Universidad de La Laguna

Escuela Técnica Superior de Ingeniería Informática

SIMDE

Un Simulador para el Apoyo Docente en la Enseñanza de las Arquitecturas ILP con Planificación Dinámica y Estática

Memoria de Proyecto de Fin de Carrera de la Ingeniería Superior en Informática

Autor:

Iván Castilla Rodríguez

Directores:

Lorenzo Moreno Ruiz

José Francisco Sigut Saavedra

Carina Soledad González

Junio 2004

Lorenzo Moreno Ruiz, José Francisco Sigut Saavedra y Carina Soledad González, catedrático y profesores asociados respectivamente del Departamento de Física Fundamental y Experimental, Electrónica y Sistemas de la Universidad de La Laguna,

CERTIFICAN:

Que el Proyecto de Fin de Carrera de título "**SIMDE: Un Simulador para el Apoyo Docente en la Enseñanza de las Arquitecturas ILP con Planificación Dinámica y Estática**" presentado por D. **Iván Castilla Rodríguez** ha sido realizado bajo nuestra dirección.

Y para que así conste, firmamos la presente en La Laguna, a 8 de junio de 2004

Dr. Lorenzo Moreno Dr. José F. Sigut Dra. Carina González

Agradecimientos

En primer lugar debo dar las gracias a Lorenzo Moreno. De él partió la idea de este trabajo y su entusiasmo, apoyo y consejos han permitido lidiar con este "pequeño proyecto" que desbordó completamente nuestras expectativas iniciales.

También quiero agradecer a los miembros del Departamento que me han ayudado, en especial a los otros dos directores del proyecto que pusieron su empeño en seguir mis confusas explicaciones en las reuniones.

A nuestra "madrina" Coromoto. Sin ella no habría descubierto esa página de búsqueda de artículos. Su ayuda desinteresada cuando comencé a investigar los métodos de compilación VLIW se merece más que estas líneas.

No me voy a olvidar de todos mis amigos y compañeros que, buscándome un "huequito" en sus repletas agendas de informático, han aportado ideas o me han "reventado" el programa durante su desarrollo. David, Juanjo, Jesús… sin ustedes el programa no tendría tanto valor.

A los alumnos de Arquitectura e Ingeniería de Computadores de Quinto de la Ingeniería Superior en Informática de la ULL del curso 2003/2004. Lo que iban a ser unas "encuestitas" se convirtieron en sus manos en un hervidero de ideas que consiguieron que siga ilusionado con mejorar este programa.

A mis padres y a mis hermanos, que me han aguantado este tiempo hablándoles de unas extrañas máquinas y rumiando ideas incluso en las comidas. Yo no me habría aguantado…

Y por último, pero desde luego no menos importante, a Arelis. ¿Cuántas tardes-noches habré pasado discutiendo contigo la última idea o el último obstáculo que no me dejaba avanzar? Tus aportaciones a este proyecto, tanto las que han quedado escritas como aquellas que no se pueden valorar, son muchas más de las que piensas.

A los "chicos", por estar siempre ahí

ÍNDICE

Índice de ilustraciones

Índice de tablas

Índice de ejemplos

Índice de gráficas

INTRODUCCIÓN

SIMDE es un simulador de Arquitecturas que tratan de aprovechar el **Paralelismo a Nivel de Instrucción** (ILP) mediante técnicas tanto de **planificación dinámica** de instrucciones (principalmente basadas en mecanismos hardware), como de **planificación estática** de instrucciones (principalmente basadas en mecanismos software).

El objetivo de este simulador es servir de apoyo docente en la enseñanza de estos temas. El simulador está ideado para ser empleado como herramienta de la asignatura de **Arquitectura e Ingeniería de Computadores** de 5º curso de **Ingeniería Informática**, aunque su utilidad final puede ampliarse a cursos de doctorado o a cualquier actividad que precise una aproximación interactiva y pedagógica a este tipo de máquinas.

1.1. Los simuladores en Arquitectura de Computadores

La **Arquitectura de Computadores** es una materia imprescindible en la formación de alumnos de una Ingeniería Superior en Informática. Sin embargo, su enseñanza conlleva una serie de complicaciones que difícilmente pueden ser solventadas con las herramientas clásicas de docencia. Un aspecto clave es que los alumnos de la asignatura consigan trasladar esos conceptos teóricos acerca de unas máquinas que nunca han visto a una experiencia práctica con la que afiancen esos conocimientos.

Ante la imposibilidad material de disponer de las máquinas de las que se habla, una opción bastante atractiva es el uso de **simuladores**. Gracias a las mejoras del software existen una gran variedad de simuladores para casi cualquier aspecto de la asignatura como puede verse en la Tabla 1-1[1].

1.2. ILP como materia de Arquitectura de Computadores

Las arquitecturas ILP (*Instruction Level Parallelism*) ocupan un lugar destacado dentro de las materias de la asignatura. Como se verá más adelante, el ILP es la capacidad de las instrucciones de poder ejecutarse en paralelo; y son las arquitecturas ILP aquellas

[1] En la página http://www.sosresearch.org/caale/caalesimulators.html se puede encontrar una amplia variedad de referencias a simuladores existentes.

que emplean diversas técnicas para, aprovechando este paralelismo, mejorar el rendimiento de la máquina. La importancia de este tema se hace aún más notoria si se tiene en cuenta que es la base del diseño de la mayoría de los equipos de sobremesa más familiares para el alumnado, como son la mayoría de PCs de AMD e Intel.

Simulador	Descripción
Dinero IV: Trace-Driven Uniprocessor Cache Simulator http://www.cs.wisc.edu/~markhill/DineroIV/ (Univ. of Wisconsin Computer Sciences)	Simulador de caches para trazas de referencias a memoria.
WinMIPS64 http://www.computing.dcu.ie/~mike/winmips64.html (Simulador para el [13])	Simulador del repertorio de instrucciones MIPS64, diseñado para sustituir al WinDLX
WinDLXV (Universidad Politécnica de Valencia)	Simulador de repertorio de instrucciones vectorial basado en [13]
MIDAS http://icaro.eii.us.es/descargas/R10kSim.zip (Institut für Technische Informatik, Vienna Technical University)	Simulador de arquitectura superescalar.
SATSim http://ece.gatech.edu/research/pica/SATSim/satsim.html (Wolf y Wills, Georgia Institute of Technology)	Simulador de arquitectura superescalar que emplea ficheros de trazas.

Tabla 1-1. Algunos simuladores relacionados con la asignatura de Arquitectura de Computadores

Aunque no es un requisito indispensable, es difícil hablar de ILP sin hablar de la iniciación de múltiples instrucciones por ciclo de reloj. Es de este tipo de arquitecturas ILP en las que se hace mayor hincapié en este documento. Para caracterizarlas, puede hacerse una clasificación simple pero bastante práctica sólo teniendo en cuenta qué parte de la máquina realiza la mayoría del trabajo en la búsqueda y aprovechamiento del paralelismo a nivel de instrucción: el hardware o el software.

- Se habla de **Planificación Dinámica** cuando es el hardware el que hace la mayoría del trabajo. Las instrucciones se emiten en el mismo orden en que están en el programa secuencial original, pero luego el hardware se encarga de ejecutar las instrucciones fuera de orden (*out-of-order*), según van estando disponibles los operandos y los recursos. Al final el mismo hardware se encarga de asegurar que el resultado del programa sea el mismo que si se hubiese respetado el orden original de ejecución. Los procesadores **superescalares** suelen emplear este tipo de planificación (aunque también pueden encontrarse casos de superescalares con planificación estática, como el UltraSPARC-III de Sun [19], o el Alpha 21164 de DEC).
- La **Planificación Estática**, por el contrario, delega las decisiones en el software, más concretamente en el compilador. De esta manera el hardware de la máquina se simplifica enormemente. Los procesadores **VLIW** (*Very Long Instruction Word*) son un claro ejemplo de esta vertiente del ILP.

Cada una de las dos formas de planificar tiene sus ventajas y desventajas, y un grupo de adeptos bastante importante. Actualmente la tendencia es aprovechar lo mejor de los dos enfoques, haciendo uso del trabajo del compilador, pero delegando en el hardware la adaptación a las situaciones cambiantes (arquitecturas **EPIC**, como la IA-64 de Intel [15]).

1.3. Justificación del trabajo

En la Tabla 1-1 se nombraron 2 simuladores superescalares y ninguno VLIW. En realidad existe una gran variedad de simuladores de máquinas superescalares, enfocados a casi cualquier ámbito: experimentación, docencia… Sin embargo, el trabajo en simuladores VLIW se restringe habitualmente a los simuladores de procesadores comerciales, poco útiles en la docencia debido a su excesiva especificidad.

¿Por qué esta nula aportación de los simuladores VLIW al mundo de la docencia? Una de las características más destacables de los procesadores VLIW es que el **compilador suele ser tremendamente complejo**, por lo que escribir en código ensamblador directamente es inabordable para un programador. Por tanto, diseñar un simulador que lo único que va a hacer es tomar un código secuencial, compilarlo y hacerlo funcionar sobre una máquina de hardware sencillísimo no parece tener un interés excesivo para un alumno. Este interés sí podría surgir en un investigador que necesita saber a priori cómo se va a comportar su programa sobre una máquina concreta; de ahí que los simuladores comerciales orientados a este fin se empleen con mayor asiduidad.

Tras exponer esto, ¿dónde está entonces el interés de diseñar un simulador VLIW para apoyar a la docencia? En primer lugar hay que decidir qué es lo que se quiere enseñar al alumno. Ya que el hardware es tan sencillo (al menos en los esquemas VLIW más puros) y la planificación estática, no parece merecer la pena ver una traza de un programa en la máquina, puesto que probablemente siempre se obtenga el mismo resultado. La clave está en la **construcción de los programas** y las **técnicas** que se emplean para realizar una planificación estática. En definitiva, se quiere que sea el alumno el que haga el papel del compilador y optimice el código secuencial y construya las instrucciones largas que empleará la máquina, empleando para ello las técnicas que se ven en clase (desenrollado de bucles, *software pipelining*...). No obstante, se ha dicho anteriormente que el uso del lenguaje ensamblador por un programador es inabordable… Esta afirmación es cierta para una máquina real de propósito general en la que se quiera ejecutar cualquier programa. Sin embargo, si se emplea un subconjunto de las instrucciones de la máquina en programas de tamaño reducido y un esquema del hardware lo suficientemente claro, este problema no resulta tan inabordable y puede convertirse en una experiencia de aprendizaje incluso grata para el alumno.

Queda por tanto justificada la decisión de diseñar un simulador VLIW basándose en la novedad del planteamiento, ya que viene a ocupar un espacio para el que no hay (al menos conocidos por el autor) ningún simulador semejante. Pero todavía puede añadirse un nivel más a esta experiencia: ¿por qué conformarse con un simulador VLIW cuando puede emplearse la misma base para diseñar también un simulador superescalar? En este terreno hay muchos trabajos realizados, pero la innovación es el **enfoque conjunto** de las dos arquitecturas. Empleando una **base hardware común** (unidades funcionales, registros, memoria, repertorio de instrucciones...) puede diseñarse una herramienta que permita ver desde las dos perspectivas, dinámica y estática, el mismo problema: el ILP. De esta manera puede incidirse no sólo en las diferencias entre los dos enfoques, sino también en las semejanzas. Y sobre todo pueden **compararse** y así tener una experiencia más rica en contenidos y una visión más global de esta parte de la asignatura.

1.4. Resumen del documento

En este documento se retomarán algunos de los temas ya nombrados en esta introducción, junto con toda la información necesaria para comprender los entresijos de

la herramienta diseñada. Se comienza con un amplio repaso a la teoría de Arquitectura de Computadores, y en especial a toda la parte de ILP. Estos conceptos sirven de base a la disertación posterior acerca de las decisiones de diseño de las máquinas del simulador. El capítulo 5 incluye un pequeño manual de usuario que desglosa las funcionalidades del simulador. Finalmente, se incluyen los resultados de una validación de la herramienta realizada mediante una encuesta entre los alumnos de Arquitectura e Ingeniería de Computadores de 5º curso de Ingeniería Informática.

1.5. Disponibilidad del simulador

El resultado de este proyecto es un simulador que funciona sobre cualquier plataforma Windows. Puede obtenerse desde la web en la página:

http://www.cyc.ull.es/simde/

O descargarse directamente vía ftp desde:

ftp://ftp.etsii.ull.es/asignas/ARQUIT/SIMDE/SIMDE%20v1.0.zip

FUNDAMENTOS TEÓRICOS

En este capítulo se verá un repaso de todos aquellos conceptos teóricos que son, en muchos casos, imprescindibles para comprender los fundamentos de la herramienta realizada. Todo el capítulo está principalmente basado en dos textos: [13] y en menor medida [23], y es básicamente un resumen de los mismos.

2.1. Definición de Arquitectura de Computadores

Pese a ser un término básico en el mundo de la computación, no existe un criterio fijo entre los autores a la hora de ponerse de acuerdo en delimitar qué es Arquitectura de Computadores y qué no lo es.

Si nos fijamos en [13] se toma el término como la unión de tres aspectos bien diferenciados: el **repertorio de instrucciones**, la **organización** de la máquina y su **hardware**.

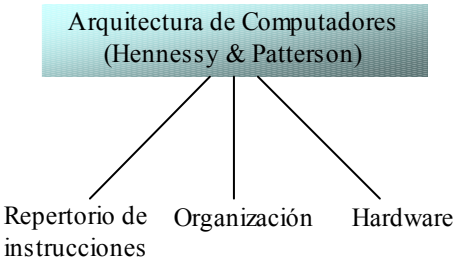


Ilustración 2-1. Definición de Arquitectura de Computadores [13]

El término *arquitectura del repertorio de instrucciones* se entiende como la parte del repertorio visible por el programador, que se convierte en el nexo entre el hardware y el software de la máquina.

El término *organización* engloba aquellos elementos de más alto nivel en el diseño de la máquina como la jerarquía de memoria, la estructura del bus y el diseño interno de la CPU.

El *hardware* denota la tecnología de la máquina: el diseño lógico, la estructura interna de cada componente...

Otra visión distinta la da [23], que primero habla de un nivel de abstracción superior: el **modelo de computación**, que engloba la propia arquitectura y el lenguaje de la computadora. El **lenguaje de computación** se interpreta como una herramienta para formular una tarea computacional dentro de un modelo de computación; mientras la **arquitectura** se ve como la herramienta para implementar este modelo de computación. El concepto se extiende según el punto de vista que se quiera aplicar. Así se habla de **arquitectura abstracta** cuando se quiere ver la especificación funcional de la máquina (como una caja negra), mientras que se habla de **arquitectura concreta** al referirse a la implementación o una descripción de la estructura de la máquina.

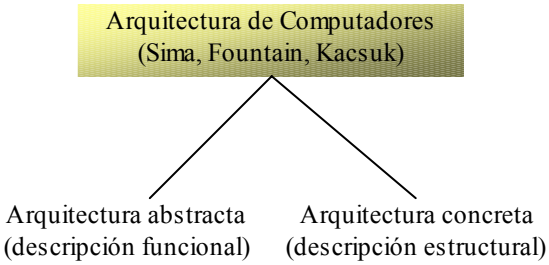


Ilustración 2-2. Definición de Arquitectura de Computadores [23]

En este documento se emplea una nomenclatura más próxima a la empleada en [13], aunque el *hardware* se denota en ocasiones como **estructura** o **implementación** de la máquina, y se emplea el término **arquitectura** para denominar toda la parte funcional de la máquina, incluyendo *repertorio de instrucciones* y *organización*. Este uso del término "arquitectura" pretende destacar el trabajo del "arquitecto" de computadores que, independientemente de la tecnología existente, debe buscar mecanismos y técnicas que permitan optimizar el rendimiento de las máquinas que diseña.

2.2. Rendimiento del procesador

La Arquitectura de Computadores tiene varios objetivos finales, como son la mejora de los costes, potencia o rendimiento de las máquinas que diseñan. Pese a que el resto de criterios (especialmente el coste) son muy importantes, las explicaciones posteriores se centrarán en las medidas del rendimiento del procesador.

2.2.1. Ley de Amdahl

El principio fundamental que se aplica para mejorar el rendimiento de un procesador es hacer el **caso común más rápido**. Esto significa incidir en la eficiencia de los eventos que con más frecuencia se producen en la máquina aun perdiendo rendimiento en los casos menos frecuentes. La pérdida de eficiencia en los casos menos frecuentes es aceptable siempre que la mejora del caso común sea lo suficientemente buena. La manera de cuantificar este principio es emplear la **Ley de Amdahl**.

La Ley de Amdahl permite calcular la **aceleración** (*speedup*) obtenida al mejorar una parte de un procesador, y viene a concluir que la máxima aceleración que se puede obtener está limitada por la fracción de tiempo que puede usarse la parte mejorada. La aceleración se define como sigue:

$$Speedup = \frac{Rendimiento\ de\ la\ tarea\ usando\ la\ mejora\ donde\ sea\ posible}{Rendimiento\ de\ la\ tarea\ sin\ \ usar\ la\ mejora}$$

O también de esta forma:

$$Speedup = \frac{Tiempo\ de\ ejecución\ de\ la\ tarea\ sin\ usar\ la\ mejora}{Tiempo\ de\ ejecución\ de\ la\ tarea\ usando\ la\ mejora\ donde\ sea\ posible}$$

La aceleración obtenida a partir de una mejora está determinada por dos factores:

- La **fracción** del tiempo de cómputo de la máquina original que puede ser usada la mejora. Por ejemplo: si el tiempo total de un programa eran 60 segundos, y una mejora puede aplicarse durante 20 segundos de este tiempo, se tendrá una fracción (F) de 20/60. Este valor es siempre menor o igual que la unidad.
- La **ganancia** obtenida al aplicar el modo de ejecución mejorado; es decir, cuánto más rápido se ejecutaría una tarea si la mejora pudiese aplicarse a todo el programa. Por ejemplo: si una parte de un programa que se ejecuta totalmente empleando una mejora tarda 2 segundos, y originalmente (sin la mejora) tardaba 5 segundos, la ganancia (G) será de 5/2. Este valor siempre es mayor que 1.

El cálculo del tiempo de ejecución de un programa puede hacerse sumando el tiempo que tardan las partes del programa que hacen uso de la porción mejorada de la máquina con las partes que emplean la máquina no mejorada.

$$Tiempo\ ejecución_{nuevo} = Tiempo\ ejecución_{viejo} \times \left((1 - F) + \frac{F}{G} \right)$$

Lo que permite obtener una expresión para la aceleración global de la máquina aplicando una mejora:

$$Speedup_{global} = \frac{Tiempo\ ejecución_{viejo}}{Tiempo\ ejecución_{nuevo}} = \frac{1}{(1 - F) + \frac{F}{G}}$$

2.2.2. Tiempo CPU

Existen muchas medidas del rendimiento de un procesador. Una de las medidas más importantes, y que casi puede considerarse la ecuación fundamental de la Arquitectura de Computadores, es la siguiente:

$$Tiempo\ CPU = \frac{tiempo}{programa} = \frac{instrucciones}{programa} \times \frac{ciclos\ de\ reloj}{instrucción} \times \frac{tiempo}{ciclo\ de\ reloj}$$

Cada uno de los factores de la ecuación repercute directamente en el rendimiento final de la máquina. Lamentablemente, los factores no son independientes, ya que la modificación de alguno de ellos suele repercutir en el resto. Sin embargo, puede observarse cuál es el aspecto que influye más en cada factor:

- El tiempo por ciclo de reloj es un parámetro fuertemente tecnológico, y depende principalmente del hardware de la máquina, aunque también de su organización.
- Los ciclos de reloj por instrucción tienen que ver con la organización de la máquina pero también con el repertorio de instrucciones.
- El número de instrucciones de un programa está determinado por el repertorio de instrucciones de la máquina y por el diseño del compilador, así como por la organización de la máquina.

El tiempo por ciclo de reloj puede mejorarse con mejoras tecnológicas: memorias de acceso más rápido, unidades funcionales optimizadas…

Para disminuir el número de instrucciones de un programa una de las opciones que tuvo más popularidad fue usar repertorios con instrucciones complejas (*CISC: Complex Instruction Set Computer*). De esta forma, se tenían instrucciones muy específicas que hacían tareas complejas pero de forma optimizada (como operaciones matriciales, raíces cuadradas…). Esta opción ha ido perdiendo popularidad, y actualmente sólo suele encontrarse como pequeños subconjuntos de instrucciones para tareas específicas (como el repertorio *MMX* para operaciones multimedia en los Pentium de Intel).

De estos tres factores, sin embargo, hay que destacar el segundo, normalmente denominado **CPI** (ciclos por instrucción). Este valor se calcula con la siguiente expresión:

$$CPI = CPI\ ideal + \text{“burbujas}^{2}\text{”}\ estructurales + \text{“burbujas” por datos} + \\ + \text{“burbujas” de control}$$

El CPI es la inversa del IPC (instrucciones por ciclo), que es el parámetro fundamental que tratan de incrementar las técnicas que trabajan con el paralelismo a nivel de instrucción (ILP, ver sección 2.4). Este incremento se consigue reduciendo cualquiera de los tres términos que “atascan” el flujo de ejecución de la máquina, que se corresponden con los tres tipos posibles de dependencias entre instrucciones que se verán en 2.4.1. En la Tabla 2-1 puede verse un resumen de algunas de las técnicas más usadas para reducir el CPI, junto con el término de la ecuación afectado y el tipo de ILP en el que se puede encuadrar la técnica.

Técnica	Termino reducido	Tipo de ILP
Adelantamiento (*forwarding*)	Dependencias de datos	Segmentación
Salto retardado y planificación básica del salto	Dependencias de control	Segmentación
Scoreboarding (planificación dinámica básica)	Dependencias de datos (RAW)	Segmentación
Planificación dinámica con *renaming*	Dependencias de datos	Superescalares
Predicción dinámica de salto	Dependencias de control	Superescalares
Emisión de múltiples instrucciones por ciclo	CPI ideal	Superescalares/VLIW
Especulación	Dependencias de datos y de control	Superescalares
Desambiguación dinámica de memoria	Dependencias de datos (memoria)	Superescalares
Desenrollado de bucles	Dependencias de control	VLIW
Planificación básica del *pipeline* con el compilador	Dependencias de datos	VLIW
Análisis de dependencias con el compilador	CPI ideal, dependencias de datos	VLIW
Software pipelining, *trace scheduling*	CPI ideal, dependencias de datos	VLIW
Especulación con el compilador	CPI ideal, dependencias de datos y de control	VLIW

Tabla 2-1. Principales técnicas empleadas para reducir el CPI. Fuente [13]

La mayoría de estas técnicas se explicarán a lo largo de este documento.

[2] *stall*

2.3. Elementos básicos de un procesador

Antes de entrar en el detalle de las arquitecturas ILP (que es realmente el tema en el que se quiere profundizar), conviene definir una serie de conceptos y describir algunos elementos más básicos que pueden ser de utilidad.

2.3.1. Repertorio de instrucciones

En todo este documento se emplea la arquitectura **RISC**[3] (*Reduced Instruction Set Computer*) para ilustrar los conceptos más básicos. En concreto se va a trabajar con una arquitectura MIPS[4], aunque el nivel de detalle al que se trata hace innecesaria la distinción en el texto. Las arquitecturas RISC pueden caracterizarse a grandes rasgos de la siguiente forma:

- Todas las operaciones de datos se aplican a los registros y normalmente afectan al registro completo.
- La única forma de acceder a la memoria es mediante operaciones de carga (*LOAD*) y almacenamiento (*STORE*), que mueven datos de memoria a un registro y de un registro a memoria respectivamente.
- Los formatos de instrucción suelen ser pocos y de un tamaño fijo.

Estas propiedades permiten simplificar mucho el diseño del hardware de la máquina.

Las instrucciones de este tipo de arquitecturas pueden clasificarse según tres categorías:

- **Instrucciones de ALU**: Estas instrucciones emplean dos registros como operandos fuente, o bien un registro y un valor inmediato; las operan y almacenan el resultado en otro registro.

```
DADDUI R3, R0, #4
// Suma entera inmediata
ADDI  R3, R2, R1
// Suma entera entre dos registros
```

Ejemplo 2-1. Instrucciones de la ALU

- **Instrucciones de acceso a memoria**: Estas instrucciones toman un registro, el *registro base*, y un valor inmediato, el *offset*, y los suman para obtener una dirección de memoria, la *dirección efectiva*. En el caso de un *LOAD* se toma un segundo registro como destino del valor almacenado en esa posición de memoria; si se trata de un *STORE* es el valor contenido en el registro el que se almacena en esa dirección de memoria.

[3] En el otro extremo se encuentran las arquitecturas CISC (*Complex Instruction Set Computer*), que no son el objetivo de este trabajo.

[4] Para estudiar en más detalle la arquitectura MIPS IV puede descargarse desde la página de MIPS Technologies: www.mips.com registrándose previamente. También puede descargarse directamente el repertorio de instrucciones de la dirección www-2.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15740-f97/public/doc/mips-isa.pdf.

```
LD F0, 4(R1)
// Carga de un valor de punto flotante doble precisión en un registro
SI R3, 8(R1)
// Almacenamiento de un valor entero en memoria
```

Ejemplo 2-2. Instrucciones de acceso a memoria

- **Saltos condicionales** (*branches*) **e incondicionales** (*jump*): Los saltos condicionales suelen emplear un conjunto de bits para chequear la condición o bien una comparación entre dos registros. En este trabajo sólo se emplean saltos condicionales que realizan comparación entre registros. La dirección destino del salto suele calcularse sumando un valor inmediato contenido en la propia instrucción al contador de programa (PC).

```
BNE R1, R2, LOOP
// Salto condicional
J LOOP
// Salto incondicional
```

Ejemplo 2-3. Instrucciones de salto

2.3.2. Implementación simple del repertorio RISC

Para entender mejor el funcionamiento básico del repertorio de instrucciones RISC se presenta una implementación sencilla dividida en etapas:

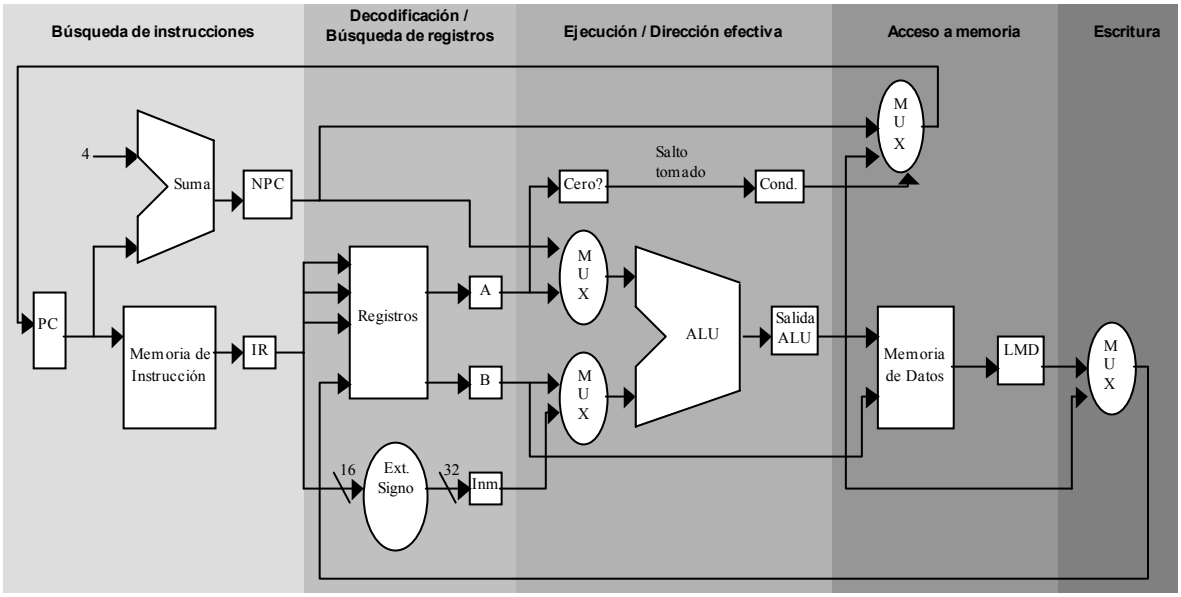


Ilustración 2-3. Esquema de una máquina RISC. Fuente Pág. A-29 [13]

1. Búsqueda de la instrucción (*Instruction fetch*: **IF**): Usando el PC se busca la instrucción actual en memoria y se actualiza el PC para poder buscar la siguiente.
2. Decodificación de la instrucción / Búsqueda de registros (*Instruction decode / register fetch*: **ID**): Se decodifica la instrucción y se buscan los registros fuente en el banco de registros.

3. Ejecución / Dirección efectiva (*Execution / efective address*: **EX**): si se trata de una instrucción de memoria se calcula la dirección efectiva; si es una instrucción de ALU se opera para obtener el resultado.
4. Acceso a memoria (*Memory access*: **MEM**): Si la instrucción es un LOAD se lee la memoria; si es un STORE, se escribe en ella.
5. Escritura (*Write-back*: **WB**): Se escribe el resultado (de un LOAD o de una operación de la ALU) en el registro correspondiente.

Un esquema de una implementación hardware de una máquina RISC puede verse en la Ilustración 2-3. En este esquema se observan las 5 etapas nombradas y los elementos más importantes en cada una de ellas.

2.4. ILP

El **paralelismo a nivel de instrucción** es la capacidad potencial de un grupo de instrucciones de poder ser ejecutadas en paralelo. Para poder ejecutar un conjunto de instrucciones en paralelo se requiere:

- Determinar las relaciones de dependencia entre estas instrucciones.
- Adecuar los recursos hardware a la ejecución de varias instrucciones en paralelo.
- Diseñar estrategias que permitan determinar cuando una instrucción está lista para ejecutarse.
- Diseñar técnicas para pasar valores de una a otra instrucción.

Como ya se nombró anteriormente, para hacer uso del ILP se debe intentar aumentar el número de instrucciones por ciclo de reloj que puede ejecutar la máquina (IPC).

El primer acercamiento a la explotación del ILP es la **segmentación** (*pipelining*). La segmentación funciona como una cadena de montaje, dividiendo el procesamiento de una instrucción en varias etapas. De esta manera, una instrucción que completa una etapa puede pasar a la siguiente, y una nueva instrucción puede ocupar la etapa recién abandonada. Se observa rápidamente que pueden ejecutarse tantas instrucciones en paralelo como etapas se hayan definido. Normalmente se definen de 2 a 10 etapas dependiendo del tipo de unidad funcional de la que se trate. En el caso del esquema de ejecución explicado en la sección 2.3.2 se puede hacer una etapa por cada paso que se nombraba allí, como se ve en la Tabla 2-2. Esto se conseguiría añadiendo un buffer (*latch*) entre cada una de las etapas.

	Ciclo de reloj								
Nº instrucción	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
instrucción i	IF	ID	EX	MEM	WB				
instrucción i + 1		IF	ID	EX	MEM	WB			
instrucción i + 2			IF	ID	EX	MEM	WB		
instrucción i + 3				IF	ID	EX	MEM	WB	
instrucción i + 4					IF	ID	EX	MEM	WB

Tabla 2-2. Esquema simple de un *pipeline* aplicado a una máquina RISC

Cuando se habla de procesadores segmentados sin más, debe tenerse en cuenta que cada ciclo se está emitiendo como máximo una instrucción, ya que se está empleando un único flujo de ejecución. La evolución lógica de este diseño es el empleo de varias unidades funcionales segmentadas que ejecutan varias instrucciones simultáneamente. Para mantener ocupadas a estas unidades funcionales es necesaria la emisión de más de

una instrucción por ciclo (idealmente tantas instrucciones como unidades funcionales haya). Las 5 aproximaciones al uso de la **emisión múltiple** se resumen en la Tabla 2-3.

Nombre común	Estructura de la emisión	Detección de riesgos	Planificación	Característica distintiva	Ejemplos
Superescalar (estática)	dinámica	hardware	estática	ejecución en orden	Sun UltraSPARC II/III
Superescalar (dinámica)	dinámica	hardware	dinámica	ejecución fuera de orden (limitada)	IBM Power2
Superescalar (especulativa)	dinámica	hardware	dinámica	ejecución fuera de orden con especulación	Pentium III/4, MIPS R10K, Alpha 21264, HP PA 8500, IBM RS64III
VLIW/LIW	estática	software	estática	No hay riesgos entre los paquetes emitidos	Trimedia, Trace 7/200
EPIC	mayormente estática	mayormente software	mayormente estática	Dependencias explícitas marcadas por el compilador	Itanium

Tabla 2-3. Las cinco aproximaciones principales a los procesadores con múltiple emisión de instrucciones. Fuente: [13]

En la Tabla 2-3 se observa que los cinco grupos pueden verse reducidos fácilmente a dos: máquinas **superescalares** por un lado y **VLIW/EPIC** por otro. Esta división puede hacerse atendiendo a dos motivos principales: el tipo de emisión y el dominio del componente hardware o software en cada una de las vertientes. La razón es clara: una emisión dinámica de las instrucciones no permite conocer a priori los riesgos entre las instrucciones, lo que obliga a detectarlos de forma dinámica mediante dispositivos hardware. La emisión estática sólo es posible habiendo detectado los riesgos mediante software (al menos en su mayor parte).

2.4.1. Dependencias entre instrucciones

Se denomina **orden del programa** (*program order*) a la ordenación original de las instrucciones en el código fuente. Partiendo de este término se distingue entre dos tipos de ejecución: la que sigue fielmente el orden del programa o **ejecución en orden** (*in-order*); y la que puede modificar este orden o **ejecución fuera de orden** (*out-of-order*), que podría llevar a la obtención de resultados incorrectos. Las técnicas hardware y software que pretendan aprovechar el paralelismo entre las instrucciones deben preservar el orden de ejecución del programa siempre que éste afecte al resultado final.

Para poder ejecutar dos instrucciones en paralelo, y por tanto romper el orden del programa, lo primero a tener en cuenta es que no existan dependencias[5] entre ellas. Existen tres tipos de dependencias posibles entre instrucciones:

Dependencias estructurales

Ocurre si dos instrucciones necesitan el mismo recurso para ser ejecutadas. Las dependencias estructurales tienen fácil solución: redundancia de hardware. Sin

[5] Aunque se emplee el término **dependencia** es muy común hablar igualmente de **riesgos**. En ciertas fuentes distinguen entre ambos términos haciendo ver que el riesgo es consecuencia de la dependencia.

embargo, esta solución tiene la clara limitación del coste de duplicar los recursos hardware de la máquina.

Dependencias de datos

Ocurre cuando una instrucción depende de los datos de otra. Esto se puede explicar suponiendo que se dispone de dos instrucciones de un mismo programa i_k e i_l tales que la primera precede a la segunda en el orden de ejecución. Estas dos instrucciones serán dependientes siempre que tengan un operando (registro o memoria) común, a menos que ambas lo empleen como operando fuente. La dependencia se produce también si existe una tercera instrucción i_j, tal que i_j depende de i_k e i_l depende de i_j (transitividad). Se pueden clasificar estas dependencias en tres tipos:

- **RAW** (*Read After Write*) o dependencia verdadera: Se produce cuando una instrucción i_l emplea como operando fuente el resultado de la instrucción i_k.

```
ik:   LI    R1, 0(R4)
il:   ADDI  R3, R2, R1
```

El registro R1 es destino de la primera instrucción y se emplea como operando fuente de la suma. La instrucción i_l no puede ser ejecutada correctamente hasta que la instrucción i_k haya finalizado su ejecución.

Ejemplo 2-4. Dependencia RAW

Este tipo de dependencia es considerado por ciertos autores como la única dependencia de datos auténtica, ya que es la única que se traduce en un flujo efectivo de datos de la primera instrucción a la segunda.

- **WAR** (*Write After Read*) o antidependencia: Se produce cuando una instrucción i_l tiene como resultado uno de los operandos fuente de la instrucción i_k.

```
ik:   MULTI R2, R1, R5
il:   ADDI  R1, R3, R4
```

Si la instrucción de suma adelantara por cualquier motivo su resolución a la de multiplicación el registro R1 cambiaría su valor como operando fuente de la multiplicación. La multiplicación obtendría entobces un resultado erróneo.

Ejemplo 2-5. Dependencia WAR

Este tipo de dependencia se considera una dependencia *falsa* porque se elimina empleando técnicas de renombrado de registros.

- **WAW** (*Write After Write*) o dependencia de salida: Se produce cuando dos instrucciones escriben en el mismo destino.

```
ik:   MULTI R1, R2, R5
il:   ADDI  R1, R3, R4
```

Si la instrucción de multiplicación tarda más en escribir en R1 que la suma, R1 contendrá un resultado incorrecto después de estas dos instrucciones.

Ejemplo 2-6. Dependencia WAW

Este tipo de dependencia, al igual que la dependencia WAR, se considera una dependencia *falsa*, y puede eliminarse empleando la misma técnica de renombrado.

Hasta el momento se han nombrado dependencias en código lineal (*straight-line code*). Sin embargo, existen otro tipo de dependencias de datos más difíciles de descubrir a primera vista pero tan determinantes como éstas: las dependencias en bucles o **recurrencias**[6]. Las recurrencias son referencias en el cuerpo de un bucle a datos que han sido computados en iteraciones anteriores. En el bucle del Ejemplo 2-7 puede observarse una situación de este tipo.

```
for i = 1 to n do
    X[i] = A*X[i-1] + B;
endfor
```

Para poder hallar el valor de X[i] es necesario tener calculado el valor de X[i-1], que se ha computado en la iteración anterior.

Ejemplo 2-7. Recurrencia

Las recurrencias se pueden clasificar según la "distancia" medida en iteraciones a la que se encuentra el dato del que depende. Por ejemplo: `X[i] = X[i-5] + B` es de grado 5. Esta distancia es importante porque cuanto mayor sea, mayor es el grado de paralelismo que puede obtenerse empleando distintas técnicas.

Dependencias de control

La ejecución de las instrucciones que siguen a un salto condicional depende de éste. Las dependencias de control tienen una gran influencia en el rendimiento de la máquina, pero su impacto varía mucho de un tipo de programa a otro. En los programas de propósito general (compiladores, SO, programas de aplicaciones no numéricas) se estima que se puede encontrar un salto condicional con una frecuencia de un 20-30%, lo que significa un salto cada 3-6 instrucciones. En los programas científicos, sin embargo, este porcentaje decrece hasta el 5-10%, lo que significa un salto cada 10-20 instrucciones.

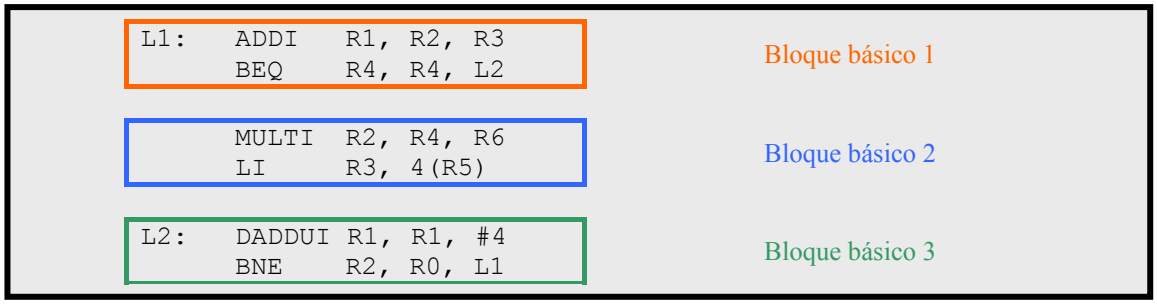
2.5. Bloque Básico

Un término muy importante cuando se habla de las dependencias de control, y que se usará bastante en el resto del documento es el de **bloque básico**. Un bloque básico es una secuencia lineal de código que tiene una única entrada y una única salida, es decir, no puede haber saltos hacia (excepto hacia la primera instrucción del bloque) o desde (excepto desde la última instrucción del bloque) un bloque básico.

Un bloque básico empieza al comienzo de un programa o subprograma; en cualquier instrucción destino de un salto; y justo después de una instrucción de salto. El final de un bloque básico puede ser el final del programa o la instrucción anterior al comienzo de otro bloque básico. En el Ejemplo 2-8 se señalan los bloques básicos de un pequeño código.

---

[6] Además de recurrencia (*recurrence*), suelen usarse otros términos como *inter-iteration data dependencies* o más frecuentemente ***loop-carried dependencies***.



Ejemplo 2-8. Bloques básicos

2.6. Superescalar

Los procesadores superescalares tienen como principal característica la iniciación de más de una instrucción por ciclo (frente a los procesadores "escalares", que emiten una única instrucción). Esta característica se explota mediante una planificación estática o, en la gran mayoría de los casos, dinámica de las instrucciones.

Un procesador superescalar implementa:

- Estrategias de **búsqueda simultánea** de varias instrucciones, incluyendo predicción de saltos condicionales.
- Métodos para **determinar dependencias** verdaderas y mecanismos para **comunicar esos valores** a donde se necesitan durante la ejecución.
- Métodos para la **emisión simultánea** de varias instrucciones.
- Recursos que permitan **ejecutar en paralelo**, incluyendo varias unidades funcionales segmentadas y una jerarquía de memoria capaz de resolver varias referencias a memoria simultáneamente.
- Métodos para **comunicar valores a través de la memoria** (mediante instrucciones de *LOAD* y *STORE*) e interfaces con la memoria que tengan en cuenta el comportamiento dinámico e impredecible de las jerarquías de memoria. Estos interfaces deben construirse acordes con las estrategias de ejecución de instrucciones.
- Métodos para "entregar" (*commit*) el estado del procesador en orden correcto, de tal manera que los programas **aparenten una ejecución secuencial**.

La necesidad de iniciar más de una instrucción por ciclo tiene una serie de implicaciones. La primera es el hecho de tener que buscar múltiples instrucciones por ciclo desde la caché. El soporte de este gran ancho de banda de instrucciones hace prácticamente obligatorio **separar la caché de datos de la caché de instrucciones**. El número de instrucciones buscadas por ciclo debe satisfacer las necesidades del decodificador y las unidades de ejecución. Normalmente se maneja un margen amplio de instrucciones para poder controlar fallos de caché de instrucciones y otras situaciones donde no pueden buscarse tantas instrucciones. Para almacenar estas instrucciones suele emplearse un *buffer* (el ***buffer* de instrucciones**) entre la caché y el decodificador.

El corazón de la máquina superescalar lo componen una serie de **unidades funcionales** que pueden operar en paralelo. También denominadas **unidades de ejecución**, se encargan precisamente de ejecutar todos los tipos de instrucciones e interactuar con la memoria y los bancos de registros. Este esquema se complica bastante cuando se quiere aprovechar toda la funcionalidad de esta máquina.

En los siguientes puntos se explicarán los distintos mecanismos que permiten aprovechar el paralelismo a nivel de instrucción mediante el hardware en el ámbito de una máquina superescalar.

2.6.1. Planificación dinámica

Una de las mayores limitaciones de la segmentación simple es que procesa las instrucciones siempre en el mismo orden en que las va recibiendo. Como las instrucciones se emiten en el orden que tienen en el programa, es en ese orden como se ejecutan. Con esta limitación, si existe una dependencia entre dos instrucciones consecutivas, se producirán sin remedio "burbujas" en el flujo de ejecución. Incluso disponiendo de varias unidades funcionales, algunas de ellas quedarán inutilizadas a la espera de que termine la instrucción que creó la dependencia.

```
MULTF F0, F2, F3
ADDF  F1, F0, F4
ADDF  F5, F6, F7
```

Existe una dependencia debido al registro F0 entre la primera y segunda instrucciones. En una ejecución en orden la tercera instrucción debería esperar hasta que finalizaran las dos primeras, incluso habiendo recursos disponibles para ejecutarla, y siendo evidente que su ejecución no afecta a las otras dos instrucciones.

Ejemplo 2-9. Ejecución en orden

La planificación dinámica viene a resolver muchos de los problemas derivados del estatismo que implica la ejecución en orden. La clave para conseguir una ejecución fuera de orden (*out-of-order*) es dividir la emisión de la instrucción en dos etapas. En la primera etapa (la emisión propiamente dicha) se comprueban simplemente las dependencias estructurales, es decir, que estén disponibles los recursos necesarios para ejecutar la instrucción. En la segunda etapa, denominada envío (*dispatch*), las instrucciones se retienen hasta tener los operandos disponibles, comprobándose las dependencias entre los datos.

La ejecución fuera de orden añade varias complicaciones a la ejecución normal de las instrucciones. La primera es la aparición de los riesgos WAR y WAW, que en la ejecución en orden no existían. Estos dos tipos de riesgo se eliminan empleando renombrado de registros como se verá más adelante. La otra complicación es el manejo de las excepciones. Empleando ejecución fuera de orden pueden producirse **excepciones imprecisas**, que son excepciones que se producen cuando el estado del procesador no es exactamente el mismo que si se hubiera ejecutado el programa en orden. Existen dos situaciones donde se pueden producir excepciones imprecisas:

- El *pipeline* ha completado ya instrucciones posteriores, según el orden del programa, a la instrucción que produjo la excepción.
- El *pipeline* aún no ha completado la ejecución de alguna instrucción que, según el orden del programa, es anterior a la instrucción que produjo la excepción.

Son necesarios componentes hardware que permitan implementar los mecanismos de la ejecución fuera de orden. Más adelante en este mismo capítulo se verán algunos de estos componentes.

2.6.2. Estaciones de Reserva

La ejecución fuera de orden introdujo la necesidad de dividir la emisión de las instrucciones en dos etapas. Para llevar a cabo esta división pueden emplearse varios mecanismos. Uno de ellos, denominado ***shelving*** [23], consiste en el empleo de unos *buffers* (*shelving buffers*) antes de cada unidad funcional para almacenar las instrucciones en el momento de su emisión. Las únicas restricciones que se tienen en cuenta cuando se emite una instrucción a estos *buffers* son restricciones de recursos hardware. El chequeo de las dependencias de datos se realiza en un segundo momento al enviar (*dispatch*) las instrucciones desde estos *buffers* a su unidad funcional correspondiente.

Dentro de las estructuras empleadas para el *shelving* se encuentran dos tipos bien diferenciados: los buffers de uso exclusivo (como las estaciones de reserva) y los *buffers* combinados (el *reorder buffer*). Existen tres esquemas básicos de diseño de las estaciones de reserva:

- Estaciones de reserva **individuales**: Cada estación de reserva se asocia con una unidad funcional (Ej. PowerPC 620). Normalmente tienen un número bastante limitado de instrucciones que pueden mantener (de 2 a 4).
- Estaciones de reserva **agrupadas**: La misma estación de reserva almacena las instrucciones de un grupo de unidades funcionales (Ej. R10000). Obviamente tienen un número mayor de entradas que el caso anterior. Puesto que tienen asociadas varias unidades funcionales suelen estar preparadas para enviar más de una instrucción por ciclo.
- Estación de reserva **centralizada**: Una única estación de reserva para todas las unidades funcionales con, obviamente, un número mucho mayor de entradas (Ej. PentiumPro).

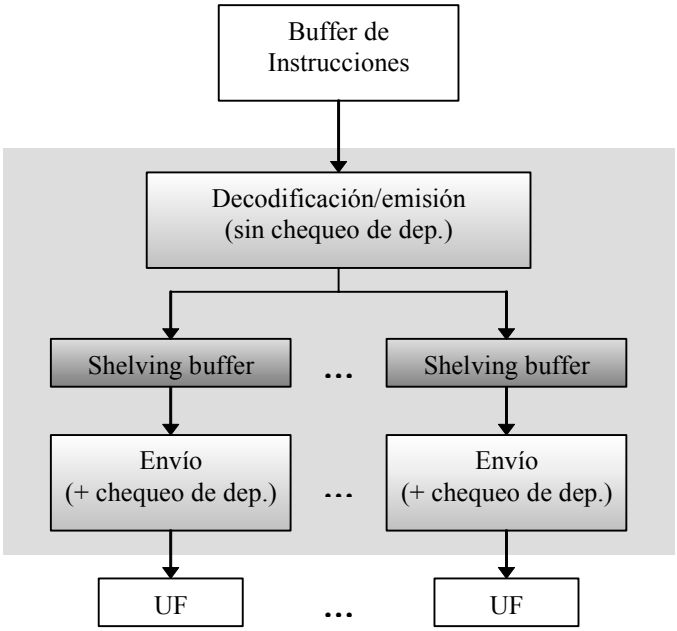


Ilustración 2-4. Esquema de la emisión de instrucciones empleando *shelving buffers*.

2.6.3. Accesos a memoria con planificación dinámica

Otra de las complicaciones añadidas con la ejecución fuera de orden son las dependencias entre las instrucciones que acceden a memoria. Si acceden a diferentes

direcciones de memoria no existe ningún problema en cambiar su orden de ejecución; sin embargo, si acceden a la misma dirección puede producirse cualquiera de los tres tipos de dependencias de datos:

- Si un *LOAD* que debía ejecutarse antes que un *STORE* cambia su orden se produce un riesgo WAR.
- Si es el *STORE* el que se ejecuta antes que el *LOAD* y cambian su orden, se tiene un riesgo RAW.
- Si son dos *STORES* los que cambian su orden, entonces es un riesgo WAW el que se produce.

Para solucionar este problema deben almacenarse las direcciones a las que se va accediendo. Para determinar si un *LOAD* puede ejecutarse se comprueba que no exista ningún *STORE*, anterior según el orden del programa, que acceda a la misma dirección de memoria y que aún no se haya completado. En el caso del *STORE*, debe comprobarse que no exista ningún *LOAD* o *STORE* anterior que comparta la misma dirección de memoria.

El almacenamiento de las direcciones se realiza en unos *load buffers* y *store buffers*, que son dos estructuras que vienen a comportarse de forma muy parecida a una estación de reserva.

Disponiendo de estas estructuras se divide la ejecución de las instrucciones que acceden a memoria en dos fases:

- En la primera fase se espera a que el *registro base* esté disponible y se calcula la dirección efectiva. Una vez calculada, se almacena en el *buffer* (*store* o *load*) correspondiente.
- La segunda fase realiza el acceso a memoria propiamente dicho. En el caso de los *LOAD* este acceso se hace tan pronto como se tiene la dirección efectiva calculada y hay una unidad de memoria libre. Los *STORE* deben tener también disponible el valor que quieren escribir en memoria antes de hacer uso de la unidad funcional.

Simplemente asegurando que la primera fase se realizara en el orden del programa podrían detectarse los riesgos, aunque en la práctica sólo es necesario mantener el orden relativo entre los *STORES* y el resto de accesos a memoria.

2.6.4. Predicción de salto

En un esquema dinámico como el de las máquinas superescalares son indispensables mecanismos que faciliten la resolución de los saltos condicionales. Si no se empleara ningún mecanismo que permitiera en cierta forma "predecir" el resultado de una instrucción de salto se introducirían burbujas en el flujo de ejecución intolerables para el rendimiento de la máquina. La **predicción dinámica de salto** consiste en adelantar el resultado del salto basándose en alguna información (historial del salto, saber si es un salto hacia delante o hacia atrás…) para poder cargar la siguiente instrucción inmediatamente. Estos mecanismos no son infalibles, así que al emplearlos hay que dotar a la máquina de herramientas para poder afrontar una equivocación en la predicción.

Predicción de salto básica

La forma más sencilla de predecir un salto es emplear una **tabla de predicción** de salto. Esta tabla es una pequeña memoria indexada por la porción más baja de la dirección de las instrucciones de salto. Esta tabla puede contener de uno a varios bits que permitan predecir si el salto debe tomarse o no. En el esquema más simple, de 1 bit, el valor del bit indica si un salto que coincide con esa dirección en la tabla fue tomado anteriormente o no. Este salto puede no ser el mismo salto por el que se está preguntando en este momento, pero eso no importa de cara a la predicción, que se toma siempre como correcta. Se busca la siguiente instrucción en la dirección predicha, y si al final resulta que la predicción era incorrecta, se cambia el valor del bit.

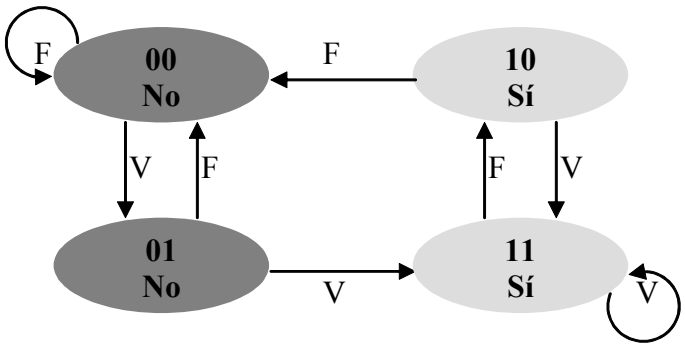


Ilustración 2-5. Esquema de predicción de 2 bits. Se indica la configuración binaria de cada estado y si el salto se toma o no

El rendimiento obtenido con 1 bit es fácilmente mejorable añadiendo un segundo bit a cada entrada de la tabla. El esquema de 2 bits obliga a que una predicción falle dos veces antes de cambiarla, como se ve en la Ilustración 2-5.

Este esquema, que sigue siendo bastante simple, tiene sin embargo una precisión bastante alta. De hecho, el uso de un esquema más general de *n* bits no aporta una excesiva mejora a los resultados obtenidos. Resulta mucho más determinante el número de entradas que tenga la tabla que el número de bits de cada entrada, pero incluso este parámetro no tiene una influencia decisiva en los resultados obtenidos. Para mejorar los resultados habría que buscar otros mecanismos de predicción complementarios, que usaran más información[7].

2.6.5. Ejecución Especulativa

La predicción de salto es un buen mecanismo para facilitar la aplicación de la ejecución fuera de orden. La planificación dinámica sólo busca y emite las instrucciones predichas, pero para poder aprovechar todo el potencial de una máquina que emite más de una instrucción por ciclo esto es claramente insuficiente. La clave está en permitir ejecutar las instrucciones predichas asumiendo que la predicción fue correcta, realizando **ejecución especulativa**.

La **especulación basada en hardware** (existen también técnicas de especulación que puede emplear el compilador, que se nombrarán al ver las máquinas VLIW en la sección 2.7) se apoya en tres ideas principales:

- Predicción dinámica de saltos para elegir las instrucciones que ejecutar.

[7] Para ver algunos de estos mecanismos puede hojearse por ejemplo [H&P-03] en las páginas 200-215, o [SIM-97] de la 325 a la 348.

- Especulación para permitir ejecutar estas instrucciones antes de haber resuelto la dependencia de control (y herramientas para poder deshacer esta ejecución en el caso que se compruebe que la predicción fue incorrecta).
- Planificación dinámica para poder emitir instrucciones de diferentes bloques básicos.

Para implementar la especulación basada en hardware puede emplearse el algoritmo de Tomasulo como se verá en la sección 2.6.9.

2.6.6. Predicción de salto con ejecución especulativa

La búsqueda de instrucciones estándar emplea el PC como apuntador a la primera de las instrucciones de cada bloque de instrucciones en simplemente incrementa el PC en un número igual al de instrucciones buscadas, para luego buscar el siguiente bloque de instrucciones. La búsqueda en un esquema con ejecución especulativa necesita proporcionar a la máquina las instrucciones que supuestamente van a ser ejecutadas. Esto implica que, ante una instrucción de salto, deben poderse buscar las instrucciones destino del salto. Esta tarea puede dividirse en las siguientes etapas:

- **Reconocimiento de que la instrucción es un salto condicional**. Esto implica una **predecodificación** anterior a la caché de instrucciones que genera unos bits adicionales que se añaden a cada instrucción. De esta manera, ya se sabría que se trata de un salto al llegar a la etapa de búsqueda.
- **Determinación del comportamiento del salto**. Esta tarea queda cubierta con el uso de algún método de predicción de salto condicional como los vistos en la sección 2.6.4.
- **Cálculo de la dirección destino del salto**. Esta tarea suele requerir una suma entera, debido a que los destinos de los saltos suelen ser relativos al PC (e incluso pueden depender del valor de algún registro). Para acelerar este cálculo se emplea un *buffer* de destino de saltos que almacena la última dirección asociada a ese salto. Un ejemplo es usado en el PowerPC 604 [12].
- **Transferencia de control**. Todas las tareas anteriores provocan al menos un ciclo de retraso en la búsqueda de la siguiente instrucción en el caso de un salto tomado. Para enmascarar este retraso se suele emplear el *buffer* de instrucciones, o incluso técnicas como cargar los dos caminos (tomado y no tomado) del salto simultáneamente.

2.6.7. Reorder Buffer

Ya en el punto 2.6.2 se introdujo el concepto de *Reorder Buffer* (**ROB**) como un tipo de *shelving buffer* combinado. Originalmente fue planteado como un sistema de control preciso de interrupciones [25], pero actualmente puede definirse como una herramienta que asegura la **consistencia secuencial** de la ejecución cuando se emplean varias unidades funcionales.

El ROB puede verse como un *buffer* circular con un puntero a su cabeza indicando la siguiente posición libre, y un puntero a su cola indicando la siguiente instrucción que puede ser retirada. Las instrucciones se van introduciendo y retirando del ROB en el orden del código secuencial. De esta manera se puede asegurar la consistencia en la ejecución del programa.

Las dos funciones principales del ROB son el apoyo a la **ejecución especulativa** y al **manejo de las interrupciones**. Cuando se emplea ejecución especulativa todas las instrucciones especuladas quedan marcadas en el ROB. Al comprobar la condición que ha dado lugar a la especulación, se marcan las instrucciones especuladas como "verdaderas" o "falsas". Aquellas marcadas como falsas se anulan y no se finaliza su ejecución. En cuanto al control preciso de las interrupciones, se consigue permitiendo únicamente las peticiones de interrupción de las instrucciones que se van a retirar.

Un ROB es una herramienta bastante versátil. En un primer acercamiento puede verse únicamente como una herramienta que proporciona un reordenamiento de las instrucciones, para conseguir que la ejecución efectiva del código se haga en orden. A esta función puede añadírsele la capacidad de renombramiento (*renaming*), haciendo que el ROB contenga también el resultado de la instrucción. Por último, puede contemplarse la función de *shelving* como se nombraba al principio. Esta última opción se suele denominar DRIS (*Deferred scheduling, Register renaming Instruction Shelf*), y se traduce en que el ROB añade un espacio para contener los números de registro de los operandos o los propios valores.

2.6.8. Renombrado de registros

Un concepto que ya ha sido nombrado en varias ocasiones es el renombrado de registros (*register renaming*). Esta técnica se emplea para eliminar las dependencias WAR y WAW, que obligan al programador[8] a reutilizar frecuentemente los mismos registros debido a su número limitado.

El *renaming* se basa en distinguir entre los registros de la máquina visibles para el programador (**registros arquitecturales** o lógicos), y los elementos reales de almacenamiento de los que dispone la máquina (**registros físicos** de la máquina, aunque en este caso se extiende un tanto la definición de *registro*).

Esta técnica puede afrontarse principalmente desde dos perspectivas. La primera forma de tratar el problema consiste en disponer de un conjunto de registros físicos mucho mayor que el de los registros lógicos visibles para el programador. Al decodificar una instrucción se asocia al registro lógico destino de la operación un registro físico. Esta asociación se controla mediante una **tabla de *mapeo***, de tal manera que cualquier instrucción posterior que quiera leer de ese registro lógico sabrá que ese valor se encuentra realmente en el registro físico indicado en la tabla. La disponibilidad de los registros físicos se controla mediante una **lista de registros físicos libres** que, en caso de vaciarse, obliga a detener el *envío* de las instrucciones a las unidades de ejecución. Puede verse esto con más claridad en el Ejemplo 2-10.

Un registro físico se retira de la lista de registros libres una vez utilizado y debe volver cuando haya finalizado completamente la ejecución de la última instrucción que lo empleaba como registro fuente. La determinación de la disponibilidad del registro es quizás el problema más complejo de este método.

Una primera solución podría ser el uso de un contador asociado con cada registro físico. Este contador se incrementaría cada vez que un registro fuente de una instrucción fuese renombrado con el registro físico; y sería decrementado cada vez que una instrucción emitida leyese el valor de ese registro. Al llegar el contador a 0 el registro se libera.

---

[8] Cuando se habla aquí del programador, puede entenderse también como el compilador de la máquina; en definitiva se está hablando de la visión a nivel lógico (arquitectural) de la máquina.

Existe una solución bastante más simple, consistente en esperar simplemente a que la última instrucción que empleaba ese registro físico como operando fuente llegue a su etapa de graduación. En ese momento puede liberarse el registro físico sin ningún riesgo.

Antes del renombrado…

Instrucción	Tabla de mapeo		Lista de registros libres
`ADDI R3, R3, #4`	R0	rf8	rf2, rf6, rf12…
	R1	rf22	
	R2	rf10	
	R3	rf1	
	…		

Después del renombrado…

Instrucción	Tabla de mapeo		Lista de registros libres
`ADDI rf2, rf1, #4`	R0	rf8	rf6, rf12…
	R1	rf22	
	R2	rf10	
	R3	rf2	
	…		

Ejemplo 2-10. *Renaming* empleando registros. Los registros arquitecturales se denotan como "R#", mientras los registros físicos se nombran "rf#"

Aún puede buscarse otra solución, menos eficiente pero incluso más simple de implementar, consistente en liberar el registro cuando se gradúe otra instrucción que emplee el mismo registro lógico como destino. Esto conlleva un claro desaprovechamiento de los registros, pero su simplicidad hace que sea empleada en la práctica, por ejemplo por el R10000 de MIPS.

La segunda perspectiva que se enfrenta al problema del renombrado de registros emplea estructuras ya vistas: el ROB y las estaciones de reserva. Lo que se hace es aprovechar que el ROB tiene una entrada por cada instrucción emitida para añadir un campo que contenga el resultado de la operación. En las estaciones de reserva se hace referencia a posiciones del ROB en lugar de a registros como operandos fuente, para lo cual sigue siendo necesaria la tabla de *mapeo* que indique en qué posición del ROB se encuentra el valor asociado a ese registro lógico. Al terminar la ejecución de la instrucción se actualizará efectivamente el registro destino con el resultado que se dispone en la entrada del ROB. Este método se verá con mayor claridad al explicar el algoritmo de Tomasulo en el siguiente punto.

2.6.9. Algoritmo de Tomasulo

Una vez definidos los elementos fundamentales empleados en una máquina superescalar, hay que definir un algoritmo que permita el control de estos componentes y la implementación de la ejecución especulativa y la planificación dinámica.

El algoritmo de Tomasulo fue un diseño original de Robert Tomasulo para el control de las dependencias en la unidad funcional de punto flotante de la 360/91 de IBM. El esquema diseñado introdujo un seguimiento de los operandos, para minimizar los riesgos RAW, y renombrado de registros, para eliminar las dependencias WAR y WAW. En este algoritmo las instrucciones se procesaban en tres etapas: emisión, ejecución (que incluye la etapa de *dispatch* que se vio al hablar de planificación dinámica) y escritura de resultados.

La adaptación realizada para controlar la ejecución especulativa mediante el algoritmo de Tomasulo consiste en separar la escritura de los resultados de las instrucciones de la finalización de la instrucción en sí. De esta forma, pueden pasarse los resultados a otras instrucciones de manera temporal, permitiendo deshacer cualquier operación, hasta que sepamos que la instrucción no es especulativa. En el momento en que deja de ser especulativa, se realizan las escrituras definitivas en los registros o en la memoria de los resultados de la instrucción. Esta fase recibe el nombre de **graduación** (*commit*), y puede realizarse varios ciclos después de la finalización de la ejecución de la instrucción. Es precisamente para almacenar el resultado de la instrucción durante este periodo para lo que se introduce el ROB en este esquema.

El ROB tiene más funciones en este esquema, pues se usa para el renombrado de registros y también asume las funciones del *store buffer*, almacenando las direcciones y valores de las operaciones de memoria.

Un elemento muy importante para la máquina superescalar cuando se emplea este algoritmo y que aún no se ha nombrado es el **CDB** (*Common Data Bus*). Este bus permite a todas las unidades que están esperando por un resultado cargarlo simultáneamente.

Para explicar el algoritmo se propone un esquema (Ilustración 2-6) muy simple en el únicamente se muestra la ejecución de instrucciones de ALU y de acceso de memoria. Toda la parte de búsqueda y decodificación de instrucciones también se ha obviado.

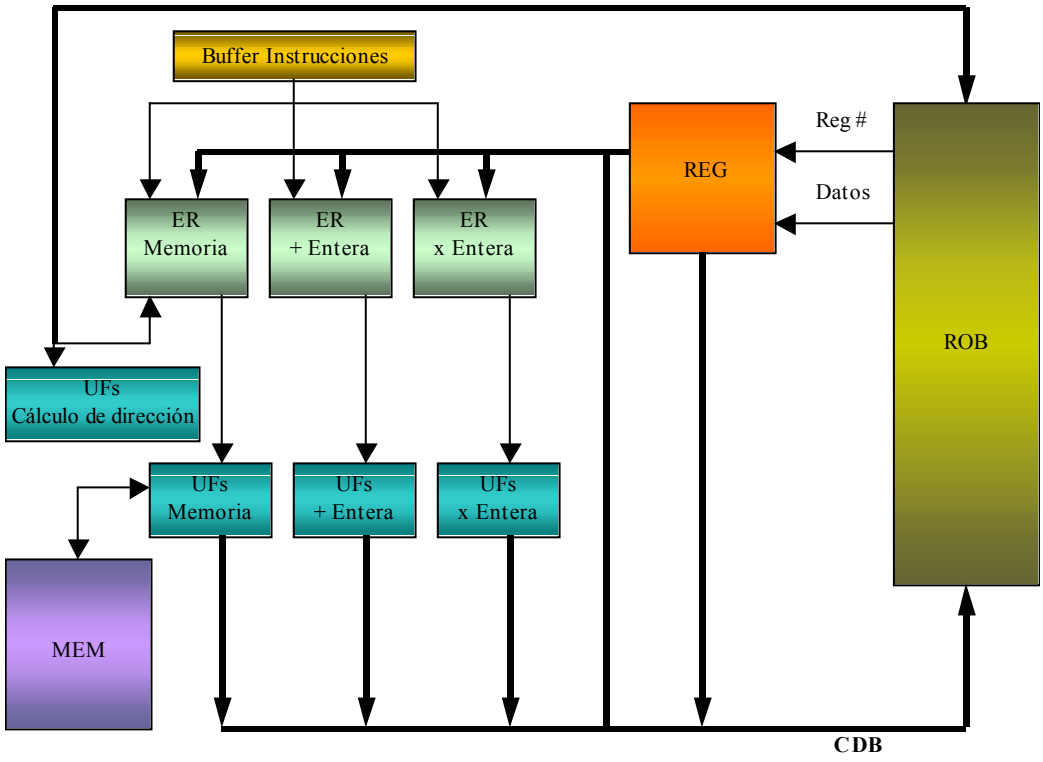


Ilustración 2-6. Esquema simplificado de una máquina superescalar que emplea el algoritmo de Tomasulo.

Ciertos elementos del esquema poseen una serie de campos imprescindibles para la comprensión del algoritmo:

- **ROB**: Estructura del *Reorder Buffer*. Cada entrada contiene los siguientes campos:

- instrucción: Instrucción asociada a esa entrada
- listo: Indica si el resultado asociado a esa instrucción está disponible
- valor: Resultado de la instrucción
- destino: Registro destino de la operación que realiza la instrucción
- dirección: En el caso de una instrucción de acceso a memoria, dirección efectiva calculada

➢ **ER**: Estructura de las Estaciones de Reserva. Cada entrada se compone de los siguientes campos:

- ocupada: Indica si esa entrada está en uso o no.
- rob: Posición del ROB donde se almacenará el resultado de la instrucción que se opera en esta entrada de la ER
- Qj/Qk: Indicador de disponibilidad del primer/segundo operando (-1 indica que el resultado se encuentra disponible en Vj/Vk, en otro caso se pone el índice de la entrada del ROB que contiene ese resultado).
- Vj/Vk: Valor del primer/segundo operando.
- A: Contiene primero el *offset* de la dirección de una instrucción de memoria y después la dirección efectiva calculada.

➢ **REG**: Banco de registros. Cada registro tiene asociados los campos:

- ocupado: Indica si el valor de ese registro se está calculando en una entrada del ROB.
- rob: Entrada del ROB donde se está calculando el valor del registro si (ocupado = verdadero).

Con todos estos elementos puede verse como funciona el algoritmo explicando cada etapa. Debe tenerse en cuenta que el algoritmo está planteado originalmente para emisión simple de instrucciones, y se detallará para ese caso por simplicidad. Extender el concepto a múltiple emisión no conlleva mayores dificultades en cuanto a la funcionalidad del algoritmo. Las etapas son las siguientes:

- **Emisión**
  - Se obtiene una instrucción desde el *buffer* de instrucciones.
  - Se comprueba que haya alguna entrada libre en el ROB (*b*) y en la estación de reserva (*e*) que le corresponda. Si no tiene espacio para emitirse se detiene la ejecución hasta que se libere.
  - Si la instrucción tiene un operando fuente (ro1) se realiza el siguiente chequeo:

```
si (REG[ro1].ocupado) {
    h = REG[ro1].rob;
    si (ROB[h].listo) {
        ER[e].Vj = ROB[h].valor;
        ER[e].Qj = -1;
    } else
        ER[e].Qj = h;
} else {
    ER[e].Vj = REG[ro1];
```

```
    ER[e].Qj = -1;
}
```

El mismo chequeo se realiza si la instrucción tiene un segundo operando fuente (ro2) pero sustituyendo Vj y Qj por Vk y Qk respectivamente.

Si uno de los operandos es un valor inmediato (*Imn*), se almacena directamente en el Vk o Vj correspondiente.

- Tanto las instrucciones de ALU como los *LOAD* tienen como operando destino un registro (*rd*). Éste debe marcarse para indicar que su resultado está en una entrada del ROB; y en el ROB se indica el destino de la instrucción como ese registro.

```
REG[rd].rob = b;
REG[rd].ocupado = sí;
ROB[b].destino = rd;
```

- La entrada de la estación de reserva debe saber qué entrada del ROB le corresponde para poder enviar los resultados por el CDB.

```
ER[e].ocupada = sí;
ER[e].rob = b;
```

- Las instrucciones de acceso a memoria almacenan además el *offset* de la dirección.

```
ER[e].A = offset;
```

- **Envío/Ejecución**
  - La base de esta etapa es esperar, chequeando el CDB, hasta tener los operandos disponibles para entonces iniciar la ejecución.
  - Las instrucciones de ALU chequean el valor de Qj y Qk hasta que valen –1, y en ese momento envían la instrucción a la unidad funcional correspondiente.
  - Las instrucciones de acceso a memoria chequean que no haya *STORES* anteriores en el ROB antes de calcular la dirección, aunque ya se tenga el registro base disponible (Qj = -1). De esta manera se garantiza el orden relativo de estas instrucciones y se evitan los riesgos de datos.

```
ER[e].A = ROB[b].dirección = ER[e].Vj + ER[e].A;
```

  - Las instrucciones de *LOAD* realizan su segunda fase en esta etapa después de tener calculada la dirección y tras comprobar que no exista ningún *STORE* anterior en el ROB que haga referencia a la misma dirección de memoria.
  - Las instrucciones de *STORE* no realizan ninguna otra tarea en esta etapa, y simplemente se quedan esperando a tener disponible su operando fuente antes de llegar a la etapa de graduación. En cuanto dispongan de este valor harán:

```
ROB[b].valor = ER[e].Vk;
```

- **Escritura de resultados**
  - Cuando la ejecución en la unidad funcional finaliza, el resultado (*resul*) se envía a través del CDB hasta el ROB, y también a cualquier estación de reserva que esté esperando por ese resultado como operando fuente. Tras esto, la estación de reserva se libera.

```
b = ER[e].rob;
```

```
ER[e].ocupado = no;
Para toda entrada x de una estación de reserva hacer
    si (ER[x].Qj = b) {
        ER[x].Vj = resul;
        ER[x].Qj = 0;
    }
    si (ER[x].Qk = b) {
        ER[x].Vk = resul;
        ER[x].Qk = 0;
    }
ROB[b].valor = resul;
ROB[b].listo = sí;
```

- **Graduación**
  - La instrucción del frente del ROB (*h*) puede graduarse si ha finalizado las etapas anteriores.
  - Si la instrucción es un salto que se comprueba que fue incorrectamente predicho debe limpiarse el flujo de ejecución que ha sido especulado: ROB, estaciones de reserva...
  - Si se trata de un *STORE* que ya tiene el valor disponible se realiza la escritura en memoria.

    ```
    MEM[ROB[h].dirección] = ROB[h].valor;
    ```
  - Si es cualquier otro tipo de operación cuyo destino es un registro (*d*), se escribe definitivamente el resultado en ese registro.

    ```
    REG[d] = ROB[h].valor;
    ```
  - En cualquier caso se libera la entrada del ROB.

    ```
    ROB[h].ocupado = no;
    if (REG[d].rob == h)
        REG[d].ocupado = no;
    ```

2.7. VLIW

Los procesadores **VLIW** (*Very Long Instruction Word)* se caracterizan por emitir cada ciclo un número fijo de operaciones[9] formateadas como una instrucción larga o como un paquete de operaciones donde se indican explícitamente las dependencias (en este caso, se habla de máquinas **EPIC**: *Explicitly Parallel Instruction Computers*). Por tanto, la planificación estática realizada por el compilador tiene un papel fundamental en este tipo de máquinas.

El corazón de las máquinas VLIW es muy similar al de las máquinas superescalares. Se compone como aquellas de una serie de **unidades funcionales** que operan en paralelo e interactúan con la memoria y los registros. El resto del hardware, al menos en las aproximaciones más clásicas, poco tiene que ver con el visto en las máquinas superescalares.

La aproximación más clásica a la arquitectura VLIW [7] define un único flujo de ejecución que incluye un contador de programa y una única unidad de control. Esta unidad de control emite una instrucción larga por ciclo compuesta de operaciones independientes que permiten controlar directamente todas las unidades funcionales en

[9] Cuando se hable de máquinas VLIW se empleará el término **operación** para referirse a las instrucciones básicas (originarias del repertorio RISC) de la máquina y no confundirlas con las **instrucciones largas**.

cada ciclo de reloj. Cada operación requiere un número pequeño y predecible de ciclos para ejecutarse y puede (idealmente debería) ser segmentada.

Existen dos diferencias básicas con las máquinas superescalares: cómo se construyen las instrucciones y cómo se lleva a cabo su planificación.

Siguiendo con el esquema clásico VLIW (se descartará por ahora la aproximación de las máquinas EPIC, con una presencia mucho más activa del hardware) el tamaño de la instrucción larga depende del número de unidades funcionales de cada tipo de que disponga la máquina y del tamaño del código requerido para controlar cada unidad funcional. Por regla general los procesadores VLIW incorporan del orden de 5 a 30 unidades funcionales, cada una con una palabra de control de en torno a 16-32 bits. Una instrucción de este tamaño permite una explotación del paralelismo casi imposible de conseguir con una máquina superescalar, pero existe un problema evidente: surtir a esta instrucción de suficientes operaciones que puedan ejecutarse en paralelo para aprovechar estas capacidades. Desgraciadamente es muy difícil extraer suficiente paralelismo de un programa como para satisfacer las necesidades de estas instrucciones largas, lo que suele llevar a que las instrucciones no están nunca completamente llenas. El mayor inconveniente no es, sin embargo, el no aprovechamiento del paralelismo de la máquina, sino el gran desperdicio de memoria que se produce al almacenar un código de instrucciones largas.

La planificación en las máquinas superescalares se vio que era básicamente dinámica. En el caso de las VLIW, el control de la planificación recae exclusivamente en el compilador. Esta **planificación estática** implica una importante reducción de la complejidad del hardware. Ciertas tareas imprescindibles en los procesadores superescalares (decodificación, detección de dependencias, emisión de instrucciones...) se convierten en actividades triviales en este esquema. Esto permite ciertas optimizaciones de rendimiento, como el incremento de la frecuencia del reloj o un mayor aprovechamiento del paralelismo.

Pese a las ventajas nombradas, existen una serie de inconvenientes asociados al uso de las máquinas VLIW. La construcción de un compilador para un procesador VLIW es una tarea extremadamente compleja, pero éste no es su mayor defecto. El hecho de delegar en el compilador toda la planificación implica un fuerte conocimiento de la estructura interna de la máquina por parte del mismo. Por tanto, el compilador es tecnológicamente dependiente de la máquina para la que construye los códigos, lo que desemboca en la poca portabilidad, incluso en la misma familia de máquinas, de los códigos binarios de los ejecutables. Además, un problema derivado de éste es la imposibilidad de programar una máquina VLIW directamente en lenguaje ensamblador.

Otro problema que resulta evidente es el tratamiento de problemas dinámicos, que no pueden preverse durante la planificación, como los fallos de caché. El compilador debe planificar las operaciones teniendo en cuenta el peor caso, lo que significaría una pérdida importante de rendimiento. Este problema se ve agravado si se tiene en cuenta que en una máquina VLIW clásica, una burbuja en una unidad funcional implica la detención de todas las unidades funcionales, ya que debe asegurarse un sincronismo en la ejecución.

2.7.1. Técnicas del compilador para explotar el ILP

Ya que se está hablando del compilador, en esta sección se desarrollarán algunas de las técnicas software empleadas para entresacar el paralelismo a nivel de instrucción de un código secuencial. Para realizar esta tarea, el compilador debe buscar secuencias de

operaciones independientes que puedan ejecutarse simultáneamente en las unidades funcionales disponibles.

Si se quiere evitar la aparición de burbujas en el flujo de ejecución debe asegurarse que una operación que depende de otra no comience a ejecutarse hasta que esta operación que ya está en ejecución no haya finalizado. El número de ciclos de espera se denomina **latencia**[10]. En una unidad funcional segmentada cuyas etapas tarden exactamente un ciclo cada una, la latencia será igual a su número de etapas[11], que es precisamente el tiempo de ejecución de una operación en esa unidad funcional. Las latencias que se emplearán en los ejemplos de esta sección se pueden ver en la Tabla 2-4.

NOTA: Para los ejemplos de esta sección se supondrá que los registros fuente de una operación se leen siempre en el primer ciclo de la misma. De esta manera se evitan riesgos WAR como se verá en 4.5.2.

El ejemplo que se empleará en la mayor parte de esta sección se corresponde con este código:

```
for i = 0 to 1023 do
    X[i] = X[i] + B;
endfor
```

que traducido a código máquina sería un único bloque básico:

```
Loop: LF     F1, 0(R1)        ; R1: dirección memoria X
      ADDF   F2, F1, F0       ; F0 = B
      SF     F2, 0(R1)
      DADDUI R1, R1, #4       ; 4 bytes por simple precisión
      BNE    R1, R2, Loop     ; R2: dirección de X[1024] + 4
```

Planificado en una máquina con emisión simple de instrucciones y con las latencias vistas en la Tabla 2-4, el código tardaría 12 ciclos en ejecutar cada iteración.

```
Loop: LF     F1, 0(R1)        ; R1: dirección memoria X
      Burbuja
      Burbuja
      Burbuja
      ADDF   F2, F1, F0       ; F0 = B
      Burbuja
      Burbuja
      Burbuja
      SF     F2, 0(R1)
      DADDUI R1, R1, #4       ; 4 bytes por simple precisión
      BNE    R1, R2, Loop     ; R2: dirección de X[1024] + 4
      Burbuja
```

Ejemplo 2-11. Ejemplo usado para ilustrar técnicas software de ILP

[10] Esta definición no coincide exactamente con la dada en [13]. En ese texto se cuantifica la latencia en un ciclo **menos** que el tiempo de ejecución de la unidad funcional, ya que se entiende este término como el número de ciclos que debe haber entre dos operaciones dependientes.

[11] A menos que se emplee alguna estrategia de adelantamiento (*forwarding*) de resultados. Esta estrategia, que se emplea a menudo en máquinas segmentadas, se aprovecha de que muchas veces los resultados de una operación están calculados algún ciclo antes de escribirlos definitivamente en su registro destino o memoria. De esta manera están disponibles antes de que la operación haya terminado efectivamente su ejecución y pueden **adelantarse** a otra operación posterior en el flujo de ejecución que necesite ese valor. Puede verse este mecanismo en más detalle en [14], en todo el tema de Segmentación.

Operaciones	Latencia en los ejemplos
ADDI, DADDUI	1
MULTI	3
ADDF	4
MULTF	6
LI, SI, LF, SF	4 (9 con fallo de caché)
BNE, BEQ	2

Tabla 2-4. Latencias empleadas en los ejemplos

Planificación de bloques básicos

La aproximación más básica a las técnicas de planificación de código son las técnicas que trabajan únicamente con el código **dentro de un bloque básico** (ver 2.5). Estas optimizaciones locales tienen limitada la aceleración que puede obtenerse por las dependencias verdaderas y de control.

Este tipo de técnicas buscan una **ordenación** de las operaciones dentro del bloque básico que reduzca el número de burbujas que debe insertarse, como puede verse en el Ejemplo 2-12.

Aplicando planificación local al bloque básico de código:

```
Loop: LF     F1, 0(R1)
      Burbuja
      Burbuja
      Burbuja
      ADDF   F2, F1, F0
      Burbuja
      DADDUI R1, R1, #4
      BNE    R1, R2, Loop
      SF     F2, -4(R1)
```

Se reduce el número de ciclos a 9 por iteración (debe tenerse muy en cuenta que no hay dependencia entre la operación SF de una iteración y el LF de la siguiente; si no, esto no podría hacerse).

(Se ha aplicado en la planificación la técnica del salto retardado al poner el SF tras el salto. Esta técnica se verá en el siguiente punto).

Ejemplo 2-12. Planificación local de un bloque básico

Predicción estática de saltos

Como contrapunto a la predicción dinámica de salto que se veía en 2.6.4, existen una serie de mecanismos que tratan de aprovechar la información *a priori* de un salto para predecir en tiempo de compilación su comportamiento. Estos mecanismos se pueden usar también como complemento a los elementos hardware de la predicción dinámica.

Un primer acercamiento es el uso de **saltos retardados**. El salto retardado es una técnica que se introdujo desde las máquinas RISC. Consiste en aprovechar los ciclos que un salto tarda en resolverse. En ese tiempo se planifican operaciones que se sepa con seguridad que van a ejecutarse independientemente del resultado del salto. Un ejemplo de esto se observa en el Ejemplo 2-12, al poner la operación de *STORE* para aprovechar la burbuja después del salto. De esta manera se asegura la ejecución de la operación, ya que no se decide el salto hasta el siguiente ciclo, y se aprovecha ese ciclo que se perdía.

Otras técnicas tratan de predecir el comportamiento del salto en tiempo de compilación. Para ello deciden desde considerar el salto como verdadero siempre; seleccionar el destino en base a si es un salto hacia delante (se presupone no tomado) o hacia detrás (un bucle: se presupone tomado); o almacenar información de ejecuciones anteriores.

En definitiva, la utilidad de la predicción estática incluye:

- El apoyo a la planificación de operaciones cuando el retardo del salto queda expuesto por la arquitectura.
- La asistencia a los mecanismos de predicción dinámicos (la arquitectura IA-64 es un ejemplo de esto).
- La determinación de los caminos de código más frecuentes. Herramienta que se emplea en las técnicas de planificación global del código.

Desenrollado de bucles

El **desenrollado de bucles** (*loop unrolling*) se emplea por sí sólo o como componente básico de otros métodos más complejos. Se basa en la idea de **replicar el contenido** del cuerpo de un bucle varias veces y eliminar código superfluo, como decrementar/incrementar el contador del bucle o comprobar la condición final del bucle. Su utilización es muy simple cuando en tiempo de compilación se conoce el número de iteraciones del bucle.

Su aplicación, sin embargo, no es tan directa como podría parecer. En primer lugar debe tenerse en cuenta que distintas iteraciones del bucle precisan del uso de distintos registros para evitar riesgos de datos. Además, para eliminar operaciones de incremento del registro base para acceder a memoria, debe integrarse esa operación en el propio acceso a memoria como *offset* (Ejemplo 2-13).

Como el número de iteraciones (1024) es múltiplo de 4 se puede desenrollar cuatro veces el bucle, con lo que se dispone del siguiente código:

```
Loop: LF     F1, 0(R1)
      ADDF   F2, F1, F0
      SF     F2, 0(R1)        ; Fuera DADDUI y BNE
      LF     F3, 4(R1)        ; Usa el incremento (4) como offset
      ADDF   F4, F3, F0
      SF     F4, 4(R1)        ; Fuera DADDUI y BNE
      LF     F5, 8(R1)        ; Usa el incremento (8) como offset
      ADDF   F6, F5, F0
      SF     F6, 8(R1)        ; Fuera DADDUI y BNE
      LF     F7, 12(R1)       ; Usa el incremento (12) como offset
      ADDF   F8, F7, F0
      SF     F8, 12(R1)
      DADDUI R1, R1, #16
      BNE    R1, R2, Loop     ; R2: dirección de X[1024] + 16
```

En este caso, contabilizando las burbujas, cada iteración de este bucle son 39 ciclos.

Ejemplo 2-13. Desenrollado de un bucle simple

Una de las formas más sencillas de aprovechar esta técnica es combinarla con la planificación de bloques básicos. Por el mero hecho de eliminar las operaciones de saltos se dispone de un número mayor de operaciones con las que poder explotar su paralelismo (Ejemplo 2-14).

Existen cuatro tipos de limitación a la ganancia que puede obtenerse desenrollando un bucle:

- Si se desenrolla demasiadas veces, llega un momento en que desenrollar más no incrementa el paralelismo entre las operaciones. Pero no sólo eso, el límite de unidades funcionales de que dispone la máquina también impide aprovechar el paralelismo más allá de un cierto umbral.
- **Tamaño del código**. Cuantas más iteraciones se desenrolla, más crece el código. Este aumento de tamaño puede causar trastornos en entornos con limitaciones de memoria o simplemente por los fallos de caché de instrucciones.
- **Presión de registros** (*register pressure*). Al combinar el desenrollado con planificación agresiva en bucles muy largos se puede dejar a la máquina sin registros disponibles, lo que es bastante perjudicial, en especial cuando se emiten múltiples operaciones por ciclo.
- Las propias **limitaciones de los compiladores**. En especial al irse complicando las situaciones con la aparición de recurrencias (*loop-carried dependences*) y otras situaciones que requieren análisis bastante complejos.

Si al desenrollado se le suma la aplicación de planificación local:

```
Loop: LF     F1, 0(R1)
      LF     F3, 4(R1)
      LF     F5, 8(R1)
      LF     F7, 12(R1)
      ADDF   F2, F1, F0
      ADDF   F4, F3, F0
      ADDF   F6, F5, F0
      ADDF   F8, F7, F0
      SF     F2, 0(R1)
      SF     F4, 4(R1)
      SF     F6, 8(R1)
      DADDUI R1, R1, #16
      BNE    R1, R2, Loop
      SF     F8, -4(R1)
```

Se consigue un código sin una sola burbuja, y cada iteración tarda exactamente 14 ciclos (los 3 ciclos de más del último SF se solapan con la siguiente iteración).

En resumen:

Método	Ciclos en resolver 1 iteración del bucle básico
Sin desenrollar	12
Sin desenrollar (planificado)	9
Desenrollado	9.75
Desenrollado (planificado)	3.5

Se puede comprobar que la mejora obtenida sumando los beneficios de los dos métodos es bastante notoria.

Ejemplo 2-14. Combinación de desenrollado de bucles y planificación de bloque básico

Software pipelining

El *software pipelining* es una técnica para **reorganizar el código de un bucle** y combinar en cada iteración del nuevo bucle creado operaciones de diferentes iteraciones del código original.

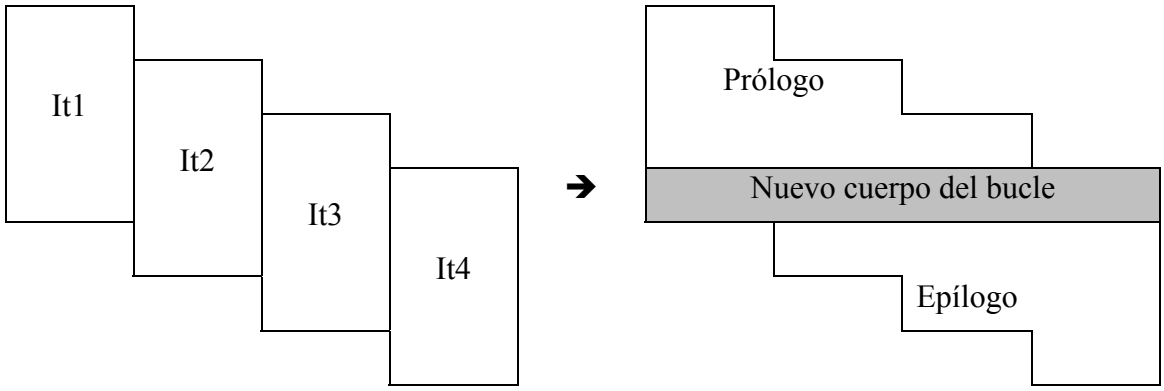


Ilustración 2-7. Software pipelining. Esquema básico de la aplicación de la técnica

Se puede entender esta técnica como un **desenrollado simbólico** del bucle, al estilo de lo que hace el algoritmo de Tomasulo en el hardware. Se toman operaciones de distintas iteraciones del bucle e intentan combinarse en una única iteración con el fin de evitar la aparición de burbujas. Como se ve en la Ilustración 2-7, no basta con construir el código del cuerpo del bucle; también es necesario un código de inicialización (prólogo) y un código de finalización (epílogo). Para ilustrar mejor este método conviene revisar el Ejemplo 2-15.

Aplicando software pipelining al Ejemplo 2-11:

```
// Prólogo
      LF     F1, 0(R1)        ; Se carga X[0]
      ADDF   F2, F1, F0       ; Se suma X[0]
      LF     F1, 4(R1)        ; Se carga X[1]
// Cuerpo
Loop: SF     F2, 0(R1)        ; Se almacena X[i]
      ADDF   F2, F1, F0       ; Se realiza la suma de X[i+1]
      LF     F1, 8(R1)        ; Se carga X[i+2]
      DADDUI R1, R1, #4
      BNE    R1, R2, Loop
// Epílogo
      SF     F2, 0(R1)        ; Se almacena X[1022]
      ADDF   F2, F1, F0       ; Se suma X[1023]
      SF     F2, 4(R1)        ; Se almacena X[1023]
```

Obviando por ahora los códigos de prólogo y epílogo se puede calcular que el tiempo que tarda una iteración del bucle es 6 ciclos. Si además se aplica planificación local:

```
// Cuerpo
Loop: SF     F2, 0(R1)        ; Se almacena X[i]
      DADDUI R1, R1, #4
      ADDF   F2, F1, F0       ; Se realiza la suma de X[i+1]
      BNE    R1, R2, Loop
      LF     F1, 4(R1)        ; Se carga X[i+2]
```

El bucle tarda únicamente 5 ciclos por iteración.

Ejemplo 2-15. Software pipelining

La gran ventaja de esta metodología frente al desenrollado de bucles original es que los códigos obtenidos tienen un **tamaño más reducido**. Sin embargo, hay que tener claro que sus objetivos son distintos: mientras el desenrollado de bucles intenta reducir la sobrecarga de operaciones debida a demasiados saltos o incrementos de contadores, el

software pipelining trata de reducir el tiempo que el bucle no está ejecutándose a pleno rendimiento al principio y el final (prólogo y epílogo).

Las limitaciones al aplicar este método pueden proceder de bucles con pocas operaciones y altas latencias. En ese caso conviene aplicar conjuntamente un desenrollado de bucles.

Un aspecto muy importante de este método es que encontrar una planificación óptima es un problema NP-completo [18]. Por tanto, los métodos generales propuestos son heurísticos y tratan de buscar una solución lo suficientemente buena en lugar de la óptima.

Planificación Global

Las técnicas que se han visto hasta ahora funcionan bien con fragmentos de código que son bloques básicos. Cuando un bucle contiene en su interior otras operaciones de control su planificación resulta significativamente más complicada.

El problema es que el paralelismo que puede obtenerse de los bloques básicos, en especial en los programas de propósito general, que suelen ser altamente impredecibles, no es suficiente para mantener ocupadas las unidades funcionales de una máquina VLIW. La **planificación global** trata de atravesar los límites de los bloques básicos e incluso de subsiguientes iteraciones de un bucle, con lo que puede extraer mucho más paralelismo.

Se dispone del siguiente código:

```
if (w > 0) then
    x = x + 1;
else
    x = x - 2;
y = 2 * x;
z = u * v;
```

Aplicando planificación global:

Moviendo `z = u * v`	Moviendo `x = x - 2`

`z = u * v;`
`if (w > 0) then`
`  x = x + 1;`
`else`
`  x = x - 2;`
`y = 2 * x;`
	`x = x - 2;`
	`if (w > 0) then {`
	`  x = x + 2;`
	`  x = x + 1;`
	`}`
	`y = 2 * x;`
	`z = u * v;`

a) Sin *bookkeeping*	**b) Con *bookkeeping***

El ejemplo **a** muestra como la operación `z = u * v` es independiente de cualquiera de las dos ramas del salto, por lo que cualquier código adicional es innecesario.

En el ejemplo **b**, la operación `x = x - 2` proviene de uno de los dos caminos del salto, por lo que su movimiento hace necesario un código de resguardo en la otra rama del salto: `x = x + 2`.

Ejemplo 2-16. Aplicación de planificación global. Fuente: [23]

Al mover las operaciones debe tenerse mucho cuidado en preservar las dependencias de datos y de control. Sin embargo, para estas técnicas no es suficiente mover operaciones independientes fuera de los límites de un bloque básico; además intentan adivinar el resultado de saltos condicionales para poder adelantar las operaciones del camino más probable. Estos movimientos no son gratuitos, y en muchas ocasiones obligan a añadir

un **código de resguardo** (*bookkeeping code*) para poder deshacerlos en el camino contrario del salto.

El Ejemplo 2-16 incluye una planificación (b) que mueve una operación de un camino de un salto condicional. Estos movimientos de código restringidos a un camino se denominan **especulativos** (al fin y al cabo, la idea es la misma que en 2.6.5). Estos cambios pueden aumentar en gran medida el paralelismo obtenido de un código; sin embargo, su mala aplicación puede provocar el crecimiento innecesario del código e incluso una degradación del rendimiento debida a los códigos de resguardo. Las **técnicas para adivinar el camino más frecuente** son el factor determinante en la eficacia de la planificación global.

Trace Scheduling

Una implementación concreta de la planificación global es la **planificación basada en traza** (*trace scheduling*). Este tipo de planificación se basa en el concepto de **traza**: una traza es un camino de ejecución dentro de un bucle que puede incluir saltos condicionales o puntos de destino de otros saltos (por tanto, NO es un bloque básico).

La técnica se compone de una primera fase en la que el compilador identifica las trazas del código. En esta tarea el desenrollado de bucles desempeña un papel fundamental para disponer de las trazas más largas posibles. Después elige la traza que presupone más frecuente y la planifica como si fuera un único bloque básico. A continuación escoge la siguiente traza más frecuente y así sucesivamente.

Las trazas sucesivas necesitan un código de resguardo para mantener la ejecución correcta del programa. Esta sobrecarga es aceptada porque la mejora obtenida con la planificación de la traza más frecuente debe compensarla.

La utilidad de esta técnica ha sido demostrada en aplicaciones científicas, donde hay un alto número de saltos y se puede disponer de suficiente información sobre ejecuciones previas como para elegir correctamente las trazas más adecuadas. Su aplicación en programas de uso más general no parece tan adecuada de momento, al ser excesiva la sobrecarga añadida por los códigos de resguardo.

Otras técnicas más avanzadas

Otros mecanismos software empleados en las máquinas VLIW son:

- Técnicas de detección de dependencias de datos: *Points-to analysis*, *Interprocedural analysis*...
- Técnicas de eliminación de dependencias de datos: Propagación de copia, reducción del tamaño del árbol, optimizaciones algebraicas...

Estos mecanismos entran más en el terreno de las técnicas de optimización empleadas por un compilador, y no es el objetivo de esta memoria profundizar excesivamente en estos temas.

2.7.2. Mecanismos Hardware

Todas las técnicas vistas en el punto anterior resultan útiles en entornos predecibles por el compilador. Sin embargo, existen multitud de situaciones donde se hacen necesarios componentes hardware que complementen a estas técnicas. Estas situaciones van desde saltos difícilmente predecibles de manera estática a la aparición de dependencias entre referencias de memoria que no se podían saber en tiempo de compilación.

Predicación

El concepto de **operaciones predicadas o condicionales** se basa en asociar a una operación una condición. Esta condición se evalúa como parte de la ejecución de la operación, de tal manera que si resulta ser verdadera se finaliza normalmente la ejecución, pero si resulta ser falsa, la operación se trata como si fuera una no-operación (una operación nula que no realiza ninguna tarea). En el Ejemplo 2-17 se muestra el uso de una operación predicada.

La tarea más compleja es decidir **cuándo anular** una operación. Si se quiere que se anulen pronto pueden ocurrir burbujas a la espera de disponer de los datos para valorar la condición; si se quiere retrasar su anulación se tiene a la operación consumiendo recursos y afectando negativamente al rendimiento de la máquina. En cualquier caso, la opción de adelantar la anulación no suele aplicarse en ningún diseño, ni siquiera al combinar el adelantamiento de resultados con esta técnica.

El siguiente código:

```
if (R1 == 0) then
    R2 = R3;
```

Se pondría en ensamblador como:

```
      BNE   R1, R0, L         ; Recordar que R0 = 0
      ADDI  R2, R3, R0
L:
```

Usando predicación se podría disponer, por ejemplo, de una operación de movimiento condicional, de tal manera que el código anterior se vería sustituido por:

```
      CMOVZ R2, R3, R1
```

De esta forma, se elimina el salto y la sobrecarga que supone su evaluación.

Ejemplo 2-17. Uso de operaciones condicionales

Una ampliación directa de este método, llamada **predicación total** (*full predication*) es usado en muchos procesadores. Frente a limitar el uso de la predicación a ciertas operaciones, como las de movimiento, se extiende su uso a todas las operaciones. De esta forma pueden convertirse largas ramas de un salto a un conjunto de operaciones, todas con el mismo predicado.

La predicación es especialmente útil cuando se emplea conjuntamente con planificación global, ya que puede eliminar saltos que no son bucles y reducir la sobrecarga que produce este tipo de planificación. También se destaca su utilidad para eliminar algunos saltos difícilmente predecibles o simplemente para implementar un flujo de control más corto.

Obviamente no todo son ventajas:

- Las operaciones anuladas siguen consumiendo un mínimo de recursos.
- Su aplicación queda reducida a flujos de control simples, ya que en saltos con numerosas alternativas se vuelve muy costosa.
- Las operaciones condicionales tardan más en ejecutarse que las operaciones que no lo son, debido a la sobrecarga que añade la comprobación de la condición.

Predicación en la arquitectura IA-64

Una aplicación concreta de la predicación total es la que se emplea en la arquitectura IA-64 [15]. En este caso, se dispone de un conjunto de **registros de predicado** que

pueden tomar un valor verdadero (1) o falso (0). Mediante unos bits en un campo de cada operación se indica qué registro de predicado está asociado con esta operación.

El valor de los registros se asigna mediante operaciones de comparación que permiten escoger entre diez tipos de comprobaciones y seleccionar dos registros de predicado como destino. Uno de los registros se activa (1) si la condición es verdadera, y el otro si es falsa. En cualquiera de los casos el registro que no se activa pone su valor a falso (0). El Ejemplo 2-18 propone el uso de la predicación para una condición típica *if-then-else*.

Una consecuencia de la aplicación de esta técnica es que los saltos condicionales se convierten en un salto con un predicado asociado que responde a esa condición.

El siguiente código:

```
if (R1 == 0) then
    R2 = 1; R3 = 2;
else
    R2 =-1; R3 = 3;
```

Empleando predicación:

```
      CMP.EQ      p1, p2, R0, R0; Recordar que R0 = 0
(p1)  DADDUI      R2, R0, #1
(p1)  DADDUI      R3, R0, #2
(p2)  DADDUI      R2, R0, #-1
(p2)  DADDUI      R3, R0, #3
```

La operación de comparación indica los dos registros de predicado: p1 para si la condición es verdadera, y p2 para si la condición es falsa.

Ejemplo 2-18. Uso de predicación en la IA-64

Multiway branching

Una extensión inmediata al uso de la predicación en un entorno VLIW es que, al disponer de un número considerable de unidades funcionales, pueden planificarse las operaciones de los dos caminos de un salto condicional simultáneamente. No sólo eso, dentro de uno de los caminos del salto puede haber a su vez una nueva operación de salto, con lo que el número de caminos se iría incrementando. A esta técnica se le denomina **salto multicamino** (*multiway branching*) y es, de alguna manera, el contrapunto a la ejecución especulativa de las máquinas superescalares (al fin y al cabo, se está especulando mediante la predicación). Su aplicación puede verse en el Ejemplo 2-19.

Retomando el código del Ejemplo 2-18, y suponiendo que se dispone de 2 unidades de suma entera y una para comprobar las condiciones:

UF Suma Entera 0	UF Suma Entera 1	UF Salto
(p1) DADDUI R2, R0, #1	(p2) DADDUI R2, R0, #-1	CMP.EQ p1, p2, R0, R0
(p1) DADDUI R3, R0, #2	(p2) DADDUI R3, R0, #3	

De esta manera se aprovechan los recursos de la máquina que de otra forma no tendrían trabajo mientras se decide el salto.

Ejemplo 2-19. Multiway branching

Especulación

La predicación es un primer mecanismo de especulación mediante el compilador asistido por hardware. Sin embargo, para obtener toda la potencia de la especulación son necesarios otros mecanismos que presten su apoyo en dos grandes problemas: el

control de las excepciones y la posibilidad de adelantar especulativamente la ejecución de operaciones de memoria que pueden tener conflictos de direcciones.

Para manejar el problema de las excepciones existen cuatro alternativas:

- Dejar que el sistema operativo y el hardware colaboren para ignorar las excepciones de las operaciones especuladas. Este método tiene buen resultado en programas correctos, pero no así en programas con errores. Su aplicación tiene sentido en ciertos contextos donde se desea una ejecución "rápida" a costa de perder precisión en el control de las excepciones.
- Hacer que las versiones especulativas de las operaciones no produzcan excepciones, para detectarlas después mediante operaciones de chequeo.
- Emplear unos bits de estado (*poison bits*) para marcar los registros resultado de una operación que causó una excepción. Si una operación especulativa trata de usar este registro, el *poison bit* del registro resultado de esta nueva operación queda marcado también, propagando de esta manera la excepción. La excepción se emite cuando una operación no especulativa trata de usar alguno de los registros marcados.
- Proveer de un mecanismo que indique que la operación es especulativa y que debe ser almacenada en un *buffer* (por ejemplo, un *reorder* buffer) hasta que se sepa con seguridad que ha dejado de serlo.

El otro gran problema es el de las operaciones de acceso a memoria. Los movimientos de estas operaciones los realiza el compilador cuando puede comprobar que las direcciones a las que acceden no van a entrar en conflicto. Sin embargo, restringiéndose a estos casos se pierde muchísima flexibilidad.

Para adelantar la ejecución de un *LOAD* del que no se tiene completa seguridad de los conflictos que pueda causar se introduce un nuevo tipo de operación de chequeo. Esta operación se pone en el lugar que ocupaba el *LOAD*, mientras éste se adelanta indicando que es especulativo. Tras ejecutar el *LOAD* especulativo, si se intenta escribir en la dirección de memoria a la que hacía referencia antes de llegar a la operación de chequeo, la especulación habrá fallado. En caso de fallo pueden ocurrir dos cosas: si sólo se ejecutó el *LOAD* de forma especulativa, basta con rehacerlo; si además se ejecutaron operaciones dependientes del mismo, debe ejecutarse un código de reparación que rehaga todas estas operaciones a partir del *LOAD*.

Especulación en la arquitectura IA-64

El control de las **excepciones diferidas** (*deferred exception*) para las operaciones especulativas se realiza mediante una variante de los *poison bits*: los bits **NaT** (*Not a Thing*) para los registros de propósito general; y un valor especial, **NaTVal** (*Not a Thing Value*), para los registros de punto flotante. Sólo un *LOAD* especulativo puede generar uno de estos valores, aunque cualquier operación puede propagarlos.

Cuando una operación no especulativa intenta usar un registro que tenga este valor como operando fuente genera una excepción inmediatamente. También pueden incluirse operaciones de chequeo en el código (*chk.s*) para detectar estos valores y redirigir la ejecución a una rutina de recuperación, como se ve en el Ejemplo 2-20.

```
ADDI   R5, R6, R8
MULTI  R8, R9, R10
...
BEQ    R1, R2, loop
LI     R1, 0(R3)
...; operaciones que usan R1
```

a) Planificación original

```
LI.s   R1, 0(R3)
...; operaciones que usan R1
ADDI   R5, R6, R8
MULTI  R8, R9, R10
...
BEQ    R1, R2, loop
chk.s
```

b) Planificación especulativa

Ejemplo 2-20. Especulación de un load en la arquitectura IA-64.[12]

Una operación de carga de memoria que adelanta su ejecución de forma especulativa a un *STORE* se llama en esta arquitectura *advanced load*. Cuando se ejecuta un *advanced load* se crea una entrada en una tabla especial denominada *ALAT*. Esta tabla almacena el registro destino del *LOAD* y la dirección de memoria accedida. Cuando se ejecuta un *STORE* se comprueba en esta tabla que la dirección a la que hace referencia no esté en uso. En caso que alguna entrada coincida se marca como incorrecta.

Como en el caso anterior, se añade una operación de chequeo en el lugar donde originalmente estaba planificado el *LOAD* para comprobar que su entrada en la tabla *ALAT* siga siendo válida. Si la entrada no es válida pero sólo se especuló la operación de carga puede utilizarse una operación *ld.c* que intenta cargar de nuevo el dato en este punto. Si se especularon más operaciones junto con el LOAD se emplea una operación de chequeo (*chk.a*) que especifica la dirección de un código de ajuste para rehacer todas las operaciones. El funcionamiento de esta técnica puede verse en el Ejemplo 2-21.

```
ADDI   R5, R6, R8
MULTI  R8, R9, R10
...
SI     R4, 3(R7)
LI     R1, 0(R3)
...; operaciones que usan R1
```

a) Planificación original

```
LI.a   R1, 0(R3)
...; operaciones que usan R1
ADDI   R5, R6, R8
MULTI  R8, R9, R10
...
SI     R4, 3(R7)
chk.a
```

b) Uso de *advanced load*

Ejemplo 2-21. Uso de *advanced load.* En el caso b se ha adelantado especulativamente la ejecución del *LOAD* y una serie de operaciones que dependen de él.

[12] Por comodidad se han empleado los mnemónicos de las instrucciones del repertorio MIPS IV y no los del repertorio original IA-64 en este ejemplo y el siguiente

2.8. Resumen: Diferencias Superescalar – VLIW

Se propone a modo de resumen una tabla con las diferencias principales entre los dos enfoques más destacados de las arquitecturas ILP con emisión múltiple.

Característica	Superescalar	VLIW
Emisión de instrucciones	Aprovechan el mismo flujo del código secuencial para emitir más de una instrucción por ciclo	Mediante software se construyen instrucciones multioperación, donde cada campo controla una UF.
Planificación de la emisión de instrucciones (en general)	Dinámica mediante hardware	Estática mediante software (compilador)
Hardware	Bastante complejo	En general, bastante más simple
Frecuencia del reloj	Más baja	Alta
Compilador	Simple	Muy complejo.
Optimizaciones de código	Simples	El compilador puede ver "más lejos" y obtiene mejores resultados
Portabilidad del código máquina	Código poco dependiente del hardware	Código muy dependiente del hardware
Lenguaje ensamblador	Fácil de utilizar	Escritura directa en código ensamblador inabordable
Comportamiento frente al flujo de control	Mejor en situaciones menos predecibles	Peor ante situaciones poco predecibles
Comportamiento frente a fallos de caché	No tiene ningún problema	Obliga a planificaciones conservadoras o a mecanismos de Load especulativos

Tabla 2-5. Diferencias Superescalar-VLIW. Aquellas características en las que una de las máquinas resulta mejor que la otra han sido coloreadas en la tabla.

2.9. Paralelismo a nivel de thread

Los avances en ILP no han impedido que intenten explotarse otras vías para mejorar el rendimiento de las máquinas. En lugar de tratar de paralelizar la ejecución de instrucciones, el paralelismo a nivel de *thread* (TLP: *Thread-Level Parallelism*) trata de ejecutar ***threads*** (procesos ligeros que representan un "hilo" de ejecución dentro de un proceso) simultáneamente.

Pese a no ser el objetivo de este documento, no puede darse por finalizado este repaso teórico sin nombrar, al menos someramente, estas arquitecturas. Su importancia, en el caso del SMT, es aún mayor al tratarse de una evolución de las máquinas superescalares.

2.9.1. MT: Multithreading

El *Multithreading* (MT) es una técnica para explotar el TLP que permite a varios *threads* compartir las unidades funcionales de un procesador de manera solapada.

El concepto es muy parecido al que se emplea al hablar de **máquinas multitarea**. En ese caso el procesador "desplanifica" un **proceso** en ejecución cuando está esperando por una operación muy costosa en tiempo, como operaciones de E/S o de sincronización. En ese momento se realiza lo que se denomina un **cambio de contexto**: el proceso actual se suspende, salvando su estado completo en un cierto espacio de almacenamiento (algunos registros especiales o memoria), y un nuevo proceso (probablemente también salvado anteriormente) se pone en ejecución. Más adelante el proceso suspendido volverá a ponerse en ejecución siguiendo alguna estrategia predefinida.

En el caso del MT el procesador debe duplicar el estado de cada *thread* para permitir este tipo de ejecución: bancos de registros, PC, tabla de páginas… Además, el hardware debe adaptarse para permitir un cambio de contexto rápido entre *threads*, con un grado de exigencia mucho mayor que en los cambios de contexto habituales entre procesos.

Existen dos aproximaciones principales al *multithreading*:

- ***Multithreading* de grano fino** (*Fine-grained*): Cambia de *thread* en cada instrucción, entrelazando su ejecución. El procesador debe disponer de mecanismos hardware que le permitan cambiar de *thread* en cada ciclo de reloj.
- ***Multithreading* de grano grueso** (*Coarse-grained*): Sólo cambia de *thread* cuando ocurre una burbuja costosa en la ejecución, como un fallo de caché de segundo nivel.

Sus principales virtudes y defectos pueden verse en la Tabla 2-6.

	Grano fino	Grano grueso
Ventajas	Puede ocultar la pérdida de rendimiento debida a burbujas cortas o excesivamente largas.	No suele ralentizar la ejecución de un único *thread*.
Desventajas	Ralentiza la ejecución de un único *thread*.	Limitado en la recuperación de pérdidas de rendimiento, especialmente por burbujas de poca duración. Gran costo inicial, al tener que llenar el *pipeline* cada vez que se cambia de *thread*.

Tabla 2-6. Comparativa entre tipos de *multithreading*.

2.9.2. SMT: Simultaneous Multithreading

Hasta el momento, al hablar de *multithreading* se estaba suponiendo que el procesador tan sólo ejecutaba una instrucción por ciclo, pese a dar la impresión de simultanear la ejecución de varios *threads*. El **SMT** (*Simultaneous Multithreading*) [32] es una variante del *multithreading* que emplea los recursos de un procesador dinámicamente planificado con emisión múltiple para explotar el TLP a la vez que el ILP. La justificación del empleo de esta técnica es que en un procesador de este tipo un único *thread* no suele ser capaz de proporcionar suficientes instrucciones para mantener ocupados todos los recursos de la máquina. Se convierte, por tanto, en una evolución natural de las máquinas superescalares como puede verse en la Ilustración 2-8.

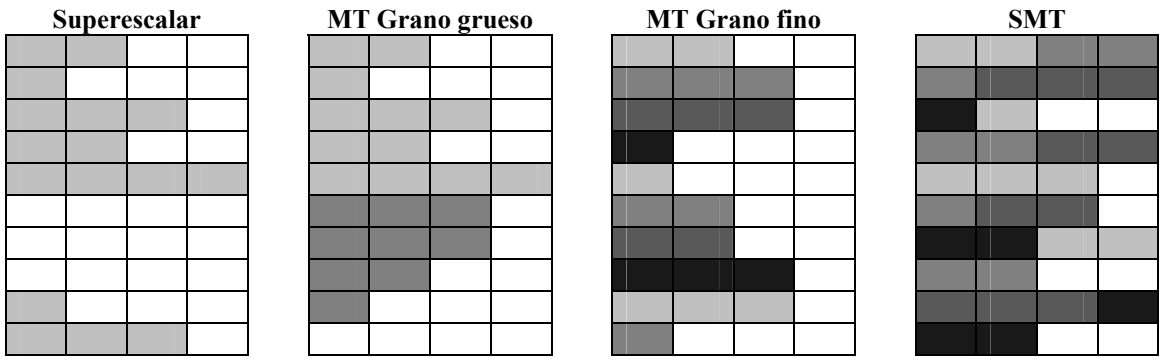


Ilustración 2-8. Cuatro aproximaciones diferentes que usan las posibilidades de emisión de un procesador superescalar. El eje horizontal representa el grado de emisión (en este caso, 4 instrucciones por ciclo) y el eje vertical una secuencia de ciclos de reloj. Los tonos de gris representan distintos *threads*. Fuente: Pág. 610 [13]

Dos componentes del procesador dinámicamente planificado facilitan especialmente la aplicación de esta técnica:

- El renombrado de registros permite que cada *thread* pueda emplear su propio conjunto de registros.
- La graduación independiente de instrucciones de varios *threads* puede hacerse mediante la separación lógica del *reorder buffer* para cada *thread*.

Debido a la profundidad del *pipeline* de un procesador superescalar dinámicamente planificado, la aplicación de SMT no repercutiría en una mejora significativa del rendimiento si el *multithreading* fuese de grano grueso. Sin embargo, una implementación de grano fino repercute negativamente en el rendimiento de un único *thread*. Para minimizar este efecto se definen ***threads* preferidos**, cuya ejecución intenta prolongarse en el tiempo todo lo posible. Esto crea un dilema entre el mantenimiento de la ejecución de un *thread* y el objetivo de aprovechar mejor los "huecos" mediante la planificación simultánea de muchos *threads*.

Este problema en apariencia tan complejo se salva fácilmente gracias a la capacidad de emisión de las máquinas superescalares (en general, de 4 a 8 instrucciones por ciclo). Suele ser suficiente tener un número pequeño de *threads* activos y un número aún menor de *threads* preferidos.

Otros problemas interesantes de diseño son:

- El uso de bancos de registros lo suficientemente grandes para mantener múltiples contextos.
- La sobrecarga en el ciclo de reloj, en especial durante la emisión y la graduación, al disponer de un número mucho mayor de instrucciones entre las que elegir.
- Los conflictos de caché generados por la ejecución simultánea de varios *threads*, que puede degradar considerablemente el rendimiento global del procesador.

Conviene finalizar con dos observaciones importantes a estos problemas:

- En la mayoría de los casos, la sobrecarga que añade el *multithreading* al rendimiento es bastante pequeña.
- La eficiencia de las máquinas superescalares actuales está lo suficientemente limitado como para que una mejora de este tipo sea una opción altamente recomendable.

2.10. Un poquito de historia

El ILP ha estado presente en la arquitectura de computadoras desde hace décadas en la forma de segmentación (*pipelining*). La segmentación comenzó a utilizarse a finales de los años 50 y tuvo una significativa presencia en grandes computadores de los 60. De esta época merece destacarse la CDC 6600 [30], que explotaba el ILP mediante el uso de varias unidades funcionales y en menor medida, de algo de segmentación. Esta máquina tan sólo era capaz mantener una frecuencia de ejecución de una instrucción por ciclo en el mejor de los casos; sin embargo sirvió de base a muchos de los modelos de máquinas superescalares posteriores.

Otro procesador que ha sido de influencia innegable en los procesadores superescalares posteriores fue el IBM 360/91 [4], fuertemente segmentado. Este procesador introdujo

una gran cantidad de innovaciones como el renombrado de registros, la detección dinámica de riesgos de memoria, el adelantamiento de operandos (*forwarding*) generalizado, e incluso la predicción de salto. Además empleó un mecanismo de emisión dinámica de instrucciones conocido como Algoritmo de Tomasulo [31], que también ha servido de base a multitud de procesadores posteriores.

Durante este periodo tan sólo hubo un par de tímidos acercamientos a la iniciación de más de una instrucción por ciclo que no llegaron al mercado. Uno de ellos fue el proyecto ACS de IBM[13][29], con un mecanismo de planificación dinámica más simple que el propuesto por Tomasulo, y que incluso realizaba la búsqueda (*fetching*) de los dos caminos de un salto. Tras varios cambios y traslados de sede, el proyecto fue finalmente cancelado.

La limitación impuesta por la iniciación de una única instrucción por ciclo se convirtió en un cuello de botella para las máquinas segmentadas, y se abrieron nuevos caminos a la mejora del rendimiento, como el procesamiento vectorial. No fue hasta la segunda mitad de los años 80 que se retomaron estas ideas y se propuso un procesador que tomara decisiones de emisión dinámicamente. Este diseño, llamado America (1987) [2], hizo que sus creadores, Tilak Agerwala y John Cocke (parte integrante del grupo que trabajó con ACS), acuñaran el término superescalar. La arquitectura Power1 de IBM (la serie RS/6000) se basó en estas ideas [5].

Otra línea de investigación fue llevada por J. E. Smith (1987) en la universidad de Wisconsin, donde se planteó una aproximación que incluía emisión múltiple con una planificación dinámica del *pipeline* limitada. La idea clave de este diseño fue la inclusión de colas para cada tipo de instrucción, que permitían mantenerlas en orden, y además adelantar o esperar a instrucciones de otros tipos. Estos conceptos se aplicaron a la máquina ZS-1 de Astronautics [26]. La Power2 de IBM también empleó estas colas de forma similar.

La especulación ya estaba presente de una forma muy limitada en el IBM 360/91. Smith y Pleszkun introdujeron el concepto de *Reorder Buffer* en 1988 [25], y Sohi añadió el *renaming* y la planificación dinámica a este esquema en 1990 [28]. Un uso más agresivo de estas técnicas se plasmó en una extensión del algoritmo de Tomasulo, el HPSm [16].

El K-5 de AMD se convirtió en uno de los primeros procesadores superescalares especulativos. Este proyecto estuvo guiado por Johnson, un estudioso del tema del uso de la especulación [17] como una técnica para los procesadores de emisión múltiple junto a Smith y Horowitz [27].

Otras máquinas superescalares más o menos recientes son: toda la gama Pentium de Intel a partir del Pentium Pro (Pentium II, III y 4); K-5, K-6 y Athlon de AMD; UltraSPARC III de Sun; Alpha 21164 y 21264; MIPS R10000, R12000 y R14000; PowerPC 603, 604, 620, G3 y G4; HP 8000...

La primera máquina que puede considerarse VLIW es la Floating Point Systems' FPS-120B, aparecida en el mercado en 1975, y seguida por otras dos máquinas similares: la FPS-164 y la CDC AFP. Sin embargo el término VLIW no fue acuñado hasta 1983 [11], con el famoso prototipo desarrollado en Yale, el ELI-512. Este proyecto partió de la optimización de microcódigo horizontal, y es conocido además por su compilador basado en planificación guiada por traza (*trace-scheduling*), Bulldog. Multiflow

[13] www.cs.clemson.edu/~mark/acs.html

desarrolló para su comercialización sobre 1987 la familia Trace [7] de máquinas, basadas en el ELI-512. Otra línea importante de investigación fue la arquitectura Policíclica [21] que evolucionó hacia productos comerciales como CYDRA-5 de Cydrome sobre 1990 [22]. Curiosamente, estas dos empresas quebraron a principios de los 90. Otros modelos que merece la pena destacar son la iWarp de CMU [8], LIFE de Philips, la StaCs 2.2 de C.E.A. [9] y el TM-1 de Philips en 1996.

En los últimos tiempos ha cobrado nuevo interés el uso de la planificación estática en los diseños de nuevas máquinas, pero intentando tener una solución que aproveche también las innovaciones en hardware. Es esta presencia más activa del hardware la que hace que estas arquitecturas se denominen EPIC (*Explicitly Parallel Instruction Computers*), y se consideren una evolución del esquema VLIW clásico. El ejemplo más claro de este tipo de máquinas es la arquitectura IA-64 [15], fruto de la colaboración entre Intel y HP. Usando esta arquitectura, hay ya dos implementaciones comerciales: los procesadores Itanium[14] e Itanium II[15].

El otro campo en el que se están empleando mucho las arquitecturas VLIW es el mundo de los sistemas embebidos: máquinas en las que la presencia del procesador no es tan evidente como en las computadoras. Dentro de este campo, conviene destacar su aparición en DSPs (como la familia TMS320C6xxx de Texas Instruments) o en procesadores para dispositivos móviles (el chip Crusoe de Transmeta).

En cuanto al *multithreading*, sus orígenes pueden situarse en el TX-2 (1959), uno de los primeros ordenadores construido con transistores y cuya arquitectura se describe en [6]. Su uso se extendió más adelante al procesador de E/S del CDC 6600, mediante un esquema de grano fino. También usó grano fino el procesador HEP [24], un procesador segmentado diseñado por Denelcor y lanzado al mercado en 1982 que no disponía de caché. El procesador Tera [3] extendió todas las ideas empleadas en este diseño.

De finales de los 80 a principios de los 90 se incidió más en el uso del *multithreading* de grano grueso. El procesador SPARCLE [1] del sistema Alewife y el Pulsar de IBM emplearon este tipo de ideas.

A principios de los 90 muchos grupos habían llegado a la conclusión de que la aproximación mediante grano grueso no tenía demasiado futuro. El siguiente paso fue pensar en combinar el *multithreading* con la emisión múltiple para poder usar varias unidades funcionales. La arquitectura XIMD [33] entrelazaba estáticamente *threads* en un procesador VLIW. En el mismo sentido se describieron otros procesadores aunque seguía sin quedar muy claro cómo balancear la carga de las unidades funcionales entre el ILP y el TLP.

A mediados de los 90, con la aparición en masa de los procesadores superescalares planificados dinámicamente, se escogió este tipo de arquitecturas como candidato ideal para combinarse con los conceptos del *multithreading*. El primero en proponer esta opción fue Yamamoto [34] junto con otros investigadores en 1994. Inmediatamente Tullsen, Eggers y Levy continuaron este trabajo para acuñar el término de *simultaneous multithreading* [32]. Este mismo grupo ha seguido trabajando en muchos de los problemas derivados del SMT.

---

[14] http://www.intel.com/products/server/processors/server/itanium

[15] http://www.intel.com/products/server/processors/server/itanium2

ESPECIFICACIÓN DE REQUISITOS

3.1. Requisitos funcionales del simulador

Como primer paso para justificar todas las decisiones de diseño del programa, y más en concreto de las máquinas simuladas, se estableció una serie de requisitos que describieran las funcionalidades de que debía disponer el usuario.

3.1.1. Requisitos acerca del diseño de las máquinas

- Estructura común a las dos máquinas simuladas coherente y robusta, que permita garantizar una base para las comparaciones entre ambas.
- El nivel de detalle de los componentes no debe ser demasiado elevado. Las funcionalidades tienen que quedar ampliamente ilustradas, pero no así su estructura interna.
- Esquema hardware de la máquina VLIW lo más simple posible (ver 4.5).
- Máquina Superescalar con control basado en el algoritmo de Tomasulo (ver 4.6).

3.1.2. Requisitos acerca de los problemas de entrada

- Posibilidad de ejecutar distintos códigos de ejemplo en las máquinas.
- Los códigos de ejemplo se presuponen sencillos y no excesivamente grandes, puesto que su objetivo es poder seguir la traza en la máquina. Por tanto no hace falta un número excesivo de registros o un tamaño demasiado grande de la memoria.
- Creación interactiva de códigos de instrucciones largas para la máquina VLIW. El usuario hará el papel del compilador y, partiendo de un código secuencial, realizará la planificación de las operaciones construyendo manualmente las instrucciones largas. Esto implica la inclusión de herramientas que faciliten su trabajo: manejo de bloques básicos, corrección de códigos…

3.1.3. Requisitos de la simulación

- Capacidad de poder avanzar, parar y reiniciar la simulación tantas veces como se quiera.
- El tiempo del ciclo de reloj no debe ser un parámetro determinante. Al realizar trazas, debe destacarse el número de ciclos utilizados, pero no el tiempo gastado.
- Posibilidad de acceder a todos los componentes de la máquina durante la simulación y, preferiblemente, capacidad de modificar el contenido de estos componentes.

3.2. Requisitos no funcionales del simulador

Las condiciones de usabilidad y apariencia que se consideraron primordiales para el simulador fueron:

- **Configurable**: El usuario debe ser capaz de cambiar los parámetros de las máquinas para conseguir suficiente variedad de experimentos.
- **Visual**: No se quiere un entorno orientado a línea de comandos, sino una herramienta que presente en pantalla el esquema de la máquina a simular y permita observar el flujo de ejecución de las instrucciones.
- **Ampliable**: El simulador debe ser un proyecto abierto que permita posteriores ampliaciones: jerarquías de memoria, más elementos hardware, otras estrategias de predicción de salto…
- **Modularidad**: Para facilitar el requisito anterior, se requiere independizar el diseño de los distintos componentes en la medida de lo posible. No obstante, sí es un objetivo primordial una total independencia del diseño de las máquinas y sus componentes con respecto a la presentación visual de los mismos.
- **Facilidad de uso**: Ya que su objetivo es la docencia, no se debe agobiar al usuario con multitud de opciones de simulación y botones de uso poco frecuente.
- **Documentación accesible**: La ayuda del programa debe ilustrar todos los conceptos empleados en el simulador.

DECISIONES DE DISEÑO

El diseño del simulador necesitaba una reflexión profunda y la combinación de conceptos de distintas fuentes. Puesto que la idea era realizar un simulador de una arquitectura genérica, parecía lógico fijar la atención en las máquinas que se describen en los libros de texto para ilustrar sus ejemplos. La documentación de máquinas reales debía servir para recabar información más concreta sobre la implementación de ciertos mecanismos hardware o software que dotaran de mayor realismo al simulador. El problema de los libros de texto eran las simplificaciones que realizaban para facilitar la presentación de un aspecto concreto de la arquitectura. Estas simplificaciones normalmente presentaban incoherencias e incluso incompatibilidades entre sí cuando se ilustraban conceptos distintos.

4.1. Elección del lenguaje de programación

Ya que se quería emplear un lenguaje visual, había dos opciones:

- Utilizar **programación web**. Pese a las muchas ventajas de la programación web en aspectos como la distribución y accesibilidad del programa, la realidad es que el uso que se iba a dar a esta herramienta no invitaba a la utilización de esta opción. El simulador no estaba planteado como una herramienta con arquitectura cliente-servidor, ni tampoco se pretendía una ejecución distribuida. La idea era que se procesara en la máquina del alumno y no en un servidor. El uso de applets de java tampoco era una opción atractiva debido a la lentitud de ejecución de java por ser interpretado.
- Emplear algún **lenguaje visual**: Visual Basic, Visual C++, Delphi, Builder… De esta forma se perdía en portabilidad pero se ganaba en velocidad del producto final y en sencillez de desarrollo. Esta fue la opción escogida.

La elección de **C++ Builder** como entorno de programación tuvo varias motivaciones principales:

- Basado en **C++**: robusto, potente y eficiente (frente a **Visual Basic**: ineficiente y poco potente; o **Delphi**, basado en pascal).
- Orientado a Objetos: modularidad asegurada.

- Entorno de programación visual muy sencillo (al contrario que **Visual C++**, que trabaja más directamente con la *API* de Windows y es más complejo).
- Aunque **C++ Builder** limita la distribución del simulador a usuarios de Windows, siempre existe la posibilidad de compilarlo usando **Kylix** (versión de C++ Builder para Linux). De esta manera se tendría una versión del simulador para Linux con unas pocas modificaciones.
- La propia experiencia del autor programando con esa herramienta.

4.2. Estrategia de simulación

Las motivaciones para la elección del lenguaje de programación presentadas en el punto anterior tenían que ver básicamente con el diseño del entorno del simulador y las funcionalidades que se pretendía que cumpliera. **C++ Builder** parecía una herramienta más que suficiente para esta tarea.

Sin embargo, no se entraba a valorar la implementación de la simulación de las máquinas en sí. En todo momento se estaba hablando de unas máquinas con un alto grado de paralelismo (a nivel de instrucción), mientras **C++** por sí sólo es un lenguaje totalmente secuencial.

Existen paradigmas de programación que permiten la simulación de eventos paralelos como la simulación orientada a eventos discretos [10]. No obstante, el nivel de detalle al que había que definir los componentes de la máquina para poder implementar uno de estos mecanismos hacía poco práctica su aplicación.

Se buscó por tanto una forma de simular el paralelismo mediante un código secuencial. Para ello había que delimitar perfectamente las etapas por las que pasaban las instrucciones en cada una de las máquinas durante su ejecución. Una vez realizada esta delimitación, bastaba con realizar las etapas en el orden inverso al flujo de ejecución normal. Dentro de cada etapa las tareas sí se harían de forma secuencial. Esta forma de ejecución puede verse más claramente con el Ejemplo 4-1.

Esta función se llamaría cada vez que quisiera avanzarse un ciclo en un *pipeline* diseñado para el esquema de máquina RISC propuesto en la Ilustración 2-3.

```
function ticRISC () {
      writeBack ();
      memoryAccess ();
      execute ();
      decode ();
      fetch ();
}
```

En cada etapa se comprobaría si tiene o no una instrucción que procesar; se efectuarían las tareas asociadas a esa etapa con esa instrucción; y se pasaría la instrucción a la siguiente etapa.

Con esta forma de ejecutar se garantiza que cuando una instrucción pasa por una etapa (por ejemplo de ejecución – *execute* –) ya se ha liberado la etapa siguiente (siguiendo con el ejemplo, de acceso a memoria – *memoryAccess* –).

Ejemplo 4-1. Simulación de un *pipeline* en un esquema de máquina RISC

4.3. Características comunes

Una cuestión fundamental era definir qué elementos y características serían comunes a las dos máquinas. Cuantos más elementos en común, más sencilla la programación de las máquinas y mayor modularidad.

4.3.1. Palabra de 32 bits

Hacer la máquina de 32 bits fue una de las decisiones más sencillas. Bastó con comprobar que se disponía de tipos definidos para flotante y entero en C++ (int y float) de ese tamaño. No se consideró necesario emplear un tipo de datos con mayor precisión o rango debido al uso que se le iba a dar al simulador.

4.3.2. Repertorio de instrucciones

El repertorio de instrucciones debía satisfacer varias condiciones:

- Ser fácil de entender y de usar por los alumnos.
- Preferiblemente basado en un repertorio conocido.
- Válido para las dos máquinas con el mínimo número de cambios posible.
- Intérprete de fácil implementación.

Se decidió construir un repertorio inspirado en **MIPS IV**. Este repertorio tiene un subconjunto de instrucciones muy sencillas que se emplean continuamente en [13].

Sintaxis	Instrucción	Resultado
DADDUI Rd, Ro1, #Inm	Suma Entera con Inmediato sin signo	Rd = Ro1 + Inm
ADDI Rd, Ro1, Ro2	Suma Entera	Rd = Ro1 + Ro2
MULTI Rd, Ro1, Ro2	Multiplicación Entera	Rd = Ro1 * Ro2
ADDF Fd, Fo1, Fo2	Suma Punto Flotante	Fd = Fo1 + Fo2
MULTF Fd, Fo1, Fo2	Multiplicación Punto Flotante	Fd = Fo1 * Fo2
LI Rd, Inm(Ro)	Load Entero	Rd = MEM[Ro + Inm]
LF Fd, Inm(Ro)	Load Flotante	Fd = MEM[Ro + Inm]
SI Ro, Inm(Rd)	Store Entero	MEM[Rd + Inm] = Ro
SF Fo, Inm(Rd)	Store Flotante	MEM[Rd + Inm] = Fo
BNE Ro1, Ro2, Etiqueta	Salta si distinto	Si (Ro1 != Ro2) Saltar a instrucción apuntada por Etiqueta
BEQ Ro1, Ro2, Etiqueta	Salta si igual	Si (Ro1 == Ro2) Saltar a instrucción apuntada por Etiqueta

Tabla 4-1. Repertorio de instrucciones del simulador

Sólo son necesarias un par de aclaraciones acerca del repertorio:

- No se incluyeron instrucciones de salto incondicional ya que podían ser fácilmente simuladas mediante los saltos condicionales (BEQ R0 R0 etiqueta).
- Tampoco hay ninguna instrucción de fin de programa. Al ser la máquina monoprograma, el fin del programa se controla al no encontrar más instrucciones que ejecutar.
- En el repertorio MIPS las instrucciones de precisión simple de punto flotante de simple precisión (32 bits) se indican con el sufijo “S”. Al ser poco intuitivo se decidió sustituirlo por “F”.

4.3.3. Modos de Direccionamiento

Como se vio en el repertorio de instrucciones, sólo se incluyó el direccionamiento con desplazamiento para acceder a memoria. Se decidió asimismo no permitir el acceso a bytes de la memoria o a medias palabras, sólo a palabras completas. Los "índices" por tanto se corresponden con palabras de memoria y no con bytes, como se ve en el Ejemplo 4-2.

```
DADDUI R1, R0, #4
LF     F0, (R1)   // F0 carga la 4ª palabra de memoria
LF     F1, 1(R1)  // F1 carga la 5ª palabra de memoria
LF     F2, 4(R1)  // F2 carga la 8ª palabra de memoria
```

Ejemplo 4-2. Direccionamiento de memoria

4.4. Componentes genéricos

Existen una serie de componentes cuya presencia era totalmente necesaria en los dos tipos de máquina.

4.4.1. Memoria

El esquema de la memoria no era una prioridad en este proyecto. Sin embargo, su gran influencia en el rendimiento general de la máquina hizo necesaria una correcta definición de su funcionamiento, aun cuando no se pretendía definir en detalle su estructura.

Básicamente el esquema aplicado consistió en una memoria principal, una caché de instrucciones que contuviese completamente el programa y una caché de datos con acceso controlado mediante una probabilidad.

Memoria principal

La memoria principal no necesitaba un tamaño variable o excesivamente grande, así que se estableció a **1024 palabras de 32 bits**. El tiempo de acceso a esta memoria se corresponde con el tiempo de fallo de caché.

Caché de instrucciones

La caché de instrucciones en principio debía diseñarse de forma diferente para la máquina Superescalar (con instrucciones "cortas") y la máquina VLIW (con instrucciones largas). Sin embargo, el nivel de detalle hardware exigido permitía un diseño común atendiendo solamente a una serie de características:

- **Sin fallos** de caché de instrucciones. El programa (secuencial o de instrucciones largas) cabe siempre completamente en esta caché.
- El código del programa se accede a partir de la **posición 0** de memoria en cualquier caso, por lo que los saltos son a direcciones absolutas y no necesitan ningún tipo de cálculo adicional.
- El acceso a las instrucciones en la caché es **inmediato** (1 ciclo)

Caché de datos

Los detalles de su estructura o su tamaño no se definieron, sino que se controló su acceso mediante un **porcentaje de fallos** cuyo valor podía modificar el usuario.

4.4.2. Buffer de instrucciones

Debido a lo dicho en el punto anterior acerca de la ausencia de fallos de caché de instrucciones no era necesaria la inclusión de estructuras tales como un *buffer* de instrucciones. Una estructura de este tipo únicamente añadiría una etapa adicional por la que tendrían que pasar las instrucciones, pero carecería de ninguna utilidad práctica para el alumno y complicaría el esquema general.

4.4.3. Registros de Propósito General (GPR)

Los registros de Propósito General son básicos en el esquema de cualquier máquina Superescalar o VLIW. Puesto que no se intentaba ejecutar grandes programas se decidió fijar el **número de registros a 64**, más que de sobra para el tamaño habitual de los códigos de ejemplo. También se adoptó el convenio de muchas máquinas de fijar el **valor del primer registro a 0**, con lo que se tendría una referencia continua. Con **32 bits** por registro quedaron cubiertas las necesidades de rango de los valores empleados en los códigos de ejemplo.

Tanto en el juego de instrucciones como en cualquier parte del programa o este documento se hará referencia a estos registros como **R0, R1, …, R63**.

4.4.4. Registros de Punto Flotante (FPR)

Como complemento a los registros de Propósito General se añadieron registros para manejar datos en formato de Punto Flotante. Al igual que en el caso de los GPR, su número quedó establecido en **64 registros**. Como no había necesidad de manejar números muy grandes se dejó que estos registros fuesen de **simple precisión** (32 bits).

Tanto en el juego de instrucciones como en cualquier parte del programa o este documento se hará referencia a estos registros como **F0, F1, …, F63**.

4.4.5. Unidades Funcionales (UF)

Las unidades funcionales o **unidades de ejecución** son la unidad fundamental de ejecución de las máquinas. Se perseguían varios objetivos:

- Permitir varios tipos de UF.
- Estructura sencilla.
- Facilitar la ampliación con nuevos tipos de UF.
- Control de la UF simple.

En definitiva, una UF simple y fácilmente adaptable y modificable.

Para poder aprovechar las posibilidades de las máquinas con arquitectura ILP las UF debían ser **segmentadas**, de manera que soportaran un flujo más constante de instrucciones. Se propusieron dos formas de segmentar las UF:

- Asignar a todas las etapas del pipeline la misma duración y dejar como parámetro el número de etapas de cada UF.
- Dejar fijo el número de etapas del pipeline de cada UF y permitir asignar una duración específica a cada etapa.

Se decidió no entrar en los detalles del funcionamiento interno de la UF para evitar cualquier problema derivado de la excesiva especialización de una UF. Por tanto, se descartó la segunda opción. Se estableció la duración de cada etapa a un ciclo y se dejó

en manos del usuario definir cuántas etapas iba a tener cada tipo de UF. De esta manera quedó fijado el **tiempo de ejecución de una UF en su número de etapas**. A este parámetro se le denominó por simplicidad **latencia** de la unidad funcional, ya que equivalía al número de ciclos que tenía que esperar una instrucción para ejecutarse si se planificaba después de otra instrucción con la que tuviera una dependencia verdadera.

La especialización de las UF debía basarse en las instrucciones que pudieran ejecutar, así que se decidió conjuntamente con el juego de instrucciones. Los tipos de unidades funcionales que quedaron se pueden ver en la Tabla 4-2:

Unidad Funcional	Instrucciones asociadas	Latencia en los ejemplos
Suma entera	ADDI, DADDUI	1
Multiplicación entera	MULTI	3
Suma flotante	ADDF	4
Multiplicación flotante	MULTF	6
Memoria	LI, SI, LF, SF	4 (9 con fallo de caché)
Salto	BNE, BEQ	2

Tabla 4-2. Tipos de Unidades Funcionales. Los valores de latencia son los que se emplearán en todos los ejemplos a menos que se indique lo contrario.

El **número de UF** de cada tipo era uno de los parámetros más interesantes de cara a la experimentación por parte del alumno, así que se dejó como un parámetro que pudiera establecer el usuario antes de cada simulación.

4.4.6. Contador de Programa (PC)

El contador de programa también se incluyó en esta relación de elementos comunes ya que tiene la misma función en las dos máquinas, si bien apunta a instrucciones secuenciales en el caso del código para la máquina superescalar, y a instrucciones largas en la VLIW.

4.5. Máquina VLIW

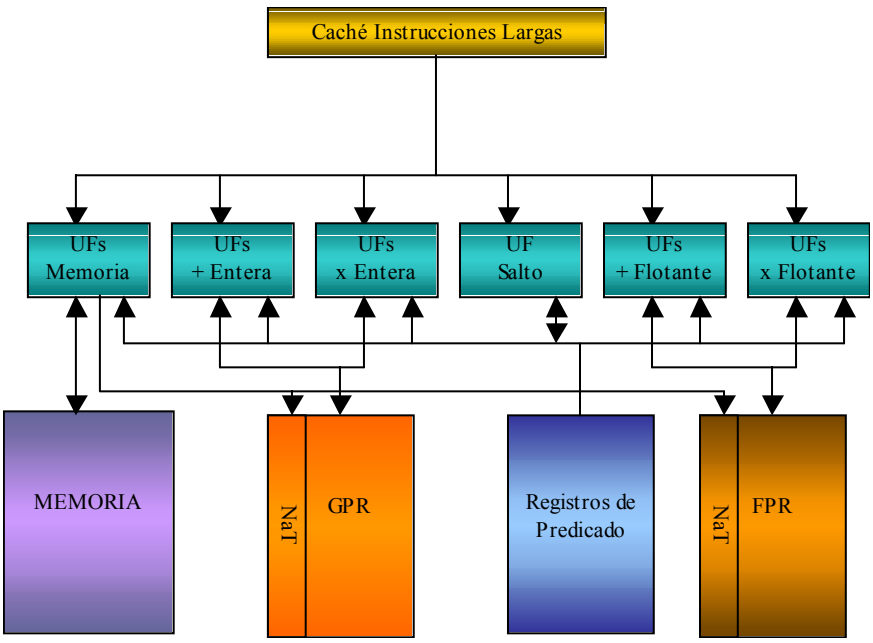


Ilustración 4-1. Esquema básico de la máquina VLIW

La máquina VLIW fue la primera en diseñarse debido a que su hardware debía ser el más sencillo. Se tuvo en mente dos conceptos durante todo el diseño de esta máquina: "**simplicidad**" y "**estático**". Un esquema de la máquina puede verse en la Ilustración 4-1.

En toda esta sección en la que se habla de máquinas VLIW hay que tomar un convenio de nombres ligeramente distinto al empleado hasta ahora para evitar confusiones con los conceptos: así se empleará el término **operación** para referirse a las instrucciones secuenciales originales (las que son ejecutadas en las UF); y se empleará indistintamente **instrucción** o **instrucción larga** para hacer referencia a las instrucciones largas compuestas de operaciones y que componen el código VLIW.

4.5.1. Idea original

Antes de comenzar a detallar cómo se diseñó definitivamente el modelo de máquina VLIW propuesto conviene aclarar que la idea original distaba un poco del resultado final.

En un principio se pretendía basar todo el diseño en la arquitectura **IA-64** [15]. Esta arquitectura resultaba especialmente interesante al constituirse, mediante dos implementaciones en el mercado (**Itanium** e **Itanium II**), en una de las más interesantes alternativas actuales dentro de las máquinas VLIW, y más en concreto dentro de las arquitecturas EPIC.

La arquitectura IA-64 emplea un conjunto de elementos que, en principio, parecían facilitar su implementación y hacer más sencilla para un alumno la tarea de aprendizaje[16]. Sin embargo, a medida que se fue profundizando en los problemas concretos de diseño empleando este esquema se hicieron patentes varios motivos para desechar este modelo:

- El uso de ***bundles*** como paquetes de tamaño fijo de operaciones, y de marcas de *stop* para delimitar los grupos de operaciones implicaba un algoritmo de control del flujo de ejecución bastante complejo.
- Las operaciones de salto condicional requerían ser sustituidas por operaciones de comparación seguidas de saltos incondicionales, lo que suponía la modificación del repertorio original de instrucciones secuenciales.
- Las instrucciones de acceso a memoria (*LOAD*) requerían, para aprovechar la potencia de la máquina, la implementación de un nuevo tipo de operación: los **LOAD avanzados** (*advanced load*). Este tipo de operaciones implicaban la inclusión de una nueva estructura: la tabla **ALAT**.
- Y como aspecto más importante: la carga hardware de una máquina de este estilo implicaba al final un esquema casi tan complejo como el de la máquina superescalar propuesta (o incluso más).

Al final se optó por un esquema mucho más sencillo, al estilo de la Trace 7/200 [23], que facilitara la implementación de la máquina y que, sobre todo, **destacara las diferencias** claras entre estas arquitecturas (eminentemente estáticas) y las arquitecturas superescalares (dinámicas).

---

[16] De hecho, es uno de los ejemplos que se desarrolla con mayor grado de detalle en el capítulo cuarto de [H&P-03]

De cualquier manera, algunos conceptos de la IA-64 sí se aprovecharon, como la **predicación** (ver 4.5.5) a la hora de resolver los saltos condicionales, o el control de los fallos de acceso a memoria empleando bits **NaT** (ver 4.5.6).

4.5.2. Adaptación de los registros

Pese a las premisas iniciales, lo primero que hubo que plantearse durante el diseño de esta máquina fue un cambio en uno de los componentes genéricos: los registros (tanto FPR como GPR).

Aunque de cara al usuario todas las unidades funcionales avanzan en sus etapas simultáneamente con cada ciclo de reloj, realmente este avance se realiza unidad funcional a unidad funcional (recordar el uso de un falso paralelismo en la estrategia de simulación descrita en 4.2). Además hay que partir de la base de que la ejecución dentro de cada unidad funcional no está descrita, es decir, no puede saberse exactamente en qué ciclo se lee un operando o cuándo se escribe el resultado. Esta falta de detalle obliga a que durante toda la ejecución de una operación sus operandos no alteren su valor. Por supuesto, esto provoca que el rendimiento total de la máquina se vea perjudicado (ver Ejemplo 4-3).

Si se dispone del siguiente código secuencial, que tiene una dependencia WAR producida por el registro R4:

```
MULTI R2, R4, R1
ADDI R4, R3, R5
```

Si no se dispone de ningún detalle acerca del funcionamiento de las unidades funcionales, no puede saberse en qué momento se leerá el registro R4 para la operación de multiplicación. Esto provoca que durante los tres ciclos de ejecución no pueda planificarse la suma entera.

UF Suma 0	UF Multiplicación 0	Resto UFs
	`MULTI R4, R3, R5`	…
		…
		…
`ADDI R2, R4, R1`		…

Si se intentara definir más en detalle el funcionamiento de la UF de multiplicación y se supusiera, por ejemplo, que los registros fuente se leen en el primer ciclo, el mismo código podría planificarse de la siguiente manera:

UF Suma 0	UF multiplicación 0	Resto UFs
	`MULTI R4, R3, R5`	…
`ADDI R2, R4, R1`		…

Se observa claramente la pérdida de rendimiento de la máquina en la primera planificación ya que, aunque la multiplicación acabaría el mismo ciclo, la suma acabaría 2 ciclos más tarde.

Ejemplo 4-3. Dependencias WAR en código VLIW

Pero esta pérdida de rendimiento no era un problema por sí misma. El verdadero problema surgía por ciertas planificaciones aparentemente equivalentes, pero que podían llevar a errores. En el Ejemplo 4-4 se observa más claramente este conflicto.

Para solucionar esto, se recurrió a una solución inspirada en un mecanismo que se introdujo en [14]. En ese texto, para explicar la segmentación, se definía un banco de registros que realizaba la escritura de los registros en la primera mitad del ciclo, y la

lectura, en la segunda mitad para permitir el adelantamiento de resultados. En este caso concreto interesaba justamente lo contrario: **leer los registros en la primera mitad del ciclo** y **escribir en la segunda mitad del ciclo**. De esta manera, se evitaban los riesgos WAR, al asegurar que los operandos se leerían antes de ser escritos en el mismo ciclo.

Si se dispone del siguiente código secuencial, con una dependencia WAR entre las dos operaciones en el registro R4.:

```
ADDI R2, R4, R1
ADDI R4, R3, R5
```

La siguiente configuración de la instrucción larga no causaría ningún problema:

UF Suma 0	UF Suma 1	Resto UFs
`ADDI R2, R4, R1`	`ADDI R4, R3, R5`	…

Esto es debido a que se ejecutaría primero la operación que va primera en el orden de programa.

Si embargo, si se cambiara la UF donde se ejecutaran estas operaciones:

UF Suma 0	UF Suma 1	Resto Ufs
`ADDI R4, R3, R5`	`ADDI R2, R4, R1`	…

Se ejecutaría primero la segunda operación, con lo que el valor de R4 no sería el correcto.

Ejemplo 4-4. Justificación de la adaptación de los registros

Para implementar esta solución se introdujo un *buffer* de entrada en los registros que almacenara los valores que se escribían en cada registro. Sólo al finalizar el ciclo de ejecución se hacían efectivas estas escrituras. Se permitió que el uso de este *buffer* fuera opcional puesto que en el caso de la máquina superescalar era completamente innecesario (en ese caso, este tipo de errores se controla con el propio mecanismo de *renaming* del algoritmo de Tomasulo).

4.5.3. Código de instrucciones largas

El código de la máquina VLIW necesitaba satisfacer una serie de necesidades:

- El código lo construye el usuario, no un compilador. Por tanto, debía ser **fácil de construir manualmente**.
- Se quería que las operaciones permanecieran preferiblemente **sin cambios** al pasar de un código secuencial al código de instrucciones largas.
- Quería evitarse el uso de cualquier hardware superfluo. Para ello, lo mejor era que la **estructura de las instrucciones fuese lo más sencilla posible**.

Teniendo esto en cuenta se decidió hacer un **formato de instrucción completamente horizontal**, es decir, que hubiese una correspondencia directa entre los recursos (UF) de la máquina y las operaciones de la instrucción. En definitiva, la instrucción larga debía, en el mejor de los casos, ser capaz de llenar completamente el flujo de ejecución de la máquina en cada ciclo. Este formato permitía además **limitar a una el número de instrucciones por ciclo**, con lo que se obtenía una importante simplificación del hardware de búsqueda e incluso decodificación de la instrucción (de hecho, se prescindió completamente de él).

En cuanto a las modificaciones hechas a las instrucciones secuenciales para convertirlas en operaciones, sólo hizo falta incluir la opción de predicación. Para ello, se asoció un

registro de predicado a todas las operaciones, y dos registros más de predicado a las operaciones de salto (para más detalle ver sección 4.5.5).

4.5.4. Operaciones de salto

El uso de saltos condicionales en los códigos de instrucciones largas (algo aparentemente inocuo) rompía completamente con la idea de la planificación estática de las máquinas VLIW. La aparición de una operación de salto condicional implicaba tener que finalizar su ejecución antes de saber la siguiente instrucción en leerse de memoria. Usando la predicación se solucionaba parcialmente este inconveniente, ya que muchos saltos podían obviarse. Sin embargo, ante la presencia de bucles, no se podía prescindir del salto de ninguna manera[17].

El primer problema que surgió era descubrir cuál debía ser el **destino de un salto** al pasar de instrucción secuencial a operación en un código de instrucciones largas. No debe olvidarse que en el código secuencial el destino de los saltos es otra operación, pero en una máquina VLIW no se ejecutan operaciones individuales sino instrucciones completas. Esto obligaba a **indicar el destino de los saltos explícitamente**. El Ejemplo 4-5 clarificará esta explicación.

Se parte del siguiente código secuencial:

```
.
.
Loop: ADDI   R1, R0, R3
      // Operaciones bucle
      .
      DADDUI R2, R2, #1
      BNE    R2, R5, Loop
.
.
```

Se observa cómo el destino del salto marca la primera operación del bloque básico. Al planificar el código como instrucciones largas no hay porqué respetar el orden relativo de las operaciones a menos que haya alguna dependencia verdadera entre ellas, así que nada impediría realizar una planificación como:

UF + Entera	Otras UFs	UF Salto
DADDUI R2, R2, #1	…	
Operaciones Bucle		
ADDI R1, R0, R3	…	BNE R2, R5, ¿?

¿Cuál sería ahora el destino del salto? No sirve dejar como destino la operación original. Lo lógico sería convertir en destino del bucle la primera operación planificada del bloque básico. Además debe asegurarse que en ninguna de estas instrucciones haya operaciones de otro bloque básico que no deban ejecutarse.

Ejemplo 4-5. Operaciones de salto en código VLIW

Otro problema delicado era el del **número de operaciones de salto permitido por instrucción larga**. Muchas arquitecturas hablaban de instrucciones con 2, 4… operaciones de salto controladas mediante prioridades u otros mecanismos [23]. Estos

[17] Por supuesto que muchos bucles pueden reducirse e incluso eliminarse con técnicas como el desenrollado de bucles. Pero si el bucle tiene el suficiente número de iteraciones al final siempre habrá que añadir un salto condicional para repetir una porción del código VLIW.

mecanismos no eran excesivamente complejos de implementar, pero presuponían añadir información adicional a las operaciones y un trabajo extra del usuario-compilador.

De cualquier manera, incluso los mecanismos de prioridades son meros artificios para decidir, en última instancia, una única instrucción larga de destino. Se consideró que el interés de este mecanismo, en contraposición a la complicación que le añadía al usuario no hacía provechosa su implementación. La decisión final fue no complicar más la máquina y **limitar a uno** el número de operaciones de salto por instrucción larga.

4.5.5. Predicación

Debido a la restricción de simplicidad de la máquina VLIW no parecía una buena idea añadir mecanismos de predicción de salto o ejecución especulativa, ya que suponían una carga de hardware demasiado elevada. Sin embargo, parecía necesario el uso de algún mecanismo que permitiese aprovechar mejor los saltos. La solución fue la predicación, y más en concreto, **predicación total** (ver 2.7.2), que además permitía la ejecución de operaciones de los dos caminos de un salto simultáneamente (***multiway branching***).

Para su aplicación, el aspecto fundamental era asociar a cada operación un **Registro de Predicado**. Se usó un total de **64 registros** (por homogeneidad con el resto de bancos de registros) a los que se denominó **p0, p1, …, p63**. Estos registros sólo podían tener valor verdadero o falso. Debido a que un registro que se ponga a falso anularía la ejecución de una operación y a que no todas las operaciones debían ser predicadas (la mayoría de las operaciones se ejecutan incondicionalmente), se estableció que el registro **p0 tuviese siempre el valor verdadero** y fuese el registro de predicado por defecto de todas las operaciones. Se estableció el valor por defecto del resto de registros de predicado a falso.

La adaptación de las operaciones de salto para soportar predicación se enfocaba en algunas fuentes [15] como un intento de convertir cada operación de salto condicional en una operación de comparación seguida de un salto incondicional debidamente predicado. Esta solución tenía dos inconvenientes principales: había que cambiar el repertorio de instrucciones, ampliándolo con instrucciones de comparación y saltos incondicionales; y el paso de código secuencial a código VLIW se volvía más complicado para el usuario.

La opción que se escogió fue **indicar explícitamente dos registros de predicado** (para el camino verdadero y el camino falso) que asociar a la operación de salto condicional. En el momento de comenzar la ejecución de una operación de salto se pondrían a falso estos dos registros de predicado. Una vez obtenido el resultado del salto, se pondría a verdadero el registro de predicado del camino escogido y a falso el otro. La ejecución de aquellas operaciones que tuvieran asociado el registro de predicado del camino no escogido se anularía. La forma de aplicar este mecanismo puede verse con mayor claridad en el Ejemplo 4-6.

Sin embargo, el uso de este mecanismo produce una limitación importante en los códigos generados: no se puede asignar un registro de predicado a una operación que termine su ejecución antes que el salto que declaró ese registro de predicado, tal como se ve en el Ejemplo 4-7. Si se hiciera esto, la operación nunca se ejecutaría ya que, al no haber terminado la ejecución del salto, aún no se habría asignado el valor verdadero a ninguno de los dos registros. Este problema tiene un agravante si se recuerda la forma de realizar la simulación y las complicaciones que esto producía, tal como se vio en 4.5.2. La solución a este problema fue obligar a evaluar antes que ninguna otra la unidad funcional de salto.

El siguiente código:

```
while (R1 == 0) do {
    F2 = F2 + F1;
    F3 = F3 + F4;
    // más instrucciones del bucle
}
F2 = F8 + F6;
F5 = F4 + F1;
```

Se planificará como instrucciones largas:

ID	UF + flotante 0	UF + flotante 1	Otras UFs	UF Salto
i	// más instrucciones del bucle			
j	**(p1)** ADDF F3, F3, F4	**(p2)** ADDF F2, F8, F6	...	**(p0)** BEQ R1, R0, i **(V = p1, F = p2)**
j+1	**(p2)** ADDF F5, F4, F1	**(p1)** ADDF F2, F2, F1	...	

Como se observa, la operación de salto tiene asignados el registro p1 como predicado para el camino verdadero y p2 para el falso. Todas las operaciones están predicadas, incluido el propio salto, aunque éste con p0 (siempre verdadero).

Los dos ciclos que tarda en resolverse el salto se aprovechan para planificar operaciones de las dos ramas del salto.

Ejemplo 4-6. Uso de predicación en la máquina VLIW diseñada

Se sustituyen las operaciones de suma flotante del ejemplo anterior por sumas enteras (que tardan sólo 1 ciclo en resolverse):

ID	UF + entera 0	UF + entera 1	Otras UFs	UF Salto
i	etiq: // más instrucciones del bucle			
j	(p1) ADDI R3, R3, R4	(p2) ADDI R2, R8, R6	...	(p0) BEQ R1, R0, i (p1 = V, p2 = F)
j+1	(p2) ADDI R5, R4, R1	(p1) ADDI R2, R2, R1	...	

Las dos operaciones de suma de la instrucción *j* se resuelven en un único ciclo, y por tanto, antes de haber evaluado el salto. Esto implica que no pueden ser planificadas en esa instrucción.

Sin embargo, las operaciones de la instrucción *j+1* no deberían tener problema siempre que primero se comprobara la unidad funcional de salto (asignando correctamente el valor de los registros de predicado) y después se terminara la ejecución de estas operaciones.

Ejemplo 4-7. Problemas de la predicación en la máquina VLIW diseñada

4.5.6. Bits NaT

Otra cuestión que no tenía fácil solución era la de los fallos de caché. Una de las características más apreciadas de una máquina VLIW es su **previsibilidad**: en condiciones normales puede asegurarse que el mismo código tardará siempre lo mismo en ejecutarse. Esta afirmación se cumple hasta que comienza a tenerse en cuenta aspectos dinámicos de la ejecución, como los fallos de caché de datos. El tiempo de

cualquier acceso de lectura a memoria puede variar en función de si ocurre un fallo o no. Evidentemente esta cuestión podría soslayarse si siempre se planificaran las operaciones suponiendo accesos a memoria principal, pero no es menos evidente que esta solución no es, ni mucho menos, deseable.

Era indispensable disponer de un **mecanismo que detectara los fallos de caché y actuara en consecuencia**, pero no se quería añadir una excesiva complejidad al esquema básico de la máquina.

Para la primera parte, la detección del fallo, se asoció a cada registro (GPR y FPR) un bit **NaT**[18] (*Not a Thing*). Este bit se activaría cuando un registro se convirtiera en destino de una operación de carga de memoria y permanecería activo hasta que la operación hubiese finalizado correctamente.

La segunda parte de la acción era qué hacer cuando se intentaba leer un registro con este bit activo. En ese caso se sabía que el dato aún no estaba disponible y por tanto no podía ejecutarse esa operación que estaba intentando leer el registro. La forma más sencilla de actuar era simplemente **detener el flujo de ejecución** en este punto hasta tener el valor disponible. De esta manera se aseguraba que no se podía ejecutar ninguna operación que directa o indirectamente dependiera de los resultados de esa carga. En el instante en que el dato estuviese disponible se podría continuar con la ejecución normal.

4.6. Máquina Superescalar

Si bien el hardware de la máquina VLIW era más sencillo, esto no quiere decir que su diseño fuese menos problemático que el de la máquina Superescalar. De hecho, la fuerte restricción de un hardware simple se convirtió en una pesada carga durante todo el diseño. La máquina Superescalar, por el contrario, tenía la ventaja de que no había que inventar nada; solamente era necesario aplicar los conceptos del algoritmo de Tomasulo e implementar los elementos hardware que allí se nombraban. Un esquema sencillo de los componentes que se implementaron puede verse en la Ilustración 4-2.

4.6.1. Grado de emisión

Al contrario que en la máquina VLIW, donde el número máximo de instrucciones[19] emitidas por ciclo queda establecido por el número de unidades funcionales, en la máquina superescalar este parámetro tenía que ser fijado con un valor determinado. Para facilitar la creación de nuevos experimentos y aprovechar la posibilidad de cambiar el número de unidades funcionales, se dejó en manos del usuario establecer el **grado de emisión** entre 2 y 16 instrucciones por ciclo. Este mismo valor serviría también para determinar el número máximo de instrucciones que se graduarían por ciclo de reloj.

4.6.2. Salto condicional

Para tratar el problema de los saltos condicionales se estableció en primer lugar el tipo de estrategia de salto que quería emplearse. Se quería un mecanismo de predicción dinámica de salto de fácil implementación, así que se optó por el uso de una tabla de 2

---

[18] La solución originalmente no contempla el uso de bits NaT para los registros de Punto Flotante. En ese caso se asigna el valor NaTVal (*Not a Thing Value*) al registro. Extender el uso de los bits a los dos tipos de registros es una forma de simplificar el diseño del banco de registros.

[19] En esta sección se vuelve a la nomenclatura normal en la que se hablará indistintamente de **instrucciones** u **operaciones**.

bits (ver sección 2.6.4). Como los códigos de ejemplo se presuponían pequeños, se optó por un tamaño de tabla bastante reducido: 16 entradas (4 bits de indexación).

La implementación de la tabla no presentó mayores problemas y su control se realizó actualizando la predicción con cada acceso a una instrucción de salto.

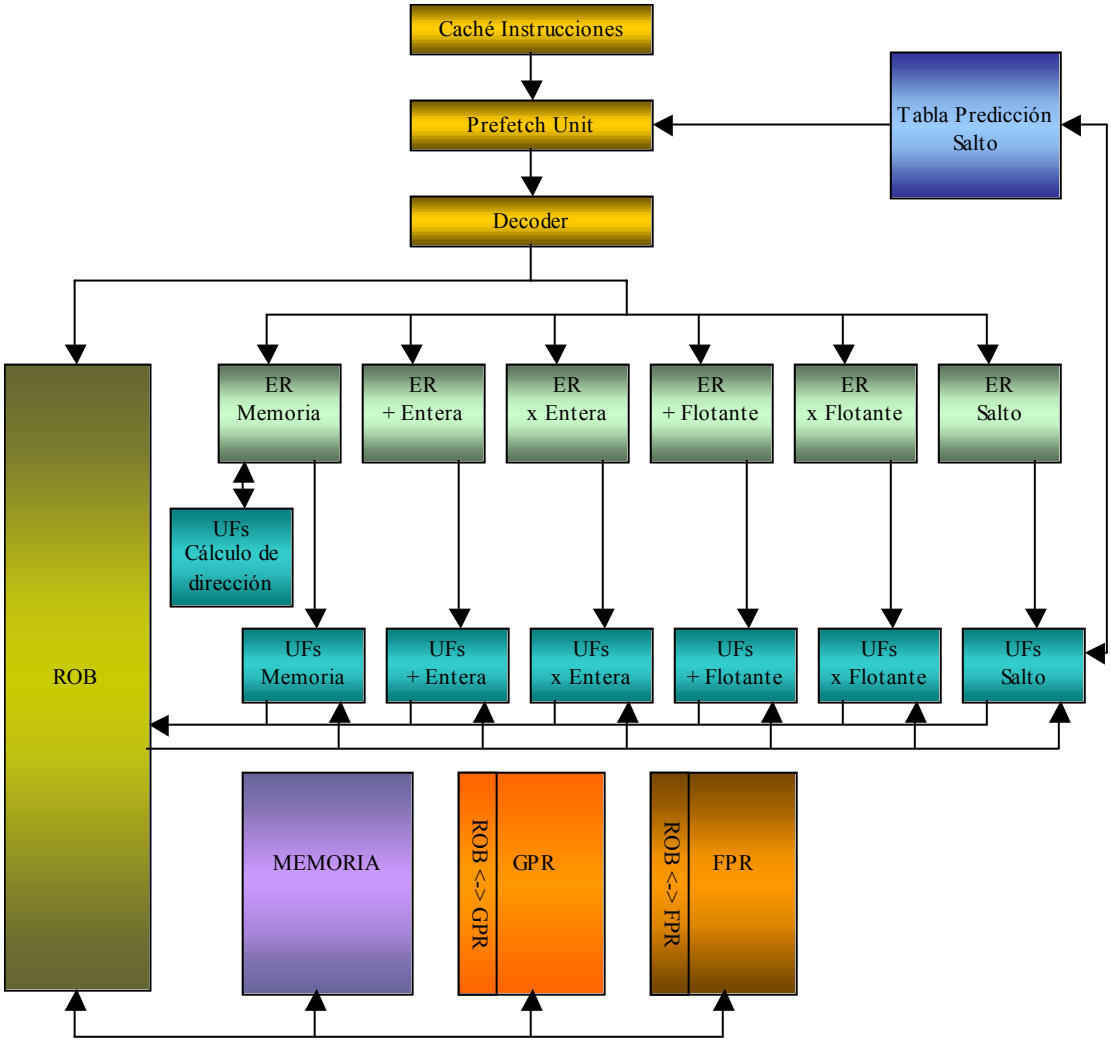


Ilustración 4-2. Esquema básico de la máquina Superescalar

4.6.3. Unidad de prebúsqueda

La unidad de prebúsqueda (*prefetch unit*) fue la respuesta a todos los problemas causados por los saltos si se quería realizar ejecución especulativa. Esta unidad vino a ocuparse de cargar las instrucciones teniendo en cuenta las predicciones de la tabla de saltos. En definitiva, se requería una unidad que, de forma completamente transparente al resto del esquema de la máquina superescalar, fuese capaz de surtir de un flujo continuo de instrucciones al resto de unidades (en concreto al decodificador), de tal manera que pudiese llevarse a cabo ejecución especulativa.

Su diseño en detalle habría sido realmente complejo, así que en este caso se optó por alejarse un poco del realismo para asegurarse su funcionalidad.

Lo que se dispuso que hiciera la unidad es ir cargando instrucción a instrucción hasta tantas como el **doble del *grado de emisión* por ciclo**. Para cada instrucción se realizaría una **predecodificación** para averiguar si se trataba de un salto. En caso de ser un salto, se consultaría la tabla de predicción de saltos para decidir si la siguiente instrucción que

debía cargarse sería la siguiente instrucción según el orden secuencial del código, o la instrucción destino del salto. En caso de tener que tomar el salto, se cargaría **de forma completamente transparente** la instrucción destino como siguiente instrucción del flujo de ejecución.

Esta implementación presupone dos simplificaciones bastante significativas:

- Cargar la instrucción destino de un salto no es muy costoso debido a que todas las direcciones de instrucciones son **absolutas** (esto ya se vio al hablar de la caché de instrucciones en la sección 4.4.1). Si no fuera así, habría que sumar la dirección del salto al PC, complicando considerablemente el esquema y obligando a esta unidad a ciertos retardos.
- La predecodificación se supone de costo en tiempo 0 o, como máximo, una fracción del ciclo de reloj igual al doble del grado de emisión.

Como puede comprobarse, esta unidad se convirtió en un híbrido entre una unidad de predecodificación y un *buffer* de instrucciones. Realmente, dadas las simplificaciones aplicadas, no era necesario cargar un número tan alto de instrucciones por ciclo, pero se decidió dejar este parámetro en previsión de futuras modificaciones.

4.6.4. Decodificador

El decodificador era una etapa indispensable desde el punto de vista funcional de la máquina. Es esta unidad la que decide a qué estación de reserva va cada instrucción y, en definitiva, se encarga de la emisión tal como se explica en el algoritmo de Tomasulo (sección 2.6.9). Por tanto, implementa la lógica de control que permite retener el flujo si el *Reorder buffer* o la estación de reserva a la que va la próxima instrucción están llenos

Desde el punto de vista de su implementación, esta unidad se comportaba simplemente como un *buffer* de paso para las instrucciones entre la unidad de prebúsqueda, y el ROB y las estaciones de reserva; y su funcionamiento interno era transparente al usuario.

4.6.5. Estaciones de Reserva (ER)

Para implementar las estaciones de reserva se optó por la solución más flexible, que era la de las **estaciones de reserva agrupadas**. En concreto se empleó una estación de reserva por cada tipo de unidad funcional definido.

Puesto que no había limitaciones reales de coste, el número de entradas de cada estación de reserva podía ser tan grande como se quisiera. Lo que sí existía era un **límite inferior** si se quería poder aprovechar totalmente el paralelismo en las unidades funcionales. Este valor se explica porque cualquier instrucción que se esté ejecutando en una unidad funcional debe estar contenida en la estación de reserva correspondiente durante todo el tiempo que dure su ejecución. Por tanto, si se llama *latUF* al número de etapas del *pipeline* de un tipo de unidad funcional, y *numUF* al número de unidades funcionales de ese tipo:

$$N^o \text{ de entradas } ER \geq latUF \times numUF$$

De hecho, dadas las características del diseño, podía suceder (en especial con unidades funcionales con pocas etapas) que este número de entradas fuese insuficiente. Para ello basta recordar que una estación de reserva debe mantener no sólo las instrucciones actualmente en ejecución, sino aquellas que están literalmente “en reserva” (a la espera de algún operando). Para reducir este problema se consideró poner un número ligeramente superior de entradas que quedó finalmente en:

$$N^{o}\ de\ entradas\ ER = (latUF + 1) \times numUF$$

Los campos que contenía cada entrada de la estación de reserva se tomaron directamente del esquema de [13]:

- **Instrucción**: Identificador de la instrucción.
- **Qj**: Disponibilidad del primer operando. -1 indica que el valor se encuentra en el campo Vj de la estación de reserva; otro valor indica la posición del ROB donde se está procesando el operando.
- **Vj**: Valor del primer operando si (Qj = -1).
- **Qk**: Disponibilidad del segundo operando. -1 indica que el valor se encuentra en el campo Vk; otro valor indica la posición del ROB donde se está procesando el operando.
- **Vk**: Valor del segundo operando si (Qk = -1).
- **A**: Dirección de memoria
- **ROB**: Posición del ROB donde está contenida esta instrucción.

4.6.6. Reorder Buffer (ROB)

El ROB se diseñó como un *buffer* circular, tal como se vio en la teoría. El primer problema que surgió fue el de su tamaño. Como en el caso de las estaciones de reserva existía una limitación con respecto al mínimo número de entradas que debía tener. Este número se obtiene fácilmente sabiendo que cualquier instrucción que esté en una estación de reserva debe estar también en el ROB (se puede comprobar esto estudiando el algoritmo de Tomasulo con especulación del punto 2.6.9). Aunque se valoró la posibilidad de dejar la responsabilidad de fijar el tamaño a los usuarios del programa, se optó finalmente por **igualar el tamaño a la suma del número de entradas de todas las estaciones de reserva**.

Se quería que el ROB diseñado facilitase la ejecución especulativa y además que pudiese ser empleado como mecanismo para el renombrado de registros. Estos dos objetivos se consiguieron con el esquema básico propuesto para el algoritmo de Tomasulo con especulación. Sin embargo, otra serie de problemas fueron resueltos mediante el ROB, como los derivados de los accesos a memoria que se verán en el apartado 4.6.8.

Los campos que se emplearon en el ROB fueron:

- **Instrucción**: Instrucción que está contenida en esa posición del ROB.
- **Listo**: Indica si el resultado de la instrucción ya ha sido calculado.
- **Valor**: Contiene el resultado de la instrucción
- **Destino**: Índice del registro destino de la instrucción
- **Dirección**: Sirve para el control de las instrucciones de acceso a memoria (ver 4.6.8)
- **Etapa**: Indica la etapa actual de procesamiento de la instrucción entre ISSUE, EXECUTE, WRITE RESULT y COMMIT.

4.6.7. Adaptación de los registros

Como se comentó en el punto anterior, el ROB se empleó como mecanismo de renombrado de registros. Esto hacía necesario un campo asociado a cada registro de la máquina (tanto de propósito general como de punto flotante) que indicara en qué posición del ROB se estaba resolviendo el resultado de ese registro. Añadir este campo presuponía modificar la estructura de los registros, así que se decidió, para no realizar más modificaciones en elementos comunes, añadir dos estructuras que funcionaran como tablas de *mapeo*:

- Una tabla de *mapeo* para los registros de propósito general: **ROB<->GPR**.
- Una tabla de *mapeo* para los registros de punto flotante: **ROB<->FPR**.

En estas estructuras, cualquier valor distinto de –1 indica una posición del ROB donde puede encontrarse (o se encontrará) el resultado de una operación que emplea ese registro como destino.

4.6.8. Acceso a memoria

El procesamiento de las instrucciones de *LOAD* y *STORE* en la máquina superescalar no era un tema demasiado sencillo.

El primer problema era la división en dos etapas de los accesos a memoria que se hace al ejecutar las instrucciones fuera de orden (ver 2.6.3). Esta división contemplaba una etapa con el cálculo de la dirección efectiva y una etapa con el acceso a memoria propiamente dicho. Con los elementos de que se disponía había dos posibles formas de implementar la división:

- Controlando mediante algún mecanismo el paso por las dos etapas en la unidad funcional de memoria.
- Dividiendo el trabajo entre dos unidades funcionales distintas.

Puesto que se quería alterar lo menos posible la estructura común de las máquinas, se optó por dividir el procesamiento entre dos UF: el cálculo de direcciones se haría en una UF específica de suma entera, y del resto del acceso a memoria se encargaría la UF de memoria.

El siguiente problema era la implementación de los *buffers* donde se almacenarían estas instrucciones. Lo que se hizo fue prescindir de estas estructuras e integrar su funcionalidad en otros elementos de los que ya se disponía: estaciones de reserva y ROB. Concretamente el campo *dirección* del ROB junto con el campo *A* de las estaciones de reserva asociadas a las unidades funcionales de memoria se adaptaron para sustituir la funcionalidad de estos *buffers*.

En la sección 2.6.3 se planteaba que era necesario mantener el orden relativo de cálculo de las direcciones efectivas de los *STORES* respecto al resto de instrucciones de acceso a memoria para evitar los riesgos de datos. La forma en que se implementó esta solución fue relajando el criterio del orden fijo en el cálculo de las direcciones, pero asegurando antes de ejecutar la lectura de memoria de un *LOAD* que no hubiese *STORES* anteriores cuya dirección no se hubiese calculado o coincidiese con la del *LOAD*. Para mantener el orden relativo de los *STORES* no era necesaria ninguna medida adicional, ya que la escritura a memoria se hacía durante la etapa de *commit*, y esta etapa se realiza en orden estricto de programa. Mediante el esquema de la Ilustración 4-3 se puede entender mejor esta idea. Este esquema está extraído del algoritmo de Tomasulo del siguiente punto.

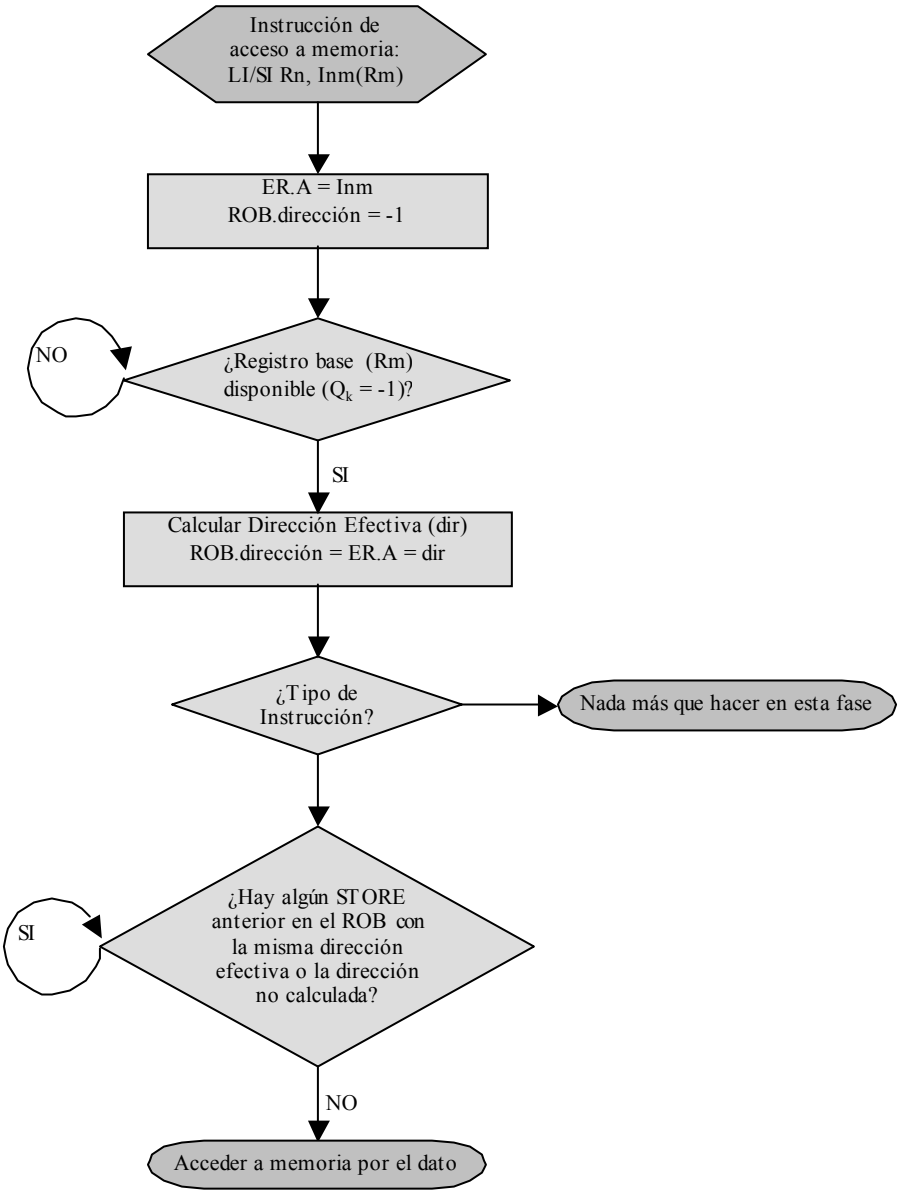


Ilustración 4-3. Esquema de la ejecución en dos fases de las instrucciones de acceso a memoria

4.6.9. Algoritmo de Tomasulo modificado

El algoritmo de Tomasulo visto en 2.6.9 necesitaba varias adaptaciones para funcionar con los elementos propuestos. En la Tabla 4-3 se puede ver el algoritmo tal como se usó[20]. La única excepción a los nombres de los elementos, añadida por simplicidad, es considerar la tabla de *mapeo* de los registros de propósito general como un campo de los propios registros (**`rob`**) en lugar de cómo una estructura aparte (ROB<->GPR).

Los elementos empleados siguen el mismo convenio de nombres que se empleó en la sección 2.6.9. En las instrucciones los registros origen se denominan **`ro1`** y **`ro2`**, el registro destino **`rd`** y los valores inmediatos **`inm`**.

[20] Por simplificar se ilustra sólo el algoritmo para las operaciones enteras. Las operaciones de punto flotante se tratan exactamente igual pero empleando los registros de punto flotante en lugar de los registros de propósito general.

Estado	Esperar hasta...	Acción
Emisión de todas las instrucciones	Estación de reserva (er) y ROB (b) estén libres	ROB[b].instrucción = instrucción; ROB[b].destino = -1; ROB[b].dirección = -1; ROB[b].listo = no; ROB[b].valor = -1; ER[er].ocupada = sí; ER[er].rob = b; ER[er].instrucción = instrucción; ER[er].A = -1;

DADDUI		si (!REG[ro1].ocupado) {
		ER[er].Vj = REG[ro1]; ER[er].Qj = -1;
		} else if (ROB[REG[ro1].rob].listo) {
		ER[er].Vj = ROB[REG[ro1].rob].Value; ER[er].Qj = -1;
		} else
		ER[er].Qj = REG[ro1].rob;
		ER[er].Qk = -1; ER[er].Vk = Inm;

ADDI, MULTI, LI		si (!REG[ro1].ocupado) {
		ER[er].Vj = REG[ro1]; ER[er].Qj = -1;
		} else if (ROB[REG[ro1].rob].listo) {
		ER[er].Vj = ROB[REG[ro1].rob].Value; ER[er].Qj = -1;
		} else
		ER[er].Qj = REG[ro1].rob;
		
		si (!REG[ro2].ocupado) {
		ER[er].Vk = REG[ro2]; ER[er].Qk = -1;
		} else if (ROB[REG[ro2].rob].listo) {
		ER[er].Vk = ROB[REG[ro2].rob].Value; ER[er].Qk = -1;
		} else
		ER[er].Qk = REG[ro1].rob;

DADDUI, ADDI, MULTI, LI		REG[rd].rob = rob; REG[rd].ocupado = sí; ROB[rob].destino = rd;
Loads		ER[er].A = inm; REG[ro2].rob = b; REG[ro2].ocupado = sí; ROB[rob].destino = ro2;
Stores		ER[er].A = inm;
Ejecutar	(ER[er].Qj = -1) y (ER[er].Qk = -1)	Operar resultados /* Operandos listos en Vj y Vk */
Load y Store paso 1	(ER[er].Qk = -1)	ER[er].A = ER[er].Vk + ER[er].A; ROB[ER[er].rob]] = ER[er].A;
Load paso 2	Paso 1 del load hecho y no hay ningún store anterior en el ROB con la misma dirección o la dirección no calculada	Leer desde MEM[ER[er].A];
Escribir resultado (todas menos store)	Ejecución hecha en er y CDB disponible	b = ER[er].rob; ER[er].ocupado = no; ∀x(si (ER[x].Qj = b) { ER[x].Vj = resul; ER[x].Qj = -1;}); ∀x(si (ER[x].Qk = b) { ER[x].Vk = resul; ER[x].Qk = -1;}); ROB[b].valor = resul; ROB[b].listo = sí;
Store	Ejecución hecha en er y (ER[er].Qj = 0)	ROB[h].valor = ER[er].Vj;

Graduación	Instrucción al frente del ROB (entrada h) y ROB[h].listo = sí;	d = ROB[h].destino;
		if (REG[d].rob == h)
		REG[d].ocupado = no;

Salto		if (salto mal predicho) {
		limpiar flujo completo de ejecución;
		buscar destino salto;
		}

Store		MEM[ROB[h].dirección] = ROB[h].valor;
Resto		REG[d] = ROB[h].valor;

Tabla 4-3. Algoritmo de Tomasulo aplicado en el Simulador

5. PROGRAMA: MANUAL DE USUARIO

En este capítulo se describirá el uso de la aplicación, presentando todas las opciones del simulador y algunos ejemplos. La aplicación puede conseguirse desde la página http://www.cyc.ull.es/simde/ o directamente vía ftp en el enlace ftp://ftp.etsii.ull.es/asignas/ARQUIT/SIMDE/SIMDE%20v1.0.zip.

5.1. Primer contacto

Para comenzar a usar el programa basta con hacer *doble clic* sobre el icono del ejecutable. Se mostrará la pantalla principal del programa, desde la que se tiene acceso a todas las opciones del simulador mediante el menú y las barras de herramientas.

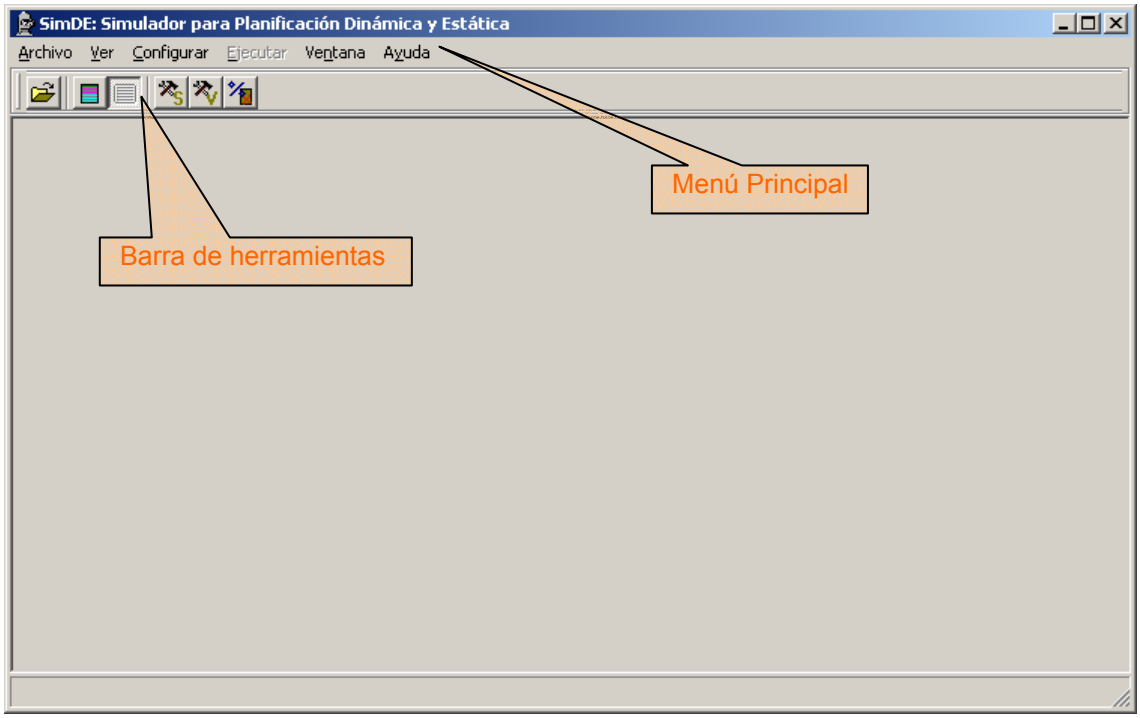


Ilustración 5-1. Pantalla principal del simulador

Un esquema del uso del programa puede verse en la Ilustración 5-2. En este esquema pueden verse a grandes rasgos las funcionalidades del programa.

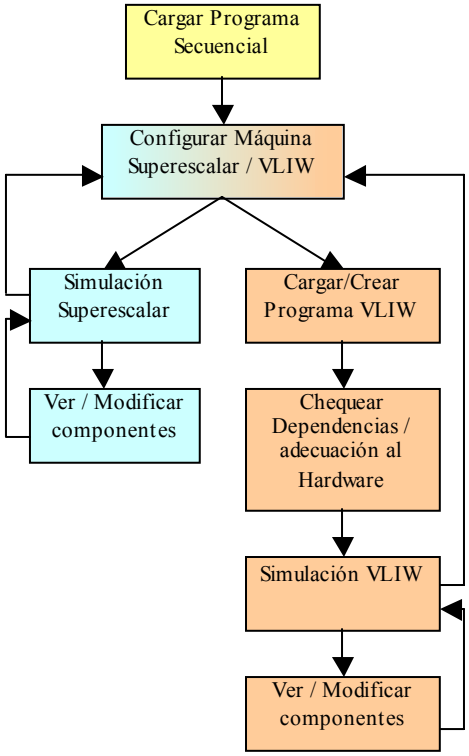


Ilustración 5-2. Esquema básico de uso del programa

5.1.1. Menú

Los menús del programa pueden resumirse de la siguiente manera:

- **Archivo**: Contiene las opciones para abrir un código secuencial que está en un fichero y para cerrar la aplicación.
- **Ver**: Permite activar o desactivar la visualización de las barras de herramientas, y también jugar con las opciones de visualización del código secuencial.
- **Configurar**: Contiene las opciones de configuración de los parámetros de las máquinas, así como el resto de parámetros más generales (como fallos de caché). Además contiene el acceso a las herramientas de construcción de código VLIW.
- **Ejecutar**: Permite escoger la máquina con la que realizar la simulación y controlar la simulación en sí, así como acceder a los componentes de las máquinas (memoria, registros…).
- **Ventana**: Son las opciones de visualización de las distintas ventanas de la aplicación
- **Ayuda**: Accede a la ayuda de la herramienta.

5.1.2. Acceso a la ayuda

En cualquier momento se tiene acceso a la ayuda usando la tecla **F1**.

Existe ayuda acerca del uso del simulador en general, detallando el uso de menús y barras de herramientas, así como las opciones disponibles desde cada ventana. Además hay una amplia sección con una descripción de las máquinas y todos sus elementos, y un apéndice con referencias a otros simuladores y documentos.

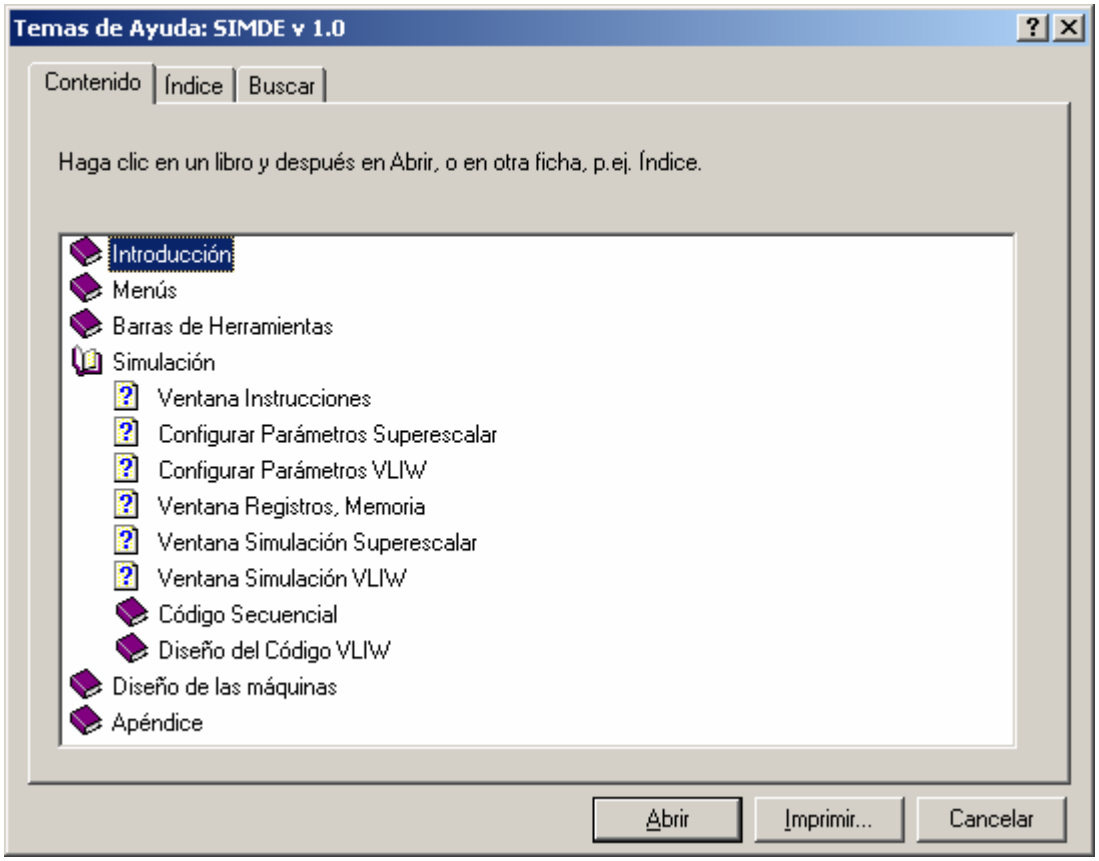


Ilustración 5-3. Ayuda de la aplicación

5.1.3. Barra de Herramientas Principal

La barra de herramientas principal tiene a mano algunas de las opciones más importantes del simulador:

Abrir: Permite abrir un fichero de código secuencial (.pla) para usarlo en las simulaciones.

Colorear bloques básicos: Asigna un color diferente a cada bloque básico del código para poder distinguirlos. Esta opción sirve de ayuda en la construcción de códigos de instrucciones largas.

Mostrar/Ocultar código secuencial: Si está seleccionada muestra el código secuencial. Al desmarcar esta opción se oculta el código (pero no se descarga, y puede continuarse con las simulaciones).

Configurar Superescalar: Abre la ventana de configuración de la máquina superescalar.

Configurar VLIW: Abre la ventana de configuración de la máquina VLIW.

Porcentaje de fallos de caché: Permite modificar el porcentaje de fallos de la caché de datos.

5.2. Crear un código secuencial

Los ficheros básicos para la aplicación son los ficheros de código secuencial. Para crear un código secuencial debe usarse un **editor de texto** (*notepad* o similar), respetando la siguiente estructura:

- La primera línea del fichero (que no sea un comentario) contiene el número de instrucciones del fichero.
- Cada instrucción debe ponerse en una nueva línea.
- Se permiten como separadores de operandos tabuladores o espacios.
- Las etiquetas se ponen al principio de la línea, permiten cualquier carácter alfanumérico y deben terminar con ":". Etiquetas válidas son: "LOOP:", "bucle1:", "2:"…
- Los comentarios se indican con "//" y abarcan hasta el final de la línea.

La sintaxis exacta de las instrucciones puede verse en la Tabla 5-1.

Tipo de instrucción	Sintaxis
Entera	DADDUI Rn Rm #i ADDI Rn Rm Rp MULTI Rn Rm Rp
Punto Flotante	ADDF Fn Fm Fp MULTF Fn Fm Fp
Memoria	LI Rn i(Rm) LF Fn i(Rm) SI Rn i(Rm) SF Fn i(Rm)
Salto	BNE Rn Rm LAB BEQ Rn Rm LAB

Tabla 5-1. Sintaxis de las instrucciones en los ficheros .pla

Un resumen de la nomenclatura empleada se detalla en la Tabla 5-2.

Símbolo	Explicación	Ejemplo
Rn	Registro de Propósito General n	R1, R0…
Fm	Registro de Punto Flotante m	F1, F0…
#i	Valor inmediato i	#12, #0…
i(Rm)	Dirección de memoria	(R1), 3(R4)…
LAB	Etiqueta destino de un salto	LOOP1, END…

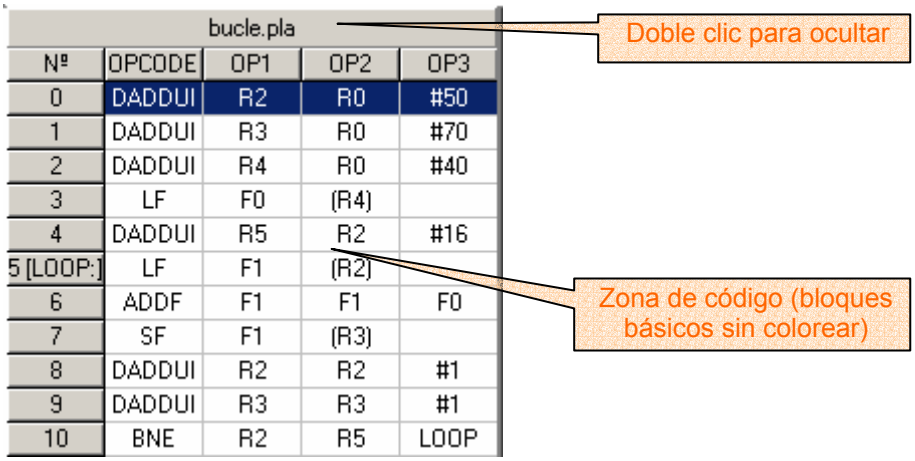
Tabla 5-2. Nomenclatura empleada en el código de los ficheros de entrada

El fichero creado debe guardarse con la extensión "**.pla**". Un primer fichero para realizar pruebas puede verse el Ejemplo 5-1.

5.3. Abrir un fichero de código secuencial

Ilustración 5-4. Abrir un fichero secuencial

Cargar el programa creado para usarlo en las simulaciones puede hacerse usando el menú **Archivo** y eligiendo la opción de **Abrir**, o presionando "**CTRL + A**". Se permitirá navegar por los directorios para buscar el fichero "**.pla**" creado.



Nº	OPCODE	OP1	OP2	OP3
0	DADDUI	R2	R0	#50
1	DADDUI	R3	R0	#70
2	DADDUI	R4	R0	#40
3	LF	F0	(R4)	
4	DADDUI	R5	R2	#16
5 [LOOP:]	LF	F1	(R2)	
6	ADDF	F1	F1	F0
7	SF	F1	(R3)	
8	DADDUI	R2	R2	#1
9	DADDUI	R3	R3	#1
10	BNE	R2	R5	LOOP

Ilustración 5-5. Código secuencial cargado

El fichero abierto se mostrará en la parte izquierda de la pantalla (ver Ilustración 5-5). El código se muestra en el orden secuencial original. En la primera columna aparece el número de orden de la instrucción, que servirá como **identificador**. Si la instrucción tenía una etiqueta asociada, se mostrará ésta entre corchetes ([etiqueta:]) justo después del identificador. La segunda columna es el código de la operación (*opcode*) y en las siguientes se muestran los operandos.

```
// SIMDE v1.0
// Autor: Iván Castilla Rodríguez
// Descripción: El programa presupone q en la posición 50
// (R2) de memoria tienes un vector de 16 elementos y quieres
// sumar a cada elemento una cantidad fija (en la posición de
// memoria 40). El resultado se coloca a partir de la
// posición 70 (R3) de memoria.
11
      DADDUI      R2 R0 #50
      DADDUI      R3 R0 #70
      DADDUI      R4 R0 #40
      LF          F0 (R4)
      DADDUI      R5 R2 #16
LOOP:
      LF          F1 (R2)
      ADDF        F1 F1 F0
      SF          F1 (R3)
      DADDUI      R2 R2 #1
      DADDUI      R3 R3 #1
      BNE         R2 R5 LOOP
// Fin del programa
```

Ejemplo 5-1. Fichero bucle.pla

Una vez abierto el fichero, pueden probarse algunas opciones. Para ocultar el código y dejar más espacio en la pantalla para seguir las simulaciones se puede acudir al menú **Ver** y seleccionar **Código Secuencial**. La misma opción está accesible desde la barra de herramientas () o haciendo *doble clic* sobre la parte superior del código, donde se

muestra el nombre del fichero cargado. Para volver a ver el código basta con seleccionar nuevamente esta opción. El código oculto no se descarga y puede continuarse con las simulaciones.

Otra de las opciones interesantes es el **coloreado de bloques básicos**. Esta opción puede activarse/desactivarse desde el menú **Ver**, desde la barra de herramientas () o pulsando "**CTRL + B**". Los bloques básicos son especialmente útiles para construir códigos VLIW.

Tras cargar un código secuencial, puede comenzarse inmediatamente con las simulaciones superescalares. Las simulaciones VLIW necesitan construir o cargar un código VLIW previamente.

5.4. Ejecución de una simulación

Una vez cargado el fichero secuencial se permite el acceso a las opciones de ejecución mediante la barra de herramientas de ejecución (ver Ilustración 5-6) o el menú **Ejecutar**.

Ilustración 5-6. Barra de herramientas Ejecución

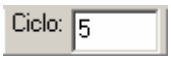
En ambos casos se puede acceder a las mismas opciones, que se detallan a continuación, incluyendo entre paréntesis la tecla de acceso rápido:

Iniciar (*F9*): Permite comenzar la ejecución continua de la simulación. Si se pulsa mientras está ejecutándose otra simulación, se puede escoger entre comenzar desde el principio la ejecución o continuar la ejecución actual. La ejecución continua sólo se detiene al llegar al final del programa o si encuentra un *breakpoint*.

Paso a Paso (*F8*): Permite avanzar ciclo a ciclo en la ejecución de una simulación.

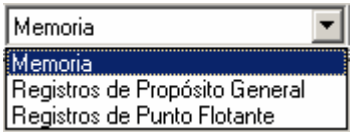
Parar (*F6*): Detiene una ejecución y pone el reloj a 0.

Pausa (*F7*): Permite pausar una ejecución iniciada en modo continuo. Una ejecución pausada puede proseguirse en modo continuo o en modo paso a paso.



Contador de ciclos (sólo en la barra de herramientas): Indica el ciclo de reloj actual en la simulación que se está ejecutando. Este valor es solamente informativo y no puede modificarse.

Velocidad de la simulación: Permite incrementar o decrementar la velocidad a la que se visiona la simulación continua. Se puede establecer cualquier valor desde 1 hasta 10 mediante los botones y  , donde 1 indica la velocidad más lenta de simulación y 10 es la velocidad más rápida.



Componentes: Mediante este cuadro combinado o el menú correspondiente pueden mostrarse los distintos componentes genéricos de las máquinas VLIW y Superescalar. Al seleccionar la memoria o cualquiera de los dos bancos de registros se mostrará una ventana de componentes como se verá en el punto 5.4.1. El componente que se muestra se corresponde con la máquina seleccionada para realizar la simulación.

Seleccionar Máquina (*F2* para la máquina Superescalar, *F3* para la VLIW): Con este cuadro combinado se escoge la máquina con la que se quiere llevar a cabo la simulación. La máquina VLIW sólo se permite si hay un código de instrucciones largas correctamente cargado o creado además del código secuencial.

Una ejecución, tanto continua como paso a paso, avanzará hasta llegar el final del programa. En ese momento aparecerá un mensaje informando de tal evento (ver Ilustración 5-7).

Ilustración 5-7. Fin de ejecución

Pueden realizarse tantas ejecuciones con una máquina como se quiera. Lo que hay que tener en cuenta es si se desea que con cada nueva ejecución la máquina se resetee o conserve los valores de la última ejecución. Esto se consigue en el menú **Configurar → Opciones**, y en el submenú se puede marcar o no la opción de **Resetear máquina al iniciar** ().

5.4.1. Ventanas de Componentes

Existen tres tipos de componentes genéricos: la memoria, los registros de propósito general (GPR) y los registros de punto flotante (FPR). Estos tres componentes aparecen en las dos máquinas, VLIW y Superescalar. Los componentes que se muestran en pantalla siempre se corresponden con la máquina actualmente seleccionada para hacer la simulación, es decir, no pueden verse simultáneamente la memoria de la máquina VLIW y la de la máquina Superescalar. En la primera columna se muestra el índice o la posición del elemento, mientras en la segunda columna puede verse (y **modificarse**) el valor correspondiente (por claridad, los números en punto flotante se presentan siempre redondeados a 3 decimales). Para este valor se permiten números enteros en el caso de los GPR y también flotantes en el caso de la memoria o FPR.

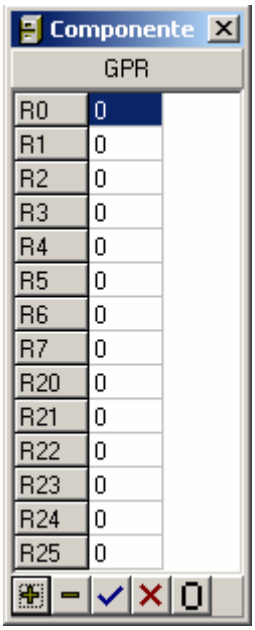


Ilustración 5-8. Ventana de componente genérico. En concreto, banco de GPR.

Tipo de número	Sintaxis permitida	Ejemplos
Entero	[+-]?[0-9]+	-98 1452 0
Flotante	[+-]?[0-9]*(“,”[0-9]+)?([eE][+-]?[0-9]+)?	-1 0,345 0,1E-4 3E10

Tabla 5-3. Sintaxis permitida de los valores de los componentes

Por defecto se muestran los 8 primeros elementos del componente, aunque se dispone de una serie de botones con los que manipular la presentación y el propio contenido de los mismos:

Añadir: Muestra un cuadro de diálogo (Ilustración 5-9) mediante el que puede seleccionarse un subconjunto de elementos para **mostrar** mediante una lista de números o intervalos separados por comas.

Quitar: Muestra un cuadro de diálogo (Ilustración 5-9) mediante el que puede seleccionarse un subconjunto de elementos para **ocultar** mediante una lista de números o intervalos separados por comas.

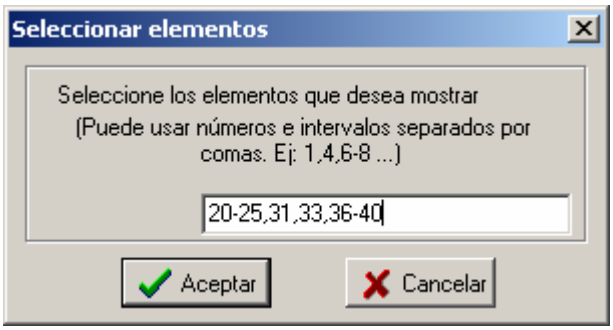


Ilustración 5-9. Cuadro de diálogo de selección de elementos. En el ejemplo se están seleccionando los elementos: 20, 21, 22, 23, 24, 25, 31, 33, 36, 37, 38, 39 y 40.

Guardar cambios: Guarda los valores modificados en el componente de la máquina.

Cancelar cambios: Carga de nuevo los valores que había almacenados en ese componente obviando los cambios realizados.

Resetear valores: Pone a 0 todos los componentes seleccionados de la máquina.

5.4.2. Porcentaje de fallos de caché de datos

Uno de los parámetros fundamentales durante la ejecución es el **porcentaje de los fallos de la caché de datos**. Este porcentaje establece la probabilidad de que un acceso de lectura a memoria no encuentre el dato en la caché y deba acudir a la memoria principal, con el consecuente incremento en el tiempo de acceso. Para modificar este valor puede irse al menú **Configurar ➔ Opciones** y escoger el botón correspondiente (). La misma opción está disponible desde la barra de herramientas principal. Cualquier valor entre 0 (sin fallos) y 100 (todos los accesos se hacen a memoria principal) está permitido.

5.5. Simulación Superescalar

5.5.1. Configuración de la máquina Superescalar

Antes de comenzar una simulación superescalar debe establecerse la configuración de la máquina sobre la que se harán las pruebas. Para ello se puede ir al menú **Configurar** y escoger la opción **Configurar Superescalar**, o presionar "**ALT + F2**". La ventana de configuración (Ilustración 5-10) permite modificar el número de unidades funcionales de cada tipo, la latencia o duración de la ejecución en cada unidad, y el grado de emisión de la máquina.

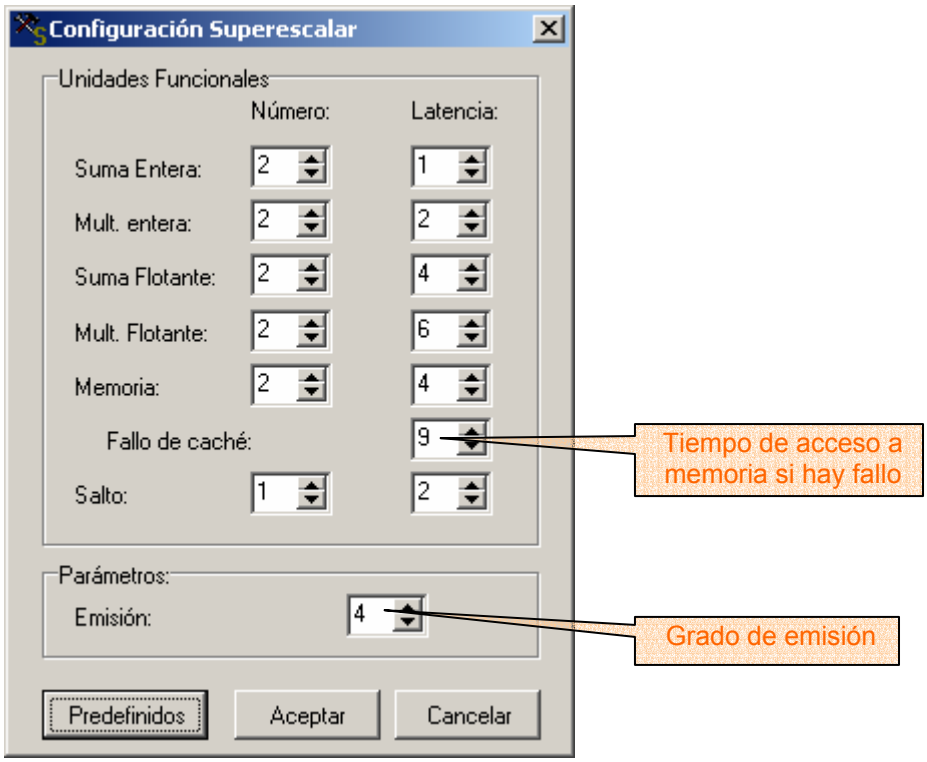


Ilustración 5-10. Pantalla de configuración de la máquina Superescalar

El grado de emisión se corresponde con el número de instrucciones que se emiten desde el decodificador al ROB y las estaciones de reserva por ciclo, pero también con el número de instrucciones graduadas (*commit*) por ciclo.

Entre las latencias puede indicarse también la duración del acceso a memoria si ocurre un fallo de caché. Este parámetro no se suma a la latencia de la unidad funcional de memoria, sino que es un tiempo absoluto. P. Ej. si la latencia de la UF de memoria es 4 y el tiempo del fallo de caché es 9, en caso de fallo se tardarán 5 ciclos adicionales en acceder a esa palabra de memoria.

La modificación de los valores se hace efectiva al hacer *clic* en el botón de **Aceptar**. Con **Cancelar** no se guardan los cambios y se puede volver a los valores por defecto con el botón de **Predefinidos**.

5.5.2. Ejecución de una simulación Superescalar

Para ejecutar una simulación superescalar basta con tener cargado un código secuencial y seleccionar la máquina Superescalar (desde la barra de herramientas de ejecución, desde el menú **Ejecutar ➔ Seleccionar Máquina**, o presionando **F2**). Al iniciar una ejecución continua o "paso a paso" se presentará en pantalla el esquema de la máquina Superescalar (Ilustración 5-11) con los siguientes componentes:

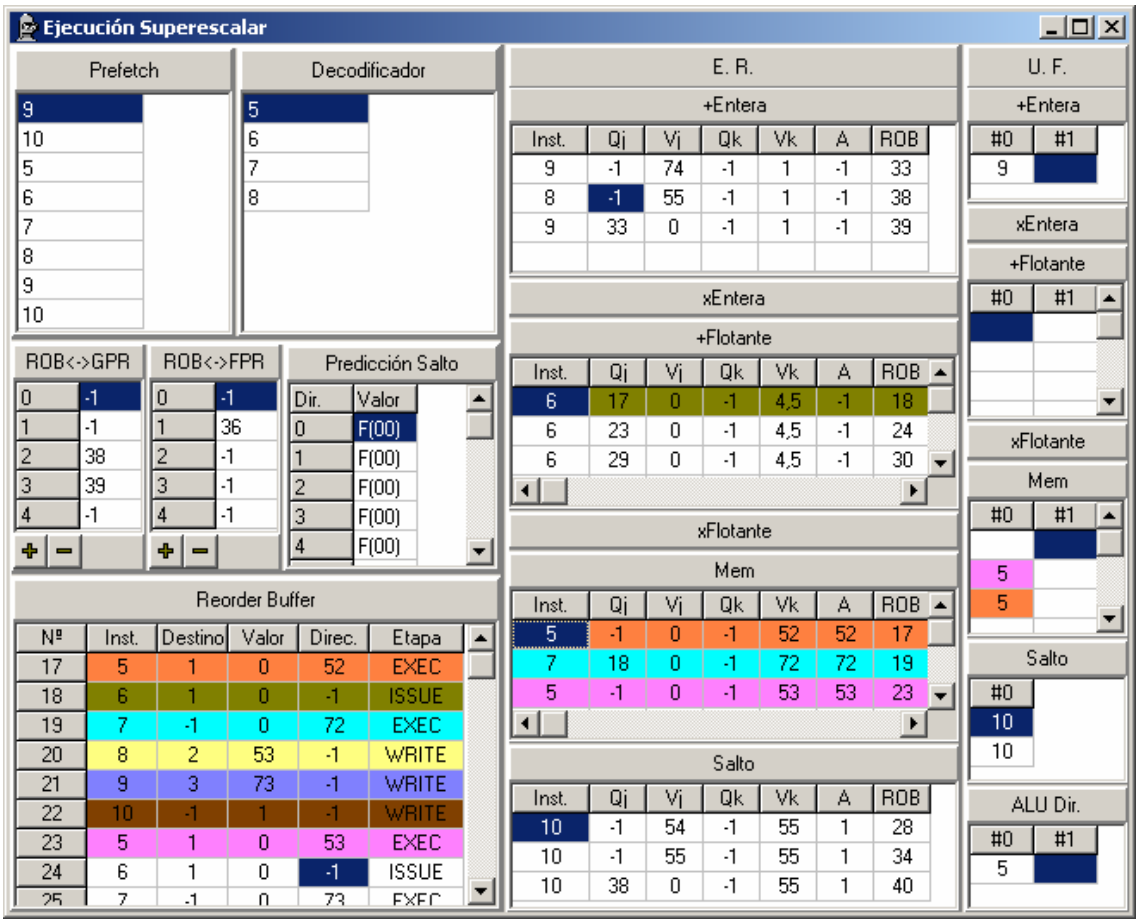


Ilustración 5-11. Ventana de ejecución Superescalar

- **Prefetch**: Unidad de prebúsqueda de instrucciones. Esta unidad busca las 2**emisión* próximas instrucciones del programa. Sin embargo no las busca en orden secuencial, sino que tiene en cuenta la tabla de predicción de salto para cargar aquellas instrucciones que se correspondan con la predicción.
- **Decodificador**: Decodificador de las instrucciones. A esta unidad pasan las *emisión* primeras instrucciones de la unidad de prebúsqueda. Después se encarga

de decodificarlas y de la emisión en sí, comprobando si hay espacio libre en el *reorder buffer* y en las estaciones de reserva correspondientes.

- **ROB<->GPR**: Posición del ROB donde se está procesando un registro de propósito general (-1 si el registro no está asociado a ninguna entrada del ROB). Se muestra sólo un subconjunto de los elementos. Se pueden añadir o quitar elementos mediante los botones  y  , indicándolos como una lista de números (o intervalos) separada por comas (Ej. 2,12-18,20).
- **ROB<->FPR**: Posición del ROB donde se está procesando un registro de punto flotante (-1 si el registro no está asociado a ninguna entrada del ROB). Se muestra sólo un subconjunto de los elementos. Se pueden añadir o quitar elementos mediante los botones  y  , indicándolos como una lista de números (o intervalos) separada por comas (Ej. 2,12-18,20).
- **Predicción Salto**: Tabla de predicción de salto de 16 entradas (4 bits). La predicción de un salto se coloca en la entrada cuyo índice se corresponda con los 4 últimos bits de la dirección en memoria de dicho salto. Si una entrada tiene el valor *F* es que el salto que coincide con esa dirección no se toma; si vale *V* el salto se toma. Además de esto se muestra el valor binario de la entrada de la tabla entre paréntesis.
- **Reorder Buffer (ROB)**: Buffer circular que contiene todas las instrucciones que han sido emitidas y aún no han finalizado completamente su ejecución (no se han graduado). Si está lleno obliga a detener la emisión de las instrucciones hasta que disponga de una posición libre. Muestra la siguiente información de cada instrucción:
  - **Identificador** de la instrucción.
  - Índice del registro **Destino** de la operación que realiza la instrucción (si lo tiene).
  - **Valor** resultado de la operación que ha realizado la instrucción (si lo tiene).
  - **Dirección** de memoria destino (*STORE*) u origen (*LOAD*) de la instrucción
  - **Etapa** en la que está en ese momento la instrucción. Su valor puede ser *ISSUE*, *EXECUTE*, *WRITERESULT* ó *COMMIT*.
- **E.R.**: Estaciones de Reserva. Son *buffers* que contienen a las instrucciones desde que son emitidas hasta que termina su ejecución (terminan de usar la unidad funcional correspondiente). La instrucción se mantiene mientras está a la espera de tener todos sus operandos disponibles, es decir, hasta que puede ser enviada (*dispatched*) a una unidad funcional para ser ejecutada. La instrucción se retira de la ER cuando ha finalizado su procesamiento en esa unidad funcional. Existen una ER por cada tipo de unidad funcional, estando etiquetadas en su parte superior indicando esta relación. Los campos que almacena la ER son:
  - **Identificador** de la instrucción.
  - **Q_j:** Disponibilidad del primer operando. -1 indica que el valor se encuentra en V_j; otro valor indica la posición del ROB donde se está procesando el operando.

- **V_j:** Valor del primer operando si Q_j vale –1.
- **Q_k:** Disponibilidad del segundo operando. -1 indica que el valor se encuentra en V_k; otro valor indica la posición del ROB donde se está procesando el operando.
- **V_k:** Valor del segundo operando si Q_k vale –1.
- **A:** Almacenamiento temporal para calcular la dirección efectiva de una instrucción de memoria. Inicialmente contiene el *offset*.
- **ROB:** Posición del ROB donde se encuentra también la instrucción, y donde se almacenará, si corresponde, el resultado de la misma.

➢ **U.F.**: Unidades Funcionales. Son las unidades básicas de ejecución de la máquina. En la parte superior aparece el tipo de UF de que se trata entre suma entera (+Entera), multiplicación entera (xEntera), suma flotante (+Flotante), multiplicación flotante (xFlotante), memoria (Mem) y salto. Por cada tipo de UF hay una tabla con tantas columnas como UF de ese tipo estén declaradas. Las filas representan el *pipeline* de la UF.

5.5.3. Seguimiento de instrucciones

Debido a la ingente cantidad de información que se muestra en la ejecución superescalar resulta complicado el seguimiento de una instrucción en concreto. Para facilitar esta tarea se permite colorear cada instrucción del ROB con un color elegido por el usuario. Haciendo *doble clic* sobre una entrada del ROB que contenga una instrucción se presentará una ventana con la que escoger el color. La instrucción se coloreará en el ROB y también en la estación de reserva y/o unidad funcional que corresponda.

Para quitarle el color a la instrucción basta con volver a hacer *doble clic* con el ratón en la entrada correspondiente del ROB.

Ilustración 5-12. Cuadro de diálogo de selección de color de la instrucción

5.5.4. Breakpoints

Los *breakpoints* sirven para detener una ejecución continua en un punto del código determinado. En el caso de la máquina Superescalar, el *breakpoint* se indica haciendo *doble clic* sobre la instrucción en la ventana que muestra el código secuencial cargado. El identificador de la instrucción queda marcado en rojo hasta que se vuelva a hacer *doble clic* en la instrucción.

La ejecución se detendrá cuando se intente cargar la instrucción marcada en la unidad de prebúsqueda (*prefetch*), y podrá reiniciarse en modo continuo ( ) o paso a paso ( ).

5.5.5. Redimensionamiento de los componentes

Otra de las opciones más interesantes de la ventana de ejecución superescalar es la posibilidad de cambiar el tamaño de casi cualquier componente para ajustar mejor la presentación a las necesidades del usuario. El **alto y ancho de la mayoría de los componentes puede modificarse** poniendo el ratón sobre alguno de sus bordes y arrastrando manteniendo presionado el botón izquierdo del ratón.

Tanto las estaciones de reserva como las unidades funcionales pueden **ocultarse** haciendo *doble clic* sobre el nombre del componente correspondiente. Para volver a mostrarlo basta con volver a hacer *doble clic*.

5.6. Código de instrucciones largas

Para poder realizar simulaciones con la máquina VLIW debe disponerse de un código de instrucciones largas, construido a partir de algún código secuencial. El código VLIW depende también de la configuración actual de la máquina (ver sección 5.7.1), ya que existe una correspondencia directa entre el número y tipo de unidades funcionales y el formato de la instrucción larga.

Estos códigos pueden crearse directamente a partir de un código secuencial o cargarlos desde un fichero. Para cargar un programa VLIW desde un fichero hay que ir al menú **Configurar ➔ Instrucciones VLIW** y escoger la opción de "**Cargar Programa VLIW…"** ( ). Se presentará un cuadro de diálogo para escoger el fichero "**.vliw**" deseado. Tras escoger un fichero se mostrará la ventana de edición de código VLIW (Ilustración 5-13).

5.6.1. Diseño del código VLIW

Tanto si se quiere editar un código VLIW ya creado cargado desde un fichero, como si lo que se quiere es construir un nuevo código partiendo de cero, hay que trabajar con la ventana de diseño de Código VLIW. Esta ventana se compone de una **Rejilla de Diseño** en la que cada fila es una instrucción y cada columna, una UF de la máquina. La primera columna es el identificador de la instrucción larga.

Para crear un código partiendo de cero hay que ir al menú **Configurar ➔ Instrucciones VLIW** y escoger la opción de **"Crear Nuevo Programa VLIW"** ( ).

Se dispone de una serie de botones para facilitar la creación del código:

Añadir: Permite añadir más instrucciones (filas) a la Rejilla de Diseño. Al pulsarlo se solicitará el número de instrucciones que se quiere añadir.

Eliminar: Permite eliminar la instrucción seleccionada de la Rejilla de Diseño. Al eliminar una instrucción todas las instrucciones posteriores decrementan su identificador.

Limpiar: Deja la ventana de diseño en blanco, borrando todo el código construido hasta el momento.

Aceptar: Guarda los cambios realizados en el código. Al pulsar **Aceptar** se realizan una serie de comprobaciones sobre el código si está marcada la opción **"Chequear código VLIW"** en el menú **Configurar ➔ Opciones**. Los mensajes de error se devuelven de la forma: "*Error en la operación # de la instrucción larga #*", con lo que resulta sencillo localizar el fallo.

Cancelar: Cancela los cambios realizados desde la última vez que se usó el botón de **Aceptar**.

Guardar: Guarda el código en un fichero. Se presenta un cuadro de diálogo que permite escoger el nombre del fichero "**.vliw**" donde guardar el código creado. Este código puede cargarse posteriormente con la opción "**Cargar Programa VLIW…**" ( ) del menú **Configurar ➔ Instrucciones VLIW**.

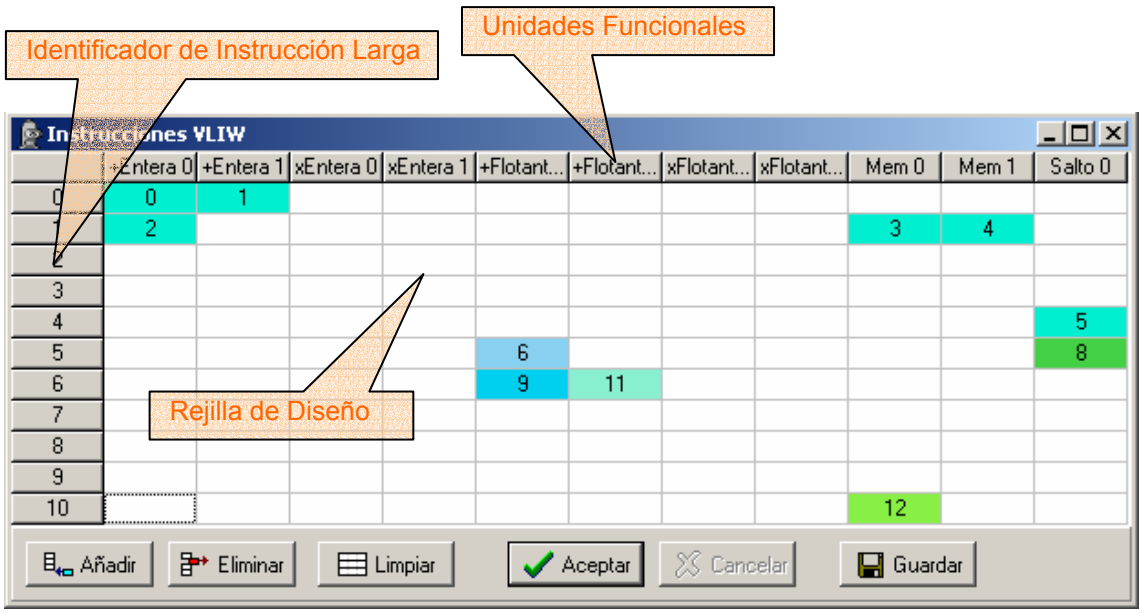


Ilustración 5-13. Pantalla de construcción de código VLIW

Añadiendo-Eliminando operaciones

La creación del código VLIW es muy sencilla. Basta con arrastrar las operaciones desde la ventana de código secuencial hasta la rejilla de diseño. El programa sólo permite arrastrar las operaciones a una UF donde puedan ser ejecutadas. Una vez copiada una operación, puede eliminarse seleccionándola y pulsando la tecla **SUPR**.

Zona de influencia

Durante la creación del código puede observarse la zona de influencia de una operación haciendo **CTRL** + ***clic*** con el botón izquierdo del ratón sobre la misma. De esta forma se colorean las instrucciones largas durante las cuales no pueden planificarse operaciones que tengan operandos dependientes de la operación marcada o, en el caso

de los saltos, las instrucciones en las que pueden usarse los registros de predicado asociados a ese salto.

Operaciones de salto

Las operaciones de salto deben ser configuradas en el momento de arrastrarlas. Para ello se mostrará un cuadro de diálogo (ver Ilustración 5-14) en el que debe indicarse:

- **Instrucción destino**: La instrucción larga destino del salto.
- **Registro Predicado Verdadero**: El Registro de Predicado que cambiará su valor a VERDADERO si el salto es tomado.
- **Registro Predicado Falso**: El Registro de Predicado que cambiará su valor a VERDADERO si el salto NO es tomado.
- **Registro Predicado**: Registro de Predicado asociado a esta operación.

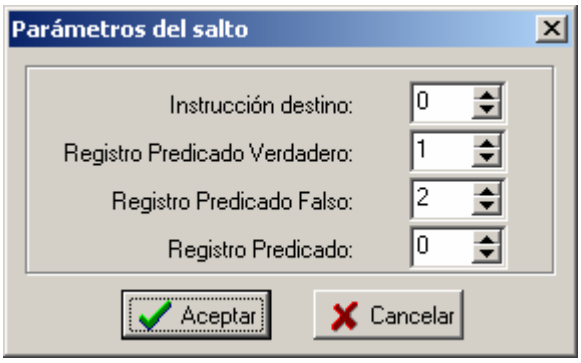


Ilustración 5-14. Cuadro de diálogo de parámetros del salto VLIW

Parámetros de las operaciones

Los parámetros de una operación pueden ser modificados en cualquier momento haciendo *doble clic* sobre la operación en concreto. Si se trata de una operación de salto aparecerán los parámetros que se nombraron antes; en otro caso simplemente se solicita el Registro de Predicado asociado a la operación.

5.6.2. Consejos en la creación del código

La creación del código VLIW a partir del código secuencial es una tarea compleja, más aun si es el propio usuario el que tiene que hacer el trabajo del compilador. Se incluye en esta sección una serie de consejos y recomendaciones que pueden ayudar a obtener un código que funcione lo más rápidamente posible.

Consejos Generales

- A partir de un código secuencial sólo se puede jugar con la colocación de las operaciones en el código VLIW. Las optimizaciones del código (desenrollado de bucles, software-pipelining…) se realizan sobre el código secuencial, creando un nuevo fichero de entrada.
- El hardware de la máquina VLIW no realiza ninguna optimización sobre el código que se le da, sino que todas las optimizaciones las realiza el compilador-usuario. Para obtener tiempos de ejecución similares a la máquina Superescalar debe tenerse especial cuidado en la colocación de las operaciones en las instrucciones largas y la mayor optimización posible del código secuencial.

Consejos de creación de código

- La forma más sencilla de construir el código es comenzar a arrastrar las operaciones en orden desde el código secuencial a la primera UF libre del tipo correspondiente que se encuentre.
- Es importante fijarse en las dependencias RAW antes de arrastrar una operación. Colorear la zona de influencia de una operación (haciendo doble clic) puede ayudar a saber dónde colocar la siguiente instrucción.
- Los bucles son una parte fundamental del código. Es importante tener bien presente que una operación de salto cuyo destino sea una operación secuencial se convierte en un salto a una instrucción larga al pasarlo al código VLIW.
- El destino más obvio de un salto es la primera instrucción que contenga una operación del bloque básico destino del salto, pero no siempre es la mejor opción.
- Debe recordarse que las instrucciones largas se ejecutan completamente, no la mitad o sólo algunas de las operaciones. Hay que tener esto en cuenta a la hora de poner en una misma instrucción operaciones que son destino de un salto con otras que no lo son.
- Es interesante aprovechar la posibilidad de repetir operaciones en el código. Tal vez se quiera que la primera iteración de un bucle se ejecute de una forma para después aplicar predicación el resto de iteraciones.
- El uso de registros de predicado es una herramienta bastante potente que permite ejecutar en paralelo operaciones de las dos ramas de un salto.
- Debe recordarse que se puede predicar cualquier operación, incluso otra operación de salto.
- La opción de Chequear Código VLIW no asegura que el código obtenido sea 100 % fiable, pero sí ayudará a detectar algunos errores.
- El aprovechamiento de la configuración de los bancos de registros, que leen en la primera mitad del ciclo y escriben en la segunda mitad puede ahorrar un ciclo en muchas ocasiones.

Posibles errores

- Hay que tener cuidado al asignar un registro de predicado a una operación. Debe asegurarse que esté en la zona de influencia del salto que utiliza esos registros de predicado.
- Para poder predicar una operación no debe terminar su ejecución antes de evaluar el salto, porque en ese caso no podrá cancelarse a tiempo si resulta pertenecer a la rama no tomada del salto.
- El algoritmo empleado para chequear el código creado es básicamente heurístico. En ocasiones detectará dependencias RAW que realmente no lo son, o no detectará otras que sí lo son. Su uso es válido como una referencia pero la comprobación final debe realizarla el usuario.

5.7. Simulación VLIW

5.7.1. Configuración de la máquina VLIW

Al igual que con la máquina Superescalar, antes de comenzar una simulación VLIW debe establecerse la configuración de la máquina sobre la que se harán las pruebas. Para ello se puede ir al menú **Configurar** y escoger la opción **Configurar VLIW**, o usar la combinación de teclas "**ALT + F3**". La ventana de configuración (Ilustración 5-15) permite modificar el número de unidades funcionales de cada tipo excepto de la de salto, y la latencia o duración de la ejecución en cada unidad. El número de unidades funcionales de cada tipo establece también el formato de la instrucción larga; es por eso que el número de unidades funcionales de salto no puede modificarse, ya que está limitado a una operación de salto por instrucción larga.

El parámetro de los *Fallos de caché* se interpreta igual que en la máquina superescalar.

La modificación de los valores se hace efectiva al hacer *clic* en el botón de **Aceptar**. Con **Cancelar** no se guardan los cambios y se puede volver a los valores por defecto con el botón de **Predefinidos**.

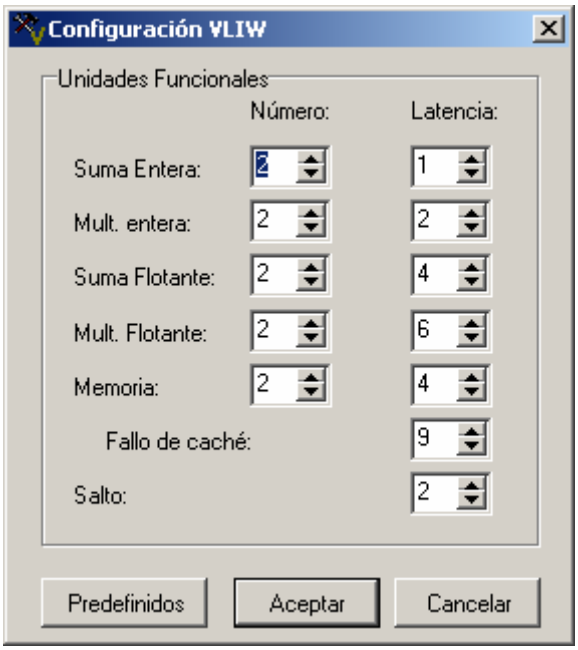


Ilustración 5-15. Pantalla de configuración de la máquina VLIW

5.7.2. Ejecución de una simulación VLIW

Para ejecutar una simulación VLIW se debe tener cargado un código secuencial, pero también debe haberse creado un código de instrucciones largas como se vio en la sección 5.6. Tras seleccionar la máquina VLIW (desde la barra de herramientas de ejecución, desde el menú **Ejecutar ➔ Seleccionar Máquina**, o presionando **F3**) puede iniciarse una ejecución continua o "paso a paso" que presentará en pantalla el esquema de la máquina VLIW (Ilustración 5-16) con los siguientes componentes:

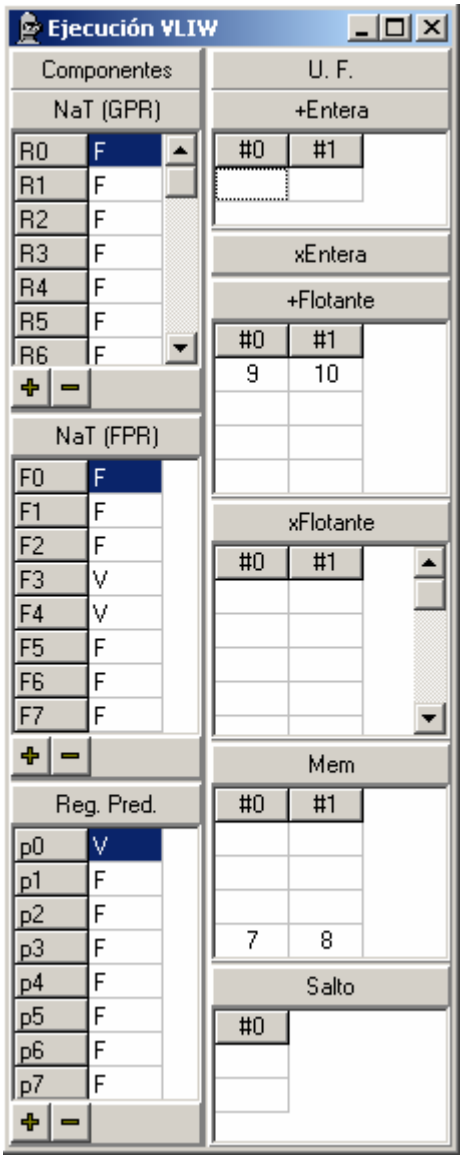


Ilustración 5-16. Ventana de ejecución VLIW

- **NaT (GPR)**: Bits de NaT (*Not a Thing*) asociados a los registros de propósito general. Si muestran una *V* es que el registro está siendo usado como destino de un *LOAD* y por tanto su valor no está disponible; una *F* indica que el valor del registro es válido. Se pueden añadir o quitar elementos mediante los botones + y −, indicándolos como una lista de números (o intervalos) separada por comas (Ej. 2,12-18,20).
- **NaT (FPR)**: Bits de NaT (*Not a Thing*) asociados a los registros de punto flotante. Si muestran una *V* es que el registro está siendo usado como destino de un *LOAD* y por tanto su valor no está disponible; una *F* indica que el valor del registro es válido. Se pueden añadir o quitar elementos mediante los botones + y −, indicándolos como una lista de números (o intervalos) separada por comas (Ej. 2,12-18,20).
- **Reg. Pred.**: Registros de Predicado. Una *V* indica que las operaciones asociadas a este registro se ejecutarán. Con *F* no se ejecutarán. Se pueden añadir o quitar elementos mediante los botones + y −, indicándolos como una lista de números (o intervalos) separada por comas (Ej. 2,12-18,20).

- **U.F.**: Unidades Funcionales. Son las unidades básicas de ejecución de la máquina. En la parte superior aparece el tipo de UF de que se trata entre suma entera (+Entera), multiplicación entera (xEntera), suma flotante (+Flotante), multiplicación flotante (xFlotante), memoria (Mem) y salto. Por cada tipo de UF hay una tabla con tantas columnas como UF de ese tipo estén declaradas. Las filas representan el *pipeline* de la UF.

5.7.3. Breakpoints

Los *breakpoints* sirven para detener una ejecución continua en un punto del código determinado. En el caso de la máquina VLIW, el *breakpoint* se indica haciendo *doble clic* sobre el identificador de la instrucción larga en la ventana del código VLIW. El identificador de la instrucción queda marcado en rojo hasta que se vuelva a hacer *doble clic* en el mismo.

La ejecución se detendrá cuando se intente ejecutar la instrucción marcada, y podrá reiniciarse en modo continuo () o paso a paso ().

5.7.4. Redimensionamiento de los componentes

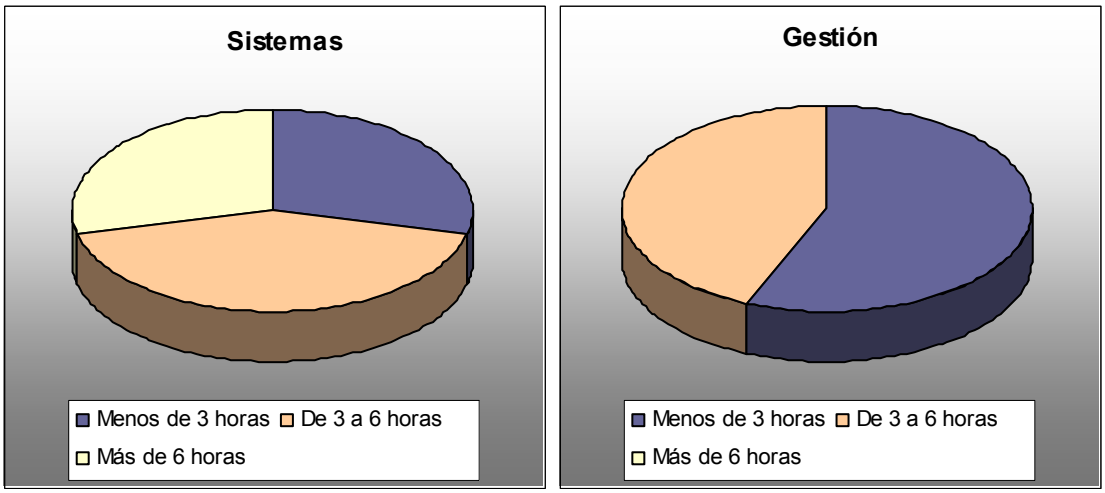
El **alto y ancho de la mayoría de los componentes puede modificarse** poniendo el ratón sobre alguno de sus bordes y arrastrando manteniendo presionado el botón izquierdo del ratón. Las unidades funcionales pueden **ocultarse** haciendo *doble clic* sobre su nombre. Para volver a mostrarla hay que hacer *doble clic* otra vez.

6. VALIDACIÓN DE LA HERRAMIENTA

El resultado de este proyecto fue un programa de simulación que debía servir como herramienta de apoyo a la docencia. La única forma de valorar fundamentadamente la herramienta de simulación era mediante su validación; y las personas más adecuadas para realizar esta validación eran los usuarios finales de esta aplicación: los alumnos de Arquitectura e Ingeniería de Computadores de quinto curso de la Ingeniería Informática. Para ello se realizó una presentación de la herramienta en el aula, y se les facilitó el simulador para que experimentaran con él y lo evaluaran mediante una encuesta (ANEXO B).

6.1. Muestra

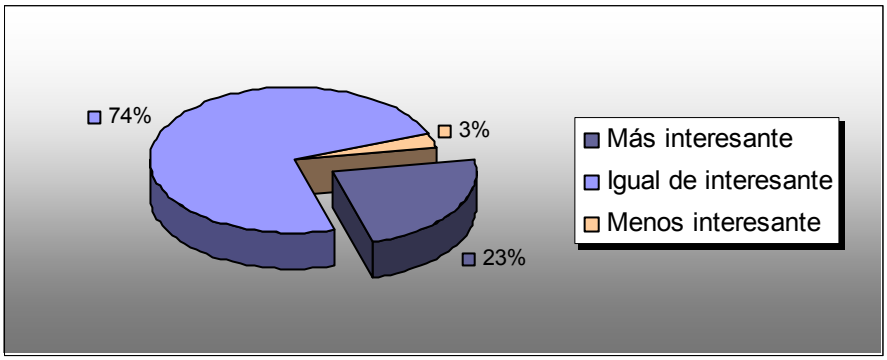
Los resultados se obtuvieron sobre 30 alumnos de la asignatura de Arquitectura e Ingeniería de Computadores de quinto curso de la Ingeniería Informática en el curso académico 2003/2004. La gran mayoría de los encuestados cursaban por primera vez la asignatura (sólo 4 alumnos eran repetidores).



Gráfica 6-1. Comparativa Sistemas-Gestión entre el tiempo dedicado por los alumnos para la validación de la herramienta

Los resultados obtenidos ofrecían algunas diferencias significativas entre los alumnos provenientes de la Ingeniería Técnica en Informática de Sistemas (14 alumnos) y los provenientes de la Ingeniería Técnica en Informática de Gestión (16 alumnos), por lo que ciertos resultados se presentan comparados entre ambos. Esta diferencia de resultados puede tener su motivación en la diferente formación de los alumnos: los alumnos provenientes de Sistemas tienen una asignatura obligatoria de *Introducción a la Arquitectura de Computadores* (plan 96), donde ya se introducen temas como la segmentación o las jerarquías de memoria; también realizan un acercamiento más profundo a los temas hardware con asignaturas como *Estructura y Tecnología de Computadores IV* (plan 96), que estudia las estructuras internas de microprocesadores de la familia 80x86 de Intel y las compara con otros microprocesadores[21]. Pese a que la formación en el segundo ciclo tiende a homogeneizar los conocimientos de los alumnos, es indudable que el propio interés en este tipo de materias viene bastante influenciado por la formación elegida por los propios alumnos.

Los alumnos emplearon una media de 4 horas probando la aplicación, aunque fueron los alumnos provenientes de Sistemas los que dedicaron en media más horas (ver Gráfica 6-1). En la Gráfica 6-2 se muestra el interés que despertaba esta parte entre los alumnos con respecto al resto de temas de la asignatura. Aunque la mayoría de los alumnos no destacaban el ILP como tema frente a otros contenidos de la asignatura, sí se observa en esta gráfica como casi 1 de cada 4 destacaban la importancia de este tipo de arquitecturas.



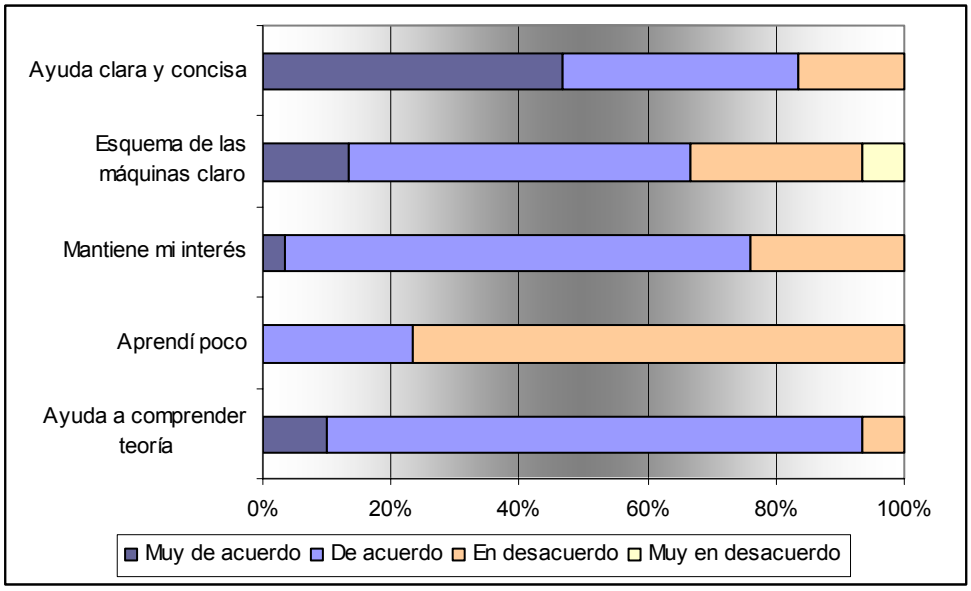
Gráfica 6-2. Interés del ILP para los alumnos con respecto al resto de temas de la asignatura

6.2. Esquema de la encuesta

La encuesta se dividió en tres secciones principales:

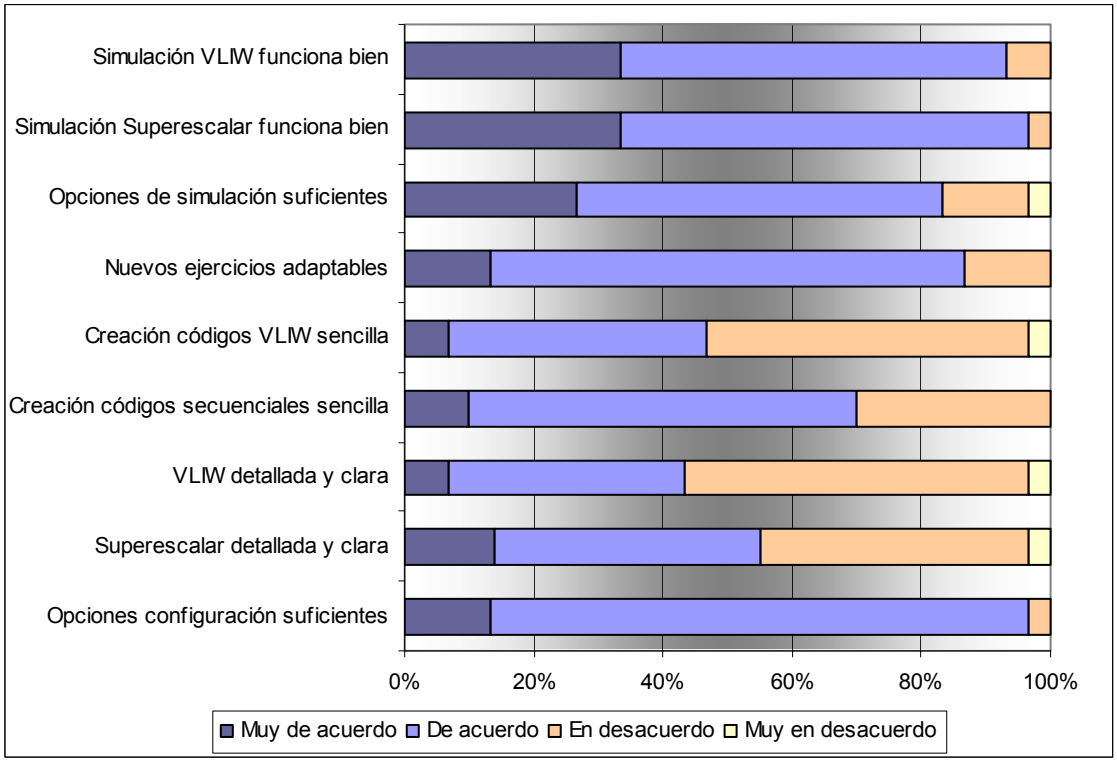
- La primera sección intentaba obtener información acerca de la adecuación de los contenidos teóricos del simulador y su aspecto educativo (Gráfica 6-3). Se pretendía destacar si los elementos del simulador favorecían el aprendizaje del alumno.

[21] Más información sobre el plan de estudios de la Ingeniería Informática en la Universidad de La Laguna puede obtenerse accediendo a la web de la Escuela Técnica Superior de Ingeniería Informática, en el enlace a los planes de estudio: http://www.etsii.ull.es/asignas.php



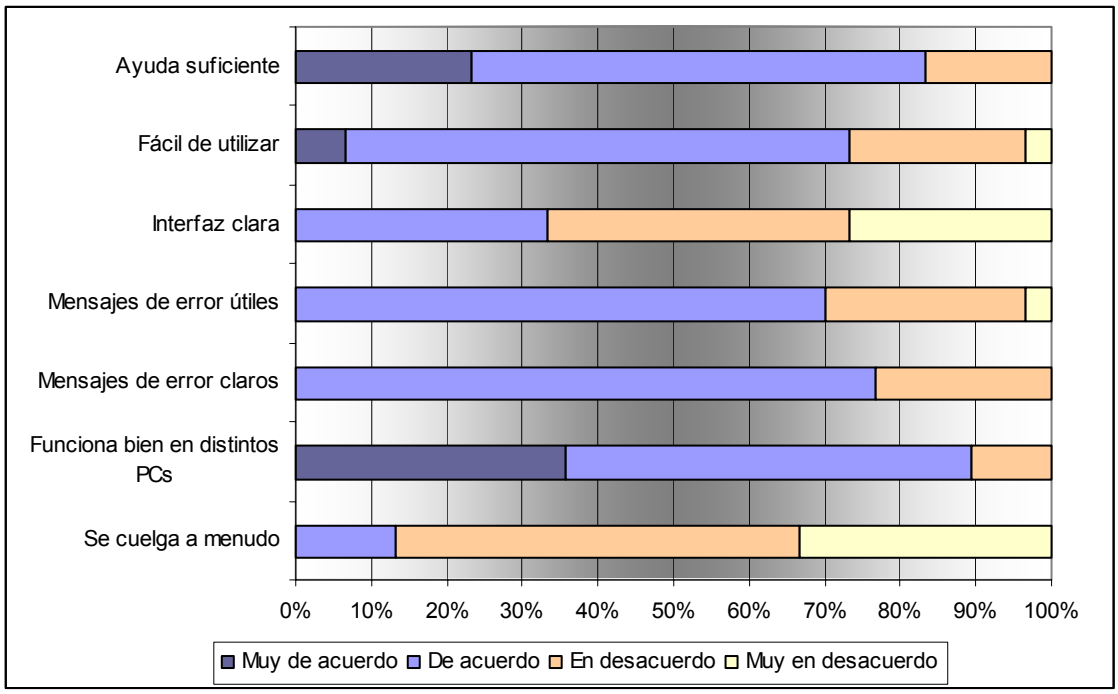
Gráfica 6-3. Valoración del aspecto educativo del simulador

- Después había una serie de preguntas sobre las funcionalidades del simulador (Gráfica 6-4). En este grupo de cuestiones se trataba de descubrir si las opciones que presentaba el simulador eran suficientes para el alumno y estaban presentadas de forma correcta.



Gráfica 6-4. Valoración de las funcionalidades del simulador

- Por último se presentaban unas cuestiones que pretendían comprobar el correcto funcionamiento de los aspectos técnicos del programa diseñado (Gráfica 6-5).



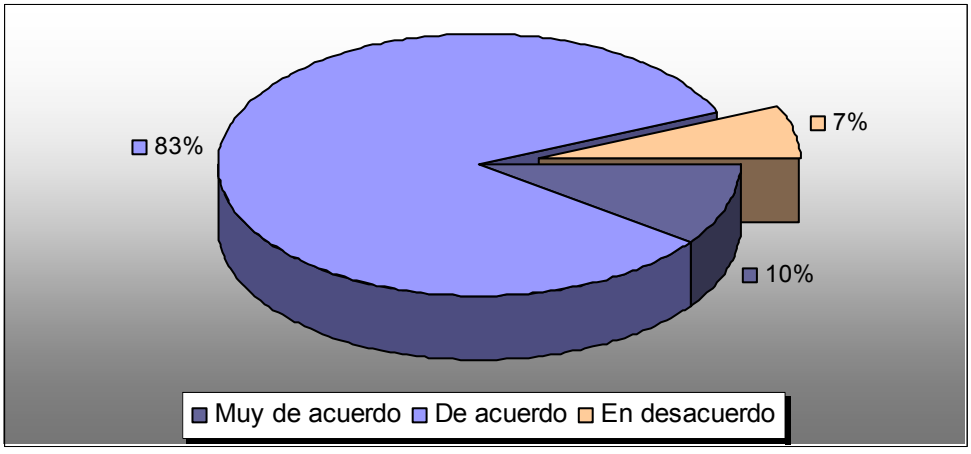
Gráfica 6-5. Valoración de los aspectos técnicos del simulador

Después de estas cuestiones se incluían una serie de preguntas en las que los alumnos podían aportar ideas y sugerencias, o criticar abiertamente las carencias del simulador.

6.3. Detalle de los resultados

Tomando tanto los comentarios individuales de los alumnos, como los resultados de las preguntas más concretas de las encuestas se obtuvieron una serie de conclusiones respecto a ciertos aspectos de la herramienta de simulación. Los alumnos, lejos de una visión negativa, aportaron multitud de críticas constructivas basadas en su experiencia con el programa y se mostraron bastante satisfechos con esta primera versión de la herramienta.

6.3.1. Aspecto docente



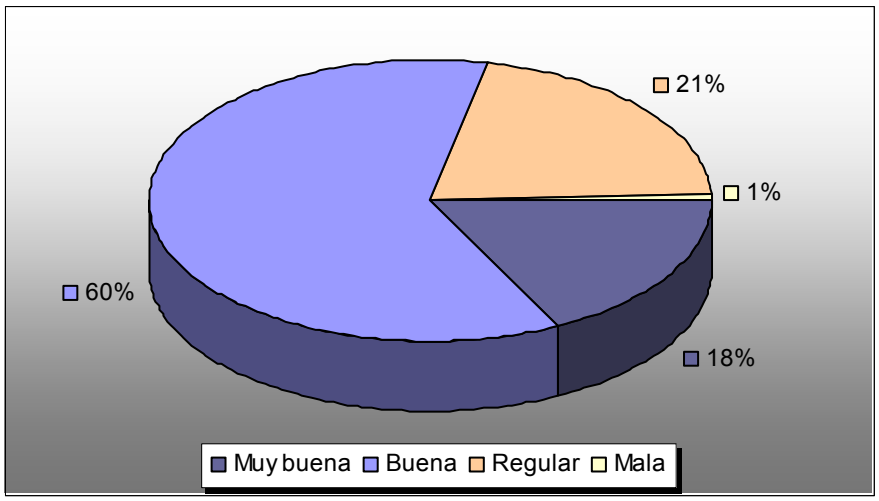
Gráfica 6-6. El programa ayuda a comprender los contenidos teóricos de la asignatura

En las encuestas los alumnos destacaron el aspecto docente del simulador y su función como herramienta para afianzar los conocimientos de la asignatura (Gráfica 6-6). De

hecho, las únicas puntualizaciones que hubo al respecto se debían (y no sin justificación) a que el simulador no es tan útil para alumnos que no dominan los contenidos de la asignatura. Esta objeción se hizo extensiva a la propia ayuda del simulador, que si bien está muy bien valorada por la mayoría de los alumnos, sí se comenta la falta de información que permita un acercamiento a la herramienta sin tener ningún conocimiento previo acerca de la teoría de ILP.

6.3.2. Valoración de la ayuda

Volviendo a la valoración de la ayuda, esto se observa más claramente presentando conjuntamente los resultados de las cuestiones acerca de la ayuda con las opiniones acerca de los mensajes de error de la aplicación. En la Gráfica 6-7 se puede comprobar que más de un 75% de los alumnos consideran la ayuda suministrada buena o muy buena. Las críticas a la ayuda se deben (aparte de su escasa utilidad para usuarios sin base en ILP como ya se nombró) a la ausencia de botones que permitan acceder de forma explícita a la ayuda en determinados puntos del programa (el acceso mediante la tecla F1 se consideraba insuficiente) o al poco contenido de ciertos mensajes de error. En general se valora muy positivamente el contenido actual de la ayuda y simplemente se sugiere su ampliación con más información, en especial a un nivel más básico, o con la inclusión de más ejemplos, tales como una traza completa de la simulación de un código en las máquinas comentada de forma didáctica.



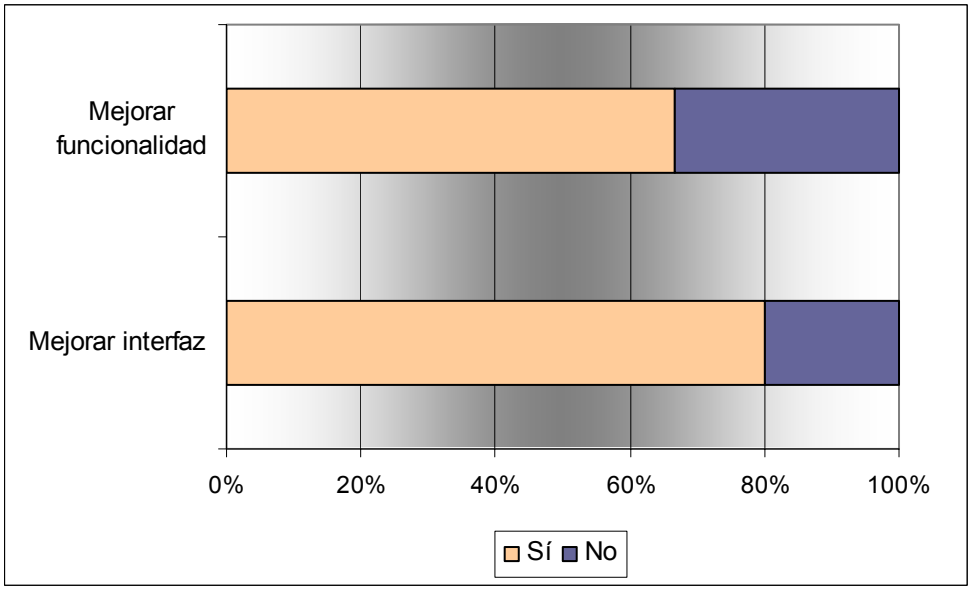
Gráfica 6-7. Valoración de la ayuda de la aplicación

6.3.3. Mejoras de la aplicación

A los alumnos se les preguntó explícitamente si veían mejorable la aplicación, tanto en sus funcionalidades, como en su interfaz gráfica (ver Gráfica 6-8).

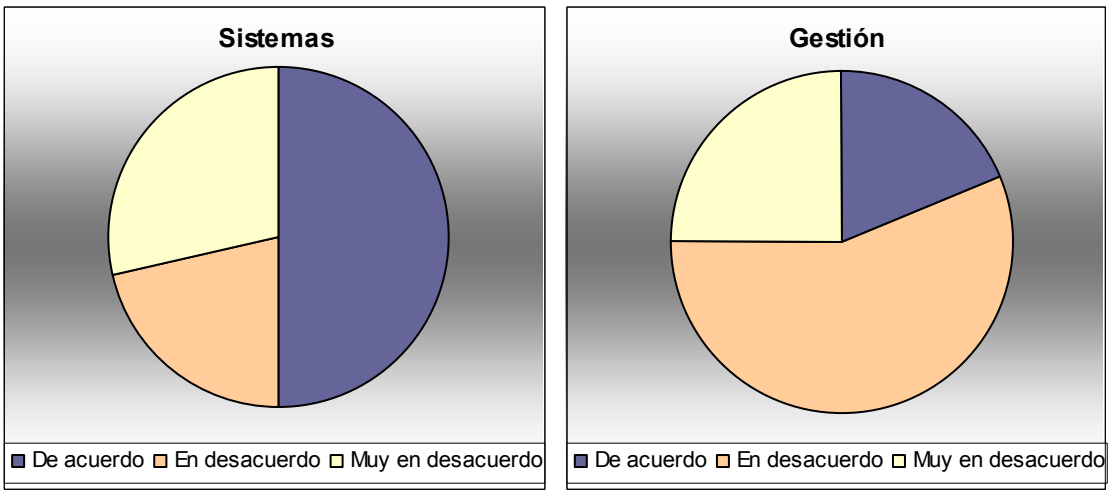
Preguntar por la interfaz gráfica era inevitable, ya que es una de las partes cruciales del programa, y es ésta realmente la parte más mejorable del simulador. Las críticas más repetidas entre los alumnos se referían a la accesibilidad de ciertas opciones, y aportaban soluciones como añadir menús contextuales con cada ventana o ampliar las opciones de las barras de herramientas. Otra crítica bastante extendida tenía que ver con la presentación en pantalla de la máquina superescalar, excesivamente cargada de información, aunque los mismos alumnos reconocían que el mayor impacto se producía en los primeros usos del programa y después se iban acostumbrando. Algunas soluciones propuestas incluían el uso de pestañas para solapar la información (Ilustración 6-1).

Ilustración 6-1. Uso de pestañas para presentar la información



Gráfica 6-8. Posibilidad de mejoras en el simulador

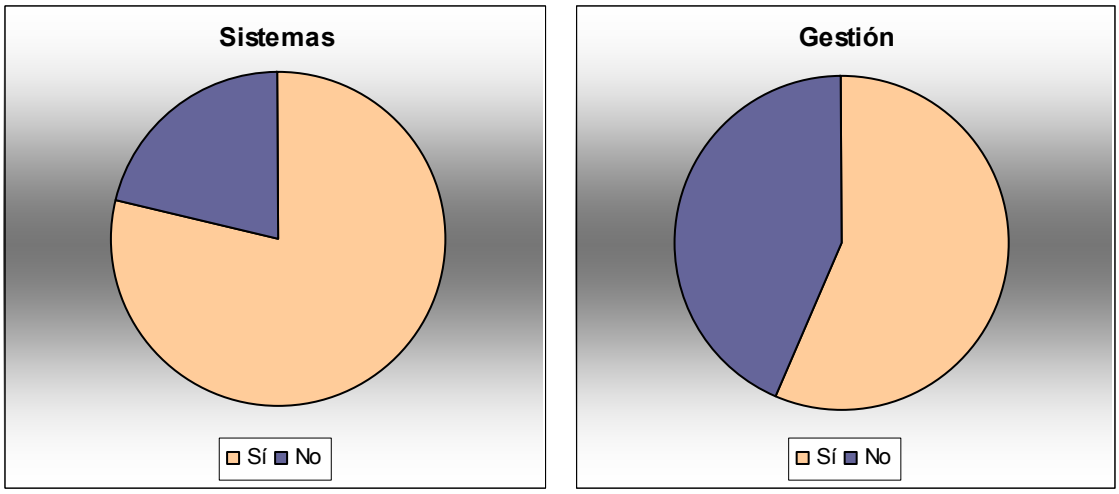
Se debe destacar que, a pesar del alto índice de alumnos que opinaban que la interfaz era mejorable, a la pregunta de si es "lo suficientemente clara" sí que se presentaban discrepancias entre los alumnos provenientes de Sistemas y los de Gestión (Gráfica 6-9). La mitad de los alumnos de Sistemas consideraron que la interfaz se entendía sin problemas, mientras que los de Gestión encontraron más dificultades en su comprensión.



Gráfica 6-9. Comparativa Sistemas-Gestión: "La interfaz es suficientemente clara"

En la presentación que se hizo en clase se les pidió a los alumnos que intentaran ser especialmente críticos y, pese a ello, más de un tercio de los alumnos consideró suficientes las funcionalidades del simulador. De hecho, existe incluso una clara diferencia en este aspecto entre los alumnos provenientes de Gestión y los de Sistemas, como se observa en la Gráfica 6-10. Los alumnos de Sistemas, quizás por su mayor familiaridad con el tema por su formación, consideraron un mayor número de mejoras,

mientras los provenientes de Gestión se sintieron más satisfechos con las funcionalidades existentes.



Gráfica 6-10. Comparativa Sistemas-Gestión entre la necesidad de mejora de las funcionalidades

Aquellos alumnos que requerían más funcionalidades presentaron un buen número de ideas, algunas de las cuales se desarrollarán en la sección 7.2.

7. CONCLUSIONES

7.1. Conclusiones finales

Tras finalizar este proyecto, se dispone de un simulador de Arquitecturas ILP con planificación dinámica y estática que puede emplearse como herramienta de apoyo en la docencia de Arquitectura de Computadores.

Las dos máquinas que componen el simulador se han diseñado de tal manera que dispongan de una robusta base común (unidades funcionales, memoria, registros…). La clave de la base común era permitir el uso de un repertorio de instrucciones basado en el repertorio MIPS IV en los programas de ejemplo del simulador para los dos tipos de máquina. Sobre esta base se han tomado las correspondientes decisiones de diseño para construir:

- Un procesador superescalar, cuyo control está basado en el algoritmo de Tomasulo. Su estructura hardware incluye, aparte de los elementos comunes, una unidad de prebúsqueda, un decodificador, un *reorder buffer*, estaciones de reserva y una tabla de predicción de salto que emplea una estrategia de 2 bits. Además se han añadido algunos campos de control en componentes comunes.
- Un procesador VLIW, de gran simplicidad en su hardware. Tan sólo incluye respecto a los componentes comunes, registros de predicado y bits NaT asociados a los registros. Para poder emplear las operaciones del repertorio MIPS se puso a disposición de los usuarios una serie de herramientas para construir el código de instrucciones largas partiendo del código secuencial, y de esta forma reemplazar al compilador.

Este diseño se obtuvo tras numerosas reuniones y la consulta de una gran diversidad de fuentes tras algo más de 4 meses de trabajo (Agosto 2003 – Noviembre 2003). La implementación del simulador con C++ Builder llevó desde el final del diseño hasta Marzo de 2004, con más de 500 horas de trabajo y el resultado final de 589696 líneas de código. Como sabrá cualquiera que haya programado una aplicación con interfaz gráfica, fueron los detalles de la presentación y el diseño de los componentes gráficos los problemas más costosos en tiempo. Cabe destacar el esfuerzo realizado en mantener independientes la implementación de las máquinas y sus componentes con respecto a la

interfaz gráfica del programa. Esto facilitará futuras mejoras de las estructuras del simulador o de la presentación gráfica sin que los cambios afecten a los dos ámbitos conjuntamente.

Una vez terminado el programa, se presentó a los alumnos de la asignatura de Arquitectura e Ingeniería de Computadores de 5º curso de la Ingeniería Informática para su validación. El resultado de este estudio fue de satisfacción general entre los alumnos por esta primera versión de la herramienta, destacando su aspecto pedagógico y, sobre todo, la ayuda integrada con el programa. En el lado negativo, se incidió en la complejidad intrínseca a la presentación visual de las máquinas, en especial la máquina superescalar. Sin embargo, las críticas vinieron acompañadas de un buen número de sugerencias para mejorar la presentación de la máquina.

Como ya se nombró en la introducción, no existen más herramientas con las características de ésta. La combinación del punto de vista de la planificación dinámica mediante una máquina superescalar, y el punto de vista de la planificación estática mediante una máquina VLIW, ofrece una visión global de las arquitecturas ILP que difícilmente puede obtenerse mediante los simuladores existentes, incluso haciendo uso de varios simuladores. En especial, el acercamiento que se hace a las máquinas VLIW es totalmente novedoso al hacer que el alumno se convierta en compilador y madure, mediante la interacción con el programa, el uso de las diversas técnicas que emplean los auténticos compiladores. No se puede ocultar algo que el alumno también descubrirá: la enorme complejidad de la tarea del compilador, que compite con la complejidad del hardware en la máquina superescalar.

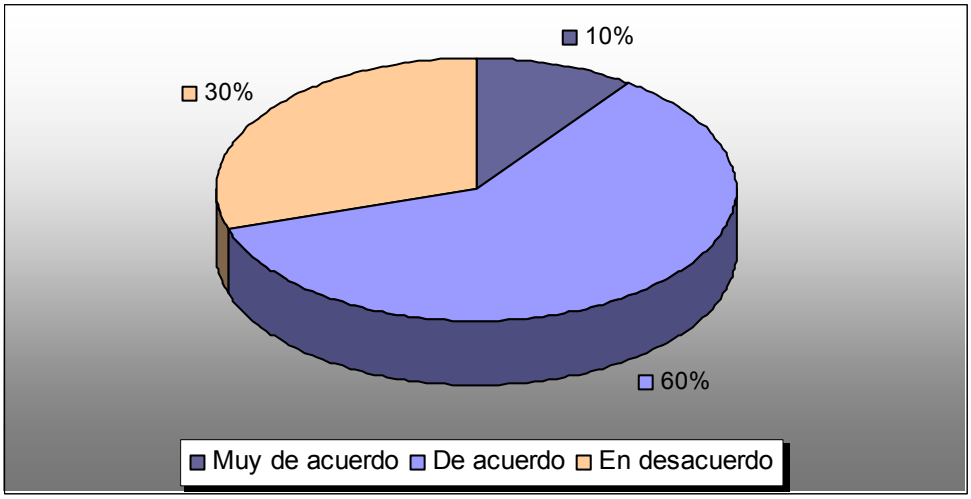
La herramienta está diseñada como un complemento a la enseñanza teórica de las clases de Arquitectura y no pretende sustituir la información que de forma magnífica presentan textos como el de Hennessy y Patterson [13]. Su utilización debe entenderse, por tanto, como un medio de afianzar los conocimientos adquiridos mediante el estudio de la teoría. En el mejor de los casos también puede servir para iluminar ciertos pasajes que sobre el papel quedan oscuros y, solamente viendo el funcionamiento real de la máquina, pueden comprenderse de una forma más amplia.

7.2. Líneas de trabajo abiertas

Como colofón de esta memoria, resulta interesante resaltar las posibles mejoras y ampliaciones que se podrían aplicar con mayor o menor dificultad al simulador. Estas ideas, provenientes de las opiniones vertidas por los alumnos en las encuestas y combinadas con el trabajo del autor, permitirían desde aprovechar en mayor grado la potencia del programa hasta facilitar su uso o comprensión.

7.2.1. Mejora de las funcionalidades

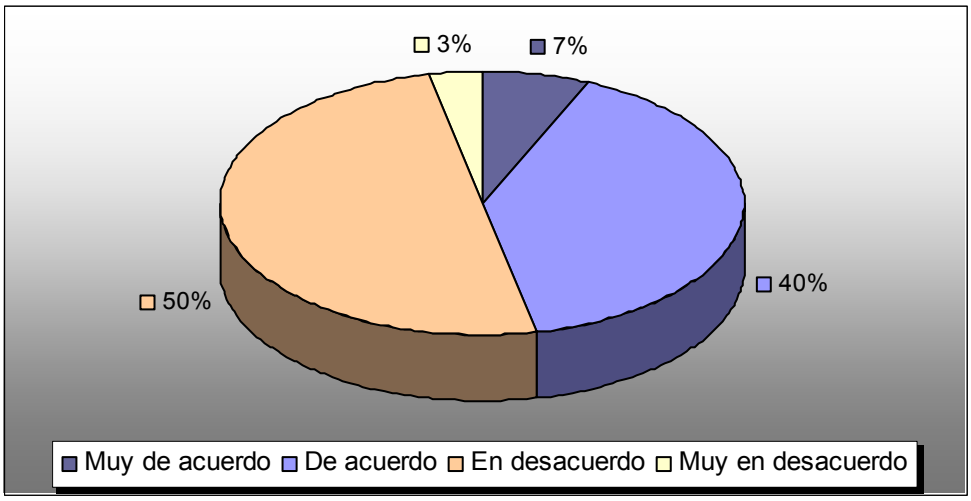
- Incluir un **editor de código secuencial** integrado en el mismo entorno. Aunque en las encuestas la mayor parte de los alumnos afirmaba que la creación de códigos secuenciales no era complicada tal como estaba planteada (Gráfica 7-1), sí que sugerían integrar la edición de los códigos en el programa. De esta manera se puede conseguir un mayor dinamismo en el desarrollo de nuevos ejemplos y en la corrección de los ejemplos ya creados.



Gráfica 7-1. La creación de nuevos códigos secuenciales es sencilla

- Agrupar los ficheros de entrada para ejemplos en **proyectos**. La clave de esta idea es evitar los problemas que causan los ficheros de entrada de códigos VLIW, que presuponen un código secuencial y una configuración de máquina determinada. El "proyecto" consistiría en un código secuencial, una configuración o configuraciones válidas de la máquina y un conjunto de ficheros de código de instrucciones largas asociados. Partiendo de esta base se puede potenciar muchísimo la creación de ejemplos: añadir o quitar ficheros del proyecto, delimitar la configuración máxima y mínima de la máquina, establecer un conjunto de códigos VLIW válidos…
- Añadir una opción de **ejecución simplificada**. Para facilitar la adaptación del alumno al esquema de la máquina superescalar se podría activar una opción que simplificara la ventana de ejecución, mostrando sólo los elementos y campos imprescindibles para su comprensión. El exceso de información de la ejecución superescalar (un aspecto que criticaron bastante los alumnos, aunque admitiendo que era la información que debía mostrarse) podría verse aligerado al menos en los primeros usos del simulador. Una vez familiarizado el alumno con el esquema podría activar la versión **completa** de la simulación y profundizar en el funcionamiento superescalar.
- Completar los **modos de ejecución** de las simulaciones (continua o "paso a paso") con nuevas opciones: ejecución hasta un ciclo determinado, posibilidad de retroceder en la ejecución...
- Facilitar el seguimiento de las instrucciones no sólo con colores, sino con una especie de **diagrama de flujo de ejecución** que muestre todas las fases por las que ha pasado una instrucción y los ciclos de reloj en los que ha sucedido. Este nuevo tipo de visión de la simulación permitiría enfocar la ejecución no sólo desde el punto de vista de la máquina, sino también del de una instrucción individual.
- Añadir un **asistente de creación de códigos VLIW**. Otra de las partes más complejas del simulador era la construcción de códigos de instrucciones largas (Gráfica 7-2). Una forma de ayudar al alumno consistiría en la inclusión en el simulador de un asistente que le oriente acerca de posibles ubicaciones de una

operación, o que cree un código automáticamente que sirva de base para su posterior modificación.



Gráfica 7-2. La creación de nuevos códigos VLIW se realiza de forma clara y sencilla

- Guardar las **trazas** de las simulaciones realizadas en ficheros, de forma que puedan revisarse posteriormente y comparar los resultados con otras simulaciones Esta utilidad podría verse incrementada en caso de adaptar estas trazas para que puedan servir como entrada a otros programas, como programas de simulación de caches.
- Permitir la ejecución de ejemplos en **modo "bat"** o desde **línea de comandos**, de manera que puedan ejecutarse varios ejemplos sin interacción del usuario. Esta opción debería ser complementada con la anterior de creación de ficheros de trazas, o al menos el almacenamiento del estado final de la máquina.
- Incluir **herramientas de verificación y de estadísticas** que permitan comparar varias simulaciones. Estas utilidades podrían complementarse con las dos anteriores. Servirían para comprobar si dos ejecuciones dieron el mismo resultado o para comparar el número de ciclos que tardó la simulación, porcentaje de ocupación de las unidades funcionales, ciclos en los que se añadieron burbujas (debido a los *LOAD* en la máquina VLIW o a recursos llenos en la máquina superescalar)…
- Al margen de las herramientas del punto anterior, también podría incluirse una utilidad de **estimación del coste hardware** de una máquina. Esta herramienta heurística asignaría un coste a cada componente y a cada ampliación de la máquina. De esta forma podría tenerse un parámetro adicional para comparar la ejecución en una máquina superescalar (donde cada ampliación de unidades funcionales tiene un coste muy alto) a la ejecución en una máquina VLIW (que puede disponer de más unidades funcionales sin casi coste añadido) del mismo código.

7.2.2. Ampliaciones de las máquinas

En cuanto a mejoras que podrían aplicarse a las propias máquinas siempre debe tenerse en cuenta el objetivo del simulador, que es la docencia. Teniendo esto en mente resulta evidente que complicar mucho más los esquemas de las máquinas sería contraproducente, así que añadir nuevos elementos hardware no parece ser una solución

excesivamente acertada. Sin embargo, hay elementos cuya adición dotaría de mayores posibilidades docentes al simulador sin complicar el esquema ya existente:

- Ampliar el repertorio de instrucciones, de tal manera que se puedan diseñar ejemplos más realistas.
- Añadir nuevos tipos de estrategias de predicción de salto en la máquina superescalar, como predicción múltiple, y permitir elegir una u otra para comparar los resultados.
- Construir una jerarquía de memoria que permitiera controlar de una forma más precisa que con porcentajes los fallos de caché. Esta jerarquía se aplicaría a su vez a la caché de instrucciones, lo que supondría añadir elementos (como *buffer* de instrucciones) para tratar los fallos de caché de instrucciones.
- Diseñar un pequeño compilador para la máquina VLIW que preprocesara los códigos secuenciales: desenrollado de bucles, *software pipelining*…

Algunos de estos elementos podrían añadirse como módulos independientes para reforzar otro tipo de conocimientos. Una opción sería que emplearan como entrada ficheros de trazas generados por el propio simulador o que se usaran siempre en ejecuciones sin interacción del usuario, simplemente para comprobar resultados.

Se puede pensar en multitud de ampliaciones hardware más, como *trace cache [20]* o el incremento del detalle de ciertos elementos. En cualquier caso, todas estas ampliaciones tendrían que implantarse con el mayor de los cuidados para evitar incrementar excesivamente la complejidad del programa y romper de esta forma con la idea original de ilustración de conceptos que debe preceder al simulador.

REFERENCIAS

[1] Agarwal A., Kubiatowicz J., Kranz D., Lim B.-H., Yeung D., D'Souza G., Parkin M., *Sparcle: An evolutionary processor design for large-scale multiprocessors*, IEEE Micro 13, págs. 48-61, Junio 1993

[2] Agerwala T., Cocke J., *High performance reduces instruction set processors*, IBM Tech. Rep., Marzo 1987

[3] Alverson G., Alverson R., Callahan D., Koblenz B., Porterfield A., Smith B., *Exploiting heterogeneous parallelism on a multithreaded multiprocessor*, Proc. 1992 Int'l Conf. on Supercomputing, págs. 188-197, Noviembre 1992

[4] Anderson D.W., Sparacio F.G., Tomasulo R.M., *The IBM 360 Model 91: Processor philosophy and instruction handling*, IBM J. Research and Development 11:1, págs. 8-24, Enero 1967

[5] Bakoglu H.B. et al., *IBM second-generation RISC processor organization*, Proc. Int'l Conf. on Computer Design, IEEE, Rye, N.Y., págs. 138-142, Octubre 1989

[6] Clark W.A., *The Lincoln TX-2 computer development*, Proc. Western Joint Computer Conference, Institute of Radio Engineers, Los Angeles, págs. 143-145, Febrero 1957

[7] Colwell et al., *A VLIW Architecture for a Trace Scheduling Compiler*, IEEE Transactions on Computers, Vol. 37, Nº 8, Agosto 1988

[8] Cohn R., Gross T., Lam M., Tseng P.S., A*rchitecture and compiler tradeoffs for a long instruction word microprocessor.* Proc. ASPLOS III, págs. 2-14, 1989

[9] De Dinechin B.D., *A static control superscalar architecture*. Proc. MICRO-25, págs. 282-291, 1992

[10] Ferscha, A., Tripathi, S.K., *Parallel and Distributed Simulation of Discrete Event Systems*, Universidad de Maryland en el College Park Technical Report nº CS-TR-3336, Agosto 1994

[11] Fisher J.A., *Very long instruction word architectures and the ELI-512*. Proc. 10th AISCA, págs. 140-150, 1983

[12] Gwennap L., *PPC 604 Past Pentium*, Microprocessor Report, págs. 5-8, 18 Abril 1994

[13] Hennesy J.L., Patterson D.A., *Computer Architecture: A Quantitative Approach (3rd edition)*, Morgan Kaufmann, 2003

[14] Hennesy J.L., Patterson D.A., *Estructura y Diseño de Computadores. Interficie Circuitería/Programación*. Ed. Reverté S.A., 2000

[15] Huck J., Morris D., Ross J., Knies A., Mulder H., Zahir R., *Introducing the IA-64 Architecture*, IEEE MICRO, Sept-Oct 2000

[16] Hwu W.-M., Y. Patt, *HPSm, a high performance restricted data flow architecture having minimum functionality*, Proc. 13th Symposium on Computer Architecture, Tokyo, págs. 297-307, Junio 1986

[17] Johnson M., *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, N.J., 1990

[18] Lam M., *Software pipelining: An effective scheduling technique for VLIW machines*. Proc. SIGPLAN '88 Conference on Programming Language Design and Implementation, págs. 318-328, 1988

[19] Lauterbach G., Horel T., *UltraSPARC-III: Designing third generation 64-bit performance*, IEEE Micro 19:3, Mayo/Junio 1999

[20] Patel S.J., *Trace Cache Design for Wide-Issue Superscalar Processors*, PhD Thesis, The University of Michigan, 1999

[21] Rau B.R., Glaeser C.D., Picard R.L., *Efficient code generation for horizontal architectures: compiler techniques and architectural support*. Proc. 9th AISCA, págs. 131-139, 1982

[22] Rau B.R., Yen D.W.L., Yen W., Towle R., *The Cydra 5 departmental supercomputer*. Computer, 22, Enero, págs. 112-125. 1989

[23] Sima, D., Fountain, T., Kacsuk, P., *Advanced Computer Architectures: A Design Space Approach*. Addison-Wesley, Harlow, England, 1997.

[24] Smith B.J., *A pipelined, shared resource MIMD computer*, Proc. 1978 ICPP, págs. 6-8, Agosto 1978.

[25] Smith J.E., Pleszkun A.R. *Implementation of precise interrupts in pipelined processors*. En Proceedings of the 12th International Symposium on Computer Architecture, págs. 36-44, Junio 1985

[26] Smith J.E., *Dynamic instruction scheduling and the Astronautics ZS-1*, Computer 22:7, págs. 21-35, Julio 1989

[27] Smith M.D., Johnson M., Horowitz M.A., *Limits on multiple instruction issue*, Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems, IEEE/ACM, Boston, págs. 290-302, Abril 1989.

[28] Sohi G.S., *Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers*, IEEE Trans. on Computers 39:3, págs. 349-359, Marzo 1990

[29] Sussenguth E., *IBM's ACS-1 Machine*, IEEE Computer 22:11, Noviembre 1999

[30] Thornton J.E., *Design of a Computer, the Control Data 6600*, Scott, Foresman, Glenview, I11, 1970

[31] Tomasulo R.M., *An efficient algorithm for exploiting multiple arithmetic units*, IBM J. Research and Development 11:1, págs. 25-33, Enero 1967

[32] Tullsen D.M., Eggers S.J., Levy H.M., *Simultaneous multithreading: Maximizing on-chip parallelism*, Proc. 22nd Int'l Symposium on Computer Architecture, págs. 392-403, Junio 1995

[33] Wolfe A., Shen J.P., *A variable instruction stream extension to the VLIW architecture*, Proc. Fourth Conf. on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, págs. 2-14, Abril 1991

[34] Yamamoto W., Serrano M.J., Talcott A.R., Wood R.C., Nemirosky M., *Performance estimation of multistreamed, superscalar processors*, Proc. 27th Hawaii Int'l Conf. on system Sciences, I:195-204, Enero 1994

ANEXO A. CÓDIGO FUENTE

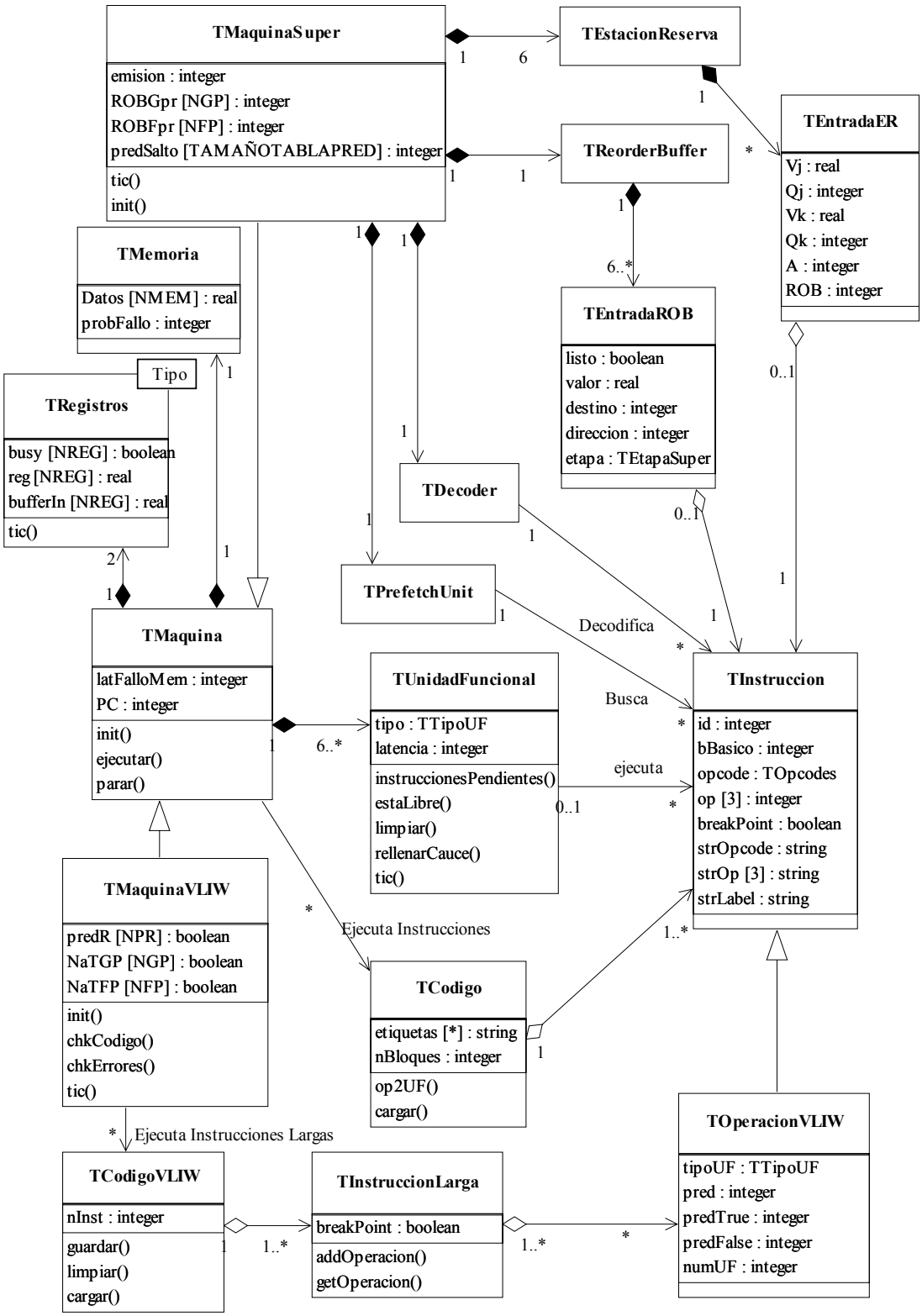


Ilustración A-1. Diagrama de clases de SIMDE

En este anexo se incluye el código en C++ de las clases que tienen un papel más destacado en el funcionamiento del programa. Se han obviado las clases que implementan los formularios, y en general las referidas a la presentación gráfica, para

hacer hincapié en aquellas clases que describen el funcionamiento interno de las máquinas. En la ilustración A-1 puede observarse un diagrama que representa las relaciones entre las distintas clases.

A.1. Analizador léxico de los códigos secuenciales

Este primer código es el analizador léxico de los ficheros de código secuencial que se emplean como ejemplos para el simulador. Está escrito en **flex**.

```
%option noyywrap
%{
/*************************************************************
 * simdeLex.l
 * ParserWizard generated Lex file.
 *
 * AUTOR: Iván Castilla Rodríguez
 *
 * UTILIDAD: Fichero de análisis léxico para los ficheros de
 * entrada del simulador
 *
 * FECHA: miércoles, 27 de agosto de 2003
 *************************************************************/
#include "simdeLex.h"
%}

H               [A-Fa-f0-9]
D               [0-9]
E               [Ee][+-]?{D}+
id              [A-Za-z_][A-Za-z0-9_]*
espacio         [ \t\v\f]
direccion       {D}*"("[Rr]{D}+")"
%%

^{D}+           {/* Esto es el número de líneas del fichero */
                    return LEXNLINEAS;
                }
#{D}+           { return LEXINMEDIATO; }
[Ff]{D}+        { return LEXREGFP; }
[Rr]{D}+        { return LEXREGGP; }
{id}            { return LEXID; }
{id}":"         { return LEXETIQUETA; }
{direccion}     { return LEXDIRECCION; }
"//".*          { /* Comentario */ }
{espacio}+      { /* Espacio en blanco */ }
(.|\n)          { /* Cosas extrañas y retornos de carro */ }
%%

int setYyin(char *nombre) {
    yyin = fopen(nombre, "r");
    if (yyin == NULL)
        return -1;
    return 0;
}

int unsetYyin() {
    return fclose(yyin);
}

char *getYytext() {
    return yytext;
}
```

A.2. Clase TCola

A.2.1. Fichero TCola.h

```
#ifndef TColaH
#define TColaH

/*****************************************************************
 * CLASE: TCola
 *
 * FICHERO: TCola.h
 *
 * AUTOR: Iván Castilla Rodríguez
 *
 * DESCRIPCION: Plantilla para una cola de tamaño fijo que almacena
 * cualquier tipo de datos
 *
 * PARAMETROS DEL TEMPLATE:
 *  - T: Clase base de la cola
 *
 * COMENTARIOS: Está implementada como una cola circular con una
 * posición de más que sirve como "centinela" de final.
 *****************************************************************/
template <class T>
class TCola {
private:
    int size;  // El núm. de elementos real de la cola (1 más de los
útiles)
    T **elements;
    int last, first;

public:
    // Constructores
    __fastcall TCola() : size(0), elements(NULL), last(0), first(0) {}
    __fastcall TCola(int n) : size(n + 1), last(0), first(0) {
        elements = new T*[n + 1];
        memset(elements, NULL, sizeof(T *) * (n + 1));
    }

    // Destructor
    __fastcall ~TCola() {
        delete []elements;
    }

    // Init
    void __fastcall init(int n) {
        size = n + 1;
        delete []elements;
        elements = new T*[n + 1];
        memset(elements, NULL, sizeof(T *) * (n + 1));
        first = last = 0;
    }

    void __fastcall clear() {
        for (int i = 0; i < size; i++) {
            delete elements[i];
            elements[i] = NULL;
        }
        first = last = 0;
    }
```

```
    // Getters
    int __fastcall getSize() { return size - 1; }
    int __fastcall getFirst() { return first; }
    int __fastcall getLast() { return last; }
    int __fastcall getCount() {
        return (last >= first) ? (last - first) : (last + size -
first);
    }
    bool __fastcall isEmpty() { return (first == last); }
    bool __fastcall isFull() { return ((last + 1) % size == first); }
    int __fastcall rel2Abs(int pos) { return (first + pos) % size; }

    T* operator[] (int ind) const {
        if (ind >= size)
            return NULL;
        return elements[ind];
    }

    int __fastcall add(T *obj) {
        if (isFull())
            return -1;
        int oldUltimo = last;
        elements[last] = obj;
        last = (last + 1) % size;
        return oldUltimo;
    }
    T *__fastcall remove() {
        if (isEmpty())
            return NULL;
        T *aux = elements[first];
        elements[first] = NULL;
        first = (first + 1) % size;
        return aux;
    }
    T *__fastcall remove(int pos) {
        if (pos == first)
            return remove();
        if (last > first) {
            if ((pos >= last) || (pos < first))
                return NULL;
        }
        else {
            if ((pos >= last) && (pos < first))
                return NULL;
        }
        T *aux = elements[pos];
        last = (last > pos) ? last : last + size;
        for (int i = pos; i < last; i++)
            elements[i % size] = elements[(i + 1) % size];
        last = (last - 1) % size;
        return aux;
    }
    T * __fastcall top() { return elements[first]; }

    class iterator;
    friend class iterator;
    class iterator {
    private:
        TCola& queue;
        int index;
    public:
```

```
        iterator(TCola& c) : queue(c), index(queue.first) {}
        // Para crear el centinela del final
        iterator(TCola& c, bool) : queue(c), index(queue.last) {}
        // Para crear un iterador empezando en una posición concreta
        iterator(TCola& c, int pos) : queue(c), index(pos) {}
        // Constructor de copia:
        iterator(const iterator& rv) : queue(rv.queue),
index(rv.index) {}
        iterator& operator=(const iterator& rv) {
            queue = rv.queue;
            index = rv.index;
            return *this;
        }
        iterator& operator++() {
            index = (index + 1) % queue.size;
            return *this;
        }
        iterator& operator++(int) {
            return operator++();
        }
        iterator& operator--() {
            index = (index + queue.size - 1) % queue.size;
            return *this;
        }
        iterator& operator--(int) {
            return operator--();
        }
        T* current() const { return queue.elements[index]; }
        T* operator*() const { return current(); }
        T* operator->() const { return current(); }
        // Borrar el elemento actual
        T* remove() { return queue.remove(index); }
        // Test de comparación:
        bool operator==(const iterator& rv) const {
            return index == rv.index;
        }
        bool operator!=(const iterator& rv) const {
            return index != rv.index;
        }
        // Para obtener el índice actual del iterador
        int getIndex() { return index; }
    };
    iterator begin() { return iterator(*this); }
    iterator end() { return iterator(*this, true); }
    iterator position(int pos) { return iterator(*this, pos); }
};
#endif
```

A.3. Clase TInstruccion

A.3.1. Fichero TInstruccion.h

```
#ifndef TInstruccionH
#define TInstruccionH
/**************************************************************
 * CLASE: TInstruccion
 *
 * FICHERO: TInstruccion.h, TInstruccion.cpp
 *
 * AUTOR: Iván Castilla Rodríguez
 *
```

```
 * DESCRIPCION: Estructura básica de una instrucción (operación) de un
 * código secuencial
 *
 * COMENTARIOS: Representa una instrucción de código secuencial.
 * El formato de las instrucciones es siempre el mismo: un código de
 * operacion (opcode) y 3 operandos (op) que pueden contener un valor
 * válido o no según el opcode.
 ****************************************************************/
class TInstruccion {
private:
    int id;          // Identificador, igual al orden en código secuen.
    int bBasico;     // Índice del bloque básico al que pertenece
    int opcode;      // Código de operación
    int op[3];       // Valores de los 3 operandos
    AnsiString strOpcode;    // Cadena de caracteres con el opcode
    AnsiString strOp[3];     /* Cadena de caracteres que representa
cada operando */
    AnsiString strLabel;     /* Cadena de caracteres que sirve de
etiqueta que apunta a esta instrucción. Vacío si la instrucción
no tiene ninguna etiqueta asociada */
    bool breakPoint;         /* Indica si la instrucción ha sido
marcada como punto de ruptura de la ejecución */
    TColor color;    // Color con el que se representaría esta instr.

    void __fastcall copiar(TInstruccion &inst);

public:
    // Constructor
    __fastcall TInstruccion() {
        breakPoint = false;
        color = clWindow;
    }
    __fastcall TInstruccion(TInstruccion &inst);


    // Destructor
    __fastcall ~TInstruccion() {}

    // Función de copia
    TInstruccion & operator=(TInstruccion &inst);

    // Getters
    int getId() { return id; }
    int getBBasico() { return bBasico; }
    int getOpcode() { return opcode; }
    int getOp(int ind) { return op[ind]; }
    AnsiString getStrLabel() { return strLabel; }
    AnsiString getStrOpcode() { return strOpcode; }
    AnsiString getStrOp(int ind) { return strOp[ind]; }
    AnsiString getStr();
    bool getBreakPoint() { return breakPoint; }
    TColor getColor() { return color; }

    // Setters
    void setId(int i) { id = i; }
    void setBBasico(int bb) { bBasico = bb; }
    void setOpcode(int op) { opcode = op; }
    void setOpcode(int op, const AnsiString &strOp) {
        opcode = op;
        strOpcode = strOp;
    }
```

```
    void setOp(int ind, int val) { op[ind] = val; }
    void setOp(int ind, int val, const AnsiString &text) {
        op[ind] = val;
        strOp[ind] = text;
    }
    void setStrLabel(const AnsiString &lab) { strLabel = lab; }
    void setStrOpcode(const AnsiString &op) { strOpcode = op; }
    void setStrOp(int ind, const AnsiString &op) { strOp[ind] = op; }
    void setBreakPoint(bool b) { breakPoint = b; }
    void setColor(TColor c) { color = c; }
};

#endif
```

A.3.2. Fichero TInstruccion.cpp

```
#include <vcl.h>
#pragma hdrstop

#include "TInstruccion.h"

//---------------------------------------------------------------------------
#pragma package(smart_init)

__fastcall TInstruccion::TInstruccion(TInstruccion &inst) {
    copiar(inst);
}

TInstruccion & TInstruccion::operator=(TInstruccion &inst) {
    copiar(inst);
    return *this;
}

/*****************************************************************
 * copiar: Copia los atributos de una instrucción en la instrucción
 * actual
 * Parámetros:
 *  - inst: Instrucción origen de la copia
 *****************************************************************/
void __fastcall TInstruccion::copiar(TInstruccion &inst) {
    id = inst.id;
    bBasico = inst.bBasico;
    opcode = inst.opcode;
    op[0] = inst.op[0];
    op[1] = inst.op[1];
    op[2] = inst.op[2];
    strOpcode = AnsiString(inst.strOpcode);
    strOp[0] = AnsiString(inst.strOp[0]);
    strOp[1] = AnsiString(inst.strOp[1]);
    strOp[2] = AnsiString(inst.strOp[2]);
    strLabel = AnsiString(inst.strLabel);
    breakPoint = inst.breakPoint;
    color = inst.color;
}

/*****************************************************************
 * getStr: Obtiene una representación como cadena de caracteres de la
 * instrucción actual.
 *****************************************************************/
AnsiString TInstruccion::getStr() {
    AnsiString aux;
```

```
    if (strOp[1] != AnsiString(""))
        aux = " " + strOp[1];
    if (strOp[2] != AnsiString(""))
        aux = aux  + " " + strOp[2];
    return strOpcode + AnsiString(" ") + strOp[0] + aux;
}
```

A.4. Clase TCodigo

A.4.1. Fichero TCodigo.h

```
#ifndef __SIMDECODE__
#define __SIMDECODE__
/*****************************************************************
 * CLASE: TCodigo
 *
 * FICHERO: TCodigo.h, TCodigo.cpp
 *
 * AUTOR: Iván Castilla Rodríguez
 *
 * FECHA: 16/10/03
 *
 * DESCRIPCION: Contiene el código secuencial que sirve de base para
 * las máquinas simuladas.
 *
 * ABREVIATURAS EN ESTE FICHERO:
 *  - bb: bloque básico
 *
 * COMENTARIOS: El código se carga desde un fichero y se construye la
 * lista de bloques básicos simultáneamente.
 *****************************************************************/
#include "TInstruccion.h"
#include "TUnidadFuncional.h"

#define PREF_FP 'F'
#define PREF_GP 'R'
#define SUF_ETIQUETA ':'
#define CODERR_FICHERO -1
#define CODERR_SINTAX -2

/* Estructura de la lista de bloques básicos */
typedef struct bBasico {
    int nLinea;                 // Línea de la primera instr. del bb
    int id;                     // Nº identificador del bb
    struct bBasico *sig;        // Siguiente bb de la lista
    struct bSucesor *sucesor;   // Lista de sucesores de este bb
} bBasico;

/* Estructura de la lista de sucesores de bloques básicos */
typedef struct bSucesor {
    bBasico *bloque;        // Apunta al bb sucesor
    struct bSucesor *sig;   // Siguiente sucesor
} bSucesor;

//---------------------------------------------------------------------
class TCodigo {
public:
    typedef enum {NOP = 0, ADDI, ADDF, DADDUI, MULTI, MULTF, SI, SF,
LI, LF, BNE, BEQ, OPERROR} TOpcodes;
    // No cuento OPERROR como opcode en sí
    static const int NOPCODES = OPERROR - NOP;
```

```
private:
    static const AnsiString nomOpcode[];     // Nombres de los opcodes
    int nLineas;                             // Nº de líneas de código
    TInstruccion *instrucciones;             // Array de instrucciones
    TStringList *etiquetas;                  // Lista de etiquetas
    bBasico *bloques;                        // Lista de bb
    int nBloques;                            // Contador del núm. de bb

    int __fastcall sustituirEtiquetas();
    TOpcodes __fastcall str2Opcode(AnsiString str);
    int __fastcall str2Registro(AnsiString str);
    int * __fastcall str2Direccion(AnsiString str);
    int __fastcall str2Inmediato(AnsiString str);
    bBasico *__fastcall addEtiqueta(AnsiString str, int nLinea,
bBasico *actual);
    int __fastcall chkEtiqueta(AnsiString str, bBasico *actual);

public:
    // Constructor
    __fastcall TCodigo();

    // Destructor
    __fastcall ~TCodigo();

    // GETTERS
    TInstruccion * __fastcall getInstrucciones() { return
instrucciones; }
    TInstruccion *__fastcall getInstruccion(int ind) { return
&instrucciones[ind]; }
    AnsiString __fastcall getEtiquetaString(int ind) { return
etiquetas->Strings[ind]; }
    bBasico * __fastcall getEtiquetaValor(int ind) { return (bBasico
*) etiquetas->Objects[ind]; }
    AnsiString __fastcall getNomOpcode(int ind) { return
nomOpcode[ind]; }
    int __fastcall getNLineas() { return nLineas; }
    int __fastcall getInstruccionBB(int indBB);

    int __fastcall cargar(AnsiString nombreFichero);
    TTipoUF __fastcall getTipoUF(int ind);


    /*****************************************************************
     * op2UF: Dado un opcode, devuelve el tipo de unidad funcional
     * donde debe ejecutarse esa instrucción.
     *****************************************************************/
    static TTipoUF __fastcall op2UF(int op) {
        switch (op) {
            case ADDI:
            case DADDUI: return SUMAENT;
            case ADDF: return SUMAFLOT;
            case MULTI: return MULTENT;
            case MULTF: return MULTFLOT;
            case SI:
            case SF:
            case LI:
            case LF: return MEM;
            case BNE:
            case BEQ: return SALTO;
            default: return SUMAENT;
```

```
        }
    }
};

// Declaración de las constantes estáticas
const AnsiString TCodigo::nomOpcode[] = {
"NOP", "ADDI", "ADDF", "DADDUI", "MULTI", "MULTF", "SI", "SF", "LI",
"LF", "BNE", "BEQ"};

#endif
```

A.4.2. Fichero TCodigo.cpp

```
#include <vcl.h>
#pragma hdrstop

#include <FStream.h>
#include <stdlib.h>
#include "TCodigo.h"
#include "simdeLex.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)

extern "C" int yylex(void);
extern "C" int setYyin(char *nombre);
extern "C" int unsetYyin();
extern "C" char *getYytext();

//---------------------------------------------------------------------------
__fastcall TCodigo::TCodigo() {
    etiquetas = new TStringList;
    etiquetas->Sorted = true;
    etiquetas->Duplicates = dupError;
    instrucciones = NULL;
    bloques = NULL;
    nBloques = 0;
}

//---------------------------------------------------------------------------
__fastcall TCodigo::~TCodigo() {
    if (instrucciones != NULL)
        delete []instrucciones;
    // Los objetos de las etiquetas son bbs que se eliminan después
    delete etiquetas;
    // Eliminar la estructura de bloques básicos
    while (bloques !=  NULL) {
        bBasico *baux = bloques->sig;
        bSucesor *suc = bloques->sucesor;
        while (suc != NULL) {
            bloques->sucesor = suc->sig;
            delete suc;
            suc = bloques->sucesor;
        }
        delete bloques;
        bloques = baux;
    }
}

/*****************************************************************
 * str2Opcode: Dada una cadena con un opcode, devuelve el código
 * numérico asociado a ese opcode
```

```
 ***************************************************************/
TCodigo::TOpcodes __fastcall TCodigo::str2Opcode(AnsiString str) {
    int i;
    for (i = 1; i < NOPCODES; i++)
        if (nomOpcode[i] == str)
            break;
    if (i == NOPCODES)
        return OPERROR;
    return (TOpcodes) i;
}
/***************************************************************
 * str2Registro: Dada una cadena con un registro de la forma "Rn" o
 * "Fn", devuelve el número (n) del registro.
 ***************************************************************/
int __fastcall TCodigo::str2Registro(AnsiString str) {
    return str.SubString(2, str.Length() - 1).ToInt();
}

/***************************************************************
 * str2Direccion: Dada una cadena con una dirección de la forma
 * "offset(Rn)", devuelve un array que contiene en su posición 0 el
 * "offset" y en la posición 1 el índice n del registro.
 ***************************************************************/
int * __fastcall TCodigo::str2Direccion(AnsiString str) {
    int *resultado = new int[2];
    int pos = str.Pos("(");
    if (pos == 1)
        resultado[0] = 0;
    else
        resultado[0] = str.SubString(1, pos - 1).ToInt();
    resultado[1] = str2Registro(str.SubString(pos + 1, str.Length() -
pos - 1));
    return resultado;
}

/***************************************************************
 * str2Inmediato: Dada una cadena con un valor inmediato de la forma
 * "#Inm", devuelve el valor de este inmediato.
 ***************************************************************/
int __fastcall TCodigo::str2Inmediato(AnsiString str) {
    return str.SubString(2, str.Length() - 1).ToInt();
}

/***************************************************************
 * chkEtiqueta: Tratamiento de etiquetas en una instrucción de salto
 * Parámetros:
 * - str: Nombre de la etiqueta
 * - actual: Puntero al bb actual
 * Devuelve: El índice
 * Comentarios:
 * Una instrucción de salto obliga a añadir un sucesor al bb actual
 * (el bb destino del salto, el otro sucesor se añade al encontrar la
 * primera instrucción del siguiente bb). Si la etiqueta no existe ya
 * (salto hacia delante) se debe crear el bb destino del salto aunque
 * no se añadirá a la lista de bbs ni se rellenará su nº de línea (se
 * pone un -1) hasta encontrar la instrucción marcada por la etiqueta
 * como destino. También se debe añadir la etiqueta a la lista de
 * etiquetas. En caso de existir ya la etiqueta basta con tomar su bb
 * asociado.
 ***************************************************************/
int __fastcall TCodigo::chkEtiqueta(AnsiString str, bBasico *actual) {
```

```
    int ind;
    bBasico *bbas;

    // Se crea el nuevo sucesor
    actual->sucesor = new bSucesor;
    actual->sucesor->sig = NULL;
    if (etiquetas->Find(str + AnsiString(":"), ind)) // Ya está creada
        bbas = (bBasico *)etiquetas->Objects[ind];
    else {
        // Se crea un nuevo bloque básico
        bbas = new bBasico;
        bbas->sig = NULL;
        bbas->sucesor = NULL;
        bbas->nLinea = -1;
        // Se añade la etiqueta
        ind = etiquetas->Add(str + AnsiString(":"));
        etiquetas->Objects[ind] = (TObject *) bbas;
    }
    actual->sucesor->bloque = bbas;
    return ind;
}

/*****************************************************************
 * addEtiqueta: Tratamiento de etiquetas que marcan una línea de
 * código
 * Parámetros:
 * - str: Nombre de la etiqueta
 * - nLinea: Nº de línea de código donde está la etiqueta
 * - actual: Puntero al bb actual
 * Devuelve: El nuevo bb que se crea o NULL en caso de error
 * Modifica:
 * - bloques: En caso de ser la primera instrucción del código
 * Comentarios:
 * Al encontrar una etiqueta debe introducirse un nuevo bb en la lista
 * de bbs. La función debe comprobar si existe ya la etiqueta
 * (proviene de un salto hacia delante)o no. En caso de existir ya la
 * etiqueta se toma su bb asociado y se enlaza en la lista de bbs,
 * rellenándose su campo de nº de línea (si este campo ya tiene un
 * valor <> -1 quiere decir que la etiqueta está repetida y es un
 * error). Si la etiqueta no existe se crea un nuevo bb que se enlaza
 * a la lista y se añade una nueva entrada a la lista de etiquetas.
 *****************************************************************/
bBasico *__fastcall TCodigo::addEtiqueta(AnsiString str, int nLinea,
bBasico *actual) {
    int ind;
    bBasico *bbas;

    if (etiquetas->Find(str, ind)) {// Ya está creada
        bbas = (bBasico *)etiquetas->Objects[ind];
        if (bbas->nLinea != -1) // Etiqueta repetida
            bbas = NULL;
        else {
            bbas->nLinea = nLinea;
            bbas->id = nBloques - 1;
            actual->sig = bbas;
        }
    }
    else {  // Etiqueta nueva
        // Se crea un nuevo bloque básico
        bbas = new bBasico;
        bbas->sig = NULL;
```

```
            bbas->sucesor = NULL;
            bbas->nLinea = nLinea;
            bbas->id = nBloques - 1;
            // Se añade la etiqueta
            ind = etiquetas->Add(str);
            etiquetas->Objects[ind] = (TObject *) bbas;
            if (bloques == NULL) // primer bloque
                bloques = bbas;
            else {
                actual->sig = bbas;
                // Se añade a la lista de sucesores
                bSucesor *suc = new bSucesor;
                suc->bloque = bbas;
                suc->sig = actual->sucesor;
                actual->sucesor = suc;
            }
        }
        return bbas;
}

/*****************************************************************
 * sustituirEtiquetas: Sustituye en el tercer operando de las
 * instrucciones de salto el índice de la etiqueta asociada por el
 * Identificador del bb destino del salto
 * Devuelve: 0 si no hay ningún error, -1 si no se resolvió el destino
 * de algún salto
 *****************************************************************/
int __fastcall TCodigo::sustituirEtiquetas() {
    for (int i = 0; i < nLineas; i++) {
        if (instrucciones[i].getOpcode() == BNE ||
instrucciones[i].getOpcode() == BEQ) {
            bBasico *bbas = (bBasico *) etiquetas-
>Objects[instrucciones[i].getOp(2)];
            if (bbas->nLinea == -1)
                return -1;
            instrucciones[i].setOp(2, bbas->id);
        }
    }
    return 0;
}

/*****************************************************************
 * cargar: Rellena la estructura que contiene las instrucciones que se
 * emplearán leyendo de un fichero.
 * Parámetros:
 * - nombreFichero: Nombre del fichero de entrada
 *****************************************************************/
int __fastcall TCodigo::cargar(AnsiString nombreFichero) {
#define CHKLEXEMA(l)    if (lexema != (l)) { \
                            unsetYyin(); \
                            return CODERR_SINTAX; \
                        }
#define GETLEXEMA lexema = yylex(); yytext = getYytext();

    int *res;
    int lexema;
    char *yytext;
    bBasico *actual;
    bool nuevoBloque = true;

    if (setYyin(nombreFichero.c_str()) != 0)
```

```
        return CODERR_FICHERO;

    // Lo primero que debe encontrarse es el nº de líneas de código
    GETLEXEMA;
    if (lexema != LEXNLINEAS) {
        unsetYyin();
        return CODERR_SINTAX;
    }
    nLineas = AnsiString(yytext).ToInt();

    // Se inicializa el array de instrucciones
    instrucciones = new TInstruccion[nLineas];
    for (int i = 0; i < nLineas; i++) {
        instrucciones[i].setId(i);
        GETLEXEMA;
        if (lexema == LEXETIQUETA) {
            nBloques++;
            instrucciones[i].setStrLabel(yytext);
            actual = addEtiqueta(yytext, i, actual);
            if (actual == NULL) {
                unsetYyin();
                return CODERR_SINTAX;
            }
            // Leo el siguiente lexema
            GETLEXEMA;
        }
        else {
            instrucciones[i].setStrLabel("");
            if (nuevoBloque) {
                nBloques++;
                bBasico *bbas = new bBasico;
                bbas->nLinea = i;
                bbas->sig = NULL;
                bbas->sucesor = NULL;
                bbas->id = nBloques - 1;
                if (bloques == NULL) // primer bloque
                    bloques = actual = bbas;
                else {
                    actual->sig = bbas;
                    // Se añade a la lista de sucesores
                    bSucesor *suc = new bSucesor;
                    suc->bloque = bbas;
                    suc->sig = actual->sucesor;
                    actual->sucesor = suc;
                    actual = actual->sig;
                }
            }
        }
        nuevoBloque = false;
        CHKLEXEMA(LEXID);
        int op = str2Opcode(yytext);
        instrucciones[i].setOpcode(op, yytext);
        // Se asigna el identificador de bloque básico
        instrucciones[i].setBBasico(nBloques - 1);
        switch (op) {
            case NOP:   // No se esperan operandos
                instrucciones[i].setOp(0, 0, "");
                instrucciones[i].setOp(1, 0, "");
                instrucciones[i].setOp(2, 0, "");
                break;
            case ADDI:
```

```
            case MULTI:
                GETLEXEMA;
                CHKLEXEMA(LEXREGGP);
                instrucciones[i].setOp(0, str2Registro(yytext),
yytext);
                GETLEXEMA;
                CHKLEXEMA(LEXREGGP);
                instrucciones[i].setOp(1, str2Registro(yytext),
yytext);
                GETLEXEMA;
                CHKLEXEMA(LEXREGGP);
                instrucciones[i].setOp(2, str2Registro(yytext),
yytext);
                break;
            case ADDF:
            case MULTF:
                GETLEXEMA;
                CHKLEXEMA(LEXREGFP);
                instrucciones[i].setOp(0, str2Registro(yytext),
yytext);
                GETLEXEMA;
                CHKLEXEMA(LEXREGFP);
                instrucciones[i].setOp(1, str2Registro(yytext),
yytext);
                GETLEXEMA;
                CHKLEXEMA(LEXREGFP);
                instrucciones[i].setOp(2, str2Registro(yytext),
yytext);
                break;
            case DADDUI:
                GETLEXEMA;
                CHKLEXEMA(LEXREGGP);
                instrucciones[i].setOp(0, str2Registro(yytext),
yytext);
                GETLEXEMA;
                CHKLEXEMA(LEXREGGP);
                instrucciones[i].setOp(1, str2Registro(yytext),
yytext);
                GETLEXEMA;
                CHKLEXEMA(LEXINMEDIATO);
                instrucciones[i].setOp(2, str2Inmediato(yytext),
yytext);
                break;
            case SI:
            case LI:
                GETLEXEMA;
                CHKLEXEMA(LEXREGGP);
                instrucciones[i].setOp(0, str2Registro(yytext),
yytext);
                GETLEXEMA;
                CHKLEXEMA(LEXDIRECCION);
                res = str2Direccion(yytext);
                instrucciones[i].setOp(1, res[0], yytext);
                instrucciones[i].setOp(2, res[1], "");
                delete []res;
                break;
            case SF:
            case LF:
                GETLEXEMA;
                CHKLEXEMA(LEXREGFP);
```

```
                    instrucciones[i].setOp(0, str2Registro(yytext),
yytext);
                    GETLEXEMA;
                    CHKLEXEMA(LEXDIRECCION);
                    res = str2Direccion(yytext);
                    instrucciones[i].setOp(1, res[0], yytext);
                    instrucciones[i].setOp(2, res[1], "");
                    delete []res;
                    break;
                case BNE:
                case BEQ:
                    GETLEXEMA;
                    CHKLEXEMA(LEXREGGP);
                    instrucciones[i].setOp(0, str2Registro(yytext),
yytext);
                    GETLEXEMA;
                    CHKLEXEMA(LEXREGGP);
                    instrucciones[i].setOp(1, str2Registro(yytext),
yytext);
                    GETLEXEMA;
                    CHKLEXEMA(LEXID);
                    instrucciones[i].setOp(2, chkEtiqueta(yytext, actual),
yytext);
                    // Comienza el tratamiento del bloque básico
                    nuevoBloque = true; // Siguiente inst.: nuevo bloque
                    break;
                case OPERROR:
                default:
                    unsetYyin();
                    return CODERR_SINTAX;
            }
        }
        unsetYyin();
        if (sustituirEtiquetas() == -1)
            return CODERR_SINTAX;
        return 0;
}

/*****************************************************************
 * getTipoUF: Devuelve el tipo de U.F. asociado a la instrucción que
 * ocupa la posición "ind"
 * Parámetros:
 * - ind: Número de la instrucción
 *****************************************************************/
TTipoUF __fastcall TCodigo::getTipoUF(int ind) {
    switch (instrucciones[ind].getOpcode()) {
        case ADDI:
        case DADDUI: return SUMAENT;
        case ADDF: return SUMAFLOT;
        case MULTI: return MULTENT;
        case MULTF: return MULTFLOT;
        case SI:
        case SF:
        case LI:
        case LF: return MEM;
        case BNE:
        case BEQ: return SALTO;
        default: return SUMAENT;
    }
}
```

```
/*********************************************************************
 * getInstruccionBB: Devuelve el índice de la primera instrucción del
 * BB indicado
 * Parámetros:
 * - indBB: Indice del BB
 *********************************************************************/
int __fastcall TCodigo::getInstruccionBB(int indBB) {
    if (indBB > nBloques)
        return -1;
    bBasico *actual = bloques;
    for (int i = 0; i < indBB; i++)
        actual = actual->sig;
    return actual->nLinea;
}
```

A.5. Clase TOperacionVLIW

A.5.1. Fichero TOperacionVLIW.h

```
#ifndef TOperacionVLIWH
#define TOperacionVLIWH
/*********************************************************************
 * CLASE: TOperacionVLIW
 *
 * EXTIENDE A: TInstruccion
 *
 * FICHERO: TOperacionVLIW.h
 *
 * AUTOR: Iván Castilla Rodríguez
 *
 * FECHA: 25/03/04
 *
 * DESCRIPCION: Estructura básica de una operación básica de un código
 * VLIW.
 *
 * COMENTARIOS: Representa una operación de una instrucción larga, es
 * decir, una instrucción con información adicional que indica la UF
 * donde debería ejecutarse y los registros de predicado que emplea.
 *********************************************************************/
#include "TUnidadFuncional.h"
#include "TInstruccion.h"

class TOperacionVLIW : public TInstruccion {
private:
    TTipoUF tipoUF; // Tipo de U.F. donde se ejecuta esta operación
    int numUF;      // Número de U.F. donde se ejecuta esta operación
    int pred;       // Reg. de predicado al que está asociada la oper.

    int predTrue;   /* (sólo con instrucciones de salto): Registro de
predicado que se activa cuando el salto es verdadero (se toma) */
    int predFalse;   /* (sólo con instrucciones de salto): Registro de
predicado que se activa cuando el salto es falso (no se toma) */
public:
    // Constructores
    __fastcall TOperacionVLIW() :TInstruccion() {
        pred = 0;
        predTrue = 0;
        predFalse = 0;
    }
    __fastcall TOperacionVLIW(TOperacionVLIW &oper)
:TInstruccion(oper) {
```

```
        tipoUF = oper.tipoUF;
        numUF = oper.numUF;
        pred = oper.pred;
        predTrue = oper.predTrue;
        predFalse = oper.predFalse;
    }
    __fastcall TOperacionVLIW(TInstruccion &inst, TTipoUF t, int n)
        :TInstruccion(inst), tipoUF(t), numUF(n), pred(0),
predTrue(0), predFalse(0) {}

    // Destructor
    __fastcall ~TOperacionVLIW() {}

    // GETTERS
    TTipoUF __fastcall getTipoUF() { return tipoUF; }
    int __fastcall getNumUF() { return numUF; }
    int __fastcall getPred() { return pred; }
    int __fastcall getPredTrue() { return predTrue; }
    int __fastcall getPredFalse() { return predFalse; }

    // SETTERS
    void __fastcall setTipoUF(TTipoUF t) { tipoUF = t; }
    void __fastcall setNumUF(int n) { numUF = n; }
    void __fastcall setPred(int p) { pred = p; }
    void __fastcall setPredTrue(int p) { predTrue = p; }
    void __fastcall setPredFalse(int p) { predFalse = p; }
};

#endif
```

A.6. Clase TInstruccionLarga

A.6.1. Fichero TInstruccionLarga.h

```
#ifndef TInstruccionLargaH
#define TInstruccionLargaH

/*****************************************************************
 * CLASE: TInstruccionLarga
 *
 * FICHERO: TInstruccionLarga.h
 *
 * DESCRIPCION: Representa una instrucción larga de una máquina VLIW.
 *
 * COMENTARIOS: Una instrucción larga se compone de un número
 * indeterminado a priori de operaciones (que se corresponden con
 * instrucciones básicas del código secuencial). Mediante la
 * estructura de vector pueden añadirse tantas operaciones como se
 * desea a la instrucción larga.
 *****************************************************************/
#include <vector.h>
#include "TUnidadFuncional.h"
#include "TOperacionVLIW.h"

class TInstruccionLarga {
private:
    vector<TOperacionVLIW *> operaciones; // Operaciones asociadas
    bool breakPoint;    /* Indica si el ejecutar esta instrucción
implica detener el flujo */
public:
    // Constructor
```

```
    __fastcall TInstruccionLarga() {
        breakPoint = false;
    }

    // Destructor
    __fastcall ~TInstruccionLarga() {
        for (unsigned int i = 0; i < operaciones.size(); i++) {
            TOperacionVLIW *oper = operaciones[i];
            delete oper;
        }
    }

    // GETTERS
    TOperacionVLIW * __fastcall getOperacion(int ind) {
        return operaciones[ind];
    }
    int __fastcall getNOper() { return operaciones.size(); }
    bool getBreakPoint() { return breakPoint; }

    // SETTERS
    void __fastcall setBreakPoint(bool b) { breakPoint = b; }

    /*****************************************************************
     * addOperacion: Añade una nueva operación a esta instrucción
     * larga.
     * Parámetros:
     *  - oper: Operación que se añade
     *****************************************************************/
    void __fastcall addOperacion(TOperacionVLIW *&oper) {
        operaciones.push_back(oper);
    }
};

#endif
```

A.7. Clase TCodigoVLIW

A.7.1. Fichero TCodigoVLIW.h

```
#ifndef TCodigoVLIWH
#define TCodigoVLIWH

/*****************************************************************
 * CLASE: TCodigoVLIW
 *
 * FICHERO: TCodigoVLIW.h, TCodigoVLIW.cpp
 *
 * AUTOR: Iván Castilla Rodríguez
 *
 * FECHA: 05/03/03
 *
 * DESCRIPCION: Contiene el código de instrucciones largas que sirve
 * de base para la máquina VLIW.
 *
 * COMENTARIOS: El código de instrucciones largas se construye como un
 * array de tipo TInstruccionLarga. Cada elemento de este array puede
 * contener un número indeterminado de operaciones (instrucciones
 * básicas). El código VLIW puede guardarse o cargarse desde un
 * fichero.
 *****************************************************************/
#include "TInstruccionLarga.h"
```

```
#include "TUnidadFuncional.h"
#include <vector.h>

//---------------------------------------------------------------------------
class TCodigoVLIW {
private:
    TInstruccionLarga *instrucciones;    // Array de instr. largas
    int nInst;                           // Núm. de inst. largas

public:
    // Constructores
    __fastcall TCodigoVLIW() {
        instrucciones = NULL;
        nInst = 0;
    }
    __fastcall TCodigoVLIW(int n) {
        instrucciones = new TInstruccionLarga[n];
        nInst = n;
    }

    // Destructor
    __fastcall ~TCodigoVLIW() {
        delete []instrucciones;
    }

    // GETTERS
    int __fastcall getNInst() { return nInst; }
    TInstruccionLarga *__fastcall getInstruccionLarga(int ind) {
        if ((ind < 0) || (ind >= nInst))
            return NULL;
        return &instrucciones[ind];
    }
    bool __fastcall getBreakPoint(int ind) { return
instrucciones[ind].getBreakPoint(); }

    // SETTERS
    void __fastcall setNInst(int n) {
        instrucciones = new TInstruccionLarga[n];
        nInst = n;
    }
    void __fastcall setBreakPoint(int ind, bool b) {
instrucciones[ind].setBreakPoint(b); }

    /*************************************************************
     * addOperacion: Añade una nueva operación a una instrucción
     * larga.
     * Parámetros:
     *  - ind: Indice de la instrucción larga a la que se añade la
     * operación
     *  - oper: Operación que se añade
     *************************************************************/
    void __fastcall addOperacion(int ind, TOperacionVLIW *&oper) {
        instrucciones[ind].addOperacion(oper);
    }
    /*************************************************************
     * limpiar: Borra todas las instrucciones largas y pone el
     * contador de instrucciones a 0.
     *************************************************************/
    void __fastcall limpiar() {
        delete []instrucciones;
        instrucciones = NULL;
```

```
        nInst = 0;
    }

    void __fastcall guardar(AnsiString nombre);
    bool __fastcall cargar(AnsiString nombre, TCodigo *&cod);
};

#endif
```

A.7.2. Fichero TCodigoVLIW.cpp

```
#include <vcl.h>
#pragma hdrstop

#include<iostream>
#include<fstream>
#include "TCodigo.h"
#include "TCodigoVLIW.h"

//---------------------------------------------------------------------------
#pragma package(smart_init)

/*****************************************************************
 * guardar: Permite guardar un código de instrucciones largas en un
 * fichero
 * Parámetros:
 *  - nombre: Nombre del fichero
 *****************************************************************/
void __fastcall TCodigoVLIW::guardar(AnsiString nombre) {
    ofstream out(nombre.c_str());

    out << nInst << endl;
    for (int i = 0; i < nInst; i++) {
        int noper = instrucciones[i].getNOper();
        out << noper;
        for (int j = 0; j < noper; j++) {
            TOperacionVLIW *oper = instrucciones[i].getOperacion(j);
            out << "\t" << oper->getId();
            out << " " << oper->getTipoUF();
            out << " " << oper->getNumUF();
            out << " " << oper->getPred();
            /* Si es una operación de salto necesita destino, reg.
               pred. verdadero y reg. pred. falso */
            if ((oper->getOpcode() == TCodigo::BNE) ||
                (oper->getOpcode() == TCodigo::BEQ)) {
                out << " " << oper->getOp(2);
                out << " " << oper->getPredTrue();
                out << " " << oper->getPredFalse();
            }
        }
        out << endl;
    }
}
/*****************************************************************
 * cargar: Lee de un fichero una configuración de las operaciones
 * básicas que constituye un programa completo VLIW
 * Devuelve:
 * - true: Todo correcto
 * - false: Ocurrió algún error
 * Parámetros:
 * - nombre: Nombre del fichero de entrada
```

```
 * - cod: Código secuencial del que se extraen las operaciones
 * originales
 ***********************************************************************/
bool __fastcall TCodigoVLIW::cargar(AnsiString nombre, TCodigo *&cod)
{
    int n, noper, ind, tipo, num;
    ifstream in(nombre.c_str());

    in >> n;
    instrucciones = new TInstruccionLarga[n];
    nInst = n;
    for (int i = 0; i < nInst; i++) {
        in >> noper;
        for (int j = 0; j < noper; j++) {
            int tipo, num, pred, predTrue = 0, predFalse = 0;
            in >> ind;
            in >> tipo;
            in >> num;
            in >> pred;

            if (cod->getTipoUF(ind) != tipo) {
                limpiar();
                return false;
            }
            TOperacionVLIW *oper = new TOperacionVLIW(*cod-
>getInstruccion(ind), (TTipoUF)tipo, num);
            oper->setPred(pred);
            if ((oper->getOpcode() == TCodigo::BNE) ||
                (oper->getOpcode() == TCodigo::BEQ)) {
                int destino, predTrue, predFalse;
                in >> destino;
                oper->setOp(2, destino);
                in >> predTrue;
                in >> predFalse;
                oper->setPredTrue(predTrue);
                oper->setPredFalse(predFalse);
            }
            instrucciones[i].addOperacion(oper);
        }
    }
    return true;
}
```

A.8. Clase TRegistros

A.8.1. Fichero TRegistros.h

```
#ifndef TRegistrosH
#define TRegistrosH

/***********************************************************************
 * CLASE: TRegistros
 *
 * FICHERO: TRegistros.h
 *
 * AUTOR: Iván Castilla Rodríguez
 *
 * FECHA: 21/01/04
 *
 * DESCRIPCION: Plantilla de los bancos de registros
 *
```

```
 * PARAMETROS DEL TEMPLATE:
 *  - T: Clase de los elementos que se almacenan en cada registro
 *  - NREG: Número de registros del banco de registros
 *
 * COMENTARIOS:
 * Existen dos formas de usar los registros:
 * - accediendo directamente a los datos para escritura
 * - diviendo el acceso en dos partes: Primero se escribe en unos
 * buffer de entrada y después se escribe efectivamente el dato en el
 * registro al usar la función "tic". Este segundo modo es útil para
 * la máquina VLIW.
 ****************************************************************/
template <class T, int NREG = 64>
class TRegistros {
private:
    T reg[NREG];         // Contenido del registro
    T bufferIn[NREG];    /* Buffer de entrada (sólo acceso en mitad de
ciclo) */
    bool busy[NREG];     // Reg. con nuevo valor en el bufferIn

public:
    __fastcall TRegistros() {
        memset(busy, false, sizeof(bool) * NREG);
    }

    // GETTERS
    int __fastcall getNReg() { return NREG; }
    T& __fastcall getReg(int ind) { return reg[ind]; }
    bool __fastcall getBusy(int ind) { return busy[ind]; }

    // SETTERS
    void __fastcall setReg(T valor) {
        memset(reg, valor, sizeof(T) * NREG);
        memset(busy, false, sizeof(bool) * NREG);
    }
    /****************************************************************
     * setReg: Asigna un valor a un registro
     * Parámetros:
     *  - ind: Número de registro
     *  - valor: Valor a asignar
     *  - usaBuffer: Indica si se debe emplear el buffer de entrada
     *    (true) o escribir directamente en el registro (false)
     ****************************************************************/
    void __fastcall setReg(int ind, T valor, bool usaBuffer) {
        if (usaBuffer) {
            bufferIn[ind] = valor;
            busy[ind] = true;
        }
        else
            reg[ind] = valor;
    }
    void setBusy(int ind, bool valor) {
        busy[ind] = valor;
    }
    void setBusy(bool valor) {
        memset(busy, false, sizeof(bool) * NREG);
    }

    /****************************************************************
     * tic: Simula un tic de reloj en el fichero de registros
     * Comentarios: Esta función actualiza los valores de los
```

```
     * registros con los valores del buffer de entrada. Por tanto sólo
     * tiene sentido si se ha asignado valores a los registros
     * empleando la opción "usarBuffer"
     ***************************************************************/
    void tic() {
        for (int i = 0; i < NREG; i++)
            if (busy[i]) {
                busy[i] = false;
                reg[i] = bufferIn[i];
            }
    }
};
#endif
```

A.9. Clase TMemoria

A.9.1. Fichero TMemoria.h

```
#ifndef TMemoriaH
#define TMemoriaH

/*****************************************************************
 * CLASE: TMemoria
 *
 * FICHERO: TMemoria.h
 *
 * AUTOR: Iván Castilla Rodríguez
 *
 * FECHA: 18/02/04
 *
 * DESCRIPCION: Estructura de la memoria de la máquina
 *
 * COMENTARIOS:
 * - La memoria se compone de 1024 palabras de 32 bits.
 * - El control de fallos de caché se realiza mediante una
 *   probabilidad de fallo
 * - Al ocurrir un fallo se marca la posición de memoria donde ocurrió
 *****************************************************************/
#include <stdlib.h>
#include <time.h>

class TMemoria {
public:
    static const int NMEM = 1024;    // Nº de palabras de memoria
private:
    float datos[NMEM];  // Contenido de la memoria
    bool fallo[NMEM];   /* Marcador de si ha habido un fallo en una
posición de memoria */
    int probFallo;      // Probabilidad de fallo de caché (en %)
public:
    // Constructores
    __fastcall TMemoria() {
        randomize();
        probFallo = 0;
    }

    // Destructores
    __fastcall ~TMemoria() {}

    // Getters
    /***************************************************************
```

```
     * getDato: Lee un dato de memoria
     * Parámetros:
     *  - dir: Dirección de memoria
     *  - dato: El dato leído
     * Devuelve:
     *  - true: Si la lectura ocurrió sin problemas
     *  - false: Si ocurrió un fallo de caché al realizar la lectura
     ************************************************************/
    bool getDato(int dir, int &dato) {
        dato = (int) datos[dir];
        int valFallo = random(100);
        // Habrá fallo sólo si no hubo un fallo en esta misma posición
        if ((valFallo < probFallo) && !fallo[dir]) {
            fallo[dir] = true;
            return false;
        }
        fallo[dir] = false;
        return true;
    }
    bool getDato(int dir, float &dato) {
        dato = datos[dir];
        int valFallo = random(100);
        // Habrá fallo sólo si no hubo un fallo en esta misma posición
        if ((valFallo < probFallo) && !fallo[dir]) {
            fallo[dir] = true;
            return false;
        }
        fallo[dir] = false;
        return true;
    }
    // Para obtener el dato directamente
    float getDato(int dir) { return datos[dir]; }
    int getProbFallo() { return probFallo; }

    // Setters
    void setMem(float dato) { memset(datos, dato, sizeof(float) *
NMEM); }
    void setDato(int dir, int dato) { datos[dir] = (float) dato; }
    void setDato(int dir, float dato) { datos[dir] = dato; }
    void setProbFallo(int prob) { probFallo = prob; }

};
#endif
```

A.10. Clase TMaquina

A.10.1. Fichero TMaquina.h

```
#ifndef TMaquinaH
#define TMaquinaH
/*****************************************************************
 * CLASE: TMaquina
 *
 * FICHERO: TMaquina.h, TMaquina.cpp
 *
 * AUTOR: Iván Castilla Rodríguez
 *
 * FECHA: 29/01/04
 *
 * DESCRIPCION: Estructura básica de la máquina que se empleará en las
 * simulaciones
```

```
 *
 * ABREVIATURAS EN ESTE FICHERO:
 *  - UF: Unidad Funcional
 *
 * COMENTARIOS:
 * - Tamaño de los operandos: Tanto para entero como punto flotante se
 * emplean 32 bits, que para el Builder se corresponden con el tipo
 * "int" y el tipo "float" respectivamente
 * - La máquina se compone de los siguientes elementos:
 *   - Unidades Funcionales de número y latencia parametrizables
 *   - Banco de 64 registros de propósito general
 *   - Banco de 64 registros de punto flotante
 *   - Memoria
 *   - Contador de Programa (PC)
 * - Y se caracteriza por un estado de ejecución actual
 *****************************************************************/
#include "TUnidadFuncional.h"
#include "TRegistros.h"
#include "TMemoria.h"

/* Representación del estado de la máquina */
typedef struct {
    int ciclo;          // Contador de ciclos
    bool ejecutando;    // Indica si la máquina está funcionando
    bool breakPoint;    // Indica si se activó un breakpoint
} TMaqStatus;

//---------------------------------------------------------------------
class TMaquina {
public:          // User declarations
    static const int LAT_MAX[NTIPOSUF]; // Latencia máxima de cada UF
    static const int LAT_MIN[NTIPOSUF]; // Latencia mínima de cada UF
    static const int LAT_DEF[NTIPOSUF]; // Latencia por defecto de
cada UF
    static const int NUF_MAX[NTIPOSUF]; // Número máximo de cada tipo
de UF
    static const int NUF_MIN[NTIPOSUF]; // Número mínimo de cada tipo
de UF
    static const int NUF_DEF[NTIPOSUF]; // Número por defecto de cada
tipo de UF
    static const int LATFALLOMEM_DEF = 9;   // Latencia por defecto
del fallo de caché
    static const int LATFALLOMEM_MIN = 0;   // Latencia mínima del
fallo de caché
    static const int LATFALLOMEM_MAX = 100; // Latencia máxima del
fallo de caché

    // Constantes de la máquina
    static const int PALABRA = 32;  // Dimensión de la palabra en bits
    static const int NGP = 64;      // Nº de Reg. de Propósito General
    static const int NFP = 64;      // Nº de Reg. de Punto Flotante

protected:      // User declarations

    // Características generales
    int nUF[NTIPOSUF];          // Nº de UF de cada tipo
    int latenciaUF[NTIPOSUF];   // Latencia de cada tipo de UF
    int latFalloMem;            // Latencia del fallo de memoria

    // Componentes
    TUnidadFuncional *UF[NTIPOSUF]; // Unidades Funcionales
```

```
    TRegistros<int, NGP> gpr;       // Registros de Propósito general
    TRegistros<float, NFP> fpr;     // Registros de Punto Flotante
    TMemoria memoria;               // Memoria de la máquina

    unsigned int PC;                // Contador de Programa
    TMaqStatus status;              // Estado de la máquina

public:

    // Constructores
    __fastcall TMaquina();

    // Destructores
    __fastcall ~TMaquina();

    // GETTERS
    int __fastcall getNUF();
    int __fastcall getNUF(int ind) { return nUF[ind]; }
    int __fastcall getLatenciaUF(int ind) { return latenciaUF[ind]; }
    TUnidadFuncional * __fastcall getUF(TTipoUF tipo, int ind) {
return &UF[tipo][ind]; }
    TUnidadFuncional * __fastcall getUFs(TTipoUF tipo) { return
UF[tipo]; }
    int __fastcall getLatFalloMem() { return latFalloMem; }
    int __fastcall getGpr(int ind) {
        if (ind >= NGP || ind < 0)
            return -1;
        else
            return gpr.getReg(ind);
    }
    float __fastcall getFpr(int ind) {
        if (ind >= NFP || ind < 0)
            return -1;
        else
            return fpr.getReg(ind);
    }
    TMemoria *__fastcall getMemoria() { return &memoria; }
    unsigned int __fastcall getPC() { return PC; }
    int __fastcall getCiclo() { return status.ciclo; }
    bool __fastcall getEjecutando() { return status.ejecutando; }
    bool __fastcall getBreakPoint() { return status.breakPoint; }

    // SETTERS
    void __fastcall setNUF(int ind, int n) { nUF[ind] = n; }
    void __fastcall setLatenciaUF(int ind, int l) { latenciaUF[ind] =
l; }
    void __fastcall setLatFalloMem(int f) { latFalloMem = f; }
    void __fastcall setGpr(int ind, int val) {
        if (ind < NGP && ind >= 0)
            gpr.setReg(ind, val, false);
    }
    void __fastcall setFpr(int ind, float val) {
        if (ind < NGP && ind >= 0)
            fpr.setReg(ind, val, false);
    }

    // Inicializadores
    void __fastcall init(bool reset);

    /*************************************************************
     * ejecutar: Activa el estado de ejecución de la máquina
```

```
     ************************************************************/
    void __fastcall ejecutar() {
        status.ejecutando = true;
        status.breakPoint = false;
    }
    /************************************************************
     * parar: Desactiva el estado de ejecución de la máquina
     ************************************************************/
    void __fastcall parar() { status.ejecutando = false; }
};

// Declaración de las constantes estáticas
const int TMaquina::LAT_MAX[NTIPOSUF] = {100, 100, 100, 100, 100,
100};
const int TMaquina::LAT_MIN[NTIPOSUF] = {1, 1, 1, 1, 1, 1};
const int TMaquina::LAT_DEF[NTIPOSUF] = {1, 2, 4, 6, 4, 2};
const int TMaquina::NUF_MAX[NTIPOSUF] = {10, 10, 10, 10, 10, 10};
const int TMaquina::NUF_MIN[NTIPOSUF] = {1, 1, 1, 1, 1, 1};
const int TMaquina::NUF_DEF[NTIPOSUF] = {2, 2, 2, 2, 2, 1};
#endif
```

A.10.2. Fichero TMaquina.cpp

```
#include <vcl.h>
#pragma hdrstop

#include "TMaquina.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)

__fastcall TMaquina::TMaquina() {
    int size = NTIPOSUF * sizeof(int);
    // Inicializamos las variables a sus valores por defecto
    memcpy(latenciaUF, LAT_DEF, size);
    memcpy(nUF, NUF_DEF, size);
    latFalloMem = LATFALLOMEM_DEF;
    for (int i = 0; i < NTIPOSUF; i++)
        UF[i] = NULL;
    init(true);
}

__fastcall TMaquina::~TMaquina() {
    for (int i = 0; i < NTIPOSUF; i++)
        delete [] (UF[i]);
}

/************************************************************************
 * getNUF: Obtiene el número total de UF de que dispone la máquina
 ************************************************************************/
int __fastcall TMaquina::getNUF() {
    int suma = 0;
    for (int i = 0; i < NTIPOSUF; i++)
        suma += nUF[i];
    return suma;
}

/************************************************************************
 * init: Inicializa los componentes de la máquina para comenzar una
 * ejecución
 * Parámetros:
 *  - reset: Si es verdadero obliga a poner a 0 a todos los
```

```
 * componentes de la máquina.
 * Comentarios: Para inicializar la estructura que contiene las UF
 * primero se inicializa el array de cada tipo de UF, después se
 * inicializa cada UF con su tipo correspondiente y su latencia.
 ***************************************************************/
void __fastcall TMaquina::init(bool reset) {
    PC = 0;
    for (int i = 0; i < NTIPOSUF; i++) {
        delete [](UF[i]);
        UF[i] = new TUnidadFuncional[nUF[i]];
        for (int j = 0; j < nUF[i]; j++) {
            UF[i][j].setTipo((TTipoUF)i);
            UF[i][j].setLatencia(latenciaUF[i]);
        }
    }
    status.ciclo = 0;
    status.breakPoint = false;
    status.ejecutando = false;
    if (reset) {
        gpr.setReg(0);
        fpr.setReg(0.0);
        memoria.setMem(0.0);
    }
}
```

A.11. Clase TMaquinaSuper

A.11.1. Fichero TMaquinaSuper.h

```
#ifndef TMaquinaSuperH
#define TMaquinaSuperH
/***************************************************************************
 * CLASE: TMaquinaSuper
 *
 * EXTIENDE A: TMaquina
 *
 * FICHERO: TMaquinaSuper.h, TMaquinaSupercpp
 *
 * AUTOR: Iván Castilla Rodríguez
 *
 * FECHA: 16/04/04
 *
 * DESCRIPCION: Estructura de la máquina Superescalar que se empleará
 * en las simulaciones.
 *
 * COMENTARIOS:
 * - Se caracteriza por un grado de emisión de instrucciones
 * - Se compone de los siguientes elementos:
 *   - Estaciones de Reserva (ER). Construidas como listas de
 *     entradas, ya que pueden eliminarse entradas intermedias
 *   - Reorder Buffer (ROB). Construido como una cola circular.
 *   - Tabla de predicción de salto de 2 bits
 *   - Unidad de prebúsqueda. Construida como una cola circular
 *   - Decodificador. Construido como una cola circular
 *   - Unidad funcional de cálculo de direcciones
 *   - Tablas de mapeo entre los registros (FPR y GPR) y el ROB
 * - Además se le asocia un código secuencial como programa para ser
 *   ejecutado
 ***************************************************************************/
#include "TMaquina.h"
#include "TCodigo.h"
```

```
#include "TCola.h"
#include <list.h>

// Definición de valores de error
typedef enum {SUPER_COMMITOK = 0, SUPER_COMMITEND, SUPER_COMMITMISS,
SUPER_COMMITNO} TCommitStatus;
typedef enum {SUPER_ISSUE = 0, SUPER_EXECUTE, SUPER_WRITERESULT,
SUPER_COMMIT} TEtapaSuper;
typedef enum {SUPER_ENDEXE = -2, SUPER_BREAKPOINT = -1, SUPER_OK = 0}
TSuperscalarStatus;

//---------------------------------------------------------------------
class TMaquinaSuper : public TMaquina {
public:
    // Núm. de bits de predicción de salto
    static const int NBITSPRED = 2;
    // Núm. de bits del tamaño de la tabla de predicción de salto
    static const int NBITSTABLAPRED = 4;
    // Tamaño de la tabla de predicción de salto
    static const int TAMANOTABLAPRED = 1 << NBITSTABLAPRED;

    /* Estructura de cada entrada del ROB */
    typedef struct {
        TInstruccion *instruccion;  // Instrucción que ocupa esa
posición del ROB
        bool listo;                 // Indica si el resultado ya está
calculado
        float valor;                // El resultado de la instrucción
calculado
        int destino;                // Índice del registro destino
        int direccion;              // Simula en cierta forma el
buffer de stores
        TEtapaSuper etapa;          // Indicador de etapa actual de
ejecución
    } TEntradaROB;

    /* Estructura de cada entrada de las ER */
    typedef struct {
        TInstruccion *instruccion; // Puntero a la instr. en la UF
        int Qj;         /* Entrada del ROB que producirá el primer
operando (0 si el op. ya está disponible en Vj) */
        int Qk;         /* Entrada del ROB que producirá el segundo
operando (0 si el op. ya está disponible en Vk) */
        float Vj;       // Valor del primer operando
        float Vk;       // Valor del segundo operando
        int A;          // Valor inmediato o dir. efectiva calculada
        int ROB;        // Entrada del ROB que contiene la instrucción
        int numUF;      /* Número de la UF donde se está ejecutando
(-1 si no se está ejecutando) */
        int posUF;      // Posición de la UF donde se está ejecutando
    } TEntradaER;

    /* Estructura de cada entrada del Decodificador */
    typedef struct {
        TInstruccion *instruccion;
    } TEntradaDecoder;

    /* Estructura de cada entrada de la unidad de prebúsqueda */
    typedef struct {
        TInstruccion *instruccion;
    } TEntradaPrefetch;
```

```
    typedef TCola<TEntradaROB> TReorderBuffer;
    typedef list<TEntradaER> TEstacionReserva;
    typedef TCola<TEntradaPrefetch> TPrefetchUnit;
    typedef TCola<TEntradaDecoder> TDecoder;
private:
    int emision;        // Grado de emisión
    TCodigo *codigo;    // Código secuencial a ejecutar
    // Componentes de la máquina
    int ROBGpr[NGP];                // Tabla de mapeo ROB<->GPR
    int ROBFpr[NFP];                // Tabla de mapeo ROB<->FPR
    TEstacionReserva ER[NTIPOSUF];  // Estaciones de Reserva
    TReorderBuffer ROB;             // Reorder Buffer
    unsigned int predSalto[TAMANOTABLAPRED];  /* Tabla de predicción
de salto */
    TPrefetchUnit prefetchUnit;     // Unidad de prebúsqueda
    TDecoder decoder;               // Decodificador
    TUnidadFuncional *aluMem;       // UFs de cálculo de direcciones

    void __fastcall chkRegistro(int reg, bool fp, float &v, int &q);
    void __fastcall emitirInstruccion(TInstruccion *inst, int tipo,
int indROB);
    void __fastcall ejecutarInstruccion(TTipoUF tipo, int num);
    void __fastcall escribirInstruccion(TTipoUF tipo, int num);
    int __fastcall ticPrefetch();
    int __fastcall ticDecoder();
    int __fastcall ticIssue();
    int __fastcall ticExecute();
    int __fastcall ticWriteResult();
    TCommitStatus __fastcall ticCommit();
    /*****************************************************************
     * prediccion: Devuelve si el salto correspondiente a una
     * dirección debe tomarse o no.
     * Parámetros:
     *  - dir: Dirección de la instrucción de salto
     *****************************************************************/
    bool __fastcall prediccion(int dir) {
        return (predSalto[dir % TAMANOTABLAPRED] >= 2);
    }
    /*****************************************************************
     * cambiarPrediccion: Actualiza el valor de una predicción de
     * salto en función del resultado del último salto.
     * Parámetros:
     *  - dir: Dirección de la instrucción de salto
     *  - resul: Resultado del último salto en esa dirección
     *****************************************************************/
    void __fastcall cambiarPrediccion(int dir, bool resul) {
        dir = dir % TAMANOTABLAPRED;
        switch (predSalto[dir]) {
            case 0:     predSalto[dir] = (resul) ? 1:0; break;
            case 1:     predSalto[dir] = (resul) ? 3:0; break;
            case 2:     predSalto[dir] = (resul) ? 3:0; break;
            case 3:     predSalto[dir] = (resul) ? 3:2; break;
            default:    predSalto[dir] = 0; break;
        }
    }
    bool __fastcall chkSalto(TEntradaROB *rob);
    bool __fastcall chkStore(int indROB, int dir);

    // Funciones de predicado para las listas
    bool __fastcall predOpDisponibles(TEntradaER e);
```

```
public:
    // CONSTANTES
    static const int EMISION_DEF = 4;    // Valor por defecto del grado
de emisión
    static const int EMISION_MIN = 2;    // Valor mínimo del grado de
emisión
    static const int EMISION_MAX = 16;   // Valor máximo del grado de
emisión
    static const AnsiString NOMBREETAPA[];  /* Nombres de las etapas
de proceso de una instrucción */

    // Constructores
    __fastcall TMaquinaSuper();

    // Destructores
    __fastcall ~TMaquinaSuper();

    // GETTERS
    int __fastcall getEmision() { return emision; }
    TCodigo *getCodigo() { return codigo; }
    TEstacionReserva * __fastcall getER(int ind) { return &ER[ind]; }
    int __fastcall getTamER(TTipoUF tipo) { return (nUF[tipo] *
(latenciaUF[tipo] + 1)); }
    TReorderBuffer * __fastcall getROB() { return &ROB; }
    TPrefetchUnit * __fastcall getPrefetchUnit() { return
&prefetchUnit; }
    TDecoder * __fastcall getDecoder() { return &decoder; }
    TUnidadFuncional * __fastcall getAluMem(int ind) { return
&aluMem[ind]; }
    TUnidadFuncional * __fastcall getAluMem() { return aluMem; }
    int * __fastcall getROBGpr() { return ROBGpr; }
    int * __fastcall getROBFpr() { return ROBFpr; }
    unsigned int * __fastcall getPredSalto() { return predSalto; }
    unsigned int __fastcall getPredSalto(int ind) { return
predSalto[ind]; }

    // SETTERS
    void __fastcall setEmision(int e) { emision = e; }
    void __fastcall setCodigo(TCodigo *cod) { codigo = cod; }

    // Inicializador
    void __fastcall init(bool reset);

    TSuperscalarStatus __fastcall tic();
};

// Definición de constantes estáticas
const AnsiString TMaquinaSuper::NOMBREETAPA[] = {"ISSUE", "EXEC",
"WRITE", "COMMIT"};
#endif
```

A.11.2. Fichero TMaquinaSuper.cpp

```
#include <vcl.h>
#pragma hdrstop

#include "TMaquinaSuper.h"

//---------------------------------------------------------------------
#pragma package(smart_init)
```

```
__fastcall TMaquinaSuper::TMaquinaSuper() :TMaquina() {

    emision = EMISION_DEF;

    memset(ROBGpr, -1, sizeof(int) * NGP);
    memset(ROBFpr, -1, sizeof(int) * NFP);
    memset(predSalto, 0, sizeof(unsigned int) * TAMANOTABLAPRED);
    // Calculo el tamaño del ROB
    int total = 0;
    for (int i = 0; i < NTIPOSUF; i++) {
        ER[i].clear();
        total += getTamER(i);
    }
    ROB.init(total);
    prefetchUnit.init(2 * emision);
    decoder.init(emision);
    codigo = NULL;
    aluMem = new TUnidadFuncional[nUF[MEM]];
    for (int j = 0; j < nUF[MEM]; j++) {
        aluMem[j].setTipo(SUMAENT);
        aluMem[j].setLatencia(latenciaUF[SUMAENT]);
    }
}

__fastcall TMaquinaSuper::~TMaquinaSuper() {
    delete []aluMem;
}

/*****************************************************************
 * init: Inicializa los componentes de la máquina para comenzar una
 * ejecución
 * Parámetros:
 *  - reset: Si es verdadero obliga a poner a 0 a todos los
 * componentes de la máquina.
 *****************************************************************/
void __fastcall TMaquinaSuper::init(bool reset) {
    TMaquina::init(reset);
    memset(ROBGpr, -1, sizeof(int) * NGP);
    memset(ROBFpr, -1, sizeof(int) * NFP);
    memset(predSalto, 0, sizeof(unsigned int) * TAMANOTABLAPRED);
    // Calculo el tamaño del ROB
    int total = 0;
    for (int i = 0; i < NTIPOSUF; i++) {
        ER[i].clear();
        total += getTamER(i);
    }
    ROB.init(total);
    prefetchUnit.init(2 * emision);
    decoder.init(emision);
    delete []aluMem;
    aluMem = new TUnidadFuncional[nUF[MEM]];
    for (int j = 0; j < nUF[MEM]; j++) {
        aluMem[j].setTipo(SUMAENT);
        aluMem[j].setLatencia(latenciaUF[SUMAENT]);
    }
}

/*****************************************************************
 * ticPrefetch: Ejecuta la etapa de prefetch (pre-búsqueda) de
 * instrucciones en la memoria de instrucciones.
 * Devuelve:
```

```
 *  - El número de instrucciones que se han podido "pre-buscar"
 * Comentarios: Simula el funcionamiento de la unidad de PREFETCH, que
 * permite cargar las instrucciones del flujo de ejecución de manera
 * transparente al resto de la máquina.
 ***********************************************************************/
int __fastcall TMaquinaSuper::ticPrefetch() {
    while (!prefetchUnit.isFull() && (PC < codigo->getNLineas())) {
        TEntradaPrefetch *aux = new TEntradaPrefetch;
        /* Importante: Hago una copia de la instrucción original para
distinguir las distintas apariciones de una misma inst. */
        aux->instruccion = new TInstruccion(*codigo-
>getInstruccion(PC));
        if ((TCodigo::op2UF(aux->instruccion->getOpcode()) == SALTO)
&& prediccion(PC))
            PC = codigo->getInstruccionBB(aux->instruccion->getOp(2));
        else
            PC++;
        prefetchUnit.add(aux);
    }
    return prefetchUnit.getCount();
}

/***********************************************************************
 * ticDecoder: Lee las "emision" primeras intrucciones de la unidad de
 * Prefetch y las pone en el Decodificador
 * Devuelve:
 *  - El número de instrucciones que se han podido introducir en el
 *  decodificador
 ***********************************************************************/
int __fastcall TMaquinaSuper::ticDecoder() {
    TPrefetchUnit::iterator it = prefetchUnit.begin();
    for (; (!decoder.isFull()) && (it != prefetchUnit.end()); it++) {
        TEntradaPrefetch *aux = it.remove();
        TEntradaDecoder *nuevaDec = new TEntradaDecoder;
        nuevaDec->instruccion = aux->instruccion;
        decoder.add(nuevaDec);
        delete aux;
    }
    return decoder.getCount();
}
/***********************************************************************
 * chkRegistro: Comprueba la disponibilidad del valor de un registro
 * Parámetros:
 *  - reg: Número de registro a comprobar
 *  - fp: Verdadero indica que es un FPR; si no, es un GPR
 *  - v: Referencia al campo Vj/Vk de la estación de reserva donde
 *       hace falta ese valor
 *  - q: Referencia al campo Qj/Qk de la estación de reserva donde
 *       hace falta ese valor
 ***********************************************************************/
void __fastcall TMaquinaSuper::chkRegistro(int reg, bool fp, float &v,
int &q) {
    q = -1;
    v = 0;
    if (fp) {
        // El registro tiene su valor listo
        if (!fpr.getBusy(reg))
            v = fpr.getReg(reg);
        // El registro no está listo pero el valor ya está en el ROB
        else if (ROB[ROBFpr[reg]]->listo)
            v = ROB[ROBFpr[reg]]->valor;
```

```
            // El valor aún está calculándose
            else
                q = ROBFpr[reg];
        }
        else {
            if (!gpr.getBusy(reg))
                v = gpr.getReg(reg);
            else if (ROB[ROBGpr[reg]]->listo)
                v = ROB[ROBGpr[reg]]->valor;
            else
                q = ROBGpr[reg];
        }
    }
    /**************************************************************************
     * emitirInstruccion: Resuelve la etapa de emisión para una
     * instrucción
     * Parámetros:
     *   - inst: Instrucción a emitir
     *   - tipo: Tipo de UF/ER
     *   - indER: Posición de la ER
     *   - indROB: Posición del ROB
     *************************************************************************/
    void __fastcall TMaquinaSuper::emitirInstruccion(TInstruccion *inst,
    int tipo, int indROB) {
        ER[tipo].back().instruccion = inst;
        ER[tipo].back().ROB = indROB;
        ER[tipo].back().numUF = -1;
        ER[tipo].back().A = -1;
        switch(inst->getOpcode()) {
            case TCodigo::ADDI:
            case TCodigo::MULTI:
                chkRegistro(inst->getOp(1), false, ER[tipo].back().Vj,
    ER[tipo].back().Qj);
                chkRegistro(inst->getOp(2), false, ER[tipo].back().Vk,
    ER[tipo].back().Qk);
                ROBGpr[inst->getOp(0)] = indROB;
                gpr.setBusy(inst->getOp(0), true);
                ROB[indROB]->destino = inst->getOp(0);
                break;
            case TCodigo::DADDUI:
                chkRegistro(inst->getOp(1), false, ER[tipo].back().Vj,
    ER[tipo].back().Qj);
                ER[tipo].back().Qk = -1;
                ER[tipo].back().Vk = inst->getOp(2);
                ROBGpr[inst->getOp(0)] = indROB;
                gpr.setBusy(inst->getOp(0), true);
                ROB[indROB]->destino = inst->getOp(0);
                break;
            case TCodigo::ADDF:
            case TCodigo::MULTF:
                chkRegistro(inst->getOp(1), true, ER[tipo].back().Vj,
    ER[tipo].back().Qj);
                chkRegistro(inst->getOp(2), true, ER[tipo].back().Vk,
    ER[tipo].back().Qk);
                ROBFpr[inst->getOp(0)] = indROB;
                fpr.setBusy(inst->getOp(0), true);
                ROB[indROB]->destino = inst->getOp(0);
                break;
            case TCodigo::SI:
                chkRegistro(inst->getOp(0), false, ER[tipo].back().Vj,
    ER[tipo].back().Qj);
```

```
            chkRegistro(inst->getOp(2), false, ER[tipo].back().Vk,
ER[tipo].back().Qk);
            ER[tipo].back().A = inst->getOp(1);
            ROB[indROB]->direccion = -1;
            break;
        case TCodigo::SF:
            chkRegistro(inst->getOp(0), true, ER[tipo].back().Vj,
ER[tipo].back().Qj);
            chkRegistro(inst->getOp(2), false, ER[tipo].back().Vk,
ER[tipo].back().Qk);
            ER[tipo].back().A = inst->getOp(1);
            ROB[indROB]->direccion = -1;
            break;
        case TCodigo::LI:
            chkRegistro(inst->getOp(2), false, ER[tipo].back().Vk,
ER[tipo].back().Qk);
            ER[tipo].back().Qj = -1;
            ER[tipo].back().Vj = 0;
            ER[tipo].back().A = inst->getOp(1);
            ROBGpr[inst->getOp(0)] = indROB;
            gpr.setBusy(inst->getOp(0), true);
            ROB[indROB]->destino = inst->getOp(0);
            ROB[indROB]->direccion = -1;
            break;
        case TCodigo::LF:
            chkRegistro(inst->getOp(2), false, ER[tipo].back().Vk,
ER[tipo].back().Qk);
            ER[tipo].back().Qj = -1;
            ER[tipo].back().Vj = 0;
            ER[tipo].back().A = inst->getOp(1);
            ROBFpr[inst->getOp(0)] = indROB;
            fpr.setBusy(inst->getOp(0), true);
            ROB[indROB]->destino = inst->getOp(0);
            ROB[indROB]->direccion = -1;
            break;
        case TCodigo::BEQ:
        case TCodigo::BNE:
            chkRegistro(inst->getOp(0), false, ER[tipo].back().Vj,
ER[tipo].back().Qj);
            chkRegistro(inst->getOp(1), false, ER[tipo].back().Vk,
ER[tipo].back().Qk);
            ER[tipo].back().A = inst->getOp(2);
            break;
        default:
            break;
    }
}
/*****************************************************************
 * ticIssue: Simula la etapa de "Issue" (emisión), con la que las
 * instrucciones pasan del decodificador a las estaciones de reserva y
 * el ROB.
 * Devuelve: El número de instrucciones que pudieron emitirse
 *****************************************************************/
int __fastcall TMaquinaSuper::ticIssue() {
    int cont = 0;
    TDecoder::iterator it = decoder.begin();
    for (; it != decoder.end(); it++, cont++) {
        TInstruccion *inst = it->instruccion;
        if (ROB.isFull())
            break;
        TTipoUF tipoUF = TCodigo::op2UF(inst->getOpcode());
```

```
            if (ER[tipoUF].size() == getTamER(tipoUF))
                break;
            TEntradaROB *nuevaROB = new TEntradaROB;
            nuevaROB->valor = 0.0;
            nuevaROB->destino = -1;
            nuevaROB->direccion = -1;
            int posROB = ROB.add(nuevaROB);
            TEntradaER nuevaER;
            ER[tipoUF].push_back(nuevaER);
            emitirInstruccion(inst, tipoUF, posROB);
            ROB[posROB]->instruccion = inst;
            ROB[posROB]->listo = false;
            ROB[posROB]->etapa = SUPER_ISSUE;
            TEntradaDecoder *auxDec = it.remove();
            delete auxDec;
        }
        return cont;
}

/*****************************************************************
 * chkStore: Comprueba que no haya ningún store anterior en el cauce
 * del ROB que no tenga calculada su dirección destino o tenga la
 * misma dirección que la pasada por param.
 * Parámetros:
 *  - indROB: Indice en el ROB de la instrucción con la que se compara
 *  - dir: Dirección con la que se compara
 *****************************************************************/
bool __fastcall TMaquinaSuper::chkStore(int indROB, int dir) {
    // Compruebo que no haya algún store anterior...
    TReorderBuffer::iterator itROB = ROB.begin();
    for (; itROB != ROB.position(indROB); itROB++) {
        int opcode = itROB->instruccion->getOpcode();
        if ((opcode == TCodigo::SI) || (opcode == TCodigo::SF)) {
            //... sin la dir. calculada...
            if (itROB->direccion == -1)
                break;
            // ...o con la misma dir.
            else if (itROB->direccion == dir)
                break;
        }
    }
    return (itROB == ROB.position(indROB));
}
/*****************************************************************
 * ejecutarInstruccion: Comprueba si una instrucción está lista para
 * ser ejecutada
 * Parámetros:
 *  - tipo: Tipo de UF
 *  - num: Número de UF de ese tipo libre
 * Comentarios: Lo que se hace es comprobar si los operandos están
 * disponibles.
 *****************************************************************/
void __fastcall TMaquinaSuper::ejecutarInstruccion(TTipoUF tipo, int
num) {
    TEstacionReserva::iterator it = ER[tipo].begin();

    switch(tipo) {
        case SUMAENT:
        case MULTENT:
        case SUMAFLOT:
        case MULTFLOT:
```

```
        case SALTO:
            // Operandos disponibles
            while (it != ER[tipo].end() && !((it->Qj == -1) && (it->Qk
== -1) && (it->numUF == -1)))
                ++it;
            if (it != ER[tipo].end()) {
                it->numUF = num;
                it->posUF = UF[tipo][num].rellenarCauce(it-
>instruccion);
                ROB[it->ROB]->etapa = SUPER_EXECUTE;
            }
            break;
        case MEM:
            // Fase 2 (Sólo los LOAD): Poner a ejecutar
            for (; it != ER[tipo].end(); it++) {
                int opcode = it->instruccion->getOpcode();
                if (((opcode == TCodigo::LI) || (opcode ==
TCodigo::LF)) // Es un LOAD
                    && ((it->numUF == -1)    // No se está ejecutando
                    && (ROB[it->ROB]->direccion != -1))    // Dir. ya
calculada
                    && chkStore(it->ROB, ROB[it->ROB]->direccion))  //
Comprobación de orden
                    break;
            }
            if (it != ER[tipo].end()) {
                it->numUF = num;
                it->posUF = UF[tipo][num].rellenarCauce(it-
>instruccion);
            }
            break;
        default:
            break;
    }
}
/*********************************************************************
 * ticExecute: Solamente marca aquellas instrucciones que están listas
 * para ser ejecutadas
 * Comentarios: En esta parte del código realmente NO se ejecuta nada
 *********************************************************************/
int __fastcall TMaquinaSuper::ticExecute() {

    for (int i = 0; i < NTIPOSUF; i++)
        for (int j = 0; j < nUF[i]; j++)
            if (UF[i][j].estaLibre())
                ejecutarInstruccion(i, j);

    /* Después de pasar por todas las UF me encargo de las UF de
       cálculo de dir.
       Fase 1b: Cálculo de la dirección
       Primero termino la ejecución del cálculo de dir. en la ALU */
    for (int i = 0; i < nUF[MEM]; i++) {
        if (aluMem[i].getTopInstruccion() != NULL) {
            // Busco la entrada de la ER que coincide con esa
instrucción
            TEstacionReserva::iterator it = ER[MEM].begin();
            while ((it->numUF !=  nUF[MEM] + i) || (it->posUF !=
aluMem[i].getUltima()))
                it++;
            ROB[it->ROB]->direccion = it->Vk + it->A;
            it->A = ROB[it->ROB]->direccion;
```

```
            it->numUF = -1; // Vuelve a no tener una UF asociada
        }
        aluMem[i].tic();
    }
    // Fase 1a: Cálculo de la dirección
    // Relleno la ALU de cálculo de direcciones asociada a esta UF
    for (int i = 0; i < nUF[MEM]; i++) {
        TEstacionReserva::iterator it = ER[MEM].begin();
        for (; it != ER[MEM].end(); it++)
            if ((it->Qk == -1) // Valor del operando disponible
                && (ROB[it->ROB]->direccion == -1)  // Dirección
todavía no calculada...
                && (it->numUF == -1))    // ...y no está calculándose
ahora mismo
                break;
        if (it != ER[MEM].end()) {
            it->numUF = i + nUF[MEM];  // Así las distingo de las UF
de Memoria
            it->posUF = aluMem[i].rellenarCauce(it->instruccion);
            ROB[it->ROB]->etapa = SUPER_EXECUTE;
        }
    }
}

/**********************************************************************
 * escribirInstruccion: Resuelve la etapa de escritura de resultados
 * de una instrucción.
 * Parámetros:
 *  - tipo: Tipo de UF
 *  - num: Número de UF de ese tipo
 * Comentarios: Lo único que se hace es recoger el resultado de la
 * instrucción (si hay una instrucción lista en esa UF) y difundirlo
 * entre todas las ER que lo requieran como resultado.
 **********************************************************************/
void __fastcall TMaquinaSuper::escribirInstruccion(TTipoUF tipo, int
num) {
    float resul;
    TInstruccion *inst = UF[tipo][num].getTopInstruccion();
    if (inst != NULL) {
        TEstacionReserva::iterator it = ER[tipo].begin();
        while ((it->numUF != num) || (it->posUF !=
UF[tipo][num].getUltima()))
            it++;
        int opcode = inst->getOpcode();
        switch(opcode) {
            case TCodigo::ADDI:
            case TCodigo::DADDUI:
            case TCodigo::ADDF:
                resul = it->Vj + it->Vk;
                break;
            case TCodigo::MULTI:
            case TCodigo::MULTF:
                resul = it->Vj * it->Vk;
                break;
            // En esta fase no se hace nada con los STORES
            case TCodigo::LI:
            case TCodigo::LF:
                if (!memoria.getDato(it->A, resul))
                    UF[tipo][num].setStall(latFalloMem -
UF[tipo][num].getLatencia());
                break;
```

```
            case TCodigo::BEQ:
                resul = (it->Vj == it->Vk) ? 1 : 0;
                break;
            case TCodigo::BNE:
                resul = (it->Vj != it->Vk) ? 1 : 0;
                break;
        }

        // Finalizó la ejecución de la instrucción
        if (UF[tipo][num].getStall() == 0) {
            if ((opcode != TCodigo::BNE) && (opcode != TCodigo::BEQ))
{
                // Actualizo todas las ER
                for (int i = 0; i < NTIPOSUF; i++) {
                    TEstacionReserva::iterator itER = ER[i].begin();
                    for(; itER != ER[i].end(); itER++) {
                        if (itER->Qj == it->ROB) {
                            itER->Vj = resul;
                            itER->Qj = -1;
                        }
                        if (itER->Qk == it->ROB) {
                            itER->Vk = resul;
                            itER->Qk = -1;
                        }
                    }
                }
            }
            ROB[it->ROB]->valor = resul;
            ROB[it->ROB]->etapa = SUPER_WRITERESULT;
            ROB[it->ROB]->listo = true;
            // Elimino la entrada de la ER
            ER[tipo].erase(it);
        }
    }
}

/*********************************************************************
 * ticWriteResult: Coge las instrucciones marcadas como ejecutadas y
 * actualiza su resultado
 * Comentarios: En esta parte del código es donde se efectúa realmente
 * la ejecución
 *********************************************************************/
int __fastcall TMaquinaSuper::ticWriteResult() {
    // En primer lugar compruebo si hay STORES listos
    TEstacionReserva::iterator it = ER[MEM].begin();
    while (it != ER[MEM].end()) {
        int opcode = ROB[it->ROB]->instruccion->getOpcode();
        if (((opcode == TCodigo::SI) || (opcode == TCodigo::SF)) &&
(it->Qj == -1) && (ROB[it->ROB]->direccion != -1)) {
            ROB[it->ROB]->valor = it->Vj;
            ROB[it->ROB]->etapa = SUPER_WRITERESULT;
            ROB[it->ROB]->listo = true;
            // Elimino la entrada de la ER
            if (it == ER[MEM].begin()) {
                ER[MEM].erase(it);
                it = ER[MEM].begin();
            }
            else {
                TEstacionReserva::iterator itAux = it;
                it--;
                ER[MEM].erase(itAux);
```

```
                it++;
            }
        }
        else
            it++;
    }

    // Después recorro todas las UF para recoger los resultados
    for (int i = 0; i < NTIPOSUF; i++)
        for (int j = 0; j < nUF[i]; j++) {
            if (UF[i][j].getStall() == 0)
                escribirInstruccion(i, j);
            // Avanzo el reloj de esa UF
            UF[i][j].tic();
    }
}

/**************************************************************************
 * chkSalto: Comprueba si la especulación en la ejecución realizada a
 * partir de una instrucción de salto fue correcta o no en el momento
 * de graduarla.
 * Parámetros:
 *  - rob: Entrada del ROB que contiene la instrucción de salto
 * Comentarios: En caso de ser una predicción correcta se deja todo
 * como está.
 * En caso contrario, hay que vaciar todo el flujo de ejecución.
 * En cualquiera de los casos hay que actualizar el valor de la tabla
 * de predicción de saltos.
 **************************************************************************/
bool __fastcall TMaquinaSuper::chkSalto(TEntradaROB *rob) {
    // Se comprueba si la predicción acertó
    if (prediccion(rob->instruccion->getId()) ^ (bool) rob->valor) {
        cambiarPrediccion(rob->instruccion->getId(), (bool) rob-
>valor);
        // Se cambia el PC
        if ((bool) rob->valor)
            PC = codigo->getInstruccionBB(rob->instruccion->getOp(2));
        else
            PC = rob->instruccion->getId() + 1;
        // Se limpia el ROB
        TReorderBuffer::iterator itROB = ROB.begin();
        for (; itROB != ROB.end(); itROB++) {
            TEntradaROB *aux = itROB.remove();
            delete aux->instruccion;
            delete aux;
        }
        // Y libero la mem. de la entrada del ROB q contenía el salto
        delete rob->instruccion;
        delete rob;
        // Se limpian las UF y ER
        for (int i = 0; i < NTIPOSUF; i++)
            for (int j = 0; j < nUF[i]; j++) {
                UF[i][j].limpiar();
                ER[i].clear();
            }
        // y las UF de cálculo de direcciones
        for (int i = 0; i < nUF[MEM]; i++)
            aluMem[i].limpiar();
        // Se limpia el decoder
        TDecoder::iterator itDec = decoder.begin();
        for (; itDec != decoder.end(); itDec++) {
```

```
            TEntradaDecoder *aux = itDec.remove();
            delete aux->instruccion;
            delete aux;
        }
        // Se limpia la unidad de Prefetch
        TPrefetchUnit::iterator itPre = prefetchUnit.begin();
        for (; itPre != prefetchUnit.end(); itPre++) {
            TEntradaPrefetch *aux = itPre.remove();
            delete aux->instruccion;
            delete aux;
        }
        // Limpio también las estructuras asociadas a los registros
        memset(ROBGpr, -1, sizeof(int) * NGP);
        memset(ROBFpr, -1, sizeof(int) * NFP);
        gpr.setBusy(false);
        fpr.setBusy(false);
        return false;
    }
    cambiarPrediccion(rob->instruccion->getId(), (bool) rob->valor);
    return true;
}
/*****************************************************************
 * ticCommit: Simula la etapa de "Commit" (graduación)
 * Devuelve:
 *  - SUPER_COMMITOK: Se pudieron graduar las "emision" instrucciones
 *  - SUPER_COMMITNO: Faltaron algunas instruciones por graduar porque
 *    aún no estaban listas
 *  - SUPER_COMMITEND: Se vació el ROB
 *  - SUPER_COMMITMISS: Se vacío el ROB a causa de un fallo de
 *    predicción de salto
 * Comentarios: La graduación se realiza de manera secuencial, para
 * asegurar el orden del programa.
 *****************************************************************/
TCommitStatus __fastcall TMaquinaSuper::ticCommit() {
    for (int i = 0; i < emision; i++) {
        if (ROB.isEmpty())
            return SUPER_COMMITEND;
        else if (!ROB.top()->listo)
            return SUPER_COMMITNO;
        else {
            int h = ROB.getFirst();
            TEntradaROB *aux = ROB.remove();
            switch (aux->instruccion->getOpcode()) {
                case TCodigo::SI:
                case TCodigo::SF:
                    memoria.setDato(aux->direccion, aux->valor);
                    break;
                case TCodigo::BEQ:
                case TCodigo::BNE:
                    if (!chkSalto(aux))
                        return SUPER_COMMITMISS;
                    break;
                case TCodigo::ADDI:
                case TCodigo::DADDUI:
                case TCodigo::MULTI:
                case TCodigo::LI:
                    gpr.setReg(aux->destino, aux->valor, false);
                    // Pase lo que pase R0 vale 0
                    gpr.setReg(0, 0, false);
                    if (ROBGpr[aux->destino] == h)
                        gpr.setBusy(aux->destino, false);
```

```
                    break;
                case TCodigo::ADDF:
                case TCodigo::MULTF:
                case TCodigo::LF:
                    fpr.setReg(aux->destino, aux->valor, false);
                    if (ROBFpr[aux->destino] == h)
                        fpr.setBusy(aux->destino, false);
                    break;
                default:
                    break;
            }
            // Importante: Elimino la instrucción copia de la original
            delete aux->instruccion;
            delete aux;
        }
    }
    return SUPER_COMMITOK;
}
/**********************************************************************
 * tic: Avanza un ciclo el reloj de la máquina y realiza la simulación
 * Devuelve:
 *  - SUPER_ENDEXE: Finalizó la ejecución del programa.
 *  - SUPER_BREAKPOINT: La ejecución se encontró con un breakpoint
 *  - SUPER_OK: Este ciclo de reloj finalizo sin problemas
 **********************************************************************/
TSuperscalarStatus __fastcall TMaquinaSuper::tic() {
    status.ciclo++;
    // ETAPA DE COMMIT
    TCommitStatus comm = ticCommit();
    if ((comm != SUPER_COMMITEND) && (comm != SUPER_COMMITMISS)) {
        // ETAPA DE WRITE RESULT
        ticWriteResult();
        // ETAPA DE EXECUTE
        ticExecute();
    }
    // ETAPA DE ISSUE
    int resulIssue = ticIssue();
    // ETAPA DE DECODER
    int resulDecoder = ticDecoder();
    // ETAPA DE PREFETCH
    int resulPrefetch = ticPrefetch();
    if ((resulIssue + resulDecoder + resulPrefetch == 0) && (comm ==
SUPER_COMMITEND))
        return SUPER_ENDEXE;
    TPrefetchUnit::iterator it = prefetchUnit.begin();
    for (; it != prefetchUnit.end(); it++)
        if (it->instruccion->getBreakPoint()) {
            status.breakPoint = true;
            return SUPER_BREAKPOINT;
        }
    return SUPER_OK;
}
```

A.12. Clase TMaquinaVLIW

A.12.1. Fichero TMaquinaVLIW.h

```
#ifndef TMaquinaVLIWH
#define TMaquinaVLIWH
/**********************************************************************
 * CLASE: TMaquinaVLIW
```

```
 *
 * EXTIENDE A: TMaquina
 *
 * FICHERO: TMaquinaVLIW.h, TMaquinaVLIW.cpp
 *
 * AUTOR: Iván Castilla Rodríguez
 *
 * FECHA: 25/03/04
 *
 * DESCRIPCION: Estructura de la máquina VLIW que se empleará en las
 * simulaciones
 *
 * COMENTARIOS:
 * - Se compone de:
 *   - Banco de 64 registros de predicado
 *   - Conjunto de bits NaT asociados a los registros de própósito
 *     general y a los de punto flotante.
 * - Se le asocia un código de instrucciones largas, que a su vez está
 *   asociado a un código secuencial.
 * - La máquina sólo admite una única unidad de salto.
 * - El registro de predicado "0" siempre vale true.
 ******************************************************************/

#include "TMaquina.h"
#include "TCodigo.h"
#include "TCodigoVLIW.h"
#include "TInstruccionLarga.h"

/* TChequeo: Estructura para el chequeo de errores del código VLIW.
Permite almacenar las latencias del banco de registros */
typedef struct {
    int lat;     // Latencia máxima de ese registro
    int reg;     // Número del registro
} TChequeo;

// Definición de valores de error
typedef enum {VLIW_PCOUTOFRANGE = -3, VLIW_ENDEXE = -2,
VLIW_BREAKPOINT = -1, VLIW_OK = 0} TVLIWStatus;
typedef enum {VLIW_ERRRAW = -4, VLIW_ERRHARD = -3, VLIW_ERRBRANCHDEP =
-2, VLIW_ERRPRED = -1, VLIW_ERRNO = 0} TVLIWError;

class TMaquinaVLIW : public TMaquina {
public:
    static const int NPR = 64;  // Nº de Registros de Predicado
private:
    bool predR[NPR];            // Registros de Predicado
    bool NaTGP[NGP];            // Registros NaT para Prop. General
    bool NaTFP[NFP];            // Registros NaT para Punto flotante
    TCodigoVLIW *codigo;        // Código de instrucciones largas

    bool __fastcall chkNaT(TOperacionVLIW *oper);
    void __fastcall ejecutarOperacion(TOperacionVLIW *oper,
TUnidadFuncional &uf);
    int __fastcall ejecutarSalto(TOperacionVLIW *oper);
    void __fastcall chkDestinoOp(TOperacionVLIW *oper, TChequeo
*chkGPR, TChequeo *chkFPR);
    bool __fastcall chkFuenteOp(TOperacionVLIW *oper, TChequeo
*chkGPR, TChequeo *chkFPR);
    TVLIWError __fastcall chkDependencias(int &fila, int &id);
    TVLIWError __fastcall chkPredicados(int &fila, int &id);
```

```
public:
    // Constructores
    __fastcall TMaquinaVLIW();

    // Destructores
    __fastcall ~TMaquinaVLIW() {
        delete codigo;
    }

    // GETTERS
    bool __fastcall getPredReg(int ind) { return predR[ind]; }
    bool __fastcall getNaTGP(int ind) { return NaTGP[ind]; }
    bool __fastcall getNaTFP(int ind) { return NaTFP[ind]; }
    bool *__fastcall getPredReg() { return predR; }
    bool *__fastcall getNaTGP() { return NaTGP; }
    bool *__fastcall getNaTFP() { return NaTFP; }
    TCodigoVLIW * __fastcall getCodigo() { return codigo; }

    // SETTERS
    void __fastcall setPredReg(int ind, bool p) { predR[ind] = p; }
    void __fastcall setNaTGP(int ind, bool n) { NaTGP[ind] = n; }
    void __fastcall setNaTFP(int ind, bool n) { NaTFP[ind] = n; }
    /* Sobreescribo el método de TMaquina para asegurar que sólo haya
una unidad de salto */
    void __fastcall setNUF(int ind, int n) {
        nUF[ind] = ((TTipoUF) ind == SALTO) ? 1 : n;
    }

    TVLIWError __fastcall chkCodigo();
    TVLIWError __fastcall chkErrores(int &fila, int &id);
    TVLIWStatus __fastcall tic();
    void __fastcall init(bool reset);
};
#endif
```

A.12.2. Fichero TMaquinaVLIW.cpp

```
#include <vcl.h>
#pragma hdrstop

#include "TMaquinaVLIW.h"
#include <list.h>

//---------------------------------------------------------------------------
#pragma package(smart_init)

__fastcall TMaquinaVLIW::TMaquinaVLIW() :TMaquina() {
    register int i;
    codigo = new TCodigoVLIW();
    memset(predR, false, sizeof(bool) * NPR);
    predR[0] = true;
    memset(NaTGP, false, sizeof(bool) * NGP);
    memset(NaTFP, false, sizeof(bool) * NFP);
}

/*****************************************************************
 * chkNaT: Chequea si algún operando fuente de la operación es un NaT
 * Devuelve:
 *  - false: Ningún operando es un NaT
 *  - true: Algún operando es un NaT y se produce una excepción
 *****************************************************************/
```

```
bool __fastcall TMaquinaVLIW::chkNaT(TOperacionVLIW *oper) {
    bool resul;
    switch(oper->getOpcode()) {
        case TCodigo::ADDI:
        case TCodigo::MULTI:
            resul = NaTGP[oper->getOp(1)] || NaTGP[oper->getOp(2)];
            break;
        case TCodigo::DADDUI:
            resul = NaTGP[oper->getOp(1)];
            break;
        case TCodigo::ADDF:
        case TCodigo::MULTF:
            resul = NaTFP[oper->getOp(1)] || NaTFP[oper->getOp(2)];
            break;
        case TCodigo::SI:
            resul =  NaTGP[oper->getOp(0)] || NaTGP[oper->getOp(2)];
            break;
        case TCodigo::SF:
            resul =  NaTFP[oper->getOp(0)] || NaTGP[oper->getOp(2)];
            break;
        case TCodigo::LI:
        case TCodigo::LF:
            resul = NaTGP[oper->getOp(2)];
            break;
        case TCodigo::BEQ:
        case TCodigo::BNE:
            resul = NaTGP[oper->getOp(0)] || NaTGP[oper->getOp(1)];
            break;
        default:
            resul = true;
            break;
    }
    return resul;
}

/*****************************************************************
 * ejecutar: Ejecuta una operación en la máquina (excepto los saltos)
 * Parámetros:
 *  - oper: Operación a ejecutar
 *  - uf: Unidad funcional de donde se obtiene la operación
 *****************************************************************/
void __fastcall TMaquinaVLIW::ejecutarOperacion(TOperacionVLIW *oper,
TUnidadFuncional &uf) {
    switch(oper->getOpcode()) {
        case TCodigo::ADDI:
            gpr.setReg(oper->getOp(0), gpr.getReg(oper->getOp(1)) +
gpr.getReg(oper->getOp(2)), true);
            break;
        case TCodigo::MULTI:
            gpr.setReg(oper->getOp(0), gpr.getReg(oper->getOp(1)) *
gpr.getReg(oper->getOp(2)), true);
            break;
        case TCodigo::DADDUI:
            gpr.setReg(oper->getOp(0), gpr.getReg(oper->getOp(1)) +
oper->getOp(2), true);
            break;
        case TCodigo::ADDF:
            fpr.setReg(oper->getOp(0), fpr.getReg(oper->getOp(1)) +
fpr.getReg(oper->getOp(2)), true);
            break;
        case TCodigo::MULTF:
```

```
            fpr.setReg(oper->getOp(0), fpr.getReg(oper->getOp(1)) *
fpr.getReg(oper->getOp(2)), true);
            break;
        case TCodigo::SI:
            memoria.setDato(gpr.getReg(oper->getOp(2)) + oper-
>getOp(1), gpr.getReg(oper->getOp(0)));
            break;
        case TCodigo::SF:
            memoria.setDato(gpr.getReg(oper->getOp(2)) + oper-
>getOp(1), fpr.getReg(oper->getOp(0)));
            break;
        case TCodigo::LI:
            int datoI;
            if (!memoria.getDato(gpr.getReg(oper->getOp(2)) + oper-
>getOp(1), datoI))
                uf.setStall(latFalloMem - uf.getLatencia());
            else {
                gpr.setReg(oper->getOp(0), datoI, true);
                NaTGP[oper->getOp(0)] = false;
            }
            break;
        case TCodigo::LF:
            float datoF;
            if (!memoria.getDato(gpr.getReg(oper->getOp(2)) + oper-
>getOp(1), datoF))
                uf.setStall(latFalloMem - uf.getLatencia());
            else {
                fpr.setReg(oper->getOp(0), datoF, true);
                NaTFP[oper->getOp(0)] = false;
            }
            break;
        default:
            break;
    }
    // Pase lo q pase R0 = 0 y PRED[0] = true
    gpr.setReg(0, 0, true);
    predR[0] = true;
}

/*****************************************************************
 * ejecutarSalto: Ejecuta una operación de salto, modificando si es
 * preciso el PC.
 * Parámetros:
 *  - oper: Operación a ejecutar
 * Devuelve:
 *  - El nuevo PC de la máquina cambiado si el salto se toma.
 *****************************************************************/
int __fastcall TMaquinaVLIW::ejecutarSalto(TOperacionVLIW *oper) {
    int nuevoPC = PC;
    if (oper->getOpcode() == TCodigo::BEQ) {
        if (gpr.getReg(oper->getOp(0)) == gpr.getReg(oper->getOp(1)))
{
            nuevoPC = oper->getOp(2);
            predR[oper->getPredTrue()] = true;
            predR[oper->getPredFalse()] = false;
        }
        else {
            predR[oper->getPredTrue()] = false;
            predR[oper->getPredFalse()] = true;
        }
    }
```

```
    else if (oper->getOpcode() == TCodigo::BNE) {
        if (gpr.getReg(oper->getOp(0)) != gpr.getReg(oper->getOp(1)))
{
            nuevoPC = oper->getOp(2);
            predR[oper->getPredTrue()] = true;
            predR[oper->getPredFalse()] = false;
        }
        else {
            predR[oper->getPredTrue()] = false;
            predR[oper->getPredFalse()] = true;
        }
    }
    return nuevoPC;
}

/*****************************************************************
 * tic: Avanza un ciclo el reloj de la máquina y realiza la simulación
 * Devuelve:
 *  - El nuevo PC de la máquina (puede ser el mismo si se detiene la
 *    ejecución por alguna burbuja)
 *  - VLIW_PCOUTOFRANGE si el PC no corresponde con una instrucción
 *    válida
 *  - VLIW_ENDEXE si no hay más instrucciones por leer ni ejecutar
 * Comentarios:
 * El "stall" de las UF tiene una incoherencia cuando se usa con un
 * mecanismo de fallos de caché aleatorios. Si ese es el caso, podría
 * ser que tras el tiempo de espera por el fallo ocurriese nuevamente
 * otro fallo (ABSURDO!!)
 *****************************************************************/
TVLIWStatus __fastcall TMaquinaVLIW::tic() {
    int i, j;
    bool pendiente = false;
    bool detenerFlujo = false;

    status.ciclo++;
    // Se intentan ejecutar todas las operaciones del "top" de cada UF
    // IMPORTANTE:
    // Primero se ejecutan las de la UF de salto
    if (UF[SALTO][0].instruccionesPendientes())
        pendiente = true;
    if (UF[SALTO][0].getStall() == 0) {
        TOperacionVLIW *oper = (TOperacionVLIW
*)UF[SALTO][0].getTopInstruccion();
        if (oper != NULL) {
            if (predR[oper->getPred()])
                PC = ejecutarSalto(oper);
        }
    }
    // Avanzo el reloj de esa UF
    UF[SALTO][0].tic();

    // Hago hasta NTIPOSUF -1 pq sé que el último tipo es el SALTO
    for (i = 0; i < NTIPOSUF - 1; i++)
        for (j = 0; j < nUF[i]; j++) {
            if (UF[i][j].instruccionesPendientes())
                pendiente = true;
            if (UF[i][j].getStall() == 0) {
                TOperacionVLIW *oper = (TOperacionVLIW
*)UF[i][j].getTopInstruccion();
                if (oper != NULL) {
                    if (predR[oper->getPred()])
```

```
                        ejecutarOperacion(oper, UF[i][j]);
                    }
                }
                // Avanzo el reloj de esa UF
                UF[i][j].tic();
            }

        // Hago las escrituras en los registros para evitar dep. WAR
        gpr.tic();
        fpr.tic();

        if (!detenerFlujo) {
            // Leo la siguiente instrucción emitida
            TInstruccionLarga *inst = codigo->getInstruccionLarga(PC);
            if (inst == NULL) {
                if (pendiente)
                    return VLIW_PCOUTOFRANGE;
                // No hay más instrucciones por leer ni ejecutar
                return VLIW_ENDEXE;
            }
            /* Primero compruebo si hay alguna UF ocupada que debería
               estar libre o algún operando marcado como NaT */
            for (i = 0; i < inst->getNOper(); i++) {
                TTipoUF tipo = inst->getOperacion(i)->getTipoUF();
                int num = inst->getOperacion(i)->getNumUF();
                if (!UF[tipo][num].estaLibre() || chkNaT(inst-
>getOperacion(i))) {
                    detenerFlujo = true;
                    break;
                }
            }
            // Si todo va bien pongo la siguiente instrucción larga
            if (!detenerFlujo) {
                for (i = 0; i < inst->getNOper(); i++) {
                    TOperacionVLIW *oper = inst->getOperacion(i);
                    UF[oper->getTipoUF()][oper-
>getNumUF()].rellenarCauce(oper);
                    // Si son LOADS marco los registros destino como NaT
                    if (oper->getOpcode() == TCodigo::LI)
                        NaTGP[oper->getOp(0)] = true;
                    if (oper->getOpcode() == TCodigo::LF)
                        NaTFP[oper->getOp(0)] = true;
                    /* Si es un salto reseteo los registros de Predicado
                      que emplea. Con esto evito que se ejecuten
                      operaciones de un salto PREDICADO */
                    if (oper->getTipoUF() == SALTO) {
                        predR[oper->getPredTrue()] = false;
                        predR[oper->getPredFalse()] = false;
                    }
                }
                // Actualizo el PC
                PC++;
            }
            if (inst->getBreakPoint()) {
                status.breakPoint = true;
                return VLIW_BREAKPOINT;
            }
        }
        return VLIW_OK;
}
```

```
/*****************************************************************
 * init: Inicializa los componentes de la máquina
 * Parámetros:
 * - reset: Si es "true" implica que además hay que resetear el
 *   contenido de los registros y memoria
 *****************************************************************/
void __fastcall TMaquinaVLIW::init(bool reset) {
    TMaquina::init(reset);
    memset(NaTGP, false, sizeof(bool) * NGP);
    memset(NaTFP, false, sizeof(bool) * NFP);
    memset(predR, false, sizeof(bool) * NPR);
    predR[0] = true;
}

/*****************************************************************
 * chkDestinoOp: Marca el tiempo que queda para que el registro
 * destino de una operación esté disponible. Siempre se queda con el
 * mayor tiempo
 *****************************************************************/
void __fastcall TMaquinaVLIW::chkDestinoOp(TOperacionVLIW *oper,
TChequeo *chkGPR, TChequeo *chkFPR) {
    switch (oper->getOpcode()) {
        case TCodigo::ADDI:
        case TCodigo::DADDUI:
            if (chkGPR[oper->getOp(0)].lat < latenciaUF[SUMAENT]) {
                chkGPR[oper->getOp(0)].lat = latenciaUF[SUMAENT];
                chkGPR[oper->getOp(0)].reg = oper->getId();
            }
            break;
        case TCodigo::MULTI:
            if (chkGPR[oper->getOp(0)].lat < latenciaUF[MULTENT]) {
                chkGPR[oper->getOp(0)].lat = latenciaUF[MULTENT];
                chkGPR[oper->getOp(0)].reg = oper->getId();
            }
            break;
        case TCodigo::ADDF:
            if (chkFPR[oper->getOp(0)].lat < latenciaUF[SUMAFLOT]) {
                chkFPR[oper->getOp(0)].lat = latenciaUF[SUMAFLOT];
                chkFPR[oper->getOp(0)].reg = oper->getId();
            }
            break;
        case TCodigo::MULTF:
            if (chkFPR[oper->getOp(0)].lat < latenciaUF[MULTFLOT]) {
                chkFPR[oper->getOp(0)].lat = latenciaUF[MULTFLOT];
                chkFPR[oper->getOp(0)].reg = oper->getId();
            }
            break;
        case TCodigo::LI:
            if (chkGPR[oper->getOp(0)].lat < latenciaUF[MEM]) {
                chkGPR[oper->getOp(0)].lat = latenciaUF[MEM];
                chkGPR[oper->getOp(0)].reg = oper->getId();
            }
            break;
        case TCodigo::LF:
            if (chkFPR[oper->getOp(0)].lat < latenciaUF[MEM]) {
                chkFPR[oper->getOp(0)].lat = latenciaUF[MEM];
                chkFPR[oper->getOp(0)].reg = oper->getId();
            }
            break;
        case TCodigo::SI:
        case TCodigo::SF:
```

```
        case TCodigo::BEQ:
        case TCodigo::BNE:
        default:
            break;
    }
}
/*********************************************************************
 * chkFuenteOp: Comprueba si los registros operando de una operación
 * están disponibles o no.
 * Devuelve:
 * - true: No hay dependencias
 * - false: Existe una dependencia con alguna operación anterior
 *********************************************************************/
bool __fastcall TMaquinaVLIW::chkFuenteOp(TOperacionVLIW *oper,
TChequeo *chkGPR, TChequeo *chkFPR) {
    bool resul = true;
    switch (oper->getOpcode()) {
        case TCodigo::ADDI:
        case TCodigo::MULTI:
            if (((chkGPR[oper->getOp(1)].lat > 0) && (chkGPR[oper-
>getOp(1)].reg < oper->getId()))
            || ((chkGPR[oper->getOp(2)].lat > 0) && (chkGPR[oper-
>getOp(2)].reg < oper->getId())))
                resul = false;
            break;
        case TCodigo::DADDUI:
            if ((chkGPR[oper->getOp(1)].lat > 0) && (chkGPR[oper-
>getOp(1)].reg < oper->getId()))
                resul = false;
            break;
        case TCodigo::ADDF:
        case TCodigo::MULTF:
            if (((chkFPR[oper->getOp(1)].lat > 0) && (chkFPR[oper-
>getOp(1)].reg < oper->getId()))
            || ((chkFPR[oper->getOp(2)].lat > 0) && (chkFPR[oper-
>getOp(2)].reg < oper->getId())))
                resul = false;
            break;
        case TCodigo::LI:
        case TCodigo::LF:
            if ((chkGPR[oper->getOp(2)].lat > 0) && (chkGPR[oper-
>getOp(2)].reg < oper->getId()))
                resul = false;
            break;
        case TCodigo::SI:
            if (((chkGPR[oper->getOp(0)].lat > 0) && (chkGPR[oper-
>getOp(0)].reg < oper->getId()))
            || ((chkGPR[oper->getOp(2)].lat > 0) && (chkGPR[oper-
>getOp(2)].reg < oper->getId())))
                resul = false;
            break;
        case TCodigo::SF:
            if (((chkFPR[oper->getOp(0)].lat > 0) && (chkFPR[oper-
>getOp(0)].reg < oper->getId()))
            || ((chkGPR[oper->getOp(2)].lat > 0) && (chkGPR[oper-
>getOp(2)].reg < oper->getId())))
                resul = false;
            break;
        case TCodigo::BEQ:
        case TCodigo::BNE:
```

```
            if (((chkGPR[oper->getOp(0)].lat > 0) && (chkGPR[oper-
>getOp(0)].reg < oper->getId()))
            || ((chkGPR[oper->getOp(1)].lat > 0) && (chkGPR[oper-
>getOp(1)].reg < oper->getId())))
                resul = false;
            break;
        default:
            resul = true;
            break;
    }
    return resul;
}

/*********************************************************************
 * chkDependencias: Comprueba que el código VLIW no contiene
 * dependencias verdaderas que impidan su correcta ejecución
 * Devuelve:
 * - VLIW_ERRNO: El código es correcto
 * - VLIW_ERRRAW: el código contiene alguna dependencia verdadera en
 *   la operación "id" de la instrucción "fila"
 * NOTA: detecta si la dependencia es con una oper. anterior en el
 * flujo de ejecución, pero esto aún no está demostrado q sea muy
 * eficaz
 *********************************************************************/
TVLIWError __fastcall TMaquinaVLIW::chkDependencias(int &fila, int
&id) {
    TChequeo chkGPR[NGP];
    TChequeo chkFPR[NFP];

    // Inicialización de los arrays de control
    for (int i = 0; i < NGP; i++)
        chkGPR[i].lat = 0;
    for (int i = 0; i < NFP; i++)
        chkFPR[i].lat = 0;
    // Comprobación de dependencias verdaderas
    for (fila = 0; fila < codigo->getNInst(); fila++) {
        TInstruccionLarga *inst = codigo->getInstruccionLarga(fila);
        // Primero compruebo los registros destino de operaciones
        for (int j = 0; j < inst->getNOper(); j++)
            chkDestinoOp(inst->getOperacion(j), chkGPR, chkFPR);
        // Compruebo los registros fuente de operaciones
        for (int j = 0; j < inst->getNOper(); j++)
            if (!chkFuenteOp(inst->getOperacion(j), chkGPR, chkFPR)) {
                id = inst->getOperacion(j)->getId();
                return VLIW_ERRRAW;
            }
        // Decremento los contadores de los registros
        for (int i = 0; i < NGP; i++)
            if (chkGPR[i].lat > 0)
                chkGPR[i].lat--;
        for (int i = 0; i < NFP; i++)
            if (chkFPR[i].lat > 0)
                chkFPR[i].lat--;
    }
    return VLIW_ERRNO;
}

/*********************************************************************
 * chkCodigo: Comprueba que el código VLIW cumple los mínimos
 * requisitos para funcionar correctamente
 * Devuelve:
```

```
 * - VLIW_ERRNO: El código se puede ejecutar en esa máquina
 * - VLIW_ERRHARD: el código no se puede ejecutar en esa máquina
 * Comentarios: Se comprueban que la configuración de la máquina
 * permite ejecutar el código actual
 ***********************************************************************/
TVLIWError __fastcall TMaquinaVLIW::chkCodigo() {
    for (int i = 0; i < codigo->getNInst(); i++) {
        TInstruccionLarga *inst = codigo->getInstruccionLarga(i);
        for (int j = 0; j < inst->getNOper(); j++) {
            TOperacionVLIW *oper = inst->getOperacion(j);
            if (oper->getNumUF() >= nUF[oper->getTipoUF()])
                return VLIW_ERRHARD;
        }
    }
    return VLIW_ERRNO;
}

/***********************************************************************
 * chkPredicados: Comprueba si se está empleando un registro de
 * predicado donde no corresponde o si una operación predicada termina
 * su ejecución ANTES de haberse resuelto el salto del que depende.
 * Devuelve:
 * - VLIW_ERRNO: El código es correcto
 * - VLIW_ERRBRANCHDEP: La operación "id" de la instrucción "fila"
 *   termina su ejecución antes de que se resuelva el salto del que
 *   depende
 * - VLIW_ERRPRED: La operación "id" de la instrucción "fila" está
 *   predicada usando un registro de predicado que no ha sido
 *   declarado previamente en una operación de salto
 ***********************************************************************/
TVLIWError __fastcall TMaquinaVLIW::chkPredicados(int &fila, int &id)
{
    list<TChequeo> chkPred;
    // Comprobación de saltos con predicaciones incorrectas
    for (fila = 0; fila < codigo->getNInst(); fila++) {
        // Se controla la lista de control
        list<TChequeo>::iterator it = chkPred.begin();
        while (it != chkPred.end()) {
            if (it->lat == 1)
                it = chkPred.erase(it);
            else {
                it->lat--;
                it++;
            }
        }
        TInstruccionLarga *inst = codigo->getInstruccionLarga(fila);
        // Busco si hay alguna instrucción de salto
        for (int j = 0; j < inst->getNOper(); j++)
            if (inst->getOperacion(j)->getTipoUF() == SALTO) {
                TChequeo chk1;
                TChequeo chk2;
                chk1.lat = latenciaUF[SALTO];
                chk2.lat = latenciaUF[SALTO];
                chk1.reg = inst->getOperacion(j)->getPredTrue();
                chk2.reg = inst->getOperacion(j)->getPredFalse();
                chkPred.push_back(chk1);
                chkPred.push_back(chk2);
            }
        // Compruebo las dependencias
        for (int j = 0; j < inst->getNOper(); j++)
            if (inst->getOperacion(j)->getPred() != 0) {
```

```
                for (it = chkPred.begin(); it != chkPred.end(); it++)
                    if (inst->getOperacion(j)->getPred() == it->reg)
                        break;
                if (it == chkPred.end()) {
                    id = inst->getOperacion(j)->getId();
                    chkPred.clear();
                    return VLIW_ERRPRED;
                }
                else if (latenciaUF[inst->getOperacion(j)-
>getTipoUF()] < it->lat) {
                    id = inst->getOperacion(j)->getId();
                    chkPred.clear();
                    return VLIW_ERRBRANCHDEP;
                }
            }
    }
    chkPred.clear();
    return VLIW_ERRNO;
}

/*****************************************************************
 * chkErrores: Comprueba si hay dependencias (de datos o de control)
 * en el código VLIW.
 * Devuelve:
 * - VLIW_ERRNO: El código es correcto
 * - VLIW_ERRBRANCHDEP: La operación "id" de la instrucción "fila"
 *   termina su ejecución antes de que se resuelva el salto del que
 * depende
 * - VLIW_ERRPRED: La operación "id" de la instrucción "fila" está
 *   predicada usando un registro de predicado que no ha sido
 *   declarado previamente en una operación de salto
 * - VLIW_ERRRAW: el código contiene alguna dependencia verdadera en
 *   la operación "id" de la instrucción "fila"
 *****************************************************************/
TVLIWError __fastcall TMaquinaVLIW::chkErrores(int &fila, int &id) {
    TVLIWError resul = chkDependencias(fila, id);
    if (resul != VLIW_ERRNO)
        return resul;
    return chkPredicados(fila, id);
}
```

ANEXO B. Encuesta de validación para el alumnado

I. PERFIL DEL ALUMNO

1. Nombre y Apellidos:
2. Viene de:
   ☐ Gestión ☐ Sistemas ☐ Otro:
3. ¿Es la primera vez que cursa esta asignatura?
   ☐ Sí ☐ No
4. ¿Cuánto tiempo ha dedicado a probar este software?
   ☐ Nada
   ☐ Menos de 3 horas
   ☐ 3 - 6 horas
   ☐ Más de 6 horas
5. ¿Qué interés tiene para usted esta parte de la asignatura (ILP) con respecto al resto de temas vistos en clase?
   ☐ Más interesante
   ☐ Igual de interesante
   ☐ Menos interesante

En el resto de esta encuesta valore las afirmaciones utilizando la siguiente escala:

4 = Muy de acuerdo	**3** = De acuerdo	**2** = En desacuerdo	**1** = Muy en desacuerdo

II. ASPECTO EDUCATIVO DEL SOFTWARE

1. Este software ayuda a comprender mejor los contenidos de la asignatura.
   ☐ 4 ☐ 3 ☐ 2 ☐ 1
2. Aprendí poco usando este software.
   ☐ 4 ☐ 3 ☐ 2 ☐ 1
3. Este software mantiene mi interés.
   ☐ 4 ☐ 3 ☐ 2 ☐ 1
4. El esquema de las máquinas se entiende fácilmente.
   ☐ 4 ☐ 3 ☐ 2 ☐ 1
5. La ayuda suministrada acerca de los aspectos teóricos de las máquinas es clara y concisa.
   ☐ 4 ☐ 3 ☐ 2 ☐ 1

IV. FUNCIONALIDAD DEL SOFTWARE

1. Las opciones de configuración me parecen suficientes.
   ☐ 4 ☐ 3 ☐ 2 ☐ 1
2. El contenido de la máquina superescalar se muestra con suficiente detalle y claridad.
   ☐ 4 ☐ 3 ☐ 2 ☐ 1
3. El contenido de la máquina VLIW se muestra con suficiente detalle y claridad.
   ☐ 4 ☐ 3 ☐ 2 ☐ 1
4. La creación de nuevos códigos secuenciales es sencilla.
   ☐ 4 ☐ 3 ☐ 2 ☐ 1
5. La creación de nuevos códigos VLIW se realiza de forma clara y sencilla.
   ☐ 4 ☐ 3 ☐ 2 ☐ 1
6. La realización de nuevos ejercicios ofrece muchas posibilidades y es lo suficientemente adaptable.

☐ 4 ☐ 3 ☐ 2 ☐ 1

7. Las opciones de simulación son suficientes.
   ☐ 4 ☐ 3 ☐ 2 ☐ 1
8. La simulación superescalar funciona correctamente.
   ☐ 4 ☐ 3 ☐ 2 ☐ 1
9. La simulación VLIW funciona correctamente.
   ☐ 4 ☐ 3 ☐ 2 ☐ 1

III. ASPECTOS TÉCNICOS DEL SOFTWARE

1. El programa se cuelga a menudo.
   ☐ 4 ☐ 3 ☐ 2 ☐ 1
2. El programa funciona bien en distintos PCs.
   ☐ 4 ☐ 3 ☐ 2 ☐ 1
3. Los mensajes de error son claros.
   ☐ 4 ☐ 3 ☐ 2 ☐ 1
4. Los mensajes de error ayudan a resolver el problema.
   ☐ 4 ☐ 3 ☐ 2 ☐ 1
5. La interfaz es suficientemente clara.
   ☐ 4 ☐ 3 ☐ 2 ☐ 1
6. El programa es fácil de utilizar.
   ☐ 4 ☐ 3 ☐ 2 ☐ 1
7. La ayuda suministrada es suficiente.
   ☐ 4 ☐ 3 ☐ 2 ☐ 1

II. OTRAS CUESTIONES

1. ¿Conoce algunos programas similares?
   ☐ Sí ☐ No
   ¿Cuáles?

2. Si conoce otros programas, valore este software en relación con ellos:
   - Interfaz ☐ Mejor ☐ Igual ☐ Peor
   - Ayuda ☐ Mejor ☐ Igual ☐ Peor
   - Adecuación de los contenidos ☐ Mejor ☐ Igual ☐ Peor
   - Interacción con el usuario ☐ Mejor ☐ Igual ☐ Peor
   - Estabilidad ☐ Mejor ☐ Igual ☐ Peor
   - Potencia ☐ Mejor ☐ Igual ☐ Peor
3. ¿Cree que la interfaz del programa podría mejorarse?
   ☐ Sí ☐ No
   Indique las posibles mejoras que aplicaría al interfaz del programa

4. ¿Cree que el programa debería mejorarse con nuevas capacidades o cambiando alguna de las disponibles?

   ☐ Sí ☐ No

   Indique los cambios que cree que harían más interesante o más útil este programa

5. Si detectó algún fallo al ejecutar el programa le ruego que me lo indique de la forma más detallada posible para poder corregirlo inmediatamente

6. En este espacio escriba cualquier comentario adicional que crea que pueda aportar algo interesante al proyecto