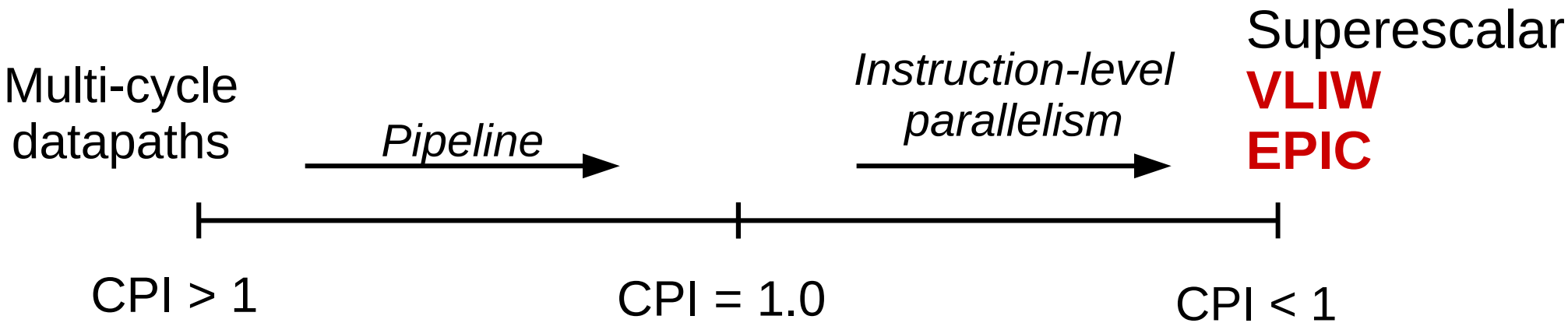
Arquitectura de Computadoras para Ingeniería

(Cód. 7526)
1° Cuatrimestre 2016

Dra. Dana K. Urribarri
DCIC - UNS

VLIW/EPIC

Instruction-level parallelism



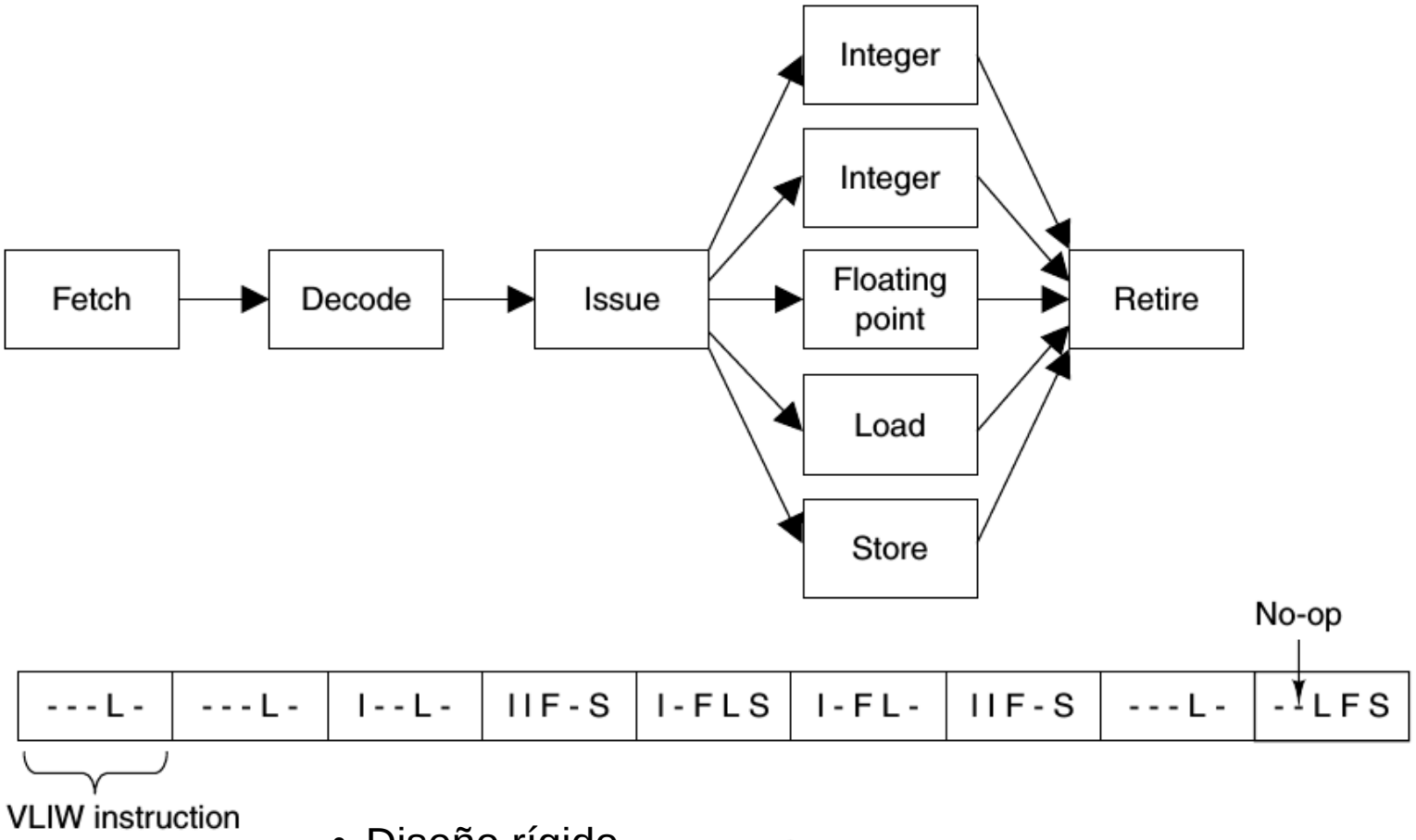
Categoría de Arquitecturas ILP

	Agrupamiento de instrucciones para ejecución paralela	*Asignación de las instrucciones a las UF en el hardware*	*Determinación del inicio de la ejecución de las instrucciones*
Superescalar	HW	HW	HW
	...	...	...
VLIW	Compilador	Compilador	Compilador

Very Long Instruction Word (VLIW)

- Planificación estática.
- El ISA VLIW define patrones para instrucciones largas.
- Una instrucción larga especifica múltiples operaciones que pueden realizarse simultáneamente.
  - Por ejemplo: un acceso a memoria, una operación aritmética de enteros y otra de punto flotante.
- Depende fuertemente del compilador
  - El compilador debe unificar en una única instrucción operaciones que puedan ejecutarse en paralelo *sin conflicto*.
  - Si no es posible, se completa con NOP.

Instrucciones largas



- Diseño rígido.
- Las instrucciones se asignan a una UF ubicándola en un determinado campo (*slot*) dentro de la instrucción larga.
- NOPs como relleno cuando no es posible usar todas las UF a la vez.

Very Long Instruction Word (VLIW)

- Una secuencia de instrucciones largas definen el *plan de ejecución* para un programa en particular en una implementación en particular.
- Problema de compatibilidad entre implementaciones:
  - Un programa es compilado para una microarquitectura en particular (determinado conjunto de unidades funcionales con determinadas latencias).

Very Long Instruction Word (VLIW)

- Técnicas de ILP en tiempo de compilación
  - ✔ Se puede tener en cuenta un mayor rango de instrucciones.
  - ✔ Se realiza una única vez y por lo tanto puede demandar tanto tiempo como sea necesario.
  - ✗ Solo se cuenta con la información disponible en tiempo de compilación: estrategias conservativas y que asumen el peor caso.

Categoría de Arquitecturas ILP

	Agrupamiento de instrucciones para ejecución paralela	*Asignación de las instrucciones a las UF en el hardware*	*Determinación del inicio de la ejecución de las instrucciones*
Superescalar	HW	HW	HW
EPIC	Compilador	HW	HW
Dynamic VLIW	Compilador	Compilador	HW
VLIW	Compilador	Compilador	Compilador

Explicitly Parallel Instruction Computing (EPIC)

- Se basan en el concepto de VLIW
- El compilador sólo determina el agrupamiento de instrucciones independientes y lo comunica mediante instrucciones del *set* de instrucciones.
- Es compatible para diferentes implementaciones.
- Simplifica el hw porque no requiere que controle dependencias.

Dynamic VLIW

- El compilador agrupa las instrucciones y asigna las unidades funcionales.
- El hardware determina cuándo comienza a ejecutar cada instrucción.
- Permite responder a eventos que el compilador no puede manejar en tiempo de compilación (*fallos en cache*).

Técnicas para ↑ILP en compilación

- Loop Unrolling
- Instrucciones predicativas
- Control especulativo

Loop Unrolling

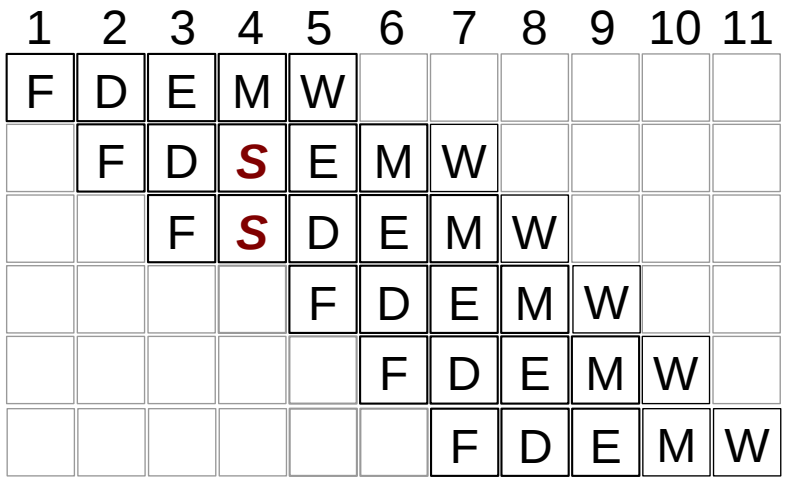
- Transformar los ciclos para aumentar el ILP.

```
For(i=999; i>=0; i--)
    x[i] = x[i] + s
```

```
// F2 ← s
// R1 ← dirección base del arreglo x
// R2 ← 1000
Loop: Ld F0, 0(R1)
      Add F4, F0, F2
      St F4, 0(R1)
      Add R1, R1, #8   //Incrementa el puntero R1
      Add R2, R2, #-1  //Decrementa en 1 la cantidad
      Bnez R2, Loop    //Salta si R2≠0
```

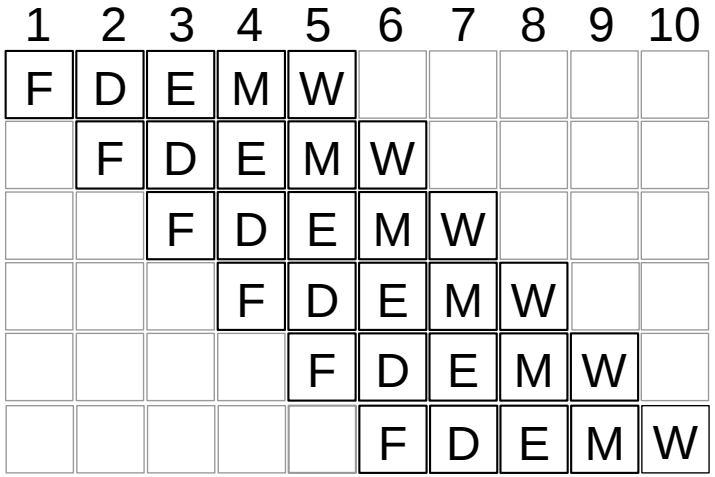
Loop Unrolling

```
Loop: Ld F0, 0(R1)
      Add F4, F0, F2
      St F4, 0(R1)
      Add R1, R1, #8
      Add R2, R2, #-1
      Bnez R2, Loop
```



Loop Unrolling

```
Loop: Ld F0, 0(R1)
      Add R2, R2, #-1
      Add F4, F0, F2
      St F4, 0(R1)
      Add R1, R1, #8
      Bnez R2, Loop
```



Loop Unrolling

La suma en PF lleva 4 ciclos en pipeline.

```
Loop: Ld F0, 0(R1)
      Add R2, R2, #-1
      Add F4, F0, F2
      St F4, 0(R1)
      Add R1, R1, #8
      Bnez R2, Loop
```

	1	2	3	4	5	6	7	8	9	10	11	12	13
Ld F0, 0(R1)	F	D	E	M	W								
Add R2, R2, #-1		F	D	E	M	W							
Add F4, F0, F2			F	D	E	E	E	E	M	W			
St F4, 0(R1)				F	D	***S***	***S***	***S***	E	M	W		
Add R1, R1, #8					F	***S***	***S***	***S***	D	E	M	W	
Bnez R2, Loop									F	D	E	M	W

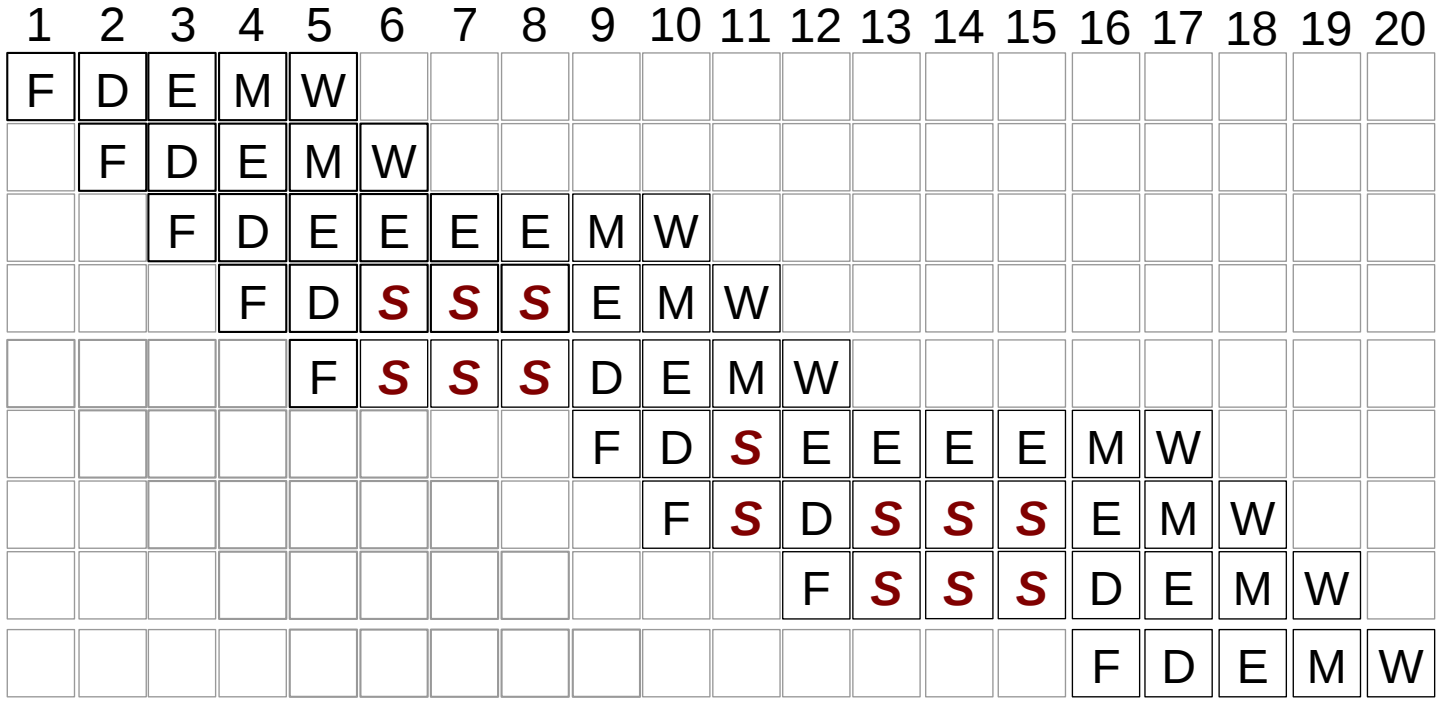
CPI = 13/6 = 2.17

Loop Unrolling

La suma en PF lleva 4 ciclos en pipeline.
Desenrollar 1 vez.

```
For(i=999; i>=0; i=i-2)
    x[i] = x[i] + s
    x[i-1] = x[i-1] + s
```

```
Loop: Ld   F0, 0(R1)
      Add  R2, R2, #-2
      Add  F4, F0, F2
      St   F4, 0(R1)
      Ld   F0, 8(R1)
      Add  F4, F0, F2
      St   F4, 8(R1)
      Add  R1, R1, #16
      Bnez R2, Loop
```



CPI = 20/9 = 2.22

Loop Unrolling

La suma en PF lleva 4 ciclos en pipeline.
Desenrollar 1 vez.

```
For(i=999; i>=0; i=i-2)
    x[i] = x[i] + s
    x[i-1] = x[i-1] + s
```

```
Loop: Ld F0, 0(R1)
      Add R2, R2, #-2
      Add F4, F0, F2
      St F4, 0(R1)
      Ld F6, 8(R1)
      Add F8, F6, F2
      St F8, 8(R1)
      Add R1, R1, #16
      Bnez R2, Loop
```

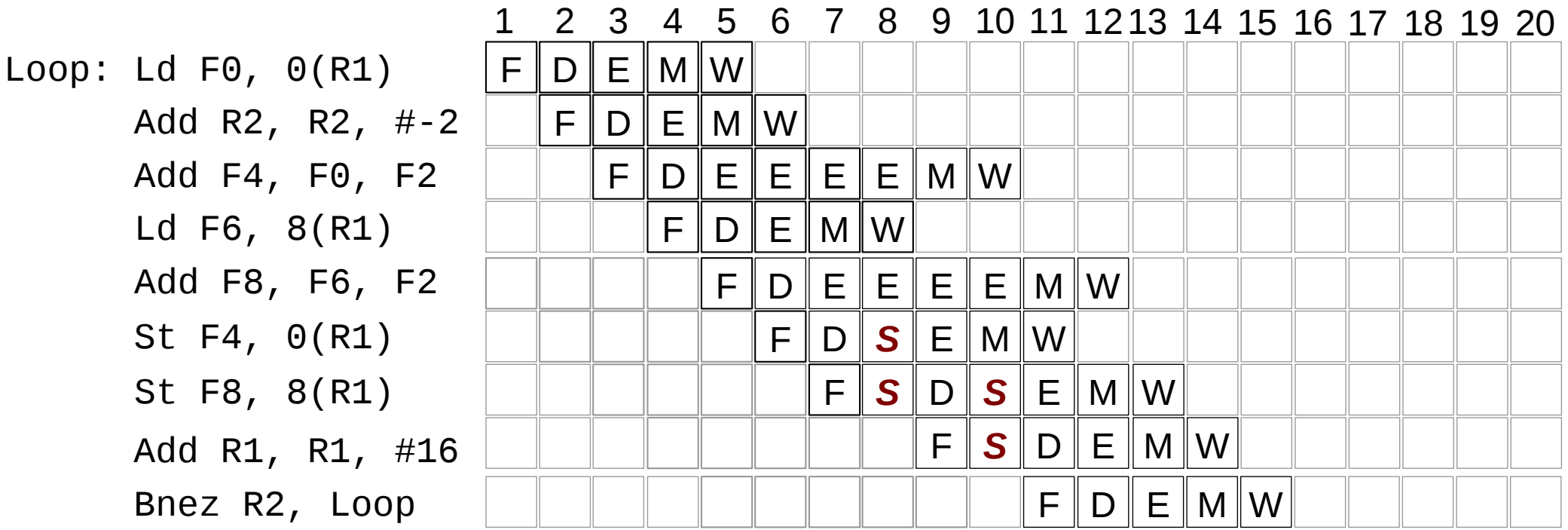
Usamos más registros

Reordenamos las instrucciones

Loop Unrolling

La suma en PF lleva 4 ciclos en pipeline.
Desenrollar 1 vez.

```
For(i=999; i>=0; i=i-2)
    x[i] = x[i] + s
    x[i-1] = x[i-1] + s
```



CPI = 15/9 = 1.67

Instrucciones predicativas

- Se puede reducir el número de *branchs* a ejecutar usando *predicados*.
- Las instrucciones predicativas van acompañadas de un predicado.
  - Si el predicado es verdadero se ejecuta la instrucción.
  - Si es falso, no se ejecuta.

```
If (R1==0) R2 ← R3
…
```

```
        Bnez R1, label
        Move R2, R3
Label: …
```

```
Cmovz R2, R3, R1 // R2 ← R3, si R1=0
…
```

Control Especulativo

- Equivalente a la predicción en superescalares
- Mover una porción de código arriba del *branch*.
- Estas instrucciones se vuelen *especulativas.*
  - No pueden terminar hasta que se sepa la condición del *branch*.

Orden Original

```
Branch label
Ld r1, 0(r9)
Ld r2, 0(r8)
add r3, r1, r2
```

Reordenado con especulación

```
Ld especulativo r1, 0(r9)
Ld especulativo r2, 0(r8)
add especulativo r3, r1, r2
Branch label
```

Control Especulativo

- *Poison bit*
  - Si una instrucción especulativa genera una excepción (por ej. div por cero), la excepción solo debe saltar si efectivamente había que ejecutar esas instrucciones.
  - A cada registro se le agrega un bit adicional que indica que el resultado de la instrucción especulativa generó una excepción.
  - El *poison bit* se propaga a las demás instrucciones especulativas que utilicen ese resultado.

Intel Itanium

Intel Itanium

- 2000 – Creada conjuntamente por Intel y HP
- EPIC
- 2000: 1era generación → 800 MHz
- 2004: 2da generación (Itanium 2) → 1.6GHz

- Es una implementación de la arquitectura IA-64 (ISA 64 bits)

Registros

- 128 registros generales de 65 bits
  - El bit 65 es el *poison bit*
  - 96 de esos registros pueden ser *rotados*.
- 128 registros de punto flotante de 82 bits
  - 96 pueden ser rotados
- 64 registros de predicados de 1 bit
  - 48 pueden ser rotados
- 8 registros de 64 bits para la funciones call y return.

Instrucciones

Bundle (paquete de instrucciones)

Instruction 2 (41 bits)	Instruction 1 (41 bits)	Instruction 0 (41 bits)	Template (5 bits)

Formato de cada instrucción

Opcode 14 bits	Register 1 7 bits	Register 2 7 bits	Register 3 7 bits	Predicate 6 bits

El patrón (*template)* codifica el tipo de cada una de las 3 instrucciones. También puede codificar *stops.* Un *stop* entre dos instrucciones o al final del *bundle*, indica que una instrucción debe completarse antes de comenzar con las siguientes (limita el paralelismo y elimina el control de dependencias).

Pipeline

- El pipeline tiene 10 etapas
  - Front-end: 3 etapas
  - Back-end: 7 etapas
- Entre el front-end y el back-end hay un buffer que puede almacenar hasta 8 bundles.
  - Permite que continúen los fetchs aunque el back-end esté frenado
  - Permite que ingresen instrucciones al back-end aunque haya problemas en el front-end (predicción errónea de *branch*, *fallo en la cache de instrucciones*)

Unidades funcionales

- 4 unidades funcionales para enteros
- 4 unidades multimedia
- 3 unidades de *branch*
- 2 unidades de punto flotante en simple precisión
- 2 unidades de punto flotante en precisión extendida

Front-end

Fetch de instrucciones y predicción de *branch*

- Predictor sofisticado de 2 niveles y *Branch Target Buffer* extendido
  - Instrucciones predicativas y especulativas producen *multiway branches* (*switch*).
- Hay 4 registros que el compilador puede programar con la dirección de salto de *branches* muy predecibles (como el final de un *loop*).
- Se insertan los *bundles* en el *buffer* de entrada al *back-end*.

Back-end

1era etapa

- Trabaja con dos *bundles* por vez. Recién toma uno nuevo cuando uno se terminó completamente.
- La asignación de las UF es relativamente simple:
  - El compilador insertó *stop* donde deben estar.
  - El patrón por *bundle* facilita el control de la disponibilidad de la UF necesaria
  - El patrón restringe a que no todos los tipos de instrucciones pueden ir en cualquier lugar del *bundle*. Esto facilita las interconexiones.

Back-end

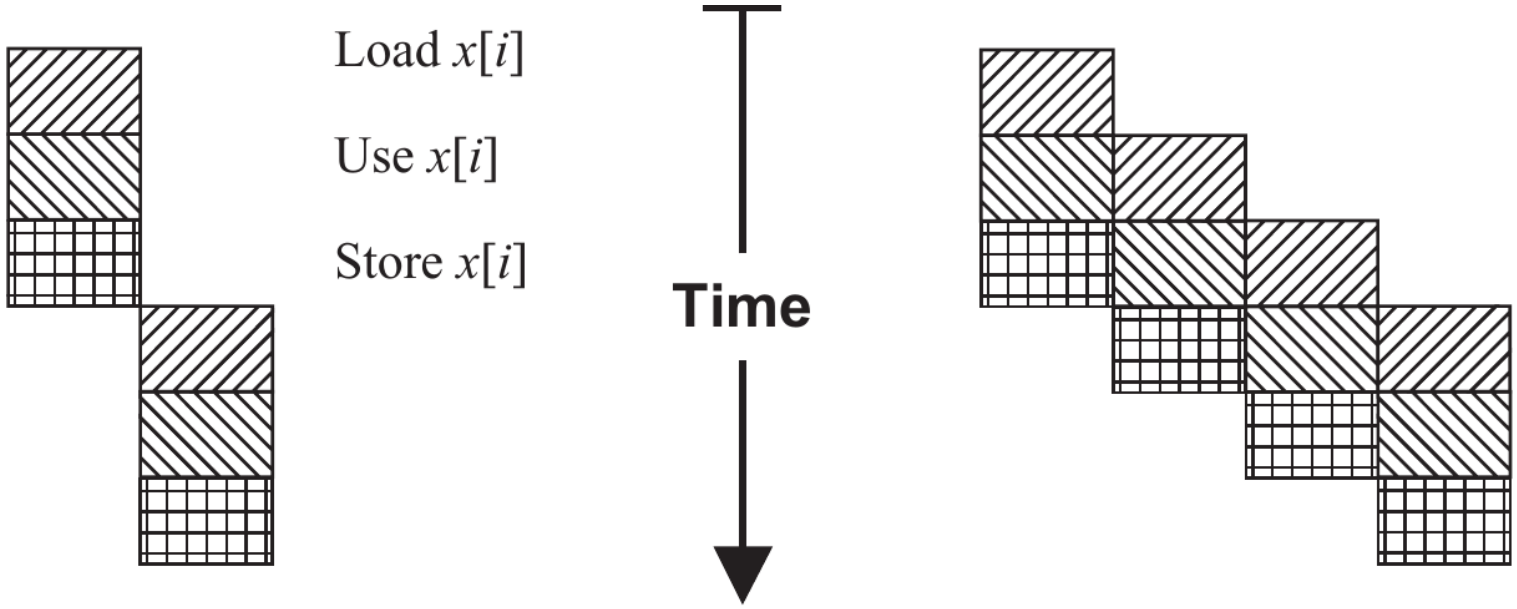
2da etapa

Renombramiento de registros:

- Rotación de registros → *Software pipelining*
- Pila de registros → Protocolo *call-return*

Software pipelining

- Método para reordenar el cuerpo de un bucle e incrementar el ILP cuando hay restricción de recursos o dependencias entre las iteraciones.



Instrucción VLIW:
Store x[i-2] Use x[i-1] Load x[i]

Rotación de registros

- Con una instrucción *branch* especial se indica que los registros deben rotar.
- Para rotar se renombran incrementando el número de registro:
  - Pr_{16}–Pr_{63} registros predicativos: $Pr_i \rightarrow Pr_{i+1}$ y $Pr_{63} \rightarrow Pr_{16}$
  - Fr_{32}–Fr_{127} registros punto flotante: $Fr_i \rightarrow Fr_{i+1}$ y $Fr_{127} \rightarrow Fr_{32}$
  - Gr_{32}–Gr_{127} son los registros generales. Se puede designar una cantidad múltiplo de 8 (comenzando por el 32) como conjunto a rotar: $Gr_i \rightarrow Gr_{i+1}$ y $Gr_{32+CANT-1} \rightarrow Gr_{32}$
- Los datos de una iteración se encuentran desplazados un lugar para la iteración siguiente.

Register Stacking

- La pila puede implementarse en memoria o en un banco de registros.
- Itanium utiliza ambas implementaciones de la pila.
- Los 96 registros $Gr_{32–127}$ pueden usarse como pila.
- Si los registros no alcanzan (stack overflow) el comienzo de la pila se copia en memoria.

Register Stacking

- El compilador indica cuántas variables de entrada (ins), cuántas locales (loc) y cuántas de salida (out) serán necesarias.

```
A() {
  loc0
  loc1
  loc2
  …
  B(out0, out1, out2)
 …
}
```

```
B(ins0, ins1, ins2) {
  …
}
```

Back-end

3era etapa:

Se leen los registros.

4ta etapa:

Un *scoreboard:*

- Planifica los *stall* y *forwarding* debido a conflictos ocasionados por las latencias del load (que no siempre se pueden prever).
- Puede anular *stalls* si el productor o el consumidor de una operación frenada tiene un predicado falso.

Back-end

5ta etapa

- Ejecución

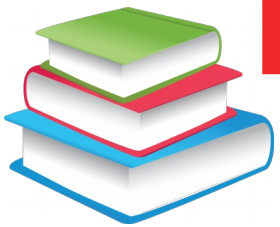
6ta etapa

- Detección de excepciones

7ma etapa

- Escritura de registros.

Bibliografía



- *Capítulo 3. Multiprocessor Architecture. From simple pipelines to chip multiprocessor*. Jean-Loup Baer. Cambridge University Press. 2010.

Suplementaria

- *Capítulo 3 y Apéndice C.* John L. Hennessy & David A. Patterson. *Computer Architecture. A Quantitative Approach*. Elsevier Inc. 2012, 5ta Ed.