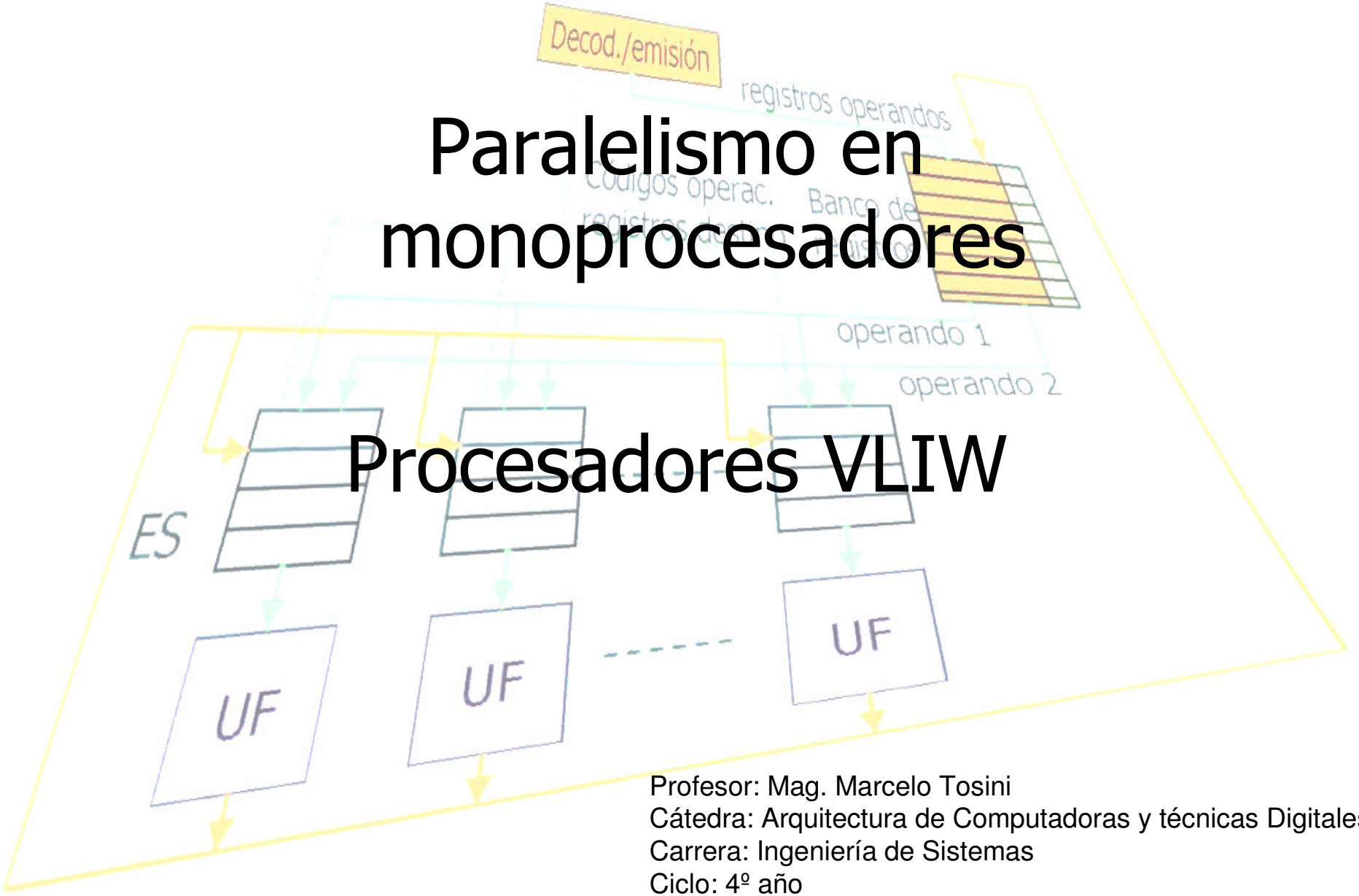
Paralelismo en monoprocesadores

Procesadores VLIW



Profesor: Mag. Marcelo Tosini
Cátedra: Arquitectura de Computadoras y técnicas Digitales
Carrera: Ingeniería de Sistemas
Ciclo: 4º año

Arquitectura VLIW básica

- Data de principios de los años 70
- Muy apoyada en la tecnología de compiladores que optimicen el código
- Una instrucción especifica varias operacíones agrupadas en un formato largo de instrucción con longitudes desde 128 a 1024 bits
- La planificación de ejecución de las operaciones es externa al procesador
- El rendimiento del procesador queda determinado por la calidad del compilador

Formato de instrucciones

Procesador tradicional

100	inst 1 (op1)
101	inst 2 (op2)
102	inst 3 (op3)
103	inst 4 (op4)
104	inst 5 (op5)
105	inst 6 (op6)
106	inst 7 (op7)
107	inst 8 (op8)
108	inst 9 (op9)
109	inst 10 (op10)

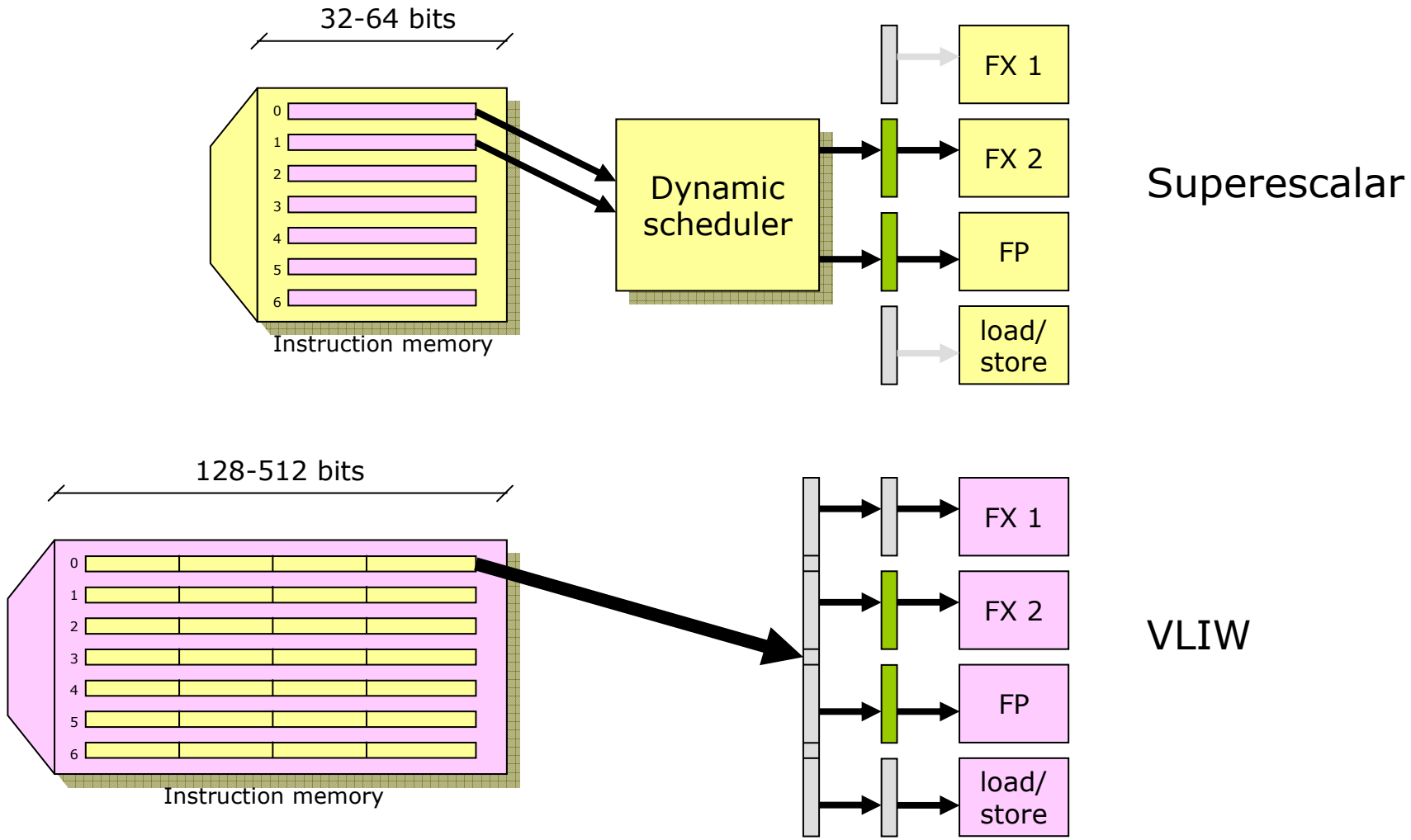
una instrucción = una operación

Procesador VLIW

100	operación 1	NOP	operación 5
101	NOP	operación 2	operación 3
102	operación 6	NOP	operación 7
103	operación 4	NOP	NOP
104	NOP	operación 8	operación 10
105	NOP	NOP	operación 9
106	NOP	NOP	NOP
107	NOP	operación 11	operación 12
108	operación 14	operación 16	NOP
109	operación 13	NOP	operación 15

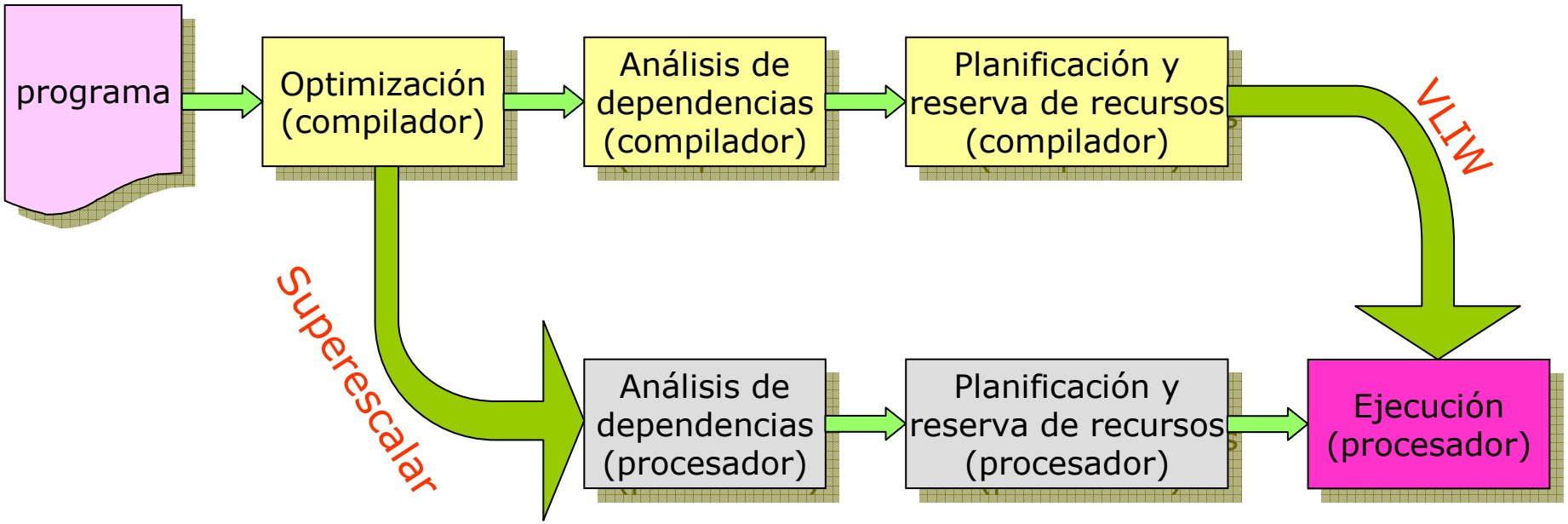
una instrucción = varias operaciones

VLIW vs. Superescalar



VLIW vs. Superescalar

Diferente división de tareas entre software (compilador) y hardware (procesador)



VLIW vs. Superescalar

En VLIW no se puede determinar si hay dependencias en ciertos casos

Ejemplo:

```
lw      r1, 100(r2)      ; si r2 = r4+100, hay dependencia!!!
sw      r3, 0(r4)
```

Procesador superescalar

comprueba si la dirección r2 + 100 es igual a r4 + 0

- igual => secuencializa las instrucciones
- NO igual => paraleliza las instrucciones

Procesador VLIW

No puede determinarlo => el compilador siempre secuencializa

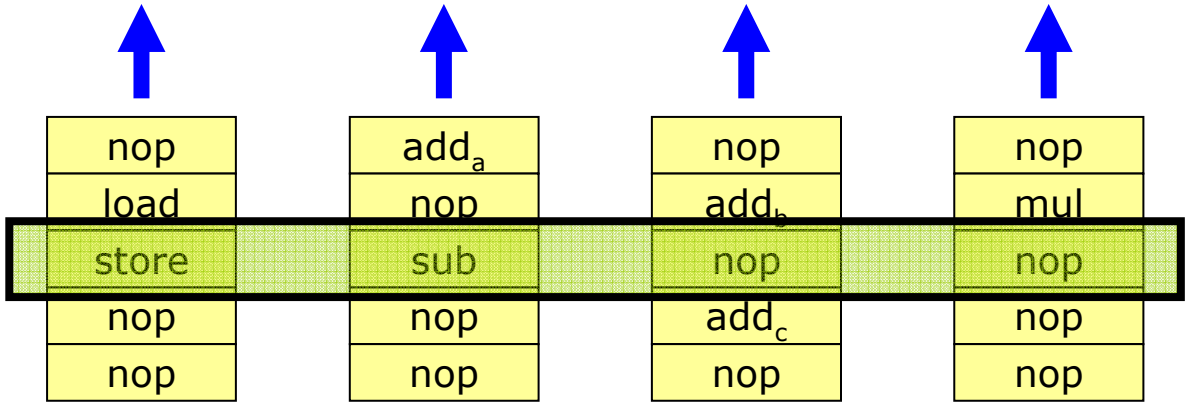
Arquitectura VLIW: ventajas

- La planificación de la ejecución estática de instrucciones es realizada por el compilador:
    - Menos lógica en la unidad de control del procesador
    - Mayor frecuencia de reloj (organización mas sencilla)
    - Mayor disponibilidad de especio para otras unidades (Pe: unidades funcionales)
- Es una arquitectura muy difundida en circuitos embebidos
    - DSP´s
    - Multimedia (placas de sonido, video, etc.)

Arquitectura VLIW: Limitaciones

- Tamaño del código
  - Mucho desperdicio de memoria por instrucciones NOP
- Compatibilidad de código objeto
  - Hay que recompilar todo el código fuente para cada nueva versión de procesador
- Baja ocupación de las unidades funcionales
- Conflictos en el acceso al banco de registros desde varias unidades
- Conflictos de predicción estática de accesos a memoria
  - Imposibilidad de determinación de latencias en caso de posibles fallos en accesos a Ram o caché
- Dificultades en la predicción estática de saltos

Tamaño del código

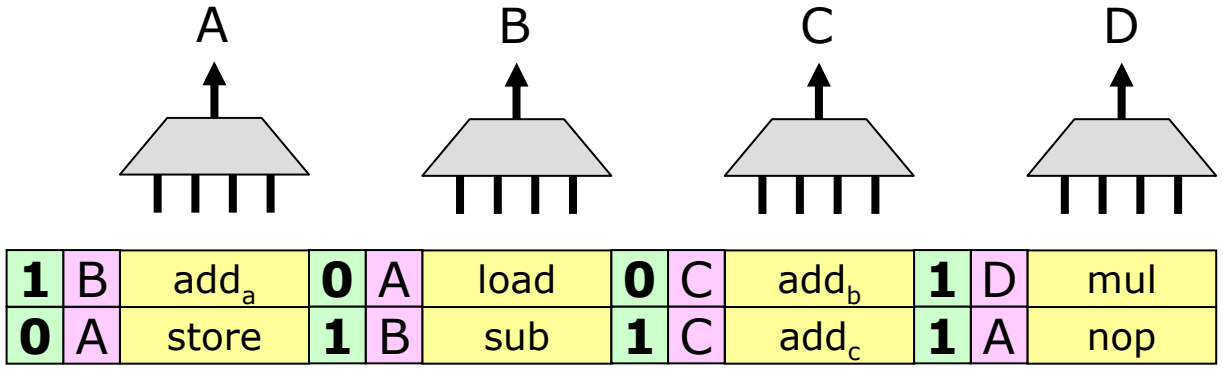


Formato de instrucción VLIW desempaquetado

- útil para ejecución en las UF

Formato de instrucción VLIW empaquetado

- útil para almacenar en la memoria



0 Separador de instrucciones VLIW (1 = última instrucción)

B Unidad funcional en la que se ejecutará la instrucción

Tamaño del código

En qué momento se descomprimen las instrucciones de memoria?

- Carga en la I-Caché
  - ↓ • Transferencia de bloques a la cache es lenta (bus de acceso a RAM cuello de botella)
  - ↑ • El algoritmo de compresión puede ser complejo ya que se dispone de tiempo
  - ↓ • se ocupa mucho especio de cache si las instrucciones están descomprimidas
- Lectura de la I-Caché (Fetching)
  - ↑ • En la caché se mantienen las instrucciones empaquetadas
  - ↓ •La descompresión en la etapa de fetching puede ser lenta (agrega una o más etapas de segmentación al proceso)

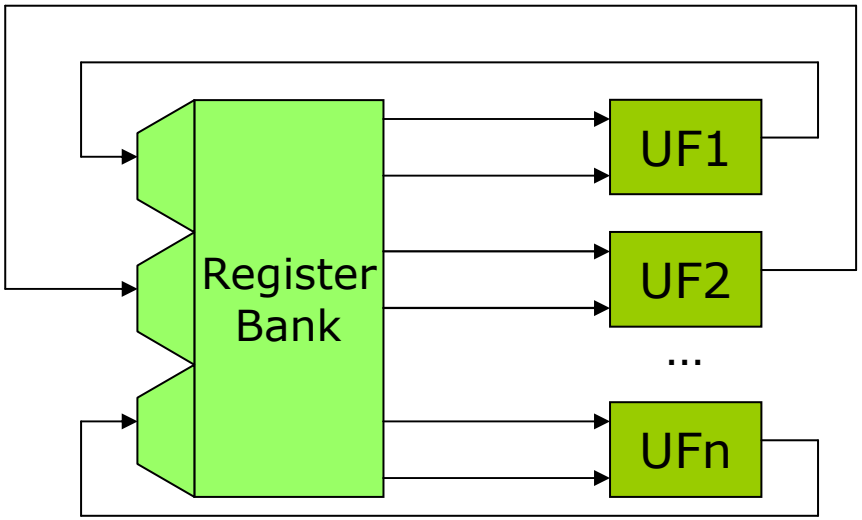
Ocupación de las unidades funcionales

Meta principal: Mantener ocupadas todas las unidades funcionales

Problema:

Para N unidades funcionales usables en cada ciclo de reloj

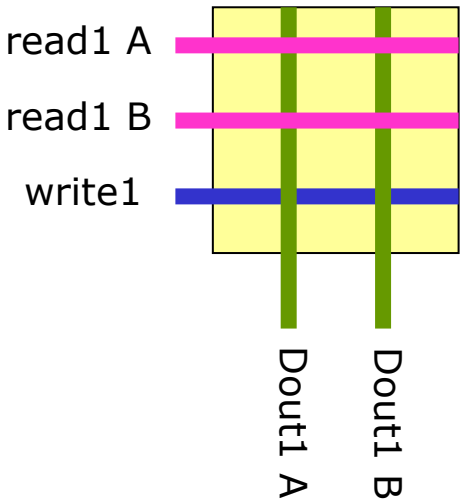
- Se leen 2 operandos fuente por unidad
- Se escribe un resultado por unidad

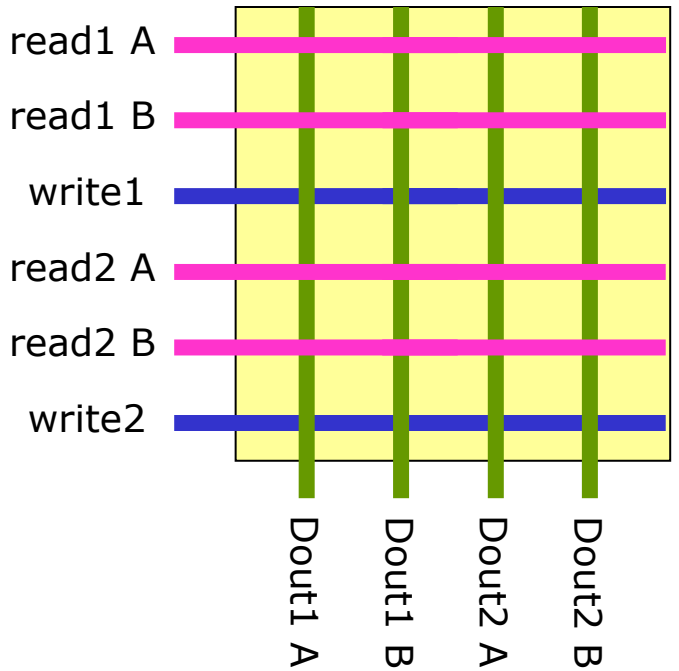


Ocupación de las unidades funcionales

Arquitectura del banco de registros

- Área banco de registros ≈ cuadrado del número de puertos (de entrada o salida)
- Diseño limitado por el número de puertos





Ocupación de las unidades funcionales

Arquitectura del banco de registros

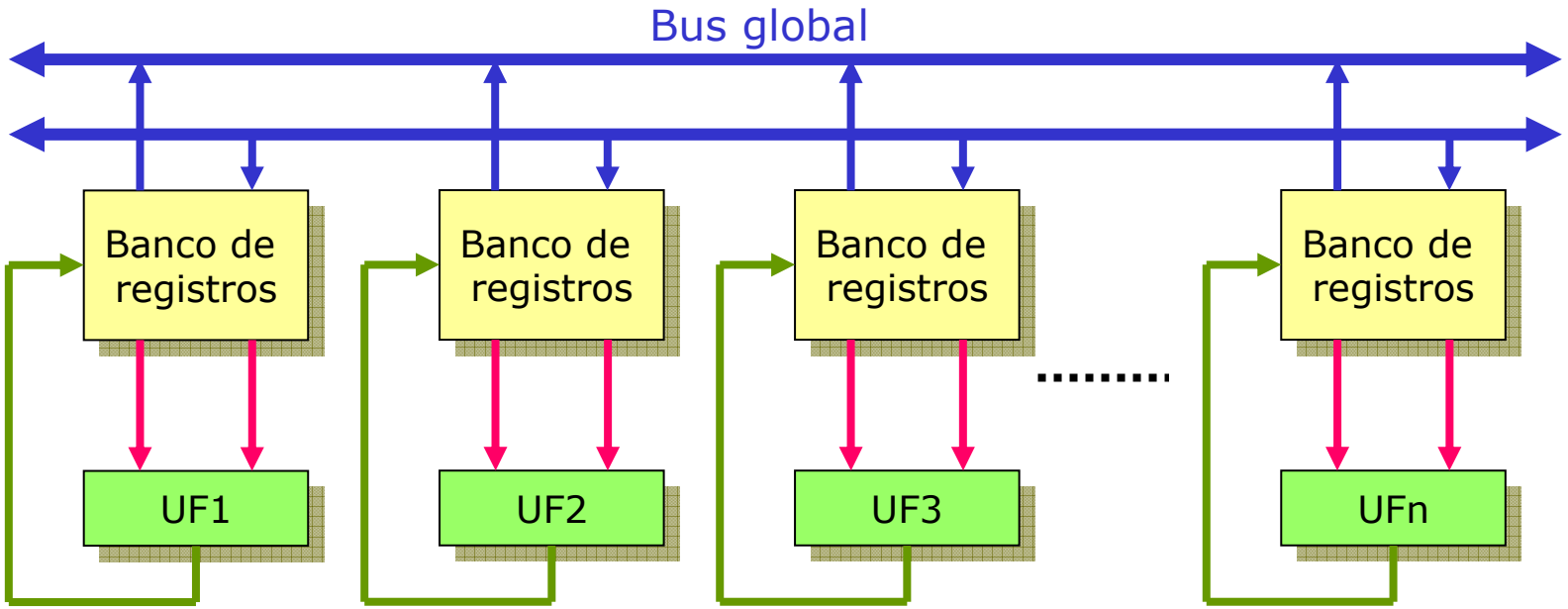
- Tiempo de acceso al banco de registros proporcional al número de registros
  - Al crecer las áreas de celda básica y decodificadores hay mayores retardos
- A partir de aproximadamente 20 puertos (I/O) la velocidad se degrada bastante
- Aproximadamente 14 lecturas y 7 escrituras

=> hasta 7 unidades funcionales como máximo

Ocupación de las unidades funcionales

Optimización de la arquitectura del banco de registros

- Particionar el banco de registros en bancos locales a cada unidad funcional y con canales de comunicación globales



Planificación del compilador

- La eficiencia del compilador influye mucho en el rendimiento final de la arquitectura

  Rendimiento alcanzado por los compiladores actuales es inferior al rendimiento potencial máximo que alcanza la arquitectura

- Código generado por el compilador muy dependiente de la arquitectura VLIW de destino

  Ante actualizaciones de hardware es necesario reescribir el compilador (mayores costos de actualización)

- Para optimizar mejor el código es útil que el programador ayude al compilador

  Lenguajes deben usar meta comandos (pragmas) especiales

Planificación del compilador

Optimizaciones del compilador: Análisis global del código

- Modificación del código fuente para incrementar el paralelismo a nivel de instrucción
- Aplicación de técnicas usadas para superescalares:
    - Desenrollamiento de lazos
    - Segmantación de software
    - Actualización de referencias
    - Renombre de registros

Planificación del compilador

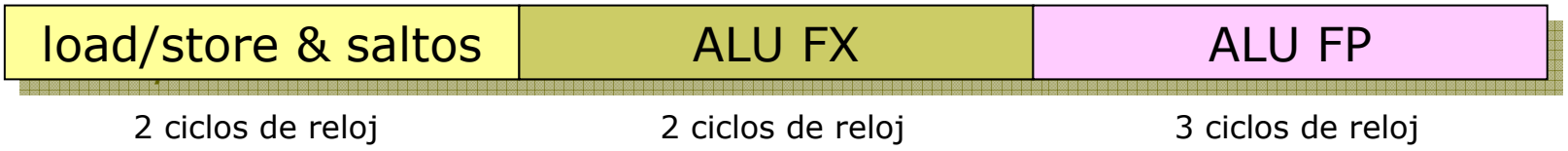
Ejemplo: Suma de un valor a un arreglo

```
for (i=10; i>0; i--)
  x[i] = x[i] + s;
```

En ensamblador:

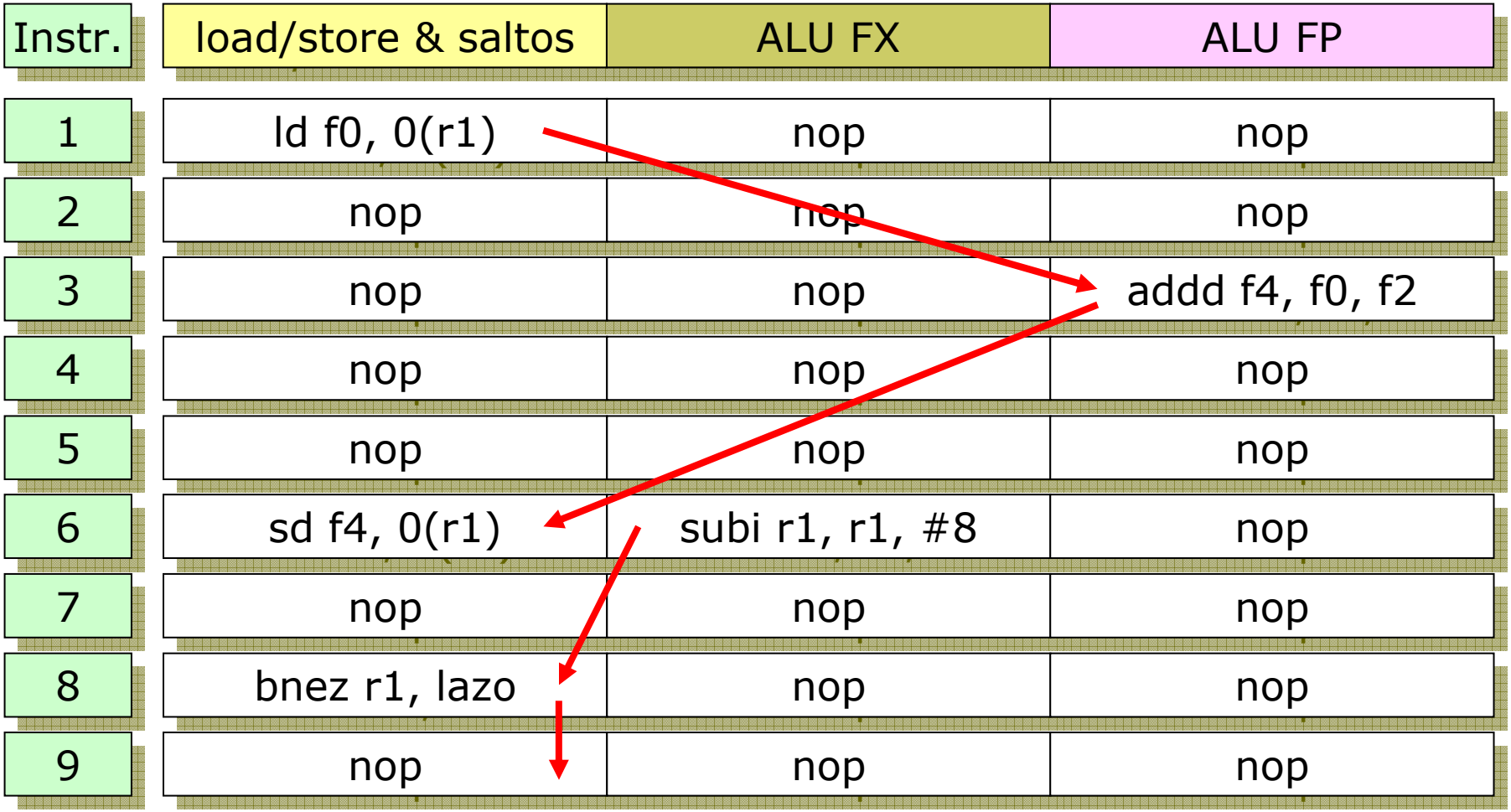
```
lazo:   ld      f0, 0(r1)       ; f0 = x[i]
        addd    f4, f0, f2      ; f4 = f0 + s
        sd      f4, 0(r1)       ; x[i] = f4
        subi    r1, r1, #8      ; i--
        bnez    r1, lazo        ; if i ≠ 0 goto lazo
```

Suponer un procesador VLIW de 3 unidades funcionales:



Planificación del compilador

Instrucciones VLIW del programa



Instr.	load/store & saltos	ALU FX	ALU FP
1	ld f0, 0(r1)	nop	nop
2	nop	nop	nop
3	nop	nop	addd f4, f0, f2
4	nop	nop	nop
5	nop	nop	nop
6	sd f4, 0(r1)	subi r1, r1, #8	nop
7	nop	nop	nop
8	bnez r1, lazo	nop	nop
9	nop	nop	nop

Planificación del compilador

Replicación de código

```
lazo:   ld      f0, 0(r1)
        addd    f4, f0, f2
        sd      f4, 0(r1)
        subi    r1, r1, #8
        bnez    r1, lazo
```

```
lazo:   ld      f0, 0(r1)
        addd    f4, f0, f2
        sd      f4, 0(r1)

        ld      f0, 0(r1-8)
        addd    f8, f6, f2
        sd      f4, 0(r1-8)

        ld      f0, 0(r1-16)
        addd    f12, f10, f2
        sd      f4, 0(r1-16)

        ld      f0, 0(r1-24)
        addd    f16, f14, f2
        sd      f4, 0(r1-24)

        ld      f0, 0(r1-32)
        addd    f20, f18, f2
        sd      f4, 0(r1-32)

        subi    r1, r1, #8
        bnez    r1, lazo
```

Planificación del compilador

Código VLIW optimizado del programa

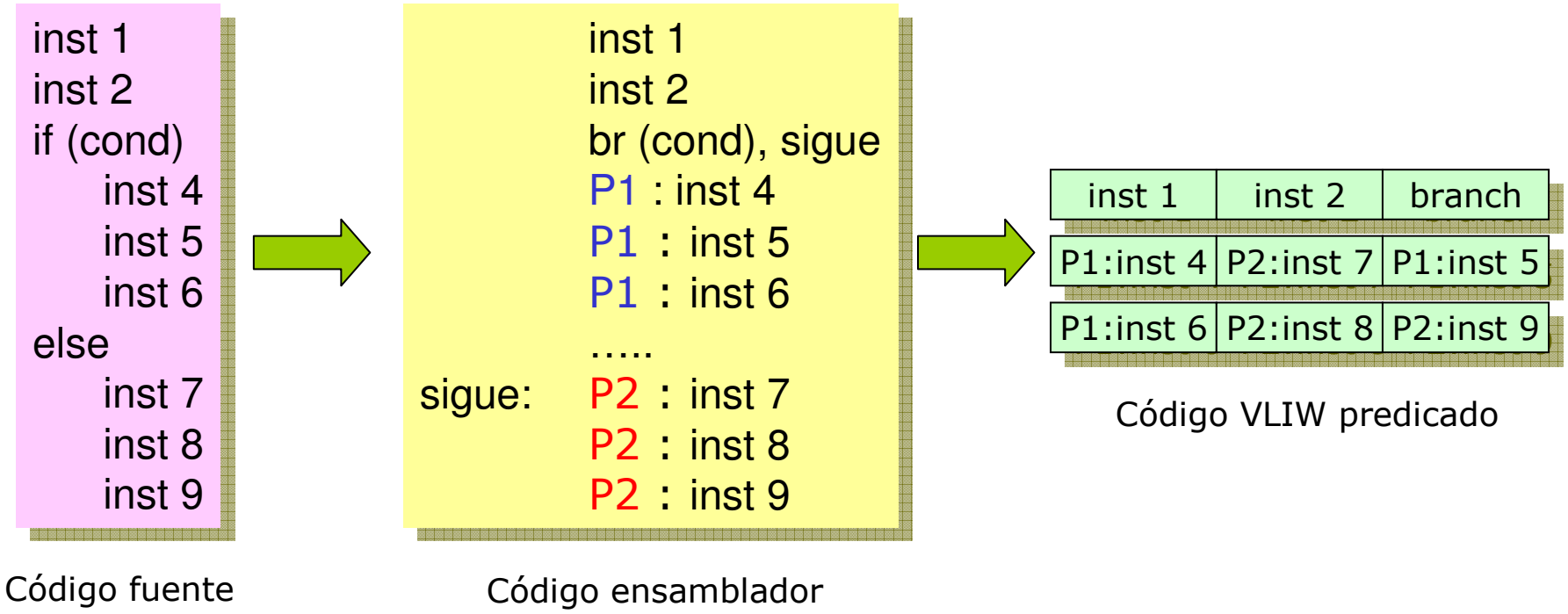
Instr.	load/store & saltos	ALU FX	ALU FP
1	ld f0, 0(r1)	nop	nop
2	ld f0, 0(r1-8)	nop	nop
3	ld f0, 0(r1-16)	nop	addd f4, f0, f2
4	ld f0, 0(r1-24)	nop	addd f8, f6, f2
5	ld f0, 0(r1-32)	nop	addd f12, f10, f2
6	sd f0, 0(r1)	nop	addd f16, f14, f2
7	sd f0, 0(r1-8)	nop	addd f20, f18, f2
8	sd f0, 0(r1-16)	nop	nop
9	sd f0, 0(r1-24)	nop	nop
10	sd f0, 0(r1-32)	subi r1, r1, #8	nop
11	nop	nop	nop
12	bnez r1, lazo	nop	nop
13	nop	nop	nop

Instrucciones predicadas

Ante una bifurcación el compilador asigna a las instrucciones de las dos ramas posibles un predicado a cada una.

De esta manera es posible identificar la rama inválida cuando se resuelve la condición del branch

Las instrucciones de ambas ramas pueden ejecutarse en paralelo ya que no poseen interdependencias



Instrucciones predicadas

Ejemplo:

```
if (a < b)
      c = a;
else
      c = b;
if (d < e)
      f = d;
else
      f = e;
```

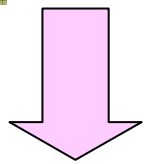
Queda en VLIW como

P1 = CMPP.< a, b	P2 = CMPP.>= a, b	P3 = CMPP.< d, e	P4 = CMPP.>= d, e
<P1> MOV c, a	<P2> MOV c, b	<P3> MOV f, d	<P4> MOV f, e

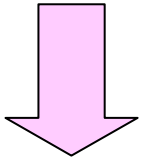
instrucciones predicadas

La instrucción `P1 = CMPP.< a, b` verifica si **a<b** en cuyo caso pone **P1=1**

La instrucción `P2 = CMPP.>= a, b` verifica si **a>=b** en cuyo caso pone **P2=1**



Algunos procesadores soportan la versión de 2 salidas de CMPP



`P1, P2 = CMPP.W.<.UN.UC a, b`

.W : **W**ide compare. compara generando 2 predicados
.UN: **U**nconditional **N**ormal. Asigna la condición a<b a P1 de forma directa
.UC: **U**nconditional **C**omplementary. Asigna la condición a<b a P2 de forma negada (a>=b)

Instrucciones predicadas

Ejemplo: Con la nueva instrucción CMPP

```
if (a < b)
      c = a;
else
      c = b;
if (d < e)
      f = d;
else
      f = e;
```

Queda en VLIW como

P1, P2 = CMPP.W.<.UN.UC a, b	P3, P4 = CMPP.W.<.UN.UC d, e

<P1> MOV c, a	<P2> MOV c, b	<P3> MOV f, d	<P4> MOV f, e

Ejemplo: Arquitectura EPIC

EPIC (Explicit Parallel Instruction Computer) Es el término dado por Intel y HP a su nueva generación de procesadores VLIW, Itanium

Se comporta a nivel arquitectural como un VLIW cásico

Se distingue de una arquitectura VLIW pura en:

- Paralelismo explícito
- Predicación completa de instrucciones
- Banco de registros rotativo
- Palabra de instrucción muy larga que puede codificar grupos de instrucciones no paralelas
- Especulación
- Compatibilidad con programas IA-32 (Pentium II)

Ejemplo: Arquitectura Intel IA-64

Registros

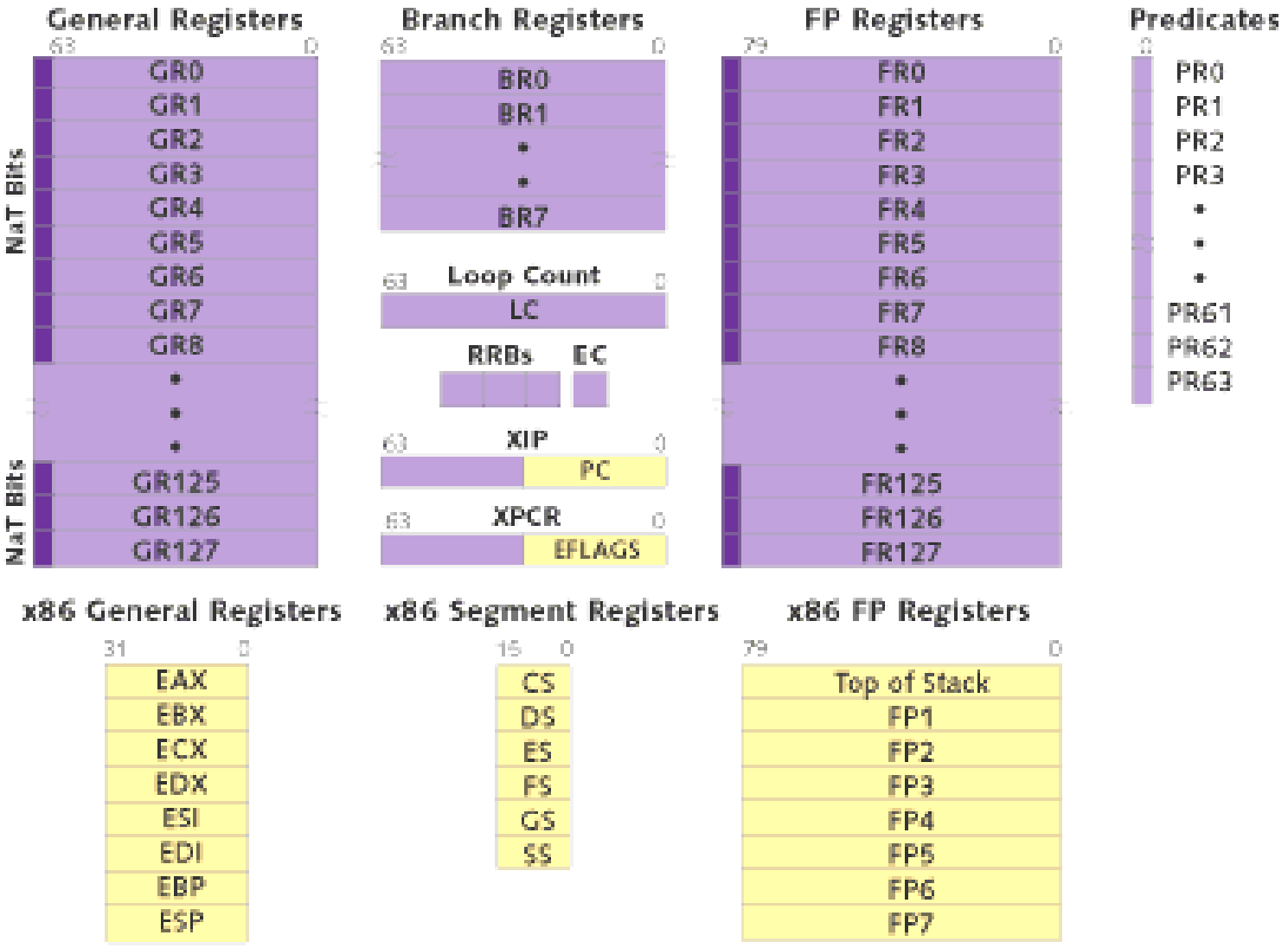
- **128** registros de enteros ó generales, cada uno de 64 bits de longitud.
- **128** registros de punto flotante, cada uno de 80 bits de longitud.
- **64** bits para la predicación.
- **256** NAT bits.
- **8** registros para saltos.
- 128 registros de aplicación
- **3** RRB's o registros base rotativos.
- LC ó Loop Count.
- EC.
- IP ó contador de instrucciones

Todos los registros pueden ser accedidos por software, ya que son visibles al programador y de acceso aleatorio.

Son numéricamente entre 4 y 8 veces el numero de los registros de las arquitecturas RISC.

Proveen compatibilidad con x86.

Ejemplo: Arquitectura Intel IA-64

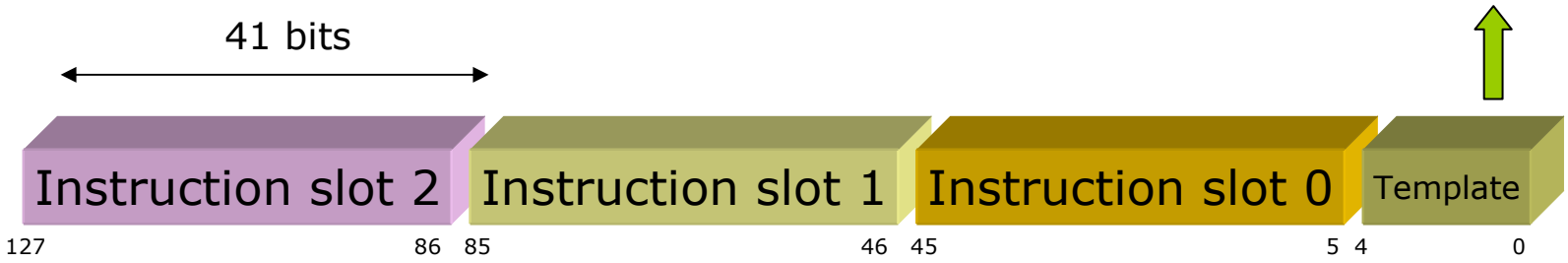


Ejemplo: Arquitectura Intel IA-64

Instrucciones largas: de 128 bits

Bundle (paquete): Conjunto de **3 instrucciones** y un **template**

- Indica en que unidad funcional se ejecuta cada operación
- La dependencia entre este bundle y los siguientes

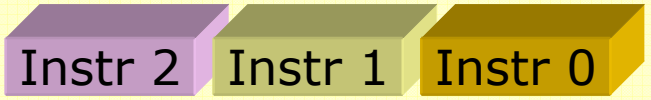


- Código de operación
- 6 bits de registro predicado
- 7 bits de operando fuente 1
- 7 bits de operando fuente 2
- 7 bits de operando destino
- Extensión de código de operación (saltos, etc.)

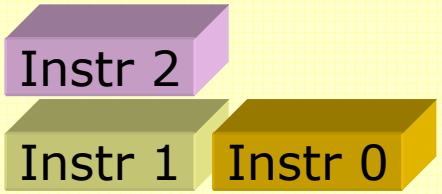
Ejemplo: Arquitectura Intel IA-64

Combinaciones de instrucciones de un bundle

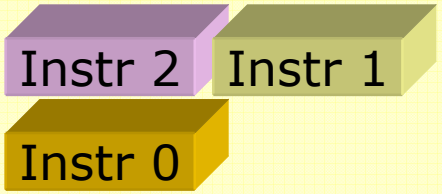
Todas las instrucciones ejecutadas en paralelo.
i2 || i1 || i0



Primero i2, luego i1 y i0 ejecutadas en paralelo.
i2 & i1 || i0



i2 y i1 ejecutadas en paralelo, luego i0.
i2 || i1 & i0



i2, i1 y i0 ejecutadas en serie.
i2 & i1 & i0

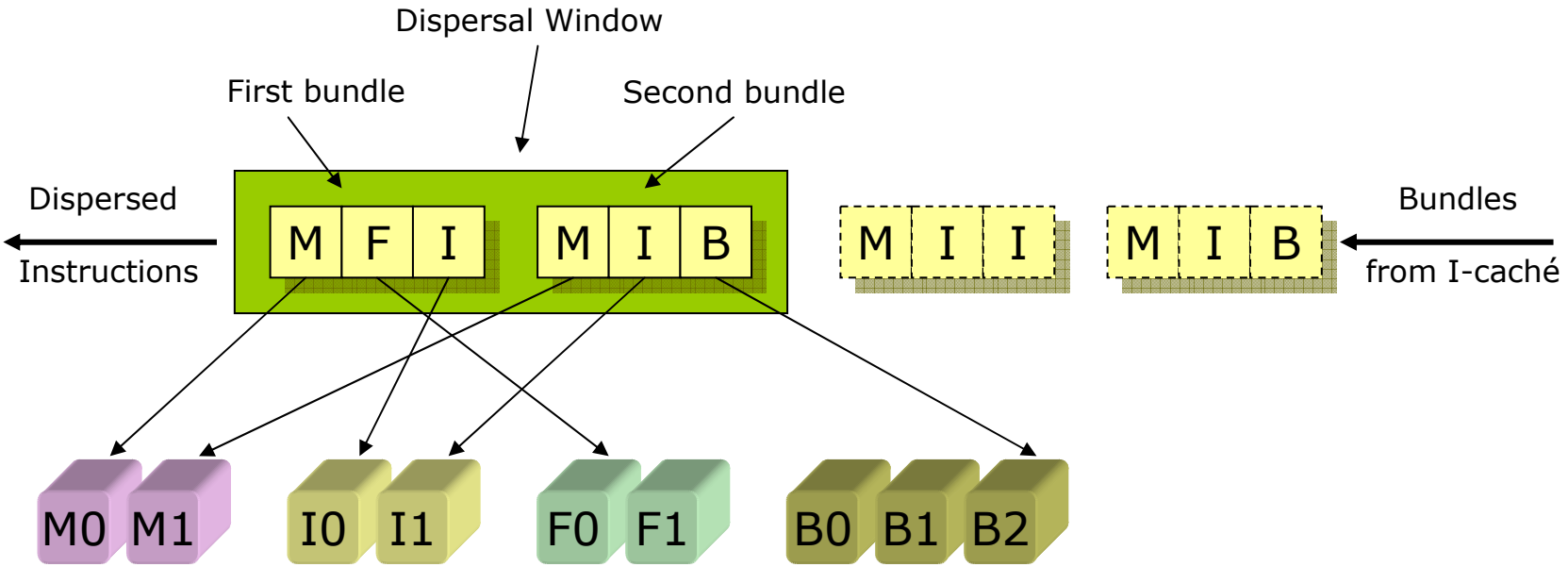
Ejemplo: Arquitectura Intel IA-64

Tipos de instrucciones:

- M : Memory
- I : Integer
- F : Floating Point
- B : Branch

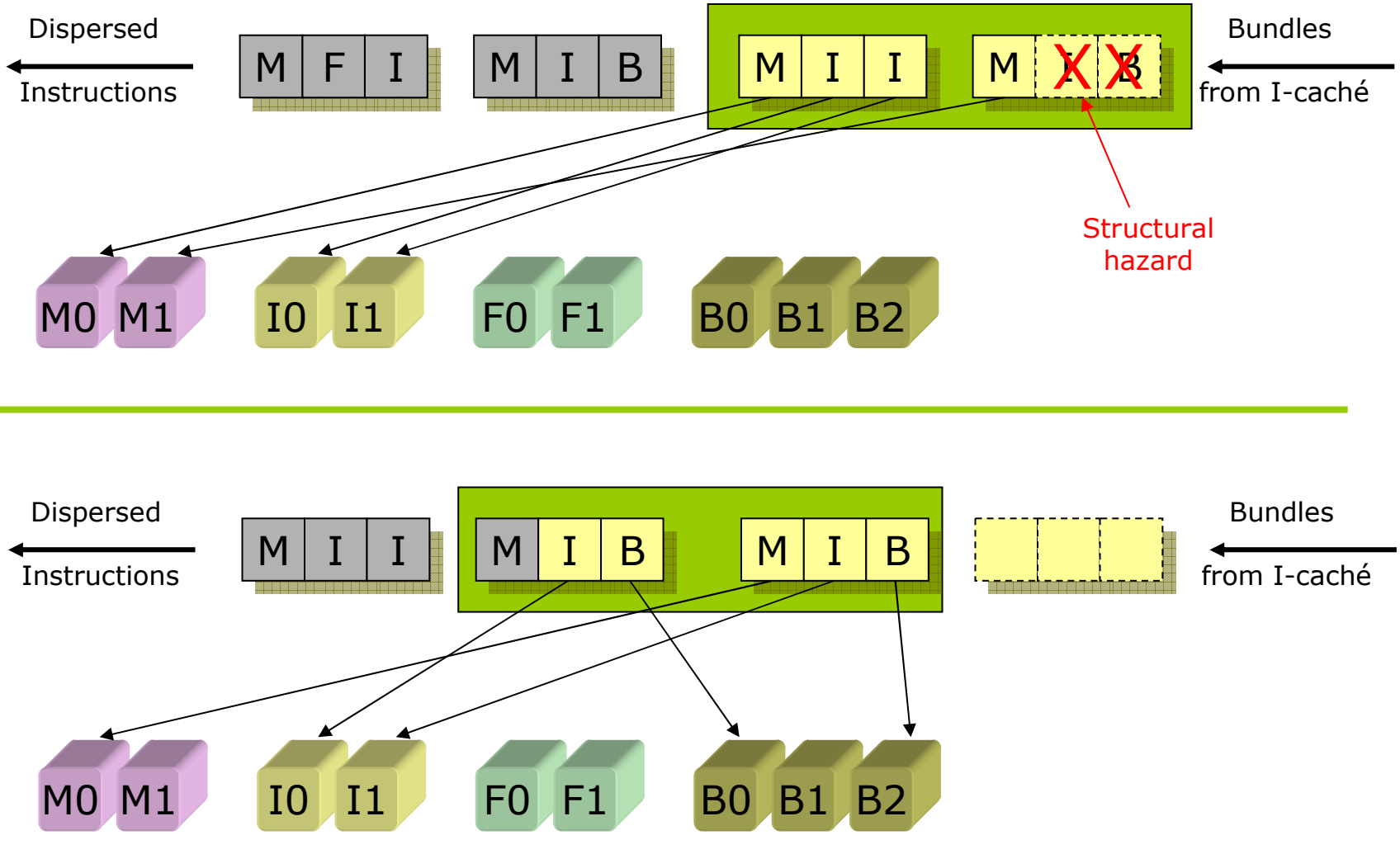
Arquitectura Hardware

- Unidades funcionales
  - 2 de Memoria
  - 2 de Enteros
  - 2 de Punto Flotante
  - 3 de Saltos
- Tamaño de ventana : 2 instr.



Source: Itanium Processor, Intel Corp, 2000

Ejemplo: Arquitectura Intel IA-64



Source: Itanium Processor, Intel Corp, 2000

Arquitectura Intel IA-64
Instrucciones predicadas

Predicados

- Asociados a todas las instrucciones en las 2 rutas (efectiva y no efectiva)
- El compilador añade a la instrucción un valor adicional que es el predicado cuyo valor es complementario en cada una de las ramas
- Las insttrucciones con predicado complementario son paralelizables
- El predicado es calculado por el resultado de la condición de salto asociada
- Las operaciones asociadas a las instrucciones son siempre ejecutadas, pero el estado del procesador sólo se modifica si el predicado asociado es cierto

Arquitectura Intel IA-64
Instrucciones predicadas: Ejemplo

Código fuente

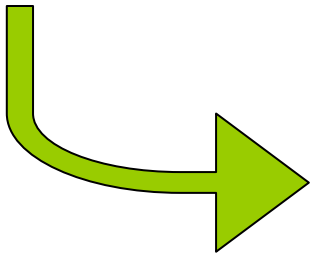
```
V1 = V1 + 1;
if (a & b) {
        V2 = V2 / 2;
        V3 = V3 * 3;
} else {
        V4 = V4 – 4;
        V5 = V5 + 5;
}
V6 = V6 -6;
```

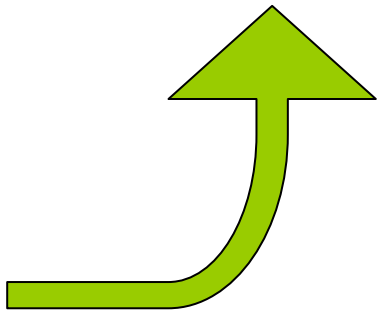
Código ensamblador

```
        add   data_V1, #1

        bez   data_a, _blkelse
        bez   data_b, _blkelse

        div   data_V2, #2
        mul   data_V3, #3

        jmp   _endif

_blkelse: sub   data_V4, #4
        add   data_V5, #5

_endif:   sub data_V6, #6
```

Código predicado

```
        add   data_V1, #1

P1, P2 := cmp  data_a, 0
   <P2>  cmp  data_b, 0

   <P2>  div   data_V2, #2
   <P2>  mul  data_V3, #3

   <P1>  sub   data_V4, #4
   <P1>  add   data_V5, #5

        sub data_V6, #6
```

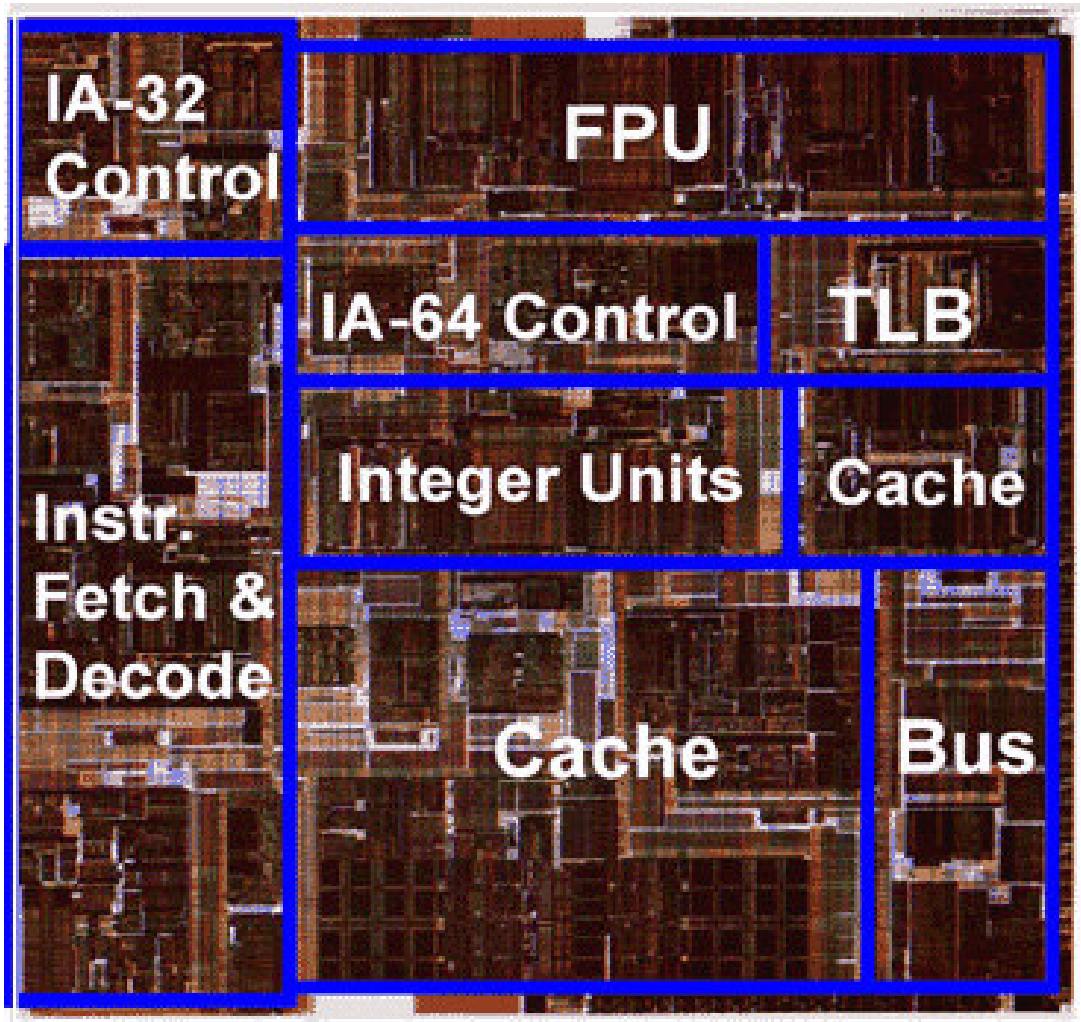




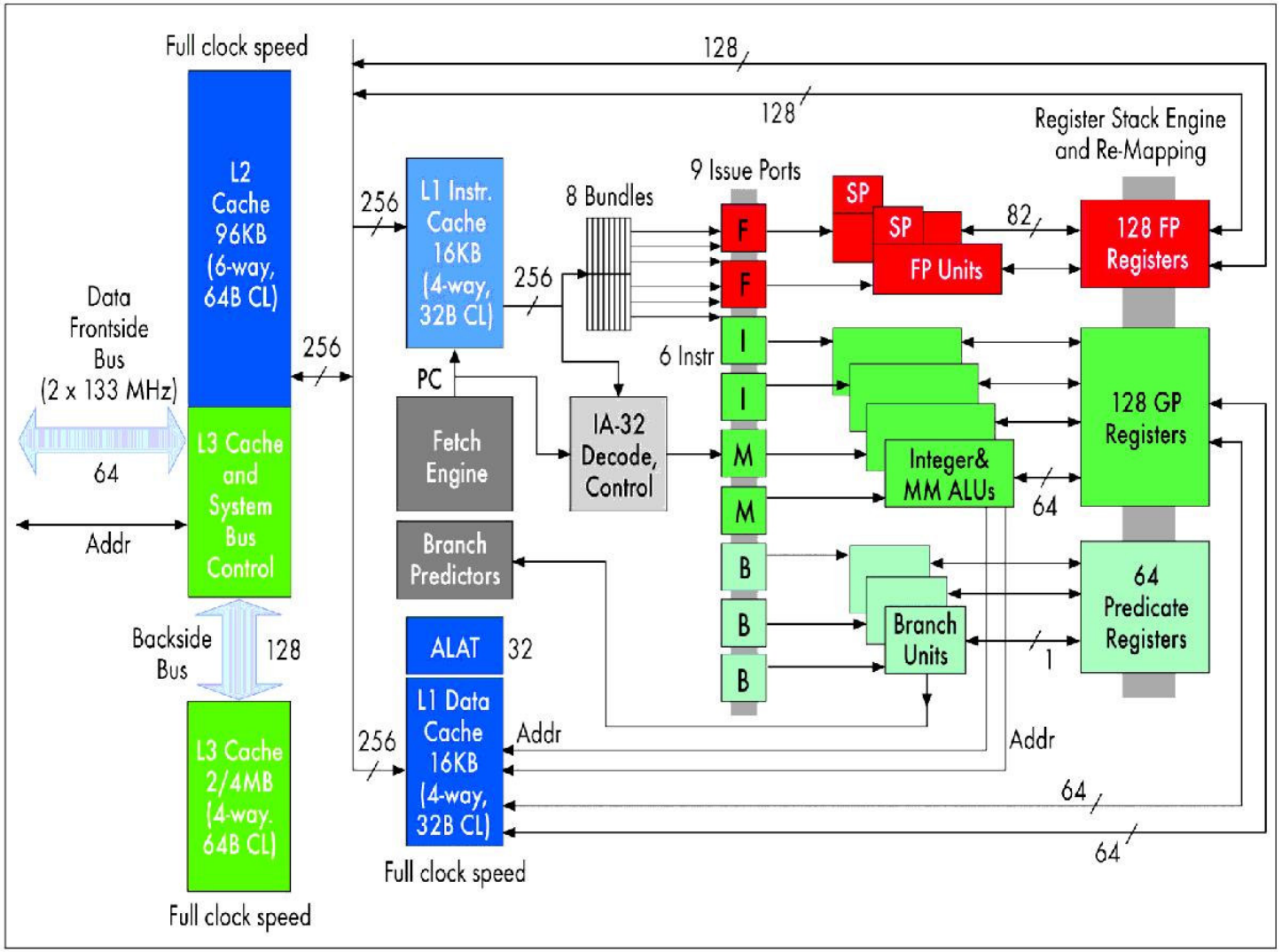
Características del Itanium

Frecuencia	800 Mhz
Número de transistores	CPU: 25.400.000 - L3: 295.000.000
tecnología de fabricación	CMOS de 0,18µ con 6 capas de metal
Ancho de cálculo	6 instrucciones por ciclo de reloj
Unidades funcionales	4 FX (2 FX + 2 MM) – 2 ld/st – 2 FP – 3 Br
Registros	banco de 14 puertos con 128 Gr y 128 FR 64 bits de predicados
Mecanismo de especulación	ALAT de 32 entradas/excepciones diferidas
Predicción de saltos	esquema de predicción de 4 niveles
Velocidad punto flotante	3,2 Gflops a 6,4 Gflops
Velocidad de memoria	8 operandos pto flotante por ciclo de reloj

Itanium Merced



Itanium


Full clock speed
L2 Cache 96KB (6-way, 64B CL)
Data Frontside Bus (2 x 133 MHz)
64
Addr
L3 Cache and System Bus Control
256
Backside Bus
128
L3 Cache 2/4MB (4-way. 64B CL)
Full clock speed
128
128
Register Stack Engine and Re-Mapping
256
L1 Instr. Cache 16KB (4-way, 32B CL)
256
8 Bundles
9 Issue Ports
F
F
I
I
M
M
B
B
B
6 Instr
PC
Fetch Engine
IA-32 Decode, Control
Branch Predictors
ALAT
32
L1 Data Cache 16KB (4-way, 32B CL)
256
Full clock speed
SP
SP
FP Units
82
128 FP Registers
Integer& MM ALUs
64
128 GP Registers
Branch Units
1
64 Predicate Registers
Addr
Addr
64
64