# UNIVERSIDAD DE CASTILLA-LA MANCHA

## ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

## GRADO EN INGENIERÍA INFORMÁTICA: TECNOLOGÍA ESPECÍFICA DE INGENIERÍA DE COMPUTADORES

## TRABAJO FIN DE GRADO

Implementación de recogida y visualización de estadísticas de rendimiento en el Simula3MS

Pablo Abril Torres

**Julio, 2013**


**UNIVERSIDAD DE CASTILLA-LA MANCHA**

**ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA**

**DEPARTAMENTO DE SISTEMAS INFORMÁTICOS**


**TRABAJO FIN DE GRADO**

# Implementación de recogida y visualización de estadísticas de rendimiento en el Simula3MS


Autor: Pablo Abril Torres

Directores: Francisco José Alfaro Cortes

Pedro Javier García García


**Julio, 2013**

# RESUMEN

Simula3MS es un simulador de una arquitectura básica del procesador MIPS en sus versiones monociclo, multiciclo y segmentado. Es un simulador orientado a la enseñanza de la arquitectura del procesador MIPS en las asignaturas que tratan la arquitectura y organización del computador, en el cual se pueden observar y aplicar los conocimientos adquiridos en las clases de teoría. Cuenta con un entorno de trabajo gráfico y de fácil manejo que permite depurar cómodamente los programas, observar la evolución de la memoria, así como la ejecución de las instrucciones sobre distintos caminos de datos. La presencia de las distintas implementaciones en la misma herramienta permite observar las diferencias de ejecución de un mismo código según cuales sean las características del procesador.

En esta memoria se describen los cambios realizados al simulador para la implementación de recogida y visualización de estadísticas de rendimiento, ya que la versión que se utilizaba hasta ahora no ofrecía esta serie de datos que son muy interesantes para la docencia. Este documento comienza haciendo un recorrido por el estado del arte de la arquitectura MIPS, viendo su repertorio de instrucciones y diferentes tipos de procesadores ya sea monociclo, multiciclo o segmentado. A continuación se explican los objetivos de este trabajo y la metodología de desarrollo que se va a emplear. Después se ve en detalle el proceso de desarrollo y su implementación. Finalmente se incluyen algunos ejemplos de uso del simulador y las conclusiones y líneas de trabajo futuro que han quedado abiertas.

# AGRADECIMIENTOS

Este trabajo ha sido realizado gracias al apoyo de las personas que han confiado en mí y el esfuerzo por lograr los objetivos que nos planteamos.

Agradecer a mis tutores Francisco José Alfaro y Pedro Javier García por la comprensión y confianza que depositaron en mí para que pudiera finalizar el trabajo.

También agradecer a mi familia, mis amigos y especialmente mi novia por no perder la fe en mí.

# ÍNDICE GENERAL

# ÍNDICE DE FIGURAS

# ÍNDICE DE TABLAS

# CAPÍTULO 1. INTRODUCCIÓN

Este capítulo muestra las características básicas del proyecto, es decir, el contexto de partida sobre el que se ha construido y que ha motivado su desarrollo, los objetivos o las metas que se han propuesto alcanzar. Finalmente, se incluye una descripción de la organización de la memoria.

## 1.1 CONTEXTO

Los simuladores de procesadores son ampliamente utilizados en docencia [13] [15] debido a que ofrecen un entorno más sencillo de manipular que una máquina real, además de ofrecer otras ventajas: pueden detectar errores, ofrecen más posibilidades que un ordenador real y no modifican elementos físicos del computador.

Simula3MS [1] (ver figura 1.1) es un simulador resultado de un proyecto del grupo de Arquitectura de Computadores de la Universidad de A Coruña. El proyecto abarca la implementación de un simulador de una arquitectura de procesador básica en sus versiones monociclo, multiciclo y segmentado. Actualmente 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.

Simula3MS está orientado al estudio de las asignaturas que tratan la arquitectura y organización del computador, en el cual se pueden observar y aplicar los conocimientos adquiridos en las clases de teoría. Cuenta con un entorno de trabajo gráfico y de fácil manejo que permite depurar cómodamente los programas, observar la evolución de la memoria, así como la ejecución de las instrucciones sobre distintos caminos de datos. La presencia de las distintas implementaciones en la misma herramienta permite observar las

diferencias de ejecución de un mismo código según cuales sean las características del procesador.

Simula3MS implementa un subconjunto de instrucciones basadas en el repertorio de instrucciones del procesador MIPS R2000/R3000. Consta de una primera parte que incluye un editor y que permite analizar sintácticamente las instrucciones antes de pasar a observar la ejecución de las mismas. Una vez analizadas sintácticamente las instrucciones se puede seguir la evolución del segmento de datos, así como de los registros y del resto de elementos de la ventana de ejecución. Por otra parte, Simula3MS cuenta con un coprocesador de punto flotante en todas sus configuraciones, e incluye dos técnicas de gestión de entrada/salida: encuesta e interrupciones.

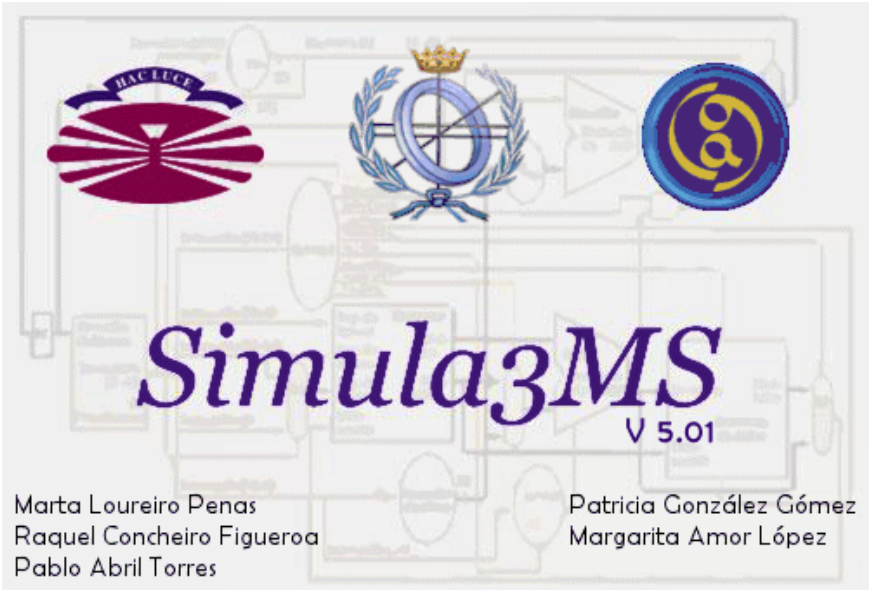



Figura 1.1: Logo de la nueva versión del simulador MIPS Simula3MS.

## 1.2 MOTIVACIÓN

Aunque Simula3MS está operativo, y de hecho se emplea en la docencia de asignaturas relacionadas con la arquitectura de computadores, las prestaciones que ofrece son mejorables.

Concretamente, al analizar las funcionalidades ofrecidas en su actual estado por el simulador Simula3MS, se detectó que no dispone de la recogida y visualización de estadísticas de rendimiento, las cuales serían de vital importancia para un correcto seguimiento de la ejecución de un programa y su posterior análisis. Contar con estas estadísticas seria de gran importancia para la docencia

## 1.3 OBJETIVOS Y TAREAS

El principal objetivo de este proyecto es, disponiendo del código fuente de la última versión del simulador, ampliarlo para desarrollar nuevas funcionalidades orientadas a la reogida y visualización de estadísticas de rendimiento.

Para conseguir este objetivo, se plantean las siguientes tareas:

1. Estudio del diseño de la ruta de datos de un procesador y su segmentación. Es necesario empezar comprendiendo la teoría relacionada con el simulador y así entender y comprender su funcionamiento y conocer sus limitaciones.
2. Familiarizarse con la aplicación Simula3MS en su última versión, ya que es imprescindible conocer a fondo todas las características que ofrece esta herramienta para hacer un desarrollo efectivo y eficiente del proyecto de modelado
3. Estudio exhaustivo del código fuente de la herramienta Simula3MS, necesario para entender cómo funcionan las clases y los métodos que componen la herramienta.
4. 4. Mejora en el código existente y creación de nuevo código que cumpla con el objetivo final del trabajo: Implementación de recogida y visualización de estadísticas de rendimiento. Esta tarea es la que supone mayor esfuerzo dada la complejidad del código.

## 1.4 ESTRUCTURA DE LA MEMORIA

Para proporcionar al lector una visión global de la estructura del documento, a continuación se expone un resumen de los capítulos del mismo:

**Capitulo 1: Introducción.** El lector puede hacerse una idea general de qué es lo que va a encontrar en el documento. Se hace un recorrido por el contexto en el que estamos, se describen las motivaciones que guían al proyecto, y los objetivos principales que se persiguen durante su desarrollo.

**Capítulo 2: Implementación básica MIPS.** Se presentan los aspectos básicos de la arquitectura MIPS resumidos.

**Capítulo 3: Planteamiento del proceso de desarrollo.** Este capítulo presenta el estado del simulador MIPS Simula3MS, los cambios a introducir en el mismo para mejorarlo y se comenta la metodología que se ha utilizado para ello.

**Capítulo 4: Diseño e implementación de recogida de estadísticas.** Este capítulo describe las modificaciones que se han realizado en el simulador original para la inclusión de las nuevas funcionalidades.

**Capítulo 5: Ejemplo de uso.** En este capítulo se muestran los resultados obtenidos por el nuevo simulador desarrollado ejemplos de las nuevas funcionalidades.

**Capítulo 6: Conclusiones y trabajo futuro.** En el ultimo capitulo del proyecto se hace una descripción de las conclusiones del proyecto y se plantean objetivos de cara a un desarrollo futuro.

**Bibliografía.** En este apartado se recoge la fuente a todas las referencias que se hacen durante el desarrollo de este trabajo.

**Anexos.** Esta última parte está compuesta por tres anexos: El primer anexo incluye todo el código fuente creado y modificado para este trabajo. El segundo anexo comenta el contenido que se incluye en el CD adjunto. Por ultimo esta el manual de usuario que explica cómo hacer funcionar el simulador Simula3MS.

# CAPÍTULO 2. IMPLEMENTACIÓN BÁSICA MIPS

El diseño del camino de datos de un procesador está condicionado por la elección del repertorio de instrucciones que éste va a ejecutar. El simulador Simula3MS realiza una implementación de un camino de datos de un subconjunto del repertorio de instrucciones del MIPS [12] [8] [7] como se puede ver el el Anexo A.

La elección de este subconjunto condiciona la circuitería necesaria para el diseño del camino de datos, aunque ésta se verá también supeditada a otro tipo de factores como puede ser la estrategia arquitectónica, así como la realización lógica y su sincronización.

Muchos de los conceptos utilizados para implementar este subconjunto de instrucciones MIPS son los mismos que se utilizan para construir una amplia variedad de computadores, desde arquitecturas de altas prestaciones hasta microprocesadores de propósito general o de propósito específico.

## 2.1 ARQUITECTURA DEL MIPS

La arquitectura MIPS (ver figura 2.1) posee 32 registros genéricos ($0-$31) de 32 bits para uso del procesador, siendo el registro $0 de sólo lectura y con valor cero. De los restantes registros, sólo el $31 es implícitamente usado por una instrucción (jal de invocación de una subrutina, para guardar la dirección de retorno) [5].

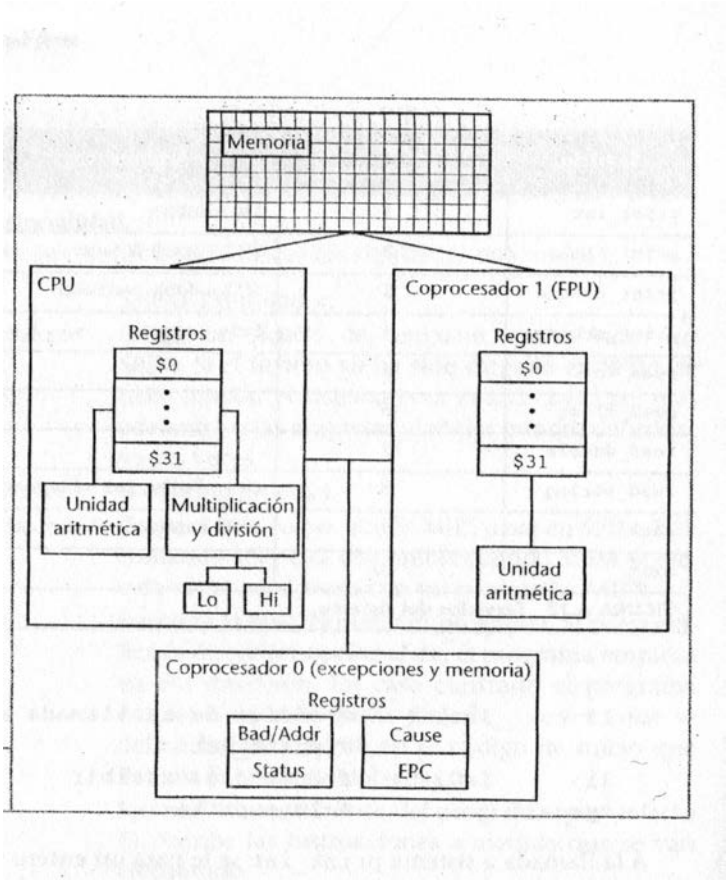



Figura 2.1: Arquitectura MIPS.

Adicionalmente, el MIPS contiene dos registros para poder operar con operandos de 64 bits, como sucede en el caso de la multiplicación y la división, llamados hi (high) y lo (low).

La arquitectura MIPS tiene en cada coprocesador 32 registros (FP0-FP31) de 32 bits para punto flotante, que pueden ser organizados en 16 registros de doble precisión, con 64 bits (en este caso se utilizan las designaciones par de los registros).

## 2.2 REPERTORIO DE INSTRUCCIONES DEL MIPS

Los tipos básicos de instrucciones que soporta el MIPS son: aritméticas y lógicas, de transferencia de datos, de salto condicional y de bifurcación.Las características más importantes de estas instrucciones en un repertorio MIPS son:

- La longitud de todas las instrucciones MIPS es fija y de 32 bits.
- Los operandos de las operaciones aritméticas son siempre registros.

- El acceso a memoria se hace a través de las operaciones de carga y almacenamiento (transferencia de datos).
- Para acceder a una palabra en memoria hay que indicar su dirección. MIPS direcciona bytes individuales.
- Al guardar todas las instrucciones con la misma longitud se requieren diferentes clases de formatos de instrucción para diferentes tipos de instrucciones.

Los tres tipos de formatos de instrucciones en MIPS son:

- Formato tipo R: utilizado por las instrucciones aritméticas y lógicas.
- Formato tipo I: utilizado por las instrucciones de transferencia, las de salto condicional y las instrucciones con operandos inmediatos.
- Formato tipo J: utilizado por las instrucciones de bifurcación

Como se muestra en la tabla 2.1, a los campos MIPS se les da una serie de nombres para identificarlos fácilmente:

**Tipo R**

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

**Tipo I**

| op | rs | rt | dirección |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

**Tipo J**

| op | dirección |
|---|---|
| 6 bits | 26 bits |

Tabla 2.1: Codificación de instrucciones en el MIPS.

- op: operación básica de la instrucción, tradicionalmente llamado código de la operación.
- rs: primer registro, operando fuente.
- rt: segundo registro, operando fuente.
- rd: registro operando destino, donde se almacena el resultado de la operación.
- shamt: tamaño del desplazamiento.
- funct: función. Este campo selecciona la variante específica de la operación del campo op, y a veces se le denomina código de función.

Aunque tener múltiples formatos complica la circuitería, se puede reducir la complejidad guardándolos de forma similar. Por ejemplo, los tres primeros campos de los formatos tipo R y tipo I son del mismo tamaño y tienen los mismos nombres. Los formatos se distinguen por el valor del primer campo: a cada formato se le asigna un conjunto de valores distintos en el primer campo y por lo tanto la circuitería sabe cómo ha de tratar la última mitad de la instrucción, bien como tres campos (tipo R) o como un campo simple (tipo I), o si la instrucción es tipo J.

## 2.3 INTRODUCCIÓN AL CAMINO DE DATOS

El procesador [12] es el elemento del computador que se encarga de ejecutar las instrucciones especificadas por el programa y coordina las actividades de las otras unidades del computador. Existen diferentes estrategias en el desarrollo de procesadores.

Un procesador monociclo se caracteriza porque cada instrucción se ejecuta en un único ciclo de reloj (CPI=1). En otra alternativa conocida como procesador multiciclo las instrucciones diferentes duran diferentes ciclos de reloj (CPI>1). Y en los procesadores segmentados se permite solapar la ejecución de instrucciones. Hoy en día los procesadores monociclo y multiciclo tienen sólo valor pedagógico pues en la realidad han sido reemplazados por procesadores segmentados por su mayor rapidez y rendimiento.

La segmentación incrementa el rendimiento incrementando la productividad, en lugar de reducir la ejecución de cada instrucción individual. La productividad de las instrucciones es una medida importante ya que los programas reales ejecutan miles de millones de instrucciones.

## 2.4 PROCESADORES MONOCICLO

Un camino de datos monociclo se caracteriza porque la ejecución de cada instrucción tiene una duración de un ciclo de reloj. Como el ciclo de reloj debe estar adaptado a las necesidades de la ejecución de todas las instrucciones y necesita ajustar su duración a la de aquella instrucción que sea más larga. Pero el tiempo necesario para la ejecución puede variar sensiblemente de una instrucción a otra. Así una jump o salto incondicional necesita mucho menos tiempo que, por ejemplo, una carga.

El concentrar la totalidad de la ejecución en un ciclo de reloj influye de forma manifiesta en la circuitería necesaria ya que obliga a tener duplicado hardware porque hay casos en los cuales es indispensable utilizarlo en más de una ocasión en cada ciclo. En la figura 2.2 puede verse el camino de datos para el subconjunto del repertorio MIPS que utiliza el simulador Simula3MS.

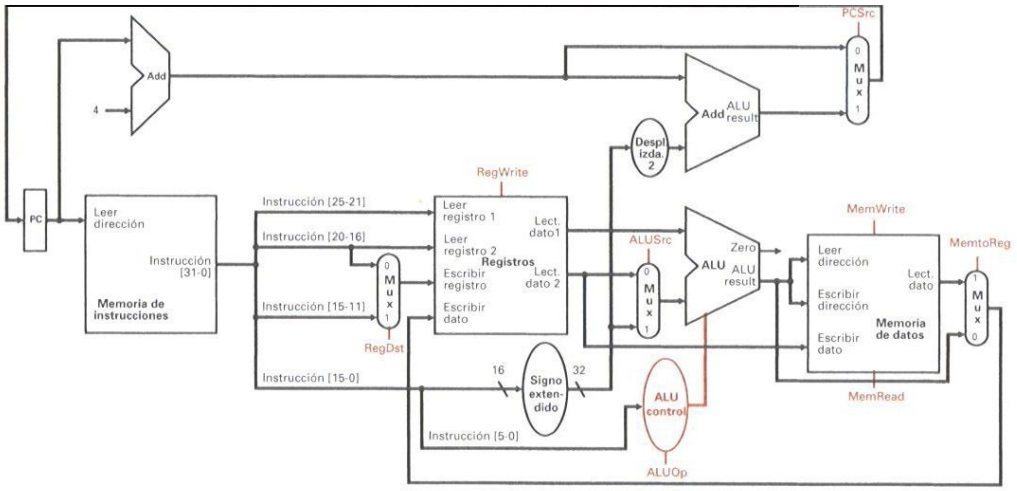



Figura 2.2: Camino de datos monociclo.

Este camino de datos se caracteriza, entre otras cosas, por tener una memoria de datos y otra de instrucciones, necesarias ambas porque al ejecutarse cada instrucción en un ciclo de reloj puede ser necesario acceder a ambas durante ese ciclo. Así en la memoria de instrucciones se accede a aquella instrucción almacenada en la posición asociada al valor del contador de programa (PC), y en la memoria de datos se pueden obtener los datos almacenados en los registros relacionados con la instrucción a ejecutar.

También es necesario el uso de tres Unidades Aritmético-Lógicas (ALUs) ya que dos de ellas son utilizadas para calcular el siguiente valor del PC y la otra se hace cargo de realizar las operaciones necesarias para las instrucciones aritmético-lógicas.

Uno de los principales componentes de un camino de datos es la unidad de control, que a partir de los seis bits correspondientes al código de operación modifica los valores de las distintas señales de control del camino de datos. Hay algunos elementos como el banco de registros que necesitan una señal de escritura, o como la memoria de datos que requiere una señal de escritura y otra de lectura. Cada multiplexor necesita una línea de control y además, hay una unidad de control de la ALU que dependiendo de los valores de las líneas ALUOp controla el funcionamiento de la misma. En las tablas 2.2 y 2.3 se pueden ver las distintas señales de control y su efecto

Estas señales de control se activan en función de las seis líneas de entrada a la unidad de control. Estas líneas se corresponden con los 6 bits más significativos de la codificación de cada instrucción.

| Señal de control | Efecto cuando no está activa | Efecto cuando está activa |
|---|---|---|
| *RegDest* | El identificador del registro destino viene determinado por el campo rt. | El identificador del registro destino viene determinado por el campo rd. |
| *EscrReg* | Ninguno | El registro destino se actualiza con el valor a escribir |
| *FuenteALU* | El segundo operando de la ALU proviene del segundo registro leído del banco de registros. | El segundo operando de la ALU son los 16 bits de menor peso de la instrucción con el signo extendido. |
| *LeerMem* | Ninguno | El valor de la posición de memoria designada por la dirección calculada por la ALU se coloca en la salida de lectura. |
| *EscrMem* | Ninguno | El valor de la dirección de memoria calculada por la ALU se reemplaza por el valor de la entrada de datos. |
| *MemaReg* | El valor de entrada del banco de registros proviene de la ALU | El valor de entrada del banco de registros proviene de la memoria de datos. |
| *SaltoCond* | El PC es reemplazado por su valor anterior más cuatro | Si la línea Cero de la ALU está activa, el PC se reemplaza por la dirección de salto. |
| *EscrDato* | El dato a escribir se corresponde con la salida de la ALU o el dato leído de memoria, dependiendo del valor de *MemaReg* | Se escribe en el registro el valor del PC actual más cuatro. |

Tabla 2.2: Señales de control de una línea para una realización monociclo.

| Señal de control | Valor | Efecto |
|---|---|---|
| *ALUOp* | 00 | La ALU realiza una operación de suma. |
| | 01 | La ALU realiza una operación de resta |
| | 10 | El campo de función de la instrucción determina la operación a realizar por la ALU. |
| *FuentePC* | 00 | El valor que se escribirá en el PC dependerá de *SaltoCond*. Si ésta está activada se modifica el valor del PC al salto, si no se escribe con el PC siguiente (PC+4). |
| | 10 | El PC se actualiza con el valor del registro indicado por el campo rs. |
| | 11 | La dirección destino de un *jump* desplazada 2 bits a la izquierda y concatenada con los bits de mayor peso de PC+4 se envía para su escritura en el PC. |

Tabla 2.3: Señales de control de más de una línea para una realización monociclo.

## 2.5 PROCESADORES MULTICICLO

Como se ha visto, aunque la aproximación monociclo es fácil de entender, no es práctica ya que todas las instrucciones tardan el mismo tiempo (un ciclo) en completarse; el ciclo de reloj tendrá que adaptarse a la más lenta. Sería mejor una realización que permitiese a las distintas instrucciones tardar un número diferente de ciclos, el tamaño del ciclo mucho más corto.

Así, las instrucciones se pueden dividir en distintos pasos según las unidades funcionales que vayan a utilizar durante la ejecución de las distintas etapas. Éstas permiten compartir la circuitería dejando que una unidad funcional pueda utilizarse en más de una ocasión durante la ejecución de la misma instrucción, siempre y cuando lo haga durante ciclos distintos.

Este punto se observa claramente al comparar el camino de datos monociclo (figura 2.2) con el multiciclo (figura 2.3) ya que este ultimo solamente tiene una ALU, frente a la ALU y los dos sumadores que eran necesarios para el monociclo. Este reemplazo ocasiona que todas las líneas de control que antes estaban repartidas entre las tres ALUs ahora se direccionen a una única ALU. Como consecuencia, por tener más líneas de entrada se necesitan más multiplexores y más grandes. Otro ejemplo importante es la presencia de una única unidad de memoria para datos y para instrucciones, en vez de las dos memorias por separado que se utilizaban en el camino de datos monociclo.

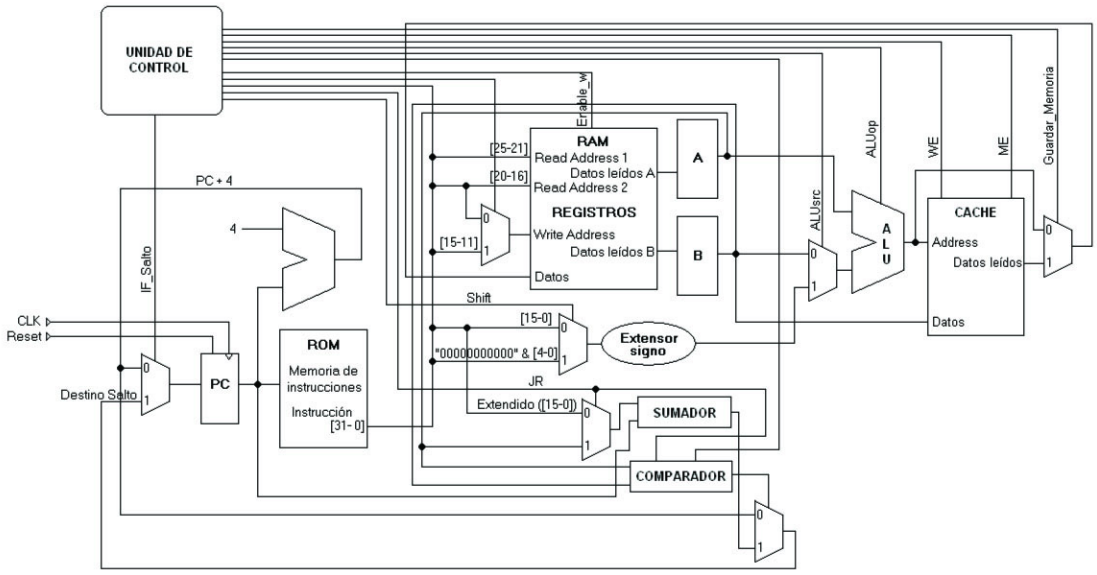



Figura 2.3: Camino de datos multiciclo.

Por contra son necesarios varios registros adicionales (A, B, IR, MDR, SalidaALU) tras cada unidad funcional para mantener los resultados obtenidos entre las distintas etapas de las instrucciones.

Como el camino de datos multiciclo tarda más de un ciclo en ejecutar una instrucción, se va a necesitar un conjunto de señales de control diferente al utilizado en la realización monociclo. Los elementos de estado visible por el programador (el PC, la memoria y los registros), así como el registro IR necesitan señales de control de escritura. Cada uno de los multiplexores de dos vías requiere una única señal de control, mientras que los de cuatro necesitan dos líneas. La ALU sigue teniendo una unidad de control asociada cuyo funcionamiento depende del valor de las líneas ALUOp. En el camino de datos multiciclo hay catorce líneas de control cuyo nombre y efecto viene explicado en los cuadros 2.4 y 2.5.

| **Señal de control** | **Efecto cuando no está activa** | **Efecto cuando está activa** |
|---|---|---|
| *RegDest* | El identificador del registro destino viene determinado por el campo rt. | El identificador del registro destino viene determinado por el campo rd. |
| *EscrReg* | Ninguno. | El valor del registro destino se actualiza con el dato a escribir. |
| *SelALU* | El primer operando de la ALU es el valor del PC. | El primer operando de la ALU está en el registro A. |
| *LeerMem* | Ninguno. | El valor de la posición indicada por la dirección se coloca en la salida de lectura. |
| *EscReg* | Ninguno. | El registro del escritura se actualiza con el dato a escribir. |
| *MemaReg* | El valor que se pone a la entrada del banco de registros proviene del registro SalidaALU. | El valor que se pone a la entrada del banco de registros viene del registro MDR. |
| *IoD* | El PC suministra la dirección para acceder a memoria. | SalidaALU suministra la dirección para acceder a memoria. |
| *EscrIR* | Ninguno. | La salida de memoria se escribe en el registro IR. |
| *EscrPC* | Ninguno. | Se escribe el PC. Su origen estará controlada por *FuentePC*. |
| *EscrPCCond1* | Ninguno | Se escribe en el PC si la señal de Cero de la ALU está también activa. |
| *EscrPCCond2* | Ninguno | Se escribe en el PC si la señal de Cero de la ALU no está activa. |

Tabla 2.4: Señales de control de una línea para una realización multiciclo.

| Señal de control | Valor | Efecto |
|---|---|---|
| *ALUOp* | 00 | La ALU realiza una operación de suma. |
| | 01 | La ALU realiza una operación de resta |
| | 10 | El campo de función de la instrucción determina la operación a realizar por la ALU. |
| *SelALUB* | 00 | El segundo operando de la ALU es el registro B. |
| | 01 | El segundo operando de la ALU es la constante 4. |
| | 10 | El segundo operando de la ALU son los 16 bits de menor peso del IR con el signo extendido. |
| | 11 | El segundo operando de la ALU son los 16 bits de menor peso de IR con el signo extendido y desplazado 2 bits a la izquierda. |
| *FuentePC* | 00 | El resultado de la ALU (PC+4) se envía para su escritura en el PC. |
| | 01 | El contenido de SalidaALU (destino del salto) se envía para su escritura en el PC. |
| | 10 | El contenido del registro A se envía para su escritura en el PC. |
| | 11 | La dirección destino de un *jump* desplazada 2 bits a la izquierda y concatenada con los bits de mayor peso de PC+4 se envía para su escritura en el PC. |

Tabla 2.5: Señales de control de más de una línea para una realización multiciclo.

## 2.6 PROCESADORES SEGMENTADOS

La segmentación [12] [14] [8] [7] es una técnica que permite solapar en el tiempo la ejecución de instrucciones. Actualmente la segmentación es la clave para hacer que los procesadores sean rápidos. En condiciones ideales, el incremento de la velocidad con la segmentación iguala al número de etapas que se pueden solapar.

La segmentación incrementa el rendimiento aumentando la productividad, en lugar de reducir el tiempo de la ejecución de cada instrucción individual. Se entiende por productividad la cantidad total de trabajo realizada en un determinado tiempo. La productividad de las instrucciones es una medida importante ya que los programas reales ejecutan miles de millones de instrucciones.

La segmentación es una técnica que explota el paralelismo existente entre instrucciones de un flujo secuencial, añadiendo la ventaja de ser totalmente invisible al programador. Las propiedades del repertorio de instrucciones MIPS favorecen la división de cada instrucción en cinco etapas, lo que implica una segmentación en cinco etapas.

Esto significa que durante un ciclo de reloj se podrían estar ejecutando simultáneamente hasta cinco instrucciones consecutivas. Estas etapas son:

1. IF: Búsqueda de instrucciones.
2. ID: Decodificación de instrucciones, lectura del banco de registros y realización de los saltos.
3. EX: Ejecución o cálculo de direcciones.
4. MEM: Acceso a memoria de datos.
5. WB: Escritura retardada.

En la figura 2.4 se muestra el esquema del camino de datos segmentado.

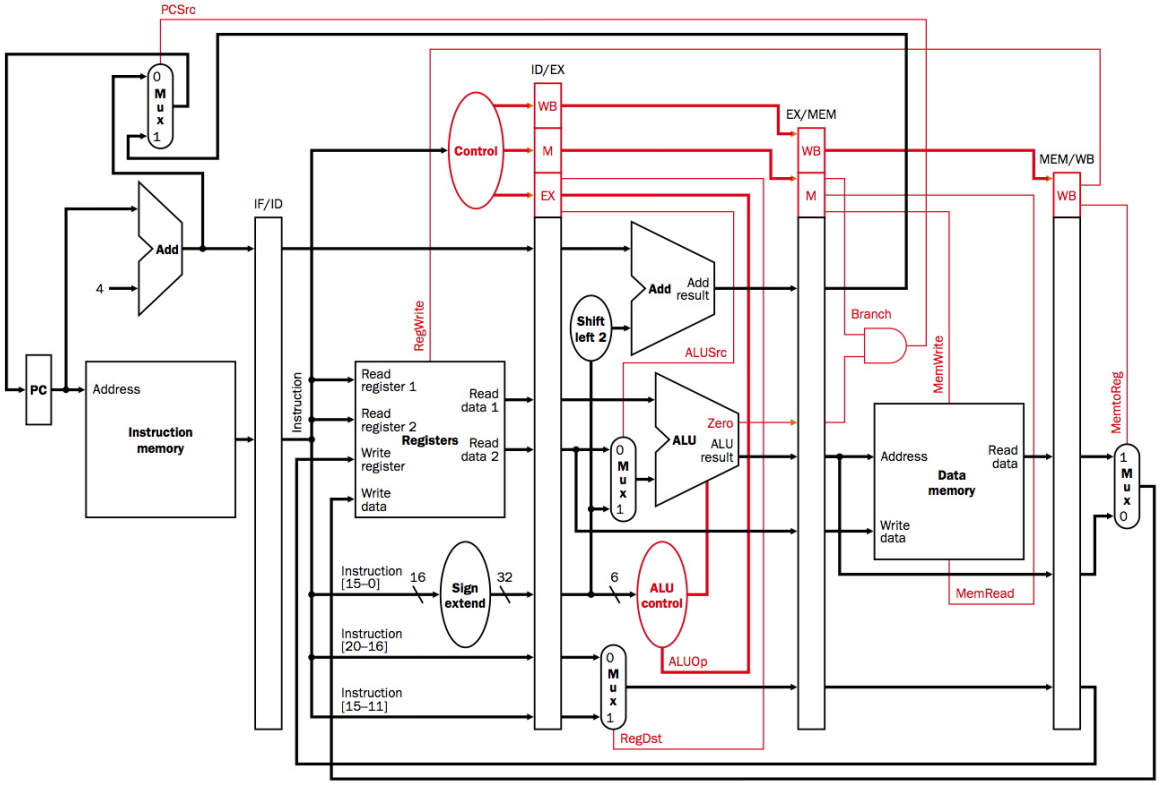



Figura 2.4: Camino de datos segmentado.

Las instrucciones de datos se mueven por regla general de izquierda a derecha a través de las cinco etapas hasta que completan la ejecución.Entre una etapa y la siguiente del camino de datos se sitúan registros de segmentación, para conservar el valor de los campos necesarios de una instrucción individual durante varias etapas. Dichos registros almacenan los valores que necesitará la instrucción en las etapas siguientes para continuar su ejecución. Los registros de segmentación también guardan información de las líneas de control, pues se necesita activar los valores de control en cada etapa de segmentación. Ya que cada línea de control se asocia con un componente activo en una sola etapa, se

pueden dividir las líneas de control en cinco grupos de acuerdo con las etapas de segmentación.

Durante la ejecución de un código en un procesador segmentado se pueden producir riesgos, es decir, situaciones que impiden que se ejecute la siguiente instrucción del flujo de instrucciones durante su ciclo de reloj. Los riesgos reducen el rendimiento de la velocidad ideal lograda por la segmentación. Se distinguen tres tipos de riesgos:

- **Riesgos estructurales:** Surgen de conflictos en los recursos, cuando el hardware no puede soportar todas las combinaciones posibles de instrucciones en ejecuciones solapadas simultáneamente.La solución adoptada en el MIPS consiste en la segmentación de unidades funcionales y duplicación de los recursos para permitir todas las posibles combinaciones de instrucciones. Sin embargo, esta solución sólo es necesaria en el caso de operaciones que precisen de etapas que ocupan más de un ciclo de reloj, como por ejemplo las operaciones en punto flotante.
- **Riesgos de control:** Surgen de la segmentación de los saltos y otras instrucciones que cambian el PC.En el camino de datos de Simula3MS la decisión del salto se toma en la etapa ID (se adelanta la comprobación del salto desde la etapa EX para ganar un ciclo de reloj). En el simulador esta implementada una técnica de salto denominada salto retardado, que consiste en introducir en el cauce la siguiente instrucción al salto y dejar que se ejecute completamente, tanto si el salto se realiza como si no lo hace.
- **Riesgos de datos:** Surgen cuando una instrucción depende de los resultados de una instrucción previa que aún está en el cauce. La solución a este tipo de riesgo consiste en añadir al procesador una unidad de detección de riesgos. La unidad de detección de riesgos puede usar distintas técnicas para evitar el mismo. Simula3MS usa bloqueo o burbuja, que consiste en detener por hardware las instrucciones, insertando burbujas, hasta que se resuelva el riesgo.

La segmentación se puede representar gráficamente mediante dos tipos de diagramas. Los diagramas uniciclo (ver figura 2.5) muestran el estado del camino de datos durante un ciclo de reloj, y las cinco instrucciones del cauce se identifican con etiquetas encima de sus correspondientes etapas. Se usa este tipo de diagrama para mostrar con más detalle que está ocurriendo dentro del cauce durante cada ciclo del reloj. Los diagramas multiciclo (ver figura 2.6) se utilizan para dar una perspectiva general de diferentes situaciones dentro de la segmentación. Se considera que el tiempo avanza de izquierda a derecha y las instrucciones de la parte superior a la inferior del diagrama.

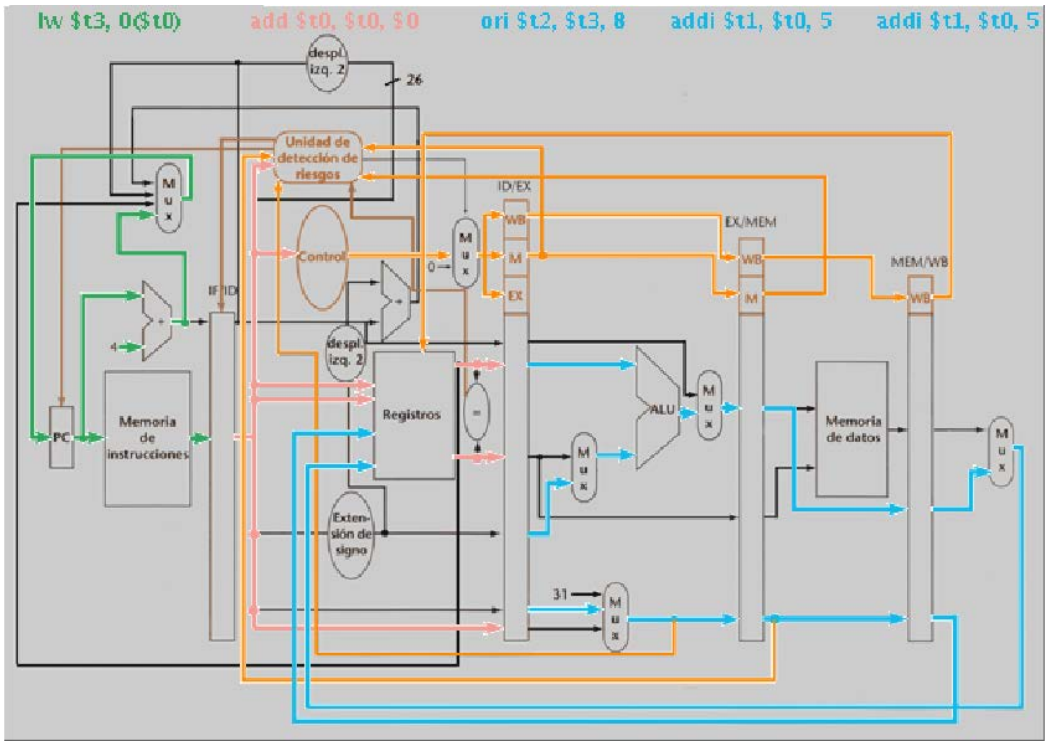



Figura 2.5: Diagrama uniciclo para el camino de datos de un procesador.

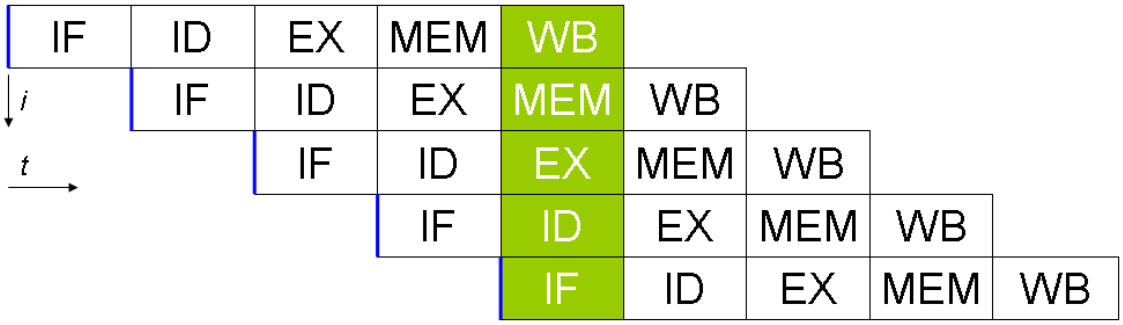



Figura 2.6: Diagrama multiciclo para el camino de datos de un procesador.

# CAPÍTULO 3. PLANTEAMIENTO DEL PROCESO DE DESARROLLO

Después de tratar los aspectos básicos de la arquitectura MIPS, en este capítulo se plantea el proceso de desarrollo para la implementación de nuevas funcionalidades en el simulador MIPS Simula3MS.

Tal y como se describe en el primer capítulo, se parte de una herramienta de finalidad didáctica que se utiliza en diversas asignaturas que tratan la arquitectura y organización del computadores. Obviamente, el uso del Simula3MS ayuda y simplifica la labor docente para que los alumnos entiendan y pueda practicar sobre un simulador real el funcionamiento de la arquitectura MIPS. Esta herramienta, aunque es bastante completa, puede beneficiarse de mejoras en su funcionamiento e incorporación de nuevas funcionalidades. Por eso es muy interesante plantearse qué cambios se le pueden incluir a la herramienta actual para que amplíe y mejore en distintos aspectos.

El capitulo se organiza de la siguiente manera: inicialmente se describe de manera más extensa el estado y funcionalidades que ofrece la herramienta sobre la que se va a trabajar; a continuación se describen los cambios concretos que se plantean para la mejora del simulador, así como la metodología empleada para su desarrollo.

## 3.1 SIMULA3MS

Simula3MS [1] es un simulador cuya principal utilidad es su uso pedagógico por lo que el diseño y la implementación de la herramienta ha intentado paliar los defectos detectados en otras herramientas similares como Spim [11], SimuProc [16] o DLX [4]. Y aunque Simula3MS es ya un simulador útil, aún está en desarrollo.

Simula3MS es un simulador de un procesador RISC (Reduced Instruction Set Computer) [12] [14], que implementa un subconjunto de instrucciones basadas en el repertorio de instrucciones del procesador MIPS [12]. La elección de un procesador RISC frente a uno CISC (Complex Instruction Set Computer) está basada en la mayor relevancia actualmente de este tipo de arquitectura. También cabe destacar que estos procesadores presentan una estructura y un repertorio de instrucciones fácil de comprender. Usando una arquitectura RISC, los estudiantes aprenden los fundamentos básicos del repertorio de instrucciones y de la programación en lenguaje ensamblador en menos tiempo que utilizando un procesador CISC. Simula3MS es un simulador de un procesador configurable, que cuenta con un entorno de trabajo sencillo que permite depurar fácilmente los programas, observar la evolución de la memoria, así como la ejecución de las instrucciones sobre distintos tipos de caminos de datos. El camino de datos puede seleccionarse monociclo, multiciclo o segmentado. La interacción del usuario con la herramienta se hace por medio de una interfaz gráfica implementada con Java [6] [9] [10].

Otro punto importante consiste en poder observar la evolución de todos estos componentes ciclo a ciclo, pero tener también la opción de ejecutar conjuntamente todas las instrucciones y observar únicamente el efecto que produce la ejecución completa del programa cargado.

Simula3MS consta de una primera parte que incluye un editor y que permite cargar un programa ya existente en un fichero o editarlo desde cero en la propia herramienta. En esta herramienta se analizan sintácticamente las instrucciones antes de pasar a la ejecución de las mismas. En esta parte también se permite escoger entre las distintas opciones disponibles para la configuración de los distintos tipos de caminos de datos.Después de analizar sintácticamente las instrucciones se puede seguir la evolución del segmento de datos, así como de los registros y del resto del camino de datos (ver figura 3.1).

En el diseño del simulador se observan dos partes claramente separadas, el modelo y la interfaz gráfica, aunque ambas no son independientes si no que están estrechamente relacionadas. El diagrama de clases de la figura 3.2 muestra una aproximación al conjunto de clases e interfaces, así como las relaciones entre ellos.

Java proporciona una serie de paquetes estándar que facilitan la labor de crear las Interfaces Gráficas de Usuario (GUI) necesarias. Estos paquetes se encuentran englobados dentro de Java Foundation Classes (JFC), de los cuales, los más utilizados son AWT y Swing. Simula3MS utiliza Swing como herramienta para la interfaz gráfica.

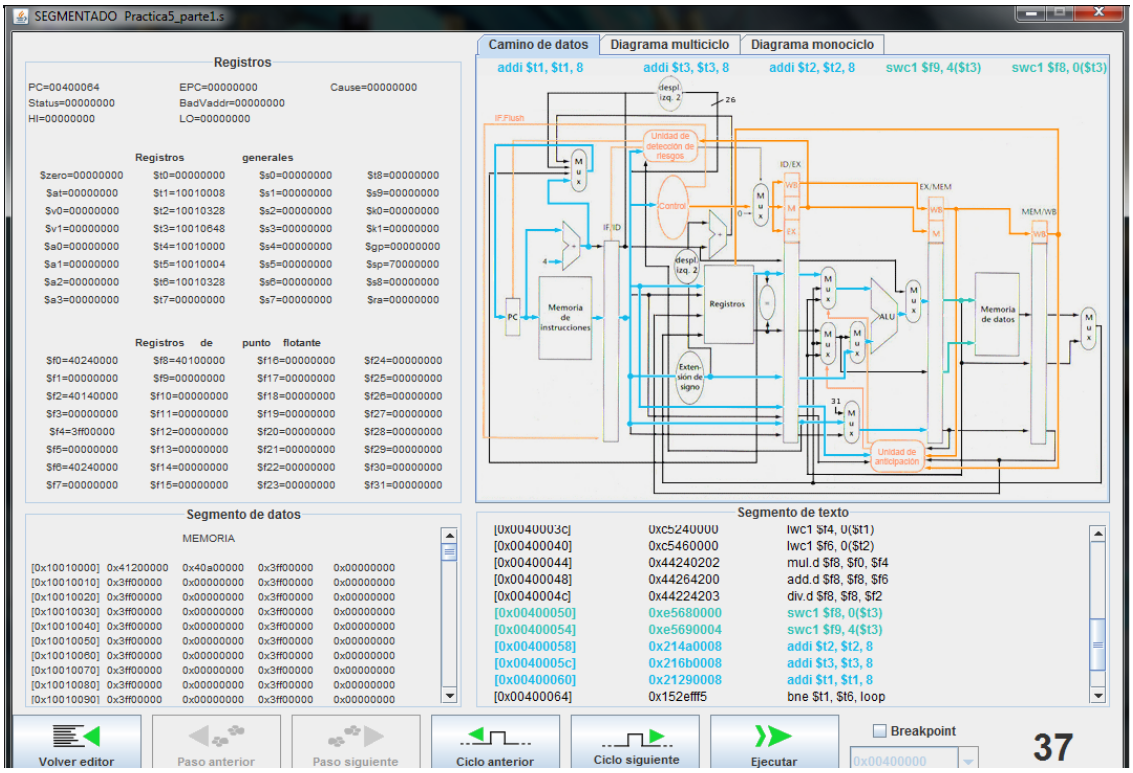



Figura 3.2. Ejemplo de ventana del camino de datos en el Simula3MS.

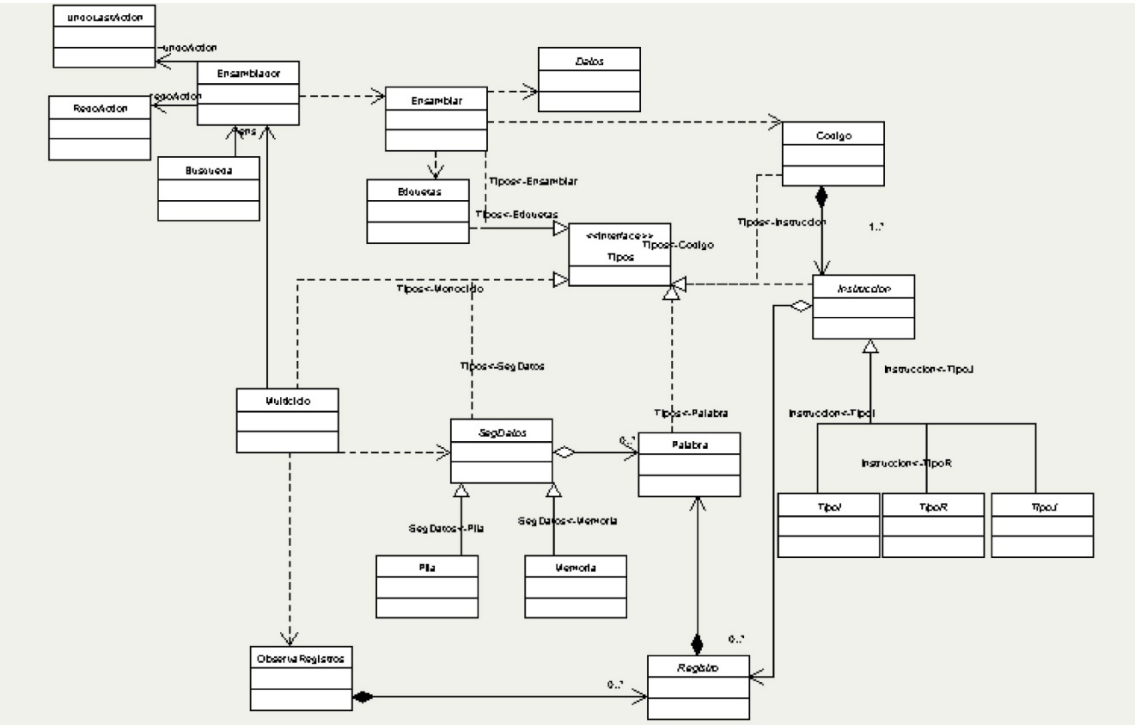



Figura 3.1: Diagrama UML de Simula3MS.

Una vez se ha mencionado lo que nos puede ofrecer la herramienta se pueden detectar una serie de funcionalidades de las que carece que, con el debido estudio y tiempo para desarrollar, se pueden implementar:

- Permitir elegir la codificación de la visualización: hexadecimal, binario, decimal, etc.
- Permitir que se configure el cauce con adelantamiento o sin él.
- Permitir el uso de técnicas especulativas para la ejecución de instrucciones fuera de orden.
- Nuevas estrategias a la hora de resolver los saltos.
- Permitir obtener trazas de los accesos a memoria que se volcarían en un fichero.
- Recogida y visualización de estadísticas de rendimiento.

En este proyecto se abordara el último punto de la lista anterior, es decir, la implementación de recogida y visualización de estadísticas de rendimiento, al igual que la obtención de los accesos a memoria en el Simula3MS como se describe de manera más exhaustiva en el apartado siguiente.

## 3.2 CAMBIOS A INTRODUCIR EN SIMULA3MS

El principal objetivo de este proyecto es modificar y crear nuevas funcionalidades para la mejora del simulador y por lo tanto la mejora de su uso en el ámbito de estudio. Más concretamente, la implementación de recogida y visualización de estadísticas de rendimiento para el camino de datos monociclo, multiciclo y sementado, lo que supone ampliar el simulador en dos aspectos principales:

**Recogida de datos:** esto implica la modificación del código original para que, en tiempo de ejecución, permita recoger una serie de datos necesarios para generar las estadísticas deseadas. Estos datos deben poder obtenerse paso a paso durante la ejecución de un programa, o la obtención de su totalidad cuando se utiliza la funcionalidad de ejecución completa hasta el final. Además los datos que son interesantes de obtener dependen del camino de datos que se está ejecutando, a saber:

- Monociclo:
    - Número de instrucciones ejecutadas.
    - Número de ciclos ejecutados.
    - CPI resultante.
    - Uso de la memoria de instrucciones (ciclos de uso y porcentaje).
    - Uso de la memoria de datos (ciclos de uso y porcentaje).
    - Uso de la ALU (ciclos de uso y porcentaje).

- Multiciclo:
    - Número de instrucciones ejecutadas.
    - Número de ciclos ejecutados.
    - CPI resultante.
    - Uso de la memoria de instrucciones (ciclos de uso y porcentaje).
    - Uso de la memoria de datos (ciclos de uso y porcentaje).
    - Uso de la ALU (ciclos de uso y porcentaje). Se desglosa en:
        - UF de enteros (latencia, ciclos de uso y porcentaje).
        - UF de suma en punto flotante (latencia, ciclos de uso y porcentaje).
        - UF de multiplicación en punto flotante (latencia, ciclos de uso y porcentaje).
        - UF de división en punto flotante (latencia, ciclos de uso y porcentaje).

- Segmentado:
    - Tipo de salto (salto fijo o salto retardado).
    - Número de instrucciones ejecutadas.
    - Número de ciclos ejecutados.
    - CPI resultante.
    - Uso de la memoria de instrucciones (ciclos de uso y porcentaje).
    - Uso de la memoria de datos (ciclos de uso y porcentaje).
    - Uso de la ALU (ciclos de uso y porcentaje). Se desglosa en:
        - UF de enteros (latencia, ciclos de uso y porcentaje).
        - UF de suma en punto flotante (latencia, ciclos de uso y porcentaje).
        - UF de multiplicación en punto flotante (latencia, ciclos de uso y porcentaje).
        - UF de división en punto flotante (latencia, ciclos de uso y porcentaje).
    - Detenciones del cauce (ciclos detenidos y porcentaje). Se desglosa en:
        - Riesgos estructurales (ciclos detenidos y porcentaje).
        - Riesgos de datos (ciclos detenidos y porcentaje).
        - Riesgos de control (ciclos detenidos y porcentaje).

**Visualización de estadísticas:** aparte de la recogida de datos para la generación de estadísticas, éstas se deben poder mostrar al usuario. Esto se podrá hacer de dos maneras:

- Ventana de estadísticas: Esto implica la incorporación de un nuevo botón “Estadísticas” a la herramienta, que abre una ventana independiente donde se

muestran los datos antes descritos para cada tipo de camino de datos. Esta ventana se actualiza en tiempo de ejecución, lo que permite un seguimiento completo de las estadísticas en cada momento.

- Fichero de texto: El programa podrá generar un fichero de texto simple donde escribirá todos los datos obtenidos durante la ejecución total del código, lo que se consigue al pulsar el botón de "Ejecutar" que recorre y finaliza la ejecución. Los datos serán los mismos que ofrece la ventana de estadísticas. Este fichero permitirá tener de manera permanente las estadísticas referentes a un código con un camino de datos y unas opciones determinadas para su futura consulta.

## 3.3 METODOLOGÍA

Para incorporar los cambios planteados en el apartado anterior desde el comienzo del trabajo de ha seguido una metodología que tiene unos pasos muy claros que se han seguido para la finalización del trabajo:

1. **Estudio del código de Simula3MS:** Una vez se han asentado los conocimientos teóricos y se sabe utilizar el simulador, se debe desenmarañar su código fuente. Para ello se utiliza la herramienta de programación Eclipse [17] (ver figura 3.3) donde se puede ver y editar el código de la última versión disponible. Simula3MS está programado en Java [6] [9] [10], lo que supone programación orientada a objetos [4], y en concreto adopta una arquitectura modelo-vista-controlador. Esto requiere un conocimiento extenso en Java para trabajar con una herramienta compleja con miles de líneas de código. Se debe descubrir qué hace cada clase y cada método, sobre todo los relacionados con la ejecución del procesador con camino de datos monociclo, multiciclo y segmentado.
2. **Ampliación y modificación del código Simula3MS:** Una vez identificada la función de cada clase, se necesita la capacidad de crear clases totalmente nuevas y modificar código existente para incluir nuevas funcionalidades a la herramienta. En este caso, añadir la recogida y visualización de estadísticas de rendimiento. Para ello se crea una nueva clase *Estadisticas.java* que crea y muestra estadísticas de rendimiento, además se modifica el código ya existente con el que se visualizaba el camino de datos para los procesadores monociclo, multiciclo y segmentado.

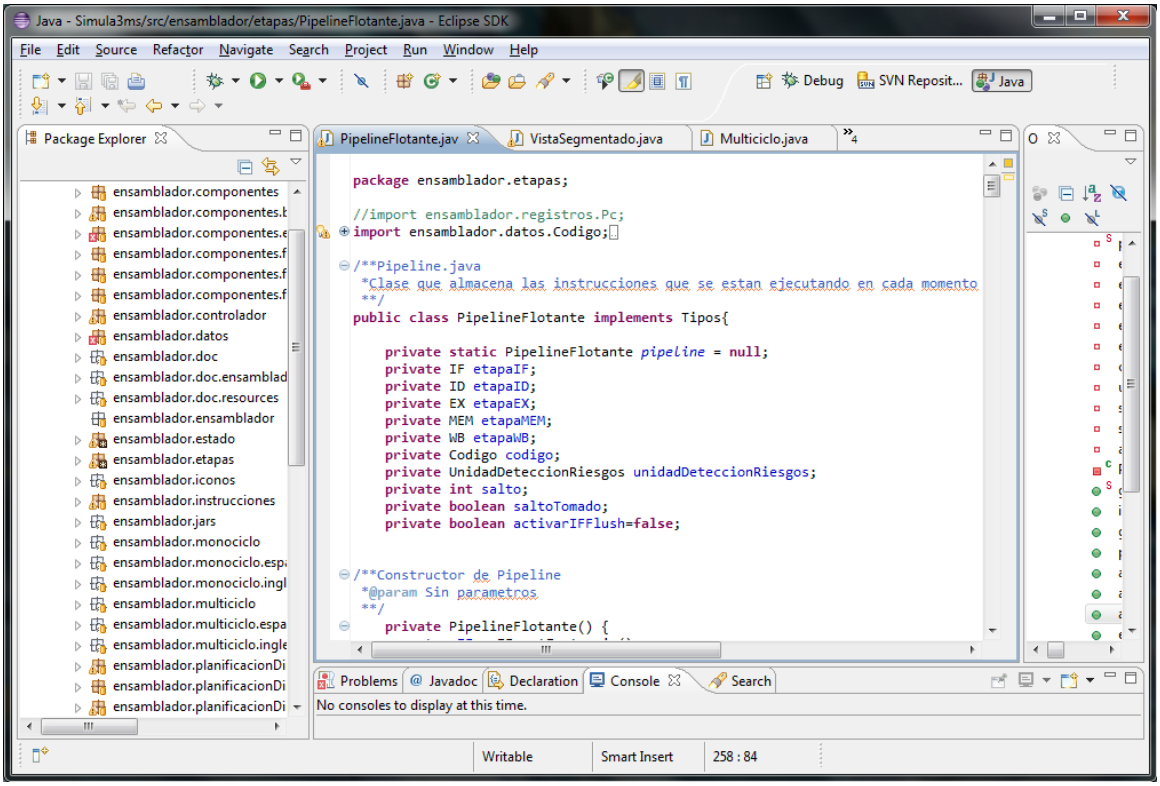



Figura 3.3: Herramienta de programación Eclipse.

# CAPÍTULO 4. DISEÑO E IMPLEMENTACIÓN DE RECOGIDA DE ESTADÍSTICAS

Una vez se tienen claros los objetivos, y habiendo estudiado el código del simulador, se llega a la conclusión que lo mejor para obtener los resultados deseados es crear una clase que se encargue de guardar los datos necesarios para el uso y muestra de estadísticas en tiempo real mientras se ejecuta el simulador. Esta clase se llama *Estadisticas.java*.

En este capítulo primero se describe cómo funciona la nueva clase *Estadisticas.java*, y a continuación se explica los cambios que se han realizado al código original para la inclusión de esta clase para los caminos de datos monociclo, multiciclo y segmentado. Todo el código creado y modificado para la resoucion de este trabajo se puede encontrar en Anexo A: Codigo fuente.

## 4.1 NUEVA CLASE ESTADISTICAS.JAVA

Esta nueva clase que se tiene que crear desde cero se aloja en la subcarpeta /ensamblador/estado/, ya que el uso que se le dará será para consultar los datos que ofrece la ejecución del simulador en un estado concreto, y mediante una serie de métodos, mostrar datos ya tratados al usuario ya sea a través de una ventana que se actualiza en tiempo real o un fichero de texto que se genera al finalizar la ejecución.

Las propiedades más importantes en este método y que ayudan a llevar un seguimiento eficaz son:

- **int numInstrucciones:** representa el número de instrucciones que lee el simulador.
- **int ciclos:** representa el número de ciclos que ejecuta el simulador.
- **int riesgosControl:** representa el número de riesgos de control que se detectan y suponen detenciones en la ejecución del código.
- **int riesgosEstructurales**: representa el número de riesgos estructurales que se detectan y suponen detenciones en la ejecución del código.
- **int riesgosRAW**: representa el numero de riesgos de datos en los que se trata de leer un operando antes de que lo escriba una instrucción anterior y esto supone la detención de la ejecución del código.
- **int riesgosWAW**: representa el número de riesgos de datos en los que una escritura posterior se produce antes que otra escritura anterior en el mismo destino y esto supone la detención de la ejecución del código.
- **int numNOPs**: representa el número detenciones que tienen que inyectarse en la ejecución del código a consecuencia de los riesgos antes mencionados.
- **int memoriaInstruccion**: representa el número de ciclos de uso de la memoria de instrucciones.
- **int memoriaDatos**: representa el número de ciclos de uso de la memoria de datos.
- **int ALU:** representa el número de ciclos de uso total de la ALU.
- **int integer**: representa el número de ciclos de uso de la unidad funcional de enteros.
- **int add**: representa el número de ciclos de uso de la unidad funcional de suma en punto flotante.
- **int mult**: representa el número de ciclos de uso de la unidad funcional de multiplicación en punto flotante.
- **int div**: representa el número de ciclos de uso de la unidad funcional de división en punto flotante.

Todas estas propiedades pueden examinarse y modificarse mediante métodos o funciones miembro, que acceden a dicha propiedad, y pueden ser de dos tipos:

- **get method:** lee el valor de la propiedad.
- **set method:** cambia el valor de la propiedad.

Además para hacer de manera más intuitiva el incremento y decremento de estas propiedades cuando avanzamos y retrocedemos en la ejecución del simulador, cuentan con dos tipos más de métodos para cada propiedad:

- **aumentar”propiedad”:** aumenta en uno el numero actual del valor de la propiedad.
- **disminuir”propiedad”:** disminuye en uno el numero actual del valor de la propiedad.

Una vez se ha visto como se puede manipular las propiedades principales, a continuación se describen los principales métodos que permiten el funcionamiento correcto de la clase Estadísticas:

- **estadísticas():** es el constructor de la clase y llama al método inicializar.
- **inicializar():** da un valor inicial (0 por defecto) a todas las propiedades.
- **ventana(modo):** método que crea la ventana donde el usuario podrá consultar las estadísticas en tiempo real de ejecución. Dependiendo del cauce elegido para la ejecución ("modo") prepara que la ventana contenga los datos referentes al procesador monociclo (modo=1), multiciclo (modo=2) o segmentado (modo=3).
- **actualizar(modo)**: método que actualiza los datos de la ventana de estadísticas. Dependiendo del "modo" actualiza los datos referentes al procesador monociclo (modo=1), multiciclo (modo=2) o segmentado (modo=3).
- **monociclo(estilo, estado):** método que se encarga de actualizar los valores de las propiedades internas de la clase Estadísticas para el procesador monociclo. Dependiendo del "estado" se sabe si se deben utilizar los métodos para aumentar los valores (estado=1), o si se está retrocediendo y se deben disminuir los valores (estado=0). Además utiliza el "estilo" de la instrucción que se está ejecutando en cada momento para saber si se trata de una instrucción tipo R, tipo I o tipo J, y modificar de manera acorde el uso de la ALU o la memoria de datos.
- **multiciclo(estilo, número, estado, etapa):** método que se encarga de actualizar los valores de las propiedades internas de la clase Estadísticas para el procesador multiciclo. Dependiendo del "estado" se sabe si se debe aumentar (estado=1) o disminuir (estado=0) los valores. Como en el procesador multiciclo se puede avanzar etapa a etapa dada una instrucción se debe conocer en que "etapa" se encuentra actualmente, al igual que el "número" y "estilo" de la instrucción, indicadores que sirven para saber cuándo se debe modificar el uso de la memoria de instrucciones, memoria de datos o de qué tipo especifico de unidad funcional hace uso.
- **segmentado(pipeline):** método que se encarga de actualizar los valores de las propiedades internas de la clase Estadísticas para el procesador segmentado. Para ello hace uso del objeto "pipeline" que tiene información del estado de todas las instrucciones que se encuentran en cada etapa y del uso que hacen de las unidades funcionales. De esta manera recorre cada etapa y modifica los valores de manera acorde obteniendo el "estilo" y "número" de cada instrucción.
- **escribirMono():** método que genera un fichero de texto con el nombre del programa en ejecución para el procesador monociclo. En este fichero se escriben todos los datos referentes a las estadísticas de rendimiento recogidos durante la ejecución del simulador.

- **escribirMulti():** método que genera un fichero de texto con el nombre del programa en ejecución para el procesador multiciclo. En este fichero se escriben todos los datos referentes a las estadísticas de rendimiento recogidos durante la ejecución del simulador.
- **escribirSeg():** método que genera un fichero de texto con el nombre del programa en ejecución para el procesador segmentado. En este fichero se escriben todos los datos referentes a las estadísticas de rendimiento recogidos durante la ejecución del simulador.

## 4.2 IMPLEMENTACIÓN EN EL PROCESADOR MONOCICLO

A continuación se explica cómo se ha modificado el código existente del simulador para la implementación y uso de la nueva clase Estadísticas en el procesador monociclo.

Primeramente se identifica qué clases se utilizan para visualizar el simulador cuando ejecuta un código con el procesador monociclo. Esta clase es VistaMonociclo.java, que se encuentra en la subcarpeta /ensamblador/vista/. Después del estudio y comprensión del funcionamiento de esta clase se realizan una serie de modificaciones y diferentes métodos que harán posible el uso de la clase Estadísticas.

En el caso de la implementación monociclo, interesa saber el número de instrucciones y ciclos ejecutados (al ser monociclo, se ejecuta una instrucción por ciclo), el CPI obtenido (debe resultar en 1), y los ciclos de uso y correspondiente porcentaje de la memoria de instrucciones, memoria de datos y la ALU.

Una vez hecho esto se debe importar la clase Estadísticas localizada en la subcarpeta /ensamblador/estado/. A continuación se declara una instancia privada de la clase Estadísticas y se modifica el constructor para crear un nuevo objeto de tipo Estadísticas llamado “stats”. Con este objeto se pueden acceder a los diferentes métodos. En concreto se utilizan los métodos referentes a la recopilación de estadísticas para el procesador monociclo.

El método *botonEjecutarActionPerformed()* completa la ejecución del código asociado al botón de “ejecutar”. Este método se modifica de manera que dentro del bucle que avanza la ejecución instrucción a instrucción, se hace uso del objeto creado anteriormente “stats” llamando al método *monociclo(estilo, estado)* que se encarga de actualizar los valores internos, siendo “estado”=1 ya que se avanza la ejecución. Una vez se ejecuta todo el código se llama al método *ventana(modo)* que actualiza los datos en la ventana de estadísticas, siendo “modo”=1 ya que representa el procesador monociclo, y el

método *escribirMono()* que genera un fichero de texto y escribe en él las estadísticas de rendimiento recogidas.

El método *botonCicloSigActionPerformed()* avanza la ejecución del código un ciclo y está asociado al botón "ciclo siguiente". Este método se modifica para incluir en cada uso la llamada al método *monociclo(estilo, estado),* siendo "estado"=1, y *ventana(modo),* siendo "modo"=1, para que aumente las variables del objeto "stats" y se representen los nuevos datos en la ventana de estadísticas respectivamente.

El método *botonCicloAntActionPerformed()* retrocede la ejecución del código hasta el ciclo anterior, está asociado al botón "ciclo anterior". Este método se modifica para que de manera similar incluya en cada uso los métodos *monociclo(estilo, estado)* y *ventana(modo)* con la diferencia de que "estilo"=0 ya que se está retrocediendo la ejecución y se deben disminuir las variables en el objeto "stats".

Otra de las modificaciones realizadas en la clase VistaMonociclo es la inclusión de un nuevo botón "Estadísticas" que es el que permite abrir y cerrar la ventana de estadísticas para que el usuario pueda ver como se recopilan los datos en tiempo de ejecución paso a paso o ejecutando todo el código hasta el final.Para ello se modifica la interfaz gráfica en la clase Vista.java que se encuentra en la misma subcarpeta /ensamblador/vista/. Aquí se declara un nuevo tipo de botón llamado "botonEstadisticas" y una vez definidas sus características se incorpora para que se aloje a la derecha del "segmento de texto" donde había un espacio adecuado que no estropea la armonía del resto de paneles. Al nuevo botón se le asigna un icono estadísticas.png previamente creado para que se asemeje lo máximo posible a los botones ya existentes.

Una vez definido el botón, de nuevo en la clase VistaMonociclo se le atribuye una funcionalidad mediante el método *botonEstadisticasActionPerformed(),* que se encarga de abrir la ventana de estadísticas, o cerrar en el caso de que se pulse el botón y ya se encuentre abierto.

## 4.3 IMPLEMENTACIÓN EN EL PROCESADOR MULTICICLO

A continuación, y de manera similar que en el procesador monociclo, se explica qué código existente del simulador se ha modificado para la implementación y uso de la nueva clase Estadísticas en el procesador multiciclo.

Primeramente se identifica qué clases se utilizan para visualizar el simulador cuando ejecuta un código con el procesador multiciclo. Esta clase es Multiciclo.java, que se encuentra en la carpeta principal /ensamblador/. Después del estudio y comprensión del

funcionamiento de esta clase se realizan una serie de modificaciones y se crean diferentes métodos que harán posible el uso de la clase Estadísticas.

Para la implementación multiciclo es interesante saber las mismas estadísticas que se recogen con la ejecución monociclo, a saber: número de instrucciones, número de ciclos, CPI, y ciclos de uso y porcentaje de la memoria de instrucciones, memoria de datos y ALU; y además nuevos datos como los ciclos de uso y porcentaje de las unidades funcionales de enteros, suma en punto flotante, multiplicación en punto flotante y división en punto flotante, siendo la suma del uso de estas unidades funcionales igual a la totalidad de ciclos de uso de la ALU. Al ser el cauce multiciclo una instrucción puede requerir varios ciclos de reloj, lo cual, y a diferencia del monociclo, se debe tener en cuenta de manera separada el número de instrucciones y ciclos ejecutados.

Primero se debe importar la clase Estadísticas localizada en la subcarpeta /ensamblador/estado/. A continuación se declara una instancia privada de la clase Estadísticas y se modifica el constructor para crear un nuevo objeto de tipo Estadísticas llamado “stats”. Con este objeto se pueden acceder a los diferentes métodos, en concreto se utilizan los métodos referentes a la recopilación de estadísticas para el procesador multiciclo.

El método *botonEjecutarActionPerformed()* completa la ejecución del código asociado al botón de “ejecutar”. Este método se modifica de manera que dentro del bucle que avanza la ejecución paso a paso, se hace uso del objeto creado anteriormente “stats” llamando al método que aumenta el número de ciclos *aumentarCiclos()* y al método *multiciclo(estilo, número, estado, etapa)* que se encarga de actualizar los valores internos, siendo “estado”=1 ya que se avanza la ejecución, y “estilo”, “número” y “etapa” datos referentes al estado de la instrucción actual de cada iteración. Una vez se ejecuta todo el código se llama al método *ventana(modo)* que actualiza los datos en la ventana de estadísticas, siendo “modo”=2 ya que representa el procesador multiciclo, y el método *escribirMulti()* que genera un fichero de texto y escribe en él las estadísticas de rendimiento recogidas.

El método *botonCicloSigActionPerformed()* avanza la ejecución del código un ciclo y está asociado al botón “ciclo siguiente”. Funciona como una sola iteración del método visto anteriormente para ejecutar todo el código. Este método se modifica para incluir en cada uso la llamada al método *multiciclo(estilo, número, estado, etapa)*, siendo “estado”=1, y “estilo”, “número” y “etapa” datos referentes al estado de la instrucción actual. Finalmente se llama al método *ventana(modo)*, siendo “modo”=2, para que aumente las variables del objeto “stats” y se representen los nuevos datos en la ventana de estadísticas, respectivamente.

El método *botonCicloAntActionPerformed()* retrocede la ejecución del código hasta el ciclo anterior, por lo que está asociado al botón “ciclo anterior”. Este método se modifica

para que incluya en cada uso los métodos *multiciclo(estilo, número, estado, etapa)* y *ventana(modo)* con la diferencia de que “estilo”=0 ya que se está retrocediendo la ejecución y se deben disminuir las variables en el objeto “stats”.

El método *botonPasoSigActionPerformed(),* asociado al botón “paso siguiente” aumenta la ejecución una instrucción. Esto supone que, dada una instrucción que se encuentra en una etapa X de su ejecución, se deberá avanzar un número de etapas hasta que termine su ejecución y obtengamos una nueva instrucción. Por cada etapa que avanza se modifica para que haga uso de los métodos del objeto “stats” *aumentarCiclos()* y *multiciclo(estilo, número, estado, etapa),* aumentando así las variables adecuadas. Finalmente se llama al método *ventana(modo)* que actualizará los datos en la ventana de estadísticas.

El método *botonCicloAntActionPerformed(),* asociado al botón “paso anterior”, retrocede la ejecución en una instrucción. Esto supone que, de manera similar a cuando se avanza un paso, dada una instrucción en una etapa X, se debe retroceder la ejecución un número de etapas hasta que alcance a la instrucción anterior. Se modifica el código para que por cada etapa que se retroceda se hace uso de los métodos *disminuirCiclos()* y *multiciclo(estilo, número, estado, etapa),* siendo “estado”=0 ya que se deben disminuir las variables. Por último, se llama al método *ventana(modo)* que actualiza la ventana de estadísticas con los nuevos datos recogidos.

A diferencia del procesador monociclo, la interfaz gráfica está definida en la misma clase Multiciclo.java, luego no se cambia de clase para realizar los cambios pertinentes para la inclusión de un nuevo botón “Estadísticas” igual que se ha visto en el procesador monociclo.

Se declara un nuevo tipo de botón llamado “botonEstadisticas”, se definen sus características y se incorpora para que se aloje a la derecha del “segmento de texto”. Una vez creado el botón se le atribuye una funcionalidad mediante el nuevo método *botonEstadisticasActionPerformed(),* que se encargará de abrir y cerrar la ventana de estadísticas.

## 4.4 IMPLEMENTACIÓN EN EL PROCESADOR SEGMENTADO

En este apartado se explica qué código existente del simulador se ha modificado para la implementación y uso de la nueva clase Estadísticas en el procesador segmentado.

Primero se identifica qué clases se utilizan para visualizar el simulador cuando ejecuta un código con el procesador segmentado. A diferencia de para el procesador

monociclo o multiciclo, el segmentado es más complejo, luego requiere de más clases, sobre todo para controlar el pipeline que da información sobre el estado de todas las etapas.

La clase principal es VistaSegmentado.java, que se encuentra en la subcarpeta /ensamblador/vista/ al igual que el monociclo. Esta clase principal se ayuda de una clase controlador llamada ControladorSegmentado.java que se encuentra en la subcarpeta /ensamblador/controlador/ y a su vez hace uso de las clases incluidas en la subcarpeta /ensamblador/etapas/ que contiene las clases referentes a las etapas de un procesador segmentado, el pipeline que lo controla, al igual que las unidades de detección y anticipación. Estas clases son: Etapa.java, IF.java, ID.java, EX.java, MEM.java, WB.java, UnidadAnticipacion.java, UnidadDetencionRiesgos.java y PipelineFlotante.java. Después del estudio y comprensión del funcionamiento de estas clases se realizan una serie de modificaciones y se crean diferentes métodos que harán posible el uso de la clase Estadísticas.

Para la implementación segmentada interesa saber las mismas estadísticas que se recogen en el modo monociclo y multiciclo, pero también es interesante recoger datos respecto a los riesgos que se producen con la segmentación. De esta manera se incorpora la recogida de datos relevantes a detenciones, riesgos estructurales, riesgos de datos y riesgos de control. Además se tiene en cuenta que se pueden declarar más de una unidad funcional del mismo tipo si ésta no se encuentra segmentada, lo que supone tener un seguimiento individual para cada una de las unidades funcionales independientemente de su tipo.

Despues se debe importar la clase Estadísticas localizada en la subcarpeta /ensamblador/estado/ a las clases VistaSementado.java, ControladorSegmentado.java, PipelineFlotante.java y UnidadDeteccionRiesgos.java para que pueda acceder al objeto Estadísticas que se creará. A continuación se declara una instancia privada de la clase Estadísticas en VistaSegmentado.java y se modifica el constructor para crear un nuevo objeto de tipo Estadísticas llamado “stats”. Con este objeto se puede acceder a los diferentes métodos y se envía a las demás clases antes mencionadas para su uso. En concreto, se utilizan los métodos referentes a la recopilación de estadísticas para el procesador segmentado.

El método *botonEjecutarActionPerformed()* de VistaSegmentado.java completa la ejecución del código y está asociado al botón “ejecutar”. En este método hay un bucle que avanza un ciclo cada vez hasta que acaba la ejecución. De esta manera modificamos el código para que en cada iteración, cuando el controlador debe avanzar el pipeline un ciclo, le pasamos el objeto “stats” actual, lo cual como se verá más adelante en este apartado, acaba llamando al método *segmentado(pipeline)* explicado en el apartado 4.1.

El resultado es que por cada iteración se actualizan los valores del objeto “stats”, para finalmente llamar al método *actualizar(modo)*, donde modo=3, que actualizará el valor de

los datos en la ventana de estadísticas, y al método *escribirSeg()* que crea el fichero de texto con las estadísticas de rendimiento obtenidos en la ejecución.

El método *botonCicloSigActionPerformed()* de VistaSegmentado.java avanza la ejecución del programa un ciclo y está asociado al botón "ciclo siguiente". Este método es como una iteración del bucle mencionado en el método que ejecuta todo el código, de esta manera se modifica para que cuando el controlador avance el pipeline, se le pasa por parámetros el objeto "stats" actual para acabar llamando a *segmentado(pipeline)* y actualizar los datos necesarios. Una vez actualizados los datos se llama al método *actualizar(modo)* , donde modo=3, que actualiza los datos que se ven en la ventana de estadísticas.

El método *botonCicloAntActionPerformed()* de VistaSementado.java, asociado al botón "ciclo anterior", retrocede la ejecución del programa un ciclo. A diferencia de cómo funciona el retroceso de la ejecución para el procesador monociclo y multiciclo, el procesador segmentado al ser más complejo, ya que maneja la información de diferentes instrucciones en diferentes etapas, además de los riesgos que produce la segmentación, no utiliza una variable "estado" que indique si se está avanzando o retrocediendo la ejecución. De esta manera para retroceder la ejecución, lo que hace es almacenar en una variable un "contador" que representa el número de ciclos realizados hasta ahora menos uno. Después inicializa los valores del procesador segmentado llamando a los métodos *inicializando()* y *stats.inicializar(),* lo que representa volver el procesador a su estado inicial. Una vez hecho esto, entra en un bucle parecido al que ocurre cuando se ejecuta la totalidad del código, pero se para cuando llega al ciclo indicado por el "contador", resultando efectivamente en el retroceso de la ejecución un ciclo de cara al usuario.

Para incluir la recogida de estadísticas, se modifica el bucle para que en cada iteración se pasa al controlador el objeto "stats" y se llama al método *segmentado(pipeline).* Al final se llama al método *actualizar(modo)* siendo modo=3, que actualiza los datos en la ventana de estadísticas.

El método *avanzarPipeline(instrucción, ciclo, stats)* que se encuentra en la clase PipelineFlotante.java es el que se encarga de avanzar el pipeline un ciclo. Este es el método que el controlador llama cuando se avanza la ejecución ya sea ciclo a ciclo o en cada iteración del bucle para la ejecución completa. Este método recibe la "instrucción" que se añade al pipeline, el número de "ciclo" actual en el que se encuentra el procesador y, para la versión modificada, el objeto "stats" que contiene los datos de estadísticas de rendimiento que se han recopilado hasta ese momento. Para la ejecución de este método primero comprueba si hay salto y si este es "salto tomado". De ser así, se elimina la instrucción en la etapa IF, lo que disminuye el número de instrucciones ejecutadas y aumenta los riegos de control del objeto "stats". Independientemente de esta acción, se comprueba si la instrucción que va a entrar en el flujo es *syscall* (final del programa), si no lo es, comprueba si el flujo se va a detener llamando a la unidad de detección de riesgos, en

caso de que detecte un riesgo que produce la detención del cauce (se inserta una burbuja en la etapa EX), se aumentan los riesgos de control o datos del objeto “stats” dependiendo en cada caso del origen de la detención.

Si el cauce no se detiene se empieza a ejecutar las instrucciones y avanzar el flujo empezando por la etapa WB. Cuando llega a la etapa EX comprueba si la unidad funcional que utilizará la instrucción que entra a EX está ocupada, de ser asi aumenta el número de riesgos estructurales del objeto “stats” e inserta una burbuja en la etapa EX, si la unidad funcional correspondiente no está ocupada se inserta la instrucción que provenía de IF, una nueva instrucción entra en la etapa IF y se aumenta el número de instrucciones y el uso de la memoria de instrucciones del objeto “stats”.

De la misma manera que ocurre con el procesador multiciclo, la interfaz gráfica está definida en la misma clase VistaSegmentado.java. Para insertar el botón “Estadísticas” se declara un nuevo tipo de botón llamado “botonEstadisticas”, se definen sus características y se incorpora para que se aloje a la derecha del “segmento de texto”. Una vez creado el botón se le atribuye una funcionalidad mediante el nuevo método *botonEstadisticasActionPerformed()*, que se encarga de abrir y cerrar la ventana de estadísticas.

# CAPÍTULO 5. EJEMPLOS DE USO

En este capítulo se muestran los resultados ofrecidos por el simulador Simula3MS con las modificaciones que se ha desarrollado en este trabajo. El trabajo final dessarrolado en este TFG permite generar y mostrar una serie de estadísticas de rendimiento al usuario. Este capítulo se divide en tres partes, una por cada camino de datos modificado, siendo estos monociclo, multiciclo y segmentado, en donde se explicara la ejecución de código en el simulador y se ilustrará con las figuras necesarias para el mejor entendimiento de la funcionalidad añadida.

## 5.1 EJEMPLO PARA EL CAMINO DE DATOS MONOCICLO

Para el procesador monociclo se utiliza un código MIPS llamado *APXPY_MIPS.s*. Este código se encuentra en el libro de prácticas de alumnos para la asignatura de Organización de Computadores [2].

Primero se ejecuta la aplicación Simula3MS, después se carga el código *APXPY_MIPS.s*, una vez ensamblado se elige en la configuración que se va a usar el camino de datos monociclo y finalmente se pulsa ejecutar (ver figura 5.1).

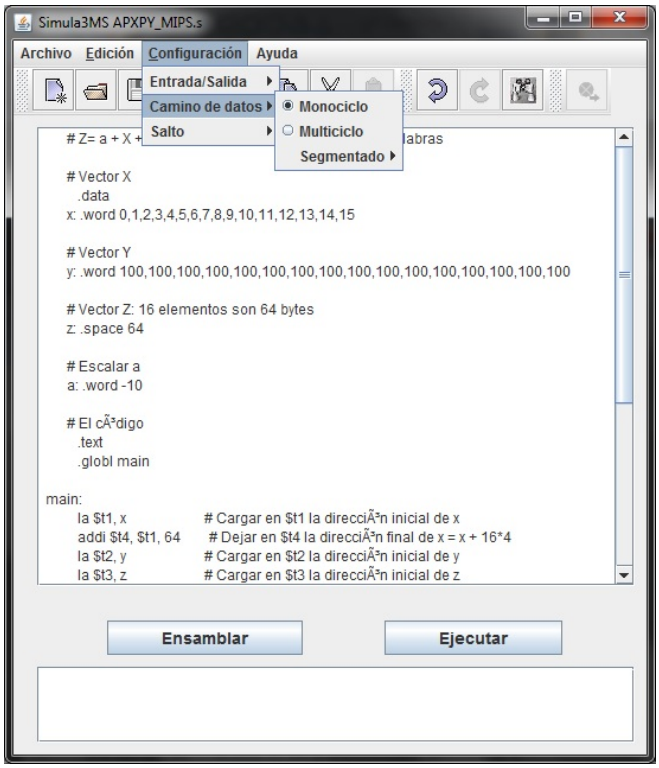



Figura 5.1: Ventana de edición para el camino de datos monociclo.

Una vez en la ventana del camino de datos monociclo se puede ejecutar la simulación, para ello se avanza o retrocede ciclo a ciclo o ejecutando todo el código hasta el final. En esta ventana del simulador se dispone ahora de un nuevo botón “Estadísticas” situado a la derecha del segmento de texto. Esta ventana nos ofrece un seguimiento en tiempo real de las estadísticas más interesantes para una simulación con el camino de datos monociclo (ver figura 5.2), estas son:

- Numero de instrucciones ejecutadas.
- Numero de ciclos ejecutados.
- CPI resultante.
- Uso de la memoria de instrucciones (ciclos de uso y porcentaje).
- Uso de la memoria de datos (ciclos de uso y porcentaje).
- Uso de la ALU (ciclos de uso y porcentaje).

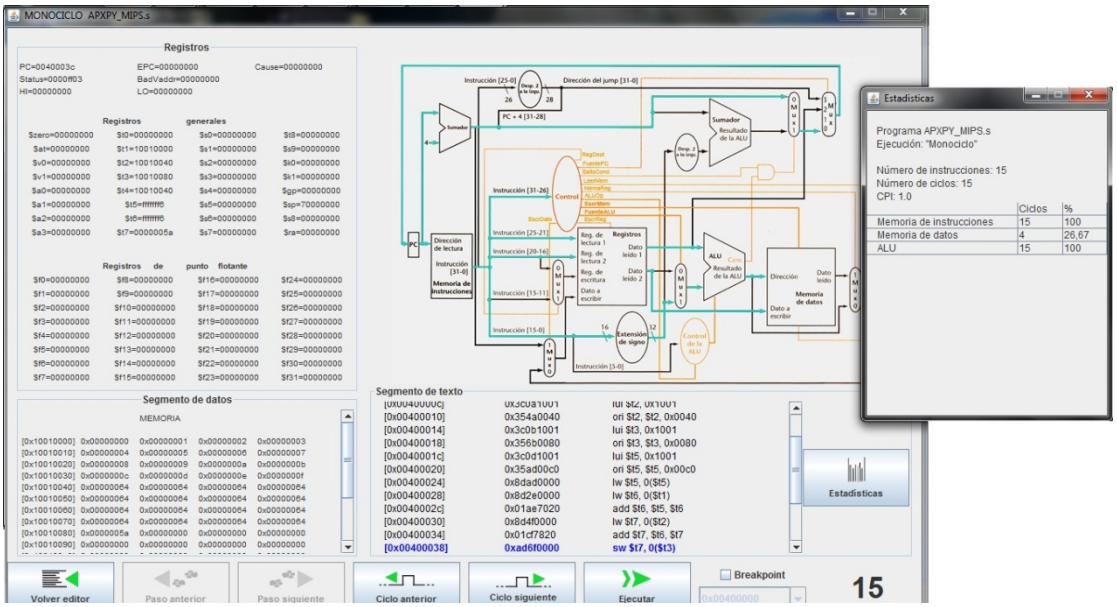



Figura 5.2: Ventana del camino de datos monociclo.

Una vez terminada la ejecución se observan las estadísticas de rendimiento finales (ver figura 5.3). Además se ha generado un fichero de texto llamado *MONOCICLO-APXPY_MIPS.s_estadisticas.txt*; donde se recogen las estadísticas resultantes de la ejecución completa del código (ver figura 5.4).

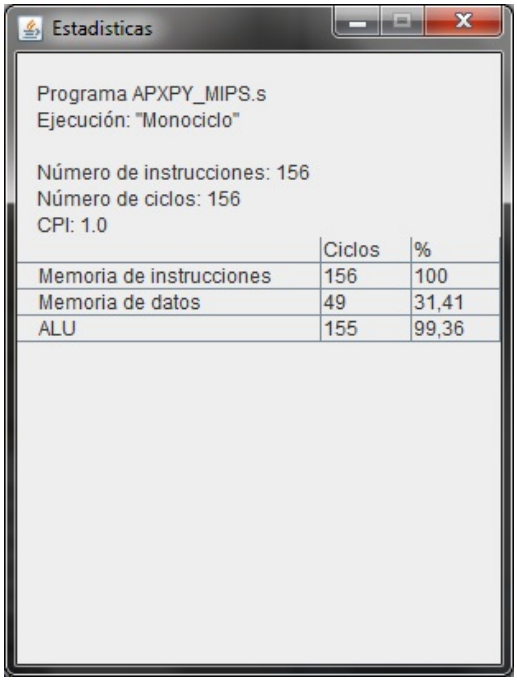



Figura 5.3: Ventana de estadísticas en el camino de datos monociclo.

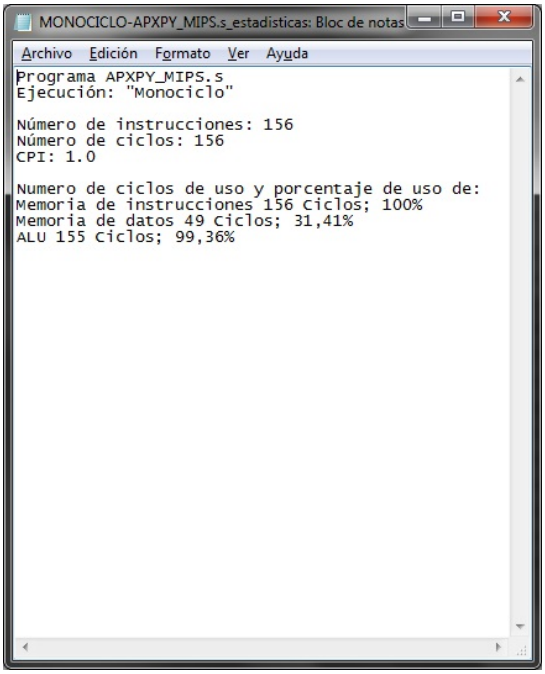



Figura 5.4: Archivo de texto de estadísticas en el camino de datos monociclo.

## 5.2 EJEMPLO PARA EL CAMINO DE DATOS MULTICICLO

Para el procesador multiciclo se va a utilizar un código llamado *Practica5_parte1.s*, correspondiente al código utilizado en la práctica 5 del libro de prácticas de alumnos para la asignatura de Organización de Computadores [2].

Primero se ejecuta la aplicación Simula3MS y después se carga el código *Practica5_parte1.s* Una vez ensamblado se elige en la configuración que se va a usar el camino de datos multiciclo. Esto abre una nueva ventana donde se elige la latencia para las unidades funcionales de suma en punto flotante, multiplicación en punto flotante y división en punto flotante. Para este ejemplo de uso se elige 2, 4 y 5 ciclos respectivamente (ver figura 5.5). Finalmente se pulsa ejecutar.

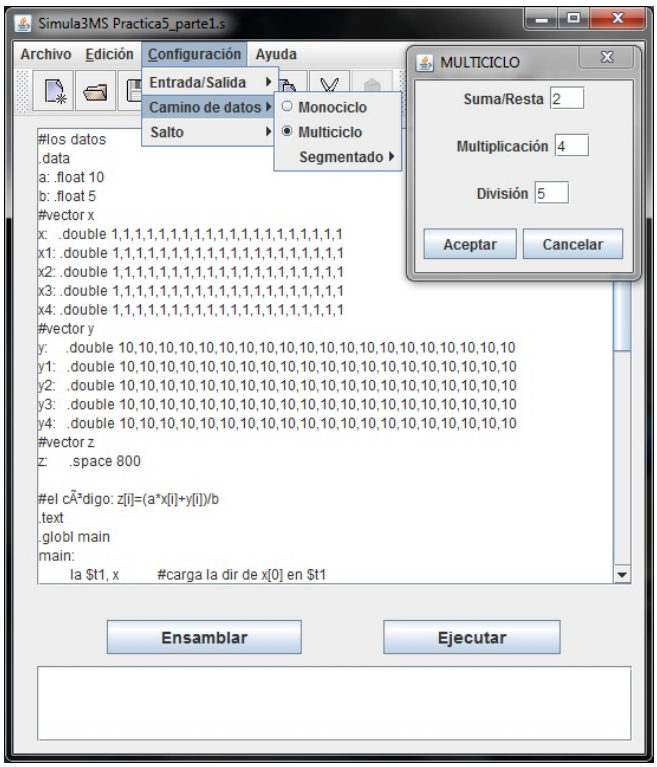



Figura 5.5: Ventana de edición para el camino de datos multiciclo.

Dentro de la ventana del camino de datos multiciclo se puede ejecutar la simulación, para ello se puede avanzar o retroceder paso a paso, ciclo a ciclo o ejecutando todo el código hasta el final. Al igual que se explico para el camino de datos monociclo, aquí también con la modificación a la aplicación realizada en este trabajo se dispone ahora de un nuevo botón “Estadísticas” situado a la derecha del segmento de texto. Esta ventana (ver figura 5.6) nos ofrece un seguimiento en tiempo real de las estadísticas para una simulación con el camino de datos multiciclo, estas son:

- Número de instrucciones ejecutadas.
- Número de ciclos ejecutados.
- CPI resultante.
- Uso de la memoria de instrucciones (ciclos de uso y porcentaje).
- Uso de la memoria de datos (ciclos de uso y porcentaje).
- Uso de la ALU (ciclos de uso y porcentaje). Esto a su vez se desglosa en:
    - UF de enteros (latencia, ciclos de uso y porcentaje).
    - UF de suma en punto flotante (latencia, ciclos de uso y porcentaje).
    - UF de multiplicación en punto flotante (latencia, ciclos de uso y porcentaje).
    - UF de división en punto flotante (latencia, ciclos de uso y porcentaje).

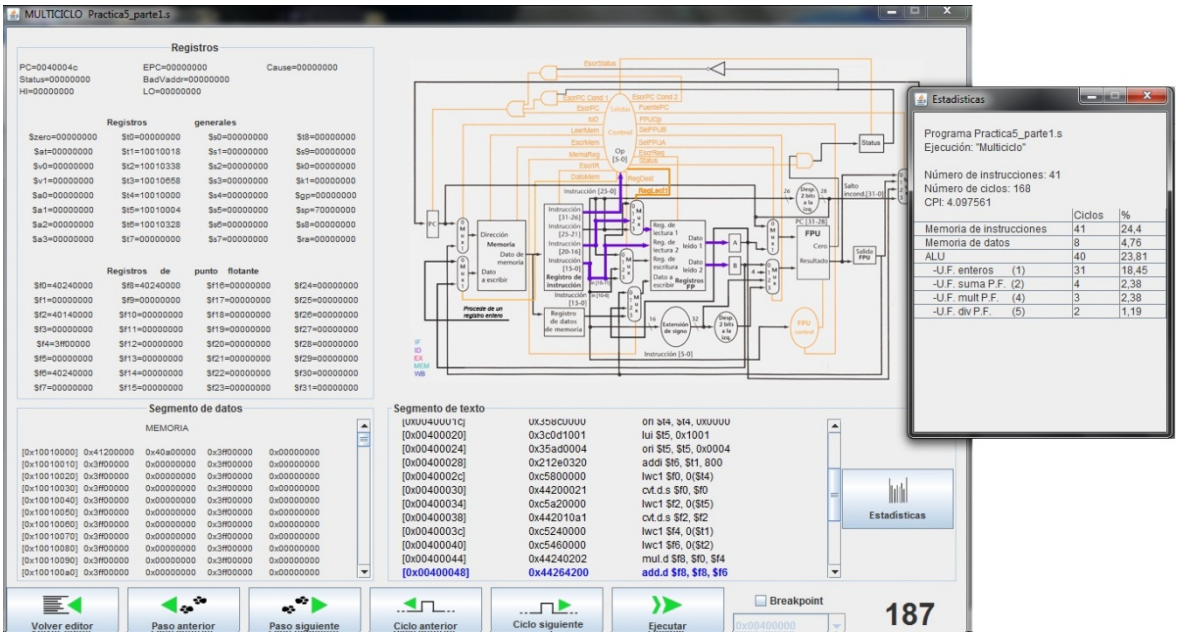



Figura 5.6: Ventana del camino de datos multiciclo.

Al igual que en el procesador monociclo, después de terminar la ejecución completa se observan las estadísticas de rendimiento finales (ver figura 5.7). Además se genera un nuevo fichero de texto llamado *MULTICICLO-Practica5_parte1.s_estadisticas*, donde fichero también se recogen las estadísticas, resultantes de la ejecución completa del código (ver figura 5.8).

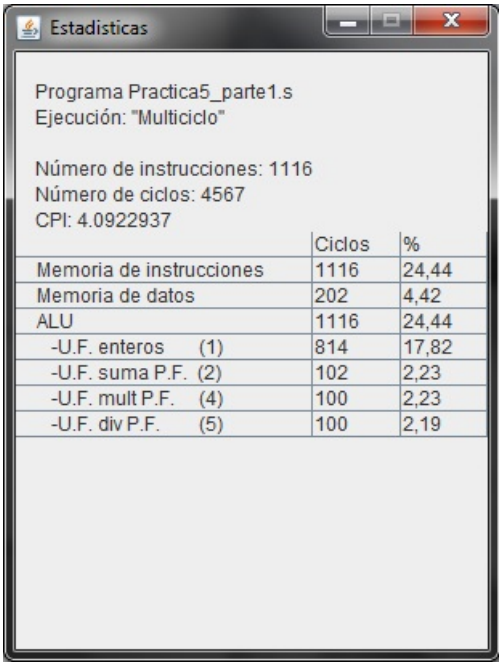



Figura 5.7: Ventana de estadísticas en el camino de datos multiciclo.

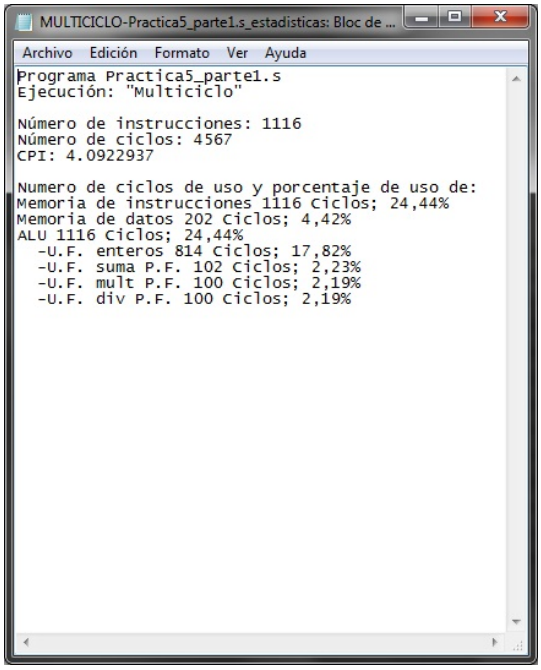



Figura 5.8: Archivo de texto de estadísticas en el camino de datos multiciclo.

## 5.3 EJEMPLO PARA EL CAMINO DE DATOS SEGMENTADO

Para el último ejemplo de uso se utiliza el procesador segmentado. Como en el caso del multiciclo, se utiliza un código llamado *Practica5_parte1.s*, correspondiente al código descrito en la práctica 5 del libro de prácticas de alumnos para la asignatura de Organización de Computadores [2].

Primero se ejecuta la aplicación Simula3MS, después se carga el código *Practica5_parte1.s*. Una vez ensamblado se elige en la configuración que se va a usar el camino de datos segmentado son salto retardado, esto abre una nueva ventana donde se elige para cada unidad funcional, si ésta se encuentra segmentada, y en caso contrario cuantas unidades habrá disponibles para cada una de ellas y además se indica la latencia de cada una. Para este ejemplo de uso se elige una unidad funcional segmentada para la suma en punto flotante con 2 ciclos de latencia, dos unidades funcionales para la multiplicación en punto flotante sin segmentar con 4 ciclos de latencia y una unidad funcional para la división en punto flotante sin segmentar con 5 ciclos de latencia (ver figura 5.9). Finalmente se pulsa ejecutar.

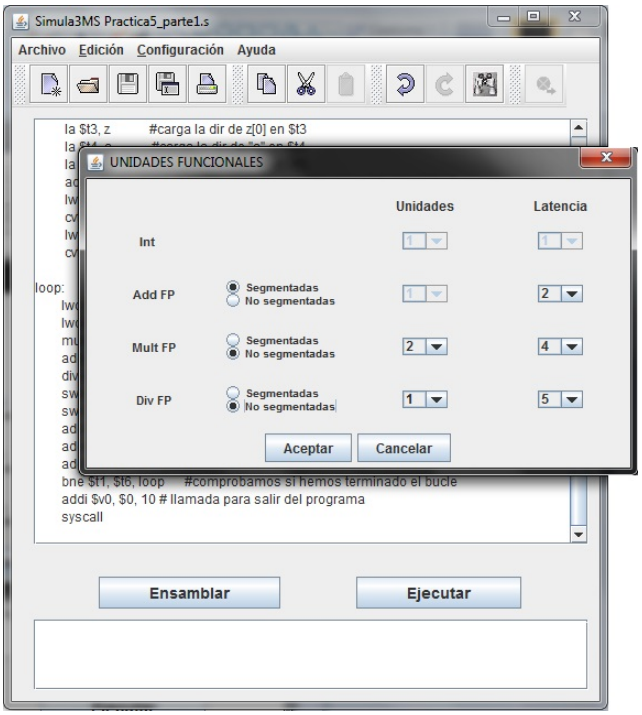



Figura 5.9: Ventana de edición para el camino de datos segmentado.

Dentro de la ventana del camino de datos segmentado se puede ejecutar la simulación, para ello se puede avanzar o retroceder ciclo a ciclo o ejecutando todo el código hasta el final. Al igual que con los caminos de datos anteriores, con la modificación realizada en este trabajo se dispone ahora de un nuevo botón “Estadísticas” situado a la derecha del segmento de texto. Esta ventana (ver figura 5.10) nos ofrece un seguimiento en tiempo real de las estadísticas para una simulación con el camino de datos segmentado, estas son:

- Tipo de salto (salto fijo o salto retardado).
- Número de instrucciones ejecutadas.
- Número de ciclos ejecutados.
- CPI resultante.
- Uso de la memoria de instrucciones (ciclos de uso y porcentaje).
- Uso de la memoria de datos (ciclos de uso y porcentaje).
- Uso de la ALU (ciclos de uso y porcentaje). A su vez esto se desglosa en:
    - UF de enteros (latencia, ciclos de uso y porcentaje).
    - UF de suma en punto flotante (latencia, ciclos de uso y porcentaje).
    - UF de multiplicación en punto flotante (latencia, ciclos de uso y porcentaje).
    - UF de división en punto flotante (latencia, ciclos de uso y porcentaje).
- Detenciones del cauce (ciclos detenidos y porcentaje). A su vez se desglosa en:
    - Riesgos estructurales (ciclos detenidos y porcentaje).

- Riesgos de datos (ciclos detenidos y porcentaje).
- Riesgos de control (ciclos detenidos y porcentaje).

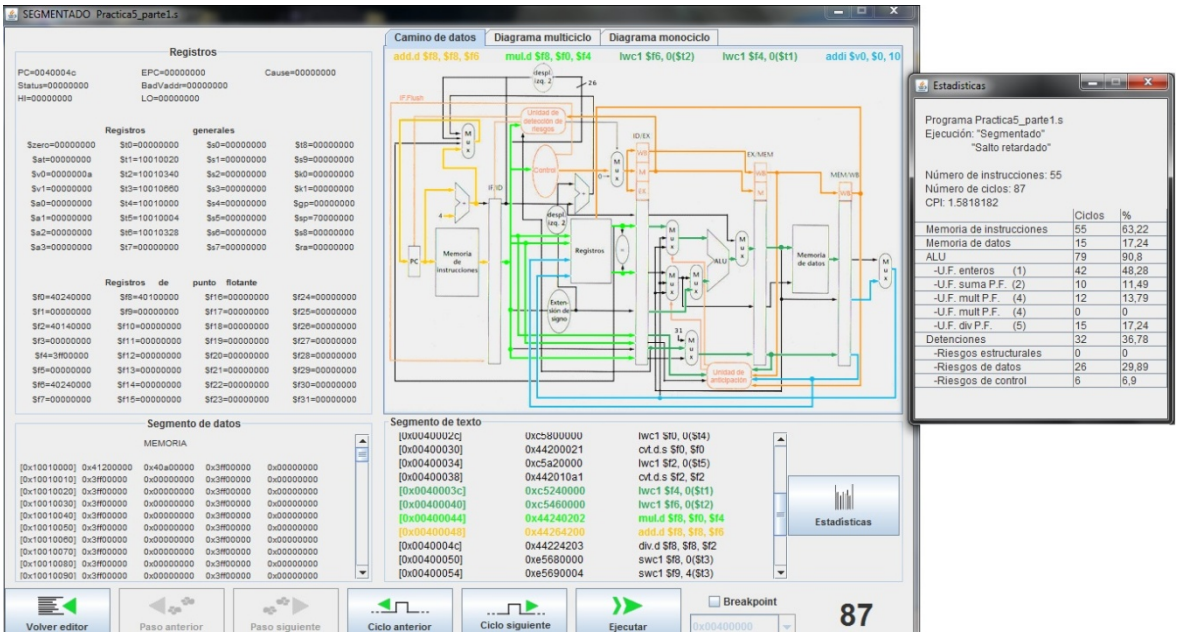
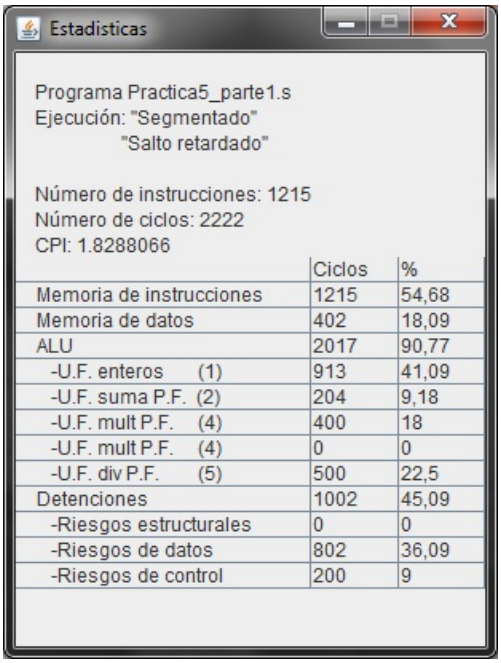



Figura 5.10: Ventana del camino de datos segmentado.

Una vez terminada la ejecución completa se pueden observar las estadísticas de rendimiento finales (ver figura 5.11), además estas estadísticas se recogen en un nuevo fichero de texto que se genera llamado SEGMENTADO-*Practica5_parte1.s_estadisticas (ver figura 5.12).*




| | Ciclos | % |
|---|---|---|
| Memoria de instrucciones | 1215 | 54,68 |
| Memoria de datos | 402 | 18,09 |
| ALU | 2017 | 90,77 |
| -U.F. enteros (1) | 913 | 41,09 |
| -U.F. suma P.F. (2) | 204 | 9,18 |
| -U.F. mult P.F. (4) | 400 | 18 |
| -U.F. mult P.F. (4) | 0 | 0 |
| -U.F. div P.F. (5) | 500 | 22,5 |
| Detenciones | 1002 | 45,09 |
| -Riesgos estructurales | 0 | 0 |
| -Riesgos de datos | 802 | 36,09 |
| -Riesgos de control | 200 | 9 |

Figura 5.11: Ventana de estadísticas en el camino de datos segmentado.

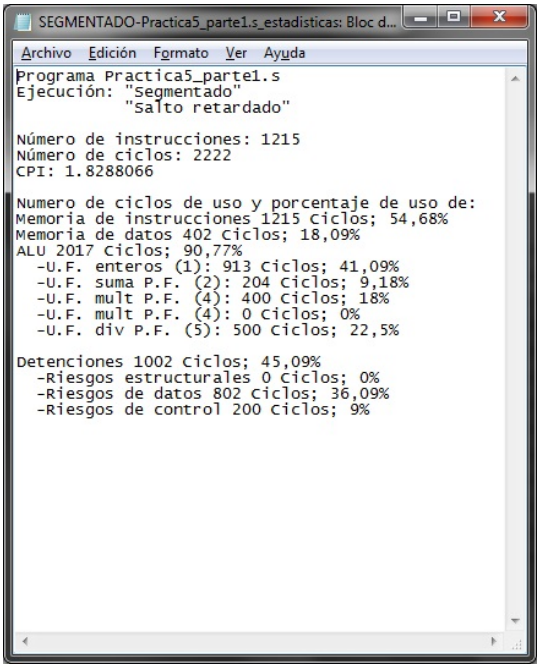



Figura 5.12: Archivo de texto de estadísticas en el camino de datos segmentado.

# CAPÍTULO 6. CONCLUSIONES Y TRABAJO FUTURO

A lo largo de todo este documento se ha detallado como se ha podido modificar el simulador MIPS Simula3MS para la inclusión de nuevas funcionalidades, en el caso de este trabajo la implementación de recogida y visualización de estadísticas de rendimiento. Desde las motivaciones que lo hicieron originarse, pasando por las diversas fases de su desarrollo, describiendo la metodología de trabajo, hasta la finalización e inclusión de ejemplo de uso. En este capítulo se comentan las contribuciones principales que ha supuesto este trabajo de fin de grado. También se describe el trabajo futuro que se puede realizar para seguir ampliando las funcionalidades del simulador y conseguir obtener una herramienta lo más completa posible.

## 6.1 CONTRIBUCIONES PRINCIPALES

En la sección 1.3 en el capítulo de introducción están descritos los objetivos que se establecieron para el desarrollo de este trabajo de fin de grado. Una vez cumplidos estos objetivos se ha obtenido un simulador mejorado. Esta herramienta servirá para su uso en la docencia en diversas asignaturas que tratan la arquitectura y organización del computador. Además a partir de esta nueva versión se puede seguir trabajando para que otros grupos de desarrollo añadan nuevas funcionalidades.

Como conclusión a esta sección, se ha de recalcar el esfuerzo empleado en asimilar toda la cantidad de documentación requerida para el proyecto, así como la comprensión del funcionamiento del Simula3MS y en especial, la capacidad de modificar su código para la creación y modificación de nuevas funcionalidades.

## 6.2 TRABAJO FUTURO

Como se ha comentado en la sección 3.1 en el capítulo de proceso de desarrollo, una vez estudiado el funcionamiento del simulador y se descubre lo que nos puede ofrecer la herramienta, podemos detectar una serie de funcionalidades de las que carece y que no se han resuelto con este proyecto:

- Permitir elegir la codificación de la visualización de los datos: hexadecimal, binario, decimal, etc.
- Permitir que se configure el cauce con adelantamiento o sin él.
- Permitir el uso de técnicas especulativas para la ejecución de instrucciones fuera de orden.
- Nuevas estrategias a la hora de resolver los saltos.

Todas estas funcionalidades, y otras que se pueden considerar interesantes de incluir, deberían ser implementadas por otros grupos de trabajo o ser presentados como proyectos con la finalidad de, recogidos todos en una misma aplicación, tener el simulador más completo con el que muchos centros docentes podrían beneficiarse para su uso en el estudio de la arquitectura MIPS y los diferentes caminos de datos.

# BIBLIOGRAFÍA


[1] Simula3ms. http://simula3ms.des.udc.es

[2] Francisco José Alfaro Cortés, Pedro Javier García García. Prácticas de Organización de Computadores, Manual de Laboratorio, 2010

[3] Erich Boehm. DLX Distribution Homepage. http://www.wu-wien.ac.at/usr/h93/h9301726/dlx.html, 1996.

[4] Grady Booch. Análisis y diseño orientado a objetos con aplicaciones.Addison-Wesley, 1996.

[5] Robert L. Britton. MIPS Assembly Languaje Programming. Prentice-Hall, 2004.

[6] Bruce Eckel. Piensa en Java. Prentice-Hall, 2000.

[7] C. Hamacher, Z. Vanesia, and S. Zaky. Organización de Computadores. McGraw-Hill, 2003.

[8] John L. Hennessy and David A. Patterson. Computer Architecure. A Cuantitative Approach. Morgan-Kaufmann, 2003.

[9] Cay S. Horstmann and Gary Cornell. Java 2. Características avanzadas. Prentice Hall, 2003.

[10] Cay S. Horstmann and Gary Cornell. Java 2. Fundamentos. Prentice Hall, 2003.

[11] James R. Larus. SPIM A MIPS R2000/R3000 Simulator. http://www.cs.wisc.edu/~larus/spim.html.

[12] David A. Patterson and John L. Hennessy. Estructura y diseño de computadores. Interficie circuitería/programación. Reverté, 2000.

[13] Fermín Sánchez. Características deseables en un procesador pedagógico parala enseñanza básica de la arquitectura de computadores. Jornadas de Enseñanza Universitaria de la Informática (JENUI), 2002.

[14] William Stallings. Organización y arquitectura de computadores. Prentice Hall, 2000.

[15] Miguel A. Vega and Juan A. Gómez Juan M. Sánchez. Innovación docente en la Arquitectura de Computadores mediante el uso de Simuladores. Jornadas de Enseñanza Universitaria de la Informática (JENUI), 2000.

[16] Simupro. http://www33.brinkster.com/vlaye/software/simuproc/simuproc.html.

[17] Eclipse http://www.eclipse.org/

# ANEXO A. **CÓDIGO FUENTE**

En este apartado se va a mostrar todo el código fuente utilizado para la modificación del nuevo simulador. En el capítulo 4: “Diseño e implementación de recogida de estadísticas” se explica con detalle la funcionalidad de los métodos más importantes de los que se muestra el código aquí. El código esta en el lenguaje de programación Java y utiliza una arquitectura modelo-vista-controlador. Primero se muestra la nueva clase *Estadisticas.java*, y a continuación se ven los cambios que se han realizado al código original para la inclusión de esta clase para los caminos de datos monociclo, multiciclo y segmentado.

## A.1 ESTADISTICAS.JAVA

```
/**Estadisticas.java
 *Clase que recoge y muestra estadísticas de rendimiento para los caminos de datos
 *monociclo, multiciclo y segmentado
 **/
public class Estadisticas extends Observable {
        private Lenguaje lenguaje;
        private int riesgosControl=0;
        private int riesgosEstructurales=0;
        private int riesgosRAW=0;
        private int riesgosWAW=0;
        private int riesgosWAR=0;
        private int numNOPs=0;
        private boolean ventana = false;
        private int numInstrucciones=0;
        private int integer=0;
        private int add=0;
        private int mult=0;
        private int div=0;
        private String nombre;
        private int latInteger=0;
        private int latAdd=0;
        private int latMult=0;
        private int latDiv=0;
        private int ciclos=0;
        private int memoriaInstruccion=0;
        private int memoriaDatos=0;
        private int ALU=0;
```

```
	private int salto=0;
	private JFrame frame = null;
	private JTextArea texto = null;
	private JTable table = null;
	private Vector<UFuncional> unidFuncionales;

	/**Constructor de Estadisticas
	 *@param Sin parametros
	 **/
	public Estadisticas() {
		inicializar();
		lenguaje = Lenguaje.getInstancia();
	}

	/**Inicia los valores
	 *@param Sin parametros
	 **/
	public void inicializar(){
		riesgosControl=0;
		riesgosEstructurales=0;
		riesgosRAW=0;
		riesgosWAW=0;
		riesgosWAR=0;
		numNOPs=0;
		numInstrucciones=0;
		integer=0;
		add=0;
		mult=0;
		div=0;
		latInteger=0;
		latAdd=0;
		latMult=0;
		latDiv=0;
		ciclos=0;
		memoriaInstruccion=0;
		memoriaDatos=0;
		ALU=0;
	}

	/**Cambia el valor de las unidades funcionales
	 *@param Vector<UFuncional> unidFuncionales
	 **/
	public void setUnidFuncionales(Vector<UFuncional> unidFuncionales) {
		this.unidFuncionales = unidFuncionales;
		for(UFuncional uf:unidFuncionales) {
			switch(uf.getTipo()){
				case 0: latInteger = uf.getLatencia();
				break;
				case 1: latAdd = uf.getLatencia();
				break;
				case 2: latMult = uf.getLatencia();
				break;
				case 3: latDiv = uf.getLatencia();
				break;
			}
		}
	}

	/**crea una cadena de texto con estadisticas basicas
	 *@param Sin parametros
	 *@return cadena de texto
	 **/
	public String toStringCPI(){
		String cpi=new String();
		cpi+="\n    "+lenguaje.getString("Numerodeinstrucciones")+" "+this.getNumInstrucciones();
		cpi+="\n    "+lenguaje.getString("Numerodeciclos")+" "+this.getCiclos();
		cpi+="\n    CPI: "+new Float(new Float(this.getCiclos())/new
Float(this.getNumInstrucciones())).toString();

		return cpi;
	}

	/**crea una cadena de texto con datos relevantes al monociclo
	 *@param Sin parametros
	 *@return cadena de texto
	 **/
	public String stringMonociclo(){
		String resultados=new String();
		if(nombre!="")
			resultados+="\n    "+lenguaje.getString("Programa")+" "+nombre;
		resultados+="\n    "+lenguaje.getString("EjecucionMonociclo");
		resultados+="\n\n    "+lenguaje.getString("Numerodeinstrucciones")+"
"+this.getNumInstrucciones();
		resultados+="\n    "+lenguaje.getString("Numerodeciclos")+" "+this.getCiclos();
```

```
                resultados+="\n    CPI: "+new Float(new Float(this.getCiclos())/new
Float(this.getNumInstrucciones())).toString();

                return resultados;
        }

        /**crea una cadena de texto que contiene un valor con 2 decimales
         *@param float val
         *@return cadena de texto
         **/
        private String parse(float val){
                DecimalFormat df = new DecimalFormat();
                df.setMaximumFractionDigits(2);
            return df.format(val).toString();
        }

        /**crea una cadena de texto con datos relevantes al multiciclo
         *@param Sin parametros
         *@return cadena de texto
         **/
        public String stringMulticiclo(){
                String resultados=new String();
                if(nombre!="")
                        resultados+="\n    "+lenguaje.getString("Programa")+" "+nombre;
                resultados+="\n    "+lenguaje.getString("EjecucionMulticiclo");
                resultados+="\n\n    "+lenguaje.getString("Numerodeinstrucciones")+"
"+this.getNumInstrucciones();
                resultados+="\n    "+lenguaje.getString("Numerodeciclos")+" "+this.getCiclos();
                resultados+="\n    CPI: "+new Float(new Float(this.getCiclos())/new
Float(this.getNumInstrucciones())).toString();

                return resultados;
        }

        /**crea una cadena de texto con datos relevantes al segmentado
         *@param Sin parametros
         *@return cadena de texto
         **/
        public String stringSegmentado(){
                String resultados=new String();
                if(nombre!="")
                        resultados+="\n    "+lenguaje.getString("Programa")+" "+nombre;
                resultados+="\n    "+lenguaje.getString("EjecucionSegmentado");
                if(salto==0)
                        resultados+="\n                    \""+lenguaje.getString("saltoRetardado")+"\"";
       else
        resultados+="\n                    \""+lenguaje.getString("saltoFijo")+"\"";
                resultados+="\n\n    "+lenguaje.getString("Numerodeinstrucciones")+"
"+this.getNumInstrucciones();
                resultados+="\n    "+lenguaje.getString("Numerodeciclos")+" "+this.getCiclos();
                resultados+="\n    CPI: "+new Float(new Float(this.getCiclos())/new
Float(this.getNumInstrucciones())).toString();
                return resultados;
        }

        /**crea el fichero de texto con las estadisticas del modo monociclo
         *@param Sin parametros
         **/
        public void escribirMono() {
       String fileName = lenguaje.getString("MONOCICLO")+"-"+nombre+"_estadisticas.txt";
       try {
           FileWriter fileWriter = new FileWriter(fileName);
           BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);
           if(nombre!=""){
                 bufferedWriter.write(lenguaje.getString("Programa")+" "+nombre);
                 bufferedWriter.newLine();
           }
           bufferedWriter.write(lenguaje.getString("EjecucionMonociclo"));
           bufferedWriter.newLine();
           bufferedWriter.newLine();
           bufferedWriter.write(lenguaje.getString("Numerodeinstrucciones")+" "+this.getNumInstrucciones());
           bufferedWriter.newLine();
           bufferedWriter.write(lenguaje.getString("Numerodeciclos")+" "+this.getCiclos());
           bufferedWriter.newLine();
           bufferedWriter.write("CPI: "+new Float(new Float(this.getCiclos())/new
Float(this.getNumInstrucciones())).toString());
           bufferedWriter.newLine();
           bufferedWriter.newLine();
           bufferedWriter.write(lenguaje.getString("NumeroCiclosUso"));
           bufferedWriter.newLine();
           bufferedWriter.write(lenguaje.getString("MemoriaInstrucciones")+" "+this.getMemoriaInstruccion()+"
"+lenguaje.getString("Ciclos")+"; "+parse(new Float(this.getMemoriaInstruccion())/new
Float(this.getCiclos())*100)+"%");
           bufferedWriter.newLine();
```

```
            bufferedWriter.write(lenguaje.getString("MemoriaDatos")+" "+this.getMemoriaDatos()+"
"+lenguaje.getString("Ciclos")+"; "+parse(new Float(this.getMemoriaDatos())/new Float(this.getCiclos())*100)+"%");
            bufferedWriter.newLine();
            bufferedWriter.write("ALU "+this.getALU()+" "+lenguaje.getString("Ciclos")+"; "+parse(new
Float(this.getALU())/new Float(this.getCiclos())*100)+"%");
            bufferedWriter.newLine();
            bufferedWriter.close();
        }
        catch(IOException ex) {
            System.out.println("Error writing to file '" + fileName + "'");
        }
    }

         /**crea el fichero de texto con las estadisticas del modo multiciclo
          *@param Sin parametros
          **/
         public void escribirMulti() {
        String fileName = lenguaje.getString("MULTICICLO")+"-"+nombre+"_estadisticas.txt";
        try {
            FileWriter fileWriter = new FileWriter(fileName);
            BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);
            if(nombre!=""){
                  bufferedWriter.write(lenguaje.getString("Programa")+" "+nombre);
                  bufferedWriter.newLine();
            }
            bufferedWriter.write(lenguaje.getString("EjecucionMulticiclo"));
            bufferedWriter.newLine();
            bufferedWriter.newLine();
            bufferedWriter.write(lenguaje.getString("Numerodeinstrucciones")+" "+this.getNumInstrucciones());
            bufferedWriter.newLine();
            bufferedWriter.write(lenguaje.getString("Numerodeciclos")+" "+this.getCiclos());
            bufferedWriter.newLine();
            bufferedWriter.write("CPI: "+new Float(new Float(this.getCiclos())/new
Float(this.getNumInstrucciones())).toString());
            bufferedWriter.newLine();
            bufferedWriter.newLine();
            bufferedWriter.write(lenguaje.getString("NumeroCiclosUso"));
            bufferedWriter.newLine();
            bufferedWriter.write(lenguaje.getString("MemoriaInstrucciones")+" "+this.getMemoriaInstruccion()+"
"+lenguaje.getString("Ciclos")+"; "+parse(new Float(this.getMemoriaInstruccion())/new
Float(this.getCiclos())*100)+"%");
            bufferedWriter.newLine();
            bufferedWriter.write(lenguaje.getString("MemoriaDatos")+" "+this.getMemoriaDatos()+"
"+lenguaje.getString("Ciclos")+"; "+parse(new Float(this.getMemoriaDatos())/new Float(this.getCiclos())*100)+"%");
            bufferedWriter.newLine();
            bufferedWriter.write("ALU "+this.getALU()+" "+lenguaje.getString("Ciclos")+"; "+parse(new
Float(this.getALU())/new Float(this.getCiclos())*100)+"%");
            bufferedWriter.newLine();
            bufferedWriter.write("  -"+lenguaje.getString("ALUEnteros")+" "+this.getInteger()+"
"+lenguaje.getString("Ciclos")+"; "+parse(new Float(this.getInteger())/new Float(this.getCiclos())*100)+"%");
            bufferedWriter.newLine();
            bufferedWriter.write("  -"+lenguaje.getString("ALUSumPF")+" "+this.getAdd()+"
"+lenguaje.getString("Ciclos")+"; "+parse(new Float(this.getAdd())/new Float(this.getCiclos())*100)+"%");
            bufferedWriter.newLine();
            bufferedWriter.write("  -"+lenguaje.getString("ALUMulPF")+" "+this.getMult()+"
"+lenguaje.getString("Ciclos")+"; "+parse(new Float(this.getMult())/new Float(this.getCiclos())*100)+"%");
            bufferedWriter.newLine();
            bufferedWriter.write("  -"+lenguaje.getString("ALUDivPF")+" "+this.getDiv()+"
"+lenguaje.getString("Ciclos")+"; "+parse(new Float(this.getDiv())/new Float(this.getCiclos())*100)+"%");
            bufferedWriter.newLine();
            bufferedWriter.close();
        }
        catch(IOException ex) {
            System.out.println("Error writing to file '" + fileName + "'");
        }
    }

         /**crea el fichero de texto con las estadisticas del modo segmentado
          *@param Sin parametros
          **/
         public void escribirSeg() {
        String fileName = lenguaje.getString("SEGMENTADO")+"-"+nombre+"_estadisticas.txt";
        try {
            FileWriter fileWriter = new FileWriter(fileName);
            BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);
            if(nombre!=""){
                  bufferedWriter.write(lenguaje.getString("Programa")+" "+nombre);
                  bufferedWriter.newLine();
            }
            bufferedWriter.write(lenguaje.getString("EjecucionSegmentado"));
            bufferedWriter.newLine();
            if(salto==0)
                  bufferedWriter.write("           \""+lenguaje.getString("saltoRetardado")+"\"");
            else
                  bufferedWriter.write("           \""+lenguaje.getString("saltoFijo")+"\"");
```

```
            bufferedWriter.newLine();
            bufferedWriter.newLine();
            bufferedWriter.write(lenguaje.getString("Numerodeinstrucciones")+" "+this.getNumInstrucciones());
            bufferedWriter.newLine();
            bufferedWriter.write(lenguaje.getString("Numerodeciclos")+" "+this.getCiclos());
            bufferedWriter.newLine();
            bufferedWriter.write("CPI: "+new Float(new Float(this.getCiclos())/new
Float(this.getNumInstrucciones())).toString());
            bufferedWriter.newLine();
            bufferedWriter.newLine();
            bufferedWriter.write(lenguaje.getString("NumeroCiclosUso"));
            bufferedWriter.newLine();
            bufferedWriter.write(lenguaje.getString("MemoriaInstrucciones")+" "+this.getMemoriaInstruccion()+"
"+lenguaje.getString("Ciclos")+"; "+parse(new Float(this.getMemoriaInstruccion())/new
Float(this.getCiclos())*100)+"%");
            bufferedWriter.newLine();
            bufferedWriter.write(lenguaje.getString("MemoriaDatos")+" "+this.getMemoriaDatos()+"
"+lenguaje.getString("Ciclos")+"; "+parse(new Float(this.getMemoriaDatos())/new Float(this.getCiclos())*100)+"%");
            bufferedWriter.newLine();
            bufferedWriter.write("ALU "+this.getALU()+" "+lenguaje.getString("Ciclos")+"; "+parse(new
Float(this.getALU())/new Float(this.getCiclos())*100)+"%");
            bufferedWriter.newLine();
            for(UFuncional uf:unidFuncionales) {
                  switch(uf.getTipo()){
                                    case 0: bufferedWriter.write("  -"+lenguaje.getString("ALUEnteros")+"
("+uf.getLatencia()+"): "+uf.getCiclosUso()+" "+lenguaje.getString("Ciclos")+"; "+parse((uf.getCiclosUso())/new
Float(this.getCiclos())*100)+"%");
                                    break;
                                    case 1: bufferedWriter.write("  -"+lenguaje.getString("ALUSumPF")+"
("+uf.getLatencia()+"): "+uf.getCiclosUso()+" "+lenguaje.getString("Ciclos")+"; "+parse((uf.getCiclosUso())/new
Float(this.getCiclos())*100)+"%");
                                    break;
                                    case 2: bufferedWriter.write("  -"+lenguaje.getString("ALUMulPF")+"
("+uf.getLatencia()+"): "+uf.getCiclosUso()+" "+lenguaje.getString("Ciclos")+"; "+parse((uf.getCiclosUso())/new
Float(this.getCiclos())*100)+"%");
                                    break;
                                    case 3: bufferedWriter.write("  -"+lenguaje.getString("ALUDivPF")+"
("+uf.getLatencia()+"): "+uf.getCiclosUso()+" "+lenguaje.getString("Ciclos")+"; "+parse((uf.getCiclosUso())/new
Float(this.getCiclos())*100)+"%");
                                    break;
                  }
                bufferedWriter.newLine();
                              }
            bufferedWriter.newLine();
            bufferedWriter.write(lenguaje.getString("Detenciones")+" "+this.getNumNOPs()+"
"+lenguaje.getString("Ciclos")+"; "+parse(new Float(this.getNumNOPs())/new Float(this.getCiclos())*100)+"%");
            bufferedWriter.newLine();
            bufferedWriter.write("  -"+lenguaje.getString("RiesgosEstructurales")+"
"+this.getRiesgosEstructurales()+" "+lenguaje.getString("Ciclos")+"; "+parse(new
Float(this.getRiesgosEstructurales())/new Float(this.getCiclos())*100)+"%");
            bufferedWriter.newLine();
            bufferedWriter.write("  -"+lenguaje.getString("RiesgosDatos")+" "+this.getRiesgosDatos()+"
"+lenguaje.getString("Ciclos")+"; "+parse(new Float(this.getRiesgosDatos())/new Float(this.getCiclos())*100)+"%");
            bufferedWriter.newLine();
            bufferedWriter.write("  -"+lenguaje.getString("RiesgosControl")+" "+this.getRiesgosControl()+"
"+lenguaje.getString("Ciclos")+"; "+parse(new Float(this.getRiesgosControl())/new
Float(this.getCiclos())*100)+"%");
            bufferedWriter.newLine();
            bufferedWriter.close();
        }
        catch(IOException ex) {
            System.out.println("Error writing to file '" + fileName + "'");
        }
    }

         /**crea la ventana de estadisticas
          *@param int modo
          **/
         public void ventana(int modo){
                  frame = new JFrame(lenguaje.getString("Estadisticas"));
                  frame.setSize(300, 400);
                  frame.setResizable(false);
                  frame.setLocation(1074, 250);
                  frame.setBackground(new java.awt.Color(238, 238, 238));
                  texto = new JTextArea();
                  texto.setEditable(false);
                  texto.setBackground(new java.awt.Color(238, 238, 238));
                  TableColumn column = null;
                  switch (modo){
                  case 1:
                           Object rowMono[][] = { { "", lenguaje.getString("Ciclos"), "%" },
                                                  { "    "+lenguaje.getString("MemoriaInstrucciones"),
getMemoriaInstruccion(), parse(new Float(this.getMemoriaInstruccion())/new Float(this.getCiclos())*100) },
                                                { "    "+lenguaje.getString("MemoriaDatos"), getMemoriaDatos(), parse(new
Float(this.getMemoriaDatos())/new Float(this.getCiclos())*100) },
```

```
                                { "    ALU", getALU(), parse(new Float(this.getALU())/new
Float(this.getCiclos())*100) } };
                        Object columnMono[] = { "", lenguaje.getString("Ciclos"), "%" };
                        table = new JTable(rowMono, columnMono);
                        break;
                case 2:
                        Object rowMulti[][] = { { "", lenguaje.getString("Ciclos"), "%" },
                                        { "    "+lenguaje.getString("MemoriaInstrucciones"),
getMemoriaInstruccion(), parse(new Float(this.getMemoriaInstruccion())/new Float(this.getCiclos())*100) },
                                { "    "+lenguaje.getString("MemoriaDatos"), getMemoriaDatos(), parse(new
Float(this.getMemoriaDatos())/new Float(this.getCiclos())*100) },
                                { "    ALU", getALU(), parse(new Float(this.getALU())/new
Float(this.getCiclos())*100) },
                                { "       -"+lenguaje.getString("ALUEnteros")+"       (1)", getInteger(),
parse(new Float(this.getInteger())/new Float(this.getCiclos())*100) },
                                { "       -"+lenguaje.getString("ALUSumPF")+"  ("+getLatAdd()+")", getAdd(),
parse(new Float(this.getAdd())/new Float(this.getCiclos())*100) },
                                { "       -"+lenguaje.getString("ALUMulPF")+"     ("+getLatMult()+")",
getMult(), parse(new Float(this.getMult())/new Float(this.getCiclos())*100) },
                                { "       -"+lenguaje.getString("ALUDivPF")+"        ("+getLatDiv()+")",
getDiv(), parse(new Float(this.getDiv())/new Float(this.getCiclos())*100) } };
                        Object columnMulti[] = { "", lenguaje.getString("Ciclos"), "%" };
                        table = new JTable(rowMulti, columnMulti);
                        break;
                case 3:
                        int row=unidFuncionales.size();
                        Object rowSeg[][] = new Object[row+8][3];
                        Object columnSeg[] = { "", lenguaje.getString("Ciclos"), "%" };

                        table = new JTable(rowSeg, columnSeg);
                        table.setValueAt("", 0, 0);
                        table.setValueAt(lenguaje.getString("Ciclos"), 0, 1);
                        table.setValueAt("%", 0, 2);
                        table.setValueAt("    "+lenguaje.getString("MemoriaInstrucciones"), 1, 0);
                        table.setValueAt("    "+lenguaje.getString("MemoriaDatos"), 2, 0);
                        table.setValueAt("    ALU", 3, 0);
                        int x=4;
                        for(UFuncional uf:unidFuncionales) {
                        switch(uf.getTipo()){
                                case 0: table.setValueAt("       -"+lenguaje.getString("ALUEnteros")+"
("+uf.getLatencia()+")", x, 0);
                                break;
                                case 1: table.setValueAt("       -"+lenguaje.getString("ALUSumPF")+"
("+uf.getLatencia()+")", x, 0);
                                break;
                                case 2: table.setValueAt("       -"+lenguaje.getString("ALUMulPF")+"
("+uf.getLatencia()+")", x, 0);
                                break;
                                case 3: table.setValueAt("       -"+lenguaje.getString("ALUDivPF")+"
("+uf.getLatencia()+")", x, 0);
                                break;
                                }
                                x++;
                        }
                        table.setValueAt("    "+lenguaje.getString("Detenciones"), x, 0);
                        table.setValueAt("       -"+lenguaje.getString("RiesgosEstructurales"), x+1, 0);
                        table.setValueAt("       -"+lenguaje.getString("RiesgosDatos"), x+2, 0);
                        table.setValueAt("       -"+lenguaje.getString("RiesgosControl"), x+3, 0);
                        break;
                default: texto.setText(toStringCPI());
                        break;
                }
                for (int i = 0; i < 3; i++) {
                    column = table.getColumnModel().getColumn(i);
                    if (i == 0) {
                        column.setPreferredWidth(180);
                    } else {
                        column.setPreferredWidth(50);
                    }
                }
                table.setBackground(new java.awt.Color(238, 238, 238));
                frame.getContentPane().add( texto, BorderLayout.NORTH );
                frame.getContentPane().add( table, BorderLayout.CENTER );
            frame.setVisible(true);
            ventana=true;
        }

        /**actualiza los datos en la ventana de estadisticas
         *@param int modo
         **/
        public void actualizar(int modo){
        switch (modo){
        case 1: texto.setText(stringMonociclo());
        table.setValueAt(getMemoriaInstruccion(), 1, 1);
        table.setValueAt(parse(new Float(this.getMemoriaInstruccion())/new Float(this.getCiclos())*100), 1, 2);
```

```
table.setValueAt(getMemoriaDatos(), 2, 1);
table.setValueAt(parse(new Float(this.getMemoriaDatos())/new Float(this.getCiclos())*100), 2, 2);
table.setValueAt(getALU(), 3, 1);
table.setValueAt(parse(new Float(this.getALU())/new Float(this.getCiclos())*100), 3, 2);
break;
case 2: texto.setText(stringMulticiclo());
table.setValueAt(getMemoriaInstruccion(), 1, 1);
table.setValueAt(parse(new Float(this.getMemoriaInstruccion())/new Float(this.getCiclos())*100), 1, 2);
table.setValueAt(getMemoriaDatos(), 2, 1);
table.setValueAt(parse(new Float(this.getMemoriaDatos())/new Float(this.getCiclos())*100), 2, 2);
table.setValueAt(getALU(), 3, 1);
table.setValueAt(parse(new Float(this.getALU())/new Float(this.getCiclos())*100), 3, 2);
table.setValueAt(getInteger(), 4, 1);
table.setValueAt(parse(new Float(this.getInteger())/new Float(this.getCiclos())*100), 4, 2);
table.setValueAt(getAdd(), 5, 1);
table.setValueAt(parse(new Float(this.getAdd())/new Float(this.getCiclos())*100), 5, 2);
table.setValueAt(getMult(), 6, 1);
table.setValueAt(parse(new Float(this.getAdd())/new Float(this.getCiclos())*100), 6, 2);
table.setValueAt(getDiv(), 7, 1);
table.setValueAt(parse(new Float(this.getDiv())/new Float(this.getCiclos())*100), 7, 2);
break;
case 3: texto.setText(stringSegmentado());
table.setValueAt(getMemoriaInstruccion(), 1, 1);
table.setValueAt(parse(new Float(this.getMemoriaInstruccion())/new Float(this.getCiclos())*100), 1, 2);
table.setValueAt(getMemoriaDatos(), 2, 1);
table.setValueAt(parse(new Float(this.getMemoriaDatos())/new Float(this.getCiclos())*100), 2, 2);
table.setValueAt(getALU(), 3, 1);
table.setValueAt(parse(new Float(this.getALU())/new Float(this.getCiclos())*100), 3, 2);
int x=4;
for(UFuncional uf:unidFuncionales) {
table.setValueAt(uf.getCiclosUso(), x, 1);
table.setValueAt(parse(new Float(uf.getCiclosUso())/new Float(this.getCiclos())*100), x, 2);
x++;
}
table.setValueAt(getNumNOPs(), x, 1);
table.setValueAt(parse(new Float(this.getNumNOPs())/new Float(this.getCiclos())*100), x, 2);
table.setValueAt(getRiesgosEstructurales(), x+1, 1);
table.setValueAt(parse(newFloat(this.getRiesgosEstructurales())/newFloat(this.getCiclos())*100),x+1, 2);
table.setValueAt(getRiesgosDatos(), x+2, 1);
table.setValueAt(parse(new Float(this.getRiesgosDatos())/new Float(this.getCiclos())*100), x+2, 2);
table.setValueAt(getRiesgosControl(), x+3, 1);
table.setValueAt(parse(new Float(this.getRiesgosControl())/new Float(this.getCiclos())*100), x+3, 2);
break;
default: texto.setText(toStringCPI());
break;
}
}

/**aumenta o retrocede las varibales referentes al monociclo
 *@param int estilo, int estado
 **/
public void monociclo(int estilo, int estado){
        if(estado==1){
                aumentarCiclos();
                aumentarNumInstrucciones();
                aumentarMemoriaInstruccion();
                switch (estilo){
                        case 0:/*rfe (vuelta desde una excepcion)*/
                                break;
                        case 1:/*desconocido*/
                                break;
                        case 2:/*tipo R*/aumentarALU();
                                break;
                        case 3:/*inmediatas*/aumentarALU();
                                break;
                        case 4:/*carga*/aumentarALU();
                                aumentarMemoriaDatos();
                                break;
                        case 5:/*almacenamiento*/aumentarALU();
                                aumentarMemoriaDatos();
                                break;
                        case 6:/*salto condicional*/aumentarALU();
                                break;
                        case 7:/*tipo J*/
                                break;
                        case 8:/*nop*/aumentarNumNOPs();
                                break;
                        case 9:/*Syscall*/
                                break;
                        case 10:/*suma PF*/
                                break;
                        case 11:/*multiplicacion PF*/
                                break;
                        case 12:/*division PF*/
                                break;
```

```
                        default:
                                break;
                }
        }else{
                disminuirCiclos();
                disminuirNumInstrucciones();
                disminuirMemoriaInstruccion();
                switch (estilo){
                        case 0:
                                break;
                        case 1:
                                break;
                        case 2:  disminuirALU();
                                break;
                        case 3:  disminuirALU();
                                break;
                        case 4:  disminuirALU();
                                disminuirMemoriaDatos();
                                break;
                        case 5:  disminuirALU();
                                disminuirMemoriaDatos();
                                break;
                        case 6:  disminuirALU();
                                break;
                        case 7:
                                break;
                        case 8: disminuirNumNOPs();
                                break;
                        case 9:
                                break;
                        default:
                                break;
                }
        }
}

/**aumenta o retrocede las varibales referentes al multiciclo
 *@param int estilo, int numero, int estado, int etapa
 **/
public void multiciclo(int estilo, int numero, int estado, int etapa){
        if(estado==1){
                switch(etapa){
                        case 0:if(numero==10||numero==11){
                                aumentarInteger();
                                aumentarALU();
                                }
                                break;
                        case 1:
                                break;
                        case 2: aumentarMemoriaInstruccion();
                                        aumentarNumInstrucciones();
                                break;
                        case 3:
                                break;
                        case 4: if(numero==25)return;
                                aumentarALU();
                                if(numero<25||numero==56||numero==57)
                                        aumentarInteger();
                                else{
                                        if(numero==28||numero==32)
                                                aumentarMult();
                                        else{
                                                if(numero==29||numero==33)
                                                        aumentarDiv();
                                                else
                                                        aumentarAdd();
                                        }
                                }
                                break;
                        case 5: if(estilo==4||estilo==5) aumentarMemoriaDatos();
                                break;
                }
        }else{
                if(numero==25)
                        return;
                switch(etapa){
                        case 0:if(numero==10||numero==11){
                                disminuirInteger();
                                disminuirALU();
                                }
                                break;
                        case 1:
                                break;
                        case 2: //disminuirMemoriaInstruccion();
                                        //disminuirNumInstrucciones();
```

```
                                        break;
                                case 3:
                                        break;
                                case 4: disminuirALU();
                                                if(numero<25||numero==56||numero==57)
                                                        disminuirInteger();
                                                else{
                                                        if(numero==28||numero==32)
                                                                disminuirMult();
                                                        else{
                                                                if(numero==29||numero==33)
                                                                        disminuirDiv();
                                                                else
                                                                        disminuirAdd();
                                                        }
                                                }
                                        break;
                                case 5: if(estilo==4||estilo==5) disminuirMemoriaDatos();
                                        break;
                        }
                }
        }

        /**aumenta o retrocede las variables referentes al segmentado
         *@param PipelineFlotante pipeline
         **/
        public void segmentado(PipelineFlotante pipeline){
                int estilo0;
                int estilo1;
                int numero;
                EstadoInstruccion[] ei;
                aumentarCiclos();
                //MEM
                estilo0 = pipeline.getEtapaMEM().getEstadoInstrucciones()[0].getInstruccion().estilo();
                estilo1 = pipeline.getEtapaMEM().getEstadoInstrucciones()[1].getInstruccion().estilo();
                if((estilo0!=8)||(estilo1!=8)){
                        if((estilo0==4)||(estilo0==5)||(estilo1==4)||(estilo1==5)){
                                aumentarMemoriaDatos();
                        }
                }
                //EX
                for(UFuncional uf:pipeline.getEtapaEX().getUnidFuncionales()) {
                        ei = uf.getInstrucciones();
                        for(EstadoInstruccion e:ei) {
                                if(e.getInstruccion().estilo()!=8) {
                                        aumentarALU();
                                        numero = e.getInstruccion().numero();
                                        if(numero<25||numero==56||numero==57)
                                                aumentarInteger();
                                        else{
                                                if(numero==28||numero==32)
                                                        aumentarMult();
                                                else{
                                                        if(numero==29||numero==33)
                                                                aumentarDiv();
                                                        else
                                                                aumentarAdd();
                                                }
                                        }
                                }
                        }
                }
        }
```

# A.2 MODIFICACIONES MONOCICLO

```
/**VistaSegmentado.java
 *Clase encargada de visualizar el procesador monociclo
 **/
public class VistaMonociclo extends Vista implements Observer, Tipos {
```

```
    Instruccion ins;
    int i=0;
    int contador;
    //cambio
    private Estadisticas stats;
    //
    Registro registro;
    int num_ciclos;
    private String la_memoria;
    private String la_pila;
    protected Vector los_datos;
    LimitedStyledDocument lpd;
    static int MAX_CHARACTERS;
    Hashtable actions;
    String newline;
    Palabra p;
    boolean seguir;
    ObservaRegistros ObReg=new ObservaRegistros();

    /**
     *Constructor de la clase monociclo
     **/
    public VistaMonociclo(){}
    public VistaMonociclo(Ensamblador edit) {
         super(edit);
         //cambios
        stats = new Estadisticas(editor.getNombreF());
        //
        seguir=true;
         controlador=new ControladorMonociclo();
         breakpoint=new Palabra(INICIO_PC.getDec()+controlador.getPc().length*4);

         initComponents();
        inicializando();

        setTitle(lenguaje.getString("MONOCICLO")+"  "+editor.getNombreF());

        this.setSize(1124, 740);
        controlador.anhadirObservador(this);

        setMem(obtDatos(controlador.visualizarMemoria()));
        setPil(obtDatos(controlador.visualizarPila()));
        visualizarDatos();
        MAX_CHARACTERS=controlador.tamanhoCodigo()*1000;
        this.subrayar("0");

        $sp.setText("$sp="+Sp.getRegistro().getPalabra().getHex().substring(2,10));
        breakpoint=new Palabra(INICIO_PC.getDec()+controlador.getPc().length*4);
        los_datos=new Vector();
  }


/**
     *Metodo que hace retroceder la ejecucion hasta el ciclo anterior
     *@param ItemEvent evt
     *@return void
     **/
    protected void botonCicloAntActionPerformed(java.awt.event.ActionEvent evt) {
          botonCicloSig.setEnabled(true);
         botonEjecutar.setEnabled(true);
         if(!seguir)
          seguir=true;
         try{
             if(controlador.ejecutarCicloAnterior(breakpoint)){
                   this.actualizarCiclos(((ControladorMonociclo)controlador).getCiclos());
                    if(((ControladorMonociclo)controlador).getCiclos()==0){
                      botonCicloAnt.setEnabled(false);
                    }
                  p=controlador.getPalabraPC();
                  subrayar(p.sumar(-4).getHex());
                  caminoDatos.setIcon(new
javax.swing.ImageIcon(getClass().getResource(((ControladorMonociclo)controlador).getImagenActual().toLowerCase()))
);

                //cambios
                  int estilo=controlador.getInstruccionActual().estilo();
                  stats.monociclo(estilo,0);
                  stats.actualizar(1);
                  //
             }
         }catch(EjecucionException e){
          this.analizaExcep(e.getTipo(), e);
         }
```

```
    }

    /**
     *Metodo que completa la ejecucion del codigo
     *@param ActionEvent evt
     *@return void
     * @throws IOException
     **/
    protected void botonEjecutarActionPerformed(java.awt.event.ActionEvent evt){
         botonCicloAnt.setEnabled(true);
        while(seguir){
         int estilo=controlador.getInstruccionActual().estilo();
                try{
                          p=controlador.getPalabraPC();
                         if(controlador.ejecutarCiclo(breakpoint)){
                          this.actualizarCiclos(((ControladorMonociclo)controlador).getCiclos());
                                  //cambios
                                 stats.monociclo(estilo,1);
                                 //
                         }
                }catch(EjecucionException e){
                 this.analizaExcep(e.getTipo(), e);
                 seguir=false;
                 //cambios
                    stats.monociclo(estilo,1);
                    stats.actualizar(1);
                    stats.escribirMono();
                    //
                }
        }
    }

     /**
     *Metodo que avanza la ejecucin hacia el ciclo siguiente
     *@param ActionEvent evt
     *@return void
     **/
    protected void botonCicloSigActionPerformed(java.awt.event.ActionEvent evt) {
          botonCicloAnt.setEnabled(true);
          int estilo=controlador.getInstruccionActual().estilo();
         try{
          p=controlador.getPalabraPC();
                 if(controlador.ejecutarCiclo(breakpoint)){
                  this.actualizarCiclos(((ControladorMonociclo)controlador).getCiclos());
                  subrayar(p.getHex());
                     caminoDatos.setIcon(new
javax.swing.ImageIcon(getClass().getResource(((ControladorMonociclo)controlador).getImagenActual().toLowerCase()))
);
                     //cambios
                     stats.monociclo(estilo,1);
                     stats.actualizar(1);
                     //
                 }
         }catch(EjecucionException e){
         this.analizaExcep(e.getTipo(), e);
         //cambios
         stats.monociclo(estilo,1);
         stats.actualizar(1);
         //
         }
    }
     /**
     *Metodo que vuelve a la ventana del editor
     *@param ActionEvent evt
     *@return void
     **/
    protected void botonVolverEditActionPerformed(java.awt.event.ActionEvent evt) {
         // cambios
        if(stats.getVentana()){
                  stats.getFrame().setVisible(false);
                  stats.setVentana(false);
        }
         //
         this.dispose();
        editor.setVisible(true);
    }

    /**
     *Metodo que abre la ventana de estadisticas
     *@param ActionEvent evt
     *@return void
     **/
    protected void botonEstadisticasActionPerformed(java.awt.event.ActionEvent evt) {
         if(stats.getVentana()==false){
                  stats.ventana(1);
```

```
                stats.actualizar(1);
        }else{
         stats.getFrame().setVisible(false);
                stats.setVentana(false);
        }
    }

protected void initSegTexto(){
         jPanel26.setLayout(new javax.swing.BoxLayout(jPanel26, javax.swing.BoxLayout.X_AXIS));

        jPanel26.setBorder(new javax.swing.border.TitledBorder(null, lenguaje.getString("segmentoDeTexto"),
javax.swing.border.TitledBorder.LEFT, javax.swing.border.TitledBorder.TOP));
        jPanel26.setMaximumSize(new java.awt.Dimension(1180, 32767));
        jPanel26.setMinimumSize(new java.awt.Dimension(150, 37));
        jPanel26.setPreferredSize(new java.awt.Dimension(650, 35));
        jScrollPane2.setBorder(null);
        jScrollPane2.setMaximumSize(new java.awt.Dimension(600, 32767));
        jScrollPane2.setMinimumSize(new java.awt.Dimension(0, 22));
        jScrollPane2.setPreferredSize(new java.awt.Dimension(0, 15));
        jScrollPane2.setAutoscrolls(true);
        segTexto.setBackground(new java.awt.Color(238, 238, 238));
        segTexto.setEditable(false);
        segTexto.setMinimumSize(new java.awt.Dimension(650, 15));
        segTexto.setPreferredSize(new java.awt.Dimension(0, 15));
        jScrollPane2.setViewportView(segTexto);

        jPanel26.add(jScrollPane2);

         // cambios
        botonEstadisticas.setFont(new java.awt.Font("Dialog", 1, 11));
        botonEstadisticas.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/ensamblador/iconos/estadisticas.png")));
        botonEstadisticas.setText("Estadísticas");
        botonEstadisticas.setHorizontalTextPosition(javax.swing.SwingConstants.CENTER);
        botonEstadisticas.setMaximumSize(new java.awt.Dimension(130, 70));
        botonEstadisticas.setMinimumSize(new java.awt.Dimension(130, 70));
        botonEstadisticas.setPreferredSize(new java.awt.Dimension(130, 70));
        botonEstadisticas.setVerticalTextPosition(javax.swing.SwingConstants.BOTTOM);
        botonEstadisticas.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                botonEstadisticasActionPerformed(evt);
            }
        });
        jPanel26.add(botonEstadisticas);
        jPanel24.add(jPanel26);
         //
         }
```

# A.3 MODIFICACIONES MULTICICLO

```
/**
*Multiciclo.java
*Clase encargada de visualizar el procesador multiciclo
**/
public class Multiciclo extends javax.swing.JFrame implements Observer, Tipos {
    //cambio
    private Estadisticas stats;
    //
   Lenguaje lenguaje;
    private Palabra breakpoint;
    private Ensamblador editor;
    Codigo instrucc=Codigo.getInstacia();
    Pc regPC=Pc.getRegistro();
    Instruccion ins;
    int numero;
    int i=0;
    int etapa=0;
    String codigo;
    int contador;
    int num1,num2,num3;
    Memoria memoria=Memoria.getMemoria();
    Pila pila=Pila.getPila();
    Registro registro;
    int num_ciclos;
```

```
int num_instrucciones;
private String la_memoria;
private String la_pila;
private Vector los_datos;
LimitedStyledDocument lpd;
static int MAX_CHARACTERS;
Hashtable actions;
String newline;
ObservaRegistros ObReg=new ObservaRegistros();

 /**
 *Constructor de la clase multiciclo
 **/
public Multiciclo(Ensamblador edit,int entero1,int entero2,int entero3) {

     lenguaje=Lenguaje.getInstancia();
     initComponents();
    inicializando();
    editor=edit;

    //cambios
    stats = new Estadisticas(editor.getNombreF());
    num_instrucciones=0;
    num1=entero1;
    num2=entero2;
    num3=entero3;
    stats.setLatAdd(num1);
    stats.setLatMult(num2);
    stats.setLatDiv(num3);
    //
    setTitle(lenguaje.getString("MULTICICLO")+"  "+editor.getNombreF());
    codigo=editor.Editor.getText();
    this.setSize(1124, 740);
    memoria.addObserver(this);
    pila.addObserver(this);
    String m=obtDatos(memoria.visualizar());
    setMem(m);
    String p=obtDatos(pila.visualizar());
    setPil(p);
    visualizarDatos();
    MAX_CHARACTERS=instrucc.tamanho()*100;
    this.subrayar("0");
    segTexto.setText(instrucc.visualizar());
    segTexto.requestFocus();
    segTexto.select(0,0);
    ObReg.addObserver(this);
    $sp.setText("$sp="+Sp.getRegistro().getPalabra().getHex().substring(2,10));
    breakpoint=new Palabra(INICIO_PC.getDec()+instrucc.getPc().length*4);
    los_datos=new Vector();
}
 /**
 *Metodo que hace retroceder la ejecucion hasta el paso anterior
 *@param ItemEvent evt
 *@return void
 **/
private void botonPasoAntActionPerformed(java.awt.event.ActionEvent evt) {
    int excep=-1;
    int nuevo_cont;
    inicializando();
    if(contador>0)
    {
      //cambios
     int estilo=ins.estilo();
     int num=ins.numero();
     int etapa_act=ins.getEtapa();
      if(etapa_act>=4){
                for(int i=etapa_act;i>=0;i--)
                {
                      stats.multiciclo(estilo,num,0,i);
                }
       }
      else{
               if(etapa_act==0){
                      for(int i=5;i>=0;i--)
                {
                      stats.multiciclo(estilo,num,0,i);
                }
               }
      }
     //
       inicializando();
       numero=analizaInstruc(ins.numero());
       if(ins.getEtapa()==0)
           contador=contador-ins.numEtapas()-numero+1;
       else if(ins.getEtapa()==4)
```

```
            contador=contador-ins.getEtapa()-numero+1;
        else
            contador-=ins.getEtapa();
        nuevo_cont=obtienePaso();
        contador=nuevo_cont;

        ins=instrucc.obtener((int)regPC.getPalabra().getDec());
        numero=analizaInstruc(ins.numero());
        subrayar(regPC.getPalabra().getHex());
        excep=ejecutaEtapa(1,ins);
        ins.setEtapa(2);

        contador++;

        String imagen=ins.visualizarEtapa();
        imagen=imagen+".gif";
        caminoDatos.setIcon(new javax.swing.ImageIcon(getClass().getResource(imagen.toLowerCase())));

        num_ciclos=contador;
        ciclos.setText(new Integer(num_ciclos).toString());
        botonesSig(true);

        //cambios
        estilo=ins.estilo();
        num=ins.numero();
        for(int i=ins.numEtapas();i>=3;i--)
        {
              stats.multiciclo(estilo,num,0,i);
        }
        stats.setCiclos(num_ciclos);
        if(stats.getNumInstrucciones()!=1){
              stats.disminuirNumInstrucciones();
              stats.disminuirMemoriaInstruccion();
        }
        stats.actualizar(2);
        //
    }
    if(contador==0)
        botonesAnt(false);
    if(contador<ins.numEtapas())
        botonPasoAnt.setEnabled(false);
}

 /**
 *Metodo que hace avanzar la ejecucion hasta el paso siguiente
 *@param ItemEvent evt
 *@return void
 **/

private void botonPasoSigActionPerformed(java.awt.event.ActionEvent evt) {
    long posPc=regPC.getPalabra().getDec();
    int excep=-1;
    int error=0;
    int estilo=0;
    int num=0;
    int precontador=contador;
    if(seguir(posPc, ins))
    {
        if(ins==null || (etapa=ins.getEtapa())==0)
        {
            if(ins!=null)
            {
                ins.cambiosVisibles();
                ins.setEtapa(1);
            }
             ins=instrucc.obtener((int)regPC.getPalabra().getDec());
             subrayar(regPC.getPalabra().getHex());
             ins.setEtapa(1);

        }
        estilo=ins.estilo();
        num=ins.numero();
        int etapa_act=ins.getEtapa();
        if(etapa_act>=0/*1*/)
        {
            numero=analizaInstruc(ins.numero());
            if(etapa_act!=4)
                contador=contador+numero-1;
            for(int i=etapa_act;i<=ins.numEtapas();i++)
            {
                excep=ejecutaEtapa(i,ins);       /*La excepcion si ocurre es en la ultima etapa*/
                ins.setEtapa(i+1);
                contador++;
                //cambios
              stats.aumentarCiclos();
```

```
                stats.multiciclo(estilo,num,1,ins.getEtapa());
                //
            }
        }
        error=analizaExcep(excep);
        if(error==javax.swing.JOptionPane.ERROR_MESSAGE)
            botonesSig(false);
             String imagen=ins.visualizarEtapa();
             imagen=imagen+".gif";
             caminoDatos.setIcon(new javax.swing.ImageIcon(getClass().getResource(imagen.toLowerCase())));
             num_ciclos=contador;
             ciclos.setFont(new java.awt.Font("Dialog", 1, 36));
             ciclos.setText(new Integer(num_ciclos).toString());
             //cambios
             stats.actualizar(2);
    }    /**
 *Metodo que hace retroceder la ejecucion hasta el ciclo anterior
 *@param ItemEvent evt
 *@return void
 **/

private void botonCicloAntActionPerformed(java.awt.event.ActionEvent evt) {
    botonCicloSig.setEnabled(true);
    int excep=-1;
    int nuevo_cont;
    int num=0;
    int estilo=0;
    int etapa_act=0;
    Palabra contpr=Pc.getRegistro().getPalabra();
    if(contador>0)
    {
        inicializando();
        //cambios
        etapa_act=ins.getEtapa();
        //
        numero=analizaInstruc(ins.numero());
        contador--;
        if(ins.getEtapa()==4)
            contador=contador-numero+1;
        nuevo_cont=obtienePaso();
        etapa=1;
        while(nuevo_cont<contador)
        {
            numero=analizaInstruc(ins.numero());
            if(etapa==3)
                nuevo_cont=nuevo_cont+numero-1;
            excep=ejecutaEtapa(etapa, ins);
            etapa++;
            ins.setEtapa(etapa);
            nuevo_cont++;
        }
        ins.cambiosVisibles();
        String imagen=ins.visualizarEtapa();
        Palabra p=Pc.getRegistro().getPalabra();
        //Aqui se hace diferente cuando se trata de un salto condicional
        subrayar(p.sumar(-4).getHex());
        imagen=imagen+".gif";
        caminoDatos.setIcon(new javax.swing.ImageIcon(getClass().getResource(imagen.toLowerCase())));
        num_ciclos=nuevo_cont;
        ciclos.setText(new Integer(num_ciclos).toString());
        botonesSig(true);
        //cambios
        if(ins.getEtapa()==0||contador==0){
              stats.disminuirNumInstrucciones();
              stats.disminuirMemoriaInstruccion();
        }
        stats.disminuirCiclos();
        estilo=ins.estilo();
     num=ins.numero();
     stats.multiciclo(estilo,num,0,etapa_act);
        stats.actualizar(2);
        //
    }
    if(contador==0)
    {
        botonCicloAnt.setEnabled(false);
        botonPasoAnt.setEnabled(false);
    }
    if(stats.getNumInstrucciones()<2){
     botonPasoAnt.setEnabled(false);
    }
}

    /**
 *Metodo que completa la ejecucion del codigo
```

```
     *@param ActionEvent evt
     *@return void
     **/
    private void botonEjecutarActionPerformed(java.awt.event.ActionEvent evt) {
        long posPc=regPC.getPalabra().getDec();
        int excep=-1;
        int estilo=0;
        int num=0;
        while(seguir(regPC.getPalabra().getDec(), ins))
        {
         if(ins!=null){
                  numero=analizaInstruc(ins.numero());
                  if(ins.getEtapa()==3) //estamos en la etapa de EX --> EX=3
                           contador=contador+numero;
                  else
                           contador++;
         }else{
                  contador++;
         }
            if(ins==null || (etapa=ins.getEtapa())==0)
            {
                if(ins!=null)
                {
                    ins.cambiosVisibles();
                }
                ins=instrucc.obtener((int)regPC.getPalabra().getDec());
                subrayar(regPC.getPalabra().getHex());
                ins.setEtapa(etapa++);
                if(contador!=0)
                    botonesAnt(true);
            }
            if(etapa<=ins.numEtapas())
            {
                  //cambios
                  estilo=ins.estilo();
                  num=ins.numero();
                  stats.aumentarCiclos();
                  stats.multiciclo(estilo,num,1,ins.getEtapa());
                  //
                excep=ejecutaEtapa(etapa, ins);
                etapa++;
                ins.setEtapa(etapa);
            }
           int error=analizaExcep(excep);
            if(error==javax.swing.JOptionPane.ERROR_MESSAGE)
                break;
        }
        String imagen=ins.visualizarEtapa();
        imagen=imagen+".gif";
        caminoDatos.setIcon(new javax.swing.ImageIcon(getClass().getResource(imagen.toLowerCase())));
        num_ciclos=contador;
        ciclos.setFont(new java.awt.Font("Dialog", 1, 36));
        ciclos.setText(new Integer(num_ciclos).toString());
        //cambios
        stats.escribirMulti();
        stats.actualizar(2);
        //
        if((((regPC.getPalabra().getDec()-regPC.valorBase())/4)>=instrucc.tamanho()) ||
(regPC.getPalabra().getDec()>=breakpoint.getDec()))
        {
            botonesSig(false);
            segTexto.requestFocus();
        }
        if(((regPC.getPalabra().getDec()-regPC.valorBase())/4)<instrucc.tamanho())
            checkBreakpoint.setEnabled(true);
        segTexto.requestFocus();
        botonesAnt(true);
    }
     /**
     *Metodo que hace avanzar la ejecucion hasta el ciclo siguiente
     *@param ItemEvent evt
     *@return void
     **/
    private void botonCicloSigActionPerformed(java.awt.event.ActionEvent evt) {
        long posPc=regPC.getPalabra().getDec();
        int excep=-1;
        int estilo=0;
        int num=0;
        if(seguir(posPc, ins))
        {
            contador++;
            if((ins==null || (etapa=ins.getEtapa())==0))
            {
                if(ins!=null)
                {
```

```
                    ins.cambiosVisibles();
                }
                 ins=instrucc.obtener((int)regPC.getPalabra().getDec());
                 subrayar(regPC.getPalabra().getHex());
                 ins.setEtapa(etapa++);
                 if(contador!=0)
                     botonesAnt(true);
            }
            if(etapa<=ins.numEtapas())
            {
                  //cambios
                  stats.aumentarCiclos();
                  estilo=ins.estilo();
                  num=ins.numero();
                  //
                excep=ejecutaEtapa(etapa, ins);
                etapa++;
                ins.setEtapa(etapa);
                //cambios
                stats.multiciclo(estilo,num,1,etapa);
                //
            }
                 String imagen=ins.visualizarEtapa();
                 analizaExcep(excep);
                 imagen=imagen+".gif";
                 caminoDatos.setIcon(new javax.swing.ImageIcon(getClass().getResource(imagen.toLowerCase())));
                 numero=analizaInstruc(ins.numero());
                 if(ins.getEtapa()==4)
                     contador=contador+numero-1;
                 num_ciclos=contador;
                 ciclos.setFont(new java.awt.Font("Dialog", 1, 36));
                 if(ins.getEtapa()==4)
                     ciclos.setText(new Integer(num_ciclos).toString());
                 else
                     ciclos.setText(new Integer(num_ciclos).toString());
                 //cambios
                 stats.actualizar(2);
                 //
        }
    /**
     *Metodo que vuelve a la ventana del editor
     *@param ActionEvent evt
     *@return void
     **/
    private void botonVolverEditActionPerformed(java.awt.event.ActionEvent evt) {
         // cambios
        if(stats.getVentana()){
             stats.getFrame().setVisible(false);
             stats.setVentana(false);
        }
         //
        this.dispose();
        editor.setVisible(true);
    }

    /**
     *Metodo que abre la ventana de estadisticas
     *@param ActionEvent evt
     *@return void
     **/
    private void botonEstadisticasActionPerformed(java.awt.event.ActionEvent evt) {
         if(stats.getVentana()==false){
         stats.ventana(2);
         stats.actualizar(2);
        }else{
         stats.getFrame().setVisible(false);
         stats.setVentana(false);
        }
}
..
..
    jPanel26.add(jScrollPane2);

        botonEstadisticas.setFont(new java.awt.Font("Dialog", 1, 11));
        botonEstadisticas.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/ensamblador/iconos/estadisticas.png")));
        botonEstadisticas.setText("Estadísticas");
        botonEstadisticas.setHorizontalTextPosition(javax.swing.SwingConstants.CENTER);
        botonEstadisticas.setMaximumSize(new java.awt.Dimension(130, 70));
        botonEstadisticas.setMinimumSize(new java.awt.Dimension(130, 70));
        botonEstadisticas.setPreferredSize(new java.awt.Dimension(130, 70));
        botonEstadisticas.setVerticalTextPosition(javax.swing.SwingConstants.BOTTOM);
        botonEstadisticas.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                botonEstadisticasActionPerformed(evt);
```

```
            }
        });

        jPanel26.add(botonEstadisticas);

        jPanel24.add(jPanel26);

         ..
         ..
```

# A.4 MODIFICACIONES SEGMENTADO

```
/**VistaSegmentado.java
 *Clase encargada de visualizar el procesador segmentado
 **/
public class VistaSegmentado extends Vista implements Observer, Tipos {
         //cambio
    private Estadisticas stats;
    //
         private Palabra breakpoint;
    Instruccion ins;
    int i=0;
    int sel=0;
    private boolean saltoTomado=false;
    int contador;
    Registro registro;
    int num_ciclos;
    int num_instrucciones;
    private String la_memoria;
    private String la_pila;
    private Vector los_datos;
    LimitedStyledDocument lpd;
    static int MAX_CHARACTERS;
    Hashtable actions;
    String newline;
    ObservaRegistros ObReg=new ObservaRegistros();
    private ControladorSegmentado controlador;


    /**
     *Constructor de la clase segmentado
     **/
    public VistaSegmentado(Ensamblador edit, int salto, Vector<Vector> uf, Vector<Integer> tiposUF) {
        super(edit);
        setTitle(lenguaje.getString("SEGMENTADO")+"  "+editor.getNombreF());
        //cambios
        stats = new Estadisticas(editor.getNombreF());
        num_instrucciones = 0;
        controlador = new ControladorSegmentado(salto, uf, tiposUF, stats);
        inicializando();
        stats.setSalto(salto);
        stats.setUnidFuncionales(controlador.getPipeline().getEtapaEX().getUnidFuncionales());
        //
        controlador.anhadirObservador(this);
        initComponents();
        jTabbedPane1.setSelectedIndex(0);
        setMem(obtDatos(controlador.visualizarMemoria()));
        setPil(obtDatos(controlador.visualizarPila()));
        visualizarDatos();
        breakpoint=new Palabra(INICIO_PC.getDec()+controlador.getPc().length*4);
        MAX_CHARACTERS=controlador.tamanhoCodigo()*100;
        $sp.setText("$sp="+Sp.getRegistro().getPalabra().getHex().substring(2,10));
        subrayar();
        los_datos=new Vector();
        this.setSize(1064, 740);
    }
     /**
     *Metodo que hace retroceder la ejecucion hasta el ciclo anterior
     *@param ItemEvent evt
     *@return void
     **/
    protected void botonCicloAntActionPerformed(java.awt.event.ActionEvent evt) {
      if(contador>0)
```

```
      {
          inicializando();
          stats.inicializar();
          contador--;
        for(int i=0;i<contador;i++)
        {
               ins=controlador.getInstruccionActual();
               controlador.ejecutar(i,stats);
               //cambios
               stats.segmentado(controlador.getPipeline());
               //
        }
        subrayar();
        paintComponents(null);
        num_ciclos=contador;
        ciclos.setFont(new java.awt.Font("Dialog", 1, 36));
        ciclos.setText(new Integer(num_ciclos).toString());
        botonCicloSig.setEnabled(true);
        botonEjecutar.setEnabled(true);
        //cambios
        stats.setUnidFuncionales(controlador.getPipeline().getEtapaEX().getUnidFuncionales());
        stats.actualizar(3);
        //
      }
      if(contador==0)
            botonCicloAnt.setEnabled(false);
    }

    /**
     *Metodo que completa la ejecucion del codigo
     *@param ActionEvent evt
     *@return void
     **/
    protected void botonEjecutarActionPerformed(java.awt.event.ActionEvent evt) {
       int excep;
       int error = JOptionPane.WARNING_MESSAGE;
       while((((controlador.getDecimalPC()-controlador.getValorBasePC())/4)<controlador.tamanhoCodigo()) &&
(controlador.getDecimalPC()<breakpoint.getDec()))
       {
            contador++;
           ins=controlador.getInstruccionActual();
           excep = controlador.ejecutar(contador,stats);
           analizaExcep(excep);
           if(error == javax.swing.JOptionPane.ERROR_MESSAGE)
              break;
           //cambios
           stats.segmentado(controlador.getPipeline());
           //
        }
        botonCicloSig.setEnabled(false);
        botonEjecutar.setEnabled(false);
        botonCicloAnt.setEnabled(true);
        subrayar();
        paintComponents(null);
        num_ciclos=contador;
        ciclos.setFont(new java.awt.Font("Dialog", 1, 36));
        ciclos.setText(new Integer(num_ciclos).toString());
        //cambios
        stats.setUnidFuncionales(controlador.getPipeline().getEtapaEX().getUnidFuncionales());
        stats.actualizar(3);
        stats.escribirSeg();
        //
    }

    /**
     *Metodo que avanza la ejecucion un ciclo
     *@param ItemEvent evt
     *@return void
     **/
    protected void botonCicloSigActionPerformed(java.awt.event.ActionEvent evt) {
      int excep;
      botonCicloAnt.setEnabled(true);
      long posPc=controlador.getDecimalPC();
      if((((posPc-controlador.getValorBasePC())/4)<controlador.tamanhoCodigo()) && (posPc<breakpoint.getDec()))
      {
          contador++;
         ins=controlador.getInstruccionActual();
         excep = controlador.ejecutar(contador, stats);
         analizaExcep(excep);
         subrayar();
         paintComponents(null);
         num_ciclos=contador;
         ciclos.setFont(new java.awt.Font("Dialog", 1, 36));
         ciclos.setText(new Integer(num_ciclos).toString());
         //cambios
```

```
        stats.setUnidFuncionales(controlador.getPipeline().getEtapaEX().getUnidFuncionales());
        stats.segmentado(controlador.getPipeline());
        stats.actualizar(3);
        //
      }
     if((((controlador.getDecimalPC()-controlador.getValorBasePC())/4)>=controlador.tamanhoCodigo()) ||
(controlador.getDecimalPC()>=breakpoint.getDec()))
      {
          botonCicloSig.setEnabled(false);
          botonEjecutar.setEnabled(false);
          segTexto.requestFocus();
      }
    }
     /**
     *Metodo que vuelve a la ventana del editor
     *@param ActionEvent evt
     *@return void
     **/
    protected void botonVolverEditActionPerformed(java.awt.event.ActionEvent evt) {
         // cambios
        if(stats.getVentana()){
                  stats.getFrame().setVisible(false);
                          stats.setVentana(false);
        }
                  //
         this.dispose();
        editor.setVisible(true);
    }

    /**
     *Metodo que abre la ventana de estadisticas
     *@param ActionEvent evt
     *@return void
     **/
    protected void botonEstadisticasActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
FIRST:event_botonVolverEditActionPerformed
         if(stats.getVentana()==false){
                  stats.ventana(3);
         stats.actualizar(3);
        }else{
         stats.getFrame().setVisible(false);
                  stats.setVentana(false);
        }
    }
         jPanel26.add(jScrollPane2);

        botonEstadisticas.setFont(new java.awt.Font("Dialog", 1, 11));
        botonEstadisticas.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/ensamblador/iconos/estadisticas.png")));
        botonEstadisticas.setText("Estadísticas");
        botonEstadisticas.setHorizontalTextPosition(javax.swing.SwingConstants.CENTER);
        botonEstadisticas.setMaximumSize(new java.awt.Dimension(130, 70));
        botonEstadisticas.setMinimumSize(new java.awt.Dimension(130, 70));
        botonEstadisticas.setPreferredSize(new java.awt.Dimension(130, 70));
        botonEstadisticas.setVerticalTextPosition(javax.swing.SwingConstants.BOTTOM);
        botonEstadisticas.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                botonEstadisticasActionPerformed(evt);
            }
        });

        jPanel26.add(botonEstadisticas);

        jPanel24.add(jPanel26);

        Panel.add(jPanel24);

         /**Metodo para avanzar el pipeline
     *@param Instruccion instruccion que se anhade al pipeline
     *@return int tipo de avance
     **/
        public int avanzarPipeline(Instruccion inst, int ciclo, Estadisticas stats)
        {
         int avance = 0;
         int errorEX = -1, errorMEM = -1;
         //con salto fijo, si se toma el salto, se elimina la inst del hueco de retardo
            if(activarIFFlush()) {
                  etapaIF.inicializar();
                  //cambios
                  stats.aumentarRiesgosControl();
                  stats.aumentarNumNOPs();
                  stats.disminuirNumInstrucciones();
            }
            if (!(inst.toString().equals("syscall")))
            {
```

```
            if (!unidadDeteccionRiesgos.detener(stats)) {
              etapaWB.avanzar(etapaMEM);
              etapaWB.ejecutar();
              etapaMEM.avanzar(etapaEX);
              errorMEM = etapaMEM.ejecutar();
              etapaEX.moverCiclo(etapaMEM);
              etapaEX.recordarCiclo(ciclo);
              avance = etapaEX.avanzar(etapaID);
              errorEX = etapaEX.ejecutar();
              if(avance == -1){  //avance=-1 -> se ha insertado instruccion en UF
                       //cambios
                       stats.aumentarMemoriaInstruccion();
                       stats.aumentarNumInstrucciones();
                       //
                       etapaID.avanzar(etapaIF);
                       etapaID.ejecutar();
                       saltoTomado = etapaID.getEstadoInstruccion().getInstruccion().esSaltoTomado();
                       etapaIF.anhadirInstruccion(inst);
                        etapaIF.avanzar(etapaIF);
                       if(!saltoTomado) {
                       if(etapaID.getEstadoInstruccion().getInstruccion() instanceof SaltoCondicional) {;}
                       else {
                                etapaIF.ejecutar();
                       }
                   }
              }else {
                       //cambios
                       stats.aumentarRiesgosEstructurales();
                       stats.aumentarNumNOPs();}
                       //
              avance = 0;       //avance=0-->avance de todo el pipeline
            }
            else {
              etapaWB.avanzar(etapaMEM);
              etapaWB.ejecutar();
              etapaMEM.avanzar(etapaEX);
              errorMEM = etapaMEM.ejecutar();
              etapaEX.moverCiclo(etapaMEM);
              errorEX = etapaEX.ejecutar();
              etapaID.ejecutar();
              saltoTomado = etapaID.getEstadoInstruccion().getInstruccion().esSaltoTomado();
              //cambios
              stats.aumentarNumNOPs();
              //
              avance = 1;         //avance=1-->se inserta burbuja en etapa EX
            }
        }
        else {
            if (!estaParado()) {
                if (!unidadDeteccionRiesgos.detener(stats)) {
                       etapaWB.avanzar(etapaMEM);
                       etapaWB.ejecutar();
                       etapaMEM.avanzar(etapaEX);
                       errorMEM = etapaMEM.ejecutar();
                       etapaEX.moverCiclo(etapaMEM);
                       etapaEX.recordarCiclo(ciclo);
                       avance = etapaEX.avanzar(etapaID);
                       errorEX = etapaEX.ejecutar();
                       if(avance == -1){
                       etapaID.avanzar(etapaIF);
                       etapaID.ejecutar();
                       saltoTomado = etapaID.getEstadoInstruccion().getInstruccion().esSaltoTomado();
                       etapaIF.anhadirInstruccion(new Nop());
                       etapaIF.avanzar(etapaIF);
                       }
                       avance = 2;      //avance=2-->se inserta burbuja en etapa IF
                }
                else {
                       etapaWB.avanzar(etapaMEM);
                       etapaWB.ejecutar();
                       etapaMEM.avanzar(etapaEX);
                       errorMEM = etapaMEM.ejecutar();
                       etapaEX.moverCiclo(etapaMEM);
                       errorEX = etapaEX.ejecutar();
                       etapaID.ejecutar();
                       saltoTomado = etapaID.getEstadoInstruccion().getInstruccion().esSaltoTomado();
                       avance = 1;         //avance=1-->se inserta burbuja en etapa EX
                }
            }
            else {
              etapaWB.avanzar(etapaMEM);
              etapaWB.ejecutar();
              errorEX =  inst.ejecutar();
              avance = 3;         //avance=3-->ejcucion de SYSCALL
            }
```

```
    }
    activarIFFlush(inst);
    construirDiagMult();
    return ((errorMEM != -1) ? errorMEM : errorEX);
}
```

# ANEXO B. **CONTENIDO DEL CD**

En el contenido del CD que acompaña a la memoria podemos encontrar los siguientes recursos:

- Memoria del trabajo en el formato PDF dentro del directorio Memoria.
- Código fuente del simulador modificado dentro del directorio Código fuente.
- Dentro del directorio Simulador se encuentra la nueva versión del simulador MIPS Simula3MS modificado en este trabajo, el manual de usuario y los 2 códigos que se utilizan en el ejemplo: APXPY_MIPS.s y Practica5_parte1.s.

# Apéndice A

# Manual de usuario

En este manual se explica como utilizar *Simula3MS*. En la sección A.1 se enumeran los pasos básicos para usar la herramienta. A continuación se describen de forma más detallada la edición de programas en lenguaje ensamblador para este procesador, los distintos elementos del simulador: el segmento de texto, el segmento de datos, los registros, etc. así como las distintas configuraciones.

## A.1. Guía rápida

A continuación se indican los pasos básicos para empezar a trabajar con *Simula3MS*:

1. Una vez abierta la ventana de *Simula3MS*, existen dos opciones:
   - Cargar un fichero que ha sido editado con anterioridad.
   - Editar un nuevo código en lenguaje ensamblador.
2. Una vez editado o cargado el fichero, el siguiente paso es ensamblarlo, para ello hay que pulsar el botón *Ensamblar*. A partir de aquí hay dos posibles resultados:

- Si el código que queremos ejecutar no tiene errores sintácticos se activará el botón *Ejecutar* que permite acceder a la ventana de la simulación de la ejecución del código analizado.
- En caso de que el código no sea correcto, en la parte inferior de la ventana aparecerá un listado con todos los errores y el primero de ellos aparecerá remarcado. Se puede acceder a los siguientes, en caso de que los hubiera, por medio del botón *Error siguiente*. Una vez corregidos estos fallos se vuelve a pulsar botón *Ensamblar* y se repite este paso.

3. El paso siguiente, previo a ejecutar, será escoger la configuración del simulador sobre la que queremos que se ejecute el código. Para ello, en el menú *Configuración* tenemos tres posibles opciones: *Entrada/Salida, Camino de datos* y *Técnicas de salto*. Por defecto la opción activada es el camino de datos *Monociclo* con la *Entrada/Salida* deshabilitada.

   - *Entrada/Salida*. La *Entrada/Salida* aparece inicialmente desactivada. En caso de querer realizar una simulación orientada al estudio de los mecanismos de *Entrada/Salida* se puede elegir entre *Entrada/Salida* con encuesta o *Entrada/Salida* con interrupciones.
   - *Camino de datos*. La opción seleccionada por defecto es el camino de datos *Monociclo*. Al escoger *Multiciclo* o cualquiera de las implementaciones del procesador *Segmentado*, se abrirá una nueva ventana formulario que permite configurar la latencia de las operaciones en punto flotante.
   - *Técnicas de salto*. Actualmente hay implementadas en *Simula3MS* dos técnicas de salto: *Salto retardado* y *Salto fijo*. Ambas aparecen inicialmente desactivadas y la selección de cualquiera de estas técnicas implica escoger el pipeline básico.

4. Una vez obtenido, el código correcto y configurado el simulador, se pulsa *Ejecutar* y tenemos acceso a la ventana en la cual se simula la ejecución

del código escogido. En esta ventana se puede observar la ejecución del programa completo usando el botón ejecutar, de modo que se mostrarán sólo los valores finales, o bien ciclo a ciclo, mediante los botones ciclos siguiente y ciclo anterior, pudiendo ver así las modificaciones que cada instrucción realiza en cada ciclo.

## A.2. Manual extendido

### A.2.1. Edición del código

El primer paso al iniciar *Simula3MS* es editar un código en lenguaje ensamblador, para ello se puede cargar el código de un fichero o bien editarlo. La sintaxis básica utilizada por *Simula3MS* tiene las siguientes características:

- Los comentarios empiezan por el símbolo #, todo lo que aparezca en la misma línea a continuación de este símbolo es ignorado.
- Los programas se dividen en dos partes:
  - *.text:* sección obligatoria en todos los programas, contiene el conjunto de las instrucciones del programa.
  - *.data:* sección opcional, aunque normalmente necesaria. Es la sección de declaración de las variables del programa.

#### Características de .text

La sección *.text* es obligatoria en todos los programas, contiene el conjunto de instrucciones del programa. Los elementos siguientes se guardan en el segmento de texto.

Las primeras líneas de la sección .text, obligatorias ya que indican al simulador donde debe empezar la ejecución del programa, son:

*.globl main*

*.main:*

A continuación se escribe el código del programa, siguiendo estas reglas:

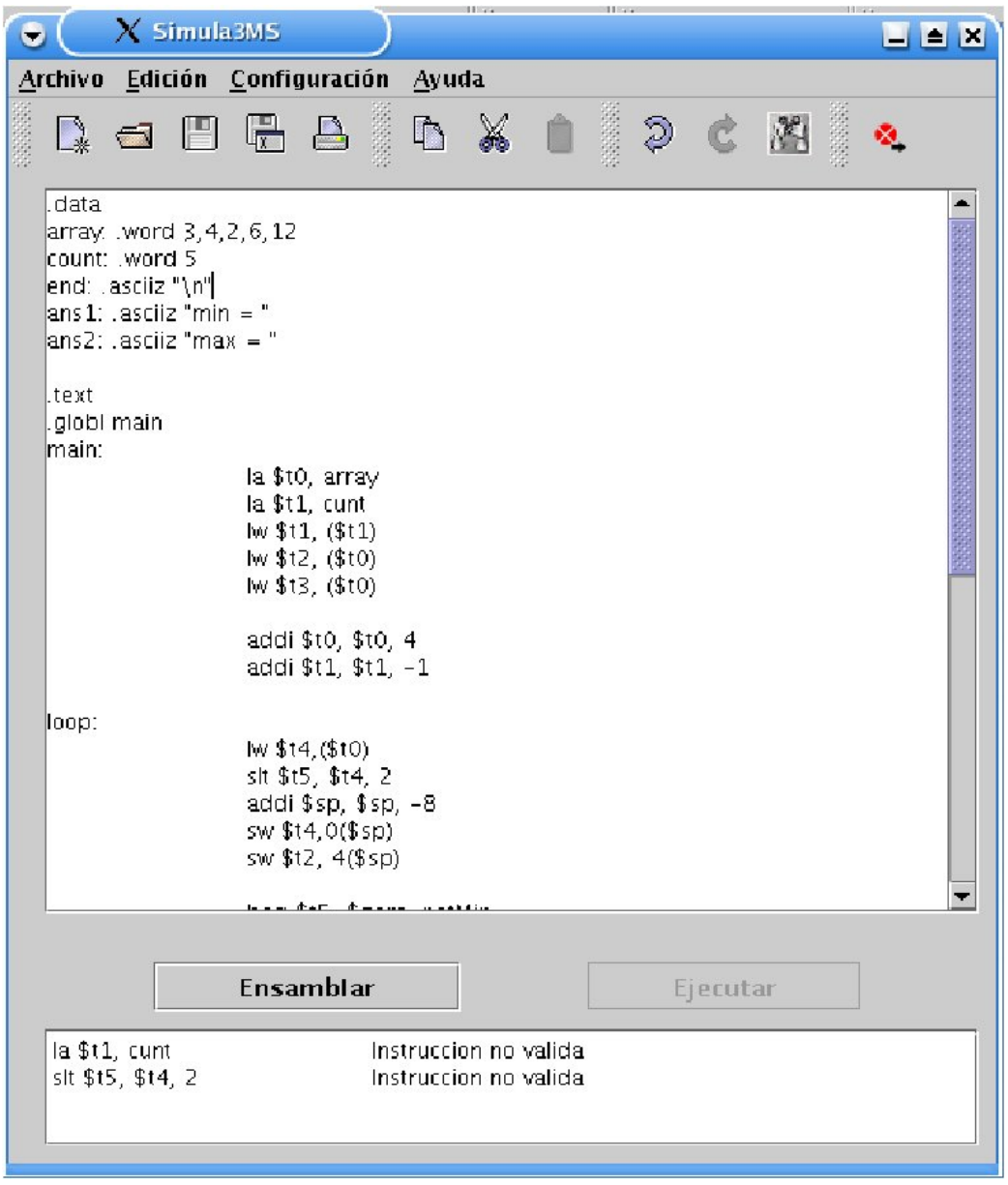



*Figura A.1:* Ventana del editor

- Las etiquetas van seguidas por dos puntos (:). Una etiqueta válida es una secuencia de caracteres alfanuméricos, subrayados (_) y puntos (.).
- En la línea de la etiqueta no puede haber una instrucción, las instrucciones se situarán a partir de la línea siguiente a la etiqueta.
- Las instrucciones válidas implementadas en este simulador se muestran en tablas según su tipo. Las enteras tipo R en el cuadro A.1, las tipo I en el A.2 y las tipo J en la tabla A.3. En cuanto a las de punto flotante, las de tipo R se muestran en los cuadros A.4, A.5 y A.6, mientras que las de tipo I están en el cuadro A.7.
- Los registros enteros pueden indicarse por su número de orden o por su nombre, por ejemplo $t1=$9. Las equivalencias se muestran en el cuadro A.9. En el caso de los registros de punto flotante, también se permiten ambas nomenclaturas pero la equivalencia se limita a suprimir la f. Ejemplo: $f6=$6.
- Por defecto los números se representan en base 10.
- Se pueden incluir llamadas al sistema operativo (*syscall*), para solicitar algún servicio como impresión por pantalla, ver sección A.5.
- Con el objetivo de ayudar al usuario a cargar variables, *Simula3MS* tiene también implementadas las pseudoinstrucciones `la` y `li`. Por ejemplo:

  `la $1, variable`

  carga en el registro $1 la dirección de memoria donde está almacenada la variable. La ejecución de está pseudoinstrucción se divide en dos instrucciones:

  `lui $1, 0x1001` (representa los 16 bits de mayor peso de la dirección de memoria)

  `ori $ra, 0x0004` (16 bits de menor peso de la dirección de memoria)

| **Inst.** | **Definición** | **Formato** |
|---|---|---|
| add | Almacena en el registro $1 el valor de la suma de los otros dos | add $1, $2, $3 |
| sub | Almacena en el $1 el valor de la resta de $2 (minuendo), menos el $3 (sustraendo) | sub $1, $2, $3 |
| and | Almacena en el $1 el resultado de hacer una operación AND entre los otros dos registros | and $1, $2, $3 |
| or | Almacena en el $1 el resultado de hacer una operación OR entre los registros $2 y $3 | or $1, $2, $3 |
| slt | Coloca un 1 en el registro $1 si el valor almacenado en $2 es menor que el de $3. Si no lo es se almacena un 0 en $1. | slt $1, $2, $3 |
| jr | Modifica la dirección del PC, por aquel valor almacenado en el registro $1 | jr $1 |
| div | Se almacena en el HI el resto y en LO el cociente de dividir $1 (dividendo) entre $2(divisor). | div $1, $2. |
| mult | Se almacena en el registro LO los 32 bits de menor peso y en HI los 32 de mayor peso de la multiplicar $1 y $2. | mult $1, $2. |
| mfhi | Copia el valor almacenado en el registro HI en $1 | mfhi $1. |
| mflo | Copia el valor almacenado en LO en el registro $1 | mflo $1. |
| mfc0 | Transfiere el registro $2 del coprocesador 0 al registro $1 de la CPU. | mfc0 $1, $2 |
| sll | Almacena en el registro $1 el valor del registro $2 desplazado n bits a la izquierda | sll $1, $2, n |
| srl | Almacena en el registro $1 el valor del registro $2 desplazado n bits a la derecha | srl $1, $2, n |
| sra | Almacena en el registro $1 el valor del registro $2 después de hacer un desplazamiento aritmético de n bits a la derecha | sra $1, $2, n |

*Cuadro A.1:* Instrucciones tipo R

| **Inst.** | **Definición** | **Formato** |
|---|---|---|
| addi | Se almacena en el registro $1 la suma del registro $2 y el valor de la constante | addi $1, $2, constante |
| andi | El registro $1 tiene el resultado de una operación AND entre $2 y la constante | andi $1, $2, constante |
| ori | El registro $1 tiene el resultado de una operación OR entre $2 y la constante | ori $1, $2, constante |
| slti | Almacena en $1 un 1 si el valor registro $2 es menor que el de la constante. En caso contrario almacena un 0 | slti $1, $2, constante |
| lui | Carga la constante en los 16 bits más significativos del registro $1 | lui $1, constante |
| lw | Carga en el registro $1 la palabra almacenada en la dirección de memoria que contiene el registro $2 más el desplazamiento. La nueva dirección calculada debe ser múltiplo de cuatro | lw $1, desplazamiento($2) |
| lb | Carga en el registro $1 el byte de memoria apuntado por la dirección almacenada en el registro $2 más el desplazamiento | lb $1, desplazamiento($2) |
| sw | Almacena en memoria en la posición obtenida de sumarle el desplazamiento a la dirección del registro $2, la palabra del registro $1. La dirección debe ser múltiplo de 4. | sw $1, desplazamiento($2) |
| sb | Almacena en la posición de memoria correspondiente al valor de $2 más el desplazamiento, el primer byte de la palabra almacenada en $1. | sb $1, desplazamiento($2). |
| beq | Si el valor de $1 y $2 es igual se modifica el valor del PC para pasar a ejecutar el trozo de código apuntado por la etiqueta. | beq $1, $2, etiqueta. |
| bne | Si el valor de $1 y $2 no es igual se modifica el valor del PC para pasar a ejecutar el trozo de código apuntado por la etiqueta. | bne $1, $2, etiqueta. |

*Cuadro A.2:* Instrucciones tipo I

| Inst. | Definición | Formato |
|---|---|---|
| j | Modifica el valor del PC para ejecutar las instrucciones siguientes a la etiqueta | j etiqueta |
| jal | Modifica el valor del PC por aquel al que apunta la etiqueta y almacena la dirección actual del PC en $ra | jal etiqueta |

*Cuadro A.3:* Instrucciones tipo J

### Características de .data

La sección *.data* contiene la declaración de las variables del programa. Los elementos siguientes se guardan en el segmento de datos, en concreto en la memoria. Esta sección es opcional.

La declaración de las variables del programa se ajusta a las siguientes reglas:

- En cada línea no puede haber más de una etiqueta.
- La declaración de una variable sigue este formato:
  - En primer lugar debe ir un identificador de variable válido (etiqueta). Se considera válida cualquier secuencia de caracteres alfanuméricos, subrayados y puntos.
  - A continuación se indica el tipo de variable. La tabla A.8 muestra una relación de los tipos implementados en *Simula3MS*, sus características y la estructura de la definición.
  - Finalmente se inicializa la variable.
- Las cadenas de caracteres se encierran entre comillas dobles.
- Los números se consideran en base 10 por defecto. Si van precedidos del prefijo 0x se interpretan en hexadecimal.

La Figura A.1 muestra un ejemplo de código.

| **Inst.** | **Definición** | **Formato** |
|---|---|---|
| add.s | Se almacena en el registro $f1 el valor de la suma de los otros dos (precisión simple) | add.s $f1, $f2, $f3 |
| add.d | Se almacena en el registro $f0 el valor de la suma de los otros dos (precisión doble) | add.d $f0, $f2, $f4 |
| sub.s | Se almacena en el registro $f1 el valor de la resta de los otros dos (precisión simple) | sub.s $f1, $f2, $f3 |
| sub.d | Se almacena en el registro $f0 el valor de la resta de los otros dos (precisión doble) | sub.d $f0, $f2, $f4 |
| mul.s | Se almacena en el registro $f1 el valor de la multiplicación de los otros dos (precisión simple) | mul.s $f1, $f2, $f3 |
| mul.d | Se almacena en el registro $f0 el valor de la multiplicación de los otros dos (precisión doble) | mul.d $f0, $f2, $f4 |
| div.s | Se almacena en el registro $f1 el valor de la división de los otros dos (precisión simple) | div.s $f1, $f2, $f3 |
| div.d | Se almacena en el registro $f0 el valor de la división de los otros dos (precisión doble) | div.d $f0, $f2, $f4 |

*Cuadro A.4:* Instrucciones tipo R con 3 argumentos

| Inst. | Definición | Formato |
|---|---|---|
| mov.s | Transfiere el número en punto flotante de precisión simple del registro $f2 al $f1 | mov.s $f1, $f2 |
| mov.d | Transfiere el número en punto flotante de precisión doble del registro $f2 al $f0 | mov.s $f0, $f2 |
| abs.s | Calcula el valor absoluto del número en punto flotante de precisión simple del registro $f2 y lo almacena en $f1 | abs.s $f1, $f2 |
| abs.d | Calcula el valor absoluto del número en punto flotante de precisión doble del registro $f2 y lo almacena en $f0 | abs.d $f0, $f2 |
| neg.s | Niega el número en punto flotante de precisión simple del registro $f2 y lo almacena en $f1 | neg.s $f1, $f2 |
| neg.d | Niega el número en punto flotante de precisión doble del registro $f2 y lo almacena en $f0 | neg.d $f0, $f2 |
| c.eq.s | Si los valores de los registros de precisión simple $f1 y $f2 son iguales se escribe un 1 en el registro status | c.eq.s $f1, $f2 |
| c.eq.d | Si los valores de los registros de precisión doble $f0 y $f2 son iguales se escribe un 1 en el registro status | c.eq.d $f0, $f2 |
| c.le.s | Si el valor del registro $f1 es menor o igual que el del registro $f2, ambos de precisión simple, se escribe un 1 en el registro status | c.le.s $f1, $f2 |
| c.le.d | Si el valor del registro $f0 es menor o igual que el del registro $f2, ambos de precisión doble, se escribe un 1 en el registro status | c.le.d $f0, $f2 |
| c.lt.s | Si el valor del registro $f1 es menor que el del registro $f2, ambos de precisión simple, se escribe un 1 en el registro status | c.lt.s $f1, $f2 |
| c.lt.d | Si el valor del registro $f0 es menor que el del registro $f2, ambos de precisión doble, se escribe un 1 en el registro status | c.lt.d $f0, $f2 |
| cvt.d.s | Convierte el número en punto flotante de precisión simple del registro $f1 a un número de precisión doble y lo guarda en $f2 | cvt.d.s $f2, $f1 |
| cvt.s.d | Convierte el número en punto flotante de precisión doble del registro $f2 a un número de precisión simple y lo guarda en $f1 | cvt.s.d $f1, $f2 |

*Cuadro A.5:* Instrucciones tipo R con 2 argumentos

| **Inst.** | **Definición** | **Formato** |
|---|---|---|
| mtc1 | Transfieren el registro $t1 de la CPU al registro $f1 del coprocesador de punto flotante | mtc1 $f1, $t1 |
| mfc1 | Transfieren el registro $f1 del coprocesador de punto flotante al registro $t1 de la CPU | mfc1 $t1, $f1 |
| cvt.d.w | Convierte el número entero del registro $t1 a un número de precisión doble y lo guarda en $f2 | cvt.d.w $f2, $t1 |
| cvt.s.w | Convierte el número entero del registro $t1 a un número de precisión simple y lo guarda en $f1 | cvt.s.w $f1, $t1 |
| cvt.w.d | Convierte el número en punto flotante de precisión doble del registro $f2 a un número entero y lo guarda en $t1 | cvt.w.d $t1, $f2 |
| cvt.w.s | Convierte el número en punto flotante de precisión simple del registro $f1 a un número entero y lo guarda en $t1 | cvt.w.s $t1, $f1 |

*Cuadro A.6:* Instrucciones que usan registros enteros

| **Inst.** | **Definición** | **Formato** |
|---|---|---|
| lwc1 | Carga en el registro $f1 la palabra almacenada en la dirección de memoria que contiene el registro $s2 más el desplazamiento.La nueva dirección debe ser múltiplo de 4 | lwc1 $f1, desplazamiento($s2) |
| swc1 | Se almacena la palabra del registro $f1 en la posición de memoria obtenida al sumar la dirección que contiene el registro $s2 más el desplazamiento.La dirección debe ser múltiplo de 4 | swc1 $f1, desplazamiento($s2) |
| bc1t | Si status=1 se modifica el valor del PC para ejecutar el trozo de código apuntado por la etiqueta | bc1t etiqueta |
| bc1f | Si status=0 se modifica el valor del PC para ejecutar el trozo de código apuntado por la etiqueta | bc1f etiqueta |

*Cuadro A.7:* Instrucciones tipo I

| Tipo | Definición | Estructura de la definición |
|---|---|---|
| .ascii | Almacena en memoria el string como una lista de caracteres | variable: .ascii "string a almacenar" |
| .asciiz | Almacena en memoria el string como una lista de caracteres y lo termina con 0 | variable: .asciiz "string a almacenar" |
| .word | Almacena la lista de palabra en posiciones secuenciales de memoria | variable: .word palabra1, palabra2, ... |
| .space | Reserva n bytes de espacio en la memoria. | variable: .space n |
| .float | Almacena la lista de números en punto flotante de simple precisión en posiciones sucesivas de memoria. | variable: .float f1, f2, ... |
| .double | Almacena la lista de números en punto flotante de doble precisión en posiciones sucesivas de memoria. | variable: .double d1, d2, ... |

*Cuadro A.8:* Tipos de datos en *Simula3MS*

**Ensamblar**

Una vez editado o cargado el código del programa es necesario ensamblarlo, para ello se utiliza el botón *Ensamblar*. Si el código es sintácticamente correcto se activará el botón *Ejecutar* y se puede avanzar a la simulación del programa, si no es necesario corregir el código teniendo en cuenta los errores. Para facilitar esta tarea podemos ayudarnos del botón *ErrorSiguiente* que, como su nombre indica, avanza al siguiente error de forma cómoda.

### A.2.2. Configuración del camino de datos del simulador

Por defecto al inicializar *Simula3MS* la configuración del procesador es monociclo, pero puede cambiarse en cualquier momento, bien antes de editar el código o después de terminar de escribir el programa en ensamblador, o bien después de realizar cualquier simulación previa.

Para cambiar la configuración acceder al menú *Configurar/Camino de datos* en la barra de herramientas y seleccionar la opción adecuada. En caso de escoger la configuración multiciclo, o cualquiera de las opciones de la configuración segmentada, se mostrará una nueva ventana en la que además, podremos configurar la latencia de las unidades funcionales de punto flotante.

## A.3. Ventana de registros

Los registros se agrupan en tres clases:

- **Registros especiales:**
  - *PC* es el contador del programa (*program couter*). Este registro se inicializa por el sistema operativo apuntando a la dirección de la primera instrucción del programa en memoria. Al cargar cualquier instrucción de memoria el PC se incrementa de forma que la CPU tendrá el valor de la dirección de la siguiente instrucción que va a ser cargada.

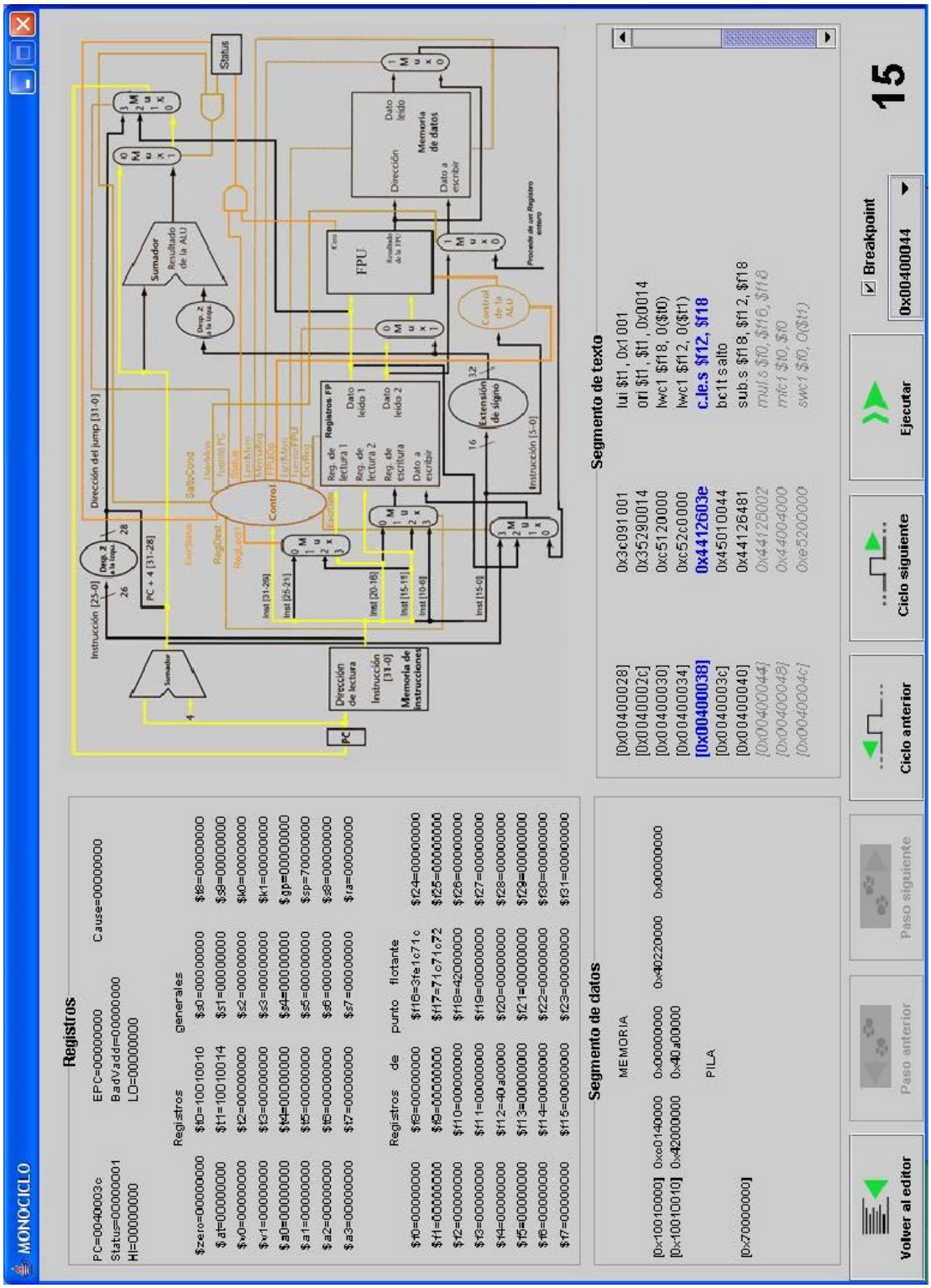



*Figura A.2:* Ventana de la configuración monociclo

- *EPC, Cause, Status* y *BadVaddr* son registros utilizados para el manejo de instrucciones e interrupciones.
  - El registro *EPC* contiene la dirección de la instrucción que ha causado la excepción. En el caso de *Entrada/Salida con interrupciones* indica la instrucción que se estaba ejecutando en el momento que se produjo la interrupción.
  - El registro *Cause* almacena el tipo de excepción y bits de interrupción pendiente.
  - El registro *Status* se usa como condición de salto en las instrucciones *bc1t* y *bc1f*. En la *Entrada/Salida con interrupciones* almacena la máscara de interrupciones y los bits de autorización.
  - El registro *BadVaddr* contiene la dirección en la que ha ocurrido la referencia de memoria. Este registro no se usa actualmente en *Simula3MS*.
- *Hi* y *Lo* son dos registros para poder operar con operandos de 64 bits, como sucede en el caso de la multiplicación y la división.

- **Registros generales:** La arquitectura MIPS posee 32 registros genéricos de 32 bits ($0-$31) para uso del programador (ver tabla A.9). Existen reglas de *uso de los registros*, también llamadas *convención de llamada a procedimiento*, que indican cual debe ser el uso de los registros, especialmente en las llamadas a procedimientos, que hace el programa. Tal como indica su nombre, estas reglas son, en su mayor parte, convenciones seguidas por los programas más que reglas forzadas por la circuitería. De todos modos la mayoría de los programadores se esfuerzan por seguir estas convenciones porque violarlas sería fuente de un mayor número de errores.

- **Registros de punto flotante:** La arquitectura MIPS R2000 (que simula esta herramienta) no dispone de unidad de punto flotante incluida

| Nombre | Número | Uso |
|---|---|---|
| $zero | 0 | Valor constante 0. Sólo es de lectura. |
| $at | 1 | Reservado por el ensamblador. |
| $v0-$v1 | 2-3 | Valores para resultados y evaluación de expresiones. |
| $a0-$a3 | 4-7 | Argumentos. |
| $t0-$t7 | 8-15 | Registros temporales. |
| $s0-$s7 | 16-23 | Registros salvados. |
| $t8-$t9 | 24-25 | Registros temporales. |
| $k0-$k1 | 26-27 | Reservados por el Sistema Operativo. |
| $gp | 28 | Puntero global. No implementado en *Simula3MS*. |
| $sp | 29 | Puntero de pila. |
| $fp | 30 | Puntero de bloque de activación. No implementado en *Simula3MS*. |
| $ra | 31 | Dirección de retorno de las subrutinas. |

*Cuadro A.9:* Convención de los registros MIPS

en el microprocesador si no que implementa estas funciones en coprocesadores separados. La arquitectura MIPS tiene en cada coprocesador 32 registros de 32 bits para punto flotante ($f0-$f31), que pueden ser organizados en 16 registros de doble precisión, con 64 bits (para ello se toman las designaciones par de los registros).

## A.4. Memoria

*Simula3MS* divide la memoria en cuatro partes.

- La primera parte, junto al extremo inferior del espacio de direcciones, es el *segmento de texto*, que contiene las direcciones del programa y empieza en la dirección 0x00400000.
- La segunda parte, sobre el segmento de texto, es el *segmento de datos*, que se divide en dos partes:

  - Los *datos estáticos*, a partir de la dirección 0x10010000. Estos datos contienen objetos cuyo tamaño es conocido por el compilador y su tiempo de vida es la ejecución entera del programa.
  - Inmediatamente después de los datos estáticos están los *datos dinámicos*. Como su nombre indica, estos datos son reservados por el programa durante su ejecución. Como el compilador no puede predecir cuánta memoria va a reservar un programa el sistema operativo extiende el área de datos dinámicos para satisfacer la demanda.

- La tercera parte, la *pila* del programa reside en el extremo superior del espacio de direcciones virtual, situado en la dirección 0x70000000. El tamaño máximo de la pila que usará un programa no se puede conocer previamente, por lo tanto es necesario que el usuario desplace el puntero de pila hacia abajo antes de insertar nuevos elementos.

- *Las direcciones de Entrada/Salida* se sitúan a partir de la dirección 0xffff0000. Un procesador MIPS R2000 utiliza *Entrada/Salida mapeada en memoria*. *Simula3MS* cuenta con cuatro posiciones de memoria de *Entrada/Salida*, dos de ellas dedicadas a la entrada y las otras dos a la salida.

  - *Control del receptor*, se encuentra en la posición 0xffff0000. Se pone a 1 el último bit para indicar que ha llegado un carácter del teclado que todavía no se ha leído. Se pone a 0 en cuanto se lee el dato de *Datos del receptor*. Por otro lado, el segundo bit menos significativo indica si las interrupciones están activadas.
  - *Datos del receptor*, se encuentra en la posición 0xffff0004 y contiene el último carácter escrito en el teclado.
  - *Control del transmisor*, se encuentra en la posición 0xffff0008. Un 0 en el último bit, indicaría que el transmisor todavía está ocupado escribiendo el carácter anterior, pero esto no se contempla

en *Simula3MS*, por lo que siempre tendrá a 1 el último bit. Al igual que en el caso de *Control del receptor*, el segundo bit menos significativo indica si las interrupciones están activadas.

- *Datos del transmisor*, se encuentra en la posición 0xffff000c. Esta posición de memoria almacena en sus 8 bits menos significativos el valor en ASCII del carácter que será visualizado por el monitor.

Esta división de la memoria no es la única posible. De todos modos, tiene como característica más importante que la pila y la memoria están lo más alejadas posibles, de forma que la pila puede crecer hasta ocupar el espacio de direcciones del programa por entero.

### A.4.1. Segmento de datos

En la representación de *Simula3MS*, la pila y el segmento de datos explicado anteriormente se agrupan en *segmento de datos* ya que ambos almacenan los datos necesarios durante la ejecución del programa.

En *Simula3MS* la memoria está dividida en palabras de 32 bits, por lo que hay 4 bytes almacenado en cada palabra. Debido a esta organización de la memoria nos encontramos con un requisito denominado *restricción de alineación* para el acceso a las palabras de memoria, ya que las direcciones deben ser siempre múltiplo de 4 y la dirección de una palabra es la de su primer byte. Esta restricción de alineación facilita la transferencia de datos más rápidamente.

Otro aspecto curioso está relacionado con la forma en que se representan y referencian los bytes dentro de una palabra. En *Simula3MS* se sigue el convenio *Little Endian* ya que se usa la dirección del byte de más a la derecha o de menor peso como dirección de palabra.

Cada línea del segmento de texto de *Simula3MS* (ver figura A.2) sigue el formato:

El primer valor, entre corchetes, indica la dirección en memoria a partir de la cual se almacena la primera palabra, las otras palabras están situadas

en posiciones sucesivas de memoria.

Las restantes cuatro palabras representan los valores, en hexadecimal, de los datos almacenados en esas direcciones.

En la pila el formato es aproximadamente el mismo, pero sólo se indica la dirección a la que apunta el puntero de pila. Hay que recordar que para que la pila crezca es necesario desplazar el puntero de pila hacia posiciones inferiores.

### A.4.2. Segmento de texto

El *segmento de texto* muestra las instrucciones del programa que se cargan automáticamente cuando se empieza la simulación. Cada instrucción se muestra en una línea (ver figura A.2) de la siguiente forma:

- El primer número de la línea, entre corchetes, es la dirección de memoria hexadecimal de la instrucción.
- El segundo número es la codificación numérica de la instrucción, es decir, el valor que almacenaría el procesador para la instrucción en lenguaje máquina (también se muestra en hexadecimal).
- En último lugar se muestra la descripción de la instrucción. En todos los casos la descripción de la instrucción coincide con la instrucción del código del programa, excepto en el caso de las pseudoinstrucciones *la* y *li*, las únicas aceptadas por *Simula3MS*.

## A.5. Llamadas al sistema

Simula3MS ofrece un pequeño conjunto de servicios de llamada al sistema operativo a través de la instrucción de llamada al sistema (*syscall*). Para pedir un servicio el programa carga el código de llamada al sistema (tabla A.10) en el registro $v0 y los argumentos en los registros $a0 y $a1 (o $f12 para valores

de punto flotante). Las llamadas al sistema que devuelven valores ponen sus resultados en el registro $v0 (o $f0 para resultados de punto flotante).

Existen llamadas al sistema para salida de datos:

- A la llamada al sistema *print_ int* se le pasa un valor y lo muestra como un entero en un diálogo de información.
- *print_float* muestra en el diálogo el valor en punto flotante que se le pasa.
- *print_ double* hace lo mismo que *print_float* pero permite un rango más amplio de números en punto flotante.
- *print_ string* muestra en el diálogo la cadena almacenada a partir de la dirección que indica $a0 hasta que encuentra el carácter de finalización de cadena.

También hay llamadas al sistema para entrada de datos, estos datos se solicitan al usuario a través de diálogos. Son:

- *read_ int* almacena el entero que introduce el usuario en $v0.
- *read_float* almacena el número de punto flotante que le introduce el usuario en el registro $f0 con formato de simple precisión.
- *read_ double* almacena el número de punto flotante que le introduce el usuario en los registros $f0 y $f1, puesto que el formato de doble precisión necesita dos registros para almacenar un valor.
- *read_ string* almacena la dirección de comienzo de la cadena que introduce el usuario en el registro $a0 y la longitud de dicha cadena en $a1. Estos datos son reservados por el programa durante su ejecución y por tanto se consideran datos dinámicos.

Otra llamada al sistema muy utilizada es *exit* utilizada para indicar al procesador que se ha terminado la ejecución del programa.

| **Código** | **Nombre** | **Operación** |
|---|---|---|
| 1 | *print_int* | Imprime como un entero aquello que se encuentra en $a0. |
| 2 | *print_float* | Imprime como un número en punto flotante de aquello que se encuentra en $f12. |
| 3 | *print_double* | Imprime como un número en punto flotante aquello que se encuentra en $f12 y $f13, considerando que forman un sólo registro de doble precisión. |
| 4 | *print_string* | Imprime como un string aquello que se encuentra en la posición indicada por $a0. |
| 5 | *read_int* | Solicita un entero que se almacenará en el registro $v0. |
| 6 | *read_float* | Solicita un número en punto flotante que se almacenará en el registro $f0 con formato de simple precisión. |
| 7 | *read_double* | Solicita un número en punto flotante que se almacenará en el registro $f0 (y $f1) con formato de doble precisión. |
| 8 | *read_string* | Solicita un string que se almacena en $a0 y cuya longitud se guarda en $a1. |
| 10 | *exit* | Finaliza la ejecución. |

*Cuadro A.10:* Códigos asociados a syscall

## A.6. Punto de ruptura

Los puntos de ruptura se utilizan para parar la ejecución del simulador en una instrucción determinada.

En el caso de *Simula3MS* pueden insertarse y desactivarse en cualquier momento de la ejecución mediante un menú desplegable que contiene una lista de las direcciones de memoria donde están almacenadas las instrucciones. Esta lista de direcciones comprende desde la de la instrucción siguiente a la que está en ejecución hasta la última del código.

## A.7. Simulación con el procesador monociclo

Un procesador monociclo se caracteriza por la ejecución de cada instrucción en un ciclo de reloj.

La simulación de este procesador en *Simula3MS* se representa en la figura A.2 cuyas características han sido detalladas en los apartados anteriores.

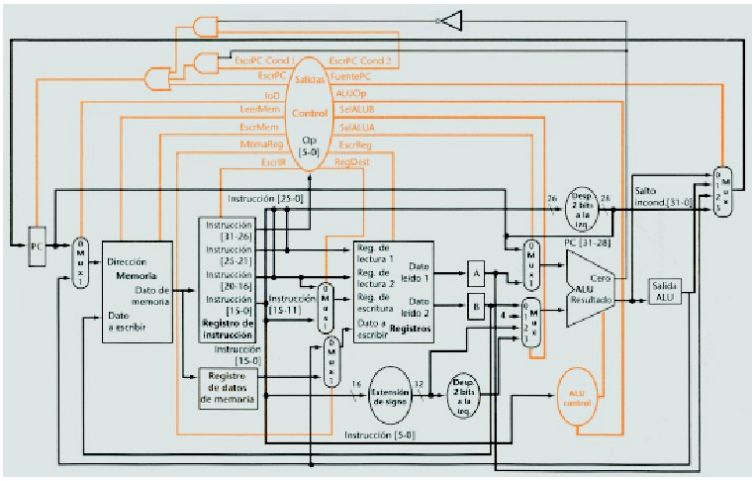



*Figura A.3:* Primer modelo de camino de datos

Tanto en el procesador monociclo como en el muticiclo, podemos distinguir 3 tipos de caminos de datos. En primer lugar están los correspondientes

a instrucciones enteras (ver figura A.3) que constan de una ALU y un banco de registros enteros.

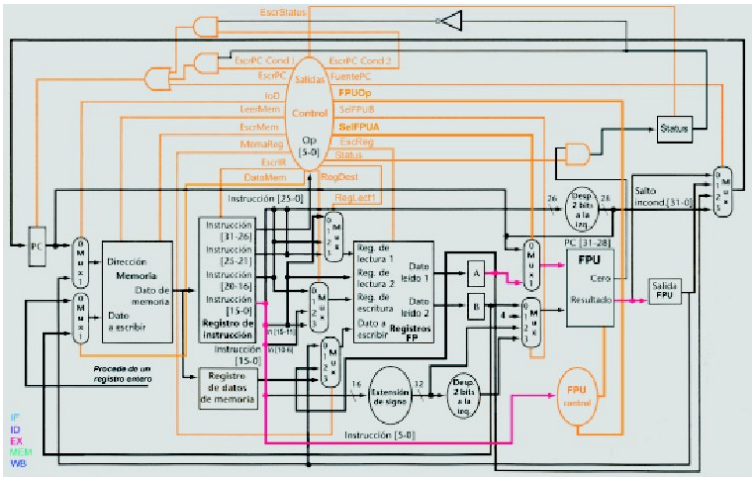



*Figura A.4:* Segundo modelo de camino de datos

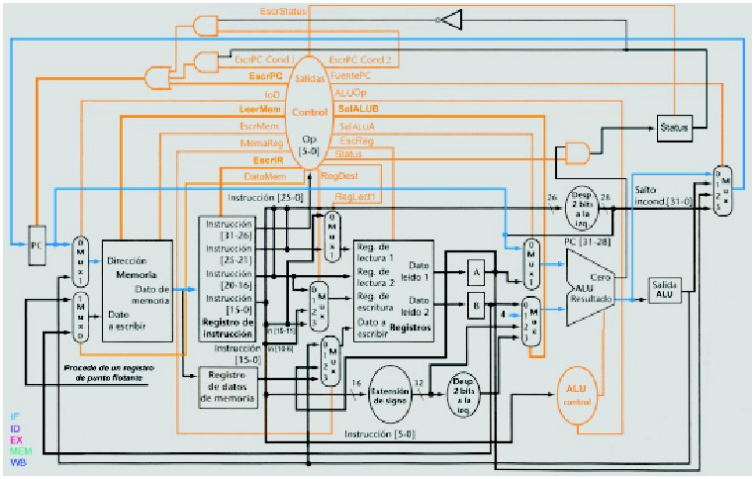



*Figura A.5:* Tercer modelo de camino de datos

La mayoría de las instrucciones de punto flotante usan un camino de datos que substituye la ALU por una FPU (Floating Point Unit) y los registros enteros por los de punto flotante. Este modelo incluye además la representación

del registro especial *Status* (ver figura A.4).

Por último, existe un modelo híbrido de los dos. Se trata de un camino de datos con una ALU y un banco de registros enteros, pero incluyendo el registro Status (con sus líneas de control correspondientes) y unas líneas que nos indican cuando leer o escribir en un registro de punto flotante. Este modelo (figura A.5) es necesario en instrucciones de punto flotante que operen con ambos tipos de registros, como pueden ser las instrucciones de carga o almacenamiento de punto flotante.

El funcionamiento de los botones de esta herramienta es el siguiente:

**Volver al editor.** En el caso de que se produzca algún error en ejecución será el único botón que permanecerá activado.

**Paso anterior.** Sólo aparece habilitado en la configuración multiciclo.

**Paso siguiente.** Sólo aparece habilitado en la configuración multiciclo.

**Ciclo anterior.** Retrocede un ciclo en la ejecución. Si se trata de una operación aritmética en punto flotante y se encuentra en la etapa de ejecución del simulador multiciclo, retrocederá el número de ciclos que se haya especificado en la configuración.

**Ciclo siguiente.** Avanza un ciclo en la ejecución. Si se trata de una operación aritmética en punto flotante y se encuentra en la etapa 2 del simulador multiciclo, avanzará el número de ciclos que se haya especificado en la configuración.

**Ejecutar.** Ejecuta la totalidad del código, a menos que se haya insertado un punto de ruptura en cuyo caso se ejecutará hasta ese punto.

**Breakpoint.** Si el botón de selección correspondiente está activado se despliega una lista con los PCs comprendidos entre el de la instrucción que en ese momento se está ejecutando y el último. La identificación de las instrucciones por medio de su PC correspondiente no resulta complicado, ya que en el segmento de texto ambos aparecen relacionados.

**Ciclos** Contador del número de ciclos que ha consumido la ejecución de las instrucciones hasta ese momento.

## A.8. Simulación con el procesador multiciclo

Un procesador multiciclo se caracteriza por subdividir las instrucciones en diferentes etapas, cada una de las cuales se ejecuta en un ciclo de reloj. Si la instrucción que se está ejecutando es una suma, resta, multiplicación o división en punto flotante, la etapa de ejecución ocupará tantos ciclos de reloj como se le especifique en la configuración.

El entorno gráfico asociado a esta simulación (ver figura A.6) es análogo al de una simulación monociclo, la diferencia más significativa es la activación de los botones *Paso siguiente* y *Paso anterior*.

**Paso anterior.** Sitúa la ejecución en el primer ciclo de la instrucción anterior.

**Paso siguiente.** Sitúa la ejecución en el último ciclo de la instrucción actual. Si ésta instrucción está en el último ciclo avanza hasta el último de la instrucción siguiente.

## A.9. Simulación con el procesador segmentado

Un procesador segmentado se caracteriza por la ejecución solapada de diferentes instrucciones.

### A.9.1. Segmentación básica

Es necesario destacar que:

- Los saltos se deciden en la segunda etapa de la segmentación, y su comportamiento va a ser diferente según cual fuese la técnica de salto escogida en la fase previa de configuración.

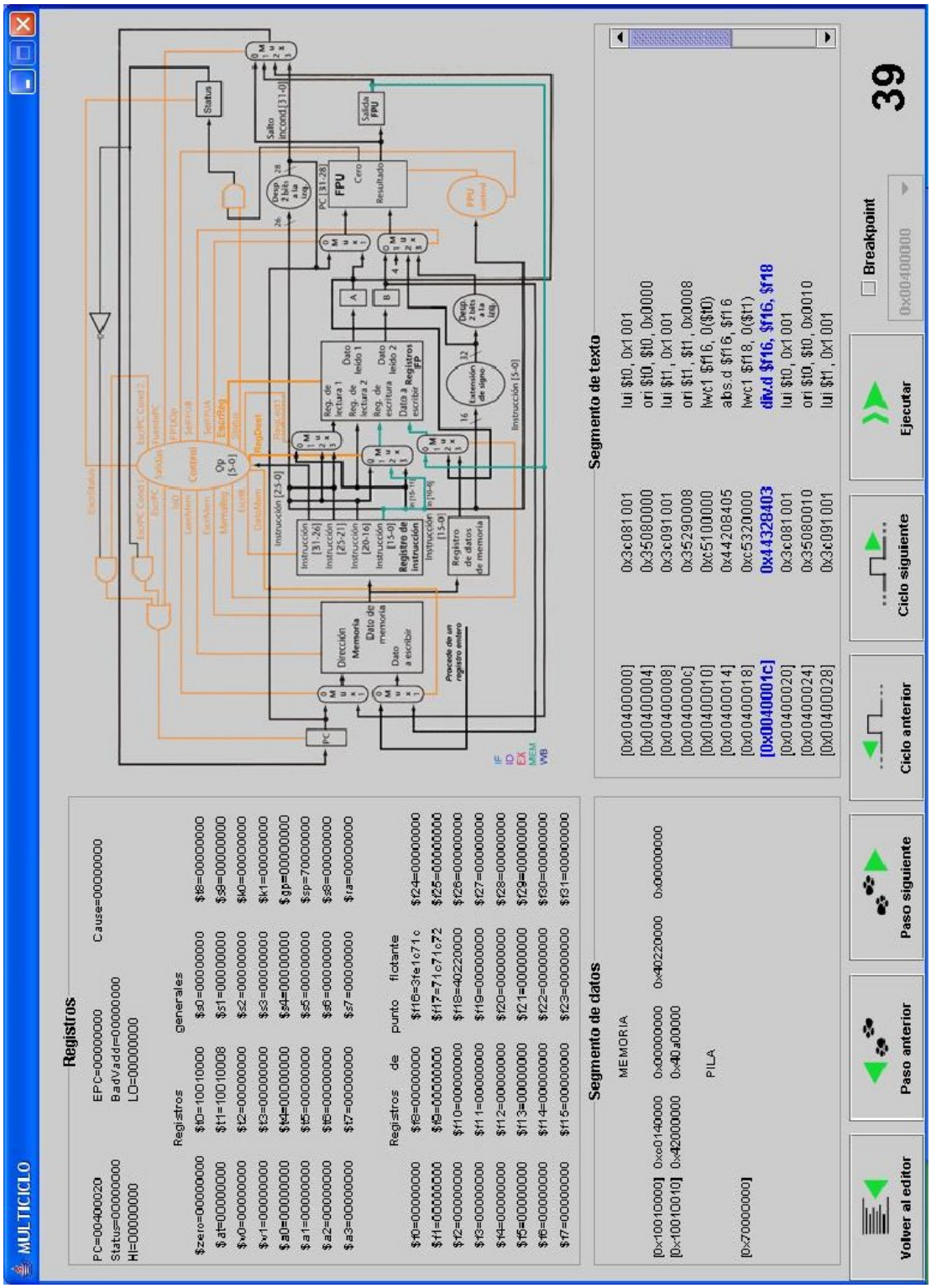



*Figura A.6:* Configuración multiciclo

  - Con la estrategia de *salto retardado* la instrucción que va a continuación del salto se ejecutará siempre, tanto si se realiza el salto como si no lo hace. En ocasiones será útil utilizar una instrucción *nop* después de instrucciones de salto para evitar que el programa funcione erróneamente.
  - La técnica de *salto fijo no efectivo* permite que la instrucción posterior al salto empiece su ejecución. Si el salto es no efectivo, no se toma, se continúa con la ejecución de dicha instrucción. Si es efectivo dicha instrucción es eliminada del cauce creando una burbuja.

- Para gestionar riesgos RAW el procesador usa dos técnicas:
  - *Anticipación*: permite adelantar un dato obtenido en las etapas EX o MEM en un ciclo $t$ a la etapa EX en el ciclo $t+1$.
  - *Detección*: se detiene la ejecución de la instrucción con riesgo en la etapa ID, insertando burbujas, hasta que éste se resuelve.
- Los riesgos WAW se gestionan con la *técnica de detección*. Si una instrucción va a alcanzar la etapa WB antes que otra instrucción previa que tiene el mismo registro destino, se detiene la primera en la última etapa de EX hasta que la anterior avance a la etapa MEM.
- Los riesgos de postescritura aparecen cuando dos instrucciones en punto flotante alcanzan la etapa WB en el mismo ciclo de reloj. Para gestionarlos, cuando dos o más instrucciones en punto flotante alcanzan la última etapa de ejecución en el mismo ciclo de reloj son detenidas, a excepción de aquélla que esté en la unidad funcional de mayor latencia.
- Antes de realizar una llamada al sistema el procesador espera a que el cauce se vacíe. Se ha considerado que la ejecución de una llamada al sistema consume un ciclo de reloj y no se contabilizan los ciclos que consume el SO al realizar el servicio requerido. Después de una llamada al sistema se sigue ejecutando el resto del código secuencialmente.

- Si la ejecución del programa en ensamblador no terminase con la llamada al sistema de finalización, sino porque en el segmento de texto no hay más instrucciones o porque se ha insertado un punto de ruptura, el cauce parará antes de terminar la ejecución de todas las instrucciones, por lo que no se ejecutará el programa completo.

- La segmentación se puede representar gráficamente mediante tres tipos de diagramas.

  - **Diagrama "camino de datos"**: muestra el estado del camino de datos durante un ciclo de reloj, identificando cada instrucción con etiquetas encima de la etapa correspondiente.

    Se usa este tipo de diagrama para mostrar con más detalle que está ocurriendo en el procesador durante cada ciclo del reloj, ver figura A.7.

  - **Diagrama monociclo**: muestra el estado general del cauce durante un ciclo de reloj, incluyendo tanto el procesador como el co-procesador de punto flotante. En los diagramas de este tipo se ve con más detalle que está ocurriendo en la segmentación durante cada ciclo de reloj. Para diferenciar las unidades funcionales segmentadas de las no segmentadas, las primeras se representan separadas por flechas, mientras que las últimas están todas unidas, ver figura A.8.

  - **Diagrama multiciclo** (figura A.9): se utiliza para dar una perspectiva general de diferentes situaciones dentro de la segmentación. Muestra la evolución de todas las instrucciones que están en ese momento en el cauce hasta el ciclo actual de ejecución. Se considera que el tiempo avanza de izquierda a derecha y las instrucciones se colocan siguiendo el orden de las etapas del cauce.

- El simulador dispone de botones que permiten ejecutar el programa escrito en ensamblador completo, o bien ciclo a ciclo. También se tiene

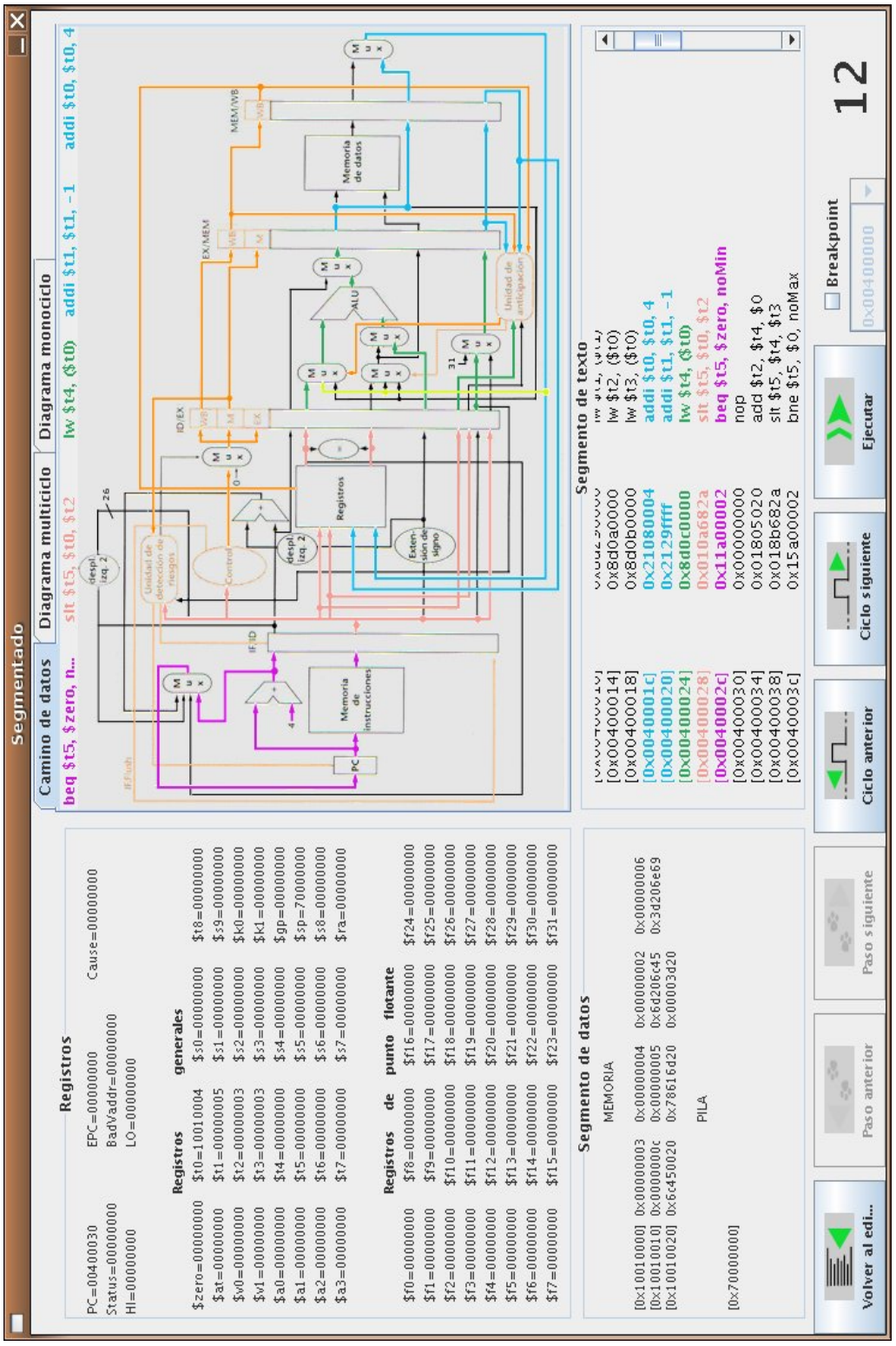



*Figura A.7:* Diagrama "camino de datos" para un solo ciclo de ejecución en procesador segmentado

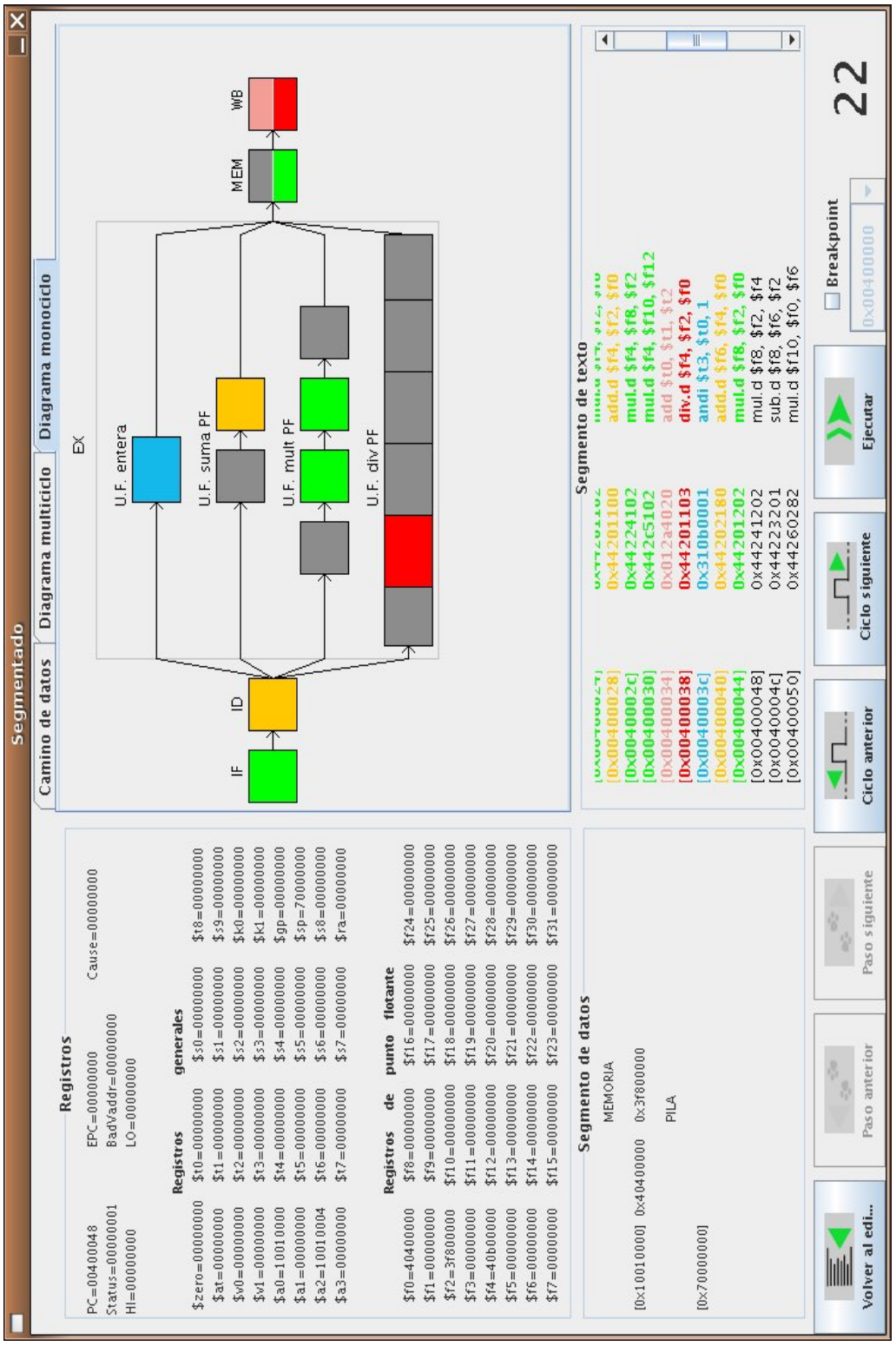



*Figura A.8:* Diagrama monociclo de la ejecución en procesador segmentado

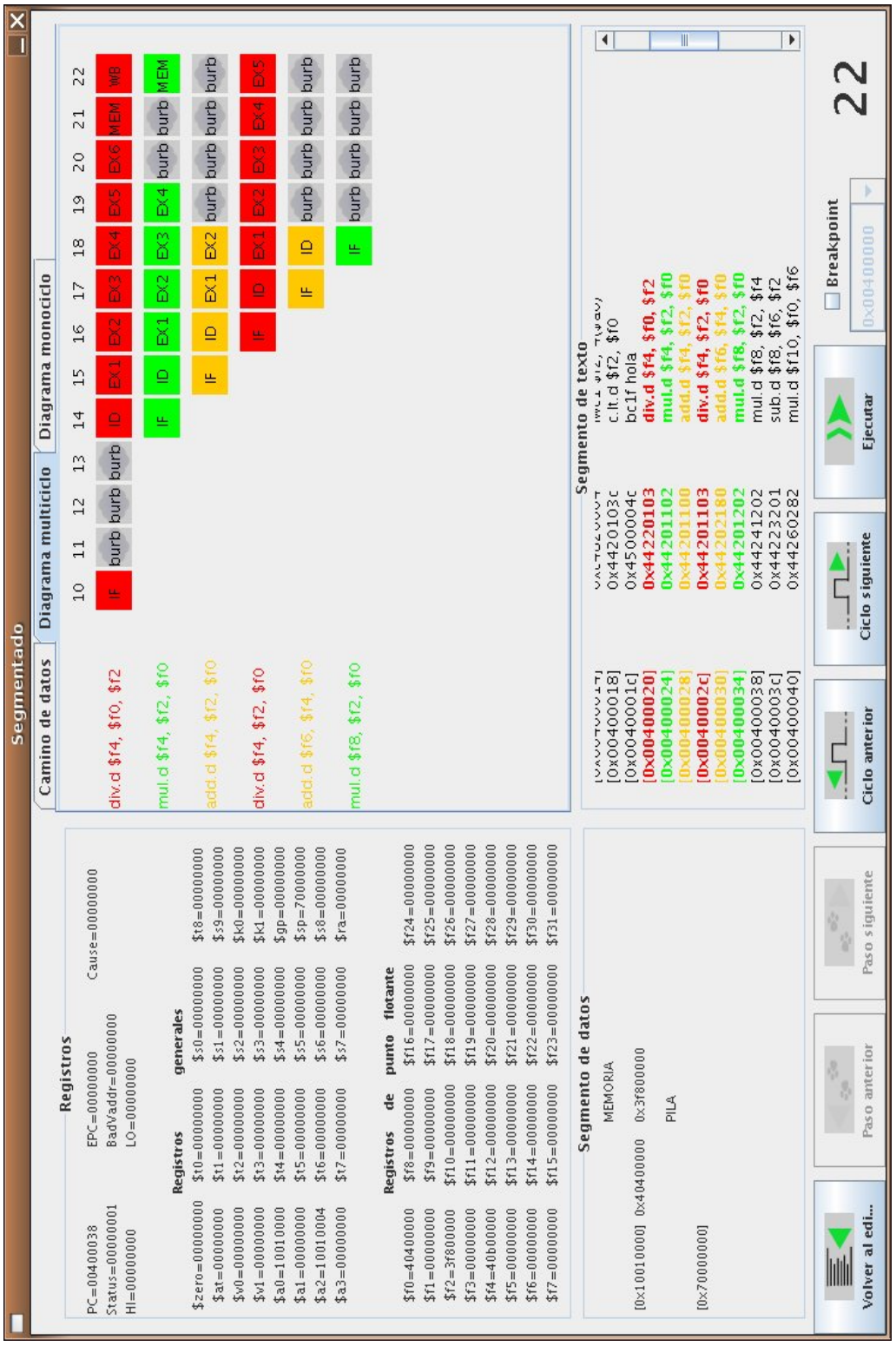



*Figura A.9:* Diagrama que muestra multiples ciclos en un procesador segmentado

la posibilidad de retroceder a ciclos anteriores. En cualquier momento de la ejecución se podrá poner un punto de ruptura.

### A.9.2. Simulación con planificación dinámica

Dentro de la configuración segmentada se puede escoger dos técnicas de planificación dinámica: las técnicas de Tomasulo y Marcador.

Es necesario destacar que:

- Las instrucciones aparecerán en la tabla de instrucciones desde la etapa de búsqueda.
- Los saltos enteros se deciden en la segunda etapa, pero los saltos en punto flotante se deciden en la etapa de ejecución correspondiente a cada una de las técnicas. La ejecución de un salto en punto flotante supone la detención de todo el procesador hasta que éste se haya resuelto, mientras que la ejecución de un salto entero sólo detendrá el co-procesador flotante.
- Aunque la ejecución de un programa en ensamblador no terminase con la correspondiente llamada al sistema, sino porque en el segmento de texto no hay más instrucciones o porque se ha insertado un punto de ruptura, el procesador segmentado terminará la ejecución de todas las instrucciones.
- Al finalizar la ejecución de estas técnicas se activará el botón de *informe*, que muestra un pequeño resumen con las características más relevantes de la ejecución.

#### Informe

Estas técnicas ofrecen la posibilidad de generar un informe (ver Figura A.10) que resume las características más relevantes de la ejecución de ese código.

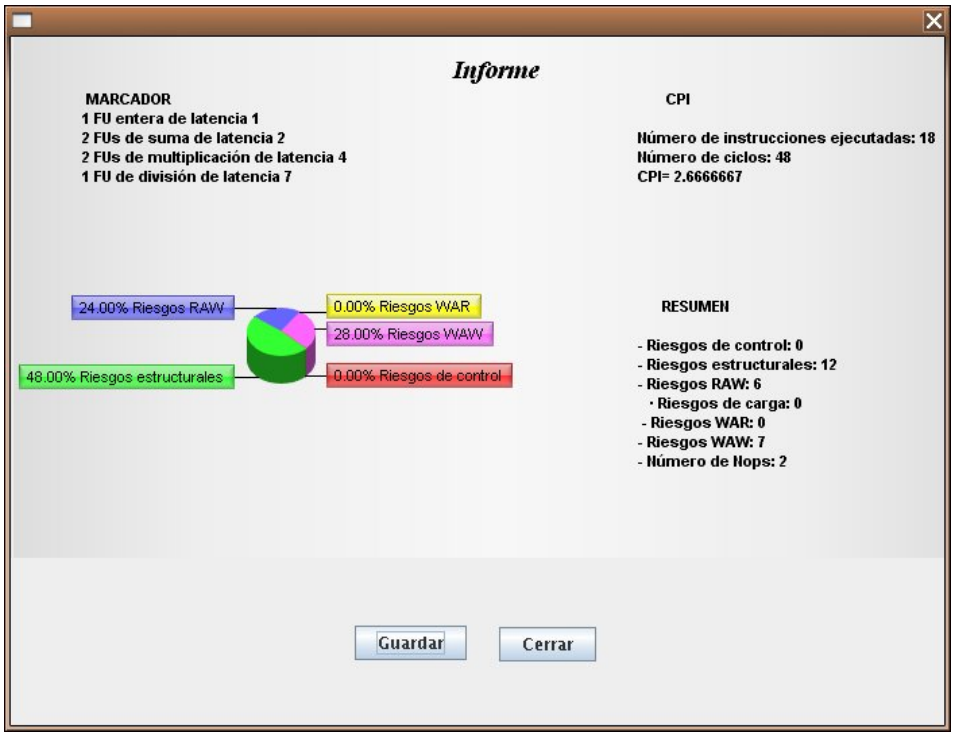



*Figura A.10:* Informe

Este informe se puede dividir en cuatro secciones. La primera de ellas se encuentra en la parte superior izquierda e indica las características hardware de la configuración escogida. En la esquina superior derecha aparecen el CPI. En la parte inferior izquierda un gráfico muestra un porcentaje de los riesgos detectados en la ejecución que aparecen más detallados a la derecha.

### Técnica de Marcador

La representación específica de la técnica de Marcador consta de tres tablas, como se puede ver en la Figura A.11. En la primera de ellas es la tabla de las instrucciones y se muestra la etapa en la que se encuentra actualmente cada una de las instrucciones.

La segunda tabla es la de las unidades funcionales. En ella se representa toda la información de las unidades funcionales que el usuario ha escogido.

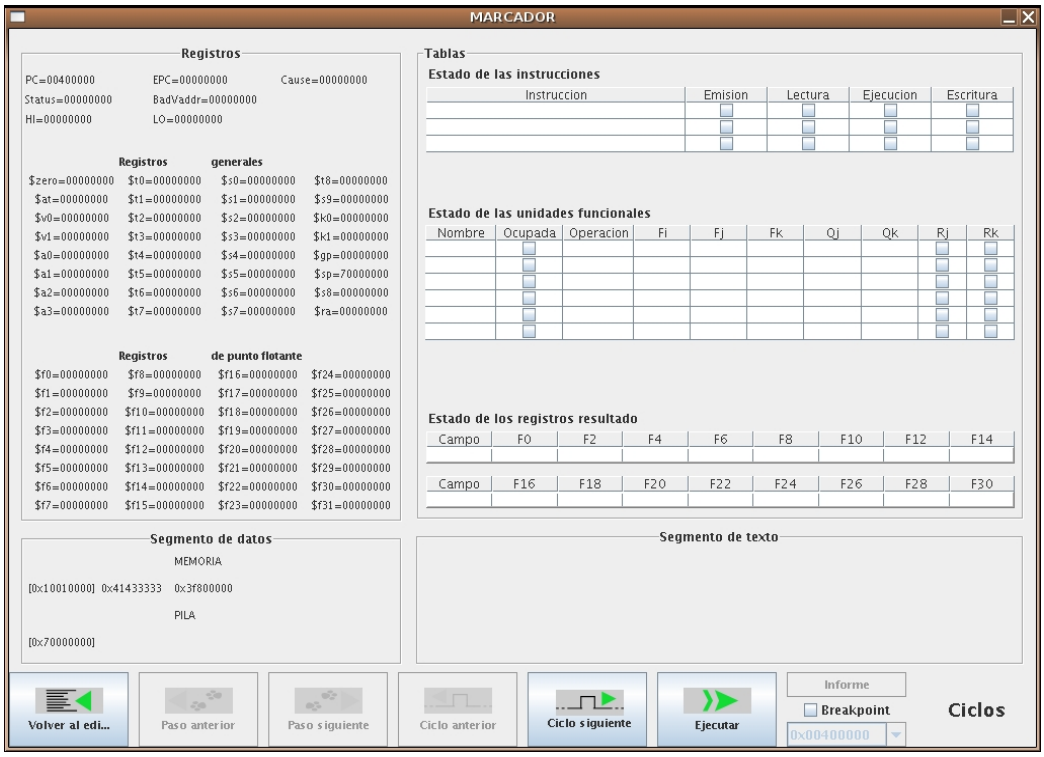



*Figura A.11:* Configuración Marcador

Esta información variará en función de si la unidad funcional está ocupada o no, y en caso afirmativo, también influirá la etapa en la que se encuentre la instrucción que ocupa la unidad.

La última tabla es una representación de los registros en punto flotante e indica si alguno de ellos va a ser escrito por una instrucción que se encuentre actualmente en ejecución.

## Técnica de Tomasulo

Al igual que la técnica de Marcador, ésta también se representa con tres tablas (ver Figura A.12). A pesar de que la implementación del algoritmo de Tomasulo no incluye la representación de la información en tablas, aquí se añade de todas formas debido a que la información que muestran sobre

el estado actual de cada una de las instrucciones, las estaciones de reserva y registros resultado es muy útil para seguir la ejecución de un código. La primera tabla es la tabla de instrucciones, similar a la que se usa en el caso del Marcador, con la excepcion de que en este caso hay sólo tres pasos en la ejecución.

La segunda tabla se corresponde con las estaciones de reserva. Aquí se incluyen también los buffers de carga y almacenamiento, ya que aunque su funcionamiento no es igual que el de las estaciones de reserva, es bastante similar.

Al igual que las unidades funcionales su estado variará en función de la instrucción que esté ocupándola. La principal diferencia con las unidades funcionales de Marcador, es que aquí sólo puede estar en ejecución una estación de reserva de cada tipo.

Las tablas de los registros indica cuales de ellos son el registro destino de alguna estación de reserva.

Por último, destacar que aunque en el informe asociado a esta técnica no aparecen nunca reflejados riesgos WAW, ya que sólo se producen cuando existen problemas de accesos ambiguos a memoria (varios registros diferentes apuntan a la misma dirección), estos si están controlados y se producirán detenciones en las estaciones de reservas correspondientes cuando se detecte alguno.

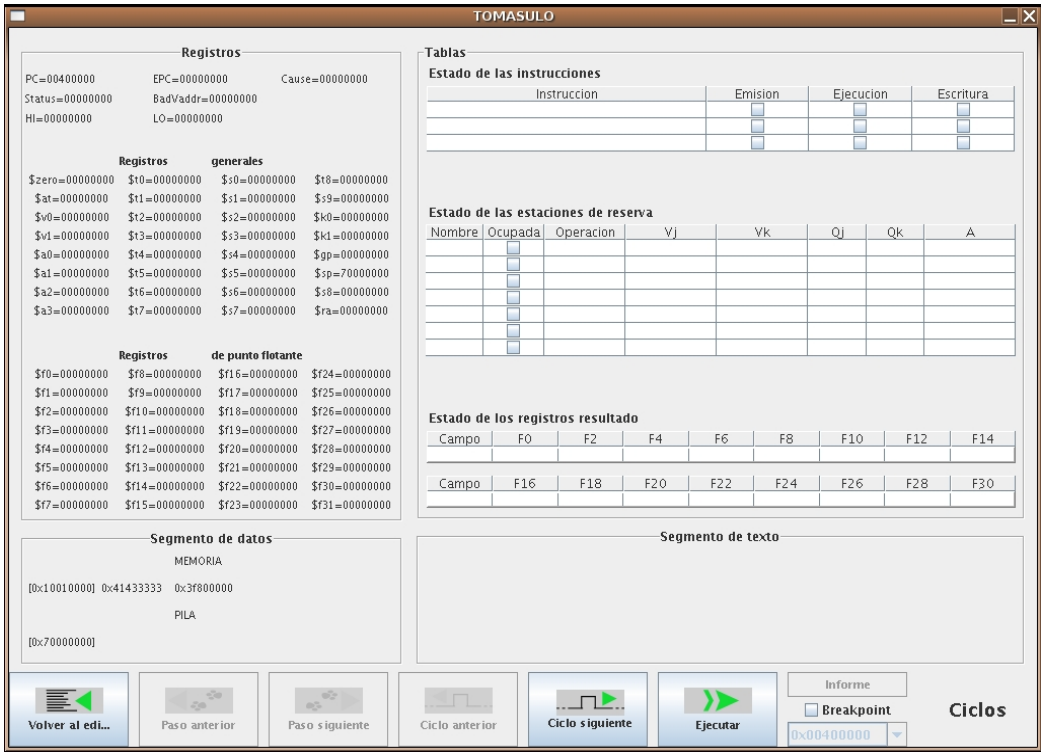



*Figura A.12:* Configuración Tomasulo