

Introducción

Estructura de Computadores
1^a-2^a Semana

Bibliografía:

[TOC] Temas 1-3	Apuntes Tecnología y Organización de Computadores
[HAM03] Cap.1	Organización de Computadores. Hamacher, Vranesic, Zaki. McGraw-Hill 2003 Signatura ESIIT/ C.1 HAM org
[BRY11] Cap.1	Computer Systems: A Programmer's Perspective. Bryant, O'Hallaron. Pearson, 2011 Signatura ESIIT/ C.1 BRY com
[PRI10]	Introducción a la Informática. Prieto, Lloris, Torres. McGraw-Hill Interamericana 2010 Signatura ESIIT/ A.0 PRI int

Guía de trabajo autónomo (4h/s)

■ Repaso

- Apuntes TOC

■ Lectura

- Cap.1 Hamacher
- Cap.1 CS:APP (Bryant/O'Hallaron)
- Guión de la Práctica 1

Bibliografía:

[TOC] Temas 1-3	Apuntes Tecnología y Organización de Computadores
[HAM03] Cap.1	Organización de Computadores. Hamacher, Vranesic, Zaki. McGraw-Hill 2003 Signatura ESIIT/ C.1 HAM org
[BRY11] Cap.1	Computer Systems: A Programmer's Perspective. Bryant, O'Hallaron. Pearson, 2011 Signatura ESIIT/ C.1 BRY com
[PRI10]	Introducción a la Informática. Prieto, Lloris, Torres. McGraw-Hill Interamericana 2010 Signatura ESIIT/A.0 PRI int

TOC

■ Tecnología y Organización de Computadores

■ TEMARIO TEÓRICO:

- 1. Introducción
 - 1.1 Conceptos básicos
 - 1.2 Estructura funcional de un computador
 - 1.3 Niveles conceptuales de descripción de un computador
 - 1.4 Clasificación de computadores
 - 1.5 Parámetros que caracterizan las **prestaciones** de un computador
- 2. Unidades funcionales de un computador
 - 2.1. El procesador
 - 2.2. La memoria
 - 2.3. Periféricos de E/S
 - 2.4. Estructuras básicas de interconexión
- 3. Representación de la información en los computadores
 - 3.1 Representación de textos
 - 3.2 Representación de sonidos
 - 3.3 Representación de imágenes
 - 3.4 Representación de datos numéricos

Vocabulario

■ Arquitectura

- Aspectos necesarios para redactar programa ensamblador correcto
- Incluye: registros CPU, repertorio instrucciones, modos direccionamiento

■ Organización (del computador, de la CPU, de la ALU)

- **Estructura:** componentes y su interconexión (“foto fija”)
- **Funcionamiento:** dinámica procesamiento información

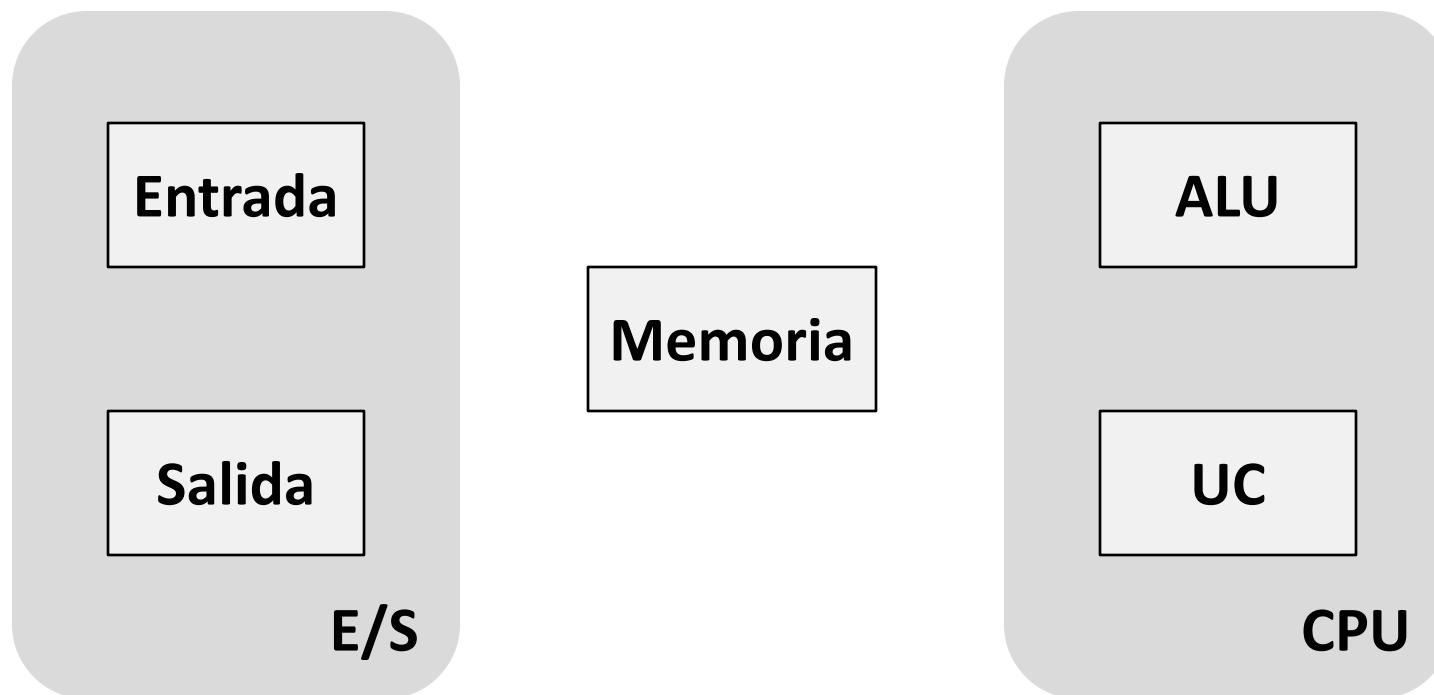
■ Computador (digital): E/S, M, CPU (ALU+UC)

- Computador personal
 - Sobremesa (desktop)
 - Portátil (laptop)
- Estación de trabajo (más prestaciones, gráficos)
- Sistemas de empresa (más CPU y almacenamiento)
- Servidores (bases de datos, gran volumen peticiones)
- Supercomputadores (cálculos científicos)

Introducción

- **Unidades funcionales**
- **Conceptos básicos de funcionamiento**
- **Estructuras de bus**
- **Rendimiento**
- **Perspectiva histórica**

Unidades Funcionales



Unidades Funcionales

■ Arquitectura von Neumann

- Distingue 5 componentes: E/S, M, CPU (ALU+UC)

■ E: codificar / digitalizar / transmitir (lectura)

- teclado, ratón, red, disco, CD...

■ M: almacenar

- programas, datos E, resultados operaciones...

■ CPU: Unidad de procesamiento central

- procesa información E/M ejecutando programa
- ALU: Unidad aritmético-lógica: operaciones
- UC: Unidad de control: controla circuitos

■ S: codificar / almacenar / transmitir (escritura)

- pantalla, impresora, disco, red...

M almacena *instrucciones* y datos

■ Instrucciones máquina

- Transferencia (mov, in, out) M, E/S
- Operaciones (add, and) ALU
- Control (jmp, call, ret, set) UC

■ Concepto de “programa almacenado”

- Determina comportamiento máquina (salvo IRQ)
 - porque instrucciones reproducibles y flujo programa predeterminado

■ Datos

- En memoria, todo son datos
 - interpretado como programa (codop): si leido en etapa captación
 - Compilar, desensamblar: código usado como datos
- Codificación:
 - Instrucciones: codops (codificación en bloque, por extensión, según fabricante)
 - Enteros: binario (complemento a dos), BCD...
 - Alfabéticos: ASCII, EBCDIC...
 - Punto flotante: IEEE-754 simple/doble precisión...

TOC: 1.1 Conceptos básicos. Lenguaje máquina

- El *lenguaje máquina* es el único que entienden los circuitos del computador (CPU). Las instrucciones se forman por bits agrupados en campos:
 - **Campo de código de operación** indica la operación correspondiente a la instrucción.
 - **Campos de dirección** especifican los lugares (o posición) donde se encuentra o donde ubicar los datos con los que se opera.

E/S

■ Entrada:

- codificar información **operador** → M / CPU
 - teclado, ratón/palanca (junto con pantalla), micrófono
- recuperar información **previamente** almacenada
 - HD, CD/DVD, lector tarjetas magnéticas...
- comunicar **ordenadores** entre sí
 - tarjeta de red, módem...

■ Salida:

- codificar información resultado → **operador humano**
 - impresora, pantalla
- almacenar para uso **posterior**
 - HD, CD/DVD...
- comunicar con otros **computadores**
 - red, módem...
- muchos dispositivos son duales E/S (aceptan R/W)

M

■ Memoria:

- Almacenamiento primario (memoria semiconductor)
 - Palabras n bits accesibles en 1 operación básica R/W
 - Longitudes palabra típicas: 16-64bits
 - Muy frecuente: memoria de bytes (asuntos alineamiento, ordenamiento)
 - Accesible aleatoriamente (RAM) por dirección (posición)
 - Bus direcciones, bus datos, bus control (R/W), T_{acceso}
 - Tamaños memoria típicos (PC): 8GB...32GB
 - Tiempos acceso típicos: ~ns (DDR4-2400 19.2GB/s CL15 Lat 12.5ns)
 - » DDR4-2400 → $F_{bus}=1200\text{MHz}$, $T_{cyc}=0.833\text{ns}$, $\text{Lat}_{\text{CAS}} \text{ CL15} \rightarrow 12.5\text{ns}$
 - Jerarquía memoria: cache L1, L2 (on-chip), L3, MP
 - Programa almacenado en MP
- Almacenamiento secundario (óptico/magn. E/S)
 - No es memoria von-Neumann, es E/S
 - Fichero swap se considera como parte de la jerarquía memoria

CPU

■ ALU:

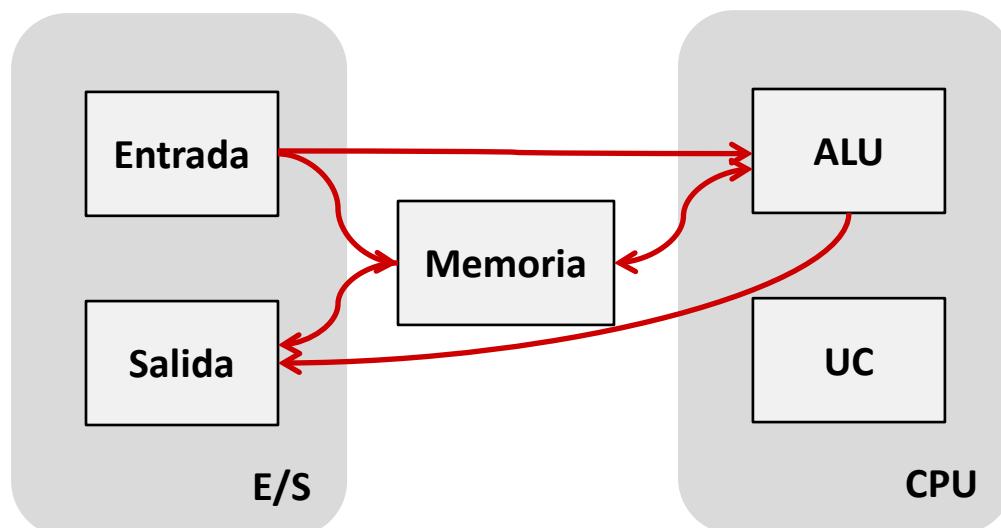
- Componente más rápido del computador (junto con UC)
- **Registros**: almacenamiento más rápido (más que L1)
 - Operandos/Resultado de/a memoria/registros
 - Arquitecturas R/R, R/M, M/M
- Operaciones **aritméticas** (add, mul, div...)
 - Enteras y punto flotante
- Operaciones **lógicas** (and, rol...)
 - Bit a bit

■ UC:

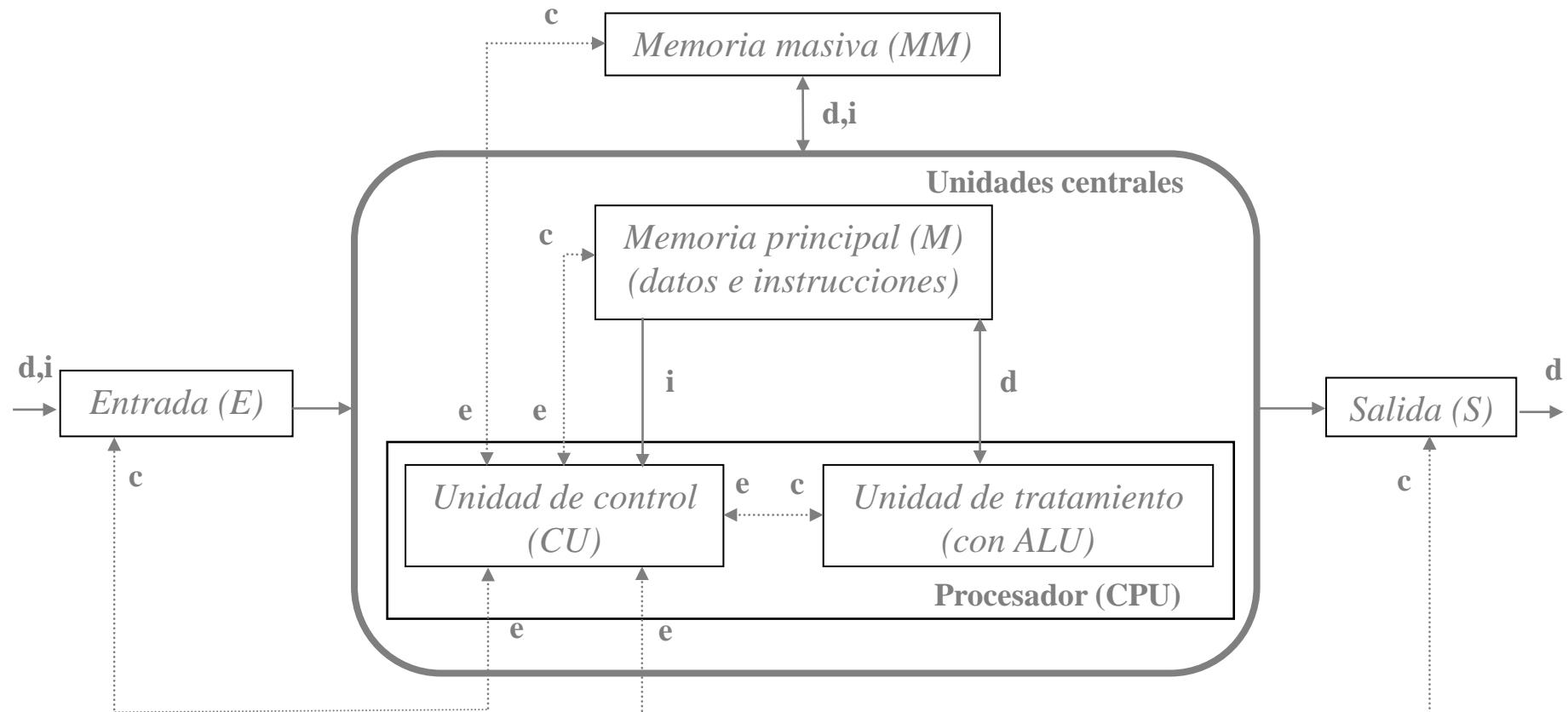
- Componente más rápido del computador (con ALU)
- **Controla** todos los demás circuitos (ALU, M, E/S)
 - Según lo indicado por el programa almacenado en MP
 - Instrucciones transferencia → señales control **M y E/S**
 - Instrucciones aritm/lógicas → señales control **ALU**
 - **Temporización** señales (dirección, datos, R/W)

■ Posibilidades funcionamiento

- Programa E → MP
- Datos E → MP
- Ejecución programa: Datos → ALU → resultados
 - E / M → ALU → S / M
- Resultados → S
- Todo controlado según indique programa MP
 - Interpretado por la UC



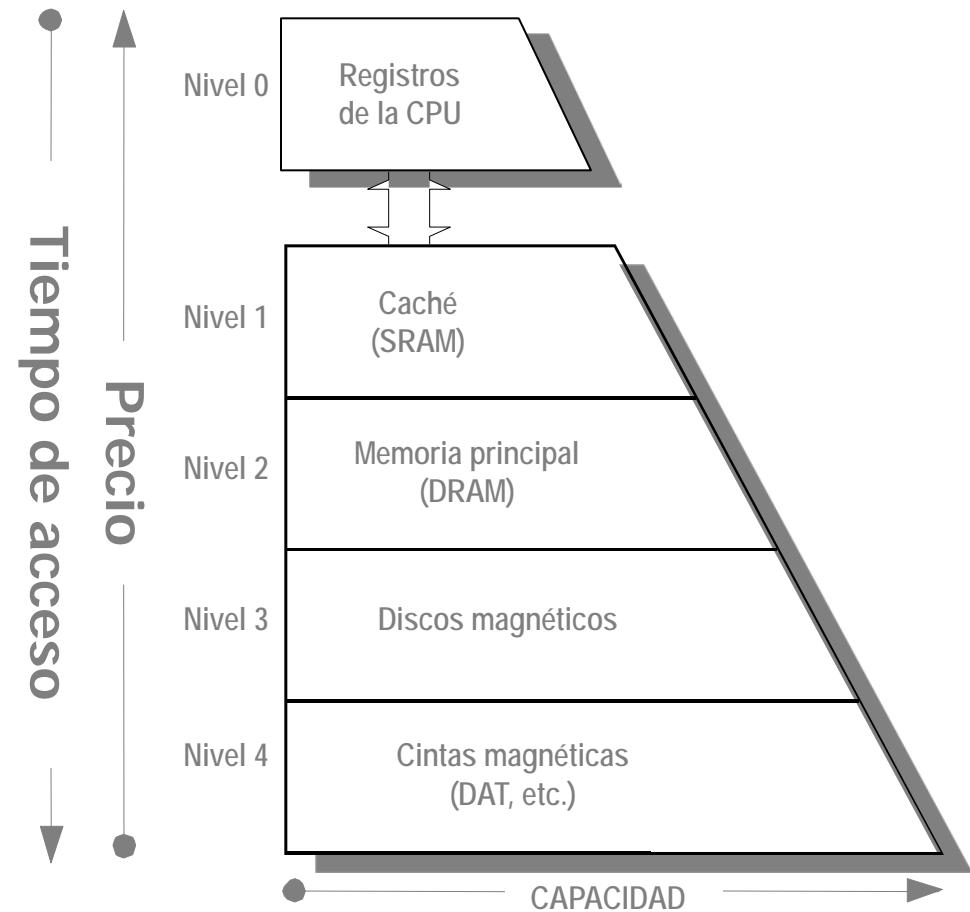
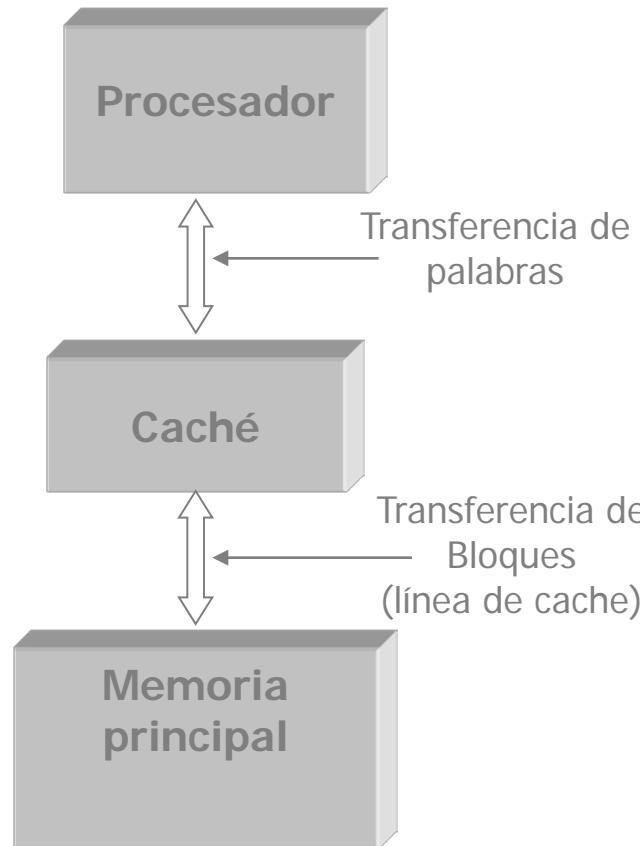
TOC: 2. Unidades funcionales de un computador



d: datos ; *i*: instrucciones

e: señales de estado *c*: señales de control

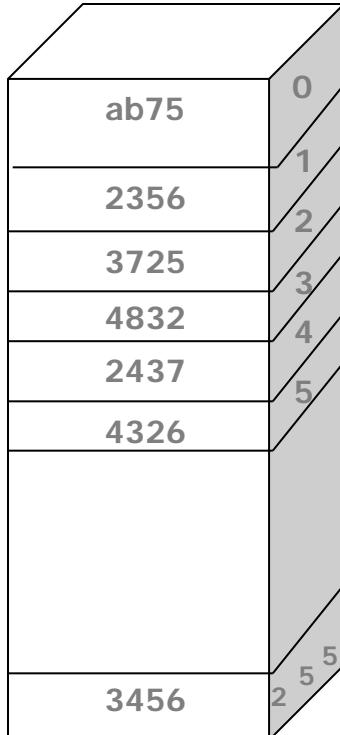
TOC: 2.2 Jerarquía de memoria



sobre Memoria

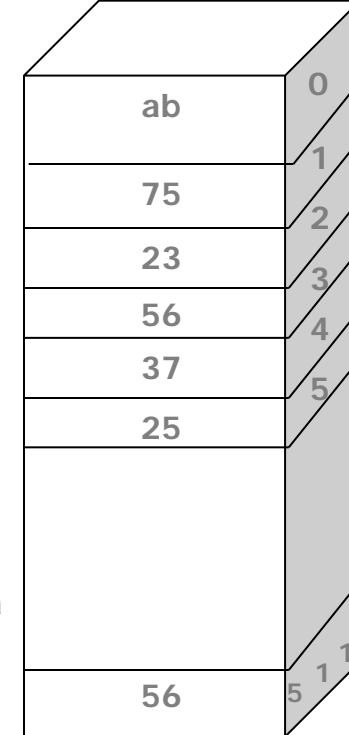
■ Organización en bytes

- ¿tamaño posición M = registro CPU (longitud palabra)?
 - ideal, pero no frecuente
 - típicamente, posiciones 1B (direcciónamiento por bytes)
 - no necesidad empaquetamiento cadenas (strings)
 - Problemas: **alineamiento, ordenamiento**



TOC: hipotética memoria
256 palabras de 16bits
(apuntes TOC §1.2)

misma cantidad de memoria
organizada como 512 bytes
words alineadas, big-endian



sobre Memoria de bytes

■ Ordenamiento en memoria de bytes

- Criterio del extremo menor (**little-endian**)
 - Primero se almacena el byte menos significativo (LSB)
 - LSB en posición M más baja, MSB en posición más alta
- Criterio del extremo mayor (**big-endian**)
 - Primero el MSB (en posición M más baja)

Dirección Contenido

0	ab
1	75
2	23
3	56
4	37
5	25
.	⋮
.	⋮
1FF	56

mismo contenido
en big-endian

Dirección Contenido

0	75
1	ab
2	56
3	23
4	25
5	37
.	⋮
.	⋮
1FF	34

mismo contenido
en little-endian

sobre Memoria de bytes

■ Alineamiento en memoria de bytes

- Palabra de n bytes alineada \Leftrightarrow comienza en dirección múltiplo de n
 - Algunas CPUs requieren alineamiento accesos M (si no, bus error)
 - Otras acceden más rápido si acceso alineado
 - palabra no cruza línea de cache, página, etc

Dirección	Contenido
0	01
1	00
2	00
3	00
4	FE
5	FF
6	FF
7	FF
.	.
.	.
.	.

palabra 16bits
valor 1, alineada

pal. 32bits, v
255, no alineada

palabra 32bits
valor -2, alineada

byte suelt
valor -1

(máquina little-endian)

Dirección	Contenido
0	AB
1	75
2	FF
3	00
4	00
5	00
6	48
7	FF
.	.
.	.
.	.

Clasificaciones m/n y pila-acumulador-RPG

■ Tipos de CPU según operandos de las instrucciones ALU

- también suele afectar a operandos instrucciones transferencia

■ Clasificación m/n

- Operaciones ALU admiten n operandos, m de ellos de memoria

■ Combinaciones típicas

- Máquinas **pila**: 0/0

- Repertorio: Push M, Pop M, Add, And...

- Máquinas de **acumulador**: 1/1

- Operando implícito: registro acumulador A (más rápido que M)

- Repertorio: Load M, Store M, Add M, And M...

- Máquinas de **RPG** (Registros de Propósito General): (x/2, x/3)

- Múltiples “acumuladores”

- Repertorio: Move R/M R/M, Add R/M R/M R/M

RPG: Clasificación R/M

■ Para máquinas RPG

- Arquitecturas **R/R** (registro-registro)
 - 0/2, 0/3
 - Add R1, R2, R3
 - típico de RISC
- Arquitecturas **R/M** (registro-memoria)
 - 1/2, 1/3 (2/3 poco frecuente)
 - Add R1, A
 - típico de CISC
- Arquitecturas **M/M** (memoria-memoria)
 - 2/2, 3/3 (poco frecuente)
 - Add A, B
 - permite operar directamente en memoria
 - demasiados accesos memoria por instrucción máquina

sobre Repertorios

■ ISA

- Arquitectura del Repertorio (Instruction Set Architecture)
- Registros, Instrucciones, Modos de direccionamiento...

■ RISC

- Comput. repertorio reducido (Reduced Instruction Set Computer)
- 0/2, 0/3
- Pocas instrucciones, pocos modos, formato instrucción sencillo
- UC sencilla → muchos registros

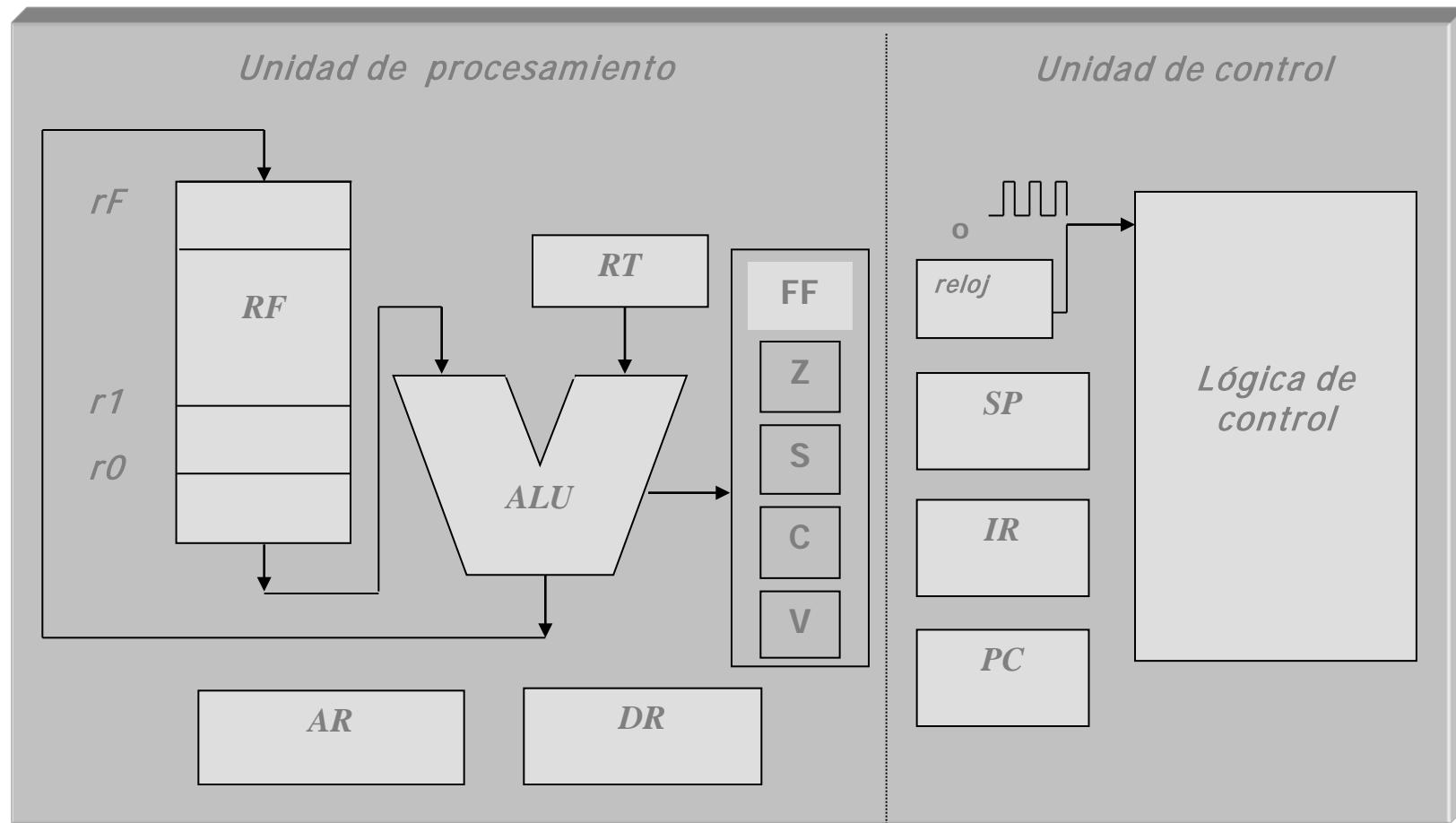
■ CISC

- Comput. repertorio complejo (Complex Instruction Set Computer)
- 1/2, 1/3 (y resto)
- “más próximos a lenguajes alto nivel”
- Debate RISC/CISC agotado, diseños actuales mixtos

Introducción

- Unidades funcionales
- **Conceptos básicos de funcionamiento**
- Estructuras de bus
- Rendimiento
- Perspectiva histórica

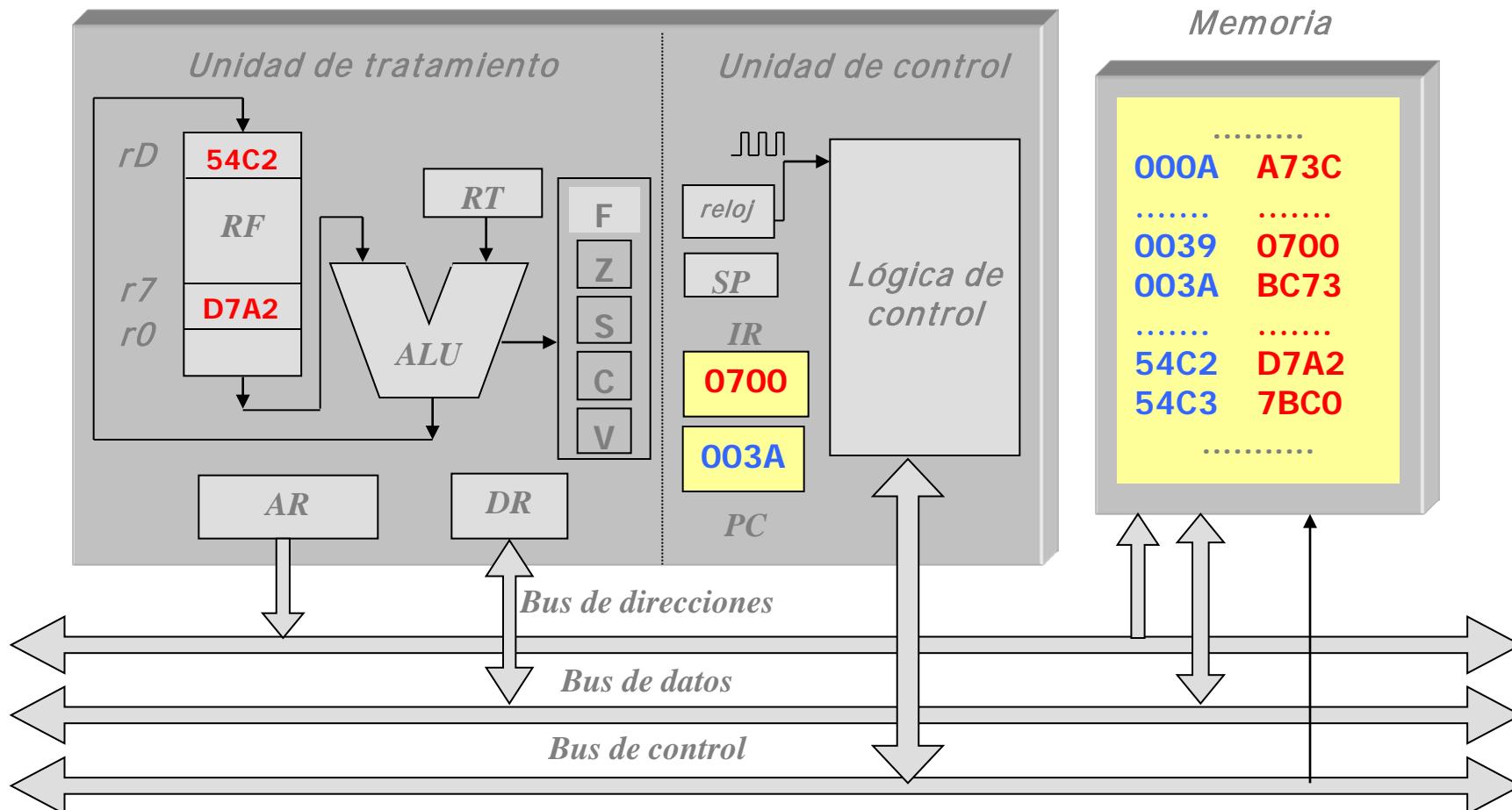
TOC: 2.1 Elementos internos de un procesador



TOC: 2.1 Ejecución de Mov (rD), r7

Direc.	Contenidos	Fase	Microoperación	Contenidos de los registros					
				PC	IR	AR	DR	r7	
0000	7AC4	Valores iniciales							
0007	65C9	Captación de instrucción	AR ← PC	0039		0039			
	0700		DR ← M(AR)	0039		0039	0700		
	607D		IR ← DR	0039	0700	0039	0700		
	2D07		PC ← PC+1	003A	0700	0039	0700		
	C000		AR ← rD	003A	0700	54C2	0700		
54C2	D7A2	Ejecución de instrucción	DR ← M(AR)	003A	0700	54C2	D7A2		
	3FC4		r7 ← DR	003A	0700	54C2	D7A2	D7A2	
rD	54C2								

TOC: 2.1 Situación después de la ejecución



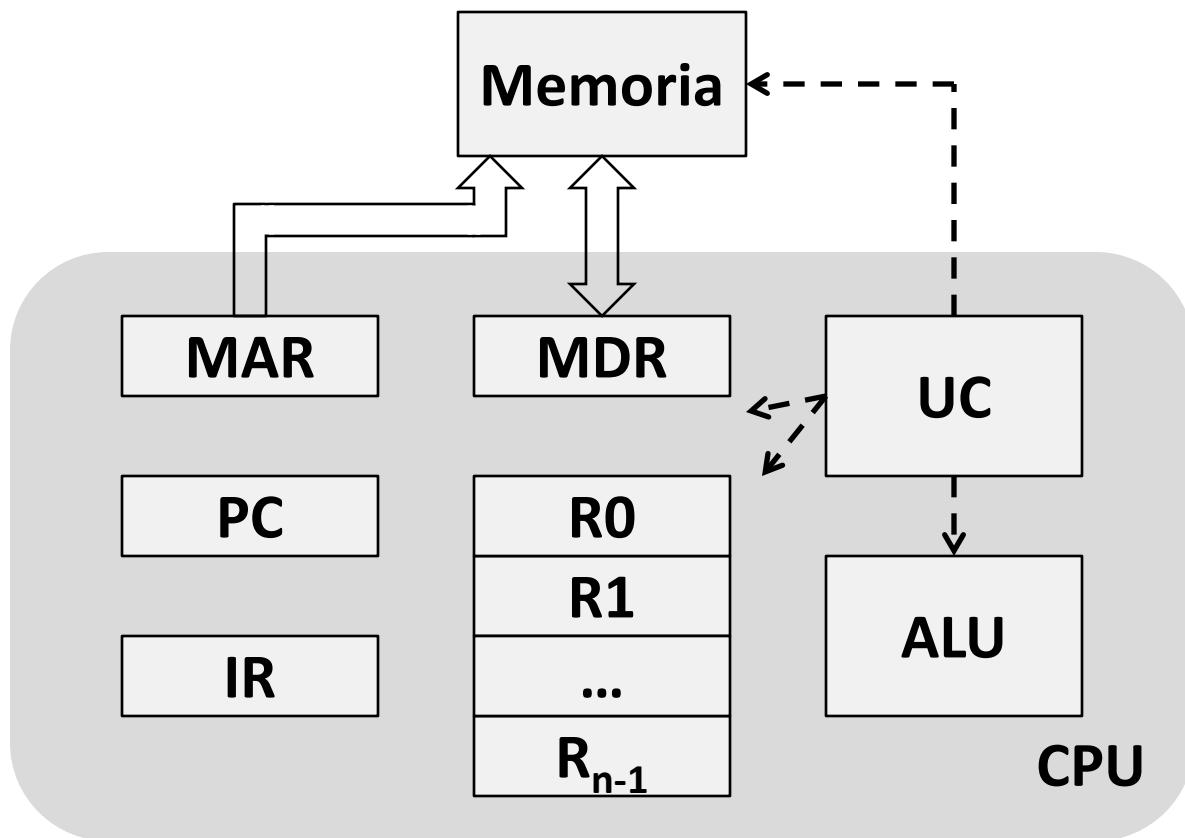
Ciclo ejecución instrucciones: fases

- Programa en MP
- CPU (UC) tiene PC (*program counter*)
 - posición MP de la siguiente instrucción
 - *Captación*: leer dicha posición $IR \leftarrow M[PC]$
 - Usando MAR/MDR (Memory Address/Data Register), Instruction Register (IR)
 - Se interpreta como codop
 - Incrementar PC
 - *Decodificación*: desglosar codop/operandos(regs)
 - Posible etapa *Operando(M)*: captar dato/ incrementar PC
 - *Ejecución*: llevar datos ALU / operar
 - *Almacenamiento*: salvar resultado regs / MP
 - Nombres en inglés:
 - Fetch, Decode, Operand, eXecute, Write/Store

Ciclo ejecución instrucciones

■ Pensar tareas realizadas por UC para ejecutar instrucción

- Por ejemplo: Add A, R0
 - $M[A] + R0 \rightarrow R0$
- Detalles en [HAM03] Cap-1.3
- Ejercicios similares en TOC §2.1



Add A, R0

- $M[POS_A] + R0 \rightarrow R0$
 - Ensamblador traduce p.ej: $POS_A = 100$
 - Valor anterior R0 perdido, el de POS_A se conserva
 - Arquitectura R/M
- **Pasos básicos de la UC**
 - PC apunta a posición donde se almacena instrucción
 - **Captación:** $MAR \leftarrow PC$, Read, $PC \leftarrow PC + 1$, $T_{acc} \leftarrow$ MDR \leftarrow bus, $IR \leftarrow MDR$
 - **Decodificación:** se separan campos instrucción
 - Codop: ADD $mem + reg \rightarrow reg$
 - Dato1: 100 direcciónamiento directo, habrá que leer $M[100]$
 - Dato2: 0 direcciónamiento registro, habrá que llevar R0 a ALU
CPUs con longitud instrucción variable – dirección (100) en siguiente palabra
 - **Operando:** $MAR \leftarrow 100$, Read, $T_{acc} \leftarrow$ MDR, $ALU_{in1} \leftarrow MDR$
 - **Ejecución:** $ALU_{in2} \leftarrow R0$, add, T_{alu}
 - **Almacenamiento:** $R0 \leftarrow ALU_{out}$

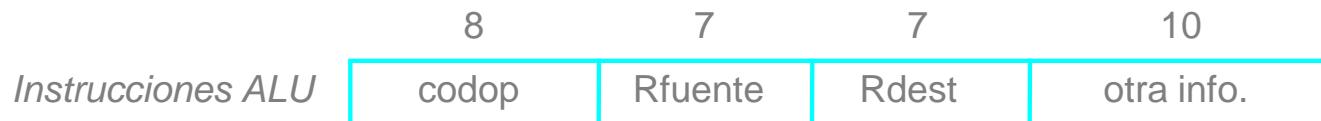
Otras consideraciones

- Arquitectura M/M: **varias captaciones** operando
 - PC++, si las direcciones ocupan más posiciones M
- Arquitectura R/M: cuando **resultado en M** (Add R0, A):
 - acceso memoria adicional (Write): $M[\text{MAR}] \leftarrow \text{MDR} \leftarrow \text{ALU}_{\text{out}}$
 - UC activa señal Write
- Arquitectura R/R: **varias instrucciones** (Load A, R1 / Add R1, R0)
 - efecto colateral: R1 perdido
 - ventaja: CPU más simple, veloz, pequeña (longitud/formato instrucción)
- Ciclo interrumpido por **IRQ**→ISR
 - mecanismo subrutina / salvar contexto (PC/estado)
 - salvo eso, comportamiento totalmente predeterminado por programa
- CPU completa (+L1+L2...+L3) en 1 chip VLSI
 - La CPU nunca lee de memoria un dato aislado
 - **lee de cache**
 - si hay fallo, se trae un bloque entero
 - se explica en clase así por motivos académicos

sobre Formatos de Instrucción

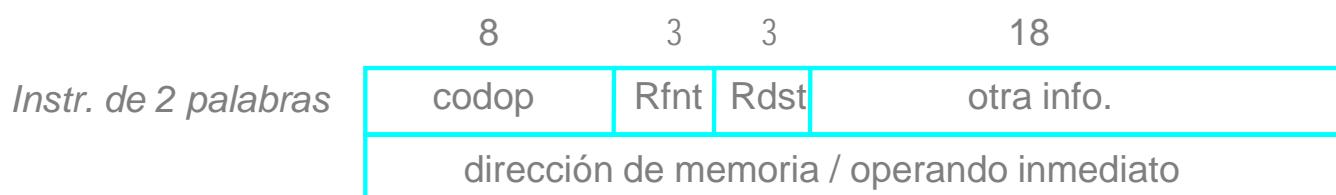
■ RISC

- Pocas instrucciones, pocos modos, muchos registros, 0/2-0/3
 - formato instrucción sencillo, tal vez sólo 2-3: transferencia, ALU, ctrl
 - ej: formato instrucciones ALU de un RISC 32bits 128regs tipo 0/2



■ CISC

- Muchas instrucciones y modos, menos registros, 1/2-1/3 (y resto)
 - varios formatos de instrucción, distintas longitudes, codops long. var. también



Formatos de Instrucción

■ ejemplo: IA-32 (Intel 64)

- Instrucciones de longitud variable, 1-15 bytes (memoria de bytes)
 - prefijos modificar detalles de algunas instrucciones
 - **codop** de 4bits a 3B + 3bits (campo reg en ModRM)
 - Mod-R/M modo de direccionamiento (5 bits)
 - Reg para indicar registro (hasta 8 regs)
 - SIB para indicar 2 registros y escala índice (x1,x2,x4,x8)
 - desplazamiento 32bits dirección memoria (u offset)
 - inmediato 32bits valor operando

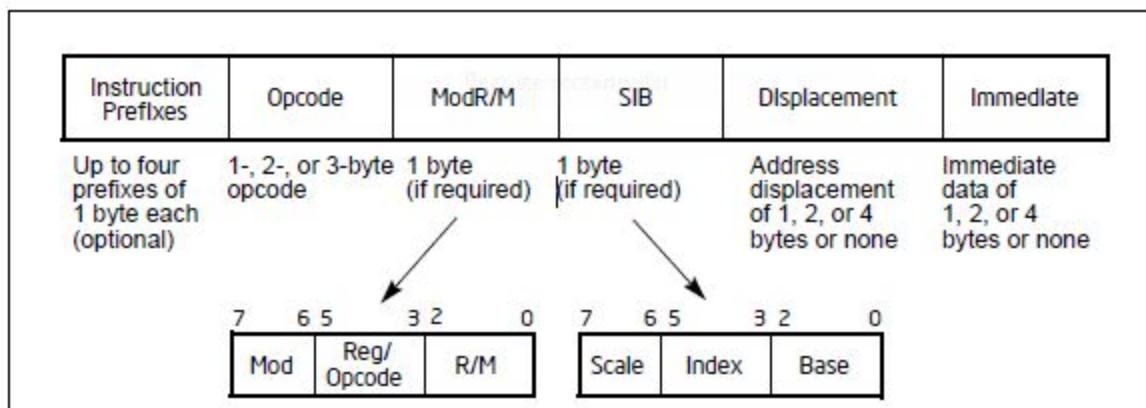


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

Modos de Direcccionamiento

- un número acompañando a un codop puede significar muchas cosas
 - según el formato de instrucción, la instrucción concreta, etc
- cada operando de la instrucción tiene su modo de direccionamiento

■ Inmediato (ej: \$0, \$variable)

- El número es el valor del operando

■ Registro (ej: %eax, %ebx...)

- El número es un índice de registro (ese registro es el operando)

■ Memoria (en general: disp(%base,%index,scale))

- instrucción lleva índices de registros y/o desplazamiento (dirección memoria)
- La dirección efectiva (EA) es la suma de todos ellos. El operando es M[EA].

■ Directo	sólo dirección (disp)	op=M[disp]
■ Indirecto a través reg.	sólo registro (reg)	op=M[reg]
■ Relativo a base	registro y desplazamiento	op=M[reg+disp]
■ Indexado	índice (x escala) y dirección	op=M[disp + index*scale]
■ Combinado	todo	op=M[disp+base+idx*sc]

ej: modos IA-32

a veces puede ser ventajoso
+ instrucciones -tamaño

inmediato ≠ directo

resto modos indirectos

Código fuente ASM:

```
.section .text
_start: .global _start

mov    $0, %eax      # inm - registro
xor    %ebx, %ebx    # reg - registro
inc    %ebx          # reg
mov    $array, %ecx  # inmediato - reg
mov    array, %edx   # directo - reg

mov    (%ecx)        , %edx # indirecto
add    (%ecx,%ebx,4), %edx # combinado
add    array( ,%ebx,4), %edx # indexado
mov    -8(%ebp)      , %edx # rel.base
```

Desensamblado del ejecutable:

Disassembly of section .text:

08048074 <_start>:

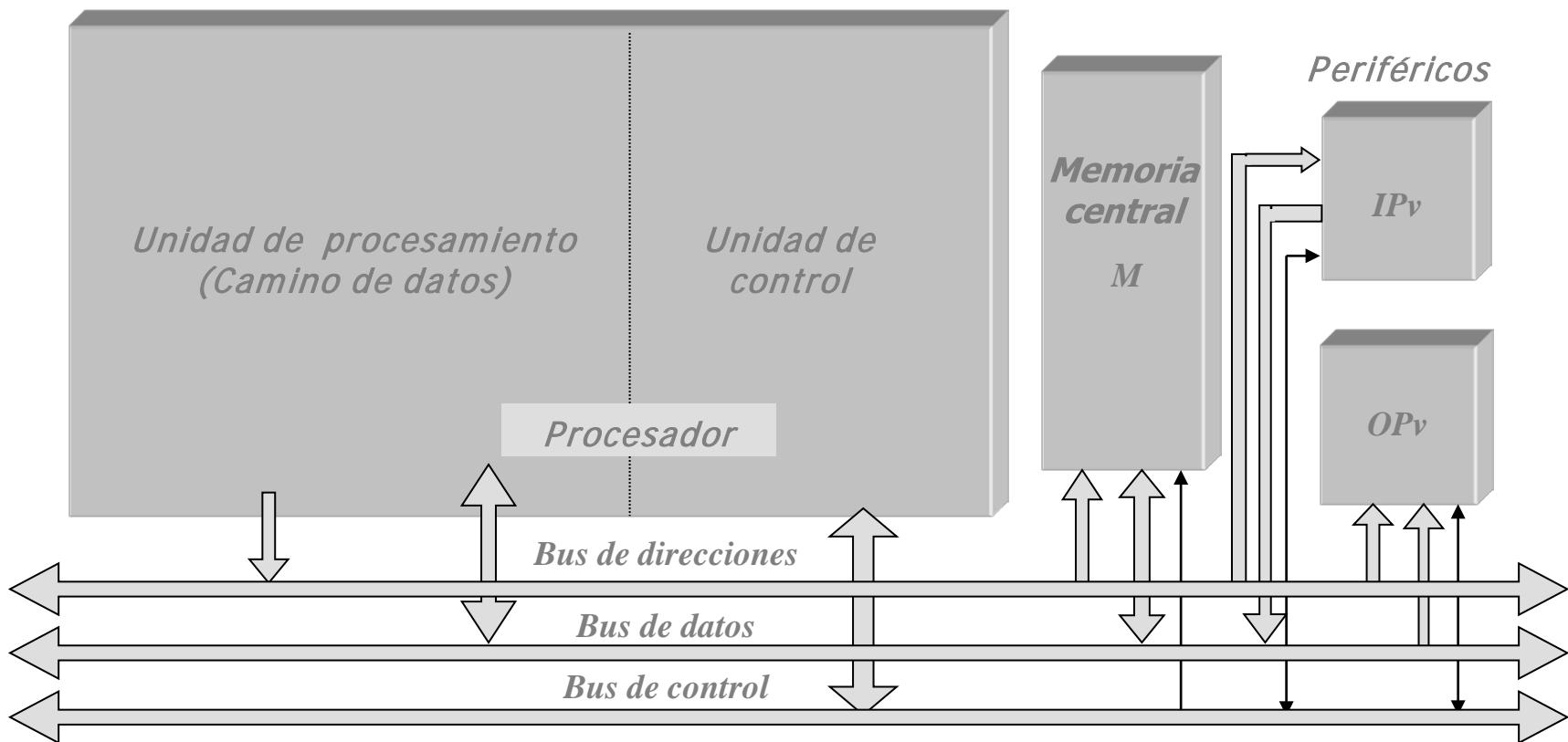
8048074:	b8	00 00 00 00
8048079:	31	db
804807b:	43	
804807c:	b9	98 90 04 08
8048081:	8b	15 98 90 04 08
8048087:	8b	11
8048089:	03	14 99
804808c:	03	14 9d 98 90 04 08
8048093:	8b	55 f8

mov	\$0x0,%eax
xor	%ebx,%ebx
inc	%ebx
mov	\$0x8049098,%ecx
mov	0x8049098,%edx
mov	(%ecx),%edx
add	(%ecx,%ebx,4),%edx
add	0x8049098(%ebx,4),%edx
mov	-0x8(%ebp),%edx

Introducción

- **Unidades funcionales**
- **Conceptos básicos de funcionamiento**
- **Estructuras de bus**
- **Rendimiento**
- **Perspectiva histórica**

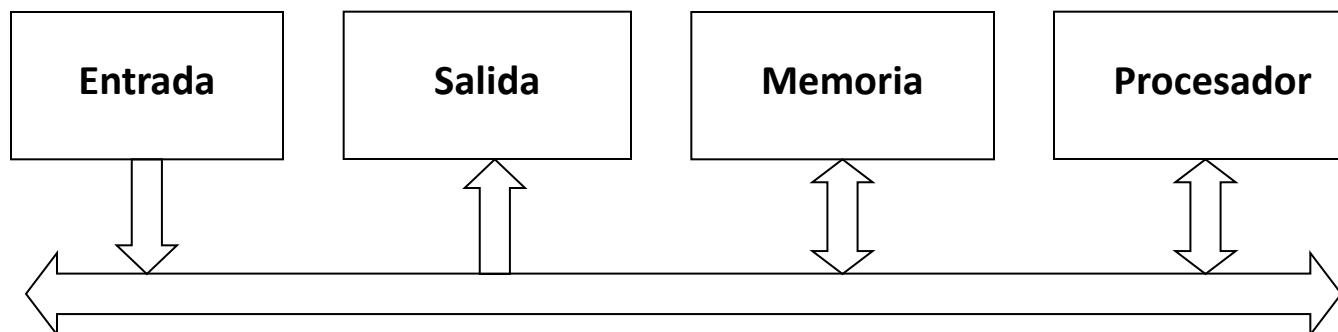
TOC: 2.1 Interconexión de las distintas unidades



Estructuras de bus

■ Justificación buses (paralelos):

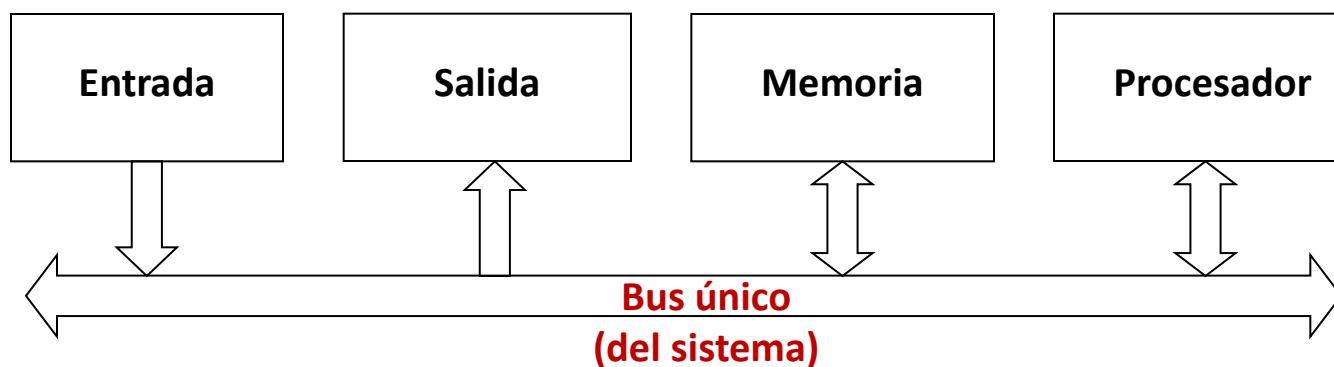
- E/S, M, CPU deben conectarse para pasar datos
- Representación binaria / velocidad transferencia
 - palabras n bits M/ALU → bus **datos** n bits
 - direcciones m bits M → bus **addr** m bits
 - bus **control** para líneas UC (R/W, etc)



Estructuras de bus

■ Bus único

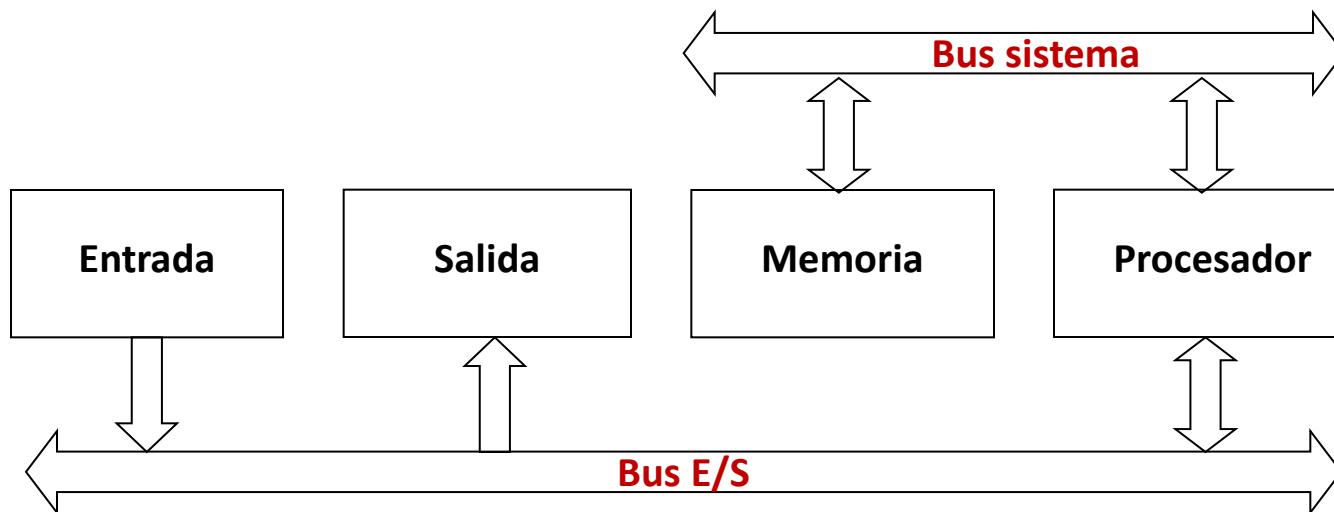
- CPU escribe bus dirección y control R/W
 - también escribe bus datos si Write
 - puede haber señales IOR/W separadas de MemR/W
- E/S/M comprueban si es su dirección
 - sólo en ese caso se conectan al bus de datos
 - evitar cortocircuito bus datos
- Ventaja: sencillez, bajo coste, flexibilidad conexión
 - fácil añadir más dispositivos
 - posibilidad líneas control arbitraje para varios master



Estructuras de bus

■ Buses múltiples

- típicamente: bus sistema (CPU-M) y bus E/S
 - también: múltiples buses E/S
 - separar dispositivos según velocidades
 - incluso: doble bus sistema
 - memoria datos/programa (arquitectura Harvard)
- Ventajas: uno más rápido, ambos funcionan en paralelo
- Inconveniente: coste, complejidad



Adaptación de velocidades

■ Velocidad componentes

- CPU > Memoria >> E/S

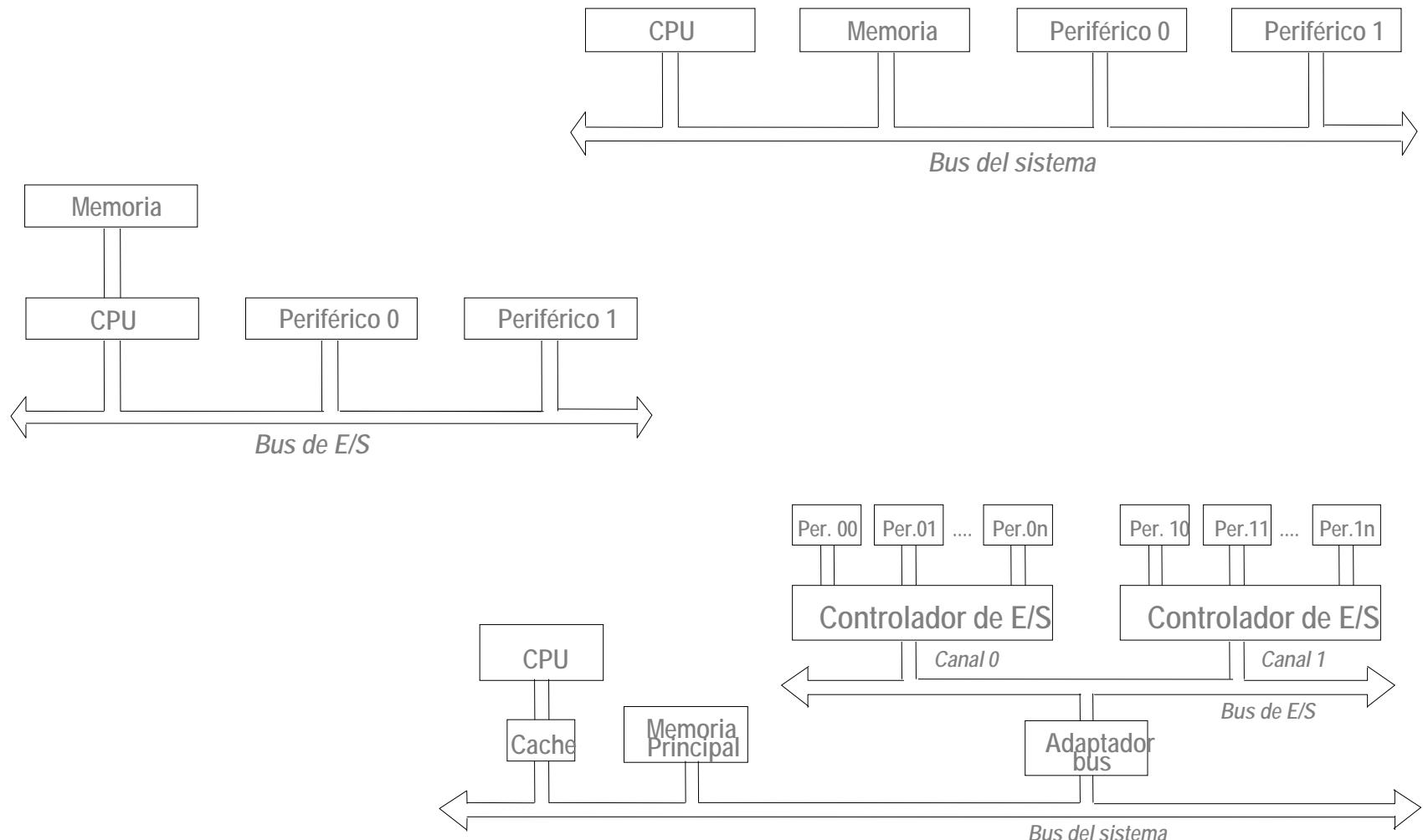
■ Estados de espera:

- alargar ciclo bus si no se activa señal RDY (bus control)
- permite conectar periféricos lentos a bus único

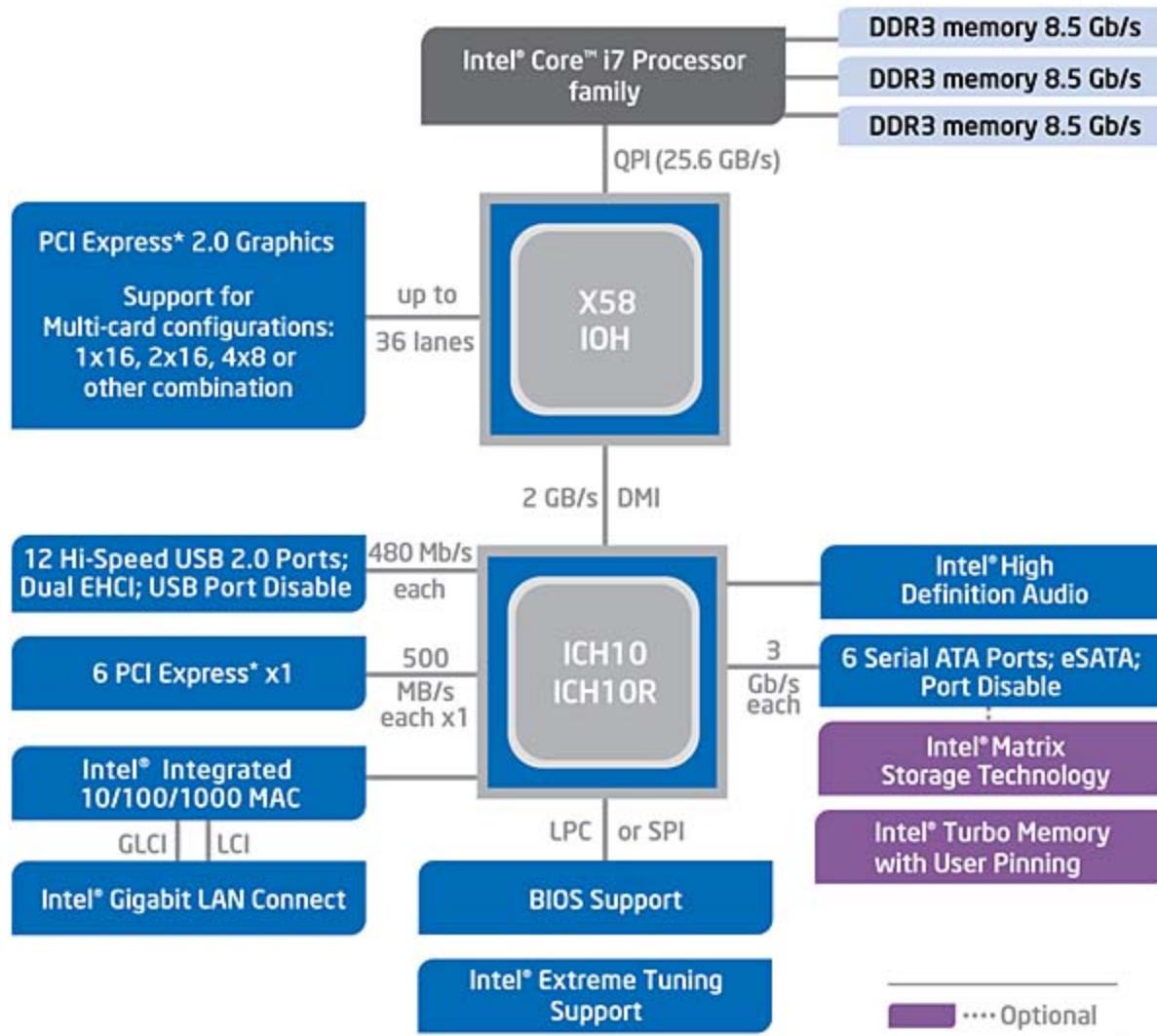
■ Buffers/IRQ:

- dispositivo lento almacena datos en buffer rápido
 - evita retrasar CPU con estados de espera
 - CPU se dedica a otra tarea mientras tanto
- transferencia CPU a velocidad buffer (normal Memoria)
- Write: CPU escribe buffer, dispositivo genera IRQ al final
 - Ej: impresora
- Read: CPU encarga lectura, dispositivo hace IRQ cuando listo
 - Ej: escáner

TOC: 2.4 Estructuras básicas de interconexión



TOC: 2.4 Estructuras básicas de interconexión



Decodificación

■ Evitar cortocircuito bus datos

- Suponer por ejemplo que CPU es único dispositivo **activo** del bus
 - es decir, que puede escribir bus Addr. y Ctrl.
 - cuando lo hace, se convierte en **maestro** del bus
 - Luego veremos multiprocesadores, controladores DMA, etc
 - varios activos requiere arbitraje para escoger maestro
- demás dispositivos **pasivos**
 - sólo “escuchan” bus Addr, no pueden escribir, sólo leer
 - Cuando la CPU les habla, se convierten en **esclavos**
 - Es decir, se conectan al bus de datos y obedecen la orden R/W
- #bits bus Addr. determina el “**espacio de memoria**”

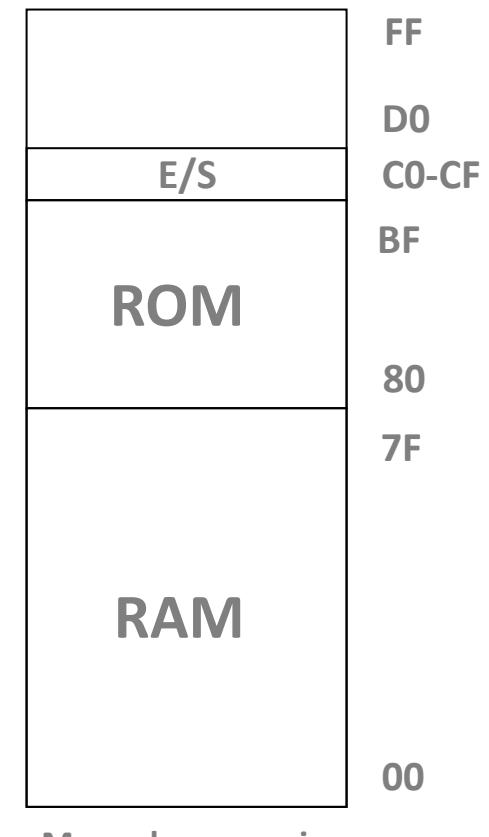
■ Mapa de Memoria

- Dibujo de dónde está cada dispositivo en espacio Memoria
- E/S puede ser “mapeada a memoria” o en espacio E/S separado

Decodificación

■ Ej: diseñar mapa memoria para

- 1 CPU 8bits
 - 8bits Addr. A7...A0
 - 8bits Data: D7...D0
- 1 RAM 128 Bytes
 - decodificada en 0...127
- 1 ROM 64 Bytes
 - a continuación
- 1 Puerto Serie 16 Regs
 - 16 puertos de 8 bits
 - a continuación

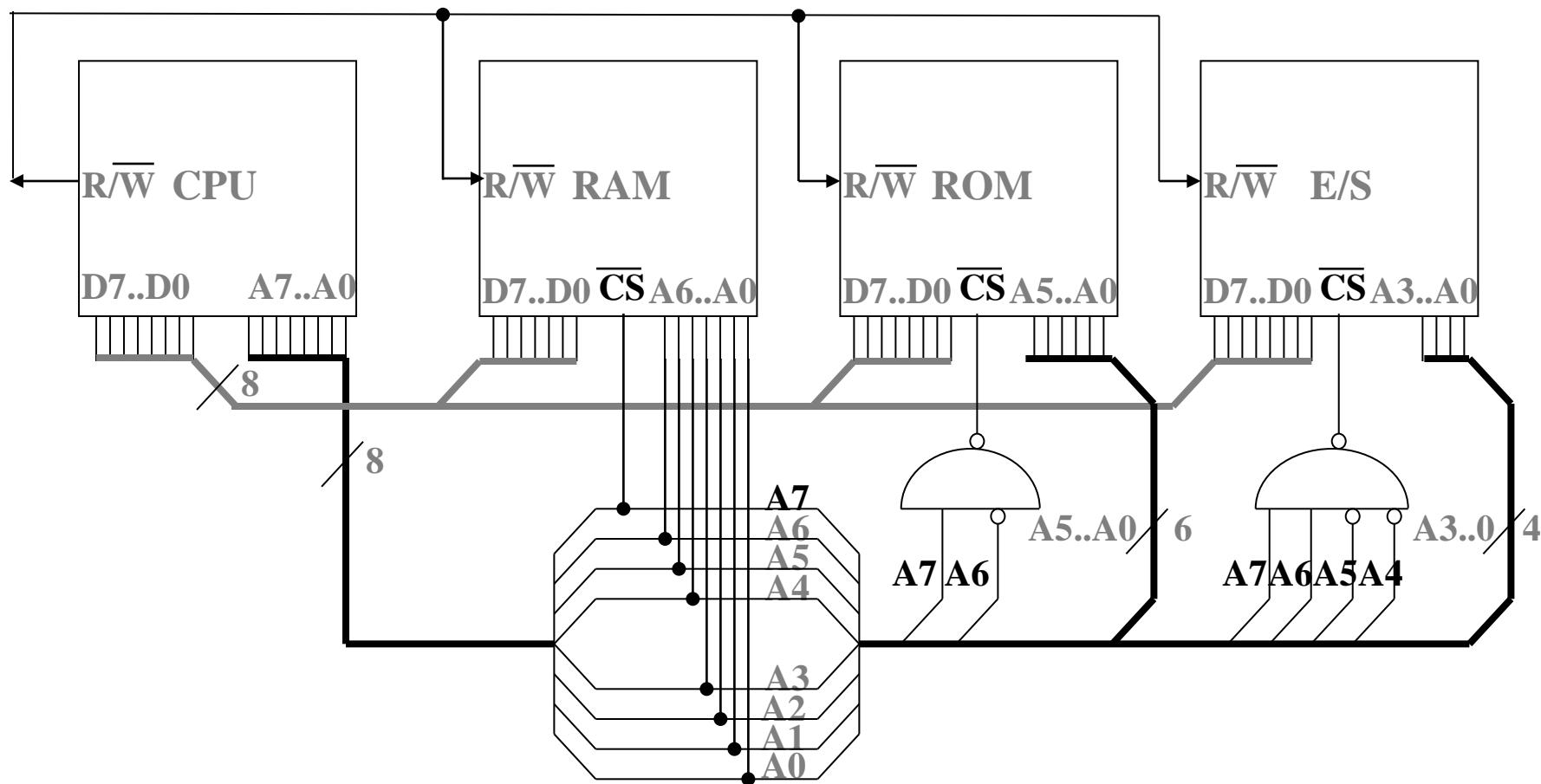
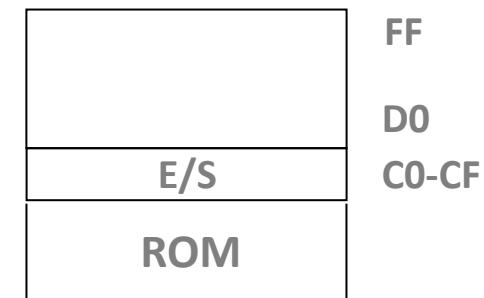


Mapa de memoria

(ejemplo académico, realmente no existen tamaños tan pequeños)

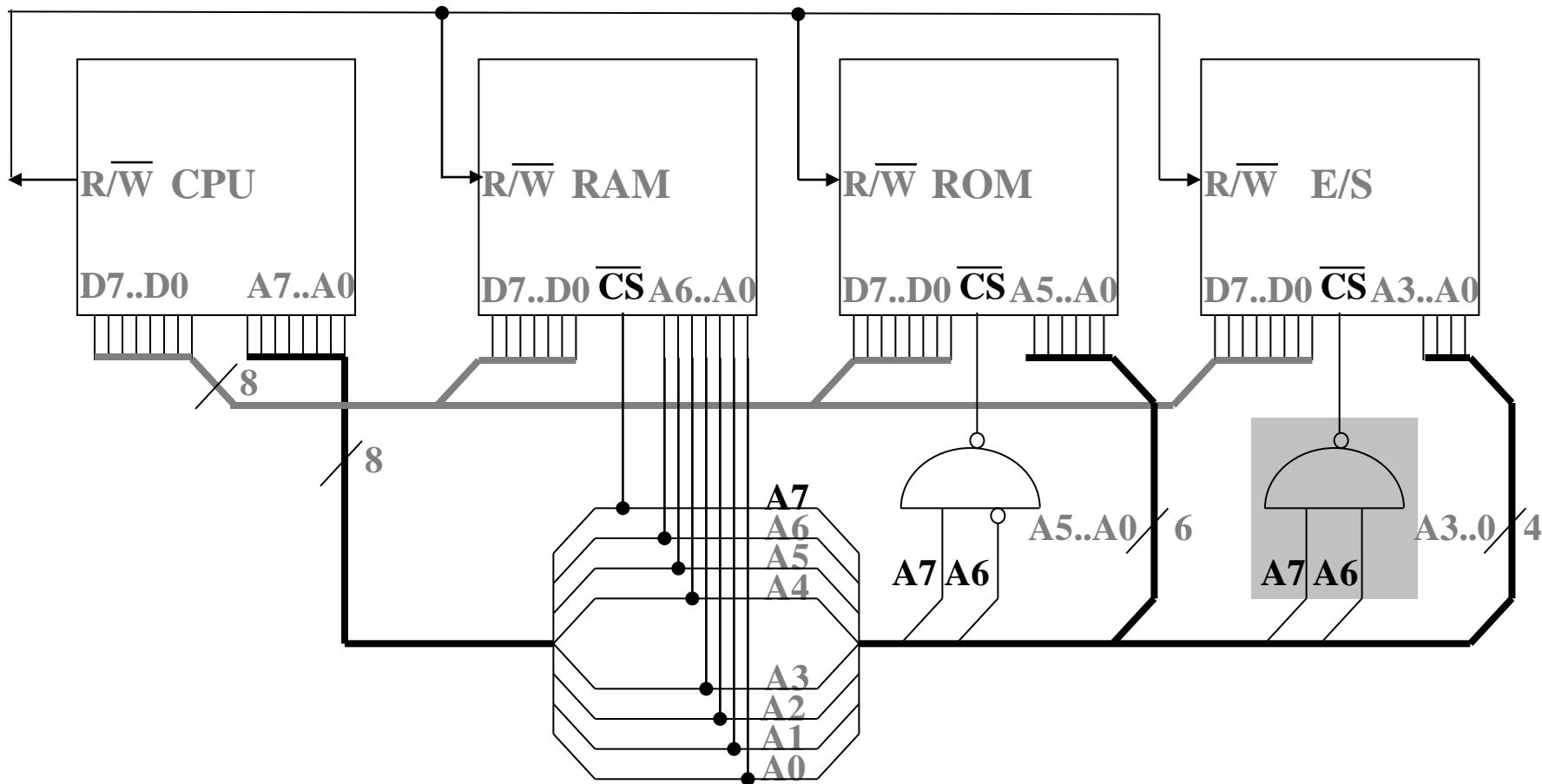
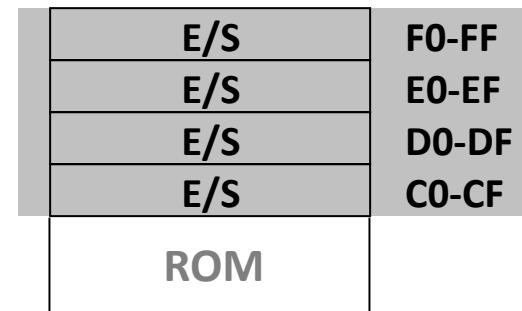
Decodificación completa

- se usan **todos los bits** Addr.
 - MSB decodifican el dispositivo/módulo (CS)
 - LSB direccionan dentro del dispositivo (Addr)



Decodificación parcial

- algunos (m) bits Addr. sin usar
 - El dispositivo aparece repetido 2^m veces en Memoria



Software de sistema

■ Cómo conseguir crear programa → MP → ejecutar

- Software de sistema implicado:
 - Shell (intérprete comandos): recibe órdenes usuario
 - EXEC: llamada para cargar y ejecutar aplicación
 - Editor: permite crear código fuente (y archivar!)
 - Compilador / Enlazador: código objeto / ejecutable
 - Sistema de ficheros (crear, copiar, abrir, leer)
 - desde Shell / desde programa usuario
 - Sistema E/S

■ Cómo se consigue encender → arrancar SO

- soporte hardware: dirección de bootstrap
- [Boot-P]ROM en espacio memoria apuntado
- Bootloader primario, carga arranque HD/FD/CD...
- Bootloader secundario (menú escoger SO, etc)...

Software de sistema

■ Llamadas al sistema (ej: aplicación lee fichero/calcula/imprime)

- usuario teclea nombre aplicación → **EXEC**
 - el shell invoca EXEC, proporcionando nombre fich.
- EXEC carga aplicación HD → M, pasa control
 - el propio SO proporciona zona M cargar aplicación
 - EXEC retorna a aplicación, y ella retornará a shell
- aplicación invoca **OPEN/READ/CLOSE**
 - proporciona zona memoria donde leer contenido
- aplicación calcula resultado, invoca **PRINT/EXIT**
 - proporciona datos a imprimir / código retorno

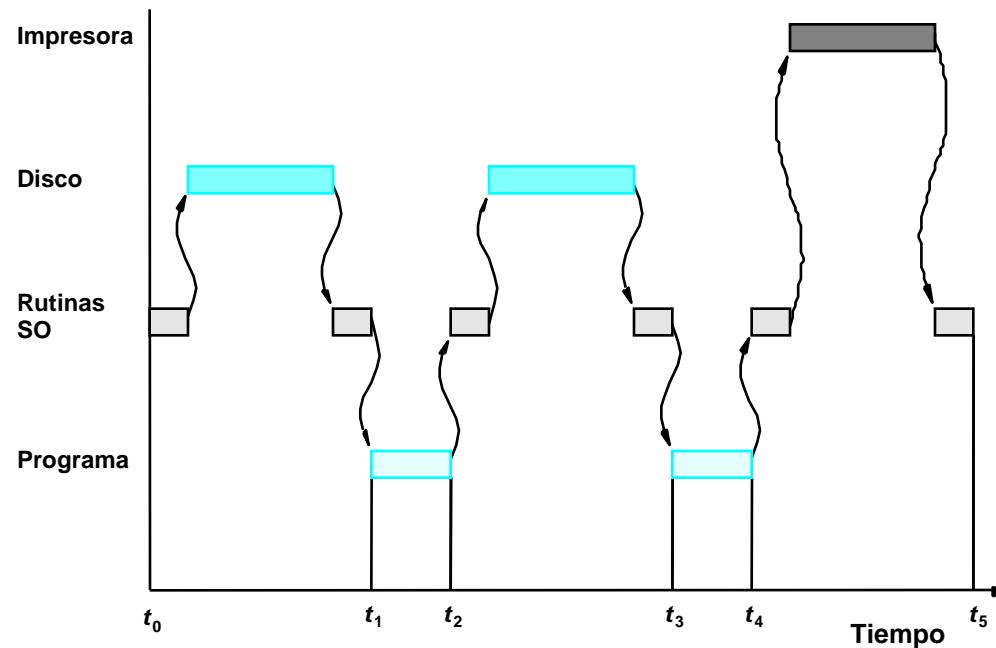
■ SO gestiona recursos (especialmente multiuser/multitask)

- ej: solapar E/S final con carga siguiente tarea
- ej: conmutar proceso en cuanto haga E/S

Software de sistema

■ Pensar tareas realizadas por SO para ejecutar aplicación

- Por ejemplo: leer datos HD, cálculos, imprimir resultados
- Pensar entonces cómo solapar varias de esas aplicaciones
- Detalles en [HAM03] Cap-1.5



Introducción

- Unidades funcionales
- Conceptos básicos de funcionamiento
- Estructuras de bus
- Rendimiento
- Perspectiva histórica

Rendimiento

■ Medida definitiva: **tiempo ejecución programa**

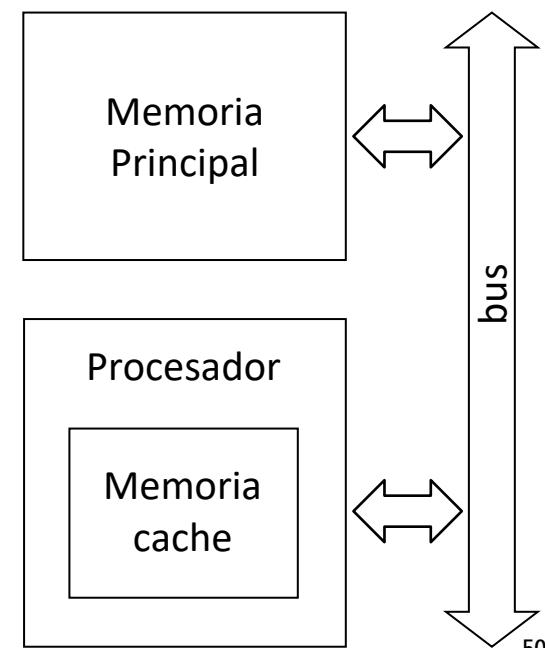
- Problema: ¿Cuál programa? Acordar benchmarks
- Depende de diseño CPU, repertorio instrucciones...
 - y del **compilador!!!** (benchmarks en lenguaje alto nivel)
 - y versión **SO, librerías**, etc.

■ Ejemplo anterior: **t5-t0 incluye HD, LPR**

- Tiempo transcurrido (wall-clock time)
- Mide rendimiento sistema completo
 - Influido por prestaciones CPU, HD, LPR, etc

■ **Benchmarks CPU ejercitan sólo CPU**

- Tiempo de procesamiento (CPU time)
- Influido por prestaciones **CPU, M, caches, buses**
 - cache conserva lo accedido recientemente/más rápida
 - ventaja en ejecución bucles, p.ej.



Rendimiento

■ Reloj del procesador

- UC emplea **varios ciclos** de reloj en ejecutar una instrucción
- pasos básicos 1 ciclo (comutar señales control)
- Frecuencia $R = 1/P$
 - $500\text{MHz} = 1 / 2\text{ns}$
 - $1.25\text{GHz} = 1 / 0.8\text{ns}$

■ Ecuación básica de rendimiento

- T tiempo para ejecutar programa benchmark
- **N instrucciones** (recuento dinámico bucles/subrutinas)
 - N no necesariamente igual a #instr. progr. objeto.
- **S ciclos/instr.** (“pasos básicos” de media)

$$T = \frac{N \times S}{R} \quad \begin{matrix} \text{ciclos} \\ \text{ciclos/s} \end{matrix}$$

Rendimiento

■ Ecuación básica de rendimiento

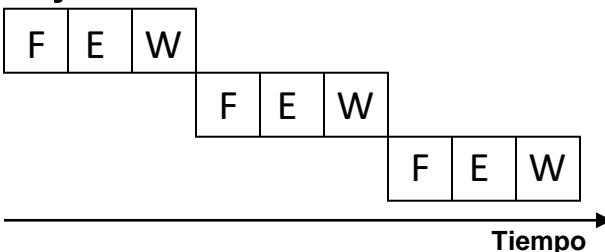
$$T = \frac{N \times S}{R} \quad \begin{matrix} \text{ciclos} \\ \text{ciclos/s} \end{matrix}$$

- Ideal: N y S ↓↓, R ↑↑
 - N (instrucciones) depende de compilador/repertorio
 - S (ciclos/instr) depende de implementación CPU
 - R (MHz - GHz) depende de tecnología (y diseño CPU)
- alterar uno modifica los otros
 - aumentar R puede ser a costa de aumentar S
 - lo importante es que al final T↓

Segmentación de cauce (intenta que $S \approx 1$)

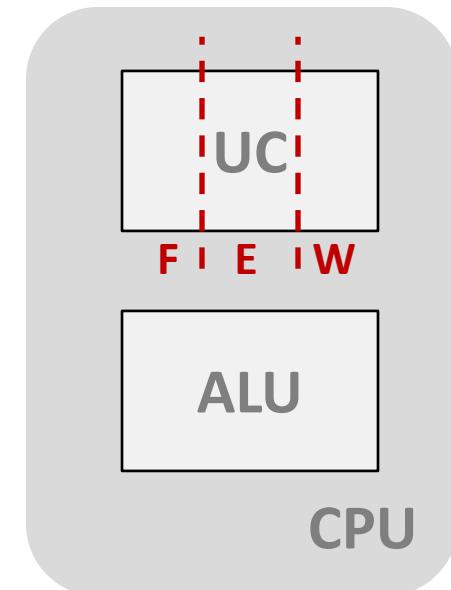
- NxS es suponiendo ejecución individual instrucciones

ADD R1,R2, R3



MUL R4,R5, R5

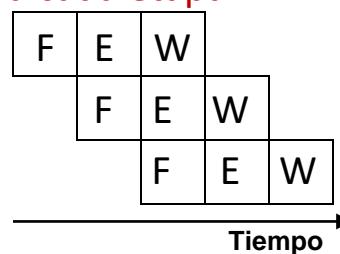
SUB R3,R5, R5



- Pero las distintas etapas hacen tareas distintas

- UC puede tener **circuitería separada para cada etapa:**

- Fetch: captación
- Exec: ejecución
- Write: actualización registro



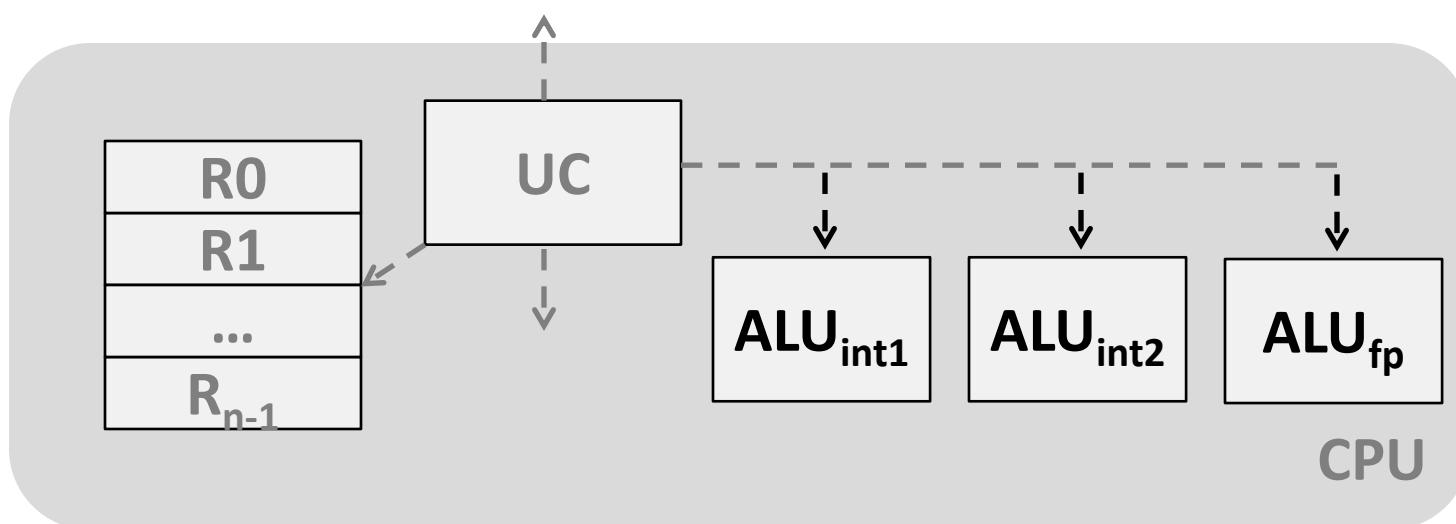
- una vez lleno el cauce**, valor efectivo $S=1$ ciclo/instr

- dependencias datos
- competición recursos
- saltos
- $S \geq 1, S \approx 1$

- (ej: MUL-SUB arriba, dependencia R5)
- (ej: almacenar resultado M/fetch instrucción+2)
(pipeline flush)

Funcionamiento superescalar (que S<1)

- Conseguir paralelismo a base de **reduplicar UFs** (unidades funcionales)
 - ej: 2 ALU enteros, 2 ALU FP
 - emitir hasta 4 operaciones simultáneas (2int+2fp)
 - si orden apropiado instrucciones (en secuencia programa)
 - combinado con segmentación, puede hacer S<1
 - se completa más de 1 instrucción por ciclo
- común en CPUs actuales. Dificultades:
 - **emisión desordenada**
 - **corrección** (mismo resultado que ejecución escalar)



Otras formas de reducir T

■ Velocidad del reloj (R↑, S/R)

- Tecnología $\uparrow \Rightarrow R\uparrow$
 - Si no cambia nada más, Rx2 $\Rightarrow T/2?$ ($T = NS / R$)
 - Falso: Memoria también Rx2 !!! o mejorar cache L1-L2
- Alternativamente, S $\uparrow \Rightarrow R\uparrow$
 - “supersegmentación”, reducir tarea por ciclo reloj
 - difícil predecir ganancia, puede incluso empeorar

■ Repertorio RISC/CISC (N·S)

- RISC: instr. simples para $R\uparrow\uparrow$, pero $S\downarrow \Rightarrow N\uparrow$
- CISC: instr. complejas para $N\downarrow\downarrow$, pero $S\uparrow$
 - corregir $S\uparrow$ con segmentación \Rightarrow competición recursos
- actualmente técnicas híbridas RISC/CISC

Otras formas de reducir T

■ Compilador $(N \downarrow)$

- optimizador espacial ($N \downarrow$) o temporal ($N \times S \downarrow$)
 - usualmente contrapuestos
- espacial: requiere conocimiento **arquitectura**
 - repertorio, modos direccionamiento, alternativas traducción...
- temporal: requiere conocimiento detallado **organización**
 - reordenación instrucciones para ahorrar ciclos
 - evitar competición recursos
- optimización debe ser **correcta** (mismo resultado)

Medida del rendimiento

■ Interesante para:

- diseñadores CPUs: evaluar mejoras introducidas
- fabricantes: marketing
- **compradores:** prestaciones/precio

■ Benchmark: 1 único programa acordado ?!?

- programas **sintéticos** no predicen bien T_{app}
- programas **reales** muy específicos
- colección programas considerados “**frecuentes**” (representativos)
- reducir a un único número usando **media geométrica**
 - evitar influencia computador referencia

Medida del rendimiento

■ SPEC: System Performance Evaluation Corporation

- tests: CPU92, CPU95, CPU2000, CPU2006 (CPUv6)
- CPU2006:
 - Referencia: WS UltraSPARC II 300MHz
 - CINT2006: gzip, bzip2, gcc, chess, gene seq., Perl... (12)
 - CFP2006: CFD, chromodynamics, ray-tracing, meteo... (17)

$$\text{vel}_{\text{gzip}} = T_{\text{ref}}_{\text{gzip}} / T_{\text{gzip}} \quad (\text{vel}=50 \Rightarrow 50x \text{ uSPARC II})$$

$$\text{vel}_{\text{SPEC}} = \sqrt[n]{\prod_i \text{vel}_{\text{prgi}}} \quad (n=12/17, \text{ media geom.})$$

■ mide efecto combinado

- CPU, M, SO, compilador
- <http://www.spec.org> (no es gratuito)

Introducción

- Unidades funcionales
- Conceptos básicos de funcionamiento
- Estructuras de bus
- Rendimiento
- Perspectiva histórica

Perspectiva histórica

■ 2ª Guerra Mundial

- Tecnología **relés** electromagnéticos $T_{\text{conmut.}} = O(s)$
 - Previamente: engranajes, palancas, poleas
- Tablas logaritmos, aprox. func. trigonométricas
- Generaciones 1-2-3-4ª 1945-55-65-75-etc

■ 1ª Generación (45-55): tubos de vacío

- von Neumann: concepto **prog. almacenado**
- tubos vacío 100-1000x $T_{\text{conmut.}} = O(ms)$
- M: líneas retardo mercurio, **núcleos magn.**
- E/S: lect/perf. tarjetas, cintas magnéticas
- software: lenguaje máquina / ensamblador
 - 1946-47 **ENIAC** UNIVAC
 - 1952-57 EDVAC UNIVAC II
 - 1953-55 IBM 701 702



Perspectiva histórica

■ 2ª Generación (55-65): transistores

- invento Bell AT&T 1947 $T_{\text{conmut.}} = O(\mu\text{s})$
- E/S: procesadores E/S (cintas) en paralelo con CPU
- software: compilador FORTRAN
 - 1955-57 IBM704 DEC PDP-1
 - 1964 IBM 7094



■ 3ª Generación (1965-75): Circuito Integrado

- velocidad CPU/M ↑ $T_{\text{conmut.}} = O(\text{ns})$
- arquitectura: μProgr, segm. cauce, M cache
- software: SO multiusuario, memoria virtual
 - 1965 IBM S/360
 - 1971-77 DEC PDP-8



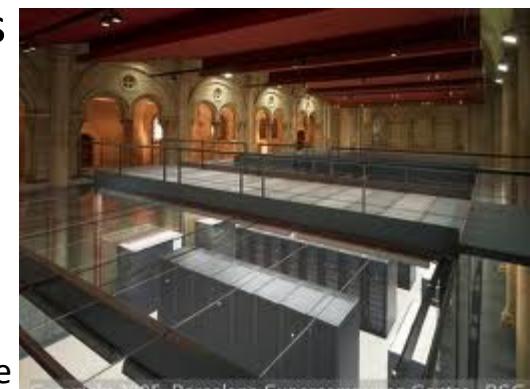
Perspectiva histórica

■ 4^a Generación (75-...): VLSI

- μProcesador: procesador completo en 1 chip
 - MP completa en uno o pocos chips
 - Intel, Motorola, AMD, TI, NS
- arquitectura: mejoras segm. cauce, cache, M virtual
- hardware: **portátiles**, PCs, WS, **redes**
- mainframes siguen sólo en grandes empresas
 - 1972-74-78 i8008 i8080 i8086
 - 1982-85-89 i80286 i80386 i80486

■ Actualidad

- Computadores sobremesa potentes/asequibles
- Internet
- Paralelismo masivo (Top500, **MareNostrum**, **Magerit**)
 - 1995-97-99-01 Pentium PII PIII P4
 - 2004-06-08 Pentium 4F, Core 2 Duo, Core i7
 - 2011-15-20 Core i7 2nd-6thgen, Kaby/Coffee/Cannon/Ice Lake



Introducción

■ Unidades funcionales

- E/S, M, CPU (ALU+UC)
- Memoria de bytes, alineamiento, ordenamiento
- Clasificación arq. m/n, pila, acumulador, RPG (R/R, R/M, M/M)
- Repertorios RISC/CISC, modos de direccionamiento

■ Conceptos básicos de funcionamiento

- Ciclo de ejecución

■ Estructuras de bus

- Bus único, buses múltiples, decodificación parcial/completa

■ Rendimiento

- Software de sistema, ecuación básica rendimiento, benchmarks
- Segmentación, funcionamiento superescalar, SPEC

■ Perspectiva histórica

- generaciones

Guía de trabajo autónomo (4h/s)

■ Estudio

- Cap.1 Hamacher (incluye problemas)

■ Lectura

- Guión de la Práctica 2
- Cap.3 CS:APP (Bryant/O'Hallaron)

■ Para los entusiastas: Ubuntu en el portátil (Ubuntu LTS 18.04 en ETSIIT)

- Posibilidades de usar Ubuntu en portátil:
 - Instalación directa (además de, o en lugar de, Windows)
 - VirtualBox + Ubuntu LTS (es +complicado, pero +ventajoso)
 - » no requiere rebotar, no toca MBR, se puede usar Windows a la vez
 - instalar paquetes **g++/make/ghex** (usar apt o Synaptic), y **default-jre** (para eclipse)
 - instalar snap **eclipse 4.8** (usar snap o UbuntuSoftware) (evitar paquete **eclipse 3.8**)
 - comprobar firewall con “`sudo ufw [status | enable]`”
- Instálarselo e intentar Ejercicios 1-4 del guión P2

Programación a Nivel-Máquina I: Conceptos Básicos y Aritmética

Estructura de Computadores
Semana 3

Bibliografía:

[BRY16] Cap.3 Computer Systems: A Programmer's Perspective 3rd ed. Bryant, O'Hallaron. Pearson, 2016
 Signatura ESIIT/[C.1 BRY com](#)

Transparencias del libro CS:APP, Cap.3

Introduction to Computer Systems: a Programmer's Perspective

Autores: Randal E. Bryant y David R. O'Hallaron

<http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/schedule.html>

Guía de trabajo autónomo (4h/s)

■ Lectura: del Cap.3 CS:APP (Bryant/O'Hallaron)

- Historical perspective, Program Encodings
 - § 3.1 – 3.2 pp.199-213
- Data Formats, Accessing Info.
 - § 3.3 – 3.4 pp.213-227
- Arithmetic and Logical Operations
 - § 3.5 pp.227-236

■ Ejercicios: del Cap.3 CS:APP (Bryant/O'Hallaron)

- Probl. 3.1 – 3.5 § 3.4, pp.218, 221, 222, 223, 225
- Probl. 3.6 – 3.12 § 3.5, pp.228, 229, 230, 231, 232, 233, 236

Bibliografía:

[BRY16] Cap.3

Computer Systems: A Programmer's Perspective 3rd ed. Bryant, O'Hallaron. Pearson, 2016

Signatura ESIIT/[C.1 BRY com](#)

Programación Máquina I: Conceptos Básicos

- Historia de los procesadores y arquitecturas de Intel
- Lenguaje C, ensamblador, código máquina
- Conceptos básicos asm: Registros, operandos, move
- Operaciones aritméticas y lógicas

Procesadores Intel x86

- **Dominan el mercado portátil/sobremesa/servidor**
- **Diseño evolutivo**
 - Compatible ascendentemente hasta el 8086, introducido en 1978
 - Va añadiendo características conforme pasa el tiempo
- **Computador con repertorio instrucciones complejo (CISC)**
 - Muchas instrucciones diferentes, con muchos formatos distintos
 - Pero sólo un pequeño subconjunto aparece en programas Linux
 - Difícil igualar prestaciones Computadores Repertorio Instr. Reducido (RISC)
 - Sin embargo, ¡Intel ha conseguido justo eso!
 - En lo que a velocidad se refiere. No tanto en (bajo) consumo.

Evolución Intel x86: Hitos significativos

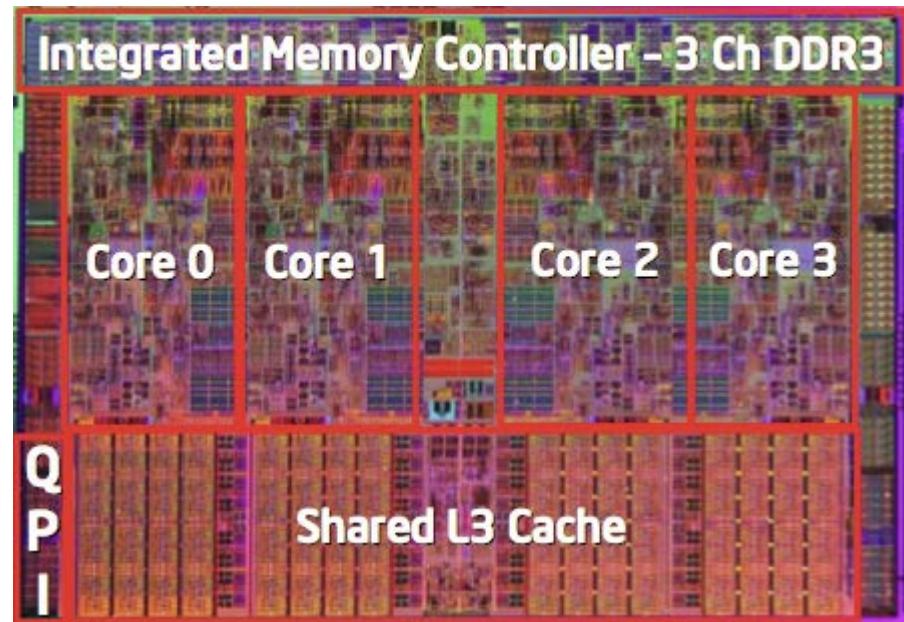
<i>Nombre</i>	<i>Fecha</i>	<i>Transistores</i>	<i>MHz</i>
■ 8086	1978	29K	5-10
			<ul style="list-style-type: none">■ Primer procesador Intel 16-bit. Base para el IBM PC & MS-DOS■ Espacio direccionamiento 1MB
■ 386	1985	275K	16-33
			<ul style="list-style-type: none">■ Primer procesador Intel 32-bit de la familia (x86 luego llamada) IA32■ Añadió “direcciónamiento plano”[†], capaz de arrancar Unix
■ Pentium 4E	2004	125M	2800-3800
			<ul style="list-style-type: none">■ 1^{er} proc. Intel 64-bit de la familia (x86, llamada x86-64, EM64t) Intel 64
■ Core 2	2006	291M	1060-3500
			<ul style="list-style-type: none">■ Primer procesador Intel multi-core
■ Core i7	2008	731M	1700-3900
			<ul style="list-style-type: none">■ Cuatro cores, hyperthreading (2 vías)

[†] “flat addressing” 5

Procesadores Intel x86: Visión general

■ Evolución de las máquinas

■ 386	1985	0.3M
■ Pentium	1993	3.1M
■ Pentium/MMX	1997	4.5M
■ PentiumPro	1995	6.5M
■ Pentium III	1999	8.2M
■ Pentium 4	2001	42M
■ Core 2 Duo	2006	291M
■ Core i7	2008	731M
■ Core i7 Skylake	2015	1.9B



Microfotografía de un dado Core i7

■ Características añadidas

- Instrucciones de soporte para operación multimedia (ops. en paralelo)
- Instrucciones para posibilitar operaciones condicionales más eficientes
- Transición de 32 bits a 64 bits
- Más núcleos (cores)

Procesadores Intel x86

■ Generaciones pasadas Tecnología del proceso

- 1st Pentium Pro 1995 600 nm
- 1st Pentium III 1999 250 nm
- 1st Pentium 4 2000 180 nm
- 1st Core 2 Duo 2006 65 nm

Tamaño tecnología proceso
= anchura mínima trazo
(10 nm ≈ 100 átomos ancho)

■ Generaciones recientes y venideras

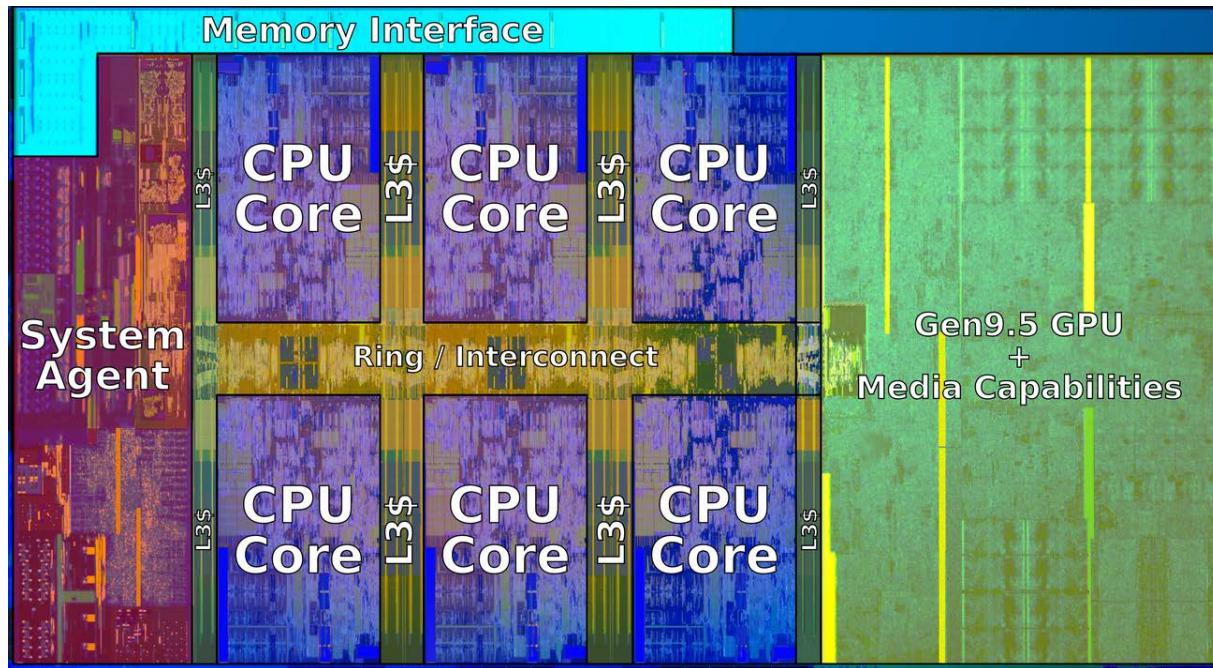
1. Nehalem 2008 45 nm
2. Sandy Bridge 2011 32 nm
3. Ivy Bridge 2012 22 nm
4. Haswell 2013 22 nm
5. Broadwell 2014 14 nm
6. Skylake 2015 14 nm
7. Kaby Lake 2016 14 nm
8. Coffee Lake 2017 14 nm
- Cannon Lake 2019? 10 nm

Lo último[†] en 2018

- Core i7 CoffeeLake 2018 (SkyLake 6xxx, KabyLake 7xxx, CoffeeLake 8xxx)

■ Modelo móvil: Core i7

- 2.2-3.2 GHz
- 45 W



■ Sobremesa: Core i7

- Gráficos integrados
- 2.4-4.0 GHz
- 35-95 W

■ Modelo servidor: Xeon E

- Gráficos integrados
- Habilitado para multi-zócalo[#]
- 3.3-3.8 GHz
- 80-95 W

[†] "state of the art"

[#] "multi-socket" 8

Clones x86: Advanced Micro Devices (AMD)

■ Históricamente

- AMD ha ido siguiendo a Intel en todo
- CPUs un poco más lentas, mucho más baratas

■ Y entonces

- Reclutaron los mejores diseñadores de circuitos de Digital Equipment Corp. y otras compañías con tendencia descendente
- Construyeron el Opteron: duro competidor para el Pentium 4
- Desarrollaron x86-64, su propia extensión a 64 bits

■ En años recientes

- Intel ha empezado a organizarse para ser más efectiva
 - Lidera el mundo de tecnologías de semiconductores
- AMD se ha quedado rezagada
 - Recurre a fabricante de semiconductores externalizado

La historia de los 64-bit de Intel

- **2001: Intel intenta un cambio radical de IA32 a IA64**
 - Arquitectura totalmente diferente (Itanium)
 - Ejecuta código IA32 sólo como herencia[†]
 - Prestaciones decepcionantes
- **2003: AMD interviene con una solución evolutiva**
 - x86-64 (ahora llamado “AMD64”)
- **Intel se sintió obligada a concentrarse en IA64**
 - Difícil admitir error, o admitir que AMD es mejor
- **2004: Intel anuncia extensión EM64T[‡] de la IA32** (ahora llamada Intel64)
 - Extended Memory 64-bit Technology
 - ¡Casi idéntica a x86-64!
- **Todos los procesadores x86 salvo gama baja soportan x86-64**
 - Pero gran cantidad de código se ejecuta aún en modo 32-bits

[†] “legacy” = herencia de características

[‡] Intel usa ahora “IA32” e “Intel64” para

distinguir IA32 de EM64T y evitar confusión con IA64 10

Nosotros cubrimos:

■ IA32

- El ~~x86 tradicional~~
- Para EC: ~~RIP, verano 2018~~

■ x86-64 / Intel64

- El **estándar**
- ubuntu_18> gcc hello.c
- ubuntu_18> gcc -m64 hello.c

■ Presentación

- El libro cubre x86-64. Transparencias, prácticas, ejercicios... todo en x86-64.
- En el libro hay un “añadido Web”[†] sobre IA32
- En SWAD puede quedar material (tests/exámenes/...) sobre IA32
- Sólo algunos **detalles** querremos recordar de IA32 (alineamiento, pila...)

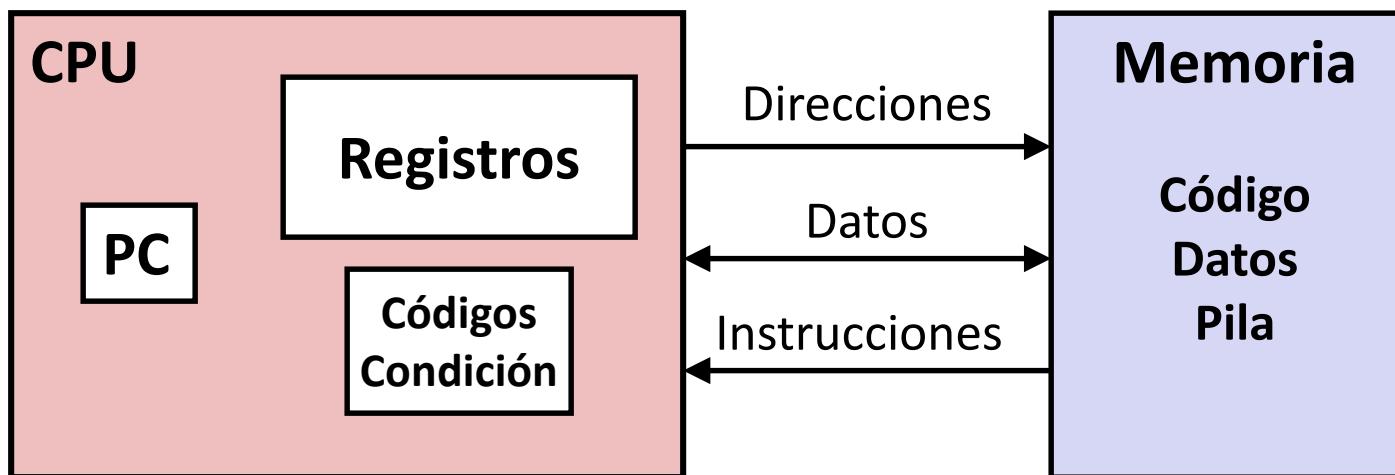
Programación Máquina I: Conceptos Básicos

- Historia de los procesadores y arquitecturas de Intel
- Lenguaje C, ensamblador, código máquina
- Conceptos básicos asm: Registros, operandos, move
- Operaciones aritméticas y lógicas

Definiciones

- **Arquitectura:** (también arquitectura del repertorio de instrucciones: ISA) Las partes del diseño de un procesador que se necesitan entender para escribir **código ensamblador**.
 - Ejemplos: especificación del repertorio de instrucciones, registros.
- **Formas del código:**
 - **Código máquina:** Programas (codops, bytes) que ejecuta el procesador
 - **Código ensamblador:** Representación textual del código máquina
- **Microarquitectura: Implementación de la arquitectura.**
 - Ejemplos: tamaño de las caches y frecuencia de los cores.
- **Ejemplos de ISAs:**
 - Intel: (x86 =) IA32, Itanium (= IA64 = IPF), x86-64 (= Intel 64 = EM64t)
 - ARM: Usado en casi todos los teléfonos móviles
 - RISC V: nueva ISA open-source

Perspectiva Código Ensamblador/Máquina

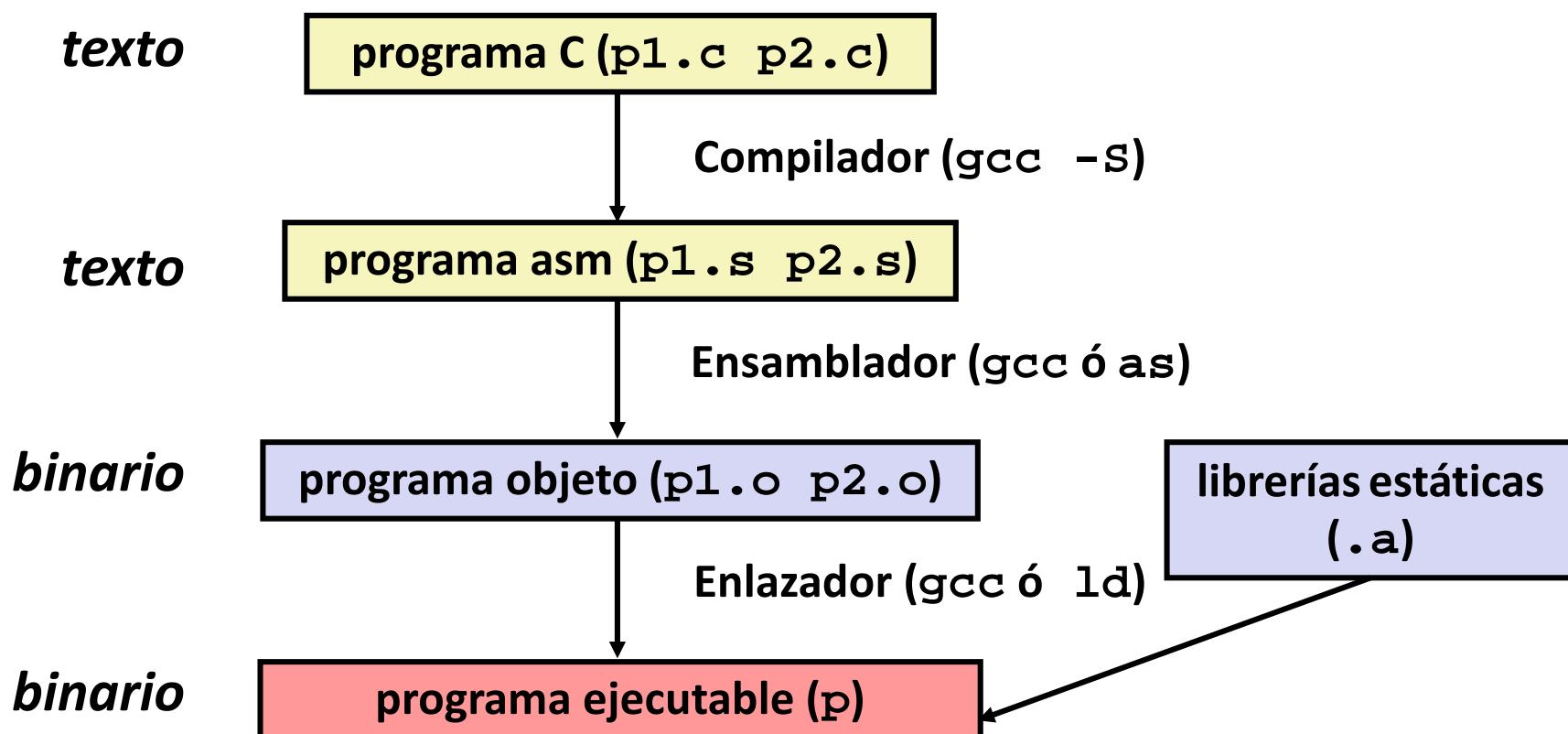


Estado visible al programador

- **PC: Contador de programa**
 - Dirección de la próxima instrucción
 - Llamado “RIP” (x86-64)
- **Archivo de registros**
 - Datos del programa muy utilizados
- **Códigos de condición / flags de estado**
 - Almacenan información estado sobre la operación aritmética/lógica más reciente
 - Usados para bifurcación condicional
- **Memoria**
 - Array direccionable por bytes
 - Código y datos usuario
 - Pila soporte a procedimientos

Convertir C en Código Objeto

- Código en ficheros `p1.c p2.c`
- Compilar con el comando: `gcc -Og p1.c p2.c -o p`
 - Usar optimizaciones básicas (`-Og`) [versiones recientes de GCC[†]]
 - Poner binario resultante en fichero `p`



Compilar a ensamblador

Código C (sum.c)

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

Ensamblador x86-64 generado[†]

sumstore:

pushq	%rbx
movq	%rdx, %rbx
call	plus
movq	%rax, (%rbx)
popq	%rbx
ret	

Obtenerlo con el comando

```
gcc -Og -S sum.c
```

Produce el fichero sum.s

Aviso: Se obtendrán resultados diferentes en cuanto se usen diferentes versiones de gcc y diferentes ajustes del compilador[†]

[†] Añadir *-fno-asynchronous-unwind-tables*
en GCC 7.3 Ubuntu 18.04 16

Representación Datos C, IA32, x86-64

■ Tamaño de Objetos C (en Bytes)

<i>Tipo de Datos C</i>	<i>Normal 32-bit</i>	<i>Intel IA32</i>	<i>x86-64</i>
▪ unsigned	4	4	4
▪ int	4	4	4
▪ long int	4	4	8
▪ char	1	1	1
▪ short	2	2	2
▪ float	4	4	4
▪ double	8	8	8
▪ long double	8	10/12	16
▪ char *	4	4	8

– o cualquier otro puntero

Características Ensamblador: Tipos de Datos

- **Datos “enteros” de 1, 2, 4 u 8 bytes**
 - Valores de datos
 - Direcciones (punteros sin tipo)
- **Datos en punto flotante de 4, 8 ó 10 bytes**
- **Código: secuencias de bytes codificando serie de instrucciones**
- **No hay tipos compuestos como arrays o estructuras**
 - Tan sólo bytes ubicados contiguamente (uno tras otro) en memoria

Características Ensamblador: Instrucciones

■ Realizan función aritmética sobre datos en registros o memoria

- “Operaciones” = Instrucciones aritmético/lógicas

■ Transfieren datos entre memoria y registros

- Cargar datos de memoria a un registro
- Almacenar datos de un registro en memoria
- “Instrucciones de transferencia”

■ Transferencia de control

- Incondicionales: saltos, llamadas a procedimientos, retornos desde procs.
- Saltos condicionales
- “Instrucciones de control”

Código Objeto

Código de sumstore

0x0400595:

0x53

0x48

0x89

0xd3

0xe8

0xf2

0xff

0xff

0xff

- 14 bytes total

- Cada instrucción

- 1, 3, ó 5 bytes

- Empieza en direcc.

0xc3 0x0400595

■ Ensamblador

- Traduce .s pasándolo a .o
- Instrucciones codificadas en binario
- Imagen casi completa del código ejecutable
- Le faltan enlaces entre código de ficheros diferentes

■ Enlazador

- Resuelve referencias entre ficheros
- Combina con libs. de tiempo ejec. estáticas[†]
 - P.ej., código para **malloc**, **printf**
- Algunas libs. son *dinámicamente enlazadas*[#]
 - El enlace ocurre cuando el programa empieza a ejecutarse

[†] “static run-time libraries” = bibliotecas estáticas para soporte en tiempo de ejecución
[#] “dynamically linked libraries”, o también “shared libs”

Ejemplo de Instrucción Máquina

```
*dest = t;
```

■ Código C

- Almacenar valor **t** adonde indica (apunta) **dest**

```
movq %rax, (%rbx)
```

■ Ensamblador

- Mover un valor de 8-byte a memoria
 - “Palabra Quad”[†] en jerga x86-64
- Operандos:
 - t:** Registro **%rax**
 - dest:** Registro **%rbx**
 - *dest:** Memoria **M[%rbx]**

```
0x40059e: 48 89 03
```

■ Código Objeto

- Instrucción de 3-byte
- Almacenada en dir. **0x40059e**

Desensamblar Código Objeto

Desensamblado

```
0000000000400595 <sumstore>:  
 400595: 53          push    %rbx  
 400596: 48 89 d3    mov      %rdx,%rbx  
 400599: e8 f2 ff ff ff    callq   400590 <plus>  
 40059e: 48 89 03    mov      %rax,(%rbx)  
 4005a1: 5b          pop     %rbx  
 4005a2: c3          retq
```

■ Desensamblador

`objdump -d sum`

- Herramienta útil para examinar código objeto
- Analiza el patrón de bits de series de instrucciones
- Produce versión aproximada del código ensamblador (correspondiente)
- Puede ejecutarse sobre el fich. a.out (ejecutable completo) ó el .o

Desensamblado Alternativo

Objeto

```
0x0400595:  
 0x53  
 0x48  
 0x89  
 0xd3  
 0xe8  
 0xf2  
 0xff  
 0xff  
 0xff  
 0x48  
 0x89  
 0x03  
 0x5b  
 0xc3
```

Desensamblado

```
Dump of assembler code for function sumstore:  
 0x0000000000400595 <+0>: push    %rbx  
 0x0000000000400596 <+1>: mov     %rdx,%rbx  
 0x0000000000400599 <+4>: callq   0x400590 <plus>  
 0x000000000040059e <+9>: mov     %rax,(%rbx)  
 0x00000000004005a1 <+12>:pop    %rbx  
 0x00000000004005a2 <+13>:retq
```

■ Desde el Depurador gdb

```
gdb sum
```

```
disassemble sumstore
```

- Desensamblar procedimiento

```
x/14xb sumstore
```

- Examinar 14 bytes a partir de sumstore



¿Qué se puede Desensamblar?

```
% objdump -d WINWORD.EXE  
  
WINWORD.EXE:      file format pei-i386  
  
No symbols in "WINWORD.EXE".  
Disassembly of section .text:  
  
30001000 <.text>:  
30001000:  
30001001:  
30001003:  
30001005:  
3000100a:  Ingeniería inversa prohibida por licencia  
            Microsoft End User License Agreement  
            (EULA)
```

- Cualquier cosa que se pueda interpretar como código ejecutable
- El desensamblador examina bytes y reconstruye el fuente asm.

Programación Máquina I: Conceptos Básicos

- Historia de los procesadores y arquitecturas de Intel
- Lenguaje C, ensamblador, código máquina
- **Conceptos básicos asm: Registros, operandos, move**
- Operaciones aritméticas y lógicas

Registros enteros x86-64

%rax	%eax	%ax	%al
-------------	-------------	------------	------------

%r8	%r8d	%r8w	%r8b
------------	-------------	-------------	-------------

%rbx	%ebx
-------------	-------------

%r9	%r9d
------------	-------------

%rcx	%ecx
-------------	-------------

%r10	%r10d
-------------	--------------

%rdx	%edx
-------------	-------------

%r11	%r11d
-------------	--------------

%rsi	%esi	%si	%sil
-------------	-------------	------------	-------------

%r12	%r12d
-------------	--------------

%rdi	%edi
-------------	-------------

%r13	%r13d
-------------	--------------

%rsp	%esp
-------------	-------------

%r14	%r14d
-------------	--------------

%rbp	%ebp
-------------	-------------

%r15	%r15d
-------------	--------------

- Pueden referenciarse los 4 bytes de menor peso[†] (los 4 LSBs)
 - (también los 2 LSB y el 1 LSB)

[†] “low-order 4 bytes” 26

Un poco de historia: registros IA32

propósito general

%eax	%ax	%ah	%al
%ecx	%cx	%ch	%cl
%edx	%dx	%dh	%dl
%ebx	%bx	%bh	%bl
%esi	%si		
%edi	%di		
%esp	%sp		
%ebp	%bp		

Motivos nombre
(mayoría obsoletos)

acumulador

contador

datos

base

*índice
fuente*

*índice
destino*

*puntero
de pila*

*puntero
base*

registros virtuales 16-bit
(compatibilidad ascendente)

Mover Datos

■ Mover Datos

`movq Source, Dest†`

■ Tipo de Operandos

- **Inmediato:** Datos enteros constantes
 - Ejemplo: `$0x400, $-533`
 - Como constante C, pero con prefijo '`$`'
 - Codificado mediante 1, 2, ó 4 bytes[‡]
- **Registro:** Alguno de los 16 registros enteros
 - Ejemplo: `%rax, %r13`
 - Pero `%rsp` reservado para uso especial
 - Otros tienen usos especiales con instrucciones particulares
- **Memoria:** 8 bytes consecutivos mem. en dirección dada por un registro
 - Ejemplo más sencillo: (`%rax`)
 - Hay otros diversos “modos de direccionamiento”

`%rax`

`%rcx`

`%rdx`

`%rbx`

`%rsi`

`%rdi`

`%rsp`

`%rbp`

`%rN`

[†] luego veremos `movabsq` para literales 8B

[‡] “source/destination” = fuente/destino 28

Combinaciones de Operandos movq

	Source	Dest	Src,Dest	Análogo C
movq	<i>Imm^t</i>	<i>Reg</i>	movq \$0x4,%rax	temp = 0x4;
		<i>Mem</i>	movq \$-147,(%rax)	*p = -147;
	<i>Reg</i>	<i>Reg</i>	movq %rax,%rdx	temp2 = temp1;
	<i>Mem</i>	movq %rax,(%rdx)	*p = temp;	
	<i>Mem</i>	<i>Reg</i>	movq (%rax),%rdx	temp = *p;

Ver resto instrucciones transferencia (incluyendo pila) en el libro

No se puede transferir Mem-Mem con sólo una instrucción

Modos Direcccionamiento a memoria sencillos

■ Normal[†]

(R)

Mem[Reg[R]]

- El registro R indica la dirección de memoria
- ¡Exacto! Como seguir (*desreferenciar[#]*) un puntero en C

movq (%rcx), %rax

■ Desplazamiento D(R)

Mem[Reg[R]+D]

- El registro R indica el inicio de una región de memoria
- La constante de desplazamiento D indica el *offset[#]*

movq 8(%rbp), %rdx

[#] “offset”=compensación, para nosotros “desplazamiento”

[†] “indirecto a través de registro” según otros autores

[#] “dereferencing” en el original

Ejemplo Modos Direcccionamiento sencillos

```
void adiv(<tipo> a, <tipo> b)
{
    ????
    ?? = ???;
    ???
    ?? = ???;
    ???
    ?? = ??;
    ???
    ?? = ??;
}
```

%rdi

%rsi

adiv:

movq	(%rdi), %rax
movq	(%rsi), %rdx
movq	%rdx, (%rdi)
movq	%rax, (%rsi)
ret	

Ejemplo Modos Direcccionamiento sencillos

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

swap:

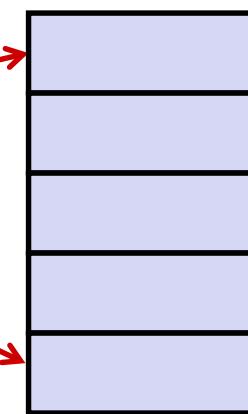
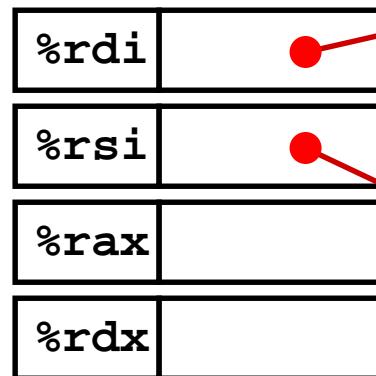
movq	(%rdi), %rax
movq	(%rsi), %rdx
movq	%rdx, (%rdi)
movq	%rax, (%rsi)
ret	

Comprendiendo swap()

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Memoria

Registros



Registro	Valor
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

swap:

```
        movq    (%rdi), %rax    # t0 = *xp
        movq    (%rsi), %rdx    # t1 = *yp
        movq    %rdx, (%rdi)    # *xp = t1
        movq    %rax, (%rsi)    # *yp = t0
        ret
```

Comprendiendo swap()

Registers

%rdi	0x120
%rsi	0x100
%rax	
%rdx	

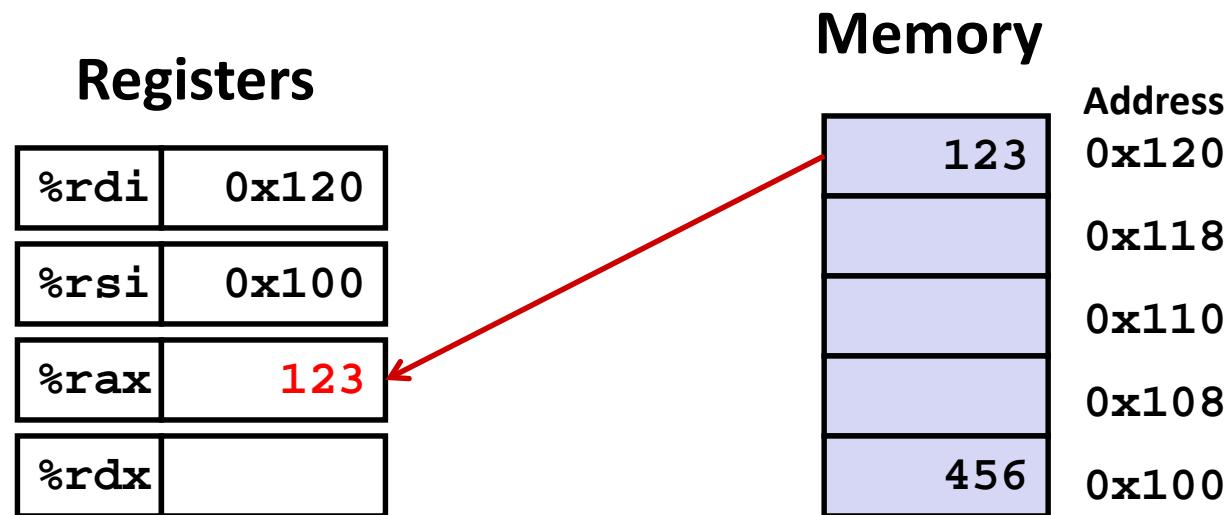
Memory

123	Address 0x120
	0x118
	0x110
	0x108
456	0x100

swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

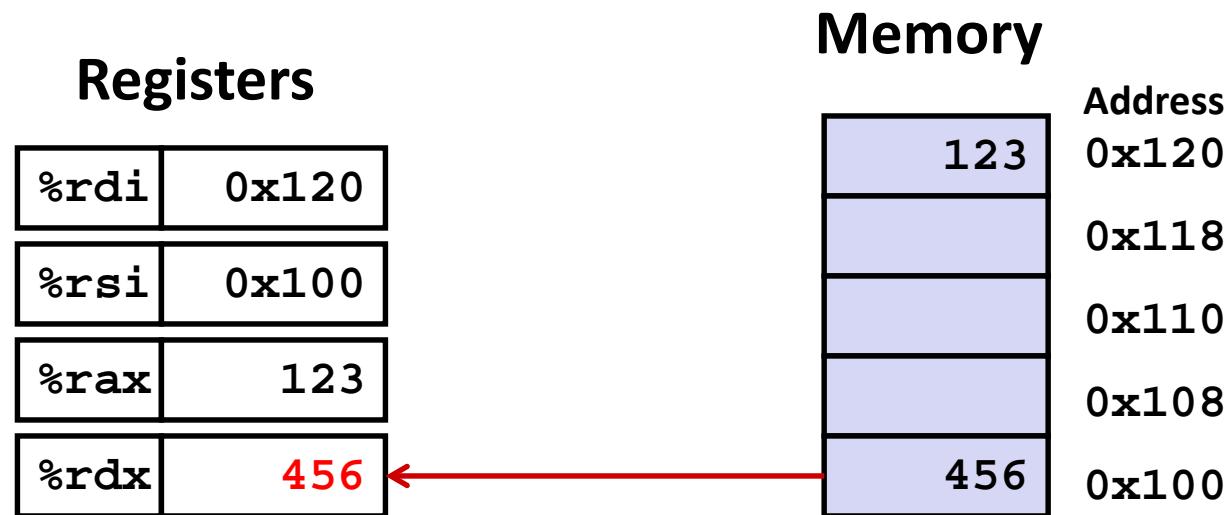
Comprendiendo swap()



swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

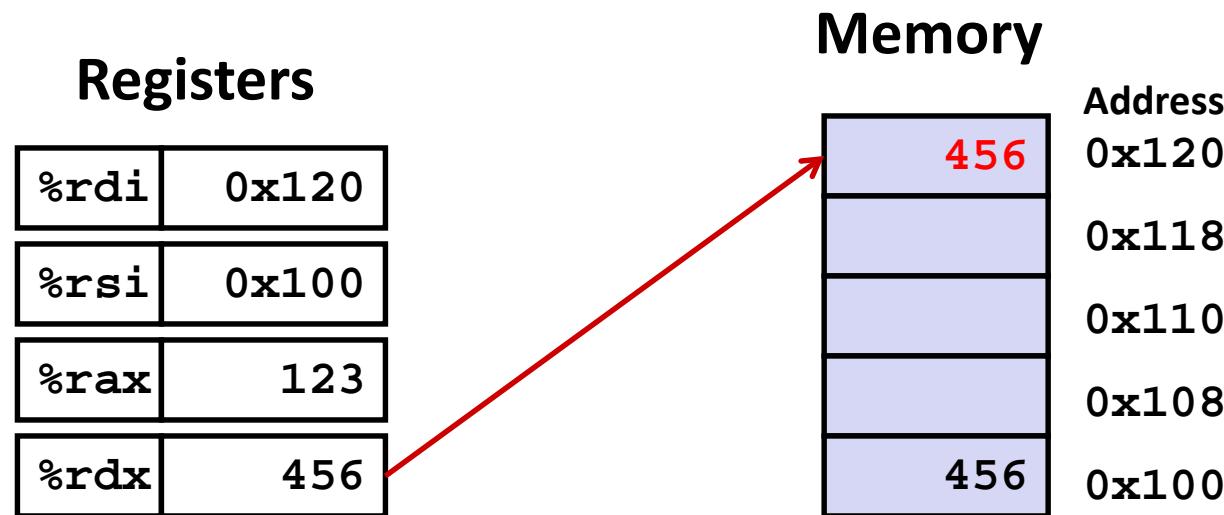
Comprendiendo swap()



swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

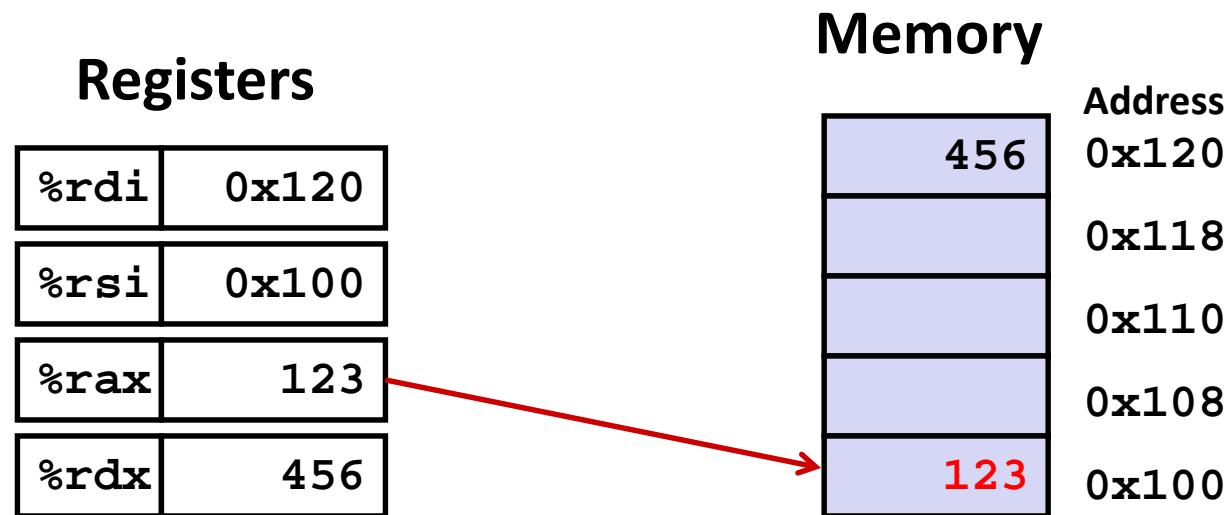
Comprendiendo swap()



swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

Comprendiendo swap()



swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

Modos Direcccionamiento a memoria sencillos

■ Normal[‡]

(R)

Mem[Reg[R]]

- El registro R indica la dirección de memoria
- ¡Exacto! Como seguir (*desreferenciar*[†]) un puntero en C

`movq (%rcx), %rax`

■ Desplazamiento D(R)

Mem[Reg[R]+D]

- El registro R indica el inicio de una región de memoria
- La constante de desplazamiento D indica el *offset*^{*}

`movq 8(%rbp), %rdx`

Modos Direcccionamiento a memoria completos

■ Forma más general

$$D(Rb, Ri, S) \quad \text{Mem}[Reg[Rb]+S*Reg[Ri]+ D]$$

- D: “Desplazamiento” constante 1, 2, ó 4 bytes
- Rb: Registro base: Cualquiera de los 16 registros enteros
- Ri: Registro índice: Cualquiera, excepto %rsp
- S: Factor de escala: 1, 2, 4, ú 8 (*¿por qué esos números?*)

■ Casos Especiales

$$(Rb, Ri) \quad \text{Mem}[Reg[Rb]+Reg[Ri]]$$

$$D(Rb, Ri) \quad \text{Mem}[Reg[Rb]+Reg[Ri]+D]$$

$$(Rb, Ri, S) \quad \text{Mem}[Reg[Rb]+S*Reg[Ri]]$$

Ejemplos de Cálculo de Direcciones

%rdx	0xf000
%rcx	0x0100

D(Rb,Ri,S)

- D: “Desplazamiento” constante 1, 2, ó 4 bytes
- Rb: Registro base: Cualquiera de los 16 registros enteros
- Ri: Registro índice: Cualquiera, excepto %rsp
- S: Factor de escala: 1, 2, 4, ó 8 (*¿por qué esos números?*)

Mem[Reg[Rb]+S*Reg[Ri]+ D]

Expresión	Cálculo de Dirección	Dirección
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)	2*0xf000 + 0x80	0x1e080

Programación Máquina I: Conceptos Básicos

- Historia de los procesadores y arquitecturas de Intel
- Lenguaje C, ensamblador, código máquina
- Conceptos básicos asm: Registros, operandos, move
- Operaciones aritméticas y lógicas

Instrucción para el Cálculo de Direcciones

■ **leaq Src, Dest^t**

- *Src* es cualquier expresión de modo direccionamiento (a memoria)
- Ajusta *Dest* a la dirección indicada por la expresión

■ **Usos**

- Calcular direcciones sin hacer referencias a memoria
 - P.ej., traducción de $p = \&x[i]$;
- Calcular expresiones aritméticas de la forma $x + k*y$
 - $k = 1, 2, 4 \text{ ú } 8$

■ **Ejemplo**

```
long m12(long x)
{
    return x*12;
}
```

Traducción a ASM por el compilador:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax           # return t<<2
```

Algunas Operaciones Aritméticas

■ Instrucciones de Dos Operandos:

<i>Formato</i>	<i>Operación[†]</i>	
addq <i>Src,Dest</i>	Dest = Dest + Src	
subq <i>Src,Dest</i>	Dest = Dest – Src	
imulq <i>Src,Dest</i>	Dest = Dest * Src	
salq <i>Src,Dest</i>	Dest = Dest << Src	<i>También llamada shlq</i>
sarq <i>Src,Dest</i>	Dest = Dest >> Src	<i>Aritméticas</i>
shrq <i>Src,Dest</i>	Dest = Dest >> Src	<i>Lógicas</i>
xorq <i>Src,Dest</i>	Dest = Dest ^ Src	
andq <i>Src,Dest</i>	Dest = Dest & Src	
orq <i>Src,Dest</i>	Dest = Dest Src	

- ¡Cuidado con el orden de los argumentos! (Intel vs. AT&T)
- No se distingue entre enteros con/sin signo (*¿por qué?*)

Hora de hacer un test!

Conectarse a:

<https://swad.ugr.es/es?crs=5101>

Algunas Operaciones Aritméticas

■ Instrucciones de Un Operando:

<i>Formato</i>		<i>Operación</i>
incq	<i>Dest</i>	$Dest = Dest + 1$
decq	<i>Dest</i>	$Dest = Dest - 1$
negq	<i>Dest</i>	$Dest = - Dest$
notq	<i>Dest</i>	$Dest = \sim Dest$

■ Consultar más instrucciones en el libro†

- Aritméticas: [i]mulq Src, [i]divq Src, cqto
- Transferencia: movX (bwlq), movabsq,
movzXX (bw,bl,bq,wl,wq),‡
movsXX (bw,bl,bq,wl,wq,lq), cltq,
pushq, popq

† Según cómo se cuenten, hay entre 2000-3700 instrucciones en el manual

‡ luego veremos que movlq %eax, %rbx
sería lo mismo que mov %eax, %ebx 46

Ejemplo de Expresiones Aritméticas

```
long arith  
(long x, long y, long z)  
{  
    long t1 = x+y;  
    long t2 = z+t1;  
    long t3 = x+4;  
    long t4 = y * 48;  
    long t5 = t3 + t4;  
    long rval = t2 * t5;  
    return rval;  
}
```

arith:

leaq	(%rdi,%rsi), %rax
addq	%rdx, %rax
leaq	(%rsi,%rsi,2), %rdx
salq	\$4, %rdx
leaq	4(%rdi,%rdx), %rcx
imulq	%rcx, %rax
ret	

Instrucciones interesantes

- **leaq**: cálculo de direcciones
- **salq**: desplazamiento aritmético
- **imulq**: multiplicación
 - pero sólo se usa una vez

Comprendiendo arith()

```
long arith
(long x, long y, long z)
{
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

arith:

leaq	(%rdi,%rsi), %rax	# t1
addq	%rdx, %rax	# t2
leaq	(%rsi,%rsi,2), %rdx	
salq	\$4, %rdx	# t4
leaq	4(%rdi,%rdx), %rcx	# t5
imulq	%rcx, %rax	# rval
ret		

Registro	Uso(s)
%rdi	Argumento x
%rsi	Argumento y
%rdx	Argumento z
%rax	t1, t2, rval
%rdx	t4
%rcx	t5

Programación a Nivel-Máquina I: Resumen

- **Historia de los procesadores y arquitecturas de Intel**
 - Diseño evolutivo lleva a demasiados artefactos y peculiaridades
- **Lenguaje C, ensamblador, código máquina**
 - Nuevas formas de estado visible[†]: contador de programa, registros, ...
 - El compilador debe transformar sentencias, expresiones, procedimientos, en secuencias de instrucciones a bajo nivel
- **Conceptos básicos asm: Registros, operandos, move**
 - Las instrucciones x86-64 `mov` cubren un amplio rango de variedades de movimientos de datos (transferencia)
- **Operaciones aritméticas y lógicas**
 - El compilador C saldrá con diversas combinaciones de instrucciones para realizar los cálculos

Guía de trabajo autónomo (4h/s)

■ **Estudio:** del Cap.3 CS:APP (Bryant/O'Hallaron)

- Historical perspective, Program Encodings
 - § 3.1 – 3.2 pp.199-213
- Data Formats, Accessing Info.
 - § 3.3 – 3.4 pp.213-227
- Arithmetic and Logical Operations
 - § 3.5 pp.227-236

■ **Ejercicios:** del Cap.3 CS:APP (Bryant/O'Hallaron)

- Probl. 3.1 – 3.5 § 3.4, pp.218, 221, 222, 223, 225
- Probl. 3.6 – 3.12 § 3.5, pp.228, 229, 230, 231, 232, 233, 236

Bibliografía:

[BRY16] Cap.3

Computer Systems: A Programmer's Perspective 3rd ed. Bryant, O'Hallaron. Pearson, 2016

Signatura ESIIT/[C.1 BRY com](#)

Programación a Nivel-Máquina II: Control

Estructura de Computadores
Semana 4

Bibliografía:

[BRY16] Cap.3 Computer Systems: A Programmer's Perspective 3rd ed. Bryant, O'Hallaron. Pearson, 2016
 Signatura ESIIT/[C.1 BRY com](#)

Transparencias del libro CS:APP, Cap.3

Introduction to Computer Systems: a Programmer's Perspective

Autores: Randal E. Bryant y David R. O'Hallaron

<http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/schedule.html>

Guía de trabajo autónomo (4h/s)

■ Lectura: del Cap.3 CS:APP (Bryant/O'Hallaron)

- Control
 - § 3.6 pp.236-274

■ Ejercicios: del Cap.3 CS:APP (Bryant/O'Hallaron)

- Probl. 3.13 – 3.14 § 3.6.2, pp.240, 241
- Probl. 3.15 § 3.6.4, pp.245[†]
- Probl. 3.16 – 3.18 § 3.6.5, pp.248₂, 249
- Probl. 3.19 – 3.21 § 3.6.6, pp.252[‡], 255₂
- Probl. 3.22 – 3.29 § 3.6.7, pp.257, 258, 260, 262, 264, 267₂, 268
- Probl. 3.30 – 3.31 § 3.6.8, pp.272, 273

Bibliografía:

[BRY16] Cap.3

Computer Systems: A Programmer's Perspective 3rd ed. Bryant, O'Hallaron. Pearson, 2016

Signatura ESIIT/C.1 BRY com

[†] direccionamiento relativo a contador de programa, “PC-relative”

[‡] penalización por predicción saltos 2

Programación Máquina II: Control

- **Control: Códigos de condición**
- **Saltos condicionales**
- **Bucles**
- **Sentencias switch**

Estado del Procesador (x86-64, Parcial)

■ Información sobre el programa ejecutándose actualmente

- Datos temporales (`%rax`, ...)
- Situación de la pila en tiempo de ejecución[†] (`%rsp`)
- Situación actual del contador de programa (`%rip`)
- Estado de comparaciones recientes (`CF, ZF, SF, OF`)

Tope de pila actual

Registros de propósito general

<code>%rax</code>	<code>%r8</code>
<code>%rbx</code>	<code>%r9</code>
<code>%rcx</code>	<code>%r10</code>
<code>%rdx</code>	<code>%r11</code>
<code>%rsi</code>	<code>%r12</code>
<code>%rdi</code>	<code>%r13</code>
<code>%rsp</code>	<code>%r14</code>
<code>%rbp</code>	<code>%r15</code>

`%rip` Puntero de instrucción[‡]

`CF` `ZF` `SF` `OF` Códigos de condición

[‡] “instruction pointer”, de ahí `%rip`

[†] “runtime stack”

Códigos de Condición (su ajuste implícito)

■ Registros de un solo bit[†]

- Ajustados implícitamente por las operaciones aritméticas
(interpretarlo como efecto colateral)

Ejemplo: $\text{addq } Src, Dest \leftrightarrow t = a+b$

CF puesto a 1 si sale acarreo del bit más significativo (desbord. op. sin signo)

ZF a 1 sii t == 0

SF a 1 sii $t < 0$ (como número con signo)

OF a 1 sii desbord. en complemento a dos (desbord. op. con signo)
$$(a>0 \&\& b>0 \&\& t<0) \mid\mid (a<0 \&\& b<0 \&\& t>=0)$$

■ No afectados por la instrucción `lea`

[†] “flag” = “bandera”, deberíamos traducir “flag” por “indicador”, pero se suele dejar así, “optimization flags” debería ser “modificador/comutador”, también se suele dejar así.

*# “overflow” = “desbordamiento”, y los otros
“carry/zero/sign” = “acarreo/cero/signo” 5*

Códigos de Condición (ajuste explícito: Compare)

■ Ajuste Explícito mediante la Instrucción Compare

- `cmpq Src2, Src1`
- `cmpq b,a` equivale a restar $a-b$ pero sin ajustar el destino

- **CF a 1** sii sale acarreo del MSB[†] (c_n) (hacer caso cuando comp. sin signo)
- **ZF a 1** sii $a == b$
- **SF a 1** sii $(a-b) < 0$ (como número con signo)
- **OF a 1** sii overflow[‡] en complemento a dos (atender si comp. con signo)
 $(a>0 \&\& b<0 \&\& (a-b)<0) \mid\mid (a<0 \&\& b>0 \&\& (a-b)>0)$
definición overflow OF = $(c_n \wedge c_{n-1})$ mientras que acarreo CF = c_n

[†] “MSB” = bit más significativo

[‡] dejar sin traducir “overflow” ayuda didácticamente a distinguirlo
del acarreo (desbordamiento sin signo) al recordar los flags OF/CF 6

Códigos de Condición (ajuste explícito: Test)

■ Ajuste Explícito mediante la Instrucción Test

- `testq Src2, Src1`
- `testq b,a` equivale a hacer `a&b` pero sin ajustar el destino

- **ZF a 1** si $(a \& b) == 0$
- **SF a 1** si $(a \& b) < 0$

- Ajusta los códigos de condición según el valor de *Src1* & *Src2*
- Útil cuando uno de los operandos es una máscara
- Para comprobar si un valor es 0, gcc usa `testq`, no `cmpq`

```
    cmpq $0, %rax  
    testq %rax, %rax
```

Consultando Códigos de Condición

■ Instrucciones SetCC Dest

- Ajustar el byte destino a 0/1 según el código de condición indicado con CC[†] (combinación de flags deseada)
- *Dst registro* debe ser tamaño byte, *Dst memoria* sólo se modifica 1^{er} LSByte[†]

SetCC	Condición	Descripción
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Sign (negativo)
setns	~SF	Not Sign
setg	~(SF^OF)&~ZF	Greater (signo)
setge	~(SF^OF)	Greater or Equal (signo)
setl	(SF^OF)	Less (signo)
setle	(SF^OF) ZF	Less or Equal (signo)
seta	~CF&~ZF	Above (sin signo)
setb	CF	Below (sin signo)

"CC" = "condition code"

† "LSByte" = byte menos significativo, 8

Registros enteros x86-64

%rax	%al	%r8	%r8b
%rbx	%bl	%r9	%r9b
%rcx	%cl	%r10	%r10b
%rdx	%dl	%r11	%r11b
%rsi	%sil	%r12	%r12b
%rdi	%dil	%r13	%r13b
%rsp	%spl	%r14	%r14b
%rbp	%bpl	%r15	%r15b

- Se puede referenciar el LSByte, tiene nombre de registro propio

Consultando Códigos de Condición (Cont.)

■ Instrucciones SetCC Dest

- Ajustar un byte suelto *Dest* según el código de condición

■ Uno de los registros byte direccionables

- No se alteran los restantes bytes
- Típicamente se usa `movzbl†` para terminar trabajo
 - las instrucciones de 32-bit también ponen los 32 MSB a 0

```
int gt (long x, long y)
{
    return x > y;
}
```

Registro	Uso(s)
%rdi	Argumento x
%rsi	Argumento y
%eax	Valor de retorno

```
gt:
    cmpq    %rsi, %rdi  # Comparar x:y
    setg    %al           # Poner a 1 si >
†   movzbl  %al, %eax # Resto %rax a cero
    ret
```

[†] "Move with Zero-extend Byte to Long" mnemotécnico MOVZX según Intel 10

Consultando Códigos de Condición (Cont.)

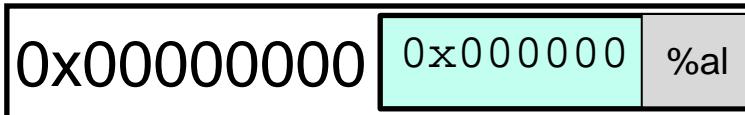
■ Instrucciones SetCC Dest

- A Las operaciones que modifican un registro de 32 bits, ponen a 0 el resto hasta 64 bits
- Un
- N
- T

movzbl %al, %eax

gt:

Puesto todo a 0



† **movzbl %al, %eax # Resto %rax a cero**
ret

+ "Move with Zero-extend Byte to Long"
mnemotécnico MOVZX según Intel 11

Programación Máquina II: Control

- Control: Códigos de condición
- Saltos condicionales
- Bucles
- Sentencias switch

Saltos

■ Instrucciones jCC

- Saltar a otro lugar del código si se cumple el código de condición CC

jCC	Condición	Descripción
jmp	1	Incondicional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Sign (negativo)
jns	~SF	Not Sign
jg	~(SF^OF)&~ZF	Greater (signo)
jge	~(SF^OF)	Greater or Equal (signo)
jl	(SF^OF)	Less (signo)
jle	(SF^OF) ZF	Less or Equal (signo)
ja	~CF&~ZF	Above (sin signo)
jb	CF	Below (sin signo)

Ejemplo de Salto Condicional (al viejo estilo)

■ Generación[†]

```
shark> gcc -Og -S -fno-if-conversion control.c
```

```
long absdiff(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

`absdiff:`

```

    cmpq    %rsi, %rdi  # x:y
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:                                # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret

```

Registro	Uso(s)
%rdi	Argumento x
%rsi	Argumento y
%rax	Valor de retorno

[†] en el gcc-7.3.0 que viene con Ubuntu-18.04

-fif-conversion activada para -O123s pero no para -Og 14

Expresándolo con código Goto

- C permite la sentencia goto
- Salta a la posición indicada por la etiqueta

```
long absdiff(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
Else:
    result = y-x;
Done:
    return result;
}
```

Traducción en General Expresión Condicional (usando saltos)

Código C

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x>y ? x-y : y-x;
```

Versión Goto

```
ntest = !Test;  
if (ntest) goto Else;  
val = Then_Expr;  
goto Done;  
Else:  
    val = Else_Expr;  
Done:  
    . . .
```

- Crear regiones de código separadas para las expresiones Then y Else
- Ejecutar sólo la adecuada

Usando Movimientos Condicionales

■ Instrucciones Mvmt. Condicional

- Las instrucciones implementan:
 $\text{if } (\text{Test}) \text{ Dest} \leftarrow \text{Src}$
- En procesadores x86 posteriores a 1995
(Pentium Pro/II)
- GCC intenta utilizarlas
 - Pero sólo cuando sepa que es seguro

■ ¿Por qué?

- Ramificaciones muy perjudiciales para flujo instrucciones en cauces[†]
- Movimiento condicional no requiere transferencia de control

Código C

```
val = Test
    ? Then_Expr
    : Else_Expr;
```

Versión Goto

```
result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
return result;
```

Ejemplo de Movimiento Condicional[†]

```
long absdiff(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

Registro	Uso(s)
%rdi	Argumento x
%rsi	Argumento y
%rax	Valor de retorno

absdiff:

movq	%rdi, %rax	# x
subq	%rsi, %rax	# result = x-y
movq	%rsi, %rdx	
subq	%rdi, %rdx	# eval = y-x
cmpq	%rsi, %rdi	# x:y
cmovele	%rdx, %rax	# if <=, result = eval
ret		

[†] generar con `gcc -fif-conversion -Og -S control.c`
 ó incluso con `gcc -O -S control.c`
 en el `gcc-7.3.0` que viene con `Ubuntu-18.04` 18

Malos Casos para Movimientos Condicionales

- Recordar que se calculan ambos valores

Cálculos costosos

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Sólo tiene sentido cuando son cálculos muy sencillos

Cálculos arriesgados

```
val = p ? *p : 0;
```

- Pueden tener efectos no deseables

Cálculos con efectos colaterales

```
val = x > 0 ? x*=7 : x+=3;
```

- No deberían tener efectos colaterales

Ejercicio

`cmpq b,a` equiv. restar $t=a-b$ pero sin ajustar destino

- **CF a 1** si $c_n == 1$, porque $a < b$ sin signo
- **ZF a 1** si $t == 0$, porque $a == b$
- **SF a 1** si $t_n == 1$, porque $a < b$ con signo
- **OF a 1** si $(c_n \wedge c_{n-1}) == 1$, pq. $a-b$ (signo) mal hecha

SetCC	Condición	Descripción
sete	ZF	Equal / Zero
setne	$\sim ZF$	Not Equal / Not Zero
sets	SF	Sign (negativo)
setns	$\sim SF$	Not Sign
setg	$\sim (SF \wedge OF) \wedge \sim ZF$	Greater (signo)
setge	$\sim (SF \wedge OF)$	Greater or Equal (signo)
setl	$(SF \wedge OF)$	Less (signo)
setle	$(SF \wedge OF) \mid ZF$	Less or Equal (signo)
seta	$\sim CF \wedge \sim ZF$	Above (sin signo)
setb	CF	Below (sin signo)

<code>xorq</code>	<code>%rax, %rax</code>
<code>subq</code>	<code>\$1, %rax</code>
<code>cmpq</code>	<code>\$2, %rax</code>
<code>setl</code>	<code>%al</code>
<code>movzbl</code>	<code>%al, %eax</code>

%rax	SF	CF	OF	ZF

Notar: `setl` y `movzblq` no ajustan códigos de condición

Ejercicio

`cmpq b,a` equiv. restar $t=a-b$ pero sin ajustar destino

- **CF a 1** si $c_n == 1$, porque $a < b$ sin signo
- **ZF a 1** si $t == 0$, porque $a == b$
- **SF a 1** si $t_n == 1$, porque $a < b$ con signo
- **OF a 1** si $(c_n \wedge c_{n-1}) == 1$, pq. $a-b$ (signo) mal hecha

SetCC	Condición	Descripción	
sete	ZF	Equal / Zero	
setne	$\sim ZF$	Not Equal / Not Zero	
sets	SF	Sign (negativo)	
setns	$\sim SF$	Not Sign	
setg	$\sim(SF \wedge OF) \& \sim ZF$	Greater	(signo)
setge	$\sim(SF \wedge OF)$	Greater or Equal	(signo)
setl	$(SF \wedge OF)$	Less	(signo)
setle	$(SF \wedge OF) \mid ZF$	Less or Equal	(signo)
seta	$\sim CF \& \sim ZF$	Above	(sin signo)
setb	CF	Below	(sin signo)

	<code>xorq %rax, %rax</code>	<code>subq \$1, %rax</code>	<code>cmpq \$2, %rax</code>	<code>setl %al</code>	<code>movzbl %al, %eax</code>

%rax	SF	CF	OF	ZF
0x0000 0000 0000 0000	0	0	0	1
0xFFFF FFFF FFFF FFFF	1	1	0	0
0xFFFF FFFF FFFF FFFF	1	0	0	0
0xFFFF FFFF FFFF FF01	1	0	0	0
0x0000 0000 0000 0001	1	0	0	0

Notar: `setl` y `movzbl` no ajustan códigos de condición

Programación Máquina II: Control

- Control: Códigos de condición
- Saltos condicionales
- Bucles
- Sentencias switch

Ejemplo de bucle “Do-While”

Código C

```
long pcount_do  
(unsigned long x) {  
    long result = 0;  
    do {  
        result += x & 0x1;  
        x >>= 1;  
    } while (x);  
    return result;  
}
```

Versión Goto

```
long pcount_goto  
(unsigned long x) {  
    long result = 0;  
loop:  
    result += x & 0x1;  
    x >>= 1;  
    if(x) goto loop;  
    return result;  
}
```

- Contar el número de 1's en el argumento x (“popcount” [†])
- Usar salto condicional para seguir iterando o salir del bucle

[†] “population count”= peso Hamming,
distancia Hamming (al 0), suma lateral... 23

Compilación del bucle “Do-While”

Versión Goto

```
long pcount_goto
(unsigned long x) {
    long result = 0;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    return result;
}
```

Registro	Uso(s)
%rdi	Argumento x
%rax	result

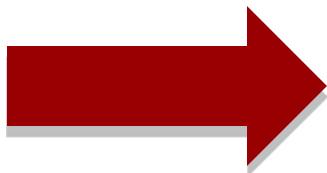
```
    movl    $0, %eax      # result = 0
.L2:                           # loop:
    movq    %rdi, %rdx
    andl    $1, %edx      # t = x & 0x1
    addq    %rdx, %rax    # result += t
    shrq    %rdi          # x >>= 1
    jne     .L2            # if (x) goto loop
    rep; ret
```

*+ problema predicción saltos Opteron y Athlon 64 (2003-2005) en RET 1B tras flowctrl.
Software Optimization Guide for AMD64 Family 10-12h (2010-2011) recomienda RET O*

Traducción en General de “Do-While”

Código C

```
do  
    Body  
    while ( Test );
```



Versión Goto

```
loop:  
    Body  
    if ( Test )  
        goto loop
```

■ **Body:** {
 Sentencia₁;
 Sentencia₂;
 ...
 Sentencia_n;
}

“body” = cuerpo,
“statement” = sentencia,
“test” = comprobación.

Traducción en General de “While” (#1)

Código C

```
while ( Test )  
    Body
```



Versión Goto

```
goto test;  
loop:  
    Body  
test:  
    if ( Test )  
        goto loop;  
done:
```

- Traducción tipo “**salta-en-medio**” †
- Usada con **-O0 / -Og**

Ejemplo de bucle “While” (#1)

Código C

```
long pcount_while  
  (unsigned long x) {  
    long result = 0;  
    while (x) {  
        result += x & 0x1;  
        x >>= 1;  
    }  
    return result;  
}
```

Salta-en-medio[†]

```
long pcount_goto_jtm  
  (unsigned long x) {  
    long result = 0;  
    goto test;  
loop:  
    result += x & 0x1;  
    x >>= 1;  
test:  
    if(x) goto loop;  
    return result;  
}
```

- Comparar con la versión do-while de la misma función
- El goto inicial empieza el bucle por **test** (“en medio”)

Traducción en General de “While” (#2)

Versión While

```
while ( Test )
    Body
```



Versión Do-While

```
if ( !Test )
    goto done;
do
    Body
    while( Test );
done:
```

- Traducción tipo “copia-test”
 - Conversión a “do-while”
- Usada con $-O1^t$

Versión Goto

```
if ( !Test )
    goto done;
loop:
    Body
    if ( Test )
        goto loop;
done:
```



^t evitar test al principio con gcc -O123 -fno-tree-ch

ó con gcc -Os -fno-reorder-blocks 28

Ejemplo de bucle “While” (#2)

Código C

```
long pcount_while
(unsigned long x) {
    long result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```

Copia-test

```
long pcount_goto_ct
(unsigned long x) {
    long result = 0;
    if (!x) goto done;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
done:
    return result;
}
```

- Comparar con la versión do-while de la misma función
- El primer condicional guarda la entrada al bucle

Forma del bucle “For”

Forma General

```
for (Init; Test; Update )  
    Body
```

```
#define WSIZE 8*sizeof(int)  
long pcount_for(unsigned long x)  
{  
    size_t i;  
    long result = 0;  
    for (i = 0; i < WSIZE; i++)  
    {  
        unsigned bit =  
            (x >> i) & 0x1;  
        result += bit;  
    }  
    return result;  
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{  
    unsigned bit =  
        (x >> i) & 0x1;  
    result += bit;  
}
```

“Init” = inicialización,
“test” = comprobación,
“update” = actualización,
“body” = cuerpo. 30

Bucle “For” → Bucle While

Versión For

```
for ( Init; Test; Update )  
    Body
```



Versión While

```
Init;  
  
while ( Test ) {  
    Body  
    Update;  
}
```

Conversión For-While

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{  
    unsigned bit =  
        (x >> i) & 0x1;  
    result += bit;  
}
```

```
long pcount_for_while  
(unsigned long x)  
{  
    size_t i;  
    long result = 0;  
    i = 0;  
    while (i < WSIZE)  
    {  
        unsigned bit =  
            (x >> i) & 0x1;  
        result += bit;  
        i++;  
    }  
    return result;  
}
```

Conversión Bucle “For” a Do-While

Código C

```
long pcount_for
(unsigned long x)
{
    size_t i;
    long result = 0;
    for (i = 0; i < WSIZE; i++)
    {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    return result;
}
```

Versión Goto

```
long pcount_for_goto_dw
(unsigned long x) {
    size_t i;
    long result = 0;
    i = 0;
if (! (i < WSIZE))
    goto done;
loop:
{
    unsigned bit =
        (x >> i) & 0x1;
    result += bit;
}
i++;
if (i < WSIZE)
    goto loop;
done:
return result;
}
```

Init

! Test

Body

Update

Test

- La comprobación inicial se puede optimizar (quitándola)

Programación Máquina II: Control

- Control: Códigos de condición
- Saltos condicionales
- Bucles
- Sentencias switch[†]

[†] “switch”=comutador,
tampoco traducimos
“for” o “while” 34

```
long switch_eg
    (long x, long y, long z)
{
    long w = 1;
    switch(x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
            /* Fall Through */
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

Ejemplo de sentencia switch

- Múltiples etiquetas de caso
 - Aquí: 5 y 6
- Caídas en cascada[†]
 - Aquí: 2
- Casos ausentes[†]
 - Aquí: 4

[†] “fall-through cases”, “missing cases” 35

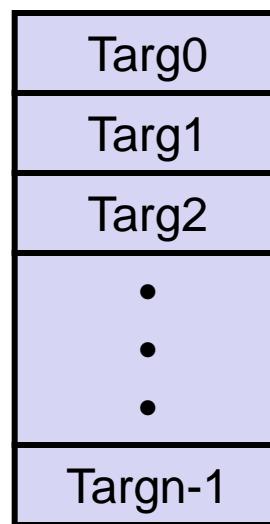
Estructura de una Tabla de Saltos

Forma switch

```
switch(x) {
    case val_0:
        Block 0
    case val_1:
        Block 1
    ...
    case val_n-1:
        Block n-1
}
```

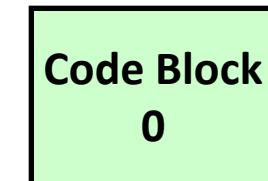
Tabla Saltos[†]

JTab:

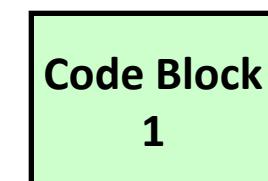


Destinos salto[†]

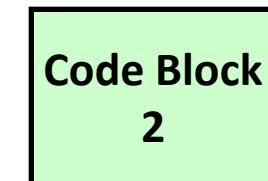
Targ0:



Targ1:

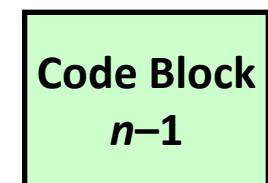


Targ2:



•
•
•

Targn-1:



Traducción aprox. (C ficticio)

```
goto *JTab[x];
```

[†] “jump table”, “jump targets” 36

Ejemplo de Sentencia Switch

```
long switch_eg
    (long x, long y, long z)
{
    long w = 1;
    switch(x) {
        . . .
    }
    return w;
}
```

Registro	Uso(s)
%rdi	Argumento x
%rsi	Argumento y
%rdx	Argumento z
%rax	Valor de retorno

Inicialización:

```
switch_eg:
    movq %rdx, %rcx      # z → %rcx
    cmpq $6, %rdi         # x:6
    † ja .L8              # default: ←
    jmp * .L4(,%rdi,8)   # goto *Jtab[x]
```

Notar que w no se inicializa aquí

¿Qué rango de valores cubre default?

Ejemplo de Sentencia Switch

```
long switch_eg
    (long x, long y, long z)
{
    long w = 1;
    switch(x) {
        . . .
    }
    return w;
}
```

Tabla de saltos[†]

.section	.rodata
.align 8	
.L4:	
.quad	.L8 # x = 0
.quad	.L3 # x = 1
.quad	.L5 # x = 2
.quad	.L9 # x = 3
.quad	.L8 # x = 4
.quad	.L7 # x = 5
.quad	.L7 # x = 6

Inicialización:

```
switch_eg:
    movq %rdx, %rcx      # z → %rcx
    cmpq $6, %rdi         # x:6
    † ja .L8              # default:
    jmp * .L4(,%rdi,8)   # goto *Jtab[x]
```

*Salto
indirecto*

Explicación Inicialización Ensamblador

■ Estructura de la Tabla

- Cada destino salto requiere 8 bytes
- Dirección base es .L4

■ Saltos

- **Directo:** `jmp .L8`
- Destino salto indicado por etiqueta .L8
- **Indirecto:** `jmp * .L4(,%rdi,8)`
- Inicio de la tabla de saltos: .L4
- Se debe escalar por un factor de 8 (direcciones ocupan 8 bytes)
- Captar destino salto desde la Dirección Efectiva $.L4 + x*8$
 - Sólo para $0 \leq x \leq 6$

Tabla de saltos

```
.section    .rodata
.align 8
.L4:
.quad      .L8    # x = 0
.quad      .L3    # x = 1
.quad      .L5    # x = 2
.quad      .L9    # x = 3
.quad      .L8    # x = 4
.quad      .L7    # x = 5
.quad      .L7    # x = 6
```

Tabla de Saltos

Tabla de saltos

```
.section .rodata
.align 8
.L4:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # x = 6
```

```
switch(x) {
    case 1:          // .L3
        w = y*z;
        break;
    case 2:          // .L5
        w = y/z;
        /* Fall Through */
    case 3:          // .L9
        w += z;
        break;
    case 5:
    case 6:          // .L7
        w -= z;
        break;
    default:         // .L8
        w = 2;
}
```

Bloques de Código ($x == 1$)

```
switch(x) {  
    case 1:      // .L3  
        w = y*z;  
        break;  
    . . .  
}
```

.L3:

```
    movq    %rsi, %rax  # y  
    imulq   %rdx, %rax  # y*z  
    ret
```

Registro	Uso(s)
%rdi	Argumento x
%rsi	Argumento y
%rdx	Argumento z
%rax	Valor de retorno

Tratamiento de Caídas en Cascada

```
long w = 1;  
.  
.  
switch(x) {  
.  
. . .  
case 2:  
    w = y/z;  
    /* Fall Through */  
case 3:  
    w += z;  
    break;  
.  
.  
}
```

case 2:
 w = y/z;
 goto merge;

case 3:
 w = 1;

merge:
 w += z;

Bloques de Código ($x == 2$, $x == 3$)

```

long w = 1;
    . . .
switch(x) {
    . . .
case 2:
    w = y/z;
    /* Fall Through */
case 3:
    w += z;
    break;
    . . .
}

```

```

.L5:                                # Case 2
    movq    %rsi, %rax
    † cqto
    idivq   %rcx      # y/z
    jmp     .L6        # goto merge
.L9:                                # Case 3
    movl    $1, %eax    # w = 1
.L6:                                # merge:
    addq    %rcx, %rax # w += z
    ret

```

Registro	Uso(s)
%rdi	Argumento x
%rsi	Argumento y
%rdx	Argumento z
%rax	Valor de retorno

mnemotécnico CQO según Intel

† “Convert Quad to Oct”

Bloques de Código ($x == 5$, $x == 6$, default)

```
switch(x) {  
    . . .  
    case 5: // .L7  
    case 6: // .L7  
        w -= z;  
        break;  
    default: // .L8  
        w = 2;  
}
```

```
.L7:                      # Case 5,6  
    movl $1, %eax      # w = 1  
    subq %rdx, %rax   # w -= z  
    ret  
.L8:                      # Default:  
    movl $2, %eax      # 2  
    ret
```

Registro	Uso(s)
%rdi	Argumento x
%rsi	Argumento y
%rdx	Argumento z
%rax	Valor de retorno

Resumen

■ Control C

- if-then-else
- do-while
- while, for
- switch

■ Control Ensamblador

- Salto condicional
- Movimiento condicional
- Salto indirecto (mediante tablas de saltos)
- Compilador genera secuencia código p/implementar control más complejo

■ Técnicas estándar

- Bucles convertidos a forma do-while (ó salta-en medio ó copia-test)
- Sentencias switch grandes usan tablas de saltos
- Sentencias switch poco densas → árboles decisión (if-elseif-elseif-else)

Guía de trabajo autónomo (4h/s)

■ **Estudio:** del Cap.3 CS:APP (Bryant/O'Hallaron)

- Control
 - § 3.6 pp.236-274

■ **Ejercicios:** del Cap.3 CS:APP (Bryant/O'Hallaron)

- Probl. 3.13 – 3.14 § 3.6.2, pp.240, 241
- Probl. 3.15 § 3.6.4, pp.245[†]
- Probl. 3.16 – 3.18 § 3.6.5, pp.248₂, 249
- Probl. 3.19 – 3.21 § 3.6.6, pp.252[‡], 255₂
- Probl. 3.22 – 3.29 § 3.6.7, pp.257, 258, 260, 262, 264, 267₂, 268
- Probl. 3.30 – 3.31 § 3.6.8, pp.272, 273

Bibliografía:

[BRY16] Cap.3

Computer Systems: A Programmer's Perspective 3rd ed. Bryant, O'Hallaron. Pearson, 2016

Signatura ESIIT/C.1 BRY com

[†] direccionamiento relativo a contador de programa, “PC-relative”

[‡] penalización por predicción saltos

Programación a Nivel-Máquina III: Procedimientos

Estructura de Computadores
Semana 5

Bibliografía:

[BRY16] Cap.3 Computer Systems: A Programmer's Perspective 3rd ed. Bryant, O'Hallaron. Pearson, 2016
 Signatura ESIIT/[C.1 BRY com](#)

Transparencias del libro CS:APP, Cap.3

Introduction to Computer Systems: a Programmer's Perspective

Autores: Randal E. Bryant y David R. O'Hallaron

<http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/schedule.html>

Guía de trabajo autónomo (4h/s)

■ Lectura: del Cap.3 CS:APP (Bryant/O'Hallaron)

- Procedures
 - § 3.7 pp.274-291
- Understanding Pointers, Using GDB.
 - § 3.10.1-2 pp.312-316

■ Ejercicios: del Cap.3 CS:APP (Bryant/O'Hallaron)

- Probl. 3.32 § 3.7.2, pp.280
- Probl. 3.33 § 3.7.3, pp.282
- Probl. 3.34 § 3.7.5, pp.288
- Probl. 3.35 § 3.7.6, pp.290

Bibliografía:

[BRY16] Cap.3

Computer Systems: A Programmer's Perspective 3rd ed. Bryant, O'Hallaron. Pearson, 2016

Signatura ESIIT/[C.1 BRY com](#)

Programación Máquina III: Procedimientos

■ Procedimientos

- Mecanismos
- Estructura de la pila
- Convenciones de llamada
 - Pasando el control
 - Pasando los datos
 - Gestionando datos locales
- Ejemplos ilustrativos de Recursividad

Mecanismos en los Procedimientos

■ Transferencia de control

- al principio del código del procedimiento
- de vuelta al punto de retorno

■ Transferencia de datos

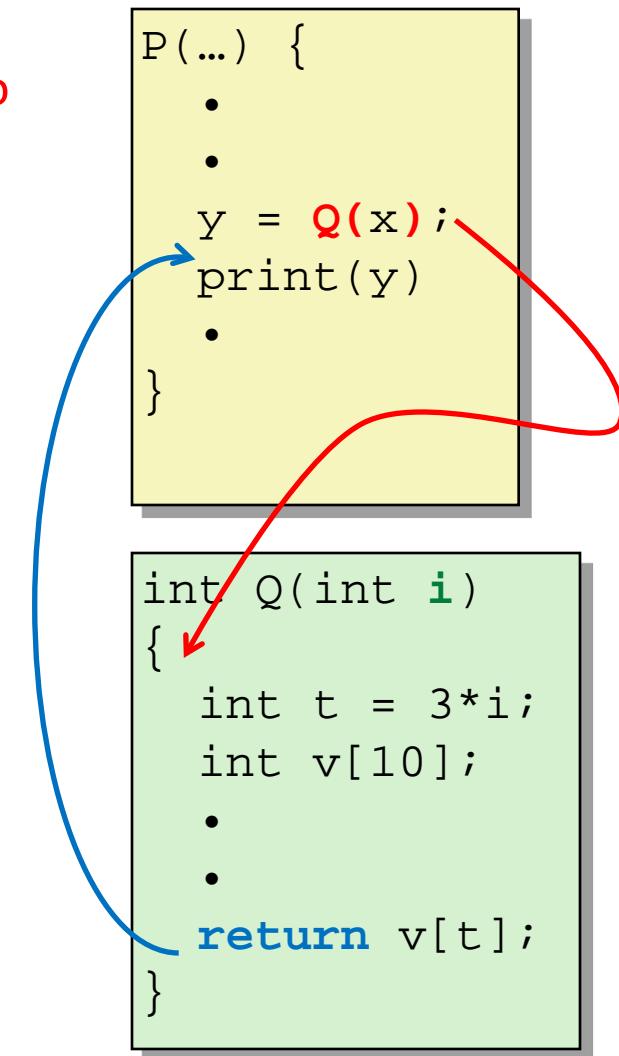
- Argumentos del procedimiento
- Valor de retorno

■ Gestión de memoria

- Reservar[†] durante ejecución procedimiento
- Liberar al retornar

■ Mecanismos todos implementados con instrucciones máquina

■ La implement. x86-64 de un proc. concreto usa sólo los mecanismos que éste requiera



Mecanismos en los Procedimientos

■ Transferencia de control

- al principio del código del procedimiento
- de vuelta al punto de retorno

■ Transferencia de datos

- Argumentos del procedimiento
- Valor de retorno

■ Gestión de memoria

- Reservar[†] durante ejecución procedimiento
- Liberar al retornar

■ Mecanismos todos implementados con instrucciones máquina

■ La implement. x86-64 de un proc. concreto usa sólo los mecanismos que éste requiera

```
P( ... ) {
    .
    .
    y = Q(x);
    print(y)
    .
}
```

```
int Q(int i)
{
    int t = 3*i;
    int v[10];
    .
    .
    return v[t];
}
```

[†] “allocate” = ubicar 5

Mecanismos en los Procedimientos

■ Transferencia de control

- al principio del código del procedimiento
- de vuelta al punto de retorno

■ Transferencia de datos

- Argumentos del procedimiento
- Valor de retorno

■ Gestión de memoria

- Reservar[†] durante ejecución procedimiento
- Liberar al retornar

■ Mecanismos todos implementados con instrucciones máquina

■ La implement. x86-64 de un proc. concreto usa sólo los mecanismos que éste requiera

```
P(...) {  
    •  
    •  
    y = Q(x);  
    print(y)  
    •  
}
```

```
int Q(int i)  
{  
    int t = 3*i;  
    int v[10];  
    •  
    •  
    return v[t];  
}
```

Mecanismos en los Procedimientos

■ Transferencia de control

- al principio del código del procedimiento
- de vuelta al punto de retorno

■ Transferencia de datos

- Argumentos del procedimiento
- Valor de retorno

■ Gestión de memoria

- Reservar[†] durante ejecución procedimiento
- Liberar al retornar

■ Mecanismos todos implementados con instrucciones máquina

■ La implement. x86-64 de un proc. concreto usa sólo los mecanismos que éste requiera

```
P(...) {  
    .  
    .  
    y = Q(x);  
    print(y)  
    .  
}
```

```
int Q(int i)  
{  
    int t = 3*i;  
    int v[10];  
    .  
    .  
    return v[t];  
}
```

Programación Máquina III: Procedimientos

■ Procedimientos

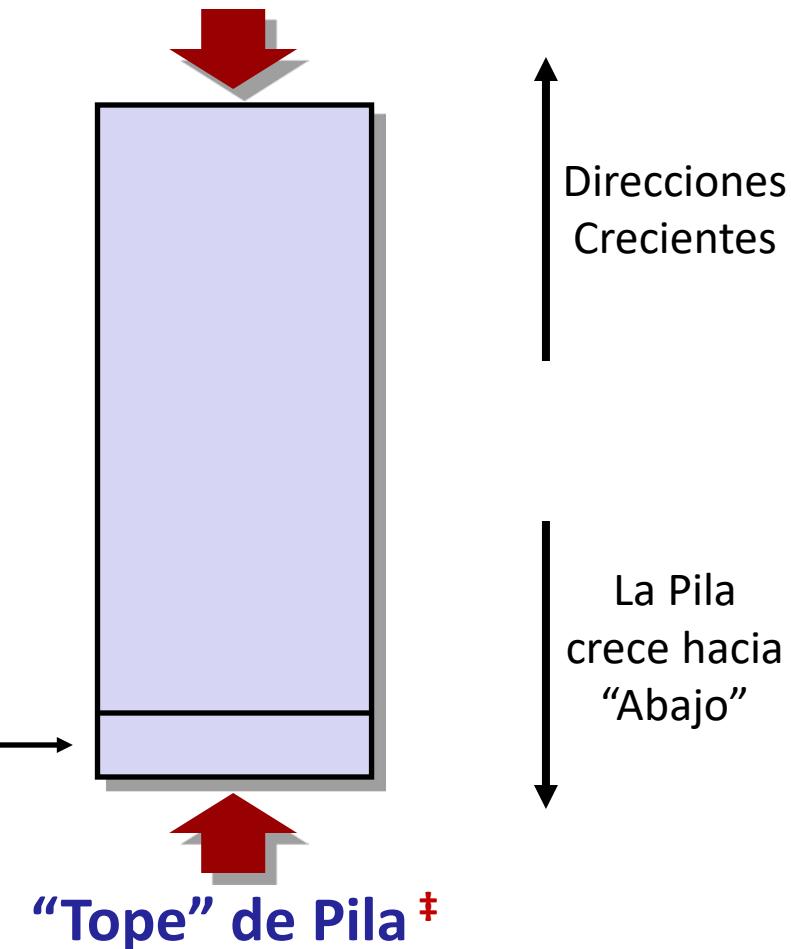
- Mecanismos
- Estructura de la pila
- Convenciones de llamada
 - Pasando el control
 - Pasando los datos
 - Gestionando datos locales
- Ejemplos ilustrativos de Recursividad

Pila x86-64

- Región de memoria gestionada con disciplina de pila
- Crece hacia posiciones inferiores
- El registro **%rsp** contiene la dirección más baja[†] de la pila
 - dirección del elemento “tope”

Puntero de Pila: **%rsp** →

“Fondo” de Pila[‡]



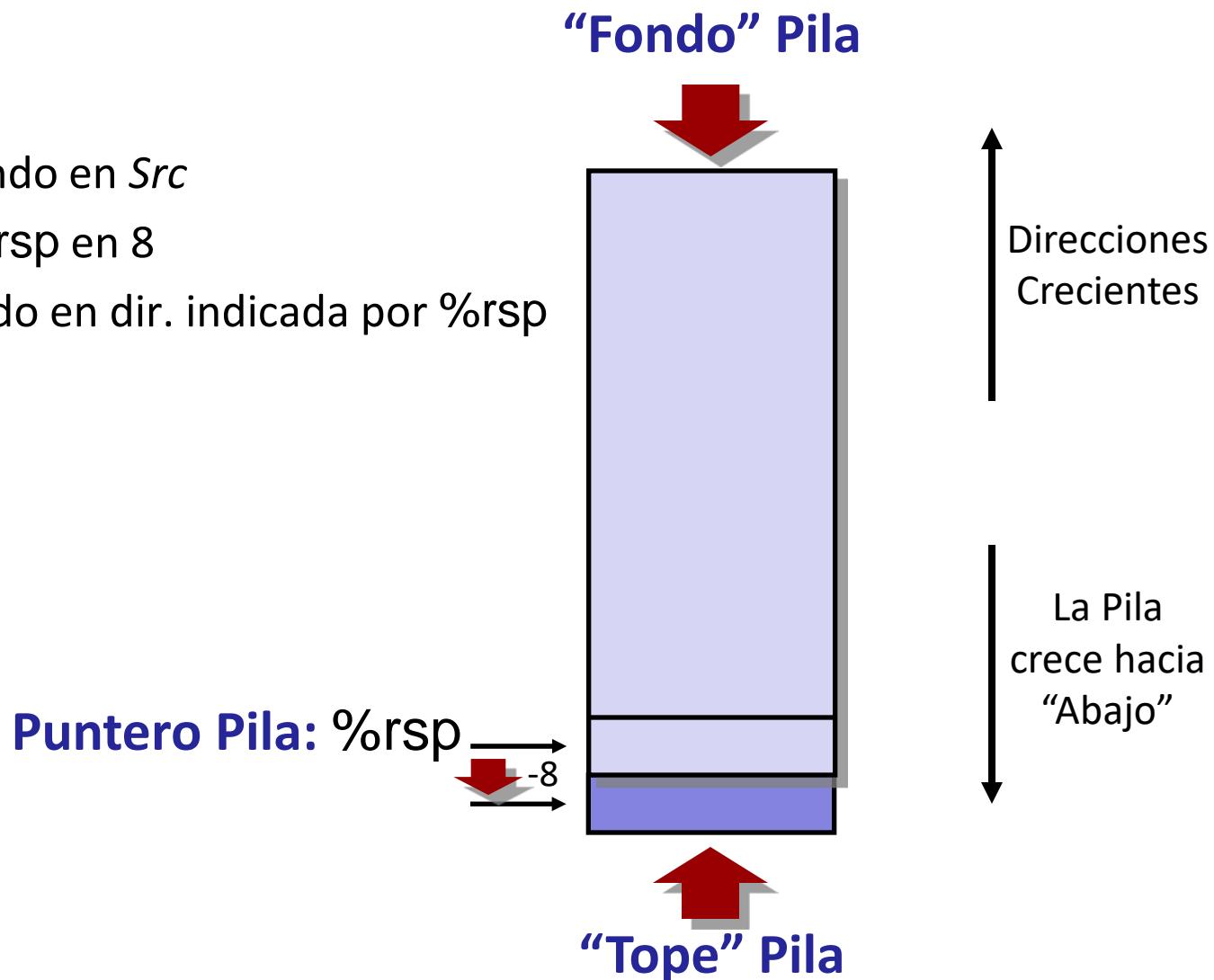
[†] “lowest” en el instante actual

[‡] “top” = tope, “bottom” = fondo 9

Pila x86-64: Push[†]

■ pushq Src

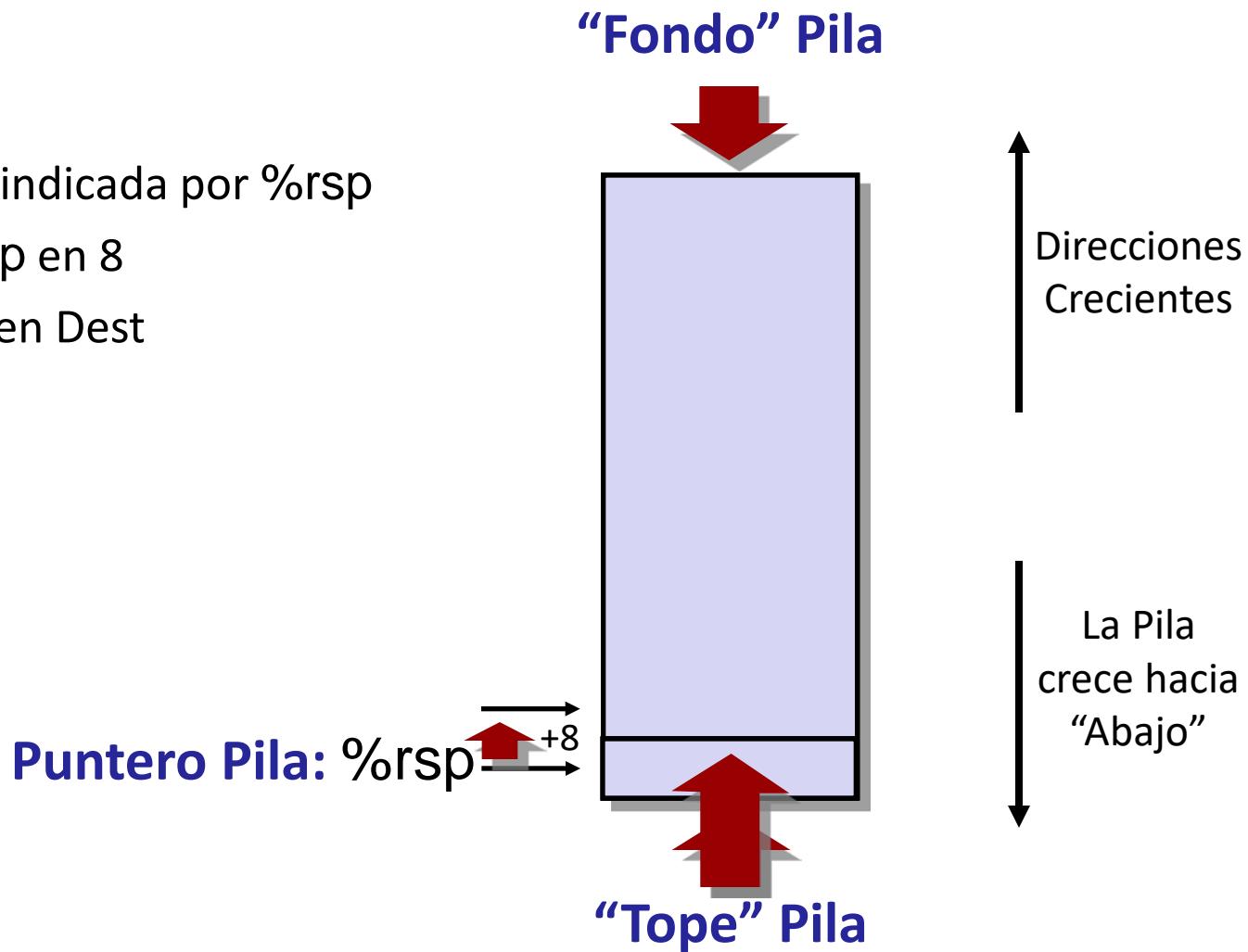
- Capta el operando en Src
- Decrementa %rsp en 8
- Escribe operando en dir. indicada por %rsp



Pila x86-64: Pop[†]

■ popq Dest

- Lee valor de dir. indicada por %rsp
- Incrementa %rsp en 8
- Almacena valor en Dest



Programación Máquina III: Procedimientos

■ Procedimientos

- Mecanismos
- Estructura de la pila
- Convenciones de llamada
 - **Pasando el control**
 - Pasando los datos
 - Gestionando datos locales
- Ejemplos ilustrativos de Recursividad

Código ejemplo

```
void multstore
    (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

0000000000400540 <multstore>:

400540: push %rbx	# preservar %rbx
400541: mov %rdx,%rbx	# conservar dest
400544: callq 400550 <mult2>	# mult2(x,y)
400549: mov %rax,(%rbx)	# salvar en dest
40054c: pop %rbx	# restaurar %rbx
40054d: retq	# retornar

```
long mult2
    (long a, long b)
{
    long s = a * b;
    return s;
}
```

0000000000400550 <mult2>:

400550: mov %rdi,%rax	# a
400553: imul %rsi,%rax	# a * b
400557: retq	# retornar

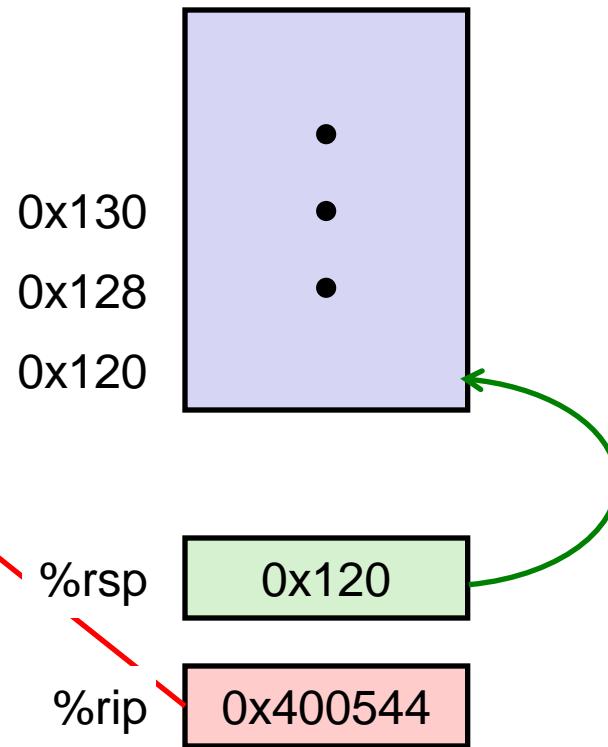
Flujo de Control en Procedimientos

- Usar la pila para soportar llamadas y retornos de procedimientos
- Llamada a procedimiento: **call label**
 - Recuerda[†] la dirección de retorno en la pila
 - Salta a etiqueta *label*
 - Codificada con *direcccionamiento relativo a IP*
- Dirección de retorno:
 - Dirección de la siguiente instrucción justo después de la llamada (call)
 - Ejemplo en el desensamblado anterior: **0x400549**
- Retorno de procedimiento: **ret**
 - Recupera[†] la dirección (de retorno) de la pila
 - Salta a dicha dirección

Ejemplo Flujo Control #1

```
0000000000400540 <multstore>:
    .
    .
    400544: callq  400550 <mult2>
400549: mov     %rax,(%rbx)
    .
    .
```

```
0000000000400550 <mult2>:
400550: mov     %rdi,%rax
    .
    .
    400557: retq
```

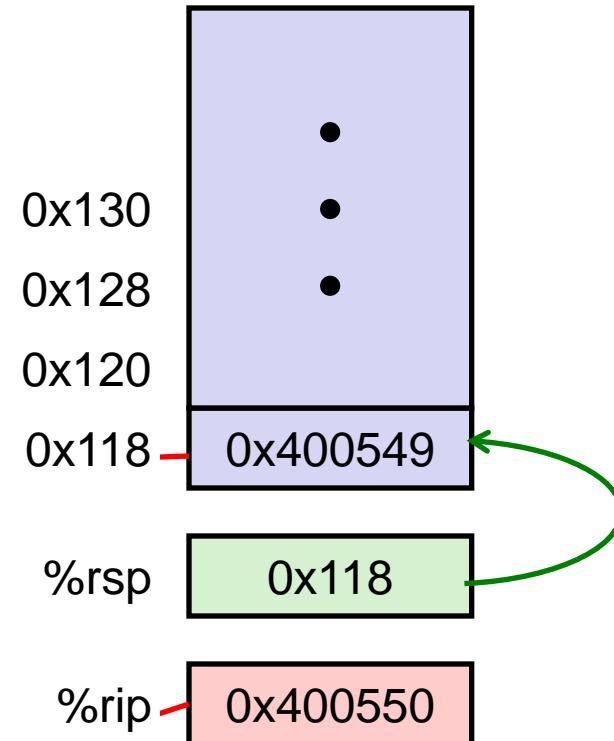


- **Direccionamiento relativo a contador de programa (RIP)**

$$\begin{array}{rcl}
 \text{RIP} & \textcolor{red}{0x00400549} & \text{(tras fetch)} \\
 +\text{offs} & \underline{0x00000007} \\
 =\text{Dst} & \textcolor{black}{0x00400550} \\
 \end{array}$$

Ejemplo Flujo Control #2

```
0000000000400540 <multstore>:
.
.
400544: callq  400550 <mult2>
400549: mov     %rax,(%rbx) ←
```

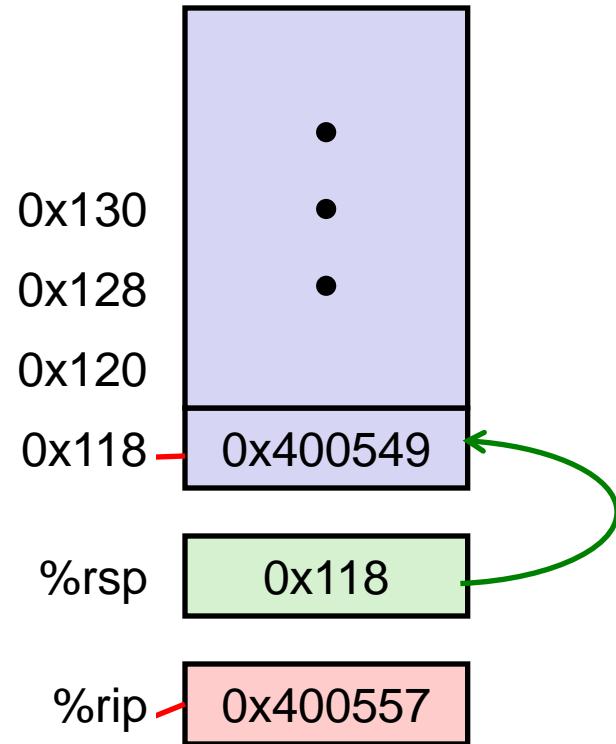


```
0000000000400550 <mult2>:
400550: mov     %rdi,%rax ←
.
.
400557: retq
```

400544: e8 07 00 00 00	callq 400550 <mult2>
400549: 48 89 03	mov %rax,(%rbx)
40054c: 5b	pop %rbx
40054d: c3	retq
40054e: 66 90	nop
0000000000400550 <mult2>:	
400550: 48 89 f8	mov %rdi,%rax

Ejemplo Flujo Control #3

```
0000000000400540 <multstore>:  
    .  
    .  
400544: callq  400550 <mult2>  
400549: mov     %rax,(%rbx) ←
```

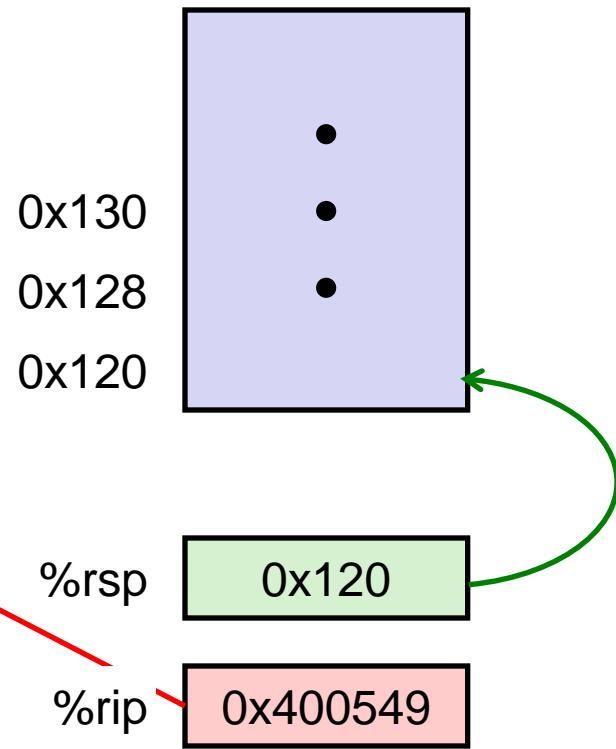


```
0000000000400550 <mult2>:  
400550: mov     %rdi,%rax  
    .  
    .  
400557: retq ←
```

Ejemplo Flujo Control #4

```
0000000000400540 <multstore>:  
    .  
    .  
    .  
    400544: callq  400550 <mult2>  
    400549: mov     %rax,(%rbx) ←
```

```
0000000000400550 <mult2>:  
    400550: mov     %rdi,%rax  
    .  
    .  
    400557: retq
```



Programación Máquina III: Procedimientos

■ Procedimientos

- Mecanismos
- Estructura de la pila
- Convenciones de llamada
 - Pasando el control
 - **Pasando los datos**
 - Gestionando datos locales
- Ejemplos ilustrativos de Recursividad

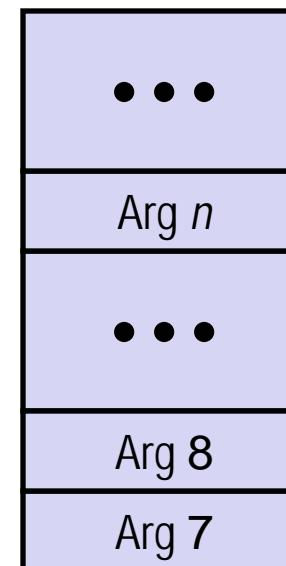
Flujo de Datos para Procedimientos

Registros

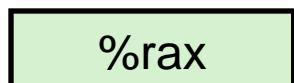
■ Primeros 6 argumentos



Pila



■ Valor de retorno



■ Sólo se reserva espacio en la pila cuando se necesita

Ejemplo

Flujo Datos

```
void multstore
    (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

0000000000400540 <multstore>:

x en %rdi, y en %rsi, dest en %rdx

```
400540: push    %rbx          # preservar %rbx
400541: mov     %rdx,%rbx      # conservar dest
400544: callq   400550 <mult2>  # mult2(x,y)
400549: mov     %rax,(%rbx)    # salvar en dest
40054c: pop     %rbx          # restaurar %rbx
# %rax libre (void), todavía conserva t=x*y
40054d: retq               # retornar
```

```
long mult2
    (long a, long b)
{
    long s = a * b;
    return s;
}
```

0000000000400550 <mult2>:

a en %rdi, b en %rsi

```
400550: mov     %rdi,%rax      # a
400553: imul   %rsi,%rax      # a * b
# s en %rax
400557: retq               # retornar
```

Programación Máquina III: Procedimientos

■ Procedimientos

- Mecanismos
- Estructura de la pila
- Convenciones de llamada
 - Pasando el control
 - Pasando los datos
 - **Gestionando datos locales**
- Ejemplos ilustrativos de Recursividad

Lenguajes basados en pila[†]

■ Lenguajes que soportan recursividad

- P.ej., C, Pascal, Java
- El código debe ser “*Reentrant*”
 - Múltiples instanciaciones[‡] simultáneas de un mismo procedimiento
- Se necesita algún lugar para guardar el estado de cada instanciaión
 - Argumentos
 - Variables locales
 - Puntero (dirección) de retorno

■ Disciplina de pila

- Estado para un procedimiento dado, necesario por tiempo limitado
 - Desde que se le llama hasta que retorna
- El invocado[‡] retorna antes de que lo haga el invocante[‡]

■ La pila se reserva en *Marcos*[‡]

- estado para una sola instanciaión de procedimiento

[†] “block structured” en terminología Intel

[‡] “callee/caller” en inglés

[#] “allocated in frames”

[‡] “instantiate” = crear nuevos ejemplares

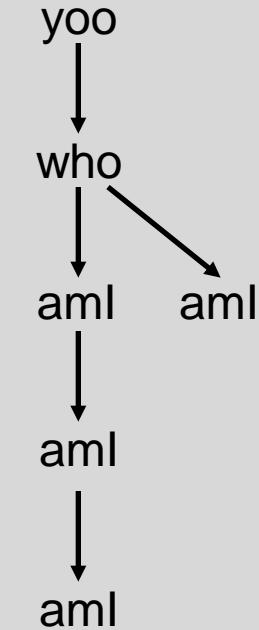
Ejemplo de secuencia[†] de llamadas

```
yoo( ... )
{
    .
    .
    who( ) ;
    .
    .
}
```

```
who( ... )
{
    .
    .
    amI( ) ;
    .
    .
    amI( ) ;
    .
    .
}
```

```
amI( ... )
{
    .
    .
    amI( ) ;
    .
    .
}
```

Ejemplo
Sec. Llamadas



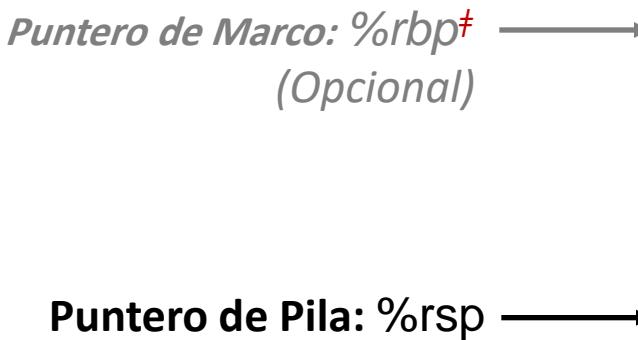
El procedimiento `amI()` es recursivo

[†] “call chain”.
“yoo/you” = tú,
“who” = quién,
“am I” = soy yo. 24

Marcos de Pila

■ Contenido

- Información de retorno
- Almacnmto[†] local (si necesario)
- Espacio temporal (si necesario)



■ Gestión

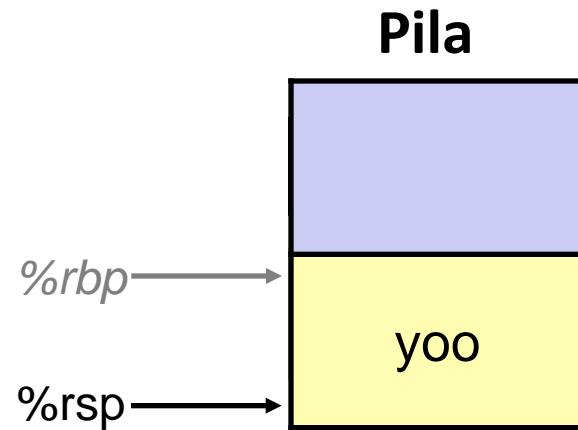
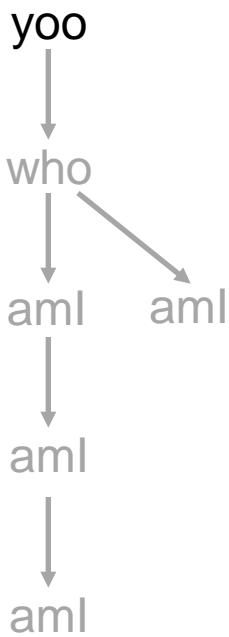
- Espacio se reserva al entrar el procedimiento
 - Código de “Inicialización” [‡]
 - Incluye el “push dir.ret.” de la instrucción **call**
- Se libera al retornar
 - Código de “Finalización” [‡]
 - Incluye el “pop cont.prog.” de la instrucción **ret**

“Tope” Pila

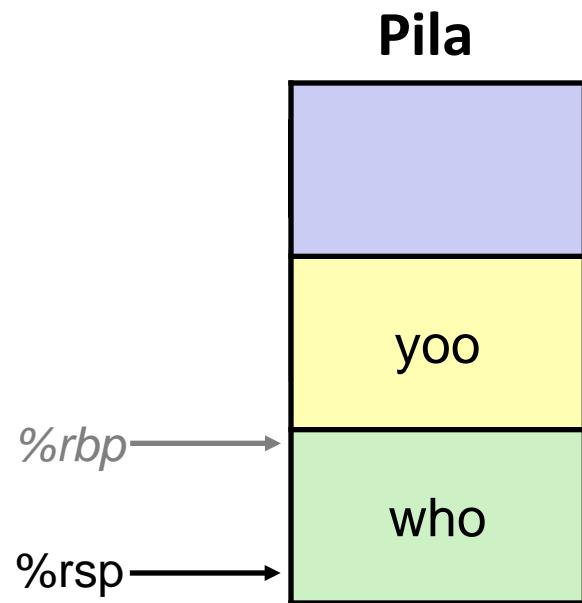
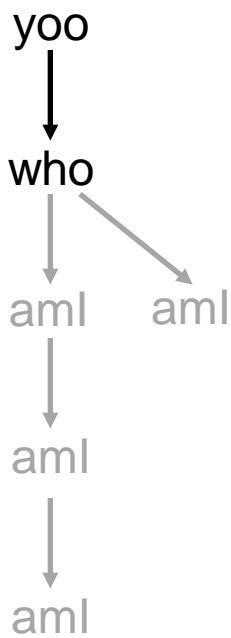
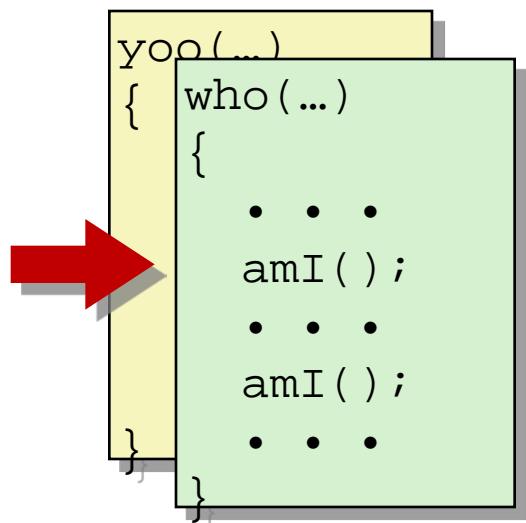
[#] si se usa *-fno-omit-frame-pointer*
[‡] “set-up/finish code”
† “local storage” 25

Ejemplo

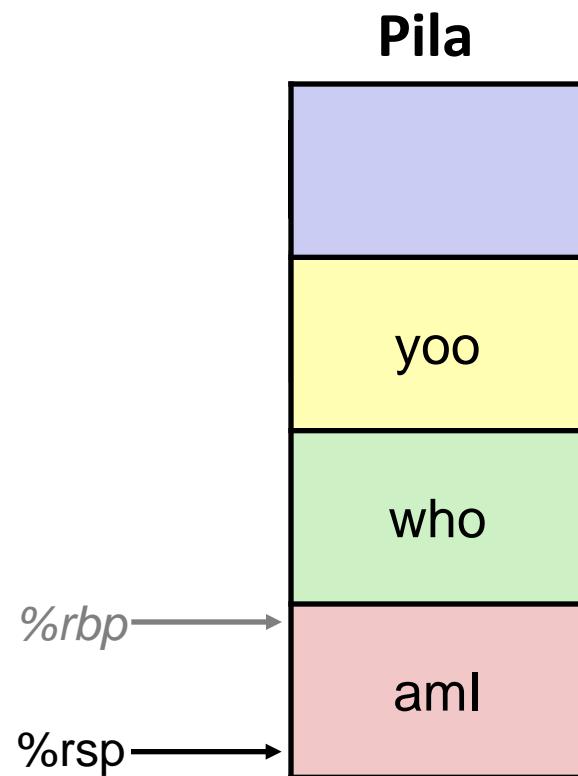
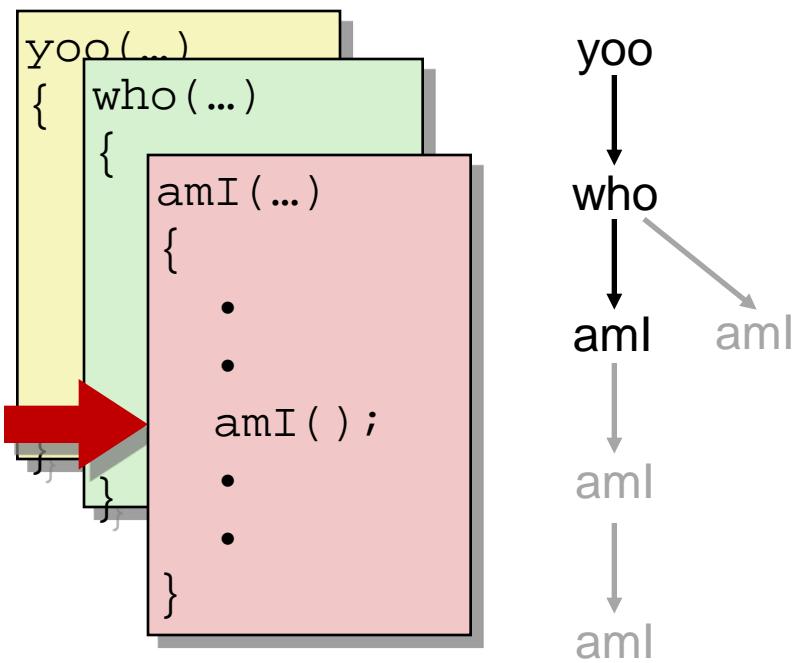
```
yoo( ... )  
{  
    •  
    •  
    who( ) ;  
    •  
    •  
}
```



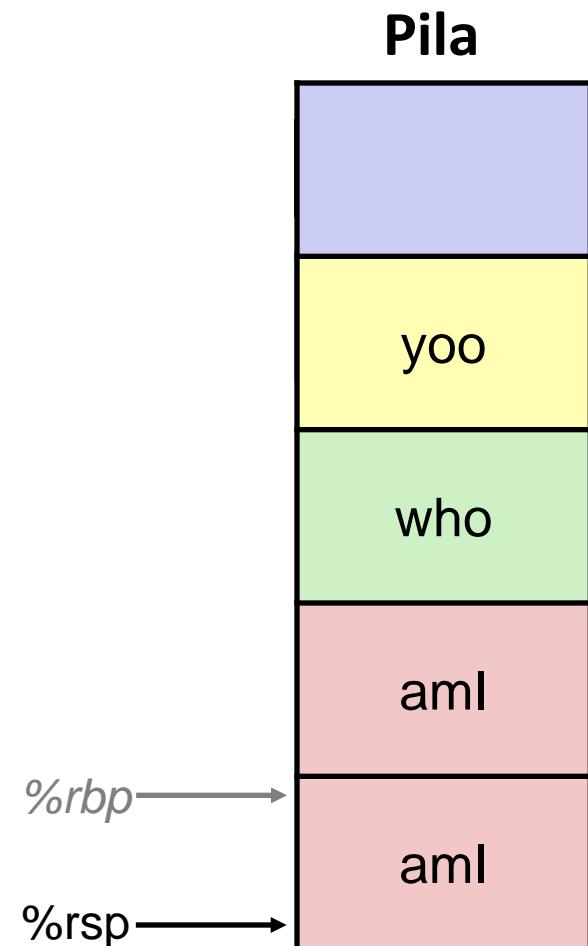
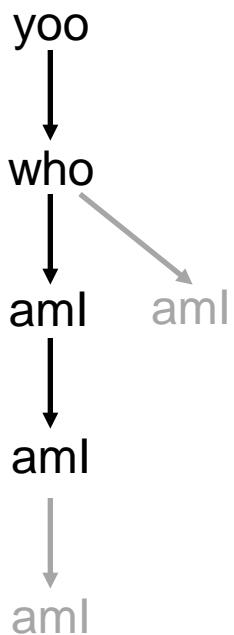
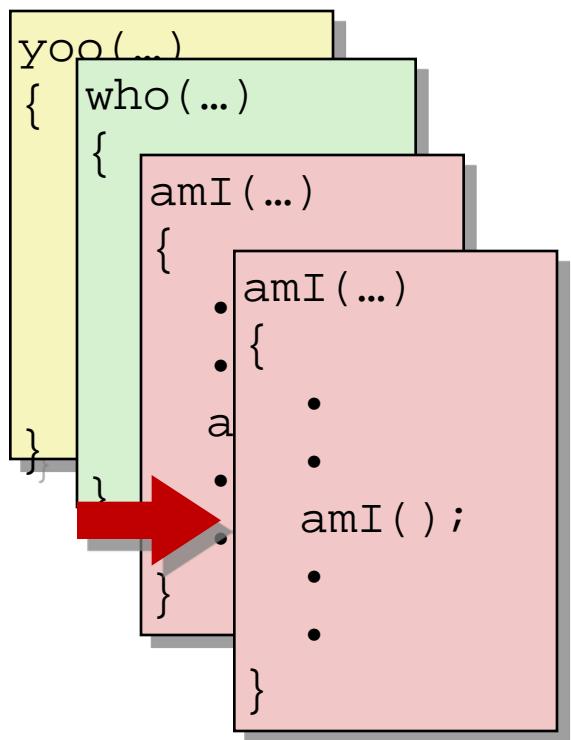
Ejemplo



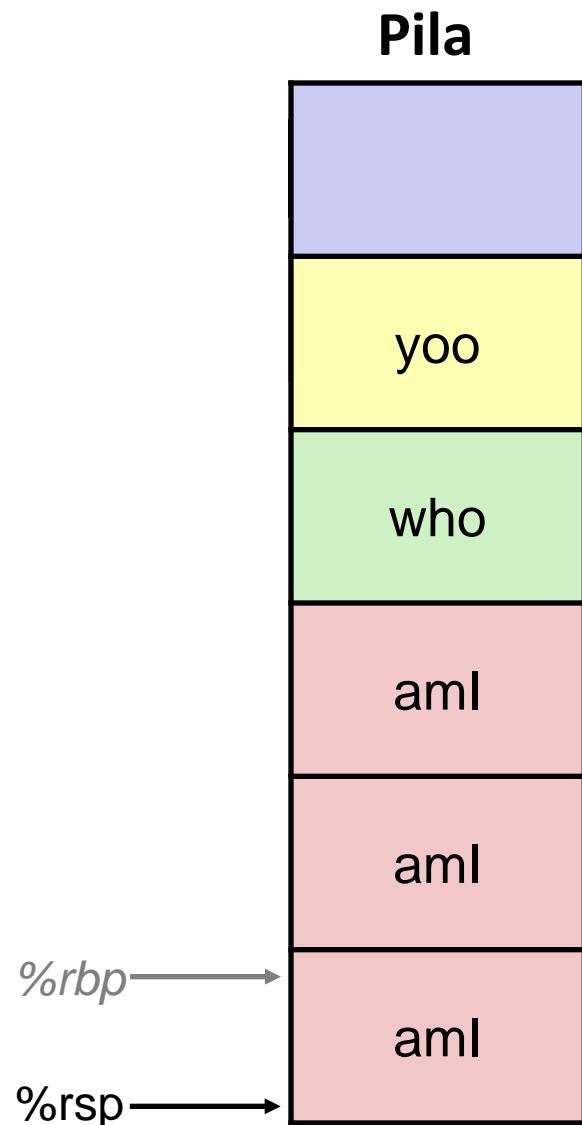
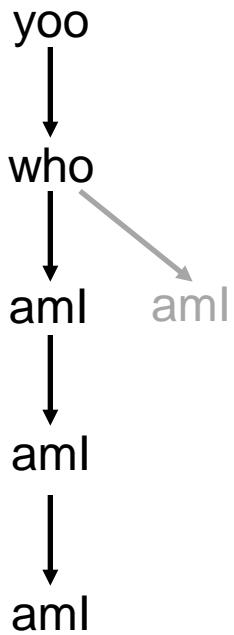
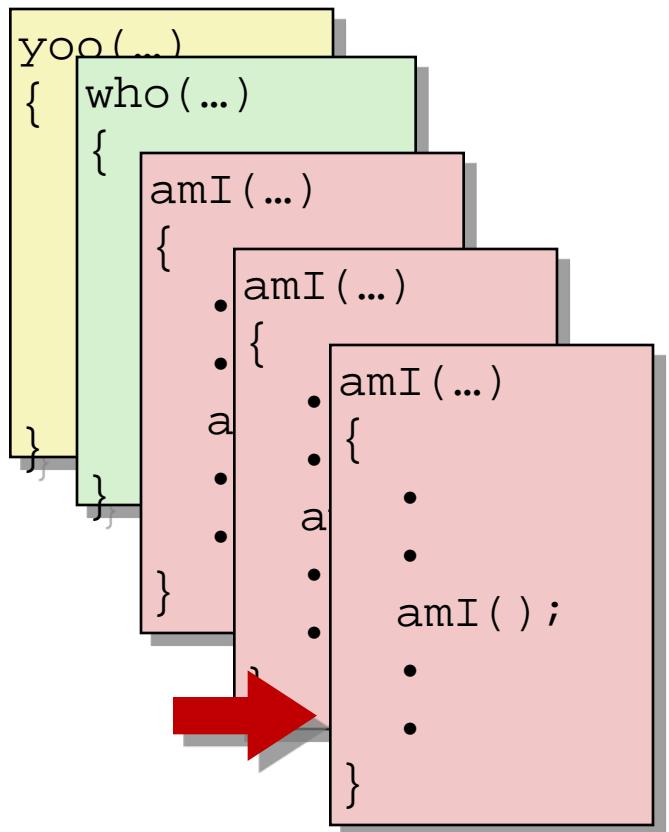
Ejemplo



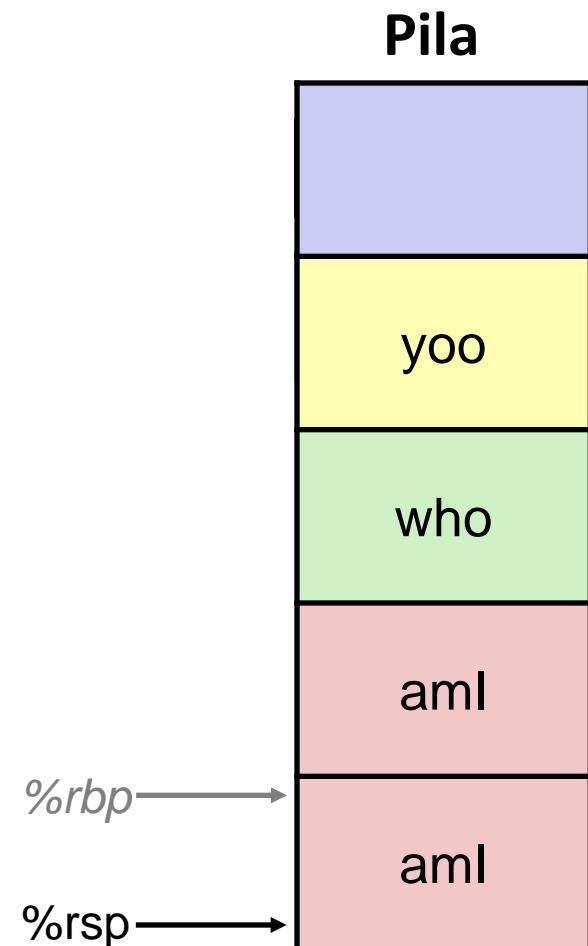
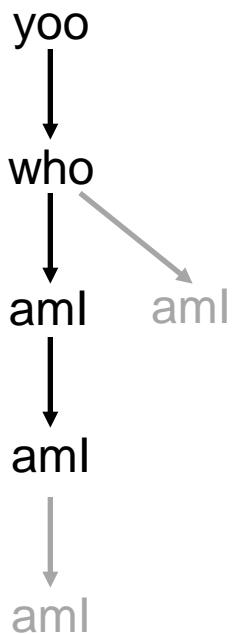
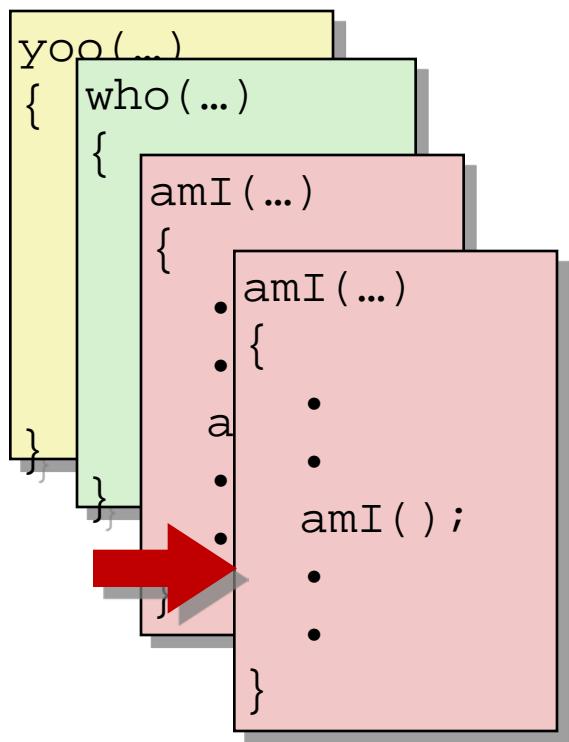
Ejemplo



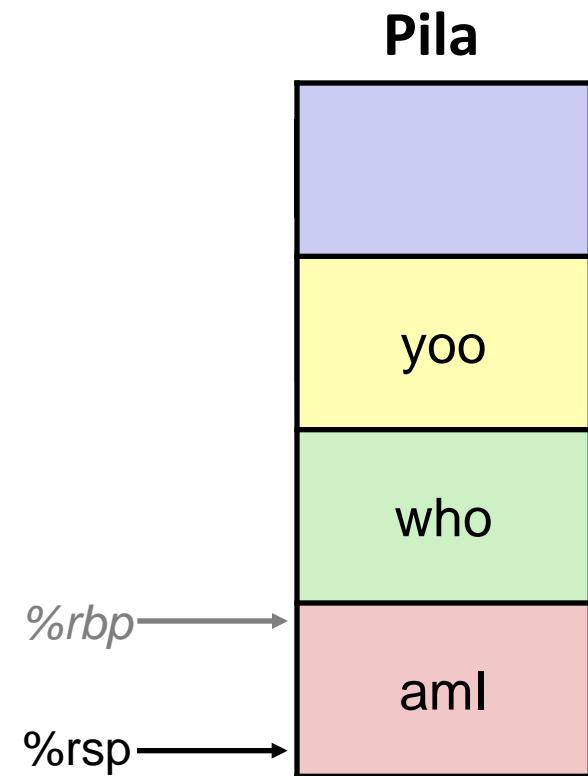
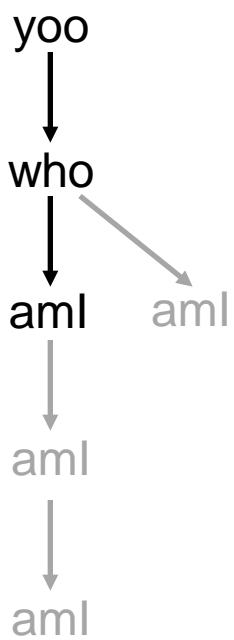
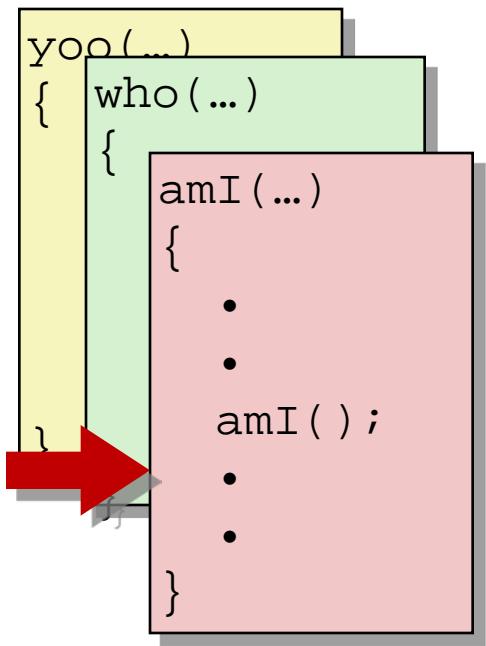
Ejemplo



Ejemplo

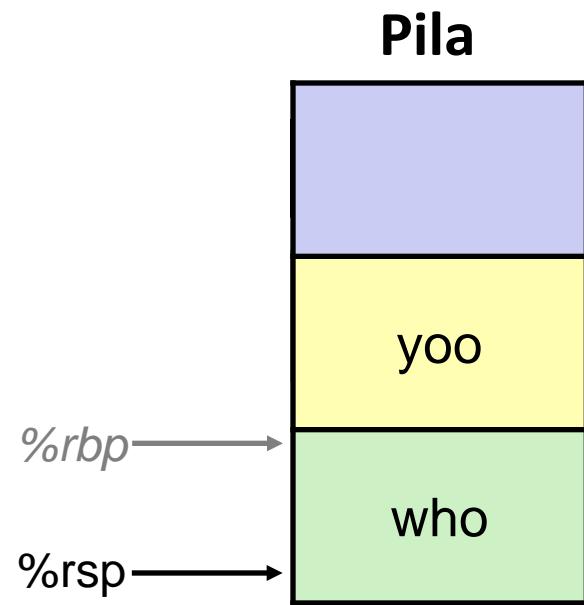
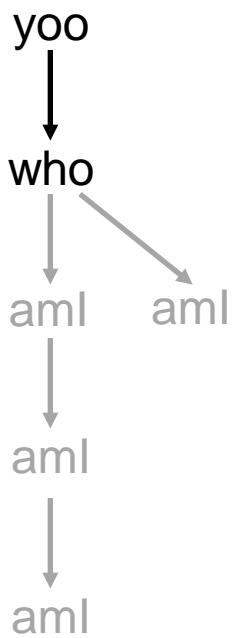


Ejemplo

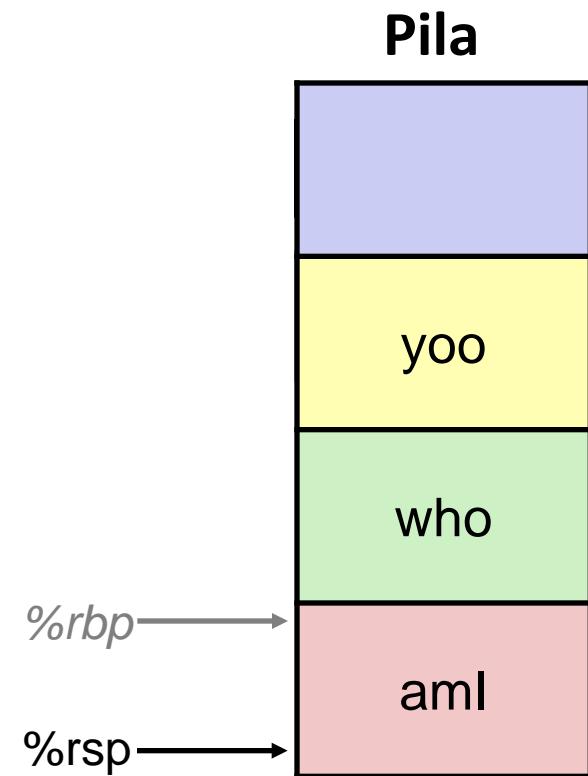
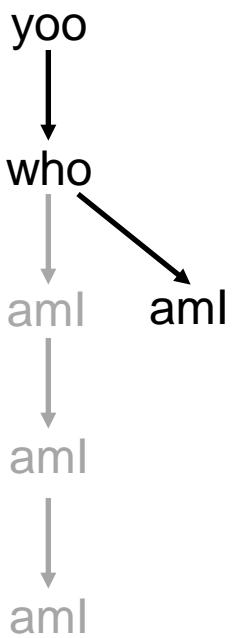
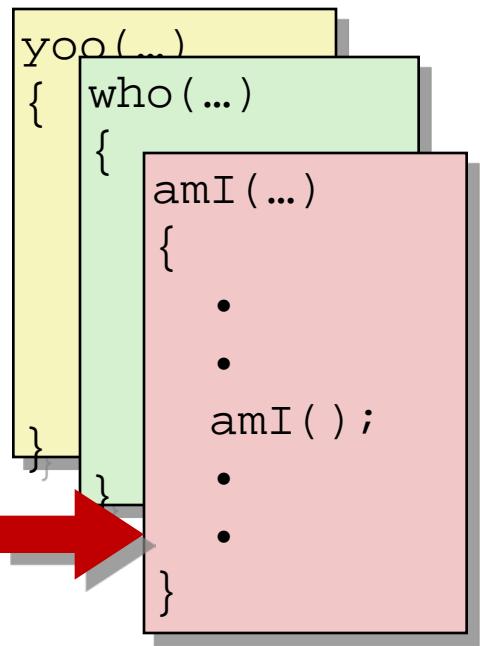


Ejemplo

```
yoo(...)  
{   who(...)  
{  
    . . .  
    amI();  
    . . .  
    amI();  
    . . .  
}
```

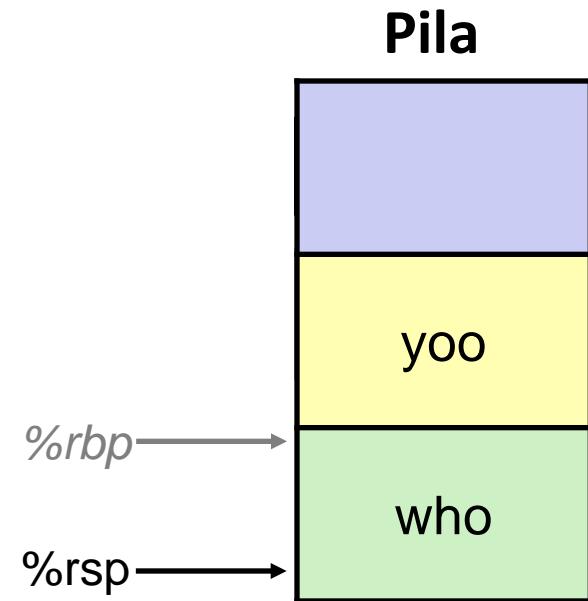
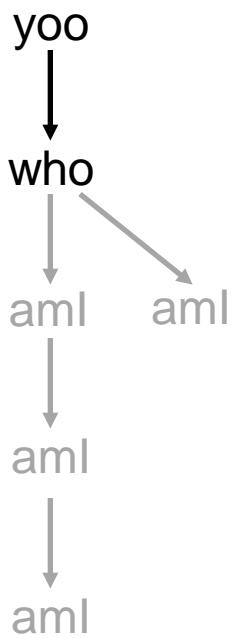


Ejemplo



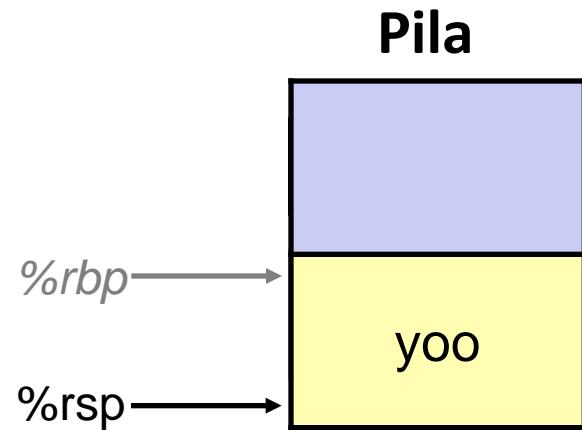
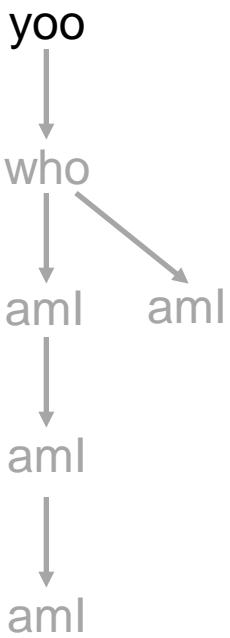
Ejemplo

```
yoo(...)  
{   who(...)  
{  
    . . .  
    amI();  
    . . .  
    amI();  
    . . .  
}
```



Ejemplo

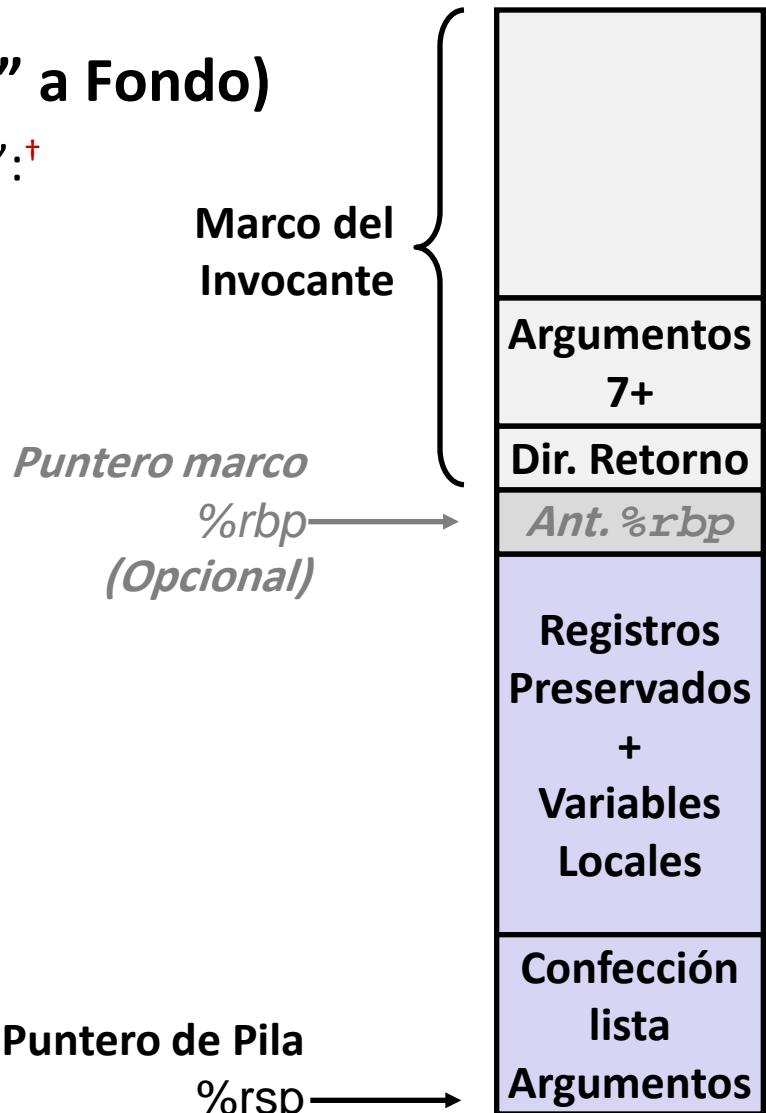
```
yoo( ... )  
{  
    •  
    •  
    who( ) ;  
    •  
    •  
}
```



Marco de Pila x86-64/Linux

■ Contenidos Marco Pila (de “Tope” a Fondo)

- “Confección de la lista de argumentos”:[†]
Parámetros (7+) función punto ser llamada
- Variables locales
Si no se pueden mantener en registros
- Contexto registros preservados
- *Antiguo puntero de marco (opcional)*
usando -fno-omit-frame-pointer (ó -O0)
 - *Dir.retorno pertenece marco anterior*



■ Marco de Pila del Invocante

- Dirección de retorno
 - Salvada por la instrucción call
- Argumentos (7+) para esta llamada

Puntero de Pila
%rsp →

[†] “Argument build” 37

Ejemplo: incr

```
long incr(long *p, long val) {  
    long x = *p;  
    long y = x + val;  
    *p = y;  
    return x;  
}
```

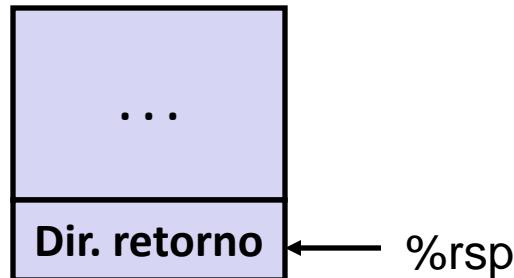
```
incr:  
    movq    (%rdi), %rax  
    addq    %rax, %rsi  
    movq    %rsi, (%rdi)  
    ret
```

Registro	Uso(s)
%rdi	Argumento p
%rsi	Argumento val , y
%rax	x , Valor de retorno

Ejemplo: llamar a incr #1

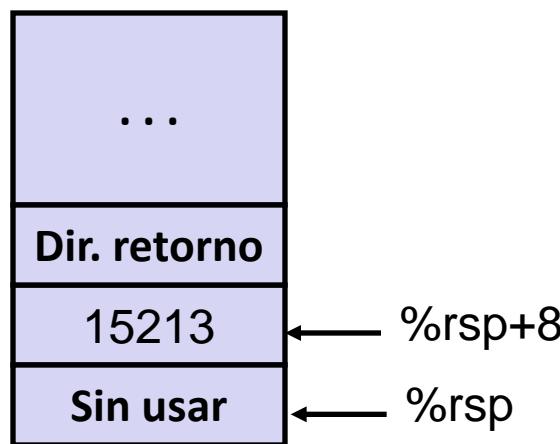
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Estructura de Pila inicial



```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Estructura de Pila resultante

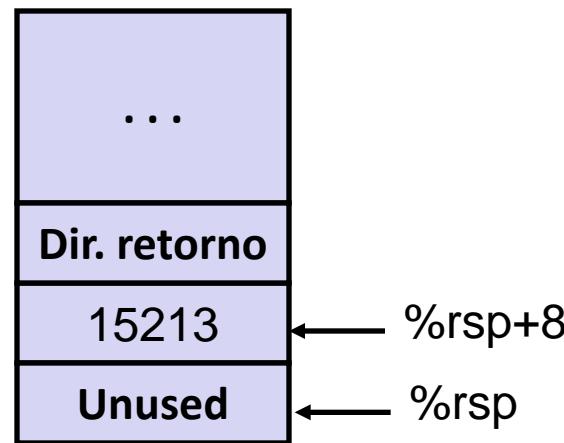


Ejemplo: llamar a incr #2

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Estructura de Pila



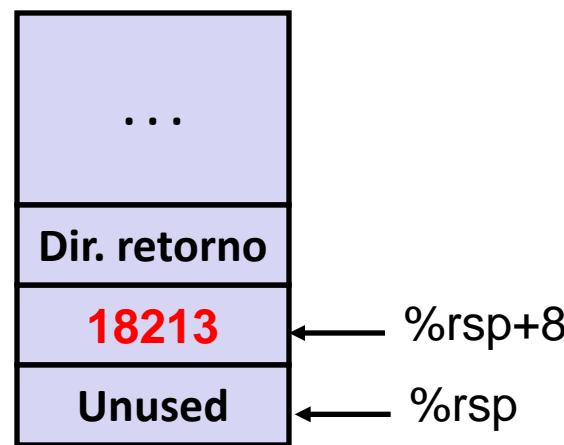
Registro	Uso(s)
<code>%rdi</code>	<code>&v1</code>
<code>%rsi</code>	3000

Ejemplo: llamar a incr #3

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Estructura de Pila

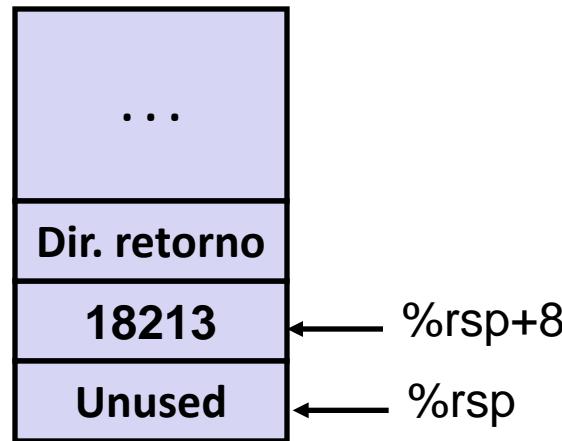


Registro	Uso(s)
%rdi	&v1
%rsi	3000

Ejemplo: llamar a incr #4

Estructura de Pila

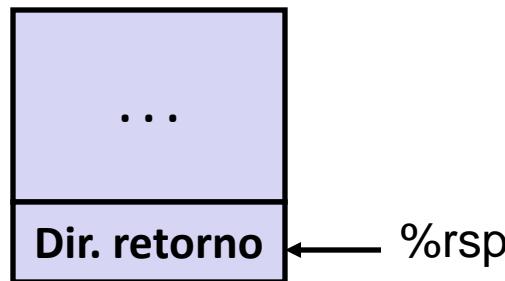
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```



```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Registro	Uso(s)
%rax	Valor de retorno

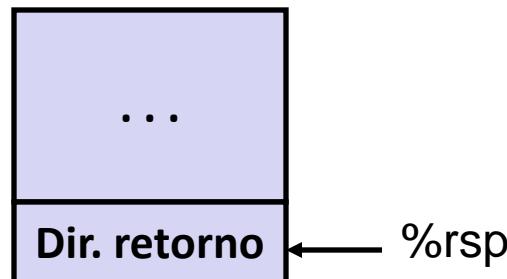
Estructura de Pila resultante



Ejemplo: llamar a incr #5

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

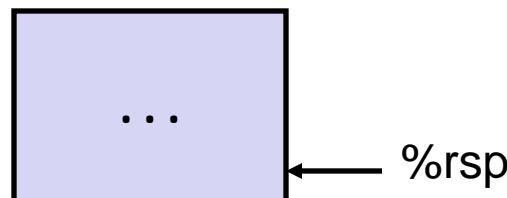
Estructura de Pila resultante



```
call_incr:
subq    $16, %rsp
movq    $15213, 8(%rsp)
movl    $3000, %esi
leaq    8(%rsp), %rdi
call    incr
addq    8(%rsp), %rax
addq    $16, %rsp
ret
```

Registro	Uso(s)
%rax	Valor de retorno

Estructura de Pila final



Convenciones de Preservación de Registros[†]

- Cuando el procedimiento yoo llama a who:
 - yoo es el **que llama (invocante, llamante)**
 - who es el **llamado (invocado)**
- ¿Se puede usar un registro para almacenamiento temporal?

```
yoo:
```

```
• • •  
    movq $15213, %rdx  
    call who  
    addq %rdx, %rax  
• • •  
    ret
```

```
who:
```

```
• • •  
    subq $18213, %rdx  
• • •  
    ret
```

- Contenidos del registro %edx sobrescritos por who
- Podría causar problemas → ¡debería hacerse algo!
 - Necesita alguna coordinación

Convenciones de Preservación de Registros

- Cuando el procedimiento *yoo* llama a *who*:
 - *yoo* es el *que llama (invocante, llamante)*
 - *who* es el *llamado (invocado)*
- ¿Se puede usar un registro para almacenamiento temporal?
- Convenciones[†]
 - “*Salva Invocante*”
 - El que llama salva valores temporales en su marco antes de la llamada
 - “*Salva Invocado*”
 - El llamado salva valores temporales en su marco antes de usar (regs.)
 - ...y los restaura antes de retornar al que llama

Uso de Registros en Linux x86-64[†] #1

■ %rax

- Valor de retorno
- También salva-invocante
- *Puede ser modificado[‡]* por el proc.

■ %rdi, ..., %r9

- Argumentos[‡]
- También salva-invocante
- Pueden ser modificados[‡] por proc.

■ %r10, %r11

- Salva-invocante
- Pueden ser modificados[‡] por proc.

Val.retorno

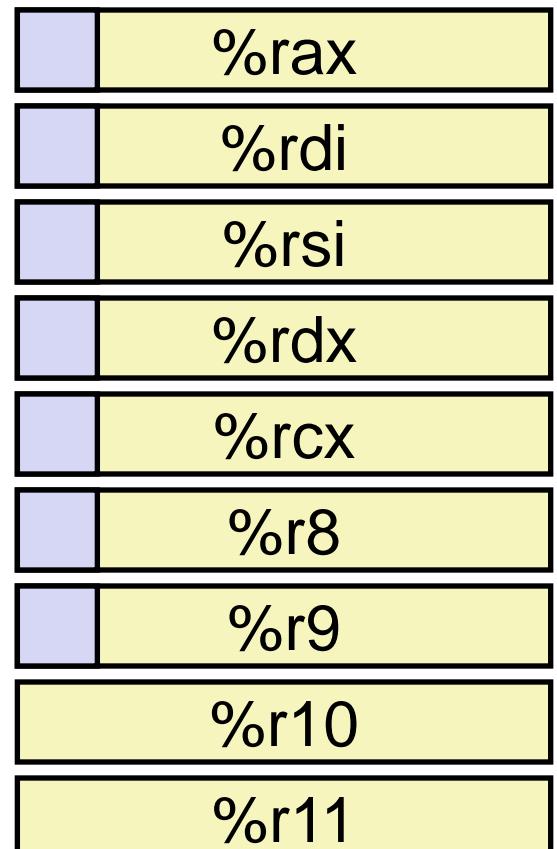
S-Invocante

Argumentos

S-Invocante

Temporales

S-Invocante



[†] Ver Wikipedia: "X86 calling conventions", Sys5 AMD64 ABI

[‡] regla mnemotécnica: invocado-cuidado, invocante-adelante

[‡] regla mnemotécnica: Diane's Silk Dress Costs \$89

Uso de Registros en Linux x86-64 #2

■ **%rbx, %r12 - %r15**

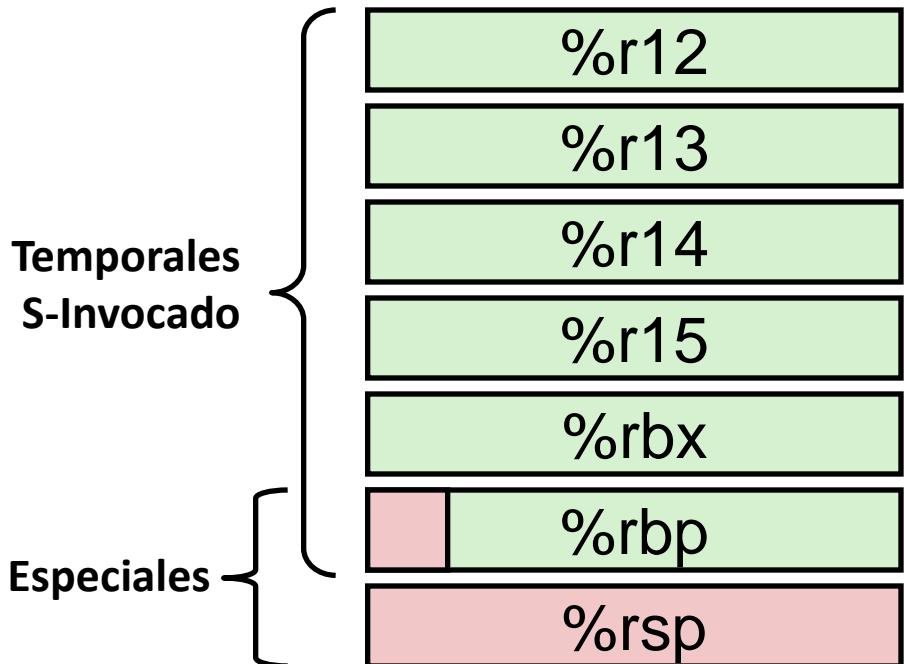
- Salva-invocado
- Invocado debe preservar y restaurar

■ **%rbp**

- Salva-invocado
- Invocado debe preservar y restaurar
- Puede que se use como marco pila
- Puede usarse intermezcladamente[†]

■ **%rsp**

- Forma especial de salva-invocado
- Restaurado a su valor original a la salida del procedimiento



[†] "mix & match", se refiere a que podrían linkarse procedimientos compilados con `gcc -fno-omit...` y `-fomit...` y no pasaría nada porque `%rbp` seguiría siendo s-invocado

Mini-Ejercicio

```

long add5(long b0, long b1, long b2, long b3, long b4) {
    return b0+b1+b2+b3+b4;
}

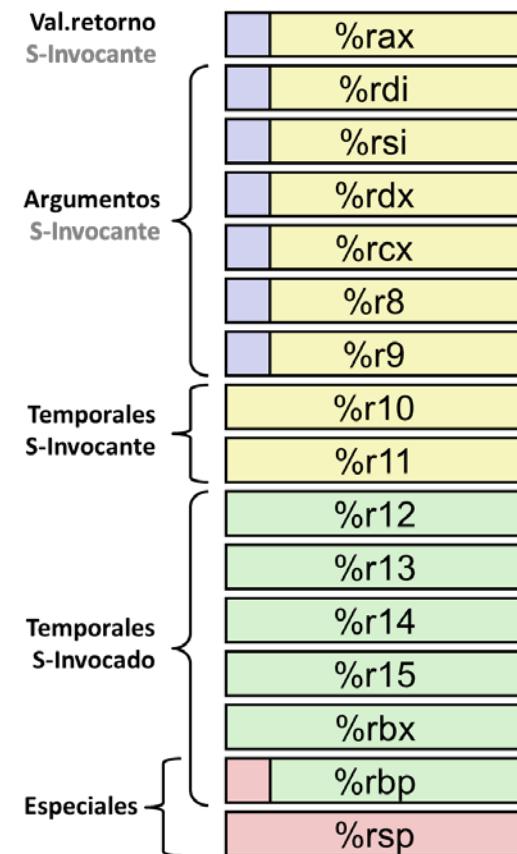
long add10(long a0, long a1, long a2, long a3, long a4,
           long a5, long a6, long a7, long a8, long a9) {
    return add5(a0, a1, a2, a3, a4) +
           add5(a5, a6, a7, a8, a9);
}

```

■ ¿Dónde se pasan **a0,..., a9?**
rdi, rsi, rdx, rcx, r8, r9, pila

■ ¿Dónde se pasan **b0,..., b4?**
rdi, rsi, rdx, rcx, r8

■ ¿Qué registros tenemos que preservar?
 Pregunta mal formulada. Requiere ver ASM.
rbx, rbp, r9 (durante 1ª llamada a **add5**)



Mini-Ejercicio

```
long add5(long b0, long b1, long b2, long b3, long b4) {
    return b0+b1+b2+b3+b4;
}

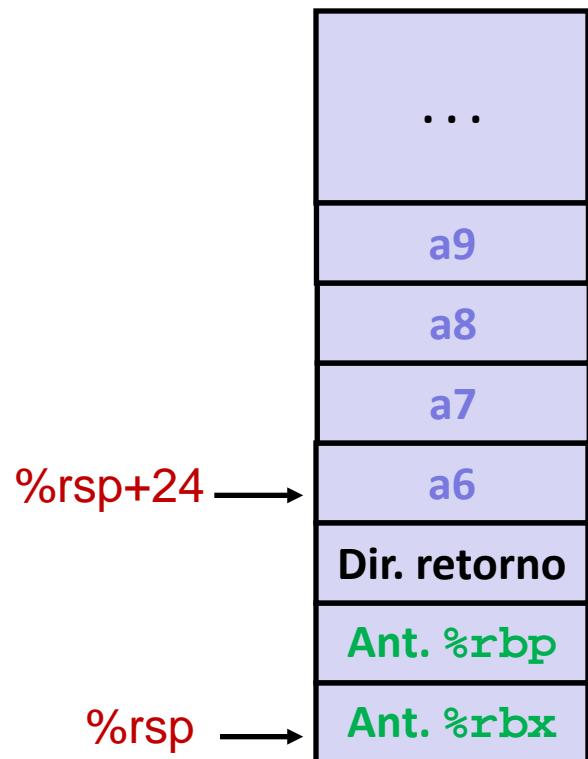
long add10(long a0, long a1, long a2, long a3, long a4,
           long a5, long a6, long a7, long a8, long a9) {
    return add5(a0, a1, a2, a3, a4)+  

           add5(a5, a6, a7, a8, a9);
}
```

```
add10:
    pushq  %rbp
    pushq  %rbx
    movq   %r9, %rbp
    call   add5
    movq   %rax, %rbx
    movq   48(%rsp), %r8
    movq   40(%rsp), %rcx
    movq   32(%rsp), %rdx
    movq   24(%rsp), %rsi
    movq   %rbp, %rdi
    call   add5
    addq   %rbx, %rax
    popq   %rbx
    popq   %rbp
    ret
```

```
add5:
    addq   %rsi, %rdi
    addq   %rdi, %rdx
    addq   %rdx, %rcx
    leaq   (%rcx,%r8), %rax
    ret
```

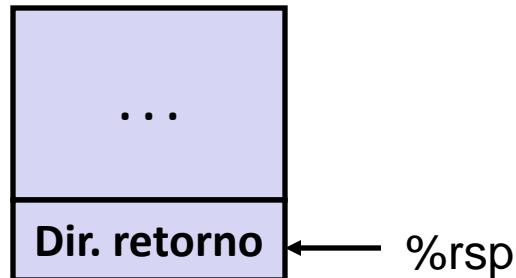
Pila durante la llamada a add10



Ejemplo Salva-invocado #1

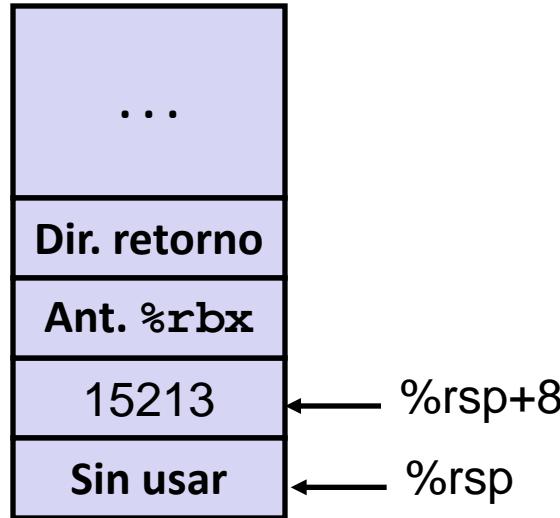
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

Estructura de Pila inicial



```
call_incr2:
pushq  %rbx
subq   $16, %rsp
movq   %rdi, %rbx
movq   $15213, 8(%rsp)
movl   $3000, %esi
leaq   8(%rsp), %rdi
call   incr
addq   %rbx, %rax
addq   $16, %rsp
popq   %rbx
ret
```

Estructura de Pila resultante

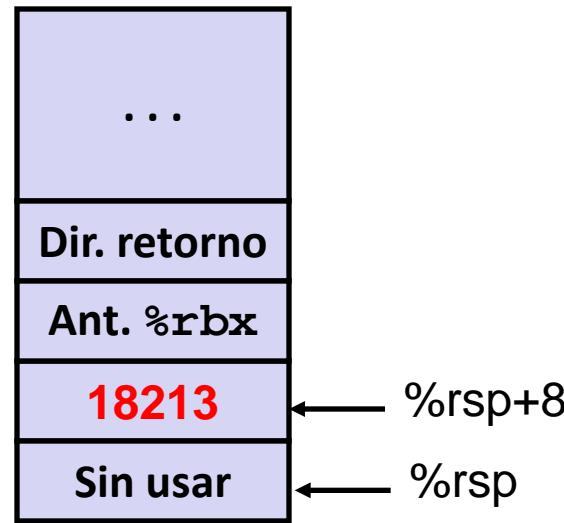


Ejemplo Salva-invocado #2

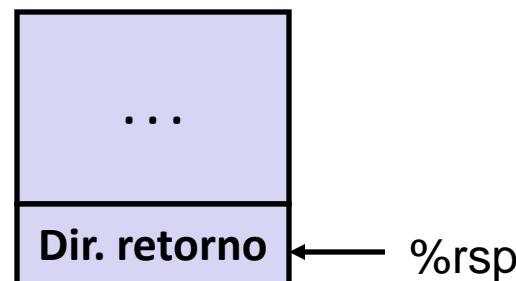
Estructura de Pila

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
    pushq %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $15213, 8(%rsp)
    movl $3000, %esi
    leaq 8(%rsp), %rdi
    call incr
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```



Estructura de Pila antes de ret



Programación Máquina III: Procedimientos

■ Procedimientos

- Mecanismos
- Estructura de la pila
- Convenciones de llamada
 - Pasando el control
 - Pasando los datos
 - Gestionando datos locales
- Ejemplos ilustrativos de Recursividad

Función Recursiva

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

pcount_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq   %rdi, %rbx
    andl   $1, %ebx
    shrq   %rdi
    call   pcount_r
    addq   %rbx, %rax
    popq   %rbx
```

.L6:

```
    rep; ret
```

Condición Terminación Función Recursiva

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

`pcount_r:`

<code>movl</code>	<code>\$0, %eax</code>
<code>testq</code>	<code>%rdi, %rdi</code>
<code>je</code>	<code>.L6</code>
<code>pushq</code>	<code>%rbx</code>
<code>movq</code>	<code>%rdi, %rbx</code>
<code>andl</code>	<code>\$1, %ebx</code>
<code>shrq</code>	<code>%rdi</code>
<code>call</code>	<code>pcount_r</code>
<code>addq</code>	<code>%rbx, %rax</code>
<code>popq</code>	<code>%rbx</code>

`.L6:`

`rep; ret`

Registro	Uso(s)	Tipo
<code>%rdi</code>	<code>x</code>	Salva-invocante 
<code>%rax</code>	Valor de retorno	Salva-invocante 

Preservar Registro para Llamada Recursiva

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

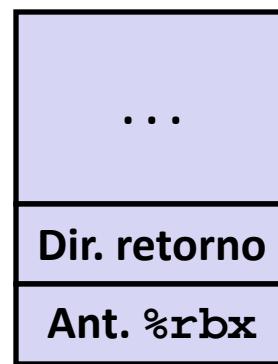
Registro	Uso(s)	Tipo
%rbx	x & 1	Salva-invocado

pcount_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq   %rdi, %rbx
    andl   $1, %ebx
    shrq   %rdi
    call   pcount_r
    addq   %rbx, %rax
    popq   %rbx
```

.L6:

rep; ret



Preparar Llamada Función Recursiva

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

`pcount_r:`

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq   %rdi, %rbx
    andl   $1, %ebx
    shrq   %rdi
    call   pcount_r
    addq   %rbx, %rax
    popq   %rbx
```

`.L6:`

```
rep; ret
```

Registro	Uso(s)	Tipo
<code>%rbx</code>	<code>x & 1</code>	Salva-invocado 
<code>%rdi</code>	<code>x >> 1</code> Argumento recurs.	Salva-invocante 

Llamada Función Recursiva

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

pcount_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
```

.L6:

rep; ret

Registro	Uso(s)	Tipo
%rbx	x & 1	Salva-invocado 
%rax	Valor de retorno de llamada recursiva	Salva-invocante 

Resultado Función Recursiva

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

pcount_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq   %rdi, %rbx
    andl   $1, %ebx
    shrq   %rdi
    call   pcount_r
    addq   %rbx, %rax
    popq   %rbx
```

.L6:

rep; ret

Registro	Uso(s)	Tipo
%rbx	x & 1	Salva-invocado 
%rax	Valor de retorno	Salva-invocante 

Terminar Función Recursiva

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

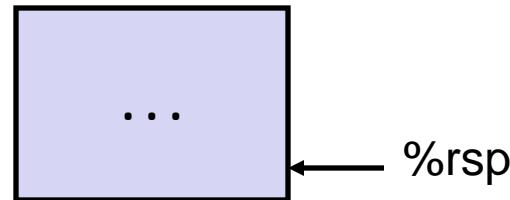
Registro	Uso(s)	Tipo
%rax	Valor de retorno	Salva-invocante 

pcount_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq   %rdi, %rbx
    andl   $1, %ebx
    shrq   %rdi
    call   pcount_r
    addq   %rbx, %rax
    popq   %rbx
```

.L6:

rep; ret



Observaciones Sobre la Recursividad

■ Manejada sin Especiales Consideraciones

- Marcos pila implican que cada llamada a función tiene almacenamiento privado
 - Variables locales y registros preservados
 - Dirección de retorno salvada
- Convenciones preservación registros previenen que una llamada a función corrompa los datos de otra
 - A menos que el código C explícitamente lo haga (p.ej. buffer overflow)
- Disciplina de pila sigue el patrón de llamadas / retornos
 - Si P llama a Q, entonces Q retorna antes que P
 - Primero en entrar, último en salir (Last-In, First-Out)[†]

■ También funciona con recursividad mutua[‡]

- P llama a Q; Q llama a P

[‡] en general con reentrantia

[†] pila = lista LIFO

60

Resumen de Procedimientos en x86-64

■ Puntos Importantes

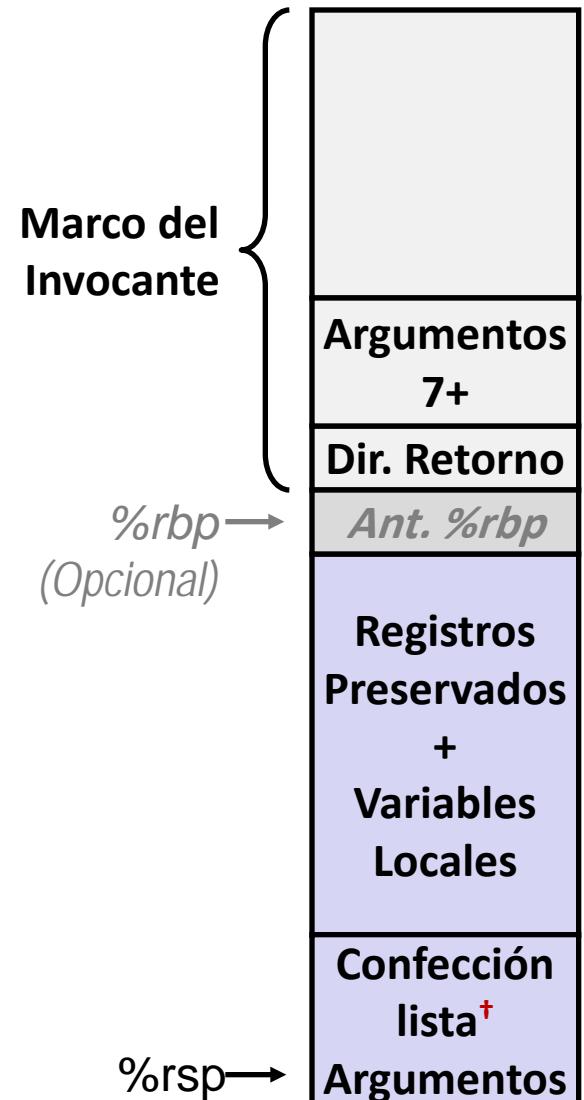
- Pila es la estructura de datos correcta para llamada / retorno procedimientos
 - P llama a Q, entonces Q retorna antes que P

■ Recursividad (y recursividad mutua) con mismas convenciones de llamada normales

- Se pueden almacenar valores tranquilamente en el marco de pila local y en registros salva-invocado
- Poner argumentos 7+ de la función en tope de pila
- Devolver resultado en %rax

■ Punteros son direcciones de valores

- Global o en pila



Guía de trabajo autónomo (4h/s)

■ **Estudio:** del Cap.3 CS:APP (Bryant/O'Hallaron)

- Procedures
 - § 3.7 pp.274-291
- Understanding Pointers, Using GDB.
 - § 3.10.1-2 pp.312-316

■ **Ejercicios:** del Cap.3 CS:APP (Bryant/O'Hallaron)

- Probl. 3.32 § 3.7.2, pp.280
- Probl. 3.33 § 3.7.3, pp.282
- Probl. 3.34 § 3.7.5, pp.288
- Probl. 3.35 § 3.7.6, pp.290

Bibliografía:

[BRY16] Cap.3

Computer Systems: A Programmer's Perspective 3rd ed. Bryant, O'Hallaron. Pearson, 2016

Signatura ESIIT/[C.1 BRY com](#)

Práctica 1: Entorno de desarrollo GNU

Estructura de Computadores

Gustavo Romero López

Updated: 24 de septiembre de 2020

Arquitectura y Tecnología de Computadores

Índice

1. Índice	6.3 C++
2. Objetivos	6.4 32 bits
3. Introducción	6.5 64 bits
4. C	6.6 ASM + C
5. Ensamblador	6.7 Optimización
6. Ejemplos	7. Compiler Explorer
6.1 hola	8. Enlaces
6.2 make	

Objetivos

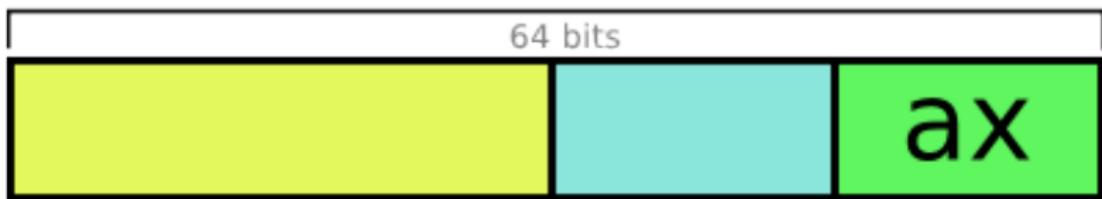
- Nociones de ensamblador 80x86 de 64 bits.
- Linux es tu amigo: si no sabes algo pregunta... **man**.
- Hoy aprenderemos varias cosas:
 - El esqueleto de un programa básico en ensamblador.
 - Como aprender de un maestro: el compilador **gcc**.
 - Herramientas clásicas del entorno de programación UNIX:
 - **make**: hará el trabajo sucio y rutinario por nosotros.
 - **as**: el ensamblador.
 - **ld**: el enlazador.
 - **gcc**: el compilador.
 - **nm**: lista los símbolos de un fichero.
 - **objdump**: el desensamblador.
 - **gdb** y **ddd** (gdb con cirugía estética): los depuradores.
 - Herramienta web: Compiler Explorer

Ensamblador 80x86

- ④ Los **80x86** son una familia de procesadores.
- ④ El más utilizado junto a los procesadores **ARM**.
- ④ En estas prácticas vamos a centrarnos en su **lenguaje ensamblador** (inglés).
- ④ El lenguaje ensamblador es el más básico, tras el binario, con el que podemos escribir programas utilizando las **instrucciones** que entiende el procesador.
- ④ Cualquier estructura de un lenguajes de alto nivel pueden crearse mediante instrucciones muy sencillas.
- ④ Normalmente es utilizado para poder acceder a partes que los lenguajes de alto nivel nos ocultan, complican o hacen de forma inconveniente.

Arquitectura 80x86: el registro A

rax



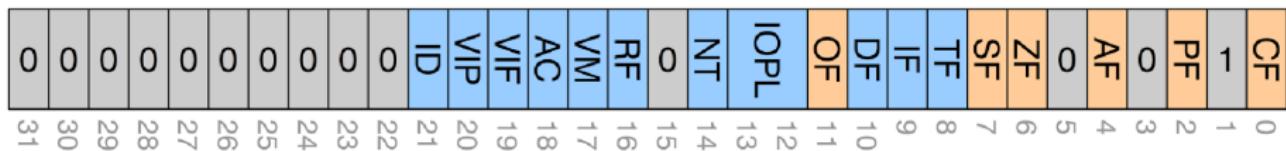
eax

Arquitectura 80x86: registros completos

ZMM0	YMM0	XMM0	ZMM1	YMM1	XMM1	ST(0)	MM0	ST(1)	MM1	ALAH	AXEAX	RAX	RW	R8D	R8	R12W	R12D	R12	CR0	CR4	
ZMM2	YMM2	XMM2	ZMM3	YMM3	XMM3	ST(2)	MM2	ST(3)	MM3	BFBH	BXBX	RBX	RW	R9W	R9D	R9	R13W	R13D	R13	CR1	CR5
ZMM4	YMM4	XMM4	ZMM5	YMM5	XMM5	ST(4)	MM4	ST(5)	MM5	CCHC	CXECX	RCX	R10W	R10D	R10	R14W	R14D	R14	CR2	CR6	
ZMM6	YMM6	XMM6	ZMM7	YMM7	XMM7	ST(6)	MM6	ST(7)	MM7	DIDM	DXEDX	RDX	R11W	R11D	R11	R15W	R15D	R15	CR3	CR7	
ZMM8	YMM8	XMM8	ZMM9	YMM9	XMM9					BPLB	PEBPRBP		DIL	DI	EDI	RDI			IP	EIP	RIP
ZMM10	YMM10	XMM10	ZMM11	YMM11	XMM11	CW	FP_IP	FP_DP	FP_CS	SIL	SI	ESI	RSI	SPN	SP	ESP	RSP		MSW	CR9	
ZMM12	YMM12	XMM12	ZMM13	YMM13	XMM13	SW													CR10		
ZMM14	YMM14	XMM14	ZMM15	YMM15	XMM15	TW													CR11		
ZMM16	ZMM17	ZMM18	ZMM19	ZMM20	ZMM21	ZMM22	ZMM23	FP_DS											CR12		
ZMM24	ZMM25	ZMM26	ZMM27	ZMM28	ZMM29	ZMM30	ZMM31	FP_OPC	FP_DP	FP_IP	CS	SS	DS	GDTR	IDTR		DRO	DR6	CR13		
											ES	FS	GS	TR	LDTR		DR1	DR7	CR14		
														FLAGS	EFLAGS	RFLAGS	DR2	DR8	CR15		
																	DR3	DR9	MXCSR		
																	DR4	DR10	DR12		
																	DR5	DR11	DR13		
																		DR12	DR14		
																		DR13	DR15		

Arquitectura 80x86: banderas

eflags register



■ Reserved flags

■ System flags

■ Arithmetic flags

TF: Trap
IF: Interrupt
DF: Direction

CF: Carry
PF: Parity
AF: Adjust
ZF: Zero
SF: Sign
OF: Overflow

Arquitectura 80x86: paso de parámetros a funciones

%rax	Return value	%r8	Argument #5
%rbx	Callee saved	%r9	Argument #6
%rcx	Argument #4	%r10	Caller saved
%rdx	Argument #3	%r11	Caller Saved
%rsi	Argument #2	%r12	Callee saved
%rdi	Argument #1	%r13	Callee saved
%rsp	Stack pointer	%r14	Callee saved
%rbp	Callee saved	%r15	Callee saved

Programa mínimo en C... todos ellos equivalentes

minimo1.c

```
int main() {}
```

minimo2.c

```
int main() { return 0; }
```

minimo3.c

```
#include <stdlib.h>
int main() { exit(0); }
```

Trasteando el programa mínimo en C


```
linux-vdso.so.1 (0x00007ffe2ddbc000)
libc.so.6 => /lib64/libc.so.6 (0x00007fbc5043a000)
/lib64/ld-linux-x86-64.so.2 (0x0000558dbe5aa000)
```

- Examinar biblioteca: `objdump -d /lib64/libc.so.6`

Ensamblador desde 0: secciones básicas de un programa

```
1 .data  
2  
3 .text
```

Ensamblador desde 0: punto de entrada

```
1 .text  
2     .global _start
```

Ensamblador desde 0: datos

```
1 .data
2 msg:      .string "¡hola, mundo!\n"
3 tam:      .quad . - msg
```

Ensamblador desde 0: código

```
1 write:    mov    $1,    %rax    # write
2           mov    $1,    %rdi    # stdout
3           mov    $msg,   %rsi    # texto
4           mov    tam,   %rdx    # tamaño
5           syscall          # llamada a write
6           ret
7
8 exit:     mov    $60,   %rax    # exit
9           xor    %rdi,   %rdi    # 0
10          syscall         # llamada a exit
```

Ensamblador desde 0: ejemplo básico hola.s

```
1 .data
2 msg:     .string "¡hola, mundo!\n"
3 tam:     .quad . - msg
4
5 .text
6     .global _start
7
8 write:   mov    $1,    %rax  # write
9         mov    $1,    %rdi  # stdout
10        mov    $msg,   %rsi  # texto
11        mov    tam,    %rdx  # tamaño
12        syscall          # llamada a write
13        ret
14
15 exit:    mov    $60,   %rax  # exit
16        xor    %rdi,   %rdi  # 0
17        syscall          # llamada a exit
18        ret
19
20 _start:  call   write      # llamada a función
21         call   exit       # llamada a función
22
```

¿Cómo hacer ejecutable mi programa?

¿Cómo hacer ejecutable el código anterior?

- ◎ opción a: ensamblar + enlazar
 - `as hola.s -o hola.o`
 - `ld hola.o -o hola`
- ◎ opción b: compilar = ensamblar + enlazar
 - `gcc -nostdlib hola.s -o hola`
- ◎ opción c: que lo haga alguien por mi → make
 - `makefile`: fichero con definiciones, objetivos y recetas.

Ejercicios:

1. Cree un ejecutable a partir de `hola.s`.
2. Use `file` para ver el tipo de cada fichero.
3. Descargue el fichero `makefile`, pruébelo e intente hacer alguna modificación.
4. Examine el código ensamblador con `objdump -d hola`.

makefile

<http://pccito.ugr.es/~gustavo/ec/practicas/1/makefile>

```
ASM = $(wildcard *.s)
SRC = $(wildcard *.c *.cc)
EXE = $(basename $(ASM) $(SRC))
ATT = $(EXE:=.att)
```

```
CFLAGS = -g -std=c11 -Wall
CXXFLAGS = $(CFLAGS:c11=c++11)
```

```
all: $(EXE) $(ATT)
```

```
clean:
    -rm -fv $(ATT) $(EXE) *~
```

Ejemplo en C++: hola-c++.cc

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "¡hola, mundo!"
6             << std::endl;
7 }
```

- ◎ ¿Qué hace gcc con mi programa?
- ◎ La única forma de saberlo es desensamblarlo:
 - Sintaxis AT&T: objdump -C -d hola-c++
 - Sintaxis Intel: objdump -C -d hola-c++ -M intel

Ejercicios:

5. ¿Qué hace ahora diferente la función main() respecto a C?

```
1 write:    movl    $4, %eax      # write
2             movl    $1, %ebx      # salida estándar
3             movl    $msg, %ecx      # cadena
4             movl    tam, %edx      # longitud
5             int     $0x80      # llamada a write
6             ret          # retorno
7
8 exit:     movl    $1, %eax      # exit
9             xorl    %ebx, %ebx      # 0
10            int    $0x80      # llamada a exit
```

Ejercicios:

6. Descargue hola32.s. Ejecute el programa instrucción por instrucción con el ddd hasta comprender como funciona.
7. Si quiere aprender un poco más estudie hola32p.s. Sobre el mismo podemos destacar: código de 32 bits, uso de “*little endian*”, llamada a subrutina, uso de la pila y codificación de caracteres.

Depuración: hola64.s

ejemplo de 64 bits

```
1 write:    mov      $1,      %rax    # write
2             mov      $1,      %rdi    # stdout
3             mov      $msg,    %rsi    # texto
4             mov      tam,    %rdx    # tamaño
5             syscall           # llamada a write
6             ret
7
8 exit:     mov      $60,    %rax    # exit
9             xor      %rdi,    %rdi    # 0
10            syscall          # llamada a exit
11            ret
```

Ejercicios:

8. Descargue hola64.s. Ejecute el programa instrucción por instrucción con el ddd hasta comprender como funciona.
9. Compare hola64.s con hola64p.s. Sobre este podemos destacar: código de 64 bits, llamada a subrutina, uso de la pila y codificación de caracteres.

¿Dónde están mis datos?

printf-c-1.c y printf-c-2.c

- ◎ ¿Sabes C? \Longleftrightarrow ¿Has usado la función printf()?

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int i = 12345;
6     printf("i=%d\n", i);
7     return 0;
8 }
```

```
1 #include <stdio.h>
2
3 int i = 12345;
4 char *formato = "i=%d\n";
5
6 int main()
7 {
8     printf(formato, i);
9     return 0;
10 }
```

Ejercicios:

10. ¿En qué se parecen y en qué se diferencian printf-c-1.c y printf-c-2.c? nm, objdump y kdiff3 serán muy útiles...

Mezclando lenguajes: ensamblador y C (32 bits) printf32.s

```
1 .data
2 i:      .int 12345          # variable entera
3 f:      .string "i = %d\n" # cadena de formato
4
5 .text
6     .extern printf        # printf en otro sitio
7     .globl _start         # función principal
8
9 _start: push (i)           # apila i
10    push $f               # apila f
11    mov $0, %eax          # n de registros vectoriales
12    call printf           # llamada a printf
13    add $8, %esp           # restaura pila
14
15    movl $1, %eax          # exit
16    xorl %ebx, %ebx       # 0
```

Ejercicios:

11. Descargue y compile printf32.s.
12. Modifique printf32.s para que finalice mediante la función exit() de C (`man 3 exit`). Solución: printf32e.s.

Mezclando lenguajes: ensamblador y C (64 bits) printf64.s

```
1 .data
2 i:      .int 12345          # variable entera
3 f:      .string "i = %d\n" # cadena de formato
4
5 .text
6     .globl _start
7
8 _start: mov $f, %rdi        # formato
9         mov (i), %rsi        # i
10        xor %rax, %rax      # n de registros vectoriales
11        call printf         # llamada a función
12
13        xor %rdi, %rdi      # valor de retorno
14        call exit           # llamada a función
```

Ejercicios:

13. Descargue y compile printf64.s.
14. Busque las diferencias entre printf32.s y printf64.s.

Optimización: sum.cc

```
1 int main()
2 {
3     int sum = 0;
4
5     for (int i = 0; i < 10; ++i)
6         sum += i;
7
8     return sum;
9 }
```

Ejercicios:

15. ¿Cómo implementa gcc los bucles **for**?
16. Observe el código de la función **main()** al compilarlo...
 - sin optimización: g++ -O0 sum.cc -o sum
 - con optimización: g++ -O3 sum.cc -o sum

Optimización: función main() de sum.cc

sin optimización (gcc -O0)

```
4005b6: 55                      push    %rbp
4005b7: 48 89 e5                mov     %rsp,%rbp
4005ba: c7 45 fc 00 00 00 00    movl    $0x0,-0x4(%rbp)
4005c1: c7 45 f8 00 00 00 00    movl    $0x0,-0x8(%rbp)
4005c8: eb 0a                   jmp    4005d4 <main+0x1e>
4005ca: 8b 45 f8                mov     -0x8(%rbp),%eax
4005cd: 01 45 fc                add    %eax,-0x4(%rbp)
4005d0: 83 45 f8 01             addl   $0x1,-0x8(%rbp)
4005d4: 83 7d f8 09             cmpl   $0x9,-0x8(%rbp)
4005d8: 7e f0                   jle    4005ca <main+0x14>
4005da: 8b 45 fc                mov     -0x4(%rbp),%eax
4005dd: 5d                      pop    %rbp
4005de: c3                      retq
```

con optimización (gcc -O3)

```
4004c0: b8 2d 00 00 00          mov     $0x2d,%eax
4004c5: c3                      retq
```

Compiler Explorer: <https://godbolt.org/z/7uIN9y>

The screenshot shows the Compiler Explorer interface with two compiler panes. The left pane displays C++ source code, and the right pane shows the assembly output generated by two different compilers.

C++ source #1:

```
template <class T>
concept bool Addable =
    requires (T t) { t + t; };

int main()
{
    int x = 1, y = 2;
    Addable a = x + y;
    return a;
}
```

x86-64 gcc 8.2 (Editor #1, Compiler #1) C++:

```
-fconcepts
```

```
main:
pushq %rbp
movq %rsp, %rbp
movl $1, -4(%rbp)
movl $2, -8(%rbp)
movl -4(%rbp), %edx
movl -8(%rbp), %eax
addl %edx, %eax
movl %eax, -12(%rbp)
movl -12(%rbp), %eax
popq %rbp
ret
```

x86-64 gcc 8.2 (Editor #1, Compiler #2) C++:

```
-fconcepts -O3
```

```
main:
movl $3, %eax
ret
```

Compiler Explorer: <https://godbolt.org/z/vUF7yA>

The image shows the Compiler Explorer interface with two compiler panes and a C++ source code editor.

Source Code:

```
1 template<typename T> T adder(T v)
2 {
3     return v;
4 }
5
6 template<typename T, typename... Args>
7 {
8     return first + adder(args...);
9 }
10
11 int main()
12 {
13     return adder(1, 2, 3, 4, 5);
14 }
```

Compiler #1 (Left): x86-64 gcc 8.2

```
1 main:
2     pushq %rbp
3     movq %rsp, %rbp
4     movl $5, %r8d
5     movl $4, %ecx
6     movl $3, %edx
7     movl $2, %esi
8     movl $1, %edi
9     call int adder<int, int, int, int, int>(%rbp)
10    nop
11    popq %rbp
12    ret
13 int adder<int, int, int, int, int>(int, int, int, int, int)
14     pushq %rbp
15     movq %rsp, %rbp
16     subq $32, %rsp
17     movl %edi, -4(%rbp)
18     movl %esi, -8(%rbp)
19     movl %edx, -12(%rbp)
20     movl %ecx, -16(%rbp)
21     movl %r8d, -20(%rbp)
22     movl -20(%rbp), %ecx
23     movl -16(%rbp), %edx
24     movl -12(%rbp), %esi
25     movl -8(%rbp), %eax
26     movl %eax, %edi
27     call int adder<int, int, int, int>(%rbp)
28     movl %eax, %edx
29     movl -4(%rbp), %eax
30     addl %edx, %eax
31     leave
32     ret
33 int adder<int, int, int, int>(int, int, int, int)
34     pushq %rbp
35     movq %rsp, %rbp
36     subq $16, %rsp
```

Compiler #2 (Right): x86-64 gcc 8.2

```
1 main:
2     movl $15, %eax
3     ret
```

Enlaces de interés

Manuales:

◎ Hardware:

- AMD
- Intel

◎ Software:

- AS
- NASM

Programación:

- ◎ Programming from the ground up
- ◎ Linux Assembly

Chuletas:

- ◎ Chuleta del 8086
- ◎ Chuleta del GDB

i para más información ver especificaciones de la instrucción.

Flags: + = Afectado por esta instrucción. ? = Indefinido luego de esta instrucción.

ARITMÉTICOS		Código	Operación	Flags							
Nombre	Comentario			O	D	I	T	S	Z	A	P
ADD	Suma	ADD Dest,Fuente	Dest:=Dest+ Fuente	±				±	±	±	±
ADC	Suma con acarreo	ADC Dest,Fuente	Dest:=Dest+ Fuente +CF	±				±	±	±	±
SUB	Resta	SUB Dest,Fuente	Dest:=Dest- Fuente	±				±	±	±	±
SBB	Resta con acarreo	SBB Dest,Fuente	Dest:=Dest-(Fuente +CF)	±				±	±	±	±
DIV	División (sin signo)	DIV Op	Op=byte: AL:=AX / Op	AH:=Resto	?			?	?	?	?
DIV	División (sin signo)	DIV Op	Op=word: AX:=DX:AX / Op	DX:=Resto	?			?	?	?	?
DIV 386	División (sin signo)	DIV Op	Op=doublew.: EAX:=EDX:EAX / Op	EDX:=Resto	?			?	?	?	?
IDIV	División entera con signo	IDIV Op	Op=byte: AL:=AX / Op	AH:=Resto	?			?	?	?	?
IDIV	División entera con signo	IDIV Op	Op=word: AX:=DX:AX / Op	DX:=Resto	?			?	?	?	?
IDIV 386	División entera con signo	IDIV Op	Op=doublew.: EAX:=EDX:EAX / Op	EDX:=Resto	?			?	?	?	?
MUL	Multiplicación (sin signo)	MUL Op	Op=byte: AX:=AL*Op	si AH=0 ◆	±			?	?	?	?
MUL	Multiplicación (sin signo)	MUL Op	Op=word: DX:AX:=AX*Op	si DX=0 ◆	±			?	?	?	?
MUL 386	Multiplicación (sin signo)	MUL Op	Op=double: EDX:EAX:=EAX*Op	si EDX=0 ◆	±			?	?	?	?
IMUL i	Multiplic. entera con signo	IMUL Op	Op=byte: AX:=AL*Op	si AL es suficiente ◆	±			?	?	?	?
IMUL	Multiplic. entera con signo	IMUL Op	Op=word: DX:AX:=AX*Op	si AX es suficiente ◆	±			?	?	?	?
IMUL 386	Multiplic. entera con signo	IMUL Op	Op=double: EDX:EAX:=EAX*Op	si EAX es suf. ◆	±			?	?	?	?
INC	Incrementar	INC Op	Op:=Op+1 (El Carry no resulta afectado !)		±			±	±	±	±
DEC	Decrementar	DEC Op	Op:=Op-1 (El Carry no resulta afectado !)		±			±	±	±	±
CMP	Comparar	CMP Op1,Op2	Op1-Op2		±			±	±	±	±
SAL	Desplazam. aritm. a la izq.	SAL Op,Cantidad		i				±	±	?	±
SAR	Desplazam. aritm. a la der.	SAR Op,Cantidad		i				±	±	?	±
RCL	Rotar a la izq. c/acarreo	RCL Op,Cantidad		i							
RCR	Rotar a la derecha c/acarreo	RCR Op,Cantidad		i							
ROL	Rotar a la izquierda	ROL Op,Cantidad		i							
ROR	Rotar a la derecha	ROR Op,Cantidad		i							

i para más información ver especificaciones de la instrucción.

♦ entonces $CF=0$, $QE=0$ sino $CF=1$, $QE=1$

LÓGICOS		Código	Operación	Flags								
Nombre	Comentario			O	D	I	T	S	Z	A	P	C
NEG	Negación (complemento a 2)	NEG Op	Op:=0-Op si Op=0 entonces CF:=0 sino CF:=1	±				±	±	±	±	±
NOT	Invertir cada bit	NOT Op	Op:=~Op (invierte cada bit)									
AND	'Y' (And) lógico	AND Dest,Fuente	Dest:=Dest&Fuente	0				±	±	?	±	0
OR	'O' (Or) lógico	OR Dest,Fuente	Dest:=Dest Fuente	0				±	±	?	±	0
XOR	'O' (Or) exclusivo	XOR Dest,Fuente	Dest:=Dest (xor) Fuente	0				±	±	?	±	0
SHL	Desplazam. lógico a la izq.	SHL Op,Cantidad		i				±	±	?	±	±
SHR	Desplazam. lógico a la der.	SHR Op,Cantidad		i				±	±	?	±	±

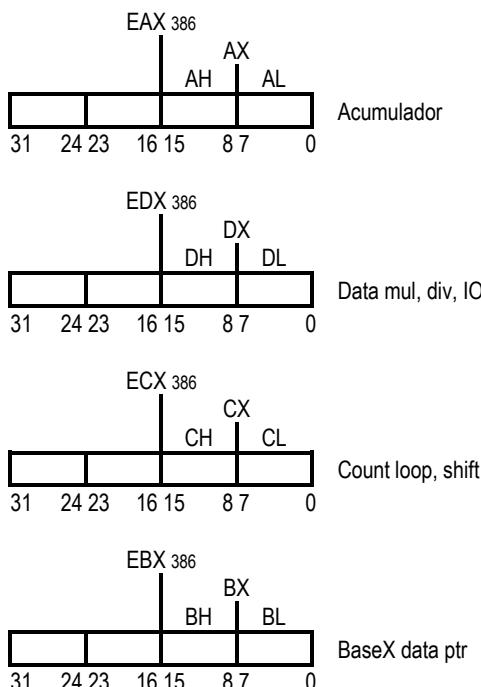
Tabla de Códigos 2/2

MISCELÁNEOS		Código	Operación	Flags							
Nombre	Comentario			O	D	I	T	S	Z	A	P
NOP	Hacer nada	NOP	No hace operación alguna								
LEA	Cargar dirección Efectiva	LEA Dest,Fuente	Dest := dirección fuente								
INT	Interrupción	INT Num	Interrumpe el progr. actual, corre la subrutina de int.		0	0					

SALTOS (generales)		Código	Operación	Name	Comentario	Código	Operación
Nombre	Comentario						
CALL	Llamado a subrutina	CALL Proc		RET	Retorno de subrutina	RET	
JMP	Saltar	JMP Dest					
JE	Saltar si es igual	JE Dest	(= JZ)	JNE	Saltar si no es igual	JNE Dest	(= JNZ)
JZ	Saltar si es cero	JZ Dest	(= JE)	JNZ	Saltar si no es cero	JNZ Dest	(= JNE)
JCXZ	Saltar si CX es cero	JCXZ Dest		JECXZ	Saltar si ECX es cero	JECXZ Dest	386
JP	Saltar si hay paridad	JP Dest	(= JPE)	JNP	Saltar si no hay paridad	JNP Dest	(= JPO)
JPE	Saltar si hay paridad par	JPE Dest	(= JP)	JPO	Saltar si hay paridad impar	JPO Dest	(= JNP)

SALTOS Sin Signo (Cardinal)				SALTOS Con Signo (Integer)			
JA	Saltar si es superior	JA Dest	(= JNBE)	JG	Saltar si es mayor	JG Dest	(= JNLE)
JAE	Saltar si es superior o igual	JAE Dest	(= JNB = JNC)	JGE	Saltar si es mayor o igual	JGE Dest	(= JNL)
JB	Saltar si es inferior	JB Dest	(= JNAE = JC)	JL	Saltar si es menor	JL Dest	(= JNGE)
JBE	Saltar si es inferior o igual	JBE Dest	(= JNA)	JLE	Saltar si es menor o igual	JLE Dest	(= JNG)
JNA	Saltar si no es superior	JNA Dest	(= JBE)	JNG	Saltar si no es mayor	JNG Dest	(= JLE)
JNAE	Saltar si no es super. o igual	JNAE Dest	(= JB = JC)	JNGE	Saltar si no es mayor o igual	JNGE Dest	(= JL)
JNB	Saltar si no es inferior	JNB Dest	(= JAE = JNC)	JNL	Saltar si no es inferior	JNL Dest	(= JGE)
JNBE	Saltar si no es infer. o igual	JNBE Dest	(= JA)	JNLE	Saltar si no es menor o igual	JNLE Dest	(= JG)
JC	Saltar si hay carry	JC Dest		JO	Saltar si hay Overflow	JO Dest	
JNC	Saltar si no hay carry	JNC Dest		JNO	Saltar si no hay Overflow	JNO Dest	
				JS	Saltar si hay signo (=negativo)	JS Dest	
				JNS	Saltar si no hay signo (=posit.)	JNS Dest	

Registros Generales:



Ejemplo:

```
.DOSSEG
.MODEL SMALL
.STACK 1024
Two EQU 2 ; Constante
.VarB DB ?
.VarW DW 1010b ; define un Byte, cualquier valor
.VarW2 DW 257 ; define un Word, en binario
.VarD DD 0AFFFFh ; define un DoubleWord, en hexa
.S DB "Hello !",0 ; define un String
.CODE
main: MOV AX,DGROUP ; resuelto por el linker
      MOV DS,AX ; inicializa el reg. de segmento de datos
      MOV [VarB],42 ; inicializa VarB
      MOV [VarD],-7 ; setea VarD
      MOV BX,Offset[S] ; dirección de "H" de "Hello !"
      MOV AX,[VarW] ; poner el valor en el acumulador
      ADD AX,[VarW2] ; suma VarW2 a AX
      MOV [VarW2],AX ; almacena AX en VarW2
      MOV AX,4C00h ; regresa al sistema
      INT 21h
END main
```



Flags: - - - - O D I T S Z - A - P - C

Flags de Control (cómo se manejan las instrucciones):

- D: Dirección 1=Los op's String se procesan de arriba hacia abajo
I: Interrupción Indica si pueden ocurrir interrupciones o no.
T: Trampa Paso por paso para debugging

Flags de Estado (resultado de las operaciones):

- C: Carry resultado de operac. sin signo es muy grande o inferior a cero
O: Overflow resultado de operac. sin signo es muy grande o pequeño.
S: Signo Signo del resultado. Razonable sólo para enteros. 1=neg. 0=posit.
Z: Cero Resultado de la operación es cero. 1=Cero
A: Carru Aux. Similar al Carry, pero restringido para el nibble bajo únicamente
P: Paridad 1=el resultado tiene cantidad par de bits en uno

CAPÍTULO 6

ANÁLISIS Y DISEÑO DE SISTEMAS SECUENCIALES

TECNOLOGÍA Y ORGANIZACIÓN DE COMPUTADORES

1º Grado en Ingeniería Informática.

TEMA 6. ANÁLISIS Y DISEÑO DE SISTEMAS SECuenciaLES.

RESUMEN:

En este tema se va a definir qué es un sistema secuencial y cómo se analizan y diseñan circuitos secuenciales sencillos. También se estudiarán algunos bloques secuenciales que realizan funciones más complejas y que no se pueden analizar a nivel de puertas lógicas.

TEMA 6. ANÁLISIS Y DISEÑO DE SISTEMAS SECuenciales.

OBJETIVOS (expresados como resultados de aprendizaje):

- Aplicar técnicas básicas de análisis y diseño de sistemas secuenciales a nivel lógico.
- Comprender las diferentes formas de representar el comportamiento de un sistema secuencial (diagramas, tablas de estados, cronogramas, etc.).
- Estimar las prestaciones de sistemas secuenciales (retardo de propagación, frecuencia máxima, etc.).
- Comprender el funcionamiento de los diferentes bloques secuenciales básicos que forman parte de la mayoría de los sistemas digitales, e identificar claramente la función que realizan.

TEMA 6. ANÁLISIS Y DISEÑO DE SISTEMAS SECuenciales.

CONTENIDOS:

- 6.1. Concepto de sistema secuencial.
- 6.2. Elementos básicos de memoria.
- 6.3. Análisis de un sistema secuencial.
- 6.4. Diseño de un sistema secuencial.
- 6.5. Componentes secuenciales estándar.

BIBLIOGRAFÍA: [GAJ97]:6,7 ; [HAY96]:6,7 ;
[LLO03]:7,8,9 ; [MAN05]:5,6,7 ; [NEL96]:6,7,8,9 ;
[ROT04]:11,12,13,14,15,16

TEMA 6. ANÁLISIS Y DISEÑO DE SISTEMAS SECuenciales.

CONTENIDOS:

- **6.1. Concepto de sistema secuencial.**
- 6.2. Elementos básicos de memoria.
- 6.3. Análisis de un sistema secuencial.
- 6.4. Diseño de un sistema secuencial.
- 6.5. Componentes secuenciales estándar.

6.1 INTRODUCCIÓN

- Un **sistema digital binario secuencial** es un sistema digital binario en el cual las salidas de dicho sistema, en un instante dado son funciones de las entradas de dicho sistema en el mismo instante de tiempo y de entadas en instantes de tiempo anteriores.

$$z_i(t) = z_i(x_{n-1}(t), x_{n-2}(t), \dots, x_0(t), x_{n-1}(t-1), x_{n-2}(t-1), \dots, x_0(t-1), \\ x_{n-1}(t-2), x_{n-2}(t-2), \dots, x_0(t-2), \dots \dots \dots, \\ x_{n-1}(0), x_{n-2}(0), \dots, x_0(0)) ; \forall i = 0, 1, 2, \dots, m-1$$

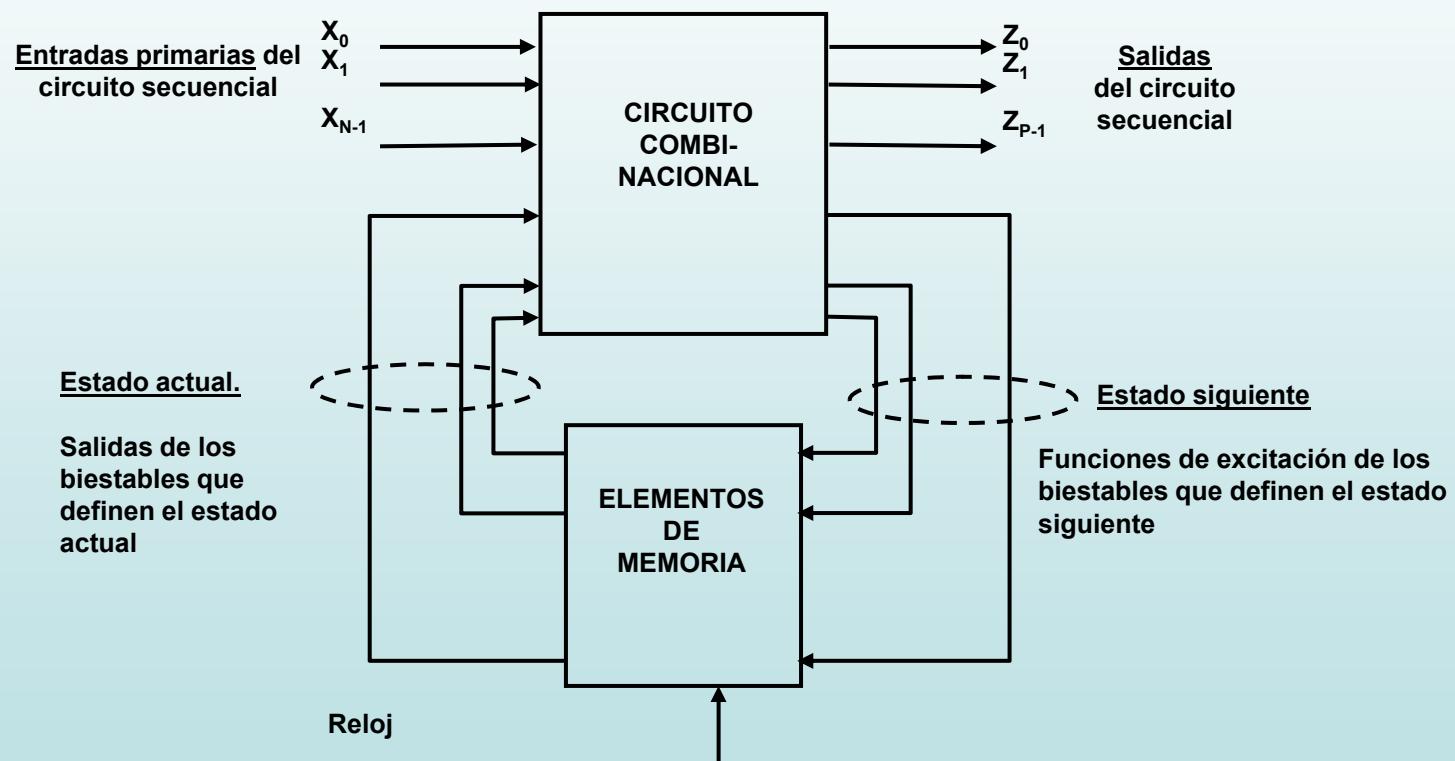
6.1 INTRODUCCIÓN

- **SISTEMA SECUENCIAL:**

- La salida en un instante dado depende de la secuencia de entradas recibida hasta dicho instante, es decir, de la “**historia**” del sistema.
- Son sistemas con **memoria** (“recuerdan” las entradas recibidas con anterioridad)
- La historia del sistema viene determinada por el **estado** del sistema en el momento en que empieza a funcionar y los valores de las entradas desde el principio.
- Los sistemas secuenciales necesitan **elementos de memoria** para poder memorizar el estado del sistema.
- En consecuencia, en un sistema secuencial hay que considerar 3 tipos de variables: **entradas, salidas y estados**.

6.1 INTRODUCCIÓN

- Estructura general de un sistema secuencial:



6.1 INTRODUCCIÓN

- **Ejemplo de sistema secuencial:**

Ascensor de un edificio de 4 plantas.

- Entradas: llamada a planta baja, llamada a 1^a planta, llamada a 2^a planta y llamada a 3^a planta
- Salidas: parado (motor parado), subir (motor girando en un sentido), bajar (motor girando en otro sentido)
Las salidas no solo dependen de las entradas sino de la historia anterior del sistema.
- Si se llama al ascensor desde la 2^a planta ¿cuál sería la salida del sistema? Depende de donde está (estado presente):
 - Si está en la misma planta → parado
 - Si está en la 3^a planta → bajar
 - Si está en la planta baja o 1^a planta → subir

6.1 INTRODUCCIÓN

- Podemos modelizar el sistema de la siguiente forma:

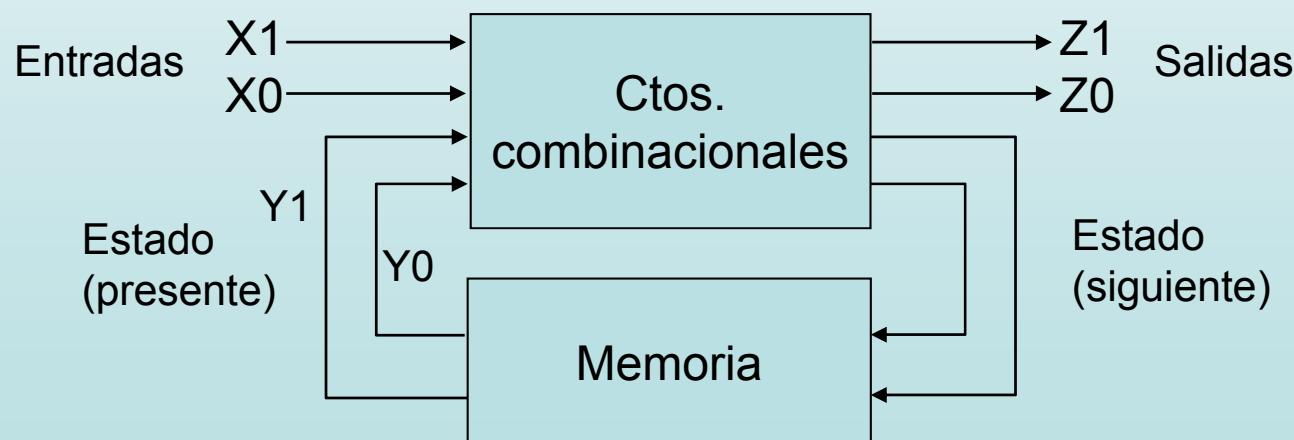
Variables de entrada		
X1	X0	Significado
0	0	Llamada desde planta baja
0	1	Llamada desde 1 ^a planta
1	0	Llamada desde 2 ^a planta
1	1	Llamada desde 3 ^a planta

Variables de salida		
Z1	Z0	Significado
0	0	Motor parado
0	1	Motor subiendo
1	0	Motor bajando

Variables de estado		
Y1	Y0	Significado
0	0	Ascensor en planta baja
0	1	Ascensor en 1 ^a planta
1	0	Ascensor en 2 ^a planta
1	1	Ascensor en 3 ^a planta

6.1 INTRODUCCIÓN

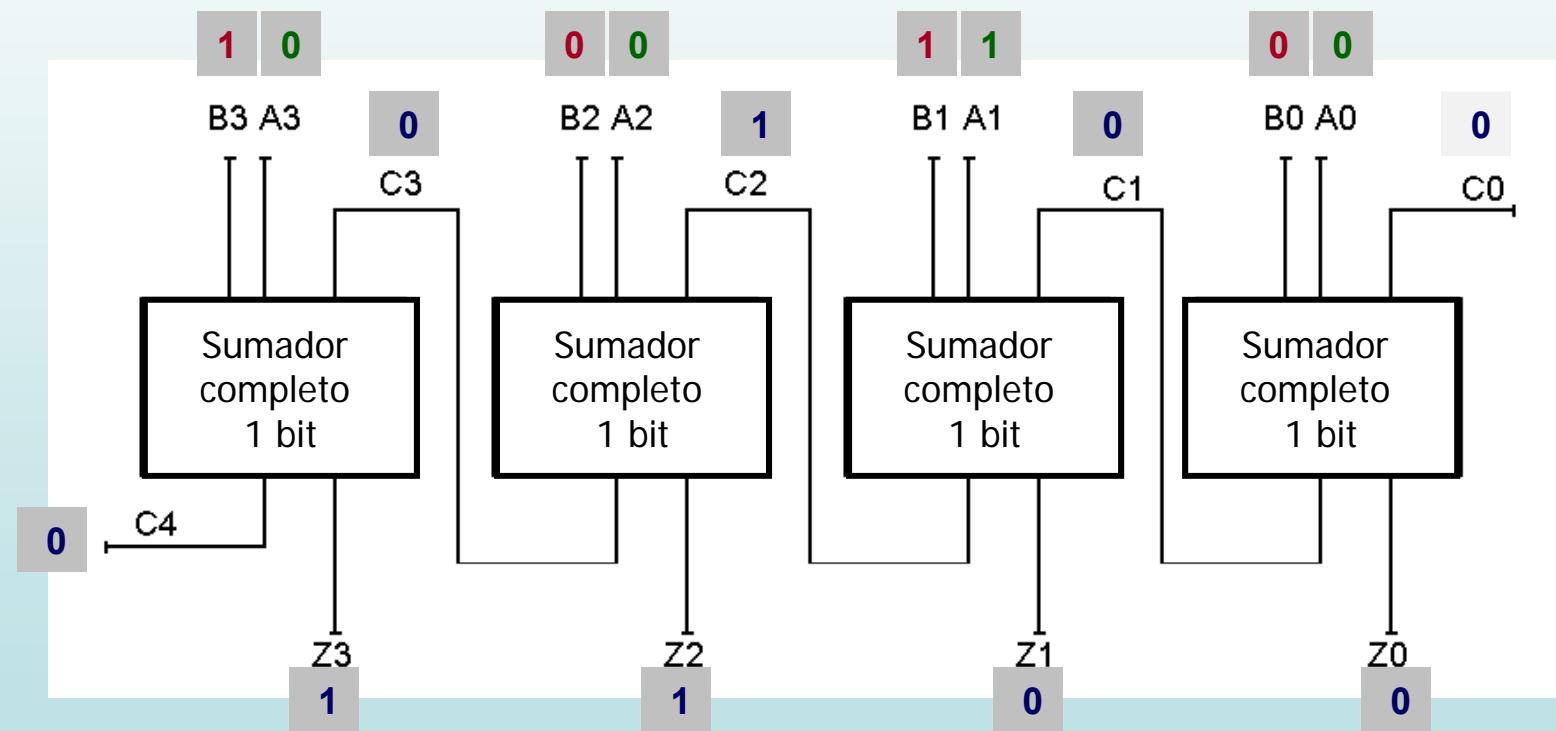
- Supongamos que el ascensor se encuentra en la 1^a planta. El estado presente es $Y_1Y_0=01$.
- Si lo llamamos desde la 3^a planta la entrada al sistema sería $X_1X_0=11$
- Entonces:
 - La salida del sistema será $Z_1Z_0=01$ (subir)
 - El estado siguiente pasa a ser $Y_1Y_0 = 11$ (3^a planta)



6.1 INTRODUCCIÓN

- Sumador combinacional:

$$\begin{array}{r} \text{Acarreo} = 010 \\ A = 0010 \\ B = 1010 \\ \hline Z = 1100 \end{array}$$



6.1 INTRODUCCIÓN

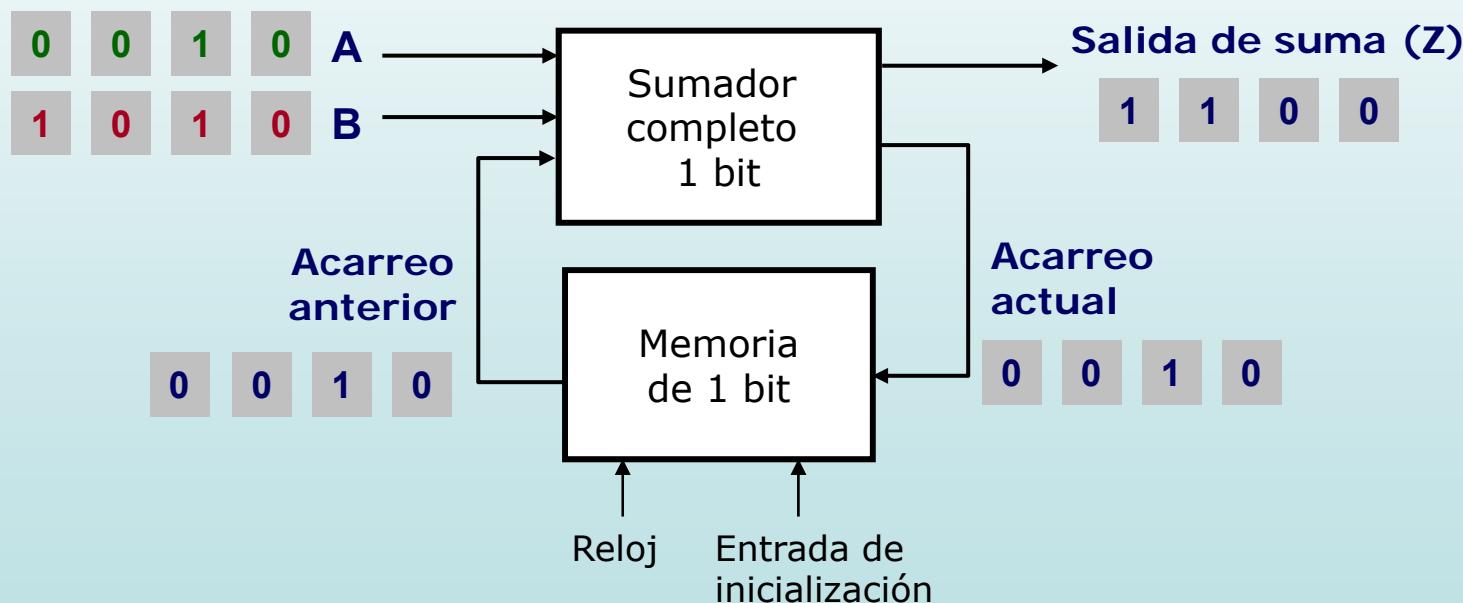
- Sumador secuencial:

$$\text{Acarreo} = 010$$

$$A = 0010$$

$$B = 1010$$

$$\underline{Z = 1100}$$



TEMA 6. ANÁLISIS Y DISEÑO DE SISTEMAS SECuenciales.

CONTENIDOS:

- 6.1. Concepto de sistema secuencial.
- 6.2. Elementos básicos de memoria.
- 6.3. Análisis de un sistema secuencial.
- 6.4. Diseño de un sistema secuencial.
- 6.5. Componentes secuenciales estándar.



6.2 ELEMENTOS BÁSICOS DE MEMORIA

- Los **elementos básicos de almacenamiento** que pueden memorizar un bit de información son los **cerrojos (latches)** o **biestables (flip-flops)**.
- En los elementos básicos de almacenamiento coincide la salida con el estado.
- Se suele denominar $Q(t)$ a la salida o **estado presente** y $Q(t+1)$ a la salida o **estado siguiente**.
- Un elemento de memoria tiene unas entradas que pueden producir un cambio en el valor (0, 1) memorizado en el circuito. La salida coincide con el valor memorizado.
- Los elementos de memoria suelen disponer de dos salidas, Q y Q' (complementaria).

6.2 ELEMENTOS BÁSICOS DE MEMORIA

- **Circuitos secuenciales síncronos:**

Es aquel en el que los cambios de estado en el sistema se producen únicamente cuando se activa una señal especial de entrada del sistema, llamada entrada de reloj.

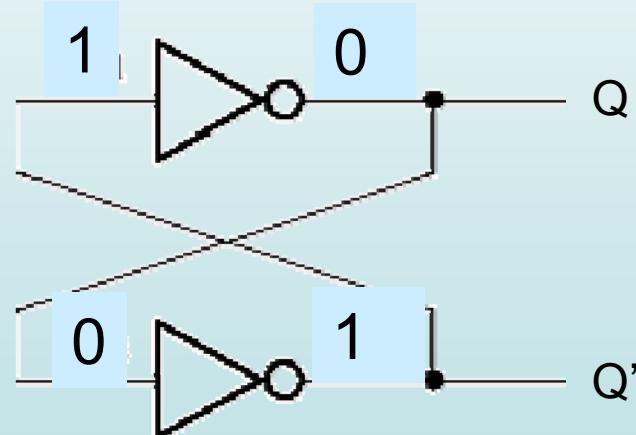
- **Circuitos secuenciales asíncronos:**

Es aquel en el que los cambios de estado se producen cuando cambia alguna/s de las entradas

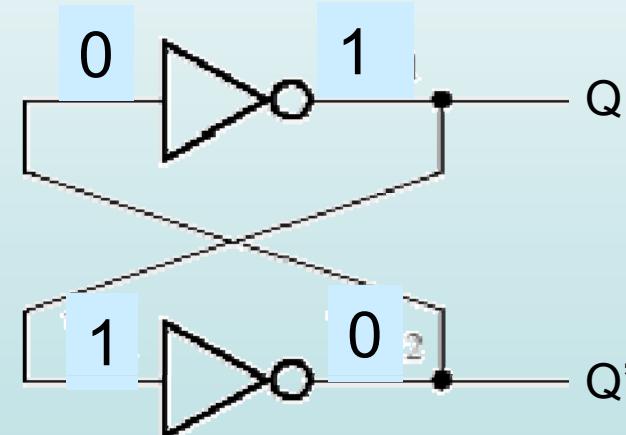
6.2 ELEMENTOS BÁSICOS DE MEMORIA

- **Biestable elemental latch:** elemento de memoria con dos estados estables: '0' y '1'.

Estado '0'
 $Q = 0; Q' = 1$



Estado '1'
 $Q = 1; Q' = 0$

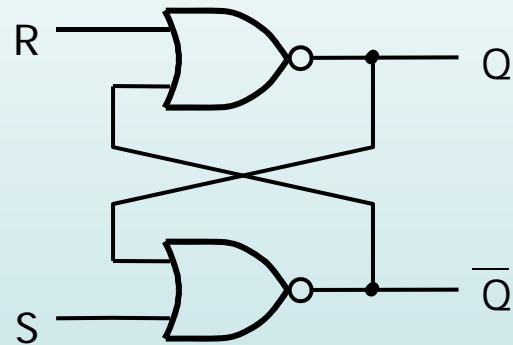


6.2 ELEMENTOS BÁSICOS DE MEMORIA

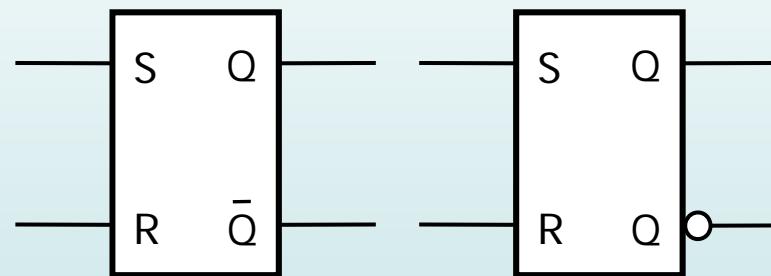
LATCH SR:

Dos puertas NOR (NAND) con las salidas realimentadas

Diagrama lógico



Símbolos



Resumen de su funcionamiento **normal**:

Si $SR=01 \rightarrow Q=0, Q'=1$ (Reset o puesta a 0)

Si $SR=10 \rightarrow Q=1, Q'=0$ (Set o puesta a 1)

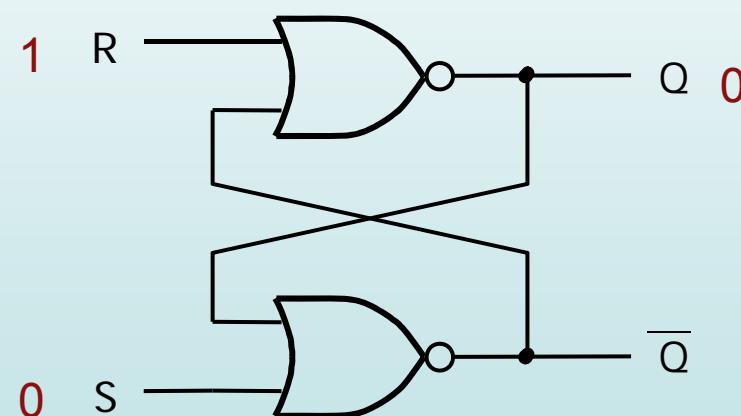
Si $SR=00 \rightarrow$ Se mantienen los valores de Q y Q' que había justo antes de hacerse $SR=00$ (Estado de Hold)

6.2 ELEMENTOS BÁSICOS DE MEMORIA

LATCH SR:

Funcionamiento:

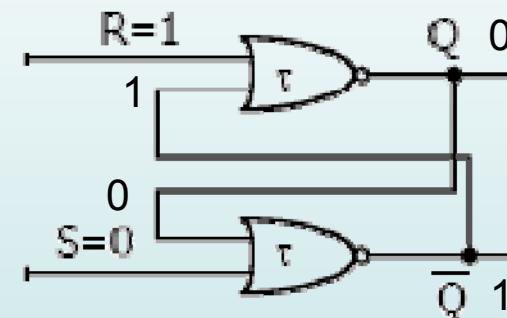
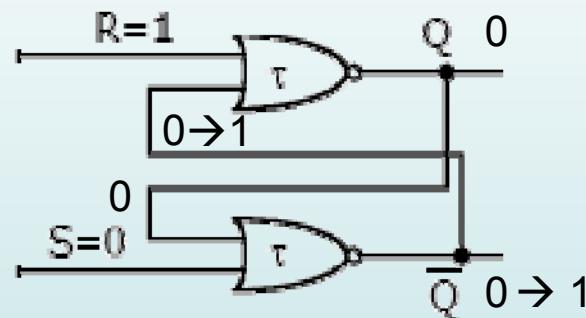
Si $SR=01 \rightarrow Q=0, Q'=1$ (Reset o puesta a 0)



XY	NOR
00	1
01	0
10	0
11	0

6.2 ELEMENTOS BÁSICOS DE MEMORIA

- Si **SR=01** → El circuito se estabiliza en **Q=0, Q'=1** (Puesta a 0, R=Reset)



XY	NOR
00	1
01	0
10	0
11	0

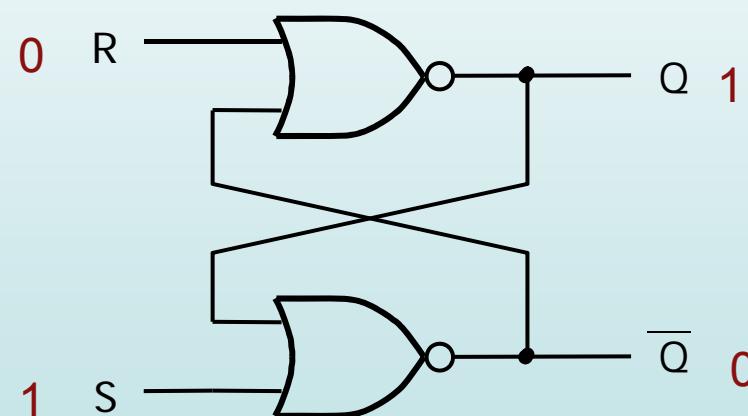


6.2 ELEMENTOS BÁSICOS DE MEMORIA

LATCH SR:

Funcionamiento:

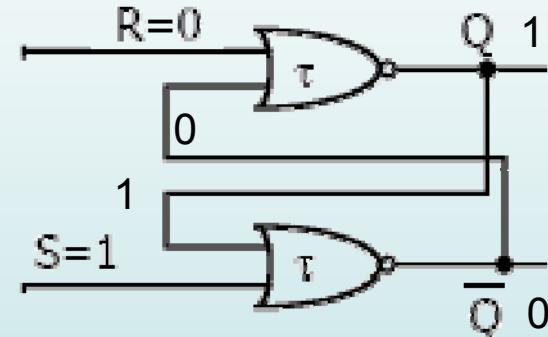
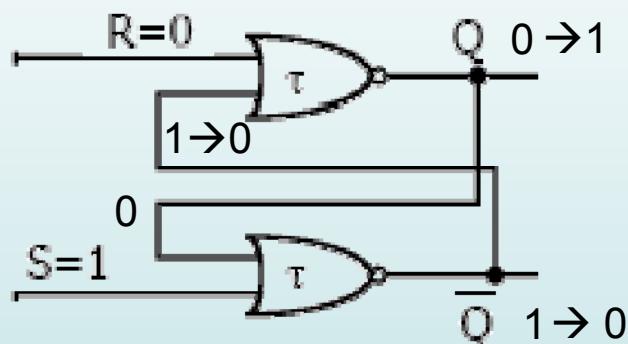
Si $SR=10 \rightarrow Q=1, Q'=0$ (Set o puesta a 1)



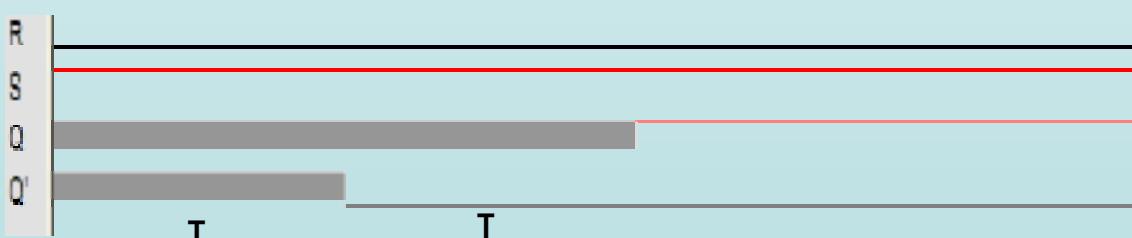
XY	NOR
00	1
01	0
10	0
11	0

6.2 ELEMENTOS BÁSICOS DE MEMORIA

- Si $SR = 10 \rightarrow$ El circuito se estabiliza en $Q=1$ y $Q'=0$ ('Puesta a 1', S=Set)



XY	NOR
00	1
01	0
10	0
11	0

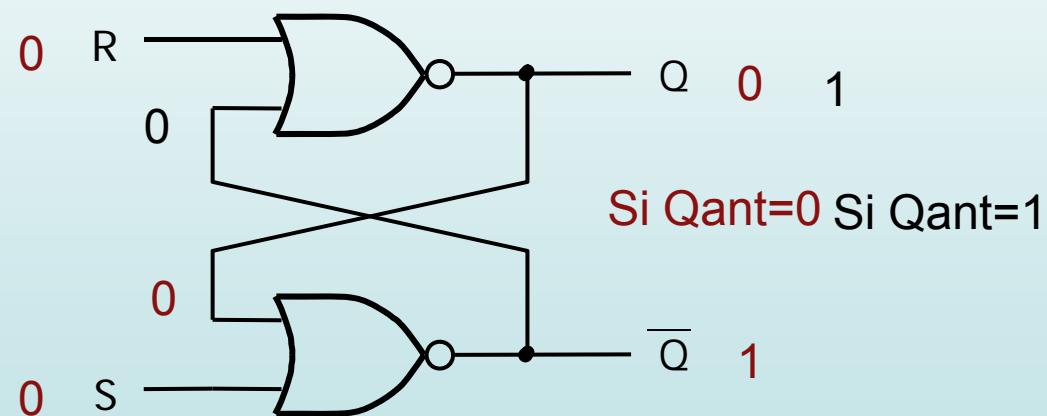


6.2 ELEMENTOS BÁSICOS DE MEMORIA

LATCH SR:

Funcionamiento:

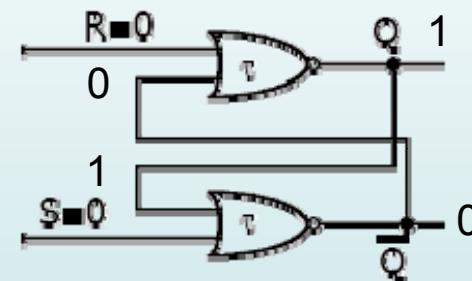
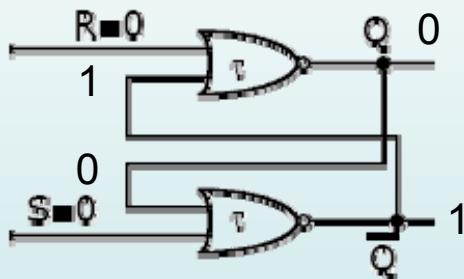
Si $SR=00 \rightarrow$ Se mantienen los valores de Q y Q' que había justo antes de hacerse $SR=00$ (Estado de Hold)



XY	NOR
00	1
01	0
10	0
11	0

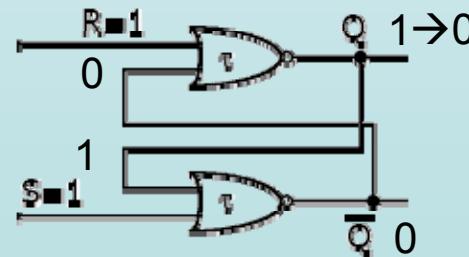
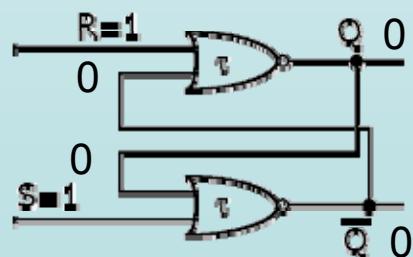
6.2 ELEMENTOS BÁSICOS DE MEMORIA

- Si **SR = 00**: su funcionamiento depende de los valores iniciales de Q y de Q'. Se almacena un bit de información.



XY	NOR
00	1
01	0
10	0
11	0

- Si **SR=11** → El circuito se estabiliza en **Q=0, Q'=0** (Estado prohibido)



6.2 ELEMENTOS BÁSICOS DE MEMORIA

- **Tabla de estados:** salidas que se producen para cualquier combinación entrada (tabla verdad).

Tabla de estados			
S	R	Q ^t	Q' ^t
0	0	Q	Q'
0	1	0	1
1	0	1	0
1	1	-	-

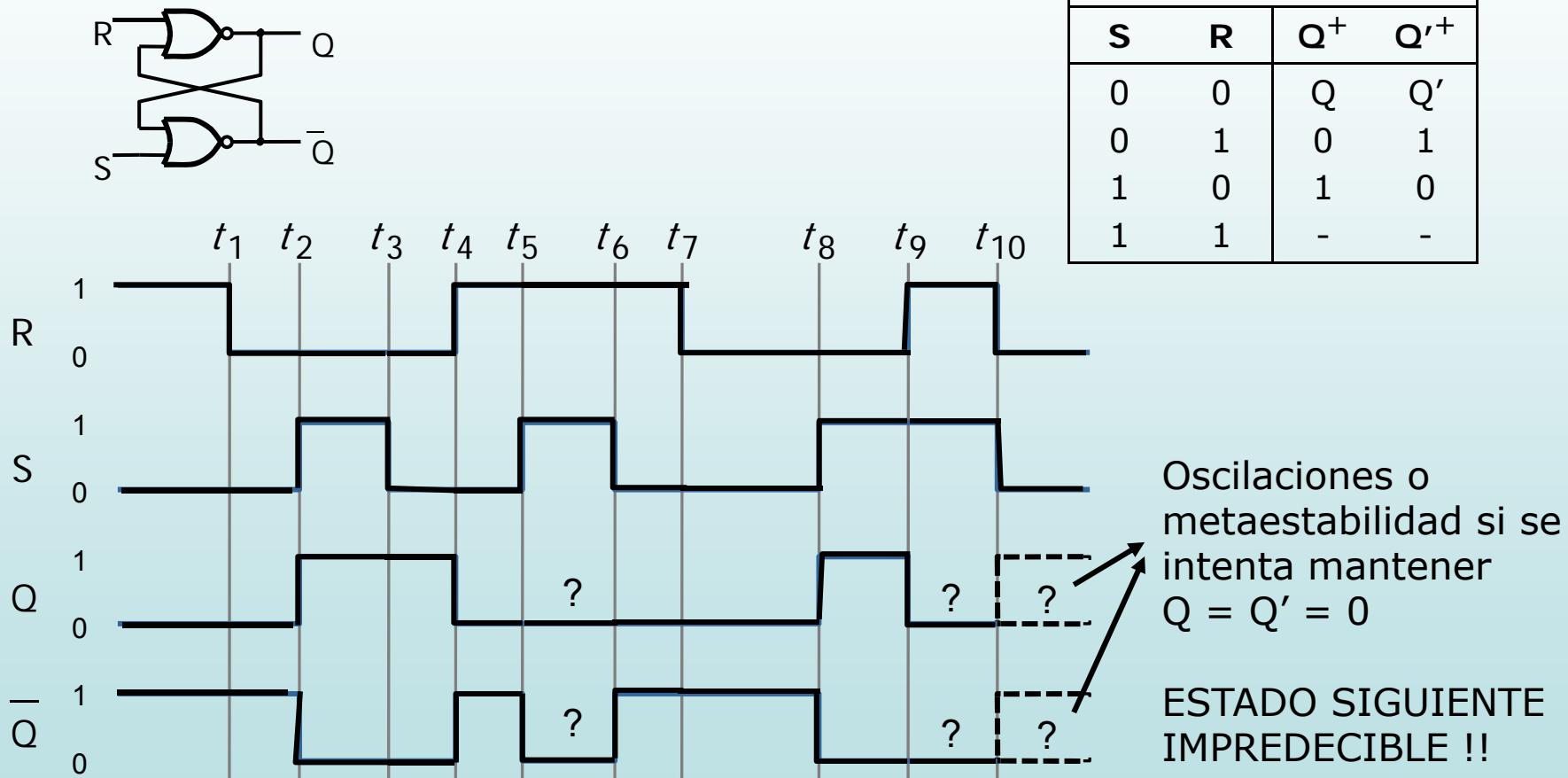
Estado anterior
Puesta a 0
Puesta a 1
No se usa

- **Tabla de excitación:** entradas que hay que proporcionar para obtener un cambio de estado.

Tabla de excitación			
Q ^t	Q ^{t+1}	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

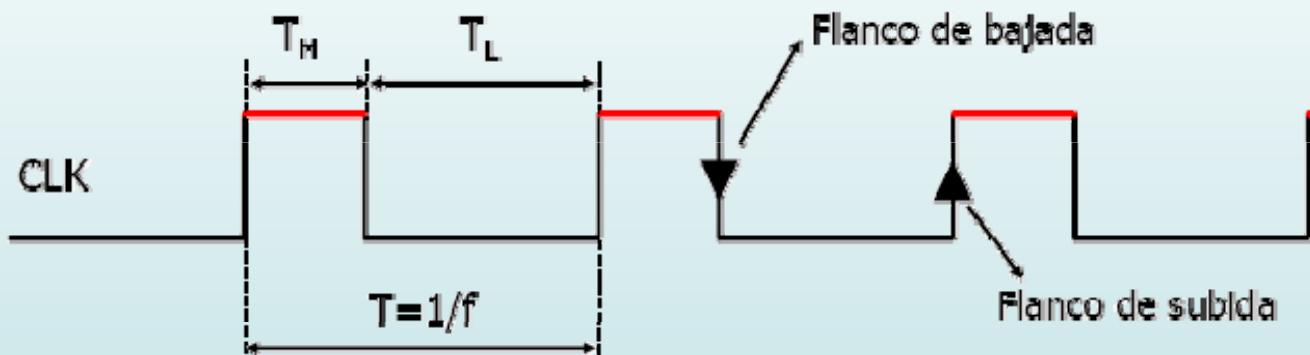
6.2 ELEMENTOS BÁSICOS DE MEMORIA

- Cronograma del Latch SR:



6.2 ELEMENTOS BÁSICOS DE MEMORIA

- Una **señal de reloj** (Clock – Ck - Clk) es una señal cuadrada periódica que se suele utilizar para sincronizar el comportamiento de la mayoría de los sistemas digitales



T = periodo (s)

f = frecuencia (Hz)

T_H = tiempo a nivel alto

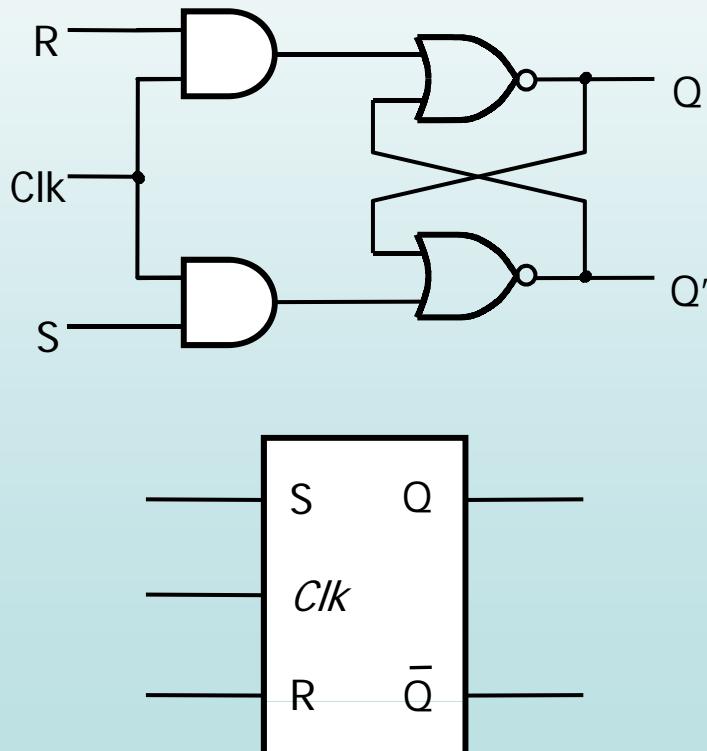
T_L = tiempo a nivel bajo

Rendimiento de ciclo (duty cycle) = $T_H/T \times 100$ (%)



6.2 ELEMENTOS BÁSICOS DE MEMORIA

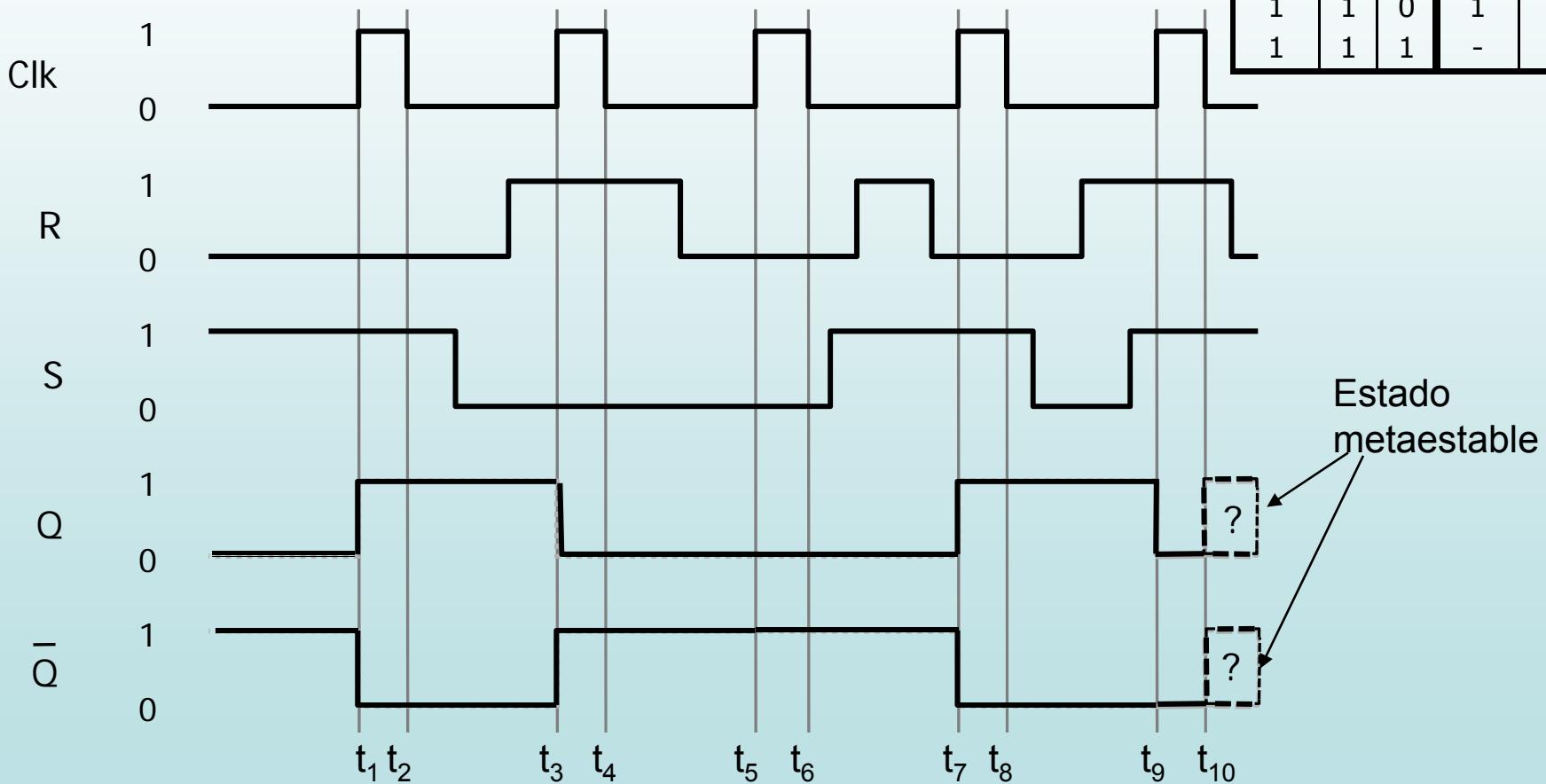
- **Latch SR sincronizado:** se le añade una tercera entrada, Clk, que habilita o inhabilita el funcionamiento del latch (si Clk=1 funciona como un latch SR).



Clk	S	R	Q ⁺	Q' ⁺
0	X	X	Q	Q'
1	0	0	Q	Q'
1	0	1	0	1
1	1	0	1	0
1	1	1	-	-

6.2 ELEMENTOS BÁSICOS DE MEMORIA

- El latch solo cambia cuando $\text{Clk} = 1$.

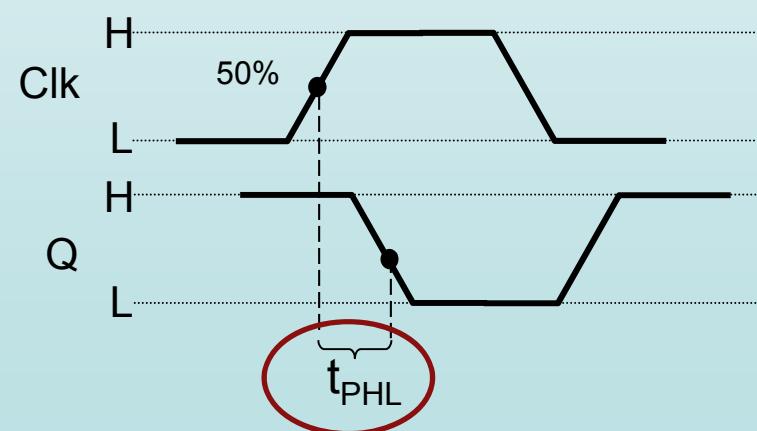
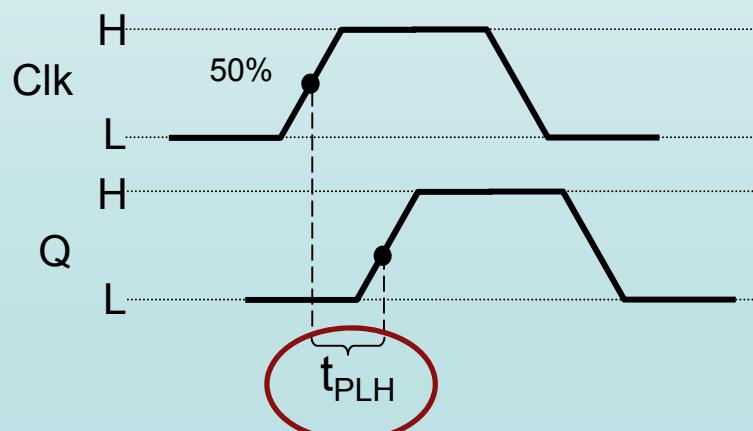


6.2 ELEMENTOS BÁSICOS DE MEMORIA

- Los latches que se habilitan durante todo el tiempo que la señal de control (reloj) vale 1, se denominan latches **disparados por nivel**.
- Son transparentes en todo el tiempo que la señal de habilitación está a 1.
- Funcionan como elementos de memoria sólo después del flanco de bajada de la señal de control, manteniendo el estado determinado por la entrada anterior al flanco de bajada de la señal de control.

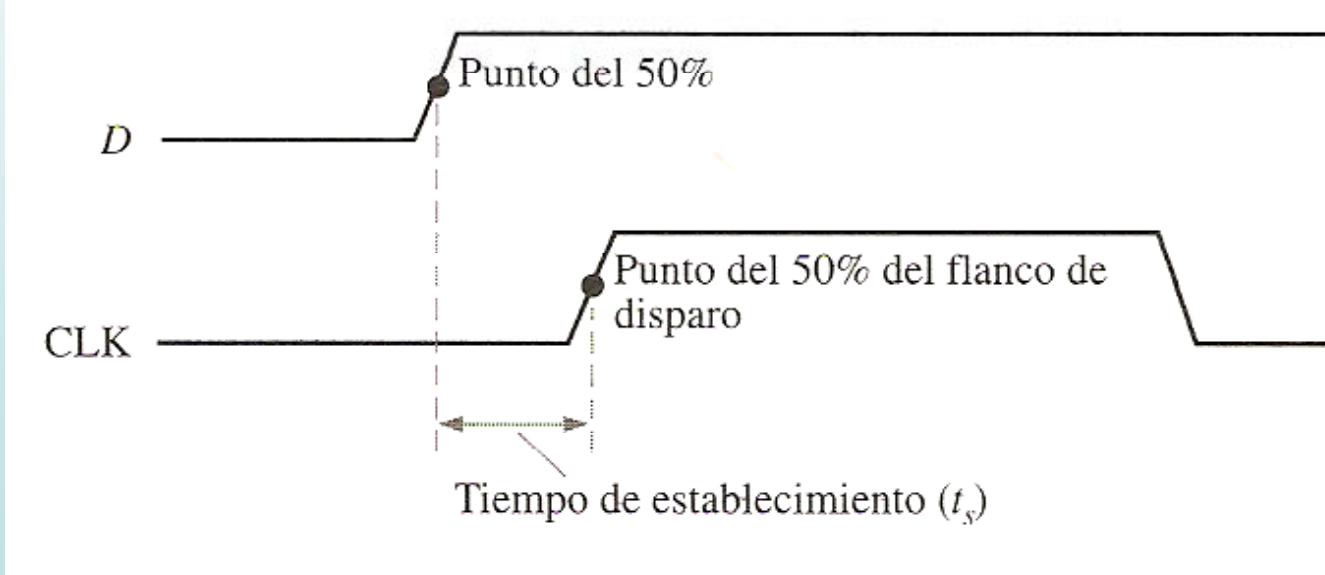
6.2 ELEMENTOS BÁSICOS DE MEMORIA

- **Retardo de propagación, t_p :** tiempo necesario para que se produzca un cambio en la salida tras producirse un cambio en las entradas.
 - **Retardo de propagación t_{PLH}** tiempo desde el disparo de reloj hasta la transición del nivel bajo al alto de la salida ($L \rightarrow H$).
 - **Retardo de propagación t_{PHL}** tiempo desde el disparo de reloj hasta la transición del nivel alto al bajo de la salida ($H \rightarrow L$).



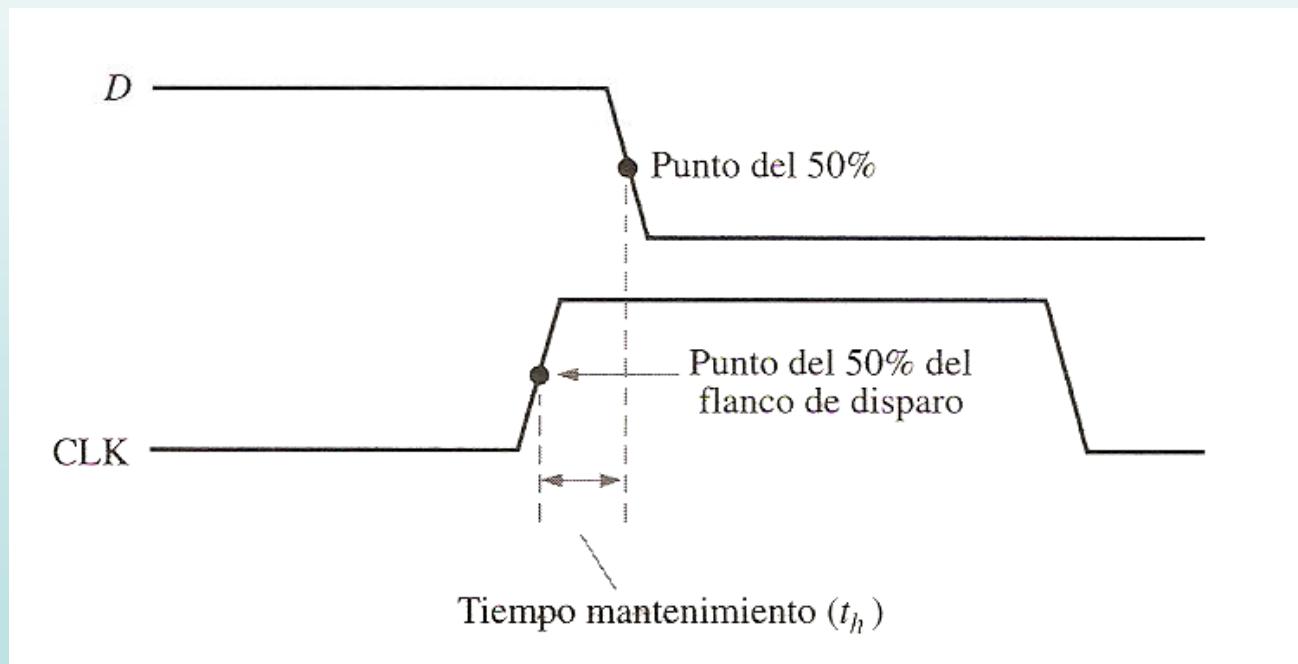
6.2 ELEMENTOS BÁSICOS DE MEMORIA

- **Tiempo de establecimiento** (t_{setup}): intervalo mínimo que los niveles lógicos deben mantenerse constantes en las entradas antes de que llegue el flanco de disparo del reloj.



6.2 ELEMENTOS BÁSICOS DE MEMORIA

- **Tiempo de mantenimiento** (t_h): intervalo mínimo que los niveles lógicos deben mantenerse constantes en las entradas después de que haya pasado el flanco de disparo de la señal de reloj.

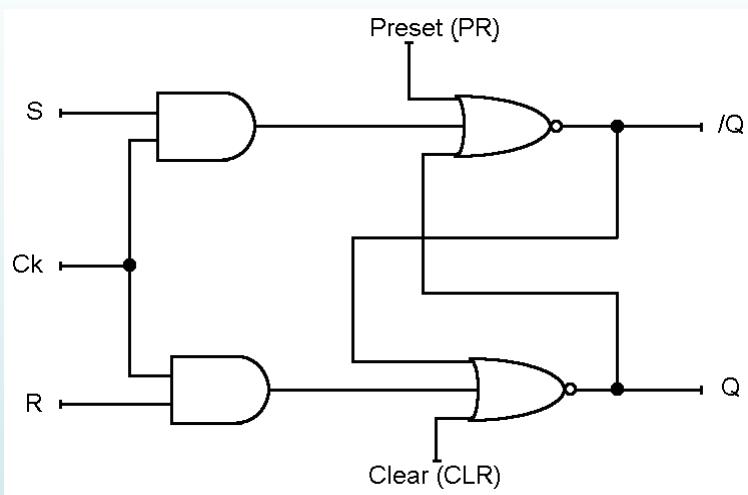


6.2 ELEMENTOS BÁSICOS DE MEMORIA

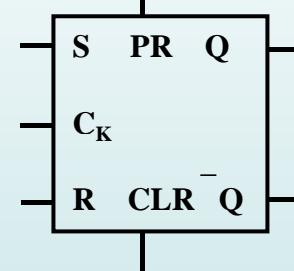
- **Frecuencia máxima de reloj:** velocidad máxima a la que se puede disparar el biestable de manera fiable.
- **Anchura del pulso:** anchura mínima de los impulsos para que funcionen adecuadamente las señales de reloj.
- **Disipación de potencia:** potencia total consumida por el dispositivo.

6.2 ELEMENTOS BÁSICOS DE MEMORIA

Circuito

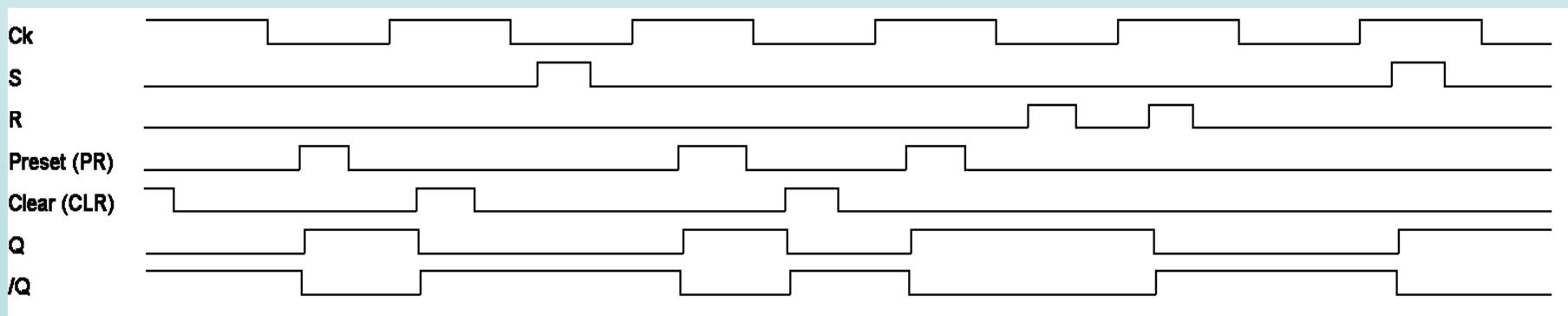


Símbolo



BIESTABLE o
LATCH SR
SÍNCRONO
(Disparado por nivel)
con entradas
ASÍNCRONAS de
Preset y Clear

Cronograma ilustrando su funcionamiento



6.2 ELEMENTOS BÁSICOS DE MEMORIA

Señales de temporización.

La mayoría de los sistemas secuenciales son SÍNCRONOS: los cambios de estado se producen con los flancos de reloj
(p.e. en flanco positivo)

Flanco positivo

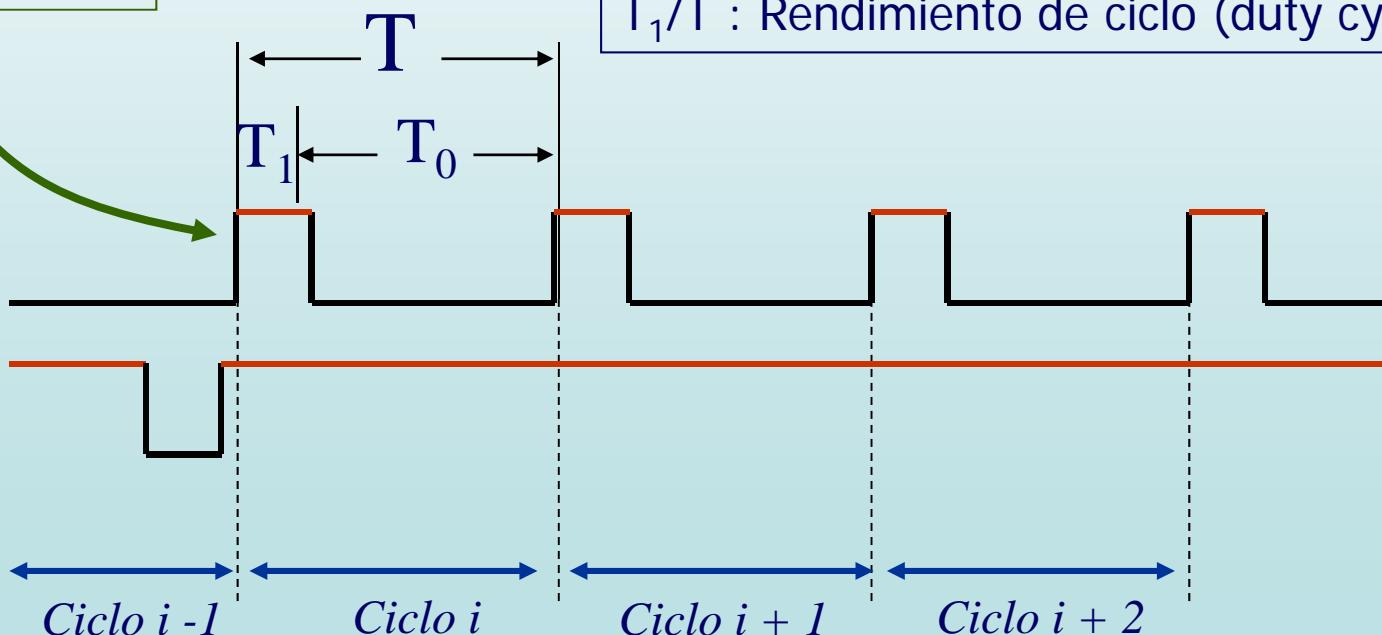
Reloj

/Reset

T : Periodo de reloj

$f = 1/T$: Frecuencia de reloj

T_1/T : Rendimiento de ciclo (duty cycle)

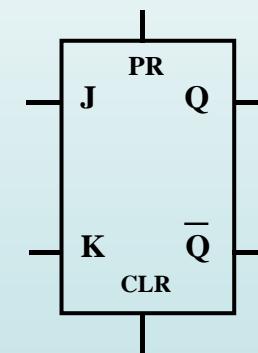


6.2 ELEMENTOS BÁSICOS DE MEMORIA

BIESTABLE JK (Asíncrono)

Definición

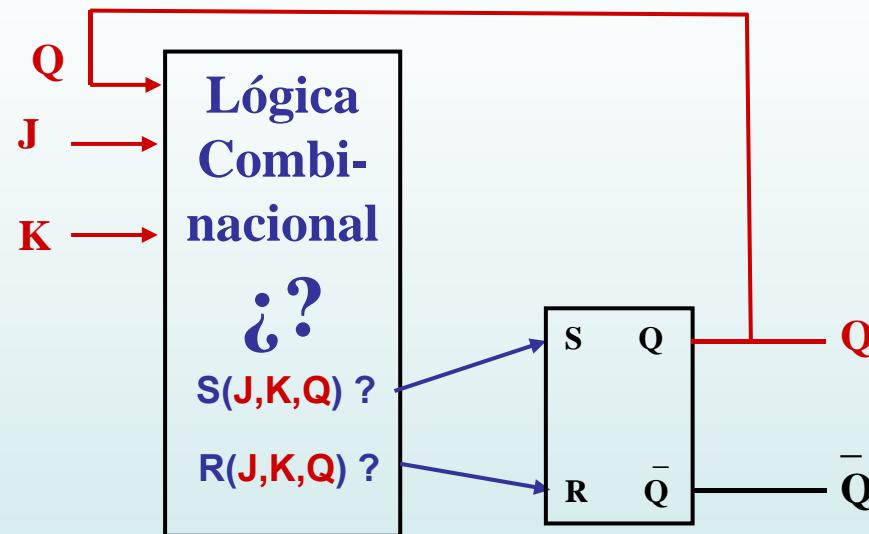
J K	Q^+ (Est. Siguiente)
0 0	Q No cambia
0 1	0 Pone a Cero
1 0	1 Pone a uno
1 1	\bar{Q} Cambia de estado



6.2 ELEMENTOS BÁSICOS DE MEMORIA

BIESTABLE JK (Asíncrono)

Diseño



Definición JK

J K	Q^+ (Est. Siguiente)
0 0	Q No cambia
0 1	0 Pone a Cero
1 0	1 Pone a uno
1 1	\bar{Q} Cambia de estado

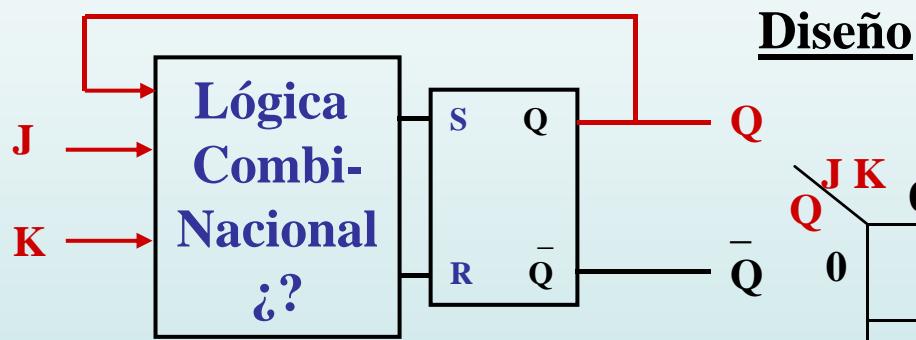
J K Q	Q^+	S R
0 0 0		
0 0 1		
0 1 0		
0 1 1		
1 0 0		
1 0 1		
1 1 0		
1 1 1		

Q Q ⁺	S R
0 0	0 -
0 1	1 0
1 0	0 1
1 1	- 0

6.2 ELEMENTOS BÁSICOS DE MEMORIA

Definición

BIESTABLE JK (Asíncrono)



J K Q	Q ⁺	S R
0 0 0 (0)	0	0 -
0 0 1 (1)	1	- 0
0 1 0 (2)	0	0 -
0 1 1 (3)	0	0 1
1 0 0 (4)	1	1 0
1 0 1 (5)	1	- 0
1 1 0 (6)	1	1 0
1 1 1 (7)	0	0 1

J K	Q ⁺ (Est. Siguiente)
0 0	Q No cambia
0 1	0 Pone a Cero
1 0	1 Pone a uno
1 1	\bar{Q} Cambia de estado

Q		00	01	11	10
0	0	2	6	4	
1	1	3	7	5	

S =

Q		00	01	11	10
0	0	2	6	4	
1	1	3	7	5	

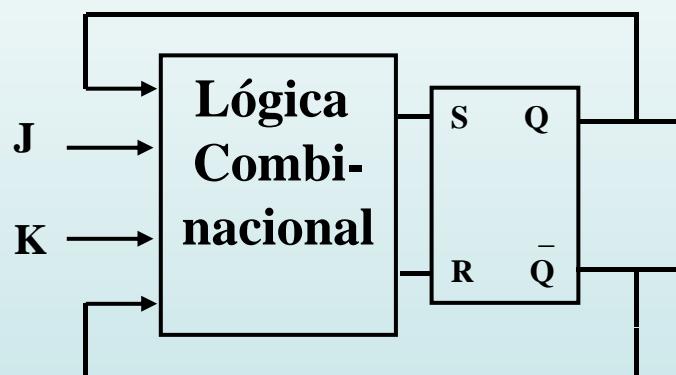
R =

6.2 ELEMENTOS BÁSICOS DE MEMORIA

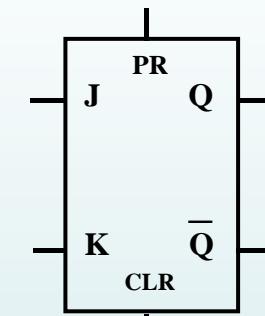
Símbolo del biestable Definición

BIESTABLE JK (Asíncrono)

Diseño



J K Q	Q ⁺	S R
0 0 0	0	0 -
0 0 1	1	- 0
0 1 0	0	0 -
0 1 1	0	0 1
1 0 0	1	1 0
1 0 1	1	- 0
1 1 0	1	1 0
1 1 1	0	0 1



J K	Q ⁺ (Est. Siguiente)
0 0	Q No cambia
0 1	0 Pone a Cero
1 0	1 Pone a uno
1 1	Q̄ Cambia de estado

		J K	00	01	11	10
		Q	0	0	1	1
J K	Q	0	0	0	---	---
0 0	0	0	0	0	1	1
0 1	1	1	1	1	0	0
1 0	0	0	0	0	---	---
1 1	0	0	0	0	---	---

$$S = \bar{Q} \cdot J$$

		J K	00	01	11	10
		Q	0	---	0	0
J K	Q	0	---	0	1	1
0 0	0	0	---	0	0	0
0 1	1	1	---	0	1	1
1 0	0	0	---	0	0	0
1 1	0	0	---	0	0	0

$$R = Q \cdot K$$

6.2 ELEMENTOS BÁSICOS DE MEMORIA

BIESTABLE JK SÍNCRONO
(Disparado por nivel)
con entradas ASÍNCRONAS
de Preset y Clear

TABLA TRANSICIÓN
COMPLETA

J K Q	Q ⁺
0 0 0	0
0 0 1	1
0 1 0	0
0 1 1	0
1 0 0	1
1 0 1	1
1 1 0	1
1 1 1	0

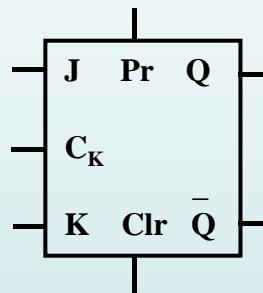


TABLA ABREVIADA

J K	Q ⁺ (Est. Siguiente)
0 0	Q No cambia
0 1	0 Pone a Cero
1 0	1 Pone a uno
1 1	\bar{Q} Cambia de estado

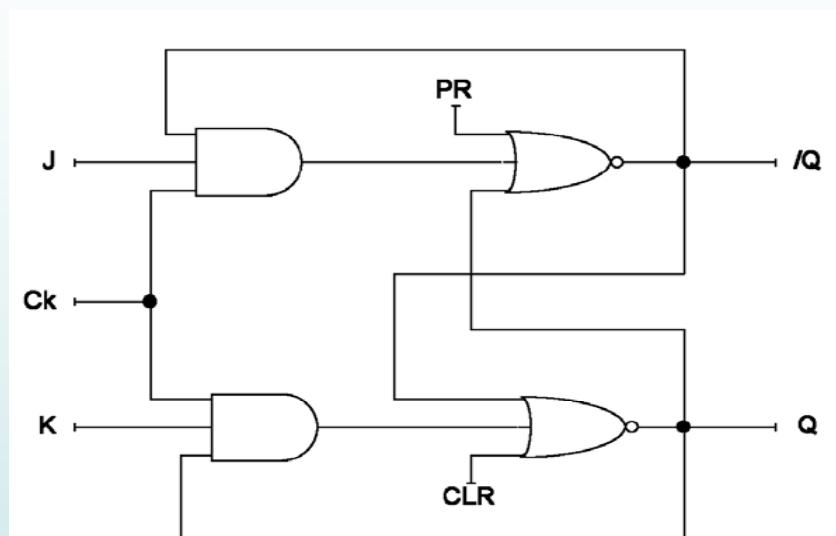


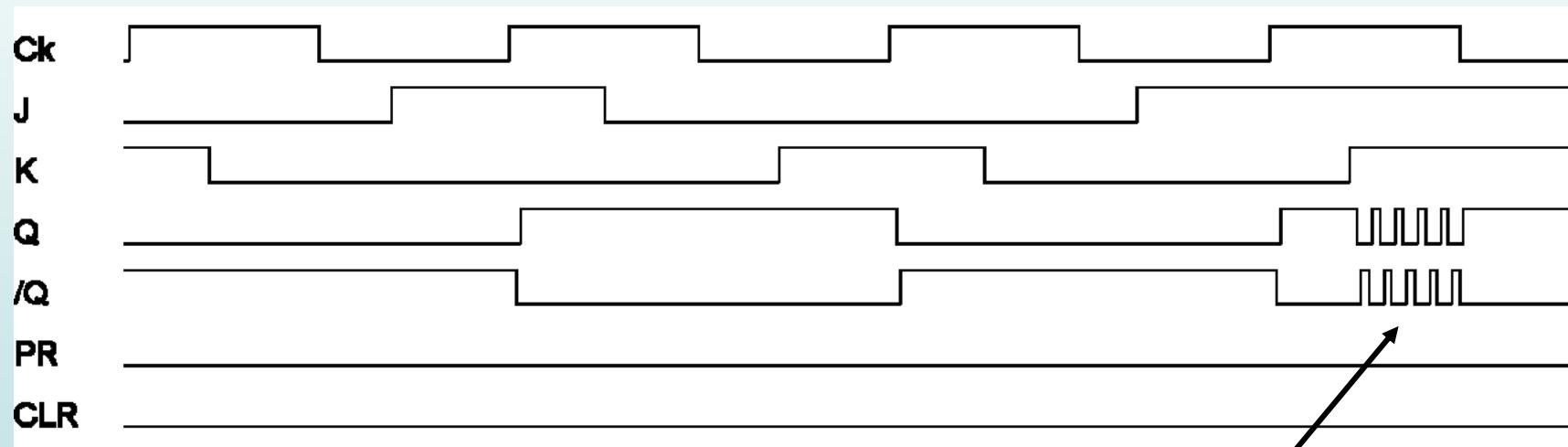
TABLA EXCITACIÓN Ó
INVERSA

Q Q ⁺	J K
0 0	0 -
0 1	1 -
1 0	- 1
1 1	- 0

6.2 ELEMENTOS BÁSICOS DE MEMORIA

PROBLEMAS CON LOS BIESTABLES DISPARADOS POR NIVEL

Ejemplo del funcionamiento del biestable JK disparado por nivel

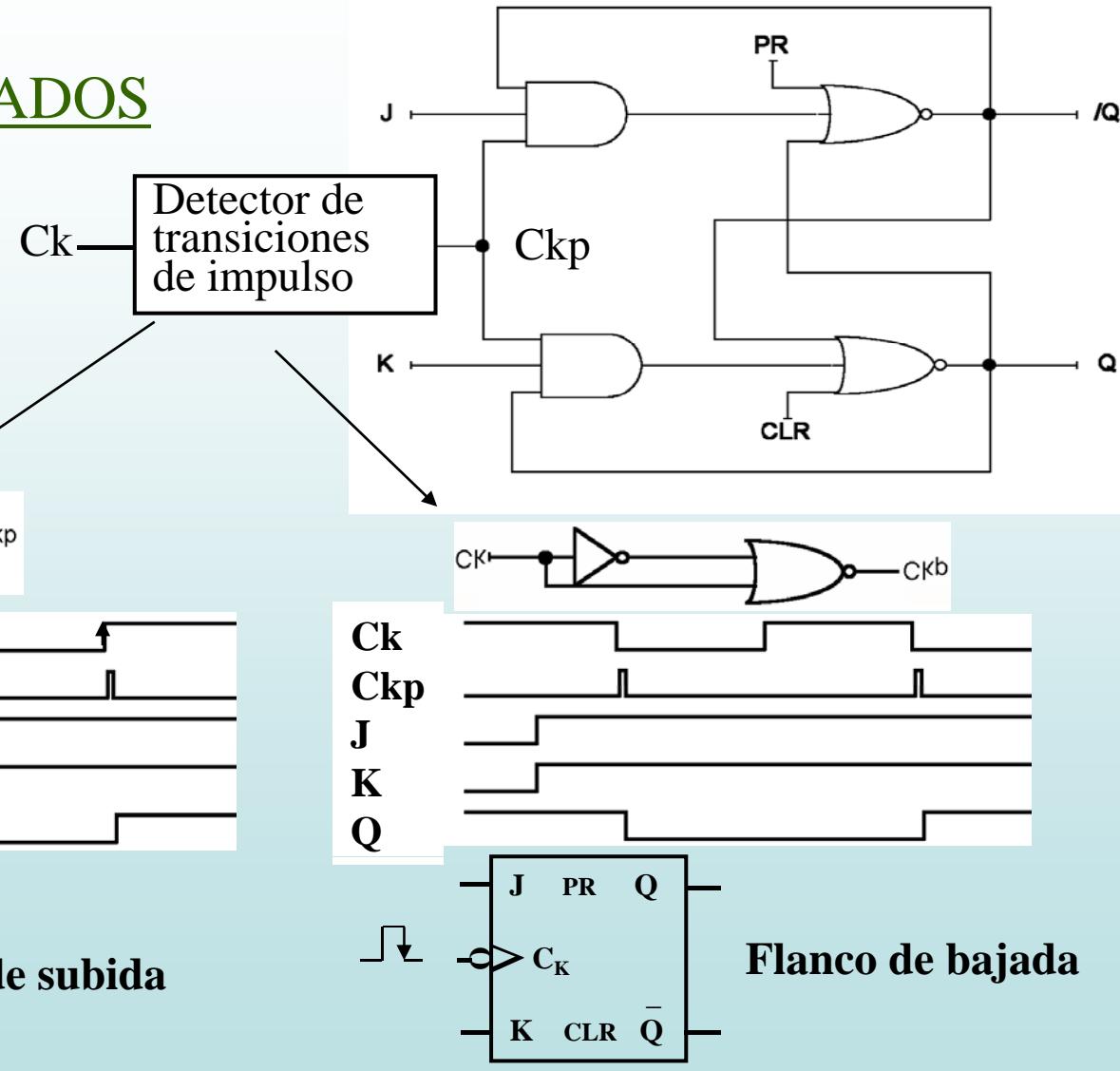


PROBLEMA

6.2 ELEMENTOS BÁSICOS DE MEMORIA

BIESTABLES DISARADOS POR FLANCO

Ejemplos sencillos que se valen de los azares. (Problema con los retardos para controlar la anchura del pulso Ckp)

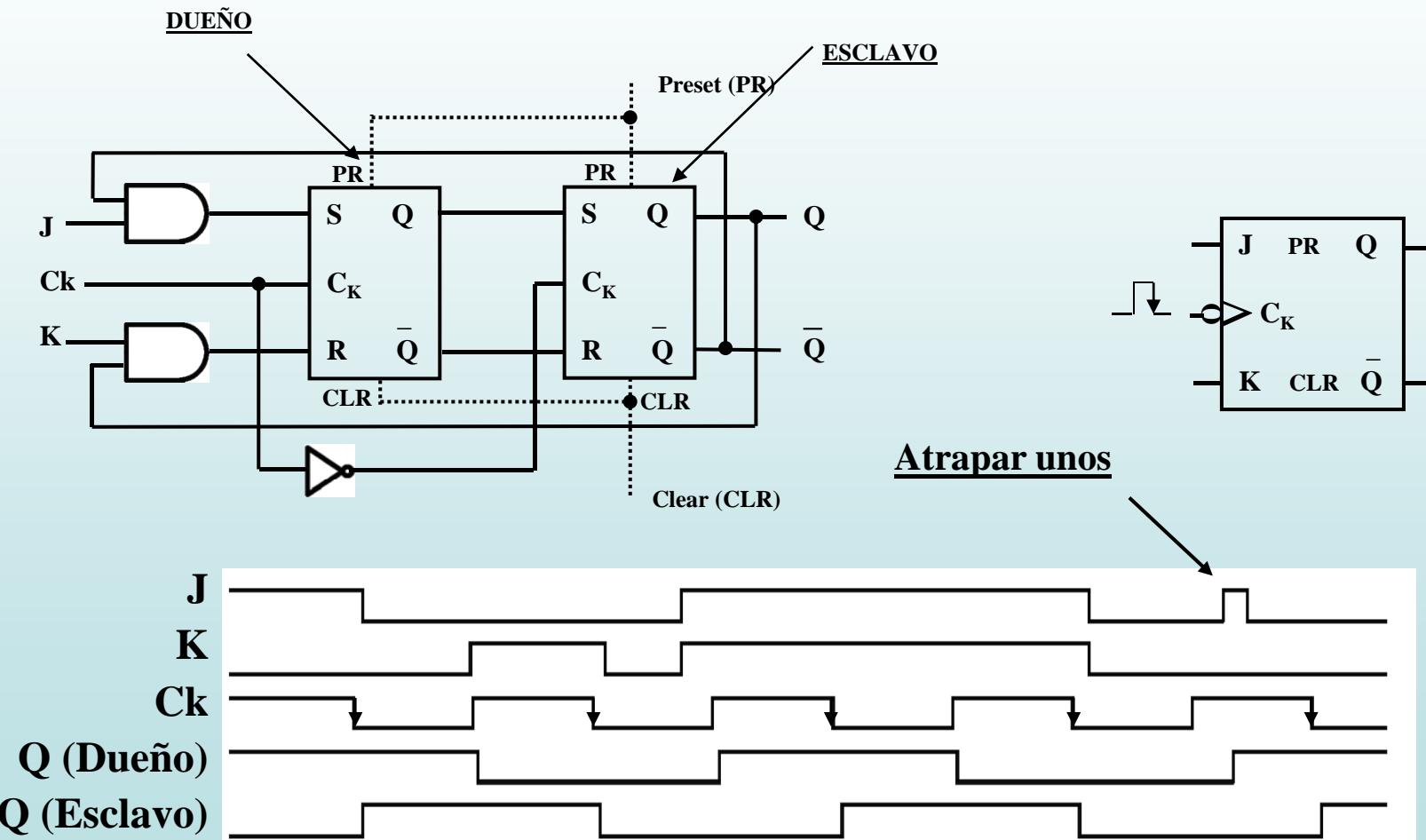


6.2 ELEMENTOS BÁSICOS DE MEMORIA

- Un **biestable maestro-esclavo** se implementa utilizando dos biestables disparados por nivel de forma que el maestro está controlado por la señal de reloj y el esclavo por la señal de reloj complementada (nunca están habilitados simultáneamente).
- El **valor de la entrada** se capta en el biestable maestro antes del flanco de subida de la señal de reloj y se transmite al biestable esclavo después de ese flanco de subida.
- Un **flip-flop disparado por flanco** sólo cambia en el flanco de subida (o positivo) o de bajada (o negativo) de la señal de reloj y sus entradas de datos no deben cambiar después del t_{setup} anterior, ni antes del t_{hold} posterior, al flanco de la señal de reloj.

6.2 ELEMENTOS BÁSICOS DE MEMORIA

BIESTABLE “Dueño-Esclavo” (Master-slave)



6.2 ELEMENTOS BÁSICOS DE MEMORIA

- **LATCH D** sincronizado (con entrada de reloj) activo por nivel alto.

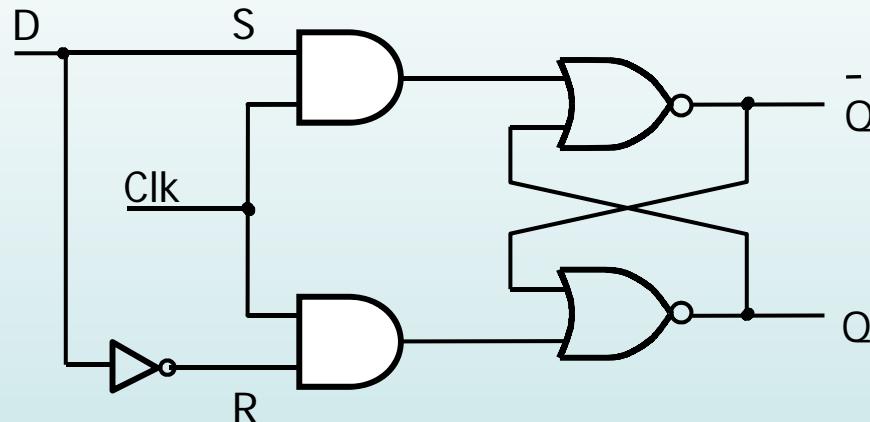


Tabla de estados		
D	Q^+	Q'^+
0	0	1
1	1	0

Puesta a 0
Puesta a 1

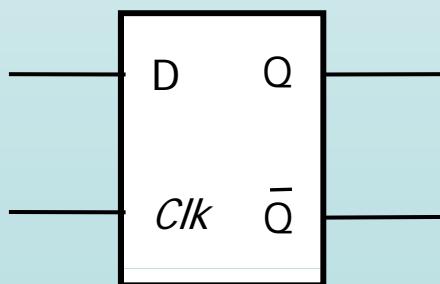
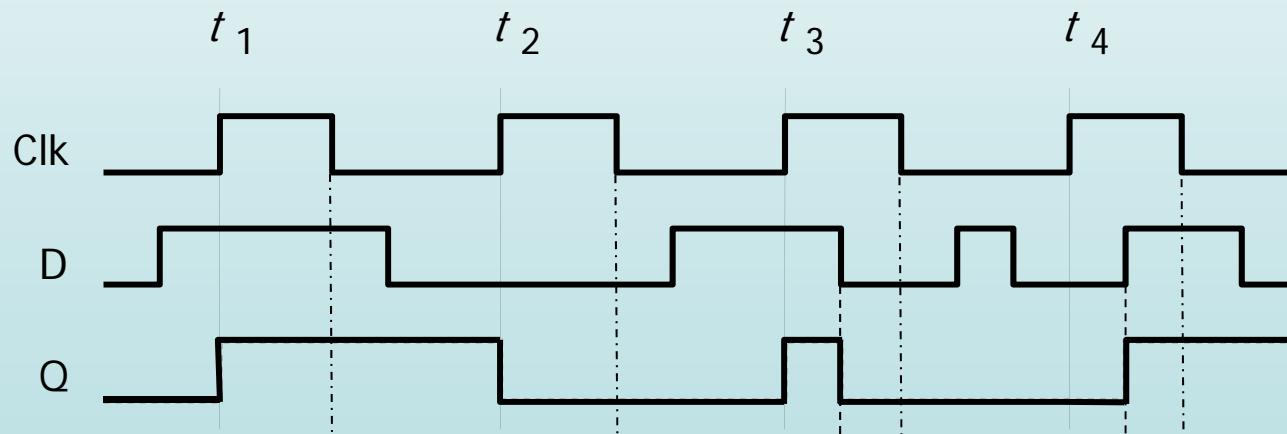
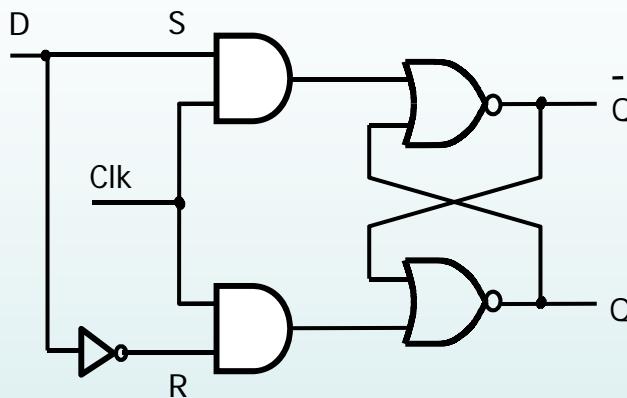


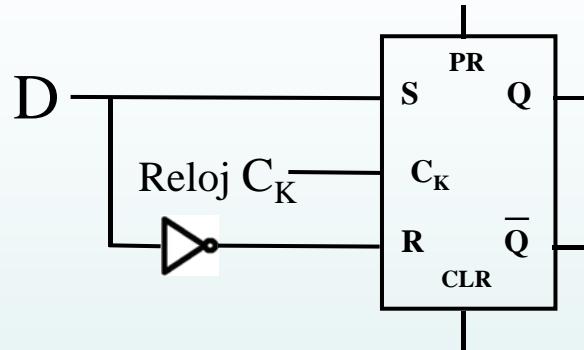
Tabla de excitación		
Q	Q^+	D
0	0	0
0	1	1
1	0	0
1	1	1

6.2 ELEMENTOS BÁSICOS DE MEMORIA

- Cronograma del latch D:



6.2 ELEMENTOS BÁSICOS DE MEMORIA



Biestable tipo D
(Realizado con un
biestable SR) activo por
nivel alto.

TABLA COMPLETA

D	Q	Q^+
0	0	0
0	1	0
1	0	1
1	1	1

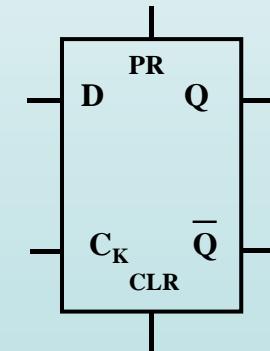
TABLA ABREVIADA

D	Q^+
0	0
1	1

TABLA INVERSA

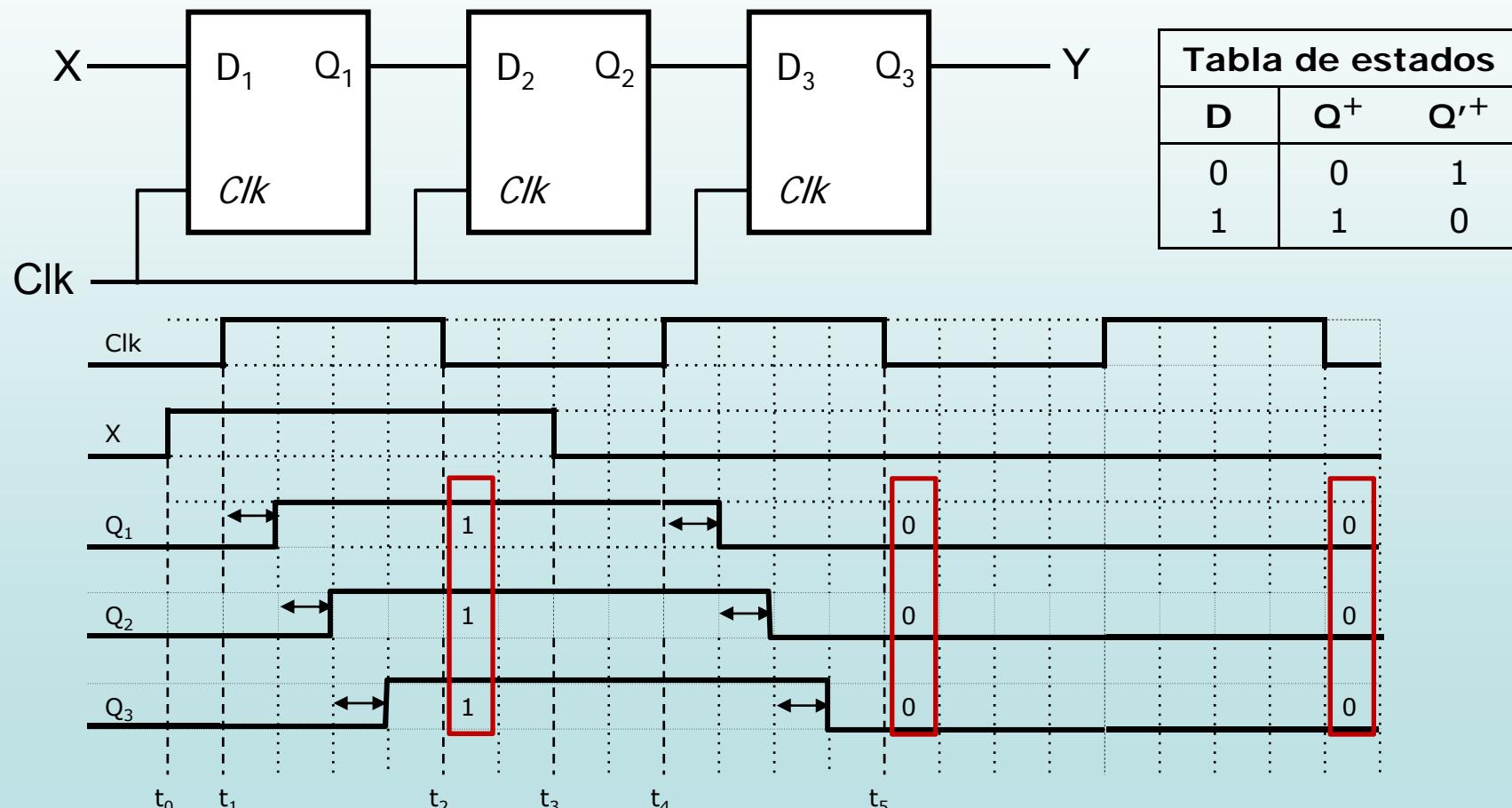
Q^+	D
0	0
1	1

SÍMBOLO



6.2 ELEMENTOS BÁSICOS DE MEMORIA

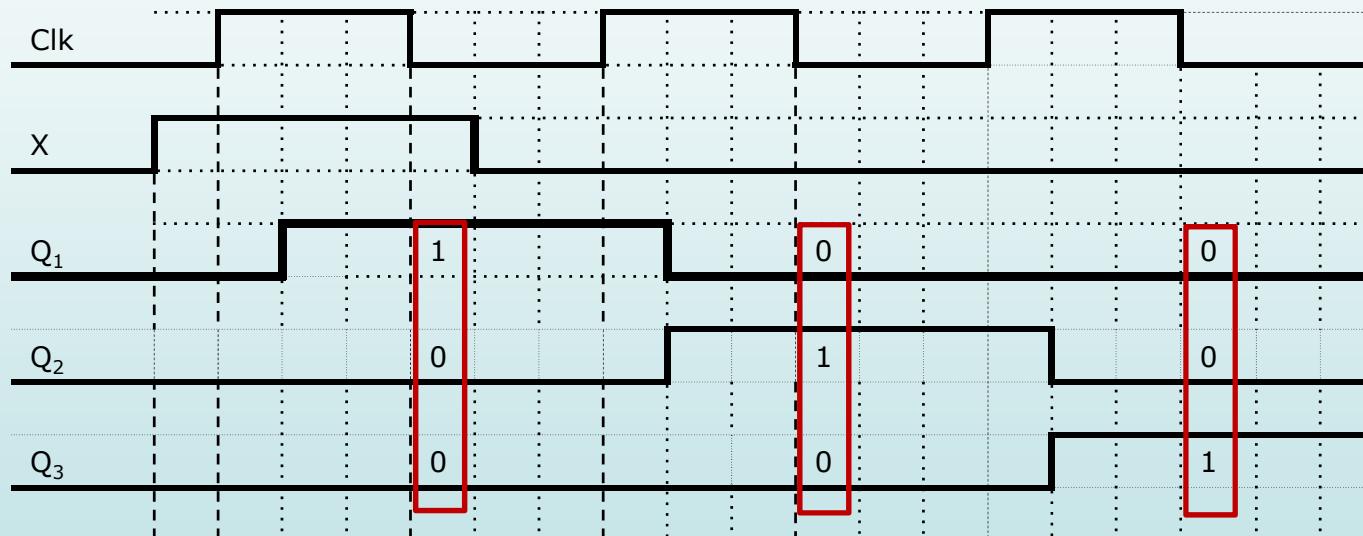
- **Ejemplo:** registro de desplazamiento con latch D:
En teoría: X se transferirá por todos los latch.



6.2 ELEMENTOS BÁSICOS DE MEMORIA

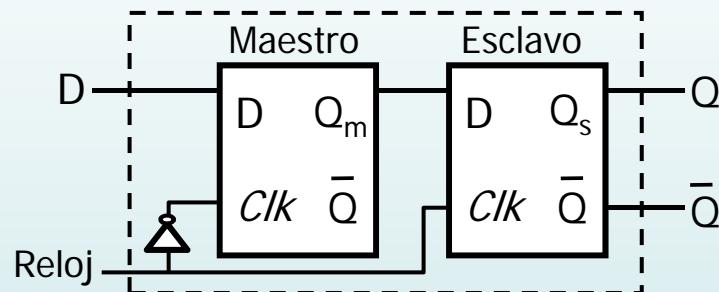
- **Soluciones:**

- Disminuir el ancho de pulso de reloj

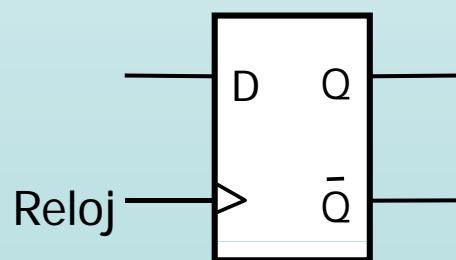


6.2 ELEMENTOS BÁSICOS DE MEMORIA

- Biestables maestro-esclavo (master-slave flip-flop)

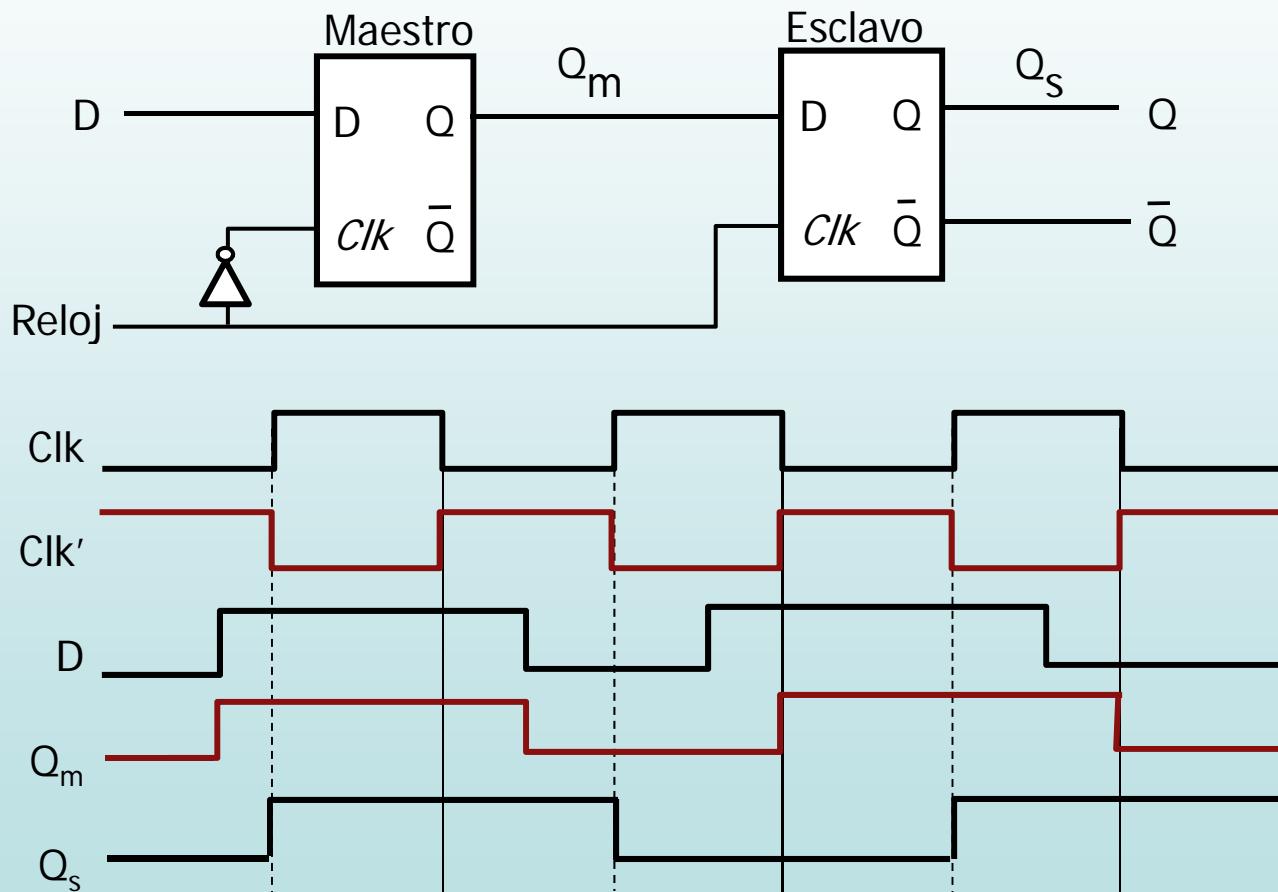


- Biestables disparados por flanco (edge triggered flip-flop)



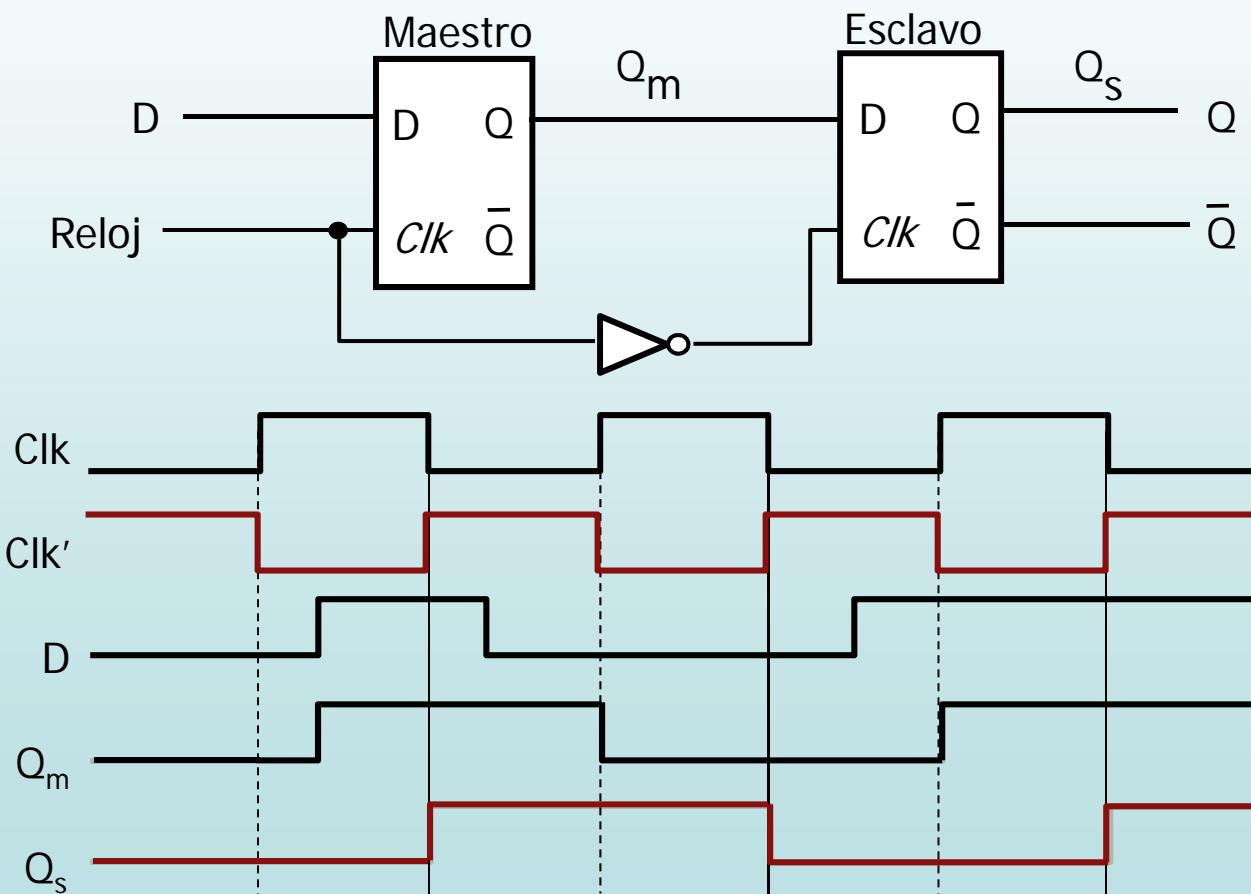
6.2 ELEMENTOS BÁSICOS DE MEMORIA

- Biestable maestro-esclavo tipo D (master-slave flip-flop)



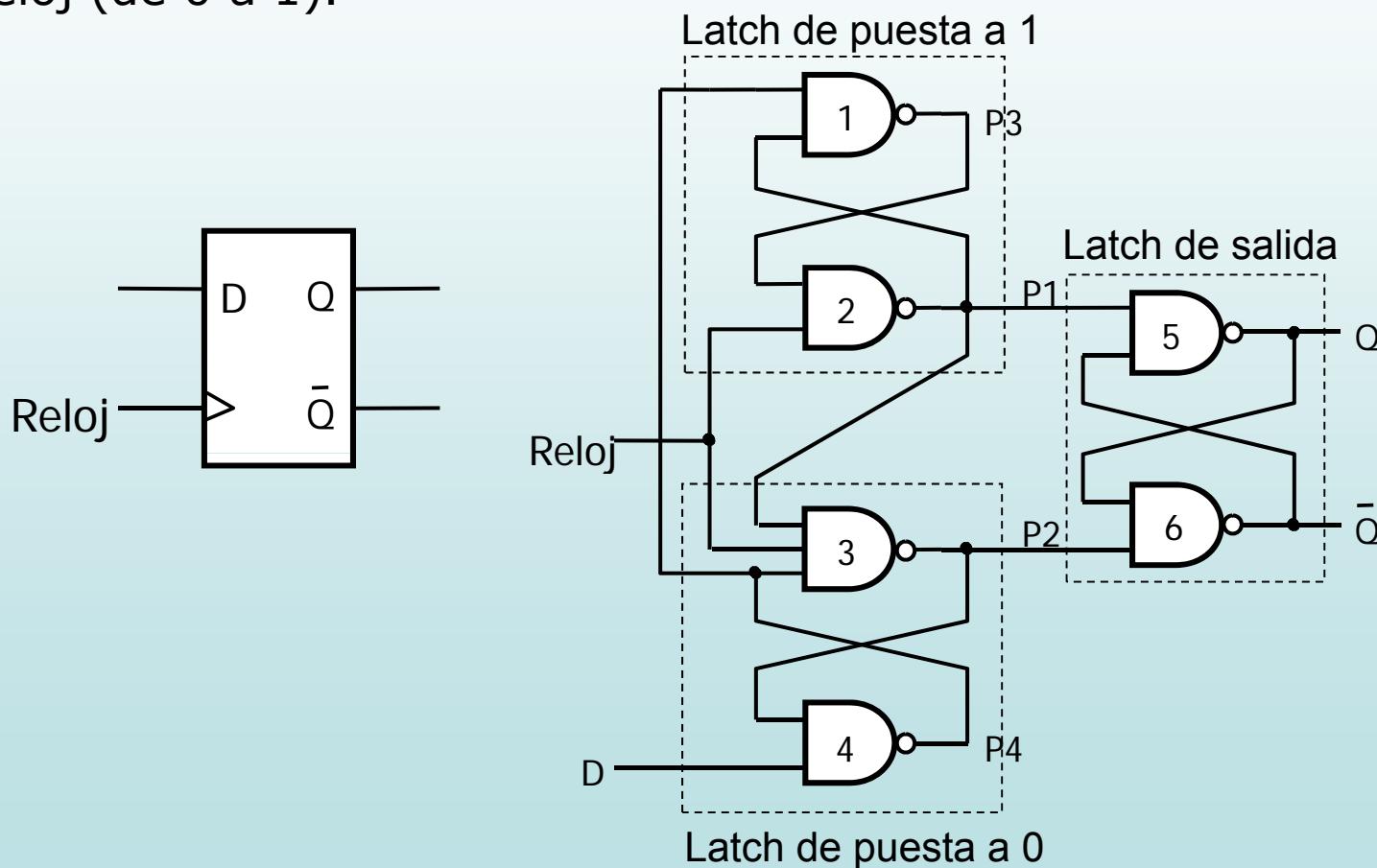
6.2 ELEMENTOS BÁSICOS DE MEMORIA

- **Biestable maestro-esclavo tipo D**, disparado por flanco negativo: La salida cambia cuando se habilita el “esclavo” (cuando el reloj pasa a valer cero)



6.2 ELEMENTOS BÁSICOS DE MEMORIA

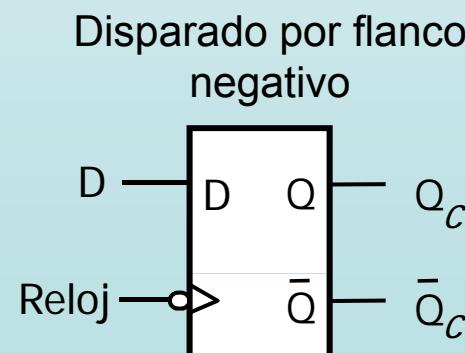
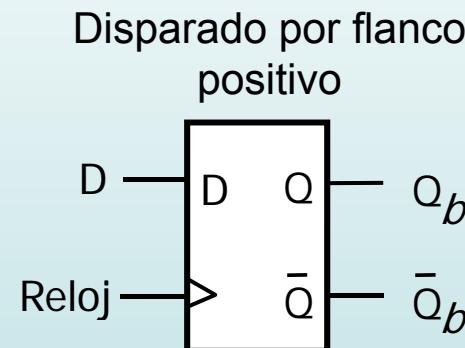
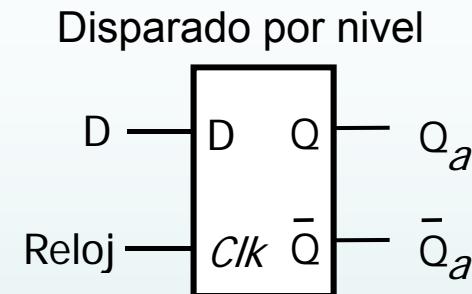
- **Flip-flop D disparado por flanco** (edge triggered flip-flop): La salida cambia sólo durante las transiciones positivas de reloj (de 0 a 1).



6.2 ELEMENTOS BÁSICOS DE MEMORIA

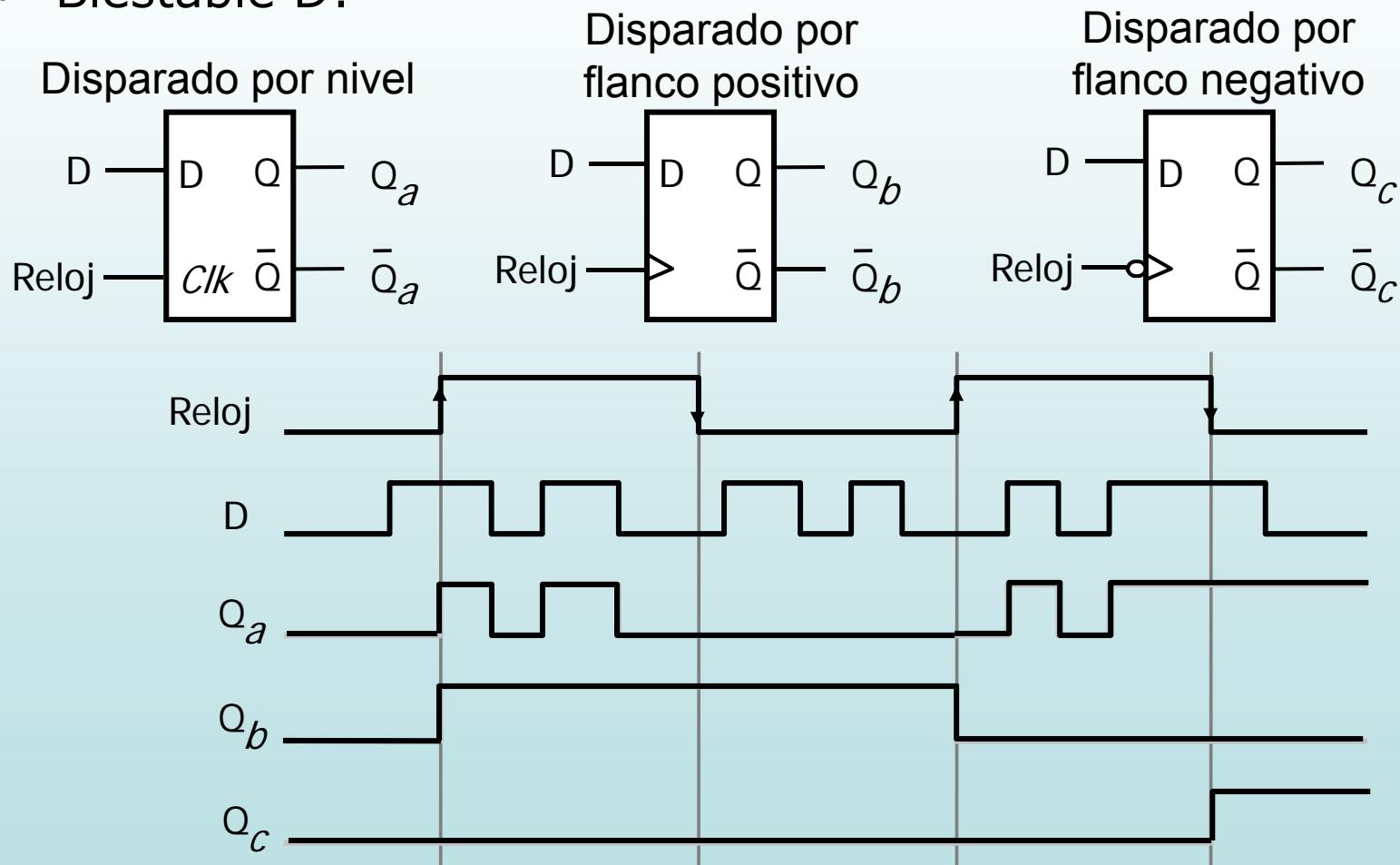
Biestable D:

- **disparado por nivel:** cambia con el flanco de subida y sigue a D (es transparente) mientras el Reloj valga 1
- **disparado por flanco positivo:** cambia con el flanco de subida y toma el valor de D en ese instante hasta el siguiente flanco de subida.
- **disparado por flanco negativo:** cambia con el flanco de bajada y toma el valor de D en ese instante hasta el siguiente flanco de bajada.



6.2 ELEMENTOS BÁSICOS DE MEMORIA

- Biestable D:



6.2 ELEMENTOS BÁSICOS DE MEMORIA

BIESTABLE JK SÍNCRONO
(Disparado por nivel)
con entradas ASÍNCRONAS
de Preset y Clear

TABLA TRANSICIÓN
COMPLETA

J K Q	Q ⁺
0 0 0	0
0 0 1	1
0 1 0	0
0 1 1	0
1 0 0	1
1 0 1	1
1 1 0	1
1 1 1	0

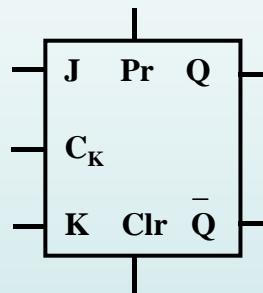


TABLA ABREVIADA

J K	Q ⁺ (Est. Siguiente)
0 0	Q No cambia
0 1	0 Pone a Cero
1 0	1 Pone a uno
1 1	\bar{Q} Cambia de estado

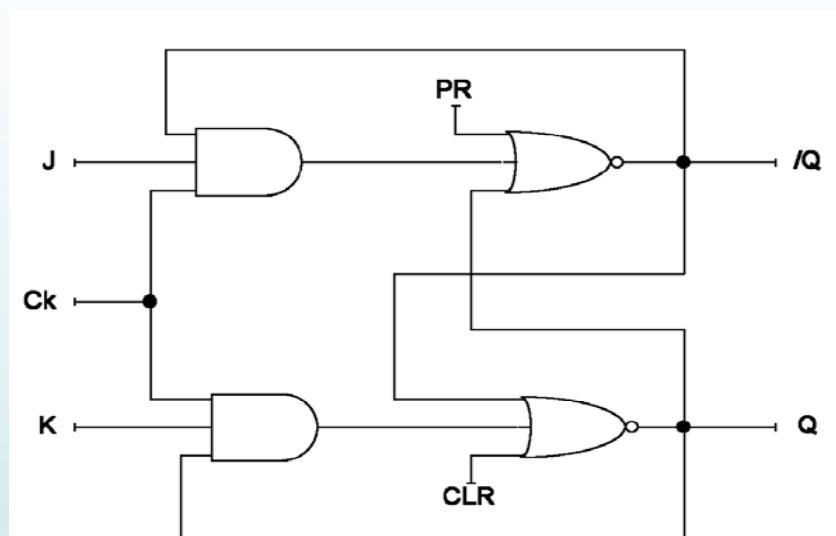


TABLA EXCITACIÓN Ó
INVERSA

Q Q ⁺	J K
0 0	0 -
0 1	1 -
1 0	- 1
1 1	- 0

6.2 ELEMENTOS BÁSICOS DE MEMORIA

BIESTABLE JK SÍNCRONO
(Disparado por flanco positivo)
con entradas ASÍNCRONAS
de Preset y Clear

TABLA TRANSICIÓN
COMPLETA

J K Q	Q ⁺
0 0 0	0
0 0 1	1
0 1 0	0
0 1 1	0
1 0 0	1
1 0 1	1
1 1 0	1
1 1 1	0

TABLA ABREVIADA

J K	Q ⁺ (Est. Siguiente)
0 0	Q No cambia
0 1	0 Pone a Cero
1 0	1 Pone a uno
1 1	Q Cambia de estado

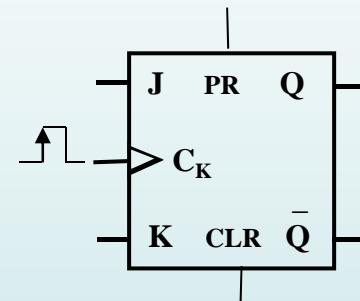
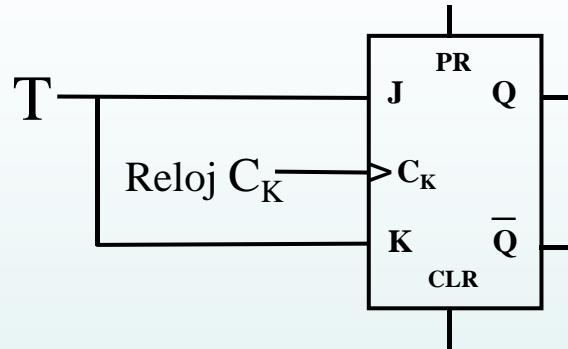


TABLA EXCITACIÓN Ó
INVERSA

Q Q ⁺	J K
0 0	0 -
0 1	1 -
1 0	- 1
1 1	- 0

6.2 ELEMENTOS BÁSICOS DE MEMORIA



Biestable tipo T
(Realizado con un
biestable JK) activo por
flanco de subida.

TABLA COMPLETA

T	Q	Q^+
0	0	0
0	1	1
1	0	1
1	1	0

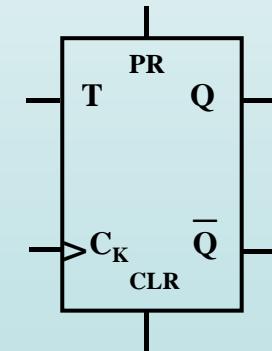
TABLA ABREVIADA

T	Q^+ (Est. Siguiente)
0	Q No cambia
1	\bar{Q} Cambia Estado

TABLA INVERSA

Q	Q^+	T
0	0	0
0	1	1
1	0	1
1	1	0

SÍMBOLO



6.2 ELEMENTOS BÁSICOS DE MEMORIA

- Ejemplo: implementar un FF-D
 - A partir de un FF-JK

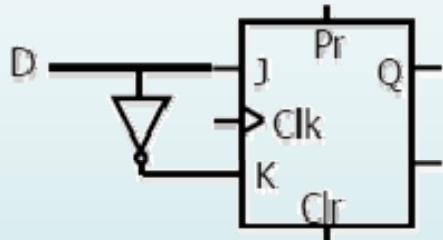


Tabla de estados		
J	K	Q^+
0	0	Q
0	1	0
1	0	1
1	1	Q'

Tabla de estados	
D	Q^+
0	0
1	1

D	$J = D$	$K = D'$	Q^+
0	0	1	0
1	1	0	1

- b) A partir de un FF-T

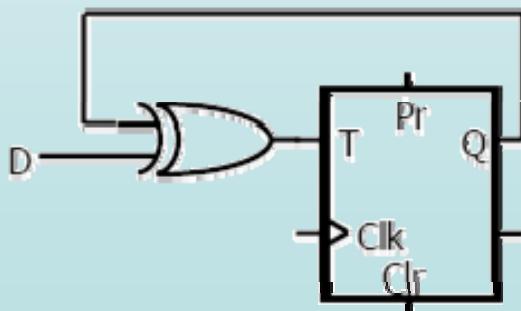


Tabla de estados	
T	Q^+
0	Q
1	Q'

D	Q	$T = D \oplus Q$	Q^+
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

6.2 ELEMENTOS BÁSICOS DE MEMORIA

- Ejemplo: implementar un FF-T

a) A partir de un FF-JK

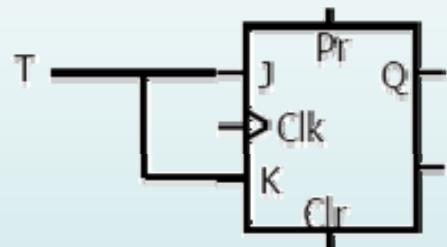


Tabla de estados		
J	K	Q^+
0	0	Q
0	1	0
1	0	1
1	1	Q'

Tabla de estados	
T	Q^+
0	Q
1	Q'

T	$J = T$	$K = T$	Q^+
0	0	0	Q
1	1	1	Q'

b) A partir de un FF-D

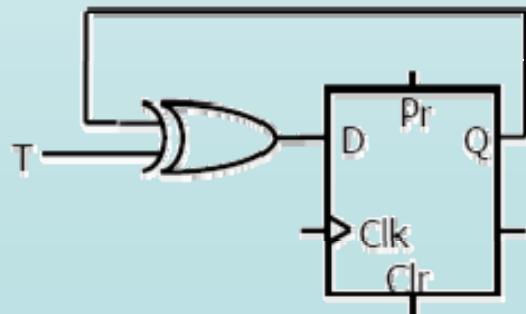
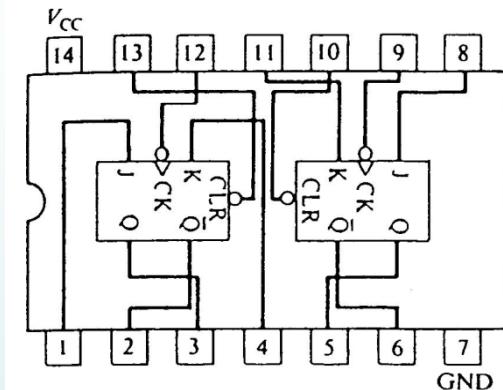
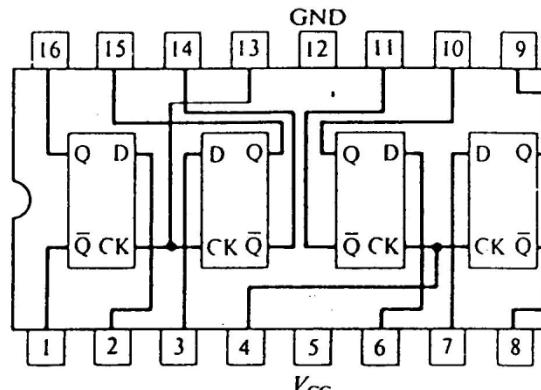
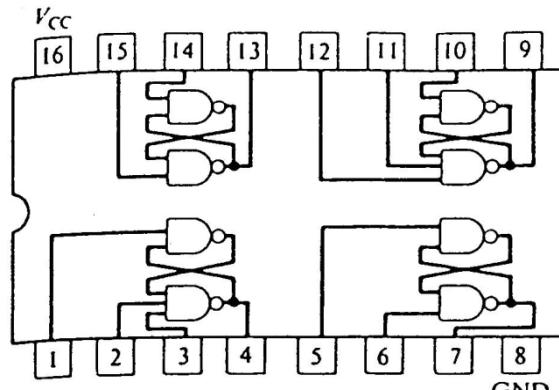


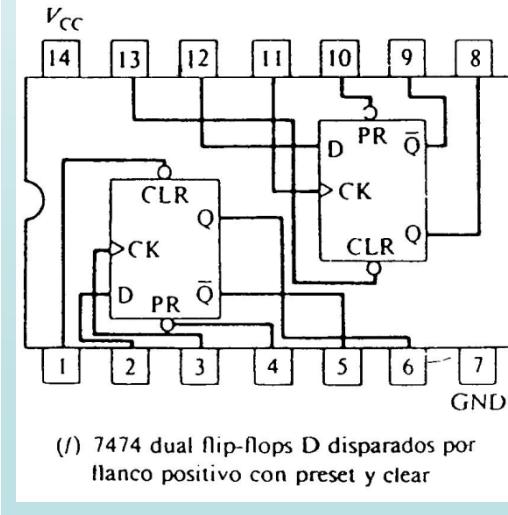
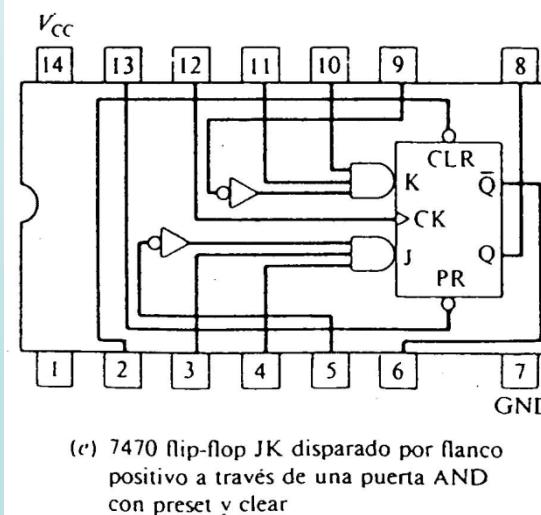
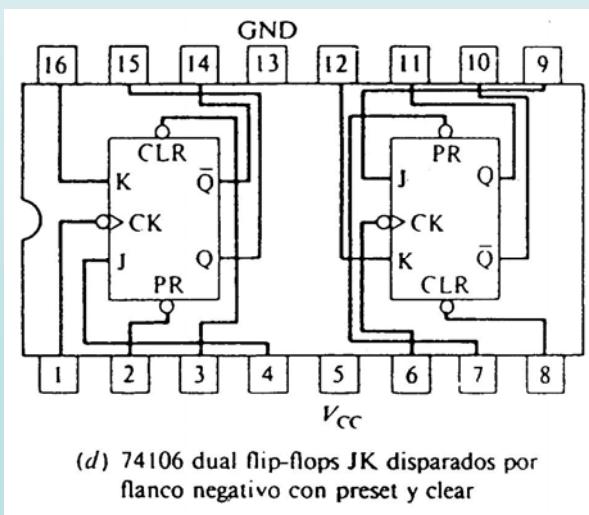
Tabla de estados	
D	Q^+
0	0
1	1

T	Q	$D = T \oplus Q$	Q^+
0	0	0	$0 = Q$
0	1	1	$1 = Q$
1	0	1	$1 = Q'$
1	1	0	$0 = Q'$

6.2 ELEMENTOS BÁSICOS DE MEMORIA

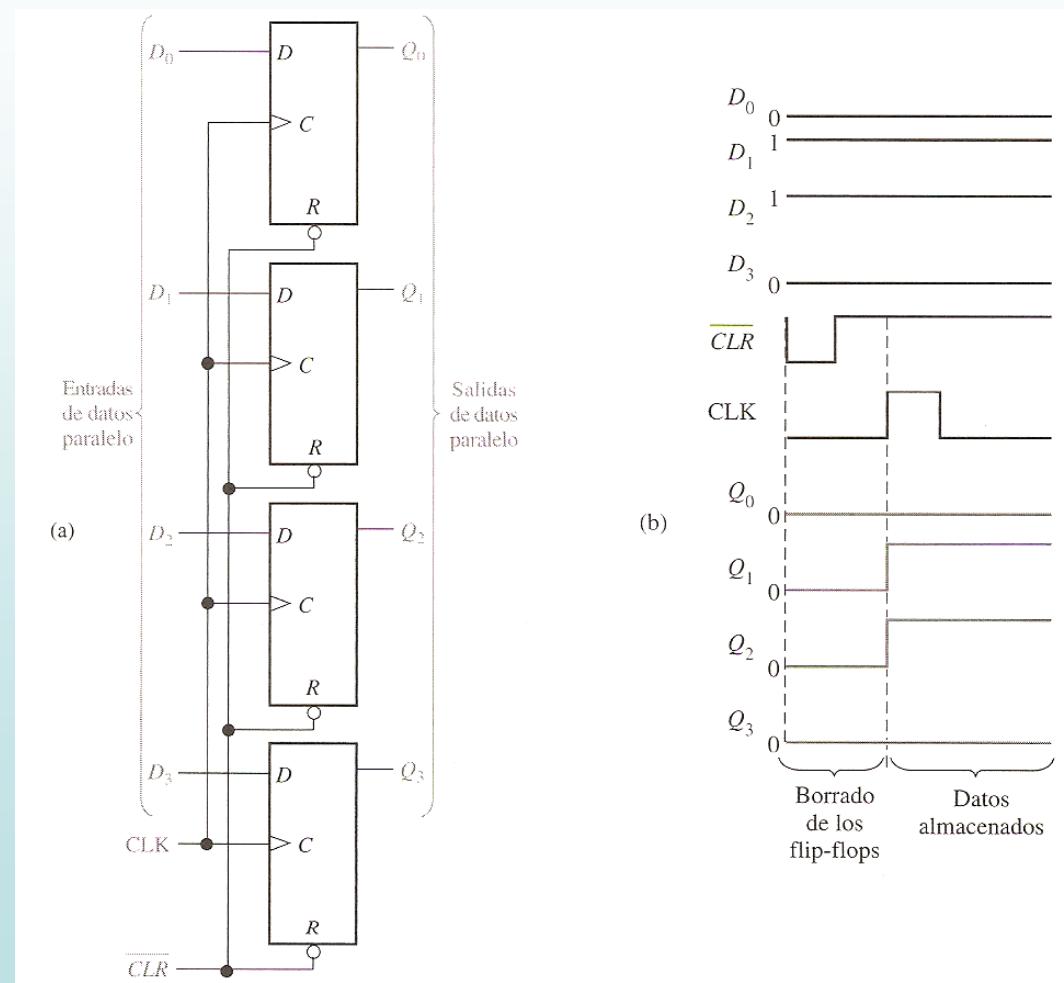


EJEMPLOS DE BIESTABLES (FLIP-FLOPs) COMO C.I.



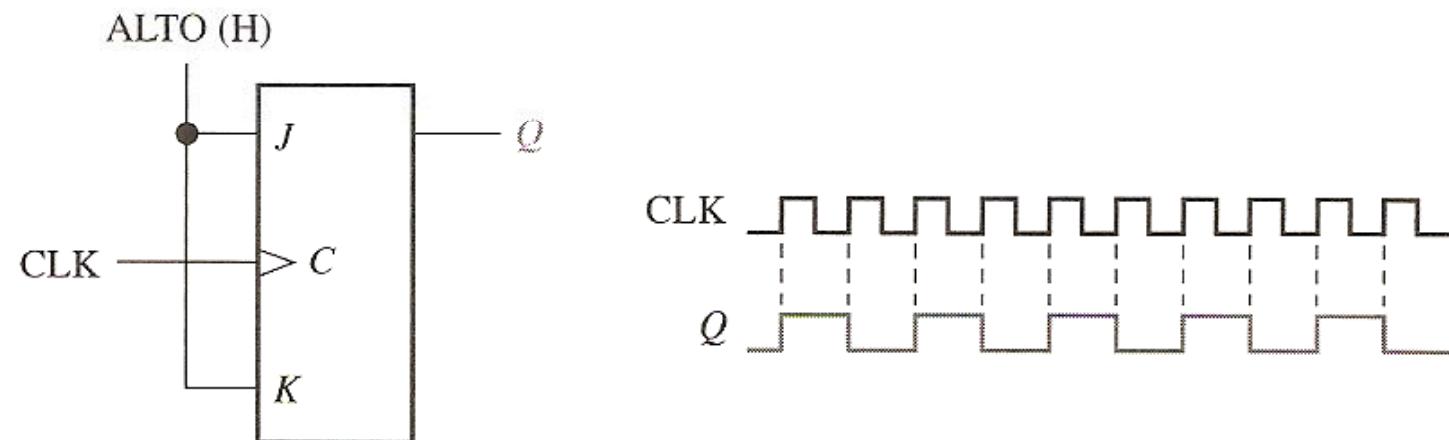
6.2 ELEMENTOS BÁSICOS DE MEMORIA: Aplicaciones

- Almacenamiento de datos en paralelo (registro sencillo)



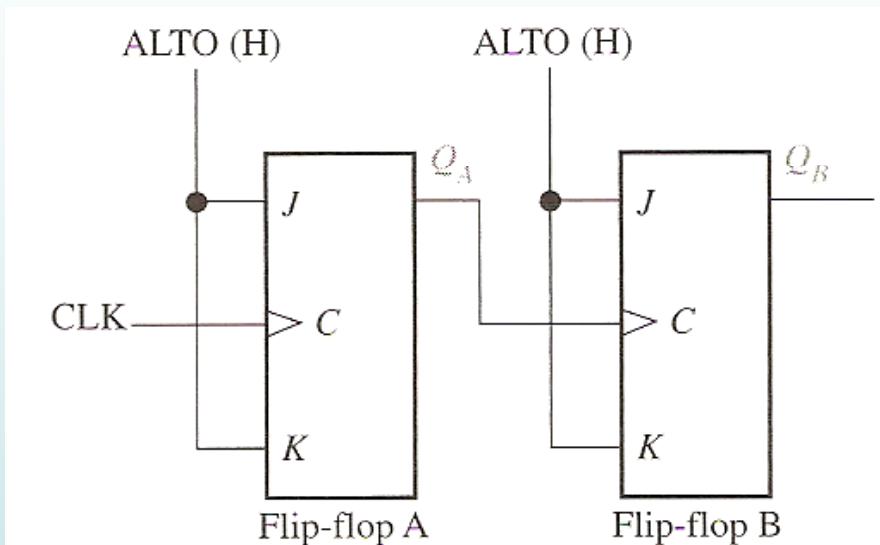
6.2 ELEMENTOS BÁSICOS DE MEMORIA: Aplicaciones

- Divisor de frecuencia: la salida es una señal con una frecuencia igual a la mitad de la que tiene el reloj.

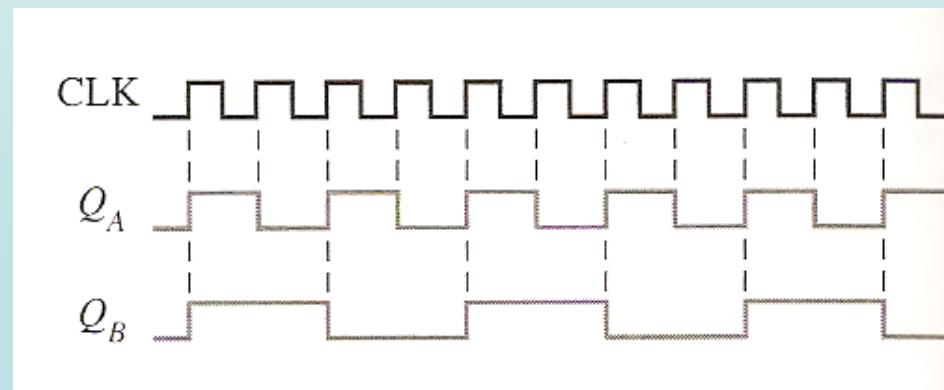


6.2 ELEMENTOS BÁSICOS DE MEMORIA

- ¿Qué hace el siguiente circuito?

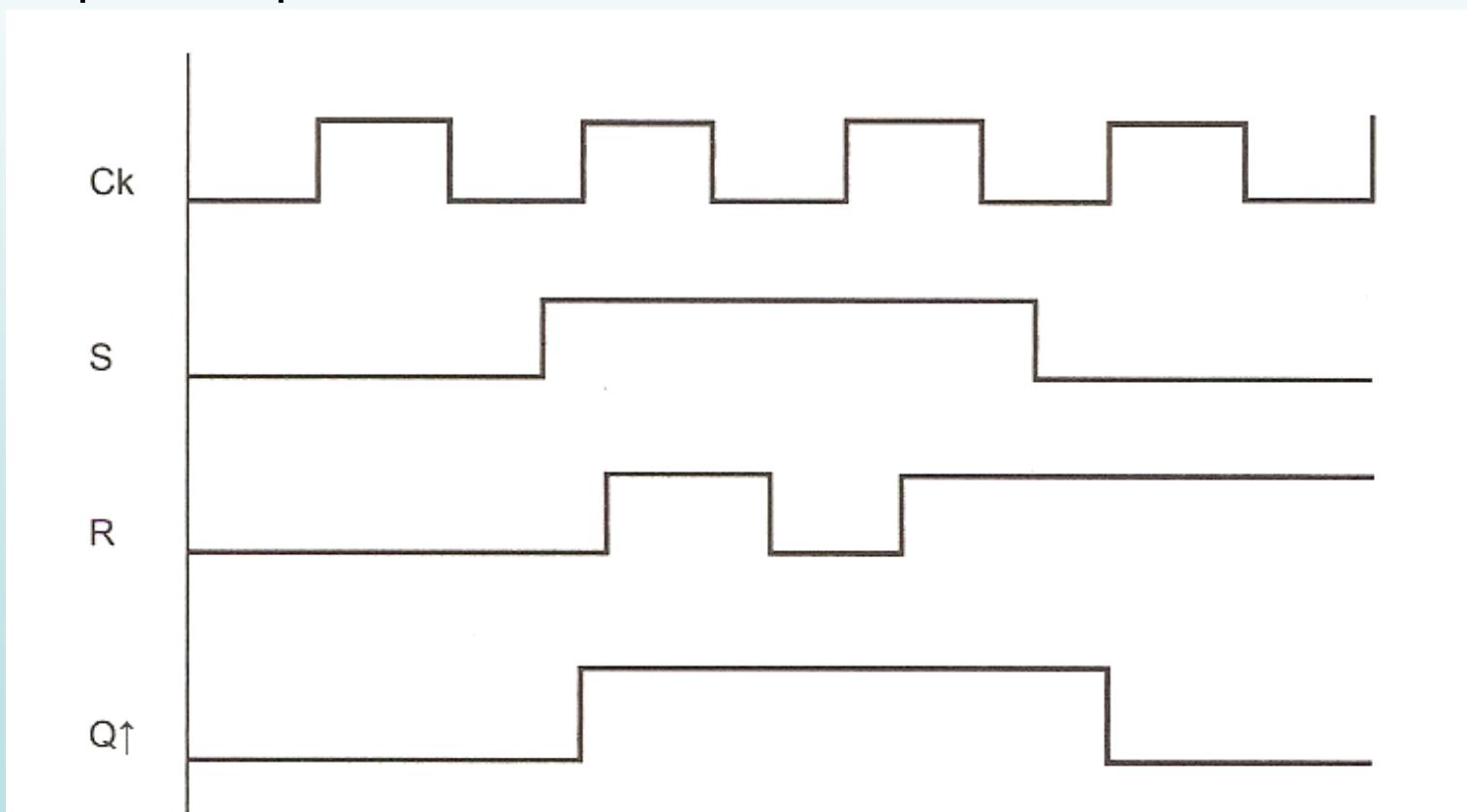


Divide la frecuencia
de reloj por 4



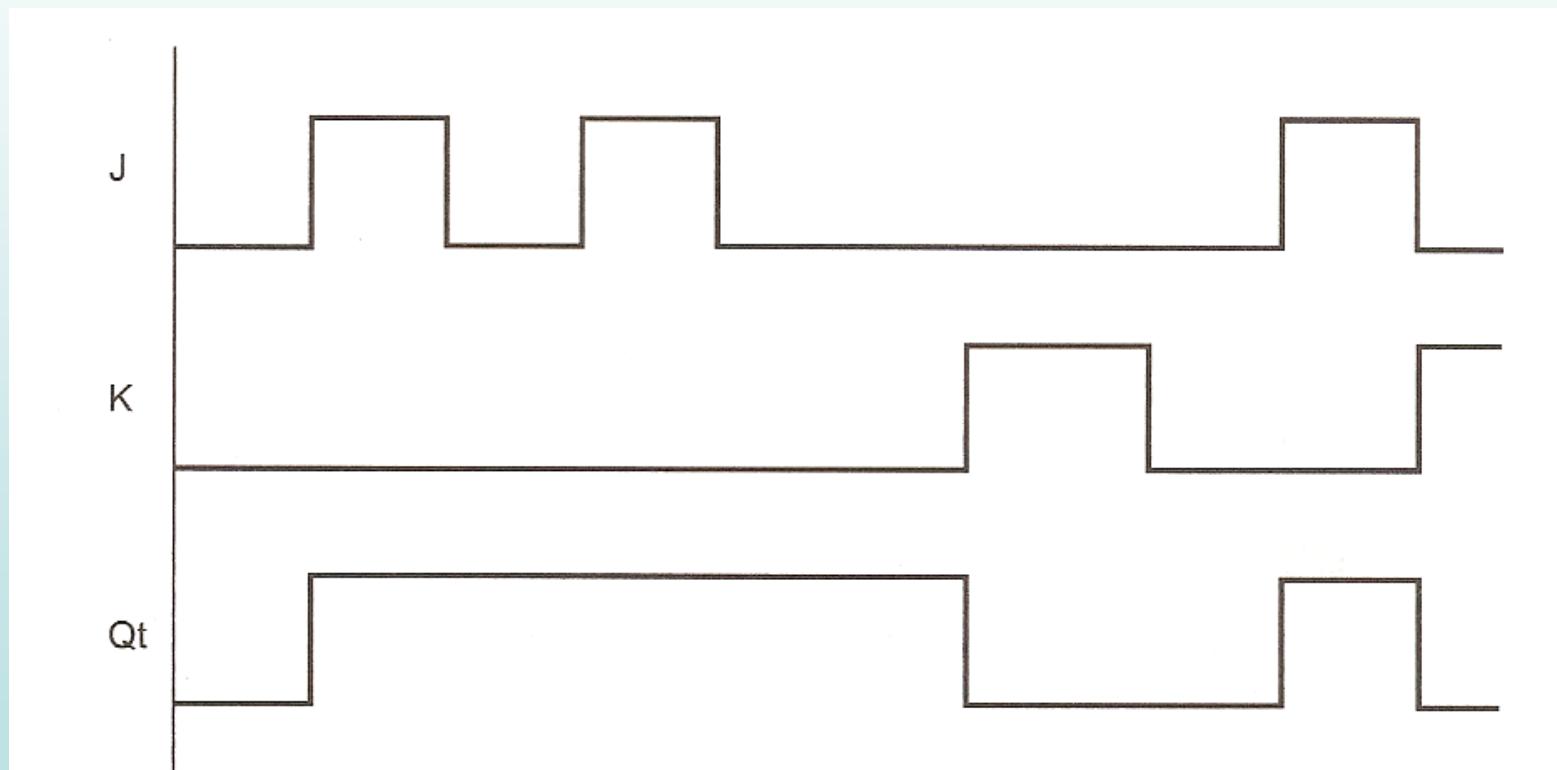
6.2 ELEMENTOS BÁSICOS DE MEMORIA: Ejercicios

- Completar el cronograma para un biestable SR síncrono disparado por flanco ascendente.



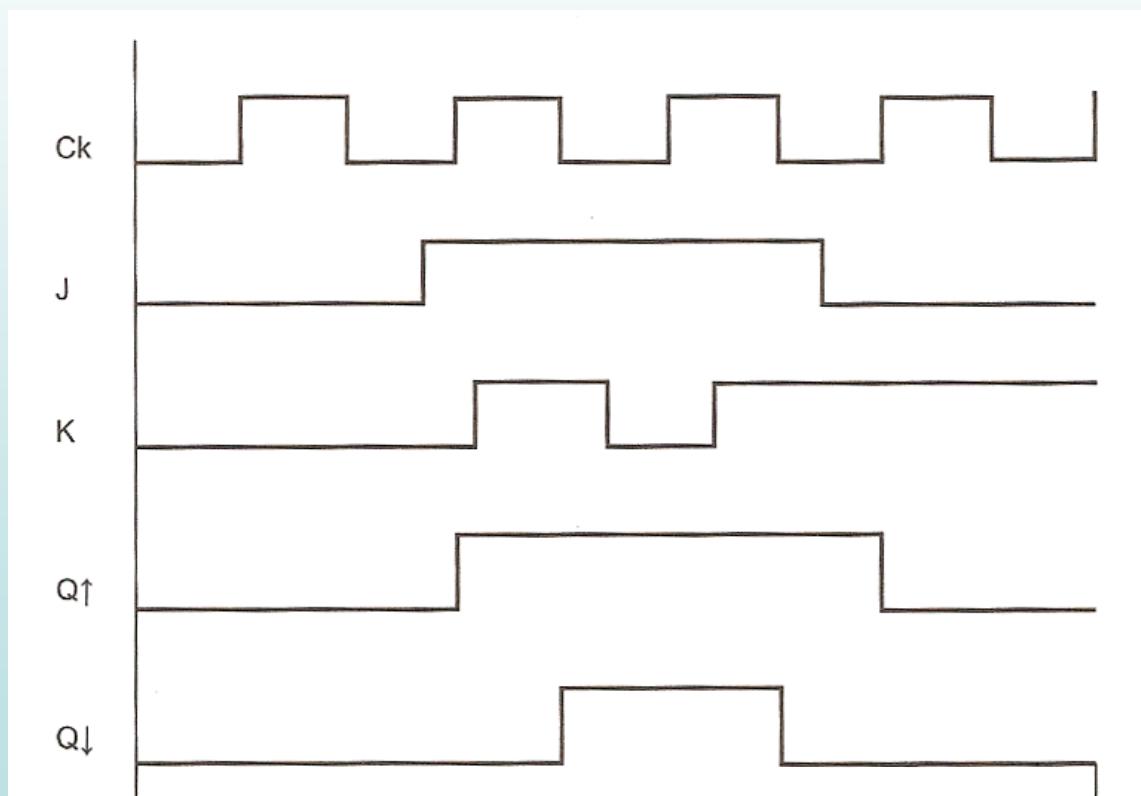
6.2 ELEMENTOS BÁSICOS DE MEMORIA: Ejercicios

- Completar el cronograma para un biestable J-K asíncrono.



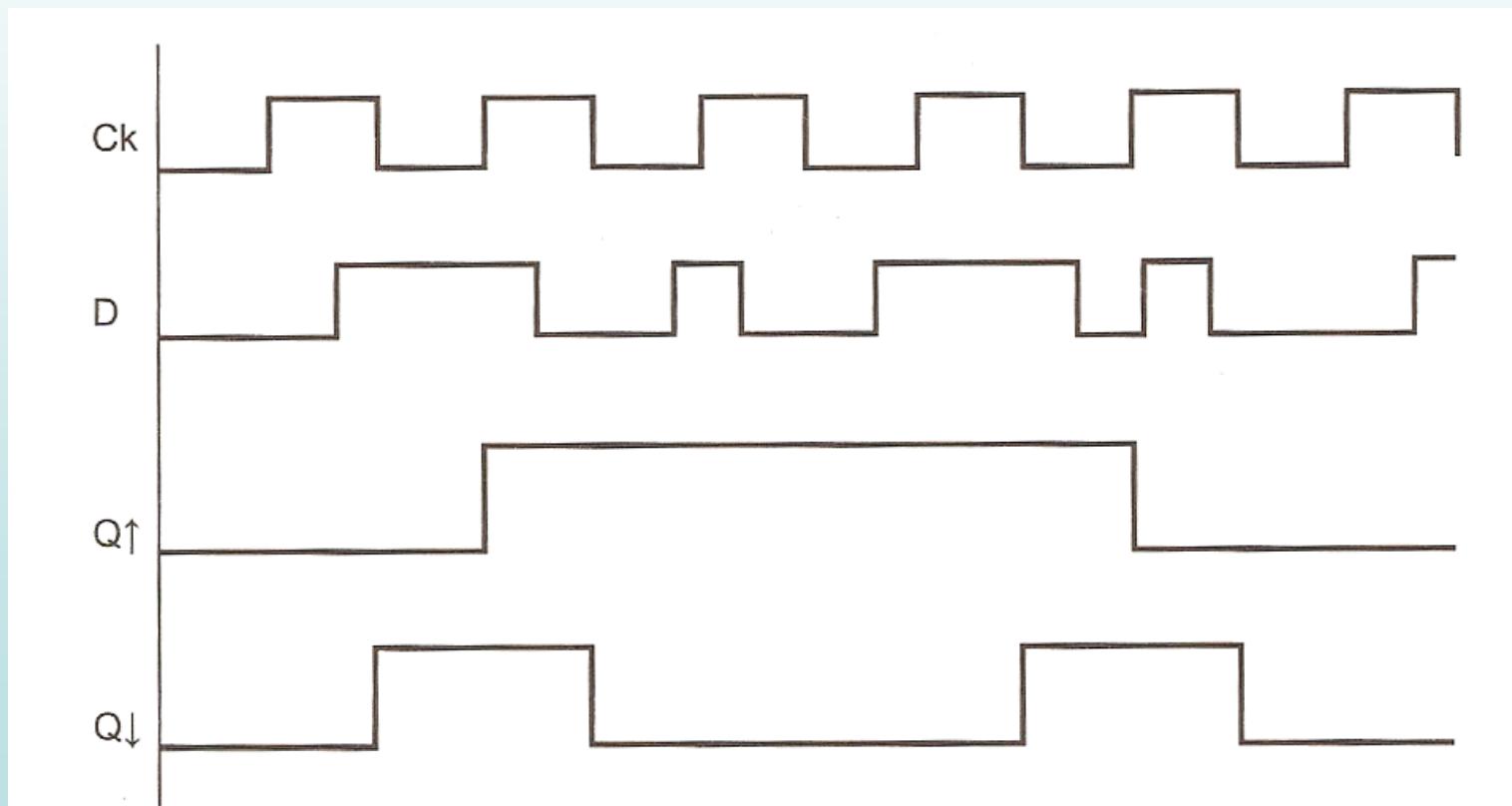
6.2 ELEMENTOS BÁSICOS DE MEMORIA: Ejercicios

- Completar el cronograma para un biestable J-K síncrono disparado por flanco ascendente y descendente



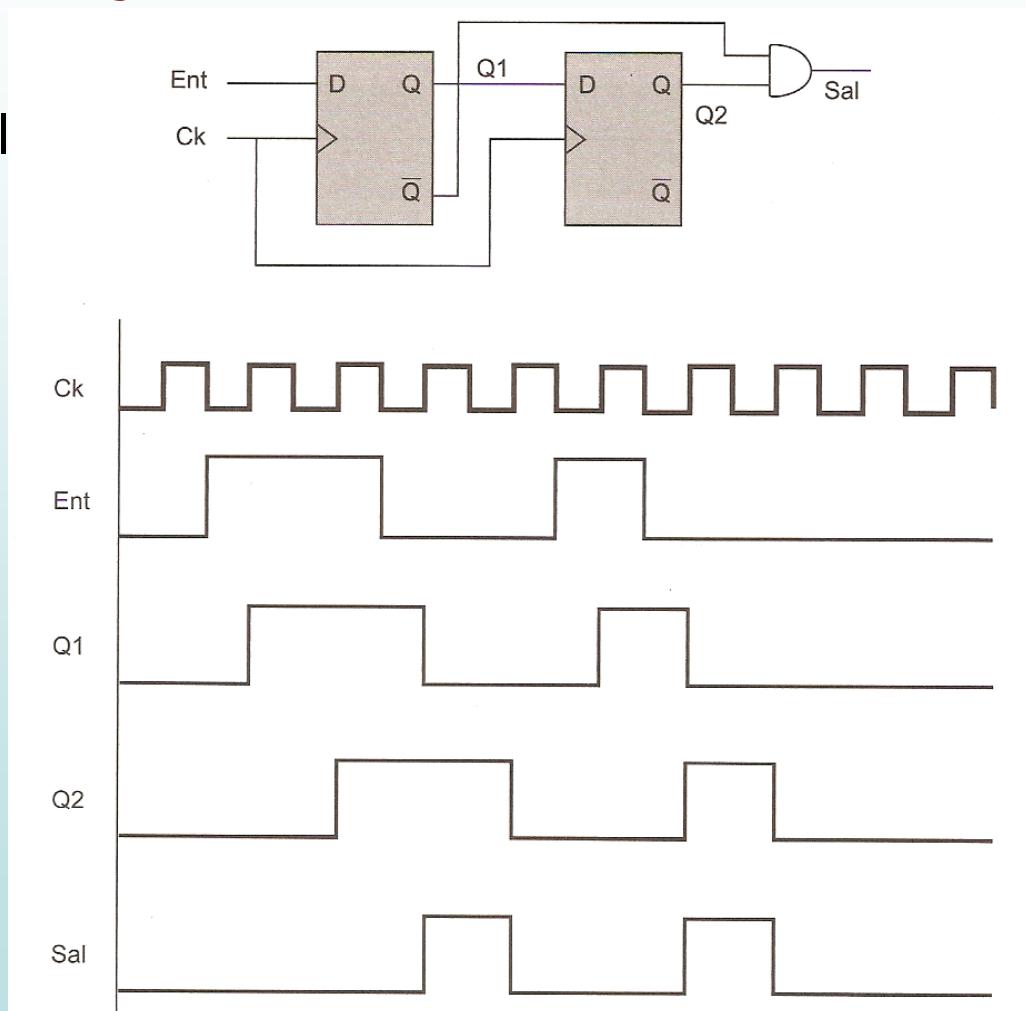
6.2 ELEMENTOS BÁSICOS DE MEMORIA: Ejercicios

- Completar el cronograma para un biestable D síncrono disparado por flanco ascendente y descendente



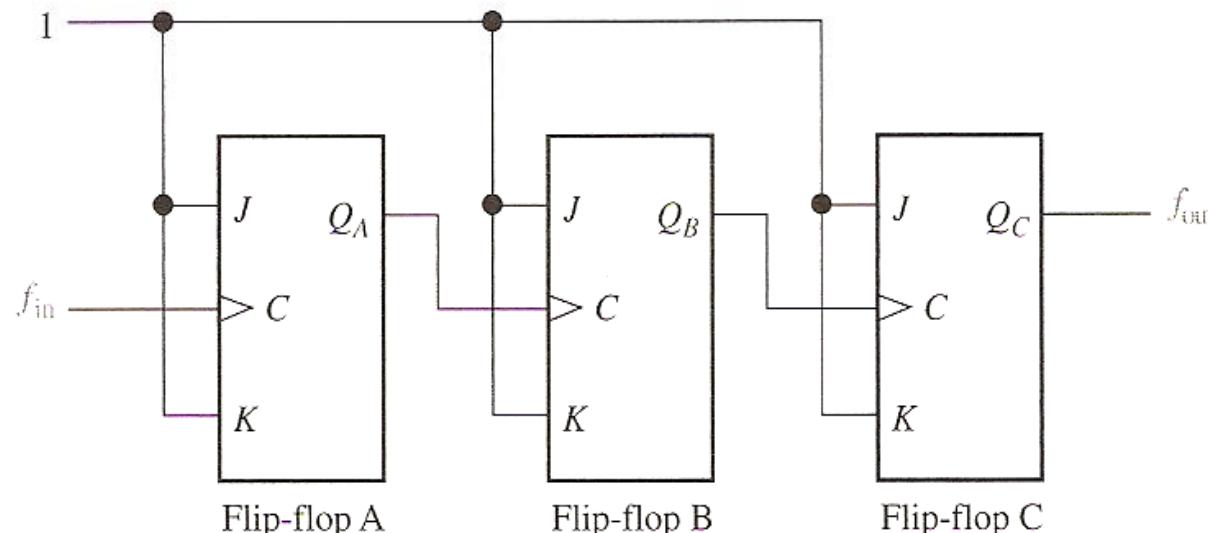
6.2 ELEMENTOS BÁSICOS DE MEMORIA: Ejercicios

- Completar el cronograma para el circuito de la figura. ¿Cuál es el cometido de dicho circuito?

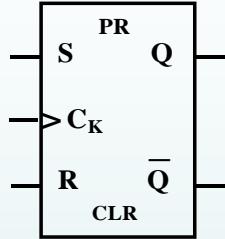
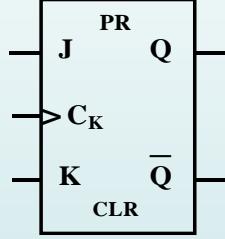
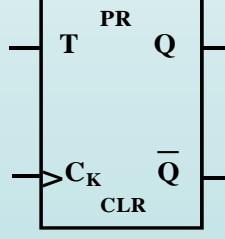
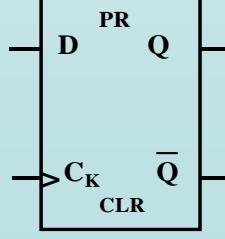


6.2 ELEMENTOS BÁSICOS DE MEMORIA: Ejercicios

Desarrollar la forma de onda f_{out} para el circuito de la Figura 8.42, cuando se aplica una señal cuadrada de 8 kHz en la entrada de reloj del flip-flop A.



6.2 ELEMENTOS BÁSICOS DE MEMORIA: Resumen

Tabla Abreviada	T. Inversa	Símbolo																				
<table border="1"> <thead> <tr> <th>S R</th><th>Q⁺(Est. Siguiente)</th></tr> </thead> <tbody> <tr> <td>0 0</td><td>Q No cambia</td></tr> <tr> <td>0 1</td><td>0 Pone a Cero</td></tr> <tr> <td>1 0</td><td>1 Pone a uno</td></tr> <tr> <td>1 1</td><td>* No utilizar</td></tr> </tbody> </table>	S R	Q ⁺ (Est. Siguiente)	0 0	Q No cambia	0 1	0 Pone a Cero	1 0	1 Pone a uno	1 1	* No utilizar	<table border="1"> <thead> <tr> <th>Q Q⁺</th><th>S R</th></tr> </thead> <tbody> <tr> <td>0 0</td><td>0 -</td></tr> <tr> <td>0 1</td><td>1 0</td></tr> <tr> <td>1 0</td><td>0 1</td></tr> <tr> <td>1 1</td><td>- 0</td></tr> </tbody> </table>	Q Q ⁺	S R	0 0	0 -	0 1	1 0	1 0	0 1	1 1	- 0	
S R	Q ⁺ (Est. Siguiente)																					
0 0	Q No cambia																					
0 1	0 Pone a Cero																					
1 0	1 Pone a uno																					
1 1	* No utilizar																					
Q Q ⁺	S R																					
0 0	0 -																					
0 1	1 0																					
1 0	0 1																					
1 1	- 0																					
<table border="1"> <thead> <tr> <th>J K</th><th>Q⁺(Est. Siguiente)</th></tr> </thead> <tbody> <tr> <td>0 0</td><td>Q No cambia</td></tr> <tr> <td>0 1</td><td>0 Pone a Cero</td></tr> <tr> <td>1 0</td><td>1 Pone a uno</td></tr> <tr> <td>1 1</td><td>Q Cambia de estado</td></tr> </tbody> </table>	J K	Q ⁺ (Est. Siguiente)	0 0	Q No cambia	0 1	0 Pone a Cero	1 0	1 Pone a uno	1 1	Q Cambia de estado	<table border="1"> <thead> <tr> <th>Q Q⁺</th><th>J K</th></tr> </thead> <tbody> <tr> <td>0 0</td><td>0 -</td></tr> <tr> <td>0 1</td><td>1 -</td></tr> <tr> <td>1 0</td><td>- 1</td></tr> <tr> <td>1 1</td><td>- 0</td></tr> </tbody> </table>	Q Q ⁺	J K	0 0	0 -	0 1	1 -	1 0	- 1	1 1	- 0	
J K	Q ⁺ (Est. Siguiente)																					
0 0	Q No cambia																					
0 1	0 Pone a Cero																					
1 0	1 Pone a uno																					
1 1	Q Cambia de estado																					
Q Q ⁺	J K																					
0 0	0 -																					
0 1	1 -																					
1 0	- 1																					
1 1	- 0																					
<table border="1"> <thead> <tr> <th>T</th><th>Q⁺(Est. Siguiente)</th></tr> </thead> <tbody> <tr> <td>0</td><td>Q No cambia</td></tr> <tr> <td>1</td><td>Q Cambia Estado</td></tr> </tbody> </table>	T	Q ⁺ (Est. Siguiente)	0	Q No cambia	1	Q Cambia Estado	<table border="1"> <thead> <tr> <th>Q Q⁺</th><th>T</th></tr> </thead> <tbody> <tr> <td>0 0</td><td>0</td></tr> <tr> <td>0 1</td><td>1</td></tr> <tr> <td>1 0</td><td>1</td></tr> <tr> <td>1 1</td><td>0</td></tr> </tbody> </table>	Q Q ⁺	T	0 0	0	0 1	1	1 0	1	1 1	0					
T	Q ⁺ (Est. Siguiente)																					
0	Q No cambia																					
1	Q Cambia Estado																					
Q Q ⁺	T																					
0 0	0																					
0 1	1																					
1 0	1																					
1 1	0																					
<table border="1"> <thead> <tr> <th>D</th><th>Q⁺</th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td></tr> <tr> <td>1</td><td>1</td></tr> </tbody> </table>	D	Q ⁺	0	0	1	1	<table border="1"> <thead> <tr> <th>Q⁺</th><th>D</th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td></tr> <tr> <td>1</td><td>1</td></tr> </tbody> </table>	Q ⁺	D	0	0	1	1									
D	Q ⁺																					
0	0																					
1	1																					
Q ⁺	D																					
0	0																					
1	1																					

TEMA 6. ANÁLISIS Y DISEÑO DE SISTEMAS SECuenciales.

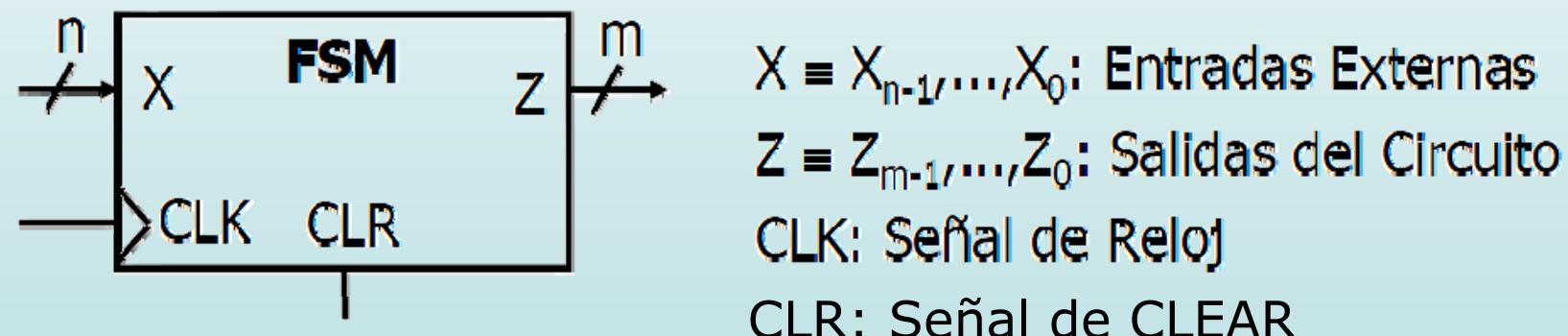
CONTENIDOS:

- 6.1. Concepto de sistema secuencial.
- 6.2. Elementos básicos de memoria.
- 6.3. Análisis de un sistema secuencial.**
- 6.4. Diseño de un sistema secuencial.
- 6.5. Componentes secuenciales estándar.



6.3 ANÁLISIS DE UN SISTEMA SECUENCIAL. MÁQUINAS DE ESTADOS FINITOS.

- Una **máquina de estados finitos** ('Finite State Machine', FSM) es un sistema secuencial en el que todos los elementos secuenciales son biestables disparados por la misma señal de reloj (CLK). Es un sistema **secuencial síncrono**.

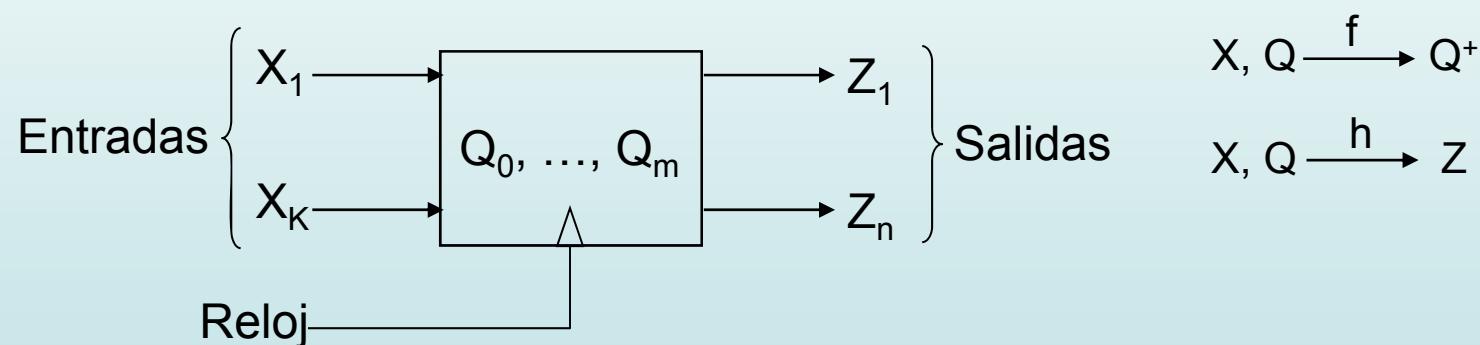


6.3 ANÁLISIS DE UN SISTEMA SECUENCIAL. MÁQUINAS DE ESTADOS FINITOS.

- Una **máquina de estados finitos (FSM)**, que está formada por:
 - Entradas (X)
 - Salidas (Z)
 - Estados (Q)
 - Función de estado siguiente, f , que asigna a cada pareja (estado, entrada) un estado.
 - Función de salida, h , que es la función de salida, que asigna a cada (estado, entrada) una salida.

6.3 ANÁLISIS DE UN SISTEMA SECUENCIAL. MÁQUINAS DE ESTADOS FINITOS.

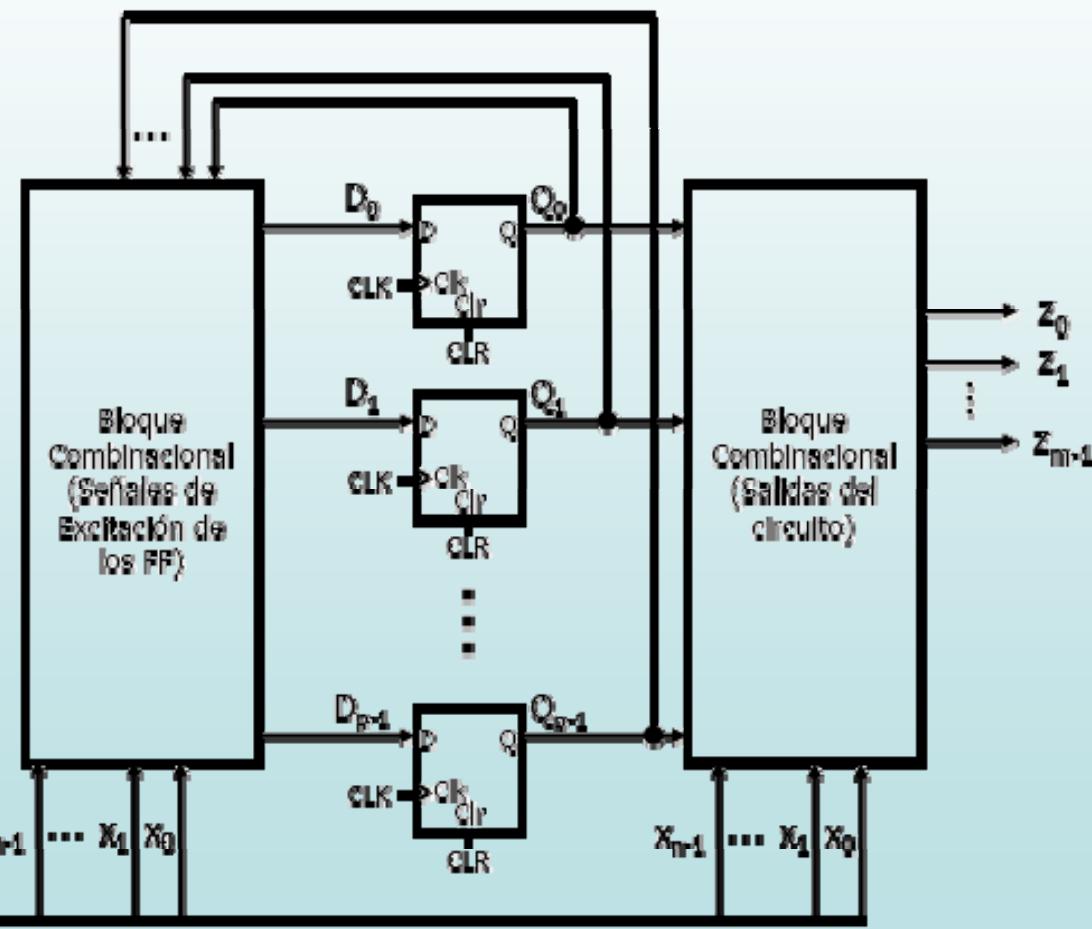
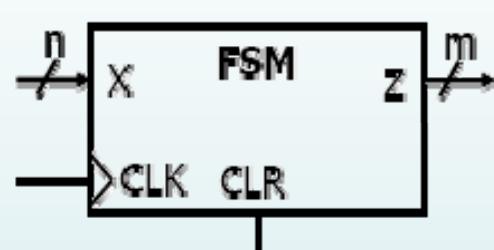
- Si el **sistema secuencial es síncrono**, la función f , va sincronizada por una señal de reloj.



Modelo FSM de un sistema secuencial síncrono

6.3 ANÁLISIS DE UN SISTEMA SECUENCIAL. MÁQUINAS DE ESTADOS FINITOS.

Estructura general de una FSM:



Estado ■ Cada posible combinación de salida de los flip-flops $\rightarrow (Q_{p-1}, \dots, Q_0)$

Si hay p flip-flops \rightarrow tendremos como máximo 2^p estados diferentes

6.3 ANÁLISIS DE UN SISTEMA SECUENCIAL. MÁQUINAS DE ESTADOS FINITOS.

- **FSM tipo Mealy:**

La función de salida, h , queda definida por el estado y las entradas. Es decir, las salidas dependen del estado actual y de las entradas externas.

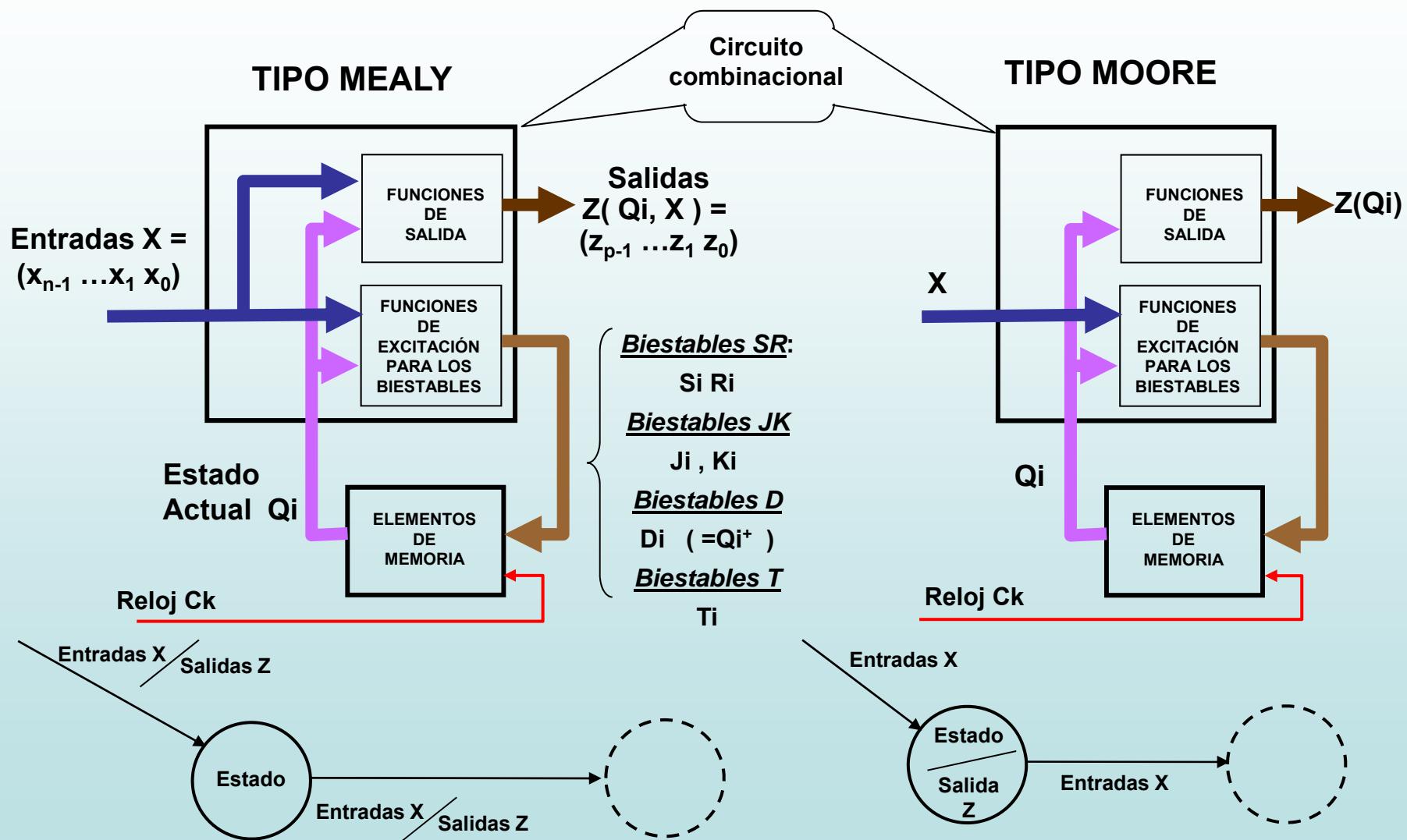
$$z_i = F_i(x_{n-1}, \dots, x_0, Q_{p-1}, \dots, Q_0), i = 0, \dots, m-1$$

- **FSM tipo Moore:**

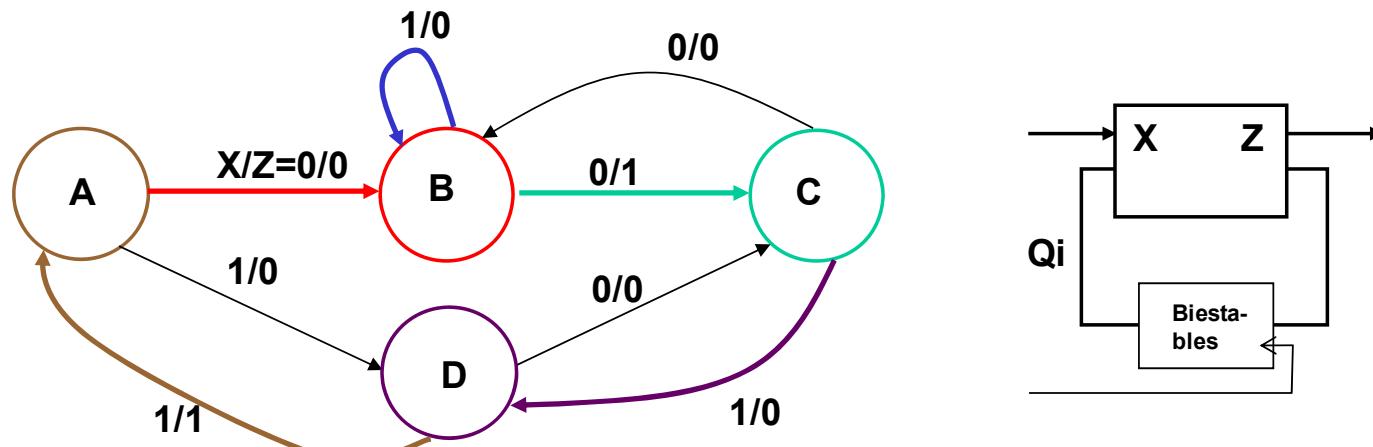
La función de salida asigna una salida a cada estado. Es decir, las salidas no dependen de las entradas externas, sólo del estado actual de la FSM.

$$z_i = F_i(Q_{p-1}, \dots, Q_1, Q_0), i = 0, \dots, m-1$$

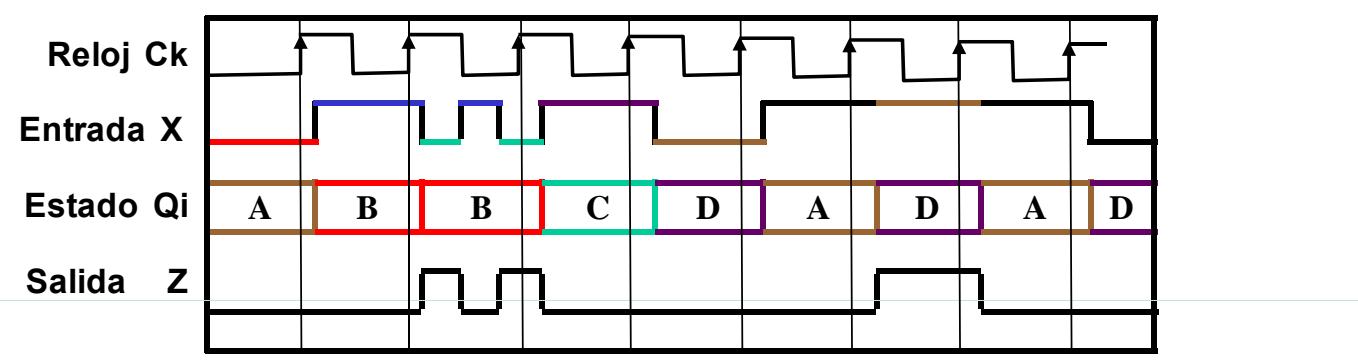
6.3 ANÁLISIS DE UN SISTEMA SECUENCIAL. MÁQUINAS DE ESTADOS FINITOS.



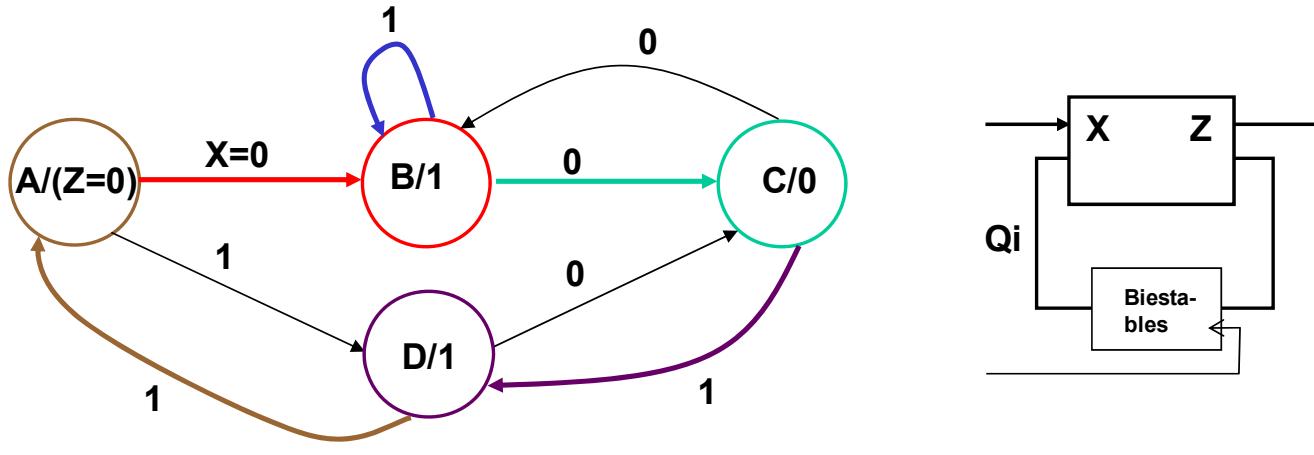
6.3 ANÁLISIS DE UN SISTEMA SECUENCIAL.



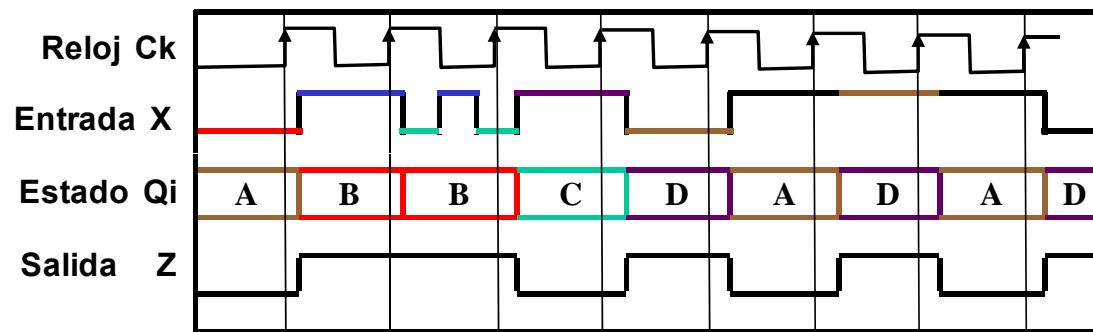
MÁQUINA TIPO MEALY



6.3 ANÁLISIS DE UN SISTEMA SECUENCIAL.



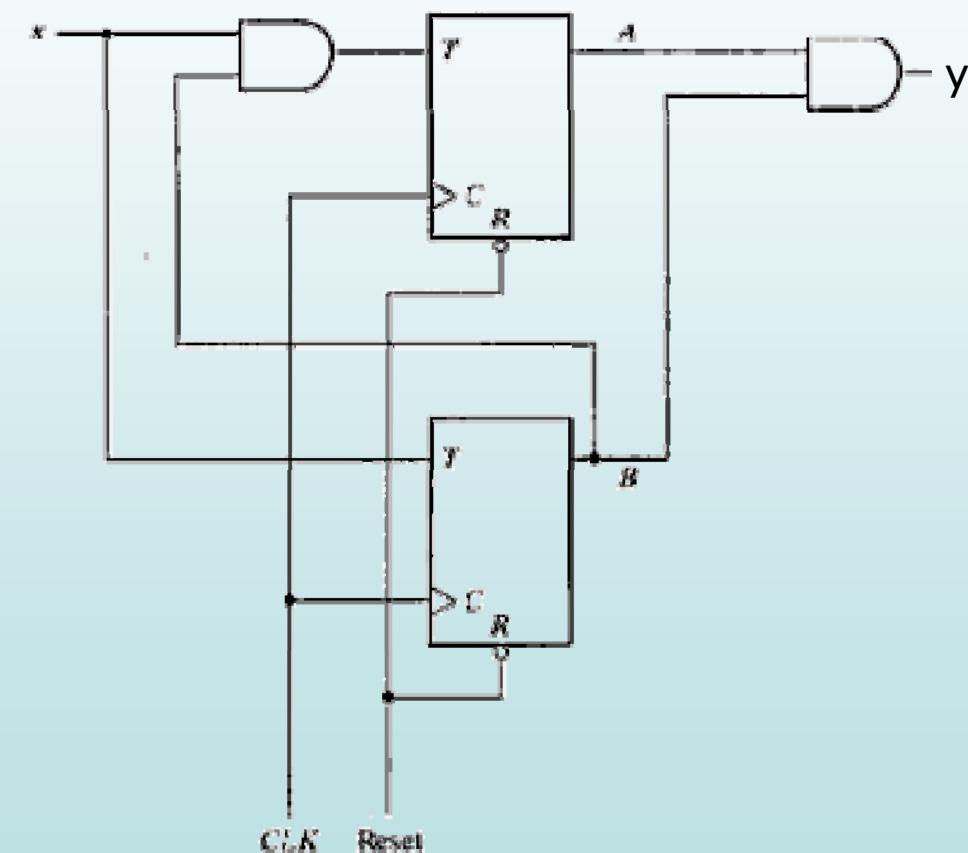
MÁQUINA TIPO MOORE



6.3 ANÁLISIS DE UN SISTEMA SECUENCIAL. MÁQUINAS DE ESTADOS FINITOS.

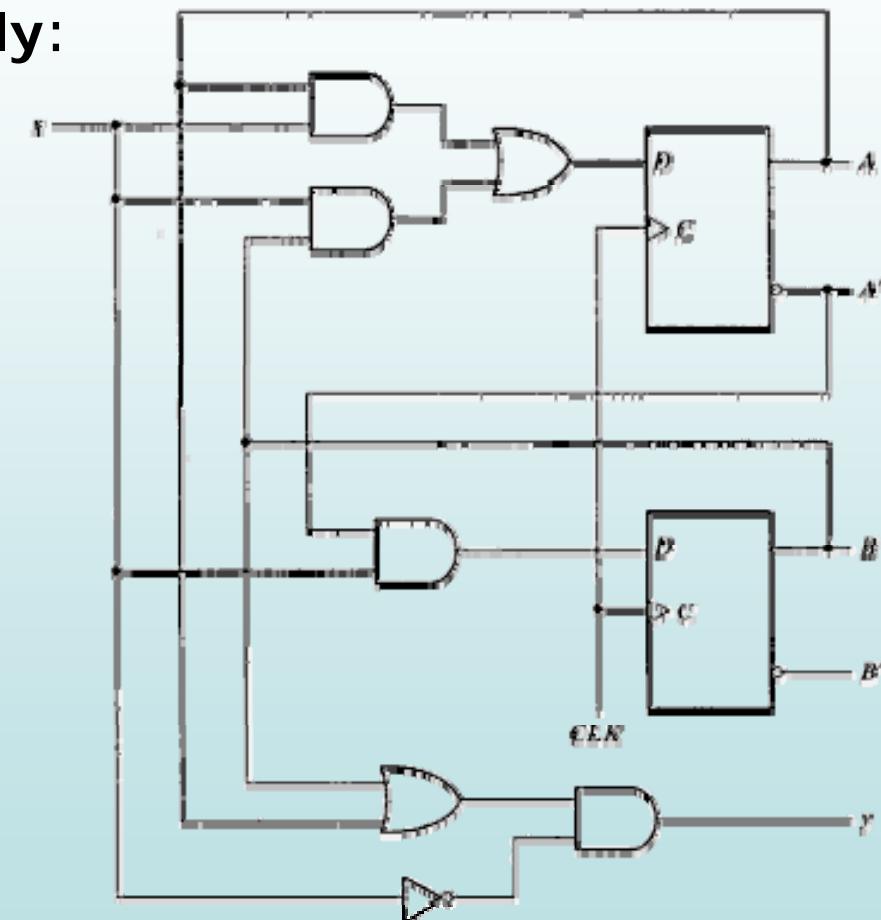
- **Ejemplo de FSM tipo Moore:**

La salida, y , depende
de los estados A y B.



6.3 ANÁLISIS DE UN SISTEMA SECUENCIAL. MÁQUINAS DE ESTADOS FINITOS.

- **Ejemplo de FSM tipo Mealy:**
la salida, y , depende de la
entrada, x , y de los estados
 A y B .



6.3 ANÁLISIS DE UN SISTEMA SECUENCIAL.

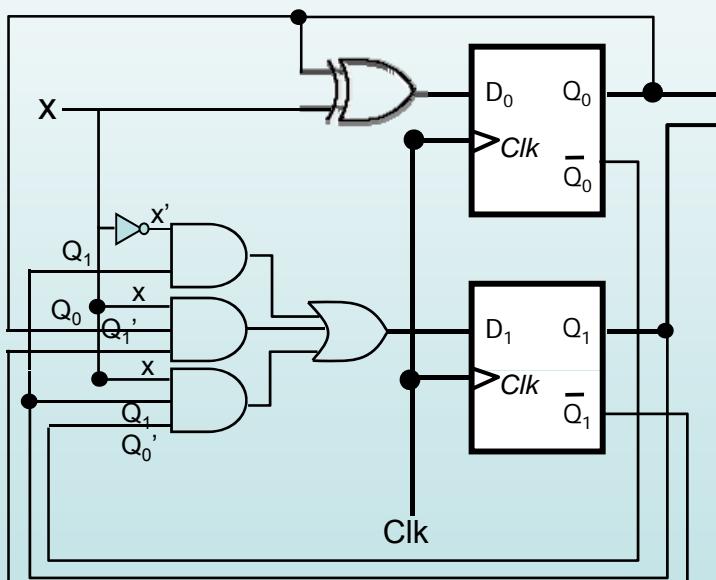
Análisis: se parte de un esquema lógico y hay que obtener una descripción del comportamiento del sistema.

Pasos a seguir:

1. Obtener las funciones de excitación de los biestables y de las salidas del sistema
2. Obtener la tabla de estados
3. Generar diagrama de estados
4. Cronograma
5. Descripción del comportamiento del sistema

6.3 ANÁLISIS DE UN SISTEMA SECUENCIAL.

- ANÁLISIS DE UN SISTEMA SECUENCIAL:



Función de excitación de los biestables y salidas:

$$Q_0^+ = D_0 = Q_0 \oplus X$$

$$Q_1^+ = D_1 = X' \cdot Q_1 + X \cdot Q_1' \cdot Q_0 + X \cdot Q_1 \cdot Q_0'$$

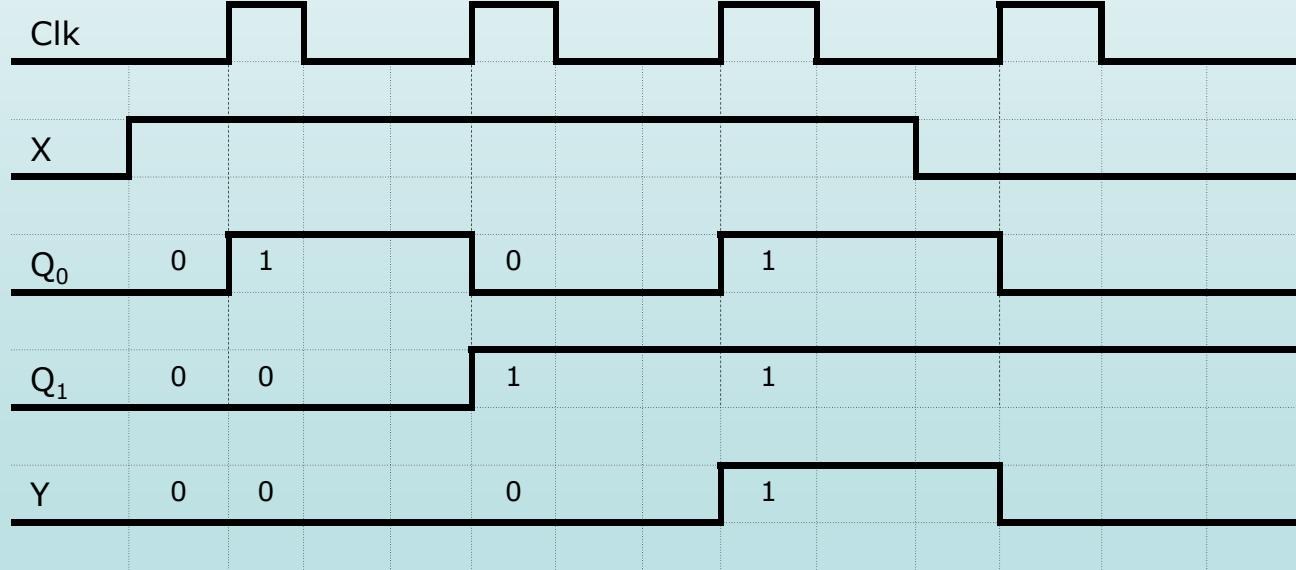
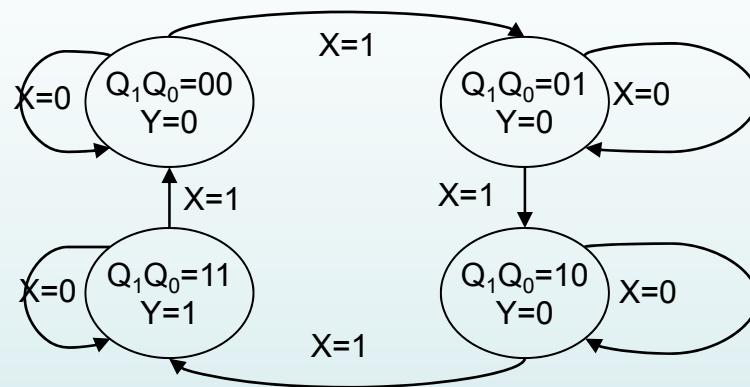
$$Y = Q_1 \cdot Q_0$$

Tabla de estados

Q_1	Q_0	Q_1^+	Q_0^+	Q_1^+	Q_0^+	Y
		$X=0$		$X=1$		
0	0	0	0	0	1	0
0	1	0	1	1	0	0
1	0	1	0	1	1	0
1	1	1	1	0	0	1

6.3 ANÁLISIS DE UN SISTEMA SECUENCIAL.

Diagrama de estados y cronograma:

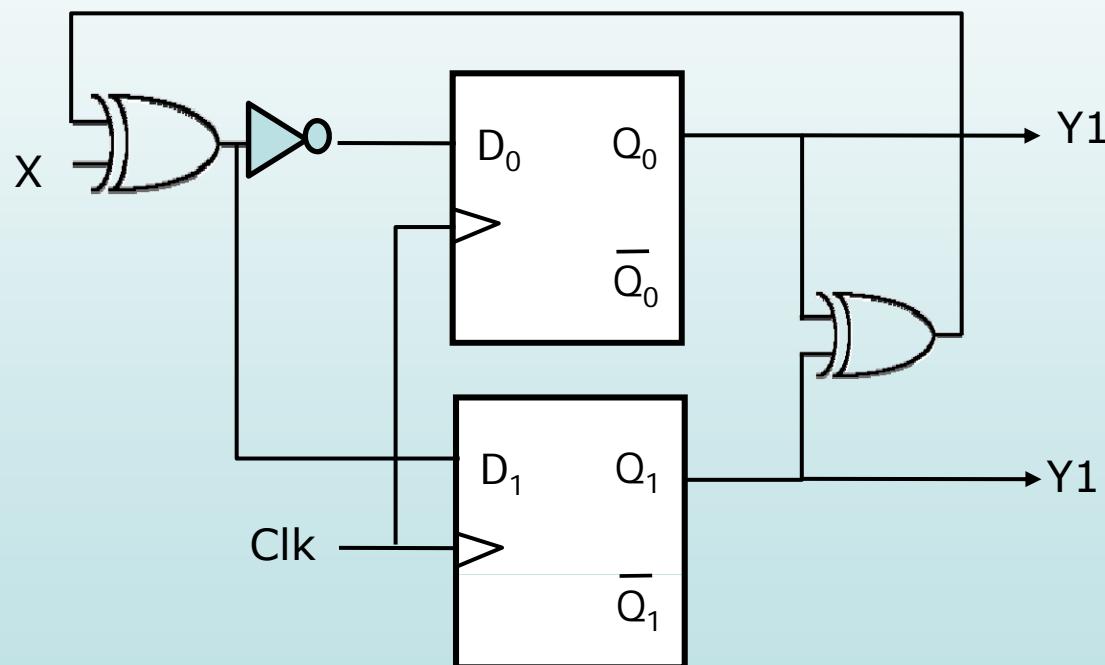


Ejemplo: Es un contador módulo 4:
0, 1, 2, 3
(Q₁Q₀ = 00, 01, 10, 11)

Cuando llega a 3 la salida, Y, vale 1

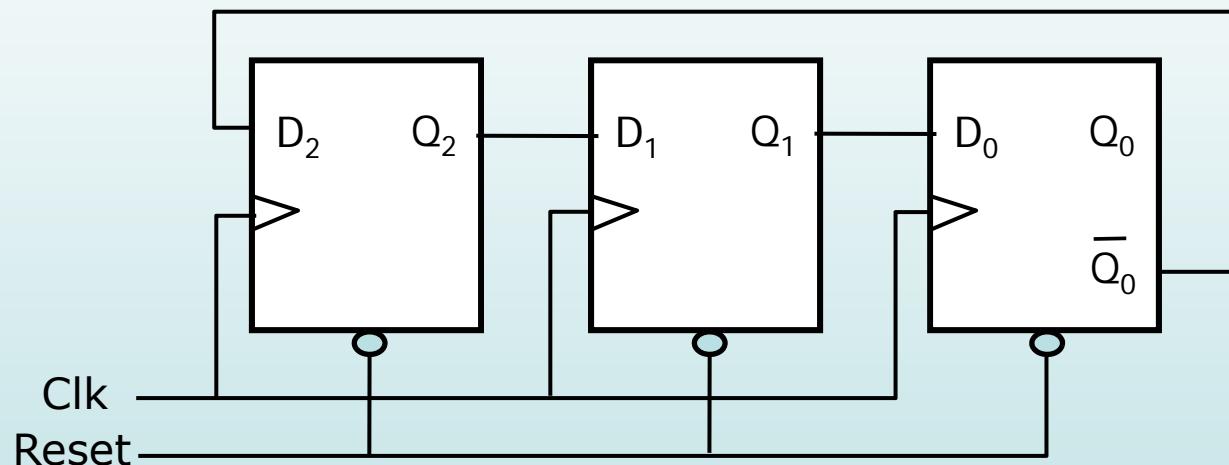
6.3 ANÁLISIS DE UN SISTEMA SECUENCIAL.

- Ejercicio 1: Analizar el siguiente circuito secuencial

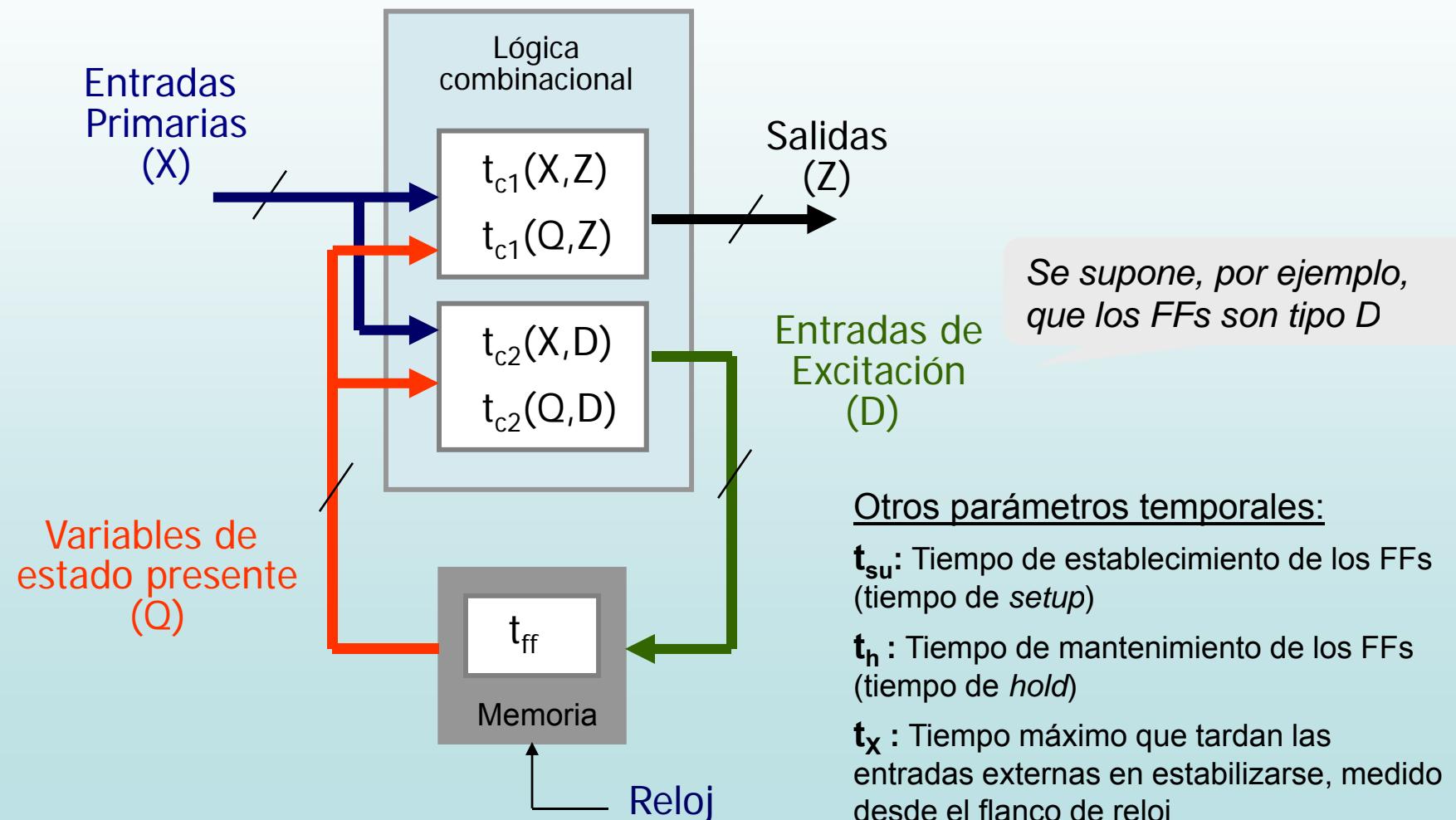


6.3 ANÁLISIS DE UN SISTEMA SECUENCIAL.

- Ejercicio 2: analizar el siguiente sistema secuencial

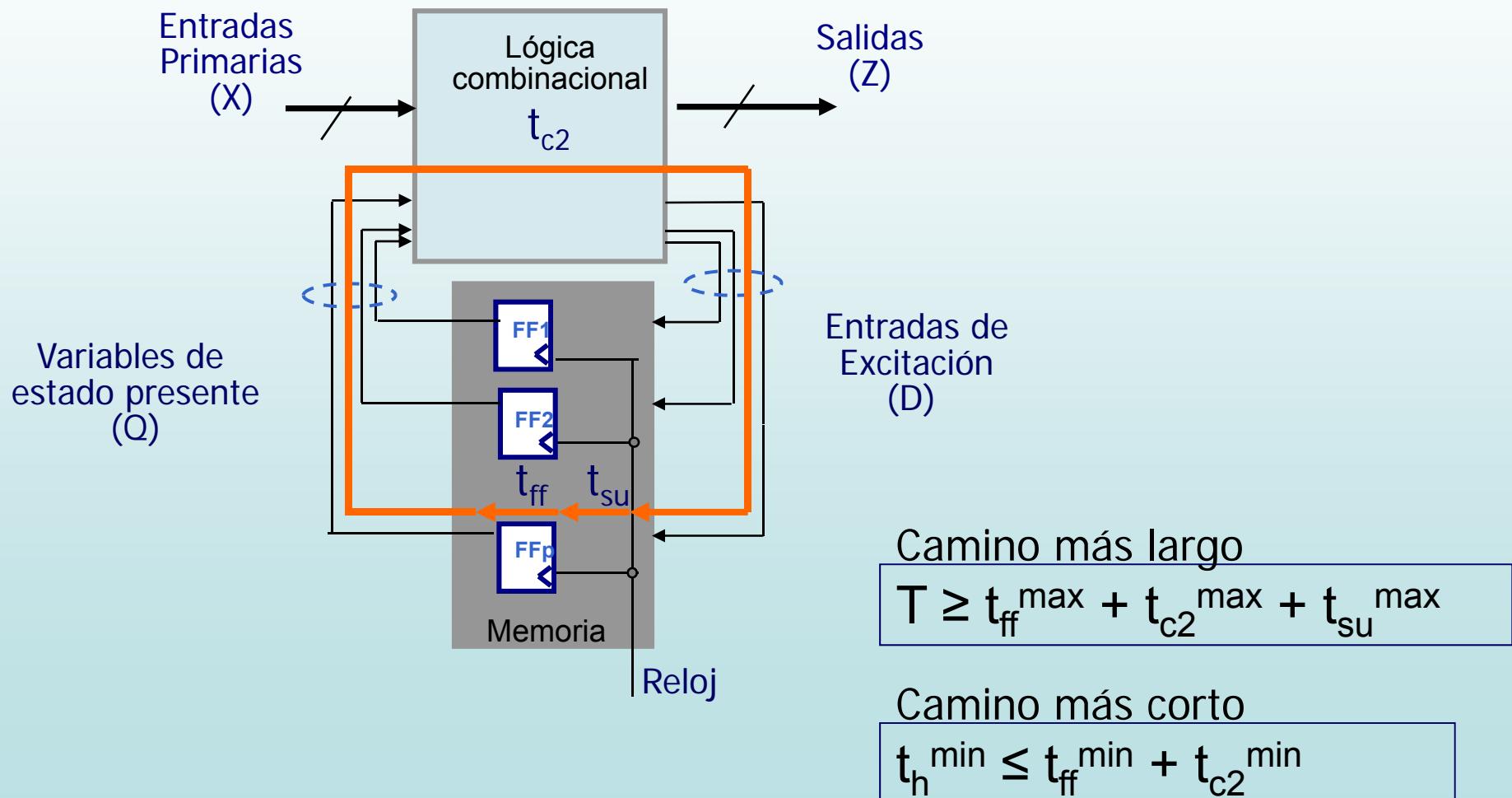


6.3 ANÁLISIS DE UN SISTEMA SECUENCIAL. Retardos de propagación en una FSM (tipo Mealy)



6.3 ANÁLISIS DE UN SISTEMA SECUENCIAL.

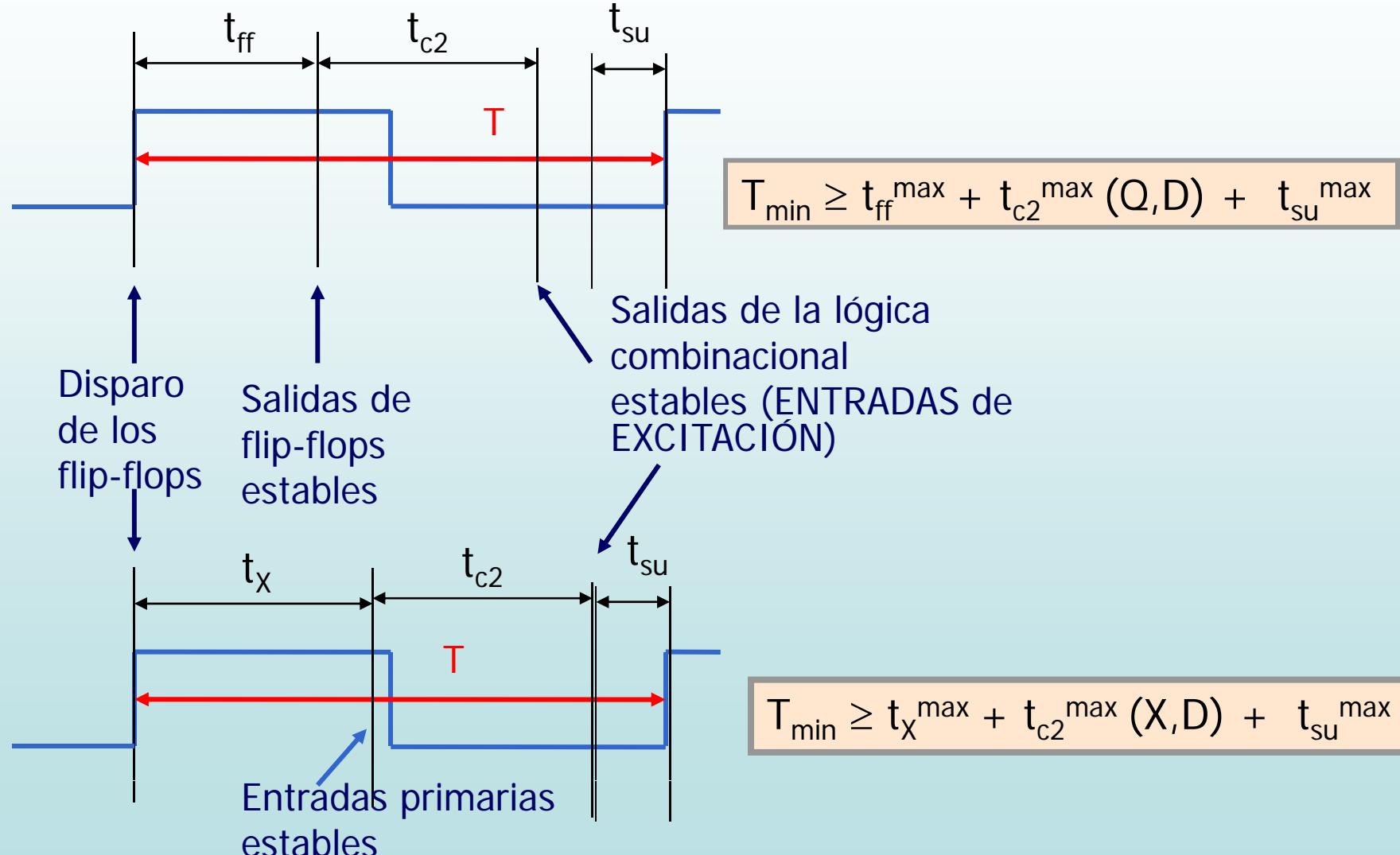
Restricciones en el camino de realimentación



6.3 ANÁLISIS DE UN SISTEMA SECUENCIAL.

Eventos durante el ciclo de reloj

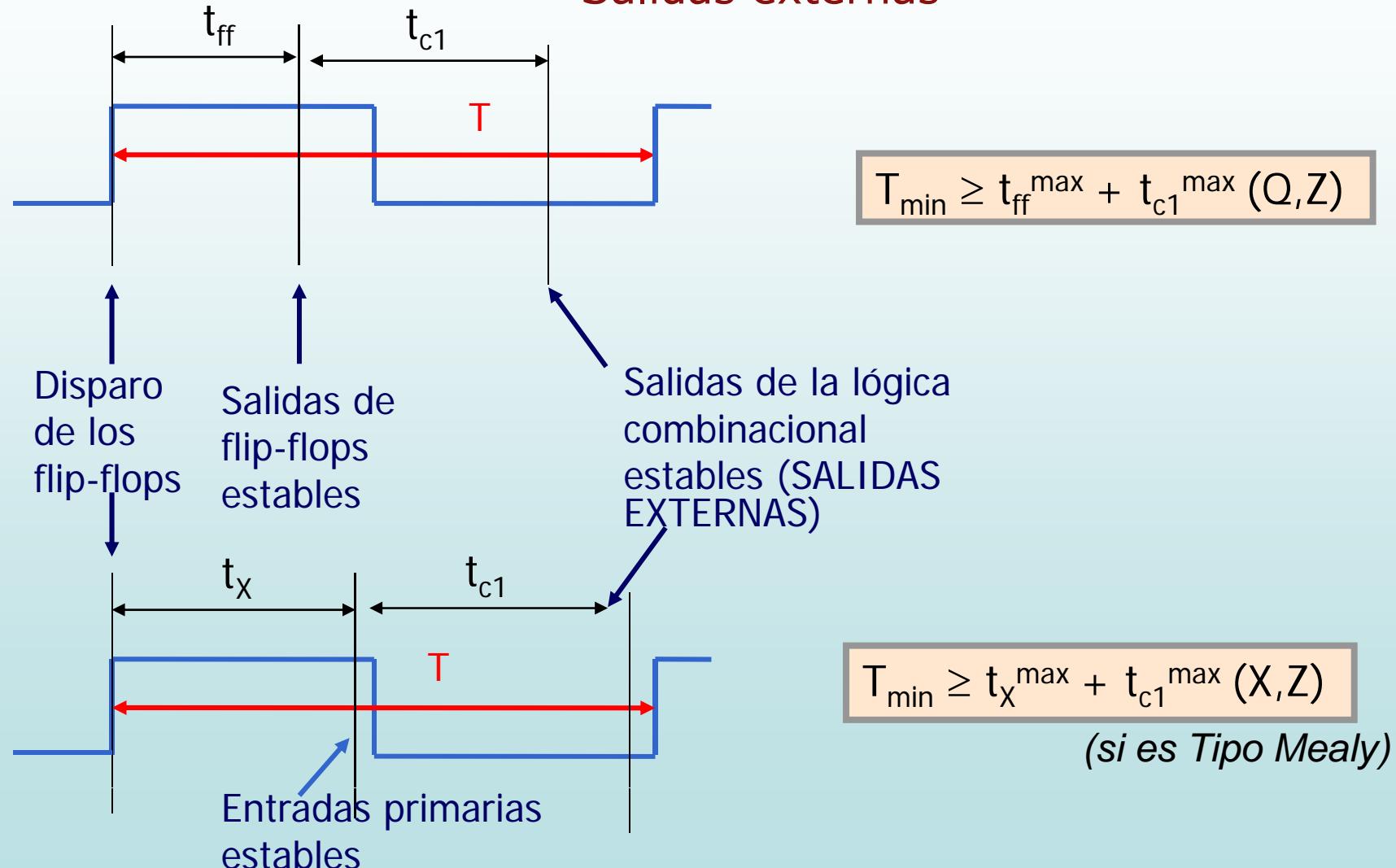
Camino de realimentación



6.3 ANÁLISIS DE UN SISTEMA SECUENCIAL.

Eventos durante el ciclo de reloj

Salidas externas



6.3 ANÁLISIS DE UN SISTEMA SECUENCIAL. Diseño de la señal de reloj

$$T_{\min} = \max\{t_{ff}^{\max} + t_{c2}^{\max}(Q,D) + t_{su}^{\max}, \\ t_{ff}^{\max} + t_{c1}^{\max}(Q,Z), \\ t_X^{\max} + t_{c2}^{\max}(X,D) + t_{su}^{\max}, \\ t_X^{\max} + t_{c1}^{\max}(X,Z)\} + E$$

$$f_{\max} = 1 / T_{\min}$$

donde

$$t_{su}^{\max} = \max \{t_{su}^{\min}(FF1), t_{su}^{\min}(FF2), \dots, t_{su}^{\min}(FFp)\}$$

$$t_{ff}^{\max} = \max \{t_{ff}^{\max}(FF1), t_{ff}^{\max}(FF2), \dots, t_{ff}^{\max}(FFp)\}$$

E es un término de tolerancia o error con el que se asume la incertidumbre en los retardos de propagación y la posibilidad de skew ($E \sim 20\%$)

TEMA 6. ANÁLISIS Y DISEÑO DE SISTEMAS SECuenciales.

CONTENIDOS:

- 6.1. Concepto de sistema secuencial.
- 6.2. Elementos básicos de memoria.
- 6.3. Análisis de un sistema secuencial.
- 6.4. Diseño de un sistema secuencial.**
- 6.5. Componentes secuenciales estándar.

6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

Fases de diseño de los circuitos secuenciales síncronos, utilizando puertas lógicas y biestables.

ETAPA 1 Descripción funcional del circuito	DIAGRAMAS Y TABLAS DE ESTADOS.
ETAPA 2 Minimización del número de estados	TABLA DE ESTADOS MINIMIZADA.
ETAPA 3 Asignación de estados, elección del tipo de biestables a utilizar. Obtención de las funciones de excitación de los biestables y funciones de salida.	TABLA DE TRANSICIÓN ASIGNADA. EXPRESIONES CANONICAS DE LAS: Funciones de excitación de los biestables y de las Funciones de Salida.
ETAPA 4 Minimización de las funciones	EXPRESIONES MINIMIZADAS DE LAS FUNCIONES Y ESQUEMA DEL CIRCUITO
ETAPA 5 Realización física del circuito	REALIZACIÓN FÍSICA DEL CIRCUITO, COMPROBACIÓN Y DEPURACIÓN.

6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

Ejemplo 1 Máquina tipo MOORE

ETAPA 1: Especificación del problema → Diagramas de estado → Tablas de estado

Enunciado:

Diseñar un circuito **TIPO MOORE** que consta de una entrada “X” y una salida “Z”, tal que $Z=1$ cuando X haya sido $X=1$ durante 3 o más ciclos consecutivos de reloj.

Realizar el diseño utilizando biestables JK

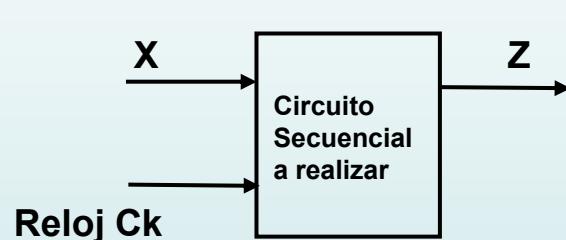


Tabla de estados

Estado Actual	Salida Actual Z	Estado siguiente	
		X=0	X=1
A	0	A	B
B	0	A	C
C	0	A	D
D	1	A	D

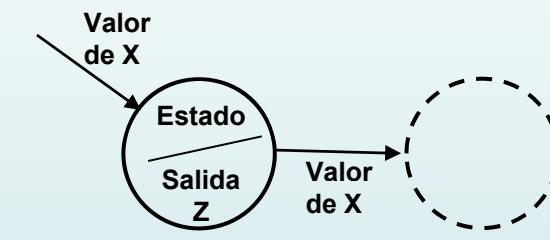
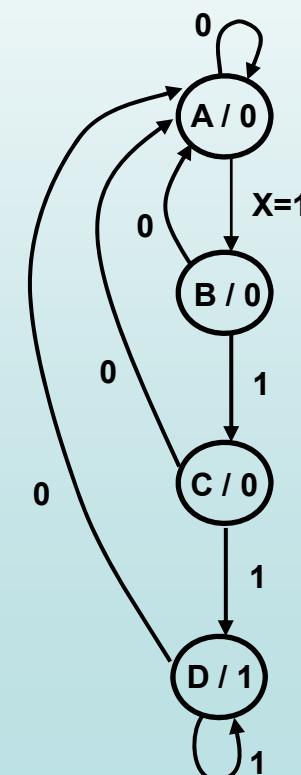
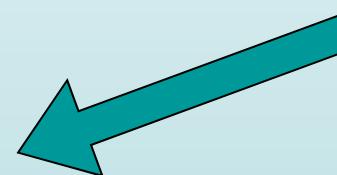


Diagrama de estados



6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

ETAPA 2: Minimización de la tabla de estados. Este punto se verá más adelante. En este ejemplo, la tabla de estados de la etapa 1 ya es mínima.

ETAPA 3: Asignación de estados. Elección del tipo de biestables a utilizar. Tabla de transiciones. El enunciado del problema dice que se realice con biestables JK.

Tabla de estados

Estado Actual	Estado siguiente / Salida actual	
	X=0	X=1
A	A / 0	B / 0
B	A / 0	C / 0
C	A / 0	C / 1

Q Q ⁺	J K
0 0	0 -
0 1	1 -
1 0	- 1
1 1	- 0

$$J_1(Q_1 Q_0 X) = \sum m_i(3) + d(4, 5, 6, 7)$$

$$K_1(Q_1 Q_0 X) = \sum m_i(4) + d(0, 1, 2, 3, 6, 7)$$

$$J_0(Q_1 Q_0 X) = \sum m_i(1) + d(2, 3, 6, 7)$$

$$K_0(Q_1 Q_0 X) = \sum m_i(2, 3) + d(0, 1, 4, 5, 6, 7)$$

$$Z(Q_1 Q_0 X) = \sum m_i(5) + d(6, 7)$$

Tabla de transiciones

Estado actual y Asignación de estados	Estado Siguiente		Funciones a realizar							
	Q ₁ ⁺ Q ₀ ⁺		J ₁ K ₁		j ₀ K ₀		Z			
	Q ₁	Q ₀	X=0	X=1	X=0	X=1	X=0	X=1	X=0	X=1
A =	0	0	0 0 0	0 1 1	0 - 0	0 - 1	0 - 0	1 - 1	0 0 0	0 1 1
B =	0	1	0 0 2	1 0 3	0 - 2	1 - 3	- 1 2	- 1 3	0 2 0	0 3 1
C =	1	0	0 0 4	1 0 5	- 1 4	- 0 5	0 - 4	0 - 5	0 4 0	1 5 1
-	1	1	- - 6	- - 7	- - 6	- - 7	- - 6	- - 7	- - 6	- - 7

6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

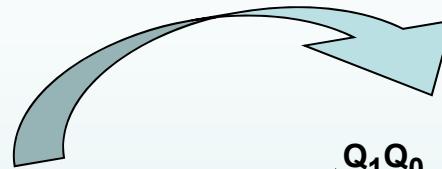
ETAPA 4: Minimización de las funciones, y esquema del circuito.

$$J_1(Q_1 Q_0 X) = \sum m_i(2) + d(4, 6, 5, 7)$$

$$K_1(Q_1 Q_0 X) = \sum m_i(4, 6) + d(0, 1, 2, 3)$$

$$J_0(Q_1 Q_0 X) = \sum m_i(1, 5) + d(2, 6, 3, 7)$$

$$K_0(Q_1 Q_0 X) = \sum m_i(2, 3, 6) + d(0, 1, 4, 5)$$



		Q ₁ Q ₀	00	01	11	10
		X	0	2	-6	-4
Q ₁	Q ₀	0	0	2	-6	-4
		1	1	3	-7	-5

$$J_1(Q_1 Q_0 X) = X Q_0$$

		Q ₁ Q ₀	00	01	11	10
		X	-0	-2	16	14
Q ₁	Q ₀	0	-0	-2	16	14
		1	-1	-3	7	5

$$K_1(Q_1 Q_0 X) = \bar{X}$$

$$Z(Q_1 Q_0) = \sum m_i(3)$$

		Q ₁	0	1
		Q ₀	0	2
Q ₁	Q ₀	0	0	2
		1	1	3

$$Z = Q_1 Q_0$$

		Q ₁ Q ₀	00	01	11	10
		X	0	-2	-6	4
Q ₁	Q ₀	0	0	-2	-6	4
		1	1	-3	-7	5

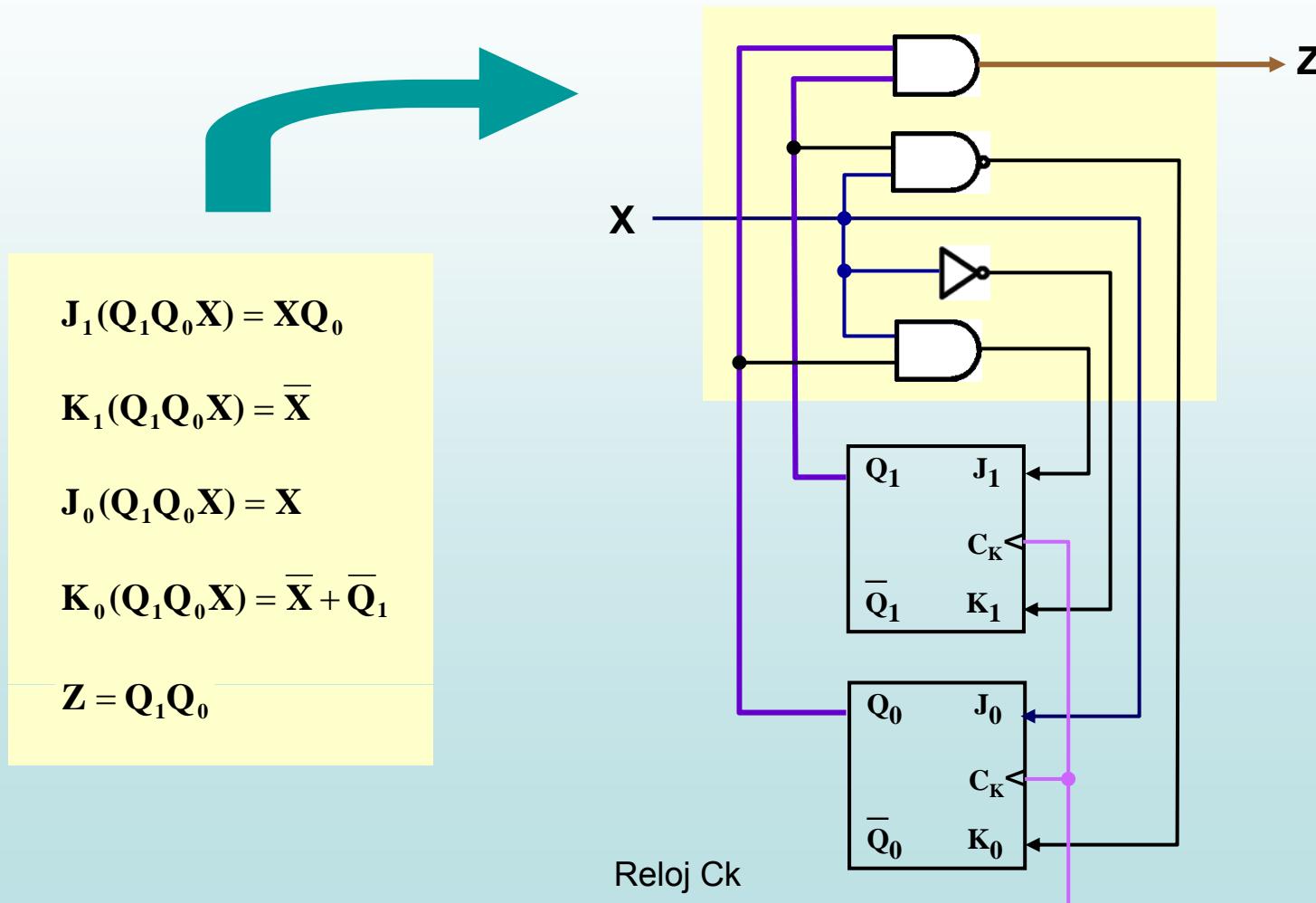
$$J_0(Q_1 Q_0 X) = X$$

		Q ₁ Q ₀	00	01	11	10
		X	-0	12	16	-4
Q ₁	Q ₀	0	-0	12	16	-4
		1	-1	13	7	-5

$$K_0(Q_1 Q_0 X) = \bar{X} + \bar{Q}_1$$

6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

ETAPA 4: (Continuación). Esquema del circuito.



6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

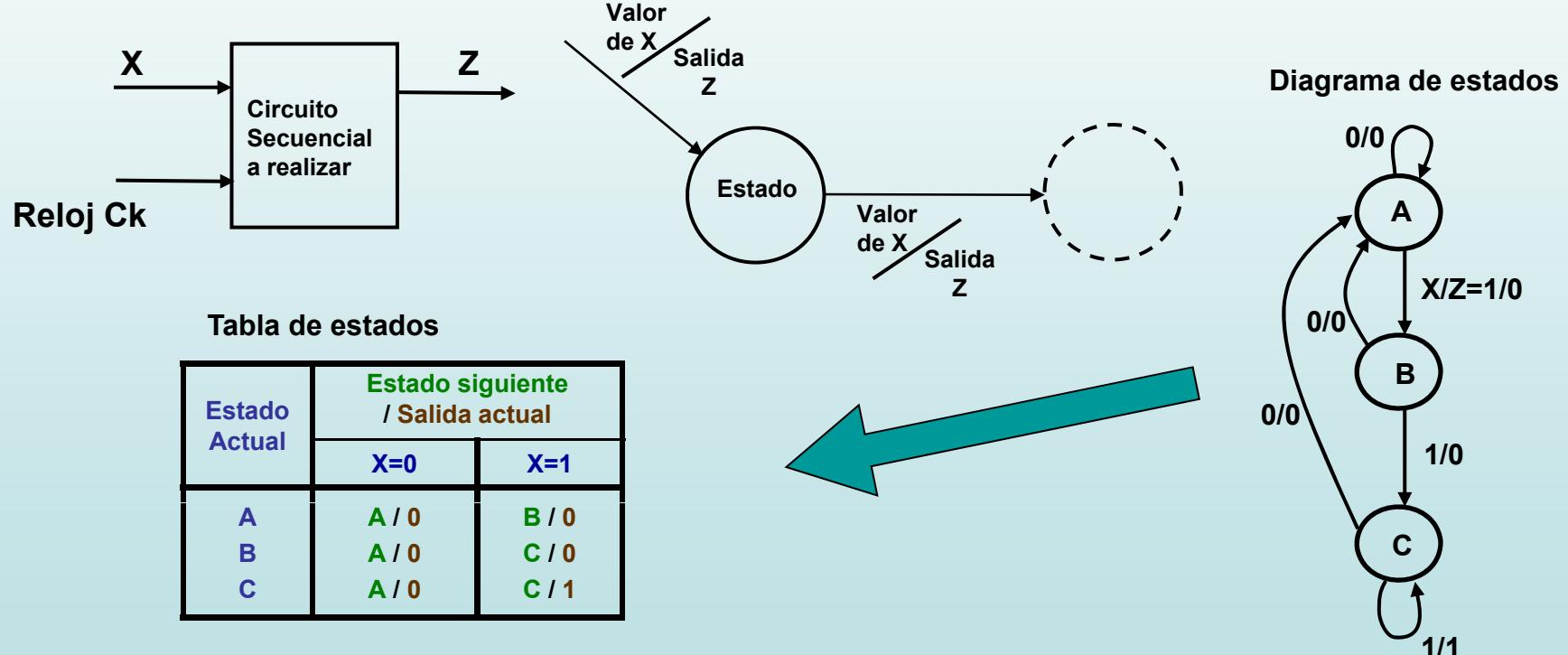
Ejemplo_2 Máquina tipo MEALY

ETAPA 1: Especificación del problema → Diagramas de estado → Tablas de estado

Enunciado:

Diseñar un circuito **TIPO MEALY** que consta de una entrada “X” y una salida “Z”, tal que $Z=1$ cuando X haya sido $X=1$ durante 3 o más ciclos consecutivos de reloj.

Realizar el diseño utilizando biestables JK



6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

ETAPA 2: Minimización de la tabla de estados. Este punto se verá más adelante. En este ejemplo, la tabla de estados de la etapa 1 ya es mínima.

ETAPA 3: Asignación de estados. Elección del tipo de biestables a utilizar. Tabla de transiciones. El enunciado del problema dice que se realice con biestables JK.

Tabla de estados

Estado Actual	Estado siguiente / Salida actual	
	X=0	X=1
A	A / 0	B / 0
B	A / 0	C / 0
C	A / 0	C / 1

Q Q ⁺	J K
0 0	0 -
0 1	1 -
1 0	- 1
1 1	- 0

$$J_1 (Q_1 Q_0 X) = \sum m_i(3) + d(4, 5, 6, 7)$$

$$K_1 (Q_1 Q_0 X) = \sum m_i(4) + d(0, 1, 2, 3, 6, 7)$$

$$J_0 (Q_1 Q_0 X) = \sum m_i(1) + d(2, 3, 6, 7)$$

$$K_0 (Q_1 Q_0 X) = \sum m_i(2, 3) + d(0, 1, 4, 5, 6, 7)$$

$$Z (Q_1 Q_0 X) = \sum m_i(5) + d(6, 7)$$

Tabla de transiciones

Estado actual y Asignación de estados	Estado Siguiente		Funciones a realizar							
	Q ₁ ⁺ Q ₀ ⁺		J ₁ K ₁		j ₀ K ₀		Z			
	Q ₁	Q ₀	X=0	X=1	X=0	X=1	X=0	X=1	X=0	X=1
A =	0	0	0	0	0	1	1	0	0	1
B =	0	1	0	0	2	1	0	3	0	2
C =	1	0	0	0	4	1	0	5	1	4
-	1	1	-	-	6	-	7	-	6	5

6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

ETAPA 4: Minimización de las funciones, (Por Eje. Mapas de Karnaugh) y esquema del circuito.

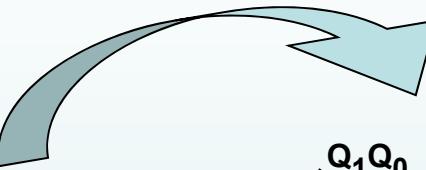
$$J_1(Q_1 Q_0 X) = \sum m_i(3) + d(4, 5, 6, 7)$$

$$K_1(Q_1 Q_0 X) = \sum m_i(4) + d(0, 1, 2, 3, 6, 7)$$

$$J_0(Q_1 Q_0 X) = \sum m_i(1) + d(2, 3, 6, 7)$$

$$K_0(Q_1 Q_0 X) = \sum m_i(2, 3) + d(0, 1, 4, 5, 6, 7)$$

$$Z(Q_1 Q_0 X) = \sum m_i(5) + d(6, 7)$$



		Q ₁ Q ₀	00	01	11	10
		X	0	2	-6	-4
		0	0	2	-6	-4
		1	1	3	-7	-5

$$J_1(Q_1 Q_0 X) = X Q_0$$

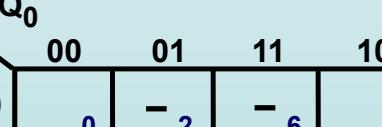
		Q ₁ Q ₀	00	01	11	10
		X	0	-2	-6	4
		0	0	-2	-6	4
		1	1	-3	-7	5

$$J_0(Q_1 Q_0 X) = X \bar{Q}_1$$



		Q ₁ Q ₀	00	01	11	10
		X	-0	-2	-6	14
		0	-0	-2	-6	14
		1	-1	-3	-7	5

$$K_1(Q_1 Q_0 X) = \bar{X}$$



		Q ₁ Q ₀	00	01	11	10
		X	-0	12	-6	-4
		0	-0	12	-6	-4
		1	-1	13	-7	-5

$$K_0(Q_1 Q_0 X) = 1$$

6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

ETAPA 4: (Continuación). Esquema del circuito.

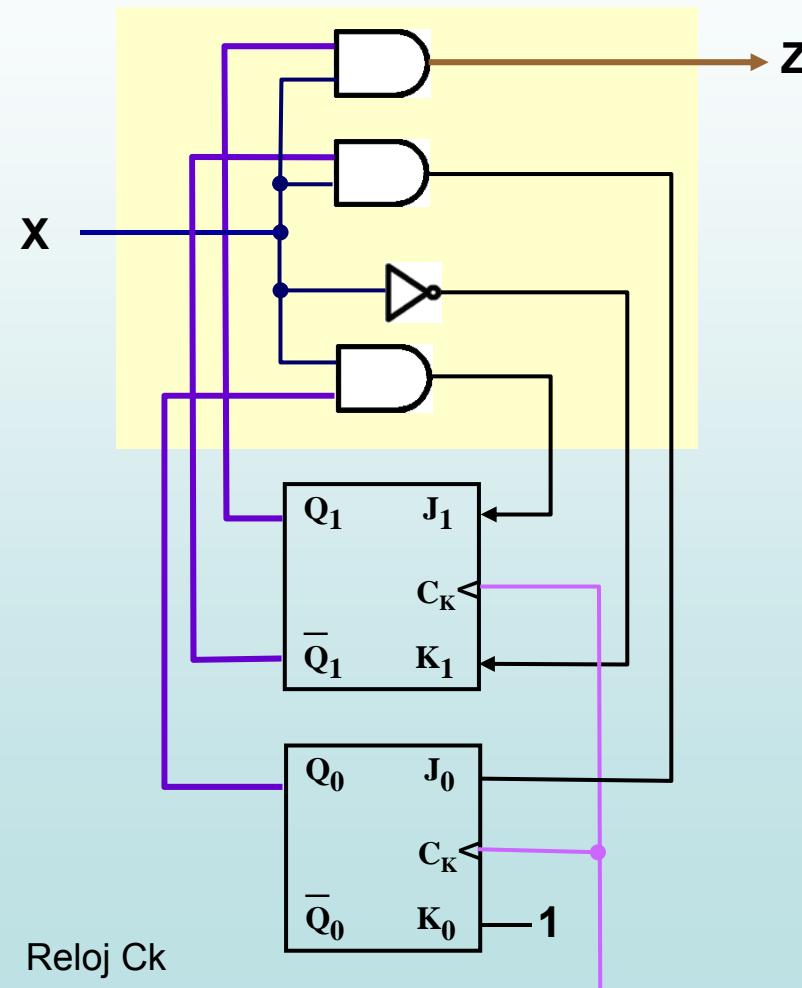
$$J_1(Q_1 Q_0 X) = X Q_0$$

$$K_1(Q_1 Q_0 X) = \bar{X}$$

$$J_0(Q_1 Q_0 X) = X \bar{Q}_1$$

$$K_0(Q_1 Q_0 X) = 1$$

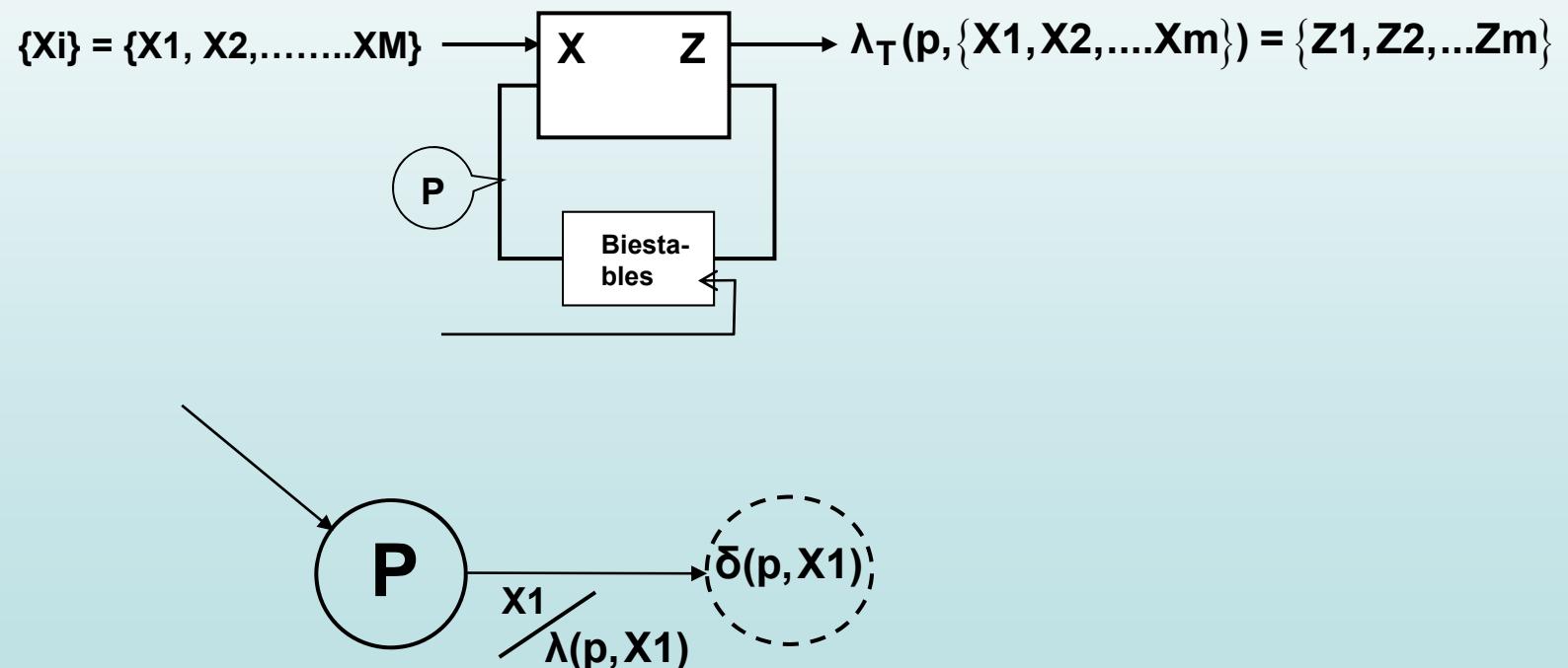
$$Z = X Q_1$$



6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

Minimización de tablas de estados

MÉTODO DE TABLAS DE IMPLICACIÓN PARA MINIMIZAR TABLAS DE ESTADO COMPLETAMENTE ESPECIFICADAS.



6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

Definición de estados equivalentes.

Estados equivalentes.

Sean S y T dos sistemas secuenciales **COMPLETAMENTE ESPECIFICADOS** sujetos a las mismas secuencias de entradas.

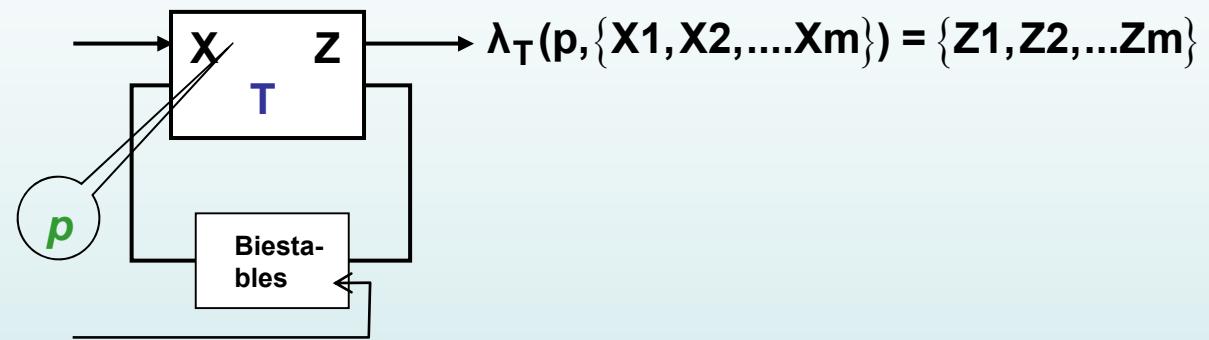
Dos estados p (de T) y q (de S) son equivalentes si y solo si: estando T en el estado p y S en el estado q , para cada secuencia posible de entrada implica que la secuencia de salida con que responden uno y otro circuito coinciden. Es decir:

Si $\{X\}=\{x_1, x_2, \dots, x_n\}$ es una secuencia de entrada genérica

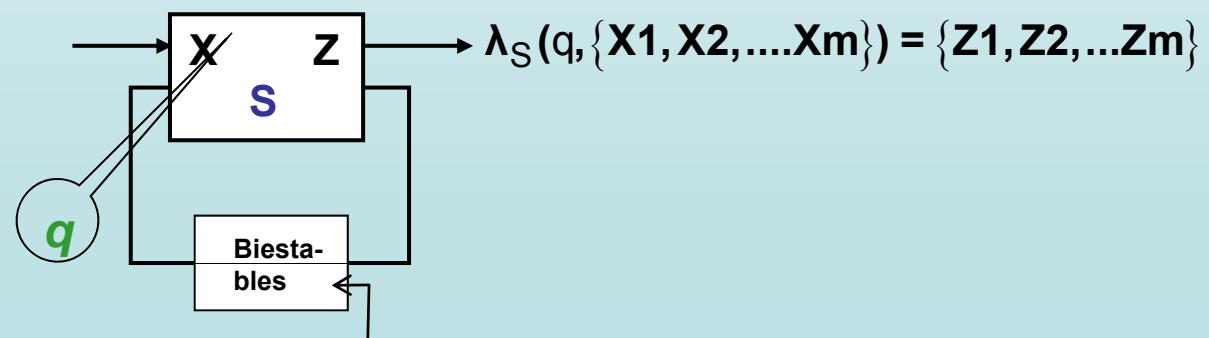
$$p \equiv q \Leftrightarrow \forall \{X\} \Rightarrow \lambda_T(p, \{X\}) = \lambda_S(q, \{X\})$$

6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

$$\{X_i\} = \{X_1, X_2, \dots, X_M\}$$



$$\{X_i\} = \{X_1, X_2, \dots, X_M\}$$



6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

Definición de Circuitos Secuenciales equivalentes.

Sistemas Secuenciales equivalentes.

Dos sistemas secuenciales S y T son equivalentes si para cada estado p (de T) existe un estado q (de S) tal que p es equivalente a q e, inversamente; para cada estado q (de S) existe un estado p (de T) tal que q es equivalente a p .

$$S \equiv T \Leftrightarrow \begin{cases} \forall p \text{ de } T \Rightarrow \exists q \text{ de } S / p \equiv q \\ \forall q \text{ de } S \Rightarrow \exists p \text{ de } T / q \equiv p \end{cases}$$

6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

Minimización de tablas de estados

TEOREMA:

Dos estados p y q de un mismo sistema son equivalentes sí y solo sí para cada entrada X las salidas que se obtienen son idénticas y los estados siguientes son equivalentes. Es decir:

$$p \equiv q \Leftrightarrow \forall X \begin{cases} 1) \quad \lambda(p, X) = \lambda(q, X) \\ 2) \quad \delta(p, X) \equiv \delta(q, X) \end{cases}$$

6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

Minimización de tablas de estados

Paso preliminar: Obtención de la tabla de estados reducida

Tabla de estados

Estado actual	Estado Siguiente / Salida Z	
	X=0	x=1
s1	s2 , 0	s3 , 0
s2	s4 , 0	s5 , 0
s3	s6 , 0	s7 , 0
s4	s8 , 0	s9 , 0
s5	s10 , 0	s11 , 0
s6	s4 , 0	s11 , 0
s7	s10 , 0	s11 , 0
s8	s8 , 0	s1 , 0
s9	s10, 1	s1, 1
s10	s4 , 0	s1 , 0
s11	s2 , 0	s1 , 0
s12	s10 , 1	s1 , 1

Tabla de estados reducida

Estado actual	Estado Siguiente / Salida Z	
	X=0	x=1
s1	s2 , 0	s3 , 0
s2	s4 , 0	s5 , 0
s3	s6 , 0	s5 , 0
s4	s8 , 0	s9 , 0
s5	s10 , 0	s11 , 0
s6	s4 , 0	s11 , 0
s7	s10 , 0	s11 , 0
s8	s8 , 0	s1 , 0
s9	s10, 1	s1, 1
s10	s4 , 0	s1 , 0
s11	s2 , 0	s1 , 0
s9	s10 , 1	s1 , 1



6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

1º paso: Construir la tabla de implicación. Marcar casillas de pares de estados no equivalentes por presentar salidas distintas. Rellenar el resto de las casillas con los pares de estados implicados.

Estado actual	Estado Siguiente / Salida Z	
	X=0	x=1
s1	s2 , 0	s3 , 0
s2	s4 , 0	s5 , 0
s3	s6 , 0	s5 , 0
s4	s8 , 0	s9 , 0
s5	s10 , 0	s11 , 0
s6	s4 , 0	s11 , 0
s8	s8 , 0	s1 , 0
s10	s4 , 0	s1 , 0
s11	s2 , 0	s1 , 0
s9	s10 , 1	s1 , 1

s2	s2 - s4 s3 - s5								
s3	s2 - s6 s3 - s5	s4 - s6							
s4	s2 - s8 s3 - s9	s4 - s8 s5 - s9	s6 - s8 s5 - s9						
s5	s2 - s10 s3 - s11	s4 - s10 s5 - s11	s6 - s10 s5 - s11	s8 - s10 s9 - s11					
s6	s2 - s4 s3 - s11	s5 - s11	s6 - s4 s5 - s11	s8 - s4 s9 - s11	s10 - s4				
s8	s2 - s8 s3 - s1	s4 - s8 s5 - s1	s6 - s8 s5 - s1	s9 - s1	s10 - s8 s11 - s1	s4 - s8 s11 - s1			
s10	s2 - s4 s3 - s1	s5 - s1	s6 - s4 s5 - s1	s8 - s4 s9 - s1	s10 - s4 s11 - s1	s4 - s4 s11 - s1	s8 - s4		
s11	s3 - s1	s4 - s2 s5 - s1	s6 - s2 s5 - s1	s8 - s2 s9 - s1	s10 - s2 s11 - s1	s4 - s2 s11 - s1	s8 - s2	s4 - s2	
s9									
	s1	s2	s3	s4	s5	s6	s8	s10	s11

6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

2º paso: Revisar todas las casillas y tachar aquellas que contengan algún par implicado de estados no equivalentes.

	s1	s2	s3	s4	s5	s6	s8	s10	s11	s9
s2	s2 - s4 s3 - s5									
s3	s2 - s6 s3 - s5	s4 - s6								
s4	s2 - s8 s3 - s9	s4 - s8 s5 - s9	s6 - s8 s5 - s9							
s5	s2 - s10 s3 - s11	s4 - s10 s5 - s11	s6 - s10 s5 - s11	s8 - s10 s9 - s11						
s6	s2 - s4 s3 - s11	s5 - s11	s6 - s4 s5 - s11	s8 - s4 s9 - s11	s10 - s4					
s8	s2 - s8 s3 - s1	s4 - s8 s5 - s1	s6 - s8 s5 - s1	s9 - s1	s10 - s8 s11 - s1	s4 - s8 s11 - s1				
s10	s2 - s4 s3 - s1	s5 - s1	s6 - s4 s5 - s1	s8 - s4 s9 - s1	s10 - s4 s11 - s1	s4 - s4 s11 - s1	s8 - s4			
s11	s3 - s1	s4 - s2 s5 - s1	s6 - s2 s5 - s1	s8 - s2 s9 - s1	s10 - s2 s11 - s1	s4 - s2 s11 - s1	s8 - s2	s4 - s2		
s9										

Se induce la NO EQUIVALENCIA de s4 con s5 ya que contiene al par implicado [s9 - s11] y resulta que s9 no es equivalente a s11 dado que la casilla de coordenadas (s9-s11) esta tachada (no son equivalentes). Por tanto se tacha la casilla de coordenadas (s4, s5)

6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

Se continúa el paso 2º : Revisar todas las casillas y tachar aquellas que contengan algún par implicado de estados no equivalentes.

s2	s2 - s4 s3 - s5								
s3	s2 - s6 s3 - s5	s4 - s6							
s4									
s5	s2 - s10 s3 - s11	s4 - s10 s5 - s11	s6 - s10 s5 - s11						
s6	s2 - s4 s3 - s11	s5 - s11	s6 - s4 s5 - s11		s10 - s4				
s8	s2 - s8 s3 - s1	s4 - s8 s5 - s1	s6 - s8 s5 - s1		s10 - s8 s11 - s1	s4 - s8 s11 - s1			
s10	s2 - s4 s3 - s1	s5 - s1	s6 - s4 s5 - s1		s10 - s4 s11 - s1	s11 - s1	s8 - s4		
s11	s3 - s1	s4 - s2 s5 - s1	s6 - s2 s5 - s1		s10 - s2 s11 - s1	s4 - s2 s11 - s1	s8 - s2	s4 - s2	
s9									
	s1	s2	s3	s4	s5	s6	s8	s10	s11

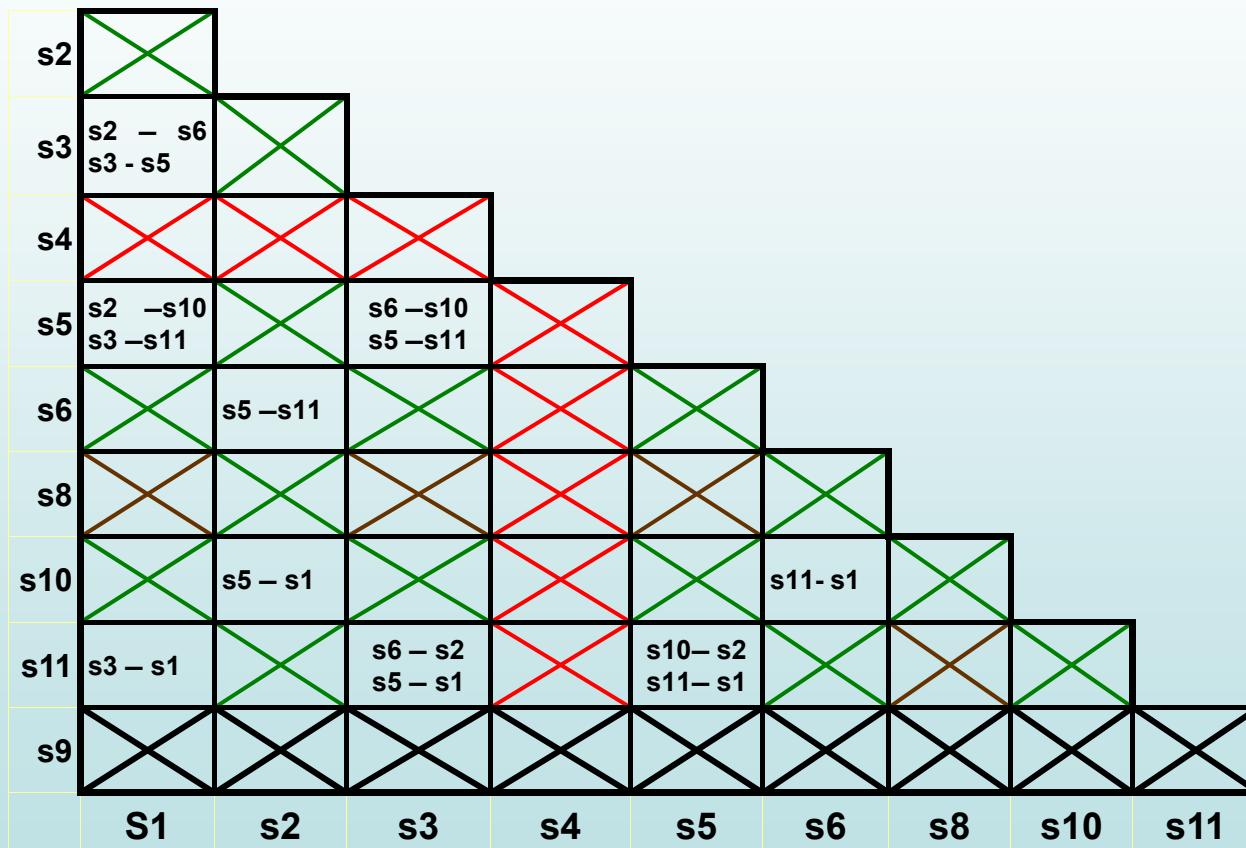
6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

Se continúa el paso 2º : Revisar todas las casillas y tachar aquellas que contengan algún par implicado de estados no equivalentes.

s2									
s3	s2 - s6 s3 - s5								
s4									
s5	s2 - s10 s3 - s11		s6 - s10 s5 - s11						
s6		s5 - s11							
s8	s2 - s8 s3 - s1		s6 - s8 s5 - s1		s10 - s8 s11 - s1				
s10		s5 - s1				s11 - s1			
s11	s3 - s1		s6 - s2 s5 - s1		s10 - s2 s11 - s1		s8 - s2		
s9									
	s1	s2	s3	s4	s5	s6	s8	s10	s11

6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

3º paso: Repetir el paso 2º hasta que al inspeccionar todas las casillas ya no se pueda tachar ninguna. En ese caso, las casillas no marcadas revelan los pares de estados coordenadas que son equivalentes.



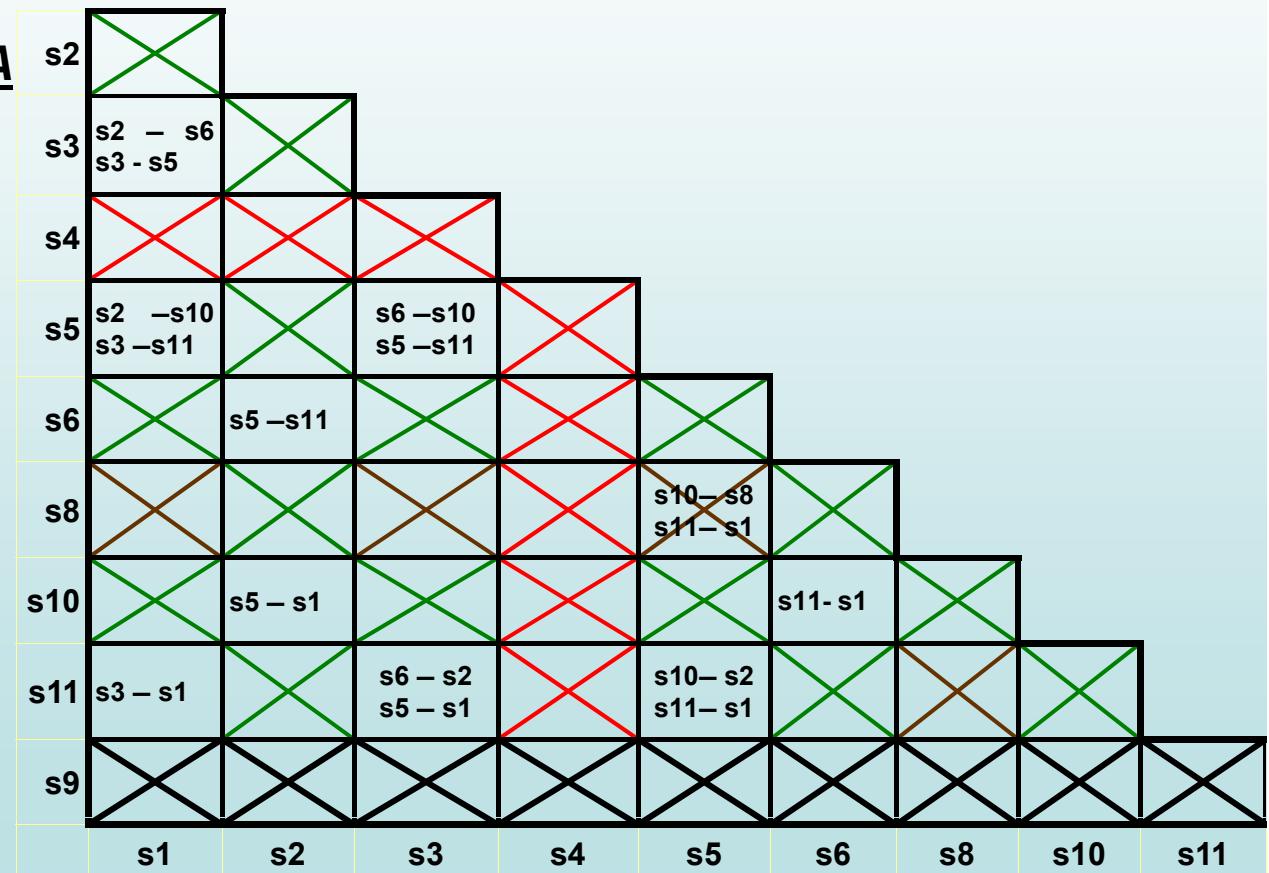
Finaliza el paso 3º): Tras revisar todas las casillas, no es posible tachar ninguna.

6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

4º paso: Finalizado el paso 3º , las casillas de la tabla de implicación no marcadas revelan los pares de estados coordinadas que son equivalentes. A partir de los pares equivalentes se obtienen las clases de equivalencia.

CLASES DE EQUIVALENCIA

$A = \{ s_1, s_3, s_5, s_{11} \}$
$B = \{ s_2, s_6, s_{10} \}$
$C = \{ s_4 \}$
$D = \{ s_8 \}$
$E = \{ s_9 \}$

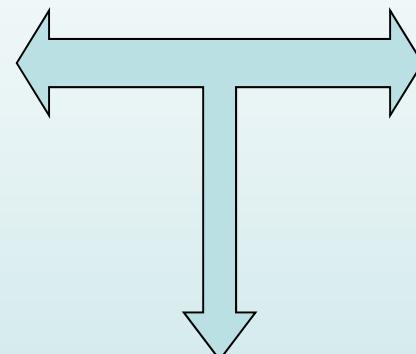


6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

5º paso: A partir de las clases de equivalencia se obtiene la tabla de estados minimizada, especificando para cada clase de estado actual (filas de la tabla) las clases siguientes y salida actual correspondientes a cada combinación de los valores de las entradas (columnas de la tabla).

A = { s1, s3, s5, s11 }
B = { s2, s6, s10 }
C = { s4 }
D = { s8 }
E = { s9 }

CLASES DE EQUIVALENCIA



**Tabla de estados
minimizada**

Estado actual	Estado Siguiente / Salida Z	
	X=0	x=1
A	B , 0	A , 0
B	C , 0	A , 0
C	D , 0	E , 0
D	D , 0	A , 0
E	B , 1	A , 1

Estado actual	Estado Siguiente / Salida Z	
	X=0	x=1
A	s2 , 0	s3 , 0
B	s4 , 0	s5 , 0
A	s6 , 0	s5 , 0
C	s8 , 0	s9 , 0
A	s10 , 0	s11 , 0
B	s4 , 0	s11 , 0
D	s8 , 0	s1 , 0
B	s4 , 0	s1 , 0
A	s2 , 0	s1 , 0
E	s10 , 1	s1 , 1

**Tabla de estados
original reducida**

6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

EJEMPLO DE DISEÑO DE UN CIRCUITO SECUENCIAL SÍNCRONO SIGUIENDO TODOS LOS PASOS DEL DISEÑO.

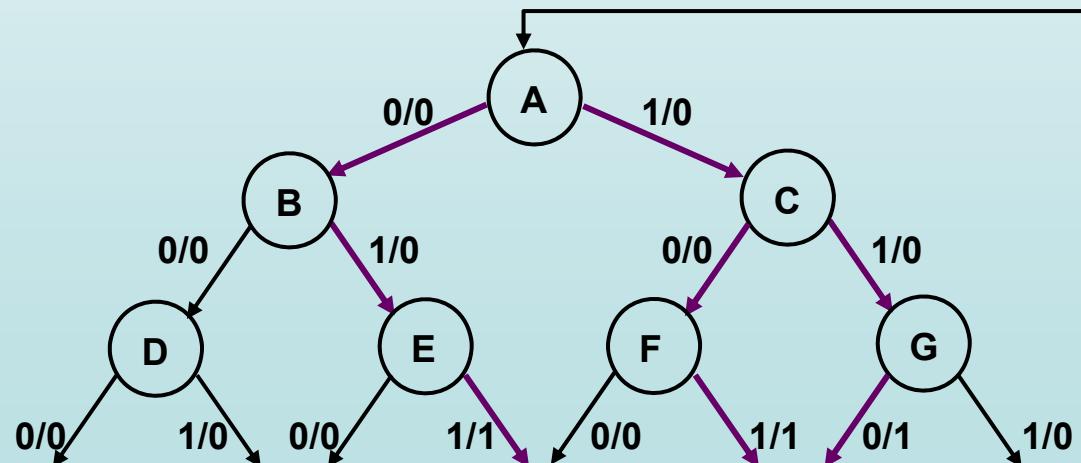
Especificación del circuito:

Un sistema recibe a través de una línea serie códigos de 3 caracteres codificados según la tabla adjunta. La salida del circuito debe indicar con un “1” la recepción de un código incorrecto. Realizar el circuito utilizando biestables tipo D.

Código	Carácter
0 0 0	A
0 0 1	E
0 1 0	I
1 0 0	O
1 1 1	U

Códigos erróneos
0 1 1
1 0 1
1 1 0

ETAPA 1: Especificación del problema → Diagramas de estado → Tablas de estado



Estado actual	Estado Siguiente / Salida Z	
	X=0	x=1
A	B , 0	C , 0
B	D , 0	E , 0
C	F , 0	G , 0
D	A , 0	A , 0
E	A , 0	A , 1
F	A , 0	A , 1
G	A , 1	A , 0

6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

Tabla de estados

Estado actual	Estado Siguiente / Salida Z	
	X=0	x=1
A	B , 0	C , 0
B	D , 0	E , 0
C	F , 0	G , 0
D	A , 0	A , 0
E	A , 0	A , 1
F	A , 0	A , 1
G	A , 1	A , 0

Son equivalentes

Tabla de estados reducida

Estado actual	Estado Siguiente / Salida Z	
	X=0	x=1
A	B , 0	C , 0
B	D , 0	E , 0
C	E , 0	G , 0
D	A , 0	A , 0
E	A , 0	A , 1
G	A , 1	A , 0

ETAPA 2: Minimización de la tabla de estados.

B	B - D C - E				
C	B - E C - G	D - E E - G			
D	B - A C - A	D - A E - A	E - A G - A		
E					
G					
	A	B	C	D	E

La tabla de implicación no presenta cuadrados sin marcar. Esto significa que la tabla reducida es mínima

6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

ETAPA 3: Asignación de estados. Elección del tipo de biestables a utilizar. Tabla de transiciones. Expresión canónica de las funciones de excitación de los biestables y de las salidas.

Estado actual	Est. Siguiente/ Sal Z	
	X=0	x=1
A	B , 0	C , 0
B	D , 0	E , 0
C	E , 0	G , 0
D	A , 0	A , 0
E	A , 0	A , 1
G	A , 1	A , 0

$$D_2(Q_2, Q_1, Q_0, X) = \sum m_i(0,1) + d(8,9,10,11)$$

$$D_1(Q_2, Q_1, Q_0, X) = \sum m_i(0,1,12,13,14) + d(8,9,10,11)$$

$$D_0(Q_2, Q_1, Q_0, X) = \sum m_i(1,13,14,15) + d(8,9,10,11)$$

$$Z(Q_2, Q_1, Q_0, X) = \sum m_i(2,7) + d(8,9,10,11)$$

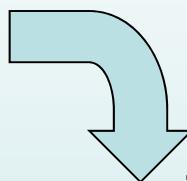


Tabla de transiciones

Estado actual y Asignación de estados $Q_2 \ Q_1 \ Q_0$	Estado Siguiente		Funciones a realizar							
	$Q_2^+ \ Q_1^+ \ Q_0^+$		$D_2 = Q_2^+$		$D_1 = Q_1^+$		$D_0 = Q_0^+$		Z	
	X=0	X=1	X=0	X=1	X=0	X=1	X=0	X=1	X=0	X=1
A = 0 0 0	1 1 0	0	1 1 1	1	1	0	1	1	0	0
G = 0 0 1	0 0 0	2	0 0 0	3	0	2	0	3	0	2
D = 0 1 0	0 0 0	4	0 0 0	5	0	4	0	5	0	4
E = 0 1 1	0 0 0	6	0 0 0	7	0	6	0	7	0	6
--	1 0 0	- - -	8	- - -	9	- -	8	- -	9	- -
--	1 0 1	- - -	10	- - -	11	- -	10	- -	11	- -
B = 1 1 0	0 1 0	12	0 1 1	13	0	12	0	13	1	12
C = 1 1 1	0 1 1	14	0 0 1	15	0	14	0	15	1	14

6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

ETAPA 4: Minimización de las funciones, (Por Eje. Con Mapas de Karnaugh) y esquema del circuito.

$Q_2 Q_1$	00	01	11	10
$Q_0 X$	1 0	0 4	0 12	-- 8
	0 1	1 1	0 5	0 13
	1 1	0 3	0 7	0 15
	0 2	0 6	0 14	-- 10

D_2

$Q_2 Q_1$	00	01	11	10
$Q_0 X$	1 0	0 4	1 12	-- 8
	0 1	1 1	0 5	1 13
	1 1	0 3	0 7	0 15
	0 2	0 6	1 14	-- 10

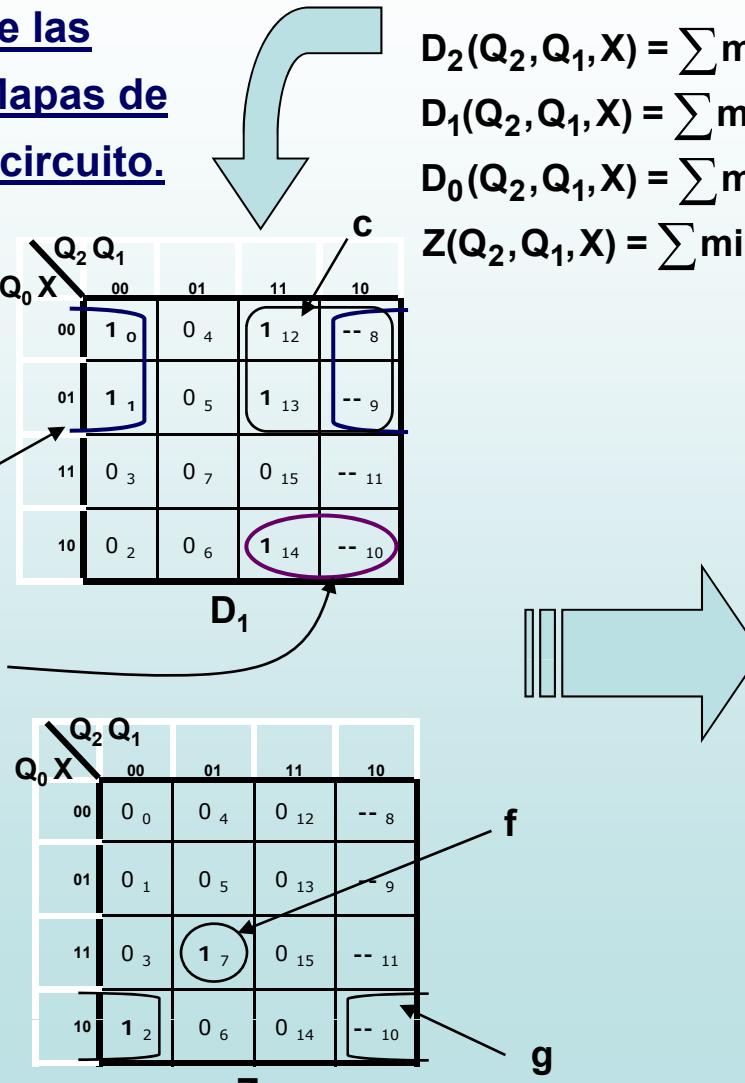
D_1

$Q_2 Q_1$	00	01	11	10
$Q_0 X$	0 0	0 4	0 12	-- 8
	1 1	0 5	1 13	-- 9
	0 3	0 7	1 15	-- 11
	0 2	0 6	1 14	-- 10

D_0

$Q_2 Q_1$	00	01	11	10
$Q_0 X$	0 0	0 4	0 12	-- 8
	0 1	0 5	0 13	-- 9
	0 3	0 7	1 7	0 15
	1 2	0 6	0 14	-- 10

Z



$$D_2(Q_2, Q_1, X) = \sum m_i(0, 1) + d(8, 9, 10, 11)$$

$$D_1(Q_2, Q_1, X) = \sum m_i(0, 1, 12, 13, 14) + d(8, 9, 10, 11)$$

$$D_0(Q_2, Q_1, X) = \sum m_i(1, 13, 14, 15) + d(8, 9, 10, 11)$$

$$Z(Q_2, Q_1, X) = \sum m_i(2, 7) + d(8, 9, 10, 11)$$

$$a = \overline{Q}_1 \overline{Q}_0$$

$$b = Q_2 Q_1 \overline{X}$$

$$c = Q_2 \overline{Q}_0$$

$$d = \overline{Q}_1 \overline{Q}_0 X$$

$$e = Q_2 X$$

$$f = \overline{Q}_2 Q_1 Q_0 X$$

$$g = \overline{Q}_1 Q_0 \overline{X}$$

$$D_2 = a$$

$$D_1 = a + b + c$$

$$D_0 = b + d + e$$

$$Z = f + g$$

6.4 DISEÑO DE UN SISTEMA SECUENCIAL.

ETAPA 4: (Continuación)

Esquema del circuito.

$$a = \overline{Q_1} \overline{Q_0}$$

$$b = Q_2 Q_1 \overline{X}$$

$$c = Q_2 \overline{Q_0}$$

$$d = \overline{Q_1} \overline{Q_0} X$$

$$e = Q_2 X$$

$$f = \overline{Q_2} Q_1 Q_0 X$$

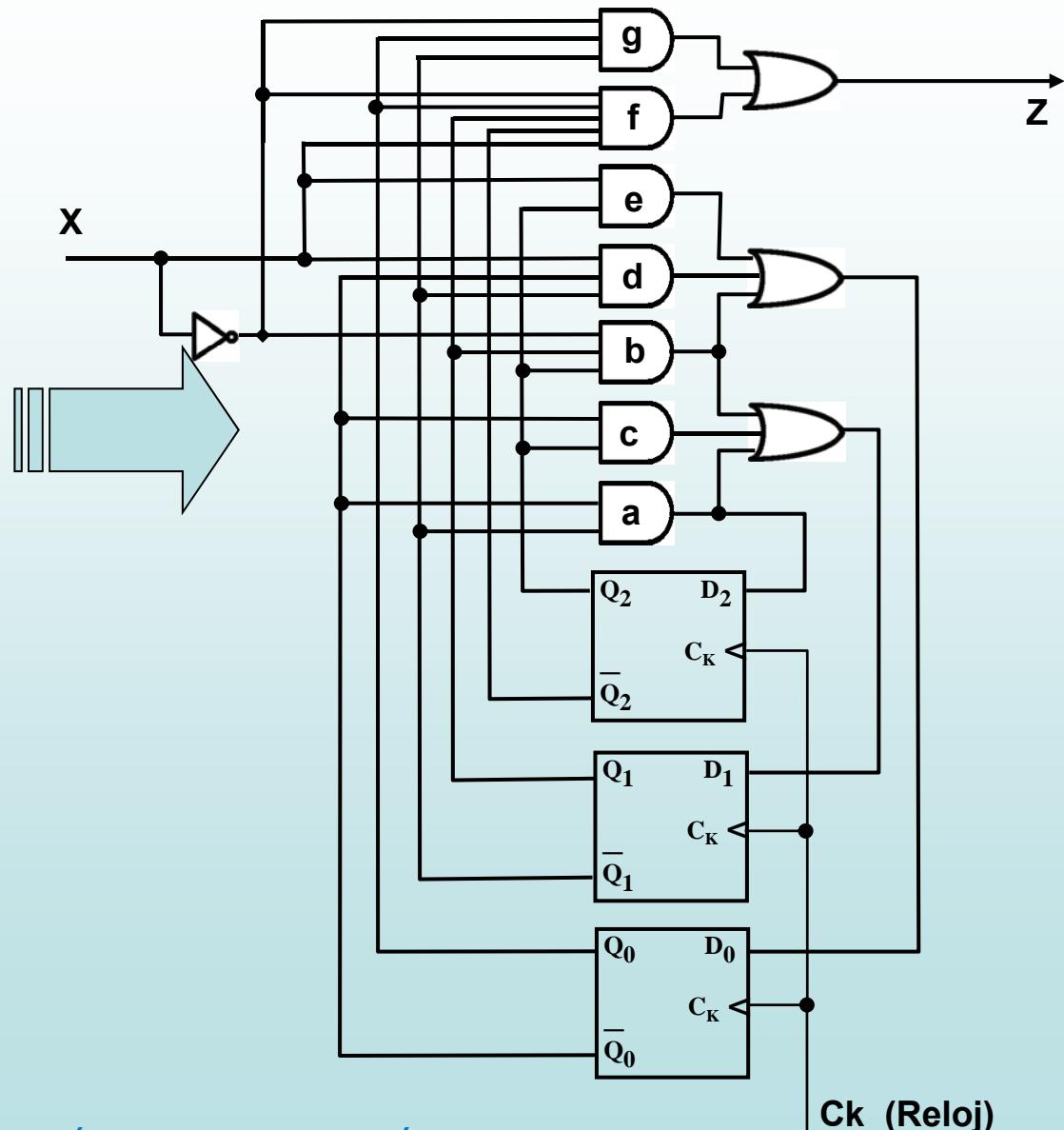
$$g = \overline{Q_1} Q_0 \overline{X}$$

$$D_2 = a$$

$$D_1 = a + b + c$$

$$D_0 = b + d + e$$

$$Z = f + g$$



6.4 DISEÑO DE UN SISTEMA SECUENCIAL. Ejemplos de minimización de tablas de estados

- Ejemplo de minimización de estados:

E.P.	X_1X_0	00	01	10	11
A		D, 0	D, 0	F, 0	A, 0
B		I, 1	D, 0	E, 1	F, 0
C		I, 1	D, 0	E, 1	A, 0
D		D, 0	B, 0	A, 0	F, 0
E		C, 1	J, 0	E, 1	A, 0
F		D, 0	D, 0	A, 0	F, 0
G		G, 0	G, 0	A, 0	A, 0
H		B, 1	D, 0	E, 1	A, 0
I		I, 1	D, 0	E, 1	A, 0
J		D, 0	D, 0	A, 0	F, 0

6.4 DISEÑO DE UN SISTEMA SECUENCIAL. Ejemplos de minimización de tablas de estados

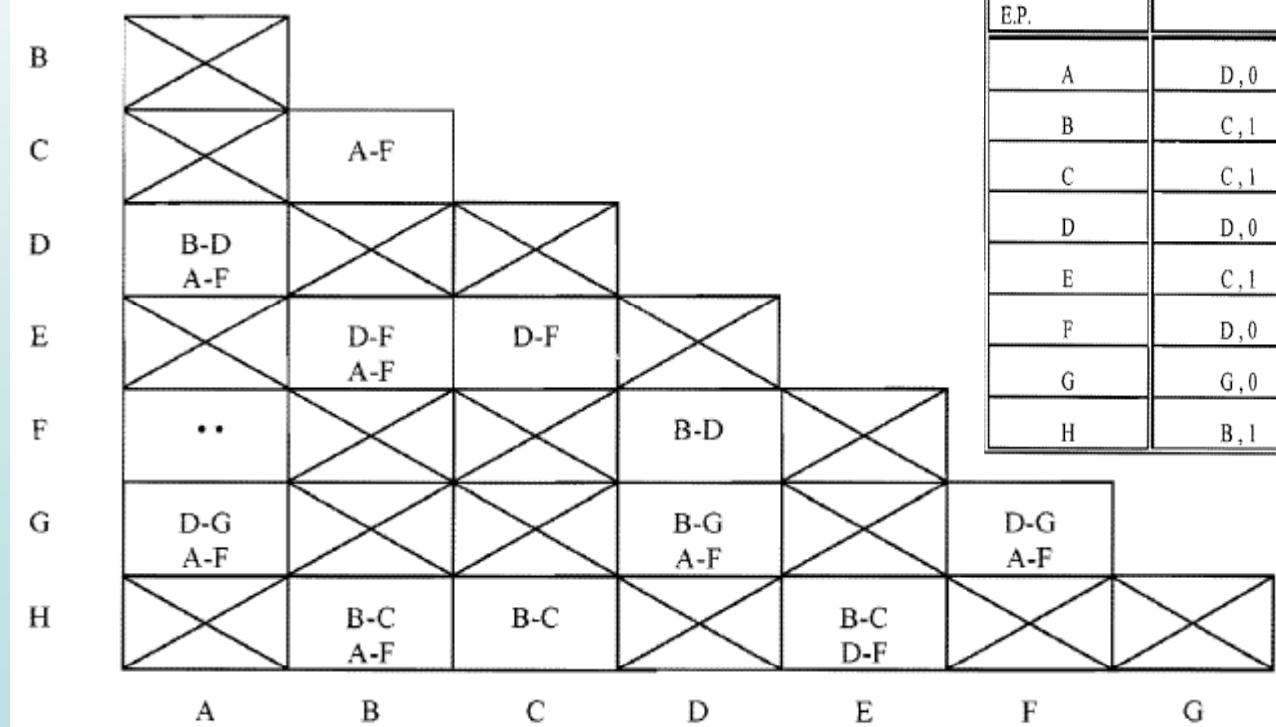
- Tabla de estados reducida:

E.P.	X ₁ X ₀	00	01	10	11
A	D , 0	D , 0	F , 0	A , 0	
B	C , 1	D , 0	E , 1	F , 0	
C	C , 1	D , 0	E , 1	A , 0	
D	D , 0	B , 0	A , 0	F , 0	
E	C , 1	F , 0	E , 1	A , 0	
F	D , 0	D , 0	A , 0	F , 0	
G	G , 0	G , 0	A , 0	A , 0	
H	B , 1	D , 0	E , 1	A , 0	

6.4 DISEÑO DE UN SISTEMA SECUENCIAL. Ejemplos de minimización de tablas de estados

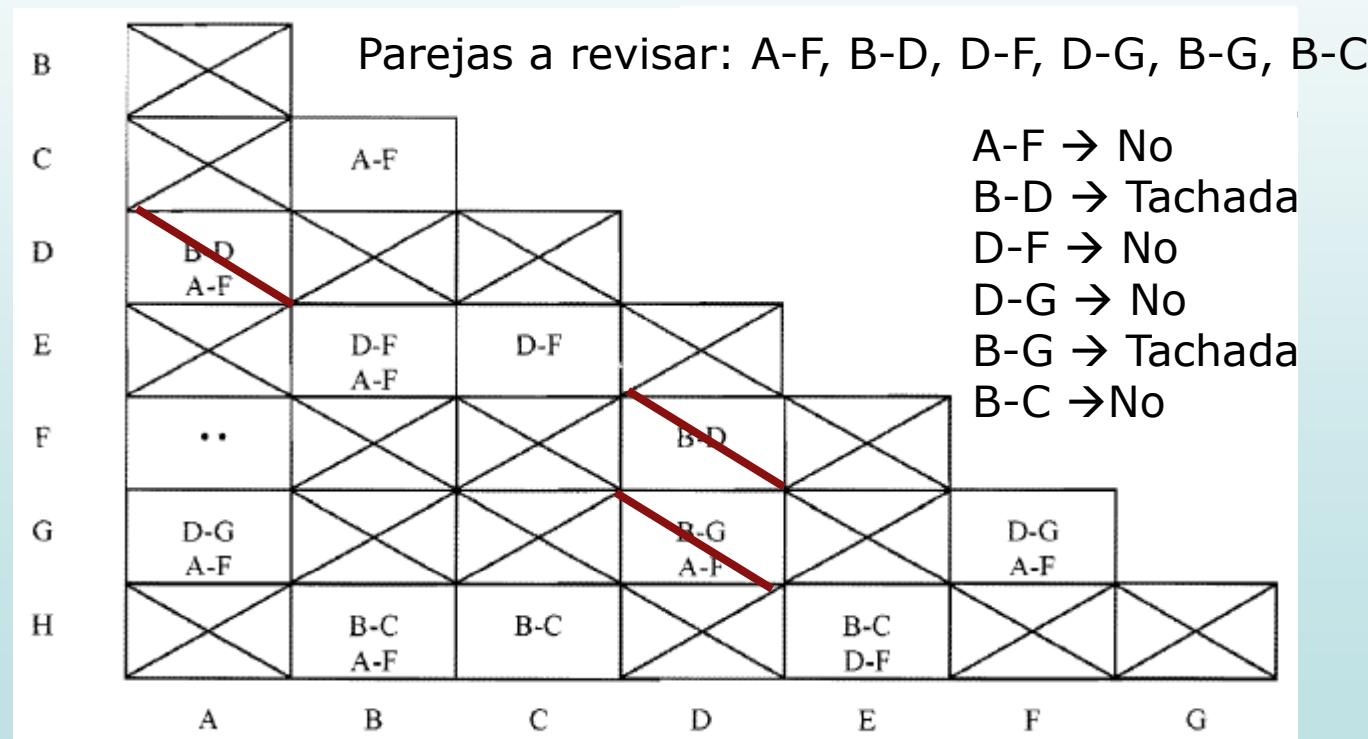
Tabla de implicaciones:

- Tachar estados con salidas diferentes
- Celdas sin tachar: anotar pares de estados implicados



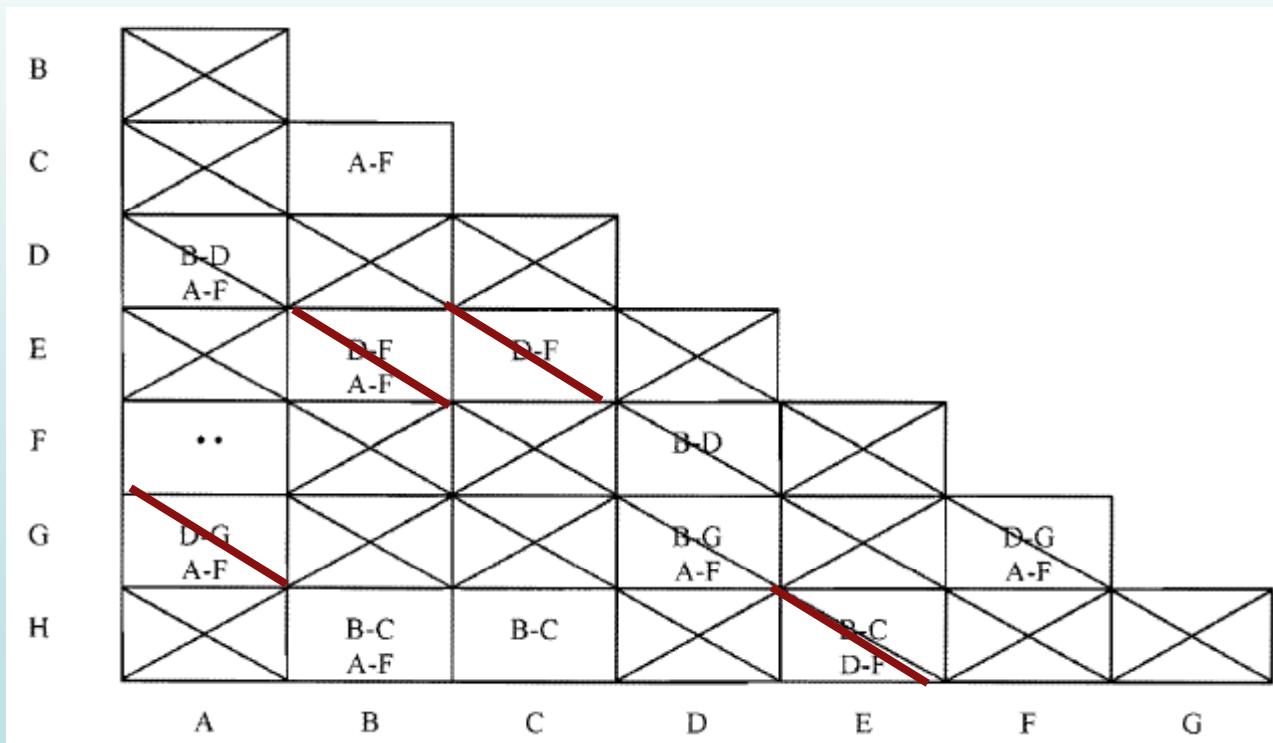
6.4 DISEÑO DE UN SISTEMA SECUENCIAL. Ejemplos de minimización de tablas de estados

- **Revisión sistemática** de la tabla, de modo que si la celda correspondiente a la pareja de estados, ha sido tachada, cualquier otra celda que contenga dicha pareja debe tacharse.



6.4 DISEÑO DE UN SISTEMA SECUENCIAL. Ejemplos de minimización de tablas de estados

- Ahora están tachadas A-D, D-F y D-G. Volvemos a revisar por si alguna contiene esas parejas.



Ahora están
tachadas:
A-G, B-E, C-E, E-H

Se vuelve a revisar
por si podemos
tachar alguna celda
más.

6.4 DISEÑO DE UN SISTEMA SECUENCIAL. Ejemplos de minimización de tablas de estados

- **Conjunto de estados equivalentes:**

Han quedado sin tachar las celdas: AF, BC, BH y CH

Eso quiere decir que son equivalentes los estados:

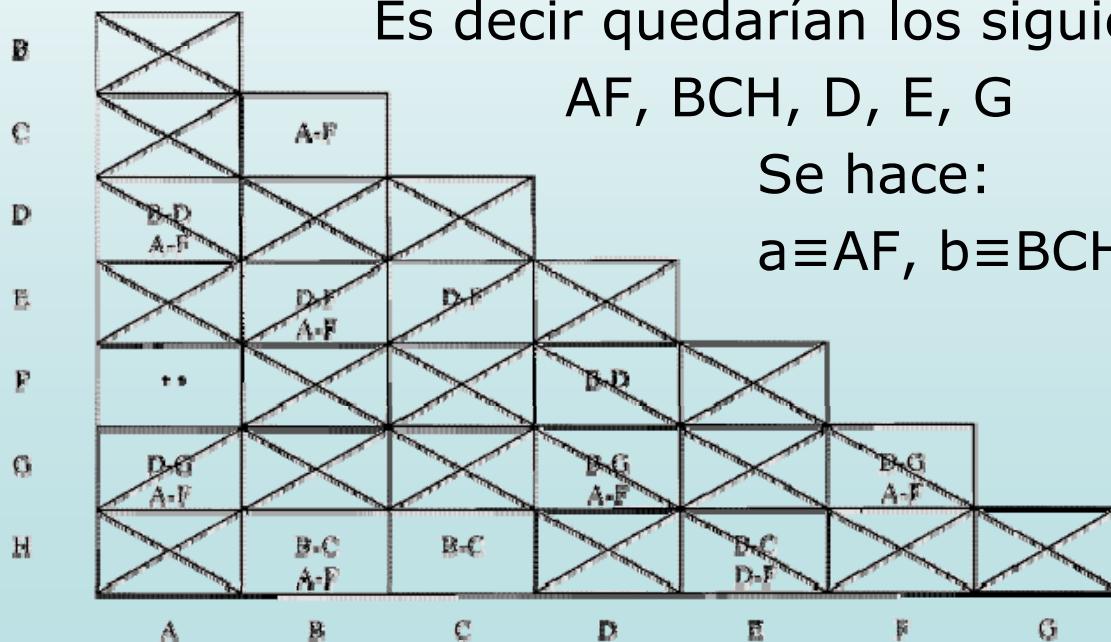
A-F y B-C-H

Es decir quedarían los siguientes estados:

AF, BCH, D, E, G

Se hace:

$a \equiv AF$, $b \equiv BCH$, $c \equiv D$, $d \equiv E$, $e \equiv G$



6.4 DISEÑO DE UN SISTEMA SECUENCIAL. Ejemplos de minimización de tablas de estados

- Tabla de estados mínima:

E.P.	X_1X_0	00	01	10	11
a	c , 0	c , 0	a , 0	a , 0	
b	b , 1	c , 0	d , 1	a , 0	
c	c , 0	b , 0	a , 0	a , 0	
d	b , 1	a , 0	d , 1	a , 0	
e	e , 0	e , 0	a , 0	a , 0	

TEMA 6. ANÁLISIS Y DISEÑO DE SISTEMAS SECuenciales.

CONTENIDOS:

- 6.1. Concepto de sistema secuencial.
- 6.2. Elementos básicos de memoria.
- 6.3. Análisis de un sistema secuencial.
- 6.4. Diseño de un sistema secuencial.
- 6.5. Componentes secuenciales estándar.**

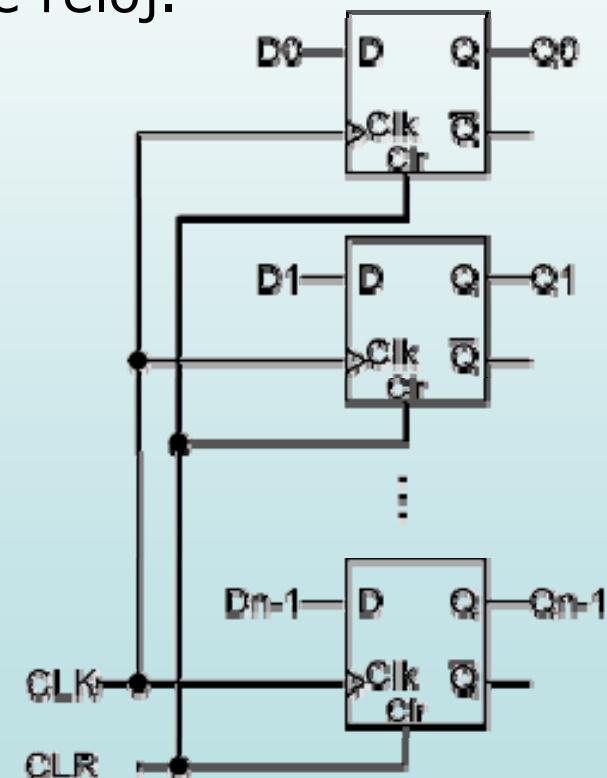
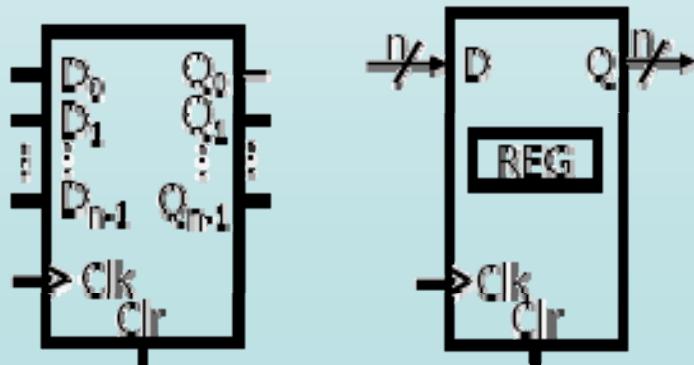


6.5 COMPONENTES SECUENCIALES ESTÁNDAR. REGISTROS.

- Un **registro básico de n bits** es una asociación de n flip-flops tipo D (FF-D) en paralelo, todos ellos compartiendo la misma señal de reloj.

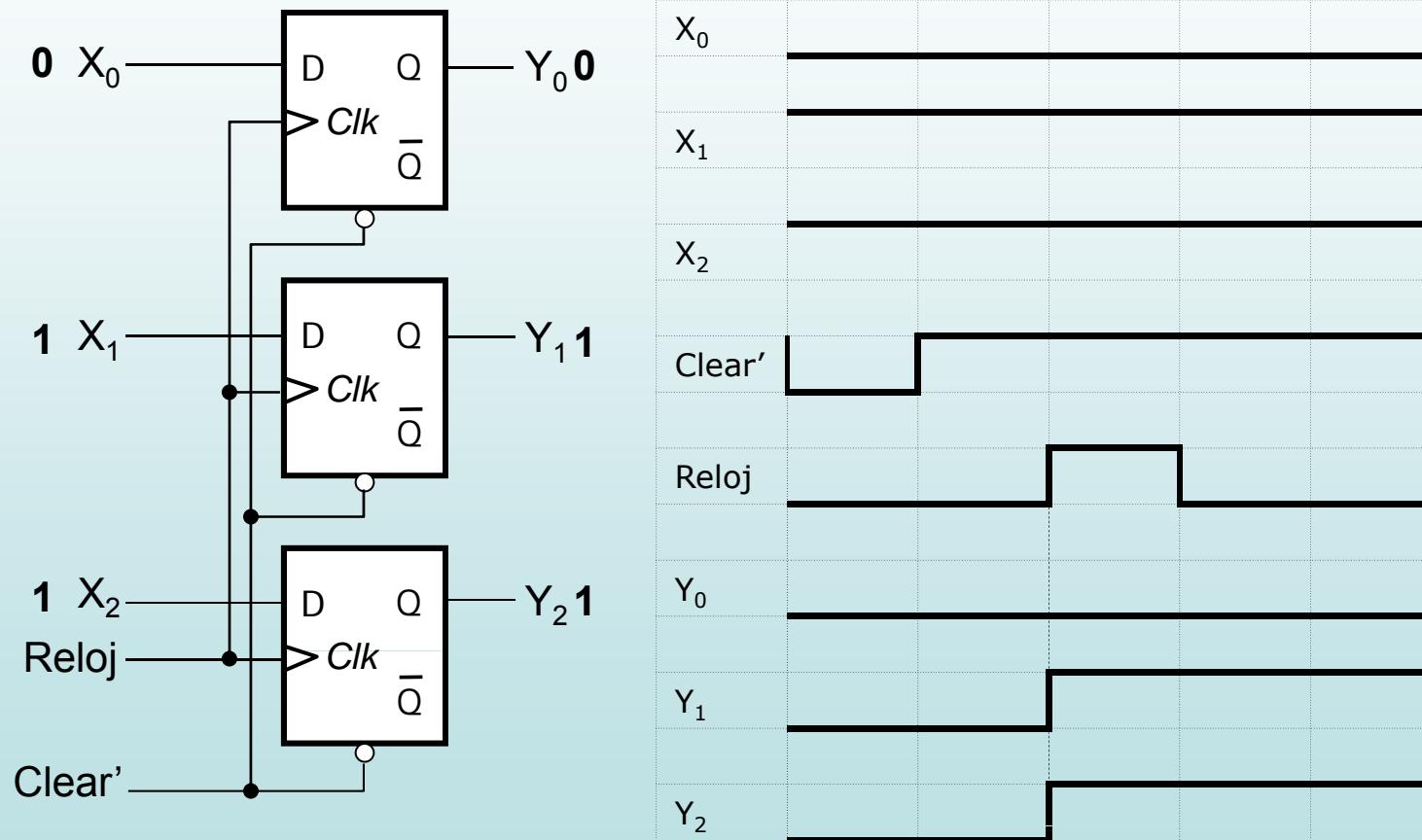
Cuando CLR=0 y en el flanco de subida de la señal de reloj el valor de D_i aparecerá en Q_i .

Posibles símbolos:



6.5 COMPONENTES SECUENCIALES ESTÁNDAR. REGISTROS.

- Registro básico con carga paralela de datos de 3 bits.



6.5 COMPONENTES SECUENCIALES ESTÁNDAR. REGISTROS.

- Los registros pueden tener una **señal de habilitación o carga** para decidir cuando se cargan los datos. Es un **registro con carga en paralelo con señal de habilitación**.

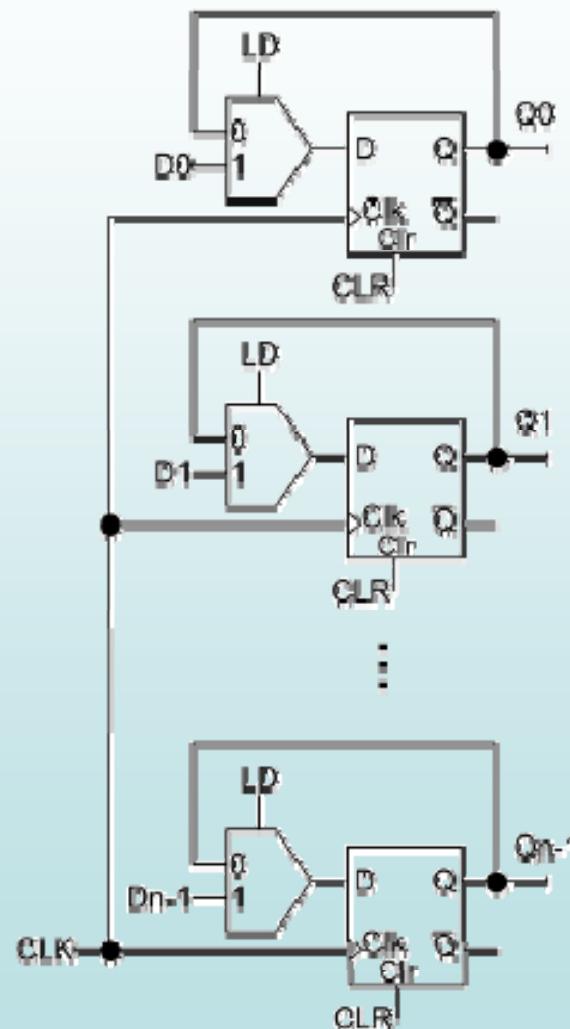
- Cuando $LD=0$, mantiene el valor que tuviera anteriormente:

$$Q_i^+ = Q_i$$

- Cuando $LD=1$, a la salida se carga lo que haya a la entrada, tras el pulso de reloj:

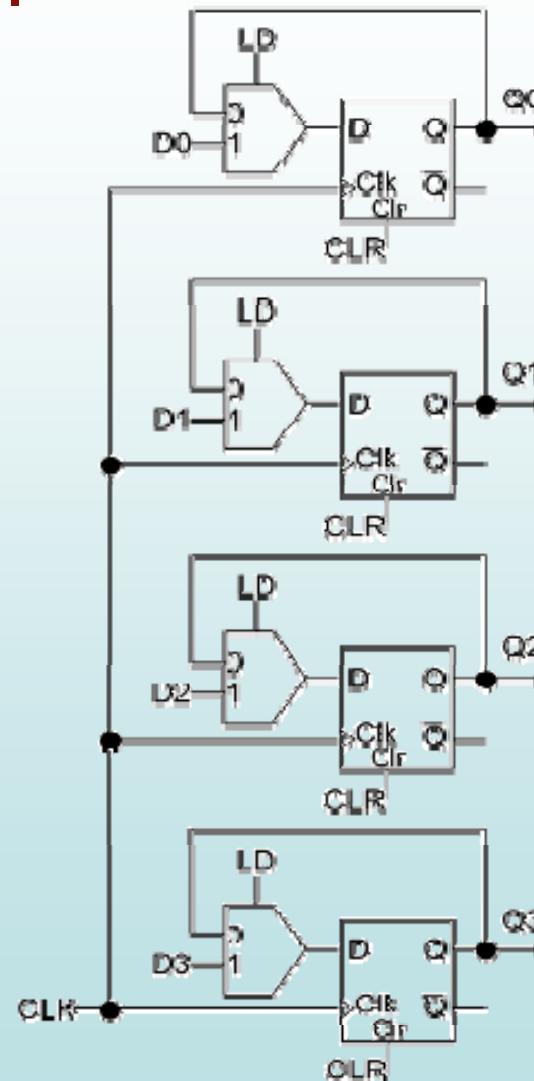
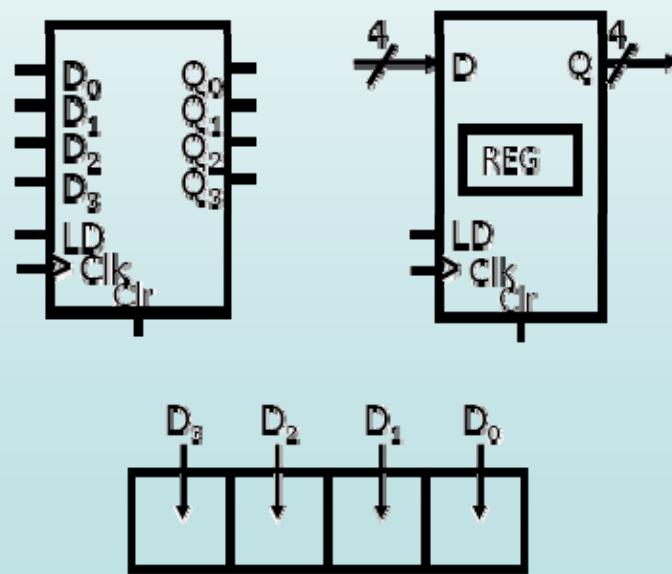
$$Q_i^+ = D_i$$

Por tanto, con LD se controla cuando se cargan los datos de la entrada



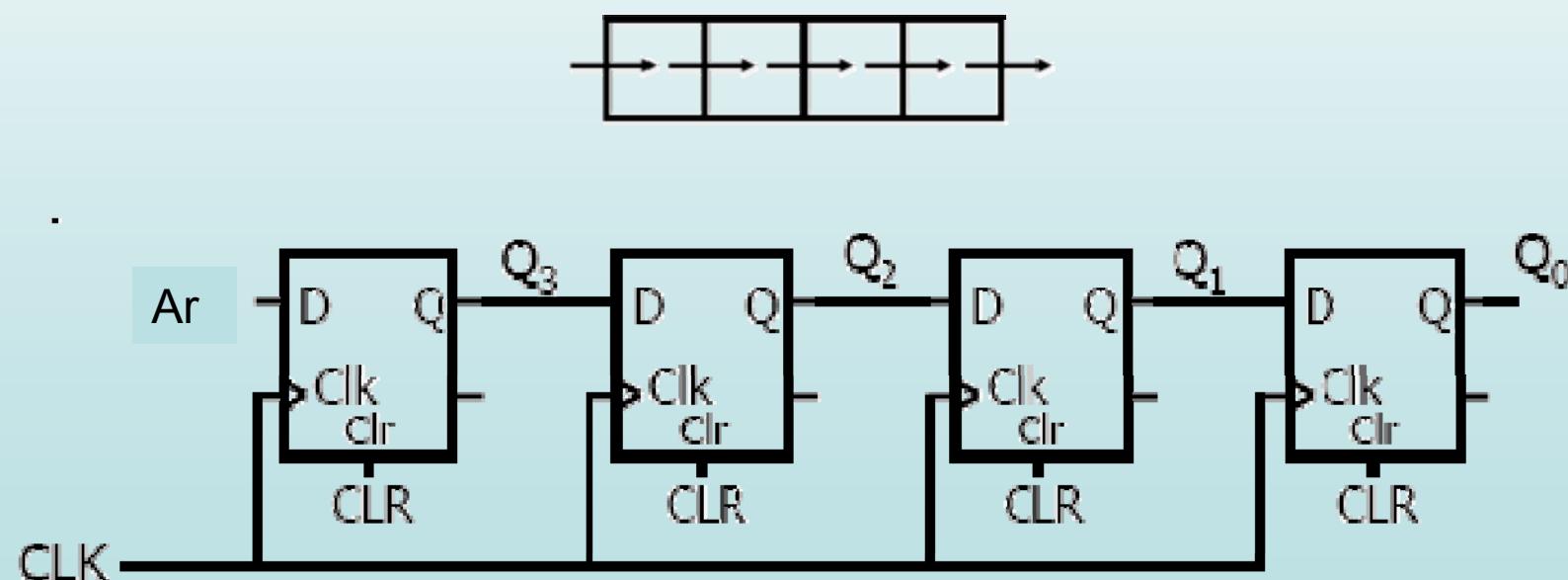
6.5 COMPONENTES SECUENCIALES ESTÁNDAR. REGISTROS.

- Ejemplo: Registro de 4 bits con señal de carga síncrona en paralelo.

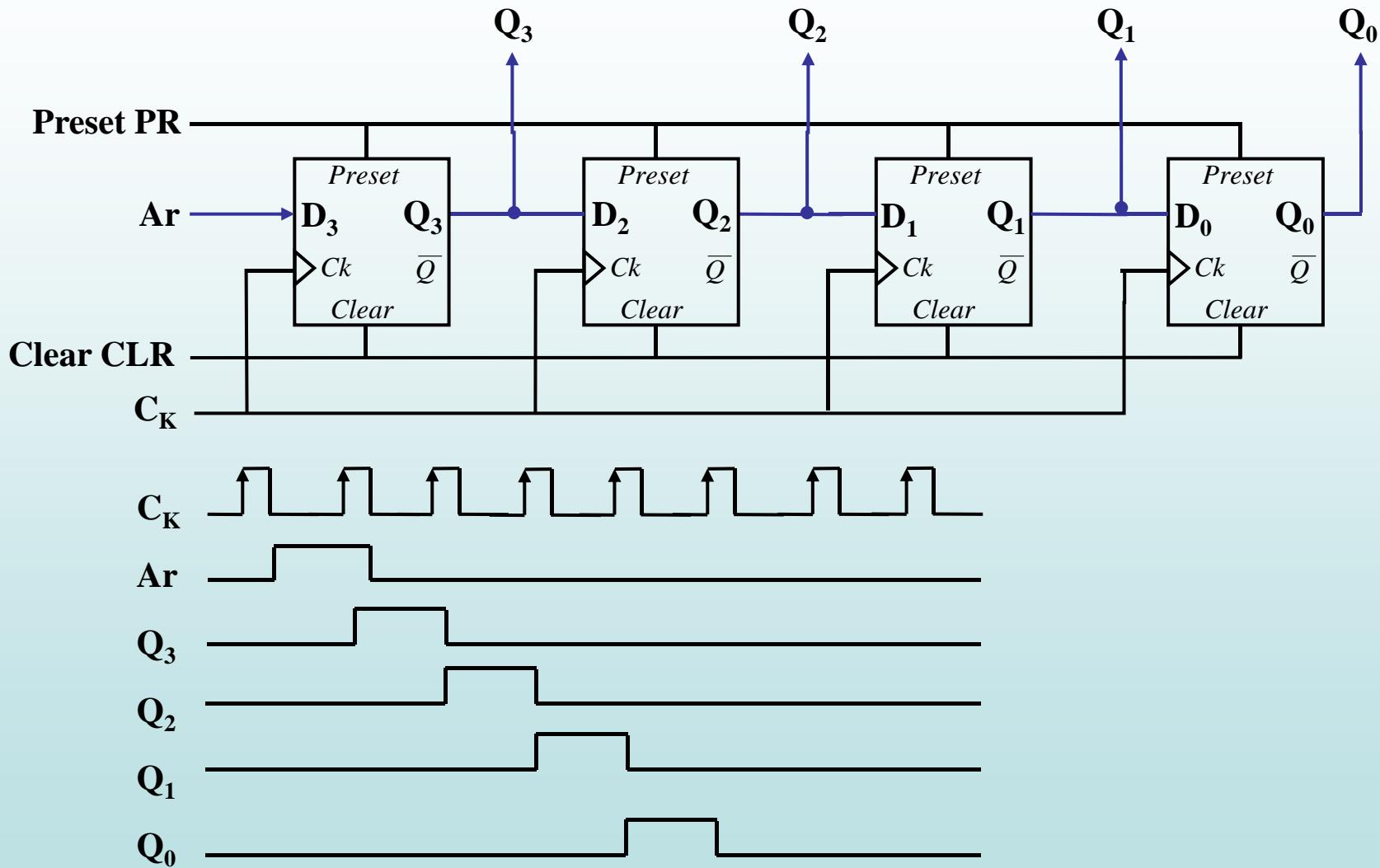


6.5 COMPONENTES SECUENCIALES ESTÁNDAR REGISTROS.

- Un **registro de desplazamiento básico de n bits (Shift Register - SHR)** es una asociación de n biestables tipo D (FF-D) en serie, compartiendo la misma señal de reloj.
- **Ejemplo:** SHR-4 bits básico.



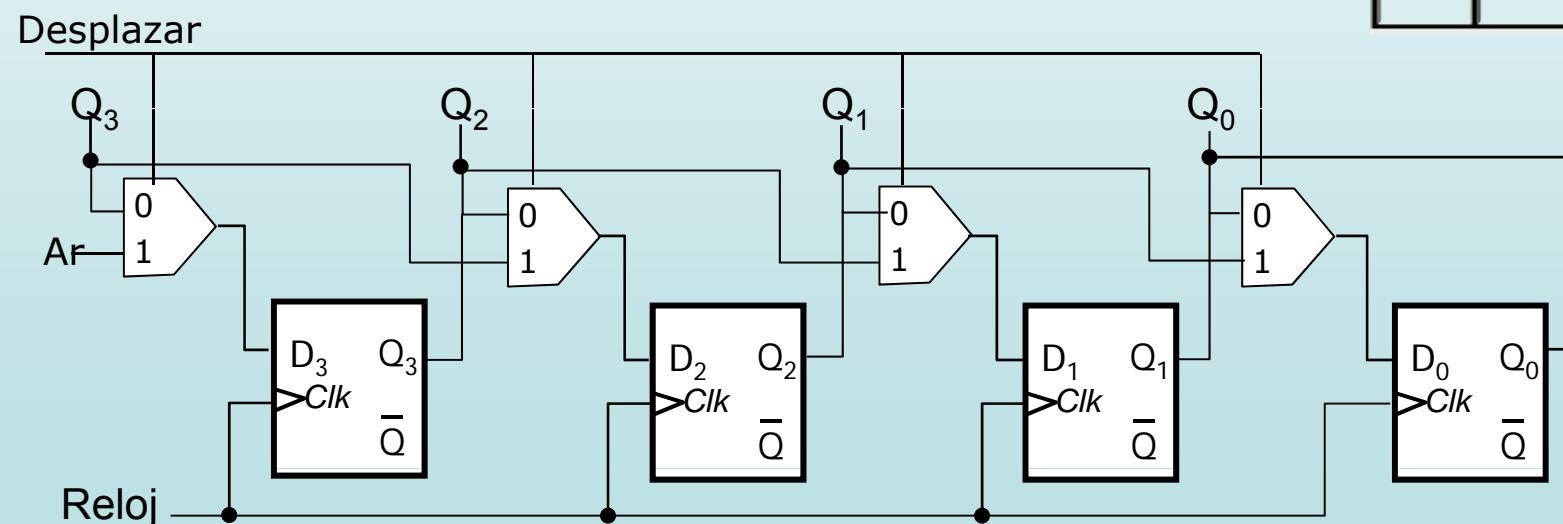
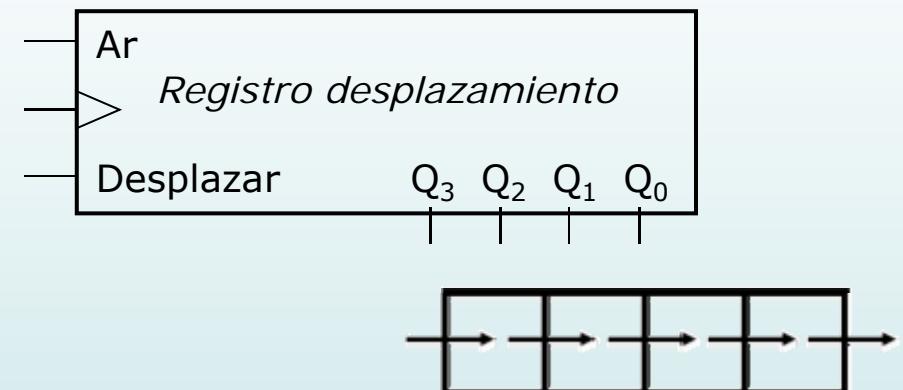
6.5 COMPONENTES SECUENCIALES ESTÁNDAR. REGISTROS.



6.5 COMPONENTES SECUENCIALES ESTÁNDAR. REGISTROS.

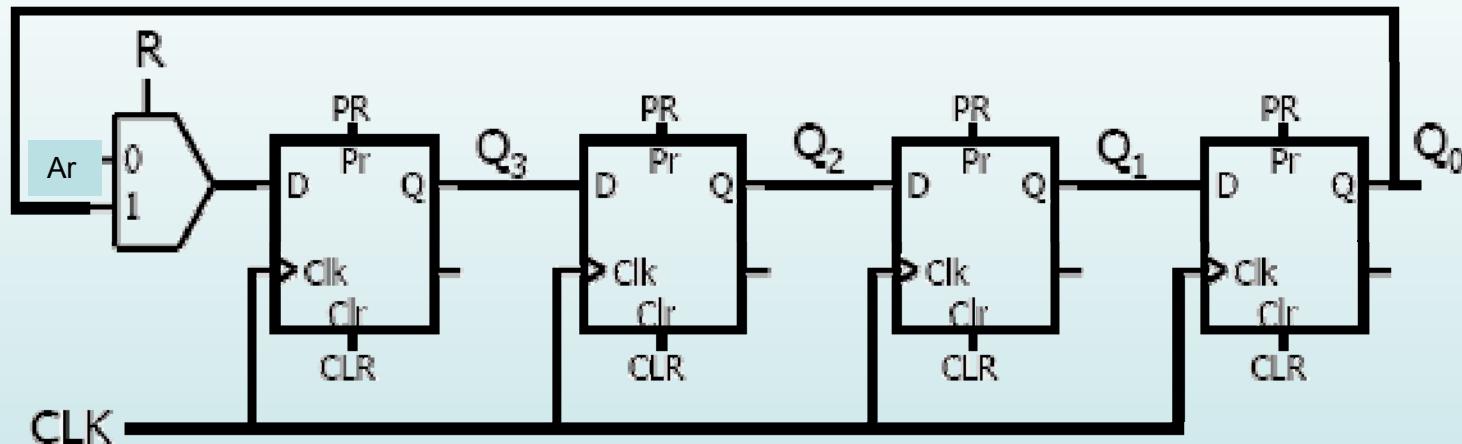
- **Registro de desplazamiento** a la derecha con entrada serie y salida paralelo de 4 bits.

EP	ES			
Desplazar	Q_3	Q_2	Q_1	Q_0
0	No cambia			
1	Ar	Q_3	Q_2	Q_1

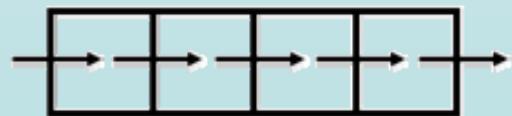


6.5 COMPONENTES SECUENCIALES ESTÁNDAR. REGISTROS.

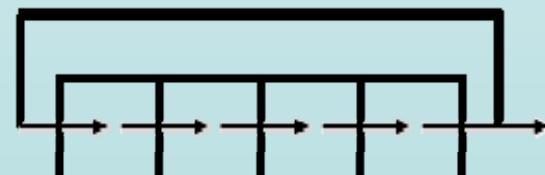
- Registro Desplazador/Rotador de 4 bits con carga serie:



Si R=0, Desplaza a la derecha



Si R=1, Rota a la derecha

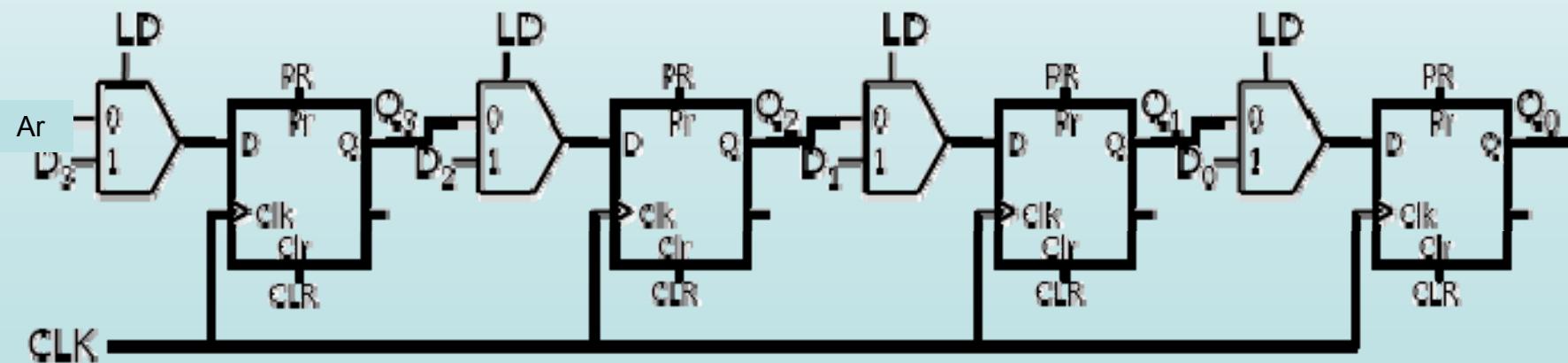
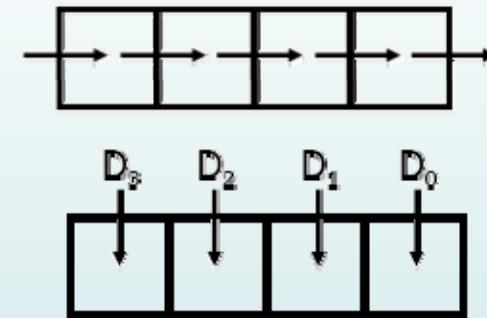


6.5 COMPONENTES SECUENCIALES ESTÁNDAR. REGISTROS.

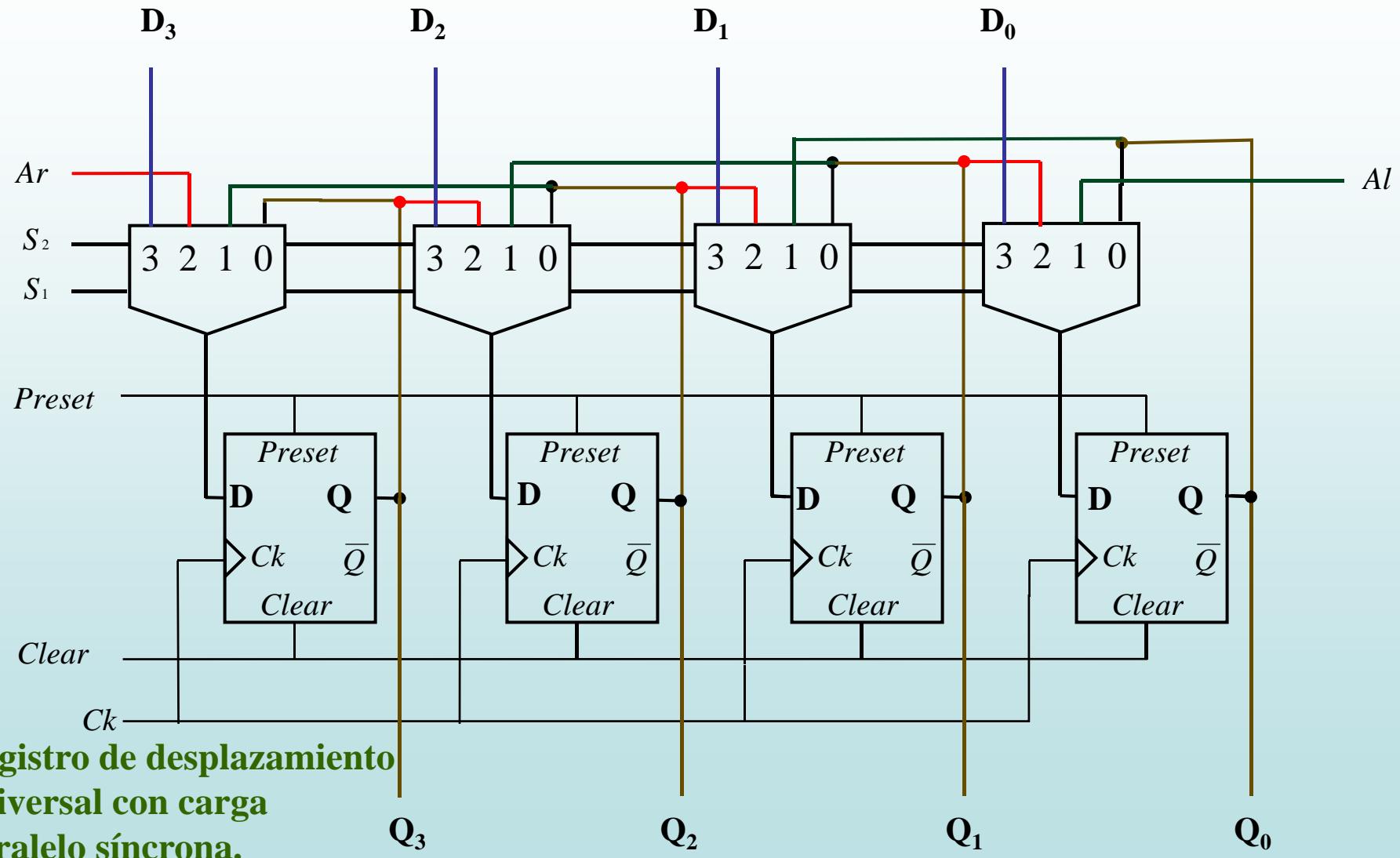
- Registro de desplazamiento de 4 bits con carga paralela:

EP	ES			
LD	Q ₃	Q ₂	Q ₁	Q ₀
0	Ar	Q ₃	Q ₂	Q ₁
1	D ₃	D ₂	D ₁	D ₀

Desplaza
Carga

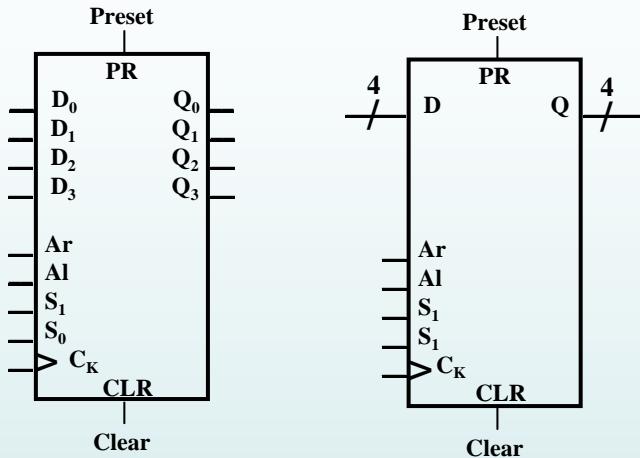


6.5 COMPONENTES SECUENCIALES ESTÁNDAR. REGISTROS.



6.5 COMPONENTES SECUENCIALES ESTÁNDAR. REGISTROS.

Símbolos

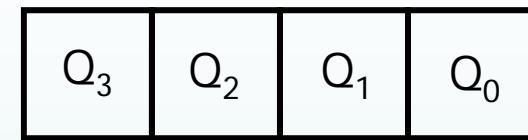


C_K	CLR	PR	$S_1 S_0$	Salidas (Q_i)
-	1	0	-	$Q_i = 0$
-	0	1	-	$Q_i = 1$
\uparrow	0	0	0 0	Hold (no cambia)
\uparrow	0	0	0 1	Desplaza a izquierda
\uparrow	0	0	1 0	Desplaza a derecha
\uparrow	0	0	1 1	$Q_i^+ = D_i$ Carga Paralelo

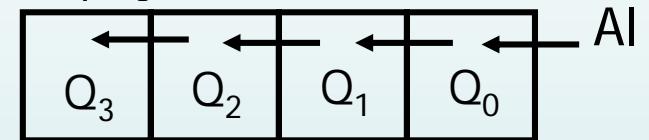
\uparrow : Flanco de subida

**Registro de desplazamiento universal
con señal de carga síncrona.**

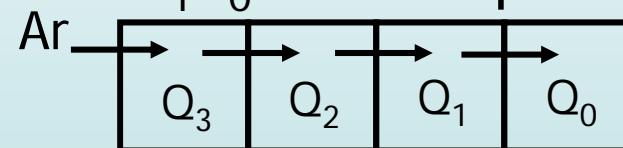
$S_1 S_0 = 00$ (Hold)



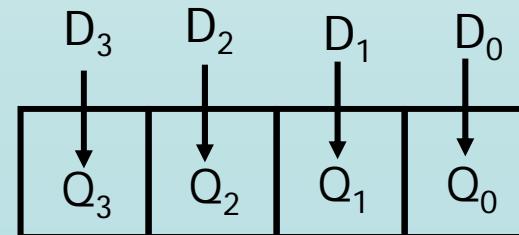
$S_1 S_0 = 01$ Desp. Izq



$S_1 S_0 = 10$ Desp. Der.



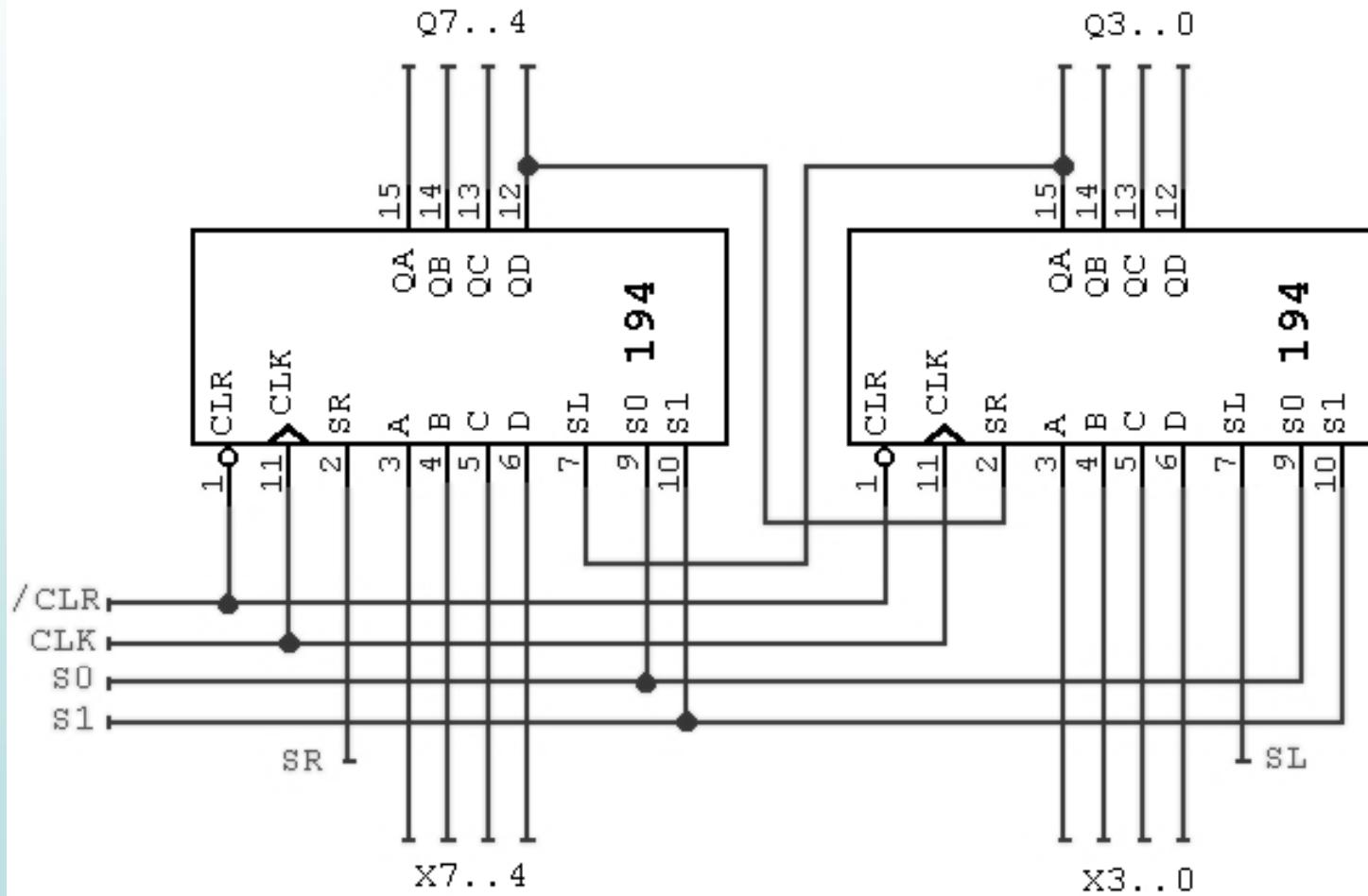
$S_1 S_0 = 11$: $Q_i^+ = D_i$



Ampliación de registros de desplazamiento

SL es entrada serie para desplazamientos a izquierda (Al)

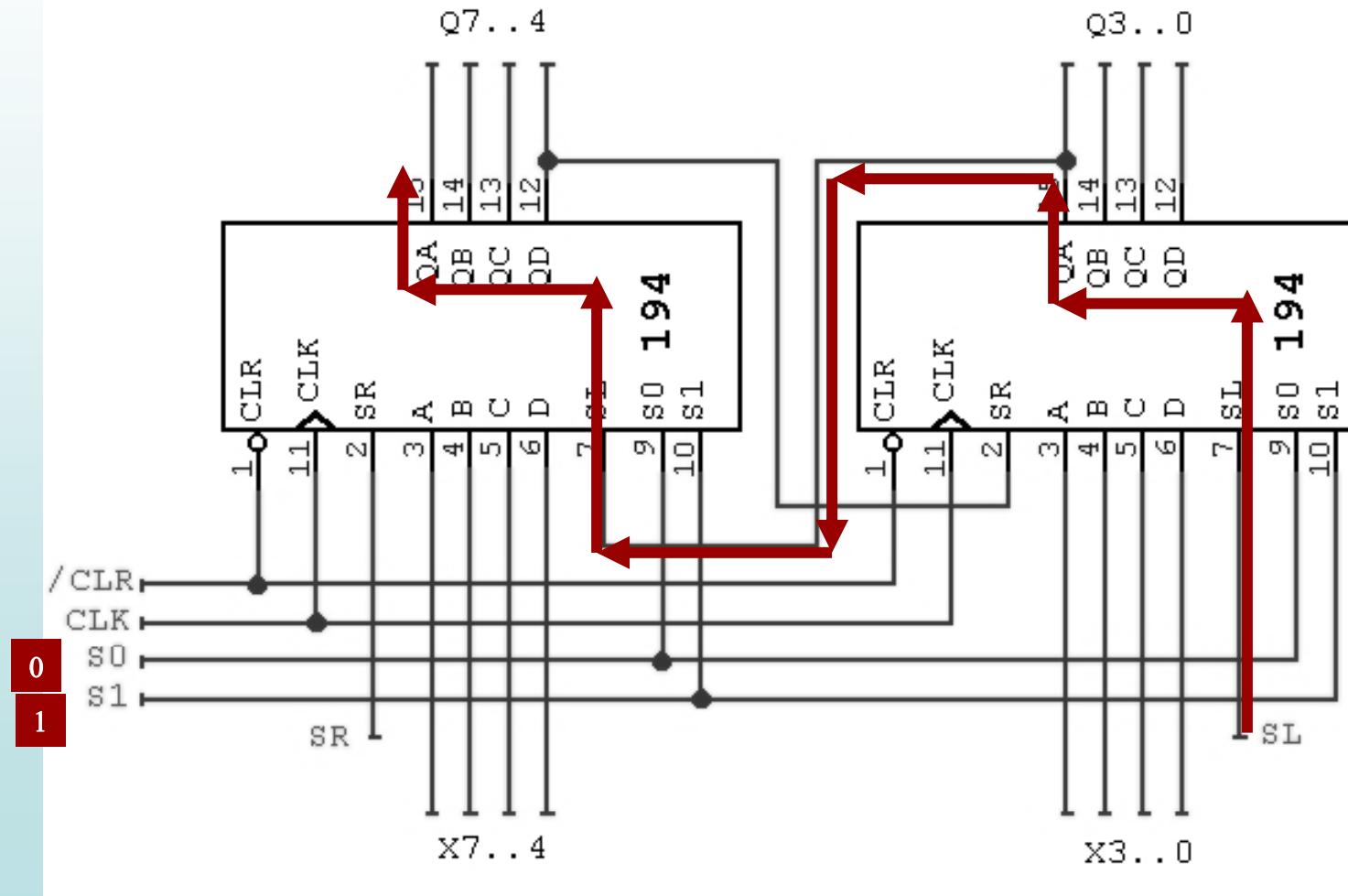
SR es entrada serie para desplazamientos a derecha (Ar)



Ampliación de registros de desplazamiento

SL es entrada serie para desplazamientos a izquierda (Al)

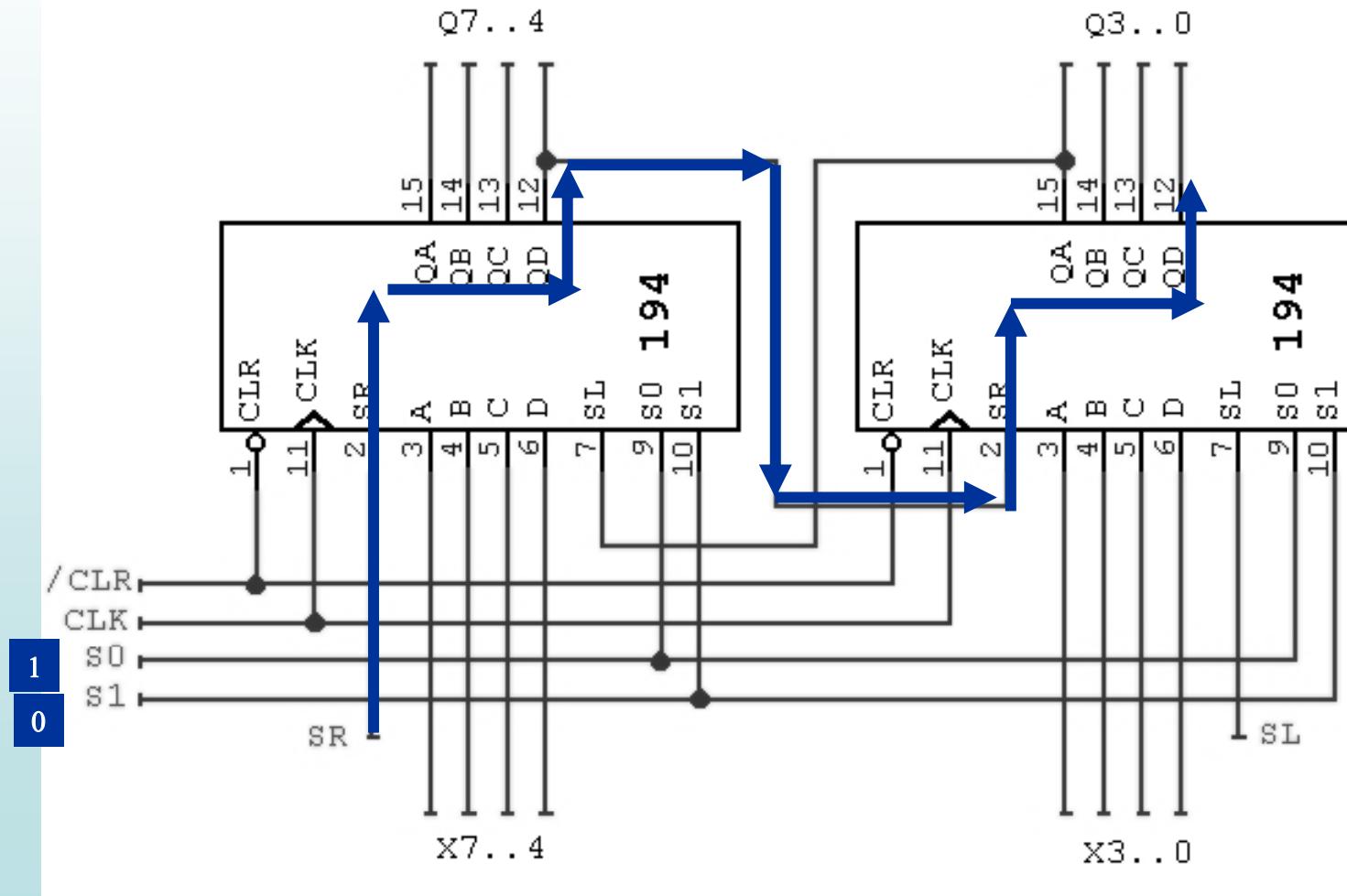
SR es entrada serie para desplazamientos a derecha (Ar)



Ampliación de registros de desplazamiento

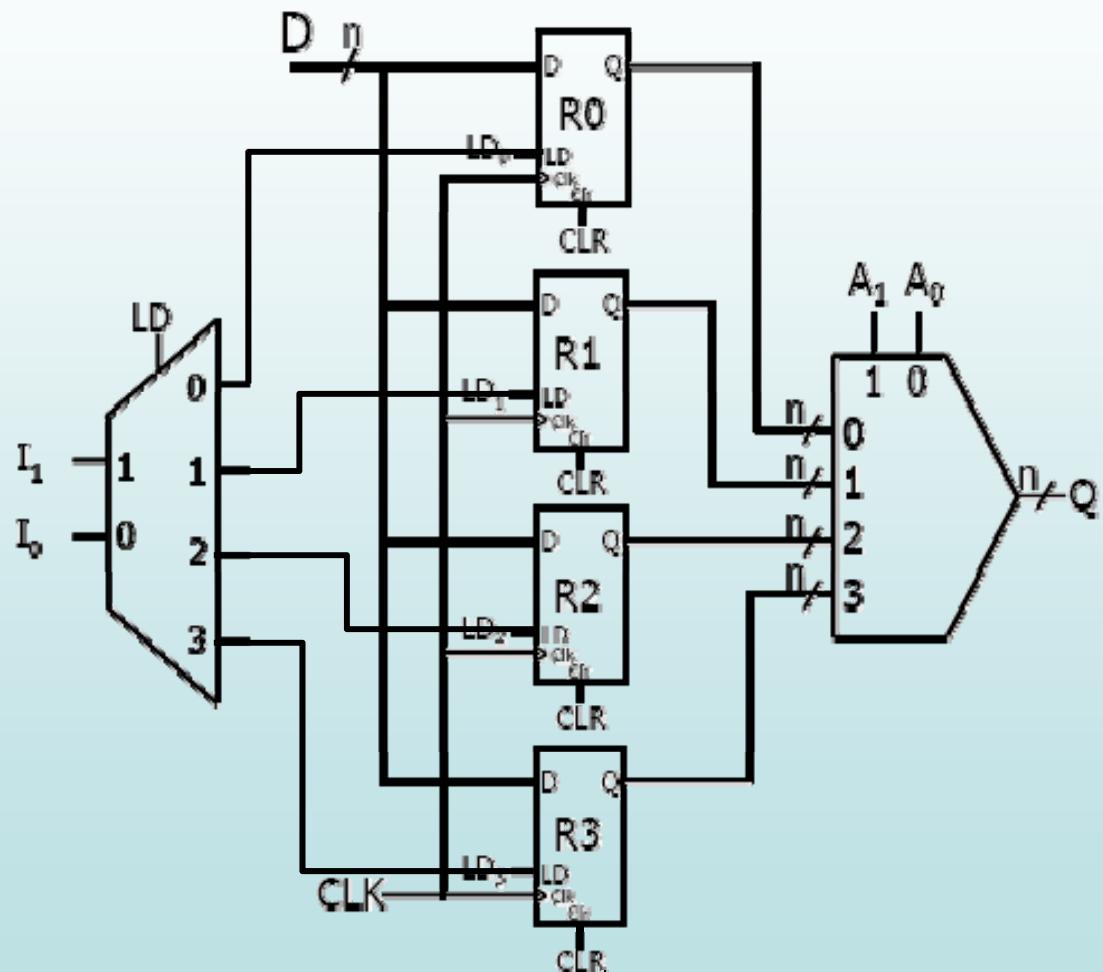
SL es entrada serie para desplazamientos a izquierda (Al)

SR es entrada serie para desplazamientos a derecha (Ar)

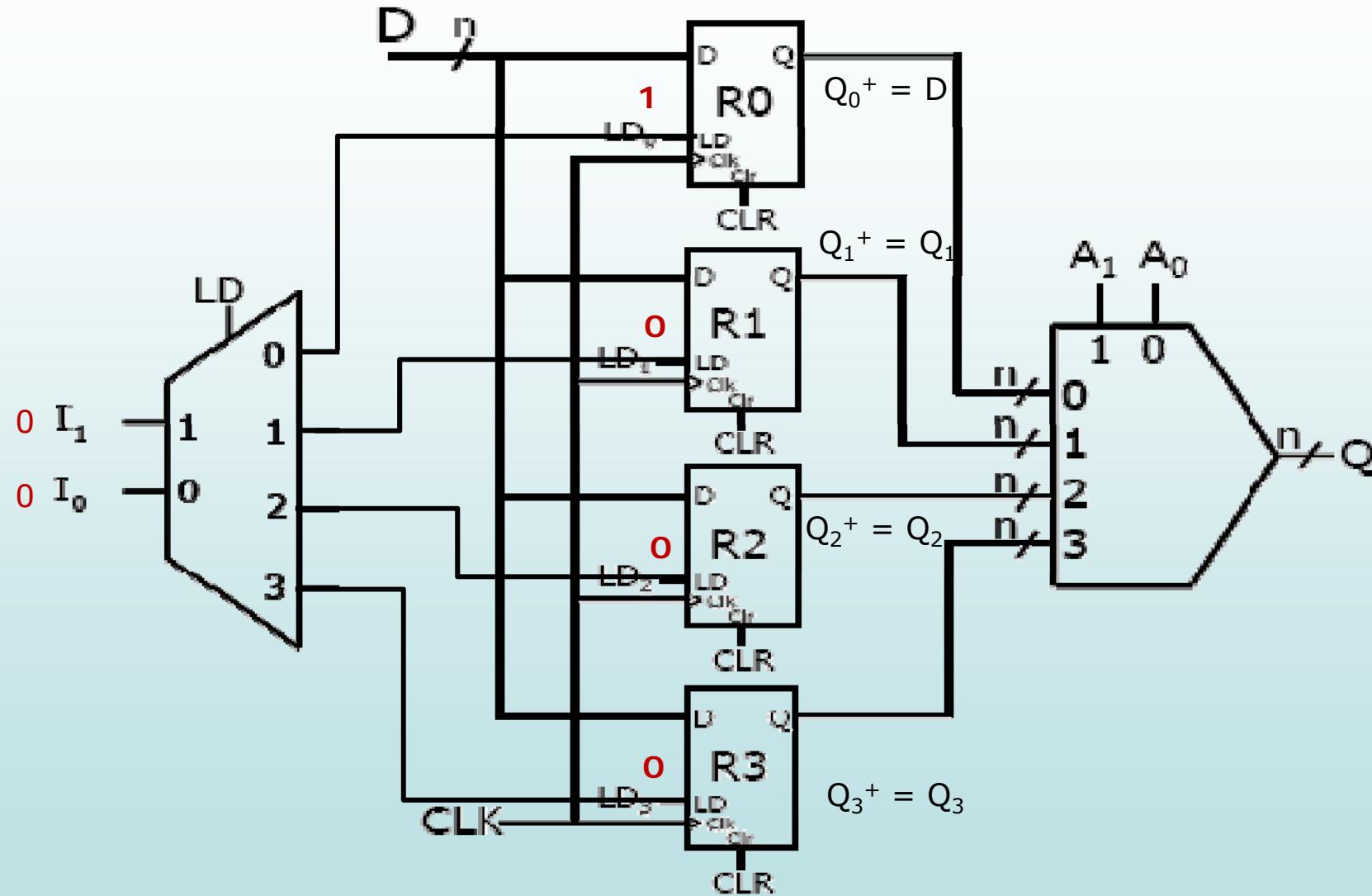


6.5 COMPONENTES SECUENCIALES ESTÁNDAR. REGISTROS.

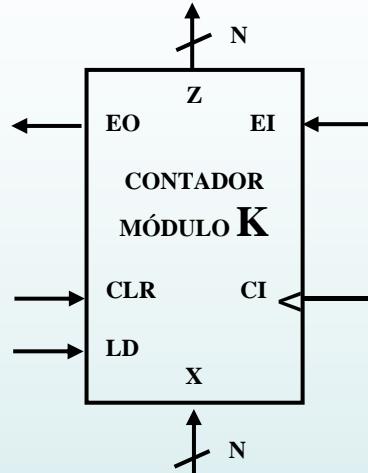
- Un **banco de registros** es un conjunto de registros que comparten, además de la misma señal de reloj, las mismas líneas de entrada y de salida.
- Banco de 4 Registros de n bits con un puerto de lectura y otro de escritura



6.5 COMPONENTES SECUENCIALES ESTÁNDAR. REGISTROS.



6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.



EI Entrada de habilitación de conteo
E0 Salida de acarreo
CLR Puesta a cero
LD Carga paralelo
CI Entrada de pulsos a contar
 $Z = (Z_{N-1} Z_{N-2} \dots Z_0)$
Número de la cuenta
 $X = (X_{N-1} X_{N-2} \dots X_0)$
Entrada de datos en carga paralelo

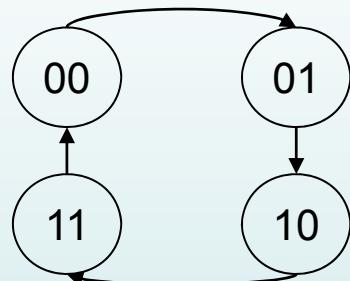
Contador genérico

- K** - Representa el número de estados por los que pasa el contador (módulo del contador)
- Si $K=2^N$ se dice que el contador es binario.
- Si la cuenta se incrementa cuando llega un pulso en **CI** el contador es ascendente y si la cuenta se decrementa se dice que es descendente
- En un contador decimal: **K=10** y **N=4**

6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.

- Contador síncrono ascendente módulo 4:

1. Diagrama de estados



2. Tabla de estados

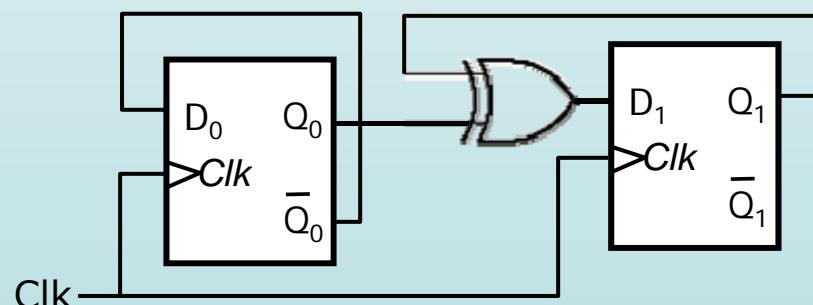
$Q_1 Q_0$	$Q_1^+ Q_0^+$
00	01
01	10
10	11
11	00

3. Tabla de excitación: $D = Q +$

$Q_1 Q_0$	$Q_1^+ Q_0^+ \equiv D_1 D_0$
00	01
01	10
10	11
11	00

$$D_1 = Q_1 \oplus Q_0$$
$$D_0 = Q_0'$$

4. Implementación:

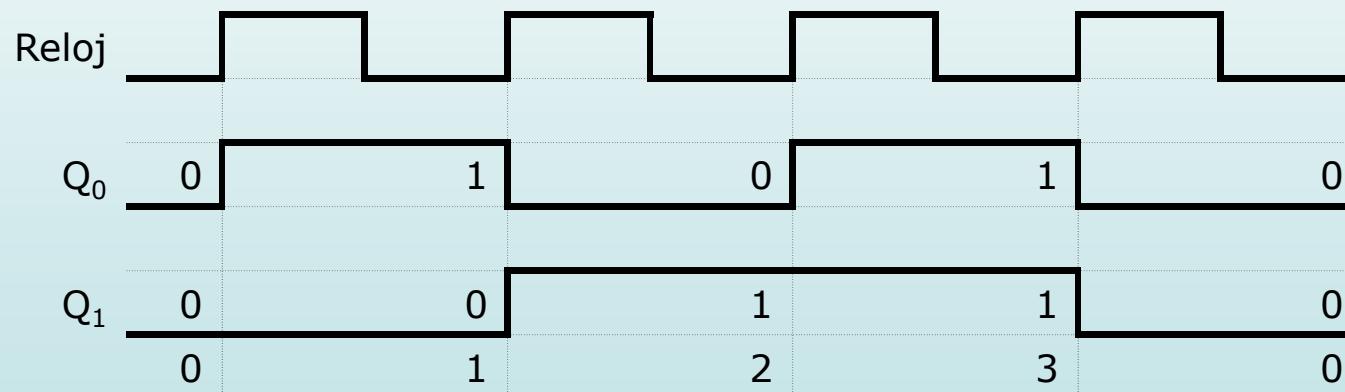


6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.

- Cronograma:

$$Q_1^+ = D_1 = Q_1 \oplus Q_0$$

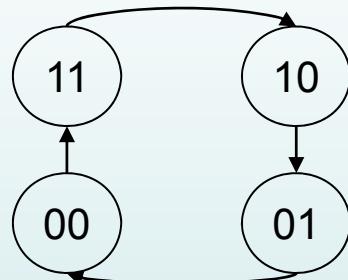
$$Q_0^+ = D_0 = Q_0'$$



6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.

- Contador síncrono descendente módulo 4:

1. Diagrama de estados



2. Diagrama de estados

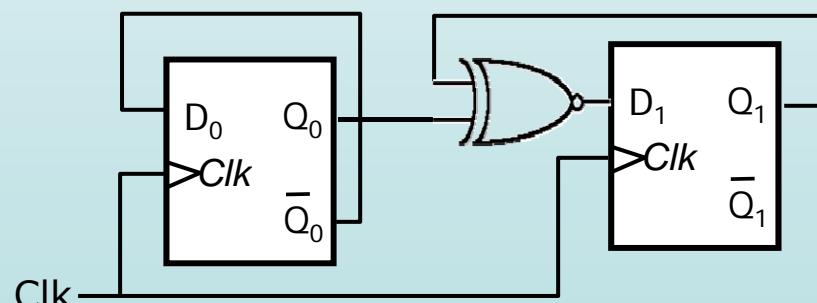
$Q_1 Q_0$	$Q_1^+ Q_0^+$
00	11
01	00
10	01
11	10

3. Tabla de excitación: $D = Q +$

$Q_1 Q_0$	$Q_1^+ Q_0^+ \equiv D_1 D_0$
00	11
01	00
10	01
11	10

$$D_1 = (Q_1 \oplus Q_0)'$$
$$D_0 = Q_0'$$

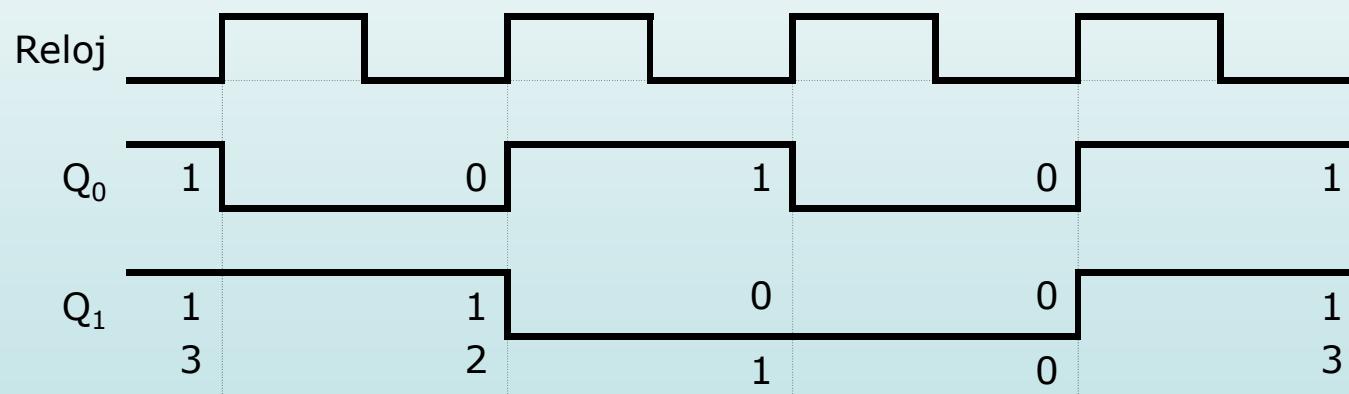
4. Implementación:



6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.

- Cronograma: el estado inicial es 11
(FF a SET)

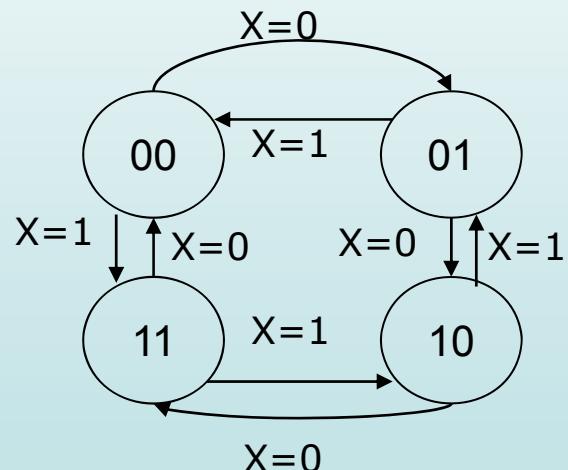
$$Q_1^+ = D_1 = (Q_1 \oplus Q_0)'$$
$$Q_0^+ = D_0 = Q_0'$$



6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.

- **Contador síncrono ascendente/descendente módulo 4:**
 $X=0$ cuenta ascendente, $X=1$ cuenta descendente

1. Diagrama de estados



2. Tabla de estados

X	Q_1	Q_0	Q_1^+	Q_0^+
0	00		01	
0	01		10	
0	10		11	
0	11		00	
1	00		11	
1	01		00	
1	10		01	
1	11		10	

6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.

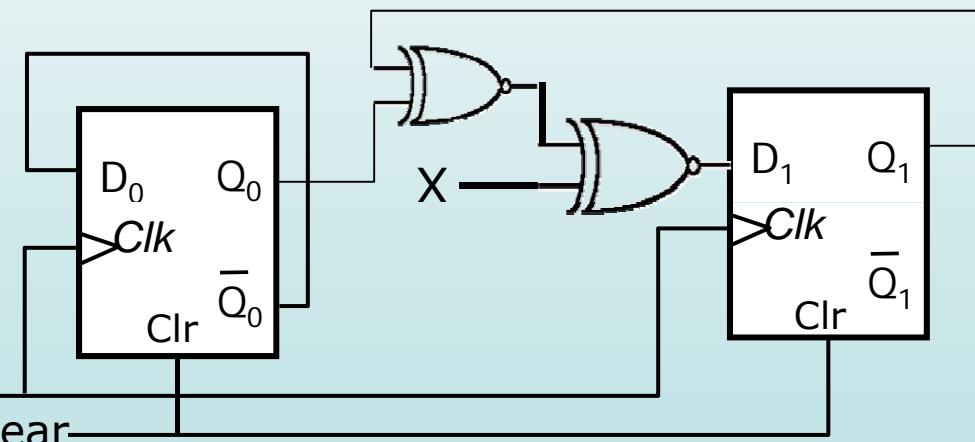
3. Tabla de excitación: $D = Q^+$

X	Q_1 Q_0	$Q_1^+ Q_0^+$	$D_1^+ D_0^+$
0	00	01	01
0	01	10	10
0	10	11	11
0	11	00	00
1	00	11	11
1	01	00	00
1	10	01	01
1	11	10	10

$$Q_1^+ = D_1 = X \oplus (Q_1 \oplus Q_0)$$

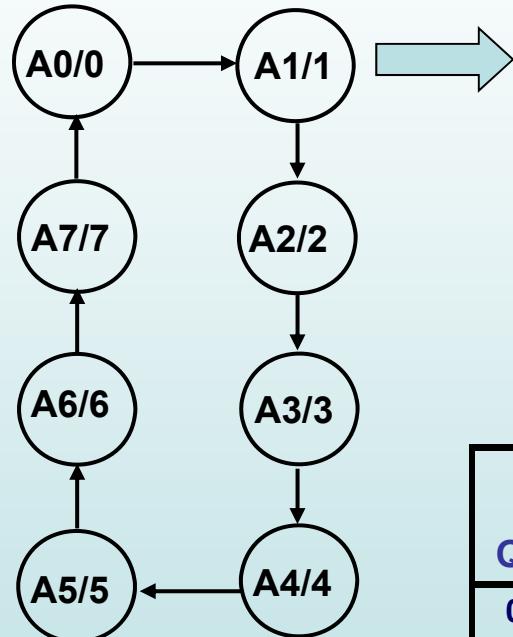
$$Q_0^+ = D_0 = Q_0'$$

4. Implementación



6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.

Diagrama de estados del contador binario



Contador binario
síncrono módulo 8
ascendente

Asignación de estados			
	Q_2	Q_1	Q_0
A0	0	0	0
A1	0	0	1
A2	0	1	0
A3	0	1	1
A4	1	0	0
A5	1	0	1
A6	1	1	0
A7	1	1	1

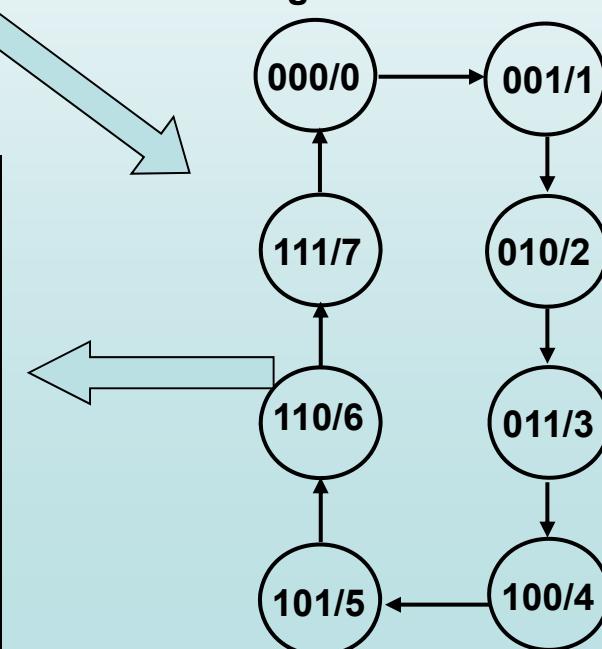
Tabla de estados

Estado actual Q_2 Q_1 Q_0	Dec	Estado Siguiente Q_2^+ Q_1^+ Q_0^+		
		Q_2^+	Q_1^+	Q_0^+
0 0 0	0	0	0	1
0 0 1	1	0	1	0
0 1 0	2	0	1	1
0 1 1	3	1	0	0
1 0 0	4	1	0	1
1 0 1	5	1	1	0
1 1 0	6	1	1	1
1 1 1	7	0	0	0

En este caso los estados se han asignado de modo que coincidan con las salidas.

NOTA. Esto es un caso particular en los contadores, en general en un sistema secuencial dicha coincidencia NO es posible.

Diagrama de estados asignado



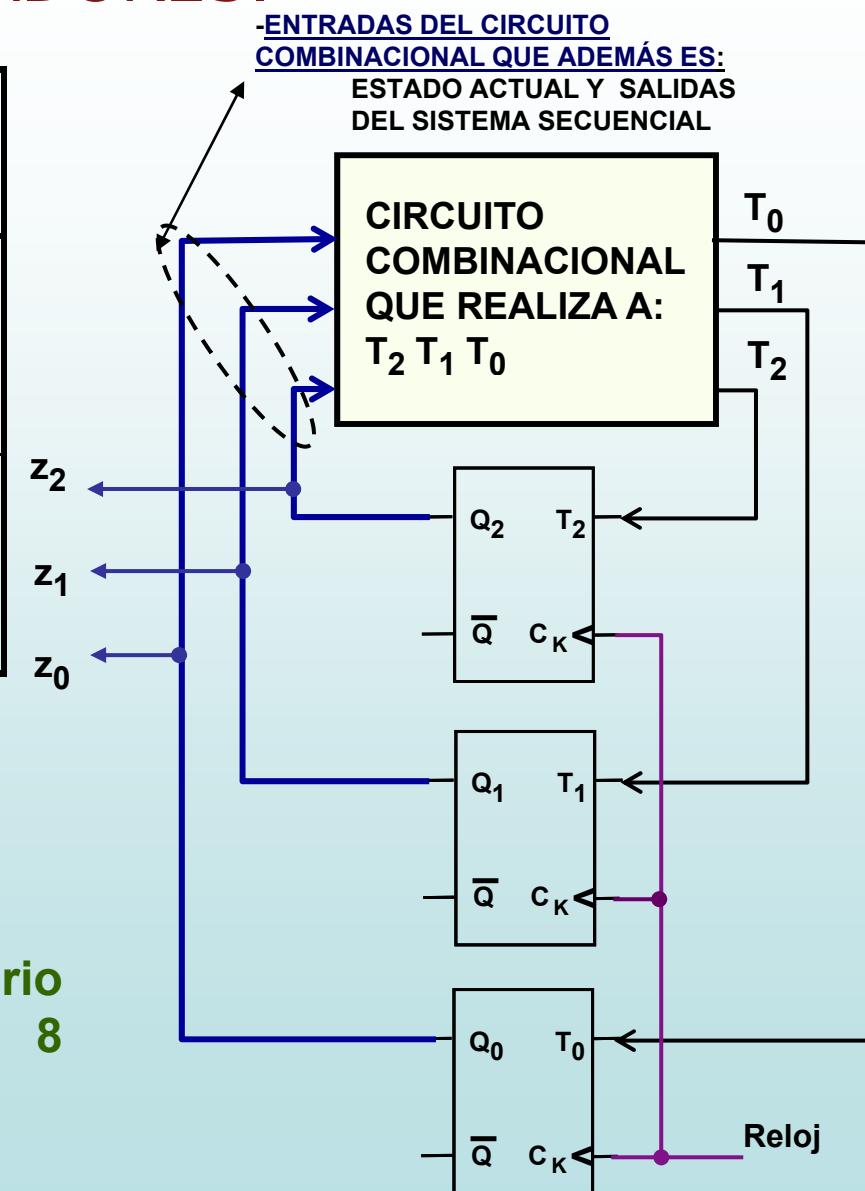
6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.

Tabla de transiciones

Estado actual $Q_2 \ Q_1 \ Q_0$	Dec	Estado Siguiente $Q_2^+ \ Q_1^+ \ Q_0^+$	Funciones a realizar $T_2 \ T_1 \ T_0$
0 0 0	0	0 0 1	
0 0 1	1	0 1 0	
0 1 0	2	0 1 1	
0 1 1	3	1 0 0	
1 0 0	4	1 0 1	
1 0 1	5	1 1 0	
1 1 0	6	1 1 1	
1 1 1	7	0 0 0	

$Q \ Q^+$	T
0 0	0
0 1	1
1 0	1
1 1	0

Contador binario síncrono módulo 8 ascendente



6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.

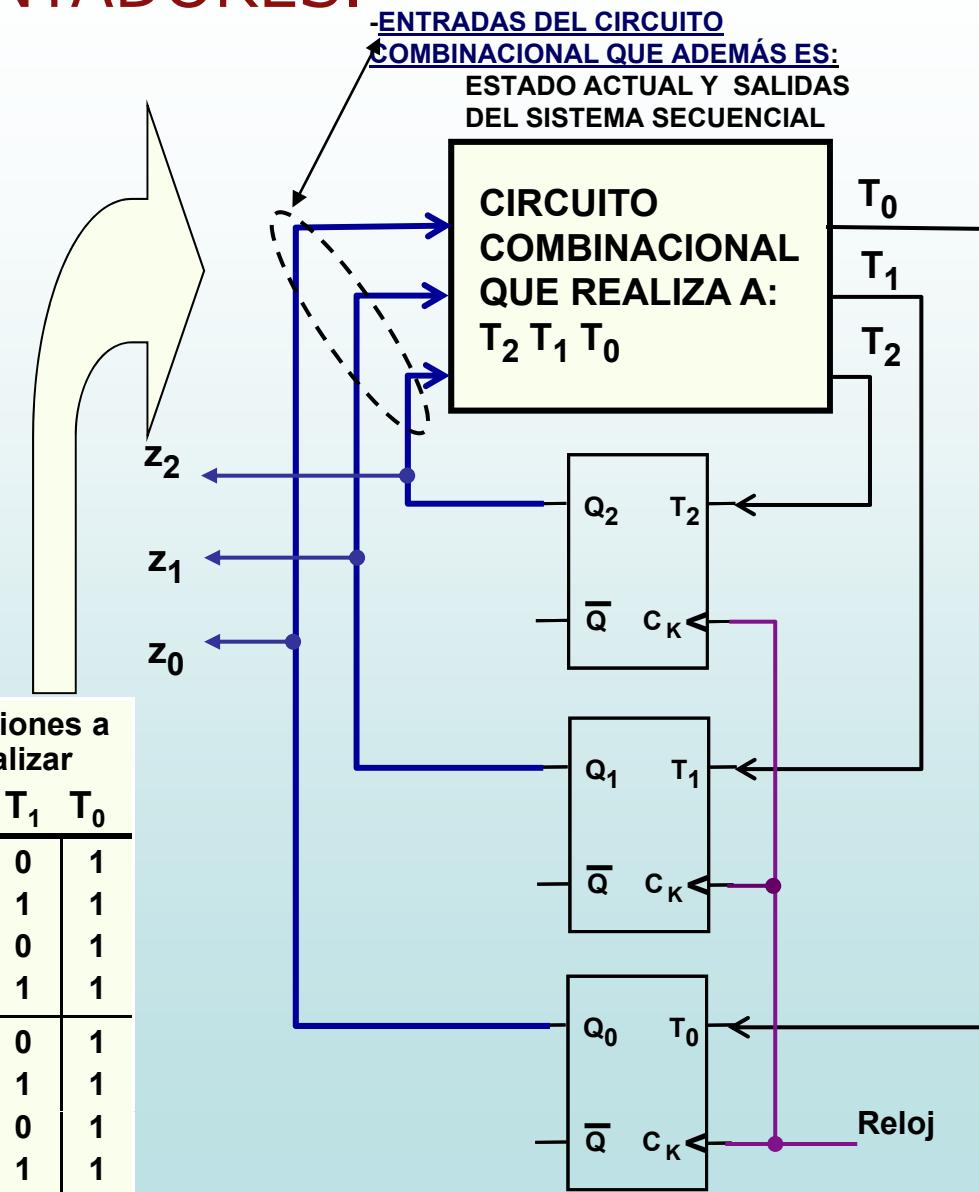
Tabla de transiciones

Estado actual $Q_2 Q_1 Q_0$	Dec	Estado Siguiente $Q_2^+ Q_1^+ Q_0^+$			Funciones a realizar		
		Q_2^+	Q_1^+	Q_0^+	T_2	T_1	T_0
0 0 0	0	0	0	1	0	0	1
0 0 1	1	0	1	0	0	1	1
0 1 0	2	0	1	1	0	0	1
0 1 1	3	1	0	0	1	1	1
1 0 0	4	1	0	1	0	0	1
1 0 1	5	1	1	0	0	1	1
1 1 0	6	1	1	1	0	0	1
1 1 1	7	0	0	0	1	1	1

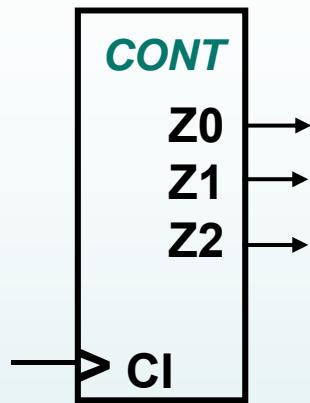
$Q Q^+$	T
0 0	0
0 1	1
1 0	1
1 1	0

Contador binario síncrono módulo 8 ascendente

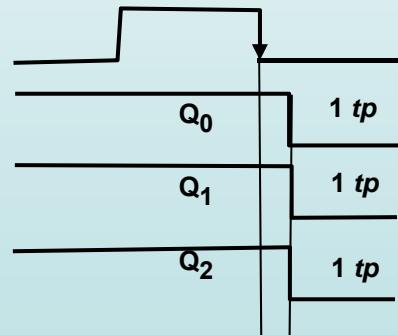
Estado actual $Q_2 Q_1 Q_0$	Funciones a realizar		
	T_2	T_1	T_0
0 0 0	0	0	1
0 0 1	1	0	1
0 1 0	2	0	1
0 1 1	3	1	1
1 0 0	4	0	0
1 0 1	5	0	1
1 1 0	6	0	0
1 1 1	7	1	1



6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.



Estado actual			Funciones a realizar		
Q_2	Q_1	Q_0	T_2	T_1	T_0
0	0	0	0	0	1
0	0	1	1	0	1
0	1	0	2	0	0
0	1	1	3	1	1
1	0	0	4	0	0
1	0	1	5	0	1
1	1	0	6	0	0
1	1	1	7	1	1

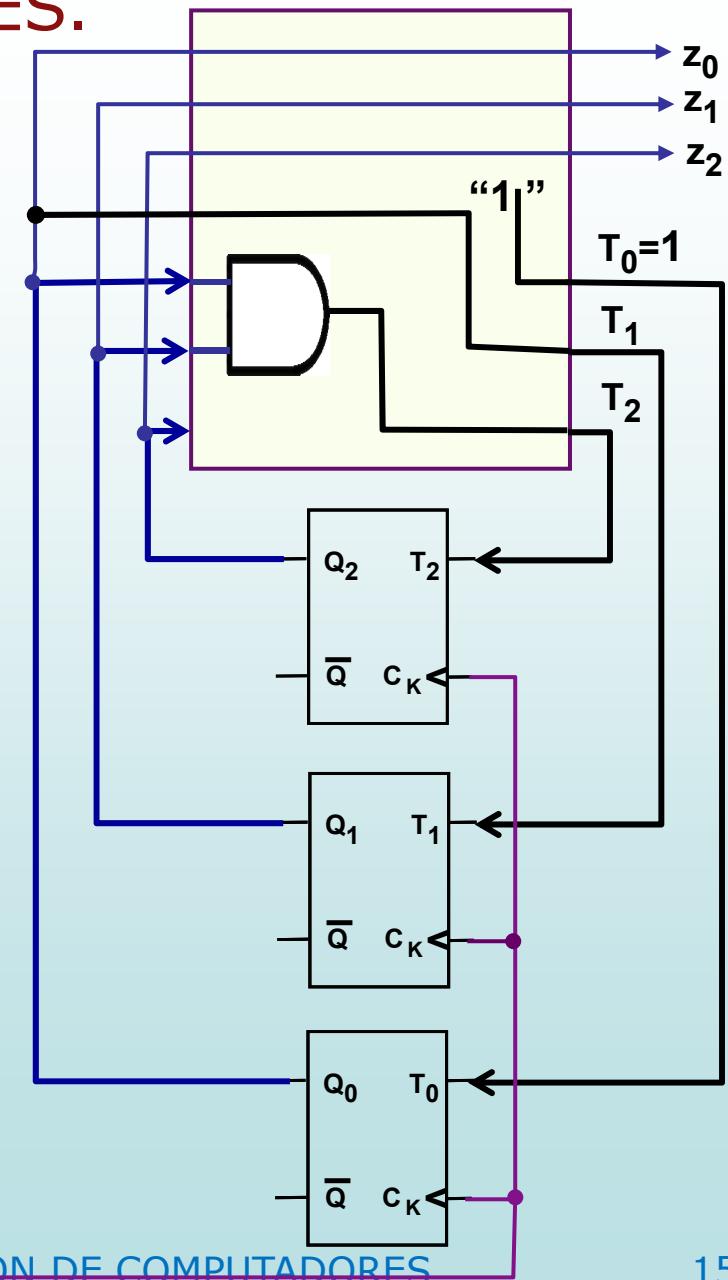


Contador binario síncrono módulo 8 ascendente

$$\begin{aligned} T_0 &= 1 \\ T_1 &= Q_0 \\ T_2 &= Q_0 \cdot Q_1 \end{aligned}$$

EN GENERAL:

$$T_n = Q_0 \cdot Q_1 \cdots Q_{n-1}$$



6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.

DISEÑO DE CONTADORES SÍNCRONOS.

Ejemplo de diseño de un contador módulo 5

Diseñar un contador módulo 5 que cuente de la forma siguiente:

$$Z=\{3,4,5,6,7; 3,4,5,6,7; \dots\}$$

a) Caso (a): Realizar el contador utilizando biestables tipo D.

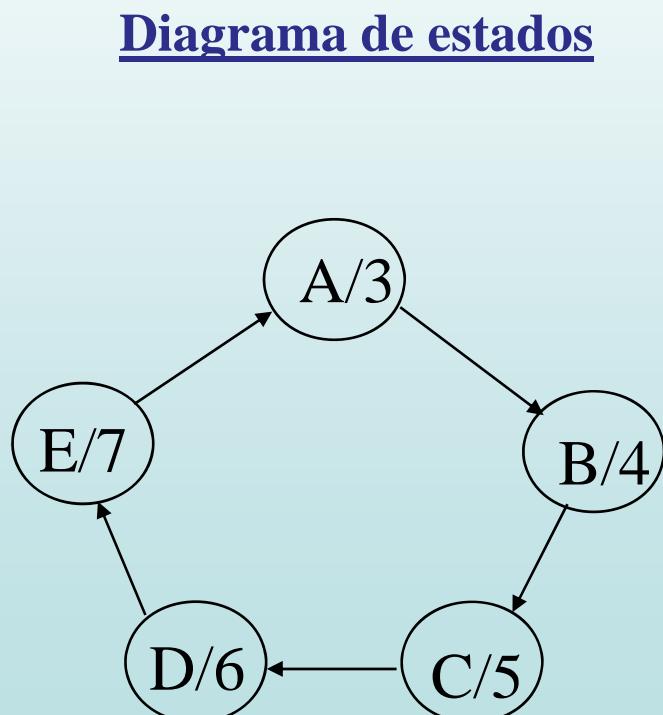
b) Caso (b): Realizar el mismo contador pero utilizando biestables tipo JK

6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.

Diseño de un contador módulo 5.

Diseñar un contador módulo 5 que cuente: $Z=\{3,4,5,6,7; 3,4,5,6,7; \dots\}$

Para ambos casos (a y b) se utiliza el mismo diagrama de estados, asignación de estados y tabla de estados.



Asignación de estados

Tabla de estados

	Estado actual $Q_2 \ Q_1 \ Q_0$			Dec	Estado Siguiente $Q_2^+ \ Q_1^+ \ Q_0^+$		
A	0	1	1	3	1	0	0
B	1	0	0	4	1	0	1
C	1	0	1	5	1	1	0
D	1	1	0	6	1	1	1
E	1	1	1	7	0	1	1

6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.

En este caso, se puede dar una asignación de estados que coincide con las salidas deseadas Z_i ya que:

- a) Ninguna salida se repite en la secuencia principal.
- b) El número de biestables mínimo necesarios es 3 (ya que 5 estados necesitan una asignación con 3 bits. Dado que las salidas requeridas Z_i son también 3 (Z_2, Z_1, Z_0), una asignación de estados coincidente con las salidas, en este caso, no incrementa el número de biestables.

Tabla del biestable D

Q^+	D
0	0
1	1

$$D = Q^+$$

Asignación de estados

Caso a: Tabla de Transición utilizando biestables tipo D

Estado actual $Q_2 \ Q_1 \ Q_0$	Dec	Estado Siguiente $D_2 \ D_1 \ D_0$ $ \ \ $ $Q_2^+ \ Q_1^+ \ Q_0^+$	Funciones de Salida.		
			Z_2	Z_1	Z_0
A 0 1 1	3	1 0 0	0	1	1
B 1 0 0	4	1 0 1	1	0	0
C 1 0 1	5	1 1 0	1	0	1
D 1 1 0	6	1 1 1	1	1	0
E 1 1 1	7	0 1 1	1	1	1

6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.

a) Caso (a): Diseño del contador con biestables tipo D

Tabla de estados

Estado actual $Q_2 \ Q_1 \ Q_0$			Dec	Estado Siguiente		
				$D_2 = Q_2^+$	$D_1 = Q_1^+$	$D_0 = Q_0^+$
0	1	1	3	1	0	0
1	0	0	4	1	0	1
1	0	1	5	1	1	0
1	1	0	6	1	1	1
1	1	1	7	0	1	1

$$D_2(Q_2 Q_1 Q_0) = \sum m_i(3,4,5,6) + d(0,1,2)$$

$$D_1(Q_2 Q_1 Q_0) = \sum m_i(5,6,7) + d(0,1,2)$$

$$D_0(Q_2 Q_1 Q_0) = \sum m_i(4,6,7) + d(0,1,2)$$

$Q_2 Q_1$		Q_0			
		00	01	11	10
Q_0	0	-- 0	-- 2	1 6	1 4
	1	-- 1	1 3	0 7	1 5

D_2

$Q_2 Q_1$		Q_0			
		00	01	11	10
Q_0	0	-- 0	-- 2	1 6	0 4
	1	-- 1	0 3	1 7	1 5

D_1

$Q_2 Q_1$		Q_0			
		00	01	11	10
Q_0	0	-- 0	-- 2	1 6	1 4
	1	-- 1	0 3	1 7	0 5

D_0

6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.

Tabla de estados

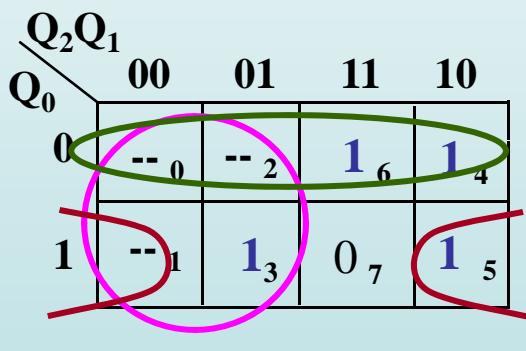
Estado actual $Q_2 \ Q_1 \ Q_0$	Dec	Estado Siguiente		
		$D_2 = Q_2^+$	$D_1 = Q_1^+$	$D_0 = Q_0^+$
0 1 1	3	1	0	0
1 0 0	4	1	0	1
1 0 1	5	1	1	0
1 1 0	6	1	1	1
1 1 1	7	0	1	1

a) Caso (a): Diseño del contador con biestables tipo D

$$D_2(Q_2 Q_1 Q_0) = \sum m_i(3,4,5,6) + d(0,1,2)$$

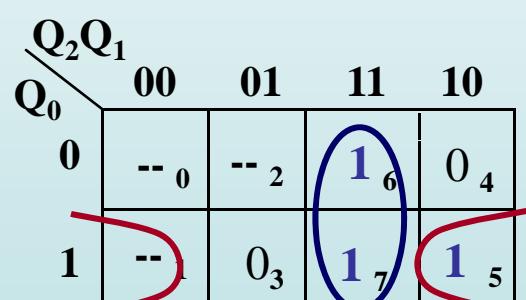
$$D_1(Q_2 Q_1 Q_0) = \sum m_i(5,6,7) + d(0,1,2)$$

$$D_0(Q_2 Q_1 Q_0) = \sum m_i(4,6,7) + d(0,1,2)$$



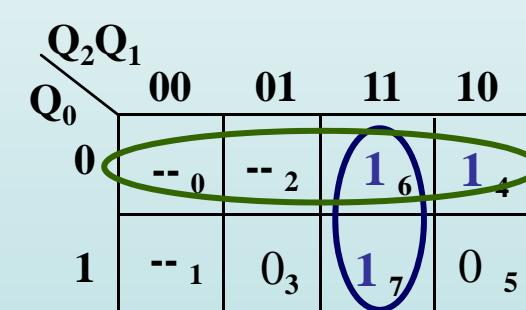
D_2

$$D_2 = \bar{Q}_0 + \bar{Q}_2 + \bar{Q}_1 Q_0$$



D_1

$$D_1 = \bar{Q}_1 Q_0 + Q_2 Q_1$$

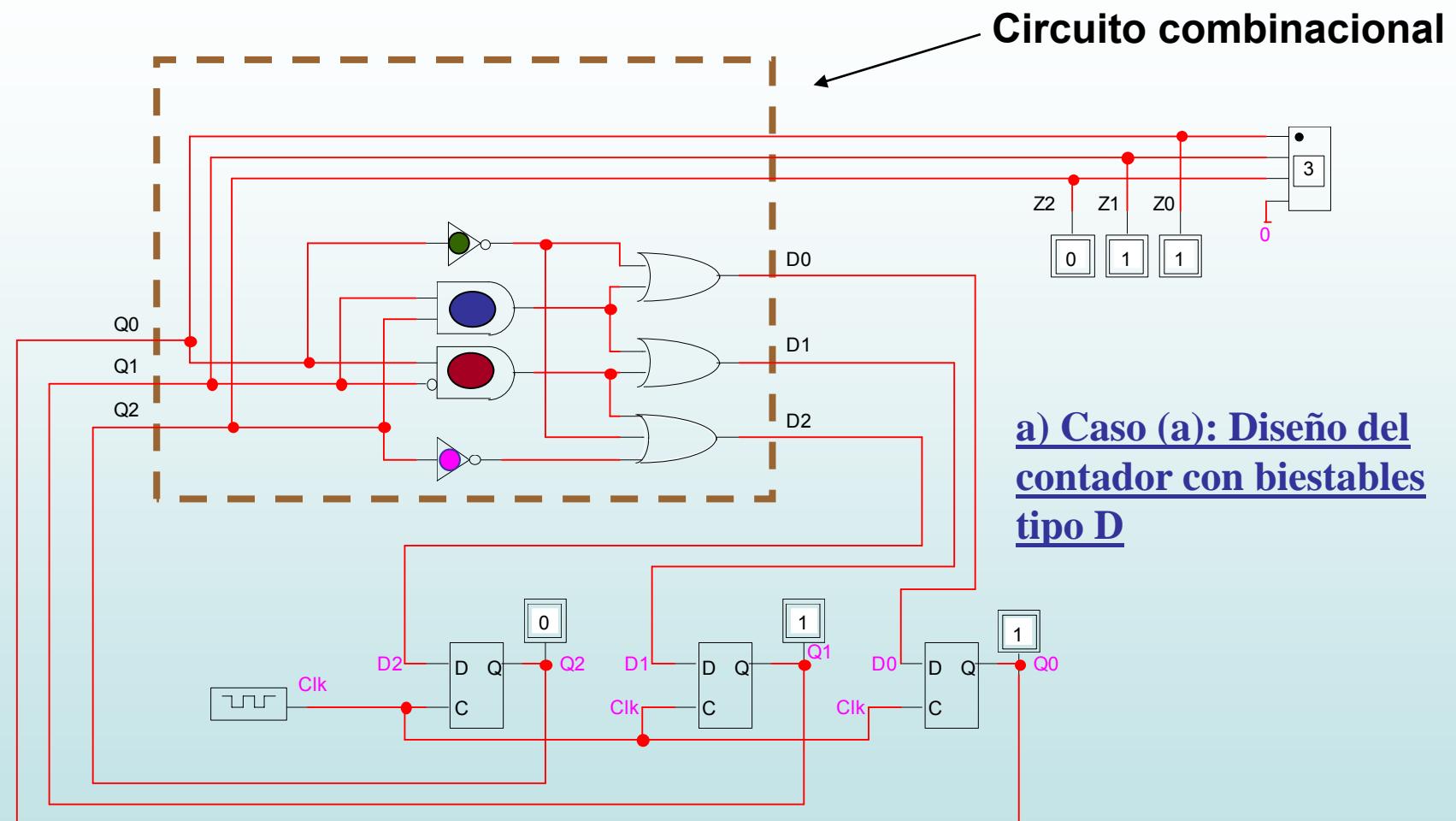


D_0

$$D_0 = \bar{Q}_0 + Q_2 Q_1$$



6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.



a) Caso (a): Diseño del contador con biestables tipo D

$$D_2 = \overline{Q}_0 + \overline{Q}_2 + \overline{Q}_1 Q_0$$

↑ ↑ ↑

$$D_1 = \overline{Q}_1 Q_0 + Q_2 Q_1$$

↑ ↑

$$D_0 = \overline{Q}_0 + Q_2 Q_1$$

↑ ↑

6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.

b) Caso (b): Diseño del contador con biestables tipo JK

En este caso, se puede dar una asignación de estados que coincide con las salidas deseadas Z_i ya que:

- a) Ninguna salida se repite en la secuencia principal.
- b) El número de biestables mínimo necesarios es 3 (ya que 5 estados necesitan una asignación con 3 bits).

Dado que las salidas requeridas Z_i son también 3 (Z_2, Z_1, Z_0), una asignación de estados coincidente con las salidas, en este caso, no incrementa el número de biestables.

Tabla de Transición

Estado actual $Q_2 \ Q_1 \ Q_0$	De c	Estado Siguiente $Q_2^+ \ Q_1^+ \ Q_0^+$	Funciones de entrada a los biestables						Funciones de Salida.		
			J_2	K_2	J_1	K_1	J_0	K_0	Z_2	Z_1	Z_0
0 1 1	3	1 0 0							0	1	1
1 0 0	4	1 0 1							1	0	0
1 0 1	5	1 1 0							1	0	1
1 1 0	6	1 1 1							1	1	0
1 1 1	7	0 1 1							1	1	1

6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.

Tabla de estados

Estado actual $Q_2 \ Q_1 \ Q_0$	Dec	Estado Siguiente $Q_2^+ \ Q_1^+ \ Q_0^+$			
			0	1	2
0 1 1	3	1 0 0			
1 0 0	4	1 0 1			
1 0 1	5	1 1 0			
1 1 0	6	1 1 1			
1 1 1	7	0 1 1			

Tabla Inversa del JK

$Q \ Q^+$	J K
0 0	0 -
0 1	1 -
1 0	- 1
1 1	- 0

b) Caso (b): Diseño del contador con biestables tipo JK

Tabla de Transición

Estado actual $Q_2 \ Q_1 \ Q_0$	De c	Estado Siguiente $Q_2^+ \ Q_1^+ \ Q_0^+$	Funciones de entrada a los biestables						Funciones de Salida.		
			$J_2 \ K_2$	$J_1 \ K_1$	$J_0 \ K_0$	Z_2	Z_1	Z_0			
0 1 1	3	1 0 0	1	-	-	1	-	1	0	1	1
1 0 0	4	1 0 1	-	0	0	-	1	-	1	0	0
1 0 1	5	1 1 0	-	0	1	-	-	1	1	0	1
1 1 0	6	1 1 1	-	0	-	0	1	-	1	1	0
1 1 1	7	0 1 1	-	1	-	0	-	0	1	1	1

6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.

Tabla de estados

Estado actual $Q_2\ Q_1\ Q_0$	Dec	Estado Siguiente $Q_2^+\ Q_1^+\ Q_0^+$
0 1 1	3	1 0 0
1 0 0	4	1 0 1
1 0 1	5	1 1 0
1 1 0	6	1 1 1
1 1 1	7	0 1 1

Tabla Inversa del JK

$Q\ Q^+$	J K
0 0	0 -
0 1	1 -
1 0	- 1
1 1	- 0

$Q_2\ Q_1$	00	01	11	10	
Q_0	0	-- 0	-- 2	-- 6	-- 4
1	-- 1	1 3	-- 7	-- 5	

$$J_2=1$$

$Q_2\ Q_1$	00	01	11	10	
Q_0	0	-- 0	-- 2	0 6	0 4
1	-- 1	-- 3	1 7	0 5	

$$K_2=Q_1 Q_0$$

b) Diseño del contador con biestables tipo JK

Tabla de Transición

Estado actual y salidas $Z_2\ Z_1\ Z_0$ $Q_2\ Q_1\ Q_0$	Dec	Estado Siguiente $Q_2^+\ Q_1^+\ Q_0^+$	Funciones de entrada a los biestables $J_2\ K_2\ J_1\ K_1\ j_0\ K_0$
0 1 1	3	1 0 0	1 -- -- 1 -- 1
1 0 0	4	1 0 1	-- 0 0 -- 1 --
1 0 1	5	1 1 0	-- 0 1 -- -- 1
1 1 0	6	1 1 1	-- 0 -- 0 1 --
1 1 1	7	0 1 1	-- 1 -- 0 -- 0

Obsérvese que no se especifican las combinaciones $(Q_2 Q_1 Q_0) = \{0, 1, 2, 3\}$ en la tabla, esto significa que corresponden a indiferencias

$Q_2\ Q_1$	00	01	11	10	
Q_0	0	-- 0	-- 2	1 6	1 4
1	-- 1	-- 3	-- 7	-- 5	

$$J_0=1$$

$Q_2\ Q_1$	00	01	11	10	
Q_0	0	-- 0	-- 2	-- 6	-- 4
1	-- 1	1 3	0 7	1 5	

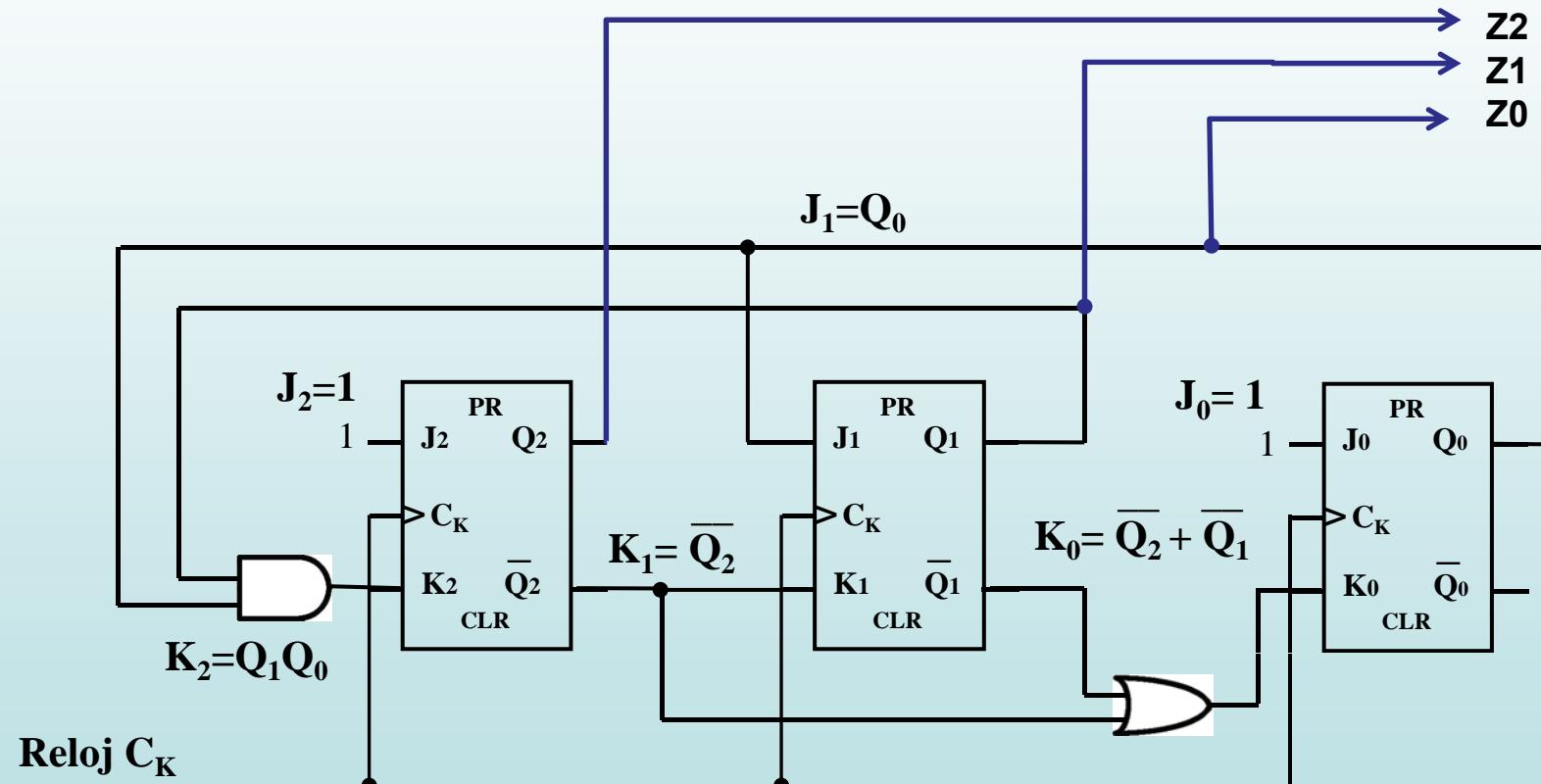
$$K_0 = \overline{Q}_2 + \overline{Q}_1$$

$Q_2\ Q_1$	00	01	11	10	
Q_0	0	-- 0	-- 2	0 6	-- 4
1	-- 1	1 3	0 7	-- 5	

$$K_1 = \overline{Q}_2$$

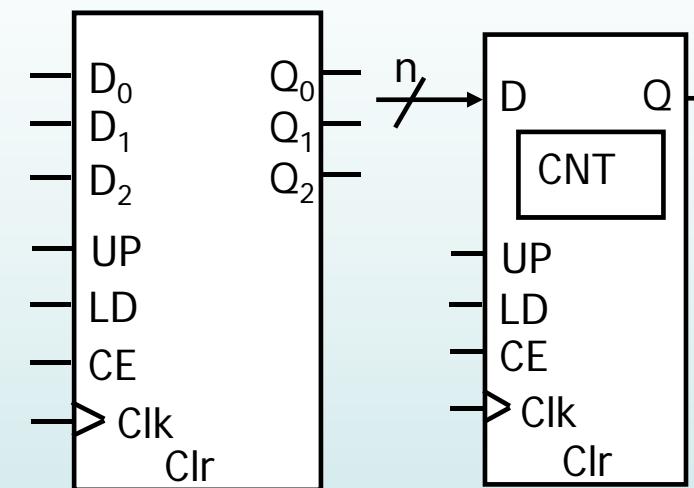
6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.

CIRCUITO DEL CONTADOR MODULO 5

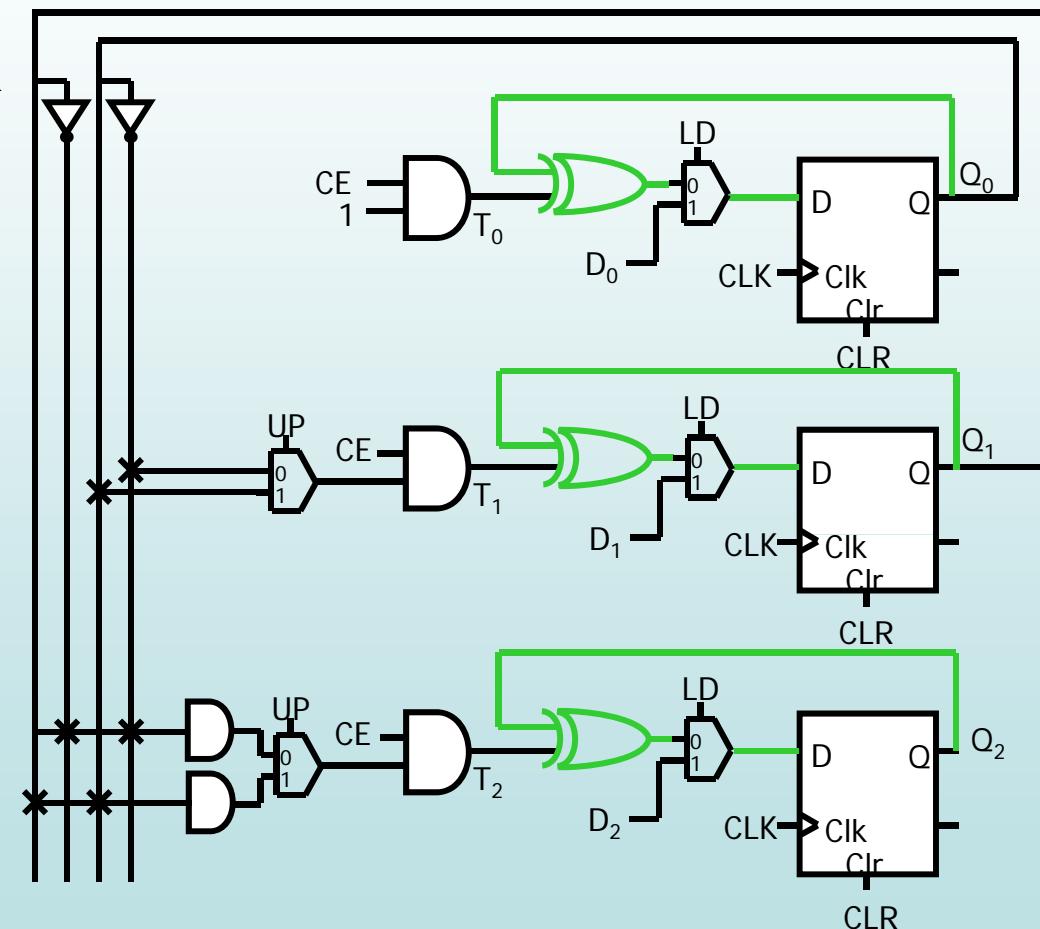


6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.

Contador binario up/down módulo 8 con señal de CE y de LD

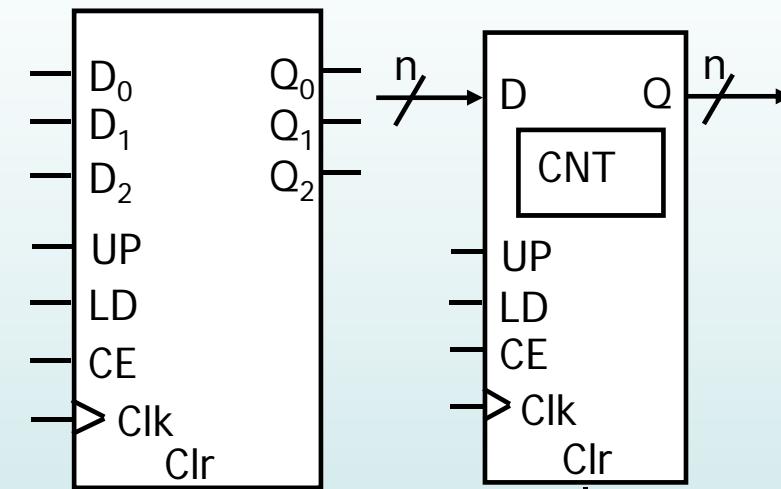


Clk	Clr	LD	CE	UP	
-	1	-	-	-	$Q_i = 0$
\uparrow	0	0	0	-	HOLD
\uparrow	0	1	-	-	$Q^+ = D$
\uparrow	0	0	1	0	$Q^+ = Q - 1$
\uparrow	0	0	1	1	$Q^+ = Q + 1$

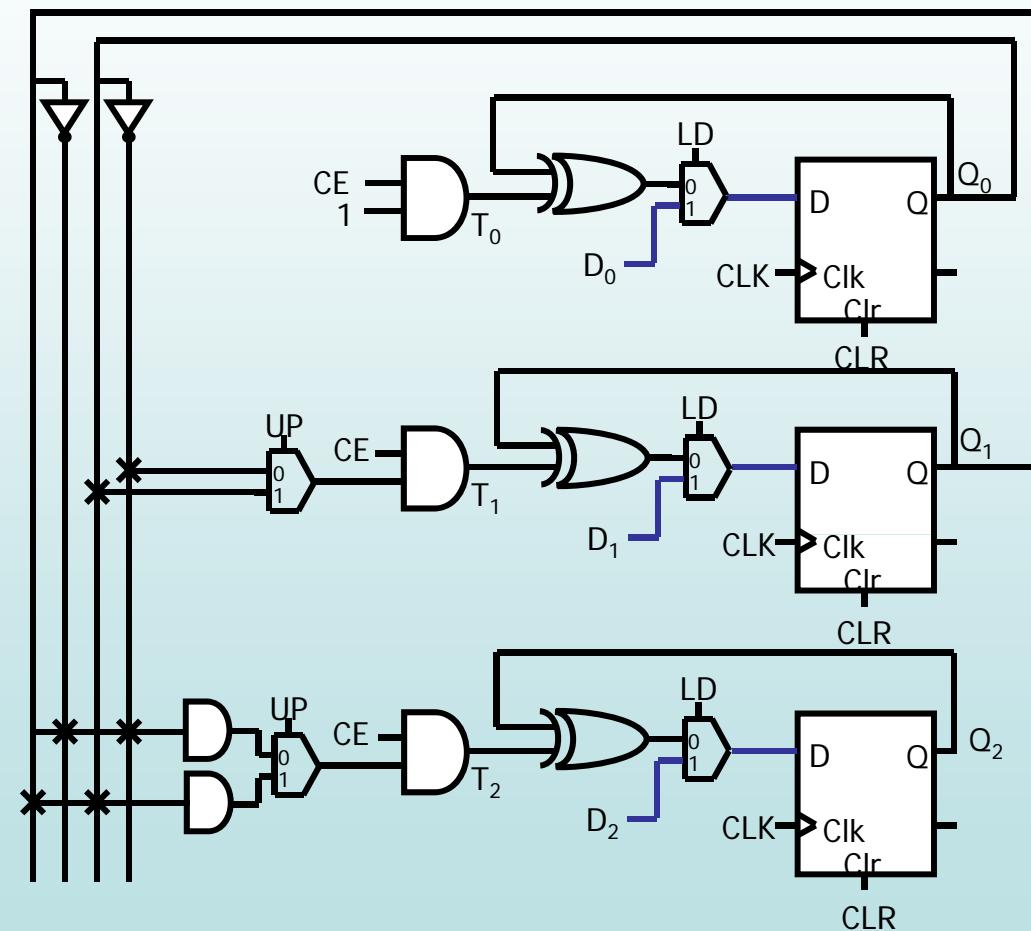


6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.

Contador binario up/down módulo 8 con señal de CE y de LD

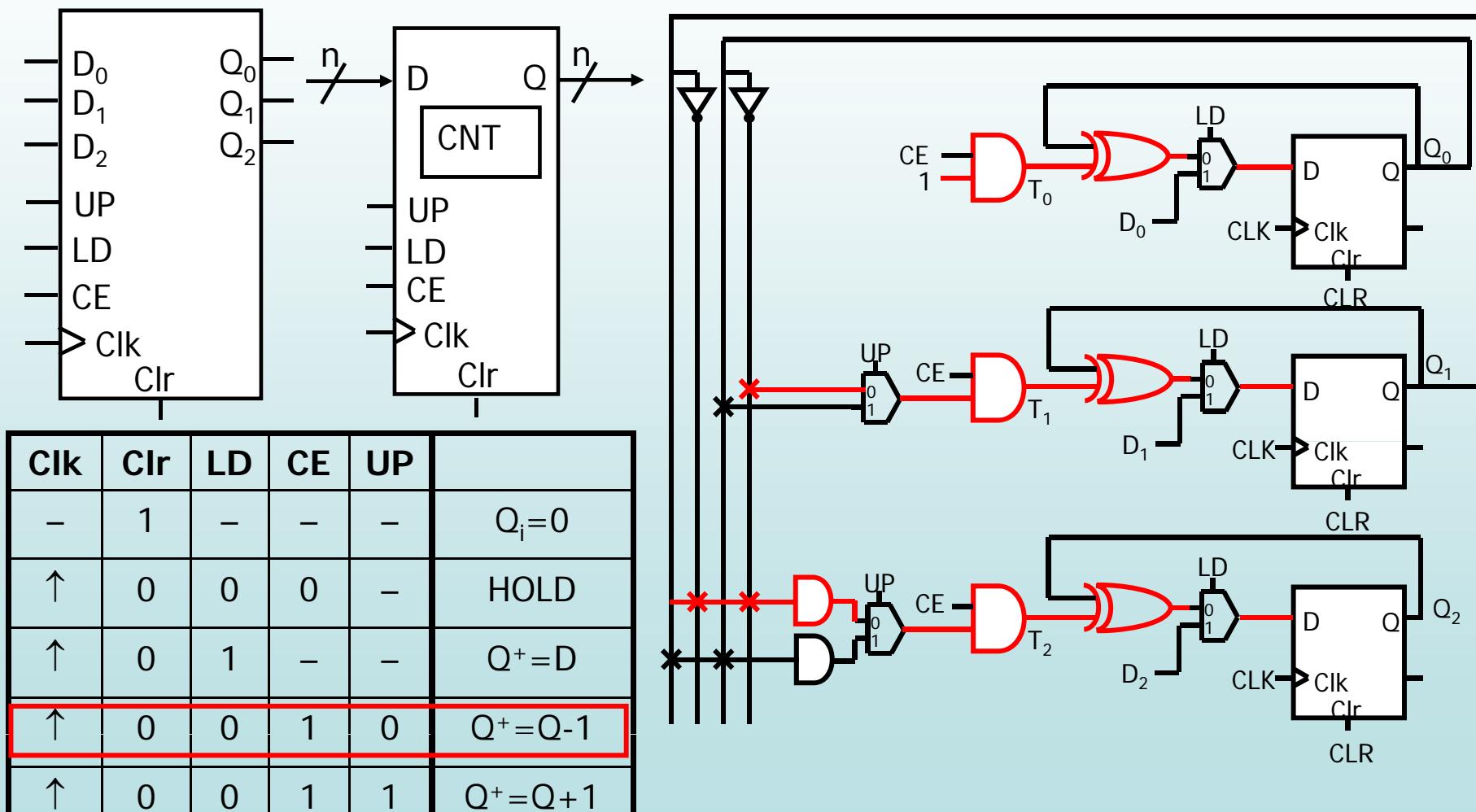


Clk	Clr	LD	CE	UP	
-	1	-	-	-	$Q_i = 0$
\uparrow	0	0	0	-	HOLD
\uparrow	0	1	-	-	$Q^+ = D$
\uparrow	0	0	1	0	$Q^+ = Q - 1$
\uparrow	0	0	1	1	$Q^+ = Q + 1$



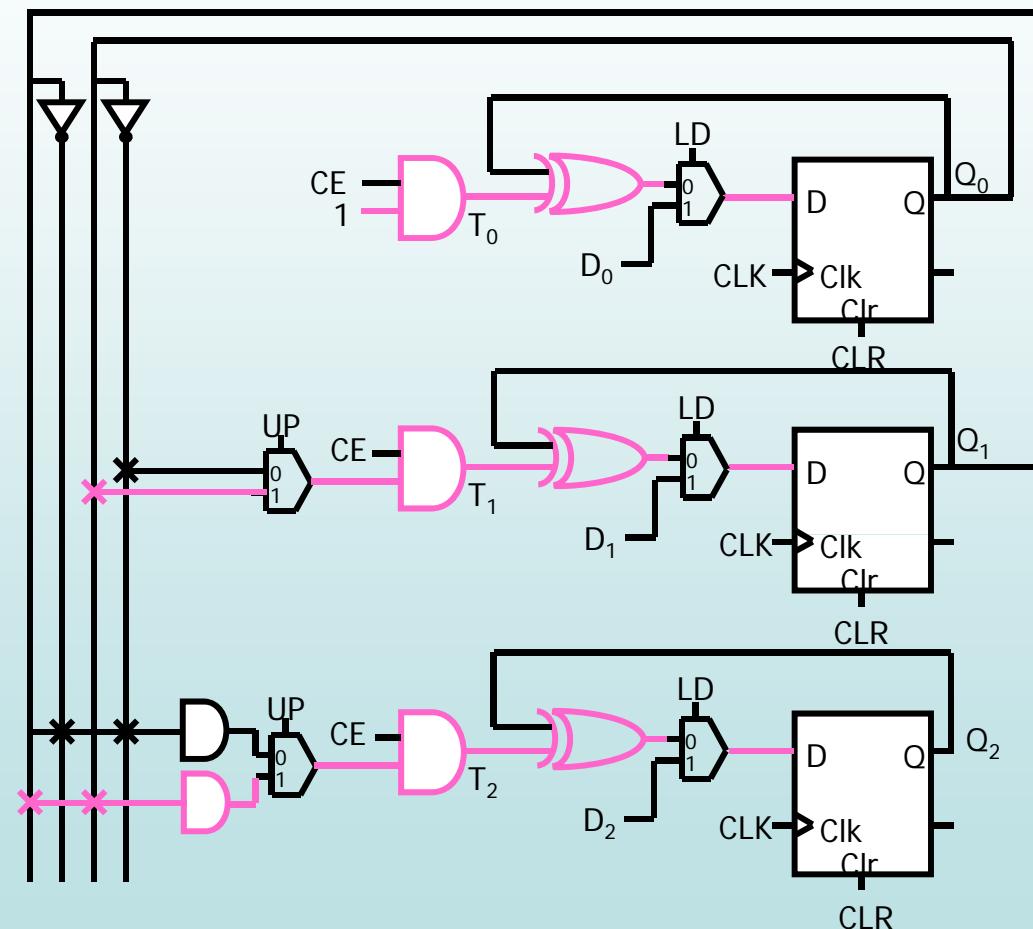
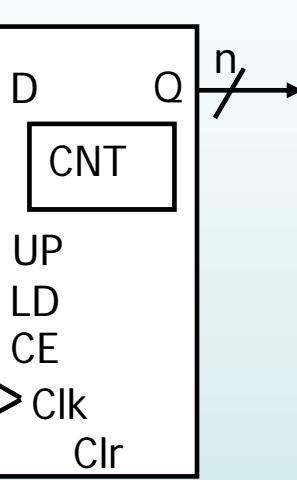
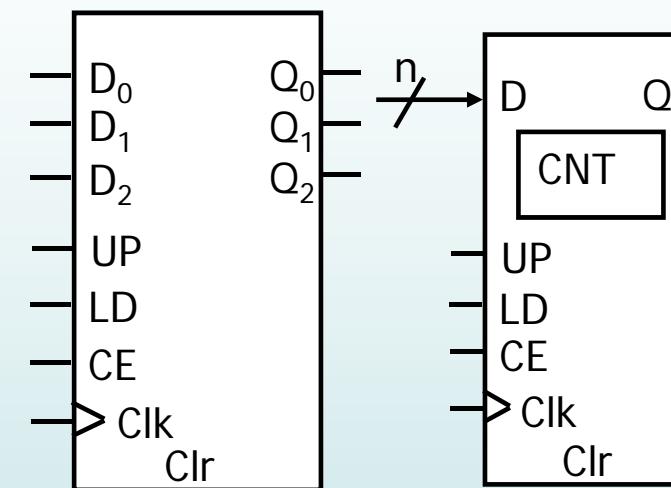
6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.

Contador binario up/down módulo 8 con señal de CE y de LD



6.5 COMPONENTES SECUENCIALES ESTÁNDAR. CONTADORES.

Contador binario up/down módulo 8 con señal de CE y de LD



Clk	Clr	LD	CE	UP	
-	1	-	-	-	$Q_i = 0$
\uparrow	0	0	0	-	HOLD
\uparrow	0	1	-	-	$Q^+ = D$
\uparrow	0	0	1	0	$Q^+ = Q - 1$
\uparrow	0	0	1	1	$Q^+ = Q + 1$

6.5 COMPONENTES SECUENCIALES ESTÁNDAR. SECUENCIADORES.

Pasos para diseñar un secuenciador o generador de secuencias:

1. Diagrama de estados
2. Tabla de estados siguientes
3. Tabla de excitación o transiciones de los flip-flops
4. Minimización
5. Implementación del contador (los generadores de secuencias se suelen implementar con biestables T).

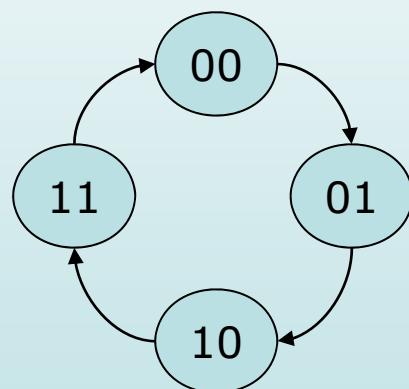
6.5 COMPONENTES SECUENCIALES ESTÁNDAR. SECUENCIADORES.

- **Ejemplo:** diseñar un sistema secuencial que genere la secuencia 1,2,5,7,1,2,... Utilizar FF-JK.
 - Hay que diferenciar entre el nº de estados diferentes por los que pasa el sistema y la cuenta que se genera.
 - El nº de estados determina el nº de biestables. En este ejemplo hay 4 **estados** diferentes, por tanto serán necesarios 2 biestables.
 - El valor máximo de la salida es 7 por lo que serán necesarias 3 **salidas** para codificar 7.

6.5 COMPONENTES SECUENCIALES ESTÁNDAR. SECUENCIADORES.

- **Ejemplo:** diseñar un sistema secuencial que genere la secuencia 1,2,5,7,1,2,.....

1. Diagrama de estados



2. Tabla de estados

Estado actual		Estado siguiente		Salidas		
Q_1	Q_0	Q_1^+	Q_0^+	Z_2	Z_1	Z_0
0	0	0	1	0	0	1
0	1	1	0	0	1	0
1	0	1	1	1	0	1
1	1	0	0	1	1	1

6.5 COMPONENTES SECUENCIALES ESTÁNDAR. SECUENCIADORES.

3. Tabla de excitación o transiciones de los FF y minimización

Q_1	Q_0	Q_1^+	Q_0^+	T_1	T_0	Z_2	Z_1	Z_0
0	0	0	1	0	1	0	0	1
0	1	1	0	1	1	0	1	0
1	0	1	1	0	1	1	0	1
1	1	0	0	1	1	1	1	1

Q	Q^+	T
0	0	0
0	1	1
1	0	1
1	1	0

		T_1	
		0	1
$Q_1 \setminus Q_0$		0	1
$T_1 = Q_0$		0	1
1			1

		Z_2	
		0	1
$Q_1 \setminus Q_0$		0	1
0			
1		1	1

		Z_0	
		0	1
$Q_1 \setminus Q_0$		0	1
0		1	
1		1	1

		T_0	
		0	1
$Q_1 \setminus Q_0$		0	1
$T_0 = 1$		1	1
0		1	1
1		1	1

		Z_1	
		0	1
$Q_1 \setminus Q_0$		0	1
0			1
1			1

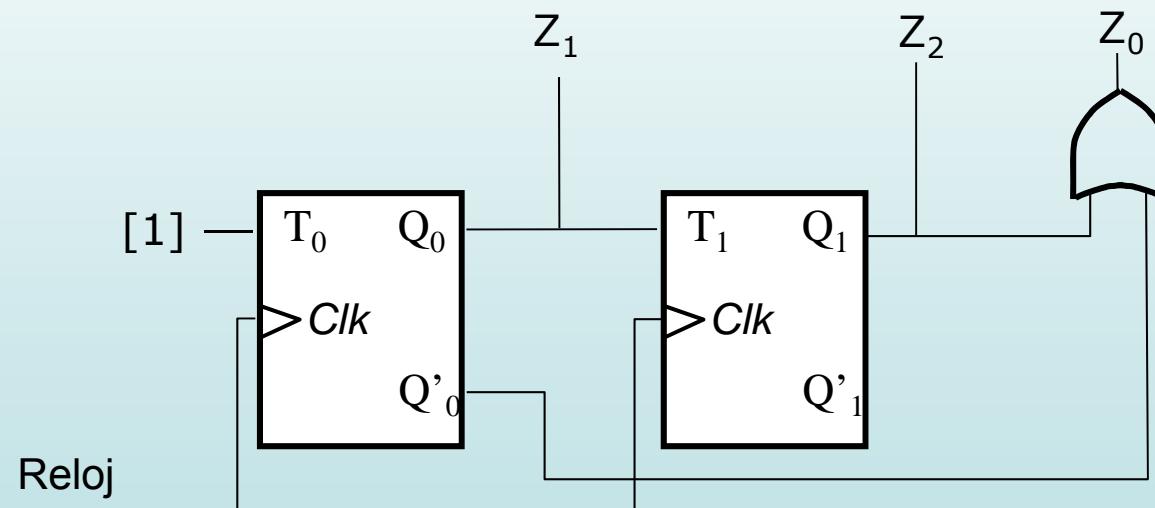
$$\begin{aligned}
 Z_2 &= Q_1 \\
 Z_1 &= Q_0 \\
 Z_0 &= Q_1 + Q'_0
 \end{aligned}$$

6.5 COMPONENTES SECUENCIALES ESTÁNDAR. SECUENCIADORES.

5. Implementación

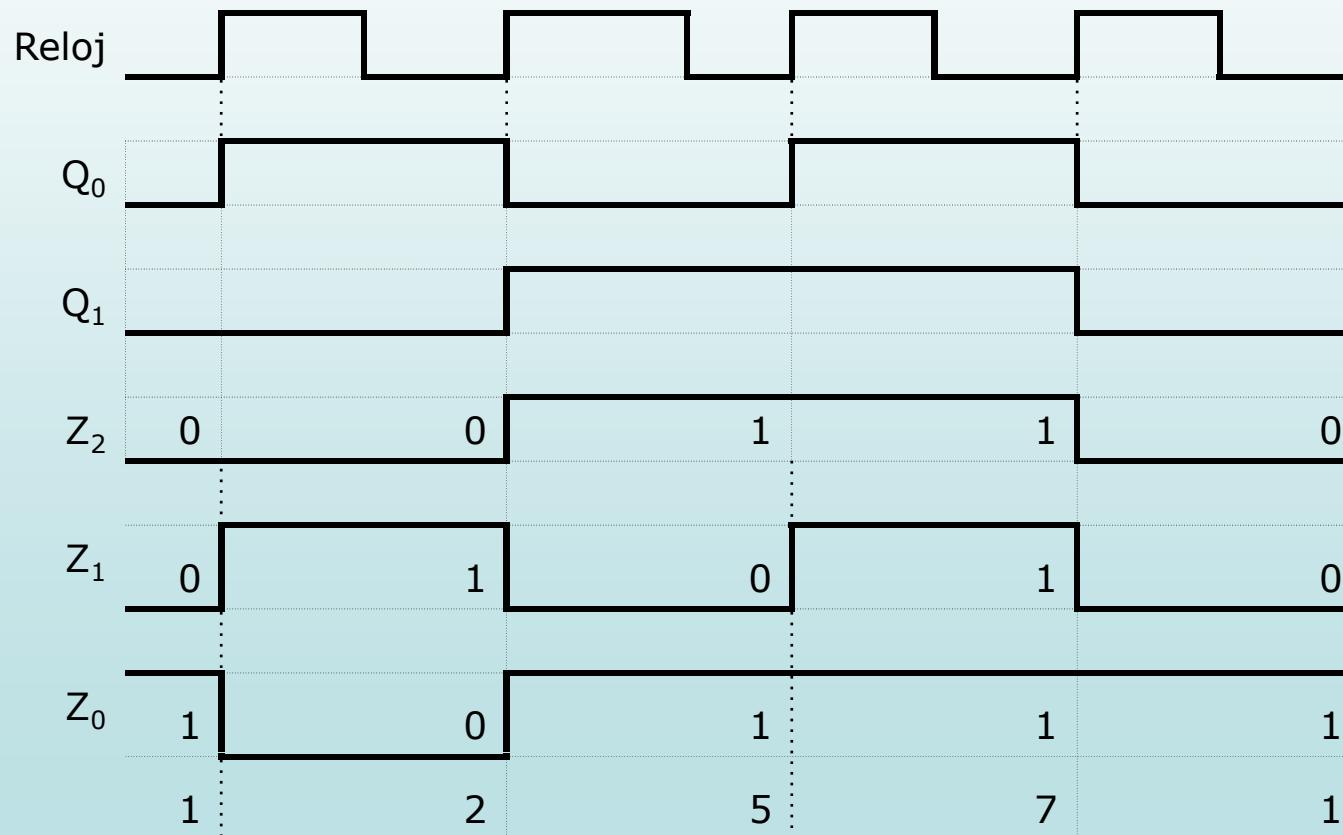
$$T_1 = Q_0 \quad Z_2 = Q_1 \quad Z_0 = Q_1 + Q'_0$$

$$T_0 = 1 \quad Z_1 = Q_0$$



6.5 COMPONENTES SECUENCIALES ESTÁNDAR. SECUENCIADORES.

- **Cronograma:** Funciones del sistema con FF-T: $Q^+ = T' \cdot Q + T \cdot Q'$
 $Q_1^+ = Q_0 \oplus Q_1$ $Q_0^+ = Q'_0$ $Z_2 = Q_1$ $Z_1 = Q_0$ $Z_0 = Q_1 + Q'_0$



6.5 COMPONENTES SECUENCIALES ESTÁNDAR. MEMORIAS RAM.

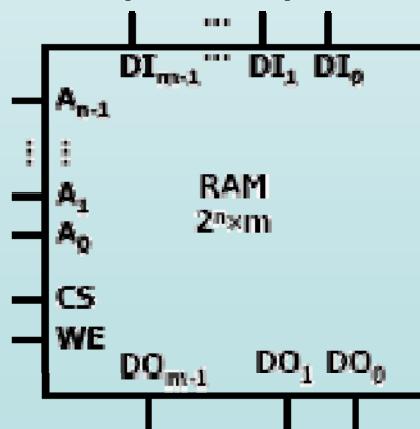
- Una memoria de acceso aleatorio (RAM) es una memoria volátil de lectura y escritura.
- Mantiene la información mientras funciona (mientras esté alimentada), no como las memorias ROM.
- Los bancos de registros son de tamaño reducido, rápidos y para memorización temporal durante los cálculos.
- Las memorias RAM son grandes, lentas, pero muy apropiadas para memorización a largo plazo de programas y datos.

6.5 COMPONENTES SECUENCIALES ESTÁNDAR. MEMORIAS RAM.

- Parámetros que caracterizan una RAM:
 - **Capacidad**: bits que puede memorizar
 - **Tiempo máximo de acceso**, t_{\max} : tiempo máximo que tarda en leer o escribir una palabra.
 - **Tiempo de ciclo**, t_c : tiempo que transcurre entre dos lecturas/escrituras consecutivas.
 - **Ancho de banda**, AB: nº máximo de palabras que se pueden transferir, por segundo, entre memoria y una unidad.
$$AB = 1/t_c$$
 - **Consumo de potencia**

6.5 COMPONENTES SECUENCIALES ESTÁNDAR. MEMORIAS RAM.

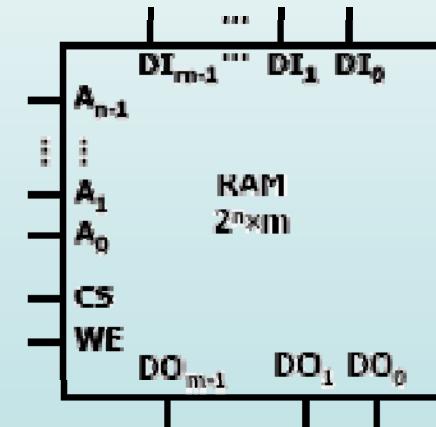
- Una **memoria RAM** de 2^n palabras de m bits es un circuito secuencial que contiene $2^n \times m$ celdas de almacenamiento (distribuidas en 2^n filas y m columnas), n entradas de dirección (A_0, \dots, A_{n-1}), m entradas de datos (DI_0, \dots, DI_{m-1}), m salidas de datos (DO_0, \dots, DO_{m-1}), y varias señales de control (CS, WE) que permite:
 - La **lectura** de cualquier palabra de m bits en cualquier fila
 - La **escritura** de cualquier palabra de m bits en cualquier fila



6.5 COMPONENTES SECUENCIALES ESTÁNDAR. MEMORIAS RAM.

Las memorias RAM suelen tener al menos dos señales de control, una para **seleccionar la celda** y otra para decidir si se hace una **lectura o escritura**.

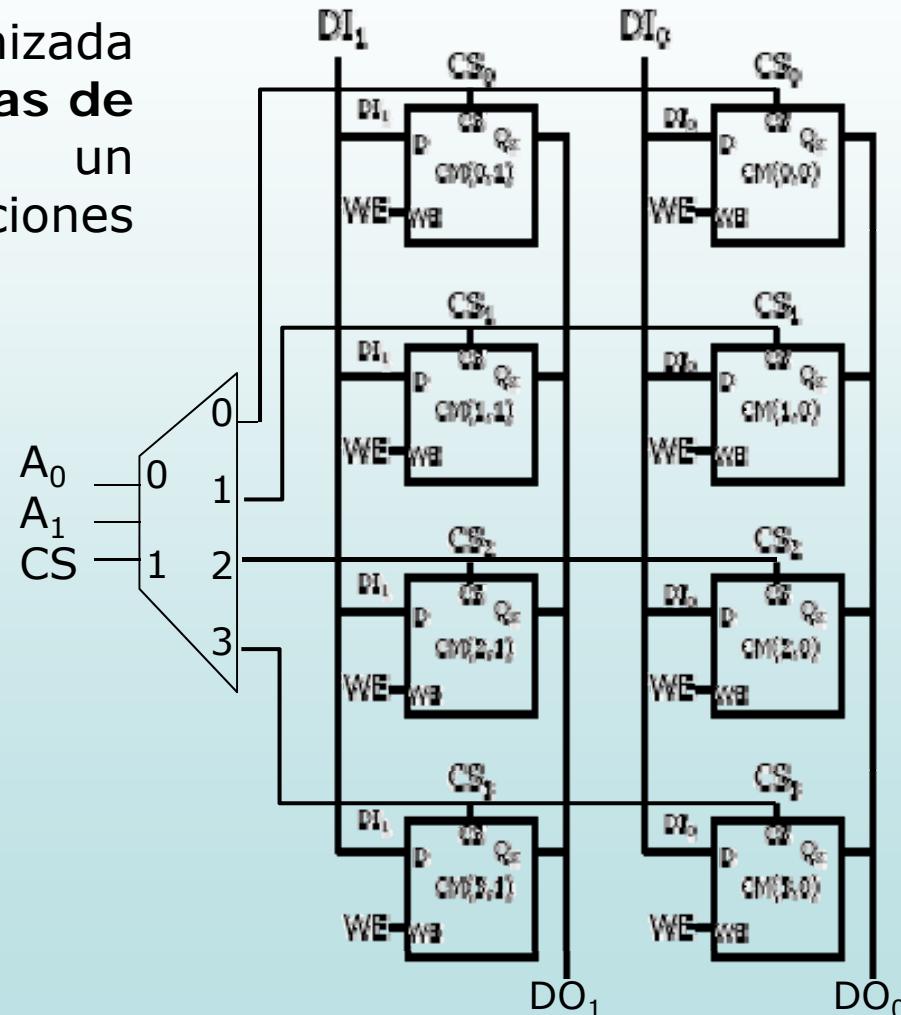
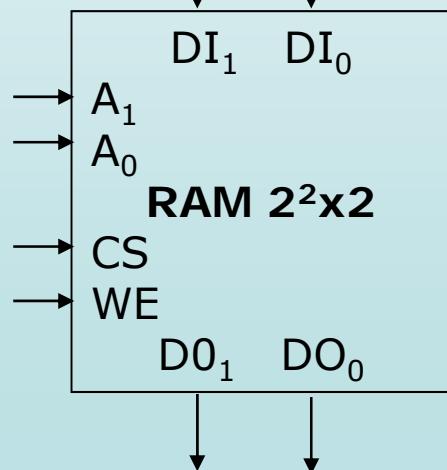
- CS = 0: No se escribe en ninguna celda y salida en alta impedancia
- CS=1:
 - Si WE = 0 (lee)
 $DO_{m-1}, \dots, DO_0 = M(A_{n-1}, \dots, A_0)$
(Lectura)
 - Si WE = 1 (escribe)
 $M(A_{n-1}, \dots, A_0) = DI_{m-1}, \dots, DI_0$
(Escritura)



6.5 COMPONENTES SECUENCIALES ESTÁNDAR. MEMORIAS RAM.

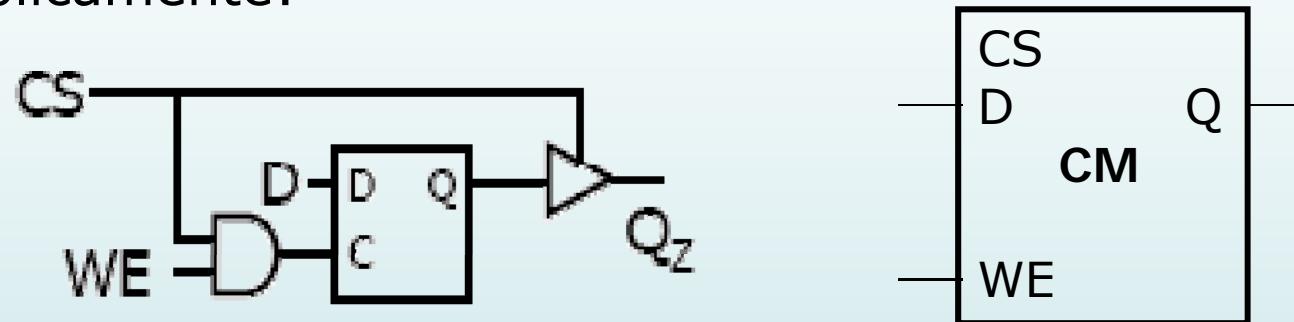
Una RAM está organizada en una matriz de **celdas de memoria** con un decodificador de direcciones y un adaptador de I/O.

Ejemplo: RAM $2^2 \times 2$



6.5 COMPONENTES SECUENCIALES ESTÁNDAR. MEMORIAS RAM.

- Cada celda de memoria (CM) se puede representar simbólicamente:

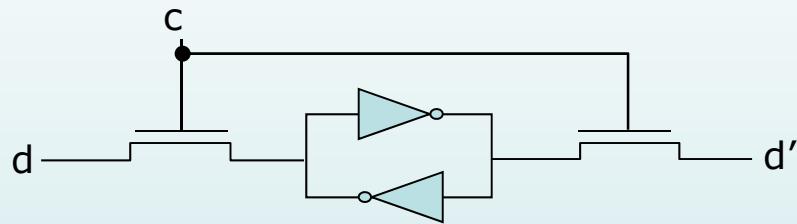


- CS=1, el bit memorizado aparece a la salida ($Q_Z = Q$)
- WE=1, la entrada se memoriza
- WE actúa como señal de reloj del FF-D

6.5 COMPONENTES SECUENCIALES ESTÁNDAR. MEMORIAS RAM.

En realidad las CM se implementan con menos transistores.

- Si se utiliza **RAM estática, SRAM**:

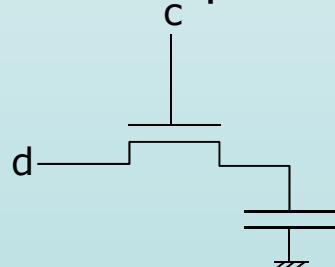


Celda elemental SRAM:

- Mantienen la información mientras haya alimentación
- Se leen muy rápido
- Entre 4 y 6 transistores

- Si se usa **RAM dinámica, DRAM**:

- hay que refrescar periódicamente la información que contienen (la información se pierde en cada lectura)
- Celdas más sencillas
- Mayor densidad
- Menor coste
- 2 transistores



Celda elemental DRAM

6.5 COMPONENTES SECUENCIALES ESTÁNDAR. MEMORIAS RAM.

- La organización de la RAM impone restricciones en la temporización de las entradas y salidas a la hora de hacer lecturas y escrituras.
 - Ciclo de lectura:

Parámetro	Significado	Descripción
t_{AA}	<i>Tiempo de acceso desde la dirección</i>	Tiempo requerido para generar un dato válido de salida después de un cambio de dirección (sup. /CE y /OE activas)
t_{ACS}	<i>Tiempo de acceso desde CS (CE)</i>	Tiempo requerido generar un dato válido de salida después de que se active /CE (sup. /OE activa y dirección estable)
t_{OE}	<i>Tiempo de habilitación de salida</i> ($t_{OE} < t_{ACS}$)	Tiempo que tardan los buffers de salida en dejar de estar en alta impedancia cuando /OE y /CE están ambas activas.
t_{OZ}	<i>Tiempo de inhabilitación de salida</i>	Tiempo necesario para que los buffers de salida se pongan en alta impedancia cuando alguna de las dos señales /OE o /CE deja de estar activa.
t_{OH}	<i>Tiempo de mantenimiento de salida</i>	Tiempo en que la salida permanece válida después de que se haya producido un cambio de dirección.

6.5 COMPONENTES SECUENCIALES ESTÁNDAR. MEMORIAS RAM.

– Ciclo de escritura:

- La dirección debe estar estable antes de que se habilite la escritura
- Los datos se almacenan en los latches cuando se desactiva /WE ó /CE (previamente activas ambas)
- Deben estar estables antes de que termine el ciclo de escritura

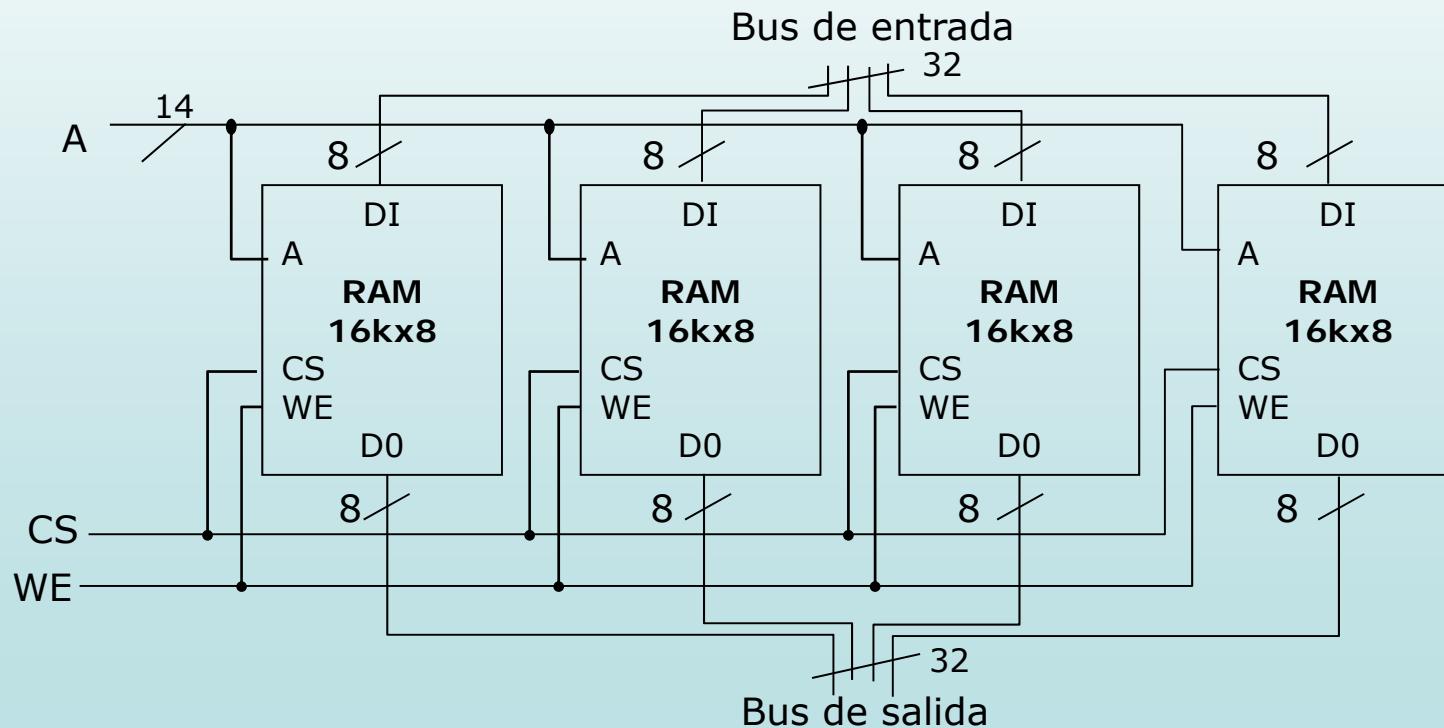
Parámetro	Significado	Descripción
t_{AS}, t_{AH}	<i>Tiempos de establecimiento y mantenimiento de direcciones</i>	Las entradas de dirección deben estar estables antes de que /CE y /WE se activen, y mantenerse después de que alguna /CE o /WE se desactive).
t_{CSW}	<i>Tiempo de establecimiento de CS(CE) antes de finalizar la escritura</i>	/CE debe estar activa un cierto tiempo antes de que finalice el ciclo de escritura para garantizar la selección de una celda
t_{WP}	<i>Anchura del pulso de escritura</i>	/WE debe estar activa al menos este tiempo para garantizar la escritura en el latch
t_{DS}, t_{DH}	<i>Tiempos de establecimiento y mantenimiento de datos</i>	Los datos deben estar estables antes de que finalice la escritura y deben permanecer un cierto tiempo después.

6.5 COMPONENTES SECUENCIALES ESTÁNDAR. MEMORIAS RAM.

- Los componentes de memoria se fabrican siempre con tamaños de $2^n \times m$.
- Cuando se necesitan memorias de otros tamaños se construyen a partir de los chips de memoria disponibles.
- Se pueden construir memorias:
 - Con mayor longitud de palabra (palabras más largas)
 - Con mayor capacidad

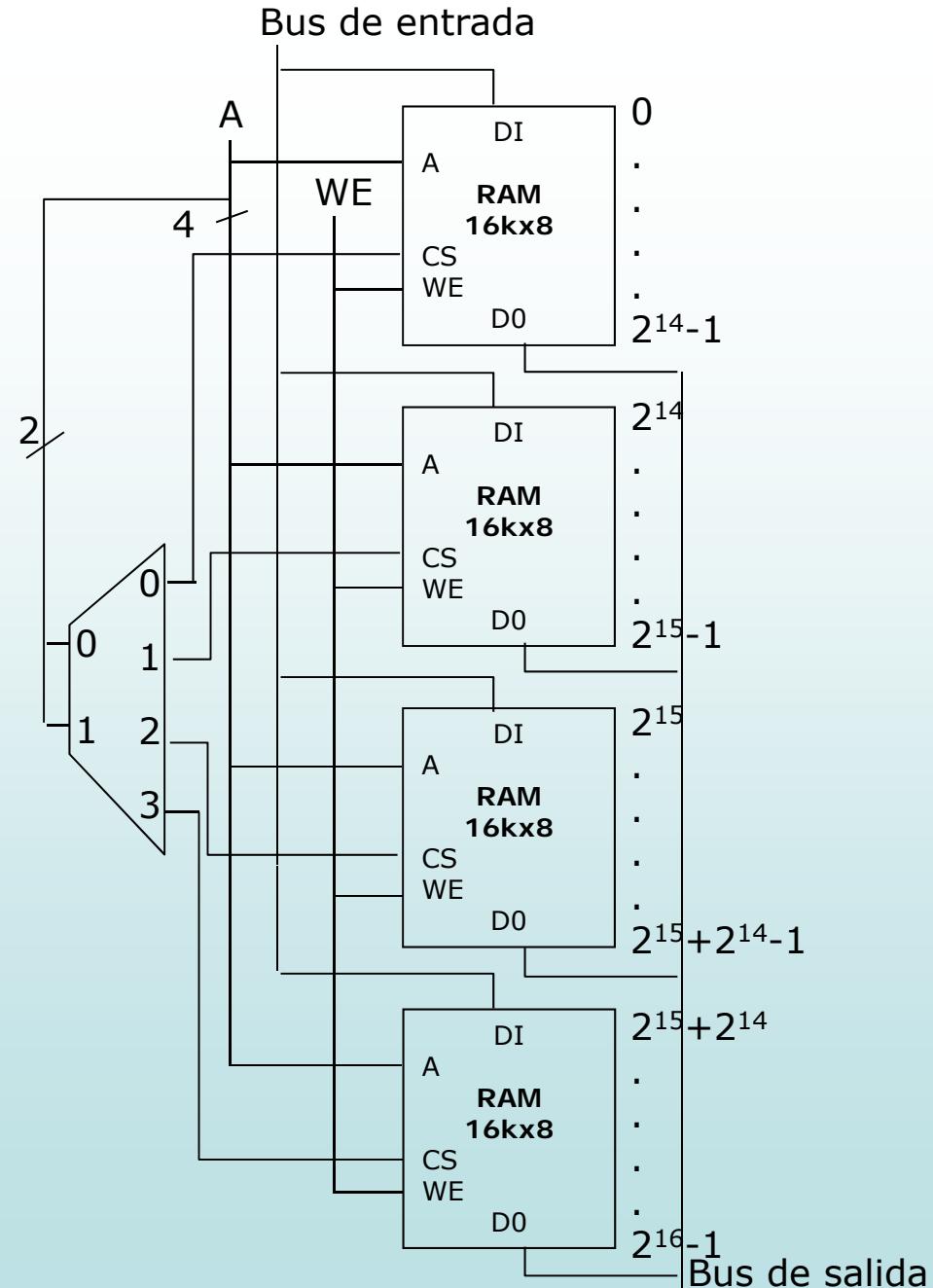
6.5 COMPONENTES SECUENCIALES ESTÁNDAR. MEMORIAS RAM.

- Para obtener **mayor longitud de palabra** se pueden conectar varios chips de memoria en paralelo.
- Por ejemplo: se puede hacer una RAM de 16K x 32 conectando en paralelo 4 RAM de 16K x 8.



6.5 COMPONENTES SECUENCIALES ESTÁNDAR. MEMORIAS RAM.

- Para obtener memorias de **mayor capacidad** se pueden conectar varios chips de memoria en serie.
- Por ejemplo: construir una RAM de 64K x 8 con 4 RAM de 16K x 8.



Entrada/salida y buses

Estructura de Computadores

12^a Semana

Bibliografía:

- | | |
|---------------|---|
| [HAM03] Cap.4 | Organización de Computadores. Hamacher, Vranesic, Zaki. McGraw-Hill 2003
Signatura ESIIT/ C.1 HAM org |
| [STA8] Cap.7 | Organización y Arquitectura de Computadores, 7 ^a Ed. Stallings. Pearson Educación, 2008.
Signatura ESIIT/ C.1 STA org |

Guía de trabajo autónomo (4h/s)

■ Lectura

- Cap. 4 Hamacher
- Cap. 7 Stallings

Bibliografía:

[HAM03] Cap.4

Organización de Computadores. Hamacher, Vranesic, Zaki. McGraw-Hill 2003

Signatura ESIIT/[C.1 HAM org](#)

[STA8] Cap.7

Organización y Arquitectura de Computadores, 7^a Ed. Stallings. Pearson Educación, 2008.

Signatura ESIIT/[C.1 STA org](#)

[

Entrada/salida y buses

- Funciones del sistema de E/S. Interfaces de E/S
- E/S programada
- Interrupciones
- DMA (Acceso directo a memoria)
- Estructuras de bus básicas
- Especificación de un bus. Transferencias. Temporización. Arbitraje
- Ejemplos y estándares

Objetivo de los sistemas de E/S

- Realizar la conexión del procesador con una gran variedad de dispositivos periféricos, teniendo en cuenta que las características de los dispositivos de E/S suelen diferir notablemente de las del procesador; en especial:
 - la **velocidad** de transmisión de los periféricos
 - normalm. **menor** que la velocidad a la que opera el procesador
 - muy **variable** (pocos bytes/s hasta >100 MB/s)
 - la **longitud de palabra**
 - los **códigos** para representar los datos
- Para compatibilizar las características de los dispositivos de E/S con el sistema procesador/memoria se usan los circuitos de interfaz o controladores de periféricos.

Interfaces de E/S

- **Circuitos de interfaz o controladores de periféricos:**
 - Circuitos de **adaptación de formato de señales y características de temporización** entre el procesador y los dispositivos de E/S.
 - Proporcionan todas las transferencias de datos necesarias entre el procesador y los periféricos, utilizando un bus de E/S.
 - **Requieren uso de software:**
 - Programas de E/S ejecutados por el procesador que controlan la transferencia de información hacia y desde los dispositivos de E/S.
 - En computadores de altas prestaciones se han utilizado procesadores especializados para las funciones de E/S: **procesadores de E/S (IOP)** o canales.
 - Procesadores cuyos conjuntos de instrucciones se restringen a aquellas que se precisan en las operaciones de E/S.
 - Se hacen cargo de todas las transferencias con los periféricos.

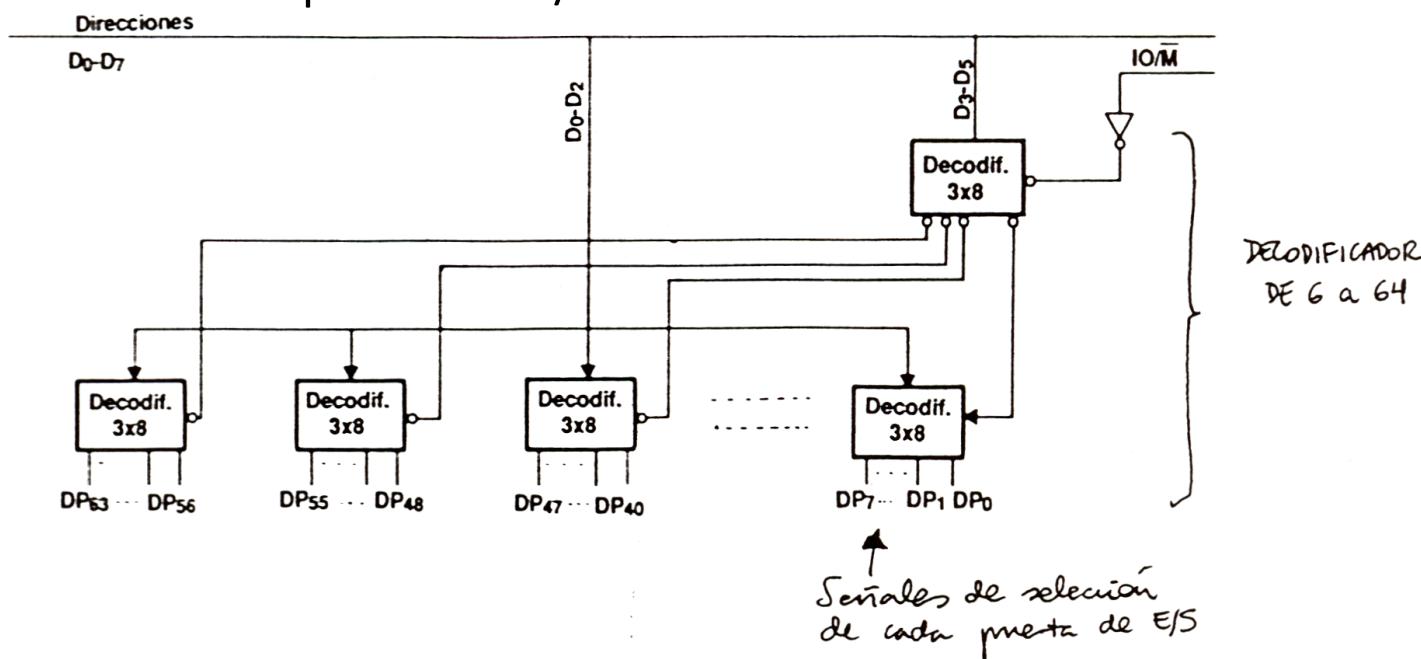
Funciones que debe incluir el sistema de E/S (I)

■ *Direccionamiento o selección del periférico:*

- El procesador sitúa en el bus de direcciones la dirección asociada con el dispositivo.
- Si se conectan varios periféricos debe preverse la forma de que no haya conflictos de acceso al bus.
- Con p bits \Rightarrow pueden direccionarse 2^p direcciones distintas (mapa de E/S).
 - Cada dirección especifica uno o dos puertos de E/S:
 - El hardware de cada dirección suele ser único (bien entrada o bien salida).
 - Pero a veces los circuitos de E y de S de una única dirección son independientes (misma dirección \Rightarrow dos puertos, uno de entrada y otro de salida).
 - Cada interfaz de periférico emplea varios puertos para comunicarse con el procesador.

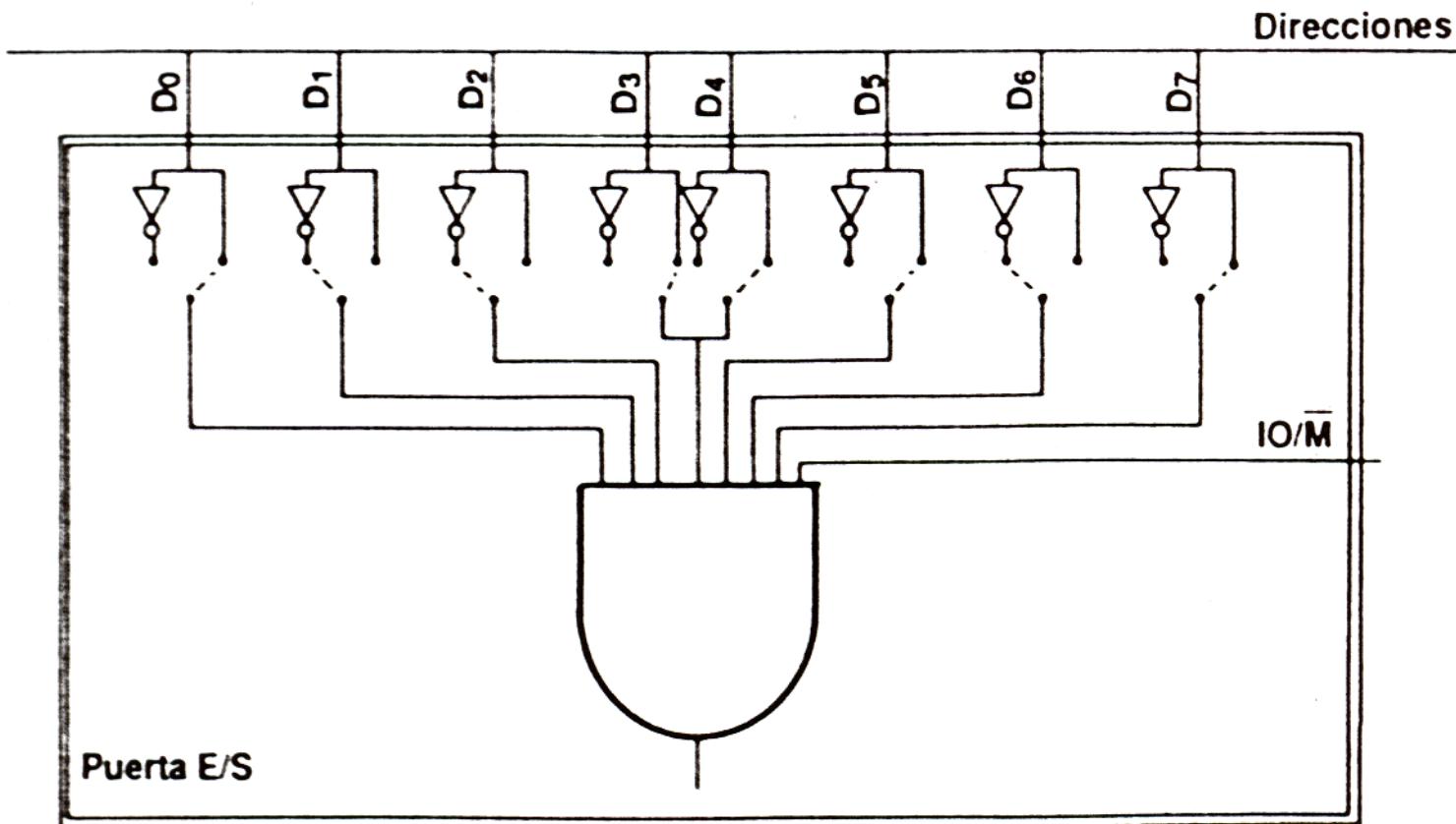
Funciones que debe incluir el sistema de E/S (II)

- Técnicas de direccionamiento
 - ✗ **Direccionamiento por selección lineal:** asignar un bit del bus de direcciones a cada puerto.
 - ✓ **Direccionamiento por decodificación:** decodificar los bits de dirección para seleccionar un puerto de una interfaz.
 - **Decodificación centralizada:** un decodificador selecciona cada puerto de E/S.



Funciones que debe incluir el sistema de E/S (III)

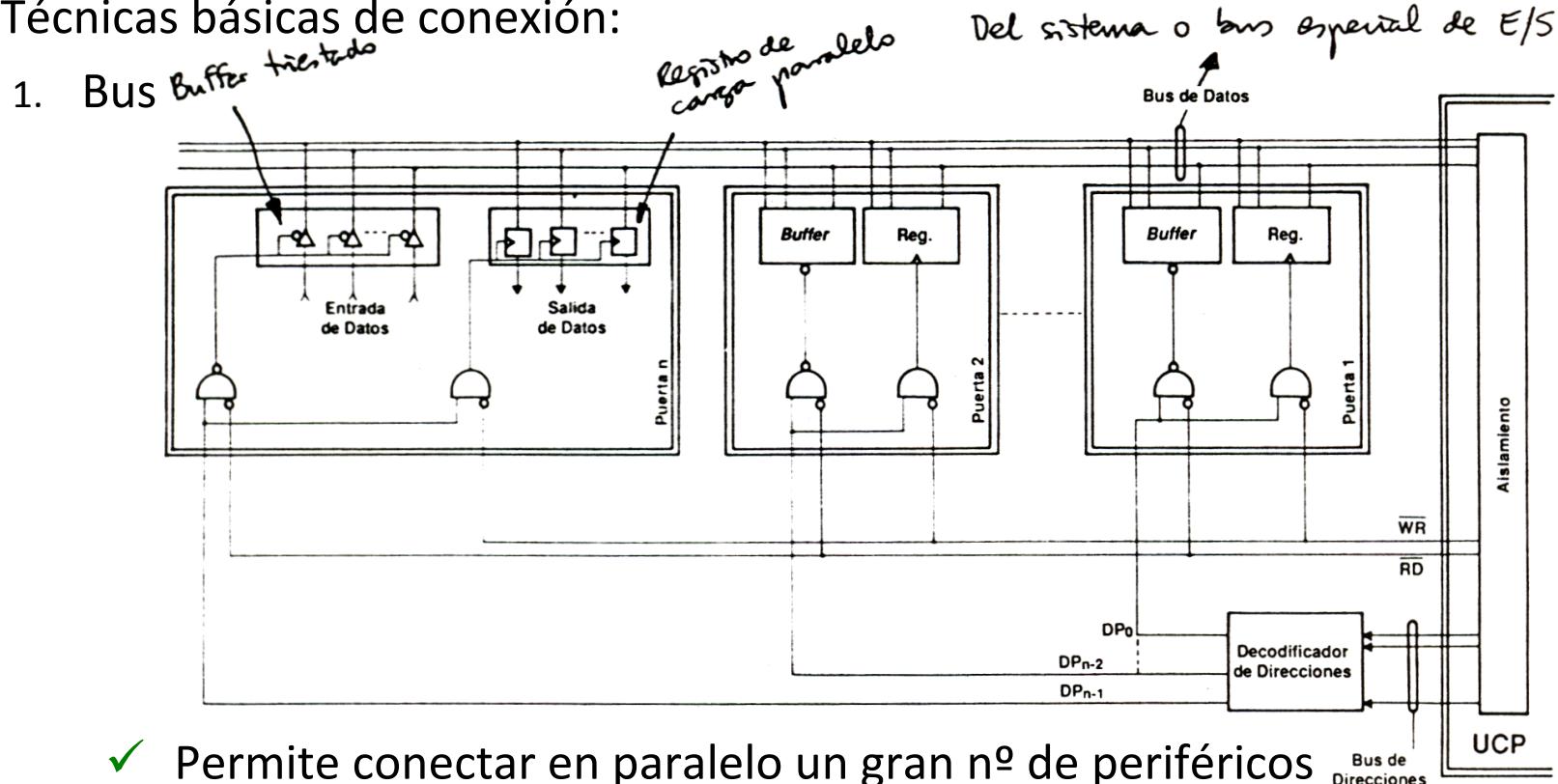
- Decodificación en cada puerto de E/S: cada puerto reconoce su propia dirección.



Funciones que debe incluir el sistema de E/S (IV)

■ Comunicación física entre el periférico y el procesador.

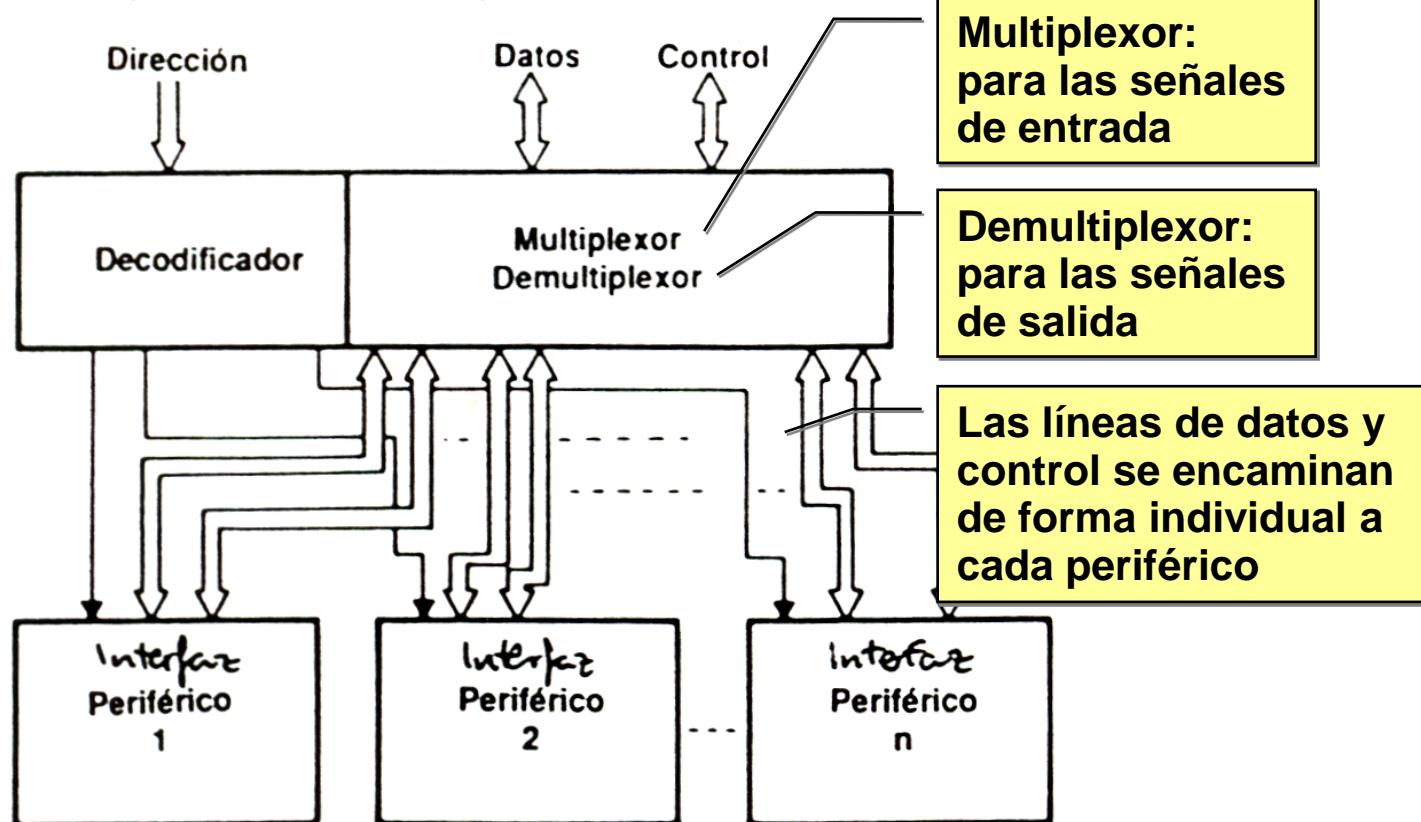
- Técnicas básicas de conexión:



- ✓ Permite conectar en paralelo un gran nº de periféricos
- ✓ Es fácil expandir el sistema (añadiendo más tarjetas o circuitos de interfaz)

Funciones que debe incluir el sistema de E/S (V)

2. Multiplexor / demultiplexor



- ✗ Expansión difícil
- ✗ Mucha circuitería

Funciones que debe incluir el sistema de E/S (VI)

■ *Sincronización:*

- Acomodación de las velocidades de funcionamiento del procesador/MP y los dispositivos de E/S.
- Hay que establecer un mecanismo para saber cuándo se puede enviar o recibir un dato.
- Deben incluirse:
 - Palabras de memoria temporal en la interfaz que sirvan como **búfer**. La entrada o salida se hace sobre este búfer intermedio. La operación de E/S real se realiza sólo cuando el dispositivo está preparado.
 - **Señales de control de conformidad** para iniciar o terminar la transferencia (listo, petición, reconocimiento).
 - La temporización de las transferencias puede ser:
 - Síncrona —————
 - Asíncrona

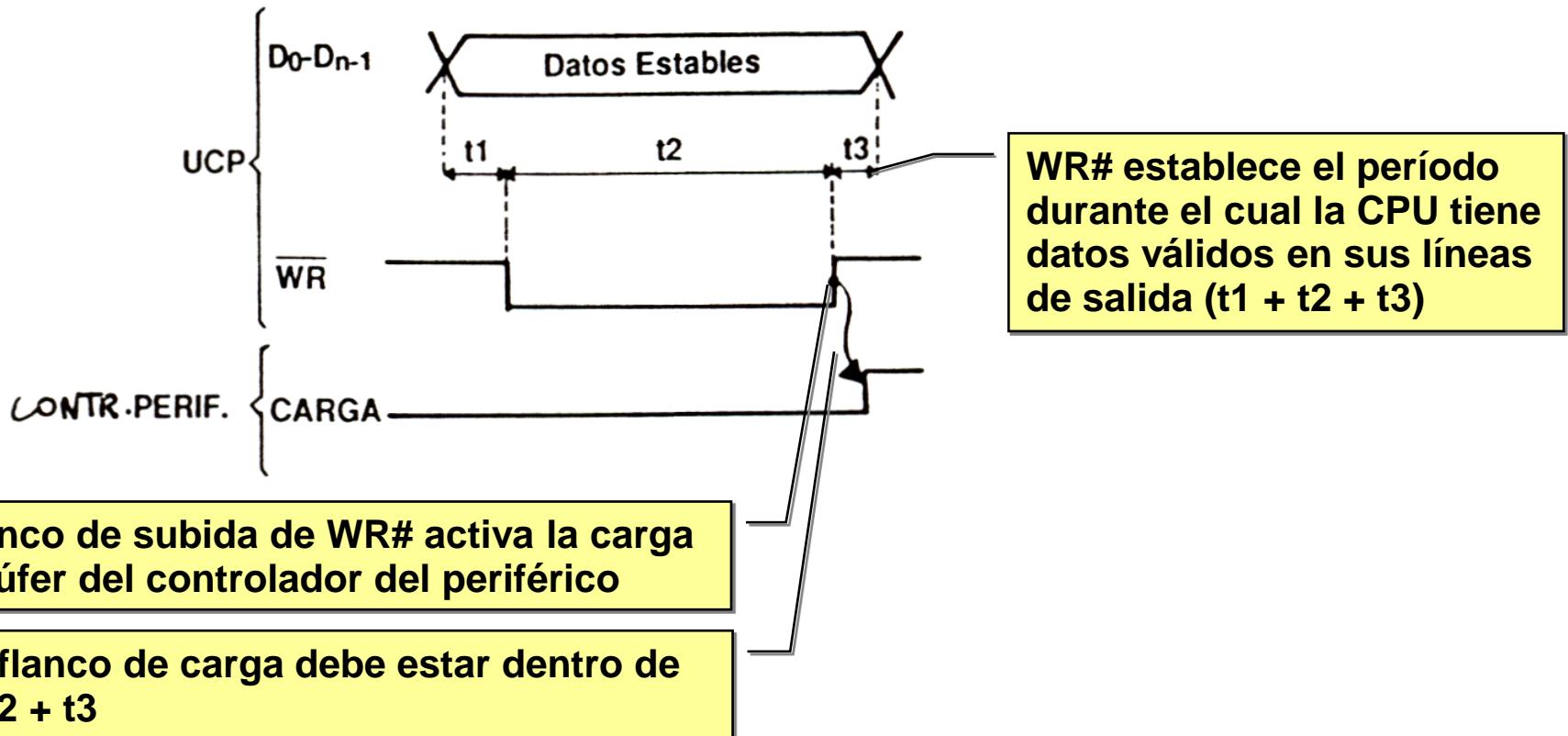
**¡Ojo! concepto confuso,
depende del autor**

Funciones que debe incluir el sistema de E/S (VII)

■ Temporización síncrona

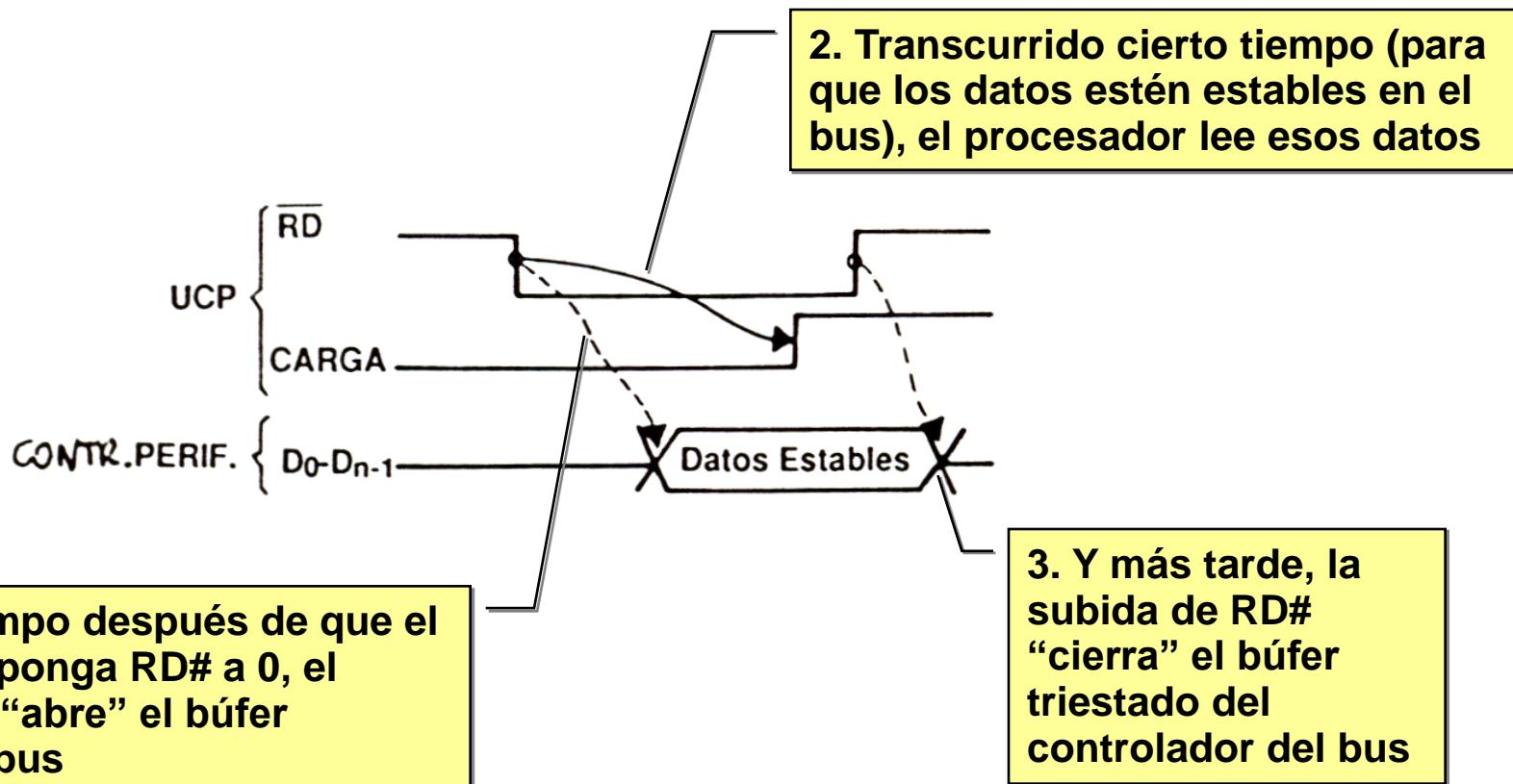
▪ Escritura

- El procesador sitúa los datos en el bus de datos y al mismo tiempo genera una señal WR# = 0.



Funciones que debe incluir el sistema de E/S (VIII)

- Lectura
 - El procesador suministra una señal de lectura RD# = 0. A partir de entonces el controlador del periférico dispone de cierto tiempo para suministrar el dato al bus de datos.



Funciones que debe incluir el sistema de E/S (IX)

- En la temporización síncrona...
 - El **procesador establece la temporización**, que debe ser seguida por el controlador del periférico.
 - ✖ Si éste no es capaz de leer el bus, o de poner el dato, en el tiempo establecido ⇒ la transferencia fracasa.
 - » Podría haberse direccionado un dispositivo lento, inexistente o apagado.
 - » En lectura el procesador almacenará un dato incorrecto.
 - » En escritura el procesador no sabrá si ha escrito con éxito.

Funciones que debe incluir el sistema de E/S (X)

- Temporización asíncrona o con “handshaking”
 - Se necesita una nueva señal de aceptación (ACK) con la que el controlador del periférico contesta a la petición de transferencia generada por el procesador.
 - Escritura
-
- Hasta que no se produce el flanco de subida de ACK, no se cierra la transferencia, subiendo el procesador las señales WR# o RD#
- Lectura
-
- Hasta que no se produce el flanco de subida de ACK, no se cierra la transferencia, subiendo el procesador las señales WR# o RD#

Funciones que debe incluir el sistema de E/S (XI)

- En la temporización asíncrona o con “handshaking”...
 - Se establece un **diálogo (*handshaking*)** para adaptar el cronograma a las necesidades de tiempo del periférico.
 - ***Handshaking***: establecimiento de una comunicación sincronizando dos dispositivos mediante acuse de recibo o intercambio de señales de control.
 - Ventajas:
 - ✓ Se pueden **conectar dispositivos con distintos requisitos de tiempo**.
 - ✓ Se tiene una **mayor garantía** de que el **dato sea válido**, puesto que se exige una contestación positiva del periférico.
 - Para evitar que el sistema quede bloqueado si no existe el periférico o éste no contesta, es necesario establecer un **período de espera máximo**, después del cual se considera la transferencia como errónea.

Funciones que debe incluir el sistema de E/S (XII)

■ *Conversión de datos:*

- Acomodación de las características físicas y lógicas de las señales de datos empleadas por el dispositivo de E/S y por el bus del sistema.
 - Conversión de códigos (BCD, ASCII, EBCDIC, UNICODE, ANSI, etc.)
 - Conversión serie / paralelo.
 - Conversión de niveles lógicos para representar 1 y 0.
 - Conversión A/D y D/A.

■ *Control de los periféricos:*

- Interrogación y modificación de su estado:
 - Encendido, apagado, disponible...
- Envío de otras señales de control al periférico.

Funciones que debe incluir el sistema de E/S (XII)

- *Mecanismo que determine la cantidad de información a transmitir en una operación de E/S y cuente el número de palabras / bytes ya transmitidos.*
- *Detección de errores*
 - En el funcionamiento del periférico...
 - ...o en los datos
 - Mediante códigos de paridad, polinomiales, etc.
 - Se repetirá la transferencia en caso necesario.

Conceptos a diferenciar (I)

■ Transferencia elemental de información

- Envío o recepción de una única unidad de información (byte o palabra), ya sea un dato o una palabra de estado o control.



■ Operación completa de E/S

- Transferencia de un conjunto de datos
 - Sector de un disco
 - Línea de pantalla

Conceptos a diferenciar (II)

■ Dispositivos de E/S físicos

- Cuando el ordenador carece de SO o *driver* adecuados.
- El programador debe tratar directamente con ellos, asumiendo sus detalles de funcionamiento y características físicas.



■ Dispositivos de E/S lógicos

- El programador efectúa las transferencias de datos activando las rutinas de E/S que proporciona el SO.
- Por ejemplo, el programador puede escribir un programa que escriba en una impresora lógica (ésta en realidad puede ser un bloque de espacio en disco). El SO asigna una de las impresoras físicas a la impresora lógica y controla el proceso de impresión (*spooling*).

Conceptos a diferenciar (III)

■ E/S aislada o independiente

- El procesador distingue internamente entre espacio de memoria y espacio de E/S.



■ E/S mapeada en memoria

- El procesador no distingue entre accesos a memoria y accesos a los dispositivos de E/S.

E/S independiente frente a E/S en memoria

■ E/S aislada, independiente, o con espacio de E/S

- Emplea la **patilla IO/M#** del procesador
 - **Nivel alto** ⇒ Indica a memoria y a dispositivos de E/S que se va a efectuar una operación de **E/S**.
 - Al ejecutar instrucciones específicas de E/S: IN y OUT.
 - **Nivel bajo** ⇒ Operación de intercambio de datos con **mem.**
 - Al ejecutar instrucciones de acceso a memoria: LOAD, STORE o MOVE.
- **Instrucciones específicas:** IN y OUT (o READ y WRITE), con poca riqueza de direccionamiento.
- Ejemplo:
 - procesadores de 8 bits empleaban dirección de 8 bits para puertos de E/S ⇒ 256 puertos: IN puerto, OUT puerto
 - y disponían de bus de direcciones de 16 bits ⇒ 64 K dir. memoria

E/S independiente frente a E/S en memoria

- Ventajas:
 - ✓ Diseño más limpio de la decodificación de las direc. de memoria.
 - ✓ Facilita la protección de E/S (por ejemplo, haciendo que las instrucciones IN, OUT,... sean privilegiadas).
 - ✓ Los programas son relativamente más rápidos por la decodificación más sencilla y el menor tamaño de las instrucciones de E/S.
- Desventajas:
 - ✗ Mayor complejidad en el diseño del procesador:
 - Hay que decodificar y ejecutar las instrucciones IN, OUT...
 - Hay que generar la señal IO/M# y se necesita una patilla más del procesador para ella.

E/S independiente frente a E/S en memoria

■ E/S mapeada en memoria

- Se usan algunas **direcciones de memoria** para acceder a los puertos de E/S, tras decodificarlas adecuadamente.
- El **procesador no distingue** entre accesos a memoria y accesos a los dispositivos de E/S.
 - **No** se usa la patilla IO/M#
- **No se dispone de instrucciones especiales**, sino que se usan LOAD, STORE o MOVE.
- Para evitar particionar el mapa dedicado a memoria, se agrupa la E/S en una zona bien definida al principio o final del mapa de memoria.

E/S independiente frente a E/S en memoria

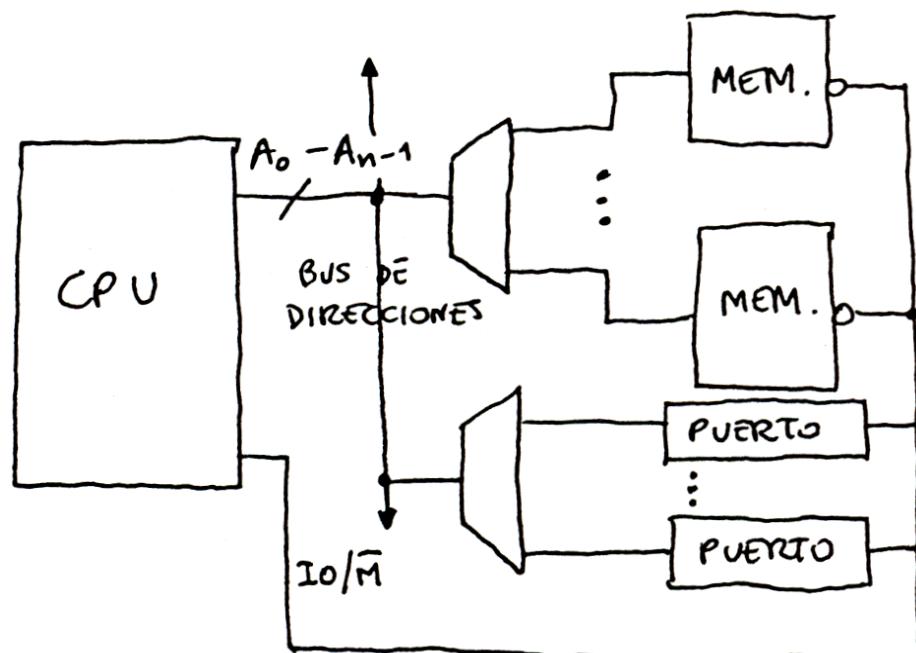
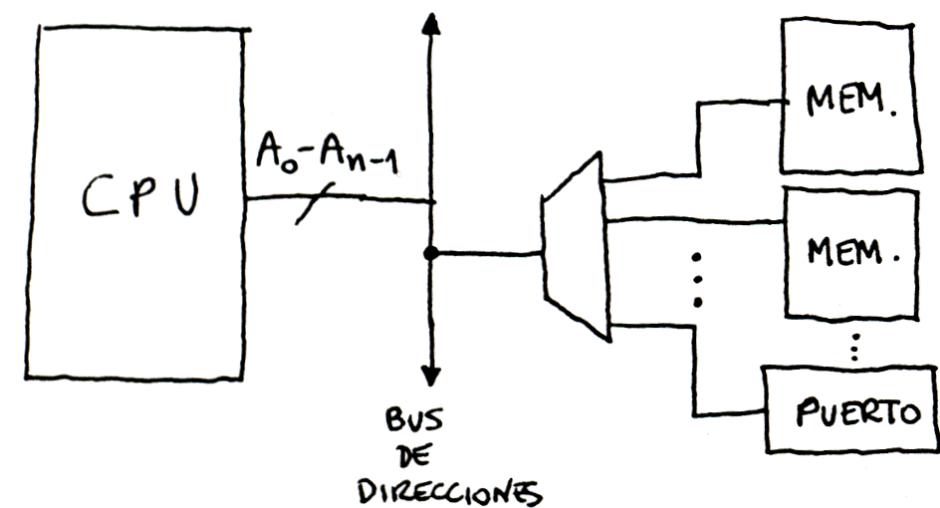
- Ventaja:
 - ✓ Menor complejidad en el diseño del procesador.
- Desventajas:
 - ✗ Cada puerto de una interfaz “ocupa” una dirección que no puede utilizarse para memoria.
 - ✗ Las instrucciones de acceso a memoria suelen ser más largas que las específicas de E/S.
 - Por ej., en los microprocesadores de 8 bits: 3 bytes frente a 2.
 - Puede disminuir la velocidad de procesamiento.
 - Aumentan los requisitos de memoria.

E/S independiente frente a E/S en memoria

Esquemas:

E/S mapeada en memoria

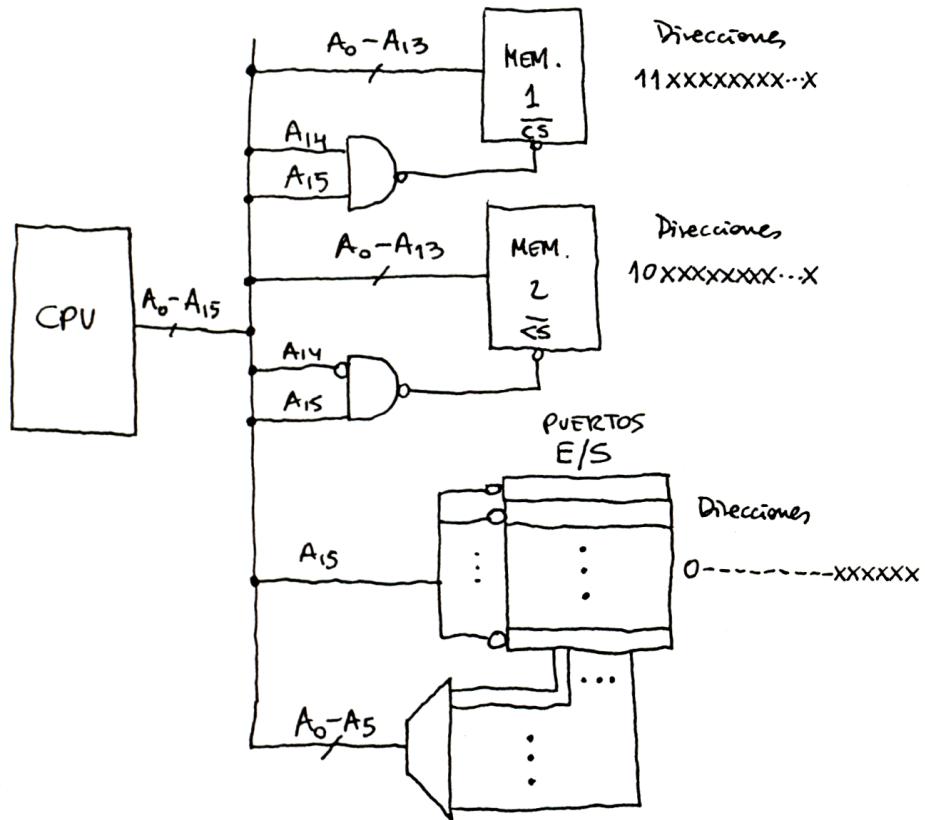
E/S independiente



E/S independiente frente a E/S en memoria

Ejemplo: Supongamos:

- 2 módulos de mem. de 16 KB (14 bits para direccionamiento de bytes dentro de un módulo).
- 64 puertos de E/S.
- bus de direcciones de 16 bits.

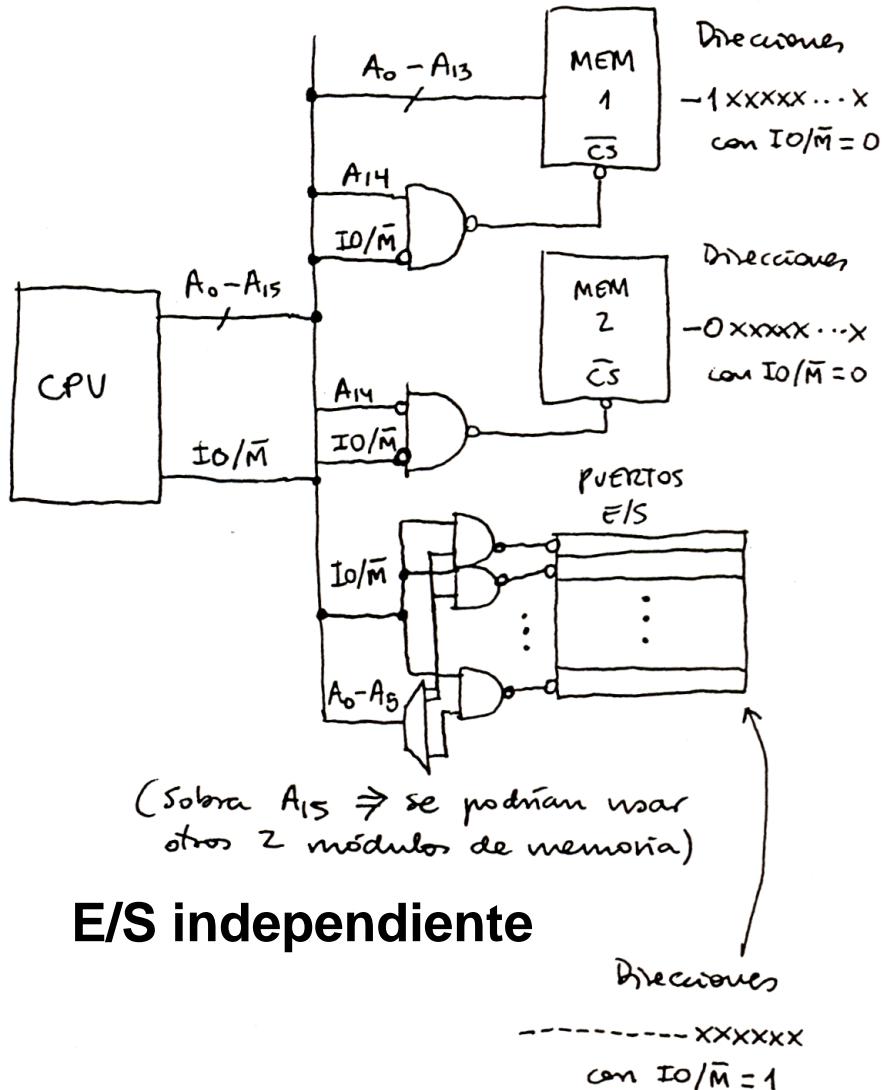


E/S mapeada en memoria

E/S independiente frente a E/S en memoria

Ejemplo: Supongamos:

- 2 módulos de mem. de 16 KB (14 bits para direccionamiento de bytes dentro de un módulo).
- 64 puertos de E/S.
- bus de direcciones de 16 bits.

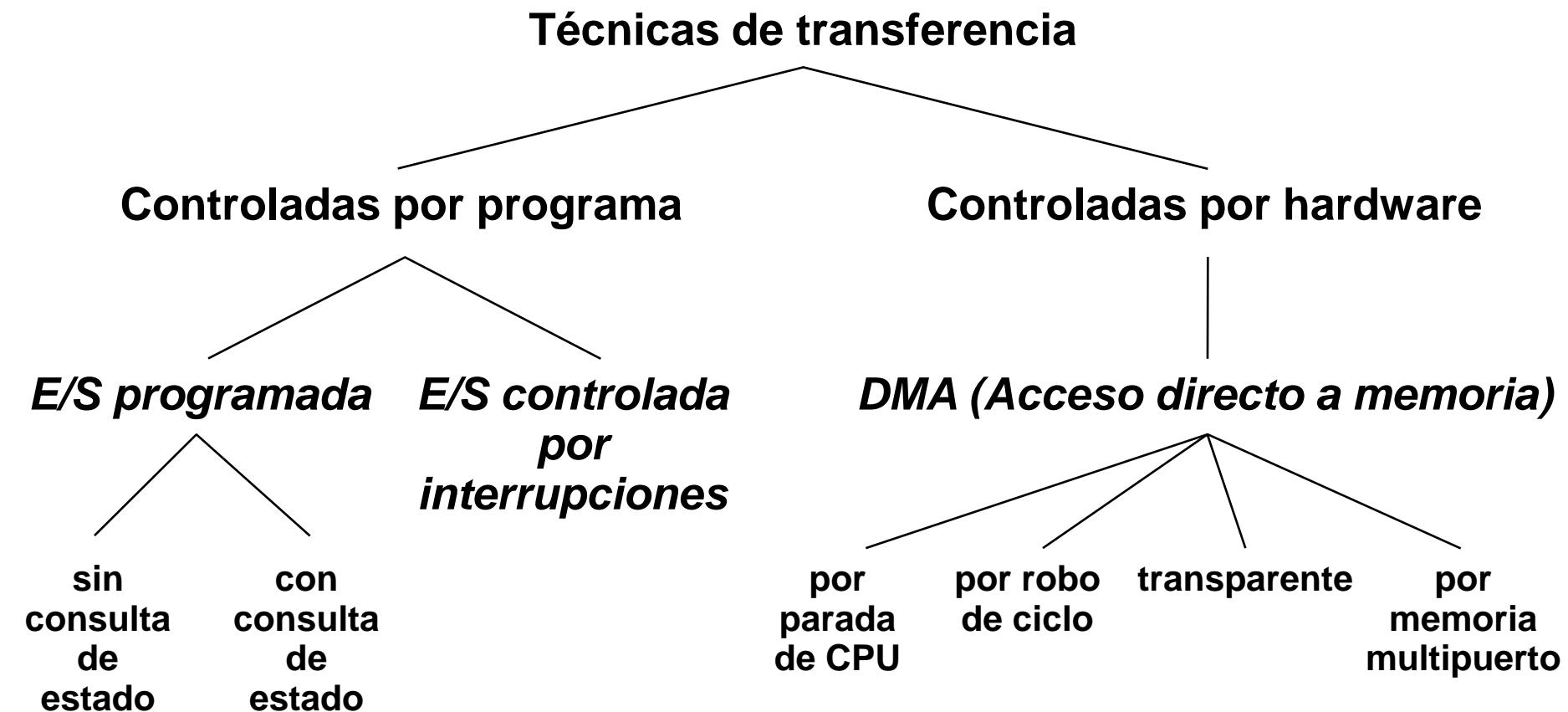


Técnicas de E/S

■ Procedimientos para realizar el intercambio de información entre procesador, memoria y E/S.

- Para realizar la comunicación con los periféricos, consideramos una versión simplificada de los controladores de periféricos:
 - Uno o varios registros (**puertos** o puertas –*ports*– de E/S):
 - datos
 - control
 - estado
 - Lógica de control
 - interpreta las señales de control/estado emitidas por el procesador
 - genera las señales de control/estado que el procesador exige

Técnicas de E/S



Técnicas de E/S

- *E/S programada*

- El procesador participa activamente ejecutando instrucciones en todas las fases de una operación de E/S: inicialización, transferencia de datos y terminación.

- *E/S controlada por interrupciones*

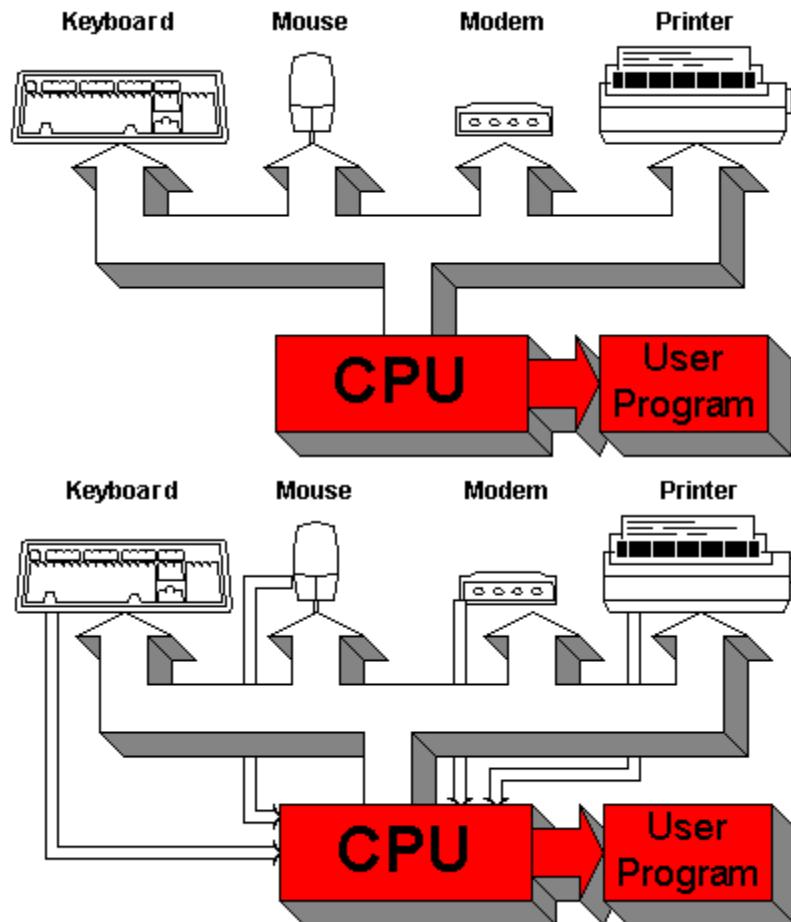
- Los dispositivos de E/S se conectan al procesador a través de líneas de petición de interrupción, que se activan cuando los dispositivos requieren los servicios del procesador.

- En respuesta, el procesador suspende la ejecución del programa en curso y ejecuta un programa de gestión de interrupción para transmitir datos con el dispositivo.

- Como en la E/S programada, los pasos de transferencia de datos están bajo el control directo de programas de control.

- *E/S mediante DMA*

- Requiere la presencia de un controlador DMA, que puede actuar como controlador del bus y supervisar las transferencia de datos entre MP y uno o más dispositivos de E/S, sin intervención directa del procesador salvo en la inicialización.



Técnicas de E/S

Método de control de E/S

<i>Función o parámetro</i>	E/S programada	E/S controlada por interrupciones	Acceso directo a memoria
<i>Comienzo de las operaciones de E/S</i>	La CPU lee y comprueba el estado de los dispositivos de E/S (en el caso de consulta de estado).	El dispositivo de E/S envía una petición de interrupción a la CPU. Ésta transfiere el control a una rutina de servicio P.	(Para cada bloque) El dispositivo de E/S envía una petición de interrupción a la CPU. La CPU transfiere el control a la rutina de servicio P'. P' inicializa el control de DMA.
<i>Transferencia de datos de E/S</i>		La CPU ejecuta un programa de transferencia de datos.	El controlador de DMA transfiere bloques de datos por el bus del sistema.
<i>Finalización de las operaciones de E/S</i>		Fin de ejecución del programa de transferencia de datos	(Para cada bloque) La palabra-contador DMA llega a 0. El controlador de DMA envía una petición de interrupción a la CPU.
<i>Complejidad del circuito de interfaz de E/S</i>	La menor	Baja	Moderada
<i>Velocidad de respuesta a una petición de transferencia de datos por el dispositivo de E/S</i>	Lenta	Rápida	La más rápida
<i>Máxima velocidad de transferencia de E/S</i>	Moderada	Moderada	Alta

Segmentación de cauce

- Funciones del sistema de E/S. Interfaces de E/S
- E/S programada
- Interrupciones
- DMA (Acceso directo a memoria)
- Estructuras de bus básicas
- Especificación de un bus. Transferencias. Temporización. Arbitraje
- Ejemplos y estándares

Concepto de E/S programada

- Todos los pasos de una operación de E/S requieren la ejecución de instrucciones por parte del procesador
- La transferencia de un byte se realiza mediante la ejecución de una instrucción de E/S: **instrucción de transferencia de datos en la que:**
 - Un operando es un registro del controlador del periférico (**puerto**).
 - El otro operando es un **registro del procesador** (o posición mem.).
- Al decodificar la instrucción de E/S, la UC del procesador envía al exterior información de:
 - dirección (sobre qué periférico se realiza la transferencia)
 - tipo de operación (lectura / escritura)
 - temporización / sincronización

...y envía el dato o se dispone a recibirlo.

Concepto de E/S programada

- **SALIDA:**
 - El procesador ejecuta una **instrucción de carga** de una palabra de memoria a un registro.
 - El procesador ejecuta una **instrucción de salida** que transfiere el dato desde el procesador a un puerto de salida.
- **ENTRADA:**
 - El procesador ejecuta una **instrucción de entrada**.
 - El procesador ejecuta una **instrucción de almacenamiento**.
- **Además se pueden necesitar instrucciones adicionales, por ejemplo para actualizar el número de palabras transferidas.**

E/S programada sin y con consulta de estado

■ Sin consulta de estado, o incondicional

- El procesador decide en qué momento se realiza la transferencia.
- El dispositivo externo debe
 - estar siempre dispuesto a recibir datos (salida)
 - o debe tener siempre datos disponibles (entrada)
- Ejemplos:
 - Salida a un display de 7 segmentos
 - Entrada desde un joystick.

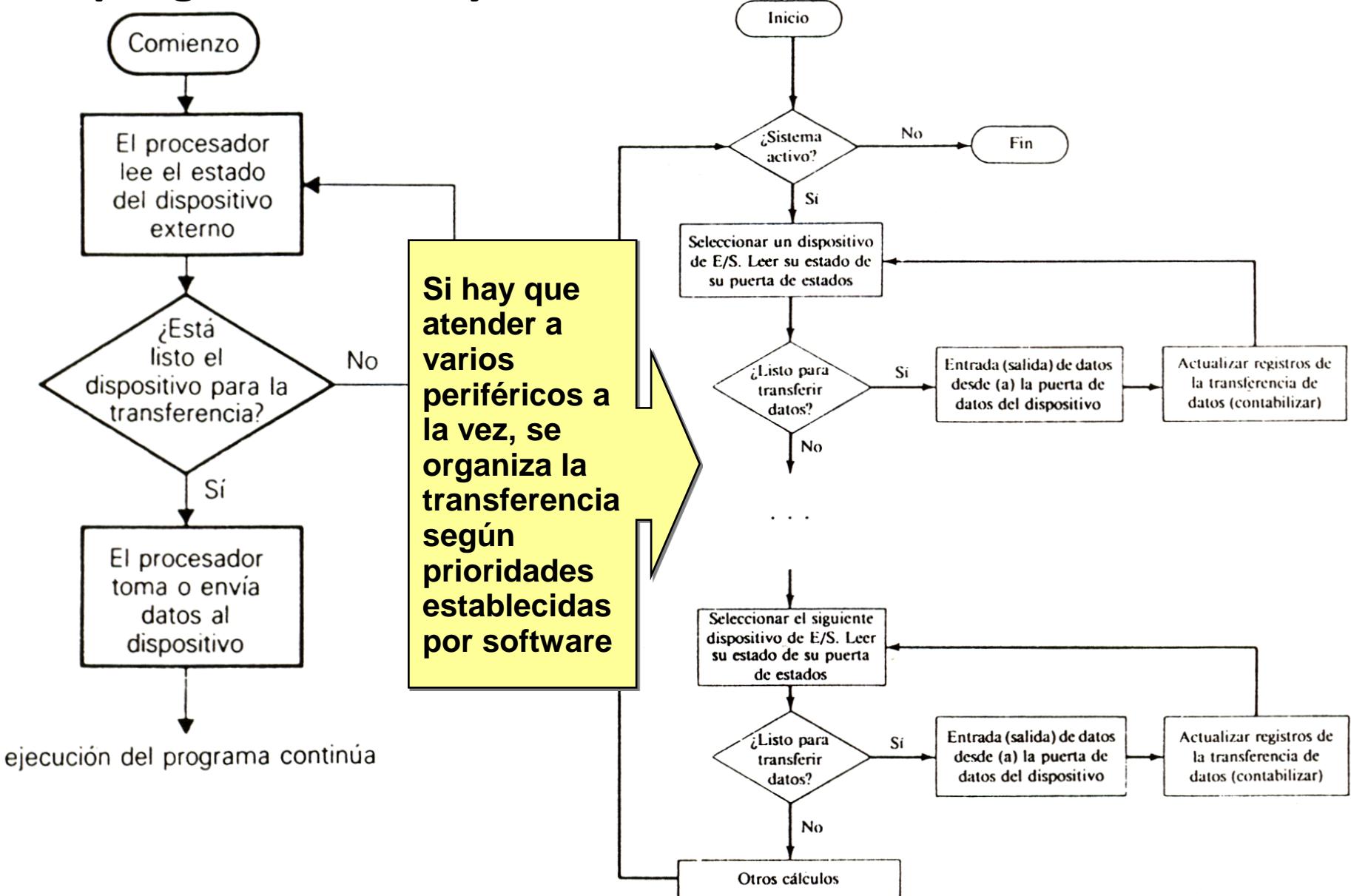
E/S programada sin y con consulta de estado

■ Con consulta de estado, condicional, o con escrutinio

- El procesador pregunta al periférico (a través de la interfaz) si está preparado o no para la transferencia.
- Además de los puertos de E/S, la interfaz tiene un registro de estado con información sobre:
 - Dato listo (si se usa como entrada)
 - Periférico libre (si se usa como salida)

¡Ojo! concepto parecido al *handshaking*, pero por software

E/S programada sin y con consulta de estado



Ej. de programa de E/S programada con consulta de estado

- Programa (para el microprocesador Motorola 6800) que consulta un dispositivo de entrada y transfiere un bloque de datos a la MP.
- Puertos de la interfaz:
 - DATA: Datos a leer
 - STATUS: Estado del dispositivo (bit más signif. == 1 ⇒ listo)
- La E/S es mapeada en memoria.
- El número de palabras a transmitir se sitúa en el registro X del 6800, y se decrementa después de cada transferencia de un dato.
- Un dato de E/S se transfiere desde el puerto de E/S al acumulador A, y después a la pila, que sirve como área de memoria intermedia de E/S.



Ej.de programa de E/S programada con consulta de estado

```
lds    IOBuf          ; Inic. puntero de pila en
                  ; ...área de mem. intermedia
ldx    Count          ; Inicializar X como contador
...
poll  lda    a Status   ; Entrada de palab. de estado
      bpl  next          ; Comprobar signo (N) de A
                  ; ...si N != 1 ==> consultar
                  ; ...otro disp. de E/S
      lda    a Data        ; Entrada de dato en A
      psh    a              ; Transferir dato a pila
      dex              ; Decrementar contador X
      bne    poll          ; Continuar consultando disp.
                  ; ...en curso si X != 0
next   ...            ; Proceder con la siguiente
                  ; ...tarea
```

Ejemplo de circuito integrado de interfaz paralela: 8255

■ Interfaz de periféricos programable 8255

(Programmable Peripheral Interface, PPI)



PA3	1	PA4	40
PA2	2	39	PA5
PA1	3	38	PA6
PA0	4	37	PA6
RD	5	36	WR
CS	6	35	RESET
GND	7	34	D0
A1	8	33	D1
A0	9	32	D2
PC7	10	8255A	31
PC6	11		D3
PC5	12		30
PC4	13		D4
PC0	14		29
PC1	15		D5
PC2	16		28
PC3	17		D6
PB0	18		27
PB1	19		D7
PB2	20		26
			Vcc
			25
			PB7
			24
			PB6
			23
			PB5
			22
			PB4
			21
			PB3

■ Permite gestionar tres puertos de E/S de 8 bits.

■ Tres modos de funcionamiento:

- E/S programada sin validación:
 - *Modo 0*: E/S básica (programada).
- E/S programada o por interrupciones con validación (handshaking).
 - *Modo 1*: E/S con validación.
 - *Modo 2*: E/S con bus bidireccional, validada.

■ Más información en apéndice

Entrada/salida y buses

- Funciones del sistema de E/S. Interfaces de E/S
- E/S programada
- Interrupciones
- DMA (Acceso directo a memoria)
- Estructuras de bus básicas
- Especificación de un bus. Transferencias. Temporización. Arbitraje
- Ejemplos y estándares

Concepto de interrupción

■ Interrupciones:

excepto en el caso de las interrupciones software

- Bifurcaciones normalmente externas al programa en ejecución, provocadas por muy diversas causas
 - externas (señales que provienen del exterior del procesador), o
 - internas (la interrupción la puede producir el propio procesador)
- ...cuyo objetivo es reclamar la atención del procesador sobre algún acontecimiento o hecho importante
- ...pidiendo que se ejecute un programa específico para tratar dicho acontecimiento,

(el código que se ejecuta en respuesta a una solicitud de interrupción se conoce como rutina de servicio de interrupción o ISR (*Interrupt Service Routine*)).

- ...de manera que el programa en ejecución queda temporalmente suspendido.

Concepto de interrupción

■ Ejemplo:

- Interrupción provocada por un controlador de DMA,
 - ...cuando termina la transmisión de un bloque de datos,
 - ...para poner en conocimiento de los programas que tratan la E/S
 - ...que ha terminado y se encuentra disponible para poder realizar otra operación.

Concepto de interrupción

■ Hay que diferenciar entre:

- Interrupción propiamente dicha:
 - Diálogo de señales necesario para que
 - ...el procesador acepte, de forma correcta, la solicitud de interrupción,
 - ...y salte al programa que se debe ejecutar.
- Tratamiento de la interrupción
 - Ejecución del programa de gestión de la interrupción.
- La frontera entre las dos funciones puede resultar difusa, ya que hay pasos que en algunas soluciones se hacen mediante diálogo de señales y en otras mediante programas.

Secuencia de eventos

Esquema correspondiente a identificación de la fuente de la interrupción por hardware, sin *polling*:

Dispositivo: efectúa una solicitud de interrupción

CPU: informa al dispositivo de que su solicitud ha sido reconocida

Dispositivo: en respuesta, desactiva la señal de solicitud de interrupción

CPU: interrumpe el programa que se estaba ejecutando en ese momento

Pila \leftarrow PC

PC \leftarrow Pila

Se reanuda la ejecución del programa interrumpido

ISR

CPU: desactiva las interrupciones durante la primera instrucción o durante toda la ISR

¿Inhabilitar / habilitar interrupciones?

Guardar en la pila todos los registros que se modifiquen

Cuerpo principal de la ISR (por ej. E/S)

Restaurar todos los registros previamente guardados

Habilitar interrupciones si previamente se habían inhabilitado

Retorno de la ISR

Causas de las interrupciones

■ Clasificación en función de las distintas situaciones que pueden requerir que el procesador sea interrumpido:

1. Fallo del hardware

- Los circuitos efectúan comprobaciones para verificar si funcionan correctamente, y si no es así, generan interrupciones
- Ejemplos:
 - Fallo de alimentación:
 - » Interrupción de la más alta prioridad que guarda registros (puede ser necesario conectar baterías).
 - Errores de paridad de memoria:
 - » Interrupción que reintenta la transferencia varias veces.

Causas de las interrupciones

2. Errores de programa

- Situaciones de error introducidas por el programador
- Ejemplos:
 - Error de desbordamiento (overflow)
 - División por cero
 - » En ambos casos la interrupción cancela la ejecución del programa o transfiere el control a una función del usuario
 - Violación de la protección de memoria
 - » El programa de usuario intenta acceder a una dirección en una parte protegida de la memoria ⇒ interrupción que cancela la ejecución y genera mensaje de error (*segmentation fault*)
 - Ejecución de códigos de operación no válidos

Causas de las interrupciones

3. Condiciones de tiempo real

- Se espera que el ordenador responda rápido a una situación
- Ejemplo:
 - Vigilancia de enfermos
 - » Interrupciones para hacer sonar timbres de alarma ante situaciones de peligro
 - Control de herramientas
 - » Interrupción para parar el equipo y evitar daños en la pieza o la máquina cuando algo falla

4. Entrada / salida

- Para poder hacer uso del procesador mientras un periférico no está listo para realizar la transferencia (E/S por interrupc.)
- ...o durante el tiempo que el perif. realiza la transfer. (DMA)
- ...es necesario que la interfaz del periférico o el controlador de DMA puedan interrumpir al procesador.

Tipos de interrupciones

■ Clasificación en función de la procedencia de la interrupción:

1. Interrupciones externas

- Se inician a petición de dispositivos externos.
- ENMASCARABLES:
 - Se pueden habilitar o inhibir usando instrucciones del tipo EI (*Enable Interrupt*) y DI (*Disable Interrupt*).
 - Ejemplo: **interrupciones de E/S**
- NO ENMASCARABLES:
 - Tienen mayor prioridad que las enmascarables.
 - Ejemplo: para gestionar **fallos del hardware externo** al procesador y **condiciones de tiempo real**.

Tipos de interrupciones

2. Interrupciones internas (o excepciones o traps)

- Se activan de forma interna al procesador mediante condiciones excepcionales, como **errores del programa o fallo del hardware interno.**

3. Interrupciones software

- Las instrucciones de interrupción software se emplean para realizar **llamadas al SO.**
- Son instrucciones más cortas que las de llamada a subrutina y no se necesita que el programa llamador conozca la dirección del SO en memoria.

Determinación de la dirección de la ISR

■ ¿Cómo determinar la dirección de comienzo de la rutina de servicio de interrupción?

- Direcciones fijas (o interrupciones no vectorizadas):
 - La dirección o direcciones se fijan y definen en los circuitos del procesador.
 - Ejemplo: una dirección fija para la rutina de servicio de cada interrupción.
- Interrupciones vectorizadas
 - Este término genérico se refiere a todos los esquemas en los cuales el dispositivo que solicita una interrupción suministra de algún modo la dirección de la rutina de servicio.



Determinación de la dirección de la ISR

- Interrupciones vectorizadas (1):
 - **Direccionamiento absoluto**
 - La interfaz suministra la dirección completa de su ISR
 - **Direccionamiento relativo**
 - La interfaz sólo envía parte de la dirección, que deberá ser completada por el procesador
 - » añadiendo más bits
 - » sumando una cantidad
 - ✓ Se reduce el número de bits que necesita transmitir la interfaz, con lo que se simplifica su diseño
 - ✗ Se limita el número de dispositivos que el procesador puede identificar automáticamente.
 - **Envío de una instrucción de bifurcación completa**
 - en lugar de una simple dirección.

Determinación de la dirección de la ISR

- Interrupciones vectorizadas (2):
 - ⌚ Con los métodos anteriores, la ISR para un dispositivo determinado siempre debe comenzar en la misma localización
 - 😊 Para conseguir mayor flexibilidad:
 - el programador puede almacenar en esta localización una instrucción de salto a la rutina adecuada
 - Esto lo puede realizar de forma automática el mecanismo de gestión de interrupciones



- **Direccionamiento indirecto (interrupciones vectorizadas propiamente dichas)**
 - La dirección enviada por la interfaz es la **posición relativa en una tabla**, residente en MP, que contiene las direc. de las ISR.
 - Cada posición de la tabla se conoce como **vector de interrupción**.

Determinación de la dirección de la ISR

- Para sincronizar la transmisión de la dirección, el procesador envía una señal de control:

- INTA# (*Interrupt Acknowledge*, o aceptación de interrupción)
- equivalente a RD# en lectura

...causando que la fuente de interrupción sitúe la dirección en un bus, siendo entonces leída por el procesador.

- Las señales de dirección se pueden enviar por un bus especial, por el bus de datos, o por el propio bus de direcciones.

Identificación de la fuente de interrup.

Los computadores pueden tener conectados una gran variedad de dispositivos con poder de interrupción



La acción requerida al recibir una interrupción dependerá de la causa de ésta
(de qué dispositivo la produjo)



Es necesario que haya diferentes rutinas de servicio de interrupción, para cada una de las causas



Es necesario identificar la causa o el dispositivo que produjo la interrupción

Identificación de la fuente de interrup.

■ Soluciones:

- Una o múltiples líneas de interrupción con un dispositivo en cada línea
 - Cada fuente de interrupción generará una señal de interrupción en la línea apropiada.
 - La identificación del dispositivo es trivial y la dirección de comienzo de la ISR puede ser fija o vectorizada.
 - Una o múltiples líneas de interrupción, con más de un dispositivo por línea
 - Es normal tener varios dispositivos (fuentes de interrupciones) conectados a la misma línea de interrupción.
 - El dispositivo que solicita la interrupción ha de ser identificado
 - Esto puede realizarse por software o por hardware
- 

Identificación de la fuente de interrup.

- Una o múltiples líneas de interrupción, con más de un dispositivo por línea
 - Identificación de la fuente de interrupción por **software** (técnica de sondeo o “polling”)
 - Se usa cuando la dirección de salto para todos los dispositivos conectados a una misma línea es única y fija.
 - La ISR debe identificar el dispositivo que solicita la interrupción, examinando de uno en uno los dispositivos de la línea, para transferir el control a la ISR del solicitante.
 - ✖ Método lento
 - ✓ Económico desde el punto de vista hardware
 - Identificación de la fuente de interrupción por **hardware**:
 - Se usan interrupciones vectorizadas.
 - ✓ Método rápido
 - ✖ Más costoso desde el punto de vista hardware

Sistemas de prioridad en las interrupciones

■ En un computador con más de un dispositivo con capacidad de interrupción, hay que establecer mecanismos de prioridad que resuelvan los problemas:

- Interrupciones simultáneas

- Cuando se produce una interrupción, ésta no se acepta hasta que la instrucción ejecutada termine. Durante ese breve tiempo pueden haberse generado otras interrupciones, que requieren ser atendidas.
- ¿Qué interrupción se atiende primero?

- Interrupciones anidadas

- Se puede producir una interrupción antes de haber atendido completamente la anterior
- ¿Debe terminarse de atender la primera interrupción, o se ha de aceptar y atender inmediatamente la nueva solicitud?

- Inhibición de interrupciones

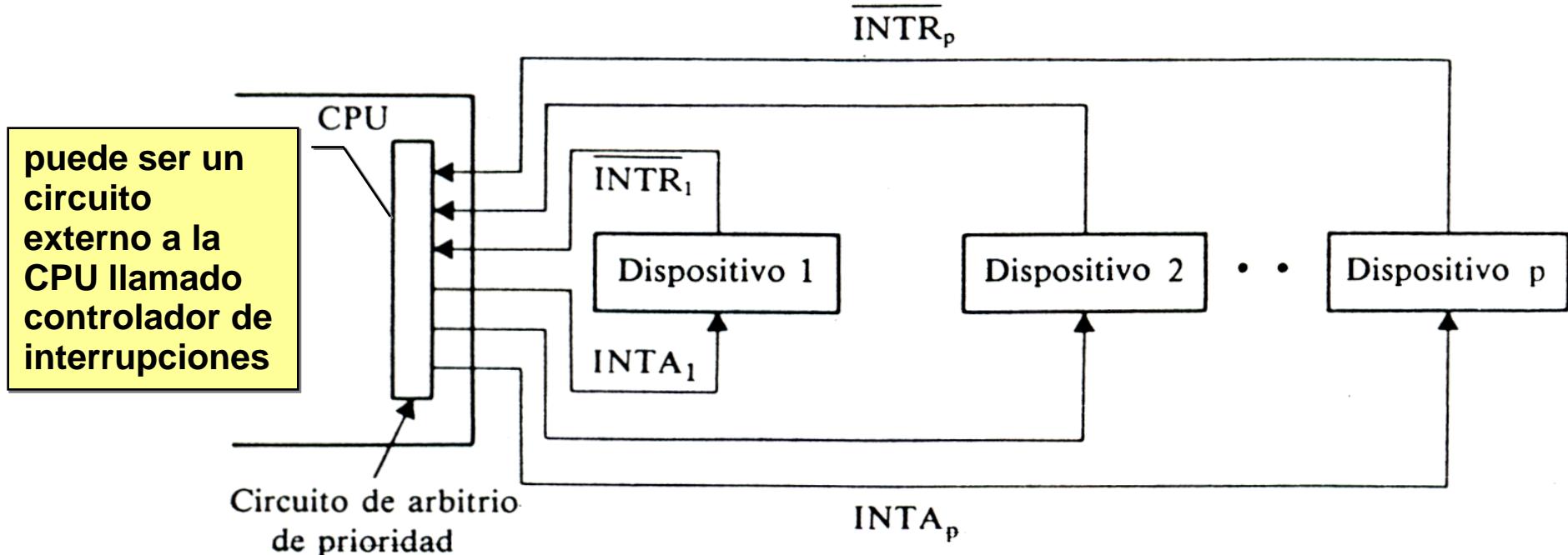
- Mecanismos de prioridad para permitir que cada programa defina los tipos de interrupciones que puede tolerar

Interrupciones simultáneas (I)

- Si llegan solicitudes de interrupción de dos o más dispositivos, el procesador debe disponer de algún medio para que sólo se dé servicio a una solicitud, y el resto se retrase o no se tenga en cuenta.

- Gestión de prioridades centralizada

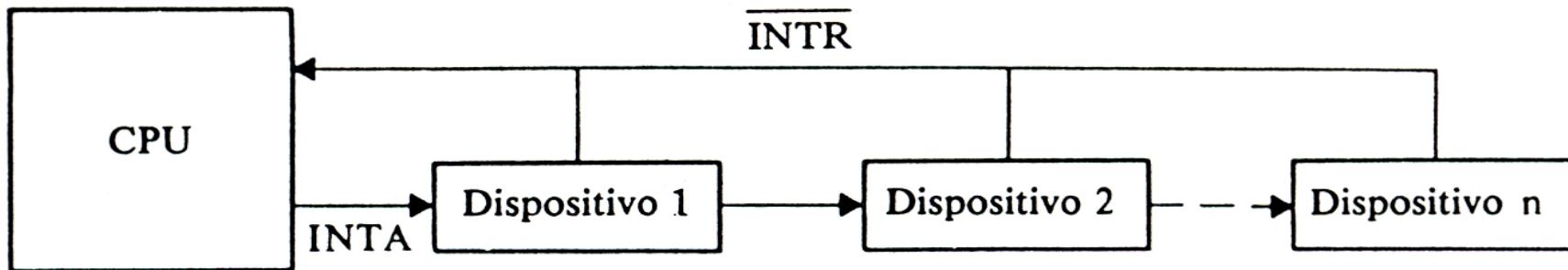
- Cuando hay un solo dispositivo en cada línea de interrupción, la



Interrupciones simultáneas (II)

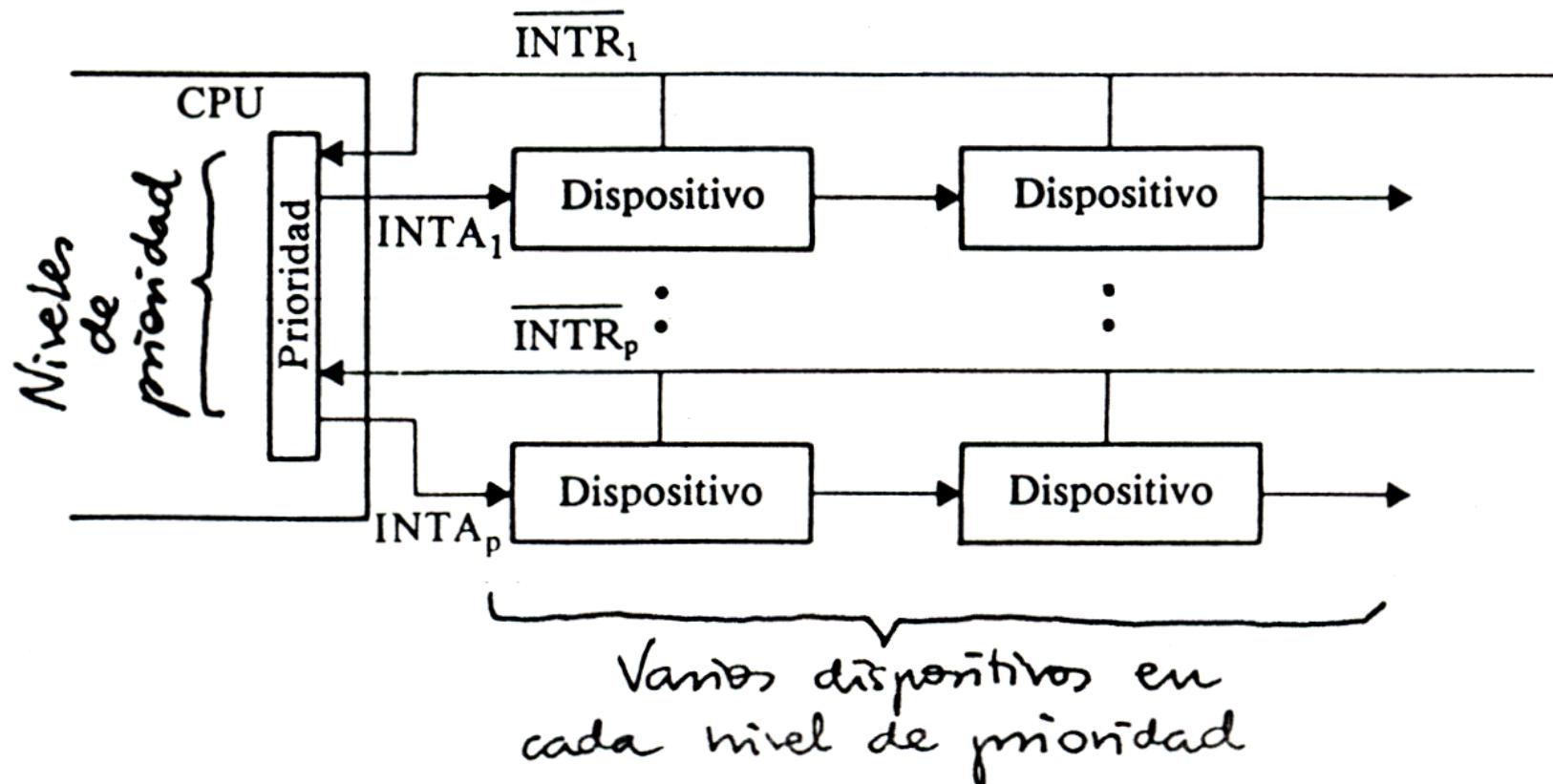
■ Gestión de prioridades distribuida

- Cuando varios dispositivos comparten una única línea de solicitud de interrupción es necesario implantar un mecanismo para asignar prioridades a estos dispositivos:
 - **Técnica de sondeo o “*polling*”**, que como hemos visto se usa para determinar por software el origen de una interrupción.
 - » La prioridad se implanta de forma automática según el orden en el que se escrutan los dispositivos.
 - » Las prioridades se pueden cambiar modificando la ISR para que sondee en un orden diferente.
 - **Técnica de encadenamiento o “*daisy-chain*”**



Interrupciones simultáneas (III)

- Gestión de prioridades híbrida
 - Combinación de esquemas centralizado y distribuido (daisy-chain).

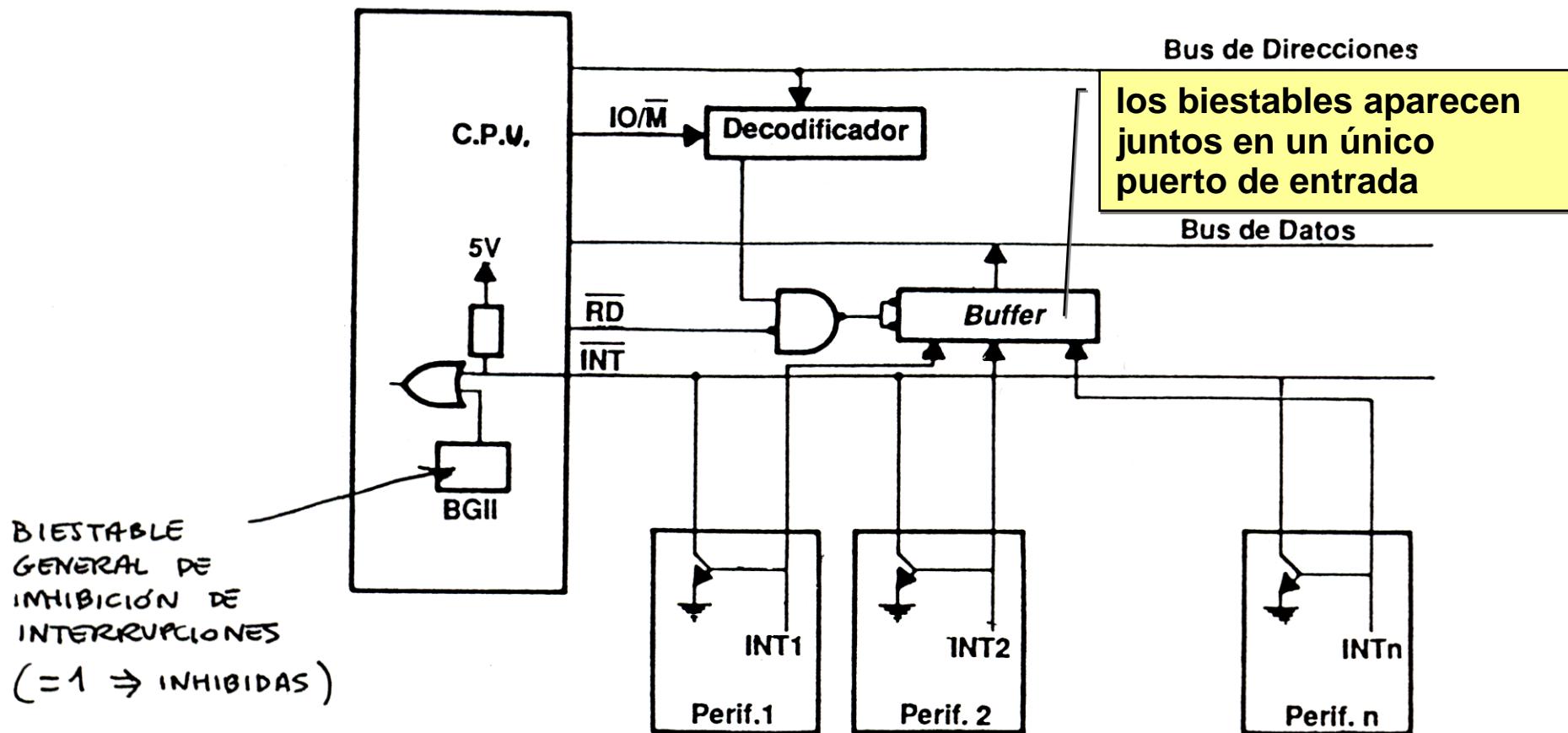


(POLLING I)

- Se usa para
 - Identificar el origen de una interrupción
 - Establecer un mecanismo software de asignación de prioridades a los dispositivos
- En esta solución, el ordenador dispone normalmente de una única línea de interrupción INT#, que sirve para que cualquier dispositivo solicite una interrupción.
- La línea INT# se organiza en colector abierto (OR cableado)
 - Cualquier dispositivo puede poner INT_i=1 para solicitar la interrupción, lo que hace que INT# se active (se ponga a 0).
- La ISR está en una posición de memoria fija y se encarga de identificar cuál es el dispositivo que interrumpió, comprobando el valor de los biestables de interrupción de los dispositivos, que estarán a 1 para aquellos dispositivos que solicitan interrupción.

(POLLING II)

- La asignación de prioridades se hace por el orden en el que la ISR analiza los biestables de interrupción de los periféricos.

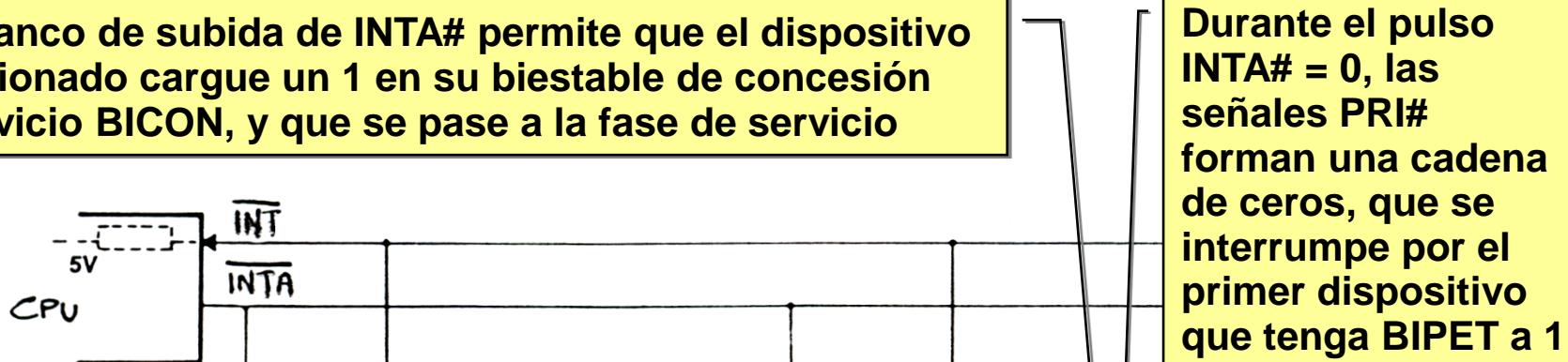


(DAISY-CHAIN I)

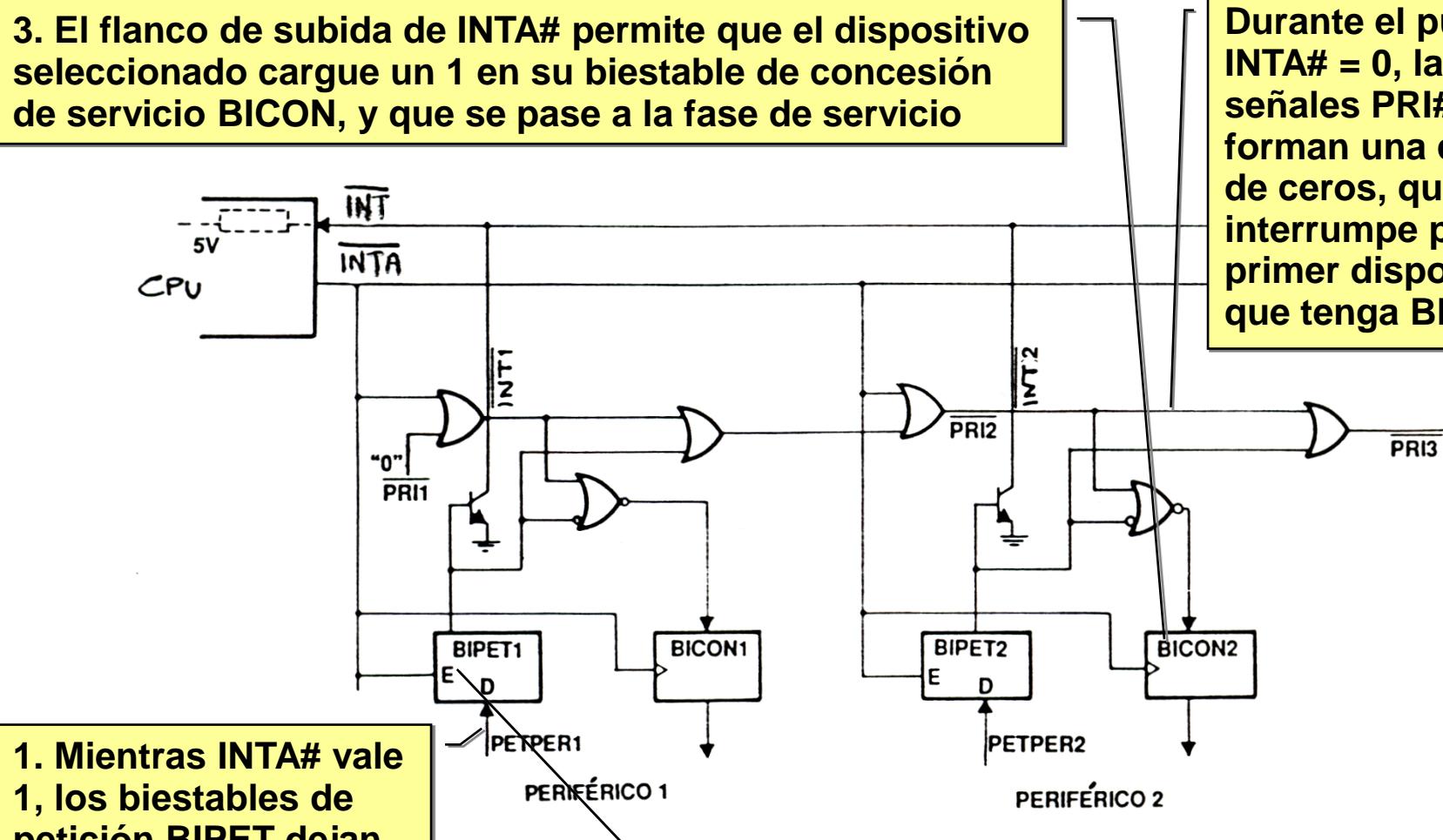
- Se usa para establecer un mecanismo hardware de asignación de prioridades a los dispositivos.
- Se basa en dos señales comunes a todos los peticionarios y a la CPU:
 - INT#: Petición de interrupción
 - INTA#: Concesión o aceptación de interrupción
- Los peticionarios se conectan a INT# en colector abierto.
 - Esto permite que uno o varios dispositivos soliciten simultáneamente la interrupción, poniendo un 0 en INT#
- La señal INTA# sirve, a modo de testigo, para que uno solo de los peticionarios sea atendido.
 - Es un pulso que recorre en serie, uno tras otro, los dispositivos.
 - Es tomado y eliminado por el primero que desea ser atendido.

(DAISY-CHAIN II)

3. El flanco de subida de INTA# permite que el dispositivo seleccionado cargue un 1 en su biestable de concesión de servicio BICON, y que se pase a la fase de servicio



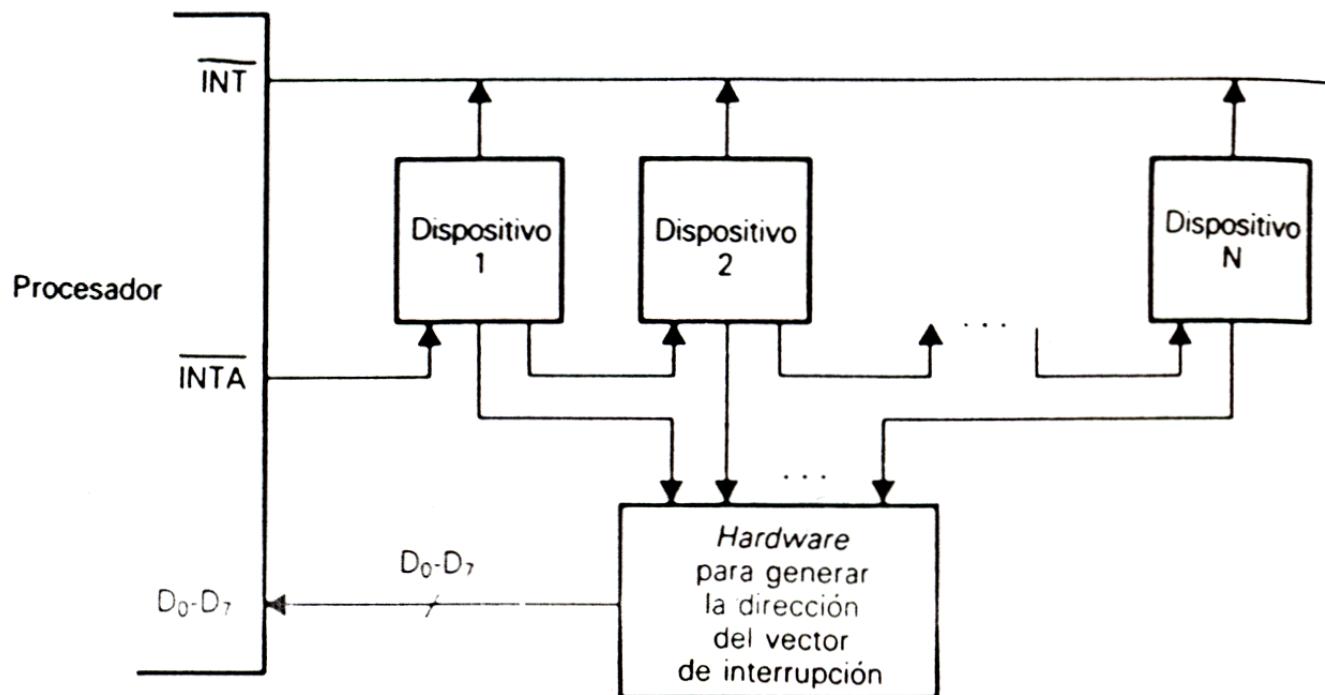
1. Mientras $\text{INTA}\#$ vale 1, los biestables de petición BIPET dejan pasar las peticiones internas PETPER de sus respectivos periféricos



2. Cuando llega un pulso de concesión $\text{INTA}\# = 0$, se congelan los biestables BIPET , de forma que la situación de solicitud de los periféricos no pueda variar

(DAISY-CHAIN III)

- Identificación de la fuente de interrupción:
 - Alternativa 1: la ISR puede determinar qué dispositivo interrumpió leyendo los biestables BICON.
 - Alternativa 2: se pueden usar interrupciones vectorizadas



Interrupciones anidadas (I)

■ En muchas ocasiones se produce una interrupción mientras se está atendiendo otra. ¿Qué hacer?

- Inhabilitar las interrupciones durante la ejecución de la ISR
 - La ejecución de la ISR, una vez iniciada, siempre continuará hasta su finalización, antes de que la CPU acepte una segunda solicitud de interrupción.
 - Esta solución es válida cuando las ISR sean breves.
 - Dispositivos simples para los cuales el retraso posible en la respuesta a la (segunda) solicitud sea aceptable.
- Permitir que la CPU acepte la segunda solicitud de interrupción durante la ejecución de la ISR, si su prioridad es mayor
 - ya que, para algunos dispositivos, un largo retraso en la respuesta a una solicitud de interrupción los podría llevar a funcionar de forma errónea.

Interrupciones anidadas (II)

- Ejemplo: Reloj de tiempo real
 - Envía solicitudes de interrupción a la CPU a intervalos regulares ⇒ ejecución de breve ISR que incrementa la hora (contadores de memoria).
⇒ Necesidad de que se atienda la interrupción antes de que se produzca de nuevo.
⇒ La CPU debe aceptar la solicitud de interrupción aunque se esté atendiendo a otro dispositivo.
- Durante la ejecución de una ISR se aceptarán solicitudes de interrupción de algunos dispositivos, pero no de otros, según sea su prioridad.
- Cada vez que se acepta una nueva interrupción, el contenido de PC se transfiere a la pila, y una vez atendida se toma de ésta.
 - La pila debe ser lo suficientemente grande para que las interrupciones se puedan anidar a suficiente profundidad.

Inhibición de interrupciones (I)

- Hay situaciones en las que conviene evitar temporalmente que se produzcan interrupciones.
- Tres niveles de desactivación de interrupciones:
 1. Desactivar todas las interrupciones
 - Para ello se puede usar un Biestable General de Inhibición de Interrupciones
 - Si está a 1 (por ej.), el programa no podrá sufrir interrupciones.
 - Se puede cambiar su estado mediante instrucciones EI (Enable Interrupts) y DI (Disable Interrupts).
 - Independientemente del valor de BGII, la CPU puede desactivar las interrupciones automáticamente:
 - Durante la ejecución de la primera instrucción de la ISR
 - Si es una instrucción DI \Rightarrow el programador se asegura de que no ocurrirán más interrupciones hasta ejecutar EI.
 - Durante toda la ISR
 - No ocurrirán más interrupciones hasta ejecutar EI o retornar de la interrupción.

Inhibición de interrupciones (II)

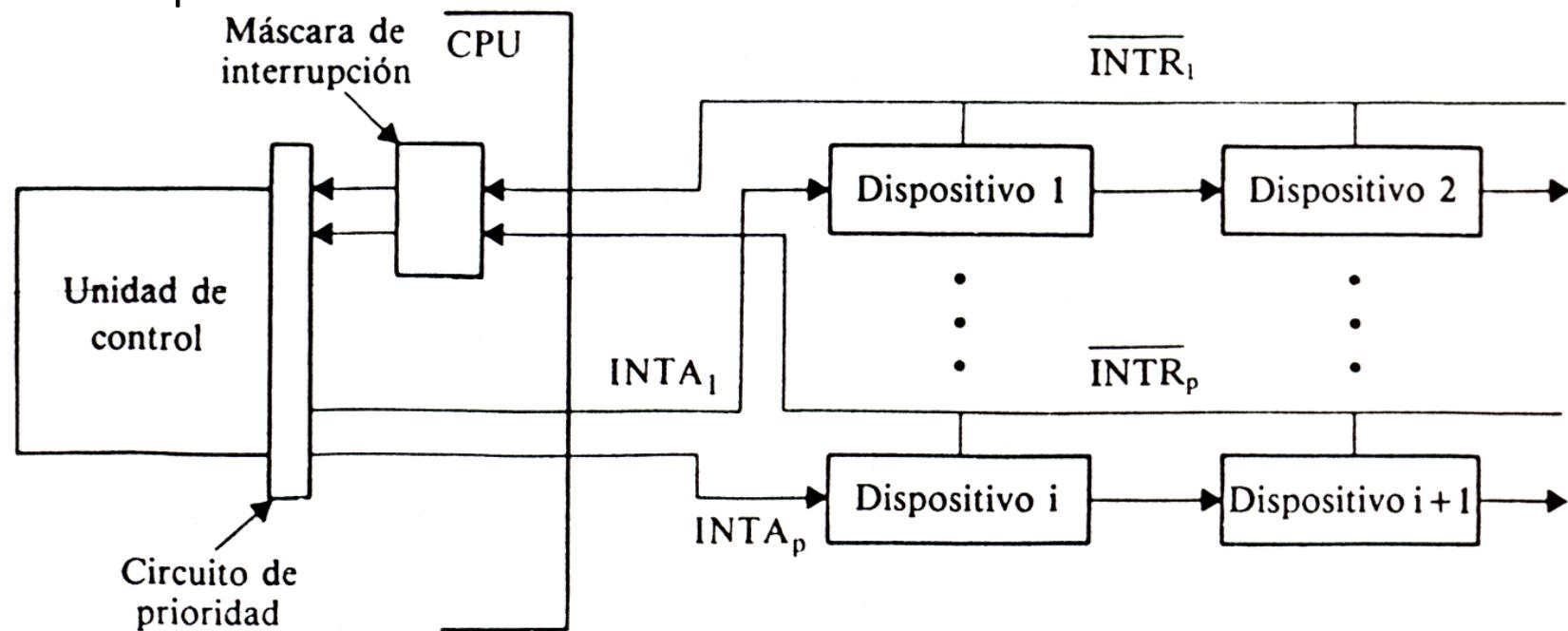
2. Desactivar interrupciones de inferior o igual prioridad

- Se puede tener un registro (o varios bits del registro de estado del procesador) con el valor del menor nivel que puede interrumpir.
- Un decodificador se encarga de desactivar todos los niveles de menor prioridad.
- La CPU tendría así un nivel de prioridad (prioridad del programa que se está ejecutando), y sólo aceptará interrupciones de dispositivos con prioridades mayores que la suya.

Inhibición de interrupciones (III)

3. Desactivar de forma selectiva determinados niveles de interrupción

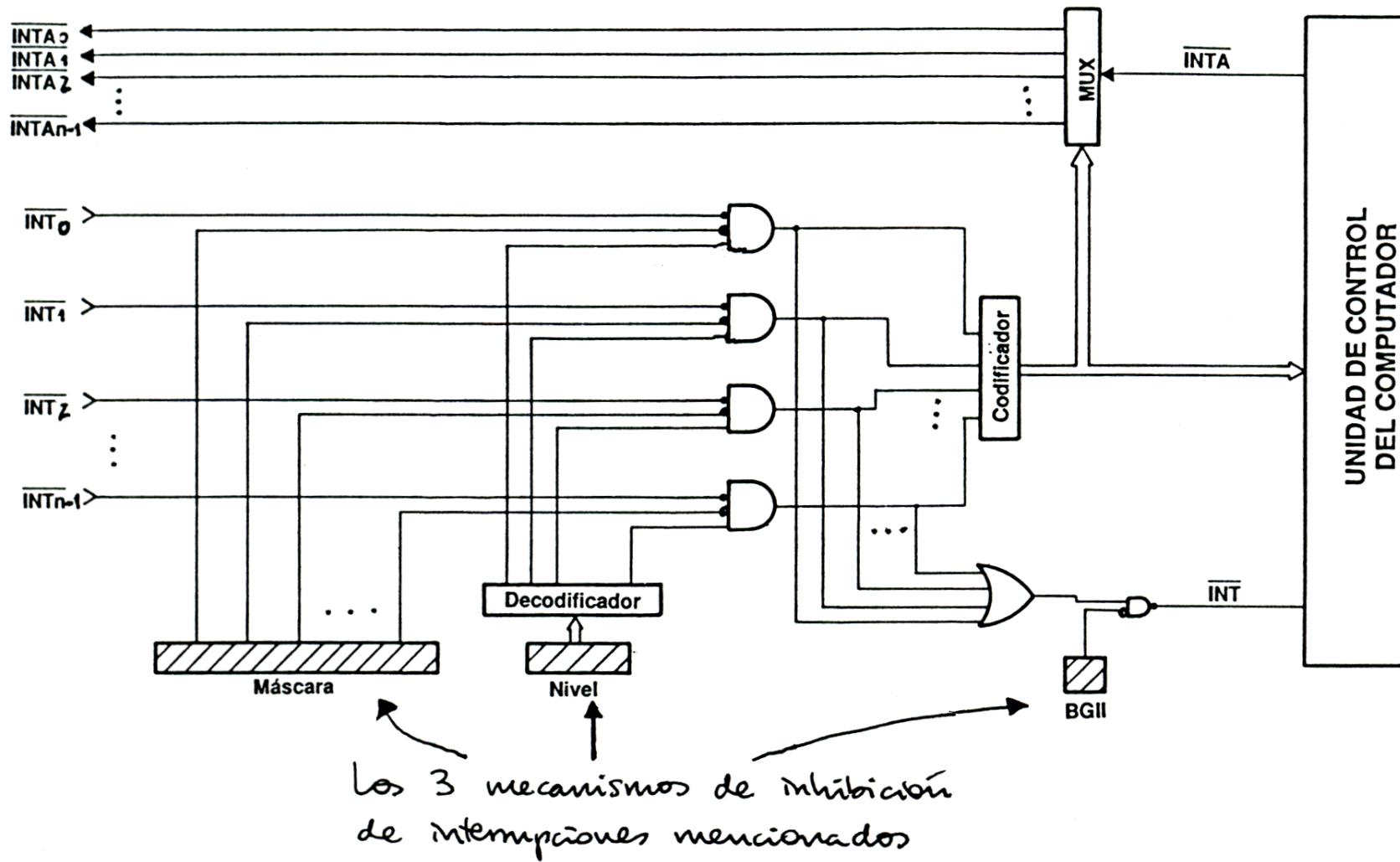
- La señal INTR# de cada nivel se puede enmascarar haciendo la operación AND con el bit correspondiente de un registro máscara de interrupción.
- El programa cargará un 1 en aquellos niveles de interrupción que se deseen inhibir.



Inhibición de interrupciones (IV)

Ejemplo:

n
NIVELES
DE
PRIORIDAD



Los 3 mecanismos de inhibición
de interrupciones mencionados

Inhibición de interrupciones (V)

- Las señales INT_i# deben atravesar un doble filtro:
 - Su correspondiente bit de máscara ($=1 \Rightarrow$ nivel inhibido)
 - La condición de que $i \leq \text{Nivel}$
- El decodificador convierte el contenido del registro de nivel en los correspondientes ceros y unos en cada puerta AND para filtrar las peticiones de interrupción.
- La señal INT# es la que realmente produce la interrupción.
 - Se genera si alguna señal INT_i# consigue atravesar su puerta AND correspondiente, y además el biestable general de inhibición de interrupciones está a 0.
- El codificador genera el valor i de la señal INT_i# de mayor nivel que atraviesa el filtro, para:
 - controlar el multiplexor que encamina la señal INTA# a su nivel correspondiente
 - uso interno de la unidad de control, indicándole cual es el nivel a atender (funcionando como un vector de interrupción)

Ejemplo de ISR

■ ISR “outwd” para 8080:

- Escribe bytes de una posición fija de memoria (data) en un dispositivo de salida (puerto 9).
- Otras interrupciones son inhabilitadas durante la ejecución de “outwd” (lo hace el 8080 automáticamente).

```
outwd:push psw      ; Salvar A e indicadores en la pila
      lda  data    ; A <-- M[data]
      out 9       ; Puerto 9 <-- A
      pop psw      ; Restaurar A e indicadores
      ei           ; Habilitar sistema de interrup.
      ret          ; Retornar a programa interrumpido
```

El sistema de interrupciones del 8080 no es habilitado por EI hasta después de ejecutada la instrucción que sigue a EI (RET en este caso). Esto asegura que la ISR se completa antes de que la CPU entre en otro ciclo de interrupción.

Ej. de circuito controlador de interrupciones: 8259

La mayoría de las familias de microprocesadores contienen circuitos especiales denominados “controladores de interrupciones”, que

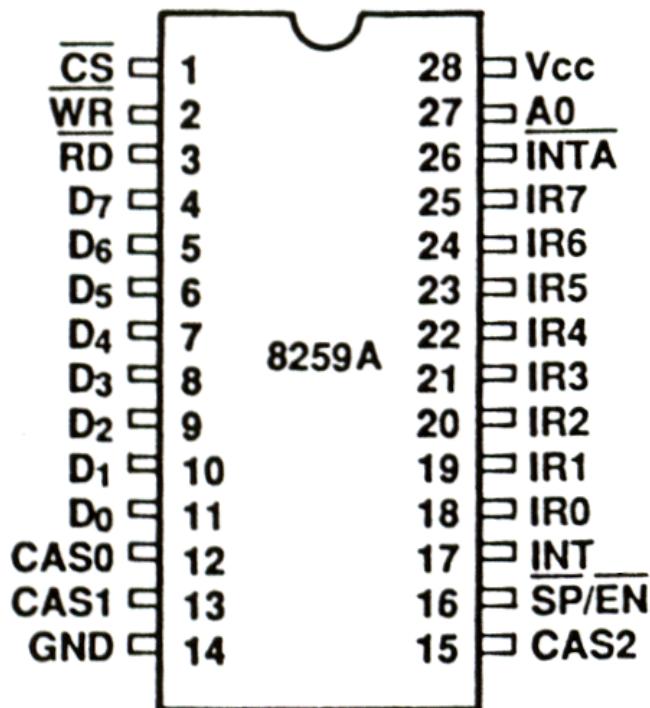
- simplifican las diversas tareas de diseño (prioridades, enmascaramiento,...) asociadas con el control de las interrupciones externas, y
- permiten aumentar el número de líneas de petición de interrupción disponibles.

Ej. de circuito controlador de interrupciones: 8259

■ Programmable Interrupt Controller, PIC

- Permite manejar 8 líneas de interrupciones vectorizadas con prioridad.
- Compatible con las familias 8085 y 8086.
- Prioridades alterables por software.
- Permite enmascaramiento individual de cada línea.
- Es posible conectar 9 controladores para manejar hasta 64 niveles de interrupción.

■ Más información en apéndice



Interrupciones en el PC (modo real)

■ Interrupciones vectorizadas:

- Existen 256 interrupciones posibles (0 a 255 = 0xFF), vectorizadas.
- Tabla de vectores de interrupción:
 - primeros 1024 bytes de memoria (0 a 0x3FF).
- Cada vector de interrupción:
 - doble palabra (32 bits, 4 bytes)
 - dirección de la ISR asociada a esa interrupción

Segmento (CS)
Desplazamiento (IP)

■ Más información en apéndice

Entrada/salida y buses

- Funciones del sistema de E/S. Interfaces de E/S
- E/S programada
- Interrupciones
- DMA (Acceso directo a memoria)
- Estructuras de bus básicas
- Especificación de un bus. Transferencias. Temporización. Arbitraje
- Ejemplos y estándares

Concepto de DMA

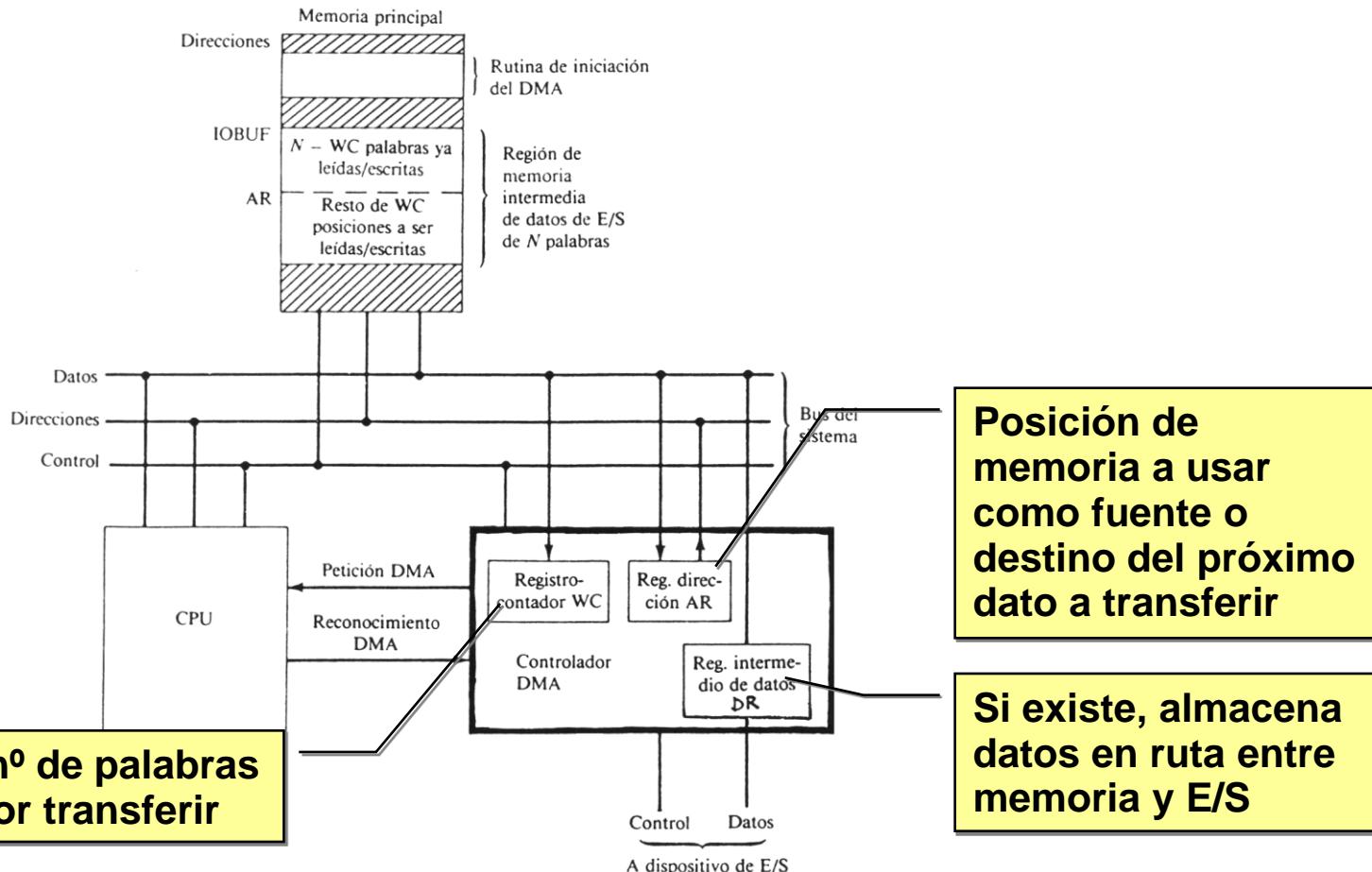
■ DMA (*Direct Memory Access*):

- Técnica que permite realizar transferencias de datos **entre la memoria y los dispositivos de E/S sin intervención directa del procesador.**
 - No se ejecutan instrucciones en el procesador para realizar la transferencia.
- Permite transferencias a la **máxima velocidad permitida por el bus del sistema, la memoria y el periférico.**
 - En sistemas con un único bus, esta velocidad es mucho mayor que la máxima posible con E/S controlada por el procesador:
 - DMA: un ciclo del bus por palabra (pocos ciclos de reloj)
 - E/S controlada por procesador: ejecución de varias instrucciones para cada palabra (muchos ciclos de reloj)
- Se utiliza con **dispositivos rápidos:**
 - Discos, tarjetas gráficas, tarjetas de red, etc.

Controlador de DMA

■ El DMA utiliza un controlador de DMA (DMAC).

- Chip que genera las señales de control y direcciones, actuando como maestro del bus.

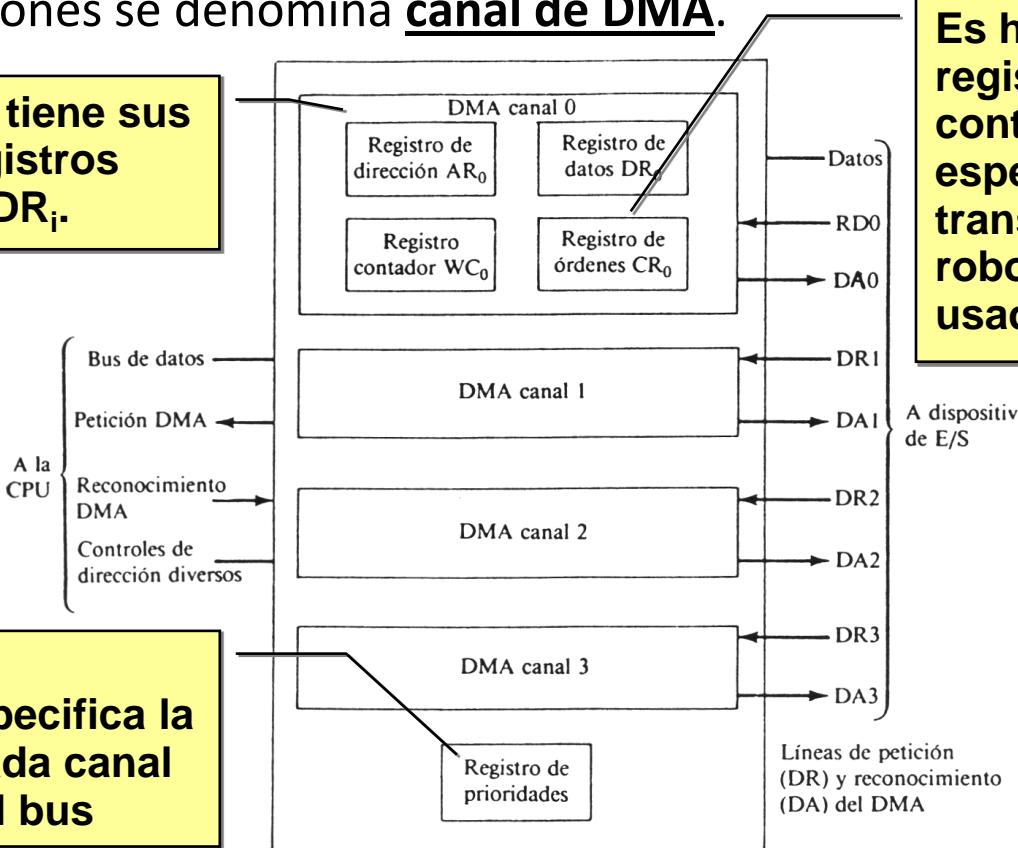


Controlador de DMA

- Existen DMAC que permiten varias operaciones de DMA independientes.

- La lógica de control y los registros asociados con cada una de esas operaciones se denominan **canal de DMA**.

Cada canal tiene sus propios registros AR_i , WC_i y DR_i .



Es habitual que existan registros de órdenes o control (CR_i) que especifican el modo de transferencia (bloques, robo de ciclo, etc.) usado por cada canal.

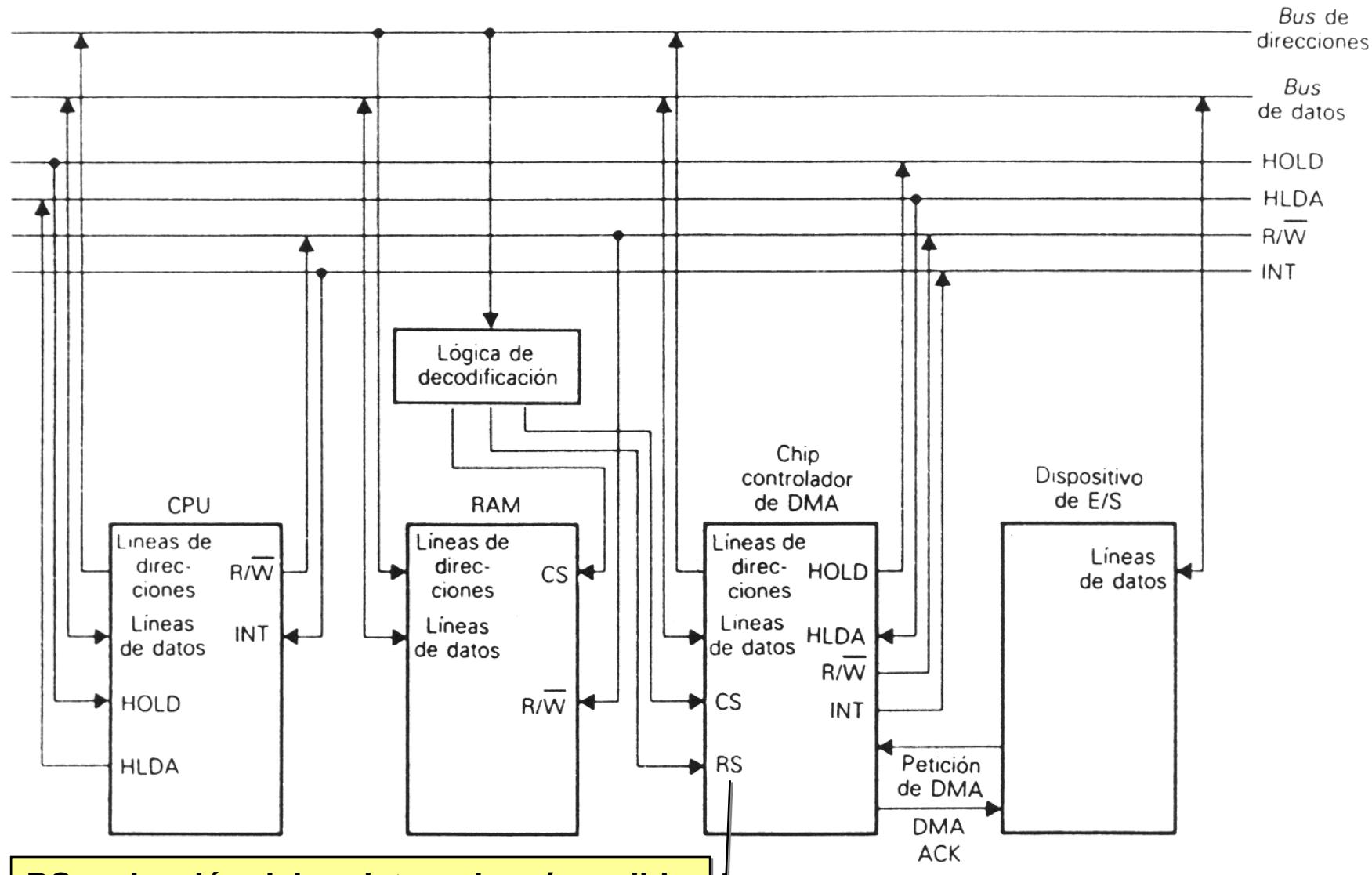
El registro de prioridades especifica la prioridad de cada canal para acceder al bus

Señales de control

■ El DMA y el procesador deben compartir el bus.

- Necesidad de un mecanismo de control de acceso al bus.
- Usualmente se usan dos líneas de control entre DMA y procesador:
 - Petición / solicitud de DMA (*DMA Request*) o del bus (*Bus Request*):
 - HOLD o BSRQ#
 - Reconocimiento / cesión de DMA (*DMA Acknowledge*) o del bus (*Bus Acknowledge*):
 - HLDA o BSAK#
- Cuando el DMA necesita obtener el control del bus, activa HOLD.
- El procesador responde aislando del bus y activando HLDA, para indicar que ha entregado el bus.
- El DMA realiza la transferencia.
- El DMA se desconecta del bus y desactiva HOLD.

Señales de control



RS: selección del registro a leer / escribir

Secuencia de eventos

- Una operación de E/S por DMA se establece ejecutando una corta rutina de inicialización:
 - Varias instr. de salida para asignar valores iniciales a:
 - AR: Dirección de memoria de la región de datos de E/S IOBUF
 - WC: Número N de palabras de datos a transferir
- Una vez inicializado, el DMAC procede a transferir datos entre IOBUF y el dispositivo de E/S:
 - Se realiza una transferencia cuando el dispositivo de E/S solicite una operación de DMA a través de la línea de **petición del DMAC**.
 - Después de cada transferencia, se hace **WC--** y **AR++**
 - **La operación termina cuando $WC = 0 \Rightarrow$** el DMAC (o el periférico) indica la conclusión de la operación enviando al procesador una **petición de interrupción**.

Secuencia de eventos detallada

- 
1. El procesador inicializa el DMA programando AR y WC.
 2. El dispositivo de E/S realiza una petición de DMA al DMA.
 3. El DMA activa la línea de petición de DMA al procesador.
 4. Al final del ciclo del bus en curso, el procesador pone las líneas del bus en alta impedancia y activa la cesión de DMA.
 5. El DMA asume el control del bus.
 6. El DMA responde al dispositivo de E/S con aceptación.
 7. El dispositivo de E/S transmite una nueva palabra de datos al registro intermedio de datos del DMA.
 8. El DMA ejecuta un ciclo de escritura en memoria para transferir el registro intermedio a la posición M[AR].
 9. El DMA decrementa WC e incrementa AR.
 10. El DMA libera el bus y desactiva la línea de petición de DMA.
 11. El DMA compara WC con 0:
 - Si $WC > 0 \Rightarrow$ se repite desde el paso 2.
 - Si $WC = 0 \Rightarrow$ el DMA se detiene y envía una petición de interrupción al procesador.

Métodos de control de DMA

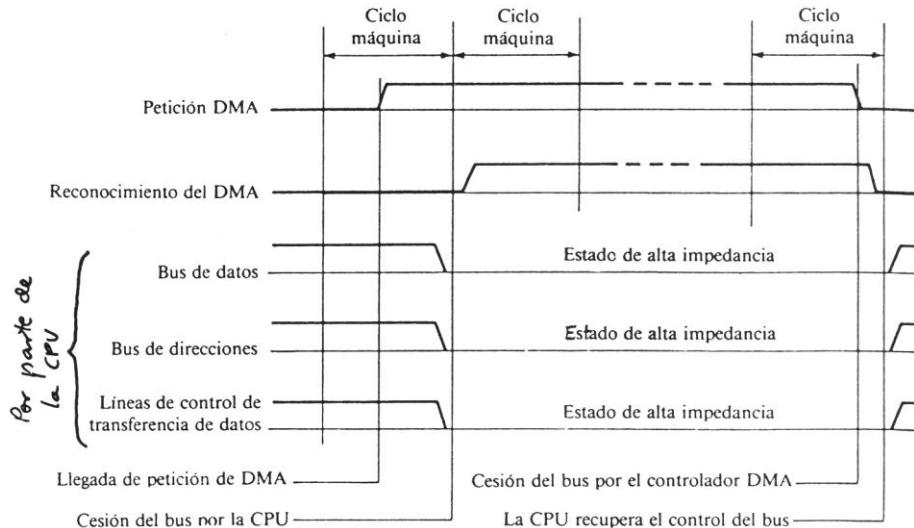
■ Formas de realizar el DMA:

- Robo de ciclo (*Cycle Stealing DMA*)
- Transferencia de bloques o parada de CPU (*Block Transfer DMA*)
- DMA intercalado o transparente (*Interleaved DMA*)
- Memoria multipuerto

Métodos de control de DMA

■ Robo de ciclo (*Cycle Stealing DMA*)

- La transferencia de un bloque se produce palabra a palabra.
- El DMAC “roba” periódicamente uno o varios ciclos máquina al procesador, durante los cuales utiliza el bus. El procesador utiliza el resto de ciclos del bus.
- Cuando el DMAC solicita el bus, debe esperar a que el procesador complete el ciclo máquina en curso.
 - Un robo de ciclo puede aceptarse en mitad de una instrucción (hace que la duración de las instrucciones no sea fija).



Métodos de control de DMA

■ Transferencia de bloques o parada de CPU (*Block Transfer DMA*)

- Se transmite un bloque (secuencia de palabras de datos) en una ráfaga continua.
- El DMAC toma el control del bus durante todo el período que dura la transferencia de datos.
- El procesador no tiene acceso al bus hasta que la transferencia termina, lo que le obliga a esperar períodos de tiempo relativamente grandes.

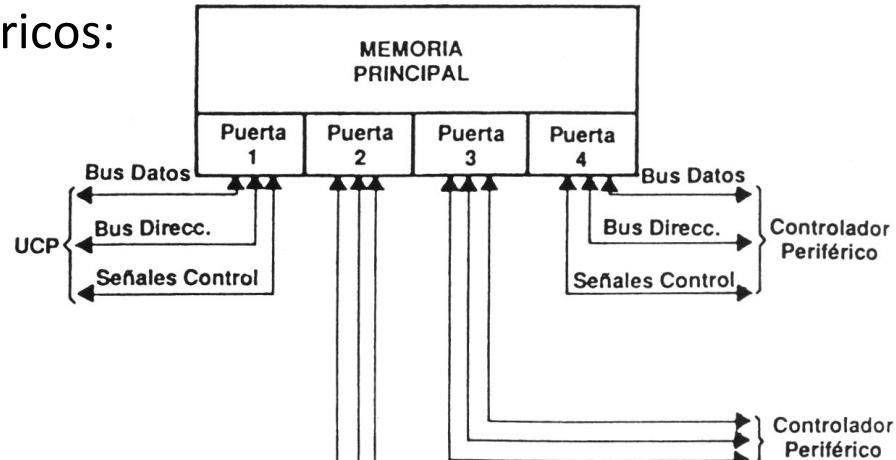
■ DMA intercalado o transparente (*Interleaved DMA*)

- El DMAC toma el bus cuando el procesador no lo utiliza.
 - Por ej., mientras está decodificando un código de operación o efectuando operaciones internas de transferencia de registros.
- Para ello, el procesador debe emitir las señales de control adecuadas.

Métodos de control de DMA

■ Memoria multipuerto

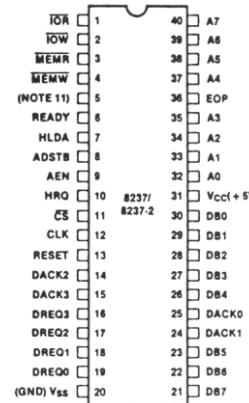
- Existen memorias con varios módulos y varios registros de dirección y de memoria que pueden operar simultáneamente conectados a varios buses, siempre que no direccionen un mismo módulo de memoria.
- Se puede conectar a un puerto el procesador y a los demás puertos varios controladores de periféricos:



- Si hay conflictos de acceso (peticiones simultáneas a un mismo módulo), se concede un acceso y se retardan los demás, según un sistema de prioridades.

Ejemplo de DMAC: 8237

- ***Programmable DMA Controller***
 - 4 canales de DMA independientes
- **Más información en apéndice**



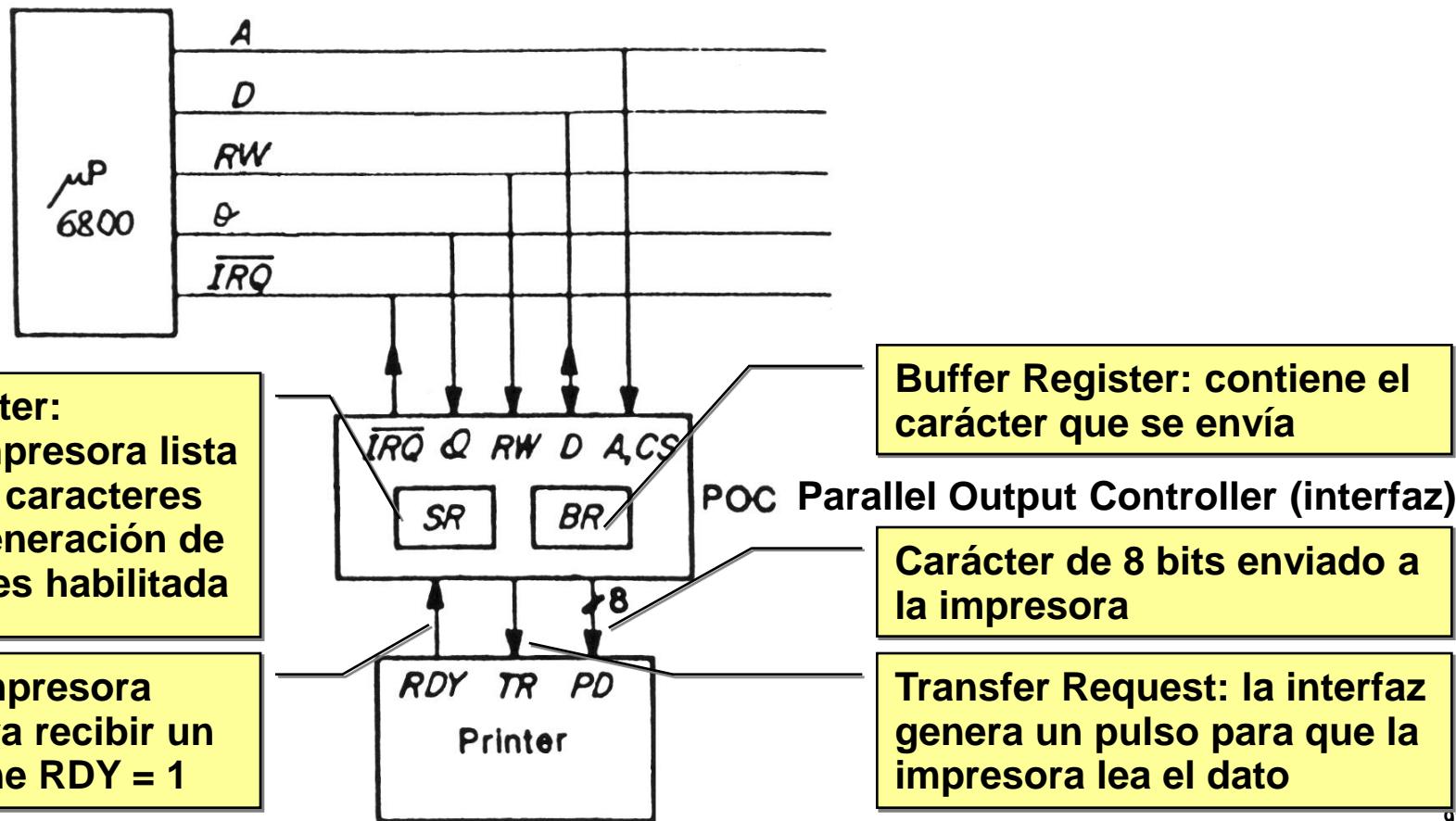
Entrada/salida y buses

- Funciones del sistema de E/S. Interfaces de E/S
- E/S programada
- Interrupciones
- DMA (Acceso directo a memoria)
- Ejemplos de E/S
- Estructuras de bus básicas
- Especificación de un bus. Transferencias. Temporización. Arbitraje
- Ejemplos y estándares

Ejemplo de transferencia de E/S usando las tres técnicas

■ Objetivo:

- Enviar un bloque a la impresora (200 caract., desde dir. 3000h)
- Configuración para E/S programada y por interrupciones:



Ejemplo de transferencia de E/S usando las tres técnicas

■ E/S por programa:

```
; Inicialización  
LDX    #$3000      ; RegIX<-3000h (bloque  
almacenado          ; a partir de la dir. 3000h)  
LDAB   #200        ; RegB<-200 (200 caract.)  
JSR    PRBLQ  
...
```

E/S programada

Ejemplo de transferencia de E/S usando las tres técnicas

; Imprimir bloque

PRBLQ LDAA 0,X
JSR PRCHR

INX

DECB

BNE PRBLQ

RTS

; Imprimir el carácter almacenado en RegA

PRCHR TST PSR
BPL PRCHR
STAA PBR
RTS

E/S programada

; RegA <- M[RegIX+0]

; RegX++ (apuntar a sig. caráct.)

; RegB-- (actualizar contador)

; Si RegB!=0 => enviar otro car.

PSR y PBR son etiquetas para los puertos SR y BR,
mapeados en memoria

Ejemplo de transferencia de E/S usando las tres técnicas

■ E/S por interrupciones:

- Cuando la interfaz está lista para recibir datos (impresora lista), interrumpe al 6800 por su línea IRQ#, forzándole a transferir el control a una ISR

E/S por interrupciones

; Inicialización

```
LDX    #$3000      ; RegIX<-3000h (bloque almacenado
                    ; a partir de la dir. 3000h)
STX    BA          ; BA (Byte Address) <- RegIX
LDAA   #200        ; RegA<-200 (bloque 200 caract.)
STAA   BC          ; BC (Byte Count) <- RegA
LDAA   #$01        ; RegA <- 1
STAA   PSR         ; SR0 <- 1 (interfaz generará
                    ; interrupciones)
```

BA	RMB	2	; Reserve Memory Byte (2 bytes)
BC	RMB	1	; Reserve Memory Byte (1 byte)

Ejemplo de transferencia de E/S usando las tres técnicas

- Puede haber varios dispositivos conectados a IRQ#.
- La ISR comienza chequeando los bits “listo” de cada dispositivo, para ver cuál ha interrumpido. Se supone que la impresora tiene la mayor prioridad en este caso ⇒ es chequeada en primer lugar.

E/S por interrupciones

```

; ISR
ORG $5000
ISR TST PSR           ; Sondar impresora
      BPL POLLTERM    ; Si SR7 = 0 => saltar a Terminal
      BRA PRINT        ; Si SR7 = 1 => saltar a PRINT
POLLTERM
      TST TSR          ; Sondar otros dispositivos,
      . . .             ; comenzando por el terminal
ORG $FFF8
FDB ISR              ; Form Double Byte
                      ; (Vector de interrupción)

```

Cuando tiene lugar una interrupción, el 6800 toma la dirección de la ISR de M[FFF8h]

Ejemplo de transferencia de E/S usando las tres técnicas

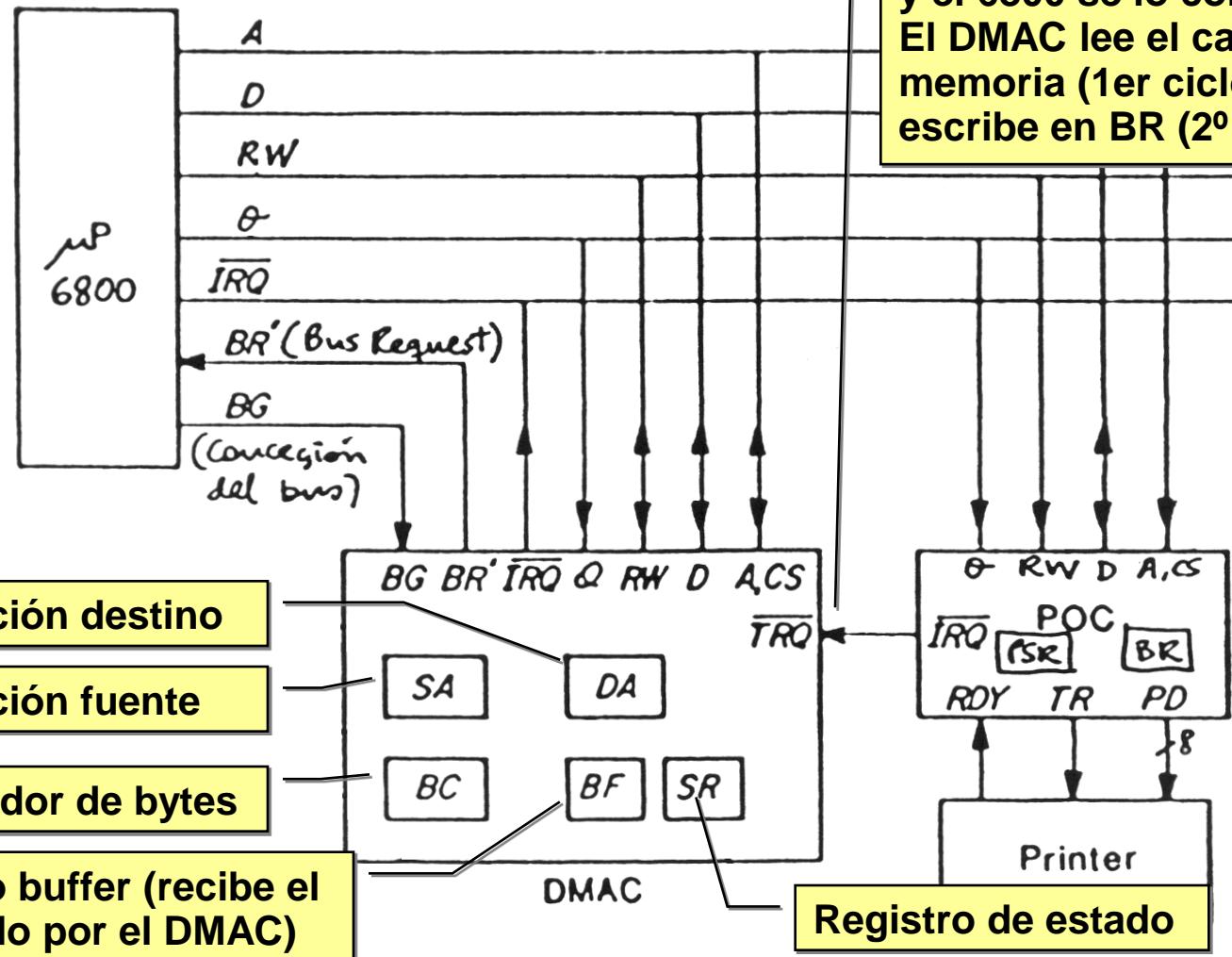
E/S por interrupciones

```
PRINT LDX BA ; RegIX <- Byte Address
               LDAA 0,X ; RegA <- M[RegIX+0] (sig. car.)
               STAA PBR ; Enviar carácter
               INX   ; RegX++ (apuntar a sig. caráct.)
               STX   BA
               DEC   BC ; RegB-- (actualizar contador)
               BNE   RETURN ; Si BC!=0 => retorno subrutina
               CLR   PSR ; SR0 <- 0 si trans. blq. compl.

RETURN
               RTI ; Return From Interrupt
TERM   ... ; Rutinas de otros dispositivos
```

Ejemplo de transferencia de E/S usando las tres técnicas

■ E/S por DMA:



Petición de DMA: cuando se activa TRQ#, el DMAC pide el bus (BR'=1) y el 6800 se lo concede (BG=1). El DMAC lee el carácter de memoria (1er ciclo del bus) y lo escribe en BR (2º ciclo del bus)

Ejemplo de transferencia de E/S usando las tres técnicas

- Escribir un 0 en SR7 inicia la transferencia de un bloque. Al completarse, el DMAC pone a 1 SR7 (bit “listo”).
- SR0 = 1 habilita las interrupciones.
- Cuando SR7 = 1 y SR0 = 1, el DMAC genera una interrupción para indicar al 6800 el fin de la transferencia.
- El tipo de transferencia se selecciona escribiendo en SR[2:1]. SR[2:1] = 10 \Rightarrow transferencia de mem. a periférico.

Ejemplo de transferencia de E/S usando las tres técnicas

; Inicialización del DMA		E/S por DMA
LDX	#\$3000	; SA <- 3000h (bloque en 3000h)
STX	DSA	; 3000h (bloque en 3000h)
LDX	#200	; BC <- 200 caracteres
STX	DBC	; DA <- dirección de BR
LDX	#PBR	; 00000101b (Memoria->periférico,
STAA	DSR	; IRQ permitida)
LDAA	#\$05	; PSR <- Petición del POC
STAA	PSR	; al DMAC permitida

Ejemplo de transferencia de E/S usando las tres técnicas

■ Comparación de tiempos (de procesador):

- E/S por programa
 - Suponiendo una impresora de 200 caracteres/s, el tiempo de procesador será 1 s (**1 000 000 µs**).
- E/S por interrupciones
 - Suponiendo un período de reloj $\theta = 1 \mu\text{s}$ ($f = 1 \text{ MHz}$), una ejecución de la ISR requiere 59 μs .
 - Además el hardware consume 9 ciclos de reloj cada vez que se atiende la interrupción.
 - En total: **13 600 µs** de tiempo de procesador.
- E/S por DMA
 - Se requieren dos ciclos del bus para transferir cada carácter.
 - Suponiendo un ciclo de reloj por ciclo del bus, el 6800 está inactivo 2 μs por cada carácter.
 - Despreciando el tiempo para inicializar el DMA, **400 µs**.

Unidad de control

Estructura de Computadores
9ª Semana

Bibliografía:

- | | |
|---------------|--|
| [HAM03] Cap.7 | Organización de Computadores. Hamacher, Vranesic, Zaki. McGraw-Hill 2003 |
| | Signatura ESIIT/ C.1 HAM org |

Guía de trabajo autónomo (4h/s)

■ Repaso

- Apuntes TOC
 - Tema 6. Análisis y diseño de sistemas secuenciales
 - Tema 7. Sistemas en el nivel transferencia entre registros (RTL)

■ Lectura

- Cap.7 Hamacher

Bibliografía:

[TOC] Temas 6-7

Apuntes Tecnología y Organización de Computadores

[HAM03] Cap.7

Organización de Computadores. Hamacher, Vranesic, Zaki. McGraw-Hill 20
Signatura ESIIT/[C.1 HAM org](#)

Guía de trabajo autónomo (4h/s)

■ Repaso

- Apuntes TOC
 - 6. Análisis y diseño de sistemas secuenciales
 - 6.1 Concepto de sistema secuencial
 - 6.2 Elementos básicos de memoria
 - 6.3 Análisis de un sistema secuencial
 - 6.4 Diseño de un sistema secuencial
 - 6.5 Componentes secuenciales estándar
 - 7. Sistemas en el nivel transferencia entre registros (RTL)*
 - 7.1 Introducción y definiciones generales
 - 7.2 Unidad de procesamiento o camino de datos
 - 7.3 Unidad de control
 - 7.4 Introducción a lenguajes de descripción hardware
 - 7.5 Fases de diseño

Unidad de control

■ Camino de datos

- Unidad de procesamiento y unidad de control
- Unidad de procesamiento con un bus
- Unidad de procesamiento con múltiples buses

■ Unidades de control cableadas y microprogramadas

- Diseño de una UC cableada
- Ejemplo de UC cableada
- Concepto de UC microprogramada

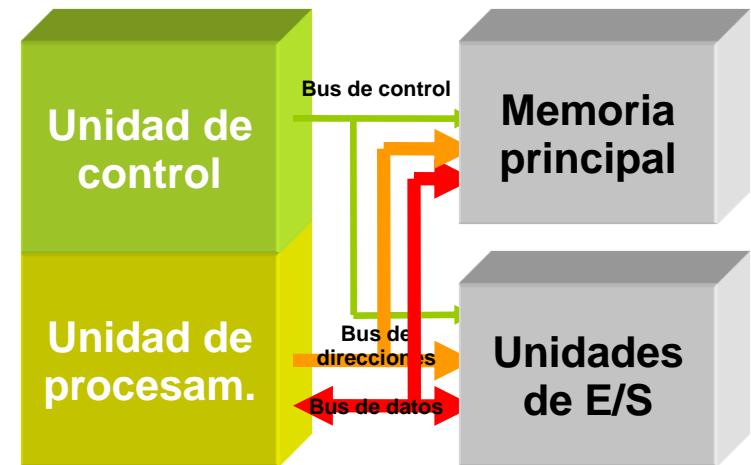
■ Control microprogramado

- Formato de las microinstrucciones
- Nanoprogramación
- Secuenciamiento de microinstrucciones
- Ejemplo de arquitectura microprogramada

Introducción

- Un computador con arquitectura Von Neumann consta de tres bloques fundamentales:

- CPU o procesador
 - Memoria principal
 - (datos + instrucciones)
 - Unidades de E/S
 - (y memoria masiva)
- } unidos mediante buses



- En este tema estudiaremos la CPU, que puede entenderse como una unidad constituida por:
 - Unidad de procesamiento o camino de datos ("datapath")
 - Unidad de control

Unidad de procesamiento

- La unidad de procesamiento comprende elementos hardware como:
 - unidades funcionales (ALU, desplazad., multiplic., etc.)
 - registros
 - registros de uso general (varios GPR)
 - registro de estado
 - contador de programa (PC)
 - puntero de pila (SP)
 - registro de instrucción (IR)
 - registro de dato de memoria (MDR / MBR)
 - registro de dirección de memoria (MAR)
 - multiplexores
 - buses internos
 - ...

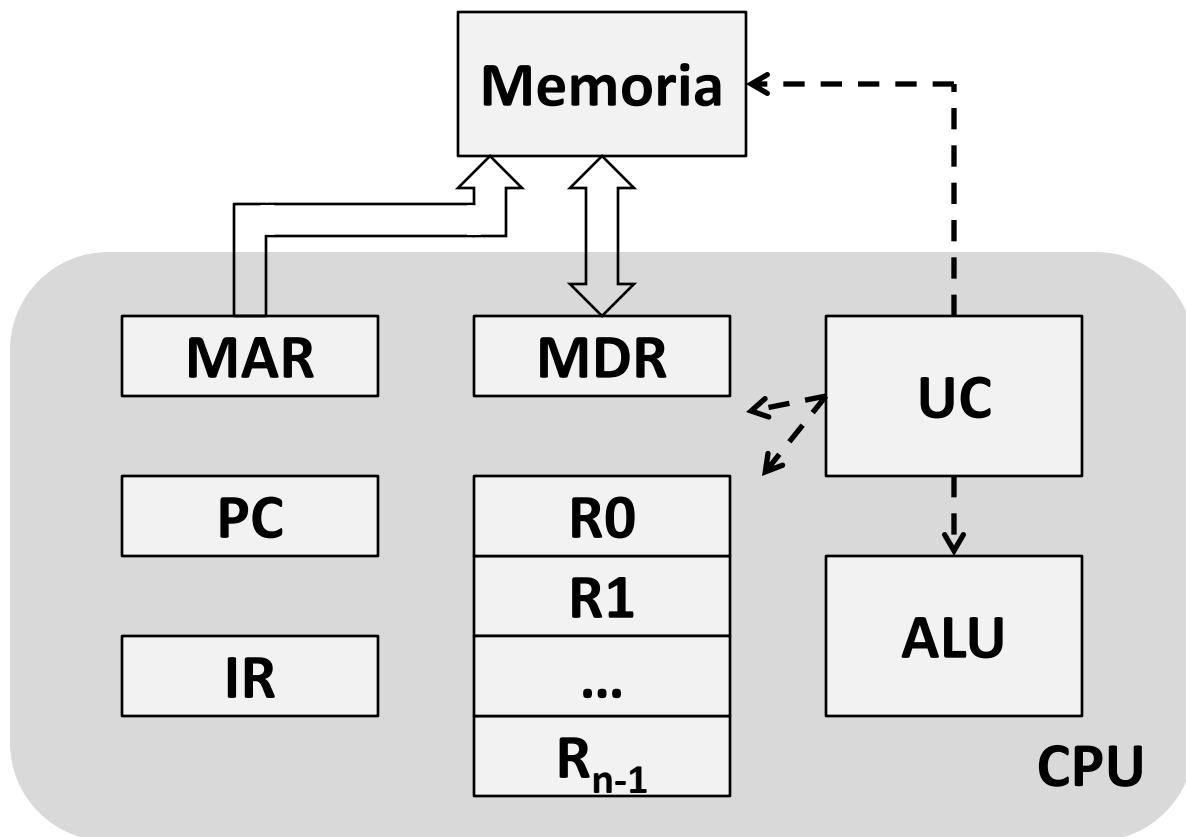
Unidad de control

- **La unidad de control interpreta y controla la ejecución de las instrucciones leídas de la memoria principal, en dos fases:**
 - secuenciamiento de las instrucciones
 - La UC lee de MP la instrucción apuntada por PC, $IR \leftarrow M[PC]^*$
 - determina la dirección de la instrucción siguiente y la carga en PC*
 - ejecución/interpretación de la instrucción en IR
 - La UC reconoce el tipo de instrucción,
 - manda las señales necesarias para tomar los operandos necesarios y dirigirlos a las unidades funcionales adecuadas de la unidad de proceso,
 - manda las señales necesarias para realizar la operación,
 - manda las señales necesarias para enviar los resultados a su destino.

Ej. de ejecución Add A, R0*

■ Pensar tareas realizadas por UC para ejecutar instrucción

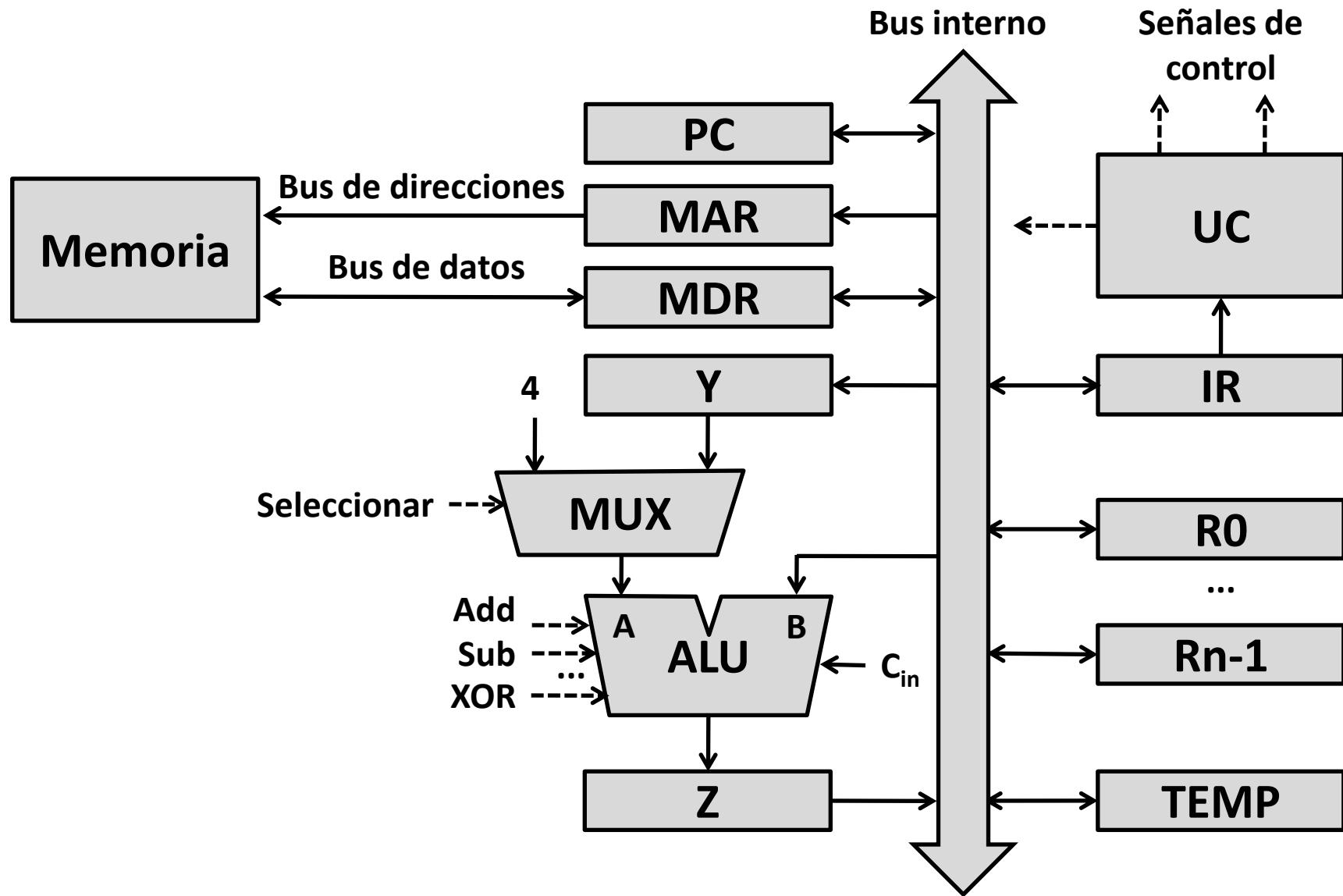
- Por ejemplo: Add A, R0
 - $M[A] + R0 \rightarrow R0$
- Detalles en [HAM03] Cap-1.3
- Ejercicios similares en TOC §2.1



Ej. de ejecución Add A, R0*

- $M[POS_A] + R0 \rightarrow R0$
 - Ensamblador traduce p.ej: $POS_A=100$
 - Valor anterior R0 perdido, el de POS_A se conserva
 - Arquitectura R/M
- **Pasos básicos de la UC**
 - PC apunta a posición donde se almacena instrucción
 - **Captación:** $MAR \leftarrow PC$, Read, $PC \leftarrow PC + 1$, $T_{acc} \leftarrow$ MDR \leftarrow bus, $IR \leftarrow MDR$
 - **Decodificación:** se separan campos instrucción
 - Codop: ADD $mem + reg \rightarrow reg$
 - Dato1: 100 direcciónamiento directo, habrá que leer $M[100]$
 - Dato2: 0 direcciónamiento registro, habrá que llevar R0 a ALU
CPUs con longitud instrucción variable – dirección (100) en siguiente palabra
 - **Operando:** $MAR \leftarrow 100$, Read, $T_{acc} \leftarrow$ MDR, $ALU_{in1} \leftarrow MDR$
 - **Ejecución:** $ALU_{in2} \leftarrow R0$, add, T_{alu}
 - **Almacenamiento:** $R0 \leftarrow ALU_{out}$

Unidad de procesamiento con un bus



Unidad de procesamiento con un bus

- Componentes interconectados mediante bus común
- MDR: dos entradas, dos salidas
- MAR: unidireccional (procesador → memoria)
- R0...Rn-1: GPR, SP, índices,...
- Y, Z, TEMP:
 - registros transparentes al programador
 - almácen temporal interno en la ejecución de una instrucción
- MUX: selecciona entrada A de la ALU
 - La constante 4 se usa para incrementar el contador de programa
- La UC genera señales internas y externas (memoria)

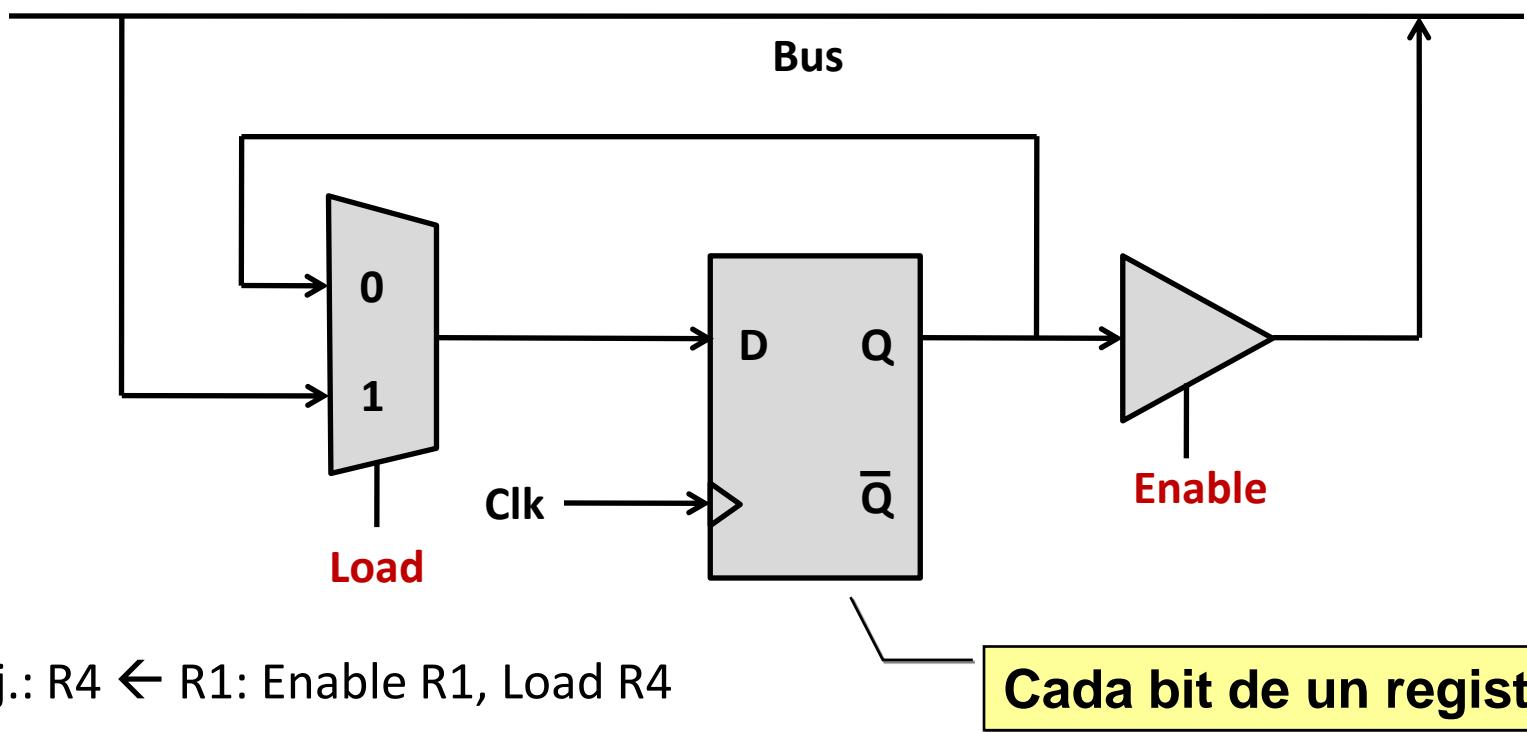
Unidad de procesamiento con un bus

- Una instrucción puede ser ejecutada mediante una o más de las siguientes operaciones:
 - Transferir de un registro a otro
 - Realizar operación aritmética o lógica y almacenar en registro
 - Cargar posición de memoria en registro
 - Almacenar registro en posición de memoria

Unidad de procesamiento con un bus

■ Transferir de un registro a otro

- Cada registro usa dos señales de control:
 - Load: Carga en paralelo
 - Enable: Habilitación de salida (buffer triestado)



Unidad de procesamiento con un bus

■ Realizar operación aritmética o lógica y almacenar en registro

- ALU: circuito combinacional sin memoria
- Resultado almacenado temporalmente en Z
- Ej.: $R3 \leftarrow R1 + R2$

1. Enable R1, Load Y
2. Enable R2, Select Y, Add, Load Z
3. Enable Z, Load R3

Cada línea: 1 ciclo de reloj

Esta transferencia no puede realizarse en el paso 2 ¿por qué?

- Las señales de control de la ALU podrían estar codificadas

Unidad de procesamiento con un bus

■ Cargar posición de memoria en registro

- Transferir dirección a MAR
- Activar lectura de memoria
- Almacenar dato leído en MDR (MDR dos entradas, dos salidas)
- La temporización interna debe coordinarse con la de la memoria
 - La lectura puede requerir varios ciclos de reloj
 - El procesador debe esperar la activación de señal de finalización de ciclo de memoria
- Ej.: **Load (R1) → R2**
 1. Enable R1, Load MAR
 2. Comenzar lectura
 3. Esperar fin de ciclo de memoria, Load MDR desde memoria
 4. Enable MDR hacia bus interno, Load R2

Unidad de procesamiento con un bus

■ Almacenar registro en posición de memoria

- Transferir dirección a MAR
- Transferir dato a escribir a MDR (MDR dos entradas, dos salidas)
- Activar escritura de memoria
- La temporización interna debe coordinarse con la de la memoria
 - La escritura puede requerir varios ciclos de reloj
 - El procesador debe esperar la activación de señal de finalización de ciclo de memoria
- Ej.: **Store R2 → (R1)**
 1. Enable R1, Load MAR
 2. Enable R2, Load MDR desde bus interno
 3. Comenzar escritura
 4. Esperar fin de ciclo de memoria

Unidad de procesamiento con un bus

■ Ejecución de una instrucción completa

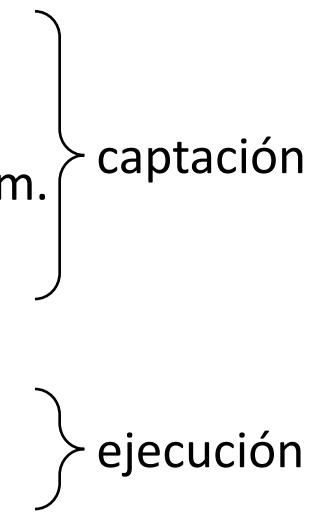
■ Ej.: **Add (R3) → R1**

1. Enable PC, Load MAR, Select 4, Sumar, Enable Z
 2. Comenzar lectura, Enable Z, Load PC, Load Y*
 3. Esperar fin de ciclo de memoria, Load MDR desde mem.
 4. Enable MDR hacia bus interno, Load IR
 5. Decodificar instrucción
 6. Enable R3, Load MAR
 7. Comenzar lectura, Enable R1, Load Y
 8. Esperar fin de ciclo de memoria, Load MDR desde mem.
 9. Enable MDR hacia bus interno, Select Y, Sumar, Load Z
 10. Enable Z, Load R1, Saltar a captación
-
- captación
- ejecución
- Saltar a captación

Unidad de procesamiento con un bus

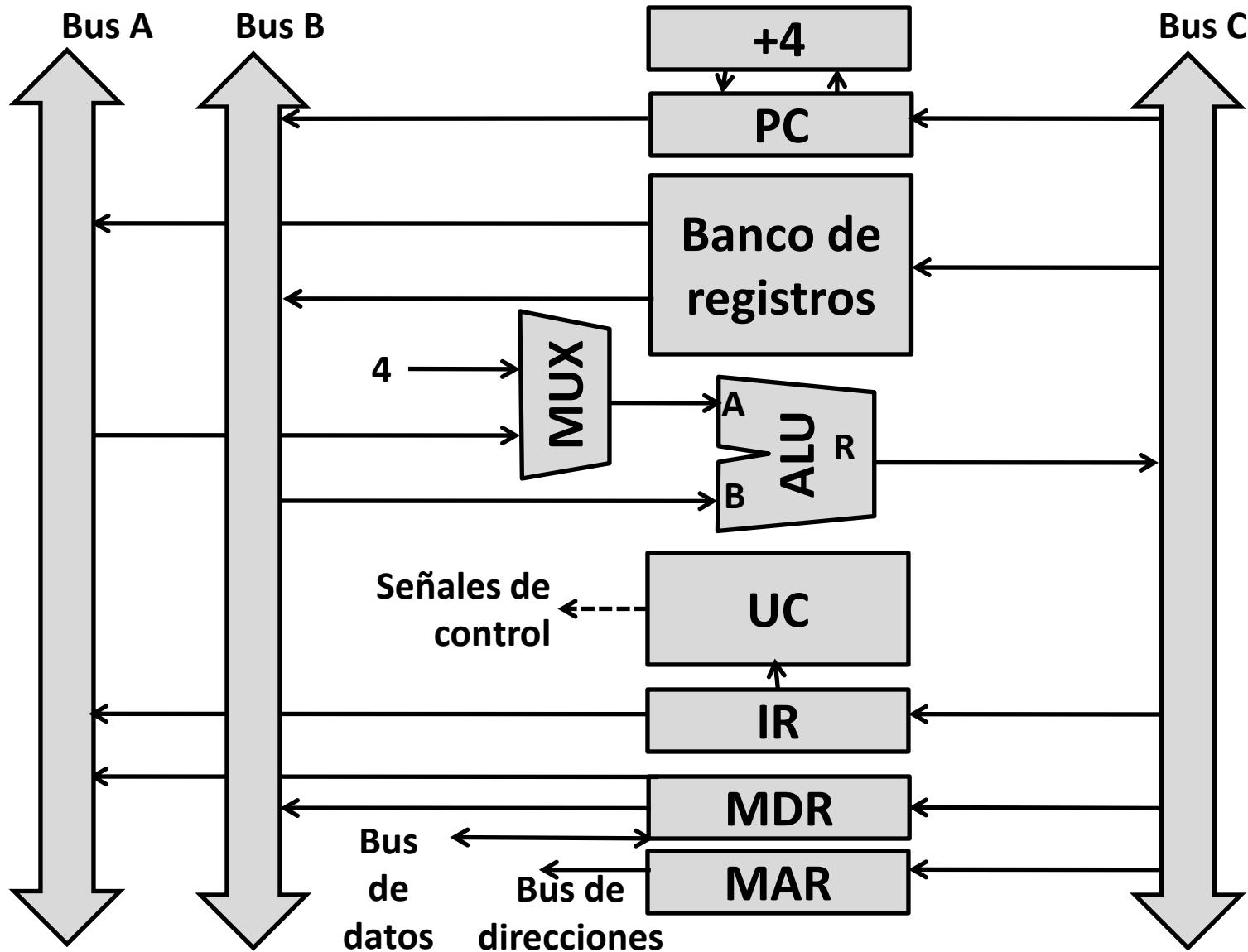
■ Ejecución de una instrucción de salto

■ Ej.: **Jmp desplazamiento**

1. Enable PC, Load MAR, Select 4, Sumar, Enable Z
 2. Comenzar lectura, Enable Z, Load PC, Load Y
 3. Esperar fin de ciclo de memoria, Load MDR desde mem.
 4. Enable MDR hacia bus interno, Load IR
 5. Decodificar instrucción
 6. Enable Campo desplazamiento en IR, Sumar, Load Z
 7. Enable Z, Load PC, Saltar a captación
- 
- captación
- ejecución

■ ¿Qué habría que añadir al paso 6 para **JS** (saltar si negativo)?

Unidad de procesam. buses múltiples



Unidad de procesam. buses múltiples

■ Banco de registros con tres puertos

- Dos registros pueden poner sus contenidos en los buses A y B
 - Un dato del bus C puede cargarse en un registro
- } En el mismo ciclo

■ ALU

- No necesita los registros Y y Z
- Puede pasar A o B directamente a R (bus C)

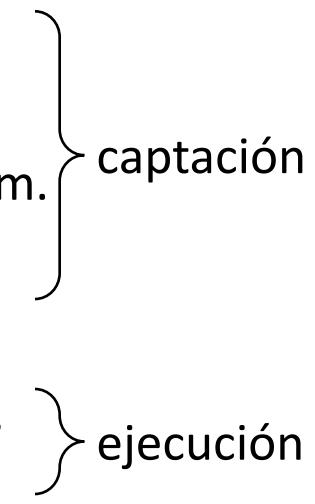
■ Unidad de incremento (+4)

- Pero la constante 4 como fuente de la ALU sigue siendo útil para incrementar otras direcciones en instrucciones de movimiento múltiple

Unidad de procesam. buses múltiples

■ Ejecución de una instrucción completa

■ Ej.: **R6 ← R4 + R5**

1. Enable PC, R=B, Load MAR
 2. Comenzar lectura, Incrementar PC
 3. Esperar fin de ciclo de memoria, Load MDR desde mem.
 4. Enable MDR hacia B, R=B, Load IR
 5. Decodificar instrucción
 6. Enable R4 hacia A, Enable R5 hacia B, Select A, Sumar,
Load R6, Saltar a captación
- 
- The diagram illustrates the execution of the instruction **R6 ← R4 + R5**. It shows a sequence of six steps. Steps 1 through 3 are grouped by a brace on the right labeled "captación" (capture), which corresponds to the initial phase of memory access. Steps 4 through 6 are grouped by another brace on the right labeled "ejecución" (execution), which corresponds to the arithmetic operation and result loading phase.

Unidad de control

■ Camino de datos

- Unidad de procesamiento y unidad de control
- Unidad de procesamiento con un bus
- Unidad de procesamiento con múltiples buses

■ Unidades de control cableadas y microprogramadas

- Diseño de una UC cableada
- Ejemplo de UC cableada
- Concepto de UC microprogramada

■ Control microprogramado

- Formato de las microinstrucciones
- Nanoprogramación
- Secuenciamiento de microinstrucciones
- Ejemplo de arquitectura microprogramada

Unidad de control

■ Señales de entrada a la UC:

- Señal de reloj
- Instrucción actual (codop, campos de direccionamiento,...)
- Estado de la unidad de proceso
- Señales externas (por ej. interrupciones)

■ Señales de salida de la UC:

- Señales que gobiernan la unidad de procesamiento:
 - Carga de registros
 - Incremento de registros
 - Desplazamiento de registros
 - Selección de entradas de multiplexores
 - Selección de operaciones de la ALU ...
- Señales externas
 - Por ej. lectura/escritura en memoria

Tipos de unidades de control

■ Existen dos formas de diseñar la UC:

- Control fijo o cableado (“hardwired”)
 - Se emplean métodos de diseño de circuitos digitales secuenciales a partir de diagramas de estados.
 - El circuito final se obtiene conectando componentes básicos como puertas y biestables, aunque más a menudo se usan PLA.
- Control microprogramado
 - Todas las señales que se pueden activar simultáneamente se agrupan para formar palabras de control, que se almacenan en una memoria de control (normalmente ROM).
 - Una instrucción de lenguaje máquina se transforma sistemáticamente en un programa (microprograma) almacenado en la memoria de control.
 - Mayor facilidad de diseño para instrucciones complejas
 - Método estándar en la mayoría de los CISC.

Diseño de una UC cableada

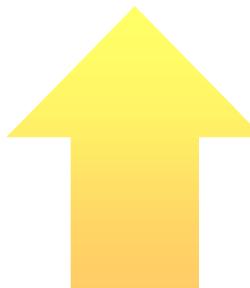
- **Se diseña mediante puertas lógicas y biestables siguiendo uno de los métodos clásicos de diseño de sistemas digitales secuenciales ya conocidos (TOC)**
 - El diseño es laborioso y difícil de modificar debido a la complejidad de los circuitos.
 - Suele ser más rápida que la misma UC microprogramada.
 - Se utilizan PLA (matrices lógicas programables) para llevar a cabo la implementación.

Debido a las modernas técnicas de diseño y a RISC ha tomado nuevo auge la realización de UC cableadas

Diseño de una UC cableada

■ Técnicas de diseño por computador (CAD) para circuitos VLSI (compiladores de silicio)

- Resuelven automáticamente la mayor parte de las dificultades de diseño de lógica cableada.
- Generan directamente las máscaras de fabricación de circuitos VLSI a partir de descripciones del comportamiento funcional del circuito en un lenguaje de alto nivel.



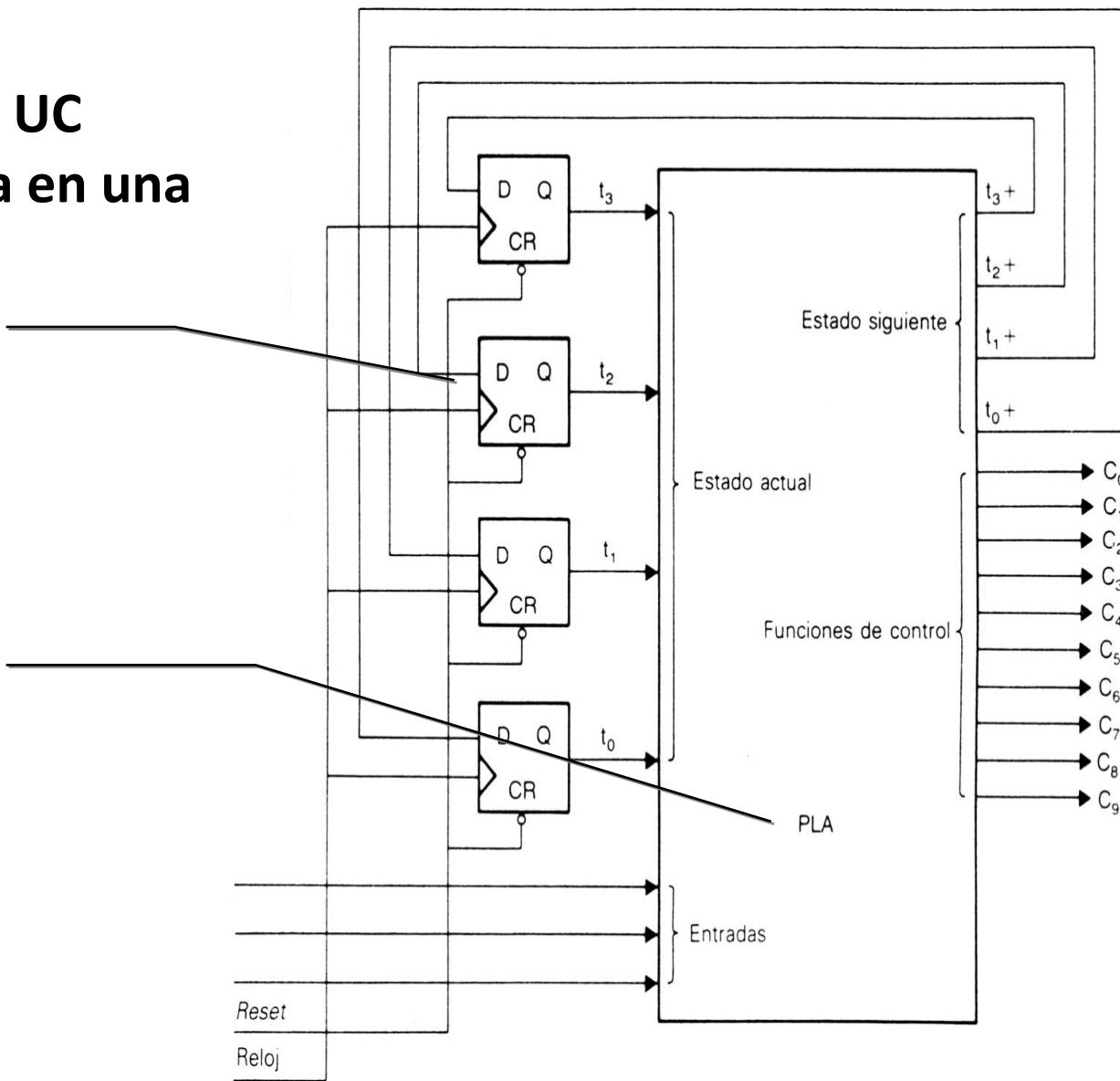
Debido a las modernas técnicas de diseño y a RISC ha tomado nuevo auge la realización de UC cableadas

Diseño de una UC cableada

- Organización de UC cableada basada en una PLA:

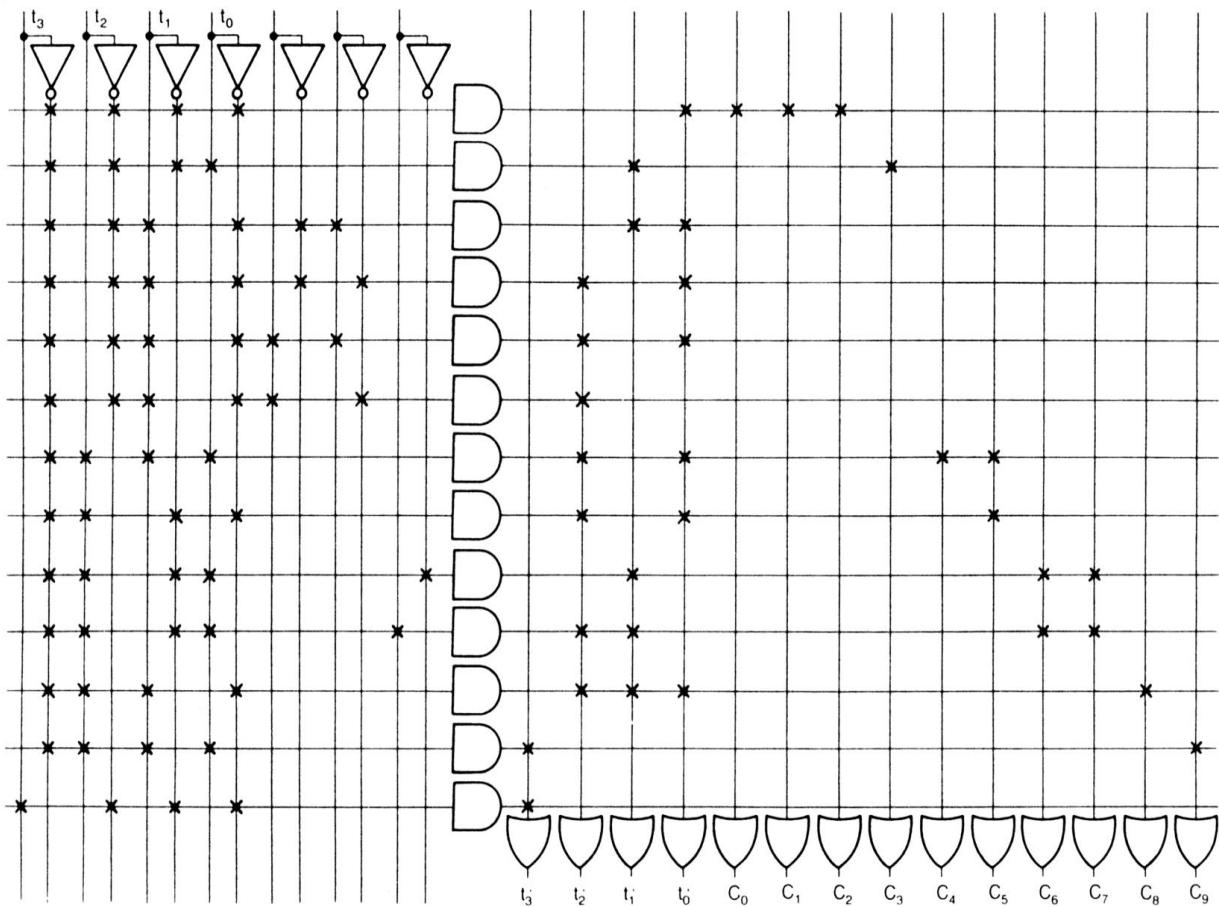
Los biestables contienen la información relativa al estado en que se encuentra el sistema

La PLA utiliza esta información de estado, junto con las entradas externas, para generar el siguiente estado



Diseño de una UC cableada

- Organización de UC cableada basada en una PLA:



Ventajas:

- ✓ Minimización del esfuerzo de diseño.
- ✓ Mayor flexibilidad y fiabilidad.
- ✓ Ahorro de espacio y potencia.

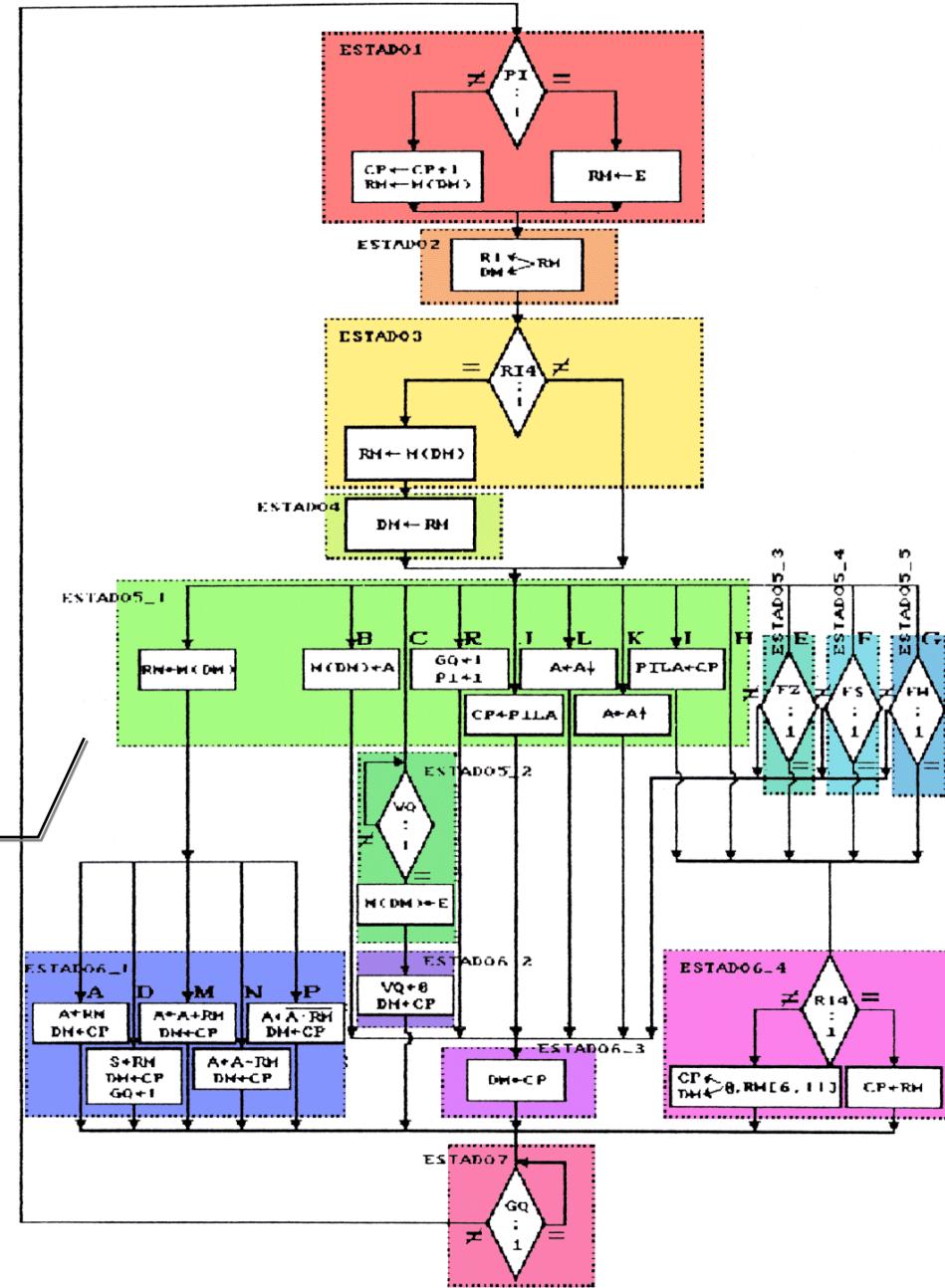
Ejemplo de UC cableada

- **Implementación de una unidad de control cableada sencilla (ODE)**
- **Pasos a seguir para llegar al diseño físico:**
 1. Definir una máquina de estados finitos
 2. Describir dicha máquina en un lenguaje de alto nivel
 3. Generar la tabla de verdad para la PLA
 4. Minimizar la tabla de verdad
 5. Diseñar físicamente la PLA partiendo de la tabla de verdad

Ej. de UC cableada

- **1. Definir una máquina de estados finitos**
 - Dado el diagrama de flujo de la UC de ODE, detallamos éste como un conjunto finito de estados y transiciones entre ellos.

Modelo *Mealy*: salidas dependen de entradas y estado presente



Ej. de UC cableada

■ 2. Describir dicha máquina en un lenguaje de alto nivel

- El lenguaje concreto depende del programa que utilicemos para “compilar” la descripción de la máquina.
- Estos lenguajes tienen sentencias para definir:
 - entradas y salidas
 - estados y transiciones condicionales e incondicionales entre estados

```

-- ***** UNIDAD DE CONTROL DE ODE *****
-- Definicion de líneas de entrada y de salida

INPUTS: RIO R11 R12 R13 R14 PI V0 G0 FS FW FZ V;
OUTPUTS: A B C D E F G H I J K L M N P R S Y T U R1 R2 R3 R4 R5 R6 INSTR_C INSTR_D INSTR_R;

-- Estado de reset

RESET ON V TO STATE ESTADO07 (A=? E=? F=? G=? I J=? L=? M=? N=? R=? S=? Y=?);

-- Definicion de estados

ESTADO01: IF PI THEN ESTADO02 (A=? E=? F=? G=? I J=0 K L=? M=? N=? R=? S=? Y=? R1)
          ELSE ESTADO02 (A=? D E=? F=? G=? I J=1 K L=? M=0 N=0 R=? S=? Y=? R1);

ESTADO02: GOTO ESTADO03 (A=? E=? F=? G=0 H I J=? L=0 M=1 N=? R=? S=? Y=? U R2);

ESTADO03: IF R14 THEN ESTADO04 (A=? E=? F=? G=? I J=1 K L=? M=? N=? R=? S=? Y=? R3)
          ELSE ESTADO05_1 (A=? E=? F=? G=? I J=? L=? M=? N=? R=? S=? Y=? R3);

ESTADO04: GOTO ESTADOS_1 (A=? E=? F=? G=0 H I J=? L=1 M=1 N=? R=? S=? Y=? R4 INSTR_C=0 INSTR_D=0 INSTR_R=0);

ESTADOS_1: CASE (RIO R11 R12 R13)
    0 0 0 1 => ESTADO06_3 (A=? E=? F=? G=? I J=? L=? M=0 N=1 R=? S=? Y=? RS);
    0 0 1 0 => ESTADO05_2 (A=? E=? F=? G=? I J=? L=? M=? N=? R=? S=? Y=? RS);
    1 1 0 1 => ESTADO06_3 (A=? E=? F=? G=? I J=? L=? M=? N=? R=? S=? Y=? RS INSTR_R);
    1 0 0 1 => ESTADO06_3 (A=? B C E=0 F=1 G=? I J=? L=? M=? N=? R=? S=? Y=? RS);
    1 0 1 1 => ESTADO06_3 (A=? E=? F=? G=? I J=? L=? M=? N=? R=1 S=0 Y=0 T RS);
    1 0 0 0 => ESTADO06_3 (A=? E=? F=? G=? I J=? L=? M=? N=? R=? S=? Y=? RS);
    0 1 0 1 => ESTADO05_3 (A=? E=? F=? G=? I J=? L=? M=? N=? R=? S=? Y=? RS);
    0 1 0 0 => ESTADO05_3 (A=? E=? F=? G=? I J=? L=? M=? N=? R=? S=? Y=? RS);
    0 1 1 0 => ESTADO05_4 (A=? E=? F=? G=? I J=? L=? M=? N=? R=? S=? Y=? RS);
    0 1 1 1 => ESTADO05_5 (A=? E=? F=? G=? I J=? L=? M=? N=? R=? S=? Y=? RS);
    0 1 0 1 => ESTADO06_4 (A=? E=? F=? G=? I J=? L=? M=? N=? R=? S=? Y=? RS);
    1 0 0 0 => ESTADO06_4 (A=? B E=1 F=0 G=? I J=? L=? M=? N=? R=? S=? Y=? RS);
    => ESTADO06_1 (A=? E=? F=? G=? I J=1 K L=? M=? N=? R=? S=? Y=? RS);

ESTADO05_2: IF V0 THEN ESTADO06_2 (A=? E=? F=? G=? I J=? L=? M=0 N=0 R=? S=? Y=? RS)
             ELSE LOOP (A=? E=? F=? G=? I J=? L=? M=? N=? R=? S=? Y=? RS);

ESTADO05_3: IF FZ THEN ESTADO06_4 (A=? E=? F=? G=? I J=? L=? M=? N=? R=? S=? Y=? RS)
             ELSE ESTADO06_3 (A=? E=? F=? G=? I J=? L=? M=? N=? R=? S=? Y=? RS);

ESTADO05_4: IF FS THEN ESTADO06_4 (A=? E=? F=? G=? I J=? L=? M=? N=? R=? S=? Y=? RS)
             ELSE ESTADO06_3 (A=? E=? F=? G=? I J=? L=? M=? N=? R=? S=? Y=? RS);

ESTADO05_5: IF FW THEN ESTADO06_4 (A=? E=? F=? G=? I J=? L=? M=? N=? R=? S=? Y=? RS)
             ELSE ESTADO06_3 (A=? E=? F=? G=? I J=? L=? M=? N=? R=? S=? Y=? RS);

ESTADO06_1: CASE (RIO R11 R12 R13)
    0 0 0 0 => ESTADO07 (A=? E=? F=? G=1 H I J=? L=1 M=1 N=? R=1 S=1 Y=0 T R6);
    0 0 1 1 => ESTADO07 (A=? E=? F=? G=1 H I J=? L=1 M=1 N=? P R=? S=? Y=? R6 INSTR_D);
    1 1 0 0 => ESTADO07 (A=? E=? F=? G=1 H I J=? L=1 M=1 N=? R=0 S=0 Y=1 T R6);
    1 1 0 1 => ESTADO07 (A=? E=? F=? G=1 H I J=? L=1 M=1 N=? R=0 S=1 Y=0 T R6);
    1 1 1 0 => ESTADO07 (A=? E=? F=? G=1 H I J=? L=1 M=1 N=? R=0 S=1 Y=1 T R6);
    ENCASE => ANY (A=? B=? C=? D=? E=? F=? G=? H=? I=? J=? K=? L=? M=? N=? P=? R=? S=? Y=? T=? U=? R1=? R2=? R3=? R4=? RS=? R6=?)

ESTADO06_2: GOTO ESTADO07 (A=? E=? F=? G=1 H I J=? L=? M=? N=? R=? S=? Y=? R6 INSTR_C);

ESTADO06_3: GOTO ESTADO07 (A=? E=? F=? G=0 H I J=? L=? M=? N=? R=? S=? Y=? R6);

ESTADO06_4: IF R14 THEN ESTADO7 (A=0 C E=? F=? G=0 I J=? L=1 M=1 N=? R=? S=? Y=? R6)
             ELSE ESTADO7 (A=0 C E=? F=? G=0 H I J=? L=0 M=1 N=? R=? S=? Y=? R6);

ESTADO07: IF GO THEN LOOP (A=? E=? F=? G=? I J=? L=? M=? N=? R=? S=? Y=?)
             ELSE ESTADO1 (A=? E=? F=? G=? I J=? L=? M=? N=? R=? S=? Y=?);

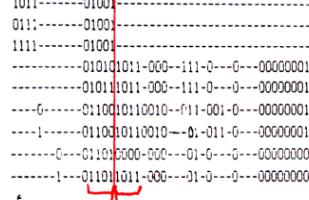
```

Ej. de UC cableada

■ 3. Generar tabla de verdad necesaria para PLA

- Según la descripción que hayamos hecho de la máquina de estados...
 - podemos usar un programa que use el modelo **Mealy**
 - salidas dependen de entradas y de estado presente
 - o uno que use el modelo **Moore**
 - salidas dependen exclusivamente de estado actual

.i 16
.o 33
.ilb RIO R11 R12 RI3 RI4 PI V0 GG FS FW FZ v StBit0 StBit1* StBit2* StBit3**
.ob StBit3 StBit2* StBit1* StBit0* A B C D E F G H I J K L M N P R S Y T U R1 R2 R3 R4 R5 R6 INSTR_C INSTR_D INSTR_R*
.p -1 **Estatus actual y siguiente**
-----1----1011-000---01-0---0---00000000000
-----0----000001000-001---0111-000---00100000000
-----1----000001000-000---0101---0---00100000000
-----0----000001000-000---0101-001---00100000000
-----0----000100010-000---01-0---0---00001000000
-----1----000101100-000---0111---0---00001000000
-----0----000100100-000---0111-011-0---00000100000
0000----001001001-000---0111---0---00000100000
1000----001000011-000---01010-01-0---00000010000
0100----001000110-000---01-0---0---00000010000
1100----001001001-000---0111---0---00000010000
0010----001001010-000---01-0---0---00000010000
0101----001001110-000---01-0---0---00000010000
1101----001001001-000---0111---0---00000010000
0011----0010001001-000---0111---0---00000010000
1011----0010001101-000---01-0---0---00000010000
0111----001001101-000---01-0---0---00000010000
1111----001001101-000---01-0---0---00000010000
-----0----001001010-000---01-0---0---00000010000
-----1----001001001-000---00-0---0---00000010000
-----00010011001-000---00-0---0---00000010000
-----10010001111001-01-0---0---000000010000
0101----0010001100-000---01-0---0---000000010000
1101----0010001001-000---0111---0---000000010000
0011----0010001001-000---0111---0---000000010000
1011----0010001101-000---01-0---0---000000010000
0111----0010000011-000---01-0---0---000000010000
1111----0010001101-000---01-0---0---000000010000
-----0----0010001010-000---01-0---0---000000010000
-----1----0010001010-000---00-0---0---000000010000
-----00010011001-000---00-0---0---000000010000
-----10010001111001-01-0---0---000000010000
0101----0010001100-000---01-0---0---000000010000
1101----0010001001-000---0111---0---000000010000
0011----0010001001-000---0111---0---000000010000
1011----0010001101-000---01-0---0---000000010000
0111----0010000011-000---01-0---0---000000010000
1111----0010001101-000---01-0---0---000000010000
-----0----0010001010-000---01-0---0---000000010000
-----1----0010001010-000---00-0---0---000000010000
-----0001000011-000---00-0---0---000000010000
-----10010001111001-01-0---0---000000010000
0100----0010001100-000---01-0---0---000000010000
1100----0010001001-000---0111---0---000000010000
0001----0010001001-000---0111---0---000000010000
1001----0010001101-000---01-0---0---000000010000
0101----0010000011-000---01-0---0---000000010000
1101----0010001011-000---111-011-0000100000010000
0011----0010001011-000---111-011-1---00000001010
1011----0010001011-000---111-011-1---00000001010
0111----0010001011-000---111-011-1---00000001010
1111----0010001011-000---111-011-0---00000001010
-----0010001011-000---111-011-0---00000001010
-----0010001011-000---111-011-0---00000001010
-----0----01100010110010-011-001-0---00000001010
-----1----01100010110010-011-011-0---00000001010
-----0----01100010110000---01-0---0---00000000000
-----1----0110001011-000---01-0---0---00000000000
.e



16 entradas

33 salidas

53 términos producto

PLANO AND

PLANO OR

Ej. de UC cableada

■ 4. Minimizar la tabla de verdad

- Mediante un programa que utiliza algoritmos heurísticos rápidos.

41 términos producto (PLA más pequeña)

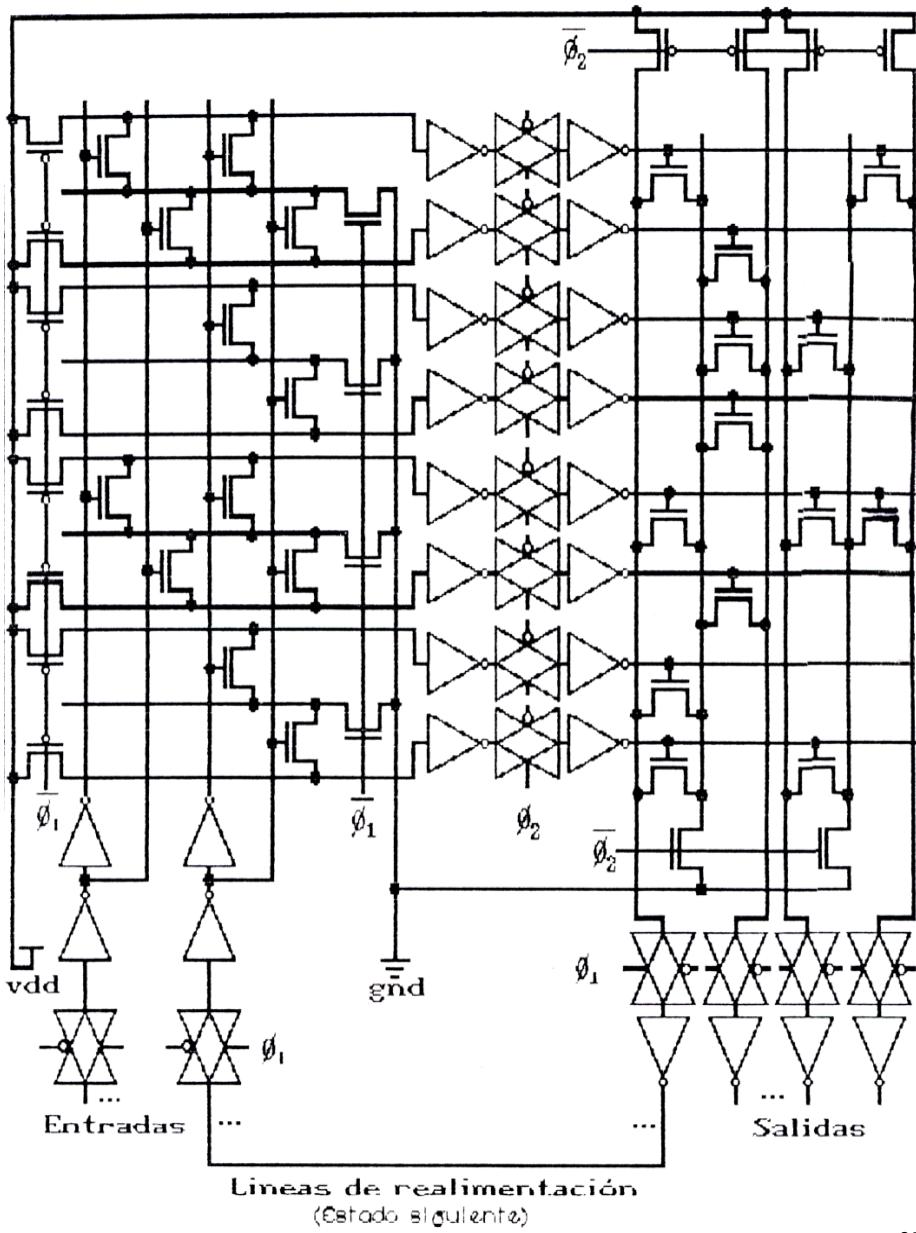


Ej. de UC cableada

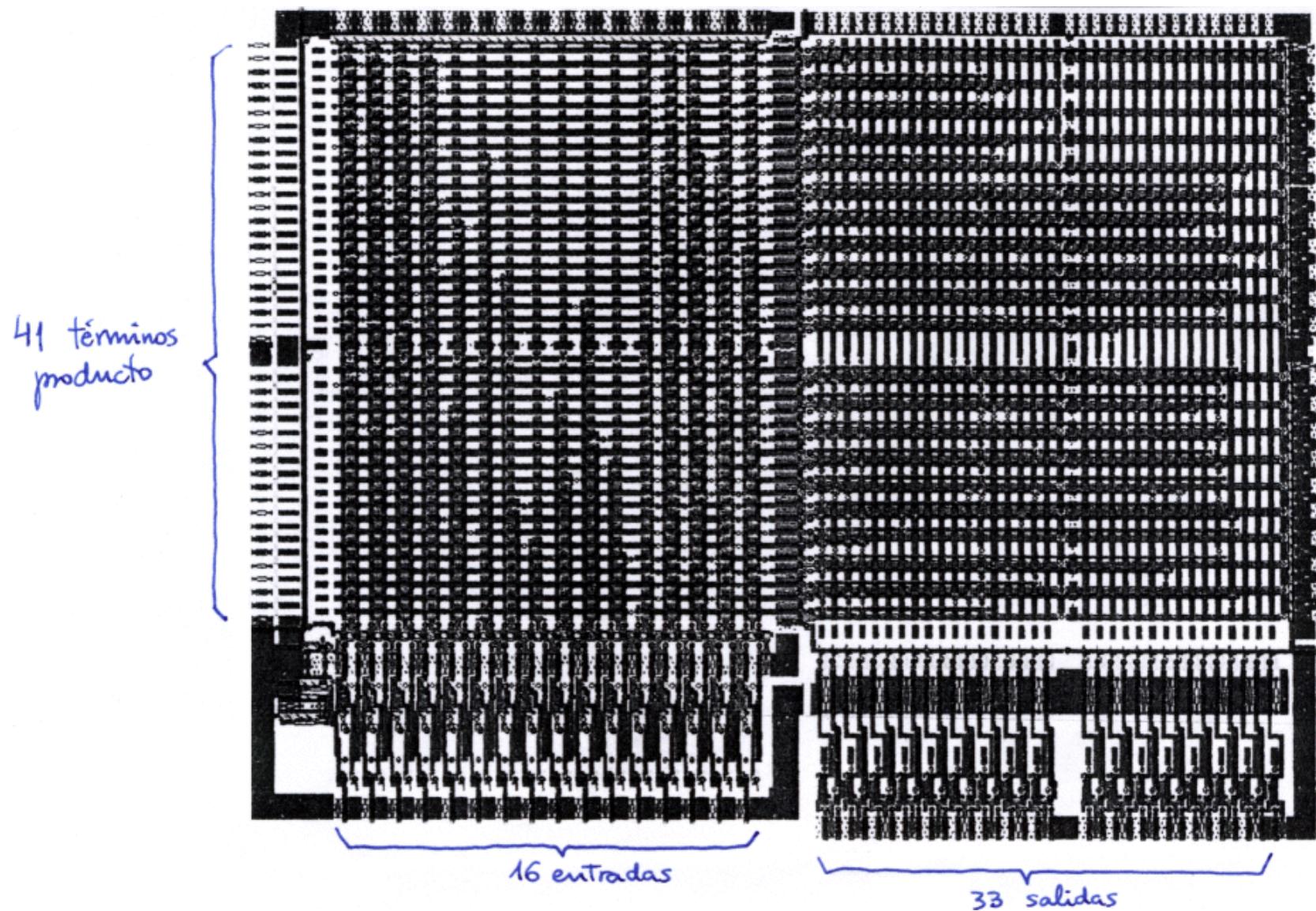
- **5. Diseñar físicamente la PLA partiendo de la tabla de verdad minimizada**
 - Automáticamente:
 - Mediante un programa especial para diseño de layouts de PLA.
 - Semiautomáticamente:
 - Diseñando mediante un programa de CAD de circuitos VLSI cada una de las celdas que, repetidas convenientemente, forman la PLA.
 - Dando una especificación de cómo han de colocarse (tabla de verdad minimizada).

Ej. de UC cableada

- Esquema simplificado de la PLA usada para la UC de ODE
 - CMOS de dos fases de reloj
 - No hacen falta biestables de estado siguiente
 - $\Phi_1=1$ se leen las entradas y se precarga el plano AND
 - $\Phi_2=1$ se evalúa el plano AND y se precarga el plano OR
 - $\Phi_2=0$ se evalúa el plano OR



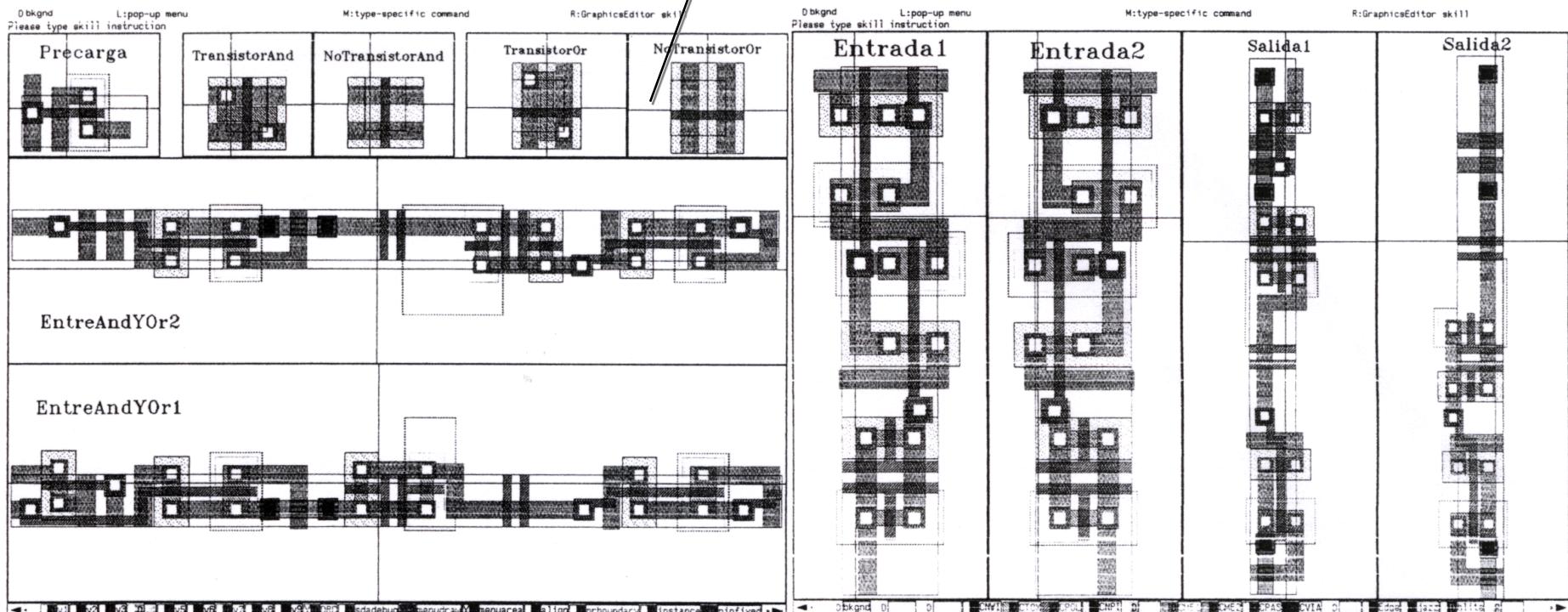
Ej. de UC cableada



Ej. de UC cableada

- PLA diseñada semiautomáticamente

Detalle de las celdas básicas que constituyen la PLA diseñadas con un programa de CAD.
Un programa puede unirlas de acuerdo con el archivo de la tabla de verdad minimizada.



Concepto de UC microprogramada

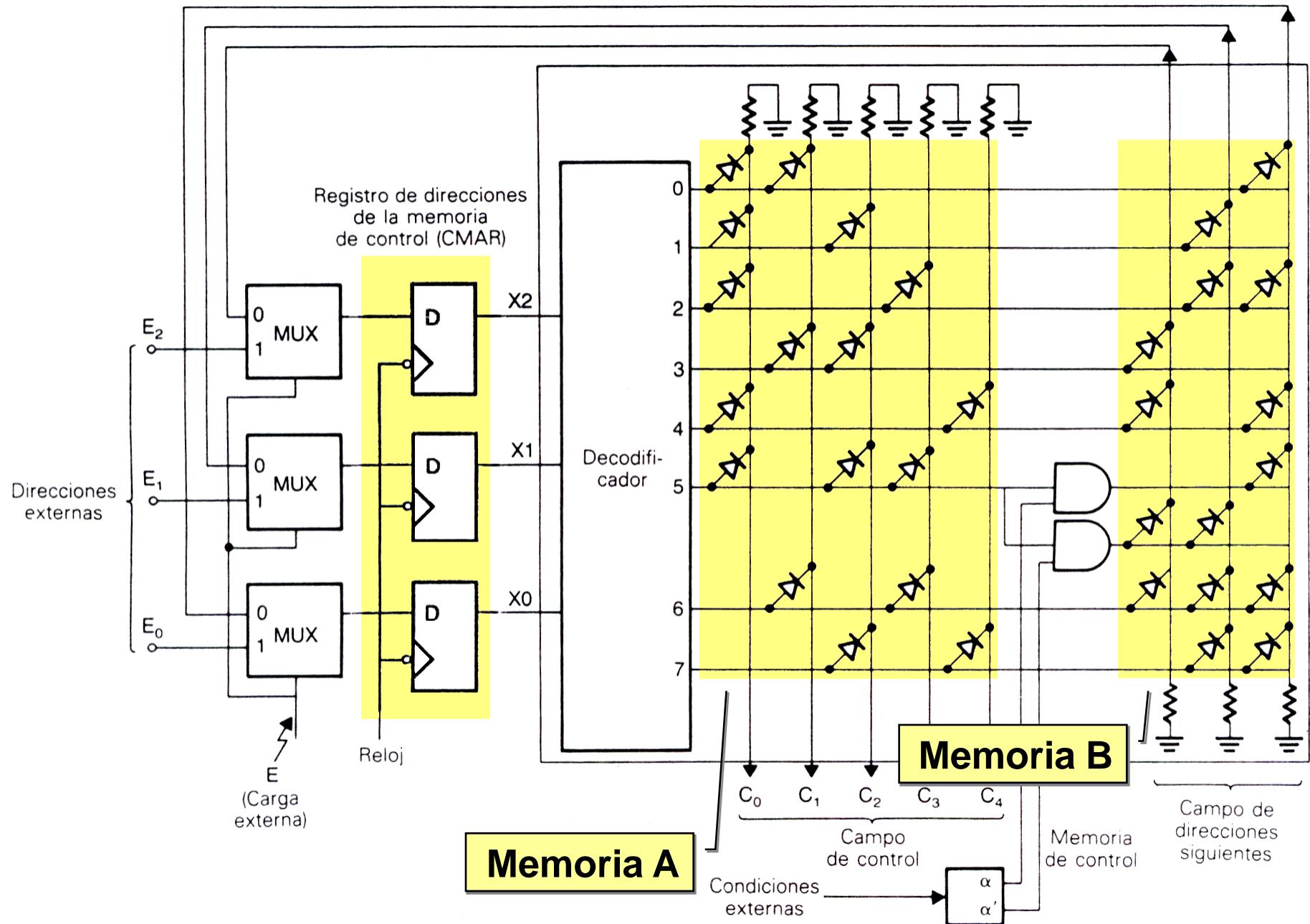
■ Idea básica:

- Emplear una memoria (de control) para almacenar las señales de control de los períodos de cada instrucción

■ Origen histórico

- Maurice V. Wilkes (1913-2010) en 1951-1953 propone el siguiente esquema:
 - Dos memorias A y B, construidas con matrices de diodos.
 - Las señales de control se encuentran almacenadas en la memoria A.
 - La memoria B contiene la dirección de la siguiente microinstrucción.
 - Se permiten microbifurcaciones condicionales, mediante un biestable y un decodificador que selecciona entre dos direcciones de la matriz B.
- Más info.: <http://www.cs.clemson.edu/~mark/uprog.html>





Concepto de UC microprogramada

■ Definiciones:

- **Microinstrucción**: cada palabra de la memoria de control
- **Microprograma**: conjunto ordenado de microinstrucciones cuyas señales de control constituyen el cronograma de una (macro)instrucción del lenguaje máquina.
- Ejecución de un microprograma: lectura en cada pulso de reloj de una de las microinstrucciones que lo forman, enviando las señales leídas a la unidad de proceso como señales de control.
- **Microcódigo**: conjunto de los microprogramas de una máquina.

Concepto de UC microprogramada

■ Ventajas de la microprogramación:

- Simplicidad conceptual.
 - La información de control reside en una memoria.
- Se pueden incluir, sin dificultades, instrucciones complejas, de muchos ciclos de duración.
 - El único límite es el tamaño de la memoria de control.
- Las correcciones, modificaciones y ampliaciones son mucho más fáciles de hacer que en una unidad de control cableada.
 - No hay que rediseñar toda la unidad, sino sólo cambiar el contenido de algunas posiciones de la memoria de control.
- Permite construir computadores con varios juegos de instrucciones, cambiando el contenido de la memoria de control (si es RAM permite emular otros computadores).

Concepto de UC microprogramada

- **Desventaja de la microprogramación:**
 - Lentitud frente a cableada, debido a una menor capacidad de expresar paralelismo de las microinstrucciones.

Unidad de control

■ Camino de datos

- Unidad de procesamiento y unidad de control
- Unidad de procesamiento con un bus
- Unidad de procesamiento con múltiples buses

■ Unidades de control cableadas y microprogramadas

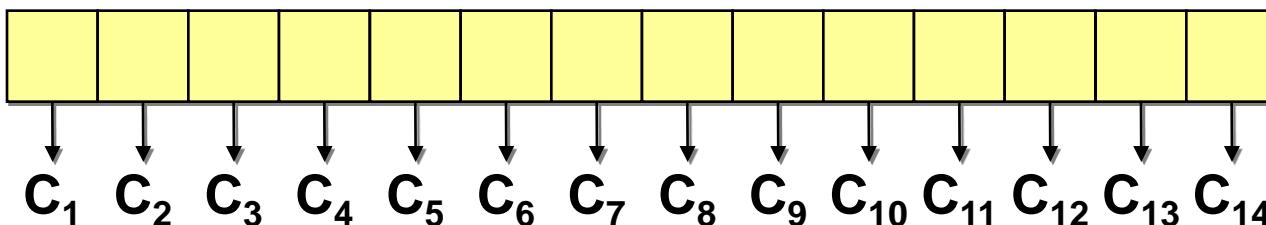
- Diseño de una UC cableada
- Ejemplo de UC cableada
- Concepto de UC microprogramada

■ Control microprogramado

- Formato de las microinstrucciones
- Nanoprogramación
- Secuenciamiento de microinstrucciones
- Ejemplo de arquitectura microprogramada

Formato de las microinstrucciones

- Las señales de control que gobiernan un mismo elemento del datapath se suelen agrupar en campos.
 - Ejemplos:
 - señales triestado que controlan el acceso a un bus
 - señales de operación de la ALU
 - señales de control de la memoria
- Formato no codificado:
 - Hay un bit para cada señal de control de un campo

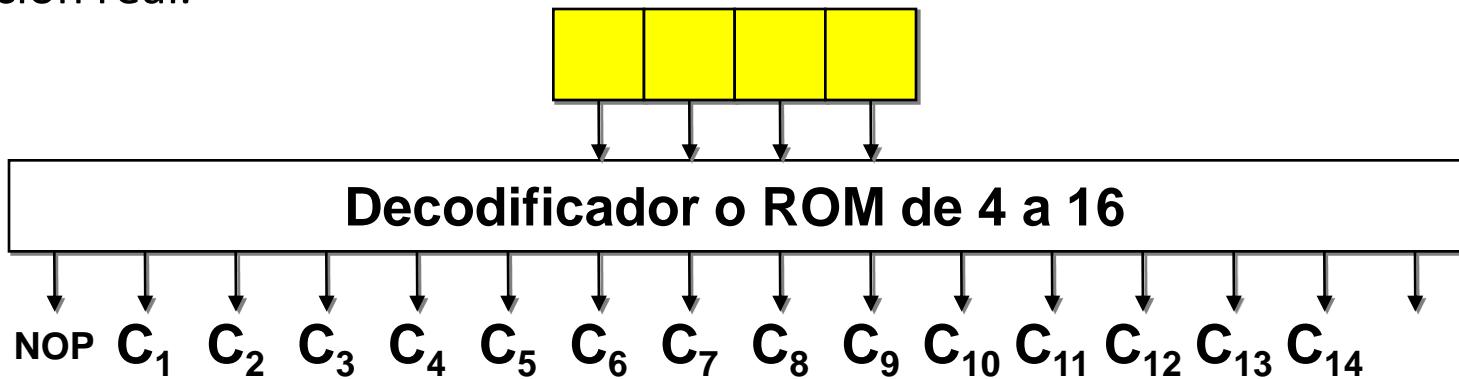


Formato de las microinstrucciones

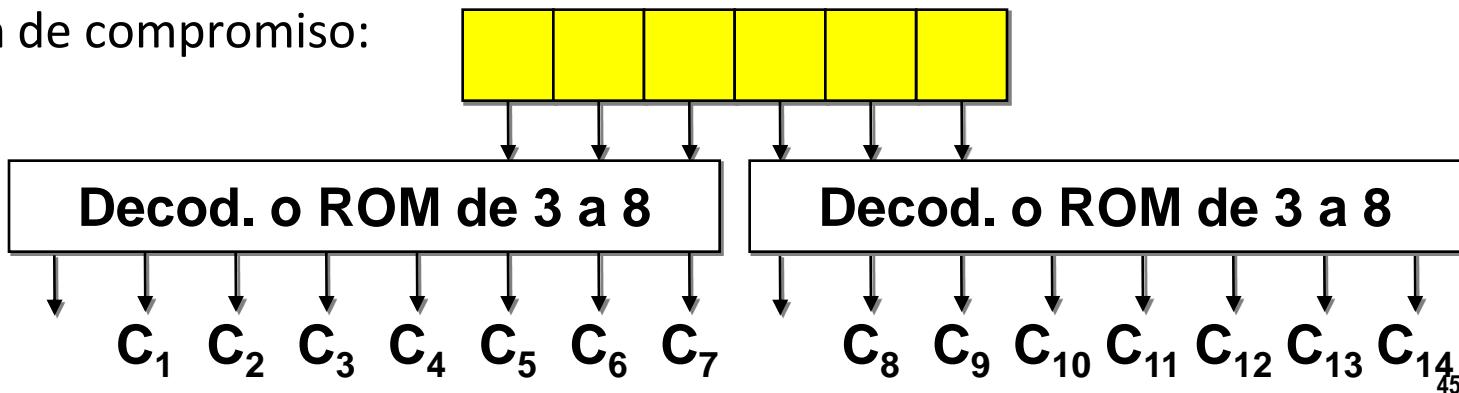
■ Formato codificado:

- Para acortar el tamaño de las microinstrucciones se codifican todos o alguno de sus campos.

Inconveniente: Hay que incluir decodificadores para extraer la información real.



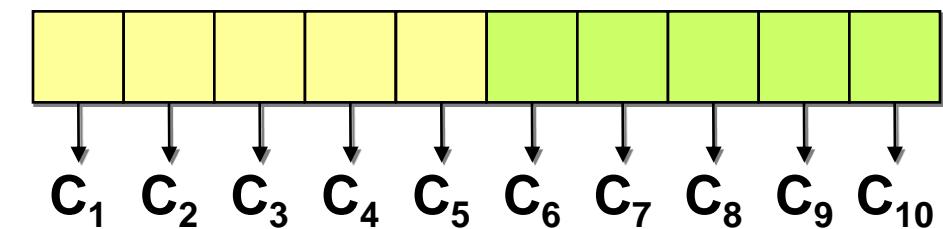
- Solución de compromiso:



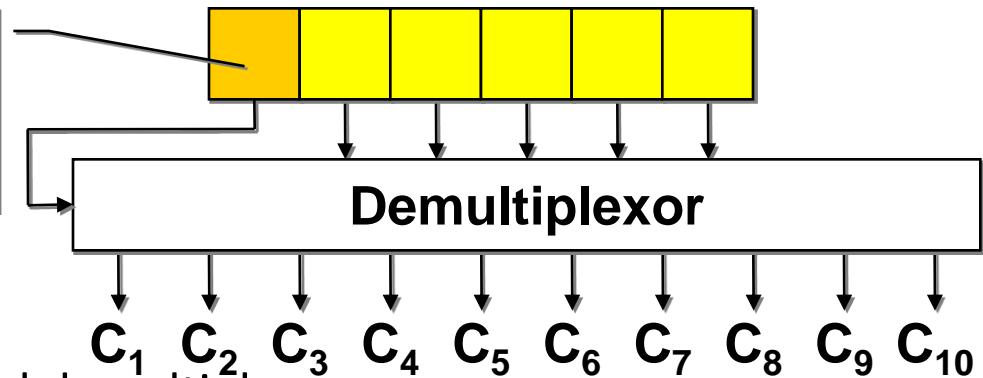
Formato de las microinstrucciones

■ Solapamiento de campos:

- Si sólo unas pocas señales de control están activas en cada ciclo, o existen con frecuencia señales excluyentes, que no se pueden activar simultáneamente...
- ...se puede acortar la longitud de las microinstrucciones solapando campos.



La señal adicional de control define si los bits corresponden al campo 1 o al campo 2



- Inconvenientes:
 - Retardo introducido por el demultiplexor.
 - Hace incompatibles las operaciones de los campos solapados.

Formato de las microinstrucciones

Micro-programación vertical:

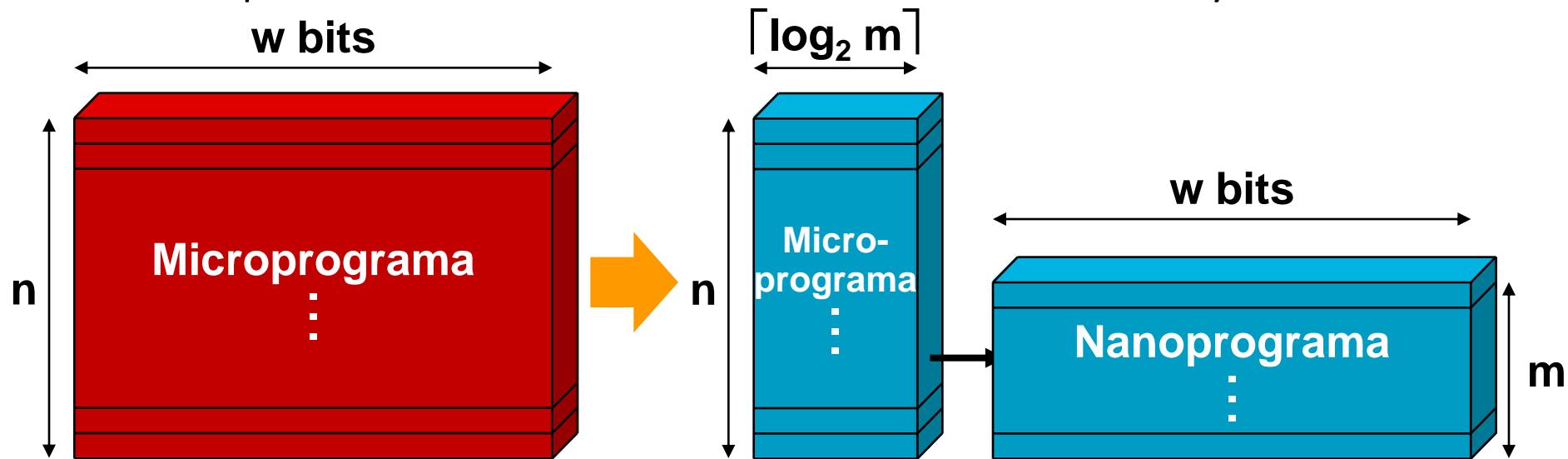
- Mucha codificación
- ✓ Microinstrucciones cortas
- ✗ Escasa capacidad para expresar paralelismo (la longitud del programa se ve incrementada)

Micro-programación horizontal:

- Ninguna o escasa codificación
- ✓ Capacidad para expresar un alto grado de paralelismo en las microoperaciones a ejecutar (simultáneamente)
- ✗ Microinstrucciones largas

Nanoprogramación

- **Objetivo: reducir el tamaño de la memoria de control**
 - Implica una memoria a dos niveles: memoria de control y nanomemoria.



Microprograma original con n μinstrucciones de w bits. Tamaño = $n \cdot w$

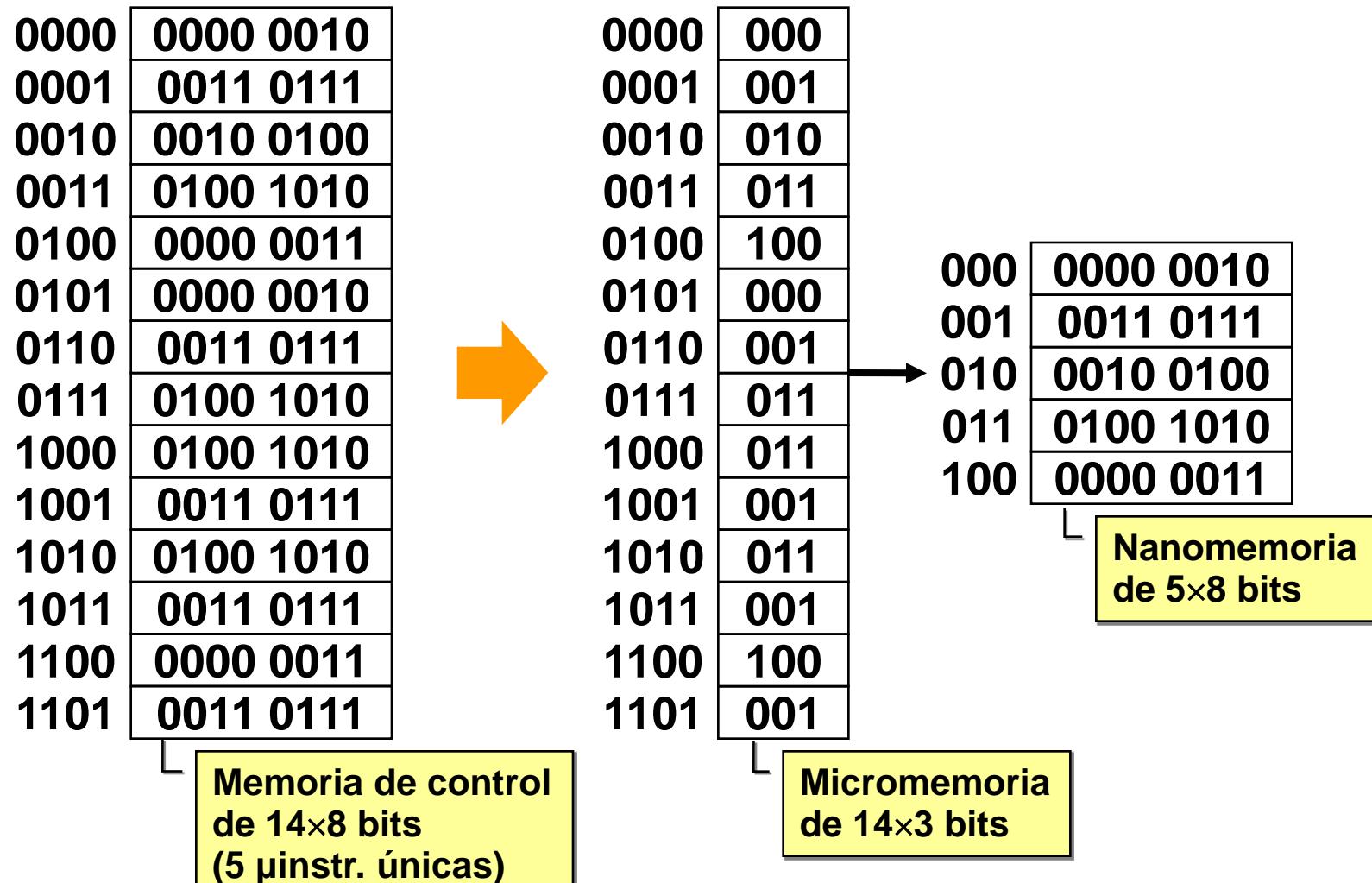
$m << n$ μinstrucciones únicas de 2^w posibles

**Se reemplaza cada μinstrucción por su dirección en la nanomemoria
Tamaño: $n \cdot \lceil \log_2 m \rceil$**

**Contiene las m μinstrucciones diferentes (cada una se incluye una sola vez).
Tamaño: $m \cdot w$**

Ahorro de memoria: $n \cdot w - (n \cdot \lceil \log_2 m \rceil + m \cdot w)$

Nanoprogramación. Ejemplo 1

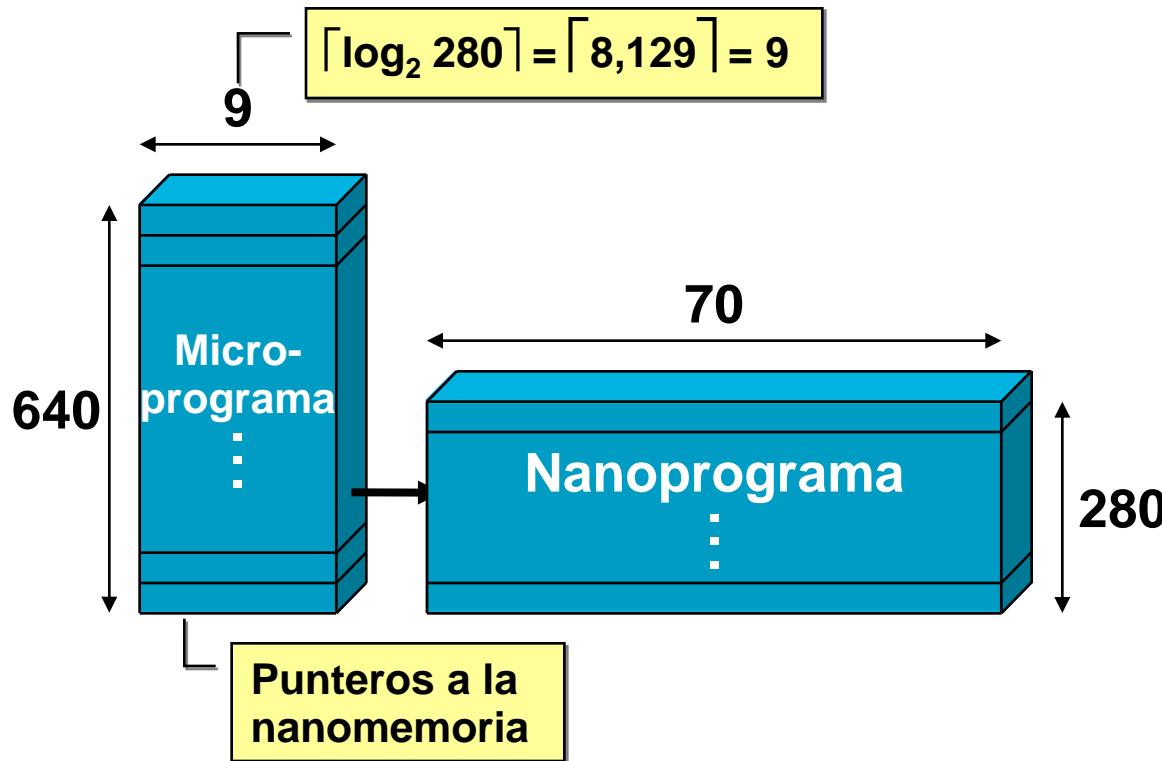


Ahorro de memoria: $14 \cdot 8 - (14 \cdot \lceil \log_2 5 \rceil + 5 \cdot 8) = 112 - 82 = 30$ bits (27%)

Nanoprogramación. Ejemplo 2

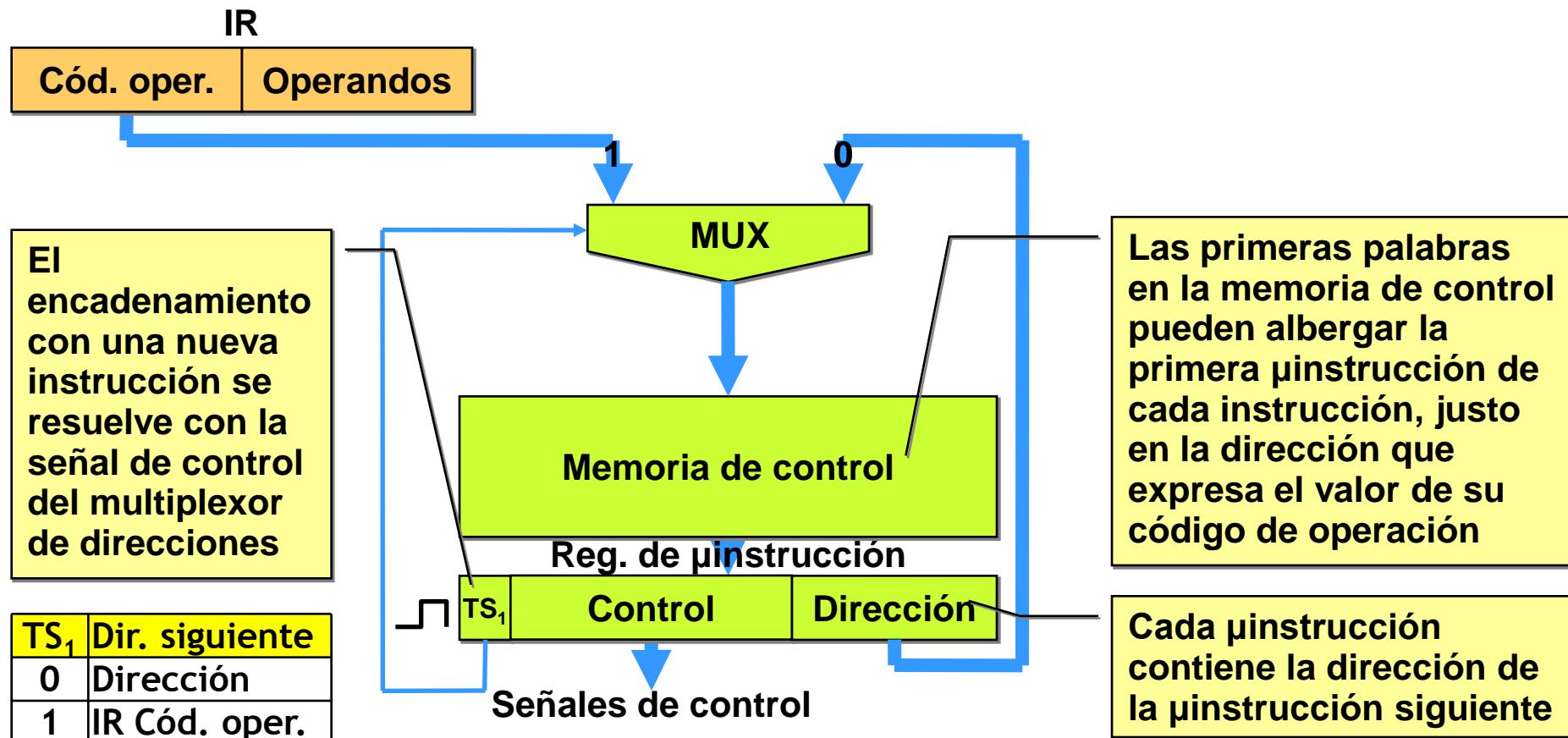
■ Estructura de la UC del Motorola 68000

- 640 microinstrucciones, de las cuales 280 son únicas



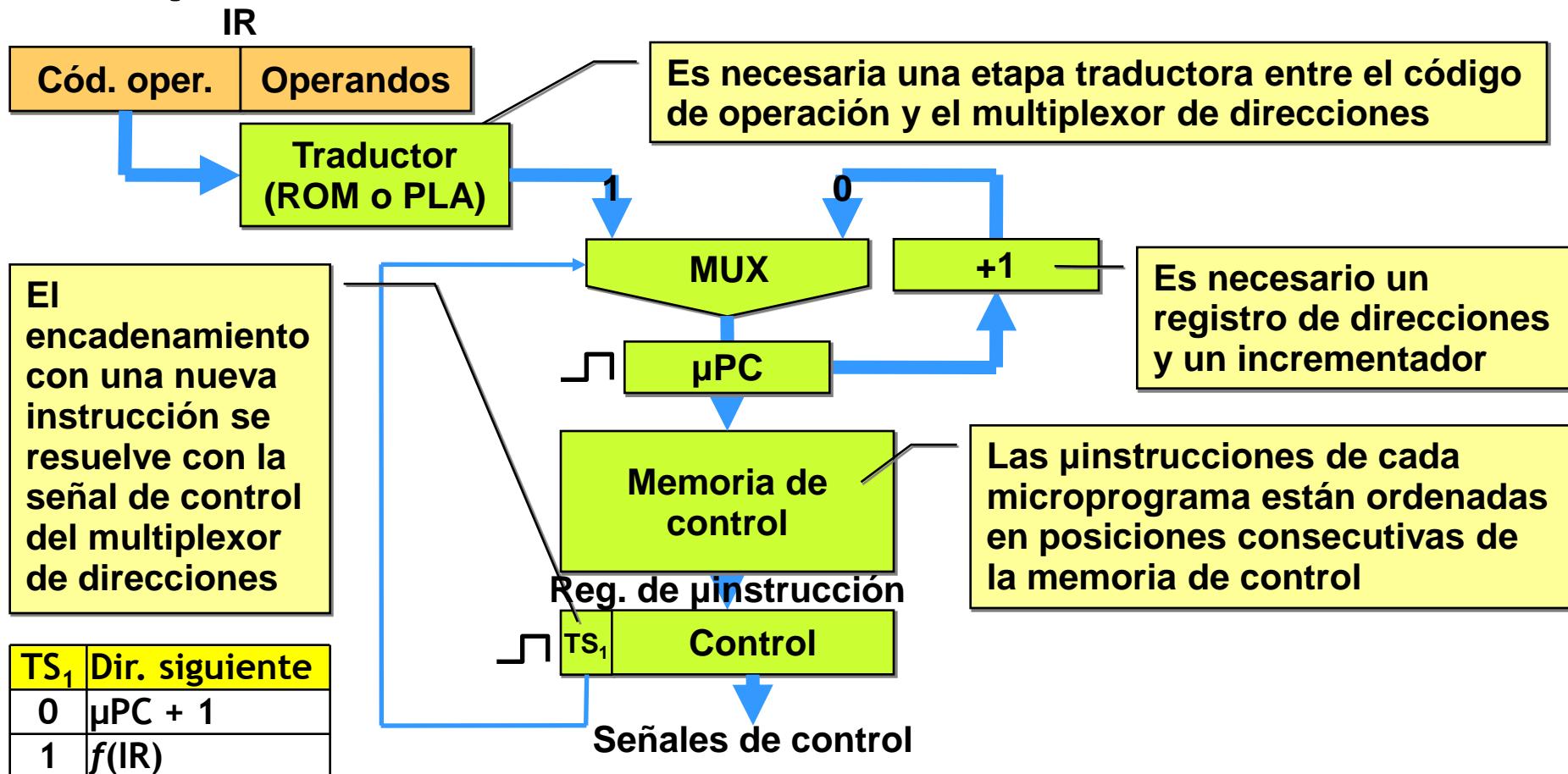
Ahorro de memoria: $640 \cdot 70 - (640 \cdot 9 + 280 \cdot 70) = 44800 - 25360 = 19440$ bits (43%)

Secuenciamiento de μinstrucciones explícito



- ✗ Inconveniente: gran cantidad de memoria empleada en el secuenciamiento (dirección de la siguiente microinstrucción)

Secuenciamiento de μinstrucciones implícito



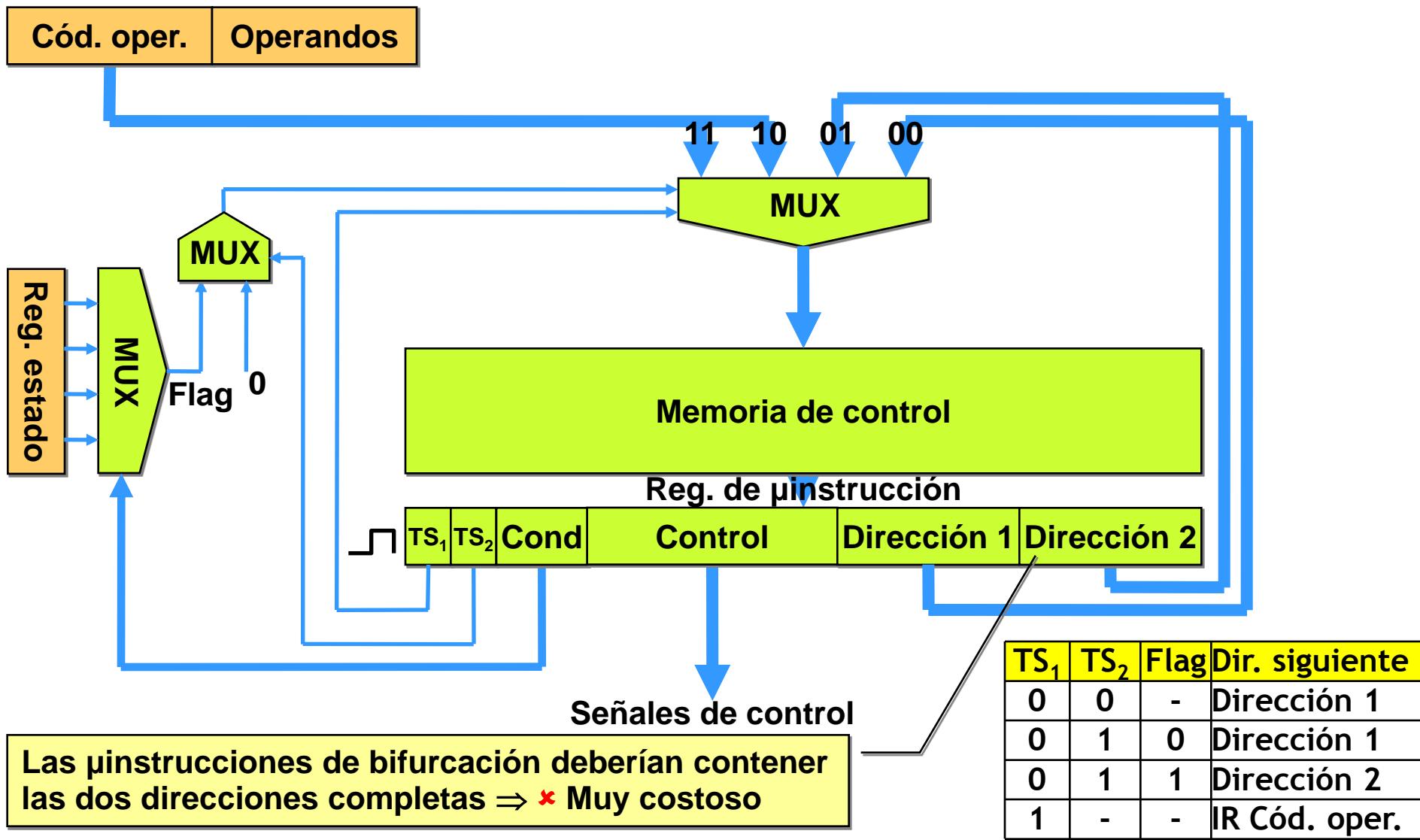
- ✗ Inconveniente: esta estructura sólo permite ejecutar programas lineales

Secuenciamiento de μinstrucciones

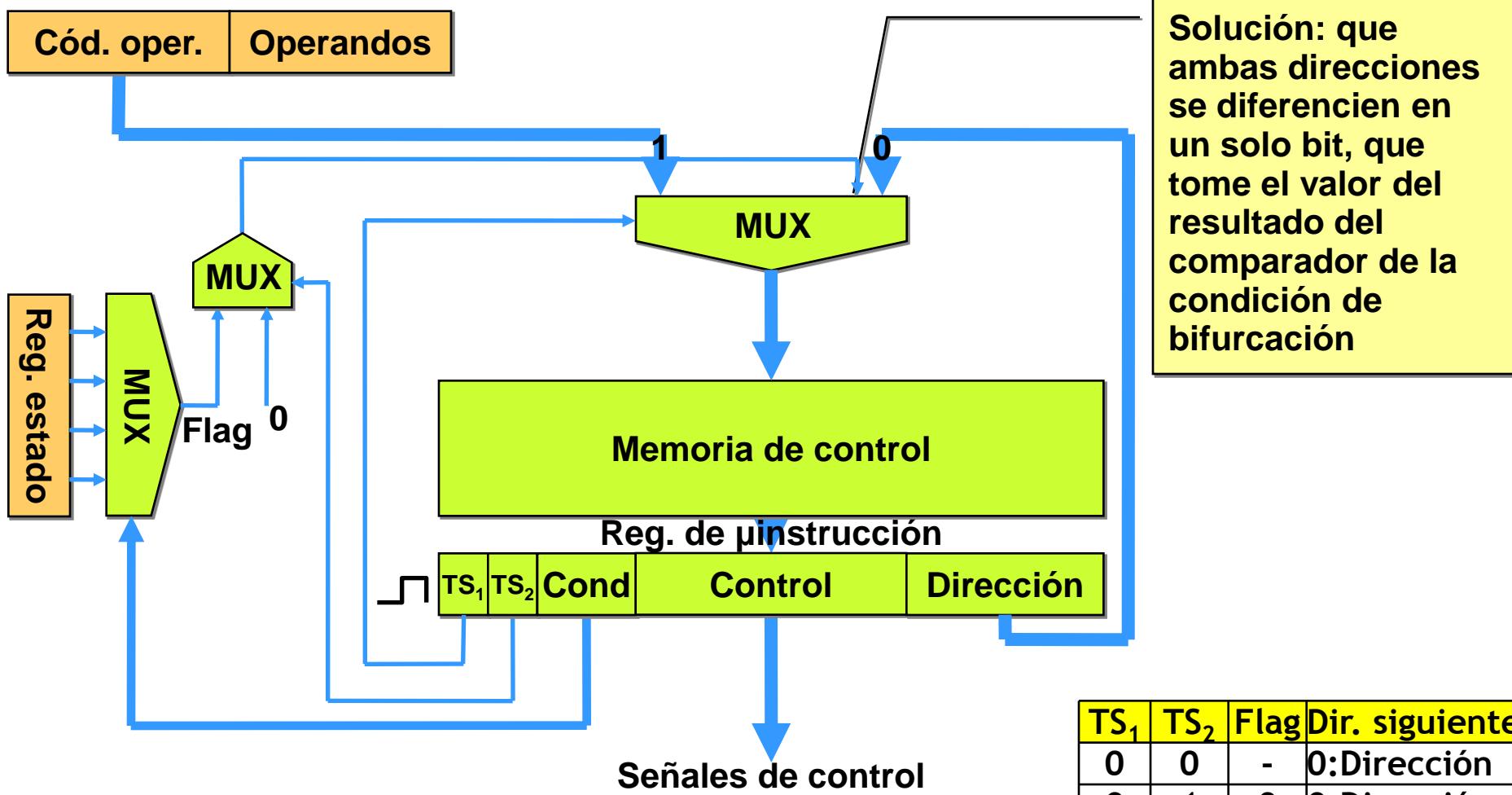
■ Microbifurcaciones condicionales

- Las instrucciones máquina de bifurcación condicional presentan dos cronogramas alternativos, diferentes a partir del punto en el que se hace la comprobación de la condición de bifurcación.
- Los microprogramas correspondientes han de presentar una microbifurcación condicional para seleccionar la rama deseada.
- Es necesario que la microinstrucción de bifurcación pueda elegir entre dos direcciones para poder seguir por uno de los dos caminos alternativos.

μ bifurcaciones condicionales en secuenciamiento explícito (1)

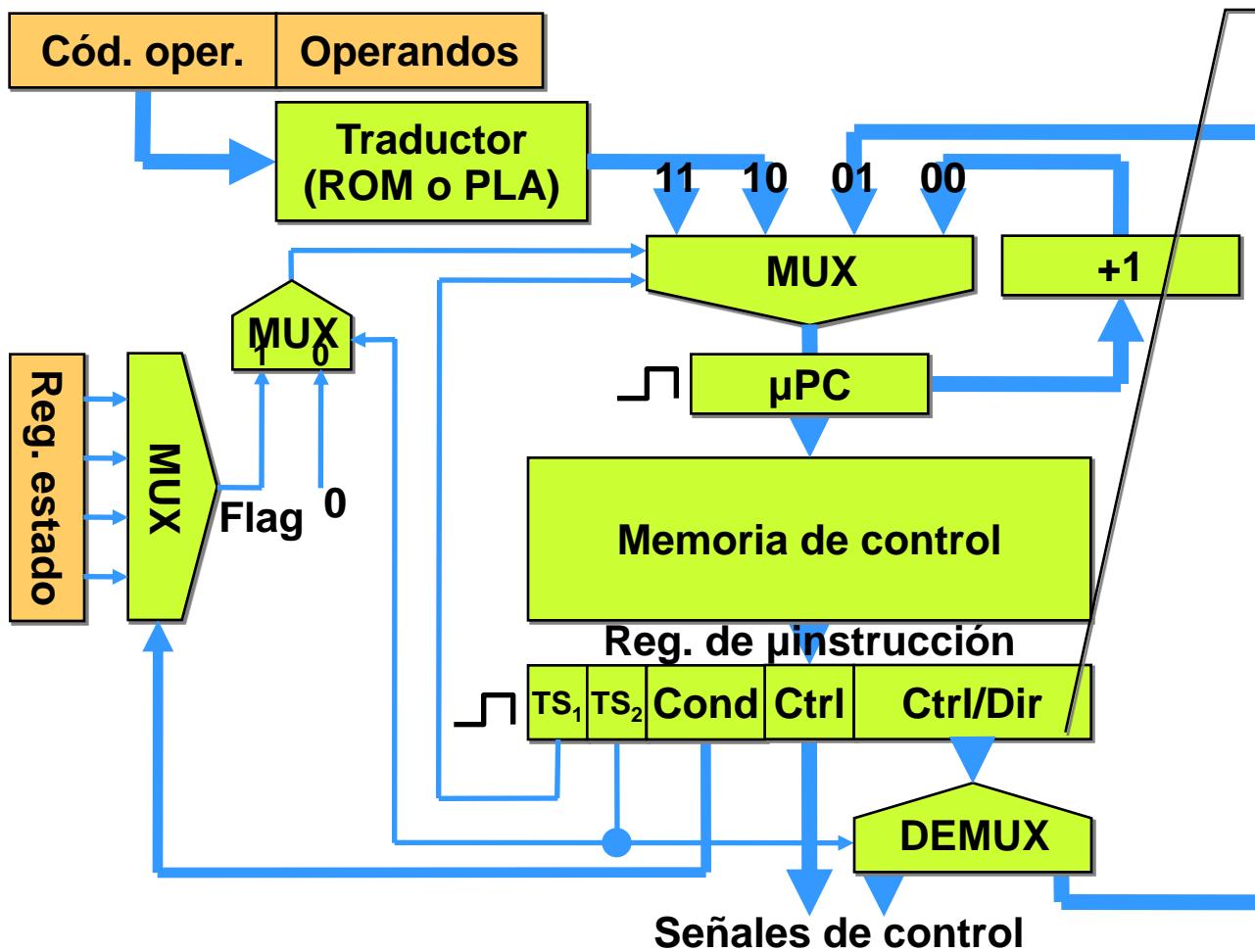


μ bifurcaciones condicionales en secuenciamiento explícito (2)



TS ₁	TS ₂	Flag	Dir. siguiente
0	0	-	0:Dirección
0	1	0	0:Dirección
0	1	1	1:Dirección
1	-	-	IR Cód. oper.

μ bifurcaciones condicionales en secuenciamiento implícito



Se debe poder elegir entre la μinstrucción siguiente u otra distinta
 \Rightarrow la μinstrucción de bifurcación debe contener dicha dirección.
 Se puede solapar el campo de dirección con otros, puesto que incluir un campo específico de dirección en todas las μinstrucciones conduciría al secuenciamiento explícito

TS ₁	TS ₂	Flag	Dir. siguiente
0	0	-	μPC + 1
0	1	0	μPC + 1
0	1	1	Dirección
1	-	-	f(IR)

Secuenciamiento de microinstrucciones

■ Microbucle

- Existen instrucciones máquina con operaciones repetidas
 - Ejs.:
 - desplazamiento múltiple
 - multiplicación
 - división
 - cadenas
 - Necesidad de microbucle.
- Se puede utilizar la bifurcación condicional más un contador con autodecremento
- Cuando el contador llegue a 0 se bifurca
- El contador debe poder inicializarse desde una microinstrucción

Secuenciamiento de μinstrucciones

■ Microsubrutinas

- Las instrucciones máquina tienen con frecuencia partes comunes
 - Necesidad de microsubrutinas.
- La llamada a microsubrutinas exige un almacenamiento para guardar la dirección de retorno.
 - La solución usual es añadir una pila al registro de direcciones (μ PC).

Control residual

■ Control inmediato:

- Hasta aquí, todas las señales de control necesarias para manipular la microarquitectura estaban codificadas en campos de la microinstrucción actual.

■ Control residual:

- En ciertos casos puede ser útil que una microinstrucción pueda almacenar señales de control en un registro (de control residual) para usarlas en ciclos posteriores.
- Objetivo: optimizar el tamaño del microprograma.
- Usos:
 - En microsubrutinas o conjuntos compartidos de μ instrucciones.
 - En caso de que parte de la información de control permanezca invariable durante muchas μ instrucciones.

Control residual

■ En microsubrutinas o conjuntos compartidos de μinstrucciones.

■ Ejemplo:

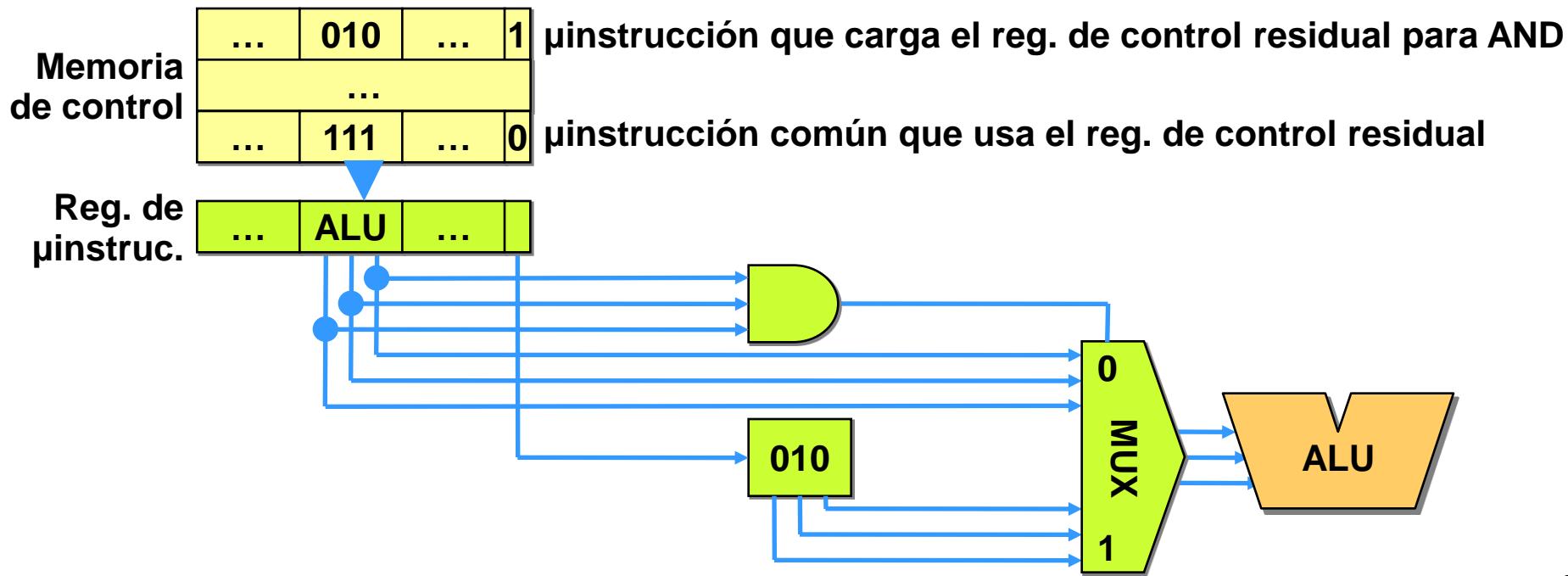
- Procesador con dos instrucciones máquina para realizar las operaciones AND y OR entre dos cadenas de bytes.
- Se puede utilizar una microsubrutina o un conjunto común de microinstrucciones para las dos instrucciones máquina, incorporando un registro de control residual en el diseño.
- Supongamos que la ALU está controlada por 3 bits:

000 Suma
111 Resta
010 AND
011 OR
100 XOR
101 NOR
110 Pasar entrada izquierda
111 No utilizada

Utilizaremos el valor 111 para especificar que las señales de control de la ALU no proceden de este campo sino del registro de control residual

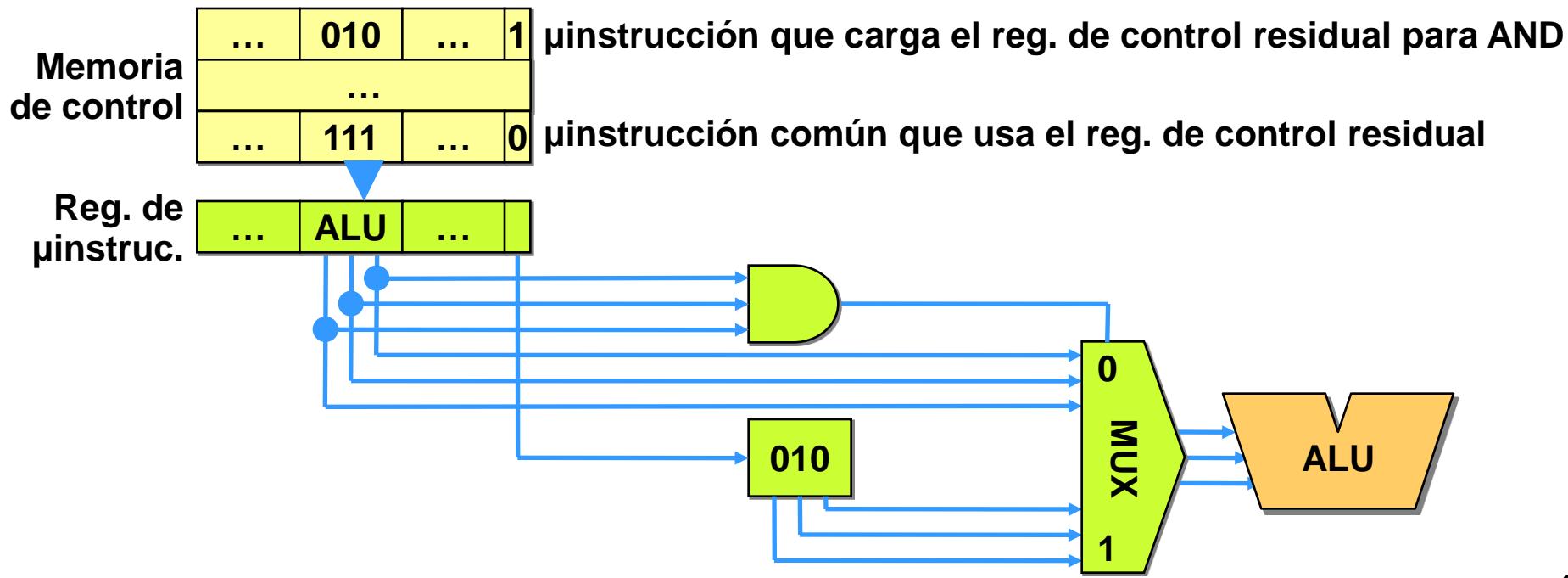
Control residual

- Necesitamos un método para cargar información en el registro de control residual, por ej. una microinstrucción especial (puede ser la microinstrucción de llamada a microsubrutina) con un campo que indique el valor a almacenar y un bit que indique que se desea almacenar información en el registro.



Control residual

- El microcódigo de la instrucción que realiza el AND almacenará 010 en el registro de control residual y la correspondiente al OR almacenará 011. Ambas saltarán o llamarán al microcódigo común.



Control residual

- El concepto del ej. anterior puede generalizarse para la especificación de otros elementos, por ej.:
 - registros destino
 - operaciones de memoria
 - direcciones o condiciones de salto
 - Se reduce el número de μ instrucciones.
-
- **En caso de que parte de la información de control no varíe durante muchas μ instrucciones.**
 - En lugar de mantener la misma información en μ instrucciones sucesivas, ésta se coloca en el registro de control residual durante un período deseado de tiempo. En este caso el registro de control residual se suele denominar registro de setup (configuración).
 - Se reduce la anchura de la μ instrucciones.

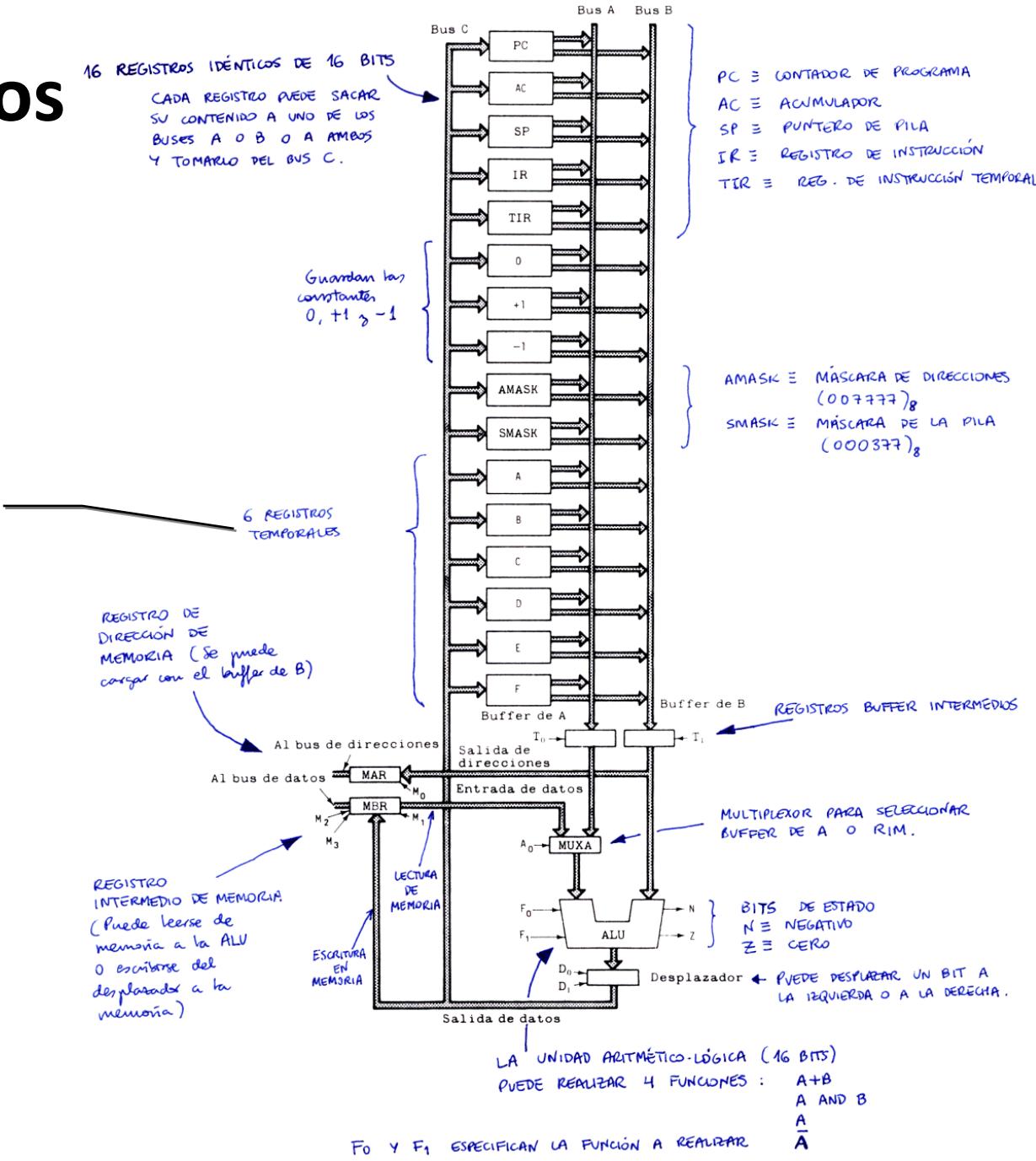
Ej. de arquitectura microprogramada

- Bibliografía: Tanenbaum: “Organización de computadoras: un enfoque estructurado”

- **Camino de datos**
- **Diseño horizontal**
 - Repertorio de instrucciones máquina
- **Diseño vertical**

Camino de datos

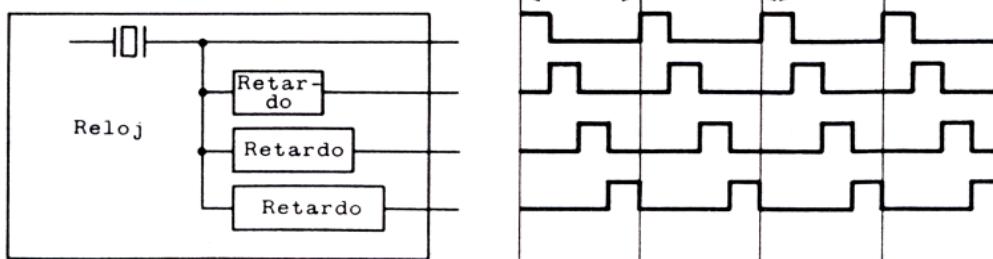
Unidad de proceso o camino de datos ("datapath")



Camino de datos

■ Temporización:

- Un ciclo básico de la UC consiste en una secuencia de 4 subciclos controlada por un reloj de 4 fases:

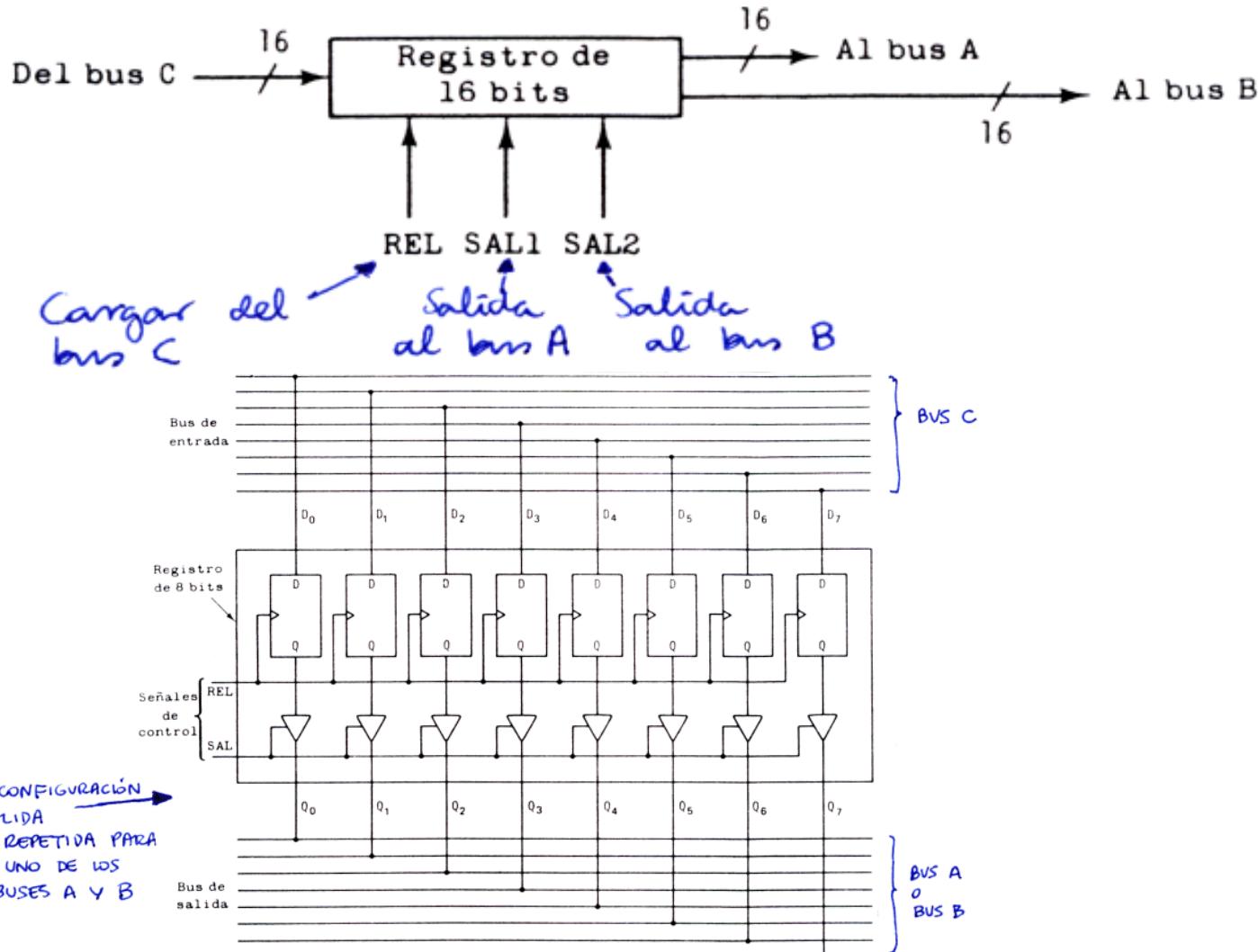


- Subciclos:

- Se lee de la memoria de control la siguiente microinstrucción a ejecutar y se carga en el registro de microinstrucción.
- Los contenidos de uno o dos registros pasan a los buses A y B y se cargan en los buffers.
- Las entradas de la ALU están estabilizadas. Se da tiempo a la ALU y al desplazador a que produzcan una salida estable y se carga MAR si es necesario.
- La salida del desplazador está estabilizada. Se almacena el bus C en un registro y en MBR si es necesario.

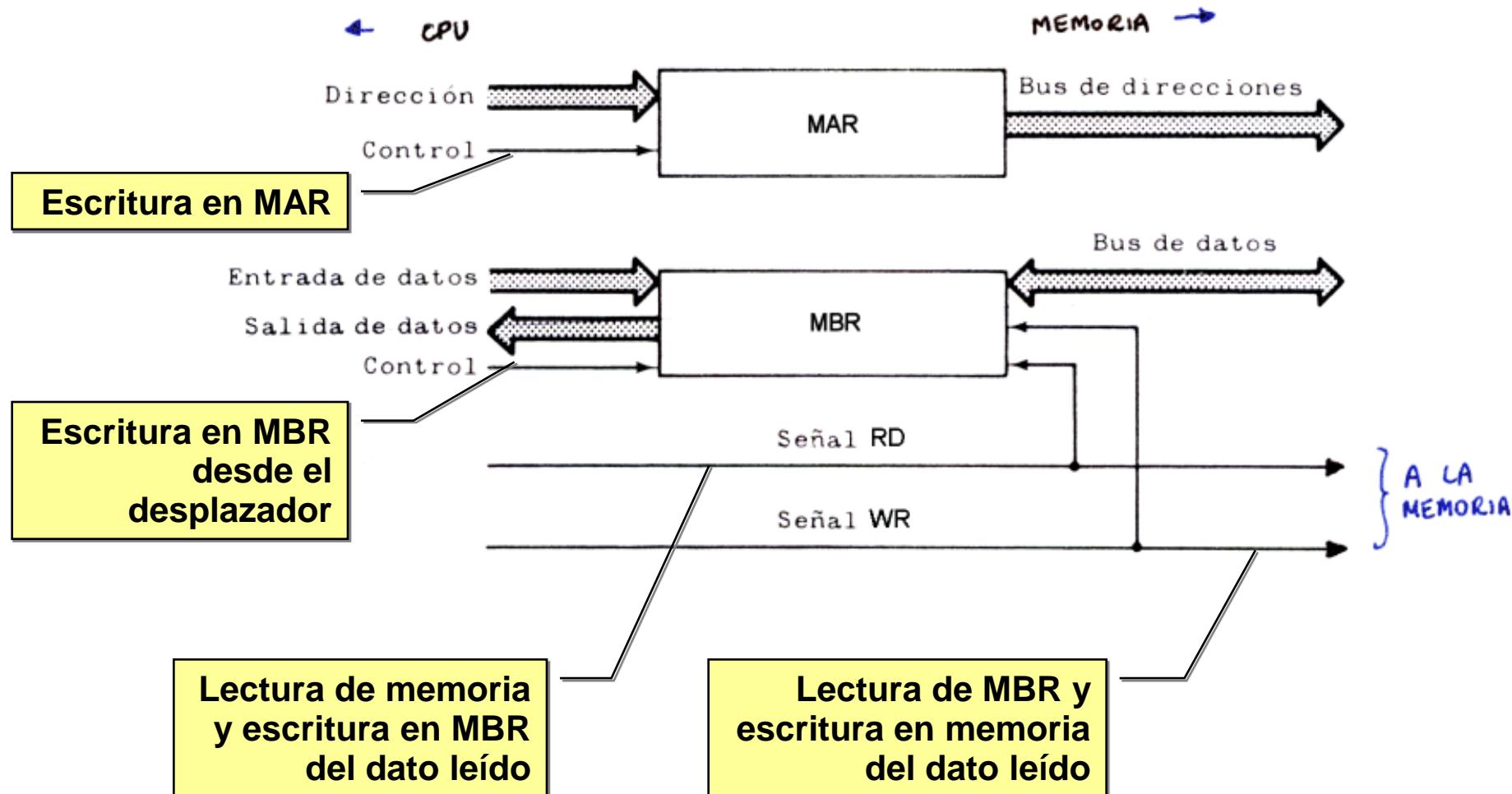
Camino de datos

■ Detalle de uno de los registros de 16 bits:



Camino de datos

■ Detalle de señales y buses de MAR y MBR:



Diseño horizontal

■ Diseño horizontal de la unidad de control

- Señales de control necesarias:
 - 16 para carga del bus A desde los registros
 - 16 para carga del bus B desde los registros
 - 16 para carga de los registros desde el bus C
 - 2 para carga de los buffers de A y B
 - 2 para controlar la función de la ALU
 - 2 para controlar el desplazador
 - 4 para controlar MAR y MBR
 - 2 para indicar lectura o escritura en memoria
 - 1 para controlar el multiplexor MUXA
- 61 señales en total

Diseño horizontal

■ Podemos reducir las señales necesarias:

- Codificando las señales de los registros (suponiendo que el contenido del bus C sólo se almacene en un registro)
 - 48 bits → 12 bits
- Eliminando los 2 bits para carga de los buffers de A y B (siempre se cargan en el mismo momento del ciclo máquina, por lo que puede activarlos el segundo subciclo del reloj)
 - 2 bits → 0 bits
- Reduciendo el nº de señales que controlan MAR y MBR (LEC puede usarse para cargar MBR desde la memoria y ESC para darle salida)
 - 4 bits → 2 bits

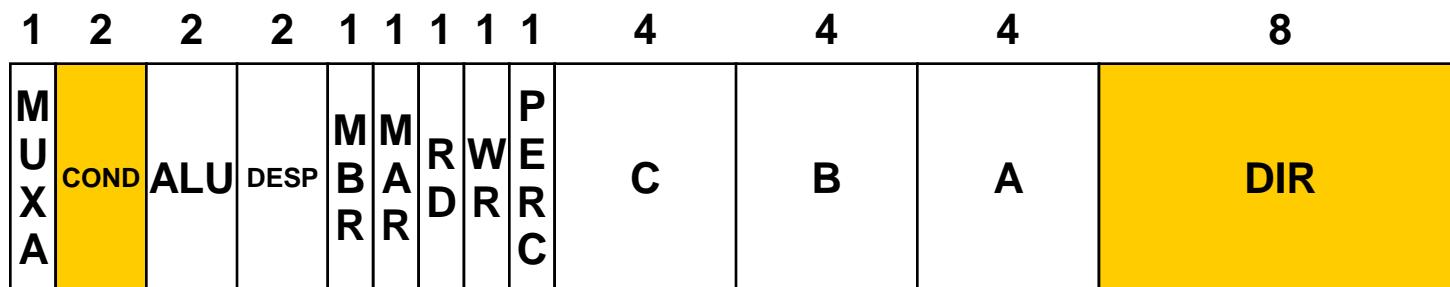
Diseño horizontal

- **Es necesario añadir otras señales:**
 - Nos puede interesar generar N y Z sin almacenar el resultado, o almacenarlo sólo en MBR. Necesitamos un bit adicional PERC (permiso de C) para que se almacene el bus C en un registro (PERC = 1) o no (PERC = 0).
- **Nos quedan 22 bits de control.**

Diseño horizontal

■ Formato de microinstrucción (32 bits):

- 22 bits de control
- 2 bits de condición de salto
- 8 bits de dirección

**MUXA**

0 = Buffer de A

1 = MBR

COND

0 = No salta

1 = Salta si N = 1

2 = Salta si Z = 1

3 = Salta siempre

ALU

0 = A + B

1 = A & B

2 = A

3 = No A

DESP

0 = No desplaza

1 = Desplaza 1 bit a la dcha.

2 = Desplaza 1 bit a la izda.

3 = (no utilizado)

A, B, C

Selección de uno de los 16 registros

DIR

Dirección de salto en la memoria de control

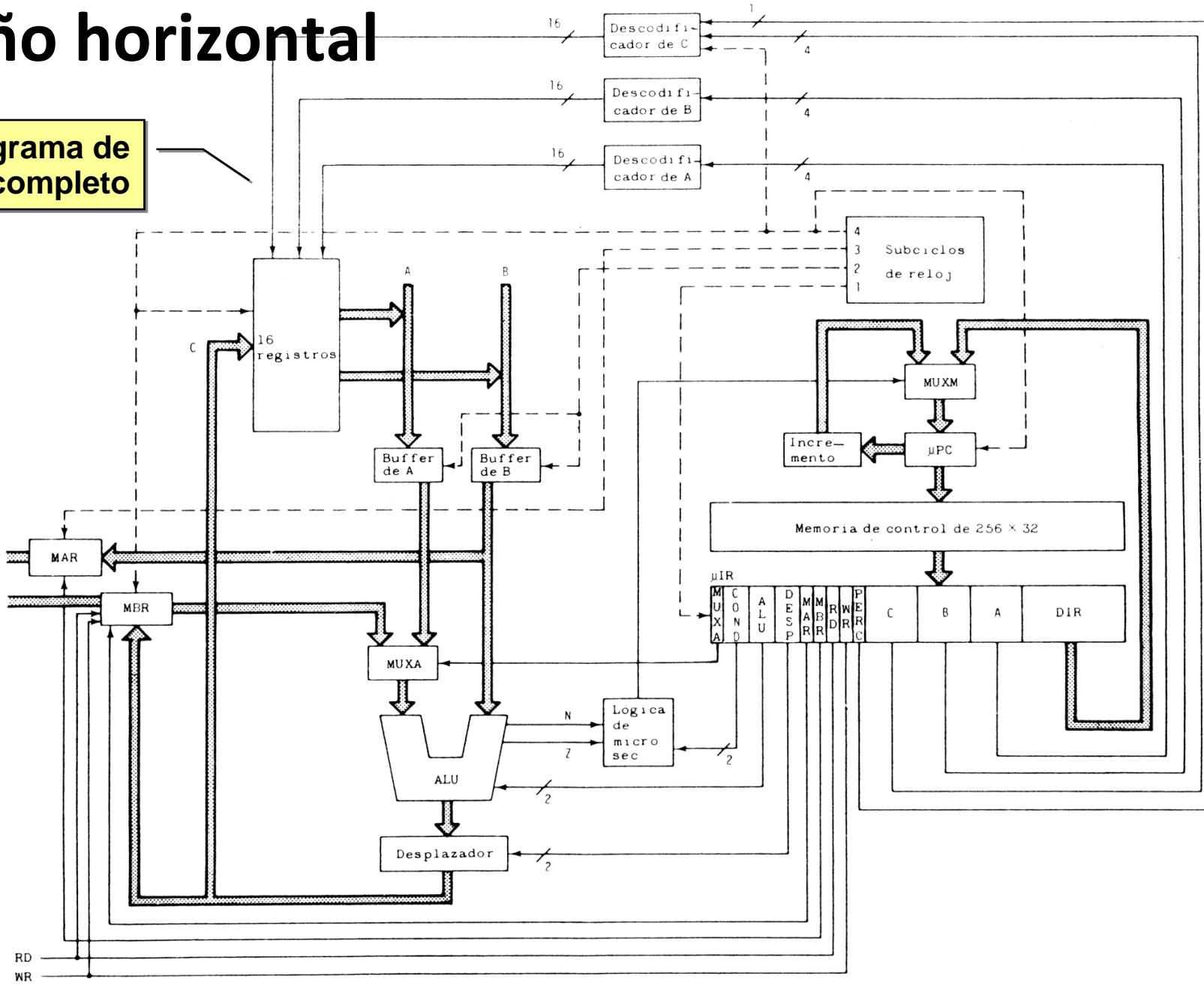
MBR, MAR, RD, WR, PERC

0 = No

1 = Sí

Diseño horizontal

Diagrama de bloques completo



Diseño horizontal

■ Unidad de control

- Memoria de control: 256 palabras de 32 bits = 8192 bits
- La unidad de incremento calcula $\mu\text{PC} + 1$
- Un ciclo de memoria principal dura dos microinstrucciones.
 - Las dos señales que controlan la memoria, RD y WR están activas mientras estén presentes en el μIR .
 - Si una microinstrucción comienza una lectura de memoria poniendo RD = 1, también debe ser RD = 1 en la siguiente microinstrucción.
- La elección de la siguiente microinstrucción la realiza la lógica de microsecuenciamiento durante el subciclo 4 (cuando N y Z son válidos), a partir de N, Z y los dos bits COND ($I = \text{Izdo.}, D = \text{dcho.}$):
 - $\text{MUXM} = /I \cdot D \cdot N + I \cdot /D \cdot Z + I \cdot D = D \cdot N + I \cdot Z + I \cdot D$

Diseño horizontal

■ Arquitectura (1)

- Memoria principal: 4096 palabras de 16 bits
- 3 registros visibles por el programador de leng. máquina:
 - PC : contador de programa
 - SP : puntero de pila
 - AC : acumulador
- 3 modos de direccionamiento:
 - Directo: los 12 bits menos significativos son una dirección de memoria
 - Indirecto: AC contiene una dirección de memoria
 - Local
 - los 12 bits menos significativos son un desplazamiento que se suma al puntero de pila
 - para direccionar variables locales de procedimientos

Diseño horizontal

■ Arquitectura (2)

- La **pila** crece hacia direcciones de memoria menores
- La **E/S** es mapeada en memoria → no hay instrucciones de E/S específicas
- El registro **AMASK** ($0FFF_{16}$) se utiliza para obtener el campo dirección en las instrucciones de direccionamiento directo y local
- El registro **SMASK** ($00FF_{16}$) se utiliza para obtener el incremento/decremento del puntero de pila en las instrucciones INSP y DESP

Diseño horizontal

■ Arquitectura (3): repertorio de 23 instrucciones máquina

Binario	Nemotécnico	Instrucción	Significado
0000xxxxxxxxxxxx	LODD	Carga directa	$ac := m[x]$
0001xxxxxxxxxxxx	STOD	Almacenamiento directo	$m[x] := ac$
0010xxxxxxxxxxxx	ADDD	Suma directa	$ac := ac + m[x]$
0011xxxxxxxxxxxx	SUBD	Resta directa	$ac := ac - m[x]$
0100xxxxxxxxxxxx	JPOS	Salto si positivo	if $ac \geq 0$ then $pc := x$
0101xxxxxxxxxxxx	JZER	Salto si cero	if $ac = 0$ then $pc := x$
0110xxxxxxxxxxxx	JUMP	Salto incondicional	$pc := x$
0111xxxxxxxxxxxx	LOCO	Carga de constante	$ac := x$ ($0 \leq x \leq 4095$)
1000xxxxxxxxxxxx	LODL	Carga local	$ac := m[sp + x]$
1001xxxxxxxxxxxx	STOL	Almacenamiento local	$m[sp + x] := ac$
1010xxxxxxxxxxxx	ADDL	Suma local	$ac := ac + m[sp + x]$
1011xxxxxxxxxxxx	SUBL	Resta local	$ac := ac - m[sp + x]$
1100xxxxxxxxxxxx	JNEG	Salto si negativo	if $ac < 0$ then $pc := x$
1101xxxxxxxxxxxx	JNZE	Salto si no cero	if $ac \neq 0$ then $pc := x$
1110xxxxxxxxxxxx	CALL	Llamada a subrutina	$sp := sp - 1$; $m[sp] := pc$; $pc := x$
1111000000000000	PSHI	Apilamiento indirecto	$sp := sp - 1$; $m[sp] := m[ac]$
1111001000000000	POPI	Desapilamiento indirecto	$m[ac] := m[sp]$; $sp := sp + 1$
1111010000000000	PUSH	Apilamiento	$sp := sp - 1$; $m[sp] := ac$
1111011000000000	POP	Desapilamiento	$ac := m[sp]$; $sp := sp + 1$
1111100000000000	RETN	Retorno de subrutina	$pc := m[sp]$; $sp := sp + 1$
1111101000000000	SWAP	Intercambio de AC y SP	$tmp := ac$; $ac := sp$; $sp := tmp$
1111110yyyyyyy	INSP	Incremento de SP	$sp := sp + y$ ($0 \leq y \leq 255$)
11111110yyyyyyy	DESP	Decremento de SP	$sp := sp - y$ ($0 \leq y \leq 255$)

Diseño horizontal

■ Lenguaje para microprogramar en alto nivel:

- Los microprogramas se pueden escribir:
 - en binario: 32 bits por microinstrucción
 - nombrando cada campo distinto de 0 y su valor (una microinstrucción por línea):
 - Ej.: PERC = 1, C = 1, B = 1, A = 10
 - con instrucciones de alto nivel tipo PASCAL

- Funciones de la ALU:

`ac:=a+ac;`

`a:=band(ir,smask);`

`ac:=a;`

`a:=inv(a);`

- Desplazamiento:

`tir:=lshift(tir+tir);`

`a:=rshift (a);`

- Saltos incondicionales:

`goto 27`

- Saltos condicionales:

`if n then goto 27;`

`if z then goto 27;`

- Examen de un registro sin almacenamiento:

`alu:=tir; if n then goto 27;`

Diseño horizontal

- Algunas sentencias y sus microinstrucciones correspondientes:

M	C	D		P		D
U	O	A	E	M	M	E
X	N	L	S	B	A	R
A	D	U	P	R	R	D
						I
				C	C	B
				B	A	R

mar:=pc; rd;	0	0	2	0	0	1	1	0	0	0	0	0	0	00
rd;	0	0	2	0	0	0	1	0	0	0	0	0	0	00
ir:=mbr;	1	0	2	0	0	0	0	0	1	3	0	0	0	00
pc:=pc+1	0	0	0	0	0	0	0	0	1	0	6	0	0	00
mar:=ir; mbr:=ac; wr;	0	0	2	0	1	1	0	1	0	0	3	1	0	00
alu:=tir; if n then goto 15;	0	1	2	0	0	0	0	0	0	0	0	4	15	
ac:=inv(mbr);	1	0	3	0	0	0	0	0	1	1	0	0	0	00
tir:=lshift(tir); if n then goto 25;	0	1	2	2	0	0	0	0	1	4	0	4	25	
alu:=ac; if z then goto 22;	0	2	2	0	0	0	0	0	0	0	0	1	22	
ac:=band(ir, amask); goto 0;	0	3	1	0	0	0	0	0	1	1	8	3	00	
sp:=sp+(-1); rd;	0	0	0	0	0	0	1	0	1	2	2	7	00	
tir:=lshift(ir+ir); if n then goto 69;	0	1	0	2	0	0	0	0	1	4	3	3	69	

Diseño horizontal

■ Microprograma:

- 79 microinstrucciones de 32 bits = **2528** bits
(diapositiva siguiente)
- La decodificación de las instrucciones se realiza examinando IR bit a bit.
 - Proporción considerable de tiempo dedicada a la decodificación
 - El CPI disminuiría si se pudiera extraer el μ PC a partir del código de operación de la instrucción máquina (**goto f(ir)**)

■ Ejercicio: ¿cómo se modificaría el microprograma suponiendo que existe el tipo de salto **goto f(ir)**?

0: mar := pc ; rd;	{ciclo principal}	40: tir := lshift(tir); if n then goto 46;	{110x o 111x?}
1: pc := pc + 1; rd;	{incrementa pc}	41: alu := tir; if n then goto 44;	{1100 o 1101?}
2: ir := mbr; if n then goto 28;	{salva y descodifica mbr}	42: alu := ac; if n then goto 22;	{1100 = JNEG}
3: tir := lshift(ir + ir); if n then goto 19;	{000x o 001x?}	43: goto 0;	
4: tir := lshift(tir); if n then goto 11;	{0000 o 0001?}	44: alu := ac; if z then goto 0;	{1101 = JNZE}
5: alu := tir; if n then goto 9;	{0000 = LODD}	45: pc := band(ir, amask); goto 0;	
6: mar := ir ; rd;	{0001 = STOD}	46: tir := lshift(tir); if n then goto 50;	
7: rd;	{0010 o 0011?}	47: sp := sp + (-1);	{1110 = CALL}
8: ac := mbr; goto 0;	{0010 = ADDD}	48: mar := sp; mbr := pc ; wr;	
9: mar := ir ; mbr := ac ; wr;	{0011 = SUBD}	49: pc := band(ir , amask); wr ; goto 0;	
10: wr ; goto 0;	{Nota: x-y = x + 1 + no y}	50: tir := lshift(tir); if n then goto 65;	{1111, examina addr }
11: alu := tir; if n then goto 15;	{010x o 011x?}	51: tir := lshift(tir); if n then goto 59;	
12: mar := ir ; rd;	{0100 o 0101?}	52: alu := tir; if n then goto 56;	
13: rd;	{0100 = JPOS}	53: mar := ac ; rd;	{1111000 = PSHI}
14: ac := mbr + ac ; goto 0;	{realiza el salto}	54: sp := sp + (-1); rd;	
15: mar := ir ; rd;	{0101 = JZER}	55: mar := sp ; wr ; goto 10;	
16: ac := ac + 1; rd;	{no se produce el salto}	56: mar := sp ; sp := sp + 1; rd;	{1111001 = POPI}
17: a := inv(mbr);	{0110 o 0111?}	57: rd;	
18: ac := ac + a; goto 0;	{0110 = JUMP}	58: mar := ac ; wr ; goto 10;	
19: tir := lshift(tir); if n then goto 25;	{0111 = LOCO}	59: alu := tir; if n then goto 62;	
20: alu := tir; if n then goto 23;	{10xx o 11xx?}	60: sp := sp + (-1);	{1111010 = PUSH}
21: alu := ac ; if n then goto 0;	{100x o 101x?}	61: mar := sp ; mbr := ac ; wr ; goto 10;	
22: pc := band(ir , amask); goto 0;	{1000 o 1001?}	62: mar := sp ; sp := sp + 1; rd;	{1111011 = POP}
23: alu := ac ; if z then goto 22;	{1000 = LODL}	63: rd;	
24: goto 0;	{1001 = STOL}	64: ac := mbr ; goto 0;	
25: alu := tir ; if n then goto 27;	{010 o 1011?}	65: tir := lshift(tir); if n then goto 73;	
26: pc := band(ir , amask); goto 0;	{1010 = ADDL}	66: alu := tir ; if n then goto 70;	
27: ac := band(ir , amask); goto 0;	{1011 = SUBL}	67: mar := sp ; sp := sp + 1; rd;	{1111100 = RETN}
28: tir := lshift(ir + ir); if n then goto 40;		68: rd;	
29: tir := lshift(tir); if n then goto 35;		69: pc := mbr ; goto 0;	
30: alu := tir ; if n then goto 33;		70: a := ac ;	
31: a := ir + sp ;		71: ac := sp ;	
32: mar := a ; rd ; goto 7;		72: sp := a ; goto 0;	
33: a := ir + sp ;		73: alu := tir ; if n then goto 76;	
34: mar := a ; mbr := ac ; wr ; goto 10;		74: a := band(ir , smask);	{1111110 = INSP}
35: alu := tir ; if n then goto 38;		75: sp := sp + a ; goto 0;	
36: a := ir + sp ;		76: a := band(ir , smask);	{1111111 = DESP}
37: mar := a ; rd ; goto 13;		77: a := inv(a);	
38: a := ir + sp ;		78: a := a + 1; goto 35	
39: mar := a ; rd ; goto 16;			

Diseño vertical

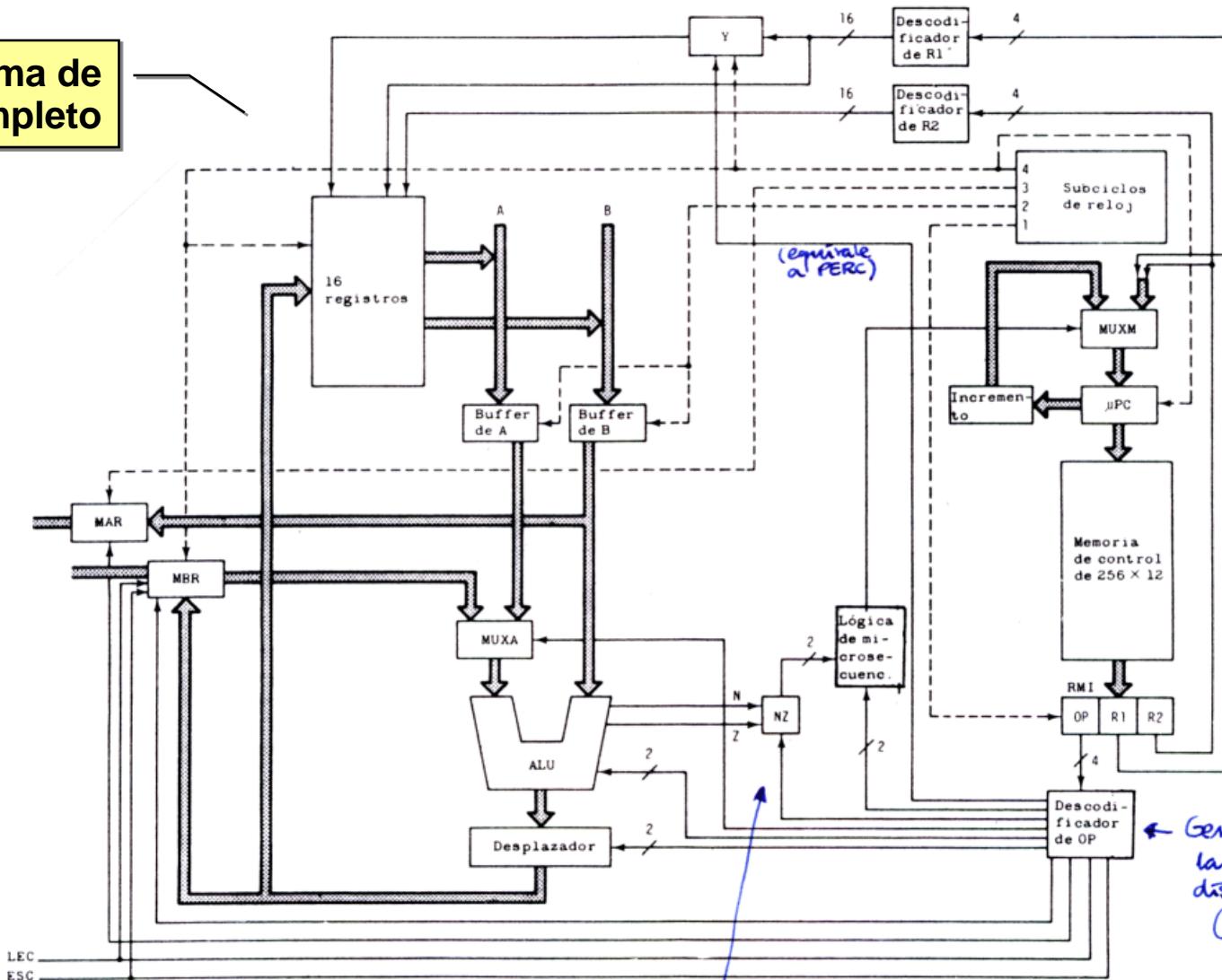
- Cada microinstrucción contiene ahora 3 campos de 4 bits:
 - OP: código de operación (dice qué hace la microinstrucción)
 - r1 y r2: dos registros (dirección en el caso de los saltos)

Binario	Nemotécnico	Instrucción	Significado
0000	ADD	Suma	$r1 := r1 + r2$
0001	AND	AND bit a bit	$r1 := r1 \& r2$
0010	MOV	Mueve reg. a reg.	$r1 := r2$
0011	NEG	Complementa	$r1 := \text{inv}(r2)$
0100	SHL	Desplaza a la izda.	$r1 := \text{lshift}(r2)$
0101	SHR	Desplaza a la dcha.	$r1 := \text{rshift}(r2)$
0110	MOV_MBR	Mueve MBR a registro	$r1 := \text{MBR}$
0111	TST	Examina registro	if $r2 < 0$ then $n := \text{true}$; if $r2 = 0$ then $z := \text{true}$
1000	LD_1	Comienza lectura	$\text{MAR} := r1; \text{RD}$
1001	ST_1	Comienza escritura	$\text{MAR} := r1; \text{MBR} := r2; \text{WR}$
1010	LD_2	Termina lectura	RD
1011	ST_2	Termina escritura	WR
1100	-	(no usado)	
1101	BN	Salta si $N = 1$	if n then goto dir
1110	BZ	Salta si $Z = 1$	if z then goto dir
1111	JMP	Salta siempre	goto dir

Diseño vertical

Se necesita Y porque el campo R1 lleva el bits A y el C.

Diagrama de bloques completo



← Genera todas las señales del diseño horizontal
(Ver siguiente figura)

Registro de 2 bits para almacenar N y Z, ya que ahora el salto condicional requiere otra microinstrucción.

Diseño vertical

■ Señales generadas por el decodificador de OP:

- Requiere una PLA con 4 entradas, 13 salidas y 15 términos producto.

Binario Nemotécnico	ALU A	ALU B	DESP A	DESP B	NZ	MUXA	Y	MAR	MBR	RD	WR	LMS A	LMS B
0000	ADD				1		1						
0001	AND		1		1		1						
0010	MOV	1			1		1						
0011	NEG	1	1		1		1						
0100	SHL	1		1	1		1						
0101	SHR	1			1	1	1						
0110	MOV_MBR	1			1	1	1						
0111	TST	1			1								
1000	LD_1	1						1		1			
1001	ST_1	1						1	1		1		
1010	LD_2	1								1			
1011	ST_2	1									1		
1100	-												
1101	BN												1
1110	BZ											1	
1111	JMP											1	1

Diseño vertical

■ Microprograma:

- 160 microinstrucciones de 12 bits = **1920** bits
(diapositiva siguiente)
- Menos memoria de control, pero UC más lenta (¿por qué? ¿cuánto más lenta (suponer todas las instrucciones equiprobables)?)

■ Ejercicio: ¿cómo se modificaría el microprograma suponiendo que existe el tipo de salto **goto f(ir)**?

```

0: mar := pc ; rd;
1: rd;
pc := pc + 1;
2: ir := mbr;
tir := lshift(ir);
if n then goto 28;
3: tir := lshift(tir);
if n then goto 19;
4: tir := lshift(tir);
if n then goto 11;
5: alu := tir;
if n then goto 09;
6: mar := ir ; rd ; {LODD}
7: rd;
8: ac := mbr;
goto 0;
9: mar := ir ; mbr := ac ; wr ; {STOD}
10: wr;
goto 0;
11: alu := tir;
if n then goto 15;
12: mar := ir ; rd ; {ADDD}
13: rd;
14: a := mbr;
ac := ac + a;
goto 0;
15: mar := ir ; rd ; {SUBD}
16: rd;
99: ac := ac + 1;
17: a := mbr;
a := inv(a);
18: ac := ac + a;
goto 0;
19: tir := lshift(tir);
if n then goto 25;
20: alu := tir;
if n then goto 23;
21: alu := ac ; {JPOS}
if n then goto 0;
22: pc := ir ;
pc := band(pc , amask );
goto 0;
23: alu := ac ; {JZER}
if z then goto 22;
24: goto 0;
25: alu := tir ;
if n then goto 27;
26: pc := ir ; {JUMP}
pc := band(pc , amask );
goto 0;
27: ac := ir ; {LOCO}
ac := band(ac , amask );
goto 0;
28: tir := lshift(tir);
if n then goto 40;
29: tir := lshift(tir);
if n then goto 35;
30: alu := tir;
if n then goto 33;
31: a := ir ; {LODL}
a := a + sp;
32: mar := a ; rd ;
rd;
ac := mbr;
goto 0;
33: a := ir ; {STOL}
a := a + sp;
34: mar := a ; mbr := ac ; wr ;
wr;
goto 0;
35: alu := tir ;
if n then goto 38;
36: a := ir ; {ADDL}
a := a + sp;
37: mar := a ; rd ;
rd;
a := mbr;
ac := ac + a;
goto 0;
38: a := ir ; {SUBL}
a := a + sp ;
39: mar := a ; rd ;
rd;
goto 99;
40: tir := lshift(tir);
if n then goto 46;
41: alu := tir ;
if n then goto 44;
42: alu := ac ; {JNEG}
if n then goto 22;
43: goto 0;
44: alu := ac ; {JNZE}
if z then goto 0;
45: pc := ir ;
pc := band(pc , amask );
goto 0;
46: tir := lshift(tir);
if n then goto 50;
47: sp := sp + (-); {CALL}
48: mar := sp ; mbr := pc ; wr ;
wr;
49: pc := ir ;
pc := band(pc , amask );
goto 0;
50: tir := lshift(tir);
if n then goto 65;
51: tir := lshift(tir);
if n then goto 59;
52: alu := tir ;
if n then goto 56;
53: mar := ac ; rd ; {PSHI}
rd;
54: sp := sp + (-);
55: a := mbr ;
mar := sp ; mbr := a ; wr ;
wr;
goto 0;
56: mar := sp ; rd ; {POPI}
57: rd ;
sp := sp + 1;
58: a := mbr ;
mar := ac ; mbr := a ; wr ;
wr;
goto 0;
59: alu := tir ;
if n then goto 62;
60: sp := sp + (-); {PUSH}
61: mar := sp ; mbr := ac ; wr ;
wr;
goto 0;
62: mar := sp ; rd ; {POP}
63: rd ;
sp := sp + 1;
64: ac := mbr ;
goto 0;
65: tir := lshift(tir);
if n then goto 73;
66: alu := tir ;
if n then goto 70;
67: mar := sp ; rd ; {RETN}
68: rd ;
sp := sp + 1;
69: pc := mbr ;
goto 0;
70: a := ac ; {SWAP}
71: ac := sp ;
72: sp := a ;
goto 0;
73: alu := tir ;
if n then goto 76;
74: a := ir ; {INSP}
a := band(a , smask );
75: sp := sp + a ;
goto 0;
76: a := ir ; {DESP}
a := band(a , smask );
77: a := inv(a );
78: a := a + 1;
sp := sp + a ;
goto 0;

```

Memoria

Estructura de Computadores
14^a y 15^a semanas

Bibliografía:

- | | |
|--------------------|---|
| [HAM03] Cap.5 | Organización de Computadores. Hamacher, Vranesic, Zaki. McGraw-Hill 2003
Signatura ESIIT/ C.1 HAM org |
| [STA8] Caps. 4 y 5 | Organización y Arquitectura de Computadores, 7 ^a Ed. Stallings. Pearson Educación, 2008.
Signatura ESIIT/ C.1 STA org |

Guía de trabajo autónomo (4h/s)

■ Lectura

- Cap. 5 Hamacher
- Caps. 4 y 5 Stallings

Bibliografía:

[HAM03] Cap.5

Organización de Computadores. Hamacher, Vranesic, Zaki. McGraw-Hill 2003

Signatura ESIIT/[C.1 HAM org](#)

[STA8] Caps.4 y 5

Organización y Arquitectura de Computadores, 7^a Ed. Stallings. Pearson Educación, 2008.

Signatura ESIIT/[C.1 STA org](#)

[

Memoria

- **Jerarquía de memoria. Concepto de localidad**
- **Memorias RAM semiconductoras. Memorias de sólo lectura. Prestaciones: velocidad, tamaño y coste**
- **Configuración y diseño de memorias utilizando varios chips**
- **Memorias asociativas**
- **Memoria cache. Influencia en las prestaciones**

Introducción

- **Las prestaciones que puede ofrecer un ordenador dependen en gran medida de:**
 - La **capacidad** de almacenamiento de memoria
 - La **velocidad** de acceso a memoria
 - La **organización** de memoria
- **Algunos objetivos a tener en cuenta en el diseño del sistema de memoria:**
 - Capacidad de almacenamiento y velocidad de acceso suficientes.
 - Coste por bit reducido.
 - Liberar a los programadores de realizar tareas de gestión de memoria.

Introducción

- Consideraremos la memoria en un sentido amplio, englobando:
 - Memoria interna: registros.
 - No hay diferencia de velocidad con la CPU.
 - Memoria caché.
 - Memoria principal.
 - La CPU no puede ejecutar directamente programas que estén más allá de la memoria principal.
 - Memoria secundaria o masiva.
 - Más lenta. Discos y cintas magnéticas, CD-ROM, etc.
- En este sentido amplio, llamaremos tiempo de acceso al tiempo que se requiere para leer (o escribir) un dato (palabra) en la memoria.

Introducción

■ Según el método de acceso las memorias se pueden clasificar en tres grupos:

- De acceso **aleatorio** (RAM). ————— **Random Access Memory**
 - El tiempo de acceso es independiente de la posición de memoria a acceder. Ej.: SRAM, DRAM, ROM, etc.
- De acceso **secuencial** (SAM). ————— **Sequential Access Memory**
 - El tiempo de acceso depende de la posición de los datos. Ej.: Cinta magnética.
- De acceso **semialeatorio o directo** (DASD). ————— **Direct Access Storage Device**
 - Principalmente discos, en los que se accede a las pistas de forma aleatoria y a los datos en las pistas de forma secuencial.

Téngase en cuenta que en SAM y DASD el tiempo de acceso a un bloque de n palabras no es $n \cdot$ Tiempo de acceso, sino:

Tiempo de acceso (a la primera palabra) + Tiempo de bloque

Aumento del ancho de banda

■ Ancho de banda de memoria de un ordenador:

- Se define como:

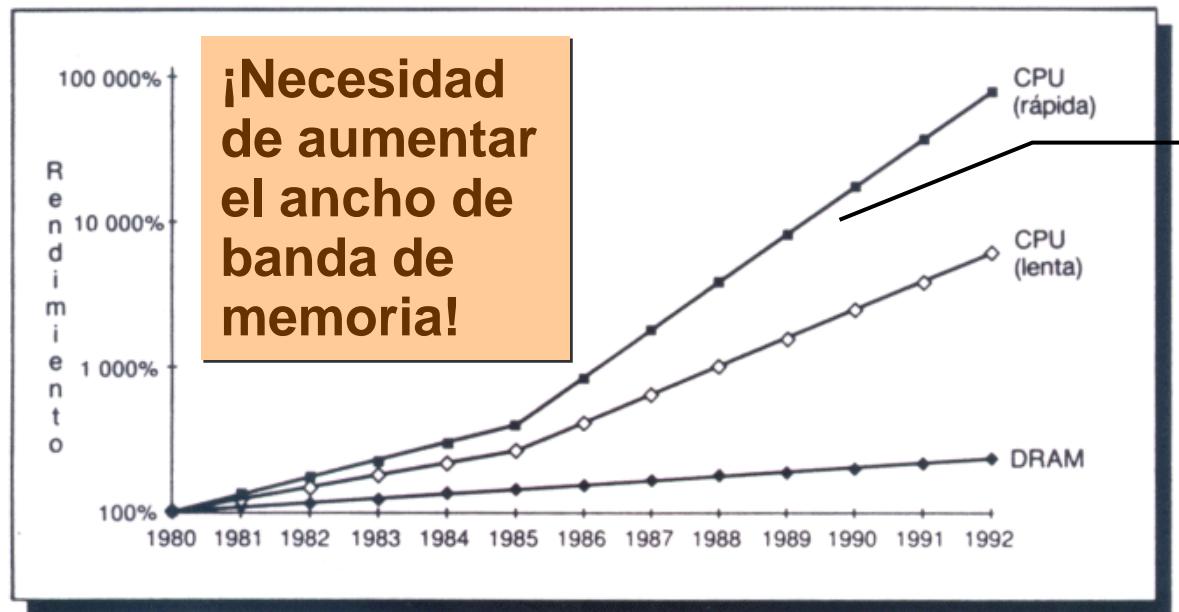
Número de palabras a las que puede acceder el procesador (o que se pueden transferir entre el procesador y la memoria) por unidad de tiempo

■ Problema:

- Diferencia entre el rendimiento del procesador y el ancho de banda de memoria (“processor-memory gap”).

Aumento del ancho de banda

- Diferencia de rendimiento entre CPU y memoria:



Ley de Moore:
“Las prestaciones del procesador se duplican cada 18 meses.”

Comenzando con el rendimiento de 1980 como punto de partida, se dibuja el rendimiento de las DRAM y CPU a lo largo del tiempo. El punto de partida de DRAM de 64 KB en 1980, con tres años para la siguiente generación. La línea de CPU lentes supone una mejora del 19 por 100 por año hasta 1985 y una mejora del 50 por 100 después. La línea de CPU rápidas supone una mejora del rendimiento del 25 por 100 entre 1980 y 1985 y un 100 por 100 por año después. Observar que el eje vertical debe estar en la escala logarítmica para registrar el tamaño del salto de rendimientos CPU-DRAM.

Aumento del ancho de banda

■ Diferencia de rendimiento entre CPU y memoria:

- Ejemplo (1994):
 - DEC Alpha 21164
 - 300 MHz
 - 4 instrucciones por ciclo
 - ⇒ 1200 MIPS (pico)
 - ⇒ Una instr. cada 0,833 ns
- Memoria DRAM 64 Mbits
 - Tiempo de acceso = 60 ns. ¡72 veces mayor!

Aumento del ancho de banda

■ Diferencia de rendimiento entre CPU y memoria:

- Ejemplo (2003):
 - AMD Athlon XP 2800+
 - 2,255 GHz
 - 9 instrucciones por ciclo
 - } **⇒ >20 GIPS (pico)**
 - } **⇒ Una instr. cada 0,049 ns**
 - Memoria DDR 333 MTransferencias/s *
- Una lectura/escritura cada 3 ns. **¡61 veces mayor!**
- (*) Téngase en cuenta que la DDR no es una DRAM sencilla, sino que utiliza mejoras de arquitectura, como el entrelazado.

Aumento del ancho de banda

■ Diferencia de rendimiento entre CPU y memoria:

■ Ejemplo (2009):

- Intel Core 2 Quad Q9650
 - 3 GHz
 - 4 núcleos
 - 4 instrucciones por ciclo / núcleo
 - Memoria DDR3 1333 MTransferencias/s *
 - Una lectura/escritura cada 0,75 ns. **¡36 veces mayor!**
 - (*) Téngase en cuenta que la DDR3 no es una DRAM sencilla, sino que utiliza mejoras de arquitectura, como el entrelazado.
- } **⇒ 48 GIPS (pico)**
} **⇒ Una instr. cada 0,02083 ns**

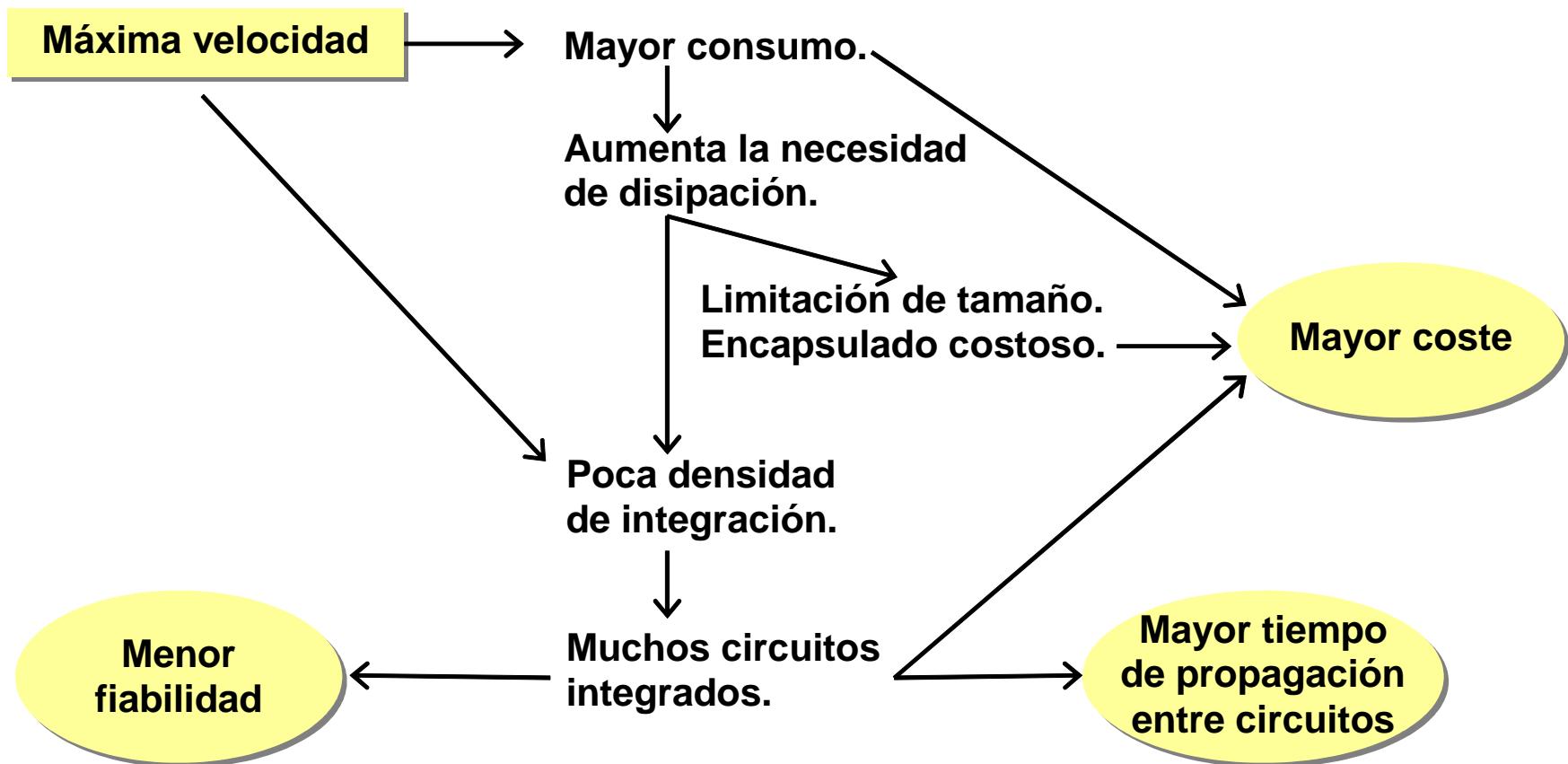
Aumento del ancho de banda

■ Métodos para aumentar el ancho de banda de memoria:

- Usar tecnología de alta velocidad.
- ✓ Organizar la memoria jerárquicamente, incluyendo memoria caché.
- ✓ Incrementar el ancho de la memoria (número de palabras accedidas simultáneamente).
- ✓ Dividir la memoria principal en un conjunto de módulos direccionables simultáneamente (memoria entrelazada).
- ✓ Emplear memorias asociativas (direcciónables por contenido) cuando sea posible.

Aumento del ancho de banda

- Disponer de memoria de la máxima capacidad con el menor tiempo de acceso y sin mejoras en la arquitectura no es sólo muy caro, sino ¡inviable!



Aumento del ancho de banda

■ Ejemplos de 1995 (memorias SRAM y DRAM clásicas sin mejoras de arquitectura):

Para cada fabricante se eligió el circuito más rápido de entre varios disponibles

(*) Demasiados chips. Ocupan mucho espacio y el retardo entre circuitos es mayor que el tiempo de acceso

Fabricante	Fecha	Memoria	Tecnolog.	T. acc.	Consumo	Nº chips	Consumo	Precio aprox.(1995)
MHS	1994	SRAM 16Kx4-Bit	BiCMOS	8 ns	Activo: 700 mW Standby: 250 mW	128 / 1 MB 2048 / 16 MB !! (*) 8192 / 64 MB !! (*)	32-89.6 W / 1 MB 512-1433 W / 16 MB !! 2048-5734.4 W / 64 MB !!	250.000 pts / 1 MB !! 4.000.000 pts / 16 MB !! 16.000.000 pts / 64 MB !!
Paradigm	1991	SRAM 32Kx8-Bit	CMOS	10 ns	Activo: 400 mW Standby: 150 mW	32 / 1MB 512 / 16MB !! (*) 2048 / 64 MB !! (*)	4.8-12.8 W / 1MB 76.8-204.8 W / 16 MB 307.2-819.2 W / 64 MB !!	65.000 pts / 1MB 1.000.000 pts / 16 MB !! 4.000.000 pts / 64 MB !!
Mosel-Vitelic	1993	DRAM 1MBx8-Bit (SIMM de 8 chips 1Mx1-Bit)	CMOS	60 ns	Activo: 3.6 W Standby: 80 mW	8 (1 SIMM) / 1MB 128 (16 SIMMs) / 16MB 512 (64 SIMMs) / 64 MB !! (*)	0.08-3.6 W / 1 MB 1.28-57.6 W / 16 MB 5.12-230.4 W / 64 MB	5.000 pts / 1 MB 80.000 pts / 16 MB 320.000 pts / 64 MB
Mosel-Vitelic	1993	DRAM 4MBx8-Bit (SIMM de 8 chips 4Mx1-Bit)	CMOS	70 ns	Activo: 3.6 W Standby: 80 mW	32 (4 SIMMs) / 16 MB 128 (16 SIMMs) / 64 MB	0.32-14.4 W / 16 MB 1.28-57.6 W / 64 MB	80.000 pts / 16 MB 320.000 pts / 64 MB
Mosel-Vitelic	1993	DRAM 4MBx32-Bit (SIMM de 32 chips 4Mx1-Bit)	CMOS	70 ns	Activo: 17.01 W Standby: 325 mW	32 (1 SIMM) / 16 MB 128 (4 SIMMs) / 64 MB	0.325-17.01 W / 16 MB 1.3-68.05 W / 64 MB	80.000 pts / 16 MB 320.000 pts / 64 MB

Localidad de las referencias

■ Aprovecha los siguientes hechos:

▪ Regla 90/10

- Una regla empírica ampliamente corroborada es que un programa pasa el 90% de su tiempo de ejecución en sólo el 10% de su código.

▪ Localidad de las referencias

- Espacial
- Secuencial
- Temporal

Basándonos en el pasado reciente de un programa, podemos predecir con una precisión razonable qué instrucciones y datos utilizará en un futuro próximo.

Localidad de las referencias

■ Localidad de las referencias:

- Localidad **espacial** $d(t+n) = d(t) + k$, con n y k pequeños.

- “Si se referencia un elemento, los elementos cercanos a él serán referenciados pronto.”
 - Justificación: Los datos relacionados se almacenan juntos.

- Localidad **secuencial** $d(t+1) = d(t) + 1$

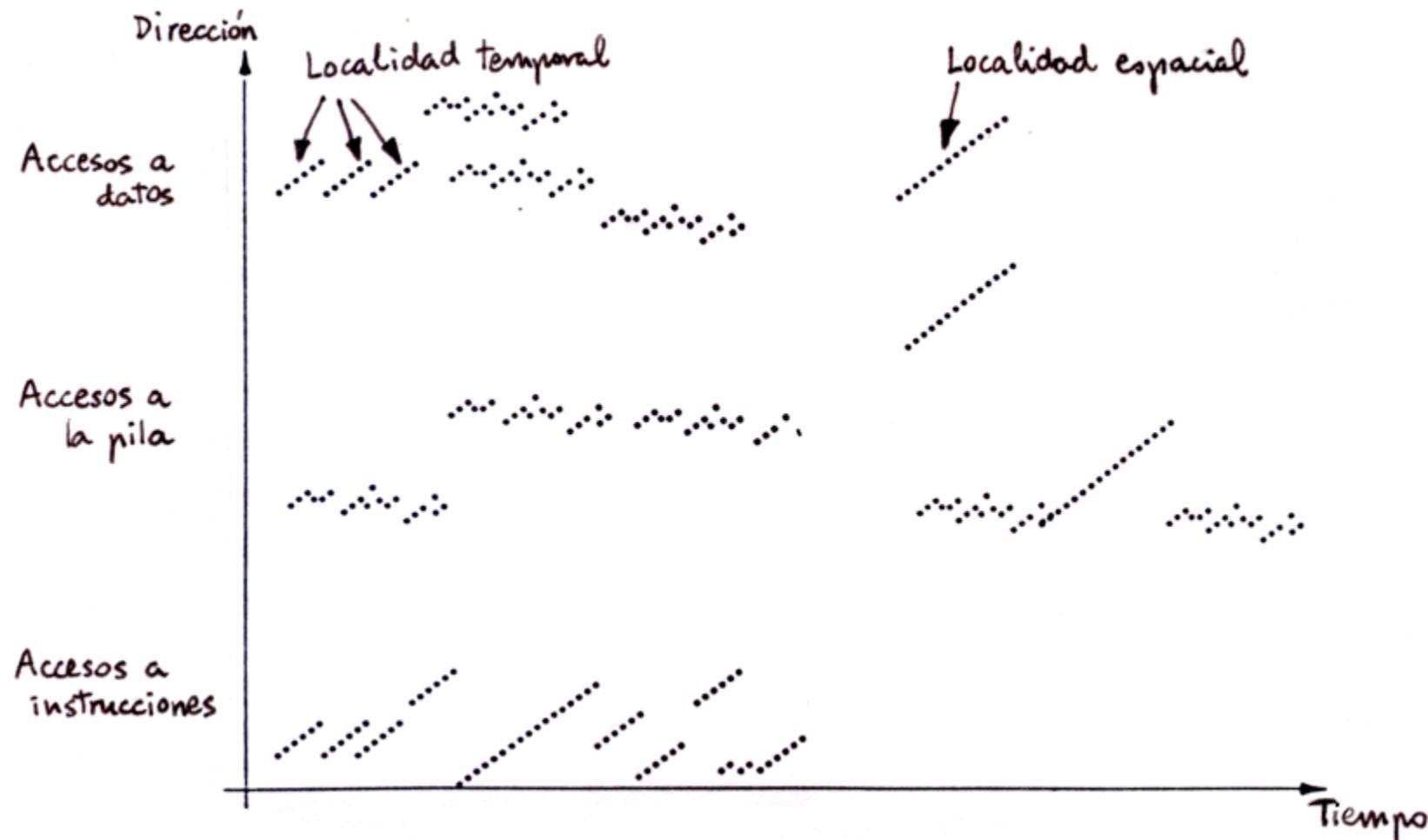
- “Las direcciones de memoria utilizadas suelen ser contiguas.”
 - Justificación: Las instrucciones se ejecutan secuencialmente.

- Localidad **temporal** $d(t+n) = d(t)$, con n pequeño.

- “Si se referencia un elemento, volverá a referenciarse pronto.”
 - “La información que se usará en un futuro próximo es aproximadamente la misma que se está usando actualmente.”
 - Justificación: Bucles, uso de datos de forma repetitiva, llamadas repetidas a subrutinas.

Localidad de las referencias

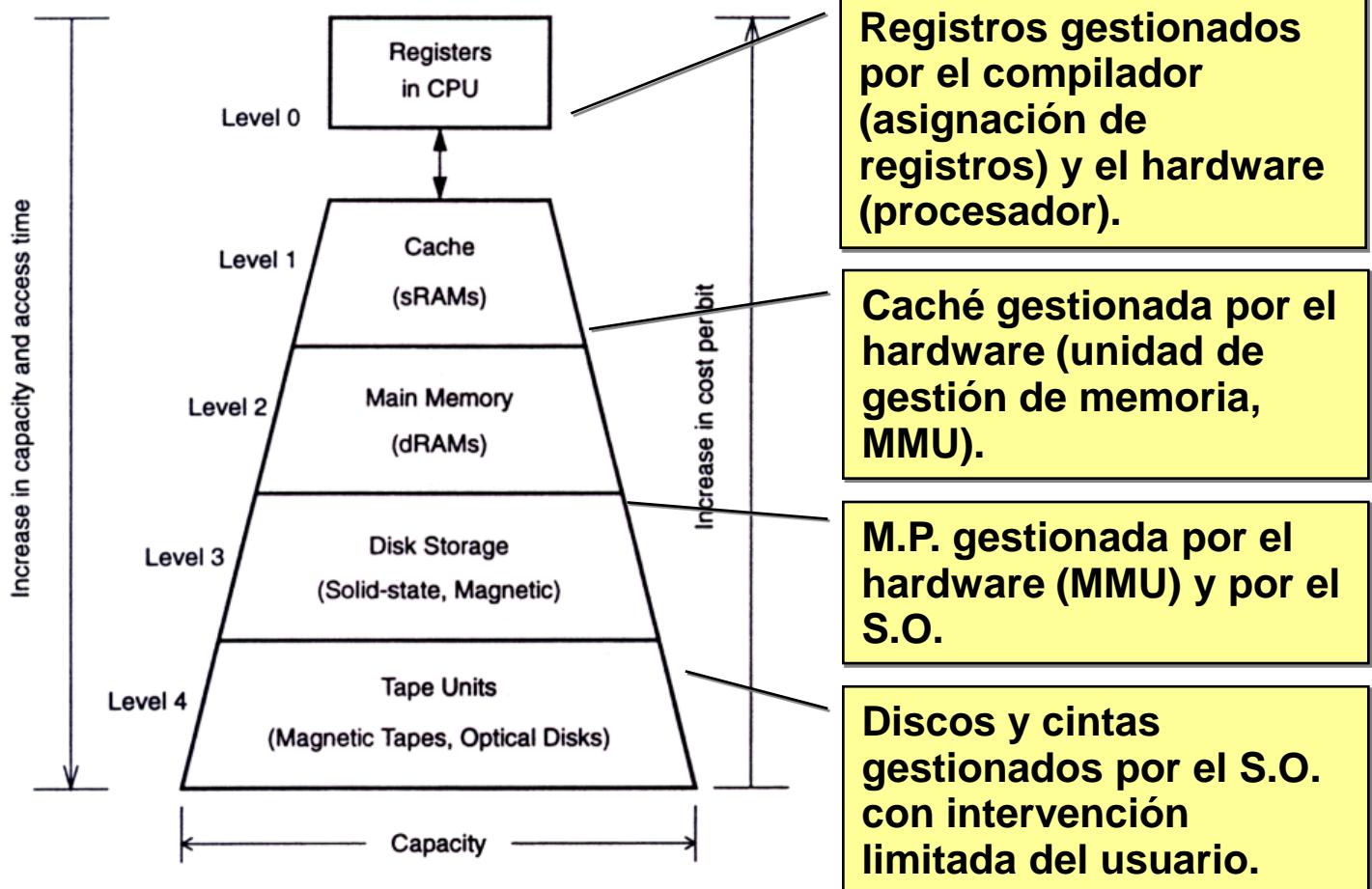
■ Patrones de referencia a memoria típicos:



Jerarquía de memoria

- Teniendo en cuenta las limitaciones tecnológicas, y aprovechando el principio de localidad se puede conseguir un sistema de memoria eficaz mediante una **jerarquía de niveles de memoria**.

- Cada nivel tiene una capacidad menor pero es más rápido que los inferiores.
- Toda la información de un nivel se encuentra almacenada también en el siguiente.



Jerarquía de memoria

■ Parámetros que caracterizan cada nivel i

- Los dispositivos de almacenamiento (registros, memorias, discos y unidades de cinta), se caracterizan por:
 - **Tiempo de acceso (t_i):**
 - Tiempo desde que se inicia una lectura hasta que llega la palabra deseada.
 - **Tamaño de la memoria (s_i):**
 - Número de bytes, palabras, sectores, etc., que se pueden almacenar en el dispositivo de memoria.
 - **Coste por bit o por byte (c_i):**
 - **Ancho de banda (b_i):**
 - Velocidad a la que se transfiere información desde un dispositivo.
 - **Unidad de transferencia (x_i):**
 - Tamaño de la unidad de información que se transfiere entre el nivel i y el $i+1$.

Jerarquía de memoria

- Características de los distintos niveles de memoria en un ordenador tipo “mainframe” de 1993:

Nivel de memoria Características	Nivel 0 Registros CPU	Nivel 1 Cache	Nivel 2 Memoria principal	Nivel 3 Almac. en disco	Nivel 4 Almac. en cinta
Tecnología del dispositivo	ECL	SRAM 256 Kbit	DRAM 4 Mbit	Unidad de disco magnético de 1 Gbyte	Unidad de cinta magnética de 5 Gbyte
Tiempo de acceso t_i	10 ns	25-40 ns	60-100 ns	12-20 ms	2-20 min (tiempo de búsqueda)
Capacidad en bytes si	512 bytes	128 Kbytes	512 Mbytes	60-228 Gbytes	512 Gbytes-2 Tbytes
Coste c_i en centavos/KB	18000	72	5,6	0,23	0,01
Coste c_i en €/KB (1 \$ = 0,8247 €)	148,45 €	59,38 cents.	4,62 cents.	0,19 cents.	0,01 cents.
Coste total en €	74,22 €	76,00 €	24.213,30 €	119.336,97 €	44.275,74 €
Ancho de banda b_i (en MB/s)	400-800	250-400	80-133	3-5	0,18-0,23
Unidad de transferencia x_i	4-8 bytes por palabra	32 bytes por bloque	0,5-1 Kbytes por página	5-512 Kbytes por fichero	Almacenamiento de seguridad

Se verifica que:

$$t_i < t_{i+1}$$

$$s_i < s_{i+1}$$

$$c_i > c_{i+1}$$

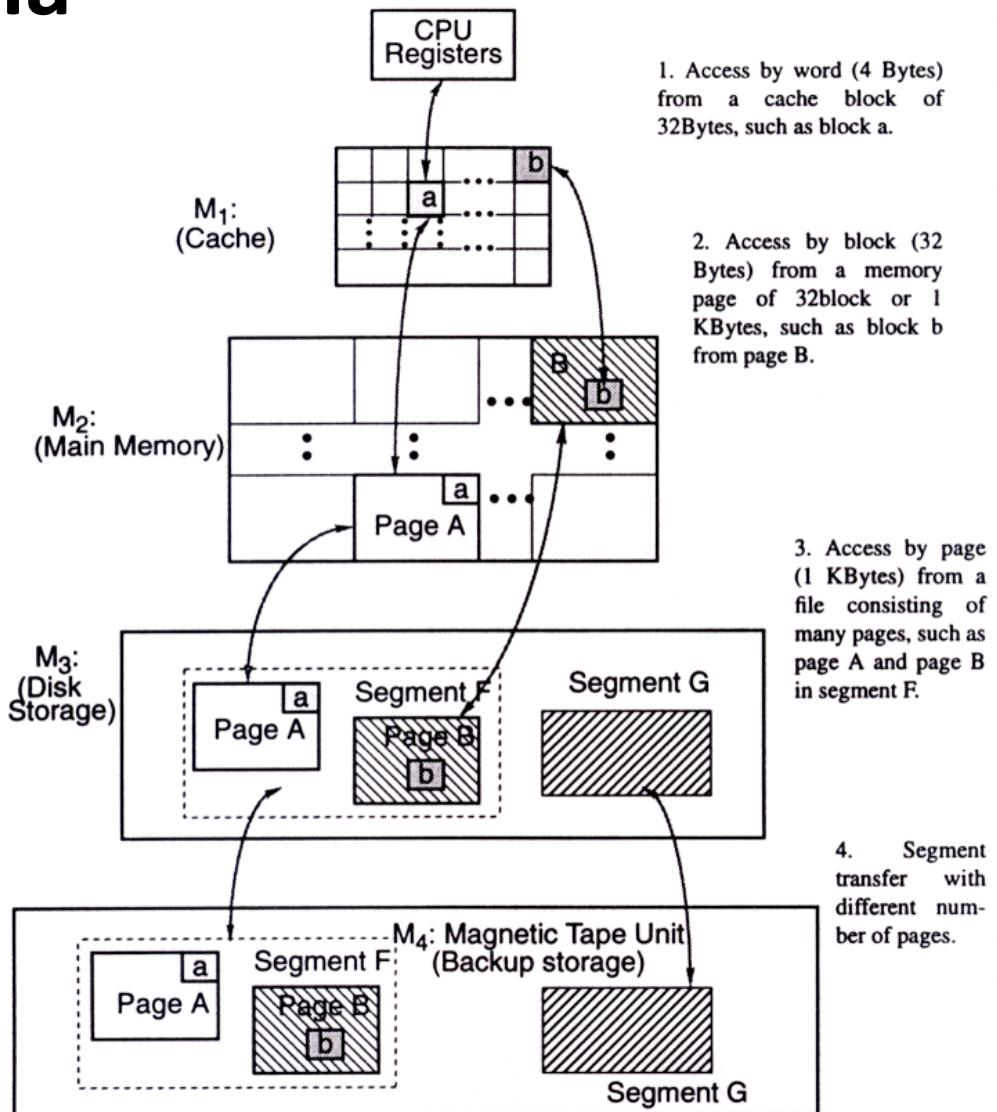
$$b_i > b_{i+1}$$

$$x_i < x_{i+1}$$

Jerarquía de memoria

■ Propiedad de inclusión

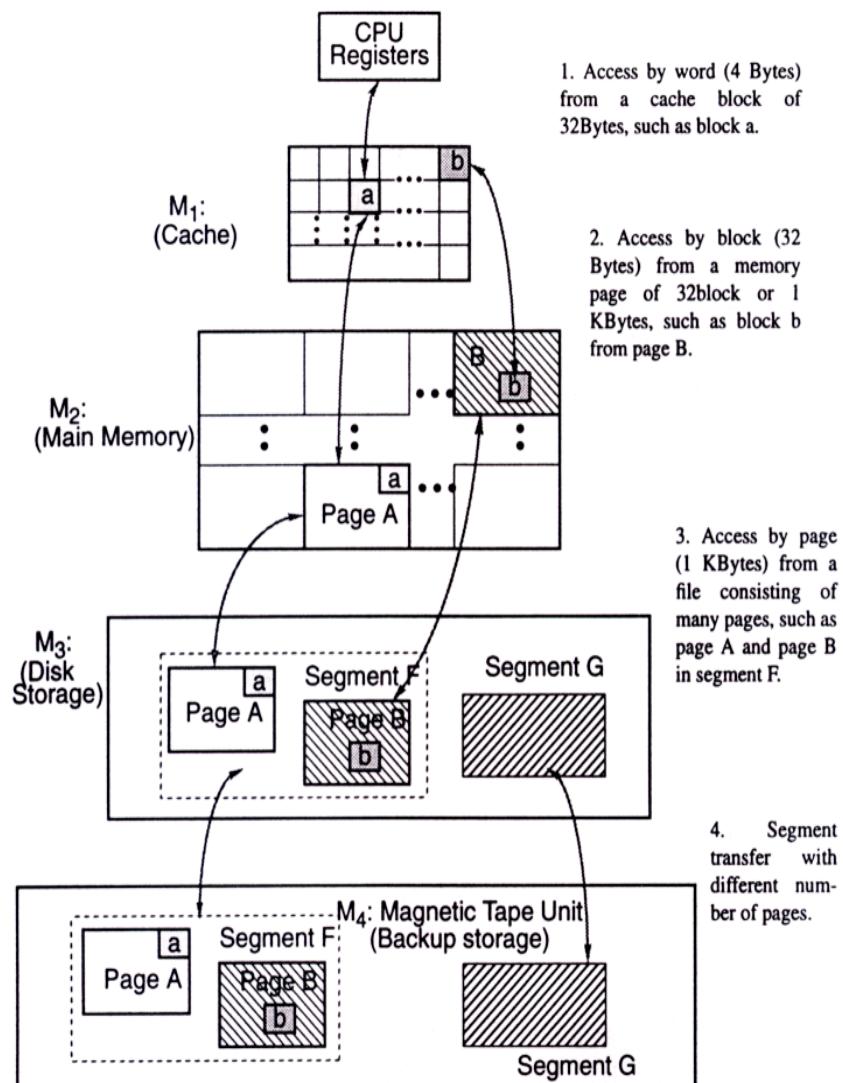
- $M_1 \subset M_2 \subset M_3 \subset \dots \subset M_n$
- Si una palabra se encuentra en $M_i \Rightarrow$ copias de esa palabra también se encuentran en $M_{i+1}, M_{i+2}, \dots, M_n$.
- Sin embargo, una palabra almacenada en M_{i+1} puede no estar en M_i , y si es así, tampoco estará en $M_{i-1}, M_{i-2}, \dots, M_1$.



Jerarquía de memoria

Lectura

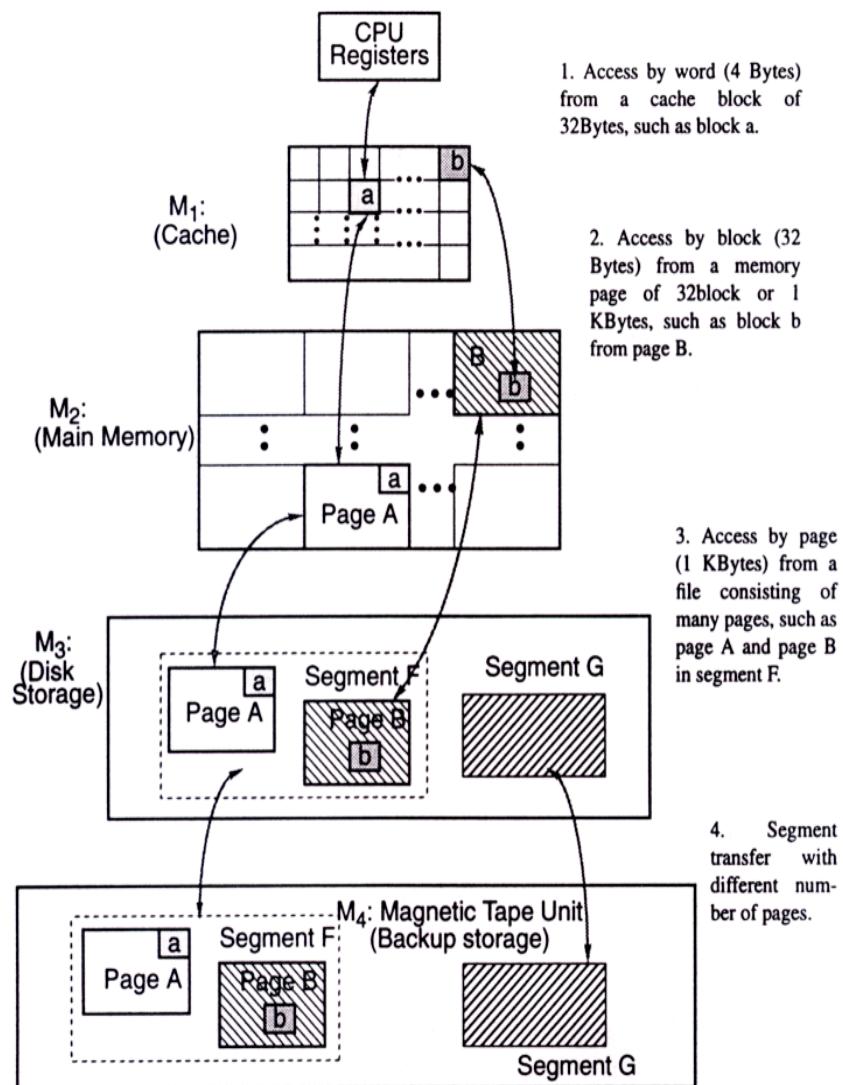
- No toda la información que necesita la CPU está en M_1 .
- Si la palabra deseada no se encuentra en el nivel 1 \Rightarrow se intentará localizarla en los niveles inferiores \Rightarrow Penalización en el tiempo de acceso.
- Supóngase que está en $M_j, j > 1$. Se transferirá a M_{j-1} , luego a M_{j-2} , y así sucesivamente hasta llegar al nivel 1, si bien pueden existir “atajos”.
- Para ahorrar tiempo y aumentar la eficiencia de las transferencias, y aprovechando el principio de localidad, se transfieren bloques completos en lugar de transferir sólo la palabra requerida.



Jerarquía de memoria

■ Escritura

- Se da un problema de consistencia o coherencia de datos entre los distintos niveles.
- Si una palabra se modifica en un nivel superior, más tarde o más temprano tiene que escribirse en niveles inferiores.
- En general se usan dos estrategias para mantener la coherencia:
- **WRITE-THROUGH (Escritura directa):**
 - Si se modifica una palabra en $M_i \Rightarrow$ se modifica inmediatamente en M_{i+1} .
- **WRITE-BACK (Post-escritura):**
 - Se retrasa la actualización en M_{i+1} hasta que la palabra modificada en M_i sea reemplazada o borrada de M_i .



Modelo de evaluación de la jerarquía

■ Modelo de evaluación del rendimiento de una jerarquía de memoria (C. K. Chow, 1974*)

- Se suele considerar que la política de administración de memoria viene caracterizada por una función de éxito o tasa de aciertos A .
- Tasa de aciertos A_i (*hit ratio*)
 - Porcentaje de información buscada en un ciclo de memoria que está presente en el nivel i .
 - En una jerarquía con n niveles:
 - $A_0 = 0$
 - $A_n = 1$
 - A_i depende de:
 - Capacidad del nivel i (s_i).
 - Granularidad de la transferencia de información.
 - Estrategia de administración de memoria.

Modelo de evaluación de la jerarquía

- Tasa de fallos F_i : (*miss ratio*)
 - $F_i = 1 - A_i, i=0, \dots, n.$
 - $F_0 = 1$
 - $F_n = 0$
- Frecuencia de accesos con éxito al nivel i a_i :
 - Probabilidad de acceder con éxito a una información en el nivel i y que esa información no se encuentre en los niveles 0 a $i-1$.
$$\sum_{i=1}^n a_i = 1$$
 - Dado que la información de M_j está también en $M_k, k > j$:
 - $a_i = A_i - A_{i-1}, i = 1, \dots, n$
 - $a_1 = A_1$
 - $a_n = 1 - A_{n-1}$

Modelo de evaluación de la jerarquía

- Los objetivos al diseñar una memoria con n niveles son:
 - ① Obtener un rendimiento cercano al de la memoria M_1 (la más rápida).
 - ② Obtener un coste por bit cercano al de la memoria M_n (la más barata).

① Rendimiento:

- Se puede medir por el tiempo medio de acceso de la jerarquía para cada referencia a memoria ().

Tiempo de acceso efectivo del procesador al nivel i -ésimo de la jerarquía:

$$T_i = \sum_{j=1}^i t_j$$

t_j = tiempo de acceso medio individual del nivel j

$$\bar{T} = \sum_{i=1}^n a_i T_i$$

Sumatoria de la frecuencia de acceso con éxito a M_i , multiplicada por el tiempo de acceso efectivo a M_i ,

Modelo de evaluación de la jerarquía

$$\bar{T} = \sum_{i=1}^n a_i T_i$$

- Sustituyendo a_i y T_i :

$$\begin{aligned}
 \bar{T} &= \sum_{i=1}^n \left[(A_i - A_{i-1}) \sum_{j=1}^i t_j \right] = (A_1 - A_0)t_1 + (A_2 - A_1)(t_1 + t_2) + (A_3 - A_2)(t_1 + t_2 + t_3) + \\
 &\quad + \dots + (A_n - A_{n-1})(t_1 + t_2 + t_3 + \dots + t_n) = \\
 &= (A_1 - A_0 + A_2 - A_1 + A_3 - A_2 + \dots + A_n - A_{n-1})t_1 + \\
 &\quad + (A_2 - A_1 + A_3 - A_2 + \dots + A_n - A_{n-1})t_2 + \\
 &\quad + (A_3 - A_2 + \dots + A_n - A_{n-1})t_3 + \\
 &\quad + \dots + (A_n - A_{n-1})t_n = \\
 &= \sum_{i=1}^n (A_n - A_{i-1})t_i = \sum_{i=1}^n (1 - A_{i-1})t_i = \\
 &= \sum_{i=1}^n F_{i-1}t_i
 \end{aligned}$$

Modelo de evaluación de la jerarquía

■ ②Coste por bit:

- El coste por bit promedio $c(n)$ de un sistema de n niveles es:

$$c(n) = \frac{\sum_{i=1}^n c_i s_i}{\sum_{i=1}^n s_i} + c_0$$

The equation is annotated with three yellow boxes:

- A box around $\sum_{i=1}^n c_i s_i$ labeled **coste total**.
- A box around $+ c_0$ labeled **coste de interconexión entre niveles**.
- A box around $\sum_{i=1}^n s_i$ labeled **tamaño total**.

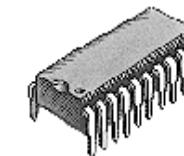
Problemas a resolver

- **Problemas a resolver en cualquier nivel de la jerarquía de memoria:**
 - **Colocación** o ubicación de bloques
 - ¿Dónde se ubicará un bloque en el nivel i cuando se transfiere a éste desde el $i+1$?
 - **Identificación** de bloques
 - ¿Cómo se encuentra un bloque en el nivel i ?
 - **Sustitución** o **reemplazo** de bloques
 - Si se produce un fallo en el nivel i y hay que traer un nuevo bloque desde M_{i+1} , y suponiendo que M_i está llena, ¿qué bloque del nivel i se reemplaza?
 - **Estrategia de escritura**
 - ¿Qué acciones hay que llevar a cabo si una escritura modifica un bloque en el nivel superior?

Memoria

- **Jerarquía de memoria. Concepto de localidad**
- **Memorias RAM semiconductoras. Memorias de sólo lectura. Prestaciones: velocidad, tamaño y coste**
- **Configuración y diseño de memorias utilizando varios chips**
- **Memorias asociativas**
- **Memoria cache. Influencia en las prestaciones**

Memorias de semiconductor



- Todas son de acceso aleatorio
- Tipos de memoria:
 - De sólo lectura o **ROM** (*Read Only Memory*):
 - ROM, PROM, EPROM, EEPROM.
 - De lectura y escritura o **RAM** (*Random Access Memory*):
 - SRAM, DRAM.

ROM

■ De sólo lectura o ROM:

■ **ROM** (*Read Only Memory*)

- La información se graba en el proceso de fabricación mediante máscaras ⇒ no puede alterarse ni borrarse.
- Razonable económicamente sólo para muchos chips.
- Ej.: *firmware* en una calculadora, BIOS de VGA, ROM de autoarranque, ROM para microcontroladores, etc.

■ **PROM** (*Programmable ROM*)

- La información se graba en un proceso posterior irreversible ⇒ programable una sola vez.
 - Los chips se fabrican con todas las celdas conectadas (a 1) y la programación (puesta a 0) consiste en romper contactos haciendo pasar una corriente elevada.
- ✖ Tecnología bipolar ⇒ alto consumo y poca densidad de integración ⇒ obsoleta desde finales de los 80.

ROM



■ **EPROM (Erasable Programmable ROM)**

- Se pueden borrar exponiéndolas a radiación ultravioleta ⇒ programable varias veces (100 a 1000).
 - Tienen una ventana de cristal de cuarzo para permitir el borrado, que se suele tapar con un adhesivo una vez programada para evitar el borrado accidental debido a la exposición prolongada al sol.
- Los datos permanecen de 10 a 100 años.
- Ejs.: Memoria de programa para un microcontrolador, BIOS.
- **OTPROM (One Time Programmable ROM)**
 - EPROM sin ventana
 - ✓ más barata debido al empaquetado
 - ✗ programable sólo la primera vez

ROM

■ EEPROM (*Electrically Erasable Programmable ROM*)

- Se programan **byte a byte** mediante corrientes elevadas.



Programador de EEPROM

- ✓ Ventaja sobre EPROM: reprogramable en la misma placa.
- Retención de datos: 10 a 100 años.
- No puede considerarse memoria RAM:
 - Reprogramable muchas, pero limitadas, veces
 - » 10 000 a >1 000 000
 - Borrado y programación lentos (5-10 ms)
- Ejs.: almacenamiento de números en teléfono o fax, almacenamiento de configuración en un monitor.

ROM

■ Memoria FLASH

- EEPROM que se puede reprogramar **por bloques** (toda la memoria o un bloque se borra a la vez).
 - Mayor densidad.
 - Menor precio por bit.
- Retención de datos: >20 años.
- Ejs.: Discos de estado sólido USB (*pen-drives*), tarjetas de memoria, BIOS.



SRAM

■ De lectura y escritura o RAM:

- Estáticas o **SRAM** (*Static Random Access Memory*).
 - **Async. SRAM, Sync. SRAM, Pipeline Burst SRAM.**
 - Para caché
 - » por velocidad.
 - Para memoria principal de microcontroladores
 - » por requerir poca memoria,
 - » por no merecer la pena añadir la circuitería extra que requiere una DRAM.

...

SRAM

...

- **CAM** (*Content Addressable Memory*).
 - Memorias asociativas. *Las estudiaremos más adelante.*
- **NOVRAM** (*Non-Volatile RAM*):
 - SRAM con una pila de litio incorporada (retención: 5 a 10 años).
 - O bien híbrido entre SRAM y EEPROM.
 - » Cada celda SRAM tiene una celda EEPROM asociada.
 - » Al desconectarse la alimentación se copia el contenido de la SRAM en la EEPROM, y viceversa al conectarse.

...

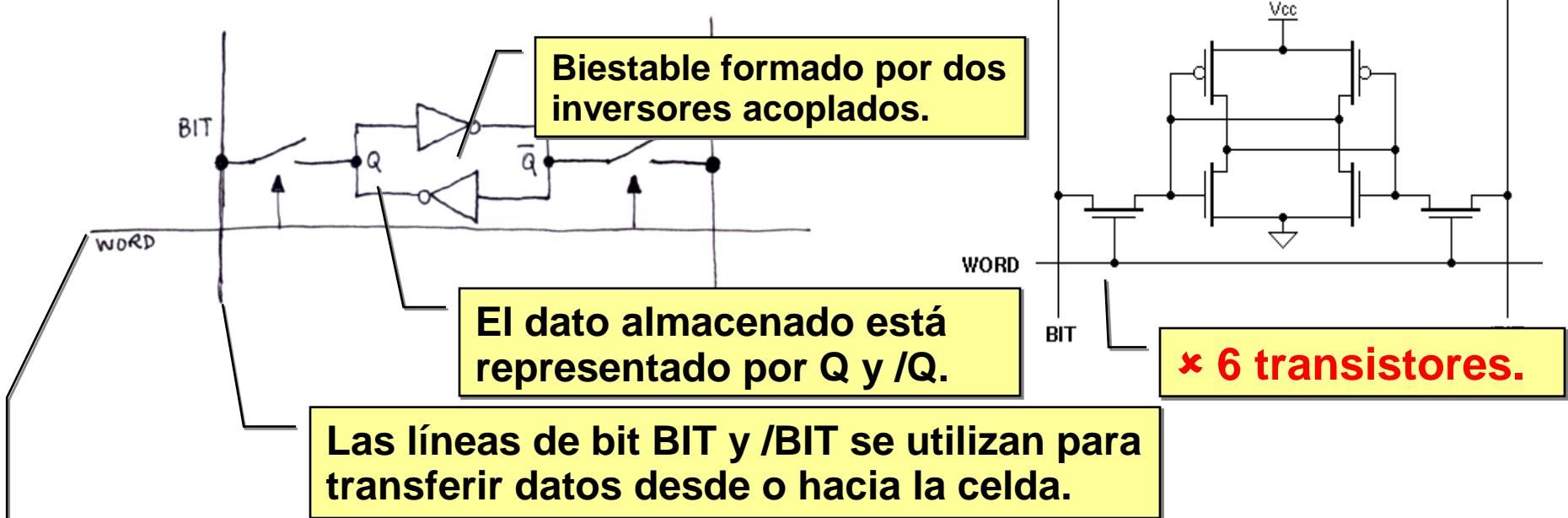
DRAM

- **Dinámicas**, o con refresco, o **DRAM** (*Dynamic RAM*).
 - Para memoria principal:
 - **PSRAM** (*Pseudo Static RAM*)
 - **FPM DRAM** (*Fast Page Mode*)
 - **Static Column DRAM**
 - **Nibble mode DRAM**
 - **DRAM EDO** (*Extended Data Out*)
 - **Burst DRAM** o **DRAM BEDO** (*Burst Extended Data Out*)
 - **EDRAM** (*Enhanced DRAM*)
 - **SDRAM** (*Synchronous DRAM*)
 - **SDRAM II** o **DDR SDRAM** (*Double Data Rate*)
 - **ESDRAM** (*Enhanced Synchronous DRAM*)
 - **RDRAM** (*Rambus DRAM*) o **DRDRAM** (*Direct Rambus DRAM*)
 - **DDR-II** y **DDR-III**
 - Para memoria de vídeo (tarjetas gráficas):
 - **VRAM** (*Video RAM*).
 - **WRAM** (*Window RAM*).
 - **SGRAM** (*Synchronous Graphic RAM*).

SRAM

■ Celda de memoria SRAM

- Estática ⇒ los datos almacenados se mantienen por un tiempo indefinido si hay alimentación.



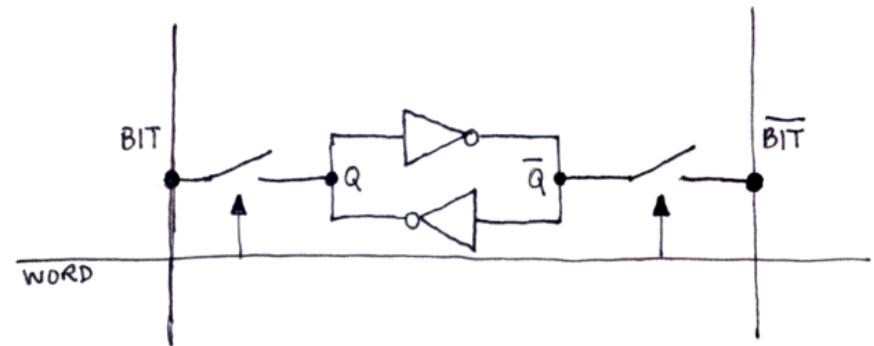
- Cuando la línea WORD esté a 1 se selecciona la celda para lectura o escritura.
- Cuando la línea WORD esté a 0 los dos transistores asociados no conducen, desconectando la celda de las líneas de bit.

SRAM

■ Celda de memoria SRAM

■ Operación de lectura:

- BIT y /BIT se ponen a 1 “débil”.
- Se selecciona la celda poniendo WORD a 1 \Rightarrow Q se conecta a BIT y /Q a /BIT.
- Si Q = 1 ($/Q = 0$) \Rightarrow la línea /BIT se pone a 0.
 - BIT = 1 y /BIT = 0 \Rightarrow se lee un 1.
- Si Q = 0 ($/Q = 1$) \Rightarrow la línea BIT se pone a 0.
 - BIT = 0 y /BIT = 1 \Rightarrow se lee un 0.



✓ Lectura no destructiva.

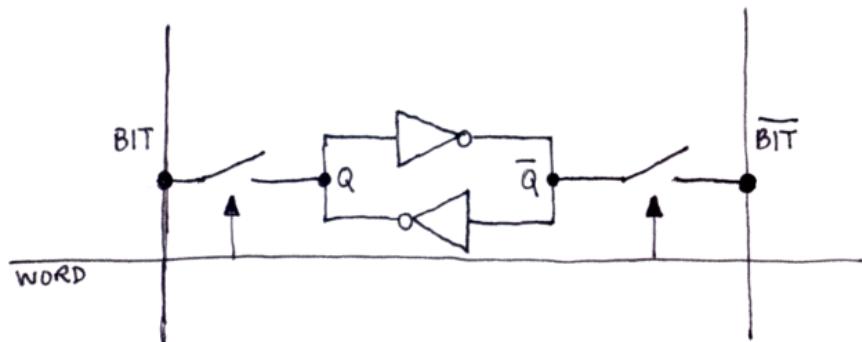
✓ Mayor velocidad que DRAM.

SRAM

■ Celda de memoria SRAM

■ Operación de escritura:

- Se requieren circuitos de control que fuercen las líneas de bit a valor 0 ó 1 “fuertes”.
- Se selecciona la celda poniendo WORD a 1, igual que en la lectura.
- Si se desea escribir un 1 y actualmente $Q = 0$ ($/Q = 1$), los circuitos de control aplican un 1 “fuerte” en BIT y un 0 “fuerte” en $/BIT$.
 - Esta combinación de entradas hace que el biestable de almacenamiento cambie de estado ($Q = 1$, $/Q = 0$). Este estado permanecerá en la celda cuando sea $WORD = 0$.
- Si la celda ya almacenaba un 1, escribir un 1 no afecta a la celda.
- Para escribir un 0, BIT y $/BIT$ se fuerzan a 0 y 1, respectivamente.

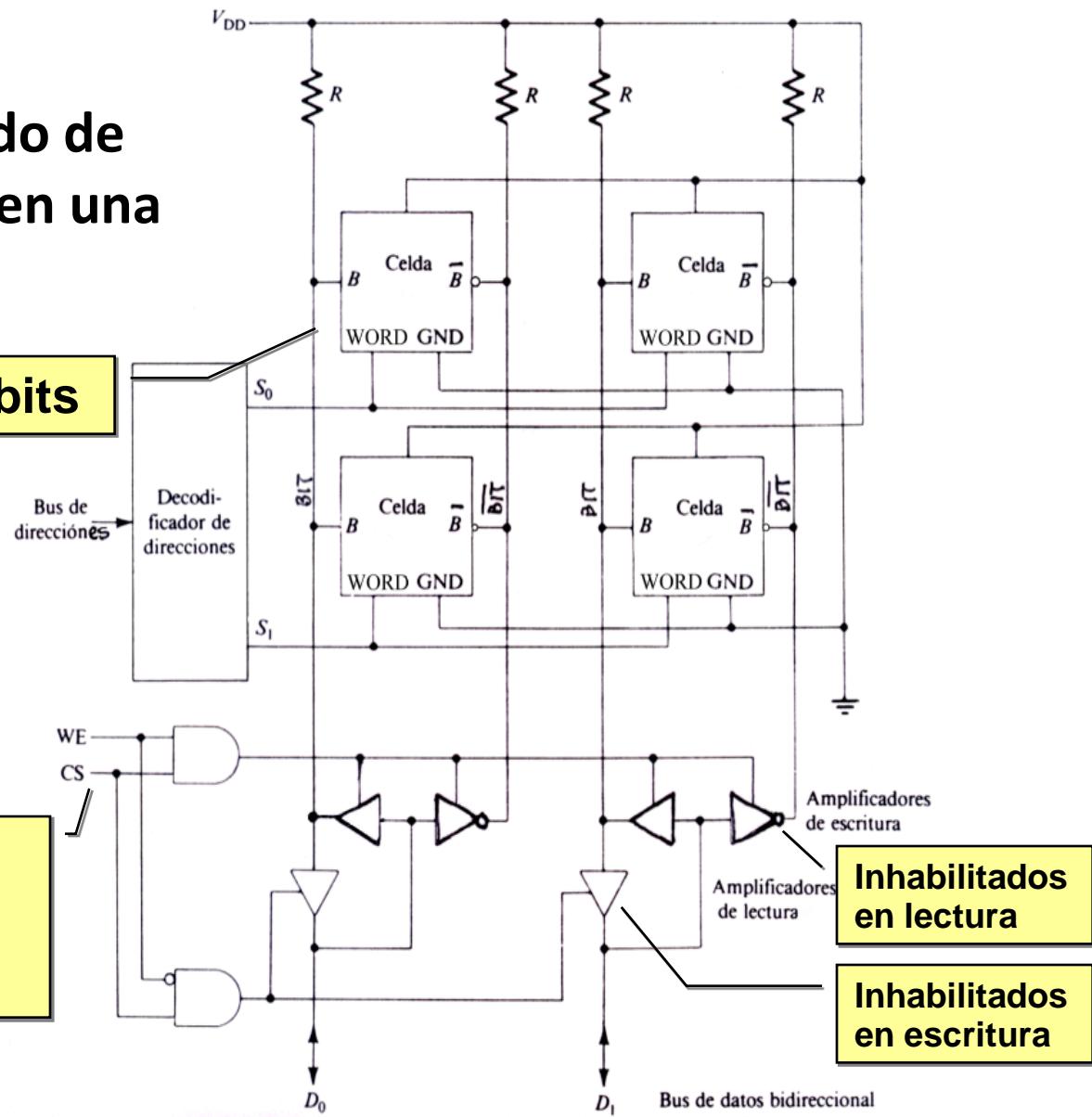


SRAM

- Esquema simplificado de conexión de celdas en una SRAM:

SRAM de 2 palabras de 2 bits

Si CS = 1 \Rightarrow
WE = 1 \Rightarrow Escritura
WE = 0 \Rightarrow Lectura



Inhabilitados
en lectura

Inhabilitados
en escritura

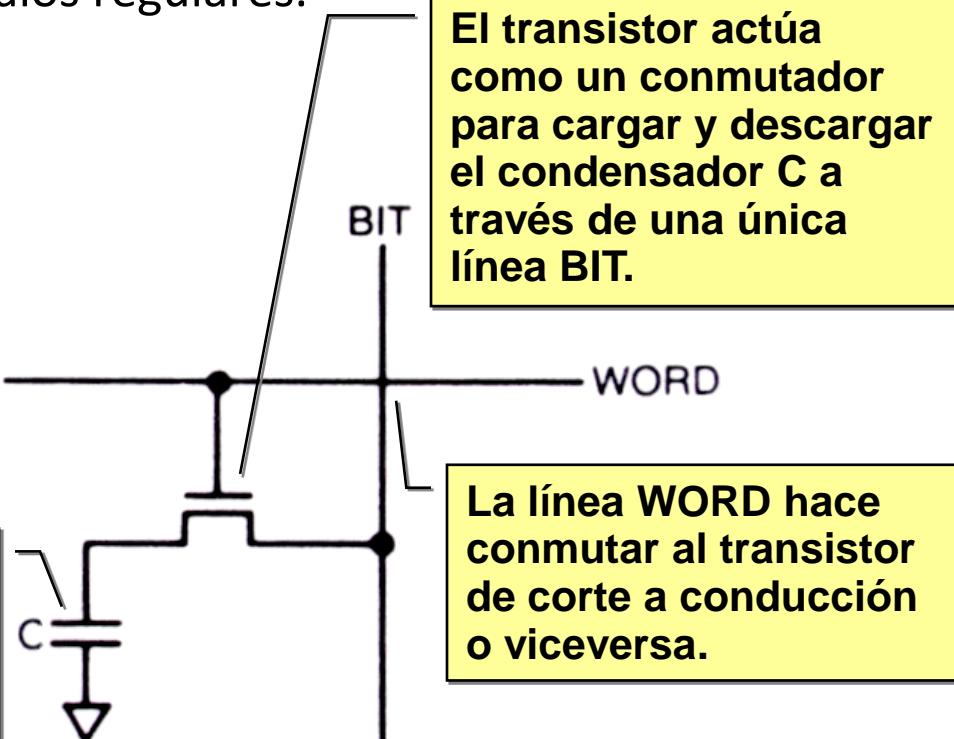
DRAM

■ Celda de memoria DRAM

- **Dinámica** ⇒ los datos almacenados decaen o se desvanecen y deben ser restaurados a intervalos regulares.

✓ Sencillez de la celda de almacenamiento.

La información se almacena en forma de carga eléctrica en un condensador C. La presencia (o ausencia) de carga indica que hay almacenado un 1 (o un 0).

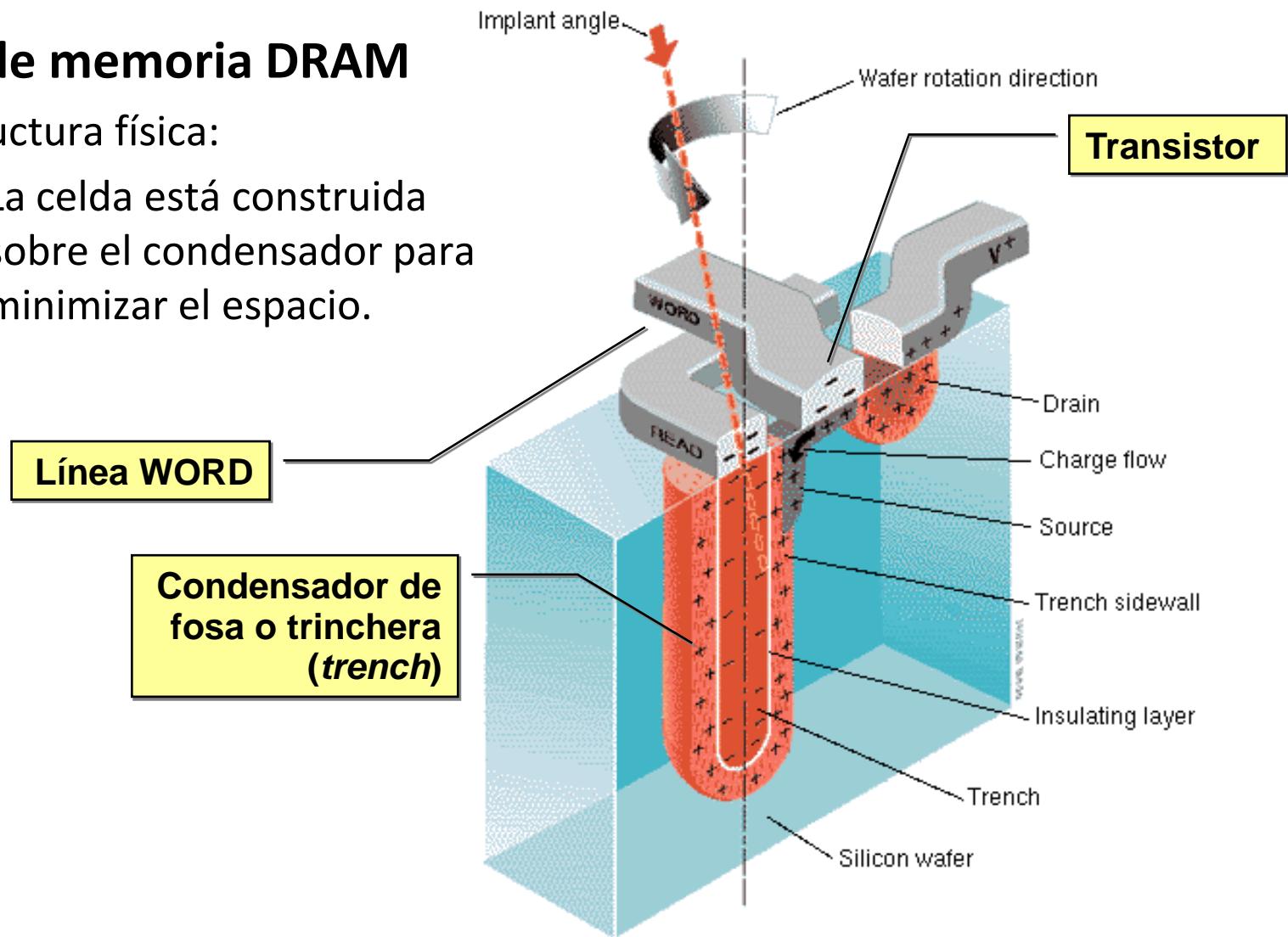


DRAM

■ Celda de memoria DRAM

■ Estructura física:

- La celda está construida sobre el condensador para minimizar el espacio.

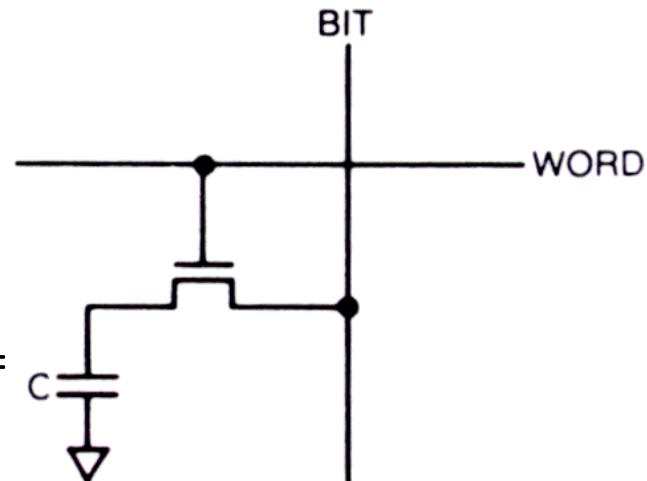


DRAM

■ Celda de memoria DRAM

▪ Operación de lectura:

- La circuitería externa convierte a BIT en una línea de salida, seleccionándose la celda con WORD = 1.
- Si C está cargado (= 1) \Rightarrow se descarga a través de la línea BIT \Rightarrow se produce un pulso de corriente que es detectado por un amplificador de salida ("*sense amplifier*") \Rightarrow aparece un 1 en la línea de datos de salida.
- Si C está descargado (= 0) \Rightarrow no se produce pulso de corriente \Rightarrow aparece un 0 en la línea de datos de salida.



✗ Lectura destructiva.

↓
Debe ir seguida de una escritura que restaure el estado original.

↓
Realizada automáticamente por los circuitos de control de la DRAM

DRAM

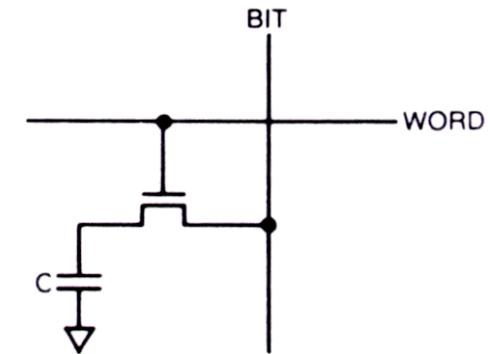
■ Celda de memoria DRAM

■ Operación de escritura:

- Se usa BIT como entrada y se pone a 0 ó a 1, seleccionándose la celda que tenga WORD a 1.
- Si BIT = 1 \Rightarrow C se carga.
- Si BIT = 0 \Rightarrow C se descarga.

■ Operación de refresco:

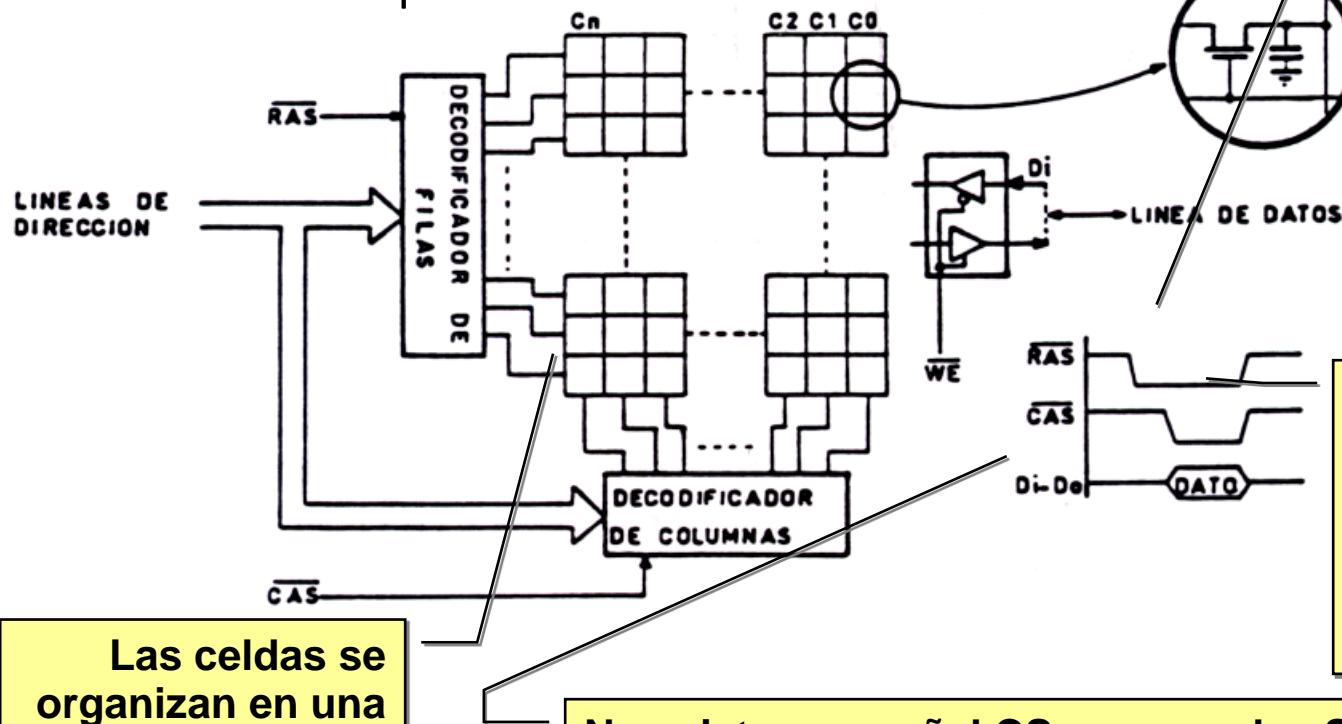
- El aislamiento del condensador C no es perfecto \Rightarrow su carga tiende a desvanecerse en pocos ms.
- El contenido de la DRAM ha de ser restaurado periódicamente. Puede haber un circuito de control externo que genere secuencialmente todas las direcciones, especificando una operación de lectura para cada dirección.



DRAM

Circuitos de memoria DRAM

- Capacidad elevada de los chips de memoria DRAM ⇒ las direcciones han de proporcionarse multiplexadas en el tiempo.



Las celdas se organizan en una matriz lo más cuadrada posible.

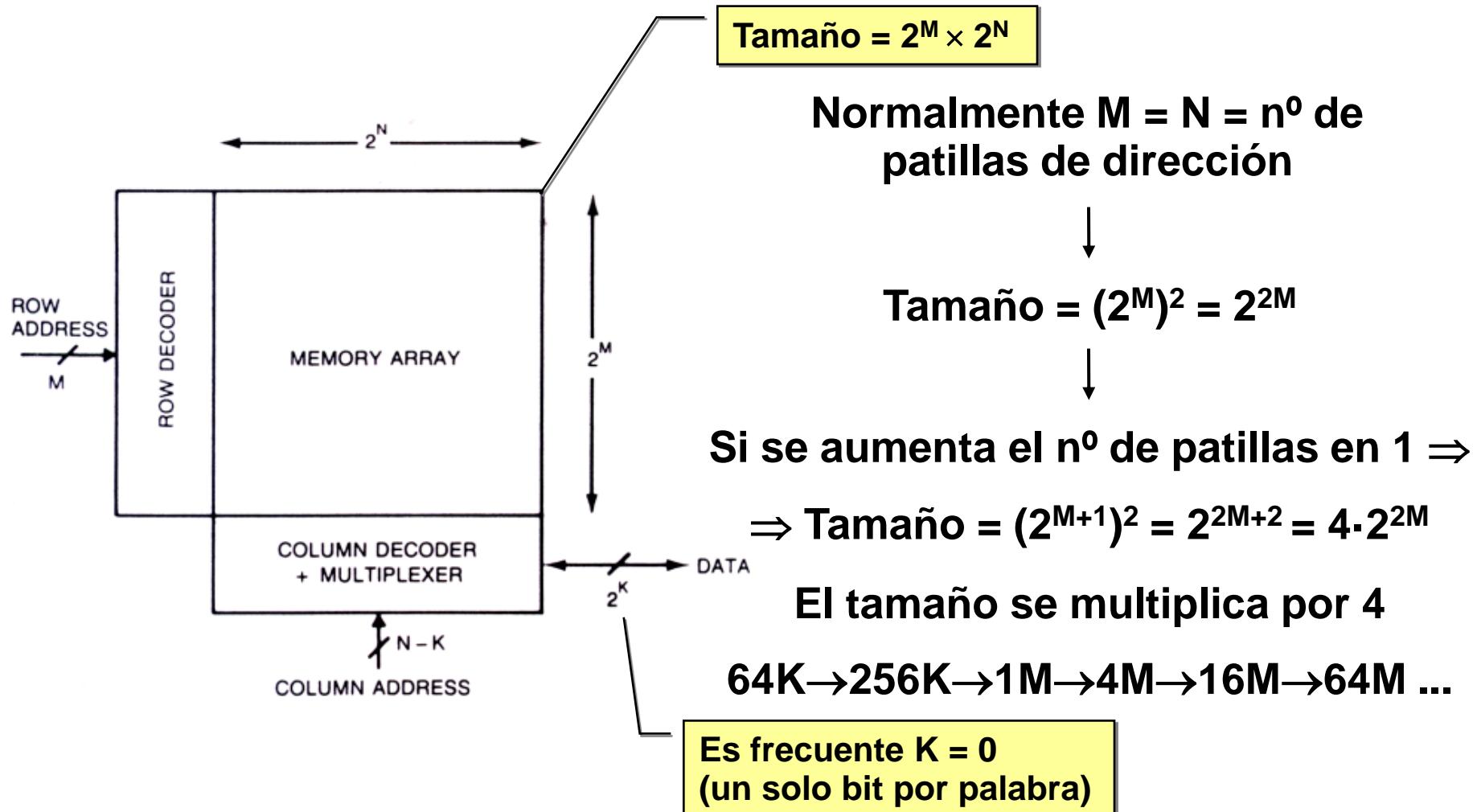
No existe una señal CS como en las SRAM. Para dar por válido un acceso a memoria es necesaria la secuencia completa /RAS - /CAS junto con la señal /WE (Write Enable).

Los bits correspondientes a las filas de la matriz de memoria se proporcionan en primer lugar, validándose por la señal /RAS (Row Access Strobe).

Después se proporcionan los bits que direccionan las columnas, validados por /CAS (Column Access Strobe).

DRAM

■ Circuitos de memoria DRAM



DRAM

- Tiempo de acceso.
- Tiempo de ciclo.

$t_{RAC} = t_a$
(tiempo de acceso):

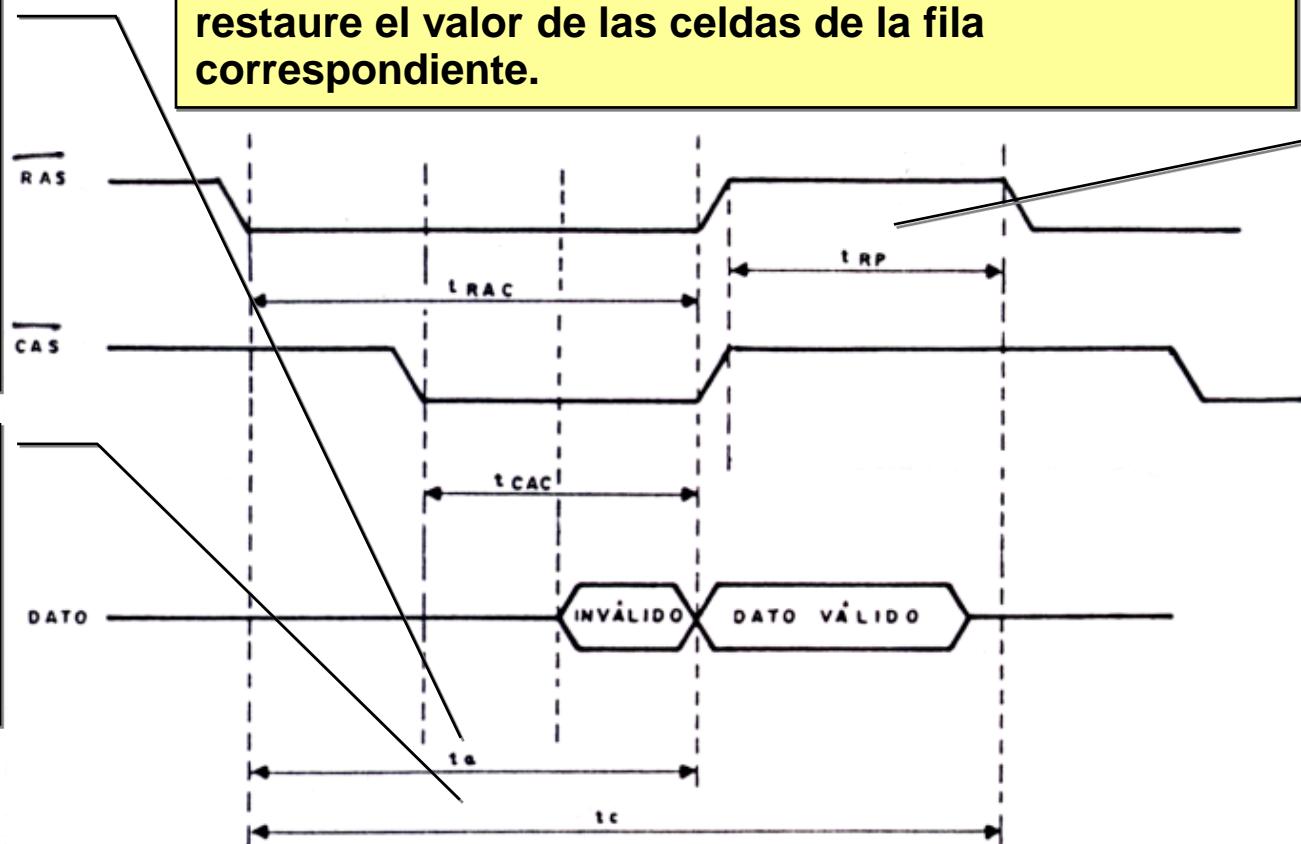
tiempo desde que se inicia una lectura ($/RAS \downarrow$) hasta que el dato está disponible en el bus de datos.

t_c (tiempo de ciclo):
 tiempo mínimo entre dos accesos consecutivos.

$$t_c \approx t_{RAC} + t_{RP}$$

$$t_c > t_a$$

No se puede comenzar un segundo acceso tras t_a , dado que las lecturas son destructivas (el condensador de la celda de memoria se descarga) y es necesario esperar **t_{RP} (tiempo de precarga de fila)** para que la circuitería interna de la DRAM restaure el valor de las celdas de la fila correspondiente.



DRAM

- Ejemplos de tiempos de acceso y de ciclo:

Año de introducción	Tamaño del chip	Acceso a filas		Acceso a columna (CAS)	Tiempo de ciclo
		DRAM más lenta	DRAM más rápida		
1980	64 Kbit	180 ns	150 ns	75 ns	250 ns
1983	256 Kbit	150 ns	120 ns	50 ns	220 ns
1986	1 Mbit	120 ns	100 ns	25 ns	190 ns
1989	4 Mbit	100 ns	80 ns	20 ns	165 ns
1992?	16 Mbit	≈ 85 ns	≈ 65 ns	≈ 15 ns	≈ 140 ns

Tiempos de las DRAM rápidas y lentas de cada generación.

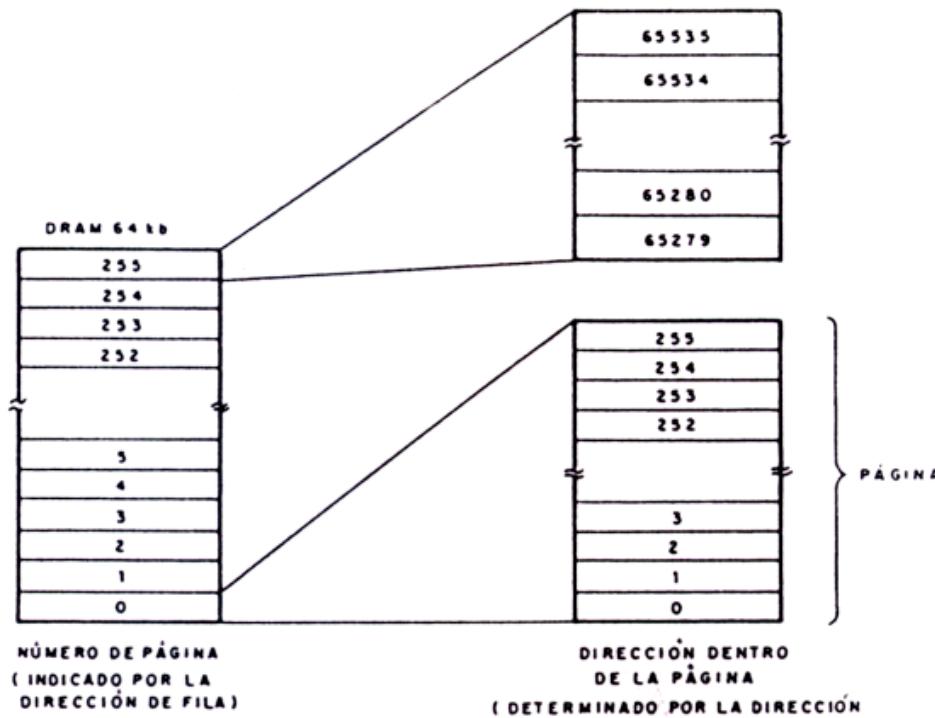
La mejora por un factor de dos en los accesos a columnas se produjo junto con el cambio de DRAM NMOS a DRAM CMOS. Con tres años por generación, la mejora de rendimiento del tiempo de acceso a filas es aproximadamente el 7 por 100 por año. Los datos de la última fila representan el rendimiento predicho para las DRAM de 16 Mbits, que no están todavía disponibles.

Hoy ~~ni~~ están disponibles.

DRAM

■ DRAM con modo página rápida (*Fast Page Mode RAM, FPM RAM*):

- Se puede acceder a cualquier dirección (columna) dentro de una página (fila), una vez seleccionada esta última, manteniendo /RAS a 0 y cambiando /CAS de 1 a 0 en cada acceso.
- Estrictamente hablando, no es una memoria RAM ya que los accesos a varias columnas de la última fila son más rápidos.



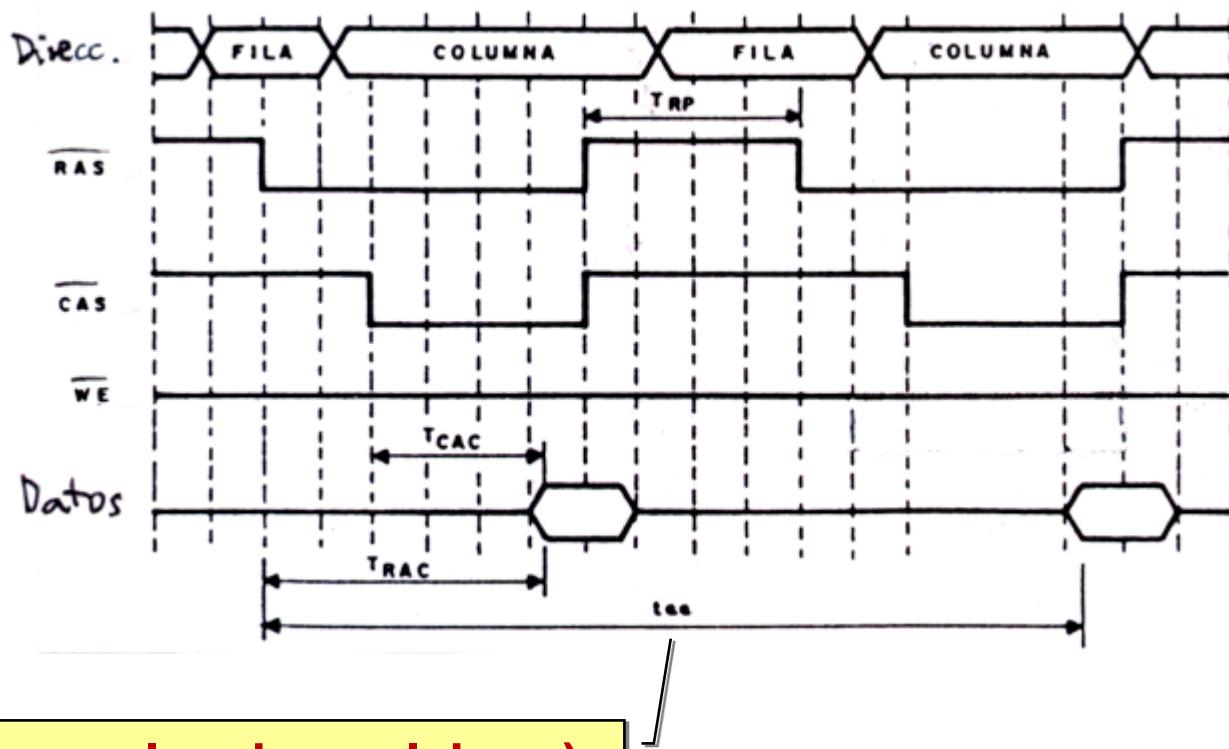
✓ **Velocidad ↑ (en comparación con acceso normal) .**

Hoy día (2011):

- ✗ **es la más lenta de todas las DRAM existentes.**
- ✗ **es cara debido a su baja demanda.**

DRAM

- Acceso a dos palabras:



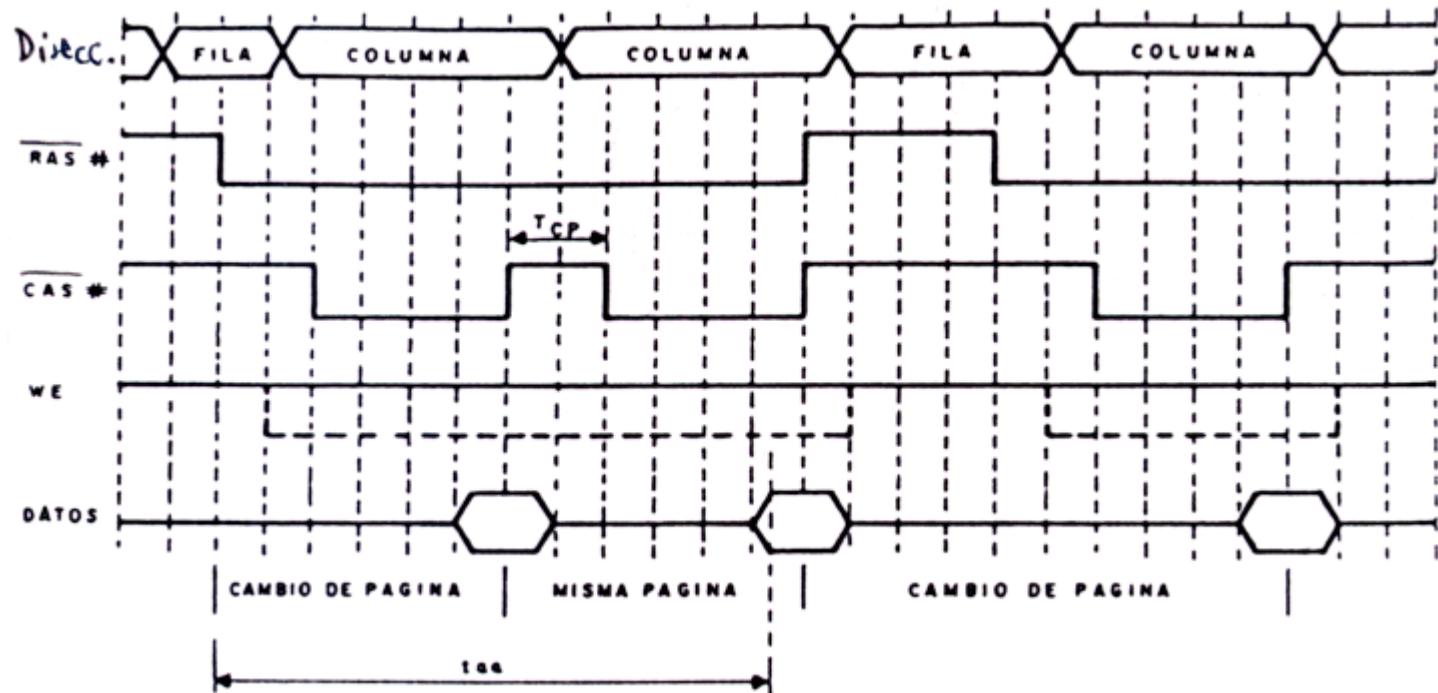
t_{aa} (tiempo de acceso a las dos palabras):

$$t_{aa} \approx t_{RAC} + t_{RP} + t_{RAC}$$

Ej.: $t_{aa} \approx 80 \text{ ns} + 60 \text{ ns} + 80 \text{ ns} = 220 \text{ ns}$

DRAM

- Acceso en modo página rápido (*Fast Page Mode*):



t_{aa} (tiempo de acceso a dos palabras en modo página):

$$t_{aa} \approx t_{RAC} + t_{CP} + t_{CAC}$$

Ej.: $t_{aa} \approx 80 \text{ ns} + 10 \text{ ns} + 20 \text{ ns} = 110 \text{ ns}$

DRAM

■ Refresco de DRAM

- Los chips DRAM refrescan automáticamente la fila accedida en cualquier ciclo de lectura o escritura, pero...
- ...se precisa una circuitería auxiliar que produzca ciclos de refresco, consistentes en recargar los condensadores de todas las celdas.
- Los ciclos de refresco deben producirse cada pocos ms
 - de 4 a 64 ms aproximadamente.
- El ciclo de refresco consiste en dar un impulso /RAS junto con la dirección de fila para todas las filas. Con ello se refrescan todas las columnas de cada fila.

DRAM

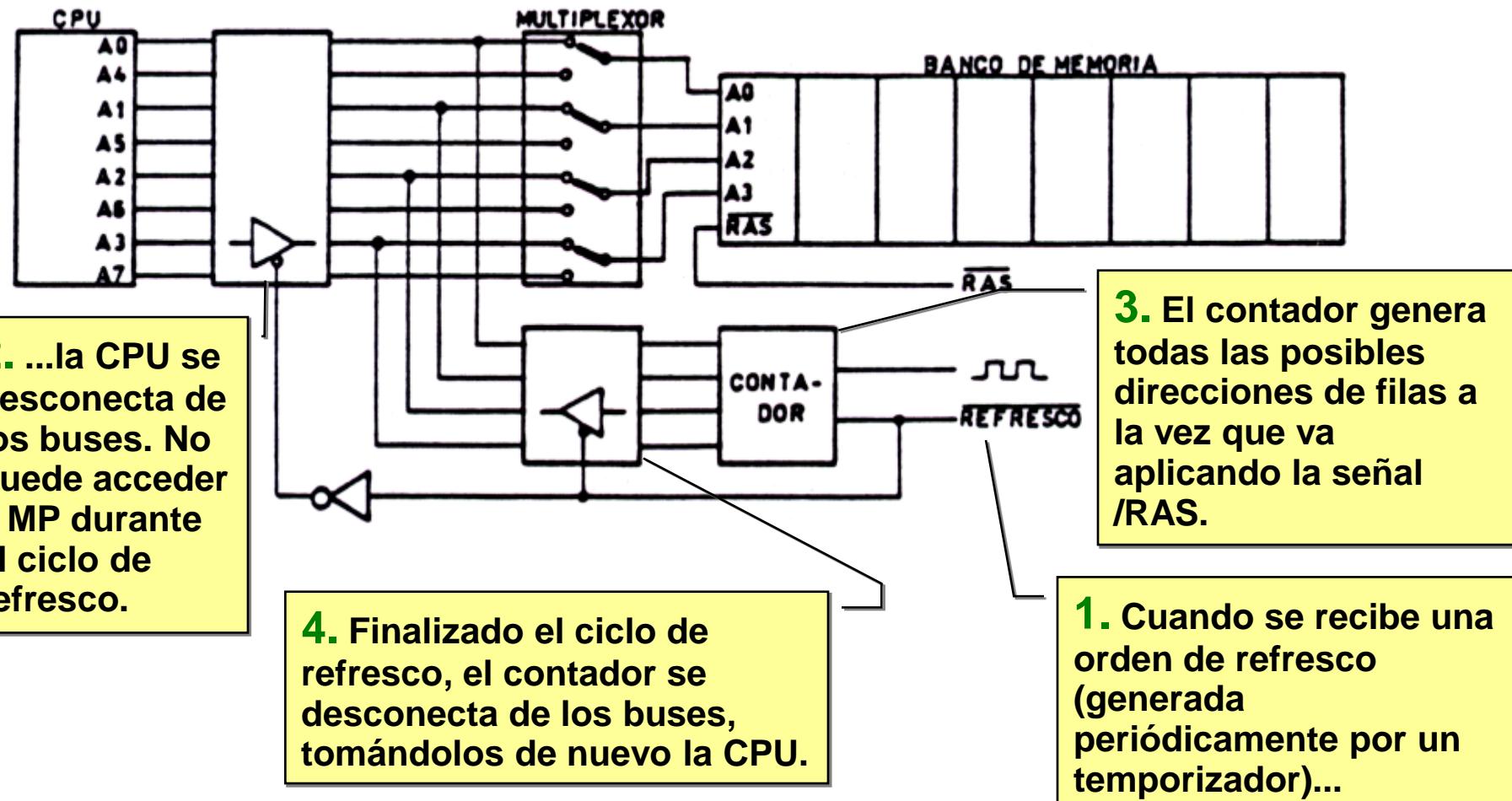
■ Refresco de DRAM

- La circuitería de refresco está basada fundamentalmente en un contador:
 - Controlador de DMA, o
 - Circuito controlador de memoria DRAM:
 - Genera las señales /RAS y /CAS.
 - Sirve de interfaz entre las señales que genera la CPU y el bus del sistema.
 - Controla el refresco.

DRAM

■ Refresco de DRAM

- Esquema genérico del circuito de refresco:



DRAM

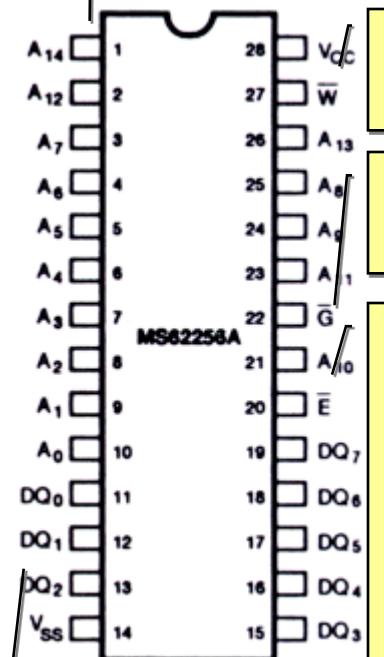
■ Refresco de DRAM

- Otra forma de hacer el refresco consiste en refrescar una única fila cada vez.
 - ✓ La CPU permanece menos tiempo seguido sin acceso al bus.
 - ✗ El refresco ha de pedirse con mayor periodicidad.
- Los chips DRAM actuales contienen:
 - Contador de refresco.
 - Modos de refresco que apenas necesitan circuitería externa.

Ejemplo de SRAM

256 K bits (32 K palabras × 8 bits)

A₀-A₁₄: 15 señales de dirección para seleccionar una de 32768 palabras de 8 bits.



/W (Write Enable) ≡ /WE. Se utiliza para indicar lectura o escritura.

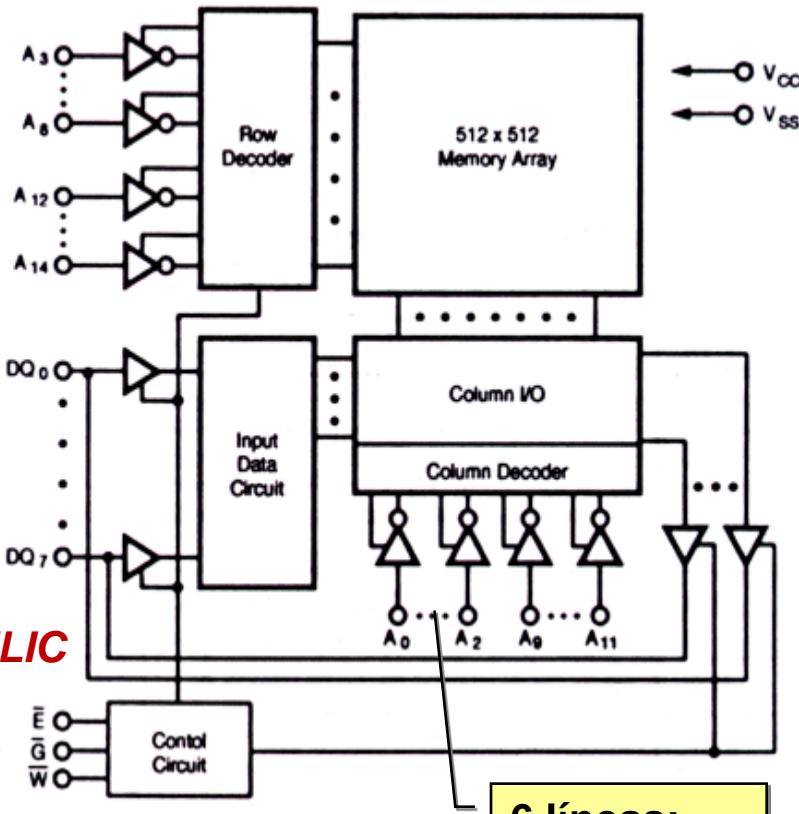
/G (Output Enable) ≡ /OE. Se selecciona (=0) para leer del chip.

/E (Chip Enable) ≡ /CS. Si no está activa ⇒ el chip no está seleccionado y permanece en “standby”, pasando su consumo de 900 mW a 50 mW.

**MOSEL-VITELIC
MS62256A
SRAM CMOS
32K × 8**

Mode	\bar{E}	\bar{G}	\bar{W}	I/O Operation
Standby	H	X	X	High Z
Read	L	L	H	D _{OUT}
Output Disabled	L	H	H	High Z
Write	L	X	L	D _{IN}

DQ₀-DQ₇ (Data Input/Output)



6 líneas:
 $9 - 6 = 3 \Rightarrow$
 se lee o
 escribe una
 palabra de
 $2^3 = 8$ bits

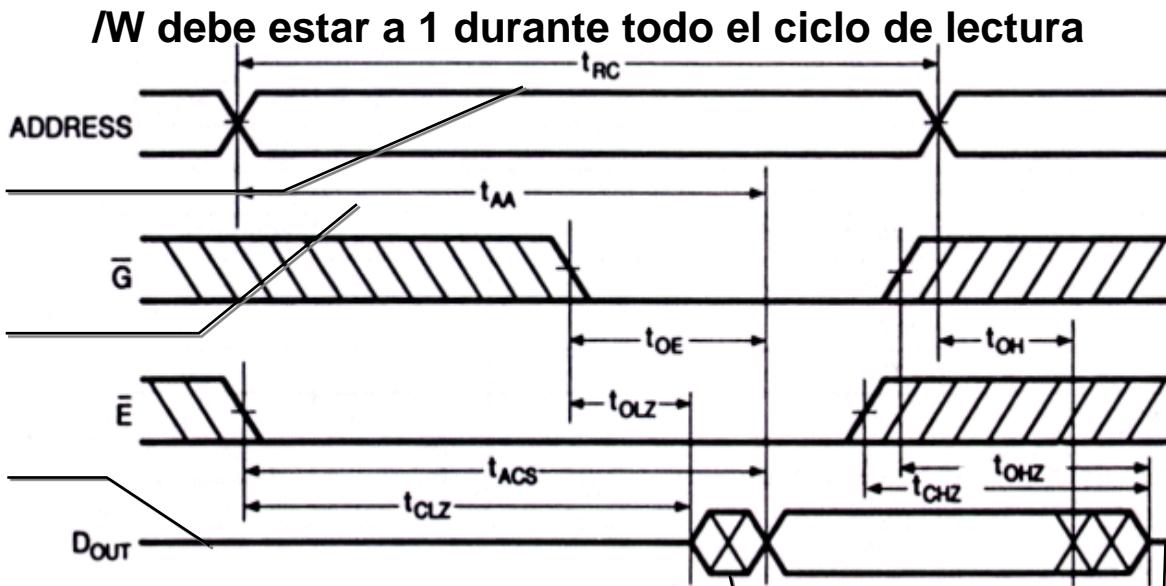
Ejemplo de SRAM

- Ciclo de lectura:**

t_{RC} : Tiempo de ciclo de lectura

t_{AA} : Tiempo de acceso

Alta impedancia



Parameter Name	Parameter	MS62256A-20		MS62256A-25		MS62256A-35		Unit
		Min.	Max.	Min.	Max.	Min.	Max.	
t_{RC}	Read Cycle Time	20	-	25	-	35	-	ns
t_{AA}	Address Access Time	-	20	-	25	-	35	ns
t_{ACS}	Chip Enable Access Time	-	20	-	25	-	35	ns
t_{OE}	Output Enable to Output Valid	-	8	-	12	-	15	ns
t_{CLZ}	Chip Enable to Output Low Z	3	-	5	-	5	-	ns
t_{OLZ}	Output Enable to Output in Low Z	0	-	0	-	0	-	ns
t_{CHZ}	Chip Disable to Output in High Z	-	8	-	10	-	15	ns
t_{OHZ}	Output Disable to Output in High Z	-	7	-	10	-	15	ns
t_{OH}	Output Hold from Address Change	3	-	5	-	5	-	ns

Cambiando.
Estado
desconocido.

Alta impedancia

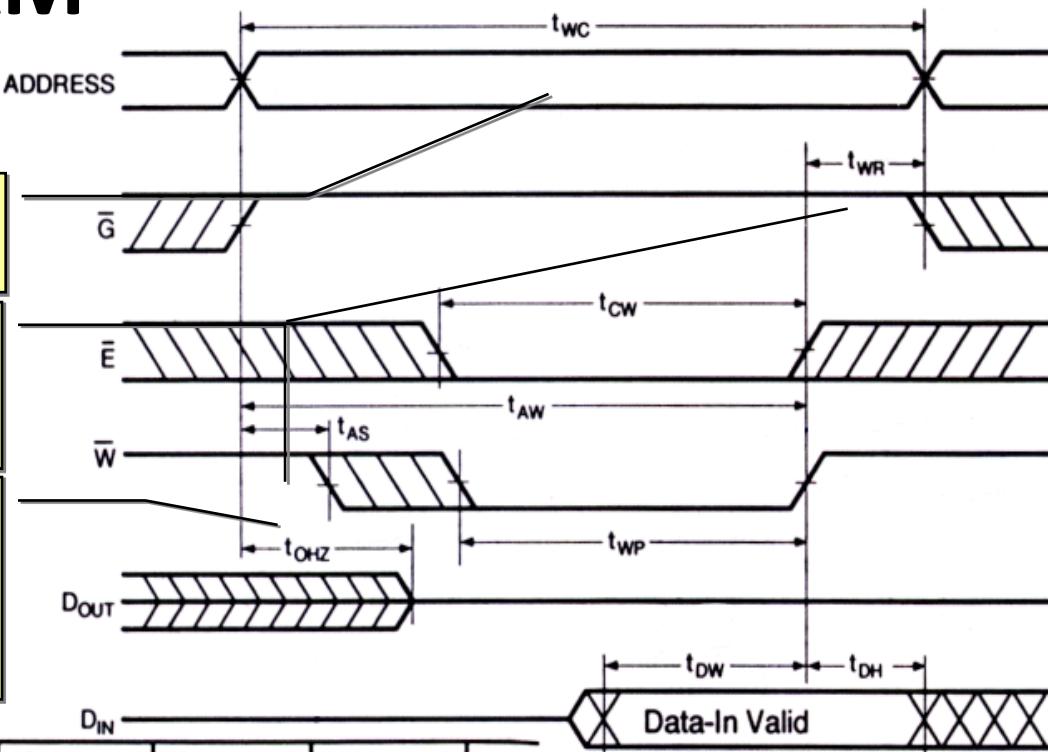
Ejemplo de SRAM

- Ciclo de escritura:**

t_{WC} : Tiempo de ciclo de escritura

Para garantizar que /W sea 1 durante la transición de dirección.

Durante t_{OHZ} las líneas D_{IN} no deben fijarse a 0 ni 1 (han de permanecer en alta impedancia).



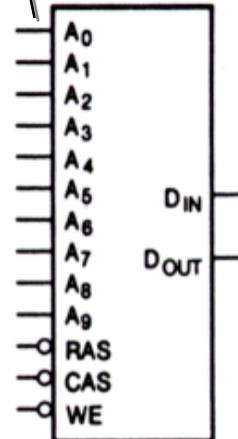
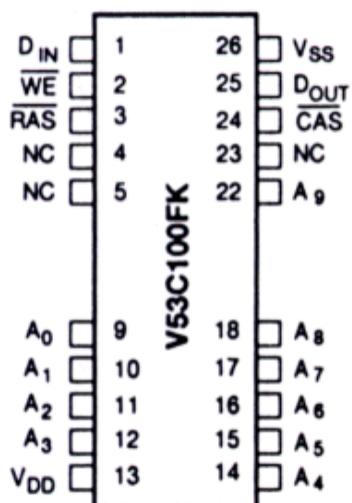
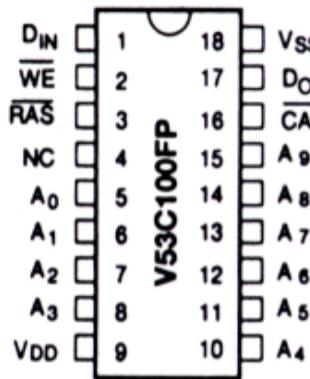
Parameter Name	Parameter	MS62256A-20		MS62256A-25		MS62256A-35		Unit
		Min.	Max.	Min.	Max.	Min.	Max.	
t_{WC}	Write Cycle Time	20	-	25	-	35	-	ns
t_{CW}	Chip Enable to End of Write	15	-	20	-	25	-	ns
t_{AS}	Address Set up Time	0	-	0	-	0	-	ns
t_{AW}	Address Valid to End of Write	15	-	20	-	25	-	ns
t_{WP}	Write Pulse Width	15	-	15	-	20	-	ns
t_{WR}	Write Recovery Time	0	-	0	-	0	-	ns
t_{OHZ}	Write to Output in High Z	0	8	0	13	0	15	ns
t_{DW}	Data to Write Time Overlap	10	-	13	-	15	-	ns
t_{DH}	Data Hold from Write Time	0	-	0	-	0	-	ns

Ejemplo de DRAM

FPM 1 M bit (1 M palabras × 1 bit)

Una dirección (20 bits) se divide (multiplexada en el tiempo) en 10 bits para la fila y 10 para la columna.

Tiempo de acceso: 60 ns
 Tiempo de ciclo: 120 ns
 T. de ciclo en modo página: 40 ns



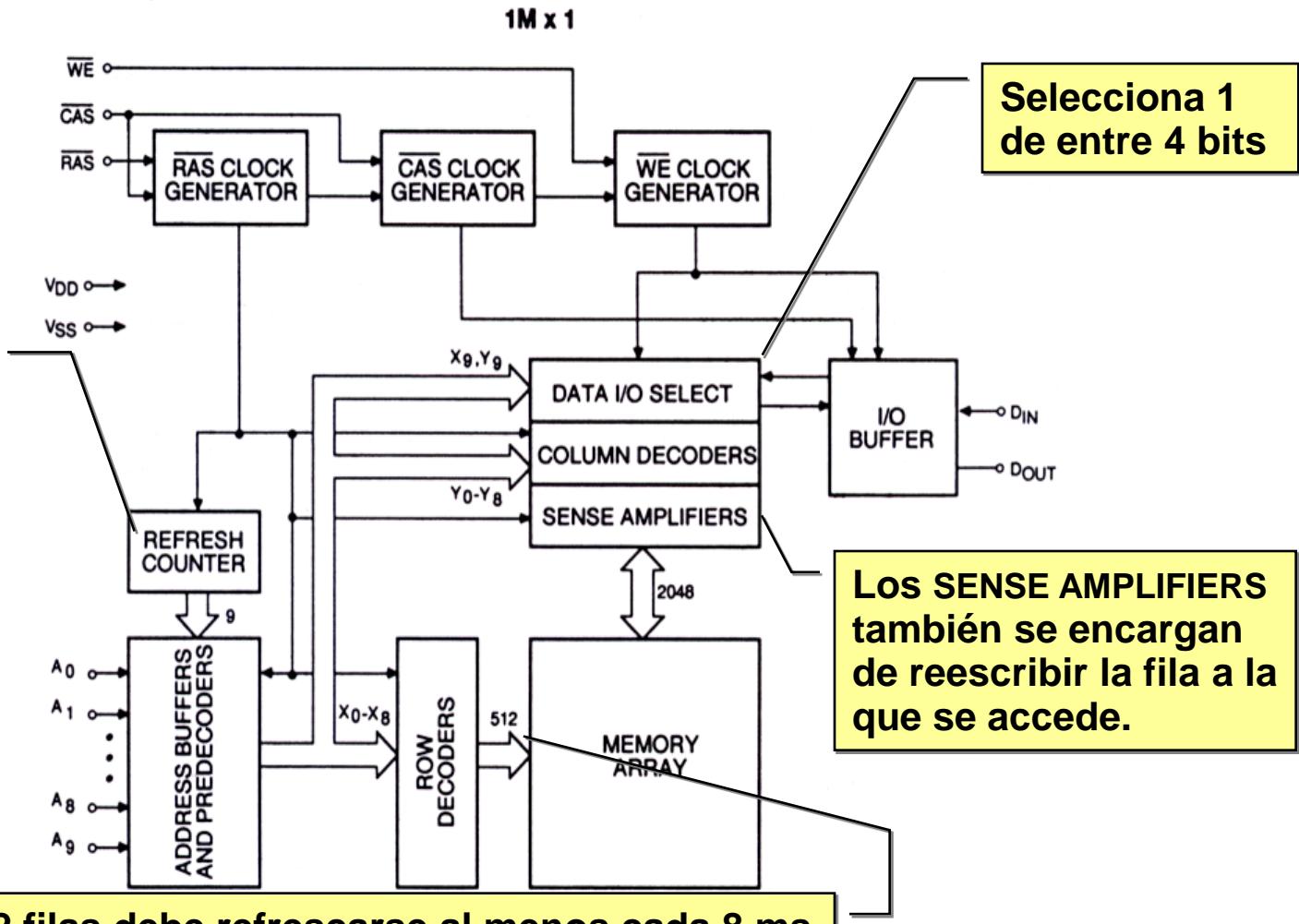
(Símbolo lógico)

$A_0 - A_9$	Address Inputs
\overline{RAS}	Row Address Strobe
\overline{CAS}	Column Address Strobe
\overline{WE}	Write Enable
D_{IN}	Data Input
D_{OUT}	Data Output
V_{DD}	+5V Supply
V_{SS}	0V Supply
NC	No Connect

MOSEL-VITELIC
FAMILIA V53C100F
DRAM CMOS
FAST PAGE MODE
1M × 1 BIT

Ejemplo de DRAM

- Diagrama de bloques funcionales:

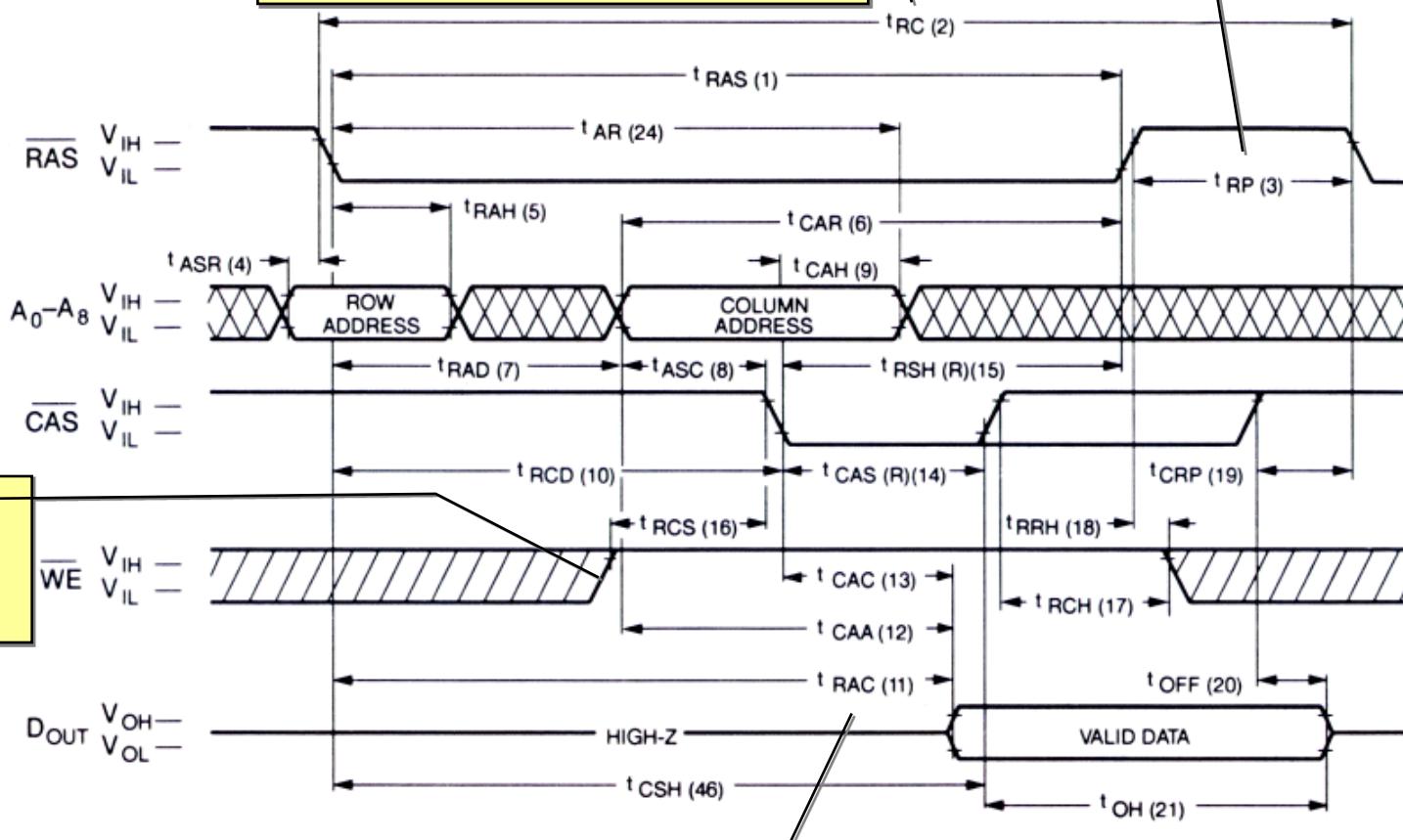


Ejemplo de DRAM

- Ciclo de lectura:**

t_{RP} : Tiempo de precarga de fila (empleado en refrescar la fila a accedida) (50 ns)

t_{RC} : tiempo de ciclo (120 ns)



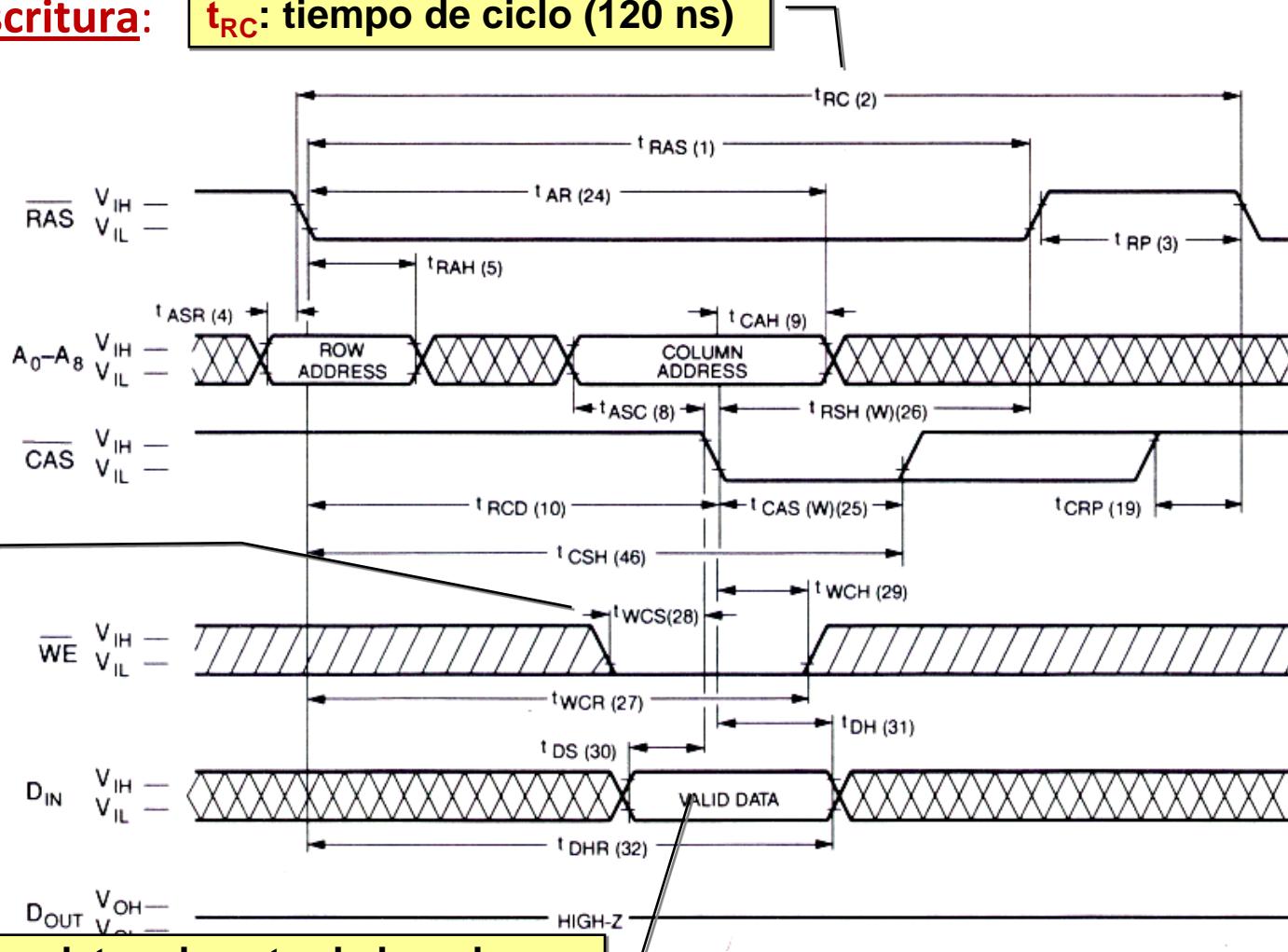
/WE ha de estar a 1 cuando /CAS↓

t_{RAC} : tiempo de acceso desde /RAS↓ (60 ns)

Ejemplo de DRAM

- Ciclo de escritura:**

t_{RC} : tiempo de ciclo (120 ns)



/WE ha de
estar a 0
cuando /CAS↓

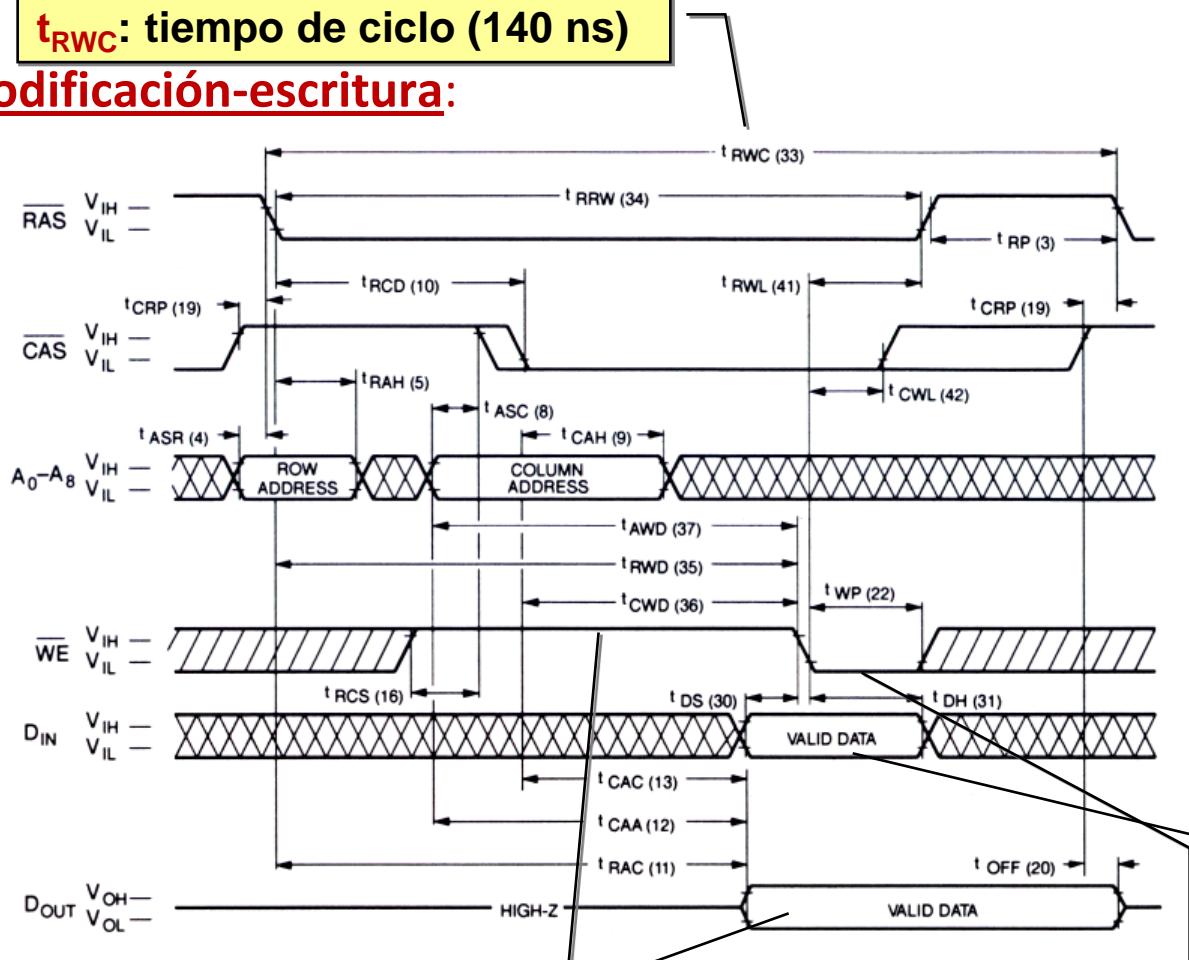
Los datos de entrada han de ser
válidos cuando /WE=0 y /CAS↓

Ejemplo de DRAM

t_{RWC} : tiempo de ciclo (140 ns)

- Ciclo de lectura-modificación-escritura:**

Permite el acceso para lectura a un bit y su posterior escritura, en la misma dirección, en un solo ciclo de duración ligeramente superior a la de un ciclo de lectura o escritura.



Primero se lee ($/WE=1$)...

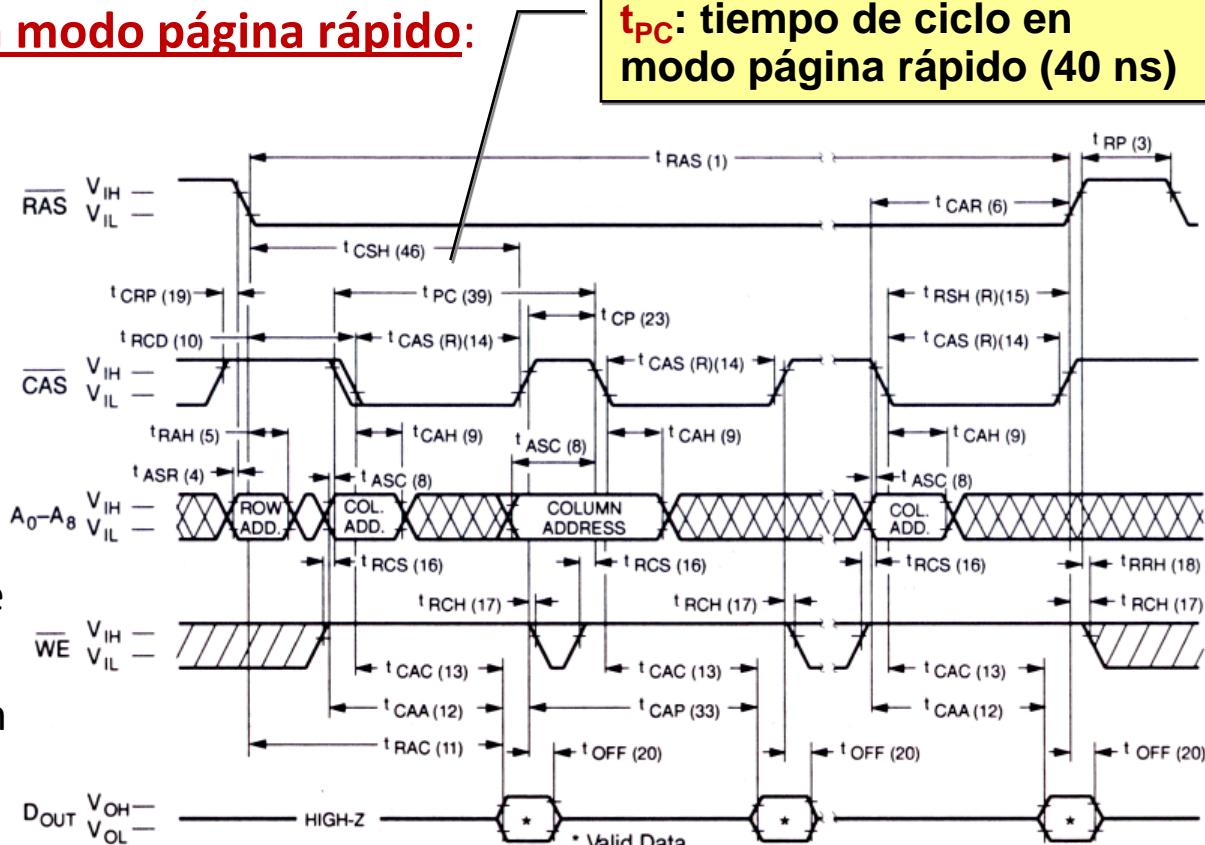
...y después se escribe ($/WE=0$).

Ejemplo de DRAM

- #### ▪ Ciclo de lectura en modo página rápida:

t_{PC}: tiempo de ciclo en modo página rápida (40 ns)

- Las 1024 columnas de una fila (página) pueden accederse a una frecuencia más alta, manteniendo /RAS a 0 mientras se suceden ciclos /CAS.
 - También existen ciclos de escritura y lectura-modificación-escritura en modo página rápido.



La máxima frecuencia de acceso a datos será:

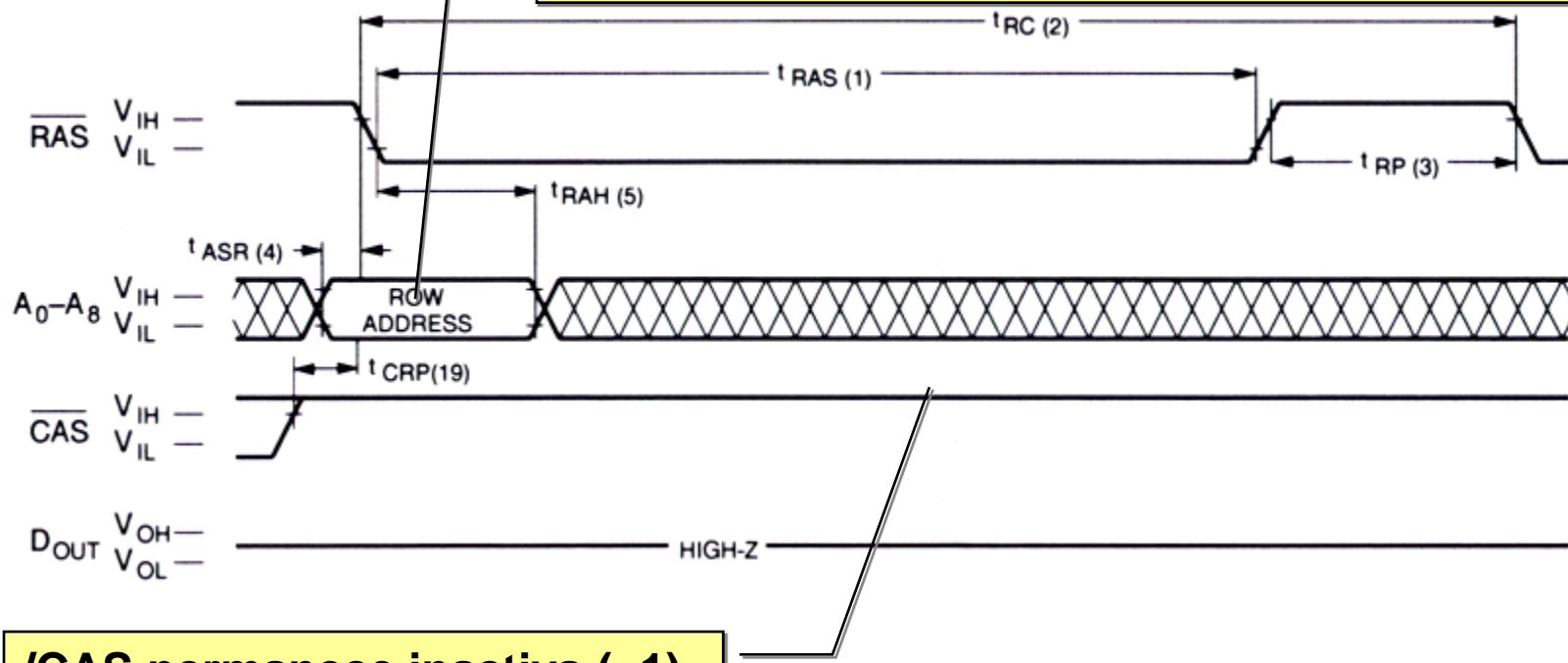
$$\frac{1024}{t_{RC} + 1023 \cdot t_{PC}}$$

Para $t_{RC} = 120$ ns y $t_{PC} = 40$ ns $\Rightarrow 25$ MHz

Ejemplo de DRAM

- Ciclo de refresco sólo-/RAS (/RAS-only):

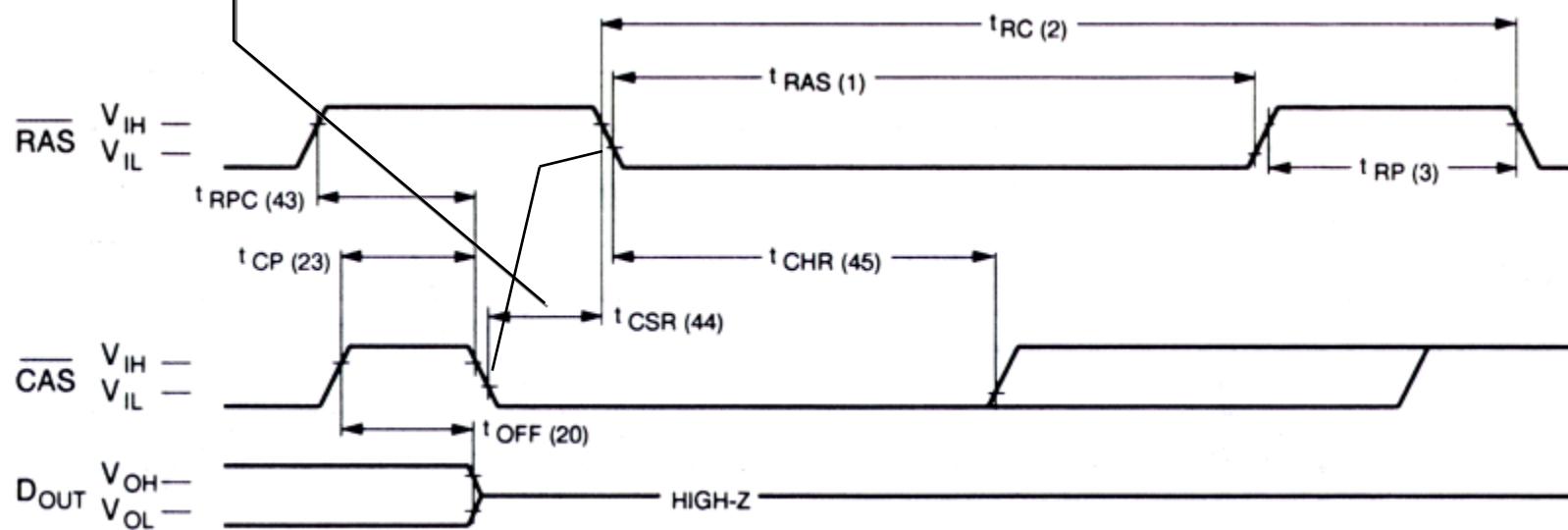
/RAS se activa ($=0$) \Rightarrow se valida la dirección de fila suministrada por un contador externo que va generando las direcciones de las filas a refrescar.



Ejemplo de DRAM

- Ciclo de refresco /CAS-antes de-/RAS (/CAS-before-/RAS):

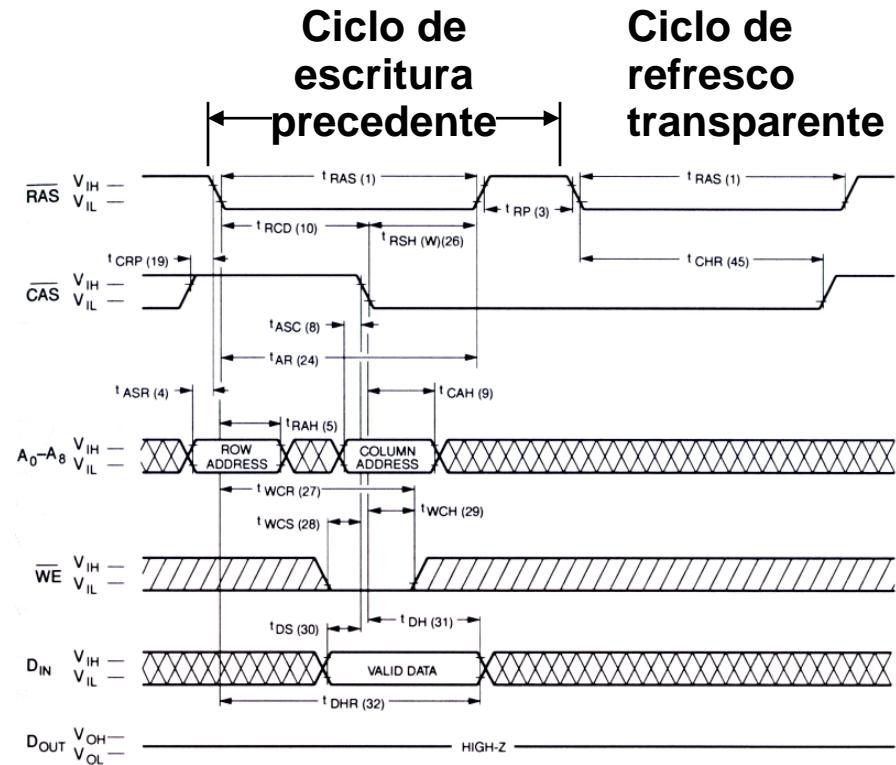
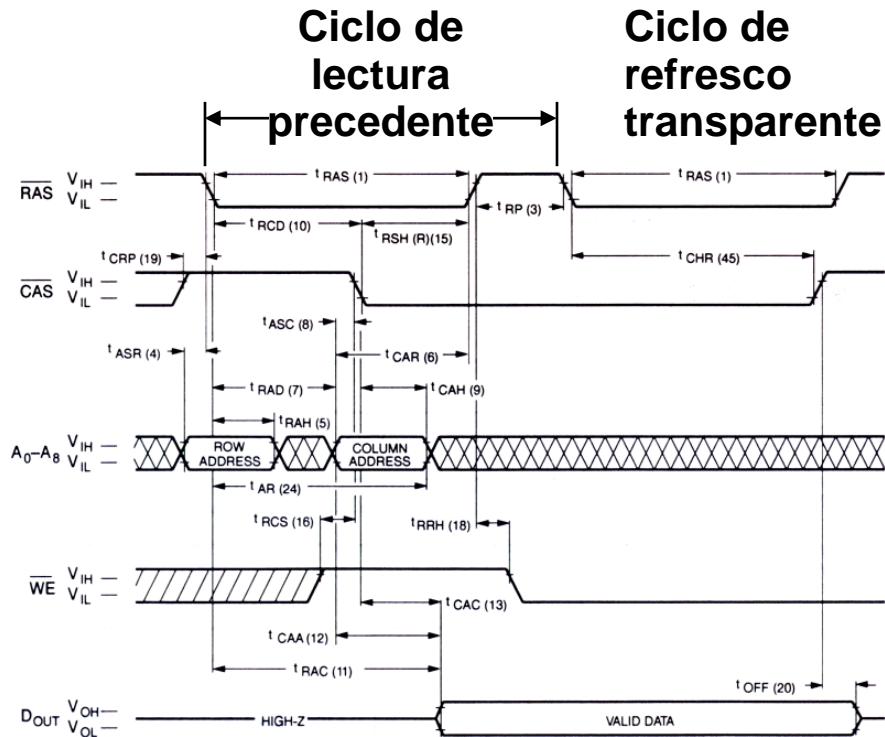
/CAS y /RAS se activan en orden inverso \Rightarrow se usa la salida del contador interno de 9 bits como fuente de la dirección de fila a refrescar, ignorándose las entradas de dirección externas.



Ejemplo de DRAM

- **Refresco transparente (Hidden refresh):**

- Similar a un ciclo /CAS-antes de-/RAS, pero en este caso /CAS no se pone a 1 y luego a 0, sino que permanece a 0 después del ciclo de lectura o escritura precedente.



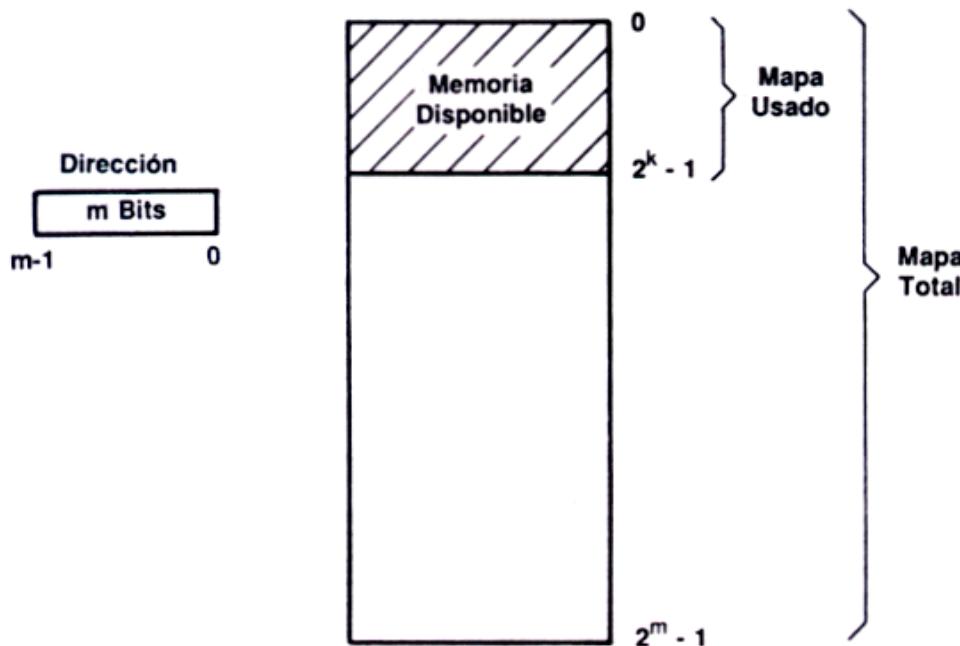
Memoria

- **Jerarquía de memoria. Concepto de localidad**
- **Memorias RAM semiconductoras. Memorias de sólo lectura. Prestaciones: velocidad, tamaño y coste**
- **Configuración y diseño de memorias utilizando varios chips**
- **Memorias asociativas**
- **Memoria cache. Influencia en las prestaciones**

Ampliación del mapa de memoria

■ Mapa de memoria

- Espacio direccionable por un procesador.
- Viene determinado por el tamaño de las direcciones.
- Generalmente un ordenador no dispone de toda la memoria necesaria para llenar el mapa de memoria, sino que queda espacio libre para posibles ampliaciones:

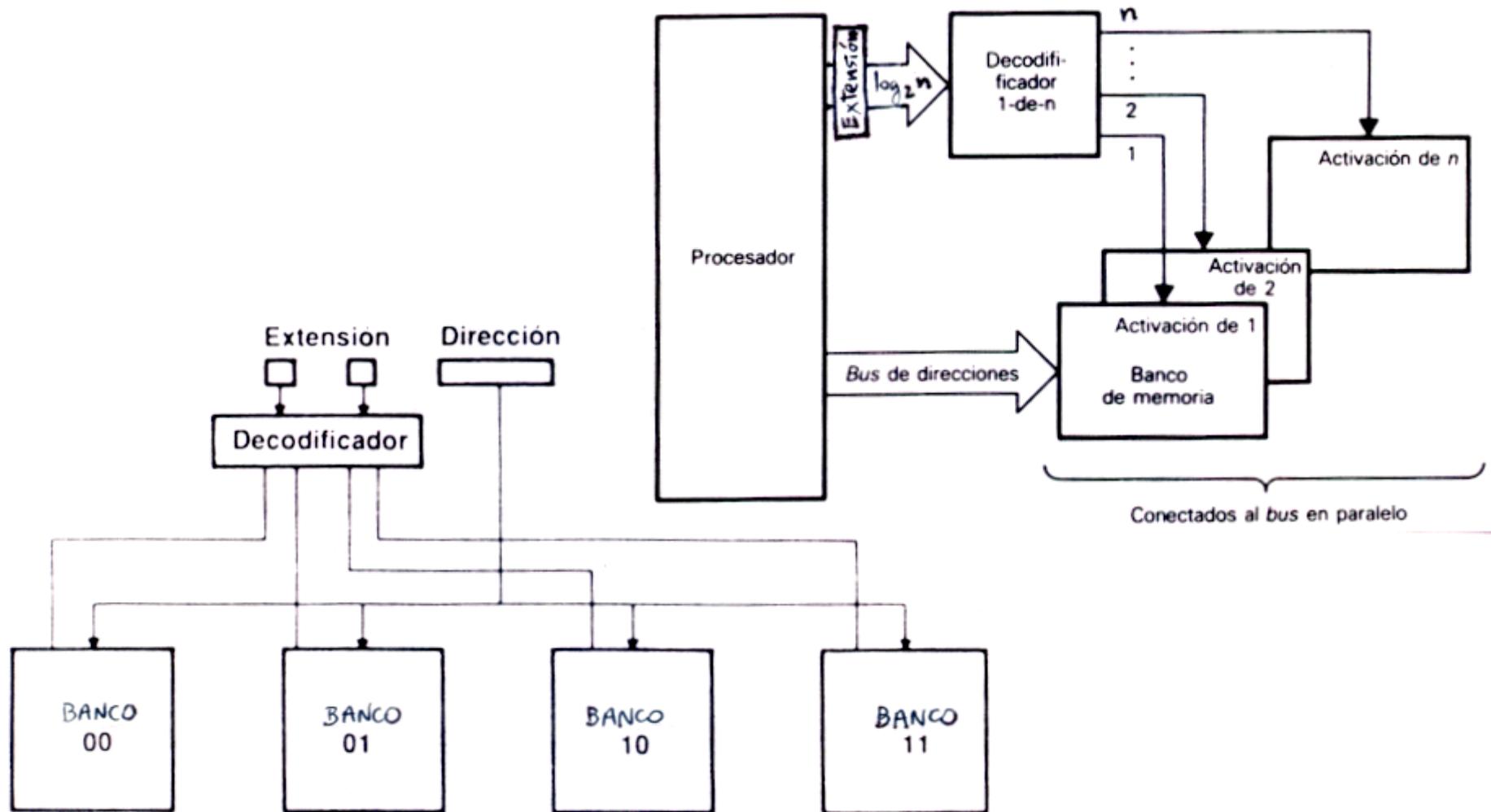


Ampliación del mapa de memoria

- Sin embargo, algunas arquitecturas con el tiempo se quedan pequeñas en cuanto a mapa de memoria.
- **Ampliación del mapa de memoria:**
 - ① Rediseñar el procesador para que sea capaz de generar direcciones con mayor número de bits.
 - ② Concatenar a las direcciones generadas por el procesador unos bits externos adicionales:
 - Se pueden modificar por programa (generalmente por medio de instrucciones de E/S).
 - Permiten seleccionar uno entre varios bancos de memoria distintos.
 - Este método es conocido como **CONMUTACIÓN DE BANCOS**.

Ampliación del mapa de memoria

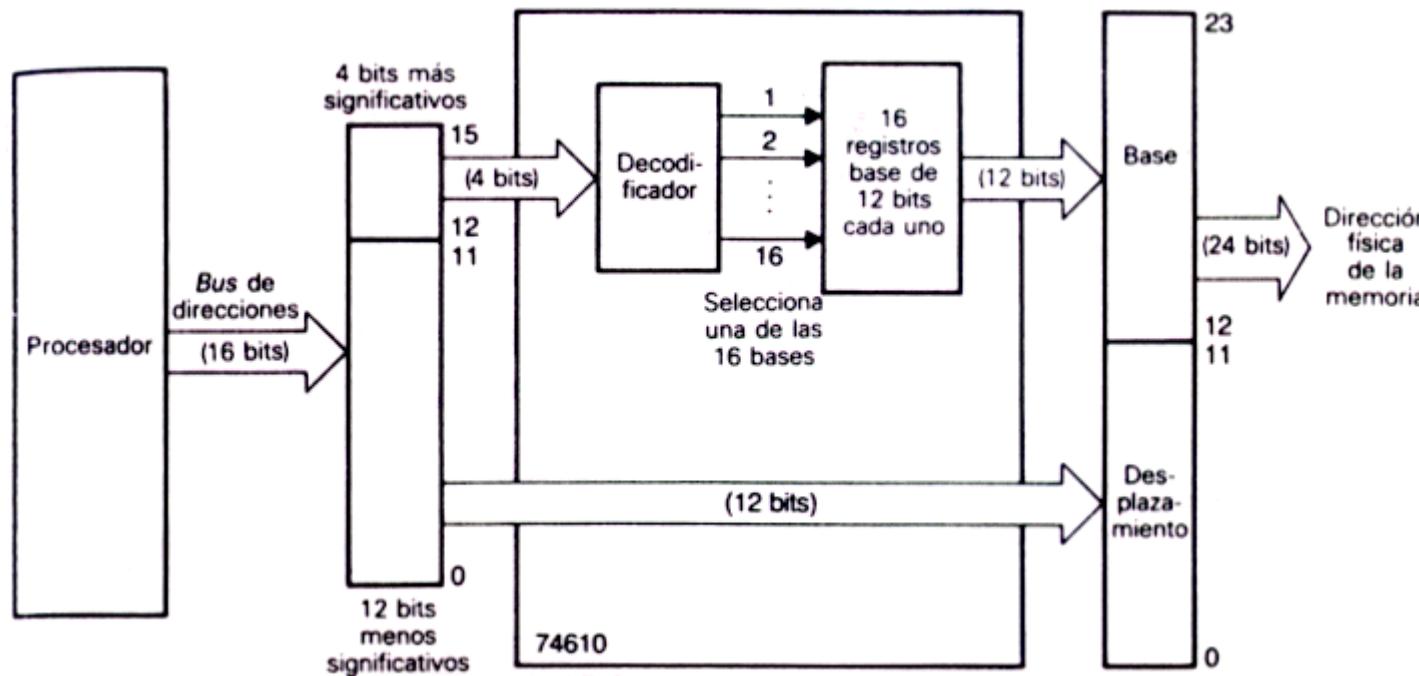
■ Conmutación de bancos:



Ampliación del mapa de memoria

- ③ Utilizar algunos de los bits más significativos del bus de direcciones para seleccionar un registro base de entre varios disponibles.
- Ejemplo:

- Chip 74610 de *Texas Instruments*.
- Con 16 bits de direcciones permite acceder a 16 MB de memoria física.



Ampliación de memoria

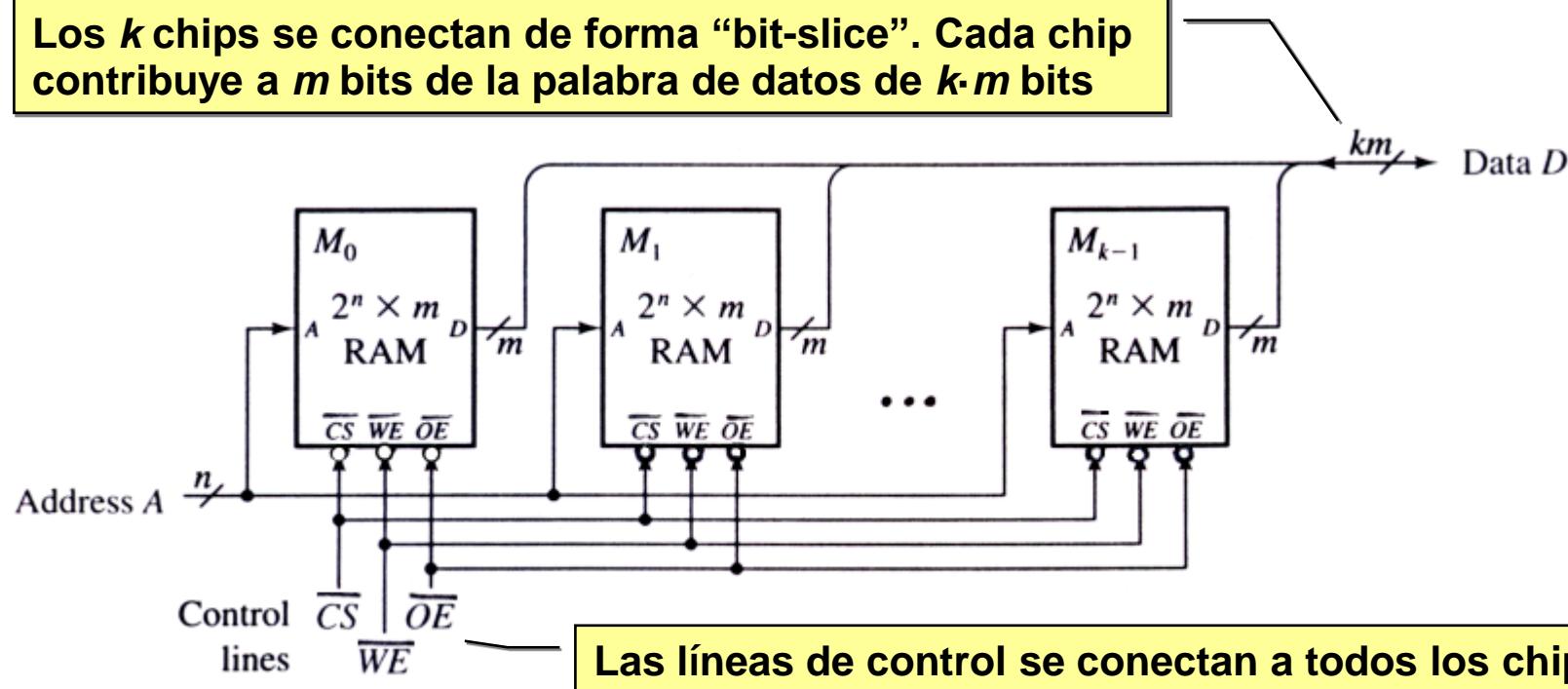
■ Problema:

- Construir una memoria de 2^N palabras de M bits a partir de chips de 2^n palabras de m bits.

① Incrementar el ancho de palabra de m a $k \cdot m = M$

- Necesitamos k circuitos de tamaño de palabra m :

Los k chips se conectan de forma “bit-slice”. Cada chip contribuye a m bits de la palabra de datos de $k \cdot m$ bits



Ampliación de memoria

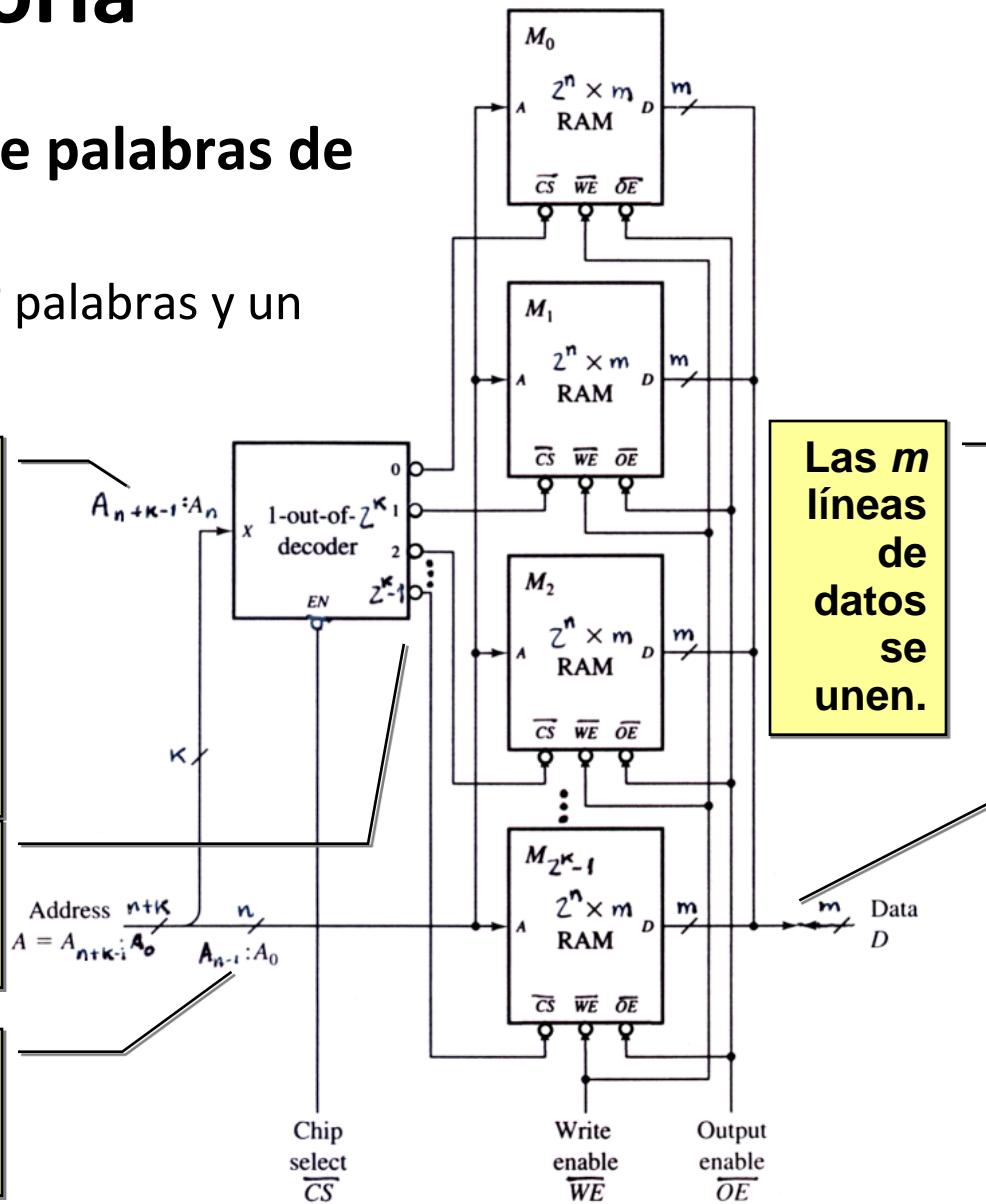
② Incrementar el número de palabras de 2^n a $2^{n+k}=2^N$

- Necesitamos 2^k circuitos de 2^n palabras y un decodificador de 1 entre 2^k .

Las k líneas de dirección de orden superior ($A_{n+k-1}:A_n$) se conectan a las entradas de un decodificador de k a 2^k \Rightarrow cada configuración de los $n+k$ bits hace que se seleccione solamente el circuito RAM indicado por los bits de dirección $A_{n+k-1}:A_n$.

Las 2^k líneas de salida del decodificador se conectan a las entradas de selección /CS de los 2^k circuitos.

Las n líneas de dirección de orden inferior se conectan en común a las líneas de dirección de los 2^k chips.



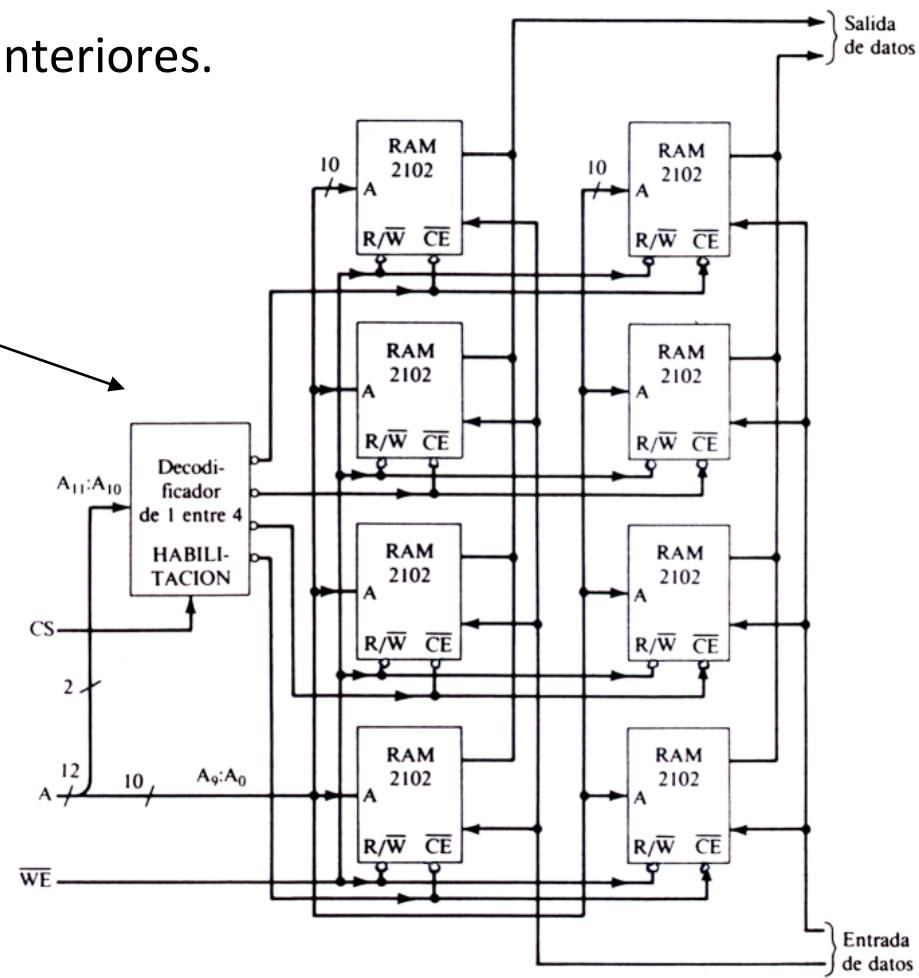
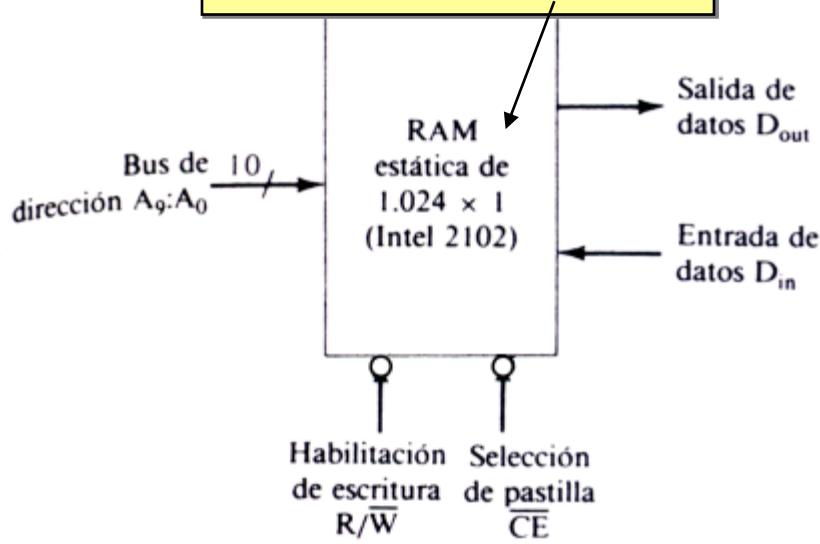
Ampliación de memoria

③ Incrementar simultáneamente el número de palabras y el ancho de palabra.

- Basta combinar las dos técnicas anteriores.

Ejemplo 1:

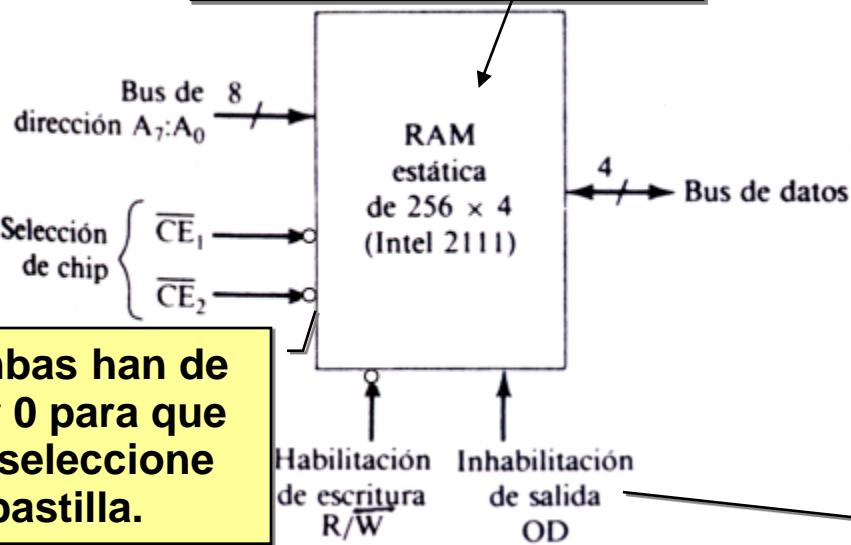
RAM de 4096×2 construida con 8 RAM de 1024×1 .



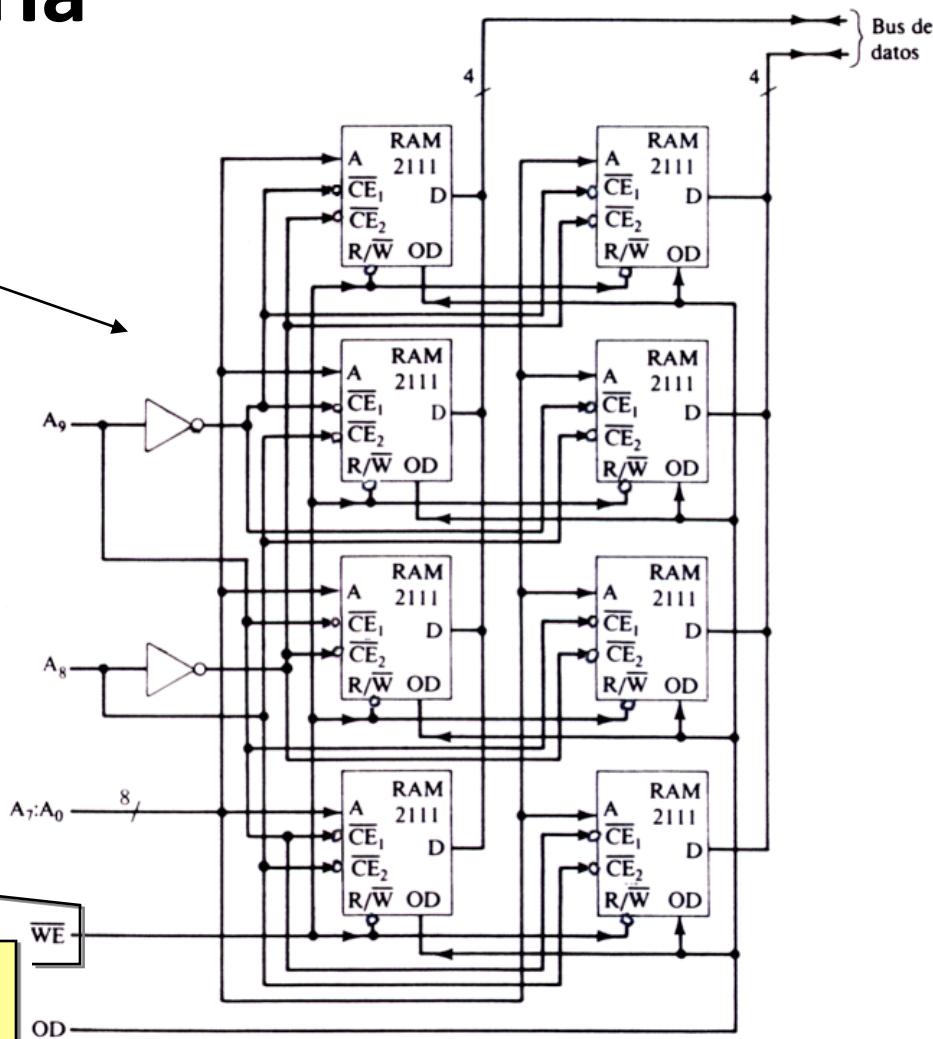
Ampliación de memoria

Ejemplo 2:

**RAM de $1\text{ K} \times 8$
construida con 8
RAM de 256×4 .**



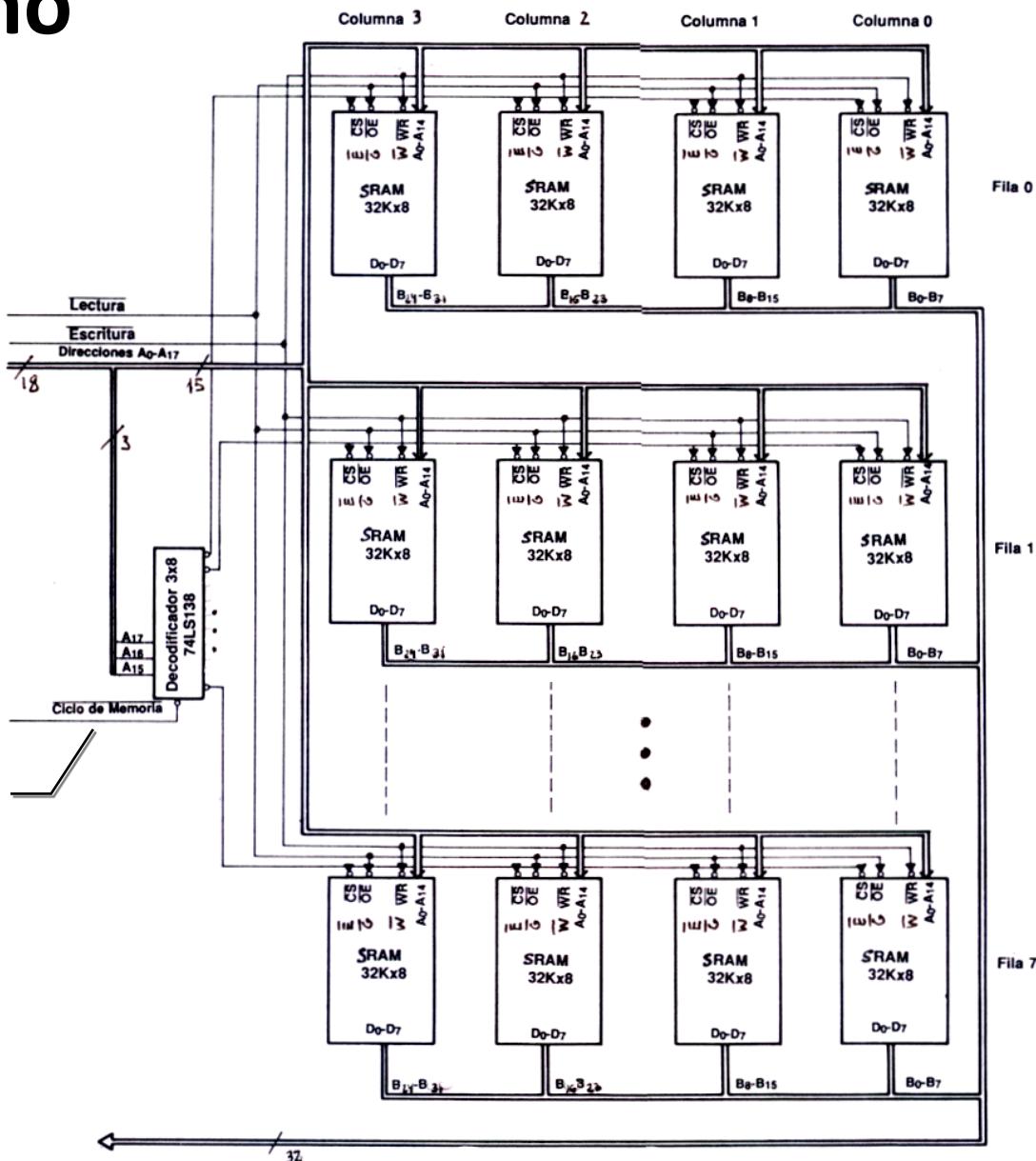
Como hay un solo bus de datos \Rightarrow la línea **OD (Output Disable)** se activa durante los ciclos de escritura para evitar que se lean datos internos desde el chip al bus mientras éste se está utilizando como bus de entrada.



Ejemplos de diseño

- Ejemplo 1:
 - SRAM de $256 \text{ K} \times 32$ empleando pastillas de $32 \text{ K} \times 8$.
 - Se requieren 8 filas de 4 pastillas = 32 pastillas.

Si /Ciclo_de_Memoria = 1 no se activa ninguna fila de pastillas.



Ejemplos de diseño

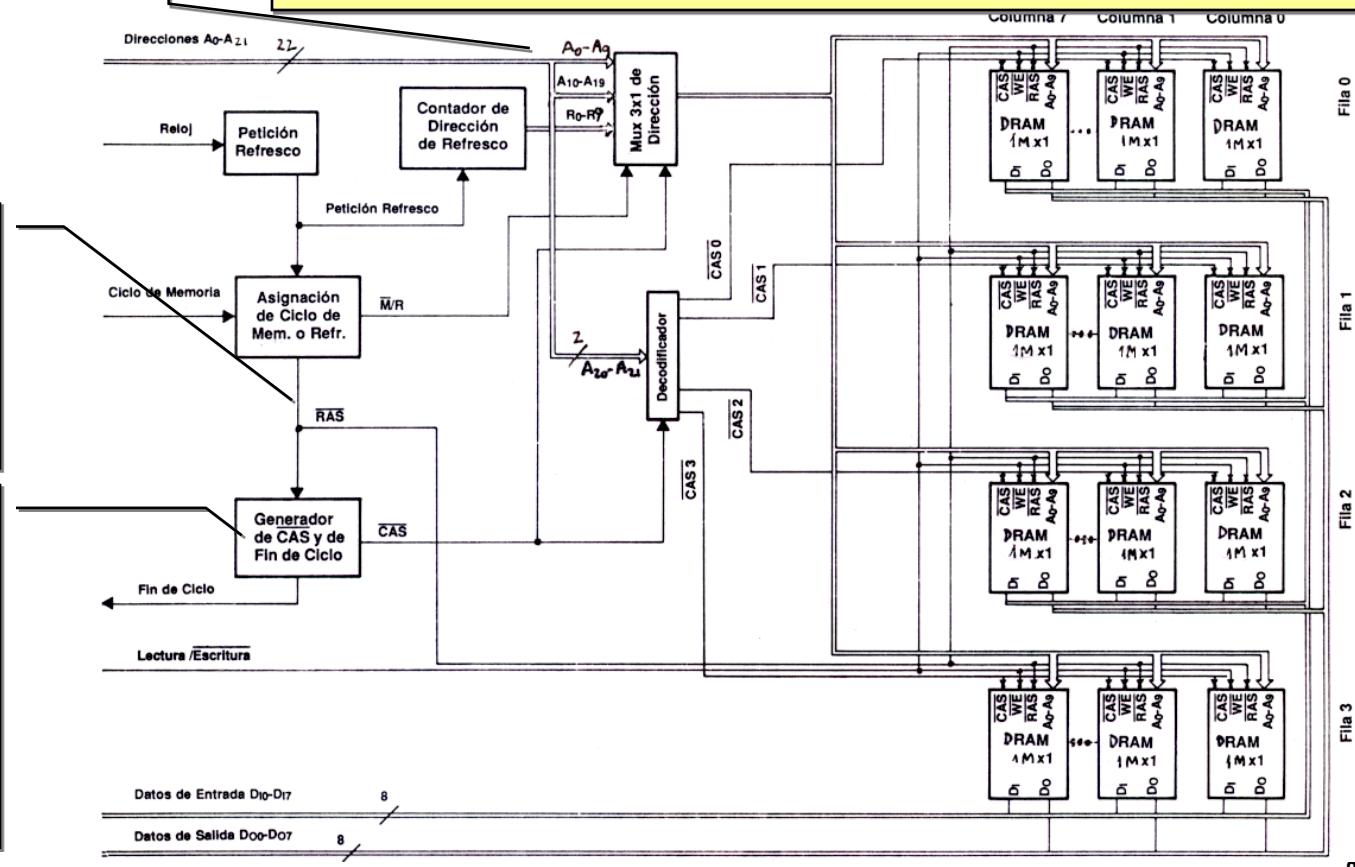
- Ejemplo 2:
 - DRAM de 4 Mbytes empleando pastillas de 1 M × 1.

/M / R	/CAS	Direcciones
0	0	A ₀ -A ₉
0	1	A ₁₀ -A ₁₉
1	X	R ₀ -R ₉

La señal /RAS especifica la primera mitad de la dirección y es común a todas las pastillas.

La señal /CAS, además de especificar la otra mitad, sirve como /CS y se activa de forma independiente para cada fila de pastillas.

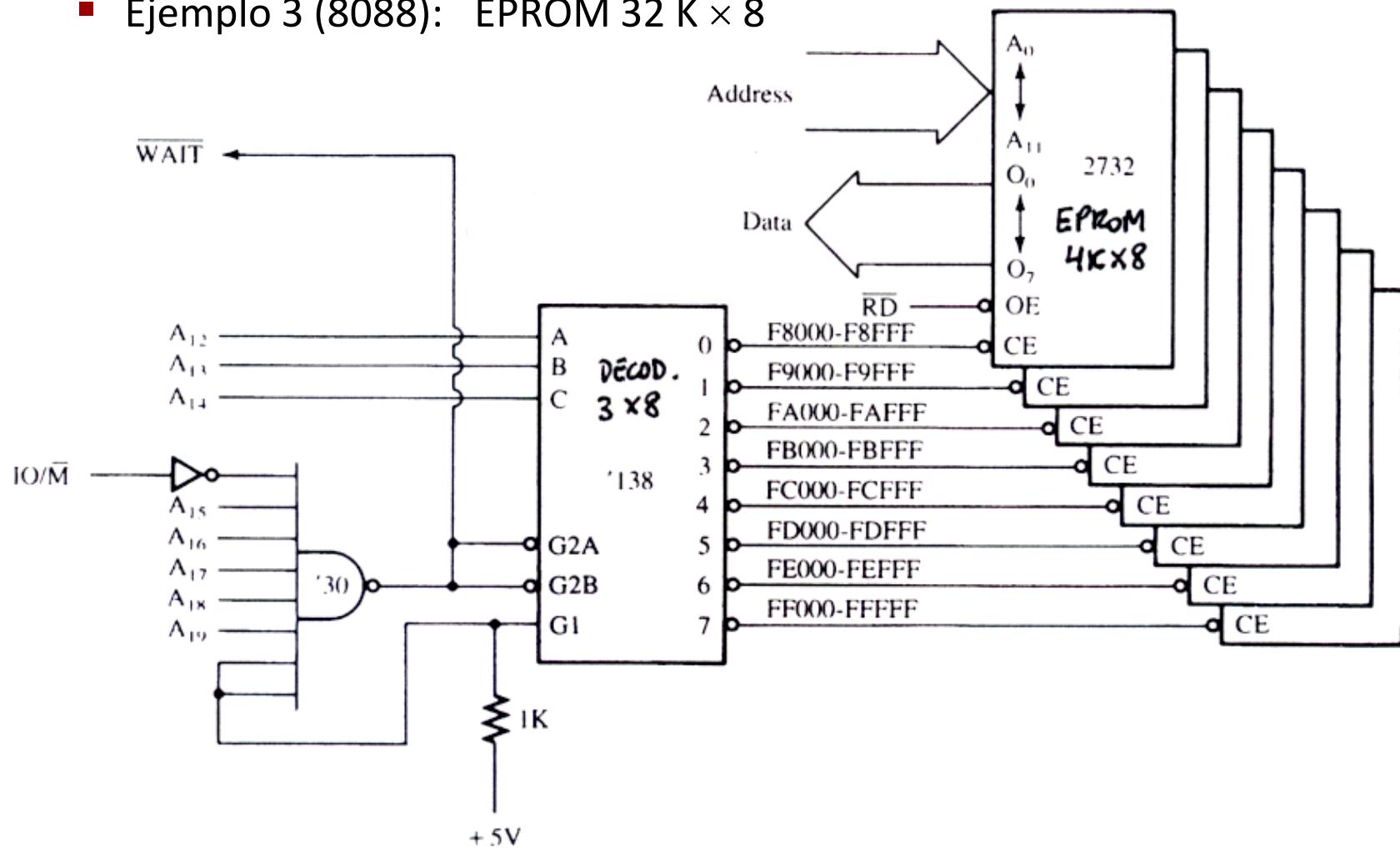
El multiplexor permite enviar a las pastillas la parte inferior o superior de los 16 bits de dirección.



Ejemplos de diseño

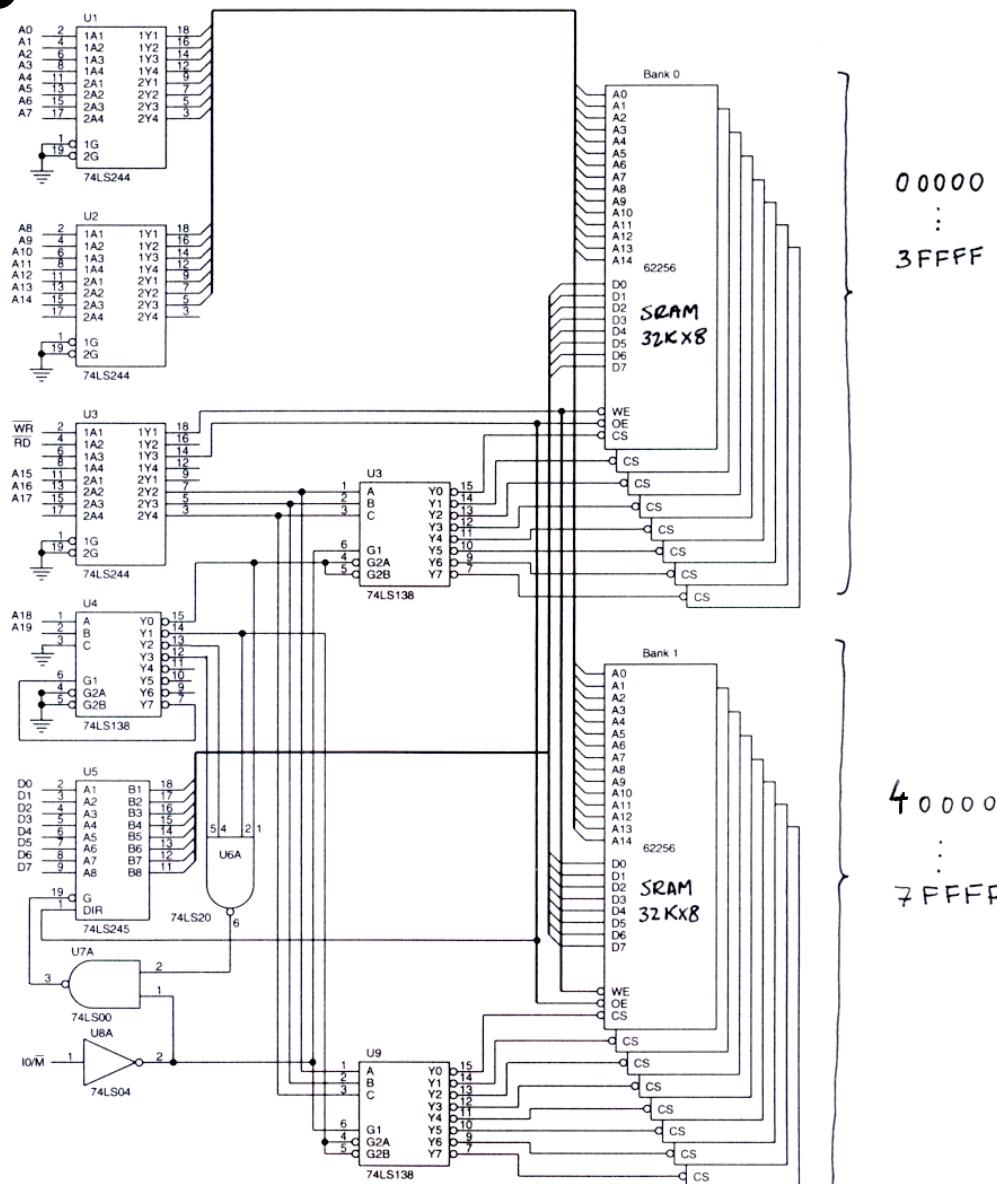
■ Configuraciones para μprocesadores Intel:

- Ejemplo 3 (8088): EPROM 32 K × 8



Ejemplos de diseño

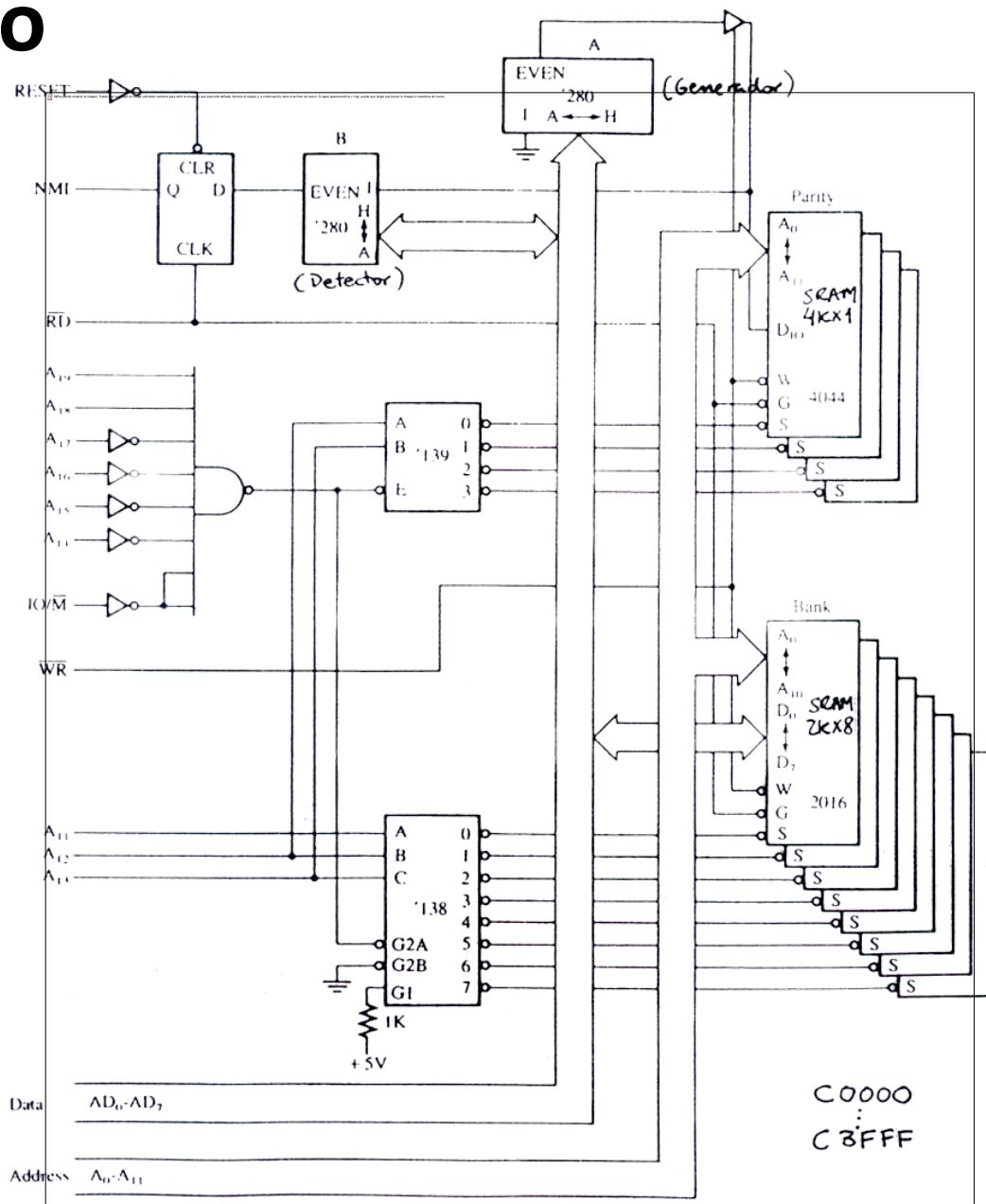
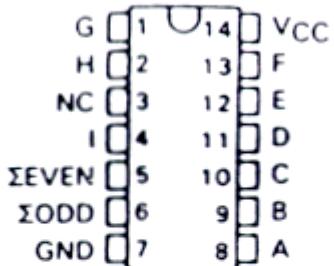
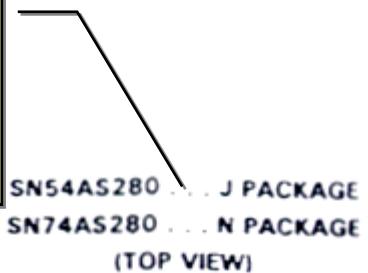
- Ejemplo 4 (8088):
SRAM 512 K × 8



Ejemplos de diseño

- Ejemplo 5 (8088):
 - SRAM 16 K × 8
 - Con paridad impar.

Circuito generador / detector de paridad.



C0000
:
C8FFF

Ejemplos de diseño

- Organización de memoria del 8086 / 286 / 386SX.

286/386SX (8MB)



286/386SX (8MB)

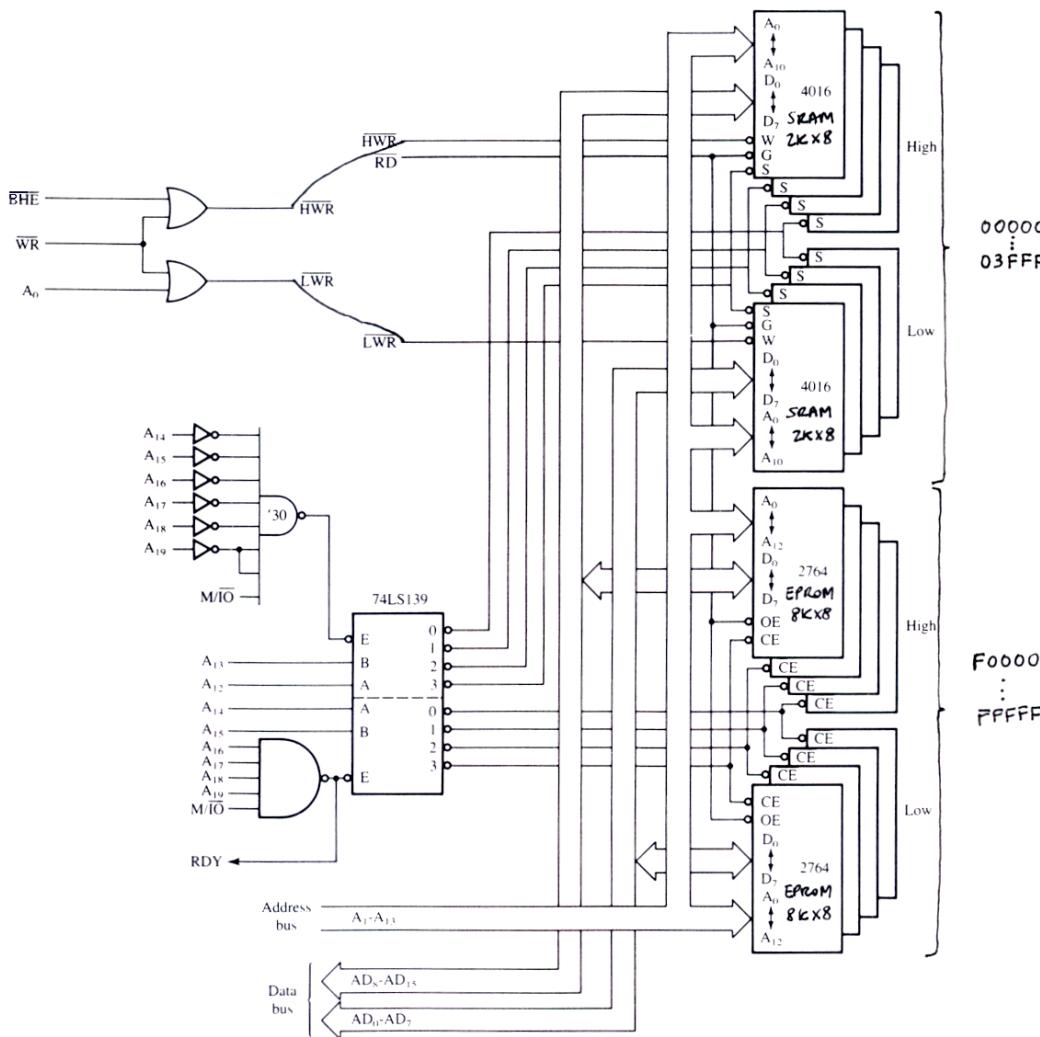


Note: A_0 is labeled \overline{BLE} (BUS Low enable) on the 80386SX.

\overline{BHE}	A_0	Function
0	0	Both banks active (16-bit transfer through $D_{15}-D_0$)
0	1	High bank active (8-bit transfer through $D_{15}-D_8$)
1	0	Low bank active (8-bit transfer through D_7-D_0)
1	1	No banks active

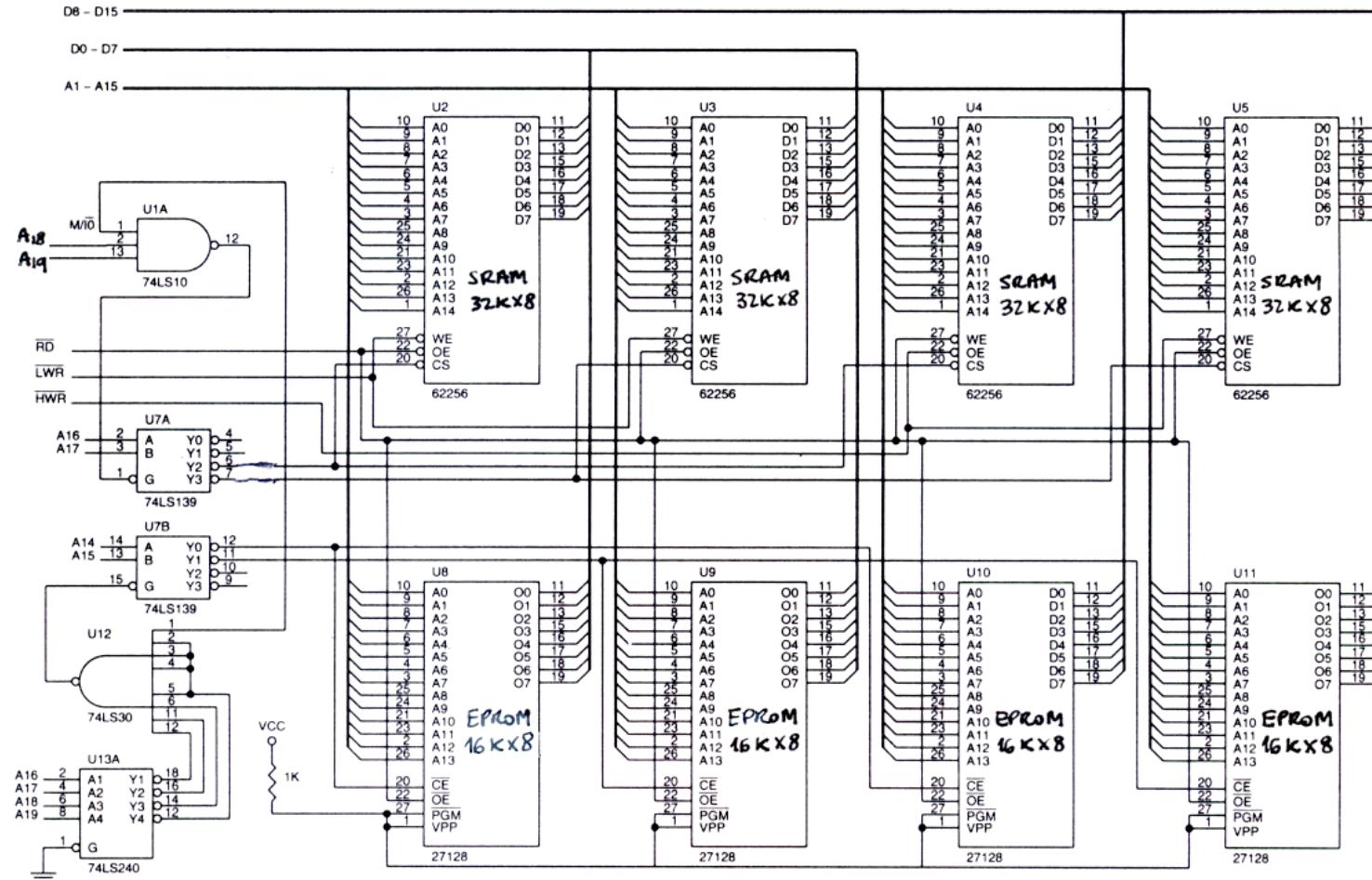
Ejemplos de diseño

- Ejemplo 6 (8086) SRAM 8 K × 16 + EPROM 32 K × 16



Ejemplos de diseño

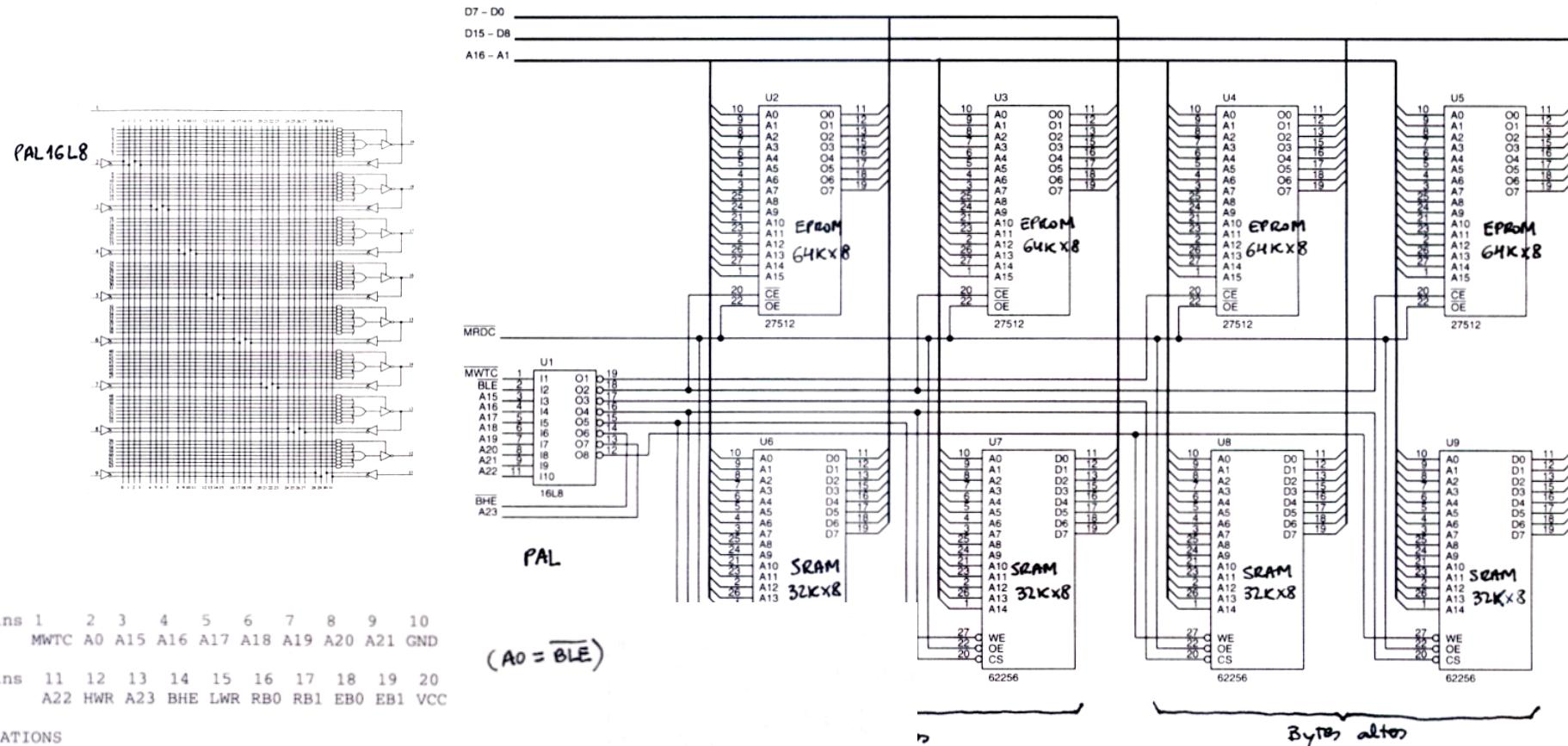
- Ejemplo 7 (8086) SRAM 64 K × 16 + EPROM 32 K × 16



EPROM: 00000-0FFFF
SRAM: E0000-FFFFF

Ejemplos de diseño

- Ejemplo 8 (386SX) EPROM 128 K × 16 + SRAM 64 K × 16



```

;pins 1 2 3 4 5 6 7 8 9 10
MWTC A0 A15 A16 A17 A18 A19 A20 A21 GND
;pins 11 12 13 14 15 16 17 18 19 20
A22 HWR A23 BHE LWR RB0 RB1 EB0 EB1 VCC

```

EQUATIONS

```

/LWR = /MWTC * /A0
/HWR = /MWTC * /BHE
/RB0 = /A23 * /A22 * /A21 * /A20 * /A19 * /A18 * /A17 * /A16
/RB1 = /A23 * /A22 * /A21 * /A20 * /A19 * /A18 * /A17 * A16
/EB0 = A23 * A22 * A21 * A20 * A19 * A18 * /A17
/EB1 = A23 * A22 * A21 * A20 * A19 * A18 * A17

```

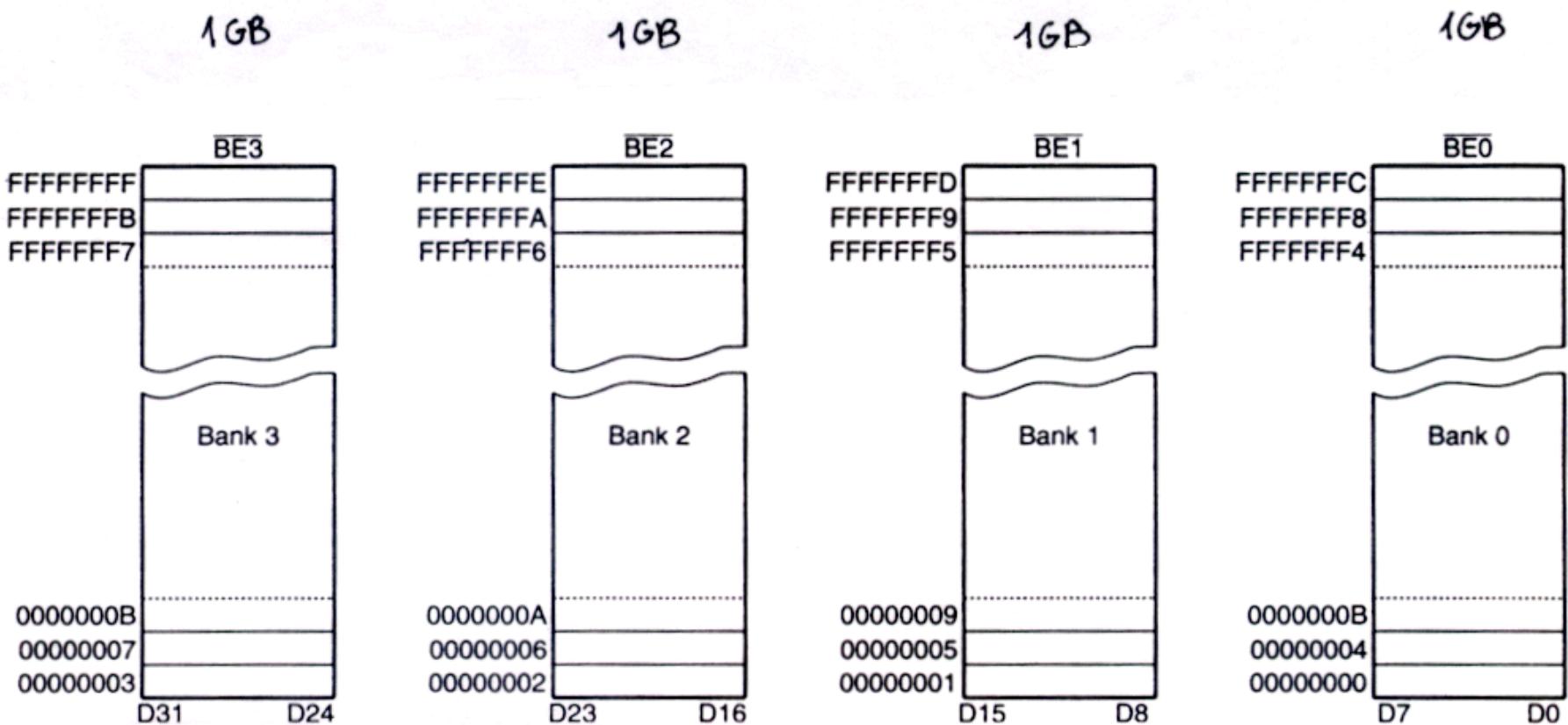
} EPROM : F0000
 } SRAM : 00000
 } : 01 PFFF
 } : FFFFFF

00 0000
:
01 PFFF

Bytes altos

Ejemplos de diseño

- Organización de memoria del 386DX / 486.



Ejemplos de diseño

- Ejemplo 9 (386 DX / 486) SRAM 64 K × 32

CHIP Decoder1U1 PAL16L8
 :pins 1 2 3 4 5 6 7 8 9 10
 MWTC BE0 BE1 BE2 BE3 A17 A28 A29 A30 GND
 :pins 11 12 13 14 15 16 17 18 19 20
 A31 RB1 U2 NC WR0 WR1 WR2 WR3 RB0 VCC

EQUATIONS

$$\begin{aligned} \text{/WR0} &= \text{/MWTC} * \text{/BE0} \\ \text{/WR1} &= \text{/MWTC} * \text{/BE1} \\ \text{/WR2} &= \text{/MWTC} * \text{/BE2} \\ \text{/WR3} &= \text{/MWTC} * \text{/BE3} \\ \text{/RB0} &= \text{/A31} * \text{/A32} * \text{/A30} * \text{/A29} * \text{/A28} * \text{/A17} * \text{/U2} \\ \text{/RB1} &= \text{/A31} * \text{/A32} * \text{/A30} * \text{/A29} * \text{/A28} * \text{/A17} * \text{/U2} \end{aligned}$$

0 0 0 0 0 0 0/4

CHIP Decoder1U2 PAL16L8

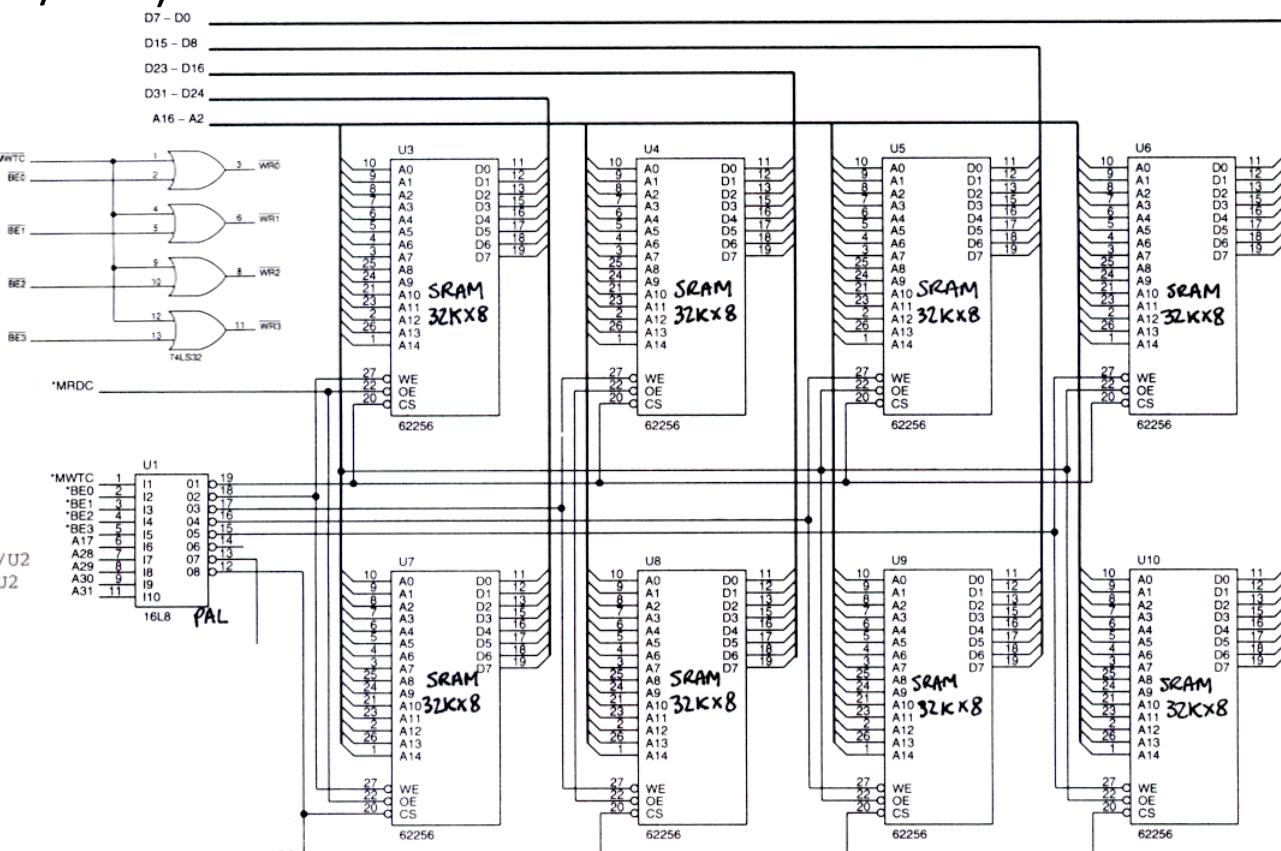
:pins 1 2 3 4 5 6 7 8 9 10
 A18 A19 A20 A21 A22 A23 A24 A25 A26 GND
 :pins 11 12 13 14 15 16 17 18 19 20
 A27 U2 NC NC NC NC NC NC NC VCC

EQUATIONS

$$\text{/U2} = \text{/A27} * \text{/A26} * \text{/A25} * \text{/A24} * \text{/A23} * \text{/A22} * \text{/A21} * \text{/A20} * \text{/A19} * \text{/A18}$$

0 0 1 0 0 0 0 0 0

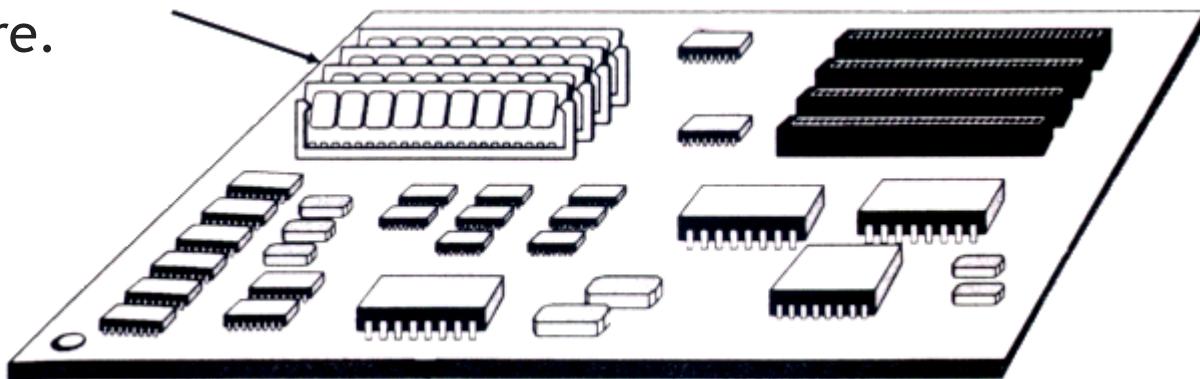
0200 0000
 :
 0203 FFFF { Range de direcciones



Módulos de memoria en línea

- En los 80, la memoria solía soldarse directamente en la placa madre del ordenador. Pero a medida que aumentaron los requisitos de memoria, esta técnica resultó poco factible.
- SIMM, DIMM, SODIMM, RIMM, SORIMM:

- Varios chips DRAM en una pequeña placa de circuito impreso (PCB) que calza en un conector en la placa madre.



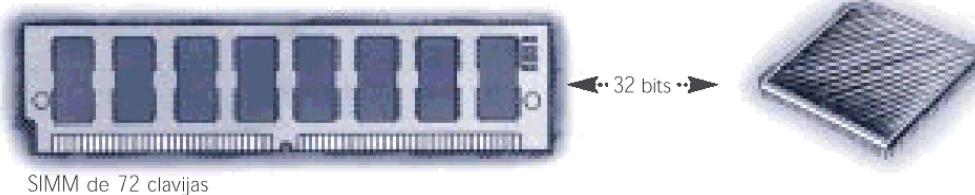
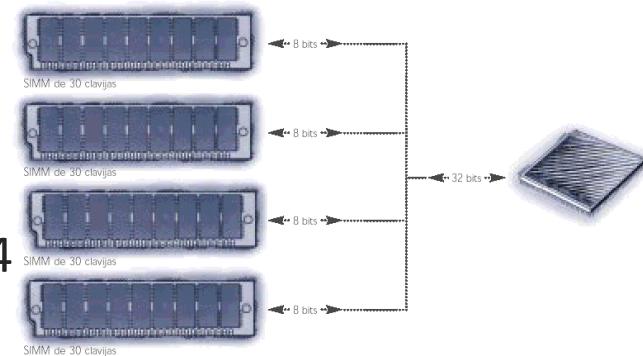
- ✓ método flexible para actualizar la memoria
- ✓ ocupa menos espacio en la placa madre.
- Normalmente sólo se amplía el bus de datos, no el nº de direcciones (no hay necesidad de decodificador).

Módulos de memoria SIMM



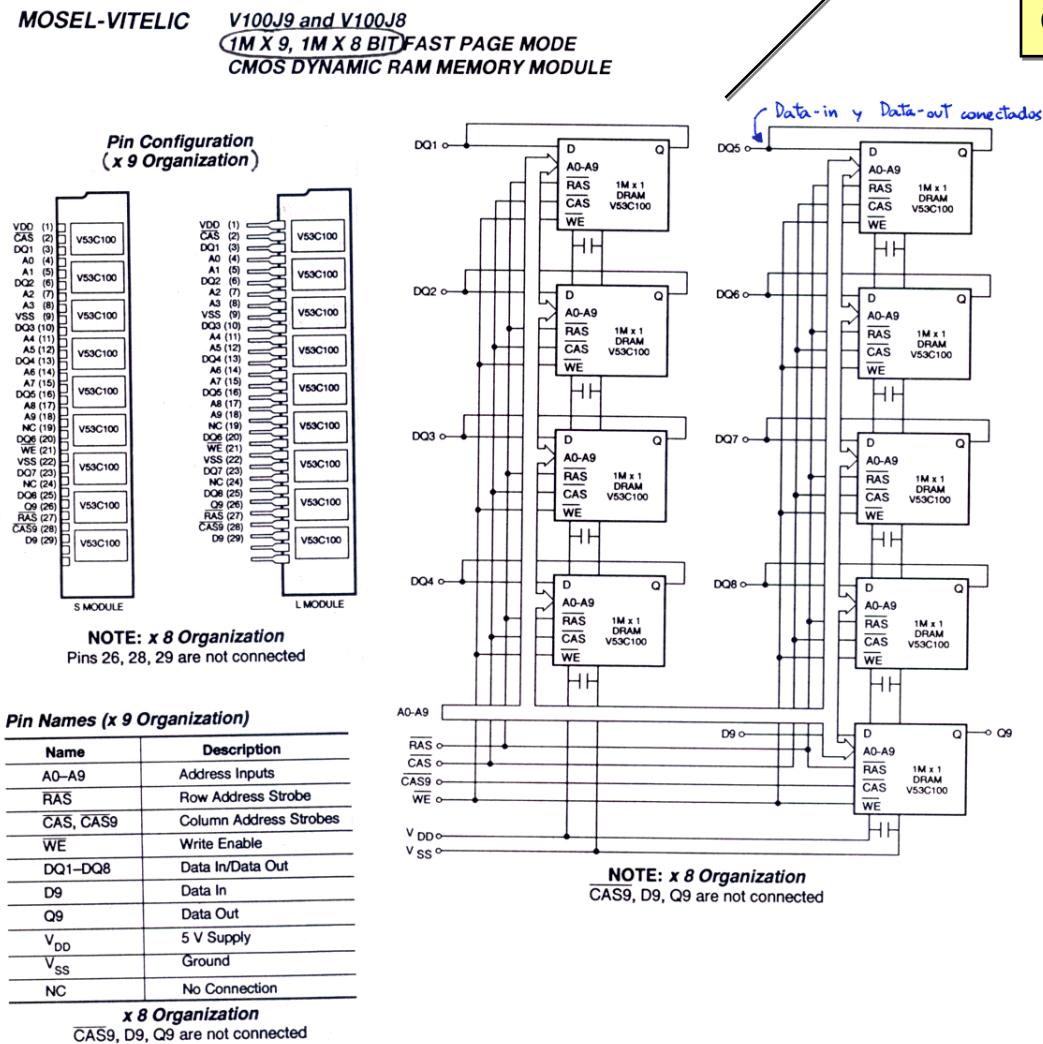
■ **SIMM (Single In-line Memory Module)**

- FPM y EDO
- 30 contactos:
 - 8 bits de datos
 - Si la placa madre tiene conectores para SIMM de 30 contactos \Rightarrow se necesitarán 4 SIMM para obtener 32 bits.
 - En un sistema de esta clase, la configuración de la memoria típicamente se divide entre dos bancos de memoria: el banco cero y el banco uno. Cada banco de memoria consiste en cuatro conectores de SIMM de 30 contactos. La CPU se dirige a un banco de memoria a la vez.
- 72 contactos:
 - 32 bits de datos.
 - Un único SIMM por banco.



Módulos de memoria SIMM

- Ejemplo de SIMM de 30 contactos:

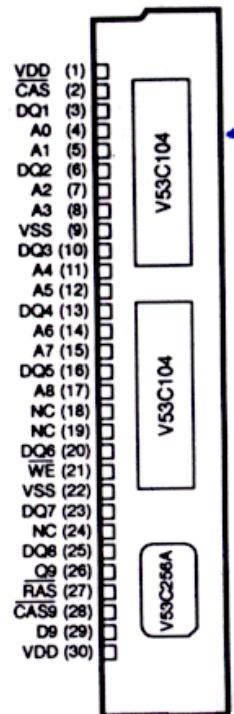


Módulos de memoria SIMM

- Más ejemplos de SIMM de 30 contactos:

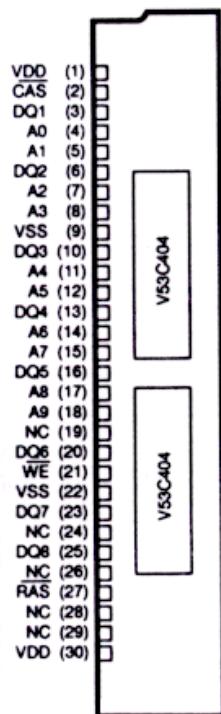
256KB

256 K X 8
256 K X 9



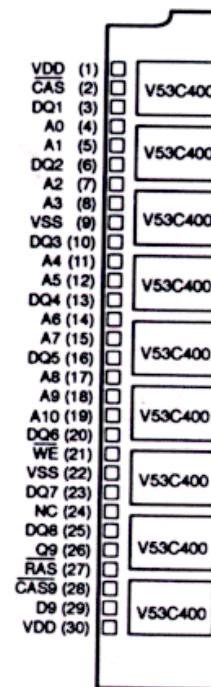
1MB

1 M X 8
1 M X 9



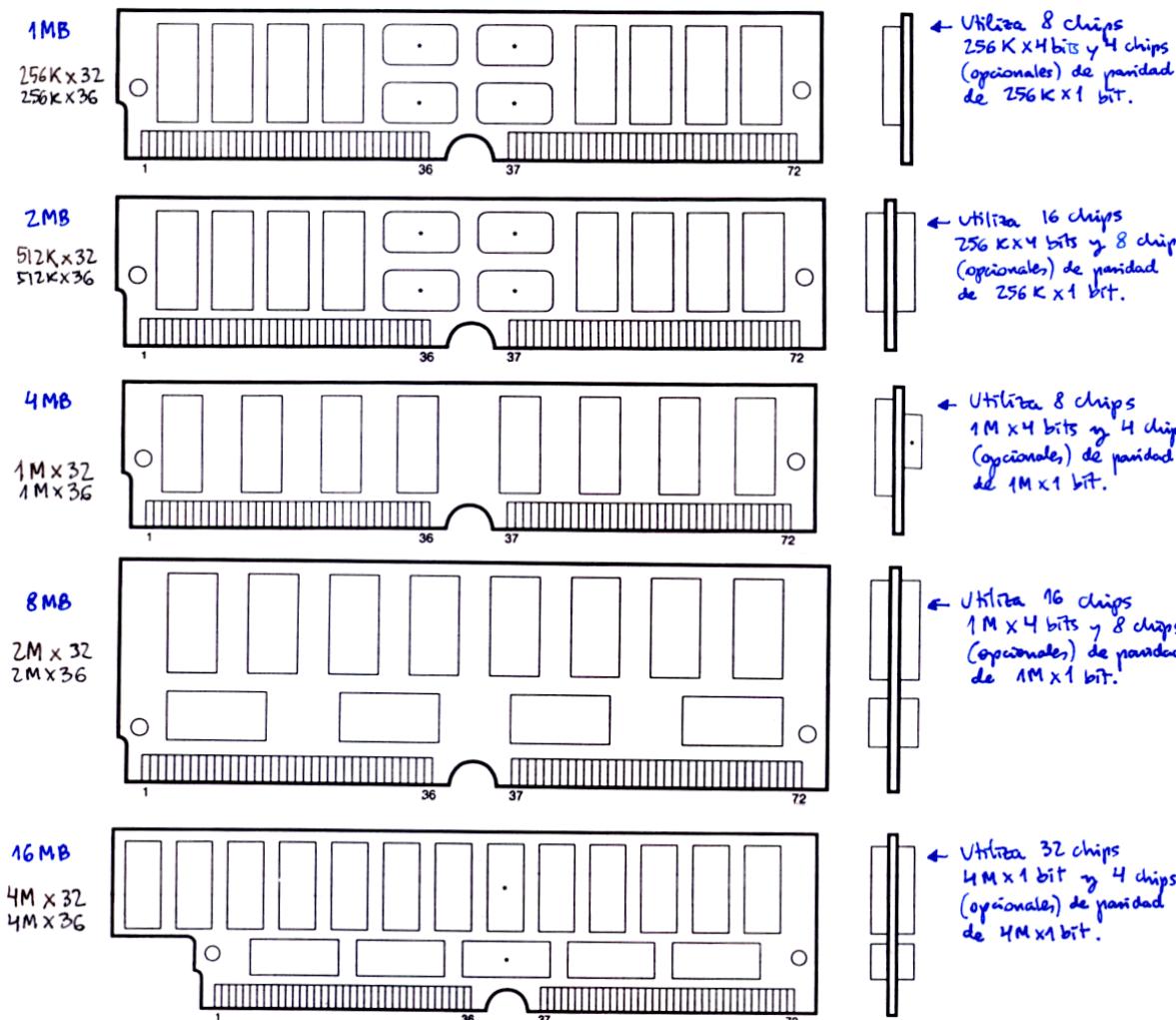
4MB

4 M X 8
4 M X 9



Módulos de memoria SIMM

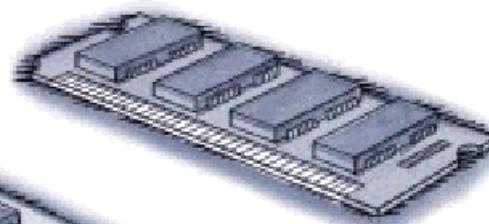
- Ejemplos de SIMM de 72 contactos:



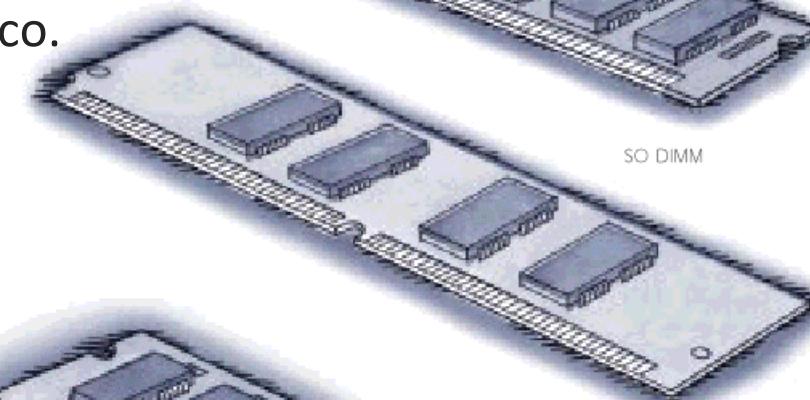
Módulos de memoria DIMM

■ **DIMM (Dual In-line Memory Module)**

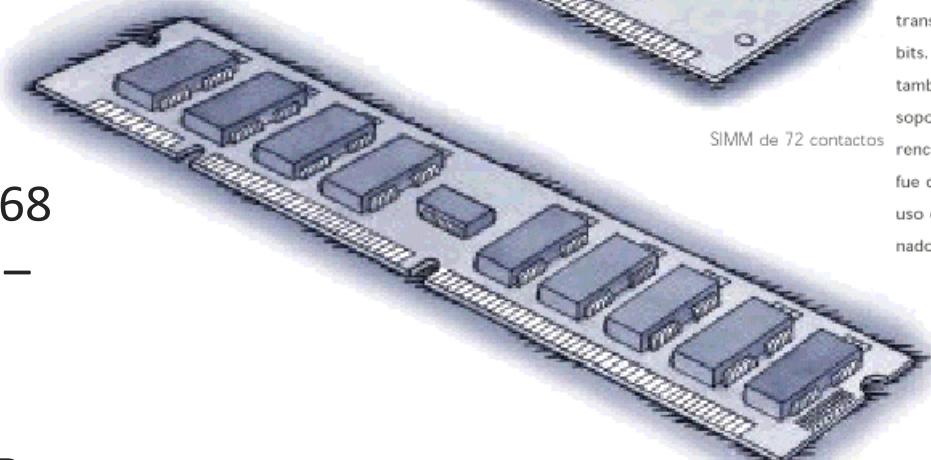
- En un SIMM, los contactos de cada fila se unen con los de la otra fila para formar uno único.
- En DIMM los contactos opuestos están aislados eléctricamente para formar **dos contactos separados**.
- 64 bits (ó 72 con ECC)
- Dos tipos de DIMM:
 - **FPM, EDO y SDRAM**, 168 contactos. Ej. Pentium – Pentium III.
 - **DDR-SDRAM**, 184 contactos. Ej. Athlon XP.



SO-DIMM



SIMM de 72 contactos



DIMM de 168 contactos

Los tres ejemplos ilustran las diferencias entre los productos SIMM, DIMM y SO-DIMM. El DIMM de 168 contactos brinda soporte para transferencias de 64 bits, sin duplicar el tamaño del SIMM de 72 contactos, el cual brinda soporte sólo para transferencias de 32 bits. El SO-DIMM también brinda soporte para transferencias de 32 bits y fue diseñado para su uso en los ordenadores portátiles.

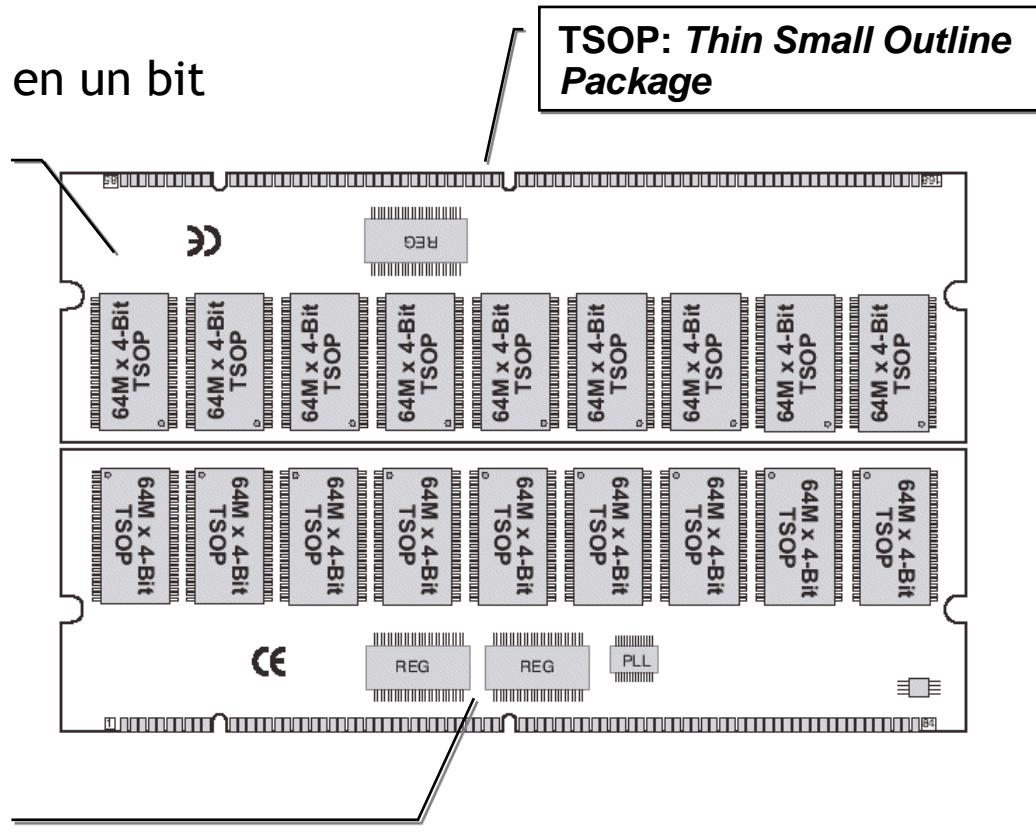
Módulos de memoria RIMM

- Ejemplo: DIMM de 512 MB + ECC ($64M \times (64+8 \text{ (ECC)})$) a 133 MHz.
 - ECC (*Error Checking and Correct / Error-Correcting Code*)
 - Detecta errores en 2, 3 o hasta 4 bits, dependiendo del controlador
 - Corrige error en un bit

18 chips (9 en cada cara)
SDRAM a 133 MHz de $64M \times 4$

Bus de datos: 72 (64 + 8) bits
 $= 18 \text{ chips} \times 4 \text{ bits/chip}$

Registros: amplifican las señales provenientes del chipset en módulos SDRAM grandes



Módulos de memoria SORIMM



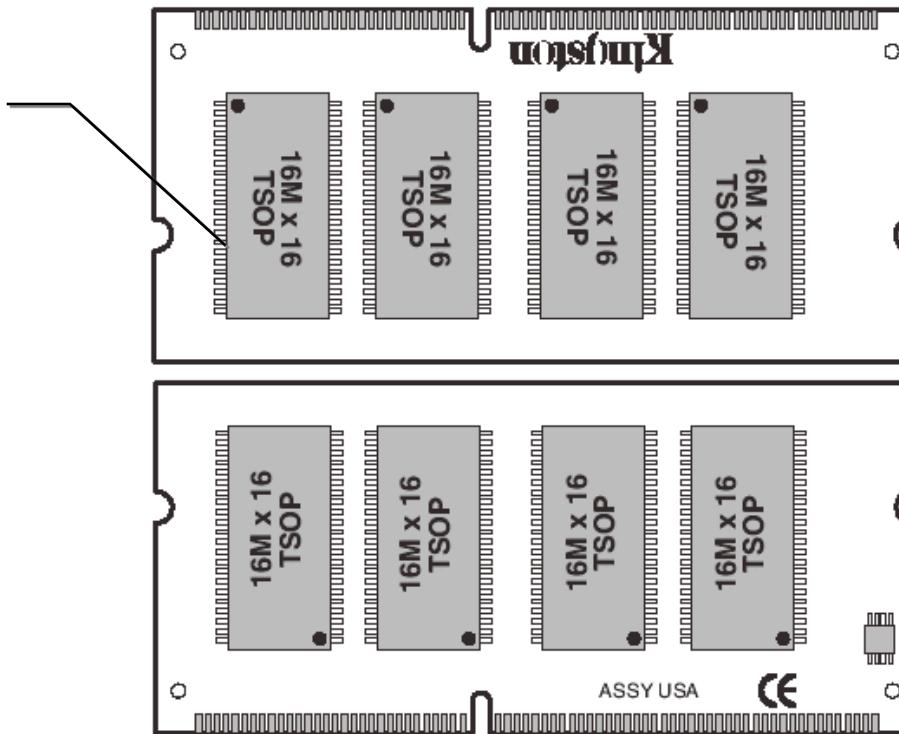
■ SODIMM (*Small Outline DIMM*)

- Mitad de tamaño que un DIMM, 144 contactos.
- Uso en portátiles.
- Ejemplo: SODIMM de 256 MB ($2 \times 16M \times 64$) a 133 MHz:

**8 chips (4 en cada cara)
SDRAM a 133 MHz de
 $16M \times 16$**

**Cada chip tiene 4
bancos de $4M \times 16$**

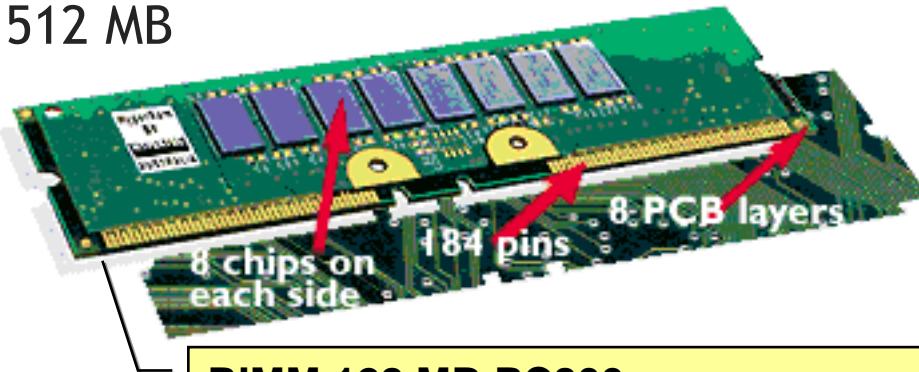
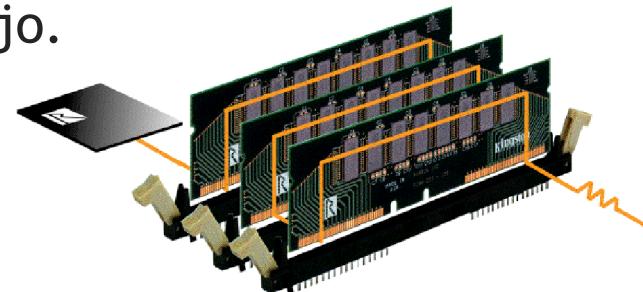
**Bus de datos: 64 bits =
4 chips \times 16 bits/chip
(¡en este caso sí se
amplía el número de
direcciones!)**



Módulos de memoria RIMM

■ **RIMM™ (Rambus In-line Memory Module)**

- Módulos de memoria RDRAM de 184 contactos encontrados en Pentium 4 y estaciones de trabajo.
- Transferencias de 16 bits.
- 1 a 32 chips RDRAM por RIMM.
- Los chips son independientes:
 - Ej.: RIMM de 8 chips y 16 bancos por chip \Rightarrow RIMM tiene 128 bancos.
- 64 MB, 128 MB, 256 MB, 512 MB

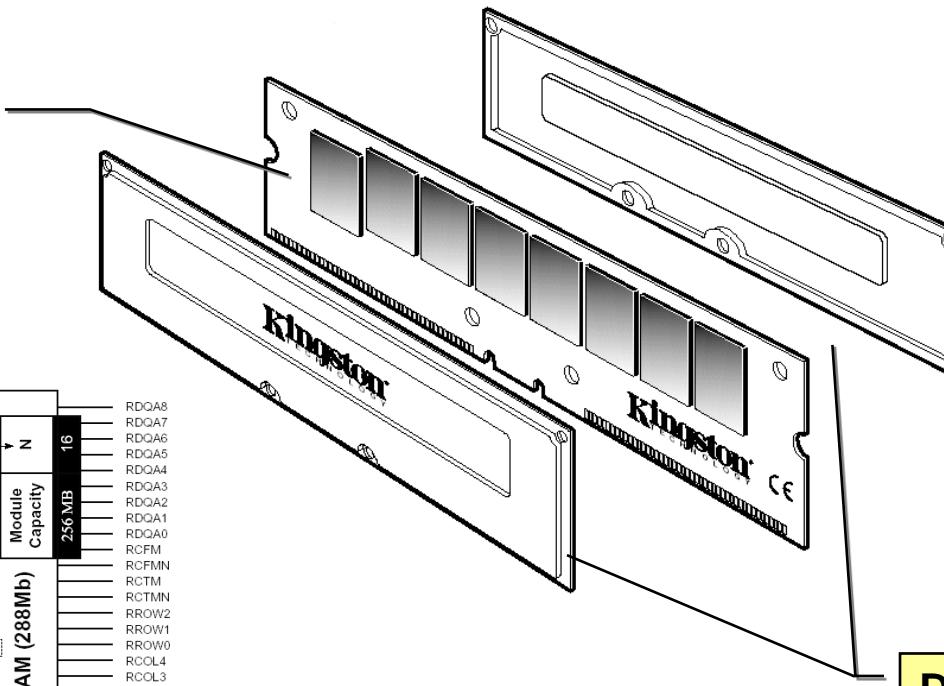
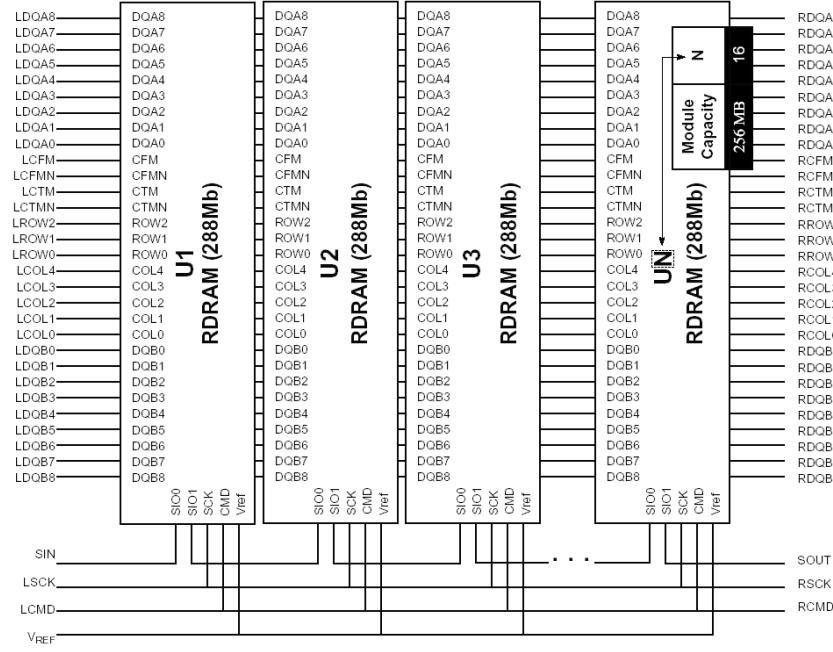


RIMM 128 MB PC800:
16 chips (8 en cada cara) de 4M x 16.

Módulos de memoria RIMM

- Ejemplo: RIMM de 512 MB (256M × 18 con ECC) a 800 Mbps / patilla:

**16 chips (8 en cada cara)
de 288 Mb (16M × 18).**

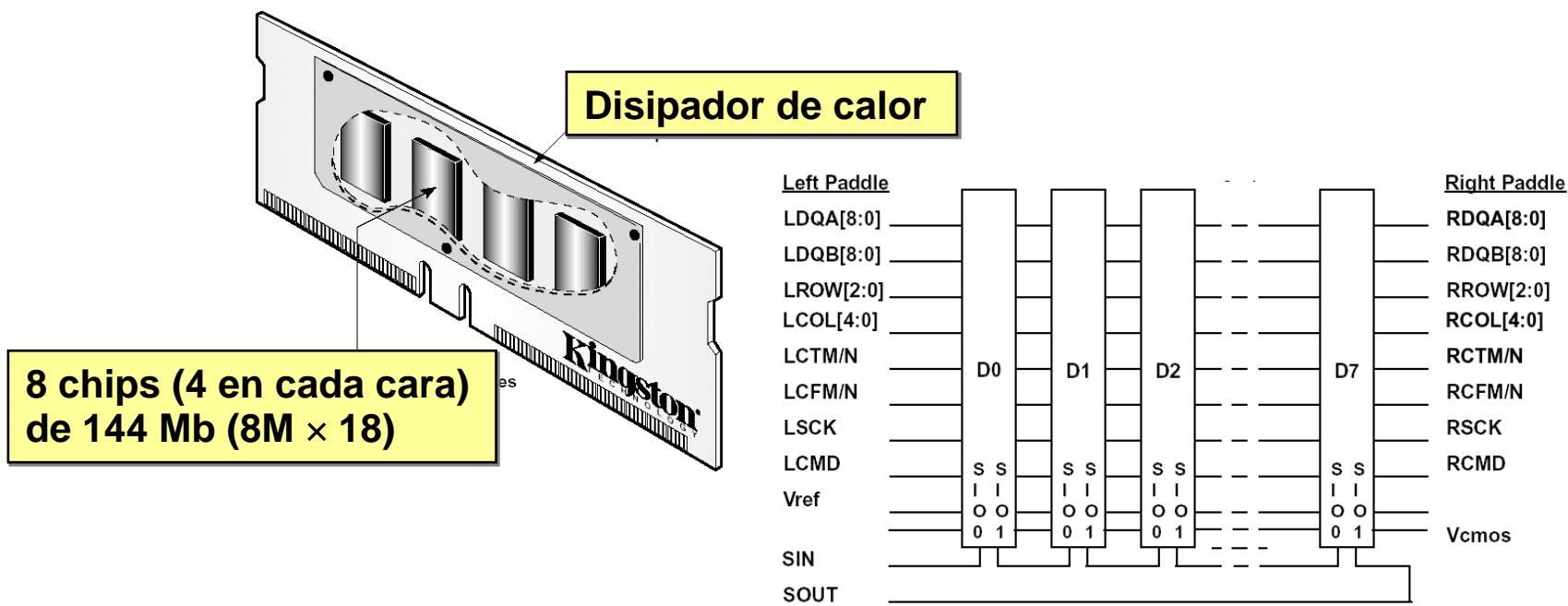


**Disipadores
de calor**

Módulos de memoria SORIMM

■ SORIMM™ (*Small Outline RIMM*)

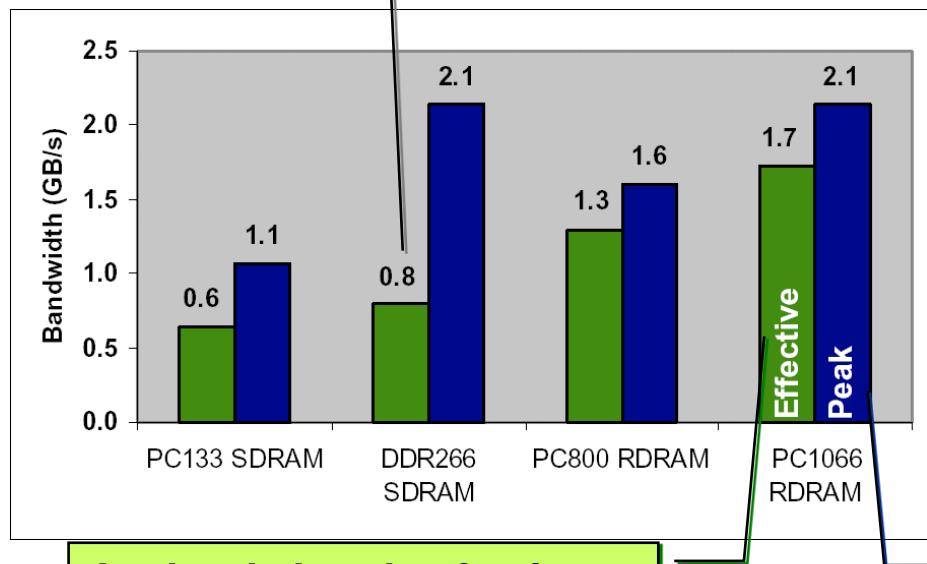
- Módulos RDRAM de 160 contactos para portátiles.
- Ejemplo: SO-RIMM de 128 MB (64M × 18 con ECC) a 800 Mbps / patilla:



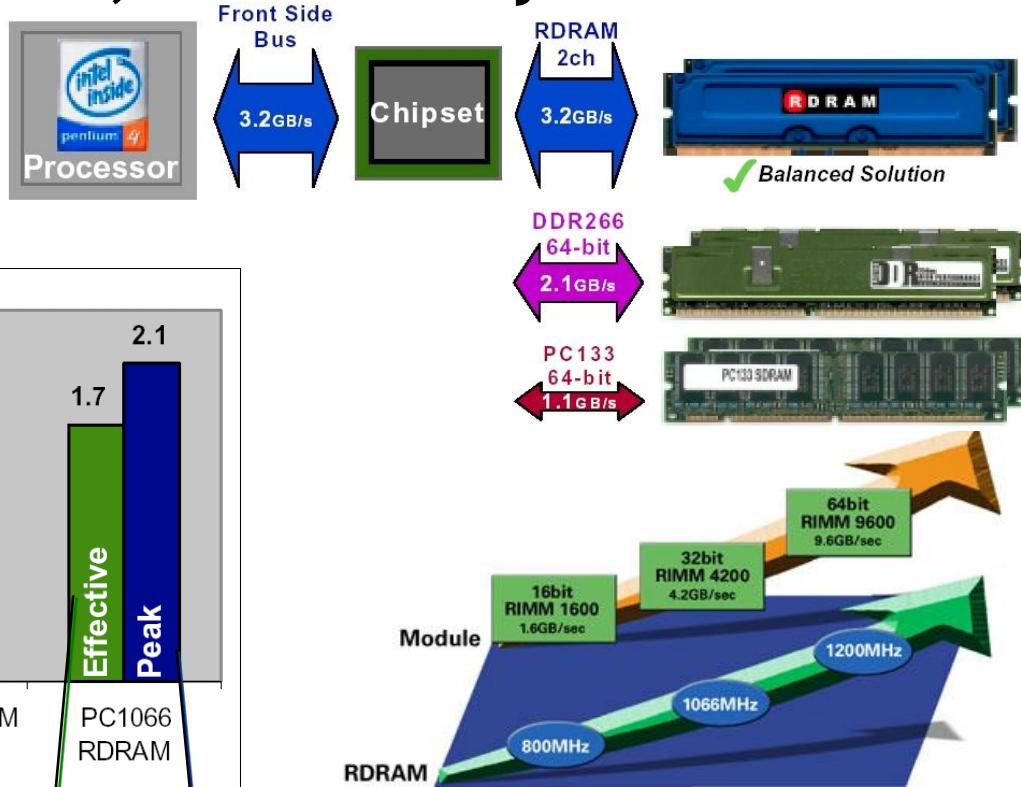
Módulos de memoria en línea

■ Comparación entre SDRAM, DDR SDRAM y RDRAM

¿Por qué el ancho de banda efectivo es menor en la SDRAM?



Ancho de banda efectivo =
Ancho de banda pico ×
eficiencia (% de ancho de
banda pico en un sistema real)

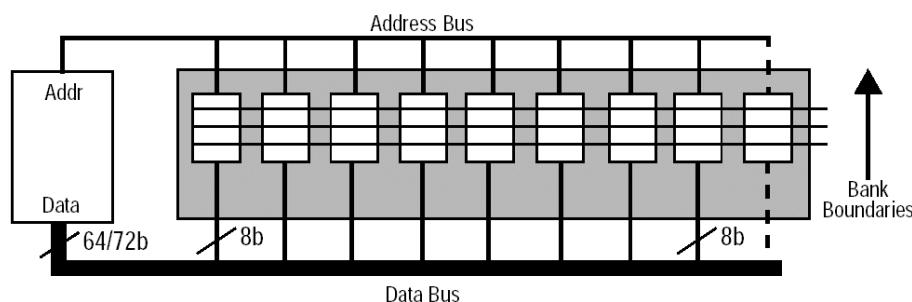


Ancho de banda pico =
Cantidad de datos máxima que
el sistema de memoria puede
proporcionar por unidad de
tiempo bajo supuestos ideales

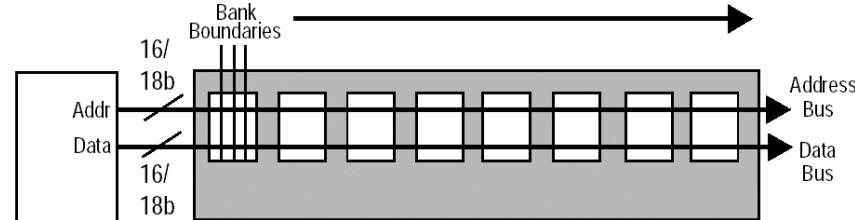
Módulos de memoria en línea

¿Por qué el ancho de banda efectivo es menor en la SDRAM?

SDRAM/DDR SDRAM Module: 8/9 - 32Mx8 Devices, 64b data bus



RDRAM Module: 8 - 256Mb, 4i Devices, Single Channel



- Address Bus loads faster than Data Bus
- 4 banks per device, 4 banks total, banks span across array, 9th device ECC
- **Probability of bank conflict:**

1 outstanding transaction:	1/4	(25%)
2 outstanding transactions:	2/4	(50%)

- Address & Data bus uniformly loaded
- 4 banks per device, 32 banks total, banks additive
- **Probability of bank conflict:**

1 outstanding transaction:	1/32	(3.1%)
2 outstanding transactions:	2/32	(6.3%)

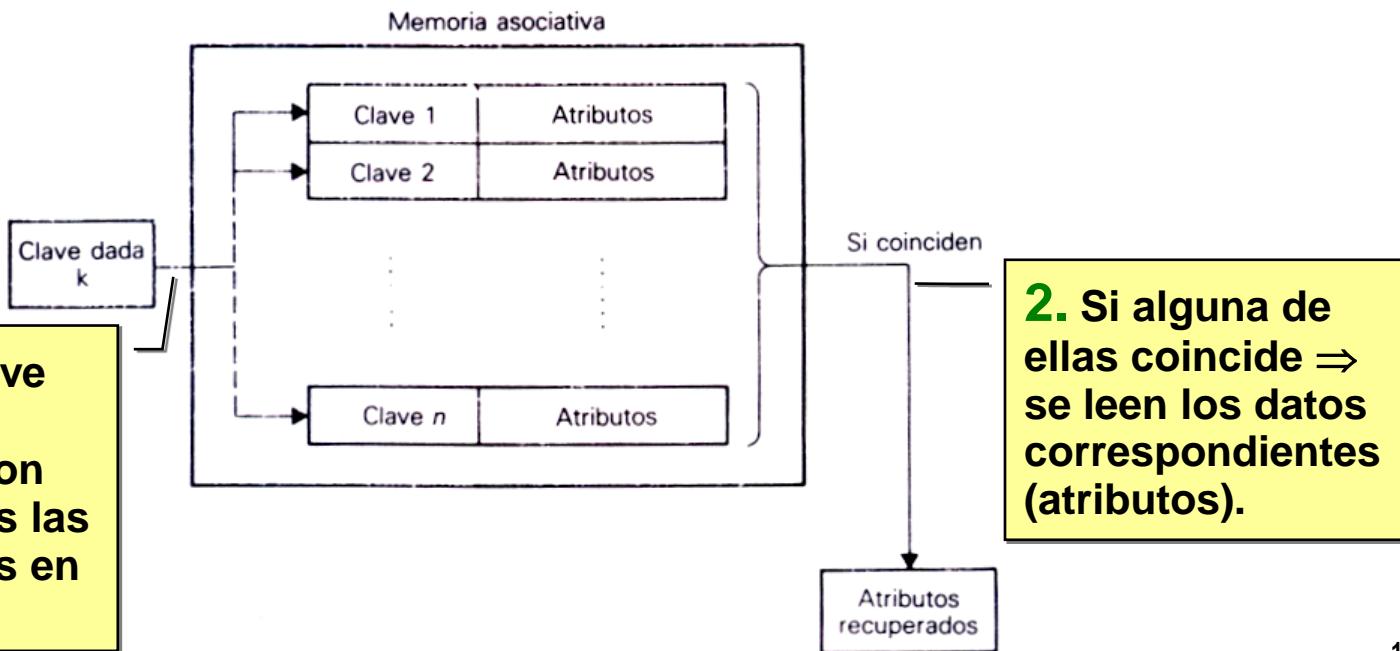
✗ Más conflictos entre bancos al haber menos bancos

Memoria

- **Jerarquía de memoria. Concepto de localidad**
- **Memorias RAM semiconductoras. Memorias de sólo lectura. Prestaciones: velocidad, tamaño y coste**
- **Configuración y diseño de memorias utilizando varios chips**
- **Memorias asociativas**
- **Memoria cache. Influencia en las prestaciones**

Concepto de memoria asociativa

- También llamada:
 - Memoria direccionable por contenido (CAM, *Content Addressable Memory*).
 - Memoria de búsqueda paralela.
 - Memoria multiacceso.
- Es una estructura hardware que permite efectuar operaciones de acceso a los datos (búsqueda, comparación) a gran velocidad.



Concepto de memoria asociativa

✓ El tiempo de búsqueda es muy pequeño ya que el hardware efectúa todas las comparaciones simultáneamente.

La búsqueda se efectúa sólo a partir del contenido de la clave, y no de su dirección.



La información ha de estar en una matriz de memoria diseñada específicamente para permitir ser accedida por contenido.



✗ Coste hardware mucho mayor que el de una RAM.

Concepto de memoria asociativa

Ejemplo:

Nombre	Peso	Talla	Edad
P. Pérez	63	1,71	27
M. García	90	1,87	45
M. Ortega	82	1,83	33
A. Álvarez	51	1,64	61

Tabla almacenada en memoria.

- Si utilizamos una mem. de acceso aleatorio convencional:
 - Necesitamos especificar una **dirección física** para poder acceder a su información asociada.
 - Esta dirección física no tiene **ninguna relación lógica** con la información que se almacena.
 - El método de acceso es artificial y **complica la programación**.
 - Por ejemplo, para ver quién mide 1,83 hay que realizar una **lenta búsqueda secuencial**.
- Alternativa (memoria asociativa):
 - Emplear uno de los campos de la tabla como clave para acceso al(a los) registro(s) que contiene(n) esa clave.

En general, una memoria asociativa tiene la capacidad de acceder a una palabra almacenada usando como dirección su contenido o el contenido de un subcampo de la misma.

Concepto de memoria asociativa

■ Operaciones (pensadas para bases de datos):

- Búsquedas extremales:
 - Valor mínimo, máximo, medio.
- Búsquedas de equivalencia:
 - Igual a, no igual a.
 - Similar a (valores idénticos en varios campos).
 - Próximo a (valores que satisfacen cierta condición de proximidad).
- Búsquedas por umbral:
 - Menor que, mayor que.
 - No menor que, no mayor que.
- Búsqueda entre límites.
 - Pertenencia a cierto intervalo (abierto/cerrado).
- Extracciones ordenadas:
 - Ascendentemente, descendentemente.

Concepto de memoria asociativa

■ Aplicaciones:

- Implementación de memorias caché asociativas.
- Implementación de TLB (buffer de traducción anticipada) en memoria virtual.
- Manipulación de información almacenada en bases de datos.
- Enrutadores de red.

Diseño de la memoria asociativa

Estructura de la memoria asociativa:

Operando que sirve de clave de búsqueda/comparación, o que contiene la información que se va a escribir.

Da las órdenes de comparación, lectura, y escritura.

Selecciona los bits del registro de entrada que intervendrán en la búsqueda. Un bit del registro de entrada se utilizará en el proceso de búsqueda si el correspondiente bit del registro de máscara está a 1.

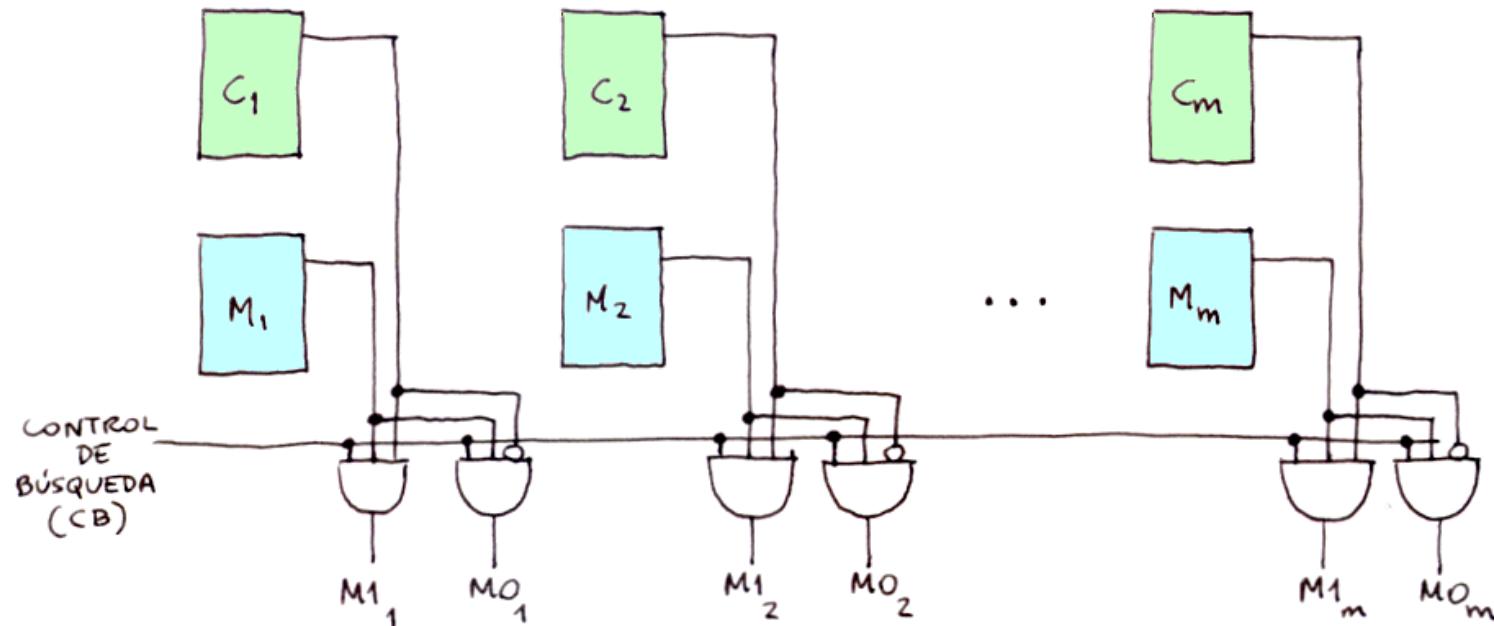
Contiene una de las palabras (o la suma lógica de varias) seleccionada por el proceso de búsqueda.



Colección de celdas básicas de memoria $S_{i,j}$, $1 \leq j \leq m$, cada una con un bit.

Ejemplo de diseño de CAM

■ Lógica de enmascaramiento:



CB	M_j	Líneas de búsqueda	
		$M0_j$	$M1_j$
0	X	0	0
1	0	0	0
1	1	$/C_j$	C_j

Ejemplo de diseño de CAM

■ Lógica de comparación:

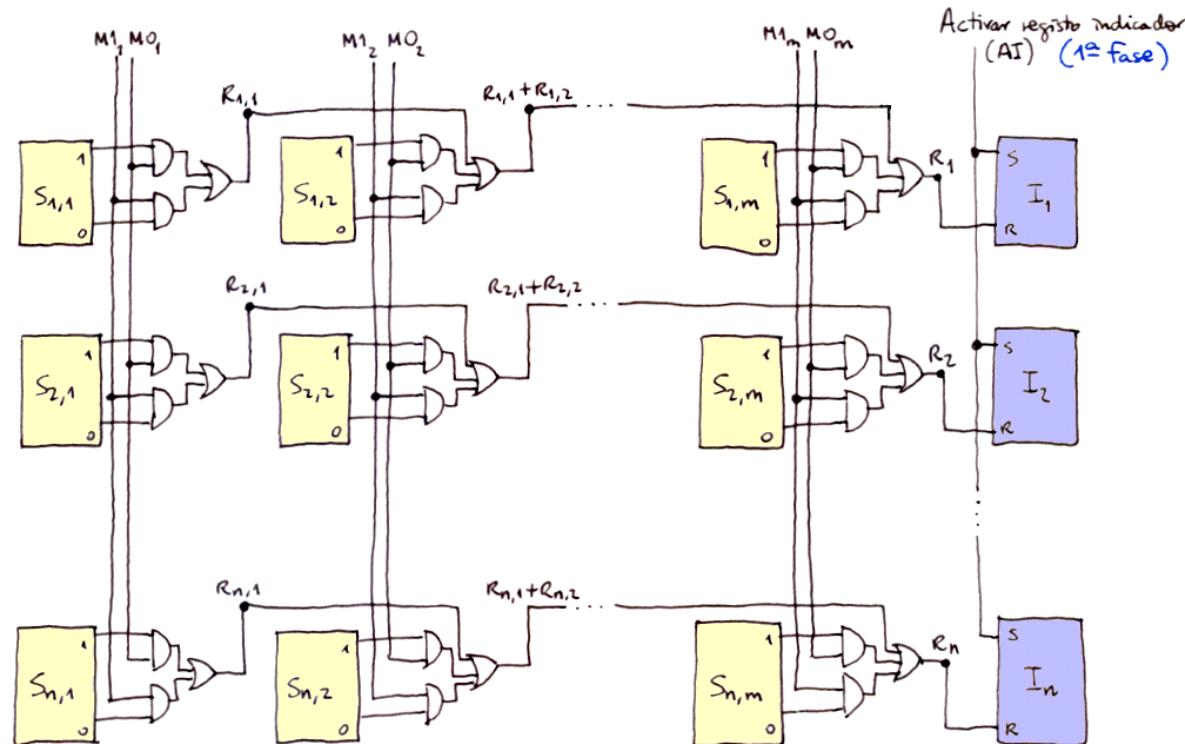
$M_{0,j}$	$M_{1,j}$	$S_{i,j}$	$R_{i,j}$
0	0	X	0
$/C_j$	C_j	C_j	0
$/C_j$	C_j	$/C_j$	1

$$R_i = \sum_{j=1}^m R_{i,j}$$

$$R_{i,j} = 1 \text{ SII } C_j \neq S_{i,j} \text{ y } M_j = 1 \text{ y } CB = 1$$

R_i : señal de desactivación de I_i

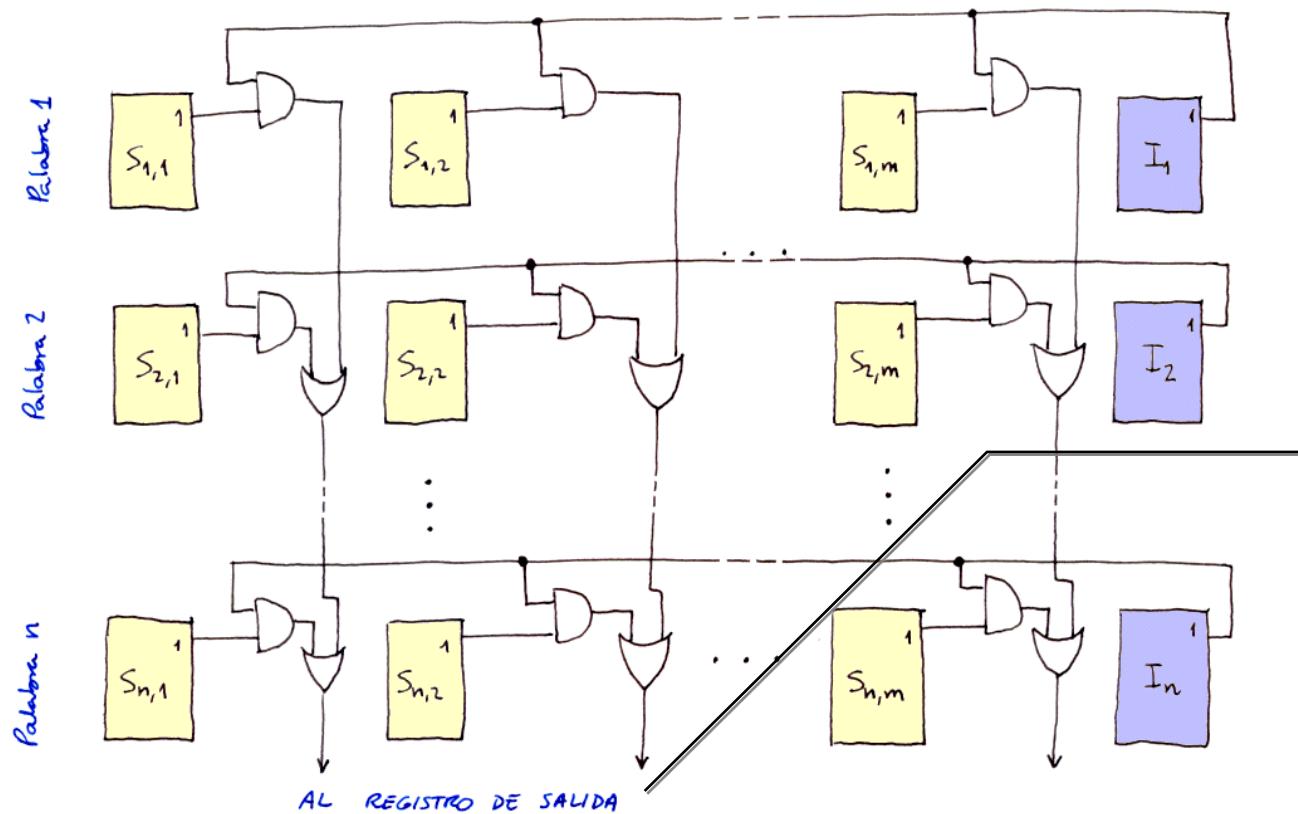
La desactivación del bit I_i ocurre si existe algún bit $S_{i,j}$ de la palabra i que no coincide con C_j . Los bits de C enmascarados no afectan al proceso de comparación.



Ejemplo de diseño de CAM

■ Lectura de una memoria asociativa:

- Dos fases:
 - 1^a: Comparación que deja activos los bits I_i de las celdas a leer.
 - 2^a: Extracción de la información.



El registro de salida contendrá la suma lógica de los contenidos de todas las celdas seleccionadas.

Ejemplo de diseño de CAM

- Si sólo se desea leer la primera palabra:

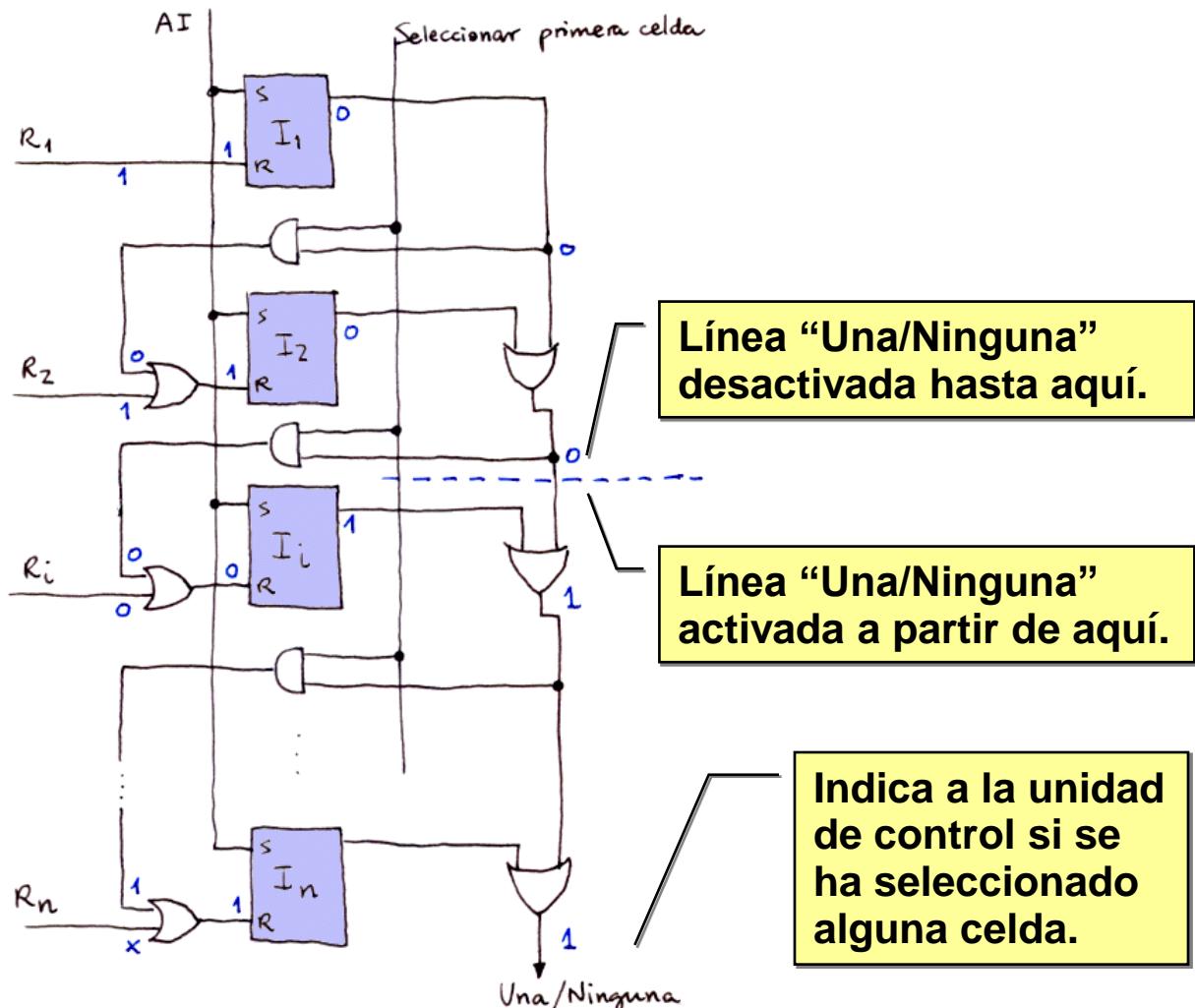
$$I_k=0, k < i$$

$$I_i=1$$

$I_k=0, k>i$ (se desactivan estos bits)

Sólo la palabra i permanece seleccionada y es la que se lee.

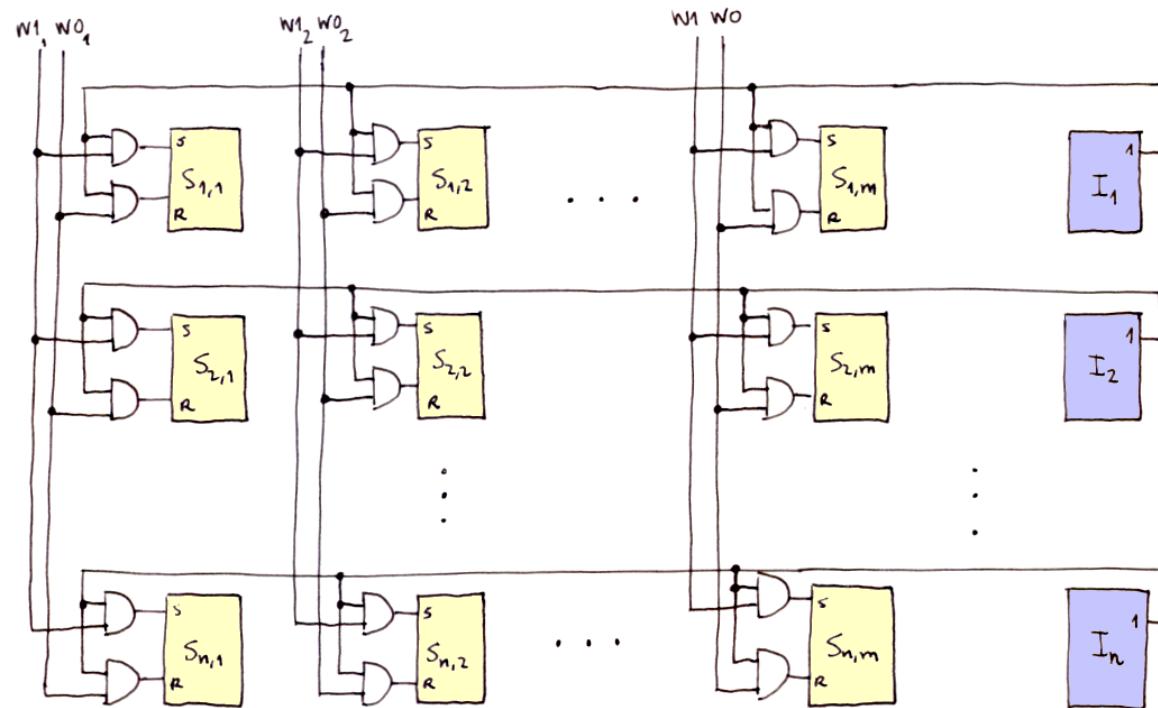
Una vez leída la palabra i -ésima, puede ser marcada de alguna forma. Posterioras comparaciones idénticas permitirán encontrar nuevas palabras.



Ejemplo de diseño de CAM

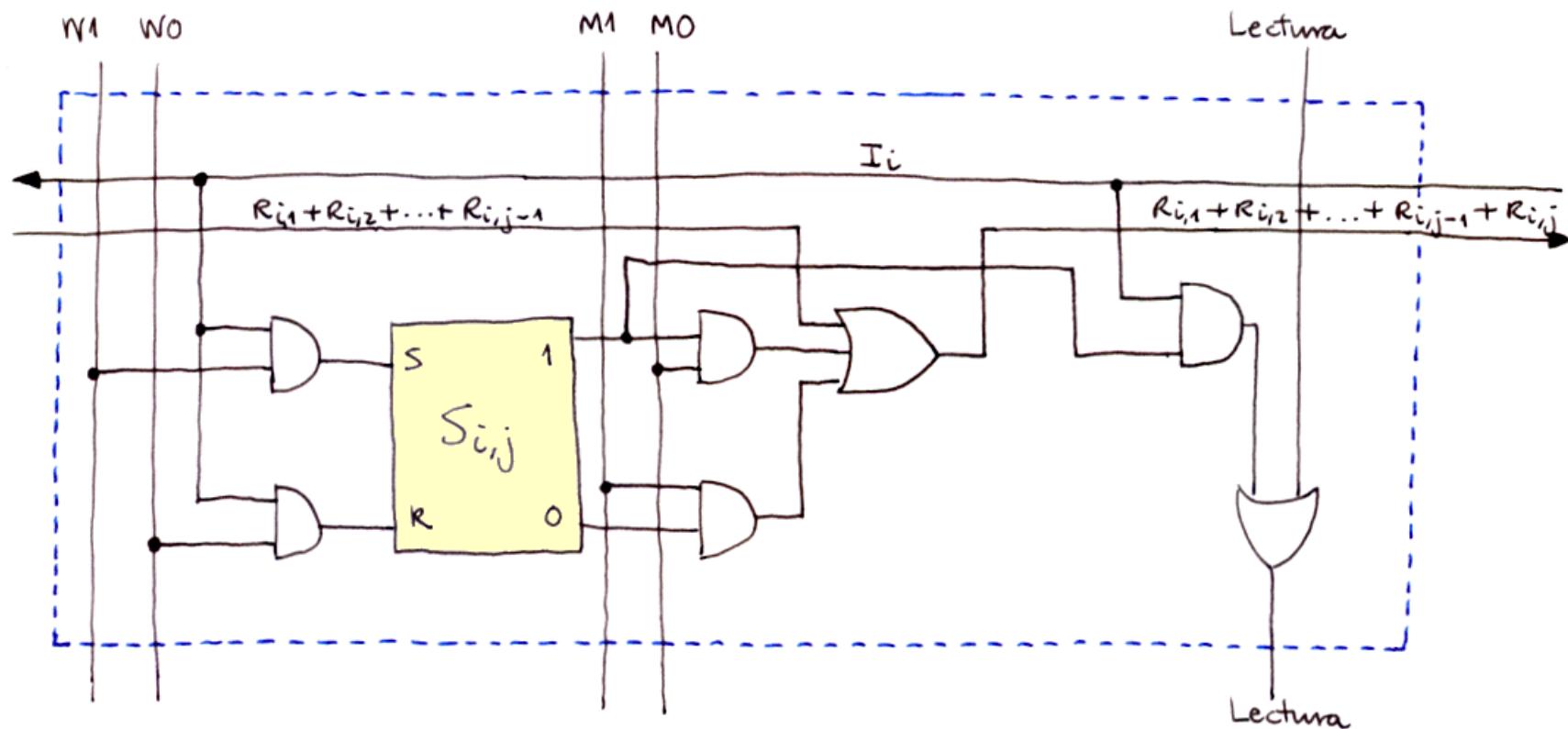
■ Escritura en una memoria asociativa:

- W1, W0 son líneas que llevan la información a escribir.
 - Se generan de forma semejante a las líneas de búsqueda M1, M0 (sustituyendo la señal de control de búsqueda CB por una señal de control de escritura).
- En los bits donde la máscara está desactivada no se escribe.



Ejemplo de diseño de CAM

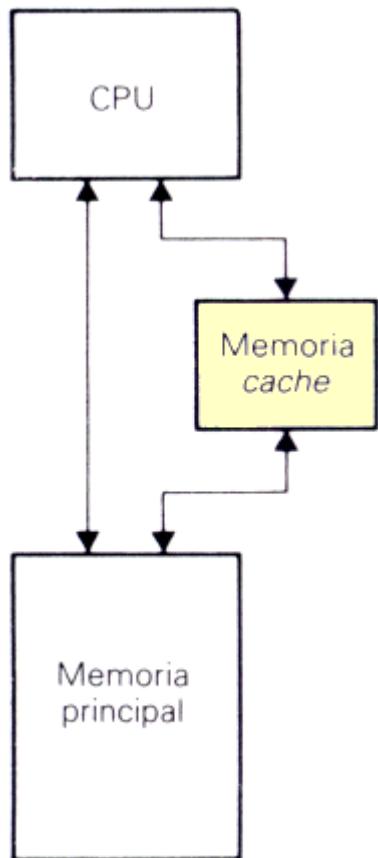
- Estructura global de una celda de un bit:



Memoria

- **Jerarquía de memoria. Concepto de localidad**
- **Memorias RAM semiconductoras. Memorias de sólo lectura. Prestaciones: velocidad, tamaño y coste**
- **Configuración y diseño de memorias utilizando varios chips**
- **Memorias asociativas**
- **Memoria cache. Influencia en las prestaciones**

Caché: concepto



- Memoria pequeña (8KB - 6MB) y rápida (5-10 veces más rápida que MP) situada entre procesador y MP.
 - Niveles más elevados de la jerarquía de memoria.
 - 1^a caché: IBM 360/85, 2^a mitad de los 60.
- En la caché se almacenan aquellas palabras de la MP actualmente en uso.
 - El procesador buscará la información en la caché.
 - Se encuentra en la caché → **acíerto o “cache hit”**
 - Los datos se transfieren de caché al procesador.
 - No se encuentra en la caché → **falta o “cache miss”**
 - Se busca en la MP. Una copia va al procesador, y otra a caché (con su bloque) sustituyéndose un bloque antiguo si no hay sitio.

Caché: concepto

- El éxito de la memoria caché se debe a la propiedad de localidad de los programas.
 - La información que se va a utilizar en un futuro próximo probablemente ya se encuentra en caché.
- Objetivos de diseño en un sistema con caché:
 - ① Maximización del índice de aciertos “A” (probabilidad de encontrar la información deseada en la caché). O minimización de $F=1-A$.
 - ② Minimización del tiempo de acceso “ t_c ” a la información almacenada en la caché.
 - ③ Minimización del retardo debido a una falta.
 - ④ Minimización de la penalización por la actualización de la MP.

Caché: modelo de evaluación

■ Modelo para evaluar las prestaciones de un sistema que utiliza memoria caché.

- t_c : tiempo de acceso a la memoria caché.
- A : razón de acierto (*hit ratio*) de la memoria caché.
- t_m : tiempo de acceso a la memoria principal.
- Tiempo medio de acceso:

$$\bar{T} = At_c + (1 - A)(t_c + t_m)$$

Acierto \Rightarrow no se accede a MP

Fallo \Rightarrow se accede a caché y a MP

- γ : razón entre los tiempos de acceso a la MP y a la caché

$$\gamma = \frac{t_m}{t_c}$$

Caché: modelo de evaluación

- Eficiencia de un sistema que emplea memoria caché:

$$E = \frac{t_c}{\bar{T}}, \quad 0 < E \leq 1$$

$$\begin{aligned} E &= \frac{t_c}{At_c + (1-A)(t_c + t_m)} = \frac{1}{A + (1-A)\left(1 + \frac{t_m}{t_c}\right)} = \\ &= \frac{1}{A + (1-A)(1+\gamma)} = \frac{1}{A+1 - A + \gamma(1-A)} = \frac{1}{1 + \gamma(1-A)} \end{aligned}$$

- E máxima cuando A=1 (todas las referencias en caché).
- $\gamma \uparrow \Rightarrow E \downarrow$; $A \downarrow \Rightarrow E \downarrow$. Interesa γ baja y A alta.
- Ejemplo:

$$\bullet \quad t_c = 15 \text{ ns} \quad \bar{T} = At_c + (1-A)(t_c + t_m) = 0,9 \cdot 15 + 0,1 \cdot 115 = 25 \text{ ns}$$

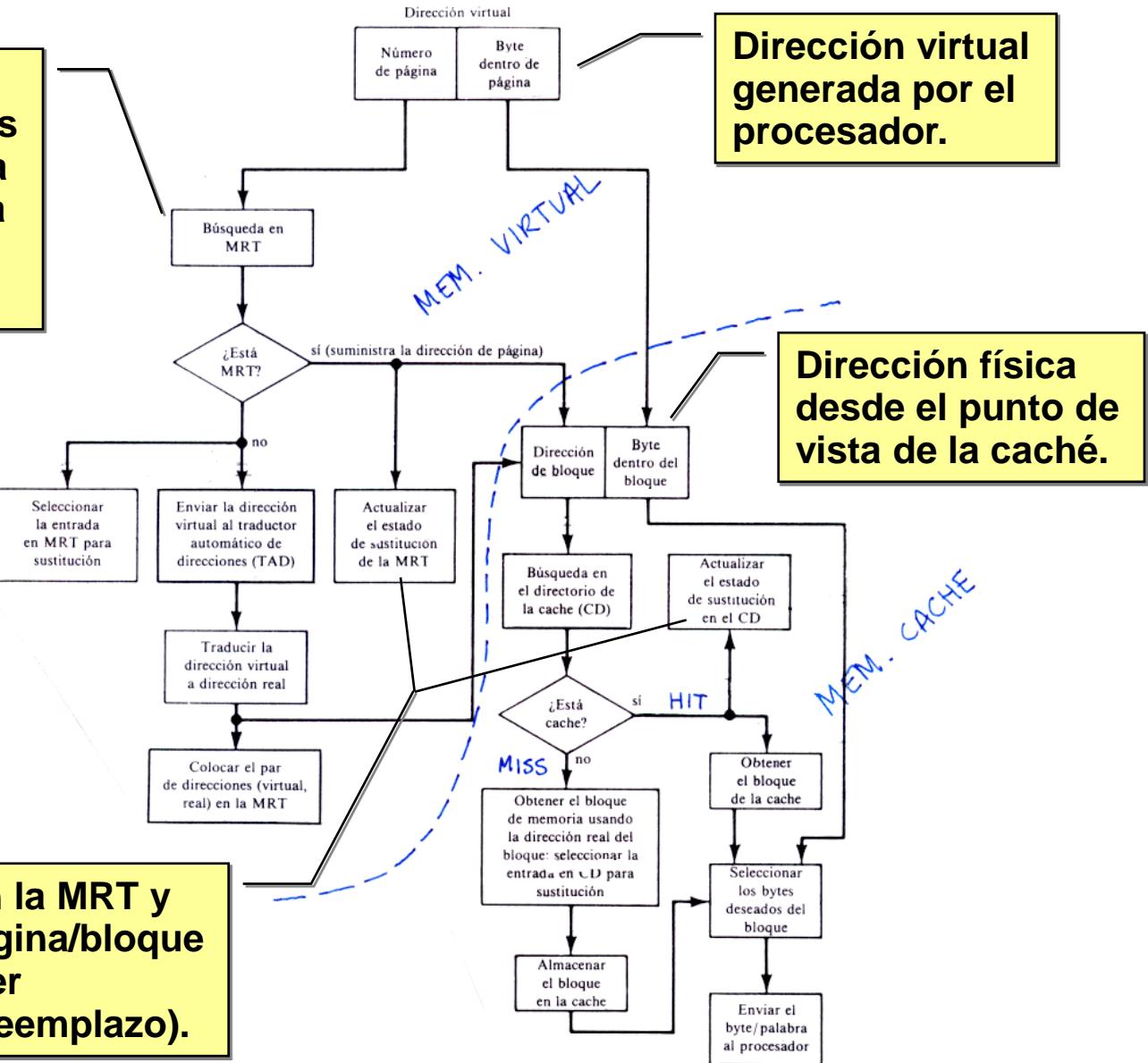
$$\begin{aligned} \bullet \quad t_m &= 100 \text{ ns} & \gamma &= \frac{t_m}{t_c} = \frac{100}{15} = 6,6 & E &= \frac{1}{1 + \gamma(1-A)} = \frac{1}{1 + 6,6 \cdot 0,1} = 0,6 \end{aligned}$$

Caché: estructura y funcionamiento

- Zona de almacenamiento.
 - Memoria **SRAM**.
 - Particionada en un **conjunto de bloques** o líneas.
 - **Bloque o línea**: unidad básica de información que se puede transferir entre la caché y la MP.
 - Cada bloque de MP va a un **marco de bloque** en caché.
 - La MP tiene un gran número de bloques; la caché tiene un pequeño número de marcos de bloque ⇒ la zona de almacenamiento de la caché mantiene copia de un cierto número de bloques, sin ningún orden particular en principio.
- Directorio caché.
 - Se implementa mediante memoria SRAM y comparadores o mediante una memoria asociativa.
 - Guarda las **direcciones de los bloques (etiquetas o marcas)** que hay actualmente en caché, más algunos **bits de control** (para administración y control de acceso a caché).
- Se puede ver la caché como:
 - **Colección de pares dirección-bloque** (dirección del bloque en MP-copia del contenido del bloque).

Caché: estructura y funcionamiento

MRT: Memoria de Reserva de Traducciones (TLB): pequeña memoria asociativa que almacena pares dirección virtual-dirección física.



Estados de sustitución en la MRT y el CD: determinan qué página/bloque respectivamente deben ser sustituidos (políticas de reemplazo).

Caché: políticas y otros parámetros

- Parámetros determinantes en el correcto funcionamiento y efectividad de un sistema de caché:
 - Política de **extracción** o algoritmo de búsqueda de caché.
 - ¿Cuándo y qué información va a ser transferida de la MP a la caché?
 - Política de **colocación** o algoritmo de ubicación de información en la caché.
 - ¿Dónde se coloca en la caché la información transferida desde MP?
 - Política o algoritmo de **reemplazo**.
 - Cuando se produce una falta y la caché está llena, ¿qué bloque se sustituye por uno nuevo?
 - Política o algoritmo de **actualización** de MP.
 - ¿Cuándo debe modificarse el contenido de MP tras escribir en caché?
 - Cachés separadas frente a caché unificada.
 - Tamaños de bloque y de caché.
 - Diseño del sistema de memoria para dar soporte a cachés.

Caché: política de extracción

■ Política de extracción o algoritmo de búsqueda

- Decide cuándo y qué información se va a introducir en caché.
- Objetivo: maximización del índice de aciertos.
- Se distinguen dos criterios de búsqueda básicos:
 - **Por demanda**
 - Búsqueda de la información cuando se necesita.
 - **Prebúsqueda**.
 - Búsqueda de la información antes de ser necesaria.
- Políticas de extracción selectivas.
 - Por demanda o anticipativas si se establecen diferencias entre datos que pueden pasar a la caché y datos que no.

Caché: política de extracción

■ Búsqueda por demanda:

- Es el algoritmo más sencillo.
- Actúa cuando el procesador solicita un bloque y no está en caché (falta).
- Se busca entonces el bloque en MP y transfiere a la caché.
- El procesador debe esperar hasta que la información solicitada le llegue desde la caché.
- Todo sistema de caché admite este tipo de búsqueda.

■ Prebúsqueda o preextracción (políticas anticipativas):

- Algoritmo más complejo y eficiente.
- Busca el bloque que el procesador va a necesitar antes de que lo solicite, reduciendo la probabilidad de falta.
- Localidad espacial \Rightarrow se preextrae usualmente el bloque siguiente al solicitado por el procesador.

...

Caché: política de extracción

- Cuándo realizar la preextracción:
 - **Preextracción siempre:**
 - Se realiza la prebúsqueda del bloque $i+1$ cada vez que el procesador referencia el bloque i . Incrementa el tráfico entre caché y MP.
 - **Preextracción por falta:**
 - Prebuscar el bloque $i+1$ si se referencia a i y se produce falta de bloque.
 - **Preextracción marcada:**
 - Se prebusca el bloque $i+1$ siempre que:
 - » se hace referencia a i produciéndose falta de bloque.
 - » se referencie por primera vez a un bloque i previamente preextraido.
- Problema:
 - se introducen en la caché bloques que posiblemente el procesador nunca referencie. Si la caché es pequeña, estos bloques reemplazan a otros que sí pueden necesitarse.

Caché: política de colocación

■ Políticas de colocación

- La caché se diseña para que sea transparente al usuario y al programador ⇒ necesidad de una función que convierta una dirección de memoria principal en una posición de caché.
- La caché y la MP están divididas en unidades de igual tamaño:
 - Bloques o líneas en MP.
 - Marcos de bloque o líneas en la caché.

La política de colocación determina la función de correspondencia entre bloques de la MP y marcos de bloque de la caché.

- Puesto que la MP es mucho mayor que la caché, cada marco de bloque almacenará a lo largo del tiempo muchos bloques de la MP.

Caché: política de colocación

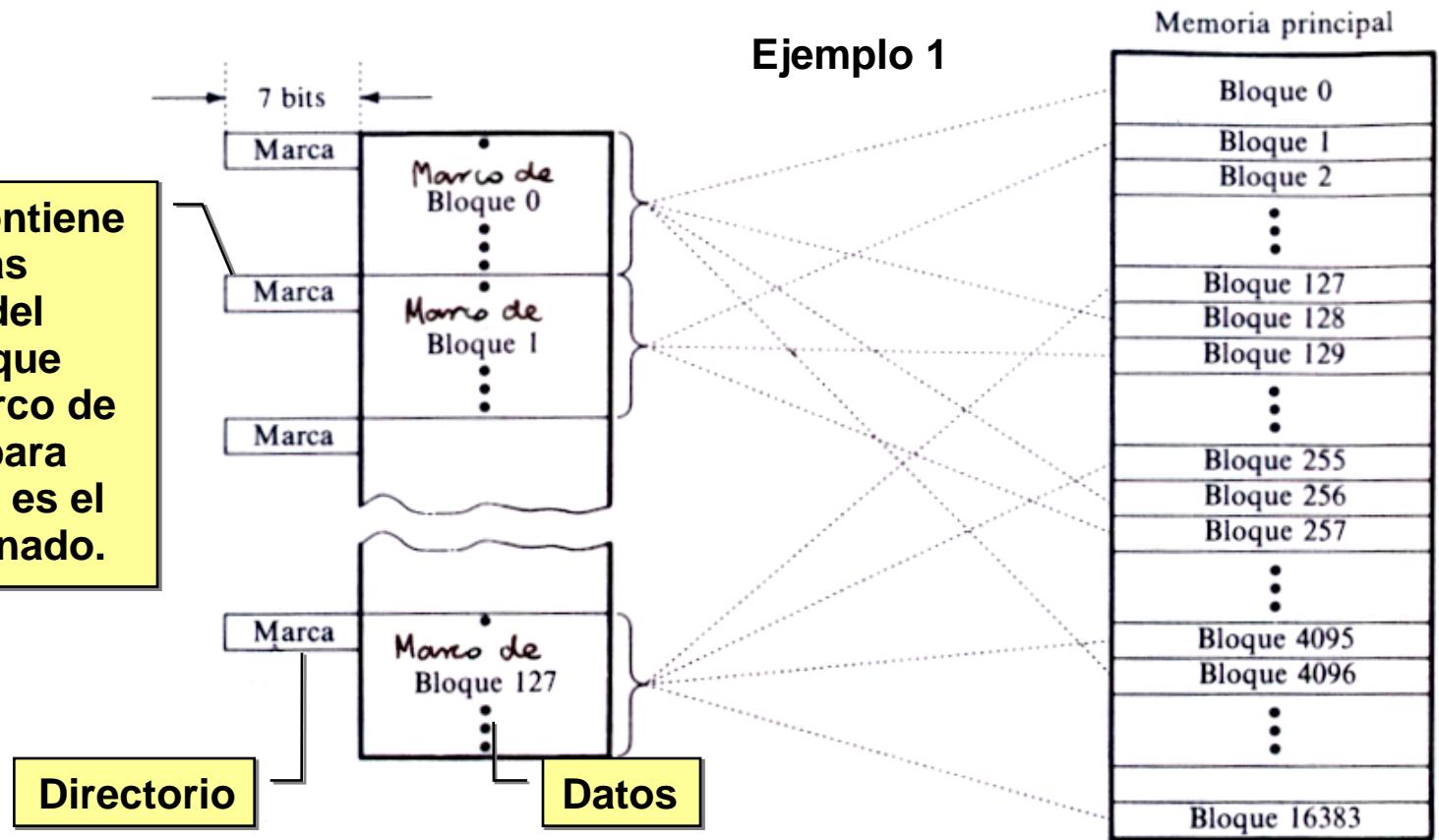
- Existen cuatro políticas de colocación:
 - Correspondencia directa.
 - Totalmente asociativa.
 - Asociativa por conjuntos.
 - Correspondencia por sectores.
 - Consideraremos dos ejemplos:
 - Ejemplo 1:
 - Tamaño de caché: 2K palabras.
 - 16 palabras por bloque.
 - Memoria principal: 256K palabras.
 - Dirección física: 18 bits.
 - Ejemplo 2:
 - Tamaño de caché: 2^{2+w} palabras.
 - 2^w palabras por bloque.
 - Memoria principal: 2^{4+w} palabras.
 - Dirección física: $4+w$ bits.
 - La MP tiene 2^N bloques, la caché tiene 2^n marcos de bloque y un bloque tiene 2^w palabras. Una dirección física tendrá $N+w$ bits.
 - Ejemplo 1: $N=14$, $n=7$, $w=4$
 - Ejemplo 2: $N=4$, $n=2$
- $\left. \begin{matrix} \\ \\ \end{matrix} \right\} \Rightarrow 128 \text{ marcos de bloque}$ $\left. \begin{matrix} \\ \\ \end{matrix} \right\} \Rightarrow 16K \text{ bloques en MP}$
 $\left. \begin{matrix} \\ \\ \end{matrix} \right\} \Rightarrow 4 \text{ marcos de bloque}$ $\left. \begin{matrix} \\ \\ \end{matrix} \right\} \Rightarrow 16 \text{ bloques en MP}$

Caché: política de colocación

■ Correspondencia directa

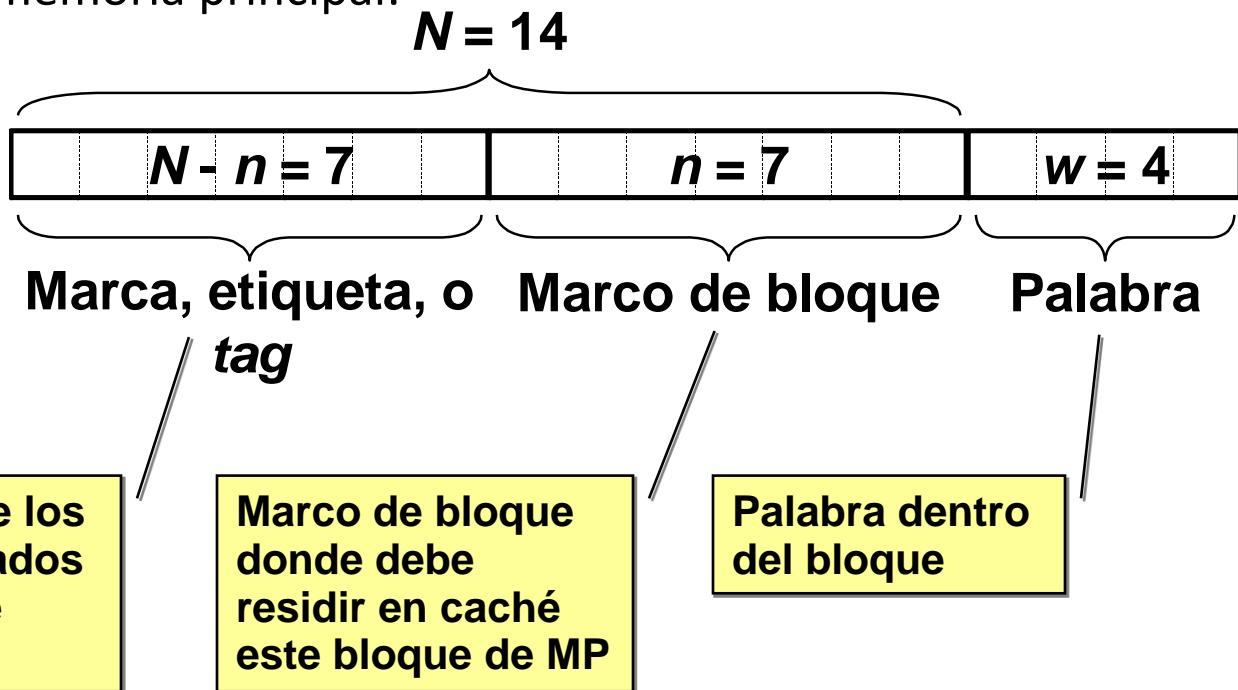
- Bloque i de MP \Rightarrow marco de bloque $i \bmod 2^n$ de caché.
- A cada marco de bloque le corresponde sólo un subconjunto de bloques de MP.

Cada marca contiene los $N-n$ bits más significativos del bloque de MP que hay en ese marco de bloque. Sirve para identificar cuál es el bloque almacenado.



Caché: política de colocación

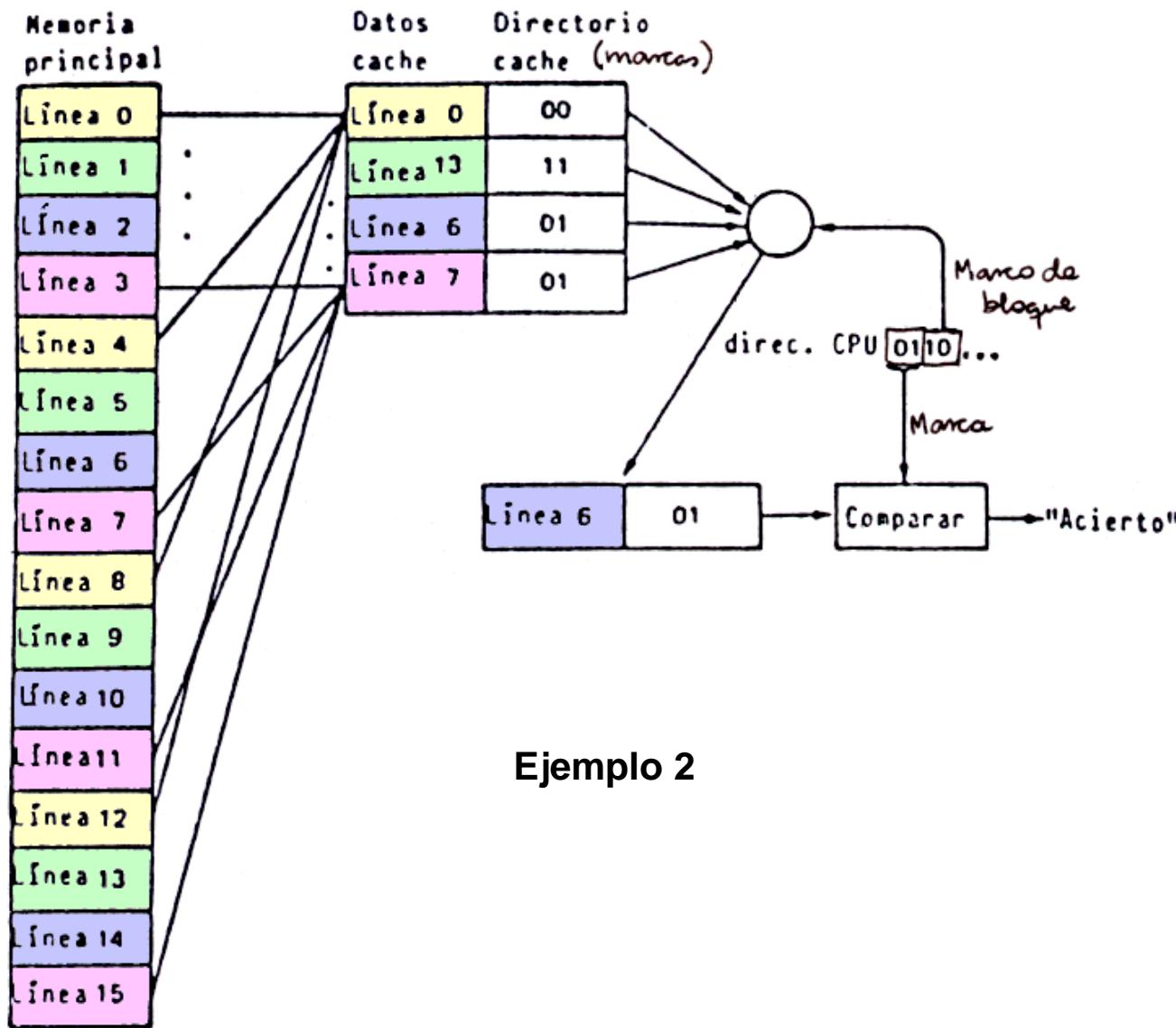
- Dirección de memoria principal:



✓ Simplicidad y bajo coste.

✗ Si dos o más bloques, utilizados alternativamente, corresponden al mismo marco de bloque \Rightarrow el índice de aciertos se reduce drásticamente.

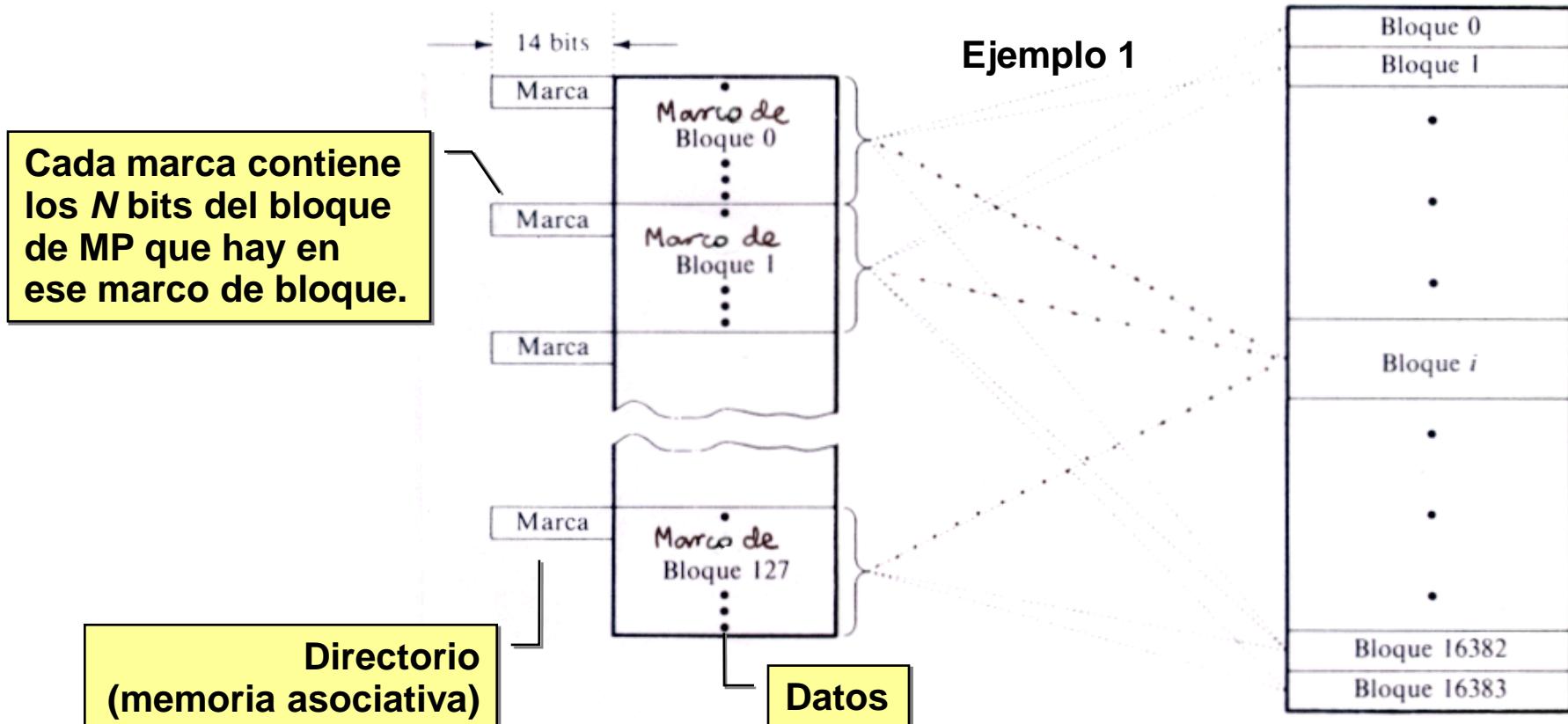
Caché: política de colocación



Caché: política de colocación

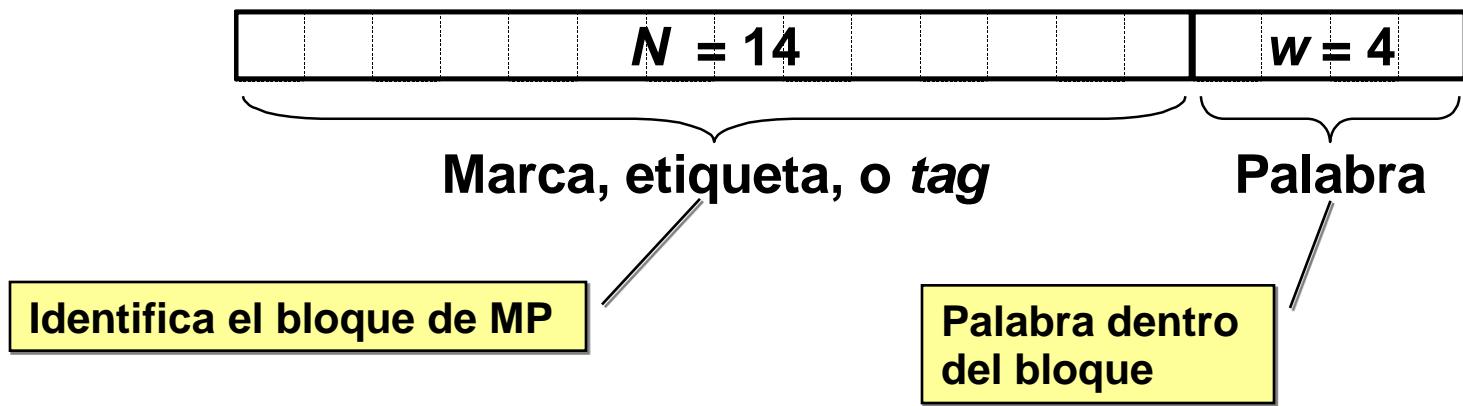
■ Correspondencia totalmente asociativa

- Un bloque de MP puede residir en cualquier marco de bloque de caché.
- Cuando se presenta una solicitud a la caché, todas las marcas se comparan simultáneamente para ver si el bloque está en la caché.



Caché: política de colocación

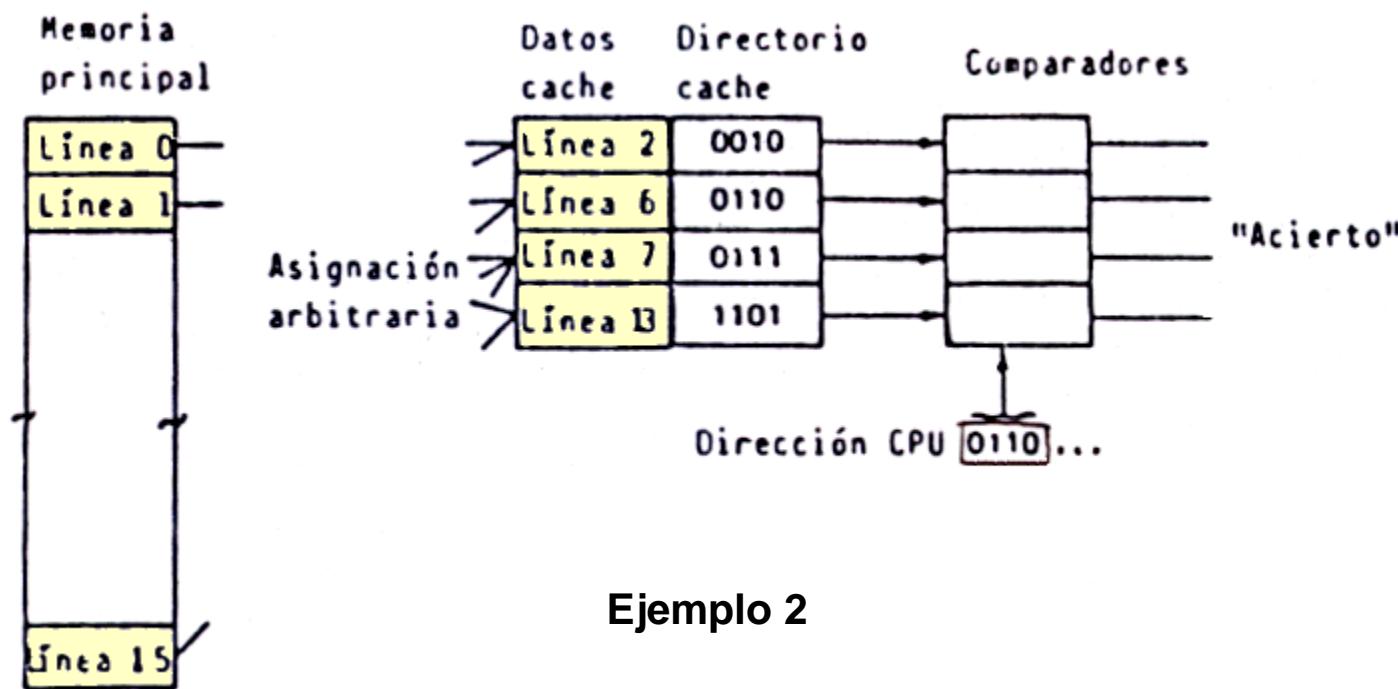
- Dirección de memoria principal:



✓ **Flexible.** Permite cualquier combinación de bloques de MP en la caché. Elimina en gran medida conflictos entre bloques.

✗ **Compleja y costosa de implementar (por la memoria asociativa).**

Caché: política de colocación



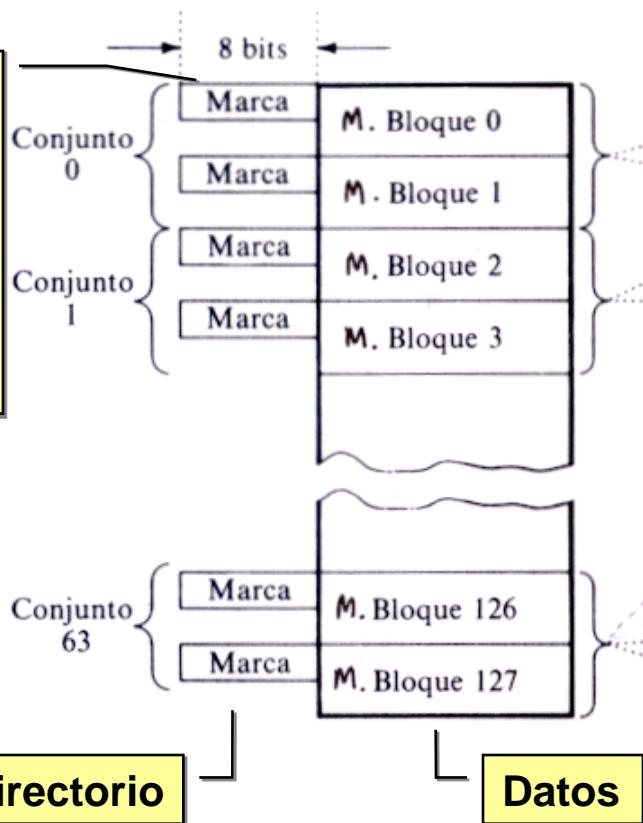
Caché: política de colocación

■ Correspondencia asociativa por conjuntos

- La caché se subdivide en 2^c conjuntos disjuntos
 - 2^{n-c} marcos de bloque / conjunto
- Bloque i de MP \Rightarrow conjunto $i \bmod 2^c$ de caché. Dentro de ese conjunto puede estar en cualquier marco de bloque.
- Hay dos fases en el acceso a caché:
 - Selección directa del conjunto donde puede estar ese bloque.
 - Búsqueda asociativa (dentro del conjunto) de la marca.
- A la correspondencia asociativa por conjuntos con 2^{n-c} marcos de bloque / conjunto también se le llama correspondencia asociativa de 2^{n-c} vías.
 - La vía i está formada por todos los marcos de bloque de la caché que ocupan el lugar i -ésimo dentro de su conjunto.
 - Tanto el ejemplo 1 como el 2 corresponden a caches asociativas de 2 vías.

Caché: política de colocación

Cada marca contiene los N_c bits más significativos del bloque de MP que hay en ese marco de bloque.

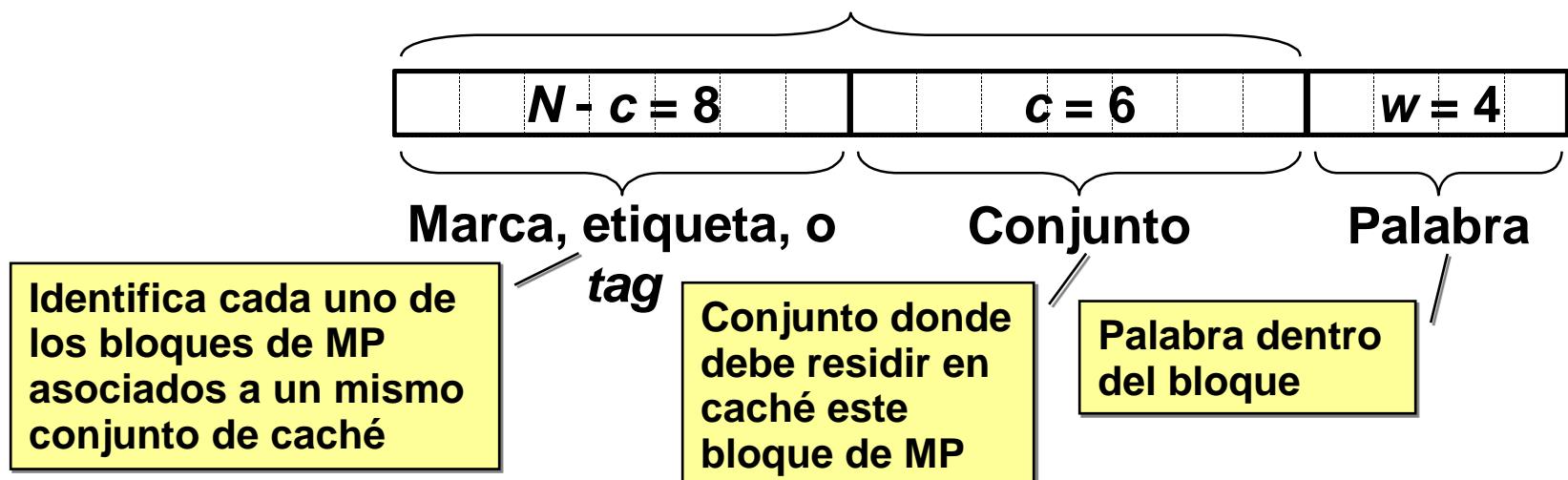


Ejemplo 1

Bloque 0
Bloque 1
⋮
Bloque 63
Bloque 64
Bloque 65
⋮
Bloque 4095
Bloque 16383

Caché: política de colocación

- Dirección de memoria principal: $N = 14$



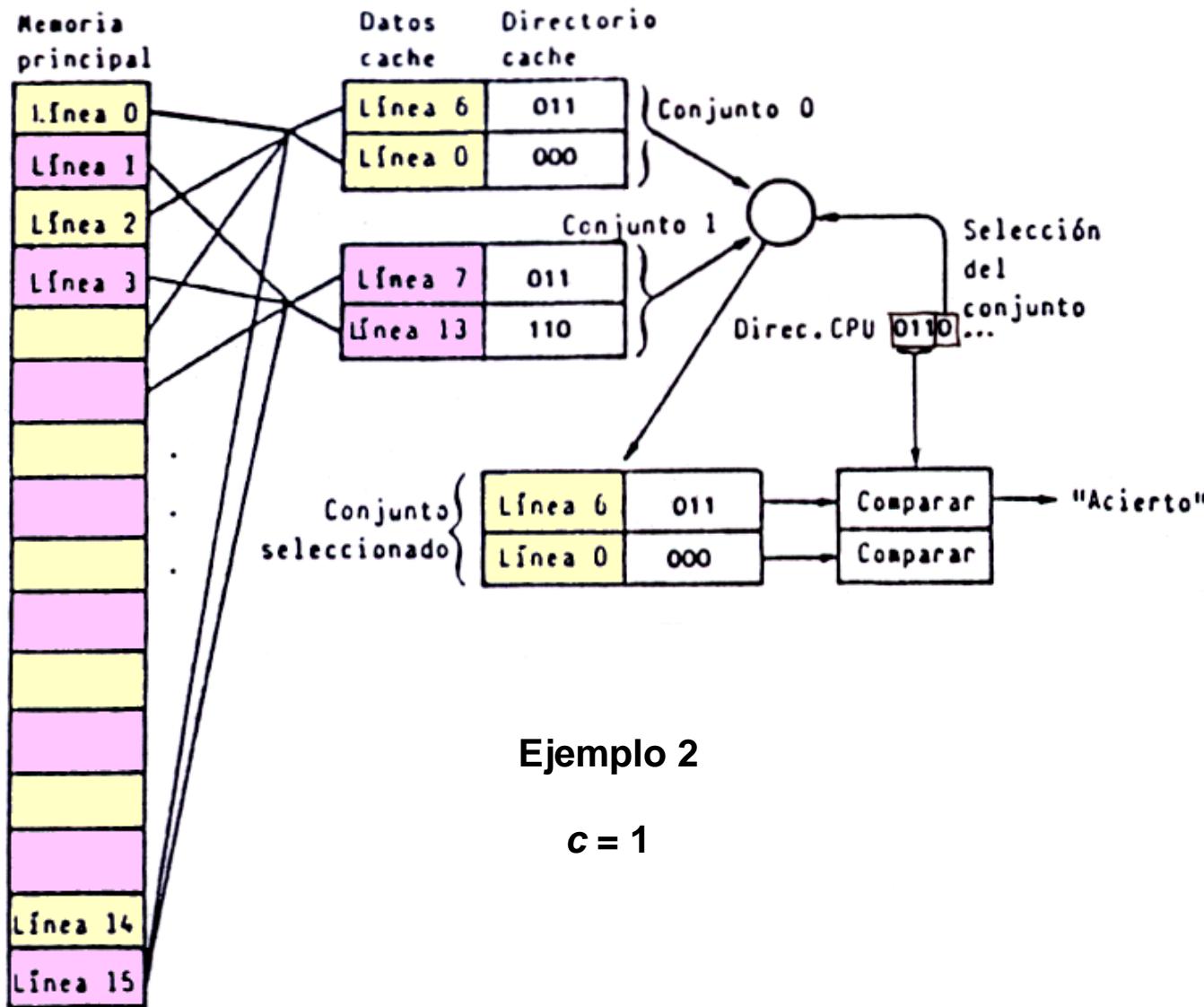
$c = 0$ (1 conjunto) \Rightarrow corresp. totalmente asociativa.

$c = n$ (1 marco de bloque / conjunto) \Rightarrow corresp. directa.

$0 < c < n \Rightarrow$ se pretende reducir el **coste de la totalmente asociativa** manteniendo un **rendimiento similar** \Rightarrow es la técnica más utilizada.

✓ Resultados experimentales demuestran que un tamaño de conjunto de 2 a 16 marcos de bloque funciona casi tan bien como una corresp. totalmente asociativa con un incremento de coste pequeño respecto de la corresp. directa.

Caché: política de colocación



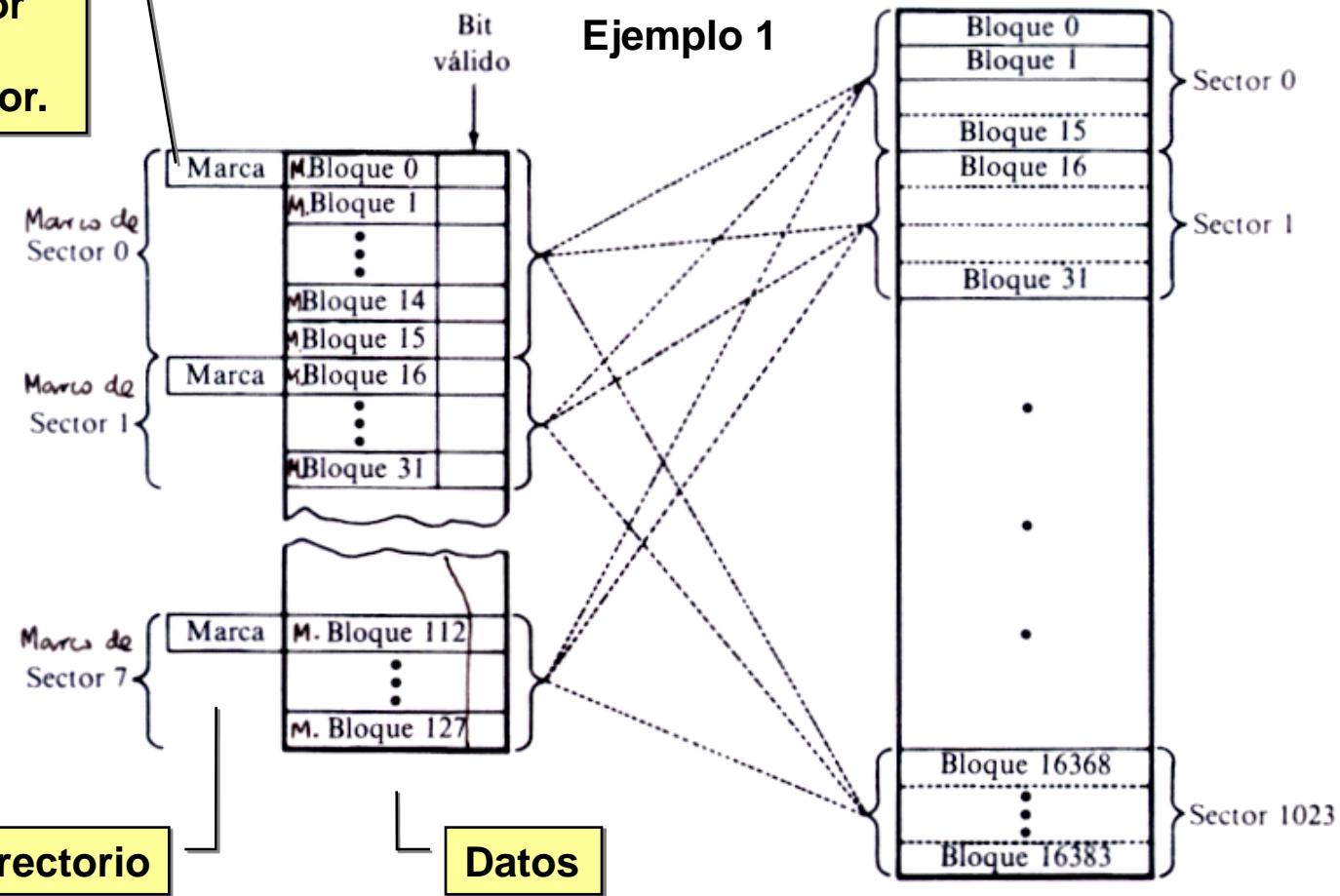
Caché: política de colocación

■ Correspondencia por sectores

- La MP se divide en 2^s grupos disjuntos llamados sectores $\Rightarrow 2^{N-s}$ bloques / sector.
- La caché se compone de $2^{n-(N-s)}$ marcos de sectores, cada uno con 2^{N-s} marcos de bloque.
- Un sector de MP puede estar en cualquier marco de sector, pero la correspondencia de bloques dentro de un sector es directa.
- Se lleva a caché el bloque que provocó la falta. Los restantes marcos de bloque dentro de ese sector se marcan como *no-válidos* al cargar un nuevo sector.
- Hay dos fases en el acceso a caché:
 - ① Búsqueda totalmente asociativa de la marca o sector.
 - ② Selección directa del marco de bloque usando los $N-s$ bits intermedios.

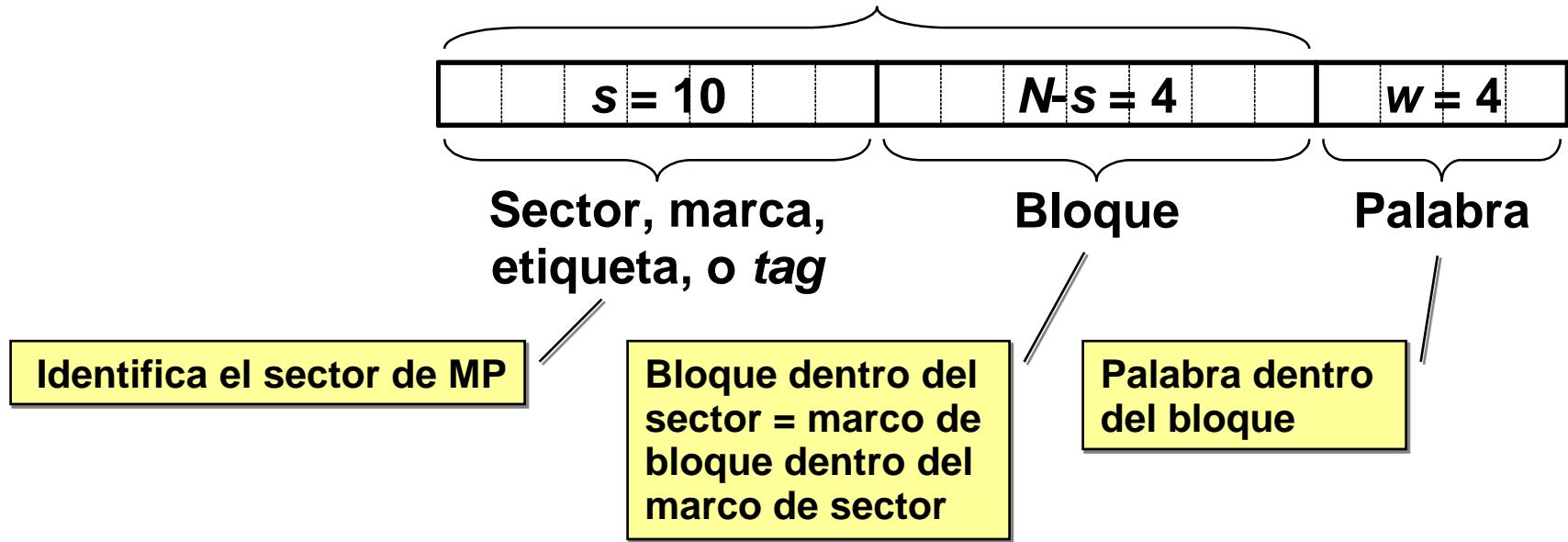
Caché: política de colocación

Cada marca contiene los s bits del sector de MP que hay en ese marco de sector.



Caché: política de colocación

- Dirección de memoria principal: $N = 14$



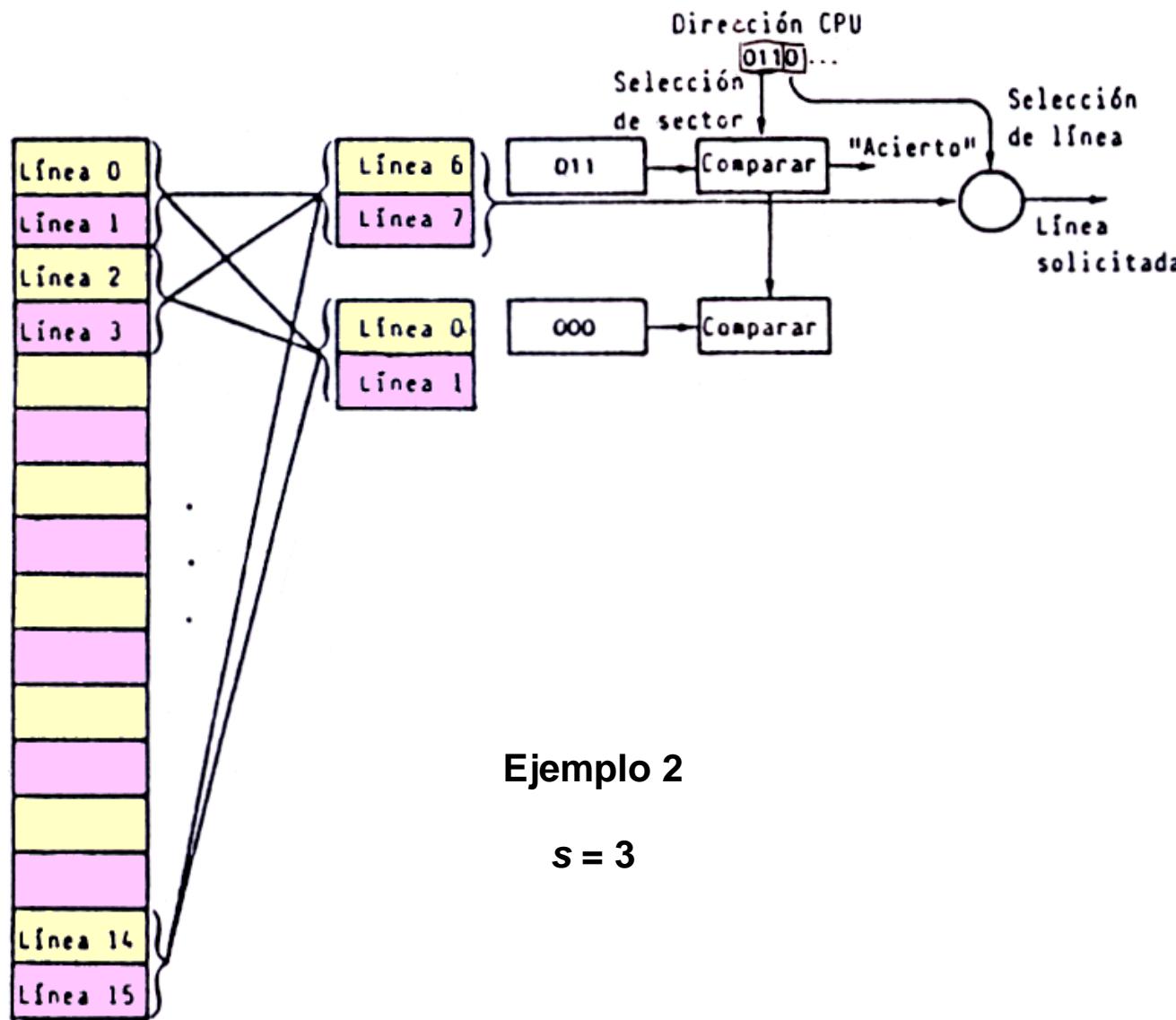
$s = 0$ (1 sector) \Rightarrow no tiene sentido (caché de igual tamaño que MP).

$s = N$ (1 bloque / sector) \Rightarrow corresp. totalmente asociativa.

✓ Pocas marcas \Rightarrow memoria asociativa pequeña.

✗ N° de combinaciones diferentes de los bloques de MP en caché es inferior al del esquema asociativo por conjuntos.

Caché: política de colocación



Caché: política de reemplazo

■ Políticas de reemplazo de bloques

- Si se produce una falta en lectura \Rightarrow hay que traer un nuevo bloque \Rightarrow debe decidirse, si la caché está llena, qué bloque sustituir por el nuevo.
- Este problema no existe para correspondencia directa.
- Para las otras tres organizaciones:

- **ALGORITMOS DE REEMPLAZO:** ¿*en qué posición de caché se copia el bloque de MP al que se accede?*

- Se utilizan los mismos algoritmos que para memoria virtual.
 - **FIFO** (*First-In First-Out*)

» De todos los bloques existentes en caché, se reemplaza el primero que se introdujo.

- **LRU** (*Least Recently Used*)
 - » Se reemplaza el menos recientemente referenciado.
 - **RAND**
 - El bloque a reemplazar se elige aleatoriamente.

El más usado

Caché: política de actualización

■ Políticas de actualización de la memoria principal

- Cuando se modifica el contenido de la caché debe enviarse una copia de los nuevos datos a MP.
- ¿Cuándo debe realizarse esa actualización?
- Escritura directa, escribir siempre o *write-through*.
 - Se actualiza la MP cada vez que se modifica el contenido de la caché.
 - Escritura directa con asignación en escritura (EDAE):
 - Se cargan bloques en caché tanto si hay faltas por lectura como por escritura.
 - Escritura directa sin asignación en escritura (EDSAE):
 - Se cargan bloques en caché si hay faltas por lectura pero no en las faltas por escritura.
 - En general, $T(\text{EDAE}) > T(\text{EDSAE})$, ya que en la segunda se transfieren menos bloques a caché.

Caché: política de actualización

- **Post-escritura**, escribir bajo falta, o *write-back*.
 - Se actualiza la MP después, normalmente cuando ese bloque debe ser reemplazado.
 - **Post-escritura siempre** (PES):
 - Los bloques que abandonan la caché se escriben en MP.
 - Excesivo trasiego de información.
 - **Post-escritura marcada** (PEM):
 - Se añade un bit que indica si el bloque se ha modificado.
 - Se escribe en MP sólo los bloques modificados.
 - En general, $T(PES) > T(PEM)$, ya que en la segunda se transfieren menos bloques a MP.

Caché separada / unificada

■ Caches separadas de datos / instrucciones

- Es común particionar la caché en dos módulos diferentes:
 - **Caché de datos.**
 - La localidad de los datos no es tan buena como la de las instrucciones ⇒ la caché de datos es **menos eficiente**.
 - Es **más compleja** por la posibilidad de **modificación** de los datos.
⇒ Muchos sistemas no admiten caché de datos.
 - **Caché de instrucciones.**
 - Es fácil que bucles y pequeñas rutinas entren totalmente en caché, permitiendo su ejecución sin necesidad de acceder a MP.
 - Se simplifica la caché, ya que es habitual que se prohíba escribir en las localizaciones donde se encuentran las instrucciones ⇒ caché de **sólo lectura**.
- ✓ Se pueden emitir direcciones de instrucción y dato a la vez, doblando el ancho de banda entre caché y procesador.
- ✓ Se puede optimizar cada caché por separado:
 - Diferentes capacidades, tamaños de bloque, asociatividades, etc.

Caché separada / unificada

Las caches de instrucciones tienen menor frecuencia de fallos que las de datos.

¿Cuál tiene una frecuencia de fallos menos: una caché de instrucciones de 16 KB + una caché de datos de 16 KB o una caché unificada de 32 KB?

Frecuencia de fallos para la caché particionada:

$$53\% \cdot 3,6\% + 47\% \cdot 5,3\% = 4,4\%$$

Una caché unificada de 32 KB tiene una frecuencia de fallos similar (4,3%).

(Ver, sin embargo, las ventajas de la transparencia anterior).

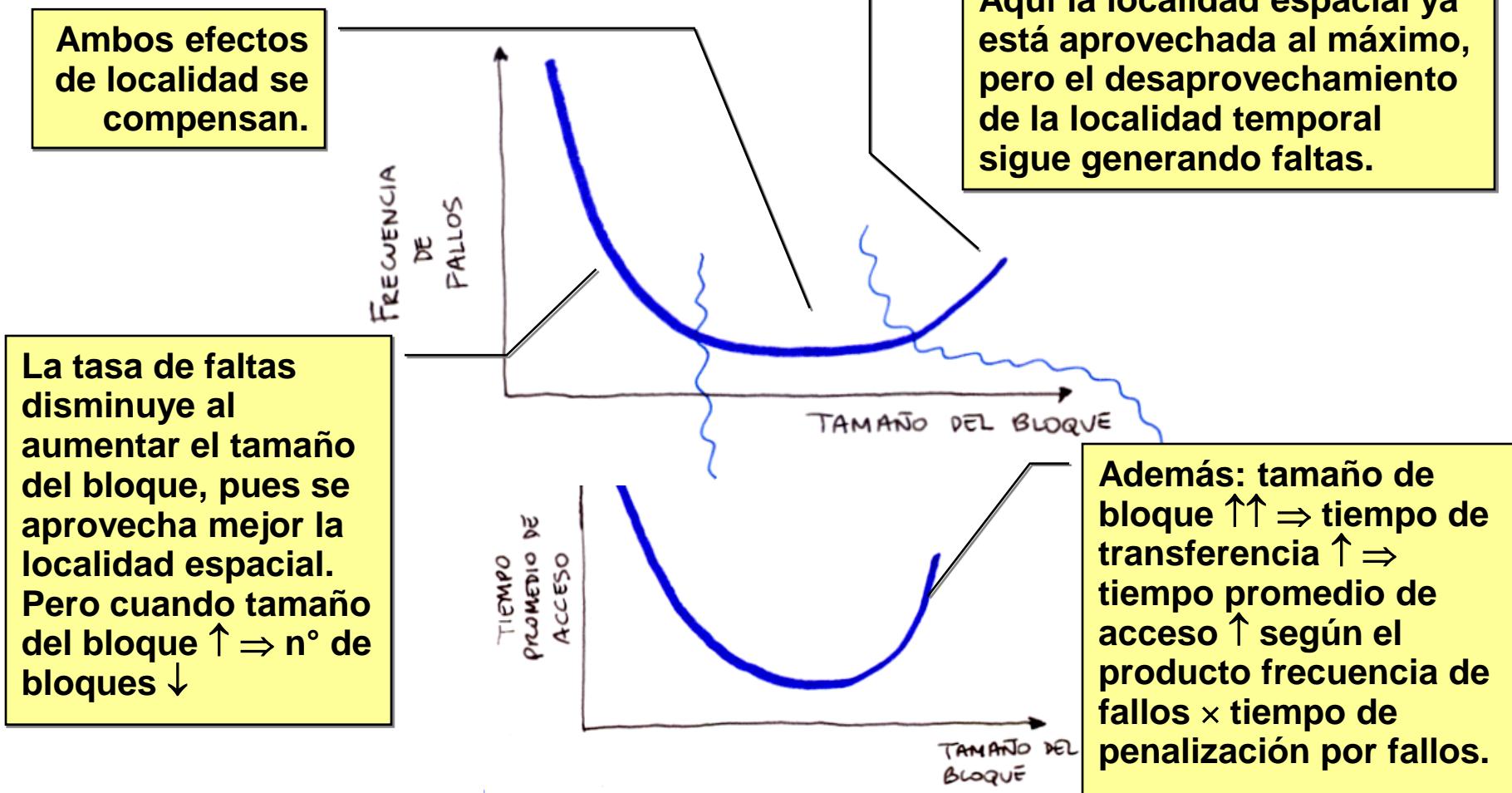
Tamaño	Instrucción sólo	Sólo datos	Unificada
0,25 KB	22,2 %	26,8 %	28,6 %
0,50 KB	17,9 %	20,9 %	23,9 %
1 KB	14,3 %	16,0 %	19,0 %
2 KB	11,6 %	11,8 %	14,9 %
4 KB	8,6 %	8,7 %	11,2 %
8 KB	5,8 %	6,8 %	8,3 %
16 KB	3,6 %	5,3 %	5,9 %
32 KB	2,2 %	4,0 %	4,3 %
64 KB	1,4 %	2,8 %	2,9 %
128 KB	1,0 %	2,1 %	1,9 %
256 KB	0,9 %	1,9 %	1,6 %

Frecuencia de fallos para caches de distintos tamaños de sólo datos, sólo instrucciones, y unificadas. Los datos son para una cache asociativa de 2 vías utilizando reemplazo LRU con bloques de 16 bytes para un promedio de trazas de usuario/sistema en la VAX-11 y trazas de sistema en el IBM 370 [Hill 1987]. El porcentaje de referencias a instrucciones en estas trazas es aproximadamente del 53 por 100.

Caché: tamaño

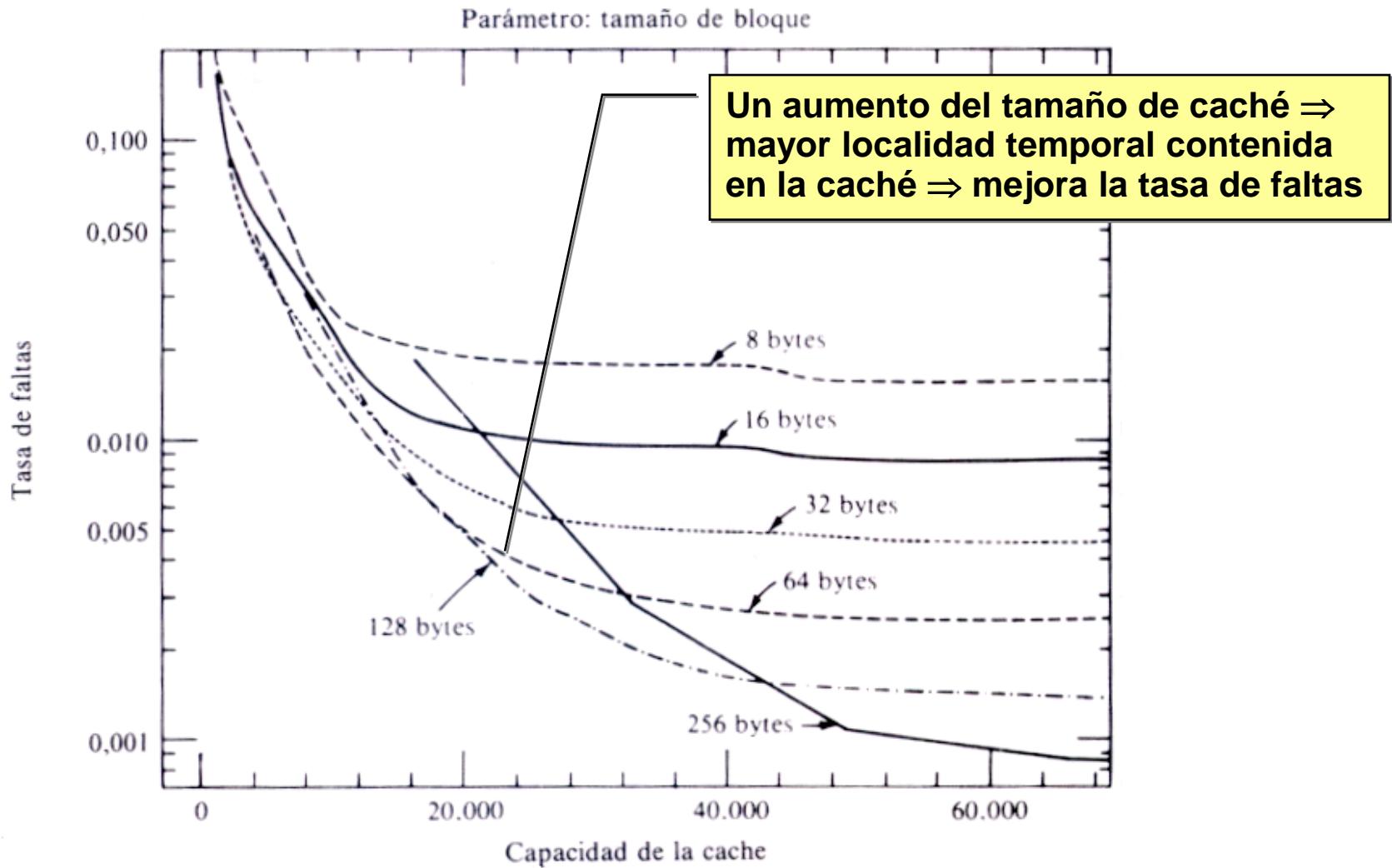
■ Tamaños del bloque y de la caché

- Para un tamaño de caché fijo:



Caché: tamaño

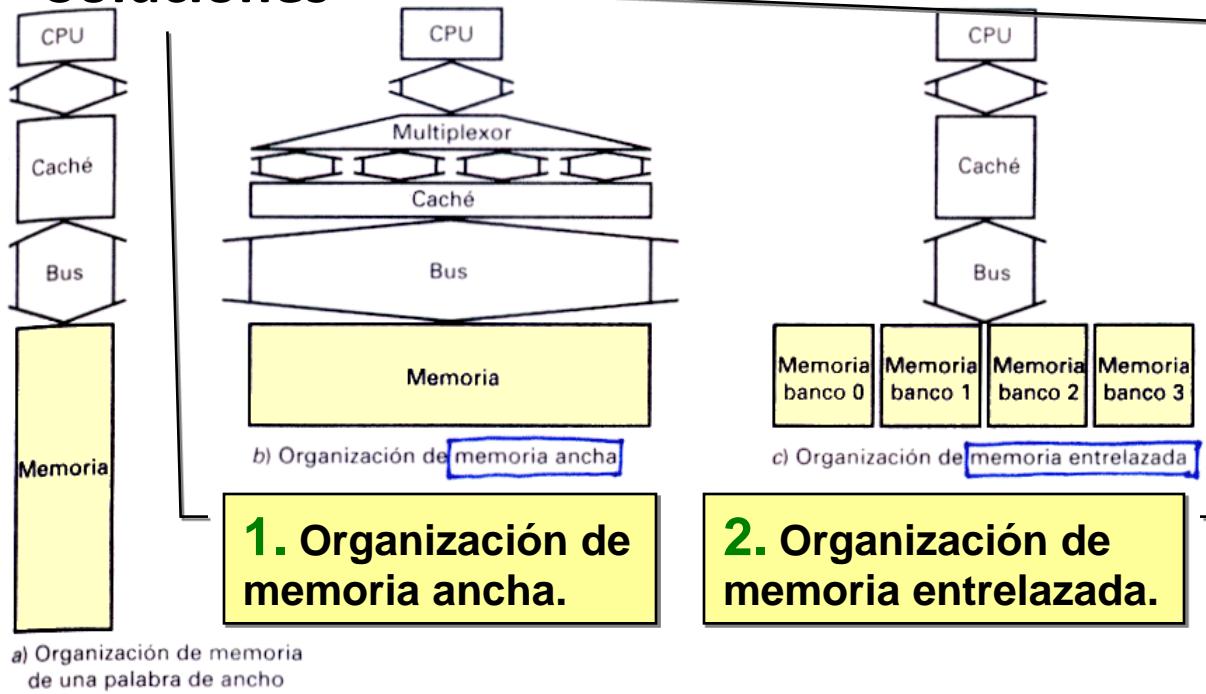
- Para un tamaño de bloque fijo:



Diseño para dar soporte a caché

Aunque sea difícil reducir el tiempo para buscar la primera palabra de memoria en el caso de un fallo de caché, se puede reducir la penalización de fallos incrementando el ancho de banda entre MP y caché.

Soluciones



3. Acceso a la memoria en modo página rápido, o utilizando algunos de los tipos de memoria DRAM más recientes (SDRAM, RDRAM).

1. Organización de memoria ancha.

2. Organización de memoria entrelazada.

a) Organización de memoria de una palabra de ancho

El método principal de conseguir mayor anchura de banda de memoria es incrementar la anchura física o lógica del sistema de memoria. En esta figura hay dos formas en las que se mejora la anchura de banda de memoria. El diseño más simple, a), utiliza una memoria donde todos los componentes son de una palabra; b) muestra memoria, bus y caché de más anchura; mientras c) muestra un bus estrecho y una caché con una memoria entrelazados.

Diseño para dar soporte a caché

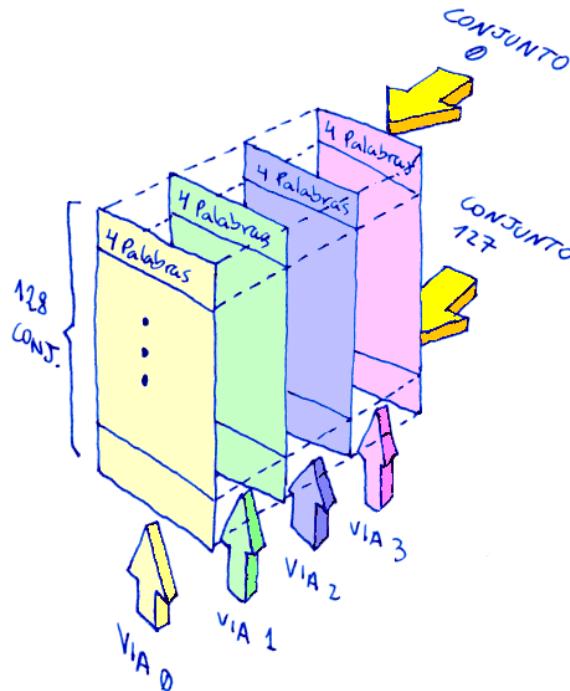
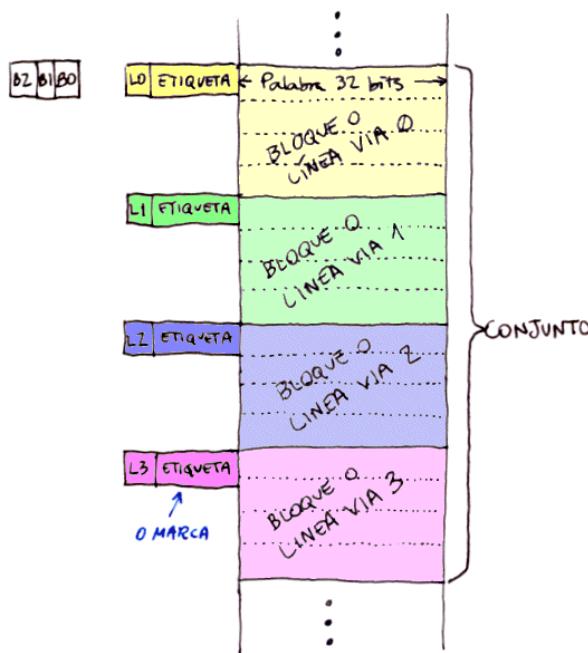
■ Ejemplo (transferencia de 4 palabras de MP a caché):

- 1 ciclo de reloj para enviar la dirección.
- 10 ciclos para cada el acceso a la DRAM.
- 1 ciclo para enviar una palabra de datos.
- Organización de una palabra de ancho:
 - $1 + 4 * 10 + 4 * 1 = 45$ ciclos
- Organización de memoria ancha (2 palabras):
 - $1 + 2 * 10 + 2 * 1 = 23$ ciclos
- Organización de memoria ancha (4 palabras):
 - $1 + 1 * 10 + 1 * 1 = 12$ ciclos
- Organización de memoria entrelazada (4 módulos):
 - $1 + 1 * 10 + 4 * 1 = 15$ ciclos

Ejemplo: caché interna del 486

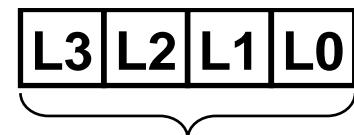
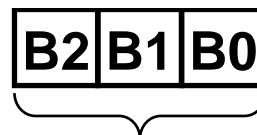
■ Ejemplo: Unidad de caché interna del i486

- Caché unificada (datos e instrucciones) de 8 KB.
- Correspondencia asociativa por conjuntos, 128 conjuntos
- Cada conjunto tiene 4 bloques o líneas \Rightarrow asociativa de 4 vías.
- Cada bloque tiene 4 palabras de 32 bits (16 bytes).



Ejemplo: caché interna del 486

- Cada marco de bloque tiene una **etiqueta o marca de 21 bits**, que se compara con los 21 bits de mayor peso ($A_{11}-A_{31}$) de la dirección física referenciada para ver si está en caché.
- Cada conjunto: **7 bits de estado**:



- **Algoritmo de extracción:**

- **Por demanda selectiva.**

- El llenado de una línea de caché se lleva a cabo siempre que se produzca una falta en lectura. Si la falta es en escritura, la caché no intentará cargar el bloque.

Utilizados por el algoritmo de reemplazo LRU
Indican qué líneas son válidas (contienen datos correctos)

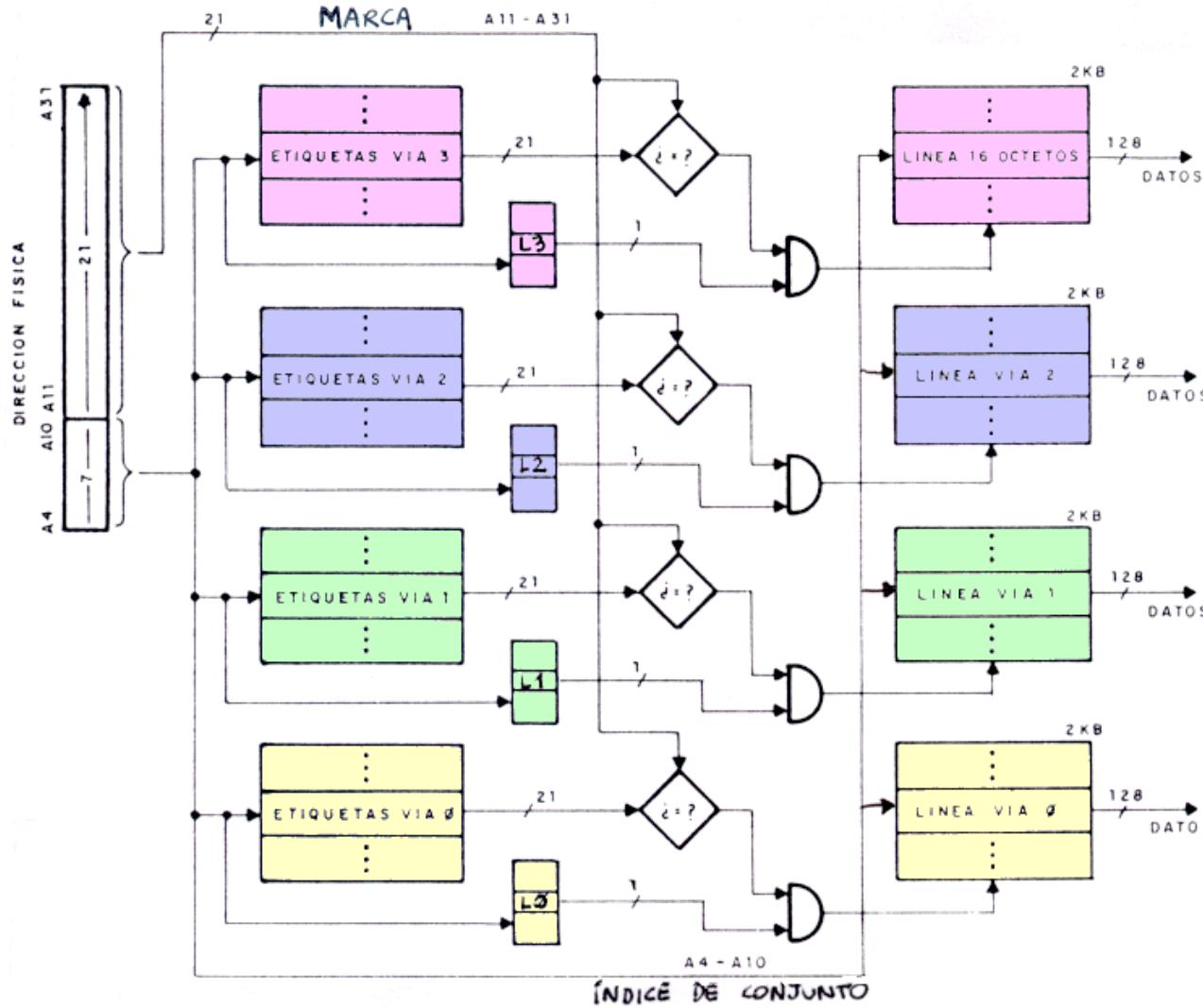
- **Actualización de la memoria principal:**

- **Escritura directa sin asignación en escritura.**

- La escritura sobre un dato contenido en la caché es inmediatamente dirigida hacia MP pasando antes por un registro intermedio de escritura (el i486 no tiene que esperar a que finalice el ciclo de escritura).

Ejemplo: caché interno del 486

■ Organización física



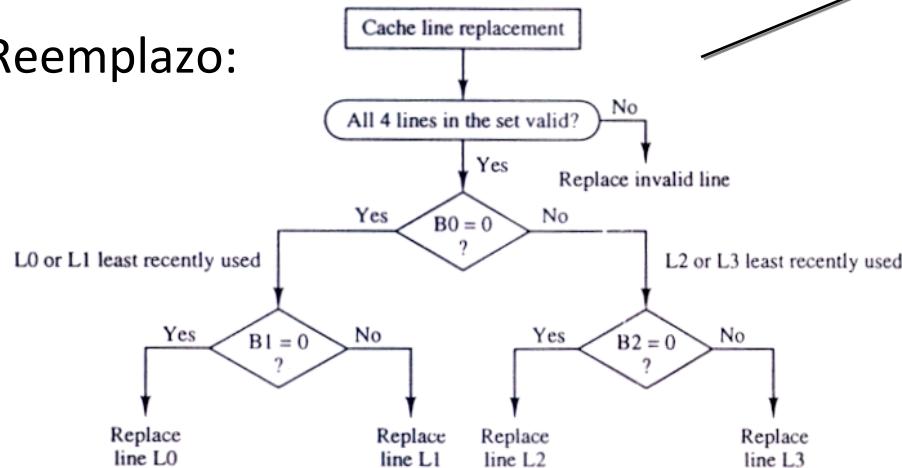
Ejemplo: caché interna del 486

- Algoritmo de reemplazo:

- **Pseudo-LRU** (no es LRU; ej.: L2 L1 L0 L3):
 - Modificación de los bits B0, B1 y B2:

```
if (acceso a L0 o a L1)
{
    B0 = 1;
    if (acceso a L0) B1 = 1;
    else B1 = 0;
}
else
{
    B0 = 0;
    if (acceso a L2) B2 = 1;
    else B2 = 0;
}
```

- Reemplazo:



Acceso a	B0	B1	B2
L0	1	1	
L1	1	0	
L2	0		1
L3	0		0

Sólo se utiliza el algoritmo de reemplazo si todas las líneas del conjunto son válidas. Si no, los nuevos datos se almacenan en alguna de las no válidas.

Segmentación de cauce

Estructura de Computadores
11^a Semana

Bibliografía:

- | | |
|-----------------|---|
| [HAM03] Cap.8 | Organización de Computadores. Hamacher, Vranesic, Zaki. McGraw-Hill 2003
Signatura ESIIT/ C.1 HAM org |
| [STA8] Sec.12.4 | Organización y Arquitectura de Computadores, 7 ^a Ed. Stallings. Pearson Educación, 2008.
Signatura ESIIT/ C.1 STA org |

Guía de trabajo autónomo (4h/s)

■ Lectura

- Cap.8 Hamacher
- Sección 12.4 Stallings

Bibliografía:

[HAM03] Cap.8

Organización de Computadores. Hamacher, Vranesic, Zaki. McGraw-Hill 2003

Signatura ESIIT/[C.1 HAM org](#)

[STA8] Sec.12.4

Organización y Arquitectura de Computadores, 7^a Ed. Stallings. Pearson Educación, 2008.

Signatura ESIIT/[C.1 STA org](#)

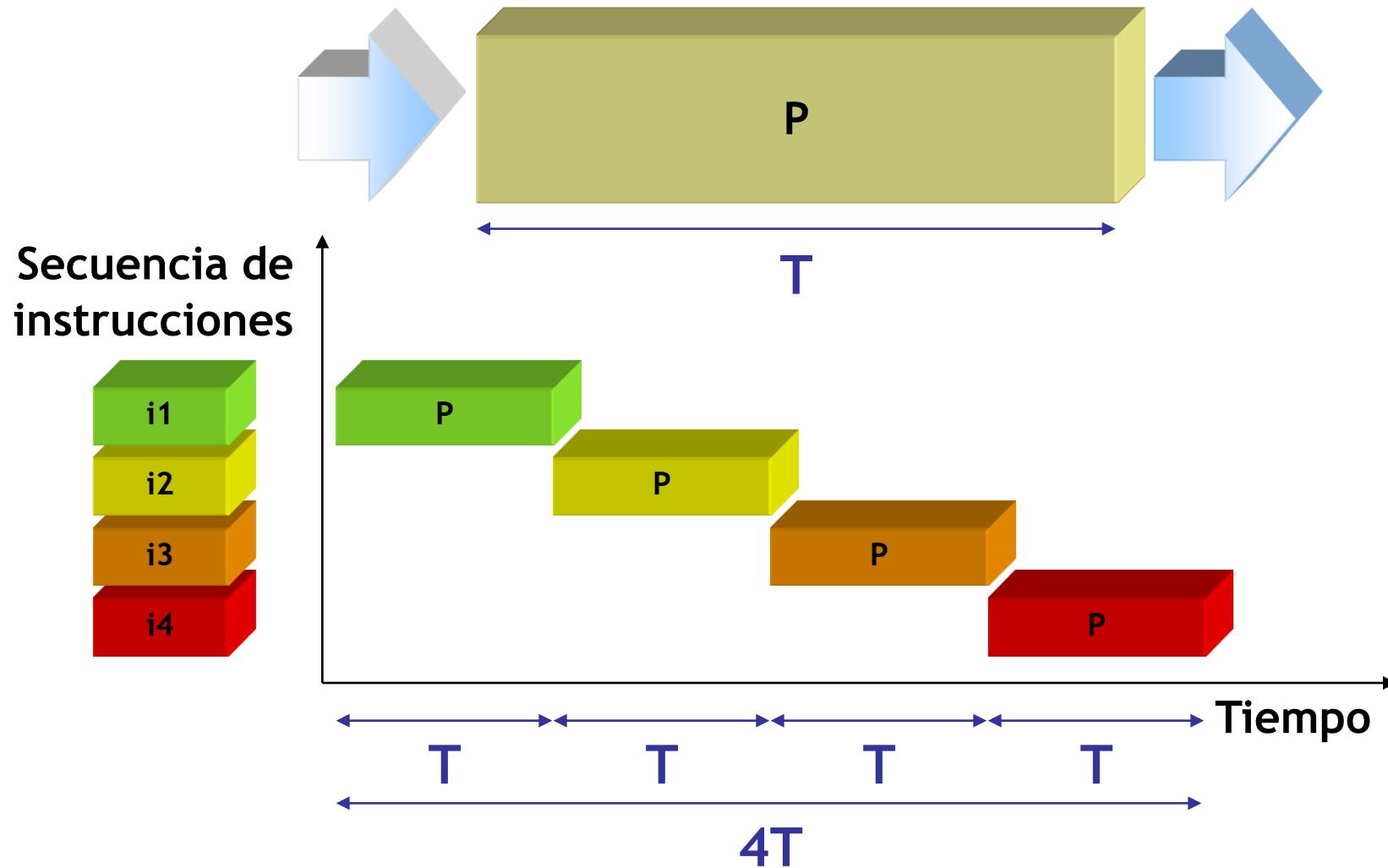
[

Segmentación de cauce

- Concepto de segmentación
- Ejemplo de segmentación
- Aceleración
- Riesgos
- Influencia en el repertorio de instrucciones
- Funcionamiento superescalar

Concepto de segmentación

Procesador sin segmentar



Concepto de segmentación

¿Es posible incrementar la velocidad de procesamiento, al margen de mejoras tecnológicas, sin incrementar el número de procesadores?

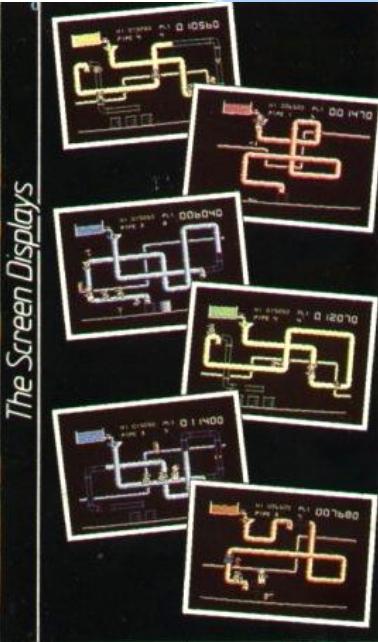
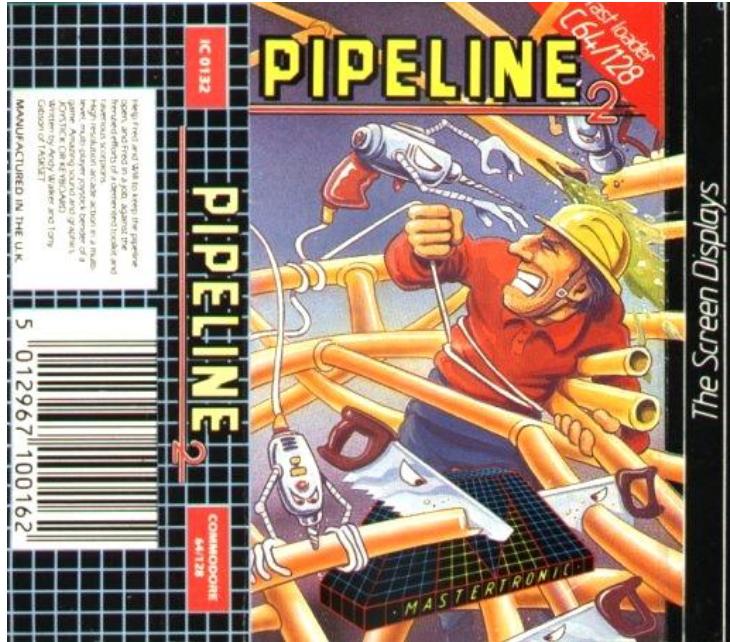


SOLUCIÓN



Segmentación de cauce
(*pipelining*)

Concepto de segmentación

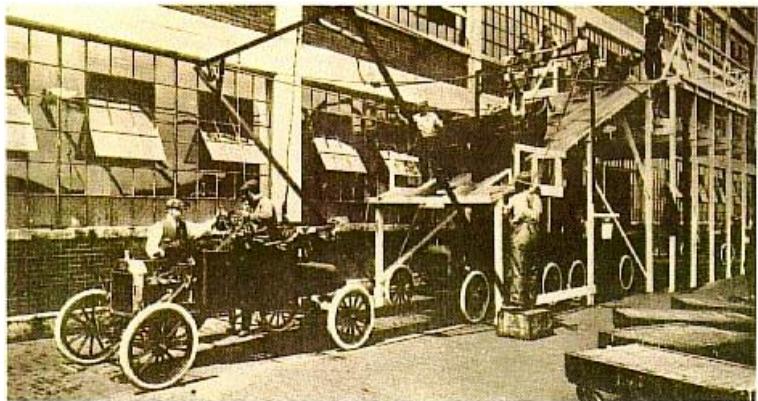


THE PIPELINE

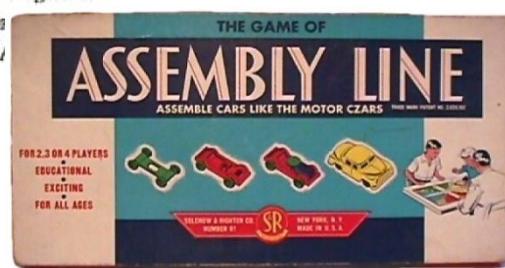
SOLUCIÓN

Segmentación de cauce
(*pipelining*)

Concepto de segmentación



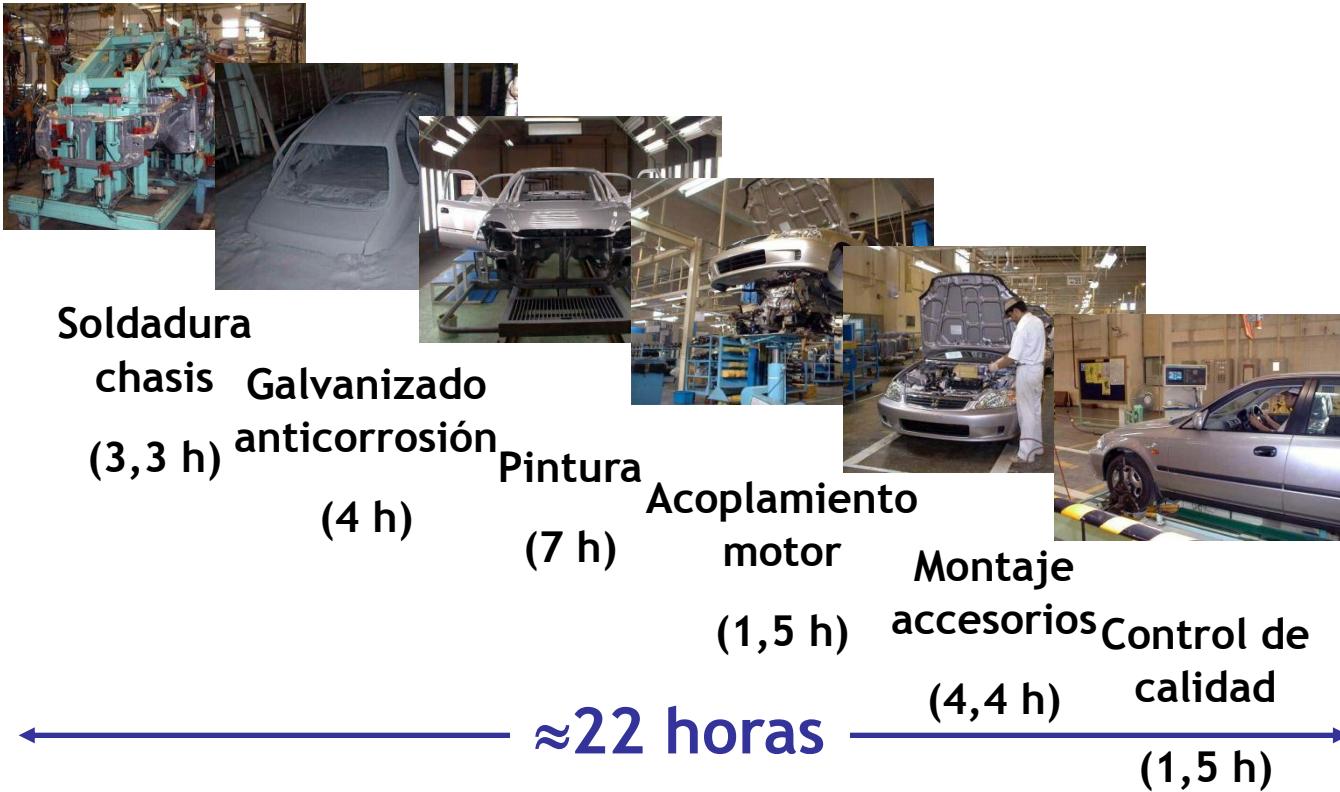
Montaje final del vehículo del modelo T en la fábrica de Highland Park (Detroit, EUA), propiedad de Ford. En ella se ensamblan grandes unidades constructivas (chasis y carrocería) que forman el vehículo.



SEGMENTACIÓN EN UNA FÁBRICA DE AUTOMÓVILES

Cadena de montaje

Concepto de segmentación

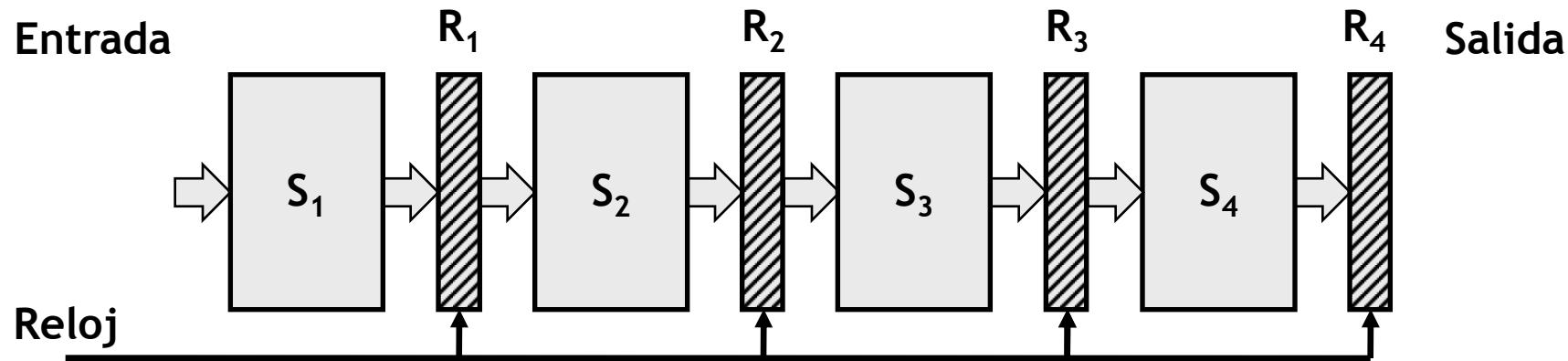


SEGMENTACIÓN EN UNA FÁBRICA DE AUTOMÓVILES
Subdividir el proceso en n etapas, permitiendo
el solapamiento en la fabricación de automóviles

Concepto de segmentación

S_i : etapa de segmentación i -ésima

R_i : registro de segmentación de la etapa i -ésima

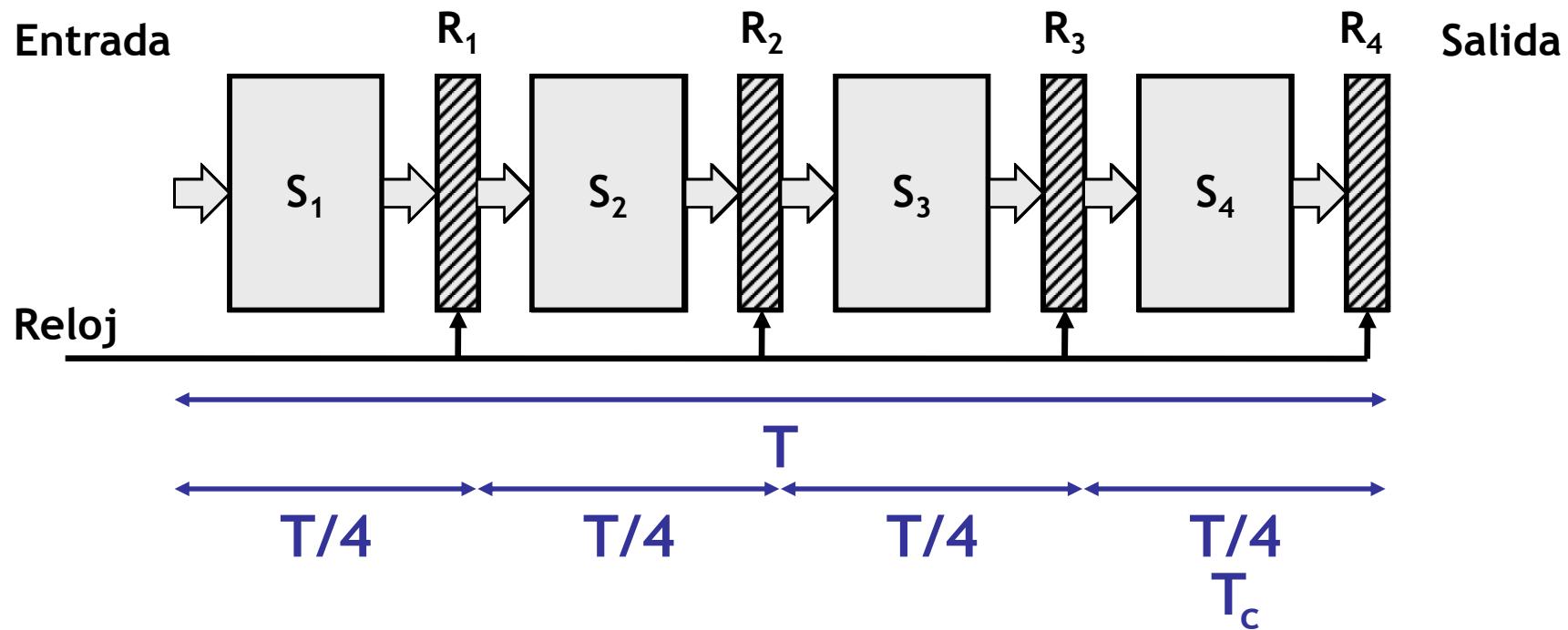


SEGMENTACIÓN EN UN PROCESADOR

Subdividir el procesador en n etapas, permitiendo el solapamiento en la ejecución de instrucciones

Concepto de segmentación

Las instrucciones entran por un extremo del cauce, son procesadas en distintas etapas y salen por el otro extremo.



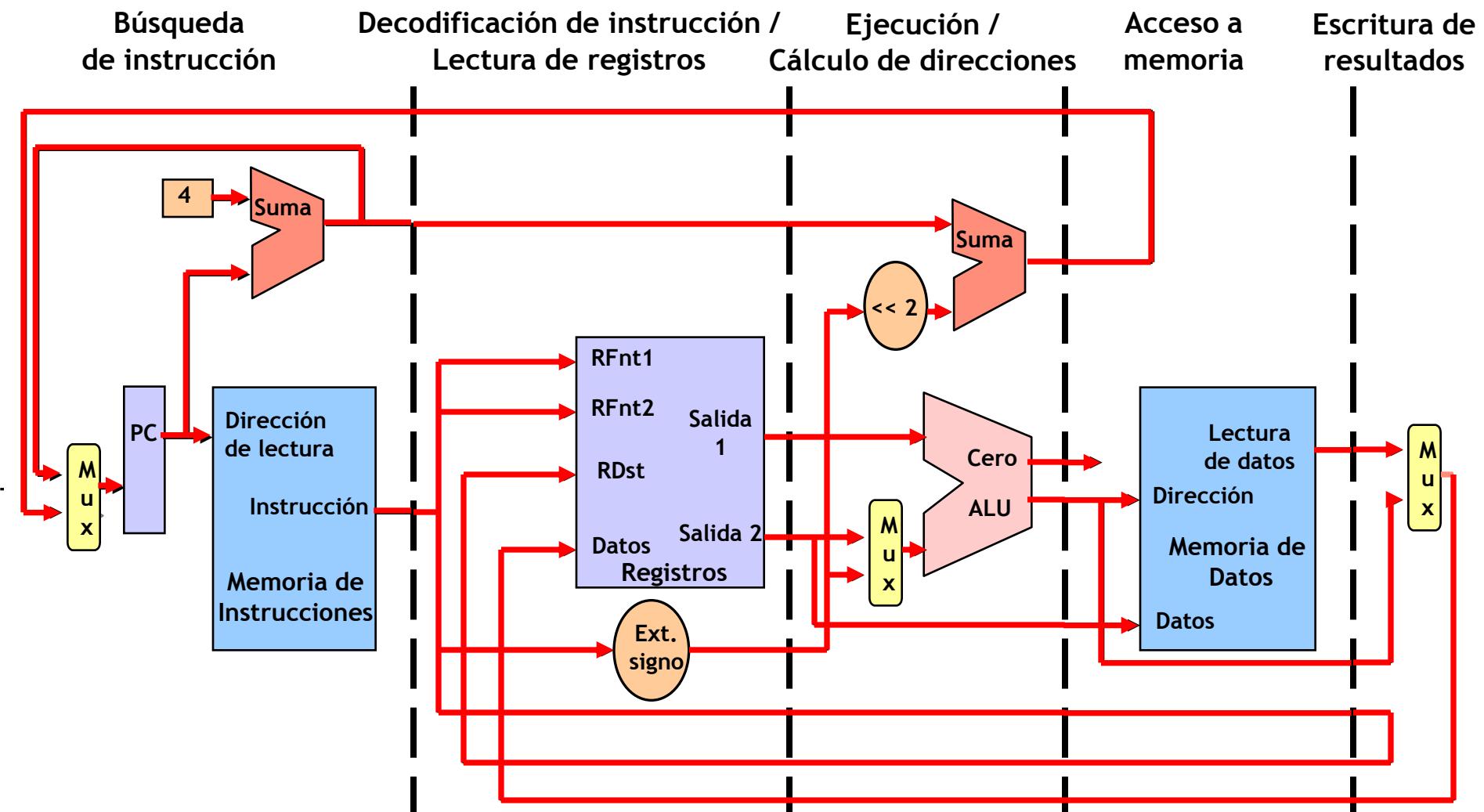
Cada instrucción individual se sigue ejecutando en un tiempo T ...

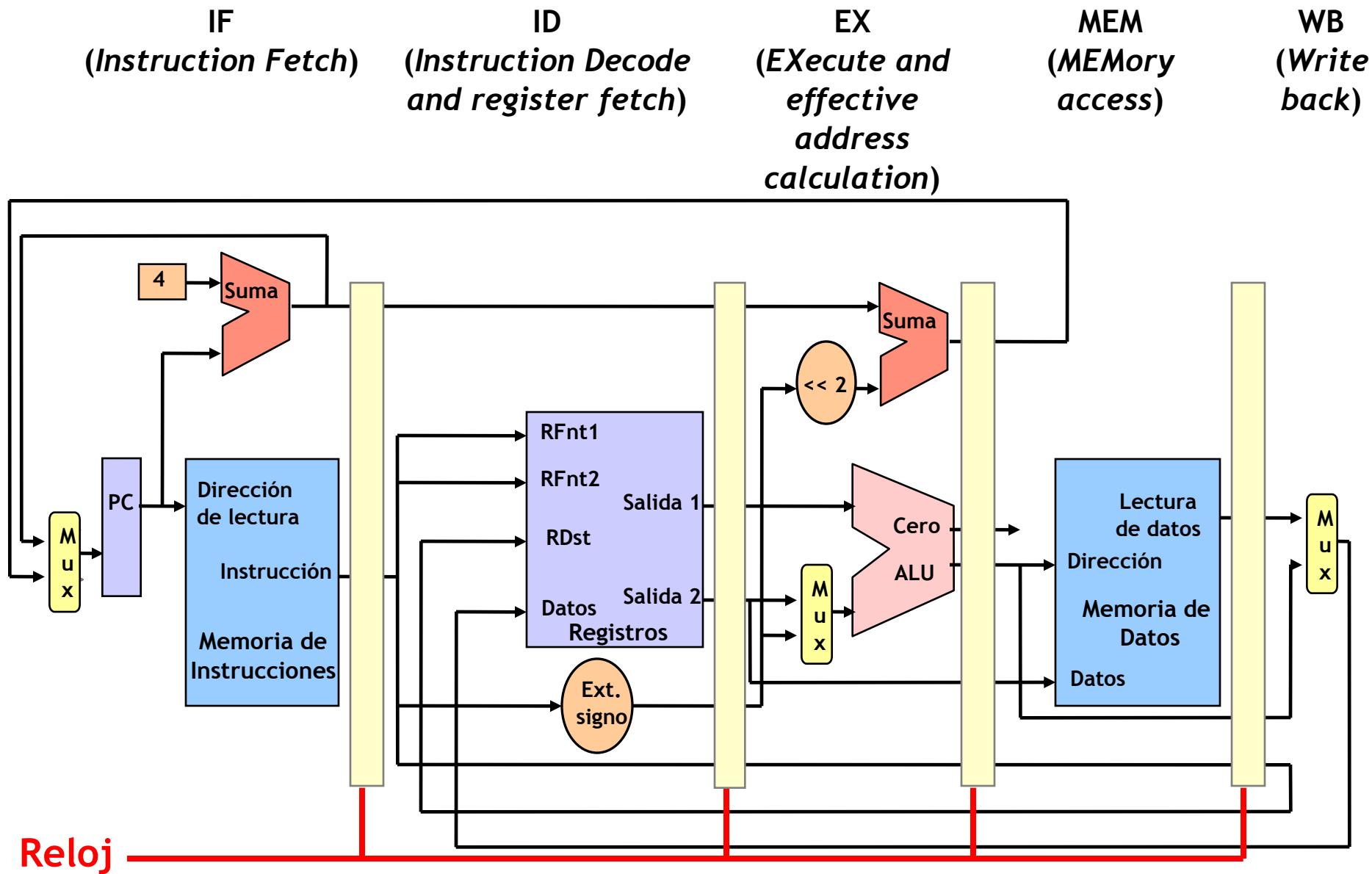
...pero hay varias instrucciones ejecutándose simultáneamente

Segmentación de cauce

- Concepto de segmentación
- Ejemplo de segmentación
- Aceleración
- Riesgos
- Influencia en el repertorio de instrucciones
- Funcionamiento superescalar

Ejemplo de segmentación





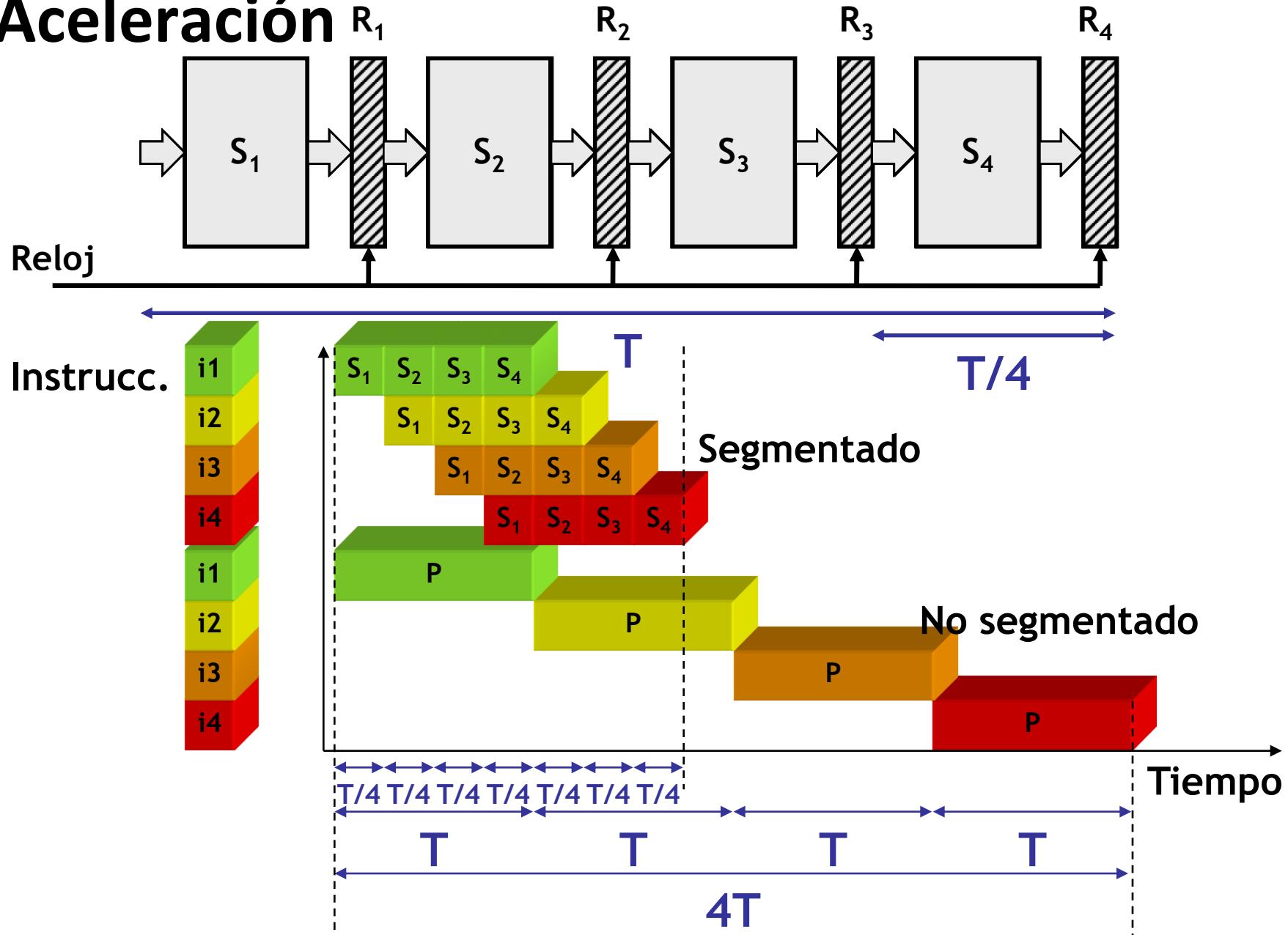
Ejemplo de segmentación

- Cada etapa del cauce debe completarse en un ciclo de reloj
- Fases de captación y de memoria
 - Si acceden a memoria principal, el acceso es varias veces más lento
 - La caché permite acceso en un único ciclo de reloj
- El periodo de reloj se escoge de acuerdo a la etapa más larga del cauce

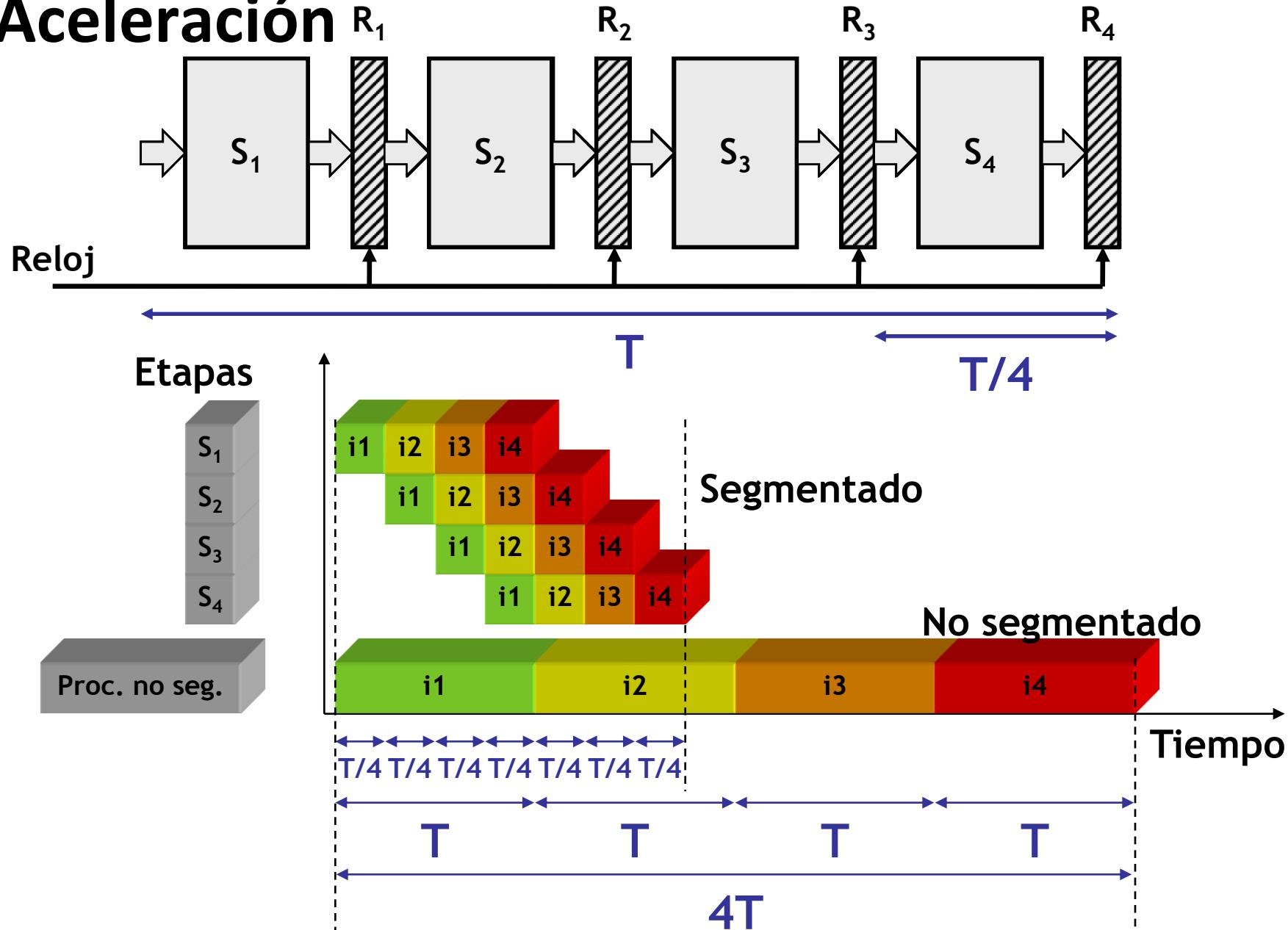
Segmentación de cauce

- Concepto de segmentación
- Ejemplo de segmentación
- Aceleración
- Riesgos
- Influencia en el repertorio de instrucciones
- Funcionamiento superescalar

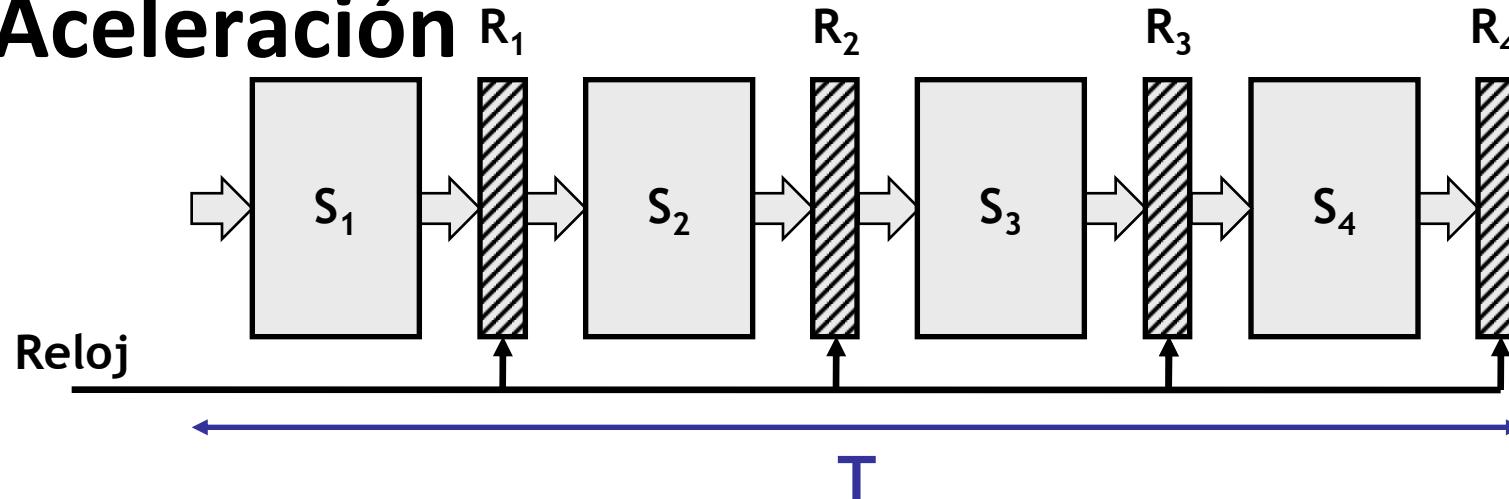
Aceleración



Aceleración



Aceleración



Aceleración en el ejemplo

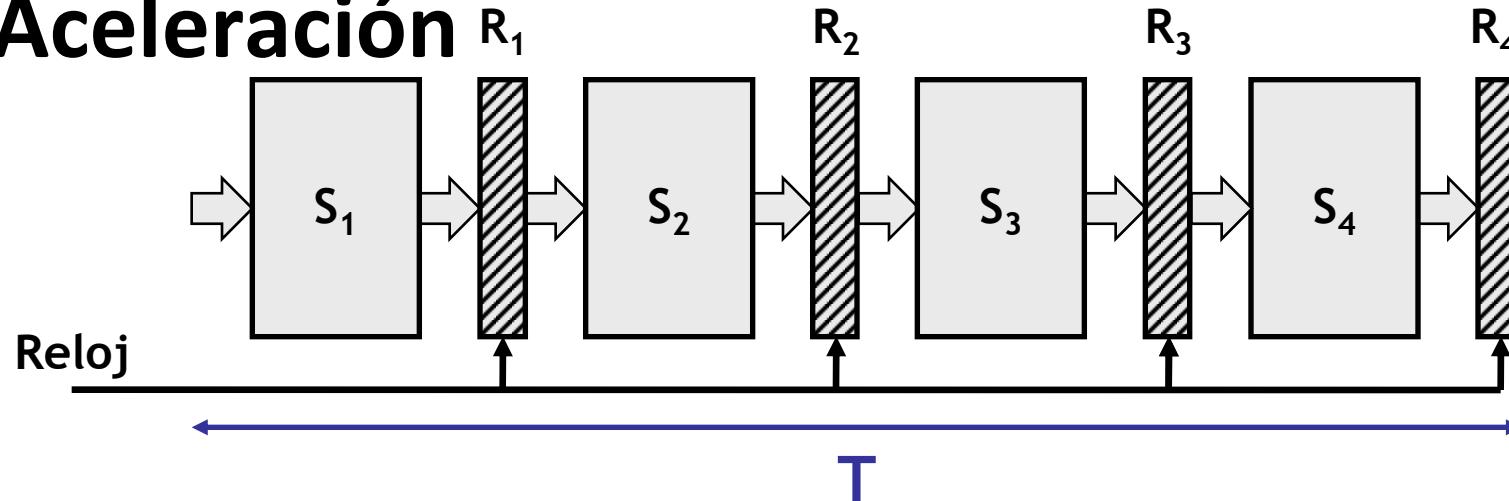
$$\text{Aceleración} = \frac{\text{Tiempo máquina no-segmentada}}{\text{Tiempo máquina segmentada}} = \frac{4T}{7T/4} = \frac{4}{7/4} = \frac{16}{7}$$

Aceleración ideal

$$\text{Aceleración ideal} = \frac{kT}{(n-1+k)T/n} = \frac{nkT}{(n+k-1)T} = n \quad k \rightarrow \infty$$

La aceleración ideal coincide con el número de etapas de segmentación

Aceleración



Causas que disminuyen la aceleración

- ✗ Coste de la segmentación
 - ✗ Duración del ciclo de reloj
impuesto por etapa más lenta
 - ✗ Riesgos (*hazards*) \Rightarrow Bloqueo del avance de instrucciones
- $\} \Rightarrow T_c > T/n$

Segmentación de cauce

- Concepto de segmentación
- Ejemplo de segmentación
- Aceleración
- Riesgos
- Influencia en el repertorio de instrucciones
- Funcionamiento superescalar

Riesgos

Riesgo

Situación que impide la ejecución de la siguiente instrucción del flujo del programa durante el ciclo de reloj designado



Obliga a modificar la forma en la que avanzan las instrucciones hacia las etapas siguientes



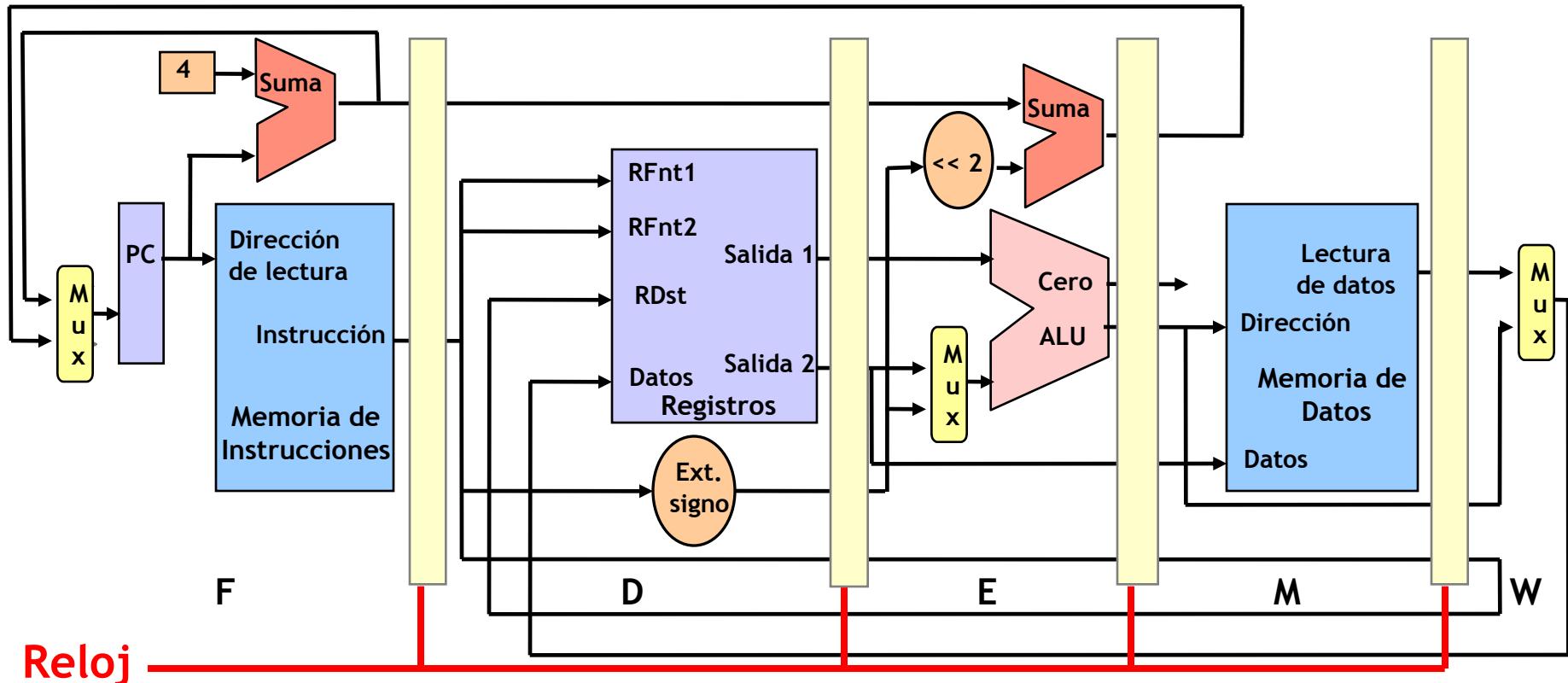
Reducción de las prestaciones logradas por la segmentación

Riesgos

■ Supongamos las siguientes etapas:

- F: búsqueda (**fetch**) de instrucción.
- D: **decodificación** de instrucción / lectura de registros.
- E: **ejecución** / cálculo de direcciones
- M: acceso a **memoria**.
- W: escritura (**write**) de resultados.

Riesgos



Riesgos estructurales

- Conflicto por el **empleo de recursos**, dos instrucciones necesitan un mismo recurso.
- Ej. 1: lectura de dato + captación suponiendo **una única memoria** para datos e instrucciones.

lw	r4,20(r1)	F D E M W
and	r7,r2,r5	F D E M W
or	r8,r6,r2	F D E M W
add	r9,r2,r2	F D E M W
lw	r4,20(r1)	F D E M W
and	r7,r2,r5	F D E M W
or	r8,r6,r2	F D E M W
add	r9,r2,r2	- F D E M W

Riesgos estructurales

- Ej. 2: ejecución de una operación con más de un ciclo en E.

add	rx,rx,rx	F D E M W
mul	rd,rs,rs	F D E E E M W
add	rx,rx,rx	F D E M W
add	rx,rx,rx	F D E M W
add	rx,rx,rx	F D E M W

add	rx,rx,rx	F D E M W
mul	rd,rs,rs	F D E E E M W
add	rx,rx,rx	F D - - E M W
add	rx,rx,rx	F - - D E M W
add	rx,rx,rx	F D E M W

Riesgos estructurales

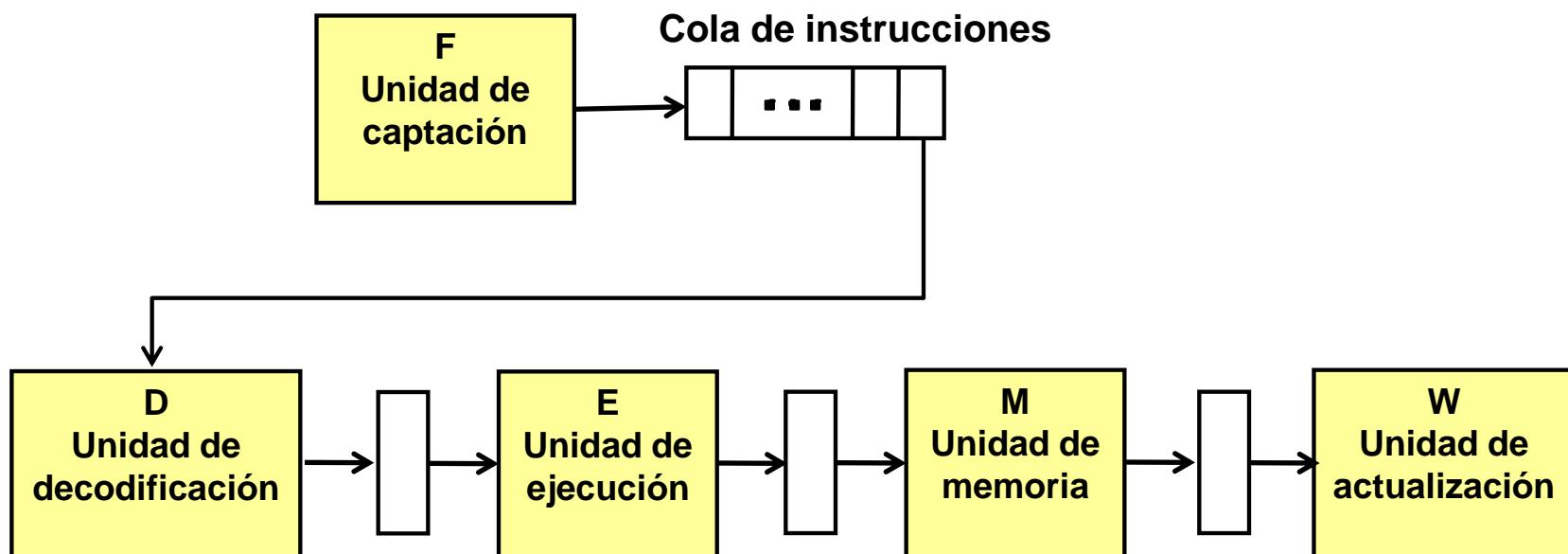
- Ej. 3: fallo de caché al captar una instrucción.

add	rx,rx,rx	F D E M W
sub	rx,rx,rx	F F F F D E M W
add	rx,rx,rx	F D E M W
add	rx,rx,rx	F D E M W
add	rx,rx,rx	F D E M W

add	rx,rx,rx	F D E M W
sub	rx,rx,rx	F F F F D E M W
add	rx,rx,rx	- - - F D E M W
add	rx,rx,rx	F D E M W
add	rx,rx,rx	F D E M W

Riesgos estructurales

- Para reducir el efecto de los fallos de caché se suelen captar instrucciones antes de que sean necesarias (precaptación) y se almacenan en una cola de instrucciones.



Riesgos (por dependencias) de datos

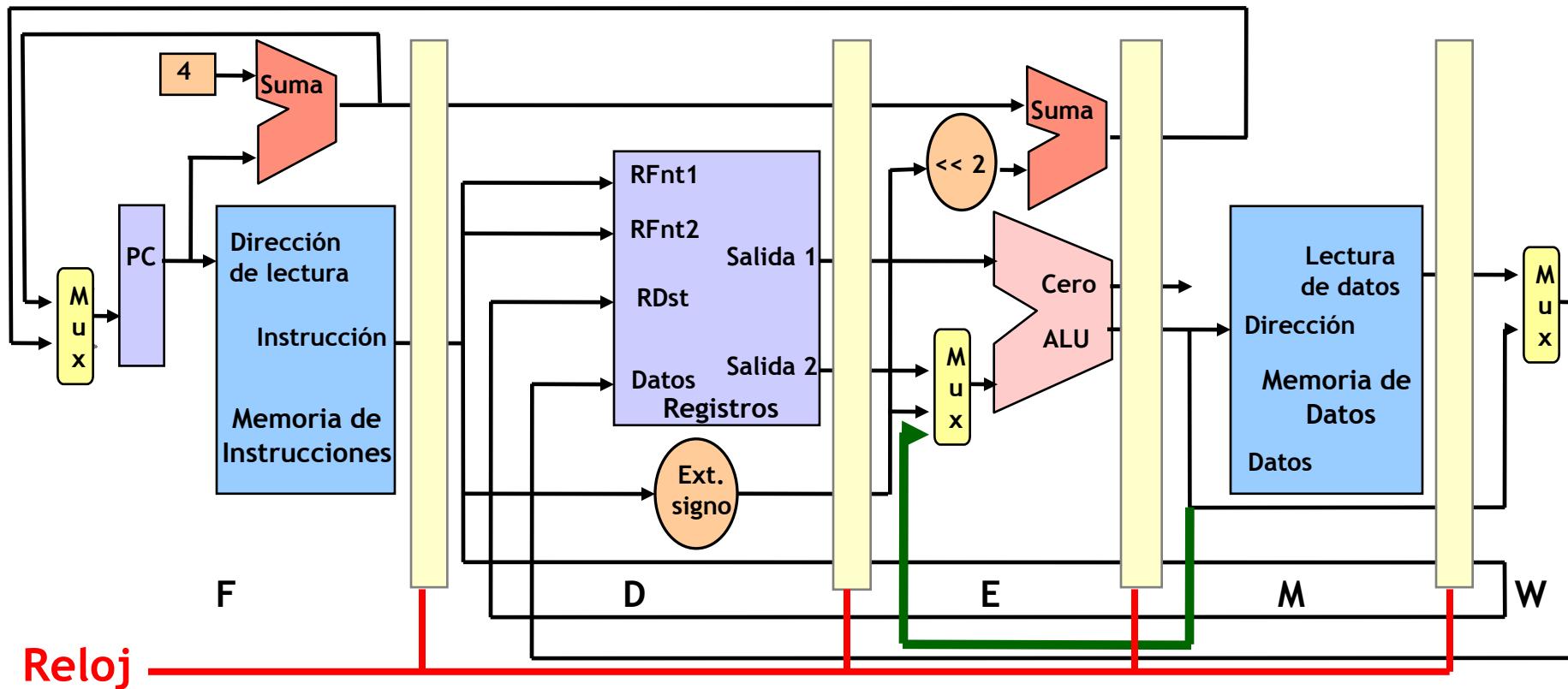
- Acceso a datos cuyo valor actualizado depende de la ejecución de instrucciones precedentes.

sub	r2,r1,r3	F D E M W
and	r7,r2,r5	F D E M W
or	r8,r6,r2	F D E M W
add	r9,r2,r2	F D W M W

sub	r2,r1,r3	F D E M W
and	r7,r2,r5	F D - - D E M W
or	r8,r6,r2	F - - - D E M W
add	r9,r2,r2	F D W M W

Riesgos (por dependencias) de datos

- El nuevo r2 está a la salida de la ALU al terminar E.
 - Si r2 se envía de nuevo a la ALU se elimina el retardo (*register forwarding*).



Riesgos (por dependencias) de datos

- Las dependencias de datos las descubre el hardware al decodificar las instrucciones.
- Alternativamente puede resolverlas el compilador:

sub	r2,r1,r3	F D E M W
nop		F D E M W
nop		F D E M W
nop		F D E M W
and	r7,r2,r5	F D E M W

- Ventajas:
 - Hardware más simple
 - Reorganizar instrucciones para hacer trabajo útil en lugar de NOP
- Inconveniente:
 - Aumenta el tamaño del código

Riesgos de control

- Consecuencia de la ejecución de instrucciones de salto.
- Salto **incondicional**:

br	L1	F D E M W
and	r2,r1,r4	F D E - -
sub	r5,r6,r7	F D - - -
or	r8,r1,r6	F - - - -
L1:add	r6,r1,r4	F D E M W

- Se pierden 3 ciclos (**huecos de retardo de salto***), ya que tras captar la instrucción br, hasta después del 4º ciclo (es decir, pasados otros 3 ciclos) no se conoce la dirección de salto.

Riesgos de control

- Importante averiguar la dirección de salto **lo antes posible**, por ej. en la etapa de decodificación:

br	L1	F D E M W
and	r2,r1,r4	F - - - -
sub	r5,r6,r7	
or	r8,r1,r6	
L1:add	r6,r1,r4	F D E M W

- Se pierde 1 ciclo, ya que tras captar la instrucción br, después del 2º ciclo ya se conoce la dirección de salto.

Riesgos de control

■ Salto condicional:

beq	r2,r3,L1	F D E M W
and	r2,r1,r4	F D E M W
sub	r5,r6,r7	F D E M W
or	r8,r1,r6	F D E M W
L1:add	r6,r1,r4	F D E M W

- Si se produce el salto se pierden 3 ciclos.
- Si no se produce el salto, no se pierden ciclos.

Riesgos de control

■ Degradación de prestaciones debida a los saltos.

- Supongamos algún mecanismo hardware que permita descartar la ejecución de las instrucciones siguientes si se produce el salto.
 - b : nº de ciclos desperdiciados cuando se produce el salto.
 - p_b : probabilidad de que se ejecute una instrucción de salto
 - (entre 0,15 y 0,30 normalmente)
 - p_t : probabilidad de que realmente se produzca el salto cuando se ejecuta una instrucción de salto
 - $p_e = p_b p_t$: probabilidad efectiva de que se produzca un salto
 - CPI: nº de ciclos de reloj por instrucción (suponer CPI = 1 sin saltos)

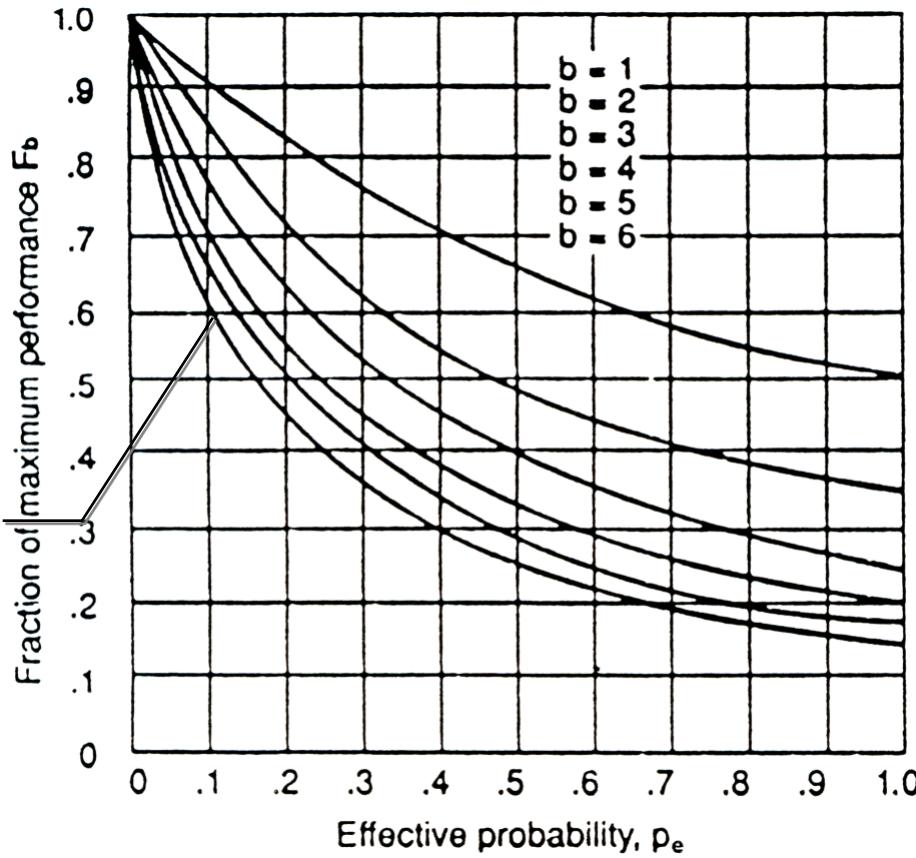
$$\text{CPI} = (1 - p_b) (1) + p_b [p_t (1 + b) + (1 - p_t) (1)] = 1 + p_b p_t b = 1 + p_e b$$

Riesgos de control

- F_b : fracción de máximas prestaciones (relación entre el nº de ciclos CPI si no hubiera saltos y el nº de ciclos CPI con saltos)

$$F_b = \frac{1}{1 + p_e b}$$

La degradación crece rápidamente al crecer p_e
(más rápidamente a medida que b es mayor)



Salto retardado (*delayed branch*)

- En lugar de desperdiciar las etapas posteriores a la de salto, una o más instrucciones parcialmente completadas se completarán antes de que el salto tenga efecto.
- El compilador busca instrucciones **anteriores** lógicamente al salto que pueda colocar tras el salto.
- Si el salto es condicional, las instrucciones colocadas detrás no deben afectar a la condición de salto.

Antes:

```
mov r1,#3  
jmp etiq  
nop
```

Después:

```
jmp etiq  
mov r1,#3
```

Salto retardado (*delayed branch*)

- Otra posibilidad es colocar la(s) instrucción(es) **destino** de un salto tras el salto.

Antes:

```
call subr
```

```
nop
```

```
...
```

subr:

```
mov r3,#100
```

Después:

```
call subr+4
```

```
mov r3,#100
```

```
...
```

subr:

```
mov r3,#100
```

- Esto no funcionaría para saltos condicionales
- No podemos “subir” una instrucción que sólo tiene que ejecutarse algunas veces (cuando el salto se produce) a una posición donde siempre se ejecuta.

Annulling branch

- Un salto de este tipo ejecuta la(s) instrucción(es) siguiente(s) sólo si el salto se produce, pero la(s) ignora si el salto no se produce.
- Con un salto de este tipo, el destino de un salto condicional sí puede colocarse tras el salto.

Antes:

```
bz else  
nop  
; código then  
...  
else:
```

```
    mov r3,#100
```

Después:

```
bz,a else+4  
mov r3,#100  
; código then  
...  
else:
```

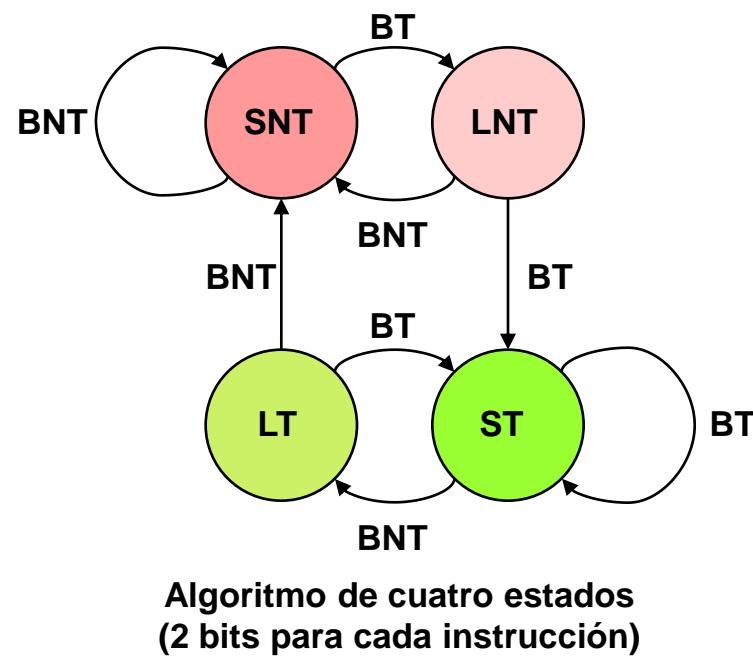
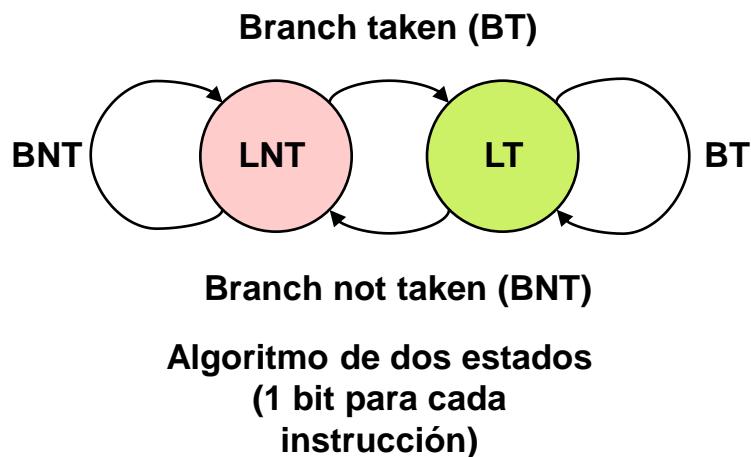
```
    mov r3,#100
```

Predicción de saltos

- Intentar predecir si una instrucción de salto concreta dará lugar a un salto (**branch taken**) o no (**branch not taken**).
 - Si los resultados de las instrucciones de salto condicional fueran aleatorios, comenzar a ejecutar las instrucciones siguientes al salto desperdiciaría ciclos en la mitad de las ocasiones.
 - Se pueden minimizar las pérdidas de ciclos inútiles si para cada instrucción de salto se puede predecir con un acierto > 50% si el salto se producirá o no.
 - Se pueden hacer predicciones distintas si el salto es hacia direcciones menores o mayores.
- Tipos de predicción:
 - **Estática**: se toma la misma decisión para cada tipo de instrucción
 - **Dinámica**: cambia según la historia de ejecución de un programa

Predicción dinámica de saltos

- ST: Muy probable saltar
- LT: Probable saltar
- LNT: Probable no saltar
- SNT: Muy probable no saltar



Segmentación de cauce

- Concepto de segmentación
- Ejemplo de segmentación
- Aceleración
- Riesgos
- **Influencia en el repertorio de instrucciones**
- Funcionamiento superescalar

Modos de direccionamiento

- Es deseable que un operando no necesite más de **un acceso a memoria**
 - Constante
 - Registro
 - Indirecto a través de registro
 - Indexado ($EA = \text{reg.} + \text{desp.}$, o $EA = \text{reg.} + \text{reg.}$)
- Es deseable que sólo puedan acceder a memoria las instrucciones de **carga y almacenamiento**

Modos de direccionamiento

■ Modo de direc. complejo, 2 accesos a memoria

lw	r4 , (20(r1))	F D E M M W
and	r7 , r2 , r5	F D E - M W

■ Modo de direc. más simple, 1 acceso a memoria

lw	r3 , 20(r1)	F D E M W
lw	r4 , (r3)	F D E M W
and	r7 , r2 , r5	F D E M W

- Idéntica duración, pero hardware más sencillo

Códigos de condición

- Las dependencias que introducen los bits de condición dificultan al compilador reordenar el código (deseable para evitar riesgos).
- En el ejemplo siguiente, la decisión de saltar se produce tras la etapa E de la instrucción cmp
- Es deseable elegir en cada instr. si afecta o no a los cód. de condición, para reordenar el código:

Antes:

```
add r1,r2  
cmp r3,r4  
jz etiq
```

Después:

```
cmp r3,r4  
add r1,r2  
jz etiq
```

Al reordenar el código, se puede tomar la decisión de salto un ciclo antes, y desperdiciar un ciclo menos tras el salto

Segmentación de cauce

- Concepto de segmentación
- Ejemplo de segmentación
- Aceleración
- Riesgos
- Influencia en el repertorio de instrucciones
- Funcionamiento superescalar

Funcionamiento superescalar

■ Procesamiento segmentado

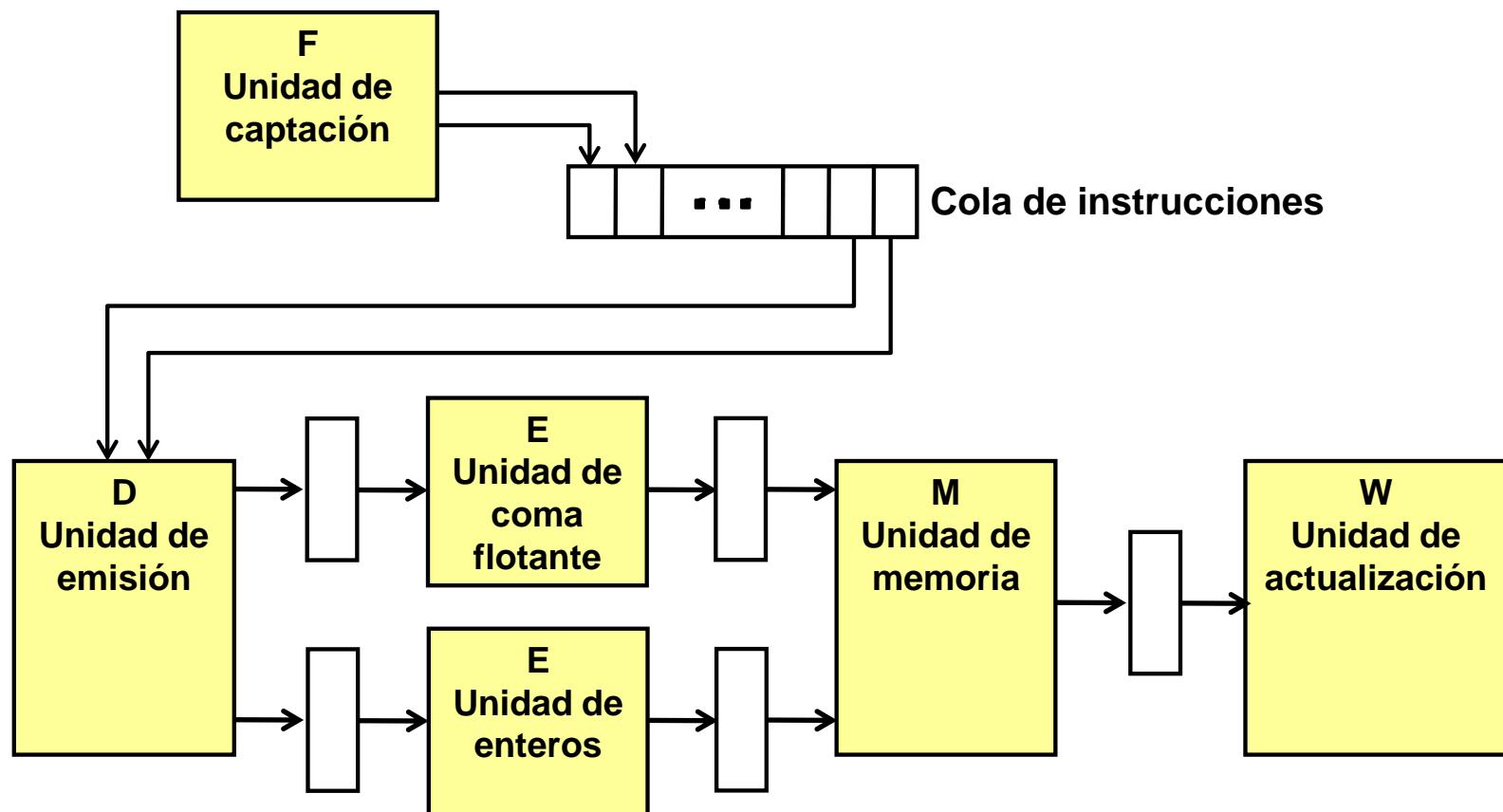
- Una instrucción tras otra
- Rendimiento ideal: una instrucción por ciclo

■ Procesamiento superescalar

- Varias instrucciones en **paralelo**
- Necesidad de **varias unidades funcionales**
- **Emisión múltiple**: se puede comenzar a ejecutar más de una instrucción por ciclo de reloj
- Rendimiento: **más de una instrucción por ciclo**
- Es fundamental poder captar instrucciones rápidamente: conexión ancha con caché + cola de instrucciones

Funcionamiento superescalar

- Ej.: Procesador con dos unidades de ejecución:



Funcionamiento superescalar

- El efecto negativo de los riesgos es más pronunciado.
- El compilador puede reordenar instrucciones para evitar riesgos.

fadd	rx, rx, rx	F D E ₁ E ₂ E ₃ M W
add	rx, rx, rx	F D E M W
fsub	rx, rx, rx	F D E ₁ E ₂ E ₃ M W
sub	rx, rx, rx	F D E M W

- Las instrucciones pueden emitirse en orden y finalizar de forma desordenada (ej. add finaliza antes que fadd)
 - Múltiples problemas y soluciones, que se verán en Arquitectura de Computadores

Core i7 (Nehalem)

- 4 cores/chip
- Segmentación:
 - 16 etapas
- Superescalar:
 - 4 instrucciones en paralelo
- Caches:
 - L1: 32KB I + 32KB D
 - L2: 256KB
 - L3: 8MB, compartida por los 4 cores

