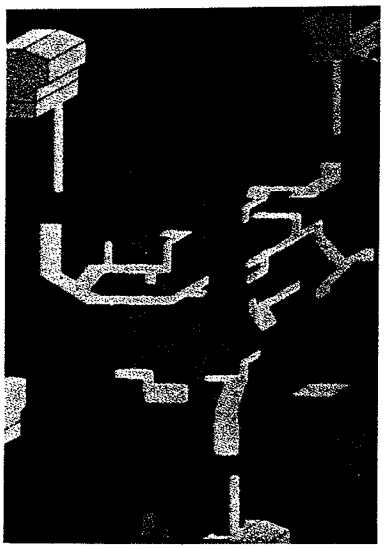
Formación (V)

El concepto RISC en el diseño de computadores

Las últimas generaciones de microprocesadores incorporan innovaciones como la arquitectura RISC, memoria cache *y otros conceptos de diseño que mejoran su eficiencia. Este artículo analiza los rasgos característicos de las arquitecturas RISC y CISC para lectores no versados en informática.*



1. Introducción

El trabajo esencial de un computador es ejecutar programas. Estos programas, codificados en lenguaje máquina (0 y 1), suelen ser el resultado de la traducción de programas escritos en lenguajes de alto nivel o en lenguaje ensamblador (figura 1).

La elección de uno de estos lenguajes, a la hora de escribir una determinada aplicación, descansa en las hipótesis siguientes:

- Se ejecuta más rápidamente una aplicación escrita directamente en lenguaje ensamblador que utilizando el lenguaje máquina generado por un compilador.
- El coste del hardware va decreciendo relativamente con respecto al del programador.
- Escribir una aplicación en lenguaje máquina es el método más lento de hacerlo.

Los factores de coste y producción han supuesto el uso generalizado de lenguajes de alto nivel.

Por otro lado, la pérdida de eficiencia por el uso de lenguajes de alto nivel ha provocado que los diseñadores de computadores concentren sus esfuerzos en conseguir la ejecución más rápida de los programas escritos en este tipo de lenguajes. Las arquitecturas denominadas RISC (*Reduced Instruction Set Computer*) son el fruto de ese esfuerzo.

2. Medida de la eficiencia

Para medir la eficiencia con que distintas soluciones arquitecturales ejecutan los lenguajes de alto nivel nos serviremos de una medida que es función del tiempo necesario para la ejecución de un programa:

$$\text{eficiencia} = 1 / (i \times tc \times nc)$$

Donde i es el número de instrucciones que se ejecutan, tc el tiempo de ciclo del procesador y nc el número medio de ciclos de procesador que necesita una instrucción. El tiempo medio de ejecución de una instrucción viene definido por el producto tc × nc.

Para hacer máxima la eficiencia es necesario:

- Minimizar el tiempo de ciclo del sistema. Esto obliga a reducir el tiempo no productivo por instrucción (ej.: decodificación del modo de direccionamiento) y a reorganizar el hardware para disminuir los retardos en cada ciclo de reloj.
- Reducir el número de ciclos necesarios para la ejecución de una instrucción. El ideal sería conseguir la ejecución de una instrucción por ciclo. Esto

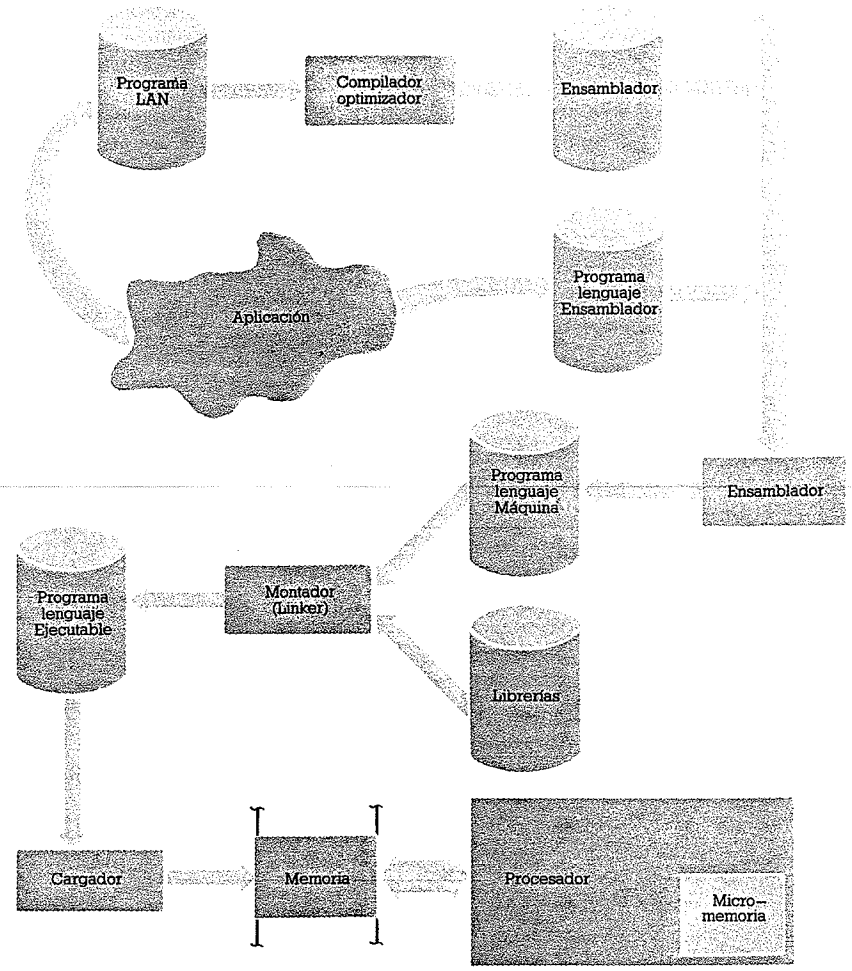


Fig. 1. Esquema general de codificación y ejecución de un programa.

puede implicar la necesidad de sacrificar la eficiencia de aquellas partes de la arquitectura menos utilizadas en beneficio de las más utilizadas.

- Hacer que el número de instrucciones necesario sea pequeño.

Generalmente, la arquitectura de la máquina define el número de ciclos por instrucción, mientras que el tiempo de ciclo depende de la tecnología utilizada y el número de instrucciones viene determinado por el compilador y por el nivel semántico de las instrucciones de lenguaje máquina.

3. Filosofías de diseño

A tenor de lo expuesto, analizaremos dos filosofías actuales de diseño: la denominada CISC *(Complex Instruction Set Computer)* y la filosofía RISC *(Reduced Instruction Set Computer)*.

3.1. CISC (Complex Instruction Set Computer)

La filosofía CISC consigue aumentar la eficiencia por medio de la reducción del número de instrucciones de lenguaje máquina necesarias. Para ello se aumenta el nivel semántico de dichas instrucciones (similares a las de los lenguajes de alto nivel). De ahí que el diseñador opte por incluir instrucciones cada vez más complejas. Por ejemplo, MC 68020 posee modos de direccionamiento para acceder a estructuras complejas; I80386 tiene instrucciones especiales para bucles; y DEC VLSI VAX posee 21 modos de direccionamiento y 304 instrucciones.

Con esta estrategia se persigue que el código resultante de la compilación contenga menos instrucciones para minimizar el término i. Esto será tanto más cierto si las instrucciones con alto nivel semántico son frecuentemente utilizadas y el compilador sabe reconocer su utilización a partir del lenguaje de alto nivel.

A nivel de arquitectura, cada instrucción de lenguaje máquina se interpreta mediante el microprograma asociado a dicha ins-

Historia

Aunque hoy en día todavía se discute la paternidad de la idea RISC, se reconoce que Seymour Cray introduce ya algunas de sus características, con computadores como el CDC6600, el CDC7600 y los mismos Cray, como son: un conjunto de instrucciones sencillas, un banco de registros, segmentación y otras.

El nombre de RISC se debe a los trabajos realizados en la Universidad de Berkeley, donde los profesores David Patterson y Carlos Séquin, junto con un grupo de alumnos, diseñaron los procesadores denominados RISC-I y RISC-II utilizando la tecnología VLSI (*Very Large Scale Integration*). Poco después, John Hennessy lideraría el diseño del procesador MIPS en la Universidad de Stanford.

Actualmente, la mayoría de los constructores siguen esta filosofía de diseño. Empezando por *IBM* (ya lo hizo en su proyecto 801, otro candidato a ser el primer RISC), se incluyen fabricantes como *Apollo, Sun, Motorola, Intel, Texas Instruments, Fairchild, AMD, Inmos, Fujitsu, AT&T, Xerox, Unisys* y otros.

trucción. El aumento de eficiencia está en función de la rapidez de acceso a la micromemoria. Generalmente, el tiempo de acceso a la micromemoria es inferior en algunos órdenes de magnitud con respecto al acceso a la memoria principal.

Interesa destacar el gran espacio de área de chip que se utiliza para lógica de control y de decodificación (en el MC68020 es el 68 %) y el elevado número de transistores necesarios en su realización (190.000 en el MC680 20 y 1,2 millones en el DEC VLSI VAX).

Esta filosofía, que aparece como una extensión de las arquitecturas clásicas a las que se les han ido añadiendo funciones tanto hardware como firmware, ha representado el claro exponente de todas las familias de computadores: *Intel, Motorola* y otros.

Las dificultades a las que se enfrenta esta filosofía se pueden resumir en las siguientes:

- El hardware es más difícil de modificar que el software. Si la función hardware no se ajusta exactamente a las necesidades del lenguaje o la aplicación, no podrá ser utilizada y, por tanto, será ineficiente. Este desajuste se denomina *mismatch* (desadaptación).
- La inclusión de nuevas funciones hardware implica una mayor complejidad, incrementando el número de ciclos por instrucción y, a menudo, el tiempo de ciclo.
- El elevado número y la complejidad de las instrucciones de lenguaje máquina, dificultan la elección, por parte del compilador, del formato de instrucción y modo de direccionamiento entre los muchos existentes. Es necesario elegir entre un compilador eficiente, más complejo y lento, y uno más rápido pero menos eficiente. Hay que recordar que en muchos entornos de trabajo el

Técnica de segmentación

Esta técnica se introdujo en la década de los 50 para conseguir que el procesador pudiera estar ejecutando varias instrucciones en el mismo instante de tiempo. El objetivo final era la ejecución de una instrucción por ciclo de procesador. Para ilustrar la explicación, usaremos como ejemplo la propuesta diseñada en el RISC-II (figura 2). Se ha dividido la ejecución de la instrucción en tres etapas:

- Obtención de la instrucción y decodificación (*fetch*).
- Obtención de operandos y cálculo (*execute*).
- Escritura de los resultados (*write*).

Si no hubiese segmentación, la instrucción I2 no iniciaría su ejecución hasta tres ciclos después de que la instrucción I1 iniciase la etapa de fetch. Idealmente, siempre que la obtención de una instrucción sea independiente de la ejecución de otra, el aumento de eficiencia es proporcional al número de etapas (en nuestro caso 3). La frecuencia de saltos y la dependencia de datos entre instrucciones dentro de los programas limita el número de etapas.

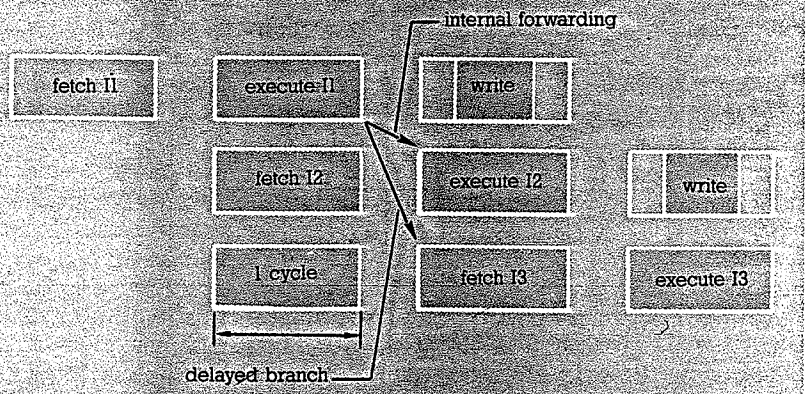


Fig. 2. Esquema de ejecución segmentada en el RISC-II.

Cuando se ejecuta una instrucción de salto, puede suceder que no se conozca la dirección de la siguiente instrucción de programa hasta que finalice la de salto. Este hecho suele ser motivado por el modo de direccionamiento utilizado o por el cálculo de códigos de condición. Si se para el flujo de instrucciones hasta la obtención de la dirección, la pérdida de eficiencia puede ser elevada y oscila entre un 30 y un 50 %.

Existen técnicas de predicción de saltos para minimizar esta pérdida. Una de las más populares es la *delayed branch*. En dicha técnica, un número determinado de instrucciones, que aparecen después de la instrucción de salto, se ejecutan siempre antes de que comience la instrucción destino de salto. El número de instrucciones previas ejecutadas con anterioridad depende de las etapas necesarias para calcular la dirección (en nuestro caso 1). El compilador, siempre que puede, reorganiza el código para que las instrucciones posteriores a la de salto sean siempre útiles. Si no encuentra ninguna, inserta la instrucción *no hacer nada* (*noop*).

El problema de la dependencia de los datos aparece cuando la instrucción i+1 requiere el resultado de alguna instrucción previa, pongamos i. Una de las soluciones propuestas consiste en permitir un camino de datos interno entre etapas, que permita pasar datos entre instrucciones i e i+1. Esta estrategia se denomina *internal forwarding* o *bypassing* y se utiliza en el RISC-II. Otra solución, implementada en hardware para el RISC-I y en software para el MIPS, consiste en impedir el uso de un registro hasta que se haya actualizado su contenido. La opción software se realiza por medio de la reorganización del código en forma similar al *delayed branch*, y la opción hardware parando la ejecución de las instrucciones consecutivas a i.

Banco de registros

La utilización de un gran número de registros (más de 30), característica de la arquitectura RISC, se debe a que la sentencia que más se ejecuta es la de asignación (x=y), por lo que se requiere un acceso rápido a los operandos. Otro motivo fundamental de la abundancia de registros, es el uso generalizado de la programación modular, en lenguajes de alto nivel, que obliga a minimizar el tiempo de llamada y retorno de las subrutinas.

El banco de registros se divide en conjuntos de registros (ventanas), cuyo tamaño intenta adaptarse al número de variables locales y parámetros que necesitan las subrutinas. Estos registros son de uso general y el tamaño de las ventanas puede ser fijo para todas las subrutinas, o variable. Para minimizar el efecto de las llamadas, se solapan parte de las ventanas de registros para pasar variables a subrutinas sin tener que mover los operandos (figura 3). La asignación de operandos a los registros la realiza el compilador.

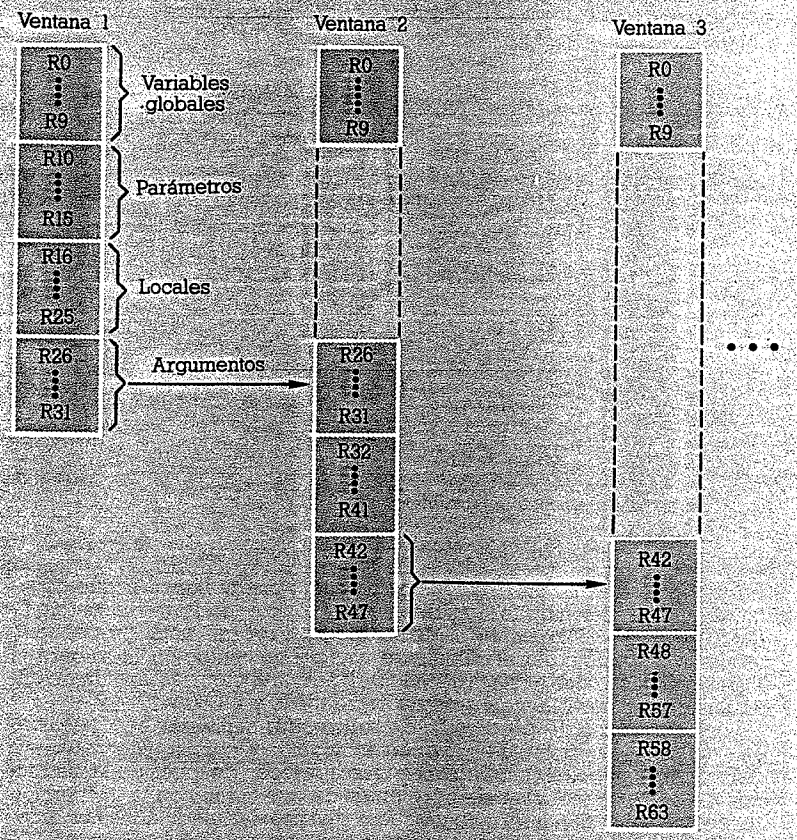


Fig. 3. Organización de ventanas de registros.

tiempo dedicado a compilación no es en absoluto despreciable.

3.2. RISC (Reduced Instruction Set Computer)

Un diseño de tipo RISC intenta minimizar el tiempo medio de ejecución de una instrucción decrementando los ciclos por instrucción (nc), acercándolo a 1, y a menudo decrementando el tiempo de ciclo (tc). Para programas compilados, el diseñador RISC estima ~~tiene la esperanza de~~ que el decremento en el tiempo medio de ejecución influye más en la eficiencia total que el número de instrucciones necesarias.

Algunas de las características clave en el diseño RISC son:

- Una instrucción cada ciclo. Para que la mayoría de instrucciones se ejecuten en un ciclo máquina, es necesario obtener una nueva instrucción cada ciclo desde la memoria, aunque la ejecución de la instrucción requiera varios ciclos. Las técnicas de segmentación y anticipación, así como una jerarquía de memorias de altos rendimientos, que incluya banco de registros y memoria *cache*, son obligatorias.
- Simplicidad del hardware. Sólo las funciones que incrementan significativamente la eficiencia son implementadas en hardware. Si no existe gran diferencia entre las soluciones hardware y software, se prefiere el software en tiempo de compilación-optimización.
- Arquitectura del tipo *load/store* (registro-registro). Sólo las instrucciones cargar (*load*) y almacenar (*store*) son usadas para acceder a memoria y todas las operaciones se realizan sobre registros. El conjunto de instrucciones se diseña de forma regular, con pocos formatos de instrucción y sin muchos modos de direccionamiento. Lo más importante del conjunto de instrucciones es su bajo nivel semántico.

Este tipo de instrucciones, acompañado del uso del banco de registros, permite que las operaciones de decodificación, obtención de operandos, cálculo y escritura de resultados sean muy rápidas, haciendo posible la reducción del tiempo de ciclo del procesador. De otro lado, la lógica de control y decodificación puede ser cableada, con poco o

nada de microcódigo.

Un efecto colateral de todas estas características es el mayor aprovechamiento del espacio de chip, con el 10 % de área de control y 41.000 transistores en la arquitectura RISC-II. La sencillez del hardware permite, utilizando la tecnología VLSI, encapsular en un solo chip todo un procesador. Piénsese, por ejemplo, en máquinas CISC: WE32000 necesita 3 chips y DEC VLSI VAX 9 chips. Si la tecnología se basa en el arseniuro de galio (GaAs), donde el poder de integración es pequeño, se hace necesaria la filosofía RISC.

4. Diferencias entre CISC y RISC

El calificativo de *reducida*, que se aplica a la filosofía RISC, no debe inducir a error, ya que no se refiere al número de instrucciones sino a la baja complejidad de las mismas. El número de instrucciones no resulta un discriminante fundamental entre los conceptos RISC y CISC. Creemos que la diferencia cualitativa se debe al objetivo de complejas que poseen las instrucciones de la filosofía CISC, o lo que es lo mismo, la arquitectura RISC se diferencia de la CISC por el bajo nivel semántico (baja complejidad) de sus instrucciones. Asimismo, no es razonable distinguir los dos tipos de arquitectura analizando exclusivamente las técnicas que se han incluido en su diseño (segmentación, adelantamiento, memoria *cache*, ventana de registros, etc.), ya que esta diferencia no pasa de ser cuantitativa.

5. Resumen

Acabaremos esta breve exposición destacando la coherencia de diseño que presenta la filosofía RISC. A diferencia de la arquitectura CISC, no suele ser de propósito general sino que intenta optimizar determinadas aplicaciones. De ahí que el conjunto de instrucciones y el camino de datos del procesador se determine a la luz del estudio estadístico de los programas de aplicación, los lenguajes de alto nivel que se van a utilizar, el estado del arte en compiladores y optimizadores y la tecnología hardware disponible. A la hora de incluir una técnica, el diseñador tiene siempre en mente la ganancia que se consigue por implementarla en hardware y la pérdida de eficiencia motivada por el aumento de la complejidad en el hardware, ya que la inclusión del hardware para aumentar la eficiencia de algunas funciones puede ir en detrimento de las otras.

6. Bibliografía recomendada

A continuación se presentan tres artículos que pueden servir de base a un estudio más profundo del RISC.

Petterson D.A.: Reduced Instruction Set Computers.
Communications of the ACM, vol. 28, No.1, Ene. 1985, pp 8-21.
Este artículo es una presentación del RISC por parte del profesor Patterson, uno de los diseñadores de RISC-I y RISC-II.

C.E. Gimarc and V.M. Multinovic: A survey of RISC processors and Computers of the Mid 1980's.
IEEE Computer, vol. 20, No. 9, Sep. 1987, pp 59-69.
En este artículo se exponen los campos de aplicación y características de los principales procesadores RISC que existen en el mercado.

R.P. Colwell et al.: Computers, Complexity and Controversy.
IEEE Computer, vol. 18, No. 9, Sep. 1985, pp 8-19.
Se trata de un artículo donde aparece la controversia entre la filosofía RISC y CISC. AeI

Clemente Rodríguez y Olatz Arregi son profesores de la Facultad de Informática de San Sebastián.