Tema 8

Procesadores Superescalares: Paralelismo Explícito a Nivel de Instrucción

IA-64 es una arquitectura de 64 bits desarrollada conjuntamente por Intel y HP (Hewlett-Packard). Está basado en una tecnología denominada EPIC (*Explicitally Parallel Instruction Computing*, procesamiento de instrucciones explícitamente paralelas), que abarca los siguientes conceptos básicos:

- Paralelismo explícito a nivel de instrucciones (explícito en las instrucciones máquina en lugar de ser identificado por el procesador en tiempo de ejecución).
- Palabras de instrucción muy grandes (VLIW).
- Ejecución de saltos basada en predicados.
- Carga especulativa.

En este tema utilizaremos la arquitectura del IA-64 para recorrer diversos aspectos organizativos de la arquitectura de los procesadores superescalares explícitamente paralelos.

8.1 Introducción a la arquitectura del IA-64

A continuación resumimos las diferencias principales entre una arquitectura superescalar y la arquitectura IA-64:

- En la implementación superescalar cada instrucción es una palabra. En la IA-64 se tienen instrucciones VLIW formadas por paquetes de 3 instrucciones de tipo RISC.
- Ambas implementaciones trabajan con múltiples unidades funcionales, si bien en la arquitectura IA-64 suelen ser más numerosas.
- La implementación superescalar utiliza predicción de saltos con ejecución especulativa de un camino, mientras que la IA-64 utiliza la ejecución especulativa de los dos caminos de una bifurcación.
- En la implementación superescalar se realiza el análisis del paralelismo, reordenación y optimización del flujo de instrucciones en tiempo de ejecución, mientras que la IA-64 lo hace en tiempo de compilación.

- En la implementación superescalar se realiza la carga de un dato desde memoria sólo cuando es necesario, mientras que en la IA-64 se pueden cargar datos especulativamente.

Cabe remarcar la diferencia fundamental entre ambas aproximaciones: *si bien un procesador superescalar analiza el paralelismo en tiempo de ejecución, la arquitectura IA-64 traslada todo ese trabajo al compilador, que lo hace en tiempo de compilación.*

Una desventaja del paralelismo explícito es que la programación en ensamblador es casi imposible.

El primer producto de esta arquitectura recibió el nombre clave de Merced. Actualmente Intel comercializa su arquitectura IA-64 con el nombre de Itanium 2.

Para Intel, crear una nueva arquitectura, incompatible con la arquitectura x86, supone un decidido paso adelante motivado por los dictados de la tecnología. La familia x86 comenzó su andadura a finales de los años setenta. Conforme aumentaron las posibilidades de integración de transistores, los procesadores de la familia comenzaron a introducir segmentación y otras técnicas que contribuían al aumento de prestaciones. Sin embargo, mantener la compatibilidad hacia atrás de los nuevos procesadores de la familia comenzó a ser un lastre. Otros fabricantes aumentaron la velocidad utilizando primero las tecnologías RISC y posteriormente la implementación superescalar. Intel, sin embargo, anclado por compatibilidad a una arquitectura CISC que debía ser compatible hacia atrás, intentó con el Pentium utilizar moderadamente técnicas superescalares y después, con el Pentium Pro y el Pentium II, tuvo que incorporar una traducción de las instrucciones CISC a microinstrucciones RISC para poder utilizar técnicas superescalares más eficazmente. Actualmente, una forma eficaz de utilizar la gran cantidad de transistores disponibles es la utilización de técnicas de paralelismo explícito, para lo cual Intel y HP han decidido crear una nueva arquitectura, independiente de que Intel mantenga la línea de procesadores de la popular familia x86.

Cuando los diseñadores han de decidir a qué dedicar los cada vez más transistores que es posible integrar en para la fabricación de un procesador, tienen pocas posibilidades. Por un lado, incrementar las cachés internas llega a un punto a partir del cual no mejora la tasa de aciertos de forma significativa. Por otro lado se puede incrementar el grado de paralelismo superescalar, añadiendo más unidades de ejecución, sin embargo la complejidad y la circuitería necesaria aumentan exponencialmente. Complejidad que sirve para mejorar muy poco las prestaciones de la máquina, ya que, como vimos en el tema anterior, en la implementación superescalar se pueden emitir 5 o más instrucciones en no más del 10% de los casos (en las condiciones más favorables).

Para solucionar este problema, Intel y HP han apostado por la implementación de un gran número de unidades funcionales en paralelo, pero utilizando el enfoque del paralelismo explícito. En esta estrategia, el compilador planifica estáticamente las instrucciones en tiempo de compilación, en lugar de que lo haga dinámicamente el procesador en tiempo de ejecución. Así, es el compilador el que determina qué instrucciones pueden ejecutarse en paralelo, e incluye esa

información en la instrucción máquina, liberando de dicha labor al procesador. Una ventaja de esto es que el procesador no requiere tanta circuitería compleja como un procesador superescalar con paralelismo implícito. Además, mientras que el procesador tiene que determinar la posibilidad de una potencial ejecución en paralelo en cuestión de nanosegundos, el compilador dispone de un plazo varios órdenes de magnitud mayor para examinar el código con detenimiento, y puede considerar el programa en toda su extensión.

8.2 Organización

La organización de la arquitectura IA-64 se muestra, esquemáticamente, en la figura 8.1. Las características más importantes son:

- Gran número de registros. El formato de instrucción de la arquitectura IA-64 permite la utilización de hasta 256 registros de 64 bits, 128 de uso general para datos enteros y lógicos y 128 para datos en coma flotante y gráficos. También dispone de 64 registros predicado de un bit.

- Múltiples unidades de ejecución. Puede utilizar muchas más unidades paralelas que las máquinas superescalares típicas, de paralelismo implícito a nivel de instrucciones, que suelen soportar cuatro cauces. Típicamente se usan ocho o más unidades funcionales. P. ej., Itanium 2 tiene 11 unidades funcionales diferentes.

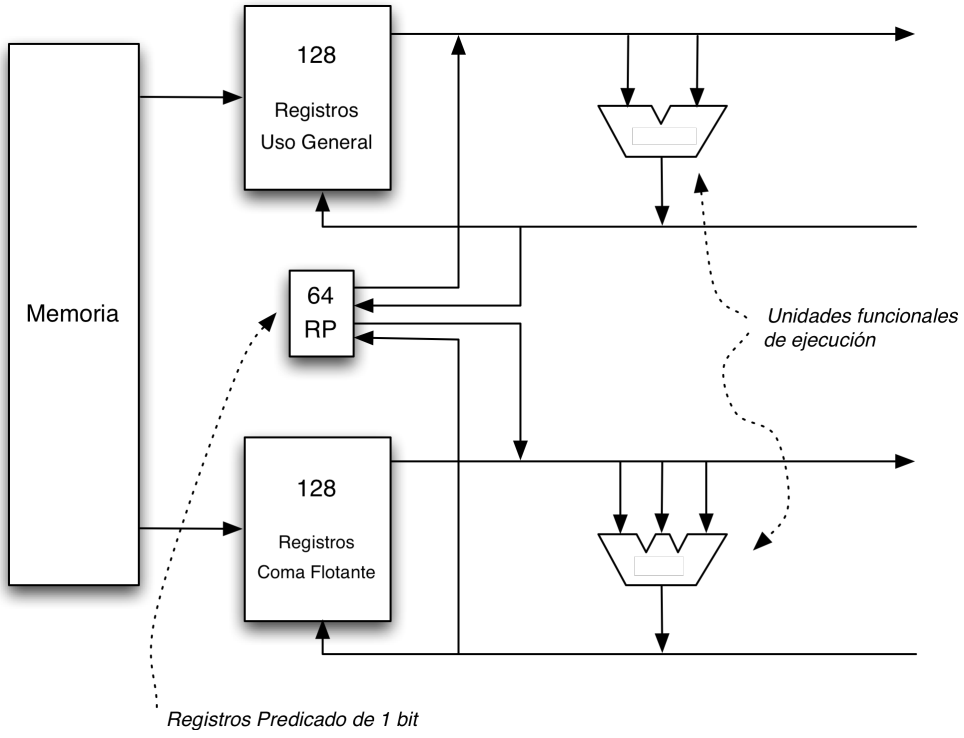


Figura 8.1: Esquema de organización de la arquitectura del IA-64.

El gran tamaño del banco de registros supone una diferencia importante respecto a las máquinas RISC y superescalares. Nótese que una máquina superescalar típica utiliza el paralelismo a nivel de instrucción de forma implícita, es decir la gestión del paralelismo a nivel de instrucción la realiza el procesador. Es por eso que este tipo de máquinas utiliza un mayor número de registros físicos que lógicos a través del renombramiento de registros. Sin embargo, el paralelismo explícito libera al procesador de la carga del renombramiento de registros y del análisis de dependencias. Esta tarea pasa a realizarla el compilador, que necesita utilizar un gran número de registros explícitos.

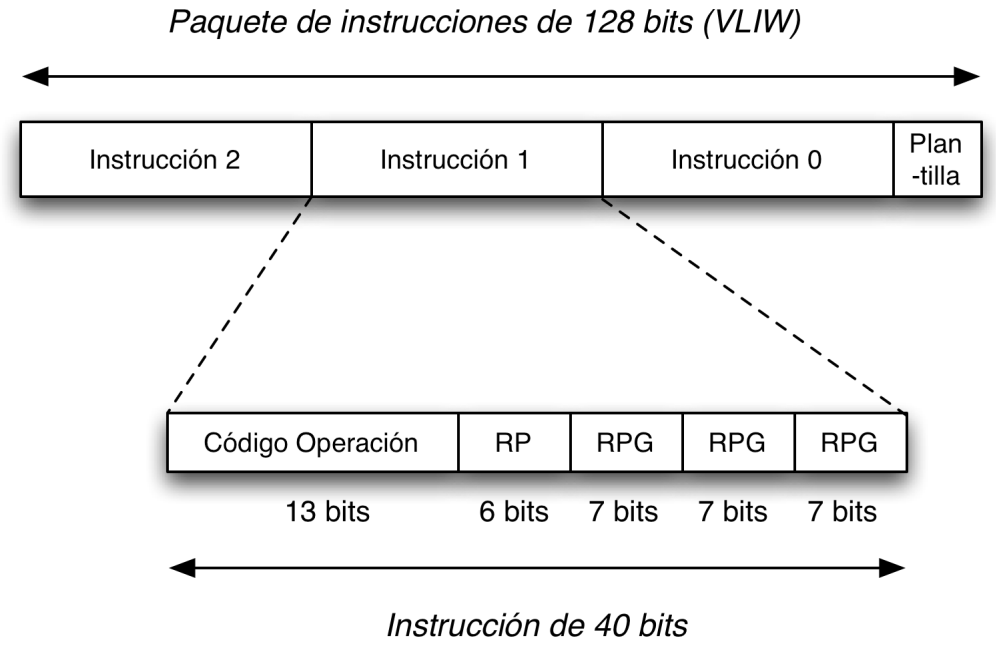


Figura 8.2 Formato de Instrucción de la arquitectura IA-64.

8.3 Formato de Instrucción

Las instrucciones de la arquitectura IA-64 responden al tipo de VLIW, *Very Long Instruction Word*, ya que cada instrucción es en realidad un paquete de 128 bits que contiene 3 instrucciones de 40 bits mas un campo plantilla de 8 bits, figura 8.2. El campo plantilla indica qué instrucciones se pueden ejecutar en paralelo, de forma que su interpretación no se limita a un solo paquete. Así, si por ejemplo el flujo de instrucciones permite la ejecución de ocho instrucciones en paralelo, el compilador las reordenará para que abarquen paquetes contiguos, y ajustará los bits de plantilla para informar al procesador cuales son las instrucciones independientes. Las instrucciones agrupadas no tienen por qué estar en el orden original.

El campo plantilla proporciona una gran flexibilidad. El compilador puede mezclar instrucciones dependientes e independientes en el mismo paquete, con tan solo indicarlo en el campo plantilla. Esto no ocurre en otros diseños VLIW, que necesitan insertar instrucciones NOP hasta completar el paquete para evitar que instrucciones dependientes compartan el mismo paquete.

La figura 8.2 muestra el formato de cada una de las instrucciones del paquete. La instrucciones son de 40 bits, lejos de los 32 bits típicos de las instrucciones RISC. Esto es debido a que un mayor número de registros implica más bits para codificarlos y, además, se añade la referencia a un campo predicado (cuya utilidad se explica en el siguiente apartado).

8.4 Ejecución con predicados

La ejecución con predicados es una técnica que posibilita que la ejecución de instrucciones pueda referirse a un registro predicado de 1 bit. Esto indica que la ejecución de dicha instrucción será especulativa, y una vez se sepan los valores de los registros predicado, la instrucción se retira si el valor final del registro predicado correspondiente es verdadero (valor 1, booleano TRUE), y se descarta si es falso. Por otro lado, todas aquellas instrucciones sin asignación de predicado se ejecutan incondicionalmente.

La siguiente sintaxis

```
<Pi>    Instruccion
```

indica la asignación de una predicado a una instrucción. Esta instrucción sólo se retirará si el valor del registro predicado Pi es 1. De lo contrario se descartará.

El valor de un predicado se obtiene a partir de la ejecución de la siguiente instrucción:

```
Pj, Pk = Expresión
```

Así, si la expresión es verdadera, entonces Pj = 1 y Pk = 0. Si es falsa Pj = 0 y Pk = 1.

También es posible que la anterior instrucción tenga, a su vez, asignado un registro predicado. A esto se le denomina asignación predicada de predicados:

```
<Pi> Pj, Pk = Expresión
```

La ejecución con predicados posibilita ejecutar en paralelo las dos ramas de una salto condicional antes de conocer la condición. De esta forma se utilizan al máximo las unidades funcionales del procesador con el propósito de evitar la introducción de ciclos de retardo.

EJEMPLO 1

Aquí tenemos un sencillo ejemplo de una bifurcación condicional:

```
if (a!=0)                P1,P2=EQ(R0,#0)
   j=j+1;                <P1> ADDI R2,R2,#1
else                     <P2> ADDI R1,R1,#1
   k=k+1;                // Las 3 ejecutan en 1 ciclo
```

Como vemos en el código ensamblador, dado que la bifurcación tiene dos posibles resultados, el compilador ha asignado un predicado diferente a las instrucciones de cada uno de los dos caminos. Las instrucciones del primer camino apuntarán al registro predicado P1, y la del segundo a la del P2. Cuando la CPU conoce el resultado de la comparación, en tiempo de ejecución, desecha los resultados de las instrucciones ejecutadas del camino no válido y entrega las del camino válido.

En este caso las tres instrucciones se podrían ejecutar en un solo ciclo, entregándose sólo los resultados de la instrucción correcta.

EJEMPLO 2

Otro ejemplo más elaborado es el del siguiente código fuente:

```
if (a&&b)
   j = j + 1;
else
   if (c)
      k = k + 1;
   else
      k = k - 1;
   end
end
i = i + 1;
```

En este ejemplo vemos cómo dos sentencias if seleccionan tres posibles caminos de ejecución. A continuación tenemos el código en lenguaje ensamblador que produciría un procesador típico:

```
     beq  a, 0, L1
     beq  b, 0, L1
     add  j, j, 1
     jump L3
L1:  beq c, 0, L2
     add  k, k, 1
     jump L3
L2:  sub k, k, 1
L3:  add  i, i, 1
```

Sin embargo el compilador del IA-64 genera el siguiente código con predicados:

```
(1)    P1, P2 = cmp (a == 0)
(2)   <P2>  P1, P3 = cmp (b == 0)
(3)   <P3>  add  j, j, 1
(4)   <P1>  P4, P5 = cmp (c != 0)
(5)   <P4>  add k, k, 1
(6)   <P5>  sub k, k, 1
(7)   add  i, i, 1
```

Las dos bifurcaciones condicionales del código ensamblador se convierten aquí en tres instrucciones de comparación. Si la primera instrucción pone P2 a falso, la segunda instrucción de comparación (I2) no se tiene en cuenta y es la instrucción de comparación I4 la que se tiene en cuenta. De otra forma, si P2 es verdadero se tiene en cuenta la instrucción de comparación en I2. Este es el procedimiento que finalmente determina cual es la instrucción que ha de ejecutarse, que será aquella de las I3, I5 o I6 cuyo predicado tenga un valor final verdadero.

Nótese que en un cauce superescalar normal utilizaríamos la predicción de saltos, y si la predicción es equivocada el cauce tendrá que vaciarse. Un procesador IA-64 inicia la ejecución de todas las instrucciones, y cuando se dispone de los valores de los predicados, se entregan sólo los resultados de la instrucción válida. Por tanto un procesador IA-64 utiliza las unidades de ejecución adicionales en paralelo para evitar los retardos debido al vaciado del cauce.

8.5 Carga especulativa

La carga especulativa es una técnica que permite al procesador cargar datos desde memoria antes de que el programa los necesite. De esta forma, evita los ciclos de retraso que se producen si la carga desde memoria se realiza en la instrucción anterior a la instrucción que la necesita. Puede verse que la función de la carga especulativa es similar a la carga retardada. Sin embargo, ¿Que diferencias hay entre ambas?

Como vemos, la figura 8.3a ilustra el diagrama de flujo de un fragmento de código. La instrucción I8 realiza una carga desde memoria de un operando que utiliza la instrucción I9. No obstante, para ahorrar ciclos de retraso se puede adelantar la carga lo máximo posible (es obvio que la carga adelantada no puede adelantar instrucciones anteriores que utilicen el mismo registro). Nótese que lo máximo que se podría hacer en este diagrama sería intercambiar la carga de la instrucción I8 con la instrucción I7 (si ésta no hiciese referencia al mismo registro que se carga). No podría adelantarse más porque se llega a una bifurcación condicional. En general la carga retardada tiene esta limitación.

Las cargas retardadas no adelantan una instrucción de bifurcación porque podría ocurrir que la carga no se produzca realmente si el flujo del programa toma otro camino en la bifurcación. En el caso de que el flujo tomara otro camino y que la carga adelantada hubiera generado una excepción o falta de página, el tiempo necesario para atender este evento es mucho mayor del que se gana utilizando una carga retardada. Este es el motivo de esta limitación.

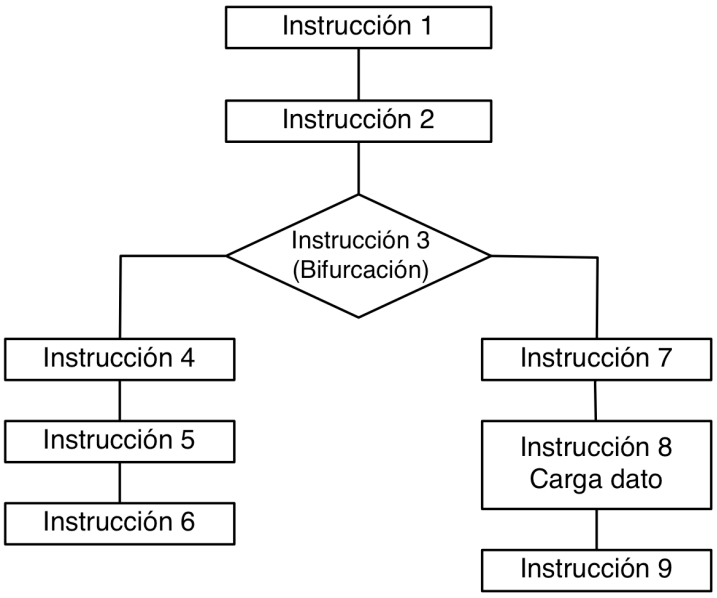
Sin embargo, la carga especulativa permite superar esta limitación permitiendo mover la carga por encima de una bifurcación. ¿Cómo lo hace? Separando el funcionamiento de la carga del de la excepción. Una instrucción de carga en el programa original se reemplaza, figura 8.3b, por dos instrucciones :

- Una carga especulativa (ld.s) ejecuta la captación desde memoria. Ésta lleva a cabo la carga y la detección de excepciones, pero, si la hubiere, no lanza la excepción. Esta instrucción se mueve a un punto que el compilador decida que sea adecuado.

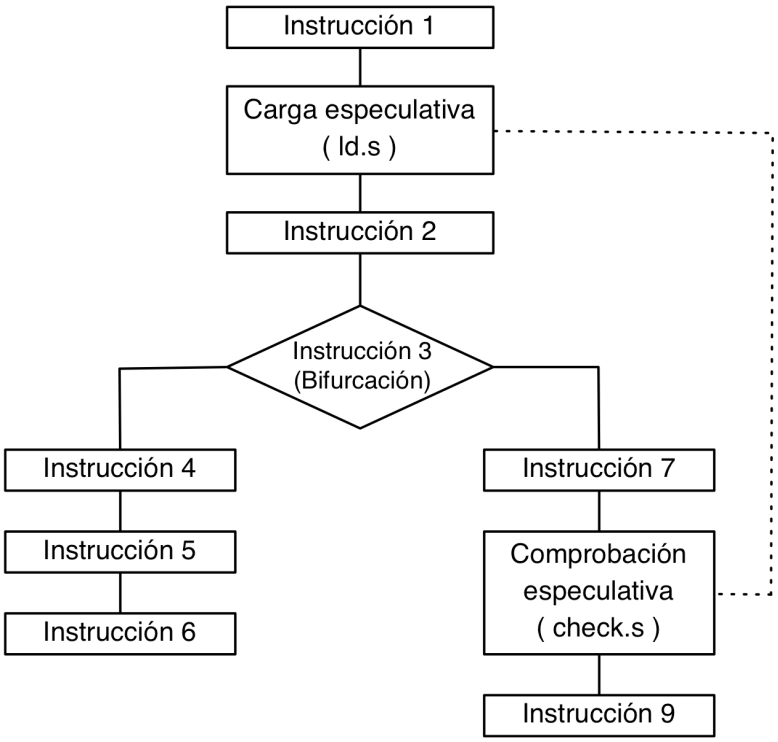
- Una instrucción de comprobación (check.s) permanece en el lugar de la carga original y lanza las excepciones. Esta instrucción podría asignarse a un predicado de forma que se ejecute sólo si el predicado es verdadero.

Como vemos, esta técnica permite cargar los datos desde memoria moviendo la carga por encima de una bifurcación, asegurando que, si la carga está en condiciones de producir una excepción, esta no se producirá, sino que se aplazará hasta encontrar la instrucción de comprobación.

Si ld.s detecta una excepción, ajusta un bit de recuerdo asociado con el registro destino, de forma que si la instrucción check.s se ejecuta y dicho bit está a uno, la instrucción check.s llama a la rutina de servicio de la excepción.



(a) Bifurcación original. Sin carga especulativa.



(b) Utilización de la carga especulativa.

Figura 8.3 Ejemplo de utilzación de la carga especulativa en el IA-64.

A continuación tenemos otro ejemplo con código que muestra cómo la utilización de la carga especulativa ha permitido mover la carga por encima de la bifurcación condicional. Nótese el uso de ld.s y check.s

```
      LD.s    R3, x       R3 ←x
      ADI     R0,#1       R0 ←R0 + 1;(integer)
      BZ      R0, L1
      ADI     R2, R2,#1   R2 ←R2 + 1;(integer)
      BR      L2
L1:   CHK.s   R3
      ADI     R3, R3,#1   R3 ←R3 + 1;(integer)
L2:   SBI     R4, R4,#1   R4 ←R4 - 1;(integer)
```