# TEMA 4. ARQUITECTURA IA-64

Stalling, W.Computer Organization and Architecture cap. 15

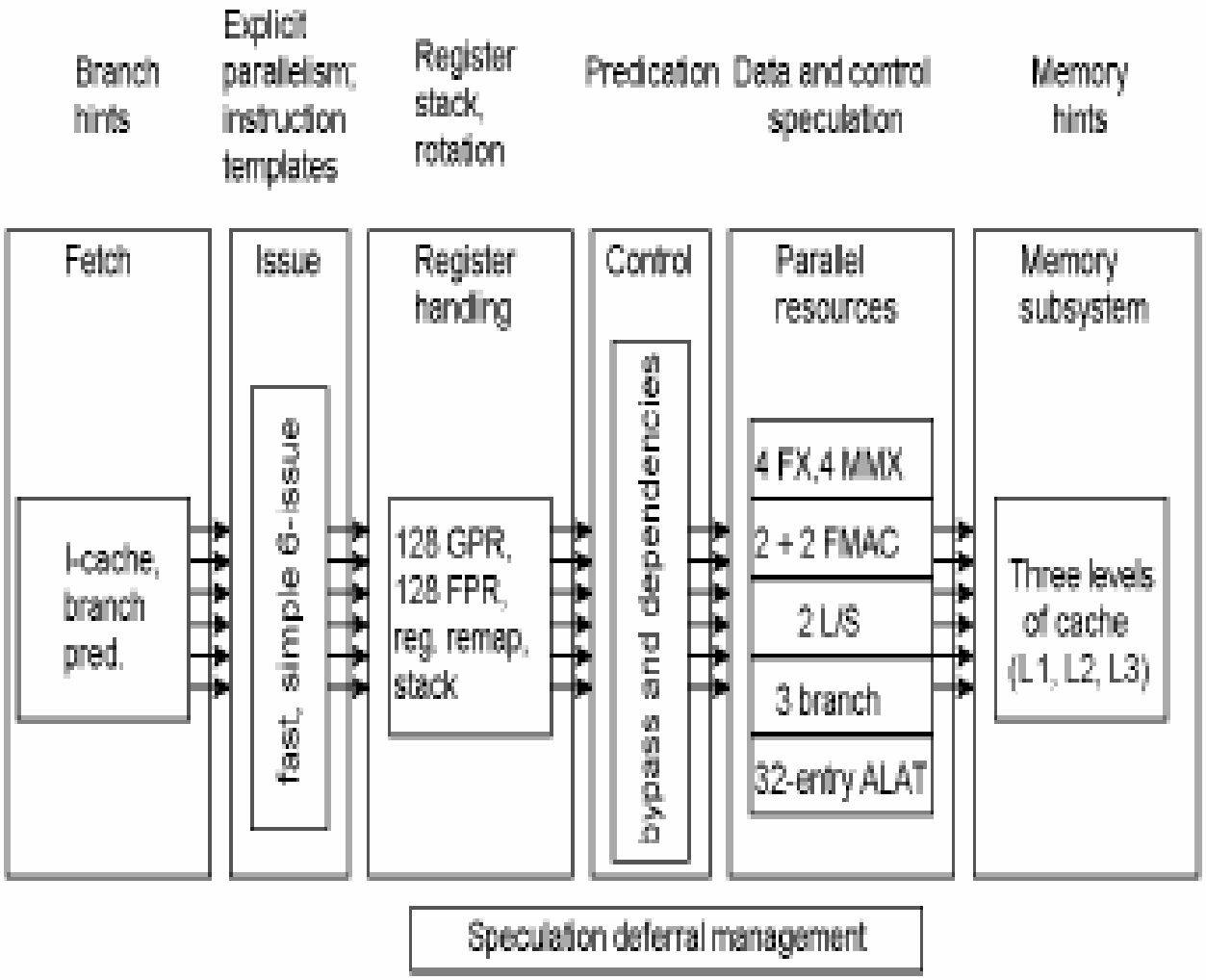
Intel IA-64 Architecture Software Developer's Manual

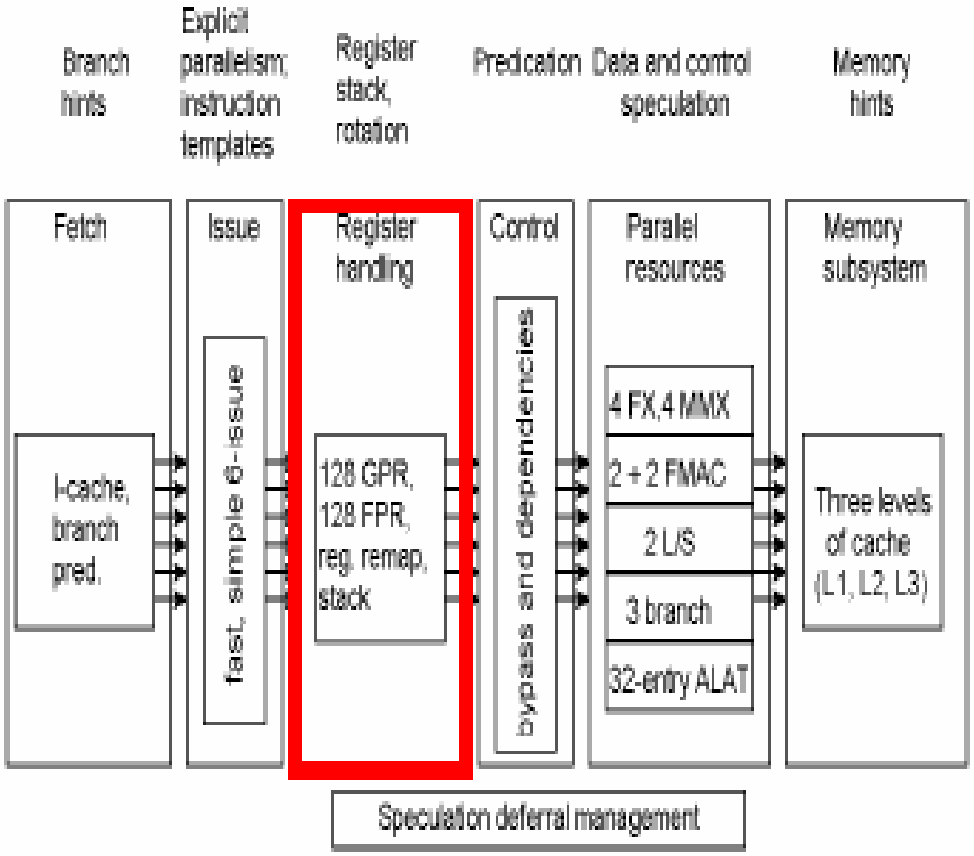
# Generalidades IA-64

- Desarrollo conjunto Intel-HP
- Nueva arquitectura de 64 bits
  - No es una extensión de IA-32
  - No es una adaptación de PA-RISC
- Tecnología CMOS (10 millones de transistores)
- Frecuencia de  800 MHz (2000)
- Uso intensivo del paralelismo
- Primera implementación : ITANIUM

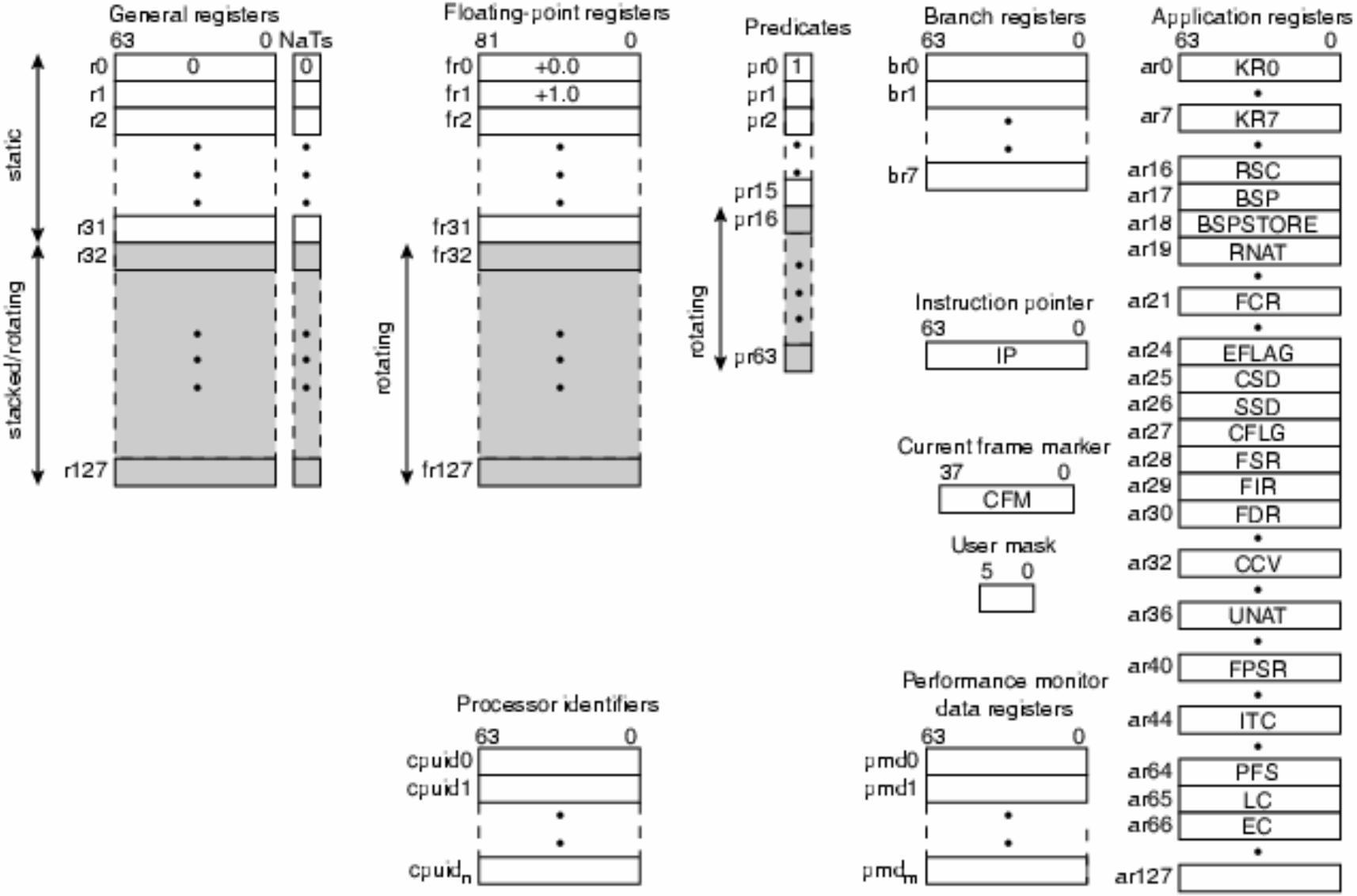
# EPIC (Explicitly parallel instruction computing)

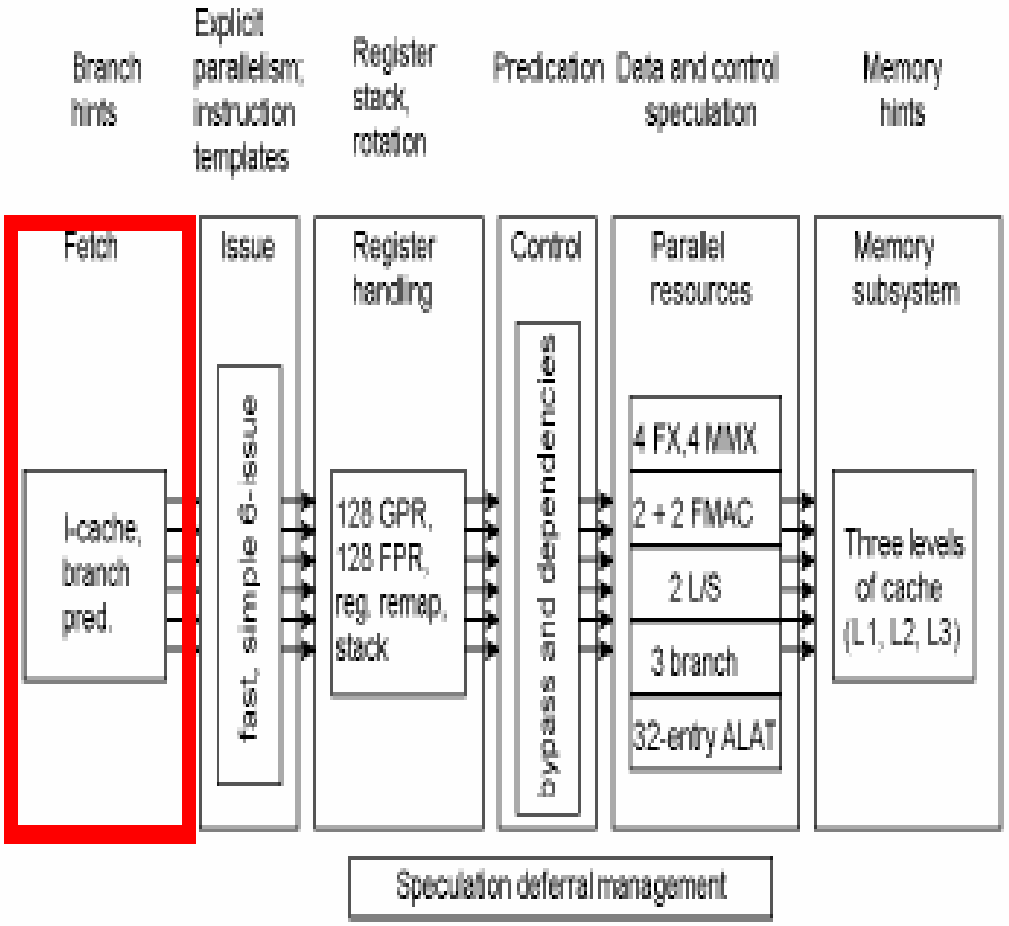
- Paralelismo a nivel de instrucción
- Instrucciones largas y muy largas (LIW/VLIW)
- Branch predication
- Carga especulativa


Branch hints
Explicit parallelism; instruction templates
Register stack, rotation
Predication
Data and control speculation
Memory hints
Fetch
Issue
Register handling
Control
Parallel resources
Memory subsystem
I-cache, branch pred.
fast, simple 6-issue
128 GPR, 128 FPR, reg. remap, stack
bypass and dependencies
4 FX,4 MMX
2 + 2 FMAC
2 L/S
3 branch
32-entry ALAT
Three levels of cache (L1, L2, L3)
Speculation deferral management

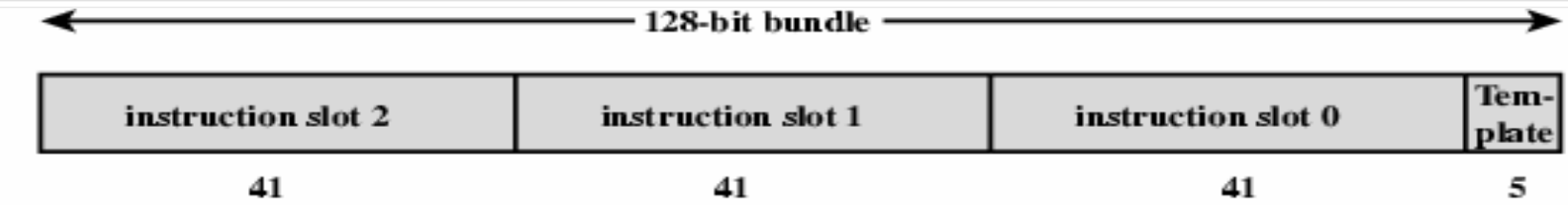



- 128 GPR. 65 bits
  - 64 bits de datos
  - 1 bit NaT (excepciones en ejecución especulativa)
- Soporte para seg. software y stack
- 128 FPR. 82 bits
- 64 registros de predicado de 1 bit
- Renombre de registros

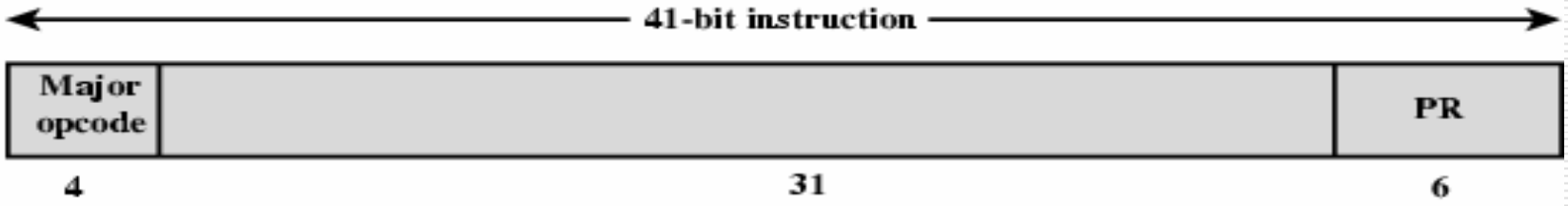

General registers
63
0 NaTs
r0
0
0
r1
r2
r31
r32
r127
static
stacked/rotating
Floating-point registers
81
0
fr0
+0.0
fr1
+1.0
fr2
fr31
fr32
fr127
rotating
Predicates
pr0
1
pr1
pr2
pr15
pr16
pr63
rotating
Branch registers
63
0
br0
br1
br7
Instruction pointer
63
0
IP
Current frame marker
37
0
CFM
User mask
5
0
Application registers
63
0
ar0
KR0
ar7
KR7
ar16
RSC
ar17
BSP
ar18
BSPSTORE
ar19
RNAT
ar21
FCR
ar24
EFLAG
ar25
CSD
ar26
SSD
ar27
CFLG
ar28
FSR
ar29
FIR
ar30
FDR
ar32
CCV
ar36
UNAT
ar40
FPSR
ar44
ITC
ar64
PFS
ar65
LC
ar66
EC
ar127
Processor identifiers
63
0
cpuid0
cpuid1
cpuid_n
Performance monitor
data registers
63
0
pmd0
pmd1
pmd_m




- Instrucciones de 41 bits
  - 7 bits para registros
  - 3 registros
  - 6 bits para predicado
  - 14 bits para código de operación
- Grupos de 3 instrucciones (*bundle*)
  - 128 bits cada bundle
  - 123 bits para instrucciones
  - 5 bits de *template*
- Predicción de saltos



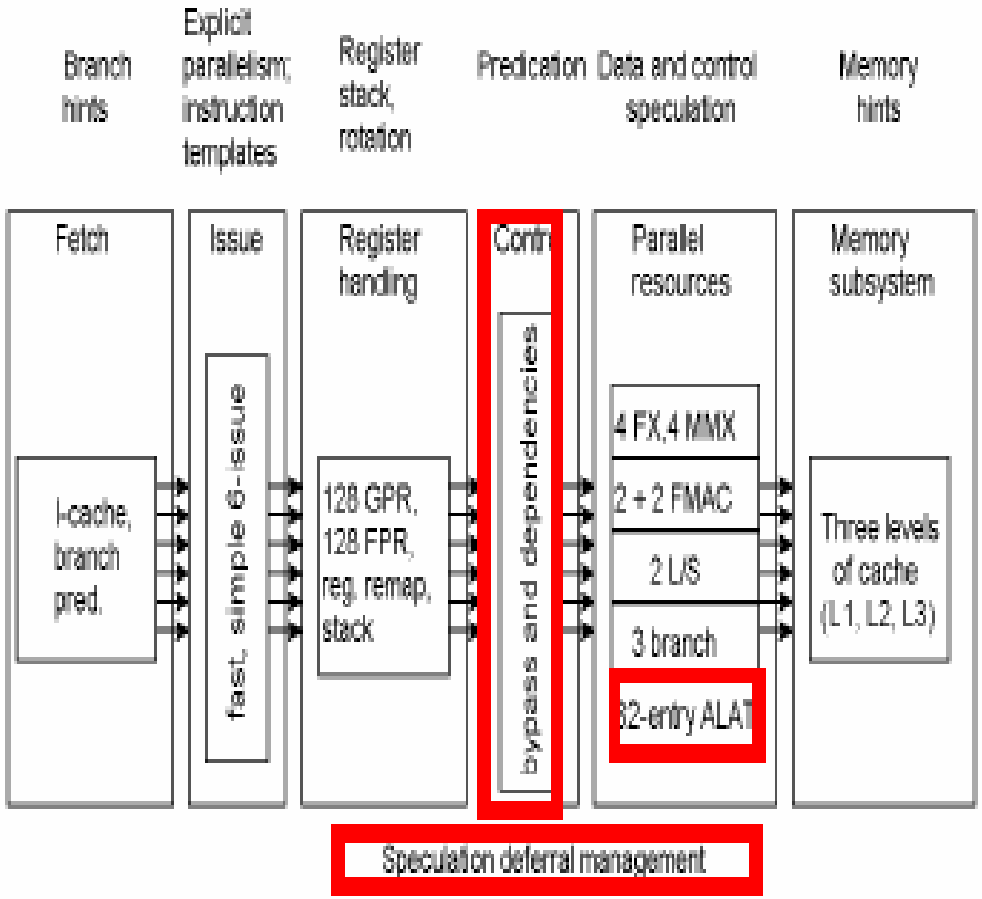

(a) IA-64 bundle



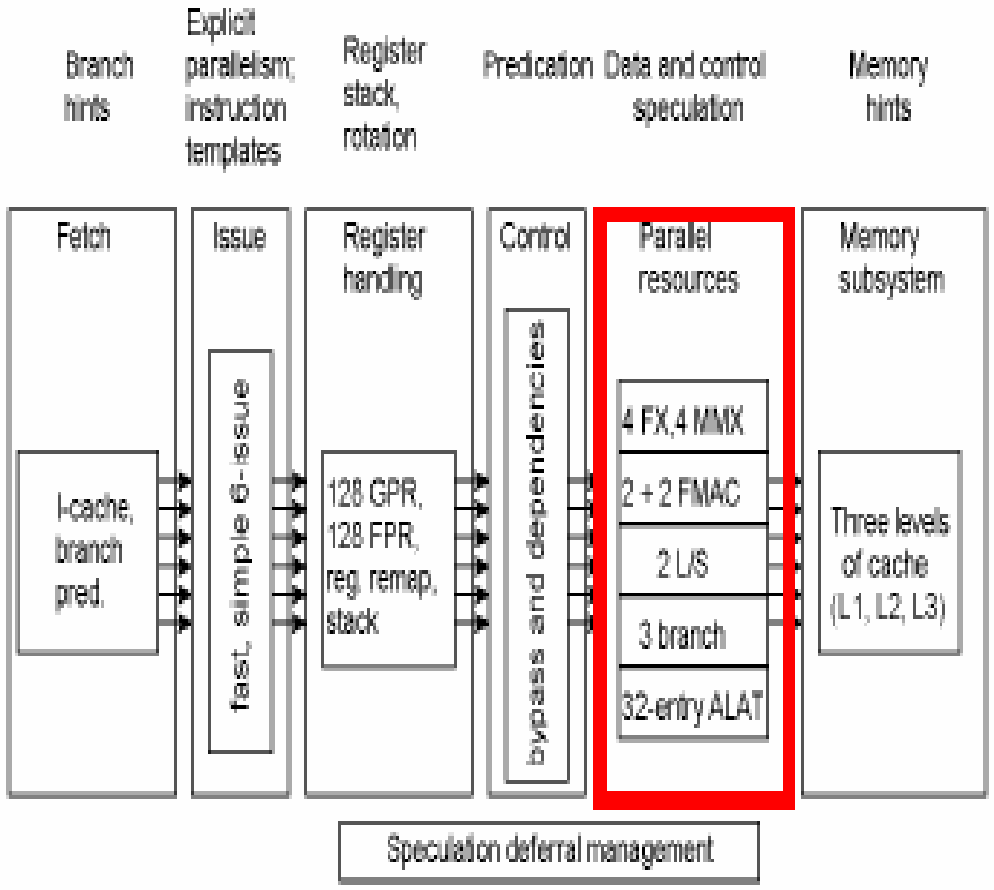

(b) General IA-64 instruction format


(c) Typical IA-64 instruction format

PR = Predicate register
GR = General or floating-point register

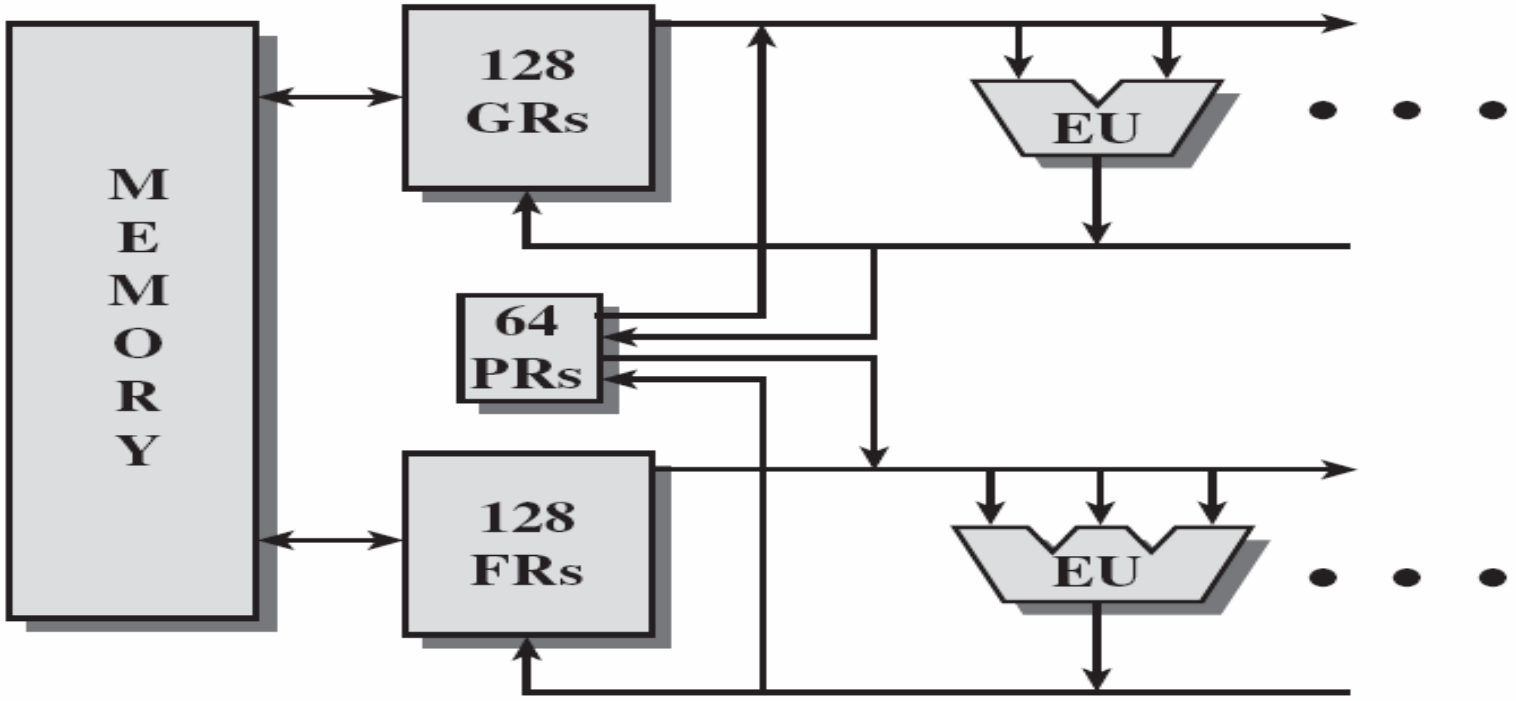



- Soporte hardware para
  - especulación de datos (loads especulativos)
  - Excepciones en ejecución especulativa
  - Dependencias no resueltas en tiempo de compliación




- Abundantes recursos para ejecución de instrucciones
  - 4 unidades enteras
  - 4 unidades multimedia
  - 2 FMAC
  - 2 unidades punto flotante de precisión simple
  - 2 unidades de load/store
  - 3 unidades de saltos

# Arquitectura interna IA-32 (1)

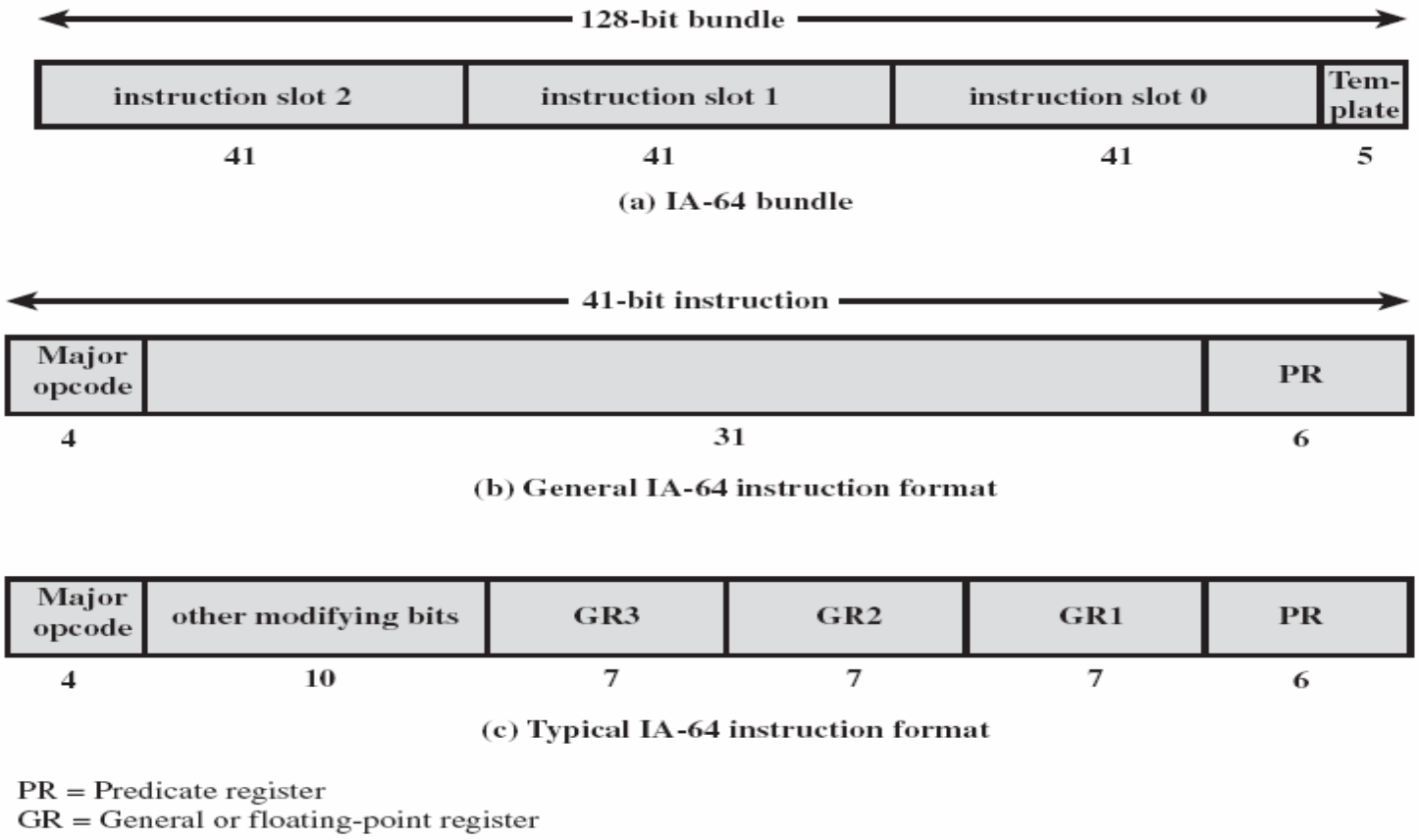



GR = General-purpose or integer register
FR = Floating-point or graphics register
PR = One-bit predicate register
EU = Execution unit

# Arquitectura interna IA-32 (2)

| Instruction Type | Description | Execution Unit Type |
|---|---|---|
| A | Integer ALU | I-unit or M-unit |
| I | Non-ALU integer | I-unit |
| M | Memory | M-unit |
| F | Floating-point | F-unit |
| B | Branch | B-unit |
| L+X | Extended | I-unit/B-unit |

# Formato de instrucciones



# Plantillas (1)

| Template | Slot 0 | Slot 1 | Slot 2 |
|---|---|---|---|
| 00 | M-unit | I-unit | I-unit |
| 01 | M-unit | I-unit | I-unit |
| 02 | M-unit | I-unit | I-unit |
| 03 | M-unit | I-unit | I-unit |
| 04 | M-unit | L-unit | X-unit |
| 05 | M-unit | L-unit | X-unit |
| 08 | M-unit | M-unit | I-unit |
| 09 | M-unit | M-unit | I-unit |
| 0A | M-unit | M-unit | I-unit |

# Plantillas (2)

| | | | |
|---|---|---|---|
| 0B | M-unit | M-unit | I-unit |
| 0C | M-unit | F-unit | I-unit |
| 0D | M-unit | F-unit | I-unit |
| 0E | M-unit | M-unit | F-unit |
| 0F | M-unit | M-unit | F-unit |
| 10 | M-unit | I-unit | B-unit |
| 11 | M-unit | I-unit | B-unit |
| 12 | M-unit | B-unit | B-unit |
| 13 | M-unit | B-unit | B-unit |

# Plantillas (3)

| | | | |
|---|---|---|---|
| 16 | B-unit | B-unit | B-unit |
| 17 | B-unit | B-unit | B-unit |
| 18 | M-unit | M-unit | B-unit |
| 19 | M-unit | M-unit | B-unit |
| 1C | M-unit | F-unit | B-unit |
| 1D | M-unit | F-unit | B-unit |

# Lenguaje ensamblador para IA-64

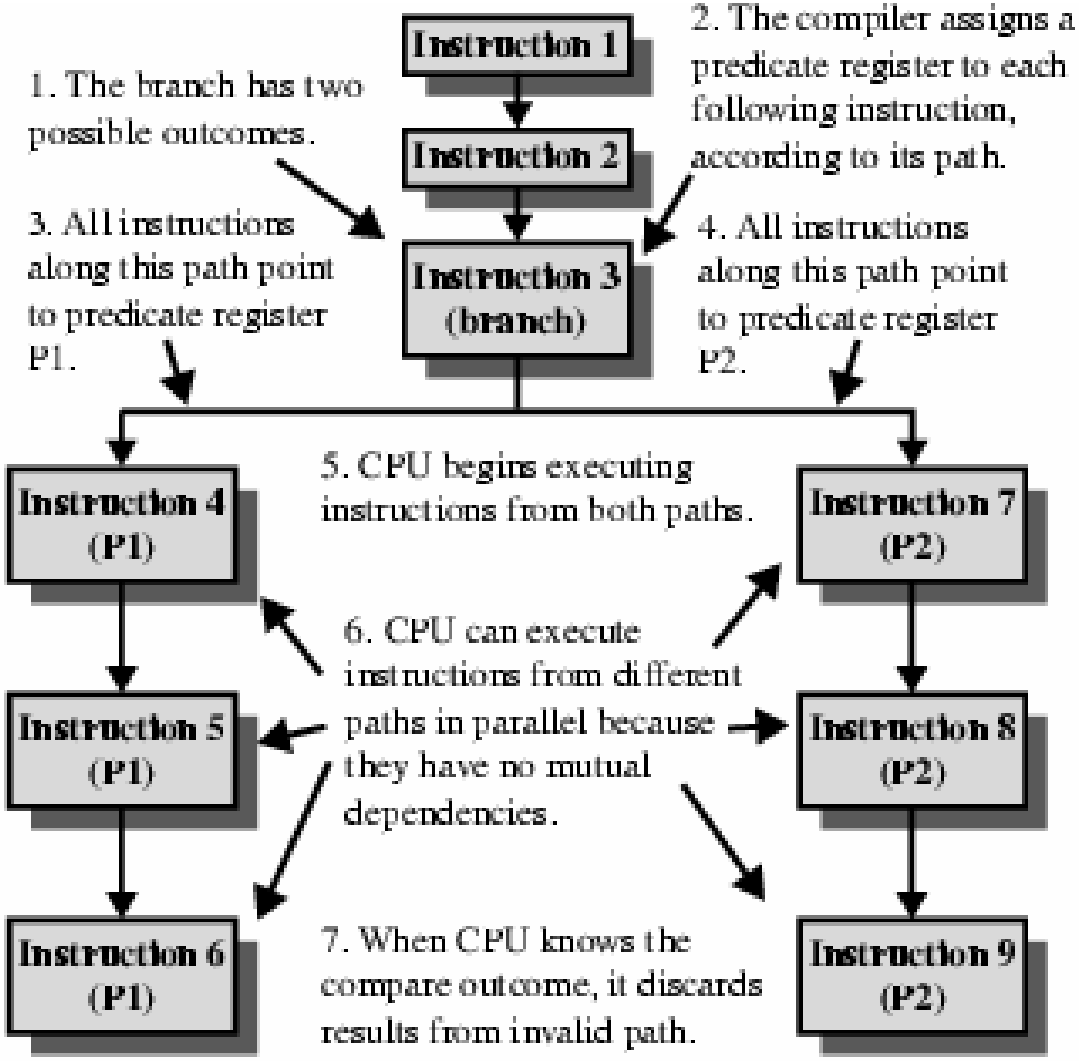
- `[qp] mnemonic [.comp] dest = srcs //`
- qp - predicate register
  - 1 at execution then execute and commit result to hardware
  - 0 result is discarded
- `mnemonic` - name of instruction
- `comp` – one or more instruction completers used to qualify mnemonic
- `dest` – one or more destination operands
- `srcs` – one or more source operands
- `//` - comment
- Instruction groups and stops indicated by `;;`
  - Sequence without read after write or write after write
  - Do not need hardware register dependency checks

# Ejemplo de instrucciones IA-64

```
ld8 r1 = [r5] ;; //first group
add r3 = r1, r4 //second group
```

- Second instruction depends on value in r1
  - Changed by first instruction
  - Can not be in same group for parallel execution

# Predication




The compiler might rearrange instructions in this order, pairing instructions 4 and 7, 5 and 8, and 6 and 9 for parallel execution.

| | | |
|---|---|---|
| Instruction 1 | Instruction 2 | Instruction 3 |
| Instruction 4 | Instruction 7 | Instruction 5 |
| Instruction 8 | Instruction 6 | Instruction 9 |

# Ejemplo de predicación

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

# Codificado en ensamblador IA-32

```
        cmp a,0
        je L1                   ; si a=0 salta a L1
        cmp b,0
        je L1                   ; si b=0 salta a L1
        add j,1                 ; j = j+1
        jmp L3
L1:     cmp c,0
        je L2                   ; si c=0 salta a L2
        add k,1                 ; k = k+1
        jmp L3
L2:     sub k,1                 ; k = k-1
L3:     add i,1         ; i = i+1
```

# Insertando predicados

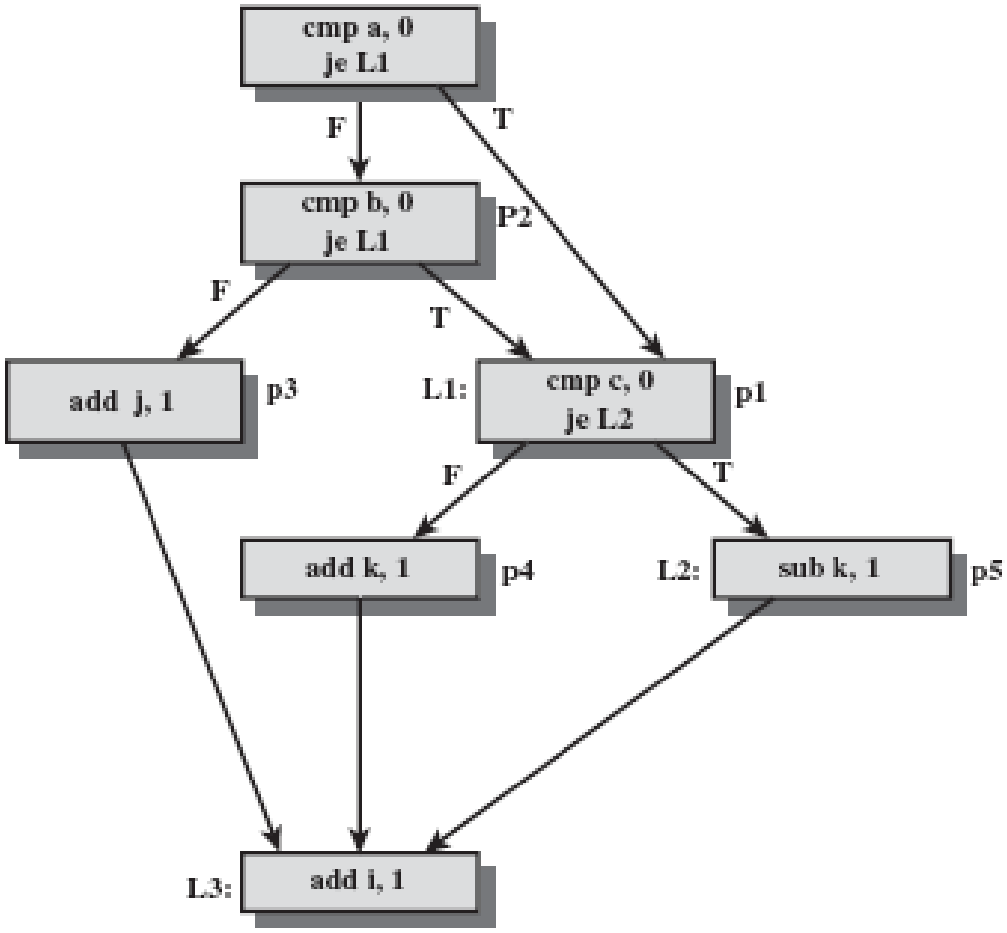



Figure 15.4 Example of Predication

# Ensamblador IA-64

```
      cmp.eq p1,p2 = 0,a ;;
(p2)  cmp.eq p1,p3 = 0,b
(p3)  add j = 1,j
(p1)  cmp.ne p4,p5 = 0,c
(p4)  add k = 1,k
(p5)  add k = -1,k
      add i = 1,i
```

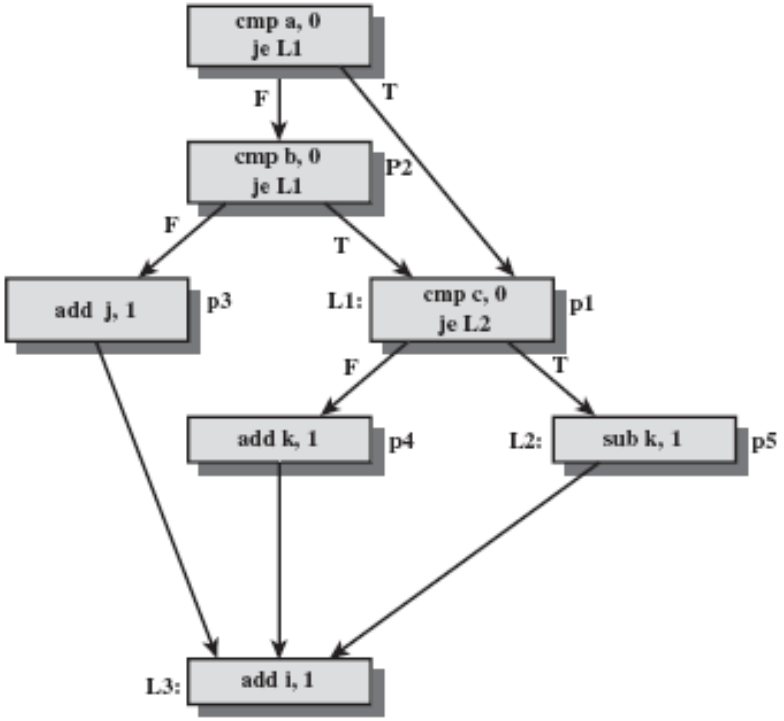



**Figure 15.4 Example of Predication**

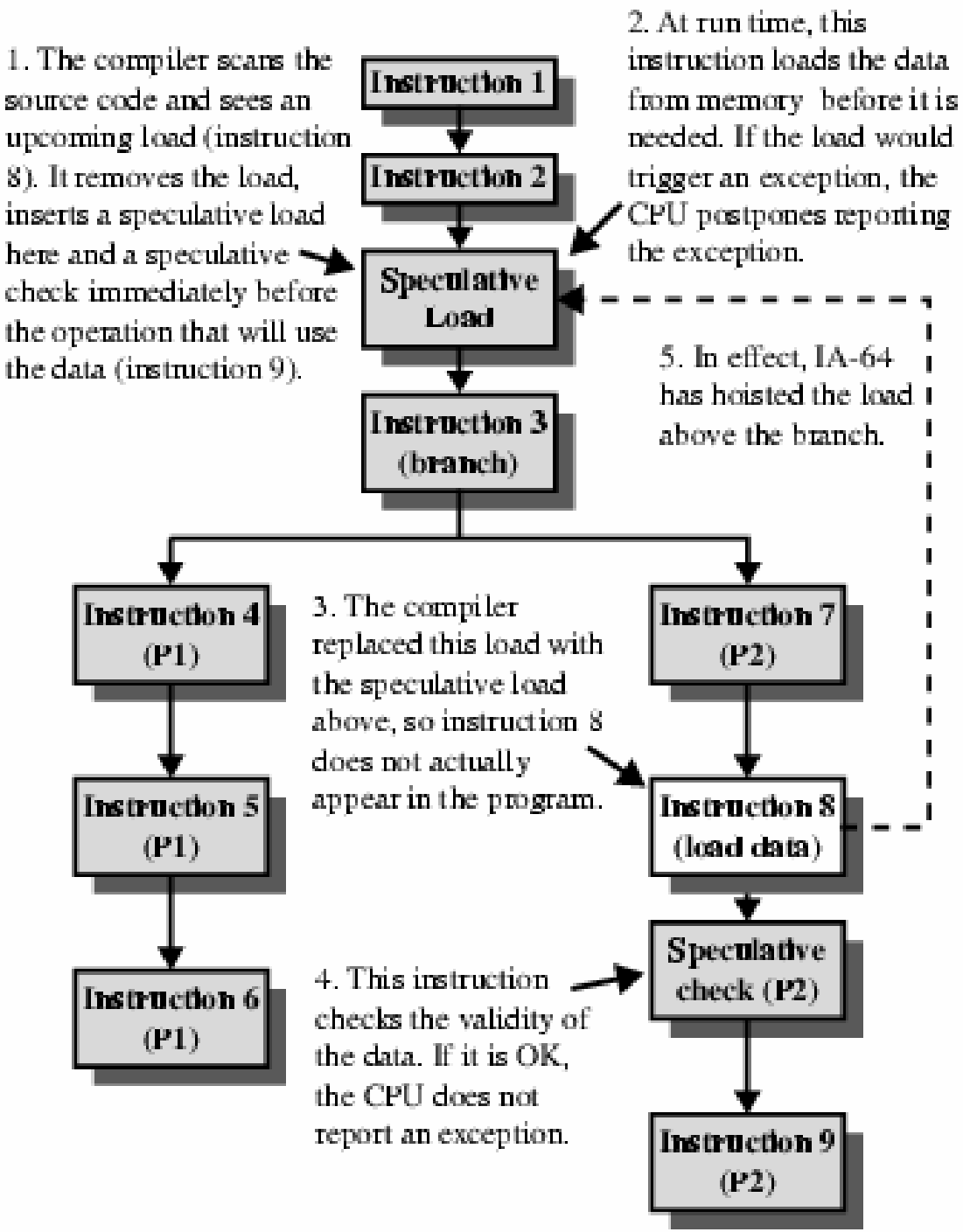
# Speculation

Consiste en la ejecución de una instrucción antes de tener la certeza de que su resultado vaya a necesitarse

- Elimina la latencia de determinadas operaciones del camino crítico del código
- Ayuda a enmascarar la alta latencia de las operaciones con memoria

- Tipos de especulación:
  - *Control Speculation*, consiste en la ejecución de una operación antes de un salto condicional que pueda controlarla
  - *Data Speculation*, es la carga de un dato desde la memoria (LOAD) antes de un STORE que *podría* modificar la misma dirección de memoria

# Speculative Loading



# Control Speculation

Las cargas especulativas de datos son también utilizadas en RISC (y en Pentium), pero :

- IA-64 define instrucciones *específicas* para la carga especulativa de datos : separa el movimiento del dato desde la memoria al registro del chequeo de las posibles excepciones generadas

- Cada registro en IA-64 contiene un bit denominado NaT (not a thing) puesto a 0 o 1 dependiendo del resultado del LOAD.

# Ejemplo de carga especulativa

```
(p1)    br etiqueta                 // ciclo 0
        ld8 r1 = [r5] ;;            // ciclo 1
        add r2 = r1,r3              // ciclo 3


        ld8.s r1 = [r5] ;;          // ciclo -2
                                    // otras inst.
(p1)  br etiqueta                   // ciclo 0
        chk.s r1, rutina            // ciclo 0
        add r2 = r1,r3              // ciclo 0
```

# Data Speculation

Es la carga de un dato desde memoria *antes* de un STORE que podría modificar su contenido

- IA-64 proporciona instrucciones específicas (cargas anticipadas y chequeos) para habilitar esta funcionalidad
- Requieren de una estructura interna (ALAT : Advanced Load Address Table) en el procesador

| Registro # | Direccion | Tamano |
|---|---|---|

- Las cargas anticipadas generan una entrada en la ALAT
- Los STOREs, chequean si existe una entrada al mismo registro y con direcciones afectadas, si es así borran la entrada
- El chequeo comprueba si la entrada en la tabla permanece intacta (la carga es válida) o ha desaparecido (la carga ya no es válida puesto que ha habido un STORE anterior)

# Ejemplo de carga adelantada

```
st8 [r4] = r12                  // ciclo 0
ld8 r6 = [r8] ;;                // ciclo 0
add r5 = r6,r7 ;;               // ciclo 2
st8 [r18] = r5                  // ciclo 3

ld8.a r6 = [r8] ;;              // ciclo -2
                                // otras inst
st8 [r4] = r12                  // ciclo 0
ld8.c r6 = [r8]                 // ciclo 0
add r5 = r6,r7 ;;               // ciclo 0
st8 [r18] = r5                  // ciclo 1
```

# Software Pipelining

```
Y[i] = x[i] + c          desde i= 0 hasta 4
```

```
L1:   ld4 r4=[r5],4 ;; //cycle 0 load postinc 4
      add r7=r4,r9 ;;  //cycle 2
      st4 [r6]=r7,4    //cycle 3 store postinc 4
      br.cloop L1 ;;   //cycle 3
```

- Si no hay conflicto entre las direcciones apuntadas por r5 y r6 podemos mejorar el rendimiento mediante segmentación software.

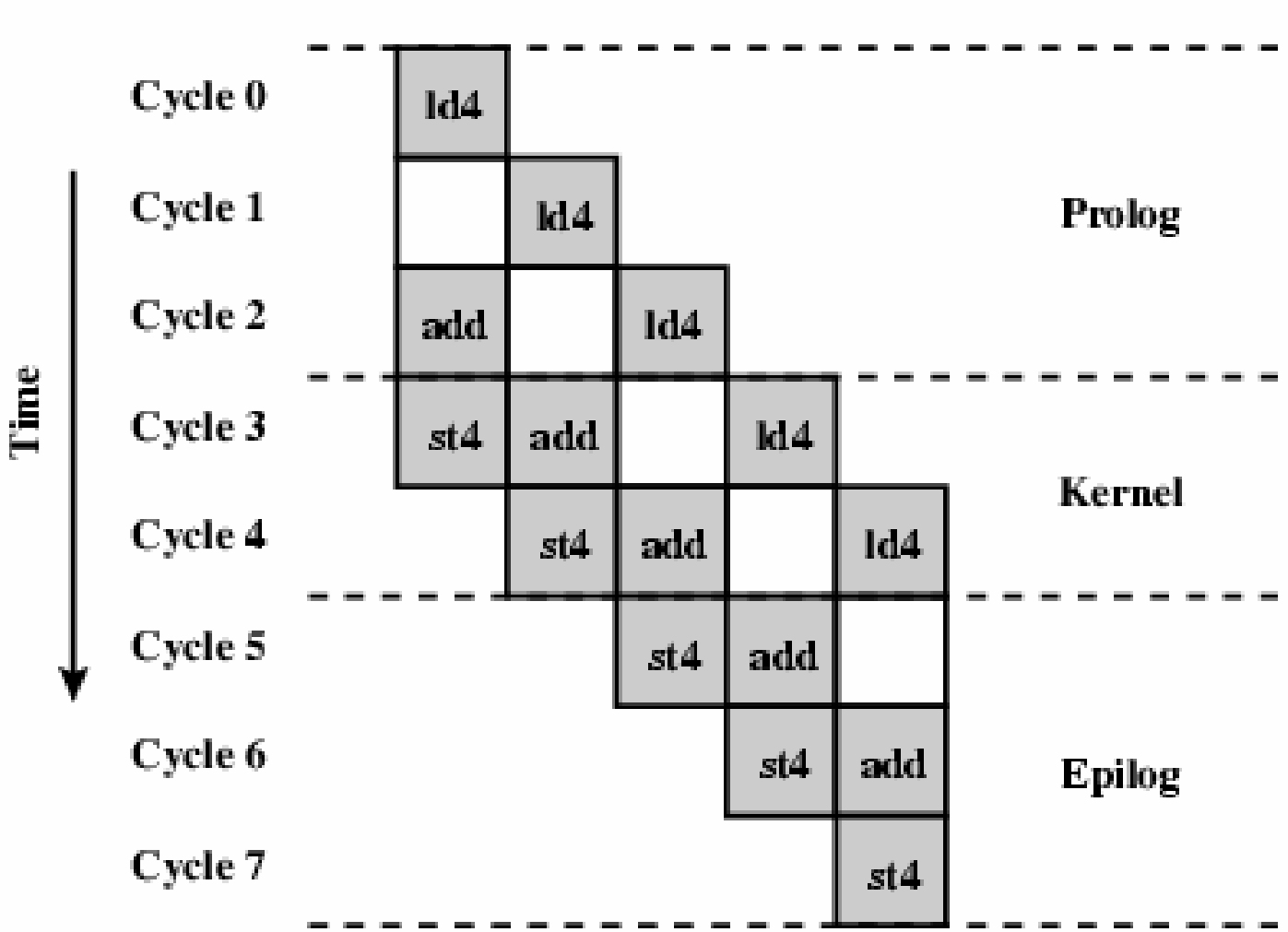
# Podemos reescribir el programa

```
ld4 r32=[r5],4;; //cycle 0
ld4 r33=[r5],4;; //cycle 1
ld4 r34=[r5],4   //cycle 2
add r36=r32,r9;; //cycle 2
ld4 r35=[r5],4   //cycle 3
add r37=r33,r9   //cycle 3
st4 [r6]=r36,4;; //cycle 3
ld4 r36=[r5],4   //cycle 3
add r38=r34,r9   //cycle 4
st4 [r6]=r37,4;; //cycle 4
add r39=r35,r9   //cycle 5
st4 [r6]=r38,4;; //cycle 5
add r40=r36,r9   //cycle 6
st4 [r6]=r39,4;; //cycle 6
st4 [r6]=r40,4;; //cycle 7
```

# El resultado es:

- Completamos las 5 iteraciones en 7 ciclos frente a los 20 ciclos del programa inicial
  - Suponemos la existencia de dos puertos de memoria
  - Las operaciones Load y Store se pueden ejecutar en paralelo ( tenemos dos unidades funcionales distintas)

# Software Pipeline



# Soporte para Software Pipelining

- Renombrado automático de registros
  - Un bloque fijo del fichero de registros de predicado y de punto flotante (p16-P32, fr32-fr127) y un bloque variable del area de registros generales (max r32-r127) están preparados para esta operación.
- Predicación
  - Las instrucciones del bucle llevan predicados para determinar la fase en la que se encuentra
- Instrucciones de final de bucle

# Programa IA-64 con segmentación software

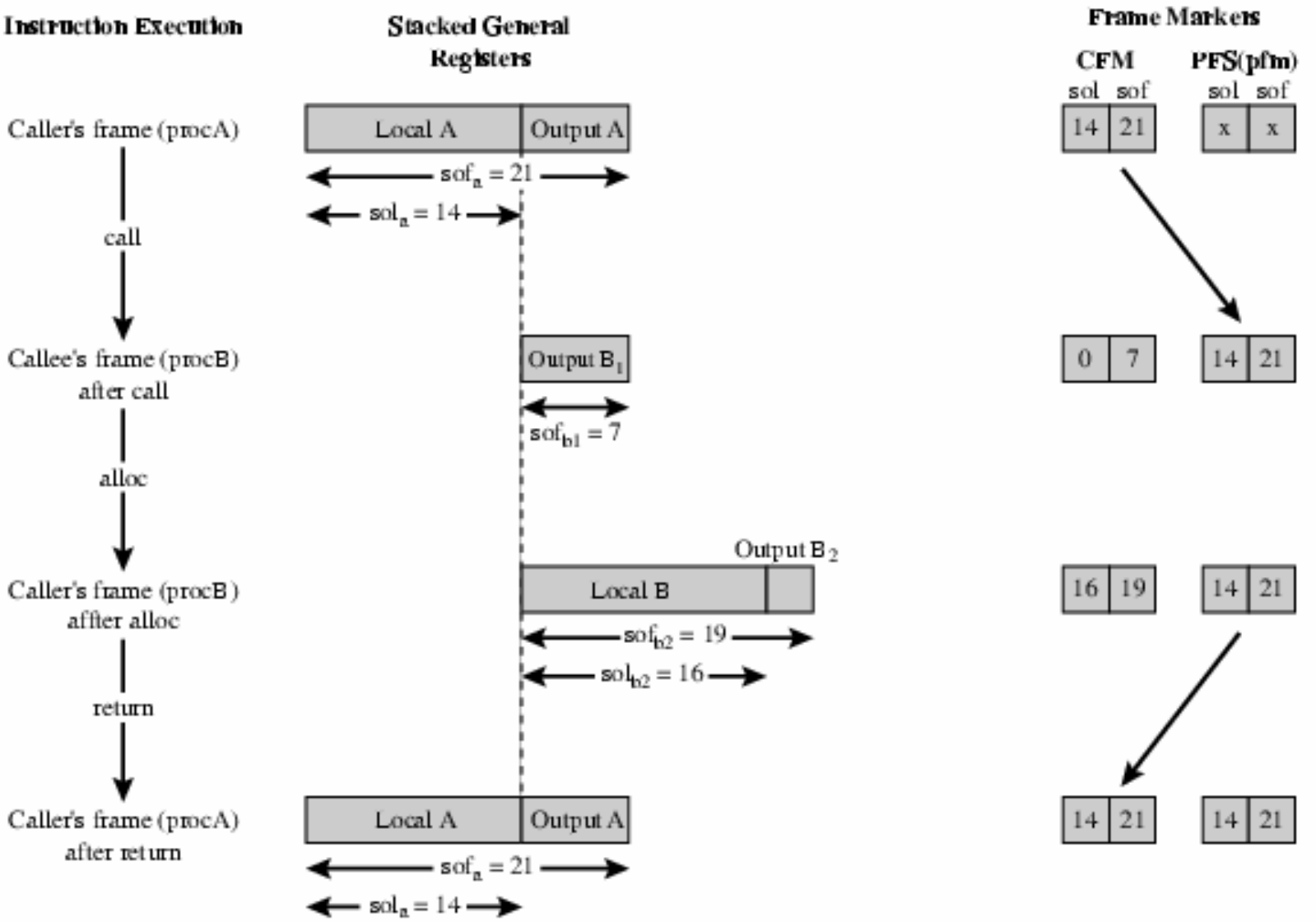
```
        mov lc = 199     //  para 200 iteraciones
        mov ec = 4       // contador de etapas de epílogo +1
        mov pr.rot = 1<<16;; // inicializa reg. de predicado
L1:     (pr16)  ld4 r32 = [r5],4          // ciclo 0
        (p17)                             // etapa vacía
        (p18)   add r38  = r34 , r9       // ciclo 0
        (p19)   st4 [r6] = r39,4          // ciclo 0
                br.ctop L1;;              // ciclo 0
```

# Traza del bucle utilizando segmentación software. Ejemplo

| Cycle | Execution Unit/Instruction | | | | State before br.ctop | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | M | I | M | B | P16 | P17 | P18 | P19 | LC | EC |
| 0 | ld4 | | | br.ctop | 1 | 0 | 0 | 0 | 199 | 4 |
| 1 | ld4 | | | br.ctop | 1 | 1 | 0 | 0 | 198 | 4 |
| 2 | ld4 | add | | br.ctop | 1 | 1 | 1 | 0 | 197 | 4 |
| 3 | ld4 | add | st4 | br.ctop | 1 | 1 | 1 | 1 | 196 | 4 |
| ••• | ••• | ••• | ••• | ••• | ••• | ••• | ••• | ••• | ••• | ••• |
| 100 | ld4 | add | st4 | br.ctop | 1 | 1 | 1 | 1 | 99 | 4 |
| ••• | ••• | ••• | ••• | ••• | ••• | ••• | ••• | ••• | ••• | ••• |
| 199 | ld4 | add | st4 | br.ctop | 1 | 1 | 1 | 1 | 0 | 4 |
| 200 | | add | st4 | br.ctop | 0 | 1 | 1 | 1 | 0 | 3 |
| 201 | | add | st4 | br.ctop | 0 | 0 | 1 | 1 | 0 | 2 |
| 202 | | | st4 | br.ctop | 0 | 0 | 0 | 1 | 0 | 1 |
| | | | | | 0 | 0 | 0 | 0 | 0 | 0 |

# Pila de registros

- Facilita el paso de parámetros entre procedimientos
- El compilador indica el nº de registros locales y de salida necesarios para cada procedimiento que mediante la instrucción alloc se asignan a cada procedimiento
- El procesador renombra los registros asignados de forma que siempre comienzan por r32.
- El control del número de registros asignados a cada procedimiento se lleva con los registros CFM y PFS.


Instruction Execution
Stacked General Registers
Frame Markers
CFM
PFS(pfm)
sol sof
sol sof
Caller's frame (procA)
Local A
Output A
14 21
x x
$sof_a = 21$
$sol_a = 14$
call
Callee's frame (procB) after call
Output $B_1$
0 7
14 21
$sof_{b1} = 7$
alloc
Output $B_2$
Caller's frame (procB) after alloc
Local B
16 19
14 21
$sof_{b2} = 19$
$sol_{b2} = 16$
return
Caller's frame (procA) after return
Local A
Output A
14 21
14 21
$sof_a = 21$
$sol_a = 14$

# Itanium Processor Diagram