Simulació d'arquitectures VLIW

Memòria del Projecte Fi de Carrera
d'Enginyeria en Informàtica
realitzat per
Alexandre Clerc Mas
i dirigit per
Juan Carlos Moure Lòpez
Bellaterra, 21 de septembre de 2007

Agradecimientos

Primero de todo y antes de continuar con la memoria agradezco a J.C. Moure que haya confiado en mi para realizar el proyecto y en el desarrollo del proyecto.
También a Paolo Faraboschi, Investigador Científico de los laboratorios HP Labs. Me ayuda con sus respuestas en el foro sobre el compilador VEX.
No puedo olvidarme del constante apoyo moral de mis padres, por haber estado animándome estos 7 meses que ha durado el proyecto.

Índice del contenido

1. Introducción

Los procesadores actuales son muy complejos. Por ejemplo, son capaces de ejecutar varias operaciones a la vez y los procesadores Very *Long Instruction Word* (VLIW) lo consiguen.
VLIW es una filosofía de diseño arquitectónico que busca el paralelismo a nivel de instrucción en la arquitectura del procesador. Se aplica a varios niveles del sistema, esto incluye la arquitectura del hardware, el repertorio de instrucciones y el compilador.
En la arquitectura de los procesadores VLIW, la mayor parte del trabajo en buscar el paralelismo de las tareas lo realiza el compilador, así se consigue simplificar el diseño hardware. Pero poder establecer un hardware común donde poder ejecutar un programa cualquiera y obtener un alto rendimiento de todos los componentes hardware, no es tarea sencilla. Pues usar la arquitectura VLIW implica:

- Por un lado el hardware se ha de adecuar al código que ejecutará.
- Por el otro lado los compiladores han de generar un código que sea capaz de maximizar el uso de recursos.

Todos estos conceptos acerca VLIW se imparten en el temario de la asignatura Arquitectura de Computadores 2. Al alumno le es difícil entender el funcionamiento de los procesadores VLIW y, cuando aplica los conocimientos adquiridos en la teoría, le resulta difícil organizar manualmente las operaciones de algoritmos sencillos. Si quiere probar con algoritmos complejos o comparar los resultados de un programa con diferentes enunciados, el alumno puede llegar a tardar bastante tiempo.

La solución sería disponer de una aplicación de uso educativo que reduzca el tiempo de cálculo de ejercicios VLIW, como por ejemplo VEX.

VEX ("**V**LIW **Ex**ample") es un programa gratuito que funciona bajo el sistema operativo (S.O.) Linux. Éste programa dispone de un conjunto de herramientas que permiten manejar un entorno de simulación de la arquitectura de procesadores VLIW. Con VEX se puede analizar, desarrollar y depurar cualquier algoritmo escrito en código C en la arquitectura de un procesador VLIW.

Las ventajas que ofrece VEX al alumno de la asignatura son:

- Entender cómo se organizan las instrucciones para aprovechar al máximo los recursos hardware que dispone el procesador. Analizar porqué el compilador ha elegido ese orden de instrucciones. La causa la encuentra mirando las dependencias entre las operaciones que conforman cada instrucción, los retardos de las unidades funcionales, etcétera.
- Permite analizar el rendimiento del procesador VLIW con un programa escrito en alto nivel.
- Ayuda a entender el comportamiento de la arquitectura superescalar, que también soporta la ejecución de operaciones en paralelo. Los resultados con VEX son similares a la ejecución del mismo código en un ordenador superescalar.

Pero VEX presenta una serie de inconvenientes para poderse considerarse una buena herramienta docente. A continuación se exponen algunos problemas:

- Se usa desde un terminal de Linux, así que necesita comandos textuales. Los comandos de este tipo suelen ser muy específicos por cada programa. Son difíciles de recordar, poco intuitivos y hay una alta probabilidad de escribir mal un comando.
- Son bastantes pasos los que hay que hacer para realizar un ejemplo sencillo. Se ha de escribir el código en un documento, compilar y ejecutar el código. Cuando se ejecuta el código se realiza la simulación.
- Hay mucha documentación en el apéndice A del libro [1], pero no hay ningún ejemplo completo y sencillo hasta el final del apéndice.
- El alumno debe escribir varias veces los mismos comandos para familiarizarse con VEX.
- Según que argumentos opcionales se quieran usar, VEX puede llegar necesitar hasta 3 archivos diferentes.
- Usa ficheros para almacenar los resultados de la simulación. En uno de los archivos se almacenan las instrucciones en formato VLIW y permite ver como las ha organizado el compilador. Este fichero es difícil de interpretar ya que se necesitan bastantes líneas de texto para mostrar unas pocas instrucciones VLIW.

Ello dificulta, por ejemplo, el estudio local de un bucle o acceder a una parte del código porque de repente el alumno se encuentra con una operación de salto.
- Se deben usar aplicaciones externas, como los editores de texto para escribir en los ficheros o cargarlos.

Resumiendo, la productividad del simulador VEX es baja.

El objetivo principal de este proyecto es diseñar e implementar en Java una interfaz gráfica para manejar VEX de forma más intuitiva, rápida y eficiente. Java es un lenguaje de alto nivel que dispone de las funciones necesarias para poder implementar una interfaz gráfica, de ahí que la aplicación del proyecto se llame JavaVEX.

JavaVEX le debe servir al alumno para analizar, desarrollar y depurar los programas en C sobre la arquitectura VLIW, pero VEX es quien realiza todo el trabajo de computación. La interfaz solo gestiona los ficheros necesarios para que VEX funcione y lee los datos de los archivos que el simulador genera. La interfaz gráfica muestra los resultados más importantes y con un diseño atractivo.

Que aporta JavaVEX a VEX?

- Evita el uso del terminal y, por consiguiente, evita el uso de comandos textuales porque ya están implícitos en el código del programa. El usuario desconoce las llamadas que lanza la interfaz al simulador VEX, no tiene ni que preocuparse de si existen y como son.
- Se ha pasado de tener comandos textuales a objetos visuales. El alumno solo debe preocuparse de descubrir o intuir que hace cada objeto de JavaVEX para luego recordarlo. Es difícil que lo olvide.
- La interfaz gráfica esta formada por elementos comunes a otras interfaces gráficas.
- Se puede simular cualquier ejemplo con un solo *click* del ratón. El usuario ve los resultados de la compilación y la simulación a la vez. Aunque en el código están implícitas su llamadas y se tratan por separado.

- Hay un manual de usuario con la información justa y necesaria para poder realizar las primeras pruebas con JavaVEX. Desde un principio se usa un ejemplo de programa en C sencillo que se va optimizando.
- Muchos datos de entrada son accesibles y configurables con un solo *click* de ratón.
    - Evita tener que usar un editor de texto.
    - Unifica el acceso a múltiples ficheros de entrada.
    - Reduce el tiempo de modificación del hardware de la arquitectura.
- Selecciona los datos más importantes de los ficheros generados por VEX. Son aquellos datos que permiten sacar conclusiones y poder comparar rápidamente entre distintos ejemplos. Evita que el usuario tenga que acceder al fichero y buscar en el texto la información. Con esto se consiguen que la información más compleja y menos susceptible a ser cambiada esté oculta al alumno.
- Es más fácil realizar estudios locales. La información de las instrucciones VLIW esta en un formato similar al usado en la asignatura AC2.
- Se ayuda de un sistema visual de líneas que permite saber a que parte del código apunta una operación.
- Tiene información visual que complementa los resultados de la simulación.

Que me permite hacer JavaVEX?

- Editar, guardar y cargar el programa C o los ficheros que definen la arquitectura hardware del procesador. En los archivos de configuración del hardware se pueden cambiar diversos aspectos del procesador como: el número de unidades funcionales y su latencia, la organización de la *cache*, etcétera…
- Ver los resultados de la compilación enteramente en formato VLIW.
- Ver los resultados de la simulación. Éstos se componen de valores numéricos (como el numero de ciclos ejecutados, la cantidad de fallos en la *cache* de datos, etcétera…) acompañados de su tanto por ciento respecto otro dato (numero total de ciclos, numero de accesos a *cache*). El porcentaje también se representa con barras. Éstas permiten visualizar fácilmente cuanta ha sido la mejora de una configuración anterior a la actual.

Como VEX y JavaVEX solo funcionan bajo Linux. Para poder usar la aplicación en cualquier ordenador, aunque no tenga Linux, se ha realizado un *LiveCD*. Un *LiveCD* permite cargar el S.O. en la memoria del ordenador y el resto de S.O instalados no se ven afectados. En el proyecto se ha incluido la distribución de Linux Fedora Core 6 (FC6) de 32 bits.

1.1 Objetivos del proyecto:

- Conocer el funcionamiento de la arquitectura VLIW, para ello se dispone de la herramienta VEX. Por la importancia del compilador VEX en la interfaz se debe aprender a manejar, estudiar su comportamiento con ejemplos y escoger de todas las opciones que ofrece VEX, cuales son las más importantes de forma que permitan reflejar cambios en el comportamiento del procesador VLIW según el código a simular, el modelo de ordenador y el grado de optimización.

- Crear una herramienta autosuficiente que permita al usuario aprender y experimentar sobre la arquitectura VLIW, sin tener que usar otro programa. El usuario no debería usar el compilador VEX a no ser que quiera probar con opciones de compilación no incluidas en la interfaz.

- Diseño de una interfaz gráfica, debe un diseño simple pero completo. Con pocas acciones la interfaz debe sacar provecho del potente compilador VEX.

- Implementar una interfaz gráfica con Java. Así se mejora la experiencia de programar con un lenguaje de alto nivel orientado a objetos. Comporta elegir una de las muchas formas que puede implementarse una interfaz gráfica en Java , de que objetos se compone, y finalmente se describe el comportamiento de cada objeto y como interacciona JavaVEX con VEX.

- Creación de un *LiveCD. A*aprender como se hacen y que se pueda ejecutar en cualquier ordenador, debe incorporar el compilador VEX y del JavaVEX.

- Realizar un manual de usuario (ver apéndice A). Explica como funciona la interfaz gráfica y que tareas permite hacer. Ayudarse de ejemplos e incluirlos en

el manual permiten al usuario entender que y como hacer una tarea con la interfaz gráfica. Luego se explicarán que cambios permite realizar JavaVEX según los resultados.

1.2 Requerimientos genéricos de la interfaz gráfica:

- De fácil uso para cualquier usuario, ya sea de la carrera de Informática o cualquiera que tenga interés en la materia. Una interfaz es fácil de usar si:

    - Se puede intuir que realizara cada uno de los componentes de que esta compuesto: un usuario sin conocimiento alguno sobre su uso debe dominar la interfaz gráfica en poco tiempo. Para hacer una interfaz gráfica se suelen usar métodos usados en otras interfaces gráficas ya existentes y más populares. En caso de duda, para consultar ayuda se ha hecho un manual de usuario que explica las tareas que permite hacer la aplicación (ver apéndice A).

    - Tiene un diseño atractivo: con la incorporación de elementos gráficos que facilitan la comprensión e interpretación de los resultados, o de sistemas de ayuda (visuales o textuales) que aportan más información sobre la disponible

    - El uso de cada objeto de la interfaz realiza la tarea que se espera que haga, por ejemplo, un menú suele desplegarse y no actuará como un botón.

- Versatilidad:

    - Todo ordenador debe ser capaz de poder ejecutar la aplicación.

    - La interfaz debe ofrecer la posibilidad de modificar algunos parámetros del VEX sin recurrir al terminal.

    - Debe incorporar un sistema de control de errores que se muestran en caso que suceda un error.

- Coherencia de la información: lo que se muestra en pantalla, es siempre el producto de la última compilación y simulación hecha sobre el código C.

- Agrupación de la información: el número de documentos a acceder no debe ser muy grande para poder probar diferentes conceptos. VEX funciona usando varios ficheros, así que en la interfaz gráfica o la implementación debe evitar usar más archivos.

- Interacción:

    - La interacción debe ser sencilla, aceptando la mayoría de acciones que el usuario desee hacer.
    - La velocidad de respuesta debe ser rápida, no se debe añadir mucho tiempo extra de latencia respecto al generado por el compilador VEX.
    - La fiabilidad de una interfaz gráfica se traduce en que su ejecución es fiable y que el numero de errores que tiene es mínimo o nulo.

- Debe ser una aplicación de soporte para el simulador VEX. Aunque no se llegará a usar todo el potencial de VEX, se debe adaptar a las necesidades básicas o de acuerdo al grado de aprendizaje del usuario.

1.3 Motivación personal del proyecto:

Este proyecto se ha realizado por iniciativa del director del proyecto. El objetivo final es usarlo en la parte final del temario que se imparte en AC2, después de realizar ejercicios sobre los procesadores segmentados y sobre arquitecturas de procesadores con varias unidades funcionales. Esta aplicación es el punto y final que resumiría el temario visto en el curso y probar hasta aspectos de la *cache* que se estudiaron en la asignatura Arquitectura de Computadores 1.

Como alumno se me ofrece la posibilidad de poder realizar una aplicación de la que otros se beneficiarán de su uso y a mí para profundizar en AC2. Ésta asignatura no disponía de ninguna aplicación sobre este tema. También permite poder trabajar y adquirir experiencia en:

- El lenguaje de alto nivel Java, que en algunas asignaturas se usa para realizar sus prácticas, pero sin llegar a aprovechar su alto potencial y de la alta variedad de elementos que dispone para adaptarse lo mejor posible a las exigencias de software actuales.
- Sistemas basados en Linux. Si bien es cierto que en bastantes asignaturas se usa, al alumno nunca se le enseña ha instalarlo y configurarlo, o bien se hace mediante una asignatura de libre elección o por voluntad propia. Dado su uso imprescindible en el proyecto fue un aliciente a escogerlo.

1.4 Organización de la memoria:

En el siguiente capítulo se describe el estado del arte con dos programas relacionados con la interfaz del proyecto y se describen algunas características relevantes.
El capítulo 3 es la base teórica sobre la que se fomenta este proyecto. Es un apartado extenso ya que la creación del *LiveCD*, funcionamiento de VEX, estudio de las arquitecturas VLIW y la programación en Java han requerido bastante investigación.
En el capítulo 4 es donde se describe la funcionalidad de la interfaz gráfica y se describen los elementos que la componen.
En el capitulo 5 se explica el trabajo realizado. Se explica como interactua JavaVEX con VEX. Sobre la implementación en Java, se mencionan las funciones más usadas y en pseudo-código se escriben las dos funciones más importantes del código, para finalizar un diagrama de clases ayuda a tener una visualización global de la implementación. Para acabar el capítulo se explica el proceso de creación del *LiveCD* con detalle.
En el capítulo 6 se explica el desarrollo del proyecto, los problemas que han ido apareciendo y las decisiones tomadas.
Las conclusiones, que objetivos se han cumplido, que he aprendido y la opinión del proyecto están en el capítulo 7. El capítulo 8 es la bibliografía y por acabar dos apéndices con el manual de usuario y los ficheros usados en la creación del *LiveCD.*

2 .Estado del arte

Es prácticamente imposible encontrar un proyecto que se pueda comparar a la dimensión del proyecto descrito en esta memoria.
Existen diversas aplicaciones relacionadas con la simulación del comportamiento de los procesadores escalares (solo pueden ejecutar una operación a la vez), superescalares. También hay aplicaciones que permiten ver las microinstrucciones de un programa.

Este apartado se centra en estudiar las interfaces gráficas de los simuladores KScalar y Misc-1. El primero permite ser configurado para simular procesadores escalares o superescalares, mientras que el segundo permite ver las microinstrucciones de la arquitectura escalar.

2.1 KScalar [8]

Es un simulador que permite observar el comportamiento de distintos procesadores como los escalares o superescalares y con distintos tipos de ejecución (en orden o fuera de orden). Una ejecución en orden es que todas las operaciones empiezan y finalizan a la vez, mientras que en fuera de orden no. KScalar permite la ejecución entera, parcial o ciclo a ciclo del código, como mejor le vaya al usuario.

Lo más interesante en el diseño de esta interfaz es:

1) En una ventana se puede ver ciclo a ciclo, los eventos de la microarquitectura que genera al ejecutarse una instrucción en el hardware del procesador. Los eventos se muestran con la ayuda de un dibujo de las unidades del procesador que requiere la instrucción seleccionada o que se este ejecutando en ese momento. En la misma ventana se muestran también los datos/direcciones que contiene en cada estado *pipeline* la instrucción.

- Esta ventana ofrece mucha información útil de cada instrucción. Al usuario le facilita el estudio del comportamiento de la arquitectura simulada.

2) En la figura 2.1 corresponde a una vista de la otra ventana, donde esta el diagrama de ciclos. Muestra la posición dentro del *pipeline* de un grupo de instrucciones a lo largo de varios ciclos. Es información comprimida de la primera ventana. Permite ver fácilmente si una instrucción se llega a ejecutar sin espera alguna, o si, debido a dependencias o falta de recursos se ha tenido que esperar en un estado del *pipeline*.
Se ayuda de distintos colores para cada estado y de un color distinto por cada bucle del algoritmo simulado para diferenciar mejor cada bucle.

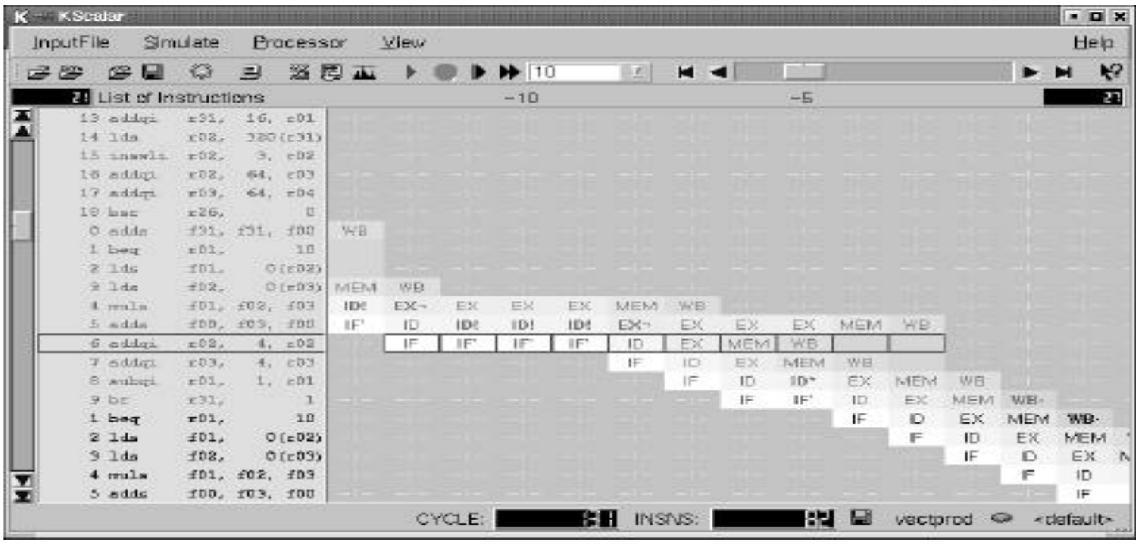


Figura 2.1 Diagrama de ciclos.

De esta ventana destaca:

- Permite ver fácilmente donde hay esperas. Bien porque un estado se repite dos o más veces en la misma línea (pertenece a una instrucción), o porque hay espacios vacíos debajo de una etapa, que si se diera el caso ideal, es donde tendría que empezar a ejecutarse una nueva instrucción.
- Dispone de información de ayuda, ofrece la posibilidad de seleccionar una instrucción y se muestra información sobre el tipo de operación y su latencia, si se quiere más se puede pulsar el botón *help*.
- Dispone de dos barras que permiten desplazarse a través del diagrama de ciclos. Estas barras suelen usarse para realizar un estudio local de los bucles anteriores y posteriores cercanos al que se estudie. Observar el comportamiento de unos pocos bucles suele bastar para saber que sucede en la mayoría de todos los bucles del programa simulado, a excepción del primero y ultimo.

Del entorno de las dos ventanas destaca (ver figura 2.1):

- Encima del diagrama de ciclos hay una barra. Contiene las herramientas más usadas, son fácilmente accesibles. Entre ellas, una barra que permite acceder a un ciclo cualquiera.
- Debajo del diagrama de ciclos se puede visualizar el número de ciclos y el número de instrucciones del código simulado.

2.1.1 Configuraciones y resultados de la simulación

Ofrece la posibilidad de configurar la microarquitectura. Por ejemplo la latencia de cada estado, la configuración del tipo de ejecución (en orden o fuera de orden). También se puede configurar el comportamiento de las *caches* L1 (de instrucción y datos) y L2 unificada.

- Las configuraciones están bien organizadas en diferentes pestañas y son fáciles de reconocer para configurar aquello que importa. Se puede salvar y cargar cualquier configuración.

KScalar también genera estadísticas al finalizar la simulación, aunque puede descartarse permitiendo que la simulación vaya más rápida.

- Las estadísticas se muestran en una tabla, si se quiere más información sobre el significado de ese dato se deja el ratón un breve tiempo encima de este.

2.2 Misc-1 [9]

Esta aplicación no permite la simulación de procesadores que soporten instrucciones paralelas. Es un simulador de las microinstrucciones en los procesadores escalares. Se ha elegido porque esta implementado en Java.
Misc-1 está destinado como herramienta docente para los alumnos que empiezan en el estudio de microinstrucciones que generan las operaciones ensamblador. El java tiene su propio lenguaje ensamblador (*JVM Java Virtual Machine*), pero el programador de la aplicación la modifico creando el IJVM.

A diferencia del KScalar o el JavaVEX, que tienen todas las herramientas o fases integradas en una sola interfaz gráfica, en este programa se debe generar previamente en 2 interfaces gráficas distintas:

- Un ejecutable que contiene el código en el conjunto de instrucciones de la arquitectura, lo que suele decirse en inglés *Instruction Set Architecture* (ISA).
- Un archivo binario adaptado a la microarquitectura del Misc-1.

Una vez creados los dos ficheros se cargan en una tercera interfaz que permite ejecutar todo el programa, o paso a paso se pueden ver todas las microinstrucciónes que ha generado.
No se puede volver atrás en la simulación. Hace falta cargar la ventana debug donde se escribe la información de todos los ciclos ejecutados hasta ahora.
La siguiente figura 2.2 es una captura de pantalla del Misc-1:

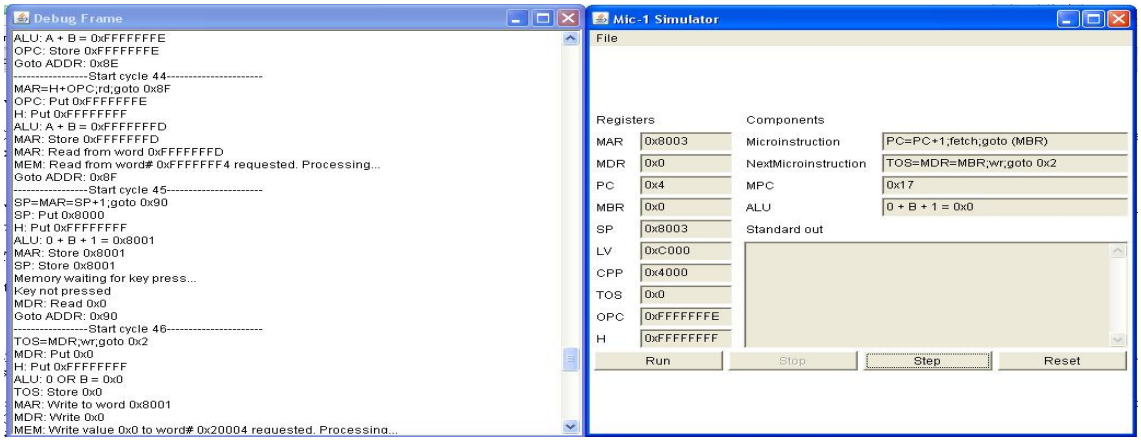


Figura 2.1 Ventana debug y ventana del simulador Misc-1.

Aunque esta aplicación no entra en la finalidad docente del JavaVEX, solo por su diseño se podría haber implementado todo bajo una misma ventana. Por ejemplo, una zona de compilación de ficheros y otra donde el debug y la simulación integradas en una.

3. Conceptos teóricos y descripción de las herramientas usadas

En este apartado se pretende explicar la base teórica en la que se basan las diversas herramientas usadas en el proyecto, así como el funcionamiento de éstas. Pretende ser un capítulo de introducción al cuerpo de la memoria, así se podrá entender mejor el desarrollo del proyecto.

3.1 La arquitectura VLIW

3.1.1 Definición de VLIW

Como se ha explicado en la introducción del proyecto. VLIW es una filosofía de diseño arquitectónico que busca el paralelismo a nivel de instrucción en la arquitectura del procesador. Se aplica a varios niveles del sistema, esto incluye la arquitectura del hardware, su repertorio de instrucciones y el compilador, todo con el propósito de encontrar tareas que se puedan ejecutar a la vez en vez de hacerlo secuencialmente.

Los procesadores VLIW tienen pocas instrucciones, pero como su nombre indica, son de un tamaño grande. Si hay tanta información de cada instrucción es porque se decide que la mayor parte del trabajo en buscar el paralelismo de las tareas lo haga el compilador, así se consigue simplificar el diseño hardware.

3.1.2 Los compiladores VLIW

El objetivo de los compiladores VLIW es coger grupos de operaciones sin dependencias, independientes y convertirlas en tipo *Very Long Instruction Word* de forma que las unidades funcionales procuren ser usadas lo más eficientemente posible en cada ciclo. Es una medida posible a no ser que se quiera incrementar la frecuencia de reloj del procesador.

Para poder agrupar las operaciones en instrucciones VLIW el compilador ha de descubrir todas las dependencias, determinar como ha de resolverlas y seguramente la solución será la reordenación del programa moviendo bloques de código según convenga.

El compilador, para la ordenación de las instrucciones, debe ir creando imágenes actuales del estado de la arquitectura, ya sea el de los registros, los accesos que se harán

a memoria, el bus del sistema, además de encontrar y optimizar la organización de las unidades funcionales teniendo en cuenta la precedencia de los datos que use. Estas imágenes son útiles porque agilizan y mucho la simulación en los programas que simulan la arquitectura VLIW.

3.1.3 VLIW vs Superescalar

A pesar de que la mayoría de procesadores actuales son superescalares, el estudio de una arquitectura similar como es la VLIW no deja de ser interesante para ver que aporta ésta sobre los superescalares. Ambas intentan explotar cada vez más el nivel de paralelismo de las instrucciones.
La reordenación en un procesador superescalar difiere del de un compilador VLIW, el primero se basa en su hardware para realizar las dependencias cuando se esta ejecutando el código. Concretando más, ambos trabajan y tratan de forma diferente la reordenación de instrucciones en los bloques básicos. Un bloque básico es una secuencia de instrucciones en las que el flujo de control entra al principio y sale al final donde. Los dos extremos del bloque suelen ser etiquetados para saltar a ellos desde cualquier parte del código.

En un procesador superescalar se determinan las dependencias, las operaciones a poner en paralelo, dentro de un bloque básico. Menospreciando el provecho que se puede sacar de mirar un poco más allá. En cambio un compilador VLIW, mira las operaciones de diferentes bloques básicos e intenta ponerlas dentro de una misma instrucción.

El objetivo primordial de VLIW es incrementar el número de operaciones ejecutables, y por tanto más instrucciones, sin saltos o esperas, haciendo bloques de instrucciones más grandes.

Otra diferencia es que VLIW reordena el programa independientemente de cómo este implementado, en cambio cuando las instrucciones son ejecutadas por un procesador superescalar, vendrán determinadas según se encuentran en el programa, es decir dependen del compilador y/o el programador.

Aunque el rendimiento real de ambas arquitecturas de procesadores esta por debajo de su máximo teórico, las razones que inducen a ello son muchas: fallos en la predicción de

saltos, fallos en la memoria supuestamente más rápida y consiguiente desaprovechamiento de ésta teniendo que acceder a otra más lenta, y el bloqueo de la unidad central de procesamiento (CPU) por dependencias de datos.

3.1.4 *Trace scheduling*

Es una de las técnicas usadas para la reordenación en VLIW sobre operaciones que están en diferentes bloques básicos.

"Un trace, o rastro, es un posible camino a través de un programa, el camino de ejecución por el cual ira dado un dato de entrada." ***Dick Pountain (artículo de byte.com)*** **[4]**

La técnica se separa en dos procesos, el primero consiste en rastrear el código para seleccionar operaciones (*trace selection*). Mira de encontrar esos bloques básicos cuyas operaciones se pueden poner dentro de un pequeño número de instrucciones. Se consigue usando algoritmos heurísticos que calculan y pronostican las direcciones que posiblemente cogerá el código una vez sea ejecutado, el compilador seleccionará el camino más probable.

Después de esta selección, actúa el segundo proceso llamado rastro de compactación (trace *compactation*). Como su nombre indica intenta compactar las instrucciones en el menor número de instrucciones que le permita la arquitectura. Intenta avanzar todas las operaciones lo antes posible, actúa como si fuera el primer proceso pero ahora lo hace con todo el código, como si de un gran bloque básico se tratase.

Hay más técnicas a parte, la mayoría de compiladores disponen del *loop unrolling* y de la conversión del *IF*, el ultimo se basa en reemplazar las dependencias de control en dependencias de datos, algunas dependencias de control ocasionadas por saltos pueden ser remplazadas por predicado seguido de un booleano, todo ello sin perjudicar el funcionamiento del programa y liberando recursos al compilador que son esenciales para poder buscar la mejor organización de las operaciones.

3.1.5 Ejemplo de un entorno de simulación de procesadores VLIW

Uno de los pocos estudios hechos para poder crear un entorno de simulación en un procesador VLIW se puede encontrar un articulo cuyo titulo es: “Simulation/Evaluation Environment for a VLIW Processor Architecture” presentado en la Jornada IBM de Desarrollo e Investigación (1997) por J. H. Moreno [5]. En su presentación se basaron en este dibujo para mostrar una visión general de su sistema y explicar su funcionamiento:

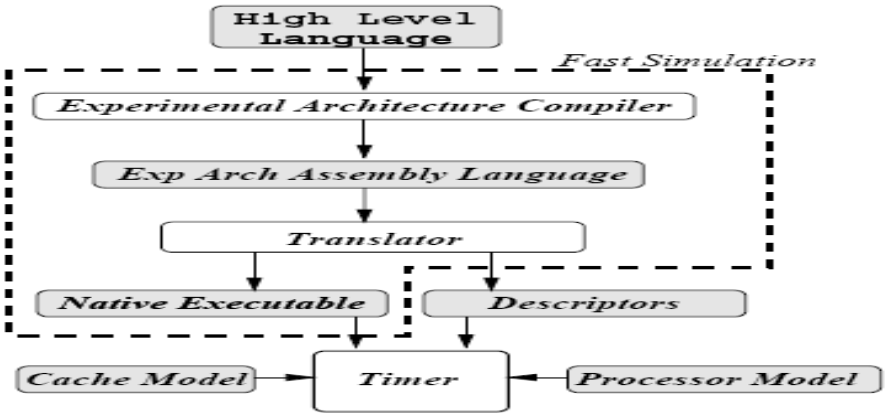


Figura 3.1 Vista general del sistema de simulación VLIW.

Se puede observar como un programa escrito en lenguaje de alto nivel es compilado y se genera su lenguaje ensamblador, que es traducido al lenguaje nativo de la máquina donde se ejecutara y se generan a su vez los descriptores que identifican cada instrucción. Finalmente una máquina será la seleccionada como modelo de procesador a ser simulada y el modelo de memoria para definir el comportamiento de la máquina.

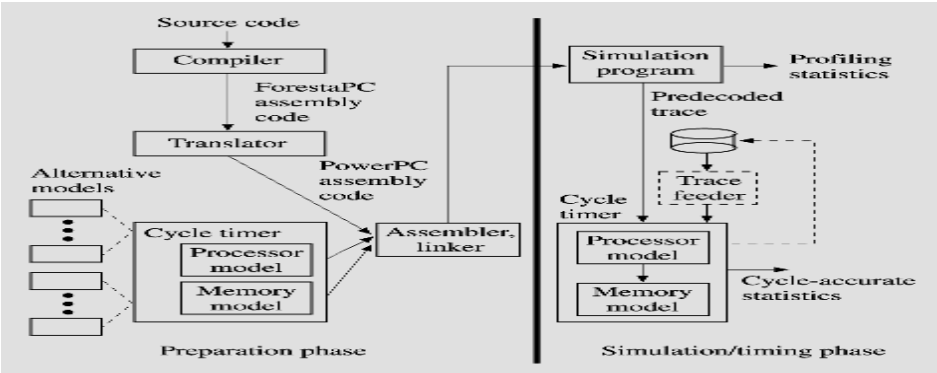


Figura 3.2 Fases de preparación y simulación.

Un modelo de procesador representa a un procesador real donde el hardware y el comportamiento están definidos.

A continuación se comenta la figura 3.2 que se asimila más al comportamiento real del simulador desarrollado por IBM. Sus dos principales fases son:

1) La fase de preparación: es donde esta el compilador VLIW. Tiene como entrada el código y genera un código ensamblador VLIW (del repertorio de instrucciones Foresta PC). Luego un traductor se encargará de traducirlo a código ensamblador de un procesador (PowerPC) que será el que emulara el comportamiento del programa VLIW.

El traductor genera unos ficheros que contienen información relacionada con el comportamiento del programa VLIW, además de información sobre las posibles rutas que se ejecutan en la siguiente fase, la de simulación.

Por ahora ya se dispone de suficiente información para experimentar con diferentes arquitecturas y algoritmos de compilación para tener una primera idea de cómo funcionan.

2) La fase de simulación: donde el programa es simulado. Genera información en tiempo de ejecución gracias a las llamadas a un contador de ciclos de donde se obtienen datos que serán la base para un estudio posterior. En esta fase la simulación variara dependiendo del:

- Modelo del procesador: contiene el contador de ciclos y datos como las dependencias entre registros y las latencias de las operaciones para la implementación de un procesador.
- Modelo de memoria: las operaciones de memoria se definen en este modelo.

Concretando el funcionamiento del contador de ciclos. Se hace una llamada por cada instrucción VLIW justo antes de llevarse a término su ejecución. Se pasa como argumentos un descriptor que identifica a la instrucción y una imagen de los registros del procesador con sus contenidos. En acabarse la ejecución, se generan los datos para esa instrucción y se vuelve a empezar todo para la siguiente.

La alta eficiencia de la simulación se consigue gracias a todos los datos realizados en la fase anterior, la de preparación, por cada instrucción VLIW, solo se requiere una lectura de la imagen para analizarla rápidamente como se comportará esa instrucción en la arquitectura.

A continuación se pasa a explicar el funcionamiento del entorno de simulación usado en el proyecto

3.1.6 El sistema VEX [1]

Es un modelo de arquitectura escalable para procesadores VLIW. Ofrece la posibilidad de cambiar el número de operaciones por instrucción, unidades funcionales, registros, permite añadir nuevas instrucciones sobre el conjunto disponible.

La arquitectura del sistema VEX esta basada en operaciones *load/store* y la ejecución en orden (todas las operaciones de cada instrucción empiezan a la vez y dejan sus resultados a la vez). Si el hardware puede completar la operación en el mismo número de ciclos que el asumido por el compilador, no comporta ningún bloqueo al procesador. En cambio, si la operación requiere más latencia que la inicialmente asumida, el hardware debe mantener en pausa la ejecución hasta que se pueda volver a reiniciar otra vez. Las esperas suele venir a causa de fallos de *cache* y de saltos, es información útil y por eso se muestra información sobre los fallos de *cache* y saltos en JavaVEX.

3.1.6.1 Arquitectura compuesta de *clusters*

Un *cluster* es un conjunto de registros y un conjunto de unidades funcionales acoplados. Permite el acceso múltiple a memoria.

Dentro un *cluster* están completamente conectados entre si los registros y éstos con las unidades funcionales, pero las unidades entre si no están conectadas; permite que el resultado de una unidad funcional se pueda dejar en el mismo *cluster* sin necesidad de dejarlo en un medio de almacenaje exterior.

3.1.6.2 Estructura de un *cluster* por defecto

Aunque JavaVEX solo usa un cluster de VEX a continuación se comentan las características más importantes de éste.

Contiene unidades funcionales de memoria, entero, y salto

- Las unidades de memoria: permiten la carga, almacenamiento y operaciones de prebúsqueda.

- Las unidades de entero: ejecutan el típico conjunto de operaciones de entero, comparación, desplazamiento y selección sobre registros o sobre operandos inmediatos.

- Las unidades de salto: son operaciones que pueden usar el resultado de una condición almacenada en registros especiales que solo pueden usar las operaciones de salto condicional, las incondicionales, o las llamadas directas/indirectas y los *returns*.

  - Las operaciones de salto se ayudan de 8 registros de 1 *bit* para las unidades de salto. Este *bit* sirve para la comparación útil en las operaciones lógicas sobre condiciones, o en operaciones de selección. También sirven para almacenar los bits de *carry*.

Para el resto de operaciones dispone de 64 registros de 32 bits de propósito general.

Las unidades funcionales por defecto de un cluster están principalmente compuestas de:

- Cuatro *Arithmetic Logic Unit* (ALU) para enteros con latencia de 1 ciclo.
- Dos multiplicadores de 2 latencias de ciclo.
- Una unidad de memoria tarda 3 ciclos para la operación de *load* y otra unidad de 1 ciclo para las operaciones *store*.

Todas las operaciones realizan sus cálculos siempre sobre 32bits y no incluyen soporte para valores que no sean de 32 bits. Hasta un máximo de 4 operaciones pueden realizarse por instrucción en cada cluster.

3.1.6.3 Aritmética y operaciones lógicas

VEX soporta el mismo formato de operaciones aritméticas que las soportadas por los mircroprocesadores *Reduced Instruction Set Computer* (RISC), además de las menos tradicionales desplazamiento y suma en una operación, selección, las operaciones

lógicas de AND y OR, mínimo, máximo y un conjunto diverso en operaciones de multiplicación.

El compilador VEX no incluye ningún control hardware para operaciones de punto flotante, lo hace a partir de una librería que emula su funcionamiento. La librería en cuestión se llama *SoftFloat*, permite compilar código en punto flotante pero el desempeño será el de un procesador sin hardware para punto flotante.

3.1.6.4 El simulador C de VEX

El uso de VEX como herramienta docente esta designado para la exploración de diferentes arquitecturas, desarrollo de aplicaciones y realización de estándares de comparación (b*enchmarks*).

La herramienta consta de 4 principales directorios. El directorio *bin* que es donde están las herramientas principales para la compilación y ejecución. La carpeta *lib* para las librerías. Finalmente el directorio *include*, usado para las cabeceras de las funciones hechas en C. Actualmente también esta la carpeta *share* que incluye ficheros y ejemplos, pero no es usada en ningún momento por VEX y por eso no viene incluida en el JavaVEX.

Explicar como funciona el compilador VEX una vez entendido el ejemplo anterior de IBM es fácil, partiendo de la misma figura 3.1 se puede hacer.

VEX únicamente traduce un lenguaje de alto nivel como lo es C. Con una llamada al *compiler C* (cc) que viene con VEX, se genera el lenguaje ensamblador VLIW y, actuando como el traductor anterior, realiza una llamada al compilador C nativo del sistema para obtener su versión ejecutable.

En otras palabras, cuando se ha creado el ejecutable, este contiene una imagen del programa compilado por VEX para ser usado por el simulador y convertida en el formato binario de la maquina donde se ejecutara, que se supone será en la misma. Si se simula en otra maquina no tiene porque funcionar.

- También incluye varias funciones de ayuda para la simulación y funciones de simulación de las *caches* de datos e instrucción.

Cuando el programa acaba, se genera un fichero que contiene los datos estadísticos de la ejecución y su comportamiento final partiendo de un contador de ciclos, como el sistema de simulación de IBM. Como el ejecutable tiene funciones de ayuda para la simulación de las *caches* de datos / instrucciones también se guarda en el fichero su comportamiento.

Si queremos información sobre el código ensamblador VLIW generado por el compilador de VEX. Se debe añadir una opción sobre el modo por defecto que simplemente evita la eliminación del fichero tanto si el programa finaliza como si hay errores, dejándolo en el disco duro (más concretamente en la carpeta */bin*).

3.1.6.5 Optimizaciones

VEX dispone de interesantes opciones que permiten experimentar sobre diferentes grados de optimización como el uso del *trace scheduling o el loop unrolling*, algunas rutinas se verán beneficiadas de ello. Para optimizaciones fuertes se tiene que estudiar si se saca algún beneficio o no. La optimización usada por VEX en modo por defecto es más que suficiente para un programa escrito en C.

3.1.6.6 Parámetros de configuración de las unidades funcionales y registros.

Como en IBM también hay dos modelos diferentes:

1) Modelo de recursos hardware. Las latencias de las operaciones ya son conocidas por el compilador, mientras que el anterior ejemplo de IBM solo se encargaba de generar imágenes sobre las unidades que serán usadas.

Si se quiere modificar los recursos de la máquina (como los registros y unidades funcionales) del procesador VLIW al compilar el código C. Se usa un fichero con extensión .mm donde se ponen los valores en forma de tupla: < Opción, propiedad, valor >.

Sin entrar en mucho detalle, lo que principalmente deja configurar el fichero se puede resumir mirando las posibles opciones:

- RES: sirve para establecer el numero de recursos de la maquina, se puede usar, por ejemplo, para establecer la cantidad máxima de operaciones que dispone una instrucción VLIW.

- DEL: sirve para establecer la latencia de un elemento de la maquina, como añadir más retraso o no a una operación.

- REG: sirve para decir el número de registros, tanto a los de propósito general como de salto.

2) Modelo de memoria. La simulación dispone de otro modelo que permite configurar diversos aspectos de la *cache* (tamaño, latencia, tamaño de bloques, etcétera) y otros parámetros que configuran la arquitectura del procesador VLIW.

Hay más información de los dos ficheros disponible en el manual de usuario (ver apéndice A).

3.1.7 Claves de los entornos de simulación VEX e IBM.

- La clara diferenciación modular entre las diferentes fases.
    - Ambos simuladores sin entrar en la fase de simulación ya disponen de suficientes datos para ser estudiados.
    - Aunque en IBM la simulación esta enteramente adaptada a una arquitectura especifica porque usa los dos modelos a la vez. VEX ya dispone de cierta información como las latencias de las operaciones antes, en la compilación, y es mejor para el compilador disponer en ese momento de las dependencias.

- La flexibilidad que ofrecen ante las distintas configuraciones.

    - Ambos disponen de configuraciones en el procesador, su arquitectura y para los accesos a la memoria.

- Los requerimientos de memoria son proporcionales al tamaño estático del programa.

- La simulación pude ser usada solo en la zona crítica del código.

- En el caso de IBM: ForestaPC puede contener el lenguaje ensamblador en VLIW de la parte crítica que ha pasado previamente por el compilador. Mientras que el resto del código puede ser directamente añadido cuando se ha traducido al lenguaje ensamblador del PowerPC.
- en VEX todo el programa o su parte crítica pueden ser simulados.

3.2*LiveCD*

En los siguientes apartados se introducirán los conceptos básicos y se explican los diferentes procesos de mayor dificultad de creación de un *LiveCD* a menor.

3.2.1 Que es un *LiveCD*?

Es la inclusión de un sistema operativo en un medio extraíble, como un USB, pero usualmente se hace sobre un Disco Compacto (del inglés *Compact Disc* o CD).

Uno de los motivos para crear un *LiveCD* suele ser para incluir un sistema operativo (la mayoría de veces de Linux) configurado para una función específica. Al ejecutarse el *LiveCD* al encender el ordenador, se carga el SO en la memoria *Random Acces Memory* (RAM) y la memoria se comporta como un disco duro virtual, una vez cargado el SO, se pueden usar muchas de las opciones permitidas en su versión completa y que suele ser al final la instalada en un ordenador. Como se trata de un CD al expulsarse del ordenador no almacena cambio alguno pues no se le puede escribir nada.

Otras veces un *LiveCD* sirve para incluir una aplicación que funciona en un S.O., tal y como se hace en este proyecto. Si la cantidad de aplicaciones es grande, puede que no quepa en un CD, así que un Disco Versátil Digital (DVD) también puede usarse, pero no es lo usual.

El software, las aplicaciones que se añaden a Linux o a un *LiveCD* se puede encontrar en forma de paquete. Éste suele contener código maquina para una distribución en concreto o código fuente.

Las principales ventajas del *LiveCD* son:

- Se puede cargar en cualquier ordenador con una cantidad de RAM igual o superior a 256MB, actualmente la mayoría de ordenadores disponen de esa cantidad.

- Es ideal para usuarios sin experiencia en una distribución en concreto del sistema operativo Linux. Cada distribución es diferente, por ejemplo la instalación de paquetes en Debian difiere de los basados en Red Hat.

- En usuarios avanzados, las nuevas versiones pueden detectar quizá los ultimos dispositivos hardware que salen en el mercado. Se consigue porque se va actualizando el código fuente del S.O. (kernel) entre una vieja versión y la nueva.

3.2.2 Introducción a la creación de un *LiveCD*

Son tantas las formas de hacer uno como distribuciones hay. Hasta entre versión y versión cambia ligeramente el proceso. La mayoría parten de una imagen ISO del sistema operativo que se puede descargar de Internet. Estas imágenes permiten clonar un S.O. y usarse en diversos ordenadores.

En la creación del *LiveCD* son especialmente importantes los módulos*:*

- AUFS o su símil UFS: hacen que el sistema de archivos que contiene el *LiveCD* sea solo de lectura. Luego una vez cargado el S.O. en la RAM ya podrán ser temporalmente escritos o modificados según los privilegios del usuario, pero no repercute en el *LiveCD*.

-SquashFS: es un compresor del sistema de archivos, suele usar el algoritmo de compresión de ficheros LZMA. Gracias al compresor es posible que pueda almacenarse un S.O. completo con muchos paquetes en un solo CD aunque su uso ralentiza la carga al ordenador.

- Entorno: En esta carpeta es donde se encuentran los paquetes descomprimidos de la imagen ISO, así que cualquier modificación de sus ficheros o la incorporación de nuevos paquetes se ve reflejado al crearse el CD.

- Un repositorio: es un sistema centralizado compuesto de uno o más servidores donde los paquetes pueden ser descargados e instalados en el ordenador. Los SO permiten realizarlo manualmente con comandos que descargan el paquete pedido y, si este depende de otros paquetes (aplicaciones o herramientas) que el ordenador no tiene, también los descarga.

- Algunas aplicaciones que se han hecho para facilitar la creación de *LiveCD* no necesitan intervención alguna del usuario. Acceden directamente a Internet para descargarlos o con una interfaz gráfica sencilla [15] permite navegar y añadirlos.

- Hay herramientas que permiten crear repositorios en el mismo ordenador.

3.2.3 Procesos de Creación

Primer proceso [16] [19]

- Si la versión del código fuente que se quiere incluir en el *LiveCD* no soporta los módulos AUFS y SquashFS (aunque las versiones kernel actuales suelen soportarlas), se debe recompilar el kernel del ordenador donde se vaya a crear el *LiveCD*, y obtener, compilar e instalar los 2 módulos al sistema.

- Después se configura el código fuente que se quiere incluir en el *LiveCD.* Para esto se puede usar el fichero (.config), sirve para especificar por ejemplo que el S.O. sea de 64o 32 bits entre otros muchos parámetros, y se compila el kernel.

- De la compilación se obtendrá una imagen del sistema y el kernel compilado, que junto el fichero de configuración se copian a la carpeta de entorno, donde por ejemplo, se añaden las aplicaciones. Finalmente se crea la imagen ISO.

Estos pasos son los comunes en la mayoría de procesos de creación y distribuciones Linux, pero es algo más complejo y peligroso que la siguiente manera hecha con Ubuntu .

Segundo proceso [18]

Si se tiene instalado Ubuntu no se requiere compilar el kernel. Se supone que el *LiveCD* a crear será la misma versión que la instalada, sino se tendrá que descargar una imagen ISO o S.O. para que sean las mismas.

En este proceso se usa la imagen ISO como base del nuevo *LiveCD.* Así no hace falta configurar el kernel del S.O. que se incluirá en el CD porque parte de los mismos parámetros preestablecidos en la imagen.

Los pasos a seguir són:

- Una vez descargada la imagen. Se monta la carpeta del la imagen ISO que contiene la imagen del sistema de archivos comprimida (y que también contiene otros archivos que son necesarios para la imagen) a un directorio que será el entorno de creación del *LiveCD*. Los restantes ficheros/directorios se montan en otro directorio.

- Si se instalan o quitan paquetes en el entorno de creación o se modifica algún fichero, comporta rehacer los ficheros que acompañan a la imagen comprimida (como por ejemplo el *checksum* o suma de verificación) y comprimir otra vez el sistema de archivos usando la herramienta SquashFS. Finalmente se crea el ISO.

3.2.3.1 Como crear un *LiveCD* para Fedora [14]

Los *LiveCD* no hacen muchos años que existen, el primero en darse a conocer fue Knoppix en 2003. El proyecto Fedora ha tardado unos cuantos años más y en Diciembre del 2006 lanzo su primer *LiveCD*, durante el transcurso de este proyecto. A diferencia de los distribuidos por la comunidad de Ubuntu no permite la instalación del S.O. que incluye al disco duro y no tiene la misma popularidad.

El *LiveCD-tools* [34.a]esta desarrollado por el creador del primer *LiveCD* con Fedora. Es el mismo programa usado en el proyecto. Se compone de:

- Ficheros de configuración: es un conjunto de ficheros que permiten elegir de que paquetes consta el S.O. a incluir en el CD. Mediante *scripts* se puede

modificar el sistema operativo una vez instalados los paquetes en una memoria temporal. Esta memoria luego será comprimida creando la imagen ISO. Los ficheros con la lista de paquetes y sus *scripts* se deben comprimir en paquetes del tipo *Red Hat Package Manager* (RPM) para poder ser usados.

- Con otra herramienta se crean dos repositorios. El primero, repoA, es donde se ponen los paquetes del apartado anterior. En repoB es donde irán los paquetes que serán los usados por repoA y es donde se deben volcar todos los RPMS de los 6 CDs del Fedora Core 6.

A partir de los RPM y los repositorios creados, *LiveCD-tools* ya crea su entorno de creación al ejecutarse y genera la imagen ISO automáticamente sin intervención manual alguna. Solo con un simple comando de texto desde una terminal.

Para que el usuario no tenga que buscar información de que paquetes son los necesarios para poder garantizar la carga y uso correcto del S.O. Fedora en un cualquier ordenador, hay disponibles 3 RPMS con diferentes configuraciones: [34.b]

1) Es la básica que contiene el bash, grub y kernel entre otros módulos.

2) Se añade a 1) el entorno de escritorio GNOME y paquetes que permiten configuraciones del sistema operativo que en la configuración básica no incluian.

3) Añade varios paquetes de diversas aplicaciones.

Que se debe hacer si se desea personalizar un *LiveCD* con otras aplicaciones o ficheros?

- Si se añade o quita un paquete en una de las listas de los ficheros de configuración, comporta crear otra vez el RPM de repoA que contenía el fichero.

- Si se quiere añadir un fichero al *LiveCD*, se incluye en los paquetes de repoA y también debe crearse otra vez el RPM

- Hay herramientas que permiten crear los paquetes RPMs →usando el comando rpmbuild.

- La generación de paquetes RPM se hace partiendo de un fichero SPEC [34.c]. Los nombres de los ficheros a añadir (sin olvidar el fichero de configuración) y su ubicación dentro del RPM es la única información que requiere.

- En los ficheros SPEC es donde a los ficheros de configuración se les impone un orden jerárquico. Pues algunas aplicaciones no se podrán instalar sin antes instalar los paquetes básicos.

- Para acabar en los *scripts* faltaría poner la ruta final y real del fichero añadido dentro del sistema de archivos del *LiveCD*. Por ejemplo, en la carpeta de escritorio */Desktop*.

Como puede observarse todo el proceso se basa en ficheros con los nombres de archivos o paquetes a usar. El trabajo de creación se deja a manos del programa. Lo hace idóneo para usuarios no avanzados de Linux, además, la mayoría de *scripts* ya están hechos en los 3 RPM de ejemplo y con un poco de información por Internet se pueden implementar o modificar.

3.3 Java [10]

Es un lenguaje orientado a objetos. La idea de la programación orientada a objetos es poder diseñar un software donde cada dato del código está unido a un conjunto de operaciones que puede usar.

En Java los objetos son los elementos para representar algo de un modelo y su comportamiento dentro de un programa. Aquellos objetos que comparten cierto comportamiento se agrupan en diferentes categorías llamadas clases.

Las clases son una sección de código que especifica como se comportan o actúan los objetos de esa clase. Así, cuando se ha definido una clase se pueden crear objetos de esa clase. Un objeto solo puede ser de una clase y al crearse se le denomina una instanciación de esa clase.

3.3.1 Extensión de clases

Si se quiere ampliar la funcionalidad de una clase definida por Java, se puede extender mediante la herencia y la implementación de nuevas funciones que permiten cambiar el estado y comportamiento de los datos.

Solo extender una clase sin escribir nada más, sin añadir aun nuevas funciones, ya es la definición de una nueva clase. Ésta puede ser extendida una y otra vez bajo nuevos nombres que la identifiquen tantas veces como se quiera.

Así es como está estructurado Java, similar a la de un árbol. Es un conjunto de clases que heredan de otras clases. A medida que se van heredando se van especificando cada vez más los comportamientos de un elemento dentro del modelo a representar.

3.3.2 Interfaces de Java

Hay ocasiones en que se quiere imponer el uso de ciertos objetos que no tienen una relación directa entre la jerarquía. Ésos objetos no siempre deben de tener un método que los permitan instanciar.

En estos casos no se puede proporcionar una única implementación porque dependen del tipo de elemento que vaya a usar sus métodos. Se denominan interfaces y no están en ninguna jerarquía de clases y no heredan nada.

Como es una clase sin métodos de otras clases superiores a las que heredar, cuando se usan las interfaces de Java se ha de garantizar la inclusión de sus funciones en el código y la implementación de aquellas funciones que se quieran usar.

Por ejemplo la interfaz de Java *Thread*. Java ya sabe como manejar los *threads*, su espacio de memoria, que es un proceso diferente del proceso que lo creo, etcétera. Pero se ha de adaptar a todas las situaciones en que se puede usar y depende de la implementación que el programador haga para modelar algo.

3.4 Interfaces gráficas

De todo este proyecto esta es la parte que finalmente importa. Por cumplir la finalidad docente del proyecto y porque la mayoría de aplicaciones disponen de una. Recordemos que la interfaz gráfica debe actuar como mediador entre el alumno y el compilador VEX.

Es muy importante centrarse en el diseño de ésta. Ir directamente a su implementación sin dedicarle tiempo es un error que suele pagarse caro. Hay que pensar el diseño como si fuera la mayor estructura de componentes de todo el código, de forma que cada componente pueda ser fácilmente comunicado con otro.
Una mala definición casi siempre provoca una alta generación de excepciones que deberán tratarse y/o controlar con bastantes estados, cuando deberían simplificarse al mínimo para ser una aplicación lo más fiable posible.

A continuación se explican algunas recomendaciones y técnicas útiles para realizar cualquier interfaz gráfica.

Lo más importante al diseñar e implementar una interfaz gráfica es:

- El usuario debe verlo como una herramienta de fácil uso pero con un alto potencial, es decir, que aprecie su aportación a cumplir tareas específicas y que cantidad de información aporta cada una.

- La comunicación debe ser efectiva tanto funcionalmente como estéticamente, aunque también ha de basarse en el perfil de los usuarios finales ya que influye tanto el contexto en que lo usara como su grado de dominio en la materia.

- La simplicidad en su diseño, que todo sea accesible y fácilmente reconocible, la curva de aprendizaje no debe ni existir, el usuario principiante debe alcanzar el grado de dominio total sin esfuerzo.

- La simplicidad suele conseguirse si el papel de un elemento puede hacerse por otro con pocas modificaciones.

3.4.1 Técnicas para una correcta visualización

- Escoger que elementos deben sobresalir en el diseño según su importancia.

Hay dos aspectos importantes al respeto:

- Ley de Fitt: cuanto más grande y más cercano al puntero del ratón es un objeto, más sencillo es hacer clic sobre el.
- Si se mueve el cursor al borde de la pantalla, va justamente al último píxel. No importa la velocidad en que se moviera, si un objeto esta en el borde es como si tuviera un tamaño infinito.

- Ordenar dentro de una misma función aquellos elementos que se comporten como un modulo, que se separen claramente del resto de módulos con una frontera imaginaria y clara.

- Imaginar la ruta natural del ojo ayuda a diseñar los módulos, implica establecer un acuerdo entre organización del objeto a poner y la acción que desarrolla en ese módulo, por ejemplo cargar un fichero y salvarlo estarán suficientemente separados pero cerca de donde se muestre el archivo ya que el ojo suele mirar alrededor.

Entrada de datos:

- Estableciendo unas condiciones mínimas para el correcto funcionamiento del programa. La entrada de datos no debe estorbar al usuario. Porque el esfuerzo utilizado en usar la aplicación es esfuerzo que no pueden utilizar en la tarea que están intentando realizar.
  - Así se garantiza un uso continuo del programa.

- Se intenta que la interfaz lo haga todo por el usuario, no debe tener que introducir muchos datos. Se debe considerar la productividad del usuario antes que la productividad de la máquina. Se debe aprovechar el potencial actual de las máquinas, para eso estan.

Generación de mensajes:

- Intentar minimizar la cantidad de distracción y de interferencias. Si hay un error debe buscarse una forma de mostrarlo sin que distraiga mucho.
- Incluir un soporte de ayuda en caso de mostrarse información compleja. Hay que establecer una relación entre el contexto donde esta situada la información y la cantidad de información extra que se debe aportar, no debe ser demasiada pero si precisa.

4 Funcionalidad de la interfaz gráfica

La interfaz gráfica pretende mejorar y facilitar el uso del VEX. Deben poderse introducir la mayoría de parámetros opcionales y las acciones para compilar y simular. Ha de hacerse de la forma más fácil posible y con pocas acciones por parte del usuario.

Las interfaz gráfica debe proporcionar bastante información y alternativas para las fases de compilación y simulación sin que el usuario tenga que llegar a usar el simulador VEX en ningún momento, a no ser que quiera experimentar con alguna de las varias opciones que permite y no incluidas en la interfaz gráfica.

Requerimientos mínimos:

El único requisito para su correcto funcionamiento es la resolución mínima para poderse visualizar la interfaz gráfica correctamente, debe ser de 1152x752 píxeles. Es una buena resolución para mostrar bastante información y su uso en los ordenadores seguramente no comportara problemas. Esta resolución hace años que esta soportada por todas las tarjetas gráficas y monitores.

El resto de requisitos como disponer de un S.O. Linux con

- El compilador gcc
- El compilador VEX
- El Java JDK 1.6.0

Ya están incluidos en el *LiveCD.*

Los siguientes apartados son una breve descripción de los módulos que se compone la interfaz gráfica. Luego se explica con más detalle los elementos y acciones que permiten hacer. Para finalizar se explican las técnicas usadas en la interfaz.

4.1 Elementos principales:

Los módulos principales son:

1) Módulo de edición del programa y de la configuración: permite la edición, selección, carga de archivo e almacenamiento en memoria de los 3 archivos necesarios (ficheros.c/ ficheros.mm / vex.cfg). También permite visualizar los errores generados en tiempo de compilación o simulación.

2) Módulo VLIW: se muestra el lenguaje ensamblador VLIW. La información se rellena en una tabla tras la compilación. Incorpora soporte gráfico para determinar que parte del código origina un salto a una etiqueta.

3) Módulo de visualización del resultado tras la simulación.

La elección de usar 3 módulos viene determinada por el número de fases del simulador VEX. La primera fase diseñada a escribir los datos de entrada del programa. Las otras dos fases son las que tiene que realizar VEX para generar los resultados.

4.2 Distribución de los módulos

La separación de los módulos es vertical mientras que los elementos de cada módulo se separan horizontalmente.
La elección viene determinada porque es la que mejor se ajusta al flujo de datos de entrada/salida del compilador. Con 3 módulos y un diseño horizontal a pesar de disponer de más espacio por la resolución de los monitores, no se considero idónea.
La tabla del segundo modulo, que es la que muestra más información, si fuera horizontal habría que hacer varios movimientos para desplazarse, además el soporte gráfico de ayuda adyacente requiere bastante espacio.
Aunque la tabla del segundo módulo es la que más datos muestra, no es necesariamente el modulo más importante, en el diseño los tres son casi igual de importantes, no hay ninguno que destaque y sea más grande que el otro.
En la figura 4.1 es una vista de JavaVEX que permite analizar el comportamiento de la función multcheck.c sobre la arquitectura por defecto del procesador VLIW. En la

figura están marcados los 3 módulos. Arriba de todo el de edición del programa y configuración, en medio el módulo VLIW, y abajo el de visualización de resultados de la simulación.

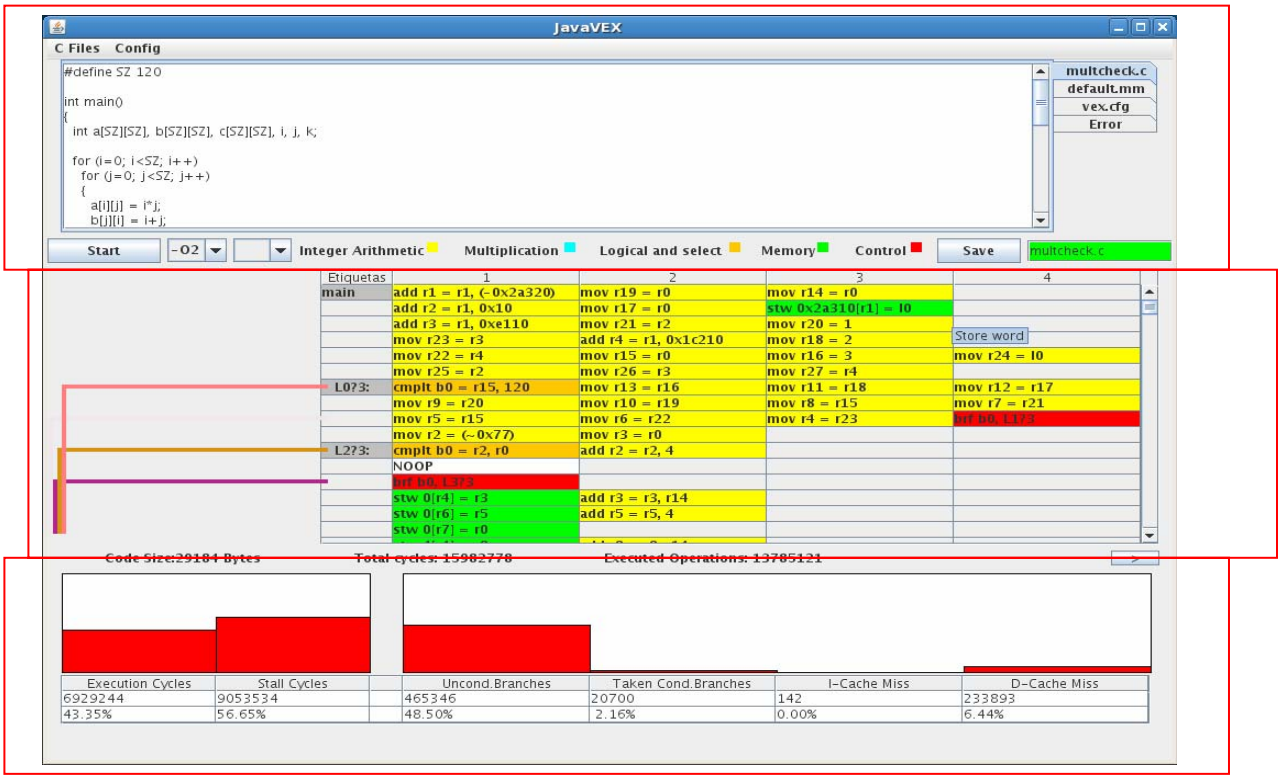


Figura 4.1 Vista de JavaVEX con un ejemplo.

4.3 Acciones que realiza la interfaz y su funcionamiento

4.3.1 Módulo de edición y configuración hardware

Solo ejecutarse la aplicación se muestran por defecto dos menús, están en la parte superior izquierda de la figura 4.2. El menú CFiles que permite seleccionar los programas C almacenados en el disco duro y, el menú Configs que deja selecionar los ficheros de configuración de recursos hardware.

Debajo de los menús esta el editor de textos. Éste tiene una función en codigo C vacía y al lado 4 pestañas que permiten visualizar: el programa C, o la arquitectura del procesador VLIW definida bajo el nombre default.mm y vex.cfg. La cuarta pestaña es para ver los errores.

En la parte baja de la figura 4.2 se encuentran de izquierda a derecha: el botón Start, las dos listas de optimización, leyendas que sirven para el siguiente módulo, el botón Save y un campo de texto para poner un nombre al fichero que se salva.

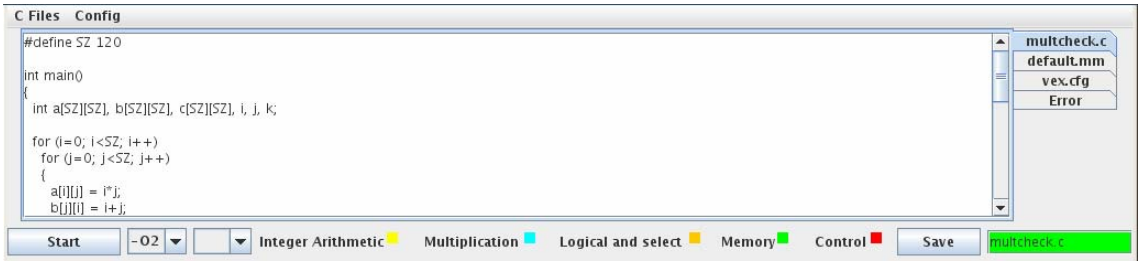


Figura 4.2 Módulo de edición del programa y configuración.

Como el código que se editara o cargara no será el de un programa C que contenga bastantes líneas de código, sino que será el de una función o pocas, no se muestran muchas líneas.

4.3.2 Acciones del módulo de edición

Menús:

Cuando se pulsa en C Files se despliega un menú que permite seleccionar uno de los ficheros.c que están en la carpeta */bin/fitxers.* La selección de uno de ellos comporta mostrar y su código en la primera pestaña. La figura 4.2 es el resultado de seleccionar en C Files el fichero multcheck.c

Lo mismo sucede para el menú Config. Es un menú que permite seleccionar un archivo con extensión .mm y que se almacenan en la carpeta */bin/configs.* La selección de un archivo.mmm comporta mostrar su código en la segunda pestaña.

- Cuando se muestra un código se antepone la pestaña donde se muestra al resto de pestañas.

Editor de textos y pestañas:

La selección de cualquiera de las 3 primeras pestañas muestra el archivo que tenga como titulo. La 3ª pestaña siempre tiene como nombre el vex.cfg que se encuentra en la carpeta */bin*, como solo es un fichero no se incluye en ningún menú.

La última pestaña sirve para ver el error que ha sucedido durante el proceso de compilación o simulación. Cuando es seleccionada, todos los botones del módulo de

edición no se podrán usar. Ni el campo de texto adyacente al botón Save, ni el área de texto de esa pestaña, evitando sobrescribir ficheros útiles.

Botones:

Save: Almacena en la carpeta correcta el archivo que se este visualizando. Al salvar un fichero tiene en cuenta que el nombre del archivo corresponda al formato de ficheros que soporta la pestaña actual.

1) Al pulsar Salvar se pregunta al usuario si esta seguro de guardar ese archivo con el nuevo nombre, por si le dio involuntariamente.

2) No se añade el archivo en los menús directamente si se salva. El usuario mientras use JavaVEX puede ir añadiendo nuevos archivos en las carpetas. Por eso los menús se actualizan al pulsar el menú.

Start: Compila y simula el programa C.
Si hay error en tiempo de compilación, como por ejemplo, que un valor asignado a un recurso no posible, o porque la función C esta mal implementada, se mostrará en la pestaña 4 el primero de los dos fallos que suceda.

1) Saldrá solo el primer error que se genere, tal y como sucede en el la compilación del VEX. Primero salen los errores de configuración sino los del gcc.
2) Al pulsarse Start y se ha modificado alguno de los dos archivos sin ser salvados, entonces aparecerán tantas ventanas como archivos afectados. Si la respuesta es no, se compila usando la última versión salvada y se actualiza el texto en la pestaña que le toque. Si la respuesta es sí, se salva el contenido actual sobrescribiendo el viejo fichero.

Si no hay error en la fase de simulación. VEX puede ejecutar el código C traducido a formato VLIW y mientras lo ejecuta se basa en los parámetros del fichero vex.cfg que entran en acción en esta fase. Si hay error en esta fase es porque los valores del vex.cfg

no son correctos o porque al ejecutar el código VLIW hay un error en la implementación que en la compilación no se detecta.

- Por ejemplo cuando hay acceso a una posición de un vector inexistente, el compilador no lo puede predecir. El solo mira que los datos sean correctos, que sigan unos patrones de sintaxis de lenguaje, pero no los simula.

El error se visualiza en la pestaña 4 con el texto "Error in code execution or some parameter value is out of range in vex.cfg".

Campo de texto:

Permite poner el nombre con el que se quiere guardar el archivo seleccionado. Solo se usa cuando se selecciona una de las dos primeras pestañas, según cual, permitirá guardarlo con extensión .c o .mm.

Ventanas:

Las ventanas se sitúan entre medio de los dos botones que las generan, tal y como muestra la figura 4.3 y en la figura 4.4

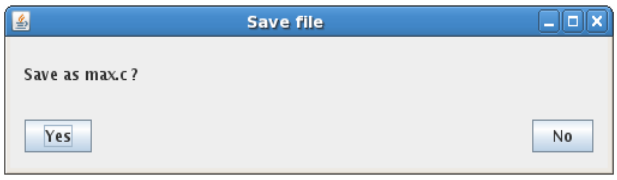


Figura 4.3 Salvar fichero.

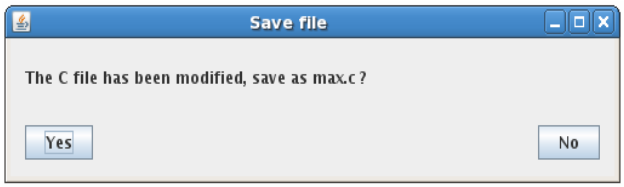


Figura 4.4 Actualizar fichero.

La figura 4.3 aparece al pulsar Save. La figura 4.4 solo aparece si al pulsar Start difiere uno de los textos de las 3 primeras pestañas con los archivos almacenados en memoria.

Se considera que el usuario no las ignora, pues su aceptación después de realizar otras acciones no garantiza el funcionamiento correcto de la aplicación.

Listas de ítems

Al pulsar en la flecha de las dos listas de optimización, aparecen todas las posibles optimizaciones que se pueden aplicar al programa C.

4.3.3 Módulo VLIW

En la parte media de la interfaz, solo ejecutarse la aplicación se ve una tabla con columnas vacías y a su izquierda un espació reservado al dibujo de líneas.

Permite visualizar 16 filas. Cada fila representa una instrucción VLIW y puede llegar a tener como máximo 4 operaciones, por eso tiene 4 celdas/columnas.

- La mayoría de operaciones tienen suficiente espacio en cada celda, solo hay algunas excepciones en que supera el tamaño. Para solucionarlo cada columna puede agrandarse.

En la siguiente figura 4.5 se puede ver mejor.

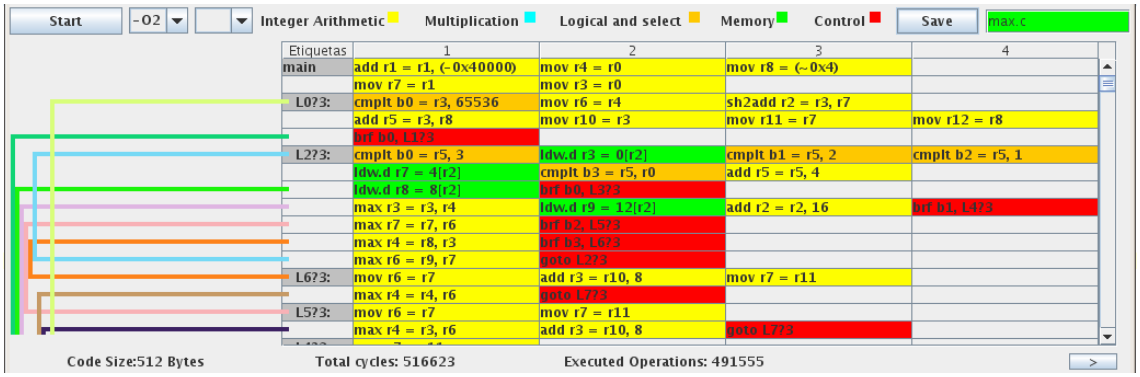


Figura 4.5 Módulo VLIW.

Sobre la tabla de la figura 4.5:

- Encima tiene 4 leyendas para ver a que tipo pertenece cada operación.
- Debajo a la izquierda esta el tamaño en *bytes* que ocupa el código estático del programa.
- A su izquierda están el soporte visual. De entre todas las líneas, por ejemplo las de color azul, rosa y naranja me dicen que en estas 16 filas hay 3 operaciones que apuntan a las etiquetas L2?3, L5?3 y L6?3 respectivamente
- Contiene las operaciones del código VLIW y el color de fondo de cada celda indica el tipo de operación. El significado del color esta explicado con 4 leyendas ubicadas encima la tabla.

La tabla puede llegar a tener hasta 1450 instrucciones VLIW pero nadie mirara cada una de ellas. Si el número de instrucciones VLIW es elevado, sirve para hacerse una idea de cómo se las ingenia el compilador según el modelo de procesador y el grado de optimización.

- Si el código supera las 1450 líneas no se mostraran las que superen este número. No suponía gran dificultad en la implementación pero si el código ensamblador resultante es tan grande, es porque solo interesara ver los datos en el módulo de visualización de resultados.

4.3.4 Acciones del módulo VLIW

Solo se permiten acciones sobre tabla. Las que más se usarán son:

1) Poder desplazar la tabla verticalmente con la ayuda de la barra vertical (ubicada a su derecha). Si se mueve, se actualizan las líneas adyacentes.
2) Agrandar/reducir cada columna de forma independientemente.
3) Visualizar un resumen de lo que realiza la operación donde este el cursor dentro la tabla. No necesita pulsar la celda.

4.3.5 Modulo de visualización de los resultados de la simulación

La tabla ubicada en la parte baja de este módulo (ver figura 4.6) se pueden ver:

1) En la primera fila: los ciclos de ejecución, ciclos de espera, numero de saltos incondicionales, numero de saltos condicionales que saltan, y el número de accesos que han originado un fallo en las *caches* de instrucción y de datos.
2) En la segunda fila: el porcentaje de los datos de la primera fila.

Los recuadros que están encima de la tabla son los porcentajes de la segunda fila representados en barras de color rojo.

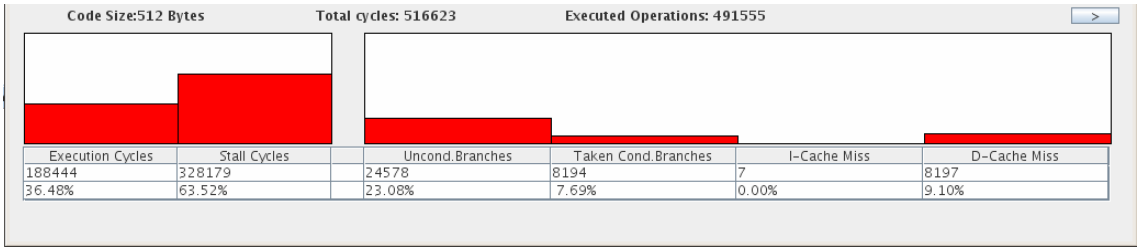


Figura 4.6 Módulo de visualización de los resultados de la simulación.

Las tres gráficas que se muestran son:

1) Gráfica de ejecución y espera: muestra los ciclos de ejecución y ciclos de espera en el procesador, se pretende maximizar los de ejecución y ello se traduce en mayor rendimiento del CPU.

2) Gráfica de salto y *cache*: es importante para el estudio de los ciclos de espera. Se muestra el número de saltos incondicionales que hay en el código, el número de saltos donde la condición se cumple, y los fallos de las *caches* de instrucción/datos.

3) Gráfica de rendimiento (ver figura 4.7): muestra el grado de utilización del máximo número de operaciones simultáneas que permite ejecutar el procesador.

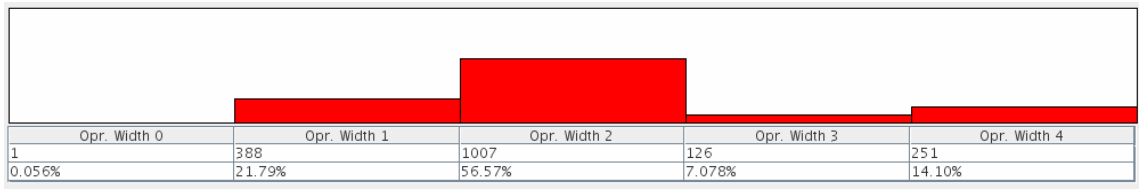


Figura 4.7 Gráfica de rendimiento.

4.3.6 Acciones del módulo 3

Solo hay un botón que permite visualizar las 2 primeras gráficas o la tercera gráfica.

4.3.7 Técnicas usadas en la interfaz

A partir de las técnicas mencionadas en el apartado 3.4.1 de la memoria, se pasa a describir cual ha sido su uso en la interfaz gráfica.

Simplicidad:

- El botón Save se puede realizar para todos los archivos (.c/.mm/ vex.cfg).
- El botón Start realiza la compilación y simulación a la vez. Simplifica los dos procesos más importantes del JavaVEX en uno. Ambos procesos se realizan sobre los archivos que aparecen en las pestañas del editor de texto, es pues, un botón polimórfico.
- Si puede elegir cualquier combinación de grado de optimización y nivel de *loop unrolling* con solo dos listas de ítems.
- Un menú para los programas y otro menú para los de configuración del hardware. El uso de menús permite la carga de un archivo fácilmente. La mayoría de interfaces gráficas tienen uno.

Entrada de datos:

Un campo de texto para escribir bajo que nombre se guardara, si el formato del nombre es correcto se mostrara una luz verde, en caso contrario será rojo. El feedback con el usuario es instantáneo. Esta idea es la misma que usa el navegador FireFox Mozilla cuando se quiere buscar texto en una página web.

Generación de mensajes:

Se muestran mensajes al salvar para confirmar si la acción es aceptada por el usuario o para negarla si ha sido pulsado el botón involuntariamente. El mensaje se vuelve a mostrar si al pulsar Start.

- El mensaje que confirma que el fichero vex.cfg ha sido guardado, siempre tiene el mismo texto porque el nombre del fichero no es modificable.

- No hay muchos mensajes. Solo los necesarios para garantizar los ficheros usados en módulo de edición sean los mismos con los que realizo la ultima simulación.

El soporte de ayuda cuando se muestra información de lo que realizan las operaciones de la tabla del modulo VLIW.

5. Implementación de la Interfaz Gráfica

En este capítulo sobre la interfaz gráfica se explica como ésta interactúa con el compilador VEX, como se implemento con Java y como se incorpora junto al S.O. Fedora Core 6 en un *LiveCD* igual al presentado en esta memoria.

5.1 Pasos para hacer un ejemplo simple con VEX.

El proceso más simple que se puede realizar en VEX es:

1)./cc –c –O2 fichero.c → compila el fichero.c a partir de la arquitectura definida por defecto (que es la inicialmente cargada en default.mm). Luego genera un archivo temporal que contiene el código en instrucciones VLIW (fichero.s). Finaliza generando el archivo binario (.o) y eliminando los archivos temporales creados en este proceso.

2)./cc –o ejecutable fichero.o → con las librerías de simulación del VEX y la ayuda del gcc nativo del sistema, se genera el ejecutable que contiene la imagen de la simulación.

3). /ejecutable → ejecuta la simulación. Genera el fichero con los resultados de la simulación. Si no hay el archivo vex.cfg parte de un modelo de procesador preestablecido.

5.1.1 Configuraciones que permite VEX

Como lo hace el simulador VEX para poder usar el archivo temporal .s que contiene las instrucciones VLIW?

- Se añade a 1) la opción –ms que evita su eliminación. El uso de otra configuración para definir el comportamiento de las unidades funcionales se hace añadiendo –fmm=<archivo.mm> a 1).

La optimización se hace eligiendo entre los parámetros –O1,-O2,-O3,-O4 más el parámetro opcional –H0,–H1,-H2,-H3,-H4.

-Ox realiza:

-O1: Optimizaciones escalares.

-O2: *Loop unrolling* mínimo y técnica de compilación *trace scheduling* (esta es la opción por defecto).

-O3: *Loop unrolling* básico y técnica de compilación *trace scheduling*.

-O4: *Loop unrolling* avanzado (como –H4).

Hx es una opción que sobrescribe el *loop unrolling* definida en cualquier optimización –Ox (por defecto no usa ninguna):

-H0: No hay *unrolling*.

-H1: *Unrolling* basico.

-H2: *Unrolling* medio.

-H3: *Unrolling* fuerte.

-H4: *Unrolling* avanzado.

5.1.2 Uso del VEX en JavaVEX

Todos los archivos usados en compilación y simulación están en la misma carpeta */bin*. Pero en JavaVEX se separan los archivos C en la carpeta */bin/fitxers*, a los de configuración (default.mm) en */bin/config*. Cuando se generan de los temporales .o y .s se mueven a */bin/fitxers*.

De esta forma, en la aplicación los comandos pasan a ser:

1) ./ cc –c –ms -Ox [Hx] ./fitxers/fichero.c –fmm=./configs/fichero.mm

2) ./ cc –o estadísticas ./fitxers/fichero.o

3) ./estadísticas

5.2 Implementación en Java

En este apartado no se pretende adentrar mucho en la implementación del código de la aplicación. En los siguientes apartados se explica:

- El uso del programa NetBeans para la implementación de Java.
- Las funciones y objetos más usados en las interfaces gráficas y el uso que hace de ellas JavaVEX. Esto sirve para dar una idea de los componentes de la clase Swing que más se usan. Bastantes funciones compartidas por la clase AWT porque ésta es la superclase de la otra.
- En pseudo-código algunas de las funciones más importantes y un diagrama de clases.

Antes de continuar se expone las razón que me llevo a usar Java en la implementación de JavaVEX. El proyecto se podía hacer con un lenguaje de similares características como C++, pero se escogió Java para adquirir más experiencia de la aprendida en esta carrera. Java es un lenguaje muy usado actualmente gracias a su alta portabilidad y su nivel extra de abstracción y protección que ofrece su interprete, el *Java Virtual Machine* (JVM), entre diferentes dispositivos o S.O. Cuando hice las primeras pruebas con este lenguaje, podía ejecutar directamente la interfaz gráfica sobre Fedora o Windows. El único gran inconveniente de Java respecto a C++ es su lentitud porque debe ser interpretado cuando el otro no.

5.2.1 NetBeans [36]

Es un programa orientado a la creación de interfaces gráficas con Java.

5.2.1.1 Caracteristicas de NetBeans

- Dispone de un entorno grafico que permite el acceso y modificación de objetos sobre la interfaz gráfica a crear de una forma sencilla y clara (ver figura 5.1).
    - En las propiedades de un elemento se pueden observar las funciones más importantes de la clase que pertenece el objeto y de las clases padre que es heredera.
- Dispone de una paleta (palette) ubicada a la derecha del programa. Es donde se muestran los elementos gráficos más usados de Java en el diseño de interfaces. Solo se debe seleccionar un elemento y arrastrar.
    - Partiendo de los más básicos, se pueden ir ensamblando con otros de nivel superior y una sintaxis visual describe como realiza los ensambles.
    - Algunos elementos ya vienen ensamblados sin que tenga que hacerlo el usuario, suelen ser objetos que probablemente conllevan el uso de otros. Por ejemplo labarra desplazadora dentro de un área de texto.
- Si no se usa NetBeans (NB), Java presenta varias formas de diseñar interfaces gráficas, pero tanto el aspecto como el grado de complejidad en implementarlas lo hacen idóneo para implementarlas.
    - Genera con código las relaciones entre los elementos según su ubicación y su alineación.

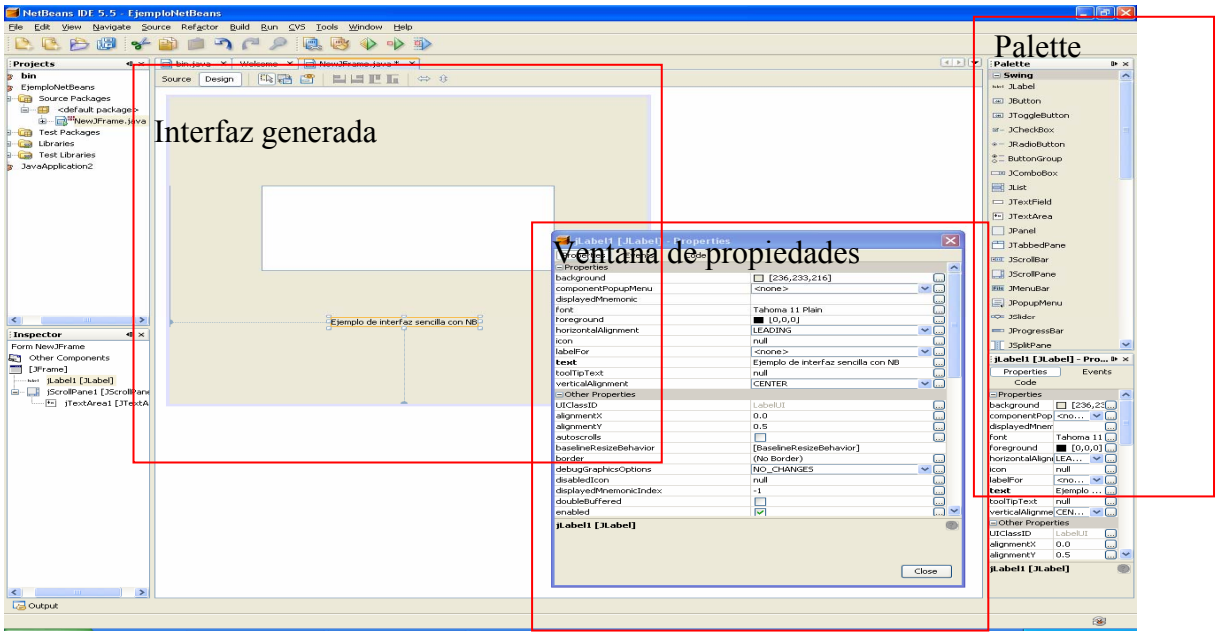


Figura 5.1 Vista de NetBeans.

5.2.1.2 Ejemplo del uso de NetBeans

Por ejemplo, para explicar como NB genera el código Java se usa la siguiente figura 5.2.

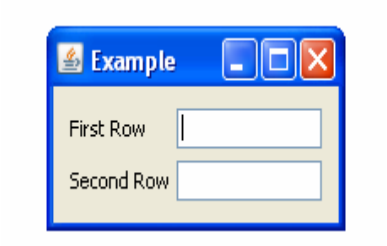


Figura 5.2 Ejemplo sencillo con NB

Una vez se ha creado la ventana y añadido los textos y campos de texto anexos. NetBeans genera con objetos y funciones de Java una rejilla sobre la ventana Example que contiene dos grupos, uno vertical y otro horizontal:

- El grupo vertical contiene dos subgrupos paralelos entre si. Un subgrupo tiene las etiquetas *First Row* y *Second Row* y el otro los dos campos de texto.

- El grupo horizontal contiene dos subgrupos paralelos entre si. Un subgrupo con la etiqueta *First Row* y su campo de texto y otro con *Second Row* y su campo de texto.

Así lo suele hacer para cada conjunto de elementos en los que pueda encontrar un patrón que los relacione y posicione entre si.

No se pretende profundizar ni ahora ni más adelante en la implementación entera de la interfaz gráfica generada por NetBeans para JavaVEX. Pero dada la cantidad de código que ha creado se considero oportuna una breve explicación sobre la idea en que se basa.

5.2.1.3 Primeros pasos con NetBeans para crear la interfaz gráfica

Al empezar a manejar NB, cuando creas un proyecto te permite escoger entre diferentes tipos de proyectos, en éste caso es el desarrollo de una aplicación Java y que contenga una clase principal.

NB crea un paquete del tipo JAR que contiene los ficheros del proyecto creados ahora y los que se vayan añadiendo posteriormente, el paquete podrá ser ejecutado con el comando java –jar.

Se añade un fichero JFrame (ventana) con un nombre y ya se dispone de la ventana principal de la interfaz, solo bastara arrastrar de la paleta a esta ventana con el ratón los elementos que se vayan a usar y colocar donde se quiera. Por ejemplo, en la figura 5.1 se han arrastrado de la *palette* un área de texto y una etiqueta.
Por ahora ya se ha creado un archivo Java que es la extensión de la clase JFrame. Con la extensión ya se pueden usar métodos declarados en la clase padre que permiten el manejo de la ventana. Como por ejemplo, establecer/obtener su posición, tamaño, titulo, etcétera.

La paleta contiene muchos objetos de diferentes clases, las más usadas son las de la clase Swing o *Abstract Window Toolkit* (AWT). Ambas son buenas clases para el diseño de interfaces gráficas (recordar que Swing es una clase que hereda de AWT).

Para el proyecto se ha usado Swing porque ya presenta componentes creados como el JTabbedPane o JTable que se forman de varios componentes unidos del AWT. El primer componente se usa para las pestañas que contienen los archivos que usa el simulador VEX y el segundo componente es la tabla donde se muestran las instrucciones VLIW.
Solo por arrastrar un objeto a la ventana creada, NB ya instancia ese objeto en la nueva clase, lo inicializa con algunos datos como su posición, nombre, y luego ya depende del tipo del elemento, por ejemplo si es un botón, se pone su texto inicial.

Algunas propiedades del elemento pueden ser modificados gracias a que NB permite un acceso rápido a algunas funciones de la misma clase o superiores, como definir el color, agregar un texto que saldrá al mover el cursor encima, su alineación respecto los otros elementos, y así una larga lista de propiedades. Pero quizá, la más interesante, es la agregación de eventos a los objetos.

5.2.1.4 La importancia de los eventos

Bien, una vez inicializados los elementos y escritas algunas funciones para modificar algunas propiedades que no ofrece NB. Se dispone de una ventana vacía, sin texto ni datos ni gráficos que mostrar.
En una interfaz gráfica es indispensable la interactividad, es el mediador entre el usuario y en este proyecto, el simulador VEX. La comunicación debe finalizar con una respuesta que deje satisfecho a cualquier usuario.

En Java ésta interactividad es únicamente alcanzable usando eventos. Cuando el sistema operativo recibe un evento, éste es enviado a los programas que harán de él un uso particular.
Igual que la estructuración de las clases de Java, los eventos son clases que extienden a otras clases de eventos en un orden jerárquico. Cuando se produce un evento, dependiendo del tipo que sea, se representa como objeto de una de estas clases.
Para recibirlo, se debe añadir un objeto receptor del tipo de evento que queremos tratar al elemento de la interfaz gráfica que queramos. En resumen, a un objeto fuente como un campo de texto se le puede añadir que trate un evento (o más) al pulsar, mientras se pulsa o se libera un botón del ratón. Son tres eventos distintos que serán recibidos por un objeto receptor del ratón.

Recordando lo comentado en el capítulo de conceptos básicos sobre interfaces de Java. Las clases de eventos son interfaces, ya que son múltiples los elementos a los que se pueden asignar y diferentes, para luego hacer un uso distinto de cada uno.

5.2.1.5 Que se hace con los eventos

NB permite agregar eventos a cada componente de la interfaz. El programa ya crea la relación del objeto fuente (el que puede recibir eventos) con el tipo de objeto receptor que queramos sobre el. Al programador le basta implementar que debe hacer cuando se activa.

En una interfaz gráfica hay bastantes eventos. Un evento en un objeto casi suele originar un cambio de estado sobre las propiedades de otros objetos. Lo que se debe hacer es:

- Cuando el usuario clica, selecciona (o el evento que sea), intuye que realiza ese elemento. Éste no suele variar su funcionalidad en el programa después de usarse.

- Al implementarse el cambio de estado, el elemento ha de adaptarse a su nueva situación, que el usuario entienda todos los cambios que origina el elemento usado y a los otros elementos y el porque de ello.

- Siempre se busca que el número de cambios sea el que se espera el usuario, y en caso contrario, el número ha de ser reducido al mínimo para no crearle confusión.

Un ejemplo del proyecto. Al compilar se espera obtener los resultados, ¿pero y si hay un error en el proceso? Se debe escribir el error en pantalla para que entienda que sucedió y se deshabilitan los botones Start y Save.

- Se deshabilitan porque no se podrá recompilar. Sino se volvería a generar el mismo error, y un error no interesa salvarlo, se quiere corregir lo antes posible para cumplir lo que se esperaba desde un inicio.

En el proyecto todos los eventos generados en la interfaz gráfica son solo recibidos por objetos receptores del ratón.

- Solo el campo de texto adyacente al botón Save es el único que responde a los eventos generados por el teclado.

5.2.2 Funciones usadas en el tratamiento de ficheros, E/S

Para acceder a los ficheros que el simulador VEX maneja, se han tenido que usar las clases siguientes clases de Java:

- BufferedReader y FileReader. Son clases que permiten leer de un corriente de datos de entrada. Ambas extienden la clase que permite la lectura de caracteres Reader. Con estas clases y sus funciones se puede realizar la lectura de un fichero.

- Cuando se genera un error porque el fichero a leer no existe, en JavaVEX se aprovecha para mostrar un error.
  Profundizando más en el tratamiento del error. Dicho archivo siempre se intenta leer al pulsar el botón Start, pero si hay algún fallo en el archivo vex.cfg, no se creará y se genera el error.

- De la misma manera que las dos clases anteriores permitían leer ficheros, las clases BufferedWriter y FileWriter sirven pero para escribir. Si no existe el archivo se crea, y si existe se sobrescribe.
  - Una vez creado el *pipeline* con los ficheros, se usa la función readLine para ir leyendo cada línea del texto o la función write para escribir el texto entero.

5.2.3 Funciones más importantes del módulo de edición y configuración hardware

Menús: en los dos menús, para cargar los ficheros que hay en las dos carpetas que acceden (*./fitxers/* y *./configs/*). Se usan las funciones de la clase File.

- File sirve para determinar el directorio, mientras que listFiles y getName para obtener los nombres de los ficheros y añadirlos al menú que corresponda.

Gestión de ficheros: la implementación de mostrar los datos de un fichero o escribir al fichero vienen por las acciones realizadas sobre el JTabbedPane (el panel con pestañas), sus funciones más usadas son:

- getTitle, permite obtener el titulo de las pestañas que contienen el fichero que muestra, son estos títulos los usados cuando se ejecuta VEX. Para poner un nuevo titulo se hace servir setTitle.

Las clases getSelectedIndex y setSelectedIndex sirven para saber que pestaña esta siendo vista ahora o para seleccionar cual se quiere mostrar. Se usa por ejemplo al salvar. Para escribir o leer del área de texto asociado a una pestaña basta con las funciones setText y getText.

Botones y campo de texto: las funciones setText y getText también se aplican al campo de texto para salvar bajo un nuevo nombre. Cuando hay error se deshabilitan: el

campo de texto con el método setEditable, los botones con la función setEnabled. Para comprobar el estado actual de ambos componentes se hace con isEditable/isEnabled.

5.2.4 Ventanas emergentes

Los mensajes que salen como ventanas emergentes, se han creado añadiendo al paquete JAR dos JFrame nuevas, es decir dos ventanas nuevas donde agregar elementos. Una tiene dos botones y la otra uno. Al pulsarse los botones se actualizan algunos elementos de la interfaz gráfica.

- Para asignar su posición en la pantalla se ha usado setLocation que se basa en las coordenadas de la pantalla.

5.2.5 Funciones más importantes del módulo VLIW

La tabla que tiene este módulo es fácilmente configurable en las propiedades de NetBeans. Estas permiten asignar las filas/columnas y a cada columna asignar de que tipo será (carácter, entero, etcétera).
La función setValueAt, pasando por parámetro una coordenada, nos permite asignar un valor a cada celda.

Con esto basta para poder asignar un dato estándar de Java a cualquier celda. En JavaVEX se usan colores de fondo para que el usuario, sin prestar atención a la operación que hay en cada celda, pueda hacer una lectura rápida de cómo se han organizado las operaciones según el tipo de operación. Por ejemplo, si se han asignado más unidades multiplicadoras, vera si se aprovechan todas las unidades fijándose solo con los cuadrados de color azul claro en cada fila de la tabla.

Todos los objetos que se usan en cualquier interfaz gráfica con las funciones setBackground o getBackGround se controla el color de ese componente. Pero hay un problema aquí, el color a cambiar es el de cada celda y no toda la tabla. Es cuando se debe añadir una interfaz de Java a la tabla. A cada celda se le añadira un objeto no estándar y se debe implementar su comportamiento. En JavaVEX se asigno una etiqueta a cada celda, así se puede escribir su texto y controlar el color de fondo.

La asignación de la interfaz Java se hace con la función setDefaultRenderer, y la interfaz que mejor se adapta es la TableCellRender. Ésta interfaz de Java, sin implementar nada, se encarga de recorrer todas las celdas. Lo único que se debe programar es mirar el contenido de esa celda, ver que tipo de operación tiene, asignarle un color, y por ultimo, agregar la descripción de que realiza esa operación.

El código será de un tamaño considerablemente superior a las 15 instrucciones que puede mostrar la tabla, para la mayoría de bloques ya es suficiente.
La barra vertical se ha de tener en cuenta para actualizar las líneas adyacentes a la tabla, y así desplazarse junto las etiquetas/instrucciones de salto; para saber el valor actual y desplazar todas las líneas tantos píxeles como se haya desplazado la barra se usa getVerticalScrollBar y getValue()

5.2.6 Funciones más importantes del módulo de simulación

Tabla: La tabla que muestra los datos relacionados con la simulación se debe modificar mientras se use la interfaz gráfica, para que sea coherente con la información dibujada en las gráficas.
Se usa la función getColumnModel para poder acceder a las propiedades de todas las columnas. Para especificar manejar las columnas se usan las clases: getColumn para escoger una, se cambia el titulo con setHeaderValue, el tamaño con setPreferredWidth, se eliminan/añaden a la tabla con removeColumn/addColumn.
Barras: Para dibujar las barras o líneas (todo se basa en rectángulos rellenados de un color), se han usado los metodos fillRect, drawRect para dibujar, y la selección del color con setColor.
La función donde deben ir es una función existente en las superclases, cada componente tiene el suyo, JavaVEX sobrescribe la función paint de la ventana JFrame
Esta función será usada para actualizar el contenido gráfico de la interfaz cada vez que se pulse el botón que permite acceder a la tercera gráfica y viceversa. Tambien se actualizan los dibujos cada vez que se mueve la ventana, se minimiza-maximiza, se mueve la barra de la tabla, etc…esto se hace así porque si se pasa alguna otra ventana por encima de la aplicación o se sobrepone algún elemento, los dibujos se van borrando.

5.2.7 Tratamiento de los *strings*

Los *strings* estan presentes en cualquiera de los 3 módulos. Por ejemplo al coger los valores de las estadísticas, para poner la descripción de las operaciones, o para comprobar el formato correcto de los ficheros a guardar. Las funciones más usadas son:

- replace, útil para eliminar espacios en blanco, frecuentemente acompañado de la función split que divide la cadena cada vez que se encuentra con la misma secuencia de caracteres que se pasa como argumento.

- startsWith, equals, contains, endsWith para comparar cadenas de caracteres, suelen ayudarse de substsring y length.

5.2.8 Pseudo-código

Se muestran las funciones al pulsar el boton el botón Start y el proceso que sirve para lanzar comandos al sistema operativo. El primer elegido por su importancia en JavaVEX y el segundo porque usa *threads* y permite interactuar con el VEX. La interfaz gráfica es quien empieza la interacción.

Proceso BotonStartPulsado (evt:evento)
{Precondición: evt es un evento originado por el ratón**}**
inicio
var control:entero ;
    acrhivoC:char;
    archivoMM:char;
fin_var
{Precondición: #(ventanas_emergentes)==0**}**
comprobar_si_se_han_modifico_pestañas1&2();
{Postcondición: Numero de archivos modificados desde la última vez que fueron salvados → #(ventanas_emergentes) =[0,1,2,3] **}**
si (#(ventanas_emergentes) == 0) →
        archivoC=obtener_titulo_pestaña(1);
        archivoMM=obtener_titulo_pestaña(2);

```
        //Compilación
        control=lanzar_comandos (“./cc –c –ms ”+ obtener_nivel_optimización()
        +“ ”+  obtener_nivel_LoopUnrolling()+ “ –fmm=./configs/” +     archivoMM “
        ./fitxers/”+archivoC , 0 );
        inicializar_gráficos_y_tablas ();

        si(control > 0 ) →
                //Hay error en la compilación, el proceso devuelve al finalizar
                //un valor mayor de 0
                mostrar_pestaña(3);
                poner_texto_Campo_de_texto_Salvar(“”);
                poner_color_Campo_de_texto_Salvar(Blanco);
        sino si (control = 0) →
                //Elimina el fichero con los  resultados de una simulación anterior
                lanzar_comandos(“rm ta.log.000”,1);
                actualizar_fichero(“./vex.cfg”);
                lanzar_comandos(“cp ”+ cambiar_ext(archivoC, “.s”)+ “ ./fitxers/”,1);
                lanzar_comandos(“rm ”+cambiar_ext(archivoC,“.s”),1);
                lanzar_comandos(“cp ”+ cambiar_ext(archivoC, “.o”)+ “ ./fitxers/”,1);
                lanzar_comandos(“rm ”+cambiar_ext(archivoC,“.o”),1);
                intepretar_fichero_S(“./fitxers/”+ cambiar_ext(archivoC, “.s”),1);
                actualizar_tabla_móduloVLIW();
                lanzar_comandos(“./cc –o ejec ./fitxers/”+ archivoC,1);
                //Simulación
                lanzar_comandos(“./ejec”,1);
                lanzar_comandos(“rm ejec”,1);
                //Actualiza modulo de visualización con los nuevos resultados
                interpretar_estadisticas_de_ta.log.000&actualizar_módulo3();
                calcular_lineas();
                dibujar_lineas_y_barras();
            fin_si
        fin_si
fin_si
fin_inicio
```

{Postcondición: Si control > 0 se muestra error. Sino se realiza la compilación y simulación. Los ficheros se mueven a ./fitxers/ y se actualizan los módulos VLIW y de visualización de la interfaz gráfica según los resultados**}**

fin_proceso

---

función lanzar_comandos (comando:char , tipo:entero) **: return** entero

{Precondición: comando contiene la secuencia de caracteres del comando a realizar, tipo = [0,1] **}**

inicio

var valor_al_salir: entero

pr: proceso

fin_var

valor_al_salir=0;

pr=Crear_proceso();

ejecutar_proceso(comando);

si (tipo == 0)→

crear_thread_de_lectura_flujodeDatos(Error);

fin_si

valor_al_salir= esperar_finalización(pr);

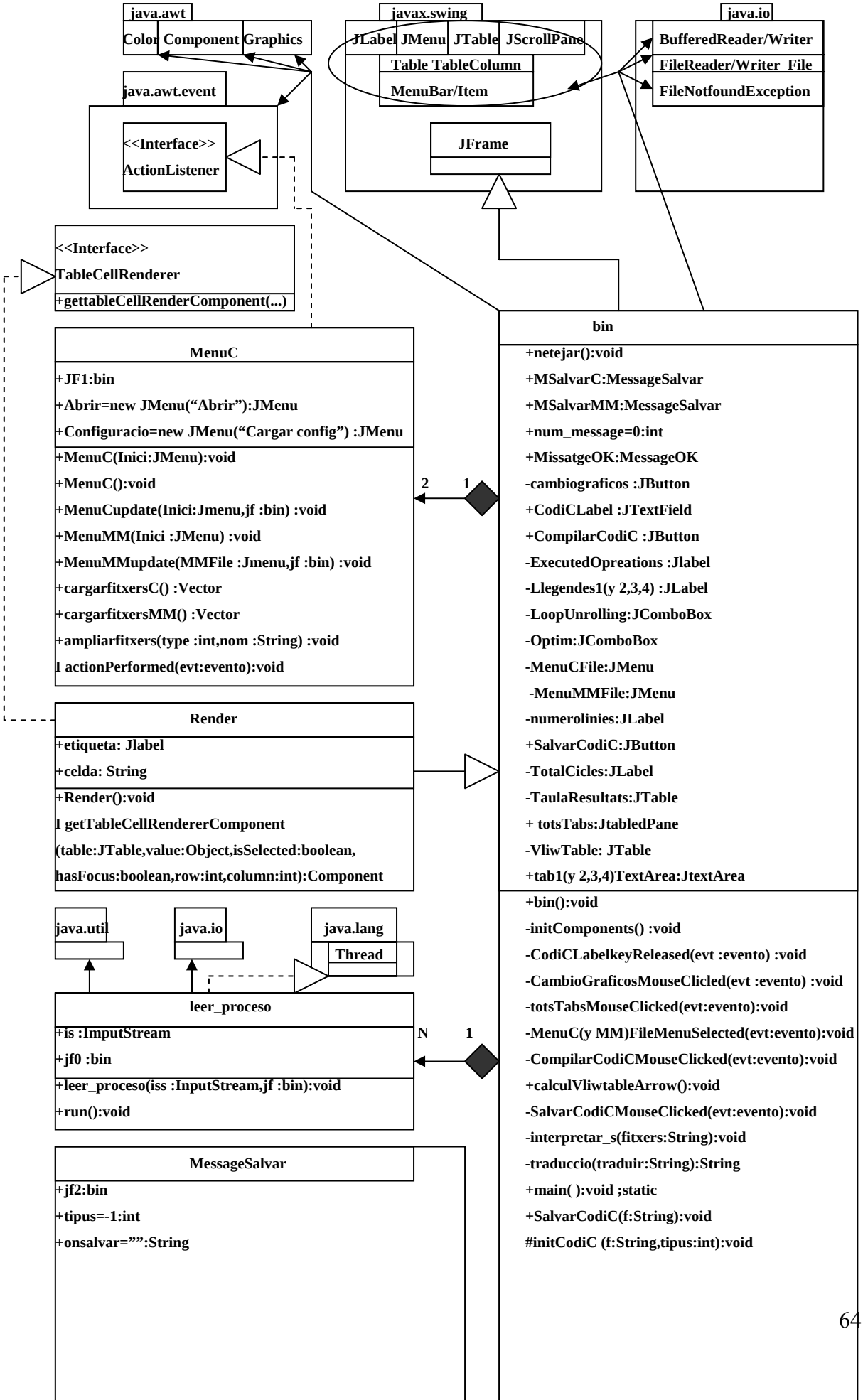
fin_inicio

{Postcondición: valor_al_salir ≥ 0 **}**

fin_proceso

El pseudo-código permite describir las funciones. No se han hecho exactamente igual todas las funciones aquí descritas. Hay algunas líneas de código que se han cambiado a funciones cuando no tiene porque ser, por ejemplo poner_texto_Campo_de_Texto_Salvar("") en Java es tan simple como usar en el objeto CampodeTextoSalvar la función setText(""), se hace con una línea de código.

5.2.9 Diagrama de Clases



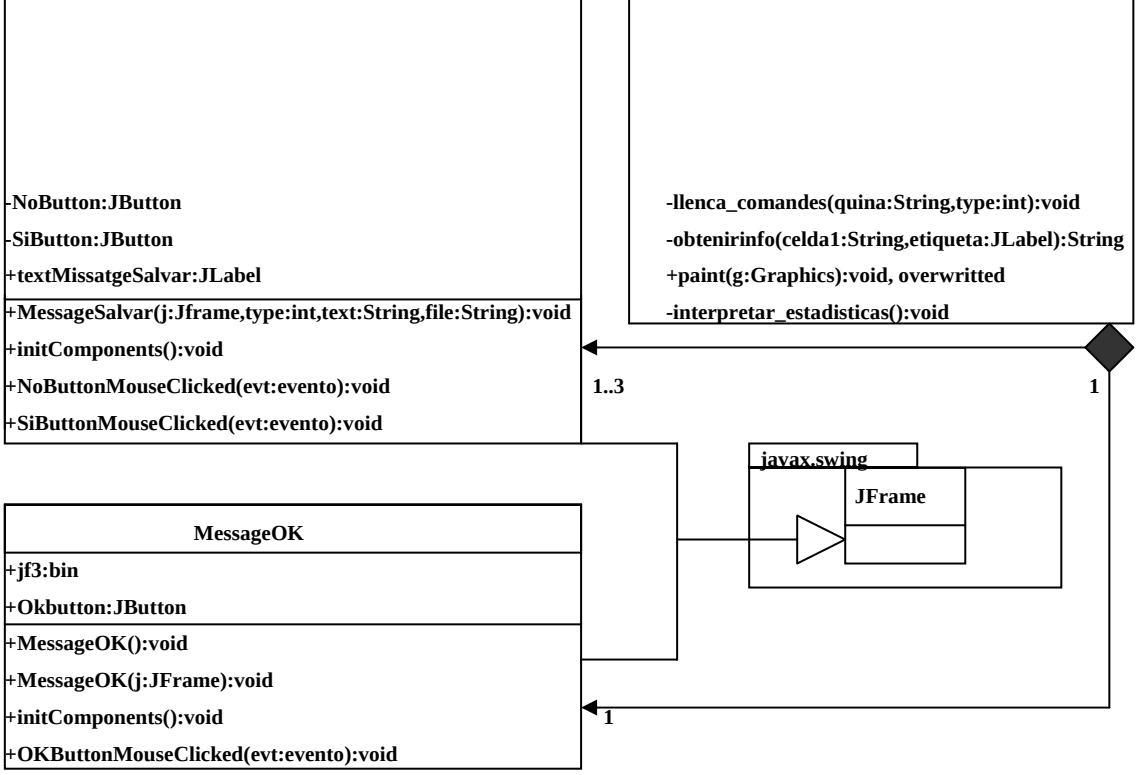


Figura 5.3 Diagrama de clases

Ahora se explican que realizan cada una de las clases:

- bin: como se puede observar en la figura 5.3, bin es la ventana de la interfaz gráfica porque es una clase que extiende la clase JFrame. Contiene:

    - La implementación de los eventos de todos los elementos que componen la interfaz gráfica (de botón, pestaña, menú o de la lista de items)
    - Las funciones para leer y traducir el fichero donde están las instrucciones VLIW generadas por el simulador VEX, para luego rellenar su fila correspondiente en la tabla.
    - Las funciones para leer el resultado de la simulación, con su correspondiente actualización de barras y el cálculo de porcentajes.

- Menu: se encarga de manejar los menus.
- leer_proceso: esta clase describe contiene la implementación de una de las dos clases que Java dispone para crear y manejar *theads*.

La mayoría de *threads* que usa la clase bin son creados cuando se quiere ejecutar un comando al simulador VEX, o para la gestión de ficheros, por eso pueden haber hasta N leer_proceso por cada interfaz gráfica.

- MessageSalvar y MessageOK: se ocupan de generar las ventanas emergentes donde se pregunta al usuario una confirmación después de pulsar Start o Save. La principal diferencia es que la clase MessageOK tiene un botón que confirma que se ha salvado correctamente un fichero, y la otra ventana tiene dos botones para elegir si o no.
De ventanas MessageOK solo pueden haber una, mientras que el número de ventanas MessageSalvar depende del número de ficheros que tengan que actualizarse, el máximo es 3.

- Render: es la interfaz en Java que implementa el comportamiento de la tabla usada en el módulo VLIW. Ésta clase es la heredera de la clase bin ya que es donde esta definida.

Todas las clases tienen instanciado algún objeto de una clase predefinida en Java. Así hace un uso particular de éste objeto en su código. Los objetos más usados más usados son los de las clases pertenecientes al paquete java.lang, java.swing, java.io, java .awt y java.util.

5.3 Creación de un *LiveCD* usando *LiveCD*-tools en el sistema operativo Fedora Core 6 [14]

Aunque en conceptos básicos ya se hace una descripción general de como funciona el programa *LiveCD-tools*. En este apartado se comenta con más detalle y se describen algunos de los comandos necesarios para crear el mismo *LiveCD* del proyecto.
Se empieza bajando e instalando el programa [34.a]

- Es muy aconsejable tener a mano el apéndice B. No se han incorporado figuras aquí porque la cantidad de texto es bastante grande.

5.3.1 Ficheros de configuración

Se descargan los RPM [34.b] y se extraen los ficheros de configuración (.conf).
La lista del fedora-LiveCD-base.conf contiene los paquetes básicos del S.O. para poder cargar y ejecutar Fedora en un CD, pero no dispone de entorno de escritorio, solo se ve un terminal. En los *scripts* de este archivo hay comandos ya escritos para configurar el sistema operativo. Se cambia:

La zona de NewYork/America por ZONE="Europe/Madrid" para configurar el reloj. El lenguaje por defecto, ingles por el español LANG="es_ES.UTF-8".
De momento es suficiente. Se guarda y se carga el archivo fedora-LiveCD-gnome.conf. Este fichero incorpora los paquetes que forman el escritorio GNOME.

En el proyecto se quitaron algunos paquetes como el editor de textos emacs (con gedit ya basta para ver/editar textos y escribir código C ya que gedit reconoce su sintaxis y facilita la programación). También se quitaron los paquetes ntfs, ntfsprogs, gnome-bluetooth. Se incorporaron el titulo de los RPM del compilador C (gcc) y la versión de Java usada en la creación del interfaz gráfica (jdk-6u1-linux-i586).
El problema vino al crearse el *LiveCD*, mientras se crea sale por el terminal una lista de los paquetes que usa, y Java no aparecía.

- Seguramente porque no consta en los repositorios Core o Extras que Fedora tiene en Internet (tiene los mismos paquetes que vienen en los 6 CDS de su versión completa). Así que la incorporación de Java con estos ficheros no era posible. Existe el paquete de Java java-gcj (*GNU Compiler Java*) que viene al instalar Fedora, pero daría problemas ya que no es la misma versión usada al implementarse la interfaz gráfica. Es una versión de Java que suele dar problemas.
  Se debe incluir en un paquete RPM que LiveCD-tools si acepte. Más adelante se explica.

En los *scripts* del archivo fedora-LiveCD-gnome.conf hay un momento en que se crea el fichero */etc/init.*dLiveCD. Este archivo se inicializa cada vez que el sistema arranca, es donde se añade la linea *system-config-keyboard es* para que el teclado cargue el mapa de caracteres español. En el mismo fichero se elimina la parte que añade un usuario sin contraseña para cuando aparezca la pantalla de *login*, con tener el usuario súper usuario (*root*) ya basta.

5.3.2 Creación de un paquete RPM [21] [22] [23]

Con el comando rpmbuild se consigue. Para ello es obligatorio que los ficheros.conf vayan a la carpeta */usr/src/redhat/SOURCES* y el archivo SPEC al directorio

/usr/src/redhat/SPECS. El comando el directorio */usr/src/redhat/BUILD* para dejar archivos temporalmente.
Para obtener un fichero SPEC de ejemplo bajarse [36.c]. El usado en el proyecto esta en el apéndice B.

En el archivo SPEC se declaran los archivos.conf a usar y los ficheros extra que queremos añadir al CD, como por ejemplo, todo el contenido de la carpeta donde esta JavaVEX. Lo que queremos añadir se declara en la etiqueta *Source*(fichero fuente). Si el nombre no contiene una extensión conocida como jpg, pdf, conf etc… da problemas al crearse el *LiveCD* ya que parece que *LiveCD-tools* intenta interpretar el código para buscar las dependencias necesarias para su correcta instalación (como por ejemplo gcc requiere el uso del paquete gclib.rpm).
La única forma de evitar este problema es comprimiéndolos, por ejemplo a tar.gz. Despues de comprimirse se añaden al directorio SOURCES).

Es importante que en el fichero SPEC se defina el BuildRoot, que contiene la ruta donde se instalarán virtualmente y temporalmente los paquetes. Instalación que se lleva a cabo antes de ser empaquetados como RPM. Sino se definiera la ruta pueden haber problemas porque serían instalados sobre el sistema operativo.

Como en el proyecto se van a usar dos ficheros.conf, se deben crear dos paquetes. Para crear un nuevo paquete se incorpora %package seguido de su nombre en el fichero SPEC. Además se deben rellenar los campos summary, group, autores y escribir si este paquete requiere de otro RPM. Aquí es donde se pone el nombre del RPM que se genera del primer fichero fedora-LiveCD-base.conf.

Cabe recordar que una vez instalado el software mínimo para ejecutar FC6 (lista que contiene fedora-LiveCD-base.conf), se debe instalar el segundo paquete con el resto de aplicaciones.

Las secciones %prep, %build, %install, %clean y %files se rellenan con comandos que se usaran en diferentes etapas del RPM. A continuación se decriben.

- %prep extrae los códigos fuentes (*sources*) a la carpeta BUILD. Si luego se añaden aquí las secciones %setup y %path sirven para desempaquetar el código fuente y aplicar un parche en caso que lo necesitara el código fuente.

- %build puede incluir los comandos necesarios para compilar y construir los paquetes.
- %install es donde se especifica los comandos que debe realizar el S.O. cuando se instalen los RPM. Tanto virtualmente como si se instalasen sobre al sistema operativo.
- %clean es la sección que permite borrar el contenido de los ficheros temporales del directorio BUILD.
- %files, aquí se escribe la ruta donde irán los códigos fuente dentro de cada RPM creado y también otorgan los privilegios.

En el proyecto los códigos fuentes ya tienen todo lo necesario para poder empaquetarse en el RPM. Sin necesidad de tener que desempaquetarse antes y aplicar un parche para luego rehacerse (%prep y %build). Así que basta poner la ruta final dentro del RPM (%files) y los comandos para instalarse al S.O. (%install).

Como códigos fuente (*sources*), finalmente están:

- La carpeta que contiene todos los archivos de JavaVEX y el ejecutable VEX que se pueden ver en el escritorio del *LiveCD* que acompaña esta memoria y permite cargar la aplicación. Todo compilado como tar.gz.
- Los dos ficheros.conf, el paquete de Java y la imagen de fondo de pantalla (*wallpaper*).

5.3.3 Ultimas modificaciones

Imaginemos que ahora se están instalando todos los paquetes RPM. En los dos RPM creados el sistema operativo mira la sección %install y debe copiar los ficheros fuente de los dos RPM en la carpeta */etc/LiveCD/* del *LiveCD*.
Se han instalado todos los paquetes. De momento están en un espacio de memoria temporal. Contiene el S.O. completo que luego se incluirá comprimido en el CD.

Faltan hacer los *scripts* en el fedora-LiveCD-base.conf para:

- Descomprimir los ficheros comprimidos tar.gz.
- Mover los ficheros al escritorio y instalar Java.

Porque se instala el java en los *scripts* de los ficheros.conf y no en la sección %install?

- El hecho de que la instalación del java-6u1-linux-i586.rpm se haga una vez estén instalados los paquetes y no se haga en el archivo SPEC es porque el comando que instala un RPM es *rpm –ivh nombre.rpm*, en ningún momento se puede poner el directorio de instalación virtual BuildRoot ($RPM_BUILD_ROOT) en el comando. Si se pusiera se instalaría en el mismo ordenador donde se éste creando el *LiveCD*, y no en el S.O. que se incluirá en el CD. Los comandos que se suelen usar en %install y que sí requieren saber el directorio, son por ejemplo el mv (*move*) o cp (*copy*).

Ahora ya se pueden crear los RPM desde la carpeta */usr/src/redhat/SPECS*. Al finalizar los paquetes creados se encuentran en */usr/src/redhat/RPM/i386/*.

5.3.4 Creación del *LiveCD*

Con los dos ficheros.conf usados hasta ahora, se han creado los paquetes fedora-LiveCD.rpm y fedora-LiveCD-gnome.rpm. Ambos se copian al que será el primer repositorio, repoA (*/root/base_packages/)*.Luego se usa la herramienta createrepo, esta herramienta genera los ficheros necesarios para que se comporte el directorio como un repositorio (se crea un fichero XML que contiene la lista de todos los RPM del repositorio).
Se insertan los 6 CDs del Fedora Core 6 y se copian todos los RPM de la carpeta */Fedora/RPM/* a la carpeta */var/www/html/repo/core/*.Éste directorio es el segundo repositorio y se usa otra vez la herramienta createrepo.

Finalmente, con un comando de texto del *LiveCD-tools* que permite crear el *LiveCD*, se pasa como argumento:

- Los directorios de los dos repositorios.
- El paquete fedora-LiveCD-gnome que se instala después del paquete fedora-LiveCD.
- El nombre que tendrá la imagen ISO y que es el mismo nombre que sale al ejecutar el *LiveCD* al ordenador.

La imagen que se crea esta en el mismo directorio donde se uso el *LiveCD*-creator.

5.3.5 Contenido del *LiveCD* del proyecto.

El *LiveCD* entregado junto la memoria incorpora:

- La aplicación JavaVEX y su ejecutable.
- El compilador VEX vesion 3.41.
- El paquete Java Development Kit 1.6.0 instalado junto a su intérprete correspondiente.
- El compilador gcc (GNU Compiler C) con sus librerías y paquetes. Es necesario porque igual que JavaVEX es el *front-end* de VEX, este lo es a su vez del gcc.
- El editor de textos gedit.
- Incorpora todos los módulos necesarios para que el sistema operativo Fedora Core 6 de 32 bits pueda ser cargado en cualquier ordenador sin tener que ser instalado en el disco duro.

6. Problemas y alternativas surgidas en las diferentes etapas del proyecto

6.1 Nacimiento de la idea de proyecto

Descripción expuesta en la Gestión de Proyectos del SIEE:

> *"Cal desenvolupar un senzill interfaz gràfic per fer servir un programa que simula una arquitectura VLIW. Bàsicament cal poder editar un text, cridar al compilador, cridar a l'execució de la simulació, i visualitzar els resultats. Finalment cal montar un live-CD amb un petit tutorial que ensenyi amb un exemple com fer servir el programari."*

6.1.1 Primera idea de proyecto

Realizar la interfaz en Java y implementar en su totalidad un simulador VLIW. Como la aplicación debe ser una herramienta docente para el alumnado de Arquitecturas de Computadores2 y sus profesores, la aplicación se centraría en mostrar gráficas similares a los ejercicios realizados en clase. Es decir, mostrar en cada uno de los ciclos que instrucciones se realizan, y sus etapas *pipeline* en el hardware. De una forma similar al que hace el diagrama de ciclos del KScalar.

6.1.2 Proyecto finalmente realizado

Después de una reunión con J.C Moure y ver que la primera idea no era la correcta. Se concreta en el uso del simulador VEX, la implementación de la interfaz gráfica y buscar un método que permita interactuar con VEX y obtener los resultados. Se tiene que:

- Buscar información sobre el compilador VEX. Se usa el libro de la bibliografía [1].
- Se investiga cómo implementar en Java una función que permita lanzar comandos exteriores y que el sistema operativo las ejecute.

6.2 Estudio de oportunidad:

Después de la aceptación del proyecto por Juan Carlos Moure y con una idea de cómo realizar el proyecto se empiezan a concretar requisitos como:

- Que sea útil para la asignatura y de fácil manejo para el alumno

    - A partir de todos los datos que genera VEX se deben mostrar los más importantes para poder comparar diferentes resultados. Hay que seleccionar los resultados y los argumentos que más les influyen.

    - La organización de los elementos de la interfaz gráfica tiene que facilitar el uso del programa para todo usuario y que sea intuitivo. Hay que ordenar los elementos adecuadamente sin confundir al alumno y debe haber gran flexibilidad de maniobra en las diferentes acciones.

    - Generación de mensajes (error/salvar/simular)que informan si la acción puede ser realizada o no.

- Dicha aplicación debe poder ejecutarse en un *LiveCD*

    - Desconozco cómo crear un *LiveCD* de una distribución cualquiera de Linux. Hay búsqueda de información en foros y en wikipedias donde otros usuarios relatan su experiencia al realizar un *LiveCD*.

    - Decidir que distribución instalar y que sea fácil la creación de un *LiveCD*.

    - Uso de un programa que permita ejecutar el *LiveCD* sin tener que copiar la imagen ISO en un Compact Disc para probarlo cada vez que se genera una imagen.

- La implementación del proyecto debe realizarse lo antes posible. Debe haber un periodo previo de prueba antes de que el usuario final lo use.

- Con lo aprendido hasta ahora y con J.C. Moure. Se puede empezar a programar he ir introduciendo pequeñas modificaciones.

6.3 Estudio detallado

Se conocen los requisitos a cumplir. Como programador se empieza a profundizar en aspectos técnicos de la implementación a fin de poder realizarlo todo.

- Con la ayuda de un proyecto realizado en la asignatura de libre elección Programació Avançada, el documento API de Java y de los foros de Internet, se diseña la interfaz gráfica y se realiza un análisis funcional básico de lo que realizará.

- Se encuentra bastante información sobre *LiveCD's* bajo sistemas operativos Fedora Core y Ubuntu.

6.4 Determinación de la carga de volumen:

Disponiendo de un ordenador de doble núcleo de 64bits, se decide por usar Fedora Core 5 o 6 de 64bits. Una vez instalado el nuevo sistema operativo y visto el volumen de código del proyecto realizado en la asignatura de libre elección, se establece una primera aproximación de la carga de trabajo que se dividirá principalmente en:

- Configuración del nuevo sistema operativo y mejorar la experiencia como usuario modificando, probando el S.O. y instalando paquetes, herramientas para usarlas en el proyecto.

- Primeras pruebas con el compilador VEX y una primera versión de la interfaz gráfica muy sencilla.

- Programación de la interfaz gráfica.

- Creación del *LiveCD* y probarlo.

- Documentación del proyecto.

6.5 Análisis orgánico:

A partir del estudio del apartado anterior, se empieza a organizar cada uno de los puntos.

- Configuración de Linux:
Es difícil planear que puede suceder en Linux teniendo escasa experiencia en el S.O. y tenerlo configurado todo para el resto del proyecto.

-Generación de la primera interfaz gráfica partiendo de:

- Un diseño del esqueleto del programa con las funciones más importantes y su orden de ejecución.
- Un esquema de la organización de los elementos que contiene y como interactuan con el usuario.
- El estudio de alternativas en objetos Java para cuando se vaya a implementar las funciones, escoger cual es el mejor. Por ejemplo, elegir entre usar el objeto: ScrollPane o ScrollBar(ambos del paquete AWT) o JScrollPane o JScrollBar(del paquete Swing). Se hacen las primeras pruebas de cada uno a medida que se van añadiendo a JavaVEX.

Debido a los módems que distribuyo la compañía MENTA de cable por USB (ahora ONO) no se consigue conectar a Internet desde Linux (tanto Ubuntu como Fedora). Cada vez que aparecía un problema comportaba reiniciar el ordenador para acceder a Windows. Se intento configurar el S.O. para que pudiera tener acceso a Internet, aunque también comportaba reiniciar cada vez que se quería solucionar una duda al respecto. Así que las primeras pruebas de las interfaces gráficas se realizan de momento bajo Windows.

- Primeras pruebas con VEX:

Realizar pruebas con este compilador para familiarizarse con sus argumentos y sus resultados. Con la ayuda de J.C Moure y de la lectura del Apéndice A del libro [1], se establecen una serie de argumentos y pasos para obtener los archivos que contienen tanto la secuencia de instrucciones VLIW como los datos estadísticos.

El compilador solo puede funcionar bajo un sistema operativo de 32 bits, teniendo instalado en el ordenador uno de 64 bits puede solucionarse instalando librerías de 32 bits y incluyendo en las llamadas de los diferentes archivos que llaman al gcc la opción –m32.

La interfaz gráfica final será aquella que se aproxime más a la acordada con el profesor. Se deja un margen de tiempo a la posible modificación de JavaVEX una vez se pruebe.

Se creará un *LiveCD* de la interfaz gráfica, así el profesor podrá manejar el programa. Pero para esta primera versión de prueba ha tenido que pasar bastante tiempo. Esto dificulta a veces la comunicación cuando se hablaba sobre el diseño de la interfaz y su evolución, por tanto el periodo de integración y prueba se iban retrasando.

6.6 Programación y juego de pruebas

- **Interfaz gráfica:**

La implementación de las interfaces estaba resultando poco satisfactoria, poco atractiva y además costosa por el gran numero de coordenadas que se debían poner (aun sin tener implementadas la mayoría de las estructuras a usar). Se opto por usar algún programa de Java que permita realizar interfaces fácilmente y que sean más vistosas como NetBeans. Eclipse era otra opción, pero NB por su facilidad y gran disponibilidad de herramientas del paquete Swing de Java fue la elección.

La compilación y ejecución de cada prueba funcionan correctamente. Pero fuera del entorno creado por NetBeans no funciona bien. Hay errores del S.O. con el interprete de Java debido a las librerías que vienen por defecto en Linux del

GnuCompilerJava(GCJ). El sistema operativo tiene y identifica solo su propio interprete, en cambio, cuando se trabaja en NB ya encuentra el adecuado (JDK 1.6.0).

- Siendo superusuario se tiene que cambiar ya que la aplicación final debe poder ejecutarse desde un terminal sin necesidad de tener instalado NB o otro programa.

Aun y empezar con una interfaz sencilla, a medida que se iban incorporando y probando resultados con el simulador VEX, se iban incorporando varios elementos a la interfaz gráfica. Se complicaba su uso y no resultaba tan cómoda e intuitiva. Se modifico para ser más simple.

También se implementaron y añadieron diferentes funcionalidades para mostrar más información al usuario, pero o bien porque luego se encontró una forma estéticamente y funcionalmente mejor, o porque de tanta información resultaba incomodo para el usuario, no se usaron.

- Por ejemplo, antes de saber implementar las ventanas emergentes que aparecen sobrepuestas en la interfaz gráfica, la información de la ventana se mostraba por el editor de textos del módulo de edición.

Al final el flujo principal de la aplicación se puede desglosar en 3 puntos:

- Escribir el código C.
- Obtener los resultados de compilar el código C con el VEX y mostrarlos por pantalla.
- Obtener los resultados estadísticos de simular con el VEX y dibujarlos por pantalla.

Es cuando se elige diseñar los 3 módulos de que consta la interfaz gráfica.

De los dos últimos puntos ya se busco información para poder realizar llamadas al compilador mientras se ejecutaba Java. Se intenta creando un proceso que contiene el comando con sus argumentos y se espera la finalización del proceso. No funciono. Tampoco funciono tratando de escribir los resultados en un fichero.

- Se requiere el uso de *threads* ya que deben leerse las salidas estándares de un proceso y de la salida de error.

- Linux:

Debido a los problemas con Internet que se podrían solucionar con más tiempo, y los problemas que podrían ocasionar el hacer una librería compatible C de 32 bits bajo un SO de 64 bits al crear el *LiveCD*. Se opta por usar el VMWare [36] bajo Windows.
VMWare es un programa que crea una maquina virtual que permite cargar un SO y actúa como si estuviera instalado. Todos los archivos y configuraciones que se apliquen a dicho programa se mantendrán la próxima vez que se carga. Para cargar un SO se usa una imagen predefinida de 32 bits del Fedora Core 6. Así ya se tiene acceso a Internet y un compilador gcc de 32 bits nativo del sistema.
El uso del VMWare empeora en el rendimiento del S.O. cargado. Se verá especialmente afectado para la creación del *LiveCD* o al simular una nueva arquitectura VLIW con muchas unidades funcionales y/o que permita procesar un gran numero de operaciones por cada instrucción, ya que ocupan bastante memoria.

- ***LiveCD*:**

Con la aplicación bastante avanzada y a falta de concretar los últimos aspectos se empieza a probar el programa *LiveCD-tools* desarrollado por IBM.

Surgen varios problemas con su uso:

- Cada prueba para generar una imagen ISO tarda aproximadamente 2 horas y media. Si el resultado final no es satisfactorio debe volverse a consumirse ese tiempo y realizar además un estudio previo a porque fallo más el tiempo de configuración de los ficheros que usa el programa.
    - Se usa VMWare que permite cargar los CDs de la misma forma que puede cargarse un S.O.

- Excesivo uso de memoria del disco duro. Una imagen final de aproximadamente 550mb puede llegar a ocupar 3GB, ya que debe crear un espacio de trabajo con los paquetes especificados en los ficheros.conf y con los paquetes que dependen. Luego desempaquetarlos todos, instalarlos y por ultimo etiquetar y crear la imagen ISO.

    - El *LiveCD-tools* comose ha explicado se basa en el uso de repositorios de paquetes RPM. En un principio se volcaron todas las imágenes del sistema operativo completo de Fedora Core 6, ocupando varios GB. Pero la maquina virtual por defecto del VMWare viene con 4,4GB disponibles. La solución fue ir quitando los paquetes que no eran necesarios para conseguir tener el espacio suficiente en crear una imagen.

- Uno de los problemas fue la incorporación del Java en el *LiveCD*. No se conseguía instalar su paquete RPM incorporándolo al repositorio de paquetes. Finalmente se conseguía añadiéndolo a los RPM que contienen los archivos de configuración.

- La incorporación de todos los ficheros que conforman el JavaVEX se baso en la misma idea que para incorporar Java, pero el LiveCD-tools parecía que interpretaba los ficheros he intentaba buscar sus dependencias, ello era fatal porque el JavaVEX tiene mucho código en Java, C, y librerías de C.

    - Se comprimían como archivos tar.gz y funcionaba al ser un fichero comprimido como el paquete RPM de Java.

- Paradójicamente, el uso del VMWare originó uno de los mayores problemas del proyecto, y que más hizo perder el tiempo. Se usaba porque se evitaba quemar las imágenes ISO en CD y no se necesitaba asi reiniciar el ordenador.
    - El problema vino cuando se cargaban las imágenes en el VMWare. La resolución siempre es de 800 por 600, es invariable. Se intentó establecer una resolución mayor al cargar el SO en el ordenador y

suficiente para poder ejecutar la aplicación. De esta forma no se obligaba al usuario a configurar manualmente la resolución de pantalla.
- Después de hacer bastantes CD para poder enseñar la evolución del proyecto en las reuniones con J.C Moure se soluciono, pero hasta entonces siempre había usado el VMWare.

- Ya se ha dicho que modificar o añadir los *scripts* se hacia con un poco de información. Todo y ésto algunos comandos requirieron varias pruebas. Los resultados no se ven hasta que no tienes la imagen hecha, no es como una terminal donde se ve el resultado al instante por si hay errores. Un más el tiempo de configuración de los ficheros que usa el programa. Por ejemplo:

  - Cuando se crean los directorios y se mueven los ficheros al escritorio del sistema de archivos del *LiveCD*, salían errores de directorios no válidos. La creación de las carpetas con mkdir no funcionaba porque se debía añadir la opción mkdir –p.

  - También cuando se añadía el comando para cargar el teclado español entre los *scripts* (una vez instalados todos los paquetes), se acababa manteniendo el teclado como inglés. Para solucionarlo se tuvo que poner el comando en el fichero init.d que es un archivo que se ejecuta después de la carga del FC6 en la memoría.

7 Conclusiones

A continuación se comenta como se han cumplido los objetivos mencionados en la introducción:

- Se ha estudiado sobre la arquitectura VLIW debido al uso constante del compilador VEX. Se han probado algunos de los comandos y argumentos opcionales del VEX para observar su funcionamiento y su aplicación a programas de alto nivel escritos en C.

- JavaVEX es un programa sencillo y que refleja rápidamente como se comporta un procesador VLIW. Al usar la interfaz gráfica no tiene porque conocerse la existencia del VEX, se ha conseguido reducir tiempo que el usuario emplearía en realizar la misma tarea con VEX, también se reduce el tiempo gracias a la gestión de los ficheros que permite JavaVEX, lo cual evita usar constantemente un editor de textos. Al incluir una breve explicación de las operaciones se evita tener que buscar información disponible en el libro [1].

- Se ha diseñado una interfaz que permite configurar muchos parámetros de la arquitectura con pocas acciones, la interfaz no es complicada y el nivel de información recibida a cambio es alta, es pues, una herramienta potente y productiva.

- Se ha conseguido que sea la maquina la que se encargue de realizar muchas tareas sin que el usuario tenga que intervenir, en gran parte, gracias a la implementación que interactúa el Java con el compilador VEX.

- Se ha creado un *LiveCD* con Fedora Core 6 de 32 bits, aunque el número de ordenadores a los que se ha probado es poco, debe poder ejecutarse en la mayoría de ordenadores de sobremesa o portátiles. Al *LiveCD* se han incluido aplicaciones externas a Fedora (JavaVEX y el compilador VEX) además se ha elegido los paquetes que conforman el S.O. incluido en el CD y la configuración del lenguaje y del teclado entre otros aspectos.

- Se ha escrito un manual de usuario para que quien use JavaVEX pueda dominar la aplicación rápidamente. Incluye ejemplos cada vez más óptimos, diseñados para aprender a interpretar mejor los resultados de la interfaz gráfica. Con los ejemplos se pretende enseñar al lector a ver que mejoras hay entre distintos programas escritos en C y que hardware es necesario para realizar aquella función.

El resultado de cumplir los objetivos inicialmente propuestos, supone una gran ventaja para la finalidad docente que se ha hecho JavaVEX, ayudar en las prácticas de AC2.
Al alumno de AC2, con los ejercicios de clase, se le enseña a buscar cual es la mejor organización de las operaciones en una arquitectura VLIW sin que haya problemas de dependencias, las mejoras que derivan de aplicar el *loop unrolling*, se le hace calcular cual es el numero mínimo de unidades funcionales que se requieren para que funcione el algoritmo del enunciado, la cantidad de ciclos que se ejecutaran, la mejora que hay ante diferentes configuraciones (calculo del *speedup)*, el camino crítico, etcétera.
Con JavaVEX no solo se pone en práctica todos estos aspectos, sino que aporta datos sobre el comportamiento de la *cache*, añade la posibilidad de ver el comportamiento de un lenguaje de alto nivel muy usado en la carrera con la arquitectura VLIW, todo ello con muy poco tiempo de ejecución del JavaVEX.
El único problema que se le presenta al alumno es el tener que aprender a modificar el hardware a simular pero para eso hay un manual de usuario, porque en lo que refiere al aspecto de la interfaz gráfica es muy sencilla, no hay muchos objetos y ninguno es difícil de interpretar o muy poco usado por otras interfaces. De la información que se muestra, toda es enseñada en la asignatura exceptuando algunas de las operaciones y que en la interfaz gráfica se ayudan de con una breve explicación.
Si además JavaVEX puede ejecutar desde cualquier ordenador, lo convierte en una herramienta versátil, potente, sencilla e ideal para muchos niveles de usuario. Desde el usuario principiante que compara diferentes modelos de arquitectura de procesador basándose en los resultados de la simulación, hasta el usuario experto que puede estudiar la parte critica de un programa escrito en C, probar con los diferentes grados de optimización o de *loop unrolling*, y si quiere, también puede usar algunas de las funciones disponibles en VEX y que éste interpreta en tiempo de compilación, como por ejemplo, las funciones que sirven para controlar el algoritmo heurístico que realiza el *loop unrolling* en VEX.

7.1 Experiencia personal del proyecto:

Para cumplir con los objetivos, como desarrollador del proyecto, ha sido una tarea laboriosa y con mucha investigación detrás. Al finalizar el proyecto puedo decir que:

- He aprendido a manejar, configurar y familiarizarme en un SO que no sea alguna de las versiones de Windows. Aun y tener un conocimiento muy básico sobre Linux no ha servido de mucho, para la configuración del hardware o la instalación de Java, entre otros problemas, se necesito buscar información. El no poder usar Internet desde FC6 obligo a usar programas como el VMWare.

- Observando el compilador VEX, es una aplicación que puede servirme en un futuro si algún día decido realizar una herramienta docente. Con los problemas que comporta no disponer de un entorno visual al usuario. Un problema de VEX es tener que usar ficheros para almacenar la información de la distintas fases de la simulación, también es un problema encontrar/recordar aquel argumento que permite hacer una función en concreto. Creo conveniente que toda herramienta docente debe tener una interfaz gráfica, siempre que sea beneficioso para el programador ya que cuesta bastante tiempo hacerla.

- Del VEX me quedo con detalles que no se me habrían ocurrido incluir en una herramienta docente, como por ejemplo: una carpeta donde poner ejemplos para ayudar al usuario o el disponer de un foro en Internet en caso de dudas.

- Ha mejorado mi conocimiento sobre el comportamiento de la arquitectura VLIW. Como estudiante era muy difícil poder organizar un ejercicio que partía de un pequeño algoritmo escrito en lenguaje ensamblador. Ahora es VEX quien organiza el algoritmo y los resultados tienen un nivel de optimización superior.

- Al diseñar la interfaz me ha permitido ver la importancia de realizar un buen diseño, de estructurar bien todos los componentes que la formarán y de la comunicación entre ellos. Aunque la primera versión que hice empezó bien,

cuando se fueron añadiendo más funcionalidades y opciones se fue complicando.

- Como programador de Java, me ha permitido descubrir el NetBeans, mejorando y bastante respecto la interfaz que en un inicio empecé a programar desde cero. Respecto a las clases y funciones que forman Java, he adquirido un mayor conocimiento en elegir aquellos elementos gráficos que me ofrece éste lenguaje y que mejor se adaptaban al diseño de la interfaz gráfica. También he mejorando la experiencia con hacer *threads* y interfaces de clases en Java.

- Poder hacer un *LiveCD* de una distribución Linux en concreto, pero ahora tengo mucha más experiencia y entiendo mejor el proceso de creación en el resto de distribuciones.

Desde febrero cuando empecé el proyecto han sido muchas las horas dedicadas para poder presentar el *LiveCD* con JavaVEX. La poca experiencia en manejar Linux ha pasado factura por la multitud de problemas que me han ido surgiendo. Ahora me es más cómodo usar Linux gracias a la solución de la mayoría de problemas que han ido saliendo.
Cuando me cogí el proyecto, tenía claro que seria implementado en Java, ahora que he acabado he aprendido un poco más. La experiencia adquirida en este lenguaje espero que se vea recompensada en un futuro.
Para crear el *LiveCD* es la parte que más investigación y dolores de cabeza ha comportado, nunca había hecho un *LiveCD* y era difícil entender el porque de muchos procesos descritos por otras personas, dependía de la distribución Linux, de usar herramientas muy específicas que variaban entre proceso y proceso y de la experiencia del lector. Después de realizar una treintena de *LiveCD* creo que puedo incluir cualquier aplicación y configuración de Fedora Core 6 en un CD.
Ya me han dicho que realizar una memoria no es fácil, y ahora que he hecho ésta lo confirmo. Lo más difícil al escribirla ha sido, el capitulo de conceptos básicos que me izo investigar más acerca el funcionamiento de los programas (algunos aspectos los desconocía mientras estaba usando los programas), también costo organizar la memoria, y sobretodo, hacer una buena redacción para que sea lo más clara posible.

El tiempo empleado en hacer el proyecto ha superado en creces al estimado, en un principio creía que la mayor parte del tiempo la dedicaría a la implementación en Java, y más o menos le he dedicado lo estimado en programarlo, pero no contaba con todos los problemas que han ido saliendo de Linux y que hacer un *LiveCD* fuera tan complicado.

Para hacer el proyecto se pedía un nivel básico de Linux, de arquitecturas de computadores y un nivel medio en lenguajes visuales (Java o C++), se han usado los conocimientos adquiridos durante la carrera en las asignaturas de sistemas operativos, arquitectura por computador, ingeniería del software, programación avanzada y otras asignaturas que usaban Java en las prácticas. Ahora que estoy satisfecho con el resultado y que se han ampliado mis conocimientos, no me queda más que hacer una buena presentación del proyecto y esperar que tanto el profesor de AC2 como sus alumnos se beneficien de JavaVEX.

8. Bibliografía

Simulador VEX y VLIW

[1] Joseph A.Fisher, Paolo Faraboschi,Cliff Young. 2005. *Embedded Computing: A VLIW Approach to Architecture, Compilers, and Tools.* Morgan Kaufmann Publishers Inc, San Francisco, p. 539-605.

[2] Ankur Chowdhry, Nitesh Batra, Puneet Jain, Tina Checker.Agosto 2001. *Nitesh Batra Project: Architecture of VLIW* [en línea]. Workstations At Maryland, University of Maryland. http://www.wam.umd.edu/~nbatra/411proj/vliwopening.htm

[3] Forum *VEX compiler and tools*. http://www.vliw.org/vex/

[4] Disck Pountain. Abril 1996. *The Word on VLIW : Intel and HP hope to speed CPUs with VLIW technology that's riskier than RISC.*[en línea]. Byte.com. http://www.byte.com/art/9604/sec8/art3.htm

[5] J. H. Moreno, M. Moudgill, K. Ebcioglu, E. Altman, C. B. Hall, R. Miranda, S.-K. Chen. Marzo 1998. *Simulation/evaluation environment for a VLIW processor architecture* en *IBM Technical Journals*[en línea]. IBM Corporation. http://www.research.ibm.com/journal/rd/413/moreno.html

[6] Michael Gschwind. Febrero 2001. *VLIW at IBM Research* [en línea].IBM Corporation. http://www.research.ibm.com/vliw/simulator.html

[7] Laurent Fournier, Shai Rubin. Agosto 2006.*Dynamic test program generator for VLIW simulation* [en línea]. Patent Store, Washington. http://www.patentstorm.us/patents/ 7085964-description.html

Capitulo 2.Estado del arte.

[8]J. C. Moure, Dolores I. Rexachs, Emilio Luque 2002. *The KScalar Simulator*. ACM Journal of Educational Reources in Computing(JERIC), Vol. 2, No. 1, p. 73-116. Disponible en Web: http://citidel.villanova.edu/bitstream/10117/114/8/ jerikMoureCover.pdf.

[9] Ray Ontko. Abril 1999. *Misc-1 Homepage* [en línea]. http://www.ontko.com/mic1/

Java

[10] David Arnow, Gerald Weiss. 2000. *Introducción a la programación con JAVA: Un enfoque orientado a objetos.* (trad. Diego Sevilla Ruiz, José Rafael García Bermejo, Paloma Centenero, Mª Luisa Díez Platas). 1ª ed. PEARSON EDUCACIÓN, S.A., Madrid.

[11] Chuidiang Roxas. *Ejemplos y tutoriales Java* [en línea]. 4 feb 2007. http://www.chuidiang.com/java/

Linux

[12] Christian González. *Mini-guía de Fedora Core 6* [en línea]. Fedora-es. http://www.fedora-es.com/node/373

[13] Varios autores 2002-2004. *Manual Linux eminentemente práctico, ZonaSiete.ORG* [en línea]. ZonaSiete.ORG Editores Team. http://www.zonasiete.org/manual/index.html

LiveCD

[14] Mayank Sharma. *Build a Fedora Live CD* [en línea]. 28 Feb 2007. IBM Corporation. http://www-128.ibm.com/developerworks/linux/library/l-fedora-livecd/index.html

[15] Anonimo.2006. *Reconstructor- Ubuntu Creador* [en línea]. Joomla Open Source Matters. http://reconstructor.aperantis.com/index.php?option=com_frontpage&Itemid=1

[16] Tomas M. 2007.*Linux Live for CD & USB* [en linea]. http://www.linux-live.org/

[17] Varios Autores. *Live CD Creation Resources* [en línea]. Ultima modificación 2006. LiveCDlistwiki.http://www.livecdlist.com/wiki/index.php/LiveCD_Creation_Resources

[18] Varios autores. *LiveCDCustomization* [en línea]. Ultima modificación Septiembre 2007. Comunidad Ubuntu. https://help.ubuntu.com/community/LiveCDCustomization

[19] Luc Parson. Marzo 2007. *Creating your own Linux Live CD.*[en línea, e-book] http://www.linux-live.org/create%20a%20livecd.pdf

[20] Madan Puruskar Pustakalya. Mayo 2005. *Research report on creating a bootable Live-CD* en PAN Localization Project [en línea, pdf] http://www.nepalinux.org/docs/researchreponlivecd.pdf

Creacion de paquetes RPM

[21] Dan Poirier .Noviembre 2001. *Usando RPM en Red Hat Linux 7.1* [en línea] Traducido por Carlos Eduardo H. PlazaLinux. http://www.plazalinux.com/modules.php?name = News&file=print&sid=546

[22] Dennis Stephen Cohn Muroy. Abril 2007. *Creación de Paquetes RPM y deb* [en línea]. Joomla! Open Source Matters. http://tuxpuc.pucp.edu.pe/content/view/624/1/

[23] Joel Barrios Dueñas. 5 febrero 2007. *Como crear paquetería con rpmbuild* [en línea]. Linux Para Todos / Factor Evolucion SA de CV. http://www.linuxparatodos.net/portal/staticpages/index.php?page=como-rpmbuild

Objetivos, requisitos y técnicas en las interfaces gráficas

[24] Eduardo Mercovich. Julio 2002. *Ponencia sobre Diseño de Interfaces y Usabilidad* [en línea]. GaiaSur. http://www.gaiasur.com.ar/infoteca/siggraph99/diseno-de-interfaces-y-usabilidad.html

[25] Dr. Pere Marquès (UAB). 2000. *Diseño y evaluación de programas educativos.* [en línea] .XTEC. http://www.xtec.es/~pmarques/edusoft.htm

[26] Benjamin Roe. Diciembre 2004. *Diseño de Interfaces de Usuario Usables: Una Guía Rápida para Desarrolladores de Software Libre y de Código Abierto* [en línea]. MundoGeek. http://mundogeek.net/traducciones/interfaces-usuario-usables/gui.html

[27] Leopoldo Sebastián M. Gómez. *Diseño de Interfaces de Usuario Principios, Prototipos y Heurísticas para Evaluación* [en línea] Monografías.com S.A. http://www.monografias.com/trabajos11/heuri/heuri.shtml#pri

[28] Noe Sierra Romero, Sergio V. Chapa Vergara. Marzo 2001. *Diseño de interfaces* [en línea] Departamento de Computación CINVESTAV, México. http://www.cs.cinvestav.mx/CursoVis/prinvisual.html

Problemas y alternativas surgidas en las diferentes etapas del proyecto

[29] Anonimo. Septiembre 2005. *Etapas de un proyecto – Fases, estructura, etapas…* [en línea]. GETEC. http://www.getec.etsit.upm.es/docencia/gproyectos/planificacion/etapas.htm

Capitulo 3. Conceptos básicos

[30] Varios autores. 2006-2007. [en línea]. Wikipedia, Wikimedia Foundation.
Superescalar http://es.wikipedia.org/wiki/Superescalar .
VLIW http://en.wikipedia.org/wiki/Very_long_instruction_word ,
http://es.wikipedia.org/wiki/VLIW .
LiveCD http://es.wikipedia.org/wiki/LiveCD , http://en.wikipedia.org/wiki/LiveCD .

Diagrama de classes

[31] Anonimo. *Tutorial de UML.* http://www.dcc.uchile.cl/~psalinas/uml/modelo.html [Offline, disponible en *cache* de Google guardado el 9 de Julio del 2007] http://64.233.183.104/search?q=cache:gDMVqdA8nxwJ:www.dcc.uchile.cl/~psalinas/uml/modelo.html+http://www.dcc.uchile.cl/~psalinas/uml/modelo.html&hl=es&ct=clnk&cd=1&gl=es&client=firefox-a.

[32] Varios autores. *IBM developerWorks: Rational software developer resources.*[en línea] Ultima actualización Sep 2007. IBM Corporation. http://www.ibm.com/developerworks /rational/

[33] Charlie Calvert. 2002. *Interfaces and Loose Coupling in Java by Charlie Calvert* [en línea]. Borland Software Corporation. http://dn.codegear.com/article/30372

Para dudas sobre Arquitecturas de Computadores o la creación del diagrama UML se usaron los apuntes, pdfs y documentos disponibles en las asignaturas de Arquitectura de Computadores 2 e Ingeniería del Software 1 y 2.

Para intentar solucionar los diversos problemas que dieron las distintas distribuciones Linux, o dudas, consultas sobre el SO, se visitaron entre otras páginas/foros:
www.linuxparatodos.com , http://www.linux-es.org/ , http://www.todo-linux.com/
Java: www.thinkdigit.com , http://www.javaworld.com/

8.1 Programas usados en el proyecto

[34.a] David Zeuthen's, diciembre 2006, *LiveCD-tools*. Pilgrim [en línea] http://people.redhat.com/davidz/livecd/i386/livecd-tools-001-1.i386.rpm,
[34.b] ejemplos de paquetes RPM [en línea]
http://people.redhat.com/davidz/livecd/i386/fedora-livecd-6-1.i386.rpm ,
http://people.redhat.com/davidz/livecd/i386/fedora-livecd-gnome-6-1.i386.rpm ,
[34.c] ejemplo fichero SPEC [en línea]
http://people.redhat.com/davidz/livecd/spec/fedora-livecd.spec

[35] TQMcube. Createrepo[en línea] http://tqmcube.com/files/createrepo-0.4.3-5.1.noarch.rpm

[36] Sun Microsystems, NetBeans IDE 5.5 [en línea]
http://dlc.sun.com/netbeans/download/5_5/mlfcs/200612070100/netbeans-5_5-linux-es.bin

[37] WMWare Inc. WMWare Player. Versión 1.0.2 [en línea]
http://download3.vmware.com/software/vmplayer/VMware-player-1.0.2-29634.i386.rpm

[38] Joseph A.Fisher, Paolo Faraboschi,Cliff Young. VEX Toolchain. Version 3.41 [en línea] http://www.hpl.hp.com/downloads/vex/vex-3.41.i586.tgz

Apéndice A. Manual de usuario para JavaVEX

1 Introducción

JavaVEX es una herramienta orientada a familiarizarse con las arquitecturas *Very Long Instruction Word* (VLIW). Sirve para introducir los conceptos de paralelismo a nivel de instrucción, ya que las arquitecturas VLIW explotan eficientemente los recursos del procesador.

JavaVEX interactúa con VEX, que es un software ya existente, solo funciona bajo Linux y con comandos de texto. JavaVEX permite escribir y editar código C, compilarlo al repertorio VLIW, visualizar el código VLIW y mostrar los resultados de la simulación con gráficos.

Las siguientes páginas son una guía para poder manejar esta aplicación. Con la ayuda de ejemplos se describen todas las partes que componen la interfaz gráfica y, se muestra su utilidad.

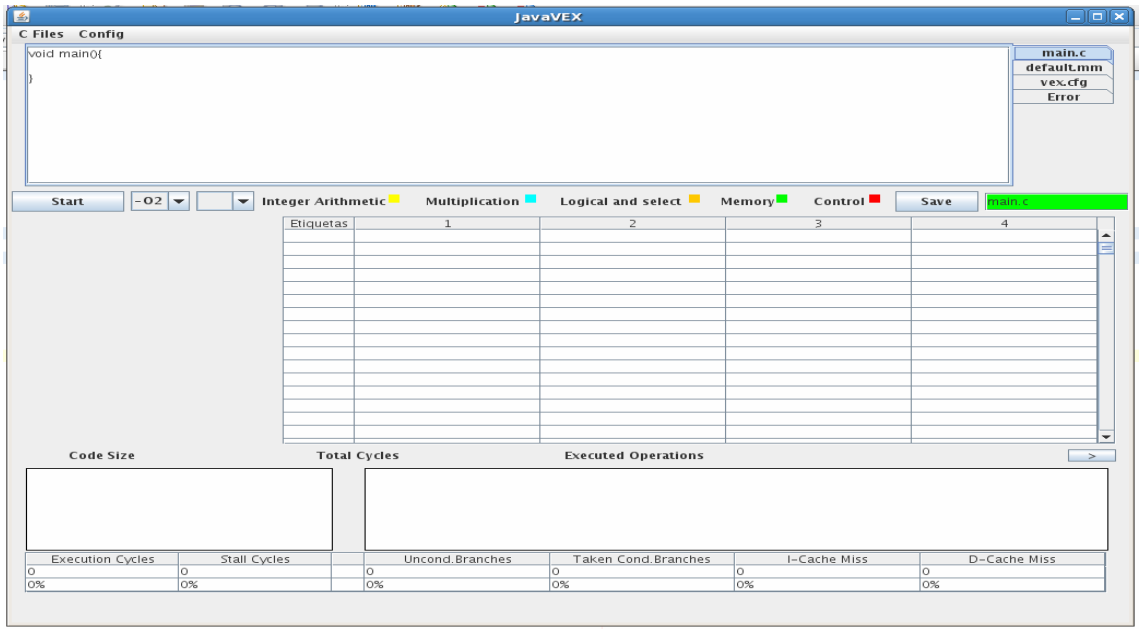
2 Requisitos del sistema

- Tarjeta gráfica con resolución mínima de 1152x752 *pixels* y 256 colores
- 256MB de memoria RAM
- Lector de CD

3 Ejecución de JavaVEX

Los pasos necesarios para cargar la interfaz gráfica son:

1) Encender el ordenador e insertar el CD en el lector de CD.
2) No pulsar nada, dejar que cargue Fedora Core 6 en el ordenador.
3) Cuando aparezca la pantalla de login escribir root y pulsar la tecla enter,
4) Cuando cargue el escritorio pulsar dos veces en el ejecutable JavaVEX.
5) Aparece la ventana de JavaVEX (ver figura a.1).



4 Primeros pasos

4.1 Implementar el código

Como puede verse en la anterior figura a.1, en medio de la parte superior hay una área de texto que sirve para escribir el código C. En la figura a.2 se muestra un ejemplo. Consiste en buscar un número dentro de un vector de elementos. Se supone que el vector está ordenado de menor a mayor

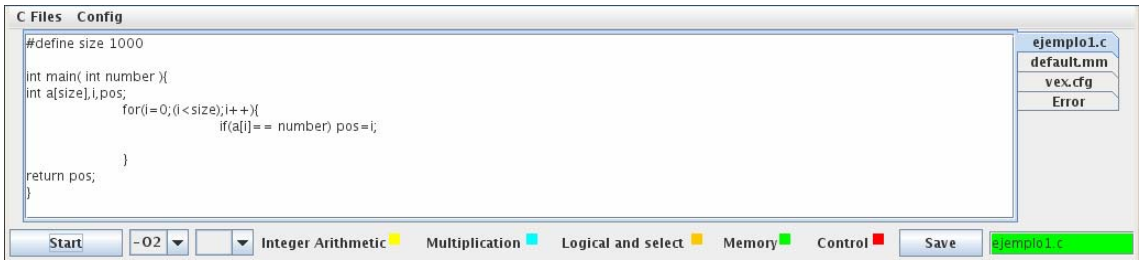


Figura a.2 Código ejemplo1.c

4.2 Salvar un archivo

Cuando se desea salvar el código con el programa:
Se escribe el nombre del archivo que se quiere salvar en el cuadro de texto (ejemplo1.c) y se pulsa el botón Save ubicado debajo a la izquierda del área de texto.

- El nombre escrito debe cumplir el formato *nombre_del_archivo*.c. Si no lo cumple aparece de fondo el color rojo y no deja salvar. Si lo cumple el color de fondo es verde.

Tras apretar el botón Save aparece una ventana pidiendo la confirmación (similar al de la figura a.3). Esto permite evitar perder el programa en caso de pulsarse el botón involuntariamente.

- Para usuarios avanzados: Cuando se salva un fichero, este se encuentra en el directorio */Desktop/JavaVEX/bin/fitxers/*. De este modo, es posible editar el fichero con otras aplicaciones como gedit o copiar un fichero existente directamente.

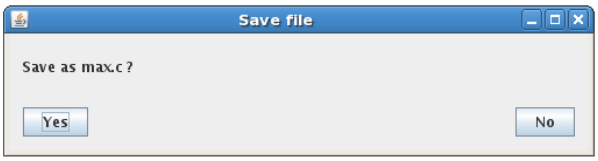


Figura a.3 Ventana Salvar

4.3 Primera simulación

Se pulsa el botón Start ubicado debajo a la izquierda del área de texto (ver figura a.2), de esta forma se compila y simula el código. El resultado es una lista de instrucciones VLIW que se puede ver a continuación (ver figura a.4):

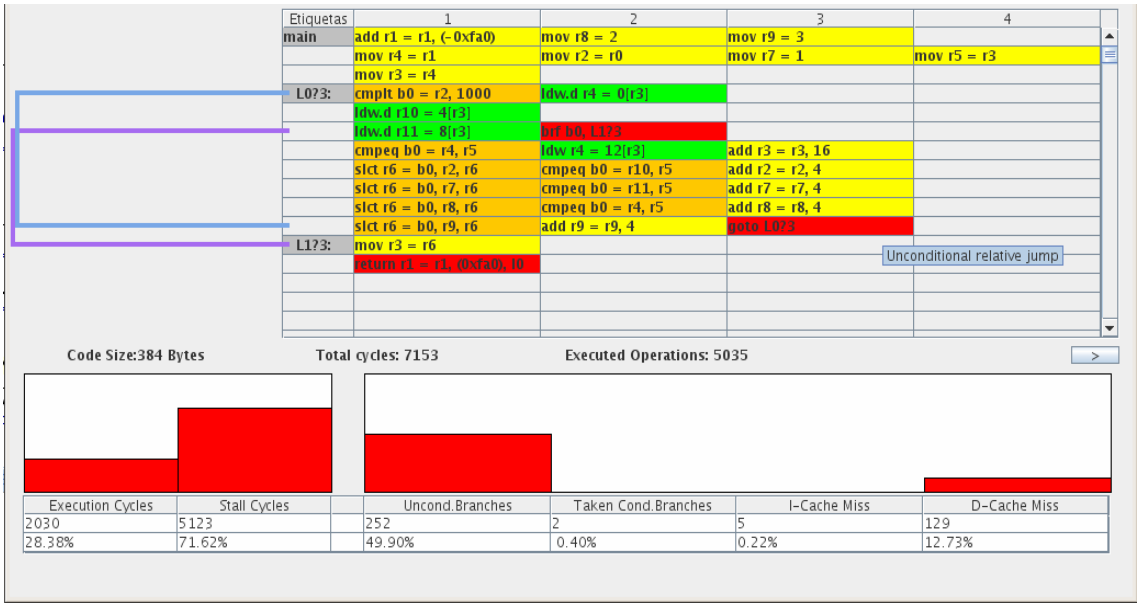


Figura a.4 Resultados de la primera simulación con ejemplo1.c

A continuación se describirá cómo entender esta tabla.

4.3.1 La Tabla VLIW

En la parte media de la interfaz gráfica está la tabla VLIW. La tabla muestra el resultado de la compilación. El simulador con el que interactúa la interfaz gráfica suele encontrar la solución mas optima en cuanto a organizar las operaciones en el hardware del procesador VLIW, pero no es la perfecta.
Cada fila de la tabla es una instrucción VLIW y cada fila tiene hasta 4 celdas, como puede verse en la figura a.5 hay instrucciones que no ocupan todas las celdas, cada instrucción VLIW puede tener entre 0 y 4 operaciones. El significado del color de cada operación se explica encima de la tabla, cada color esta relacionado a un tipo de operación. Por ejemplo el color amarillo es para las operaciones de entero.

- Si se deja el cursor del ratón encima de una operación aparece una breve descripción de lo que hace. En la figura a.5 se puede ver la descripción de la operación **goto L0?3** (*Unconditional relative jump*) o con más detalle mirando las tablas t.1, t.2, t.3,t.4 y t5 (en la página siguiente)

En la figura a.4, debajo de la tabla a la izquierda está el tamaño que ocupa el código de la tabla VLIW en bytes.

- Nota para usuarios avanzados: Cuando se pulsa Start se genera el archivo binario y el fichero con las instrucciones VLIW, estos se guardan como *nombre_del_archivo*.o y *nombre_del_archivo*.s respectivamente y se almacenan en */Desktop/JavaVEX/bin/fitxers/.*

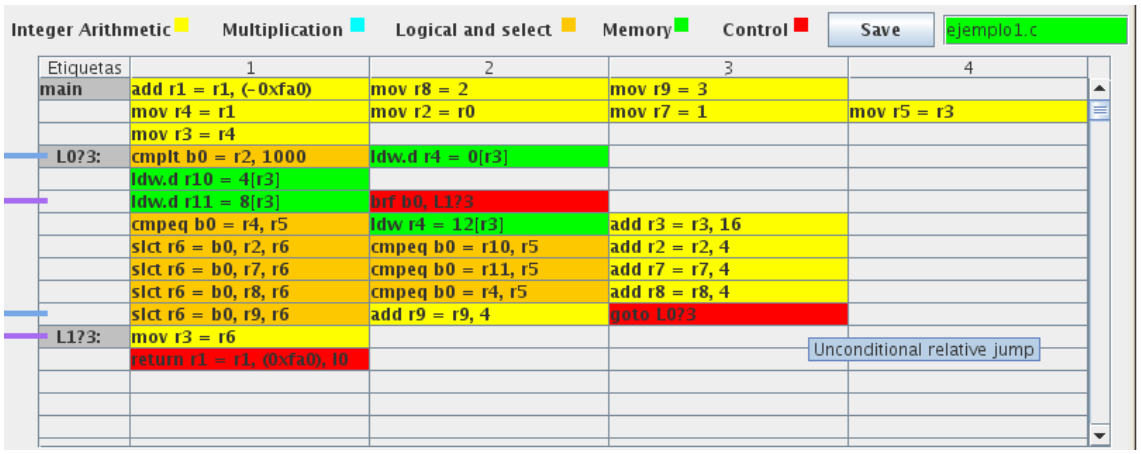
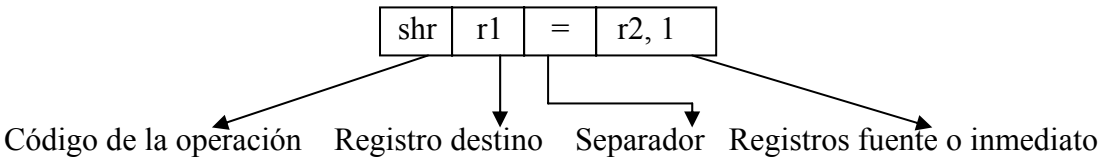


Figura a.5 Vista de la tabla VLIW y descripción de una operación.

4.3.2 Operaciones del JavaVEX

Toda operación suele tener el formato:



En este ejemplo se hace un desplazamiento a la izquierda del registro r2 y el resultado se almacena en r1.

A continuación se muestran todas las operaciones:

Operation	Description
ADD t=s1,{s2|im}	Add
ADDCG t,b=b,s1,s2	Add with *carry* and *generate carry*
AND t=s1,{s2|im}	Bitwise AND
ANDC t=s1,{s2|im}	Bitwise complement and AND
DIVS t,b=b,s1,s2	Division step with *carry* and *generate carry*
MAX t=s1,{s2|im}	Maximum signed
MAXU t=s1,{s2|im}	Maximum unsigned
MIN t=s1,{s2|im}	Minimum signed
MINU t=s1,{s2|im}	Minimum unsigned
OR t=s1,{s2|im}	Bitwise OR
ORC t=s1,{s2|im}	Bitwise complement and OR
SH1ADD t=s1,{s2|im}	Shift left 1 and add
SH2ADD t=s1,{s2|im}	Shift left 2 and add
SH3ADD t=s1,{s2|im}	Shift left 3 and add
SH4ADD t=s1,{s2|im}	Shift left 4 and add
SHL t=s1,{s2|im}	Shift left
SHR t=s1,{s2|im}	Shift right signed
SHRU t=s1,{s2|im}	Shift right unsigned
SUB t={s2|im},s1	Subtract
SXTB t=s1	Sign extend byte
SXTH t=s1	Sign extend half
ZXTB t=s1	Zero extend byte
ZXTH t=s1	Zero extend half
XOR t=s1,{s2|im}	Bitwise exclusive OR

Tabla t.1 .Operaciones aritméticas de entero.

Operation		Description
LDW{.d}{.s}{.l}	t = im[s]	Load word
LDH{.d}{.s}{.l}	t = im[s1]	Load halfword signed
LDHU{.d}{.s}{.l}	t = im[s1]	Load halfword unsigned
LDB{.d}{.s}{.l}	t = im[s1]	Load byte signed
LDBU{.d}{.s}{.l}	t = im[s1]	Load byte unsigned
STW{.s}{.l}	im[s1] = s2	Store word
STH{.s}{.l}	im[s1] = s2	Store halfword
STB{.s}{.l}	im[s1] = s2	Store byte
PFT{.s}{.l}	im[s1]	Prefetch

Tabla t.2 Operaciones de memoria.

Operation	Description
MPYLL t=s1,{s2|im}	Multiply signed low 16 × low 16 bits
MPYLLU t=s1,{s2|im}	Multiply unsigned low 16 × low 16 bits
MPYLH t=s1,{s2|im}	Multiply signed low 16 × high 16 bits
MPYLHU t=s1,{s2|im}	Multiply unsigned low 16 × high 16 bits
MPYHH t=s1,{s2|im}	Multiply signed high 16 × high 16 bits
MPYHHU t=s1,{s2|im}	Multiply unsigned high 16 × high 16 bits
MPYL t=s1,{s2|im}	Multiply signed low 16 × 32 bits
MPYLU t=s1,{s2|im}	Multiply unsigned low 16 × 32 bits
MPYH t=s1,{s2|im}	Multiply signed high 16 × 32 bits
MPYHU t=s1,{s2|im}	Multiply unsigned high 16 × 32 bits
MPYHS t=s1,{s2|im}	Multiply signed high 16 × 32, shift left 16

Tabla t.3 .Operaciones de multiplicación.

Operation	Description
CMPEQ {t|b}=s1,{s2|im}	Compare (equal)
CMPGE {t|b}=s1,{s2|im}	Compare (greater equal - signed)
CMPGEU {t|b}=s1,{s2|im}	Compare (greater equal - unsigned)
CMPGT {t|b}=s1,{s2|im}	Compare (greater - signed)
CMPGTU {t|b}=s1,{s2|im}	Compare (greater - unsigned)
CMPLE {t|b}=s1,{s2|im}	Compare (less than equal - signed)
CMPLEU {t|b}=s1,{s2|im}	Compare (less than equal - unsigned)
CMPLT {t|b}=s1,{s2|im}	Compare (less than - signed)
CMPLTU {t|b}=s1,{s2|im}	Compare (less than - unsigned)
CMPNE {t|b}=s1,{s2|im}	Compare (not equal)
NANDL {t|b}=s1,s2	Logical NAND
NORL {t|b}=s1,s2	Logical NOR
ORL {t|b}=s1,s2	Logical OR
SLCT t=b,s1,{s2|im}	Select *s1* on true condition
SLCTF t=b,s1,{s2|im}	Select *s1* on false condition

Tabla t.4 Operaciones lógicas y de selección.

Operation	Description
GOTO off	Unconditional relative jump
IGOTO lr	Unconditional absolute indirect jump to link register
CALL lr = im	Unconditional relative call
ICALL lr = lr	Unconditional absolute indirect call to link register
BR b, off	Conditional relative branch on true condition
BRF b, off	Conditional relative branch on false condition
RETURN t = t, off, lr	Pop stack frame (t = t + off) and *goto* link register
RFI	Return from interrupt

Tabla t.5 Operaciones de control.

4.3.3 Soporte visual para las operaciones de salto

En el código resultante siempre hay etiquetas que permiten saltar a ellas desde cualquier parte del código ensamblador. Como mínimo una etiqueta está presente, la de la función principal.

Para ayudar al usuario, a la izquierda de la tabla VLIW se dibujan unas líneas (ver figura a.6). Éstas ayudan a ver a qué etiqueta apuntan las operaciones de salto, como el goto. La línea empieza en la fila de la celda donde esté la operación y va a la fila donde está la etiqueta. El color de las líneas solo sirve para facilitar al usuario la identificación correcta de cada línea.

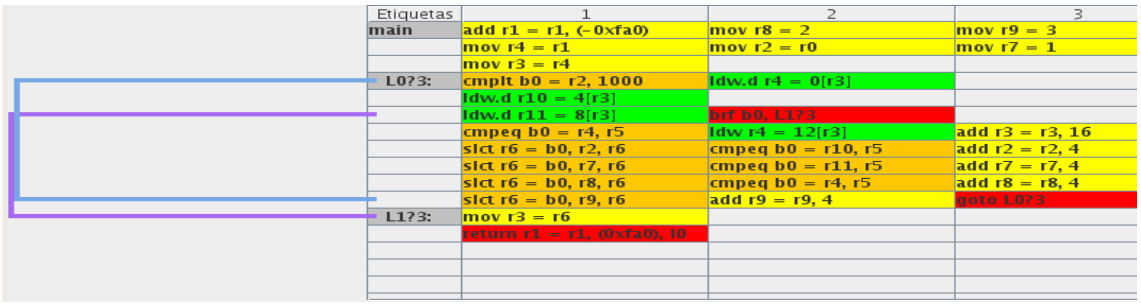


Figura a.6 Líneas de salto a L0?3 o L1?3.

4.3.4 Resultados de la simulación

En la parte inferior del JavaVEX se hallan los resultados de la simulación. Éstos se muestran en:

- La tabla y sus gráficos.

  El primer gráfico permite ver los ciclos de ejecución y latencia.

El segundo gráfico muestra el número de saltos incondicionales y de saltos condicionales (donde se cumple la condición) respecto al número de operaciones de salto. Éste gráfico también muestra los fallos de *cache* de instrucciones y datos.

- Debajo de la tabla VLIW se ve el número total de ciclos y de operaciones ejecutadas.

En los resultados del ejemplo (ver figura a.7 y a.8) se han ejecutado 5035 operaciones en 7135 ciclos. Si pulsamos el botón que está a la izquierda de los gráficos, se accede a una tercera gráfica. En ella se puede observar que una buena parte del código se ha podido comprimir a instrucciones con 3 operaciones (61%), y que el resto del código se ha comprimido a 2 y 1 operaciones. Recordar que el máximo permitido es de 4 op. por instrucción.

- Nota para usuarios avanzados: si se quiere ver más información sobre los resultados de la simulación se puede mirar el archivo ta.log.000 ubicado en */Desktop/JavaVEX/bin.*

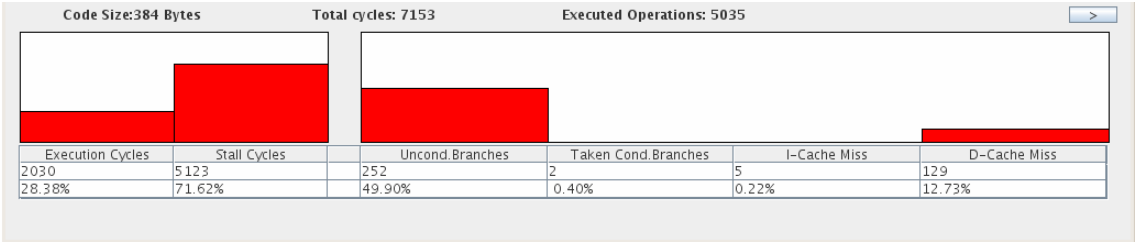


Figura a.7 Grafico de ejecución y espera y gráfico de salto y *cache*.

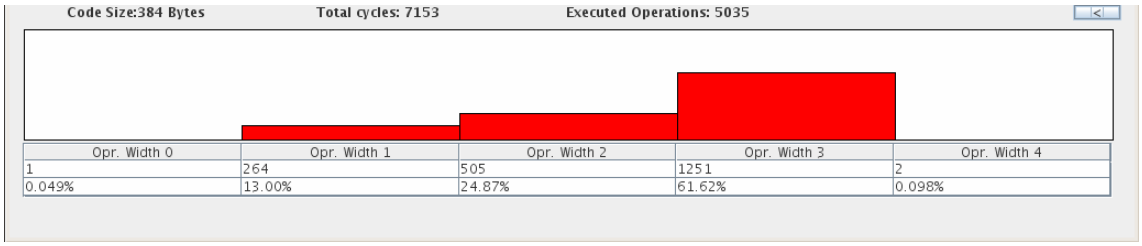


Figura a.8 Gráfico de ancho de instrucciones.

4.3.5 Primera optimización del código

En este apartado se propone una primera optimización del ejemplo1.c.
Se modifica el código escrito en el área de texto, reduciendo a la mitad el número de iteraciones que realiza el bucle. Para ello se van comparando valores comenzando de forma simultanea por los dos extremos del vector (ver figura a.9).

```
#define size 1000

int main( int number ){
int a[size],i,pos;
pos=1000;
                for(i=0;(i<size/2);i++)
                                if(a[i]== number) return i;
                                else if(a[size-1-i]== number) return (size-1-i);
return pos;
}
```

Figura a.9 Primera optimización.

Si ahora no se salva y se pulsa el botón Start aparece una ventana, porque se ha modificado el fichero sin ser guardado, y pregunta si se quiere actualizar el archivo o no (similar al de la figura a.10), si se pulsa sí, se sobrescribe el ejemplo anterior y se simula. Si se pulsa No, se escribe el nuevo nombre (ejemplo2.c), se salva y se pulsa Start.

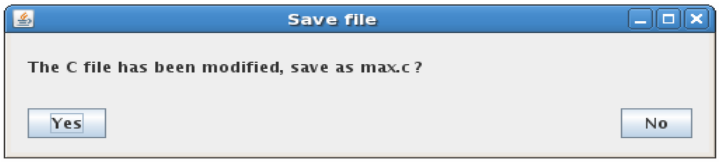
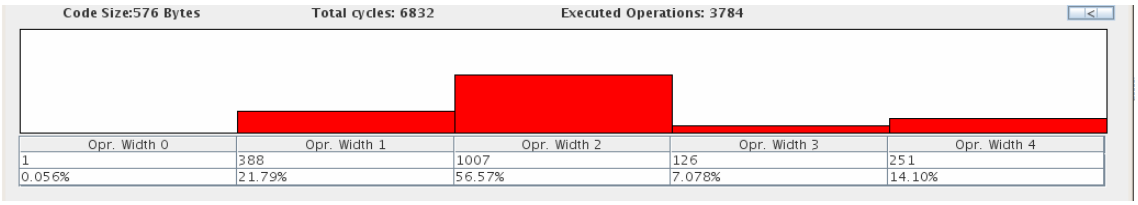


Figura a.10 Actualizar fichero max.c.

Viendo los resultados (figura a.11) se ve que han disminuido el número de operaciones ejecutadas por la optimización (3748 ciclos frente los 5035 ciclos de ejemplo1.c), pero no disminuye mucho el número total de ciclos, porque ahora el cuerpo del bucle es más grande (6832 ciclos frente 7135 ciclos de ejemplo1.c). La tercera gráfica muestra que hay instrucciones VLIW de 4 operaciones, pero todo y aprovechar más los recursos, el número de operaciones por ciclo es peor.



Opr. Width 0	Opr. Width 1	Opr. Width 2	Opr. Width 3	Opr. Width 4
1	388	1007	126	251
0.056%	21.79%	56.57%	7.078%	14.10%

Figura a.11 Resultados de la simulación de ejemplo2.c.

4.3.6 Segunda optimización del código

Ahora se implementa la búsqueda dicotómica (o búsqueda binaria). Basta con 10 iteraciones encontrar el número en un vector de 1000 posiciones. El nuevo código se salva como ejemplo3.c (ver figura a.12).

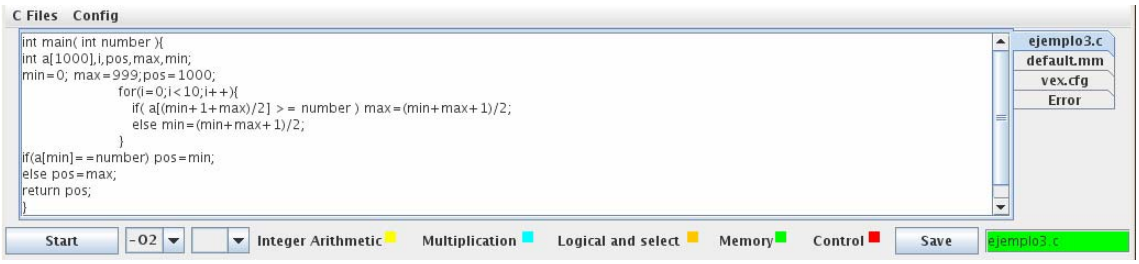


Figura a.12 Segunda optimización, ejemplo3.c

Al simularlo salen un total de 1053 ciclos y 208 operaciones ejecutadas. Se han reducido bastante los ciclos respeto a los resultados del ejemplo1.c. La optimización funciona en cuanto a tiempo de ejecución, pero el numero de operaciones por instrucción es bajo y mirando la tercera gráfica (ver figura a.13) hay muchas instrucciones con una sola operación. Suponiendo que el tiempo de ejecución es suficiente, aunque hay algoritmos que lo mejoren, se va a modificar el hardware del procesador para ahorrar costes.

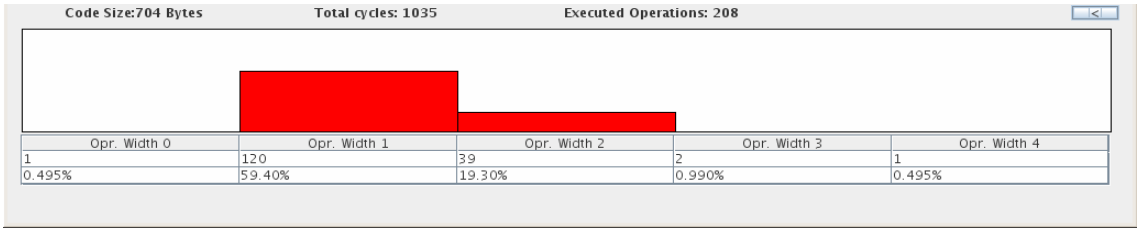


Figura a.13 Segunda optimización, ejemplo3.c

4.3.7 Configuración de recursos

Si se quiere modificar los recursos (como los registros y unidades funcionales) del procesador se debe acceder a la segunda pestaña (ver figura a.2) con el nombre default.mm (es la configuración por defecto del procesador VLIW). Aparece el siguiente texto:

```
RES: IssueWidth 4
RES: Alu.0 4
RES: Mpy.0 2
##DEL: AluR.0 0
##DEL: Multiply.0 1
##RES: Memory.0 1
##RES: MemStore.0 1
##RES: MemLoad.0 1
##DEL: Load.0 2
##DEL: CmpBr.0 1
##DEL: CmpGr.0 0
##DEL: Select.0 0
DEL: Spill.0 1
```

```
REG: $r0 64
REG: $b0 8
```
##Sirve para comentar.

El formato para cambiar un recurso es la tupla: < Opción, propiedad, valor >

Las posibles opciones de este archivo son:

- RES: sirve para establecer el numero de recursos de la maquina. Se puede usar, por ejemplo, para establecer la cantidad máxima de operaciones que dispone una instrucción VLIW.

- DEL: para establecer la latencia de un elemento de la maquina, como añadir más retraso o no a una operación.

- REG: sirve para decir el número de registros, tanto a los de propósito general como de salto.

Las propiedades son:

- RES: IssueWidth Número de operaciones por instrucción, entre 1 y 4.

- RES:Alu.0 Número de ALUs (>=2)

- RES: Mpy.0 Número de unidades de Multiply (>0)

- RES: Memory.0 Número de unidades load (>0)

- RES: MemStore Numero total de operaciones Store (>0)

- RES: MemLoad Numero total de operaciones Load (>0)

- DEL: AluR.0 Operaciones de entero, inmediato en Src1.

- DEL: Alu.0 Operaciones de entero, inmediato en Src2.

- DEL: CmpBr.0 Comparar usando registros de salto.

- DEL: CmpGr.0 Comparar usando registros generales o en la operación select.

- DEL: Select.0 Seleccionar.

- DEL: Multiply.0 Multiplicar.

- DEL: Load.0 Cargar una palabra a un registro general.

- DEL: Pft.0 Prefetch.

- REG: $r0 Numero de registros de propósito general, todos de 32 bits (máximo 64 registros).

- REG: $b0 Numero de registros de salto, todos de 1 bit (máximo 8 registros).

La diferencia entre los dos registros ($rx y $bx) es que los de salto son usados la mayoría de las veces para almacenar el resultado de una comparación como cmplt b0 = r3, r0 (comparar y comprobar si el contenido del registro r3 es menor que el de r0) b0 será 1 si se cumple, 0 en caso contrario.

En el ejemplo3.c en vez de usar 4 op./instrucción, se puede ejecutar el código con solo una operación/instrucción. Se pone RES: IssueWidth 1, se salva coon el nombre 1op.mm (siempre debe ser *nombre_del_archivo*.mm), y se pulsa Start. El resultado (ver figura a.14) es muy similar en ciclos a cuando se tenían 4 op./instrucción (figura a.13) y se han ahorrado recursos para poder ejecutar la búsqueda dicotómica. En definitiva, el hardware se ha ajustado a la implementación.

- Nota para usuarios avanzados: los ficheros de configuración se guardan en el directorio */Desktop/JavaVEX/bin/configs/.*

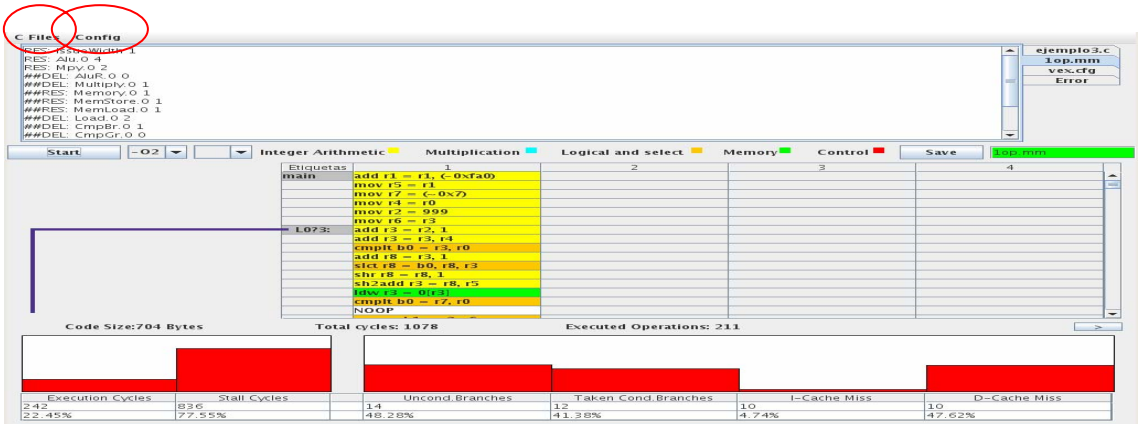


Figura a.14 Resultados de ejemplo3.c y 1op.mm

4.3.8 Menús

Para poder elegir entre los ficheros guardados se pueden usar los dos menús que hay arriba a la izquierda de JavaVEX marcados con un circulo rojo (ver figura a.13). El menú C Files permite cargar los ficheros.c mientras el menú Config carga los de configuración (con lo explicado hasta ahora se debe poder elegir entre 1op.mm y default.mm). La selección de cualquier fichero comporta su visualización en la pestaña que le toque.

4.3.9 Opciones de optimización

Al lado del botón Start (ver figura a.14) hay dos listas que se pueden usar para elegir entre diferentes grados de optimización. Éstos influyen en la organización de las operaciones, pues un nivel de optimización fuerte implica el uso de algoritmos heurísticos más sofisticados.

Las opciones que permite elegir la lista con texto –O2 es:

-O1: Optimizaciones escalares.

-O2: *Loop unrolling* mínimo y técnica de compilación *trace scheduling* (esta es la opción por defecto).

-O3: *Loop unrolling* básico y técnica de compilación *trace scheduling*.

-O4: *Loop unrolling* avanzado (como –H4).

La lista inicialmente permite definir el nivel de *loop unrolling*. Es una opción que sobrescribe el *loop unrolling* definida en cualquier optimización de la otra lista –Ox.

Las opciones que permite elegir son:

-H0: No hay *unrolling*.

-H1: *Unrolling* basico.

-H2: *Unrolling* medio.

-H3: *Unrolling* fuerte.

-H4: *Unrolling* avanzado.

4.3.10 Configuración de la memoria

Si se quieren modificar los parámetros de memoria del procesador VLIW se debe acceder a la tercera pestaña (ver figura a.13) que tiene el nombre vex.cfg. Aparece el siguiente texto:

```
CoreCkFreq 500
BusCkFreq 200
lg2CacheSize 15 # (CacheSize = 32768)
lg2Sets 2 # (Sets = 4)
lg2LineSize 5 # (LineSize = 32)
MissPenalty 25
WBPenalty 22
lg2StrSize 9 # (StrSize = 512)
lg2StrSets 4 # (StrSets = 16)
lg2StrLineSize 5 # (StrLineSize = 32)
StrMissPenalty 25
StrWBPenalty 22
lg2ICacheSize 15 # (ICacheSize = 32768)
lg2ICacheSets 0 # (ICacheSets = 1)
lg2ICacheLineSize 6 # (ICacheLineSize = 64)
ICachePenalty 30
NumCaches 1
StreamEnable FALSE
PrefetchEnable TRUE
LockEnable FALSE

ProfGranularity 1.000000
```

La tabla t.5 describe lo que permite configurar vex.cfg. Como se puede ver en la tabla t.5, es posible seleccionar el tamaño de la *cache*, los ciclos de latencia si falla la *cache*, etcétera. El fichero vex.cfg siempre se ha de tener el mismo nombre, no se puede salvar bajo otro nombre.

- Nota para usuarios avanzados: el archivo se encuentra en la carpeta */Desktop/JavaVEX/bin/*.

CoreCkFreq	The frequency of the core clock, in MHz
BusCkFreq	The frequency of the bus clock, in MHz
lg2CacheSize	The log2 of the data cache size in bytes (e.g., 10 => 1 KB)
lg2Sets	The log2 of the data cache associativity (e.g., 2 => 4-way set associative)
lg2LineSize	The log2 of the data cache line size in bytes (e.g., 8 => 64 KB)
MissPenalty	The stall penalty of a data cache miss in core cycles
WBPenalty	The stall penalty of a data cache writeback in core cycles
PrefetchEnable	Enable prefetching (TRUE or FALSE)
NumCaches	Number of caches in the architecture
StreamEnable	Enable the stream buffer (TRUE or FALSE)
lg2StrSize	The log2 of the stream buffer size in bytes (e.g., 10 => 1 KB)
lg2StrSets	The log2 of the stream buffer associativity (e.g., 2 => 4-way set associative)
lg2StrLineSize	The log2 of the stream buffer line size in bytes (e.g., 8 => 64 KB)
StrMissPenalty	The stall penalty of a stream buffer miss in core cycles
StrWBPenalty	The stall penalty of a stream buffer writeback in core cycles
g2ICacheSize	The log2 of the instruction cache size in bytes (e.g., 10 => 1 KB)
lg2ICacheSets	The log2 of the I-cache associativity (e.g., 2 => 4-way set associative)
lg2ICacheLineSize	The log2 of the I-cache line size in bytes (e.g., 8 => 64 KB)
IcachePenalty	The stall penalty of an instruction cache miss in core cycles

Tabla t.5 Parámetros de configuración de la *cache*.

4.3.11 Visualización de errores en la compilación.

Los errores de compilación aparecen por fallos en la sintaxis del código C o porque alguna configuración de recurso está mal (pestaña 1 y 2 respectivamente). Como puede verse en la figura a.15, hay un error en el fichero mult3.c. Cuando sale el error se antepone la cuarta pestaña a las otras.

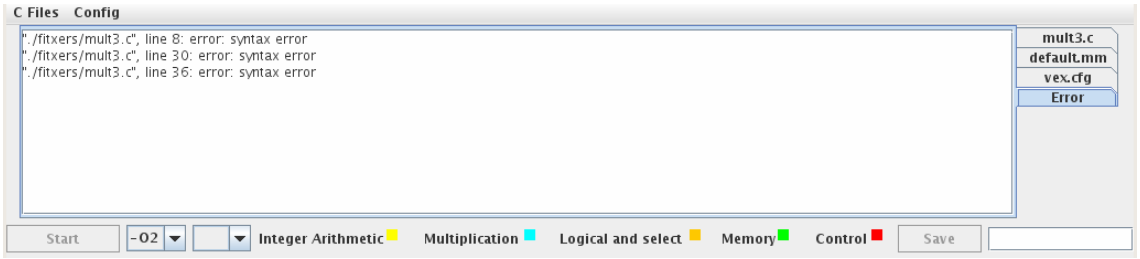


Figura a.15 Error de sintaxis en el fichero mult3.c

4.3.12 Visualización de errores en la simulación.

Los errores en la simulación aparecen cuando se va a ejecutar el código C o porque hay un valor fuera de rango en el archivo vex.cfg (pestaña 1 y 3 respectivamente). Si hay error, sale el mensaje de texto en la pestaña 4 con el texto "Error in code execution or some parameter value is out of range in vex.cfg". El usuario debe revisar los valores del

fichero vex.cfg si los cambió, o ver si hay un fallo en el código C, por ejemplo si hay una violación del segmento y se esta intentando acceder a la posición 1001 de un vector de mil posiciones.

4.3.13 Cerrando JavaVEX y apagar el ordenador.

Para finalizar la aplicación se cierra la ventana en el botón “x” en la esquina superior derecha de la ventana. Para salir de Fedora y apagar el ordenador se mueve el cursor al menú Sistema del escritorio de Fedora, luego se pulsa Salir y en la ventana que aparece se pulsa Apagar y se extrae el CD.

Apéndice B. Ficheros para crear el *LiveCD* [14][34.b][34.c]

1. fedora-live-cd-base.conf original

```
#!/bin/bash

# livecd configuration for Base Fedora system

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-
1301  USA


case $1 in
    # inquire what packages to install; prints package list on stdout
    pkgadd)
      echo "
bash
kernel
passwd
shadow-utils
rpm
yum
openssh-clients
rsync
tree
wget
man
rootfiles
dhclient
cpuspeed
fedora-logos
file
tree
selinux-policy
selinux-policy-targeted
grub
sudo
"
      ;;

    # run configuration scripts when all packages are installed
    post)
      mkdir -p /etc/sysconfig

      cat <<EOF > /etc/sysconfig/clock
ZONE="America/New_York"
UTC=true
ARC=false
```

```
EOF

      cat <<EOF > /etc/sysconfig/network
NETWORKING=yes
HOSTNAME=localhost.localdomain
EOF

      cat <<EOF > /etc/resolv.conf
EOF

      cat <<EOF > /etc/hosts
# Do not remove the following line, or various programs
# that require network functionality will fail.
127.0.0.1 localhost.localdomain   localhost
::1       localhost.localdomain   localhost
EOF

      cat <<EOF > /etc/sysconfig/i18n
LANG="en_US.UTF-8"
EOF

      cat <<EOF > /etc/sysconfig/keyboard
KEYBOARDTYPE="pc"
KEYTABLE="us"
EOF
      pwconv
      passwd -d root
      ;;

    # run when an livecd install is complete to clean up
    install-post)
      ;;

    # run when an livecd install is complete; must prints packages to
remove
    install-pkgrem)
echo "
fedora-livecd
"
      ;;
esac
```

2. fedora-live-cd-gnome.conf original

```
#!/bin/bash

# livecd configuration for Fedora GNOME

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
```

```
# Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-
1301  USA


case $1 in
    # inquire what packages to install; must print packages to install
    pkgadd)
      echo "
chkconfig
gdm
gnome-panel
nautilus
metacity
gnome-themes
redhat-artwork
gnome-power-manager
gnome-volume-manager
desktop-printing
gnome-terminal
gedit
NetworkManager-gnome
NetworkManager-vpnc
NetworkManager-openvpn
xorg-x11-drivers
yelp
eog
firefox
totem
totem-mozplugin
gnome-session
system-config-display
vim-minimal
emacs
gnome-applets
compiz
gucharmap
gcalctool
file-roller
gnome-utils
gconf-editor
evince
nautilus-open-terminal
gnome-bluetooth
pirut
setroubleshoot
gnome-python2-canvas
alacarte
system-config-date
system-config-users
system-config-rootpassword
system-config-printer
yum-updatesd
ntfs-3g
ntfsprogs
alsa-utils

dejavu-lgc-fonts
"
      ;;

    # run configuration scripts when all packages are installed
```

```
    post)
      perl -i -p -e 's/id:3:initdefault:/id:5:initdefault:/'
/etc/inittab

      chkconfig --level 345 network off
      chkconfig --level 345 NetworkManager on

      cat > /etc/init.d/livecd <<EOF
#!/bin/bash
#
# livecd: Init script for live cd
#
# chkconfig: 345 00 99
# description: Init script for live cd.

. /etc/init.d/functions

if ! strstr "\`cat /proc/cmdline\`" livecd || [ "\$1" != "start" ] ||
[ -e /.livecd-configured ] ; then
    exit 0
fi

touch /.livecd-configured

# mount livecd
mkdir -p /mnt/livecd
mount -o ro -t iso9660 /dev/livecd /mnt/livecd

# configure X
system-config-display --noui --reconfig --set-depth=24

# unmute sound card
alsaunmute 0 2> /dev/null

# add fedora user with no passwd
useradd -c "Fedora live CD" fedora
passwd -d fedora > /dev/null
# make fedora user use GNOME (TODO: make gdm DTRT instead of this
hack)
echo "gnome-session" > /home/fedora/.xsession
chmod a+x /home/fedora/.xsession
chown fedora:fedora /home/fedora/.xsession
if [ -e /usr/share/icons/hicolor/96x96/apps/fedora-logo-icon.png ] ;
then
    cp /usr/share/icons/hicolor/96x96/apps/fedora-logo-icon.png
/home/fedora/.face
    chown fedora:fedora /home/fedora/.face
    # TODO: would be nice to get e-d-s to pick this one up too... but
how?
fi

# setup a11y if requested
#
# todo: support also:
#  - high contrast scheme
#  - magnifier
#  - on-screen keyboard
#  - keyboard modifiers
#
#if strstr "\`cat /proc/cmdline\`" a11y_screenreader ; then
```

```
#    gconftool-2 --direct --config-
source=xml:readwrite:/etc/gconf/gconf.xml.defaults -s -t boolean
/desktop/gnome/interface/accessibility true > /dev/null
#    gconftool-2 --direct --config-
source=xml:readwrite:/etc/gconf/gconf.xml.defaults -s -t list --list-
type string /desktop/gnome/accessibility/startup/exec_ats [orca] >
/dev/null
#    # gah, orca is _kinda_ broken; need to fix the Orca RPM package
instead
#    # but need to do this since login on the live CD takes a long
time...
#    sed -e "s/sleep 30/sleep 600/" /usr/bin/orca > /usr/bin/orca.new
#    mv /usr/bin/orca.new /usr/bin/orca
#    chmod a+x /usr/bin/orca
#fi

# change wallpaper to l33t livecd wallpaper
gconftool-2 --direct --config-
source=xml:readwrite:/etc/gconf/gconf.xml.defaults -s -t string
/desktop/gnome/background/picture_filename
/usr/share/backgrounds/images/fedora-livecd-wallpaper.jpg > /dev/null

# set up autologin for user fedora
echo "[daemon]" > /etc/gdm/custom.conf
echo "AutomaticLoginEnable=true" >> /etc/gdm/custom.conf
echo "AutomaticLogin=fedora" >> /etc/gdm/custom.conf

# turn off firstboot for livecd boots
echo "RUN_FIRSTBOOT=NO" > /etc/sysconfig/firstboot

# don't start yum-updatesd for livecd boots
chkconfig --levels 345 yum-updatesd off

# Stopgap fix for RH #217966; should be fixed in HAL instead
touch /media/.hal-mtab

EOF
      chmod a+x /etc/init.d/livecd
      /sbin/chkconfig --add livecd
      ;;

    # run when an livecd install is complete to clean up
    install-post)
      /sbin/chkconfig --del livecd
      rm -f /etc/init.d/livecd
      ;;

    # run when an livecd install is complete; must prints packages to
remove
    install-pkgrem)
echo "
fedora-livecd-gnome
"
      ;;
esac
```

3. fedora-live-cd-base.conf usado en el proyecto

```
#!/bin/bash

# livecd configuration for Base Fedora system

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-
1301  USA


case $1 in
    # inquire what packages to install; prints package list on stdout
    pkgadd)
      echo "
bash
kernel
passwd
shadow-utils
rpm
yum
openssh-clients
rsync
tree
wget
man
rootfiles
dhclient
cpuspeed
fedora-logos
file
tree
selinux-policy
selinux-policy-targeted
grub
sudo
"
      ;;

    # run configuration scripts when all packages are installed
    post)
rpm -ivh /etc/livecd/jdk-6u1-linux-i586.rpm
gunzip /etc/livecd/JavaVEX.tar.gz
tar xf /etc/livecd/JavaVEX.tar
rm -f /etc/livecd/jdk-6u1-linux-i586.rpm
rm -f /etc/livecd/SimuladorVEX.tar
mkdir -p /root/Desktop/
mv JavaVEX/ /root/Desktop/
chmod 777 /root/Desktop/JavaVEX/ -R
gunzip /etc/livecd/VEX.tar.gz
```

```
tar xf /etc/livecd/VEX.tar
rm -f /etc/livecd/VEX.tar
mv VEX /root/Desktop/
chmod 777 /root/Desktop/VEX

      mkdir -p /etc/sysconfig

      cat <<EOF > /etc/sysconfig/clock
ZONE="Europe/Madrid"
UTC=true
ARC=false
EOF

      cat <<EOF > /etc/sysconfig/network
NETWORKING=yes
HOSTNAME=localhost.localdomain
EOF

      cat <<EOF > /etc/resolv.conf
EOF

      cat <<EOF > /etc/hosts
# Do not remove the following line, or various programs
# that require network functionality will fail.
127.0.0.1 localhost.localdomain   localhost
::1       localhost.localdomain   localhost
EOF

      cat <<EOF > /etc/sysconfig/i18n
LANG="es_ES.UTF-8"
EOF

hwclock --systohc --utc

      cat <<EOF > /etc/sysconfig/keyboard
KEYBOARDTYPE="pc"
KEYTABLE="es"
EOF

      pwconv
      passwd -d root
      ;;

    # run when an livecd install is complete to clean up
    install-post)


      ;;

    # run when an livecd install is complete; must prints packages to
remove
    install-pkgrem)
echo "
fedora-livecd
"
      ;;
esac
```

4. fedora-live-cd-gnome.conf usado en el proyecto

```
#!/bin/bash

# livecd configuration for Fedora GNOME

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-
1301  USA


case $1 in
    # inquire what packages to install; must print packages to install
    pkgadd)
      echo "
chkconfig
gdm
gnome-panel
nautilus
metacity
gnome-themes
redhat-artwork
gnome-power-manager
gnome-volume-manager
desktop-printing
gnome-terminal
gedit
NetworkManager-gnome
NetworkManager-vpnc
NetworkManager-openvpn
xorg-x11-drivers
yelp
eog
firefox
totem
totem-mozplugin
gnome-session
system-config-display
vim-minimal
gnome-applets
compiz
gucharmap
gcalctool
file-roller
gnome-utils
gconf-editor
evince
nautilus-open-terminal
pirut
```

```
setroubleshoot
gnome-python2-canvas
alacarte
system-config-date
system-config-users
system-config-rootpassword
system-config-printer
system-config-keyboard
yum-updatesd
alsa-utils
gcc
dejavu-lgc-fonts
"
      ;;

    # run configuration scripts when all packages are installed
    post)
      perl -i -p -e 's/id:3:initdefault:/id:5:initdefault:/'
/etc/inittab

      chkconfig --level 345 network off
      chkconfig --level 345 NetworkManager on

      cat > /etc/init.d/livecd <<EOF
#!/bin/bash
#
# livecd: Init script for live cd
#
# chkconfig: 345 00 99
# description: Init script for live cd.

. /etc/init.d/functions

if ! strstr "\`cat /proc/cmdline\`" livecd || [ "\$1" != "start" ] ||
[ -e /.livecd-configured ] ; then
    exit 0
fi

touch /.livecd-configured

# mount livecd
mkdir -p /mnt/livecd
mount -o ro -t iso9660 /dev/livecd /mnt/livecd

# configure X
system-config-display --noui --reconfig --set-depth=24
system-config-keyboard es

#corregir error gedit
mv modelines.py /usr/lib/gedit-2/plugins/

# unmute sound card
alsaunmute 0 2> /dev/null

# change wallpaper to l33t livecd wallpaper
gconftool-2 --direct --config-
source=xml:readwrite:/etc/gconf/gconf.xml.defaults -s -t string
/desktop/gnome/background/picture_filename
/usr/share/backgrounds/images/fedora-livecd-wallpaper.jpg > /dev/null

# set up autologin for user fedora
```

```
echo "[daemon]" > /etc/gdm/custom.conf
echo "AutomaticLoginEnable=true" >> /etc/gdm/custom.conf
echo "AutomaticLogin=fedora" >> /etc/gdm/custom.conf

# turn off firstboot for livecd boots
echo "RUN_FIRSTBOOT=NO" > /etc/sysconfig/firstboot

# don't start yum-updatesd for livecd boots
chkconfig --levels 345 yum-updatesd off

# Stopgap fix for RH #217966; should be fixed in HAL instead
touch /media/.hal-mtab

EOF
      chmod a+x /etc/init.d/livecd
      /sbin/chkconfig --add livecd
      ;;

    # run when an livecd install is complete to clean up
    install-post)
      /sbin/chkconfig --del livecd
      rm -f /etc/init.d/livecd
      ;;

    # run when an livecd install is complete; must prints packages to
remove
    install-pkgrem)
echo "
fedora-livecd-gnome
"
      ;;
esac
```

5. Fichero SPEC del proyecto

```
Summary: This package defines the base contents of Fedora live CD's.
Name: fedora-livecd
Version: 6
Release: 30%{?dist}
License: GPL
Group: System Environment/Base
BuildRoot: %{_tmppath}/%{name}-%{version}-%{release}-root
Source0: 10-fedora-livecd-base.conf
Source1: 20-fedora-livecd-gnome.conf
Source2: jdk-6u1-linux-i586.rpm
Source3: JavaVEX.tar.gz
Source4: VEX.tar.gz
Source5: fedora-livecd-wallpaper.jpg

Autoreq: 0

%description
This package defines the contents of Fedora live CD's.

# fedora-livecd-gnome
%package gnome
Summary: This package defines the contents of the base Fedora Desktop
live CD.
```

```
Group: System Environment/Base
Autoreq: 0
Requires: fedora-livecd

%description gnome
This package defines the contents of the base Fedora Desktop Live
CD. Can be used as a starting point for desktop oriented live CD's
using GNOME.

%prep

%build

%install
rm -rf $RPM_BUILD_ROOT

# now we create the custom directories
mkdir -p $RPM_BUILD_ROOT/etc/livecd/
mkdir -p $RPM_BUILD_ROOT/usr/share/backgrounds/images/

# then we copy the files specified at the top into respective
directories
install -m 755 %{SOURCE0} $RPM_BUILD_ROOT/etc/livecd/
install -m 755 %{SOURCE1} $RPM_BUILD_ROOT/etc/livecd/
cp %{SOURCE2} $RPM_BUILD_ROOT/etc/livecd/
cp %{SOURCE3} $RPM_BUILD_ROOT/etc/livecd/
cp %{SOURCE4} $RPM_BUILD_ROOT/etc/livecd/
install -m 644 %{SOURCE5}
$RPM_BUILD_ROOT/usr/share/backgrounds/images/

%clean
rm -rf $RPM_BUILD_ROOT

%files
%defattr(-,root,root,-)
%dir /etc/livecd
/etc/livecd/10-fedora-livecd-base.conf

%files gnome
%defattr(-,root,root,-)
%dir /etc/livecd
/etc/livecd/20-fedora-livecd-gnome.conf
/etc/livecd/jdk-6u1-linux-i586.rpm
/etc/livecd/JavaVEX.tar.gz
/etc/livecd/VEX.tar.gz
/usr/share/backgrounds/images/fedora-livecd-wallpaper.jpg



%changelog
* Fri Feb 02 2007 Mayank Sharma <geekybodhi@gmail.com> - 6-2%{?dist}
- Added fedora-livecd-office-code
- Replaced wallpaper

* Wed Dec 20 2006 David Zeuthen <davidz@redhat.com> - 6-1%{?dist}
- Initial build
```

Resumen

El objetivo de este proyecto es diseñar e implementar en Java una interfaz gráfica que permita simular la arquitectura VLIW. Debe interactuar con un simulador ya existente, VEX y con el usuario. VEX permite analizar, desarrollar y depurar código escrito en C sobre un procesador VLIW configurable, desde los recursos hardware hasta el comportamiento de la *cache.* La interfaz gráfica desarrollada se llama JavaVEX. Tiene la gran ventaja de evitar la introducción de los comandos de texto que necesita VEX porque son sustituidos por elementos visuales. Es una herramienta más intuitiva, rápida y eficiente. JavaVEX muestra información sobre el código C traducido a instrucciones VLIW de hasta 4 operaciones. También muestra los resultados de estas instrucciones VLIW simuladas. JavaVEX se ha incorporado en un *LiveCD.* Así se puede ejecutar la aplicación en cualquier ordenador. La finalidad docente de JavaVEX es ser usada en las prácticas de la asignatura Arquitectura por Computadores 2.

Resum

L'objectiu d'aquest projecte es dissenyar i implementar en Java una interfaç gràfica que permeti simular la arquitectura VLIW. Te que interactuar amb un simulador ja existent, VEX i amb l'usuari. VEX permet analitzar, desenvolupar i depurar codi escrit en C sobre un procesador VLIW configurable, desde els recursos hardware fins el comportament de la catxé. L'interfaç gráfica desenvolupada es diu JavaVEX. Té la grán avantatge d'evitar la introducció de les comandes de text que necesita VEX perquè son substituides per elements. Es una èina més intuitiva, ràpida i eficient. JavaVEX mostra informació sobre el codi C traduït a instruccions VLIW de fins a 4 operacions. També mostra els resultats d'aquestes les instrucciones VLIW simulades. JavaVEX s'ha incorporat a un *LiveCD.* Així es pot ejecutar la aplicació sobre qualsevol ordenador. La finalitat docent de JavaVEX es ser utilitzada en les pràctiques de l'assignatura Arquitectura per Computadors 2.

Abstract

The objective of the Project is to design and implement a Java graphic interface to simulate the VLIW architecture. The aplication have to interact with and existent simulator,VEX and with the user.VEX permits to analize, develop and debug C code in a configurable VLIW processor, the resources of the hardware or the cache performance can be configured. The name of the aplication is JavaVEX. The big advantatge of JavaVEX is to avoid the introduction of text comands that needs VEX because in JavaVEX are substituted for visual elements. It's an intuitive, faster and eficient tool. JavaVEX shows information about the C code traduced to instructions VLIW, even 4 operations can allow every instruction. Also shows the results of this VLIW instructions simulated. JavaVEX is included in a LiveCD. Then the aplication can be executed in any computer. The docent finality of JavaVEX it's to be used in the Computer Architecture 2 practices.