Diseño de la unidad de control

AMPLIACIÓN DE ESTRUCTURA DE COMPUTADORES

Daniel Mozos, José Luis Risco, Daniel Chaver
Facultad de Informática

contenidos

1. Introducción
2. Diseño de la ruta de datos monociclo
3. Diseño del controlador monociclo
4. Diseño de la ruta de datos multiciclo
5. Diseño del controlador multiciclo
6. Estudio comparativo: monociclo vs. multiciclo
7. Microprogramación
8. Tratamiento de excepciones
9. Introducción a la segmentación
10. Diseño de la ruta de datos segmentada
11. Diseño del controlador segmentado

Bibliografía:

"Computer organization and design. The hardware/software interface" D. Patterson, J. Hennessy, 3ª ed. Ed. Elsevier, 2005

"Principios de diseño digital" D.D. Gajski", Prentice Hall, 1997

1. introducción

Importancia del diseño del procesador

- El rendimiento de un computador está determinado por el tiempo que la CPU tarda en ejecutar programas:

$$\text{tiempo de CPU} = (\text{instrucciones por programa}) \times \text{CPI} \times (\text{tiempo de ciclo})$$

- El diseño del procesador determina:
  - La duración del ciclo de reloj
  - Número de ciclos de reloj por instrucción, en promedio
- Comúnmente estos dos factores tienen una relación inversa:
  - **Procesador monociclo**
    - 1 ciclo por instrucción
    - Tiempo de ciclo largo
  - **Procesador multiciclo**
    - Varios ciclos por instrucción
    - Tiempo de ciclo corto

1. introducción

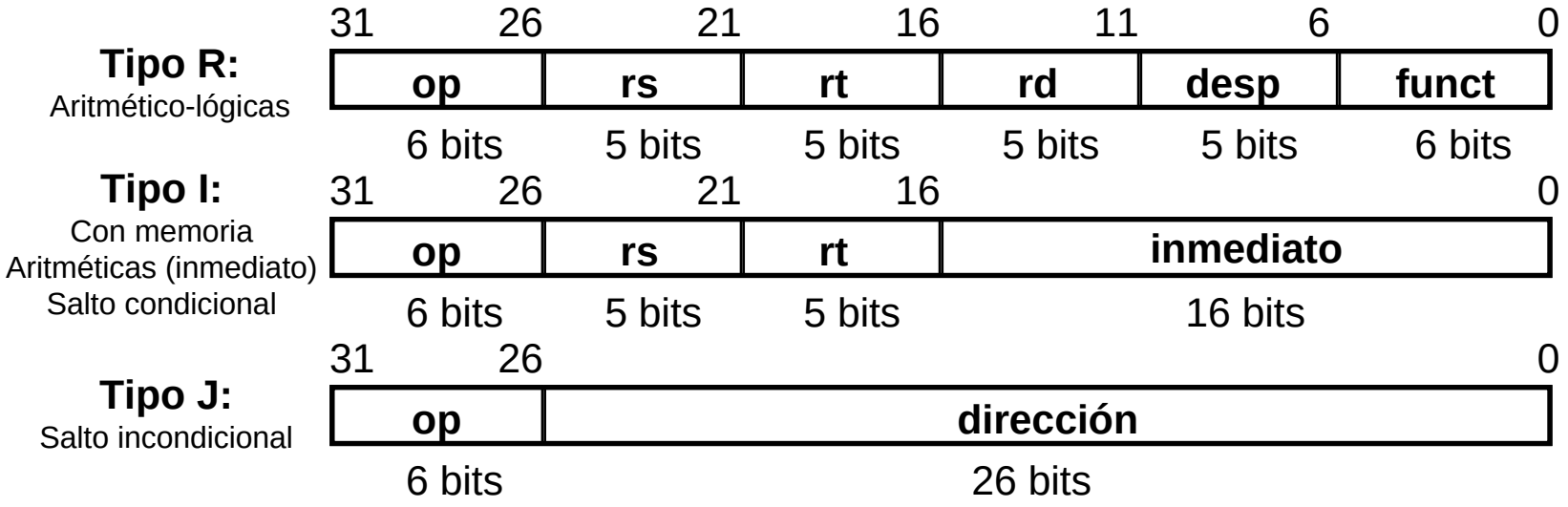
Metodología para el diseño de un procesador

- **Paso 1**: **Analizar el repertorio de instrucciones** para obtener los requisitos de la ruta de datos
  - La ruta de datos debe incluir tantos **elementos de almacenamiento** como registros sean visibles por el programador. Además puede tener otros elementos de almacenamiento transparentes.
  - La ruta de datos debe incluir tantos tipos de **elementos operativos** como tipos de operaciones de cálculo se indiquen en el repertorio de instrucciones
  - El significado de cada instrucción vendrá dado por un conjunto de transferencias entre registros. La ruta de datos debe ser capaz de soportar dichas transferencias.
- **Paso 2: Establecer la metodología de temporización**
  - **Monociclo (CPI = 1)**: todas las transferencias entre registros implicadas en una instrucción se realizan en un único ciclo de reloj.
  - **Multiciclo (CPI > 1)**: las transferencias entre registros implicadas en una instrucción se reparten entre varios ciclos de reloj.
- **Paso 3**: **Seleccionar el conjunto de módulos** (de almacenamiento, operativos e interconexión) que forman la ruta de datos.
- **Paso 4**: **Ensamblar la ruta de datos** de modo que se cumplan los requisitos impuestos por el repertorio, **localizando los puntos de control.**
- **Paso 5**: **Determinar los valores de los puntos de control** analizando las transferencias entre registros incluidas en cada instrucción.
- **Paso 6: Diseñar la lógica de control.**

1. introducción

Arquitectura MIPS: Formato de la instrucción máquina

⌦ Todas las instrucciones del repertorio del MIPS tienen 32 bits de anchura, repartidas en 3 formatos de instrucción diferentes:



⌦ El significado de los campos es:

- **op**: identificador de instrucción
- **rs**, **rt**, **rd**: identificadores de los registros fuentes y destino
- **desp**: cantidad a desplazar (en operaciones de desplazamiento)
- **funct**: selecciona la operación aritmética a realizar
- **inmediato**: operando inmediato o desplazamiento en direccionamiento a registro-base
- **dirección**: dirección destino del salto

1. introducción

Arquitectura MIPS: Subconjunto del repertorio de instrucciones

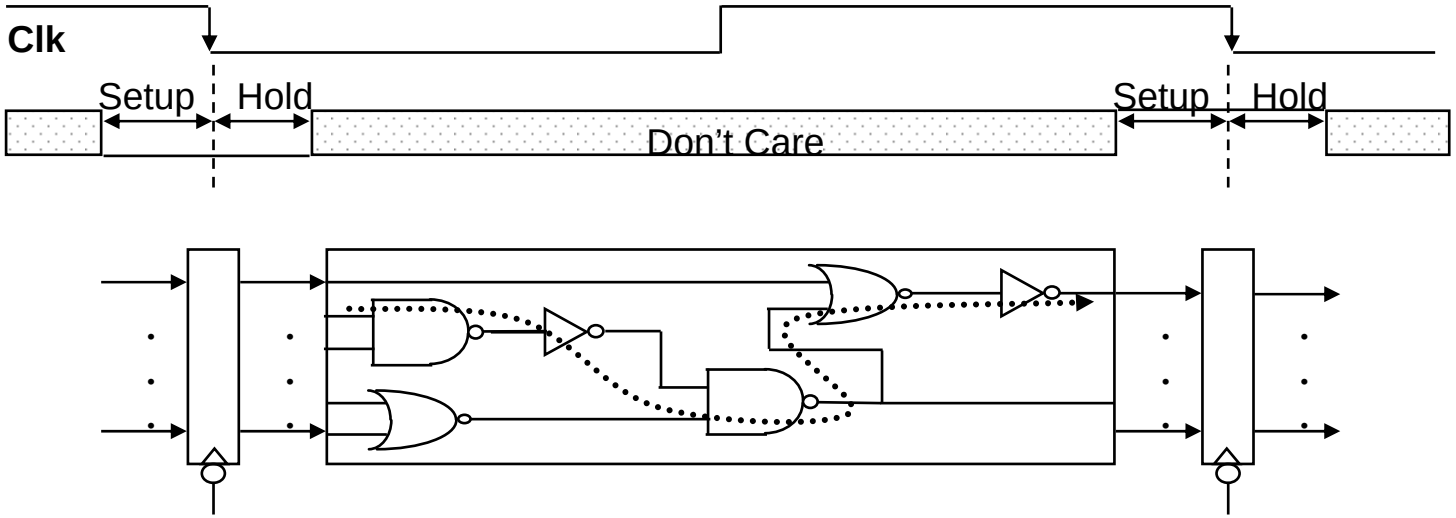
- **Instrucciones con referencia a memoria** (formato tipo I):
  - lw rt, inmed(rs) — rt ← Memoria(rs + SignExt(inmed)) , PC ← PC + 4
  - sw rt, inmed(rs) — Memoria(rs + SignExt(inmed)) ← rt , PC ← PC + 4
- **Instrucciones aritmético-lógicas con operandos en registros** (formato tipo R)
  - add rd, rs, rt — rd ← rs + rt, PC ← PC + 4
  - sub rd, rs, rt — rd ← rs - rt , PC ← PC + 4
  - and rd, rs, rt — rd ← rs and rt , PC ← PC + 4
  - or rd, rs, rt — rd ← rs or rt , PC ← PC + 4
  - slt rd, rs, rt — (si (rs < rt) entonces (rd ← 1) en otro caso (rd ← 0)), PC ← PC+4
- **Instrucciones de salto condicional** (formato tipo I)
  - beq rs, rt, inmed — si (rs = rt) entonces (PC ← PC + 4 + 4·SignExt(inmed)) en otro caso PC ← PC + 4

1. El ciclo de instrucción comienza buscando la instrucción en memoria (*fetch*)
   - instrucción ← Memoria(PC)
2. En función del tipo de instrucción se realiza una de las anteriores operaciones
3. Se vuelve a comenzar

2. diseño de la ruta de datos (monociclo)

Temporización monociclo

- Ejecución típica (de una instrucción)
  - Todos los registros se cargan simultáneamente (de modo selectivo)
  - Todos los valores se propagan a través de las redes combinacionales hasta estabilizarse en las entradas de los registros
  - Se repite indefinidamente el proceso
- Todos los elementos de almacenamiento están sincronizados al mismo flanco de reloj:
  - Tiempo de ciclo = CLK-to-Q + Camino con retardo máximo + Setup + Clock Skew
  - (CLK-to-Q + Camino con retardo mínimo - Clock skew) > Hold



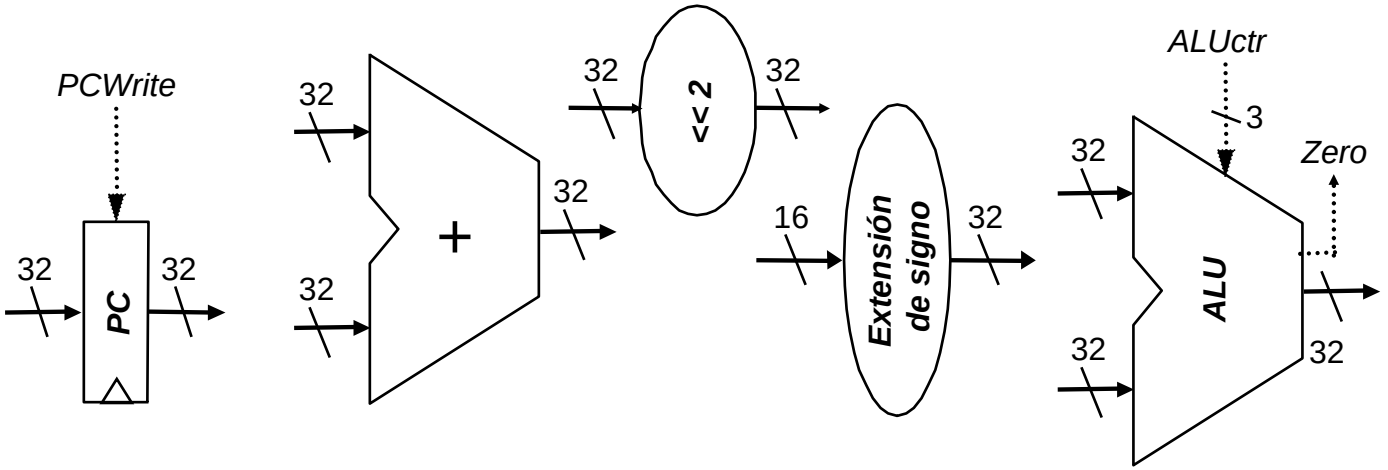
Setup y Hold: valores de tiempo en que la entrada a un elemento de almacenamiento debe permanecer estable antes y después, respectivamente, del flanco activo del reloj
CLK-to-Q: retardo de tiempo que media entre el flanco activo del reloj y la aparición de un nuevo valor en la salida
Skew: desvío de tiempo entre los relojes aplicados a dos elementos de almacenamiento diferentes

2. diseño de la ruta de datos (monociclo)

Componentes de la ruta de datos

⌦ Para implementar el subconjunto del repertorio del MIPS en una implementación monociclo se requieren:

- **Memoria de instrucciones**
- **Memoria de datos**
- **32 registros de datos:** visibles por el programador.
- **Contador de programa**
- **2 Sumadores:** para sumar 4 al PC, y para sumar al PC el valor inmediato de salto.
- **ALU:** capaz de realizar suma, resta, and, or, comparación de mayoría e indicación de que el resultado es cero (para realizar la comparación de igualdad mediante resta)
- **Extensor de signo:** para adaptar el operando inmediato de 16 bits al tamaño de palabra.
- **Desplazador a la izquierda:** para implementar la multiplicación por 4.

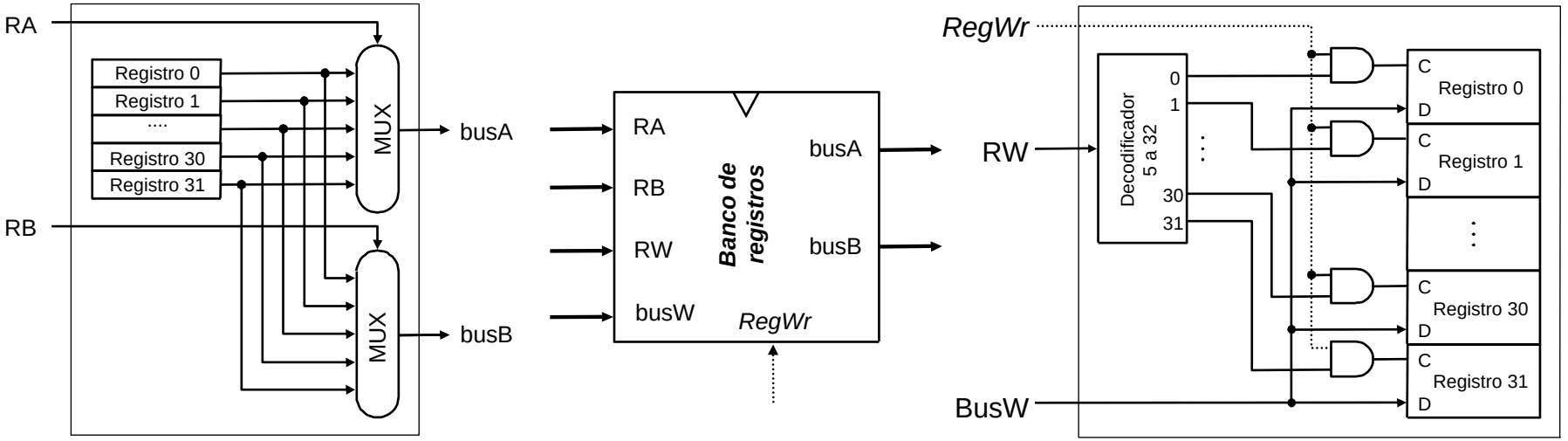


ALUctr	*función*
000	A and B
001	A or B
010	A + B
110	A – B
111	1 si (A<B), sino 0

2. diseño de la ruta de datos (monociclo)

Componentes de la ruta de datos (cont.)

- Los 32 registros se almacenan en un **banco de registros**. Dado que en las instrucciones de tipo R, se requiere un acceso simultáneo a 3 registros:
  - 2 salidas de datos de 32 bits
  - 1 entrada de datos de 32 bits
  - 3 entradas de 5 bits para la identificación de los registros
  - 1 entrada de control para habilitar la escritura sobre uno de los registros
  - 1 puerto de reloj (sólo determinante durante las operaciones de escritura, las de lectura son combinacionales)

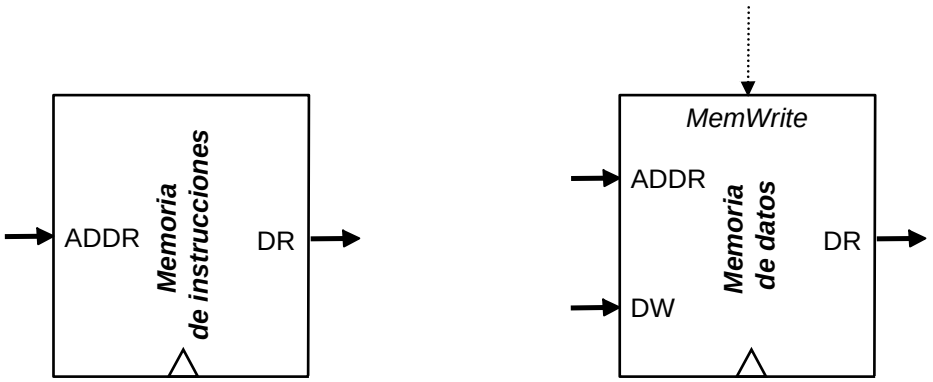


Mecanismo de lectura ***Mecanismo de escritura***

2. diseño de la ruta de datos (monociclo)

Componentes de la ruta de datos (cont.)

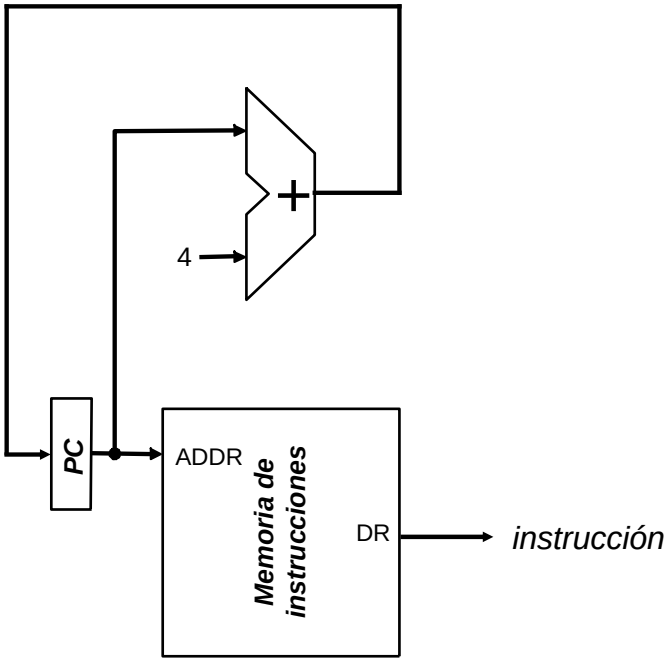
- La memoria tendrá un comportamiento idealizado.
  - "Integrada" dentro de la CPU.
  - Direccionable por bytes, pero capaz de aceptar/ofrecer 4 bytes por acceso
    - 1 entrada de dirección
    - 1 salida de datos de 32 bits
    - 1 entrada de datos de 32 bits (sólo en la de datos)
  - Se supondrá que se comporta temporalmente como el banco de registros (síncronamente) y que tiene un tiempo de acceso menor que el tiempo de ciclo
  - Se supondrá dividida en dos para poder hacer dos accesos a memoria en el mismo ciclo:
    - Memoria de instrucciones
    - Memoria de datos
- 1 entrada de control, MemWrite para seleccionar la operación de lectura/escritura sobre la memoria de datos



2. diseño de la ruta de datos (monociclo)

Ensamblaje de la ruta de datos

- La **búsqueda de instrucciones** implica:
  - Leer la instrucción ubicada en la dirección de la **memoria de instrucciones** indicada por el **contador de programa**.
- La **ejecución secuencial** de programas implica:
  - Actualizar el **contador de programa** para que apunte a la siguiente instrucción (sumando 4 por ser una memoria direccionable por bytes y una arquitectura con tamaño de palabra de 32 bits)

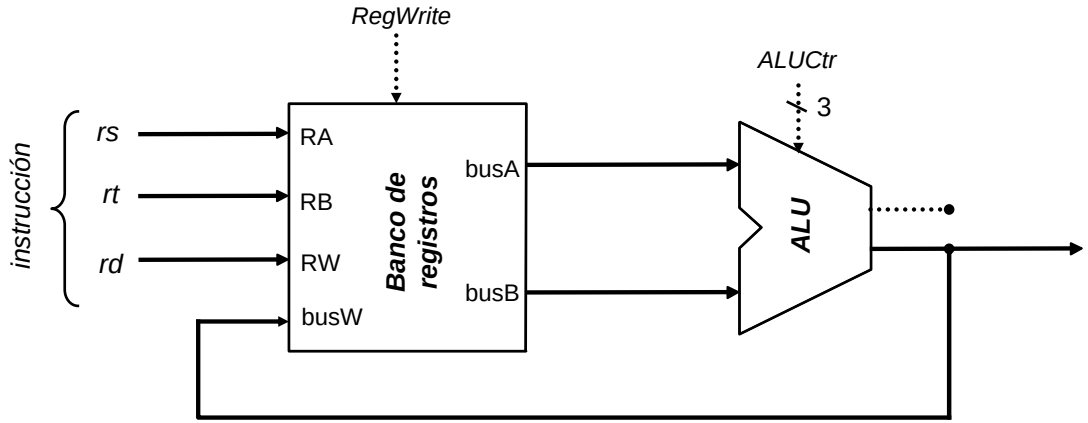


2. diseño de la ruta de datos (monociclo)

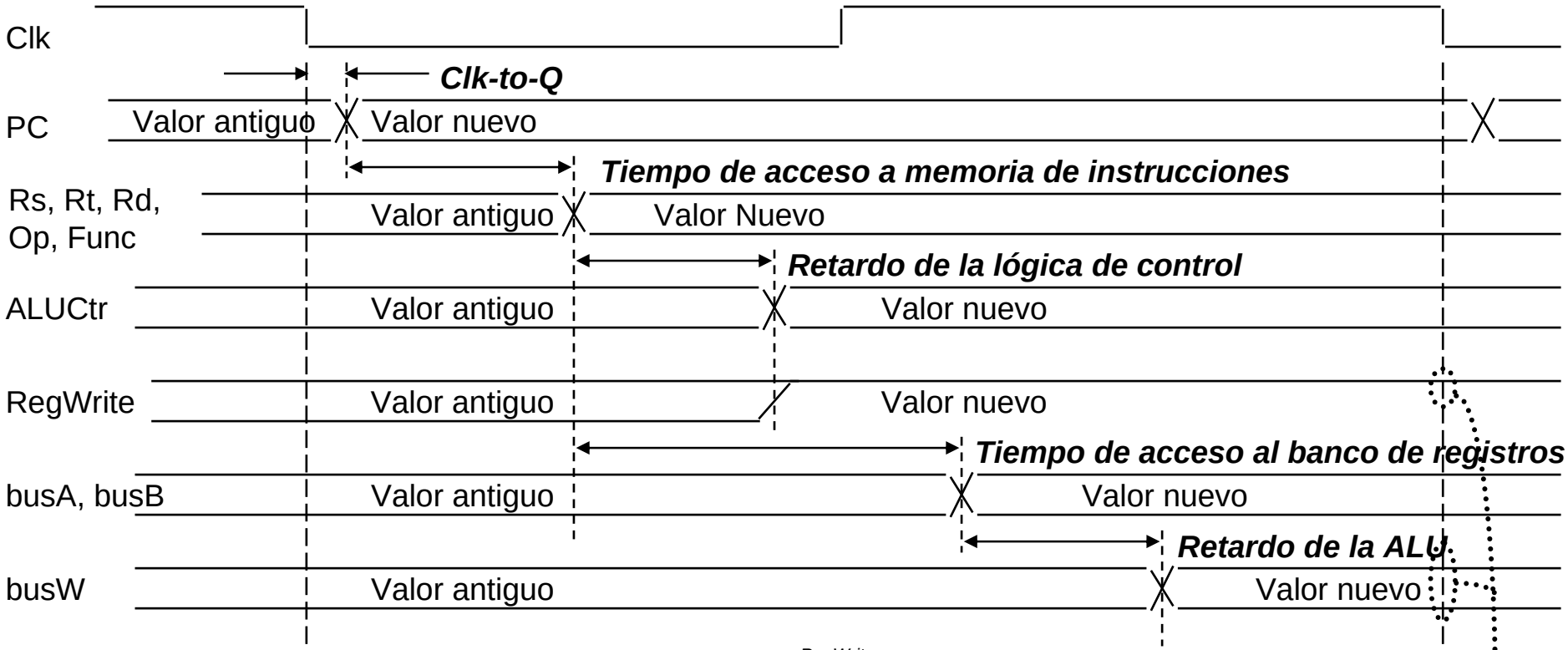
Ensamblaje de la ruta de datos (cont.)

⌦ Las instrucciones **aritmético lógicas** (tipo-R) implican:

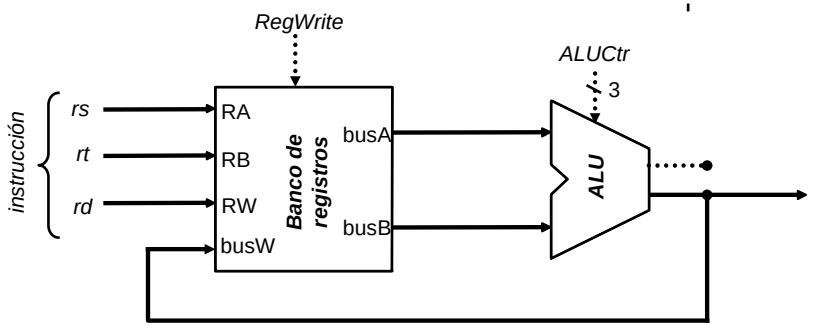
- **BR(rd) ← BR(rs) funct BR(rt)**
- Leer dos registros cuyos identificadores se ubican en los campos **rs** y **rt** de la instrucción:
- Operar sobre ellos según el contenido del campo de código de operación aritmética (**funct**) de la instrucción
- Almacenar el resultado en otro registro cuyo identificador se localiza en el campo **rd** de la instrucción



2. diseño de la ruta de datos (monociclo)



cronograma de una operación arimético-lógica



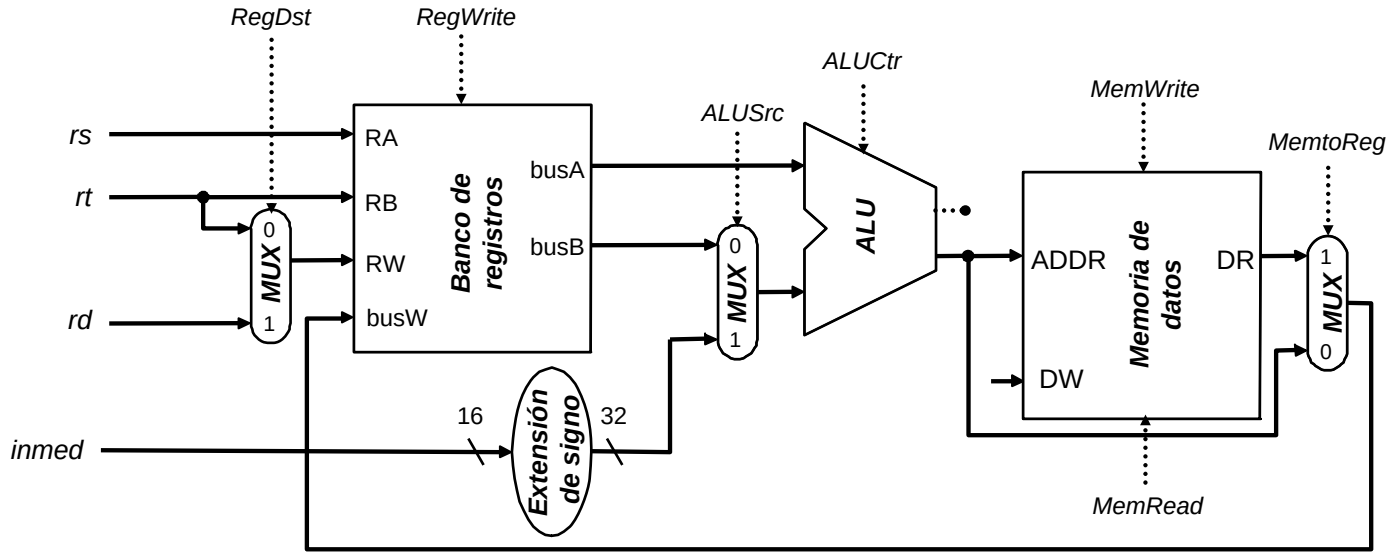
aquí se escribe el registro

2. diseño de la ruta de datos (monociclo)

Ensamblaje de la ruta de datos (cont.)

⌦ La **instrucción de carga** (lw) implica:

- **BR(rt) ← Memoria(BR(rs) + SignExt(inmed))**
- Calcular la dirección efectiva de memoria:
  - ⇨ Leyendo el *registro base* cuyo identificador se ubica en el campo **rs** de la instrucción
  - ⇨ Obteniendo un *desplazamiento* de 32 bits a partir de la **extensión** del campo de operando inmediato (**inmed**) de la instrucción
  - ⇨ **Sumando** base y desplazamiento.
- Leer dato ubicado en la **memoria de datos** cuya dirección es la anteriormente calculada
- Almacenar el dato leído de memoria en el registro cuyo identificador se especifica en el campo **rt** de la instrucción

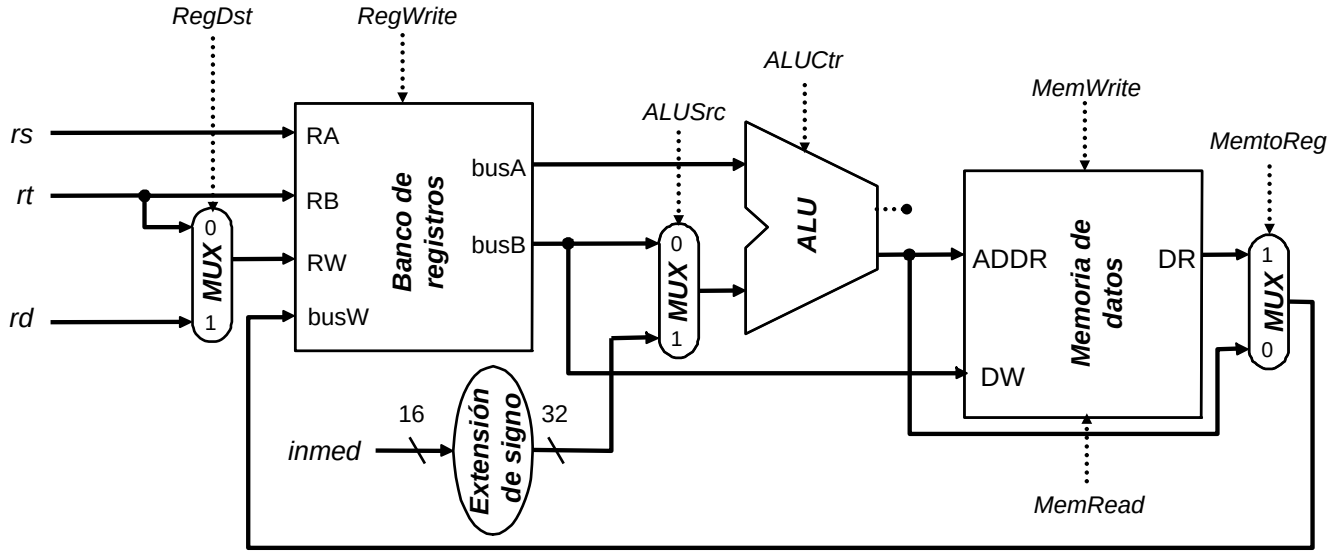


2. diseño de la ruta de datos (monociclo)

Ensamblaje de la ruta de datos (cont.)

⌦ La **instrucción de almacenaje** (sw) implica:

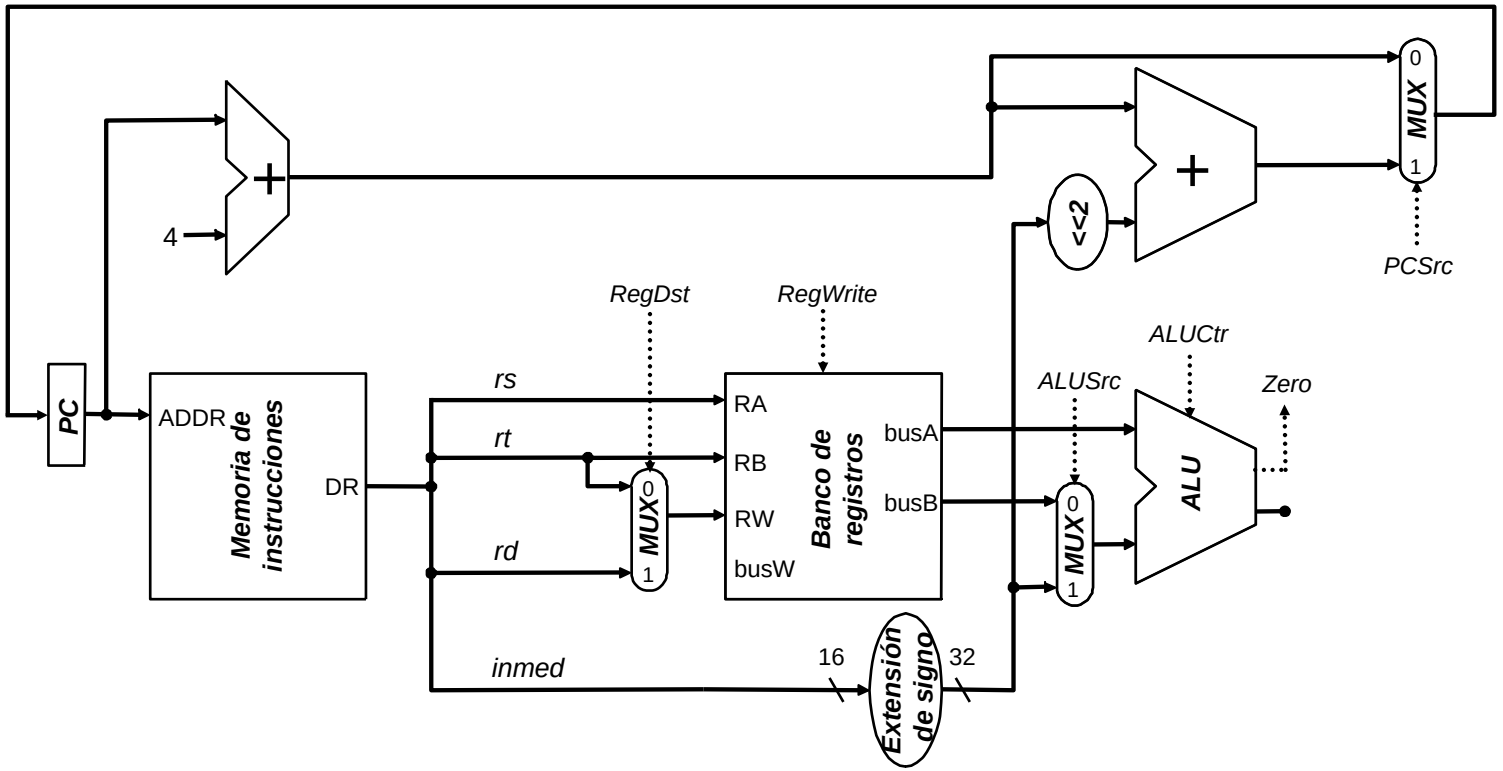
- Memoria(BR(rs) + SignExt(inmed)) ← BR(rt)
- Leer el dato almacenado en el registro cuyo identificador se especifica en el campo **rt** de la instrucción
- Calcular la dirección efectiva de memoria:
  - ⇨ Leyendo el *registro base* cuyo identificador se ubica en el campo **rs** de la instrucción
  - ⇨ Obteniendo un *desplazamiento* de 32 bits a partir de la **extensión** del campo de operando inmediato (**inmed**) de la instrucción
  - ⇨ **Sumando** base y desplazamiento.
- Almacenar el dato leído en la **memoria de datos** en la dirección anteriormente calculada



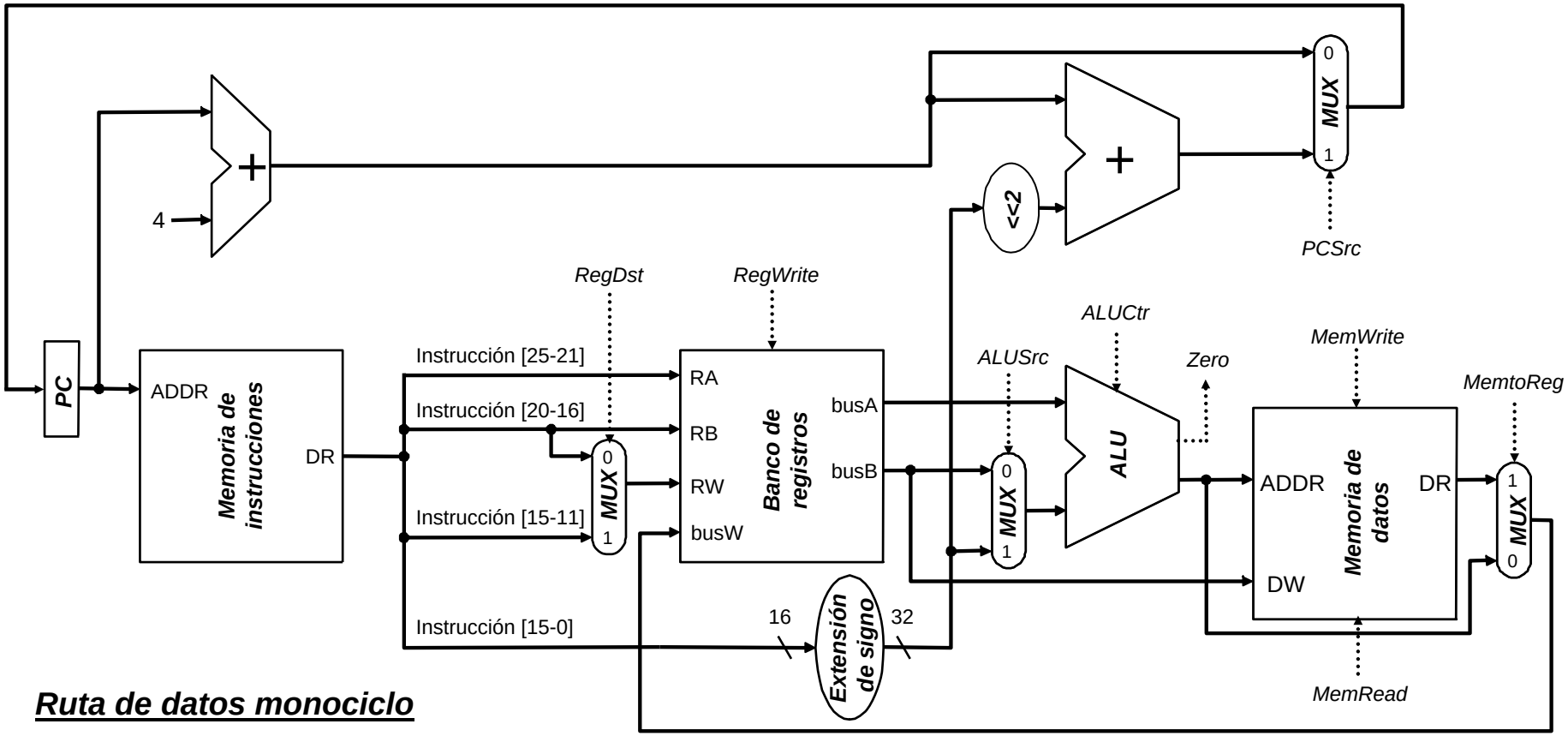
2. diseño de la ruta de datos (monociclo)

Ensamblaje de la ruta de datos (cont.)

- La **instrucción de salto condicional** (beq) implica
  - si (BR(rs) = BR(rt)) entonces (PC ← PC + 4·SignExt(inmed))
  - Leer dos registros cuyos identificadores se ubican en los campos **rs** y **rt** de la instrucción:
  - Comparar la igualdad de sus contenidos y en función del resultado:
    - No hacer nada o
    - **Sumar** al contador del programa un *desplazamiento* de 32 bits obtenido a partir de la **extensión** del campo de operando inmediato (**inmed**) de la instrucción desplazado 2 posiciones a la izquierda



2. diseño de la ruta de datos (monociclo)



Ruta de datos monociclo

⌦ La ejecución monociclo ha obligado a:

- No usar más de una vez por instrucción cada recurso. Duplicarlo si es necesario
- Memoria de instrucciones y datos separadas
- Añadir multiplexores cuando un valor pueda provenir de varias fuentes

3. diseño del controlador (monociclo)

Determinación de los valores de los puntos de control

⌦ La tarea del **controlador** es:

- Seleccionar las operaciones a realizar por los módulos multifunción (ALU, read/write, ...)
- Controlar el flujo de datos, activando la entrada de selección de los multiplexores y la señal de carga de los registros

Instrucción de carga (lw)

rt ← Memoria(rs + SignExt(inmed)), PC ← PC + 4

RegDest ← 0, RegWrite ← 1, ALUsrc ← 1, ALUctr ← 010, PCSrc ← 0, MemWrite ← 0, MemRead ← 1, MemtoReg ← 1

Instrucción de almacenaje (sw)

Memoria(rs + SignExt(inmed)) ← rs, PC ← PC + 4

RegDest ←X, RegWrite ← 0, ALUsrc ← 1, ALUctr ← 010, PCSrc ← 0, MemWrite ← 1, MemRead ← 0, MemtoReg ← X

Instrucción *and*

rd ← rs and rt, PC ← PC + 4

RegDest ← 1, RegWrite ← 1, ALUsrc ← 0, ALUctr ← 000, PCSrc ← 0, MemWrite ← 0, MemRead ← 0, MemtoReg ← 0

Instrucción de salto condicional (beq)

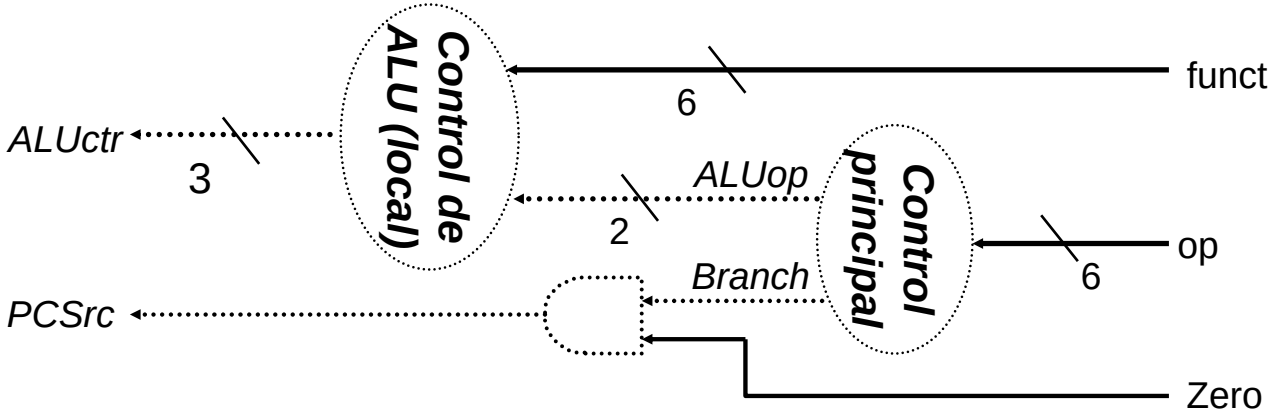
si (rs = rt) entonces (PC ← PC + 4 + 4·SignExp(inmed)) en otro caso PC ← PC + 4

RegDest ←X, RegWrite ← 0, ALUsrc ← 0, ALUctr ← 110, PCSrc ← Zero, MemWrite ← 0, MemRead ← 0, MemtoReg ← X

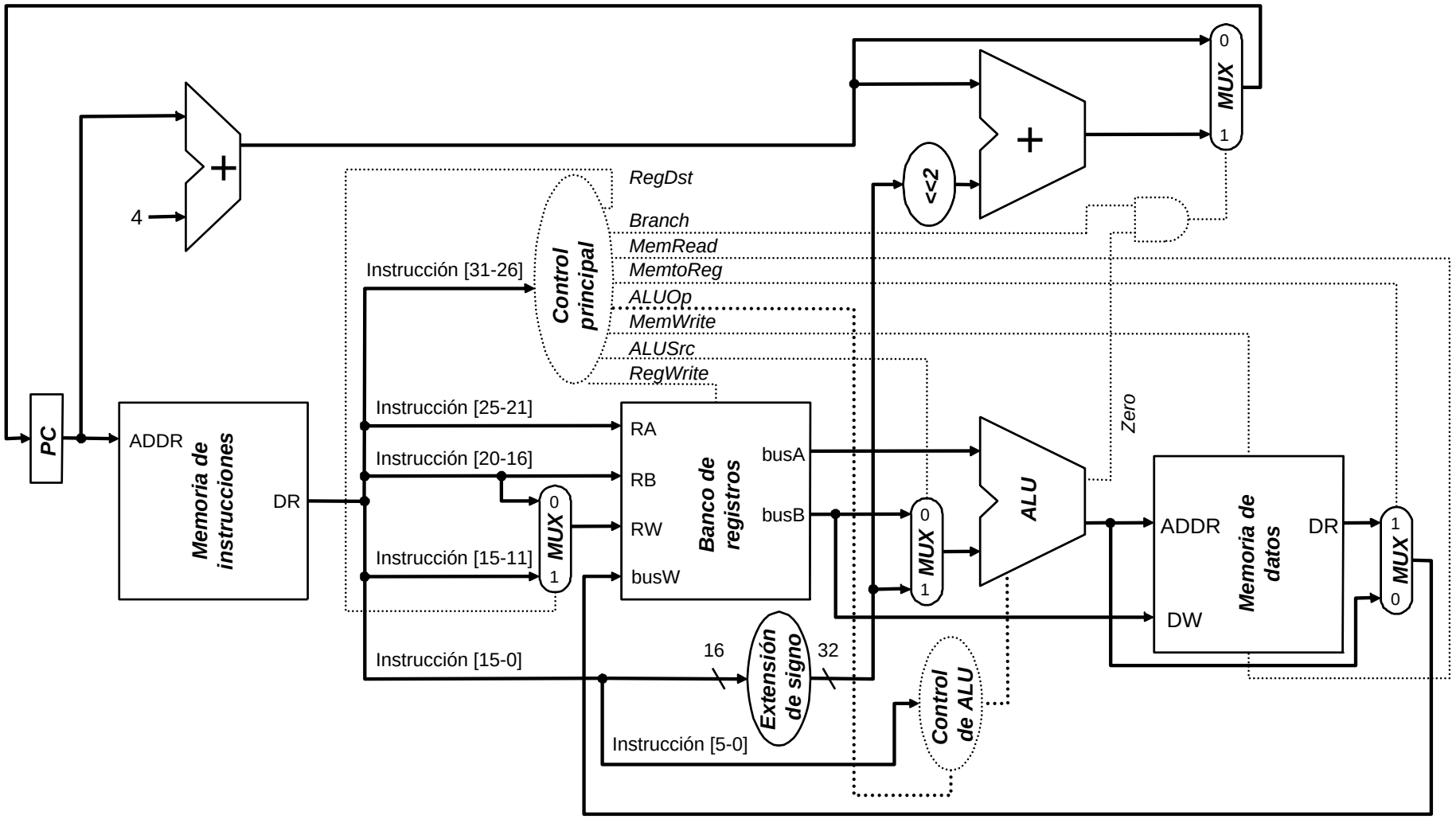
3. diseño del controlador (monociclo)

Control global vs. Control local

- Todas las operaciones aritméticas comparten el mismo código de operación y durante su ejecución todas las señales generales de la ruta de datos son iguales. Por ello, utilizaremos:
  - Un **control principal** para decodificar el campo de código de operación (**op**) y configurar globalmente la ruta de datos
  - Un **control local** a la ALU que decodifique el campo de operación aritmética (**funct**) y seleccione la operación que debe realizar la ALU
  - Adicionalmente en operaciones no aritméticas (lw, sw y beq) el control principal puede ordenar alguna operación a la ALU para calcular las DE o realizar comparaciones.
- Utilizaremos la señal intermedia ***ALUop*** cuyo valor será:
  - 00 en operaciones con acceso a memoria
  - 01 en operaciones de salto
  - 10 en operaciones aritméticas
- Del mismo modo para controlar qué dirección debe cargar el PC se utilizará una señal intermedia ***Branch*** (activada durante la instrucción beq) a la que se hará la y-lógica con la señal ***Zero*** que genera la ALU.



3. diseño del controlador (monociclo)

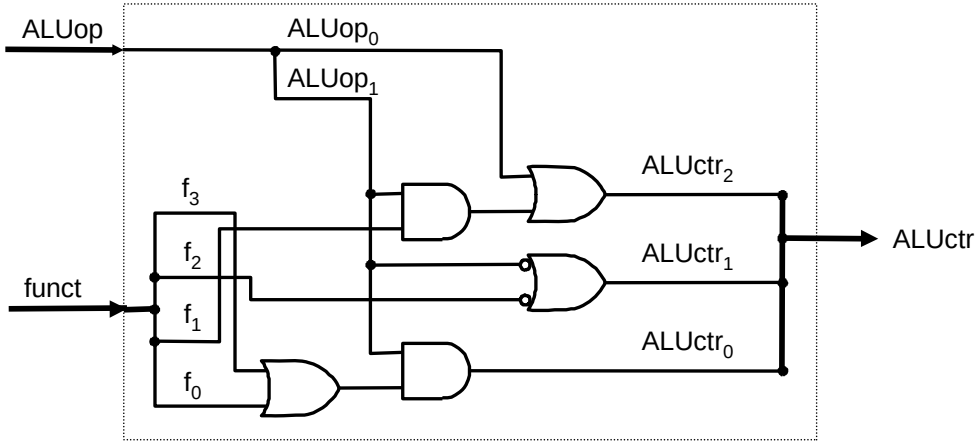


Ruta de datos monociclo + controlador

3. diseño del controlador (monociclo)

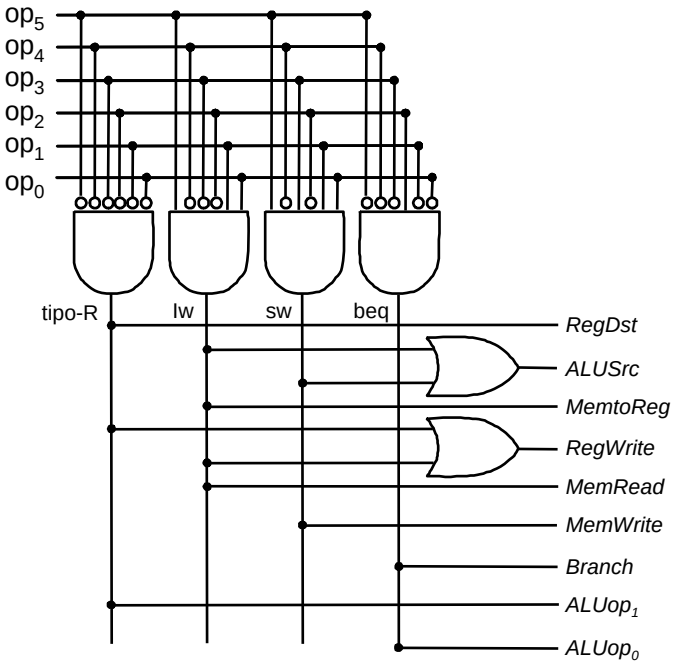
Control de la ALU

op	funct	ALUop	ALUctr
100011 (lw)	XXXXXX	00	010
101011 (sw)		00	010
000100 (beq)		01	110
000000 (tipo-R)	100000 (add)	10	010
	100010 (sub)	10	110
	100100 (and)	10	000
	100101 (or)	10	001
	101010 (slt)	10	111

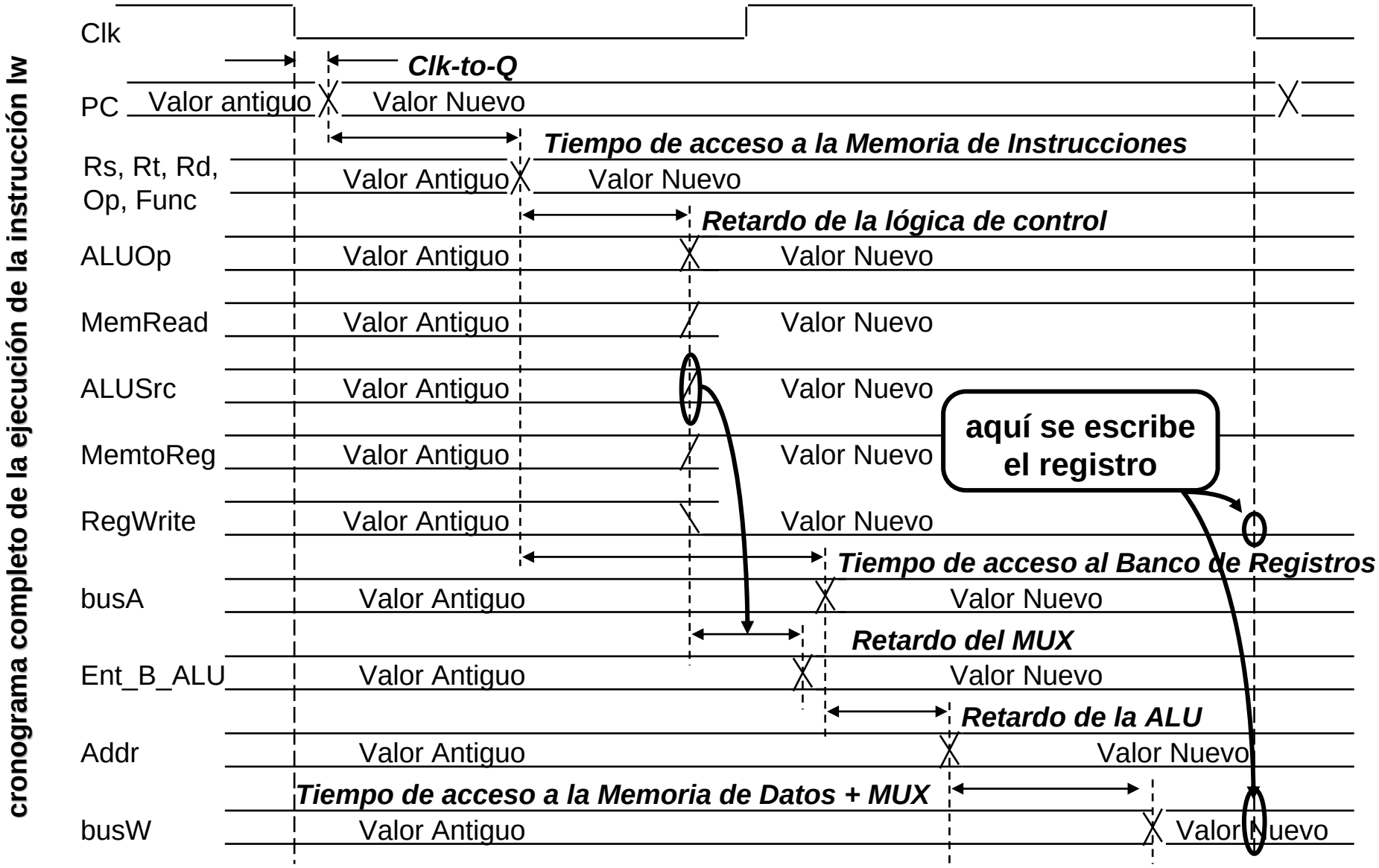


Control principal

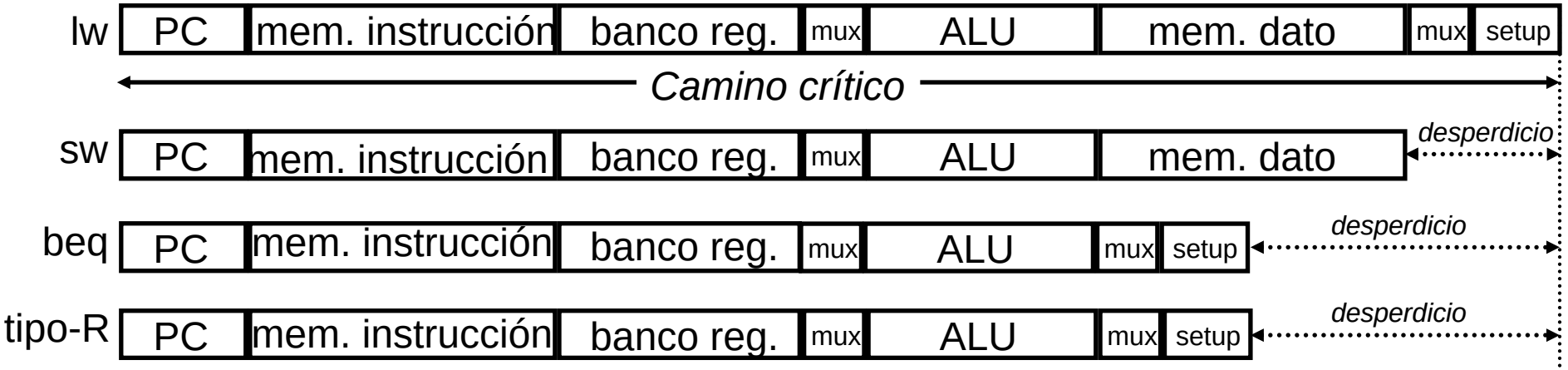
op	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Brach	ALUop
100011 (lw)	0	1	1	1	1	0	0	00
101011 (sw)	X	1	X	0	0	1	0	00
000100 (beq)	X	0	X	0	0	0	1	01
000000 (tipo-R)	1	0	0	1	0	0	0	10



4. diseño de la ruta de datos (monociclo)

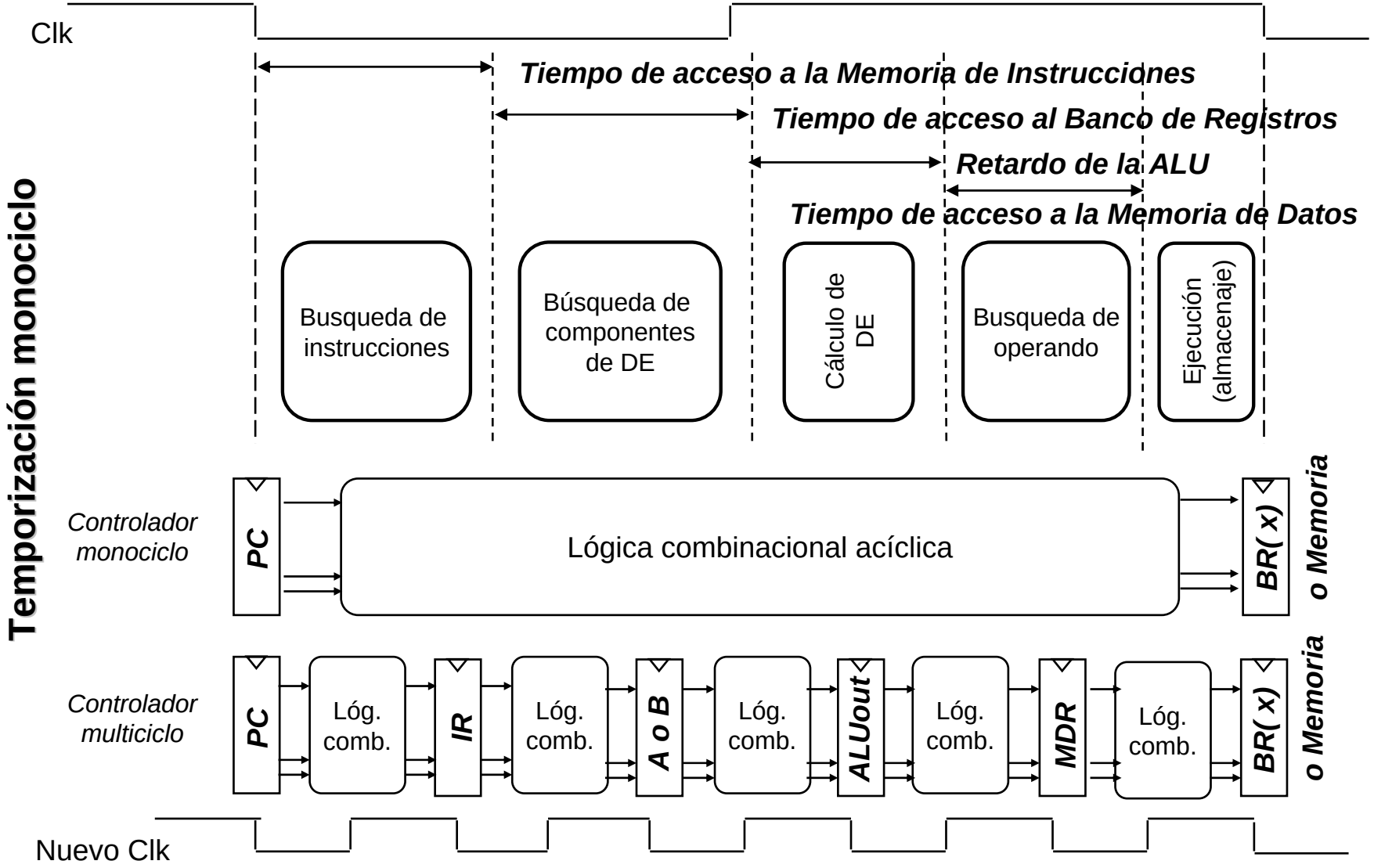


4. diseño de la ruta de datos (multiciclo)

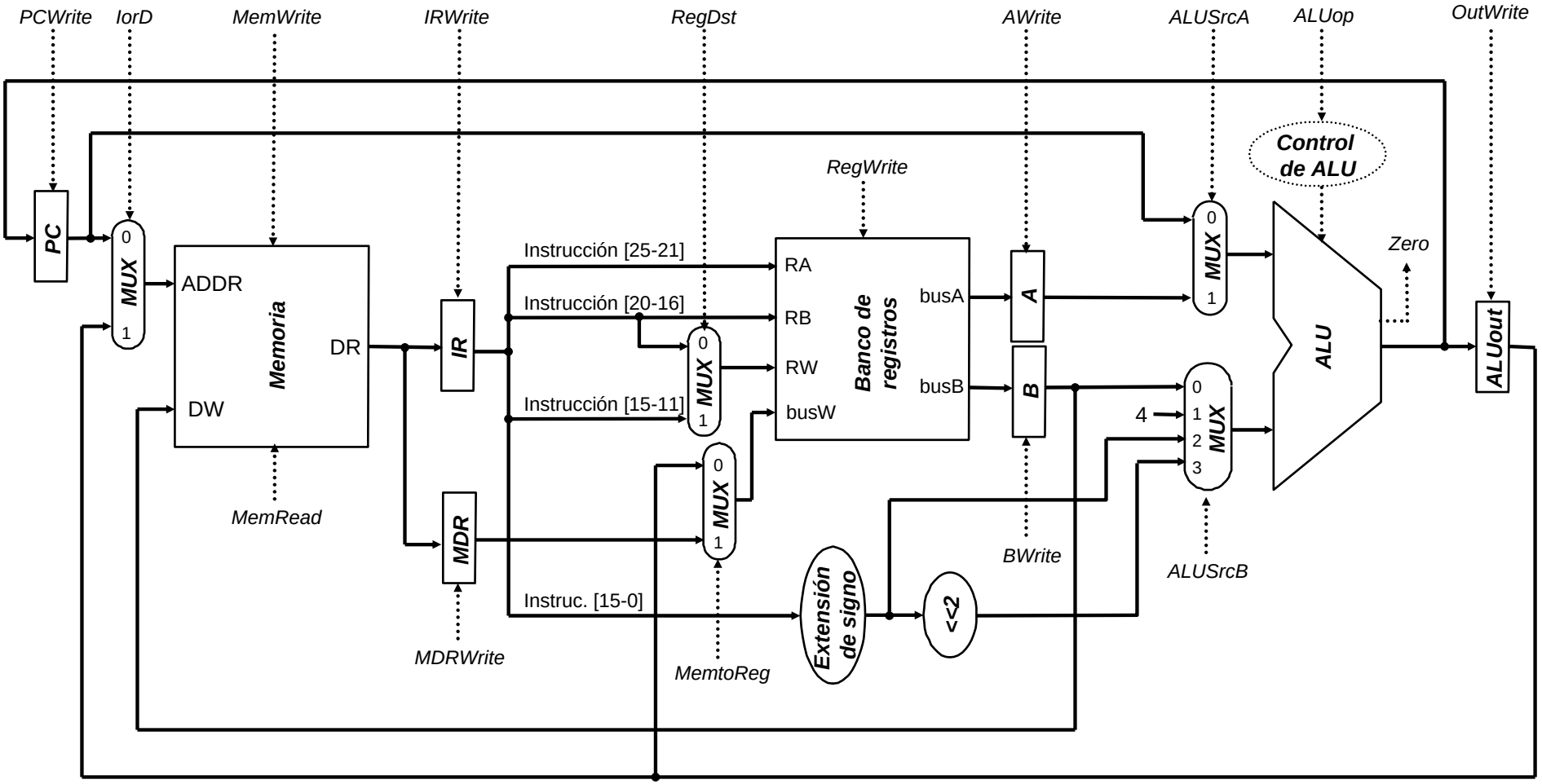


- **Problema**: en un controlador monociclo:
  - **El reloj debe tener igual periodo que la instrucción más lenta**
    - Dado que dicho periodo es fijo, en las instrucciones rápidas se desperdicia tiempo.
    - En repertorios reales, existen instrucciones muy largas: aritmética en punto flotante, modos de direccionamiento complejos, etc.
  - **No se puede reusar hardware**
    - Si en una instrucción se necesitara hacer 4 sumas (resolver los 3 modos de direccionamiento de los operandos y sumarlos) se necesitarían 4 sumadores.
- **Solución**: dividir la ejecución de la instrucción en varios ciclos más pequeños:
  - Cada instrucción usará el número de ciclos que necesite.
  - Un mismo elemento hardware puede ser utilizado varias veces en la ejecución de una instrucción si se hace en ciclos diferentes.
  - Se requieren elementos adicionales para almacenar valores desde el ciclo en que se calculan hasta el ciclo en que se usan.

4. diseño de la ruta de datos (multiciclo)

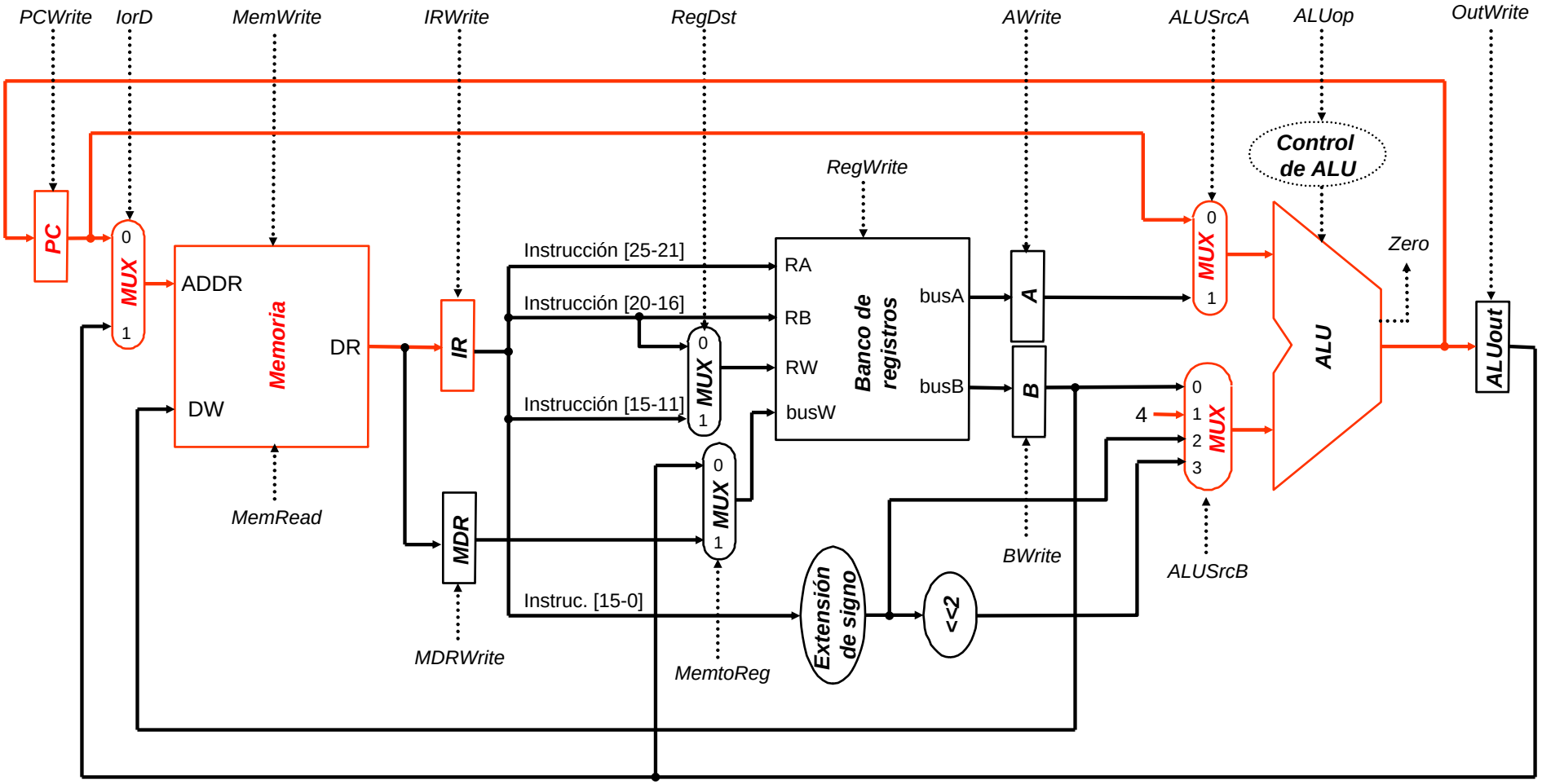


4. diseño de la ruta de datos (multiciclo)



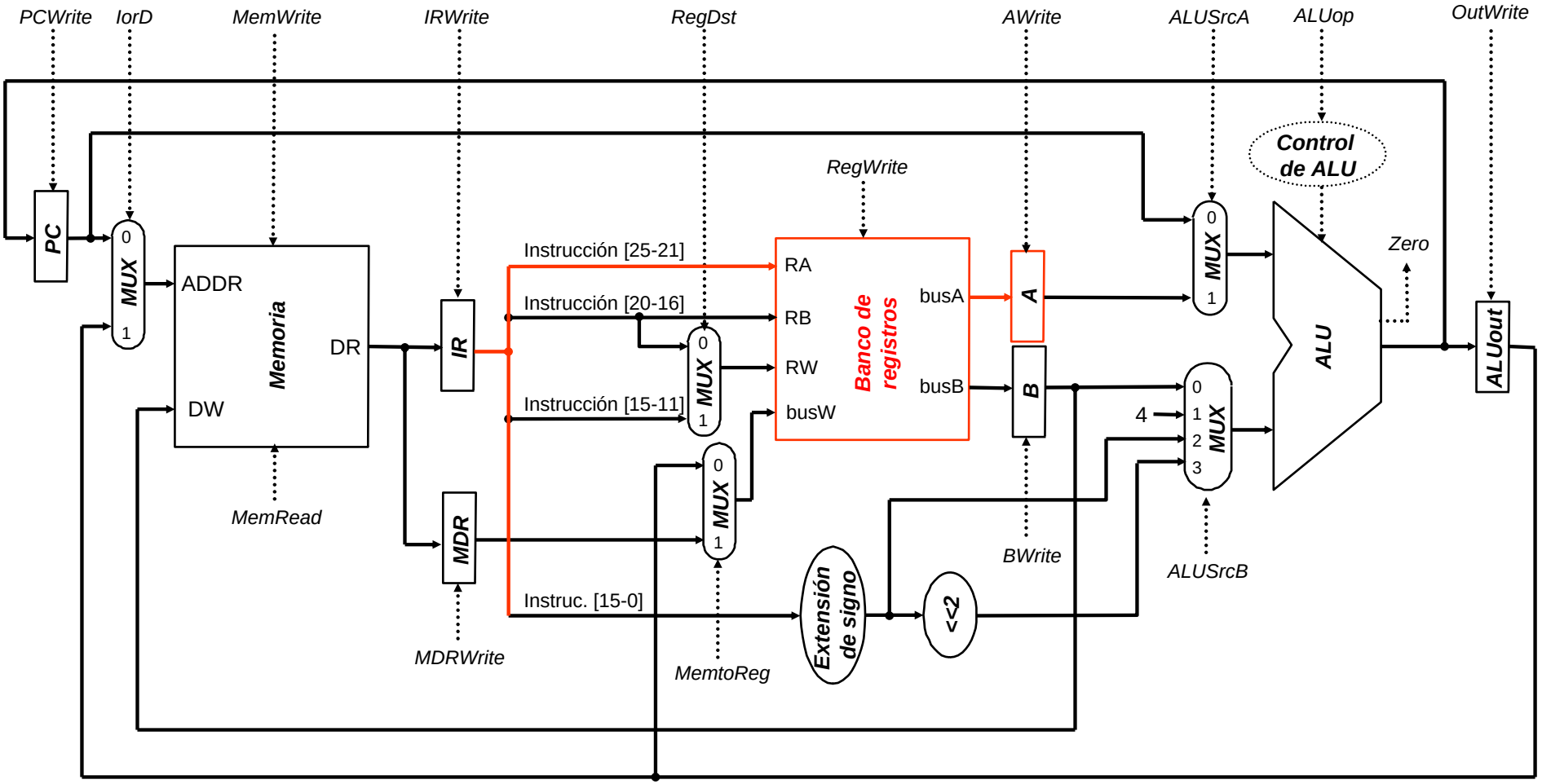
Ruta de datos multiciclo

4. diseño de la ruta de datos (multiciclo)



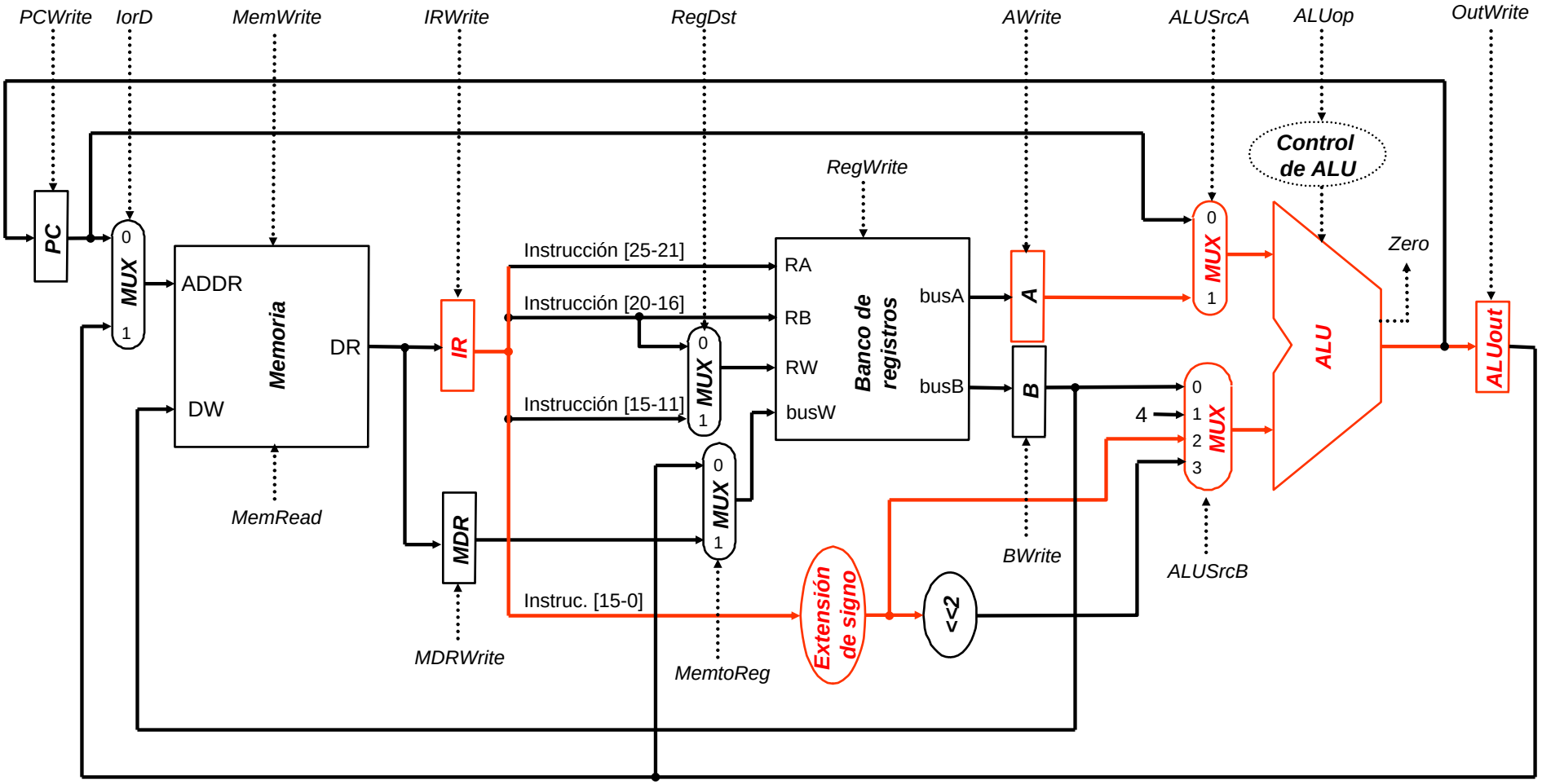
Ejecución del primer ciclo de una instrucción LW

4. diseño de la ruta de datos (multiciclo)



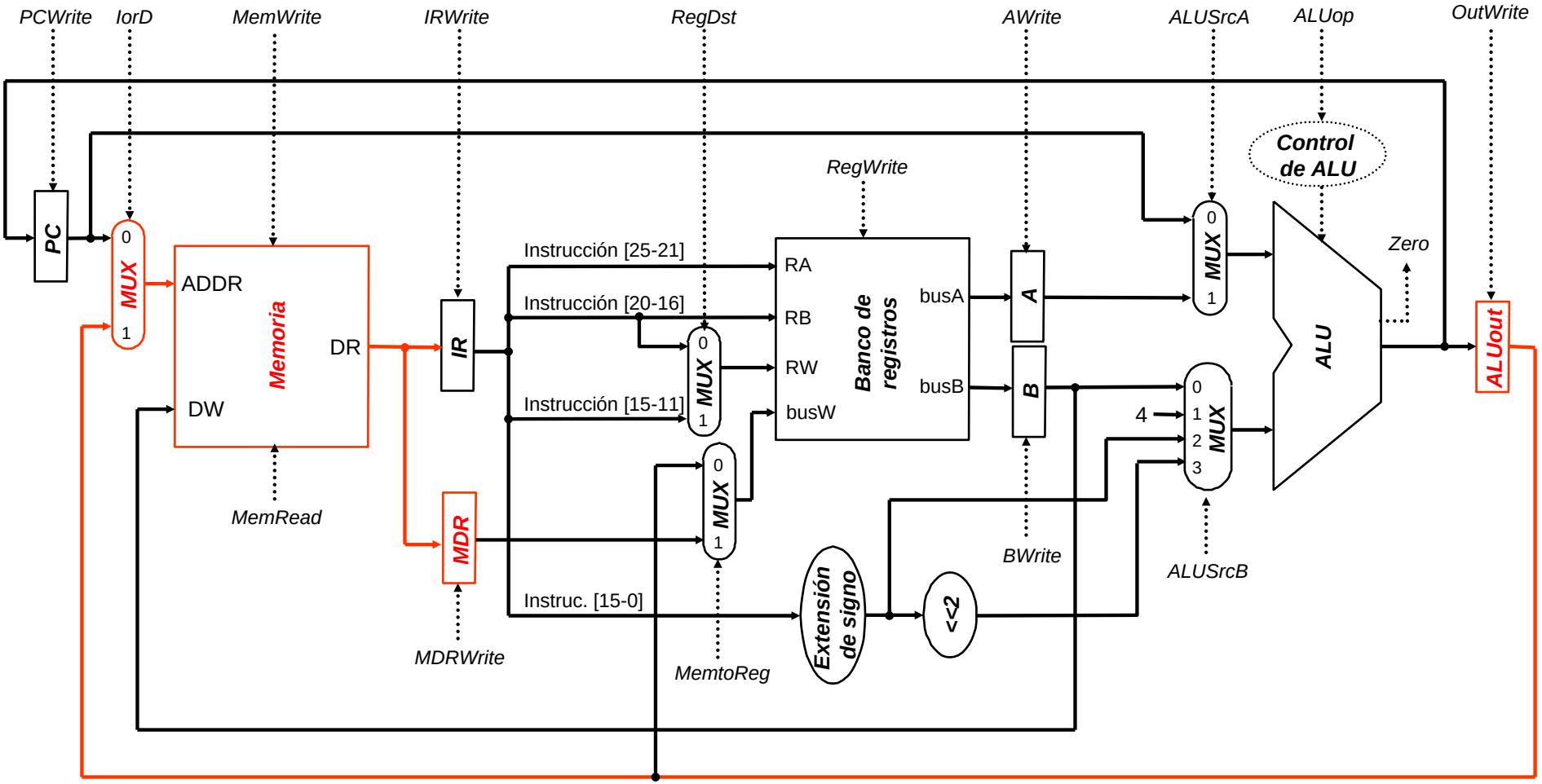
Ejecución del segundo ciclo de una instrucción LW

4. diseño de la ruta de datos (multiciclo)



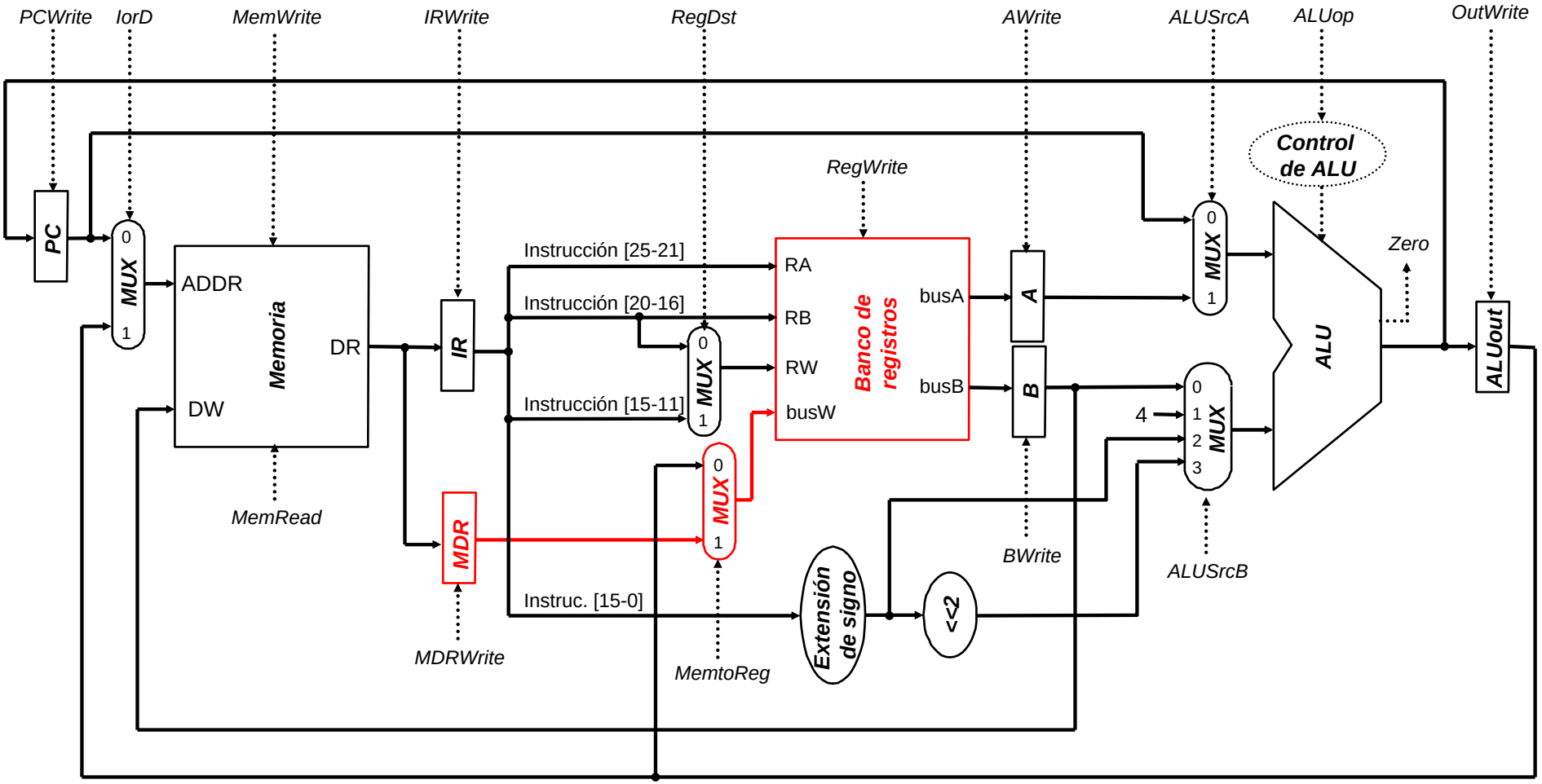
Ejecución del tercer ciclo de una instrucción LW

4. diseño de la ruta de datos (multiciclo)



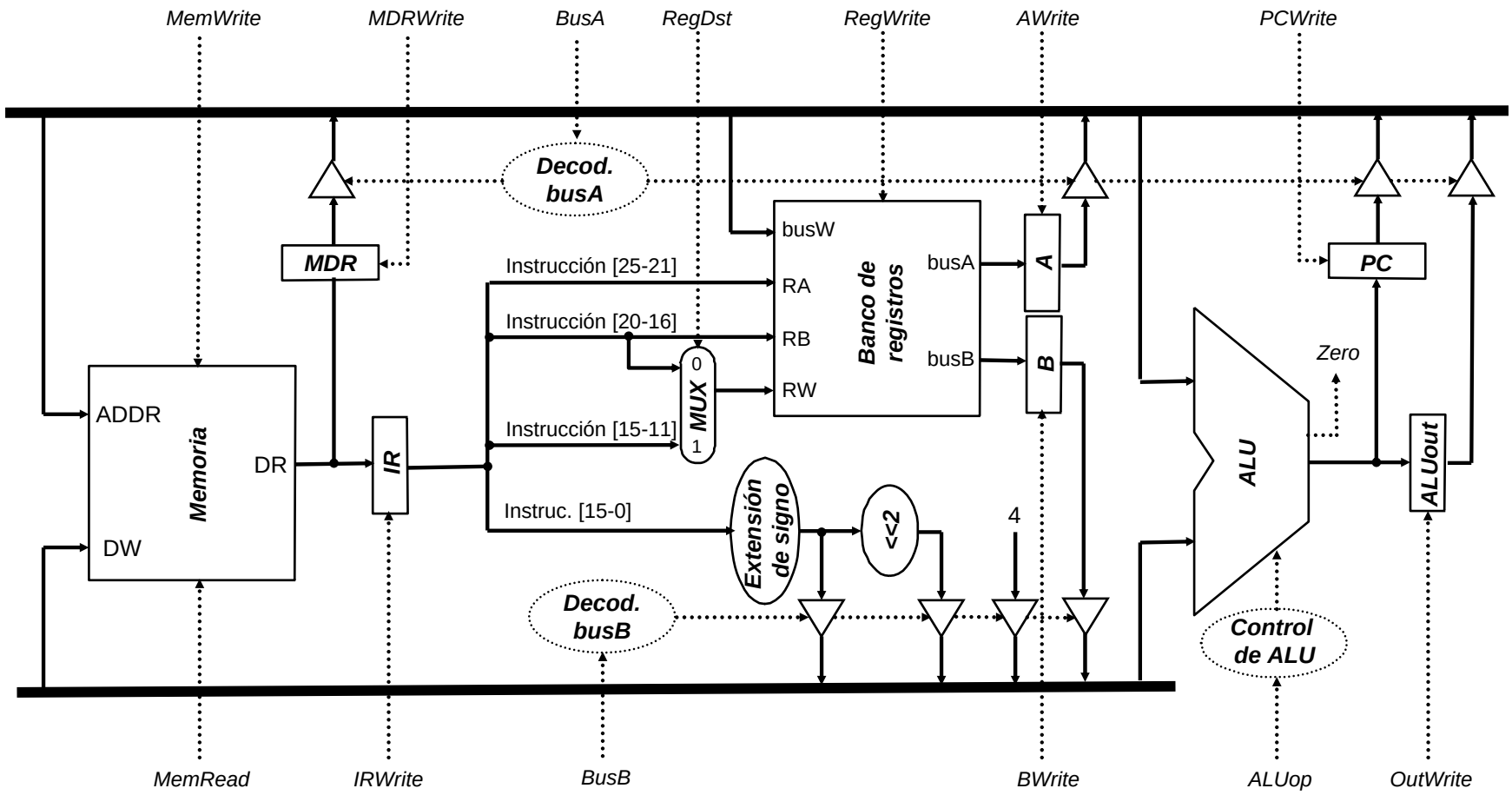
Ejecución del cuarto ciclo de una instrucción LW

4. diseño de la ruta de datos (multiciclo)



Ejecución del quinto ciclo de una instrucción LW

4. diseño de la ruta de datos (multiciclo)



Ruta de datos multiciclo con buses

Cada bus sólo puede ser usado para escribir desde una fuente (aunque puede ser leído desde varios destinos)

5. diseño del controlador (multiciclo)

Instrucción de carga (lw)

Transferencias entre registros “lógicas”

BR(rt) ← Memoria(BR(rs) + SignExt(inmed)),
PC ← PC + 4

Transferencias entre registros “físicas”

1. IR ← Memoria(PC), PC ← PC + 4
2. A ← BR(rs)
3. ALUout ← A + SignExt(inmed)
4. MDR ← Memoria(ALUout)
5. BR(rt) ← MDR

Instrucción de almacenaje (sw)

Transferencias entre registros “lógicas”

Memoria(BR(rs) + SignExt(inmed)) ← BR(rt),
PC ← PC + 4

Transferencias entre registros “físicas”

1. IR ← Memoria(PC), PC ← PC + 4
2. A ← BR(rs), B ← BR(rt)
3. ALUout ← A + SignExt(inmed)
4. Memoria(ALUout) ← B

Instrucción aritmética (tipo-R)

Transferencias entre registros “lógicas”

BR(rd) ← BR(rs) funct BR(rt), PC ← PC + 4

Transferencias entre registros “físicas”

1. IR ← Memoria(PC), PC ← PC + 4
2. A ← BR(rs), B ← BR(rt)
3. ALUout ← A funct B
4. BR(rd) ← ALUout

Instrucción de salto condicional (beq)

Transferencias entre registros “lógicas”

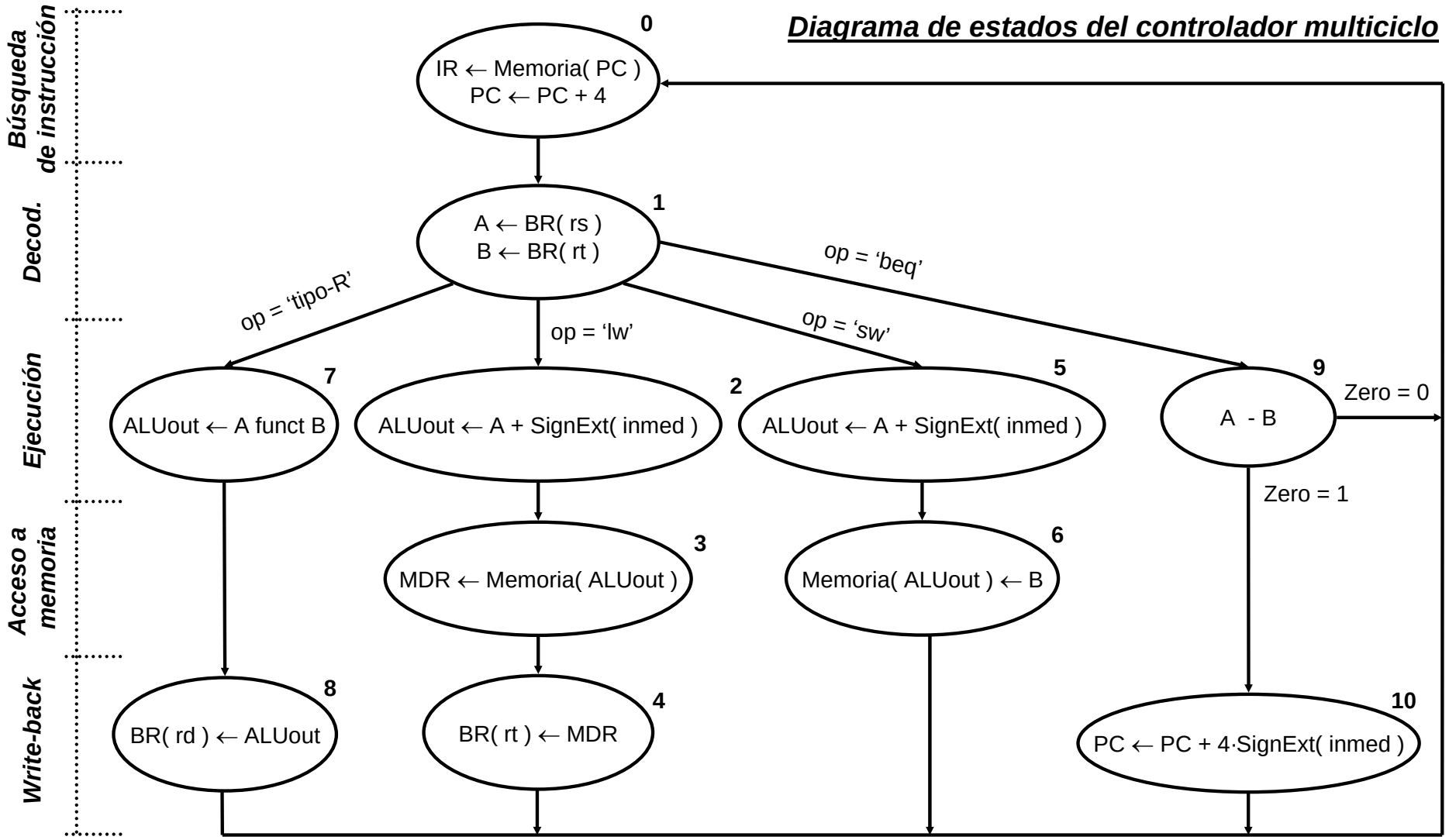
si (BR(rs) = BR(rt))
entonces PC ← PC + 4 + 4·SignExt(inmed)
sino PC ← PC + 4

Transferencias entre registros “físicas”

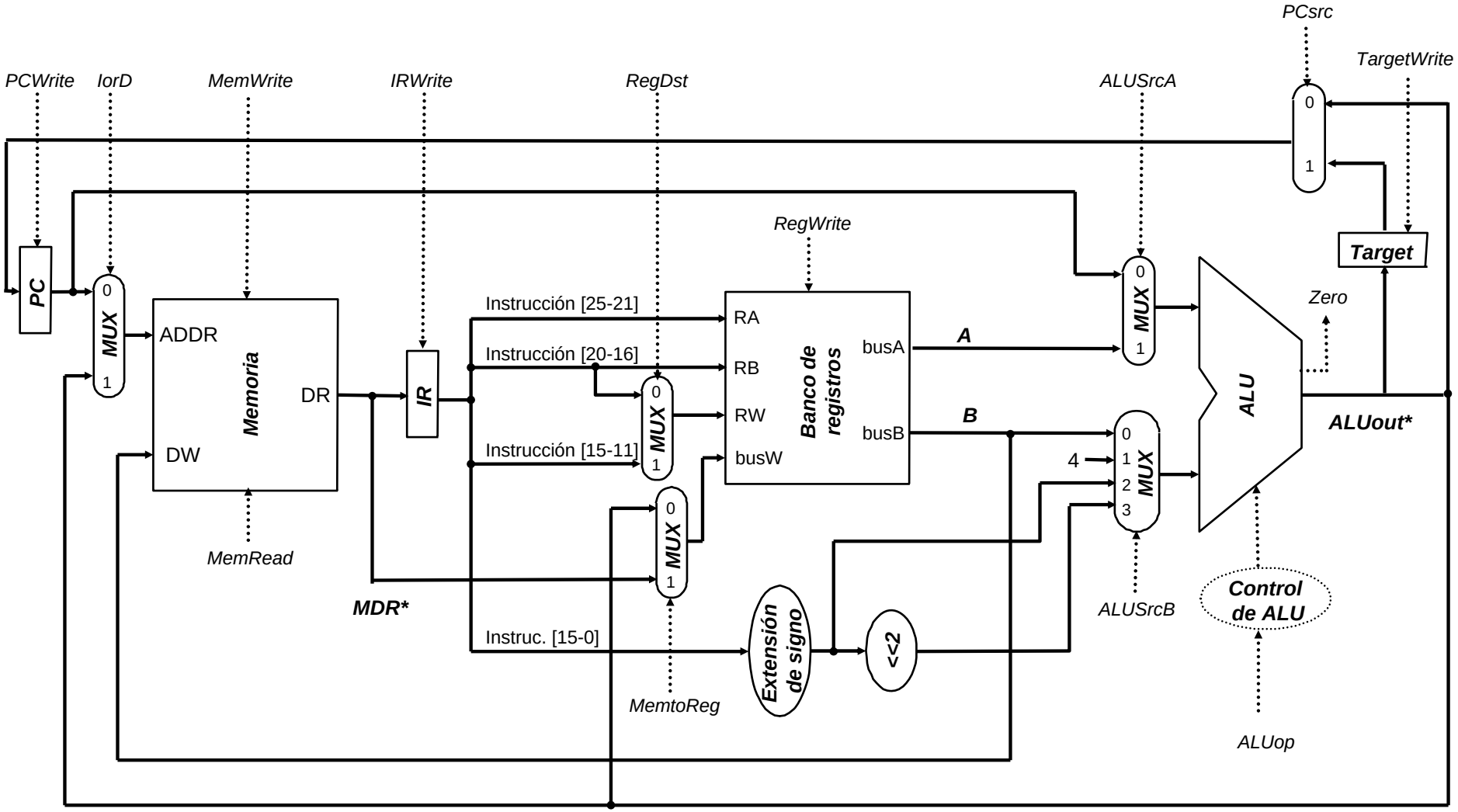
1. IR ← Memoria(PC), PC ← PC + 4
2. A ← BR(rs), B ← BR(rt),
3. A - B
4. si Zero entonces PC ← PC + 4·SignExt(inmed)

Observaciones: en todas las instrucciones las acciones 1. y 2. son iguales (excepto en lw, pero no habría problema en modificarla)

5. diseño del controlador (multiciclo)



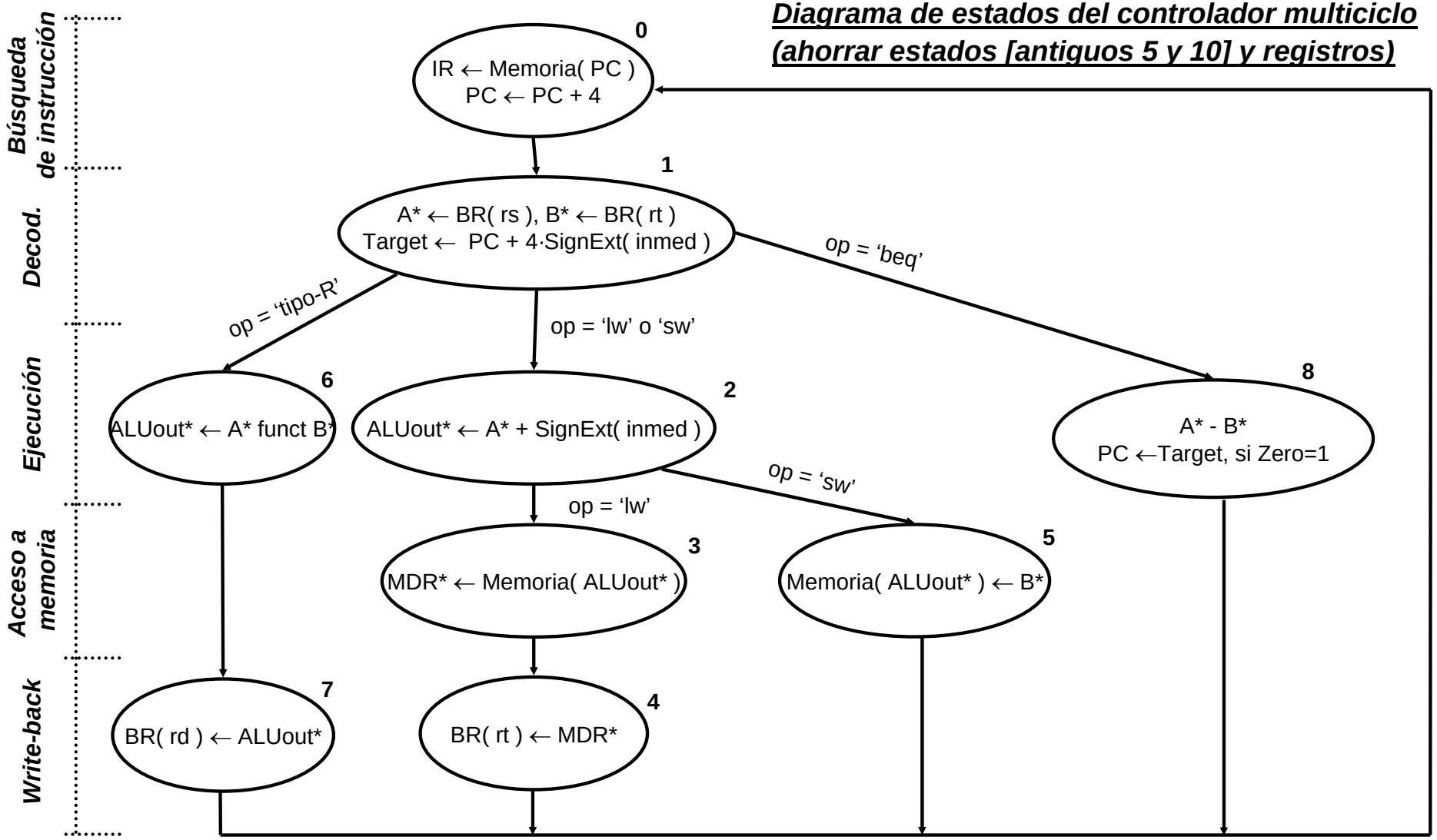
4’. diseño de la ruta de datos (multiciclo)



Ruta de datos multiciclo (ahorro de registros temporales y tiempo de ejecución)

5’. diseño del controlador (multiciclo)

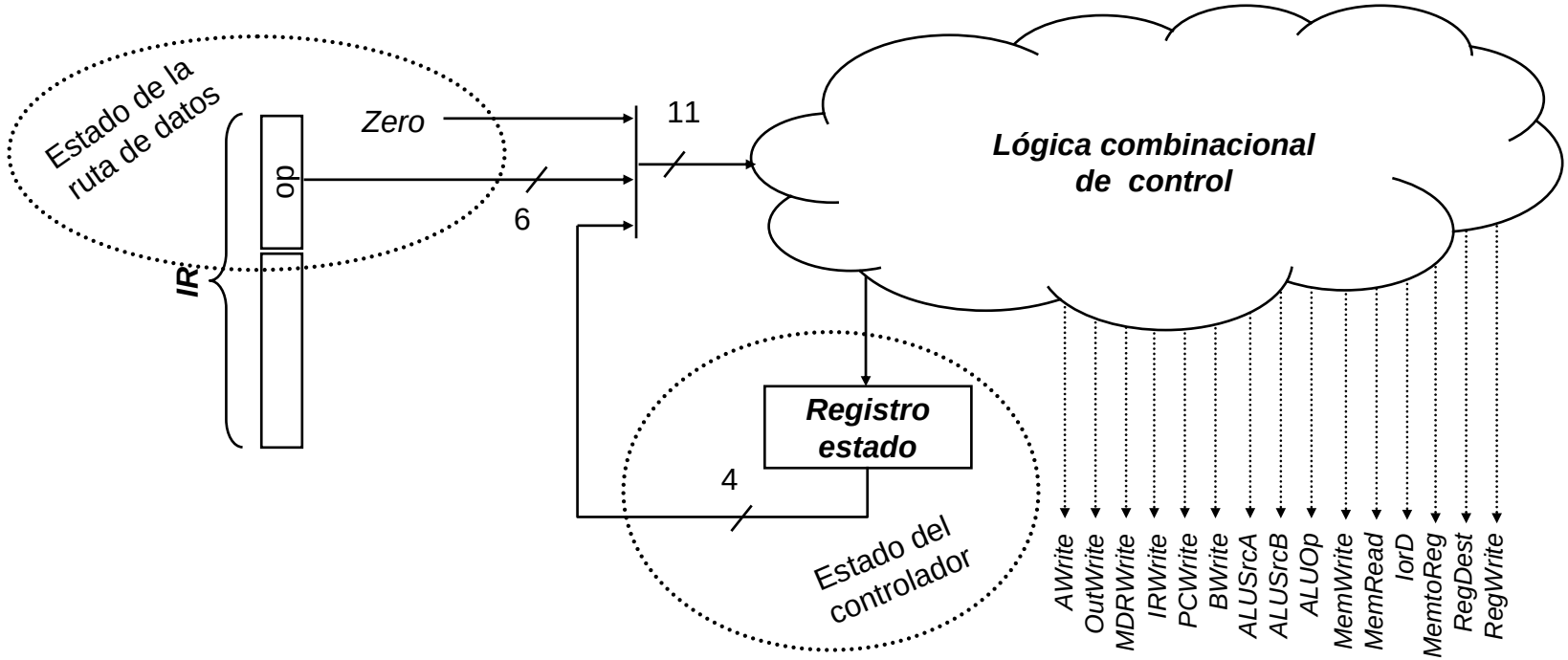
Diagrama de estados del controlador multiciclo (ahorrar estados [antiguos 5 y 10] y registros)



5. diseño del controlador (multiciclo)

El controlador como FSM (finite state machine)

1. Se **traducen** las transferencias entre registros como **conjuntos de activaciones** de los puntos de control de la ruta de datos
2. Se **codifican** los estados
3. Mediante **tablas de verdad** se describen:
   - ✓ las **transiciones de estado** en función del código de operación y del estado de la ruta de datos
   - ✓ el **valor de las señales** de control en función del estado (controlador tipo Moore) y adicionalmente en función del estado de la ruta de datos (controlador tipo Mealy).
4. La **estructura del controlador** estará formada por:
   - ✓ **Registro de estado**
   - ✓ Conjunto de **lógica combinacional de control** que implementa las anteriores tablas



5. diseño del controlador (multiciclo)

Tabla de verdad del controlador

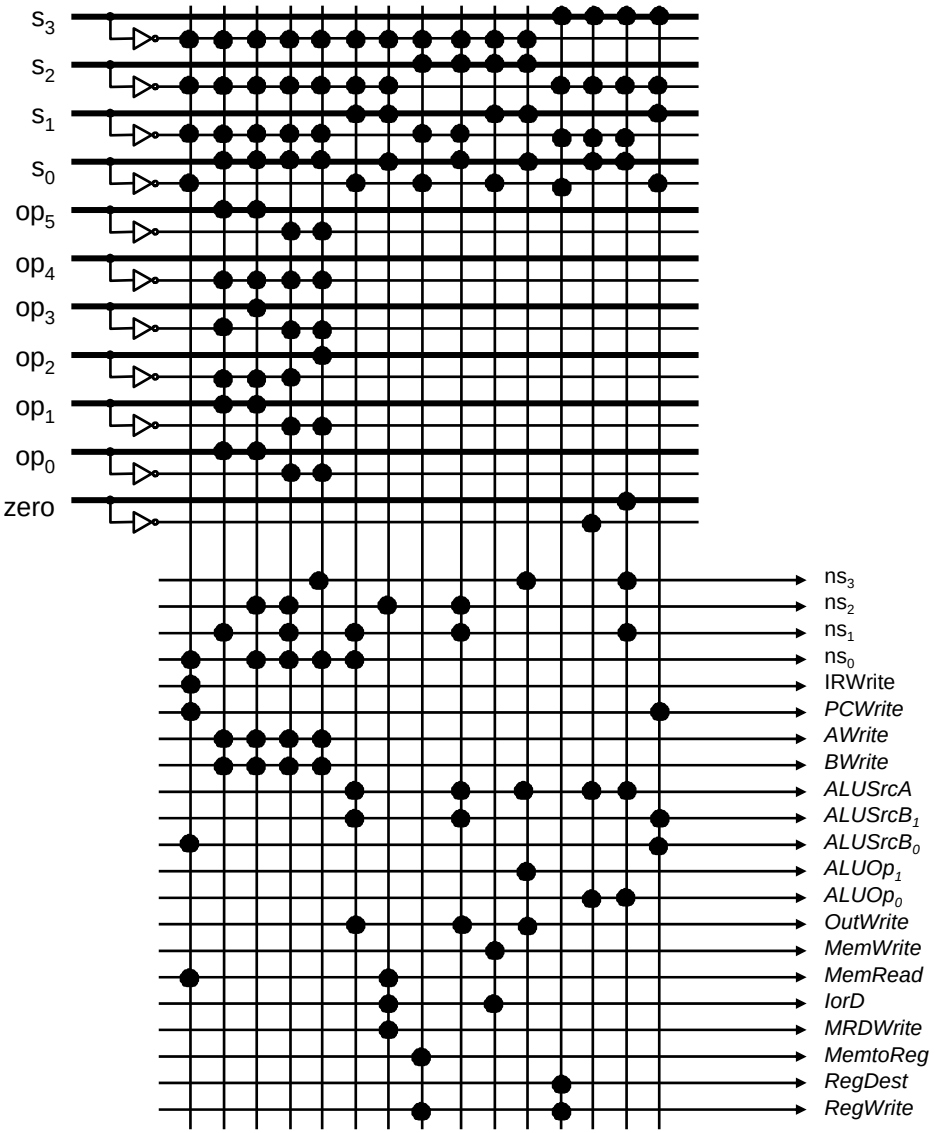
Estado actual	op	Zero	Estado siguiente	IRWrite	PCWrite	AWrite	BWrite	ALUSrcA	ALUScrB	ALUOp	OutWrite	MemWrite	MemRead	IorD	MDRWrite	MemtoReg	RegDest	RegWrite
0000	XXXXXX	X	0001	1	1			0	01	00 (add)		0	1	0				0
0001	100011 (lw)	X	0010	0	0	1	1					0	0					0
0001	101011 (sw)	X	0101															
0001	000000 (tipo-R)	X	0111															
0001	000100 (beq)	X	1001															
0010	XXXXXX	X	0011	0	0			1	10	00 (add)	1	0	0					0
0011	XXXXXX	X	0100	0	0							0	1	1	1			0
0100	XXXXXX	X	0000	0	0							0	0			1	0	1
0101	XXXXXX	X	0110	0	0		0	1	10	00 (add)	1	0	0					0
0110	XXXXXX	X	0000	0	0							1	0	1				0
0111	XXXXXX	X	1000	0	0			1	00	10 (funct)	1	0	0					0
1000	XXXXXX	X	0000	0	0							0	0			0	1	1
1001	XXXXXX	0	0000	0	0			1	00	01 (sub)		0	0					0
1001	XXXXXX	1	1010															
1010	XXXXXX	X	0000	0	1			0	11	00 (add)		0	0					0

5. diseño del controlador (multiciclo)

Alternativas de implementación de la lógica de control

- Lógica discreta:
  - 21 funciones de conmutación
  - 11 variables diferentes
- 1 PLA
  - 11 entradas
  - 21 salidas
  - 15 términos producto
- 1 ROM (~42 Kbits):
  - 11 bits de dirección (2^{11} palabras)
  - palabras de 21 bits
- 2 ROM (~10 Kbits)
  - ROM de control:
    - ⇨4 bits de dirección (2^4 palabras)
    - ⇨palabra de 17 bits
  - ROM de siguiente estado:
    - ⇨11 bits de dirección (2^{11} palabras)
    - ⇨palabras de 4 bits

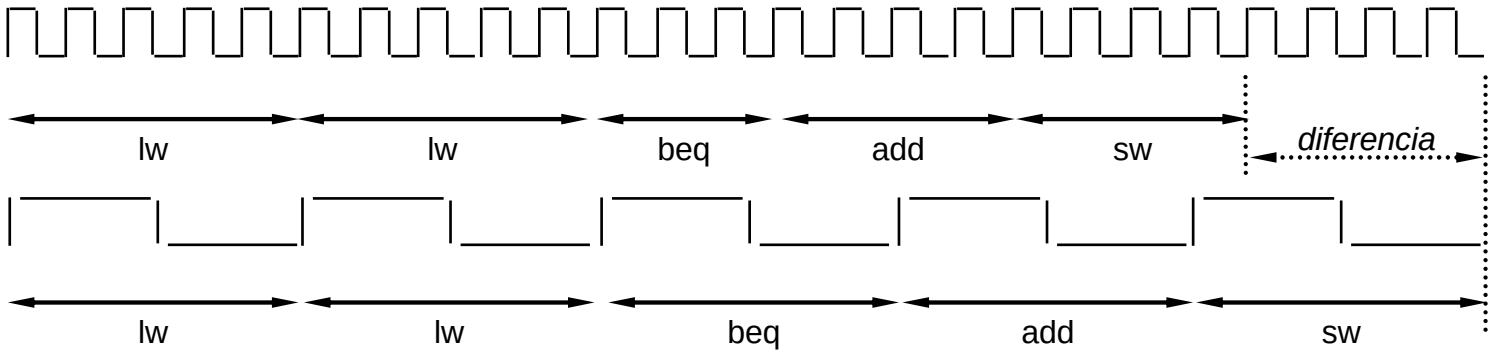
- Ventajas de la lógica discreta:
  - velocidad y coste
- Ventajas de la lógica almacenada:
  - facilidad de diseño
  - adaptabilidad



6. Comparación: monociclo vs. multiciclo

Asumir que el tiempo de ciclo en el procesador multiciclo es 5 veces menor que el tiempo de ciclo en el procesador monociclo

```
lw $t2, 0($t3)
lw $t3, 4($t3)
beq $t2, $t3, Label  #asumir que no se salta
add $t5, $t2, $t3
sw $t5, 8($t3)
```



10^6 instrucciones tardan en ejecutarse:

- ✓ $t_{multi} = 10^6 \cdot CPI_{multi} \cdot t_{multi} = 10^6 \cdot 4.03 \cdot t_{multi}$
- ✓ $t_{mono} = 10^6 \cdot CPI_{mono} \cdot t_{mono} = 10^6 \cdot 1 \cdot 5 \cdot t_{multi}$

$t_{multi} / t_{mono} = 4.03 / 5 = 0.8$
los programas tardan un **20% menos** en ejecutarse en el computador multiciclo

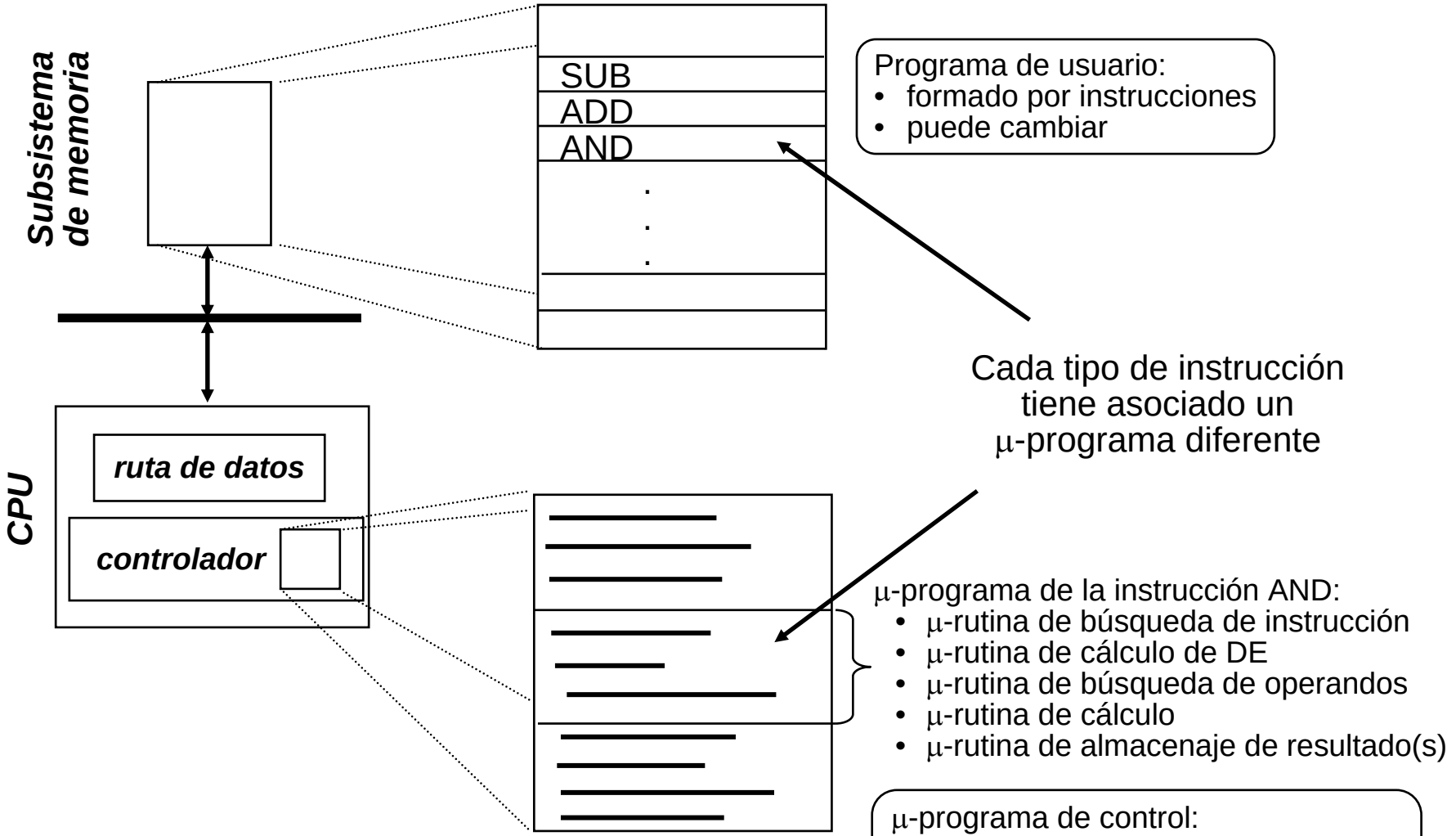
Operación	*Frecuencia*	*Ciclos*	*CPI*
tipo-R	50%	4	2.0
lw	20%	5	1.0
st	10%	4	.4
beq (salta)	2.5%	4	.1
beq (no salta)	17.5%	3	0.53
			4.03

7. control microprogramado

Comparación de las alternativas de implementación de la lógica de control

- El diseño con **lógica discreta** o PLA:
  - **No es sistemático**
  - Una vez diseñado **es inflexible**: un error o modificación requieren el rediseño completo de la lógica de control
  - Una mayor complejidad del repertorio se traduce en una mayor complejidad del diseño
- El diseño con **lógica almacenada**:
  - **Es sistemático**: el método de diseño es similar a los métodos de programación:
    - traducir cada instrucción en una secuencia de palabras de control que se almacenan en la memoria de control.
  - **Es flexible**: para modificar o corregir una acción de control basta con modificar el contenido de una palabra de la memoria (ROM o PROM) sin modificar la estructura del controlador
  - Mayor complejidad de las instrucciones sólo implica mayor tamaño de la memoria
- Además en cualquiera de ambas alternativas:
  - Mucha de la lógica de control está dedicada a especificar el estado siguiente, siendo en realidad gran parte de ese secuenciamiento consecutivo:
    - En sistemas reales el número de estados es grande
  - Muchas de las palabras de control están repetidas, luego existe lógica desperdiciada
    - Por ejemplo, cálculo de D.E., acceso a memorias, etc..

7. control microprogramado

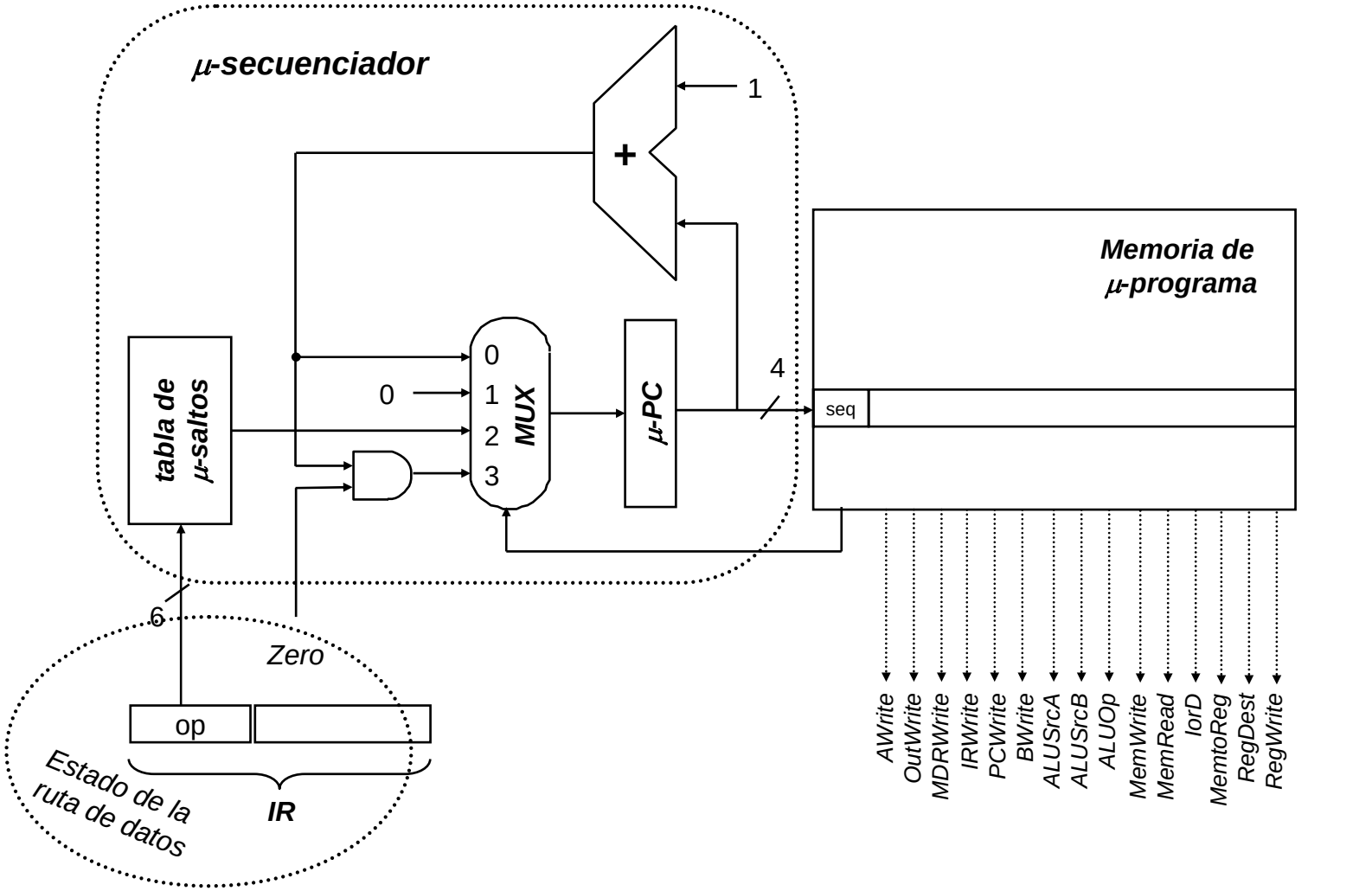


7. control microprogramado

Nomenclatura

- **μ-órdenes**: conjunto de señales de control que gobiernan las transferencias entre registros que realiza la ruta de datos.
- **μ-instrucción**: palabra de control almacenada. Incluye una colección de bits que indican las μ-órdenes que se realizan en un ciclo de reloj
- **formato de μ-instrucción**: distribución, codificación y tamaño de cada una de las μ-órdenes dentro de una μ-instrucción.
  - **Formato horizontal**: cada bit de la μ-instrucción está asociado a un punto de control de la ruta de datos
  - **Formato vertical**: definir todas las μ-instrucciones diferentes y codificarlas con el menor número de bits posibles. Requieren de un decodificador complejo que a veces se resuelve mediante nano-programación
  - **Formato vertical por campos**: compactar el formato de la microinstrucción para cada clase de μ-operación, y decodificarla localmente para generar los valores de las señales de control
- **μ-rutina**: secuencia de μ-instrucciones encargadas de implementar una instrucción máquina o una porción de ella (por ejemplo, cálculo de la DE de memoria)
- **memoria de μ-programa**: memoria ROM (o PROM) donde se almacenan cada una de las μ-rutinas que implementan el repertorio de instrucciones de un computador
- **μ-secuenciador**: módulo del controlador encargado de direccionar adecuadamente la memoria de μ-programa para ejecutar las μ-instrucciones que forman una instrucción máquina.

7. control microprogramado



Controlador microprogramado
con formato de μ-instrucción horizontal

7. control microprogramado

Contenido de la memoria de μ-programa

	μ-dirección (estado actual)	Seq	IRWrite	PCWrite	AWrite	BWrite	ALUSrcA	ALUScrB	ALUOp	OutWrite	MemWrite	MemRead	IorD	MDRWrite	MemtoReg	RegDest	RegWrite
fetch y decod.	0000	00 (inc)	1	1			0	01	00 (add)		0	1	0				0
	0001	10 (saltar)	0	0	1	1					0	0					0
lw	0010	00 (inc)	0	0			1	10	00 (add)	1	0	0					0
	0011	00 (inc)	0	0							0	1	1	1			0
	0100	01 (cero)	0	0							0	0			1	0	1
sw	0101	00 (inc)	0	0		0	1	10	00 (add)	1	0	0					0
	0110	01 (cero)	0	0							1	0	1				0
tipo-R	0111	00 (inc)	0	0			1	00	10 (funct)	1	0	0					0
	1000	01 (cero)	0	0							0	0			0	1	1
beq	1001	11 (cond)	0	0			1	00	01 (sub)		0	0					0
	1010	01 (cero)	0	1			0	11	00 (add)		0	0					0

Tabla de μ-saltos

op	μ-dirección de salto
100011 (lw)	0010
101011 (sw)	0101
000000 (tipo-R)	0111
000100 (beq)	1001

Problemas del formato horizontal:

- μ-instrucciones muy largas: grandes memorias de μ-programa
- la mayor parte de los bits de la μ-instrucción están a 0, o no son relevantes

7. control microprogramado

Solución:

- Detectar señales constantes: ***Awrite***, ***OutWrite*** y ***MDRWrite*** pueden ser igual a 1 en todos los estados. Con eso reducimos en 3 las líneas de control
- Agrupar aquellas señales que no se activan simultáneamente y codificarlas
- Sólo una μ-operación por grupo se especifica en cada μ-instrucción
- Se requieren decodificadores locales para generar las señales de control

Método de agrupamiento: Se suelen agrupar aquellas señales que tienen fines similares:

- ✓ Control de la ALU
- ✓ Control de la Memoria
- ✓ Control del banco de registros

Formato de μ-instrucción

seq	IR	PC	B	ALU	MEM	WB

					ALU			MEM			WB		
μ-dirección (estado actual)	seq	IRWrite	PCWrite	BWrite	ALUSrcA	ALUScrB	ALUOp	MemWrite	MemRead	IorD	MemtoReg	RegDest	RegWrite
0000	00	1	1		0	01	00	0	1	0			0
0001	10	0	0	1				0	0				0
0010	00	0	0		1	10	00	0	0				0
0011	00	0	0					0	1	1			0
0100	01	0	0					0	0		1	0	1
0101	00	0	0	0	1	10	00	0	0				0
0110	01	0	0					1	0	1			0
0111	00	0	0		1	00	10	0	0				0
1000	01	0	0					0	0		0	1	1
1001	11	0	0		1	00	01	0	0				0
1010	01	0	1		0	11	00	0	0				0
					5 μ-op			4 μ-op			3 μ-op		

7. control microprogramado

Codificación de los campos de μ-instrucción

ALU	ALUSrcA	ALUSrcB	ALUop	μ-operación
000	1	00	10	A funct B → ???
001	0	01	00	PC + 4 → ???
010	1	10	00	A + SignExt(IR) → ???
011	0	11	00	PC + 4·SignExt(IR) → ???
100	1	00	01	A – B → ???
resto	-	--	--	nop

WB	MemtoReg	RegDest	RegWrite	μ-operación
00	-	-	0	nop
01	0	1	1	ALUout → BR
10	1	0	1	MDR → BR
11	-	-	-	nop

MEM	MemWrite	MemRead	IorD	μ-operación
00	0	0	-	nop
01	1	0	1	B → Memoria(ALUout)
10	0	1	0	Memoria(PC) → ???
11	0	1	1	Memoria(ALUout) → ???

7. control microprogramado

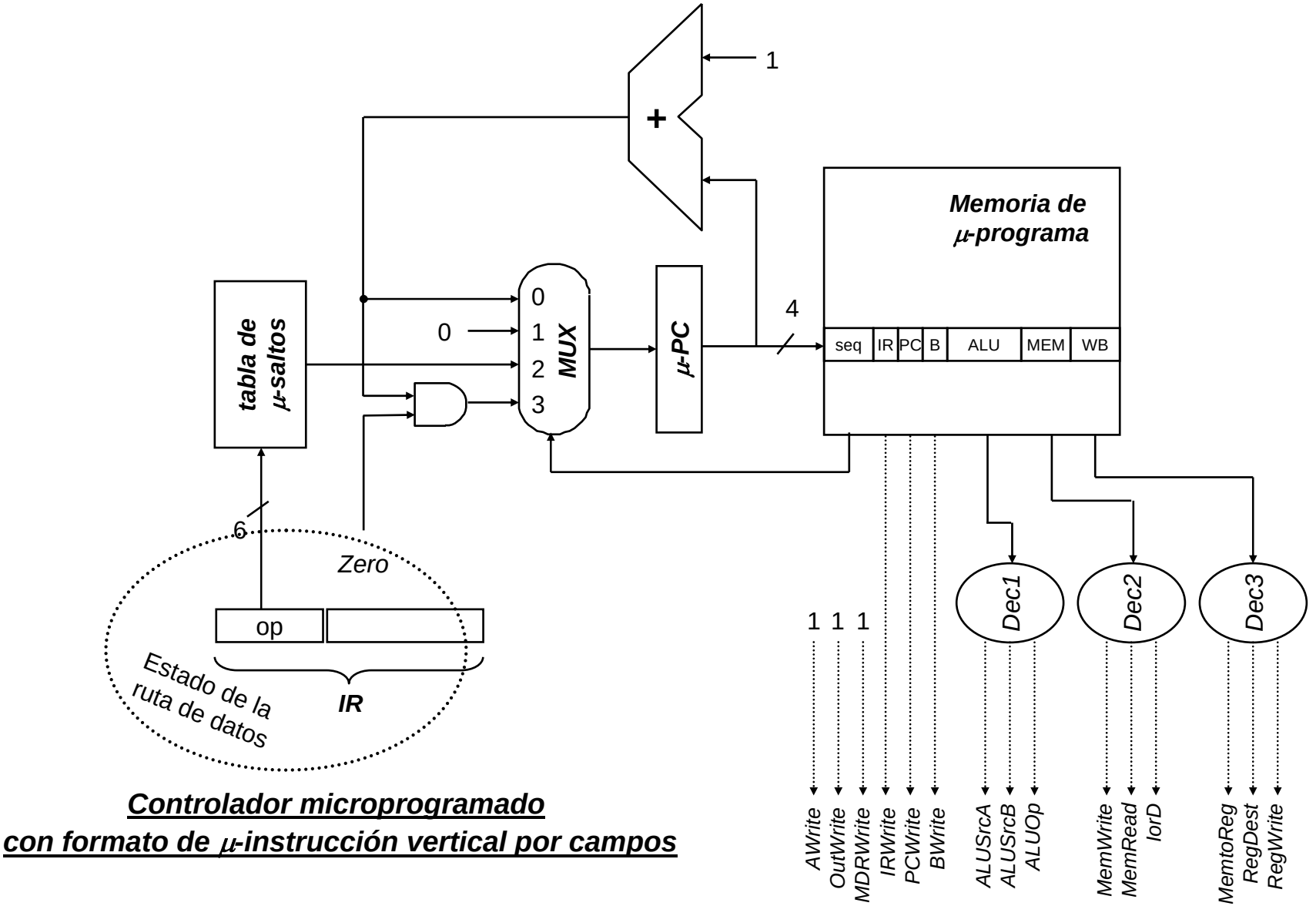
Contenido de la memoria de μ-programa

μ-dirección (estado actual)	seq	IRWrite	PCWrite	BWrite	ALU	MEM	WB
0000	00	1	1		001	10	00
0001	10	0	0	1		00	00
0010	00	0	0		010	00	00
0011	00	0	0			11	00
0100	01	0	0			00	10
0101	00	0	0	0	010	00	00
0110	01	0	0			01	00
0111	00	0	0		000	00	00
1000	01	0	0			00	01
1001	11	0	0		100	00	00
1010	01	0	1		011	00	00

Alternativas de implementación de la lógica de control

- **FSM con 1 ROM** (~42 Kbits):
  - 11 bits de dirección (2^{11} palabras)
  - palabras de 21 bits
- **μ-programado horizontal** (~0.3 Kbits)
  - 4 bits de dirección (2^4 palabras)
  - palabra de 19 (o 16) bits
- **μ-programado vertical por campos** (~0.2 Kbits)
  - 4 bits de dirección (2^4 palabras)
  - palabra de 12 bits

7. control microprogramado



Controlador microprogramado
con formato de μ-instrucción vertical por campos

8. Tratamiento de excepciones

Nomenclatura del MIPS

- **Excepción**. Cualquier cambio inesperado en el flujo de control.
- **Interrupción**. Cambio inesperado en el flujo de control debido a un evento externo.

Gestión de las excepciones: Instrucción indefinida y overflow aritmético

Acciones básicas:

- Salvar el contador de programa de la instrucción interrumpida en el *registro EPC* (exception program counter)
- Transferir el control al sistema operativo en alguna dirección especificada.
- El S.O. realizará la acción apropiada, como ejecutar alguna tarea asociada al overflow o detener la ejecución del programa.
- Volver a la ejecución normal del programa en la dirección guardada en EPC.

Hardware añadido:

Registro de estado: *Cause register* (32 bits) con un campo que indica la causa de la excepción:

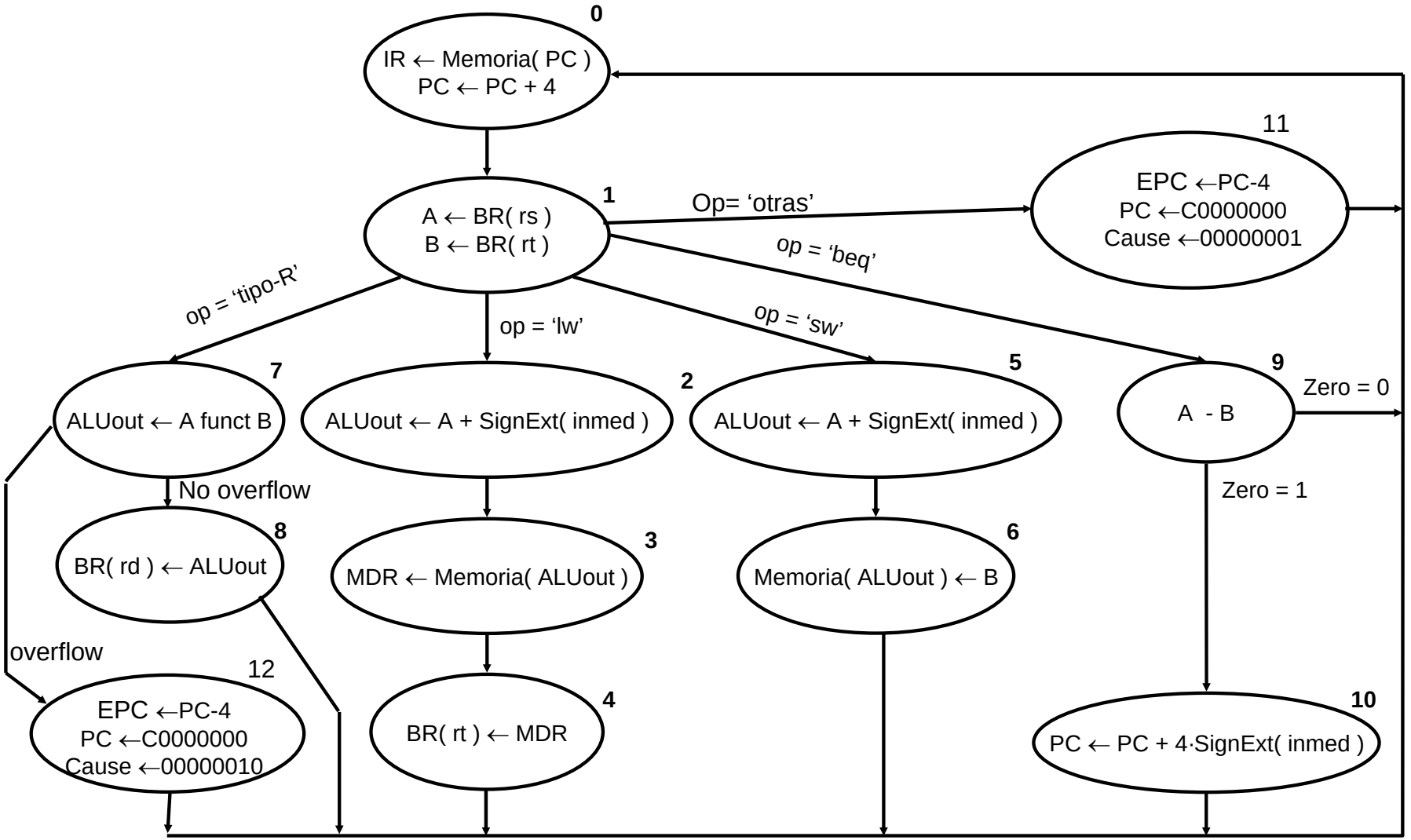
Bit 0: Instrucción indefinida.
Bit 1: Overflow aritmético.

Se añaden las señales de control:

EPCwrite. Escribe en EPC. (EPC <= PC – 4)
CauseWrite. Escribe en Cause
IntCause. Escribe un 1 sobre el bit apropiado de Cause.

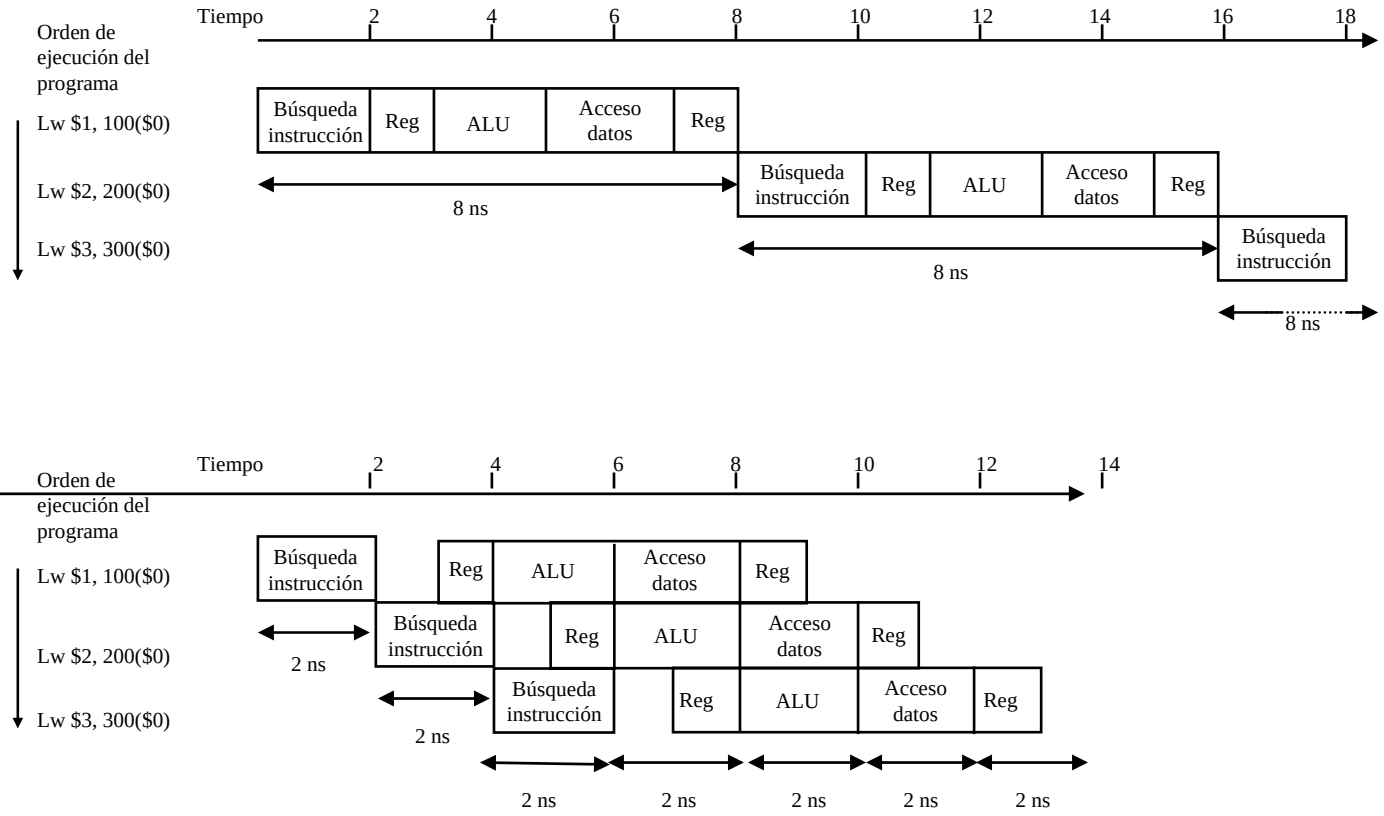
Para dar la dirección de la rutina de tratamiento de excepción, se añade una entrada al multiplexor que controla la carga del PC, con la dirección de esta rutina, por ej. $C0000000_{hex}$

8. Tratamiento de excepciones



9.- Introducción a la segmentación

Mejora el rendimiento incrementando el número de instrucciones que solapan su ejecución



El speed-up ideal es igual al número de etapas del pipeline. ¿Es posible?

9.- Introducción a la segmentación

- ¿Qué facilita la segmentación?

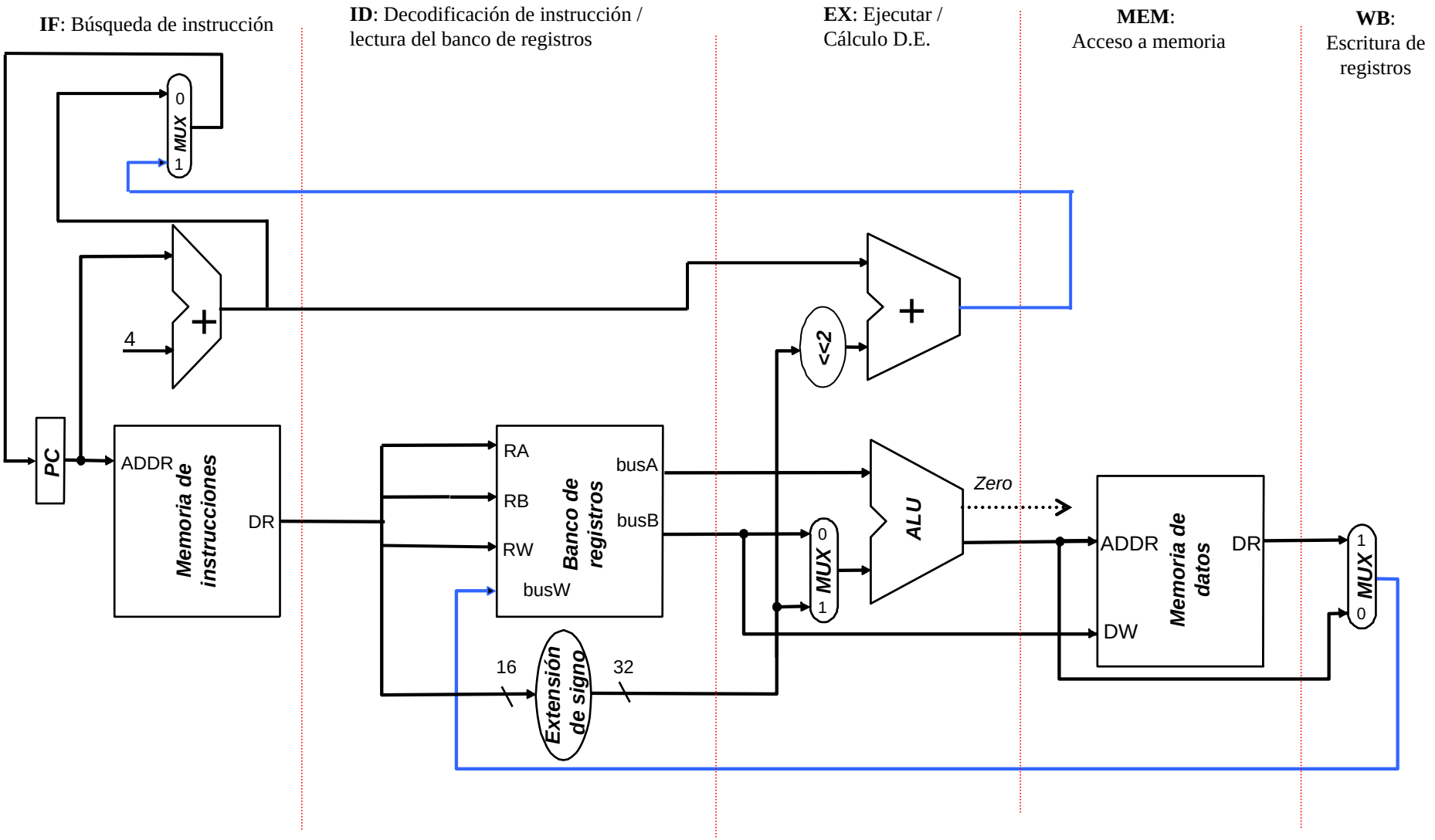
  - Todas las instrucciones de igual anchura
  - Pocos formatos de instrucción
  - Búsqueda de operandos en memoria sólo en operaciones de carga y almacenamiento

- ¿Qué dificulta la segmentación?

  - Conflictos estructurales
  - Conflictos de datos
  - Conflictos de control

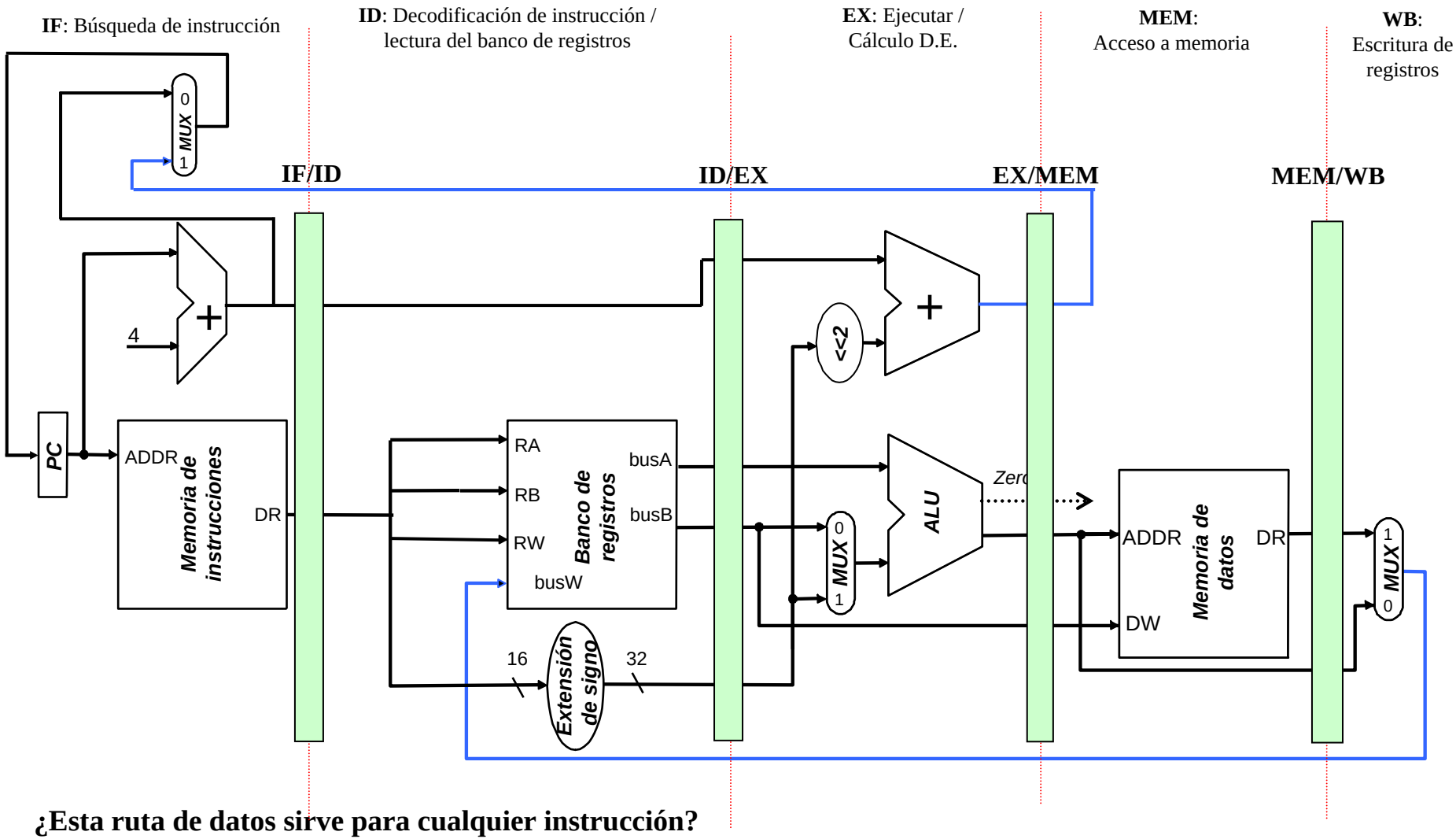
  - Gestión de interrupciones
  - Ejecución fuera de orden

9.- Introducción a la segmentación



10.- Diseño de la ruta de datos segmentada

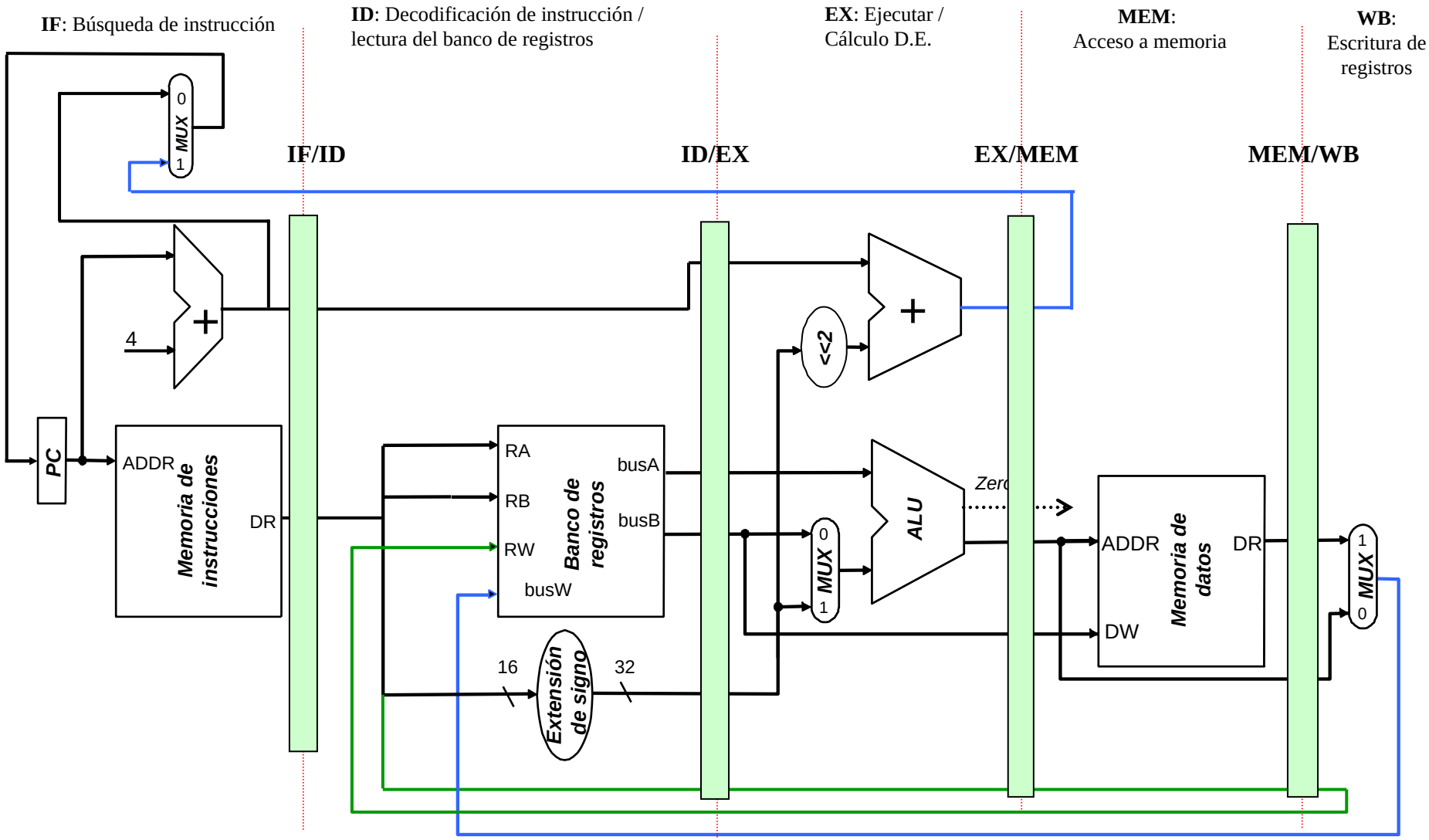
- **Necesidad de registros entre etapas**



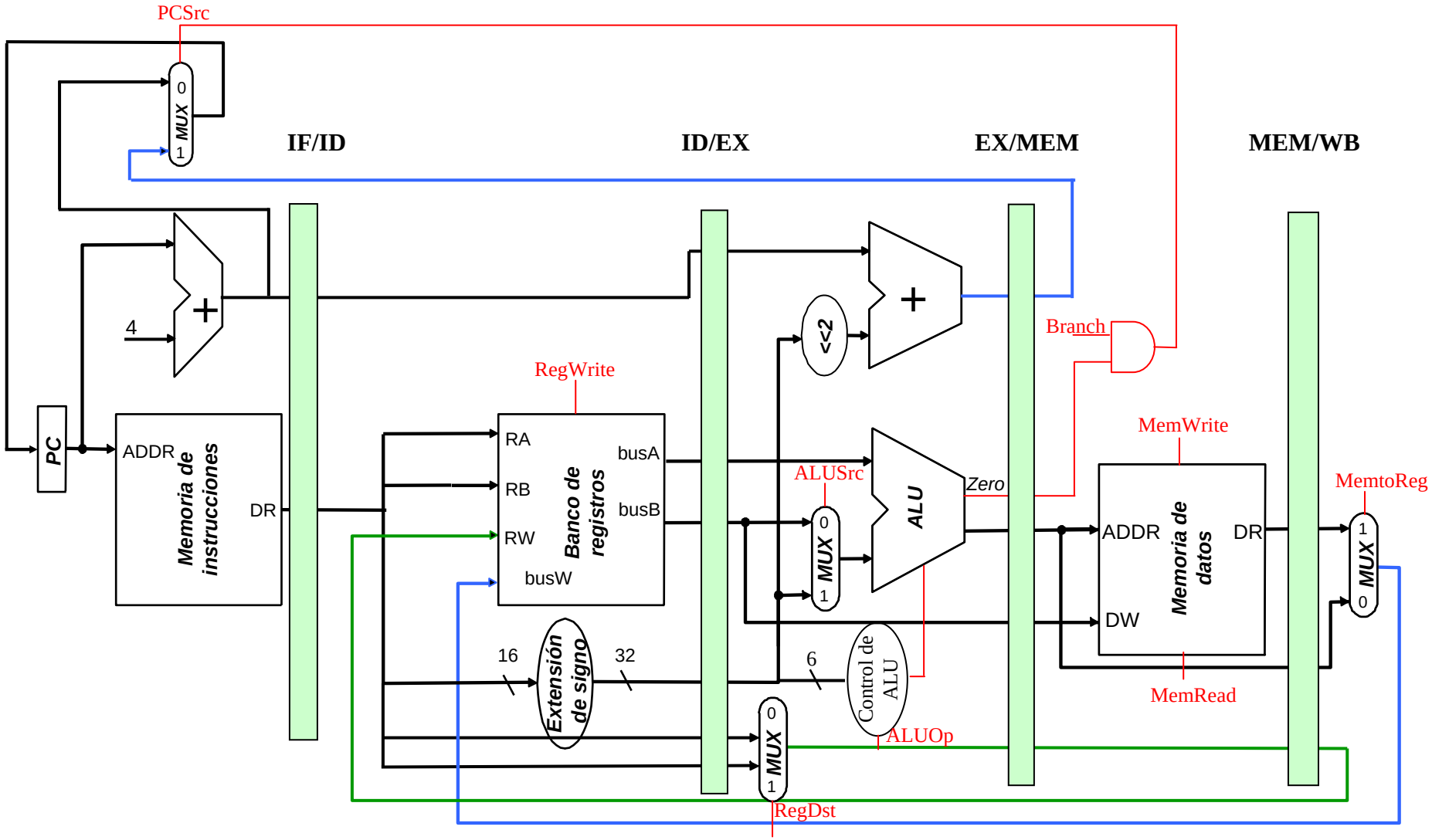
¿Esta ruta de datos sirve para cualquier instrucción?

10.- Diseño de la ruta de datos segmentada

•**Corrección:**



11.- Diseño del controlador segmentado



11.- Control de la ruta de datos segmentada

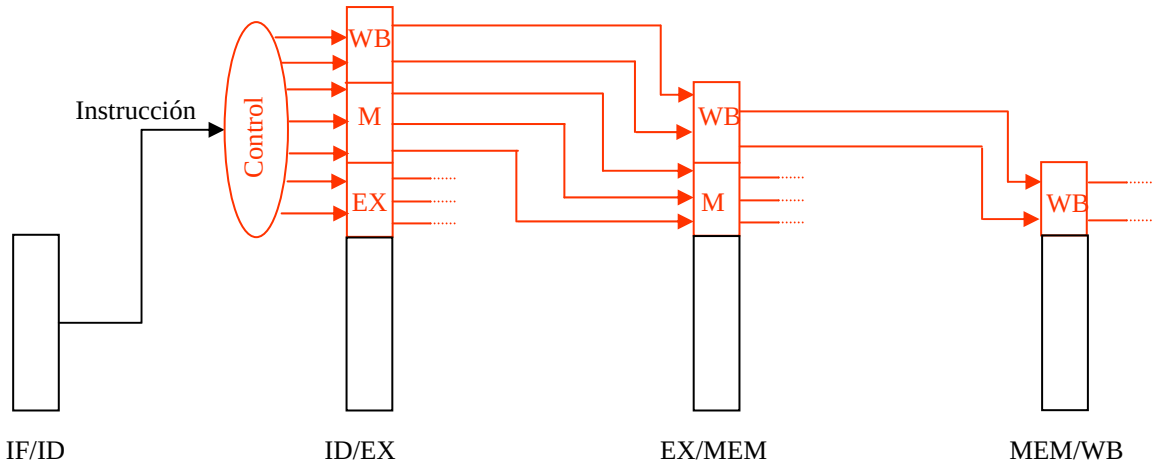
Acciones a realizar en cada una de las 5 etapas:

- Etapa IF: Búsqueda de la instrucción e incremento del PC
- Etapa ID: Decodificación de la instrucción y búsqueda de operandos en los registros
- Etapa EX: Ejecución
- Etapa Mem: Acceso a la memoria de datos
- Etapa WB: Escritura en el banco de registros.

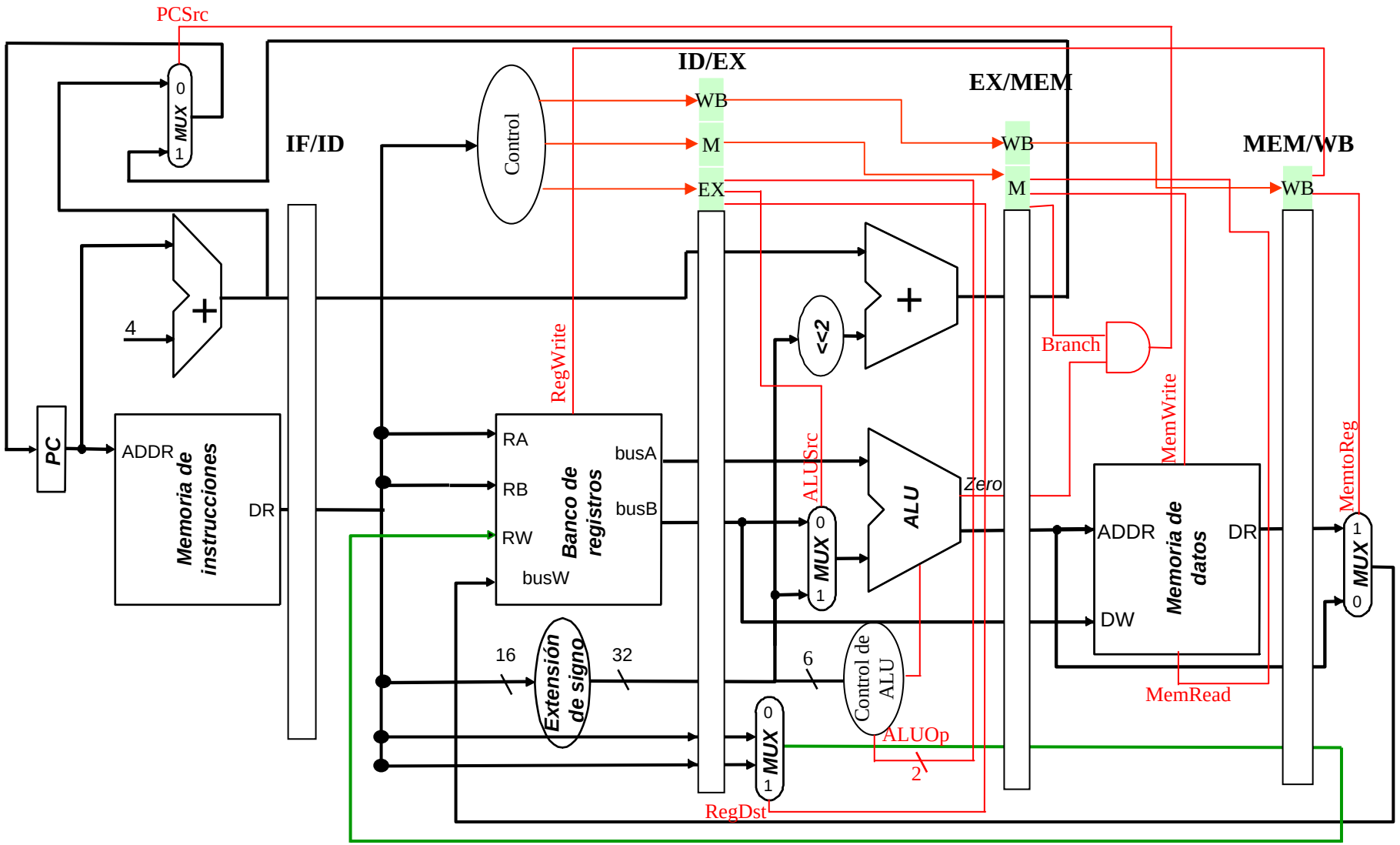
Instrucción	Líneas de control del estado de Ejecución/Cálculo de DE				Líneas de control del estado de acceso a memoria			Líneas de control del estado de WB	
	Reg Dst	**ALU Op1**	**ALU Op0**	**ALU Src**	**Branch**	**Mem Read**	**Mem Write**	**Reg Write**	**Mem to Reg**
Formato-R	1	1	0	0	0	0	0	1	0
Lw	0	0	0	1	0	1	0	1	1
Sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

11.- Control de la ruta de datos segmentada

•Las señales de control se generan en la U.C. y se van pasando de una etapa a otra como si fuesen datos.

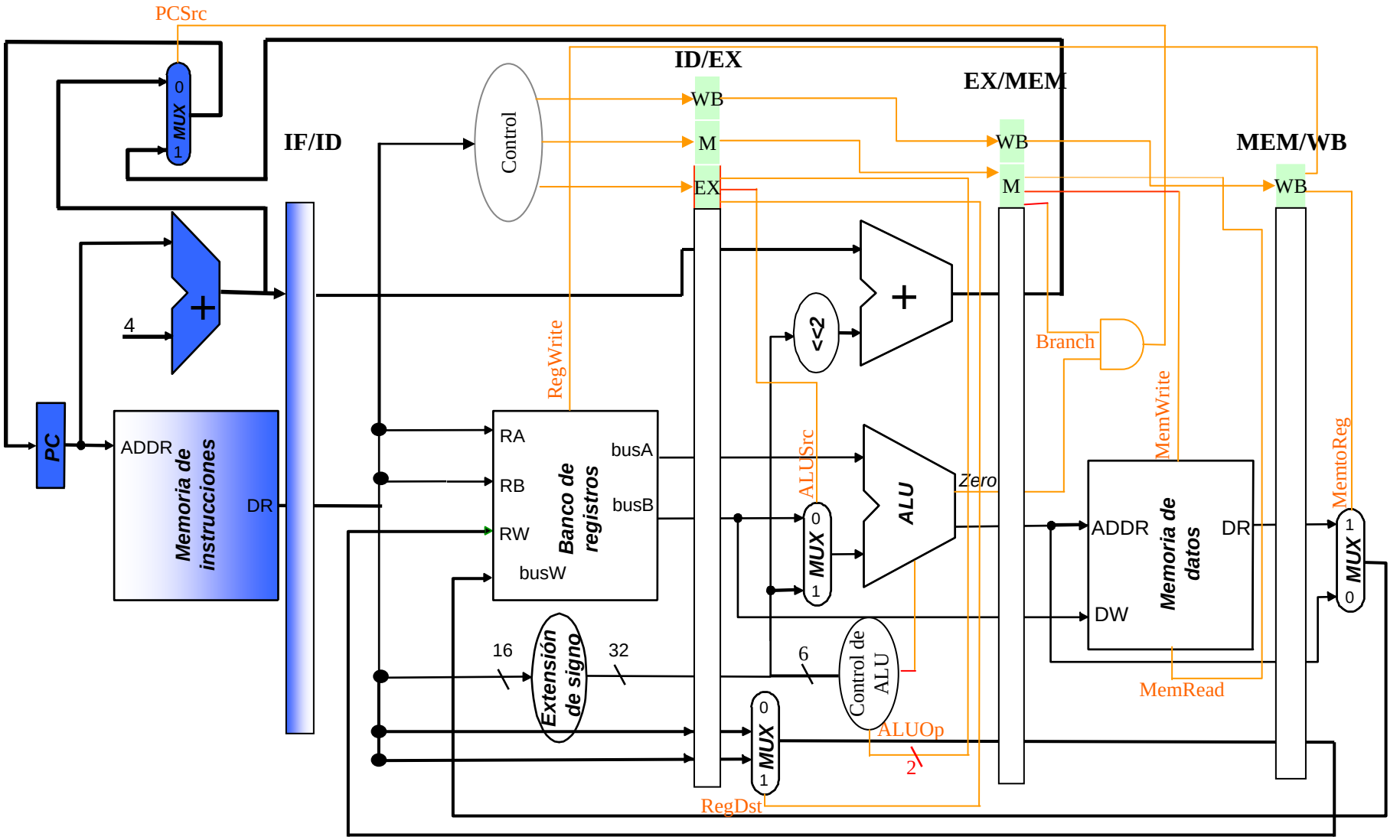


11.- Control de la ruta de datos segmentada



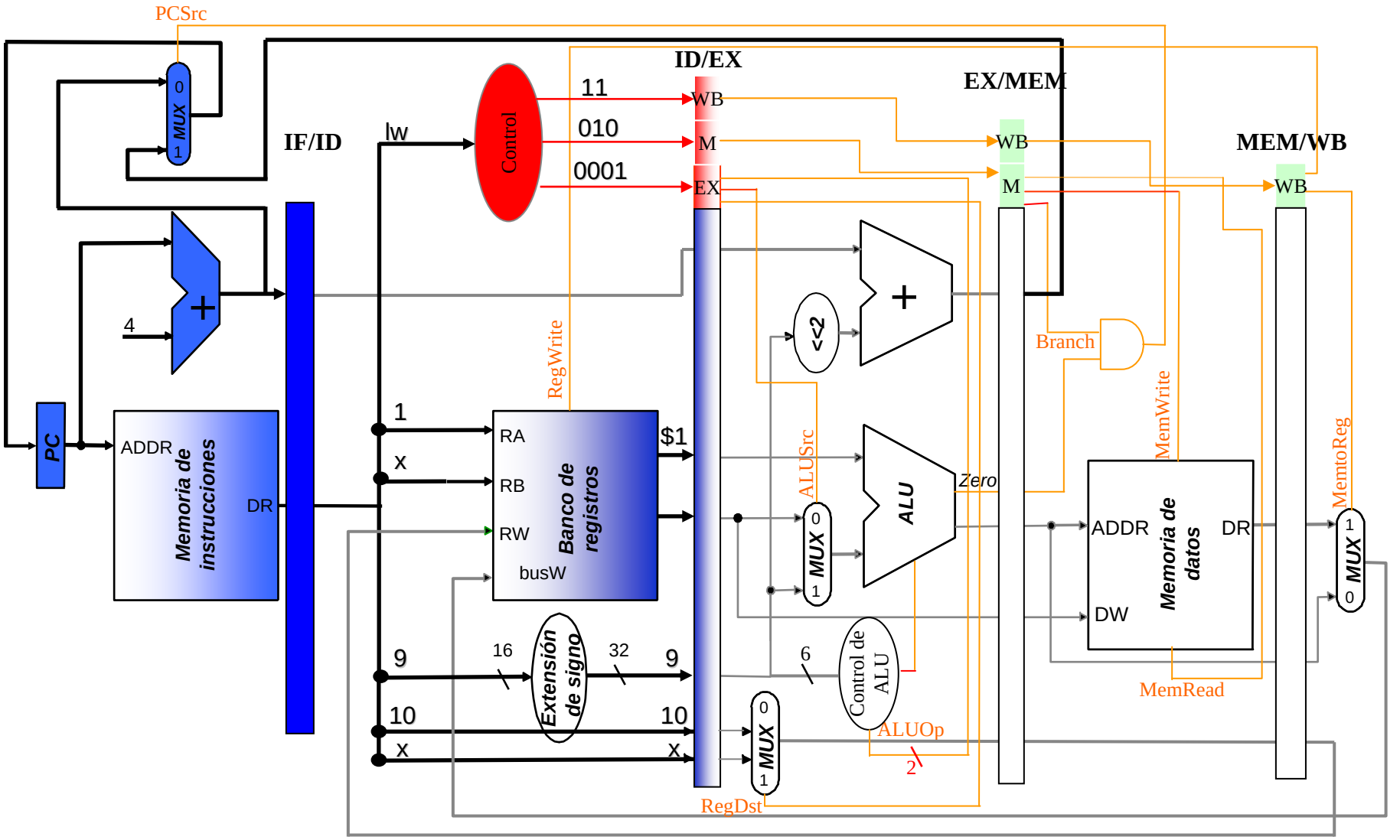
11.- Control de la ruta de datos segmentada

IF: lw $10,9($1) ID: antes<1> EX: antes<2> MEM: antes<3> WB: antes<4>



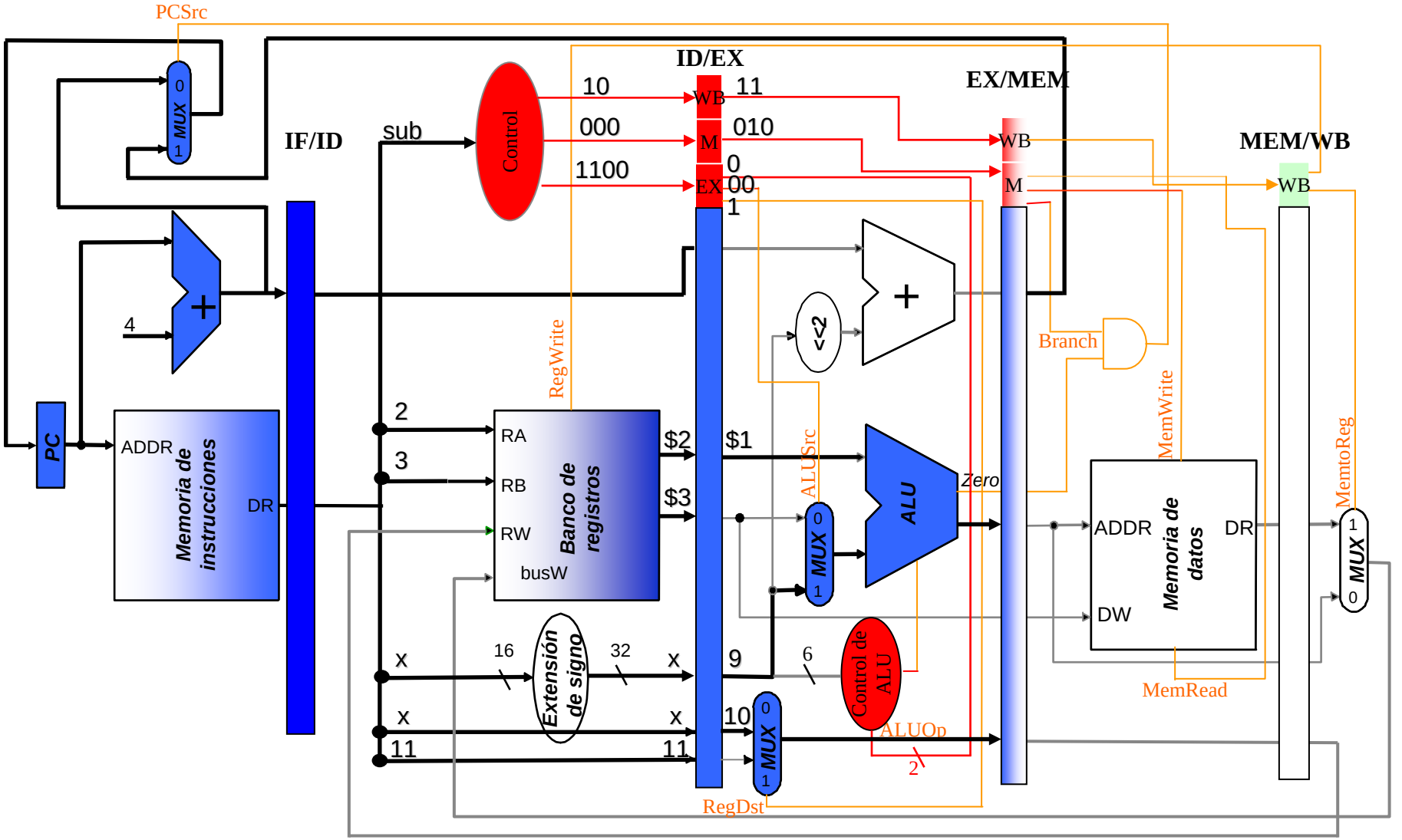
11.- Control de la ruta de datos segmentada

IF: sub $11,$2,$3 ID: lw $10,9($1) EX: antes<1> MEM: antes<2> WB: antes<3>



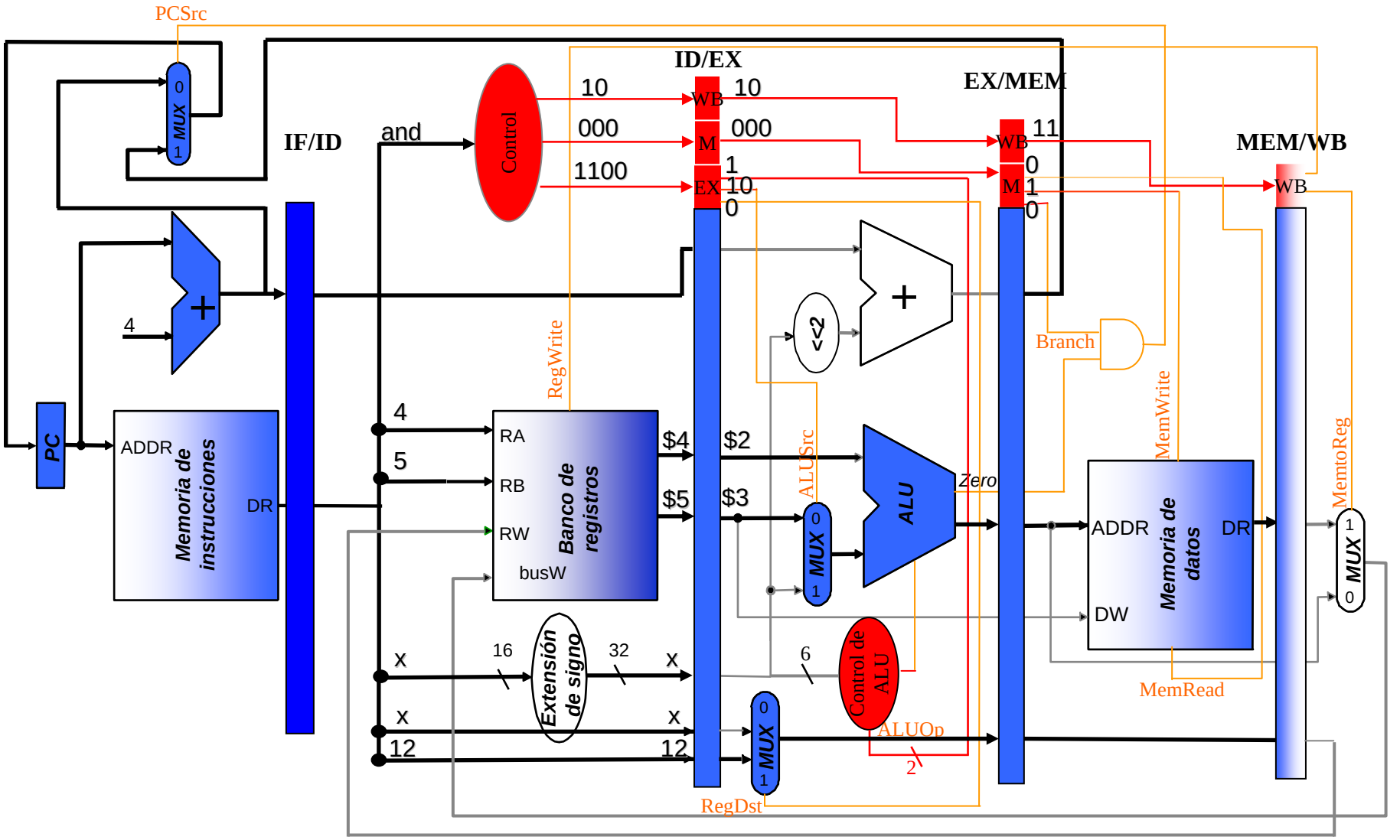
11.- Control de la ruta de datos segmentada

IF: and $12,$4,$5 ID: sub $11,$2,$3 EX: lw $10,… MEM: antes<1> WB: antes<2>



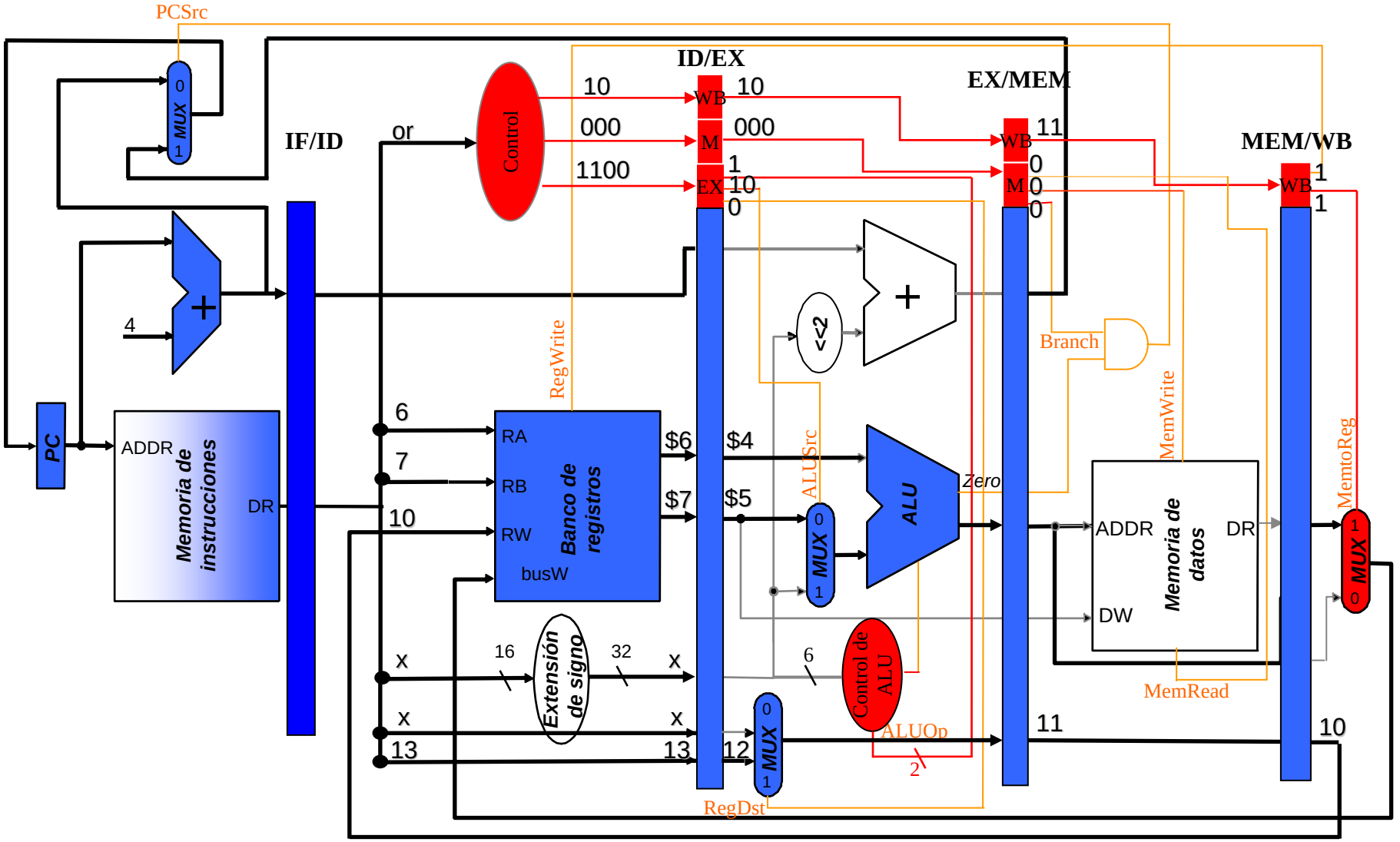
11.- Control de la ruta de datos segmentada

IF: or $13,$6,$7 ID: and $12,$4,$5 EX: sub $11,… MEM: lw $10, … WB: antes<1>



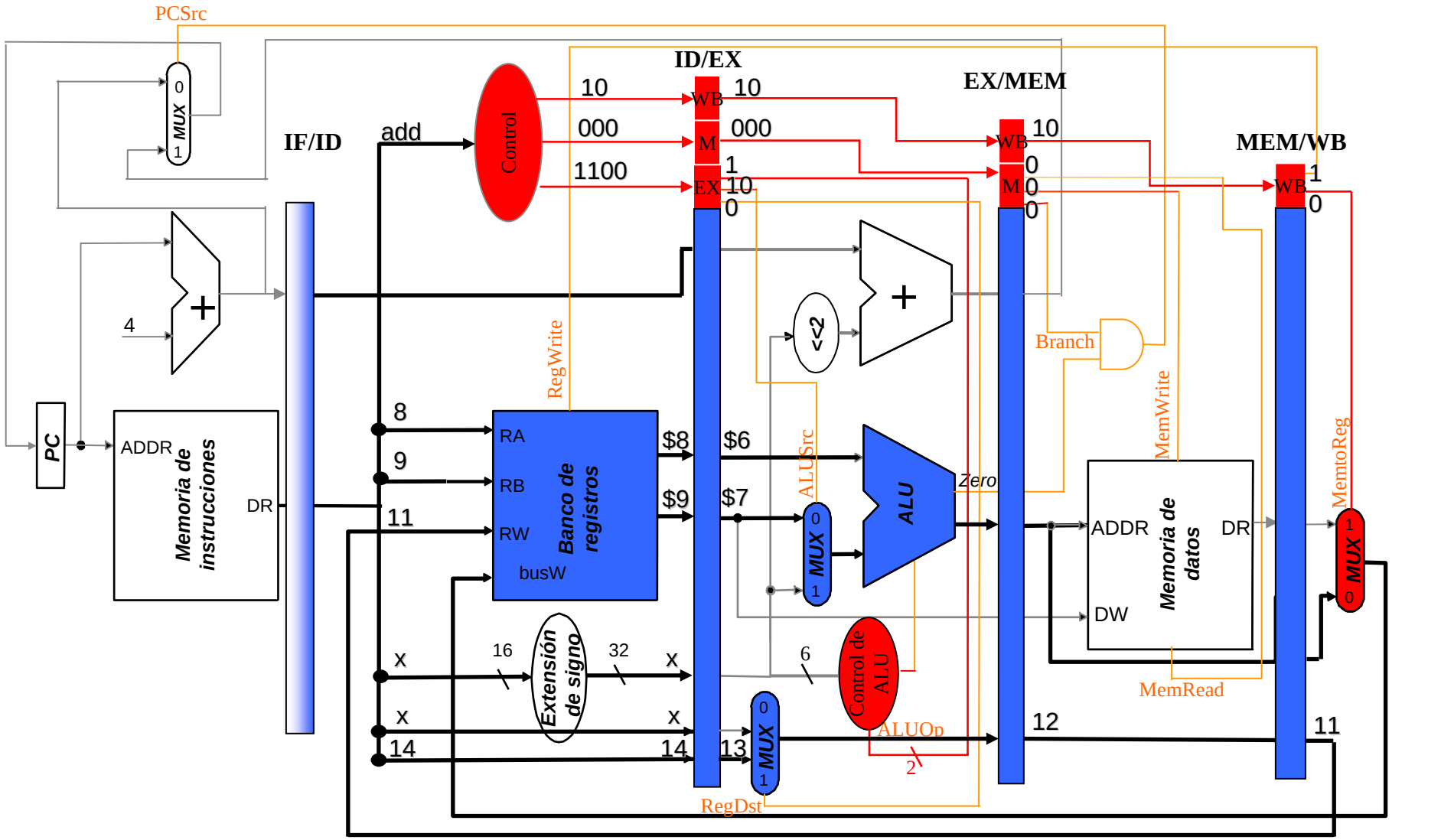
11.- Control de la ruta de datos segmentada

IF: add $14,$8,$9 ID: or $13,$6,$7 EX: and $12,… MEM: sub $11, … WB: lw $10,9($1)



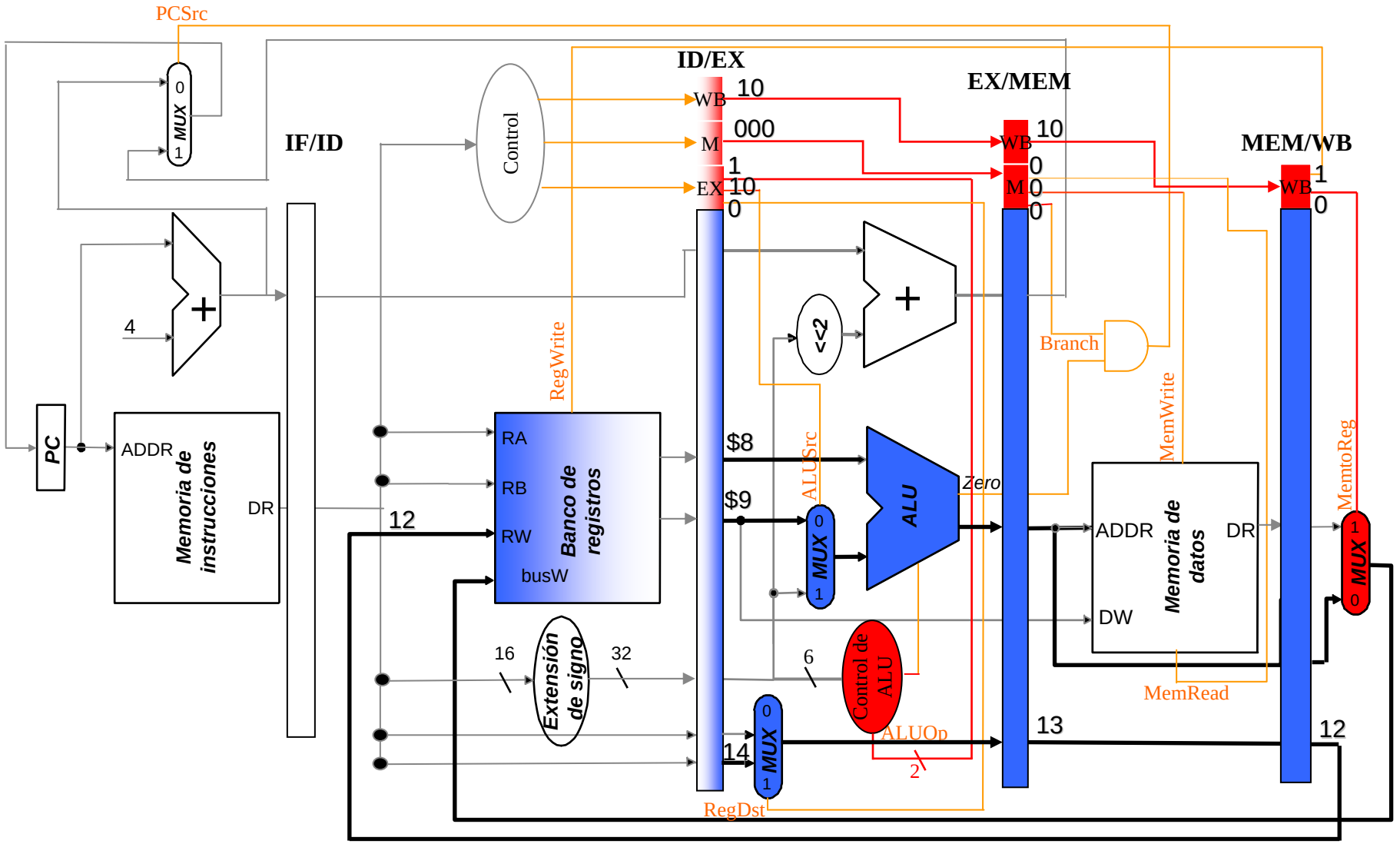
11.- Control de la ruta de datos segmentada

IF: <1> ID: add $14,$8,$9 EX: or $13,… MEM: and $12, … WB: sub $11,…



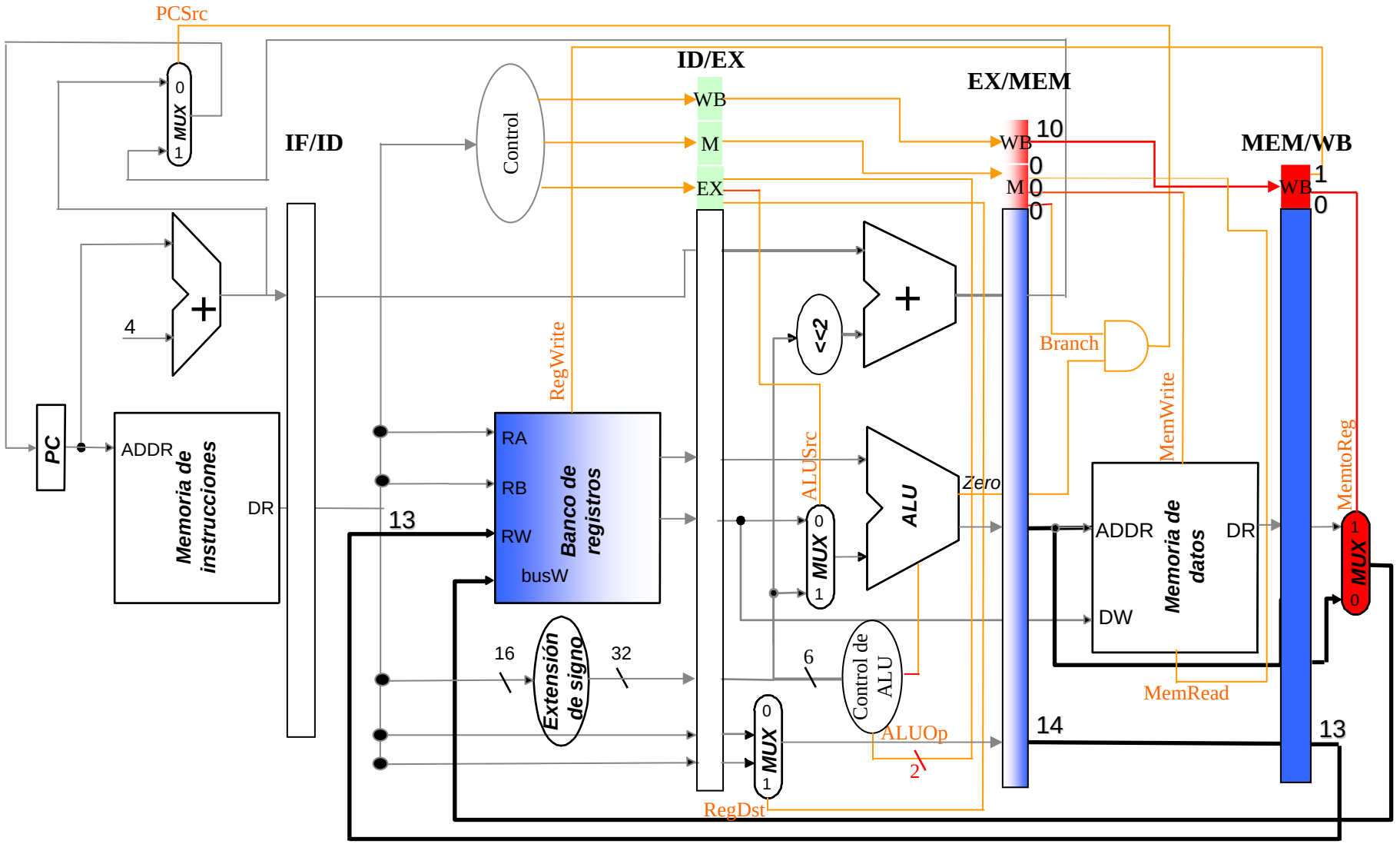
11.- Control de la ruta de datos segmentada

IF: <2> ID: <1> EX: add $14,… MEM: or $13, … WB: and $12,…



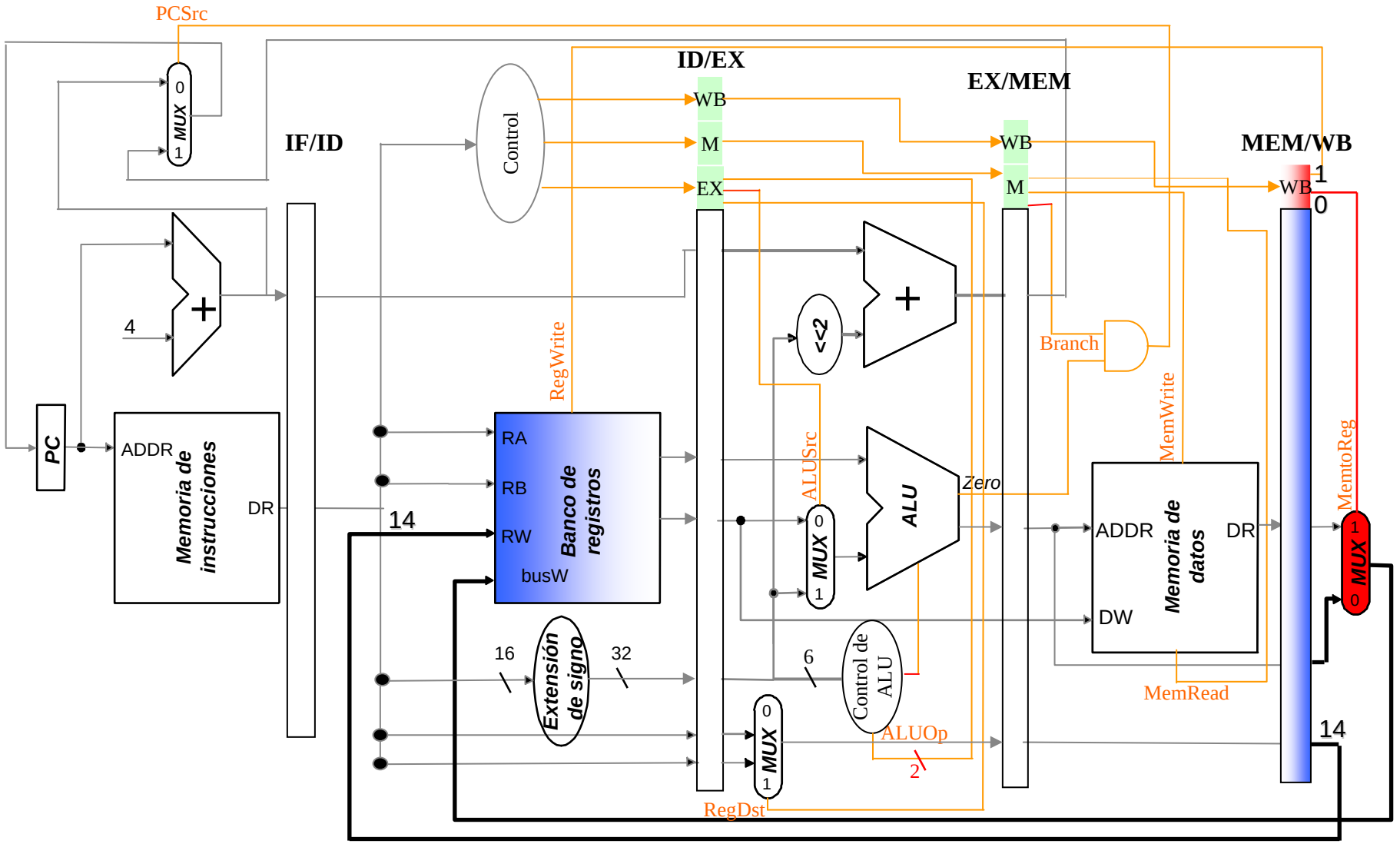
11.- Control de la ruta de datos segmentada

IF: <3> ID: <2> EX: <1> MEM: add $14, … WB: or $13,…

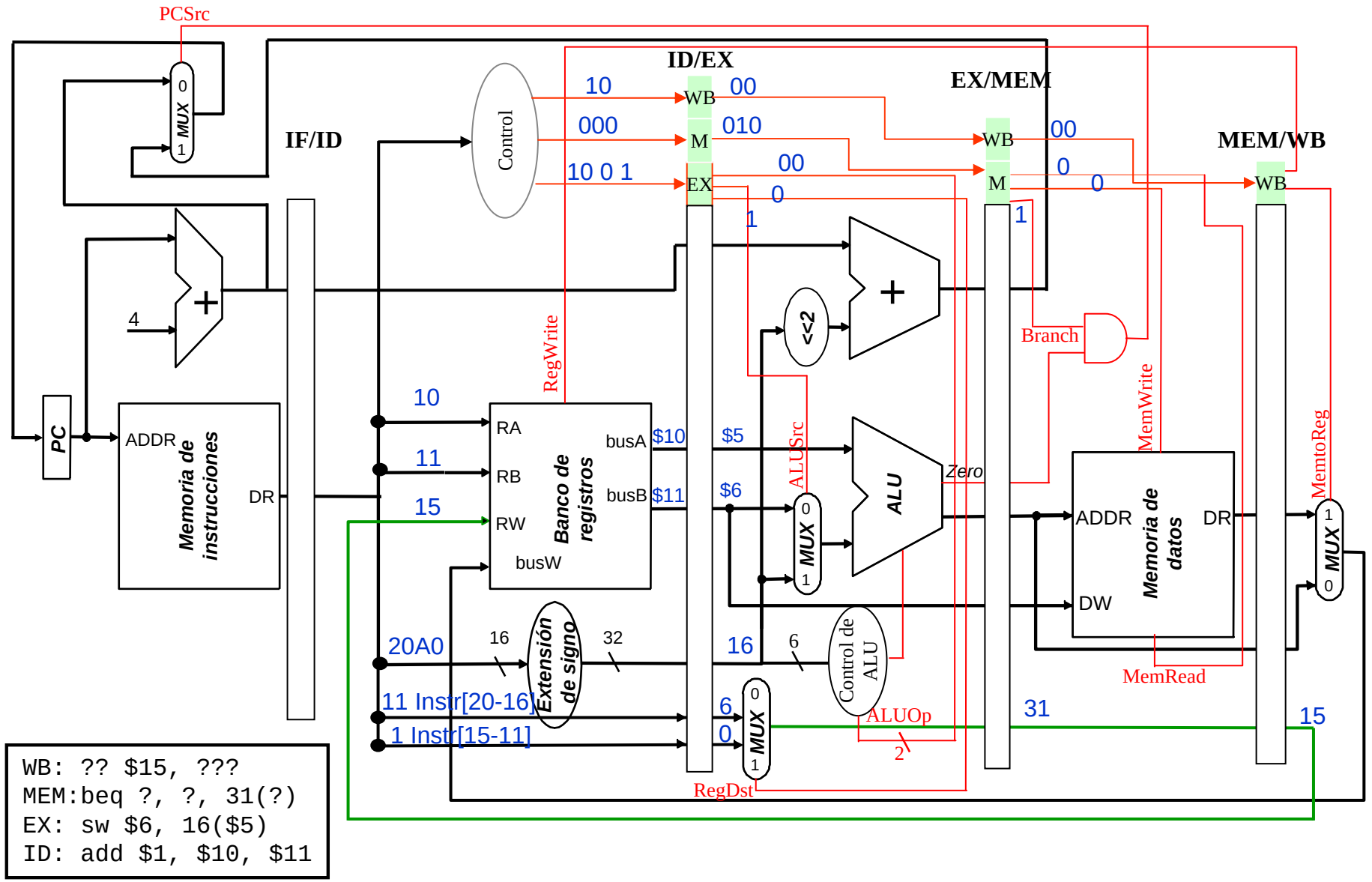


11.- Control de la ruta de datos segmentada

IF: <4> ID: <3> EX: <2> MEM: <1> WB: add $14,…



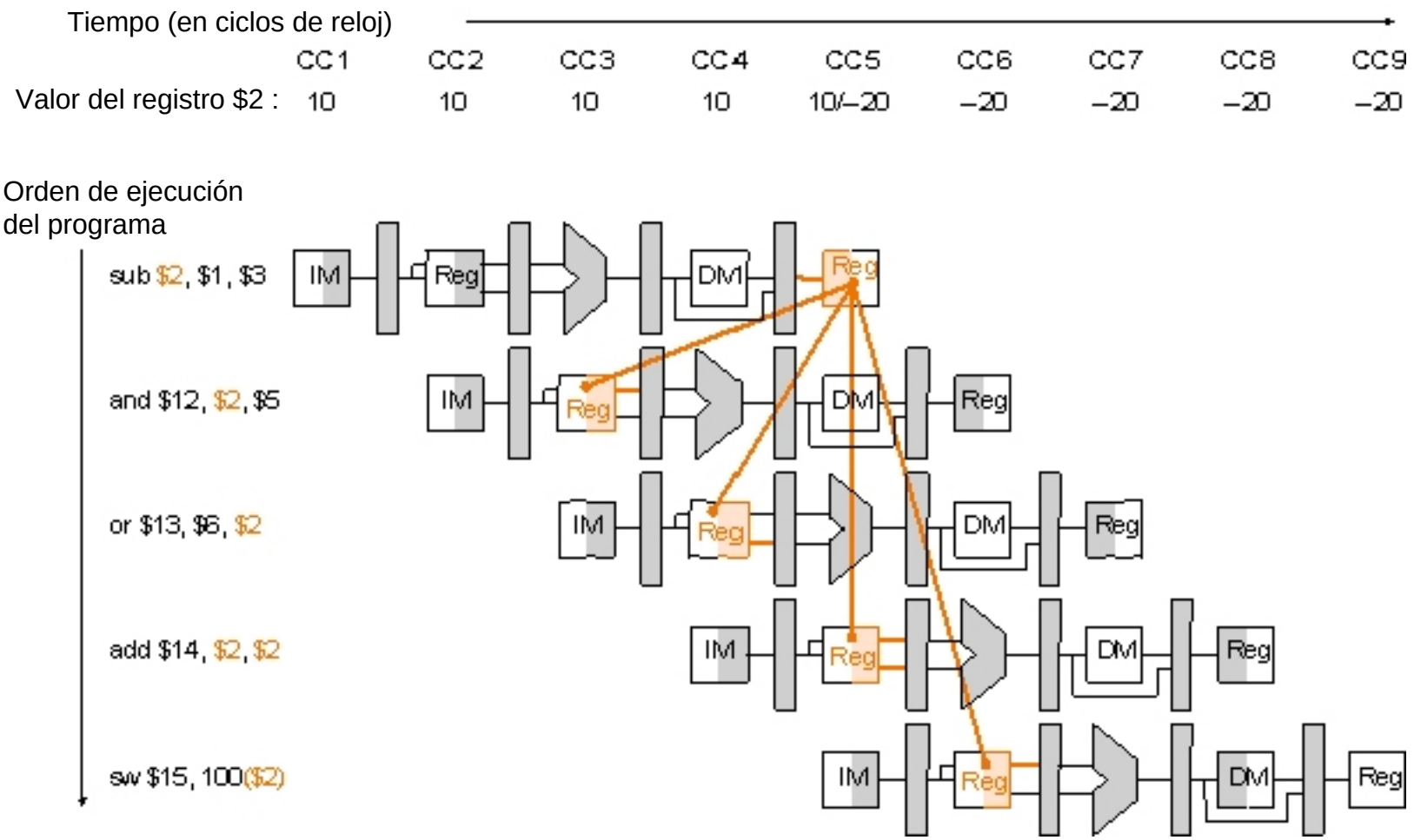
11.- Control de la ruta de datos segmentada




```
WB: ?? $15, ???
MEM:beq ?, ?, 31(?)
EX: sw $6, 16($5)
ID: add $1, $10, $11
```

Dependencias de datos

Dependencias de datos en segmentación para la ejecución de una secuencia de cinco instrucciones



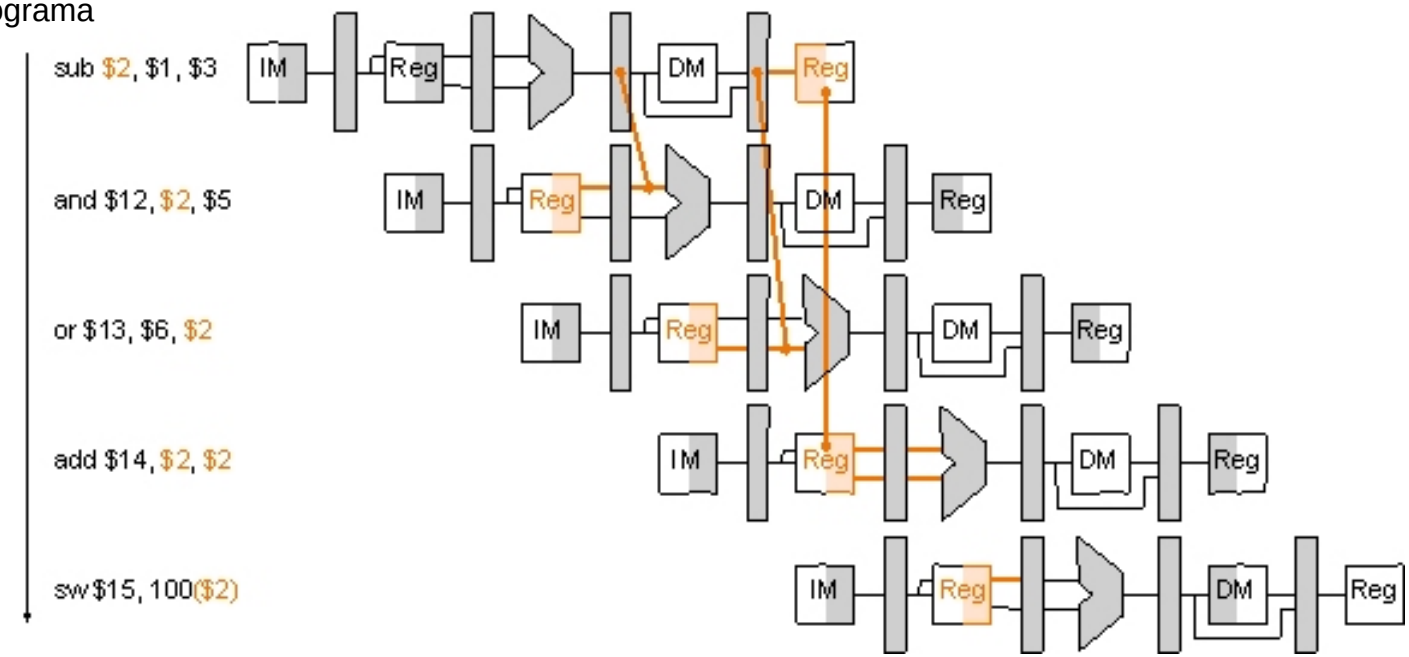
Dependencias de datos (anticipación)

Dependencias de datos en segmentación resueltas por anticipación

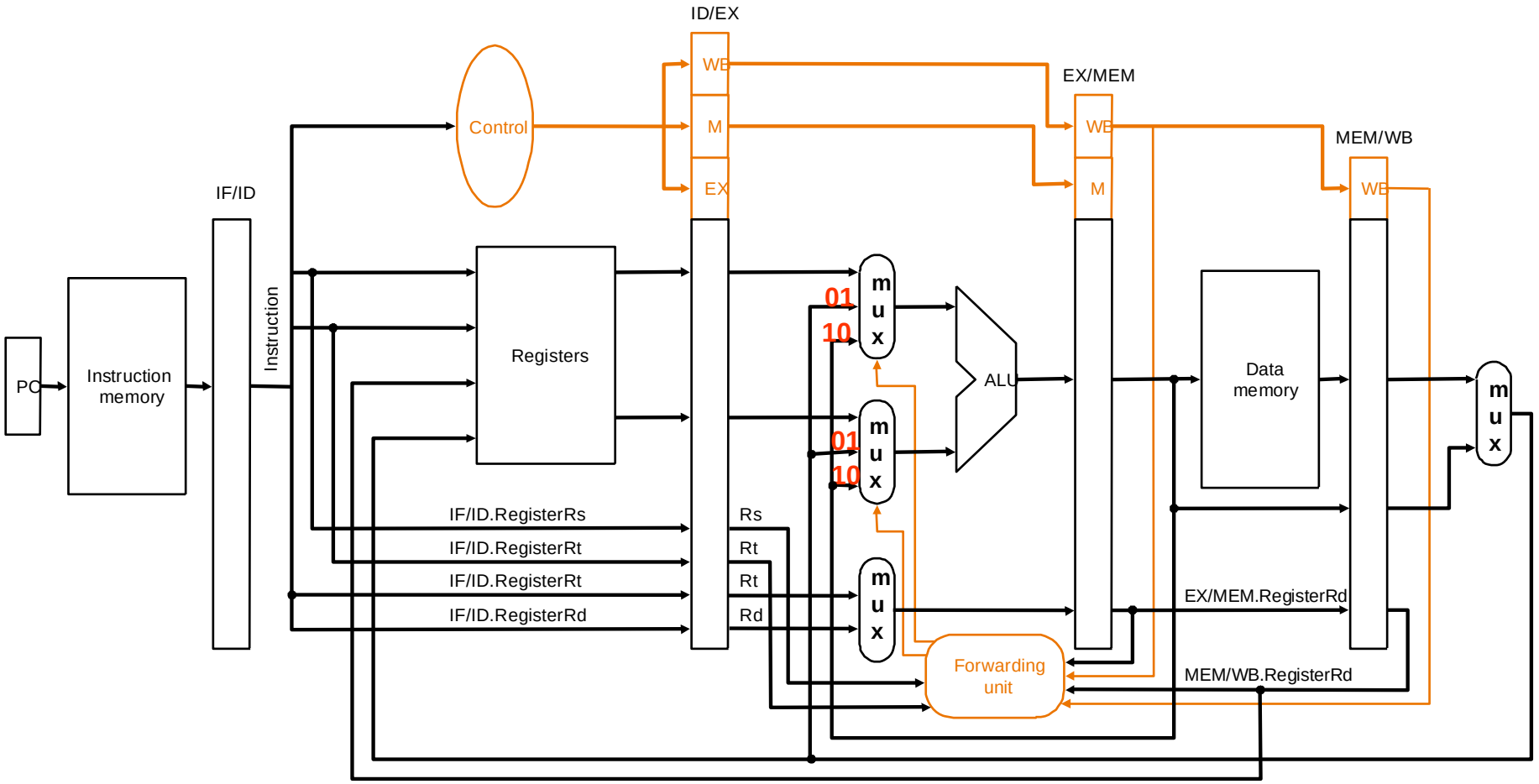
Tiempo (en ciclos de reloj)

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Valor del registro $2 :	10	10	10	10	10/–20	–20	–20	–20	–20
Valor de EX/MEM :	X	X	X	–20	X	X	X	X	X
Valor de MEM/WB :	X	X	X	X	–20	X	X	X	X

Orden de ejecución del programa



MIPS segmentado con anticipación de datos



MIPS segmentado con anticipación de datos

Riesgo EX:

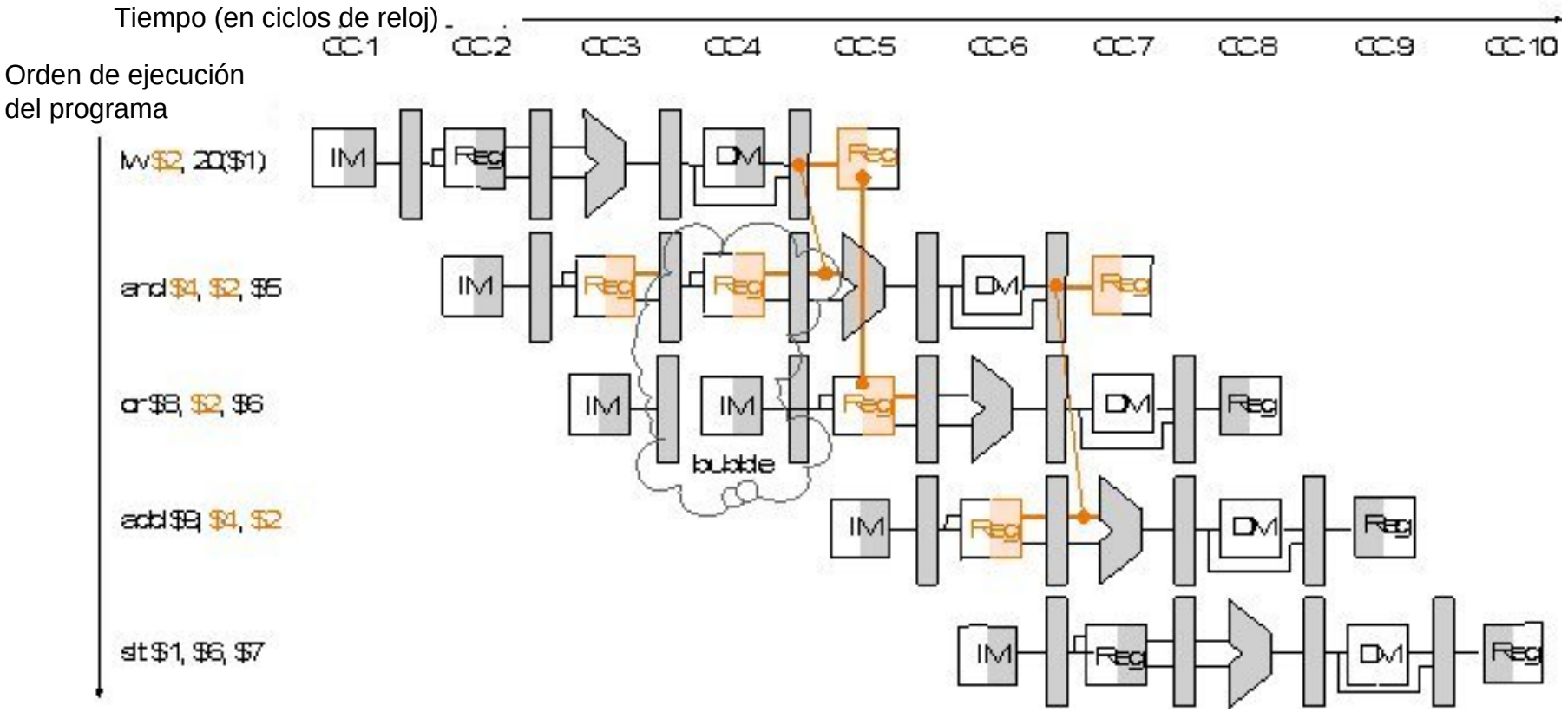
```
if      (XM.RegWrite &&
        (XM.Rd != 0)
        && (XM.Rd == DX.Rs))
                { anticiparA = 10}

if      (XM.RegWrite &&
        (XM.Rd != 0)
        && (XM.Rd == DX.Rt))
                { anticiparB = 10}
```

Riesgo MEM:

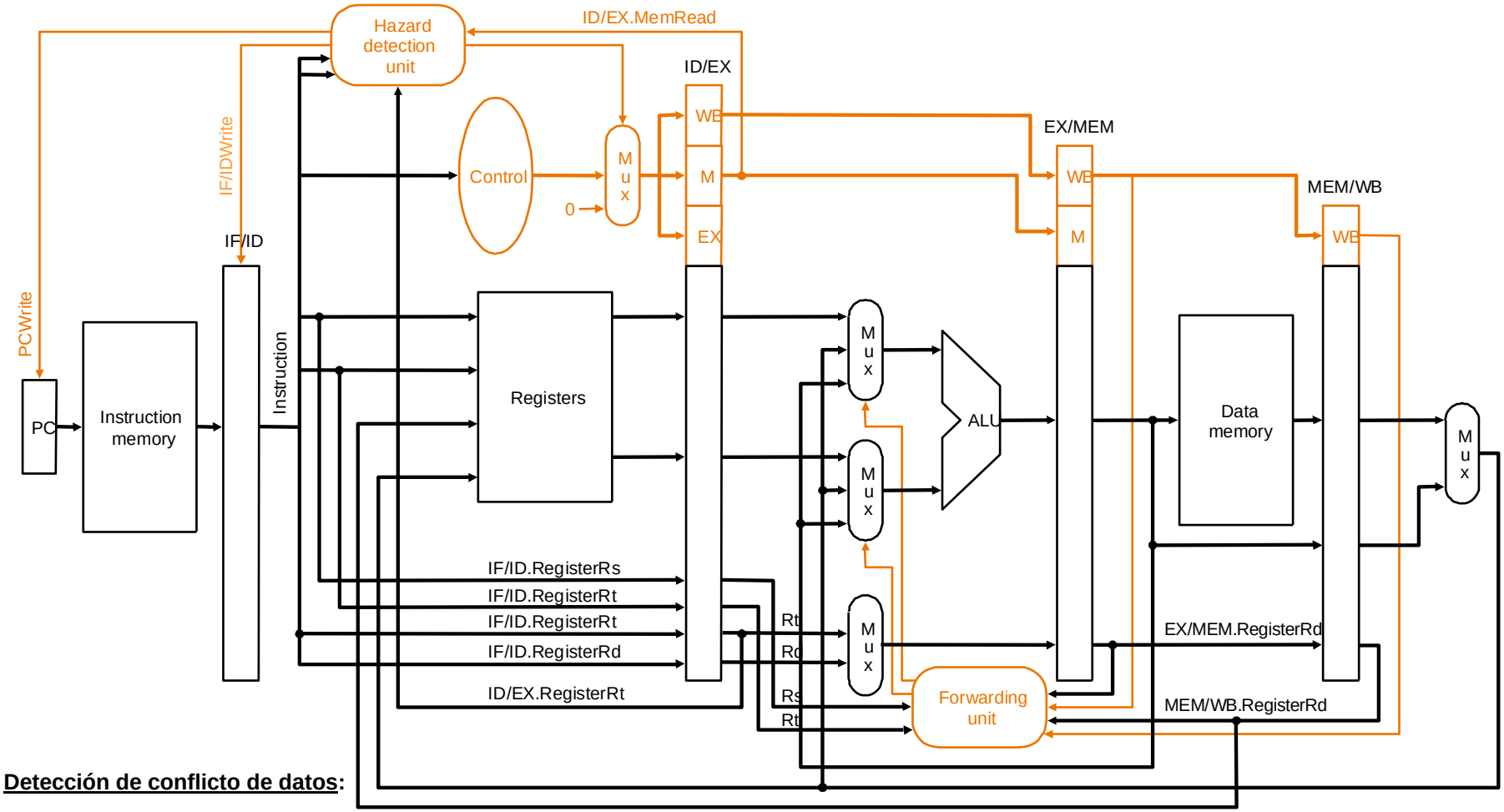
```
if      (MW.RegWrite &&
        (MW.Rd != 0) &&
        (XM.Rd != DX.Rs) &&
        (MW.Rd == DX.Rs))
                { anticiparA = 01}

if      (MW.RegWrite &&
        (MW.Rd != 0) &&
        (XM.Rd != DX.Rt) &&
        (MW.Rd == DX.Rt))
                { anticiparB = 01}
```

Dependencias de datos (bloqueo)

Dependencias de datos en segmentación resueltas por bloqueo



MIPS segmentado con anticipación y bloqueo

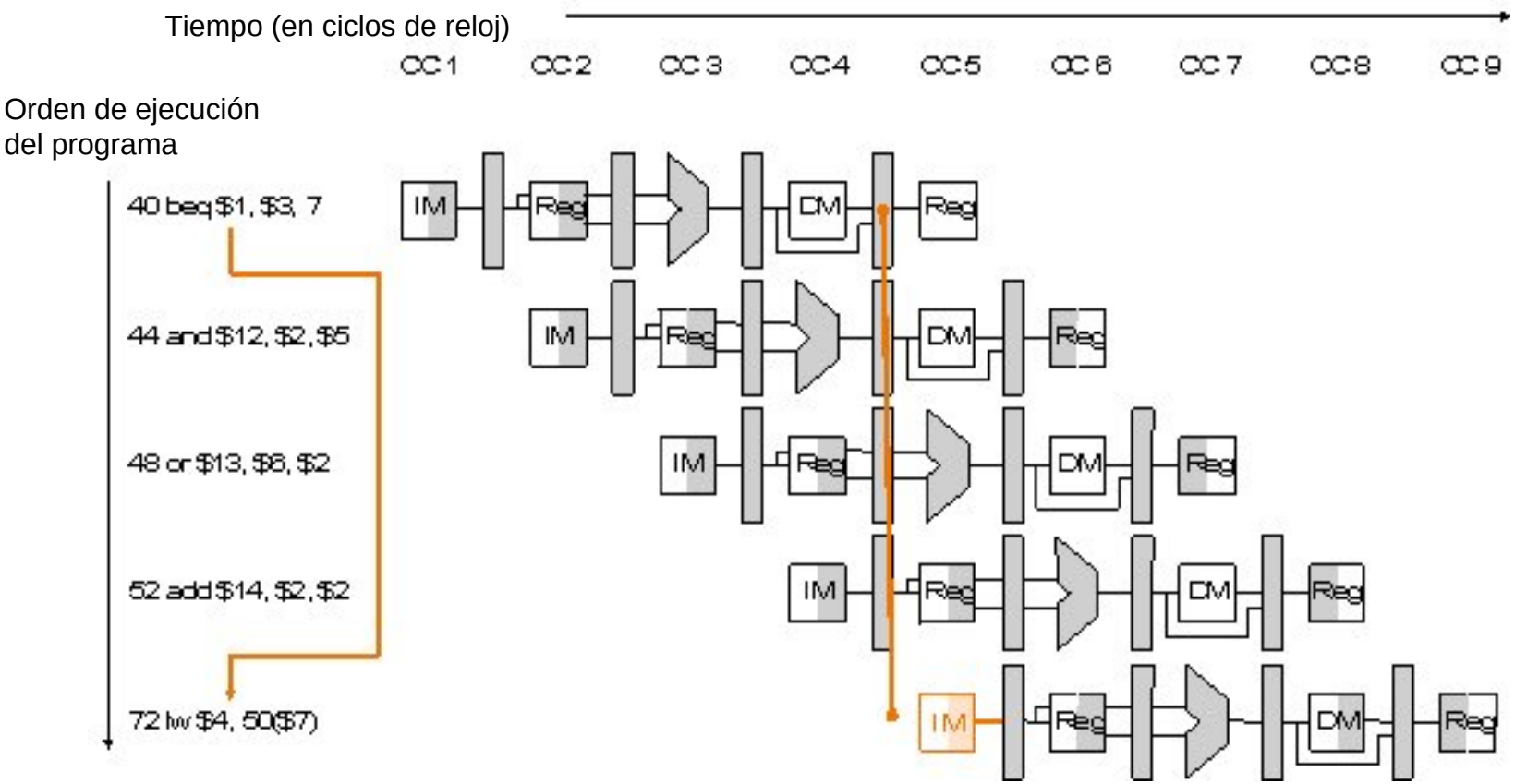


Detección de conflicto de datos:

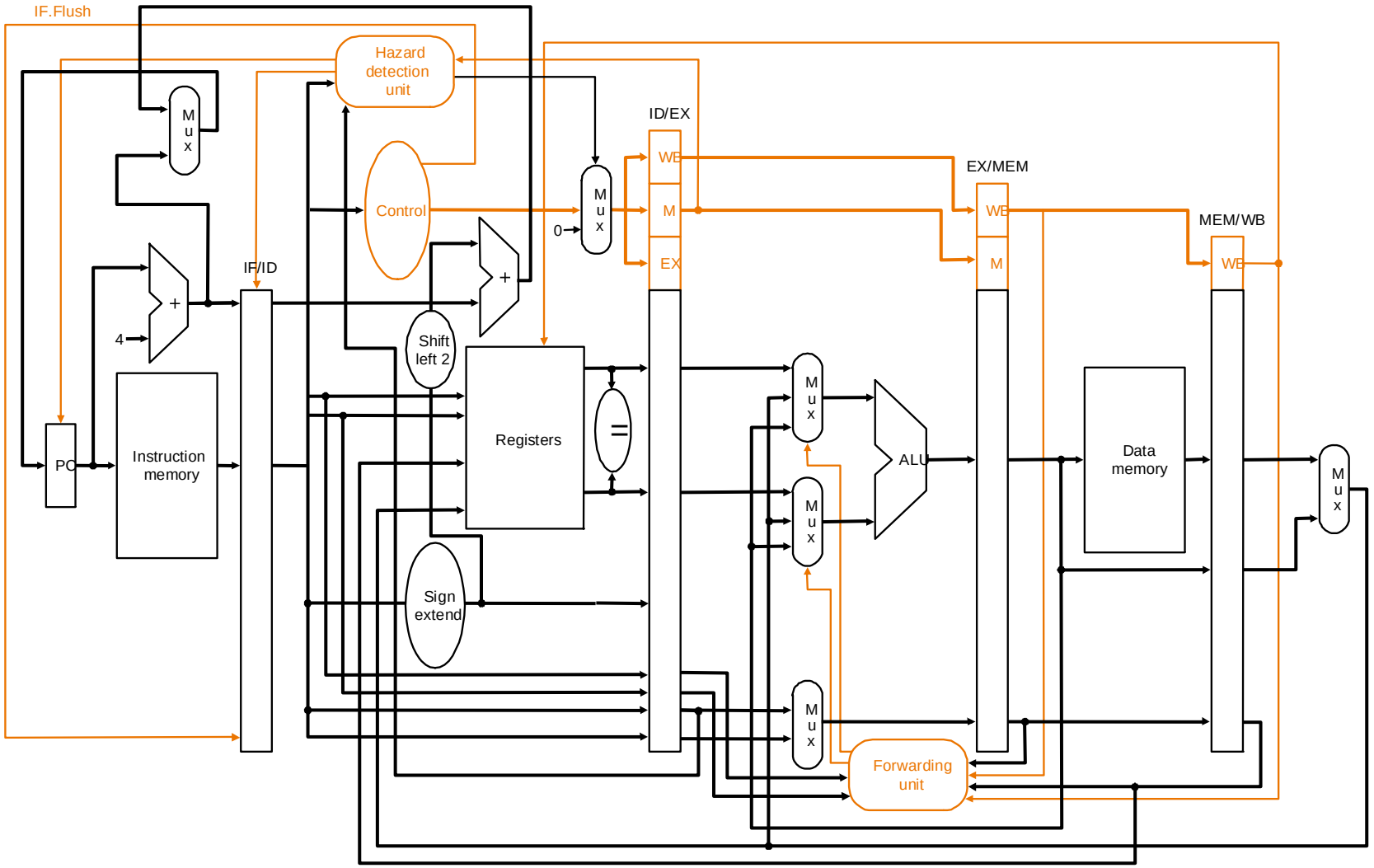
```
if (DX.MemRead
   && ((DX.Rt == FD.Rs) || (DX.Rt == FD.Rt)))
   { bloquear el pipeline;}
```

Dependencias de control/salto

Impacto de la segmentación en las instrucciones de salto

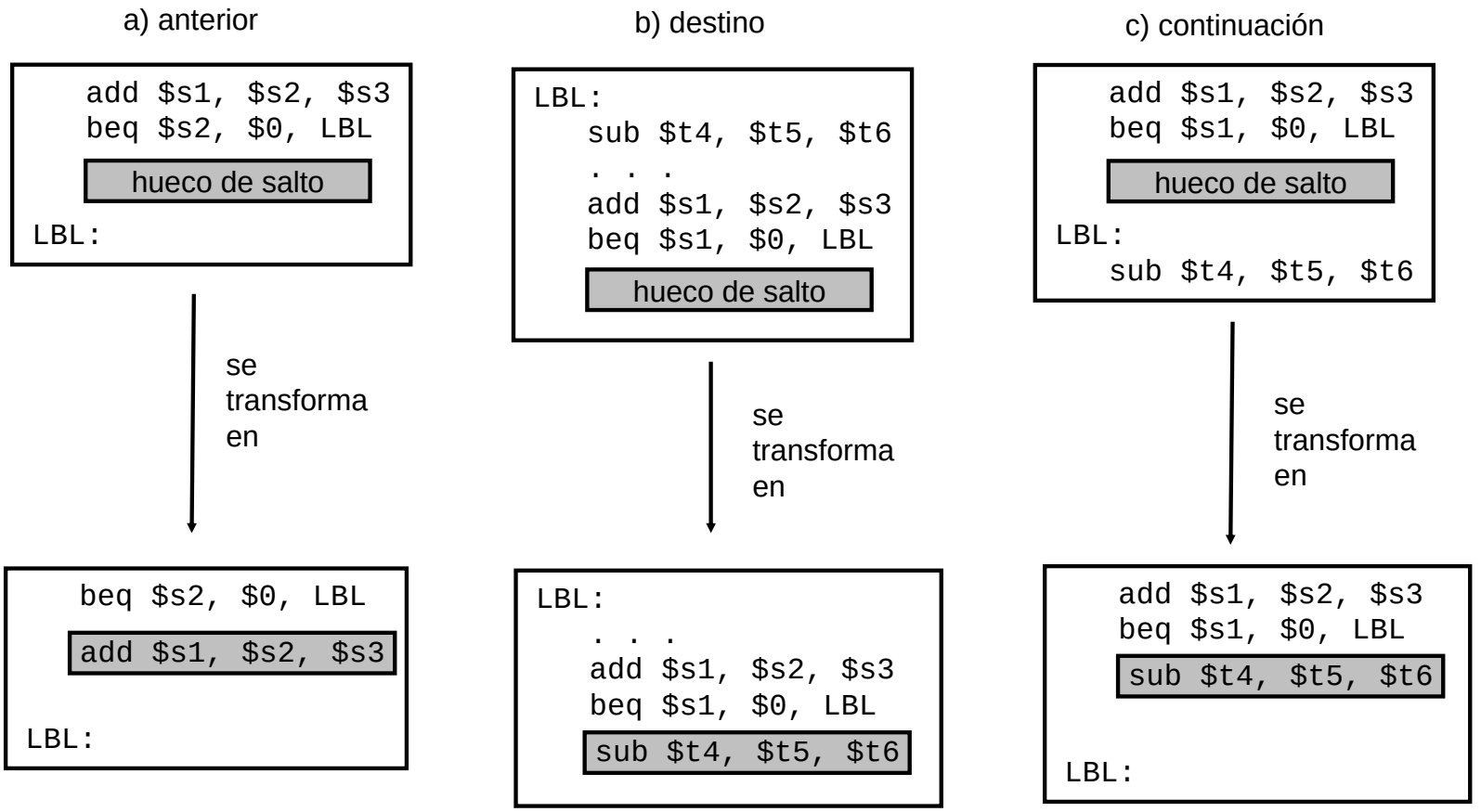


MIPS segmentado con tratamiento de riesgos de datos y de control



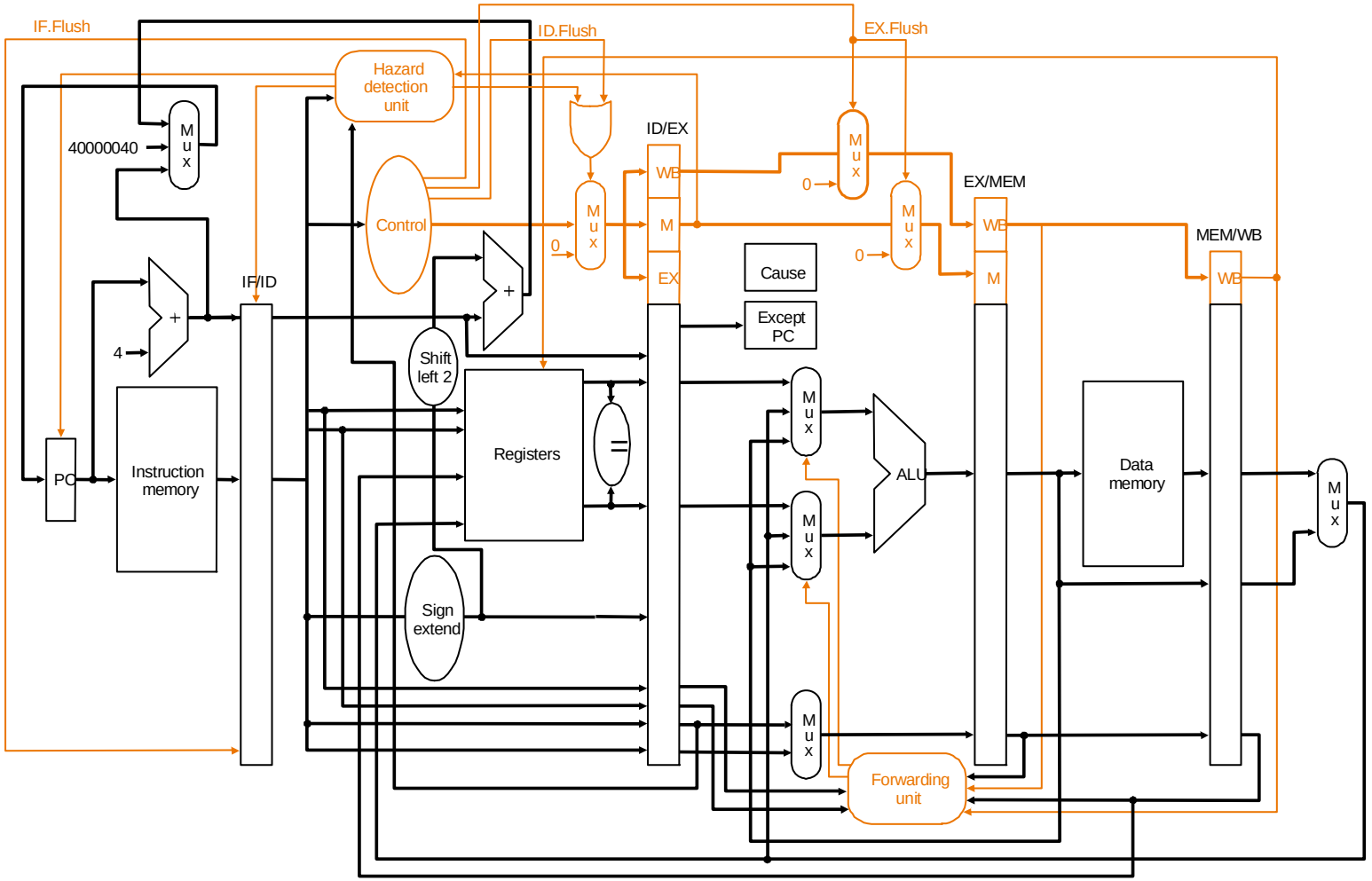
Tratamiento del riesgo de control:
- **adelantamiento de la decisión a ID (¡Cuidado con la anticipación de operandos!)**
- **si hay salto, anular la instrucción siguiente (IF.Flush)**

Gestión del hueco de salto retardado



Un salto retardado con un *pipeline* de un hueco siempre ejecuta la instrucción que hay después del salto, mientras que la segunda siguiente ya será dependiente del salto.
Es labor del compilador rellenar ese hueco con una instrucción válida y útil. Si no fuera posible encontrarla, se incluiría una NOP (en MIPS una instrucción "`sll $0, $0, 0`")

MIPS segmentado con tratamiento de riesgos de datos y de control, y de excepciones



Para gestionar excepciones, hay que vaciar el *pipeline* de las instrucciones incompletas que siguen a la que ha provocado la excepción. Para ello se incluyen en el diagrama:

Registros "Cause" y EPC; dirección constante de salto a tratamiento de excepción "4000040" y generación de las señales de control IF.Flush, ID.Flush, EX.Flush que vacían el *pipeline*