Introducción a paralelismo a nivel de instrucción

Arquitectura de Computadores

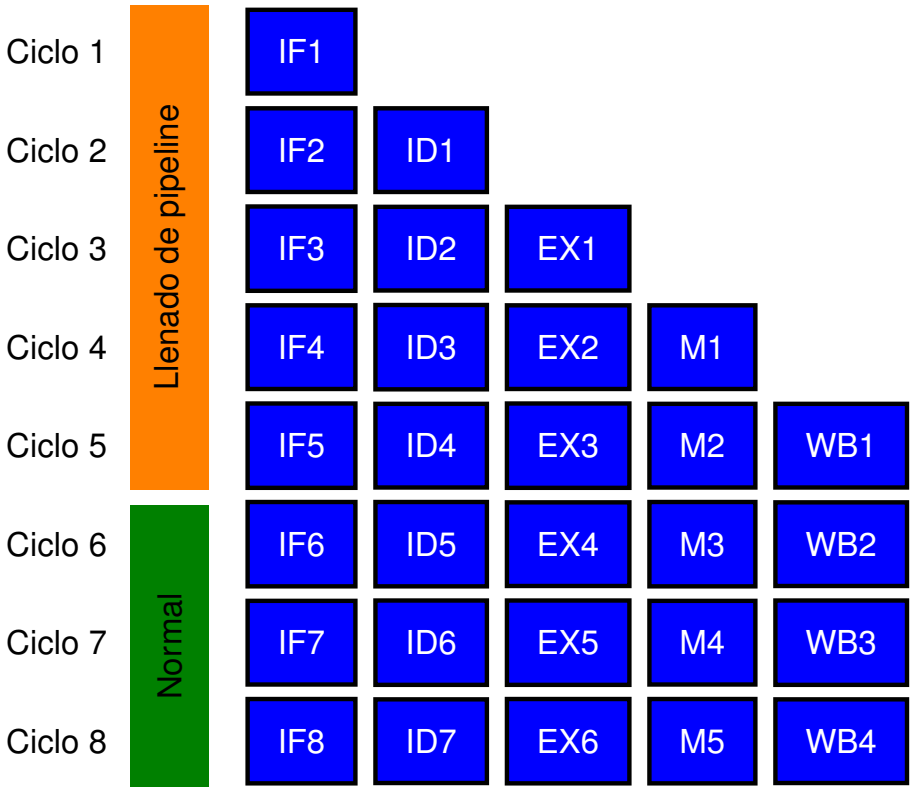
J. Daniel García Sánchez (coordinador)
David Expósito Singh
Javier García Blas
Óscar Pérez Alonso
J. Manuel Pérez Lobato

Grupo ARCOS
Departamento de Informática
Universidad Carlos III de Madrid

Segmentación

- Técnica de implementación en la que una la ejecución de múltiples instrucciones se solapan en el tiempo.
  - Se divide una operación **costosa** en varias sub-operaciones simples.
  - Ejecución de las sub-operaciones por etapas.
- **Efectos**:
  - **Aumenta** el *throughput*.
  - No disminuye la latencia.

Segmentación



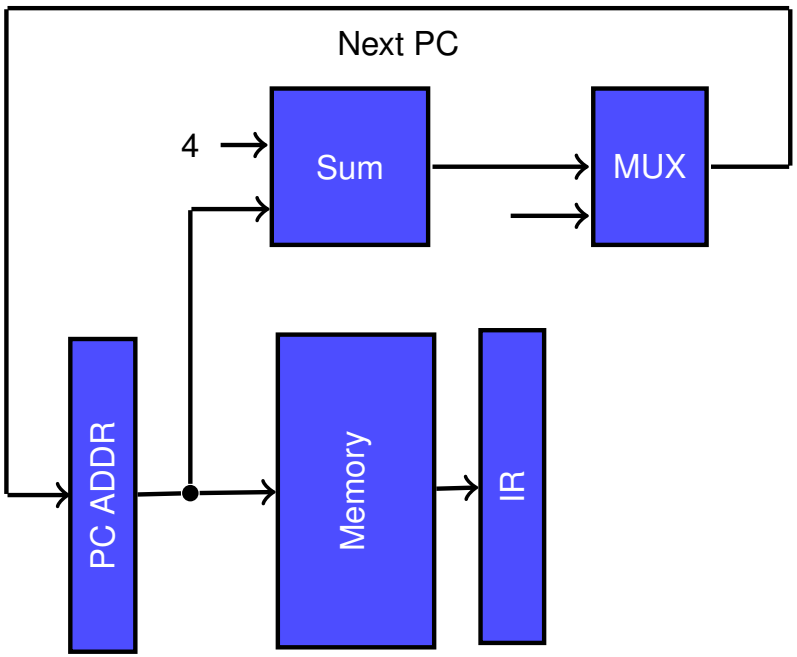
- **Latencia**:
  - Una instrucción requiere 5 etapas.
  - 5 ciclos.
- **Throughput**:
  - Termina una instrucción en cada ciclo (con pipeline lleno).
  - 1 instrucción por ciclo.

Etapas del pipeline

- Modelo simplificado:
  - 5 etapas.
  - Modelos más realistas requieren muchas más etapas.

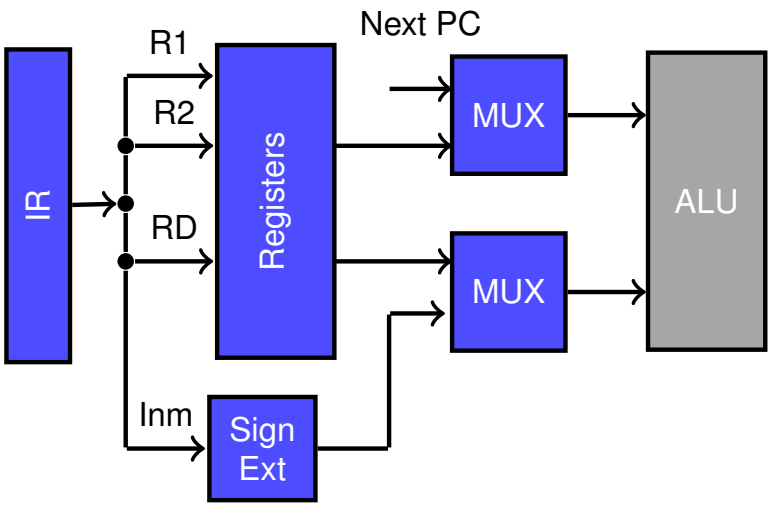
- **Etapas**:
  - **Captación** (*Instruction Fetch*).
  - **Decodificación** (*Instruction Decode*).
  - **Ejecución** (*Execution*).
  - **Memoria** (*Memory*).
  - **Post-escritura** (*Write-back*).

Captación



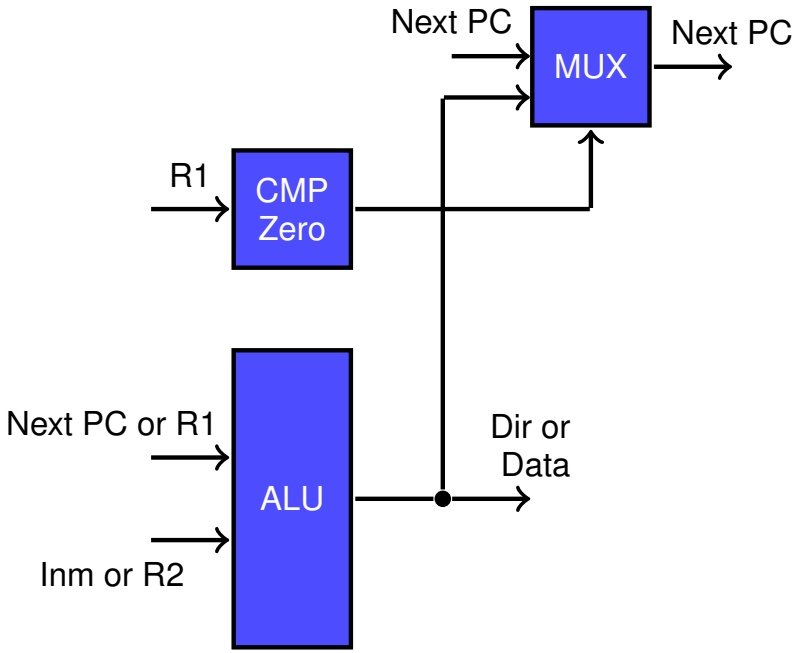
- Envío de PC a memoria.
- Lectura de instrucción.
- Actualización de PC.

Decodificación



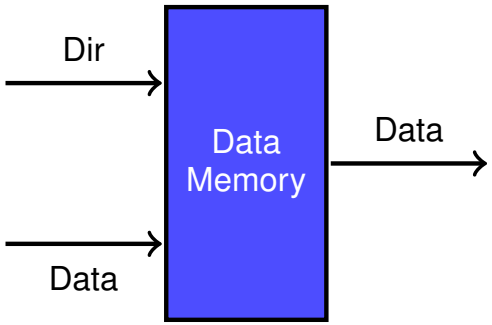
- Decodificación de instrucción.
- Lectura de registros.
- Extensión de signo de desplazamientos.
- Cálculo de posible dirección de salto.

Ejecución



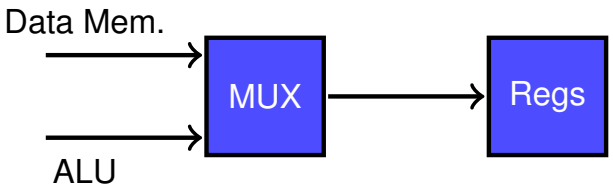
- Operación de ALU sobre registros.
- Alternativamente, cálculo de dirección efectiva.

Memoria



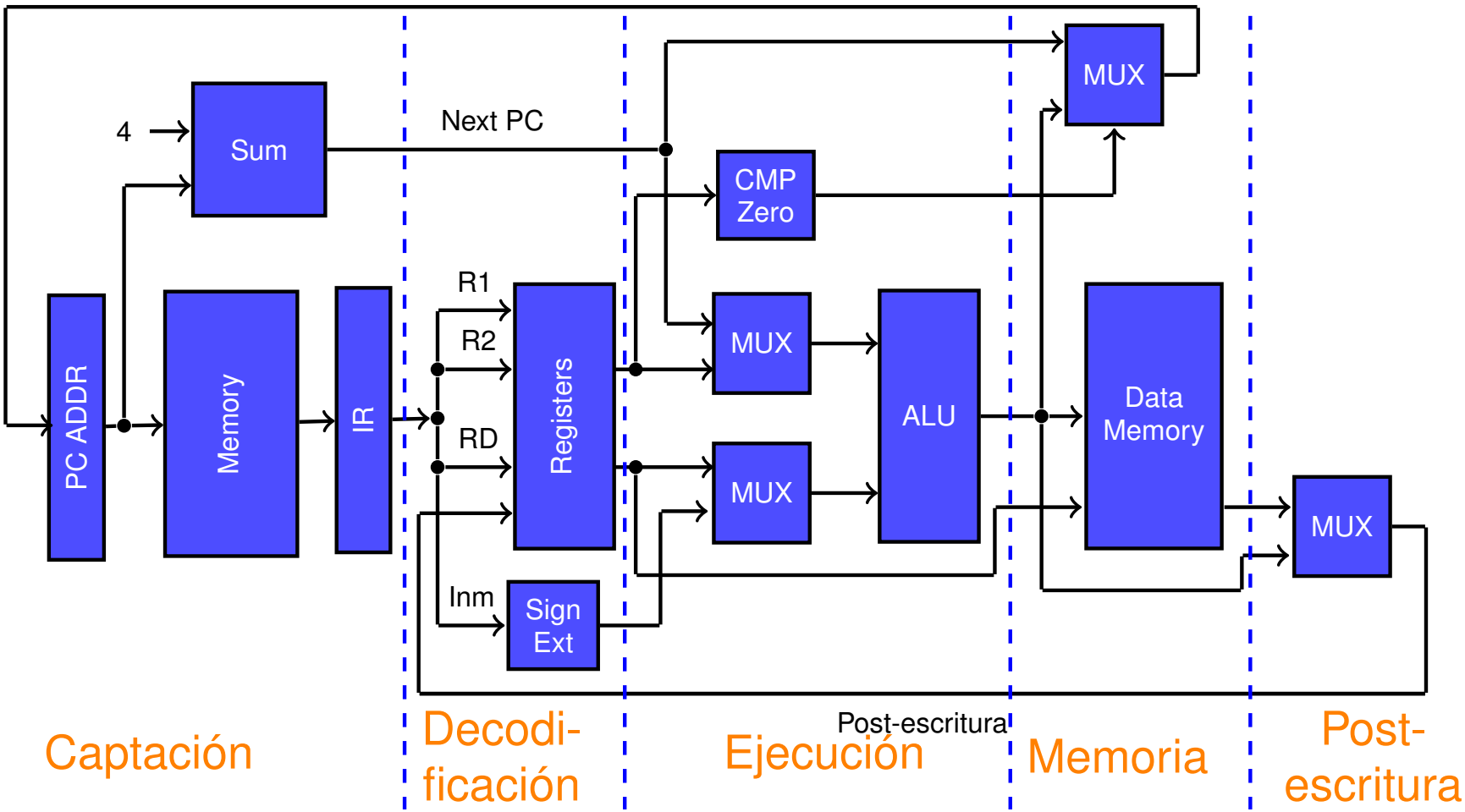
- Lectura o escritura en memoria.

Post-escritura



- Escritura de resultado en banco de registros.

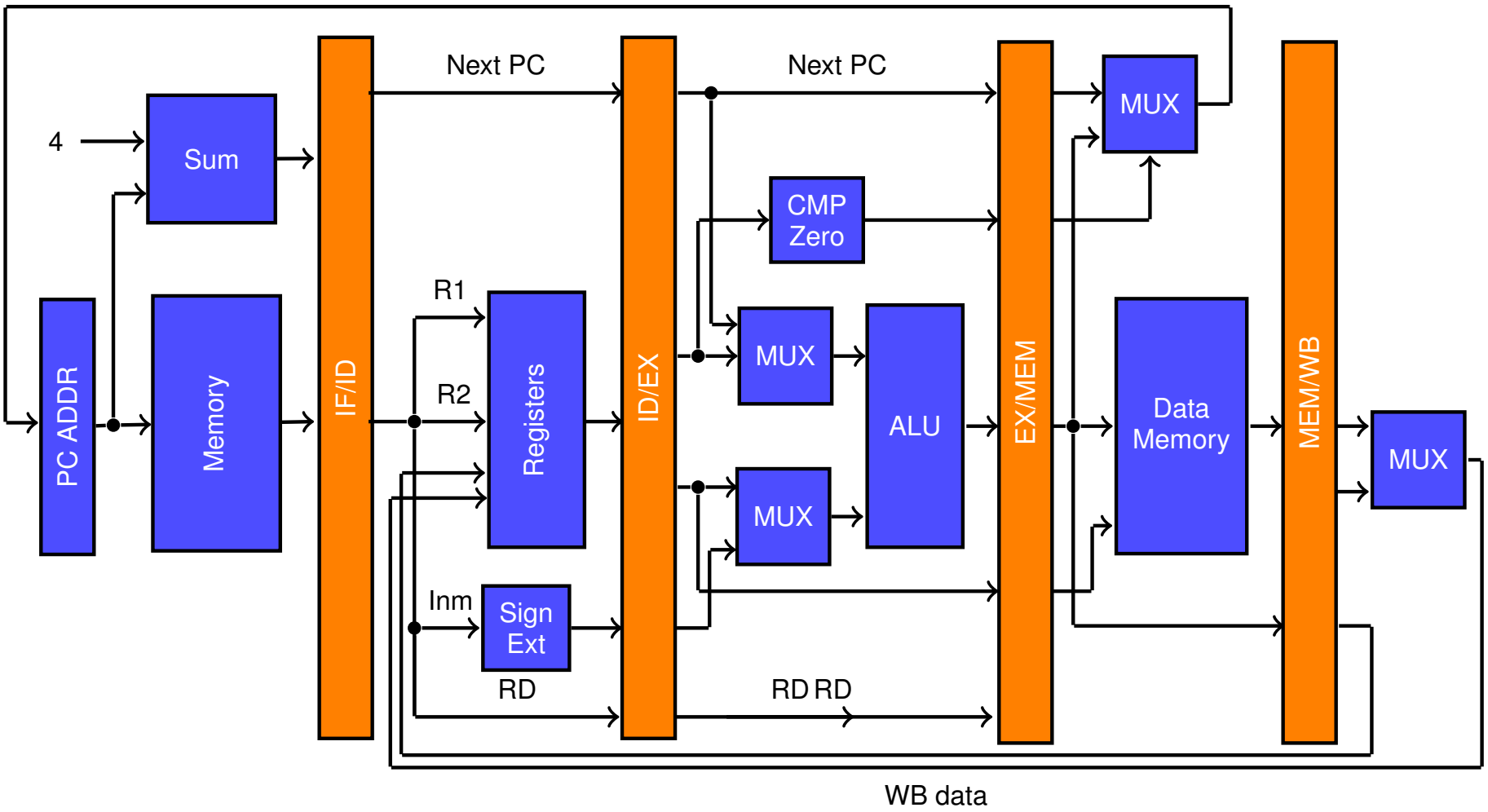
Arquitectura general



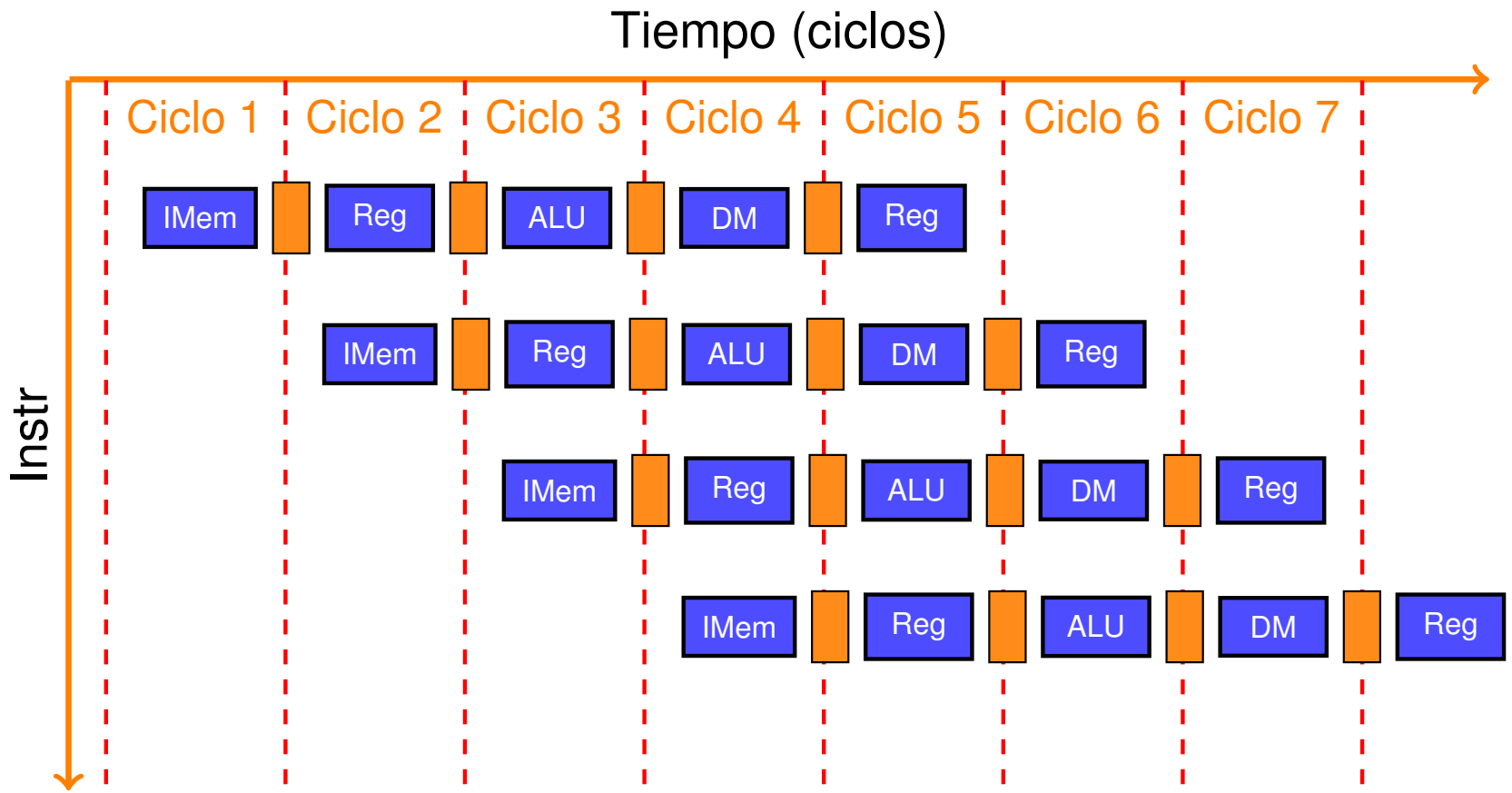
Efectos del pipeline

- Un pipeline de **profundidad** n, multiplica por n el **ancho de banda** necesario de la versión sin pipeline con la misma frecuencia de reloj.
  - Cachés, cachés, ...
- La separación de **caché** de **datos** e **instrucciones** elimina algunos conflictos de memoria.
- Las instrucciones que están en el pipeline no deberían intentar usar el mismo recurso en el mismo momento.
  - Introducción de registros de pipeline entre etapas sucesivas.

Comunicación entre etapas



El pipeline a lo largo del tiempo



- Lectura registros segunda mitad de ciclo.

Ejemplo

- Procesador no segmentado.
  - Ciclo de reloj: 1 ns.
  - 40% operaciones ALU $\rightarrow$ 4 ciclos.
  - 20% operaciones de bifurcación $\rightarrow$ 4 ciclos.
  - 40% operaciones de memoria $\rightarrow$ 5 ciclos.
  - Sobrecoste de segmentación $\rightarrow$ 0.2 ns.
- ¿Qué aceleración se obtiene con la segmentación?

$$t_{orig} = ciclo_{reloj} \times CPI = 1ns \times (0.6 \times 4 + 0.4 \times 5) = 4.4ns$$

$$t_{nuevo} = 1ns + 0.2ns = 1.2ns$$

$$S = \frac{4.4ns}{1.2ns} = 3.7$$

Riesgos

- Un **riesgo** es una situación que impide que la siguiente instrucción pueda comenzar en el ciclo de reloj previsto.
  - Los riesgos reducen el rendimiento de las arquitecturas segmentadas.
- Tipos de riesgos:
  - Riesgo estructural.
  - Riesgo de datos.
  - Riesgo de control.
- Aproximación simple ante riesgos:
  - Detener (*stall*) el flujo de instrucciones.
  - Las instrucciones que ya han iniciado continúan.

Riesgo estructural

- Se produce cuando el hardware no puede soportar todas las posibles secuencias de instrucciones.
  - En un mismo ciclo dos etapas de la segmentación necesitan hacer uso del **mismo recurso**.

- **Razones**:
  - Unidades funcionales no totalmente segmentadas.
  - Unidades funcionales no duplicadas.

- Este tipo de riesgos es evitable pero encarece el hardware.

Speedup de la segmentación

- **Speedup de la segmentación**:
  - t_{noseg}: Tiempo medio de instrucción en arquitectura no segmentada.
  - t_{seg}: Tiempo medio de instrucción en arquitectura segmentada.

$$S = \frac{t_{noseg}}{t_{seg}} = \frac{CPI_{noseg} \times ciclo_{noseg}}{CPI_{seg} \times ciclo_{seg}}$$

- En el caso ideal el **CPI** segmentado es 1.
  - Hay que añadir ciclos de detención por instrucción.

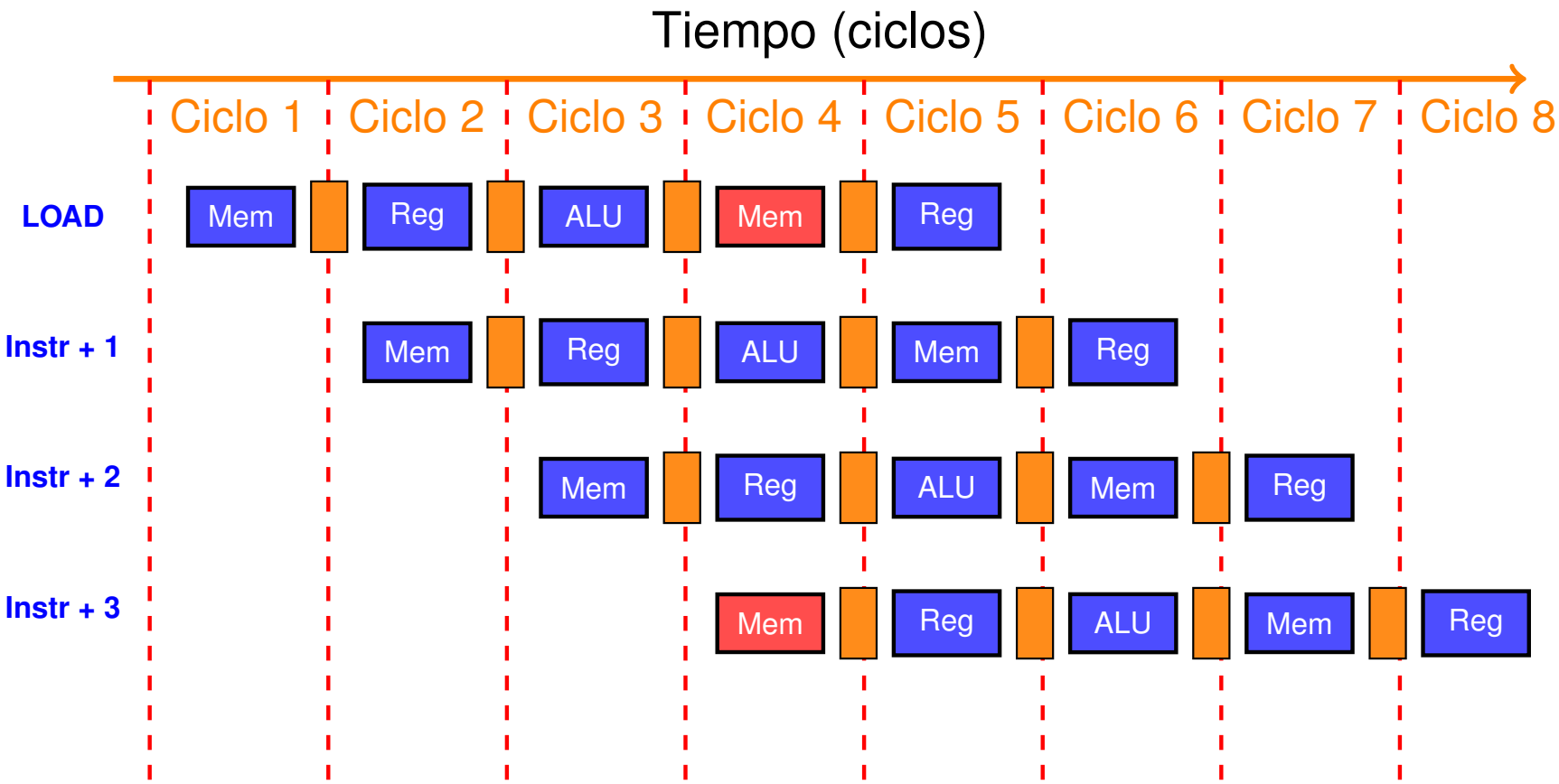
Speedup de la segmentación

- En el caso del procesador no segmentado.
  - $CPI = 1$, con $ciclo_{noseg} > ciclo_{seg}$.
  - $ciclo_{noseg} = N \times ciclo_{seg}$.
  - $N \rightarrow$ **Profundidad del pipeline**.

Speedup

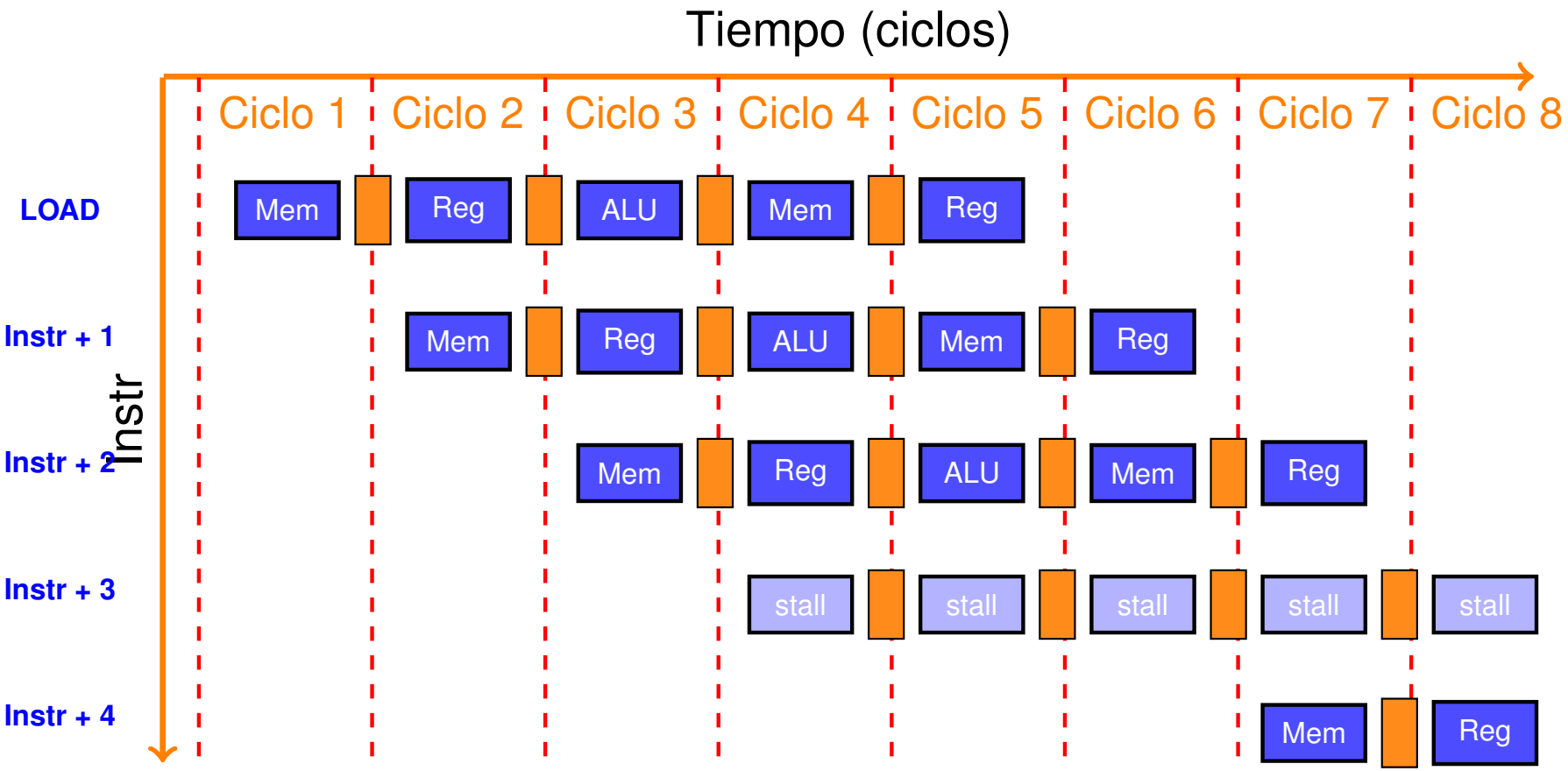
$$S = \frac{N}{1 + \text{detenciones por instrucción}}$$

Riesgos estructurales: ejemplo



- Asumiendo memoria de un único puerto.

Riesgos estructurales: ejemplo



- Asumiendo memoria de un único puerto.

Ejemplo

- Se consideran dos diseño alternativos:
  - **A**: Sin riesgos estructurales.
    - Ciclo de reloj $\rightarrow 1ns$
  - **B**: Con riesgos estructurales.
    - Ciclo de reloj $\rightarrow 0.9ns$
  - Instrucciones de acceso a datos con riesgo $\rightarrow 30\%$.
- ¿Alternativa más rápida?

$$t_{inst}(A) = CPI \times ciclo = 1 \times 1ns = 1ns$$

$$t_{inst}(B) = CPI \times ciclo = (0.7 \times 1 + 0.3 \times (1+1)) \times 0.9ns = 1.17ns$$

Riesgo de datos

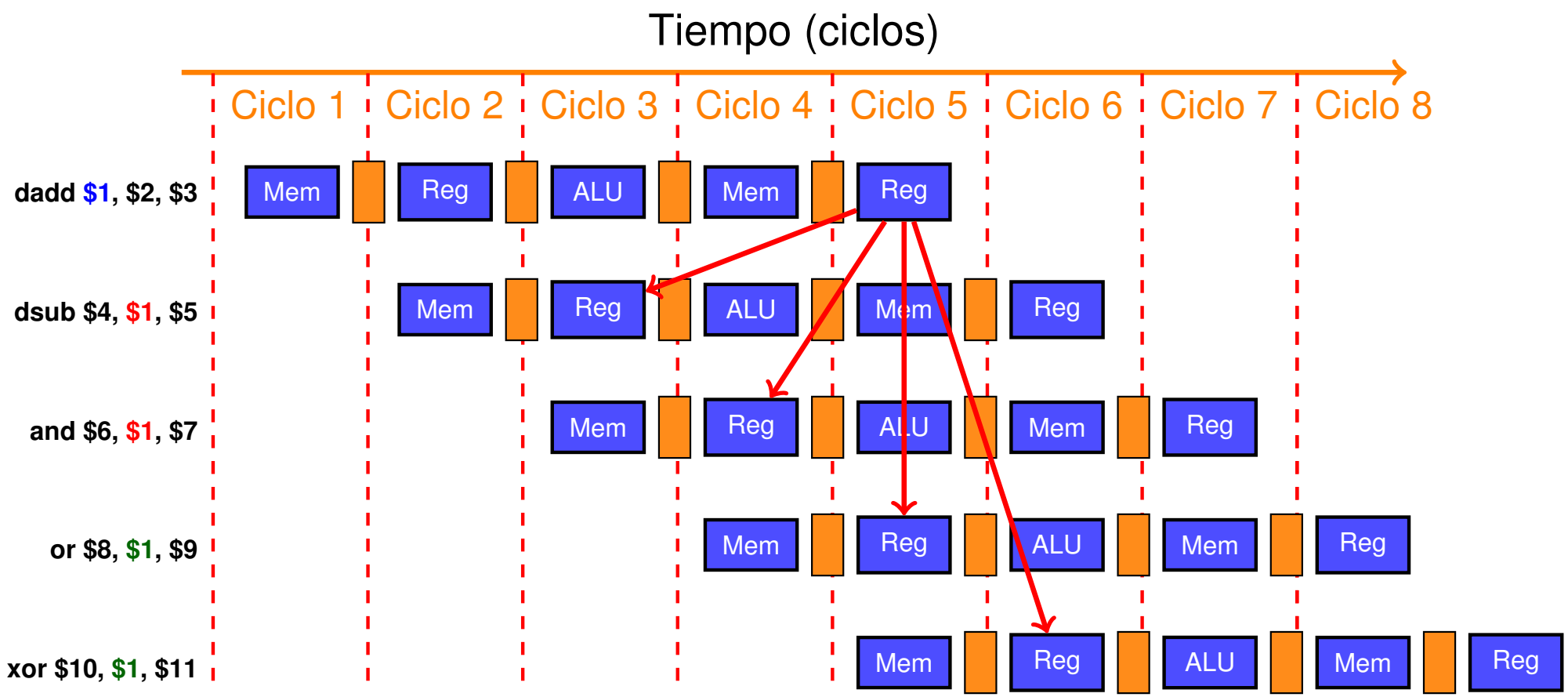
- Se produce un **riesgo de datos** cuando la segmentación modifica el orden de accesos de lectura/escritura a los operandos.

Ejemplo

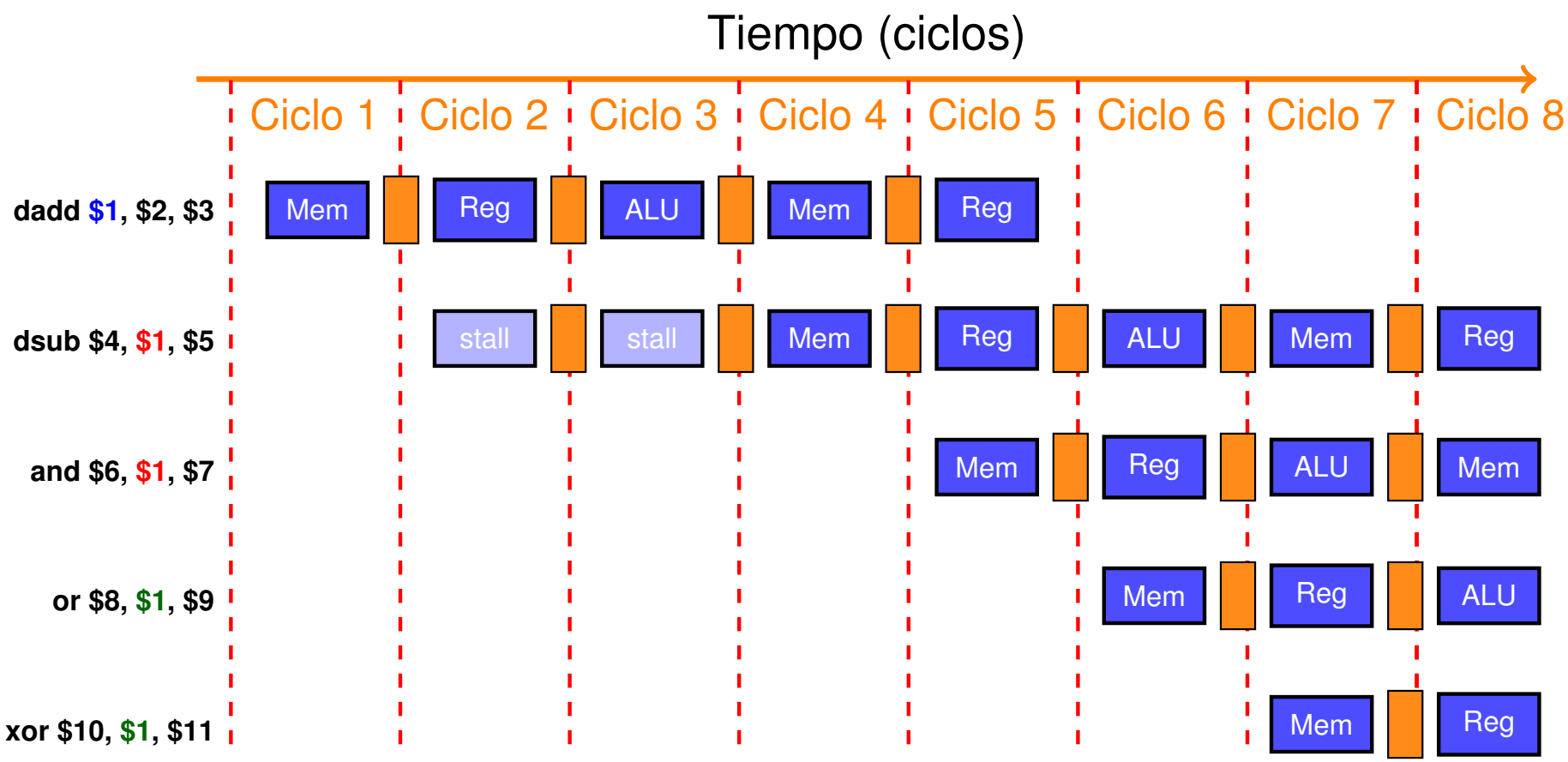
```
I1: dadd $1, $2, $3
I2: dsub $4, $1, $5
I3: and  $6, $1, $7
I4: or   $8, $1, $9
I5: xor  $10, $1, $11
```

- **I2** lee **R1** antes que **I1** la modifique.
- **I3** lee **R1** antes de que **I1** la modifique.
- **I4** obtiene valor correcto.
  - Banco de registros leído en segunda mitad de ciclo.
- **I5** obtiene valor correcto.

Riesgos de datos



Detenciones en riesgos de datos



Riesgos de datos: RAW

- Lectura después de escritura (**Read After Write**).
  - La instrucción **i+1** trata de leer un dato antes de que la instrucción **i** lo escriba.

Ejemplo

```
i :    add $1, $2, $3
i+1:   sub $4, $1, $3
```

- Si hay dependencia de datos, las instrucciones:
  - No pueden ejecutarse en paralelo ni solaparse completamente.
  - La instrucción **sub** necesita el valor de **$1** producido por la instrucción **add**.

- **Soluciones**:
  - **Detección hardware**.
  - **Control por el compilador**.

Riesgos de datos: WAR

- Escritura después de lectura (**Write After Read**).
  - La instrucción **i+1** modifica operando antes de que la instrucción **i** lo lea.

Ejemplo

```
i :   sub $4, $1, $3
i+1: add $1, $2, $3
i+2: mul $6, $1, $7
```

- Conocido como **antidependencia** en compiladores.
  - Reutilización de nombre
  - La instrucción **add** modifica **$1** antes de **sub** la lea.

- No puede ocurrir en un MIPS con **pipeline de 5 etapas**.
  - Todas las instrucciones de 5 etapas.
  - Lecturas siempre en etapa 2.
  - Escrituras siempre en etapa 5.

Riesgos de datos: WAW

- Escritura después de escritura (**Write After Write**).
  - La instrucción **i+1** modifica el operando antes de que la instrucción **i** lo modifica.

Ejemplo

```
i :   sub $1, $4, $3
i+1: add $1, $2, $3
i+2: mul $6, $1, $7
```

- Conocido como **dependencia de salida** en compiladores.
  - Reutilización de nombre
  - La instrucción **add** modifica **$1** antes de **sub** la modifique.

- No puede ocurrir en un MIPS con **pipeline de 5 etapas**.
  - Todas las instrucciones de 5 etapas.
  - Escrituras siempre en etapa 5.

Soluciones a los riesgos de datos

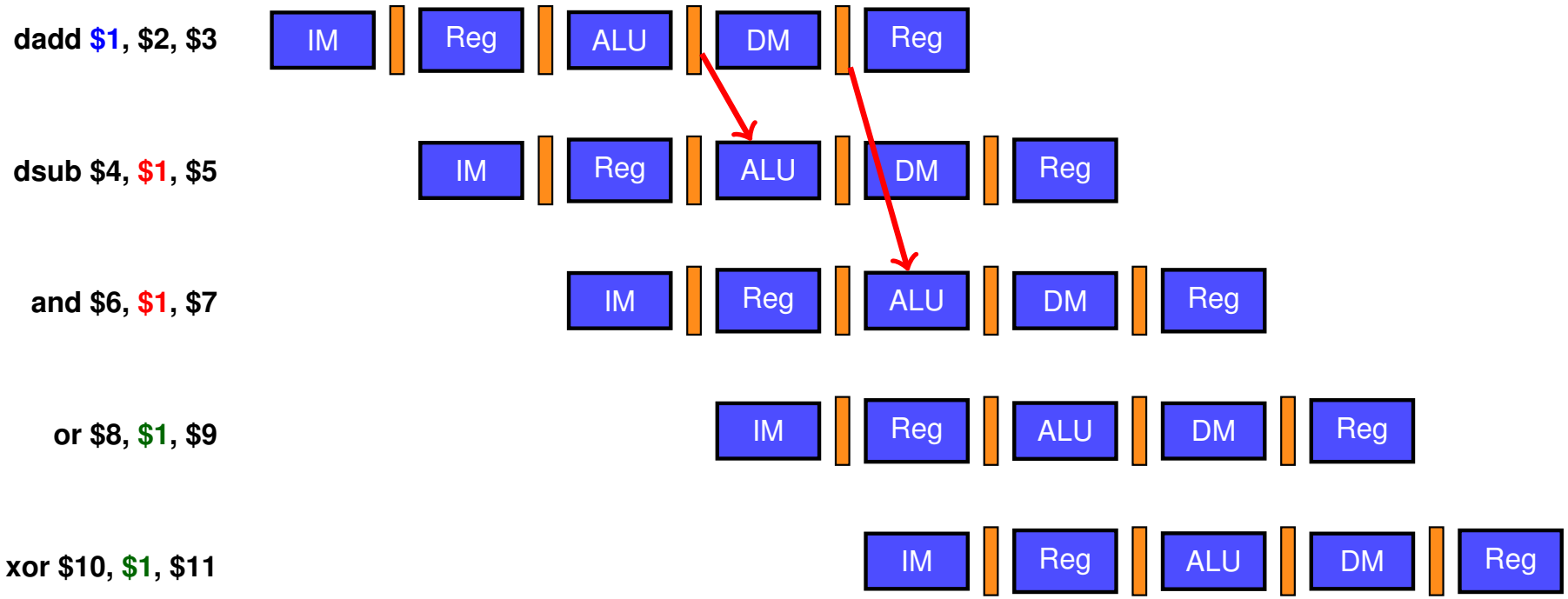
- **Dependencias RAW**:
  - Envío adelantado (**forwarding**).

- **Dependencias WAR y WAW**:
  - **Renombrado de registros**.
    - Estáticamente por el compilador.
    - Dinámicamente por el hardware.

Envío adelantado

- Técnica para evitar algunas detenciones de datos.
- **Idea básica**:
  - No hace falta esperar a que el resultado se escriba en el banco de registros.
  - Ya está en los registros de segmentación.
  - Se puede usar ese valor en vez del que hay en el banco de registros.
- **Implantación**:
  - Los resultados de las fases EX y MEM se escriben en registros de entrada a ALU.
  - La lógica de *forwarding* selecciona entre entradas reales y registros de *forwarding*.

Forwarding

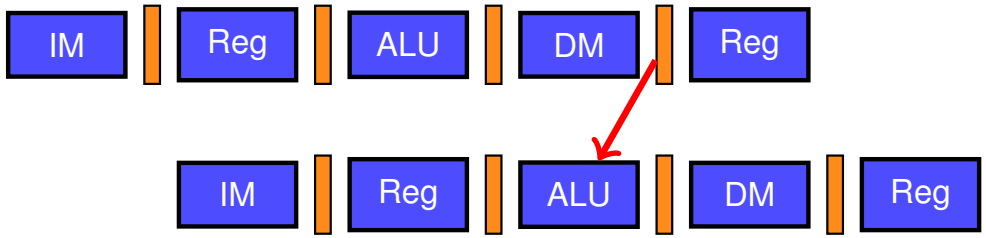

dadd $1, $2, $3
IM
Reg
ALU
DM
Reg
dsub $4, $1, $5
IM
Reg
ALU
DM
Reg
and $6, $1, $7
IM
Reg
ALU
DM
Reg
or $8, $1, $9
IM
Reg
ALU
DM
Reg
xor $10, $1, $11
IM
Reg
ALU
DM
Reg

Limitaciones del *forwarding*

- No todos los riesgos se pueden evitar con forwarding.
  - ¡No se puede viajar hacia atrás en el tiempo!

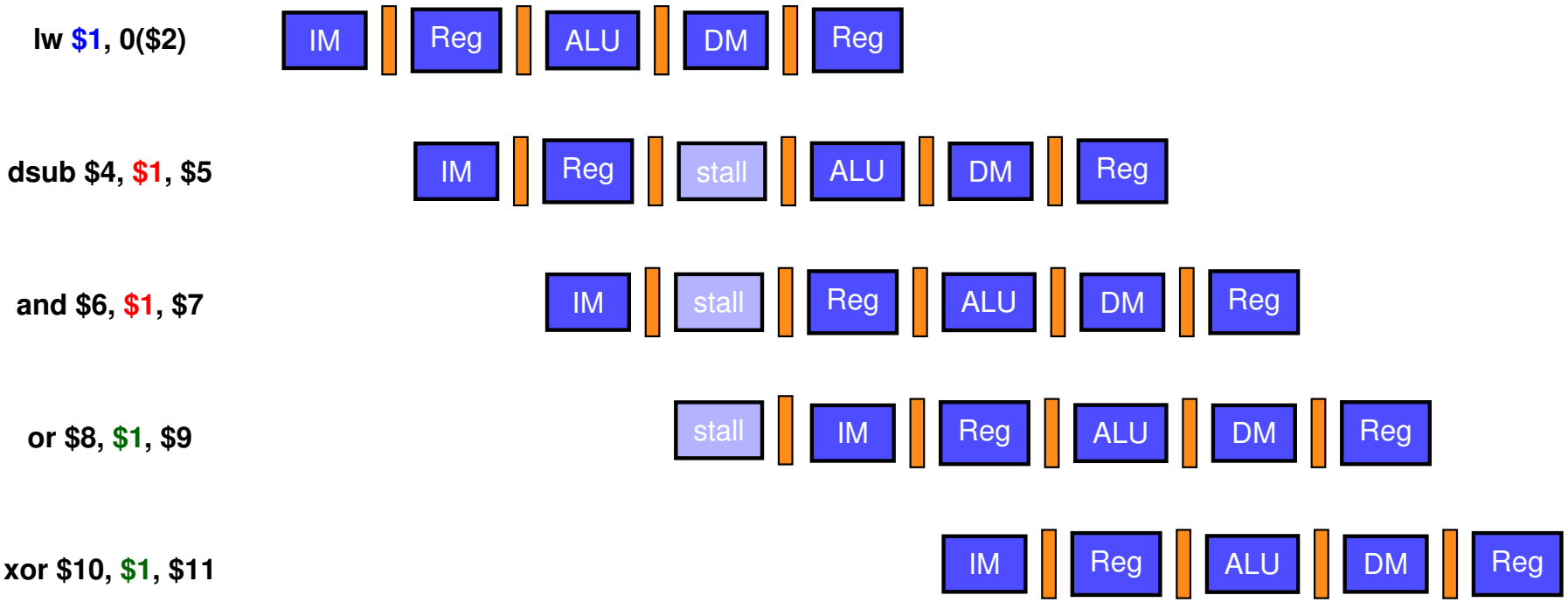
Ejemplo

```
I1: lw   $1, (0)$2
I2: dsub $4, $1, $5
I3: and  $6, $1, $7
I4: or   $8, $1, $9
I5: xor  $10, $1, $11
```



- Si el riesgo no se puede evitar se debe introducir una detención.

Detenciones en acceso a memoria



Riesgos de control

- Un **riesgo de control** se produce en una instrucción de alteración del PC.
  - **Terminología**:
    - **Bifurcación tomada**: Si se modifica el PC.
    - **Bifurcación no tomada**: Si no se modifica el PC.
  - Problema:
    - La segmentación asume que lo bifurcación **no** se tomará.
    - ¿Qué hacer si después de la etapa ID se determina que hay que tomar la bifurcación?

Alternativas ante riesgos de control

- **Tiempo de compilación**: Prefijadas durante toda la ejecución del programa.
  - El software puede intentar minimizar su impacto si conoce el comportamiento del hardware.
  - El compilador puede hacer este trabajo.

- **Alternativas en tiempo de ejecución**: Comportamiento variable durante la ejecución del programa.
  - Intentan predecir qué hará el software.

Riesgos de control: soluciones estáticas

1. Congelación del *pipeline*.
2. Predicción prefijada.
   - Siempre no tomado.
   - Siempre tomado.
3. Bifurcación con retraso.

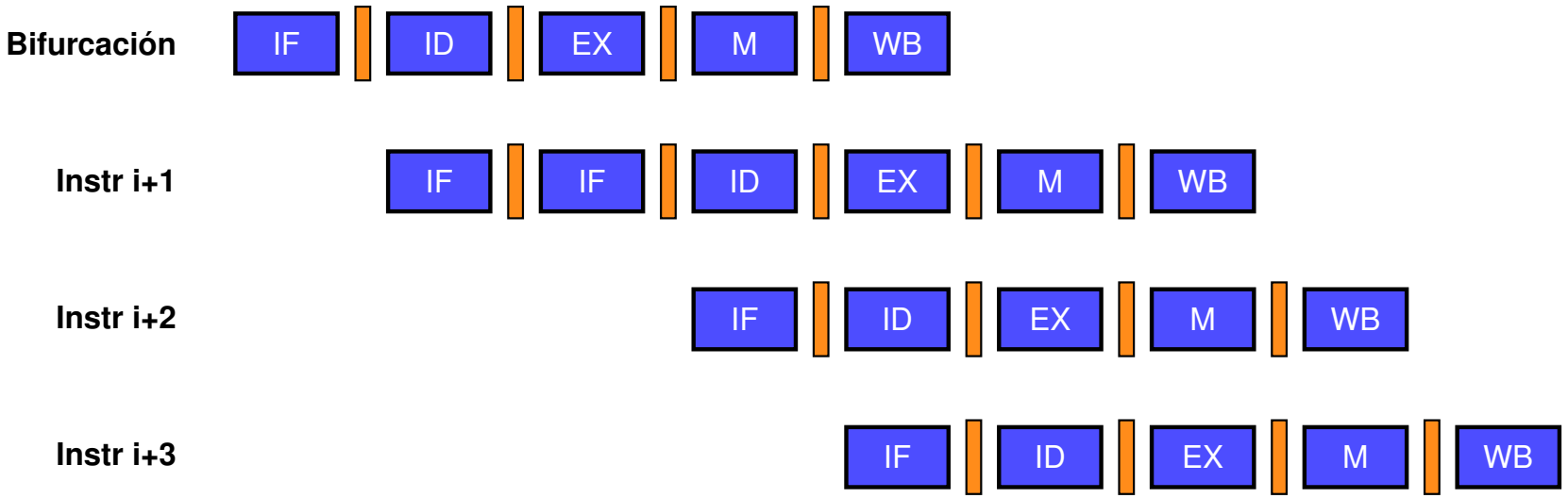
- En muchos casos el compilador necesita saber qué se va a hacer para reducir el impacto.

Congelación del pipeline

- **Idea**: Si la instrucción actual es una bifurcación $\rightarrow$ parar o eliminar del pipeline instrucciones posteriores hasta que se conozca el destino.
  - Penalización en tiempo de ejecución conocida.
  - El software (compilador) no puede hacer nada.

- El destino de la bifurcación se conoce en la etapa **ID**.
  - Repetir el **FETCH** de la siguiente instrucción.

Repetición de captación



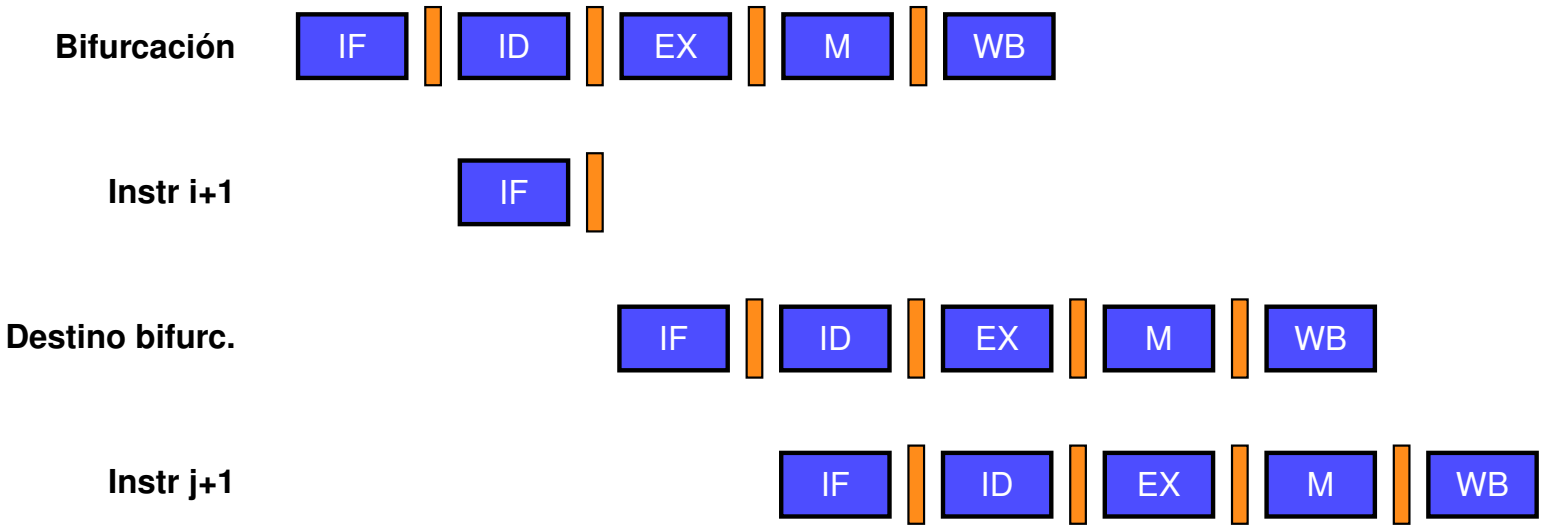
- La repetición de IF equivale a una detención.
- Una detención por bifurcación puede dar lugar a una **pérdida de rendimiento** de 10% a 30%.

Predicción prefijada: no tomada

- **Idea**: Asumir que la bifurcación será **no tomada**.
  - Se evita modificar el estado del procesador hasta que se tiene la confirmación de que la bifurcación no se toma.
  - Si la bifurcación se toma, las instrucciones siguientes se retiran del pipeline y se capta la instrucción en el destino del salto.
    - Transformar instrucciones en **NOP**.

- **Tarea del compilador**:
  - Organizar código poniendo la opción más frecuente como no tomada e invirtiendo condición si es necesario.

Predicción prefijada: no tomada



- Cuando se sabe que el salto se tomará se capta la nueva instrucción (*j+1*).

Predicción prefijada: tomada

- **Idea**: Asumir que la bifurcación será **tomada**.
  - Tan pronto como se decodifica la bifurcación y se calcula el destino se comienza a captar instrucciones del destino.
  - En pipeline de 5 etapas no aporta ventajas.
    - No se conoce dirección destino antes que decisión de bifurcación.
    - Útil en procesadores con condiciones complejas y lentas.

- **Tarea del compilador**:
  - Organizar código poniendo la opción más frecuente como tomada e invirtiendo condición si es necesario.

Bifurcación retrasada

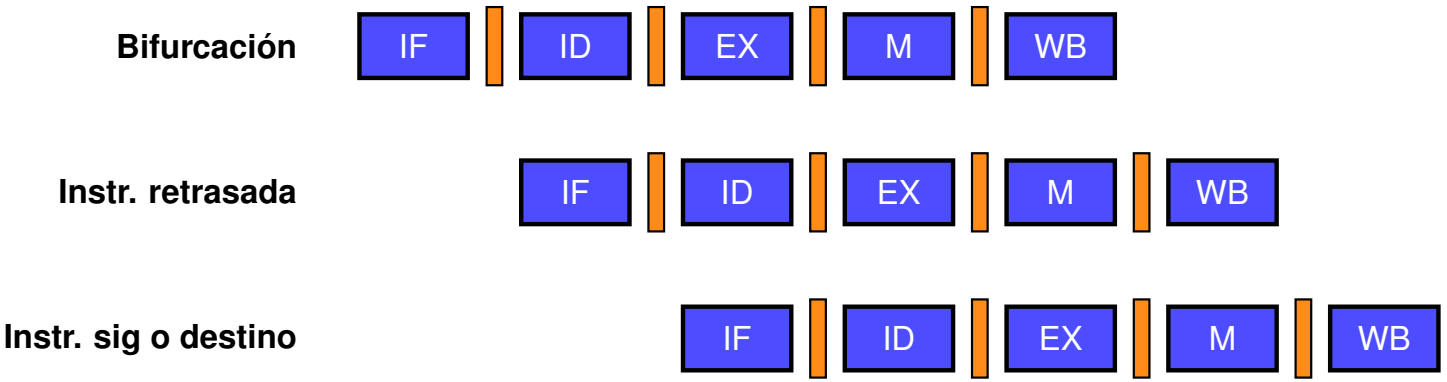
- **Idea**: La bifurcación se produce después de ejecutar las **n** instrucciones posteriores a la propia instrucción de bifurcación.
  - **En pipeline de 5 etapas** $\rightarrow$ 1 ranura de retraso (*delay slot*).

Ejemplo

```
I0:     bnez $1, etiq
I1:     addi $2, $2, 1
I2:     mul  $3, $2, $4
...
IN:     sub  $1, $1, 1
IN+1:   mul  $3, $3, $4
```

- Las instrucciones **I1**, **I2**, . . . , **IN** se ejecutan independientemente del sentido de la condición de bifurcación.
- La instrucción **IN+1** solamente se ejecuta si no se produce la bifurcación.

Bifrucación retrasada



- Caso de bifurcación retrasada con una ranura de retraso.
- Se espara siempre una instrucción antes de tomar la bifurcación.
- Es responsabilidad del programador poner código útil en la ranura.

Compilador y ranuras de retraso

Original

```
        add $1, $2, $3
        beqz $2, etiq
        nop
        xor $5, $6, $7
etiq:   and $8, $9, $10
```

Mejorado

```
        beqz $2, etiq
        add $1, $2, $3
        xor $5, $6, $7
etiq:   and $8, $9, $10
```

Original

```
etiq:   add $1, $2, $3
        xor $5, $6, $7
        beqz $5, etiq
        nop
        and $8, $9, $10
```

Mejorado

```
        add $1, $2, $3
etiq:   xor $5, $6, $7
        beqz $5, etiq
        add $1, $2, $3
        and $8, $9, $10
```

Original

```
        add $1, $2, $3
        beqz $1, etiq
        nop
        xor $5, $6, $7
etiq:   and $8, $9, $10
```

Mejorado

```
        add $1, $2, $3
        beqz $1, etiq
        xor $5, $6, $7
etiq:   and $8, $9, $10
```

Bifurcaciones retrasadas

- Efectividad del compilador para el caso de 1 ranura:
  - **Rellena** alrededor del 60% de ranuras.
  - En torno al 80% de instrucciones ejecutadas en ranuras **útiles** para computación.
  - En torno al 50% de ranuras **rellenados** de forma útil.

- Al usarse pipelines más profundos y emisión múltiple hacen falta más ranuras.
  - En desuso a favor de enfoque dinámicos.

Rendimiento y predicción prefijada

- El número de detenciones de bifurcaciones depende de:
  - Frecuencia de las bifurcaciones.
  - Penalización por bifurcación.

- Ciclos de penalización por bifurcación:

$$ciclos_{bifurc} = frecuencia_{bifurc} \times peniliz_{bifurc}$$

- Aceleración

$$S = \frac{profundidad_{pipeline}}{1 + frecuencia_{bifurc} \times peniliz_{bifurc}}$$

Caso práctico

- **MIPS R4000** (pipeline más profundo).
  - 3 etapas antes de conocer destino de bifurcación.
  - 1 etapa adicional para evaluar condición.
  - Asumiendo que no hay detenciones de datos en comparaciones.
  - **Frecuencia de bifurcaciones**:
    - **Bifurcación incondicional**: 4%.
    - **Bifurcación condicional, no-tomada**: 6%
    - **Bifurcación condicional, tomada**: 10%

Esquema bifurcación	Penalización		
	incondicional	no-tomada	tomada
Vaciar pipeline	2	3	3
Predecir tomada	2	3	2
Predecir no-tomada	2	0	3

Solución

Esquema bifurcación	Bifurcación			Total
	incondicional	no-tomada	tomada	
Frecuencia	4%	6%	10%	20%
Vaciar pipeline	$0.04 \times 2 = 0.08$	$0.06 \times 3 = 0.18$	$0.10 \times 3 = 0.30$	0.56
Predecir tomada	$0.04 \times 2 = 0.08$	$0.06 \times 3 = 0.18$	$0.10 \times 2 = 0.20$	0.46
Predecir no-tomada	$0.04 \times 2 = 0.08$	$0.06 \times 0 = 0.00$	$0.10 \times 3 = 0.30$	0.38

Contribución sobre CPI ideal

Speedup de predecir **tomada** sobre vaciar pipeline

$$S = \frac{1 + 0.56}{1 + 0.46} = 1.068$$

Speedup de predecir **no tomada** sobre vaciar pipeline

$$S = \frac{1 + 0.56}{1 + 0.38} = 1.130$$

Bifurcaciones y tiempo de ejecución

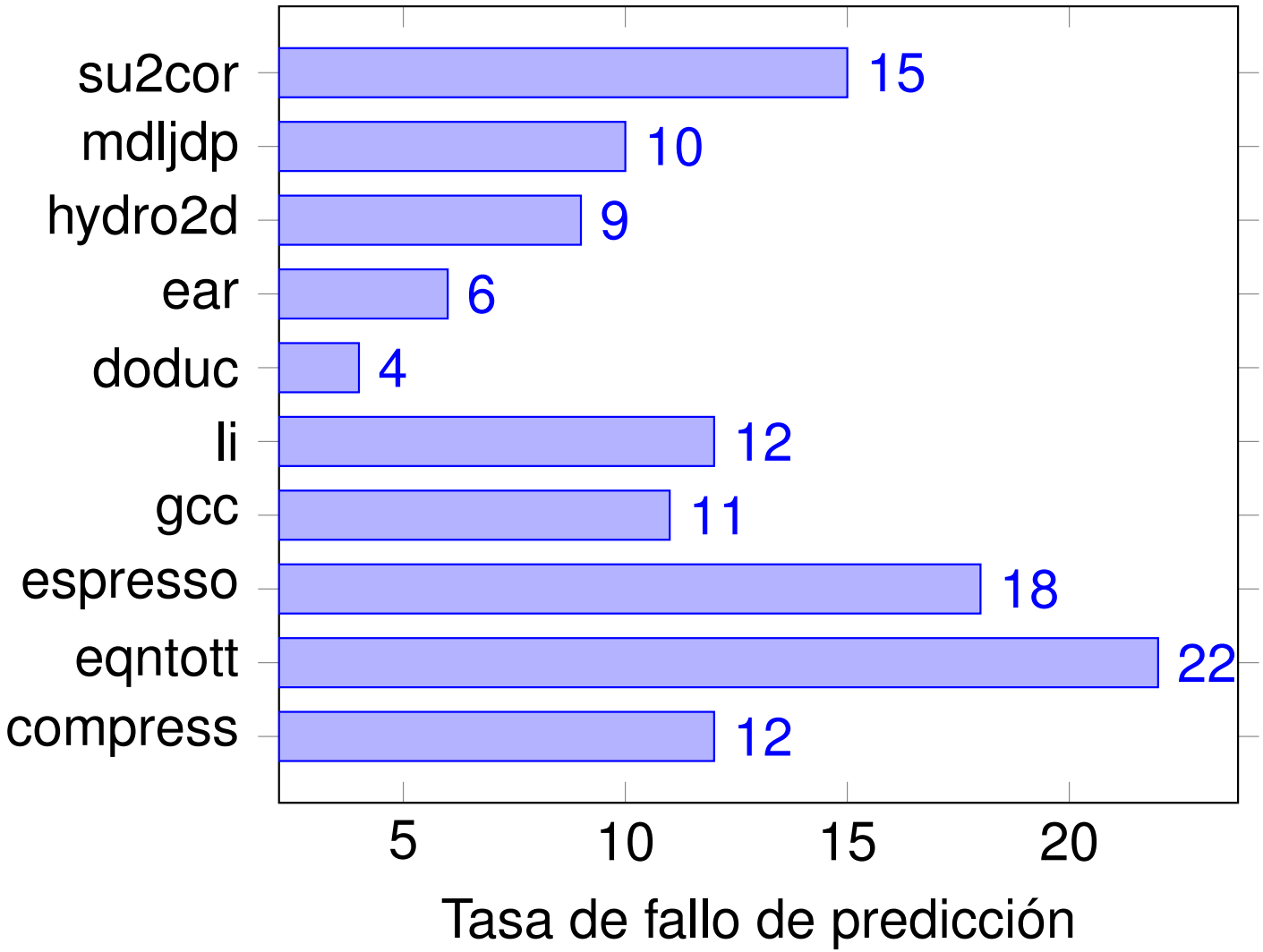
- Cada bifurcación condicional está **fuertemente sesgada**.
  - O se toma la mayoría de las veces,
  - O no se toma la mayoría de las veces.

- **Predicción basada en perfil de ejecución**:
  - Se ejecuta una vez para recoger estadísticas.
  - Se utiliza la información recogida para modificar el código y aprovechar la información.

Predicciones con perfil de ejecución

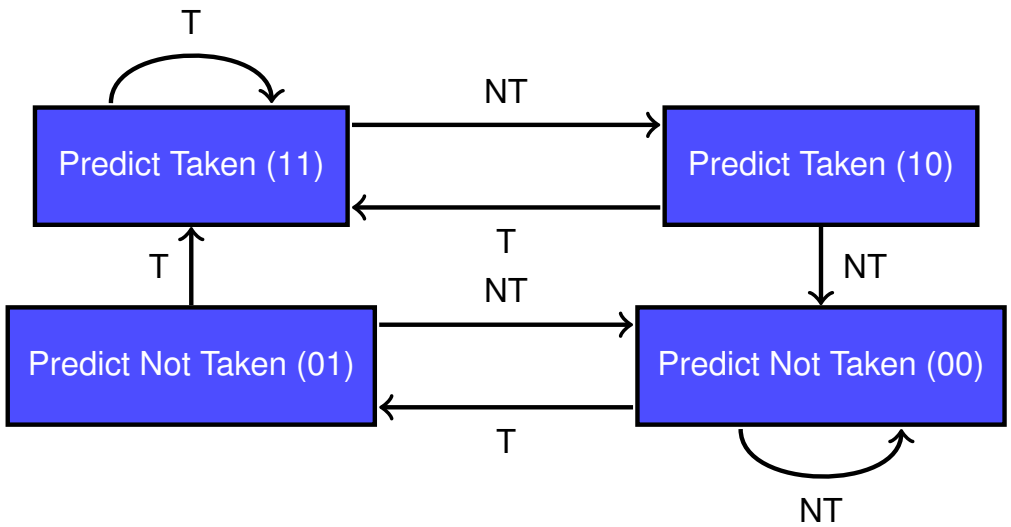
- SPEC92: Frecuencia de bifurcaciones 3% a 24%
- **Coma flotante**:
  - **Fallo de predicción**.
    - **Media**: 9%.
    - **Desviación estándar**: 4%.
- **Enteros**:
  - **Fallo de predicción**.
    - **Media**: 15%.
    - **Desviación estándar**: 5%.

Predicciones con perfil de ejecución



Predicción dinámica: BHT

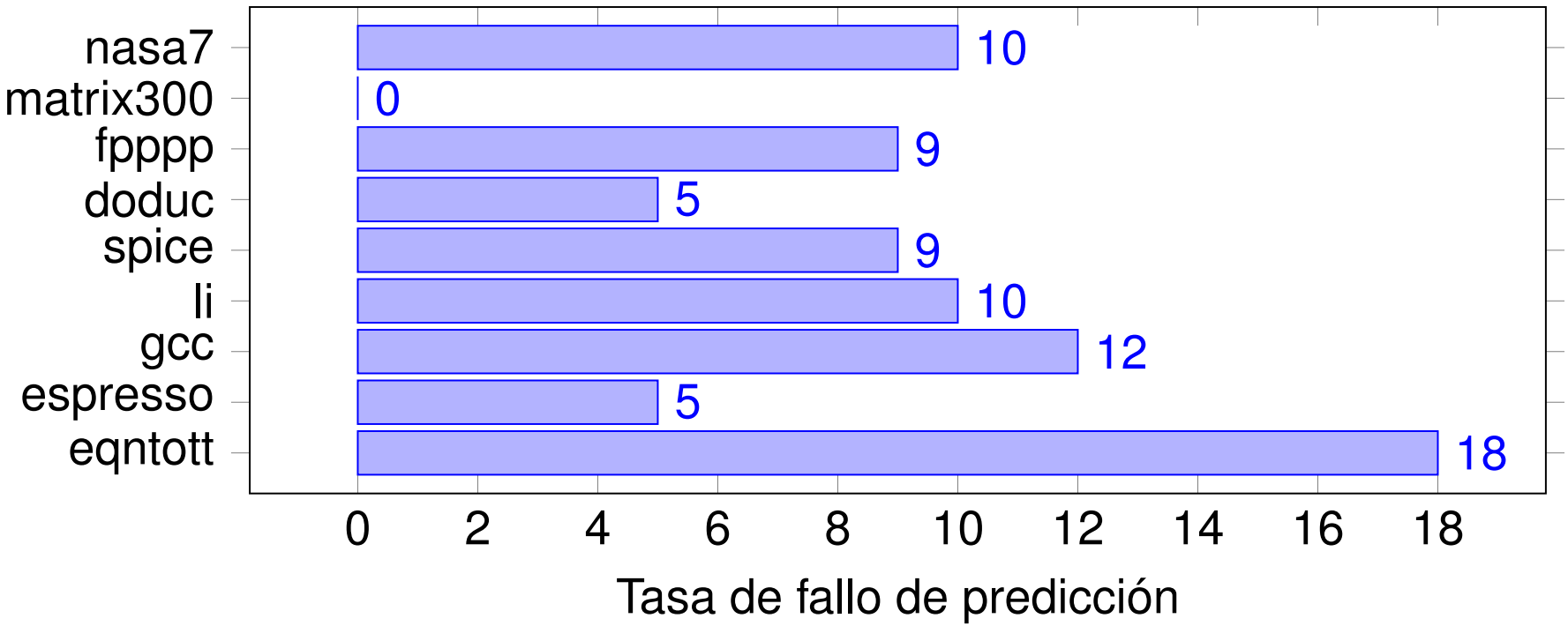
- Tabla histórica de saltos (Branch History Table)
  - **Índice**: Bits menos significativos de dirección (PC).
  - **Valor**: 1 bit (salto tomado o no la última vez).



- **Mejoras**: Uso de más bits para mejorar la precisión.

BHT: Precisión

- Fallos de predicción:
  - Predicción errónea para el salto.
  - Historia de otro salto en la entrada de la tabla.
- Resultados BHT de 2 bits y 4K entradas:



Predicción dinámica de bifurcación

- ¿Por qué funciona la predicción de saltos?
  - El algoritmo presenta regularidades.
  - Las estructuras de datos presenta regularidades.

- ¿Es mejor la predicción dinámica que la predicción estática?
  - Parecer serlo.
  - Hay un pequeño número de bifurcaciones importantes en programas con comportamiento dinámico.

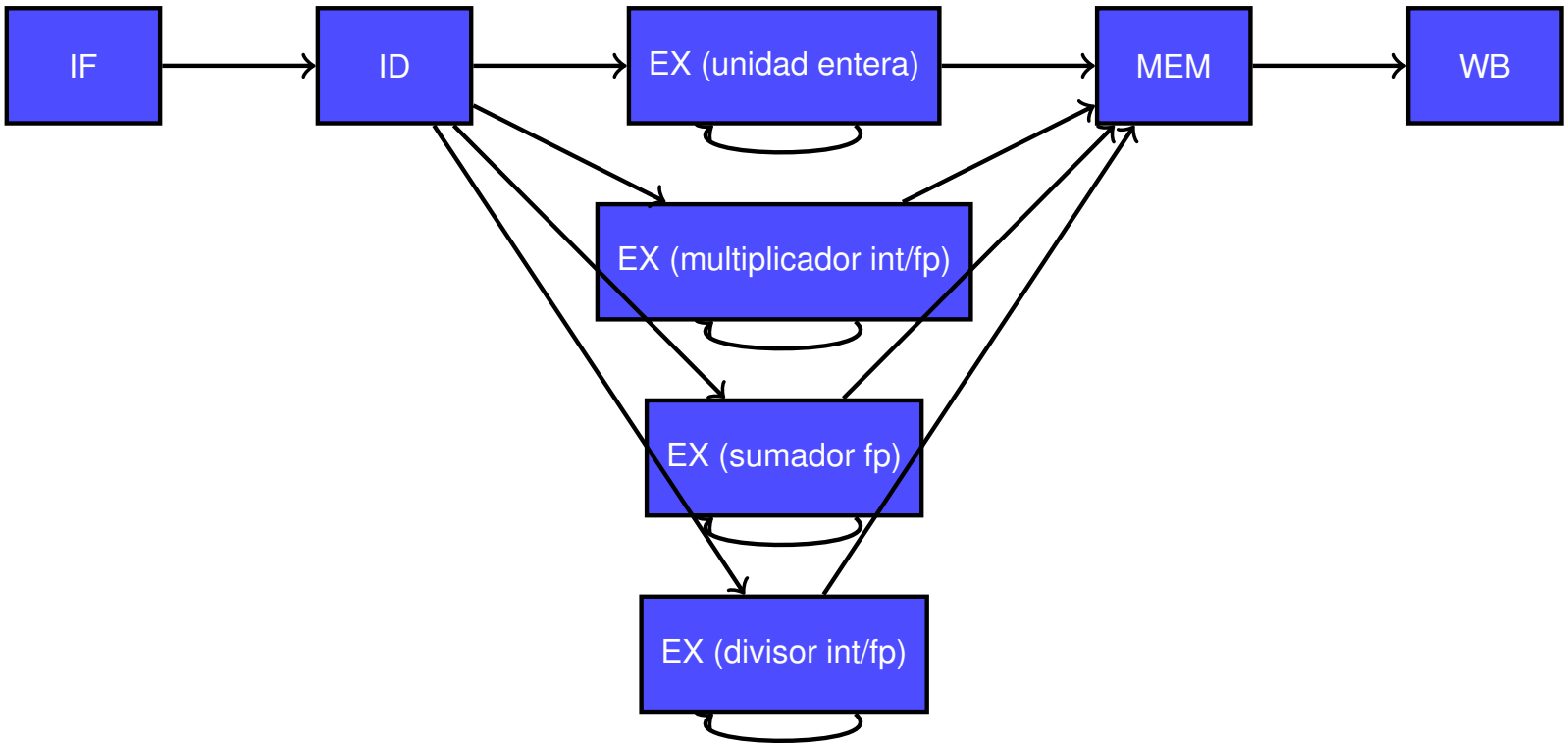
Operaciones de coma flotante

- ¿Operaciones de coma flotante en un ciclo?
  - Tener un ciclo de reloj extremadamente largo.
    - Impacto en rendimiento global.
  - Lógica de control de FPU muy compleja.
    - Consumo excesivo de recursos de diseño.

- **Alternativa**: Segmentación de coma flotante.
  - La etapa de ejecución se puede repetir varias veces.
  - Múltiples unidades funcionales en EX.
    - Ejemplo: Unidad entera, multiplicador FP y entero, sumador FP, divisor FP y entero.

Segmentación con coma flotante

- La etapa EX pasa a tener una duración mayor que 1 ciclo de reloj.



Latencia e intervalo de iniciación

- **Latencia**: Número de ciclos entre la instrucción que produce un resultado y la instrucción que usa ese resultado.
- **Intervalo de iniciación**: Número de ciclos entre la emisión de dos instrucciones que usan la misma unidad funcional.

Operación	Latencia	Intervalo iniciación
ALU entera	0	1
Instrucciones *load*	1	1
Suma FP	3	1
Multiplicación FP	6	1
División FP	24	25

Resumen

- Una arquitectura segmentada requiere mayor ancho de banda de memoria.
- Los riesgos en el pipeline provocan detenciones.
  - Degradación del rendimiento.
- Las detenciones de riesgos de datos se pueden mitigar con técnicas de compilación.
- Las detenciones de riesgos de control se pueden reducir con:
  - Alternativas en tiempo de compilación.
  - Alternativas en tiempo de ejecución.
- Las operaciones multi-ciclo permiten mantener ciclos de reloj cortos.

Referencias

- **Computer Architecture. A Quantitative Approach**
  5th Ed.
  Hennessy and Patterson.
  Secciones C.1, C.2 y C.5

- **Ejercicios recomendados**:
  - C.1, C.2, C.3, C.4 y C.5.

Introducción a paralelismo a nivel de instrucción

Arquitectura de Computadores

J. Daniel García Sánchez (coordinador)
David Expósito Singh
Javier García Blas
Óscar Pérez Alonso
J. Manuel Pérez Lobato

Grupo ARCOS
Departamento de Informática
Universidad Carlos III de Madrid