos de ruptura.

La interfaz de usuario de la aplicación tampoco es personalizable de ninguna manera, no se puede alterar la distribución de los elementos de la ventana ni modificar sus tamaños ni de los textos. Tampoco existe un tema oscuro que ayude a disminuir la fatiga visual de los usuarios. Estas limitaciones afectan a la usabilidad y accesibilidad de la aplicación, pudiendo perjudicar la experiencia de los usuarios.

## 5.2. Trabajo futuro

Teniendo en cuenta el apartado anterior, encontrar soluciones a dichas limitaciones sería la prioridad principal en caso de continuar el desarrollo del proyecto. Se investigaría la forma de incluir un editor de texto más completo, similar al que contienen los entornos de desarrollo de programación y que indique el número de línea de las instrucciones La aplicación también se vería enormemente beneficiada de la incorporación de una nueva opción en la barra de menús que permita a los usuarios personalizar su apariencia y distribución. En esta sección se podrían incluir opciones como el tema oscuro o permitir cambiar los colores de las casillas del diagrama de ciclos con el objetivo de que los usuarios con daltonismo puedan elegir una paleta de color que puedan diferenciar. Por otro lado, para facilitar la legibilidad del diagrama cuando esté es demasiado alargado, se podría añadir algún tipo de ayuda visual como alternar el color del fondo de cada fila.

Por otro lado, implementar la opción de incluir los algoritmos de “Pizarra” y “Tomasulo” a la ejecución de los programas de ensamblador mejoraría la capacidad de la aplicación de abarcar un mayor número de casos y ejercicios. Estos algoritmos implementan métodos de planificación dinámica de instrucciones, cuyo objetivo es mejorar la eficiencia y el rendimiento de los procesadores al ejecutar instrucciones. Se caracterizan por dividir el camino de datos en otra etapas diferentes a las mencionadas y por modificar el orden de ejecución de las instrucciones para evitar paradas, pero siempre asegurando que el estado final del programa sea correcto, es decir, los resultados de las instrucciones se escriben en el orden correcto establecido. Adicionalmente, se podría implementar la opción de que el procesador tuviese capacidad superescalar, característica que permite la emisión de más de una instrucción en el mismo ciclo. Estas técnicas forman parte del temario de la asignatu-

ra de Arquitectura de Computadores, estando implementados en la aplicación los alumnos dispondrían de una herramienta útil para asimilarlos.

Finalmente, el proyecto se beneficiaría considerablemente de llevar a cabo una prueba práctica con alumnos y profesores de la asignatura. Tras su realización, se les facilitaría una encuesta para conocer sus opiniones, recibir retroalimentación y analizar los resultados por si fuese necesario realizar nuevos cambios en la aplicación. Dicha encuesta estaría compuesta de varias secciones, una para valorar el funcionamiento a nivel técnico de la aplicación y si resulta útil como herramienta de estudio. Otra sección evaluaría la usabilidad de la aplicación, para conocer si resulta sencillo utilizarla o si surge algún tipo de confusión de cualquier aspecto. Finalmente, una última sección para valorar la accesibilidad y comprobar si está disponible para todo tipo de usuarios.

# Bibliografía


[1] P. Quiroga, *Arquitectura de computadoras*. Alpha Editorial, 2010.

[2] "Mips (procesador)," Último acceso 28 Junio 2024. [Online]. Available: https://mips.com/

[3] M. B. Pardo and A. G. Sacristán, *Diseño y evaluación de arquitecturas de computadoras*. Pearson Educación, 2010.

[4] M. Scott, "Winmips64," 2012, Último acceso 21 Junio 2024. [Online]. Available: http://indigo.ie/~mscott/

[5] Pete Sanderson and Kenneth Vollmar, "Mars mips simulator," 2006, Último acceso 21 Junio 2024. [Online]. Available: https://courses.missouristate.edu/kenvollmar/mars/index.htm

[6] Universidad de A Coruña, "Simula3ms," Último acceso 5 Julio 2024. [Online]. Available: https://simula3ms.des.udc.es/

[7] University of Canterbury, "Csfg mips simulator," Último acceso 21 Junio 2024. [Online]. Available: https://www.csfieldguide.org.nz/en/interactives/mips-simulator/

[8] M. Amor, R. Concheiro, P. González, M. Bóo, J. A. Lorenzo, D. Piso, R. R. Osorio, "Aprender arquitectura de computadores con la herramienta simula3ms," Último acceso 2 Julio 2024. [Online]. Available: https://upcommons.upc.edu/bitstream/handle/2099/11816/r46.pdf?sequence=1#:~:text=

[9] J. K. Yousafzai and A. K. Yousafzai, "A brick-by-brick approach to learning mips microarchitecture," in *2023 IEEE Global Engineering Education Conference (EDUCON)*, 2023, pp. 1–5.

[10] A. Gersnoviez, M. Brox, M. A. Montijano, J. A. Sújar, and C. D. Moreno, "Ucomipsim 2.0: Pipelined mips architecture simulator," in *2018 XIII Technologies Applied to Electronics Teaching Conference (TAEE)*, 2018, pp. 1–6.

[11] D. X. Lim and K. G. Smitha, "Pipelined mips simulation: A plug-in to mars simulator for supporting pipeline simulation and branch prediction," in *2019 IEEE International Conference on Engineering, Technology and Education (TALE)*, 2019, pp. 1–7.

[12] Oracle, "Java," Último acceso 1 Julio 2024. [Online]. Available: https://www.java.com/en/download/help/whatis_java.html

[13] Eclipse, "Eclipse ide," Último acceso 20 Junio 2024. [Online]. Available: https://eclipseide.org/

[14] OpenJFX, "Javafx," Último acceso 22 Junio 2024. [Online]. Available: https://openjfx.io/

[15] Gluon, "Scene builder," Último acceso 22 Junio 2024. [Online]. Available: https://gluonhq.com/products/scene-builder/

[16] R. L. Cabeza, "Repositorio github," Último acceso 1 Julio 2024. [Online]. Available: https://github.com/Rllona/TFGComputadores

[17] GitHub, Inc., "GitHub Desktop," 2015, Último acceso 24 Junio 2024. [Online]. Available: https://desktop.github.com/

[18] Apache Software Foundation, “Maven,” Último acceso 24 Junio 2024. [Online]. Available: https://maven.apache.org/

[19] G. Kowal, “Launch4j,” Último acceso 21 Junio 2024. [Online]. Available: https://launch4j.sourceforge.net/

# Apéndices

# A

# Ventana de Información y Funcionamiento

## A.1. Manual de la aplicación

### A.1.1. Botones:

Figura A.1: Botón de ejecución del código entero

Figura A.2: Botón de ejecución del código ciclo a ciclo

Figura A.3: Botón de cancelación de ejecución y limpieza del entorno

Figura A.4: Botón de guardado de imagen del diagrama de ciclos generado

### A.1.2. Formato del código:

La sección de código debe seguir un formato correcto de lenguaje ensamblador para el correcto funcionamiento de la simulación. El código debe estar dividido en 2 secciones (.data y .text/.code).

#### Sección .data:

Esta sección es opcional, sirve para introducir datos y variables en la memoria. La memoria tiene una capacidad de 1024 palabras de 32 bits, siendo la dirección de acceso a cada palabra múltiplo de 4. Se aceptan las siguientes directivas:

- **.word/.word32**: Introduce números de 32 bit.
- **.float**: Introduce números de coma flotante de 32 bit.
- **.word16**: Introduce números de 16 bit.
- **.byte**: Introduce números de 8 bit.
- **.double**: Introduce números de 64 bit.

Tras la declaración de una directiva se pueden introducir los números que se deseen, separados por comas y en la misma línea. Se aceptan tanto números decimales como hexadecimales precedidos por 0x. Para introducir variables de memoria, se deben declarar seguidas de dos puntos, la directiva deseada y su valor (nombreVariable: .word XXX). Las variables de memoria guardan la dirección de memoria del valor introducido.

#### Sección .text/.code:

Esta sección es obligatoria, contiene la secuencia de instrucciones a ejecutar en lenguaje ensamblador. El procesador simulado contiene 32 registros enteros accesibles de la forma r0-r31 o R0-R31, y 32 registros de coma flotante accesibles de la forma f0-f31 o F0-F31. La sección no debe contener errores sintácticos, de lo contrario la ejecución no será completada o tendrá fallos.

#### Código de ejemplo:

```
.data
A: .word 10, 0x0a2df36b
B: .float 2.25
.byte 0xf1, 0xf2

.text
       lw R1, A(R0)
       l.d F1, B(R0)
       li R2, 15
loop:  addi R1, R1, 1
       bne R1, R2, loop
       mul.d F2, F1, F1
       s.d F2, B(R0)
```

## A.2. Características del procesador MIPS simulado

El procesador MIPS simulado se trata de un procesador multiciclo segmentado, el cual divide cada instrucción en las etapas de Fetch (IF), Decode (ID), Execution (EX), Memory (MEM) y Write Back (WB).

### A.2.1. Detección de riesgos

Los riesgos de datos y estructurales de acceso a las unidades de operaciones de coma flotante son detectados en la etapa Decode (ID).

Los riesgos estructurales de acceso a la memoria son detectados en la etapa Execution (EX). En el caso de las unidades de operaciones de coma flotante, son detectados en el último ciclo de la operación.

### A.2.2. Gestión de saltos

Con la predicción de saltos desactivada, las instrucciones de salto son resueltas en la etapa Decode (ID), introduciendo un ciclo de parada.

Con predicción de salto no tomado, las instrucciones de salto son resueltas en la etapa Decode (ID), comenzando la ejecución de la instrucción inmediatamente posterior. En caso de que el salto finalmente se tome, se cancela la ejecución de la instrucción introducida.

Con predicciones de salto bajo estados de 1 o 2 bits, las instrucciones de salto son resueltas en la etapa Decode (ID), comenzando la ejecución de la instrucción que el predictor marque. En el caso de que el predictor falle, se cancela la ejecución de la instrucción introducida. Por defecto, el predictor de salto de 1 bit viene inicializado como 0 y el de 2 bits como 01.

### A.2.3. Segmentación de unidades de operaciones de coma flotante

Al habilitar la opción de segmentación, el sumador/restador de coma flotante, el multiplicador y el divisor pasan a estar segmentados internamente. De tal manera que pueden ejecutarse varias operaciones del mismo tipo simultáneamente.

## A.3. Listado de instrucciones soportadas

#### Instrucciones de tipo Aritmético-Lógicas:

- **add**: Suma de registros de enteros.
- **sub**: Resta de registros de enteros.
- **addi**: Suma de un registro entero y un inmediato.
- **li**: Carga de un inmediato a un registro entero (pseudo-instrucción que se transforma a addi del registro R0 y el inmediato).
- **and**: Operación AND lógica de registros enteros.
- **or**: Operación OR lógica de registros enteros.
- **xor**: Operación XOR lógica de registros enteros.
- **nor**: Operación NOR lógica de registros enteros.
- **add.d**: Suma de registros de coma flotante.
- **sub.d**: Resta de registros de coma flotante.
- **mul.d**: Multiplicación de registros de coma flotante.
- **div.d**: División de registros de coma flotante.

#### Instrucciones de tipo Memoria:

- **lw**: Carga de memoria de una palabra (32 bits) en un registro entero.
- **sw**: Guardado de un registro entero en memoria como una palabra (32 bits).
- **lb**: Carga de memoria de un byte (8 bits) en un registro entero.
- **sb**: Guardado de un registro entero en memoria como un byte (8 bits).
- **lh**: Carga de memoria de media palabra (16 bits) en un registro entero.
- **sh**: Guardado de un registro entero en memoria como media palabra (16 bits).
- **l.d**: Carga de memoria de una palabra (32 bits) en un registro de coma flotante.
- **s.d**: Guardado de un registro de coma flotante en memoria como una palabra (32 bits).

**Instrucciones de tipo Salto:**

- **j**: Salto incondicional a una etiqueta.
- **beq**: Salto a una etiqueta bajo condición si los valores de los registros enteros son iguales.
- **bne**: Salto a una etiqueta bajo condición si los valores de los registros enteros no son iguales.

## A.4. Advertencia

Esta aplicación ha sido desarrollada para un trabajo de fin de grado de la Universidad Rey Juan Carlos. No es un proyecto realizado por un equipo profesional, la aplicación puede contener errores o calcular equivocadamente ciertas combinaciones de instrucciones.

*Creado por Raúl Llona Cabeza - Estudiante del Grado en Ingeniería de Computadores, ETSII, URJC 2024.*