PARTE 1:
Introducción a los Sistemas Operativos y a su relación con la Arquitectura de Computadores

Sección 4. La CPU: funcionamiento interno

Introducción

Las CPU actuales son muy complejas, implementando además mecanismos como el *pipelining* que aumentan todavía más dicha complejidad.

Entender el funcionamiento básico de una CPU, si ésta no es complicada (juego de instrucciones reducido, pocos modos de direccionamiento, sin interrupciones, etc.) es relativamente sencillo.

Aquí vamos a usar una CPU bastante simple, basada en D-CORE (de Demo-CORE), un diseño de propósito educativo de la Universidad de Hamburgo.

Características de D-CORE

Simplificación procesador M-CORE (Motorola, 1997).
Procesador RISC.
Palabra de 16 bits.
Bus datos y direcciones de 16 bits.
Instrucciones de una palabra.
Un solo modo (“modo supervisor”).
Un solo flag (C=>Carry).
Unidad de control con lógica microprogramada.
E/S mapeada en memoria.
Sin DMA ni interrupciones.

Inciso: Mecanismo simplificado de interrupción

1. La CPU ejecuta un programa normalmente.
2. Un dispositivo requiere atención y solicita una interrupción a la CPU.
3. La CPU deja de ejecutar el programa y pasa a ejecutar otro programa denominado ISR (Interrupt Service Routine).
4. La ISR atiende al dispositivo y da por finalizada la interrupción.
5. La CPU reanuda la ejecución del programa en el punto en el que fue interrumpida.

Diagrama de bloques del sistema

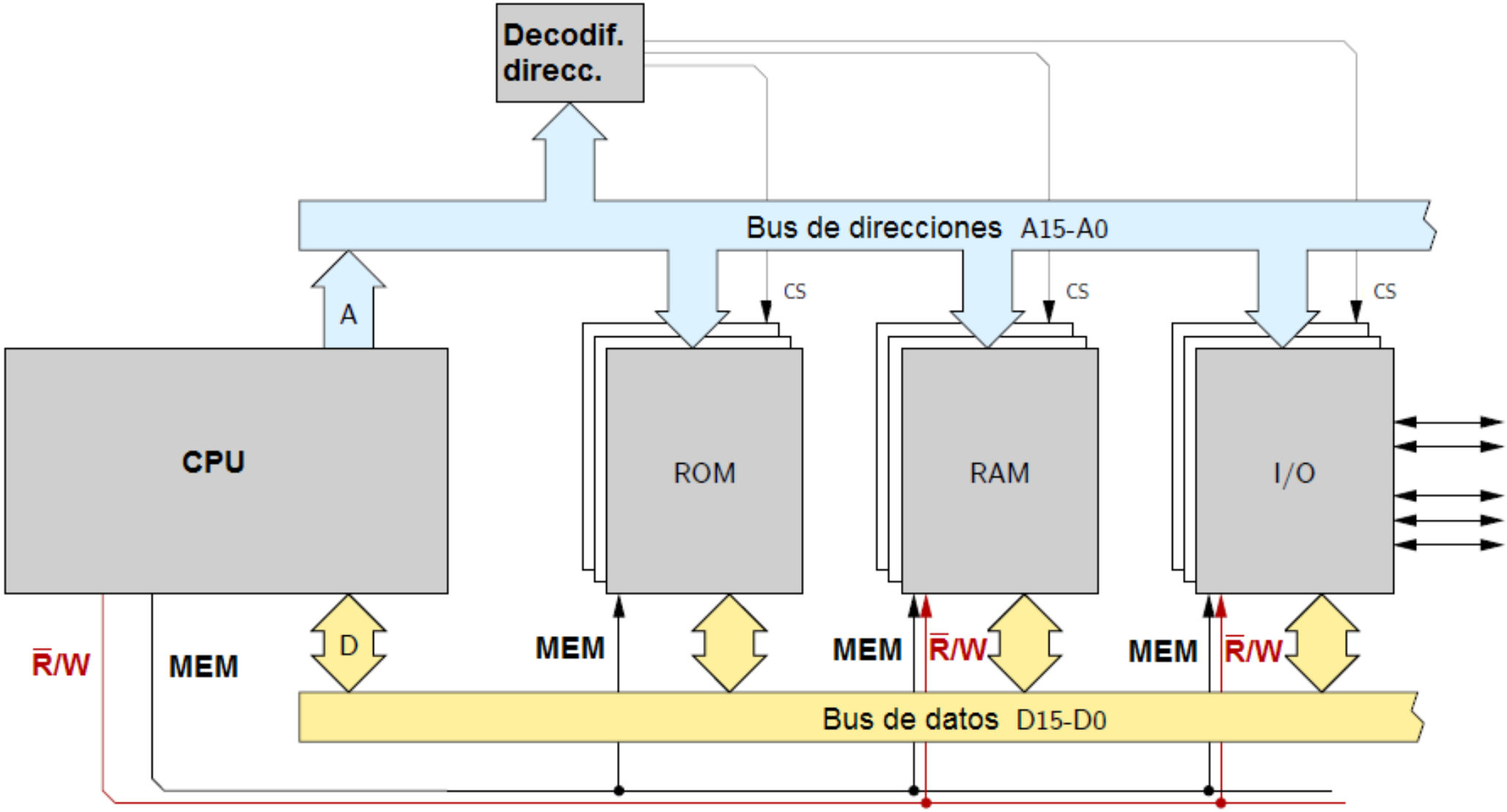
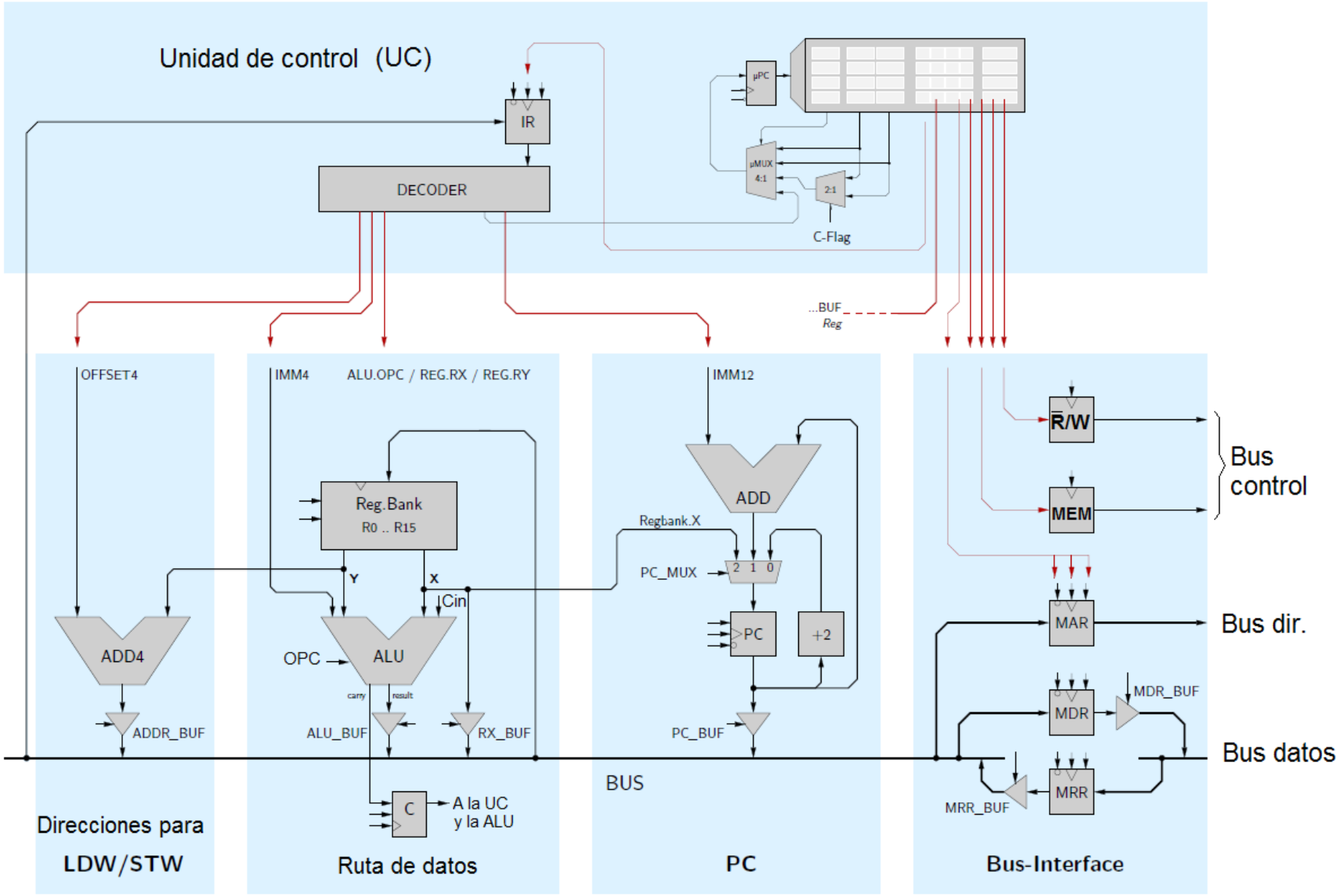
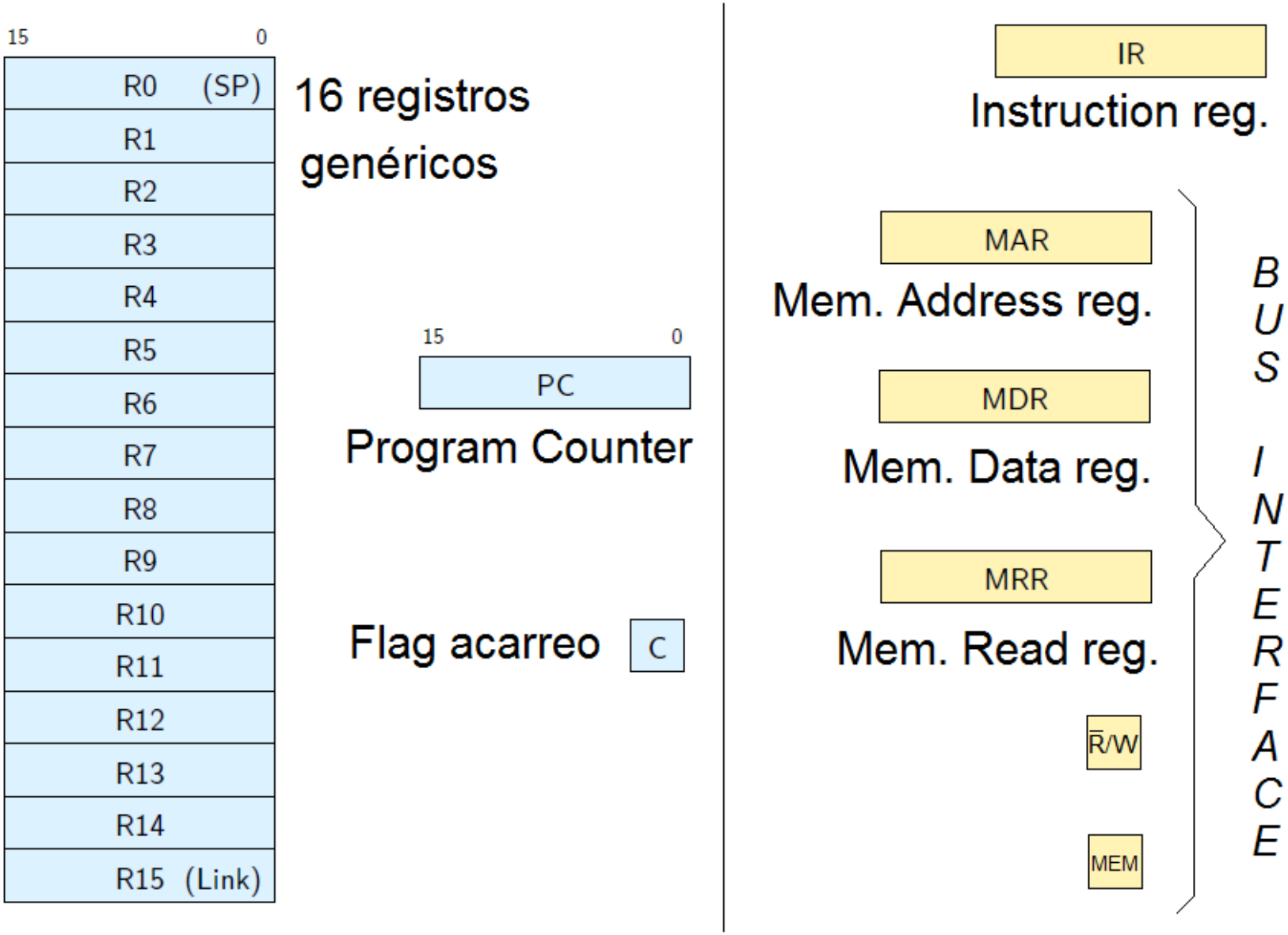


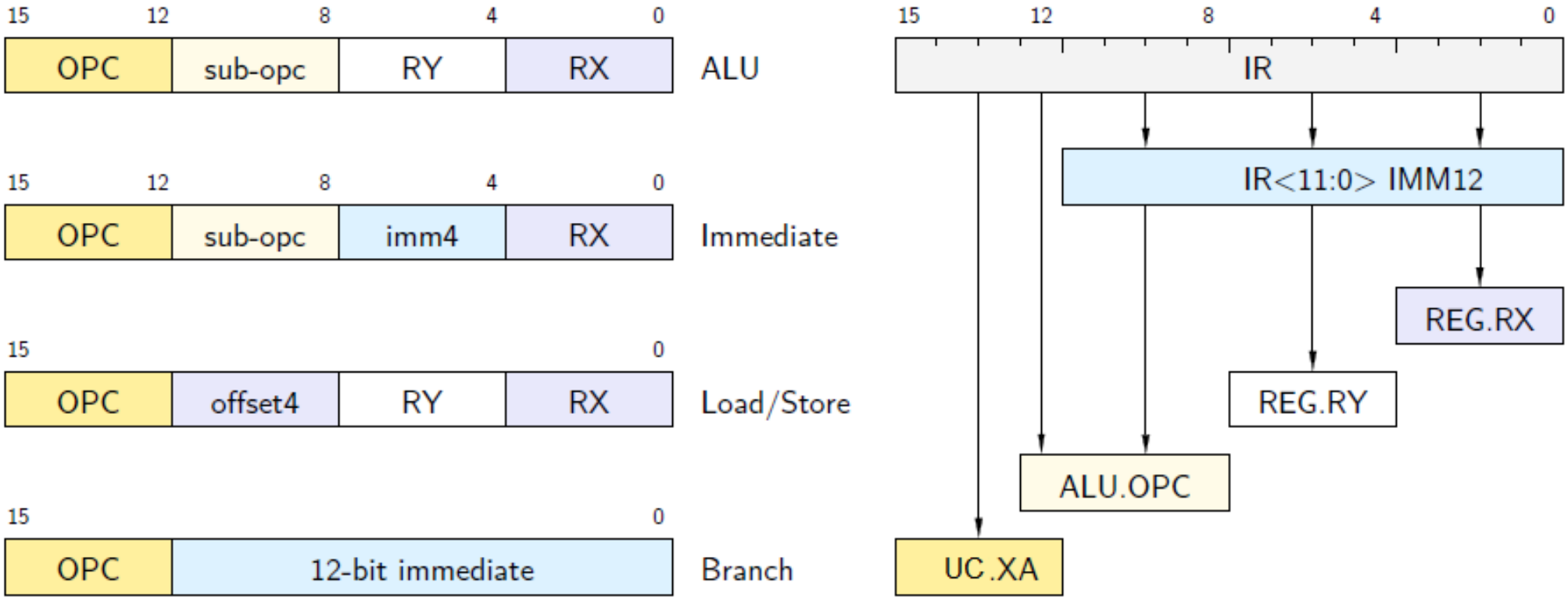
Diagrama de bloques de D-CORE



Registros


15
0
R0 (SP)
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
R15 (Link)
16 registros genéricos
15
0
PC
Program Counter
Flag acarreo
C
IR
Instruction reg.
MAR
Mem. Address reg.
MDR
Mem. Data reg.
MRR
Mem. Read reg.
R/W
MEM
BUS INTERFACE

Formatos de instrucción y decodific. IR



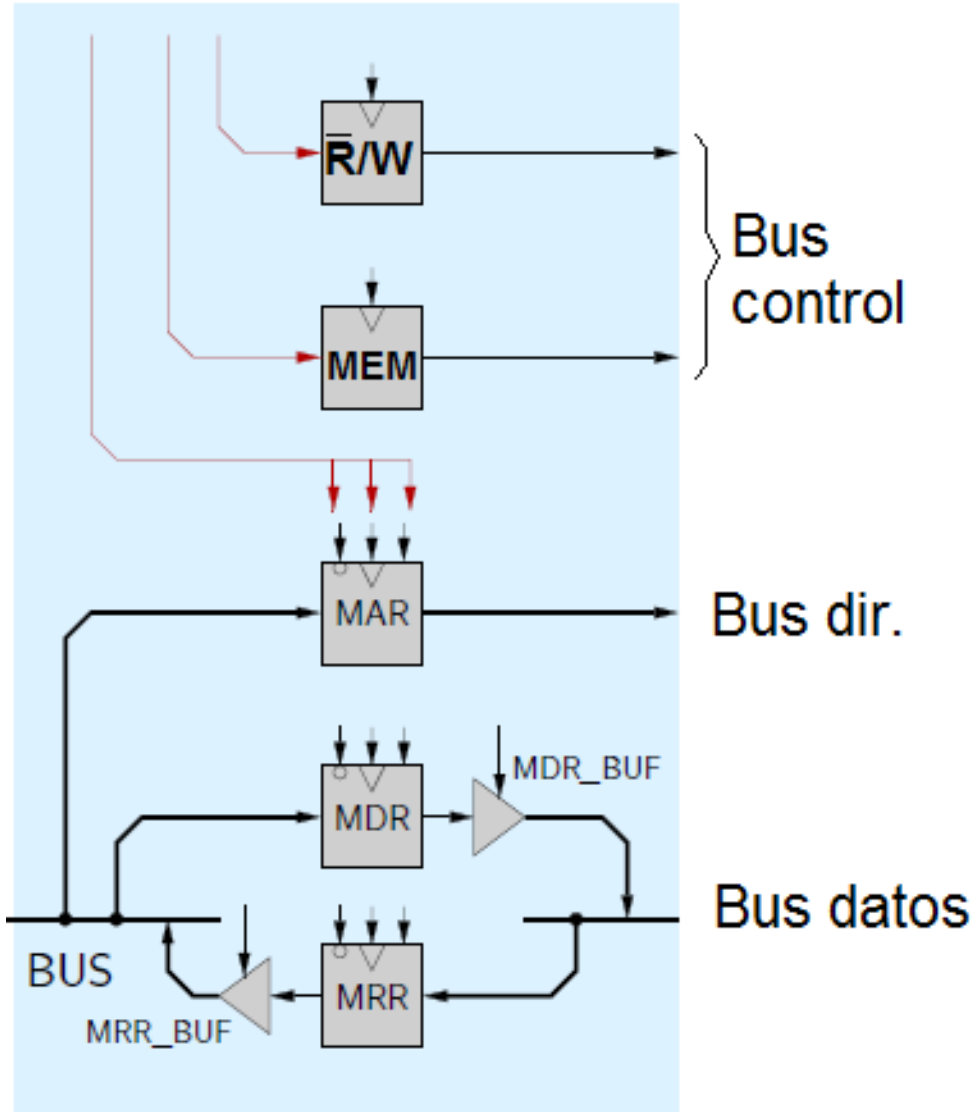
Juego de instrucciones

Mnemonic	Coding	Hex	Meaning
			ALU-Operation
mov	0010 0000 ⟨yyyy⟩ ⟨xxxx⟩	20yx	R[x] = R[y]
addu	0010 0001 ⟨yyyy⟩ ⟨xxxx⟩	21yx	R[x] = R[x] + R[y]
addc	0010 0010 ⟨yyyy⟩ ⟨xxxx⟩	22yx	R[x] = R[x] + R[y] + C;
subu	0010 0011 ⟨yyyy⟩ ⟨xxxx⟩	23yx	R[x] = R[x] - R[y]
and	0010 0100 ⟨yyyy⟩ ⟨xxxx⟩	24yx	R[x] = R[x] AND R[y]
or	0010 0101 ⟨yyyy⟩ ⟨xxxx⟩	25yx	R[x] = R[x] OR R[y]
xor	0010 0110 ⟨yyyy⟩ ⟨xxxx⟩	26yx	R[x] = R[x] XOR R[y]
not	0010 0111 ⟨****⟩ ⟨xxxx⟩	27*x	R[x] = NOT R[x]
			Shift-Operation
lsl	0010 1000 ⟨yyyy⟩ ⟨xxxx⟩	28yx	R[x] = R[x] << R[y].<3:0>
lsr	0010 1001 ⟨yyyy⟩ ⟨xxxx⟩	29yx	R[x] = R[x] >>> R[y].<3:0>
asr	0010 1010 ⟨yyyy⟩ ⟨xxxx⟩	2Ayx	R[x] = R[x] >> R[y].<3:0>
lslc	0010 1100 ⟨****⟩ ⟨xxxx⟩	2C*x	R[x] = R[x] << 1, C=R[X].15
lsrc	0010 1101 ⟨****⟩ ⟨xxxx⟩	2D*x	R[x] = R[x] >>> 1, C=R[X].0
asrc	0010 1110 ⟨****⟩ ⟨xxxx⟩	2E*x	R[x] = R[x] >> 1, C=R[X].0
			Compare-Operation
cmpe	0011 0000 ⟨yyyy⟩ ⟨xxxx⟩	30yx	C = (R[x] == R[y])
cmpne	0011 0001 ⟨yyyy⟩ ⟨xxxx⟩	31yx	C = (R[x] != R[y])
cmpgt	0011 0010 ⟨yyyy⟩ ⟨xxxx⟩	32yx	C = (R[x] > R[y]) (signed)
cmplt	0011 0011 ⟨yyyy⟩ ⟨xxxx⟩	33yx	C = (R[x] < R[y]) (signed)

			Immediate-Operation
movi	0011 0100 ⟨cccc⟩ ⟨xxxx⟩	34cx	R[x] = 0x000c
addi	0011 0101 ⟨cccc⟩ ⟨xxxx⟩	35cx	R[x] = R[x] + 0x000⟨c⟩
subi	0011 0110 ⟨cccc⟩ ⟨xxxx⟩	36cx	R[x] = R[x] - 0x000⟨c⟩
andi	0011 0111 ⟨cccc⟩ ⟨xxxx⟩	37cx	R[x] = R[x] AND 0x000⟨c⟩
lsli	0011 1000 ⟨cccc⟩ ⟨xxxx⟩	38cx	R[x] = R[x] << ⟨c⟩
lsri	0011 1001 ⟨cccc⟩ ⟨xxxx⟩	39cx	R[x] = R[x] >>> ⟨c⟩
bseti	0011 1010 ⟨cccc⟩ ⟨xxxx⟩	3Acx	R[x] = R[x] | (1 << ⟨c⟩) (set bit)
bclri	0011 1011 ⟨cccc⟩ ⟨xxxx⟩	3Bcx	R[x] = R[x] & !(1 << ⟨c⟩) (clear bit)
			Memory-Operation
ldw	0100 ⟨cccc⟩ ⟨yyyy⟩ ⟨xxxx⟩	4cyx	R[x] = MEM(R[y] + (0x000⟨c⟩<<1))
stw	0101 ⟨cccc⟩ ⟨yyyy⟩ ⟨xxxx⟩	5cyx	MEM(R[y] + (0x000⟨c⟩<<1)) = R[x]
			Flow Control
br	1000 ⟨iiii⟩ ⟨iiii⟩ ⟨iiii⟩	8iii	PC = PC+2+⟨imm12⟩
jsr	1001 ⟨iiii⟩ ⟨iiii⟩ ⟨iiii⟩	9iii	R[15] = PC+2; PC = PC+2+⟨imm12⟩ (call)
bt	1010 ⟨iiii⟩ ⟨iiii⟩ ⟨iiii⟩	Aiii	(C=1) ? PC = PC+2+⟨imm12⟩ : PC=PC+2
bf	1011 ⟨iiii⟩ ⟨iiii⟩ ⟨iiii⟩	Biii	(C=0) ? PC = PC+2+⟨imm12⟩ : PC=PC+2
jmp	1100 ⟨****⟩ ⟨****⟩ ⟨xxxx⟩	C**x	PC = R[x]
halt	1111 ⟨****⟩ ⟨****⟩ ⟨****⟩	F***	halt
			Other Opcodes illegal

<*xxxx*>, x: Índice de 4 bits registro origen y destino RX
<*yyyy*>, y: Índice de 4 bits registro origen RY
<*cccc*>, c: Constante 4 bits (sin signo) IMM4
i: Constante de 12 bits con signo IMM12 <****>, *: Irrelevante

Detalle bus-interface



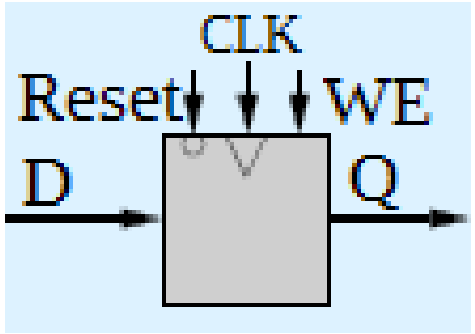
Notas:

$\overline{R}$/W y MEM registros de 1 bit, MAR, MDR y MRR de 16 bits.

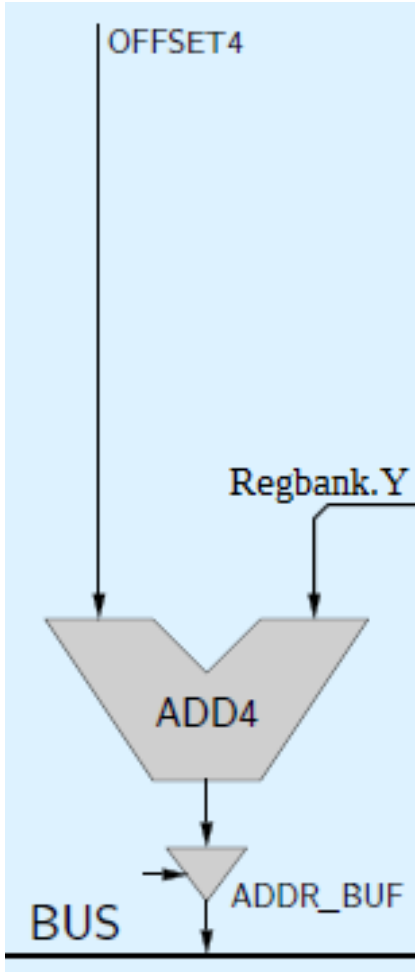
MBR => MDR + MRR

MDR => Memory Data Reg.

MRR => Memory Read Reg.

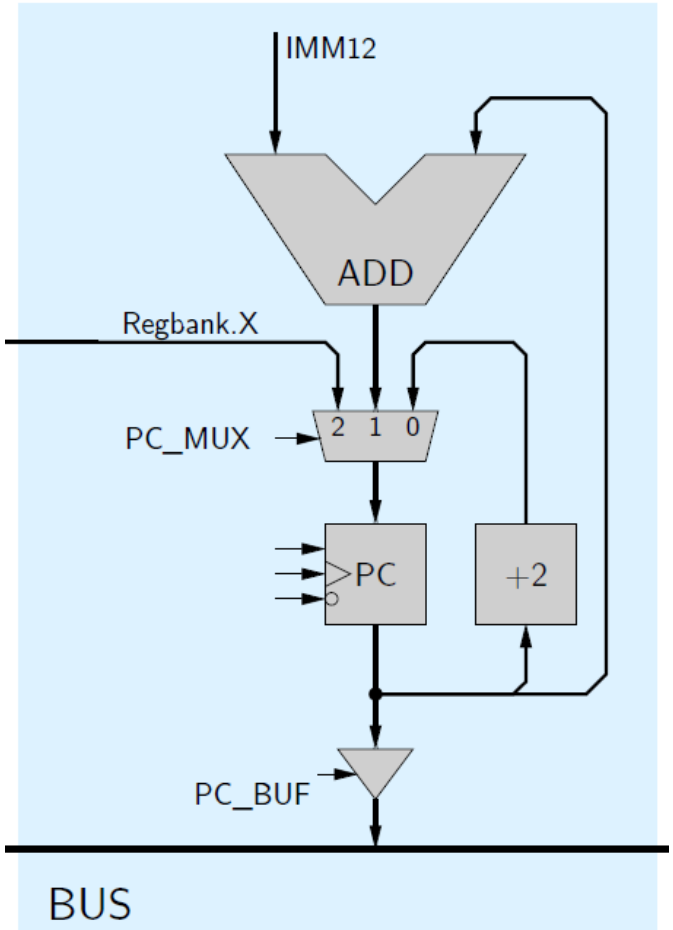


Detalle direcciones para LDW/STW



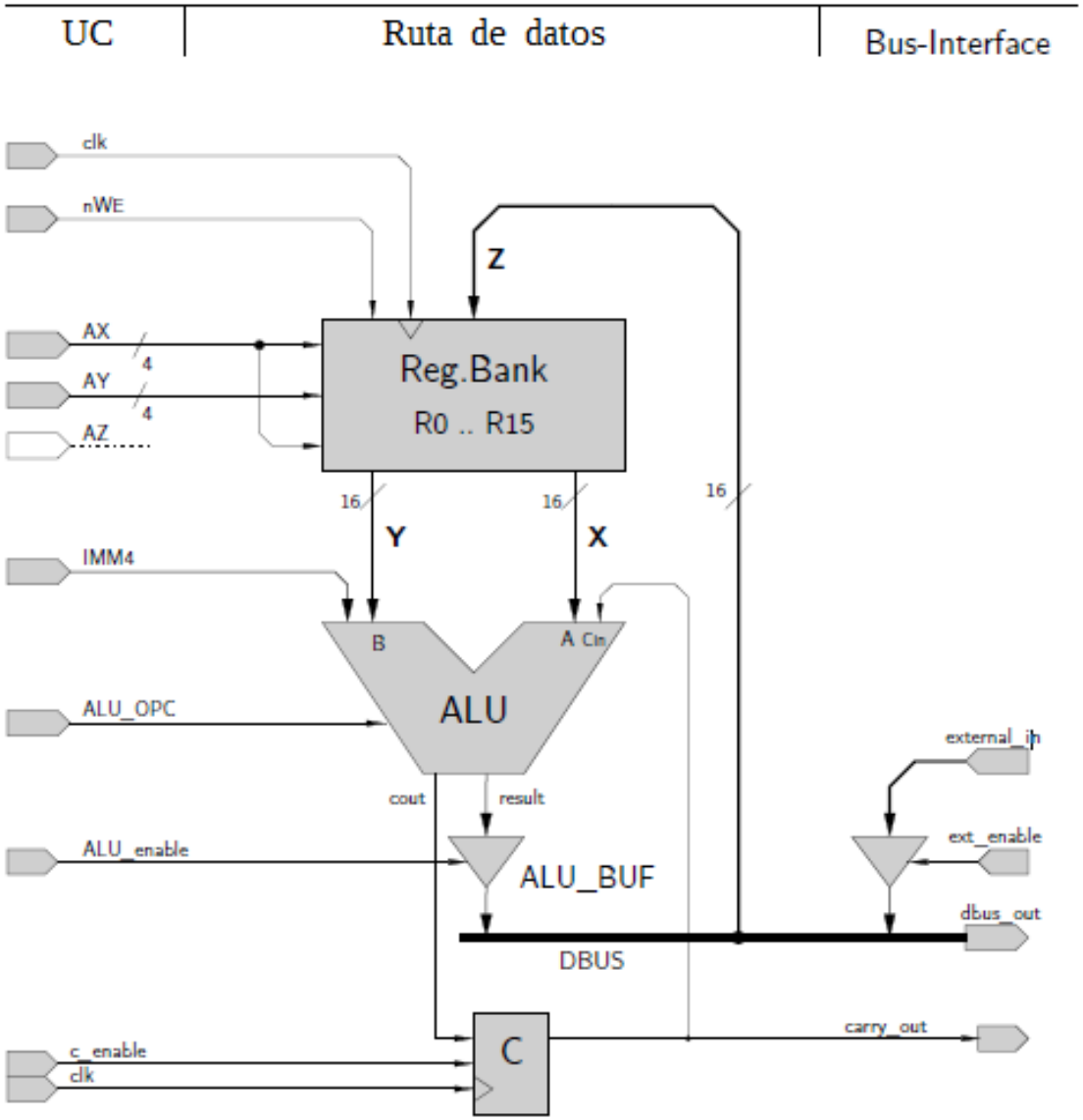
			Memory-Operation
ldw	0100 ⟨cccc⟩ ⟨yyyy⟩ ⟨xxxx⟩	4cyx	R[x] = MEM(R[y] + (0x000⟨c⟩<<1))
stw	0101 ⟨cccc⟩ ⟨yyyy⟩ ⟨xxxx⟩	5cyx	MEM(R[y] + (0x000⟨c⟩<<1)) = R[x]

Detalle conexionado PC



			Flow Control
br	1000 ⟨iiii⟩ ⟨iiii⟩ ⟨iiii⟩	8iii	PC = PC+2+⟨*imm12*⟩
jsr	1001 ⟨iiii⟩ ⟨iiii⟩ ⟨iiii⟩	9iii	R[15] = PC+2; PC = PC+2+⟨*imm12*⟩ (call)
bt	1010 ⟨iiii⟩ ⟨iiii⟩ ⟨iiii⟩	Aiii	(C=1) ? PC = PC+2+⟨*imm12*⟩ : PC=PC+2
bf	1011 ⟨iiii⟩ ⟨iiii⟩ ⟨iiii⟩	Biii	(C=0) ? PC = PC+2+⟨*imm12*⟩ : PC=PC+2
jmp	1100 ⟨****⟩ ⟨****⟩ ⟨xxxx⟩	C**x	PC = R[x]

Detalle conexionado ALU



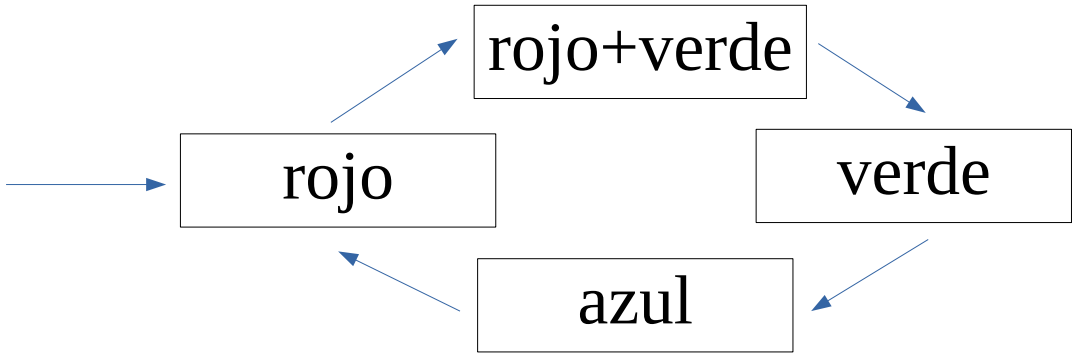
ALU_OPC	Result, Carry
0 0000	B
0 0001	A + B
0 0010	A+B+Cin, update C
0 0011	A - B
0 0100	A AND B
0 0101	A OR B
0 0110	A XOR B
0 0111	NOT A
0 1000	A << B.<3:0>
0 1001	A >>> B.<3:0>
0 1010	A >> B.<3:0>
0 1011	undefined
0 1100	A << 1, C = A.<15>
0 1101	A >>> 1, C = A.<0>
0 1110	A >> 1, C = A.<0>
0 1111	undefined
1 0000	A, C = (A == B)
1 0001	A, C = (A != B)
1 0010	A, C = (A > B)
1 0011	A, C = (A < B)
1 0100	IMM4
1 0101	A + IMM4
1 0110	A - IMM4
1 0111	A AND IMM4
1 1000	A << IMM4
1 1001	A >>> IMM4
1 1010	set bit IMM4 in A
1 1011	clear bit IMM4 in A
1 11**	undefined

Tipos de lógicas de la UC

Cableada: Por “hardware”, usando puertas. Más rápida. Aplicable si “poca” complejidad (RISC).

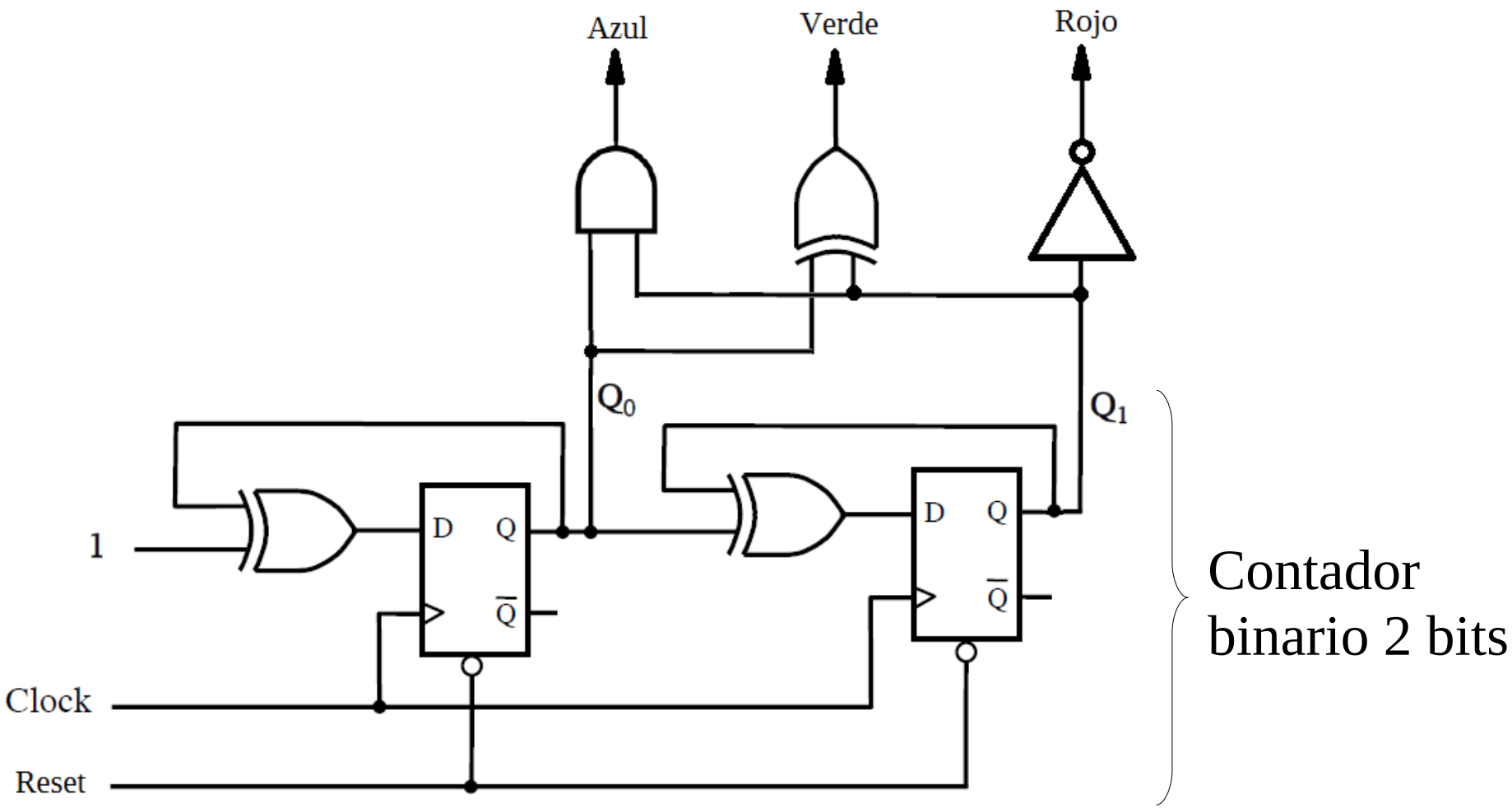
Microprogramada: Por “software”, usando una “ROM” interna. Más versátil. Aplicable RISC y CISC.

Vamos a ver un ejemplo de ambas para este diagrama de estados (generar señales para encender tres led -rojo, verde y azul- con esta secuencia):

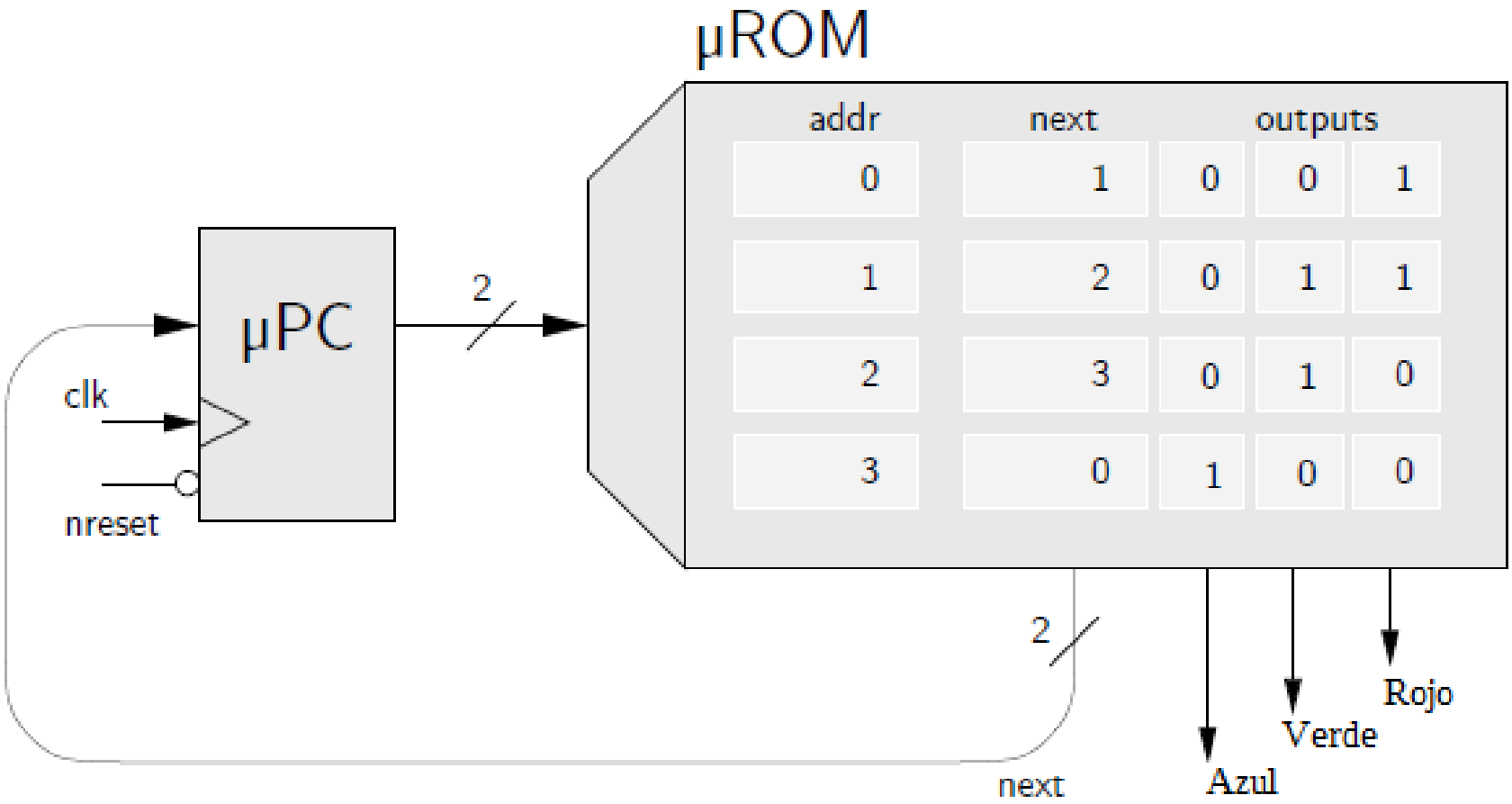


“Paso”	rojo	verde	azul
0	1	0	0
1	1	1	0
2	0	1	0
3	0	0	1

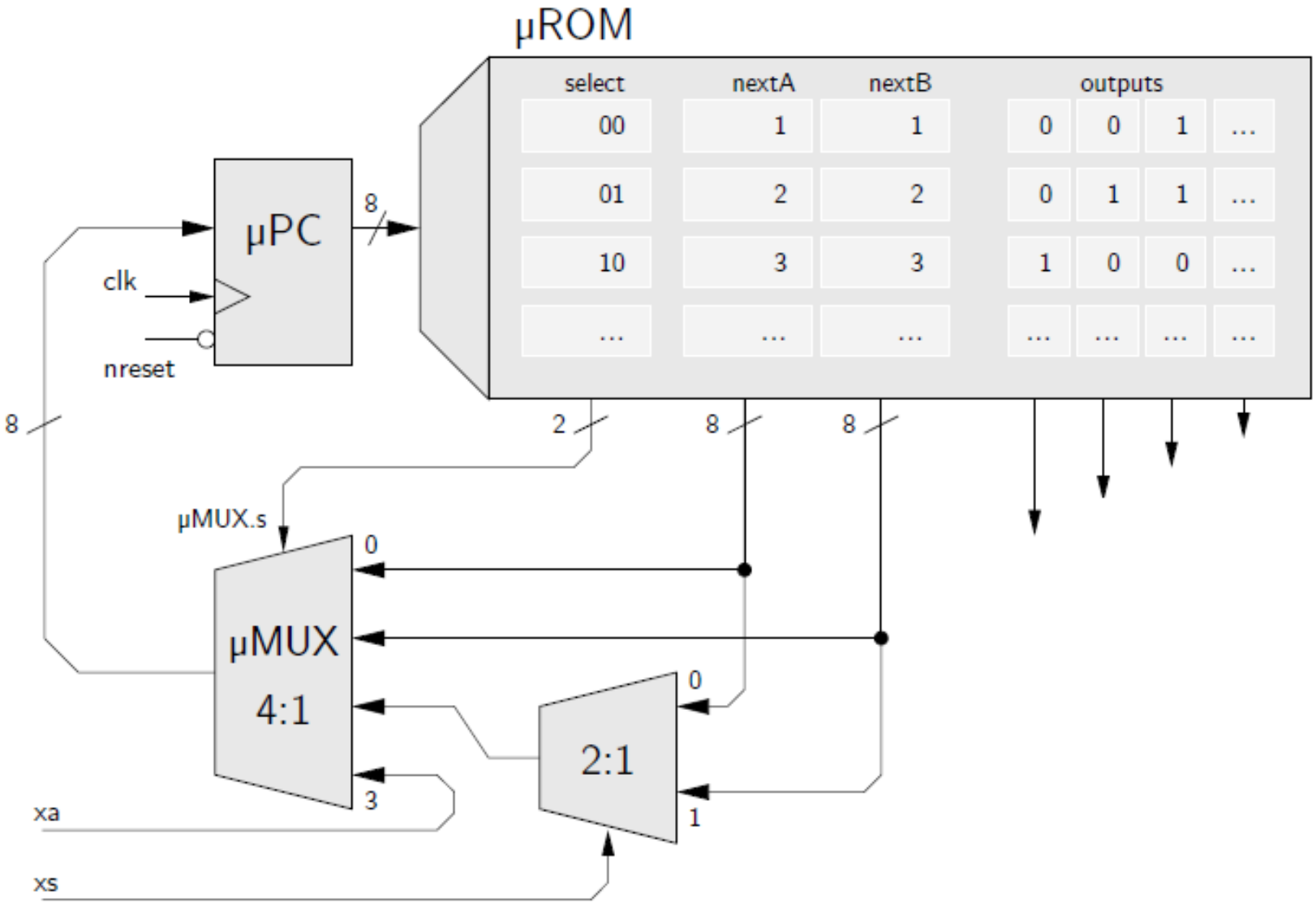
Solución con lógica cableada



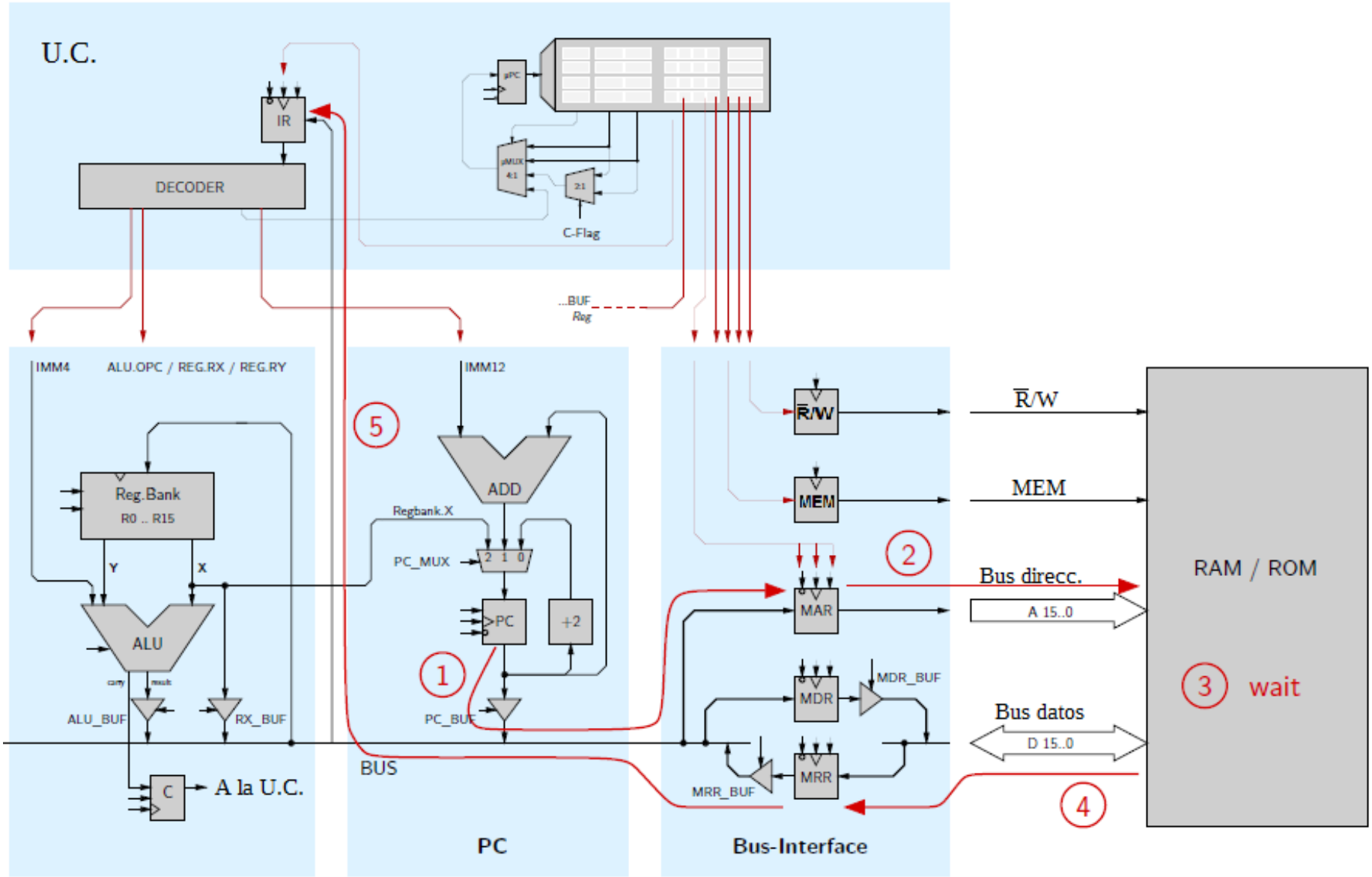
Solución con lógica microprogramada



Lógica microprogramada condicional



Pasos para la lectura de una instrucción



Pipelining

Las CPU internamente están compuestas de varios bloques funcionales.

La ejecución de una instrucción consta de varios estados y en cada estado no están “en uso” todos los bloques.

Si cada estado usa un bloque único, se podrían ejecutar instrucciones en paralelo (en teoría, tantas como estados: cada una estaría en un estado). A esto se denomina *pipelining*.

Puede precisar complicar los bloques (usar más “subbloques”, usar más buses, usar más estados, etc.) para hacerlos independientes.

Ejecución de una instrucción

Va a pasar por varios estados (más en CISC, menos en RISC).

Vamos a suponer que son los siguientes (CISC):

A- recuperar de memoria la instrucción

B- decodificar la instrucción

C- calcular operandos (si se precisa)

D- recuperar operandos de memoria (si se precisa)

E- realizar la operación/acción

F- escribir el resultado en memoria (si se precisa)

Ejemplo ejecución con *pipelining*

	T1	T2	T3	T4	T5	T6	T7	T8	T9
Instr. 1	A	B	C	D	E	F			
Instr. 2		A	B	C	D	E	F		
Instr. 3			A	B	C	D	E	F	
Inst. 4				A	B	C	D	E	F

Tiempo sin *pipelining*: 4 instrucciones . 6 T/instrucción = 24T

Tiempo con *pipelining*: 6T + (4 instrucciones – 1 instrucción)T = 9T

Dificultades en el *pipelining*

Los rendimientos ideales del *pipelining* no se pueden conseguir en la realidad por varias razones:

- Ejecuciones condicionales. Si camino a seguir depende de una condición, este no puede saberse hasta que no se evalúe dicha condición. Una posibilidad es esperar hasta que se sepa, otra tomar siempre uno de los caminos (por ejemplo, que no se cumple) y otra predecir con algún algoritmo (“cableado”) cuál es el camino más probable. En los dos últimos casos, si se falla en la predicción, hay que desechar la parte del pipeline no válida (incluso deshaciendo operaciones).

- Dependencia de datos. Cuando no se puede realizar una instrucción porque depende del resultado de una instrucción previa. Aquí existe de nuevo la posibilidad de esperar hasta que se conozca. Otra posibilidad más compleja es la ejecución fuera de orden, por ejemplo "adelantando" instrucciones que no dependen de un resultado previo, de forma que el pipeline no se pare.
- Interrupciones. Si las interrupciones están activadas, en procesadores sin *pipelining* se comprueba si hay interrupciones al terminar de ejecutar todos los estados de la instrucción en curso mientras que en procesadores con *pipelining*, se comprueban en "cualquier momento". Como la atención a la interrupción precisa la ejecución

de un programa y tenemos el pipeline “lleno”, hay dos posibilidades: desechar lo que está en el pipeline y atender inmediatamente a la interrupción o terminar todas (o algunas) de las instrucciones del pipeline antes de atender a la interrupción. La primera tiene la ventaja de ser más fácil de implementar y ser más rápida atendiendo la interrupción, pero pierde parte del trabajo ya realizado (instrucciones del pipeline).

Referencias:

- D·CORE Prozessors. Universidad de Hamburgo. https://tams.informatik.uni-hamburg.de/lectures/2015ws/praktikum/rechprak/
- “Computer Organization and Architecture”. William Stallings, 2015. Pearson.
- Wikipedia.