Desarrollo de un sistema simulador de la arquitectura SPARC mediante el ambiente operativo NeXTSTEP

Tomás Balderas Contreras
Facultad de Ciencias de la Computación
Benemérita Universidad Autónoma de Puebla

Asesor
Mtro. Hugo García Monroy
Departamento de Aplicación de Microcomputadoras
Instituto de Ciencias
Benemérita Universidad Autónoma de Puebla

Mayo, 2000

Resumen

A través de los años, los científicos e investigadores han elaborado diferentes herramientas para reproducir diversos fenómenos o bien para simular y analizar el comportamiento de diferentes sistemas, en particular los dispositivos electrónicos. El desarrollo de estas herramientas ha cambiado la forma de trabajar de los diseñadores así como la clase de habilidades requeridas para realizar su labor. Anteriormente, un diseñador de circuitos necesitaba construir un prototipo y verificarlo en un laboratorio, ahora requiere de ciertos dotes para tratar con herramientas de software que le permiten editar un diseño y observar su comportamiento, adicionalmente debe conocer el modo de resolver los problemas que se presenten.

La simulación auxilia en el proceso de diseño y manufactura de dispositivos, por medio de ella es posible construir modelos de cierto tipo de sistemas mediante el uso de diversas plataformas de computación. Dichos modelos recopilan información relativa al funcionamiento del sistema, la cual es útil para determinar el impacto del diseño en el rendimiento del sistema.

En este documento describimos las características de SPARCSim, un sistema simulador del conjunto de instrucciones de la arquitectura de procesadores SPARC V8, la forma en que se realizó su diseño e implantación y las herramientas utilizadas, entre ellas un ambiente de trabajo y desarrollo orientado a objetos denominado NeXTSTEP. El sistema simulador está concebido como una aplicación que brinda soporte a la codificación, ejecución y depuración de programas escritos en lenguaje ensamblador SPARC, además proporciona estadísticas sobre la ejecución de las instrucciones.

Contenido

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

La *simulación* es la representación del comportamiento de un sistema físico o abstracto mediante otro sistema que puede ser implantado mediante software en una computadora. Su aplicación ha tenido un gran impacto en diversas áreas de la actividad humana, en particular en el campo del diseño y verificación de dispositivos electrónicos. En este ámbito el empleo de herramientas de software ha tenido como efecto cambiar tanto la forma de trabajar de un diseñador como el tipo de habilidades requeridas para realizar su labor, conservando su capacidad creativa e intelectual [Wilkes 1990].

Esta disertación se enfoca principalmente en el estudio y desarrollo de un tipo especial de herramienta que ha cobrado gran importancia en el proceso de diseño, verificación y análisis de rendimiento de computadoras, el *simulador de conjunto de instrucciones* (*instruction set simulator*). Un simulador de conjunto de instrucciones es un programa ejecutado por un procesador existente, definido por una arquitectura específica, cuyo objetivo es modelar la ejecución del conjunto de instrucciones propio de un procesador diseñado a partir de otra arquitectura. Estas dos arquitecturas pueden ser distintas o pueden ser la misma.

Una extensión a este enfoque es el *simulador de máquina completo* (*complete machine simulator*) que modela todos los dispositivos que componen una computadora[1], los sistemas existentes dentro de este esquema son: g88 [Bedichek 1990], SimICS/sun4m [Magnusson et al. 1998] y SimOS [Herrod 1998, Witchel y Rosenblum 1996]. Desarrollar un simulador como los anteriores es un proyecto ambicioso que requiere una inversión considerable de tiempo en investigación, en programación y en verificación, necesita además el esfuerzo de un equipo de personas dedicadas a estas labores y, finalmente, el soporte de un presupuesto adecuado.

Las ventajas y utilidades de las herramientas de simulación, de cualquiera de las dos clases mencionadas anteriormente, deben ser evidentes para cualquier persona con verdaderos conocimientos del funcionamiento y desarrollo de sistemas de computación, aún así es conveniente mencionarlas:

- La simulación de todos los dispositivos presentes en una computadora permite la ejecución de sus sistemas operativos sin realizarles modificación alguna, esto a su vez hace posible el desarrollo y ejecución de diversas aplicaciones. Esto es una ventaja considerable en el caso de que no se disponga físicamente de la computadora necesaria por diversas razones, entre ellas su alto costo.

- Los componentes del simulador además de modelar el funcionamiento de un dispositivo específico recopilan información estadística referente a su comportamiento. Esta información es de gran utilidad al desarrollador del dispositivo para determinar el impacto que una mejora al diseño tendría en el rendimiento del sistema. La información estadística es importante también para el programador de software de sistema o de aplicación para mejorar la eficiencia de sus programas.

[1] CPU, memoria principal, MMU, memoria caché, TLB, dispositivos de entrada y salida junto con sus interfaces, controladores de interrupción y DMA, temporizadores, adaptador Ethernet, etc.

- Cuando una computadora basada en una nueva arquitectura de microprocesador sale al mercado esta debe estar acompañada de un sistema operativo confiable y de algunos programas de aplicación. El uso de un simulador permite que el software sea desarrollado, probado y validado al mismo tiempo que se construye la plataforma de hardware que lo soportará.

- Un simulador funciona perfectamente como una herramienta didáctica, su uso permite conocer y verificar técnicas tradicionales en el diseño y puesta a punto de computadoras y/o sistemas operativos y, al mismo tiempo, proporciona un ambiente adecuado para experimentar con nuevas propuestas.

A pesar de las virtudes anteriores el desarrollo de este tipo de sistemas simuladores puede no ser conveniente a los intereses de algunas personas. En [Taylor 1999] el lector encontrará algunos comentarios sobre UltraHLE y Virtual Game Station, dos simuladores que permiten a los usuarios ejecutar en sus computadoras personales la gran variedad de programas de juego existentes para las plataformas Nintendo 64 y Sony PlayStation, respectivamente. Tales sistemas, junto con la inmensa cantidad de sitios en el World Wide Web donde es posible descargar versiones ilegales de los programas de juego, son una verdadera amenaza para varias de las corporaciones dedicadas a desarrollar y comercializar este tipo de entretenimiento.

El objetivo general de este trabajo de tesis es desarrollar un sistema simulador del conjunto de instrucciones de la arquitectura SPARC V8 para el ambiente NeXTSTEP, al cual se ha nombrado SPARCSim [Balderas y García 1999]. La aplicación está destinada a toda persona interesada en conocer la arquitectura SPARC, permite al usuario consultar y modificar el contenido de la memoria principal y de los registros enteros, de punto flotante y de estado, además permite ensamblar instrucciones escritas en lenguaje ensamblador así como ejecutar y desensamblar instrucciones binarias. El usuario puede escribir programas a nivel de usuario para ser ejecutados por el simulador y, a continuación, puede hacer mejoras a su código para obtener un mejor rendimiento del programa, tomando como base la información estadística proporcionada por SPARCSim. El objetivo particular del autor es conocer el enfoque RISC para el diseño de arquitecturas de microprocesador, investigar técnicas que han sido publicadas y empleadas en la construcción de simuladores y, finalmente, la familiarización con un ambiente operativo innovador y poderoso.

A pesar de sus ventajas, la ejecución de cualquier programa en un simulador es mucho más lenta que su ejecución en la computadora real correspondiente. Existe, de hecho, una relación íntima entre el nivel de detalle proporcionado por un simulador y la velocidad del mismo [Herrod 1998], tal como se muestra en la figura 1.1. Mientras más detalles de la electrónica digital sean simulados más lenta es la ejecución y viceversa. En este punto hay que aclarar que nuestro interés se centra en la simulación de la ejecución del conjunto de instrucciones y en sus resultados, no en la simulación del funcionamiento de un diseño digital o de un circuito integrado.

Generalmente se piensa que NeXTSTEP está en desuso pero la verdad es que ha evolucionado notablemente, de hecho, en la actualidad algunas de sus características y componentes están presentes en algunos sistemas como OPENSTEP, Yellow Box para Windows NT y Mac OS X. Ciertamente no fue muy comercial, pero eso no implica que sea inútil o que el ambiente operativo estándar sea mejor. NeXTSTEP es un ambiente orientado a objetos, compatible con Unix BSD 4.3 y basado en el sistema operativo Mach, sus herramientas de desarrollo, entre ellas el lenguaje orientado a objetos Objective C, permiten construir aplicaciones totalmente funcionales en forma sencilla, agradable y rápida, ventajas que la programación orientada a objetos les ha conferido.

El resto de este reporte se encuentra estructurado de la siguiente forma:

- En el capítulo 2 describimos las características de los paradigmas RISC y CISC.

- El capítulo 3 contiene una amplia descripción de la arquitectura SPARC V8.

- El lenguaje Objective C es descrito con un buen nivel de detalle en el capítulo 4.

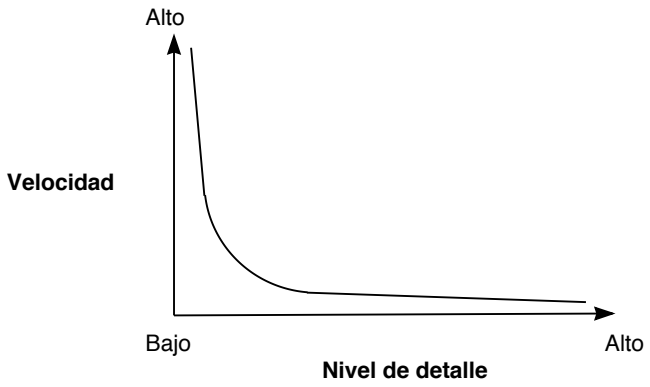


Figura 1.1: Relación entre el nivel de detalle y la velocidad de simulación

- En el capítulo 5 discutimos el ambiente NeXTSTEP y sus herramientas de desarrollo.
- En el capítulo 6 se encuentra una descripción de las funciones del sistema requeridas y de sus limitaciones.
- El diseño e implantación de SPARCSim se presenta en el capítulo 7.
- Los resultados obtenidos en algunas de las pruebas realizadas se discuten en el capítulo 8.
- En el capítulo 9 de este trabajo se encuentran las conclusiones del mismo.

Cabe señalar que este documento está escrito de tal forma que los capítulos 2–5 (el marco teórico) pueden ser consultados de manera aislada en caso de que algún lector se encuentre interesado, particularmente, en alguno de los temas que estos abordan.

A lo largo del presente reporte se utilizan las siguientes convenciones respecto al estilo del texto:

Bold face. Usado para indicar nombres de clases y nombres de archivos.

`Typewriter.` Empleado para hacer referencia a mnemónicos, identificadores de registros, código fuente escrito en lenguaje C y ensamblador así como a nombres de variables.

Italic. Se utiliza para enfatizar frases y palabras importantes.

Sans Serif. Hace referencia a los campos dentro de los formatos de instrucciones, de datos y de registros de estado.

Capítulo 2

Características de RISC y CISC

2.1 Introducción

No es el objetivo de estas páginas el proporcionar un debate sobre la superioridad de alguno de los paradigmas RISC y CISC. Con el paso de los años la presencia de los equipos RISC ha llegado a dominar el área de las estaciones de trabajo, destacándose por su alto rendimiento y por el tipo de aplicaciones que han tenido, tanto en los ámbitos científico y de ingeniería como en el financiero y el del entretenimiento. Algunas arquitecturas RISC de microprocesadores actuales y comerciales son: MIPS de SGI, SPARC de Sun Microsystems, PA-RISC de Hewlett Packard, PowerPC de Apple, IBM y Motorola, POWER de IBM, Alpha de DEC, etc.

Además es necesario notar que aunque parezca contradictorio, conforme transcurre el tiempo los diseños RISC se vuelven más complejos. Existen productos que integran dentro de un microprocesador múltiples unidades de punto flotante, soporte de gráficos e imágenes, predicción de saltos, memoria caché y unidades de manejo de memoria, entre otros atributos [Sun 1998]. A su vez, los procesadores CISC también buscan mayor integración de componentes y toman del paradigma RISC algunos principios para mejorar su rendimiento, un ejemplo bastante claro es el uso de *pipeline*.

A lo largo de las últimas cuatro décadas hemos sido testigos de una sorprendente evolución en la tecnología computacional. Desde la concepción de la computadora con programa almacenado[1] han existido importantes innovaciones en el terreno de la arquitectura y la organización de computadoras, tales innovaciones han impactado de manera notable en la industria del diseño y la fabricación de equipos electrónicos de computación. La siguiente es una lista, no exhaustiva, de dichas innovaciones:

El concepto de familia de computadoras. Introducido por IBM con su serie de computadoras System/360. Mediante este concepto es posible separar la arquitectura de su implantación, de tal forma que podemos encontrar una serie de máquinas que difieren en precio y rendimiento aunque ofrezcan la misma arquitectura al programador y ejecuten todas los mismos programas. Las diferencias antes señaladas se deben a la distinta implantación u organización de una misma arquitectura.

Unidad de control microprogramada. Ciertamente una de las tareas más complejas durante el desarrollo de un procesador es el diseño de su unidad de control. La propuesta de control microprogramado fue hecha por Maurice Wilkes durante los años cincuenta con el fin de facilitar la labor.

Un ciclo de instrucción se completa cuando se han realizado varios subciclos (recolección, decodificación, ejecución, escritura de resultados, etc.), a su vez, estos subciclos se componen de una secuencia de operaciones fundamentales y que involucran transferencias entre los registros del procesador,

[1] Arquitectura propuesta por John von Neumann.

operaciones en la unidad aritmético lógica, etc. Tales operaciones fundamentales son coordinadas por la unidad de control, la cual emite señales de control a los distintos componentes con el fin de permitir que dichas operaciones se lleven a cabo.

La propuesta de Wilkes consistió en agrupar todas las señales de control necesarias para efectuar una operación fundamental en una *palabra de control*, en la cual cada bit indica si la señal de control correspondiente está activa o no. A su vez, agrupando todas las palabras de control para cada operación fundamental de cada subciclo de cada instrucción, se obtiene lo que se conoce como el *microprograma* de tal instrucción. Si se colocan todos los microprogramas correspondientes a cada instrucción en una *memoria de microprograma*, el control del procesador se puede realizar leyendo cada palabra de control o *microinstrucción* del microprograma necesario y emitiendo las señales de control de acuerdo al valor de cada bit en la microinstrucción. Esto es, a muy grandes rasgos, el principio de operación del control microprogramado.

Memoria Caché. Consiste en la adición de una memoria de acceso rápido entre el procesador y la memoria principal. Algunas configuraciones almacenan los datos e instrucciones utilizadas recientemente y otras almacenan las últimas traducciones realizadas por el hardware dedicado al manejo de memoria virtual. De esta manera se establece una jerarquía de memoria que ha demostrado su eficiencia al incrementar considerablemente el rendimiento de los sistemas.

Segmentación encauzada. (Pipeline) Es una forma de conseguir un cierto grado de paralelismo dentro del procesador. La meta principal es reducir el número de ciclos por instrucción a un valor mínimo ideal, este valor es de únicamente *un ciclo por instrucción*.

Múltiples procesadores. Existen diferentes configuraciones de conexión tales como hipercubos, mallas, anillos, etc., así como diferentes objetivos.

Una innovación que merece una mención especial es la introducción de lo que hoy conocemos como arquitectura RISC (*Reduced Instruction Set Computer*). En nuestros días y en años anteriores, hemos sido testigos de como el terreno de las estaciones de trabajo y sistemas operativos basados en Unix, se ha encontrado dominado por este enfoque en el diseño de arquitecturas de microprocesadores.

Durante este capítulo se expondrán las propiedades compartidas por varios procesadores característicos del enfoque RISC. También se presentan los argumentos por los cuales se ha prestado atención hacia los diseños de arquitecturas de computadoras con conjuntos ricos en instrucciones, los diseños CISC (*Complex Instruction Set Computer*).

2.2 Los motivos a favor de conjuntos extensos de instrucciones

Con el propósito de desarrollar software de manera más rápida y confiable, la industria y las instituciones de investigación han concebido lenguajes de programación de alto nivel cada vez más poderosos, sofisticados y complejos, que permiten al programador expresar sus algoritmos de forma más concisa y explotar las facultades que el paradigma al cual pertenece dicho lenguaje pueda ofrecerle[2]. Sin embargo, este desarrollo en el campo de los lenguajes de programación ha ocasionado un efecto colateral denominado *vacío semántico* (*semantic gap*), que está definido como la diferencia existente entre las operaciones proporcionadas por el lenguaje de alto nivel y las operaciones soportadas por la arquitectura de un procesador. Este fenómeno tiene como consecuencia un crecimiento excesivo de las secuencias de instrucciones de máquina y la complejidad en el diseño de los compiladores.

Con la evolución continua de la tecnología de semiconductores es cada vez más viable la implantación de conjuntos de instrucciones cada vez más complejos. He aquí las razones que favorecen el desarrollo

[2]Orientación a objetos, por ejemplo.

de diseños fundamentados en el enfoque CISC, expuestas por la comunidad dedicada al desarrollo de arquitecturas de computadoras [Patterson 1985]:

La riqueza de instrucciones simplifica los compiladores. Debido a que los costos de software se elevan tanto como disminuyen los costos de hardware, es conveniente entonces la transportación de tantas funciones de software como sea posible hacia el hardware. El objetivo es contar con un conjunto de instrucciones de máquina que se asemejen mucho a las sentencias de los lenguajes de alto nivel. De esta manera es posible reducir el vacío semántico entre los lenguajes de alto nivel y los lenguajes de máquina.

La labor de construir un compilador es difícil y se vuelve más complicada para máquinas con registros. Los compiladores diseñados para máquinas con modelos de ejecución basados en pila o en operaciones entre localidades de memoria son más simples y más confiables.

La riqueza de instrucciones mejora la calidad de la arquitectura. Como ya había sido establecida la diferencia entre la arquitectura de una computadora y su implantación u organización, comenzó la búsqueda de técnicas para medir la calidad de la arquitectura. Estas debían ser diferentes a las empleadas para medir la velocidad a la que una implantación ejecuta los programas. Se reconoció ampliamente que una métrica para la arquitectura fuera, por ejemplo, el tamaño de los programas.

Por lo tanto, las características fundamentales de los conjuntos complejos de instrucciones incluyen un número extenso de instrucciones, varios modos de direccionamiento y la implantación mediante hardware de varias sentencias presentes en los lenguajes de alto nivel. De los comentarios anteriores se deduce que tales conjuntos complejos de instrucciones fueron desarrollados para satisfacer, entre otros, dos objetivos primordiales. Primero, facilitar la tarea del escritor de compiladores, quien tiene que generar una serie de instrucciones de máquina para cada sentencia en lenguaje de alto nivel. Si existe una instrucción que se asemeje directamente a dicha sentencia, la labor se hace más sencilla. Esto se encuentra muy relacionado con el segundo objetivo, proporcionar soporte a los lenguajes de alto nivel, cada vez más complejos y sofisticados. Lo anterior es posible gracias a la implantación mediante microcódigo de secuencias de operaciones complicadas.

2.3 Los motivos en contra de conjuntos extensos de instrucciones

Muy a pesar de los argumentos proporcionados anteriormente, a través de los años han sido desarrollados varios estudios que cuestionan las razones que han propiciado la tendencia hacia el enfoque CISC [Stallings 1996].

Se ha cuestionado el hecho de que realmente los conjuntos complejos de instrucciones contribuyen en gran medida a la simplificación de los compiladores y a la generación de programas más pequeños. Los investigadores del paradigma RISC han concluido que a menudo es muy difícil para el compilador explotar de la mejor manera un conjunto muy extenso de instrucciones con el fin de determinar cual de estas es la mejor adaptada a una construcción en un lenguaje de alto nivel. También se ha encontrado que la tarea de optimizar el código generado, con la finalidad de reducir su tamaño o de incrementar el rendimiento del pipeline, es también una labor compleja en una máquina con las características en cuestión. Al mismo tiempo, los estudios han demostrado que existe poca o ninguna diferencia entre el tamaño de un programa compilado para una arquitectura CISC y el tamaño de la secuencia de instrucciones correspondiente en una arquitectura RISC. Esto se debe a que mientras más grande sea el conjunto de instrucciones, el campo para el código de operación debe ser de mayor longitud en bits. Las instrucciones que indican operaciones entre memoria y registros o entre memoria y memoria, requieren de la especificación de las direcciones

necesarias, lo que hace que la instrucción se extienda. Es claro además que debido a la complejidad del conjunto de instrucciones, la longitud de las mismas puede ser muy variable. Los resultados de los estudios muestran también que aunque el código en lenguaje ensamblador para una arquitectura CISC sea más corto (con menos instrucciones) que el código correspondiente para una arquitectura RISC, puede no existir esta diferencia entre el tamaño en bytes de las secuencias de instrucciones binarias correspondientes, como ya se ha mencionado.

El rendimiento del pipeline puede llegar a ser malo. Pueden requerirse varios ciclos para recolectar de la memoria una instrucción completa, retrasando el tiempo para que la siguiente instrucción pueda ser recolectada o incluso para que alguna anterior pueda completar su proceso de ejecución. Si el objetivo es mantener uniforme el número de ciclos para la obtención de instrucciones, la solución inmediata es contar con un conjunto de instrucciones de la misma longitud.

2.4 Los rasgos más importantes de RISC

Diferentes investigadores en diversas industrias privadas e instituciones de investigación iniciaron proyectos para cambiar la orientación en el diseño de computadoras. Dejaron de lado el enfoque iniciado con el objetivo de reducir el vacío semántico y trabajaron en ideas innovadoras. Los proyectos iniciales en el desarrollo de las arquitecturas RISC son los siguientes:

IBM 801. Este proyecto fue administrado por G. Radin sobre las ideas expuestas por John Cocke, a quien se le ha atribuido la concepción de los fundamentos del enfoque RISC [Patterson 1985, Stallings 1996].

RISC I y RISC II. Son dos proyectos desarrollados en la Universidad de California en Berkeley. El primero de estos se debe a un grupo dirigido por David A. Patterson. Fueron los miembros de este equipo quienes acuñaron el término RISC [Patterson 1985, Tabak 1990]. Trabajos posteriores consistieron en implantar diseños RISC de propósito específico para soportar lenguajes como Smalltalk y Lisp.

MIPS. (Microprocessor without Interlocked Pipeline Stages) Proyecto anunciado por la Universidad de Stanford poco después del trabajo RISC de Berkeley. El proyecto fue dirigido por John L. Hennessy y más tarde se convirtió en un producto de explotación comercial por parte de SGI [Tabak 1990].

CDC 6600. Desarrollado por Seymour Cray y considerado por muchos como un "prototipo" de los diseños RISC siguientes, pues contaba con varias de las características atribuidas posteriormente a los sistemas RISC. De hecho, un significado alternativo atribuido al término RISC es "*Really Invented by Seymour Cray*".

Las siguientes son las características que estos diseños tienen en común:

- Todas las operaciones aritméticas y lógicas operan únicamente sobre los valores contenidos en los registros. Únicamente ciertas instrucciones de carga (load) y almacenamiento (store) son capaces de realizar accesos a la memoria o a los dispositivos. Mediante este enfoque se fomenta el uso de técnicas para optimizar el uso de los registros, pues es a través de ellos como es posible implantar la reutilización de operandos en los compiladores. Además se simplifica el conjunto de instrucciones, puesto que solo es necesario implantar una o dos variantes de la instrucción para cada operación aritmética y lógica.

- Existen pocos modos de direccionamiento e instrucciones reducidas. Las direcciones son calculadas mediante los valores en los registros y/o valores inmediatos, también mediante desplazamientos

relativos al valor del Contador de Programa (`PC`, *Program Counter*). Distintos modos de direccionamiento presentes en máquinas diferentes pueden ser simulados o sintetizados mediante el software. Siendo los operandos de las instrucciones valores almacenados en registros, es posible leer tales valores en un solo ciclo y en el siguiente ciclo realizar la operación indicada, de tal modo que las operaciones entre registros son ejecutadas en un ciclo.

- Existen pocos formatos simples de instrucción. El enfoque RISC dicta que todas las instrucciones sean codificadas mediante un patrón de bits de longitud fija. Como se mencionó anteriormente si todas las instrucciones tienen la misma longitud se simplifica el ciclo de lectura de instrucción. La longitud de las instrucciones suele ser igual a la longitud de la palabra del procesador y de su bus de datos. Además, dentro de este patrón, los campos para el código de operación y referencias a los operandos se encuentran en el mismo lugar a lo largo de los formatos para los diferentes tipos de instrucción. De esta forma la decodificación del código de operación puede ser realizada al mismo tiempo que el acceso a los operandos en el archivo de registros.

- Se realiza un uso extenso de segmentación encauzada o pipeline. La técnica de pipeline consiste en segmentar un proceso computacional en diferentes subprocesos, de tal manera que cada uno de estos puede ser llevado a cabo por una unidad funcional dedicada y autónoma. Cuando la primera unidad completa el primer subproceso la segunda unidad inicia el segundo subproceso y así sucesivamente hasta que el proceso ha finalizado. Sucesivos procesos pueden ser realizados en forma solapada, cada uno en una unidad diferente. La técnica puede ser aplicada a diferentes niveles. En un *nivel de sistema* podemos segmentar el proceso de ejecución de instrucciones y en un *nivel de subsistema* podemos segmentar distintas unidades aritméticas[3]. El objetivo de esta técnica es alcanzar el valor ideal de un único ciclo por instrucción. Todas las arquitecturas RISC tienen una ruta de datos segmentada y optimizan su empleo mediante el uso de una *transferencia de control retardada*. Este esquema plantea que una ramificación se lleve a cabo después de ejecutar la instrucción siguiente a la instrucción que originó dicha ramificación, es decir, la transferencia de control se lleva a cabo con un retardo de una instrucción. Mediante este principio es posible evitar las penalizaciones impuestas por la segmentación sobre las instrucciones siguientes a la de transferencia de control. Debido a que una instrucción siempre es recolectada no importando si una instrucción de transferencia de control anterior modifica el contador de programa, los compiladores para arquitecturas RISC están diseñados para colocar instrucciones útiles después de instrucciones de ramificación.

- La simplicidad del conjunto de instrucciones permite descartar el esquema de microprogramación para implantar la unidad de control. En su lugar se puede incluir lógica combinacional para cada señal de control.

En la tabla 2.1 se ilustran algunos de los conceptos descritos anteriormente aplicados a los diseños comentados al principio de esta sección.

Los puntos anteriores describen las características compartidas entre los diseños RISC comentados al principio de esta sección, sin embargo, también existen diferencias entre los mismos que es necesario resaltar. En los siguientes párrafos se discute un aspecto bastante importante, el relacionado con la forma de almacenamiento de los operandos de los procedimientos. Este rasgo específico constituye una de las características más importantes de los diseños RISC posteriores.

2.5 Archivo extenso de registros

A través de los años, diferentes investigadores han efectuado estudios sobre el código de programas compilados a partir de lenguajes de alto nivel. Recolectaban sus datos durante la ejecución de los mismos,

[3]Es posible, por ejemplo, implantar un sumador segmentado de punto flotante.

	IBM 801	**RISC I**	**MIPS**
Año	1980	1982	1983
Número de instrucciones	120	39	55
Memoria de control	No	No	No
Longitud de instrucción	32 bits	32 bits	32 bits
Tecnología	ECL MSI	NMOS VLSI	NMOS VLSI
Modelo de ejecución	reg-reg	reg-reg	reg-reg

Tabla 2.1: Características principales de tres procesadores RISC operacionales

es decir, de manera dinámica verificaban ciertos aspectos referentes a la ejecución de tales programas. Los resultados obtenidos son dignos de comentar. En algunos estudios se encontró que de todos los tipos de instrucciones, las que consumen más tiempo de ejecución son las instrucciones para la invocación y el regreso de procedimientos. Así mismo, se encontró que las instrucciones requerían del acceso constante a operandos, es decir, a datos del programa. Estos datos resultaron ser en su gran mayoría variables de tipo escalar, ya sea datos simples o apuntadores a estructuras de datos, y además eran variables locales. Por otro lado, los resultados de los estudios indican el uso de un número reducido de parámetros en las llamadas a procedimientos y de variables dentro de los mismos. De los últimos dos comentarios se deduce que las variables utilizadas por cada módulo del programa son escalares, locales y además son pocas, lo mismo que sus parámetros. Como conclusión se decidió optimizar el mecanismo de llamadas a procedimientos y la forma en que los datos de dichos módulos debían ser almacenados.

La solución concebida por el equipo de Berkeley fue una solución mediante hardware. Decidieron implantar un archivo consistente de un gran número de registros, este conjunto se encuentra dividido en secciones denominadas *ventanas*, compuestas cada una de varios registros [Patterson 1985, Stallings 1996, Tabak 1990]. Cada ventana está destinada a contener las variables escalares locales y los parámetros de un procedimiento. La ejecución de llamadas a procedimientos se vuelve más eficiente si se asigna una ventana distinta para un nuevo procedimiento, esta ventana se activa y la del procedimiento que realizó la llamada se desactiva. De manera similar, un regreso desde un procedimiento simplemente libera la ventana que dicho procedimiento utilizaba y activa la ventana del procedimiento que lo invocó. El enfoque anterior evita la necesidad de salvar el contenido de los registros en memoria y su posterior recuperación durante el llamado y el regreso de procedimientos. Con la finalidad de realizar el paso de parámetros entre procedimientos de manera eficiente y evitar copiar valores entre ventanas de registros durante cada llamado, se determinó que las ventanas se solaparan, de tal manera que algunos registros forman parte de dos ventanas simultáneamente. De esta manera el paso de parámetros consiste únicamente en escribir valores en estos registros compartidos y realizar un cambio a la siguiente ventana.

Por otro lado, los grupos en IBM y en Stanford decidieron no extender el archivo de registros y dejar la tarea de asignación de los mismos al diseñador del compilador, quien tiene que implantar técnicas para hacer corresponder las variables del procedimiento dentro del grupo de registros disponibles.

Capítulo 3

La arquitectura SPARC

3.1 Introducción

Como se mencionó en el capítulo 2, existe una diferencia muy marcada entre la arquitectura de una computadora y su implantación. A menudo se hace uso del término *arquitectura de computadoras* para hacer referencia al repertorio de instrucciones del procesador o bien para hablar de la imagen que este ofrece al programador en lenguaje ensamblador. Por otro lado, el terreno de la implantación puede ser dividido en las dos partes siguientes: la *organización*, que tiene que ver con los detalles de alto nivel en el proceso de diseño del procesador, tales como su distribución interna, el sistema de memoria, la segmentación, etc. La parte de *hardware* involucra el diseño lógico detallado de la organización y trata con ciertos aspectos más específicos, tales como la tecnología de integración, la tecnología de semiconductores, etc [Stallings 1996].

Sun Microsystems definió la arquitectura SPARC (*Scalable Processor Architecture*) entre los años 1984 y 1987. El desarrollo de esta arquitectura estuvo basado en los trabajos sobre RISC realizados en la Universidad de California en Berkeley entre 1980 y 1982, los cuales son mencionados en el capítulo 2. Los científicos de Sun, quienes poseían amplia experiencia en sistemas operativos, compiladores y diseño de hardware, realizaron varias modificaciones con el propósito de mejorar los diseños producidos en Berkeley. En julio de 1987 Sun anunció la arquitectura SPARC y un sistema completamente basado en ella, Sun-4/200. A principios del año siguiente anunció un sistema de más bajo costo, Sun-4/110 [Garner 1990].

La visión de la compañía fue establecer un ambiente de computación abierto y estándar. Sun Microsystems proporcionó licencias a distintos fabricantes para explotar la arquitectura y desarrollar implantaciones propias de la misma, con el propósito de ofrecer productos distintos en precio y rendimiento, pero capaces todos de ejecutar las aplicaciones binarias existentes para la arquitectura SPARC. Cabe señalar que los derechos correspondientes a una implantación específica de la arquitectura son propiedad de su respectivo desarrollador. Con el paso del tiempo se instituyó el consorcio SPARC International, dedicado a dirigir la evolución y la estandarización de la arquitectura SPARC. El consorcio registra la evolución de la arquitectura en documentos tales como *El manual de la arquitectura SPARC* [SPARC 1992]. Tales documentos son empleados por los fabricantes de hardware y desarrolladores de software de aplicación o de sistemas con la finalidad de que sus productos cumplan los estándares establecidos por SPARC International y de esta forma asegurar la compatibilidad. Entre las industrias que conforman dicha organización podemos mencionar a Fujitsu, Cypress Semiconductor, Texas Instruments, Phillips y Sun Microsystems.

A través de los años la arquitectura ha evolucionado notablemente. Los cambios han sido registrados en los manuales de las diferentes versiones de la arquitectura. La versión más reciente de la arquitectura SPARC es la versión 9. Las implantaciones de esta versión (UltraSPARC I, UltraSPARC II y UltraSPARC

IIi [Sun 1998], por ejemplo) tienen, entre otras características, frecuencias superiores a los 300 MHz. Es necesario aclarar que también existe compatibilidad entre las diferentes versiones de la arquitectura SPARC.

SPARC define registros enteros de propósito general, registros para valores de punto flotante, registros de estado del procesador y de control, además de 69 instrucciones básicas. Asume un espacio de 32 bits de direcciones virtuales para los programas de aplicación del usuario. SPARC no define componentes del sistema que son visibles únicamente por el sistema operativo y no por los programas a nivel del usuario[1].

En este capítulo se describirán con un cierto nivel de detalle los rasgos más importantes de la versión 8 de la arquitectura. Al final del capítulo se mencionarán los puntos más sobresalientes de la versión 9. El lector interesado puede consultar las referencias para obtener más información sobre esta y otras arquitecturas.

3.2 El procesador SPARC

Un procesador congruente con la arquitectura SPARC V8 se compone de una unidad entera, una unidad de punto flotante y un coprocesador opcional, cada componente cuenta con sus propios registros. Esta organización permite implantaciones con la máxima concurrencia entre la ejecución de instrucciones enteras, de punto flotante y de coprocesador. Todos los registros, con la posible excepción de los del coprocesador, son de 32 bits de longitud. Los operandos de las instrucciones son generalmente registros simples, pares de registros o grupos de cuatro registros.

En cualquier momento, el procesador se encuentra en uno de dos modos de operación, *modo usuario* y *modo supervisor*. Cuando se encuentra en el modo supervisor, el procesador puede ejecutar cualquier instrucción, incluyendo las instrucciones privilegiadas. En el modo usuario cualquier intento de ejecutar una instrucción privilegiada provocará una excepción hacia el software supervisor. Las aplicaciones del usuario son programas que se ejecutan mientras el procesador se encuentra en modo usuario.

3.2.1 La unidad entera

La unidad entera contiene los registros de propósito general y controla la operación general del procesador. Ejecuta las instrucciones aritméticas enteras y las instrucciones lógicas, también se encarga de calcular las direcciones de memoria para las instrucciones de carga y almacenamiento. Además debe mantener los contadores de programa y controlar la ejecución de instrucciones para la unidad de punto flotante y el coprocesador.

3.2.1.1 Los registros de propósito general

Dependiendo de una implantación particular, un microprocesador SPARC puede contener un archivo consistente de 40 a 520 registros enteros. El archivo de registros enteros está dividido en dos partes. Primero, un grupo conformado por 8 *registros globales* (referidos como `%r0 ...%r7` ó `%g0 ...%g7`), de los cuales el primero (`%r0` ó `%g0`) contiene invariablemente el valor cero. Segundo, un arreglo circular de registros, el cual está dividido a su vez en ventanas consistentes cada una de varios registros, tal y como se explicó en el capítulo 2. La figura 3.1 ilustra la organización de un archivo con 4 ventanas.

En cualquier momento, las instrucciones pueden tener acceso a 32 registros enteros. Los ocho registros globales junto con los registros contenidos en la ventana actual. Cada ventana está conformada por ocho *registros de salida* (referidos como `%r8 ...%r15` ó `%o0 ...%o7`), ocho *registros locales* (referidos como `%r16 ...%r23` ó `%l0 ...%l7`) y ocho *registros de entrada* (referidos como `%r24 ...%r31` ó `%i0 ...%i7`). La figura 3.1 nos muestra que cada ventana se encuentra solapada con las dos ventanas adyacentes. Los registros de salida de la ventana i son realmente los registros de entrada de la ventana $i-1$. De modo

[1] Registros de E/S, memoria caché y unidades de manejo de memoria, por ejemplo.

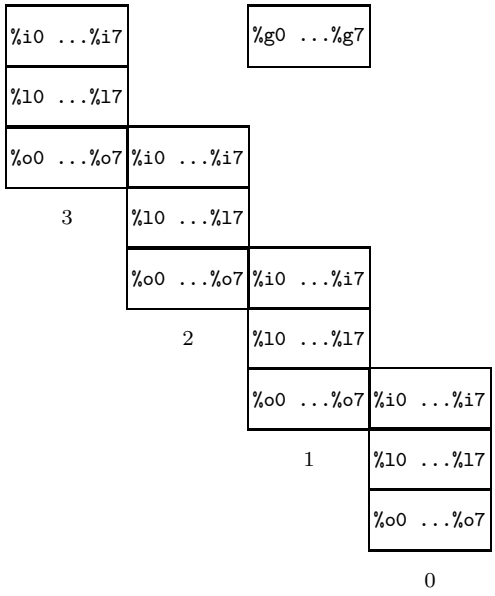


Figura 3.1: Archivo de registros enteros dividido en ventanas

similar, los registros de salida de la ventana $i+1$ son en realidad los registros de entrada de la ventana i. Para el caso de la primera ventana, sus registros de salida son en realidad los registros de entrada de la última ventana. De lo anterior podemos deducir que físicamente cada ventana está conformada por 16 registros. De tal forma que, si una implantación tiene n ventanas, entonces tiene un número total de $16n + 8$ registros[2]. Como $2 \leq n \leq 32$, entonces el número mínimo posible de registros enteros de propósito general es 40 y el máximo 520, como anteriormente se mencionó. El número total de ventanas es dependiente de la implantación e invisible a los programas de aplicación del usuario.

El objetivo de la disposición del archivo de registros en ventanas solapadas es la minimización del retardo en las llamadas a procedimientos. Los registros de salida (*outs*) y de entrada (*ins*) son utilizados primordialmente para enviar parámetros a una subrutina y recibir resultados de ella. Un procedimiento pasa argumentos a un módulo escribiendo estos valores en los registros de salida de la ventana actual, entonces el *apuntador a la ventana actual* es decrementado y el nuevo procedimiento puede ahora recoger sus parámetros desde los registros de entrada de la nueva ventana. De manera inversa, el procedimiento regresa sus valores escribiéndolos en los registros de entrada de la ventana que le corresponde, realizando el regreso del procedimiento e incrementando el apuntador a la ventana actual. De esta manera el procedimiento anterior se activa nuevamente y puede tomar los valores devueltos por la subrutina desde los registros de salida de su ventana. Un procedimiento puede emplear hasta seis de los registros de salida para realizar el paso de valores, puesto que el registro `%o6` está reservado para contener el apuntador de la pila y el registro `%o7` contiene la dirección de la instrucción (`call`) que provocó el llamado a la subrutina.

3.2.1.2 Los registros de control y de estado

El registro de estado del procesador (`PSR`) está conformado por varios campos que controlan el procesador y que contienen información sobre su estado. Una ilustración del registro se muestra en la figura 3.2 y a

[2]Se suma ocho debido a que se deben contar los registros que conforman el grupo de registros globales y que no pertenecen a alguna de las ventanas.

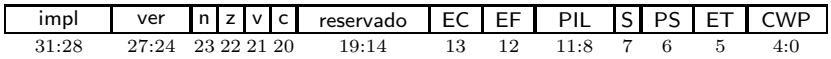


Figura 3.2: El registro de estado del procesador

continuación se proporciona una breve descripción de sus campos más relevantes:

Códigos enteros de condición. (icc) Estos bits son modificados por las instrucciones aritméticas y lógicas cuyos mnemónicos contienen el sufijo cc (por ejemplo `andcc`). Las instrucciones de ramificación condicional provocan una transferencia de control basada en el valor de estos bits, que se definen como sigue:

Negativo. (n) El bit 23 indica si el resultado de 32 bits en complemento a dos producido por la ALU fue negativo para la última instrucción que modifica los códigos de condición. 0 = no negativo, 1 = negativo.

Cero. (z) El bit 22 indica si el valor de 32 bits dado como resultado por la ALU fue cero para la última instrucción que modifica los códigos de condición. 0 = distinto de cero, 1 = cero.

Desbordamiento. (v) El bit 21 indica si el resultado proporcionado por la ALU puede ser representado en 32 bits con notación de complemento a dos. 0 = no desbordamiento, 1 = desbordamiento.

Acarreo. (c) El bit 20 se activa si hubo un acarreo fuera del bit 31 para el caso de una adición o si hubo un préstamo dentro del bit 31 para el caso de una sustracción. 0 = no acarreo, 1 = acarreo.

Coprocesador habilitado. (EC) El bit 13 determina si se encuentra habilitado o no el coprocesador[3]. 0 = no habilitado, 1 = habilitado.

Unidad de punto flotante habilitada. (EF) El bit 12 indica si la unidad de punto flotante se encuentra habilitada o deshabilitada. 0 = no habilitada, 1 = habilitada.

Modo de operación. (S) El bit 7 indica el modo en el que el procesador se encuentra operando. 0 = modo usuario, 1 = modo supervisor.

Apuntador a la ventana actual. (CWP) Un contador que identifica la ventana actual en el archivo de registros. El hardware decrementa el contador durante la ejecución de la instrucción `save` y lo incrementa como resultado de la ejecución de la instrucción `restore`.

Las instrucciones para llevar a cabo multiplicación (`umul`, `umulcc`, `smul`, `smulcc`) realizan el producto de dos valores enteros de 32 bits (con o sin signo) y generan un resultado de 64 bits, cuyos 32 bits más significativos se almacenan en un registro especial, el registro `Y`. De manera similar, las instrucciones `udiv`, `udivcc`, `sdiv` y `sdivcc` realizan la división (con o sin signo) de un valor entero de 64 bits entre otro valor entero de 32 bits, generando un resultado de 32 bits. Los 32 bits más significativos del dividendo son tomados desde el registro `Y`.

Existen dos registros contadores de programa de 32 bits de longitud, denominados `PC` y `nPC`. El registro `PC` contiene la dirección de la instrucción en proceso de ejecución. El registro `nPC` contiene la dirección de la próxima instrucción a ser ejecutada.

[3]Dependiente de la implantación.

3.2.2 La unidad de punto flotante

La unidad de punto flotante definida por la arquitectura SPARC V8 contiene 32 registros de 32 bits de longitud cada uno. Esta unidad soporta tipos de datos de precisión simple (32 bits), precisión doble (64 bits) y precisión extendida o cuádruple (128 bits). De lo anterior se deduce que el archivo de registros de punto flotante puede almacenar 32 datos de precisión simple, 16 datos de doble precisión y 8 datos de precisión extendida. La figura 3.3 muestra la disposición del archivo de registros de punto flotante.

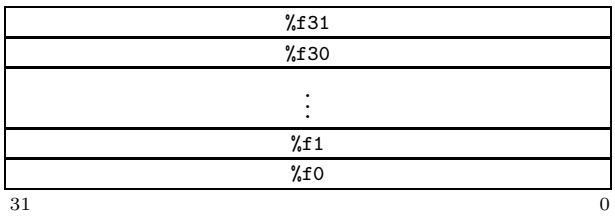


Figura 3.3: Archivo de registros de punto flotante

Los tipos de datos de punto flotante mencionados anteriormente, así como el conjunto de instrucciones están conformes al estándar IEEE para aritmética binaria de punto flotante[4] [Goldberg 1991, Patterson y Hennessy 1993]. Sin embargo, la arquitectura SPARC no requiere que todas las características del estándar sean implantadas en hardware, ya que mediante software es posible simular los aspectos no presentes en el hardware.

Un componente importante de la unidad de punto flotante es el *registro de estado de punto flotante* (`FSR`). Contiene varios campos que proporcionan cada uno cierta información con respecto al estado de tal unidad. En particular, contiene los códigos de condición de la unidad de punto flotante (bits 11 y 10). Estos bits reflejan el resultado de la ejecución de las instrucciones de comparación `fcmps` y `fcmpd`. En la tabla 3.1, f_{rs1} y f_{rs2} hacen referencia a valores en el archivo de registros de punto flotante especificados como operandos en alguna de las instrucciones anteriores. Se muestra el valor del campo fcc para cada relación posible entre tales valores. El signo de interrogación indica que no es posible establecer una relación de orden entre los valores debido a que alguno de ellos tiene el valor especial *NaN*, definido por el estándar IEEE.

fcc	**Relación**
0	$f_{rs1} = f_{rs2}$
1	$f_{rs1} < f_{rs2}$
2	$f_{rs1} > f_{rs2}$
3	$f_{rs1} ? f_{rs2}$ (sin orden)

Tabla 3.1: Valores de los códigos de condición de punto flotante

3.3 Los formatos de los datos

La arquitectura SPARC V8 reconoce tres formatos (tipos) de datos fundamentales:

Entero con signo. 8, 16, 32 y 64 bits.

[4] ANSI/IEEE Standard 754-1985.

Entero sin signo. 8, 16, 32 y 64 bits.

Punto flotante. 32, 64 y 128 bits.

La longitud de los formatos está definida como sigue:

Byte. 8 bits.

Media palabra. 16 bits.

Palabra/Palabra simple. 32 bits.

Palabra etiquetada. 32 bits[5].

Palabra doble. 64 bits.

Palabra cuádruple. 128 bits.

Los números enteros en complemento a dos se codifican empleando los formatos para enteros con signo. Los formatos para enteros sin signo son de propósito general, en el sentido de que no codifican algún tipo particular de dato, pueden representar un número entero, cadena, fracción, valor booleano, etc. Los formatos de punto flotante están acordes al estándar IEEE 754 para aritmética binaria de punto flotante, tal y como se mencionó en la sección anterior. El formato etiquetado define una palabra en la cual los dos bits menos significativos son tratados como bits de etiqueta.

En las figuras 3.4, 3.5 y 3.6 podemos observar los formatos de los datos soportados por la arquitectura SPARC V8, tanto para valores enteros como para valores de punto flotante. Para estos últimos se muestran los campos que componen cada formato, la longitud de cada campo y su posición dentro del patrón de bits.

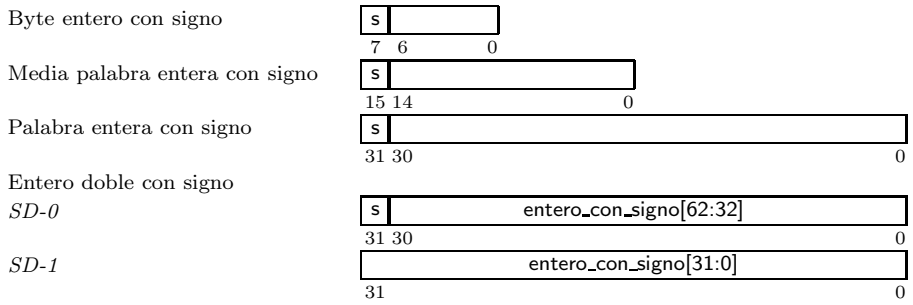


Figura 3.4: Formatos para valores enteros con signo

3.4 El conjunto de instrucciones

Las instrucciones definidas por la arquitectura pueden ser catalogadas en seis categorías distintas:

- Carga y almacenamiento.
- Aritméticas/Lógicas/Corrimiento.
- Transferencia de control.

[5] Un valor entero de 30 bits junto con una etiqueta de dos bits.

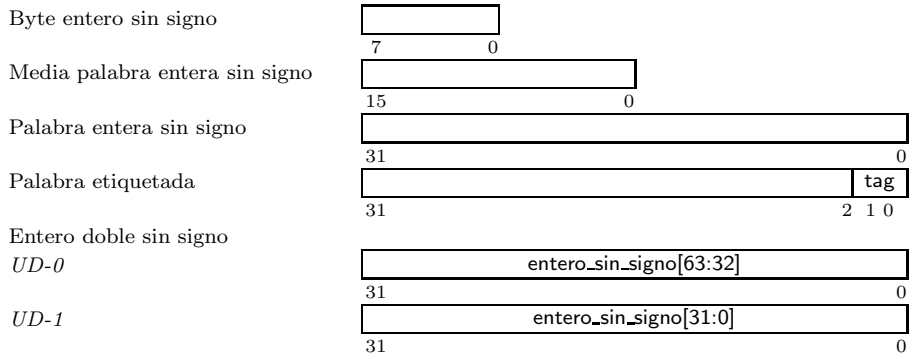


Figura 3.5: Formatos para valores enteros sin signo

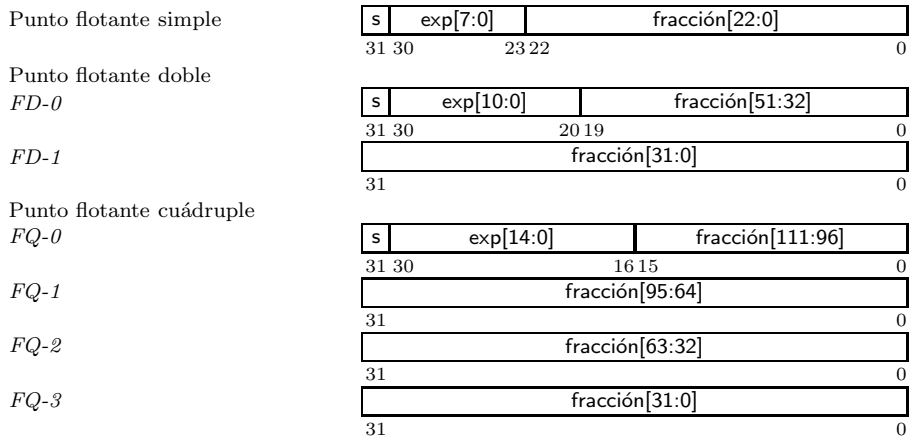


Figura 3.6: Formatos para valores de punto flotante

- Lectura/Escritura de registros de control.
- Operación de punto flotante.
- Operación del coprocesador.

3.4.1 Instrucciones de carga y almacenamiento

Las instrucciones de carga y almacenamiento desplazan bytes (8 bits), medias palabras (16 bits), palabras (32 bits) y palabras dobles (64 bits) entre la memoria y la unidad entera, la unidad de punto flotante o el coprocesador. Estas son las únicas instrucciones que realizan accesos a la memoria principal, la cual, para los programas de aplicación de los usuarios, es un espacio lineal de memoria de 32 bits (4 gigabytes).

Cuando se ejecuta una instrucción de carga o almacenamiento en la unidad de punto flotante, en la unidad entera o en el coprocesador, estos proporcionan o reciben los datos necesarios y la unidad entera genera la dirección de memoria. Las instrucciones de carga y almacenamiento de palabras, medias palabras o palabras dobles provocan una excepción si la dirección no se encuentra alineada correctamente. Cuando se realiza un acceso a una media palabra, la dirección debe estar alineada en los límites de dos

bytes, los accesos a palabras en límites de cuatro bytes y los accesos a dobles palabras en límites de ocho bytes. Existen versiones de las instrucciones de carga de datos enteros que extienden el signo de los valores de 8 y 16 bits cuando son almacenados en el registro destino.

Un aspecto importante que debe ser considerado cuando se mencionan las instrucciones de carga y almacenamiento es el orden en memoria en que los datos son almacenados. Esto tiene que ver con la forma en que el sistema almacena en memoria los bytes que componen un dato escalar (*byte ordering*). La arquitectura SPARC V8 utiliza un esquema de almacenamiento denominado *big endian* [SPARC 1992, Stallings 1996]. Este esquema dicta que la dirección de un dato compuesto de múltiples bytes (media palabra, palabra o doble palabra) es la dirección del byte más significativo y que incrementando la dirección se puede tener acceso a los bytes menos significativos. Por otra parte, el esquema *little endian* establece que la dirección de un dato es la dirección del byte menos significativo, y que incrementando el valor de tal dirección se accesa a los bytes más significativos. En la figura 3.7 se muestra la forma en que el contenido de un registro sería almacenado en memoria mediante cada uno de los enfoques comentados.

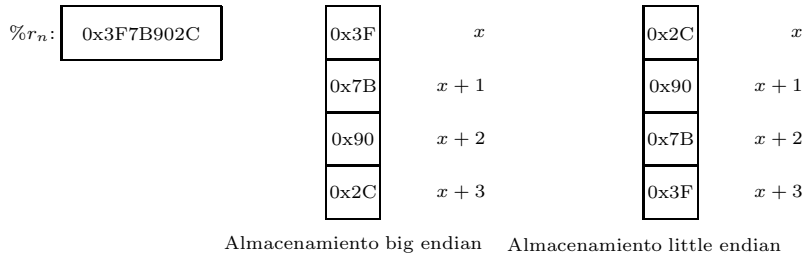


Figura 3.7: Formas de almacenamiento de datos en memoria

Para calcular las direcciones necesarias por las instrucciones de carga y almacenamiento, la arquitectura especifica dos modos de direccionamiento. Estos se definen mediante las siguientes expresiones: $\texttt{reg}_1 + \texttt{simm13}$ y $\texttt{reg}_1 + \texttt{reg}_2$, donde $\texttt{reg}_\texttt{i}$ hace referencia al contenido de cualquiera de los registros enteros en la ventana actual o en el grupo de registros globales, y `simm13` es un valor inmediato de 13 bits expresado en notación de complemento a dos, cuyo signo es extendido para formar un valor de 32 bits que se suma al contenido del registro especificado para obtener la dirección de carga o almacenamiento.

Cada vez que se realiza un acceso a memoria, la unidad entera le proporciona tanto la dirección de 32 bits como un *identificador de espacio de direcciones* o ASI de 8 bits. Algunos de los valores que toma este identificador indican si el procesador se encuentra en modo usuario o en modo supervisor, y si el acceso es a las instrucciones o a los datos del programa. Se proporcionan también versiones privilegiadas de las instrucciones en cuestión, conocidas como instrucciones de *carga y almacenamiento alternas*, las cuales pueden ser ejecutadas únicamente en modo supervisor y en ellas es posible especificar de manera directa un identificador de espacio de dirección arbitrario. Estas instrucciones son empleadas por el sistema operativo para hacer uso de ciertos recursos del sistema, tales como registros de entrada y salida, registros de la unidad de manejo de memoria (MMU), etc.

3.4.2 Instrucciones enteras aritméticas y lógicas

Estas instrucciones llevan a cabo operaciones aritméticas y/o lógicas sobre dos operandos fuente y almacenan el resultado en el registro entero especificado como destino. Se requiere que el primer operando se encuentre almacenado en un registro, mientras que el segundo operando puede encontrarse ya sea en un registro o bien como parte de la instrucción. En este caso, se tiene un valor de 13 bits expresado en notación de complemento a dos, cuyo signo debe ser extendido hasta formar un valor de 32 bits, que puede ser operado para obtener un resultado.

Para cada instrucción aritmética existen dos variantes. La primera de ellas modifica el contenido de los códigos de condición (icc) contenidos en el `PSR` como consecuencia de su ejecución, la segunda variante no realiza tal modificación. Para identificar cada versión de una misma instrucción, el lenguaje ensamblador establece la regla de añadir el sufijo cc al mnemónico de la instrucción para hacer referencia a la versión que modifica los bits de condición (por ejemplo `add` y `addcc` para la suma).

Las instrucciones de corrimiento desplazan el contenido de un registro ya sea a la izquierda o a la derecha una determinada distancia, la cual está especificada por el contenido de otro registro o por un valor inmediato contenido dentro de la misma instrucción.

A partir de la versión ocho de la arquitectura SPARC se definen instrucciones para realizar multiplicación y división entera. Las instrucciones de multiplicación operan sobre dos valores enteros de 32 bits con o sin signo y producen un valor entero de 64 bits, tal y como se mencionó anteriormente. De manera similar, las instrucciones de división producen el cociente de dos valores enteros con o sin signo, de estos, el dividendo es un número de 64 bits y el divisor es un dato de 32 bits. Cada instrucción tiene sus propias variantes para modificar los códigos de condición o no.

Las instrucciones de aritmética etiquetada (*tagged arithmetic*) operan sobre valores enteros con etiqueta (*tag*). La arquitectura define dos operaciones sobre datos de este tipo, suma y sustracción. Estas instrucciones ponen a uno el bit de desbordamiento (overflow) de los códigos de condición si ocurre un desbordamiento aritmético normal o bien si la etiqueta (los dos bits menos significativos) de alguno de los operandos es distinta de cero. De lo anterior se deduce que los valores enteros en este formato se asumen de 30 bits de longitud, justificados a la izquierda de una palabra con etiqueta igual a cero [Garner 1990, SPARC 1992].

3.4.3 Instrucciones de transferencia de control

Las instrucciones de transferencia de control modifican el valor del registro `nPC` como resultado de su ejecución. La dirección destino de la transferencia de control se calcula de distintas formas dependiendo de la instrucción. Puede ser calculada a partir de un desplazamiento relativo al contador de programa (`call` y saltos condicionales) o de acuerdo a los modos de direccionamiento comentados anteriormente (`jmpl`).

Las transferencias de control se encuentran retardadas por una instrucción. Como se mencionó en el capítulo 2, la siguiente instrucción a la de transferencia de control (*instrucción de retardo*) es ejecutada antes de que se lleve a cabo la transferencia. Tal instrucción de retardo siempre es recolectada por el procesador, incluso cuando la instrucción de transferencia de control sea un salto incondicional.

Analicemos la tabla 3.2 que contiene tres secuencias de código escritas en un lenguaje ensamblador arbitrario. La secuencia de ejecución del programa de la primera columna es $100, 101, 102, 105, \ldots$ En una arquitectura RISC con transferencia de control retardada, es necesario agregar una instrucción *no operación* a continuación de la instrucción de transferencia de control, de tal manera que la secuencia de ejecución de las instrucciones de la segunda columna es $100, 101, 102, 103, 106, \ldots$ Un compilador para una arquitectura RISC puede disponer la secuencia de instrucciones para realizar operaciones equivalentes haciendo uso de los espacios para las instrucciones de retardo. Este es el caso en la tercera columna, donde ahora la secuencia de ejecución es $100, 101, 102, 105, \ldots$ Esta nueva disposición de las instrucciones fue posible debido a que no existe alguna dependencia entre la instrucción de salto en la dirección 101 y la instrucción de suma en la dirección 102, por lo que su intercambio no modifica la semántica del programa.

Es necesario aclarar que es posible no ejecutar o anular la instrucción de retardo. Lo anterior se especifica mediante un *bit de anulación* presente en la instrucción[6]. Cuando el valor de este bit es cero, la instrucción de retardo es ejecutada. Si su valor es uno, la instrucción se cancela (no se ejecuta) a menos

[6]Denominado `a`.

Dirección	Salto normal	Salto retardado	Salto retardado optimizado
100	LOAD X, A	LOAD X, A	LOAD X, A
101	ADD 1, A	ADD 1, A	**JUMP 105**
102	**JUMP 105**	**JUMP 106**	**ADD 1, A**
103	ADD A, B	**NO-OP**	ADD A, B
104	SUB C, B	ADD A, B	SUB C, B
105	**STORE A, Z**	SUB C, B	**STORE A, Z**
106		**STORE A, Z**	

Tabla 3.2: Comparación de las instrucciones tradicionales de salto y las instrucciones retardadas de salto

que la instrucción de transferencia de control se trate de un salto condicional aceptado. Una instrucción anulada se comporta como una instrucción no operación. La tabla 3.3 resume tales comentarios.

Bit a	Tipo de ramificación	Instrucción de retardo ejecutada
a = 0	condicional, aceptada	Si
	condicional, no aceptada	Si
	incondicional, aceptada (`ba`, etc.)	Si
	incondicional, no aceptada (`bn`, etc.)	Si
a = 1	condicional, aceptada	Si
	condicional, no aceptada	No (anulada)
	incondicional, aceptada (`ba`, etc.)	No (anulada)
	incondicional, no aceptada (`bn`, etc.)	No (anulada)

Tabla 3.3: Condiciones para la ejecución de la instrucción de retardo

Mediante el empleo de un bit de anulación es posible lograr que un compilador tenga mayores posibilidades de hallar una instrucción útil para colocar en el espacio de una instrucción de retardo. Por ejemplo, supóngase que una instrucción de transferencia de control concluye una secuencia de instrucciones asociadas a un ciclo, entonces el compilador puede colocar alguna de las instrucciones que conforman el lazo en el espacio para la instrucción de retardo. Si el bit a = 1 entonces tal instrucción se ejecutará si la condición se cumple, realizando la transferencia. Si la condición no se cumple entonces la instrucción de retardo no es ejecutada y la transferencia tampoco se realiza.

3.4.4 Instrucciones de punto flotante

Estas instrucciones realizan una operación y producen un resultado de punto flotante de precisión simple, precisión doble o precisión extendida, el cual es función de dos operandos fuente obtenidos desde los registros de punto flotante y es a su vez escrito en el mismo archivo de registros. Las instrucciones de comparación para este tipo de datos modifican los códigos de condición de punto flotante que se encuentran en el registro `FSR`. Estos bits pueden ser revisados más tarde por las instrucciones de ramificación condicional para valores de punto flotante. No existen instrucciones cuyos operandos sean valores de distinta precisión, por lo que se proporcionan instrucciones para convertir datos entre los distintos formatos, incluyendo enteros de 32 bits.

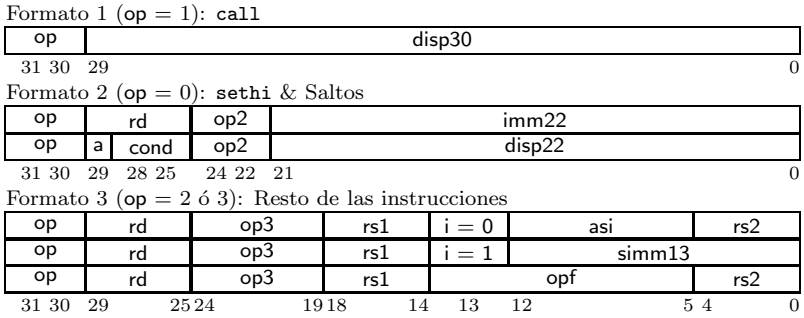


Figura 3.8: Sumario de los formatos de las instrucciones

3.5 Los formatos para las instrucciones

Todas las instrucciones se encuentran codificadas en una palabra de 32 bits. Existen tres diferentes formatos para especificar instrucciones, los cuales se muestran en la figura 3.8 y se describen a continuación:

Formato 1. Contiene un desplazamiento de 30 bits para la instrucción `call`. Este desplazamiento es extendido en signo y desplazado dos bits a la izquierda, de tal manera que se puede realizar una transferencia a una instrucción arbitrariamente distante.

Formato 2. Define instrucciones de salto y la instrucción `sethi`. El desplazamiento de 22 bits permite modificar el contador de programa para realizar una transferencia a una instrucción en un intervalo de ± 8 Mbytes con respecto al valor actual del contador de programa.

Formato 3. Se emplea para codificar el resto de las instrucciones. Permite especificar dos operandos fuente y un registro destino, ya sea para realizar una operación aritmética, lógica, de carga o de almacenamiento. El primer operando siempre hace referencia a un registro. Si el bit $\mathsf{i} = 1$ entonces el segundo operando es un valor inmediato de 13 bits con signo, contenido en la instrucción. Si el bit $\mathsf{i} = 0$ entonces el segundo operando se encuentra contenido en un registro. Dependiendo del tipo de operación, las direcciones de registros especificadas en la instrucción pueden hacer referencia a registros enteros o de punto flotante. Este formato permite especificar también un identificador de espacio de direcciones para las instrucciones privilegiadas de carga y almacenamiento.

La descripción de cada campo en los formatos de instrucciones se proporciona a continuación:

Campo op. Este campo codifica el tipo de formato al que pertenece una instrucción. La tabla 3.4 muestra los valores que puede tomar y el tipo de instrucciones asociadas a cada uno de estos valores.

Formato	op	**Instrucciones**
1	01	`call`
2	00	`sethi` y saltos
3	11	Accesos a memoria
3	10	Aritméticas, lógicas, corrimiento y otras

Tabla 3.4: Codificación del campo op

op2	**Instrucción**
000	`unimp`
010	Saltos basados en los códigos de condición enteros
100	`sethi`
110	Saltos basados en los códigos de condición de punto flotante
111	Saltos basados en los códigos de condición del coprocesador

Tabla 3.5: Codificación del campo op2 (formato 2)

Campo op2. Este campo selecciona alguna de las diferentes instrucciones que pueden ser codificadas empleando el formato 2. La tabla 3.5 muestra la asignación de las instrucciones con los valores.

Campo rd. Este campo especifica la dirección del registro (entero, de punto flotante o de coprocesador) donde se deposita el resultado de la ejecución de una instrucción aritmética, lógica o de corrimiento. En caso de que se trate de una instrucción de carga o almacenamiento, este campo indica el registro en donde se depositará el dato leido de memoria o el registro que contiene el dato a escribir en memoria, respectivamente.

Campo a. Este bit en una instrucción de transferencia de control cancela la ejecución de la siguiente instrucción, siempre que la instrucción de transferencia de control se trate de un salto condicional que no se lleva a cabo o si es un salto incondicional.

Campo cond. Este campo de 4 bits selecciona los códigos de condición que han de ser verificados durante una instrucción de ramificación condicional.

Campo imm22. Este campo contiene una constante de 22 bits que es empleada por la instrucción `sethi`.

Campos disp22 y disp30. Son valores con signo de 22 y 30 bits de longitud respectivamente. Son empleados como desplazamientos relativos al contador de programa en las instrucciones de salto y llamados a procedimientos.

Campo op3. Especifica el código de operación de una instrucción codificada mediante el formato 3.

Campo i. Este bit selecciona el segundo operando para aritmética entera y para calcular direcciones para instrucciones de carga o almacenamiento. Si $i = 0$, el operando se obtiene de un registro. Si $i = 1$, el operando es un valor inmediato.

Campo asi. Indica un identificador de espacio de dirección proporcionado por una instrucción privilegiada de carga o almacenamiento.

Campo rs1. Este campo es la dirección del registro (entero, de punto flotante o de coprocesador) donde se encuentra el primer operando fuente.

Campo rs2. Este campo es la dirección del registro (entero, de punto flotante o de coprocesador) donde se encuentra el segundo operando fuente cuando $i = 0$.

Campo simm13. Un valor de 13 bits en complemento a dos, que debe ser extendido en signo para ser usado como segundo operando en una instrucción aritmética, lógica, de corrimiento, de carga o de almacenamiento cuando $i = 1$.

Campo opf. Este campo contiene el código de operación de una instrucción de punto flotante o de coprocesador.

3.6 La arquitectura SPARC V9

La última actualización de la arquitectura SPARC es la versión nueve, la cual modifica a la versión ocho en los siguientes aspectos: registros, accesos a espacios alternos, formas de almacenamiento de datos en memoria y en el conjunto de instrucciones. En los siguientes párrafos proporcionamos un panorama bastante superficial de SPARC V9.

3.6.1 Registros

Algunos de los registros de control y de estado presentes en la versión ocho han sido eliminados en la nueva versión. En la tabla 3.6 se listan los registros que han sido suprimidos. Además, varios registros han sido extendidos de 32 bits a 64 bits. La tabla 3.7 muestra tales registros. En esta nueva versión de la arquitectura los campos que conforman el registro `PSR` se han convertido en registros individuales, tales registros se muestran en la tabla 3.8. El archivo de registros de punto flotante también ha sido extendido de 32 a 64 registros, lo que permite almacenar 16 datos adicionales de doble precisión, así como 8 datos adicionales de precisión extendida. La versión nueve de la arquitectura especifica que el valor del apuntador a la ventana actual (CWP) se incrementa como resultado de la ejecución de la instrucción `save`, y se decrementa debido a la ejecución de la instrucción `restore`. Este comportamiento es opuesto al que se observa dentro de la arquitectura SPARC V8.

`PSR`	Registro de estado del procesador
`TBR`	Registro base de trampa
`WIM`	Máscara de ventana inválida

Tabla 3.6: Registros de la versión ocho suprimidos en la versión nueve

Registros enteros	
Todos los registros de estado	`FSR`, `Y`, `PC` y `nPC`

Tabla 3.7: Registros extendidos de 32 a 64 bits

`CCR`	Registro de Códigos de Condición
`CWP`	Apuntador a la ventana actual
`PIL`	Nivel de interrupción del proceso
`TBA`	Dirección base de trampa
`TT[MAXTL]`	Tipo de trampa
`VER`	Versión

Tabla 3.8: Registros en SPARC V9 equivalentes a los campos del registro PSR en SPARC V8

3.6.2 Accesos a espacios de dirección alternos

Ciertas instrucciones de carga y almacenamiento alternas pueden ser incluidas en el código de usuario. Si el valor del identificador de espacio de dirección se encuentra en el intervalo $00_{16} \ldots 7F_{16}$, entonces tal instrucción es privilegiada, de otro modo, si tal valor se encuentra en el intervalo $80_{16} \ldots FF_{16}$ la

instrucción es no privilegiada. En SPARC V8, todo acceso a espacios de dirección alternos se encuentra restringido para el modo supervisor del procesador, tal como se mencionó en una sección anterior.

3.6.3 Almacenamiento en memoria

SPARC V9 soporta cualquiera de los modos de almecenamiento big endian y little endian únicamente para accesos a datos. Sin embargo, los accesos a las instrucciones siempre se llevan a cabo empleando el modo de almacenamiento big endian. Hay que recordar que en la versión ocho todo almacenamiento en memoria se realiza conforme al esquema big endian.

3.6.4 Cambios en el conjunto de instrucciones

Las aplicaciones escritas para una implantación de la versión ocho pueden ser ejecutadas sin cambios por un procesador que implante la versión nueve. Sin embargo, SPARC V9 especifica algunos cambios en el conjunto de instrucciones. Las instrucciones de carga de palabras (`ld` y `lda`) en SPARC V8 se convierten en instrucciones de carga de palabras sin signo (`lduw` y `lduwa`), las instrucciones para desplazamiento (`sra`, `srl` y `sll`) se encuentran divididas en versiones para 32 y 64 bits, todas las demás instrucciones aritméticas operan sobre operandos de 64 bits y producen resultados de 64 bits. Han sido añadidas instrucciones para soportar datos de 64 bits. Hay instrucciones para realizar la conversión de valores de punto flotante a valores enteros (`f[sdq]tox`) y de valores enteros a valores de punto flotante (`fxto[sdq]`). SPARC V8 especifica tres instrucciones (`fmovs`, `fabss` y `fnegs`) para realizar movimientos, cálculo de valor absoluto y complemento de valores de punto flotante de simple precisión. SPARC V9 extiende este subconjunto para realizar las mismas operaciones sobre datos de doble precisión y precisión extendida (`fmov[dq]`, `fabs[dq]` y `fneg[dq]`). Se cuenta con instrucciones para realizar la carga y almacenamiento de datos enteros de 64 bits (`ldx`, `ldxa`, `stx` y `stxa`). También se cuenta con instrucciones para realizar carga de palabras con signo (`ldsw` y `ldswa`). Las instrucciones encargadas de realizar lectura y escritura de registros de estado y de control (`rdtbr`, `wrtbr`, `rdwim`, `wrwim`, `rdpsr` y `wrpsr`) en SPARC V8 fueron eliminadas, lo anterior debido a que los registros sobre los que operaban también fueron suprimidos en SPARC V9.

Capítulo 4

Objective C

4.1 Introducción

Smalltalk, desarrollado durante la década de los setenta en el Xerox Palo Alto Research Center, está considerado como el primer lenguaje que utilizó los conceptos de programación orientada a objetos, lo cual no es tan cierto. Las ideas fundamentales sobre las que descansa la programación orientada a objetos provienen del lenguaje SIMULA[1], desarrollado en Noruega en la década de los sesenta con el fin de atacar problemas de simulación y modelado. SIMULA permite a sus usuarios crear sistemas orientados a objetos, utilizando el lenguaje procedural ALGOL para proporcionar soporte para valores numéricos, valores booleanos y estructuras básicas tanto de control como de datos. Debido a esto, los sistemas Smalltalk 72–80 extendieron el enfoque orientado a objetos a un número mayor de elementos de programación. Por ejemplo, en el sistema Smalltalk 72 es posible representar listas, aritmética y estructuras de control como objetos y mensajes, respectivamente. Smalltalk 74 introdujo la idea de las descripciones de clases como objetos. El sistema Smalltalk 80 [Xerox 1981] fue la culminación del proyecto de desarrollo de este importante lenguaje de programación. A partir de Smalltalk aparecieron algunos otros lenguajes de programación puros orientados a objetos, tales como Eiffel y Actor. Pero además, surgió la motivación de añadir características propias de orientación a objetos a los lenguajes tradicionales tales como LISP, C y Pascal.

Una de las herramientas más poderosas incluidas en el ambiente NeXTSTEP es el lenguaje Objective C [NeXT 1993, Garfinkel y Mahoney 1993], un lenguaje de programación orientado a objetos en el cual se encuentran escritas las clases que describen a los objetos dentro del sistema. Este capítulo proporciona una descripción simple de las características más relevantes de este lenguaje.

4.1.1 Los lenguajes de programación orientados a objetos puros e híbridos

Los lenguajes de programación orientada a objetos *puros* se diseñan desde su inicio considerando las características de la orientación a objetos, todos sus elementos son objetos y se describen mediante clases. Por otro lado, un lenguaje *híbrido* se construye tomando como base otro lenguaje existente. El resultado es una extensión a dicho lenguaje que además de soportar clases, objetos, herencia o polimorfismo, mantiene las características iniciales, es decir, sigue siendo procedural y estructurado, además de brindar las construcciones subyacentes tales como estructuras de control, de datos, tipos simples, apuntadores, etc.

El lenguaje Objective C fue creado por Brad Cox y Tom Love alrededor de 1982 y 1983 como un superconjunto del lenguaje C, por lo que es un lenguaje híbrido. Su compilador soporta código fuente

[1] *Simulation Language.*

tanto de Objective C, de C++ como del propio C. El compilador puede discernir entre ambos tipos de código de acuerdo a la extensión del archivo fuente, donde un archivo con extensión ".m" contiene código en Objective C y un archivo con extensión ".c" contiene código en lenguaje C. Es posible emplear Objective C como una extensión de C++, aunque esto pareciera carecer de sentido, pues en principio C++ es en si mismo una extensión de C para soportar programación orientada a objetos. La razón de lo anterior es que C++ carece de algunas de las características con las que cuenta Objective C[2]. También es necesario destacar que C++ proporciona ciertas facilidades no encontradas en Objective C. Un rasgo bastante importante de Objective C es que depende de un *sistema en tiempo de ejecución* necesario para ejecutar el código de un programa, pues muchas decisiones son pospuestas para el tiempo de ejecución en vez de ser tomadas durante la compilación. Las extensiones específicas de Objective C y el modelo de objetos que utiliza se basan precisamente en Smalltalk, y en particular en Smalltalk 76.

4.2 Objetos

Los *objetos* asocian en una sola entidad un conjunto de datos junto con las operaciones específicas que los manipulan y afectan. Tales operaciones son conocidas como *métodos* y los datos se conocen como *variables instancia*. Las variables instancia representan el estado del objeto y son internas a él, por lo tanto, la única forma de conocer su valor es a través de los métodos que para este propósito tenga el objeto. Además, el objeto puede invocar únicamente los métodos que fueron diseñados para él, le es imposible ejecutar accidentalmente métodos asignados a objetos de otro tipo. Se debe hacer notar que el objeto esconde tanto sus variables instancia como la implantación de sus métodos.

Para declarar objetos en Objective C utilizamos un tipo de datos distinto, conocido como `id`. Este tipo se define simplemente como un apuntador a un objeto, más específicamente, es un apuntador a los datos del objeto, el cual es identificado por medio de su dirección. El siguiente es un ejemplo de la declaración de un objeto.

```
id     unObjeto;
```

En las definiciones propias del lenguaje Objective C, tales como valores de regreso en métodos, `id` es el tipo por omisión. La palabra reservada `nil` denota un objeto nulo, es decir, una variable de tipo `id` con valor cero. El tipo `id` no proporciona información referente al objeto al que apunta una variable de este tipo en un determinado momento, solo sabemos que es un objeto, aunque sabemos también que no todos los objetos son de la misma naturaleza.

En algún momento un programa podría necesitar información de alguno de los objetos que contiene, como por ejemplo, qué tipo de variables instancia tiene, qué métodos puede llevar a cabo, etc. Como el tipificador `id` no proporciona dicha información al compilador, cada objeto debe ser capaz de proporcionarla durante el tiempo de ejecución. Esto es posible porque cada objeto contiene una variable instancia llamada `isa`, la cual es empleada para determinar a que clase pertenece el objeto. A través del apuntador `isa` los objetos pueden obtener información y revelarla en tiempo de ejecución. Es posible, por ejemplo, decidir si cuenta con un método particular dentro de su repertorio o bien conocer el nombre de su superclase. Los objetos con el mismo comportamiento y con datos del mismo tipo son miembros de la misma clase. De esta forma los objetos son tipificados dinámicamente durante el tiempo de ejecución, aunque es posible dar al compilador la información referente a la clase a la que pertenece un objeto mediante su *tipificación estática* dentro del código fuente.

4.3 Clases

Un programa orientado a objetos se construye a partir de un conjunto de objetos. Una aplicación basada en el ambiente NeXTSTEP puede utilizar objetos tales como ventanas, listas, matrices, objetos de sonido

[2]Por ejemplo *enlace dinámico* y *tipificación dinámica*.

o de texto, entre muchos otros. Además, dichos programas podrían utilizar más de un objeto de la misma *clase*, por ejemplo, varias ventanas.

En Objective C como en los demás lenguajes orientados a objetos estos están definidos mediante sus clases. La definición de una clase proporciona un prototipo para objetos que siguen un determinado patrón. La clase declara las variables instancia que formarán parte de cada miembro o *instancia* de esa clase, define también el conjunto de métodos que los objetos descritos por la clase en cuestión pueden invocar. Todas las instancias de una clase tienen acceso al mismo conjunto de métodos, por lo que estos se pueden compartir entre las instancias, pero cada objeto tiene sus propias variables instancia.

Para cada clase el compilador crea un objeto especial conocido como *objeto clase*. Este objeto conoce la forma de construir nuevas instancias pertenecientes a la clase que representa, por lo que es conocido también como *objeto fábrica*. El objeto clase es la versión compilada de la clase y construye instancias en tiempo de ejecución para que realicen cierto trabajo dentro del programa.

4.4 Mensajes

Para solicitar a un objeto que ejecute alguna operación es necesario enviarle un mensaje indicándoselo. En Objective C las expresiones para el envío de mensajes tienen la sintaxis que se indica a continuación.

```
[ receptor mensaje ];
```

El receptor es el objeto al cual se le envía el mensaje, el cual se compone del nombre de un método y, opcionalmente, uno o más argumentos que le sean necesarios. Cuando un mensaje es enviado, el sistema en tiempo de ejecución selecciona e invoca al método apropiado dentro del repertorio de métodos del objeto receptor.

Las siguientes expresiones muestran ciertos tipos de mensajes.

```
[ unaMatriz determinante ];
[ unPunto moverHacia: 30.0 : 50.0 ];
```

El primero de ellos invoca a un método que no requiere de parámetros. En el segundo, ciertos argumentos son especificados, en este caso, hacemos referencia al método mediante el identificador `moverHacia::`. Los argumentos se especifican después de cada ':'. Un nombre alternativo válido puede ser `moverHacia:y:`, en cuyo caso el mensaje quedaría de la siguiente manera.

```
[ unPunto moverHacia: 30.0 y: 50.0 ];
```

La sintaxis anterior para las expresiones de envío de mensajes proviene del lenguaje Smalltalk, en donde los mensajes como los anteriores se conocen como *mensajes unitarios* para el primer caso y como *mensajes de palabra clave* para el segundo y tercero [Xerox 1981]. Podemos aplicar la misma definición para el dominio de Objective C y decir que un mensaje unitario no tiene argumentos y que un mensaje de palabra clave involucra por lo menos un argumento.

Así como las funciones ordinarias en C, los métodos pueden devolver valores. Como en el siguiente ejemplo, donde el valor regresado por el mensaje es asignado a una variable entera.

```
int     Etiqueta;
Etiqueta = [ unBoton etiqueta ];
```

Esto permite, además, anidar expresiones para enviar mensajes como en el ejemplo siguiente.

```
int     Etiqueta = [ [ matriz celdaSeleccionada ] etiqueta ];
```

Un mensaje a un objeto nulo es válido sintácticamente, pero no tiene sentido.

```
[ nil moverHacia: 100.0 : 22.5 ];
```

Los métodos definidos para un objeto tienen acceso directo a las variables instancia de ese objeto sin necesidad de que sean especificadas como argumentos. Si alguno de tales métodos requiere del valor de alguna variable perteneciente a otro objeto, el método debe enviarle un mensaje solicitando que proporcione el valor de dicha variable, como lo demuestra el siguiente ejemplo.

```
int     m;
m = [ unObjeto m ];
```

Es claro que la variable externa `m` es distinta a la variable instancia `m` dentro del objeto apuntado por la variable `unObjeto`. De la misma forma, las dos variables anteriores no tienen que ver con el nombre del método `m`.

4.5 Polimorfismo

Un objeto puede invocar únicamente a los métodos que le fueron asignados. No es posible la confusión de alguno de sus métodos con los de algún otro objeto, incluso cuando existan métodos para ambos objetos que tengan el mismo nombre. Lo anterior nos indica que diferentes objetos pueden responder de manera distinta a un mismo mensaje. Esta propiedad es conocida como *polimorfismo* y juega un papel significativo en el diseño de sistemas orientados a objetos. Junto con el enlace dinámico permite escribir código que pueda ser aplicado a cualquier número de objetos distintos, sin necesidad de elegir en el momento de la escritura alguno de ellos de manera específica.

El polimorfismo se da como consecuencia de que cada clase tiene su propio *espacio de nombres*. Los nombres asignados dentro de la definición de una clase no provocan conflictos con los nombres asignados fuera de la clase. Esto se aplica tanto a las variables de las instancias como a sus métodos. El principal beneficio del polimorfismo es la simplificación de la interfaz de programación. Permite que sean establecidas ciertas convenciones que puedan ser usadas nuevamente de clase en clase. En vez de buscar un nombre para cada nueva función que se añada al programa, es posible usar uno empleado con anterioridad.

La *sobrecarga de operadores* es un caso especial de polimorfismo. Se refiere a la habilidad del lenguaje para convertir operadores del lenguaje, por ejemplo '==' ó '+' en C, en nombres de métodos de una clase, de tal modo que tengan un significado particular para cada tipo de objeto. Desafortunadamente, aunque Objective C soporta el polimorfismo en los nombres de los métodos, no permite la sobrecarga de operadores.

4.6 Enlace dinámico

Cuando se realiza el envío de un mensaje en el lenguaje Objective C, el método y el objeto receptor son definidos hasta que el programa se encuentre en ejecución y el mensaje sea enviado. El método exacto que será invocado en respuesta a un mensaje se determina en tiempo de ejecución, no mientras el código se compila.

Además, el método que será ejecutado depende del receptor del mensaje. Como ya se mencionó, el polimorfismo permite que diferentes objetos receptores tengan distintas implantaciones de un método con un nombre determinado. Un compilador tendría que conocer la clase a la que pertenece un objeto receptor para encontrar la implantación correcta a utilizar como respuesta al mensaje. Un objeto es capaz de dar a conocer esta información en tiempo de ejecución mediante la recepción de un mensaje (tipificación dinámica), ya que no está disponible desde la declaración del objeto en el código fuente.

La selección de una de las diferentes implantaciones de un método sucede en tiempo de ejecución. Cuando se envía un mensaje, se verifica si la clase del objeto receptor contiene alguna implantación del método cuyo nombre se especificó en el mensaje. En caso afirmativo, tal método recibe un apuntador a las variables instancia del receptor y es ejecutado. Como el nombre de un método en un mensaje sirve

para seleccionar la implantación adecuada, los nombres de métodos en expresiones de envío de mensajes se conocen como *selectores*.

Este enlace dinámico de métodos con mensajes trabaja de la mano con el polimorfismo para dar a la programación orientada a objetos mucho de su poder y flexibilidad. Como cada objeto puede tener su propia versión de un método, un mismo programa puede comportarse de manera distinta y arrojar resultados diferentes variando solamente los objetos receptores de los mensajes y no los mensajes en si. La selección de los objetos que recibirán los mensajes puede realizarse en tiempo de ejecución y en dependencia de factores externos, como por ejemplo, las acciones de un usuario del programa. Aún más, Objective C va más allá al permitir que el selector del mensaje que se envía sea una variable que se determina en tiempo de ejecución.

4.7 Herencia y la clase Object

La definición de una clase debe estar basada en otra clase ya existente, de la cual la nueva clase hereda los métodos y variables instancia. La nueva clase simplemente extiende o modifica la definición de la clase de la cual hereda. No es necesario duplicar los atributos heredados.

Mediante la herencia se establece una relación jerárquica entre las clases en forma de un árbol, en el cual la clase **Object** se encuentra en la raíz. Todas las clases, excepto la clase **Object**, tienen una *superclase* en un nivel superior en el árbol (de la cual heredan) y cualquier clase, incluyendo la clase **Object**, puede tener una *subclase* en un nivel inferior en el árbol (a la cual heredan). La figura 4.1 muestra un ejemplo de esta jerarquía mediante algunas clases definidas dentro del Application Kit de NeXTSTEP.

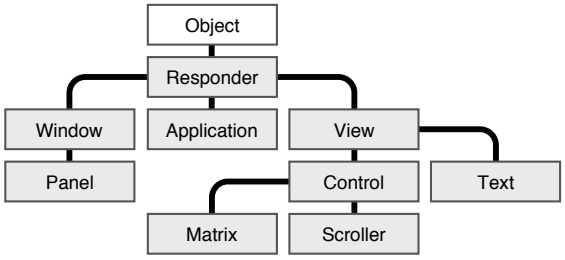


Figura 4.1: Jerarquía de clases dentro del Application Kit de NeXTSTEP

Cada clase, con excepción de **Object**, puede verse como una especialización o adaptación de otra clase, ya que modifica todo lo que acumula a través del mecanismo de herencia. La clase **Object** es la única que no tiene superclase y que se encuentra en la ruta de herencia de todas las demás. Define el marco básico y las formas de interacción entre los objetos del lenguaje. Impone a las clases e instancias que le heredan la capacidad de comportarse como objetos y la posibilidad de cooperar con el sistema en tiempo de ejecución.

Una nueva clase se enlaza a la jerarquía por medio de la especificación de su superclase. Cada clase que se crea debe ser subclase de una clase ya disponible, de lo contrario se definiría como una clase raíz. La implantación de una nueva clase raíz es una tarea arriesgada y delicada, esto debido a que la nueva clase debe realizar las mismas tareas que la clase **Object**, tales como creación de instancias, conexión de estas con sus respectivas clases y su identificación con el sistema en tiempo de ejecución.

Una instancia creada por un objeto clase contiene no solo las variables instancia definidas por su respectiva clase, también contiene las variables instancia definidas para la superclase, las definidas para la superclase de la superclase y así sucesivamente hasta llegar a las de la clase **Object**. Cada objeto

hereda de **Object** la variable `isa`, la cual conecta a una instancia con su clase. Así mismo, un objeto no solo tiene acceso a los métodos asignados a su clase, sino también a los que son asignados a cada clase de las cuales hereda y que pueden ser invocados como consecuencia de la recepción de un mensaje. Los objetos clase también heredan de las clases superiores en la jerarquía, pero como no tienen variables instancia solamente pueden heredar métodos.

Existe una excepción a la herencia. Cuando se define una nueva clase, es posible implantar un método con el mismo selector de alguno definido ya en alguna de las clases de las cuales hereda la nueva clase. Como consecuencia, el nuevo método anula al primero. Las instancias de la clase ejecutarán este método en vez del anterior, lo mismo que las subclases de la nueva clase lo heredarán en lugar de heredar el original. Un método dentro de esta clase puede hacer caso omiso de la redefinición e invocar el método anulado mediante el uso de un mensaje a `super`, como se explicará en una sección posterior. Por último, no es posible anular variables instancia de la misma forma en que los métodos son anulados.

4.8 Clases abstractas

Existen clases diseñadas con el único propósito de servir como superclases de otras clases que las heredan. Estas se conocen como *clases abstractas* y contienen definiciones de variables instancia y de métodos que serán utilizadas por diferentes clases que se agrupan bajo una misma definición. Por si mismas las clases abstractas están incompletas, pero contienen código que es útil para reducir la carga de implantación de las subclases. El mejor ejemplo de esto lo da la clase **Object**. Una instancia de la clase **Object** no serviría para realizar con ella algo productivo, sería como un objeto genérico con la capacidad de hacer nada en particular.

4.9 Creación de instancias

Como ya se mencionó, la principal función de un objeto clase es la construcción de nuevos objetos miembro o instancias de la clase que representa. Hacemos referencia al objeto clase dentro del código fuente mediante el nombre de la clase. De esta manera el siguiente código indica al objeto clase asociado a la clase **Matriz** que fabrique una instancia, cuyo apuntador es asignado a una variable.

```
id     unaMatriz;
unaMatriz = [ Matriz alloc ];
```

La función que realiza el método `alloc` es la de asignar memoria de manera dinámica para las variables instancia del nuevo objeto y asignarles a todas el valor cero, excepto a la variable `isa`, que se hace apuntar a la clase de la instancia. Cuando es necesaria cierta inicialización de las variables de un objeto, podemos utilizar algún método para este propósito y anidar mensajes de la siguiente manera.

```
unaMatriz = [ [ Matriz alloc ] init ];
```

El método de inicialización puede requerir de ciertos argumentos para otra clase específica.

4.10 Definición de clases

Mucho del trabajo necesario en el desarrollo de un proyecto de programación orientada a objetos es la creación del código que definirá nuevos objetos, es decir, la creación de nuevas clases. En el lenguaje de programación Objective C, las clases se definen en dos partes:

Interfaz. Simplemente declara las variables instancia y los métodos de la clase y nombra a la superclase.

Implantación. Es la parte donde realmente se define la clase, ya que contiene el código que especifica las acciones llevadas a cabo por los métodos.

Generalmente se acostumbra especificar dichas partes en archivos separados, aunque esto no es exigencia del compilador. El archivo de interfaz debe estar disponible para cualquiera que pretenda utilizar la clase. Generalmente a estos dos archivos se les nombra de la misma manera que a la clase que definen, pero se les asigna una extensión distinta a cada uno. El archivo de implantación tiene la extensión ".m", mientras que el archivo de interfaz puede tener cualquier otra. Como es un archivo que se incluye en otros archivos, se le asocia la misma extensión que a otros archivos de encabezado, es decir, ".h".

4.10.1 La interfaz

La declaración de la interfaz de una clase comienza con la directiva de compilación `@interface` y termina con la directiva `@end`[3]. El siguiente es un esquema de la construcción de la interfaz.

```
@interface Clase: Superclase
{

 declaración de las variables instancia

}

declaración de los métodos

@end
```

La primera línea declara el nombre de la clase y especifica su superclase, la cual determina la posición de la nueva clase dentro de la jerarquía. Si fueran omitidos el nombre de la superclase y ':' entonces se estaría definiendo una clase raíz que rivalizaría con **Object**. Después, entre los caracteres '{}' se definen las variables instancia, es decir, las estructuras de datos que forman parte de cada objeto perteneciente a la clase. Los métodos asociados a la clase son declarados a continuación. Cada método que puede ser usado fuera de la definición de la clase debe estar declarado en el archivo de interfaz y los métodos que son internos a la implantación de la clase deben ser omitidos en el archivo de interfaz. El nombre de los métodos que pueden ser invocados por el objeto clase[4] está precedido de '+'. Como ejemplo se tiene la siguiente declaración.

```
+ alloc;
```

Los nombres de los métodos que son empleados por las instancias de la clase[5] están precedidos por '-'.

```
- determinante;
```

El lenguaje permite ciertas libertades para la asignación de los nombres de las variables instancia y los métodos. Un método de clase y un método de instancia pueden tener el mismo nombre, aunque es una práctica no usada frecuentemente. Un método y una variable instancia pueden también compartir un nombre, esta situación es muy común puesto que por lo general se requieren métodos cuyo único propósito sea proporcionar el valor de las variables con las que comparten el nombre.

Para declarar el tipo del valor que regresa un método y/o el tipo de cada uno de sus argumentos, se emplea la sintaxis estándar de C para hacer "casting" de un tipo en otro. Los siguientes son ejemplos de lo anterior.

```
- (int) etiqueta;
- ponerEtiqueta: (int) unEntero;
- moverHacia: (float) X : (float) Y;
```

El propósito del archivo de interfaz es dar a conocer la clase ante los demás módulos del programa y a otros programadores. El archivo contiene toda la información necesaria para comenzar a trabajar con la clase, aunque puede ser necesario documentarlo.

[3] Todas las directivas de compilación en el lenguaje Objective C comienzan con @.
[4] Los *métodos de clase*.
[5] Los *métodos de instancia*.

4.10.2 La implantación

La definición de la clase se estructura de manera muy similar a su declaración, ya que comienza con la directiva `@implementation` y termina con `@end`.

```
@implementation Clase: Superclase
{

 declaración de las variables instancia

}

definición de los métodos

@end
```

Cada archivo de implantación debe importar el correspondiente archivo de interfaz. Debido a esto, la implantación no necesita repetir algunas de las declaraciones, de tal modo que con toda seguridad podemos omitir el nombre de la superclase y la declaración de las variables instancia. Esto simplifica el archivo de implantación, concentrando su atención en la definición de los métodos. Como se muestra a continuación.

```
#import ‘‘Clase.h’’

@implementation Clase

definición de los métodos

@end
```

La forma de definir un método es similar a la manera en que definimos una función en lenguaje C estándar, el cuerpo del método se delimita entre ‘{}’. Antes de este bloque, la definición es idéntica a la declaración del método en el archivo de interfaz, excepto por la omisión de ‘;’. Como ejemplo se muestran los siguientes esquemas.

```
+ alloc
{
  .
  .
  .
}

- (int) etiqueta
{
  .
  .
  .
}

- moverHacia: (float) X : (float) Y
{
  .
  .
  .
}
```

4.11 Alcance de las variables instancia

Para asegurar que un objeto pueda esconder su estado, almacenado en las variables instancia, el compilador limita el ámbito de las mismas, es decir, limita su visibilidad dentro del programa. Pero para proporcionar flexibilidad, es posible modificar de manera explícita dicha visibilidad en uno de tres niveles. Dichos niveles se especifican mediante directivas al compilador y se definen como sigue:

`@private.` Las variables instancia con este atributo son accesibles únicamente dentro de la clase que las declara.

`@protected.` Las variables instancia marcadas de este modo son accesibles dentro de la clase que las declara y dentro de las clases que heredan de dicha clase.

`@public.` Las variables instancia son accesibles desde cualquier lugar.

El efecto de una directiva se extiende sobre las variables instancia listadas inmediatamente después y hasta encontrar otra primitiva o bien el fin de la lista. Como ejemplo consideremos las siguientes declaraciones.

```
@interface Trabajador: Object
{
 char     * nombre;
@private
 int      edad;
 char     * evaluacion;
@protected
 id       trabajo;
 float    salario;
@public
 id       jefe;
}
```

Las variables que no se encuentran marcadas tienen el atributo predeterminado `@protected`, tal como `nombre` en la declaración anterior. Todas las variables instancia declaradas por una clase se encuentran dentro del ámbito de la clase, no importando como estén marcadas. Si una clase no tuviera acceso a sus propias variables instancia, estas no servirían. Normalmente una clase tiene acceso a las variables instancia que hereda, debido a que las variables instancia de la superclase tienen el atributo por omisión. El siguiente método puede ser implantado en alguna clase que herede la variable `trabajo`, en vez de ser definido por la misma clase **Trabajador**.

```
- promoverA: nuevaPosicion
{
 id     posicionAnterior = trabajo;
 trabajo = nuevaPosicion;
 return(posicionAnterior);
}
```

Para hacer a una variable instancia visible incluso fuera de la definición de la clase que la declara y de las que la heredan, se emplea la directiva `@public`. De manera normal, para que los módulos externos a una instancia puedan conocer el valor almacenado en una de sus variables, tales módulos deben enviar un mensaje al objeto, solicitándo el valor necesario. Sin embargo, una variable instancia declarada pública puede ser accesada desde cualquier lugar como si fuera un campo dentro de una estructura en lenguaje C. El siguiente ejemplo ilustra tal situación, notar que se hace referencia a la instancia por medio de una variable tipificada estáticamente.

```
Trabajador     * ceo = [ [ Trabajador alloc ] init ];
ceo -> jefe = nil;
```

La declaración de variables públicas impide a un objeto el ocultamiento de su estado. Esto va en contra de los principios fundamentales de la programación orientada a objetos, el encapsulamiento de datos dentro de objetos, donde se encuentran protegidos del exterior.

En la figura 4.2 se ilustra el alcance que tienen las variables instancia de una clase mediante los distintos atributos.

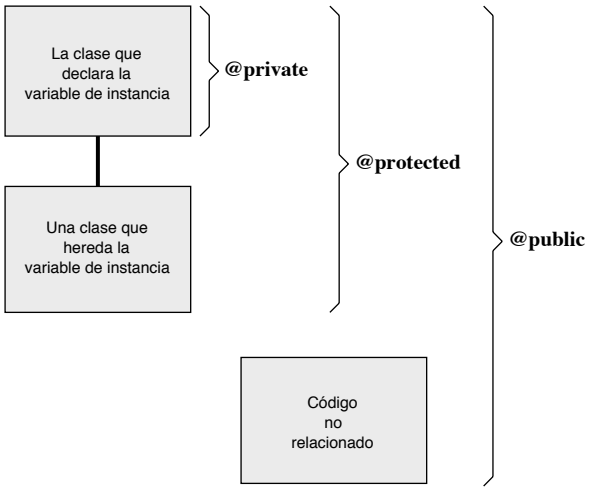


Figura 4.2: Alcance de las variables instancia de una clase

4.12 Detalles sobre el envío de mensajes

La forma en que Objective C logra enlazar en tiempo de ejecución un mensaje con la implantación adecuada de un método es bastante interesante y digna de comentar. Anteriormente se discutió un poco acerca de la sintaxis de las expresiones para enviar mensajes, las cuales, en general, tienen la siguiente forma.

```
[ receptor mensaje ];
```

Para cada expresión de envío de mensajes el compilador genera una llamada a una *función para envío de mensajes*, denominada `objc_msgSend()`. Esta función toma como argumentos el objeto receptor del mensaje, el selector contenido en el mensaje y los parámetros indicados en el mensaje, como se muestra a continuación.

```
objc_msgSend(receptor, selector, argumento 1, argumento 2, ...);
```

Esta función lleva a cabo los pasos necesarios para lograr el enlace dinámico. Tales pasos se listan a continuación:

1. Primero encuentra la implantación del método correspondiente al selector del mensaje. Como un método puede ser implantado de diferentes maneras por distintas clases, el procedimiento exacto que la función debe hallar depende de la clase del objeto receptor.

2. Una vez hallado, se invoca al procedimiento. La función debe proporcionarle un apuntador a las variables instancia del objeto receptor así como los parámetros que hallan sido especificados en la expresión de envío de mensaje.

3. Finalmente, la función regresa el valor devuelto por el procedimiento como si fuera su propio valor de retorno.

La clave para que este mecanismo funcione de manera correcta se encuentra en las estructuras creadas por el compilador para manipular cada clase y objeto. La estructura asociada a una clase incluye los siguientes elementos:

- Un apuntador a la estructura de la superclase.
- Una *tabla de despacho*. Las entradas de esta tabla asocian un selector con la dirección del método específico de la clase al cual identifica.

Como se explicó anteriormente, cuando se realiza la creación de un objeto se asigna memoria para sus variables instancia y estas son inicializadas enseguida. La variable instancia `isa` se hace apuntar a la estructura de la clase a la cual pertenece el objeto. A través de este apuntador, el objeto tiene acceso a la clase y, a través del apuntador en la estructura de la clase, a todas las clases de las cuales hereda. La figura 4.3 ilustra esta construcción para un objeto, su clase, la superclase de su clase y la clase **Object**.

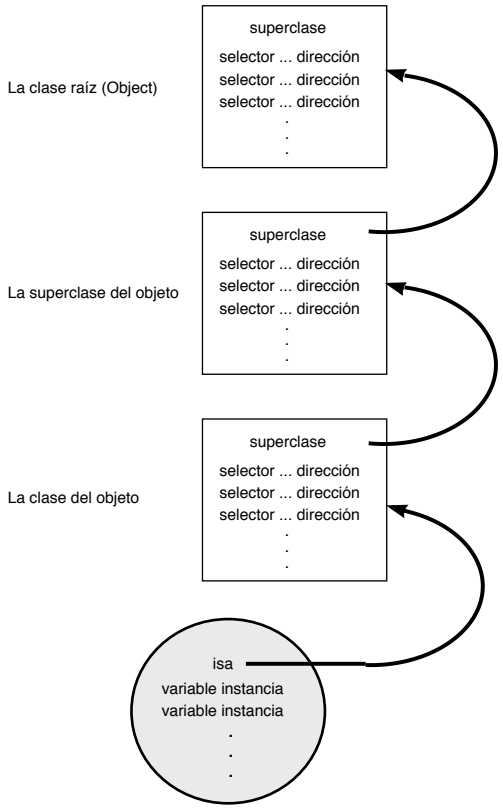


Figura 4.3: Las estructuras de clase para una jerarquía

Cuando se envía un mensaje a un objeto, la función `objc_msgSend()` localiza la estructura asociada a la clase del objeto, a través del apuntador `isa`. Más tarde realiza la búsqueda del selector en la tabla de despacho de tal estructura. Si la búsqueda es infructuosa, obtiene la estructura asociada a la superclase y continúa la búsqueda del selector en la correspondiente tabla de despacho. La función escala en la

jerarquía de clases por cada búsqueda sin éxito en una estructura hasta que logra encontrar la dirección del método apropiado. Este método entra en ejecución después de haber recibido los valores que necesita. Esta es la forma en que las implantaciones de los métodos se eligen en tiempo de ejecución o, en términos más formales, la forma en que los métodos son enlazados dinámicamente a los mensajes.

4.13 Mensajes a `self` y `super`

Como consecuencia de la recepción de un mensaje por parte de un objeto se pone en ejecución el método correspondiente, el cual trabaja sobre las variables instancia del objeto receptor. Este método puede necesitar hacer referencia a tal objeto para enviarle un nuevo mensaje. El lenguaje Objective C especifica dos formas para referirnos al objeto receptor dentro del código de alguno de sus métodos, a través de los términos `self` y `super`.

Supongamos que una clase que define un punto en un plano contiene dos métodos cuyos selectores son `moverHacia::` y `posicionOriginal`. El primero de ellos modifica las variables instancia del objeto receptor, `x` y `y` por ejemplo, con los valores que se le especifican como parámetros. El segundo método inicializa tales variables a cero y puede realizar esto mediante el envío de un mensaje `moverHacia::`, con argumentos cero, al objeto receptor. Tal objeto receptor se denota por `self` dentro del código del método `posicionOriginal`, como a continuación se muestra.

```
- posicionOriginal
{
  .
  .
  .
 [ self moverHacia: 0.0 : 0.0 ];
  .
  .
  .
}
```

El término `self` es como una variable local que puede ser empleada libremente a lo largo de la implantación de un método, así como las variables instancia. El término `super` puede sustituir a `self` únicamente como receptor en una expresión para envío de mensajes. En el ámbito de Smalltalk, estos términos se conocen como *pseudovariables*, es decir, sus valores pueden ser accesados como si fueran una variable, pero no es posible modificarlos mediante una expresión de asignación [Xerox 1981]. Objective C no impone esta última restricción a `self`.

Como receptores de mensajes, los dos términos difieren en la forma en la que cada uno afecta el proceso de envío de mensajes. A continuación se describe tal proceso para cada término:

`self`. Provoca que la búsqueda de la implantación del método solicitado se realice de la manera usual, comenzando en la tabla de despacho contenida en la estructura de la clase a la que pertenece el objeto receptor.

`super`. Hace que la búsqueda inicie en la tabla de despacho perteneciente a la superclase de la clase que define el método donde aparece `super`.

El contenido de este capítulo es únicamente introductorio y contiene información básica para comprender el desarrollo de nuestro proyecto. El lector interesado en profundizar en el tema deberá recurrir a las referencias bibliográficas donde se encuentra extensamente documentado el lenguaje de programación y su sistema en tiempo de ejecución.

Capítulo 5

NeXTSTEP

5.1 Introducción

En 1989 NeXT Computer Corporation[1] liberó los primeros modelos de sus estaciones de trabajo basadas en los procesadores Motorola 680x0. Los diseños de hardware desarrollados por la compañía eran bastante inusuales pues contaban con dispositivos revolucionarios en aquella época como circuitos de procesamiento digital de señales y lectores de discos ópticos. Sin embargo, más inusual aún era la tecnología empleada en el desarrollo del sistema operativo orientado a objetos de tales máquinas y cuyo breve estudio es el objetivo de este capítulo. NeXTSTEP es mucho más que una versión de Unix con una interfaz gráfica excelente y bastante elaborada. En términos más formales, es un ambiente de trabajo y de desarrollo, construido en base al sistema operativo Mach y un conjunto de tecnologías existentes orientadas a objetos [Redman y Silbar 1995, Ibarra y Vergara 1995].

Como NeXTSTEP es Unix, tiene el respaldo de más 30 años de investigación y desarrollo detrás de este sistema operativo. Desafortunadamente NeXTSTEP no llegó a ser tan popular como lo son otros ambientes operativos inferiores. Una computadora ejecutando NeXTSTEP puede ser integrada inmediatamente a una red local o extendida como una máquina Unix más y establecer sesiones de *telnet* o *ftp*, por ejemplo, sin problema alguno. Es importante señalar también que NeXTSTEP es ejecutado tanto por arquitecturas CISC (Motorola 680x0 e Intel 80486/Pentium) como por arquitecturas RISC (HP PA-RISC y SPARC).

Este capítulo proporciona un panorama muy general sobre NeXTSTEP, los elementos que lo conforman y las herramientas que proporciona. El sistema cuenta con una extensa documentación en línea y existen algunas referencias que el lector interesado puede consultar para familiarizarse con el ambiente, [Garfinkel y Mahoney 1993] entre ellas.

5.2 El sistema operativo Mach

Desde la aparición de Unix se han invertido muchas horas en la prueba, mejoramiento y extensión de sus capacidades. Muchas de las características que hicieron a Unix tan popular en el pasado han desaparecido durante la búsqueda de funcionalidad más allá del alcance de su diseño original. Durante tres décadas, el sistema operativo se ha desarrollado mucho. De ser un sistema diseñado para minicomputadoras de 16 bits sin sistema de paginación de memoria y sin soporte para redes, se le ha convertido en un sistema que es ejecutado por sistemas multiprocesador con memoria virtual, memoria caché, etc., además se ha incluido el soporte para redes, tanto de área local como de área extendida. Como resultado de estas

[1] Hoy extinta, desafortunadamente.

extensiones, el núcleo de Unix (originalmente más atractivo debido a su menor tamaño) ha crecido en proporciones considerables.

Mach es un sistema operativo desarrollado en la Universidad de Carnegie Mellon en la segunda mitad de la década de los ochenta y es compatible con la versión 4.3 de BSD. Se encuentra dividido en dos partes, un *micronúcleo* que proporciona las funcionalidades básicas de un sistema operativo y un *ambiente de soporte al sistema* que complementa al micronúcleo para ofrecer todas las capacidades de un verdadero sistema operativo. Mach es un sistema operativo multihilos[2] que puede ser empleado sin dificultad en sistemas de múltiples procesadores y es también adecuado para realizar computación distribuida [NeXT 1992, Silberschatz et al. 1994].

Aunque Mach mantiene compatibilidad con Unix BSD 4.3, se aleja del diseño actual de Unix y regresa a los principios sobre los cuales este se desarrolló. Se mantiene la idea de que el núcleo debe ser lo más compacto posible, conteniendo únicamente un conjunto de funciones primitivas simple pero poderoso que los programadores puden emplear para construir objetos más complejos.

5.2.1 El micronúcleo de Mach

Los diseñadores de Mach lograron minimizar el tamaño del núcleo mediante la eliminación de varios servicios y su relocalización dentro de los procesos del nivel de usuario. El núcleo mismo únicamente contiene los servicios necesarios para implantar un sistema de comunicación entre varios procesos en el nivel de usuario. Un objetivo en el diseño fue hacer de Mach una base para la simulación de otros sistemas operativos, tales simulaciones se implantan dentro de una capa que se ejecuta sobre el núcleo, en el espacio de usuario. De esta manera es posible tener varios sistemas operativos y a sus respectivas aplicaciones conviviendo dentro del mismo sistema.

5.2.1.1 Tareas e hilos en Mach

Mach modifica la concepción de un proceso en Unix[3]. El diseño del micronúcleo divide la definición de proceso en dos entidades, las cuales son descritas a continuación:

Tarea. (Task) Es el ambiente dentro del cual se lleva a cabo la ejecución. También es la unidad básica para la asignación de recursos. Consta de un espacio de direcciones paginado, que tiene como componentes al texto del programa, los datos del mismo y una o incluso más pilas, además contiene una tabla de posibilidades y derechos de acceso a sus puertos. La tarea por si misma no es ejecutable, sino que sirve de marco para la ejecución de los hilos.

Hilo. (Thread) Es la unidad básica de ejecución. Podemos concebir a un hilo como una abstracción del procesador al que representa por medio de su estado[4]. Cada hilo se ejecuta dentro de una sola tarea, aunque tal tarea puede contener otros hilos. Todos los hilos que pertenecen a una misma tarea comparten el espacio de direcciones de memoria virtual y los derechos de comunicación de dicha tarea.

Las tareas que ejecutan múltiples hilos son convenientes cuando una aplicación requiere que varias operaciones se ejecuten concurrentemente y además que estas hagan uso de memoria compartida. Otro aspecto interesante es que los hilos se adaptan perfectamente a las computadoras con arquitecturas multiprocesador. Los hilos pueden ejecutarse individualmente en procesadores separados, incrementando considerablemente el rendimiento de la aplicación.

[2] *Multithreaded.*

[3] Un programa en ejecución que involucra un espacio de direcciones, valores de los registros del procesador, identificadores de archivo, etc.

[4] Valores del contador de programa y de los demás registros.

5.2.1.2 Puertos y mensajes en Mach

En Mach, la comunicación entre los distintos objetos del sistema se lleva a cabo mediante el envío y recepción de mensajes, razón por la cual se ha calificado a Mach como un *sistema operativo orientado a objetos*. El paso de mensajes es el principal medio de comunicación entre las tareas en el sistema y entre las tareas y el núcleo. Los hilos dentro de una tarea solicitan un servicio mediante la emisión de un mensaje a la tarea que proporciona el servicio requerido. Además, todos los hilos dentro de una tarea emplean este mecanismo para comunicarse unos con otros, un hilo puede provocar o suspender la ejecución de otro hilo o de todos los hilos dentro de la tarea mediante la emisión del mensaje apropiado. El sistema de paso de mensajes en Mach se implanta con base en los dos elementos siguientes:

Puerto. (Port) Es un canal protegido de comunicación unidireccional[5] al que los mensajes pueden ser enviados y en donde son almacenados hasta su recepción. Cuando una tarea o un hilo son creados, simultáneamente el núcleo crea también un puerto que los representa.

Mensaje. (Message) Cada mensaje es un flujo de datos que consta de dos partes. Primero, un encabezado de longitud fija que contiene información referente a la longitud del mensaje, su tipo y su destino. Segundo, el cuerpo del mensaje, que es de longitud variable y almacena el contenido del mensaje o bien un apuntador al mismo. Puede contener datos en línea, apuntadores a datos y/o derechos de acceso a puertos.

El puerto es realmente uno de los mecanismos que soportan toda la comunicación en Mach. Es una estructura de datos dentro del núcleo. Cuando un hilo perteneciente a una tarea desea comunicarse con un hilo perteneciente a otra tarea, el primero de ellos (emisor) envía un mensaje a un puerto, el otro hilo (receptor) puede leer entonces dicho mensaje desde el puerto. Tanto las tareas como los hilos se encuentran representados por *puertos en el núcleo*. Los puertos en el núcleo son los argumentos empleados en las llamadas al sistema, ellos permiten al núcleo identificar al hilo o tarea que se verá afectada por la ejecución del llamado.

Como una alternativa a la comunicación mediante transferencia de mensajes, Mach soporta también la comunicación entre procesos mediante memoria compartida. Sin embargo, al hacer uso de memoria compartida, el programador se responsabiliza de la sincronización entre la escritura y lectura de los mensajes. Mediante la transferencia de mensajes Mach por si mismo realiza la planificación del envío y la recepción de los mensajes, asegurándose de que no se produce la lectura de un mensaje que no ha sido enviado completamente.

5.2.2 Comunicación en un ambiente de red

El diseño orientado a objetos de Mach se adapta muy bien a la operación en red. Los mensajes pueden ser enviados entre tareas en diferentes computadoras de la misma forma en que son enviados entre tareas dentro de una misma computadora[6]. La única diferencia es la intervención en forma transparente de un objeto en el nivel de usuario llamado *servidor de la red*.

Los servidores de red actúan como intermediarios entre los mensajes enviados entre tareas en diferentes computadoras. Cada servidor de red implanta *puertos de red* que representan a los puertos para las tareas localizadas en nodos remotos. Un único identificador de puerto de red es usado para distinguir a cada puerto de red.

Un mensaje direccionado a un puerto remoto debe ser enviado al puerto de red que le corresponde. El servidor de la red, una vez que ha tomado de dicho puerto el mensaje lo traduce a una forma que sea compatible con el protocolo de la red y lo transmite al servidor de red localizado en el nodo destino. El servidor de red en este nodo decodifica el mensaje, determina su destino final y lo despacha al puerto al cual fue direccionado.

[5] Implantado mediante una cola de mensajes, cuyo número es finito.
[6] Un nodo de la red con uno o más procesadores.

5.3 ¿Qué es NeXTSTEP?

Desde el punto de vista del usuario, NeXTSTEP es un ambiente operativo gráfico y unificado, que hace a la computadora que lo ejecuta fácil de usar. Destacan los siguientes componentes y aplicaciones:

Workspace Manager. Es la interfaz gráfica de NeXTSTEP al sistema de archivos de Unix. Algunas de sus funciones son la administración de las operaciones sobre archivos y la ejecución de aplicaciones. En la figura 5.1 se ilustra el aspecto de esta aplicación en una sesión con NeXTSTEP.

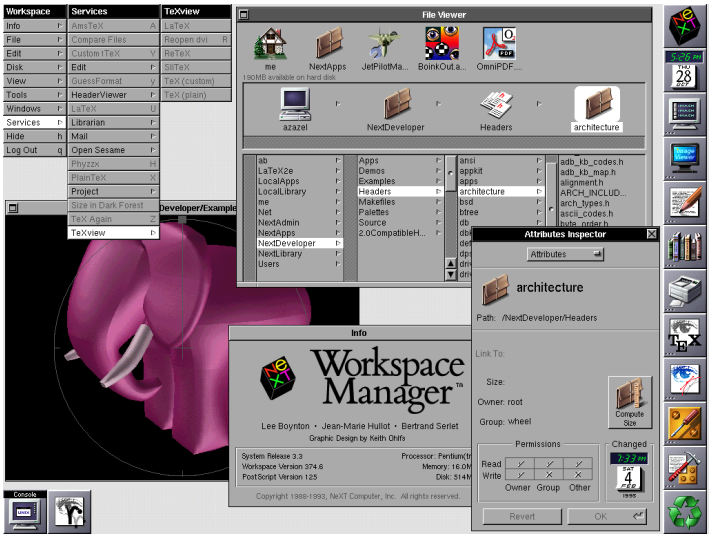


Figura 5.1: Workspace Manager en NeXTSTEP

Window Server. Se encarga de desplegar imágenes en la pantalla, detectar y enviar eventos. Cuenta con un intérprete de *Display PostScript*, una versión del lenguaje PostScript empleado en el diseño de gráficos en dos dimensiones.

Mail, Digital Librarian, Edit. Son algunos programas de aplicación proporcionados por NeXTSTEP.

Desde el punto de vista del programador, NeXTSTEP incluye los siguientes componentes:

Application Kit. (AppKit) Un conjunto de clases escritas en lenguaje Objective C. Se utiliza en cada aplicación para implantar la interfaz de usuario.

Interface Builder y Project Builder. Dos herramientas de desarrollo que permiten al programador construir la interfaz de usuario de una aplicación y mantener el control de los archivos que la componen.

NeXTSTEP Sound y Music Kits. Permiten que la aplicación realice manipulación de sonido, composición, síntesis y ejecución musical.

NeXTSTEP Database Kit. Simplifica el desarrollo de aplicaciones que emplean bases de datos.

C threads package. El paquete de hilos C proporcionado por Mach que hace posible el desarrollo de programas multihilos.

5.4 Window Server

Un componente importante de NeXTSTEP es Window Server, un proceso de bajo nivel que se ejecuta en segundo plano[7]. Las funciones de este proceso son bastante importantes. Primero, es el responsable de canalizar los eventos producidos en el sistema, debido al manejo del ratón o el teclado, a las aplicaciones adecuadas. Segundo, se encarga de desplegar imágenes y texto en la pantalla ya que cuenta con un intérprete del lenguaje PostScript llamado Display PostScript (DPS), el cual se emplea tanto para el monitor como para la impresora, de modo que lo que se muestra en el monitor es exactamente lo que se imprime, salvo que la salida de la impresora luce mejor debido a su mayor resolución.

Mediante Window Server el usuario y el desarrollador quedan aislados de las particularidades del hardware. Por ejemplo, una aplicación puede ser ejecutada por una computadora con un monitor de tamaño diferente al de la máquina en que fue desarrollada, o un teclado ligeramente distinto, o un ratón, etc. PostScript permite también que los programas que emplean colores sean presentados lo mejor posible en cualquier tipo de monitor.

5.5 Application Kit

El Application Kit define un conjunto de clases escritas en Objective C, funciones escritas en lenguaje C así como diversos tipos de datos y constantes. Es bastante amplio pues comprende más de 50 clases. En la figura 5.2 se ilustra la jerarquía de herencia de todas estas clases. El siguiente párrafo proporciona comentarios superficiales referentes a algunas clases dentro del Application Kit. Cada una de estas clases se encuentra descrita en extenso en la documentación en línea del sistema.

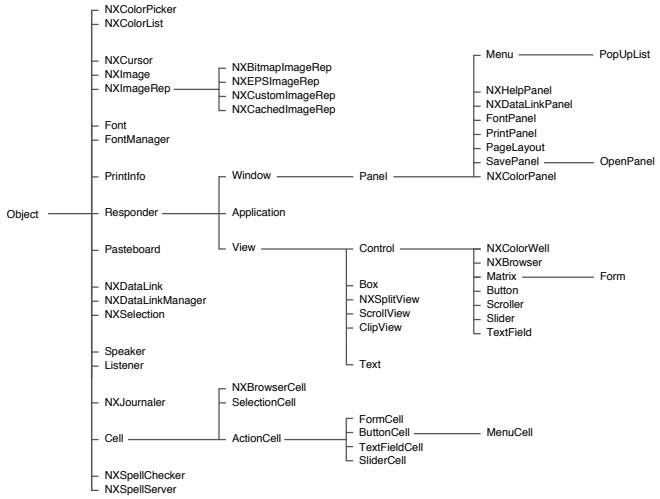


Figura 5.2: La jerarquía de clases definida por Application Kit

La clase central dentro del Application Kit es precisamente **Application**. A cada aplicación le es asignada una única instancia de esta clase, llamada `NXApp`, que se encarga de seguir a las ventanas y menús de la aplicación, controlar el ciclo principal de eventos, abrir archivos creados por Interface Builder, etc. Los objetos definidos por la clase **Window** son áreas rectangulares en la pantalla en las que trabaja el usuario. Las instancias de la clase **View** son áreas dentro de las ventanas en las que la aplicación puede dibujar. **Panel** es una subclase de **Window** empleada para desplegar información transitoria,

[7]Background.

global o urgente. La clase **Responder** define la cadena de respuesta[8], una lista ordenada de objetos que reaccionan a los eventos del usuario. El aspecto del cursor del ratón y el comportamiento de los menús presentados ante el usuario por la aplicación son definidos por las clases **NXCursor**, **Menu** y **MenuCell**. Las clases **Control**, **Cell** y sus subclases definen un conjunto de objetos de fácil identificación, tales como "buttons", "sliders" y "browsers", los cuales pueden ser manipulados gráficamente por el usuario para controlar algunos aspectos de la aplicación. Mediante las clases **Text** y **TextField** las aplicaciones pueden desplegar texto. Para presentaciones de texto sencillas es necesario emplear un pequeño grupo de métodos proporcionados por **Text**, sin embargo, las aplicaciones basadas en texto más complejas pueden explotar toda la potencialidad de esta clase. Para un buen número de aplicaciones puede ser suficiente el empleo de una versión moderada de **Text**, es decir, **TextField**. El Application Kit no suministra clases que definen objetos que hagan referencia a archivos en disco, sin embargo, proporciona las clases **SavePanel** y **OpenPanel** para examinar el sistema de archivos de manera conveniente y familiar. La clase **Pasteboard** define un depósito para datos que son copiados desde una aplicación, dejándolos a disposición de cualquier aplicación que los necesite.

5.6 Project Builder

Project Builder es la parte fundamental del proceso de desarrollo de aplicaciones en NeXTSTEP. Se encarga de administrar los componentes de una aplicación y proporcionar el acceso a las herramientas de desarrollo usadas para crear y modificar estos componentes. Project Builder se encuentra involucrado en cada una de las etapas del proceso de desarrollo, desde suministrar al programador los bloques de construcción básicos para una nueva aplicación hasta la instalación de tal aplicación cuando se encuentre terminada.

La unidad de organización empleada por Project Builder es el *proyecto*. Un proyecto puede ser definido desde dos puntos de vista, el punto de vista conceptual y el punto de vista físico. Conceptualmente, un proyecto está constituido por un conjunto de componentes fuente cuyo objetivo es generar un producto final, como por ejemplo una aplicación, aunque es posible generar otro tipo de productos finales. Físicamente, un proyecto es un directorio que contiene archivos fuente y un archivo de control empleado por Project Builder, denominado **PB.project**. Este archivo registra los componentes del proyecto, el producto final esperado y otra información. Para que un archivo se convierta en parte del proyecto debe residir en el directorio del mismo y debe estar registrado por el correspondiente archivo **PB.project**. El archivo **PB.project** es manipulado mediante la interacción con Project Builder, por ejemplo añadiendo archivos fuente, modificando el nombre del proyecto o el directorio de instalación, etc.

Durante una sesión con Project Builder el desarrollador trabajará con una ventana de proyecto como la mostrada en la figura 5.3, no importando cual sea la naturaleza de tal proyecto.

Project Builder puede ser utilizado para crear y manipular los siguientes tipos de proyectos estándar:

Application. (Aplicación) Un programa de aplicación completamente operativo. Como por ejemplo las aplicaciones Edit y Terminal, entre muchas otras.

Subproject. (Subproyecto) En aplicaciones muy grandes a menudo es conveniente agrupar componentes dentro de subproyectos, los cuales pueden ser construidos en forma independiente del proyecto principal. Project Builder construye los subproyectos cuando sea necesario y emplea sus productos finales en la elaboración del proyecto principal.

Bundle. (Paquete) Es un directorio que contiene recursos que pueden ser utilizados por una o más aplicaciones. Estos recursos pueden ser imágenes, sonidos, archivos nib o código ejecutable.

Palette. (Paleta) Una paleta que puede ser cargada y añadida a la ventana de paletas en Interface Builder que se muestra en la figura 5.4.

[8] *Responder chain.*

Figura 5.3: Ventana de proyecto en Project Builder

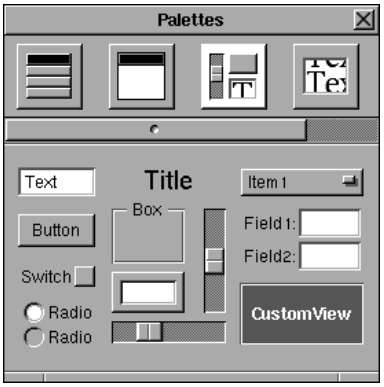


Figura 5.4: Una de las paletas de objetos proporcionadas por Interface Builder

5.7 Interface Builder

Interface Builder es una herramienta auxiliar en el proceso de diseño y construcción de aplicaciones. Acelera el desarrollo de programas permitiendo al programador elaborar la interfaz de la aplicación de una manera gráfica y sin escribir una sola línea de código. Esto es posible debido a que Interface Builder proporciona menús o paletas de objetos definidos por el Application Kit, una de las cuales se muestra en la figura 5.4. Desde esta paleta el programador toma el objeto que requiere y lo coloca en el lugar deseado dentro de la interfaz. Una vez que el objeto ha sido colocado es posible modificar las propiedades del objeto de acuerdo a su clase. Para este propósito Interface Builder cuenta con una ventana llamada *Inspector*, en la cual pueden ser modificados los atributos de un objeto que haya sido seleccionado previamente. Una ventana de este tipo para un objeto de la clase **Button** se muestra en la figura 5.5. Después de que los objetos que componen la aplicación han sido reunidos y editados, Interface Builder permite definir la forma en que estos interactuarán unos con otros.

Interface Builder almacena el trabajo realizado durante una sesión en un *archivo nib*, un archivo con extensión ".nib", que corresponde a "NeXTSTEP Interface Builder". Este archivo contiene versiones de los objetos que conforman la aplicación, datos referentes a las conexiones entre estos objetos y otra

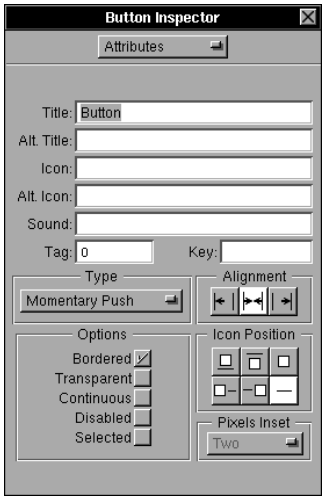


Figura 5.5: La ventana Inspector para un objeto de la clase **Button**

información. Estos archivos se almacenan con un formato binario propietario y no documentado. Sin embargo, todo el manejo de estos archivos lo realiza Interface Builder. Cuando una aplicación comienza su ejecución, obtiene de los archivos nib que la conforman la información necesaria referente a los objetos, sus atributos, conexiones, etc.

El programador tiene acceso a un archvio nib mediante la *ventana de archivo* (*File window*), la cual se muestra en la figura 5.6. Una vez que un archivo nib ha sido abierto por Interface Builder, es posible manejar los objetos y otros recursos de la aplicación contenidos en el archivo nib. Mediante esta ventana Interface Builder puede establecer conexiones entre objetos, cambiar el nombre de los objetos, crear instancias de clases definidas por el programador, examinar y modificar la jerarquía de clases de la aplicación, etc.

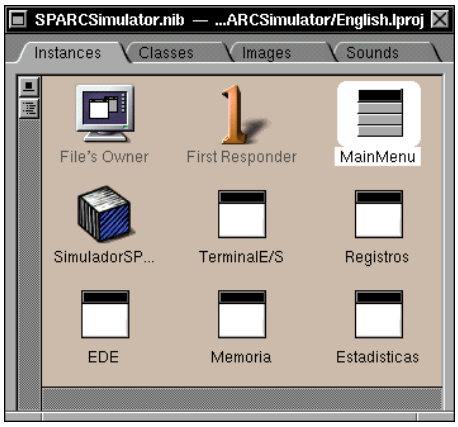


Figura 5.6: Ventana de archivo en una sesión con Interface Builder

Capítulo 6

Las funciones del sistema simulador

6.1 Introducción

Un *simulador de conjunto de instrucciones* (*instruction set simulator*) reproduce la ejecución de un programa simulando, en una plataforma específica, el efecto de cada instrucción (una a la vez) que compone el programa [Magnusson et al. 1998]. Estas instrucciones usualmente están definidas por una arquitectura diferente a la de la plataforma donde se ejecuta el simulador. Este tipo de simulación de una arquitectura permite a los programas correspondientes modelar cualquier computadora, recopilar estadísticas y además ejecutar, bajo ciertas condiciones, todo programa que la arquitectura en cuestión permita, incluyendo el sistema operativo. Además funcionan como soporte (back-end) para otro tipo de programas como depuradores interactivos y herramientas para el diseño de arquitecturas. Este tipo de simulación secuencial de la ejecución de instrucciones es lenta pero permite tener un completo control sobre la ejecución.

Supongamos que se requiere ejecutar un programa en una computadora distinta, tanto en software como en hardware, a la computadora donde fue desarrollado. Existen dos enfoques para solucionar los problemas que se presentan. Se puede realizar una simulación confiable de los llamados al sistema operativo, o bien, permitir la ejecución del mismo sobre el simulador. Este último enfoque involucra un proceso bastante complejo para reproducir todo un ambiente de ejecución, que en los sistemas modernos es muy grande. Esta simulación de máquina completa requiere modelar detalladamente los dispositivos de entrada y salida, de almacenamiento, de comunicación, etc.

El objetivo de este proyecto es construir un sistema simulador del conjunto de instrucciones correspondiente a la arquitectura SPARC V8 [SPARC 1992] que cuente con las siguientes características:

- Ensamble de instrucciones escritas en lenguaje ensamblador SPARC.
- Desensambe de instrucciones binarias.
- Ejecución de instrucciones binarias.
- Presentación al usuario de los recursos del procesador (registros enteros, de punto flotante y de estado) y la memoria.
- Obtención y presentación de información estadística.
- Obtención de datos del exterior mediante ciertos canales de entrada, así como visualización de los resultados del procesamiento por los correspondientes medios de salida.

Algunos requisitos considerados desde el inicio del proyecto son que el sistema sea lo más portable posible y que cuente con una interfaz de usuario adecuada. También es necesario enfatizar los alcances

de este proyecto pues desafortunadamente no contamos con el tiempo suficiente ni con los recursos para desarrollar inmediatamente un simulador completo.

En este capítulo especificamos las funciones del simulador mediante la descripción de las características de las instrucciones definidas por la arquitectura, tales como la forma en que son ensambladas, el modo en que su ejecución modifica el estado del procesador y la memoria principal, etc. Además discutimos las características de la interfaz, las especificaciones de los medios de entrada y salida, los requisitos de la información recolectada y, finalmente, las restricciones del sistema.

6.2 Análisis de instrucciones

En la sección anterior se mencionaron las funciones que el sistema debe llevar a cabo. De entre ellas podemos resaltar aquellas relacionadas con las instrucciones. Para comprender la forma en que el sistema realiza tales operaciones es necesario describir las instrucciones mismas. Tales descripciones son de gran ayuda para la comprensión del proceso de diseño.

6.2.1 Instrucciones de carga

Las instrucciones de carga para valores enteros transfieren un byte, una media palabra, una palabra y una palabra doble desde la memoria al registro indicado por el campo rd. Una vez transferidos un byte o una media palabra a un registro, este es extendido en signo o llenado con cero a la izquierda, dependiendo si la instrucción especifica un valor con o sin signo, respectivamente. Las instrucciones de carga desde un espacio de direcciones alternativo contienen el identificador de espacio de dirección a ser empleado en el campo asi. Cabe mencionar que estas instrucciones son privilegiadas.

Las instrucciones de carga para valores de punto flotante `ldf` y `lddf` se encargan de transferir una palabra o una palabra doble desde la memoria hacia un registro de punto flotante o hacia un par de registros de punto flotante, respectivamente. En las implantaciones existentes, la instrucción `ldfsr` espera a que otras instrucciones de punto flotante pendientes terminen de ser ejecutadas antes que una palabra en memoria sea cargada en el registro de estado de punto flotante `FSR`.

En el capítulo 3 se mencionaron los modos de direccionamiento. Si el bit i es cero, la dirección se calcula mediante la suma del contenido de los registros indicados por los campos rs1 y rs2. Si el bit es uno entonces la dirección se calcula extendiendo el signo del valor en el campo simm13 para formar un valor de 32 bits, el cual se suma al contenido del registro indicado por el campo rs1.

Estas operaciones se realizan de manera atómica. En la tabla 6.1 se muestran los códigos de operación de las instrucciones y en la figura 6.1 los formatos para construirlas.

Formato(3):

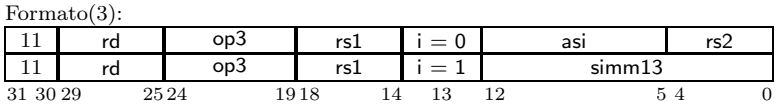


Figura 6.1: Formato de las instrucciones de carga

6.2.2 Instrucciones de almacenamiento

El almacenamiento de valores enteros se lleva a cabo copiando en la memoria el byte menos significativo, la media palabra menos significativa o la palabra en el registro indicado por el campo rd. También es posible copiar una palabra doble desde un par de registros enteros hacia la memoria. Las instrucciones

opcode	op3	Operación
ldsb	001001	Cargar byte con signo
ldsh	001010	Cargar media palabra con signo
ldub	000001	Cargar byte sin signo
lduh	000010	Cargar media palabra sin signo
ld	000000	Cargar palabra
ldd	000011	Cargar palabra doble
ldsba	011001	Cargar byte sin signo desde espacio alternativo
ldsha	011010	Cargar media palabra con signo desde espacio alternativo
lduba	010001	Cargar byte sin signo desde espacio alternativo
lduha	010010	Cargar media palabra sin signo desde espacio alternativo
lda	010000	Cargar palabra desde espacio alternativo
ldda	010011	Cargar palabra doble desde espacio alternativo
ldf	100000	Carga en un registro de punto flotante
lddf	100011	Carga en un registro doble de punto flotante
ldfsr	100001	Carga en registro de estado de punto flotante

Tabla 6.1: Instrucciones de carga

privilegiadas para almacenamiento en un espacio de dirección alternativo toman el identificador de espacio de direcciones necesario del campo asi en la instrucción.

Las instrucciones `stf` y `stdf` realizan la transferencia a memoria de una palabra almacenada en un registro de punto flotante y de una palabra doble contenida en un par de registros, respectivamente. De la misma forma que en el caso de las instrucciones de carga, una implantación de `stfsr` espera a que las instrucciones de punto flotante pendientes terminen de ser ejecutadas antes de almacenar el registro de estado de punto flotante.

La dirección en memoria para almacenamiento se calcula de la misma manera que en el caso de las instrucciones de carga. Lo mismo podemos decir del formato empleado para construir estas instrucciones, mostrado en la figura 6.1. En la tabla 6.2 se muestran los códigos de operación.

opcode	op3	Operación
stb	000101	Almacenar byte
sth	000110	Almacenar media palabra
st	000100	Almacenar palabra
std	000111	Almacenar palabra doble
stba	010101	Almacenar byte en espacio alternativo
stha	010110	Almacenar media palabra en espacio alternativo
sta	010100	Almacenar palabra en espacio alternativo
stda	010111	Almacenar palabra doble en espacio alternativo
stf	100100	Almacenar valor de punto flotante
stdf	100111	Almacenar valor de punto flotante de doble precisión
stfsr	100101	Almacenar registro de estado de punto flotante

Tabla 6.2: Instrucciones de almacenamiento

6.2.3 Instrucciones de intercambio

Las instrucciones `ldstub` y `ldstuba` copian en el registro indicado por el campo rd un byte de la memoria, después escriben en dicha localidad un valor formado únicamente por unos. Las instrucciones `swap` y `swapa` intercambian el contenido del registro indicado por rd con el contenido de una palabra en memoria. Las instrucciones se llevan a cabo de modo atómico, es decir, sin interrupciones.

Se emplean los dos modos de direccionamiento ya mencionados para calcular la dirección de memoria requerida por las operaciones. El formato con el que se codifican estas instrucciones se muestra en la figura 6.1. Las instrucciones privilegiadas `ldstuba` y `swapa` toman del campo asi el identificador de espacio de dirección requerido. En la tabla 6.3 se muestran los códigos de operación.

opcode	op3	**Operación**
`ldstub`	001101	Carga y almacenamiento atómico de un byte
`ldstuba`	011101	Carga y almacenamiento atómico de un byte en espacio alternativo
`swap`	001111	Intercambiar entre registro y memoria
`swapa`	011111	Intercambiar entre registro y espacio de direcciones alternativo

Tabla 6.3: Instrucciones de intercambio entre memoria y registros

6.2.4 Instrucciones lógicas y de corrimiento

Estas instrucciones implantan operaciones lógicas a nivel de bits. El primer operando para la operación se toma desde el registro entero cuya dirección está especificada por el campo rs1. Si el bit i es cero, el segundo operando se toma del archivo de registros enteros, desde el registro indicado por el campo rs2. Si el bit es uno, el segundo operando se construye tomando el valor en el campo simm13 y extendiendo su signo.

La instrucción `sll` desplaza hacia la izquierda el valor contenido en el registro indicado por rs1. `srl` y `sra` desplazan este mismo valor hacia la derecha. `sll` y `srl` colocan ceros en las posiciones vacantes del registro mientras que `sra` llena estas posiciones con el bit más significativo del registro. En todos los casos, el valor del contador se encuentra en los cinco bits menos significativos del registro indicado por rs2 si el bit i es cero, o en los cinco bits menos significativos del valor simm13 dentro de la instrucción si el bit i es uno.

El resultado de la operación es escrito en el registro entero indicado por el campo rd. Como se mencionó en el capítulo 3, las instrucciones cuyo mnemónico tiene el sufijo cc modifican, de acuerdo al resultado de la operación, los valores de los bits que conforman los códigos de condición de la unidad entera del procesador. En la tabla 6.4 se muestran los mnemónicos y códigos de operación de todas estas instrucciones y en la figura 6.2 sus formatos.

Formato(3):

10	rd	op3	rs1	i = 0	no usado(cero)	rs2
10	rd	op3	rs1	i = 1	no usado(cero)	shcnt
10	rd	op3	rs1	i = 1	simm13	

31 30 29 25 24 19 18 14 13 12 5 4 0

Figura 6.2: Formatos para las instrucciones lógicas y de corrimiento

opcode	op3	Operación
and	000001	And
andcc	010001	And y modificación de icc
andn	000101	Nand
andncc	010101	Nand y modificación de icc
or	000010	Or
orcc	010010	Or y modificación de icc
orn	000110	Nor
orncc	010110	Nor y modificación de icc
xor	000011	Or exclusivo
xorcc	010011	Or exclusivo y modificación de icc
xnor	000111	Nor exclusivo
xnorcc	010111	Nor exclusivo y modificación de icc
sll	100101	Desplazamiento lógico hacia la izquierda
srl	100110	Desplazamiento lógico hacia la derecha
sra	100111	Desplazamiento aritmético hacia la derecha

Tabla 6.4: Instrucciones lógicas y de corrimiento

6.2.5 Instrucciones aritméticas

En esta sección describimos, con cierto nivel de detalle, las instrucciones aritméticas definidas por la arquitectura SPARC V8. Para su mejor comprensión están clasificadas en distintas subcategorías.

6.2.5.1 Adición y sustracción

Estas instrucciones realizan sus operaciones sobre dos valores. Los operandos se obtienen de la misma forma que los operandos de las instrucciones lógicas, es decir, se trabaja con dos registros o con un registro y un valor inmediato con signo extendido. El resultado de la operación se deposita en el registro indicado por el valor del campo rd.

Las instrucciones addx y addxcc realizan la suma de los valores antes mencionados junto con el valor del bit c del registro PSR (bit de acarreo). De la misma forma, las instrucciones subx y subxcc restan el valor de dicho bit del resultado de la sustracción de los operandos.

En el capítulo 3 se mencionó un poco sobre aritmética con etiqueta y los tipos de datos asociados, las dos operaciones definidas por la arquitectura son adición (taddcc) y sustracción (tsubcc). Para cada tipo de operación existe una variante que modifica los códigos enteros de condición y que se identifica por el sufijo cc en el mnemónico de la instrucción. En la tabla 6.5 se muestran los códigos de operación y en la figura 6.2 sus formatos.

6.2.5.2 Multiplicación y división

Algunas líneas del capítulo 3 se dedicaron a describir la semántica de estas instrucciones. Los operandos para la multiplicación se toman de dos registros enteros o de un registro y un inmediato con signo extendido. Los 32 bits más significativos del producto se depositan en el registro Y, mientras que los 32 bits menos significativos de este resultado se colocan en el registro indicado por el campo rd. Para el caso de la división, se tiene un dividendo de 64 bits, de los cuales los 32 más significativos se toman del registro Y, los 32 menos significativos se encuentran en el registro indicado por rs1. El divisor es el contenido del registro rs2 o el valor inmediato simm13 con signo extendido. El cociente es un entero de 32 bits que es depositado en el registro rd.

opcode	op3	Descripción
add	000000	Adición
addcc	010000	Adición y modificación de icc
addx	001000	Adición con acarreo
addxcc	011000	Adición con acarreo y modificación de icc
taddcc	100000	Adición con etiqueta y modificación de icc
sub	000100	Sustracción
subcc	010100	Sustracción y modificación de icc
subx	001100	Sustracción con acarreo
subxcc	011100	Sustracción con acarreo y modificación de icc
tsubcc	100001	Sustracción con etiqueta y modificación de icc

Tabla 6.5: Instrucciones aritméticas de adición y sustracción

Como siempre, hay variantes que modifican los códigos de condición y que se distinguen por el sufijo cc. El formato empleado para ensamblar estas instrucciones se muestra en la figura 6.2 y los códigos de operación se muestran en la tabla 6.6.

opcode	op3	Operación
umul	001010	Multiplicación de enteros sin signo
smul	001011	Multiplicación de enteros con signo
umulcc	011010	Multiplicación de enteros sin signo y modificación de icc
smulcc	011011	Multiplicación de enteros con signo y modificación de icc
udiv	001110	División de enteros sin signo
sdiv	001111	División de enteros con signo
udivcc	011110	División de enteros sin signo y modificación de icc
sdivcc	011111	División de enteros con signo y modificación de icc

Tabla 6.6: Instrucciones aritméticas de multiplicación y división

6.2.5.3 Instrucciones para conmutar entre ventanas de registros

Teniendo como preámbulo los comentarios expuestos en el capítulo 3 sobre el archivo de registros enteros y el apuntador a la ventana actual (campo CWP del registro PSR), lo que resta es discutir estas instrucciones con más detalle.

En su forma más simple `save` decrementa el valor del campo CWP, protegiendo la ventana asociada al módulo que realiza la llamada al nuevo procedimiento. La instrucción `restore` realiza una operación inversa, incrementando el valor del apuntador restaura la ventana del procedimiento que realizó la llamada al módulo actual.

En ambos casos, estas instrucciones se comportan como una instrucción `add` normal, excepto que los operandos se toman de la ventana de registros indicada por el valor original del apuntador, y el resultado se deposita en el registro destino dentro de la ventana indicada por el nuevo valor del apuntador. La tabla 6.7 muestra los códigos de operación de estas instrucciones, que se emplean junto con los formatos mostrados en la figura 6.2 para producir las instrucciones binarias.

opcode	op3	Operación
save	111100	Salvar ventana actual
restore	111101	Restaurar ventana anterior

Tabla 6.7: Instrucciones para cambiar ventana de registros

6.2.5.4 La instrucción sethi

Esta instrucción se encarga de colocar ceros en los diez bits menos significativos del registro indicado por rd. Adicionalmente, coloca en los 22 bit más significativos el valor contenido en el campo imm22. Esta instrucción no afecta los códigos enteros de condición.

Una instrucción sethi donde imm22 = 0 y rd = 0 se define como una instrucción nop (no operación), que no altera el estado visible del procesador, lo único que hace es modificar los registros PC y nPC. La tabla 6.8 muestra la información relevante a esta instrucción y la figura 6.3 muestra el formato con el que se construye.

opcode	op	op2	Operación
sethi	00	100	Asigna los 22 bits más significativos

Tabla 6.8: La instrucción sethi

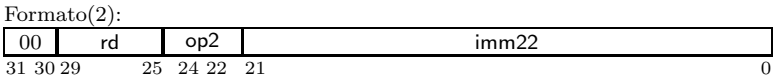


Figura 6.3: Formato de la instrucción sethi

6.2.6 Instrucciones de transferencia de control

En el capítulo 3 se mencionaron las características fundamentales de las instrucciones de transferencia de control. En los siguientes párrafos se describen con un poco más de detalle las categorías en que se pueden clasificar este tipo de instrucciones.

6.2.6.1 Ramificación basada en códigos enteros de condición

Las instrucciones dentro de esta categoría se pueden clasificar adicionalmente en dos grupos, los cuales se describen a continuación:

Saltos incondicionales. La instrucción bn actúa como una instrucción nop cuando el bit a = 0. De otro modo, la siguiente instrucción (la instrucción de retardo) es anulada. En ninguno de estos casos la instrucción provoca una transferencia de control.

La instrucción ba provoca una transferencia de control retardada relativa al registro PC, la cual se lleva a cabo sin considerar el valor de los bits de condición. La dirección de transferencia se calcula mediante la expresión PC + (4 × sign_ext(disp22))[1]. La instrucción de retardo se anula o se ejecuta considerando el valor del bit a.

[1] sign_ext(*arg*) significa que el valor de *arg* se extiende en signo hasta completar 32 bits.

Saltos condicionales. Según sea el valor del campo cond se evalúan los bits que conforman los códigos enteros de condición para determinar si la condición correspondiente se satisface. Como resultado se obtiene un valor "verdadero" o "falso". Si el salto se realiza, se calcula la dirección destino y se lleva a cabo una transferencia de control retardada.

En la tabla 6.9 se proporciona una lista de las instrucciones disponibles. El formato (2a) ilustrado en la figura 6.4 se emplea para construir estas instrucciones.

opcode	cond	Operación	Condición
ba	1000	Siempre salta	1
bn	0000	Nunca salta	0
bne	1001	Salta si es diferente	**not** z
be	0001	Salta si es igual	z
bg	1010	Salta si es mayor	**not** (z **or** (n **xor** v))
ble	0010	Salta si es menor o igual	z **or** (n **xor** v)
bge	1011	Salta si es mayor o igual	**not** (n **xor** v)
bl	0011	Salta si es menor	n **xor** v
bgu	1100	Salta si es mayor sin signo	**not** (c **or** z)
bleu	0100	Salta si es menor o igual sin signo	c **or** z
bcc	1101	Salta si es mayor o igual sin signo	**not** c
bcs	0101	Salta si es menor sin signo	c
bpos	1110	Salta si es positivo	**not** n
bneg	0110	Salta si es negativo	n
bvc	1111	Salta si no hay desbordamiento	**not** v
bvs	0111	Salta si hay desbordamiento	v

Tabla 6.9: Instrucciones de ramificación para códigos enteros de condición

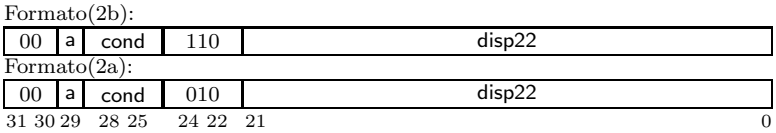


Figura 6.4: Formato para las instrucciones de ramificación

6.2.6.2 Ramificación basada en códigos de condición de punto flotante

Para estas instrucciones se aplican los mismos criterios descritos en los párrafos anteriores. Las transferencias pueden ser condicionales o incondicionales. En este caso, cada instrucción verifica que los códigos de condición de punto flotante cumplan la condición correspondiente. Todas provocan una transferencia de control retardada, anulan o no la siguiente instrucción de acuerdo al valor del bit a y calculan la dirección destino del mismo modo que las anteriores.

En la tabla 6.10 se listan los mnemónicos de las instrucciones junto con sus códigos de operación. El formato (2b) en la figura 6.4 se emplea para construir estas instrucciones.

opcode	cond	Operación	Condición
fba	1000	Siempre salta	1
fbn	0000	Nunca salta	0
fbu	0111	Salta si no hay orden	U
fbg	0110	Salta si es mayor	G
fbug	0101	Salta si no hay orden o si es mayor	G **or** U
fbl	0100	Salta si es menor	L
fbul	0011	Salta si no hay orden o si es menor	L **or** U
fblg	0010	Salta si es menor o si es mayor	L **or** G
fbne	0001	Salta si no es igual	L **or** G **or** U
fbe	1001	Salta si es igual	E
fbue	1010	Salta si no hay orden o es igual	E **or** U
fbge	1011	Salta si es mayor o igual	E **or** G
fbuge	1100	Salta si no hay orden o si es menor o si es mayor	L **or** G **or** U
fble	1101	Salta si es menor o igual	E **or** L
fbule	1110	Salta si no hay orden o si es menor o si es igual	E **or** L **or** U
fbo	1111	Salta si hay orden	E **or** L **or** G

Tabla 6.10: Instrucciones de ramificación para códigos de condición de punto flotante

6.2.6.3 Las instrucciones `call` y `jmpl`

La ejecución de la instrucción `call` provoca una transferencia de control incondicional y retardada. La dirección destino de tal transferencia es relativa al registro `PC` y se calcula mediante la expresión $\texttt{PC} + (4 \times \texttt{sign\_ext}(\text{disp30}))$. Como el desplazamiento disp30 es un valor de 30 bits, la dirección destino puede encontrarse arbitrariamente lejos. Durante la ejecución, `call` almacena el valor del registro `PC` (la dirección de la instrucción `call` misma) en el registro `%r15` (`%o7`). Los datos mostrados en la tabla 6.11 y el formato mostrado en la figura 6.5 contienen toda la información requerida para generar esta instrucción.

opcode	op	Operación
call	01	Llamado a procedimiento

Tabla 6.11: La instrucción `call`

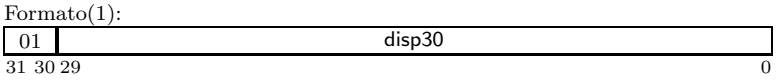


Figura 6.5: Formato de la instrucción `call`

La última instrucción de transferencia de control a considerar es `jmpl`. Esta obtiene la dirección destino de la misma forma en que lo hacen las instrucciones de carga y almacenamiento. Si el bit i es cero entonces la dirección se calcula sumando el contenido de dos registros enteros, los cuales están indicados por los valores de los campos rs1 y rs2. Si el bit i es uno la dirección viene dada como la suma de un registro y un inmediato de 13 bits con signo extendido, indicados por los campos rs1 y simm13, respectivamente. Adicionalmente esta instrucción copia el valor del registro `PC` (el cual contiene

su dirección) en el registro entero indicado por el campo rd. Para construir instrucciones de este tipo se emplean los formatos mostrados en la figura 6.2 empleando el código de operación ilustrado en la tabla 6.12.

opcode	op3	Operación
jmpl	111000	Salto

Tabla 6.12: La instrucción `jmpl`

6.2.7 Instrucciones de punto flotante

Dentro de esta categoría encontramos instrucciones que modifican los bits de condición dentro del registro FSR e instrucciones que no los modifican. Ambos tipos de instrucciones se construyen de modo ligeramente distinto. Se emplean dos variantes del mismo formato, las cuales se muestran en la figura 6.6. Estas versiones difieren únicamente en el valor del campo op3. Este tiene el valor 110100 (FPop1) cuando se trata de una instrucción que no modifica los códigos de condición y el valor 110101 (FPop2) en el otro caso, tal como se muestra en la tabla 6.13.

opcode	op3	Operación
FPop1	110100	Operación de punto flotante
FPop2	110101	Operación de punto flotante

Tabla 6.13: Códigos de operación para instrucciones de punto flotante

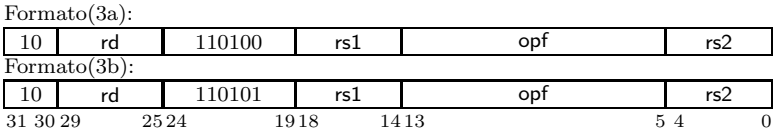


Figura 6.6: Dos formatos diferentes para instrucciones de punto flotante

6.2.7.1 Instrucciones de conversión

La arquitectura SPARC define estas instrucciones debido a que no es posible emplear operandos de diferente tipo en una misma instrucción aritmética de punto flotante. De todas las instrucciones definidas se consideraron para el proyecto de simulación únicamente las que se encuentran listadas en la tabla 6.14. En los comentarios siguientes, toda mención de registros hace referencia a registros en el archivo de punto flotante.

Las instrucciones `fitos` y `fitod` transforman un entero de 32 bits almacenado en el registro indicado por el campo rs2 en un valor de simple precisión y en un valor de doble precisión, respectivamente. El resultado es almacenado a partir del registro indicado por rd. Las instrucciones `fstoi` y `fdtoi` convierten el valor en punto flotante contenido a partir del registro rs2 en un entero de 32 bits, el cual es depositado en el registro indicado por rd. De manera análoga, las instrucciones `fstod` y `fdtos` realizan la conversión de un operando en punto flotante de precisión dada en el valor correspondiente en una precisión distinta.

opcode	opf	Operación
fitos	011000100	Convertir entero a simple precisión
fitod	011001000	Convertir entero a doble precisión
fstoi	011010001	Convertir simple precisión a entero
fdtoi	011010010	Convertir doble precisión a entero
fstod	011001001	Convertir simple precisión a doble precisión
fdtos	011000110	Convertir doble precisión a simple precisión

Tabla 6.14: Instrucciones de conversión entre formatos

Ninguna de estas instrucciones modifica los códigos de condición. Por lo tanto, para construirlas se emplea el formato (3a) de la figura 6.6 con el campo rs1 puesto a cero y utilizando la información contenida en la tabla 6.14.

6.2.7.2 Instrucciones de movimiento

Ninguna de estas instrucciones modifica los bits de condición del registro FSR. Se ensamblan utilizando también el formato (3a) de la figura 6.6 junto con los datos mostrados en la tabla 6.15. Son las siguientes:

fmovs. Copia el contenido del registro rs2 al registro rd.

fnegs. Copia el contenido del registro rs2 al registro rd con el bit de signo complementado.

fabss. Copia el contenido del registro rs2 al registro rd con el bit de signo en cero.

opcode	opf	Operación
fmovs	000000001	Mover
fnegs	000000101	Complementar
fabss	000001001	Valor absoluto

Tabla 6.15: Instrucciones de movimiento para valores de punto flotante

6.2.7.3 Cálculo de raíz cuadrada

Estas instrucciones generan la raíz cuadrada de su operando, el cual es un valor de punto flotante contenido en el archivo de registros a partir del registro indicado por el valor del campo rs2. El resultado se deposita en el archivo de registros a partir del registro indicado por el campo rd dentro de la instrucción.

Del mismo modo que las anteriores, estas instrucciones no alteran el valor de los bits de condición. En la tabla 6.16 y en la figura 6.6 se muestra la información relevante para construirlas.

opcode	opf	Operación
fsqrts	000101001	Raíz cuadrada en simple precisión
fsqrtd	000101010	Raíz cuadrada en doble precisión

Tabla 6.16: Instrucciones para cálculo de raíz cuadrada

6.2.7.4 Instrucciones de comparación

En el capítulo 3 proporcionamos una descripción de los valores que los bits de condición de punto flotante pueden tomar en función de la relación guardada por dos valores de punto flotante. Las instrucciones que determinan tal relación y modifican los bits 11 y 10 del registro `FSR` son las instrucciones de comparación de valores de punto flotante. Sus parámetros son dos datos de simple precisión o dos valores de doble precisión, almacenados en el archivo de registros e indicados por los campos rs1 y rs2 dentro de la instrucción.

En este caso se emplea el formato (3b) mostrado en la figura 6.6 junto con los códigos de operación ilustrados en la tabla 6.17 para el ensamble.

opcode	opf	**Operación**
`fcmps`	001010001	Comparación de datos de simple precisión
`fcmpd`	001010010	Comparación de datos de doble precisión

Tabla 6.17: Instrucciones de comparación

6.2.7.5 Instrucciones aritméticas

Mediante estas instrucciones es posible efectuar operaciones de suma, resta, multiplicación y división sobre valores de punto flotante. Todas operan sobre un par de valores de la misma precisión, los cuales se encuentran almacenados en el archivo de registros a partir de las direcciones indicadas por los campos rs1 y rs2. En su mayoría producen un resultado de la misma precisión de los operandos, a excepción de la instrucción `fsmuld`, la cual opera sobre dos datos de simple precisión y almacena un valor de doble precisión.

Estas instrucciones no afectan el valor de los bits de condición. Se ensamblan usando el formato (3a) de la figura 6.6 y los valores para el campo opf mostrados en la tabla 6.18.

opcode	opf	**Operación**
`fadds`	001000001	Suma en simple precisión
`faddd`	001000010	Suma en doble precisión
`fsubs`	001000101	Sustracción en simple precisión
`fsubd`	001000110	Sustracción en doble precisión
`fmuls`	001001001	Multiplicación en simple precisión
`fmuld`	001001010	Multiplicación en doble precisión
`fsmuld`	001101001	Multiplicación en simple precisión a doble precisión
`fdivs`	001001101	División en simple precisión
`fdivd`	001001110	División en doble precisión

Tabla 6.18: Instrucciones aritméticas de punto flotante

6.3 Recursos del sistema simulador

Para reproducir de la mejor manera la ejecución de las instrucciones es necesario proporcionar un entorno de ejecución apropiado. En las estaciones de trabajo actuales este ambiente incluye al procesador, unidades de manejo de memoria virtual, memoria principal, controladores de dispositivos de entrada y salida, adaptadores para soporte de redes, etc [Stallings 1996].

En nuestro caso, la extensión del ambiente de ejecución se reduce a la memoria principal y al procesador. Es necesario considerar los recursos que componen las unidades constituyentes del procesador, es decir, el archivo de registros enteros, los registros de estado y de control dentro de la unidad entera, el archivo de registros de punto flotante y su registro de estado correspondiente. Nuestros modelos de tales recursos deben contemplar también su comportamiento. Para el caso de la memoria, debemos considerar el modo de almacenamiento big endian durante las transferencias hacia y desde la memoria. Además de contar con una representación adecuada del archivo de registros, debemos proporcionar algún medio para garantizar que las lecturas y escrituras a tal archivo se realicen en la ventana adecuada, considerando el valor del apuntador actual y el hecho de que ventanas adyacentes se encuentran solapadas.

Un requerimiento que debe satisfacer el sistema es que la simulación de la ejecución de las instrucciones afecte a los recursos de la manera correcta. Como primer ejemplo, la simulación de una instrucción de transferencia de control debe modificar el valor de los "registros" `PC` y `nPC` de tal manera que sea posible reproducir el retardo en una instrucción de una transferencia de control. Como siguiente ejemplo, el cambio a la siguiente ventana de registros o a la anterior debe contemplar la organización circular del conjunto de registros.

6.4 Información estadística

Recordemos una de las motivaciones principales de este proyecto, proporcionar a los programadores y desarrolladores de software de sistemas una herramienta útil en la toma de decisiones. El sistema deberá obtener y presentar información sobre el número de instrucciones, por categoría, que fueron ejecutadas por un programa[2]. Para demostrar la utilidad de esta herramienta, mencionamos dos situaciones en donde su uso sería de gran ayuda. Primero, si un desarrollador de compiladores tiene que decidir entre diferentes métodos de generación de código, los datos presentados por el simulador pueden ayudarle a decidir cuál de dichas técnicas produce el resultado más eficiente. Segundo, un programador inexperto puede determinar si el código de su programa es eficiente o no, y en tal caso mejorarlo hasta estar conforme con un resultado que produzca el menor número de instrucciones ejecutadas, lo que se traduce en ahorro de ciclos del procesador y de tiempo de ejecución.

6.5 Requisitos de la interfaz con el usuario

La interfaz hacia el usuario es un componente que hemos considerado importante debido a los propósitos académicos de la aplicación. Existen diversos enfoques para el diseño de este tipo de elementos. Una versión preliminar del sistema simulador SPARC cuenta con una interfaz mediante comandos de línea. Claramente una interfaz de esta naturaleza es bastante inconveniente pues es necesario tener siempre en mente los comandos y sus argumentos. Una alternativa mejor consiste en desarrollar una interfaz gráfica que además de ser agradable permita el fácil acceso a los recursos, herramientas y posibilidades del sistema. En este punto es necesario recordar que se cuenta con las herramientas proporcionadas por NeXTSTEP (como Interface Builder) para llevar a cabo la labor de construcción de la interfaz de modo sencillo y rápido.

La interfaz gráfica debe permitir al usuario llevar a cabo las siguientes operaciones:

Interacción con los registros. La interfaz debe permitir consultar y/o modificar los valores contenidos en los registros enteros de propósito general y en los registros de punto flotante. También debe ser posible consultar los registros de estado, con posibilidad de alterar los valores de algunos de ellos.

Interacción con la memoria. La interfaz debe proporcionar al usuario la facilidad de visualizar el contenido de la memoria e introducir datos en la misma.

[2] *Execution profiler.*

Manipulación de instrucciones. A través de la interfaz gráfica el usuario podrá invocar los servicios de ensamble y desensamble de instrucciones y ordenar la ejecución de un programa.

Servicios de entrada y salida. Los valores, almacenados en los registros o en la memoria, que el usuario puede consultar y/o modificar deberán ser valores enteros expresados en alguna de las bases 2, 10 ó 16. Por lo anterior, el usuario en condiciones normales tendría que interpretar estos datos si es que realmente no representan valores numéricos. Para solucionar este problema se debe concebir un mecanismo que permita al usuario ingresar datos de diversos tipos (enteros, punto flotante, cadena de caracteres) desde una "consola" y desplegar datos en memoria en el formato adecuado hacia la misma. Además es conveniente tener acceso a medios de almacenamiento secundario, es decir, al sistema de archivos.

Consulta de estadísticas. Por medio de la interfaz el sistema simulador debe dar a conocer al usuario los valores estadísticos recolectados durante la última ejecución de algún programa.

6.6 Requerimientos de entrada y salida

En un ambiente de ejecución real el código dependiente de la máquina presente en el núcleo de un sistema operativo interactúa con los dispositivos de entrada y salida presentes en el sistema. Este tipo de ambientes de ejecución han sido simulados completamente de manera exitosa como resultado de diversos proyectos de investigación [Bedichek 1990, Magnusson et al. 1998, Witchel y Rosenblum 1996]. En estos trabajos se ha puesto especial atención en la simulación de los dispositivos de entrada y salida. Los investigadores han procurado realizar simulaciones apropiadas y detalladas de tales dispositivos, con la finalidad de mejorar el rendimiento general del simulador.

Como se describió con anterioridad, es requisito que el sistema permita establecer una comunicación adecuada entre un programa y el usuario. Los medios para establecer tal comunicación dentro de nuestro simulador deberán ser simples en su utilización, de tal modo que un desarrollador pueda solicitar sin ningún problema los servicios de entrada y salida para alimentar a un programa con datos, recibir resultados del mismo y almacenar información en archivos. Este mecanismo de entrada y salida debe llevarse a cabo mediante un enfoque que no involucre reproducir controladores de dispositivos, como en el caso anterior.

6.7 Restricciones

El sistema simulador desarrollado tiene algunas limitaciones con respecto a la definición completa de la arquitectura SPARC V8 y alguna implantación correspondiente. La primera restricción tiene que ver con las *trampas* (*traps*), que son transferencias de control que se generan debido a condiciones anómalas que ocurren durante la ejecución de alguna instrucción [SPARC 1992]. Las trampas provocan que el procesador ejecute una secuencia especial de instrucciones conocida como *manejador de trampa*, estos manejadores son suministrados por el sistema operativo. Algunas de las condiciones que provocan trampas son las siguientes: alineación incorrecta de una dirección en una instrucción de carga o almacenamiento en relación al tipo de dato, ejecución de una instrucción de punto flotante cuando la unidad correspondiente se encuentra deshabilitada, división entera por cero, intento de ejecución de una instrucción privilegiada en modo usuario, por efecto de la ejecución de las instrucciones `save` y `restore`, etc. Existen instrucciones definidas por la arquitectura que provocan la ejecución de manejadores de trampas dependiendo del valor de los códigos enteros de condición. Estas instrucciones no son reconocidas por nuestro simulador y en general las trampas no son soportadas.

Para calcular la dirección de un manejador de trampa se emplea un registro de estado denominado

`TBR`[3]. Como nuestra simulación no permite trampas, no es necesaria la presencia de este registro. De modo similar, las instrucciones `save` y `restore` hacen uso del registro `WIM`[4] para determinar si se genera o no una trampa. Nuevamente, para la simulación este registro es innecesario.

No fueron tomados en cuenta para construir el sistema los siguientes componentes de los procesadores que implantan la arquitectura SPARC: ruta de datos segmentada (pipeline), unidad de manejo de memoria (MMU), cachés de datos e instrucciones y buffer de traducción anticipada (TLB). Como el objetivo no es proporcionar un ambiente para la ejecución del sistema operativo a corto plazo, no se realizó la simulación de dispositivos de entrada y salida tales como adaptadores para red, controladores de disco, etc.

Algunos campos de los registros `PSR` y `FSR` también se encuentran descartados por no ser relevantes para el proceso de simulación. Finalmente, aunque la arquitectura define tipos de datos de punto flotante de 128 bits y las instrucciones correspondientes para efectuar operaciones aritméticas, el simulador no soporta estos tipos de datos y no reconoce tales instrucciones.

Para conveniencia de los programadores los ensambladores proporcionan *instrucciones sintéticas*, las cuales son mapeadas en instrucciones definidas por la arquitectura SPARC V8 [SPARC 1992]. Por ejemplo, la instrucción `mov %g6, %l5`, que transfiere el contenido del registro `%g6` al registro `%l5`, se ensambla de la misma forma que la instrucción `or %g0, %g6, %l5`. La última restricción es que el sistema simulador no sporta instrucciones sintéticas.

[3] Trap Base Register.
[4] Window Invalid Mask.

Capítulo 7

Diseño e implantación del sistema

7.1 Introducción

En el capítulo anterior se establecieron las bases para el diseño del simulador. Una vez que han quedado claros los objetivos del desarrollo, las operaciones requeridas y las limitaciones impuestas, procedemos a definir la estructura del programa de manera clara y datallada.

Inicialmente se plantea un problema de simulación que se encuentra compuesto de tres subproblemas: ensamble de instrucciones en lenguaje ensamblador, desensamble de instrucciones binarias y ejecución de las mismas. Después de realizar las etapas de diseño e implantación obtenemos una solución de software para el problema. Mediante el proceso de diseño obtenemos un modelo del sistema lo bastante general para su inmediata implantación. En la etapa de implantación se emplea un lenguaje de programación junto con algunas herramientas de desarrollo para materializar el modelo anterior. El proceso de solución se ilustra en la figura 7.1.

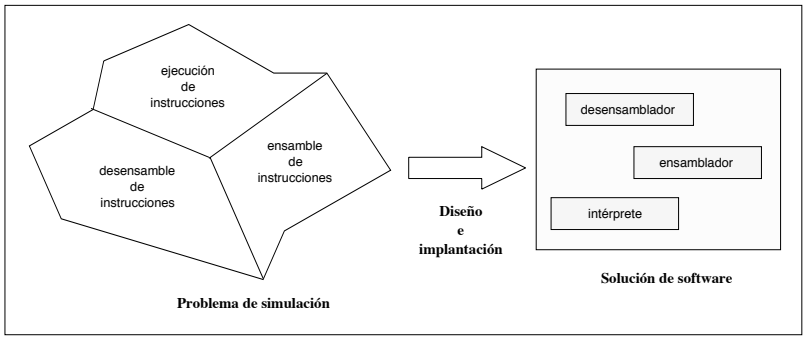


Figura 7.1: El proceso de solución al problema de simulación

Un factor importante que influyó en la estructuración definitiva del sistema fue el requisito de portabilidad. En consecuencia se desarrolló un producto cuyos componentes pueden ser, en un alto porcentaje, reutilizados para construir el mismo sistema en otras plataformas.

En las secciones que forman parte de este capítulo describimos la estructura general del sistema, las operaciones que este lleva a cabo para prestar sus servicios, las estructuras de datos y los módulos que componen la aplicación. También se explica el uso de NeXTSTEP para desarrollar la implantación.

7.2 La arquitectura del sistema simulador

Uno de los objetivos centrales de este proyecto es que el sistema se encuentre disponible a una gran variedad de usuarios. Los usuarios potenciales pueden estar familiarizados con NeXTSTEP o con otro ambiente operativo, pueden emplear computadoras personales o estaciones de trabajo. La gran diversidad en plataformas de hardware y sistemas operativos existentes provoca que la portabilidad sea un factor decisivo para cumplir nuestro objetivo. Un poco de análisis en este punto nos muestra la imposibilidad de transportar la totalidad del sistema hacia un ambiente distinto. Lo anterior debido a que las características de la interfaz son completamente dependientes de la tecnología orientada a objetos proporcionada por NeXTSTEP, tecnología no presente en otros ambientes operativos.

No es posible garantizar la portabilidad simplemente transportando todo el código fuente del sistema de una plataforma a otra y tratando de compilarlo. Para acercarnos a nuestro objetivo optamos por utilizar un principio de diseño modular. Mediante tal principio obtuvimos un sistema completamente funcional, cuyos componentes pueden ser clasificados entre portables y no portables. Establecimos que la estructura del programa se encuentre dividida en dos partes importantes, el *núcleo* y la *interfaz*. Esta división se realiza tomando en cuenta un modelo de capas, tal como lo muestra la figura 7.2.

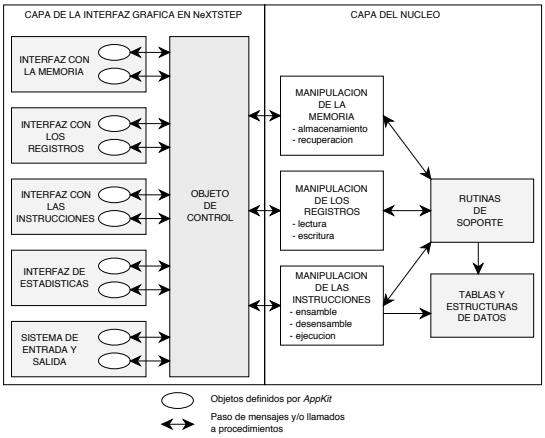


Figura 7.2: La arquitectura en capas del simulador

Mediante este modelo el usuario únicamente observa la capa superior, la interfaz, que es la capa a través de la cual envía sus solicitudes y mediante la cual recibe los resultados de dichas peticiones. Para poder proporcionar las respuestas necesarias a los usuarios, la capa de interfaz se vale de la capa inferior, el núcleo, dentro de la cual se realiza el procesamiento que el usuario ha solicitado. A continuación se describen las características de ambas capas:

- La capa del núcleo está concebida como un conjunto de módulos que permiten realizar las operaciones relacionadas con la simulación de la ejecución de las instrucciones (que involucran carga o almacenamiento de datos, procesamiento aritmético y lógico, etc.), manejo de los recursos como registros o memoria, además de proporcionar los medios necesarios para ofrecer servicios como ensamble y desensamble de instrucciones. Los módulos contenidos en esta capa deben ser portables de manera inmediata, pues se encuentran escritos en lenguaje C. La capa del núcleo por si sola no es funcional, necesita de un medio de control que invoque a sus componentes a petición del usuario.

- La capa de interfaz está dedicada a la interacción con el usuario. Permite ordenar la modificación de los recursos, la ejecución de los programas, el ensamble y/o desensamble de instrucciones, etc.

Como todas las aplicaciones dentro de NeXTSTEP, la interfaz está constituida por un conjunto de objetos controlados por otro objeto "maestro" y todos son capaces de responder a los mensajes que les sean enviados. Obviamente, la naturaleza de NeXTSTEP obliga a la adopción de un modelo orientado a objetos para llevar a cabo la implantación de esta capa. Los detalles de la implantación de estos objetos provocan que la implantación de la interfaz no pueda transportarse a un ambiente distinto.

7.3 La capa del núcleo

7.3.1 Estructura de la capa del núcleo

La capa del núcleo está constituida como un conjunto de módulos, los cuales están organizados en grupos, tal como se ilustra en la figura 7.2. Además cuenta con ciertas estructuras de datos y las funciones necesarias para manipularlas. Por último, contiene algunas funciones de soporte que brindan varios servicios. Cada grupo de módulos está dedicado a proporcionar los medios necesarios para realizar actividades específicas. Por ejemplo, el manejo de la memoria, las operaciones con los registros, el ensamble, desensamble y ejecución de instrucciones. A continuación se describen brevemente las características de cada grupo:

Manipulación de la memoria. Para realizar las operaciones de almacenamiento y recuperación en la memoria es necesario tener en cuenta el esquema de almacenamiento big endian empleado en SPARC V8. Las rutinas dentro de este grupo se encargan de transferir datos hacia y desde la memoria considerando los esquemas de almacenamiento de SPARC V8 y de la plataforma de hardware en que se ejecuta el simulador.

Manipulación de los registros. Las funciones dentro de este grupo son empleadas para realizar un manejo adecuado de las estructuras de datos usadas para simular los registros enteros de propósito general. Son necesarias puesto que toman en cuenta la organización en ventanas solapadas del archivo de registros.

Manipulación de instrucciones. Estos procedimientos llevan a cabo las tareas solicitadas al sistema para realizar el ensamble, desensamble y ejecución de instrucciones. El número de módulos implantados dentro de este grupo es grande, pues existen funciones para análisis sintáctico, análisis semántico y ensamble de instrucciones en lenguaje ensamblador así como funciones para el desensamble y la ejecución de instrucciones binarias.

Módulos de soporte. Aquí se encuentran rutinas de conversión, de manipulación de cadenas y rutinas aritméticas y lógicas sobre datos enteros de longitud arbitraria.

7.3.2 Estructuras de datos

Las estructuras de datos más importantes dentro de la capa del núcleo son las que se utilizan para representar la memoria y el conjunto de registros. Estas se describen en los siguientes párrafos.

7.3.2.1 Memoria principal

La memoria principal se representa simplemente mediante un arreglo lineal de caracteres (bytes). La longitud del arreglo se especifica dentro del código fuente y antes de comenzar la ejecución del programa se asigna dinámicamente el espacio de memoria requerido.

El núcleo contiene los módulos `Leer_Memoria` y `Escribir_a_Memoria`. Estos procedimientos realizan las operaciones de lectura y escritura de manera transparente para el resto de los módulos. Es necesaria

su intervención puesto que realizan el almacenamiento o recuperación de un dato de n bytes considerando el modo de almacenamiento big endian propio de SPARC V8. También toman en cuenta la forma de almacenamiento de la plataforma real que los ejecuta, que puede ser little endian o big endian. En la figura 7.3 se ilustra de manera general el funcionamiento del módulo `Escribir_a_Memoria`.

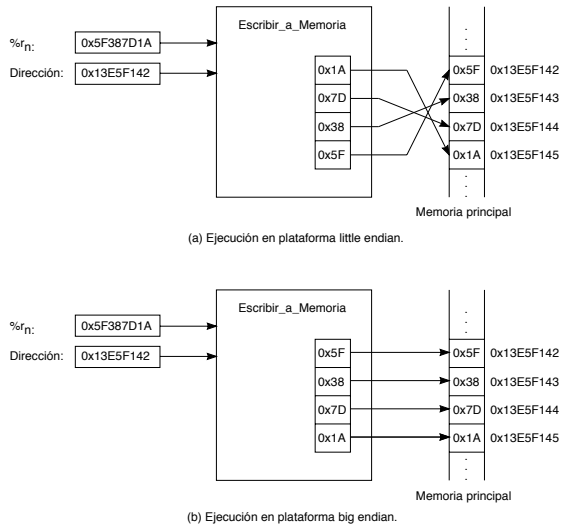


Figura 7.3: Operación de escritura a la memoria del simulador

Existe otro enfoque para simular la memoria física. En este se asignan bloques de memoria cuando alguna instrucción u otra operación lo requieran, es decir, por demanda. Se lleva un registro de los bloques asignados dinámicamente mediante un arreglo de apuntadores reservado al inicio [Bedichek 1990]. Se descartó este modelo por dos inconvenientes que presenta. Primero, la memoria corre el riesgo de tener "huecos" cuando el sistema operativo no le puede asignar más espacio. Segundo, el retardo en que se incurre al determinar el bloque que contiene la dirección referenciada, reservar un bloque no asignado y después realizar el almacenamiento o recuperación. En la figura 7.4 se ilustra una configuración de este estilo para una memoria de 128 MB con 32768 apuntadores a bloques de 4 KB. Se observa que algunos apuntadores permanecen nulos. La solución propuesta en este proyecto presenta únicamente el retardo ocasionado por la recuperación o almacenamiento de cada uno de los bytes que conforman el dato a leer o a escribir.

7.3.2.2 Registros

Para simular los registros utilizamos una estructura de datos que se compone de varios elementos. Los cinco primeros son valores enteros de 32 bits que simulan los registros `PSR`, `FSR`, `Y`, `PC` y `nPC`, respectivamente. A continuación se especifica un arreglo de 168 elementos para simular el archivo de registros enteros[1] y por último, un arreglo de 32 valores enteros de 32 bits para el archivo de registros de punto flotante. En la figura 7.5(a) se ilustra esta disposición con un esquema y en la figura 7.5(b) se muestra la declaración, en el lenguaje C, de las estructuras necesarias.

Cada elemento del arreglo `r` puede ser manipulado de tres formas distintas: como un valor entero de 32 bits con y sin signo o como un arreglo de 4 bytes. La representación con signo se emplea en la simulación de las instrucciones que realizan operaciones aritméticas con los registros enteros, la representación sin

[1] El sistema emplea 10 ventanas de registros, por lo tanto se tienen $(16 \times 10) + 8 = 168$ registros en total.

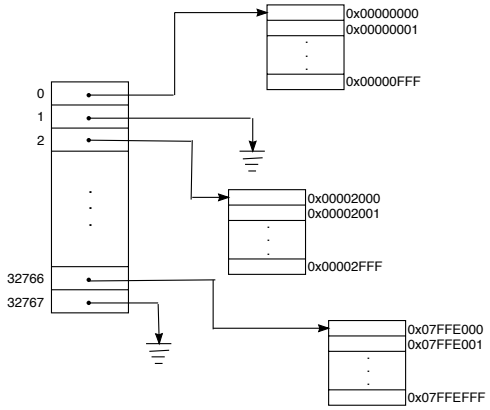


Figura 7.4: Esquema de simulación de la memoria mediante bloques

signo en las operaciones lógicas y la representación como arreglo se utiliza para accesar cada byte que compone un dato y así simplificar la simulación de las instrucciones de carga y almacenamiento.

Los módulos `Leer_Registro_Entero` y `Escribir_Registro_Entero` se emplean para llevar a cabo las operaciones de acceso a los registros enteros de manera transparente, como en el caso de la memoria. Cada procedimiento recibe como entrada un apuntador a la estructura que simula los registros, la dirección de un registro[2] y un valor entero o la dirección de una variable, dependiendo si el acceso es para escritura o lectura. Ambos módulos emplean la dirección del registro junto con los bits 0–4 de la variable `PSR` (campo CWP) dentro de la estructura para determinar el índice del elemento en el arreglo `r` en donde se almacenará o del que se leerá un dato.

7.3.2.3 Tablas

Existen tres estructuras de datos especiales definidas dentro del núcleo. Su misión es simplificar los procedimientos de ensamble, desensamble y ejecución. Son descritas a continuación:

La tabla de símbolos. Es una pieza fundamental para el proceso de ensamble. Por cada mnemónico reconocido existe una entrada que consta del mismo mnemónico, un apuntador al módulo de ensamble correspondiente y un índice para la tabla de códigos. En la figura 7.6(a) se proporciona un diagrama de la estructura de sus componentes.

La tabla de códigos. Contiene información de importancia para los módulos de ensamble, desensamble y ejeucución. Los campos que forman cada entrada contienen, respectivamente, el mnemónico de una instrucción, una máscara para construir el código binario, la especificación de los argumentos de tal instrucción y dos apuntadores a procedimientos. El primer apuntador hace referencia al módulo de desensamble y el segundo al procedimiento de ejecución[3] para dicha instrucción. En la figura 7.7(a) se puede observar un esquema de la organización de los elementos de esta tabla.

La instrucción `and`, entre otras, acepta como argumentos tres registros o bien dos registros y un valor inmediato. En casos como este, para el mismo mnemónico existirá una entrada en la tabla de códigos por cada variante posible de su grupo de argumentos. El campo `Operandos` en la entrada 66 almacena la cadena "`1,i,d`" que nos dice que la instrucción puede tener como argumentos un registro fuente entero (indicado por "`1`"), un valor inmediato (indicado por "`i`") y un registro destino

[2]La dirección de un registro entero es un valor r, tal que $0 \leq r \leq 31$.
[3]La *rutina de servicio* o *manejador de instrucción*.

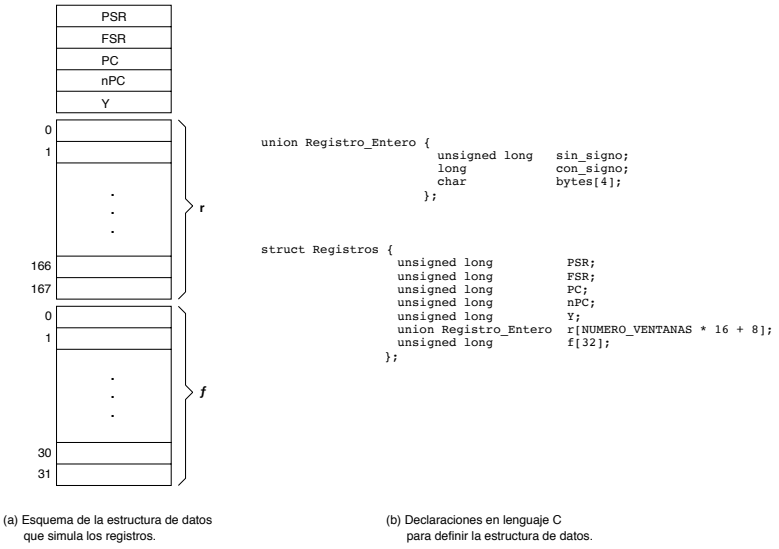


Figura 7.5: Esquema de la estructura de datos empleada para simular el conjunto de registros y su definición en lenguaje C

entero (indicado por "`d`"). La entrada 67 almacena la cadena "`1,2,d`" que indica que la instrucción también puede tomar como operandos dos registros fuente enteros ("`1`" y "`2`") y un registro destino entero ("`d`"). Para cada una de estas entradas el valor del campo `Mascara` será diferente, esto con la finalidad de que el módulo de ensamble construya el código binario correcto.

Las entradas de la tabla se agrupan de acuerdo al valor de los dos bits más significativos del campo `Mascara`, es decir, el campo `op` definido por los formatos de las intrucciones en la figura 3.8. Las instrucciones que tienen `op` $= 0$ son las primeras dentro de la tabla, les siguen las que tienen `op` $= 1$, y así sucesivamente.

La tabla de registros. Es empleada por los módulos de ensamble y desensamble. Cada entrada contiene como primer campo el identificador de un registro entero, de punto flotante o de estado. En el segundo campo se almacena un identificador equivalente, únicamente para las entradas correspondientes a los registros enteros. En el último campo se almacena la dirección del registro, siempre y cuando no se trate de un registro de estado, en cuyo caso se almacena el valor cero. La figura 7.8(a) ilustra mediante un esquema la estructura de las entradas dentro de esta tabla.

7.3.3 El proceso de ensamble

Existen varias funciones de ensamble dentro del núcleo, cada una de las cuales se encarga de llevar a cabo dicho proceso para un grupo de instrucciones con la misma sintaxis. Sin embargo, todas estas funciones se invocan mediante el mismo proceso e implantan esencialmente la misma estrategia. Para llevar a cabo el ensamble de una instrucción en lenguaje ensamblador se utiliza el siguiente procedimiento.

```
int Ensamblar_Instruccion(unsigned char * Linea, unsigned long * Instruccion,
                          struct Codigos * Tabla_Codigos, struct Simbolos * Tabla_Simbolos,
                          struct Id_Registros * Tabla_Registros)
{
 unsigned char  Mnemonico[20] = ‘‘’’;
```

	Mnemonico	Ensamblar	Indice
0	"sethi"	Ensamblar_Sethi	0
1	"ba"	Ensamblar_Branch	1
2	"bn"	Ensamblar_Branch	3
:		:	
32	"fbo"	Ensamblar_Branch	63
33	"call"	Ensamblar_Call	65
34	"and"	Ensamblar_Logic	66
:		:	
98	"fcmpes"	Ensamblar_FPop	174
99	"fcmped"	Ensamblar_FPop	175
100	"ldd"	Ensamblar_Load	176
:		:	
138	"lseek"	Ensamblar_Logic	252
139	"close"	Ensamblar_Close	254

(a) Esquema general de la tabla de símbolos y de su contenido.

```
struct Simbolos {
                  char           * Mnemonico;
                  int            ( * Ensamblar )();
                  unsigned int   Indice;
                };
```

(b) Declaración en lenguaje C de la estructura de cada entrada de la tabla de símbolos.

Figura 7.6: Esquema y declaración de la tabla de símbolos

```
    unsigned int   Ultimo_Caracter;
    int            Posicion, Resultado;

    if ( Linea[0] != '\0' )
      {
1      Convertir_a_Minusculas(Linea);
2      Delimitadores(Linea);
3      Ultimo_Caracter = Aislar_Mnemonico(Linea, Mnemonico);
4      Posicion = Buscar_Mnemonico(Mnemonico, Tabla_Simbolos);
5      if ( Posicion != NO_ENCONTRADO )
         {
6         Resultado = (Tabla_Simbolos[Posicion].Ensamblar)(Linea + Ultimo_Caracter, Posicion,
                                                            Instruccion, Tabla_Codigos,
                                                            Tabla_Simbolos, Tabla_Registros);
7         if ( Resultado != EXITO )
8           return(Resultado);
9         return(EXITO);
         }
       else
10       return(MNEMONICO_INVALIDO);
      }
    else
11    return(CADENA_VACIA);
   }
```

En las sentecias 1–3 se obtiene el primer elemento sintáctico de la cadena que contiene la instrucción, tal elemento corresponde al mnemónico. La proposición 4 realiza la búsqueda del mnemónico obtenido en la tabla de símbolos. Si la búsqueda es exitosa, en 6 se invoca a la función de ensamble correspondiente. Se trata de un llamado indirecto a través del apuntador que se encuentra en el segundo campo de la entrada hallada en la tabla de símbolos. Las sentencias 7–11 verifican condiciones de error y regresan los correspondientes códigos de error.

	Mnemonico	Mascara	Operandos	Desensamblar	Ejecutar	
0	"sethi"	0x01000000	"h,d"	Desensamblar_Sethi	Ejecutar_Sethi	
1	"ba"	0x10800000	"l"	Desensamblar_Branch	Ejecutar_Branch	
2	"ba"	0x30800000	",al"	Desensamblar_Branch	Ejecutar_Branch	
3	"bn"	0x00800000	"l"	Desensamblar_Branch	Ejecutar_Branch	
4	"bn"	0x20800000	",al"	Desensamblar_Branch	Ejecutar_Branch	op = 0
:			:			
63	"fbo"	0x1F800000	"l"	Desensamblar_Branch	Ejecutar_FBranch	
64	"fbo"	0x3F800000	",al"	Desensamblar_Branch	Ejecutar_FBranch	
65	"call"	0x40000000	"L"	Desensamblar_Call	Ejecutar_Call	op = 1
66	"and"	0x80082000	"1,i,d"	Desensamblar_Logic	Ejecutar_Logic	
67	"and"	0x80080000	"1,2,d"	Desensamblar_Logic	Ejecutar_Logic	
:			:			op = 2
174	"fcmpes"	0x81A80AA0	"e,f"	Desensamblar_FPop	Ejecutar_FCmp	
175	"fcmped"	0x81A80AC0	"e,f"	Desensamblar_FPop	Ejecutar_FCmp	
176	"ldd"	0xC1182000	"[1+i],g"	Desensamblar_Load	Ejecutar_Load	
177	"ldd"	0xC1180000	"[1+2],g"	Desensamblar_Load	Ejecutar_Load	
178	"ldd"	0xC0182000	"[1+i],d"	Desensamblar_Load	Ejecutar_Load	
179	"ldd"	0xC0180000	"[1+2],d"	Desensamblar_Load	Ejecutar_Load	op = 3
:			:			
252	"lseek"	0xC1480000	"1,2,d"	Desensamblar_Logic	NULL	
253	"lseek"	0xC1482000	"1,i,d"	Desensamblar_Logic	NULL	
254	"close"	0xC1780000	"d"	Desensamblar_Close	NULL	

(a) Esquema general de la tabla de códigos y de su contenido.

```
struct Codigos {
                    char            * Mnemonico;
                    unsigned long   Mascara;
                    char            * Operandos;
                    void            ( * Desensamblar )();
                    int             ( * Ejecutar )();
                };
```

(b) Declaración en lenguaje C de la estructura de cada entrada de la tabla de códigos.

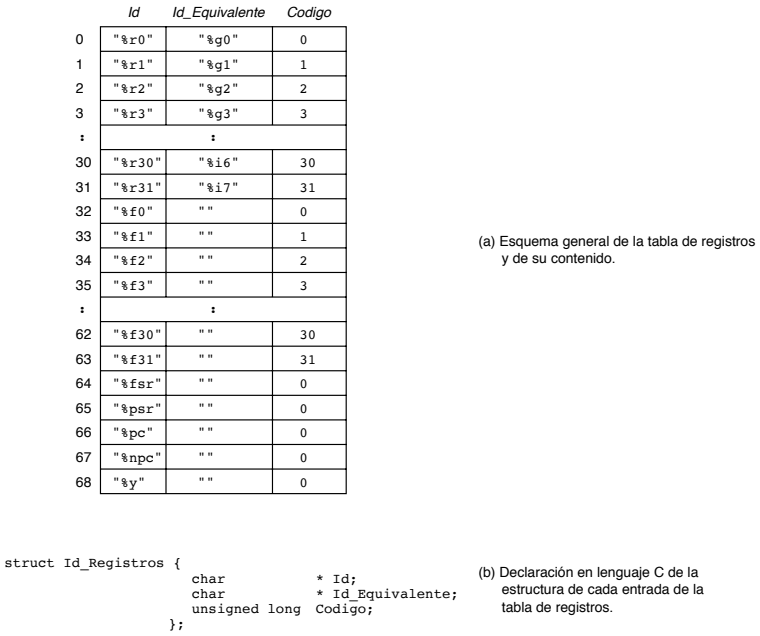
Figura 7.7: Esquema y declaración de la tabla de códigos

Una vez en ejecución el módulo de ensamble adecuado, este invoca a otro procedimiento que realiza el análisis sintáctico de la cadena y la separación de los elementos sintácticos restantes, es decir, los operandos. A continuación entra en acción otra función, cuyo objetivo es determinar si los operandos son válidos. En caso de ser así, este modulo habrá obtenido también las direcciones de los registros a los que hace referencia la instrucción, en caso de que los operandos sean identificadores de registros. También obtiene un valor numérico si existe, como operando, algún valor inmediato, un desplazamiento o un ASI. Si la verificación de los operandos fracasa, el módulo de ensamble termina regresando un código de error. Si los operandos son correctos se construye el código binario de la instrucción. En base al mnemónico de la instrucción y al tipo de sus operandos se obtiene de la tabla de códigos la máscara. Dependiendo del formato de la instrucción que se pretende ensamblar y del tipo de los operandos, se realizan operaciones `or` entre los valores obtenidos en el paso anterior y la máscara para producir finalmente la instrucción.

7.3.3.1 Ensamble de la instrucción `call`

El procedimiento anterior se ejemplifica para el ensamble de la instrucción `call -1023`, la cual provoca una transferencia de control incondiconal retardada a una instrucción que se encuentra 1023 instrucciones atrás.

Una vez obtenida la instrucción en lenguaje ensamblador el módulo `Ensamblar_Instruccion` es invocado. Este procedimiento realiza como primer paso el llamado a la función `Aislar_Mnemonico`, el cual obtiene la subcadena "`call`". El siguiente paso es hacer una búsqueda secuencial en la tabla de símbolos por la entrada que contiene dicha cadena. La búsqueda la realiza el módulo `Buscar_Mnemonico`, el cual en este caso regresa el valor índice 33. A continuación se invoca al procedimiento `Ensamblar_Call` mediante el apuntador almacenado en dicha entrada. Los parámetros para esta función son: la cadena que contiene la instrucción a ensamblar (`Linea`), el índice de la tabla de símbolos encontrado anteriormente (`Posicion`), un apuntador a la variable que contendrá el código binario (`Instruccion`) y finalmente, los



	Id	Id_Equivalente	Codigo
0	"%r0"	"%g0"	0
1	"%r1"	"%g1"	1
2	"%r2"	"%g2"	2
3	"%r3"	"%g3"	3
:	:		
30	"%r30"	"%i6"	30
31	"%r31"	"%i7"	31
32	"%f0"	""	0
33	"%f1"	""	1
34	"%f2"	""	2
35	"%f3"	""	3
:	:		
62	"%f30"	""	30
63	"%f31"	""	31
64	"%fsr"	""	0
65	"%psr"	""	0
66	"%pc"	""	0
67	"%npc"	""	0
68	"%y"	""	0


```
struct Id_Registros {
                      char           * Id;
                      char           * Id_Equivalente;
                      unsigned long  Codigo;
                    };
```


Figura 7.8: Esquema y declaración de la tabla de registros

apuntadores a las tablas de códigos, de símbolos y de registros. A continuación se muestra el código de este procedimiento.

```
   int Ensamblar_Call(unsigned char * Linea, int Posicion, unsigned long * Instruccion,
                      struct Codigos * Tabla_Codigos, struct Simbolos * Tabla_Simbolos,
                      struct Id_Registros * Tabla_Registros)
   {
    unsigned char   Operando[40] = ‘‘’’;
    unsigned char   Cadena_Operandos[10];
    unsigned int    i;
    long            Desplazamiento;
    int             Resultado;

1   Resultado = Analizar_Sintaxis_Call(Operando, Linea);
2   if ( Resultado == EXITO )
      {
3     Resultado = Analizar_Operandos_Call(Operando, Cadena_Operandos, & Desplazamiento);
4     if ( Resultado == EXITO )
        {
5       for (i = Tabla_Simbolos[Posicion].Indice;
             strcmp(Cadena_Operandos, Tabla_Codigos[i].Operandos) != 0;
             i ++);
6       * Instruccion = Tabla_Codigos[i].Mascara;
7       Desplazamiento &= 0x3FFFFFFF;
8       * Instruccion |= Desplazamiento;
9       return(EXITO);
        }
```

```
      else
10      return(Resultado);
     }
   else
11   return(Resultado);
  }
```

La sentencia 1, mediante la invocación al módulo `Analizar_Sintaxis_Call`, obtiene la subcadena que almacena el operando de la instrucción, también verifica que la sintaxis sea correcta. Para el caso de la instrucción en cuestión este módulo deposita en el arreglo `Operando` la cadena "`-1023`".

En el procedimiento `Analizar_Operandos_Call`, invocado en la sentencia 3, se realiza una conversión de la cadena a un valor numérico y se verifica que este sea un desplazamiento válido[4]. Este módulo además coloca la cadena "`L`" en el arreglo `Cadena_Operandos`, que simplemente indica que el argumento es un desplazamiento correcto.

Cuando los procedimientos anteriores terminan su ejecución se obtiene de la tabla de códigos la máscara con la cual se construye el código binario. La entrada en la tabla de símbolos que contiene el mnemónico `call` almacena en el tercer campo un índice. La máscara se busca en la tabla de códigos a partir de la entrada indicada por este índice. En general se busca la máscara apropiada recorriendo las entradas hasta que la comparación entre `Cadena_Operandos` y el campo `Operandos` sea exitosa, como se muestra en la sentencia 5 del código anterior. En este procedimiento el ciclo de la sentencia 5 no es necesario y se ilustra únicamente a manera de ejemplo, puesto que solo hay una entrada en la tabla de códigos para el mnemónico "`call`". Sin embargo para otras instrucciones pueden existir más entradas, en cuyo caso el ciclo es indispensable. En la sentencia 6 se recupera la máscara desde la entrada que se encontró. El código binario final se produce en las sentencias 6, 7 y 8, realizando una operación `or` entre la máscara (`0x40000000`) y los 30 bits menos significativos del desplazamiento (`0x3FFFFC01`). El resultado de tal operación es el código `0x7FFFFC01`.

7.3.4 El proceso de desensamble

Del mismo modo que en el caso anterior, los distintos módulos dedicados al desensamble de instrucciones se comportan de forma similar. Estos son invocados por el módulo principal `Desensamblar_Instruccion`, el cual se muestra a continuación junto con algunos comentarios sobre los pasos que lleva a cabo para realizar su función.

```
int Desensamblar_Instruccion(unsigned long Instruccion, unsigned char * Cadena_Instruccion,
                             struct Codigos * Tabla_Codigos,
                             struct Id_Registros * Tabla_Registros)
{
 unsigned long  a, op2, op3;
 unsigned int   i, Limite;

1  switch ( ( Instruccion & 0xC0000000 ) >> 30 )
   {
    case 0:
2           op2 = ( Instruccion & 0x01C00000 ) >> 22;
3           switch ( op2 )
            {
             case 4 :       /* == sethi == */
4                    i = 0;
5                    Limite = 1;
6                    break;
```

[4] Un valor entero de 30 bits con signo, que pertenece al intervalo $[-2^{29}, 2^{29} - 1]$.

```
              default:       /* == Bicc y FBicc == */
7                     a = Instruccion & 0xFFC00000;
8                     for ( i = 1;
                            ( a != Tabla_Codigos[i].Mascara ) && ( i < 65 );
                            i ++ );
9                     Limite = 65;
            }
10          break;
     case 1:                /* == call == */
11          i = 65;
12          Limite = 66;
13          break;
     case 2:
14          op3 = ( Instruccion & 0x01F80000 ) >> 19;
15          if ( ( op3 == 52 ) || ( op3 == 53 ) )  /* == FPop == */
              {
16             a = Instruccion & 0xC1F83FE0;
17             for ( i = 152;
                     ( a != Tabla_Codigos[i].Mascara ) && ( i < 176 );
                     i ++ );
18             Limite = 176;
              }
            else            /* == ALU == */
              {
19             a = Instruccion & 0xC1F82000;
20             for ( i = 66;
                     ( a != Tabla_Codigos[i].Mascara ) && ( i < 152 );
                     i ++ );
21             Limite = 152;
              }
22          break;
     case 3:                /* == ld,st,io == */
23          a = Instruccion & 0xC1F82000;
24          for ( i = 176;
                  ( a != Tabla_Codigos[i].Mascara ) && ( i < 255 );
                  i ++ );
25          Limite = 255;
    }
26  if ( i < Limite )
      {
27     ( Tabla_Codigos[i].Desensamblar )(Cadena_Instruccion, i, Instruccion, 16, Tabla_Codigos,
                                         Tabla_Registros);
28     return(EXITO);
      }
    else
29    return(CODIGO_INVALIDO);
   }
```

La estrategia consiste en buscar en la tabla de códigos el módulo apropiado para desensamblar la instrucción binaria actual. La búsqueda se realiza más eficientemente reduciendo el tamaño del espacio de búsqueda. Por ello la sentencia `switch` 1 determina el formato de la instrucción actual, mediante sus dos bits más significativos, y efectúa el proceso de búsqueda en la región de la tabla de códigos que contiene únicamente información sobre las instrucciones correspondientes a dicho formato. En el caso del formato 1 ($\mathsf{op} = 1$) existe una única instrucción y el índice de la entrada correspondiente en la tabla de

códigos se especifica inmediatamente en las sentencias 11–13. Para otros formatos como el 0 (op = 0) y el 2 (op = 2) es necesario verificar el valor de otro campo, op2 y op3 respectivamente, antes de realizar el proceso de búsqueda. Una vez que el espacio de búsqueda ha quedado bien delimitado se recorren las entradas que lo conforman, realizando una operación `and` entre la instrucción binaria y una máscara adecuada hasta que el resultado coincida con el valor almacenado en el campo `Mascara` de la entrada actual o hasta recorrer todas las entradas respectivas. Esta iteración se realiza para cada formato por los ciclos en las sentencias 8, 17, 20 y 24 respectivamente. Una vez finalizado el ciclo correspondiente a un formato, a la variable `Limite` se le asigna el índice de la tabla donde comienzan los datos para las instrucciones del siguiente formato, esta asignación se realiza en las sentencias 5, 9, 12, 18, 21 y 25.

La condición en la sentencia 26 es muy importante, cuando falla entonces la instrucción binaria es inválida y el procedimiento regresa un código de error. De otro modo, en la sentencia 27 es invocado el módulo de desensamble de forma indirecta a través del apuntador `Desensamblar`, almacenado en la entrada obtenida anteriormente. Una vez en ejecución, este módulo verifica el tipo de operandos que debe contener la instrucción, esta información la obtiene del campo `Operandos` dentro de la entrada en la tabla de códigos. En base a la información anterior, se extraen de la instrucción los operandos, que pueden ser direcciones de registros, valores inmediatos, desplazamientos, etc. Mediante el mnemónico en la tabla de códigos, los identificadores en la tabla de registros, los módulos de soporte y los valores numéricos obtenidos en el paso anterior se compone la cadena de caracteres que representa la instrucción en lenguaje ensamblador SPARC.

7.3.4.1 Desensamble de la instrucción `call`

A continuación mostramos el proceso para desensamblar el código 0x7FFFFC01 y obtener la cadena "`call HFFFFFC01`" o "`call -1023`", según la base que se indique al módulo de desensamble para esta instrucción.

El módulo `Desensamblar_Instruccion`, que reside en el núcleo, recibe como argumento el código 0x7FFFFC01. Ahora es necesario determinar la entrada en la tabla de códigos que almacena el apuntador a la función que puede desensamblar la instrucción. El primer paso es aislar los dos bits más significativos de la instrucción, es decir, el campo op. En este caso se tiene op = 01. Como se puede observar en la figura 7.7, la única entrada en la tabla de códigos asociada a este valor de op es la que contiene la información referente a la instrucción `call`. Queda entonces determinado el índice de la entrada requerida, cuyo valor es 65. En este caso no hay más que hacer que invocar al módulo `Desensamblar_Call`. Para otro valor de op es necesario operar la instrucción binaria mediante una operación `and` con el fin de obtener el "esqueleto" de la instrucción, el cual se compara con el campo `Mascara` de cada una de las entradas asociadas al valor de op. Cuando estos valores coinciden, el módulo de desensamble indicado en la entrada actual es invocado.

```
   void Desensamblar_Call(unsigned char * Cadena_Instruccion, unsigned int Posicion,
                          unsigned long Instruccion, unsigned int Base,
                          struct Codigos * Tabla_Codigos, struct Id_Registros * Tabla_Registros)
   {
    unsigned int    Indice;
    unsigned long   Desplazamiento;

1   strcpy(Cadena_Instruccion, Tabla_Codigos[Posicion].Mnemonico);
2   Indice = strlen(Tabla_Codigos[Posicion].Mnemonico);
3   Cadena_Instruccion[Indice] = ' ';
4   Indice ++;
5   Desplazamiento = Instruccion & 0x3FFFFFFF;
6   if ( ( Desplazamiento & 0x20000000 ) != 0 )
7     Desplazamiento |= 0xC0000000;
```

```
8   Entero_a_Cadena(Desplazamiento, Cadena_Instruccion + Indice, Base);
9   Indice += strlen(Cadena_Instruccion + Indice);
10  Cadena_Instruccion[Indice] = '\0';
   }
```

Todos los módulos de desensamble contenidos en el núcleo reciben los mismos parámetros, los cuales se muestran en el encabezado del procedimiento mostrado antes de este párrafo. En `Cadena_Instruccion` se almacenará la instrucción en lenguaje ensamblador producida por el módulo. El parámetro `Posicion` contiene el valor del índice hallado anteriormente (en este caso 65). En `Instruccion` se almacena el código binario de la instrucción. La base en que se expresan los valores numéricos de la instrucción en lenguaje ensamblador se indica mediante el parámetro `Base`. Por último se proporcionan los apuntadores a las tablas de símbolos y de registros.

El primer paso es copiar el mnemónico de la instrucción desde la tabla de códigos hacia la cadena, esto se realiza en la sentencia 1. En la sentencia 5 se obtiene el desplazamiento almacenado en los 30 bits menos significativos de la instrucción. De ser necesario se extiende el signo del desplazamiento en la sentencia 7. En la sentencia 8 se realiza una conversión del valor del desplazamiento a la cadena de dígitos que lo representa, esta cadena se almacena en la posición apropiada dentro del arreglo `Cadena_Instruccion`. Finalmente se marca el fin de la cadena en la sentencia 10.

El procedimiento anterior no hace uso del campo `Operandos` puesto que la instrucción `call` tiene un solo argumento. Para el caso de instrucciones con múltiples argumentos donde alguno de ellos varía, es necesario verificar el tipo de argumentos que serán obtenidos de la instrucción y utilizados para construir la cadena.

7.3.5 El proceso de ejecución

El funcionamiento del simulador debe ser comparado al de un intérprete. El código en lenguaje ensamblador introducido por el usuario es procesado para ser convertido en código objeto, que después es ejecutado. El código objeto puede ser considerado como un código interno que es decodificado por el intérprete, el que además invoca a las rutinas de servicio necesarias para ejecutarlo.

Las instrucciones contenidas en un intervalo de direcciones son interpretadas recolectando cada una a la vez y decodificándola para localizar y eventualmente ejecutar la rutina de servicio correspondiente. El núcleo contiene el módulo `Ejecutar_Instruccion` que realiza este proceso por cada instrucción y, al finalizar la rutina de servicio, incrementa los valores de los registros `PC` y `nPC` siempre y cuando la instrucción ejecutada no haya provocado una transferencia de control. En este último caso la misma rutina de servicio habrá modificado adecuadamente los registros contadores de programa.

```
   int Ejecutar_Instruccion(unsigned char * Memoria_SPARC, struct Registros * Registros_SPARC,
                            struct Codigos * Tabla_Codigos, unsigned int * Anular)
   {
    union Registro_Entero  Instruccion;
    unsigned long          Direccion;
    unsigned long          a, op2, op3;
    unsigned int           i, Limite;
    int                    Resultado;

1   if ( * Anular == FALSO )
      {
/* == RECOLECTAR INSTRUCCION  == */
2     Direccion = Registros_SPARC -> PC;
3     Leer_Memoria(Memoria_SPARC, Direccion, & Instruccion, 4);
/* == DECODIFICAR INSTRUCCION == */
4     switch ( ( Instruccion.Sin_Signo & 0xC0000000 ) >> 30 )
```

```
      {
       case 0:
5              op2 = ( Instruccion.Sin_Signo & 0x01C00000 ) >> 22;
6              switch ( op2 )
               {
                case 4 :  /* == sethi == */
7                        i = 0;
8                        Limite = 1;
9                        break;
                default:  /* == Bicc y FBfcc == */
10                       a = Instruccion.Sin_Signo & 0xFFC00000;
11                       for ( i = 1;
                               ( a != Tabla_Codigos[i].Mascara ) && ( i < 65 );
                               i ++ );
12                       Limite = 65;
               }
13             break;
       case 1:            /* == call == */
14             i = 65;
15             Limite = 66;
16             break;
       case 2:
17             op3 = ( Instruccion.Sin_Signo & 0x01F80000 ) >> 19;
18             if ( ( op3 == 52 ) || ( op3 == 53 ) ) /* == FPop == */
                 {
19                a = Instruccion.Sin_Signo & 0xC1F83FE0;
20                for ( i = 152;
                        ( a != Tabla_Codigos[i].Mascara ) && ( i < 176 );
                        i ++ );
21                Limite = 176;
                 }
               else       /* == ALU == */
                 {
22                a = Instruccion.Sin_Signo & 0xC1F82000;
23                for ( i = 66;
                        ( a != Tabla_Codigos[i].Mascara ) && ( i < 152 );
                        i ++ );
24                Limite = 152;
                 }
25             break;
       case 3:            /* == ld,st,io == */
26             a = Instruccion.Sin_Signo & 0xC1F82000;
27             for ( i = 176;
                     ( a != Tabla_Codigos[i].Mascara ) && ( i < 255 );
                     i ++ );
28             Limite = 255;
      }
/* == EJECUTAR INSTRUCCION == */
29    if ( i < Limite )
30      if ( i < 224 )
31        if ( Tabla_Codigos[i].Ejecutar != NULL )
            {
32           Resultado = ( Tabla_Codigos[i].Ejecutar )(Memoria_SPARC, Registros_SPARC,
                                                       Instruccion.Sin_Signo);
```

```
33            switch ( Resultado )
              {
               case CODIGO_1:
34                             Registros_SPARC->PC = Registros_SPARC->nPC;
35                             Registros_SPARC->nPC += 4;
36                             * Anular = FALSO;
37                             break;
               case CODIGO_2:
38                             * Anular = FALSO;
39                             break;
               case CODIGO_3:
40                             Registros_SPARC->PC = Registros_SPARC->nPC;
41                             Registros_SPARC->nPC += 4;
42                             * Anular = VERDADERO;
43                             break;
               case CODIGO_4:
44                             * Anular = VERDADERO;
45                             break;
               default      :
46                             Registros_SPARC->PC = Registros_SPARC->nPC;
47                             Registros_SPARC->nPC += 4;
48                             * Anular = FALSO;
              }
49            return(Resultado);
             }
           else
             {
50            Registros_SPARC->PC = Registros_SPARC->nPC;
51            Registros_SPARC->nPC += 4;
52            return(NO_IMPLANTADA);
             }
         else
53         return(INSTRUCCION_DELEGADA);
       else
         {
54        Registros_SPARC->PC = Registros_SPARC->nPC;
55        Registros_SPARC->nPC += 4;
56        return(INSTRUCCION_ILEGAL);
         }
      }
    else
      {
57     Registros_SPARC->PC = Registros_SPARC->nPC;
58     Registros_SPARC->nPC += 4;
59     * Anular = FALSO;
60     return(EXITO);
      }
   }
```

La sentencia 1 verifica si la instrucción actual debe ser anulada y en tal caso las sentencias 57–60 modifican los contadores de programa para que la siguiente instrucción pueda ser interpretada. De otro modo, como primer paso, se recolecta de la memoria la instrucción actual en las sentencias 2 y 3. Las sentencias 4–28 realizan la búsqueda de la respectiva rutina de servicio en la tabla de códigos mediante el mismo procedimiento implantado en las sentencias 1–25 del procedimiento `Desensamblar_Instruccion`.

Si la instrucción no es válida la condición de la sentencia 29 falla, se incrementan los contadores de programa en las sentencias 54–56 y se regresa un código de error. La proposición 32 es un llamado indirecto a la rutina que simula la ejecución de la instrucción. Todos estos módulos reciben los mismos parámetros, pero cada módulo es responsable de llevar a cabo las operaciones necesarias para simular la ejecución de la instrucción correspondiente. Por lo tanto las rutinas de servicio pueden tener poco o nada en común entre ellas.

Cada rutina de servicio devuelve un valor numérico al termino de su ejecución. En especial, la rutina que simula las instrucciones de salto puede regresar uno de varios valores, dependiendo de que haya provocado o no un salto y de que anule o no la siguiente instrucción. El procedimiento `Ejecutar_Instruccion` se encarga de revisar dicho valor para llevar a cabo las acciones pertinentes. A continuación se proporciona una descripción de los valores revisados por la sentencia `switch` 33:

`CODIGO_1.` La instrucción de salto no modificó los contadores de programa y la siguiente instrucción no debe ser anulada.

`CODIGO_2.` La instrucción de salto ha modificado los contadores de programa pero la siguiente instrucción no debe ser anulada.

`CODIGO_3.` La instrucción de salto no modificó los contadores de programa pero la siguiente instrucción debe ser anulada.

`CODIGO_4.` La instrucción de salto ha modificado los contadores de programa y la siguiente instrucción debe ser anulada.

Las últimas 31 entradas en la tabla de códigos corresponden a las instrucciones de entrada y salida, una extensión al conjunto de instrucciones definido por la arquitectura SPARC V8. La sentencia 30 verifica si se tiene una instrucción de este tipo y la sentencia 53 regresa un valor que indica que la instrucción debe ser ejecutada por un módulo externo al núcleo, es decir, el procedimiento `Ejecutar_Instruccion` delega la ejecución de una instrucción binaria de entrada y salida a ciertos módulos dentro de la capa de interfaz.

7.3.5.1 Ejecución de la instrucción `call`

Las rutinas de servicio pueden modificar el estado del procesador, almacenar información en la memoria o recuperar datos de la misma. También necesitan obtener los operandos de la instrucción binaria. Cada módulo recibe los parámetros `Memoria_SPARC` y `Registros_SPARC`, que son apuntadores al arreglo que simula la memoria y a la estructura de datos que simula el conjunto de registros, respectivamente. El parámetro `Instruccion` contiene la instrucción binaria a ejecutar. A continuación se muestra el módulo de ejecución para la instrucción `call`.

```
  int Ejecutar_Call(unsigned char * Memoria_SPARC, struct Registros * Registros_SPARC,
                    unsigned long Instruccion)
  {
   unsigned long    Contenido_PC;
   unsigned long    Disp_30;

1  Disp_30 = Instruccion & 0x3FFFFFFF;
2  Disp_30 <<= 2;
3  Contenido_PC = Registros_SPARC -> PC;
4  Escribir_Registro_Entero(Registros_SPARC, Contenido_PC, 15);
5  Registros_SPARC -> PC = Registros_SPARC -> nPC;
6  Registros_SPARC -> nPC = Contenido_PC + (long) Disp_30;
7  Estadisticas[31] ++:
```

```
8  return(CODIGO_2);
  }
```

En la sentencia 1 se extraen los 30 bits menos significativos de la instrucción binaria, que corresponden al desplazamiento. En la sentencia 2 este desplazamiento, en términos de instrucciones, se recorre 2 bits a la izquierda para convertirlo en un desplazamiento en términos de localidades de memoria. El contenido actual del registro `PC` se almacena en el registro entero `%o7` en la sentencia 4. Como la instrucción `call` provoca una transferencia retardada, el valor del registro `PC` se hace igual al valor del registro `nPC` en la sentencia 5. La dirección destino de la transferencia se calcula y se asigna al registro `nPC` en la sentencia 6. En la sentencia 7 se incrementa en uno el contador para instrucciones `call`. Finalmente se regresa un valor que le indica al intérprete que no modifique los valores de los contadores de programa ni que anule la siguiente instrucción al regreso de la función.

7.3.6 Módulos de soporte

Al analizar el código de los procedimientos anteriores, es posible percatarse de la presencia de algunos módulos que proporcionan diversos servicios a los procedimientos que los invocan. Se tienen funciones para convertir valores numéricos en cadenas de caracteres ASCII y viceversa, rutinas para reemplazar caracteres dentro de cadenas y para obtener subcadenas. Los procedimientos para la manipulación de la memoria y del archivo de registros enteros también pertenecen a este conjunto.

Mención especial requiere un conjunto de rutinas que implantan operaciones aritméticas sobre datos enteros de longitud arbitraria[5]. Tales operaciones incluyen complemento, complemento a dos, adición, sustracción, desplazamiento hacia la izquierda y hacia la derecha, multiplicación con o sin signo y división con o sin signo.

7.3.7 Información estadística

El conjunto de información estadística recolectada por un simulador es bastante útil para determinar y, en su caso, mejorar el rendimiento de un programa. Los simuladores completos pueden obtener, entre otra, la siguiente información concerniente a la ejecución de un programa:

- Número de fallos de lectura en el caché de instrucciones.
- Número de fallos de lectura o escritura en el caché de datos.
- Número de operaciones de lectura o de escritura a una dirección particular de memoria.
- Número de transferencias de control hacia una instrucción.
- Número de transferencias de control desde una instrucción.

Obviamente, para obtener toda esta información es necesario simular adecuadamente ciertos elementos como los cachés de datos e instrucciones y el caché TLB. Nuestro simulador simplemente cuenta el número de instrucciones ejecutadas y clasifica estos resultados en base al tipo de instrucción, ya que no cuenta con los medios para simular los componentes mencionados anteriormente.

Además de simular la ejecución de un tipo específico de instrucción, cada rutina de servicio se encarga de incrementar un contador asociado. Los contadores para cada tipo de instrucción se encuentran en un arreglo llamado `Estadisticas`. Cuando la ejecución de un programa termina el usuario puede consultar los valores de estos contadores y tomar decisiones sobre el comportamiento del programa.

[5]En realidad, operan sobre datos enteros de p bits de longitud tal que $p = 32n$, donde n es el número de palabras de 32 bits.

7.4 La capa de interfaz

El sistema SPARCSim ha sido implantado, hasta el momento, en tres diferentes ambientes operativos: MS-DOS, Solaris y NeXTSTEP. En los dos primeros el usuario interacciona con el sistema mediante comandos de línea en una sesión no gráfica. Una implantación más adecuada fue hecha para el ambiente NeXTSTEP, en donde el usuario realiza sus operaciones mediante la interacción con los objetos que conforman la interfaz.

El objetivo de este proyecto es el de construir una herramienta de apoyo a la comprensión de la arquitectura SPARC. Debido a esto prestamos especial interés al diseño de la interfaz hacia el usuario, puesto que es el intermediario entre el usuario y los módulos que conforman el núcleo de la aplicación. Como resultado obtuvimos una interfaz gráfica completamente interactiva, robusta, sencilla y fácil de comprender y utilizar.

La construcción de la interfaz gráfica para una aplicación de NeXTSTEP es una tarea sencilla y no tediosa gracias a la ayuda de la aplicación Interface Builder. Esta aplicación permite elaborar una interfaz a través de la manipulación de objetos definidos por el Application Kit, es posible colocarlos en diferentes lugares, cambiar sus atributos, establecer conexiones entre ellos, etc. Sin embargo esta herramienta no sustituye la capacidad del desarrollador para implantar la funcionalidad de su aplicación. El programador construye las clases necesarias y crea las instancias correspondientes, las cuales interaccionan unas con otras y con los objetos que forman la interfaz, para alcanzar el objetivo de la aplicación y obtener los resultados deseados. El simulador, como una aplicación de NeXTSTEP, debe funcionar de la misma manera. Sin embargo, el simulador no es una aplicación que explote intensivamente la programación orientada a objetos, puesto que un gran porcentaje de su funcionalidad radica en los módulos presentes dentro del núcleo.

La única clase diseñada especialmente para este sistema define al *objeto controlador* de la aplicación. Este objeto recibe mensajes provenientes de los diferentes objetos que conforman la interfaz y, de acuerdo al mensaje, invoca a alguno de sus métodos para que lleve a cabo la solicitud del usuario. Para poder cumplir con cada requerimiento, los métodos deben invocar a algún módulo dentro del núcleo y, una vez que este ha finalizado, enviar mensajes a otros objetos para mostrar al usuario los resultados del procesamiento.

7.4.1 Componentes de la interfaz

La interfaz consta de cuatro ventanas con las que el usuario puede interactuar para llevar a cabo las operaciones que requiera. A continuación se describen brevemente:

Interfaz con los registros. A través de esta ventana el usuario puede consultar y/o modificar los valores contenidos en los registros enteros, de punto flotante, contadores de programa, banderas de condición y otros registros de estado. Además puede desplazarse a lo largo de las ventanas de registros. En la figura 7.9 se muestra esta ventana.

Interfaz con la memoria. Mediante esta interfaz el usuario puede examinar intervalos de memoria, modificar el valor contenido en alguna dirección, llenar con un caracter un intervalo y desplazarse a través del contenido de la memoria. La estructura de esta interfaz se muestra en la figura 7.10.

Interfaz para los códigos de instrucciones. Con esta ventana es posible ensamblar instrucciones desde un archivo o desde un objeto campo de texto dentro de la misma interfaz. Es posible invocar la ejecución de las instrucciones contenidas en el intervalo de memoria especificado, modificar tal intervalo, trazar una instrucción o desensamblar un conjunto de instrucciones en memoria. En la figura 7.11 se muestra la interfaz.

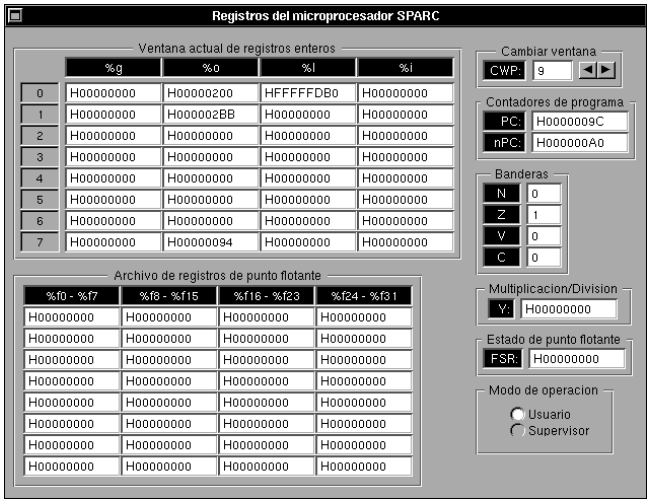


Figura 7.9: La interfaz con los registros

Ventana de estadísticas. Después de realizada la ejecución de las instrucciones contenidas en un intervalo, la interfaz de estadísticas se actualiza para mostrar el número de instrucciones por cada categoría que fueron ejecutadas. Un esquema de esta ventana se muestra en la figura 7.12.

El menú principal. Contiene los comandos que le permiten al usuario tener control sobre la aplicación, sobre las ventanas del sistema y sobre la información seleccionada. El menú principal se muestra en la figura 7.13 y a continuación se proporciona una descripción de sus submenús:

Info. Contiene un solo comando, llamado *Info Panel...*, que muestra al usuario una ventana con información sobre el sistema.

Edit. Contiene los comandos necesarios para cortar, copiar, pegar y eliminar el texto que el usuario halla seleccionado.

Windows. Los comandos de este submenú permiten al usuario manipular las ventanas que componen la aplicación. Es posible minimizarlas, cerrarlas, ordenarlas una sobre otra o colocar alguna de ellas al frente de las demás.

Hide. Oculta todas las ventanas de la aplicación, incluyendo el menú principal.

Quit. Pone fin a la ejecución de la aplicación.

La consola para entrada y salida. Es el medio a través del cual el usuario proporciona información al programa y recibe los resultados de la ejecución del mismo. La ventana que contiene esta consola se muestra en la figura 7.14.

La figura 7.15 muestra la aplicación en ejecución dentro del ambiente NeXTSTEP. Es posible observar en la figura todas las ventanas descritas anteriormente.

7.4.2 La clase SimuladorSPARC

La clase **SimuladorSPARC**, que define al objeto de control, contiene todas las variables instancia y todos los métodos necesarios para controlar adecuadamente la aplicación. El código que a continuación se muestra es el contenido del archivo de interfaz de la clase. Como puede observarse, es bastante amplio

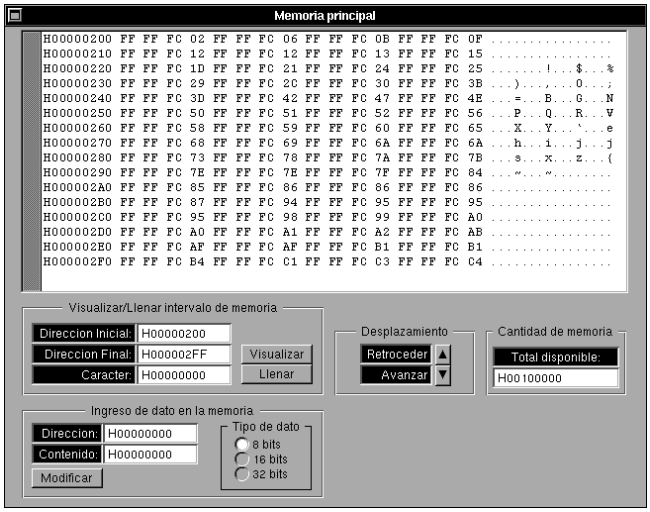


Figura 7.10: La interfaz con la memoria

pero es fácil de comprender. Las primeras líneas, que contienen la directiva `#import`, incluyen los archivos de código fuente que conforman el núcleo de la aplicación.

```
/* ============================
        SimuladorSPARC.h
   ============================ */

#import <appkit/appkit.h>

unsigned int    Estadisticas[51];

#import ‘‘./kernel/alu.h’’
#import ‘‘./kernel/tipos.h’’
#import ‘‘./kernel/general.h’’
#import ‘‘./kernel/branch.h’’
#import ‘‘./kernel/call.h’’
#import ‘‘./kernel/fpop.h’’
#import ‘‘./kernel/jmpl.h’’
#import ‘‘./kernel/load.h’’
#import ‘‘./kernel/logic.h’’
#import ‘‘./kernel/rett.h’’
#import ‘‘./kernel/sethi.h’’
#import ‘‘./kernel/store.h’’
#import ‘‘./kernel/rdwr.h’’
#import ‘‘./kernel/restsave.h’’
#import ‘‘./kernel/io.h’’
#import ‘‘./kernel/tablas.h’’

@interface SimuladorSPARC:Object
{
 id             infoPanel;
/* == Variables instancia involucradas en
```

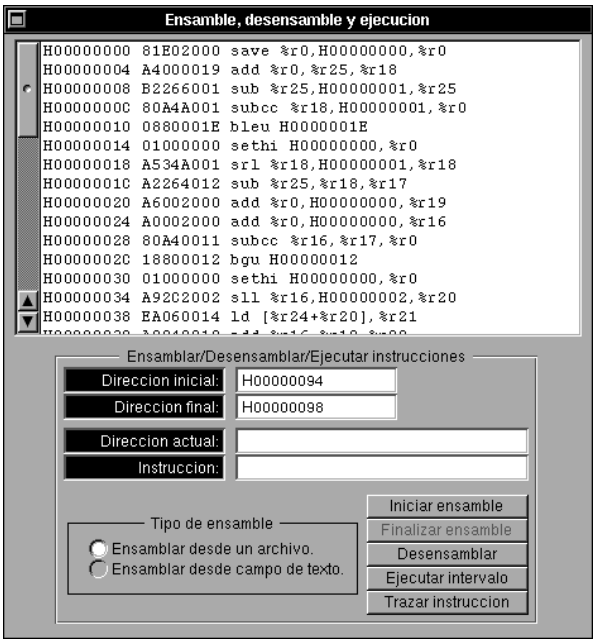


Figura 7.11: La interfaz con los códigos de instrucciones

```
      el control de la ventana de los registros. == */
 id             campoCWP;
 id             banderas;
 id             contadoresPrograma;
 id             registrosEnteros;
 id             registrosPuntoFlotante;
 id             registroFSR;
 id             registroY;
/* == Variables instancia involucradas en
      el manejo de la ventana de memoria principal. == */
 id             memoriaPrincipal;
 id             matrizIntervalo;
 id             matrizIngreso;
 id             matrizTiposDato;
 id             totalMemoria;
 unsigned long  direccionInicioIntervalo;
 unsigned long  direccionFinIntervalo;
 unsigned long  caracter;
 unsigned long  direccionIngreso;
 unsigned long  contenido;
/* == Variables instancia involucradas en el
      manejo de la ventana de operacion de codigo. == */
 id             textoCodigo;
 id             matrizIntervaloProceso;
 id             matrizDatosInstruccion;
 id             tipoEnsamble;
 id             matrizBotones;
```

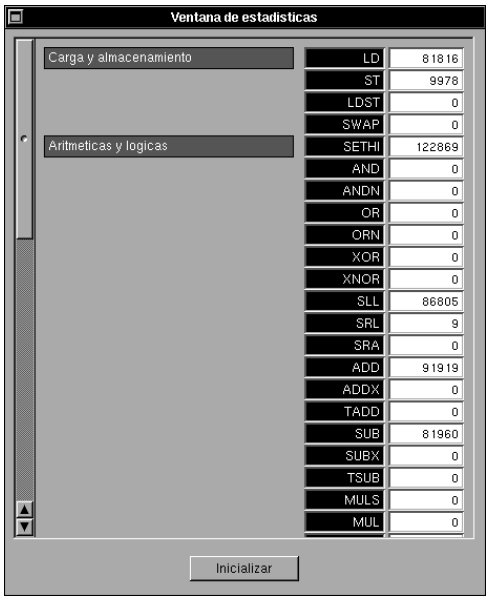


Figura 7.12: La ventana de estadísticas

```
 unsigned long  direccionInicioProceso;
 unsigned long  direccionActualProceso;
 unsigned long  direccionFinProceso;
/* == Variables instancia involucradas en el
      manejo de la ventana de estadisticas. == */
 id             matrizEstadisticas;
/* == Variables instancia involucradas con el
      manejo de la entrada y salida. == */
 id             terminalES;
}

- showInfo: sender;

- agregarTexto: (unsigned char *) Cadena en: texto;
- eliminarTextoEn: texto;

- appDidInit: sender;
- appWillTerminate: sender;
/* == Acciones y metodos enviados por los objetos
      en la interfaz con los registros. == */
- modificarRegistroEntero: sender;
- modificarRegistroPuntoFlotante: sender;
- modificarRegistroY: sender;
- modificarContadoresPrograma: sender;
- cambiarVentanaActual: sender;
- establecerModoOperacion: sender;
/* == Acciones y metodos enviados por los objetos
      en la interfaz con la memoria principal. == */
- actualizacionIntervalo: sender;
```

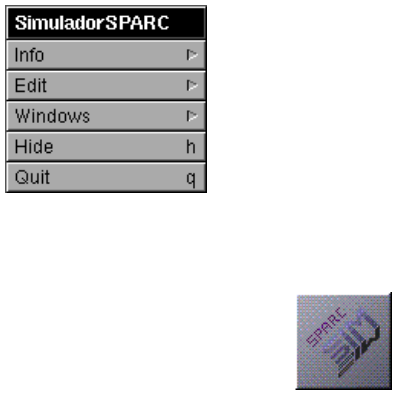


Figura 7.13: El menú principal y el ícono del simulador

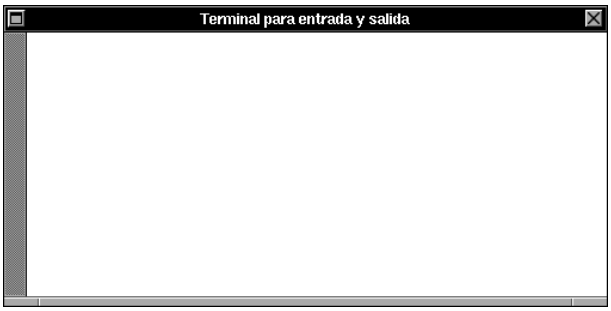


Figura 7.14: La ventana que contiene la consola para entrada y salida

```
- actualizacionDatosIngreso: sender;
- desplazarIntervaloMemoria: sender;
- visualizarIntervaloMemoria: sender;
- llenarIntervaloMemoria: sender;
- ingresarDatoEnMemoria: sender;
/* == Acciones y metodos enviados por los objetos
      en la interfaz de operacion de codigo. == */
- actualizacionIntervaloProceso: sender;
- ensamblarDesdeCampoTexto: sender;
- ensamblar: sender;
- terminarEnsamble: sender;
- desensamblar: sender;
- ejecutar: sender;
- trazar: sender;
/* == Acciones y metodos enviados por los objetos
      en la interfaz con las estadisticas. == */
- inicializarEstadisticas: sender;

@end
```

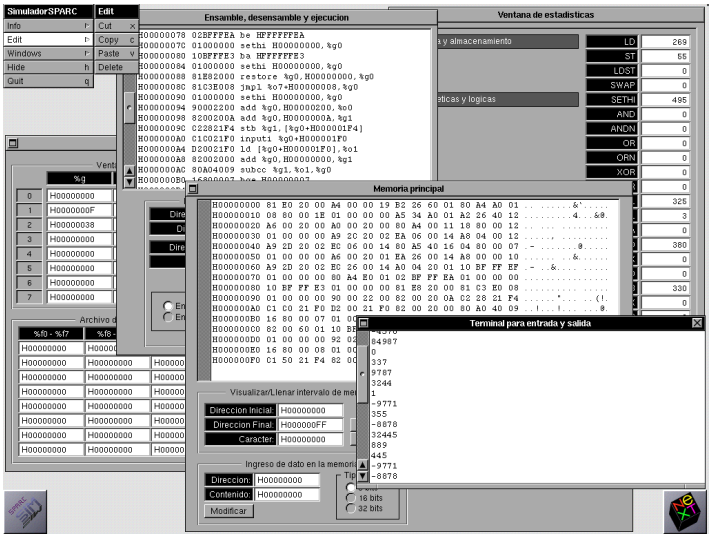


Figura 7.15: La aplicación SPARCSim en ejecución en el ambiente NeXTSTEP

7.4.2.1 Variables instancia de tipo entero

Las variables instancia de tipo entero declaradas por la clase contienen, en el momento de la ejecución, direcciones ingresadas por el usuario y ciertos valores numéricos. A continuación se proporciona una descripción de cada una de ellas:

`direccionInicioIntervalo.` Almacena la dirección inicial del intervalo de direcciones que el usuario puede visualizar y manipular mediante la ventana mostrada en la figura 7.10.

`direccionFinIntervalo.` Contiene la dirección final del intervalo mencionado en la descripción anterior.

`caracter.` El valor asignado a esta variable puede ser almacenado en cada una de las localidades que conforman el intervalo de direcciones anteriormente descrito.

`contenido.` Tiene asignado un valor que puede ser escrito en la memoria a partir de cierta dirección. El rango de valores para esta variable está determinado por el tipo que le defina el usuario (1 byte, 2 bytes o 4 bytes).

`direccionIngreso.` Contiene la dirección de memoria a partir de la cual se escribirá el valor almacenado en la variable antes descrita. La validez de su alineación será determinada por el tipo que el usuario haya determinado para la variable anterior.

`direccionInicioProceso.` Contiene la dirección inicial de una secuencia de instrucciones binarias presente en la memoria. Los valores de esta y las siguientes variables son proporcionados por el usuario mediante la ventana 7.11.

`direccionFinProceso.` Almacena la dirección final de la secuencia de instrucciones mencionada anteriormente.

`direccionActualProceso.` Durante el proceso de ensamble de una secuencia de instrucciones, que son proporcionadas por el usuario a través de la interfaz, esta variable contiene la dirección donde será almacenado el código binario producido por el ensamble de la instrucción actual.

El usuario proporciona los valores para estas variables mediante los campos de texto[6] contenidos en las ventanas 7.10 y 7.11. Cada uno de estos objetos envía un mensaje al objeto controlador siempre que el usuario halla ingresado un valor, el objeto controlador recibe el mensaje y toma el valor introducido para asignarlo a la variable apropiada.

7.4.2.2 Conectores

NeXTSTEP proporciona un esquema que permite establecer enlaces entre los objetos que conforman una aplicación, ya sea entre los objetos que componen la interfaz, entre las instancias de las clases definidas por el programador o entre estas instancias y los objetos en la interfaz. Las variables instancia de tipo `id` declaradas por las clases, denominadas conectores (*outlets*), almacenan apuntadores a otros objetos. El programador especifica, de manera interactiva, estas variables y asigna sus valores a través del mecanismo de conexión (*connection*) proporcionado por Interface Builder [Garfinkel y Mahoney 1993].

Las conexiones hechas por el programador, así como los objetos involucrados, son almacenados en los archivos nib que componen la aplicación. Los objetos son creados en el momento en que los archivos nib son cargados en memoria y a continuación se establecen automáticamente las conexiones especificadas durante la sesión de Interface Builder que generó tales archivos. Si el objeto A contiene un conector que apunta al objeto B entonces A puede enviar mensajes a B a través de dicho conector, sin embargo, A no puede recibir mensajes de B mediante ese mismo enlace. El principal archivo nib de la aplicación se llama **SPARCSimulator.nib**, este archivo almacena toda la información referente a cada objeto que compone el sistema así como las conexiones entre ellos.

Los conectores que el objeto de control emplea para comunicarse con los objetos dentro de la interfaz con los registros se describen a continuación:

`campoCWP`**.** Apunta al campo de texto donde se muestra el valor del apuntador a la ventana de registros actual.

`banderas`**.** Apunta a la matriz de campos de texto donde el usuario puede consultar el valor de los códigos de condición.

`contadoresPrograma`**.** Apunta a la matriz de campos de texto donde se muestran y se modifican los valores de los registros `PC` y `nPC`.

`registrosEnteros`**.** Apunta a la matriz de campos de texto donde el usuario puede consultar e ingresar los valores de los registros enteros contenidos en la ventana actual.

`registrosPuntoFlotante`**.** Apunta a la matriz de campos de texto en la que el usuario puede modificar y consultar los valores almacenados en los registros de punto flotante.

`registroFSR`**.** Apunta al campo de texto que permite visualizar el valor del registro de estado de punto flotante.

`registroY`**.** Apunta al campo de texto que permite ingresar y revisar el valor del registro `Y`.

El objeto de control se comunica con algunos de los objetos contenidos en la interfaz con la memoria a través de los siguientes conectores:

`memoriaPrincipal`**.** Permite al objeto controlador enviar mensajes al objeto de la clase **ScrollView** en donde se visualiza el contenido del intervalo de memoria actual.

[6] Instancias de la clase **TextField**.

`matrizIntervalo`. Apunta a la matriz de campos de texto donde el usuario puede especificar y consultar las direcciones inicial y final del intervalo de memoria actual así como el caracter con el que se va a llenar. Estos datos están almacenados en las variables `direccionInicioIntervalo`, `direccionFinIntervalo` y `caracter`, respectivamente.

`matrizIngreso`. Apunta a la matriz de campos de texto en la que el usuario puede ingresar un valor entero y la dirección donde será almacenado. Los valores anteriores están almacenados en las variables `direccionIngreso` y `contenido`, respectivamente.

`matrizTiposDato`. Permite al objeto controlador enviar mensajes a la matriz de objetos **ButtonCell** para solicitar el tipo seleccionado por el usuario para el valor entero a ser almacenado en memoria.

`totalMemoria`. Es un conector que apunta a un campo de texto donde el usuario puede consultar la cantidad total de memoria disponible.

Los siguientes conectores establecen los enlaces entre el objeto de control y los objetos presentes en la interfaz para la manipulación de instrucciones:

`textoCodigo`. Apunta a un objeto de la clase **ScrollView** cuyo propósito es desplegar secuencias de instrucciones en lenguaje ensamblador y algunos mensajes.

`matrizIntervaloProceso`. Apunta a la matriz de campos de texto donde el usuario puede introducir y consultar los valores de las variables `direccionInicioProceso` y `direccionFinIntervalo`.

`matrizDatosInstruccion`. Un conector que apunta a la matriz de campos de texto donde el usuario ingresa una instrucción a ser ensamblada y en la que consulta la dirección de memoria donde será depositado el código binario resultante. Esta dirección está contenida en la variable instancia `direccionActualProceso`.

`tipoEnsamble`. Un conector que apunta a la matriz de objetos definidos por la clase **ButtonCell** donde el usuario especifica si desea ensamblar un archivo de código fuente o si desea proporcionar cada instrucción directamente en la interfaz.

`matrizBotones`. Apunta a la matriz de objetos de la clase **Button** en la que el usuario invoca el proceso de ensamble de instrucciones en lenguaje ensamblador, el desensamble de instrucciones binarias y la ejecución de las mismas.

Finalmente, la clase contiene los siguientes conectores que permiten enviar mensajes hacia distintos objetos en la interfaz:

`infoPanel`. Es un conector que apunta a una ventana que contiene información a cerca del sistema. Esta ventana se encuentra en un archivo nib llamado **info.nib**, el cual es cargado en memoria cuando el usuario selecciona por primera vez el comando *Info Panel...* del submenú *Info* en el menú principal.

`matrizEstadisticas`. Es un apuntador a la matriz de campos de texto que proporciona la información referente al número de instrucciones ejecutadas en cada categoría.

`terminalES`. Este conector apunta hacia un objeto de la clase **ScrollView** que funciona como consola para realizar entrada y salida durante la ejecución de un programa.

La figura 7.16 muestra un esquema de la conexión entre el objeto controlador del sistema y la matriz de botones presente en la ventana de manipulación de programas. Como se puede observar el enlace se establece desde la instancia de la clase **SimuladorSPARC** hacia la matriz a través del conector `matrizBotones`. En la figura se puede observar la presencia de la ventana correspondiente al archivo **SPARCSimulator.nib**, la ventana *Inspector* y la interfaz con las instrucciones.

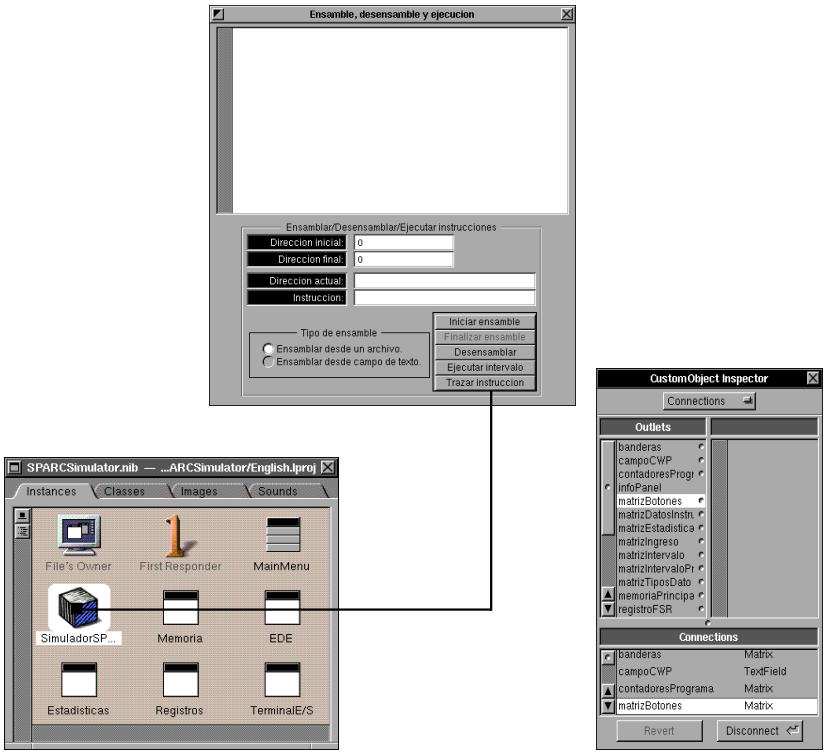


Figura 7.16: Conexión entre el objeto controlador y una matriz mediante Interface Builder

7.4.2.3 Acciones

Una acción (*action*) es un tipo especial de método definido por una clase. Las acciones funcionan principalmente como manejadores de eventos debido a que son invocadas en respuesta a los mensajes enviados por los objetos manipulados por el usuario, generalmente instancias de las clases pertenecientes al Application Kit. El único argumento de una acción se llama `sender` y es de tipo `id`, este argumento es un apuntador al objeto emisor del mensaje que provocó la ejecución de la acción. Un objeto A puede solicitar que otro objeto B ejecute alguna acción estableciendo, mediante Interface Builder, un enlace de A hacia B a través de un conector contenido en A[7]. En seguida se selecciona de entre todas las acciones de B la que será ejecutada en respuesta al mensaje enviado por A cuando este sea manipulado por el usuario.

La clase **SimuladorSPARC** define un gran número de acciones, que conforman casi la totalidad del conjunto de métodos que puede ejecutar el objeto controlador. A continuación describimos las acciones definidas por la clase para responder a los mensajes enviados por los objetos contenidos en la interfaz con los registros:

`modificarRegistroEntero:.` Es invocada cuando el usuario modifica el valor de alguna de las celdas que conforman la matriz de campos de texto que contiene los valores de los registros enteros. Esta acción invoca al módulo `Escribir_Registro_Entero`, presente dentro del núcleo, para almacenar el valor ingresado en el registro apropiado.

[7]Las clases pertenecientes al Application Kit declaran a este conector con el nombre de `target`.

`modificarRegistroPuntoFlotante:.` Se ejecuta cuando el usuario modifica el valor contenido en alguna de las celdas de la matriz de campos de texto que muestra el valor de los registros de punto flotante. Este método asigna directamente el valor ingresado al registro adecuado.

`modificarRegistroY:.` Esta acción se invoca cuando el usuario introduce un valor en el campo de texto asociado al registro `Y`. La acción asigna de forma directa el nuevo valor al registro.

`modificarContadoresPrograma:.` Se ejecuta como consecuencia del ingreso de un valor en alguna de las celdas de la matriz que muestra el contenido de los contadores de programa. Este método asigna directamente el valor ingresado al contador correspondiente.

`cambiarVentanaActual:.` Esta acción se invoca cuando el usuario presiona alguno de los botones que modifican el valor del campo CWP del registro `PSR`. Una vez que se ha realizado dicha modificación, los valores de los registros en la ventana actual son visualizados en la matriz de campos de texto correspondiente.

`establecerModoOperacion:.` Esta acción se ejecuta cuando el usuario selecciona alguna de las celdas de la matriz que le permite cambiar el modo de operación de procesador. Este método modifica directamente el valor del bit S en el registro `PSR`.

El objeto controlador responde también a los mensajes enviados por los objetos localizados en la interfaz con la memoria ejecutando alguna de las siguientes acciones:

`actualizacionIntervalo:.` Se invoca en respuesta a un mensaje enviado por la matriz en donde el usuario especifica un intervalo de memoria e ingresa un caracter. Este método modifica el valor de las variables instancia `direccionInicioIntervalo`, `direccionFinIntervalo` y `caracter`.

`actualizacionDatosIngreso:.` Se ejecuta en respuesta al mensaje enviado por la matriz en donde el usuario especifica un valor entero y la dirección donde será almacenado. Modifica el valor de las variables instancia `contenido` y `direccionIngreso`.

`desplazarIntervaloMemoria:.` El usuario puede examinar el contenido de la memoria en bloques de 256 bytes, desplazándose a lo largo de ellos mediante una matriz de botones. Este método es invocado en respuesta a un mensaje proveniente de dicha matriz, asigna a las variables instancia `direccionInicioIntervalo` y `direccionFinIntervalo` las direcciones inicial y final del nuevo bloque y muestra al usuario su contenido.

`visualizarIntervaloMemoria:.` Es posible examinar un bloque de memoria de tamaño arbitrario mediante el botón *Examinar*, habiendo especificado previamente sus direcciones inicial y final. Este método se ejecuta como respuesta a la interacción del usuario con el botón, visualizando el contenido del bloque especificado.

`llenarIntervaloMemoria:.` El usuario pulsa el botón *Llenar* cuando desea almacenar un caracter a lo largo de un intervalo de memoria. El botón envía entonces un mensaje al objeto controlador solicitando la ejecución de esta acción, la cual simplemente almacena el valor de la variable instancia `caracter` en cada localidad del intervalo.

`ingresarDatoEnMemoria:.` El objeto controlador ejecuta esta acción como consecuencia del mensaje enviado por el botón *Modificar* cuando es presionado por el usuario. Este método invoca al módulo `Escribir_a_Memoria` para almacenar el valor de la variable instancia `contenido` a partir de la dirección indicada por la variable instancia `direccionIngreso`.

Las siguientes acciones se ejecutan en respuesta a los mensajes enviados al objeto controlador por los objetos contenidos en la interfaz para la manipulación de instrucciones:

`actualizacionIntervaloProceso:.` Se invoca en respuesta al mensaje enviado por la matriz de campos de texto donde el usuario establece un intervalo de memoria que contiene instrucciones binarias. Este método modifica los valores de las variables intancia `direccionInicioProceso` y `direccionFinProceso`.

`ensamblarDesdeCampoTexto:.` Se pone en ejecución en respuesta al mensaje enviado por el campo de texto donde el usuario proporciona una instrucción en lenguaje ensamblador. Este método invoca al módulo `Ensamblar_Instruccion` para obtener la instrucción binaria correspondiente y después al módulo `Escribir_a_Memoria` para almacenarla a partir de la direccion contenida en la variable instancia `direccionActualProceso`.

`ensamblar:.` Esta acción se invoca cuando el usuario pulsa el botón *Iniciar ensamble*. Este método verifica la fuente de instrucciones seleccionada (un archivo o un campo de texto) y envía un mensaje al objeto controlador para ejecutar el método apropiado que realiza el ensamble.

`terminarEnsamble:.` Cuando el usuario ensambla instrucciones proporcionadas mediante el campo de texto, el resto de los objetos dentro de la interfaz son desactivados con el propósito de que el usuario no pueda ingresar un valor en un lugar distinto. El botón *Finalizar ensamble* al ser pulsado envía un mensaje que provoca la ejecución de esta acción, la cual tiene como objetivo activar los demás objetos para que el usuario pueda realizar otras operaciones.

`desensamblar:.` Se invoca debido a un mensaje enviado por el botón *Desensamblar* cuando es pulsado por el usuario. Este método lee, mediante el módulo `Leer_Memoria`, cada instrucción contenida en el intervalo definido por las variables instancia `direccionInicioProceso` y `direccionFinProceso`, para cada una obtiene la instrucción en lenguaje ensamblador correspondiente empleando el módulo `Desensamblar_Instruccion`. Esta instrucción es enviada en un mensaje al objeto apuntado por el conector `textoCodigo` para ser mostrada al usuario.

`ejecutar:.` Se invoca cuando el objeto controlador recibe un mensaje proveniente del botón *Ejecutar intervalo*. Este método lee y ejecuta, mediante el módulo `Ejecutar_Instruccion`, cada instrucción comprendida en el intervalo definido por las variables instancia `direccionInicioProceso` y `direccionFinProceso`. Después envía los mensajes apropiados para mostrar los nuevos valores de los registros.

`trazar:.` Es una acción invocada por el objeto de control cada vez que recibe un mensaje proveniente del botón *Trazar instruccion*. Su propósito es invocar al módulo `Ejecutar_Instruccion` para leer y ejecutar la instrucción binaria cuya dirección está almacenada en el registro `PC`. También envía mensajes a diferentes objetos para que muestren el valor de los registros que hayan sido modificados.

El único objeto dentro de la ventana de estadísticas que envía mensajes al objeto controlador es el botón *Inicializar*, que provoca la ejecución de la siguiente acción:

`inicializarEstadisticas:.` Tiene como propósito asignar el valor cero a cada elemento del arreglo `Estadisticas`.

Es posible establecer conexiones entre las celdas que componen el menú principal y el objeto controlador. Existe solo una conexión de este tipo dentro del sistema, establecida entre la celda *Info Panel...* (instancia de la clase **MenuCell**) del submenú *Info* y el objeto controlador. Como respuesta al mensaje enviado por esa celda el objeto controlador ejecuta la siguiente acción:

`showInfo:.` Su única finalidad es cargar el archivo **info.nib** en la memoria, esto provoca que se despliegue la ventana que contiene información sobre el sistema.

La implantación del simulador en el ambiente NeXTSTEP contiene, como todas las aplicaciones nativas de este ambiente, una instancia de la clase **Application** denominada `NXApp`. Este objeto tiene un conector llamado `delegate` que, mediante Interface Builder, se hace apuntar al objeto controlador. Una vez hecho lo anterior el objeto controlador se convierte en el *delegado* del objeto `NXApp`, el cual puede enviarle una variedad de mensajes de notificación específicos. Uno de estos mensajes es automáticamente enviado después de que el objeto `NXApp` termina la inicialización de la aplicación pero antes de recibir el primer evento. La acción que el objeto de control ejecuta en respuesta a este mensaje es la siguiente:

`appDidInit:.` Este método asigna valores iniciales a las variables instancia del objeto controlador y a algunos de los registros, asigna memoria dinámicamente para el arreglo que simula la memoria principal (`Memoria_SPARC`) y envía mensajes al objeto controlador para mostrar al usuario los valores iniciales de los registros.

Otro mensaje de notificación es enviado automáticamente por el objeto `NXApp` antes de terminar y eliminar la aplicación, al cual el objeto controlador responde ejecutando la siguiente acción:

`appWillTerminate:.` Esta acción libera el espacio de memoria asignado al arreglo `Memoria_SPARC`, mismo que fue reservado al inicio de la ejecución por la acción `appDidInit:`.

En la figura 7.17 observamos la forma en que se establece, mediante Interface Builder, la conexión entre una matriz de campos de texto y el objeto de control a través del conector `target`. En la figura se encuentran presentes la ventana para la manipulación de los programas, la ventana correspondiente al archivo **SPARCSimulator.nib** y el inspector. Este último indica el envio del mensaje `actualizacionIntervaloProceso:` al objeto de control en respuesta a la manipulación del objeto por el usuario.

7.4.2.4 Métodos adicionales

Los métodos restantes declarados en el archivo de interfaz de la clase son los siguientes:

`agregarTexto:en:.` Este método agrega una cadena (primer argumento) al final del texto contenido en la instancia de la clase **Text** apuntada por el segundo argumento.

`eliminarTextoEn:.` Este método elimina todo el texto contenido dentro de la instancia de la clase **Text** apuntada por su argumento.

Dentro del archivo de implantación se encuentran definidos varios métodos que no están declarados en el archivo de interfaz de la clase. Esto implica que no es posible enviar mensajes al objeto controlador que provoquen la ejecución de tales métodos fuera del archivo de implantación. Estos métodos, en su mayoría, funcionan como módulos de soporte que simplifican la definición de las acciones anteriormente discutidas. Como ejemplo describimos algunos de ellos a continuación:

`visualizarVentanaActual.` Hace uso del módulo `Leer_Registro_Entero` para leer el valor de cada uno de los registros que componen la ventana actual. Cada valor es visualizado en la celda correspondiente de la matriz apuntada por el conector `registrosEnteros`. Este método es invocado por las acciones `appDidInit:`, `cambiarVentanaActual:`, `ejecutar:` y `trazar:`.

`ensamblarDesdeArchivo.` Ensambla cada instrucción contenida en un archivo de código fuente mediante el uso del módulo `Ensamblar_Instruccion`, contenido dentro del núcleo. Las instrucciones binarias resultantes son almacenadas en memoria mediante el módulo `Escribir_a_Memoria`, que se encuentra también dentro del núcleo. Este método es invocado por la acción `ensamblar:`, simplificando su definición.

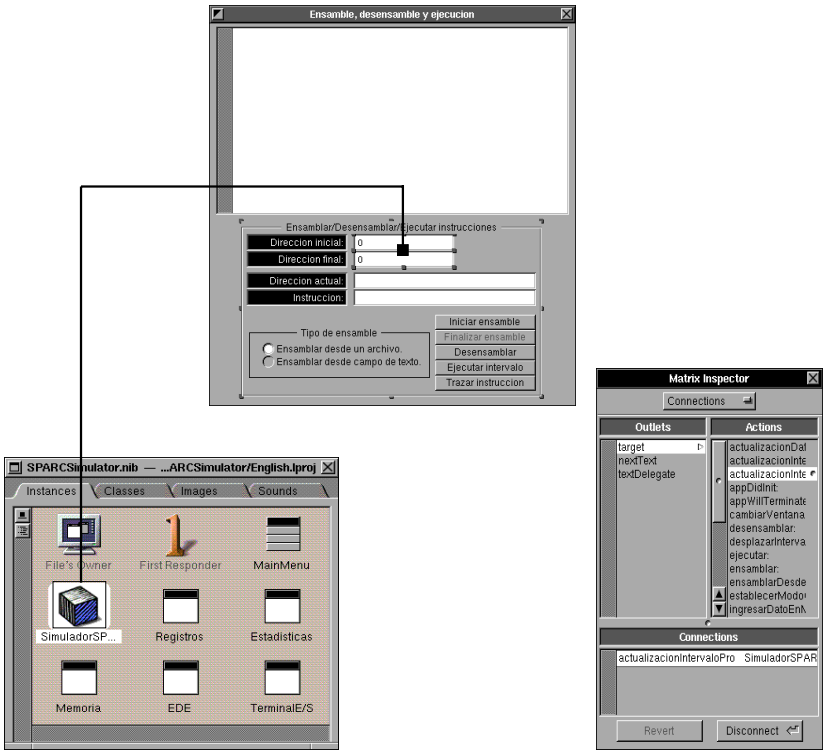


Figura 7.17: Conexión entre una matriz y el objeto controlador mediante Interface Builder

`actualizarMatrizIntervalo:`. Muestra el valor de las variables instancia `direccionInicioIntervalo`, `direccionFinIntervalo` y `caracter` en la celda correspondiente dentro de la matriz de campos de texto apuntada por el conector `matrizIntervalo` cuando se haya ingresado y verificado la validez de un nuevo valor para ellas. Es invocado cuando las acciones `ingresarDatoEnMemoria:`, `appDidInit:`, `actualizacionIntervalo:` y `desplazarIntervaloMemoria:` envían el mensaje respectivo al objeto controlador.

7.5 Extensiones para entrada y salida

En una implantación real de un sistema computacional basado en un procesador SPARC el acceso a los registros en los dispositivos de entrada y salida se realiza, principalmente, a través de las instrucciones de carga y almacenamiento privilegiadas [SPARC 1992]. Las direcciones en las que están mapeados tales registros son dependientes de una implantación particular. Sin embargo, debido a las limitaciones del sistema, no es posible seguir este esquema para satisfacer la necesidad del usuario de ingresar datos a un programa y recibir los resultados del mismo.

Para resolver este problema concebimos un esquema diferente que consiste en la extensión del conjunto de instrucciones de la arquitectura SPARC V8 con instrucciones adicionales, con las cuales es posible que un programa reciba y muestre datos a través de la consola mostrada en la figura 7.14 además de realizar operaciones en archivos. Debe ser obvio que estas instrucciones no pueden ser ejecutadas por alguna implantación de la arquitectura SPARC V8, sin embargo son muy útiles para el usuario del simulador.

Para ensamblar estas instrucciones se emplea el formato 3 definido por la arquitectura y mostrado

en la figura 3.8, teniendo especial cuidado de seleccionar valores del campo `op3` no usados por otras instrucciones que utilicen el mismo formato. Por otro lado, el requisito de portabilidad impide que el núcleo pueda ejecutar estas instrucciones, los principales motivos que originan esta situación son los siguientes:

- La ejecución de las instrucciones de entrada y salida mediante la consola es muy dependiente del esquema para el manejo de eventos del teclado en NeXTSTEP y de los mensajes enviados entre los objetos involucrados.
- Para ejecutar las instrucciones de lectura y escritura en archivos se deben realizar las llamadas al sistema, en Unix, correspondientes.

Como consecuencia del primer punto la ejecución de estas instrucciones se delega al objeto controlador, puesto que es el único que conoce a cada uno de los objetos que componen la interfaz. Sin embargo, es necesario resaltar que el núcleo puede ensamblar y desensamblar estas instrucciones. La cuestión de las llamadas al sistema no es grave puesto que pueden ser reemplazadas por invocaciones a los módulos contenidos en librerías proporcionadas por varios compiladores de C, en caso de que se requiera implantar las rutinas de servicio para las instrucciones de entrada y salida en algún sistema operativo distinto a Unix.

En la sentencia 53 del código del módulo `Ejecutar_Instruccion`, mostrado anteriormente, se observa que este regresa un valor específico cuando trata de interpretar una instrucción de entrada y salida. Cuando esto ocurre los métodos `ejecutar:` y `trazar:` envían un mensaje al objeto controlador que provoca la ejecución del método `ejecutarInstruccionES:`, el cual analiza el valor del campo `op3` de la instrucción y envía otro mensaje al objeto controlador para ejecutar el método `ejecutarInstruccionE::` si se trata de una instrucción de entrada desde la consola, `ejecutarInstruccionS::` en el caso de una instrucción de salida a la consola o `ejecutarInstruccionESArchivo:op1:op2:drd:` si se trata de una instrucción que realiza alguna operación sobre un archivo.

7.5.1 Procesamiento de eventos en NeXTSTEP

Para manejar una aplicación el usuario hace uso del teclado y el ratón, esta interacción con los dispositivos se traduce en mensajes que el Window Server envía a la instancia de la clase **Application** (`NXApp`) apropiada, tales mensajes se conocen como *eventos*. Toda la información referente al evento se proporciona al objeto `NXApp` almacenada dentro de una estructura cuya definición es la siguiente [Garfinkel y Mahoney 1993]:

```
typedef struct _NXEvent {
  int          type;       /* Tipo del evento */
  NXPoint      location;   /* Posicion del cursor del raton */
  long         time;       /* Tiempo desde el inicio */
  int          flags;      /* Shift, Control, Alt */
  unsigned int window;     /* Numero de la ventana del evento */
  NXEventData  data;       /* Datos (dependiente del tipo de evento) */
  DPSContext   ctxt;       /* Contexto PostScript */
} NXEvent, * NXEventPtr;
```

El primer campo de esta estructura describe el tipo de evento ocurrido, algunos de los valores que puede tomar esta variable están descritos por las siguientes macros:

```
/* Eventos del raton */
#define NX_LMOUSEDOWN     1
#define NX_LMOUSEUP       2
#define NX_RMOUSEDOWN     3
#define NX_RMOUSEUP       4
```

```
#define NX_MOUSEMOVED     5
#define NX_LMOUSEDRAGGED  6
#define NX_RMOUSEDRAGGED  7
#define NX_MOUSEENTERED   8
#define NX_MOUSEEXITED    9

/* Eventos del teclado */
#define NX_KEYDOWN        10
#define NX_KEYUP          11
#define NX_FLAGSCHANGED   12
```

Una vez recibido un evento, este debe ser dirigido al objeto que pueda procesarlo. Para lograrlo, el objeto `NXApp` envía un mensaje específico, dependiente del tipo de evento recibido, a la ventana apropiada. Las instancias de las clases **Window**, **View** y **Application** pueden responder a ese y otros mensajes similares debido a que dichas clases están derivadas de la clase abstracta **Responder**. Como ejemplo supongamos que el usuario presiona el botón izquierdo del ratón, `NXApp` recibe entonces un evento de tipo `NX_LMOUSEDOWN` que provoca el envío del mensaje `mouseDown:` a la ventana en la que se encontraba el cursor del ratón cuando ocurrió el evento. Una situación similar ocurre cuando se presiona el botón derecho, sin embargo en este caso el evento es de tipo `NX_RMOUSEDOWN` y el mensaje enviado a la ventana es `rightMouseDown:`. Otro evento ocurre cuando un botón deja de ser presionado, de tipo `NX_LMOUSEUP` o `NX_RMOUSEUP` según sea el caso, que provoca que el objeto `NXApp` envíe los mensajes `mouseUp:` y `rightMouseUp:` respectivamente.

Nuestra atención a partir de ahora se concentrará en los eventos originados en el teclado. Todos los eventos de tipo `NX_KEYDOWN` son enviados por el Window Server al objeto `NXApp` perteneciente a la aplicación activa, el cual a su vez los dirige, mediante el conector `keyWindow` y el mensaje `keyDown:` a la ventana que en ese momento esté en posibilidad de recibir eventos provenientes del teclado. Esta ventana a su vez envía el mensaje a un objeto de la clase **View**, contenido dentro de ella, a través de un conector llamado `firstResponder`, esto debido a que la ventana no puede procesar por si misma este tipo de eventos. Solo en caso de que el objeto referenciado por la variable instancia `firstResponder` no pueda responder al evento entonces lo envía a otra instancia de la clase **View** a través de un apuntador llamado `nextResponder`[8], si este objeto tampoco puede procesar el evento entonces lo envía a otro objeto **View** mediante su propia variable instancia `nextResponder` y así sucesivamente hasta que algún objeto responda o el mensaje regrese a la ventana, como consecuencia de que ningún objeto dentro de ella puede procesar el evento.

7.5.2 Instrucciones de entrada y salida a la consola

Las nuevas instrucciones realizan transferencias de información entre la memoria y la consola pero no por ello deben ser consideradas como instrucciones de carga y almacenamiento. La tabla 7.1 muestra los mnemónicos de las instrucciones, el valor asignado al campo `op3` para cada una y su correspondiente descripción.

Las instrucciones de entrada pueden leer, respectivamente, un caracter, un valor entero, un valor de punto flotante de presición sencilla o doble y una cadena. Los operandos de estas instrucciones pueden ser dos registros o un registro y un valor inmediato, que se emplean para calcular la dirección de memoria a partir de la cual se almacena el dato ingresado. De manera similar, existe una instrucción de salida por cada uno de los tipos de datos mencionados. En este caso los operandos de la instrucción se emplean para calcular la dirección donde se encuentra almacenada la información que será mostrada.

[8]Las clases **Window**, **View** y **Application** heredan de la clase **Responder** la variable instancia `nextResponder` cuyo propósito es establecer una *cadena de respuesta* entre varios objetos.

opcode	op3	Operación
inputc	101000	Leer un caracter
inputi	111000	Leer un valor entero
inputf	111010	Leer un valor de punto flotante de simple presición
inputd	111100	Leer un valor de punto flotante de doble presición
inputs	111110	Leer una cadena
outputc	101010	Mostrar un caracter
outputi	111001	Mostrar un valor entero
outputf	111011	Mostrar un valor de punto flotante de simple presición
outputd	111101	Mostrar un valor de punto flotante de doble presición
outputs	111111	Mostrar una cadena

Tabla 7.1: Instrucciones de lectura y escritura en la consola.

7.5.2.1 Implantación de las instrucciones de entrada desde la consola

El método `ejecutarInstruccionE::`, definido en el archivo **SimuladorSPARC.m**, es el encargado de implantar la ejecución de las instrucciones de entrada desde la consola. Es invocado por el método `ejecutarInstruccionES:` y recibe de él los parámetros `opcode` (el valor del campo `op3` de la instrucción binaria) y `direccion` (la dirección donde deberá almacenar el valor obtenido). El código de este método es el siguiente.

```
- ejecutarInstruccionE: (unsigned long) opcode : (unsigned long) direccion
{
 id                    texto;
 id                    ventanaES;
 int                   Resultado;
 unsigned char         Cadena[101];
 unsigned int          Longitud_1, Longitud_2;
 unsigned int          Bandera = 0;
 union Registro_Entero Entero;
 union Dato_Doble      Doble;
 union Dato_Simple     Simple;
 NXEvent               * Evento;

/* == Prepara la consola para recibir texto. == */
1   texto = [ terminalES docView ];
2   ventanaES = [ terminalES window ];
3   [ ventanaES makeKeyAndOrderFront: nil ];
4   [ texto setEditable: YES ];
5   [ texto setSelectable: YES ];
6   Longitud_1 = [ texto textLength ];
7   [ texto setSel: Longitud_1 : Longitud_1 ];
8   [ texto scrollSelToVisible ];
/* == Recibe los eventos del teclado que se produzcan. == */
9   [ ventanaES makeFirstResponder: texto ];
10  do
    {
11   Evento = [ NXApp getNextEvent: NX_KEYDOWNMASK ];
12   if ( Evento != NULL )
       {
13      if ( ( Evento -> data.key.charCode == 0x0D ) || ( Evento -> data.key.charCode == 3 ) )
```

```
14        Bandera = 1;
        else
15        [ NXApp sendEvent: Evento ];
       }
    } while ( Bandera == 0 );
/* == Obtiene la cadena de texto ingresada en la consola. == */
16  Longitud_2 = [ texto textLength ];
17  [ texto getSubstring: Cadena
            start: Longitud_1
            length: Longitud_2 - Longitud_1 + 1 ];
18  [ self agregarTexto: "\n" en: texto ];
/* == Ejecuta la instruccion indicada por el codigo de operacion. == */
19  switch ( opcode )
    {
     case 40 :     /* == inputc == */
20             Memoria_SPARC[direccion] = Cadena[0];
21             break;
     case 56 :     /* == inputi == */
22             Resultado = sscanf(Cadena, "%ld", & Entero.Con_Signo);
23             if ( Resultado == 1 )
24               Escribir_a_Memoria(Memoria_SPARC, direccion, Entero, 4);
25             break;
     case 58 :     /* == inputf == */
26             Resultado = sscanf(Cadena, "%f", & Simple.Real);
27             if ( Resultado == 1 )
                 {
28                Entero.Sin_Signo = Simple.Entero;
29                Escribir_a_Memoria(Memoria_SPARC, direccion, Entero, 4);
                 }
30             break;
     case 60 :     /* == inputd == */
31             Resultado = sscanf(Cadena, "%lf", & Doble.Real);
32             if ( Resultado == 1 )
                 {
#if BYTE_ORDER == LITTLE_ENDIAN
33                Entero.Sin_Signo = Doble.Entero[1];
34                Escribir_a_Memoria(Memoria_SPARC, direccion, Entero, 4);
35                Entero.Sin_Signo = Doble.Entero[0];
36                Escribir_a_Memoria(Memoria_SPARC, direccion + 4, Entero, 4);
#else
37                Entero.Sin_Signo = Doble.Entero[0];
38                Escribir_a_Memoria(Memoria_SPARC, direccion, Entero, 4);
39                Entero.Sin_Signo = Doble.Entero[1];
40                Escribir_a_Memoria(Memoria_SPARC, direccion + 4, Entero, 4);
#endif
                 }
41             break;
     case 62 :     /* == inputs == */
42             strcpy(Memoria_SPARC + direccion, Cadena);
    }
/* == Restablece los atributos del objeto texto en la consola. == */
43  [ texto setEditable: NO ];
44  [ texto setSelectable: NO ];
```

```
45  Estadisticas[43] ++;

46  return(self);
   }
```

La ventana que contiene la consola se coloca sobre las demás y se convierte en la ventana activa (*key window*) mediante el mensaje enviado en la sentencia 3. Los mensajes enviados en las sentencias 4–8 modifican los atributos del objeto texto, contenido en tal ventana, de tal modo que su contenido pueda ser seleccionado y editado por el usuario, esto provoca además que pueda recibir y procesar los eventos que se le enviarán posteriormente. Como resultado del mensaje enviado en la sentencia 9 se establece un enlace entre la ventana de la consola y el objeto texto dentro de ella a través del conector `firstResponder`, permitiendo que la ventana pueda despachar al objeto texto los eventos que recibe. A partir de la sentencia 10 comienza un ciclo que captura todos los eventos generados por la presión de las teclas, a excepción de dos todos ellos son regresados al objeto `NXApp` y eventualmente serán procesados por el objeto texto en la consola, almacenando y mostrando la información introducida por el usuario. Cuando el usuario termina de introducir caracteres pulsa la tecla `<ENTER>`, generando el evento que termina el ciclo. En las sentencias 16–18 el método obtiene la cadena ingresada por el usuario (la última contenida dentro del objeto texto), la cual después se transforma, de ser necesario, a un valor numérico que enseguida es almacenado en la memoria, todo esto en las sentencias 20–42.

7.5.2.2 Implantación de las instrucciones de salida a la consola

El método `ejecutarInstruccionES:` delega la responsabilidad de ejecutar instrucciones de salida a la consola al método `ejecutarInstruccionS::`, de manera similar que en el caso anterior. A continuación se muestra el código de dicho método, notar que recibe los mismos parámetros que el método anterior.

```
- ejecutarInstruccionS: (unsigned long) opcode : (unsigned long) direccion
{
 id                    texto;
 id                    ventanaES;
 unsigned char         Cadena[101];
 union Registro_Entero Entero;
 union Dato_Doble      Doble;
 union Dato_Simple     Simple;

/* == Prepara la consola para visualizar texto == */
1   texto = [ terminalES docView ];
2   ventanaES = [ terminalES window ];
3   [ ventanaES makeKeyAndOrderFront: self ];
/* == Construye la cadena que sera mostrada == */
4   switch ( opcode )
    {
     case 42 :     /* == outputc == */
5              Leer_Memoria(Memoria_SPARC, direccion, & Entero, 1);
6              sprintf(Cadena, "%c", (char) Entero.Sin_Signo);
7              break;
     case 57 :     /* == outputi == */
8              Leer_Memoria(Memoria_SPARC, direccion, & Entero, 4);
9              sprintf(Cadena, "%ld", Entero.Con_Signo);
10             break;
     case 59 :     /* == outputf == */
11             Leer_Memoria(Memoria_SPARC, direccion, & Entero, 4);
12             Simple.Entero = Entero.Sin_Signo;
```

```
13              sprintf(Cadena, "%f", Simple.Real);
14              break;
     case 61 :      /* == outputd == */
#if BYTE_ORDER == LITTLE_ENDIAN
15              Leer_Memoria(Memoria_SPARC, direccion, & Entero, 4);
16              Doble.Entero[1] = Entero.Sin_Signo;
17              Leer_Memoria(Memoria_SPARC, direccion + 4, & Entero, 4);
18              Doble.Entero[0] = Entero.Sin_Signo;
#else
19              Leer_Memoria(Memoria_SPARC, direccion, & Entero, 4);
20              Doble.Entero[0] = Entero.Sin_Signo;
21              Leer_Memoria(Memoria_SPARC, direccion + 4, & Entero, 4);
22              Doble.Entero[1] = Entero.Sin_Signo;
#endif
23              sprintf(Cadena, "%lf", Doble.Real);
24              break;
     case 63 :      /* == outputs == */
25              strcpy(Cadena, Memoria_SPARC + direccion);
    }
/* == Muestra la cadena en la consola == */
26  [ self agregarTexto: Cadena en: texto ];

27  Estadisticas[44] ++;

28  return(self);
   }
```

En las sentencias 1–3 son enviados los mensajes que colocan la ventana de la consola sobre las demás con el fin de que sea visible al usuario. La información a desplegar debe ser transformada, de ser necesario, en una cadena de caracteres antes de ser enviada al objeto de texto, los bloques comprendidos en las sentencias 5–25 realizan esta labor dependiendo de la instrucción que se tenga. Una vez que la cadena ha sido construida se despliega al final del objeto texto mediante el mensaje enviado en la sentencia 26.

7.5.3 Instrucciones para manipular archivos

Estas instrucciones permiten que los programas realicen operaciones en archivos. En la tabla 7.2 podemos observar los mnemónicos correspondientes a cada instrucción, el valor asignado al campo `op3` en cada una y una breve descripción.

opcode	op3	**Operación**
`create`	101011	Crea un nuevo archivo
`open`	101100	Abre un archivo
`read`	101101	Lee datos en un archivo
`write`	101110	Escribe datos en un archivo
`close`	101111	Cierra un archivo abierto
`lseek`	101001	Modifica el apuntador del archivo

Tabla 7.2: Instrucciones para realizar operaciones en archivos

El método `ejecutarInstruccionESArchivo:op1:op2:drd:` lleva a cabo la ejecución, su labor consiste simplemente en determinar la instrucción y efectuar la llamada al sistema en Unix correspondiente.

Los parámetros pueden variar mucho entre las instrucciones, a continuación proporcionamos una descripción más detallada de cada una:

Instrucción `create`. Crea un nuevo archivo o trunca un archivo existente. Su primer parámetro es la dirección de la cadena que almacena el nombre y ruta del archivo[9]. El segundo parámetro es un registro entero, indicado por el campo `rd` de la instrucción, que tiene almacenado los permisos del nuevo archivo. Como resultado de la ejecución el registro `rd` almacenará el descriptor del nuevo archivo si este fue creado exitosamente o, en caso contrario, el valor -1. Para ejecutar esta instrucción se utiliza la llamada al sistema `creat`.

Instrucción `open`. Abre un archivo. Su primer argumento es la dirección de la cadena que almacena el nombre y ruta del archivo, el segundo parámetro es un registro entero, indicado por el campo `rd`, que almacena el modo de acceso al archivo[10]. Al término de la ejecución el registro `rd` almacena el descriptor del archivo o el valor -1. La implantación se lleva a cabo mediante la llamada al sistema `open`.

Instrucción `read`. Lee un bloque de información desde un archivo. El primer parámetro es la dirección en memoria a partir de la cual serán almacenados los datos, el segundo parámetro es el registro (`rd`) que contiene el descriptor del archivo. Estra instrucción asume que el registro `%g1` almacena la longitud del bloque a leer. Si la lectura es exitosa la instrucción almacenará en el registro `rd` el número de bytes leidos, el valor cero en caso de que se alcance el fin de archivo o -1 en caso de algún error. Para su ejecución se emplea la llamada al sistema `read`.

Instrucción `write`. Almacena un bloque de información en un archivo. El primer parámetro es la dirección del bloque de memoria que será almacenado, el registro indicado por el campo `rd` almacena el descriptor del archivo y se presupone que el registro `%g1` contiene el número de bytes que serán escritos. Después de su ejecución la instrucción almacena en el registro `rd` el número de bytes almacenados realmente o, en caso de error, el valor -1. La ejecución se implanta a través de la llamada al sistema `write`.

Instrucción `close`. Cierra un archivo. Su único parámetro es el descriptor del archivo a cerrar almacenado en el registro especificado por el campo `rd` de la instrucción. Si la operación fue exitosa la instrucción almacena en tal registro el valor cero, de otro modo almacena el valor -1. Se implanta mediante la llamada al sistema `close`.

Instrucción `lseek`. Modifica el apuntador asociado a un archivo. Tiene tres parámetros, el primero especifica un desplazamiento en bytes, almacenado en el registro indicado por el campo `rs1`, relativo al origen especificado en el segunto parámetro, el cual puede ser un registro entero o un valor inmediato[11], el tercer argumento es la dirección del registro entero que contiene el descriptor del archivo. Para la ejecución de esta instrucción se emplea la llamada al sistema `lseek`.

[9] Esta dirección se calcula mediante alguno de los dos modos de direccionamiento tradicionales.

[10] Los valores de este registro pueden ser: 0 = solo lectura, 1 = solo escritura y 2 = lectura y escritura.

[11] Los valores de este parámetro son los siguientes: 0 indica el inicio del archivo, 1 indica la posición actual del apuntador y 2 indica el fin del archivo.

Capítulo 8

Evaluación de resultados

8.1 Introducción

El objetivo de este capítulo es demostrar la utilidad del simulador mediante la ejecución de algunos programas escritos en lenguaje ensamblador SPARC que implantan diversos algoritmos. Los ejemplos utilizados incrementan su dificultad gradualmente, primero se revisa un sencillo algoritmo iterativo para calcular el valor

$$\sum_{i=1}^{n} i,$$

y, a continuación, se realizan pruebas con dos programas que implantan algoritmos de ordenamiento.

En las siguientes secciones se presentan comentarios referentes a la implantación de los algoritmos, así como el análisis de los resultados proporcionados por el simulador al término de la ejecución de los programas. Estos resultados son cuantitativos respecto al tipo de instrucciones ejecutadas, el tiempo de ejecución no es reportado puesto que este puede variar aún cuando un mismo programa sea ejecutado más de una vez sobre una misma instancia del problema. Lo anterior se debe a que el simulador se ejecuta como una aplicación que compite por los recursos en un ambiente multitarea.

8.2 Algoritmo de sumatoria

Desarrollamos el primer programa compilando, de manera manual, los siguientes procedimientos escritos en lenguaje C.

```
#include <stdio.h>

unsigned int sumatoria(unsigned int n)
{
 unsigned int   a = 0;
 unsigned int   i;

 for ( i = 1; i <= n; i ++ )
    a += i;
 return(a);
}

void main(void)
{
```

```
 unsigned int   n;
 unsigned int   a;

 scanf(``%d'', & n);
 a = sumatoria(n);
 printf(``%d'', a);
}
```

Para construir la secuencia de instrucciones en lenguaje ensamblador correspondiente al módulo `void()` empleamos, entre otras, las instrucciones `inputi` y `outputi` para realizar entrada y salida. El valor de la variable entera `n` es ingresado por el usuario y después proporcionado como argumento a la función `sumatoria()`, lo cual indica la lectura de un valor entero mediante la consola, su almacenamiento en alguna dirección de memoria y su posterior transferencia hacia el registro `%o0` de la ventana actual. El valor devuelto por la función `sumatoria()` se almacena en la variable `a` y posteriormente es enviado a la salida estándar, dicho valor estará almacenado en el registro `%o1` al término de la ejecución de tal procedimiento, después se puede realizar su almacenamiento en memoria y posteriormente la visualización en la consola. El código del módulo `void()` supone que la dirección de la variable `n` está especificada en el registro `%l0` de la ventana de registros correspondiente.

El mapeo de las variables del módulo `sumatoria()` en los registros es simple. Cuando dicho procedimiento es invocado se le asigna una ventana de registros propia, en la cual se encuentra almacenado el valor del parámetro `n` en el registro `%i0`. El valor de la variable acumulador `a` debe ser regresado al término de la ejecución, esto sugiere que dicha variable sea almacenada en alguno de los registros `%i0`–`%i7`, finalmente nos decidimos por el registro `%i1`. La variable contador `i` la asignamos en el primer registro local disponible, es decir, el registro `%l0`. Una versión inicial del programa en lenguaje ensamblador es la siguiente.

```
Programa 1:

%i0 : n
%i1 : a
%l0 : i

1  sumatoria: save     %g0, 0, %g0         : decrementa CWP
2             add      %g0, 0, %i1         : a = 0;
3             add      %g0, 1, %l0         : i = 1;
4  ciclo:     subcc    %l0, %i0, %g0       : i <= n;
5             bgu      regreso
6             sethi    0, %g0              : no operacion
7             add      %i1, %l0, %i1       : a += i;
8             add      %l0, 1, %l0         : i ++;
9             ba       ciclo
10            sethi    0, %g0              : no operacion
11 regreso:   restore  %g0, 0, %g0         : incrementa CWP
12            jmpl     %o7 + 8, %g0        : return();
13            sethi    0, %g0              : no operacion

%o0 : n
%o1 : a

14 main:      inputi   %g0 + %l0           : scanf(``%d'', & n);
15            ld       [ %g0 + %l0 ], %o0
16            call     sumatoria           : a = sumatoria(n);
17            sethi    0, %g0              : no operacion
```

```
18            st       %o1, [ %g0 + %l0 ]
19            outputi  %g0 + %l0           : printf(``%d'', a);
```

Podemos realizar algunas modificaciones en la secuencia anterior con el fin de aprovechar el espacio de las instrucciones de retardo, que en este caso está ocupado por instrucciones "no operación", con la finalidad de producir un programa más corto, que requiera menos ciclos de instrucción y por lo tanto menos tiempo de ejecución. Como resultado de tales modificaciones obtenemos la nueva secuencia mostrada a continuación.

```
Programa 2:

%i0 : n
%i1 : a
%l0 : i

1  sumatoria: add      %g0, 0, %i1         : a = 0;
2             add      %g0, 1 , %l0        : i = 1;
3  ciclo:     subcc    %l0, %i0, %g0       : i <= n;
4             bgu      regreso
5             sethi    0, %g0              : no operacion
6             add      %i1, %l0, %i1       : a += i;
7             ba       ciclo
8             add      %l0, 1, %l0         : i ++;
9  regreso:   jmpl     %i7 + 8, %g0        : return();
10            restore  %g0, 0, %g0         : incrementa CWP

%o0 : n
%o1 : a

11 main:      inputi   %g0 + %l0           : scanf(``%d'', & n);
12            ld       [ %g0 + %l0 ], %o0
13            call     sumatoria           : a = sumatoria(n);
14            save     %g0, 0, %g0         : decrementa CWP
15            st       %o1, [ %g0 + %l0 ]
16            outputi  %g0 + %l0           : printf(``%d'', a);
```

La instrucción `sethi` en la línea 10 del programa 1 puede ser reemplazada por la instrucción `add` en la línea 8, puesto que la ejecución de la instrucción `ba` en la línea 9 no depende de la ejecución de la instrucción 8. Tampoco existe dependencia entre la instrucción de transferencia de control en la línea 12 y la instrucción `restore` en la línea 11, por lo que esta instrucción puede sustituir a la instrucción `sethi` de la línea 13. La instrucción de retardo `sethi` en la línea 17 pudo ser sustituida por la instrucción de carga en la línea 15, sin embargo nos dimos cuenta que colocar la instrucción `save` como instrucción de retardo porporciona una mejor comprensión del flujo del programa. Podría pensarse que es correcto reemplazar la instrucción `sethi` en la línea 6 con la instrucción `add` en la línea 7, sin embargo, esto provocaría que al final del ciclo se incrementara, incorrectamente, el valor del registro `%i1` en `n + 1`.

Ambos programas fueron ejecutados en el simulador para distintos valores de `n`, los resultados arrojados por el sistema se muestran en la tabla 8.1 para el programa 1 y en la tabla 8.2 para el programa 2. Excepto por la línea correspondiente a la instrucción `sethi` y la que muestra el número total de instrucciones ejecutadas por cada instancia del problema las dos tablas contienen los mismos resultados, sin embargo es claro que las modificaciones que dieron origen al programa 2 se reflejan en un número menor de instrucciones ejecutadas y, por lo tanto, en menos ciclos de instrucción y un tiempo menor de ejecución. El lector puede comprobar fácilmente que, para cada valor de la variable `n`, el número de

instrucciones ejecutadas por el programa 2 disminuye en poco más de un 14% con respecto al número total de instrucciones ejecutadas por el programa 1.

Tipo de instrucción	n = 10	n = 100	n = 1000	n = 10000
`ld`	1	1	1	1
`st`	1	1	1	1
`sethi`	23	203	2003	20003
`add`	22	202	2002	20002
`sub`	11	101	1001	10001
`save`	1	1	1	1
`restore`	1	1	1	1
`ba`	10	100	1000	10000
`Bicc`	11	101	1001	10001
`call`	1	1	1	1
`jmpl`	1	1	1	1
`input`	1	1	1	1
`output`	1	1	1	1
Total:	85	715	7015	70015

Tabla 8.1: Resultados de la ejecución del programa 1 proporcionados por el simulador

Tipo de instrucción	n = 10	n = 100	n = 1000	n = 10000
`ld`	1	1	1	1
`st`	1	1	1	1
`sethi`	11	101	1001	10001
`add`	22	202	2002	20002
`sub`	11	101	1001	10001
`save`	1	1	1	1
`restore`	1	1	1	1
`ba`	10	100	1000	10000
`Bicc`	11	101	1001	10001
`call`	1	1	1	1
`jmpl`	1	1	1	1
`input`	1	1	1	1
`output`	1	1	1	1
Total:	73	613	6013	60013

Tabla 8.2: Resultados de la ejecución del programa 2 proporcionados por el simulador

8.3 Algoritmos de ordenamiento

En esta sección presentamos los resultados obtenidos después de analizar la información proporcionada por el simulador al término de la ejecución de dos programas que implantan, respectivamente, el algoritmo de ordenamiento por *intercambio directo* (*burbuja*) y el algoritmo de ordenamiento por *intercambio con incrementos decrecientes* (*Shell*).

8.3.1 Algoritmo burbuja

A continuación se muestra el código, en lenguaje C, de un procedimiento que ordena `n` valores de punto flotante almacenados en un arreglo `X` mediante el método burbuja, seguido del código del módulo `main` que invoca a dicho procedimiento.

```
#include <stdio.h>

unsigned int    n = 50;
float           X[50] = { ... };

void burbuja(float * X, unsigned int n)
{
 unsigned int    i, j;
 unsigned int    Bandera = 1;
 float           Auxiliar;

 for ( i = 0; ( i < n - 1 ) && ( Bandera == 1 ); i ++ )
    {
     Bandera = 0;
     for ( j = 0; j < ( n - 1 ) - i; j ++ )
        if ( X[j] > X[j + 1] )
          {
           Bandera = 1;
           Auxiliar = X[j];
           X[j] = X[j + 1];
           X[j + 1] = Auxiliar;
          }
    }
}

void main(void)
{
 unsigned int    i;

 burbuja(X, n);
 for ( i = 0; i < n; i ++ )
    printf(``%f\n'', X[i]);
}
```

La siguiente secuencia de instrucciones en lenguaje ensamblador es consecuencia de una compilación manual de los módulos presentados anteriormente. En la tabla 8.3 se muestran los resultados obtenidos en el simulador después de la ejecución de dicho programa sobre varios arreglos de distinta longitud.

```
Programa 3:

%i0 : X
%i1 : n
%l0 : i
%l1 : j
%l2 : Bandera
%f0 : X[j]
%f1 : X[j + 1]

1  burbuja:     sub     %i1, 1, %i1
```

```
2               add     %g0, 1, %l2          : Bandera = 1;
3               add     %g0, 0, %l0          : i = 0;
4  ciclo_1:     subcc   %l0, %i1, %g0        : i < n - 1
5               bcc     regreso
6               subcc   %l2, 1, %g0          : Bandera == 1;
7               bne     regreso
8               add     %g0, 0, %l2          : Bandera = 0;
9               add     %g0, 0, %l1          : j = 0;
10              sub     %i1, %l0, %l3
11 ciclo_2:     subcc   %l1, %l3, %g0        : j < ( n - 1 ) - i;
12              bcc     fin_ciclo_1
13              sll     %l1, 2, %l4
14              ld      [ %i0 + %l4 ], %f0   : Carga X[j] en %f0
15              add     %l4, 4, %l4
16              ld      [ %i0 + %l4 ], %f1   : Carga X[j + 1] en %f1
17              fcmps   %f0, %f1             : X[j] > X[j + 1]
18              fble    fin_ciclo_2
19              sethi   0, %g0               : no operacion
20              add     %g0, 1, %l2          : Bandera = 1;
21              st      %f0, [ %i0 + %l4 ]   : intercambio
22              sub     %l4, 4, %l4
23              st      %f1, [ %i0 + %l4 ]
24 fin_ciclo_2: ba      ciclo_2
25              add     %l1, 1, %l1          : j ++;
26 fin_ciclo_1: ba      ciclo_1
27              add     %l0, 1, %l0          : i ++;
28 regreso:     add     %i1, 1, %i1
29              jmpl    %i7 + 8, %g0
30              restore %g0, 0, %g0          : incrementa CWP

%o0 : X
%o1 : n
%l0 : i

31 main:        call    burbuja              : burbuja(X, n);
32              save    %g0, 0, %g0          : decrementa CWP
33              add     %g0, H0A, %l0
34              stb     %l0, [ %g0 + H100 ]
35              add     %g0, 0, %l0          : i = 0;
36 ciclo:       subcc   %l0, %o1, %g0        : i < n;
37              bcc     fin
38              sll     %l0, 2, %l1
39              outputf %o0 + %l1            :  printf(``%f\n'', X[i]);
40              outputs %g0 + H100
41              ba      ciclo
42              add     %l0, 1, %l0          : i ++;
43 fin:         ...
```

No es necesario asignar un registro para la variable `Auxiliar`, declarada dentro del procedimiento `burbuja`, debido a que el intercambio puede realizarse transfiriendo cada valor desde el archivo de registros de punto flotante a la nueva posición dentro del arreglo. La instrucción 20 debe ser ejecutada únicamente cuando %`f0` > %`f1`, colocarla como instrucción de retardo de la instrucción 18 provocaría un asignamiento indebido cuando %`f0` ≤ %`f1`, ocasionando por lo tanto una ejecución incorrecta del programa.

Analizando las cifras en la tabla 8.3 y el código del programa 3 podemos darnos cuenta de ciertos

	n = 50	n = 100	n = 500	n = 1000	n = 5000	n = 10000
Tipos de instrucción						
ld	2378	9880	248240	998844	24976910	99985968
st	1061	5465	129047	520587	12512397	48989211
sethi	1189	4940	124120	499422	12488455	49992984
sll	1281	5136	125085	501410	12498321	50012921
add	3087	13003	314661	1263104	31252709	12452038
sub	1937	8156	191003	764667	18769117	74537337
save	1	1	1	1	1	1
restore	1	1	1	1	1	1
ba	1280	5135	125084	501409	12498320	50012920
Bicc	1365	5328	126015	503386	12508053	50032795
fble	1189	4940	124120	499422	12488455	49992984
call	1	1	1	1	1	1
jmpl	1	1	1	1	1	1
fcmps	1189	4940	124120	499422	12488455	49992984
output	100	200	1000	2000	10000	20000
Valores de registros						
%l0 (i)	41	95	464	987	4865	9986

Tabla 8.3: Resultados de la ejecución del programa 3 proporcionados por el simulador

aspectos interesantes referentes a la ejecución del programa, los cuales se enuncian enseguida y se ejemplifican para el caso de n = 50, sin embargo el lector puede verificarlos para los valores restantes de la variable n:

- El procedimiento ilustrado anteriormente ordena un arreglo a lo más en $n-1$ pasos, sin embargo el valor del registro %l0, que corresponde a la variable i, indica que para todos los arreglos y valores de la variable n considerados la secuencia se encuentra ordenada en menos de $n-1$ pasos.

- La expresión $\frac{2kn-k^2-k}{2}$ proporciona el número de comparaciones realizadas por el algoritmo burbuja después de ordenar un arreglo de n elementos en k pasos [Tenenbaum et al. 1993]. Sustituyendo en tal expresión k y n por %l0 = 41 y n = 50, respectivamente, obtenemos un total de 1189 comparaciones, lo cual coincide con el número de instrucciones fcmps ejecutadas por el programa.

- Las instrucciones de carga 14 y 16 en el programa 3 obtienen del arreglo los valores a ser comparados en cada iteración del contador j. Como consecuencia de esto para cada valor de n el número total de instrucciones de carga ejecutadas dividido entre dos es igual al número de instrucciones fcmps ejecutadas.

- El programa 3 realiza el intercambio de elementos en el arreglo mediante las instrucciones de almacenamiento 21 y 23. En el caso del arreglo de 50 elementos tenemos un total de 1061 instrucciones de almacenamiento ejecutadas, inspeccionando el código nos damos cuenta de que la instrucción 34 fue ejecutada una sola vez mientras que cada una de las instrucciones 21 y 23 fueron ejecutadas 530 veces, por lo tanto el programa realizó, en ese caso, 530 intercambios.

8.3.2 Algoritmo Shell

La versión del algoritmo Shell que se presenta a continuación particiona, mediante un incremento, un arreglo de n elementos en subarreglos más pequeños y los ordena mediante el algoritmo burbuja. En

cada iteración el incremento es, aproximadamente, la mitad del anterior hasta que, eventualmente, el incremento es uno. El código en lenguaje C del procedimiento de ordenamiento y del módulo `main` que lo invoca es el siguiente.

```
#include <stdio.h>

unsigned int    n = 50;
float           X[50] = { ... };

void shell(float * X, unsigned int n)
{
 float           Auxiliar;
 unsigned int    i, Limite;
 unsigned int    Bandera;
 unsigned int    k = n;

 while ( k > 1 )
    {
     k /= 2;
     Limite = ( n - 1 ) - k;
     do
     {
      Bandera = 0;
      for ( i = 0; i <= Limite; i ++ )
         if ( X[i] > X[i + k] )
           {
            Bandrea = 1;
            Auxiliar = X[i];
            X[i] = X[i + k];
            X[i + k] = Auxiliar;
           }
     } while ( Bandera == 1 );
    }
}

void main(void)
{
 unsigned int    i;

 shell(X, n);
 for ( i = 0; i < n; i ++ )
    printf(``%f\n'', X[i]);
}
```

Un proceso de compilación similar al realizado en el caso anterior produce el siguiente programa en lenguaje ensamblador. Este programa fue ejecutado por el simulador sobre el mismo conjunto de arreglos utilizado en las pruebas del algoritmo burbuja, los resultados obtenidos esta vez se muestran en la tabla 8.4.

```
Programa 4:

%l0 : X
%i1 : n
%l0 : i
```

```
%l1 : Limite
%l2 : k
%l3 : Bandera
%f0 : X[i]
%f1 : X[i + k]

1  shell:       add     %g0, %i1, %l2        : k = n;
2               sub     %i1, 1, %i1
3  ciclo_1:     subcc   %l2, 1, %g0          : k > 1
4               bleu    regreso
5               srl     %l2, 1, %l2          : k /= 2;
6               sub     %i1, %l2, %l1        : Limite = ( n - 1 ) - k;
7  ciclo_2:     add     %g0, 0, %l3          : Bandera = 0;
8               add     %g0, 0, %l0          : i = 0;
9  ciclo_3:     subcc   %l0, %l1, %g0        : i <= Limite;
10              bgu     fin_ciclo_2
11              sll     %l0, 2, %l4
12              ld      [ %i0 + %l4 ], %f0   : Carga X[i] en %f0
13              add     %l0, %l2, %l4
14              sll     %l4, 2, %l4
15              ld      [ %i0 + %l4 ], %f1   : Carga X[i + k] en %f1
16              fcmps   %f0, %f1             : X[i] > X[i + k]
17              fble    fin_ciclo_3
18              sethi   0, %g0               : no operacion
19              add     %g0, 1, %l3          : Bandera = 1;
20              st      %f0, [ %i0 + %l4 ]   : intercambio
21              sll     %l0, 2, %l4
22              st      %f1, [ %i0 + %l4 ]
23 fin_ciclo_3: ba      ciclo_3
24              add     %l0, 1, %l0          : i ++;
25 fin_ciclo_2: subcc   %l3, 1, %g0          : Bandera == 1;
26              be      ciclo_2
27              sethi   0, %g0               : no operacion
28              ba      ciclo_1
29              sethi   0, %g0               : no operacion
30 regreso:     add     %i1, 1, %i1
31              jmpl    %i7 + 8, %g0
32              restore %g0, 0, %g0          : incrementa CWP

%o0 : X
%o1 : n
%l0 : i

33 main:        call    shell                : shell(X, n);
34              save    %g0, 0, %g0          : decrementa CWP
35              add     %g0, H0A, %l0
36              stb     %l0, [ %g0 + H100 ]
37              add     %g0, 0, %l0          : i = 0;
38 ciclo:       subcc   %l0, %o1, %g0        : i < n;
39              bcc     fin
40              sll     %l0, 2, %l1
41              outputf %o0 + %l1            : printf(``%f\n'', X[i]);
42              outputs %g0 + H100
43              ba      ciclo
```

```
44              add     %l0, 1, %l0          : i ++;
45 fin:         ...
```

	n = 50	n = 100	n = 500	n = 1000	n = 5000	n = 10000
Tipos de instrucción						
ld	2224	6208	46796	117286	1064292	2527222
st	389	937	6215	15419	123117	300851
sethi	1142	3144	23456	58714	532268	1263754
sll	2494	6811	50454	126058	1130961	2687778
srl	6	7	9	10	13	14
add	2522	6848	50507	126123	1131074	2687911
sub	1225	3287	24017	59788	537393	1273900
save	1	1	1	1	1	1
restore	1	1	1	1	1	1
ba	1167	3210	23906	59652	537158	1273624
Bicc	1219	3280	24008	59778	537380	1273886
FBfcc	1112	3104	23398	58643	532146	1263611
call	1	1	1	1	1	1
jmpl	1	1	1	1	1	1
Fcmp	1112	3104	23398	58643	532146	1263611
output	100	200	1000	2000	10000	20000
Valores de registros						
%l0 (i)	49	99	499	999	4999	9999
%l1 (Limite)	48	98	498	998	4998	9998
%l2 (k)	0	0	0	0	0	0
%l3 (Bandera)	0	0	0	0	0	0

Tabla 8.4: Resultados de la ejecución del programa 4 proporcionados por el simulador

Nuevamente los intercambios pueder ser efectuados adecuadamente mediante el uso de las instrucciones de almacenamiento 20 y 22, esto evita reservar un registro para la variable `Auxiliar`. La sustitución de la instrucción de retardo 18 por la instrucción 19 provocaría el mismo tipo de asignamiento inválido comentado en el caso del programa anterior, para evitarlo es necesario colocar una instrucción "no operación". Con el fin de garantizar la ejecución correcta del programa, es necesario colocar una instrucción "no operación" como instrucción de retardo de la instrucción 26, en vez de colocar en dicha posición la instrucción de transferencia de control siguiente.

Las conclusiones sobre la ejecución del programa, obtenidas analizando los datos de la tabla 8.4, se enuncian en los párrafos siguientes. Nuevamente los comentarios se ejemplifican para el caso de `n` = 50, pero es sencillo comprobar su validez para los valores restantes de `n`:

- El valor del registro `%l2`, que almacena el incremento `k`, fue modificado por la instrucción 5 en seis ocasiones, como lo muestra la tabla 8.4. En las primeras cinco dicho valor es empleado por el resto de las instrucciones que componen el cuerpo del ciclo `while` externo, comprendido entre las instrucciones 3 y 29 del programa 4. El último valor asignado al registro `%l2` por la instrucción 5 es cero, sin embargo dicha modificación se realiza inmediatamente después de que la instrucción 4 transfiere el control a la instrucción 30 y, por lo tanto, no tiene efecto en los resultados.

- El número de comparaciones realizadas es igual al número de veces que fue ejecutada la instrucción `fcmps`, 1112 en este caso. El número de intercambios se obtiene tomando de la tabla 8.4 el número

de instrucciones de almacenamiento ejecutadas, restándo uno a este valor y dividiendo el resultado entre dos. En el caso del arreglo de 50 elementos tenemos un total de 194 intercambios.

- En el último paso `k` = 1 y, por lo tanto, el arreglo entero es ordenado mediante el método burbuja hasta que no ocurra intercambio alguno. Esto explica porqué el valor final del registro `%l3`, donde se encuentra asignada la variable `Bandera`, es cero, así como los valores de los registros `%l0` y `%l1`, que almacenan, respectivamente, los valores de las variables `i` y `Limite`.

A partir de los valores mostrados en las tablas 8.3 y 8.4 podemos determinar el porcentaje en que se reducen el número de comparaciones y de intercambios del algoritmo Shell propuesto con respecto a los del algoritmo burbuja para cada uno de los arreglos considerados. Estos porcentajes se muestran en la tabla 8.5. Se observa que conforme el valor de `n` aumenta se obtiene un mejor rendimiento en el algoritmo Shell que en el algoritmo burbuja.

	`n` = 50	`n` = 100	`n` = 500	`n` = 1000	`n` = 5000	`n` = 10000
Comparaciones	6.48%	37.17%	81.15%	88.26%	95.74%	97.47%
Intercambios	63.40%	17.13%	95.18%	97.04%	99.02%	99.39%

Tabla 8.5: Porcentaje de reducción del número de comparaciones e intercambios del algoritmo Shell con respecto al algoritmo burbuja

La causa de la reducción en el número de intercambios y comparaciones es la siguiente. Conforme el incremento empleado por el algoritmo Shell se hace más pequeño y se ordenan los subarreglos correspondientes, los elementos del arreglo completo se acercan cada vez más a la posición que les corresponde. Esto provoca que para cada incremento se realizen menos pasadas a los subarreglos para ordenarlos, lo que se traduce en un menor número de comparaciones e intercambios. Cuando el incremento se hace finalmente uno el arreglo completo se encuentra casi ordenado y, por lo tanto, se requieren mucho menos iteraciones para ordenarlo que las que son necesarias para el algoritmo burbuja.

En este capítulo enfocamos nuestra atención únicamente en tres algoritmos de cientos que pueden ser implantados y ejecutados por el simulador. Adicionalmente desarrollamos y verificamos programas en las siguientes áreas:

- Algoritmo de Gauss con sustitución regresiva para encontrar la solución de sistemas de ecuaciones lineales.
- Algoritmos para realizar operaciones con matrices.
- Algoritmos de búsqueda secuencial y binaria.
- Algoritmos de ordenamiento por inserción y selección directa.

Algunos problemas interesantes que pueden ser abordados son la búsqueda de formas eficientes para implantar estructuras de datos simples como listas, colas y pilas así como estructuras de datos complejas como árboles y grafos. También se pueden buscar métodos para construir algoritmos que simulen recursividad, por ejemplo.

Capítulo 9

Conclusiones

Este documento describió a la simulación de conjunto de instrucciones como una herramienta eficaz para estudiar el comportamiento de los cada vez más complejos sistemas de computación, además es una excelente alternativa para llevar a buen término los proyectos de desarrollo limitados por la carencia de infraestructura. Con la finalidad de exponer algunas ideas introductorias involucradas en el diseño e implantación de esta clase de sistemas describimos con un buen nivel de detalle las características de SPARCSim, nuestro simulador del conjunto de instrucciones de la arquitectura SPARC V8 como una aplicación del ambiente NeXTSTEP.

Con la finalidad de que la presente disertación funcione como una fuente de información y de fundamentar la documentación de diseño e implantación del sistema, se presentó una breve discusión sobre las características de RISC, información introductoria sobre el conjunto de instrucciones y otros rasgos importantes de la arquitectura SPARC V8, también se discutieron el ambiente NeXTSTEP junto con sus herramientas de desarrollo y el lenguaje Objective C.

Casi al término de este reporte de tesis presentamos los resultados obtenidos por el sistema después de la ejecución en el simulador de algunos programas. Estos resultados numéricos nos permitieron verificar que el comportamiento de los algoritmos implantados era el esperado, también logramos determinar el impacto en el rendimiento al realizar modificaciones a un programa.

Consideramos que los objetivos planteados desde el principio fueron cumplidos satisfactoriamente. La investigación que dio origen a este documento nos permitió conocer y comprender tecnologías alternativas e innovadoras a las empleadas tradicionalmente. El tiempo invertido en investigar, programar y verificar fue bastante productivo y nos condujo a una mejor concepción de la arquitectura de procesadores, la programación orientada a objetos, los sistemas operativos y la simulación.

Hemos probado la efectividad y utilidad de la herramienta que desarrollamos, sin embargo el trabajo puede ser continuado y extendido de las siguientes maneras:

- Implantación las instrucciones para manipulación de trampas y de aritmética de punto flotante de 128 bits.
- Desarrollo de modelos para el resto de los componentes que conforman los sistemas actuales de computación, con la finalidad de construir un simulador de máquina completo para soportar la ejecución del sistema operativo y sus aplicaciones.
- Implantación de módulos que recopilen la siguiente información:
    1. Número de transferencias de control hacia una instrucción particular.
    2. Número de transferencias de control desde una instrucción particular.
    3. Número de veces en que cada instrucción de un intervalo de memoria fue ejecutada.

En caso de simular los módulos de memoria caché es posible obtener la siguiente información:

1. Número de fallos en el acceso de lectura al caché de instrucciones.
2. Número de fallos en el acceso de escritura al caché de datos.
3. Número de fallos en el acceso de lectura al caché de datos.
4. Número de fallos en el acceso de lectura al TLB.

- Simulación de las diferentes etapas de pipeline para la ejecución de las instrucciones.
- Soporte para instrucciones sintéticas en el proceso de ensamble.
- Una interfaz gráfica para el núcleo de SPARCSim puede ser desarrollada en diversos ambientes para producir un sistema completamente funcional. Es posible utilizar MFC y Visual C++ para construir el sistema en ambiente Windows, o bien Sun Workshop para el ambiente Solaris.

Bibliografía

[Balderas y García 1999] Balderas, T., y H. García. 1999. Desarrollo de un sistema simulador de la arquitectura SPARC mediante el sistema operativo Mach y el ambiente NeXTSTEP. En *Memorias del Segundo Encuentro Nacional de Computación. ENC'99*. ed. J. M. Ahuactzin. Universidad Autónoma del Estado de Hidalgo, Pachuca, Hidalgo, México.

[Bedichek 1990] Bedichek, R. 1990. Some efficient architecture simulation techniques. En *Proceedings of Winter '90 USENIX Conference*.

[Garfinkel y Mahoney 1993] Garfinkel, S. L., y M. K. Mahoney. 1993. *NeXTSTEP programming. Step one: object-oriented applications*. New York: Springer-Verlag, Inc.

[Garner 1990] Garner, R. B. 1990. SPARC: Scalable processor architecture. En *The Sun technology papers*, ed. M. Hall y J. Barry, 75–100. New York: Springer-Verlag, Inc.

[Goldberg 1991] Goldberg, D. 1991. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys* 23(1):5–48.

[Herrod 1998] Herrod, S. A. 1998. Using complete machine simulation to understand computer system behavior. Ph.D. thesis, Department of Computer Science, Stanford University, Stanford. California.

[Ibarra y Vergara 1995] Ibarra, E., y G. Vergara. 1995. Evaluación del sistema operativo NeXTSTEP en el Grupo Financiero InverMéxico. *Soluciones Avanzadas* 3(18):5–11.

[Magnusson et al. 1998] Magnusson, P. S., F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, y B. Werner. 1998. SimICS/sun4m: a virtual workstation. En *USENIX Annual Technical Conference*.

[NeXT 1992] NeXT Computer, Inc. 1992. *NeXTSTEP operating system software*. Massachusetts: Addison-Wesley.

[NeXT 1993] NeXT Computer, Inc. 1993. *The Objective C language*. On-line documentation.

[Patterson 1985] Patterson, D. A. 1985. Reduced instruction set computers. *Communications of the ACM* 28(1):9–15.

[Patterson y Hennessy 1993] Patterson, D. A., y J. L. Hennessy. 1993. *Arquitectura de computadores. Un enfoque cuantitativo*. Madrid: McGraw-Hill/Interamericana de España, S.A.

[Redman y Silbar 1995] Redman, J. M., y R. R. Silbar. 1995. Selecting an operating system. Part II: NeXTSTEP. *Computers in physics* 9(3):261–266.

[Silberschatz et al. 1994] Silberschatz, A., J. L. Peterson, y P. B. Galvin. 1994. *Sistemas operativos. Conceptos fundamentales*. Tercera edición. Delaware: Addison-Wesley Iberoamericana, S.A.

[SPARC 1992] SPARC International, Inc. 1992. *The SPARC architecture manual. Version 8*. New Jersey: Prentice-Hall, Inc.

[Stallings 1996] Stallings, W. 1996. *Computer organization and architecture: designing for performance*. Fourth edition. New Jersey: Prentice-Hall, Inc.

[Sun 1998] Sun Microsystems, Inc. 1998. The UltraSPARC-IIi processor. Technology White Paper.

[Tabak 1990] Tabak, D. 1990. *RISC systems*. New York: John Wiley & Sons, Inc.

[Taylor 1999] Taylor, C. 1999. Video games get trashed. *TIME*. Latin american edition 153(10):40–41.

[Tenenbaum et al. 1993] Tenenbaum, A. M., Y. Langsam, y M. J. Augenstein. 1993. *Estructuras de datos en C*. México: Prentice-Hall Hispanoamericana, S.A.

[Wilkes 1990] Wilkes, M. V. 1990. It's all software now. *Communications of the ACM*. 33(10):19-21.

[Witchel y Rosenblum 1996] Witchel, E., y M. Rosenblum. 1996. Embra: fast and flexible machine simulation. En *Proceedings of the 1996 ACM SIGMETRICS Conference*.

[Xerox 1981] The Xerox Learning Research Group. 1981. The Smalltalk-80 system. *BYTE* agosto: 36–48.