Procesadores de palabra de instrucción muy larga (VLIW)

v.2012

William Stallings, *Organización y Arquitectura de Computadores, 5ta. ed., Capítulo 13: Paralelismo a nivel de instrucciones y procesadores superescalares. 6ta. ed., Chapter 15: The IA-64 Architecture.*

John Hennessy – David Patterson, *Arquitectura de Computadores – Un enfoque cuantitativo 4ª Ed, Capítulos 2 y 3.*

Introducción

Explotación Superescalar del ILP

VENTAJAS

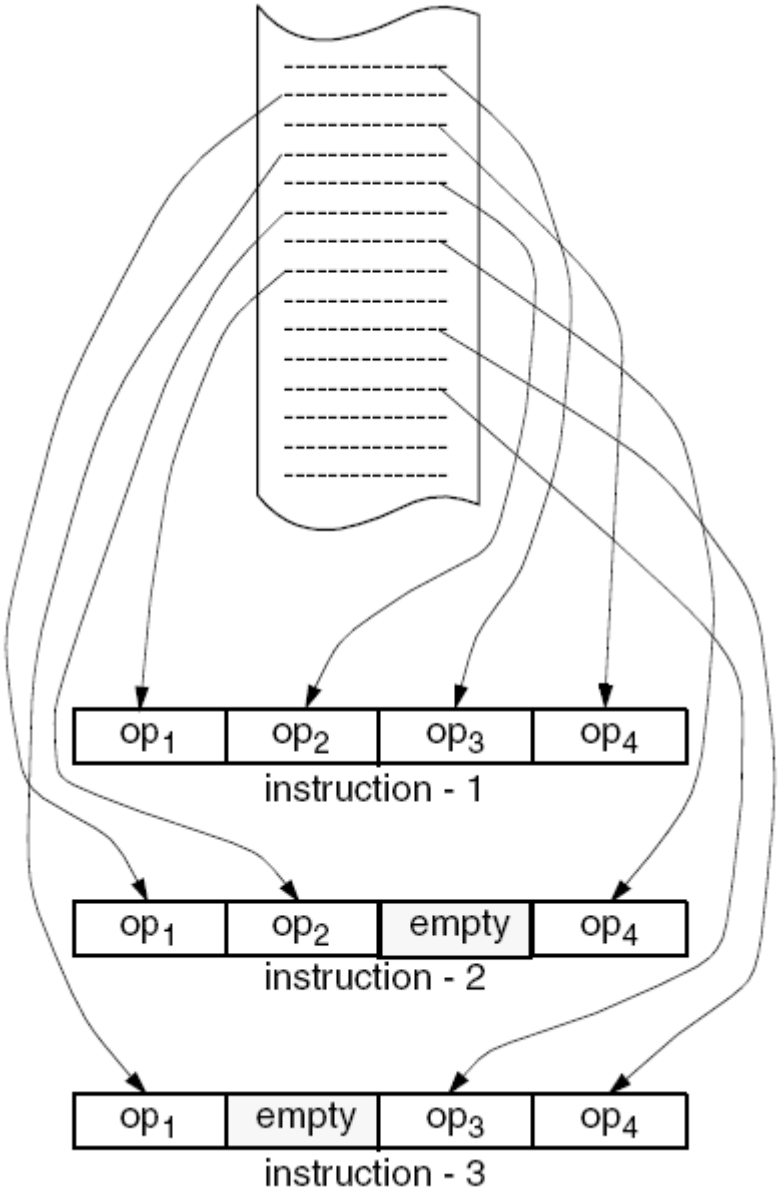
- El HW resuelve todo: detecta el paralelismo, emite, renombra registros, etc.
- Compatibilidad binaria: puedo agregar unidades funcionales sin cambiar el ISA.

DESVENTAJAS

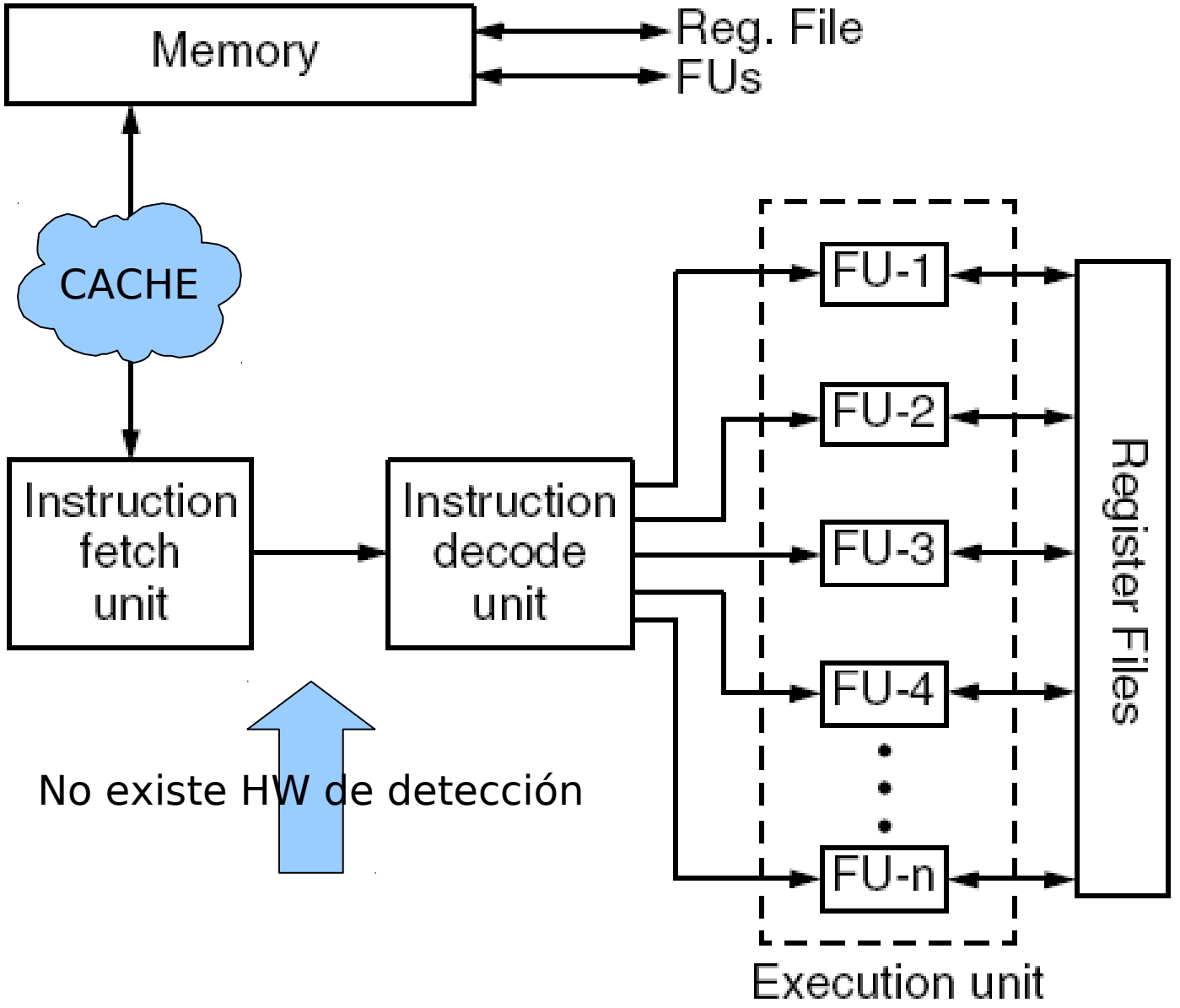
- Hardware muy complejo, se llega rápidamente a un límite.
- Ventana de ejecución limitada por el hardware, lo cual limita la capacidad de detección de instrucciones paralelas.

Introducción

La alternativa: VLIW



- Varias operaciones que pueden ser ejecutadas en paralelo (ILP) se empaquetan en una instrucción larga, una por cada UF disponible.
- La detección del paralelismo la hace el compilador (off-line).
- Luego de la captación y decodificación de una instrucción, las operaciones contenidas son emitidas en paralelo.
- Si el compilador no encuenbtra instrucciones, coloca NOP. Depende del ILP.


Memory
Reg. File
FUs
CACHE
Instruction fetch unit
Instruction decode unit
FU-1
FU-2
FU-3
FU-4
FU-n
Register Files
Execution unit
No existe HW de detección

Introducción

Ventajas VLIW

- Hardware simple.
- Puede incrementarse el número de unidades funcionales (UFs) sin agregar hardware de detección adicional.
- Bajo consumo de potencia.
- Los buenos compiladores pueden detectar paralelismo según un análisis global del programa completo, por lo tanto **se elimina el problema de la ventana de ejecución**.

Desventajas VLIW

- ✗ **Gran número de registros visibles**, necesarios para mantener las UFs activas (operandos y resultados).
- ✗ **Gran capacidad de transporte** entre las UFs y los registros, y entre los registros y la memoria.
- ✗ **Gran ancho de banda** entre el cache de instrucciones y la unidad de captación, debido a las instrucciones largas.
- **Baja densidad de código**, debido a las operaciones vacías.
- **Incompatibilidad binaria**: si agrego UFs, el número de operaciones simultáneas se incrementa, aumenta el ancho de la instrucción, y por lo tanto el código binario anterior no se puede ejecutar en la nueva máquina.

Principales técnicas utilizadas por el compilador

El compilador debe alterar la secuencialidad del código (sin alterar el resultado) para mejorar el ILP, y explotar así el paralelismo disponible en la máquina (MLP).

El problema, como siempre, son los saltos condicionales.

Estas técnicas son útiles en superescalares, pero acá son **fundamentales**, ya que no existe detección por hardware. Las principales son:

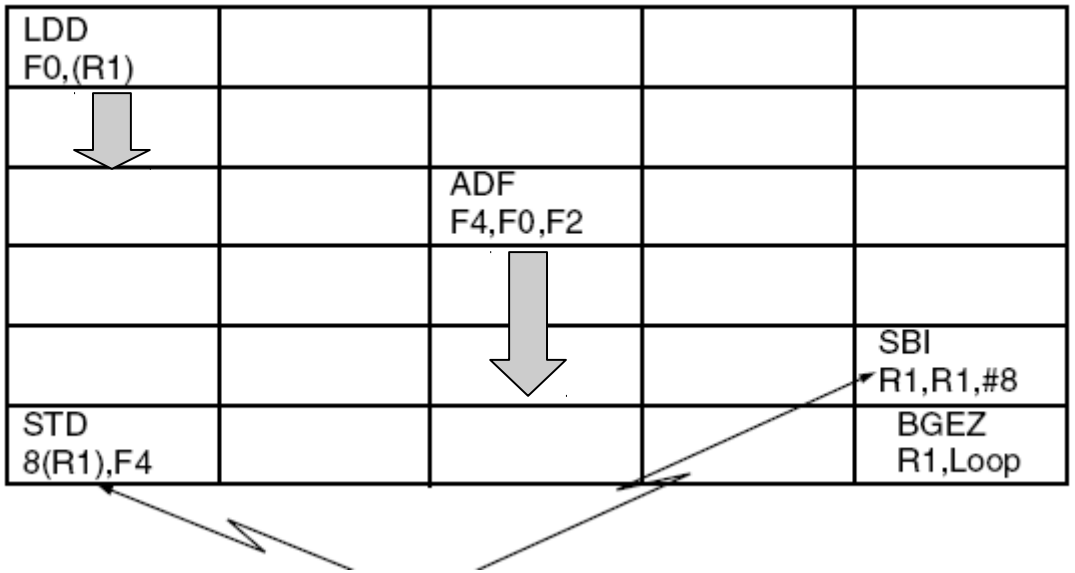
- **Loop Unrolling (para repeticiones [FOR/WHILE])**
- **Trace Scheduling (para bifurcaciones [IF/ELSE])**

Recordar el teorema de **Dijkstra**, base de la programación estructurada
Secuencia, Bifurcación y Repetición

Loop Unrolling - Ejemplo

```
for(i=959;i>=0;i--)
        x[i] = x[i] + s;
```

```
Loop: LDD    F0, (R1)     F0 ←x[i] ;(load double)
      ADF    F4,F0,F2     F4 ←F0 + F2 ;(floating pnt)
      STD    (R1),F4      x[i] ←F4 ;(store double)
      SBI    R1,R1,#8     R1 ←R1 - 8
      BGEZ   R1,Loop
```



Desplazado 8 porque ya se hizo la resta

Datos: *x* es un array y *s* una cte., ambos en punto flotante doble precisión (64 bits).

UF segmentadas:
- 2 memoria (2c para LOAD o STORE double)
- 2 FP (3c para ADD double)
- 1 enteros/salto (1c)

R1 contiene la dirección del último elemento de *x*, y F2 contiene el valor de *s*.

Secuencial: 7680c (960x8)

VLWI: 5760c (960x6)

S=8/6=1,33

Loop unrolling x2

```
for(i=959;i>=0;i-=2){
        x[i] = x[i] + s;
        x[i-1] = x[i-1] + s;}
```

```
Loop: LDD    F0, (R1)     F0 ←x[i] ;(load double)
      ADF    F4,F0,F2     F4 ←F0 + F2 ;(floating pnt)
      STD    (R1),F4      x[i] ←F4 ;(store double)
      LDD    F0, -8(R1)   F0 ←x[i-1] ;(load double)
      ADF    F4,F0,F2     F4 ←F0 + F2 ;(floating pnt)
      STD    -8(R1),F4    x[i-1] ←F4 ;(store double)
      SBI    R1,R1,#16    R1 ←R1 - 16
      BGEZ   R1,Loop
```

LDD F0,(R1)	LDD F6,-8(R1)			
		ADF F4,F0,F2	ADF F8,F6,F2	
				SBI R1,R1,#16
STD 16(R1),F4	STD 8(R1),F8			BGEZ R1,Loop

VLIW utiliza más Registros Explícitos (F6).

Secuencial: 6720c (14x480)

VLWI: 2880c (6x480)

S=14/6=2,6
[Sacc=7680/2880=2.66]

Mejora también el secuencial!

Loop unrolling x3

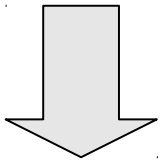
```
for(i=959;i>=0;i-=3){
        x[i] = x[i] + s;
        x[i-1] = x[i-1] + s;
        x[i-2] = x[i-2] + s;
}
```

LDD F0,(R1)	LDD F6,-8(R1)			
LDD F10,-16(R1)				
		ADF F4,F0,F2	ADF F8,F6,F2	
		ADF F12,F10,F2		
STD (R1),F4	STD -8(R1),F8			SBI R1,R1,#24
STD 8(R1),F12				BGEZ R1,Loop

Secuencial: 6400c (20x320)

VLWI: 2240c (7x320)

S=20/7=2,85



Unrolling x4: 1680c

Unrolling x6: 1280c

Loop unrolling x8

```
for(i=959;i>=0;i-=2){
        x[i]   = x[i] + s;   x[i-1] = x[i-1] + s;
        x[i-2] = x[i-2] + s; x[i-3] = x[i-4] + s;
        x[i-4] = x[i-4] + s; x[i-5] = x[i-5] + s;
        x[i-6] = x[i-6] + s; x[i-7] = x[i-7] + s;
}
```

LDD F0,(R1)	LDD F6,-8(R1)			
LDD F10,-16(R1)	LDD F14,-24(R1)			
LDD F18,-32(R1)	LDD F22,-40(R1)	ADF F4,F0,F2	ADF F8,F6,F2	
LDD F26,-48(R1)	LDD F30,-56(R1)	ADF F12,F10,F2	ADF F16,F14,F2	
		ADF F20,F18,F2	ADF F24,F22,F2	
STD (R1),F4	STD -8(R1),F8	ADF F28,F26,F2	ADF F32,F30,F2	
STD -16(R1),F12	STD -24(R1),F16			
STD -32(R1),F20	STD -40(R1),F24			SBI R1,R1,#64
STD 16(R1),F28	STD 8(R1),F32			BGEZ R1,Loop

Serían necesarios 2x(16+1) registros. Si tengo 32 no alcanza...

Notar que aumenta mucho el tamaño del programa (antes 5 inst, ahora 45).

VLWI: 1080c (S~=7)

Loop unrolling xn

TERMINAR:

Existe un nivel óptimo de unrolling.

Aceleración máxima vs. optimización de recursos.

Graficar S=f(n).

En este ejemplo debería resultar que el n óptimo es 10.

Cuánto vale Smax? Cuántos registros son necesarios?

PREGUNTA: Es correcto que con 5 unidades funcionales consiga S>5?

Trace Scheduling

Se trata de una técnica de compilación que realiza predicción de bifurcaciones (*compile time branch prediction*). [IF/ELSE]

Permite buscar instrucciones independientes más allá de **saltos condicionales.**

Se realiza en tres pasos:

1. **Trace selection**
2. **Instruction scheduling**
3. **Replacement and compensation**

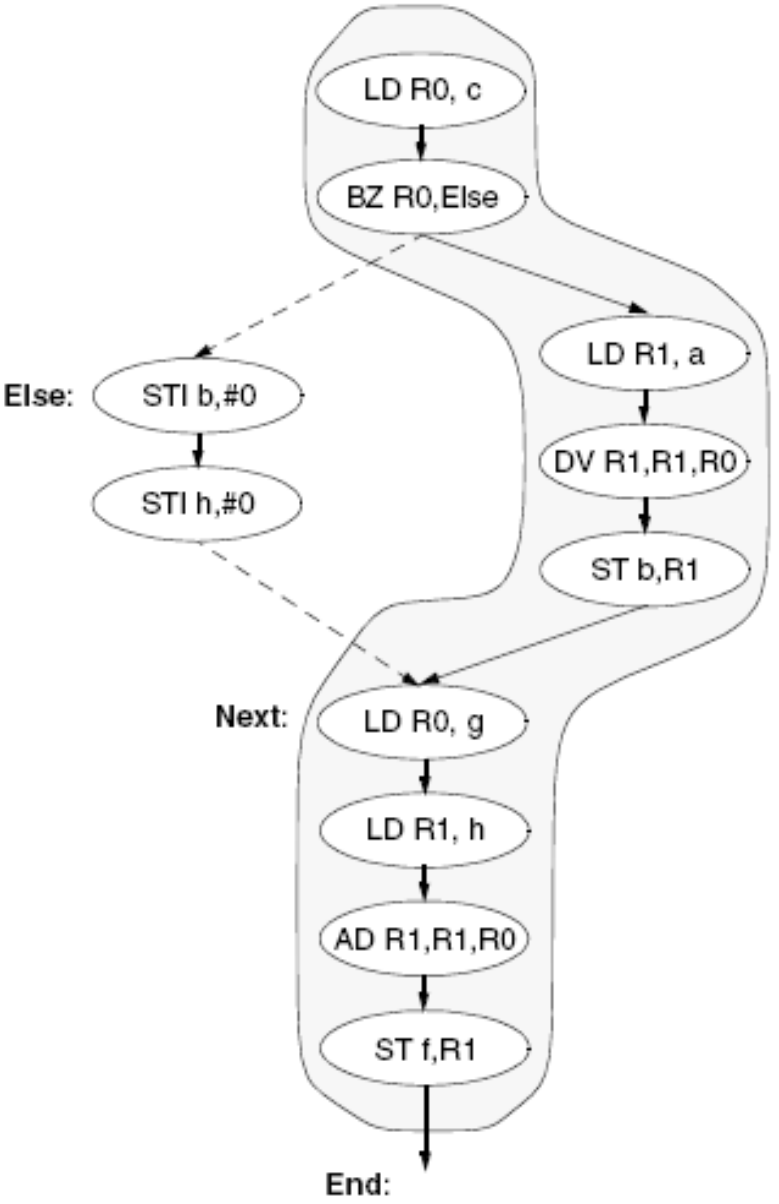
Trace Scheduling

Ejemplo

```
if(c!=0)b=a/c;
else{b=0; h=0;}
f=g+h;
```

```
        LD      R0, c       R0 ←c ;(load word)
        BZ      R0,Else
        LD      R1, a       R1 ←a ;(load integer)
        DV      R1,R1,R0    R1 ←R1 / R0 ;(integer)
        ST      b,R1        b ←R1 ;(store word)
        BR      Next
Else:   STI     b,#0        b ←0
        STI     h,#0        h ←0
Next:   LD      R0, g       R0 ←g ;(load word)
        LD      R1, h       R1 ←h ;(load word)
        AD      R1,R1,R0    R1 ←R1 + R0 ;(integer)
        ST      f,R1        f ←R1 ;(store word)
End:    ----------------
```

c!=0 ... 14c
c=0 ... 11c



Trace Scheduling

Ejemplo (cont)

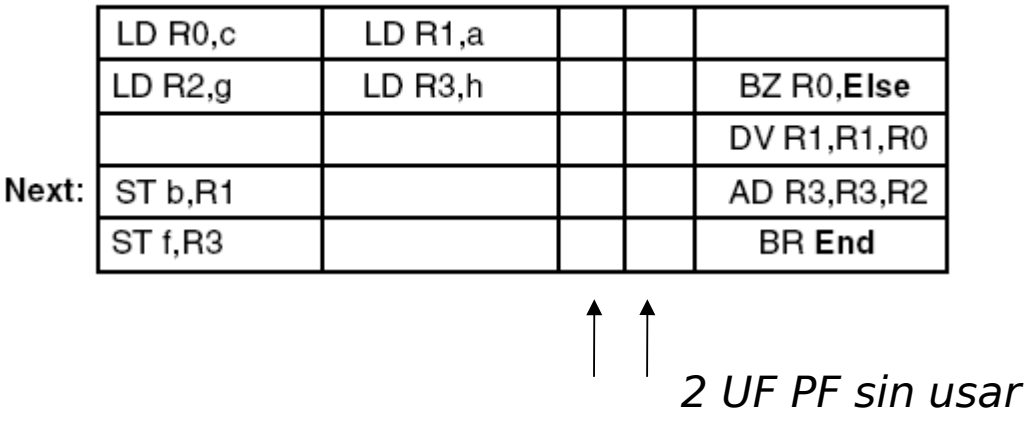
1. Trace selection

Se selecciona la secuencia mas probable (trace), utilizando predicción de saltos durante la compilación.

Se puede mejorar con profiling (ejecución del programa con secuencias típicas de entrada y recolección de estadísticas respecto de los saltos).

2. Instruction scheduling

Se reacomodan las operaciones del trace más probable en una nueva secuencia de instrucciones largas del procesador.



Trace Scheduling

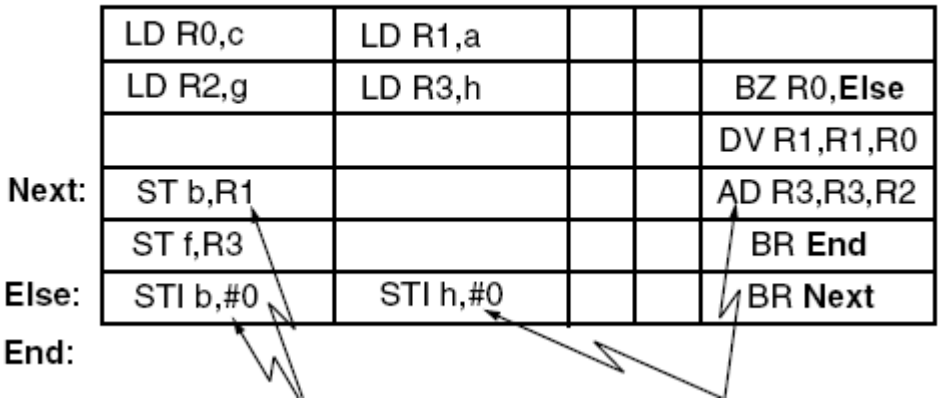
Ejemplo (cont)

3. Replacement and compensation

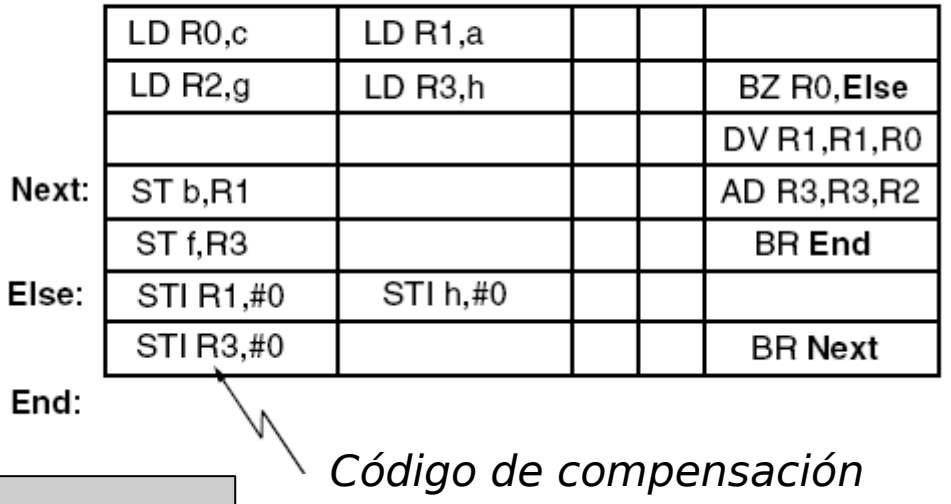
Se reemplaza el trace original con la nueva secuencia.

Se agrega código adicional a ejecutarse en caso que la predicción falle (no alcanza con agregar el trace no elegido!). El resultado debe ser el mismo.

MAL



BIEN



Si acertó ... 5c ... S=14/5=2.8
Si no acertó ... 6c ... S=11/6=1.8

Trace Scheduling

Características principales

No es lo mismo que la ejecución especulativa de los superescalares.

Es una optimización del compilador tal que el camino más probable se ejecute más rápido, a expensas de hacer más lento al menos probable.

Siempre se toma el camino correcto. Si no es el predicho por el compilador, la ejecución será más lenta a causa del código de compensación.

Esto no implica que el procesador VLIW no pueda utilizar predicción de saltos y ejecución especulativa para mejorar el rendimiento del pipeline.

Arquitecturas VLIW

Algunos ejemplos actuales

EMBEDDED:

- Philips Trimedia (nxp.com): DSP multimedia.
- Texas TMS320C6xxx (ti.com): DSP multimedia.

DESKTOP/SERVER:

- Transmeta Crusoe y Eficeon (transmeta.com): x86 compatible por Code Morphing, low power, mobile.
- Intel & HP: EPIC (Explicitly Parallel Instruction Computing) -> **Arquitectura IA-64** (Itanium e Itanium 2).

La arquitectura IA-64

Desarrollada por Intel y Hewlett-Packard. No es un VLIW puro. Lo llaman EPIC (Explicit Parallel Instruction Computing) processor.

Características VLIW:

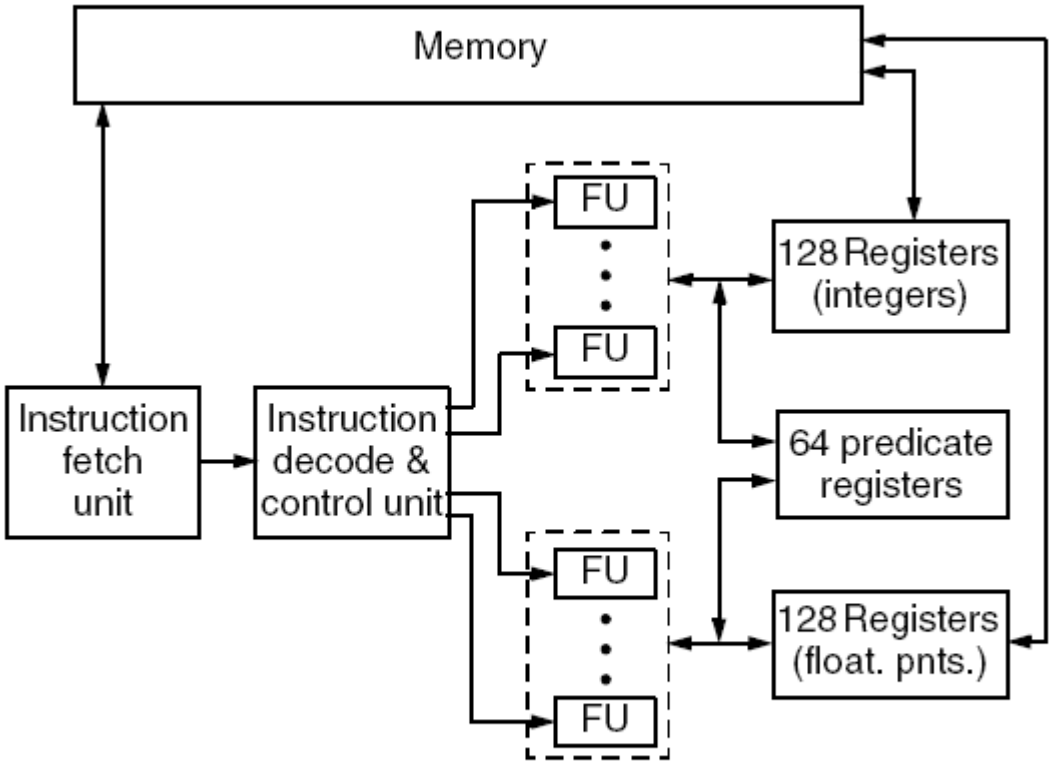
- Detección del ILP por compilador.
- Instrucción larga (128 bits con 3 operaciones).

Características propias:

- Template.
- Ejecución con predicados.
- Carga especulativa.

Arquitectura IA-64

Organización general



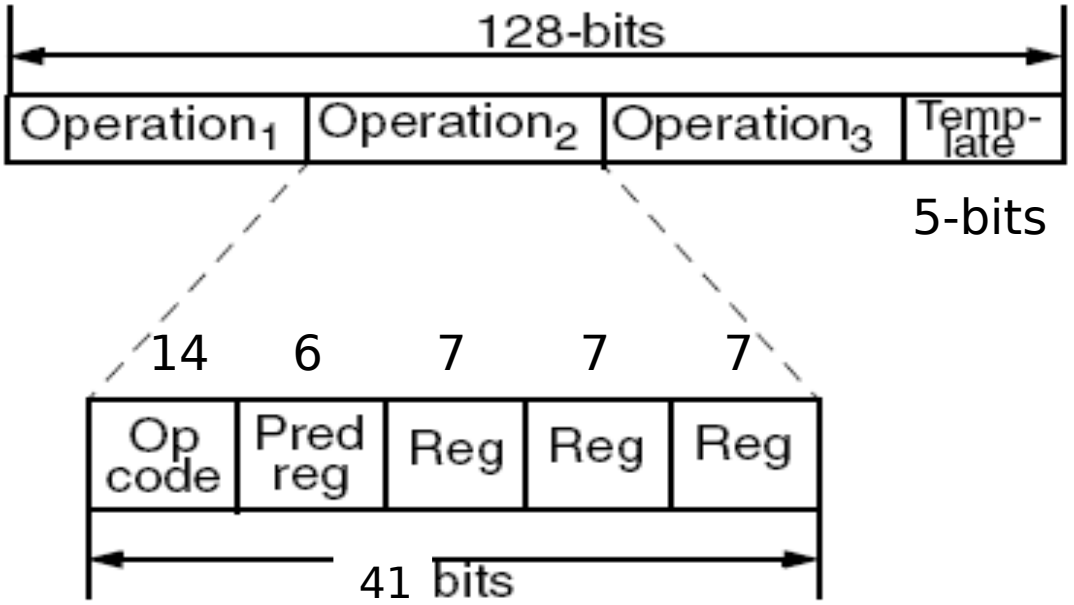
Los registros (enteros y fp) son de 64 bits.

Los registros de predicado son de 1 bit.

8 o más unidades funcionales.

Arquitectura IA-64

Formato de instrucciones



Cada instrucción contiene 3 operaciones de 41 bits y un template de 5. No quiere decir que las tres operaciones se puedan ejecutar en paralelo, eso lo indica el template. Cada operación tiene un código de 14 bits, una dirección de registro de predicado de 6 bits (para los 64 registros) y 3 direcciones de registros de 7 bits (tiene 128 registros).

Arquitectura IA-64

Template

La 'plantilla' indica qué instrucciones pueden ejecutarse en paralelo, conectando a su vez con instrucciones vecinas, por lo que operaciones de diferentes instrucciones pueden ejecutarse simultáneamente.

No es necesario dejar operaciones sin utilizar (NOP), mejorando la densidad del código.

Mejora la compatibilidad binaria, ya que puedo agregar UFs sin cambiar las instrucciones.

Si, según el tremplate, existen más operaciones listas para ser ejecutadas que UFs disponibles, se ejecutan secuencialmente.

La programación en assembler es casi imposible.

Arquitectura IA-64

Ejecución con predicado

Cada operación puede referirse a un predicado (registro de 1 bit). Sólo se realiza el commit de la operación si el predicado vale 1. Una operación sin predicado se ejecuta incondicionalmente.

Sintaxis:

```
<Pi> INSTRUCCION
```

Asignación de predicados: Si la expresión es verdadera Pj=1,Pk=0. Si es falsa Pj=0,Pk=1.

```
Pj,Pk = Expresión.
```

Asignación predicada de predicados:

```
<Pi> Pj,Pk = Expresión.
```

Ejemplo de utilización: Branch predication.

Arquitectura IA-64

Branch predication

No confundir con branch prediction. Es una forma de ejecución condicional.

Se dejan ejecutar en paralelo las dos ramas de un salto condicional, antes de conocer la condición. Se hace el commit de la rama correcta (con predicados) al conocerse la condición.

Un paso más allá de trace scheduling, en que se perdía tiempo adicional si se selecciona el trace incorrecto.

Técnica de compilación muy agresiva.

```
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
```

Arquitectura IA-64

Carga especulativa

Permite adelantar cargas desde memoria, incluso fuera de los límites del bloque, y manejarla más eficientemente si produce un *page fault*.

```
        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)
```

Conclusiones

La arquitectura presenta algunas características interesantes que merecen ser resumidas:

Tienen un muy buen rendimiento con baja frecuencia de clock, lo que le permite trabajar a bajas temperaturas.

Tienen una gran memoria cache (12 MB/core), lo cual es posible gracias a la arquitectura simple.

Con esta arquitectura Intel intenta introducir sus productos en el mercado de servidores high-end, liderado por IBM con su arquitectura Power y por Sun con SPARC.

Al contrario de estas dos arquitecturas, el diseño no es OPEN.

*Ver '*Software Development Manual*' (PDF) y comparar con Transmeta Crusoe (PDFx2) y su Code Morph: a) Upgreadable, b) sw power tuning, y c) morph a otras arquitecturas.*

APENDICE

Ejecución condicional en ARM

Todas las instrucciones tienen un campo de condición de 4 bits (PREDICADO), por lo tanto son todas condicionales. Tiene sólo dos instrucciones de salto (B y BL).

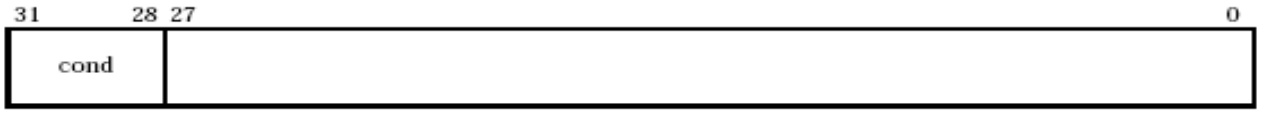


Table 3-1 Condition codes

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0,N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-
1111	(NV)	See *Condition code 0b1111* on page A3-5	-

APENDICE

Ejecución condicional en ARM

Todas las instrucciones se pueden transformar en condicionales.
Mejora notablemente el rendimiento de la segmentación.

En C:

```
while(i != j) {
        if (i > j)
            i -= j;
        else
            j -= i;
    }
```

En assembler de ARM:

```
        loop  CMP  Ri, Rj    ; set condition "NE" if (i != j),
                             ;               "GT" if (i > j),
                             ;            or "LT" if (i < j)
        SUBGT  Ri, Ri, Rj    ; if "GT" (greater than), i = i-j;
        SUBLT  Rj, Rj, Ri    ; if "LT" (less than), j = j-i;
        BNE  loop            ; if "NE" (not equal), then loop
```