Arquitectura de Computadores

Capítulo 4.
Técnicas Avanzadas en Paralelismo. Parte 3

Based on the original material of the book:
D.A. Patterson y J.L. Hennessy "Computer Organization and Design: The Hardware/Software Interface" 4th edition.

Escuela Politécnica Superior

Universidad Autónoma de Madrid

Profesores:
G131: Iván González Martínez
G130 y 136: Francisco Javier Gómez Arribas

Instruction-level parallelism

- Parallelism at the machine-instruction level
- The processor can re-order, pipeline instructions, split them into microinstructions, do aggressive branch prediction, etc.
- Instruction-level parallelism enabled rapid increases in processor speeds over the last 15 years

Thread-level parallelism (TLP)

- This is parallelism on a more coarser scale
- Server can serve each client in a separate thread (Web server, database server)
- A computer game can do AI, graphics, and sound in three separate threads
- Single-core superscalar processors cannot fully exploit TLP
- Multi-core architectures are the next step in processor evolution: explicitly exploiting TLP

Multiple Issue

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

Multiple Issue

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

Data Dependency Hazards

- Dependencies between two given instructions (i runs before j):
  - RAW (Read After Write): j tries to read a register before i writes it.
  - WAR (Write After Read): j writes a register before i reads it.
  - WAW (Write After Write): j writes a register before i has done it.

✓ Examples:

RAW

```
ADD  r1, r2, r3
SUB  r5, r1, r6
AND  r6, r5, r1
ADD  r4, r1, r3
SW   r10, 100(r1)
```

WAR

```
ADD r1, r2, r3
OR  r3,r4, r5
```

WAW

```
DIV  r1, r2, r3
AND r1,r4, r5
```

Segmented micros with in-order issue and finish have just RAW

Static Multiple Issue

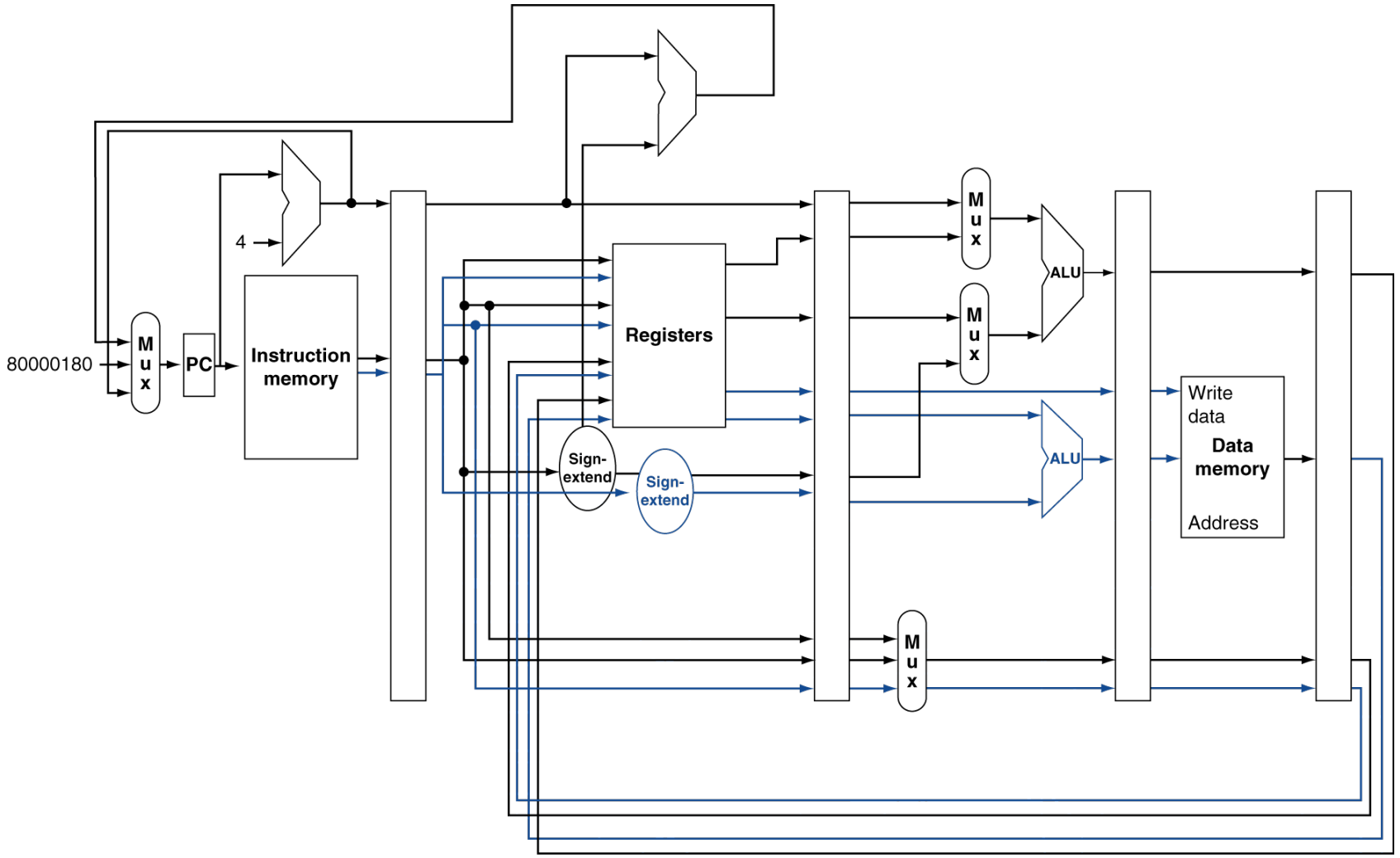
- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - Possibly dependencies between packets
  - Pad with nop if necessary

MIPS with Static Dual Issue

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

MIPS with Static Dual Issue



Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can't use ALU result in load/store in same packet
    - ```
      add  $t0, $s0, $s1
      load $s2, 0($t0)
      ```
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required
      ```

Scheduling Example

- Schedule this for dual-issue MIPS

```
for (i=n; i>0; i--)
  A[i] = A[i] + cte;
```

```
Loop: lw   $t0, 0($s1)       # $t0=array element
      addu $t0, $t0, $s2     # add scalar in $s2
      sw   $t0, 0($s1)       # store result
      addi $s1, $s1,-4       # decrement pointer
      bne  $s1, $zero, Loop  # branch $s1!=0
```

	ALU/branch	Load/store	cycle
Loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1,-4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	4

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called “register renaming”
  - Avoid loop-carried “anti-dependencies”
    - Store followed by a load of the same register
    - Aka “name dependence”
      - Reuse of a register name

Loop Unrolling Example

	ALU/branch	Load/store	cycle
Loop:	addi $s1, $s1,-16	lw $t0, 0($s1)	1
	nop	lw $t1, 12($s1)	2
	addu $t0, $t0, $s2	lw $t2, 8($s1)	3
	addu $t1, $t1, $s2	lw $t3, 4($s1)	4
	addu $t2, $t2, $s2	sw $t0, 16($s1)	5
	addu $t3, $t3, $s2	sw $t1, 12($s1)	6
	nop	sw $t2, 8($s1)	7
	bne $s1, $zero, Loop	sw $t3, 4($s1)	8

- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of registers and code size

```
for (i=n; i>0; i--)
  A[i] = A[i] + cte;
```

```
for (i=n; i>0; i=i-4) {
  A[i] = A[i] + cte; A[i-1] = A[i-1] + cte;
  A[i-2] = A[i-2] + cte; A[i-3] = A[i-3] + cte; }
```

Hazards in a Multiple Issue Processor

- In-order issue and finish (just RAW)
- In-order issue and out-of-order finish (RAW and WAW)
- Out-of-order issue and finish (RAW, WAR and WAW)

Example: Superscalar system that issues 2 instructions per cycle.

Run the following program:

```
I1:  ADDF  R1,R2,R3   (2c)        I4:  AND   R10,R11,R12
I2:  SUB   R4,R5,R6               I5:  SUB   R13,R10,R14
I3:  OR    R7,R8,R9               I6:  ADD   R15,R16,R17
```

Multiple Issue

1.- In-order issue and finish.

I1: ADDF R1,R2,R3 (2c)
I2: SUB R4,R5,R6
I3: OR R7,R8,R9
I4: AND R10,R11,R12
I5: SUB R13,R10,R14
I6: ADD R15,R16,R17

Data forwarding.
2 instructions are always fetched at the same time and they finish at the same time. Also, they are executed at the same time unless some risk exists.

		T1	T2	T3	T4	T5	T6	T7	T8
DEC	DEC1	I1	I3	$I3^D$		I5			
	DEC2	I2	I4	$I4^D$	$I4^D$	I6	$I6^D$		
EXE	EJ1(+)		I2	$I2^D$			I5	I6	
	EJ2(+CF)		I1	I1					
	LOGIC				I3	I4			
WR	WR1				I1	$I3^D$	I3	$I5^D$	I5
	WR2				I2		I4		I6

Multiple Issue

2.- In-order issue and out-of-order finish.

I1: ADDF R1,R2,R3 (2c)
I2: SUB R4,R5,R6
I3: OR R7,R8,R9
I4: AND R10,R11,R12
I5: SUB R13,R10,R14
I6: ADD R15,R16,R17

Data forwarding.
2 instructions are always fetched at the same time.
Exception/interruption, old instructions have NOT finished and new instructions have finished.
What to do? What to repeat? Can the exception be attended inmediately?
Solution: retire/commit in order (after finishing)

		T1	T2	T3	T4	T5	T6	T7
DEC	DEC1	I1	I3		I5			
	DEC2	I2	I4	$I4^D$	I6	$I6^D$		
EXE	EXJ1(+)		I2			I5	I6	
	EX2(+CF)		I1	I1				
	LOGIC			I3	I4			
WR	WR1				I1		I5	
	WR2			I2	I3	I4		I6

Multiple Issue

3.- Out-of-order issue and finish.

I1: ADDF R1,R2,R3 (2c)
I2: SUB R4,R5,R6
I3: OR R7,R8,R9
I4: AND R10,R11,R12
I5: SUB R13,R10,R14
I6: ADD R15,R16,R17

Data forwarding

Reservation stations

		T1	T2	T3	T4	T5	T6
DEC	DEC1	I1	I3	I5			
	DEC2	I2	I4	I6			
Reservation stations				I4	I5		
EX	EX1(+)		I2		I6	I5	
	EX2(+CF)		I1	I1			
	LOGIC			I3	I4		
WR	WR1			I2	I1	I6	I5
	WR2				I3	I4	

Dynamic Multiple Issue

- “Superscalar” processors
- CPU decides whether to issue 0, 1, 2, … each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

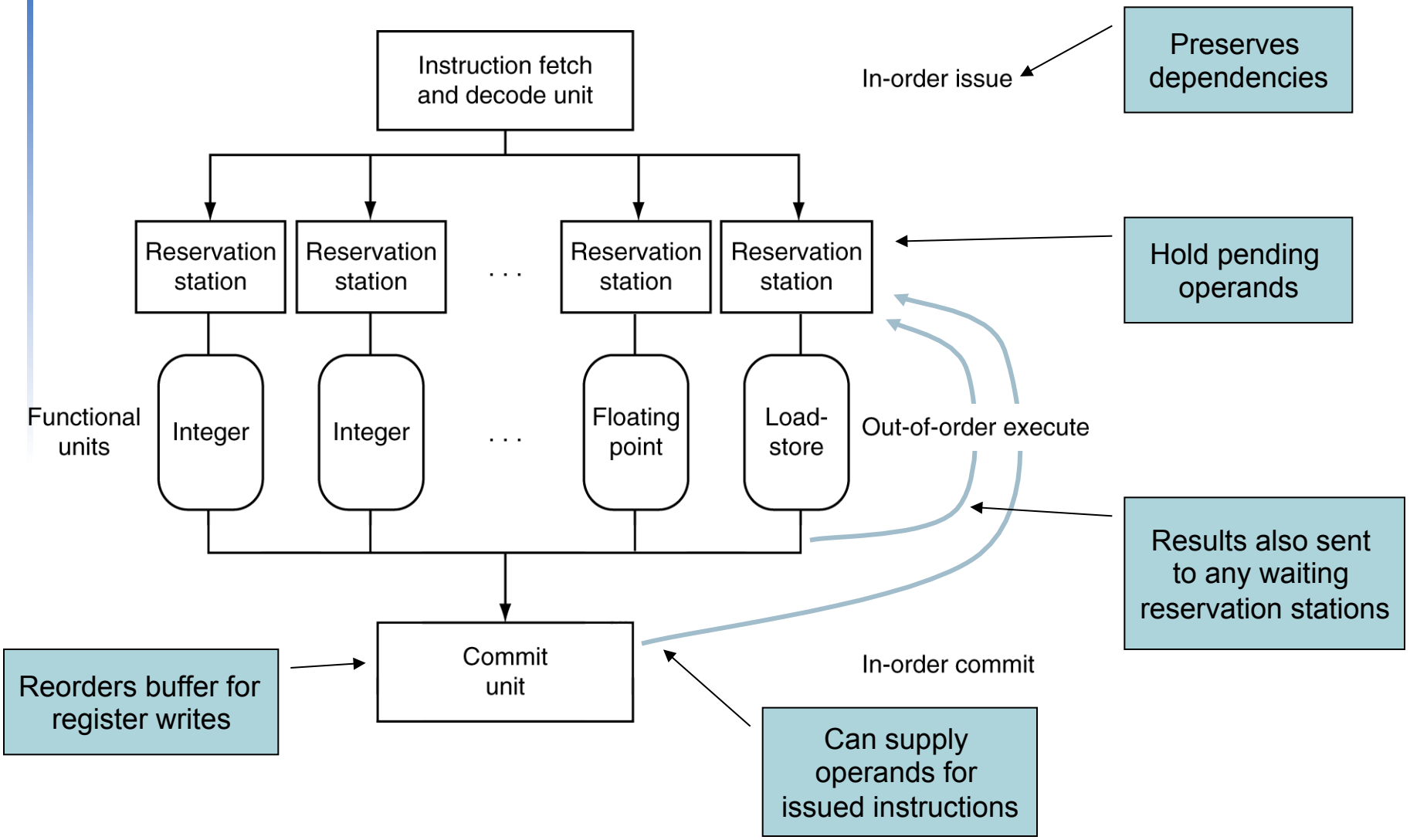
Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example

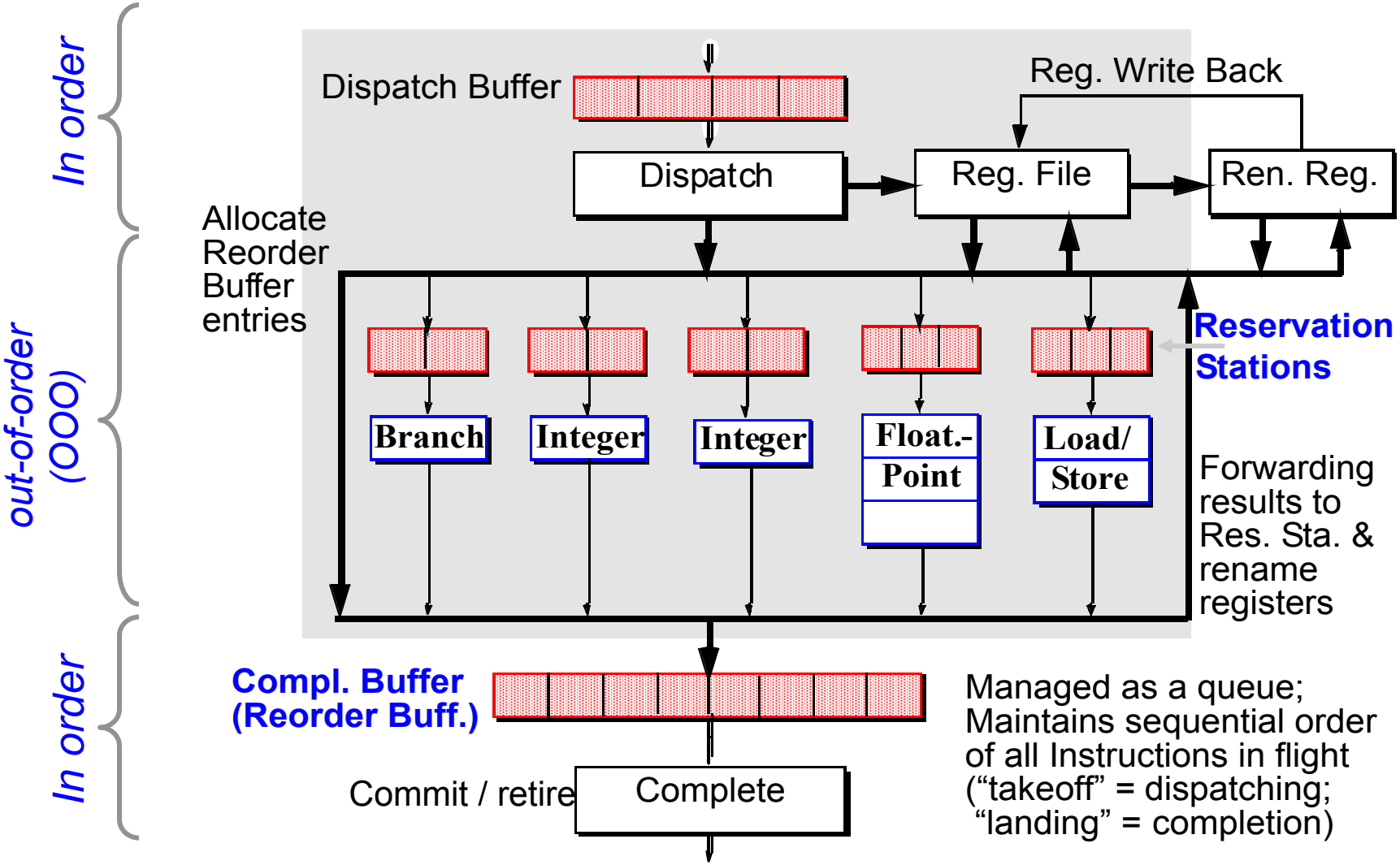
```
lw    $t0, 20($s2)
addu  $t1, $t0, $t2
sub   $s4, $s4, $t3
slti  $t5, $s4, 20
```

  - Can start sub while addu is waiting for lw

Dynamically Scheduled CPU



Dynamically Scheduled CPU

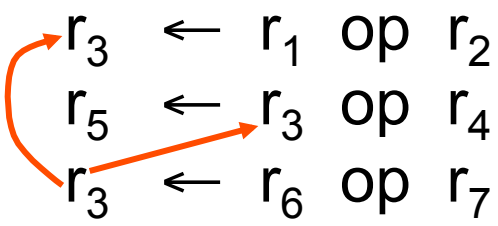


Register Renaming

- Reservation stations and reordering buffer effectively provide register renaming.
- On instruction issue to reservation station:
  - If operand is available in register file or reorder buffer:
    - Copied to reservation station.
    - No longer required in the register; can be overwritten.
  - If operand is not yet available:
    - It will be provided to the reservation station by a function unit.
    - Register update may not be required.

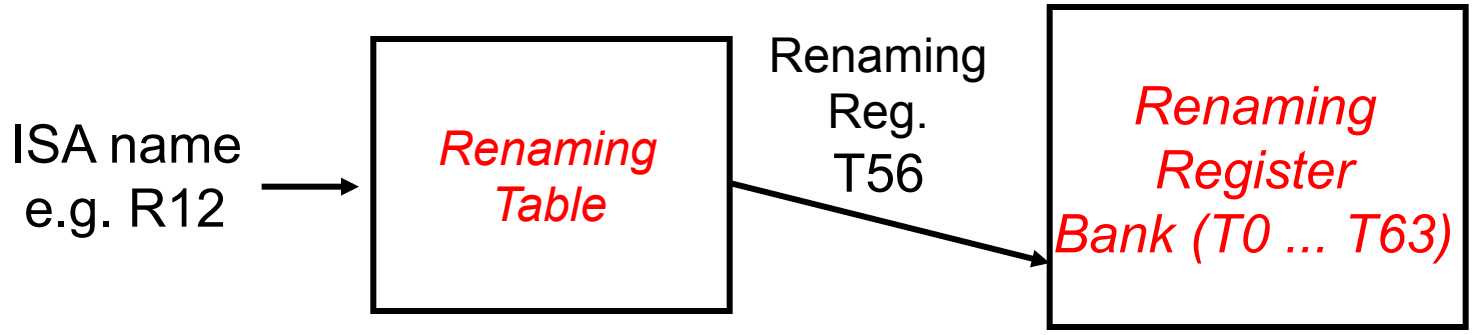
Register Renaming

WAR and WAW are false dependencies.



Original	Renaming
r1 ← r2 / r3	r11 ← r2 / r3
r4 ← r1 * r5	r14 ← r11 * r5
r1 ← r3 + r6	r21 ← r3 + r6
r3 ← r1 - r4	r19 ← r21 – r14

Register Renaming



- Pointer between ISA register and renaming registers.
- Instruction issued that writes a Dest. Reg. RD:
  - Reserve a renaming register not in use TX.
  - Register the RD and TX link.
- Unit Control decides:
  - When the link is removed.
  - When RD is updated.
  - When the reserve is removed.

R1 ← R2 / R3

R4 ← R1 * R5

R1 ← R3 + R6

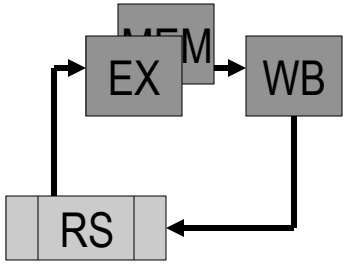
Buffer de renombramiento con acceso asociativo

Búsqueda asociativa por número de registro=3 (T02=R3)

	Asignación Válida	Registro destino	Contenido del registro	Contenido válido	Bit de última asignación
T00	0	7	67	1	1
T01	1	2	102	1	0
T02	1	3	23	1	1
T03	1	2	22	0	1
T04	…	…			

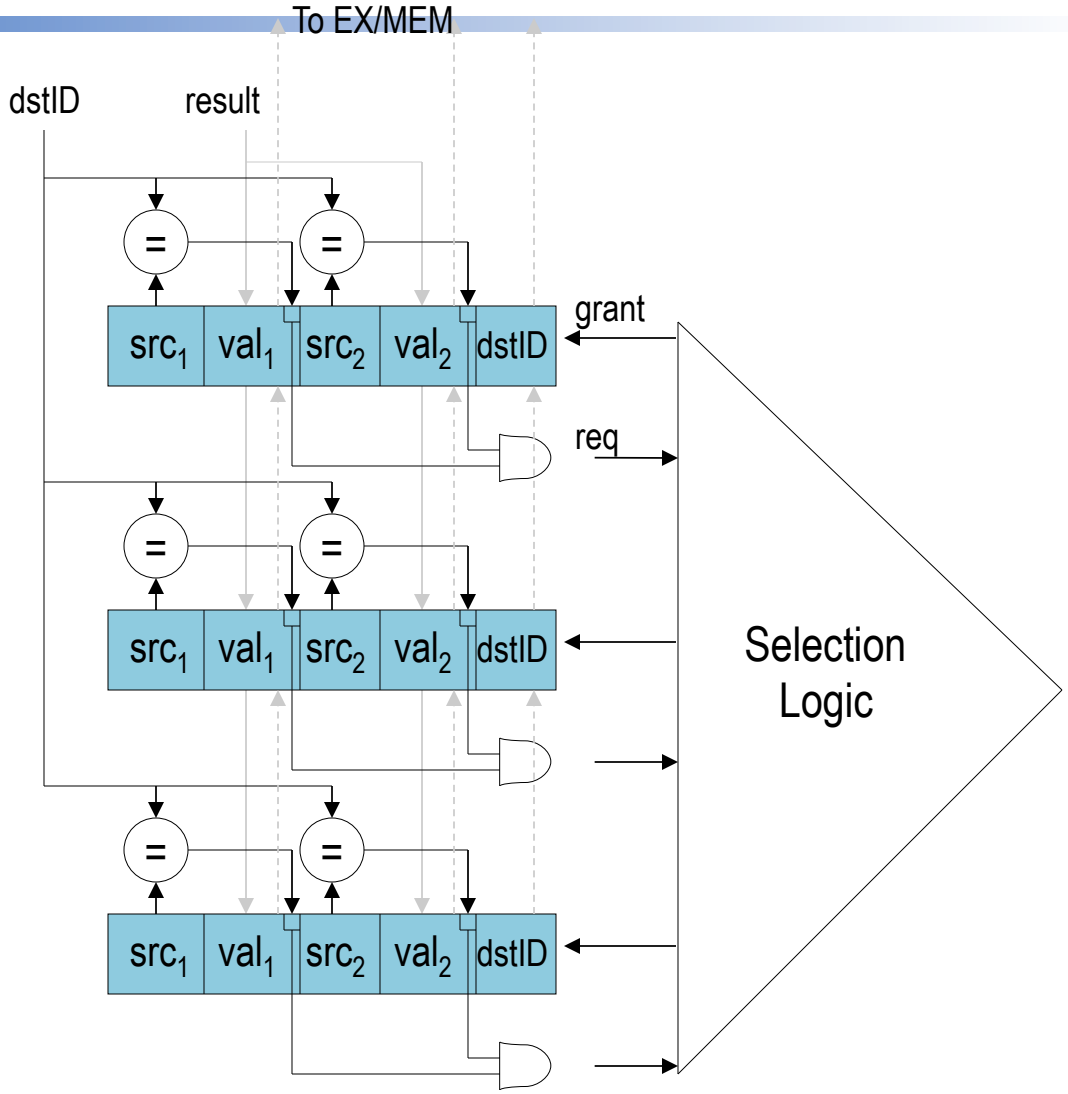
(T02=>R3=23)

Funcionamiento de las estaciones de reserva



Si el registro destino (dstID) coincide con uno de los operandos se actualiza la estación de reserva

Cuando ambos operandos están presentes (Val1 y Val2) se puede ejecutar.

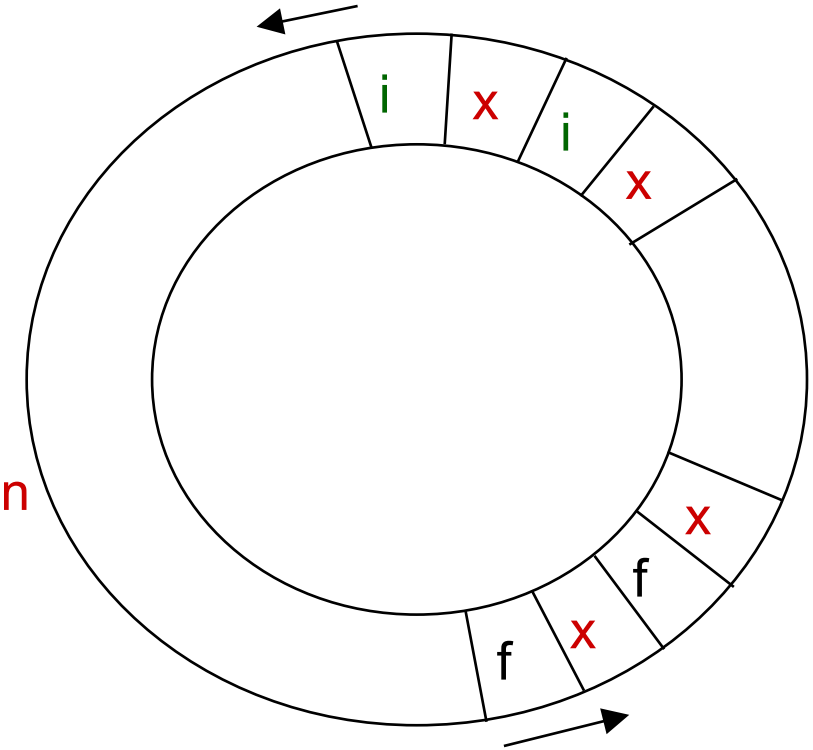


Reordering Buffer

- **Reordering buffer (ROB): As a circle FIFO.**
  - ROB entries are reserved in dispatch/issue and are removed after commit.
  - When a instruction finishes the execution step, updates its state in its entry in ROB.
  - Within the execution, the renaming registers are used.
- **Aim: ROB maintains the original order of the instructions in the program (before issue/dispatch) and allows the instructions finish in order.**

Reordering buffer

Instructions entrance

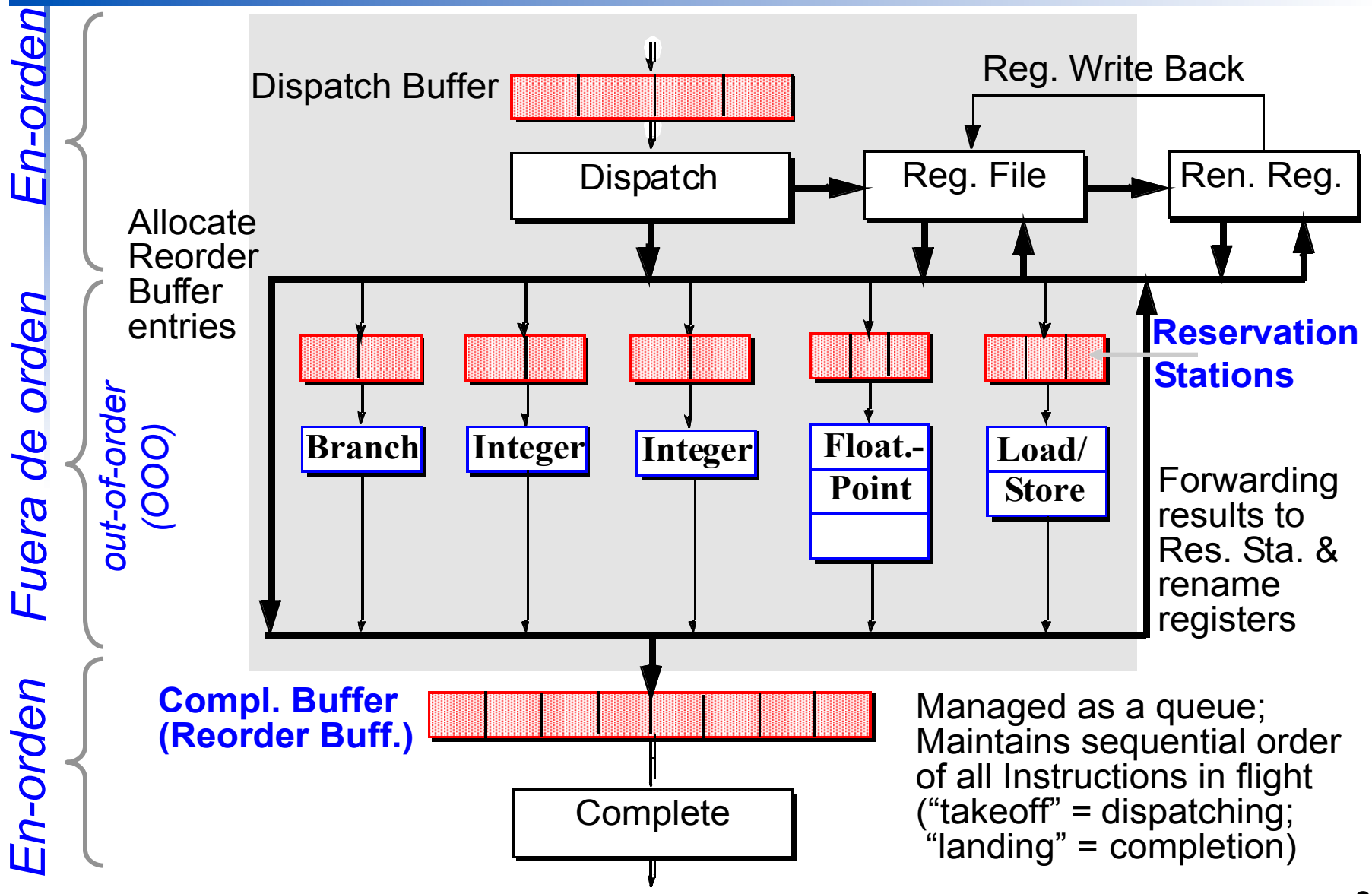


Knows in which step each instruction is.

i: issued
x: in execution
f: finished

Instruction commit/retire

Superescalar Datapath



Ejemplo: Pasos para la ejecución con planificación dinámica

- **Despacho (DISPATCH):**
  - Leer operandos del banco de registros (Arch. Reg. File -ARF) y/o el banco de registros de renombrado (Rename Register File-RRF)
    - *(RRF devuelve el valor o una etiqueta.)*
  - Reservar entrada en RRF y renombrar el registro destino
  - Reservar entrada en Buffer de Reordenamiento (ROB)
  - Avanzar la instrucción a la estación de reserva (RS) apropiada
- **Ejecución (EXECUTE):**
  - Entradas de RS monitorean el bus de resultados en busca de etiquetas de registros renombrados para capturar operandos pendientes.
  - Cuando todos los operandos están listos, se emite la instrucción a la UF y se libera la entrada de la estación de reserva (ejecución segmentada sin paradas)
  - Cuando la ejecución termina, difunde el resultado a entradas que esperen de RS y RRF.
- **Finalización (COMPLETE):**
  - Cuando este listo a finalizar (commit) en orden:
    1. Actualiza la entrada AFR desde RRF, libera la entrada RRF; y si es un store la envía al buffer de store.
    2. Libera la entrada ROB y la instrucción se considera arquitec. completada

Buffer de renombramiento con estaciones de reserva

I1: MULT R3 R1 R2
I2: ADD R4 R2 R3
I3: SUB R3 R5 R2

Banco de Registros

R1	10
R2	22
R3	30
R4	40
R5	50
…	…

BUFFER DE RENOMBRAMIENTO

dstID	A.Val.	RDest	Valor	Valido	Ult
BR1	0				
BR2	0				
BR3	0				
BR4	0				
BR5	0				
…					

BUFFER de Reordenación

Instr	etapa	dstID

ESTACIONES DE RESERVA

Func	Op1	Tag1	Op2	Tag2	dstID

ADD/SUB (2c)

ESTACIONES DE RESERVA

Func	Op1	Tag1	Op2	Tag2	dstID

MULT(5c)/DIV(10c)

Buffer de renombramiento con estaciones de reserva

Ciclo i (Emisión de I1)

I1: MULT R3 R1 R2
I2: ADD R4 R2 R3
I3: SUB R3 R5 R2

Banco de Registros

R1	10
R2	22
R3	30
R4	40
R5	50
…	…

BUFFER DE RENOMBRAMIENTO

dstID	A.Val.	RDest	Valor	Valido	Ult
BR1	1	R3	--	0	1
BR2	0				
BR3	0				
BR4	0				
BR5	0				
…	…	…	…	…	…

BUFFER de Reordenación

Instr	etapa	dstID
I1: MULT R3	i	BR1

ESTACIONES DE RESERVA

Func	Op1	Tag1	Op2	Tag2	dstID

ADD/SUB (2c)

ESTACIONES DE RESERVA

Func	Op1	Tag1	Op2	Tag2	dstID
MUL	10	0	22	0	BR1

MULT(5c)/DIV(10c)

Buffer de renombramiento con estaciones de reserva

Ciclo i +1 (Emisión de I2)

I1: MULT R3 R1 R2
I2: ADD R4 R2 R3
I3: SUB R3 R5 R2

Banco de Registros

R1	10
R2	22
R3	30
R4	40
R5	50
…	…

BUFFER DE RENOMBRAMIENTO

dstID	A.Val.	RDest	Valor	Valido	Ult
BR1	1	R3	--	0	1
BR2	1	R4	--	0	1
BR3	0				
BR4	0				
BR5	0				
…	…	…	…	…	…

BUFFER de Reordenación

Instr	etapa	dstID
I1: MULT R3..	x (1)	BR1
I2: ADD R4..	i	BR2

ESTACIONES DE RESERVA

Func	Op1	Tag1	Op2	Tag2	dstID
ADD	22	0	--	BR1	BR2

ADD/SUB (2c)

ESTACIONES DE RESERVA

Func	Op1	Tag1	Op2	Tag2	dstID
(1c)Mul	10	0	22	0	BR1

MULT(5c)/DIV(10c)

Buffer de renombramiento con estaciones de reserva

Ciclo i +2 (Emisión de I3)

I1: MULT R3 R1 R2
I2: ADD R4 R2 R3
I3: SUB R3 R5 R2

Banco de Registros

R1	10
R2	22
R3	30
R4	40
R5	50
…	…

BUFFER DE RENOMBRAMIENTO

dstID	A.Val.	RDest	Valor	Valido	Ult
BR1	1	R3	--	0	0
BR2	1	R4	--	0	1
BR3	1	R3	--	0	1
BR4	0				
BR5	0				
…	…	…	…	…	…

BUFFER de Reordenación

Instr	etapa	dstID
I1: MULT R3..	x (2)	BR1
I2: ADD R4..	i	BR2
I3: SUB R3..	i	BR3

ESTACIONES DE RESERVA

Func	Op1	Tag1	Op2	Tag2	dstID
ADD	22	0	--	BR1	BR2
SUB	50	0	22	0	BR3

ADD/SUB (2c)

ESTACIONES DE RESERVA

Func	Op1	Tag1	Op2	Tag2	dstID
(2c)Mul	10	0	22	0	BR1

MULT(5c)/DIV(10c)

Buffer de renombramiento con estaciones de reserva

Ciclo i +3 (Empieza ejecución de I3 antes que I2)

I1: MULT R3 R1 R2
I2: ADD R4 R2 R3
I3: SUB R3 R5 R2

Banco de Registros

R1	10
R2	22
R3	30
R4	40
R5	50
…	…

BUFFER DE RENOMBRAMIENTO

dstID	A.Val.	RDest	Valor	Valido	Ult
BR1	1	R3	--	0	0
BR2	1	R4	--	0	1
BR3	1	R3	--	0	1
BR4	0				
BR5	0				
…	…	…	…	…	…

BUFFER de Reordenación

Instr	etapa	dstID
I1: MULT R3..	x (3)	BR1
I2: ADD R4..	i	BR2
I3: SUB R3..	x(1)	BR3

ESTACIONES DE RESERVA

Func	Op1	Tag1	Op2	Tag2	dstID
ADD	22	0	--	BR1	BR2
(1c)SUB	50	0	22	0	BR3

ADD/SUB (2c)

ESTACIONES DE RESERVA

Func	Op1	Tag1	Op2	Tag2	dstID
(3c)Mul	10	0	22	0	BR1

MULT(5c)/DIV(10c)

Buffer de renombramiento con estaciones de reserva

Ciclo i +5 (Acaba de finalizar la ejecución de I3)

I1: MULT R3 R1 R2
I2: ADD R4 R2 R3
I3: SUB R3 R5 R2

BUFFER de Reordenación

Instr	etapa	dstID
I1: MULT R3..	x (5)	BR1
I2: ADD R4..	i	BR2
I3: SUB R3..	f	BR3

Banco de Registros

R1	10
R2	22
R3	30
R4	40
R5	50
…	…

BUFFER DE RENOMBRAMIENTO

dstID	A.Val.	RDest	Valor	Valido	Ult
BR1	1	R3	--	0	0
BR2	1	R4	--	0	1
BR3	1	R3	28	1	1
BR4	0				
BR5	0				
…	…	…	…	…	…

ESTACIONES DE RESERVA

Func	Op1	Tag1	Op2	Tag2	dstID
ADD	22	0	--	BR1	BR2
(f)SUB	50	0	22	0	BR3

ADD/SUB (2c)

ESTACIONES DE RESERVA

Func	Op1	Tag1	Op2	Tag2	dstID
(5c)Mul	10	0	22	0	BR1

MULT(5c)/DIV(10c)

Buffer de renombramiento con estaciones de reserva

Ciclo i +6 (Acaba de finalizar la ejecución de I1)

I1: MULT R3 R1 R2
I2: ADD R4 R2 R3
I3: SUB R3 R5 R2

BUFFER de Reordenación

Instr	etapa	dstID
I1: MULT R3..	f	BR1
I2: ADD R4..	i	BR2
I3: SUB R3..	f	BR3

Banco de Registros

R1	10
R2	22
R3	30
R4	40
R5	50
…	…

BUFFER DE RENOMBRAMIENTO

dstID	A.Val.	RDest	Valor	Valido	Ult
BR1	1	R3	220	1	0
BR2	1	R4	--	0	1
BR3	1	R3	28	1	1
BR4	0				
BR5	0				
…	…	…	…	…	…

ESTACIONES DE RESERVA

Func	Op1	Tag1	Op2	Tag2	dstID
ADD	22	0	220	0	BR2

ADD/SUB (2c)

ESTACIONES DE RESERVA

Func	Op1	Tag1	Op2	Tag2	dstID
(f)Mul	10	0	22	0	BR1

MULT(5c)/DIV(10c)

Buffer de renombramiento con estaciones de reserva

Ciclo i +7 (Sale de ROB I1)

I1: MULT R3 R1 R2
I2: ADD R4 R2 R3
I3: SUB R3 R5 R2

Banco de Registros

R1	10
R2	22
R3	220
R4	40
R5	50
…	…

BUFFER de Reordenación

Instr	etapa	dstID
Sale MULT R3..		BR1
I2: ADD R4..	X(1)	BR2
I3: SUB R3..	f	BR3

BUFFER DE RENOMBRAMIENTO

dstID	A.Val.	RDest	Valor	Valido	Ult
BR1	0	R3	220	1	0
BR2	1	R4	--	0	1
BR3	1	R3	28	1	1
BR4	0				
BR5	0				
…	…	…	…	…	…

ESTACIONES DE RESERVA

Func	Op1	Tag1	Op2	Tag2	dstID
(1c)ADD	22	0	220	0	BR2

ADD/SUB (2c)

ESTACIONES DE RESERVA

Func	Op1	Tag1	Op2	Tag2	dstID

MULT(5c)/DIV(10c)

Buffer de renombramiento con estaciones de reserva

Ciclo i +9 (Acaba de finalizar la ejecución de I3)

I1: MULT R3 R1 R2
I2: ADD R4 R2 R3
I3: SUB R3 R5 R2

Banco de Registros

R1	10
R2	22
R3	220
R4	40
R5	50
…	…

BUFFER DE RENOMBRAMIENTO

dstID	A.Val.	RDest	Valor	Valido	Ult
BR1	0				
BR2	1	R4	242	1	1
BR3	1	R3	28	1	1
BR4	0				
BR5	0				
…	…	…	…	…	…

BUFFER de Reordenación

Instr	etapa	dstID
I2: ADD R4..	f	BR2
I3: SUB R3..	f	BR3

ESTACIONES DE RESERVA

Func	Op1	Tag1	Op2	Tag2	dstID
(f)ADD	22	0	220	0	BR2

ADD/SUB (2c)

ESTACIONES DE RESERVA

Func	Op1	Tag1	Op2	Tag2	dstID

MULT(5c)/DIV(10c)

Buffer de renombramiento con estaciones de reserva

Ciclo i +10 (se retiran del ROB I2 e I3 en orden)

I1: MULT R3 R1 R2
I2: ADD R4 R2 R3
I3: SUB R3 R5 R2

BUFFER de Reordenación

Instr	etapa	dstID
Sale I2: ADD R4..		BR2
Sale I3: SUB R3..		BR3

Banco de Registros

R1	10
R2	22
R3	28
R4	242
R5	50
…	…

BUFFER DE RENOMBRAMIENTO

dstID	A.Val.	RDest	Valor	Valido	Ult
BR1	0				
BR2	0	R4	242	1	1
BR3	0	R3	28	1	1
BR4	0				
BR5	0				
…	…	…	…	…	…

ESTACIONES DE RESERVA

Func	Op1	Tag1	Op2	Tag2	dstID

ADD/SUB (2c)

ESTACIONES DE RESERVA

Func	Op1	Tag1	Op2	Tag2	dstID

MULT(5c)/DIV(10c)

Speculation

- “Guess” what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated

Compiler/Hardware Speculation

- Compiler can reorder instructions
  - e.g., move load before branch
  - Can include “fix-up” instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
  - Buffer results until it determines they are actually needed
  - Flush buffers on incorrect speculation

Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?
  - e.g., speculative load before null-pointer check
- Static speculation
  - Can add ISA support for deferring exceptions
- Dynamic speculation
  - Can buffer exceptions until instruction completion (which may not occur)

Interruptions and exceptions

Where does the processor stop?

i: R4 ← R6 op R8
j: R2 ← R0 op R4 ¡Exception!
k: R4 ← R0 op R8
l: R8 ← R4 op R8

If j is running and k has already written R4.

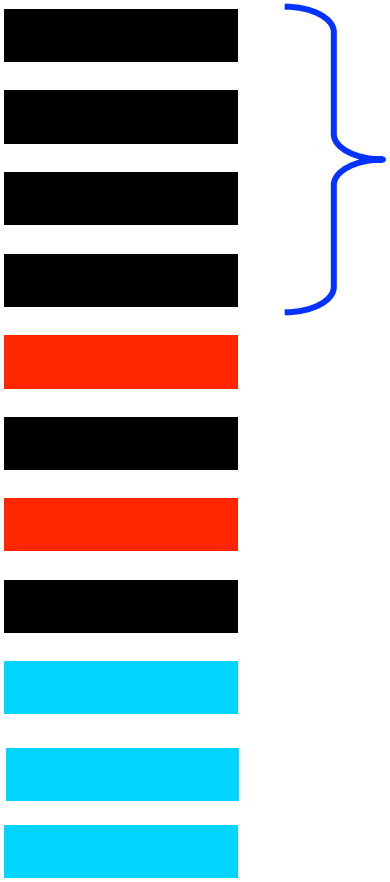
How to recover R4 before the exception?

¡¡PROBLEM!!

=> Buffer to maintain the “in-order”.

ROB: Exception/interruption management

State of the instructions when the interruption/exception occurs.



Have "finished" ("commit"). Results are in the ARF (Arch. Reg. File) or in data memory.
So everything the program had to do before these instructions have been done (and updated) when the interruption/ exception occurs.

Finish
Running
Not issued

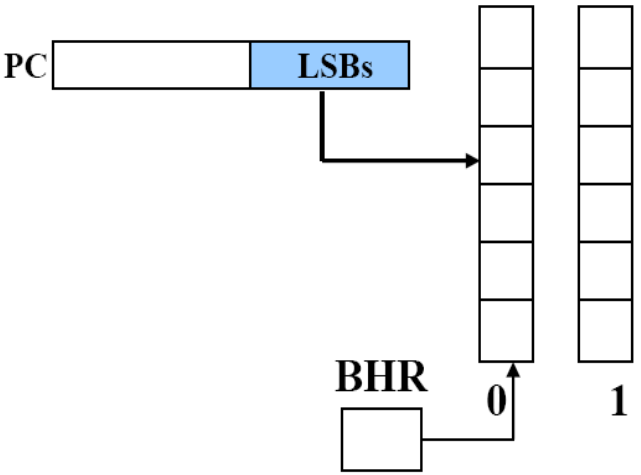
Program Order

Branch and load speculation

- Predict branch and continue issuing
  - Don’t commit until branch outcome determined
- Load speculation
  - Avoid load and cache miss delay
    - Predict the effective address
    - Predict loaded value
    - Load before completing outstanding stores
    - Bypass stored values to load unit
  - Don’t commit load until speculation cleared

Branch prediction: 2 or N levels

What we have seen before is predictor(0,1) y predictor(0,2)



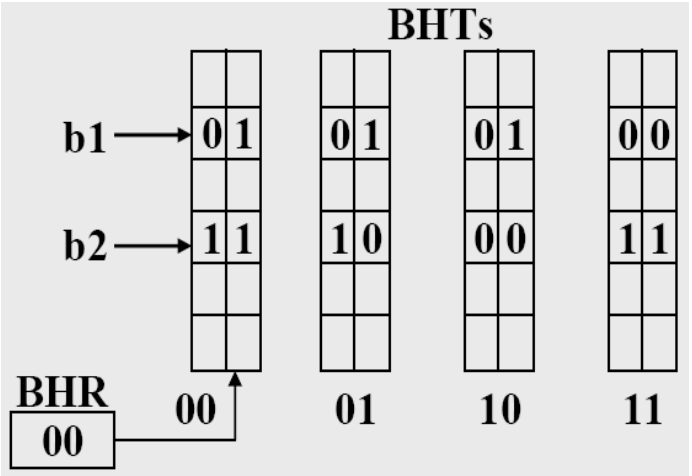
Take into account correlation between branches.

Predictor (1,1):

- Store the prediction of the current branch based on the last one.
- 2 BHTs are needed.

BHR (Branch History Register) contains 1 bit, indicating the state of the last branch (0=NE, 1=E).

Predictor (2,2):



Predictor (m,n):

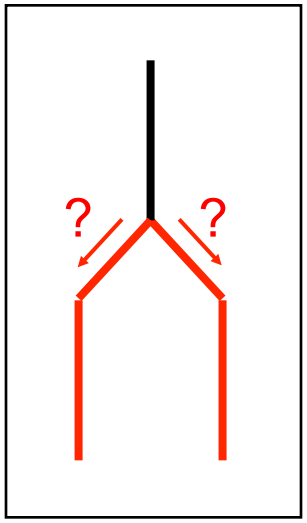
- Uses the information of the m last branches.
- 2^m BHTs of n bits are needed.

BHR (Branch History Register) is a shift register of m bits, and contains the state of the m last branches.

- It is used as index of which BHT used.

Multipath techniques for branches

- **For branches where the prediction is not reliable, both paths are executed.**
  - Pipeline does not stop.
  - Both paths compete to take the resources.
  - Require a more complex Instruction Cache, since we need 2 PCs. Memory bus requires higher bandwidth.

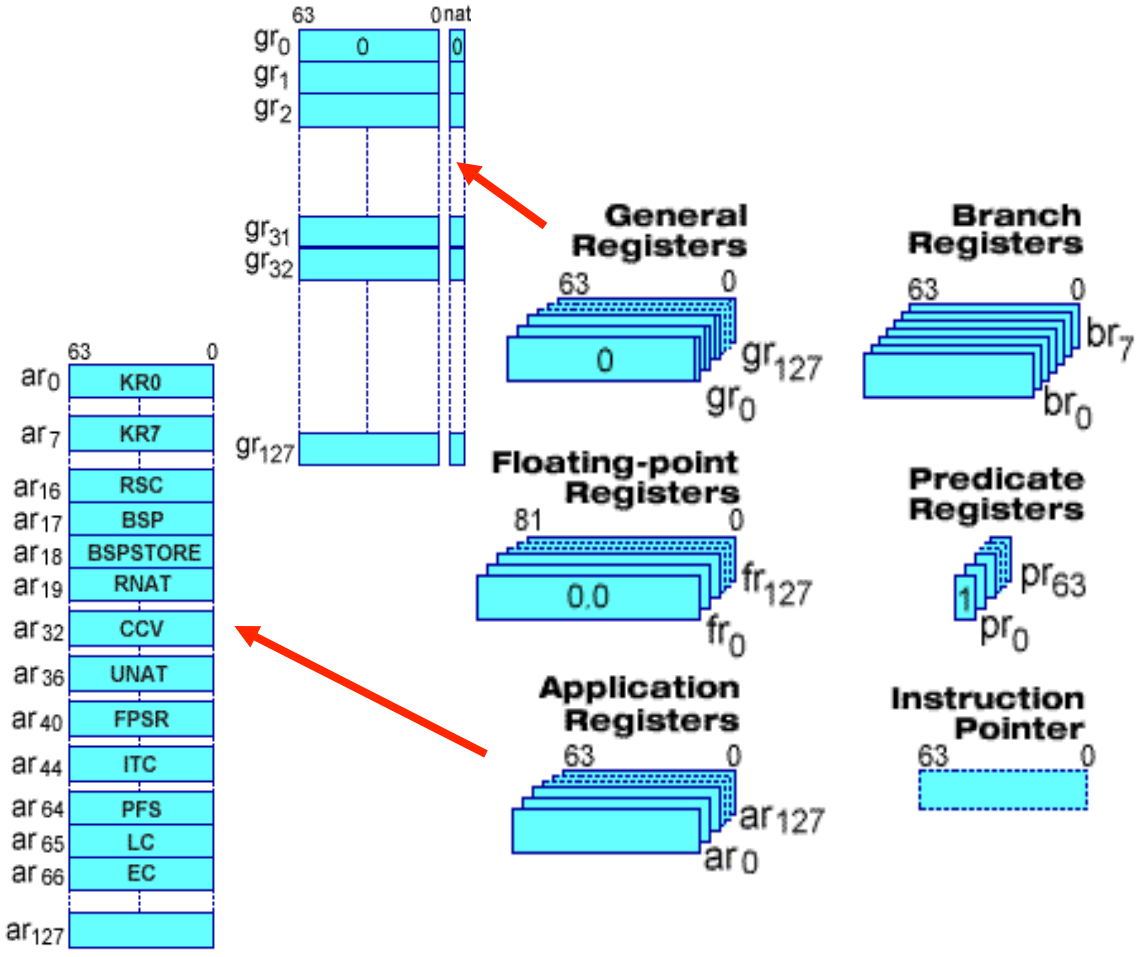


- **Different proposals based on:**
  - how the pipeline stages can be run speculative.
  - how the speculation is controled.
  - instructions fetch wrongly are removed.

Predication

- **General procedure:**
  - Branch solution (1 bit for E or NE) is stored in a predicated register.
  - Branch instructions in speculative paths make reference to these predicated registers.
  - The register or data memory is not written till the predicated is verified (do not *commit* till the predicated is verified).
- **Favourable secundary effects:**
  - Some branch instructions disappear from the code.
  - The blocks before and after the branch one can be combined in a 'hyper-block' for scheduling planification.
- **Problems:**
  - The speculation is not canceled, it is run and if needed, it is ignored.
  - ISA has to provide with instructions with predicates as additional operands.

Intel IA-64: registros de predicados

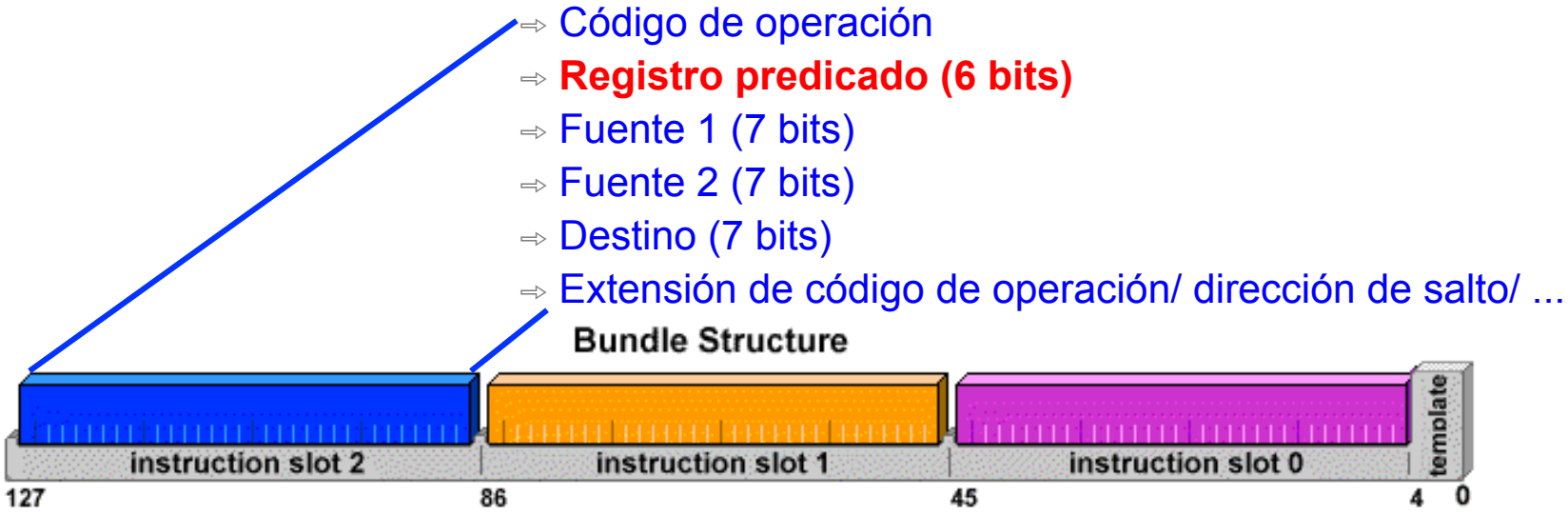


- **Registros IA-64**
  - 128 Registros generales
  - 128 Registros coma flotante
  - 64 Registros de Predicados
  - 8 Registros de Saltos
  - 128 Registros de aplicación
  - Contador de Instrucción (IP)

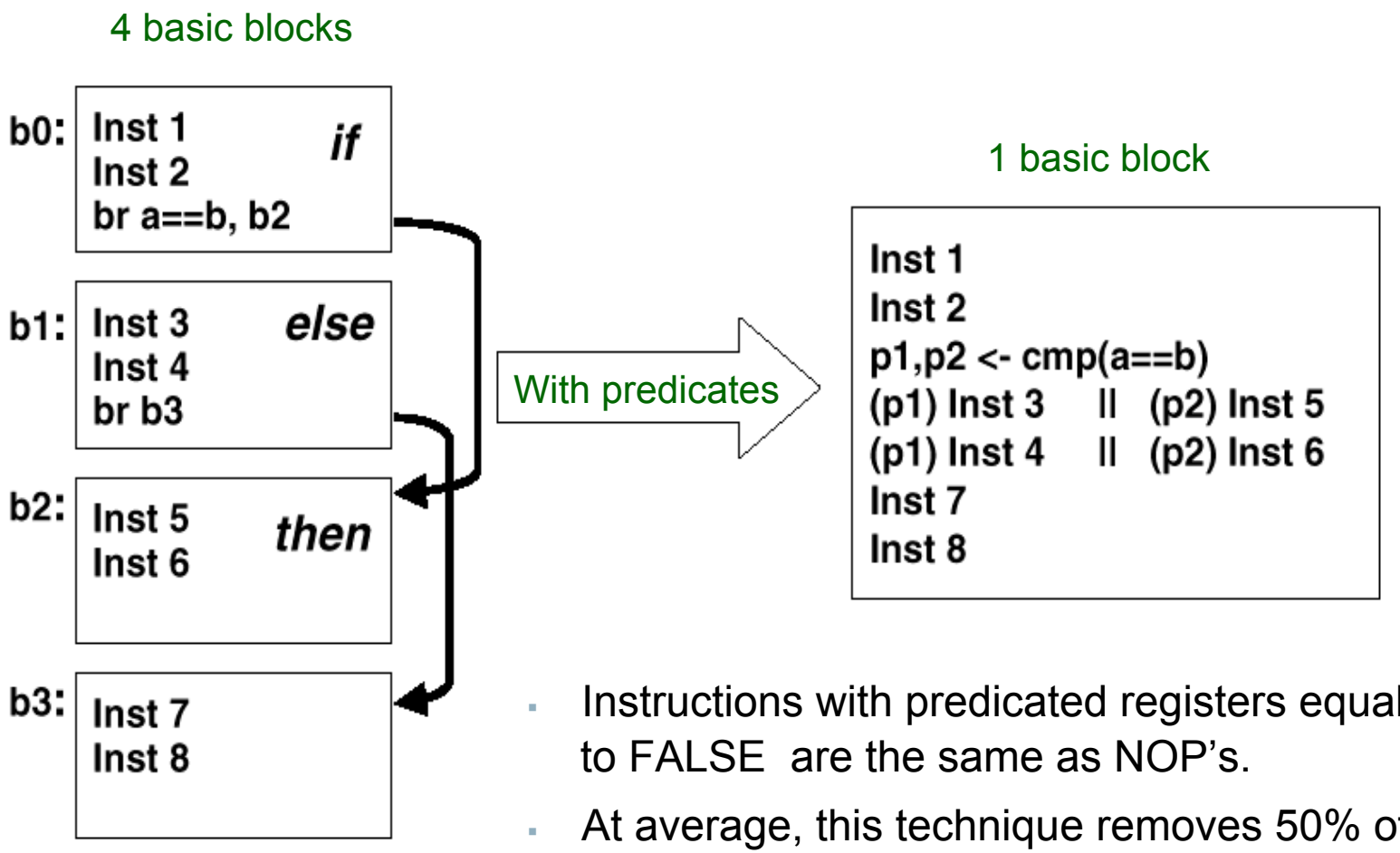
Intel IA-64: registros de predicados

- Soporte en el set de instrucciones

  Instrucciones Largas (bundles= 128 bit)



Predication: Intel IA-64



- Instructions with predicated registers equal to FALSE are the same as NOP's.
- At average, this technique removes 50% of the branches.

Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predictable
  - e.g., cache misses
- Can't always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

Does Multiple Issue Work?

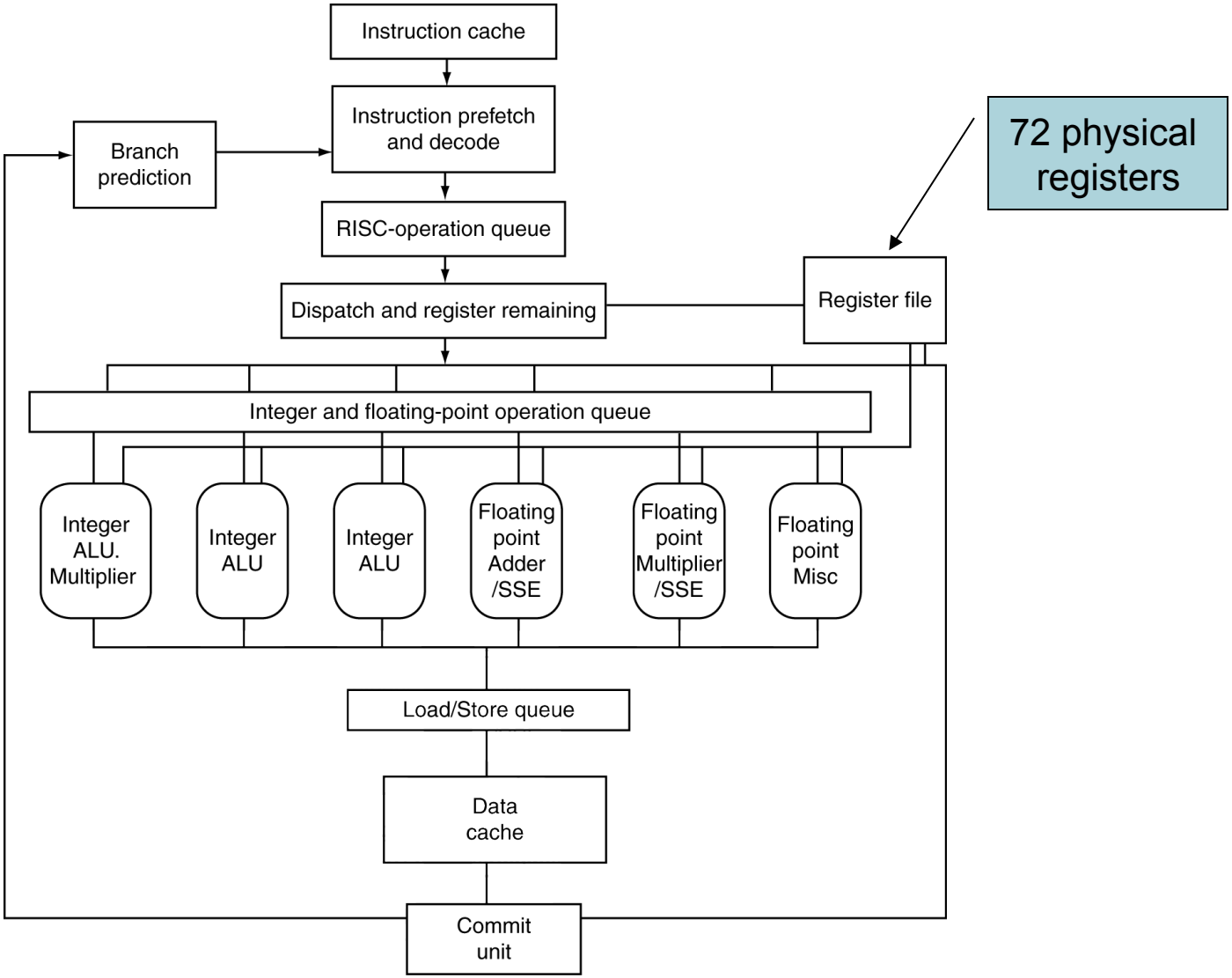
- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

Power Efficiency

- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

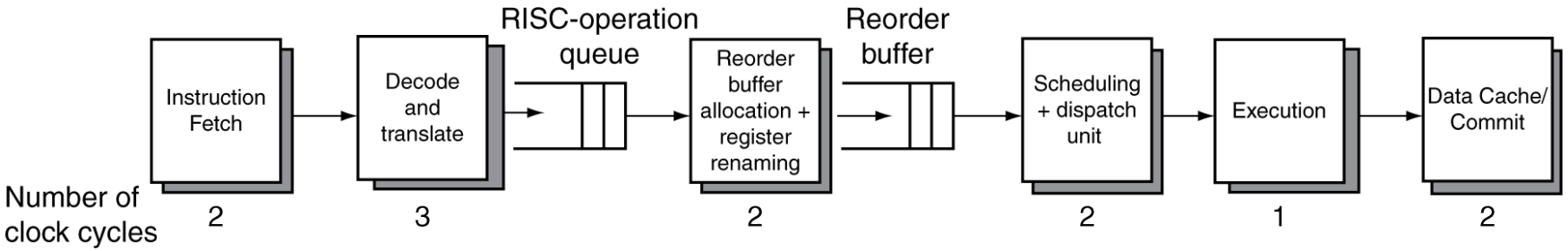
Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

The Opteron X4 Microarchitecture



The Opteron X4 Pipeline Flow

- For integer operations



  - FP is 5 stages longer
  - Up to 106 RISC-ops in progress
- Bottlenecks
  - Complex instructions with long dependencies
  - Branch mispredictions
  - Memory access delays

Procesadores VLIW

Procesadores con formato de instrucción muy larga (VLIW)

- Bases de VLIW y ventajas
- Superscalar vs VLIW
- Limitaciones y como se solucionan

Procesadores VLIW. Bases

La idea no es nueva (década de los 70)

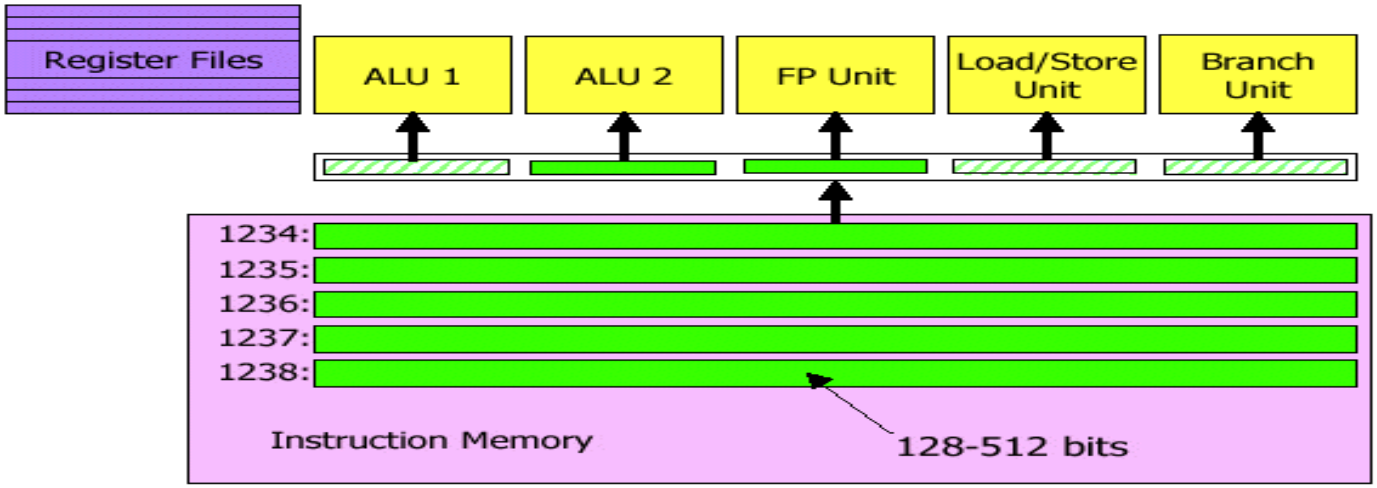
⇒ Necesita una tecnología de compiladores que la soporte

⇒ Una instrucción especifica múltiples operaciones que se agrupa en un formato largo de hasta 128-1024 bits

⇒ Microprogramación

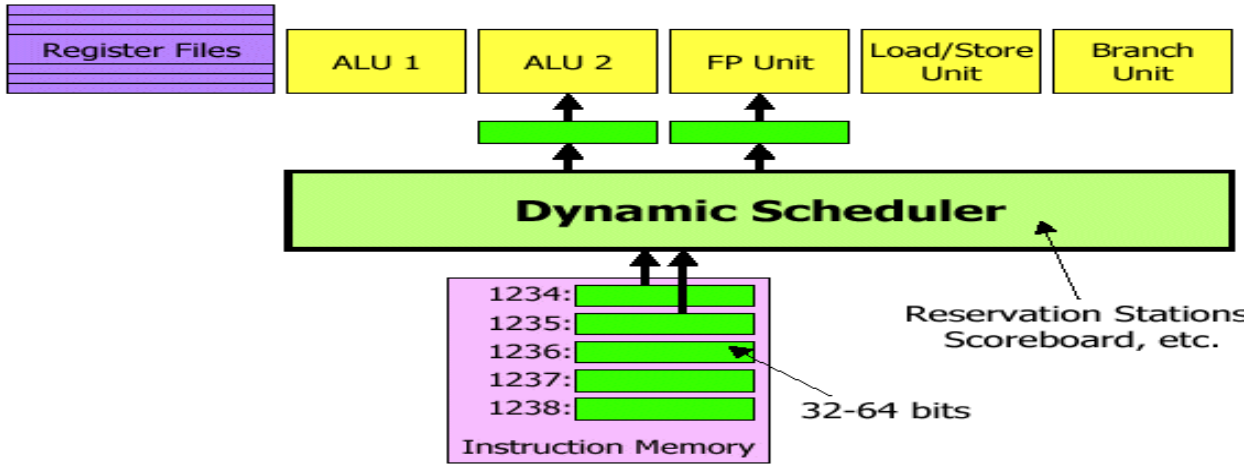
⇒ La planificación de la ejecución de las instrucciones es externa

⇒ El compilador es quien determina la efectividad del procesador

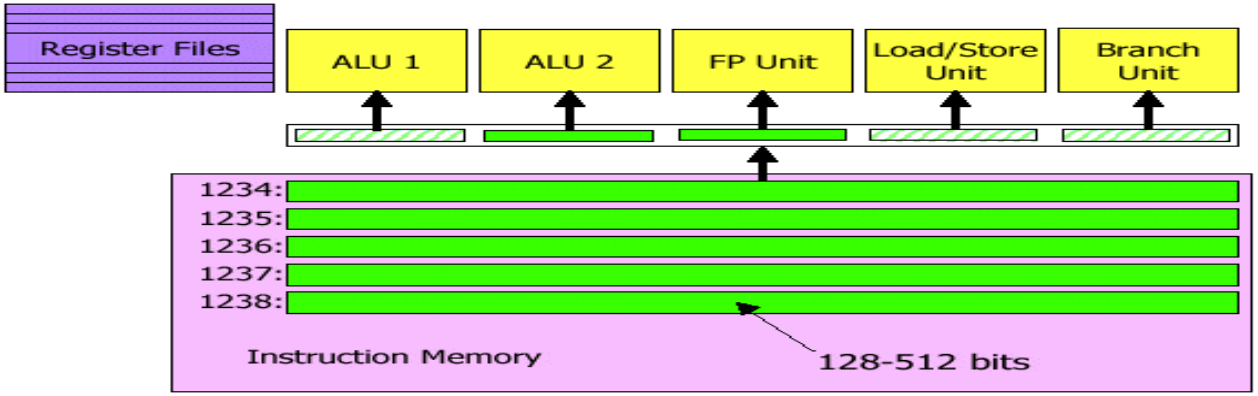


Procesadores VLIW: superescalar vs VLIW

Superescalar



VLIW



Arquitectura VLIW: ventajas

Ventajas:

⇒La planificación de la ejecución estática de instrucciones es realizada por el compilador y supone:

- **⇒ Menos lógica en la unidad de control.**
- **⇒ Mayor frecuencia de reloj.**
- **⇒ Más silicio para unidades de ejecución.**

⇒ Arquitectura popular en aplicaciones empotradas

- **⇒ DSP**
- **⇒ Multimedia**

Arquitectura VLIW: Limitaciones

Limitaciones a resolver en procesadores VLIW:

1.- Tamaño del código:

- La inclusión de NOP desperdicia espacio en memoria.

2.- Mantener la compatibilidad de código objeto:

- Es necesario recompilar todo el código para cada diseño nuevo de procesador

3.- Ocupar todas las Unidades de Ejecución:

- Todas acceden al mismo banco de registros

4.- Conseguir una planificación eficiente por parte del compilador: accesos a memoria y predicción de saltos:

- Dificultad de predicción estática de los tiempos de latencia en accesos a memoria o caches.
- La planificación óptima de instrucciones depende de la trayectoria de los saltos.

Procesadores VLIW. Código fuente: planificación estática

Código Tradicional

1000:	op 1
1001:	op 2
1002:	op 3
1003:	op 4
1004:	op 5
1005:	op 6
1006:	op 7
1007:	op 8
1008:	op 9

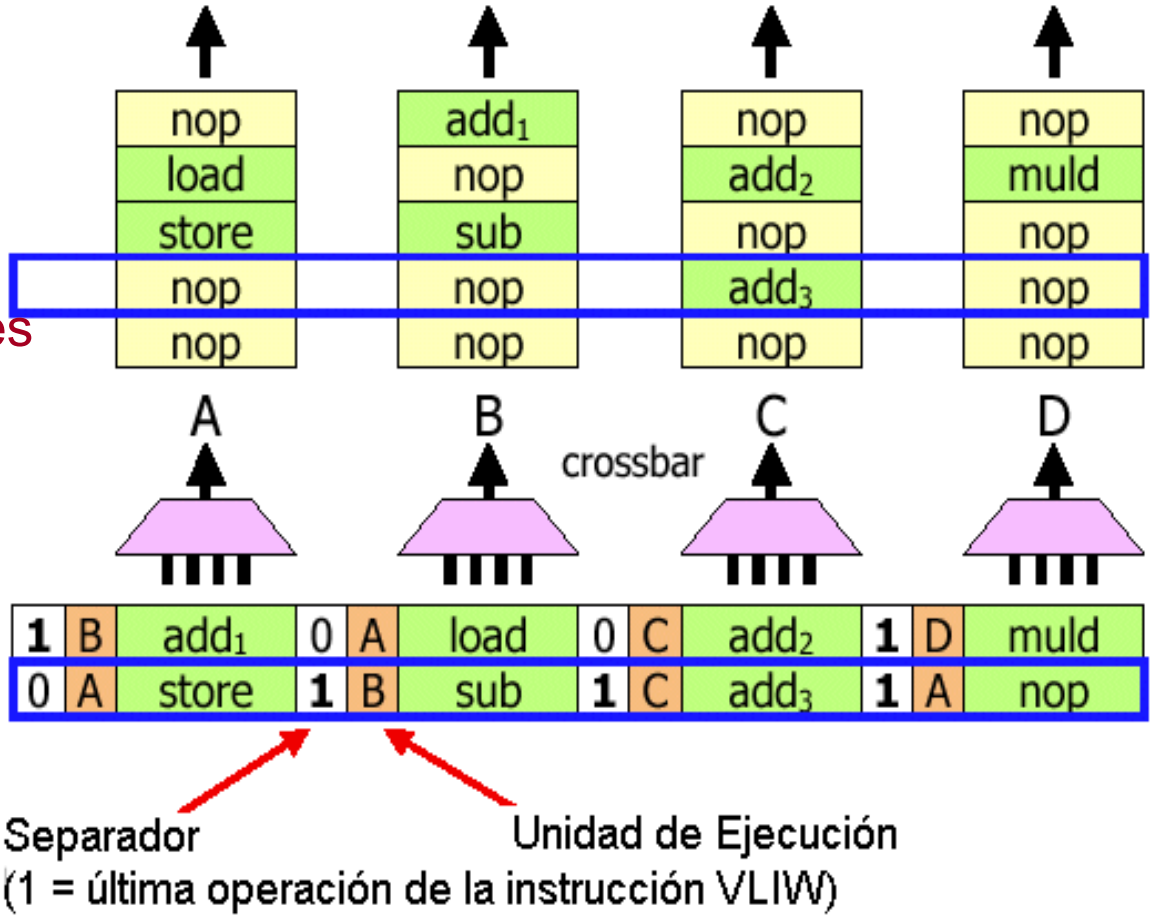
Código VLIW

1000:	op 1	op 6	op 7	NOP
1001:	NOP	NOP	op 3	op 4
1002:	NOP	op 2	NOP	NOP
1003:	NOP	op 5	op 12	NOP
1004:	NOP	NOP	NOP	op 17
1005:	NOP	NOP	op 8	op 16
1006:	op 13	op 14	op 9	NOP
1007:	NOP	NOP	op 10	NOP
1008:	NOP	NOP	op 11	op 15

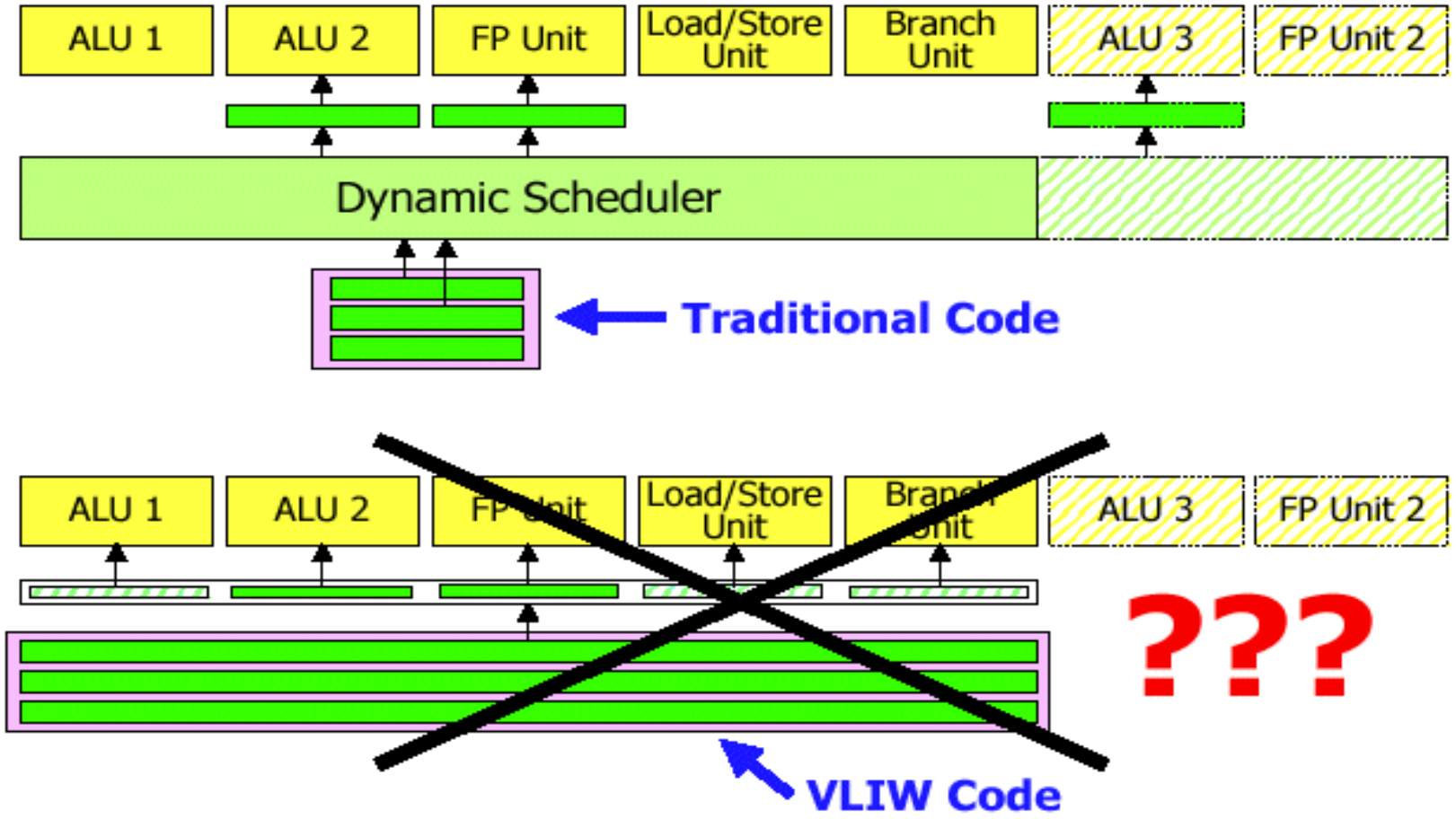
VLIW: Tamaño del código

Formato VLIW clásico. Para ser ejecutado en las unidades Funcionales

Formato empaquetado. Para ser almacenado en memoria.

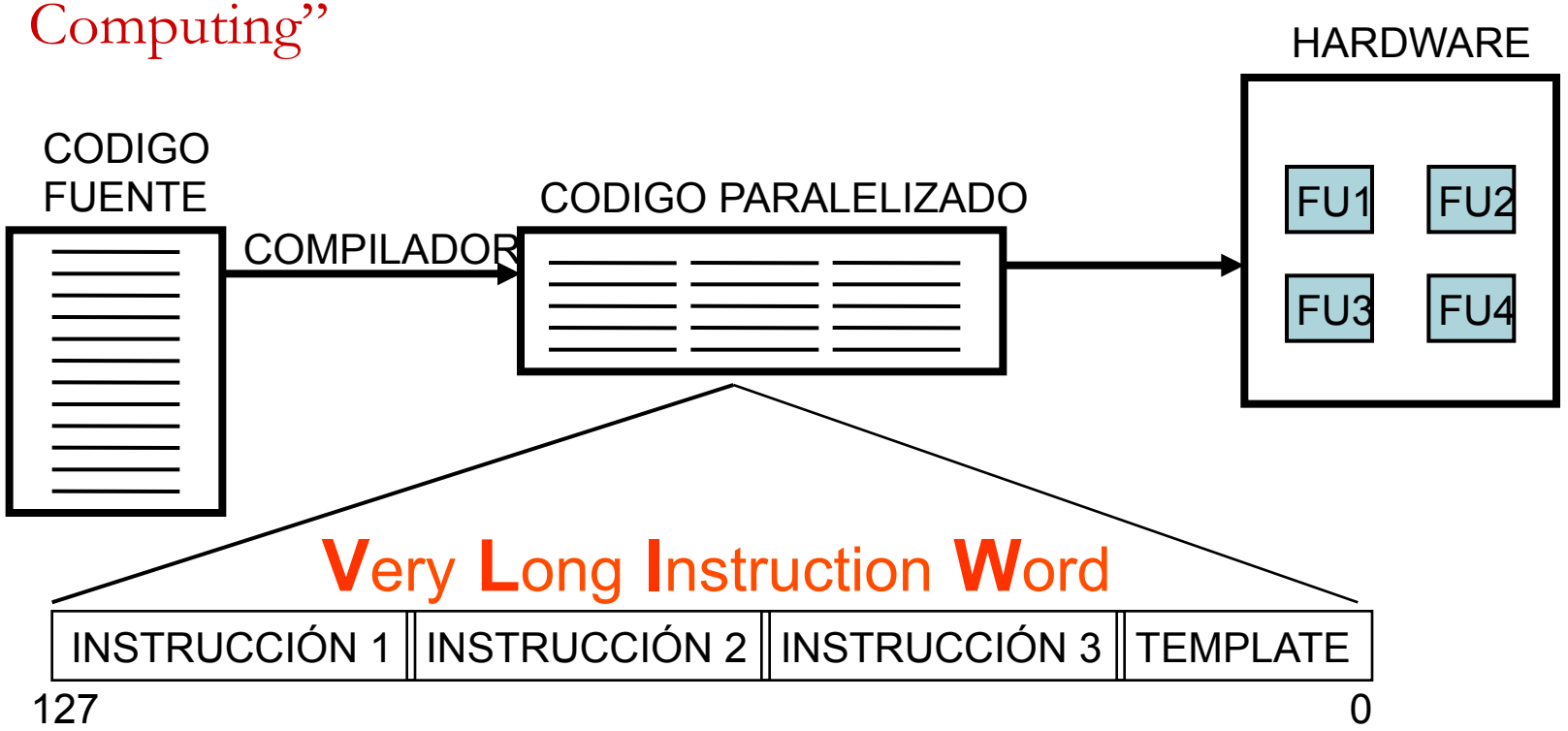


Limitaciones VLIW: compatibilidad de código



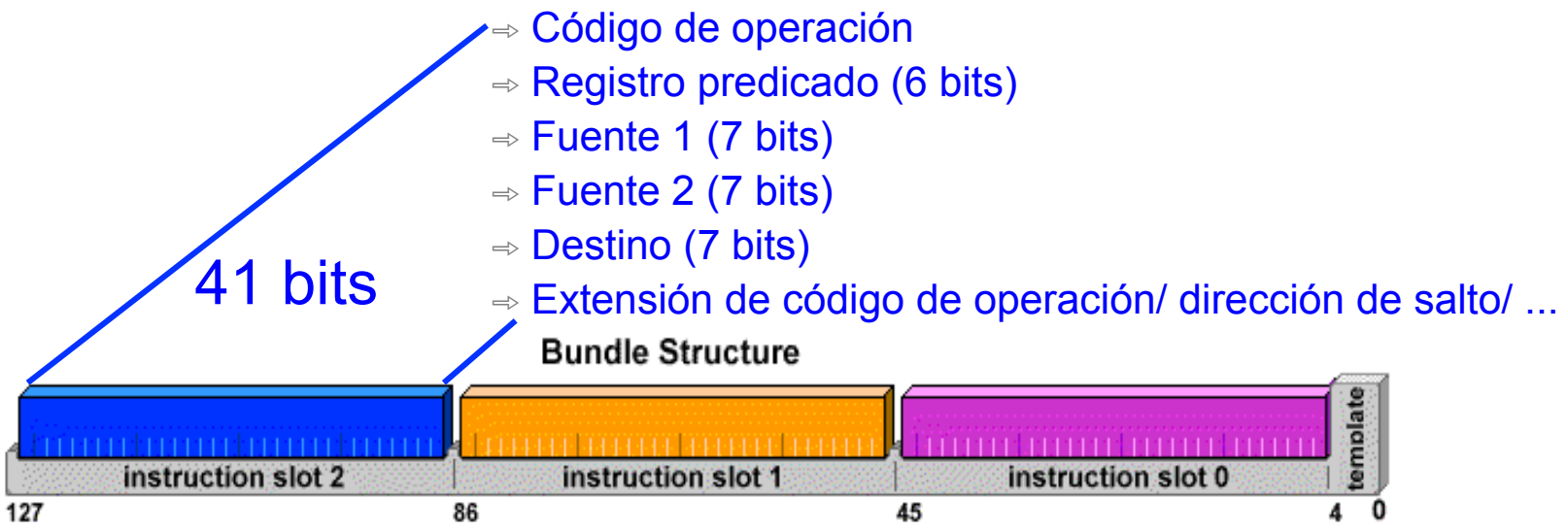
Arquitectura de Intel IA-64

- **Arquitectura EPIC:** "Explicitly Parallel Instruction Computing"



Arquitectura de Intel IA-64.

- Instrucciones Largas (bundles= 128 bit)



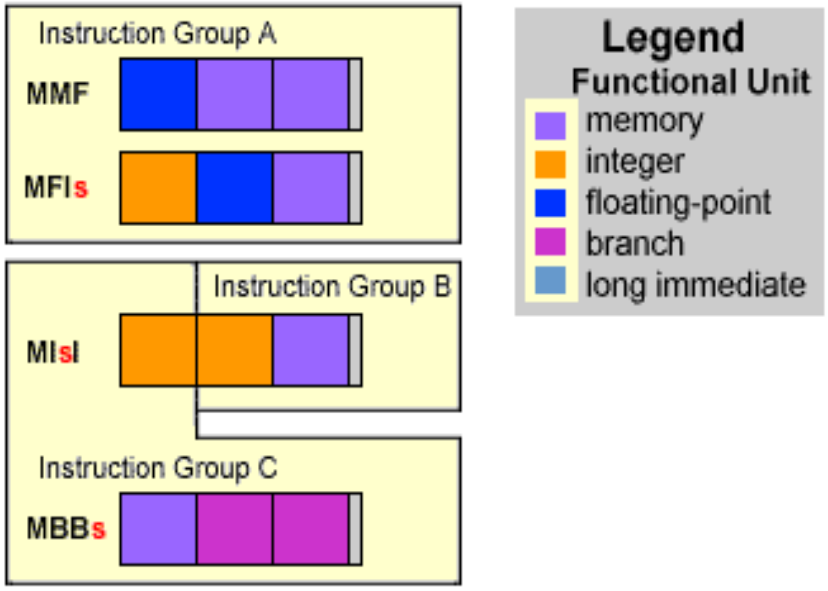
"Bundle" : Es un conjunto de 3 instrucciones y un template.

El campo de template (5 bits)

- indica en que unidad funcional se ejecuta cada operación
- la dependencia entre esta instrucción larga (bundle) y las siguientes.

Arquitectura de Intel IA-64.

- Ejemplo de Grupos de Instrucciones



Grupo de Instrucciones :.

⇨ Un grupo puede terminar en mitad de un “Bundle”.

⇨ Se muestran tres grupos: A y C acaban al final del Bundle y B termina en medio del Bundle

⇨ Los grupos de instrucciones se pueden indicar en el código fuente.

IA-64 Templates

Table 3-9. Template Field Encoding and Instruction Slot Mapping

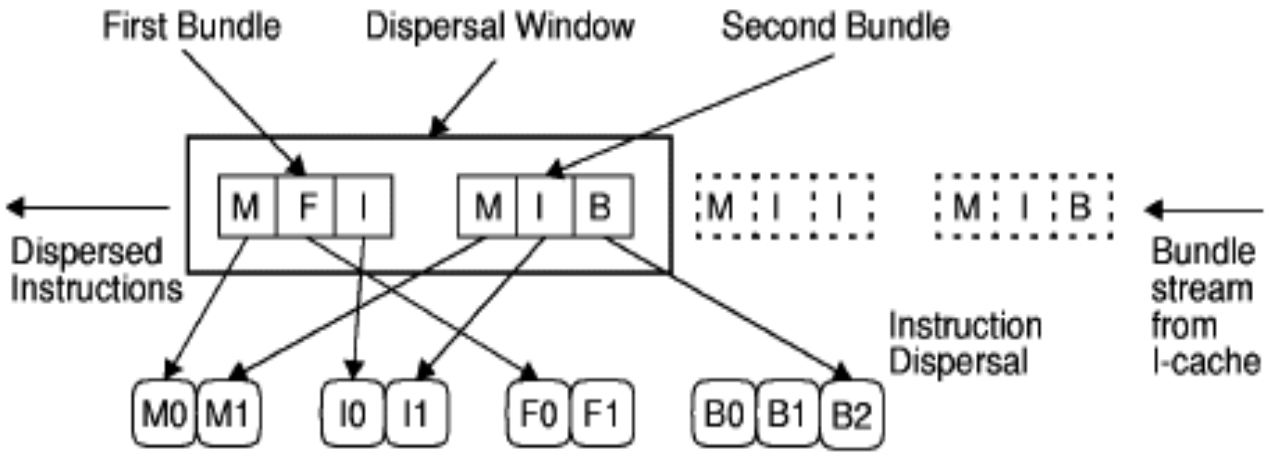
Template	Slot 0	Slot 1	Slot 2
00	M-unit	I-unit	I-unit
01	M-unit	I-unit	I-unit
02	M-unit	I-unit	I-unit
03	M-unit	I-unit	I-unit
04	M-unit	L-unit	X-unit
05	M-unit	L-unit	X-unit
06			
07			
08	M-unit	M-unit	I-unit
09	M-unit	M-unit	I-unit
0A	M-unit	M-unit	I-unit
0B	M-unit	M-unit	I-unit
0C	M-unit	F-unit	I-unit
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
14			
15			
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
1A			
1B			
1C	M-unit	F-unit	B-unit
1D	M-unit	F-unit	B-unit
1E			
1F			

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

Source: Intel/HP IA-64 Application ISA Guide 1.0

Arquitectura de Intel IA-64

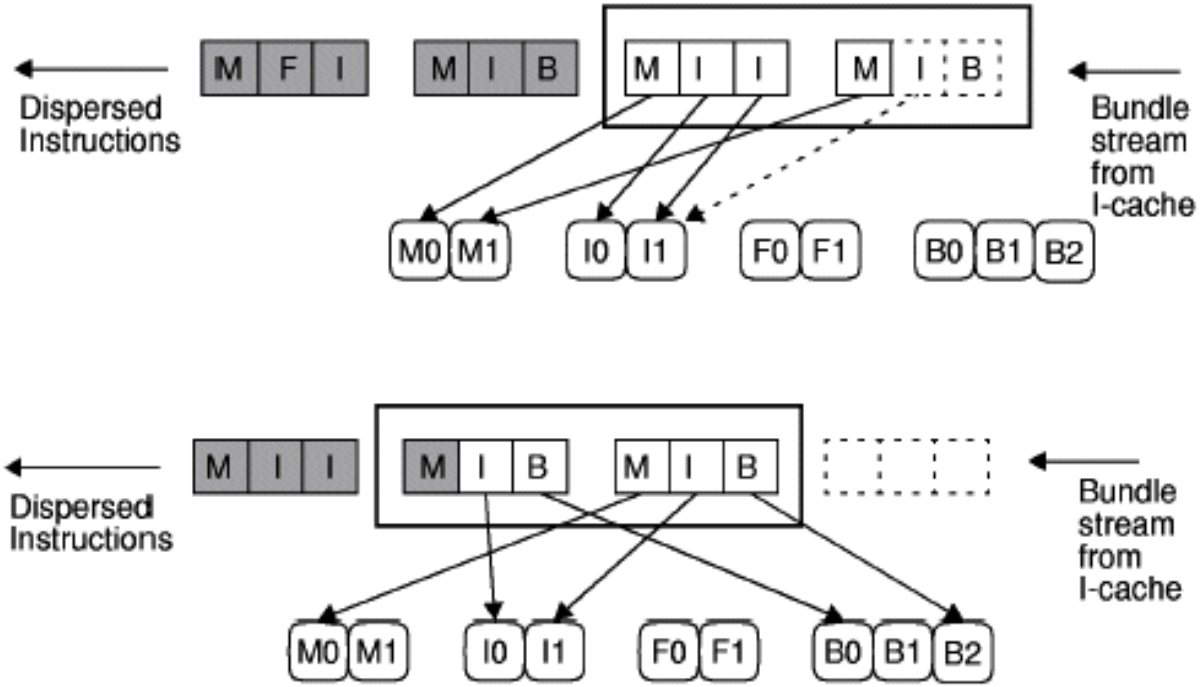
Ejecución de Instrucciones



- Arquitectura Hardware
  - UNIDADES FUNCIONALES
    - 2 Unidades de Memoria
    - 2 Unidades de Enteros
    - 2 Unidades de Coma Flotante
    - 3 Unidades de Tratamientos de Saltos
  - TAMAÑO DE VENTANA: 2 Instrucciones

Arquitectura de Intel IA-64.

- Ejecución de Instrucciones



Source: "Itanium™ Processor Microarchitecture Reference", Intel, August 2000

Limitaciones VLIW y soluciones: planificación eficiente realizada por el compilador

Optimización por el compilador: análisis global de código

⇒ Modificación del código fuente para incrementar el paralelismo a nivel de instrucción.

⇒ Son aplicables las técnicas que mejoran ILP y la eficiencia en el código para procesadores superescalares:

⇒ Desenrollamiento de bucles (replicación de código).

⇒ Actualización de referencias.

⇒ Renombramiento de registros.

Ejemplo: Bucle que suma una constante a los elementos de un vector

```
for (i=10; i>0 ; i = i –1)
 x[i]= x[i] + s;
```

Limitaciones VLIW y soluciones: planificación eficiente realizada por el compilador

Ejemplo: Bucle que suma una constante a los elementos de un vector

```
for (i=10; i>0 ; i = i –1)
  x[i]= x[i] + s;
```

Codificación

```
                                    // vector en posiciones de memoria 0-80
Loop :  ld     $f0, ($r1)           // cargar el elemento del vector ($r1=80)
        addd   $f4, $f0,$f2         // suma la constante, (f2 = s)
        sd     ($r1), $f4           // almacena elemento del vector en la misma posición

        subi $r1, $r1, 8            // apunta al siguiente elemento (double word);
        bnez $r1, Loop
```

Procesador VLIW con tres Unidades de Ejecución:

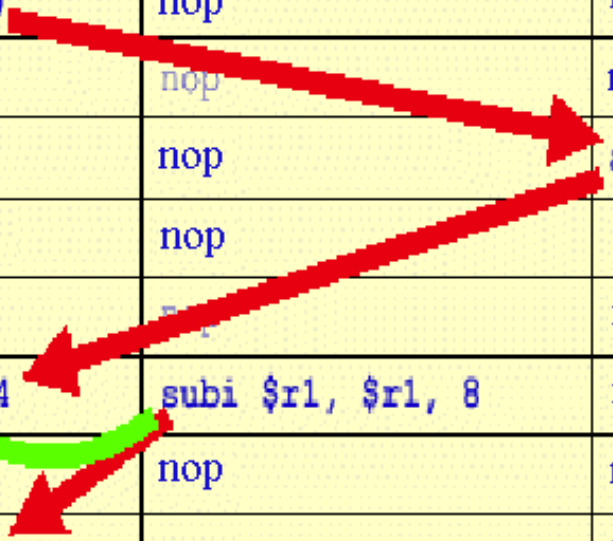
Unidad LD/ST y Saltos	Unidad de Enteros (ALU)	Unidad de Coma Flotante

Latencias: Load/Store = 2 ciclos, Enteros = 2 ciclos, Saltos = 2 ciclos y coma flotante = 3 ciclos
Unidades segmentadas. En cada ciclo puede empezar una nueva instrucción.

Limitaciones VLIW y soluciones: planificación eficiente realizada por el compilador

Ejemplo: Codificación VLIW

Unidad LD/ST y Saltos	Unidad de Enteros (ALU)	Unidad de Coma Flotante	Ciclo
ld $f0, ($r1)	nop	nop	1
nop	nop	nop	2
nop	nop	addd $f4, $f0, $f2	3
nop	nop	nop	4
nop		nop	5
sd ($r1), $f4	subi $r1, $r1, 8	nop	6
nop	nop	nop	7
bnez $r1, Loop	nop	nop	8
nop	nop	nop	9



Tiempo de ejecución: 9 ciclos/iteración * 10 iteraciones = 90 ciclos

Limitaciones VLIW y soluciones: planificación eficiente realizada por el compilador

Ejemplo: Codificación VLIW

```
Loop:  ld     $f0 , ($r1)
       addd   $f4 , $f0 , $f2
       sd     ($r1), $f4

       subi   $r1, $r1, 8
       bnez   $r1, Loop
```

```
Loop:  ld     $f0 , ($r1)
       addd   $f4 , $f0 , $f2
       sd     ($r1), $f4

       ld     $f6, ($r1-8)
       addd   $f8, $f6, $f2
       sd     ($r1-8), $f8

       ld     $f10, ($r1-16)
       addd   $f12, $f10, $f2
       sd     ($r1-16), $f12

       ld     $f14, ($r1-24)
       addd   $f16, $f14, $f2
       sd     ($r1-24), $f16

       ld     $f18, ($r1-32)
       addd   $f20, $f18, $f2
       sd     ($r1-32), $f20

       subi   $r1, $r1, 40
       bnez   $r1, Loop
```

1.- Replicación de Código

2.- Actualización de referencias

3.- Renombramiento de registros.

Desenrollar el bucle: en vez de 10 iteraciones de una operación, 2 iteraciones de 5 operaciones en este ejemplo.

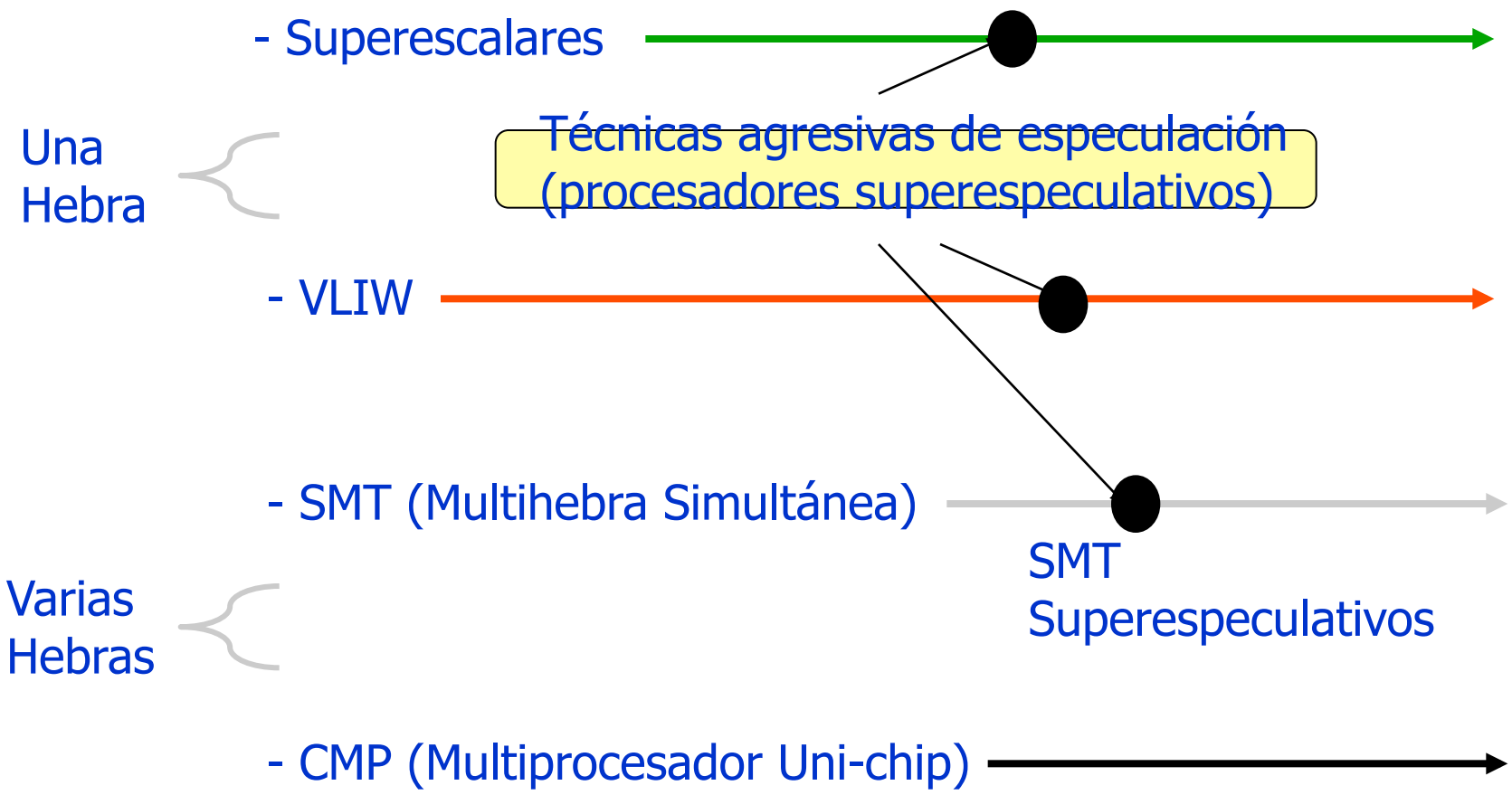
Limitaciones VLIW y soluciones: planificación eficiente realizada por el compilador

Ejemplo: Codificación VLIW optimizada

Unidad LD/ST y Saltos	Unidad de Enteros (ALU)	Unidad de Coma Flotante	Ciclo
ld $f0, ($r1)	nop	nop	1
ld $f6, ($r1-8)	nop	nop	2
ld $f10, ($r1-16)	nop	addd $f4, $f0, $f2	3
ld $f14, ($r1-24)	nop	addd $f8, $f6, $f2	4
ld $f18, ($r1-32)	[illegible]	addd $f12, $f10, $f2	5
sd ($r1), $f4	nop	addd $f16, $f14, $f2	6
sd ($r1-8), $f8	nop	addd $f20, $f18, $f2	7
sd ($r1-16), $f12	nop	nop	8
sd ($r1-24), $f16	nop	nop	9
sd ($r1-32), $f20	subi $r1, $r1, 40	nop	10
nop	nop	nop	11
bnez $r1, Loop	nop	nop	12
nop	nop	nop	13

Tiempo de ejecución: 13 ciclos/iteración * 2 iteraciones = 26 ciclos

ILP una hebra vs TLP varias Hebras



Multiproceso

- El multiproceso por multiplexación en el tiempo tiene su origen en 1962
- La ejecución concurrente de un solo proceso por multiplexación en el tiempo mejora el rendimiento.
  - Un proceso o un thread mantiene el procesador en estado hasta que se completa una operación de I/O, por ejemplo acceso a disco, entrada desde teclado,…
  - Pueden perderse sin realizar trabajo útil de miles a millones de ciclos en la ejecución de un proceso

  - Un proceso o un thread debería pasar a estado de espera en las demoras debidas a operaciones de I/O y dejar a otros threads el uso del procesador, realizando un *cambio de contexto*

Cambio de contexto

- Un "contexto" incluye el estado de los recursos del procesador asociados con un proceso particular
  - *Estado de los recursos visibles al programador:* contador de programa, contenido del banco de registros, contenidos de memoria
  - *Estado de recursos invisibles:* registros de estado y control, puntero base de la tabla de páginas, tabla de páginas, ...
  - Además hay que tomar una decisión sobre:

    Contenidos cache (virtual o real), entradas del BTB y del TLB

- Procedimiento de un cambio de contexto clásico
  - Una interrupción del timer para un programa a mitad de ejecución (interrupción precisa)
  - El sistema operativo (SO) guarda el contexto del thread parado.
  - El SO recupera el contexto de un thread parado con anterioridad (todo excepto el PC)
  - El SO con un "retorno de excepción" salta al PC que reinicia el thread.

    El thread recuperado no se entera de que fue interrumpido, eliminado del procesador y posteriormente reiniciado.

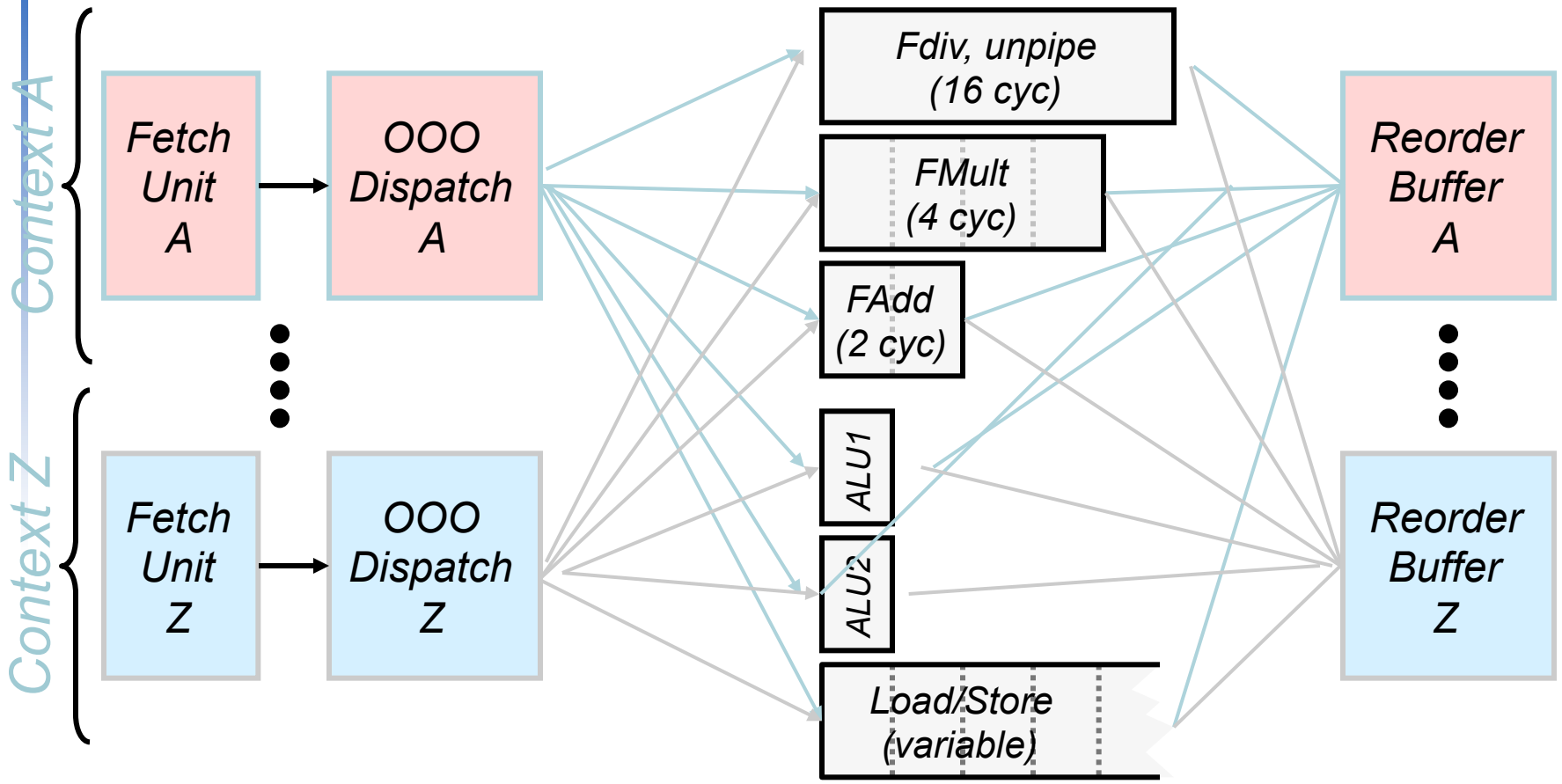
Cambio de contexto rápido

- Un procesador entra en espera (idle) cuando un thread entra en espera por un fallo de cache.
- *¿Es eficiente cambiar a otro thread?*
- Un fallo en cache supone unso diez ciclos, y el coste de un cambio de contexto por el SO es como mínimo de 64 ciclos para guardar y recuperar los registros (32 GPRs)
- Una solución: Realizar el cambio de contexto en hardware
  - Replicar en hardware registros de contexto : PC, GPRs, cntrl/status, Tabla de Pág. base ptr *se elimina el copiado*
  - Se permite que múltiples contextos compartan algunos recursos, incluyendo un "process ID" en la etiquetas de cache, BTB y TLB. *Elimina los arranques en frío.*
  - Un cambio de contexto hardware conlleva sólo unos pocos ciclos:
    - Poner el registro de PID al siguiente identificador de "process ID"
    - Seleccionar el conjunto de registros de contexto hardware a activar.

Cambio de contexto muy rápido

- Cuando un procesador segmentado se para por dependencias RAW entre instrucciones, la etapa de ejecución está en "espera"
- *¿Es posible cambiar a otro thread?*
- No sólo es necesario registros de contexto hardware, es imprescindible que el cambio entre contextos sea ventajoso.
- Si se consigue las ventajas son:
  - No se necesita lógica complicad de adelantamiento de resultados (forwarding) para evitar paradas.
  - Las dependencias RAW y las operaciones de alta latencia (multiplicación y fallos de cache) no causarán perdida de rendimiento.
- *Objetivo: Multithreading es una técnica para "ocultar latencias".*

Simultaneous Multi-Threading *[Eggers, et al.]*



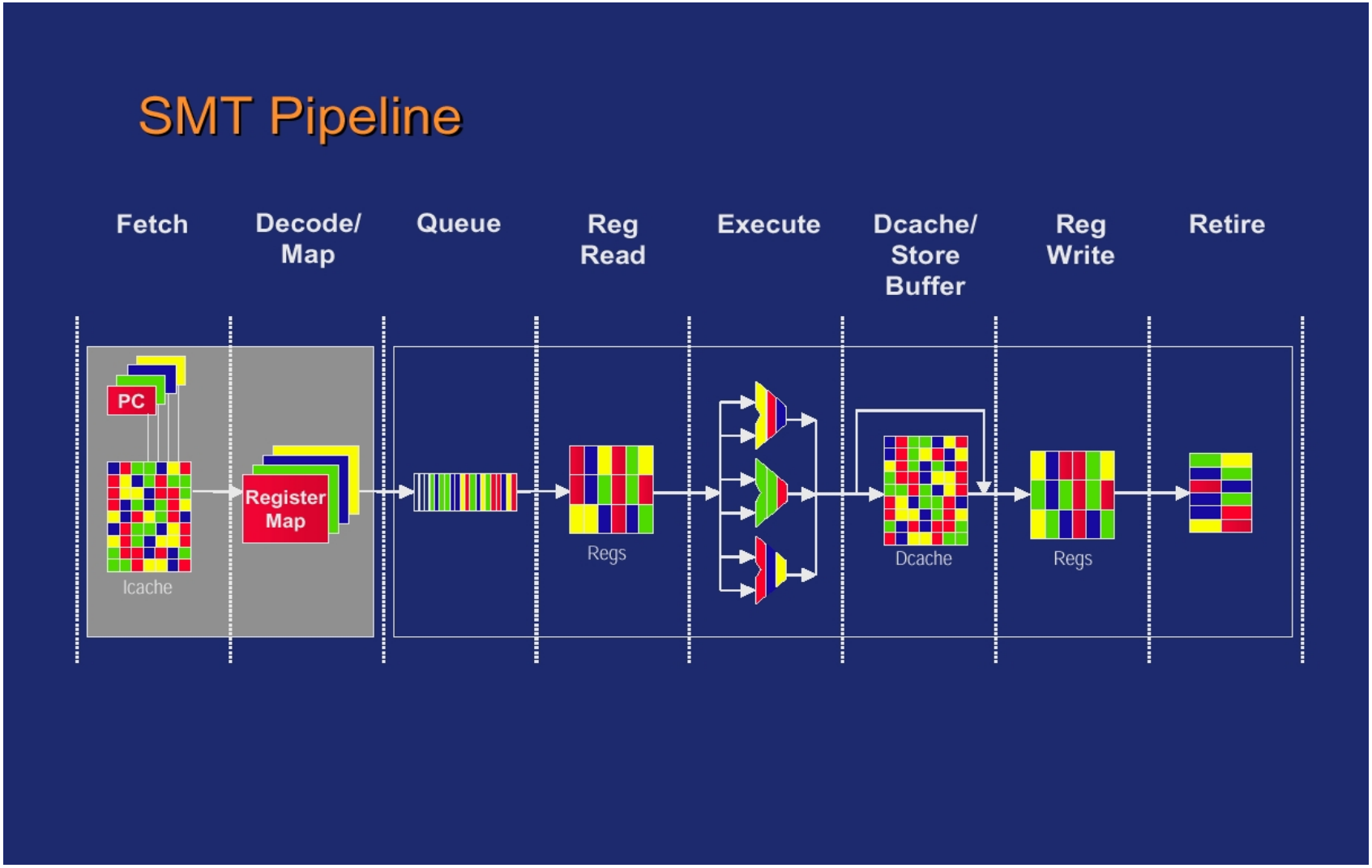
- Se comparten dinámicamente unidades funcionales entre múltiple threads

⇒ *mayor utilización* ⇒ *mayor rendimiento*

SMT Architecture

- Straightforward extension to conventional superscalar design.
  - multiple program counters and some mechanism by which the fetch unit selects one each cycle,
  - a separate return stack for each thread for predicting subroutine return destinations,
  - per-thread instruction retirement, instruction queue flush, and trap mechanisms,
  - a thread id with each branch target buffer entry to avoid predicting phantom branches, and
  - a larger register file, to support logical registers for all threads plus additional registers for register renaming.
    - The size of the register file affects the pipeline and the scheduling of load-dependent instructions.

SMT Microarchitecture [Emer,'01]

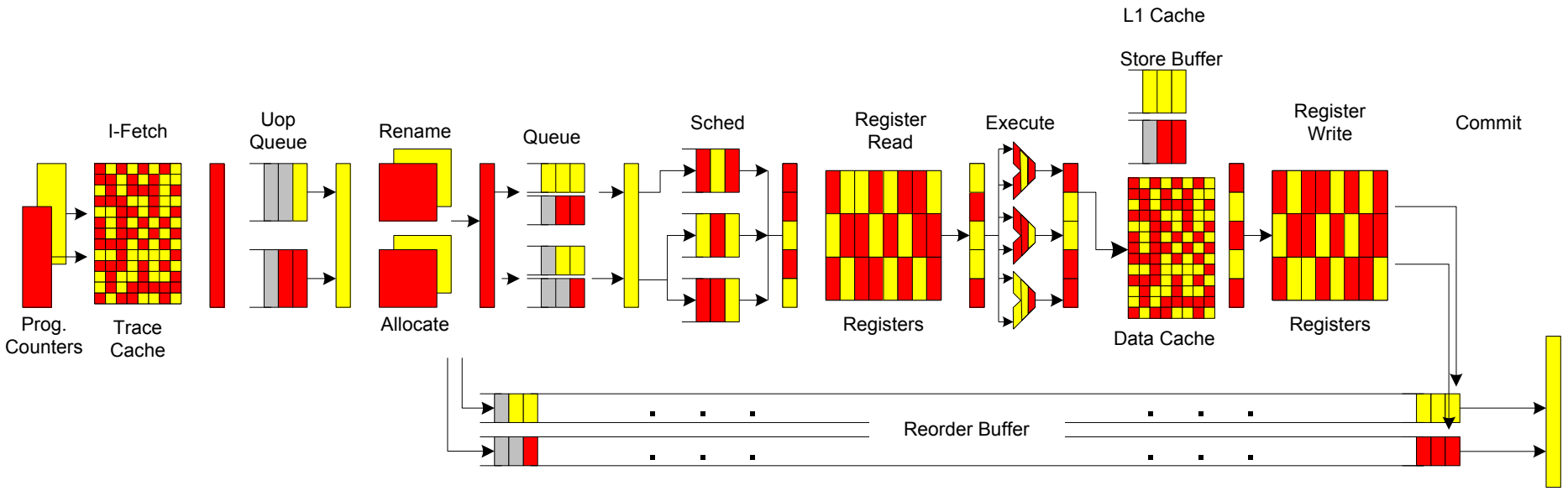


Intel Hyperthreading

- Part of Pentium 4 design (Xeon)
- Two threads per processor
- Goals
  - Low cost – less than 5% overhead for replicated state
  - Assure forward progress of both threads
    - Make sure both threads get some buffer resources
    - through partitioning or budgeting
  - Single thread running alone does not suffer slowdown

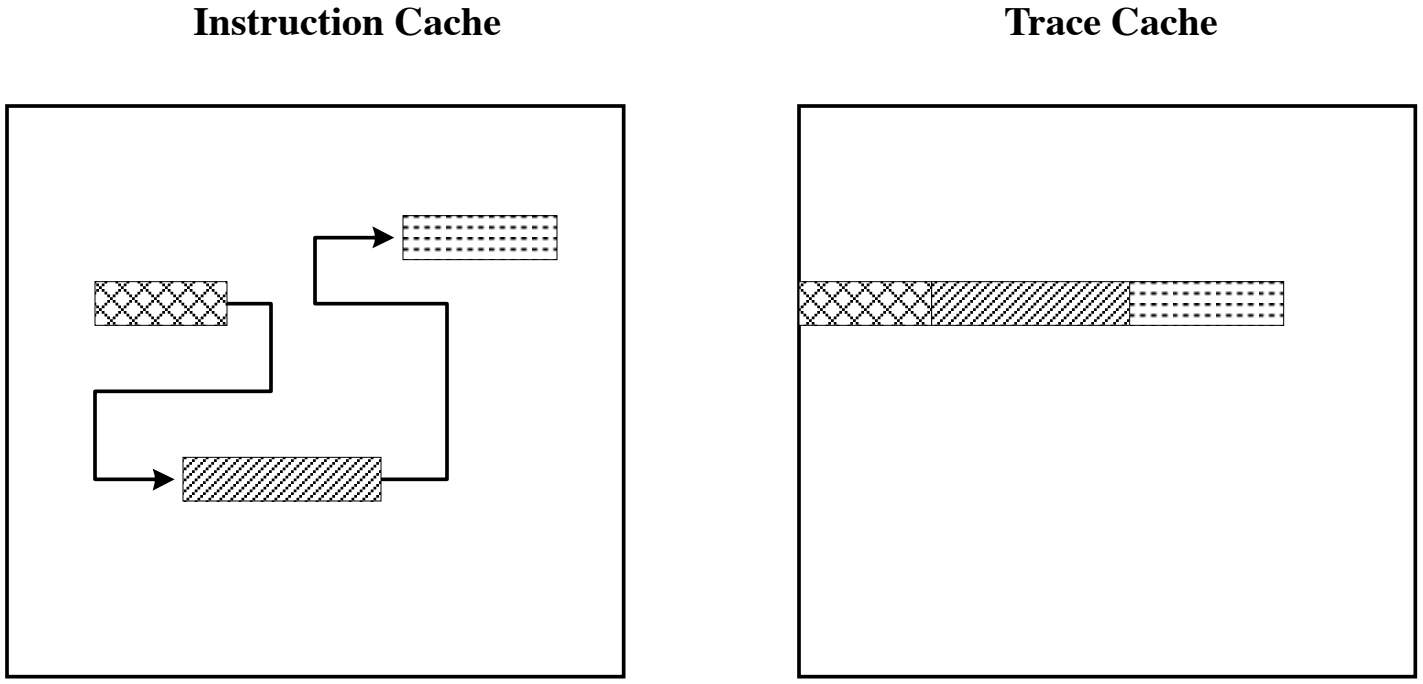
Intel Hyperthreading

- Main pipeline
  - Pipeline prior to trace cache not shown
- Round-Robin instruction fetching
  - Alternates between threads
  - Avoids dual-ported trace cache
  - BUT trace cache is a shared resource



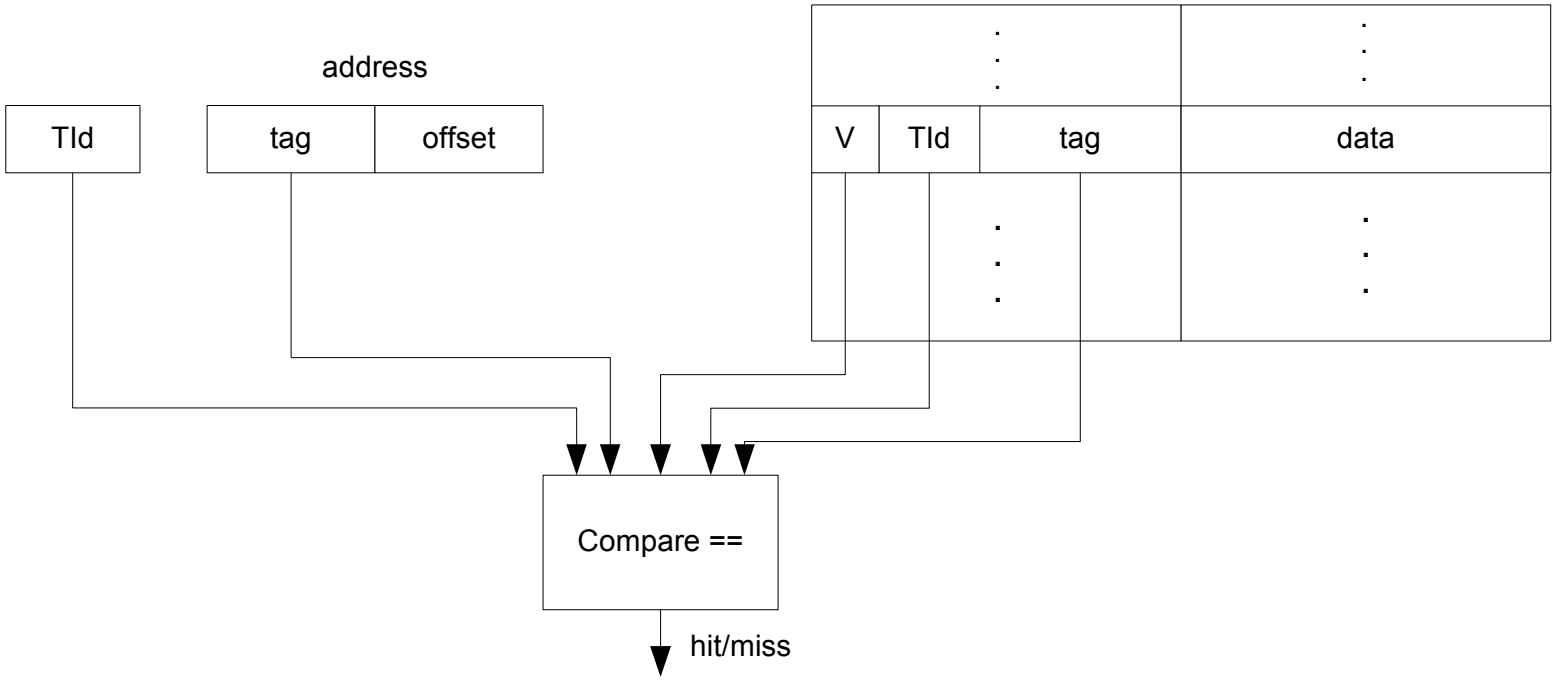
Trace Caches

- Trace cache captures dynamic traces
- Increases fetch bandwidth
- Help shorten pipeline (if predecoded)



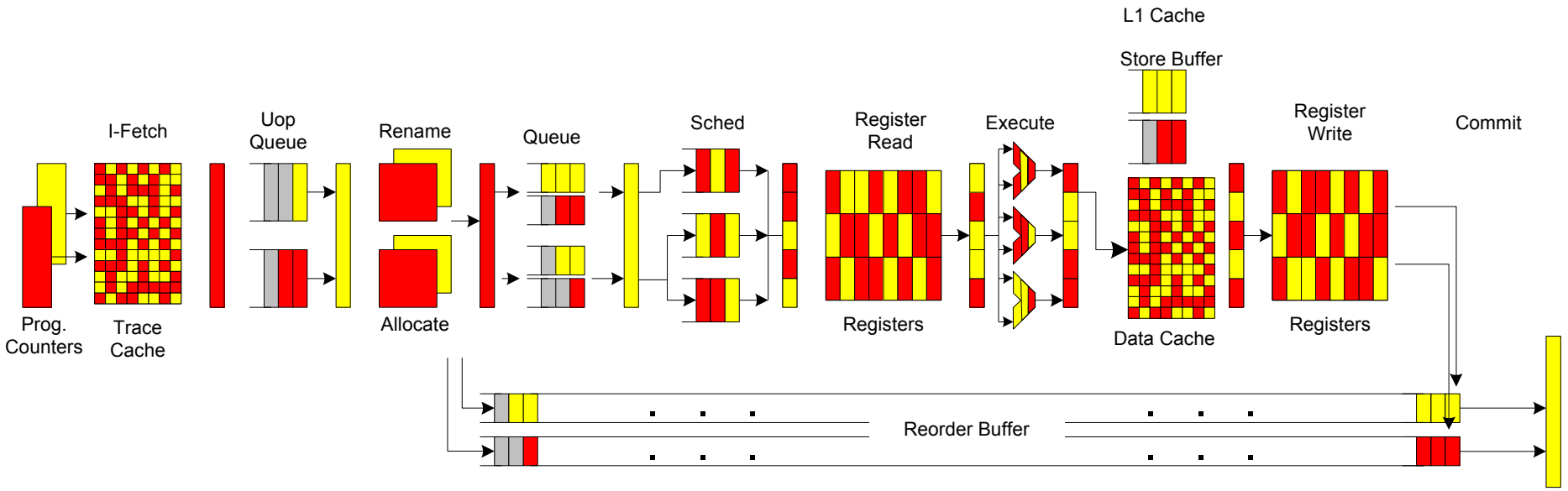
Capacity Resource Sharing

- Append thread identifier (TId) to threads in shared capacity (storage) resource
- Example: cache memory



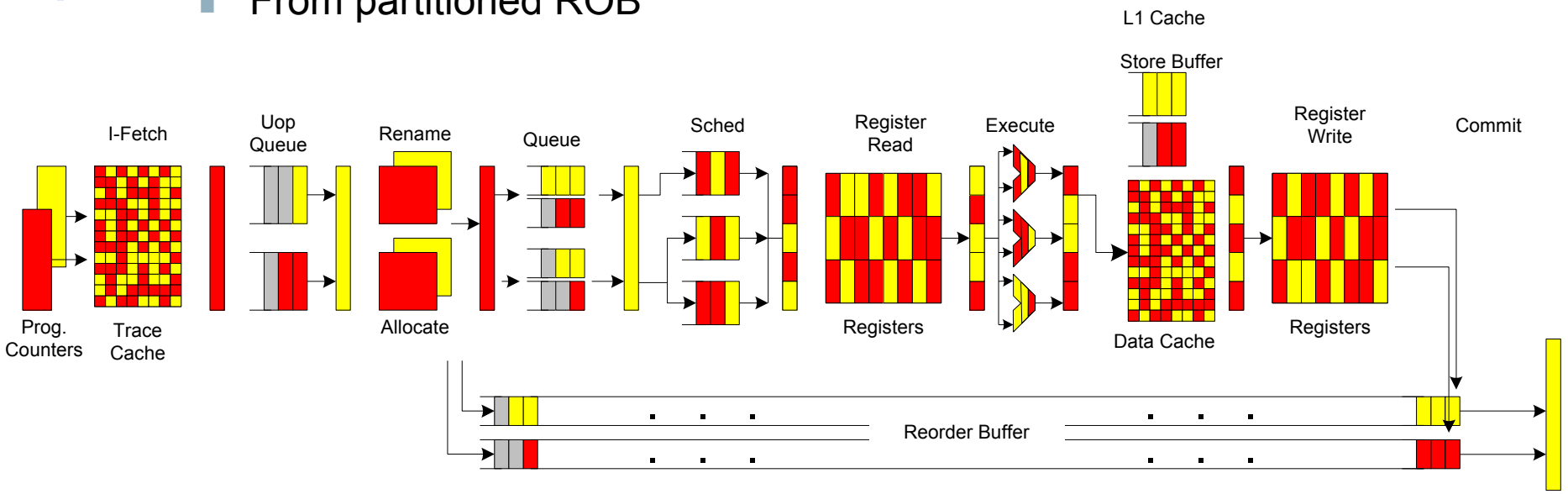
Frontend Implementation

- Partitioned front-end resources
  - Fetch queue (holds uops)
  - Rename and allocate tables
  - Post-rename queues
- Partitioning assures forward progress if other thread is blocked
  - Round-robin scheduling



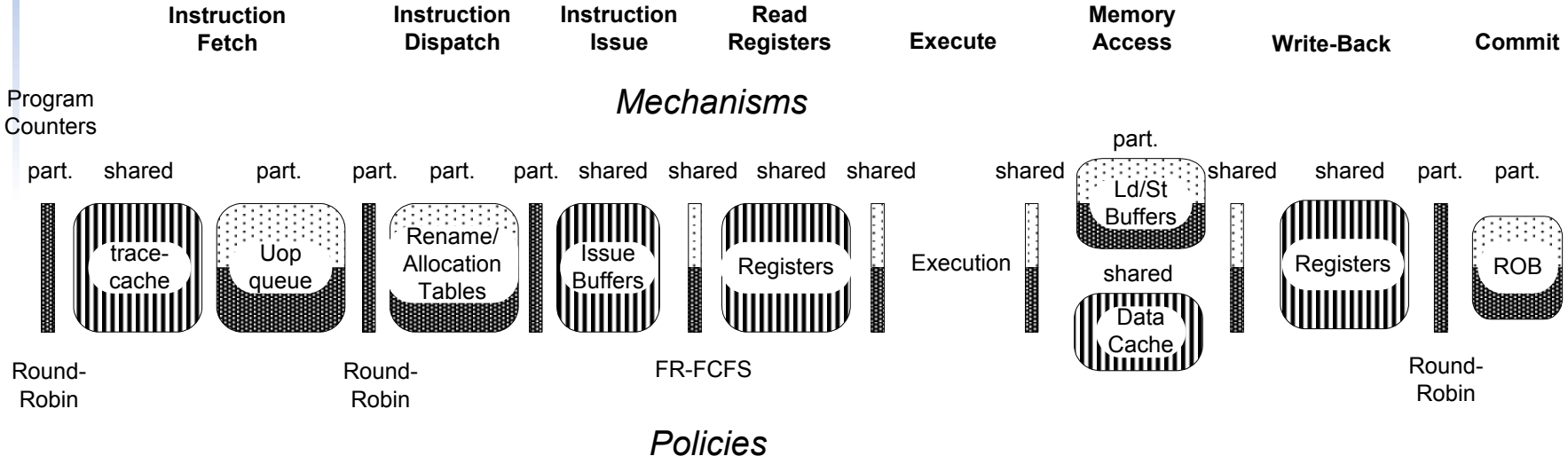
Backend Implementation

- Physical registers are pooled (shared)
- Five instruction buffers (schedulers)
  - Shared
  - With an upper limit
- Instruction issue is irrespective of thread ID
- Instruction commit is round-robin
  - From partitioned ROB



Example: Hyperthreading

- Mechanisms (and Policies) in Pentium 4



Performance

- OLTP workload
  - 21% gain in single and dual systems
  - Likely external bottleneck in 4 processor systems
    - Most likely front-side bus (FSB), i.e. memory bandwidth