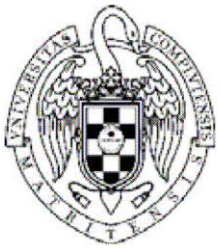


Facultad de Informática
Universidad Complutense de Madrid

Proyecto de Sistemas Informáticos
Curso 2007/08

Diseño y caracterización de un procesador sintetizable de ultra-bajo consumo para sistemas empotrados multimedia

Jordi Terns Vall-llovera

Dirigido por:
David Atienza Alonso
José Luís Ayala Rodrigo
Departamento de Arquitectura de Computadores y Automática

El autor de este proyecto autoriza a la Universidad Complutense a difundir y utilizar con fines académicos no comerciales tanto la memoria, como el código, la documentación y el prototipo de la plataforma desarrollada.

Jordi Terns Vall-llovera

Contenido

English summary

In the last years, VLIW processors are becoming a preferable solution for all those applications where high performance and low power consumption are needed. Therefore, this is a common need in embedded systems running multimedia applications.

Several architectures and VLIW processors have been proposed lately in the specialized literature. One of these architectures is the VEX processor, which integrates simplicity and efficiency. In this Project, we have used for our experimental work a VEX-like architecture designed using the LISATek environment.

The baseline architecture presented several limitations that showed out in the way of instruction hazards during the execution of the applications. One of the mail goals of our work was to detect those hazards and propose two different approaches to correct them.

The first approach has been to develop a code parser that rewrites the code generated by the compiler and avoids the instruction hazards. While it is a straightforward approach, this technique limits the processor performance by reducing the parallelism of the code. The impact on performance has been also evaluated in our work. The code parser has been developed in C++ to be executed in UNIX environments

The second approach was to add some hardware profiling registers to determine why these instruction hazards happen and allow further projects to do the needed reengineering of the architecture.

1.- Introducción

Procesadores VLIW

La proliferación de dispositivos portátiles multimedia ha revitalizado las arquitecturas VLIW (Very Long Instruction Word) que están siendo adoptadas por sistemas de muy altas prestaciones. Las Tecnologías de la Información y las Comunicaciones (TIC) demandan cada vez más recursos para poder procesar grandes volúmenes de datos. Hasta ahora, los avances en la capacidad de cómputo de los procesadores se han basado en el aumento de la velocidad del reloj y en innovaciones en la planificación, por parte del hardware, de la ejecución de instrucciones.

Este modelo actual presupone que cada nueva generación de procesadores es capaz de ejecutar más instrucciones y será difícil que las arquitecturas tradicionales continúen doblando prestaciones cada 12-18 meses sin que se emigre a una nueva tecnología. El número de instrucciones por ciclo aumenta y también lo hace el número de interdependencias entre instrucciones a comprobar para determinar qué instrucciones pueden ejecutarse de forma simultánea.

La lógica compleja requerida para la correcta planificación de instrucciones ocupa una gran parte del silicio del procesador y empieza a no tener sentido dedicar una gran parte de los recursos del procesador a la planificación de instrucciones.

En su lugar, parece tener más sentido utilizar ese silicio para poder ejecutar más instrucciones por ciclo, incorporando más unidades funcionales y aumentar así el paralelismo de ejecución, mientras que la planificación de instrucciones se realiza por el compilador. Éste es el principio en que se basa la arquitectura VLIW. Esta reducción de complejidad, hardware más sencillo y menor número de transistores, permite incrementar la velocidad del reloj y al mismo tiempo reducir el consumo.

Aplicaciones vectoriales y procesadores VLIW

Una de las finalidades de las aplicaciones vectoriales es el aprovechamiento del paralelismo de datos en las aplicaciones de procesamiento masivo de datos. Para estas aplicaciones de cálculo científico que demandan velocidades de cómputo elevadas, son muy útiles los procesadores VLIW puesto que la propia configuración del procesador está preparado para este tipo de aplicaciones.

La idea básica explotada por el procesamiento vectorial es combinar dos vectores, elemento a elemento, para producir una salida vectorial

$C=A+B$

Donde:

$ci=ai+bi \ , 0 \leq i < N$

Una instrucción vectorial codifica una gran cantidad de cálculos. De esta forma se reduce el número de instrucciones del los programas vectoriales con respecto a los programas que ejecutan los procesadores escalares.

Por ejemplo:

for(i =1; i<N; i++)

V1[i] = V2[i] + V3[i];

⇓

ADDV V1, V2, V3

Esta idea de codificar las operaciones tiene otras ventajas, puesto que también se reduce el número de accesos a memoria y evita ciertos problemas asociados al control de los bucles.

Para explotar este paradigma de programación es muy importante que el compilador aproveche al máximo el potencial de una aplicación para paralelizar al máximo las instrucciones y que sea muy eficiente generando código para un procesador VLIW determinado. Esta dependencia de compilador-procesador es a la vez su punto fuerte puesto que se reduce la lógica del procesador para lanzar las operaciones pero a la vez exige que el compilador se diseñe a la vez que el procesador, para aprovechar al máximo sus características. El problema puede surgir en que un compilador existente para un procesador VLIW determinado puede generar código ineficiente para una nueva generación de procesadores VLIW incluso si éstos tienen el mismo repertorio de instrucciones.

Lenguajes de alto nivel para el desarrollo de procesadores

Puesto que la complejidad de los procesadores aumenta, es necesario disponer de un lenguaje de alto nivel que permita realizar una descripción de la funcionalidad del procesador (qué queremos que haga) más que preocuparnos de cómo se implementa dicha funcionalidad.

Otra de las necesidades a cubrir por parte de los lenguajes de alto nivel es acelerar el ciclo de desarrollo–simulación para comprobar rápidamente si el procesador cumple las restricciones de diseño requeridas y su funcionalidad.

El hecho que sea fácil añadir y quitar funcionalidades al procesador puesto que los lenguajes de alto nivel nos ofrecen la posibilidad de un rápido prototipado, a la vez permite explorar soluciones tecnológicas con un bajo coste de desarrollo.

Prototipado rápido de procesadores VLIW

Para un rápido prototipado existen herramientas en las que podemos modificar varios parámetros de procesador fácilmente y así comprobar las diferencias en el rendimiento que dichos cambios producen.

Típicamente, las características que se pueden modificar son las siguientes:

Modelo de recursos hardware:

- Recursos: Establecer el número de recursos de la máquina, se puede usar, por ejemplo, para establecer la cantidad máxima de operaciones que dispone una instrucción VLIW.

- Latencia: Establecer la latencia de un elemento de la máquina, como añadir más retraso o no a una operación.

- Registros: Decidir el número de registros, tanto los de propósito general como de salto.

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.

1.1.- Ámbito de aplicación de procesadores VLIW

El término “Very Long Instruction Word” (o VLIW) se refiere a una arquitectura de CPU diseñada para aprovechar el paralelismo a nivel de instrucción (ILP). Un procesador que ejecuta una instrucción después de otra puede utilizar recursos del procesador de forma ineficiente, lo que podría dar lugar a un mal rendimiento. El rendimiento se puede mejorar ejecutando una instrucción en diferentes sub-etapas (esto es *pipelining*), o incluso ejecutar múltiples instrucciones simultáneamente como en las arquitecturas superescalares. Se pueden obtener mejoras mediante la ejecución de instrucciones en un orden diferente al orden en que aparecen en el programa, lo que se conoce como ejecución fuera de orden.

Estas tres técnicas tienen un coste: el incremento de la complejidad del hardware. Antes de ejecutar todas las operaciones en paralelo, el procesador debe verificar que las instrucciones no tienen interdependencias. Hay muchos tipos de interdependencias, pero un simple ejemplo de ello sería un programa en el que la primera instrucción del resultado se utiliza como operando para la segunda instrucción. Claramente estas dos instrucciones no pueden ejecutarse a la vez y la segunda instrucción no puede ser ejecutada antes que la primera. Los procesadores modernos que ejecutan instrucciones fuera de orden usan una significativa cantidad de recursos con el fin de

aprovechar las ventajas de estas técnicas, ya que el orden de las instrucciones debe determinarse dinámicamente.

En el enfoque VLIW, por otra parte, se ejecuta la operación en paralelo debido a que el paralelismo viene determinado cuando los programas son compilados. Puesto que el paralelismo viene determinado por el compilador, el procesador no necesita recursos hardware para reordenar las instrucciones que las tres técnicas descritas anteriormente requieren. Como consecuencia de todo esto, los procesadores VLIW ofrecen una elevada potencia de cálculo con menos complejidad hardware que la mayoría de CPU superescalares. La complejidad se traslada al compilador.

Historia

El término VLIW, así como el concepto de arquitectura VLIW en sí, fueron inventados por Josh Fisher y su grupo de investigación en la Universidad de Yale a principios de la década de 1980. Antes de VLIW, la noción de ordenación de las instrucciones a unidades funcionales y el paralelismo a nivel de software se desarrolla a nivel de microcódigo horizontal. En microcódigo horizontal cada microinstrucción contiene *n* bits, cada uno de los cuales controla directamente cada pieza de hardware, evitando otros niveles de decodificación. Las microinstrucciones horizontales son más largas. Las innovaciones de Fisher se centran en torno a la elaboración de un compilador que podría generar microcódigo horizontal a partir de un lenguaje de programación ordinario. Se dio cuenta de que con el fin de obtener buenos resultados, sería necesario encontrar el paralelismo más allá de lo que se encuentra dentro de un bloque de código. Desarrolló técnicas de ordenación de código más allá de los bloques de código básicos. La ordenación de traza es una técnica e implica la programación de la ruta más probable de los bloques de código más probables en primer lugar e insertar el código de compensación para hacer frente a los fallos especulativos, ordenar los segundos bloques de código más probables y así sucesivamente, hasta que se finalice el reordenamiento de todo el código.

La segunda innovación de Fisher fue la idea de que la arquitectura de una CPU debe ser diseñada para que un compilador pueda generar código para ella fácilmente. El compilador y la arquitectura VLIW deben ser co-diseñadas. Fisher ha desarrollado un conjunto de principios que caracterizan un buen diseño VLIW, por lo que se ha simplificado el proceso de generación de código rápido por parte de los compiladores.

El primer compilador VLIW se describió en la tesis doctoral de John Ellis, bajo la supervisión de Fisher. John Ruttenberg también ha desarrollado importantes algoritmos para la reordenación de instrucciones.

Diseño

En los diseños superescalares, el número de unidades de ejecución es invisible para el conjunto de instrucciones. Cada instrucción codifica una sola operación. Para la mayoría de los diseños superescalares, la instrucción es de 32 bits o menos. Por el contrario, una instrucción VLIW codifica múltiples operaciones; concretamente, una instrucción codifica al menos una operación para cada unidad de ejecución del procesador. Por ejemplo, si un procesador VLIW tiene cinco unidades de ejecución, entonces una instrucción VLIW para ese procesador tendría cinco campos donde cada campo especifica qué instrucción debe de ejecutarse en cada una de las unidades de ejecución. Para poder realizar esta función, las instrucciones VLIW son normalmente de 64 bits como mínimo pero en algunas arquitecturas son mucho más grandes.

El siguiente ejemplo es para un procesador SHARC. En un ciclo, realiza una multiplicación en punto flotante, una suma en punto flotante, y dos accesos a memoria. Todo esto en una sola instrucción de 48 bits.

```
f12 = f0 * f4, f8 = f8 + f12, f0 = dm (i0, m3), f4 = pm (i8, M9);
```

Desde el inicio de la arquitectura de computadores, algunas CPU han añadido UAL (unidad aritmética lógica) para la ejecución en paralelo. Las CPU superescalares usan lógica hardware para decidir qué operaciones pueden ejecutarse en paralelo. Como ya hemos aclarado antes, en las CPU VLIW es el compilador el que decide el orden de ejecución y la complejidad del hardware se reduce considerablemente.

Un problema similar se produce cuando el resultado de una instrucción se usa en una instrucción de salto. La mayoría de las CPUs modernas pueden "adivinar" qué salto se realizará antes incluso de que el cálculo se complete, a fin de que puedan ejecutar las instrucciones del salto de forma especulativa. Si la CPU no acierta con el salto, todas estas instrucciones y su contexto deben "vaciarse" y cargar las instrucciones correctas. Este proceso es costoso en cuanto a tiempo.

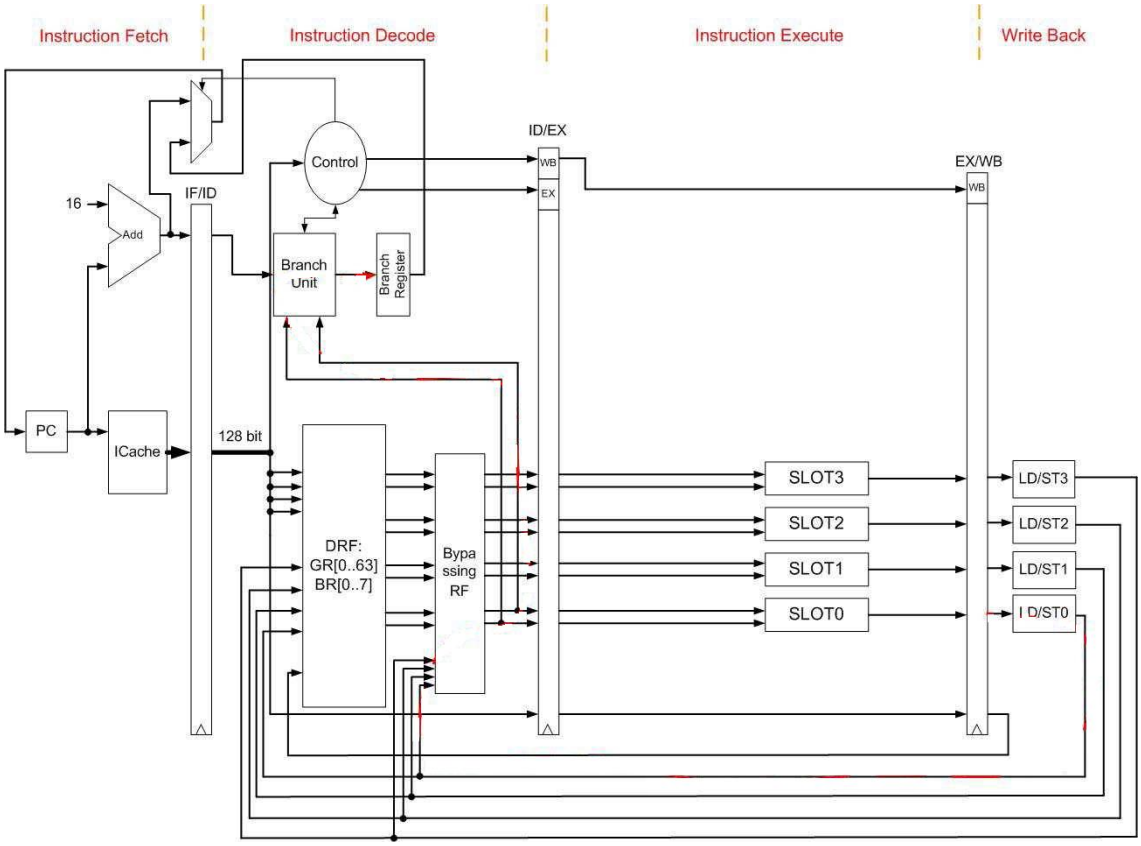
Esto ha hecho que cada vez se hagan instrucciones que tratan de adivinar mejor los saltos y se ha perdido la simplicidad inicial de los procesadores RISC. Puesto que los procesadores VLIW carecen de esta lógica, consumen menos energía y evitan posibles errores en el diseño del procesador además de otras características negativas.

En un procesador VLIW, el compilador utiliza heurística o información sobre el perfil de adivinar la dirección de salto. Esto le permite mover y reordenar de forma especulativa antes de realizar el salto. Si el destino del salto no es el esperado, el compilador ya ha generado el código para descartar resultados especulativos, a fin de preservar la integridad semántica del programa.

De todo lo anteriormente dicho deducimos que el compilador, para poder realizar su tarea, necesita conocer todas las características del hardware, tales como: número, tipo, latencia, intervalo de iniciación de las unidades funcionales, muchos de los cuales

dependen no sólo de la arquitectura sino de la tecnología de implementación. Por tanto, un mismo compilador NO puede utilizarse para distintos modelos de la misma familia.

A continuación se muestra un esquema de un procesador VLIW VEX



Implementaciones

Las CPU VLIW suelen estar constituidas de varias unidades funcionales (clusters) de tipo RISC que funcionan de forma independiente. Las VLIW actuales suelen tener entre cuatro y ocho unidades funcionales. Los compiladores generan código de una manera parecida a la de los compiladores que generan código para las CPU tradicionales. El compilador analiza el código generado buscando relaciones de dependencia y los recursos necesarios. A continuación, reordena las instrucciones en función de esas limitaciones. En este proceso, las instrucciones independientes entre sí se pueden paralelizar.

Algunos ejemplos actuales de CPU VLIW son los procesadores multimedia TriMedia de NXP (anteriormente Philips Semiconductors), los DSP SHARC de Analog Devices, la familia C6000 DSP de Texas Instruments, STMicroelectronics y el ST200 familia basada en la arquitectura Lx (también diseñado por Josh Fisher) . Estos procesadores

VLIW se usan como procesadores para sistemas empotrados en dispositivos multimedia para el mercado de consumo.

Las propiedades de los procesadores VLIW también se han incorporado a núcleos de proceso configurables. Por ejemplo, el procesador Xtensa LX2 de Tensilica incorpora una tecnología apodada FLIX (Flexible Longitud Instrucción eXtensions) que permite múltiples operaciones en una instrucción. El compilador C/C++ de Xtensa puede mezclar instrucciones de 32 o de 64 bits en una sola operación para el procesador FLIX.

Fuera del mercado de procesadores para sistemas empotrados, el Intel Itanium IA-64 EPIC aparece como el único ejemplo de procesador ampliamente utilizado en arquitectura VLIW. Sin embargo, la arquitectura EPIC raras veces se distingue de una mera arquitectura VLIW, ya que la arquitectura EPIC difiere ligeramente de una arquitectura VLIW pura.

Compatibilidad con procesadores anteriores

Cuando la tecnología permitió implementar más *clusters* (unidades de ejecución), los programas compilados para la generación anterior no se ejecutaban en la nueva configuración, puesto que el código generado hace referencia al número de *clusters* del procesador.

La compañía de microprocesadores Transmeta aborda esta cuestión mediante la inclusión de una capa software que traduce de código máquina determinado a código máquina binario del procesador (esto se llama *Code Morphing*) en su procesador Crusoe con la arquitectura x86. Básicamente, este mecanismo recompila, optimiza y traduce instrucciones x86 en tiempo de ejecución en la CPU. De este modo, el procesador de Transmeta es internamente un procesador VLIW, y desvincula la arquitectura del procesador de las instrucciones x86 que ejecuta.

La arquitectura Intel Itanium (entre otras) ha resuelto el problema de la compatibilidad hacia atrás con un mecanismo de carácter más general. Dentro de cada una de las tuplas con múltiples códigos de instrucción, existe un bit que denota la dependencia con la instrucción anterior. Estos bits se establecen en tiempo de compilación, por tanto liberan al hardware de realizar esta tarea en tiempo de ejecución.

Otra deficiencia de estas arquitecturas es cuando las instrucciones no lanzan operaciones para cada uno de los *clusters* y éstas ejecutan instrucciones de tipo NOP. Para esto es necesario permitir el vaciado de los pipelines de forma que se pueda lanzar la siguiente instrucción.

Dado que el número de transistores en un procesador ha crecido, los inconvenientes asociados a los procesadores VLIW han disminuido en importancia. La arquitectura VLIW se está haciendo cada vez más popular, sobre todo en el mercado de los

sistemas empotrados, donde es posible personalizar un procesador para realizar una tarea concreta.

En los últimos tiempos, los sistemas inalámbricos y teléfonos móviles además de otros sistemas multimedia demandan más capacidad de proceso de los DSP (*Digital Signal Processors*). Simultáneamente, los requerimientos de voltaje y consumo necesitan ser cada vez más bajos, puesto que se desea aumentar la autonomía de dichos dispositivos.

Para cubrir las insaciables necesidades de cálculo de dichos sistemas, donde voz, vídeo, audio y datos convergen es necesario incrementar drásticamente el rendimiento de dichos procesadores. Los niveles actuales de rendimiento de unos pocos centenares de millones de instrucciones por segundo (MIPS) o de unos cuantos centenares de millones de instrucciones de coma flotante por segundo (MFLOPS) ya no cubren las necesidades actuales. Las aplicaciones futuras necesitan un incremento de potencia de un orden de magnitud.

No es sorprendente que los fabricantes de DSP hayan lanzado una nueva generación de procesadores basados en tecnología VLIW combinados con lo mejor de la tecnología superescalar.

Otra de las características de los nuevos procesadores es que han tenido en cuenta a los compiladores y dichos compiladores están evolucionando para adaptarse a cada procesador. Además las nuevas arquitecturas mantienen la compatibilidad hacia atrás con los antiguos procesadores, permitiendo a los desarrolladores mantener el software generado para los DSP anteriores.

1.2.- Arquitectura VEX

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. Esta configuración variable del procesador se realiza mediante un fichero de parámetros donde se indican las características de la arquitectura. Dicha arquitectura está 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 (ciclos de reloj) que la inicialmente asumida, el hardware debe mantener en pausa la ejecución hasta que se pueda volver a reiniciar otra vez. Las esperas suelen venir a causa de fallos de cache y de saltos.

Arquitectura compuesta de clusters

Un *cluster* es un conjunto de registros y unidades funcionales acoplados. Permite el acceso múltiple a memoria. Dentro un *cluster* los registros y las unidades funcionales están completamente conectados entre si, pero las unidades no están conectadas entre si, permitiendo que el resultado de una unidad funcional se pueda dejar en el mismo *cluster* sin necesidad de dejarlo en un medio de almacenaje exterior.

Estructura de un cluster por defecto

Contiene unidades funcionales de memoria, operaciones sobre enteros, y salto

- Las unidades de memoria: permiten la carga, almacenamiento y operaciones de pre-fetch

- 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 sólo 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 de un cluster están principalmente compuestas de:

- Cuatro *Arithmetic Logic Unit* (ALU) para enteros con latencia de 1 ciclo.
- Dos multiplicadores de latencia de 2 ciclos.
- 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 32 bits 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.

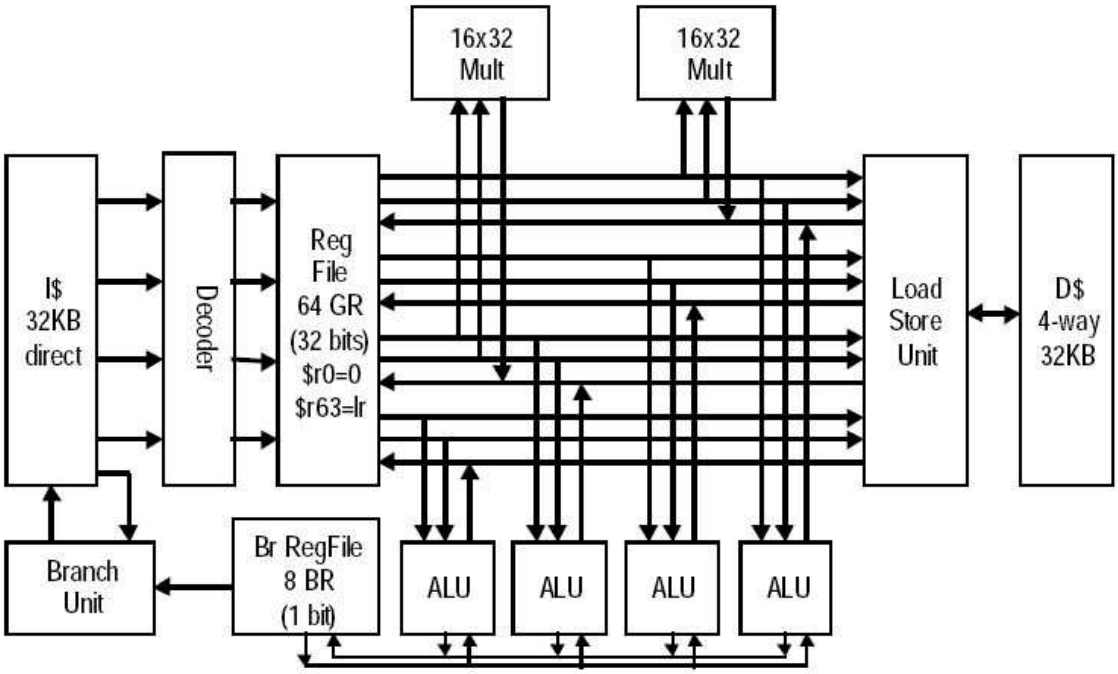
Aritmética y operaciones lógicas

VEX soporta el mismo formato de operaciones aritméticas que las soportadas por los microprocesadores RISC (*Reduced Instruction Set Computer*), 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 resultado será el de un procesador sin hardware para punto flotante.

A continuación se muestra la estructura de un cluster VEX



1.3.- Necesidad de entornos de descripción hardware de alto nivel

Nos referimos con lenguaje de descripción de hardware HDL a cualquiera de los lenguajes de programación para la descripción formal de los circuitos electrónicos. Se puede describir el funcionamiento del circuito, su diseño y organización y posteriormente realizar las simulaciones pertinentes para verificar su funcionamiento.

Los HDL se basan en expresiones para describir el comportamiento funcional y temporal de los sistemas electrónicos. En contraste con un lenguaje de programación de software, los HDL usan una sintaxis y semántica de las notaciones explícita para expresar el tiempo y la concurrencia, que son los principales atributos de hardware. Los lenguajes cuya única característica es expresar circuito de conectividad entre una jerarquía de bloques están debidamente clasificados como lenguajes *netlist* que se utilizan para el diseño CAD eléctrico.

Es posible representar una sintaxis hardware utilizando lenguajes de programación tradicionales como C++, aunque deberíamos aumentar la funcionalidad de estos lenguajes con una amplia bibliotecas de clases. Sin embargo, los lenguajes de programación de software no incluyen ninguna capacidad para expresar explícitamente el tiempo, y ésta es la razón por la que no funcionan como un lenguaje de descripción de hardware.

Una vez descrito el circuito (usando un lenguaje de programación HDL) un software llamado sintetizador de hardware puede inferir la lógica del circuito a partir de las instrucciones del programa y producir una *netlist* equivalente genérica hardware a partir de primitivas. En este punto no se tienen en cuenta las instrucciones referidas a restricciones temporales contenidas en la sintaxis del programa.

El Diseño de un sistema en HDL es generalmente mucho más difícil y consume más tiempo que escribir un programa de software para hacer la misma cosa. En consecuencia, se han realizado muchos esfuerzos para la conversión automática de código C a código HDL, pero aún no se ha alcanzado un nivel de calidad comercial.

LISATek

LISATek es un entorno de desarrollo, modelado y generación de herramientas software para procesadores empotrados. Dicho entorno está construido entorno al lenguaje LISA. A partir de una descripción de alto nivel del procesador, es posible generar automáticamente el procesador, el juego de pruebas y las herramientas de desarrollo ensamblador. En la última versión de dicho entorno incluso se genera automáticamente un compilador C para el procesador diseñado.

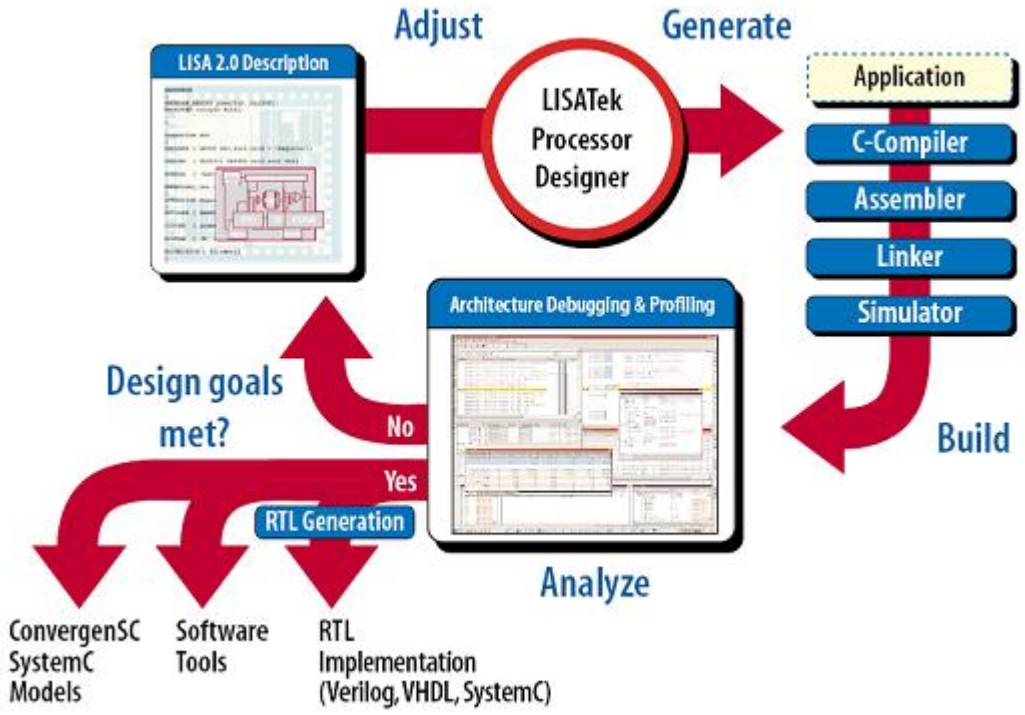
En caso que se esté diseñando de procesadores *custom* (como el caso de los ASIP) para el proceso de señales y aplicaciones de control, también se genera automáticamente el código RTL.

Cada vez más, las compañías tienen sus propios diseños, que además necesitan reutilizar para otros proyectos.

El diseñador de procesadores LISATek permite diseñar sistemas programables que puedan ser reutilizados. Así pues, es posible empezar a describir un nuevo procesador empezando por el repertorio de instrucciones y la arquitectura del procesador y de esta forma ahorrar mucho tiempo de diseño.

Otra de las ventajas del entorno de desarrollo LISATek es que cualquier cambio en la arquitectura o en el repertorio de instrucciones se refleja en las herramientas de desarrollo generadas automáticamente.

Una vez se ha finalizado el diseño del procesador, se genera el código sintetizable RTL.



En el diagrama anterior, se muestra el flujo de desarrollo de un procesador en el entorno de desarrollo LISATek.

A partir de la descripción en alto nivel de procesador mediante el lenguaje LISA, se ajusta el diseño tantas veces como sea necesario hasta que se obtiene un procesador que cumple con los requisitos de diseño. A partir del diseño del procesador, se generan automáticamente las herramientas de desarrollo de aplicaciones para el procesador. Dichas herramientas incluyen un compilador C un compilador de lenguaje ensamblador así como un *Linker*, un simulador y un desensamblador.

Si el procesador funciona correctamente, se genera automáticamente el RTL sintetizable, las herramientas de desarrollo y los modelos SystemC, VLIW y Verilog.

1.4.- Objetivos del proyecto

Entre los objetivos del proyecto está inicialmente familiarizarse con la descripción hardware de un procesador basado en arquitectura VEX. Para adquirir este conocimiento ha sido necesario documentarse acerca de la arquitectura VEX y su implementación en la arquitectura “de referencia”.

En cuanto al entorno de desarrollo se ha necesitado aprender el ciclo de codificación, compilación y depuración. Además se han incorporado trazas en el código de forma que se pueda conocer el valor de los registros de *profiling* en un momento determinado.

También se ha ejecutado la arquitectura en el depurador y se ha comprobado el funcionamiento de los programas desarrollados para dicha arquitectura. Esto también ha servido para detectar los errores de diseño de la arquitectura de referencia.

Todo este conocimiento es necesario para la implementación de un mecanismo de *hardware profiling* básico consistente en un registro que cuenta los ciclos de reloj que han pasado desde el inicio del programa. De esta forma y mediante unas trazas en el código del procesador visibles desde el depurador hacen que se pueda determinar el tiempo que tardan las unidades funcionales en devolver el resultado de un cálculo determinado. Dicha información es fundamental para determinar el origen de los *hazards* y en un futuro solucionarlos a nivel de procesador.

Aunque la mejor forma de solucionar los *hazards* es solventando las deficiencias en el diseño del procesador, es más fácil crear una aplicación que traduzca las aplicaciones para la arquitectura VEX y los adapte a la arquitectura de referencia de forma que se eviten los *hazards* detectados.

Así pues, uno de los objetivos de la aplicación es crear dicha aplicación que genere código para la arquitectura de referencia. Este proceso añade ineficiencia en el código y uno de los objetivos del proyecto es medir la penalización en el rendimiento a consecuencia de dicha traducción de instrucciones.

Para evaluar la pérdida media de rendimiento del procesador se usan varios *benchmark* para evaluar el rendimiento en aspectos como el cálculo intensivo o el manejo del direccionamiento de memoria. Además de los *benchmarks* específicos evalúa el rendimiento del procesador en aplicaciones clásicas como las torres de Hanoi, el producto de matrices o la ordenación por el método de la burbuja.

2.- Herramientas

La familia de herramientas LISATek es un entorno de diseño y optimización automatizado para procesadores de sistemas empotrados que reduce drásticamente el tiempo de diseño de procesadores estándar y custom. Estos procesadores son desde ASIP (*Application-Specific Instruction-set Processors*) que están convergiendo en cuanto a funcionalidad hacia los SoC (*System-on-a-Chip*).

Esta metodología de diseño se basa en el lenguaje de descripción de procesadores LISA 2.0 que se utiliza para crear un modelo abstracto del procesador que se quiere diseñar.

Sobre la base de este modelo de arquitectura, se realizan los siguientes pasos en el proceso de desarrollo. La familia completa de productos incluye un compilador de C, una ensamblador de macros, un ensamblador, un enlazador, un simulador del procesador y un entorno gráfico para depurar sistemas de un procesador y sistemas multiprocesador con el fin de controlar el proceso creación del procesador.

Además de esto, el generador de procesadores genera automáticamente el código personalizado para el registro *Register-Transfer-Level* (RTL) de la arquitectura. Todo el proceso de generación puede ser gestionado desde la herramienta ***processor Designer***, que permite un fácil mantenimiento y síntesis del procesador.

A continuación se introducen brevemente las herramientas de desarrollo LISATek.

1. ***Processor Designer*** para diseñar y generar el procesador.
2. ***Processor Debugger*** para depurar y evaluar las prestaciones del procesador diseñado.
3. ***Lenguaje LISA*** para describir en un lenguaje de alto nivel la funcionalidad requerida del procesador.

2.1.- Diseño de bloques hardware (Processor designer)

El *Processor Designer* es una herramienta potente para el diseño y optimización tanto de los procesadores *custom* como para procesadores empotrados. Desde este entorno se llama a otras herramientas de LISATek para diseñar un modelo de procesador, analizar, optimizar su rendimiento, generar las aplicaciones para él y luego crear el código RTL.

Lo más importante de esta herramienta es el diseño del procesador. Dicho diseño se hace con un lenguaje LISA, que permite describir con bastante facilidad la arquitectura del procesador así como los recursos hardware características, tales como registros, pipelines, memoria, instrucciones, etc.

Para iniciar la aplicación ***processor designer***, ejecutar desde la línea de comandos el comando:

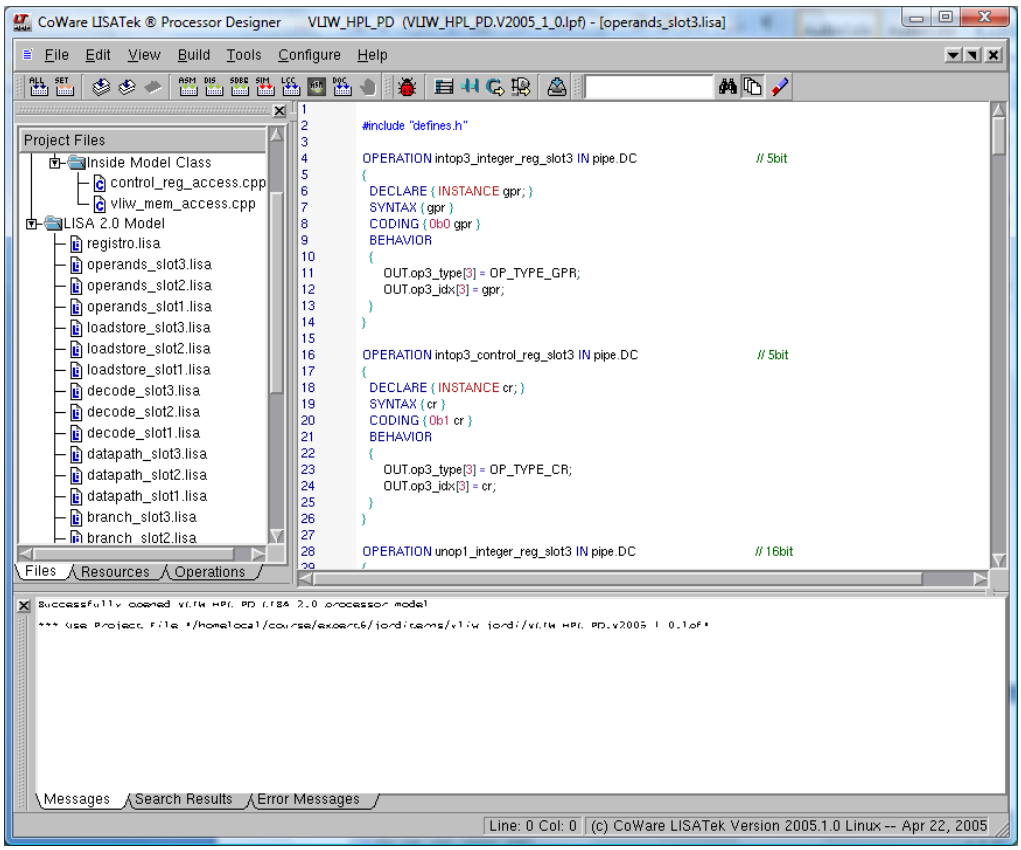
```
ldesigner &
```

Una vez cargado el entorno gráfico, se creará un proyecto para diseñar un procesador. Desde dicho entorno se describe la funcionalidad del procesador, su repertorio de instrucciones y los recursos hardware de dicho procesador.

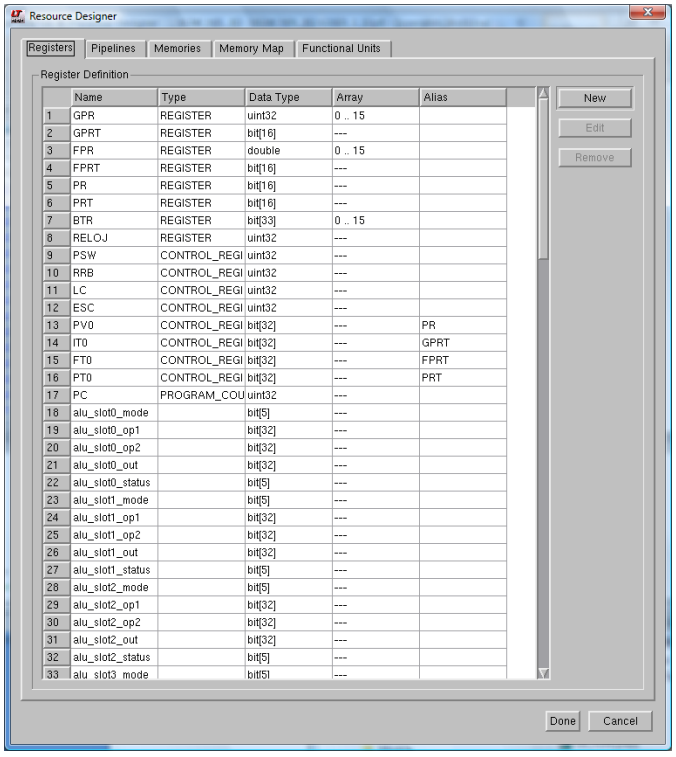
Es también desde entorno que realizaremos la compilación y depuración del procesador. La compilación del procesador genera automáticamente un compilador y un *linker* para el lenguaje ensamblador que acepta el procesador generado.

El proceso de desarrollo es parecido al de un compilador típico de un lenguaje de programación, pero para diseñar un procesador es necesario usar herramientas específicas para tal fin. El lenguaje con el que se realiza la descripción y especificación del procesador es el lenguaje LISA. Las características de dicho lenguaje serán descritas más adelante.

A continuación se muestra una captura de pantalla del entorno de desarrollo.

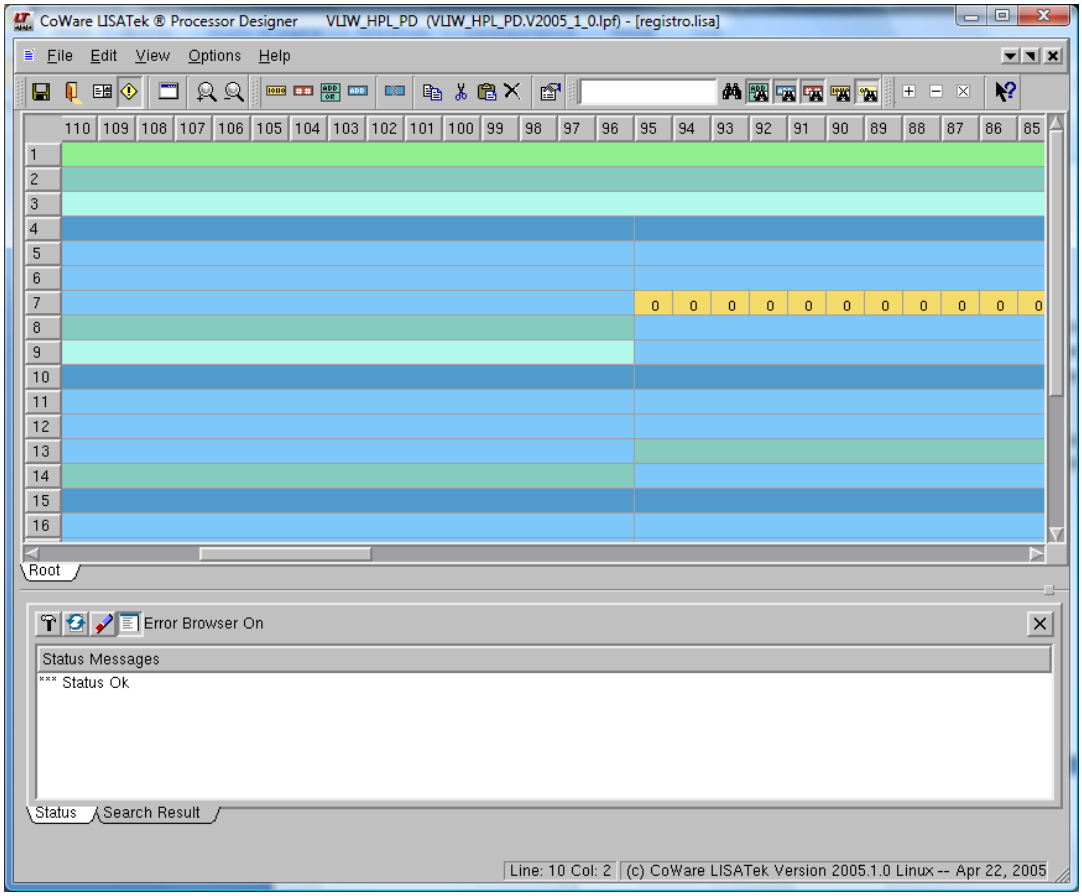


La siguiente captura de pantalla es del diseñador de recursos. Desde este cuadro de diálogo es posible añadir, modificar y eliminar registros, pipelines, unidades funcionales y recursos de memoria del procesador.



	Name	Type	Data Type	Array	Alias
1	GPR	REGISTER	uint32	0 .. 15	
2	GPRT	REGISTER	bit[16]	---	
3	FPR	REGISTER	double	0 .. 15	
4	FPRT	REGISTER	bit[16]	---	
5	PR	REGISTER	bit[16]	---	
6	PRT	REGISTER	bit[16]	---	
7	BTR	REGISTER	bit[33]	0 .. 15	
8	RELOJ	REGISTER	uint32	---	
9	PSW	CONTROL_REGI	uint32	---	
10	RRB	CONTROL_REGI	uint32	---	
11	LC	CONTROL_REGI	uint32	---	
12	ESC	CONTROL_REGI	uint32	---	
13	PV0	CONTROL_REGI	bit[32]	---	PR
14	IT0	CONTROL_REGI	bit[32]	---	GPRT
15	FT0	CONTROL_REGI	bit[32]	---	FPRT
16	PT0	CONTROL_REGI	bit[32]	---	PRT
17	PC	PROGRAM_COU	uint32	---	
18	alu_slot0_mode		bit[5]	---	
19	alu_slot0_op1		bit[32]	---	
20	alu_slot0_op2		bit[32]	---	
21	alu_slot0_out		bit[32]	---	
22	alu_slot0_status		bit[5]	---	
23	alu_slot1_mode		bit[5]	---	
24	alu_slot1_op1		bit[32]	---	
25	alu_slot1_op2		bit[32]	---	
26	alu_slot1_out		bit[32]	---	
27	alu_slot1_status		bit[5]	---	
28	alu_slot2_mode		bit[5]	---	
29	alu_slot2_op1		bit[32]	---	
30	alu_slot2_op2		bit[32]	---	
31	alu_slot2_out		bit[32]	---	
32	alu_slot2_status		bit[5]	---	
33	alu_slot3_mode		bit[5]	---	

A continuación se muestra el diseñador de instrucciones para el procesador.



2.2.- Generación de soporte software

Una vez se ha diseñado el procesador y depurado los errores de compilación del modelo, es posible generar herramientas para el desarrollo de aplicaciones para el procesador que se ha diseñado. Las herramientas básicas para el desarrollo son un compilador y un *linker* para poder generar un ejecutable.

Esto permite validar que el programa cumple la sintaxis del repertorio de instrucciones del procesador, pero no permite validar que el funcionamiento de la aplicación es correcto. Para validar el correcto funcionamiento del procesador así como evaluar su rendimiento es necesario disponer de un *debugger*.

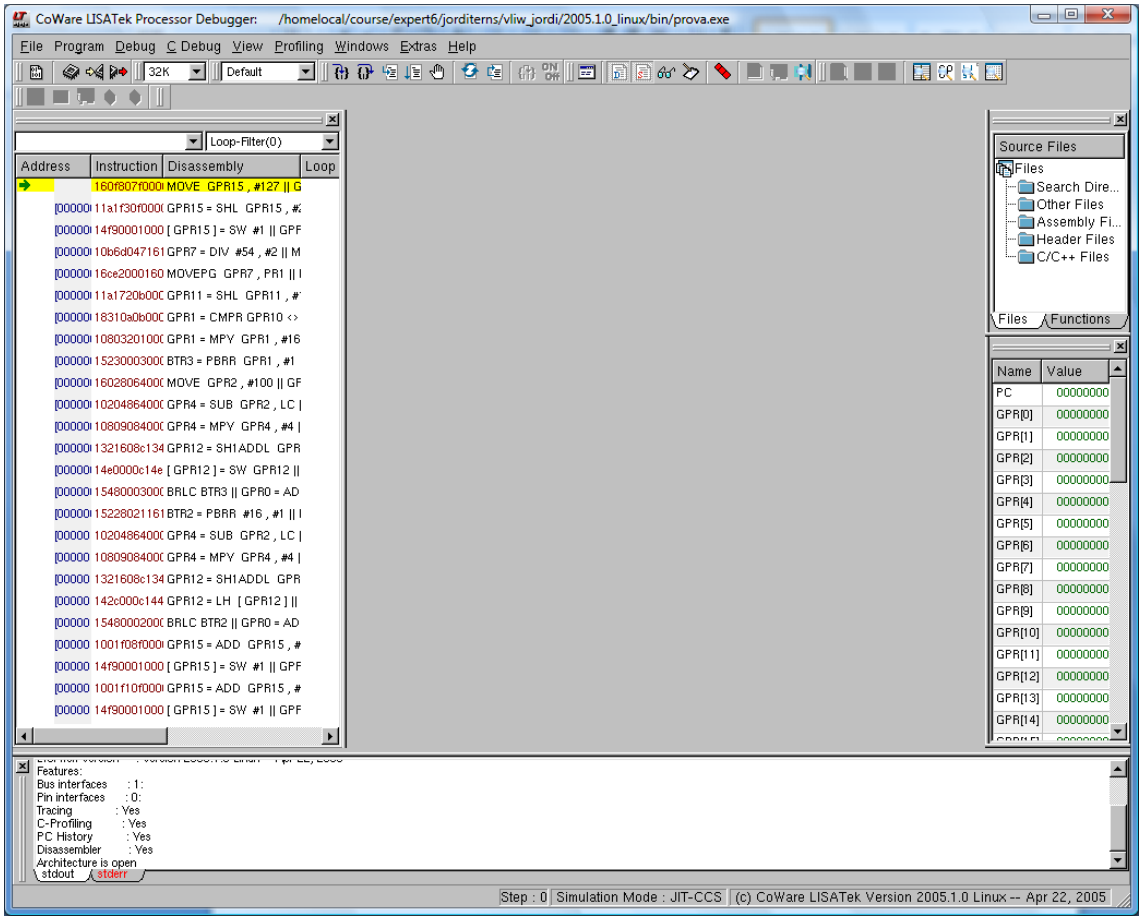
El depurador de procesadores

Además de las herramientas descritas anteriormente es necesario disponer de un depurador para que se pueda ver el contenido de los registros del procesador paso a paso. Para poder realizar una depuración del programa y ver su comportamiento en el procesador, es necesario ejecutar el depurador de procesadores. Para ello, ejecutamos el comando:

```
ldb &
```

Una vez cargado el entorno, es necesario cargar primero la arquitectura del procesador que se desarrollado y una vez cargada dicha arquitectura, se carga el programa que se quiere depurar o ver su comportamiento al ejecutarse sobre dicha arquitectura.

La aplicación permite ver el valor de los registros y la línea de código que se está ejecutando en este momento. La ejecución del programa puede realizarse paso a paso o dejar que se ejecute en su totalidad. Los mensajes de depuración introducidos en el diseño del procesador, se muestran cuando el programa ejecuta la instrucción donde están definidos.



2.3.- Lenguaje de especificación LISA

Las arquitecturas LISA de describen a partir de dos componentes básicos:

Recursos: Los elementos hardware del procesador tales como registros, memoria, pipelines, etc.

Operaciones: La descripción del comportamiento del procesador en relación a los recursos hardware que disponemos y el repertorio de instrucciones que ejecuta el procesador.

2.1.2.- La sección RESOURCE

En la sección de recursos se listan todos los elementos hardware necesarios para el modelo de memoria y demás recursos. Las declaraciones de dichos objetos tienen una sintaxis parecida a la declaración de variable en el lenguaje C y los datos almacenados en dichos recursos son tratados como variables en C. Los recursos no pueden ser declarados como una variable de una operación y por tanto, todos los recursos son visibles por todas las operaciones del procesador. Los recursos que se pueden definir en la sección RESOURCE son:

Recursos simples como:

- Registros y archivos de registro
- Señales
- Flags
- Arrays de memoria
- Otros recursos que no forman parte del estado visible de la arquitectura

Estructuras de Pipeline para instrucciones y buses.

Registros del Pipeline para pasar datos de una etapa a otra.

Memorias no lineales como:

- Cachés
- RAMs
- Los buses (como parte del subsistema de memoria)

Mapas de memoria que describen:

- Mapeos de memoria en el espacio de direcciones del procesador .
- La conexión entre la memoria y buses de datos

MEMORY MAP

En la sección MEMORY MAP Se puede especificar la forma en que las direcciones virtuales tienen que ser mapeadas a los recursos de hardware en el procesador.

Para explicar esto consideramos el siguiente ejemplo:

```
MEMORY _MAP
{BUS (sharebus),RANGE (0x0,0x2fffff) -> data_mem [(31:: 2)];}
```

Aquí se especifica que la memoria data_mem está conectada al bus sharebus, que las direcciones virtuales entre 0x0 y 0x2fffff están mapeadas en la memoria data_mem.
Los dos últimos 2 bits (de 32) en las direcciones no se tienen en cuenta para determinar qué palabra de la memoria será accedida. Se utilizan para calcular el offset (desplazamiento).

MEMORY

En las sección MEMORY instancia una memoria ideal y un rango de direcciones, aunque esta memoria no exista. Este tipo de memoria se puede acceder de la misma forma que a un vector. Por ejemplo, en esta instrucción:

```
mem = datos [b]
```

El contenido de la dirección b de la memoria se copia a la variable mem. No se calculan las latencias, debido puesto que en realidad no se transfieren datos.

BUS, RAM, CACHE

Con estas palabras clave se pueden instanciar como componentes reales, al igual que los buses, memorias y memorias cache, pudiéndose crear una jerarquía de memoria en caso que sea necesario.

Un bus puede ser instanciado con la palabra clave BUS. Cuando el modelo de procesador diseñado en LISA está compilado, se genera automáticamente una interface de bus estándar.

Esto implica que todos los núcleos de LISA utilizar la misma interfaz de manera que es posible crear interfaces para diferentes procesadores LISA.

Se pueden especificar varios parámetros al instanciar cualquiera de estos componentes:

```
RAM Uint32 data_mem
{SIZE (0x300000); ENDIANESS (BIG);
BLOCKSIZE (32); FLAGS(W|R|X);
PORT (READ = 1 OR WRITE = 1); };
```

Esta memoria RAM (es decir, una verdadera memoria) es de 32 bits sin signo con un tamaño de bloque de 32 bits. Los datos se almacenan como big-endian y la memoria es de escritura-lectura-ejecución.

El tamaño de la memoria es 0x300000 bloques, que es 0xc00000 bytes (12MB), ya que en cada bloque es de 4 bytes (0x300000 *4).

Cuenta con un puerto para leer y escribir que tiene que ser fijado para la generación de hardware. También se pueden establecer otros parámetros, como las latencias y tamaños de página entre otros.

REGISTER, PROGRAM COUNTER

Dentro de la sección RESOURCE, es posible instanciar los registros internos, registros mapeados en memoria y tipos específicos de registros como el Contador de Programa (PC). Con la palabra clave REGISTER es sencillo declarar todos los registros que son usados por el procesador así como el tipo de datos que almacenan e incluso el ancho del bus.

En el siguiente ejemplo:

```
TClocked PROGRAM_COUNTER Clocked<Uint32> FPC;
REGISTER TClocked <Uint32> GR [0..63];
```

El registro FPC es el contador de programa del procesador (con un tamaño de 32 bits). En la siguiente línea se ha declarado un conjunto de 64 registros de 32 bits cada uno. La palabra clave TClocked especifica que estos registros son síncronos con el reloj del procesador.

Para declarar registros mapeados en se usará la palabra clave ALIAS:

```
REGISTER Uint32 RTCDR ALIAS sharebus [0x80000380];
```

El registro de 32 bits RTCDR se mapeará en el bus *sharebus* en la dirección 0x80000380. Cualquier acceso a este registro se traduce en un acceso al bus en esa dirección. Esta posibilidad no está permitida para la generación de hardware, o simplemente el *Processor Generator* lo ignora porque un alias no necesita una implementación hardware que ya existe para el recurso al que hemos hecho el alias.

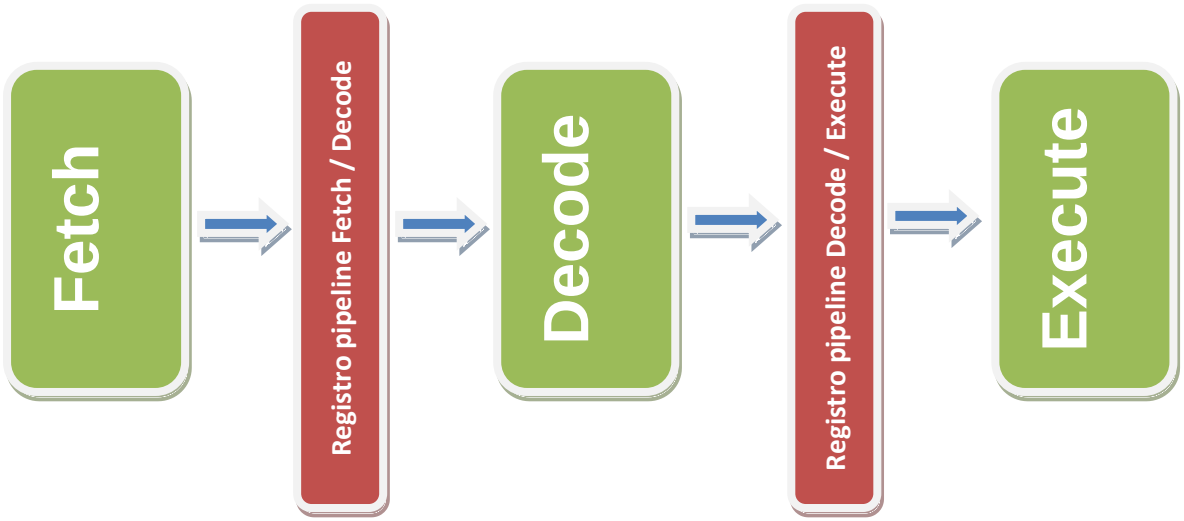
PIPELINE

Diseñar un procesador con Pipeline en LISA se realiza mediante la palabra clave PIPELINE y PIPELINE REGISTER. Se declara el Pipeline y cuantas etapas tiene. Por último se declaran los registros que hay entre las etapas del pipeline.

```
PIPELINE pipe = FE; DC; EX;
PIPELINE_REGISTER IN pipe
Uint16 pc; unsigned bit [32] iWord;
```

Este pipeline tiene tres etapas: búsqueda, decodificación y ejecución.

Entre cada etapa hay registro de 16 bits que contiene el valor del contador del programa y un registro de 32 bits que contiene la instrucción de una etapa en concreto.



Los registros del pipeline se ejecutan automáticamente (ejecutando la operación **main** y todas las cadenas de activación) y se desplazan en cada ciclo con los comandos *execute* y *shift*.

```
PIPELINE (pipe).execute();
PIPELINE (pipe).shift();
```

Otros comandos especiales para los registros del pipeline son ***.flush()*** y ***.stall(),*** que resetean e insertan un vacío en la cadena del pipeline.

PIN

Es posible declarar PIN adicionales con capacidad asíncrona. Esto es especialmente útil en la simulación multi-procesador. A continuación se muestra un ejemplo de declaración de un PIN:

```
PIN IN bool nFIQ;
PIN IN signed bit[16] Din;
PIN OUT signed bit [16] Dout;
```

PIN puede ser utilizado para manejar interrupciones u otros eventos asíncronos.

2.1.3.- Modelado la funcionalidad del procesador: operaciones LISATek

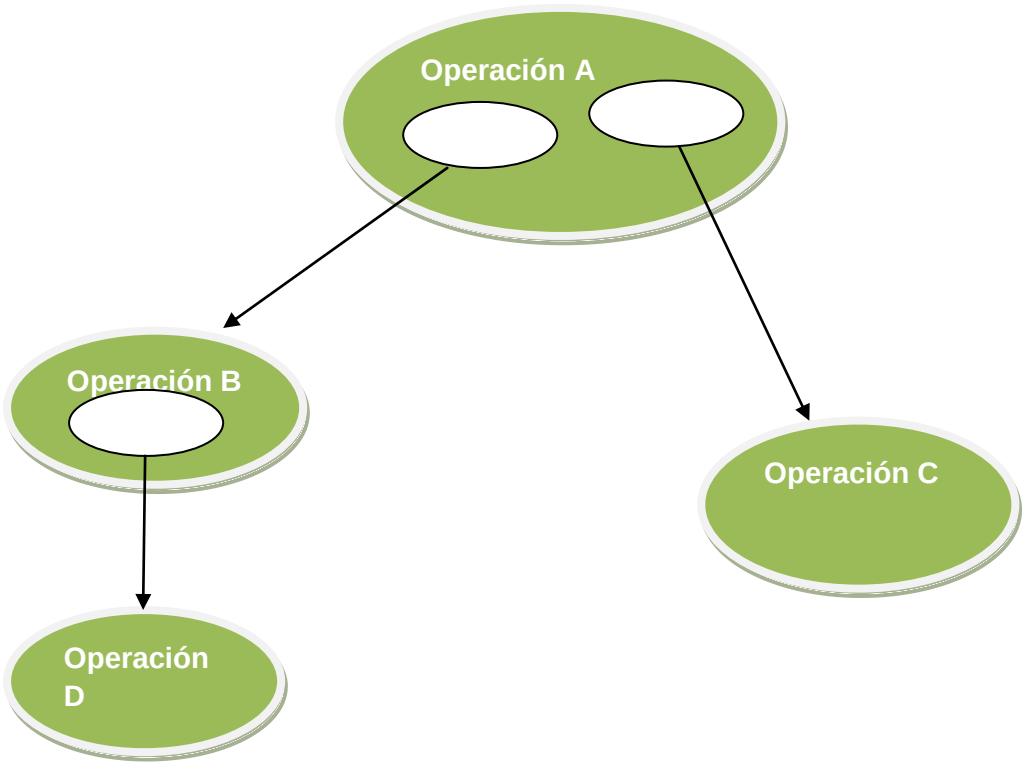
Casi todas las funciones de procesador de LISATek se basan en la palabra clave OPERATION. Esta palabra se refiere a una instrucción, un conjunto de instrucciones similares, una parte de una misma instrucción (como la fase *fetch* y *decode*) un registro único y así sucesivamente.

Hay dos operaciones que tienen que estar obligatoriamente en cualquier modelo LISATek.

OPERATION reset se invoca al inicio de la simulación: Inicializa los registros y el PC y vacía el pipeline

OPERATION main se invoca en cada ciclo de reloj. Esta operación realiza dos tipos de acciones: Execute y shift. Además, la próxima operación que se activará, se declara como una operación fetch.

Las operaciones LISATek se estructuran en una forma jerárquica. Cualquier operación no terminal de un nivel superior puede hacer referencia a otra operación en un nivel inferior. Por ejemplo, las operaciones de alto nivel A y B se completan invocando las operaciones de bajo nivel C y D. Esta idea de hacer referencia entre las operaciones es similar a la idea de llamar a funciones en C / C + +: esto permite que dos diferentes operaciones puedan llamar a la misma operación de nivel inferior. Una diferencia importante es que en LISATek se puede dar el caso contrario: una función de alto nivel puede referenciar a diferentes operaciones de bajo nivel de un conjunto dado.



La sección DECLARE

En la sección DECLARE hay las declaraciones de identificadores, los grupos de operaciones y las referencias a otras operaciones que serán utilizados en el interior de la operación actual. Los grupos de operaciones declaran con la palabra clave GROUP. Las operaciones en el mismo grupo puede ser utilizado en el mismo contexto. La elección entre ellos no puedan hacerse en tiempo de diseño, solo en tiempo de compilación.

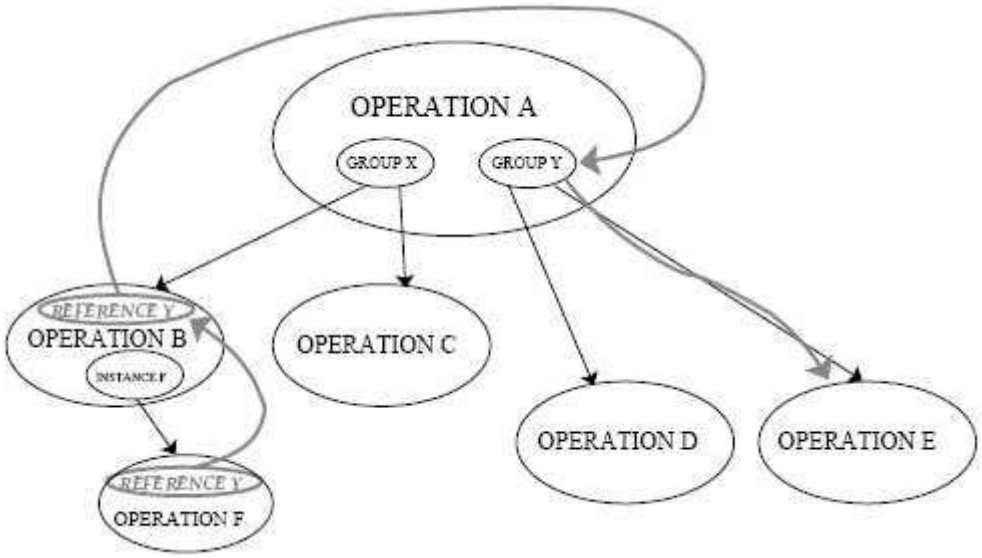
```
GROUP Instrucction = {instr_single_slot||instr_mult_slots};
```

El ejemplo anterior es un caso típico de grupo de operaciones: los procesadores VLIW pueden tener dos tipos de instrucciones: un slot o múltiples slots. La elección sólo puede hacerse en tiempo de compilación, por lo que es necesaria la instrucción *operation_group_name* al describir el modelo LISA. Las referencias entre la jerarquía de las operaciones se realizan con la palabra clave INSTANCE. El código sería similar a esto:

```
OPERATION A{DECLARE{INSTANCE B, C; }… }
OPERATION B{DECLARE{INSTANCE D; }… }
OPERATION C{…} OPERATION D{… }
```

Otra palabra clave relacionada con las referencias entre las operaciones es la palabra clave REFERENCE. La palabra clave REFERENCIA permite hacer referencia a una operación de bajo nivel F referirse a otra operación E que no está en la misma rama en el árbol de operaciones. Hay que tener en cuenta que cada operación que está en un nivel entre las operaciones de F e Y (en este caso, la operación B) tiene que utilizar una REFERENCE a Y. La palabra clave LABEL permite crear referencias cruzadas entre elementos terminales de diferentes secciones.

```
OPERATION A{DECLARE{GROUP X = {B||C}; GROUP Y = {D||E}; }…}
OPERATION B{DECLARE{INSTANCE F; REFERECE Y; }…}
OPERATION F{DECLARE{REFERENCE Y; }…}
OPERATION C{…} OPERATION D{…}
OPERATION E{…} OPERATION F{…}
```



La sección SYNTAX

La sección SYNTAX permite especificar la sintaxis de una operación. Se pude usar elementos como INSTANCE, GROUP y REFERENCE y también elementos terminales. El operador tilde indica que no se permiten espacios en blanco antes del siguiente elemento de la sintaxis.

```
SYNTAX f{"$R0." ~index = #U}
```

Por ejemplo, la línea anterior define un código ensamblador que no permite un espacio en blanco entre el registro $R0. y el índice del registro: $R0.15 es correcto pero $R0. 15 no lo es.

La sección CODING

El ensamblador LISATek usa la sección de codificación para traducir un fichero en ensamblador en un objeto binario. El simulador puede identificar en el código instrucciones válidas y a continuación, seleccionar la correspondiente operación LISATek. En una operación no terminal (una operación que no está completamente codificada y necesita de sub-operaciones para completar su codificación) algunas particularidades no se pueden establecer dentro de esa operación y sólo pueden hacerlo las operaciones de bajo nivel. Algunos elementos permitidos son los campos de tipo bit, GROUP, INSTANCE, asignaciones a LABELs y también expresiones aritméticas. Por ejemplo:

```
CODING {0b0 opcode dest_op source_op1 0bx [10]}
```

La sección EXPRESSION

Esta sección se utiliza para devolver recursos o expresiones numéricas que son devueltas a la sección BEHAVIOR.

```
EXPRESSION {15}
```

El ejemplo anterior es una muestra de la sección EXPRESSION: la operación en que se define va a devolver un valor (15 en el ejemplo) a la operación que la llama.

La sección ACTIVATION

Aquí es posible obtener la lista de operaciones que deben ser ejecutados en un posterior
paso de control. Si la operación activada está en otra etapa, se activa un registro del pipeline de forma automática para activarla en el ciclo correcto.

```
ACTIVATION {fetch}
```

La línea de arriba es la sección ACTIVATION de la operación ***main***: cada vez que se activa la operación ***main***, también se activa la operación ***fetch***.

La sección BEHAVIOR

En la sección se BEHAVIOR encuentra la descripción de lo que tiene que hacer la operación. Se realiza mediante código C y se permiten las variables, funciones y otros. Hay dos importantes diferencias la sección BEHAVIOR y el código C puro. En primer lugar no se pueden pasar parámetros a las operaciones LISATek y en segundo lugar en la sección BEHAVIOR está permitido el uso de identificadores de GROUPs, INSTANCEs y REFERENCEs.

```
BEHAVIOR
{
PIPELINE (pipe).execute();
PIPELINE (pipe).shift();
}
```

El ejemplo anterior es el de la sección BEHAVIOR de la operación ***main*** cada vez que se activa dicha operación, se ejecuta el código dentro de la sección BEHAVIOR.

3.- Trabajo práctico

3.1.-Uso de las herramientas para el trabajo práctico

Para el trabajo práctico se han usado dos tipos de herramientas.

Para el desarrollo del *parser* se ha usado un entorno de desarrollo de C++ para Windows mientras que para la implementación de los registros de *profiling* se ha usado el entorno de desarrollo LISATek.

3.2.- Análisis de la arquitectura de referencia

La arquitectura de referencia que se ha usado en el proyecto está basada en la arquitectura VEX descrita en el punto 1.2

En la arquitectura de referencia se ha encontrado una serie de fallos que impiden la correcta ejecución de los programas bajo ciertas circunstancias así como deficiencias de la arquitectura como la falta de direccionamiento indirecto de memoria.

Se han realizado una serie de pruebas para detectar qué tipo de conflictos inherentes a la arquitectura existen, y a continuación se describe tanto el conflicto como los resultados obtenidos.

3.2.1.- Diferencias de la arquitectura VEX con la arquitectura de referencia.

A pesar que la arquitectura de referencia se basa en VEX, existen diferencias con una arquitectura VEX tradicional.

A continuación se enumeran dichas diferencias y se indica cómo se ha superado dichas limitaciones.

VEX	Arquitectura de referencia	Solución
64 registros de propósito general	16 registros	Renombrado de registros
Direccionamiento indirecto de memoria	No hay direccionamiento indirecto de memoria	stw $r0.1 = 4[$r0.2] se traduce en gpr2= add gpr2, #4 gpr1 = sw [gpr2] gpr2 = sub gpr2 , #4
Valores hexadecimales negativos	No soporta valores hexadecimales negativos	Conversión de hexadecimal a decimal.

Instrucciones de salto en cualquier instrucción de la tupla de instrucciones	Las instrucciones de salto solo pueden estar en la primera instrucción de la tupla	Aplicar esta restricción al escribir código para la arquitectura de referencia
código VEX c0 add .. c0 sub .. c0 mpyl .. c0 mov ..	Código arquitectura de referencia add.. || sub.. || mpyl.. || mov..	Traducción de la sintaxis entre las diferentes arquitecturas
Comentarios en VEX: #	Comentarios en arquitectura de referencia: **;**	Traducción de la sintaxis entre las diferentes arquitecturas
Repertorio de instrucciones completo	Repertorio de instrucciones más reducido	No usar las instrucciones no soportadas
Sintaxis de lenguaje ensamblador VEX	Sintaxis de lenguaje ensamblador parecida a VEX	Traducción de la sintaxis entre las diferentes arquitecturas

3.2.2.- Deficiencias de la arquitectura de referencia

La arquitectura de referencia deriva de la arquitectura VEX, pero tiene algunas deficiencias que impiden que se puedan ejecutar determinadas instrucciones en paralelo. Para proceder a solucionar dichas deficiencias es importante determinar exactamente qué tipos de conflictos (*Hazards*) tiene dicha arquitectura.

Conflicto de primer orden (WAW)

Este conflicto se produce cuando hay un conflicto en cuanto al uso de recursos del procesador.

Conflicto WAW Suma

X14=X10+X11
X15=X12+X13

```
move  GPR10, #0x1 || move  GPR11, #0x2 || move  GPR12, #0x3 ||
move  GPR13, #0x4
GPR14 = ADD GPR10, GPR11 || GPR15 = ADD GPR12, GPR13
```

No hay conflicto en el recurso de la suma puesto que en el mismo paso se vuelcan los resultados de la suma en los registros GPR14 y GPR15.

Conflicto WAW Resta

X14=X10-X11
X15=X12-X13

```
move  GPR1, #0x1 || move  GPR2, #0x3 || move  GPR3, #0x5 ||
move  GPR4, #0x9
GPR5 = SUB GPR2, GPR1 || GPR6 = SUB GPR4, GPR3
```

No hay conflicto en el recurso de la suma puesto que en el mismo paso se vuelcan los resultados de la resta en los registros GPR5 y GPR6.

Conflicto WAW Multiplicación

X14=X10*X11

X15=X12*X13

```
move  GPR10, #0x1 || move  GPR11, #0x2 || move  GPR12, #0x3 ||
move  GPR13, #0x4
GPR14 = MPY GPR10, GPR11 || GPR15 = MPY GPR12, GPR13
```

No hay conflicto en el recurso de la suma puesto que en el mismo paso se vuelcan los resultados de la multiplicación en los registros GPR14 y GPR15.

Conflicto WAW División

X14=X10 / X11
X15=X12 / X13

```
move  GPR10, #0x1 || move  GPR11, #0x2 || move  GPR12, #0x3 ||
move  GPR13, #0x4
GPR14 = DIV GPR10, GPR11 || GPR15 = DIV GPR12, GPR13
```

No hay conflicto en el recurso de la suma puesto que en el mismo paso se vuelcan los resultados de la división en los registros GPR14 y GPR15.

Conflicto WAW Registro

X14=X10 + X11
X15=X12 * X13

```
move  GPR10, #0x1 || move  GPR11, #0x2 || move  GPR12, #0x3 ||
move  GPR13, #0x4
GPR14 = ADD GPR10, GPR11 || GPR14 = MPY GPR12, GPR13
```

Existe conflicto WAW a nivel de registro puesto que en la instrucción

`GPR14 = ADD GPR10, GPR11 || GPR14 = MPY GPR12, GPR13`

Sólo se ejecuta la primera parte: `GPR14 = ADD GPR10, GPR11`

Y no la segunda: `GPR14 = MPY GPR12, GPR13`

Conflicto de segundo orden (RAW)

X14=X10+X11
X15=**X14** / X12

```
move  GPR10, #0x1 || move  GPR11, #0x2 || move  GPR12, #0x3 ||
move  GPR13, #0x4
GPR14 = ADD GPR10, GPR11 || GPR15 = ADD GPR12, GPR13
```

Al igual que el conflicto WAW en los registros, existe una dependencia al paralelizar las instrucciones, puesto que el resultado de la ejecución en paralelo de:

`GPR14 = ADD GPR10, GPR11 || GPR15 = ADD GPR12, GPR13` es `GPR14=3` y `GPR15=7` cuando debería ser `GPR14=3` y `GPR15=10`

Soluciones propuestas para este conflicto:

1) Error sintáctico que no permite la paralelización
2) Ejecutar NOP hasta que el registro se haya modificado

Conflicto de segundo orden (RAW) cambiado de orden

X15=**X14** / X12
X14=X10+X11

```
move  GPR10, #0x1 || move  GPR11, #0x2 || move  GPR12, #0x3 ||
move  GPR13, #0x4
GPR15 = ADD GPR12, GPR13 || GPR14 = ADD GPR10, GPR11
```

El cambio de orden de las operaciones no provoca ningún cambio en el resultado puesto que el valor del `GPR14` no se propaga.

Conflicto de tercer orden (WAR)

X14 = X10 / X11
X15 = **X14** * **X12**
X12=X9+X8

```
move  GPR10, #0x1 || move  GPR11, #0x2 || move  GPR12, #0x3 ||
move  GPR13, #0x4 || move  GPR9, #0x6  || move  GPR8, #0x7
GPR14 = ADD GPR10, GPR11 || GPR15 = ADD GPR14, GPR12 ||
GPR12 = ADD GPR9, GPR8
```

La ejecución teórica correcta del programa sería: `GPR14=3`, `GPR15=10` y `GPR12=13` pero la ejecución final del programa es `GPR14=3`, `GPR15=7` y `GPR12=13` de esto se deduce que no existe un conflicto de tipo WAR (Write After Read) puesto que se ha podido paralelizar las instrucciones:

```
GPR15 = ADD GPR14, GPR12 || GPR12 = ADD GPR9, GPR8
```

Con lo que el error está en el conflicto RAW

3.2.3.- Soluciones propuestas para la arquitectura de referencia

Para solucionar las deficiencias de la arquitectura de referencia se ha diseñado un *parser* que traduce las instrucciones de la arquitectura VEX a la arquitectura de referencia de forma que se corrijan los conflictos (*hazards*) y el programa se ejecute correctamente.

Solución para el conflicto WAW

```
      .text
_init:
      MOVE GPR10, #0x1 || MOVE GPR11, #0x2 || MOVE GPR12, #0x3 ||
      MOVE GPR13, #0x4

      GPR14 = ADD GPR10, GPR11 || GPR14 = MPY GPR12, GPR13
      .end
```

Para solucionar el conflicto, pasamos el código por el *parser* y obtenemos el siguiente programa corregido:

```
      .text
_init:
      MOVE GPR10, #0x1 || MOVE GPR11, #0x2 || MOVE GPR12, #0x3 ||
      MOVE GPR13, #0x4

      GPR14 = ADD GPR10, GPR11
      .end
```

Solución para el conflicto RAW

```
      .text
_init:
      MOVE GPR10, #0x1 || MOVE GPR11, #0x2 || MOVE GPR12, #0x7 ||
      MOVE GPR13, #0x4

      GPR14 = ADD GPR10, GPR11 || GPR15 = ADD GPR14, GPR12
      .end
```

Para solucionar el conflicto, pasamos el código por el *parser* y obtenemos el siguiente programa corregido:

```
      .text
_init:
      MOVE GPR10, #0x1 || MOVE GPR11, #0x2 || MOVE GPR12, #0x7 ||
      MOVE GPR13, #0x4

      GPR15 = ADD GPR14, GPR12
      GPR14 = ADD GPR10, GPR11
      .end
```

Solución para el conflicto WAR

```
      .text
_init:
      MOVE GPR10, #0x1 || MOVE GPR11, #0x2 || MOVE GPR12, #0x7 ||
      MOVE GPR13, #0x4

      MOVE GPR9, #0x6 || MOVE GPR8, #0x7

      GPR14 = ADD GPR10, GPR11 || GPR15 = ADD GPR14, GPR12 ||
      GPR12 = ADD GPR9, GPR8
      .end
```

Para solucionar el conflicto, pasamos el código por el *parser* y obtenemos el siguiente programa corregido:

```
      .text
_init:
      MOVE GPR10, #0x1 || MOVE GPR11, #0x2 || MOVE GPR12, #0x7 ||
      MOVE GPR13, #0x4

      MOVE GPR9, #0x6 || MOVE GPR8, #0x7

      GPR12 = ADD GPR9, GPR8 || GPR14 = ADD GPR10, GPR11
      GPR15 = ADD GPR14, GPR12 ||
      .end
```

3.3.- Generación de un conjunto de *benchmarks* de evaluación de prestaciones y detección de conflictos (*hazards*)

Como se ha dicho anteriormente, se ha creado un *parser* que traduce las instrucciones de arquitectura VEX a arquitectura de referencia. Dicha traducción soluciona los conflictos existentes de instrucciones en la arquitectura de referencia a cambio de introducir una penalización en cuanto a la pérdida de paralelismo.

Por tanto, para evaluar el rendimiento de las aplicaciones en la arquitectura VEX es necesario tener en cuenta tanto las prestaciones (paralelismo) teóricas de las aplicaciones en VEX como el rendimiento real de dichas aplicaciones en la arquitectura de referencia después que el *parser* haya traducido dichas aplicaciones.

Por tanto, se usarán las siguientes medidas:

1) Rendimiento teórico en VEX.
2) Rendimiento real en arquitectura de referencia.
3) Rendimiento del código en el peor caso (Sólo utilizamos un *cluster* del procesador).

Los *benchmarks* que se han usado para evaluar el rendimiento incluyen algoritmos de ordenación (que hacen un uso intensivo de posiciones de memoria), algoritmos de

cálculo intensivo así como algoritmos clásicos como el de las torres de Hanoi o el producto de matrices.

3.4. Desarrollo de una herramienta automática de generación de código

Debido a los fallos en el diseño del procesador que impiden la correcta ejecución de las aplicaciones VEX, se decidió desarrollar una herramienta (*parser*) que tradujera y solucionara dichas deficiencias a fin de que se puedan ejecutar las aplicaciones VEX en dicho procesador.

Para la implementación del parser se usó C++ puesto que se dispone de un compilador en la máquina UNIX (Linux), al igual que experiencia previa en el desarrollo de aplicaciones en dicho lenguaje de programación.

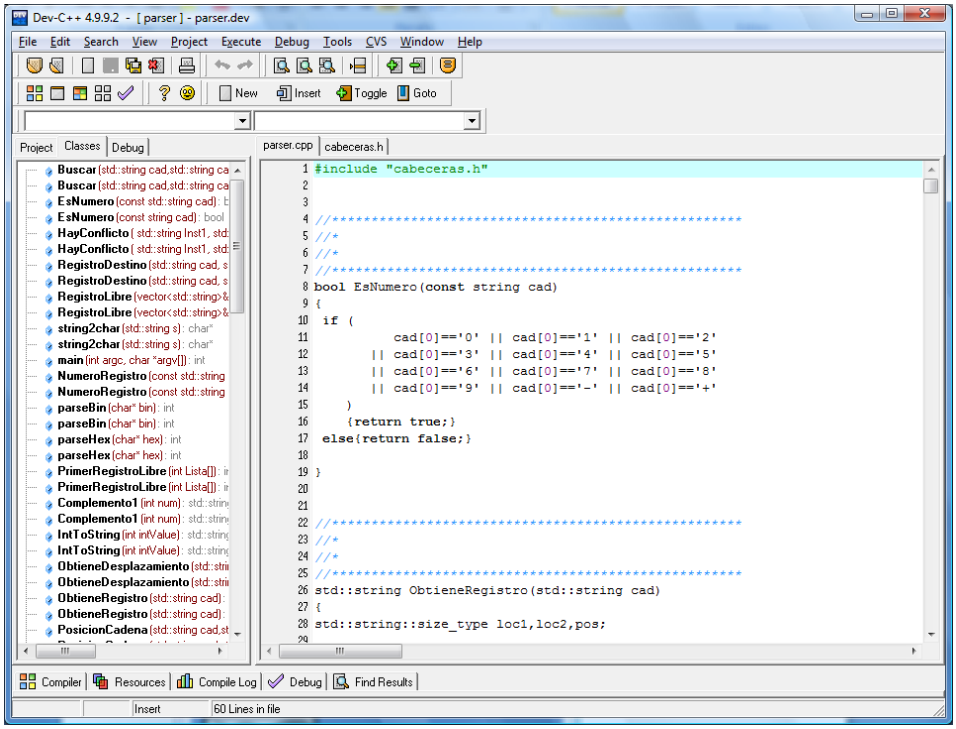
Si bien el desarrollo de la aplicación puede hacerse íntegramente en el entorno UNIX, se ha preferido desarrollar en entorno Windows puesto que de esta forma no es necesario tener abierta una conexión ssh a la máquina UNIX donde se encuentran instaladas las herramientas de LISATek.

En concreto, el desarrollo se ha realizado mediante el entorno de desarrollo Dev-C++ y el compilador C++ que se ha usado es el del proyecto Mingw.

Una vez se ha desarrollado el software en el PC, se ha copiado el código fuente y se ha compilado en la máquina UNIX (Linux) para poder ejecutar el *parser* también en la máquina UNIX. El código fuente se ha copiado mediante sftp.

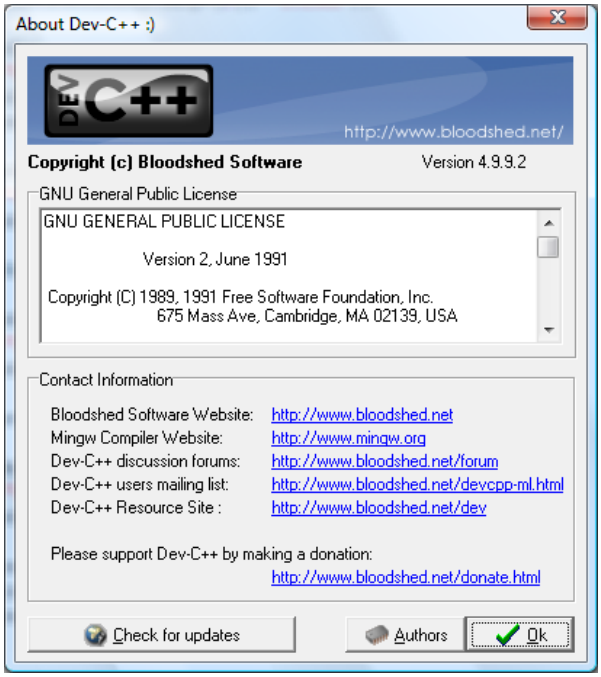
Todas las herramientas están disponibles en Internet de forma gratuita puesto que son software libre.

A continuación se muestra una captura de pantalla del entorno. El entorno gráfico permite editar, compilar y depurar fácilmente los errores producidos durante el proceso de compilación.



En la siguiente captura de pantalla se muestra el entorno usado y dónde puede descargarse.

El IDE de desarrollo está desarrollado por Bloodshed Software. El compilador compatible con g++ es el compilador Mingw.



3.4.1.- Extracción de reglas

Para poder traducir las instrucciones de un programa VEX a un programa para la arquitectura de referencia se procesa el programa línea a línea. Se identifican las instrucciones y en caso que sea necesario algún tipo de conversión se aplica. Una vez tratada la línea de instrucciones del programa, se escribe en un fichero con el programa traducido.

```
parser [-t] fichero_entrada fichero_salida
-t : traducir de VEX a la arquitectura de referencia
fichero_entrada : Fichero de entrada
fichero_salida  : Fichero de salida
```

Se consideran dos casos:

1) Traducción de un programa de VEX a la arquitectura de referencia.
2) Un programa que cumple la gramática de la arquitectura de referencia y necesita que se añada el código suficiente para resolver los *hazards* y se paralelice el código lo máximo posible.

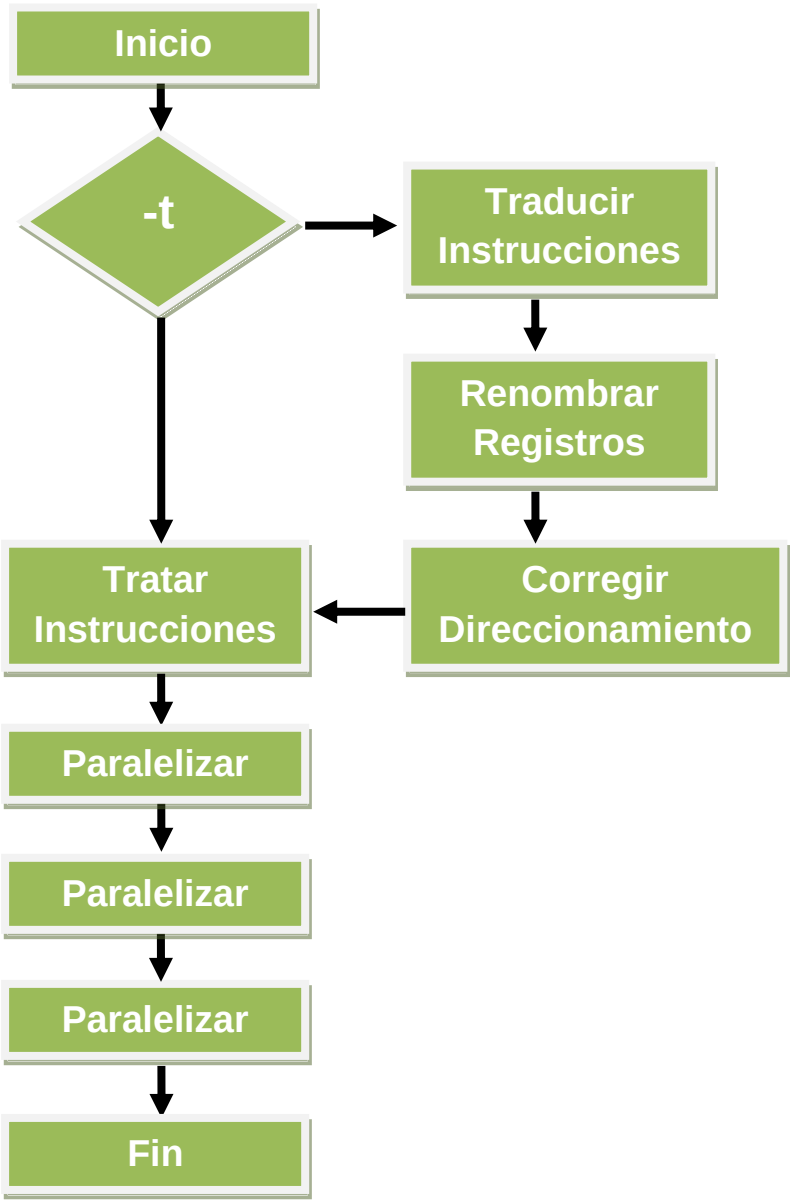
El programa acepta un parámetro `[-t]` que permite seleccionar cuál de los dos casos se usará. En caso que no se ejecute con la aplicación con el parámetro `[-t]`, sólo se añadirá el código necesario para resolver los *hazards* y se paralelizará al máximo el código.

3.4.2.- Detalles de implementación

Cada una de las cadenas de texto que se lee del fichero de código fuente del programa se carga en una variable de clase string. La razón por la que se usa la clase *string* es debido a que se simplifican mucho las tareas de buscar cadenas, concatenar y reemplazar cadenas.

Esquema de la aplicación.

En el siguiente diagrama de flujo se muestran los pasos que realiza la aplicación para pasar un programa de arquitectura VEX a la arquitectura de referencia.



Descripción de los pasos.

Los pasos que el *parser* realiza para pasar de una aplicación VEX estándar a una aplicación para la arquitectura de referencia son los siguientes y se ejecutan en el orden en que se describen:

1.- Traducir Instrucciones

La sintaxis de algunas instrucciones en el procesador de referencia no es exactamente la misma que en la arquitectura VEX, por lo que es necesaria una conversión entre la sintaxis VEX y la sintaxis de la arquitectura de referencia. En el paso de traducción de sintaxis no se realiza ningún tipo de optimización en el código.

Instrucción	Cambio
mov $r0.4 = $r0.7	move GPR4, GPR7
$R0.	GPR
$B0.	BTR
$L0.0	LC
##	;
?	_
.COMMENT	(se elimina en la arquitectura de referencia)
.SVERSION	(se elimina en la arquitectura de referencia)
.RTA	(se elimina en la arquitectura de referencia)
.EQU	(se elimina en la arquitectura de referencia)
.CALL	CALL
.SECTION	(se elimina en la arquitectura de referencia)
.RETURN	(se elimina en la arquitectura de referencia)
.PROC	(se elimina en la arquitectura de referencia)
.ALIGN	(se elimina en la arquitectura de referencia)
.ENTRY	(se elimina en la arquitectura de referencia)
.TRACE	(se elimina en la arquitectura de referencia)
.BSS	(se elimina en la arquitectura de referencia)
C0	(se elimina en la arquitectura de referencia)
Direccionamiento indirecto stw $r0.1 = 4[$r0.2]	No existe direccionamiento indirecto por lo que hay que hacer el incremento al registro donde hay la base del desplazamiento, leer la posición de memoria y restaurar el registro. GPR2= ADD GPR2, #4 GPR1 = SW [GPR2] GPR2 = SUB GPR2 , #4

2.- Renombrar Registros

En las arquitecturas VEX convencionales, hay 64 registros de propósito general, mientras que en la arquitectura de referencia sólo hay 16 registros de propósito general. Es por esto que en caso que en el programa original se utiliza un registro superior o igual al registro 16 será necesario comprobar si es posible “renombrar” dicho registro a otro registro entre 0 y 15. En caso que sea posible, el registro será renombrado y la traducción del programa continuará. En caso contrario, se detendrá la conversión del programa y se informará de que la conversión no es posible. Tampoco en este paso realiza ningún tipo de optimización en el código.

3.- Corregir Direccionamiento

Otra de las deficiencias de la arquitectura de referencia es la ausencia de direccionamiento indirecto de la memoria, por lo que también es necesario añadir cierto código para suplir esta deficiencia de la arquitectura. El uso de este recurso hace que el rendimiento de los programas en el procesador de referencia sea menor

de lo esperado y dicha pérdida de rendimiento será evaluada gracias a los *benchmarks*.

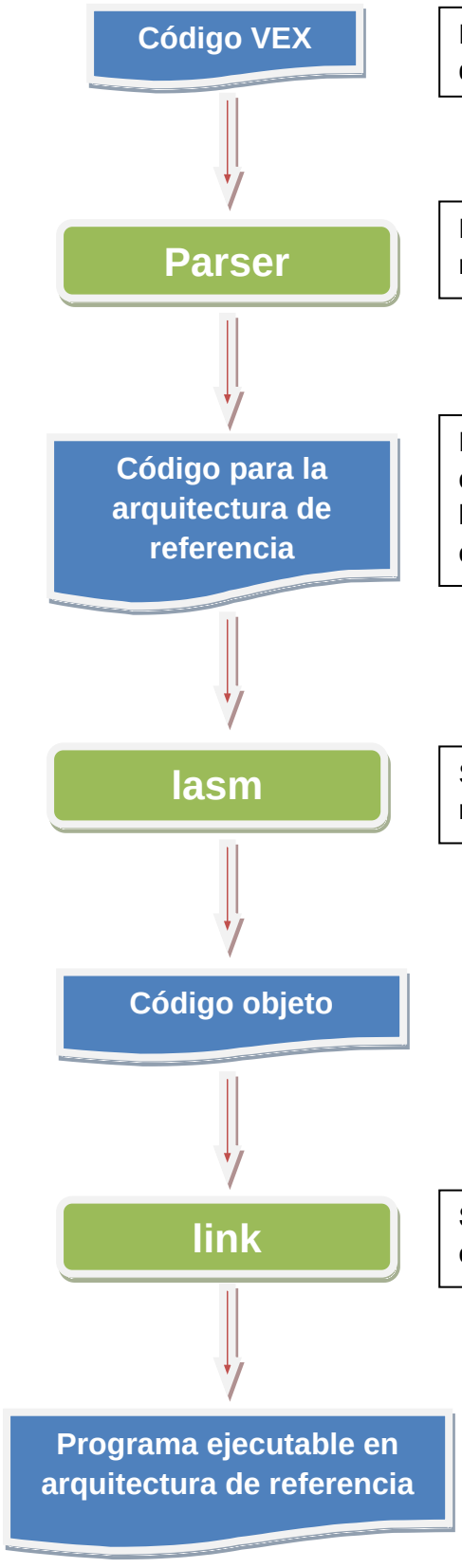
4.- Tratar Instrucciones

En este paso se solucionan los problemas con los conflictos de instrucciones (*hazards*) y se corrigen los errores en la ejecución de las instrucciones. En este paso también se introducen cambios en el código que alteran el rendimiento de la aplicación. Las soluciones propuestas para solucionar dichos *hazards* se han descrito previamente en el punto **3.1.2**

5.- Paralelizar

Los cambios realizados por el *parser* en los pasos anteriores han provocado que el código generado por el *parser* no sea todo lo más paralelo posible y por tanto, es necesario ejecutar una función que paralelice al máximo las instrucciones del programa. Dicho proceso lee las instrucciones de dos en dos y las paraleliza en lo posible. Este paso se ejecuta tres veces durante la ejecución del *parser*.

3.4.3.- Diagrama de flujo de la secuencia de compilación



Partimos de un programa en código para la arquitectura de procesadores VEX.

Para traducir el programa VEX a la arquitectura de referencia ejecutamos la aplicación *parser*.

El resultado de la ejecución es un programa que puede compilarse con el compilador generado por las herramientas CoWare para el diseño de procesadores con la que se ha diseñado la arquitectura de referencia.

Se compila el programa para la arquitectura de referencia. Obtenemos el código objeto.

Se ejecuta el "linker" y obtenemos un programa ejecutable sobre la arquitectura de referencia.

3.4.4.- Extracción de resultados y conclusiones

Se han realizado varios *benchmarks* para evaluar la pérdida de prestaciones que sufren los programas al ser traducidos de la arquitectura VEX a la arquitectura de referencia. Los *benchmarks* se han realizado para evaluar el rendimiento de las aplicaciones sobre el procesador son de varias naturalezas:

- Algoritmos clásicos
- Algoritmos de ordenación
- Algoritmos de cálculo.

La razón por la que se han elegido estas clases de *benchmarks* es porque cada tipo de algoritmo evalúa intensivamente una parte de la arquitectura de referencia y nos indica el grado de penalización (pérdida de prestaciones) en comparación con la arquitectura VEX.

En el apartado de algoritmos clásicos está el algoritmo de las Torres de Hanoi. Los algoritmos clásicos evalúan el rendimiento de una aplicación estándar ejecutándose sobre la arquitectura de referencia.

Para la evaluación del rendimiento (o penalización) se contemplan tres casos de paralelismo:

Arquitectura VEX	Caso ideal en el que se ejecutan cuatro instrucciones a la vez.
Arquitectura de referencia	Cuatro instrucciones a la vez como máximo pero pueden ser menos debido a la reordenación de código realizada para evitar los *hazards*.
Peor caso	Sólo se ejecuta una instrucción por ciclo.

Para el cálculo de la ratio de la arquitectura de referencia, se utiliza la siguiente fórmula:

```
(Arquitectura VEX / Arquitectura de referencia) * 100
```

Para el cálculo de la penalización se utiliza la siguiente fórmula:

```
1-((Arquitectura VEX / Arquitectura de referencia) * 100)
```

Análogamente, para el cálculo de la ratio del peor caso:

```
(Arquitectura VEX / Peor caso) * 100
```

Y para el cálculo de la penalización del peor caso:

```
1-((Arquitectura VEX / Peor caso) * 100)
```

A continuación se muestra el resultado de algunos *benchmarks* sobre el procesador de referencia.

Benchmark Matrices

A primera vista la penalización de la ejecución de dicho algoritmo parece excesiva pero un análisis más detallado revela que el programa usa intensivamente unos pocos registros en la mayoría de las instrucciones. Este hecho hace que debido a los *hazards* que se producen debido al diseño de la arquitectura, dichas operaciones sean muy poco paralelizables. En este caso la paralelización es del 62,88%.

Matrices

	Líneas	Ratio	Penalización
Arquitectura VEX	85		
Arquitectura de referencia	229	37,12	**62,88**
Peor caso	247	34,41	**65,59**

Porcentaje de instrucciones

add / sub	12%
mpy / div	11%
Move	15%
ldw / stw	62%

Benchmark Hanoi

La aplicación Hanoi es el algoritmo clásico de las torres de Hanoi. Los cambios de valor en las variables se realizan principalmente en los registros del procesador, por lo que la penalización sufrida debido a la conversión de VEX a la arquitectura de referencia es sólo del 21,21%.

Hanoi

	Líneas	Ratio	Penalización
Arquitectura VEX	26	1,00	
Arquitectura de referencia	33	78,79	**21,21**
Peor caso	91	28,57	**71,43**

Porcentaje de instrucciones

add / sub	10,8%
mpy / div	0%
Move	89,2%
ldw / stw	0%

Benchmark Burbuja

La aplicación burbuja realiza una ordenación por el método de la burbuja. El hecho que tenga una penalización de sólo el 18,33% es debido a que el cambio del valor de las variables se realiza en registros hardware y por tanto la penalización debida a la conversión entre arquitecturas es mínima.

Burbuja

	Líneas	Ratio	Penalización
Arquitectura VEX	49	1	
Arquitectura de referencia	60	81,67	**18,33**
Peor caso	179	27,37	**72,63**

Porcentaje de instrucciones

add / sub	18%
mpy / div	14,5%
Move	67,5%
ldw / stw	0%

Benchmark Insert Sort

El siguiente algoritmo de ordenación lee y escribe posiciones de memoria intensivamente. El código añadido por el *parser* para solventar la imposibilidad de hacer direccionamientos indirectos en la memoria por parte de la arquitectura de referencia hace que la penalización sea del 59,09%

Insert Sort

	Líneas	Ratio	Penalización
Arquitectura VEX	81		
Arquitectura de referencia	198	40,91	**59,09**
Peor caso	264	30,68	**69,32**

Porcentaje de instrucciones

add / sub	54%
mpy / div	4,4%
Move	17,6%
ldw / stw	24%

Benchmark Shell Sort

Al igual que el algoritmo Insert Sort, este algoritmo lee y escribe posiciones de memoria intensivamente. Por lo que la penalización sufrida en la conversión de VEX a la arquitectura de referencia también es muy alta; concretamente del 64,29%

Shell Sort

	Líneas	Ratio	Penalización
Arquitectura VEX	105		
Arquitectura de referencia	294	35,71	**64,29**
Peor caso	345	30,43	**69,57**

Porcentaje de instrucciones

add / sub	51,6%
mpy / div	6,7%
Move	14,7%
ldw / stw	27%

Benchmark Qsort

Al *benchmark* Qsort le pasa algo parecido al los *Benhmarks* Insert Sort y Shell Sort. En este caso la penalización es del 64,31%

Qsort

	Líneas	Ratio	Penalización
Arquitectura VEX	217		
Arquitectura de referencia	608	35,69	**64,31**
Peor caso	741	29,28	**70,72**

Porcentaje de instrucciones

add / sub	55,4%
mpy / div	5,6%
Move	13,4%
ldw / stw	25,6%

Benchmark QuickSort

Este es también un caso de algoritmo de ordenación con lecturas y escrituras intensivas de memoria. En este caso la penalización es del 63,92%

Quicksort

	Líneas	Ratio	Penalización
Arquitectura VEX	184		
Arquitectura de referencia	510	36,08	**63,92**
Peor caso	615	29,92	**70,08**

Porcentaje de instrucciones

add / sub	55,1%
mpy / div	4,2%
Move	13%
ldw / stw	27,7%

Benchmark BasicMath

A priori se esperaba un mejor un mejor rendimiento de la aplicación *BasicMath* sobre el procesador de referencia puesto que no realiza lecturas y escrituras de forma intensiva en la memoria. Sin embargo, la penalización es del 58.96%. Esto es debido a que si bien en este caso no hay direccionamiento indirecto tal como ocurría en el caso de los algoritmos de ordenación, el hecho que el cálculo intensivo se realice en pocos registros, hace que dichas instrucciones sean difícilmente paralelizables.

BasicMath

	Líneas	Ratio	Penalización
Arquitectura VEX	348		
Arquitectura de referencia	848	41,04	**58,96**
Peor caso	1272	27,36	**72,64**

Porcentaje de instrucciones

add / sub	36,6%
mpy / div	0,5%
Move	50,2%
ldw / stw	12,7%

Conclusiones

El cálculo del porcentaje de cada tipo de operación en cada una de los benchmark se ha realizado para que en una futura reingeniería del procesador pueda saberse qué parte del procesador es más urgente modificar.

Dependiendo del tipo de aplicación que va a ejecutarse en el procesador de referencia, será más urgente implementar una solución a los *hazards* detectados o bien implementar el direccionamiento indirecto de memoria o incluso implementar nuevas instrucciones.

En las pruebas realizadas, se deduce que la existencia de *hazards* impide ejecutar cuatro instrucciones por ciclo, ejecutándose normalmente dos o incluso sólo una a la vez.

La carencia de direccionamiento indirecto de memoria puede suplirse mediante instrucciones añadidas al código que si no existieran los *hazards* podrían paralelizarse y no añadir demasiada penalización en el tiempo de ejecución de la aplicación.

Por esta razón es necesario conocer cuánto tiempo tarda en ejecutar una instrucción cada una de las unidades funcionales y con estos datos, acometer la reingeniería del procesador de referencia.

3.5.- Implementación de un mecanismo de *profiling* hardware

No es suficiente saber que se producen los *hazards* y corregirlos mediante software. No es la solución ideal pero sí la penalización es aceptable. Eso se puede saber ejecutando varios *benchmarks* sobre la arquitectura de referencia. Como se ha comentado anteriormente, para poder corregir en el futuro los errores de diseño del procesador de referencia es necesario saber por qué se producen dichos errores. Es necesario determinar cuánto tardan los cálculos en realizarse en cada unidad funcional y si el error viene determinado porque se vuelcan los resultados antes de tiempo en el registro.

Para el trabajo práctico de implementación de registros de *profiling* se han usado las herramientas del entorno de desarrollo LISATek descritas anteriormente.

Se ha partido de una arquitectura de referencia basada en VEX descrita en lenguaje LISA y se ha añadido código de descripción de registros a la descripción del procesador. Además se han añadido instrucciones para mostrar el contenido de dichos registros de *profiling* en diferentes puntos de procesador.

Una vez añadido los registros de *profiling*, se genera el procesador y en entorno de desarrollo LISATek genera las herramientas software para generar código para dicho procesador

Se desarrollan aplicaciones para comprobar el funcionamiento de procesador y extraer datos de rendimiento para el procesador desarrollado. Se compilan dichas aplicaciones mediante las herramientas software desarrolladas por el entorno de desarrollo LISATek al generar el procesador.

En el *processor debugger* se carga la arquitectura de referencia y una de las aplicaciones desarrolladas para las pruebas de rendimiento de procesador. Se ejecuta la aplicación y se extraen resultados de *profiling.* Se repite el proceso por cada una de las aplicaciones desarrolladas para la arquitectura de referencia.

3.5.1.- Descripción de la propuesta

Se trata de implementar un sistema hardware para evaluar los ciclos de reloj que consumen las operaciones y de esta forma determinar por qué se producen los *hazards* en la arquitectura de referencia.

3.5.2.- Implementación arquitectural

La propuesta se implementa definiendo varios registros de 32 bits llamados RELOJ, RELOJ_SUMA, RELOJ MULTIPLICADOR entre otros. Dichos registros tienen que ser de tipo TClocked porque tienen que actualizar su valor de forma síncrona con el reloj del sistema.

```
REGISTER TClocked<uint32> RELOJ;
REGISTER TClocked<uint32> RELOJ_SUMA;
REGISTER TClocked<uint32> RELOJ_MULTIPLICADOR;
REGISTER TClocked<uint32> RELOJ_MOVE;
REGISTER TClocked<uint32> RELOJ_LOADSTORE;
REGISTER TClocked<uint32> ACCESO_SUMA;
REGISTER TClocked<uint32> ACCESO_MULTIPLICADOR;
REGISTER TClocked<uint32> ACCESO_MOVE;
REGISTER TClocked<uint32> ACCESO_LOADSTORE;
```

Para modificar el valor del registro en cada ciclo de reloj se ha implementado un incrementador en la operación ***main***. Se ha implementado de esta forma porque la operación ***main*** se ejecuta en cada ciclo del reloj del procesador y por tanto los registros de *profiling* se incrementarán cada ciclo de reloj.

```
OPERATION main
{
  DECLARE {INSTANCE fetch;}

  BEHAVIOR
    {
     /* Execute all activated operations in the pipeline */
     PIPELINE(pipe).execute();

     /* Advance the pipeline by one cycle */
     PIPELINE(pipe).shift();

     /* Incrementar el valor del registro RELOJ */
     RELOJ++;

     /* Incrementar el valor de los registros RELOJ_SUMA, */
     /* RELOJ_RESTA, RELOJ_MULTIPLICADOR y RELOJ_DIVISOR  */
     RELOJ_SUMA++;
     RELOJ_MULTIPLICADOR++;
    }
   ACTIVATION {fetch}
}
```

Para obtener el valor de los registros de *profiling*, se ha añadido unas trazas en el código LISA. De esta forma mediante el ***processor debugger*** y ejecutando una aplicación se obtiene el valor de los registros de *profiling* antes y después de la ejecución de una instrucción en una unidad funcional. Restando los valores de entrada y salida obtenidos mediante las trazas, se obtienen los ciclos de reloj que tarda dicha unidad funcional en ejecutar la instrucción.
A continuación se muestra el código de traza del contador. Como puede verse, se captura el valor del contador antes de entrar al sumador y después de salir del sumador.

```
#ifdef LISA_DEBUG
    printf(" Antes de ALU ADD en datapath_slot0. Contador Programa =%d\n",CR_PC);
    printf(" Antes de ALU ADD en datapath_slot0. RELOJ =%d\n",RELOJ);
    printf(" Antes de ALU ADD en datapath_slot0. RELOJ_SUMA =%d\n",RELOJ_SUMA);
#endif

alu_out_ext = alu_op1_ext + alu_op2_ext;
alu_slot0_out = alu_out_ext & allf_32; // and-mask with 0x0ffffffff

#ifdef LISA_DEBUG
    printf(" Despues de ALU ADD en datapath_slot0. Contador Programa =%d\n",PC);
    printf(" Despues de ALU ADD en datapath_slot0. RELOJ =%d\n",RELOJ);
    printf(" Despues de ALU ADD en datapath_slot0. RELOJ_SUMA =%d\n",RELOJ_SUMA);
#endif
```

Además del valor del contador, se han añadido trazas para mostrar el valor de los registros de *profiling* de acceso a los recursos. De esta forma, es posible saber cuántas veces se accede a un recurso por cada instrucción.

3.5.3.- Extracción de resultados y conclusiones

Para poder solucionar las deficiencias de la arquitectura detectadas, es necesario poder determinar claramente el origen de dichas deficiencias. Esto incluye el retardo de las operaciones en las unidades funcionales puesto que sabiendo cuántos ciclos tarda en calcularse una suma, una resta, un producto o una división y cuándo se vuelca el resultado en un registro, será posible determinar el porqué del *hazard* y su solución.

Los siguientes resultados se han conseguido ejecutando un programa en el *debugger*. De los resultados obtenidos, se han extraídos los retardos por cada recurso. A continuación se muestra el retardo de cada uno de dichos recursos.

Retardo del sumador / restador

Sumador / restador: 4 ciclos

Accesos a la unidad sumador / restador: 1 acceso por tupla de instrucciones

Retardo del multiplicador / divisor

Multiplicador / divisor: 8 ciclos

Accesos a la unidad multiplicador / divisor: 1 acceso por tupla de instrucciones

Retardo de instrucción MOVE

Move: 1 ciclo

Accesos a los registros: 1 acceso por instrucción

Retardo de acceso a memoria

Acceso memoria: 78 ciclos

Accesos al módulo de memoria: 1 acceso por tupla de instrucciones

Gracias a estos resultados, podemos determinar el origen de los *hazards* y las soluciones a implementar para solucionarlos. Puesto que de los resultados anteriores de la traducción de programas de arquitectura VEX a la arquitectura de referencia. En concreto, las sumas y restas se calculan el doble de rápido que las multiplicaciones y divisiones. Por tanto, el resultado de la operación suma (resta) se vuelca antes en el registro de destino y puede dar lugar a *hazards* de tipo RAW o WAR.

4.- Conclusiones y trabajos futuros

Los resultados obtenidos en el análisis de prestaciones del procesador de referencia indican que las prestaciones obtenidas modificando el código mediante el *parser* para superar las deficiencias en el diseño no son óptimas y añaden una penalización excesiva para aplicaciones que requieran un alto rendimiento.

Si bien la modificación del código tiene un coste mucho menor que una reingeniería del procesador, los resultados obtenidos en los *benchmark* indican que para obtener un rendimiento aceptable es necesario acometer dicha reingeniería.

A partir de los resultados obtenidos mediante el *profiling* del sistema, el presente trabajo puede ser el punto de para acometer la citada reingeniería, focalizando y priorizando en los aspectos que más lastran el rendimiento del procesador estudiado. Dicha reingeniería tendría dos ventajas claras. La primera sería que mejorarían las prestaciones del procesador en un 65% aproximadamente.

Una segunda ventaja de dicha reingeniería sería que si el repertorio de instrucciones sería que si la arquitectura de referencia se acerca al repertorio de instrucciones VEX, no será necesario un “traductor” entre gramáticas de procesadores. En concreto, sería muy necesario que la arquitectura soportara el direccionamiento indirecto, puesto que la adaptación de los programas VEX evitaría añadir mucho código extra a los programas adaptados a la arquitectura de referencia que evitan dicha deficiencia.