Tema 2: Ejecución segmentada de instrucciones

Parte 1

Iñigo Perona Balda (CAS)
Nestor Garay (EUS) — Olatz Arbelaitz (CAS/EUS)

Universidad del País Vasco (UPV/EHU)
Grado en Ingeniería Informática
Arquitectura de Computadores

22 de septiembre de 2023

Índice de parte 1 + parte 2

- Segmentación lineal
- Modelos avanzados (introducción)
- Técnicas de compilación

Tabla de contenidos

Introducción: Tiempo de ejecución

- La tarea de un computador es siempre la misma: ejecutar las instrucciones máquina de los programas.
- Desde el punto de vista de la arquitectura el **objetivo**: disminuir tiempo de ejecución $\equiv$ **aumentar velocidad**.
  ¿Cómo?
- A ser posible, que se lleven a cabo varias acciones, **al mismo tiempo**
  - Procesador único: que se ejecuten varias instrucciones a la vez
    $\rightarrow$ **Segmentación** (pipelining)
  - Más de un procesador: diferentes partes del programa se ejecutan a la vez
    $\rightarrow$ Multiprocesadores

Introducción: Tiempo de ejecución

- Tiempo de ejecución de un programa (simplificando):

$$Tej = NI \times Tins = NI \times CPI \times Tciclo$$

  - NI: número de instrucciones
  - Tins: tiempo de ejecución medio de una instrucción
  - CPI: número medio de ciclos para una instrucción (CPI, cycles per instruction)

Introducción: Tiempo de ejecución

- En caso de computadores con un procesador ¿qué opciones hay para aumentar el rendimiento?

$$Tej = NI \times Tins = NI \times CPI \times Tciclo$$

  - La tecnología minimiza el Tciclo
  - NI se minimiza utilizando técnicas de compilación
  - El objetivo de la **segmentación es disminuir el CPI**, ejecutando varias instrucciones a la vez:

    ILP — Instruction Level Parallelism

Tabla de contenidos

Segmentación

En la ejecución de las instrucciones se repiten de forma continua una serie de fases:

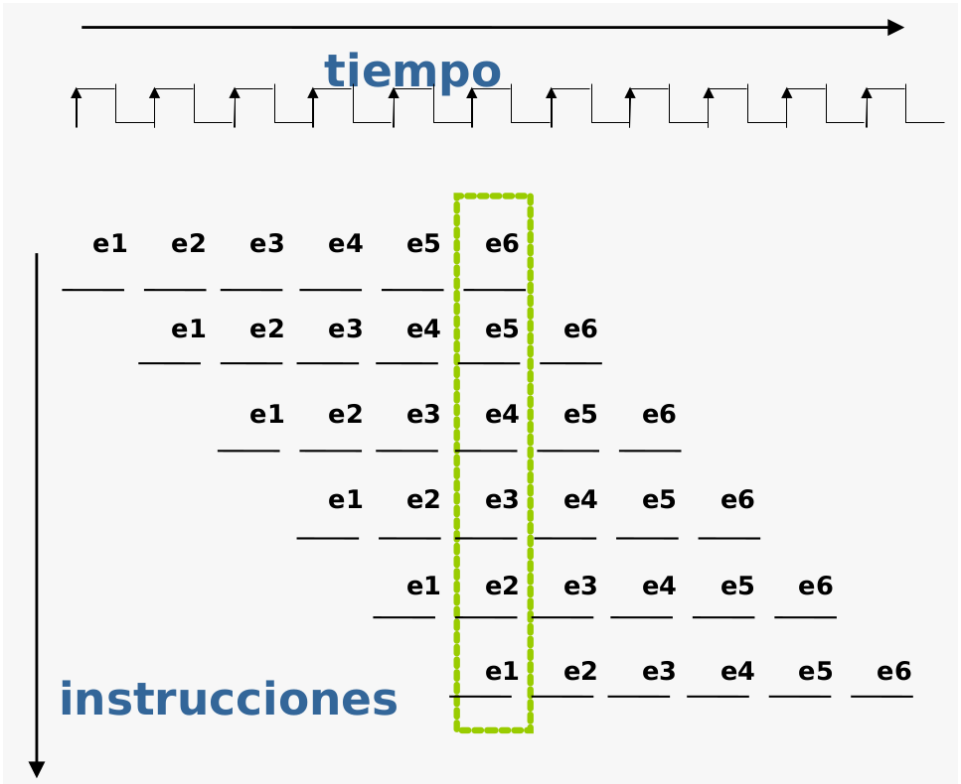
1. **búsqueda** de la instrucción (memoria)
2. **descodificación** de la instrucción (descodificador)
3. **lectura de operandos** (banco registros (BR) o memoria)
4. **ejecución** de la instrucción (ALU)
5. **escritura de resultados** (BR o memoria),
6. **actualización del PC** (sumador)

$\Uparrow$ ¡¡vuelta a empezar!!

En general, **el hardware que se utiliza en una fase, normalmente, no se utiliza en las otras.** Por tanto, si el hardware está libre, ¿se podría utilizar para ejecutar otras instrucciones?

Segmentación

- ¿Por qué no organizar la ejecución de instrucciones como una "cadena de montaje" (pipeline)?
- Cuando una instrucción pasa de la primera "fase" a la segunda, empieza la ejecución de la siguiente, sin esperar a que se complete su ejecución.



Segmentación

- En un instante de tiempo tendremos más de una instrucción en ejecución, cada una de ellas en una fase o etapa diferente.
- Llamaremos **ciclo/ clock** al tiempo que tarda cada fase. Sin segmentación, una instrucción tardaría varios ciclos en ejecutar.
- **Objetivo: ejecutar una instrucción por ciclo**.
- ATENCIÓN: el objetivo de la segmentación no es mejorar el tiempo de ejecución individual de una instrucción, sino la **productividad o throughput** del sistema; esto es, el **número de instrucciones que se ejecutan en un intervalo de tiempo**.
- Aunque el objetivo es ejecutar una instrucción por ciclo, esto no quiere decir que para ejecutar una instrucción sea necesario un único ciclo.

Segmentación

- Si la ejecución de una instrucción se divide en k etapas:

$$Tej = NI \times Tins = NI \times Tins/k$$

$$Tej = NI \times CPI \times Tclock = NI \times 1 \times Tciclo$$

  - comparando con la ejecución en un procesador secuencial, la ejecución sería k veces más rápida (pero surgen algunos problemas).

Segmentación

Conclusión

- Procesador **segmentado** (pipelining): divide la ejecución de las instrucciones en "segmentos" o etapas. Para conseguir este objetivo son más adecuadas las arquitecturas **RISC** (Reduced Instruction Set Computer).
  - Instrucciones **Load/ Store** de acceso a memoria, con modos de direccionamiento sencillos
  - Operandos y resultados de operaciones aritmético/ lógicas en el **banco de registros**
- Bibliografía básica: J. L. Hennessy, D. A. Patterson. Computer Architecture: A quantitative Approach.
  - Premio Fronteras del Conocimiento a John Hennessy y David Patterson por transformar en ciencia la arquitectura de ordenadores y diseñar los procesadores de los dispositivos actuales[1]

[1] `https://www.premiosfronterasdelconocimiento.es/version/edicion_2020/`

Tabla de contenidos

Segmentación lineal

Etapas o fases

- Hay varias alternativas para segmentar instrucciones. En el caso más sencillo, la **segmentación lineal, todas las instrucciones se tratan igual, sin tener en cuenta lo que realiza cada una**.
  - Así fueron los primeros procesadores segmentados (Hennessy & Patterson, MIPS) y usaremos esta alternativa para analizar los conceptos principales.
  - Los procesadores actuales son más complejos; se verán en la asignatura PAR en 3º

Segmentación lineal

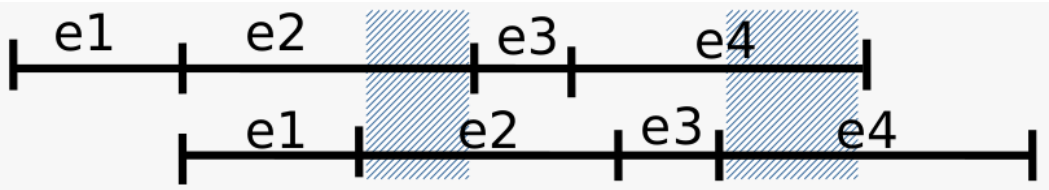
Condiciones para la segmentación lineal

- Para que la segmentación lineal sea eficiente
  - Todas las **etapas** deben tener una **duración similar**.
  - La etapa más lenta marcará el tiempo de **ciclo**.
  - Todas las instrucciones pasan **por todas las etapas**, aunque a veces no se haga nada en esos momentos.
- Así se asegura que nunca habrá dos instrucciones en la misma etapa.

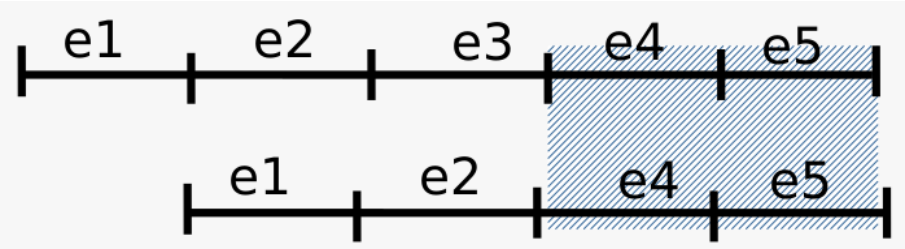
Segmentación lineal

Condiciones para la segmentación lineal

- Todas las fases del **mismo tiempo**, en caso contrario...



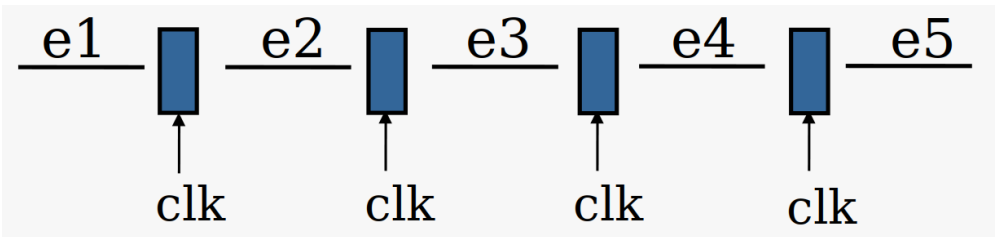
- Todas las instrucciones pasan por **todas las fases**, en caso contrario...



Segmentación lineal

Comunicación entre etapas

- Cuando la instrucción i pasa de la etapa e a la etapa $e+1$, la instrucción $i+1$ pasará de la etapa $e-1$ a la etapa e hay que guardar el trabajo parcial realizado en la etapa e; es necesaria una **comunicación entre etapas**, para que el "resultado" de una etapa (e) sea información de entrada en la siguiente ($e+1$)
- Para ello, las etapas se independizan ("aíslan") mediante registros especiales: **latches o registros de segmentación**
- Al acabar el ciclo y llegar el flanco del reloj, la información de cada instrucción y etapa **se cargan** simultáneamente **en los latches**



Segmentación lineal

Tiempo de ejecución

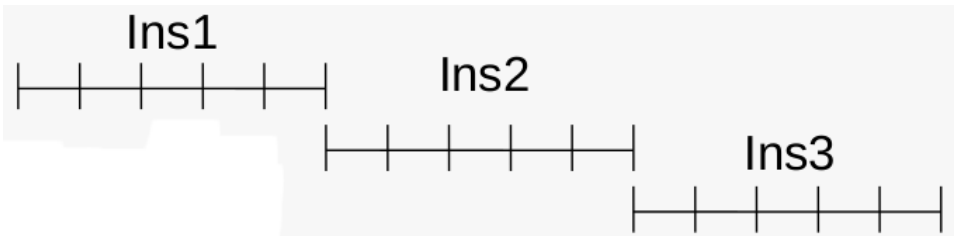
- **Registros de segmentación**
  - Los registros de segmentación añadirán algo de tiempo en la ejecución de la instrucción.

$$T_{ciclo} = T_{etapa} + T_{latch}$$

- **Ejecución secuencial**

$$Tsec = NI \times T$$

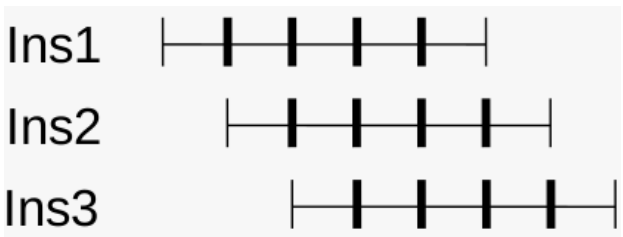
  - T, tiempo de ejecución medio de una instrucción



Segmentación lineal

Tiempo de ejecución

- **Ejecución segmentada**
  - Tiempo de ejecución de un programa cuando se segmenta una instrucción en k etapas.
  - $T_{programa} = \text{ciclos\_iniciales} \times T_{ciclo} + \\ + 1\ (\text{CPI}) \times T_{ciclo} \times NI$
  - $Tseg = (k-1) \times (t+w) + \\ (t+w) \times NI$
    - t, tiempo de una etapa de segmentación
    - w, tiempo de carga de un latch
    - k, número de etapas



Segmentación lineal

Tiempo de ejecución: Ejemplo

- En un procesador se han identificado 5 etapas, con las siguientes duraciones (ns):

$$[6] + [2, 8] + [4, 2] + [5] + [6]$$

  Si se segmenta, el tiempo de carga de los latches es de 0,5 ns.
- Calcula cuántas veces más rápido es el procesador segmentado con respecto del secuencial.
  - Secuencial (1 instrucción):
    6 + 2,8 + 4,2 + 5 + 6 = **24 ns**
  - Segmentado:
    - Tciclo = 6 ns ([¡ojo! la más larga]) + 0,5 ns (latches) = **6,5 ns**
    - Por tanto, 1 instrucción cada 6,5 ns → **3,7 veces más rápido**
      [aunque cada instrucción, todas iguales, de manera secuencial, necesiten 6,5 x 5 = 32,5 ns!]

Segmentación lineal

Problemas principales

- **Dependencias estructurales**. ¿Hay suficiente hardware para ejecutar todas las instrucciones en paralelo?
- **Dependencias de datos**. ¿Cómo ejecutamos **la instrucción** $i+1$ si ésta necesita algún dato que calculará **la instrucción** i y que todavía no ha acabado?
- **Dependencias de control**. ¿Qué instrucción ejecutamos tras un salto mientras se determina **si se salta o no**?

Segmentación lineal

Problemas principales: Dependencias estructurales

- Dos instrucciones necesitan el mismo recurso, en el mismo momento. Por ejemplo:
  - Etapa 4 de la instrucción 1: instrucción `Load (LD)` — lectura de memoria, dato
  - Etapa 1 de la instrucción 4: búsqueda de la instrucción — lectura de memoria

Instr. 1 (LD):	Mem	Des/rBR	ALU	Mem	wBR					
Instrucción 2:		Mem	Des/rBR	ALU	Mem	wBR				
Instrucción 3:			Mem	Des/rBR	ALU	Mem	wBR			
Instrucción 4:				Mem	Des/rBR	ALU	Mem	wBR		

- Segmentación para los ejemplos en 5 etapas: búsqueda de la instrucción; descodificar/lectura de operandos; realizar cálculos; acceso a memoria (rd/wr); y escritura de resultados en el banco de registros.

Segmentación lineal

Problemas principales: Dependencias de datos

- El dato generado por una instrucción lo utiliza la siguiente instrucción. Por ejemplo:

  ADD **R1**, R2, R3
  ADD R4, **R1**, R5

R1 = R2 + R3	Mem	Des/rBR	ALU	Mem	**wBR (R1)**	
R4 = **R1** + R5		Mem	**Des/rBR (R1)**	ALU	Mem	wBR

Segmentación lineal

Problemas principales: Dependencias de control

- Las instrucciones de salto rompen la secuencia de ejecución. Por

  tanto, no se sabe cual será la siguiente instrucción. Por ejemplo:

i BRx **k**	Mem	**Des/rBR**[2]	ALU	Mem	wBR	
instr. i+1?		Mem	Des/rBR	ALU	Mem	wBR
...						
instr. k?			**Mem**[3]	Des/rBR	ALU	Mem	wBR

[2] Aquí se decide realizar el salto: **pc** $\rightarrow$ **k**

[3] La siguiente instrucción no debía ser i+1, sino que **k**. **La instrucción i+1, ¡no debía ejecutarse!**

Tabla de contenidos

Procesador DLX

Características generales

- Para analizar los problemas de segmentación, se utilizará un procesador lineal simple de tipo **RISC (Reduced Instruction Set Computer)**. RISC es un subconjunto del **procesador DLX**. DLX está diseñado por Hennessy/ Patterson.
  - Instrucciones específicas de acceso a memoria: `LD` (M $\rightarrow$ Ri) y `ST` (Ri $\rightarrow$ M).
    Direccionamiento a nivel de byte.
  - Operaciones siempre entre registros generales; 32 registros generales en el banco de registros (BR);
  - Formato de la instrucción:
    - Instrucciones de 32 bits, código de operación de 6 bits
    - Direcciones de registros de 5 bits. Datos inmediatos de 16 bits,
    - Desplazamientos en los saltos condicionales de 16 bits

Procesador DLX

Conjunto de instrucciones

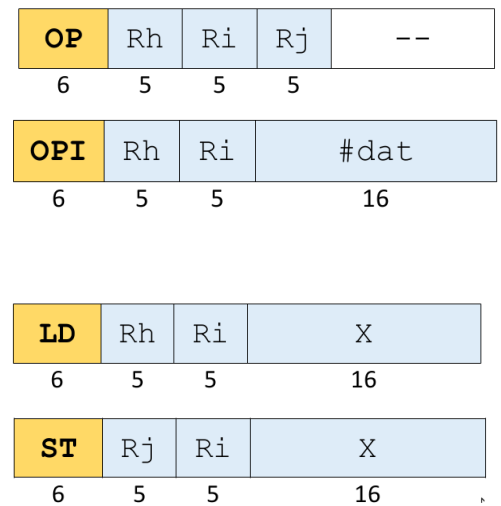
- Nomenclatura:
  - Rd (Registro Destino);
  - Rf1, Rf2 (Registro Fuente/ origen);
- OP: add, sub, mul, div, or, and...:

  OP Rd, Rf1, Rf2 Rd = Rf1 OP Rf2

  OPI Rd, Rf1, #dat Rd = Rf1 OP dat
- Instrucciones acceso a memoria:

  LD Rd, desp(Rf1) Rd = M(Rf1 + desp)

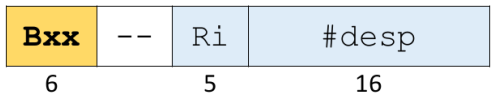
  ST desp(Rf1), Rf2 M(Rf1 + desp) = Rf2



Conjunto de instrucciones

- Saltos condicionales (Bxx): BZ, BNZ, BPOS, BNEG

  Bxx Rf1, desp if (Rf1 cmp 0) then pc=pc+desp

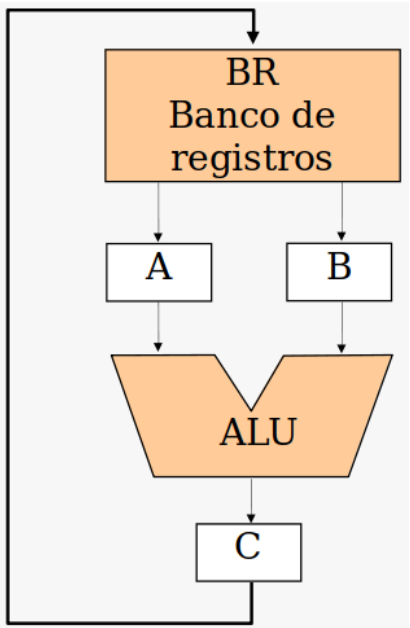


- Otros: JMP, CALL, RET, TRAP, RETI...
- Modos de direccionamiento simples: directo (Ri), inmediato (#dat), relativo (Ri+desp), absoluto (@=variable)

Procesador DLX

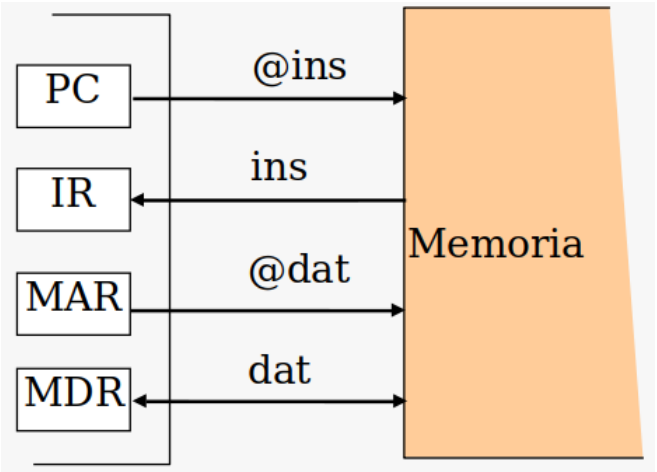
Ruta de datos

Parte aritmética



- ALU: Arithmetic Logic Unit

Comunicación con memoria



- PC: Program Counter (dirección de la instrucción)
- IR: Instruction Register (instrucción)
- MAR: Memory Address Register
- MDR: Memory Data Register

Etapas de ejecución

- **Búsqueda (B)**: búsqueda de la instrucción en memoria. En ingles: fetch.
- **Descodificación y lectura (D/L)**: descodificar la instrucción y leer los operandos del banco de registros
- **Unidad aritmética/ lógica (A)**: realización de cálculos en la ALU (cálculo de la @ de memoria (X+Ri) u operaciones aritméticos).
- **Memoria (M)**: lectura de datos de memoria (load) o escritura de datos en memoria (store).
- **Escritura de resultados (E)**: escritura de resultados en el banco de registros.

Procesador DLX

Tabla de reserva

Etapas:	1: B	2: D/L	3: A	4: M	5: E
OP Rd,Rf1,Rf2 rd = rf1 op rf2	ir=M(pc) pc=pc+4	descodificar A=BR[rf1] B=BR[rf2]	C=A op B		BR[rd]=C
LD Rd,d(Rf1) rd = M(d+rf1)	ir=M(pc) pc=pc+4	descodificar A=BR[rf1] B=d	mar=A+B	mdr=M(mar)	BR[rd]=mdr
ST d(Rf1),Rf2 M(rf1+d) = rf2	ir=M(pc) pc=pc+4	descodificar A=BR[rf1] B=d mdr=BR[rf2]	mar=A+B	M(mar)=mdr	
Bxx Rf1,d if (rf1 cmp 0) pc=pc+d	ir=M(pc) pc=pc+4	descodificar if(BR[rf1] cmp 0) pc = pc + d			

- OPI: igual que la OP, pero en la etapa 2 hay que cargar en el latch B el valor inmediato (valor en la misma instrucción) que está en el IR, tal y como se hace en el LOAD (B=d)

Tabla de contenidos

Procesador DLX: Dependencias estructurales

- Una vez decidida la tabla de reserva, hay que analizar si ocurren **conflictos de recursos**: que dos o más instrucciones necesiten a la vez el mismo recurso $\rightarrow$ **dependencias estructurales**
- **Problemas principales**:
  - utilizar el mismo recurso (memoria, ALU, sumadores...) en distintas etapas
  - escribir en el mismo latch en distintas etapas
  - se utiliza el resultado conseguido en una etapa, pero dos, tres... etapas después
- En general, las dependencias estructurales se resuelven añadiendo **más hardware**, por ejemplo en el caso del DLX:

Procesador DLX: Dependencias estructurales

Tabla de reserva: resolviendo dependencias estructurales

Etapas:	1: B	2: D/L	3: A	4: M	5: E
OP Rd,Rf1,Rf2 rd = rf1 op rf2	ir=MI(pc) pc=pc+4 pc1=pc	descodificar A=BR[rf1] [1] B=BR[rf2] [1] ir2=ir	mar=A op B ir3=ir2	mdr2=mar [3] ir4=ir3	BR[rd]=mdr2 [1]
LD Rd,d(Rf1) rd = M(d+rf1)	ir=MI(pc) pc=pc+4 pc1=pc	descodificar A=BR[rf1] B=d ir2=ir	mar=A+B ir3=ir2	mdr=MD(mar) ir4=ir3	BR[rd]=mdr
ST d(Rf1),Rf2 M(rf1+d) = rf2	ir=MI(pc) pc=pc+4 pc1=pc	descodificar A=BR[rf1] B=d mdr0=BR[rf2] [3] ir2=ir	mar=A+B mdr1=mdr0 [3] ir3=ir2	MD(mar)=mdr1 ir4=ir3	
Bxx Rf1,d if (rf1 cmp 0) pc=pc+d	ir=MI(pc) pc=pc+4 [2] pc1=pc	descodificar if(BR[rf1] cmp 0) pc=pc1+d [2] ir2=ir			

- M $\rightarrow$ MI: Cache de instrucciones; M $\rightarrow$ MD: Cache de datos;
- [1]: en el BR, 2 rd + 1 wr a la vez
- [2]: (1) 2 sumadores + 1 comparador a la vez. (2) pc se sobrescribe en dos etapas.
- [3]: para minimizar dependencias, la etapa i solamente puede cargar valores de la etapa $i-1$ $\rightarrow$ copiar el valor para hacerlo accesible a la siguiente etapa.

Procesador DLX: Dependencias estructurales

Resumen

- **Memoria**: caches separadas (instrucciones / datos) para obtener instrucciones y datos al mismo tiempo
- **Más recursos**:
  - B: sumador (pc+4)
  - D/L: el BR necesita dos puertos lectura y una de escritura (en la última etapa). Comparador (Ri cmp 0) y sumador (pc+d).
- Añadir **latches** (y minimizar la cantidad de ellos)
  - No se puede leer en la etapa $i+2$ un latch cargado en la etapa i, porque ya ha sido modificado.
  - No se puede cargar el mismo latch en distintas etapas. A excepción del registro pc (uso de multiplexor), ya que se necesita para realizar saltos.
  - Copiar IR en todas las etapas (IR, IR2, IR3 e IR4) ya que cada etapa lo sobrescribe.

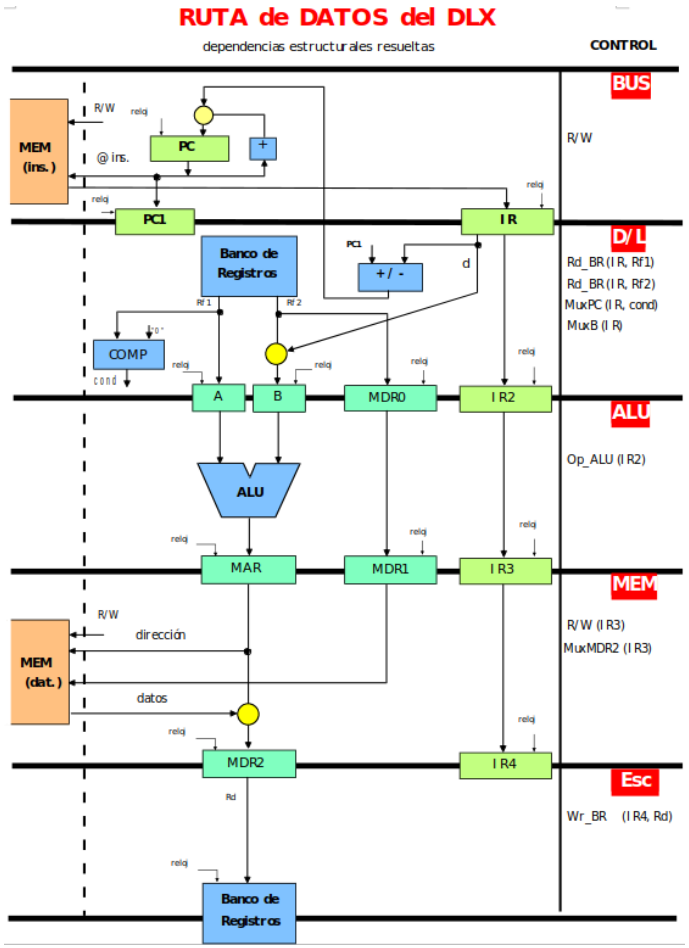
Procesador DLX: Dependencias estructurales

Tabla de reserva: dependencias estructurales resueltas

Etapas:	1: B	2: D/L	3: A	4: M	5: E
OP Rd,Rf1,Rf2 rd = rf1 op rf2	ir=MI(pc) pc=pc+4 (pc1=pc)	descodificar A=BR[rf1] B=BR[rf2] ir2=ir	mar=A op B ir3=ir2	mdr2=mar ir4=ir3	BR[rd]=mdr2
LD Rd,d(Rf1) rd = M(d+rf1)	ir=MI(pc) pc=pc+4 (pc1=pc)	descodificar A=BR[rf1] B=d ir2=ir	mar=A+B ir3=ir2	mdr=MD(mar) ir4=ir3	BR[rd]=mdr
ST d(Rf1),Rf2 M(rf1+d) = rf2	ir=MI(pc) pc=pc+4 (pc1=pc)	descodificar A=BR[rf1] B=d mdr0=BR[rf2] ir2=ir	mar=A+B mdr1=mdr0 ir3=ir2	MD(mar)=mdr1 ir4=ir3	
Bxx Rf1,d if (rf1 cmp 0) pc=pc+d	ir=MI(pc) pc=pc+4 (pc1=pc)	descodificar if(BR[rf1] cmp 0) pc=pc1+d (ir2=ir)			

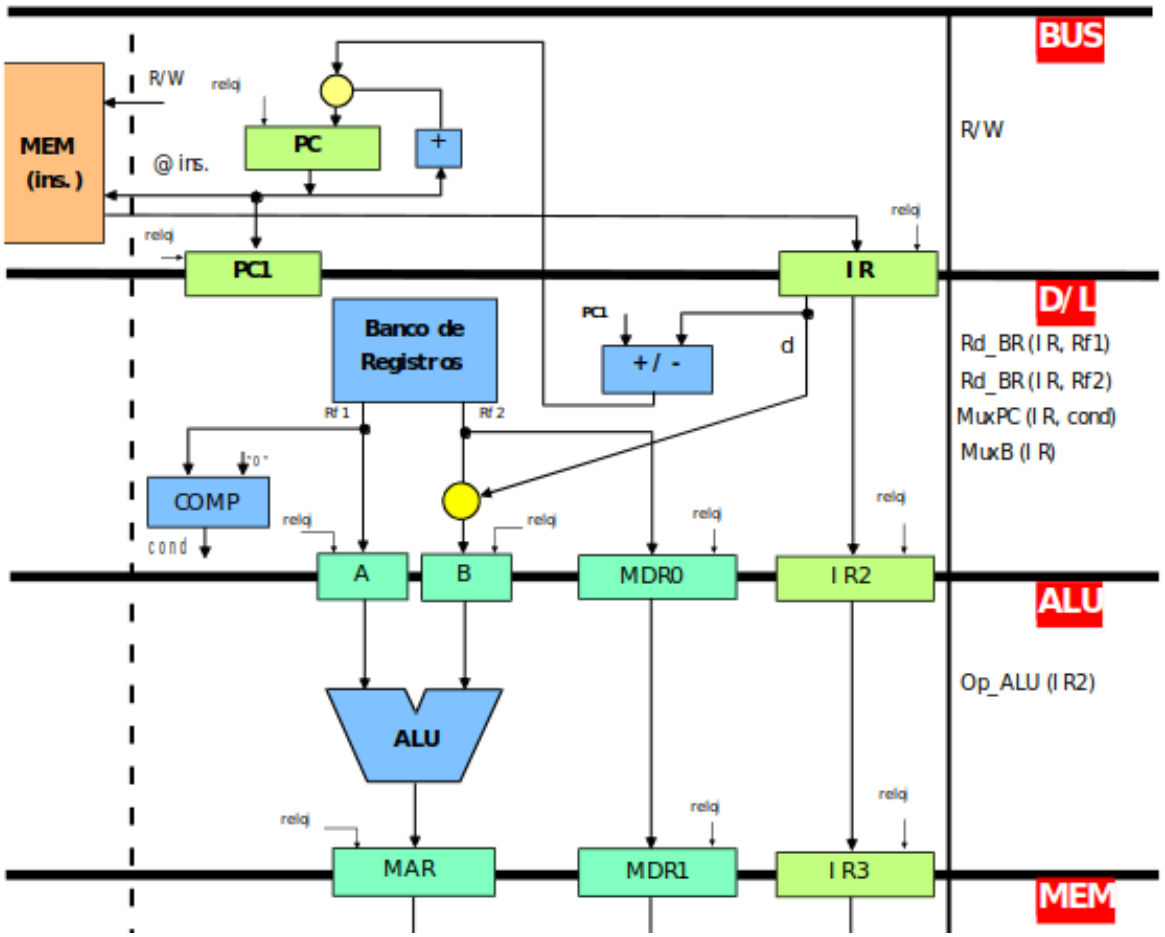
Procesador DLX: Dependencias estructurales

Ruta de datos del DLX: dependencias estructurales resueltas



Procesador DLX: Dependencias estructurales

Ruta de datos del DLX: dependencias estructurales resueltas (1)



Procesador DLX: Dependencias estructurales

Ruta de datos del DLX: dependencias estructurales resueltas (2)

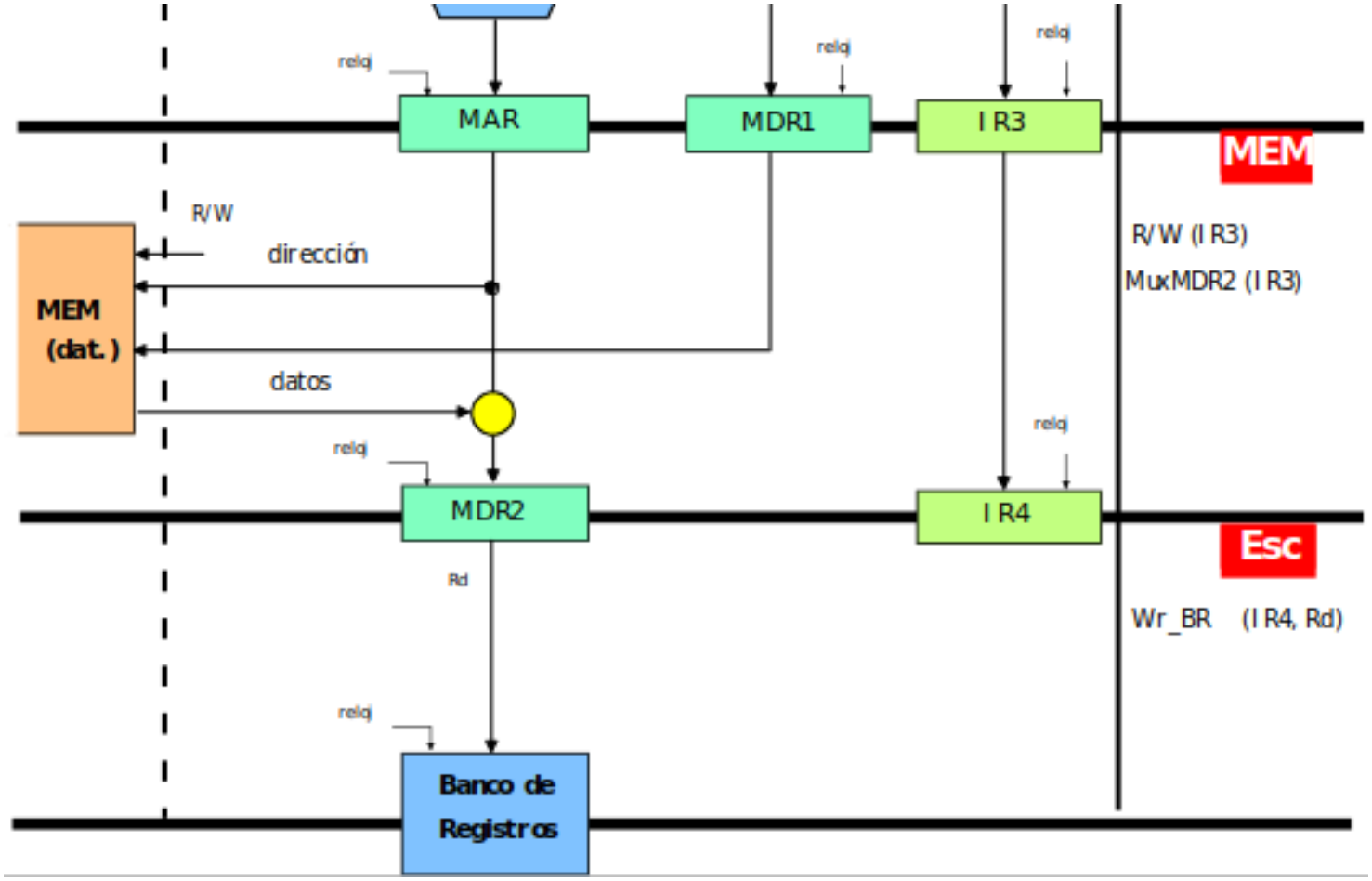


Tabla de contenidos

Procesador DLX: Dependencias de datos

Problema

- En un procesador segmentado, se quieren ejecutar las instrucciones en paralelo. A pesar de solucionar todos los problemas de hardware, muchas veces no va a ser posible, ya que el programa que se ejecuta no permite ese nivel de paralelismo (ILP).
- Por ejemplo, veamos el siguiente programa:

LD **R2**, d(R1)	B	D/L	A	M	**E**
ADDI R3, **R2**, #1		B	D/**L?**	–	–

  - ADDI no puede leer el registro R2 en su 2º ciclo. ¡El dato no está preparado!

Procesador DLX: Dependencias de datos

Clasificación

- Se dice que hay una **dependencia** entre dos instrucciones **si tienen que trabajar con los mismos registros o posiciones de memoria** (sólo analizaremos las dependencias entre registros)
- Ocurrirá la dependencia, o no, en función de los momentos en que se ejecuten las instrucciones (en algunos textos distinguen la propia dependencia y el "riesgo o hazard" del problema.
- Hay dos tipos de dependencias:
  - **Dependencias reales**, si hay una trasferencia de datos entre dos instrucciones
  - **Dependencias nominales**, relacionadas con la "escritura" del programa, por reutilizar los registros (un procesador tiene pocos registros)

Procesador DLX: Dependencias de datos

Clasificación de las dependencias

1. **Dependencia real, RAW (read-after-write)**: Una instrucción quiere leer un dato generado por una instrucción previa. Hay una trasferencia de datos de una instrucción a otra.

   ADD **R2**, R4, R5
   SUBI R6, **R2**, #1

   - Es **la dependencia más problemática**, con consecuencias en cualquier programa. Si no se consigue alguna solución, estaremos muy lejos del objetivo perseguido: ejecutar 1 instrucción por ciclo.
   - Sí o sí, habría que esperar a que la instrucción anterior escribiese en el BR, y después leer. Por tanto, se "perderían" esos ciclos al no hacer nada entretanto.

Procesador DLX: Dependencias de datos

Clasificación de las dependencias

2a **Anti-dependencia, WAR (write-after-read)**: Una instrucción quiere escribir un registro que otra instrucción previa quiere leer y cambiar.

ADD R1, R4, **R2**
SUBI **R2**, R6, #1

- Hay que asegurar que ADD lea R2 antes de que SUBI escriba el resultado en R2, de lo contrario cambiaría la semántica del programa.
- En el procesador DLX (segmentada lineal) todas las lecturas se realizan antes que las escrituras, por lo que nunca ocurre este problema.

Procesador DLX: Dependencias de datos

Clasificación de las dependencias

2b **Dependencia de salida, WAW (write-after-write)**: Dos instrucciones quieren escribir el mismo registro.

ADD **R2**, R4, R1
SUBI **R2**, R6, #1

- Hay que asegurar que las escrituras se hacen en orden: SUBI actualiza R2 a posteriori de ADD, de lo contrario cambiaría la semántica del programa.
- En el DLX todas las escrituras se realizan en la misma etapa, por lo que no hay este problema.

Procesador DLX: Dependencias de datos

Solucionar dependencias WAR y WAW

Aunque en el DLX no se dan, estas dependencias son fáciles de solucionar.

- WAR: **retrasar la escritura** hasta que la instrucción previa haya leído el dato
- WAW: atrasar o **anular la escritura** de la primera instrucción
- [En la asignatura PAR se analizarán más posibles soluciones]

Solucionar dependencias RAW: parada de la instrucción dependiente

- No hay solución. Sí o sí hay que esperar hasta la instrucción anterior escriba en el BR, y luego leerlo. Se van a perder esos ciclos de ejecución sin hacer nada (esperando).
- La instrucción SUBI debe esperar a que ADD escriba el resultado en el BR. En este momento, la lectura desde el BR será correcta.

Tabla de contenidos

Procesador DLX: Solucionar dependencias RAW

Parada de la instrucción dependiente

- Hay dos posibilidades para hacer la parada (o retrasar la ejecución segmentada):
  - **Parada por software (compilador)**: El compilador introduce instrucciones **NOP (Not Operation, no ejecutan acciones)** necesarias para distanciar las instrucciones el número de ciclos necesario. [¡Ojo!: el tamaño del programa y la necesidad de memoria crecen]
  - **Parada por hardware (Unidad de Control)**: La Unidad de Control (UC) detecta la dependencia al **descodificar y para la ejecución** de esa instrucción (y las siguientes) el número de ciclos necesario para solventarla (interlock). El tamaño del programa no crece pero la UC es más complejo.

Procesador DLX: Solucionar dependencias RAW

Parada de la instrucción dependiente

- **Parada por software (compilador, NOP)**

```
ADD R2,R4,R5    B   D/L   A    M    E
   NOP              B   D/L  A    M    E
   NOP                  B   D/L  A    M    E
   NOP                      B   D/L  A    M    E
SUBI R6,R2,#1                   B   D/L  A    M    E
```

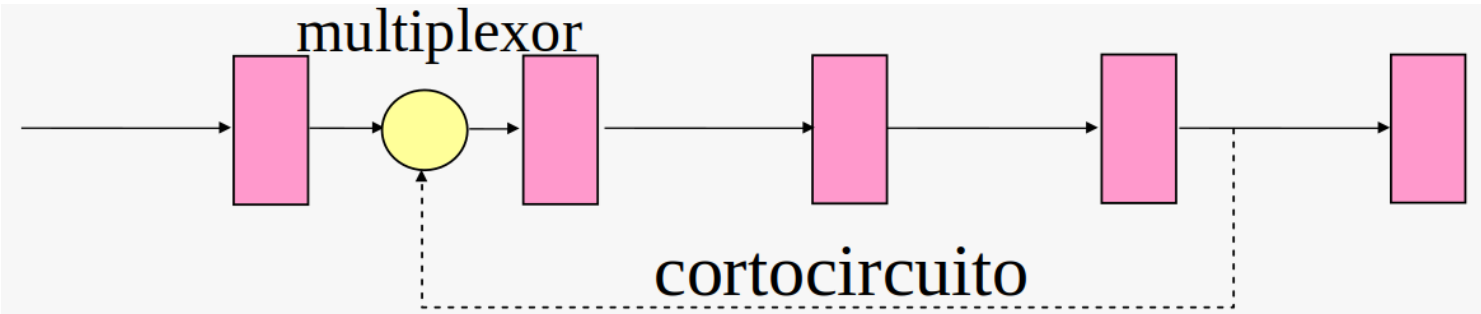
- **Parada por hardware (UC, burbuja/interlock)**

```
ADD R2,R4,R5    B   D/L   A    M    E
SUBI R6,R2,#1       B   D/L  D/L  D/L  D/L  A    M    E
```

Procesador DLX: Solucionar dependencias RAW

Cortocircuitos (forwarding)

- Objetivo: minimizar paradas
- En procesadores DLX, los resultados se escriben en el BR en la última fase, **pero puede que estén calculados antes**. Por ejemplo, en las operaciones aritméticas se calculan en la fase 3 (ALU).
- Por tanto, se puede obtener el dato desde la Unidad Funcional (UF) que lo calcula, sin necesidad de esperar a leerlo desde el BR una vez escrito.
- Se dice que se hace un cortocircuito si se pasan hacia atrás los datos de una fase a una anterior.



Procesador DLX: Solucionar dependencias RAW

Cortocircuitos (forwarding)

- Un cortocircuito supone un **bus**, un **multiplexor** y el **control** correspondiente por parte de la Unidad de Control. Además, hay que tener en cuenta el **tiempo** que supone la realización del cortocircuito: **¡no siempre es posible!**. En general, debe ocurrir:

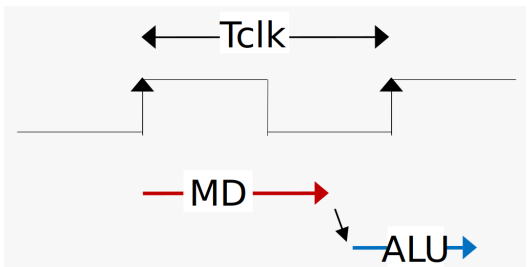
$$t_{UF} + t_{mux} + t_{latch} < t_{ciclo}$$

Procesador DLX: Solucionar dependencias RAW

Cortocircuitos (forwarding)

- Hay que analizar todos los pares de instrucciones para detectar las dependencias RAW entre ellas
- **Cortocircuitos posibles**:
  - S(UF) $\rightarrow$ E(latch) [S: Salida; E:Entrada]
  - S(latch) $\rightarrow$ E(UF)
  - S(latch) $\rightarrow$ E(latch)
- **Cortocircuitos problemáticos**:
  - S(UF) $\rightarrow$ E(UF)

  En general no dará tiempo para hacer ambas operaciones en un único ciclo. Ejemplo desde la memoria de datos (MD) al ALU:



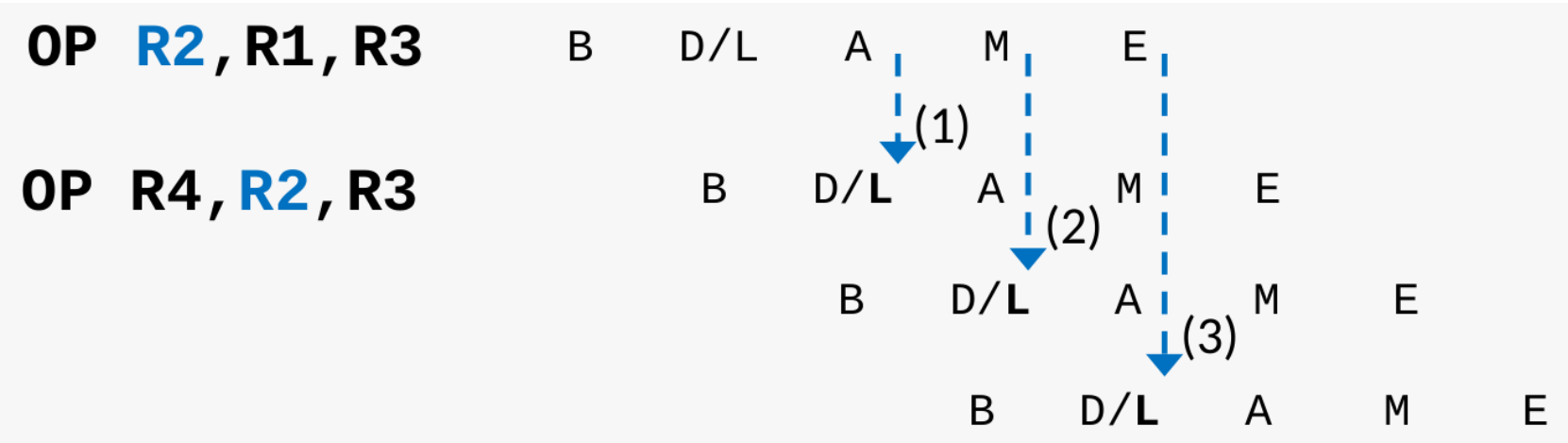
Procesador DLX: Solucionar dependencias RAW

Cortocircuitos (forwarding)

- Para saber cuántos cortocircuitos y cuáles hay que incluir, hay que analizar las posibles dependencias entre todas las parejas de instrucciones.
- Además, las dependencias pueden llegar a distancia 1, 2, 3...
- **No olvides**: al realizar un **cortocircuito**, lo que se consigue es importante $\rightarrow$ **no se para la ejecución de las instrucciones** o, lo que es lo mismo, ¡**no se pierden ciclos** y baja el tiempo de ejecución del programa!
- Vamos a analizar las posibles dependencias entre instrucciones del DLX en busca de cortocircuitos en función de la tabla de reserva que se ha definido.

Procesador DLX: Solucionar dependencias RAW

Cortocircuitos OP-OP

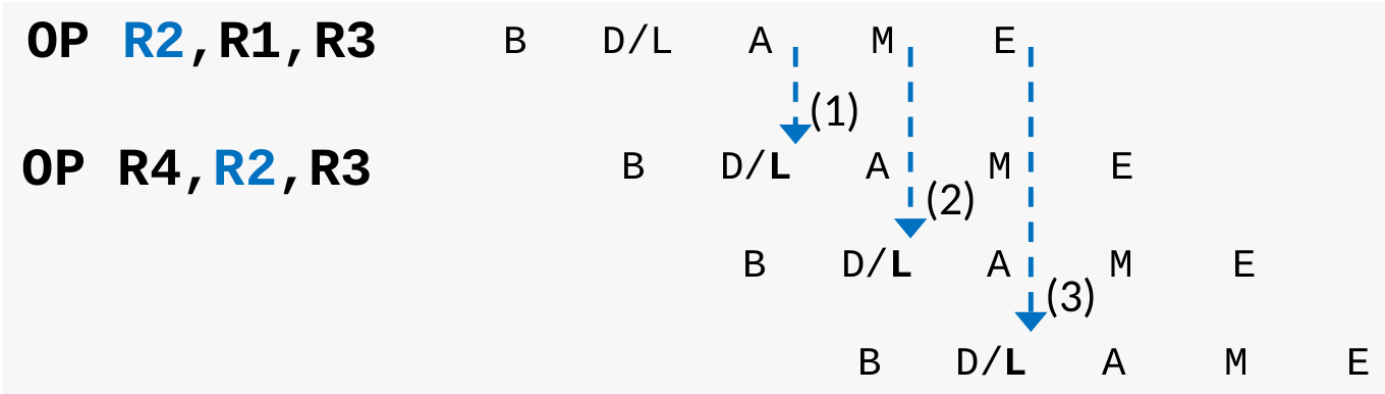


- Los resultados se calculan en la etapa 3 y se leen en la etapa 2. Hay que tener en cuenta las distancias:
  - Distancia 1: S(ALU) $\rightarrow$ E(A/B) [S: Salida; E:Entrada]
  - Distancia 2: S(mar) $\rightarrow$ E(A/B)
  - Distancia 3: S(mdr2) $\rightarrow$ E(A/B)

Procesador DLX: Solucionar dependencias RAW

Cortocircuitos OP-OP

- El resultado de la instrucción OP se genera en la fase 3 y hay que leerlo en la fase 2. Hay que tener en cuenta la distancia entre instrucciones.

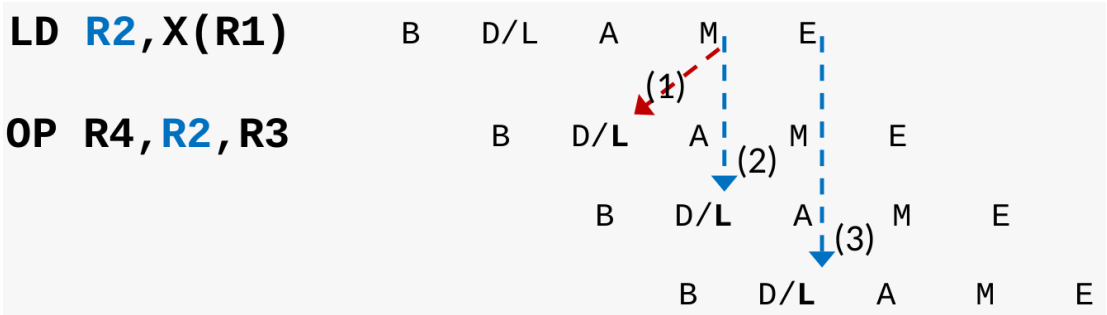


- Por tanto, se pueden hacer los siguientes cortocircuitos desde la salida (S) de un UF a la entrada (E) de otro UF:
  - Distancia 1: **S(ALU) → E(A/B)** (desde ALU a latch)
  - Distancia 2: **S(mar) → E(A/B)** (desde latch a latch)
  - Distancia 3: **S(mdr2) → E(A/B)** (desde latch a latch)

Procesador DLX: Solucionar dependencias RAW

Cortocircuitos LD-OP

- Los resultados se logran en la etapa 4 (LD) y hay que leerlos en la 2 (OP).



- A distancia 1, aunque el dato no está disponible en la etapa D/L de OP, podríamos llevarlo a la etapa A, directamente a la ALU:

  S(MD) $\rightarrow$ E(ALU). ¡No hay tiempo!

- Estos son los cortocircuitos:
  - Distancia 1: **Parar un ciclo**
  - Distancia 2: **S(MD) $\rightarrow$ E(A/B)** (desde cache de datos a latch)
  - Distancia 3: **S(mdr2) $\rightarrow$ E(A/B)** (desde latch a latch)

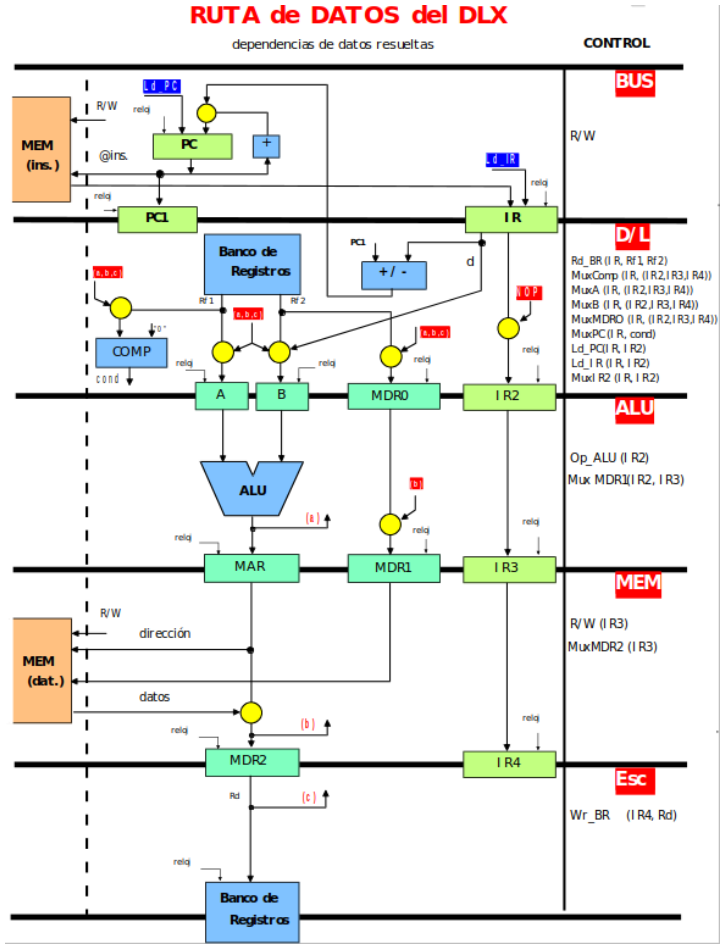
Procesador DLX: Solucionar dependencias RAW

Cortocircuitos DLX

Instrucción		**Distancia 1**	**Distancia 2**	**Distancia 3**
OP–OP		S(ALU) → E(A/B)	S(mar) → E(A/B)	S(mdr2) → E(A/B)
OP–LD	dir	S(ALU) → E(A)	S(mar) → E(A)	S(mdr2) → E(A)
OP–ST	dat	S(ALU) → E(mdr0)	S(mar) → E(mdr0)	S(mdr2) → E(mdr0)
	dir	S(ALU) → E(A)	S(mar) → E(A)	S(mdr2) → E(A)
OP–BR		S(ALU) → E(Cmp)	S(mar) → E(Cmp)	S(mdr2) → E(Cmp)
LD–OP		**PARAR**	S(MD) → E(A/B)	S(mdr2) → E(A/B)
LD–LD	dir	**PARAR**	S(MD) → E(A)	S(mdr2) → E(A)
LD–ST	dat	S(MD) → E(mdr1)	S(MD) → E(mdr0)	S(mdr2) → E(mdr0)
	dir	**PARAR**	S(MD) → E(A)	S(mdr2) → E(A)
LD–BR		**PARAR**	S(MD) → E(Cmp)?	S(mdr2) → E(Cmp)
ST/BR	–	–	–	–
Control		IR–IR2	IR–IR3	IR–IR4

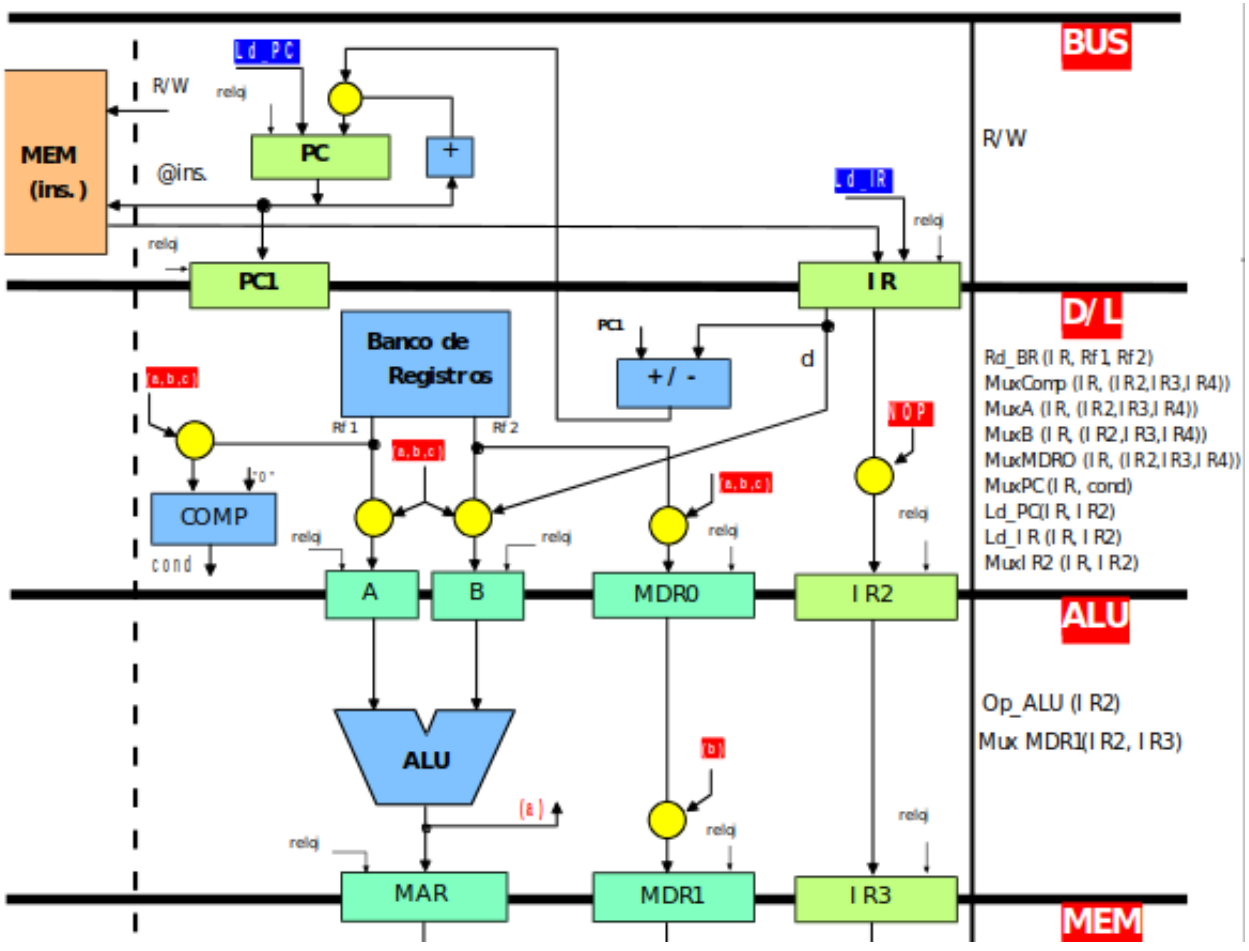
Procesador DLX: Solucionar dependencias RAW

Ruta de datos del DLX: dependencias de datos resueltas



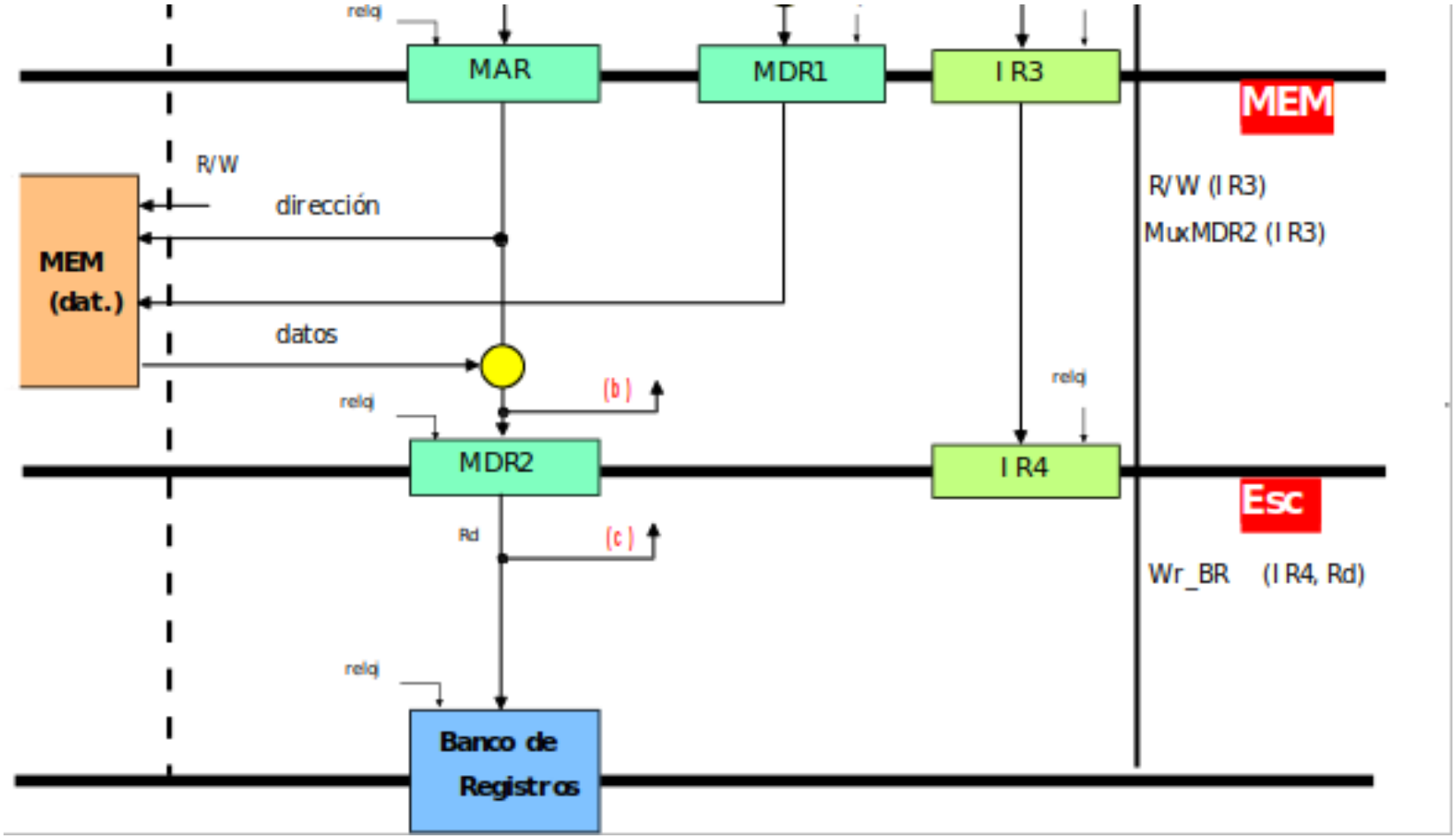
Procesador DLX: Solucionar dependencias RAW

Ruta de datos del DLX: dependencias de datos resueltas (1)



Procesador DLX: Solucionar dependencias RAW

Ruta de datos del DLX: dependencias de datos resueltas (2)



Procesador DLX: Solucionar dependencias RAW

LD-OP: compilación

- Código en C:
  A = A + B
  C = C * D
- Código ensamblador:

LD R1,0(R10);	A			LD R1,0(R10);	A
LD R2,8(R10);	B			LD R2,8(R10);	B
ADD R3,R1,R2				**LD R4,40(R10);**	**C**
ST 0(R10),R3;	A		⇒	ADD R3,R1,R2;	
LD R4,40(R10);	**C**			**LD R5,30(R10);**	**D**
LD R5,30(R10);	**D**			ST 0(R10),R3;	A
MUL R6,R4,R5;				**MUL R6,R4,R5;**	
ST 40(R10),R6;	**C**			**ST 40(R10),R6;**	**C**

- El compilador puede ayudar a minimizar las dependencias de datos, alejando entre ella las instrucciones que tienen dependencia (se profundizará luego).

Tabla de contenidos

Procesador DLX: Dependencias de control

Problema

- El flujo de ejecución de instrucciones se prevé que es secuencial: i, $i+1$, $i+2$... Pero no en los saltos.
- ¿Qué instrucción se ejecuta tras una instrucción de salto? **No se sabe** hasta determinar **si se salta o no** y **calcular la dirección de salto**, pero la ejecución segmentada continua...

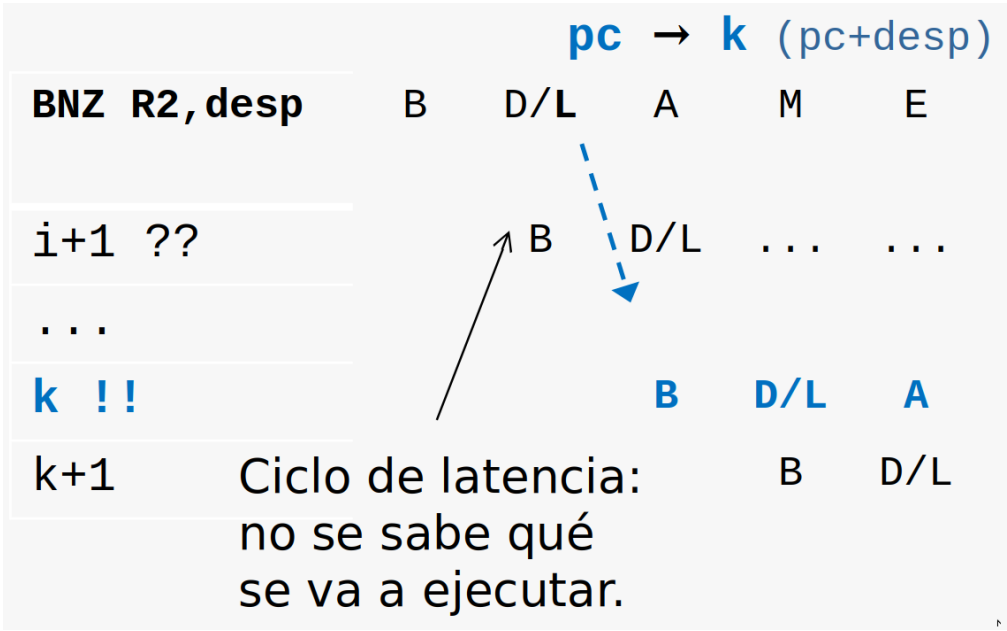
```
i:              BNZ R3 desp [ pc = pc + desp → j ]
i+1:            ¿ins1?
...
j (pc+desp):    insj
```

- Las instrucciones (i+1), (i+2)... ¡no deben ejecutarse!

Procesador DLX: Dependencias de control

Latencia del salto

- Latencia del salto: número de **ciclos que se pierden** hasta **resolver el salto** (número de etapas que tarda en resolverse el salto - 1)
- En caso de **DLX, la latencia del salto es de un ciclo**, porque la decisión (la dirección del salto) se calcula en la fase 2.



Procesador DLX: Dependencias de control

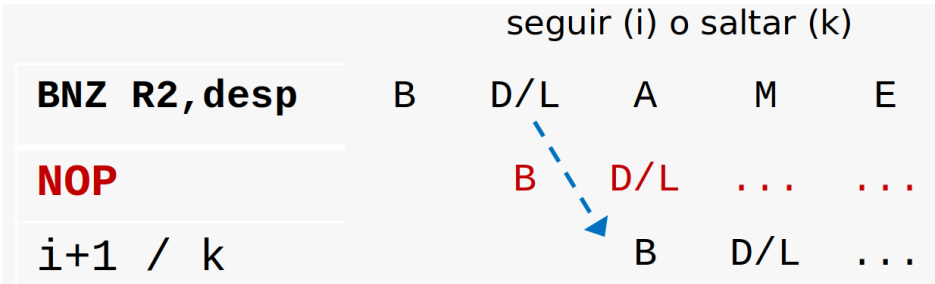
Cómo gestionar las dependencias de control

- Como hemos visto, se pueden resolver las consecuencias de las dependencias de datos con cortocircuitos (no siempre) $\rightarrow$ objetivo: no perder ciclos
- Pero, ¿qué se debe hacer para no perder ciclos de latencia de instrucciones de salto?
- En general, se resuelven por software (compilador) o por hardware (unidad de control). Opciones:
  - Perder ciclos siempre (Sw o Hw)
  - **Salto retardado** (Sw)
  - **Apostar por dónde seguirá** (Sw o Hw)
- **Nota**: utilizaremos estos símbolos para diferenciar entre diferentes tipos de salto: (1) *****, salto retardado; (2) ^, apostar a saltar; (3) **'**, apostar a seguir con el siguiente.

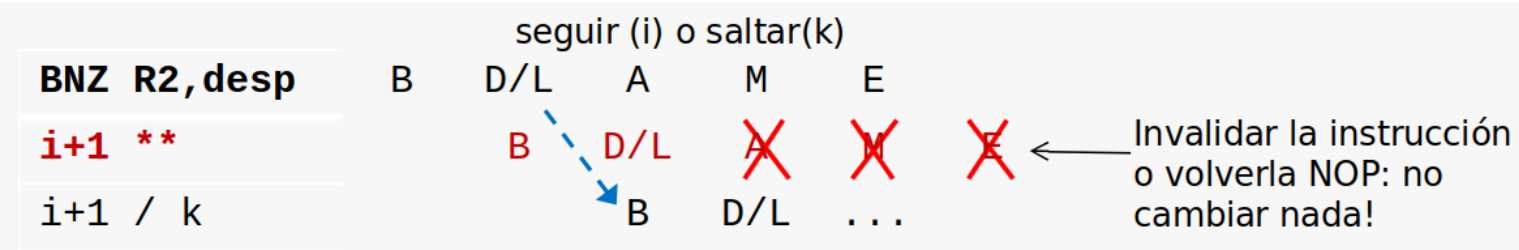
Procesador DLX: Dependencias de control

1ª opción: perder siempre los ciclos de latencia

- Para seguir con la ejecución, **se espera ha que se decida la condición del salto**; se perderán esos ciclos (en DLX, un ciclo). Dos opciones:
  - **Software**: el compilador introduce **NOP**s después del salto



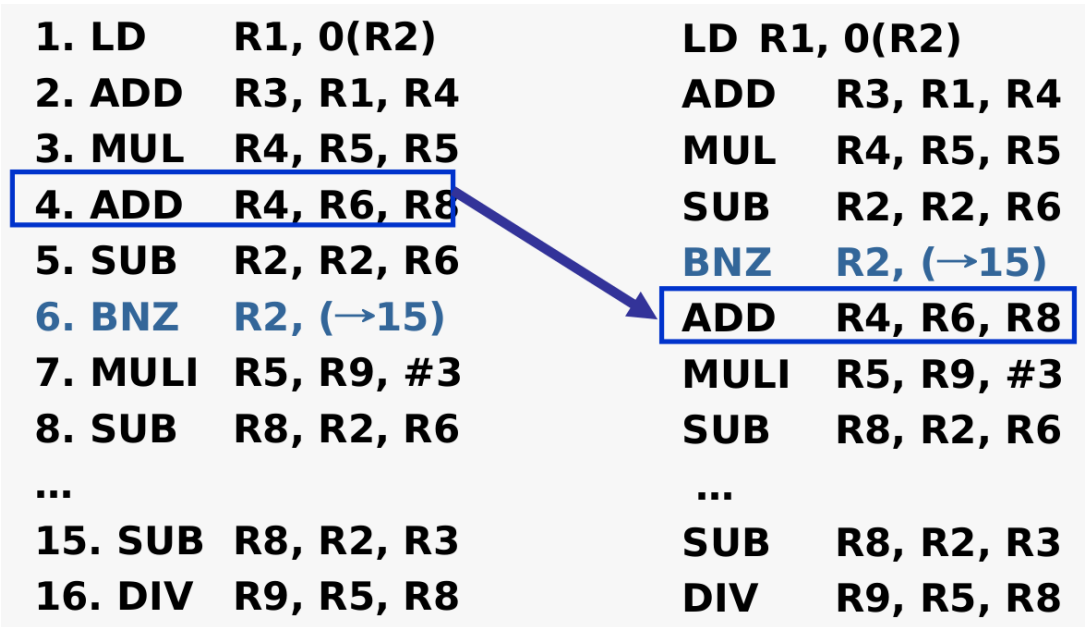
  - **Hardware**: la UC **anula** la ejecución de las siguientes instrucciones



Procesador DLX: Dependencias de control

2ª opción: salto retardado (delayed branch) – software

- Ocupar los ciclos de latencia con instrucciones útiles: el compilador **reordena el código** para **ejecutar (siempre) instrucciones** ocupando la latencia del salto, respetando las dependencias de datos.



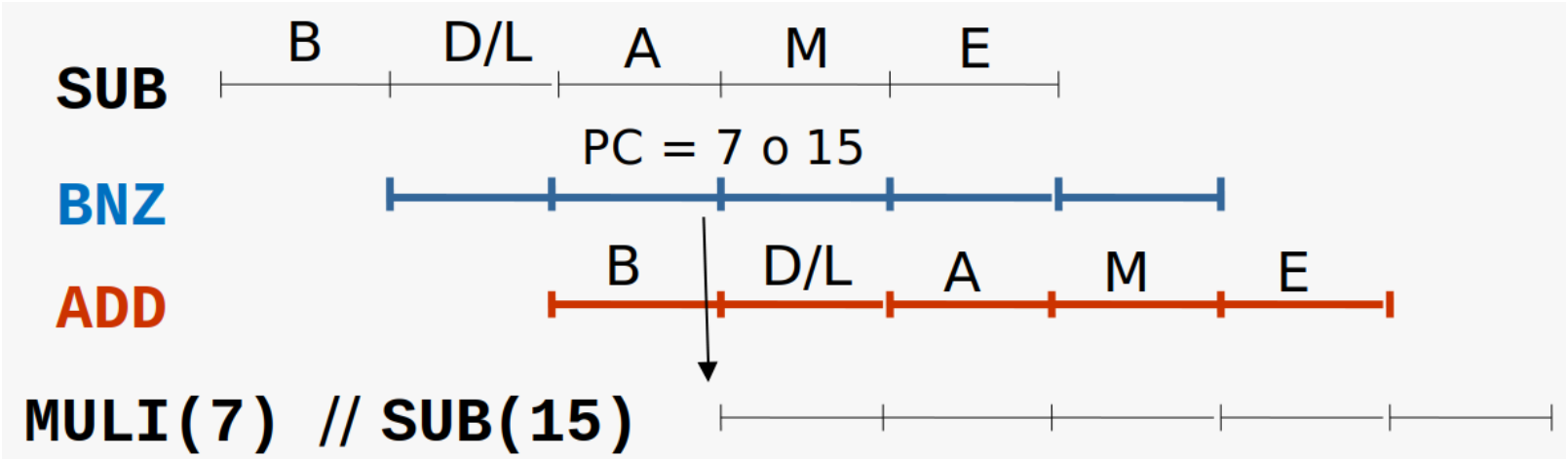
¡Atención! Actualizar el desplazamiento del salto.

En el ejemplo:
saltar a la instrucción 15

(instrucciones de 4 bytes)

Procesador DLX: Dependencias de control

2ª opción: salto retardado (delayed branch) – software



- En este ejemplo, no se pierde ningún ciclo (ADD no tiene dependencias con el SUB ni con el BNZ)

Procesador DLX: Dependencias de control

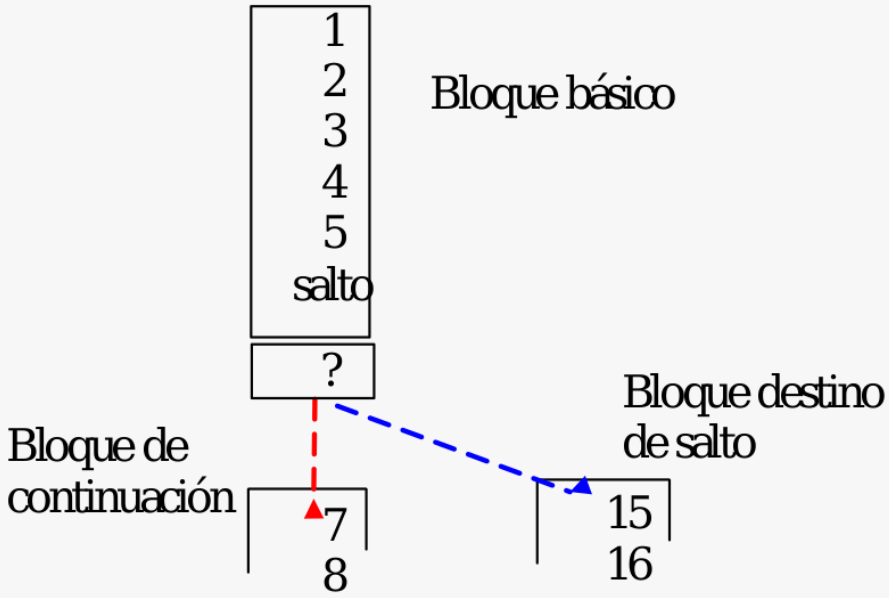
2ª opción: salto retardado (delayed branch) – software

- ¿De dónde se obtienen las instrucciones para ocupar los ciclos de latencia?
  1. **Bloque básico del salto**: Es el bloque de instrucciones en el que se encuentra el salto. Estas instrucciones siempre deben ejecutarse. El compilador examina el código para encontrar **instrucciones que puedan ejecutarse en distinto orden** y no alteren la semántica del programa (sin dependencias de datos).
  2. **NOP**: **En caso de no encontrar** suficientes instrucciones útiles, se rellenan los ciclos de latencia con **NOPs**. ATENCIÓN: ¡en este caso **se pierden ciclos**!
- La unidad de control no tiene que hacer nada porque el código viene "precocinado".

Procesador DLX: Dependencias de control

2ª opción: salto retardado (delayed branch) – software

1	LD	R1, 0(R2)
2	ADD	R3, R1, R4
3	MUL	R4, R5, R5
4	ADD	R4, R6, R8
5	SUB	R2, R2, R6
6	**BNZ**	**R2, (→ 15)**
7	MULI	R5, R9, #3
8	SUB	R8, R2, R6
...		
15	SUB	R8, R2, R3
16	DIVI	R9, R5, R8



Procesador DLX: Dependencias de control

3ª opción: apuestas estáticas (static branch prediction) – hw+sw

- Hay casos en los que el compilador no puede encontrar instrucciones seguras para ocupar los ciclos de latencia (entre las instrucciones antes del salto = en el bloque básico).
- Existe otra opción: **apostar en el salto**. Hay dos posibilidades (¡ojo! no siempre se acierta):
  - **Apuesta por saltar** (predict taken): El compilador sitúa como latencia del salto las primeras instrucciones del bloque de salto. En caso de fallar en la apuesta (no se produce el salto), la UC debe anular dichas instrucciones y se perderán esos ciclos.
  - **Apuesta por continuar** (predict not-taken): El compilador rellena los ciclos de latencia con las primeras instrucciones del bloque de continuación. En este caso, si el salto se produce (fallo en la apuesta), la UC debe anular dichas instrucciones y se perderán esos ciclos.

Procesador DLX: Dependencias de control

3ª opción: apuestas estáticas (static branch prediction) – hw+sw

- En esta sección solamente se tratara la **apuesta estática**, es decir, la que se define a la hora de compilar → siempre se realizará la misma apuesta. Pero hay la apuesta dinámica, en el que se apuesta la rama según tendencias que se observan en tiempo de ejecución. Se verá en la asignatura PAR.
- Ejemplo de apuesta por saltar:

```
5    SUB R2,R2,R6
6    BNZ R2,(→15)
7    MULI R5,R9,#3
8    SUB R8,R2,R6
     ...
15   SUB R8,R2,R3
16   DIV R9,R5,R8
```

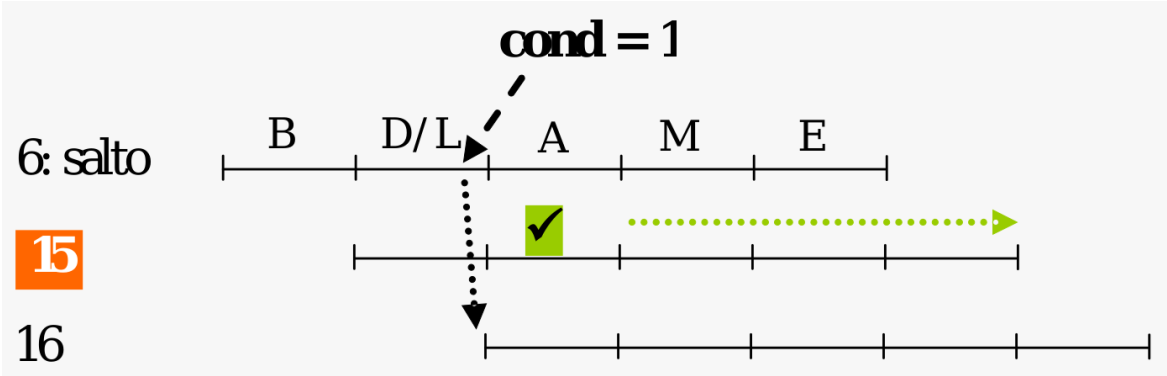
```
5    SUB R2,R2,R6
6    BNZ R2,(→16)
15   SUB R8,R2,R3
7    MULI R5,R9,#3
8    SUB R8,R2,R6
     ...
15   SUB R8,R2,R3
16   DIV R9,R5,R8
```

  - Puede ser necesario copiar instrucciones!: en casos donde a esa instrucción se puede llegar de varios puntos del código.
  - Hay que actualizar el desplazamiento

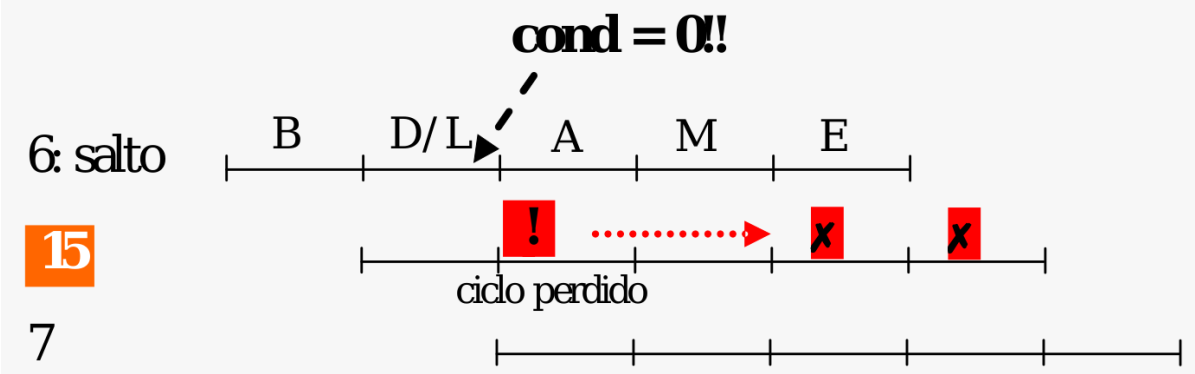
Procesador DLX: Dependencias de control

3ª opción: apuestas estáticas (static branch prediction) – hw+sw

- **Control apuesta por saltar**:
  - Se acierta la apuesta:



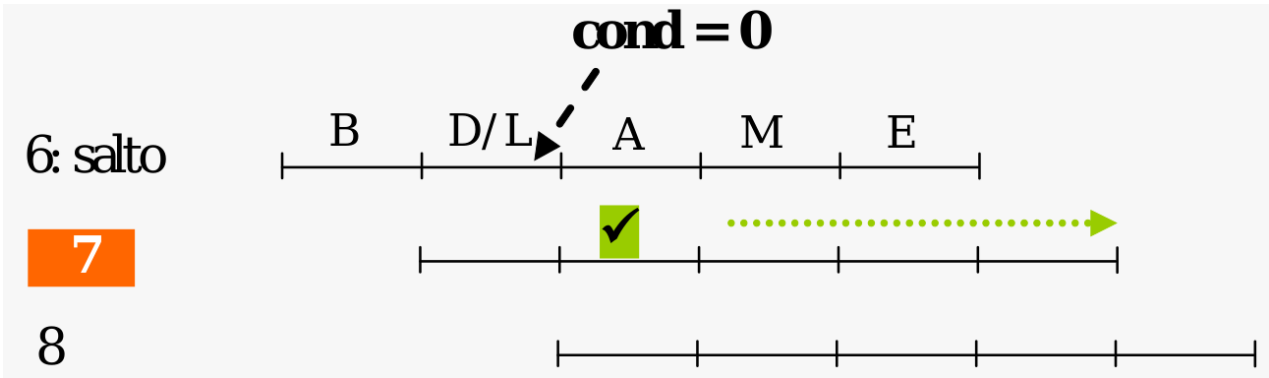
  - Se falla la apuesta:



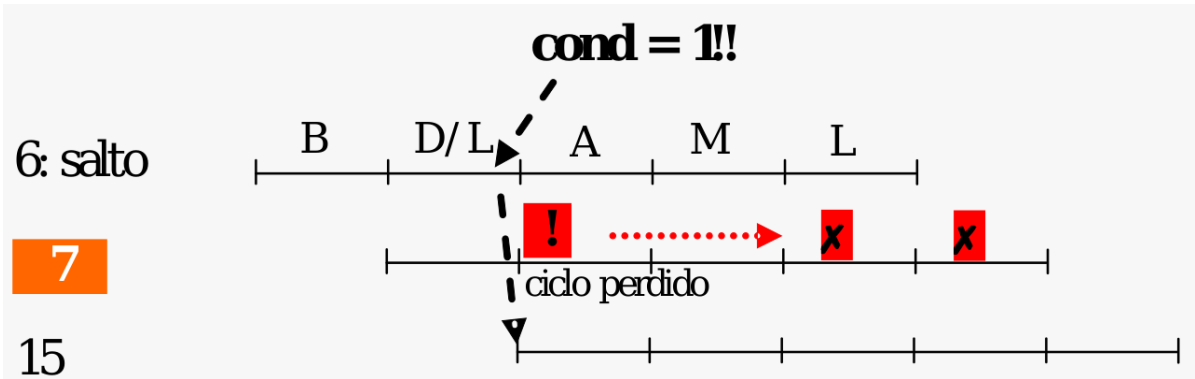
Procesador DLX: Dependencias de control

3ª opción: apuestas estáticas (static branch prediction) – hw+sw

- **Control apuesta por continuar**:
  - Se acierta la apuesta:

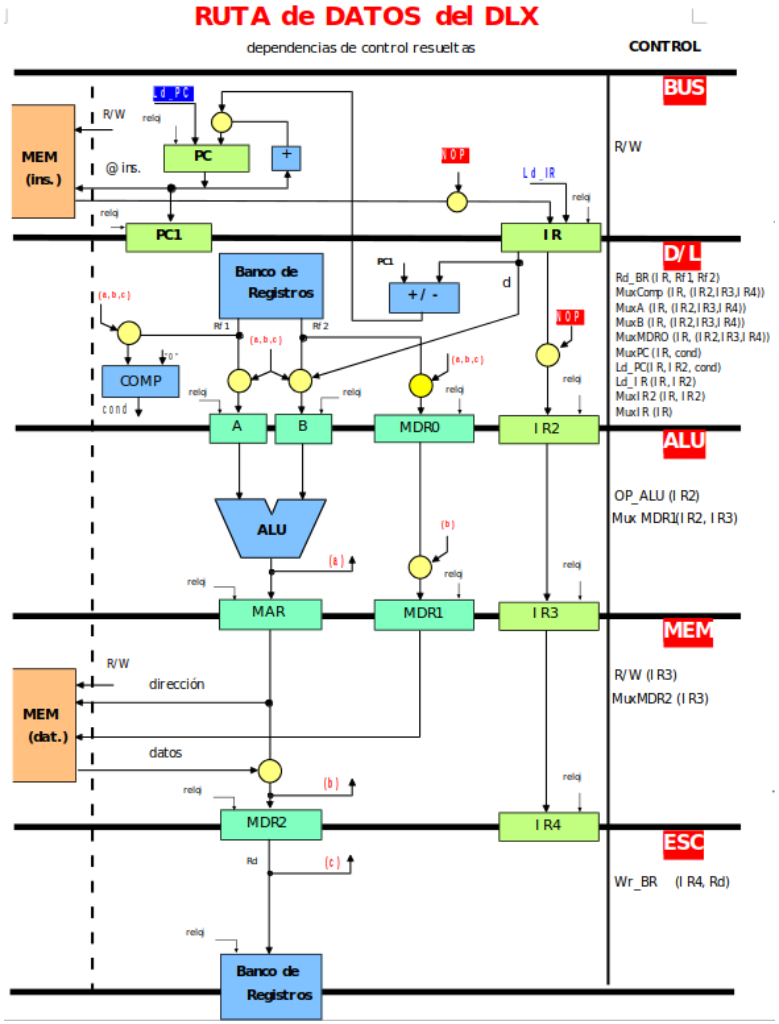


  - Se falla la apuesta:



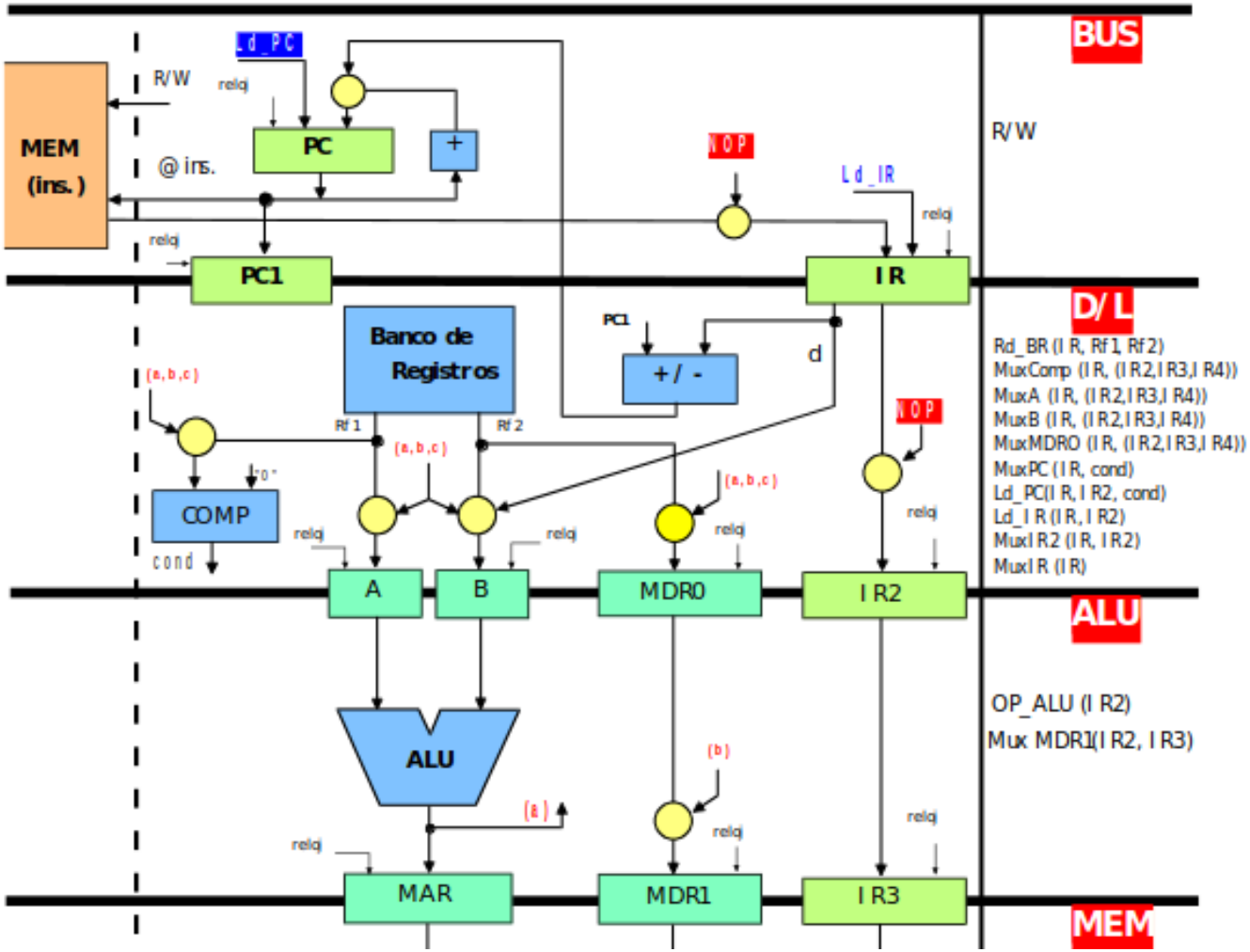
Procesador DLX: Dependencias de control

Ruta de datos del DLX (dependencias de control resueltas)



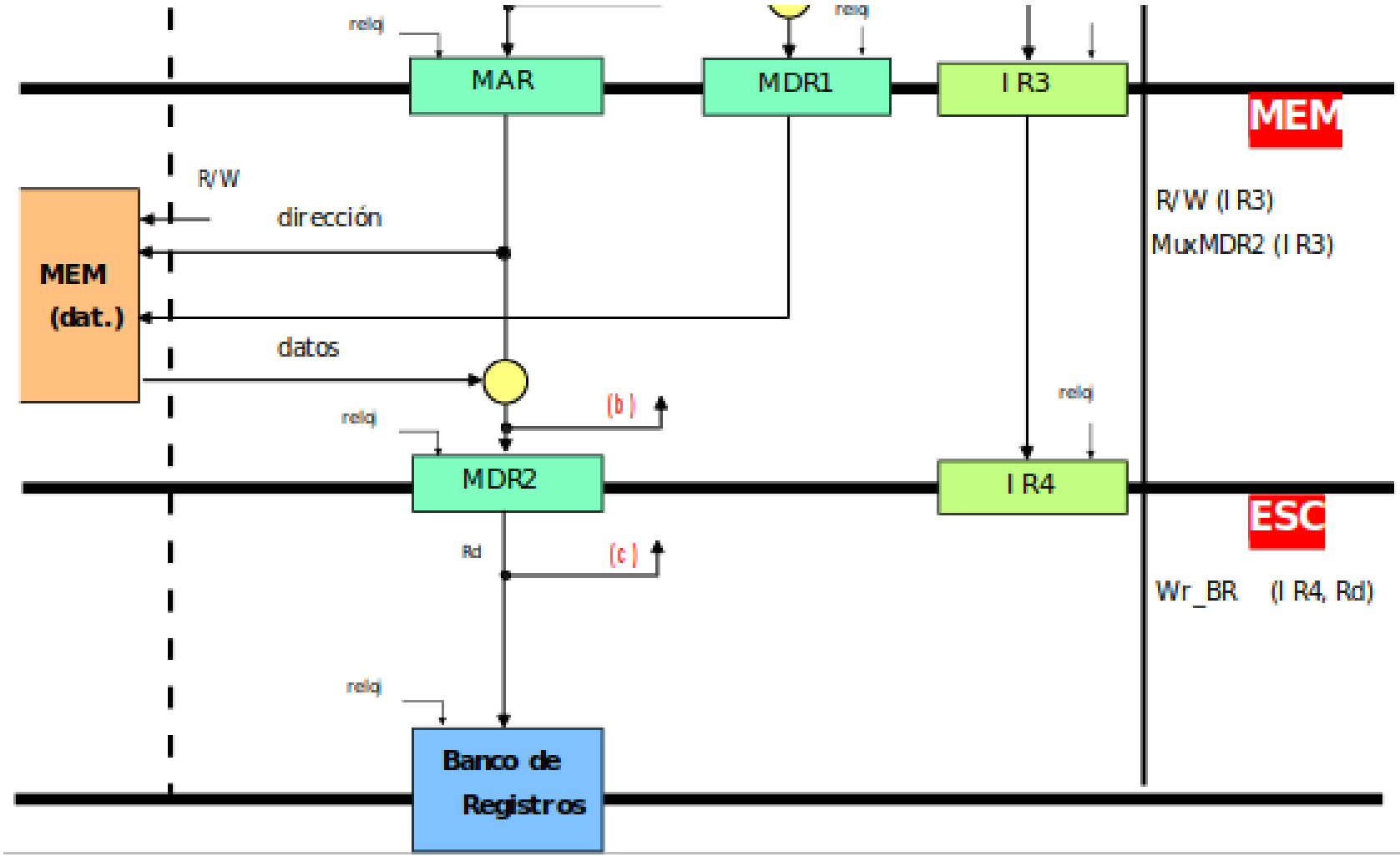
Procesador DLX: Dependencias de control

Ruta de datos del DLX: dependencias de control resueltas (1)



Procesador DLX: Dependencias de control

Ruta de datos del DLX: dependencias de control resueltas (2)



Procesador DLX: Dependencias de control

Resumen

- Mientras se decide el salto, ¿cómo seguir con la ejecución de las instrucciones?
  - **Perder ciclo(s)** $\rightarrow$ NOP (Sw/Hw)
  - **Salto retardado**: poner instrucciones adecuadas del bloque básico después del salto; si no las hay, NOP.
  - **Apuestas**: cubrir ciclos de latencia con las instrucciones que se supone que se ejecutarán después (del bloque consecutivo o del bloque de salto).
    - **Si se acierta**, no se pierden ciclos; **en caso contrario**, hay que invalidar las instrucciones con las que se haya cubierto (por Hw).
    - Cuando los saltos son hacia atrás, apostar por el salto. Cuando son hacia delante, no está claro.
- **Cuidado**: el compilador debe **"marcar"** las instrucciones de salto para que la unidad de control sepa qué estrategia se va a emplear.

Tabla de contenidos

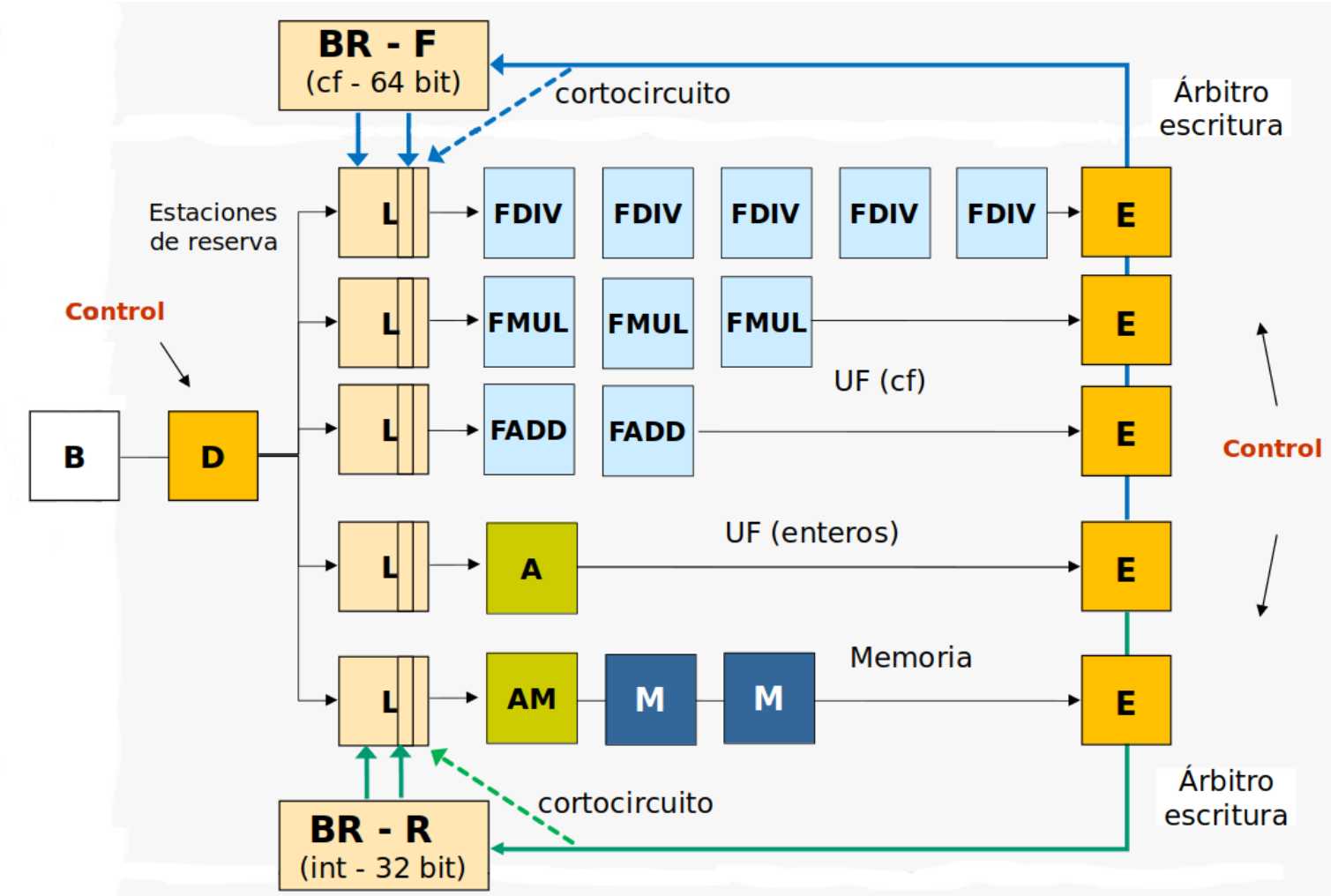
Segmentación multiciclo

Introducción

- En la segmentación lineal todas las instrucciones tienen el mismo esquema de ejecución. Por ejemplo, en el DLX: B, D/L, A, M, E
- Pero las operaciones pueden tener distinto tiempo de ejecución. Por ejemplo, suma de números enteros frente a división de números en coma flotante, o escritura en memoria.
- **Consecuencia**: todas las instrucciones deberían seguir los tiempo y etapas de las más "lentas".
  - Para ser eficientes, hay que ajustar las rutas de datos y los esquemas de segmentación **en función de las necesidades de las instrucciones** y no hacerlas todas iguales.

Segmentación multiciclo

Ejemplo: Ruta de Datos



Segmentación multiciclo

Procesadores multiciclo

- **Distintos bancos de registros**: enteros (R) y coma flotante (F). Se puede trabajar con ambos al mismo momento tanto para leer como, sobre todo, para escribir.
- **Existen diferentes y muchas unidades funcionales (segmentadas), a menudo replicadas**:
  - Para procesar números enteros
  - Para sumas/ multiplicaciones/ divisiones en coma flotante
  - Para calcular direcciones de memoria
  - ...
- Cada tipo de instrucción tiene **su tiempo de ejecución** (varios ciclos de reloj), en función de la operación a realizar.
- Cada instrucción sigue **su propio esquema de segmentación** $\rightarrow$ en cada ciclo puede empezar una operación. El control es mucho más complejo.

Procesadores multiciclo

- **El esquema de segmentación de cada instrucción es distinto** en función de la instrucción. Por tanto, **el tiempo de ejecución también lo es**. En general, se separan las etapas de descodificación y de lectura de operandos.
- Dado que los esquemas de segmentación y los números de pasos son distintos, **las instrucciones no acaban en el orden en que empiezan**: la instrucción $i+1$ puede escribir su resultado antes que la instrucción i!

Procesadores multiciclo

- **En las dependencias de datos**, no se bloquea el flujo de ejecución: se "deja de lado" la instrucción que presenta problemas en **"espacios de reserva"** de las UFs hasta que los datos estén preparados, pero continúa la ejecución del programa: las instrucciones empiezan y acaban en desorden
- Dos problemas nuevos:
  - Puede haber **anti-dependencia** (WAR) y **dependencia de salida** (WAW).
  - Pueden querer hacerse varias escrituras en el mismo BR en el mismo ciclo (normalmente no se acepta).

Segmentación multiciclo

Ejemplo de una ejecución

FDIV	B	D	L	F/	F/	F/	F/	F/	E	
FMUL		B	D	L	F*	F*	F*	**E? [1]**		
ADD			B	D	L	A	**E? [1]**			
LD				B	D	L	AM	M	M	E?
FADD					B	D	L	F+	F+	**E? [2]**
FADD						B	D	L	**F+? [3]**	–

- **[1]**: Escrituras en desorden
- **[2]**: Escrituras simultáneas
- **[3]**: UF simultáneas/ ocupadas

Tabla de contenidos

Procesadores superescalares

Tiempo de ejecución

- Tiempo de ejecución de un programa: $Tej = NI \times NCI \times Tc$
  - NI: número de instrucciones;
  - NCI: número de ciclos por instrucción (CPI)
  - Tc: tiempo de ciclo
- ¿Puede ser NCI < 1?
- **Definición**: en los procesadores superescalares las instrucciones no se buscan una a una, sino en bloques.
  - Al número de instrucciones que se buscan en bloque se le llama grado del superescalar. Generalmente, buscan 2–4 instrucciones por ciclo.
  - Hoy en día todos los procesadores son superescalares.
- **Objetivo**: Lanzar la ejecución de más de una instrucción por ciclo.

Procesadores superescalares

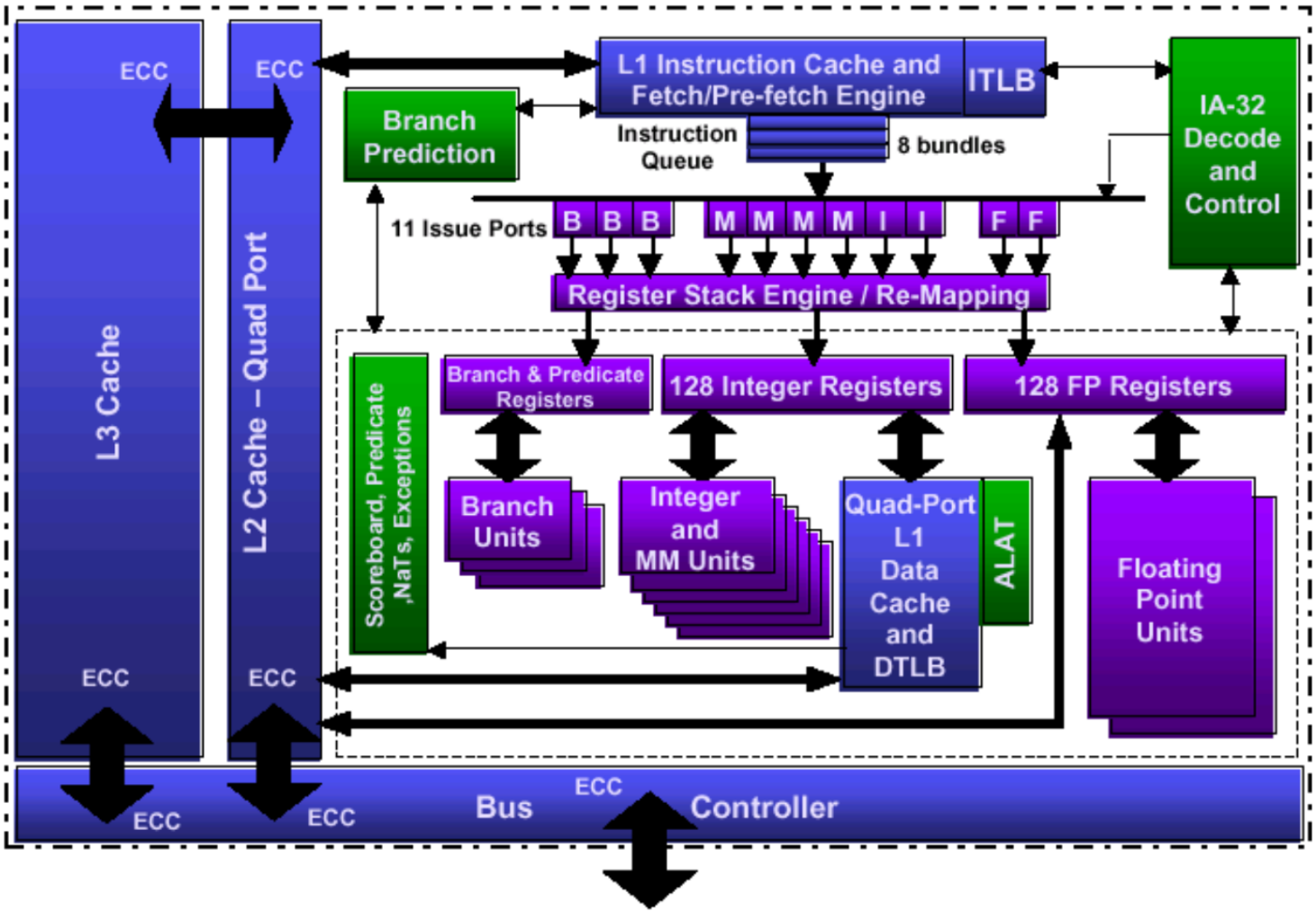
Tiempo de ejecución

```
B   D   L   A     A   A     E
B   D   L   A     A   E
    B   D   L     A   A     A   A   A   E
    B   D   L     A   A     A   A   A   A   E
        B   D     L   AM    M   E
        B   D     L   A     A   A   E
                  ...
                  ...
```

- Ejecución superescalar con s grados: $Tej = NI \times 1/s \times Tc$
- Ejemplos: (1) AMD Opteron **s=9**; (2) HP/Compaq Alpha EV7 **s=6**; (3) Itanium2 **s=6** (VLIW);
- A pesar de existir procesadores superescalares de grado alto, la tendencia actual es distinta: utilizar la complejidad para conseguir paralelismo (**multithreading**, **multicore**)

Procesadores superescalares

Un ejemplo de la micro-arquitectura Itanium 2



Tema 2: Ejecución segmentada de instrucciones

Parte 1

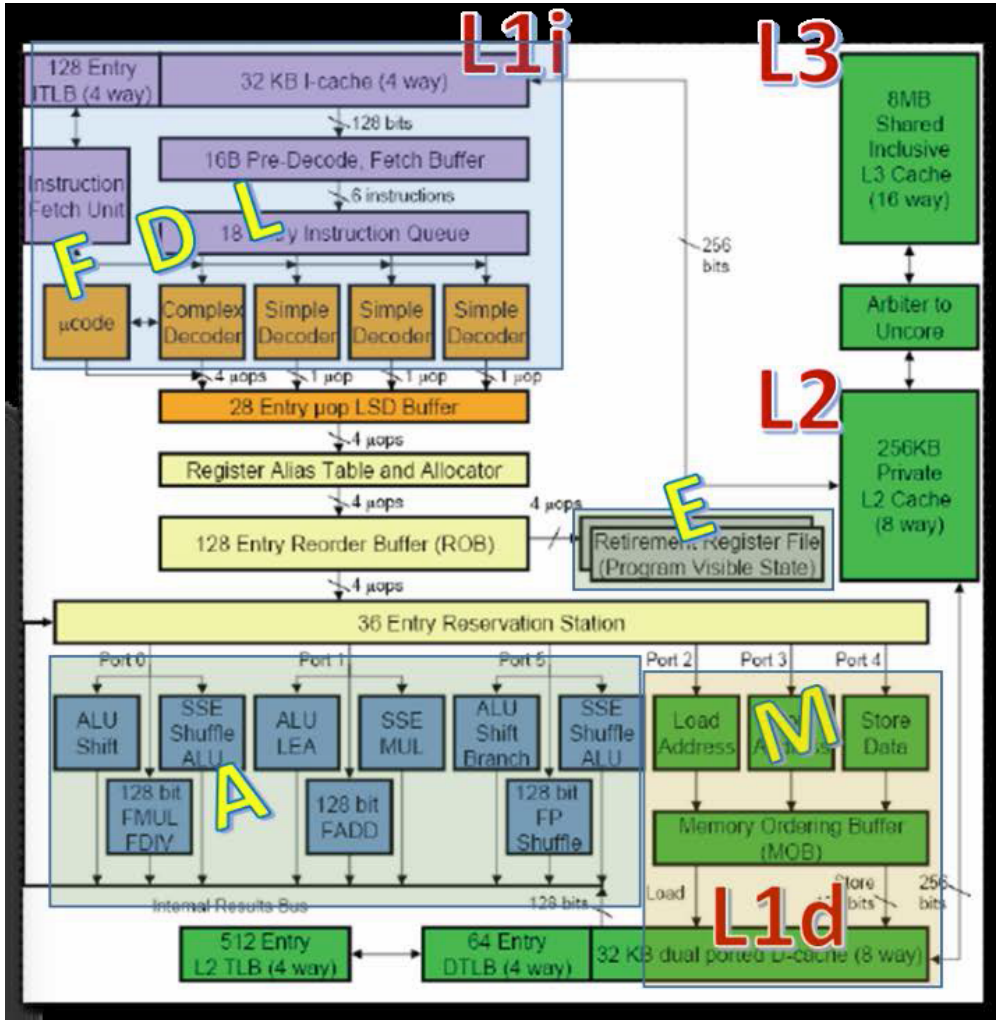
Iñigo Perona Balda (CAS)
Nestor Garay (EUS) — Olatz Arbelaitz (CAS/EUS)

Universidad del País Vasco (UPV/EHU)
Grado en Ingeniería Informática
Arquitectura de Computadores

22 de septiembre de 2023

Procesadores superescalares

Un ejemplo de una micro-arquitectura



Procesadores superescalares

Número de fases en diferentes micrp-arquitecturas

Microarchitecture	Pipeline stages
P5 (Pentium)	5
P6 (Pentium 3)	10
P6 (Pentium Pro)	14
NetBurst (Willamette)	20
NetBurst (Northwood)	20
NetBurst (Prescott)	31
NetBurst (Cedar Mill)	31
Core	14
Bonnell	16
Sandy Bridge	14
Silvermont	14 to 17
Haswell	14
Skylake	14
Kabylake	14