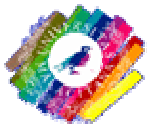
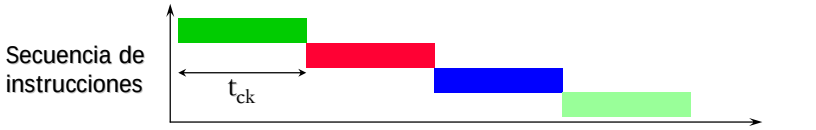
Tema 2: Segmentación



- Introducción
- Segmentación del camino de datos
- Control del sistema segmentado
- Dependencias entre instrucciones
- Riesgos entre instrucciones

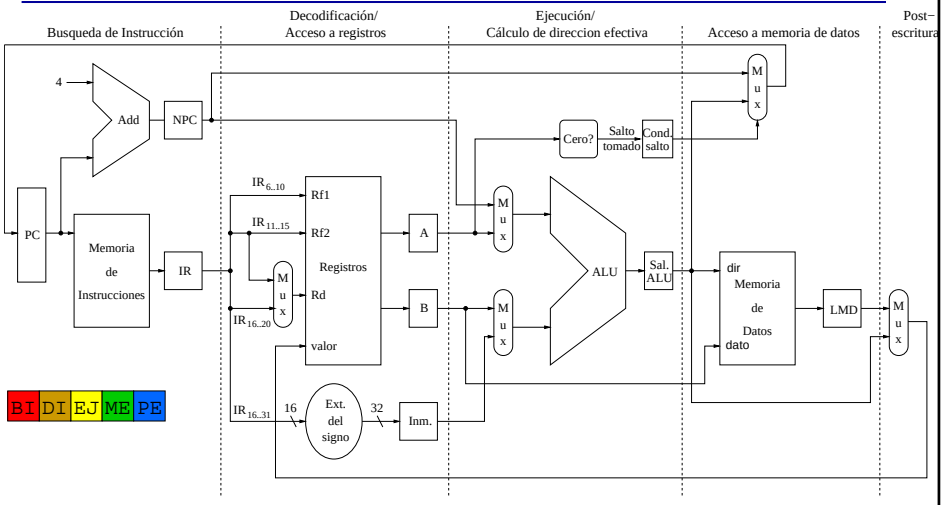
2.1 Introducción
Implementaciones no segmentadas

- Implementación **monociclo**:
  - ✓ Toda instrucción consume 1ck
  - ✓ El CPI de todas las instrucciones es 1
  - ✓ La frecuencia de reloj depende del camino crítico: instrucción más larga



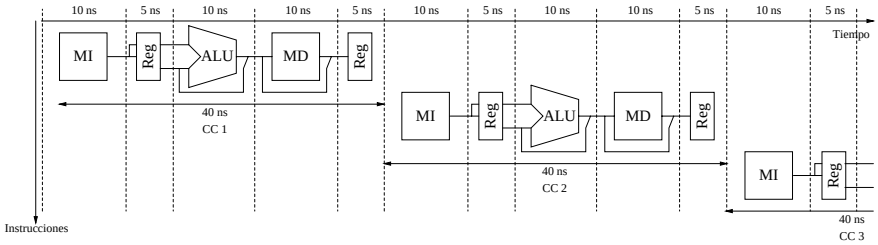
- Implementación **multiciclo**:
  - ✓ La ejecución de la instrucción se divide en etapas: cada etapa 1ck.
  - ✓ El CPI de cada instrucción depende del número de etapas que necesite ejecutar
  - ✓ La frecuencia de reloj depende de la etapa más lenta: balancear etapas

2.1 Introducción
Implementaciones no segmentadas del DLX



2.1 Introducción
Implementación monociclo del DLX

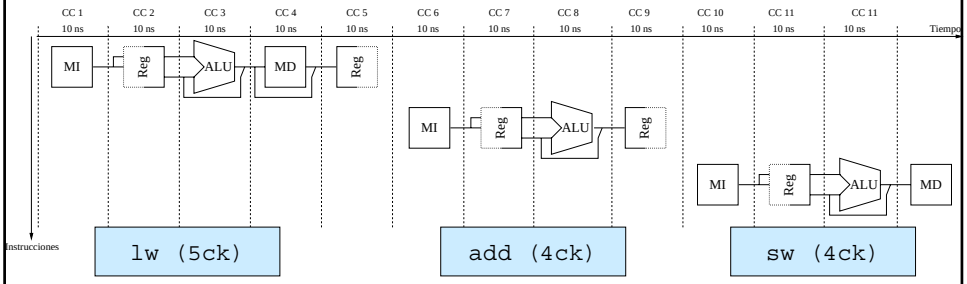
- Supongamos los siguientes tiempos:
  - ✓ Caches de instrucciones y datos: 10ns (se usa en BI y M)
  - ✓ Banco de registros: 5ns (se usa en DE y PE)
  - ✓ Operación aritmética en la ALU: 10ns (se usa en EJ)



$Tcpu = N \times CPI \times t_{ck}$ t_{ck}=40ns. CPI=1 → Tcpu=40N ns

2.1 Introducción
Implementación multiciclo del DLX

- El t_{ck}=10ns (F=100MHz). El CPI depende del tipo de instrucciones
  - ✓ Supongamos CPI(lw)=5, CPI(resto)=4

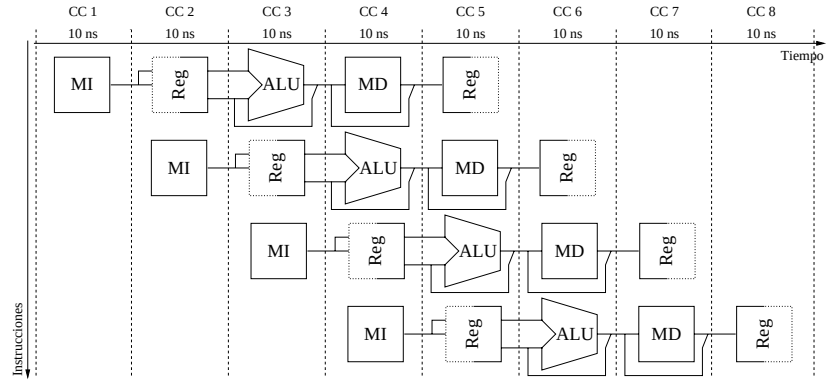


- Supongamos un programa con 20% de lw.

$$CPI = 0.2x5 + 0.8x4 \rightarrow CPI=4.2$$

$$Tcpu= N \times CPI \times t_{ck} \quad t_{ck}=10ns.\ CPI=4.2 \rightarrow Tcpu=42N\ ns$$

2.1 Introducción
Implementación segmentada del DLX



- Ventaja:

$$Tcpu= N \times CPI \times t_{ck;} \quad t_{ck}=10ns.\ CPI=1 \rightarrow Tcpu=10N\ ns$$

- Problemas:
  - ✓ Tener PC disponible a cada ck. Difícil obtener CPI=1. Mayor área de SI.

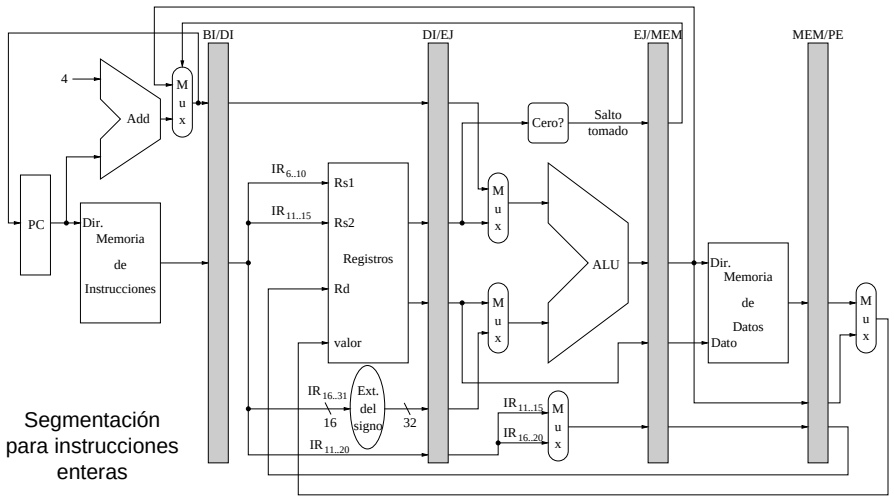
2.2 Segmentación del camino de datos Definiciones

- Segmentación:
  - ✓ Técnica para mejorar el rendimiento del procesador mediante la ejecución solapada de múltiples instrucciones
  - ✓ Ventajas:
    - Mayor aprovechamiento de recursos con un coste de diseño e implementación reducidos
    - Invisible (o casi invisible) al programador
- Etapa de segmentación:
  - ✓ Cada una de las unidades funcionales en que dividimos la ejec. de instr.
  - ✓ Cada etapa consume un ciclo de reloj
- Productividad (throughput):
  - ✓ Número de instrucciones que se terminan de ejecutar por unidad de tiempo
- Latencia:
  - ✓ Tiempo necesario para ejecutar completamente una instrucción

2.2 Segmentación del camino de datos Comparación de rendimientos

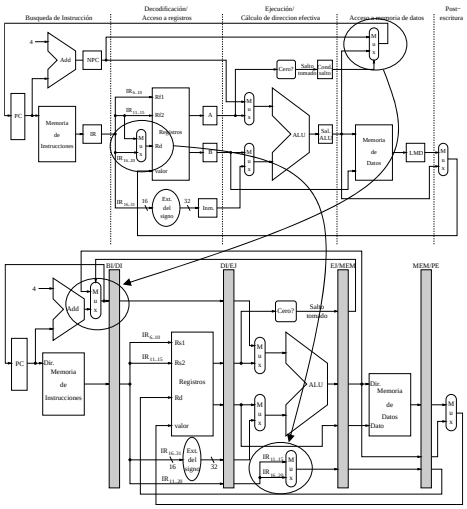
- Volviendo a las tres implementaciones del DLX
  - ✓ Implementación monociclo:
    - Latencia = 40 ns
    - Throughput = N / Tcpu = 1/40 x 10^9 = 25 MIPS.
  - ✓ Implementación multiciclo:
    - Latencia = 40 ns - 50 ns (depende de la instrucción)
    - Throughput = N/ Tcpu= 1/42 x 10^9 = 23.8 MIPS.
  - ✓ Implementación segmentada:
    - Latencia = 50 ns
    - Throughput = N/ Tcpu = 1/10 x 10^9 = 100 MIPS.
- Aceleración debido a la seg = Tcpu sin segment. / Tcpu con segment.
  - ✓ En el caso ideal: Aceleración = número de etapas = 5
  - ✓ En el ejemplo del DLX: Aceleración ≈ 4. Razones para no alcanzar ideal:
    - Etapas desbalanceadas: las etapas DI y PE consumen 10ns de los que son útiles sólo 5ns
    - Coste de la segmentación: retardos (aun no contemplados) debidos a la circuitería adicional
    - Riesgos entre instrucciones: en ocasiones hay que retrasar la ejecución de algunas instrucc.

2.2 Segmentación del camino de datos
Implementación del DLX segmentado



Segmentación para instrucciones enteras

2.2 Segmentación del camino de datos
Implementación del DLX segmentado



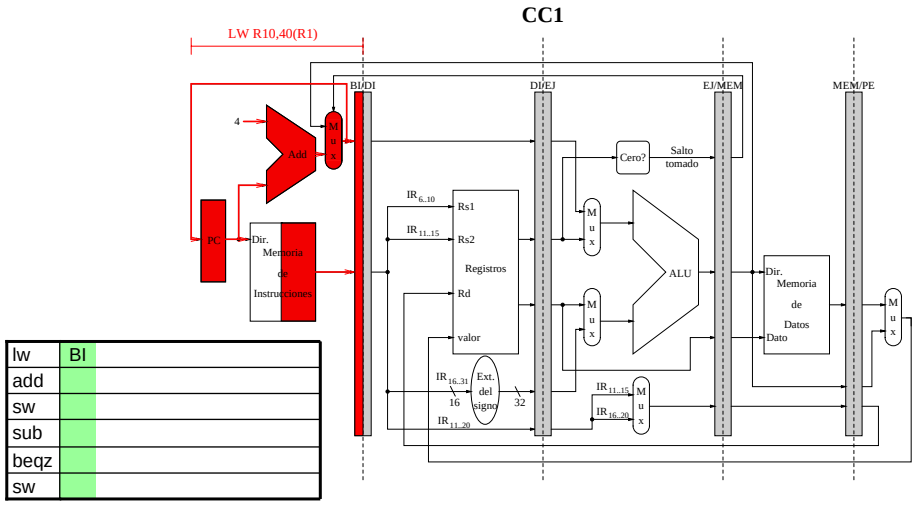
- Diferencias de implementación
  - ✓ Se introducen registros de segmentación:
    - BI/DI contiene IR y NPC
    - DI/EJ contiene A, B, Inm, NPC y NRd (dirección de registro destino)
    - EJ/ME contiene B, SalALU, condición de salto y NRd
    - ME/PE contiene LMD, SalALU, NRd
  - ✓ El multiplexor de salto se adelanta de ME a BI
    - La semántica del salto cambia
  - ✓ El multiplexor de Rd se retrasa de DI a EJ
    - Una vez decodificada la inst. sabes si es de tipo I o R, seleccionándose los bits 11..15 (I) o 16..20(R) como dir. de registro destino.

2.2.1 Ejemplo de ejecución en el DLX
Código a ejecutar

Pos. mem	etiqueta	instrucción	
I	dest:	lw	r10,40(r1)
I+4		add	r1,r2,r3
I+8		sw	0(r4),r5
I+12		sub	r2,r2,r6
I+16		beqz	r1,dest
I+20		sw	40(r1),r10

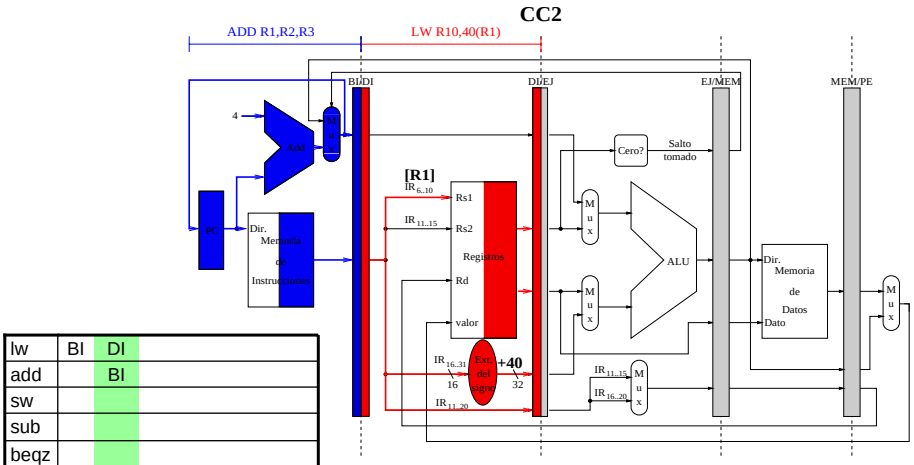
lw	1	10	40	
aritmética	2	3	1	add
sw	4	5	0	
aritmética	2	6	2	sub
beqz	1	0	-20	
sw	1	10	40	

2.2.1 Ejemplo de ejecución en el DLX
Ciclo 1



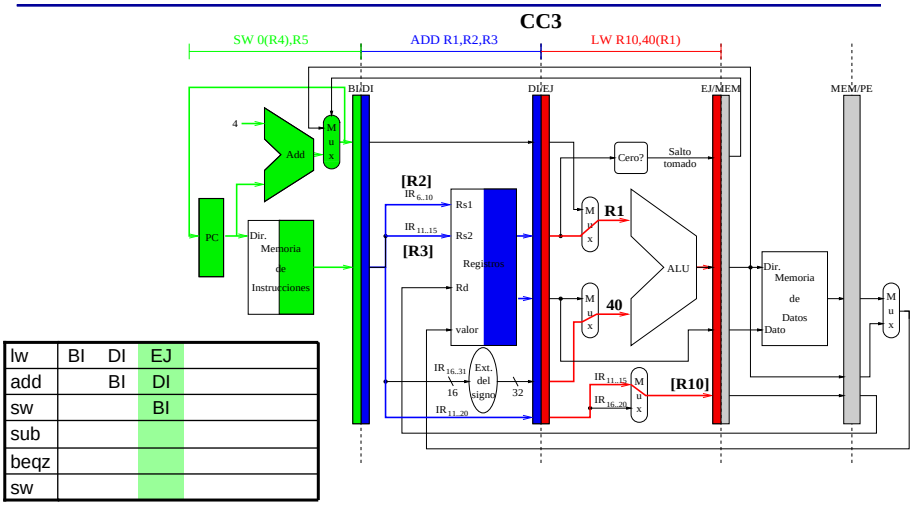
lw	BI
add	
sw	
sub	
beqz	
sw	

2.2.1 Ejemplo de ejecución en el DLX
Ciclo 2



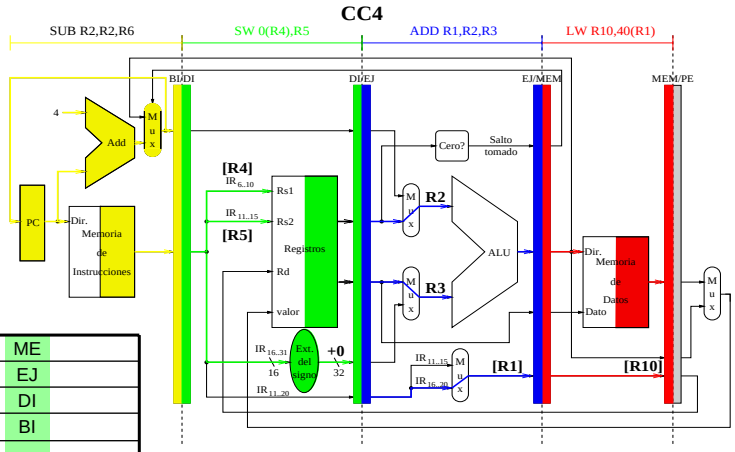
lw	BI	DI
add		BI
sw		
sub		
beqz		
sw		

2.2.1 Ejemplo de ejecución en el DLX
Ciclo 3



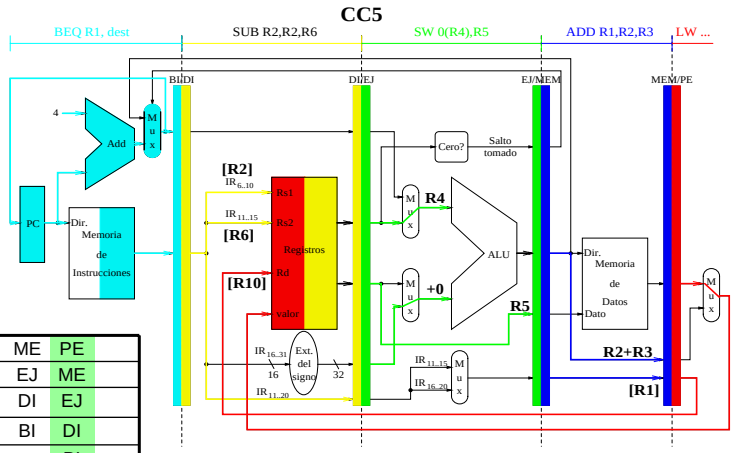
lw	BI	DI	EJ
add		BI	DI
sw			BI
sub			
beqz			
sw			

2.2.1 Ejemplo de ejecución en el DLX
Ciclo 4



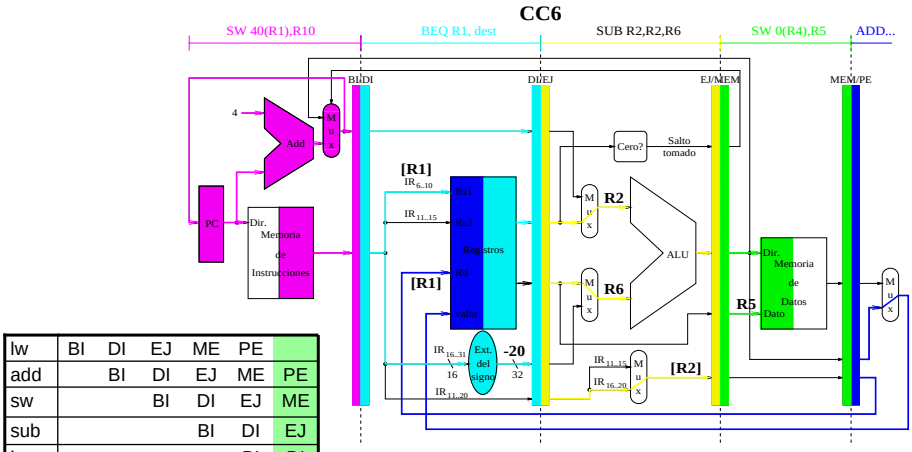
lw	BI	DI	EJ	ME	
add		BI	DI	EJ	
sw			BI	DI	
sub				BI	
beqz					
sw					

2.2.1 Ejemplo de ejecución en el DLX
Ciclo 5



lw	BI	DI	EJ	ME	PE
add		BI	DI	EJ	ME
sw			BI	DI	EJ
sub				BI	DI
beqz					BI
sw					

2.2.1 Ejemplo de ejecución en el DLX
Ciclo 6



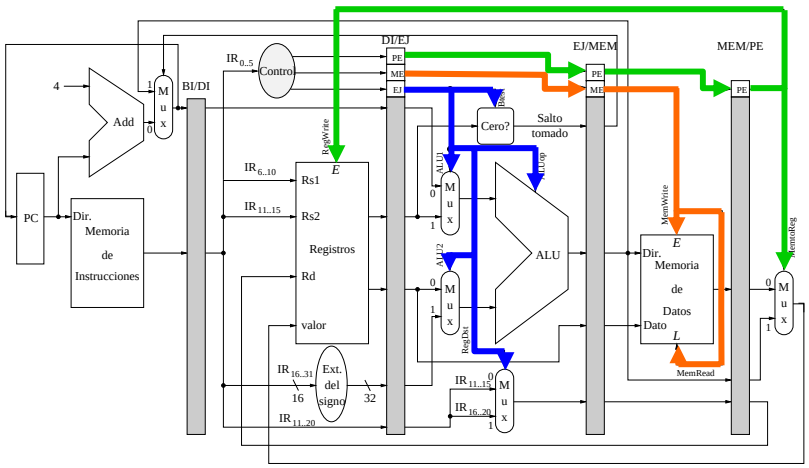
lw	BI	DI	EJ	ME	PE	
add		BI	DI	EJ	ME	PE
sw			BI	DI	EJ	ME
sub				BI	DI	EJ
beqz					BI	DI
sw						BI

2.3 Control del sistema segmentado
Señales de control necesarias

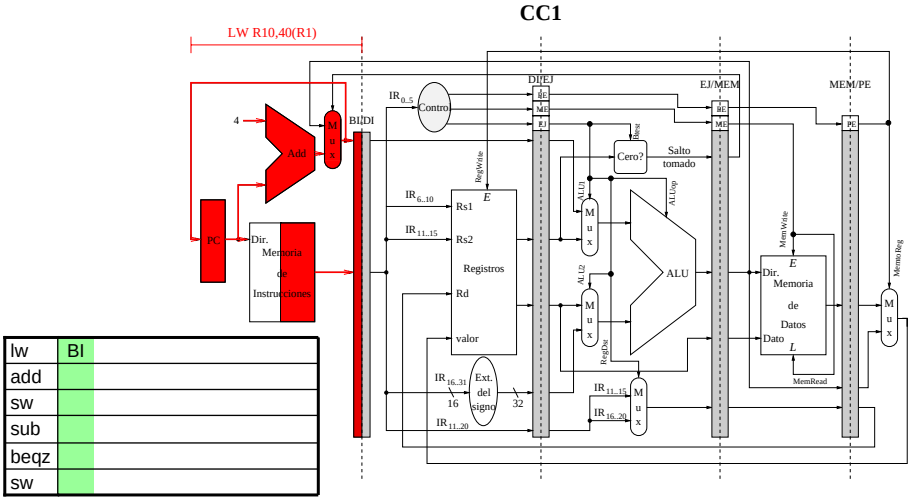
- Señales de control generadas en la etapa DI
  - ✓ Las etapas BI y DI siempre hacen igual (independ. de la inst) → no requieren control
- Los registros de segmentación transfieren las señales de control entre las etapas
  - ✓ Etapa EJ: Control de multiplexores de entrada ALU, op en ALU, selección de dir. de registro destino (tipo I o R) y control de salto
  - ✓ Etapa ME: Selección de lectura o escritura
  - ✓ Etapa WB: Selección de valor a escribir y activar escritura en banco de reg.

Etapa	Señal	Efecto si 0	Efecto si 1
EJ	ALU1	PC → ALU(1)	Reg. → ALU(1)
	ALU2	Reg. → ALU(2)	Inm. → ALU(2)
	RegDst	Reg. dest = $IR_{11..15}$ (tipo I)	Reg. dest = $IR_{16..20}$ (tipo R)
	Btest	Nada	Salto? (1 si entr.=0)
	ALUop	Suma	Resta
ME	MemWrite	Nada	Escribe en Mem
	MemRead	Nada	Lee de Mem
PE	MemtoReg	Mem. → Reg.	ALU → Reg.
	RegWrite	Nada	Escribe banco registros

2.3 Control del sistema segmentado
Señales de control necesarias

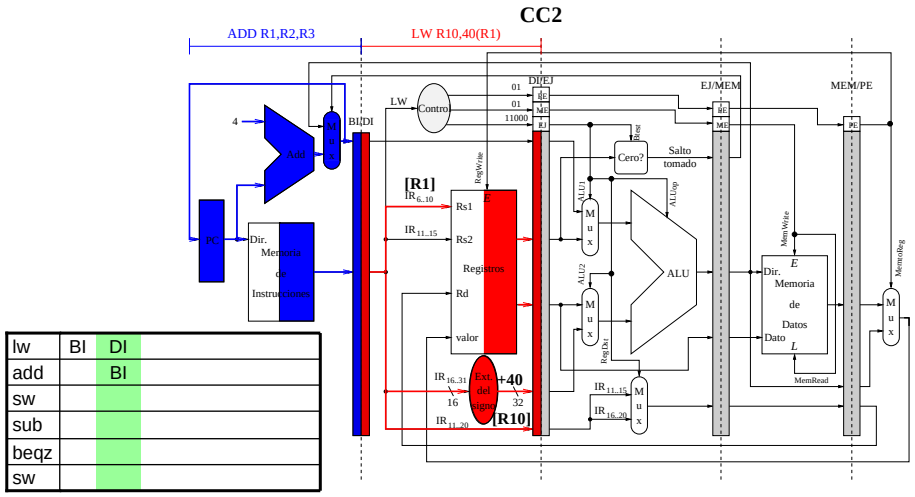


2.3.1 Ejemplo ejecución
Ciclo1



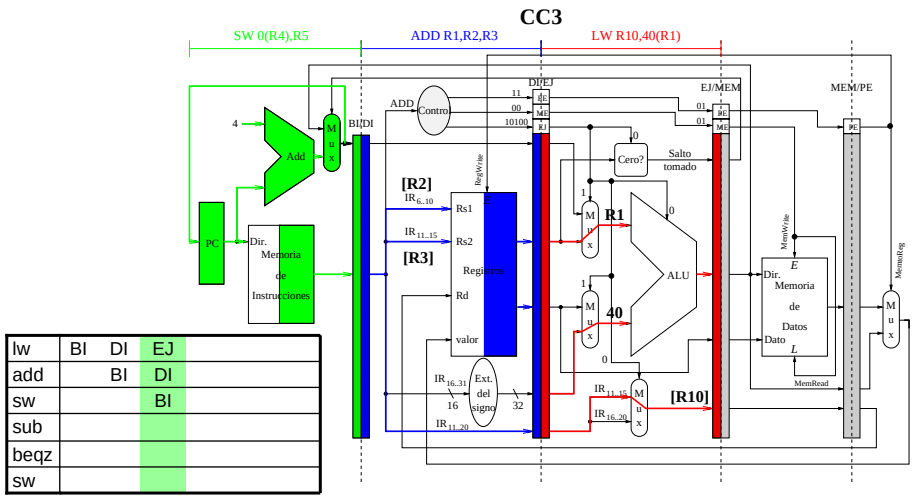
lw	BI
add	
sw	
sub	
beqz	
sw	

2.3.1 Ejemplo de ejecución
Ciclo 2



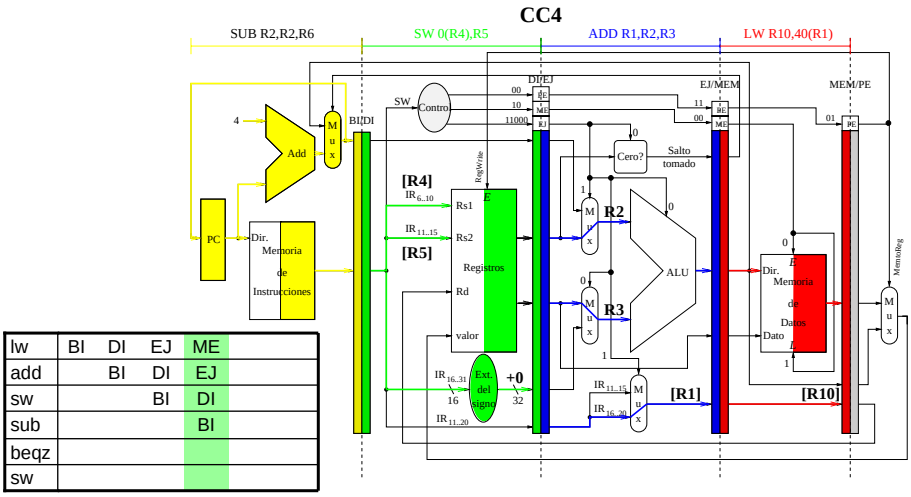
lw	BI	DI
add		BI
sw		
sub		
beqz		
sw		

2.3.1 Ejemplo de ejecución
Ciclo 3



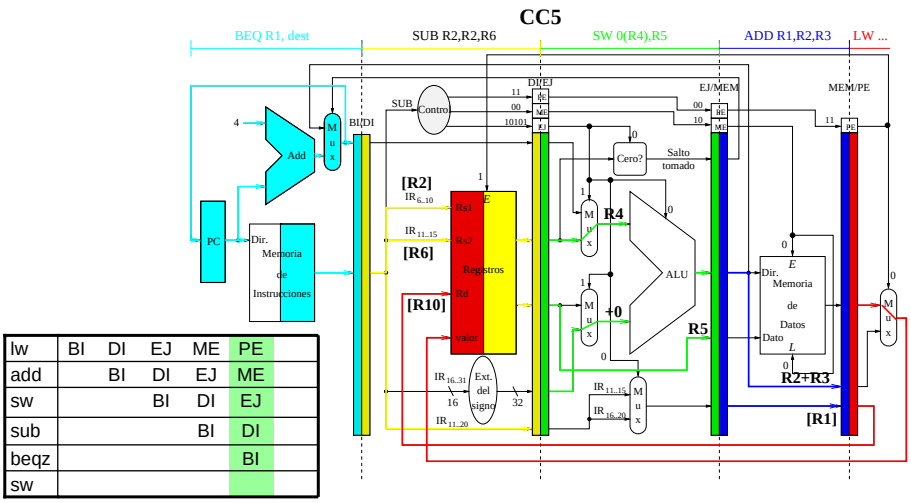
lw	BI	DI	EJ
add		BI	DI
sw			BI
sub			
beqz			
sw			

2.3.1 Ejemplo de ejecución
Ciclo 4



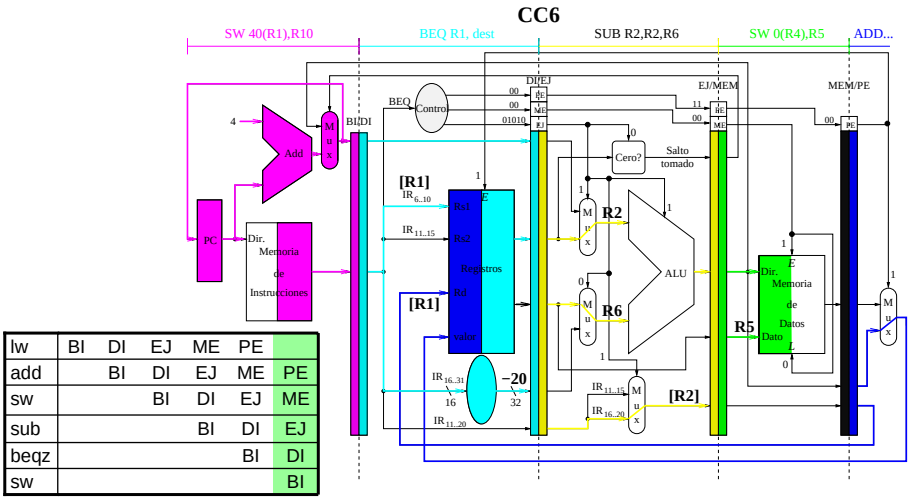
lw	BI	DI	EJ	ME
add		BI	DI	EJ
sw			BI	DI
sub				BI
beqz				
sw				

2.3.1 Ejemplo de ejecución
Ciclo 5



lw	BI	DI	EJ	ME	PE
add		BI	DI	EJ	ME
sw			BI	DI	EJ
sub				BI	DI
beqz					BI
sw					

2.3.1 Ejemplo de ejecución
Ciclo 6



lw	BI	DI	EJ	ME	PE	
add		BI	DI	EJ	ME	PE
sw			BI	DI	EJ	ME
sub				BI	DI	EJ
beqz					BI	DI
sw						BI

2.4 Dependencias entre instrucciones
Tipos de dependencias

- **Dependencia**: circunstancia que restringe la ejecución.

  Las dependencias entre instrucciones *pueden* dar lugar…
  - ✓ A la ejecución incorrecta del código si no actuamos sobre ellas.
  - ✓ A conflictos en la ejecución.
- **Tipos de dependencias** (supongamos que i precede a j):
  - **Dependencia de datos**: se debe respetar el orden de acceso al mismo operando (registro, posición de mem.) en las instrucciones i y j
    - **Dependencia verdadera**
      - i escribe el operando y j lo lee
    - **Antidependencia**
      - i lee el operando y j lo escribe
    - **Dependencia de salida**
      - i y j escriben el operando

    ➡ Dependencias falsas o dependencias por nombre
  - **Dependencia de recursos**: i y j utilizan el mismo recurso
  - **Dependencia de control**: i decide si se ejecuta o no j (saltos condicionales)

2.4.1 Dependencias de datos

- Dependencia verdadera o de flujo
  - ✓ Una instrucción j depende de otra i si se verifica que:
    - La instrucción i produce un resultado que es usado por j
  - ✓ Ejemplo:

```
i: ld   f0 , 0(r1)
j: addd f4, f0 , f2
k: sd   0(r1), f4
```

j depende de i
k depende de j

- Antidependencia:
  - ✓ Una instrucción j depende de otra i por antidependencia si:
    - Si i precede a j e i lee un reg. o posición de mem. que es escrito luego por j
  - ✓ Ejemplo:

```
i: addd f4, f0 , f2
j: ld   f0, 0(r1)
```


2.4.1 Dependencias de datos

- Dependencia de salida
  - ✓ Una instrucción j depende de otra i por dependencia de salida si:
    - i precede a j y las dos instrucciones (j e i) escriben en el mismo reg. o posición de mem.
  - ✓ Ejemplo

```
i: addd f0 ,f4,  f2
j: ld   f0, 0(r1)
```

- Grafo de dependencias:

```
S1      a = b + e
S2      if (a>10) goto L1
S3           d = b × e
S4           e = d + 1
S5 L1:  d = e / 2
```

2.5 Riesgos entre instrucciones

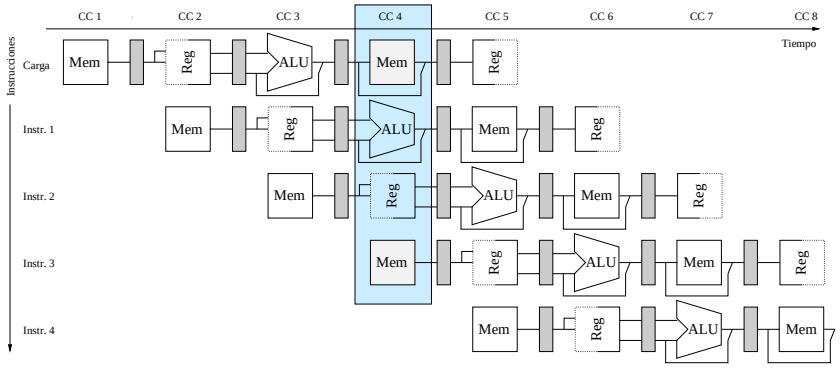
- Riesgo o Azar (Hazard):
  - ✓ Cuando la existencia de una dependencia produce una ejecución incorrecta de las instrucciones segmentadas.
- Tipos de riesgos
  - ✓ Riesgos estructurales:
    - Derivan de dependencias estructurales
    - Dos instrucciones intentan acceder en el mismo ciclo a la misma unidad funcional: conflicto
    - Una de las instrucciones tiene que esperar a la otra
  - ✓ Riesgos de control:
    - Derivan de dependencias de control
    - Consecuencia de la ejecución de instrucciones que modifican el PC: saltos, saltos condicionales, salto subrutina
    - Hasta que no se conoce la dirección de salto y si el salto es efectivo (o no) no se puede buscar la siguiente inst.
  - ✓ Riesgos de datos:
    - Derivan de dependencias de datos (verdadera, antidependencia y de salida)
    - Conflictos por el acceso a datos (registros o posiciones de memoria)

2.5.1 Riesgos estructurales

- Conflictos en los accesos a los recursos del sistema
- Por ejemplo:
  - ✓ Accesos simultáneos a la misma memoria
  - ✓ Escrituras y lecturas simultáneas en el banco de registros
  - ✓ Acceso a unidades funcionales no segmentadas que consumen más de un ciclo de reloj (por ejemplo, unidades en punto flotante)
- Solución inmediata (barata pero ineficiente)
  - ✓ Detener el flujo de instrucciones hasta que se resuelva el riesgo
  - ✓ Se insertan burbujas (stall) mientras se resuelve el riesgo
- Soluciones eficientes (pero más caras):
  - ✓ Usar caches particionadas de datos e instrucciones
  - ✓ Aumentar el número de puertos de acceso al banco de registros
  - ✓ Replicar o segmentar las unidades funcionales que consumen más de un ciclo

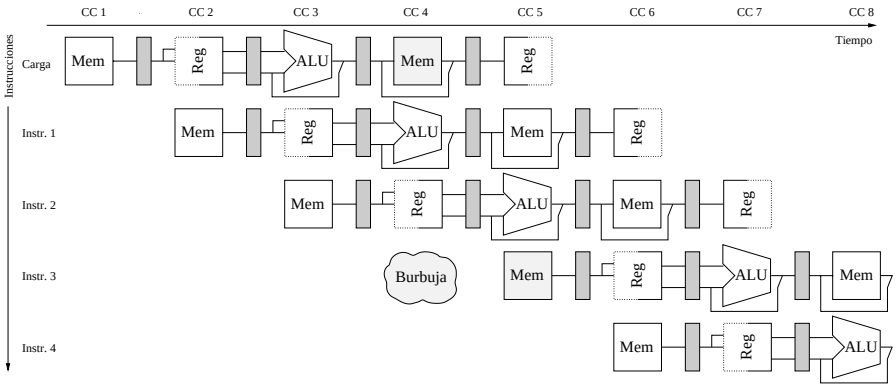
2.5.1 Riesgos estructurales
Acceso simultáneo a la misma memoria

- Supongamos que tenemos las caches de instrucciones y datos unificadas
- La primera instrucción es un lw (usa la cache en las fases BI y ME)
- La BI de la instrucción 3 colisiona con el lw en el acceso a ME (CC4)



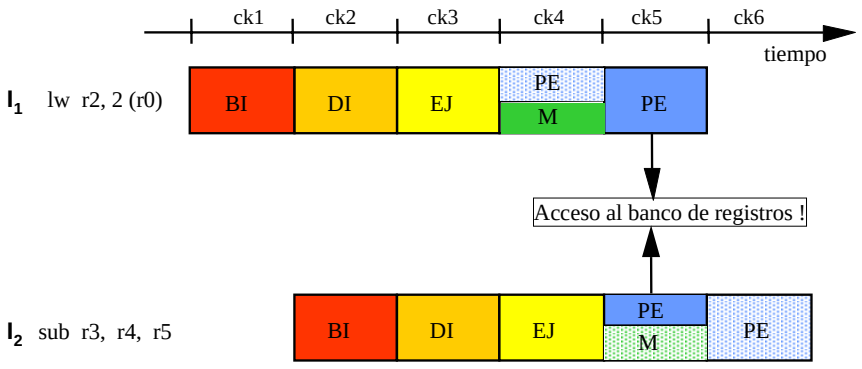
2.5.1 Riesgos estructurales
Acceso simultáneo a la misma memoria

- Una solución posible: detención de la instrucción 3 durante un ciclo
- ¿Sólo un ciclo?



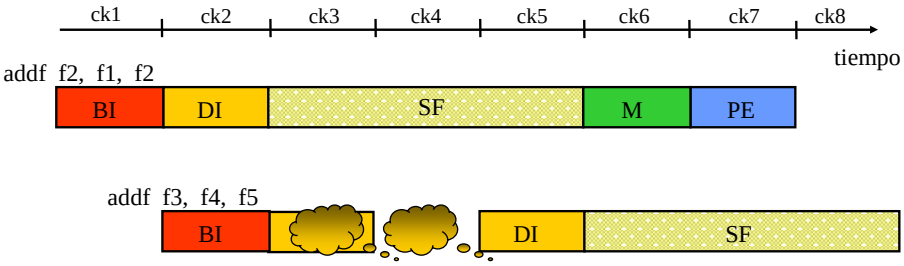
2.5.1 Riesgos estructurales
Acceso simultáneo a un puerto del banco reg.

- El banco de registros sólo tiene un puerto de escritura
- Supongamos que permitimos que las inst. aritméticas escriban el resultado en el banco de registros en la etapa MEM



2.5.1 Riesgos estructurales
Unidades funcionales de más de un ciclo

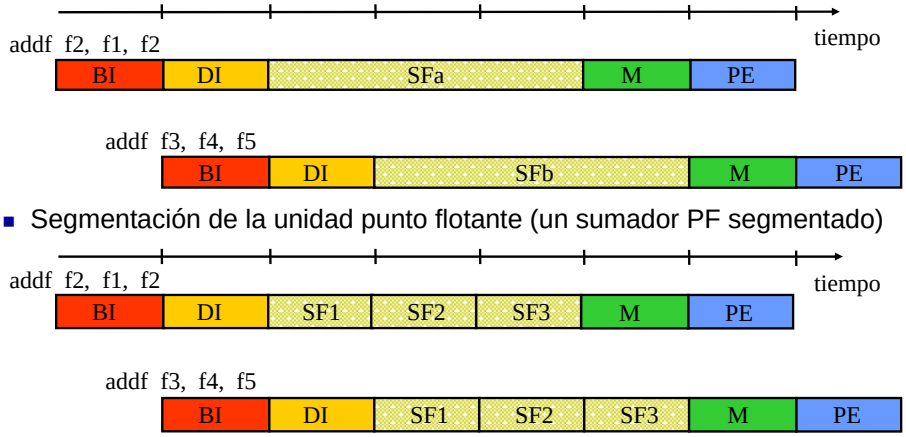
- Supongamos que la etapa de ejecución para la suma en punto flotante consume tres ciclos de reloj



- Burbuja: inhibir la actualización del PC y del reg de segmentación BI/DI.
- Soluciones
  - ✓ Replicación de la unidad de suma flotante.
  - ✓ Segmentación de la unidad de suma flotante

2.5.1 Riesgos estructurales
Unidades funcionales de más de un ciclo

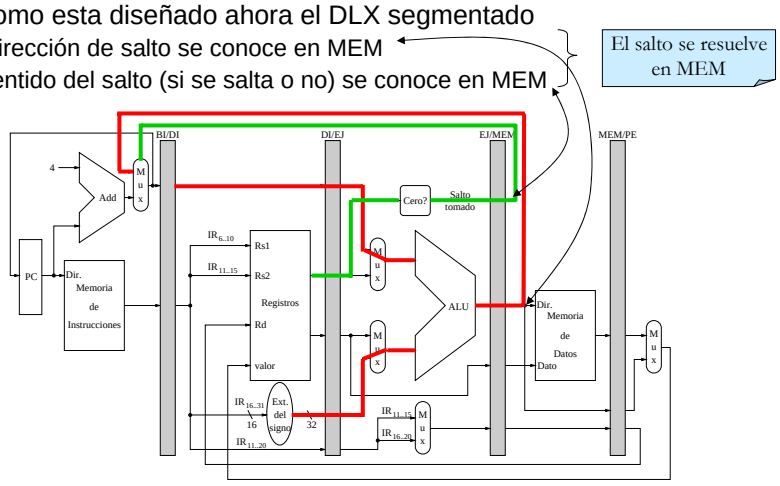
- Replicación de la unidad punto flotante (ahora hay 2 sumadores PF)



- Segmentación de la unidad punto flotante (un sumador PF segmentado)

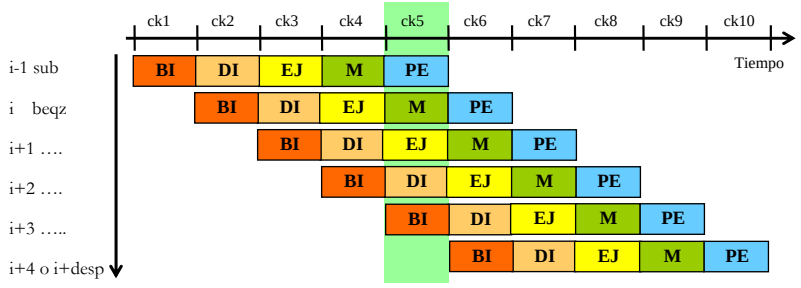
2.5.2 Riesgos de control

- Tal y como esta diseñado ahora el DLX segmentado
  - ✓ La dirección de salto se conoce en MEM
  - ✓ El sentido del salto (si se salta o no) se conoce en MEM



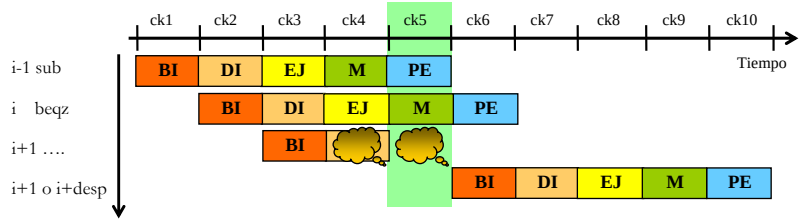
2.5.2 Riesgos de control

- En un procesador escalar (no segmentado) un salto condicional…
  - ✓ Decide si la siguiente instrucción a ejecutar es la i+1 o la i+desp
- En el DLX ha cambiado la semántica del salto
  - ✓ Ejecuta i+1, i+2 e i+3 y decide si la siguiente instrucción es la i+4 o la i+desp
  - ✓ Las instrucciones i+1, i+2 e i+3 ya se han "colado" en el cauce cuando se resuelve el salto:



2.5.2 Riesgos de control

- Si lo dejamos así, se llama procesador con salto retardado
  - ✓ No es transparente al programador en ensamblador ni al compilador
  - ✓ Los compiladores tienen que estar al tanto de la nueva semántica del salto
- Si no nos gusta esta diferencia con los procesadores escalares:
  - ✓ Solución más inmediata: detener el flujo de instrucciones hasta que se resuelva el salto

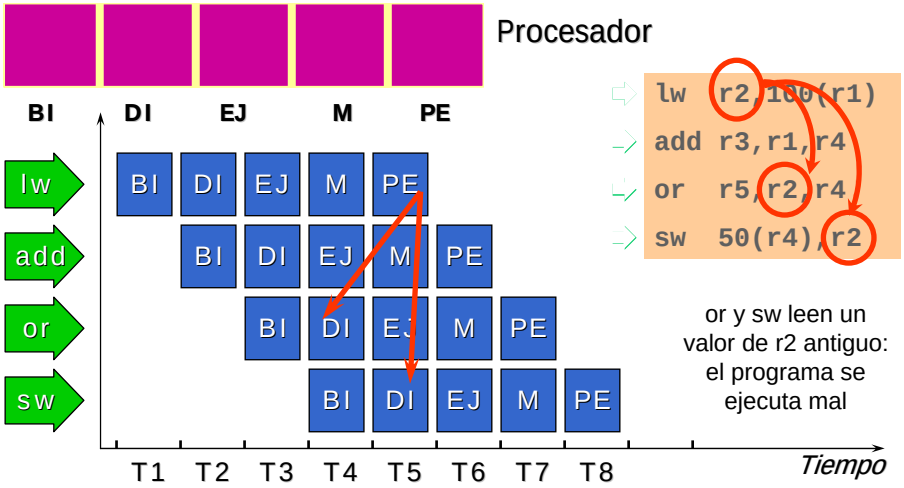


Se han perdido tres ciclos de reloj debido al salto

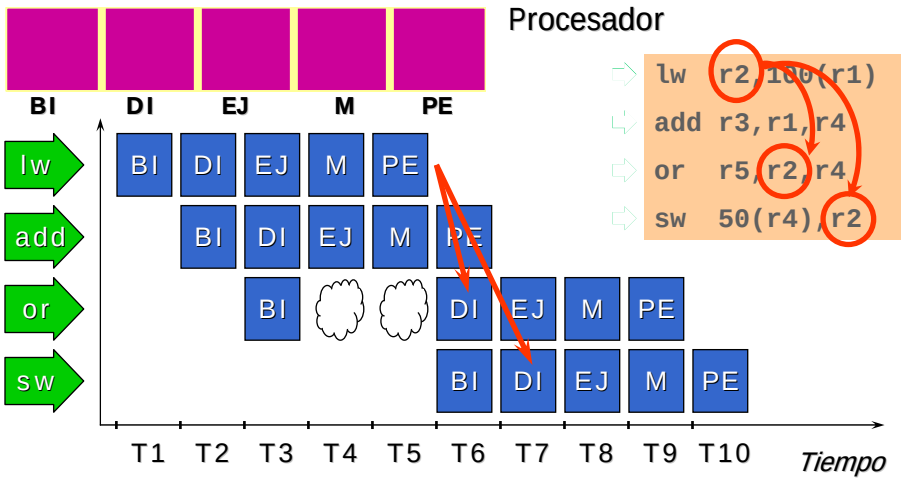
2.5.3 Riesgos de datos

- Derivados de dependencias de datos
- Dependencia verdadera → Riesgo RAW (Read After Write)
  - ✓ Una instrucción j intenta leer un operando que va a ser modificado por una instrucción previa i
- Antidependencia → Riesgo WAR (Write After Read)
  - ✓ Una instrucción j intenta escribir un destino antes de que éste sea leído por por una instrucción previa i.
- Dependencia de salida → Riesgo WAW (Write After Write)
  - ✓ Una instrucción j intenta escribir un operando antes de que sea escrito por una instrucción previa i.

2.5.3 Riesgos de datos
RAW. Ejemplo 1

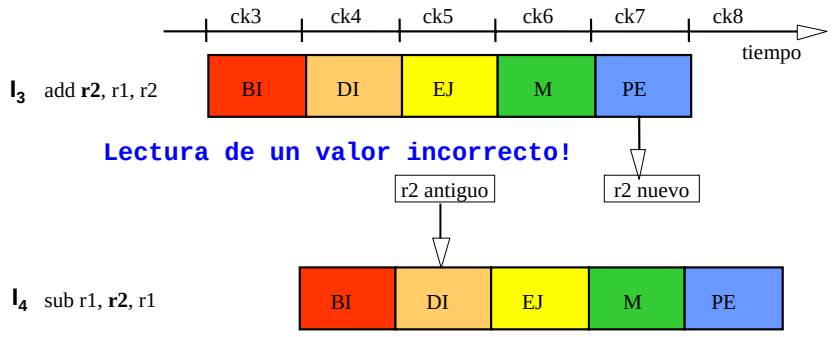


2.5.3 Riesgos de datos
RAW. Ejemplo 1: Solución con detenciones

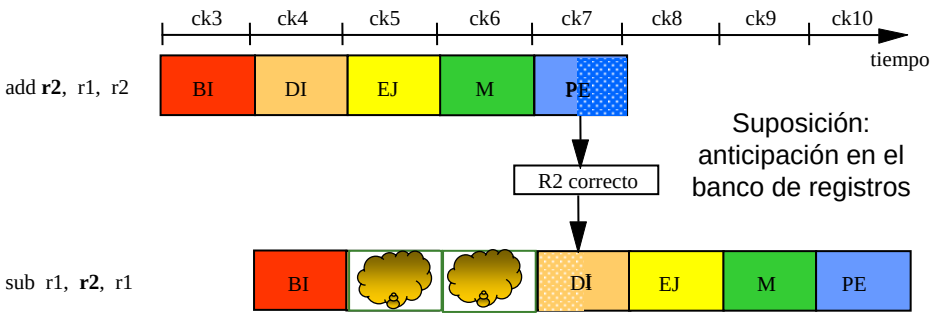


2.5.3 Riesgos de datos
RAW. Ejemplo 2.

- Mejora de la solución anterior en la implementación del DLX:
  - ✓ El valor se escribe en PE en la primera mitad del ciclo
  - ✓ El valor se lee en DI en la segunda mitad del ciclo



2.5.3 Riesgos de datos
RAW. Ejemplo 2.

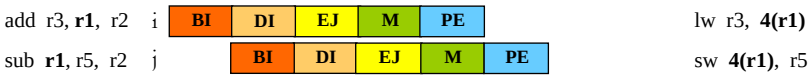


Pérdida de dos ciclos de reloj

- En el tema siguiente veremos otras soluciones para evitar la pérdida de ciclos: soluciones software y hardware.

2.5.3 Riesgos de datos
WAR

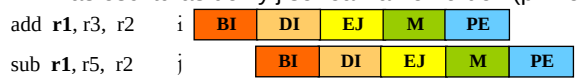
- No se dan en la ejecución de instrucciones en el DLX
- Aunque exista una antidependencia entre dos instrucciones i y j
  - ✓ La instrucción i siempre lee sus operandos antes de que j escriba



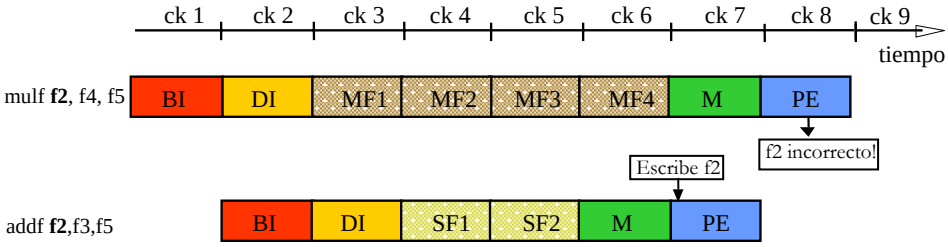
  - ✓ Operandos en registros: i lee de r1 y j escribe en r1.
    - La instrucción i lee en DI y j escribe en PE cuatro ciclos más tarde
  - ✓ Operandos en memoria:
    - i es un lw de una posición de memoria y j es un sw que escribe en la misma posición
    - Aunque hay una antidependencia entre i y j, no hay riesgo porque i lee primero y j escribe luego
- En otras arquitecturas si pueden aparecer riesgos WAR
  - ✓ Si se permiten escrituras al comienzo del cauce y lecturas al final
  - ✓ En arquitecturas con ejecución fuera de orden (a estudiar en tema 4)

2.5.3 Riesgos de datos WAW

- Si no existen UF multiciclo no hay riesgos WAW en el DLX
- Aunque exista dependencia de salida entre i y j
  - ✓ Las escrituras de i y j se realizan en orden (primero i y luego j)



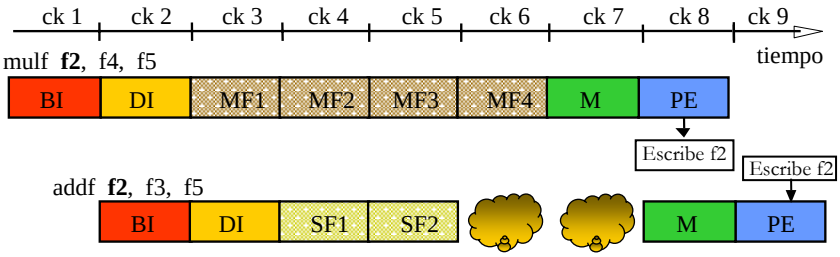
- Si existen UF multiciclo si pueden aparecer riesgos WAW
  - ✓ Ejemplo: un multiplicación PF (4ck) seguida de una suma PF (2ck)



Introducción Segmentación Control Dependencias Riesgos 45

2.5.3 Riesgos de datos WAW

- Posibles soluciones:
  - ✓ Detención de addf hasta que mulf escriba en el ciclo 8
    - Addf escribiría luego en el ciclo 9



  - ✓ Descartar la escritura de mulf (no escribir f2 en el ciclo 8)
    - Realmente la escritura de mulf es inútil
    - Pueden existir códigos con dependencias de salida y hay que tener en cuenta la posibilida de riesgos WAW

Introducción Segmentación Control Dependencias Riesgos 46

2.5.4 Como mejorar el rendimiento

- Hemos visto que los riesgos provocan perdidas de rendimiento
- Casi siempre hemos aplicado técnicas de detención
  - ✓ Se pierde ciclos de reloj sin hacer trabajo útil
  - ✓ Aumenta el CPI de las instrucciones
  - ✓ Por tanto aumenta el tiempo de ejecución de los programas
- En el tema siguiente veremos
  - ✓ Alternativas a la detención para resolver los riesgos
  - ✓ Evitar pérdida de ciclos de reloj en la medida de lo posible

Bibliografía

- J.L HENNESSY, D. PATTERSON. Computer Architecture: a quantitative approach. Ed. Morgan Kaufman,
  - ✓ Segunda Edición 1996.
    - Capítulo 3
  - ✓ Tercera Edición 2003
    - Apéndice A1
- D.A.PATTERSON, J.L.HENNESSY. Estructura y diseño de computadores. Ed. Reverté. 2000.
  - ✓ Capítulo 6