

# Introducción

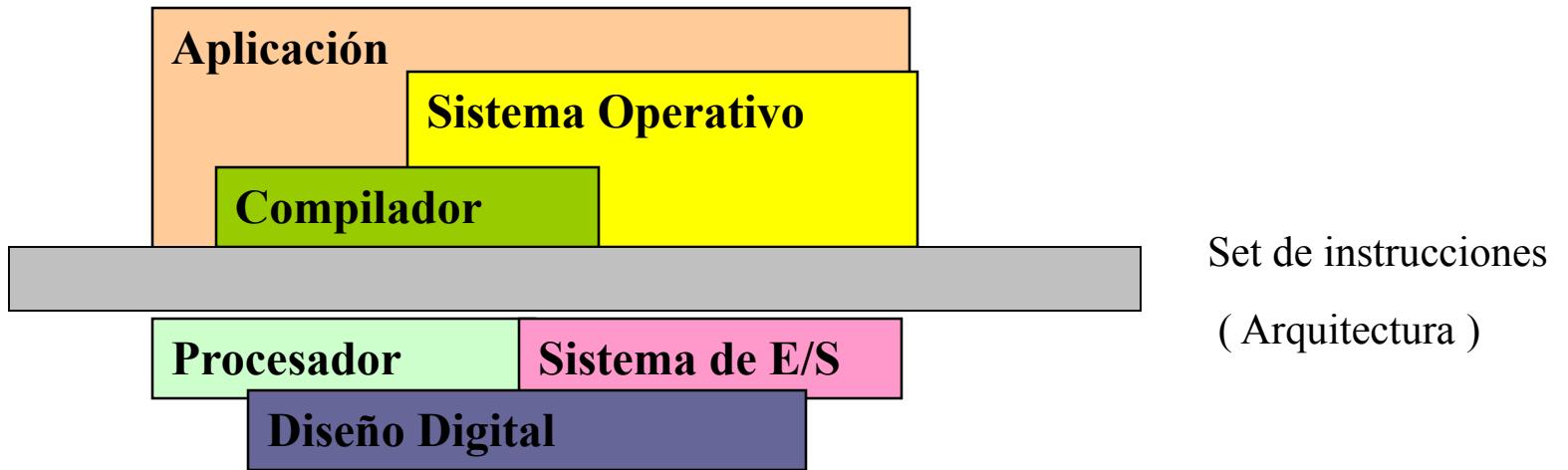
# Presentación de la materia

## Objetivos

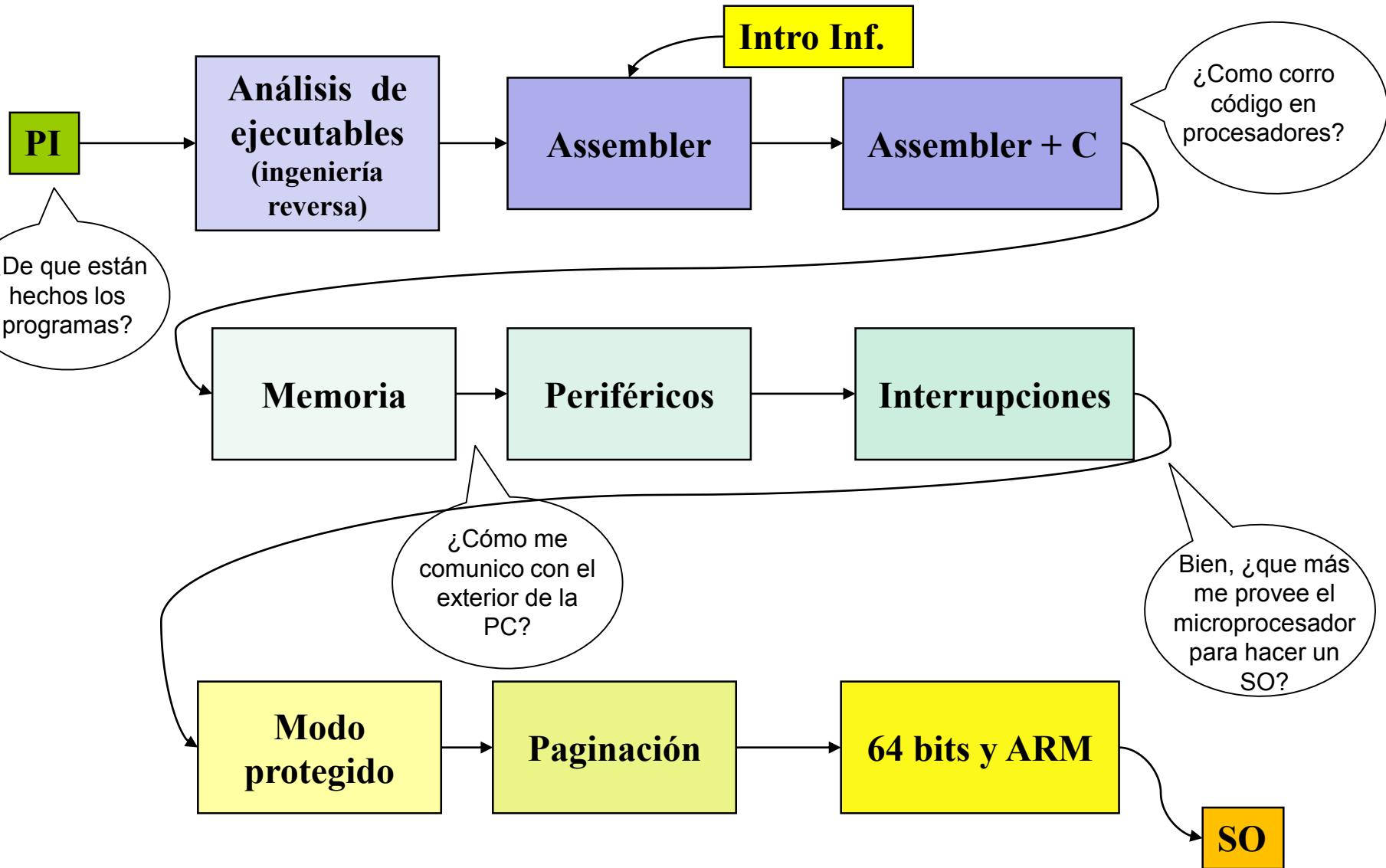
- ✓ Conocer y programar diferentes tipos de procesadores.
- ✓ Conocer y programar los periféricos de la PC.
- ✓ Programar la familia Intel en bajo nivel.
- ✓ Base de conocimientos para la materia Sistemas Operativos.

# Presentación de la materia

## Niveles de abstracción

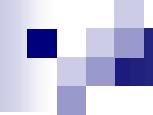


# Presentación de la materia



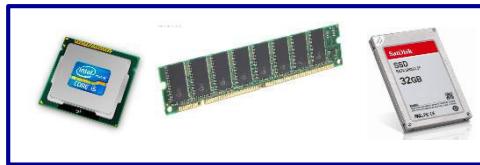
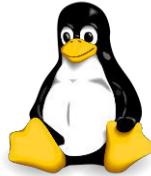
# Bibliografía

- Los microprocesadores Intel. Barry B. Brey. Prentice Hall.
- Organización de Computadoras. Andrew S. Tanenbaum. Prentice Hall
- Organización y Arquitectura de Computadoras. William Stallings. Megabyte.
- Apuntes de la materia.



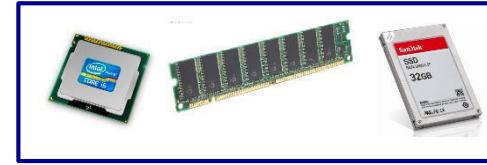
# Arquitecturas

# En la actualidad



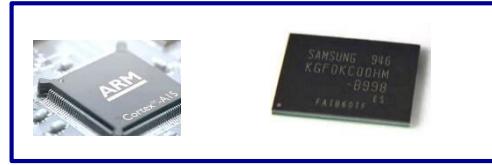
Hardware PC

Variedad de  
Fabricantes



Hardware Mac

Un fabricante



Hardware  
Smartphone

Variedad de  
Fabricantes

# Encuesta !

## En la actualidad (Consolas)



Xbox One X



PS4

Procesador AMD (Jaguar)  
Dual Core 64 bits – 4 núcleos  
c/u

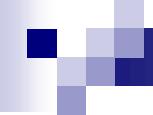
Procesador AMD (Jaguar)  
Dual Core 64 bits – 4 núcleos  
c/u

Memoria 12 GB

Memoria 8 GB

Bus a memoria 384 bits

Bus a memoria 256 bits



# Programas y binarios

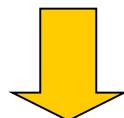
# Compilación y linkedición en C (de PI)

Un programa en C al ser compilado sigue el siguiente camino:

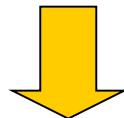
*Programa .C*



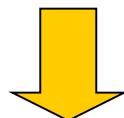
*Preprocesador ( cpp ) (.C)*



*gcc (.S)*



*gas (.O)*



*ld ( ejecutable )*

gcc primero invoca al preprocesador cpp, el cual toma el código fuente original y realiza la macroexpansion de directivas como #include y #define.

Luego gcc toma el control de esa salida y convierte el código en C en código equivalente en Assembler. Genera un archivo con extensión “.S”. Este código en ASM puede ser generado en sintaxis AT&T ó en la sintaxis Intel

Luego el compilador GNU de Assembler “gas” se encargar de generar el código objeto ( con extensión .O )

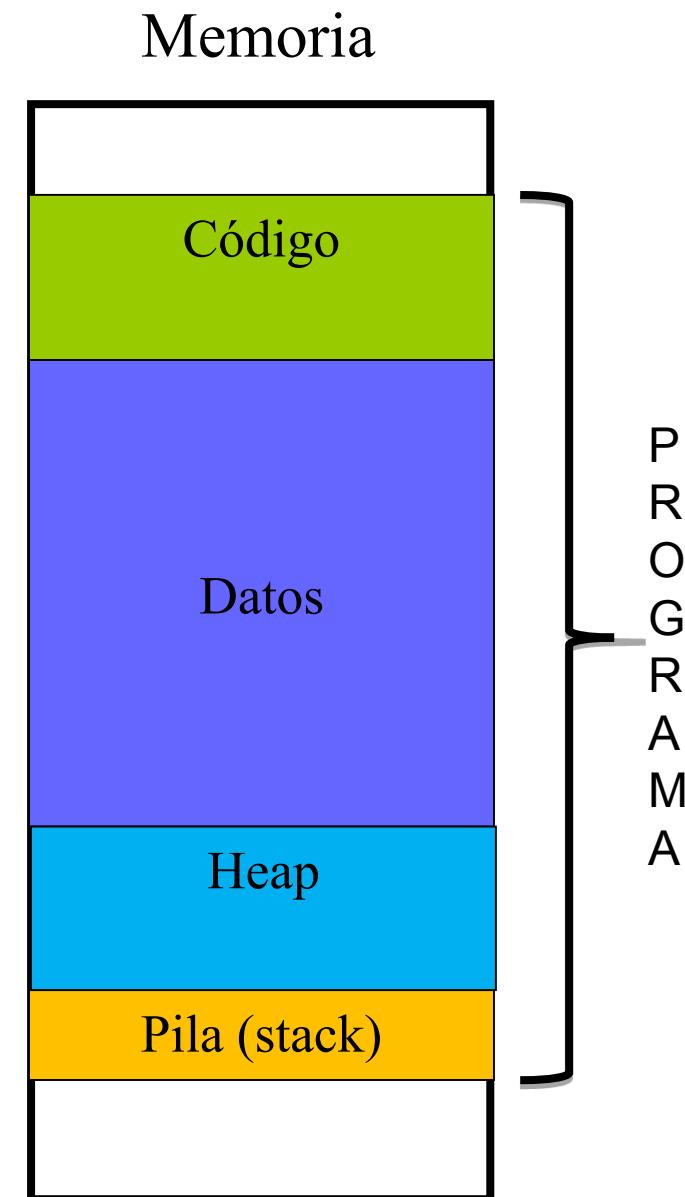
En Linux el ejecutable podrá ser de diferentes formatos, por ej, ELF (mas reciente ) y A.OUT .Se debe tener en cuenta en este punto que el likeditor “ld” también puede generar el modelo flat , también llamado binario.

# Ejecución de un programa



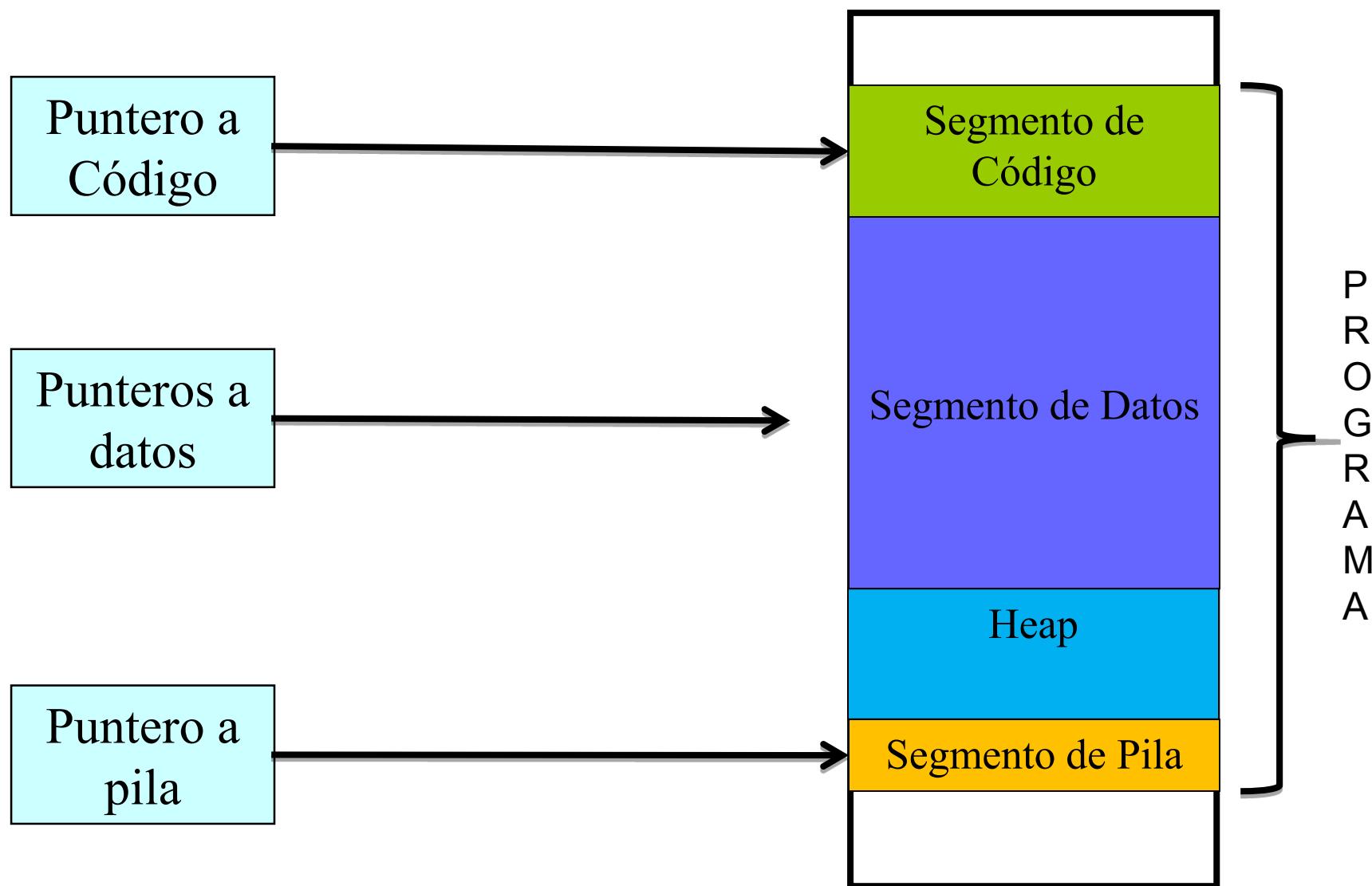
# Programa en memoria

- Código
  - Instrucciones del programa
- Datos
  - Variables estáticas y globales que se iniciar al cargar el programa.
- Heap
  - Memoria dinámica que se reserva y se libera en tiempo de ejecución.
- Pila
  - Argumentos y variables locales a la función.



# Programa en memoria (2)

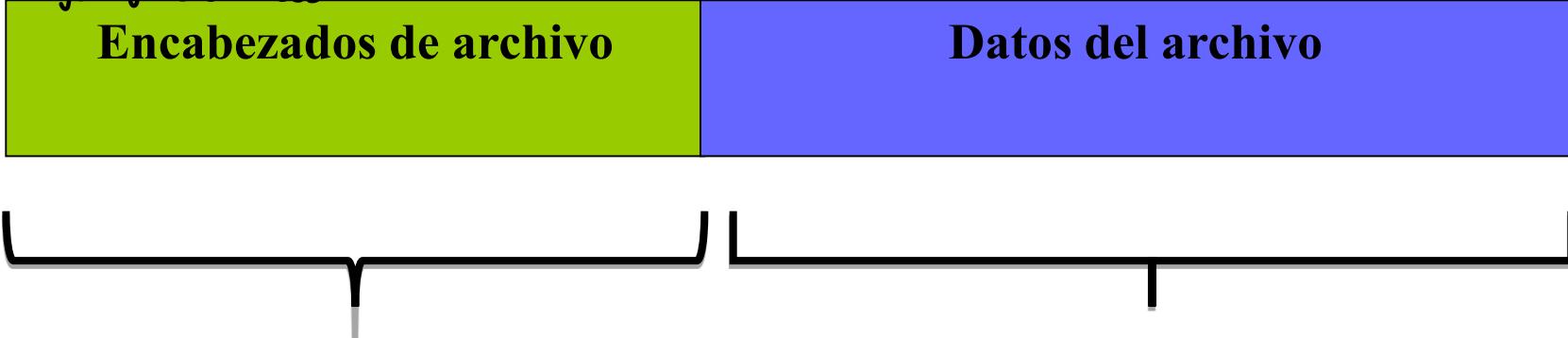
Memoria



# Programa en disco

Ejemplo archivo **a.out**

⇒ gcc genera ambos



- Encabezados de programa  
(Segmentos)
- Encabezado de Secciones
  - (Secciones: text, data, rodata, etc)
- Código
- Datos

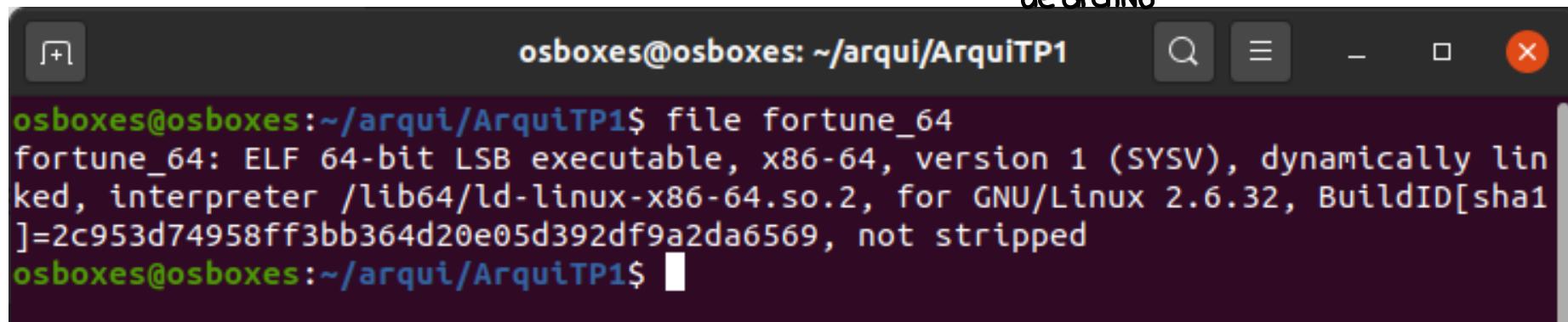
**Todo depende del S.O.**

# Análisis de binario (en Linux)

Archivo ejecutable **fortune\_64** (EJ1 de TP1)

Veamos que tipo de archivo es con el comando **file**

- busca el encabezado del archivo para saber de que tipo es
- comando que determina el tipo de archivo



```
osboxes@osboxes:~/arqui/ArquiTP1$ file fortune_64
fortune_64: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=2c953d74958ff3bb364d20e05d392df9a2da6569, not stripped
osboxes@osboxes:~/arqui/ArquiTP1$
```

Vemos que es un ELF (Executable Linux Format) de 64 bits que usa librerías dinámicas

# Análisis de binario (en Linux)

Lo corremos .....

```
osboxes@osboxes:~/arqui/ArquiTP1$ ./fortune_64
Bienvenido a al adibinador de la fortuna! Este mensaje es muy largo, y puede ser
muy molesto al usuario. Tal vez deberíamos acortarlo?
Cual es tu nombre?: Santiago

Tu fortuna es:
You will forget that you ever knew me.
osboxes@osboxes:~/arqui/ArquiTP1$ █
```

Vemos que básicamente pide un nombre e informa un texto tipo galleta de la fortuna.

# Análisis de binario (en Linux)

Veamos si podemos extraer todas las cadenas de caracteres con el programa **strings**

```
osboxes@osboxes:~/ArquiTP1$ strings fortune_64
/lib64/ld-linux-x86-64.so.2
libc.so.6
__isoc99_scanf
puts
__stack_chk_fail
printf
__libc_start_main
__gmon_start__
GLIBC_2.7
GLIBC_2.4
GLIBC_2.2.5
AWAVA
AUATL
[]A\A]A^A_
Break into jail and claim police brutality.
Never be led astray onto the path of virtue.
You will forget that you ever knew me.
Your society will be sought by people of taste and refinement.
You will be honored for contributing your time and skill to a worthy cause.
Expect the worst, it's the least you can do.
You may not get this fortune
Bienvenido a al adibinador de la fortuna! Este mensaje es muy largo, y puede ser
muy molesto al usuario. Tal vez deberíamos acortarlo?
Cual es tu nombre?:
Tu fortuna es:
;*3$"
```

Comprobamos que las cadenas de texto se guardaron de forma plana.

fortune\_64

Código

DATA

# Análisis de binario (en Linux)

Podremos cambiar un texto y alterar el ejecutable ?

Corramos el editor hexadecimal **bless** y arreglemos un error de ortografía

The screenshot shows the bless hex editor interface. The main window displays the binary content of a file named "fortune\_64". The text portion of the file contains a fortune cookie message with several spelling errors, such as "Break into j", "ail and claim police brutalit", and "will forget that you ever kn". A red box highlights the last few words of the message. Below the main window, there is a search bar with "Search for: adibi" and various conversion tools for different data types (Signed 8 bit, Unsigned 8 bit, Signed 16 bit, Unsigned 16 bit, Signed 32 bit, Unsigned 32 bit, Float 32 bit, Float 64 bit, Hexadecimal, Decimal, Octal, Binary, ASCII Text). At the bottom, there are status bars for "Offset: 0x924 / 0x22ef", "Selection: None", and "INS".

Signed 8 bit:	105
Unsigned 8 bit:	105
Signed 16 bit:	26990
Unsigned 16 bit:	26990
Signed 32 bit:	1768841572
Unsigned 32 bit:	1768841572
Float 32 bit:	1.801152E+25
Float 64 bit:	7.2671008041313E+199
Hexadecimal:	69 6E 61 64
Decimal:	105 110 097 100
Octal:	151 156 141 144
Binary:	01101001 01101110 01100001 01100100
ASCII Text:	inad

Show little endian decoding     Show unsigned as hexadecimal    Offset: 0x924 / 0x22ef    Selection: None    INS

Guardamos el ejecutable.

# Análisis de binario (en Linux)

Volvemos a comer el programa

```
osboxes@osboxes:~/arqui/ArquiTP1$ ./fortune_64
Bienvenido a al adivinador de la fortuna! Este mensaje es muy lar
go, y puede ser muy molesto al usuario. Tal vez deberíamos acorta
rlo?
Cual es tu nombre?:
```

Pudimos cambiar un carácter.  $\Rightarrow$  se puede modificar código

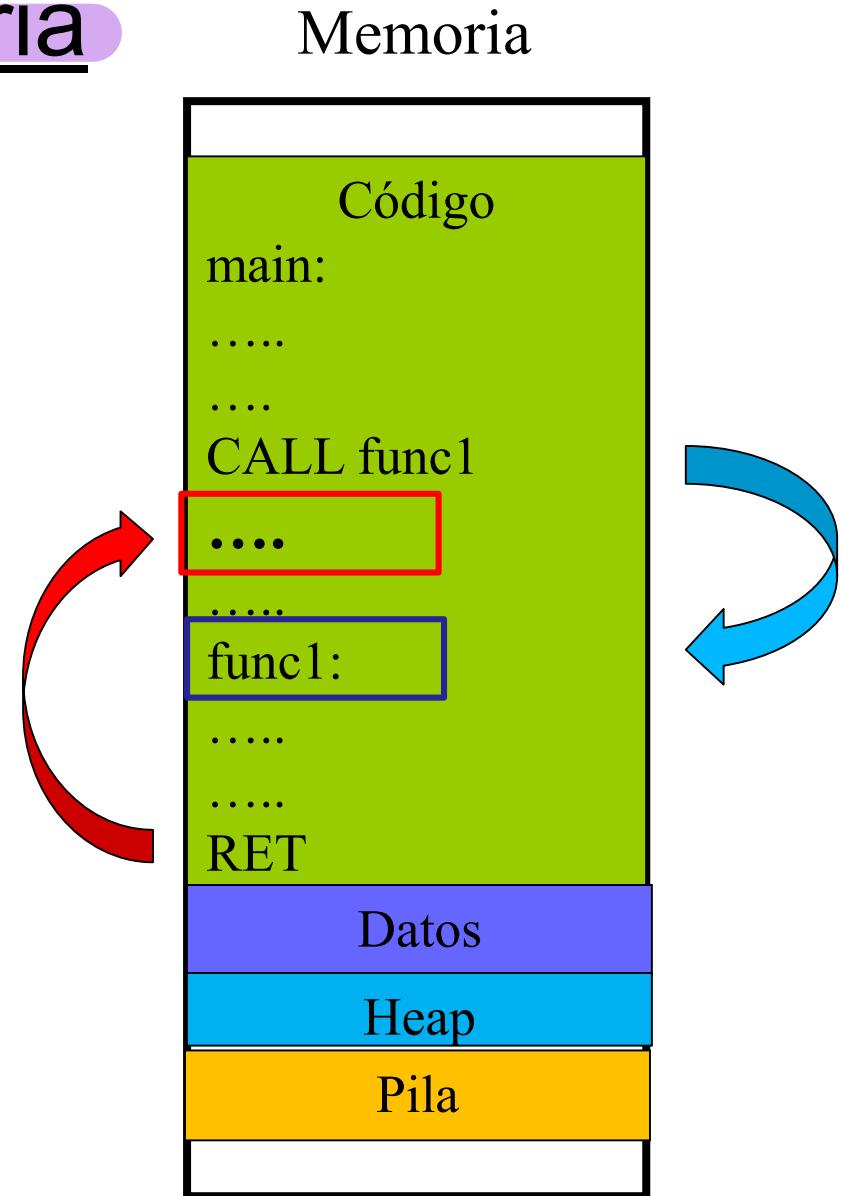
¿Qué conclusiones sacamos?

¿Qué otras cosas podríamos cambiar ?

# Llamadas a Funciones (call)

# Programa en memoria

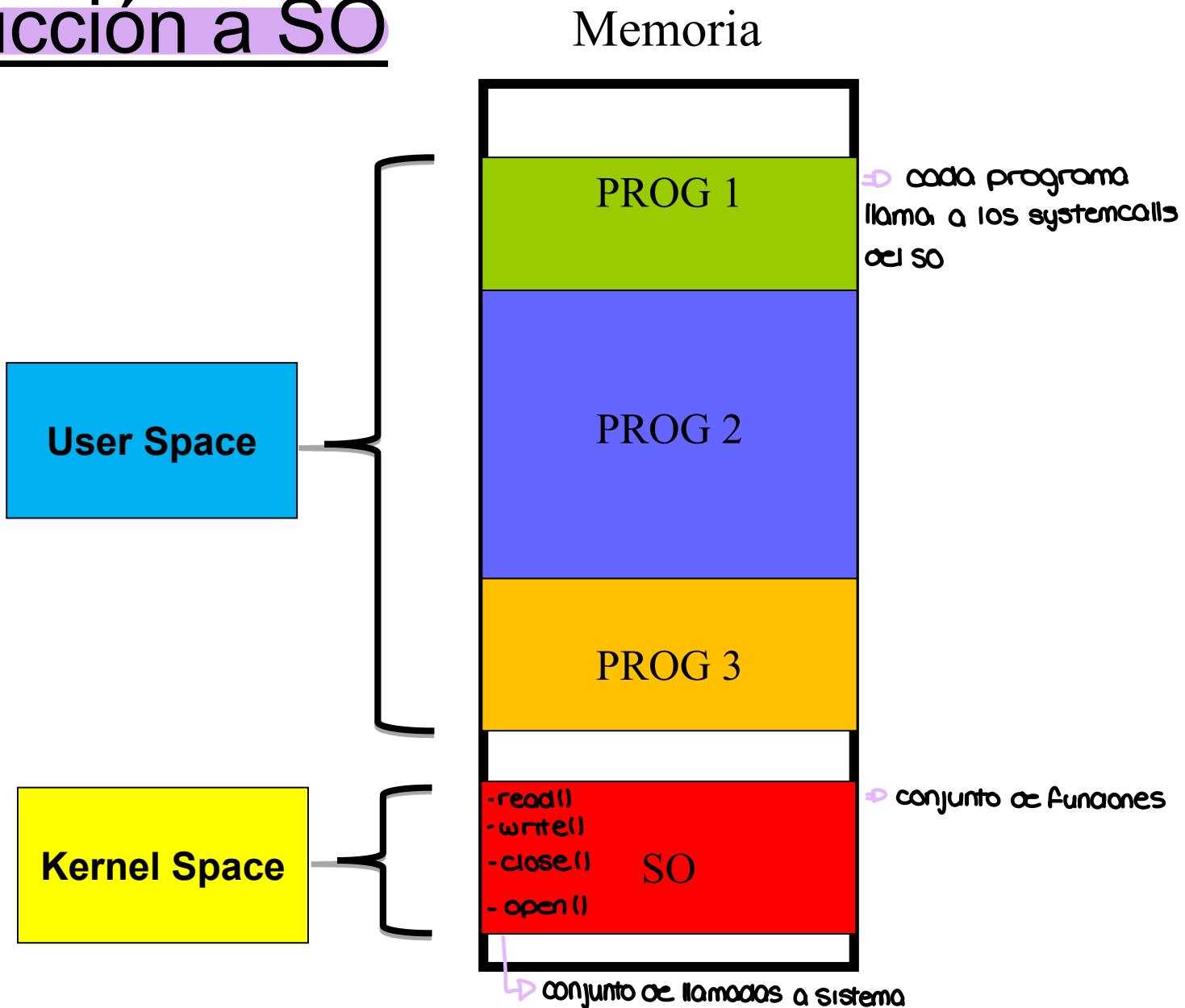
- CALL
  - Instrucción de ASM que permite el llamado a una función.
  - Guarda en la pila la dirección de retorno.
  - La función debe terminar con la instrucción RET de ASM



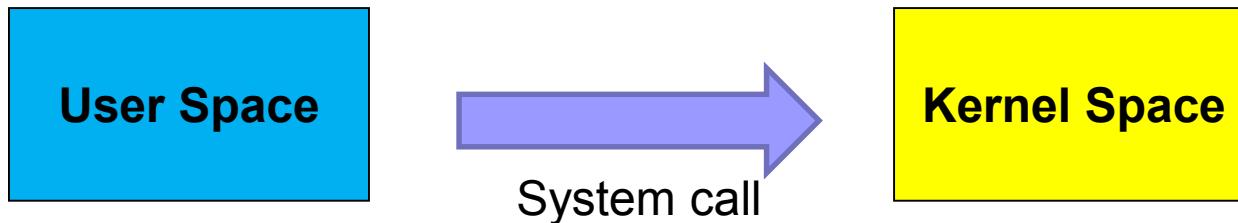


# Llamadas a Sistema Operativo (system calls)

# Introducción a SO



# System Calls



## Formas de ejecutar system call

- INT 80h
  - Interrupción nro 80. Consume mas tiempo
- Instrucciones **SYSCALL/SYSRET** (Intel) y **SYSENTER/SYSEXIT** (AMD)
  - Disponibles desde Pentium II
  - La librería de C depende de la arquitectura
- vsyscall y VDSO
  - Syscall virtuales y Virtual Dynamic Shared Object
  - Linux crea paginas de memoria en user space para acelerar tiempos

## Ejemplo de syscall

⇒ un programa simple necesita multiples llamadas a sistema

- Vemos los syscall que utiliza el programa Hola Mundo en C
  - “\$ strace ./holamundo”

# Ejemplo de syscall

```
mmap2(0xb7722000, 12288, PROT_READ|PROT_WRITE,  
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1a3) = 0xb7722000  
mmap2(0xb7725000, 10972, PROT_READ|PROT_WRITE,  
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb7725000  
close(3) = 0  
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =  
0xb757d000  
set_thread_area({entry_number:-1 -> 6, base_addr:0xb757d900, limit:1048575, seg_32bit:1,  
contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0  
mprotect(0xb7722000, 8192, PROT_READ) = 0  
mprotect(0x8049000, 4096, PROT_READ) = 0  
mprotect(0xb7756000, 4096, PROT_READ) = 0  
munmap(0xb7728000, 43016) = 0  
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 10), ...}) = 0  
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =  
0xb7732000  
write(1, "Hola Mundo", 10) = 10  
exit_group(0) = ?
```

# System Calls en Linux

Linux tiene más de 300 system calls

NR	syscall name	references	%rax	arg0 (%rdi)	arg1 (%rsi)	arg2 (%rdx)	arg3 (%r10)	arg4 (%r8)	arg5 (%r9)
0	read	<a href="#">man/cs/</a>	0x00	unsigned int fd	char *buf	size_t count	-	-	-
1	write	<a href="#">man/cs/</a>	0x01	unsigned int fd	const char *buf	size_t count	-	-	-
2	open	<a href="#">man/cs/</a>	0x02	const char *filename	int flags	umode_t mode	-	-	-
3	close	<a href="#">man/cs/</a>	0x03	unsigned int fd	-	-	-	-	-
4	stat	<a href="#">man/cs/</a>	0x04	const char *filename	struct __old_kernel_stat *statbuf	-	-	-	-
5	fstat	<a href="#">man/cs/</a>	0x05	unsigned int fd	struct __old_kernel_stat *statbuf	-	-	-	-
6	lstat	<a href="#">man/cs/</a>	0x06	const char *filename	struct __old_kernel_stat *statbuf	-	-	-	-



# Procesadores y Lenguaje ASM (en Intel)

# Registros de Intel para programas

⇒ Siempre van de a dos

## **IP: Instruction Pointer (EIP en 32 bits y RIP en 64 bits)**

Puntero a la próxima instrucción a ejecutarse.

## **SP: Stack Pointer (ESP en 32 bits y RSP en 64 bits)**

Puntero a la pila para guardar y extraer datos.

## **Registros de Manejo de Memoria:**

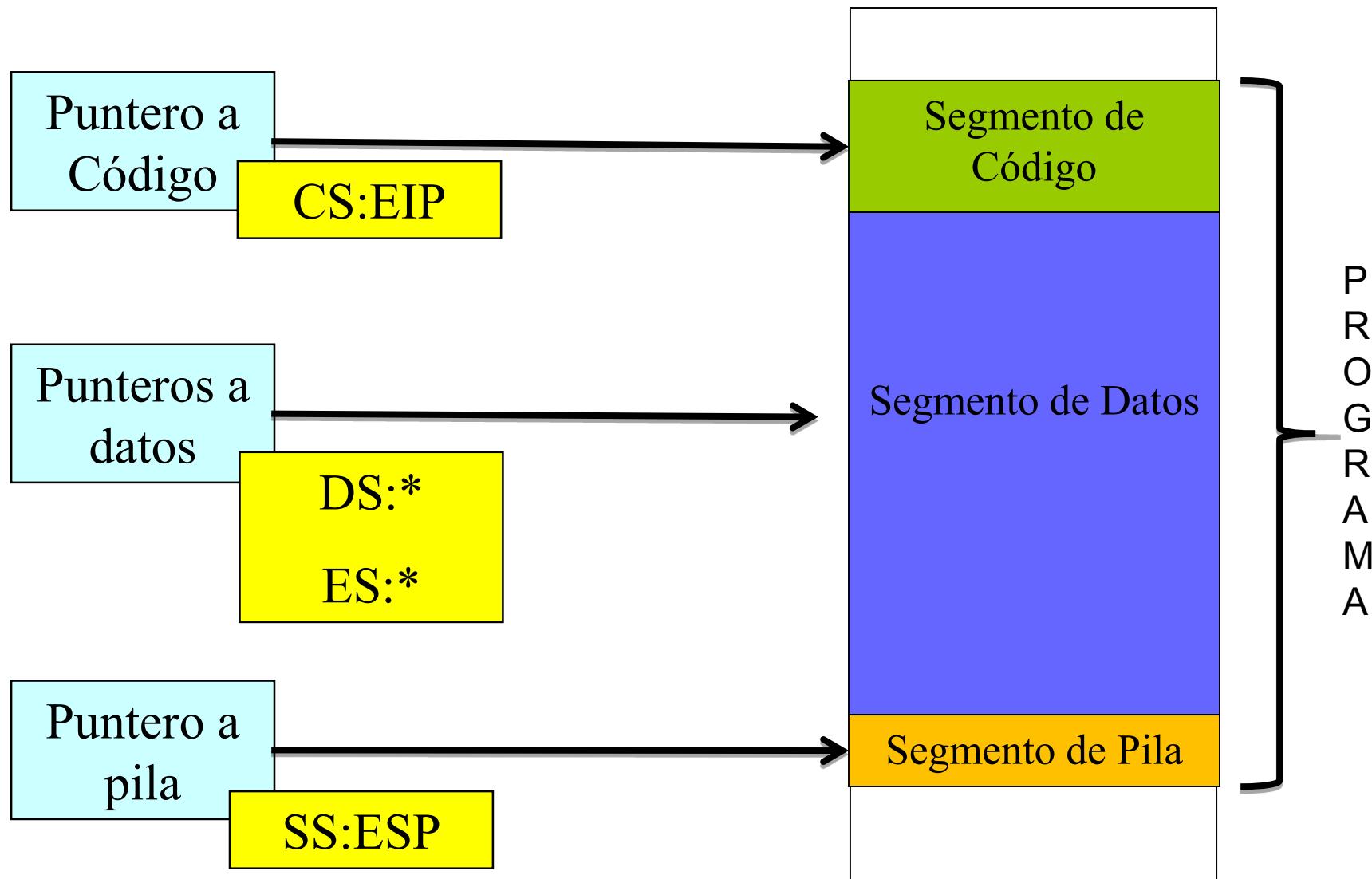
CS: Code Segment.

DS: Data Segment. (también ES, FS y GS)

SS: Stack Segment.

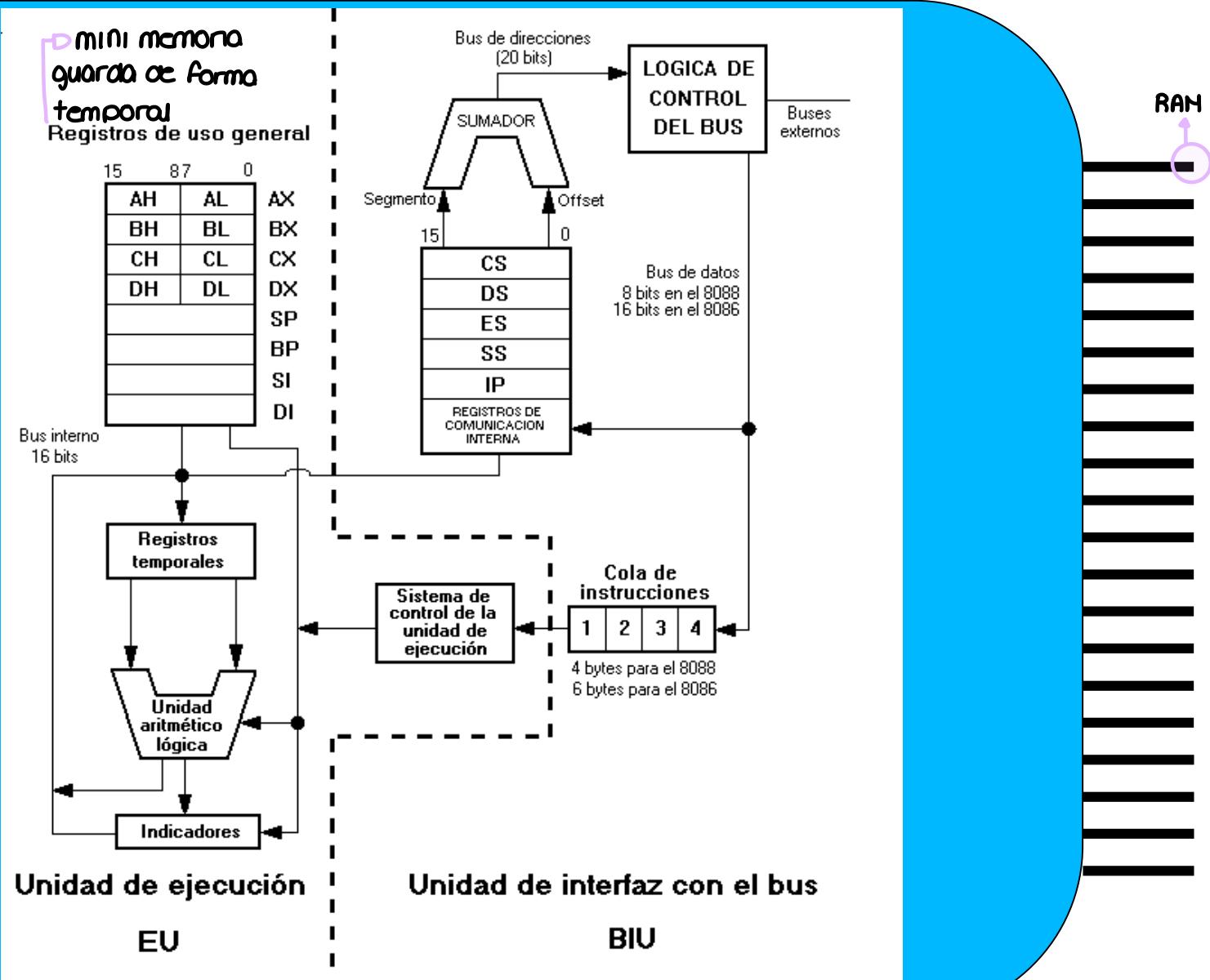
(Los registros de segmentos mantienen su tamaño en arquitectura de 32 y 64 bits)

# Programa en memoria

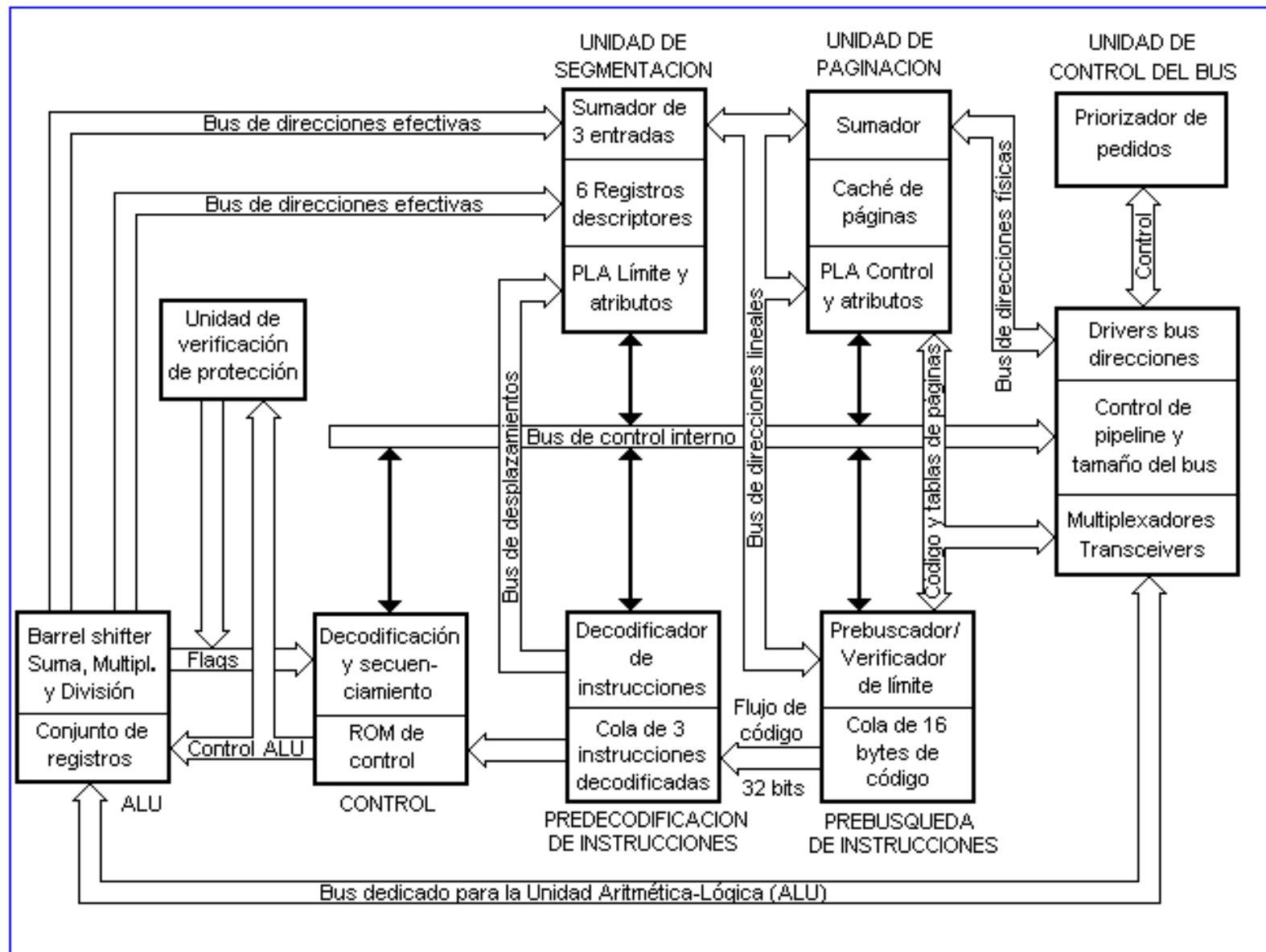


# Arquitectura de 8088/8086 (16 bits)

## Registros dentro del procesador



# Arquitectura de 80386



# Arquitectura de 80386

Los procesadores mas avanzados de hoy en día mantienen su arquitectura derivada del 80386.

Se aplica en estos procesadores el “Concepto de diseño superescalar.”, que consiste en agregar mas de una unidad de procesamiento para aumentar la capacidad del mismo.

Cumple con:

**Multitarea:** Existe un requisito importante para los sistemas operativos Multitarea que es tener espacio de memoria individual para cada tarea, y un espacio de memoria común para varias tareas

**Multiusuario:** Que mas de un usuario tenga acceso a la CPU, lo que genera mas tareas.

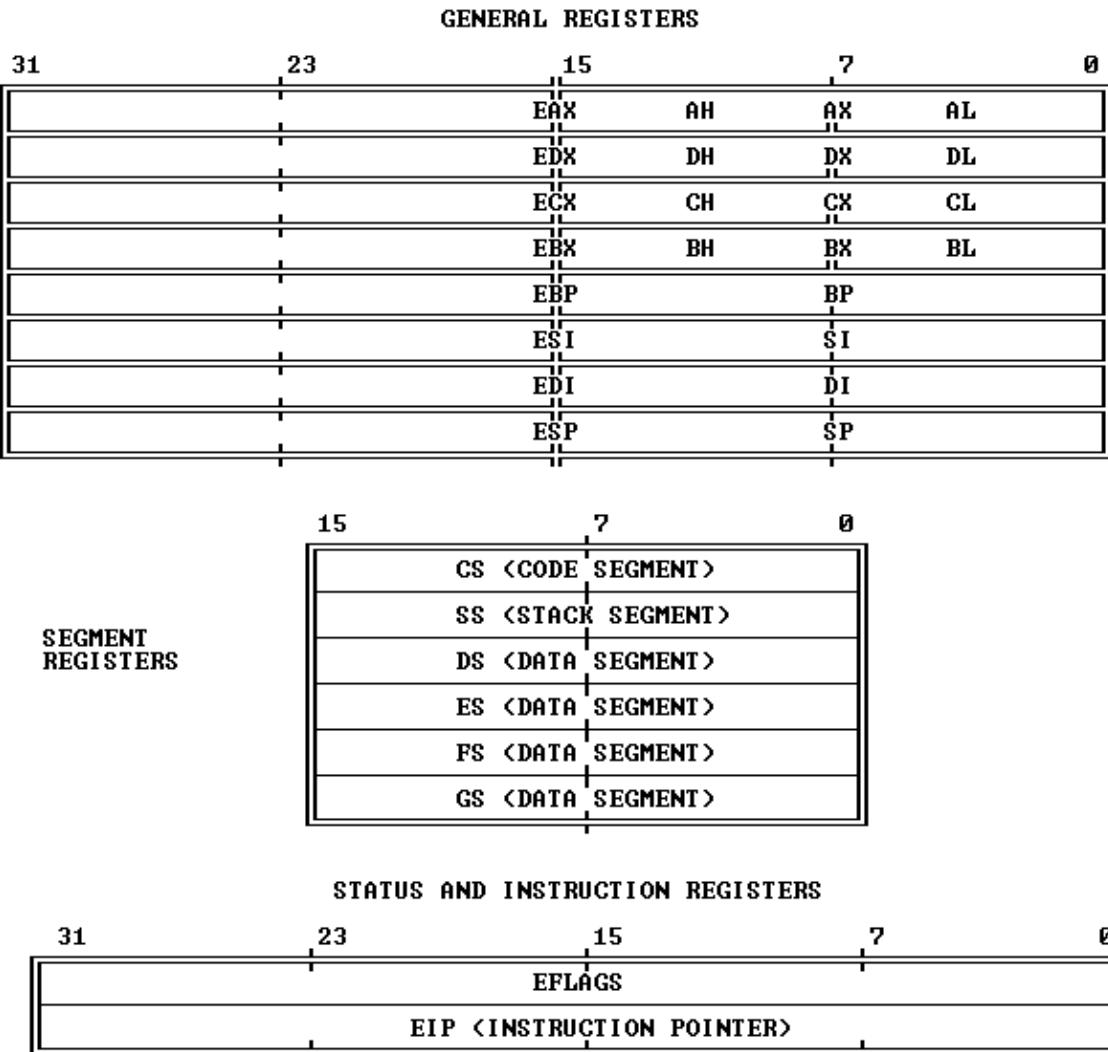
**Tiempo compartido:** El S.O asigna un tiempo para cada tarea (time slot).

**Tiempo Real:** La conmutación de tareas viene dada por acontecimientos externos.

**Sistema de protección:** Mínimo dos niveles, de Usuario y de Supervisor. Memoria virtual. Las memorias RAM o ROM siguen siendo caras si se las compara con los precios de los discos rígidos.

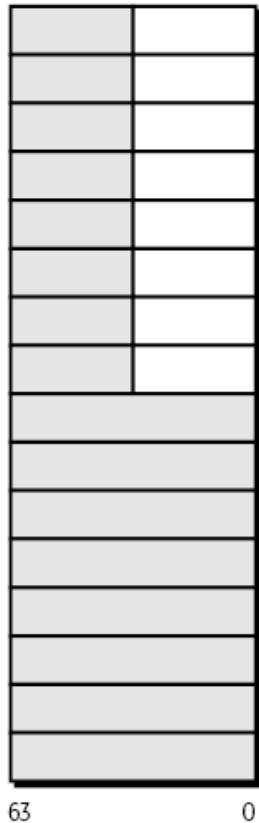
# Registros de 80386 (32 bits)

Figure 2-5. 80386 Applications Register Set

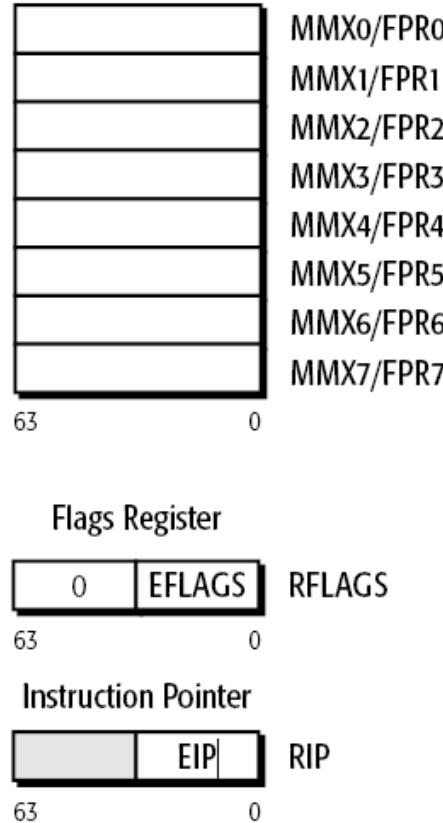


# Registros de 64 bits

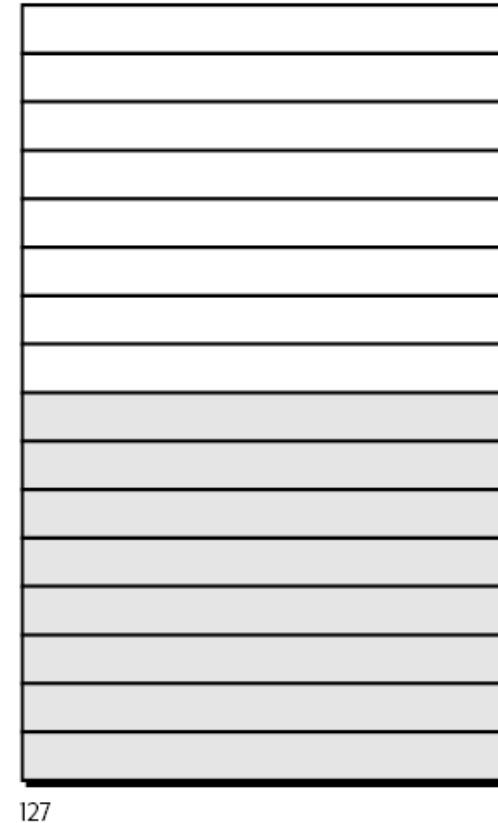
General-Purpose  
Registers (GPRs)



64-Bit Media and  
Floating-Point Registers



128-Bit Media  
Registers



Legacy x86 registers, supported in all modes



Register extensions, supported in 64-bit mode

Application-programming registers also include the 128-bit media control-and-status register and the x87 tag-word, control-word, and status-word registers



# Assembleur de Intel

# Objetivos de la clase

- ✓ Leer y escribir datos en memoria
- ✓ Realizar los primeros programas en ASM

# Assembler - Sintaxis

Existen varias sintaxis para ASM, las mas conocidas son:

- Intel
- AT & T

Veamos dos sentencias equivalentes con las dos sintaxis:

---

*mov EAX, 1*

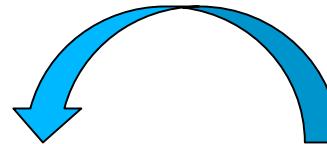
*(sintaxis Intel)*

*movl \$1, %eax*

*(sintaxis AT&T)*

⚠️ Tener en cuenta que el gcc por default genera salidas en sintaxis AT&T.

# Assembler - Sintaxis

*Destino*            *Origen*

***mov EAX, 1***

# Instrucciones

Flujo de bytes que interpretados por el procesador que realizan una acción

Instrucción: **add eax, 0x1**

Instrucción	contenido binario en mem.	contenido hexa en mem.
add eax, 0x1	1000 0011 1100 0000 0000 0001	83 c0 01

⇒ suma 0x1 a eax y lo guarda en eax

Existen instrucciones de:

1 byte    2 bytes    3 bytes    .....

ARM solo tiene instrucciones de 32 bits

# Registros de Intel

Ejemplos de uso con ASM:

mov ah, 23  
    ↳ decimal

mov bl, 99h

mov ax, 1234h

mov eax,12345678h

mov rax,12345678ABCDEF00h  
    ↓  
    64 bits

# Lectura de memoria

mov ax, [100h] ➤ registro de 16 bits ➤ copia dos posiciones  
copia 0012h en ax

0100h

0100h

mov ebx,[102h] ➤ copia 0011003Ah en ebx

0104h

mov cl,[109h] ➤ copia B8 en cl

0108h

mov ax,[bx]  
busca la posición del contenido en bx

010Ch

⇒ un byte en cada posición de memoria

D0	12	00	11
00	3A	07	FF
32	B8	C0	C1
74	7E	E2	AE

Ej. Si bx cota 77AAh ➤ es equivalente a mov ax,[77AAh]

Obs: distinto de mov ax,bx ➤ agarra el valor de bx y lo cargo en ax  
⇒ equivalente a mov ax,77AAh

⚠ Destino y fuente mismo tamaño ➤ mov ax,rbx ó mov rbx,ax NO compila

⚠ mov [ax],[bx] NO es válido

# Escritura en memoria

Mov [102h],eax

Mov [104h],bl

Mov [108], rcx

- ↳ 108 no es hexa
- ⇒ se pasa a hexa pero no aparece en la tabla

0100h

0104h

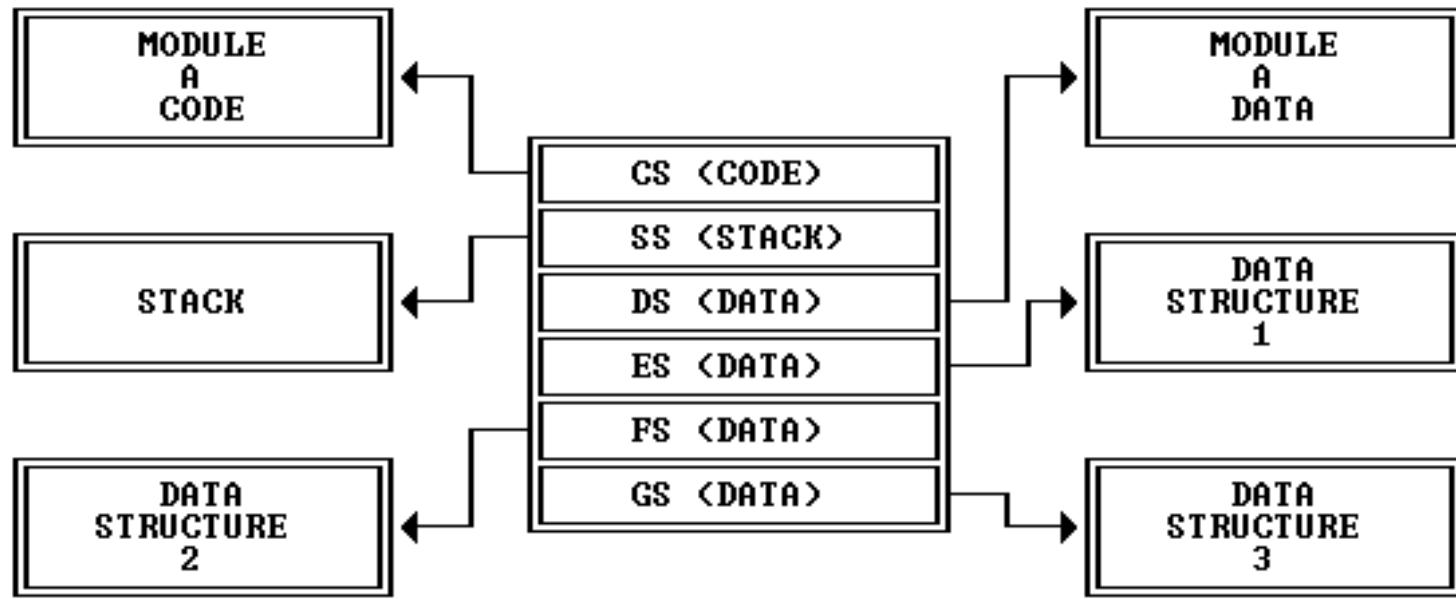
0108h

010Ch

			■	■
		■	■	

# Segmentación de memoria en 80386

Figure 2–6. Use of Memory Segmentation



Recordar pares de punteros  
para acceder a memoria

# Modos de direccionamiento

Como en otros procesadores tendremos la sintaxis general será:

*Instrucción destino , fuente*

## Direccionamiento inmediato

mov ax, 0ffffh

## Direccionamiento de registro

mov edx,eax

mov ah,al

# Modos de direccionamiento

## Direccionamiento directo o absoluto

▷ por default va al data segment

mov ax, [57D1h]

▷ nombre del segmento al que quiero acceder

mov ebx, es:[42c9h]

## Direccionamiento indirecto

mov cx, [bp]

mov es:[di],ax

## Direccionamiento con índice o indexado

mov cx, [bp+4]

mov es:[di+8],ax

# Recordatorio

NO existe el movimiento de datos de memoria a memoria en una sola instrucción:

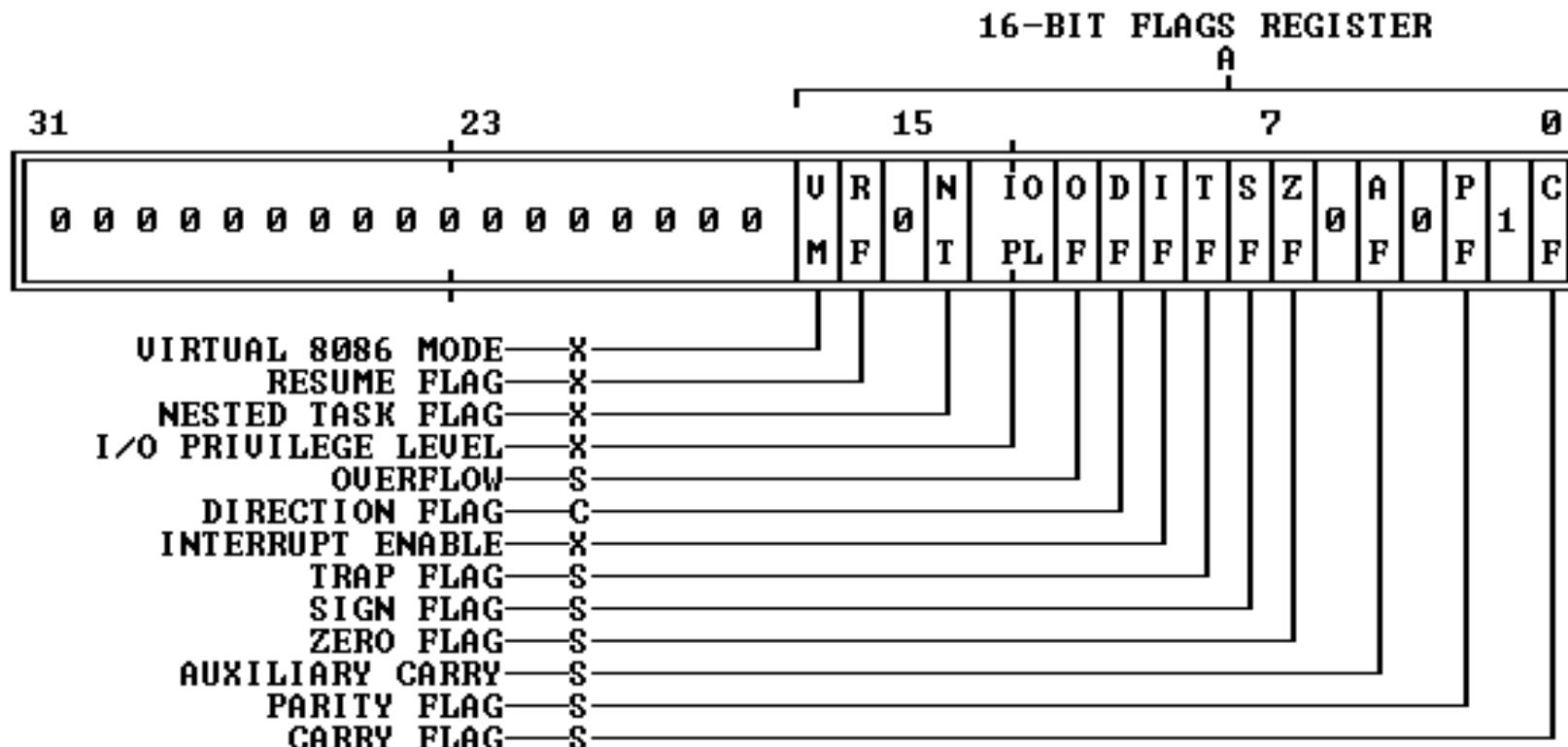
~~Mov [bx],[ax]~~

# Registro de Flags

⇒ lugar de decisión de los jumps

↳ bits que toman valores 1 y 0

Figure 2-8. EFLAGS Register



NOTE: 0 OR 1 INDICATES INTEL RESERVED. DO NOT DEFINE

# Assembler – Ejemplo 1 – teoej1.asm

```
section .text => código hasta ret  
global _start => rotulo _start global
```

\_start:

```
mov dx,0FFh  
mov bx,20h  
add dx,bx  
push dx => tomo el valor de dx y lo  
push 4     guardo en lo pilo  
pop cx => recuperar valor de lo pilo  
           y lo guardo en cx  
           => en cx guardo 4  
Ciclo:  
inc bx  
dec cx => si me da 0, no hace el jump  
jnz Ciclo  
jnz = jump not zero => hace jump si el valor anterior NO es 0  
mov eax,parametros  
     => guarda la dirección de memoria And esto parametros a cx  
mov AH,[parametros] => AH = 11h }  
mov BL,[parametros+1] => BL = 12h } parametros es dirección de memoria
```

add ah,bl => z3h en ah  
mov [salida],ah  
 => dirección de memoria  
ret 0 => termina  
 datos en las #  
 direcciones de memoria

```
section .data  
parametros } puntos db      11h,12h,13h  
salida      }         db      0 => z3h
```

=> tamaño no cambia a lo largo de la ejecución  
= segmento de datos

A Flags alterados por la ult instrucción => siempre poner la  
instrucción que queremos q altere antes del flag

A Hay distintos tipos de flags

A No sabemos la posición de salida y parametros

1 se puede no llamar add

# Assembler – Ejemplo 1

Para compilar en Linux 64 bits:

*nasm -f elf64 teoej1.asm -o teoej1.o*

Para linkeditar:

*ld teoej1.o -o teoej1*

Genera el archivo ejecutable **teoej1** como salida

Al abrirlo con Evans Debugger.....

# Assembler – Ejemplo 1 - DBG

teeoj1: No Analysis Found

→ 00000000:004000b0 66 ba ff 00	mov dx, 0xff	
00000000:004000b4 66 bb 14 00	mov bx, 0x14	
00000000:004000b8 66 01 da	add dx, bx	
00000000:004000bb 66 52	push dx	
00000000:004000bd 6a 04	push 4	
00000000:004000bf 66 59	pop cx	
00000000:004000c1 66 ff c3	inc bx	
00000000:004000c4 66 ff c9	dec cx	
00000000:004000c7 75 f8	jne tteoj1!ciclo	
00000000:004000c9 b8 d8 00 60 00	mov eax, 0x6000d8	
00000000:004000ce 00 dc	add ah, bl	
00000000:004000d0 88 24 25 db 00 60 00	mov [0x6000db], ah	
00000000:004000d7 00 11	add [rcx], dl	
00000000:004000d9 12 13	adc dl, [rbx]	
00000000:004000db 00 00	add [rax], al	
00000000:004000dd 2e 73 79	jae 0x400159	
00000000:004000e0 6d	insd [rdi], dx	
00000000:004000e1 74 61	je 0x400144	
00000000:004000e3 62	db 0x62	
00000000:004000e4 00 2e	add [rsi], ch	
00000000:004000e6 73 74	jae 0x40015c	
00000000:004000e8 72 74	jb 0x40015e	
.....	.....	

Registers

RAX 0000000000000000 orig: 0000000000000000	
RCX 0000000000000000	
RDX 0000000000000000	
RBX 0000000000000000	
RSP 00007ffd4373b60	
RBP 0000000000000000	
RSI 0000000000000000	
RDI 0000000000000000	
R8 0000000000000000	
R9 0000000000000000	
R10 0000000000000000	
R11 0000000000000000	
R12 0000000000000000	
R13 0000000000000000	
R14 0000000000000000	
R15 0000000000000000	
RIP 00000000004000b0 </home/srv/edb/edb	
C 0 ES 0000	
P 0 CS 0033	
A 0 SS 002b	
Z 0 DS 0000	
S 0 FS 0000 (0000000000000000)	
T 0 GS 0000 (0000000000000000)	

apunta a lo prox  
instrucion a ejecutar

dx = 0x0000

Data Dump

+ 0x00000000000600000-0x00000000000601000	.....
00000000:006000d8 11 12 13 00 00 2e 73 79 6d 74 61 62 00 2e 73 74	.....symtab..st
00000000:006000e8 72 74 61 62 00 2e 73 68 73 74 72 74 61 62 00 2e	rtab..shstrtab..
00000000:006000f8 74 65 78 74 00 2e 64 61 74 61 00 00 00 00 00 00	text..data.....
00000000:00600108 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000000:00600118 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000000:00600128 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000000:00600138 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000000:00600148 1b 00 00 00 01 00 00 06 00 00 00 00 00 00 00 00	.....
00000000:00600158 b0 00 40 00 00 00 00 00 b0 00 00 00 00 00 00 00	.....
00000000:00600168 27 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	'.....
00000000:00600178 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000000:00600188 21 00 00 00 01 00 00 03 00 00 00 00 00 00 00 00	!
00000000:00600198 d8 00 60 00 00 00 00 00 d8 00 00 00 00 00 00 00	.....

Stack

00007ffd:b4373b60 0000000000000001	.....
00007ffd:b4373b68 00007ffd:b4374f6a	j07@..... ASCII "tteoj1"
00007ffd:b4373b70 0000000000000000	.....
00007ffd:b4373b78 00007ffd:b4374f71	q07@..... ASCII "XDG_VTNR=7"
00007ffd:b4373b80 00007ffd:b4374f7c	[07@..... ASCII "LC_PAPER=es_AR.UTF-8"
00007ffd:b4373b88 00007ffd:b4374f91	[07@..... ASCII "LC_ADDRESS=es_AR.UTF-8"
00007ffd:b4373b90 00007ffd:b4374fa8	[07@..... ASCII "XDG_SESSION_ID=c3"
00007ffd:b4373b98 00007ffd:b4374fba	[07@..... ASCII "rvm_bin_path=/home/srv/.rvm/bin"
00007ffd:b4373ba0 00007ffd:b4374fda	[07@..... ASCII "SELINUX_INIT=YES"
00007ffd:b4373ba8 00007ffd:b4374feb	[07@..... ASCII "CLUTTER_IM_MODULE=xim"
00007ffd:b4373bb0 00007ffd:b4375001	[P7@..... ASCII "XDG_GREETER_DATA_DIR=/var/lib/lightdm-data,
00007ffd:b4373bb8 00007ffd:b4375030	[07@..... ASCII "LC_MONETARY=es_AR.UTF-8"
00007ffd:b4373bc0 00007ffd:b4375048	[HP7@..... ASCII "GIO_LAUNCHED_DESKTOP_FILE_PID=4362"
00007ffd:b4373bc8 00007ffd:b437506b	[K7@..... ASCII "SESSION=ubuntu"
00007ffd:b4373bd0 00007ffd:b437507a	[zP7@..... ASCII "GEM_HOME=/home/srv/.rvm/gems/ruby-2.3.1"

Stack Debugger Error Console

paused

# Assembler – Ejemplo 1 - Código

DIRECCIONES DE MEMORIA	CÓDIGO MAQUINA (en hexa)	INSTRUCCIONES
00000000:004000b0	66 ba ff 00	mov dx, 0xff
00000000:004000b4	66 bb 14 00	mov bx, 0x14
00000000:004000b8	66 01 da	add dx, bx
00000000:004000bb	66 52	push dx
00000000:004000bd	6a 04	push 4
00000000:004000bf	66 59	pop cx
00000000:004000c1	66 ff c3	inc bx
00000000:004000c4	66 ff c9	dec cx
00000000:004000c7	75 f8	jne teoej1!ciclo
00000000:004000c9	b8 d8 00 60 00	mov eax, 0x6000d8
00000000:004000ce	00 dc	add ah, bl
00000000:004000d0	88 24 25 db 00 60 00	mov [0x6000db], ah
00000000:004000d7	00 11	add [rcx], dl
00000000:004000d9	12 13	adc dl, [rbx]
00000000:004000db	00 00	add [rax], al
00000000:004000dd	2e 73 79	jae 0x400159
00000000:004000e0	6d	insd [rdi], dx
00000000:004000e1	74 61	je 0x400144
00000000:004000e3	62	db 0x62
00000000:004000e4	00 2e	add [rsi], ch
00000000:004000e6	73 74	jae 0x40015c
00000000:004000e8	72 74	jb 0x40015e
00000000:004000f0	c3	ret

dx = 0x0000

# Assembler – Ejemplo 1 - Datos

Data Dump																
+ 0x000000000000600000-0x000000000000601000	Sabemos que es salida porque sera como "parametros + 3"															
00000000:006000d8	11	12	13	00	00	2e	73	79	6d	74	61	62	00	2e	73	74
00000000:006000e8	72	74	61	62	00	2e	73	68	73	74	72	74	61	62	00	2e
00000000:006000f8	74	65	78	74	00	2e	64	61	74	61	00	00	00	00	00	00
00000000:00600108	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000000:00600118	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000000:00600128	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000000:00600138	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000000:00600148	1b	00	00	00	01	00	00	06	00	00	00	00	00	00	00	00
00000000:00600158	b0	00	40	00	00	00	00	b0	00	00	00	00	00	00	00	00
00000000:00600168	27	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000000:00600178	10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000000:00600188	21	00	00	00	01	00	00	03	00	00	00	00	00	00	00	!
00000000:00600198	d8	00	60	00	00	00	00	d8	00	00	00	00	00	00	00	□

direcciones de memoria del segmento de datos

contenido de las posiciones de memoria

hexa interpretado en ascii

- parametros = 006000d8  $\Rightarrow$  mov AH,[parametros] = mov AH,[006000d8h]  $\Rightarrow$  a AH le cargo 11h
- salida = 006000dB

no hay representación en ASCII  
A grafica

# Assembler – Ej 1 – TP 2

```
section .text
GLOBAL _start

_start:
    mov ecx, cadena → guarda la dirección de mem ; Puntero a la cadena
    mov edx, longitud → contiene a ecx ; Largo de la cadena
    mov ebx, 1 ; FileDescriptor (STDOUT)
    mov eax, 4 ; ID del Syscall WRITE
    int 80h → ejecuta la system call ; Ejecución de la llamada

    mov eax, 1 ; ID del Syscall EXIT
    mov ebx, 0 ; Valor de Retorno
    int 80h ; Ejecución de la llamada

section .data
cadena db "Hola Mundo!!", 10 ;"Hola Mundo!!\n"
longitud equ $-cadena

section .bss
placeholder resb 10
```

# Assembler – Linkeditar con gcc

Podemos usar el GCC para linkeditar:

Pero la función **start** se debe llamar **main**

☞ GCC hace que debamos declarar la función main. Si no el gcc no sabe dónde empieza el programa

*section .text*

*GLOBAL main*

**main:**

*mov ecx, cadena ; Puntero a la cadena*

*mov edx, longitud ; Largo de la cadena*

.....

# Assembler – Linkeditar con gcc

Para compilar: *nasm -f elf64 teoejlforc.asm -o teoejlforc.o*

Para linkeditar : *gcc teoejlforc.o -o teoejlforc*

Los archivos ejecutables tienen distintos tamaños.

*¿Por que?*

► tamaño mas grande si linkeditado con el gcc

```
-rwxrwxr-x 1 srv srv 8572 ago 13 10:06 teoejlforc
-rw-rw-r-- 1 srv srv  928 ago 13 09:58 teoejlforc.o
-rw-rw-r-- 1 srv srv   268 ago 13 09:58 teoejlforc.asm
-rw-rw-r-- 1 srv srv  928 ago 12 18:49 teoejl.o
-rwxrwxr-x 1 srv srv 1003 ago 12 18:18 teoejl
-rw-rw-r-- 1 srv srv   272 ago 12 18:17 teoejl.asm
```

linkeditar con ld = ejecutable + chico que linkeditar con gcc

¿Que agrega el gcc? Header + syscalls ➔ ejecuta syscalls

# Assembler – Linkeditar con gcc

Al abrir el ejecutable **teoej1forc** con el debugger vemos que tiene otro código al comienzo. ¿Donde está mi código ASM?

00007f15:68faf2d0	48 89 e7	mov rdi, rsp	NO es la primera linea de codigo
00007f15:68faf2d3	e8 68 37 00 00	call ld-2.19.so!_dl_start	
00007f15:68faf2d8	49 89 c4	mov r12, rax	
00007f15:68faf2db	8b 05 17 1b 22 00	mov eax, [rel 0x7f15691d0df8]	
00007f15:68faf2e1	5a	pop rdx	
00007f15:68faf2e2	48 8d 24 c4	lea rsp, [rsp+rax*8]	
00007f15:68faf2e6	29 c2	sub edx, eax	
00007f15:68faf2e8	52	push rdx	
00007f15:68faf2e9	48 89 d6	mov rsi, rdx	
00007f15:68faf2ec	49 89 e5	mov r13, rsp	
00007f15:68faf2ef	48 83 e4 f0	and rsp, 0xfffffffffffffff0	
00007f15:68faf2f3	48 8b 3d 66 1d 22 00	mov rdi, [rel 0x7f15691d1060]	
00007f15:68faf2fa	49 8d 4c d5 10	lea rcx, [r13+rdx*8+0x10]	
00007f15:68faf2ff	49 8d 55 08	lea rdx, [r13+8]	
00007f15:68faf303	31 ed	xor ebp, ebp	
00007f15:68faf305	e8 76 ee 00 00	call ld-2.19.so!_dl_init_internal	
00007f15:68faf30a	48 8d 15 1f f2 00 00	lea rdx, [rel 0x7f1568fbe530]	
00007f15:68faf311	4c 89 ec	mov rsp, r13	
00007f15:68faf314	41 ff e4	jmp r12	
00007f15:68faf317	66 0f 1f 84 00 00 00 0...	nop word [rax+rax]	
00007f15:68faf320	48 8d 05 d9 2c 22 00	lea rax, [rel 0x7f15691d2000]	
00007f15:68faf327	c3	ret	
00007f15:68faf328	cc cc cc cc cc cc cc		

Si corremos el código en un aparato con poco memoria ➔ NO conviene gcc.

# Modos del 80386

⇒ dos maneras para trabajar la memoria

Históricamente desde los microprocesadores 8088 y 8086 se utilizó el direccionamiento a memoria llamado **Modo Real**, pero no tenía las características necesarias para los sistemas operativos multitarea y multiusuario.

En el 80386, se consolidó el **Modo Protegido**.

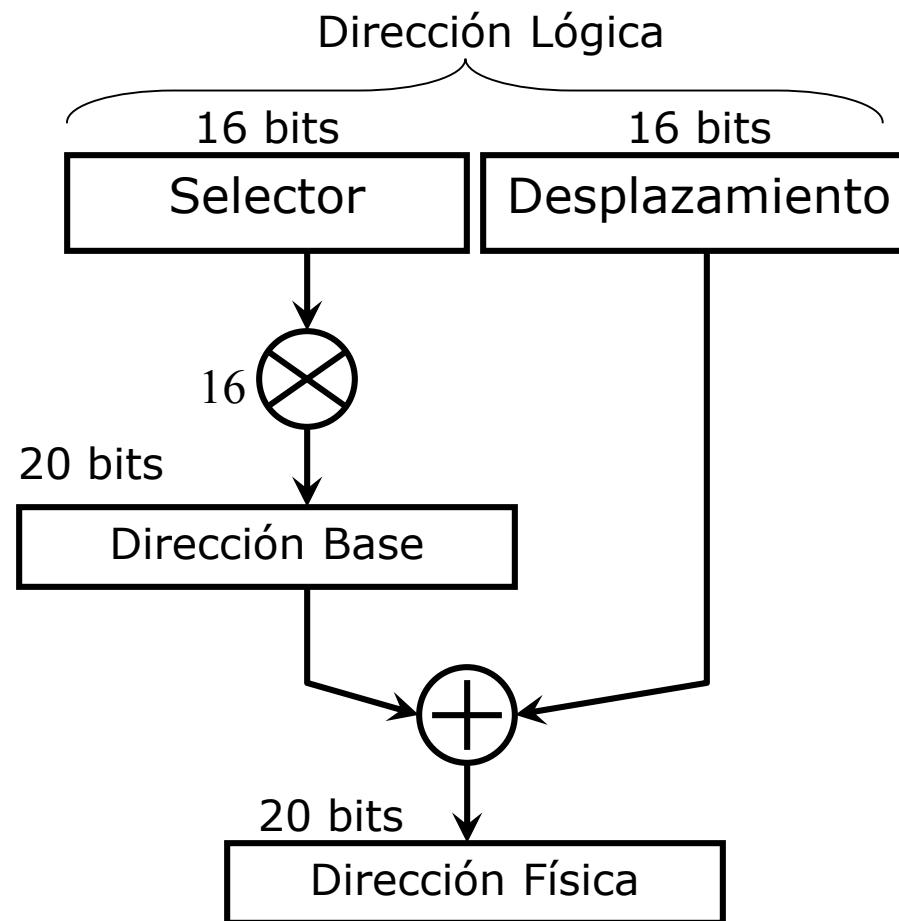
Pero se mantuvo como Modo de inicio del microprocesador al Modo Real para mantener la compatibilidad.

- Modo real = memoria como sucesión de bytes sin restricciones
- Modo protegido ⇒ NO se puede acceder directamente a mem física (pasos intermedios)

Cuando se prende la computadora en real ⇒ se debe pasar a protegido

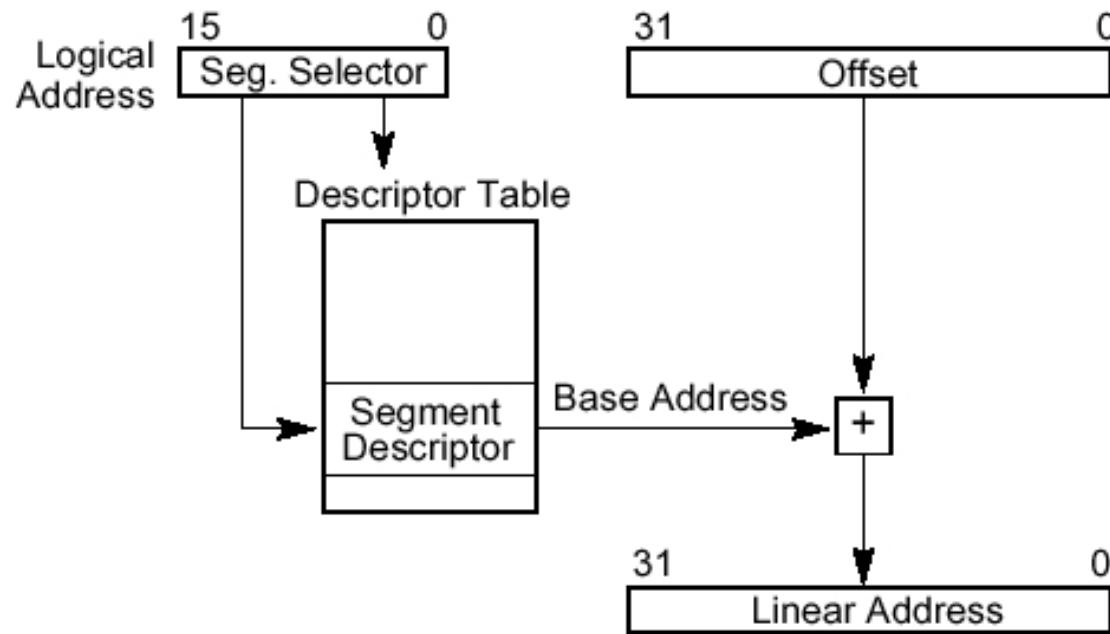
⇒ lo primero que hay que hacer es poner el procesador en modo protegido

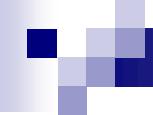
# Direccionamiento en Modo Real



# Direccionamiento en Modo Protegido

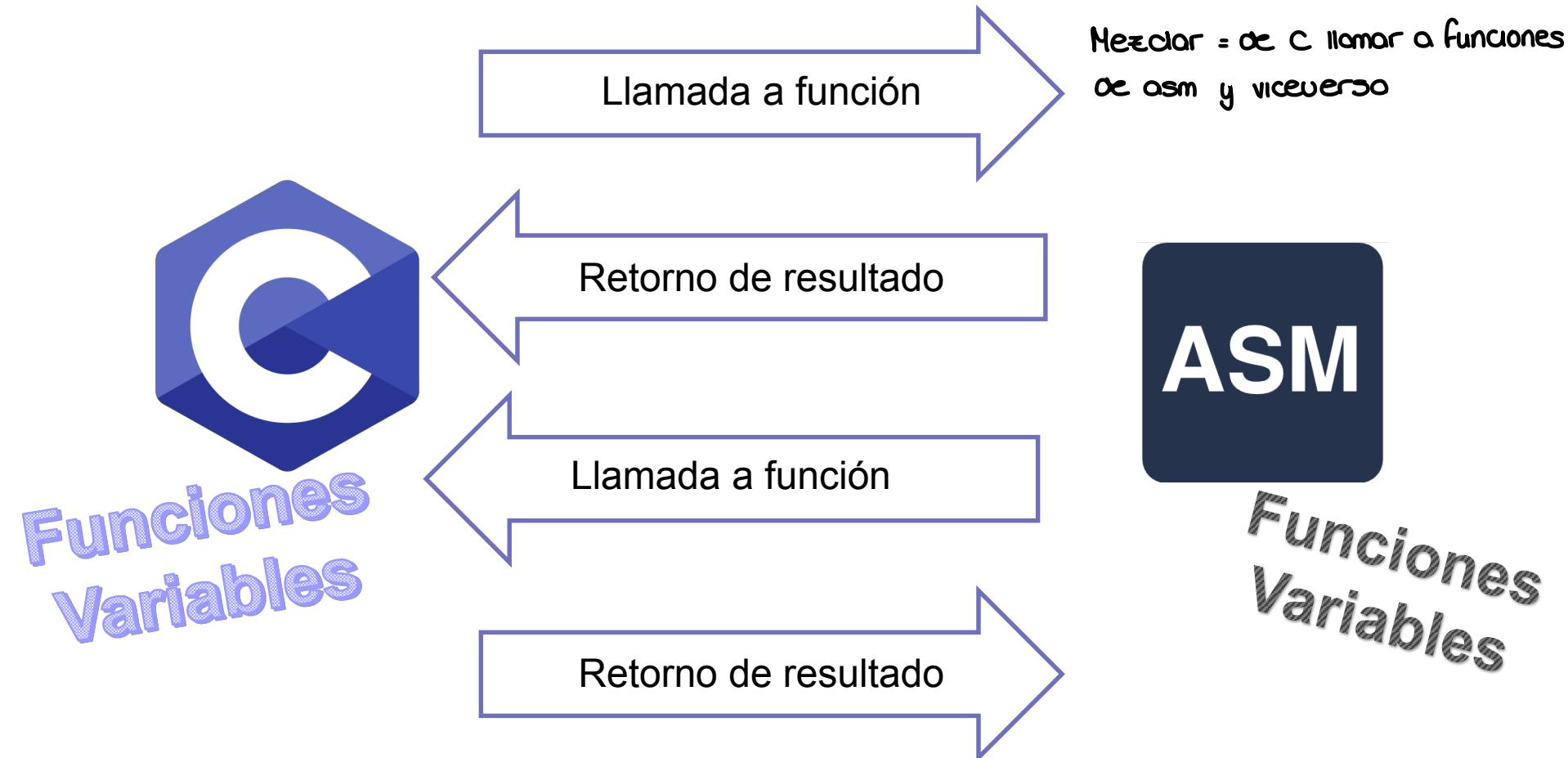
⇨ Se debe pasar a través de  
controles para llegar a memoria física





# ASM y C

# Mezcla de lenguajes en un ejecutable

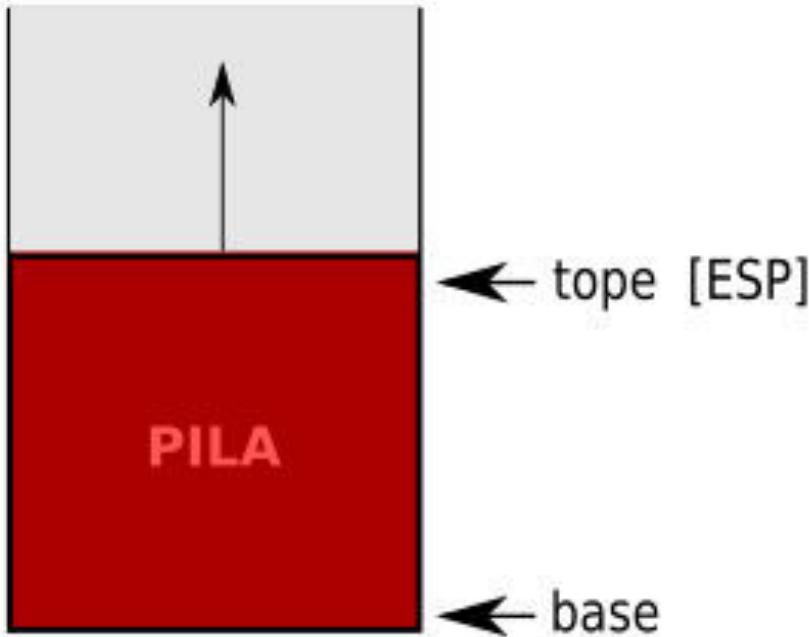


**¿Como se soluciona?**

**Con la pila y los registros**

# Repaso de Pila

direcciones  
0x....0000



El *stack pointer register* o *extended stack pointer*) apunta al tope de la pila, es decir al último elemento almacenado en ella.

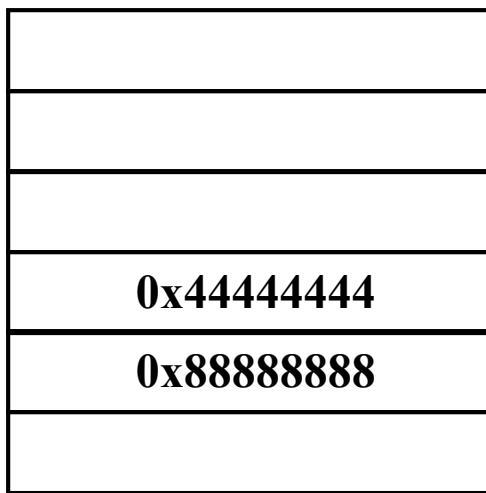
Cuando se almacena un nuevo valor en la pila con **PUSH** el valor del puntero se actualiza para siempre apuntar al tope de la pila.\*

↳ **esp decremento**

\*<https://fundacion-sadosky.github.io/guia-escritura-exploits/buffer-overflow/1-introduccion.html>

# Repaso de Pila - Push

ESP



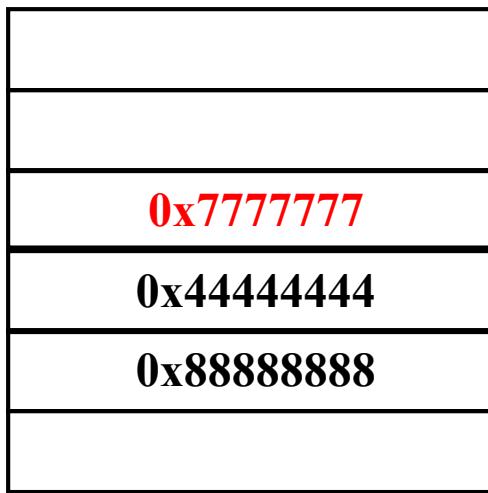
00000000h

80000004h

FFFFFFFFFFh

Cuando se ejecuta una instrucción **PUSH**, el procesador decremente el registro ESP ó RSP y guarda el valor en el stack

ESP



00000000h

80000000h

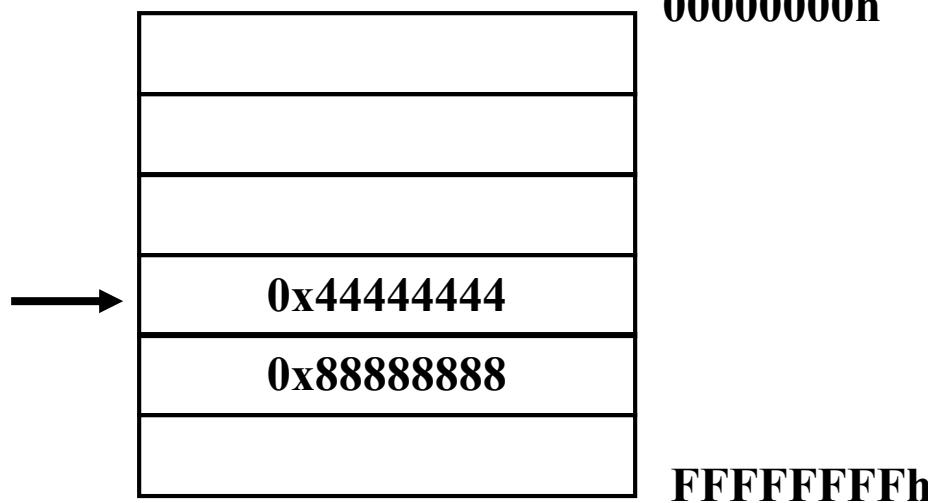
80000004h

FFFFFFFFFFh

subc de 4 xq pusheo 4 bytes  
⇒ ocupa de del tamaño de lo que puseo

# Repaso de Pila - Pop

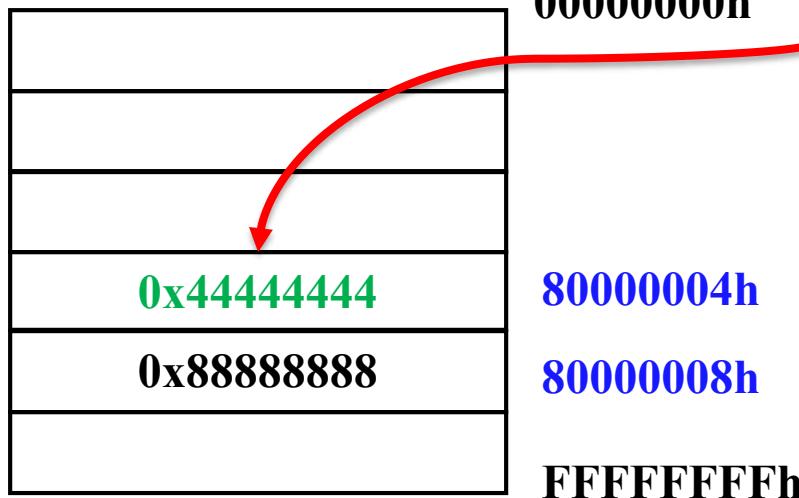
ESP



Cuando se ejecuta una instrucción POP, toma el contenido de la dirección ( en este caso el contenido es 0x44444444 ) y luego incrementa el registro ESP ó RSP.

⚠ Los destinos tienen que tener el mismo tamaño => difícil manejar datos de distintos tamaños.

ESP



¿Qué sucede con este dato?

Ej. pop ccx

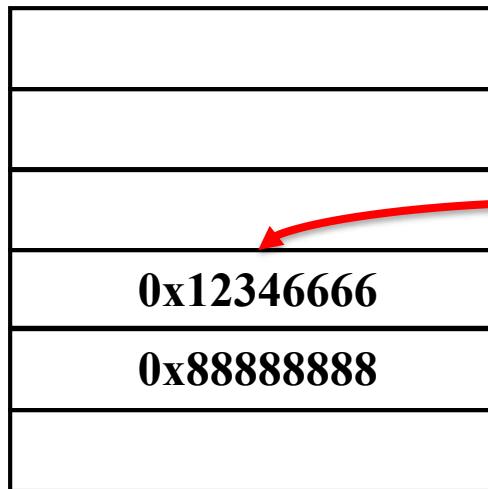
⇒ toma el contenido en 80000004h  
y lo pone en ccx  
⇒ ccx = 44444444h

# Instrucción RET

# Instrucción RET

ESP

👉 el SP tiene que apuntar a la sig instrucción cuando hago el ret



00000000h

BEBE0004h

FFFFFFFFFFh

*Mov ax, 23h*

*Cmp ax, 0*

*Jne fin:*

.....

...

*Fin:*

*ret*

Cuando se ejecuta una instrucción **RET**, el procesador toma el contenido de los apuntado por ESP y salta a esa posición de memoria

En este caso salta a la dirección de memoria 12346666h.

Es equivalente a hacer un **JMP** a esa dirección.

👉 llamado por el BIOS

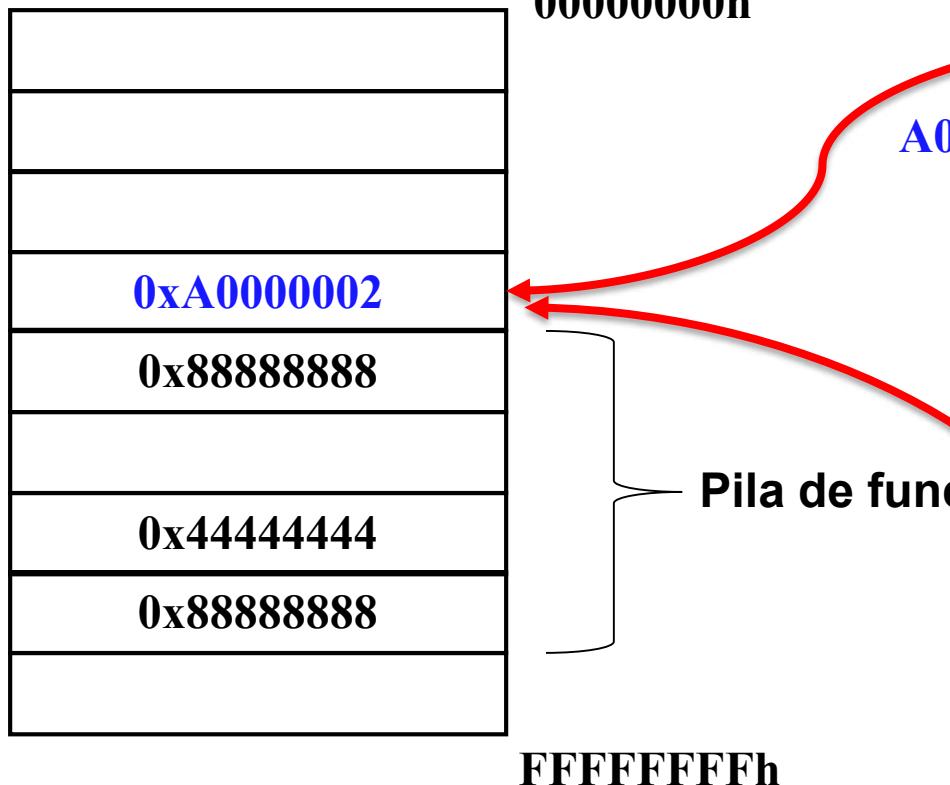
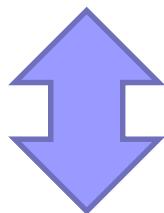
Obs: el primer proceso es el SO ➡ cdo se termina de ejecutar un programa, el RET vuelve al SO

👉 ejecuta la sig instrucción (Ej. otro programa)

# Instrucción CALL

# Instrucción RET

ESP



La instrucción CALL guarda en la pila la próxima instrucción a ejecutarse ó también llamada dirección de retorno. La función llamada (func1) tiene que dejar la pila sin modificar antes de terminar, así RET puede volver correctamente. ➔ porque vuelve al SP

Mov bx, 44h

Call func1

Add bx,ax

...

...

...

....

func1:

Push.....

Pop .....

Call .....

ret

➔ la función usa la pila  
PERO tiene que asegurarse  
que el SP apunte donde  
apuntaba antes del JMP para  
que cuando haga el RET pueda  
seguir correctamente

# Análisis de C



Entendamos como funciona el compilador de C para poder mezclarlo con ASM

## Pasaje de parámetros en funciones

```
int funcion_en_C ( int var1, char var2, int *var3 );
```

- Por Valor
- Por Referencia

# Pasaje de argumentos en C

- Según la arquitectura el compilador pasa de manera diferentes **los argumentos de las funciones**
- Arquitectura de **32 bits**
  - Se pasan por la **pila**
- Arquitectura de **64 bits**
  - Se pasan primero por **registros** y luego por la **pila**
    - ➡ Si no alcanzan los registros se utiliza la pila

# Pasaje de argumentos por registros

- Para pasar argumentos se usan los registros RDI, RSI, RDX, R10, R8, R9.
- Si la función necesita más parámetros se usa la pila
- Para punto flotante (float, double), xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6, xmm7

# Pasaje de argumentos por registros

**Registros a preservar:**

**%rbp**

**%rsp**

**%rbx**

**%r12**

**%r13**

**%r15**

Estos registros pertenecen a la función llamadora y deben mantener su valor al terminar la función

➡ NO deben ser tocados por las funciones

# Pasaje de argumentos por registros

- Según el tipo de dato que se quiere pasar se usan diferentes registros.\*
- Si los argumentos no entran en los registros se usa la pila pasando de derecha a izquierda.

Ej. Func1 var 1, var 2, var 3)

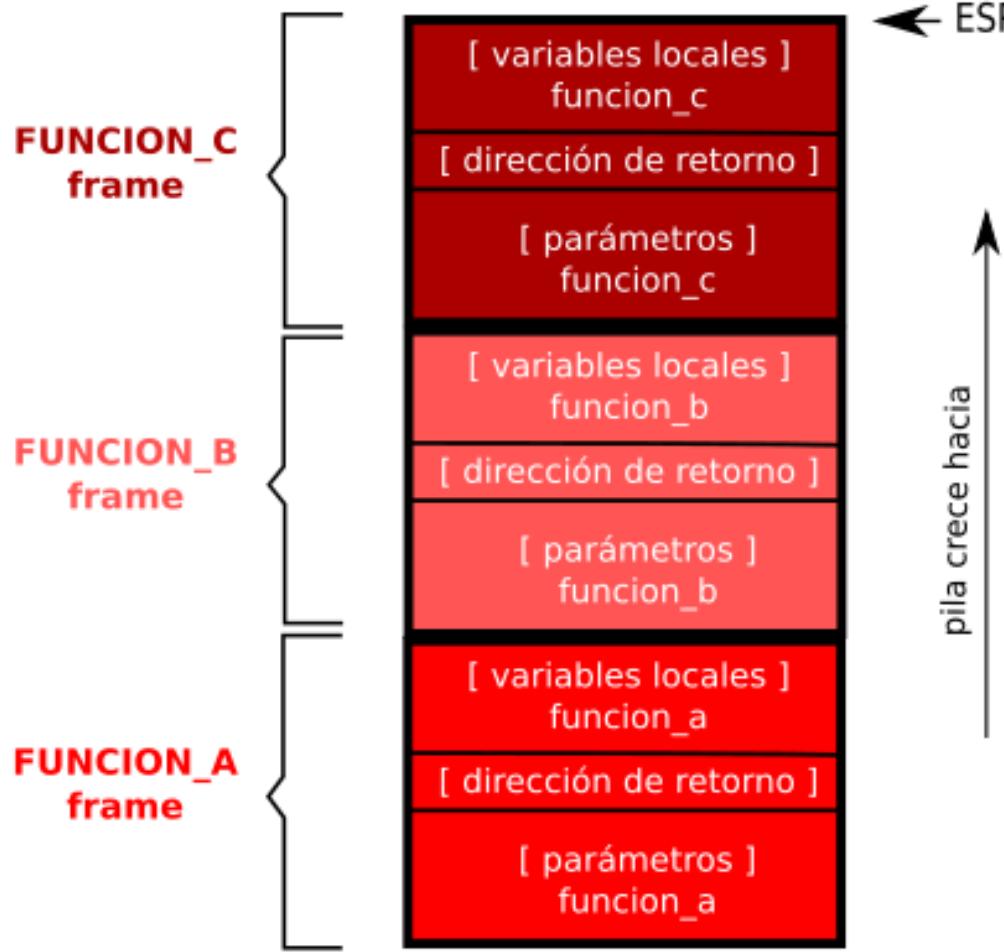
⇒ se ponen var 3, var 2 y finalmente var 1

**\*Igual a lo que vimos en las llamadas a sistemas con INT 80 ó SYSCALL**

# Manejo de la pila en C

- La pila se utiliza para pasar parámetros entre funciones.
- Al llamar a una función, ésta utiliza la pila para sus propias instrucciones PUSH y POP, por lo tanto modifica el registro ESP. ↳ es inevitable usar el SP porque las funciones se van anidando  
    ↳ backup del Stack Pointer
- Se utiliza el registro EBP para acceder a los parámetros que puede haber en la pila o a las variables locales, de esta manera no se modifica el registro ESP.

# Llamados a funciones con pila



- En la arquitectura x86, en el llamado a funciones la pila juega un rol fundamental.
- En este espacio de memoria se almacenan las variables locales de la función llamada, sus argumentos y su dirección de retorno.
- Aparece el concepto de **frame**

# Convenciones en C

## Parámetros de una función:

Por ejemplo:

***funcion\_en\_C ( param\_a, param\_b, param\_c );***

- **1. Parámetros**
- **2. Call** → guarda la dirección de retorno
- **3. Función**

Los parámetros se pushean en el stack de derecha a izquierda.

En este caso primero el param\_c luego el param\_b y luego el param\_a.

## Llamada a la función:

Se ejecuta la instrucción CALL que guarda en el stack el EIP para el retorno y ejecuta un JMP al primer byte de la función.

# Convenciones en C

## Resguardo y actualización de EBP (armado de stack frame)

Una vez en la nueva función se resguarda el valor anterior de EBP. Y luego se le asigna el valor actual del registro ESP. De esta manera se puede utilizar el registro EBP para acceder a los parámetros que quedaron en la pila.

Armado  
de stack  
frame

→ inicio de función

**Push ebp**  
**Mov ebp,esp**

Desarmado  
de stack  
frame

→ fin de función

**Mov esp, ebp**  
**Pop ebp**

## Valores a retornar

Si el valor es menor a 32 bits se retorna en EAX.

- Si es mayor retorna la parte alta en EDX y la parte baja en EAX. → 32 en cada
- Si es un dato mas complejo ( ej. Estructura de datos ) retorna un puntero formado por EDX:EAX

→ El Stackframe me ayuda a dejar la pila como estaba

# Llamada de ASM a C

; Puts.asm

; Programa que imprime utilizando puts

global main

extern puts

section .data

mensaje db 'Utilizando puts', 0Ah, 0

section .text

main:

push ebp

mov ebp,esp ; genera stack frame

push dword mensaje ; parametro  
tamaño del pun-  
tero que estamos para puts

call puts pasando ; llamada

pop eax ; saca el  
parametro de la pila

no es necesario si  
dsp esta el  
desarmado

mov esp,ebp

pop ebp ; destruye stack frame

ret

# Llamada de C a ASM

```
// cyasm1.c
#include <stdio.h>
extern unsigned int siete( void );
int main(void)
{
    printf("Devuelve el numero siete = %d\n", siete() );
    return 0;
}
```

porque valor  
de retorno en eax

```
; cyasm1_1.asm
[GLOBAL siete]
[SECTION .text]
siete:
    push    ebp
    mov     ebp,esp
    mov     eax,7
    mov     esp,ebp
    pop     ebp
    ret
```

⇒ no hace falta el armado  
y desarmado de stack  
frame porque la función no  
toca el stack

# Inline Assembler

⇒ no lo vamos a usar

```
int main(void)
{
    __asm__ ("movl $0x12345678, %eax");
}
```

```
int main (void )
{
    __asm__ ( "movl %eax, %ebx\n\t"
              "movl $56, %esi\n\t"
              "movb %ah, (%ebx)");
}
```

# Salidas en ASM

- Tenemos dos formas de ver el código C convertido en ASM
  - Compilar con “-S”  salido en asm del código en c
  - Utilizar GDB

# Ejemplo de Salida en ASM

```
/* asmycl.c */  
  
#include <stdio.h>  
  
#include <stdlib.h>  
  
int main ()  
{ int numero=10;  
    printf("Numero vale = %d", numero );  
    exit(0);  
}
```

```
gcc -S -masm=intel asmycl.c
```

# Ejemplo de Salida en ASM (32 bits)

```
.file "asmvc1.c"
.section .rodata ro read only data
.LC0: rotulo que apunta al string
.string "Numero vale = %d"

.text
.align 2
.globl main
.type main,@function

① main:
    ② push ebp
    ③ mov ebp,esp
    ④ sub esp,8 reservo lugar en la pila para la variable numero
    ⑤ and esp,-16
        no cambia pila
```

```
⑥ mov eax,0 no cambia pila
⑦ sub esp,eax no cambia pila
⑧ mov [ebp-4], 10
⑨ sub esp,8
⑩ push [ebp-4] pushca un int (no el puntero)
⑪ push .LC0
⑫ call printf los dos argumentos que tienen que ser pasados al printf
⑬ add esp,16
⑭ sub esp,12
⑮ push 0
⑯ call exit no sabemos como esta implementado el printf PERO cuando vuelve apunta a donde estaba el esp antes del call el exit recibe 0 como paron
```

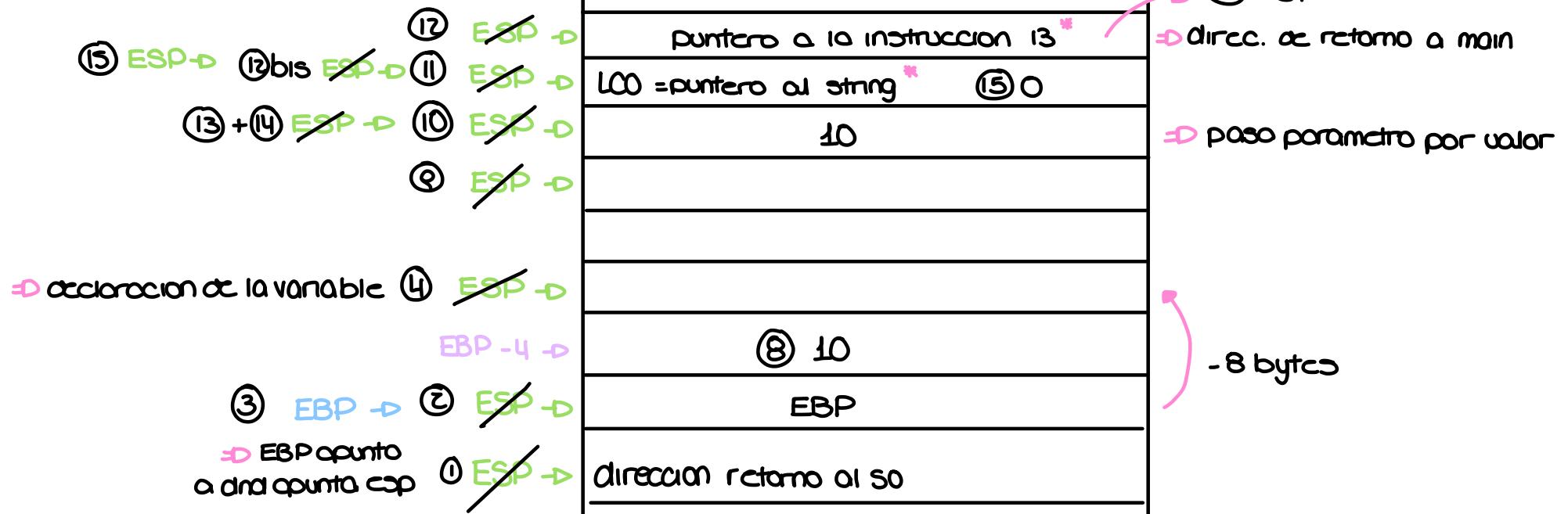
.Lfe1: *no hay desarmado porque lo hace exit*

.size main,.Lfe1-main

.ident "GCC: (GNU) 3.2 20020903 "

El gcc entrega su primera version  $\Rightarrow$  hay  
instrucciones de mas.  
Se puede usar -O para optimizar.

# RAM



# Análisis con EDB

Analicemos el programa anterior

Compilamos en 32 bits

```
gcc -m32 ejteo1.c -o ejteo1_32
```

Compilamos en 64 bits

```
gcc ejteo1.c -o ejteo1_64
```

¿Como se pasan los parámetros en cada arquitectura ?  
¿ En que orden?

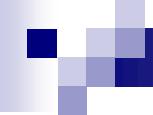
# Ejemplo de Salida en ASM 64 bits

```
.file    "asmmyc.c"
.section .rodata
.LC0:
    .string  "numero vale = %d"
.text
.globl  main
.type   main, @function
main:
.LFB2:
    push   rbp
    mov    rbp, rsp
    sub    rsp, 16
    ↳ guarda espacio para variables locales
    } armo el stack frame
```

Arguments por registros x64

```
        mov    DWORD PTR [rbp-4], 10
        mov    eax, DWORD PTR [rbp-4]
        mov    esi, eax
        mov    edi, OFFSET FLAT:.LC0
        call   printf
        mov    edi, 0 → argumento para el exit
        call   exit
.LFE2:
    .size  main, .-main
    .ident "GCC: (Ubuntu 4.8.4)"
.section .note.GNU-stack,"",@progbits
```

Obs: instrucción `lea = load effective address` Ej. `lea ax, [100h]` es como un `mov ax, 100h` → mas lento



# Análisis de manejo de pila

# Análisis detallado de manejo de pila

```
//detalle1.c
```

```
// Programa para analizar en detalle los stack frame
```

```
int suma( int sum1, int sum2)
```

```
{
```

```
    return sum1+sum2;
```

```
}
```

```
int main(void)
```

```
{
```

```
    suma(3,4);
```

```
    return 0;
```

```
}
```

# Análisis detallado de manejo de pila

(gdb) set disassembly-flavor intel

(gdb) disassemble main

Dump of assembler code for function main:

0x080483e9 <+0>: push ebp  
direc 0x080483ea <+1>: mov ebp,esp  
de memoria 0x080483ec <+3>: sub esp,0x8  $\Rightarrow$  reserva espacio en la pila  
0x080483ef <+6>: mov DWORD PTR [esp+0x4],0x4  $\Rightarrow$  guarda el 4 en la pila  
0x080483f7 <+14>: mov DWORD PTR [esp],0x3  $\Rightarrow$  guarda el 3 en la pila  
0x080483fe <+21>: call 0x80483dc <suma>  $\Rightarrow$  aparece direc de retorno cuando llama a suma  
0x08048403 <+26>: mov eax,0x0<sup>1</sup>  $= 0x08048403 \Rightarrow$  dirección a la proxima instrucción a ejecutarse  
0x08048408 <+31>: leave  $\Rightarrow$  desarmado del stackframe  
0x08048409 <+32>: ret

End of assembler dump.

<sup>1</sup> para el return 0 y en C se usa eax

# Análisis detallado de manejo de pila

(gdb) disassemble suma

Dump of assembler code for function suma:

0x080483dc <+0>: 1 push ebp

0x080483dd <+1>: 2 mov ebp,esp

0x080483df <+3>: 3 mov eax,DWORD PTR [ebp+0xc] ↛ eax = 4

0x080483e2 <+6>: 4 mov edx,DWORD PTR [ebp+0x8] ↛ edx = 3

0x080483e5 <+9>: 5 add eax,edx ↛ eax = 7 2 EBP → 1 ESP →

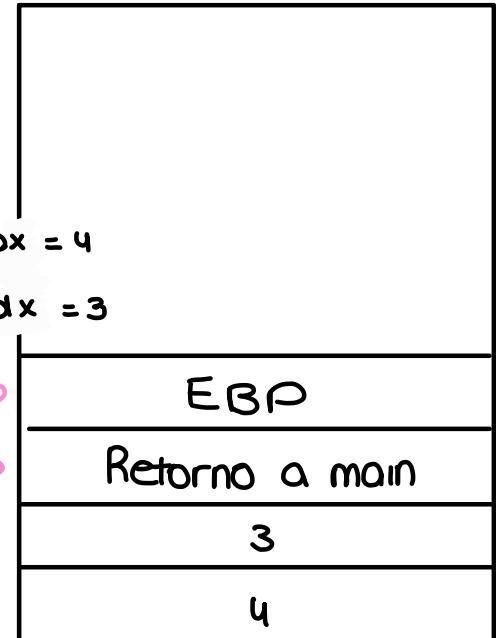
0x080483e7 <+11>: 6 pop ebp 6 ESP → ESP →

0x080483e8 <+12>: 7 ret

End of assembler dump.

(gdb)

Valor de retorno  
en eax



↗ Cuando llamo a suma, esp apunta  
a la dir de retorno a main

# Análisis – Ejemplo 2

Repetiremos el análisis pero agregando una variable local a la función main().

```
//detalle2.c

#include <string.h>

int suma( int sum1, int sum2)

{

    return sum1+sum2;

}

int main(void)

{

    int resul;

    resul=suma(3,4);

    return 0;

}
```

# Análisis – Ejemplo 2

Dump of assembler code for function main:

```
0x080483e9 <+0>: push  ebp
0x080483ea <+1>: mov   ebp,esp
0x080483ec <+3>: sub   esp,0x18 ↳ se reserva mas espacio en la pila
0x080483ef <+6>: mov   DWORD PTR [esp+0x4],0x4
0x080483f7 <+14>: mov   DWORD PTR [esp],0x3
0x080483fe <+21>: call  0x80483dc <suma>
0x08048403 <+26>: mov   DWORD PTR [ebp-0x4],eax ↳ guarda el resultado de la
0x08048406 <+29>: mov   eax,0x0
0x0804840b <+34>: leave
0x0804840c <+35>: ret
```

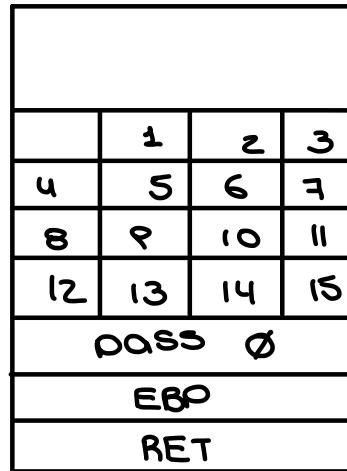
↓  
resw = suma (3,4)

guarda el resultado de la  
suma en ebp - 4

End of assembler dump.

# Análisis de pila – Ejemplo 3

```
#include <stdio.h>
#include <string.h>
    => buff
int main(void)
{
    int pass = 0;
    char buff[15];
    ⚠ Si el password tiene mas de 15
caracteres => pisa el pass
    printf("\n Enter the password : \n");
    gets(buff); => NO valida los datos de entrada
correctamente
    if(strcmp(buff, "thegeekstuff")!=0)
    {
        printf ("\n Wrong Password \n");
    }
```



```
else
{
    printf ("\n Correct Password \n");
    pass = 1;
}

if(pass!=0)
{
    /* Now Give root or admin rights to user*/
    printf ("\n Root privileges given to the user \n");
}

return 0;
}
```

## Análisis – Ejemplo 3

Al compilar en forma clásica

```
gcc detalle3.c -o detalle3
```

```
[svalles@pampero teoria]$ ./detalle3
```

```
Enter the password :  
thesecretpass
```

Wrong Password

[svalles@pampero teoria]\$

➡ el buff solo tenía 15 lugares  $\Rightarrow$  empiezo a pisar la pila

```
valles@pampero teoria]$ ./detalle3
```

Enter the password :

Wrong Password

\* stack smashing detected \*\*\*: <unknown> terminated → Si hay más de 15 caracteres, detecta que pisa el stack

res, detecta que pisa el stack

*Obs:* Si piso mucho, piso la dirección de retorno  $\Rightarrow$  cuando quiere retornar, no puede  $\Rightarrow$  segmentation fault

## Análisis – Ejemplo 3

Al compilar sin protección de stack

```
gcc detalle3.c -o detalle3 -fno-stack-protector
```

$\Rightarrow$  no detecta que pisa el stack  $\Rightarrow$  pisa el flag de pass y printea

```
[svalles@pampero teoria]$ ./detalle3sinprotect
```

```
Enter the password :  
123456789012345
```

```
Wrong Password
```

```
[svalles@pampero teoria]$ ./detalle3sinprotect
```

```
Enter the password :  
1234567890123451
```

```
Wrong Password
```

$\Rightarrow$  pisa el pass  $\Rightarrow$  falta validar los datos de entrada

```
Root privileges given to the user  
[svalles@pampero teoria]$
```

# Salida en ASM de Ejemplo 3

```
.file "detalle3.c"
.intel_syntax noprefix
.text
.section .rodata
.LC0:
.string "\n Enter the password : "
.LC1:
.string "thegeekstuff"
.LC2:
.string "\n Wrong Password "
.LC3:
.string "\n Correct Password "
.align 8
.LC4:
.string "\n Root privileges given to
the user "
.text
.globl main
.type main, @function
```

main:

.LFB0:

.cfi\_startproc

**push rbp**

.cfi\_def\_cfa\_offset 16

.cfi\_offset 6, -16

**mov rbp, rsp**

.cfi\_def\_cfa\_register 6

sub rsp, 32

**mov rax, QWORD PTR fs:40**

**mov QWORD PTR -8[rbp], rax**

xor eax, eax

mov DWORD PTR -28[rbp], 0

lea rdi, .LC0[rip]

call puts@PLT

lea rax, -23[rbp]

mov rdi, rax

mov eax, 0

variables
CANARY
EBP



# Salida en ASM de Ejemplo 3

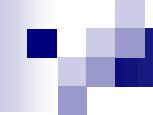
```
call  gets@PLT
    lea   rax, -23[rbp]
    lea   rsi, .LC1[rip]
    mov   rdi, rax
    call  strcmp@PLT
    test  eax, eax
    je    .L2
    lea   rdi, .LC2[rip]
    call  puts@PLT
    jmp   .L3
.L2:
    lea   rdi, .LC3[rip]
    call  puts@PLT
    mov   DWORD PTR -28[rbp], 1
.L3:
    cmp   DWORD PTR -28[rbp], 0
    je    .L4
    lea   rdi, .LC4[rip]
    call  puts@PLT
```

```
.L4:
    mov   eax, 0
    mov   rdx, QWORD PTR -8[rbp]
    xor   rdx, QWORD PTR fs:40
    je    .L6
    call  __stack_chk_fail@PLT
.L6:   ▶ chequea si el CANARY fue cambiado
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size  main, .-main
    .ident "GCC: (GNU) 7.3.0"
    .section .note.GNU-stack,"",@progbits
```

el CANARY es un valor random  
⇒ es difícil descubrir cuál es el valor

# SPP (Stack Smashing Protector)

- Lo implementa gcc
- Utiliza la función `__stack_chk_fail`
- Ubica un valor entre EBP y las variables locales que se lo denomina CANARY
- Antes de retornar verifica que el CANARY no haya sido modificado
- Si lo fue termina la ejecución.

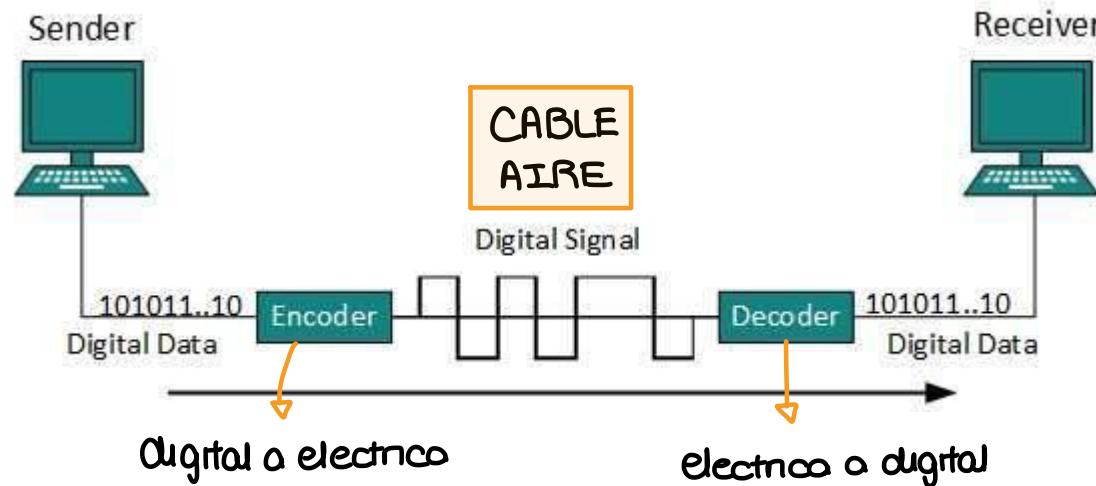


# Arquitecturas de la computadoras

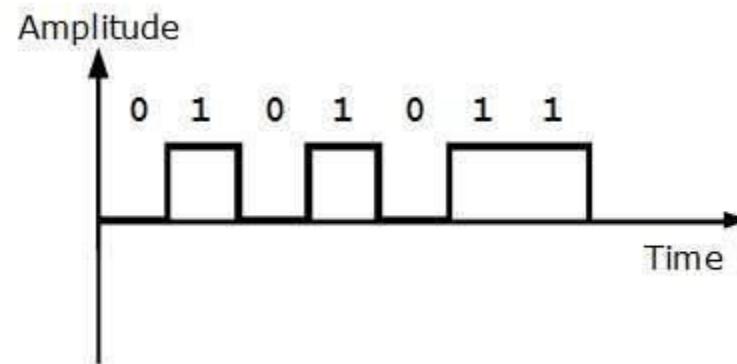
# Transmisión digital

Memoria = externo al µP  
⇒ señales digitales para comunicarse con la memoria

## Codificación de línea

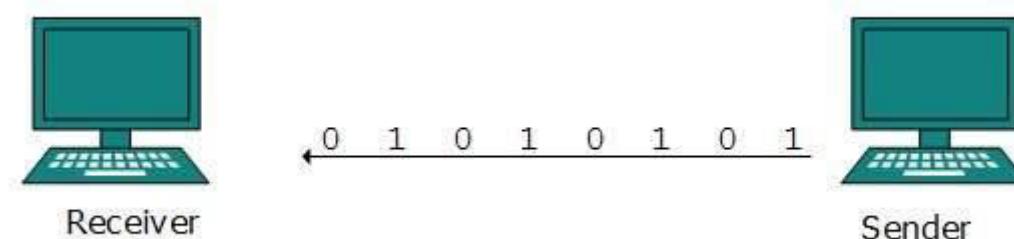


## Codificación unipolar



# Transmisión digital

## Transmisión serie



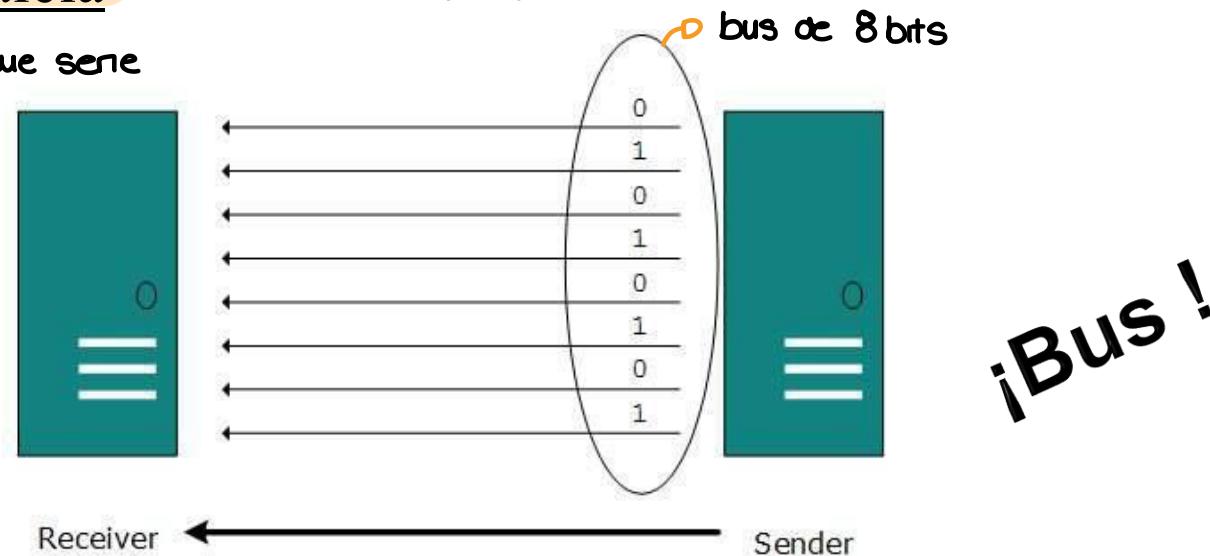
## Transmisión paralela

⇒ acelera la transmisión

⇒ mucho más performante que serie

[Bus] = conjunto de cables que  
comunican al procesador con  
la memoria

Desventaja: más caro



# Tipos de Arquitectura

## Von Neumann

- ⇒ más simple para implementar
- ⇒ más lento porque hay un solo juego de buses
- ⇒ voy dos veces a memoria

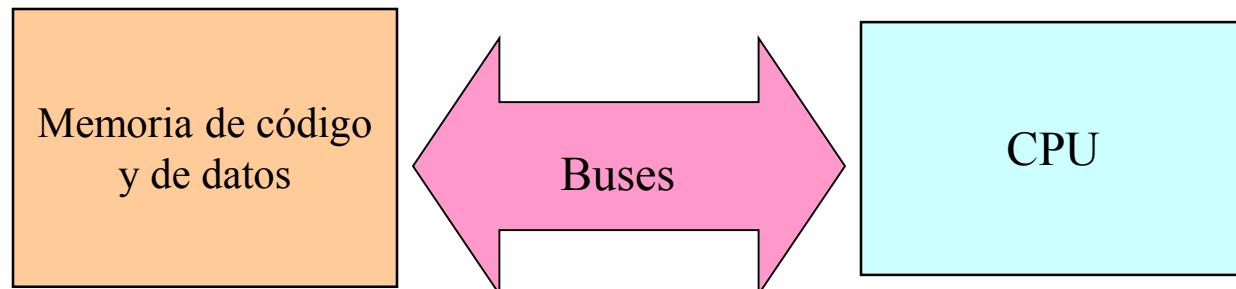
Direcciones

0000h

0001h

....

nnnn



## Harvard

- ⇒ más caro

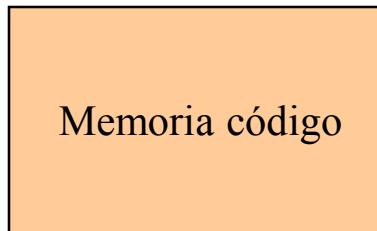
Direcciones

0000h

0001h

....

nnnn



Direcciones

0000h

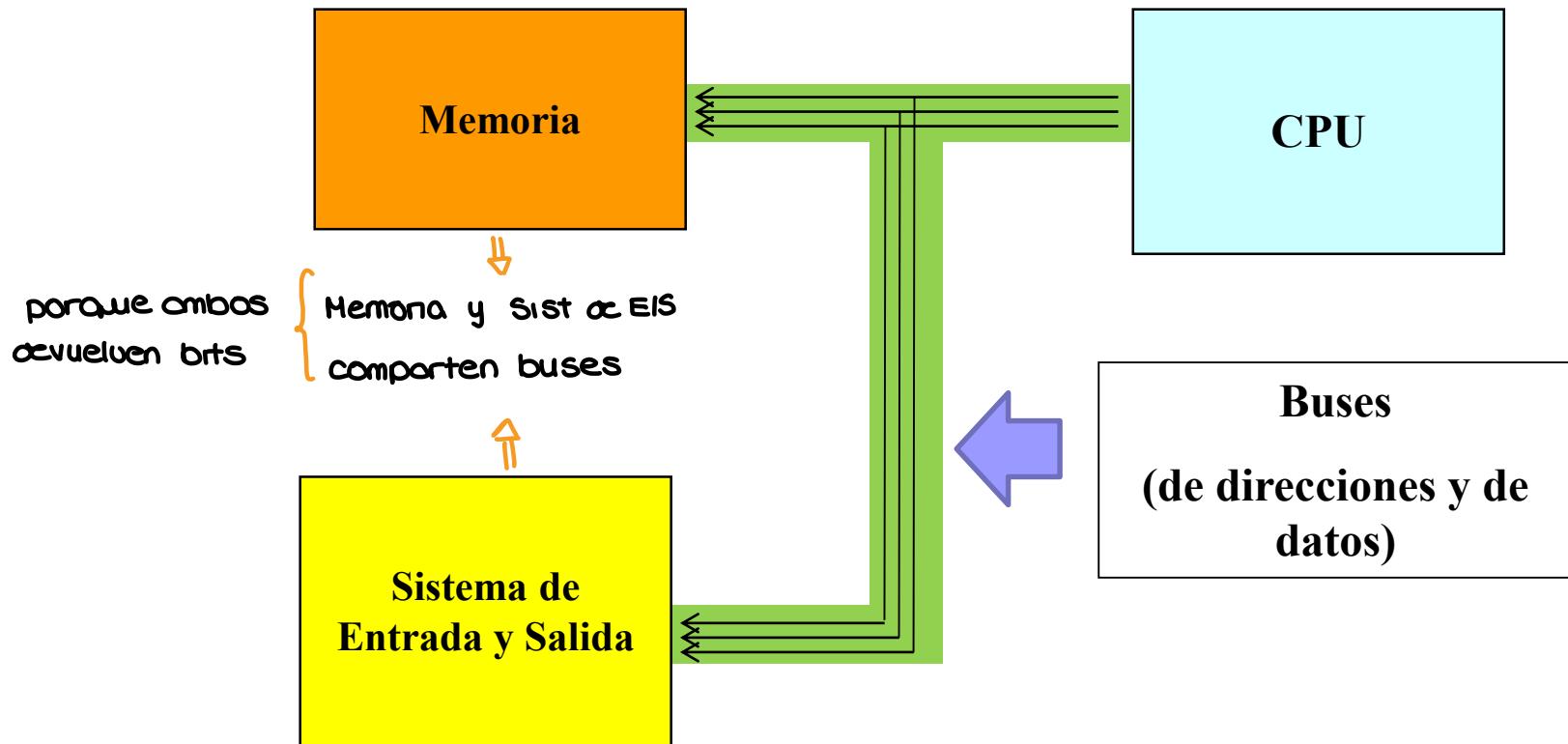
0001h

....

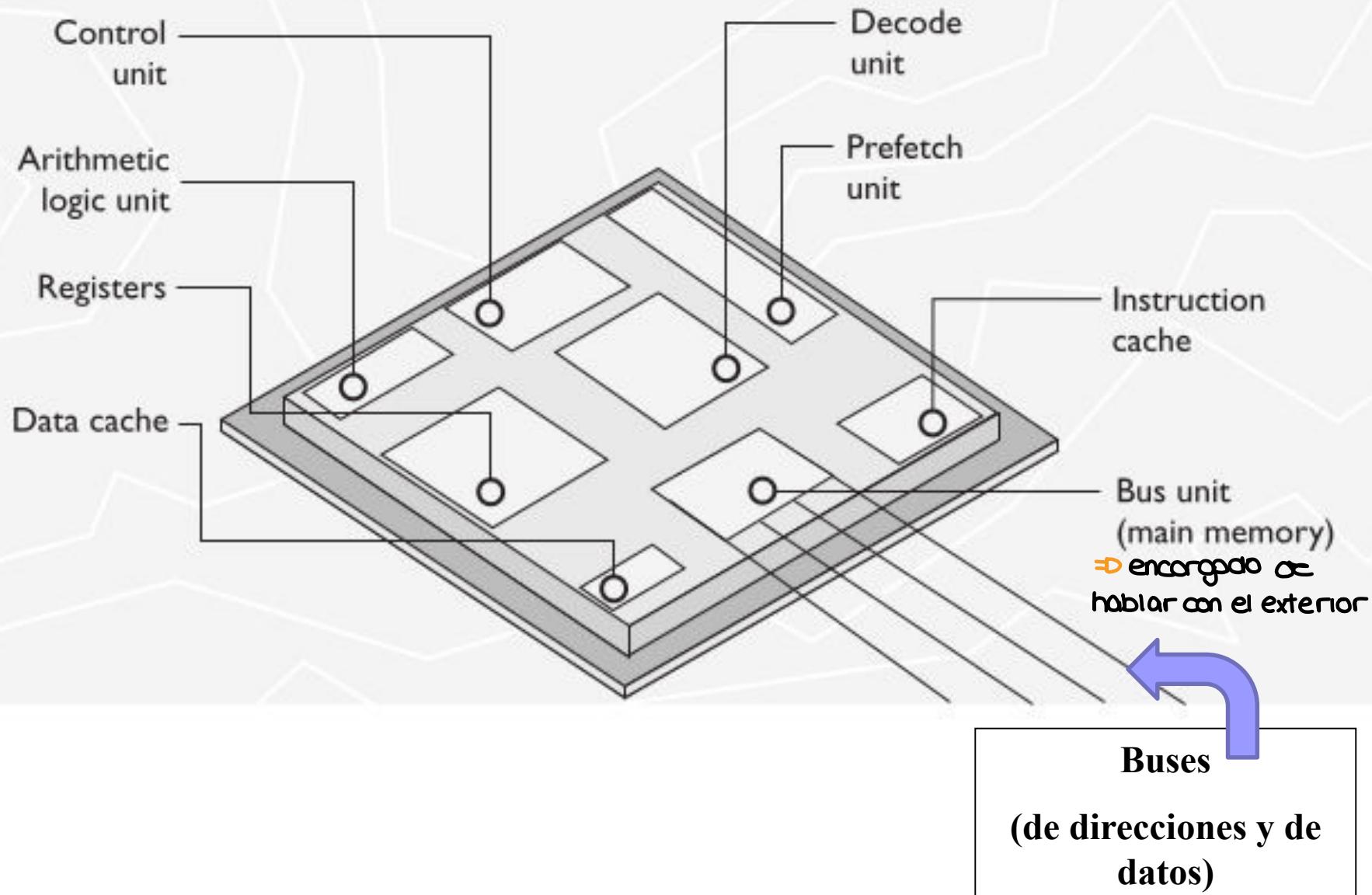
nnnn

Mientras busco la instrucción siguiente, al mismo tiempo busco los datos de la instrucción anterior

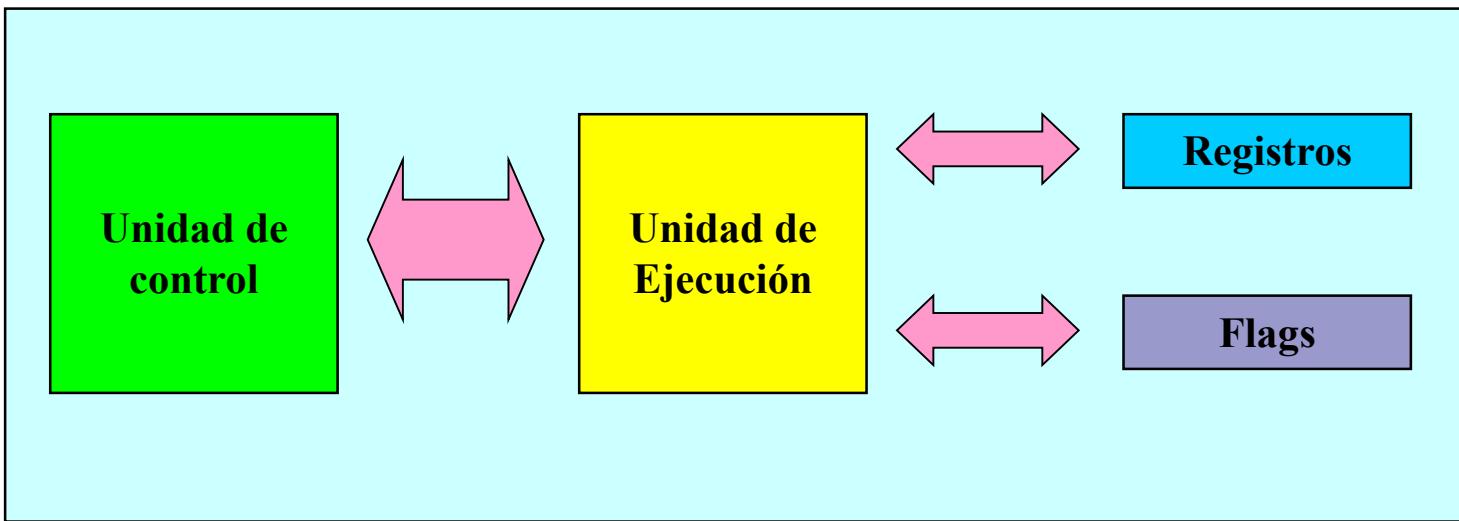
# Sistema de Entrada y Salida



# CPU



# CPU (Resumen)



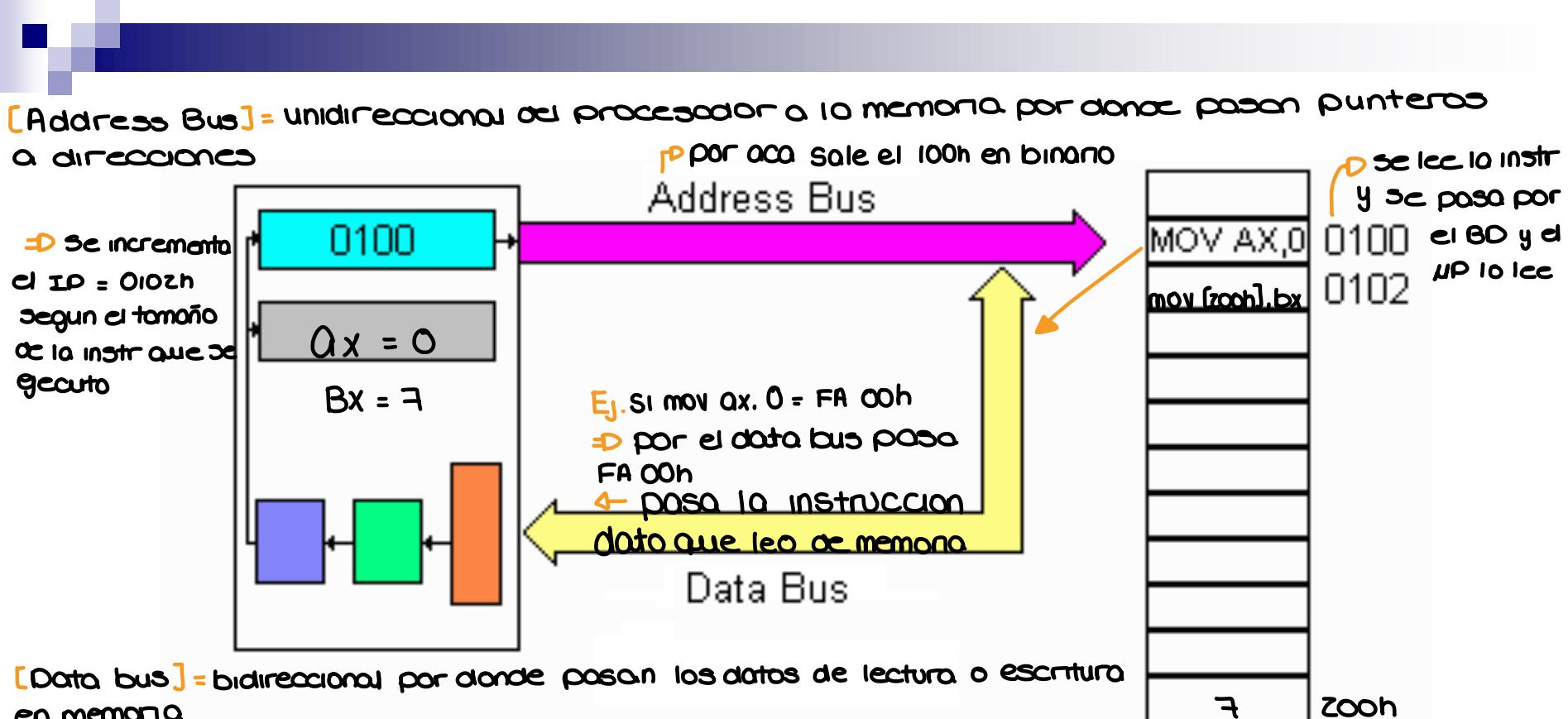
→ cambia registros, flags, etc

**Unidad de control:** Recupera Instrucciones de memoria, las decodifica, escribe en memoria

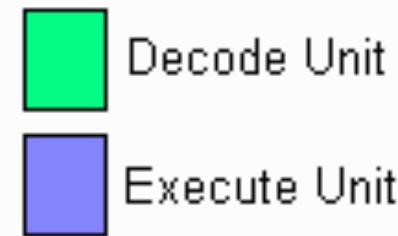
**Unidad de Ejecución:** Lleva a cabo la ejecución de la instrucción

**Registros:** Memoria interna utilizada como variable.

**Flags:** Indican eventos luego de ejecutar las instrucciones



[Data bus] = bidireccional por donde pasan los datos de lectura o escritura en memoria



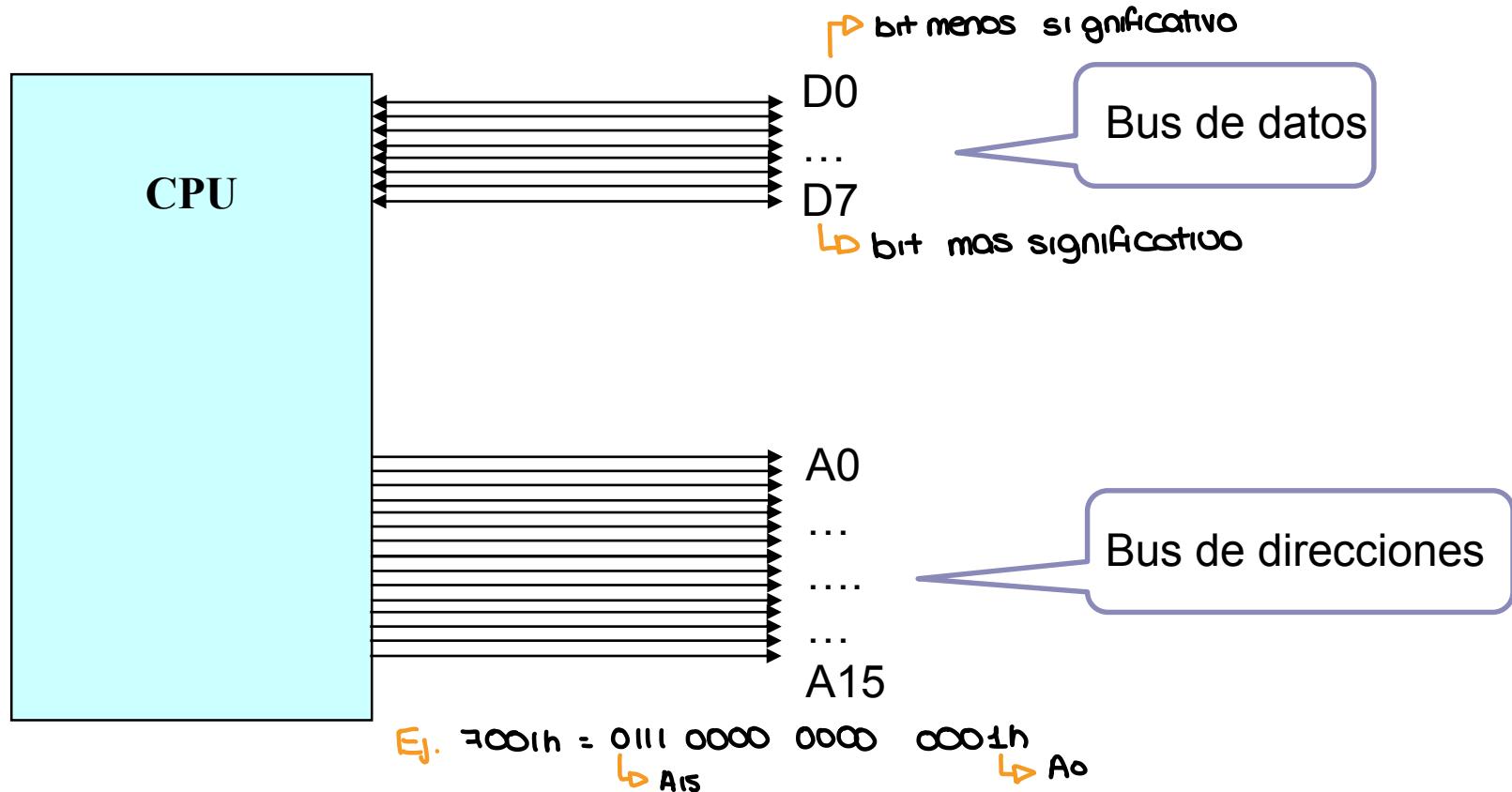
⚠ Data bus es bidireccional por escritura en memoria

Ej. mov [z00h], BX → escribir el contenido de BX en la posicion z00h en memoria  
↳ instruccion de escritura en memoria

- (1) Address Bus = 102h
- (2) Data Bus = instruccion
- (3) Adress Bus = z00h
- (4) Data Bus = bx

# Mapa de memoria

- ⇒ combinatoria de datos que puede ver el procesador
- Supongamos un procesador que tiene 16 líneas de bus de direcciones y 8 líneas de bus de datos.
- ¿ Que cantidad de información puede acceder ?



# Mapa de memoria (2)

- ¿Y un procesador que tiene 16 líneas de bus de direcciones y 16 líneas de bus de datos ?
- ¿Y un procesador con 32 líneas de datos y 32 líneas de Direcciones ?

## 1. 8 de datos y 16 de direcciones

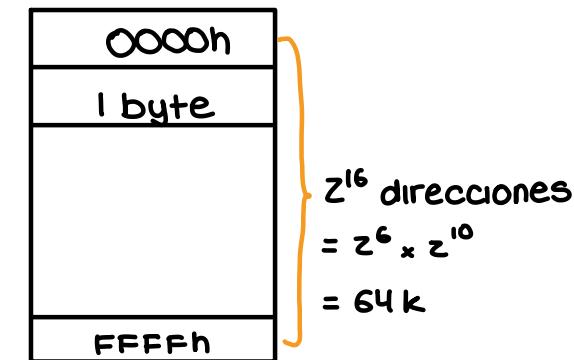
- dir minima = 0000 0000 0000 0000 b = 0000h
- dir maxima = 1111 1111 1111 1111 b = 1111h
- 8 bits = 1 byte por dirección

TOTAL:  $64\text{K} \times 1\text{B} = 64\text{kB}$  and B = byte

2. 16 de datos y 16 de direcciones  $\Rightarrow 64\text{k} \times 2\text{B} = 128\text{kB}$

3. 32 de datos y 32 de direcciones  $\Rightarrow 2^2 \times 2^{30} \times 4\text{B} = 16\text{GB}$

4. 64 de direcciones y 8 de datos  $\Rightarrow 2^4 \times 2^{60} \times B = 16\text{EB}$

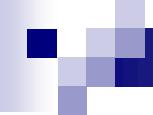


$$\begin{aligned}Z^{10} &= k \\Z^{20} &= M \\Z^{30} &= G \\Z^{40} &= T \\Z^{50} &= P \\Z^{60} &= E\end{aligned}$$

# Registros de Intel

IP: Puntero a instrucción.

¿ Cual es la primera instrucción que ejecuta el micro al encenderse ?



# Memorias

# Memorias - Clasificación

- Por el **modo** en que se accede a los datos
- Por las **operaciones** que aceptan
- Por la **duración** de los datos

# Memoria – Tipo ROM

**ROM ( Read Only Memory )**

PROM ( Programmable ROM )

EPROM ( Erasable PROM )

Flash, EEPROM ( Electric Erasable Programmable ROM )

► NO se borra la información de este memoria

- Mantienen su información **sin energía** (no volátil)
- La escritura es más lenta que la RAM.

# Memoria – Tipo RAM

## **RAM ( Random Access Memory )**

DRAM ( Dinamic RAM ) ➔ lo que se utiliza en los PC

SRAM ( Static RAM )

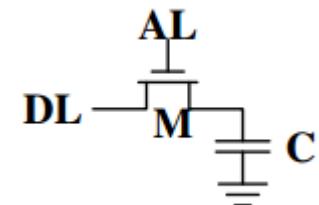
Pierde su información sin energía. (volátil)

# Memorias – Tipos - RAM

## DRAM

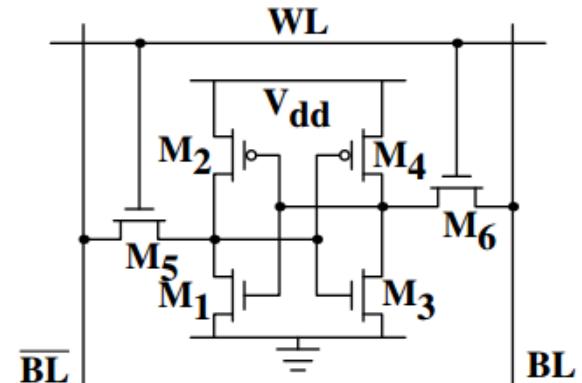
⚠ hay que leer el dato cada  $n$  ms sino se olvida del dato

- Necesita refresco de valores cada  $n$  milisegundos
- Menos compleja. Mas económica.
- Mas lenta



## SRAM

- No necesita refresco.
- Mas compleja, más costosa.
- Mas rápida
- Se suele utilizar para memoria cache.



## SDRAM ( Synchronous DRAM )

# Memorias – Tiempo de Acceso

Es el tiempo que le toma a una memoria RAM para completar un acceso después de otro.

Se compone de:

- **Latencia** ( tiempo que tarda en devolver el valor la memoria )
- **Transferencia**  $\Rightarrow$  tiempo que tarda en llevar el pedido a la memoria + lo que tarda en traer al microprocesador lo obtenido de la memoria

Las DRAM suelen tener tiempos entre 50 y 150 ns.

Las SRAM menores a 10 ns.

**Obs:** latencia + lenta que la transferencia ya que la transferencia ocurre en los buses que están compuestos de electrones que tienen una velocidad del orden de  $c/z$  con  $c$  siendo la velocidad de la luz

# Memorias - Operación

Las memorias para operar utilizan:

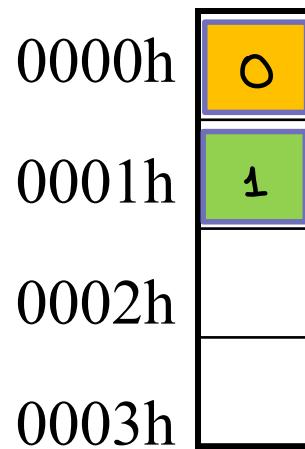
- Acción a realizar (lectura o escritura)
- Dirección de la palabra a acceder.
- Dato (entrante o saliente según acción)

# Memorias – Estructura

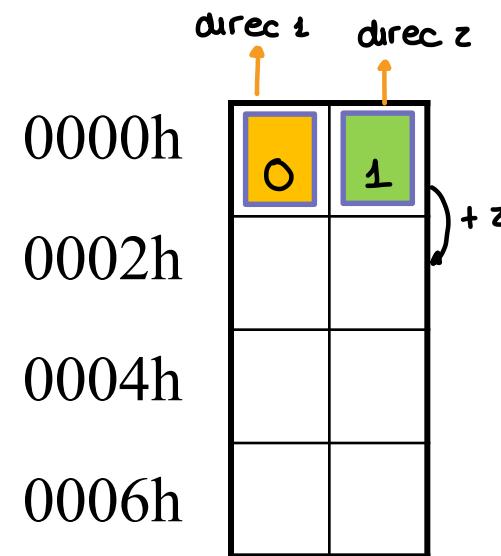
TODAS las direcciones de memoria contienen 1 byte de info.

Si el procesador, como es el caso de Intel, quiere mantener compatibilidad hacia atrás, permite acceder a la memoria a **nivel byte**.

Por lo tanto la decodificación cambia según el tipo de memoria

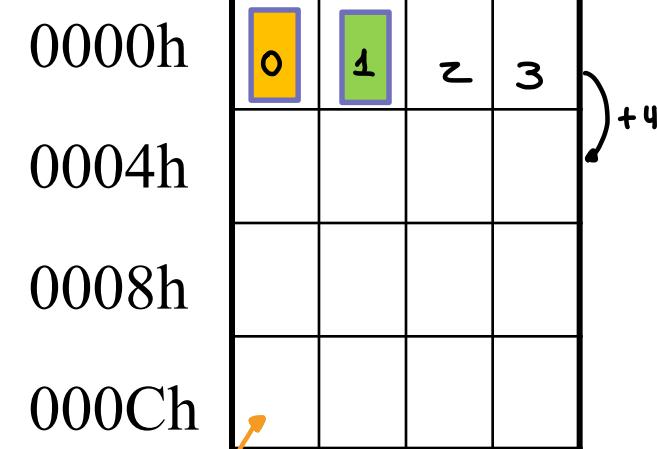


Bus de 8 líneas



Bus de 16 líneas  $\Rightarrow$  BD

$\Rightarrow$  vijan 16 bits PERO cada  
drec tiene 8 bits

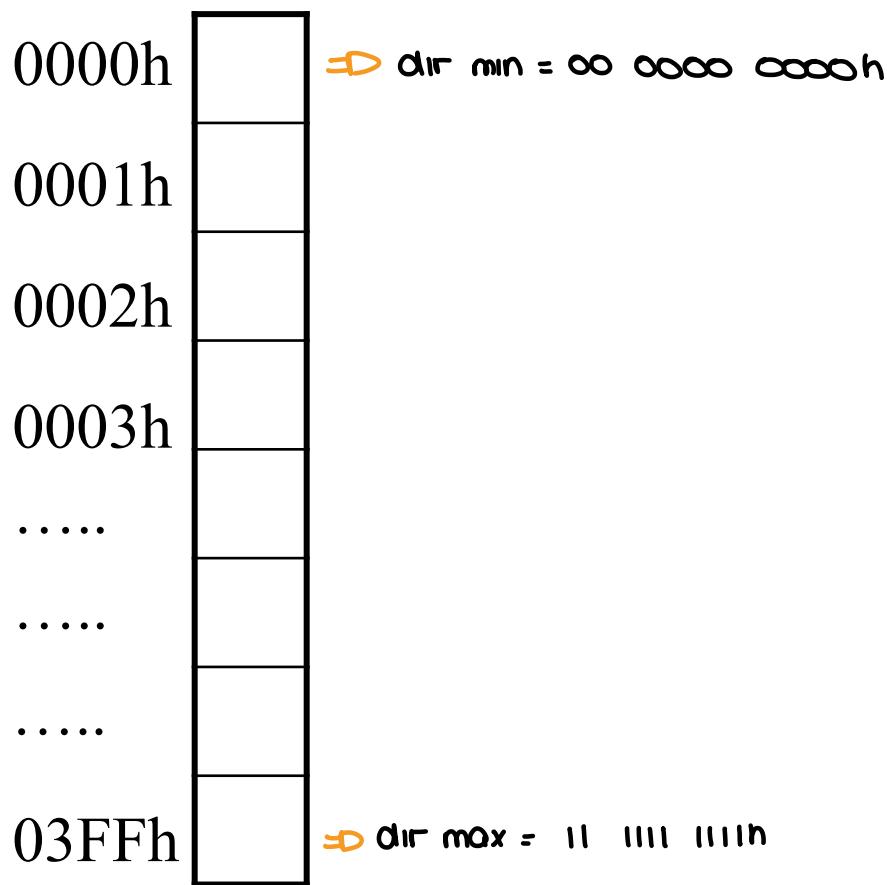


Bus de 32 líneas

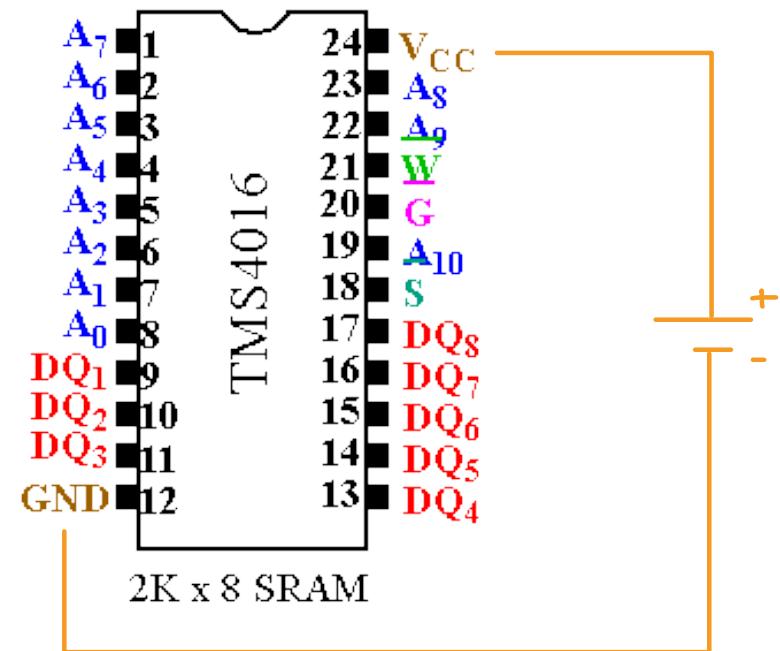
puedo combinar un solo  
cuadrado  $\Rightarrow$  NO tengo que  
combinar toda la palabra

# Memorias – Estructura

Ejemplo de memoria de 1k x 8



# Memoria Comercial (1)



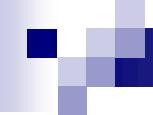
Pin(s)	Function
A <sub>0</sub> -A <sub>10</sub>	Address
DQ <sub>0</sub> -DQ <sub>7</sub>	Data In/Data Out
S (CS)	Chip Select
G (OE)	Read Enable
W (WE)	Write Enable

⇒ 11 líneas  
⇒ 8 líneas  
⇒ Si está en 1, la memoria trabaja  
Sino la memoria no hace NADA

# Memoria Comercial (2)



- SDRAM 4GB
- DDR3
- 240 pines
  - 64 pines de bus de datos
  - Bus de Address dividido en buses de 16 líneas

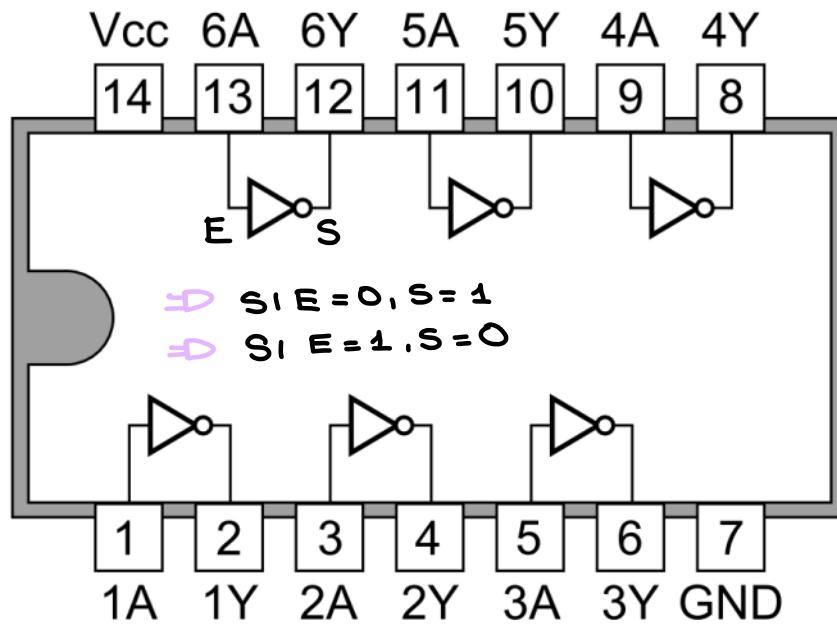


# **Integrados Compuertas y Decodificadores**

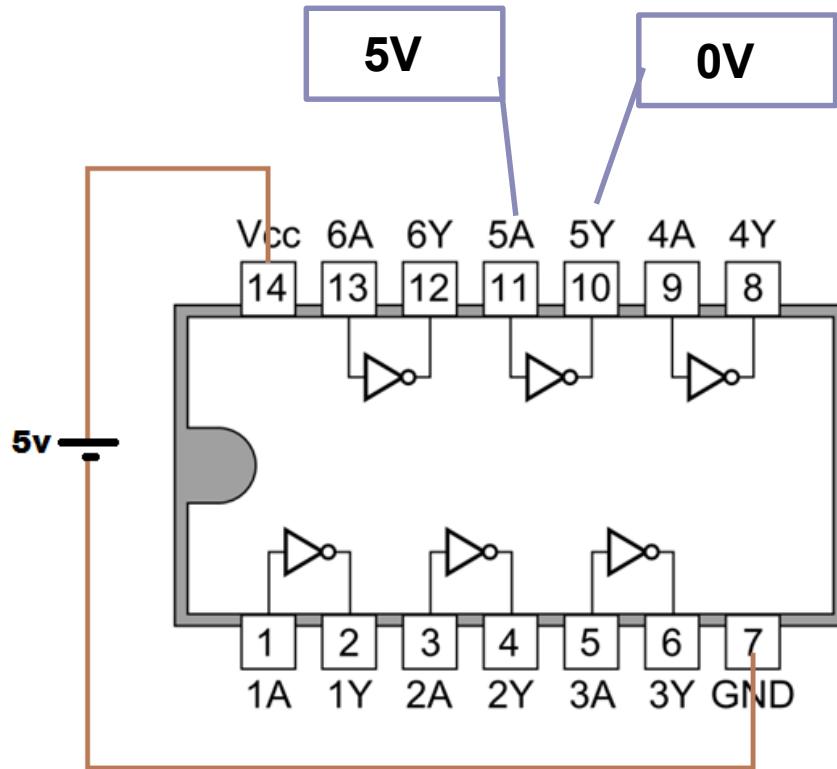
# Ejemplo de integrado (compuerta NOT)

- Código: 74LS04
- Composición: 6 compuertas NOT

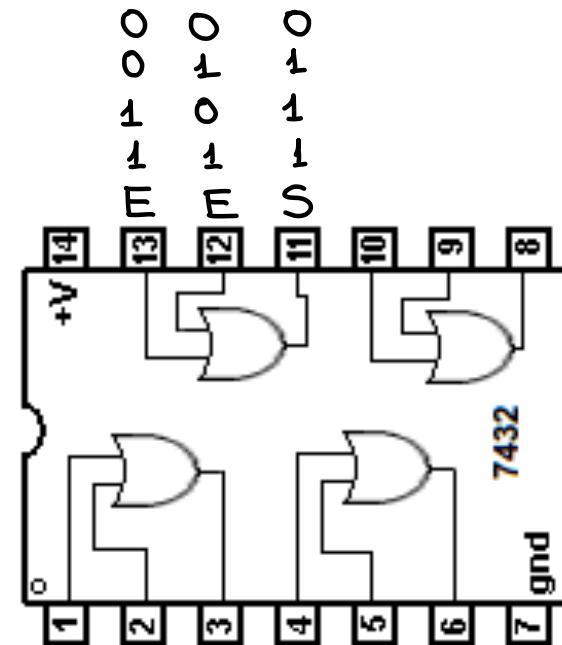
7404 Hex Inverters



# Ejemplos de integrados

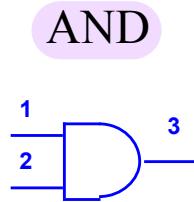


6 compuertas NOT

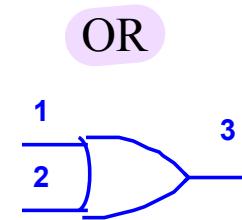


4 compuertas OR

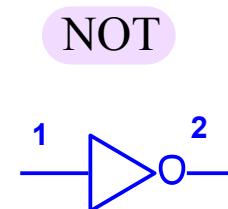
# Compuertas



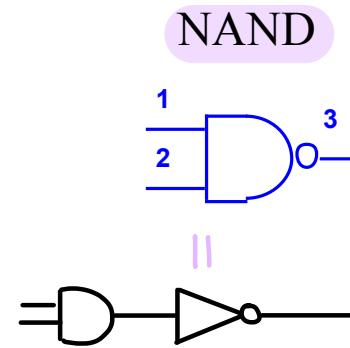
B	A	B.A
0	0	0
0	1	0
1	0	0
1	1	1



B	A	B+A
0	0	0
0	1	1
1	0	1
1	1	1



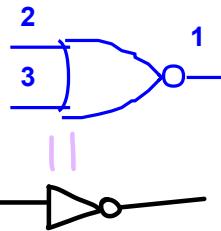
A	$\bar{A}$
0	1
1	0



B	A	$\bar{B} \cdot \bar{A}$
0	0	1
0	1	1
1	0	1
1	1	0

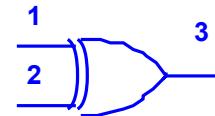
# Compuertas

NOR



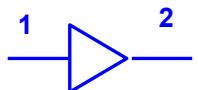
B	A	$\overline{B + A}$
0	0	1
0	1	0
1	0	0
1	1	0

XOR

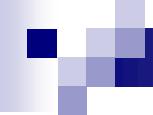


B	A	$B \oplus A$
0	0	0
0	1	1
1	0	1
1	1	0

BUFFER

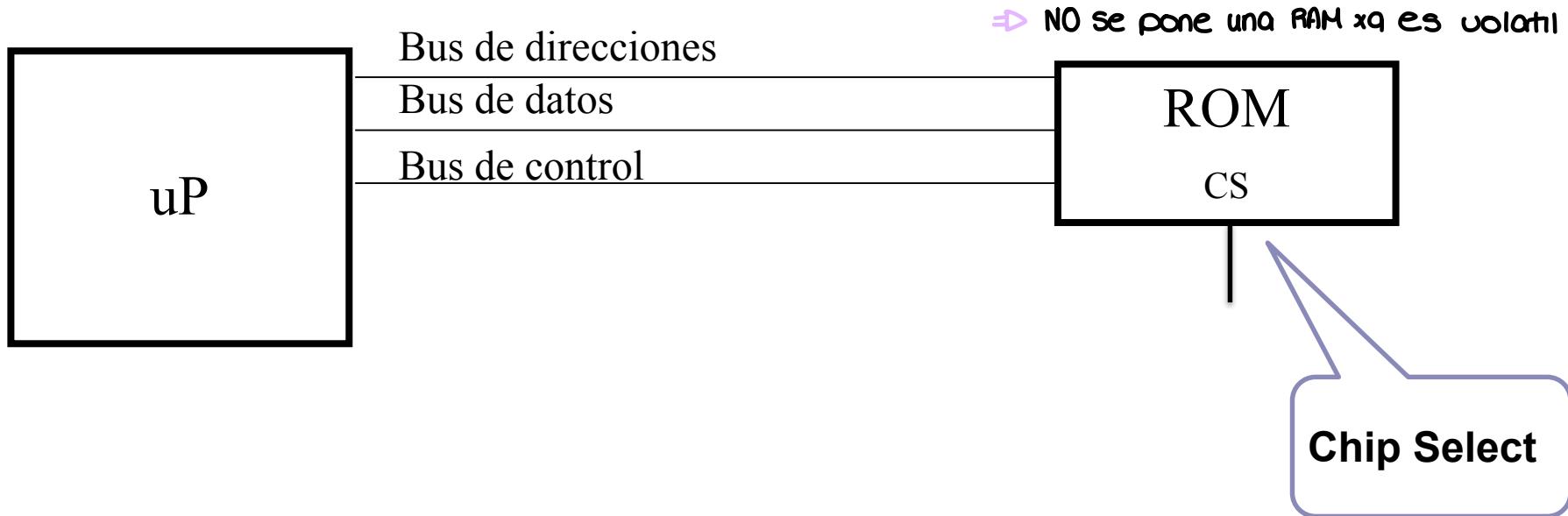


A	A
0	0
1	1



# Decodificación de Hardware

# Sistema 1



¿Qué solución/sistema puedo implementar con este hardware?

¿Que limitaciones tiene ? ⇒ es lenta

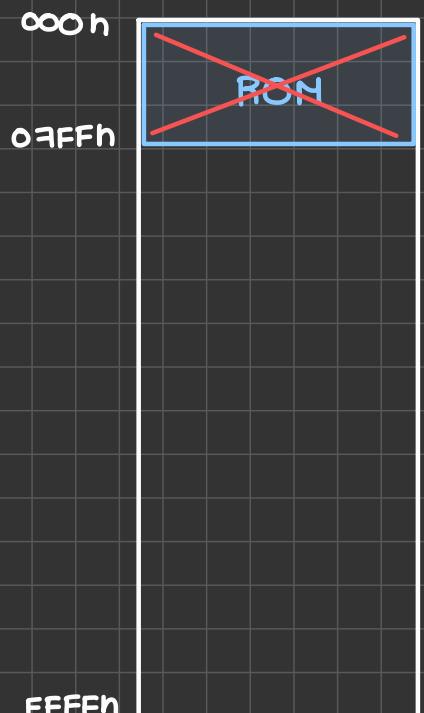
# EJ Decodificación de ROM (1)

Se dispone de un microprocesador con 16 líneas de bus de direcciones y 8 líneas de bus de datos.

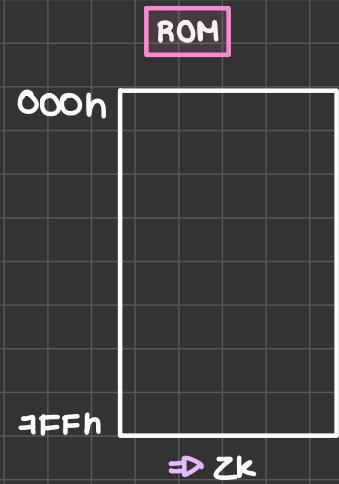
Se desea conectar el procesador con un integrado de ROM de 2K x 8



### Mapa de memoria



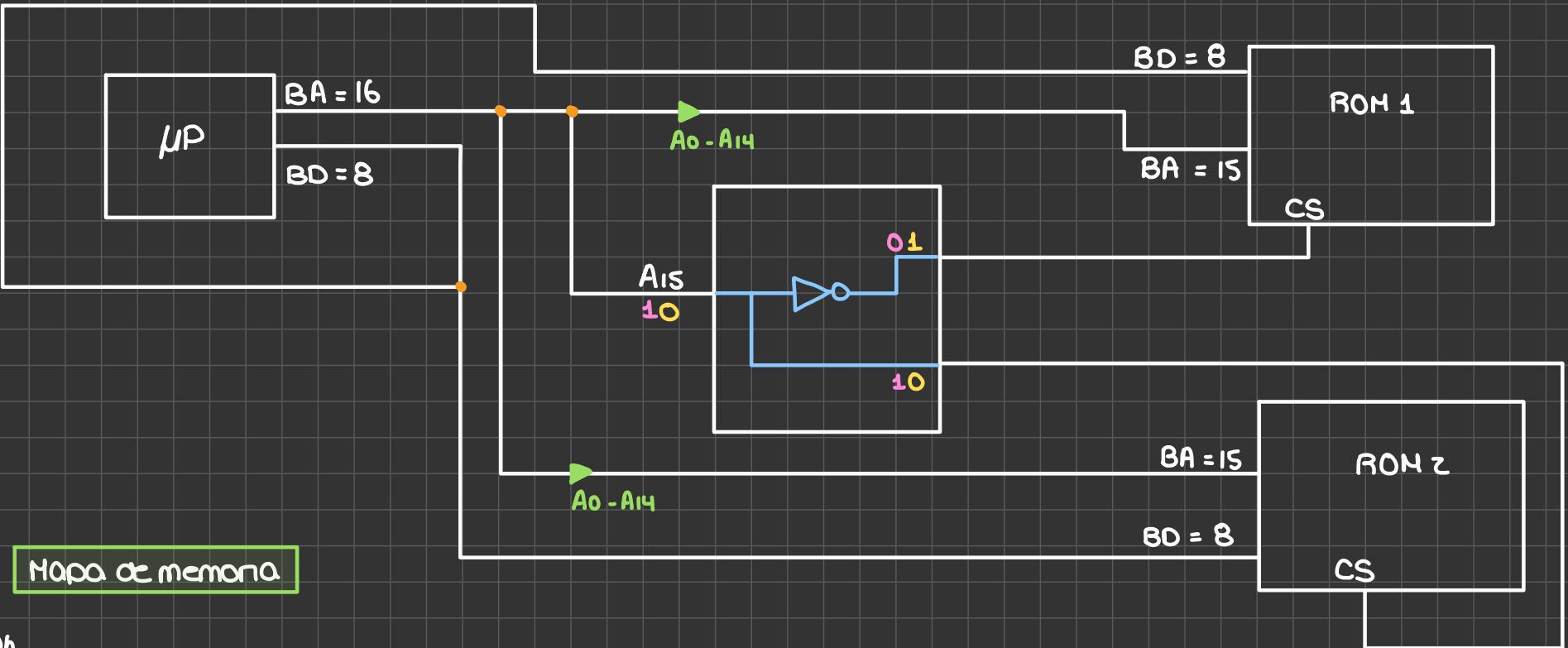
Obs: ROM entra 32 veces en memoria. PERO antes de ubicar la ROM en memoria hay que ver adonde apunta la IP  
 [Circuito decodificador] = encender a los perifericos como el μP se comunica con ellos ⇒ por el CS (chip select)



## EJ Decodificación de ROM (2)

Se dispone de un microprocesador con 16 líneas de bus de direcciones y 8 líneas de bus de datos.

Se desea conectar el procesador dos integrados de ROM de 32K x 8



Mapa de memoria



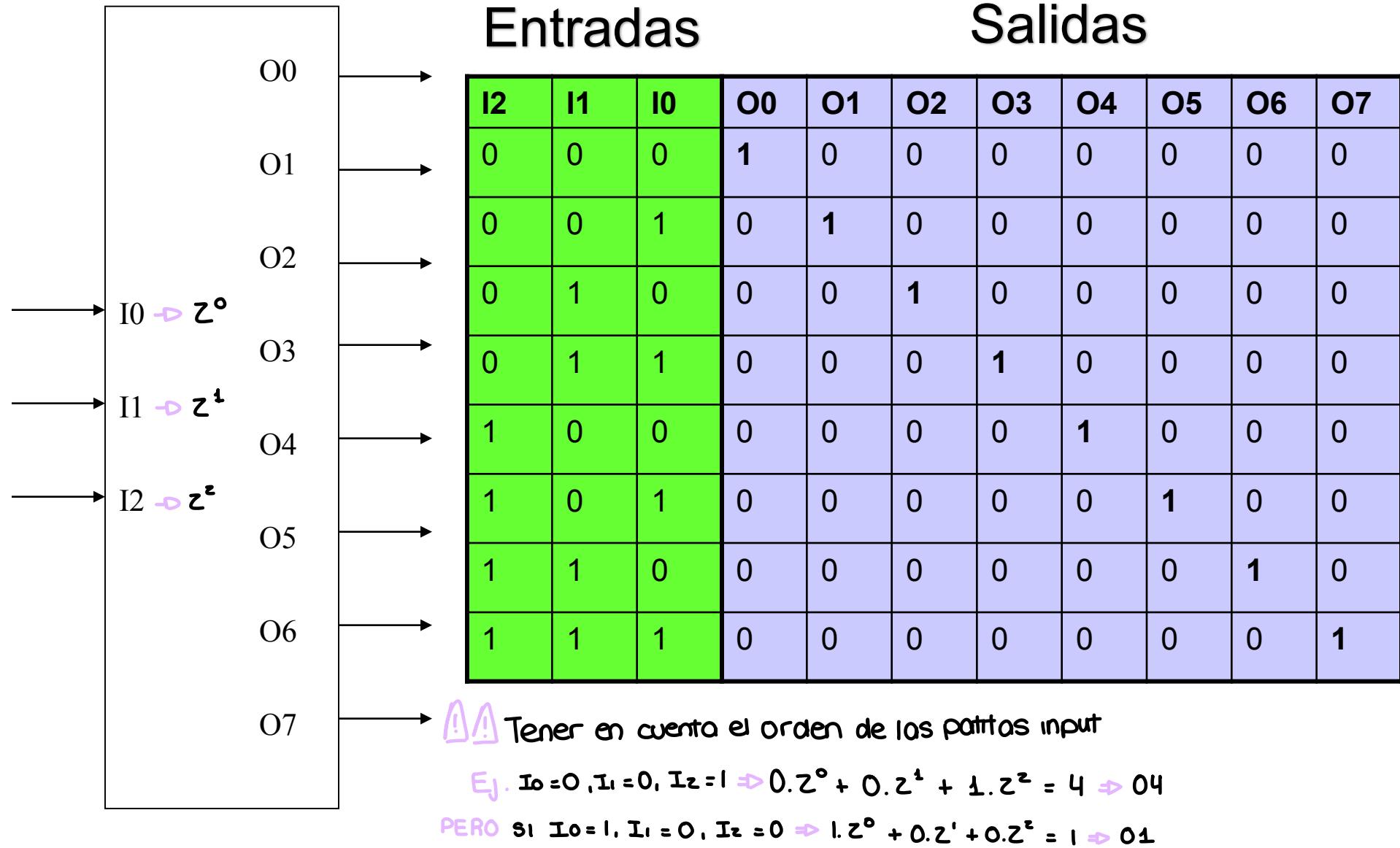
El CD engaña a la ROM para que piense que se está accediendo a una dirección en su memoria.

Notemos que las direcciones de ROM 1 tienen A<sub>15</sub> = 0 y las direcciones de ROM 2 tienen A<sub>15</sub> = 1 dnd A<sub>15</sub> = bit más significativo.

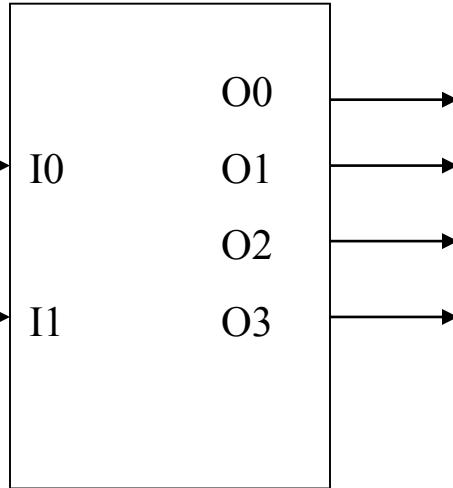
⚠️⚠️ Los BD se pueden conectar así porque las ROM NUNCA se prenden simultáneamente ➡️ NO comparten direcciones. Lo mismo pasa con A<sub>0</sub> o A<sub>14</sub>.

Obs: cada ROM es de 32k x 8 ➡️ ocupan todo el mapa ➡️ no podremos agregar más

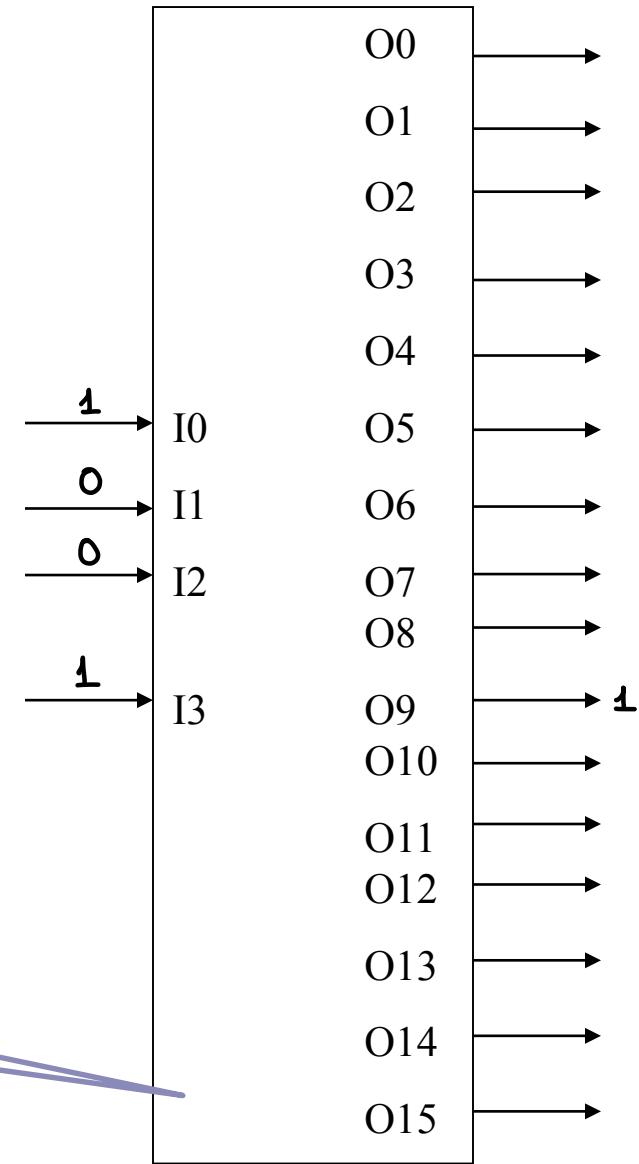
# Decodificador 3 a 8



# Otros decodificadores



Decodificador de 2 a 4



Decodificador de 4 a 16

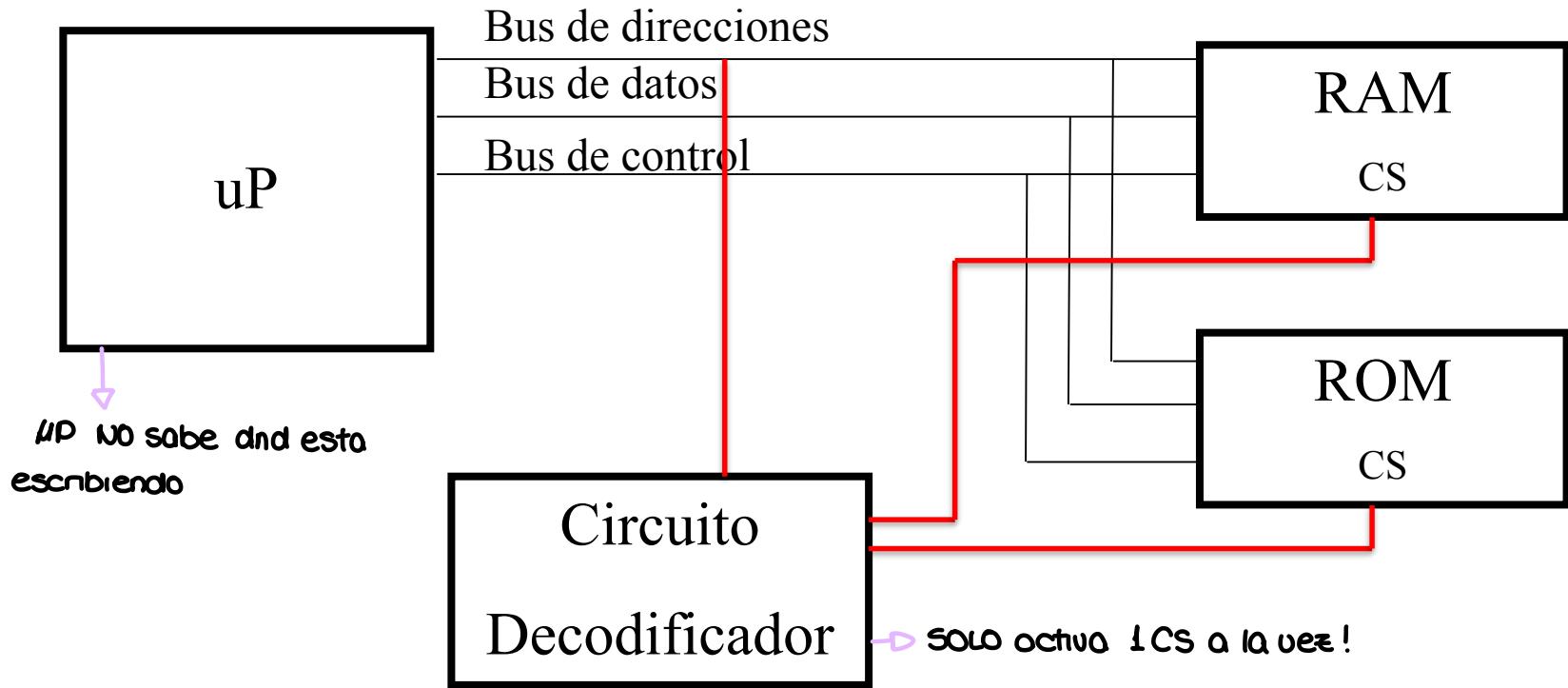
$$\text{Ej. } 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 9$$

## EJ Decodificación de ROM (3)

Se dispone de un microprocesador con 16 líneas de bus de direcciones y 8 líneas de bus de datos.

Se desea conectar el procesador dos integrados de ROM de 16K x 8 a partir de la dirección de memoria 8000h

# Sistema 2

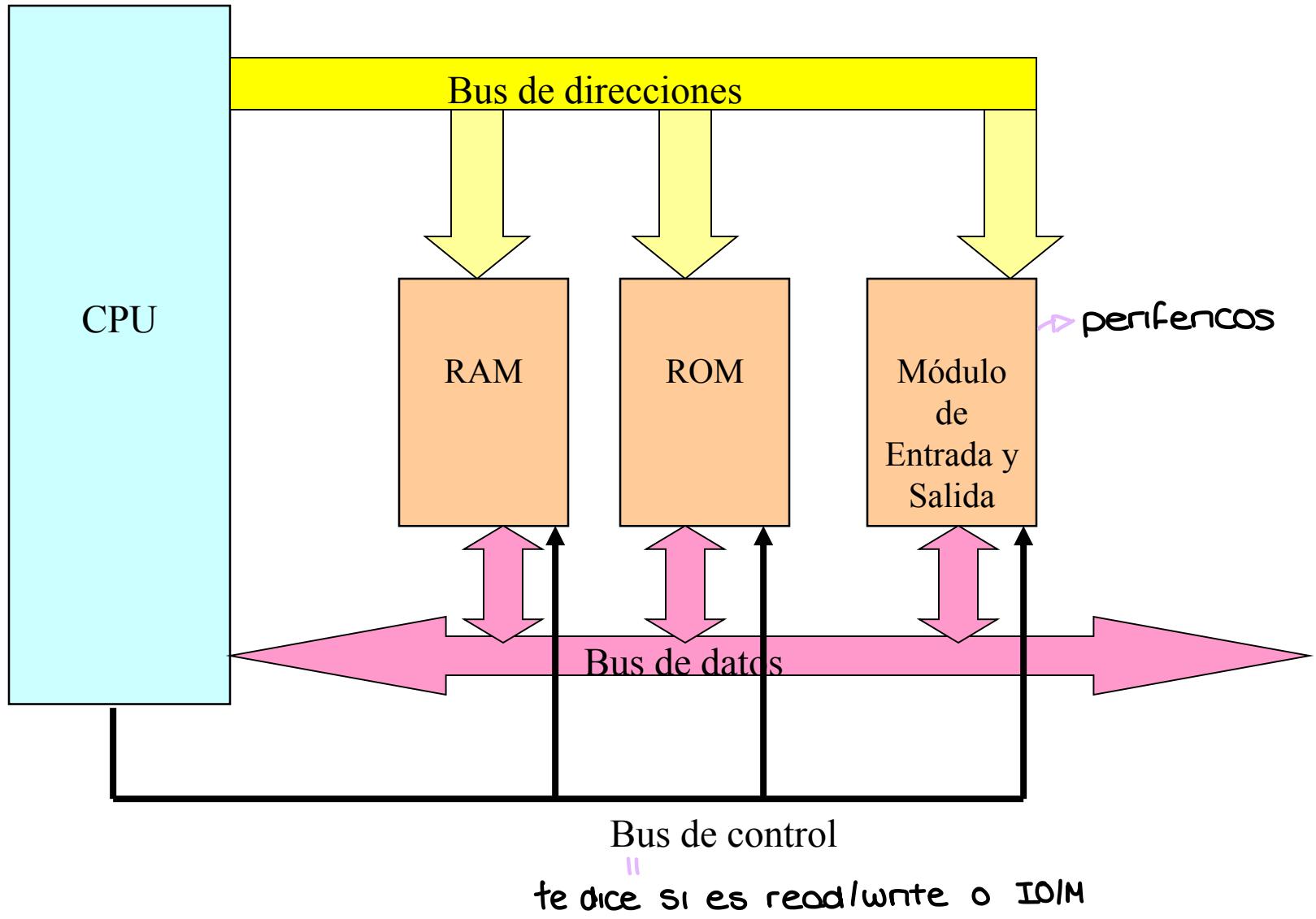


⚠ No se pueden prender dos perifericos a lo vez pues habra una colision.

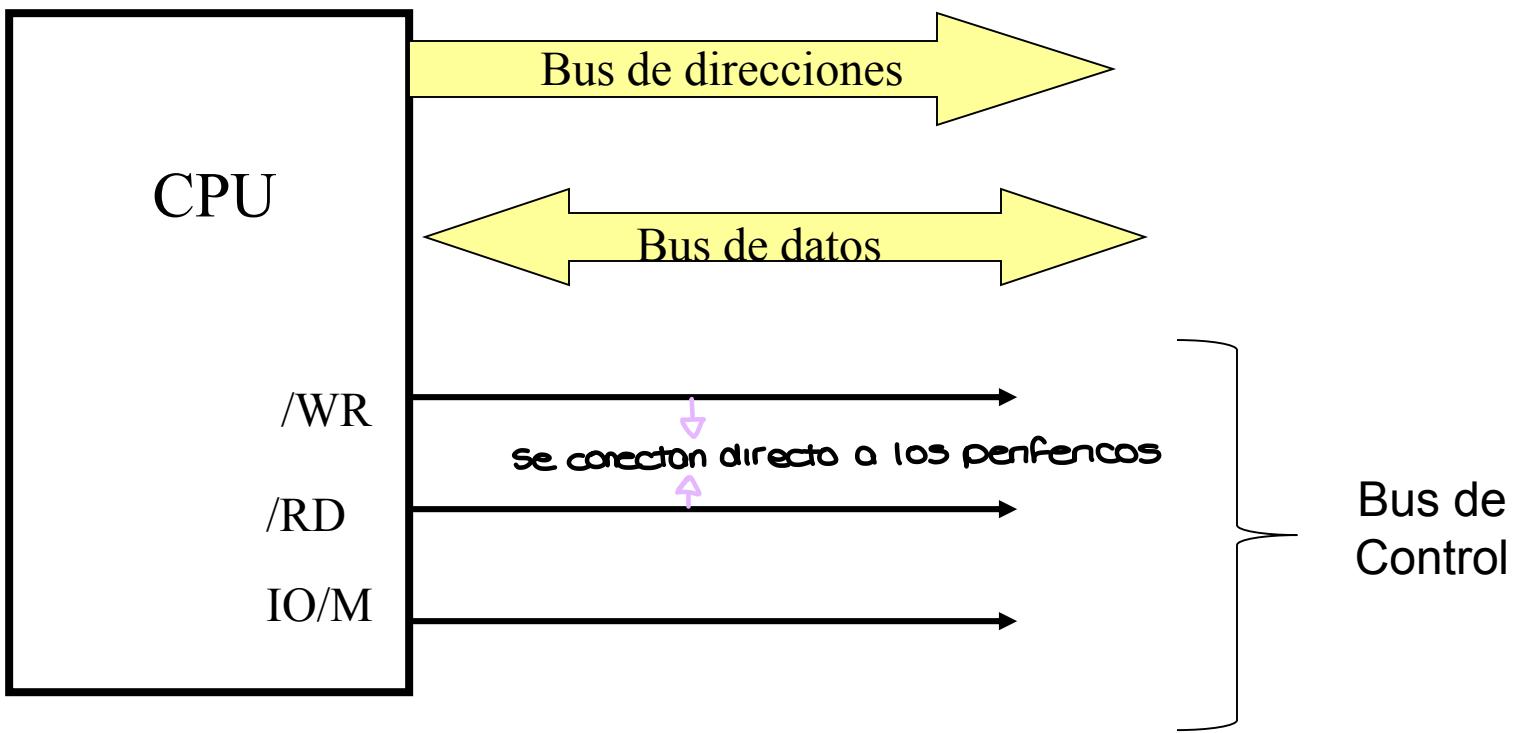
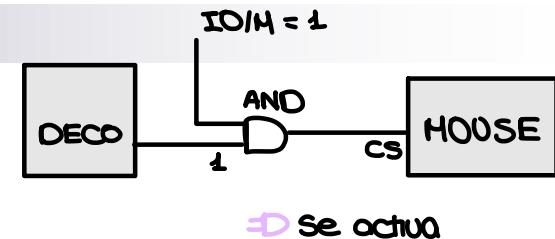
⇒ el circuito decodificador activa el chip select del periferico que quiero activar o apagar

Obs: el periferico NO sabe dnd esto ubicado en el mapa de memoria.

# Buses



# Pines básicos de control



**/WR** (Cuando vale cero hay una escritura)

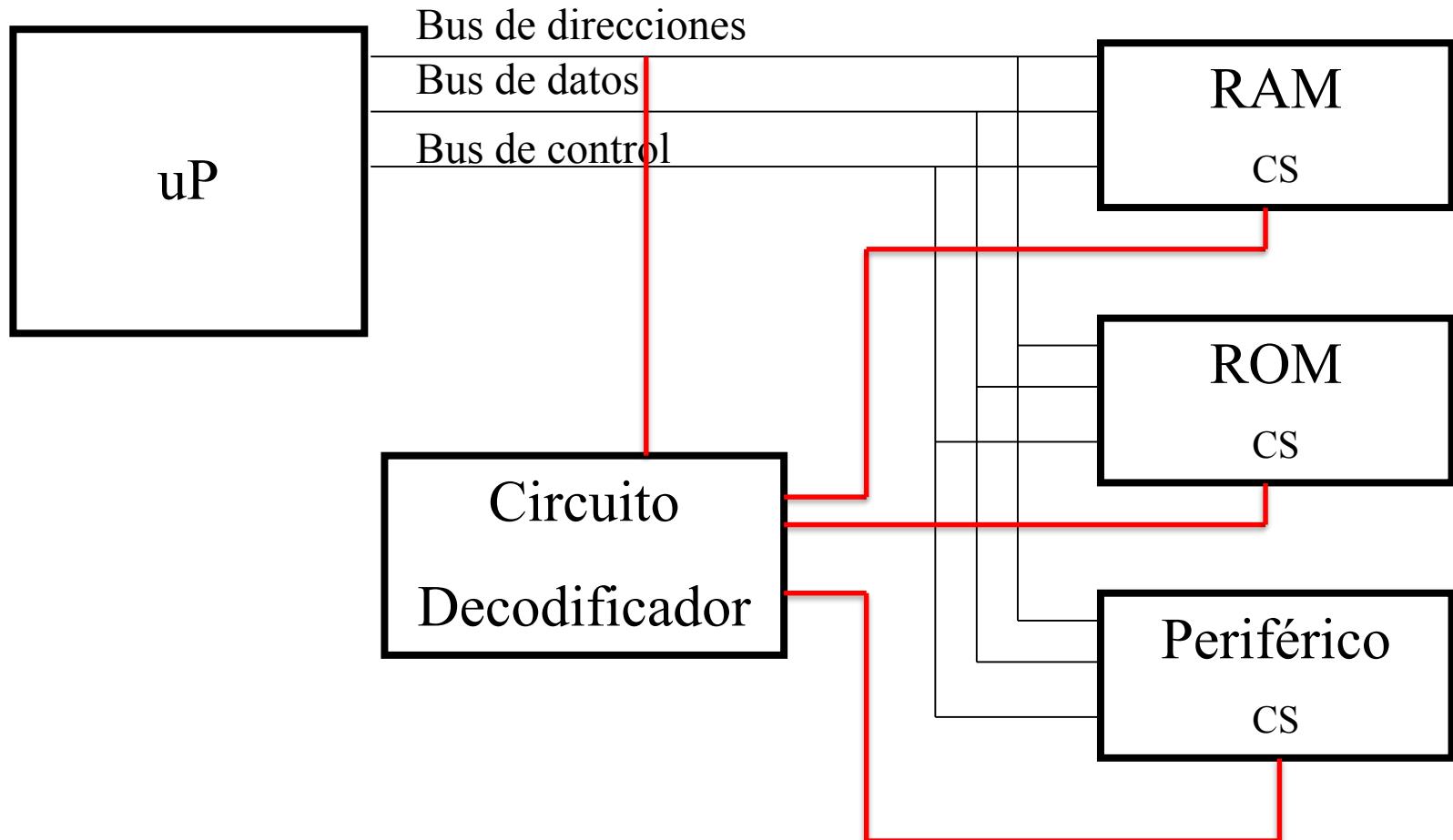
**/RD** (Cuando vale cero hay una lectura)

**IO/M** (Si vale 1: operaciones con ports, si vale 0: operaciones con la memoria)  $\Rightarrow$  ya no se usa

- $\text{IO/M} = 1 \Rightarrow$  mapa E/S  $\Rightarrow$  mismo tamaño que mapa de memoria  
Ej. IN ó OUT
- $\text{IO/M} = 0 \Rightarrow$  mapa de memoria Ej. mov

} se duplican los mapas

# Decodificación de hardware



# Decodificación de hardware

## Decodificación completa $\Rightarrow$ ninguna patrón de bus de address suelta

Se dice que una decodificación es completa cuando hay una relación biunívoca entre cada posición de memoria y cada dirección

## Decodificación incompleta

Por simplicidad o para minimizar la cantidad de componentes no se hacen llegar todas las líneas del bus de direcciones. Por lo tanto aparecen “imágenes”  $\Rightarrow$  se prenden distintos lugares de memoria

# EJ Decodificación de hardware

## Ejercicio de decodificación de Memoria

Se dispone de un microprocesador con 16 líneas de bus de direcciones y 8 líneas de bus de datos.

Se desea conectar dicho procesador con 4K x 8 de ROM para el programa, 2K x 8 de RAM para los datos

Realice la decodificación completa, sin imágenes, de éste sistema.

# Sistema de Entrada y Salida

**E/S aislada** ➔ usar IN y OUT para acceder al mapa E/S

Una señal especial del micro indica la ejecución de una operación de E/S.

## Interrupción

Una señal externa interrumpe al micro para requerir un servicio de atención

## Acceso directo a memoria (DMA)

La información se transfiere directamente a la memoria, no requiere de intervención del CPU

**Mapeo en memoria** ➔ poner referencias en mapa de memoria

Se le otorga un sector de memoria principal al dispositivo

# Periférico Estándar

⇒ Un periférico no necesita muchos registros

8 bits de datos

4 registros internos de 8 bits

↳ Solo necesita 4 direcciones  
de memoria

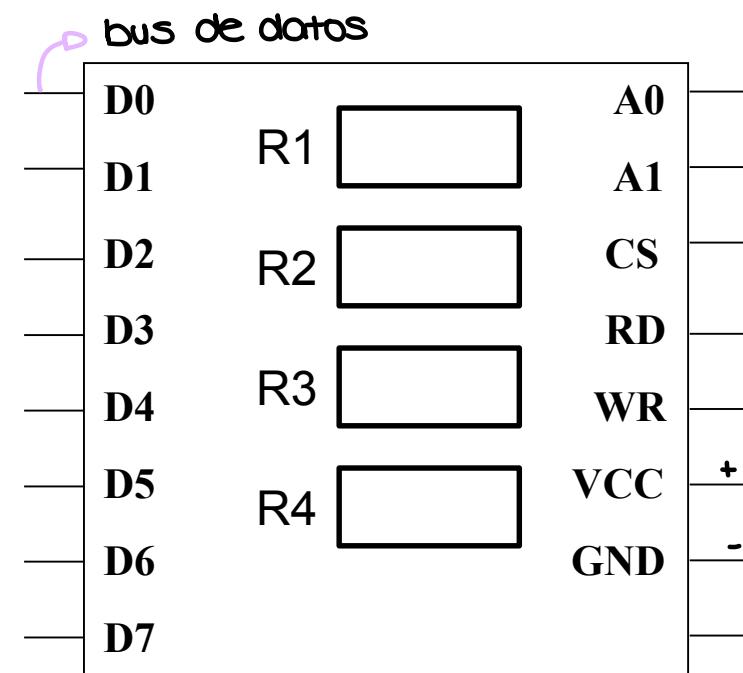
¿ Como se decodifica ?

⇒ Se decodifica igual que RAM/ROM

Obs:

El teclado solo ocupa la direc 60h

⇒ La tecla que queremos leer



## MAPA DE MEMORIA

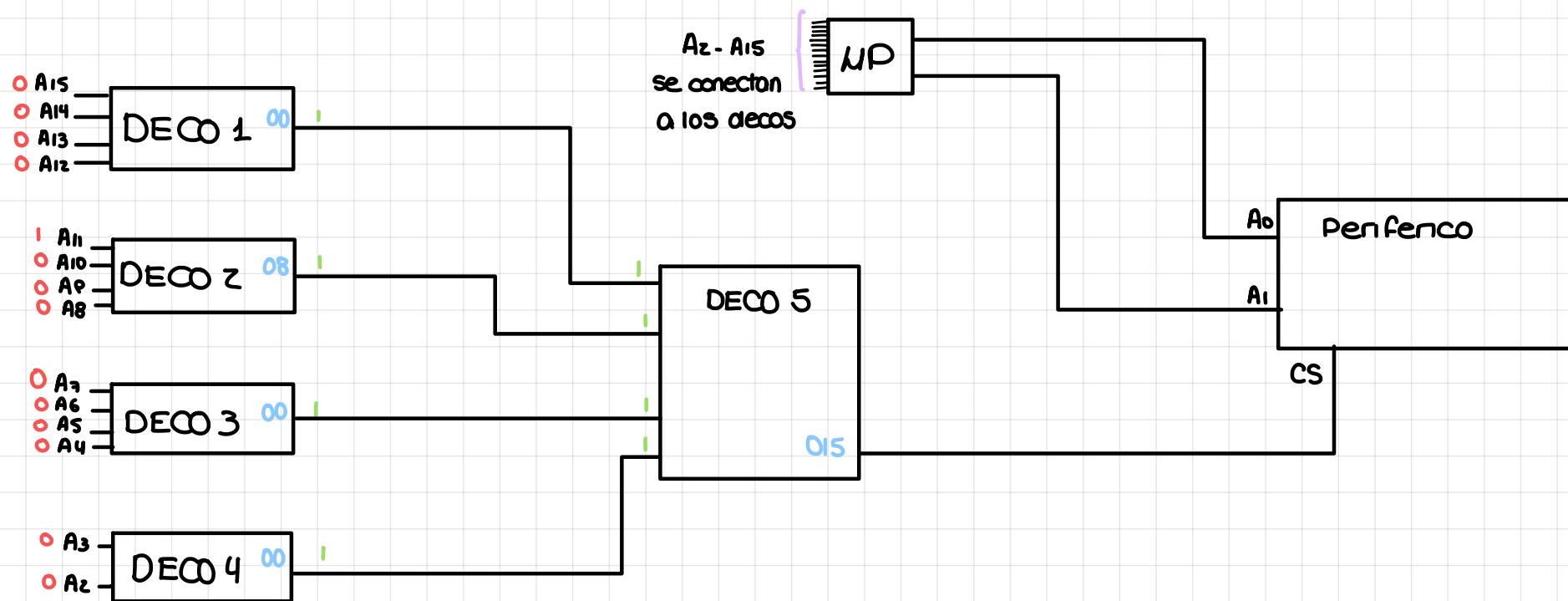


$0800\text{h} = 0000100000000000\text{b}$

$0803\text{h} = 0000100000000011\text{b}$

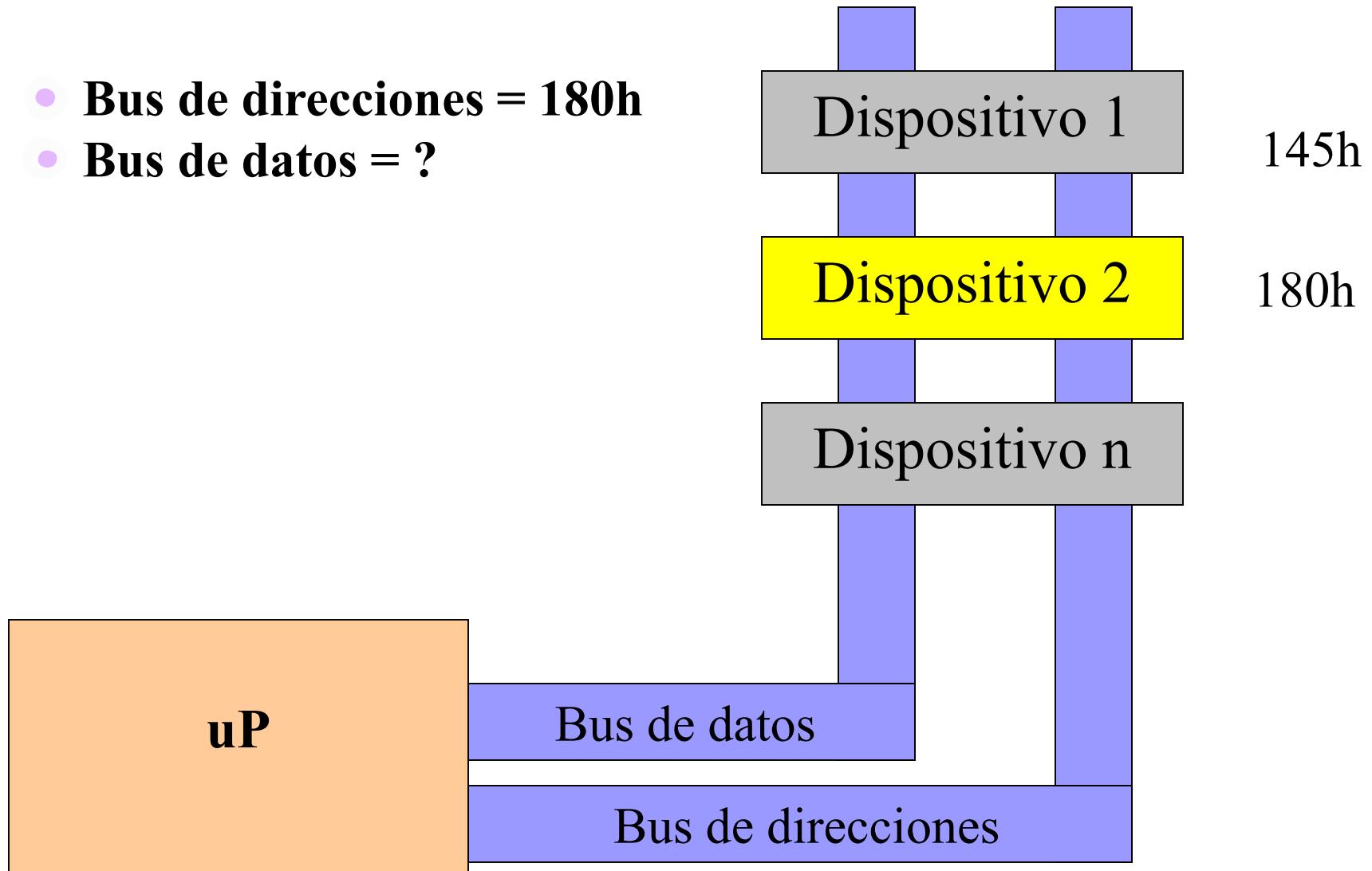
⇒ tienen los primeros 14 bits más significativos en común

NO existe deco de 14 entradas ⇒ tomamos varios decos



# Sistema de E/S

- Bus de direcciones = 180h
- Bus de datos = ?



# EJ Decodificación de hardware

## Ejercicio de decodificación de E/S

Se dispone de un microprocesador con 16 líneas de bus de direcciones y 8 líneas de bus de datos.

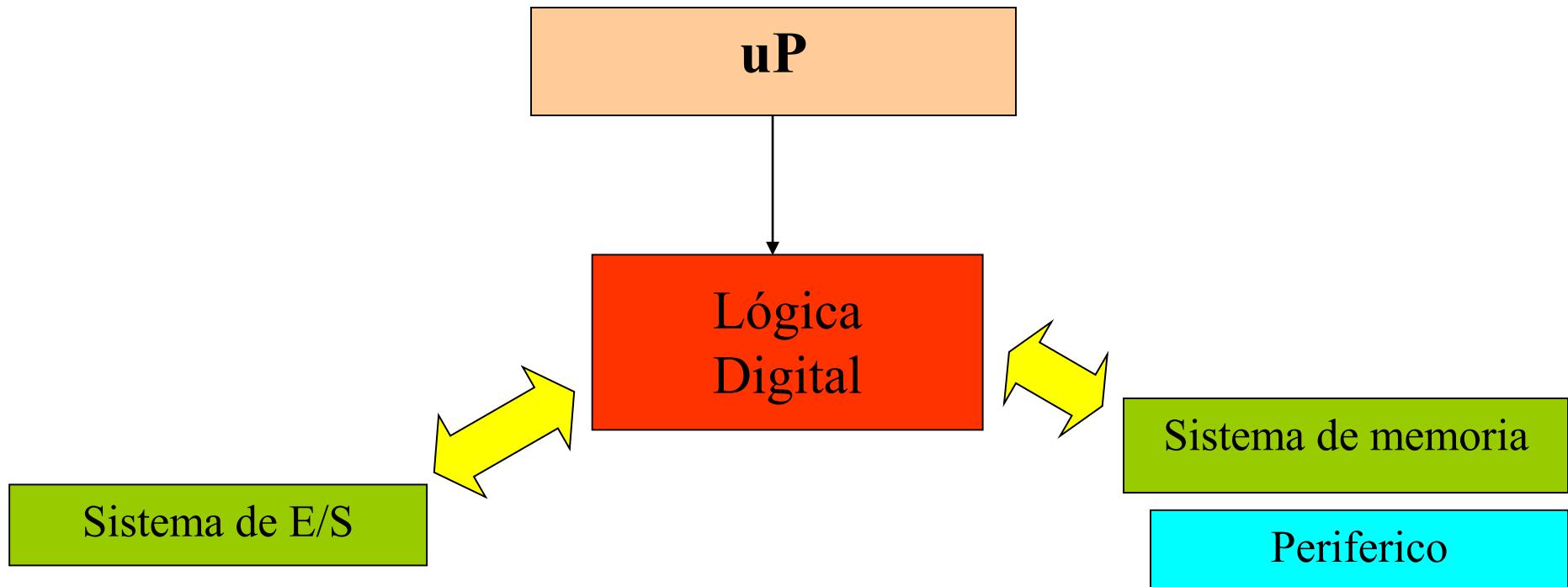
Se desea agregar a la decodificación anterior un periférico que posee 7 registros, los cuales deben poder ser escritos y leídos.

Realice la decodificación completa, sin imágenes, de éste sistema.

# Mapeo en memoria

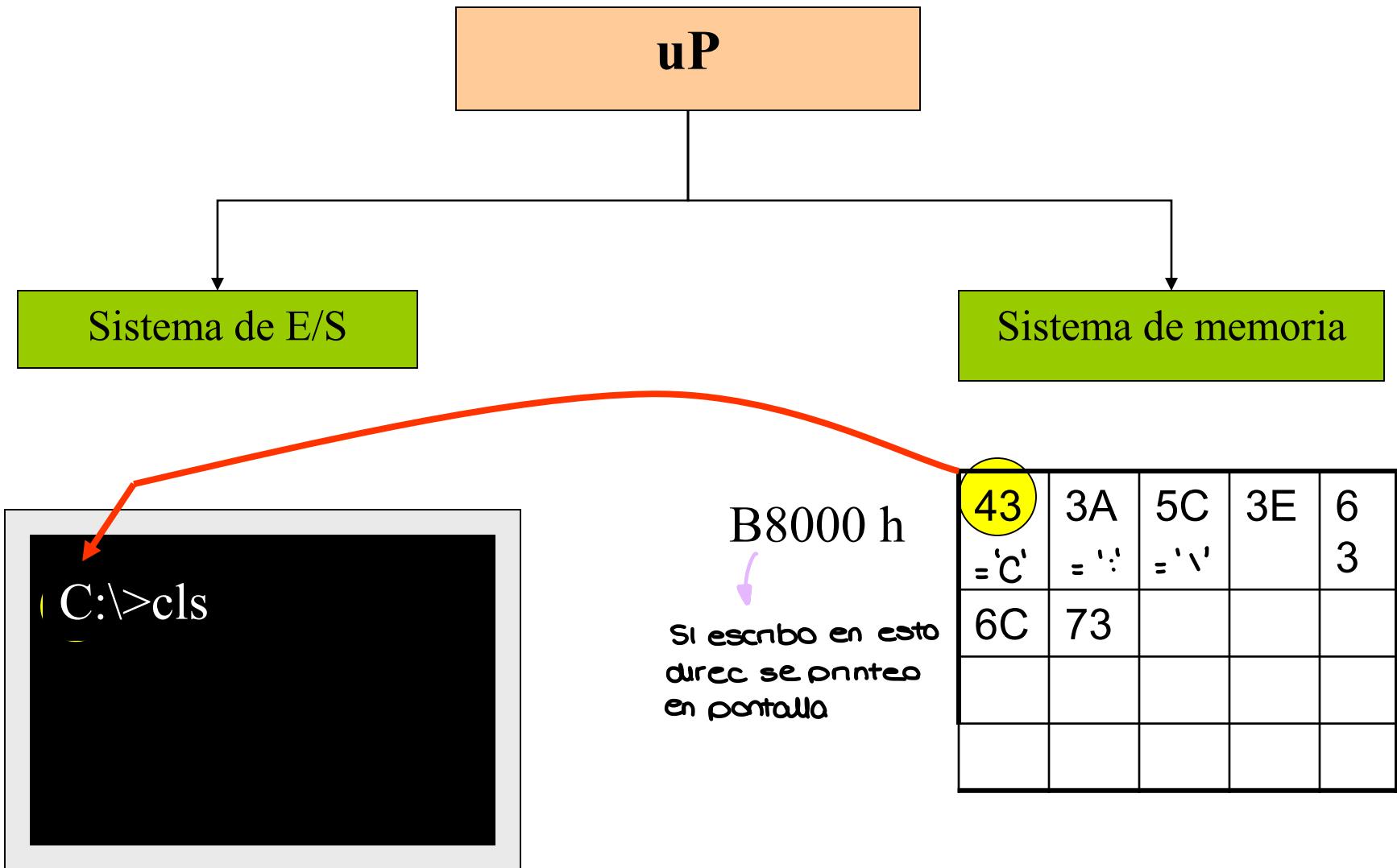
- Un dispositivo es manejado como una o varias posiciones de memoria.
- Cada posición de memoria puede ser un registro interno diferente del dispositivo.
  - *se pueden usar add, sub, mov, etc*
- Ventaja: Se utilizan las instrucciones de acceso a memoria, por lo tanto las alternativas de programación son mayores por tener mayor cantidad de instrucciones para manejo de memoria.
- Ventaja: Se modifica directamente los registros del periférico sin necesidad de obtener el valor con las instrucciones IN y OUT.
- Desventaja: Reduce cantidad de memoria. El impacto es mínimo relacionado con la cantidad de memoria que tienen las PC actuales.

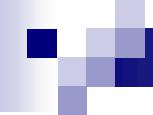
# Mapeo en memoria



Se decodifica al periferico en el mapa de memoria, cambiando la logica digital.

## Mapeo en memoria ( ej Video )





# Interrupciones

# Sistema de Entrada y Salida

## **Interrupción**

Una señal externa interrumpe al micro para requerir un servicio de atención.

## **E/S aislada**

Una señal especial del micro indica la ejecución de una operación de E/S.

## **Acceso directo a memoria (DMA)**

La información se transfiere directamente a la memoria, no requiere de intervención del CPU

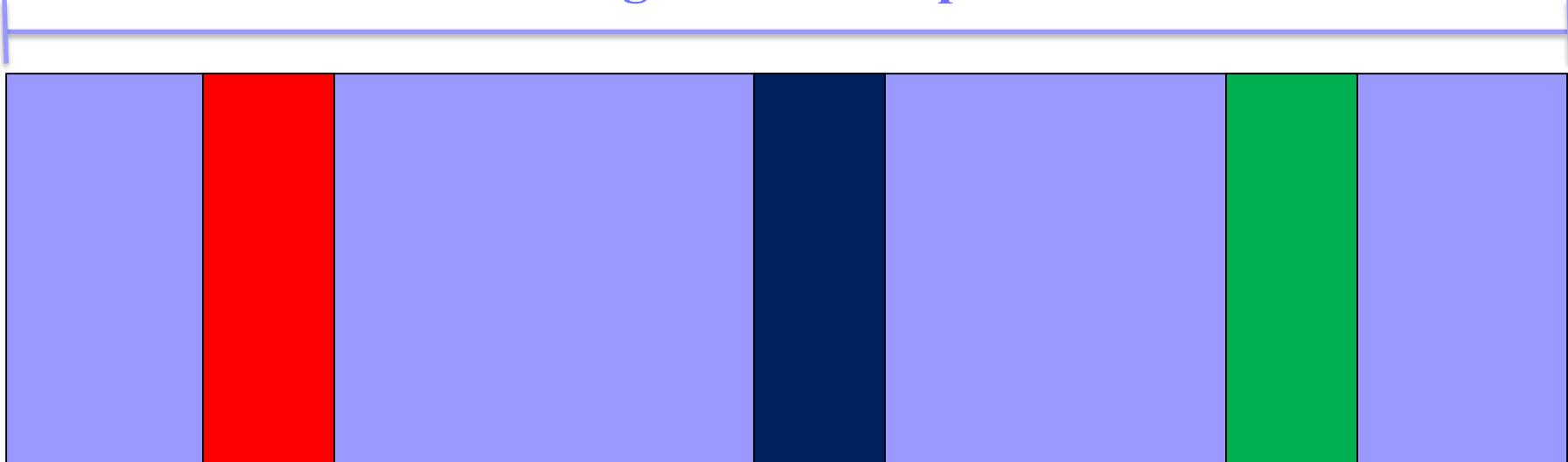
## **Mapeo en memoria**

Se le otorga un sector de memoria principal al dispositivo

# Interrupciones

⇒ un programa sufre muchas interrupciones

## Programa Principal



Interrupción  
de teclado

↓  
Corre el driver de teclado

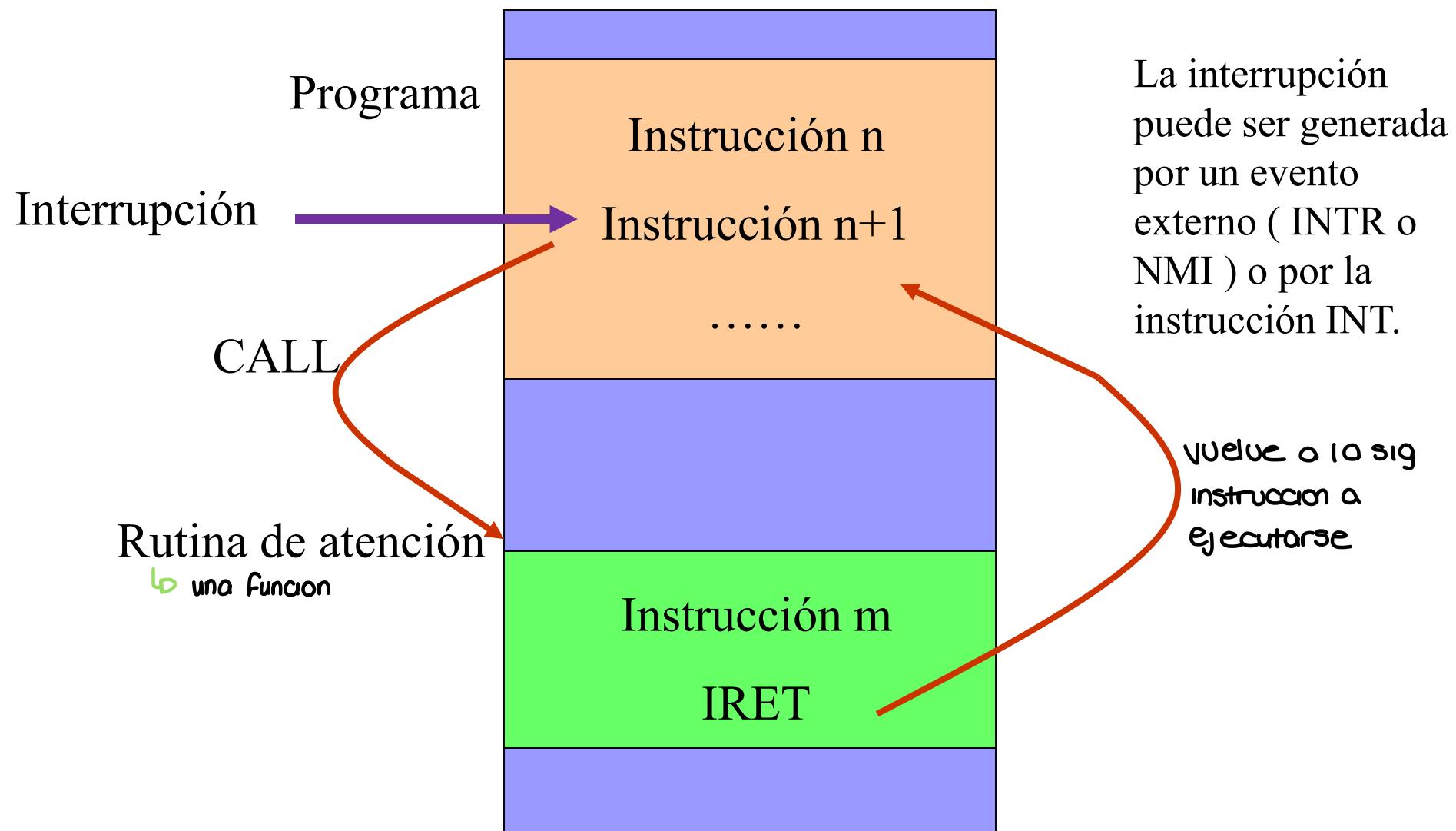
Interrupción  
de Mouse

Interrupción  
X

⇒ Por cada interrupción se corre una rutina de atención

t

# Rutina de atención de interrupción



# Interrupciones

## ❖ Interrupciones de Hardware

Se interrumpe la ejecución del programa activando alguna de las dos entradas que tiene el microprocesador ( INTR y NMI )

↳ si caen en 1 se  
interrumpe al μP

## ❖ Interrupciones de Software

Se interrumpe la ejecución del programa al ejecutar la instrucción de assembler INT. Por ejemplo INT 44h ( donde 44h es el numero de rutina de interrupción a ejecutar )

↳ 80h = entrada para las syscalls

# Interrupciones de Hardware

El flag IF indica si se debe atender a las interrupciones externas. Si IF=1 ( habilitado ) si IF=0 ( deshabilitado )

INTR son enmascarables

## ❖ Interrupción enmascarable

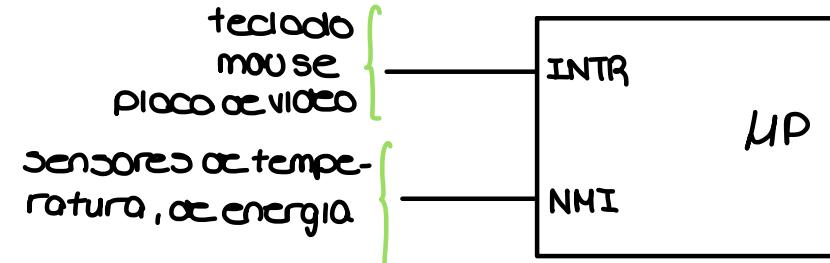
El flag IF se controla con las instrucciones *sti ( set interrupts )* y *cli ( clear interrupts )*.

desabilita

habilita

## ❖ Interrupción NO enmascarable

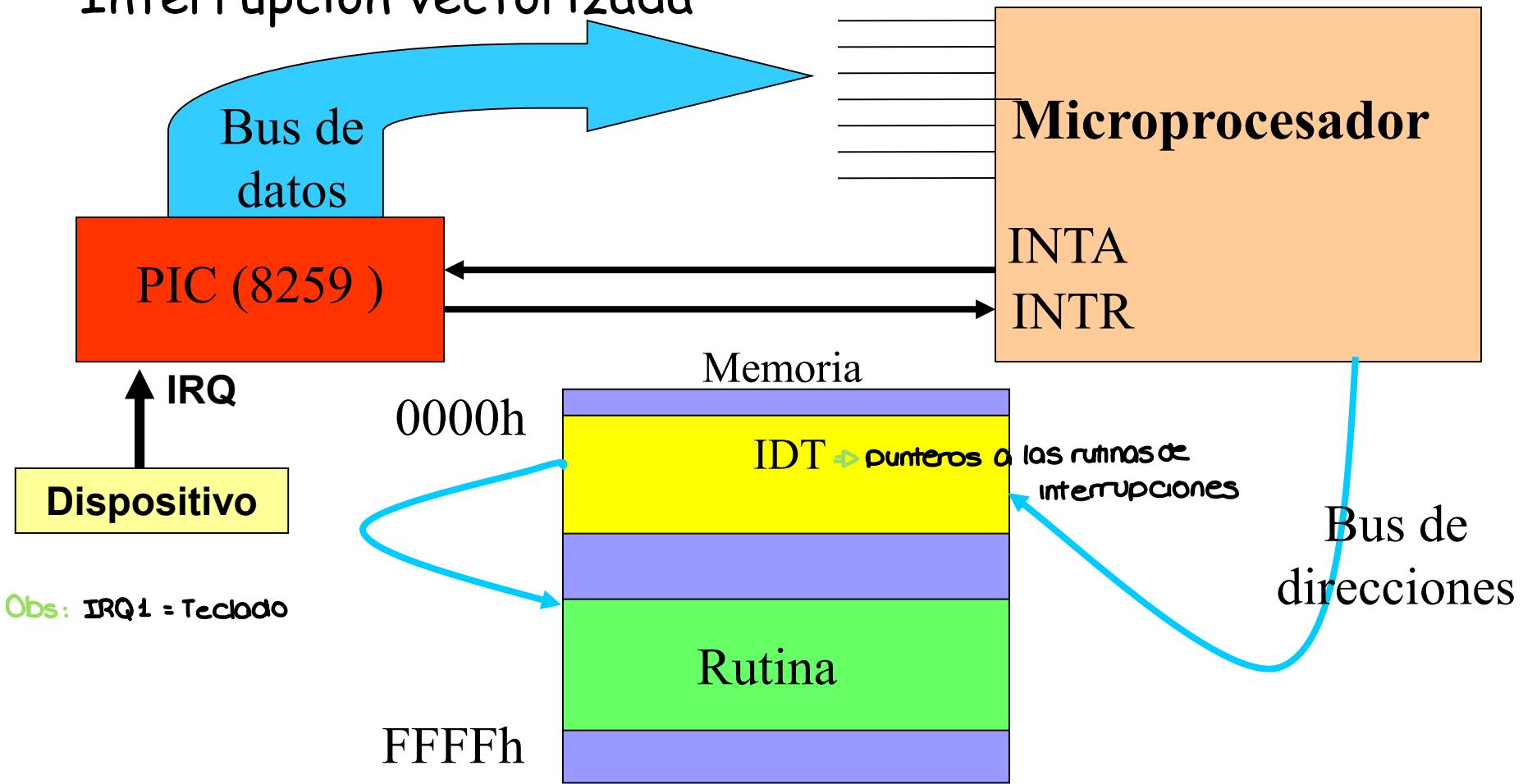
Las interrupciones que ingresan por la patita NMI no pueden ser enmascaradas. Y siempre ejecutan la rutina que se encuentra en la posición 2h del vector de interrupciones. ( INT 2h )



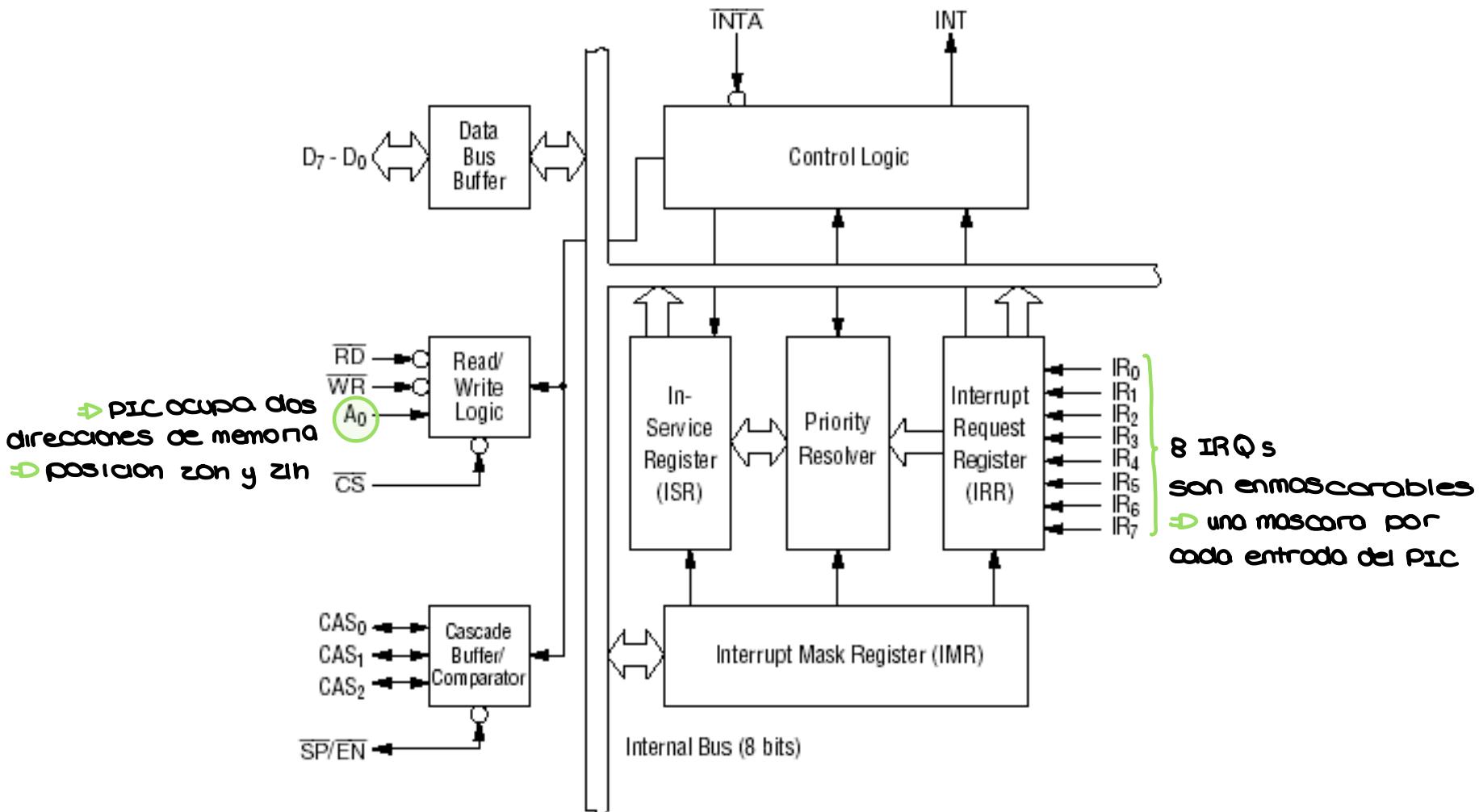
# PIC ( Controlador programable de interrupciones )

⇒ Varias líneas de entradas que se conectan a dispositivos  
↳ Se llaman IRQ

## Interrupción vectorizada



# PIC – Diagrama lógico



MSM82C59A-2 Internal Block Diagram

# PIC - IMR

Los puertos de entrada y salida en la PC son el 20h y el 21h.

El 20h se utiliza para programar el PIC ( lo utiliza el BIOS al arrancar el sistema )

En el puerto 21hs podemos acceder al registro IMR ( Interrupt Mask Register ) del 8259, donde podemos setear que interrupciones llegan al microprocesador y cuales no.

OCW1	A <sub>0</sub>	D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>
	1	M7	M6	M5	M4	M3	M2	M1	M0

Interrupt Mask  
1: Mask set  
0: Mask reset

# Acceso al PIC

In al, 21h ; leo mascara del PIC  
; 0 = mascara deshabilitada, por lo  
; tanto pasa la señal al microprocesador

Mov al,0FEh ; por ej. Solo habilito la interrupción de teclado

Out 21h, al

EOI ( End of interrupt )  $\Rightarrow$  avisa al PIC que ya se atendio la interrupcion

Cada rutina de atención de interrupción, luego de correr, debe avisar al PIC que terminó su ejecución.

Se puede especificar la IRQ que terminó o enviar un código que indica que finalizó la atención de la ultima interrupción que llegó.

Esa palabra se envía al puerto 20h y el valor que se envía para indicar que finalizó la atención de la interrupción es el valor 20h.

# PIC en cascada

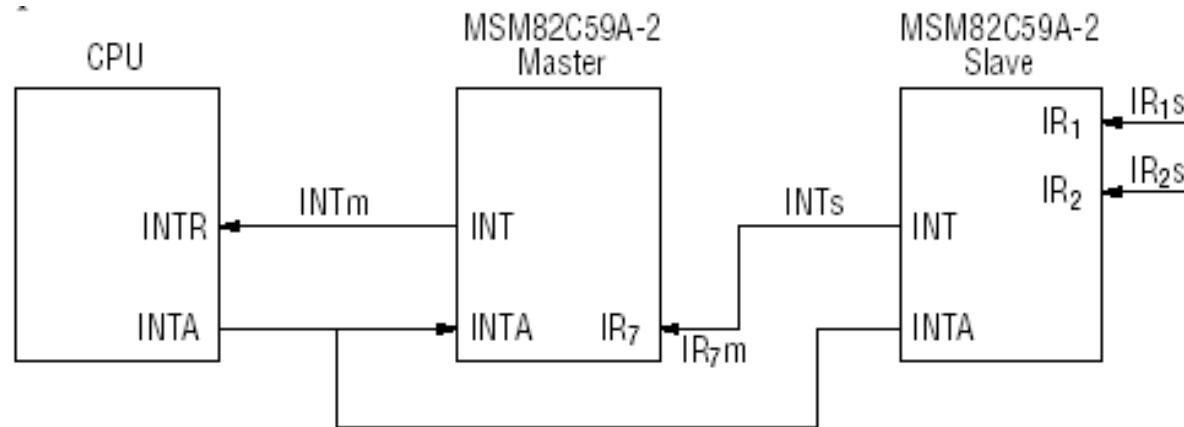


Fig. 1 System Configuration

Colocando 2 PICs en cascada se amplia la cantidad de interrupciones de hardware en la PC.

En la PC, se utiliza el IRQ2 del Master para conectar el Slave.

# Interrupciones de Software

Dentro de un código puedo realizar una interrupción con la instrucción INT

```
mov  Ax,0  
cmp  Ax,Bx  
INT  22h ↳ voy a lo tabla IDT y busco la entrada 22h  
Add  Cx,Bx  
Mov  [Cx],Bx
```

El micro accede al descriptor ubicado en la posición 22h de la IDT.

# Servicios de BIOS

↳ memoria ROM que viene en todos los PCs

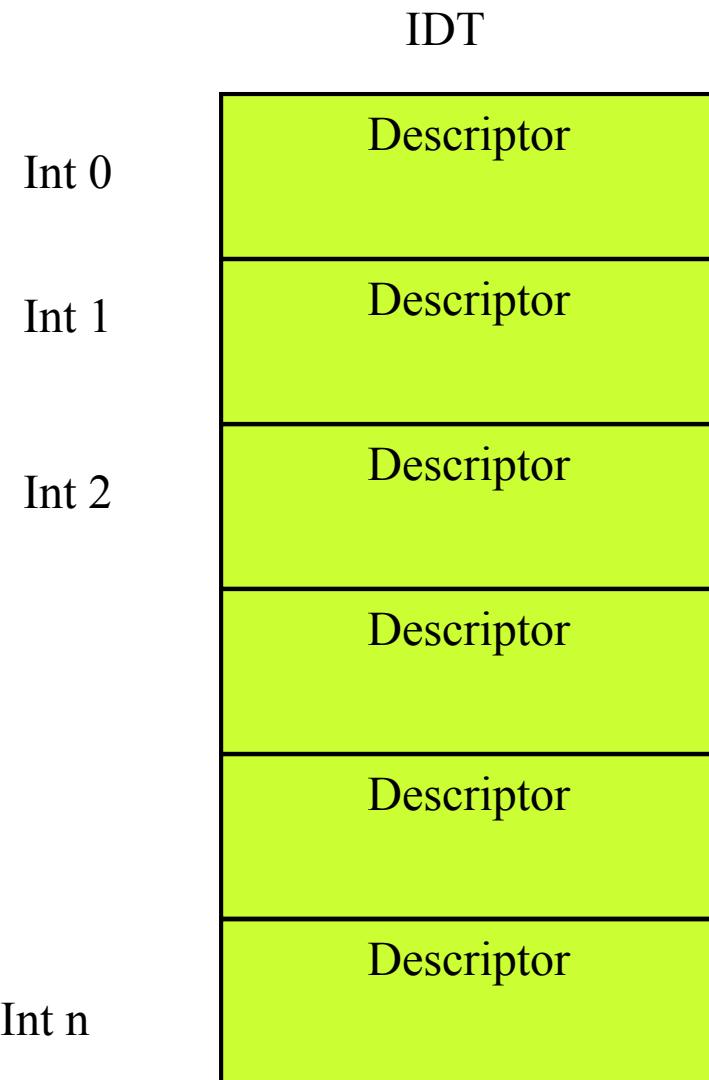
El BIOS al iniciar la PC guarda en memoria rutinas básicas para poder a empezar a operar.

INT 10h	Rutinas de video (BIOS)
INT 13h	Rutinas de disco (BIOS)
INT 14h	Rutinas para puerto Serie (BIOS)
INT 19h	Rutina para bootloader (BIOS)
INT 1Ah	Rutinas para el RTC (BIOS)

# Interrupciones de hardware por Default

Línea IRQ	INT Tipo	Descripción
IRQ0	08h	Timmer tick (18,2 veces por seg.)  cada 55 ms
IRQ1	09h	Teclado
IRQ2	0Ah	INT desde 8259A esclavo
IRQ8	70h	Servicio de reloj en tiempo real.
IRQ9	71h	Redireccionamiento por soft. a IRQ2
IRQ10	72h	Reservada
IRQ11	73h	Reservada
IRQ12	74h	Reservada.
IRQ13	75h	Coprocesador numérico.
IRQ14	76h	Controlador de disco rígido.
IRQ15	77h	Reservada.
IRQ3	0Bh	COM2
IRQ4	0Ch	COM1
IRQ5	0Dh	LPT2
IRQ6	0Eh	FLOPPY
IRQ7	0Fh	LPT1

# Interrupciones en Modo Protegido



En modo protegido cada entrada de la IDT es un descriptor de Interrupción, contiene además de las dirección de la rutina de atención de interrupción otros datos como los permisos.

Los primeros 32 descriptores son las Excepciones

# Excepciones

➡ NO las genera un perfecho, las genera el procesador ➡ se interrumpe a si mismo

Una Excepción es un evento **generado por el procesador** cuando detecta una o mas condiciones predefinidas al ejecutar una instrucción.

Existen 3 tipos de excepciones:

- Faults : Excepción que puede corregirse. El procesador guarda en la pila la dirección de la instrucción que produjo la falla.
- Trap : Se utilizan para realizar accesos al sistema operativo.
- Abort: No siempre se puede obtener la instrucción que causó la excepción. Reporta errores severos.

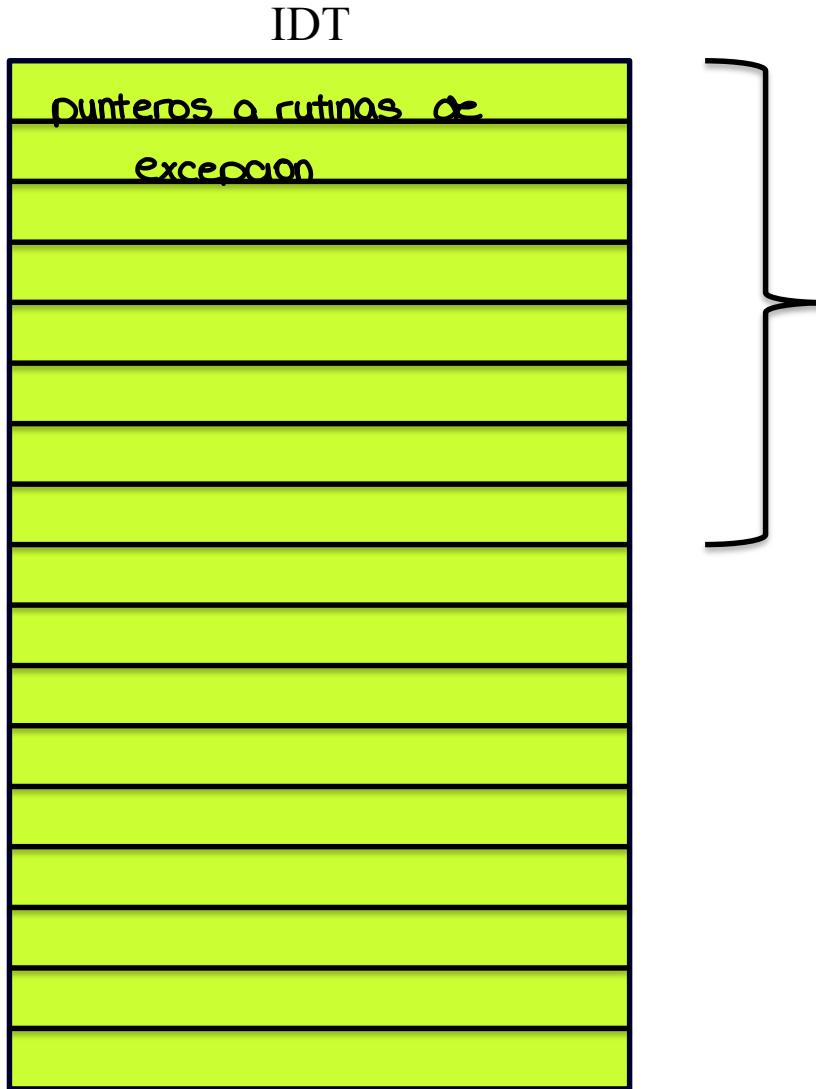
De la tabla de interrupciones ( IDT ) las primeras 32, son las excepciones.

Actualmente se están utilizando 20 de ellas y el resto quedan disponibles para uso futuro.

# Excepciones

Id	Description
0	Divide error ➔ odo el μP ve que un prog intenta dividir por cero ➔ genera una excepcion
1	Debug exceptions
2	Nonmaskable interrupt
3	Breakpoint (one-byte INT 3 instruction)
4	Overflow (INTO instruction)
5	Bounds check (BOUND instruction)
6	Invalid opcode
7	Coprocessor not available
8	Double fault
9	(reserved)
10	Invalid TSS
11	Segment not present
12	Stack exception
13	General protection
➔ el μP verifica que el programa no se pase del segmento asignado	
14	Page fault
15	(reserved)
16	Coprocessor error
17-31	(reserved)
32-255	Available for external interrupts via INTR pin

# IDT - Excepciones

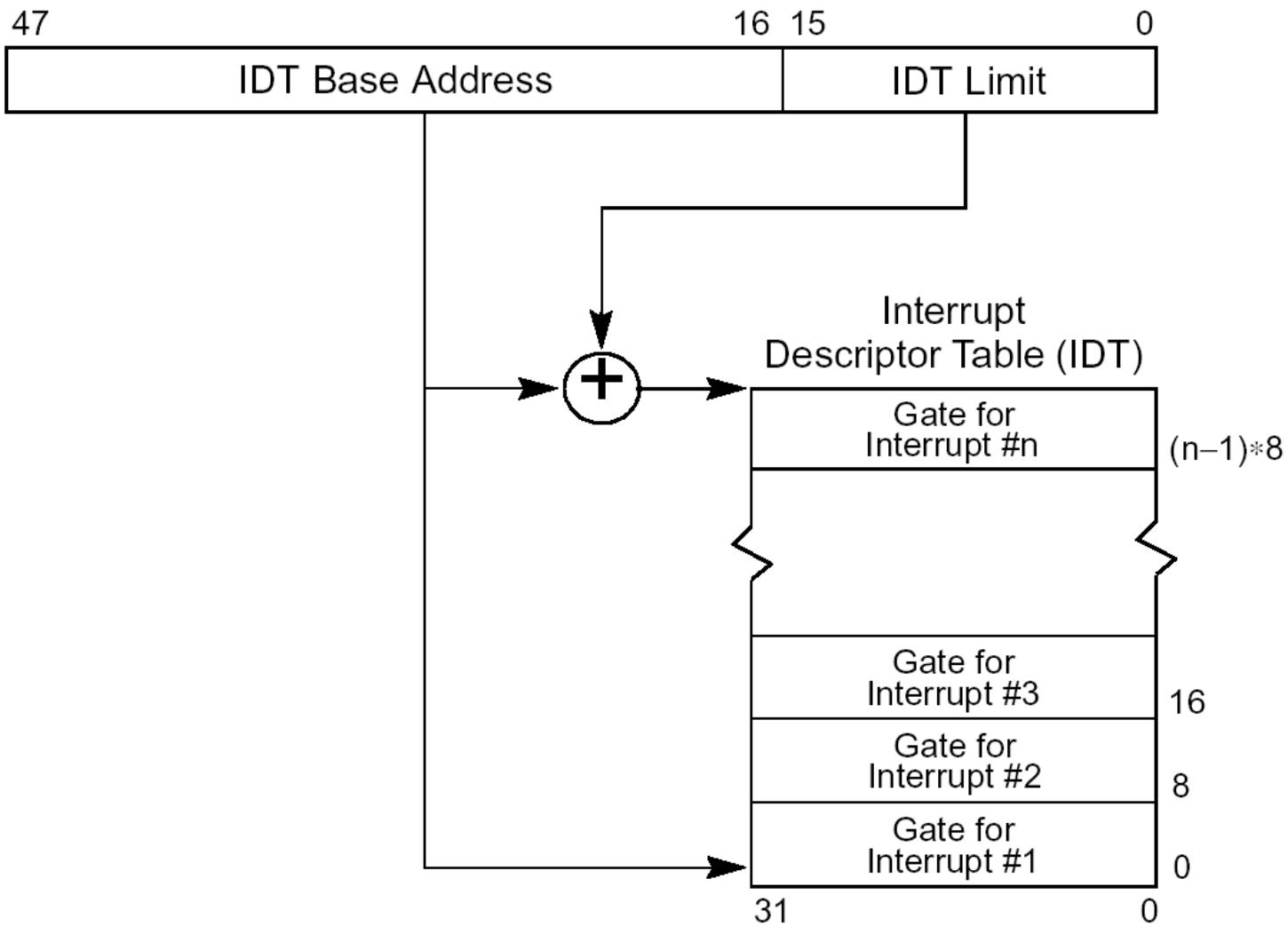


Las primeras 32 son excepciones

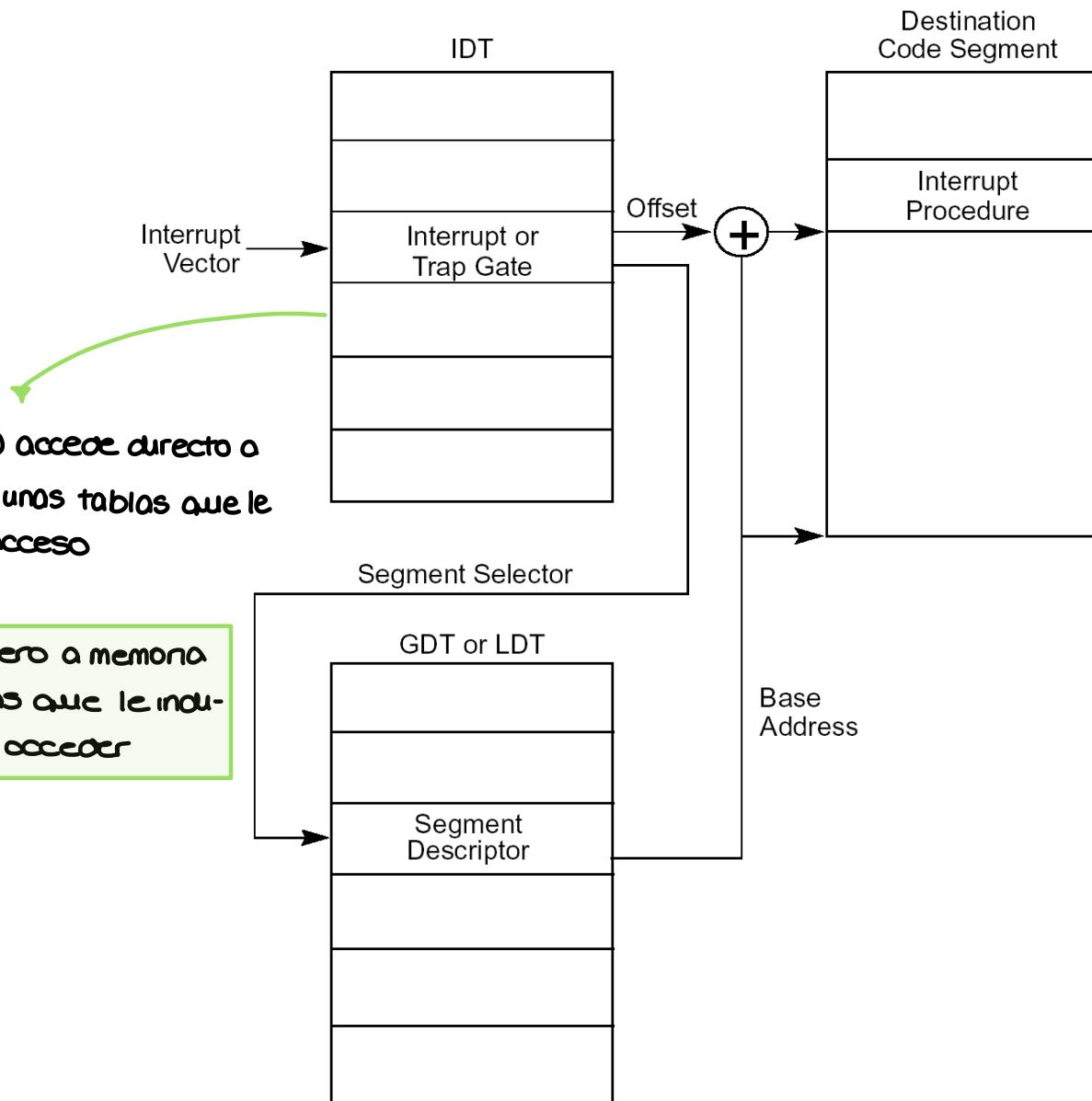
# IDTR

Puntero a la tabla

IDTR Register



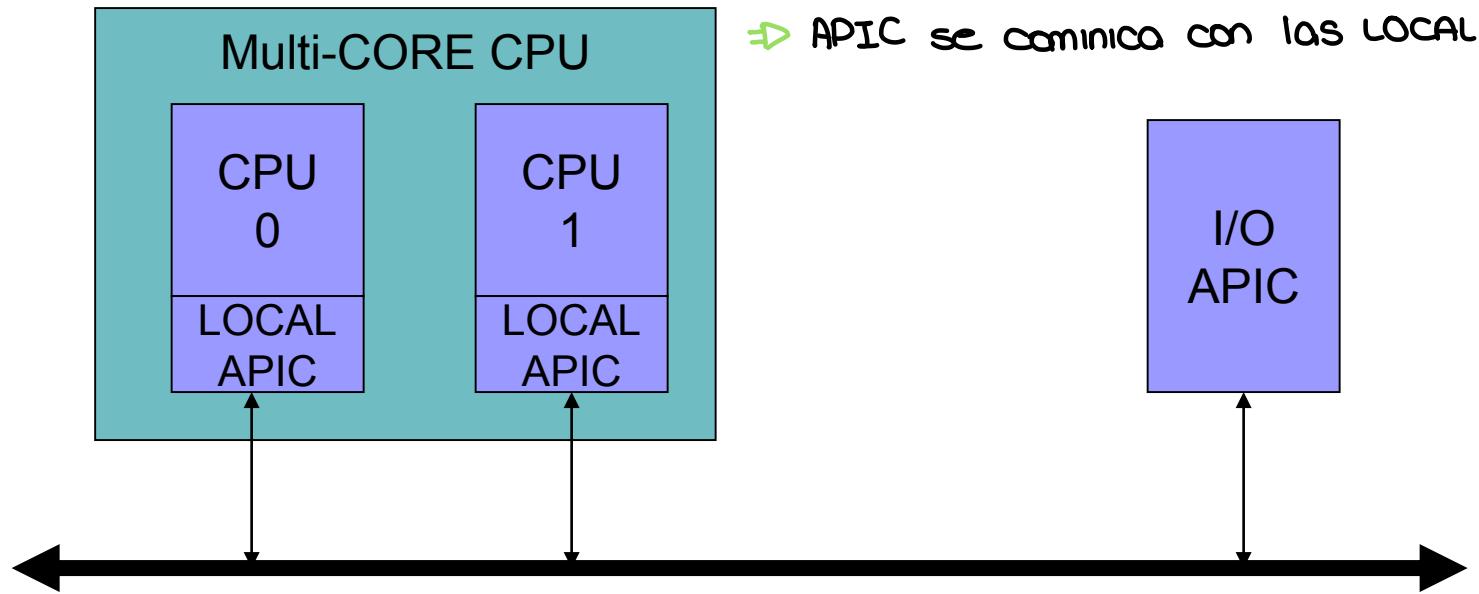
# Interrupciones en Modo Protegido



El puntero NO accede directo a mem , pasa por unos tablos que le indican si tiene acceso

cualquier puntero a memoria  
pasa por tablas que le indica-  
on si pueden acceder

# Interrupciones en Multi-Core

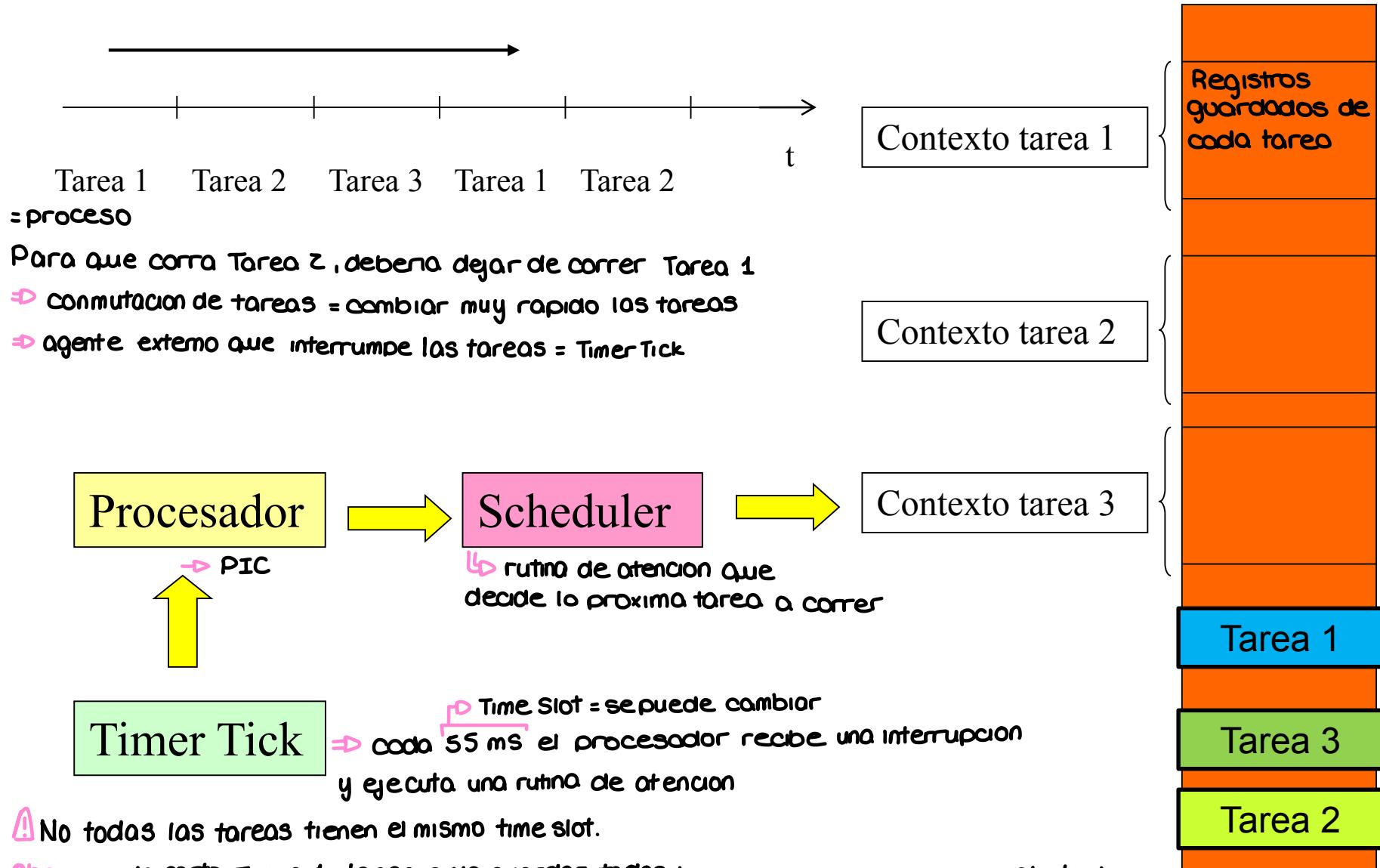


El AIPC (Advanced Programmable Interrupt Controller) realiza el “ruteo” o direccionamiento de los perifericos a las CPU



# Modo Protegido

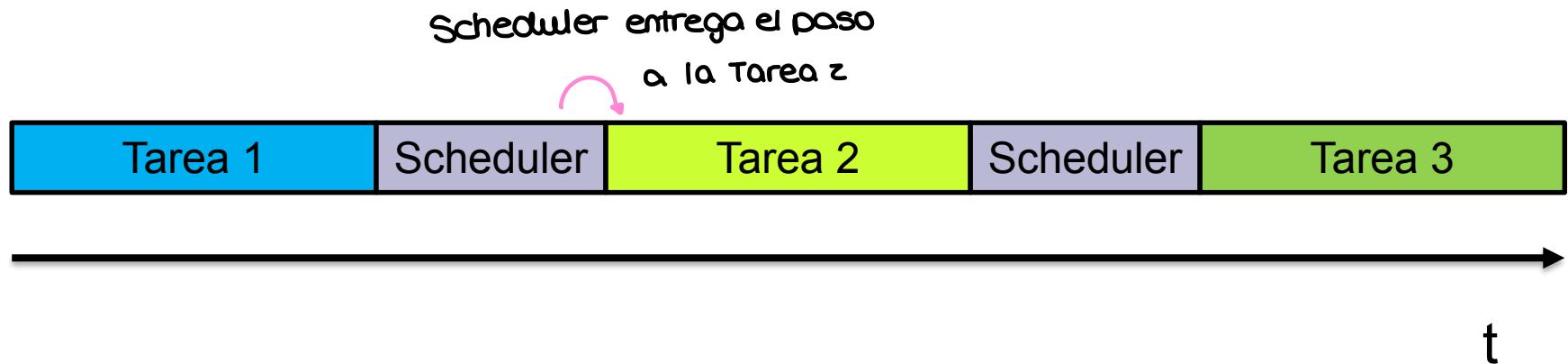
# Modo protegido-Commutación de tareas



⚠ No todas las tareas tienen el mismo time slot.

Obs: cuando corto Tarea 1 tengo que guardar todos los registros en memoria ⇒ este backup se denomina contexto de tarea

# Modo protegido-Commutación de tareas



¿El SO es una tarea ?

⚠️⚠️ Cuando corre una tarea, el SO no esta corriendo  $\Rightarrow$  el SO tambien es una tarea PERO el es quien elige que tarea sigue.

# Modo protegido- Protección de tareas

Una de las cosas a proteger: que la Tarea 1 sale de su zona de memoria y escribe otras zonas de memoria que no le corresponde

⇒ lo soluciona el procesador

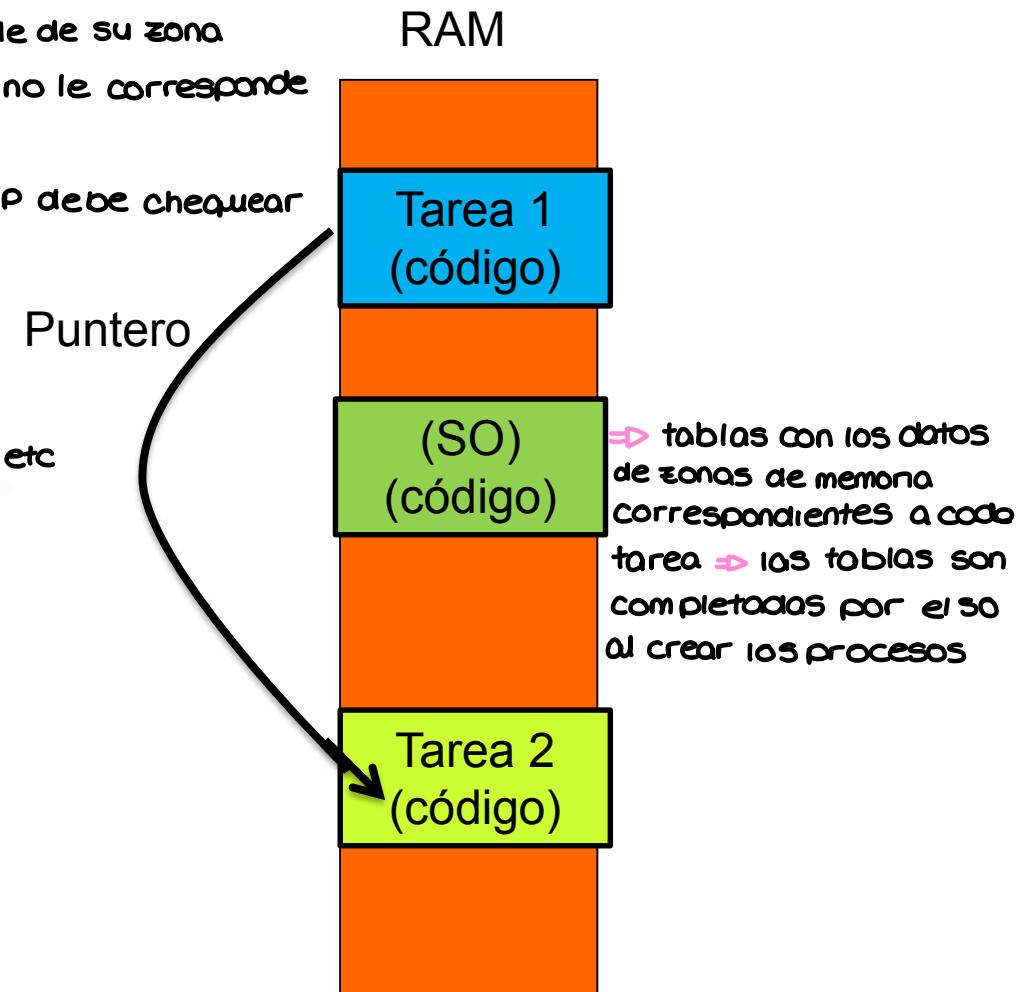
⇒ cada vez que ocurre un acceso a memoria, el MP debe chequear que la memoria a acceder sea valida

Si la zona es invalida ⇒ genera una excepcion

⇒ va a la tabla de interrupciones (IDT) y busco el puntero a la rutina de excepcion

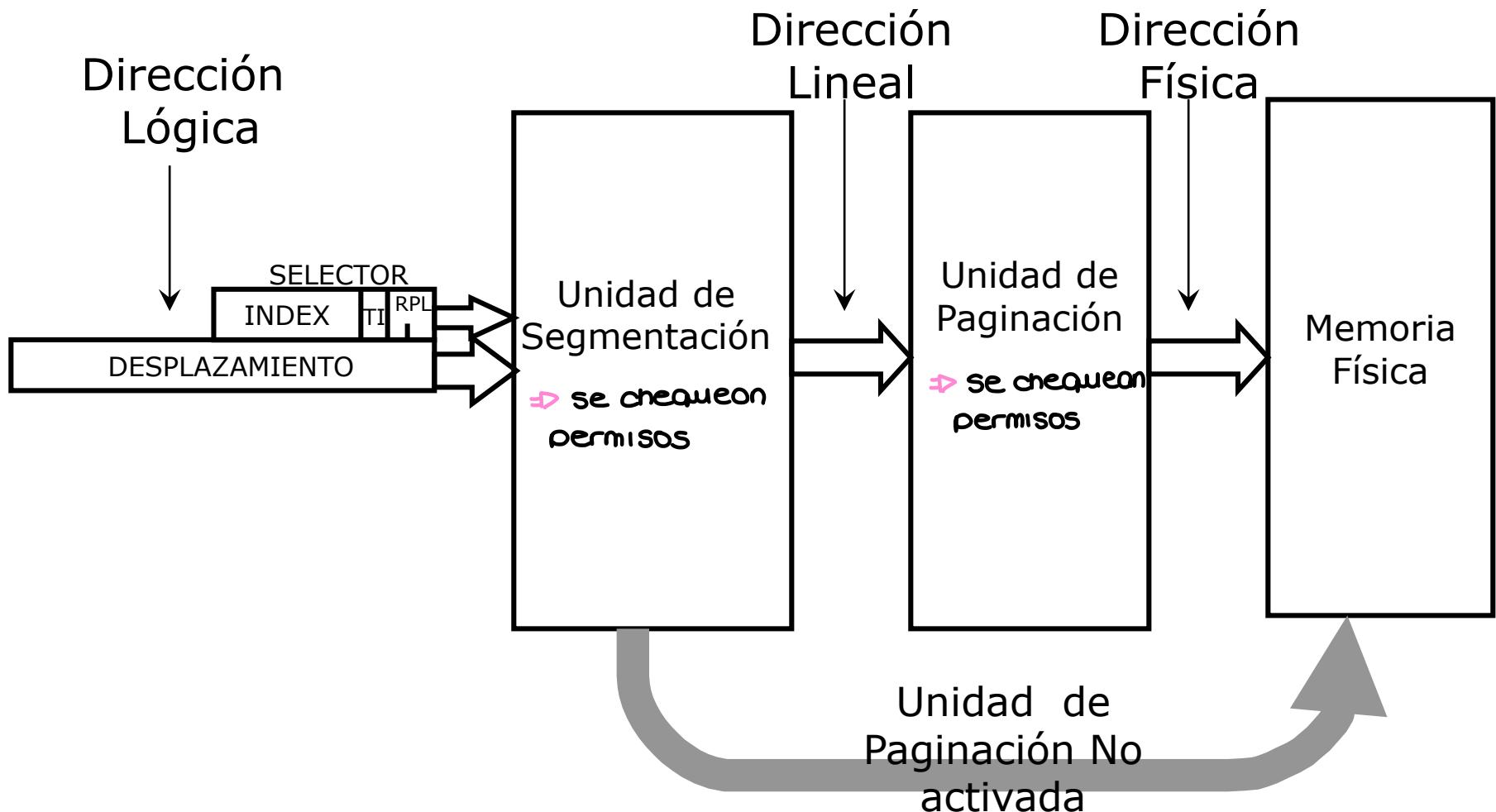
(de ahí puede cortar el proceso, lanzar excepcion, etc)

⇒ depende del SO



¿Como puede evitar el SO este acceso?

# Modo protegido-MMU



# Modo protegido-MMU

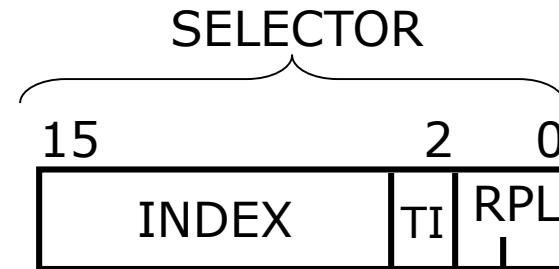
- La unidad de segmentación **NO** se puede deshabilitar.
- La unidad de paginación **SI** se puede deshabilitar.

*Debido a esta limitación de los procesadores Intel, algunos sistemas operativos eligen utilizar la memoria en modo “flat” (se verá luego) para no utilizar las reglas de segmentación pero si las de paginación.*

# Dirección lógica

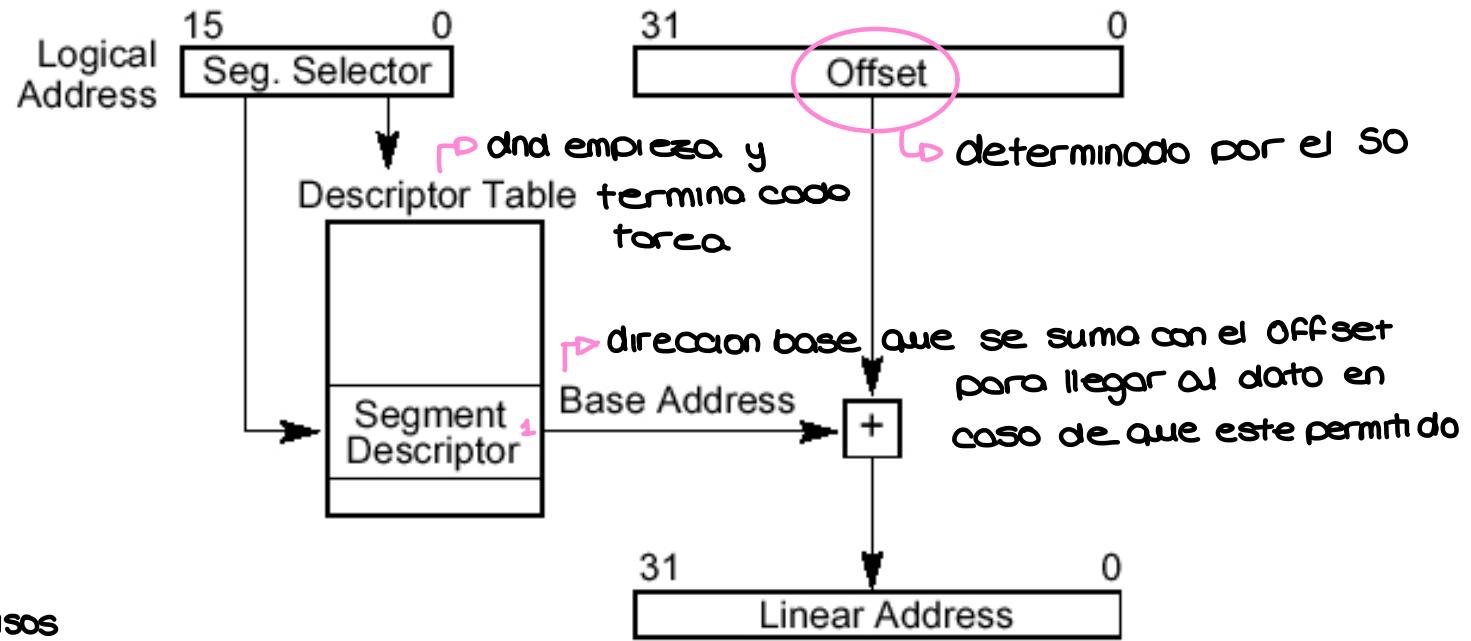
Ejemplo: DS = 20h y Offset = 1000h

Mov DS:[1000h], EAX



REGISTRO DE SEGMENTO  
(CS, DS, ES, SS, FS, o GS)

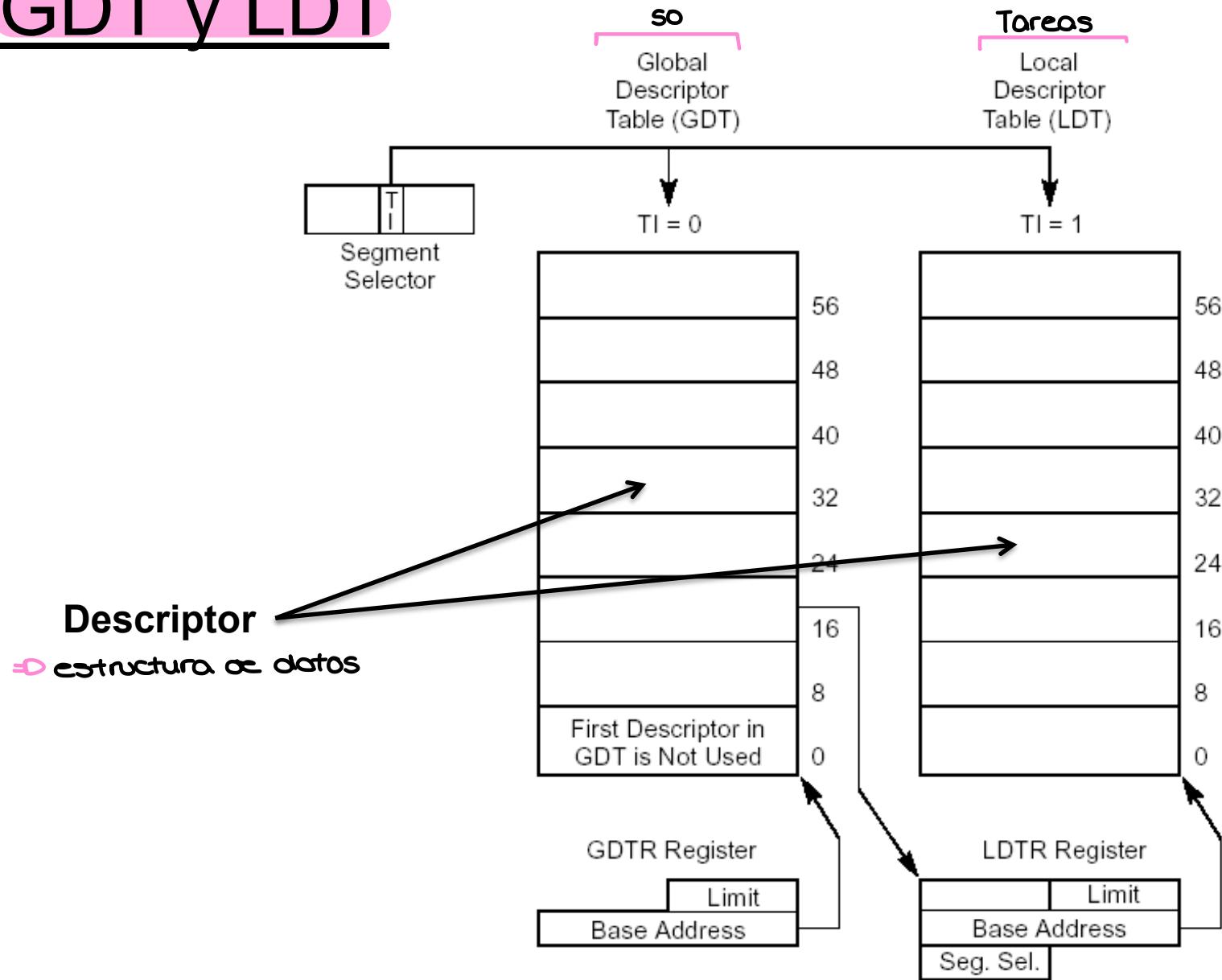
# Dirección lógica a lineal



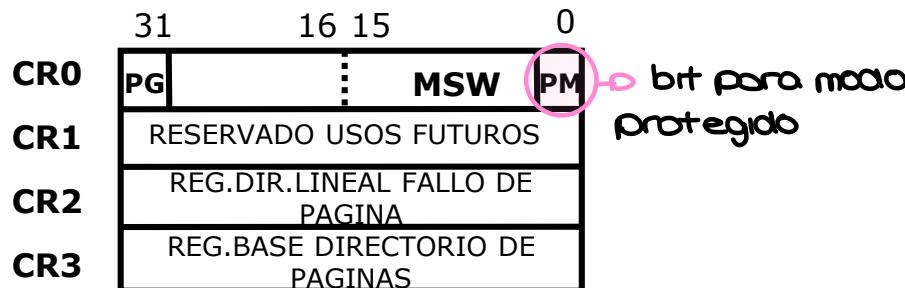
Ejemplo: DS = 20h y Offset = 1000h

Mov DS:[1000h], EAX

# GDT y LDT



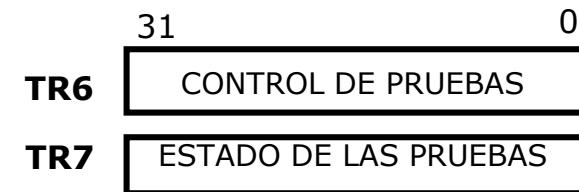
# Registros



REGISTROS DE CONTROL



REGISTROS DE DEBUGGING



REGISTROS DE PRUEBA DE LA TLB

# Descriptores

Veamos la estructura genérica de los descriptores de sistemas

31										16				0		
SEGMENT BASE 15...0										SEGMENT LIMIT 15...0				0		
BASE 31...24		G	0	0	0	LIMIT	2	P	DPL	3	0	TYPE	BASE	1	+4	
Type								Type	Defines							
0	Invalid							8	Invalid							
1	Available 80286 TSS							9	Available Intel386™ DX TSS							
2	LDT							A	Undefined (Intel Reserved)							
3	Busy 80286 TSS							B	Busy Intel386™ DX TSS							
4	80286 Call Gate							C	Intel386™ DX Call Gate							
5	Task Gate (for 80286 or Intel386™ DX Task)							D	Undefined (Intel Reserved)							
6	80286 Interrupt Gate							E	Intel386™ DX Interrupt Gate							
7	80286 Trap Gate							F	Intel386™ DX Trap Gate							

## NOTE:

In a maximum-size segment (ie. a segment with G = 1 and segment limit 19..0 = FFFFH), the lowest 12 bits of the segment base should be zero (ie. segment base 11...000 = 000H).

<sup>1</sup> Donde empieza la tarea

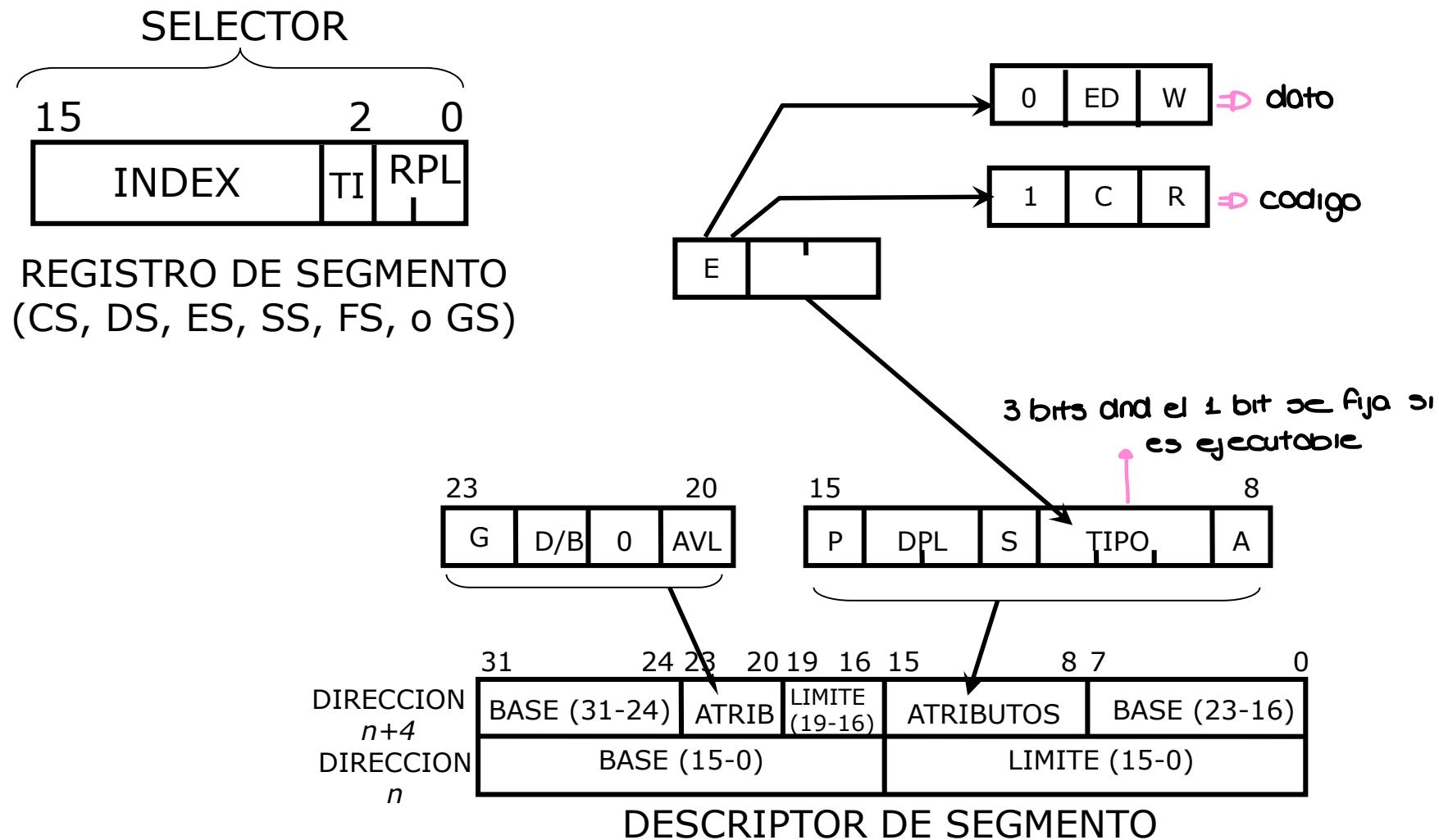
<sup>2</sup> Cheque si el offset pasa el límite

<sup>3</sup> Nivel de privilegio de la tarea  $\Rightarrow$  4 niveles  $\Rightarrow$  00, 01, 10, 11

menor nivel  
↑  
mayor nivel

# Descriptores de segmento

■ protección por  
WIR, por ejecución  
y por límite



# Atributos

- **P** (Bit de presencia)
  - Si P=1 el segmento esta cargado en la memoria física.
  - Si P=0 es segmento esta ausente.
- **DPL** (Nivel de privilegio)
  - Indica el nivel de privilegio del segmento
  - 0 = máximo privilegio, 3=minimo privilegio
- **S** (tipo de segmento)
  - Si S=1 se trata de un segmento normal (código datos o pila)
  - Si S=0 se trata de un segmento de sistema (puerta de llamada, TSS, etc).

# Atributos

- **Tipo:** Es un campo de tres bits.
  - En el caso de un segmento normal
    - Si E: Ejecutable es 1, se trata de un segmento ejecutable
    - El siguiente bit (C: Ajustable o conforming) define si el segmento al ser accedido cambia su nivel de privilegio al nivel de privilegio de segmento que lo llamo (C=1)
    - El ultimo bit (R: Leible) define si el segmento de código puede ser leído (R=1).
  - Si el bit E es 0 se trata de un segmento de datos
    - El segundo bit (ED: Expansión decreciente) define si se trata de un segmento de datos normal cuando ED=0, o de un segmento de pila cuando ED=1.
    - El tercer bit (W: Escribible) define si el segmento se puede escribir cuando W=1 o si es de solo lectura cuando W=0.

# Dirección Lineal

Ejemplo: DS = 20h y Offset = 1000h

Mov DS:[1000h], EAX

DS= 0020h = **0000 0000 0010 0000** b

- Accede al Descriptor nro 4 de la GDT.
- Obtiene dirección base.
- (ej) Dirección base= 5600 0000h
- Luego suma offset de 1000 h
- Dirección lineal 5600 1000h

# Descriptores

Cuando el bit S=0 se lo denomina “descriptor de sistema”.

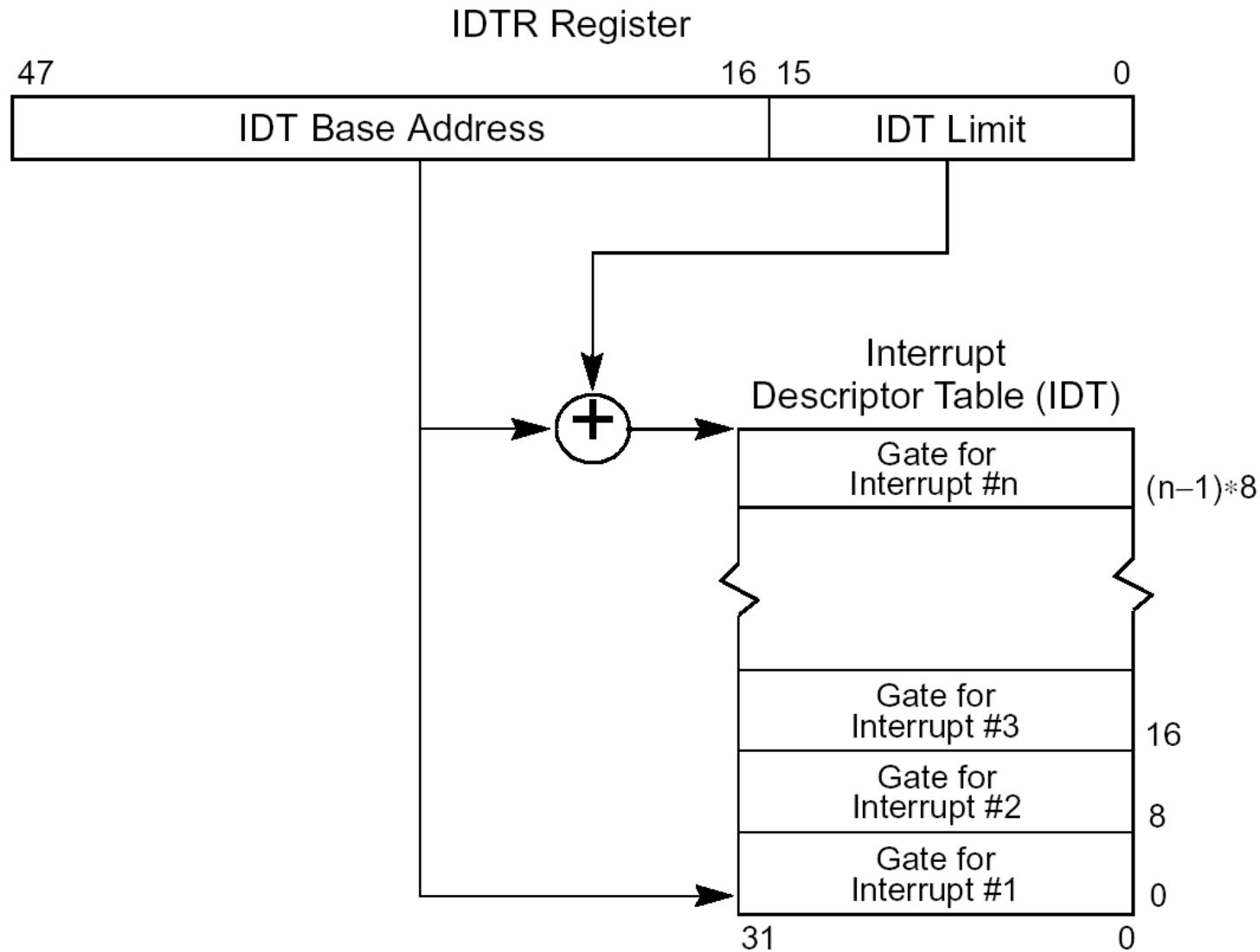
Existen diferentes tipos, los mas usuales son:

- Trap Gate ( Puerta de excepción )
- Interrupt Gate ( Puerta de interrupción )
- Task Gate

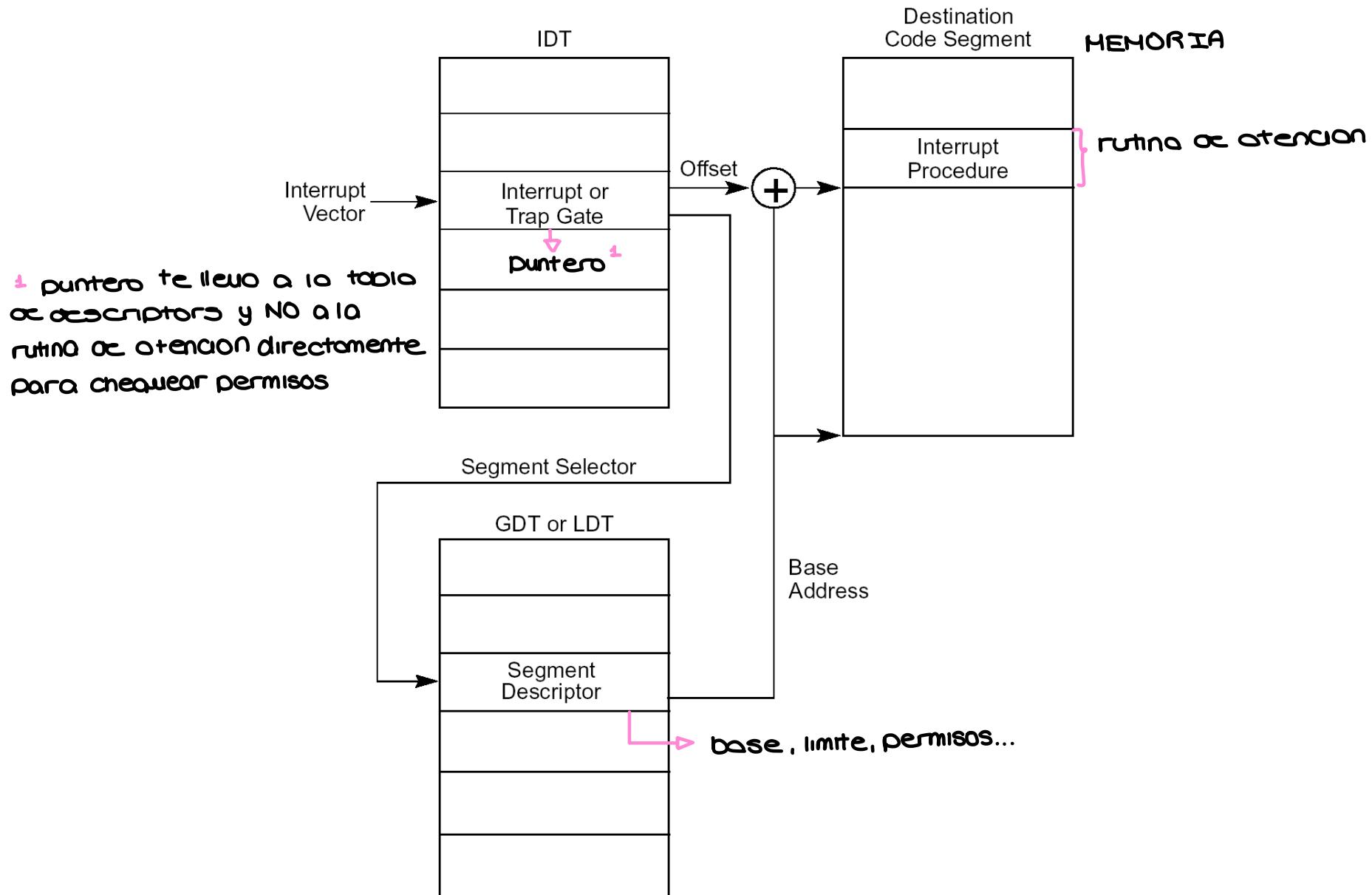
⇒ tmb existe protección para cada puntero a rutina de atención

⚠ TODO debe pasar por las tablas antes de llegar a memoria física

# Interrupciones en Modo protegido



# Interrupciones en Modo protegido

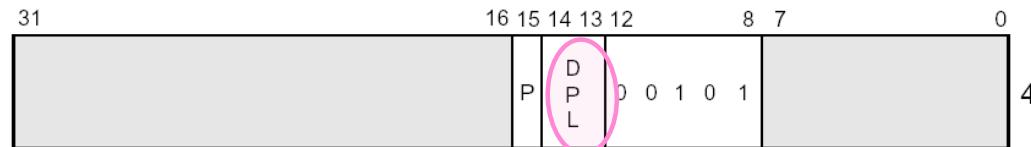


# Tabla de excepciones

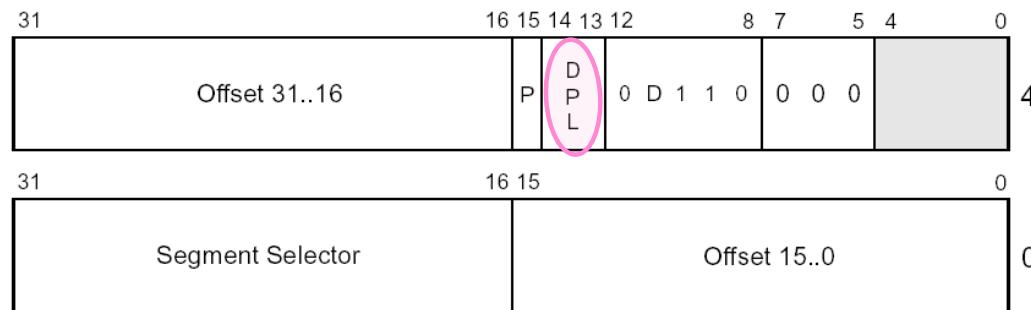
Vector No.	Mne-monic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug	Fault/Trap	No	Any code or data reference or the INT 1 instruction.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT 3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD2 instruction or reserved opcode.
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (Zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction. <sup>2</sup>
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.
15	—	(Intel reserved. Do not use.)		No	
16	#MF	x87 FPU Floating-Point Error (Math Fault)	Fault	No	x87 FPU floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Fault	Yes (Zero)	Any data reference in memory. <sup>3</sup>
18	#MC	Machine Check	Abort	No	Error codes (if any) and source are model dependent. <sup>4</sup>
19	#XF	SIMD Floating-Point Exception	Fault	No	SSE and SSE2 floating-point instructions <sup>5</sup>
20-31	—	Intel reserved. Do not use.			
32-255	—	User Defined (Non-reserved) Interrupts	Interrupt		External interrupt or INT <i>n</i> instruction.

# Tipos de descriptores en la IDT

## Task Gate



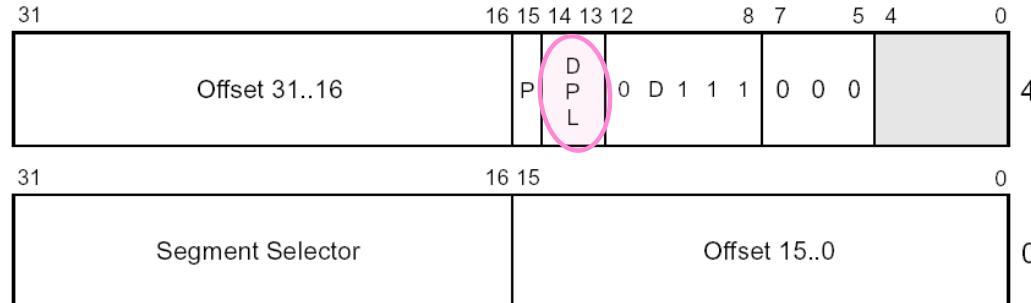
## Interrupt Gate



→ niveles de privilegio

NO son punteros directos a memoria

## Trap Gate



# Manejo de tareas

**El procesador de Intel, nos da una herramienta para guardar el contexto de ejecución. Se lo denomina TSS ( Task State Segment).**

↳ estructura de datos para backuppear y recuperar

Tiene un tamaño mínimo de 100 bytes

LTR } dos instrucciones para backuppear  
STR } y recuperar

31		15	0
	I/O Map Base Address		T 100
		LDT Segment Selector	96
		GS	92
		FS	88
		DS	84
		SS	80
		CS	76
		ES	72
		EDI	68
		ESI	64
		EBP	60
		ESP	56
		EBX	52
		EDX	48
		ECX	44
		EAX	40
		EFLAGS	36
		EIP	32
		CR3 (PDBR)	28
		SS2	24
		ESP2	20
		SS1	16
		ESP1	12
		SS0	8
		ESP0	4
		Previous Task Link	0



Reserved bits. Set to 0.

# Privilegios de E/S

El procesador provee 2 mecanismos:

- Campo de 2 bits IOPL en los EFLAGS. Que especifica el mínimo nivel de privilegio que se debe tener para poder usar instrucciones de Entrada/Salida.  
para acceder al mapa E/S
- Mapa de bits de E/S en el TSS.

Las instrucciones “sensibles” son:

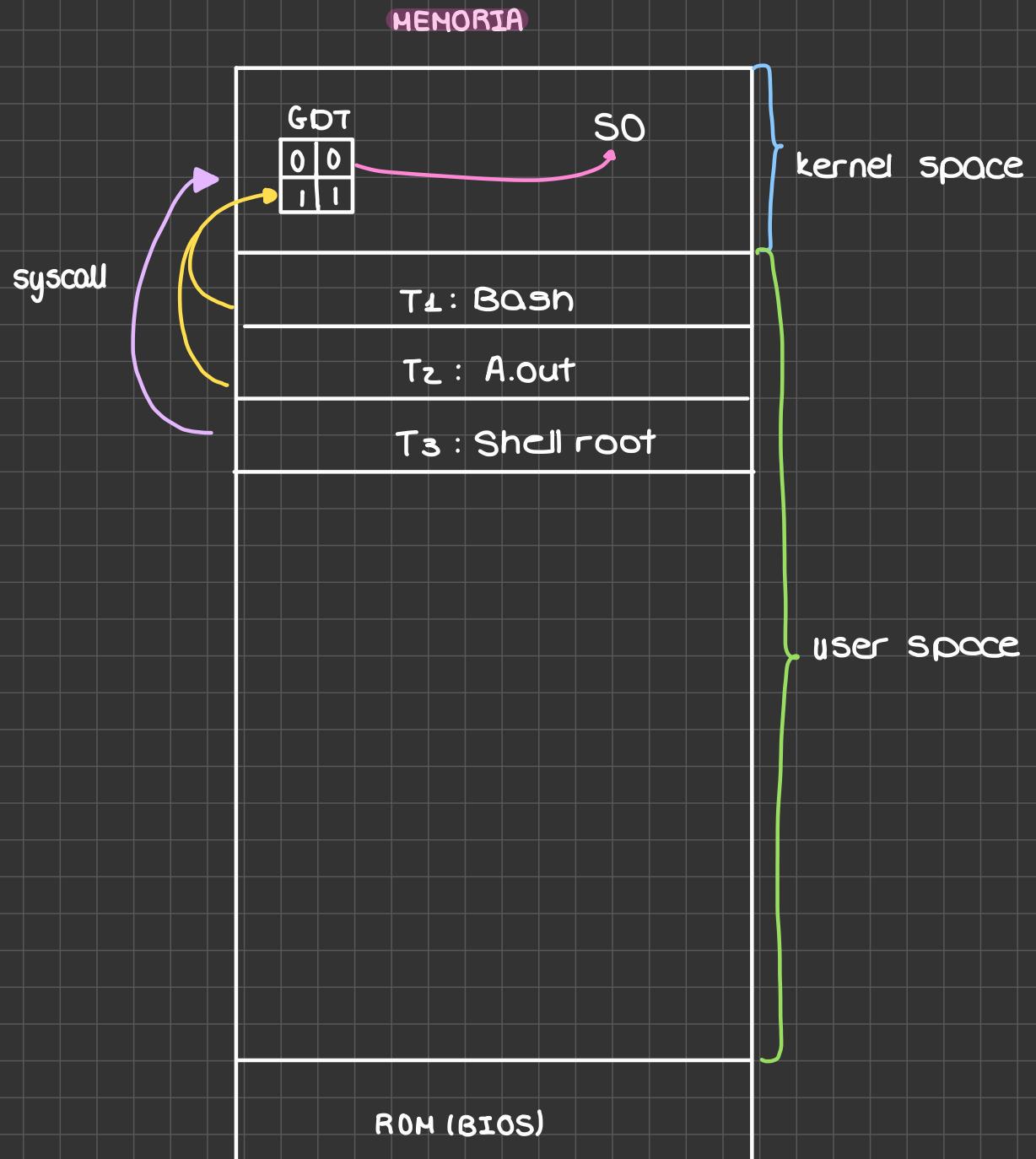
- IN ( Input )
  - OUT ( Output )
  - INS ( Input String )
  - OUTS ( Output String )
  - Cli ( Clear Interrupt )
  - Sti ( Set Interrupt )
- } para INTR
- Input Output Privilege Level

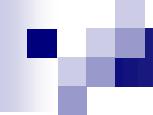
Para ejecutarlas el nivel de privilegio debe ser igual o menor que IOPL.

⇒ El que asigna los niveles es el SO

⇒ el SO se otorga a si mismo el mayor nivel de privilegio

Encendec  
↓  
BIOS  
↓  
SO



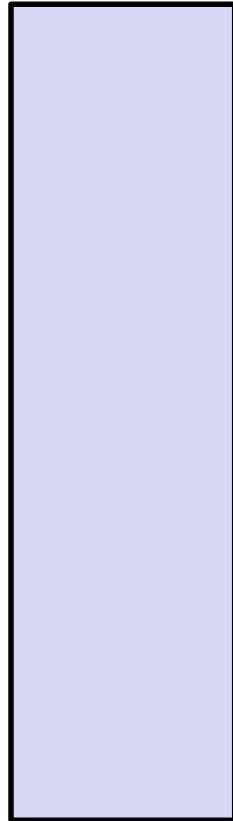


# Memoria Virtual

# Problemática

**Suponemos un procesador de 32 bits en una Pc con 1GB de Memoria**

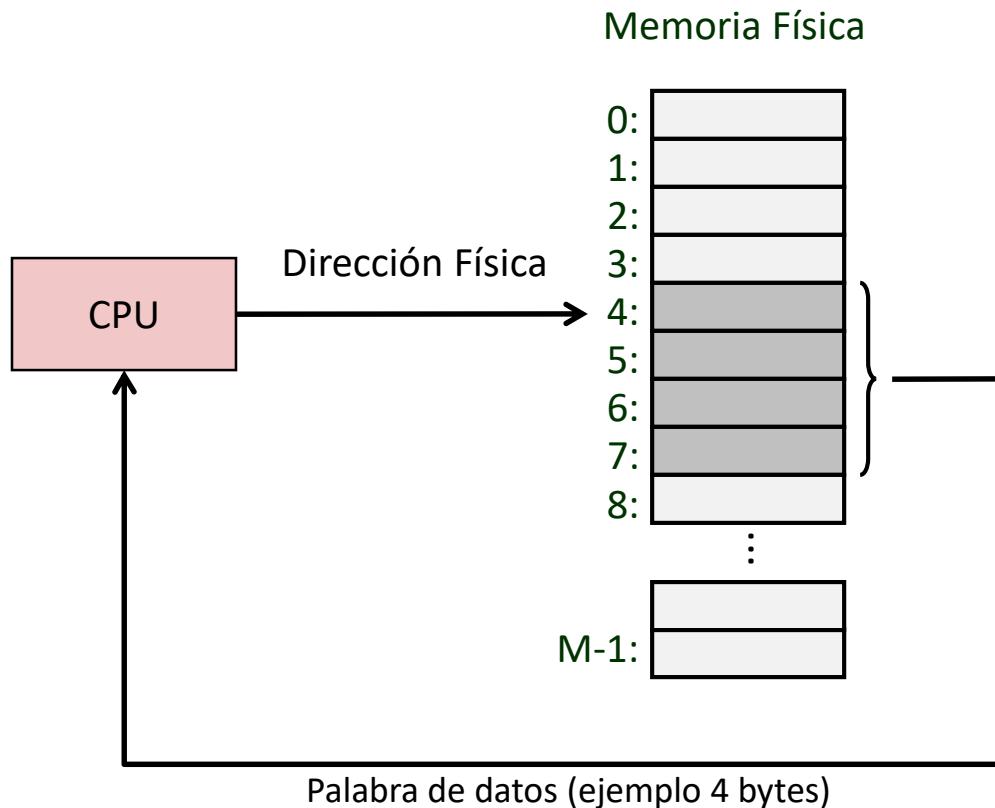
**Memoria virtual de 4GB**



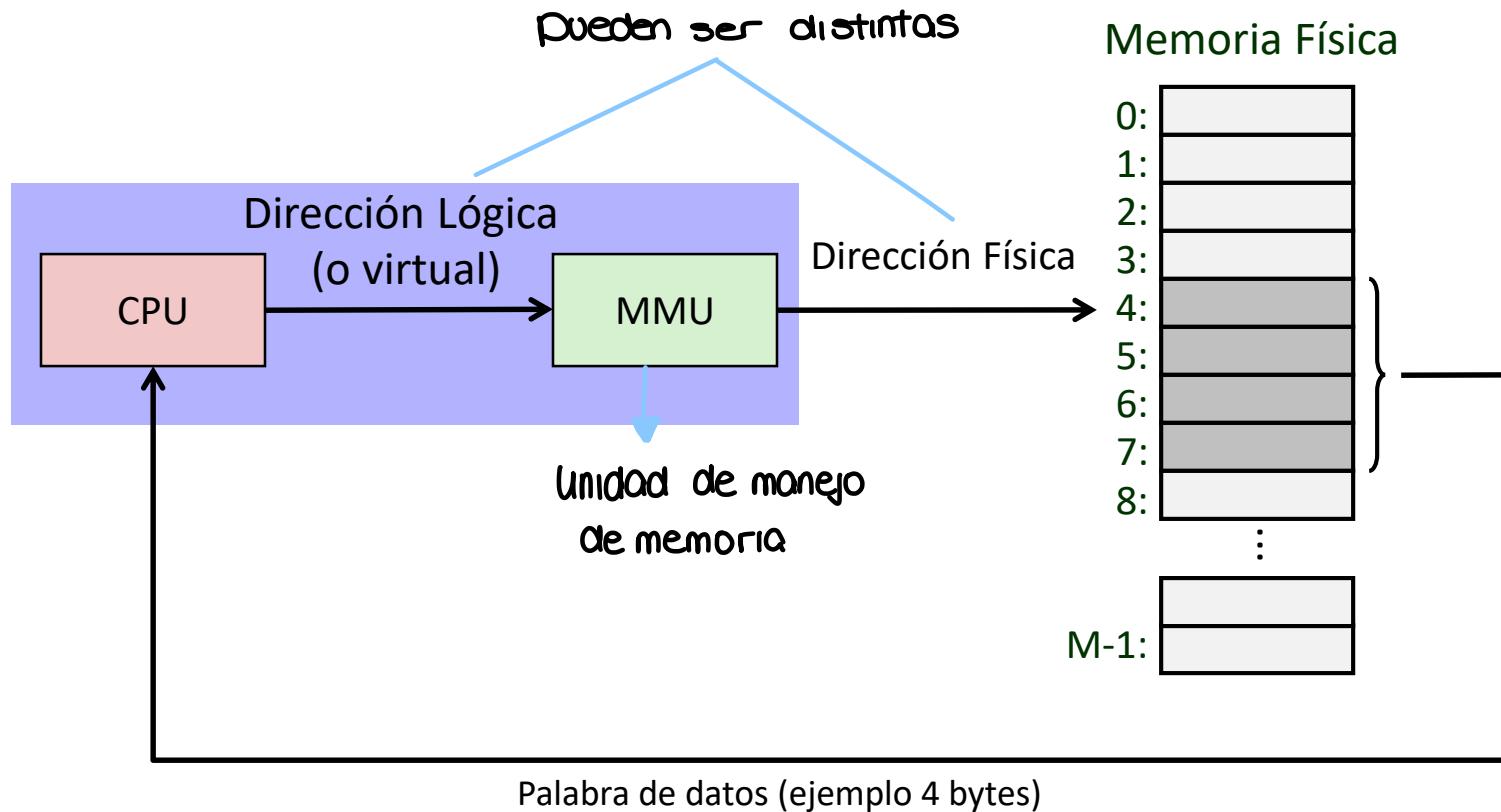
**Memoria física de 1GB**



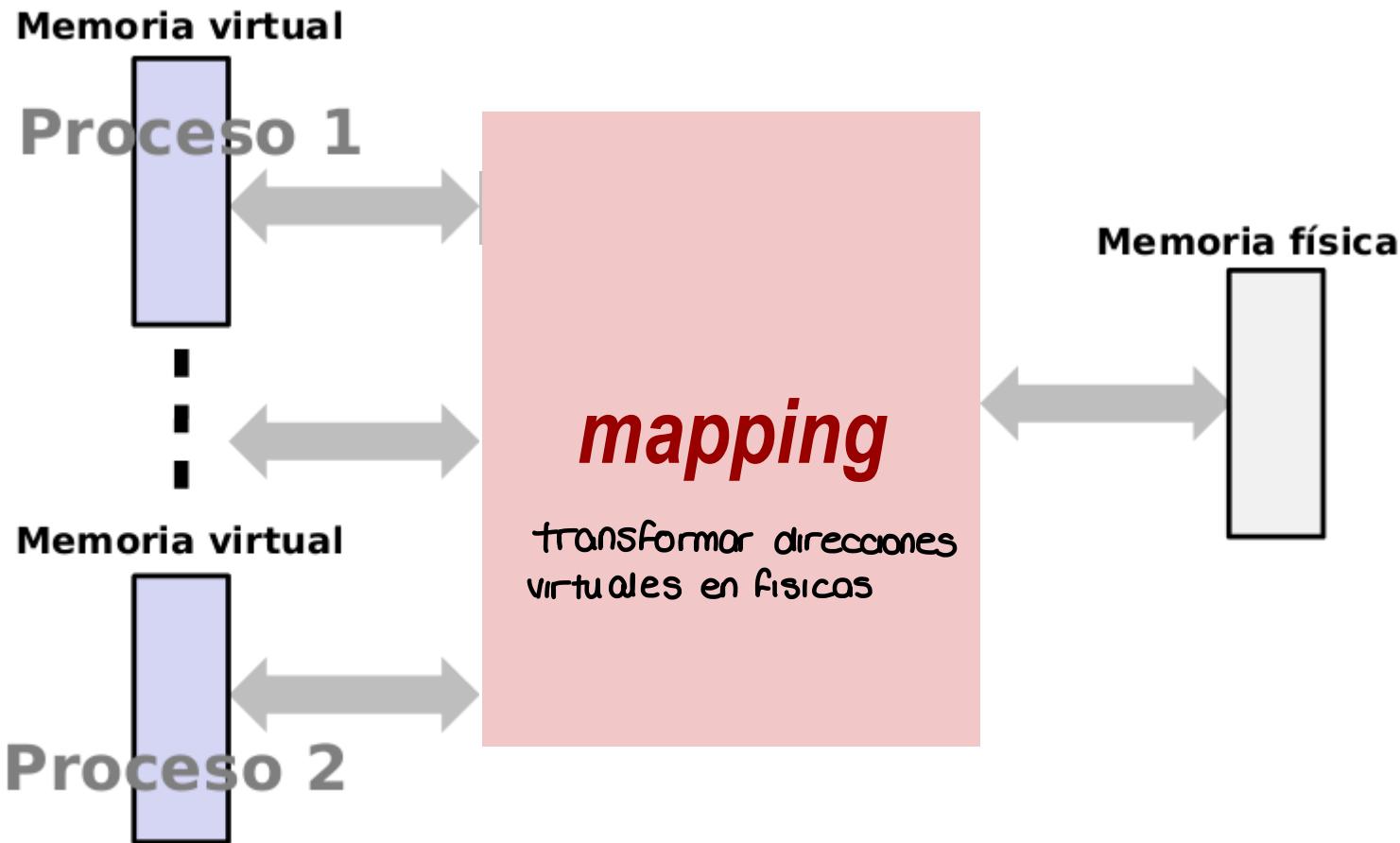
# Direccionamiento físico



# Direccionamiento virtual



# Solución: Indirección



Cada proceso es dueño de su espacio virtual de memoria

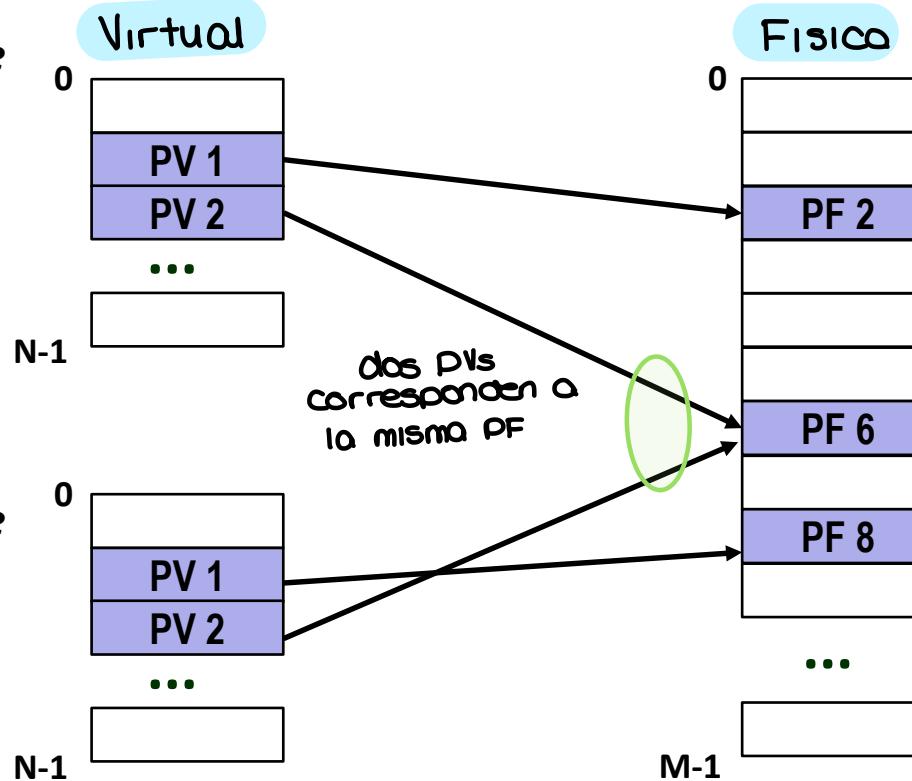
Si ocupa la memoria RAM  $\Rightarrow$  en disco rígido **PERO** no se corre en disco. Cuando se quiere correr el proceso, se sube a memoria.

**Obs:** en disco rígido se pueden correr procesos **PERO** no se hace porque tarda mucho

# VM para múltiples procesos

## *Traducción de direcciones*

*Espacio virtual de direcciones.  
Proceso 1*



*Espacio físico de direcciones*

ej, librería de solo-lectura

*Espacio virtual de direcciones.  
Proceso 2*

[Páginas] = segmentos de memoria de tamaño fijo

⇒ A cada proceso se le definen páginas virtuales que se corresponden con páginas físicas

# Paginación – Ejercicio 1

Desarrolle un sistema de mapeo para un procesador Intel de 32 bits que puede manejar hasta 4GB de memoria física.

1. ¿Qué tamaño de página elige?
2. ¿En cuantas páginas quedó dividida la memoria física y la memoria virtual ?
3. ¿Cómo reacciona su sistema para asignar y para liberar memoria?

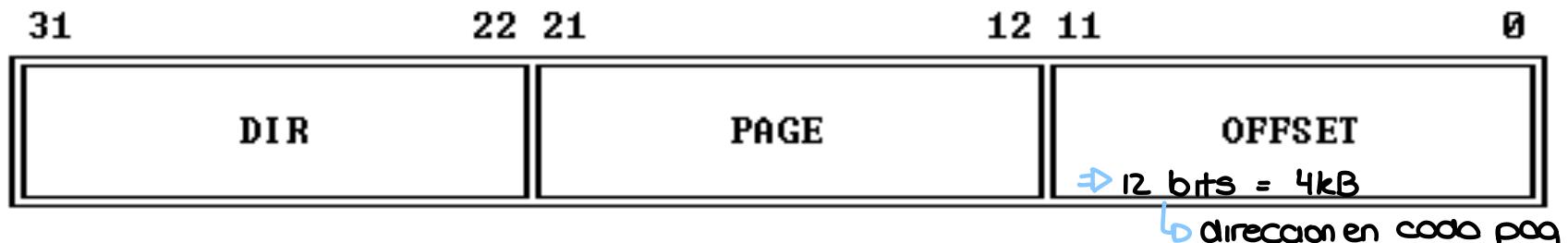
1. Para elegir la cant de paginas se podria tomar un promedio de la longitud de procesos.  
Como no lo hacemos  $\Rightarrow$  tamaño de paginas = 4k
2. Cantidad de paginas =  $\frac{4\text{GB}}{4\text{kB}} = \frac{2^{32}}{2^{12}} = 2^{20}$  paginas
3. Agrupamiento de paginas

# Paginación – Ejercicio 2

**Repita el Ejercicio 1 para un sistema con 1GB de memoria física.**

# Elección de Intel para 32 bits

Figure 5-8. Format of a Linear Address

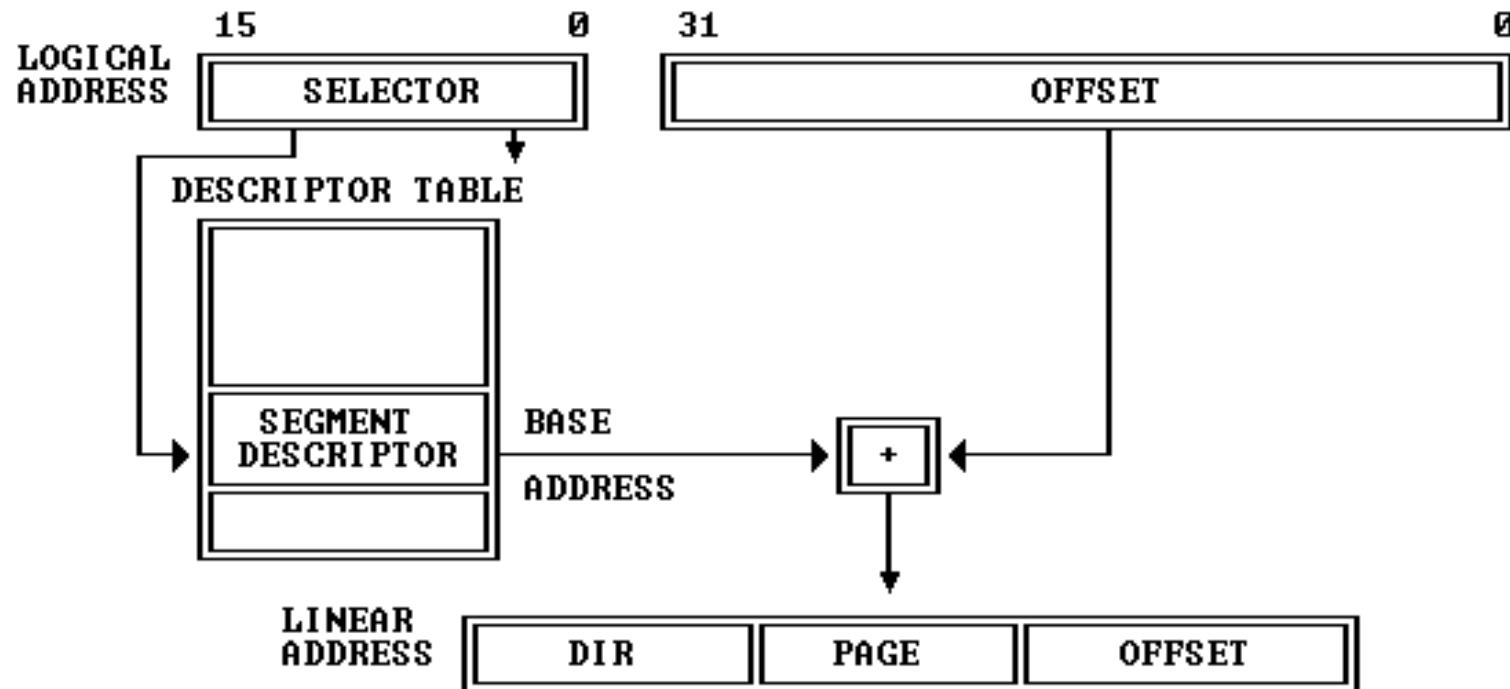


- DIR = grupo de páginas
- OFFSET = desplazamiento dentro de la página

# Paginación

Con un selector y el offset, a través de una tabla de descriptores ( GDT o LDT ), se obtiene una dirección lineal ( de 32 bits )

Figure 5-2. Segment Translation



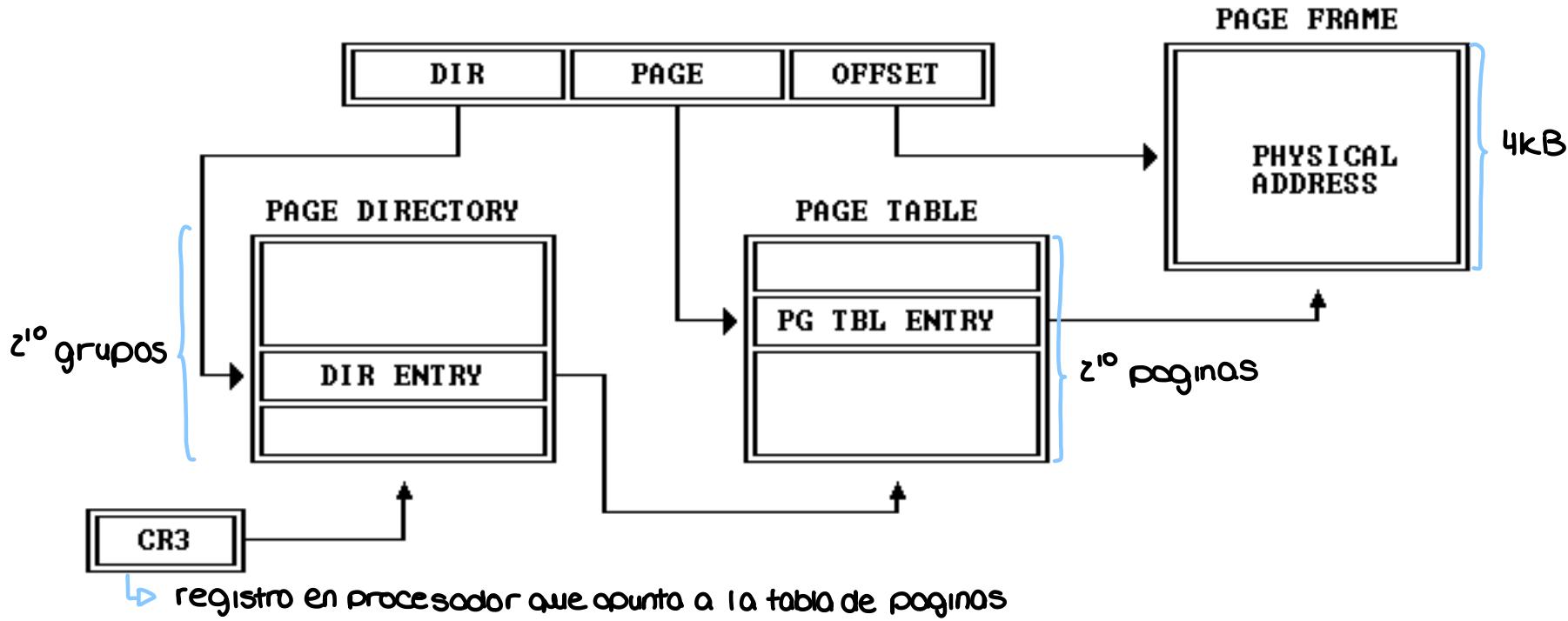
# Paginación

La dirección Lineal al pasar el módulo de paginación se convierte en dirección física. El registro CR3 de 32 bits indica la ubicación del Directorio de páginas.

Figure 5-8. Format of a Linear Address

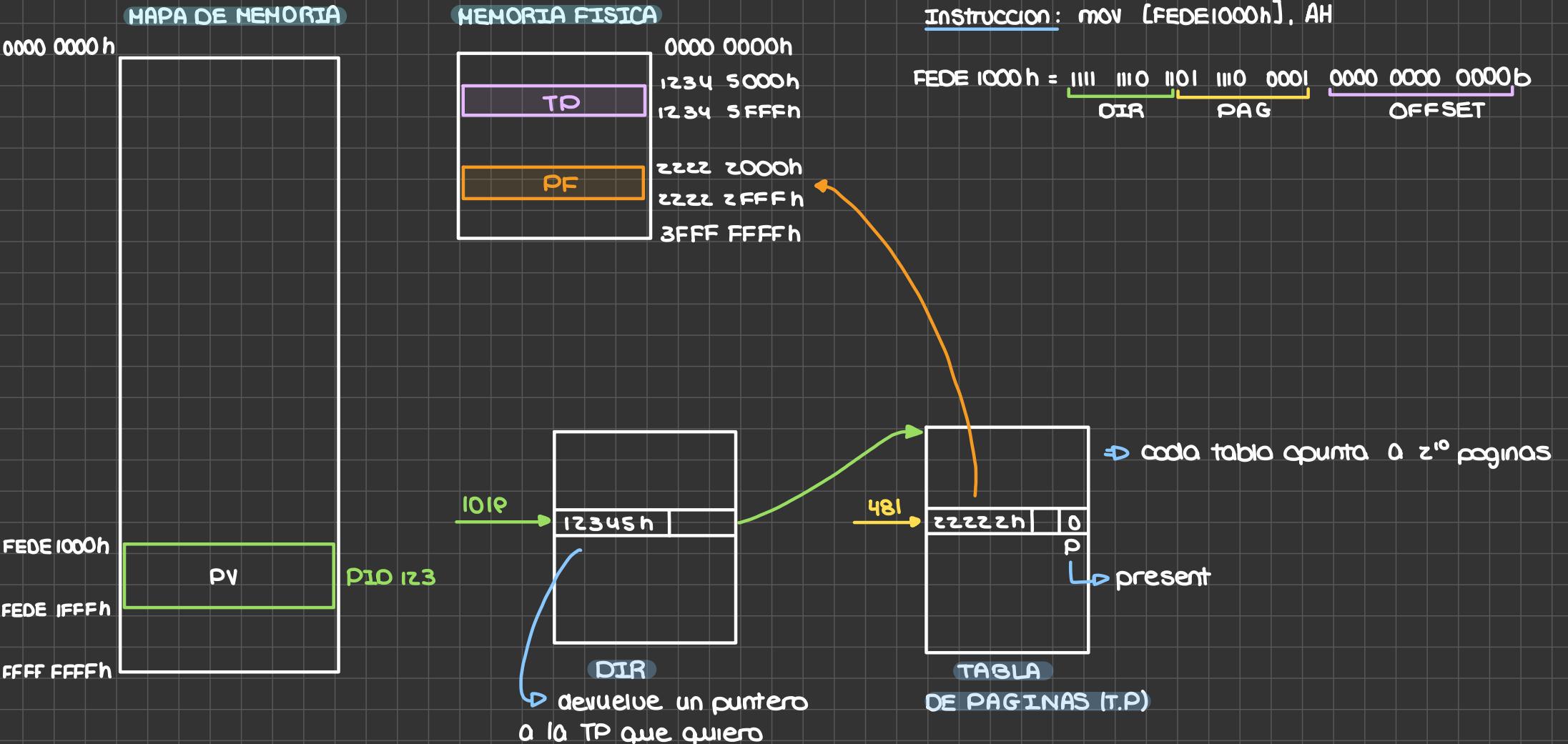


Figure 5-9. Page Translation



## EJEMPLO

1GB de RAM y 4GB de memoria



⇒ Si bajo el PID, lo marco con el bit del present ⇒ 0 = NO presente y 1 = presente

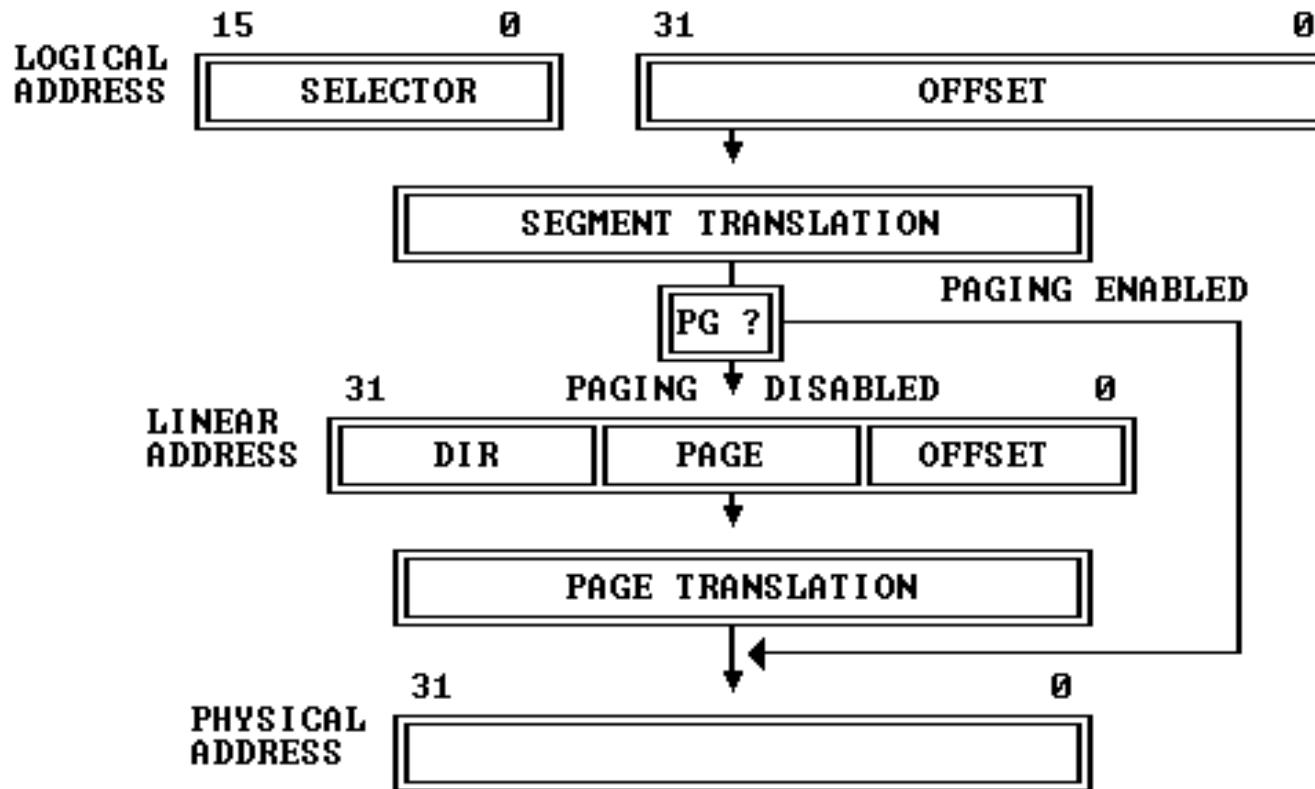
## La PV queda.

Si vuelvo a colocar P10 lo mas probable es que su pagina no apunte a la misma direccion de memoria fisica  $\Rightarrow$  se cambio la direccion en la tabla de paginacion y el btr del present.

# Paginación

Recordemos que la unidad de Paginación se puede deshabilitar, con el bit PG que se encuentra en el registro CR0.

Figure 5-1. Address Translation Overview



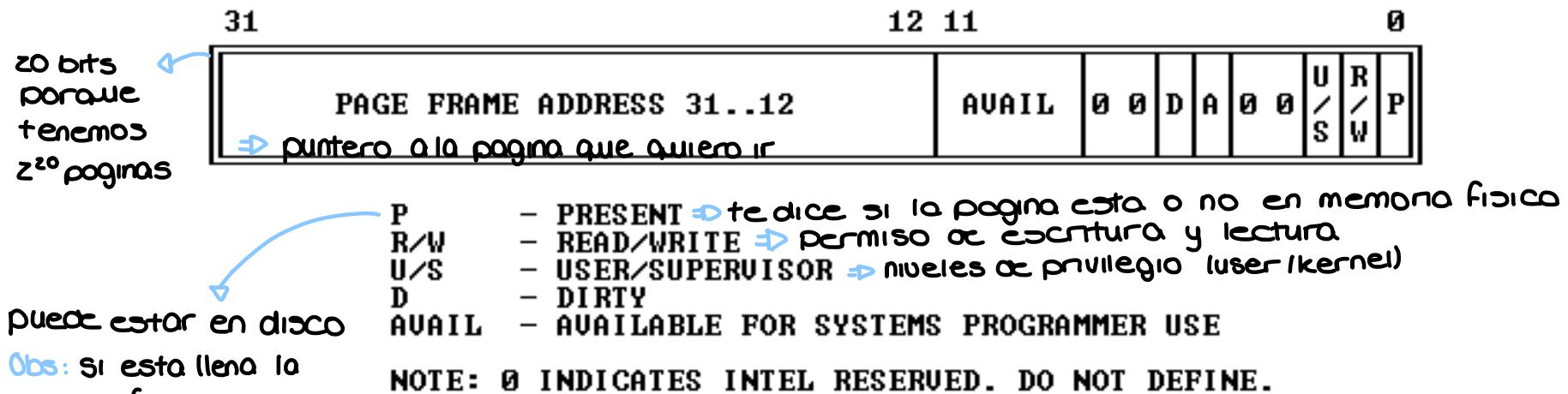
# Paginación

El Directorio de paginas , puede referencias 1024 Tablas de Paginas.

Luego cada una de esas 1024 Tablas de Paginas puede referencias 1024 Paginas, en la memoria física.

Los elementos de estas tablas tienen el mismo formato:

Figure 5-10. Format of a Page Table Entry



Se usa el algoritmo LRU = Last Recently Used

# Paginación

El bit P, o “Present Bit”, indica cuando tiene valor 1, que esa tabla se puede utilizar para obtener una dirección de memoria. Si tiene valor 0, no se puede utilizar.

**Figure 5-11. Invalid Page Table Entry**



Al encontrar un bit P=0, el procesador genera una excepción. Los sistemas operativos que tiene soporte de memoria virtual, utilizan esta excepción para disparar una rutina que lleve a memoria la página faltante. Que seguramente se encontrará en disco.

El bit U/S, divide las páginas en 2 niveles de privilegios, ( Usuario y Supervisor ).

Si no se respetan los accesos se produce una excepción. Si el procesador esta ejecutando en niveles 0,1 y 2 es nivel Supervisor, sino ejecuta en nivel 3 es Usuario.

# Protección combinada

**Cuando la paginación esta habilitada, el procesador, primero chequea los permisos en la segmentación y luego en la paginación.**

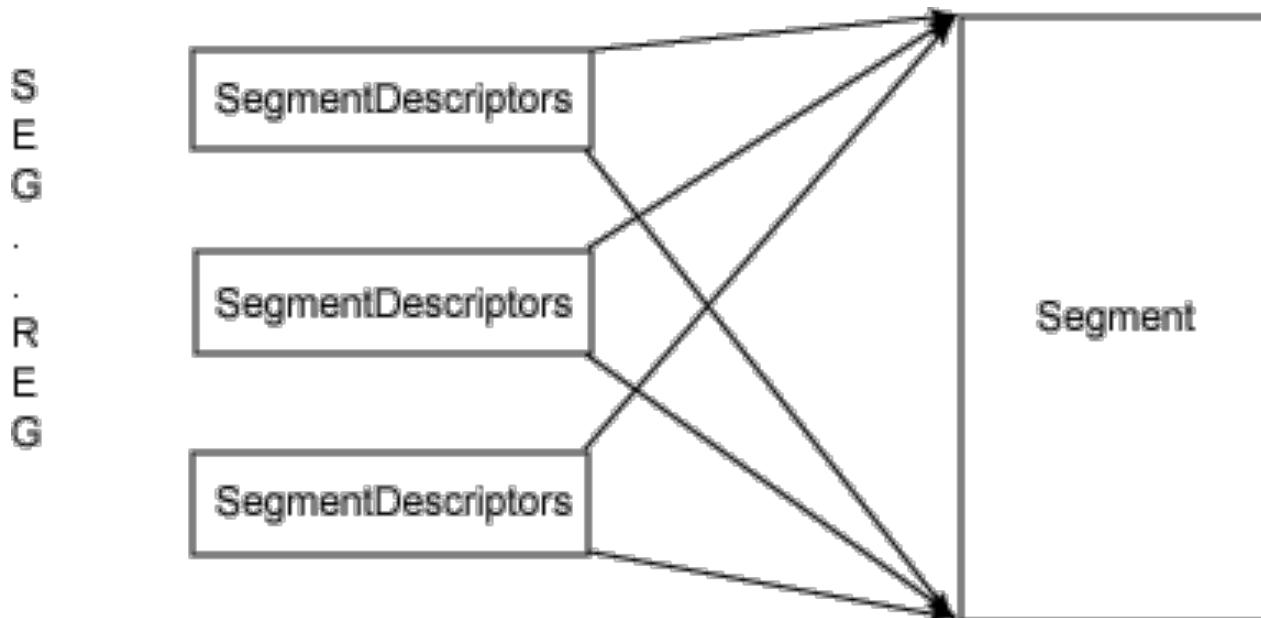
**Por ejemplo se puede definir un segmento de datos, que está compuesto por páginas, donde algunas son de escritura y otras se sólo lectura.**

**Debido a este manejo de memoria, algunos sistemas operativos como Unix, eligen utilizar el modelo flat de memoria.**

# Modelo de memoria Flat

Unix declara dos segmentos de código y de datos que referencian a toda la memoria.

Esto le permite un manejo mas simple de las mismas y a su vez mas portabilidad ya que muchos procesadores no tienen unidad de segmentación pero si de paginación.



# Linux

- **Hasta la versión 2.2 de Kernel, Linux utilizaba los TSS para realizar conmutación de tareas.**
- **A partir de la versión 2.4, realiza la conmutación de tareas mediante funciones.**
- **Los archivos que se utilizan para la arquitectura que estamos viendo se encuentran en: /usr/src/linux/arch**
- **Ahí podremos encontrar los archivos en ASM que setean los registros del microprocesador.**
- **Linux utiliza la paginacion de hardware**
- **En microprocesadores de 64 bits, agrega un nivel extra de paginas ( en el medio de las de hardware ) llamado “middle directory”.**

# Paging y swapping

Cuando se necesita una pagina en memoria y la memoria está completa, Linux elige un pagina que no ha sido accedida ultimamente y realiza un “page out”, que consiste en guardar esa pagina en disco.

Generalmente en una zona especial destinada para ello, puede ser una partición o un archivo en el filesystem

Luego setea en la pagina “vieja” el bit de Presencia en 0, y guarda en su indice la dirección donde la debe encontrar en el disco rígido.

A este concepto se lo denomina “paggig”.

El concepto de swapping se refiere a guardar en disco TODAS las paginas de un proceso.

En Linux existe un thread llamado “kswapd” que se encarga de hacer este trabajo.

Es interesante estudiar los algoritmos que se aplican para decidir que pagina debe ser “bajada”, ya que si se decide mal, dicha pagina puede llegar a tener la próxima instrucción a ejecutar, o un dato que debe ser accedido.



# Memoria Cache

# Memoria Caché – Ejemplo comercial

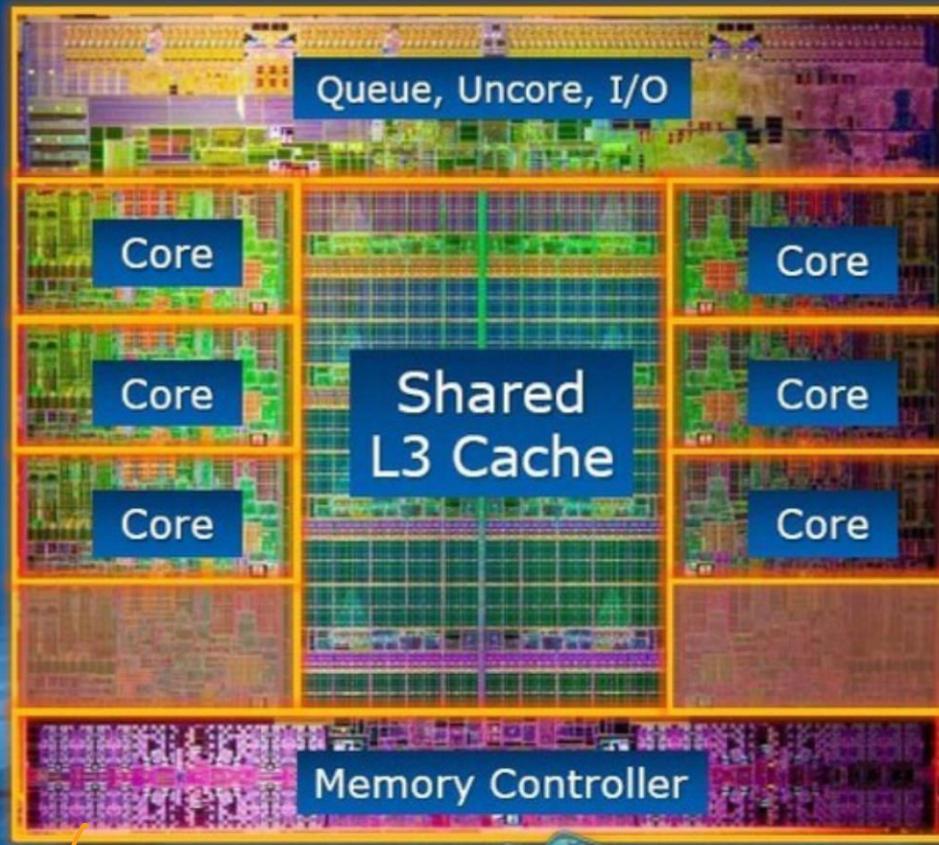
- Procesador Intel® Core™ i7 – 3960X
- 6 Núcleos
- Velocidad de clock 3,3 GHz
- 15 MB de SmartCache ➔ datos + etiquetas
  - ➔ cache tradicional de la CPU

## Caché

El Caché de CPU es un área de memoria rápida ubicada en el procesador. El Caché inteligente Intel® se refiere a la arquitectura que permite a todos los núcleos compartir dinámicamente el acceso al caché de alto nivel.

# Memoria Caché – Ejemplo comercial

Intel® Core™ i7-3960X Processor Die Detail



paginacion, segmentacion, etc.

# Memoria Caché - Causa

- Los programas se ejecutan en pasos secuenciales
- Las variables se alojan en zonas adyacentes

⇒ busca tener los datos más accedidos cerca del μP

- Recordar un bucle en ASM:

```
ciclo_a:
```

```
    mov    EAX,EBP  
    mov    [EBP+EAX-24],88  
    inc    [EBP-8]
```

```
ciclo_b:
```

```
    cmp    [EBP-8],15  
    jle    ciclo_a
```



Se accede much@s veces  
⇒ ir a buscar a la RAM cada vez es poco  
eficiente ⇒ se trae un bloque de memoria  
de la RAM o una memoria dentro de la pos-  
tilla del μP ⇒ cache

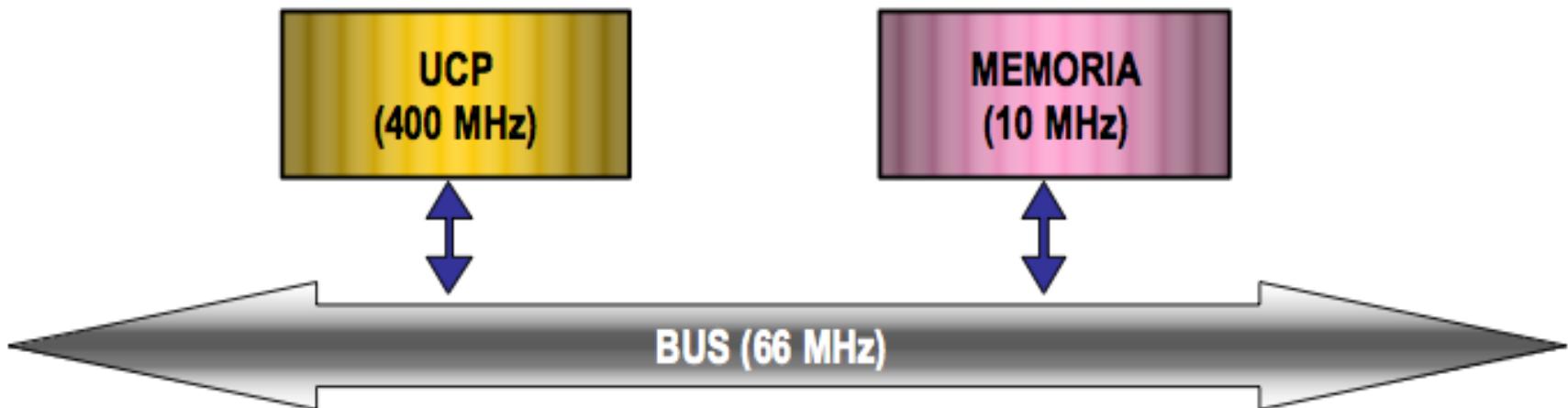
contiguos en  
memoria

# Memoria Caché - Tecnología

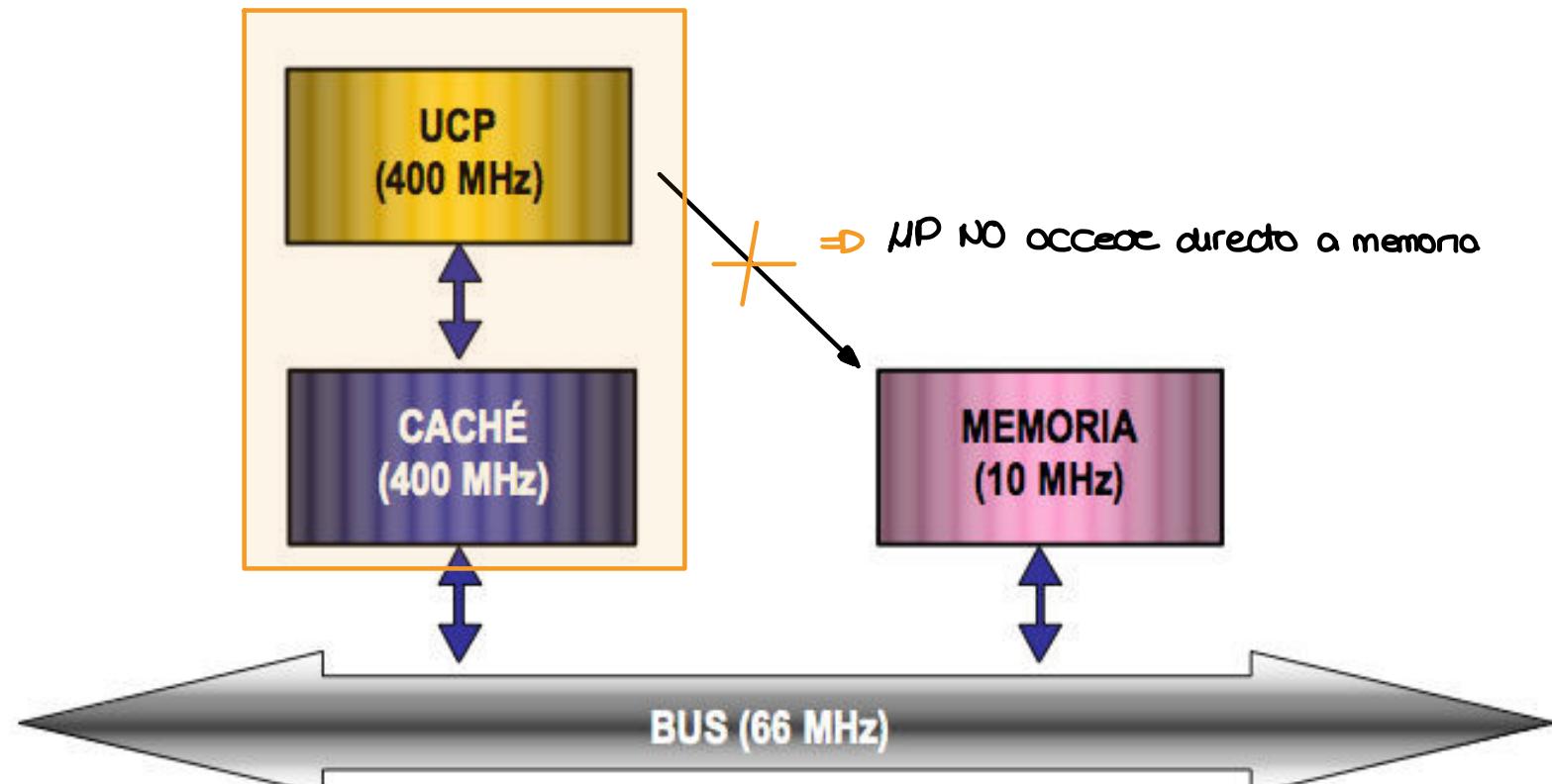
- Memoria DRAM mas económica y lenta ➔ el tiempo que tarda en devolver un dato es mucho
  - Tecnología SRAM (Static RAM)
  - Velocidad de respuesta acorde al CPU
  - Guarda copias de la DRAM
- ➔ Se toman bloques de memoria y se guardan en memoria cache
  - ↳ los que se usa con mas frecuencia.

# Memoria Caché

- Diferencias de velocidades
  - Cuello de botella
  - Pérdida de performance



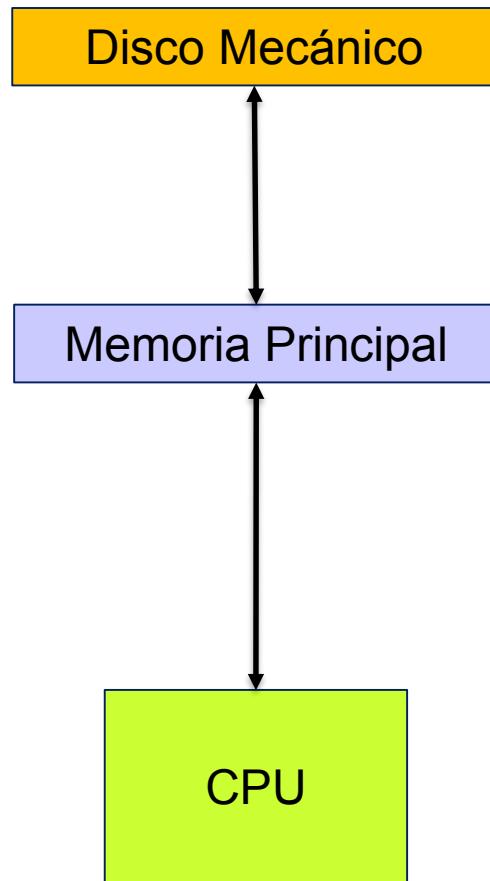
# Memoria Caché - Funcionamiento



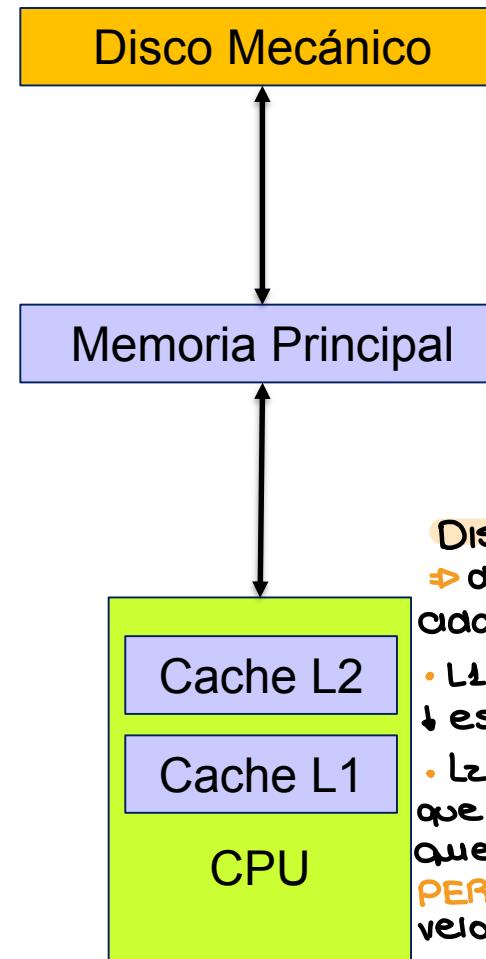
=> todas las consultas que haga el  $\mu$ P pasan por la memoria cache

# Memoria Caché - Evolución

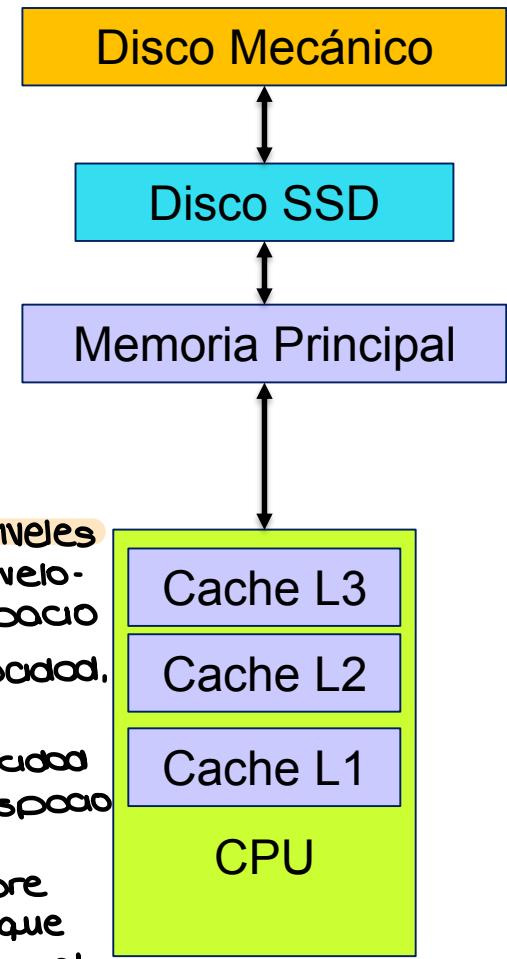
Hasta los 80's



90's y 2000's



2010



# Memoria Caché - Estructura

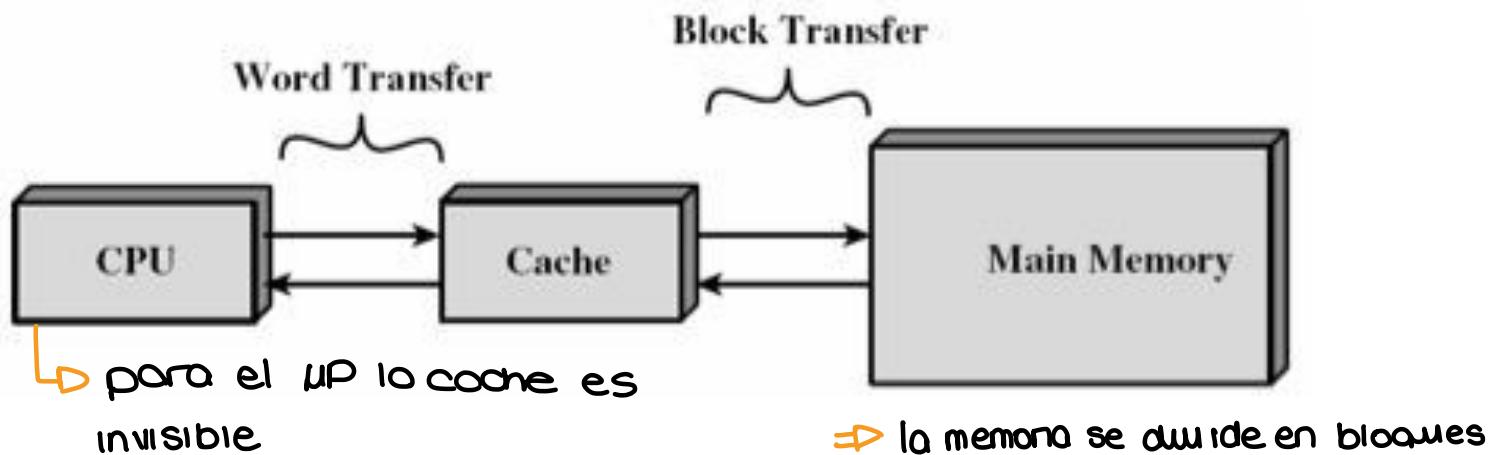
Se compone de:

- Memoria de datos
- Memoria de etiquetas
- Controlador
  - Selecciona cuantos y cuales bytes se copian a la memoria de datos. Utiliza diferentes algoritmos.

# Memoria Caché

▷ unidad de cosas a copiar

- El controlador ve a la RAM en bloques de tamaño fijo
- Por ejemplo de 32 bytes
- Entonces RAM de 1MB son 32768 bloques



## Memoria Caché - Ejemplo

- Suponemos RAM de 1 MB ( $1M \times 8$ ).
- Bloques de 32 bytes.
- Caché de 4 K para datos (sin etiquetas).

$$4k = 2^{12} \Rightarrow \frac{2^{12}}{2^5} = 128 \Rightarrow \text{bloques en la cache}$$

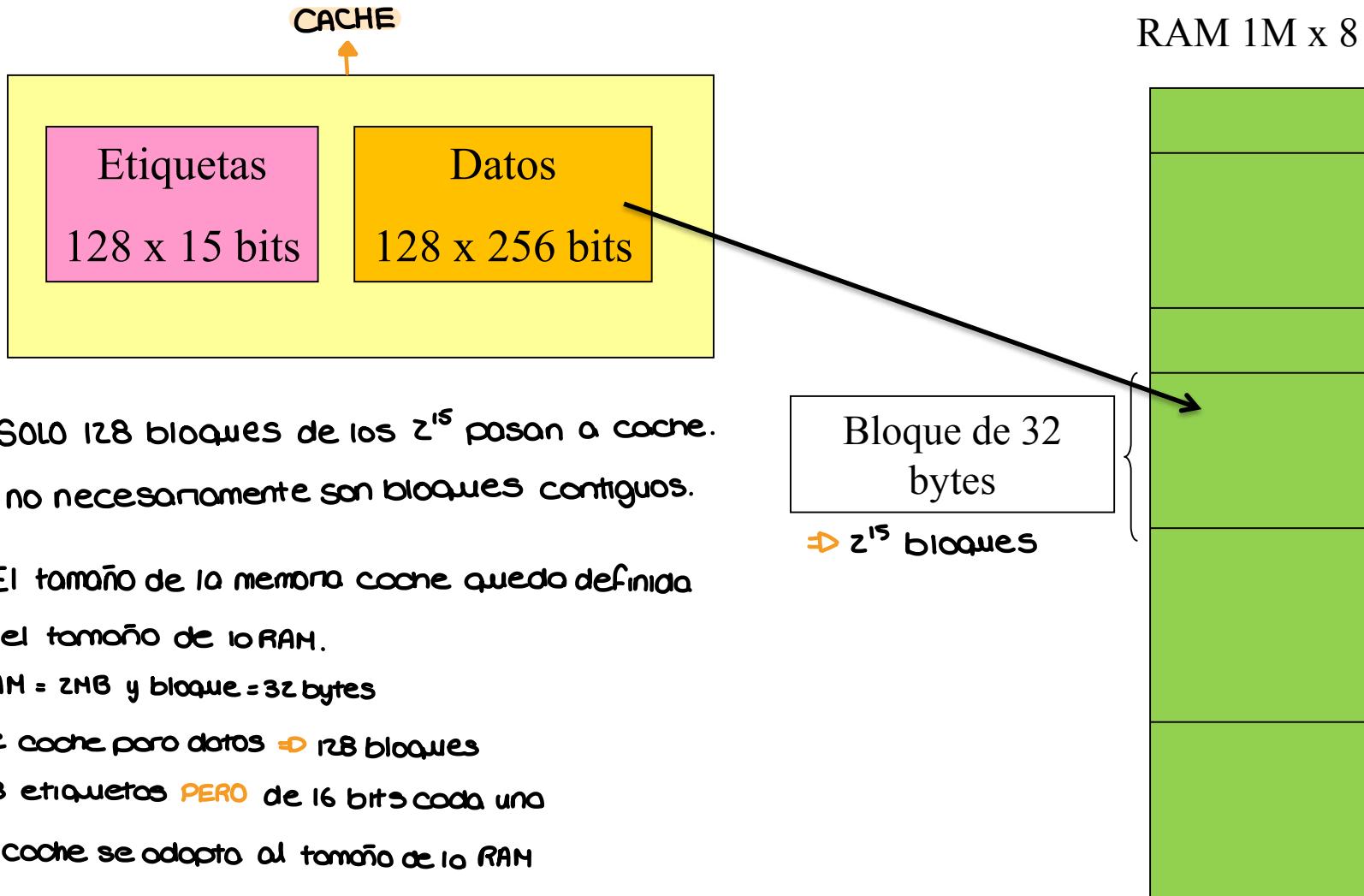
Entonces la memoria de datos tendrá 128 entradas de 256 bits cada una.

Por lo tanto la memoria de etiquetas tendrá 128 entradas de 15 bits cada una porque:

⇒ Etiqueta = id del bloque  
Si tengo  $2^5$  bloques ⇒ necesito 128 etiquetas de 15 bits.

$$\frac{2^{20}}{2^5} = 2^{15} \Rightarrow \text{bloques posibles}$$

# Memoria Caché - Ejemplo



**⚠** SOLO 128 bloques de los  $2^5$  pasan a cache.

**Obs:** no necesariamente son bloques contiguos.

**⚠⚠** El tamaño de la memoria cache quedo definida

por el tamaño de la RAM.

Ej. RAM = 2MB y bloque = 32 bytes

4k de cache para datos ⇒ 128 bloques

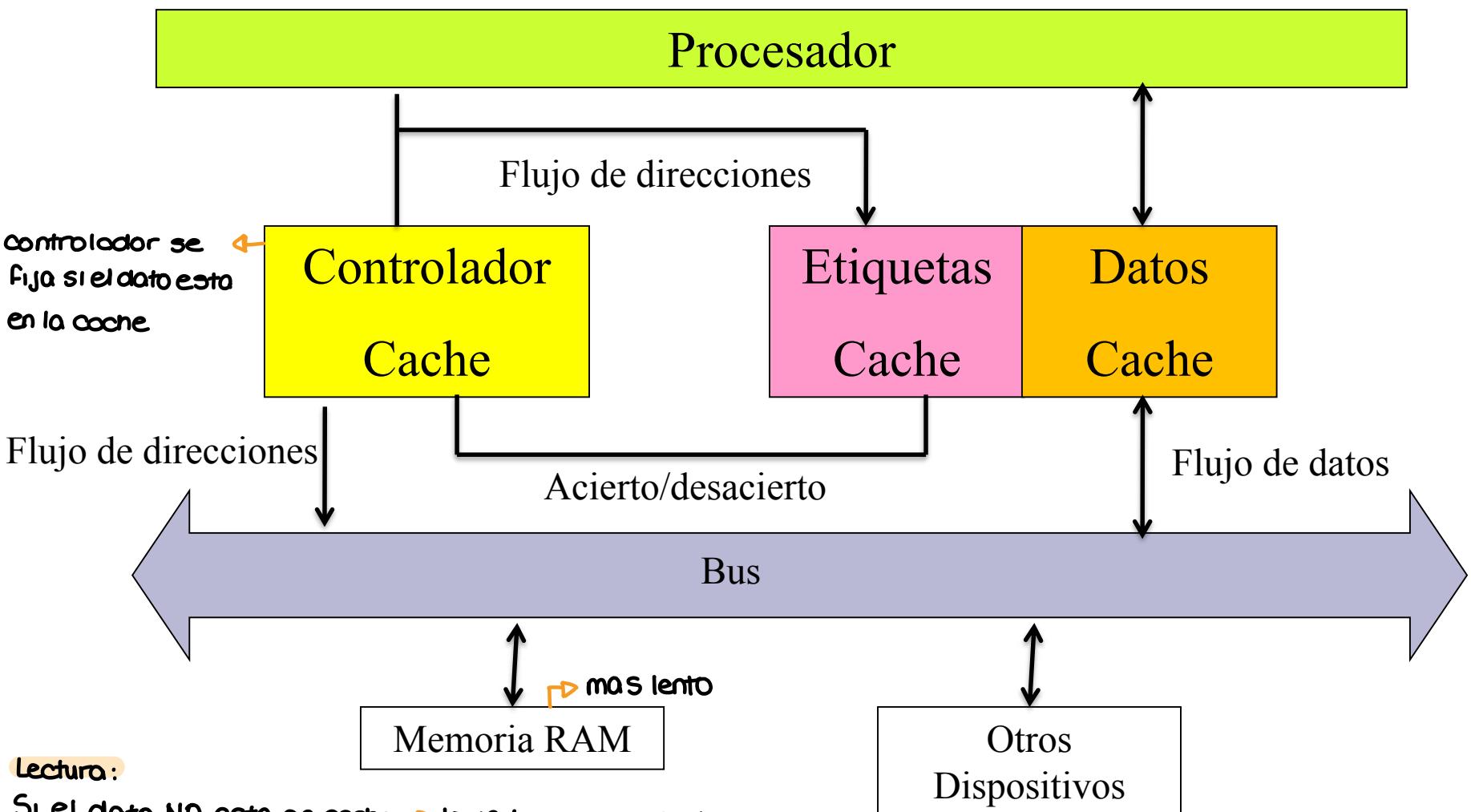
⇒ 128 etiquetas **PERO** de 16 bits cada una

⇒ la cache se adapta al tamaño de la RAM

En caso de escritura:

- (1) Cache y memoria al mismo tiempo **PERO** mas lento
- (2) Se escribe en cache y despues se actualiza la memoria

# Memoria Caché – Conexión en serie



Lectura:

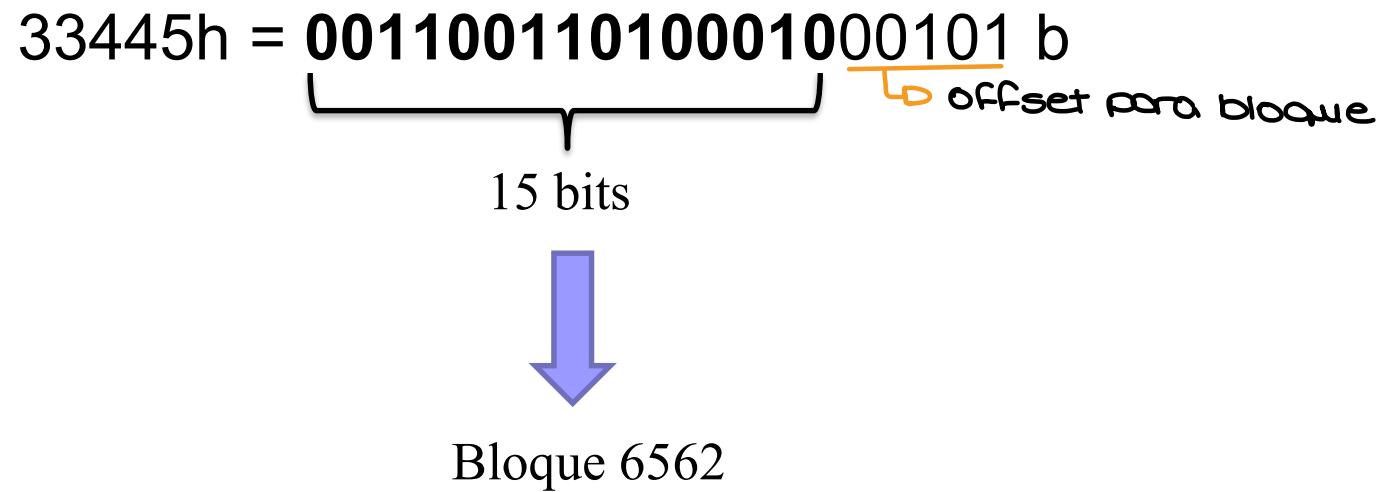
Si el dato NO esta en cache  $\Rightarrow$  lo va buscar en RAM

$\Rightarrow$  trae el dato y aquellos que esten en el mismo bloque

porque lo mas probable es que sean pedidos despues  $\Rightarrow$  se cochea el bloque SIEMPRE

## Memoria Caché - Ejemplo

Suponemos dirección física 33445h  $\Rightarrow$  1MB de memoria



Se busca en las 128 etiquetas si una tiene el contenido  
**001100110100010**

# Memoria Caché - Mapeos

- Mapeo Directo.
  - Un bloque de memoria solo se puede mapear a una única ranura en el cache.
  - Sencilla, poco utilizada
- Mapeo Asociativo.
  - Un bloque de memoria se puede mapear a cualquier ranura del cache.
  - Compleja, más utilizada.

Ventajas?

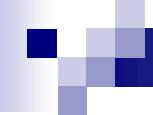
Desventajas?

## Memoria Caché – Políticas de sustitución

- Se actualiza la cache al haber fallo o ausencia de palabra buscada.
- First In First Out (FIFO)
- Least Recently Used (LRU) (necesita flag !)  $\Rightarrow$  usa hits = cantidad de accesos
- Random

# Memoria Caché – Actualización de RAM

- Escritura inmediata (write through)
    - Se actualizan ambas memorias juntas
    - Mas lento
    - Mas económico
  - Escritura obligada (write back)
    - Solo se actualiza lo estrictamente necesario
    - Mas rápido
    - Menos económica
    - Problemas con dispositivos DMA
    - Problemas con Multi-cores
- ↗ acceden directamente a memoria
- ⇒ el controlador sabe si dos núcleos están queriendo acceder a la misma zona de memoria



# Procesadores de 64 bits

## IA-32 IA-64 AMD64

▷ arquitectura de 32 bits

- **IA-32:** Tecnología Intel del 32 bits
- **Intel 64/EM64T:** Extensión Intel de 64 bits, compatible con 32 bits ➔ se puede setear para que trabaje en 32 bits ➔ procesador de 32 bits
- **AMD64:** Tecnología AMD de extensión 64 bits, esto sirve porque hay cosas que con 32 bits alcanza
- **IA-64:** Tecnología Intel de 64 bits (Itanium) no compatible con 32.

# IA-32 IA-64 AMD64

☞ SI el set de instrucciones es distinto = Distinta arquitectura

## Architecture

Instruction set definition  
and compatibility

## Microarchitecture

Hardware implementation  
maintaining instruction  
set compatibility with  
high-level architecture

☞ como se implementa  
el hardware

## Processors

Productized implementation  
of microarchitecture

## Microarchitecture History

examples:

EPIC<sup>1</sup> (Itanium®)

IA-32

IXA<sup>2</sup> (Intel XScale®)

Procesador ARM

P5

P6

Intel NetBurst®

Mobile

Intel® Pentium®

Intel® Pentium® Pro  
Intel® Pentium® II/III

Intel® Pentium® 4  
Intel® Pentium® D  
Intel® Xeon®

Intel® Pentium® M

1. EPIC (Explicitly Parallel Instruction Computing)

2. IXA (Intel® Internet Exchange Architecture)

## Set de instrucciones

- El set de instrucciones define la **Arquitectura**
- Por lo tanto define la **compatibilidad**
- IA-32: Para los Pentium, Celeron, Core Duo, Xeon y Core 2 Duo, etc
- Intel Xscale: Para los tipo ARM
- IA-64: Es el set de instrucciones para los Itanium (high end)

# Micro-Arquitecturas

- Define como se implementa el hardware
- Nuevas tecnologías
  - Multi-nucleos  $\Rightarrow$  arquitectura multicore = + nucleos que pueden ejecutar instrucciones de forma simultanea
  - Consumo de energía
  - Virtualización  $\Rightarrow$  si tiene features para correr maquinas virtuales
  - Cache
  - FSB
  - Pipelines

# Micro-Arquitecturas y Procesadores

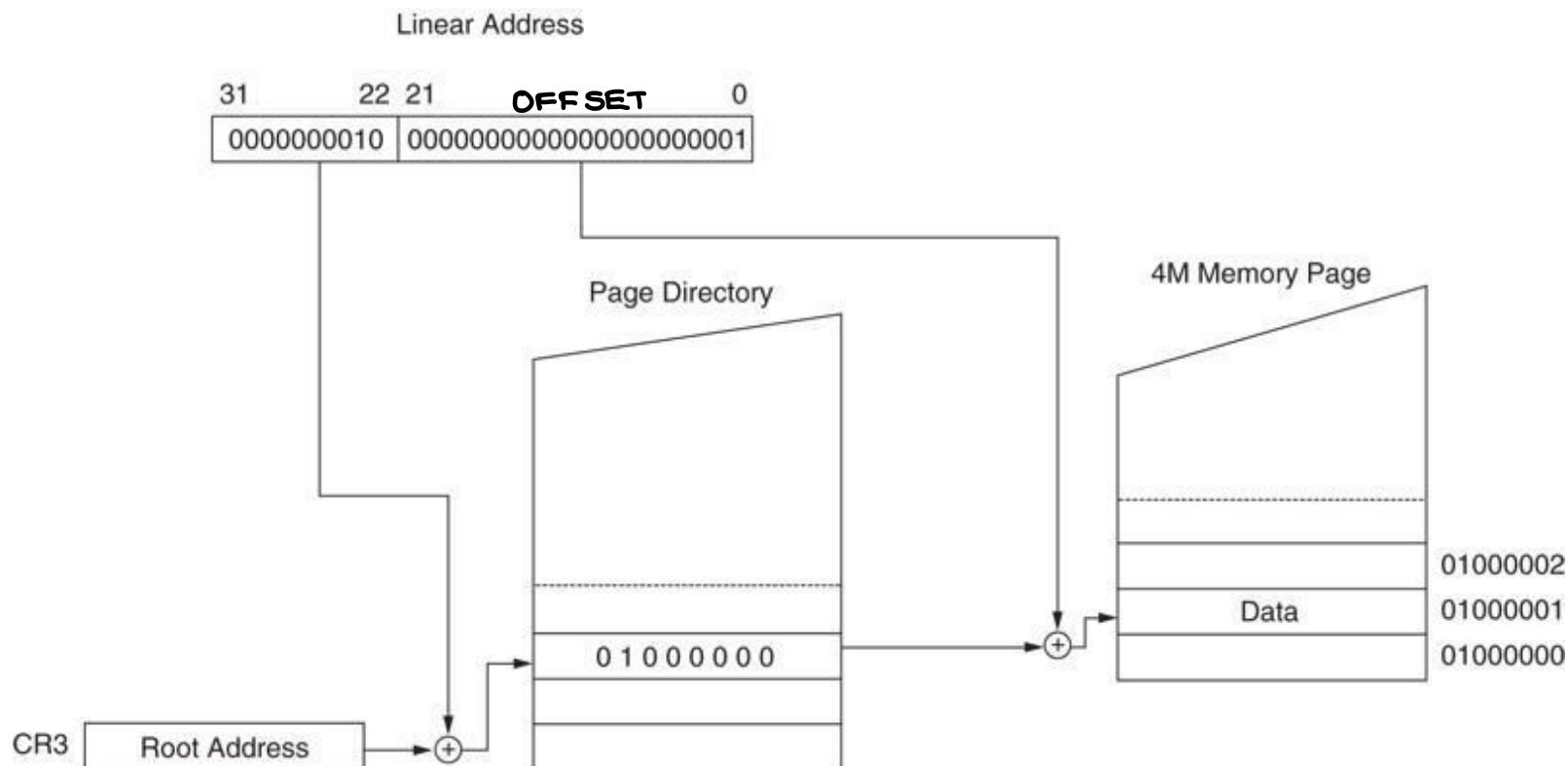
- IA-32
  - P5
    - Pentium
  - P6
    - Pentium Pro, Pentium II, Celeron, Pentium III
  - NetBurst
    - Pentium IV, Xeon, Pentium IV HT, Pentium M
  - Core
    - Core2Duo, Core2Quad Xeon, Quad Core
  - Nehalem
    - Core i7

# Pentium

- El Pentium tiene 64 bits de bus de datos
- 32 bits de bus de direcciones  $\Rightarrow$  4G
- Arquitectura Superescalar
  - Tiene 3 unidades de ejecución
    - Punto flotante
    - U-pipe
    - V-pipe
  - Podría ejecutar al mismo tiempo
    - FADD ST,ST(2)
    - MOV EAX,12h
    - MOV EBX,13h
  - Porque son instrucciones independientes

# Pentium - Paginación

- Puede manejar dos niveles
- Página de 4 MB
- Se setea con el bit PSE en el CR0



## Pentium Pro

▷ agrega 4 líneas

- 36 líneas de bus de direcciones (64 G) PERO registros  $\approx$  32
- 64 líneas de bus de datos
- Para acceder a +4Gb utiliza PAE (Physical Address Extension)
- Las tareas siguen viendo un máximo de 4GB.
- El sistema operativo tiene que proveer de un manejo de memoria para lograr utilizar los 64 Gb de direcciones físicas.



## Physical Address Extension

- Se habilita con el bit 5 del CR4 (PAE)
- El micro llega a 64 GB de memoria física
- Pero las direcciones virtuales siguen siendo de 32 bits
- Se agregan en las entradas del directorio y las tablas de paginas 24 bits para direccionar las paginas
- 16 M de paginas  $\Rightarrow$  se agranda el tamaño de las paginas  
porque los procesos ocupan mas lugar
- $16\text{ M} * 4\text{ KB} = 64\text{ GB}$

## Intel 64 / EM64T

- Extended Memory 64 Technology
- Compite con AMD64
- Direcciones virtuales de 64 bits  $\Rightarrow$  mapa de memoria =  $2^{64}$
- Direcciona 16 EB (exabytes)
- Bus de direcciones puede llegar a 40 líneas (1 TB)
- Registros de 64 bits (RAX, RCX, RIP)  
PERO siguen existiendo los registros de 32, 16 y 8 bits  $\Rightarrow$  compatible hacia atrás

## Intel 64 / EM64T

- Registros de 64 bits (RAX, RCX, RIP)
- Instrucción de direccionamiento a memoria de 64 bits
- ALU de 64 bits

## Intel 64 / EM64T

- Esta tecnología introduce un nuevo modo de funcionamiento: **IA-32e**, que tiene dos sub-modos:
  - **Modo compatibilidad con 32 bits** (Legacy Mode)
    - Similar a modo protegido de 32 bits
    - Para acceder +4GB usa PAE (Physical Adress Extension)
  - **Modo 64 bits** (Long Mode)
    - Utiliza direcciones de lógicas de 64 bits
    - Los operandos por default son de 32 bits (salvo que tengan prefijo REX)

# Microprocesadores ARM

- Introducción a los microprocesadores ARM
- Comparación con procesadores Intel

# Historia de ARM

1990



ARM

2004



ARM

2018



arm

# ARM

- ARM = Advanced RISC Machine ( antes Acorn Risc Machine )
- En 1985 comienzan con el primer micro ( ARM1 )
- Se crea entre tres empresas: Acorn, Apple y VLSI
- No fabrica procesadores sino que realiza las especificaciones
- Comercializa licencias (IP License (Intellectual Property))
  - Tiene procesadores de 32 y 64 bits.
    - ↳ para que otras empresas fabriquen
  - Muy útil para dispositivos móviles ( Alta relación: MIPS / watt )
  - Extensiones: Thumb, Jazelle, SIMD (Neon), VFP
    - ↳ muchas instrucciones por segundo y consumen poco energía

# Licencias de ARM

- Samsung
- AMD
- Broadcom
- ST-Ericsson
- Toshiba
- NVIDIA
- Texas Instruments
- Philips



compran licencias

# Diseño de procesadores

ARM diseña nuevos procesadores

↳ Software dedicado para simular un procesador

Se utiliza software para simular el  
microcódigo que realizan las compuertas de silicio  
↳ como voy a resolver nuevas instrucciones

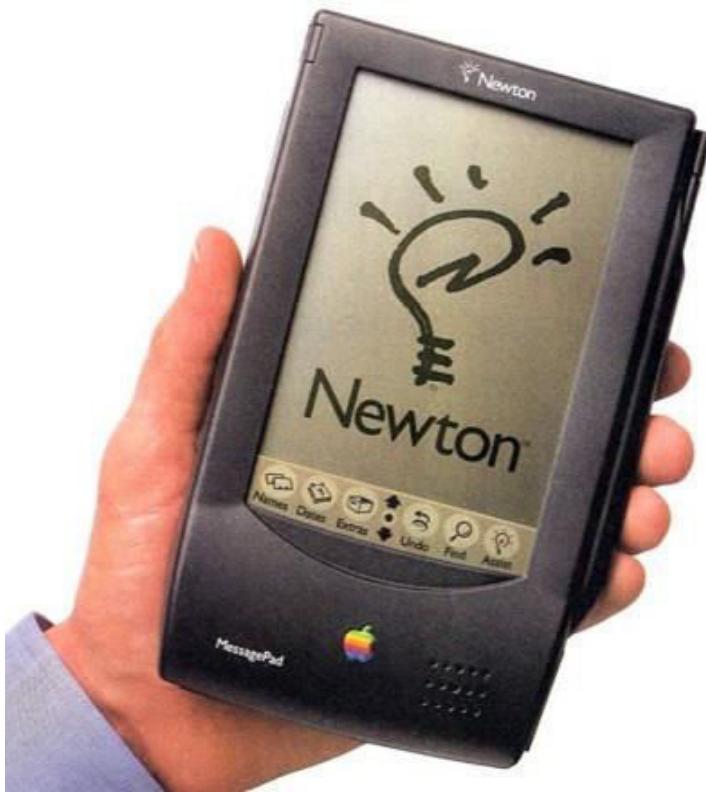
- Simulación lógica
- Simulación eléctrica
- Simulación térmica

Ej. ADD Ax, Bx ↳ como se resuelve  
esta instrucción = microcódigo

Ejemplo de lenguaje : VERILOG

Lenguaje de tipo HDL (Hardware Description Language)

# Apple Newton



- Se lanza en 1993
- ARM 610 RISC
- Muchos bugs, caro, fracasó.

# Nokia 6110



- Texas Instrument fabrica un ARM y lo prove a Nokia.
- Se lanza en 1997
- Arquitectura ARM 7
- Primer telefono GSM (Global System for Mobile communications) con ARM.
- Exito total.

# Ejemplos de productos



Features of  
ARM  
⇒ Fabricantes diseñan  
procesador que tengan  
todo inmerso (ausco, memoria, etc)

- iPod de Apple.
- Contiene un ARM7TDMI de 90Mhz
- 32 Mb de DRAM.
- Manejan discos de 20 y 40 GB.

# Crecimiento (2002-2005)

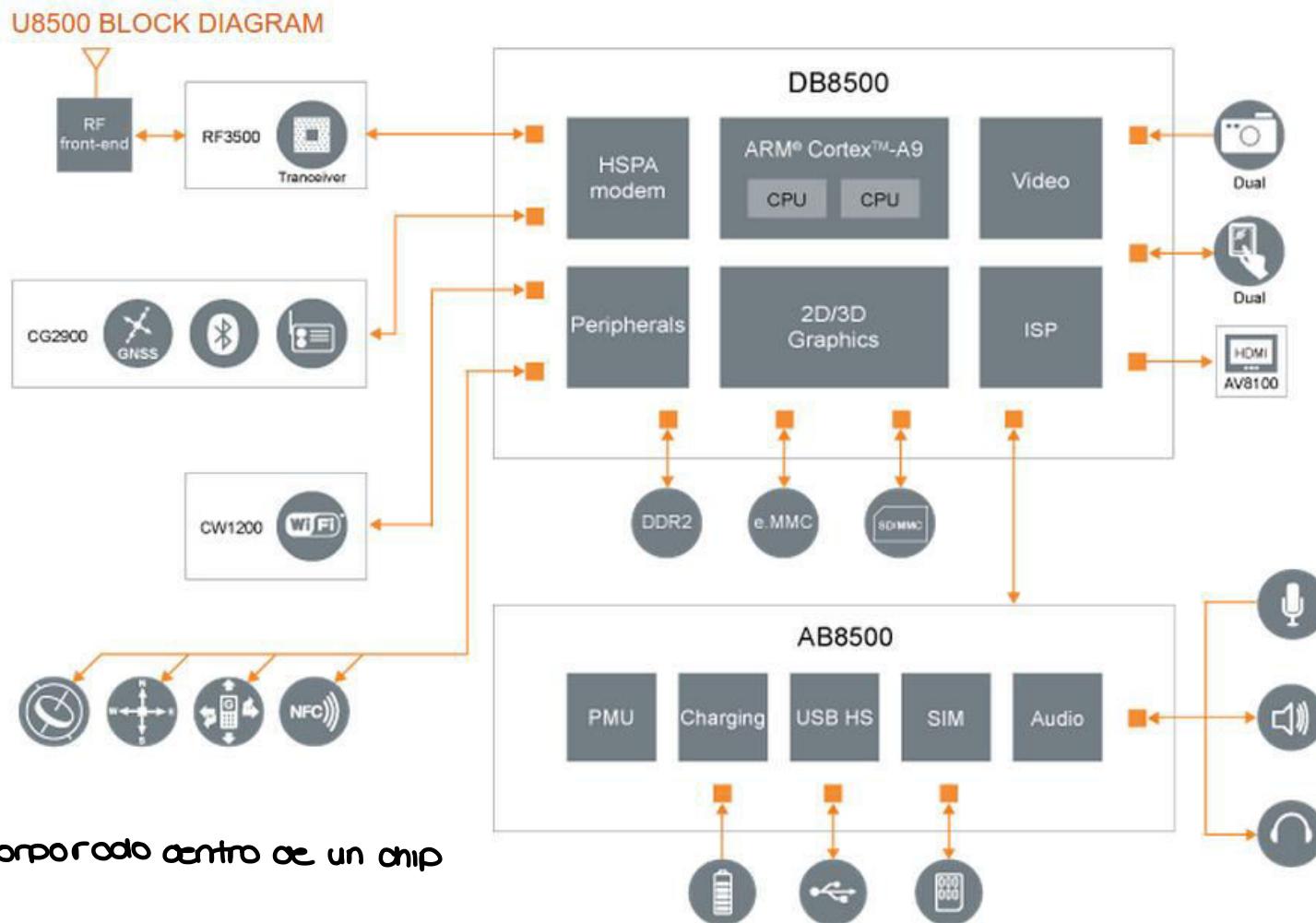
- El procesador ARM se vuelve mucho mas pequeño.
- Las empresas no tienen personal para diseñar su propio procesador o las herramientas para usarlo.
- Se orienta al SoC (System on a Chip)
  - ➡ meter todo dentro de un mismo chip
- Se lanza el **ARM926EJ-S** soporta Linux, WindowsCE y Symbian. Soporte DSP y aceleración Java. Cinco niveles de Pipeline
  - ➡ procesar señales de manera elaborada

# Systems on Chip (SoC)

- Un SoC está integrado por:
  - Procesador
  - Memorias (ROM, RAM, Flash) *ROM que se puede escribir con velocidad*
  - Osciladores
  - Conversores A/D<sup>t</sup> y D/A<sup>z</sup>
  - Interfaces (USB, Ethernet, USART)
    - <sup>t</sup> analogico / digital Ej. microfono
    - <sup>z</sup> digital / analogico Ej. parlante

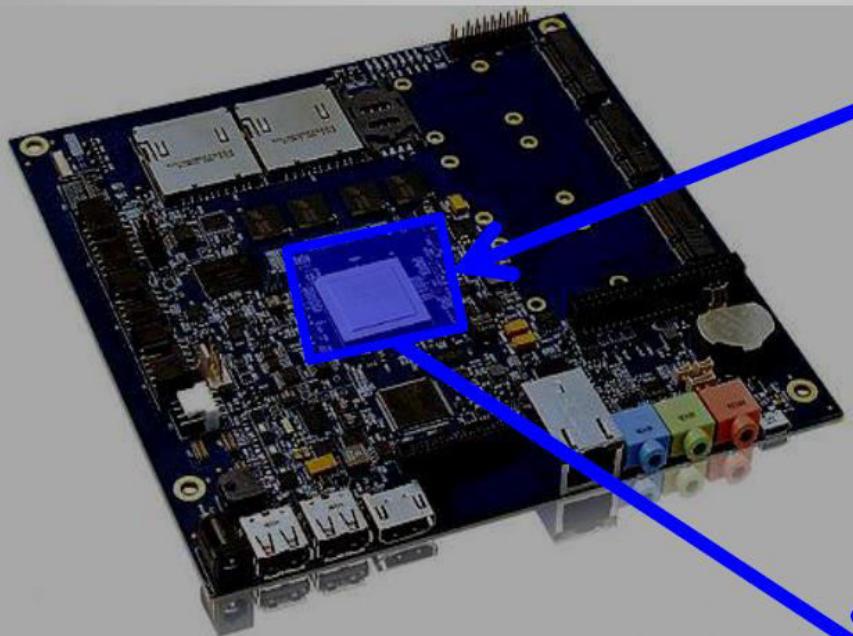
# Systems on Chip (SoC)

- Ejemplo: ST-Ericcson
  - Novathor U8500



# Systems on Chip (SoC)

## ARM Embedded Processor



Main CPU Chip  
(SoC)

ARM Embedded  
Processor



# Linksys WRT54GP2 Wireless-G Broadband Router



- Basado en un core ARM9.

- ⇒ se pueden poner imágenes de Linux
- ⇒ Se pueden crear VPN, controles de tráfico, etc

## Era Cortex (2005-2012)

- Smartphones !
- ARM responde con CortexA9
- Hoy ARM tiene el 96% del Mercado móvil.

# ARM Cortex A9

⇒ nombre del procesador

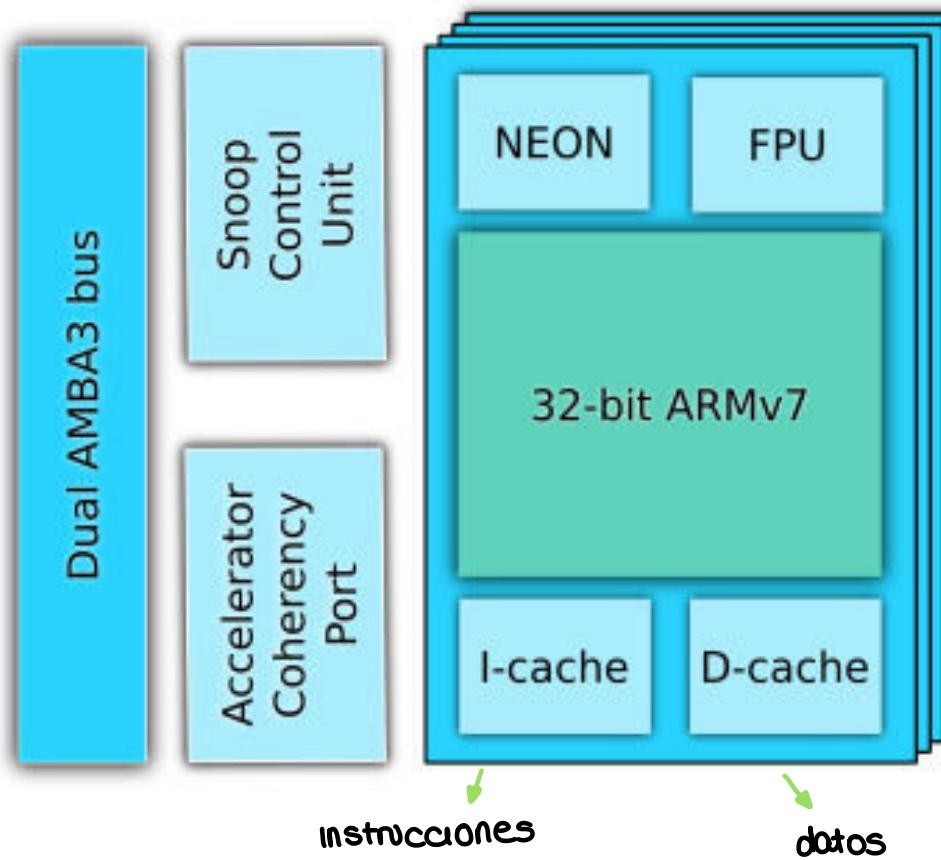
- Arquitectura ARMv7-A
- Simple ó Multi Procesador (hasta 4)
  - ↳ nucleos
- Varios niveles de control de consumo
- Basado en RISC
  - ↳ set de instrucciones reducido
- Instrucciones ARM y Thumb
- 37 registros de 32 bits

# ARM Cortex A9

- Velocidad de clock entre 800MHz y 2 GHz ↗ velocidad ac clock variable
- Floating Point
- Controlador de caché L2
- Jazelle (Optimizador Java)
- DSP (Digital Signal Processing)

# ARM Cortex A9

- Caché de Intrucciones/Datos de 16,32 o 64 KB
- Arquitectura Harvard modificada



⇒ Ram = Von Neumann

⇒ Cache = Harvard

Por eso es Harvard modificada

# ARM Cortex A9

SoC que lo usan:

- Apple A5 (Ipad 2 y 3, Iphone 4s, Ipad mini)
  - L1 Cache 32 KB de instrucciones y 32 KB de datos ➔ Harvard
  - L2 Cache 1 MB  
➔ genérico
- OMAP4 (Motorola Razr, kindle Fire)
- Exynos 4 (Samsung Galaxy S3, Samsung Note)
  - Quad Core ARM-Cortex-A9
  - CPU 1.4-1.6 GHz

# Iphone X



- A11 Bionic chip with 64-bit architecture
- Microarquitectura: ARMv8-A
- Cache L1: 32 KB instruction, 32 KB data

# Arquitecturas y familias

De cada arquitectura solo distintos procesadores

Architecture	Family
ARMv1	ARM1
ARMv2	ARM2, ARM3
ARMv3	ARM6, ARM7
ARMv4	StrongARM, ARM7TDMI, ARM9TDMI
ARMv5	ARM7EJ, ARM9E, ARM10E, Xscale
ARMv6	ARM11, ARM Cortex-M
ARMv7	ARM Cortex-A, ARM Cortex-M, ARM Cortex-R
ARMv8	ARM Cortex-A57, ARM Cortex-A53

Misma arquitectura = compatibles a nivel set de instrucciones

# Arquitecturas y familias

## Development of the ARM Architecture

comó se define la arquitectura

- Processor Architecture = Instruction Set + Programmer's model



ARM7TDMI  
ARM922T  
  
Thumb  
instruction set



ARM926EJ-S  
ARM946E-S  
ARM966E-S  
  
Improved  
ARM/Thumb  
Interworking  
  
DSP instructions  
  
Extensions:  
Jazelle (5TEJ)



ARM1136JF-S  
ARM1176JZF-S  
ARM11 MPCore  
  
SIMD Instructions  
Unaligned data support  
  
Extensions:  
Thumb-2 (6T2)  
TrustZone (6Z)  
Multicore (6K)



Cortex-A8/R4/M3/M1  
Thumb-2  
  
Extensions:  
v7A (applications) – NEON  
v7R (real time) – HW Divide  
V7M (microcontroller) – HW Divide and Thumb-2 only

- Note: Implementations of the same architecture can be very different
  - ARM7TDMI - architecture v4T. Von Neuman core with 3 stage pipeline
  - ARM920T - architecture v4T. Harvard core with 5 stage pipeline and MMU

≠ procesadores PERO misma arquitectura

# Arquitecturas y familias

## Processor Modes mas modos de trabajo

- The ARM has seven basic operating modes:
  - Each mode has access to own stack and a different subset of registers
  - Some operations can only be carried out in a privileged mode

Mode	Description	
Supervisor (SVC)	Entered on reset and when a Software Interrupt instruction (SWI) is executed	Privileged modes
FIQ	Entered when a high priority (fast) interrupt is raised	
IRQ	Entered when a low priority (normal) interrupt is raised	
Abort	Used to handle memory access violations	
Undef	Used to handle undefined instructions	
System	Privileged mode using the same registers as User mode	
User	Mode under which most Applications / OS tasks run	Unprivileged mode

# Set de Registros

## Current Visible Registers

Abort Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
cpsr
spsr

Banked out Registers

User

r13 (sp)  
r14 (lr)

FIQ

r8  
r9  
r10  
r11  
r12  
r13 (sp)  
r14 (lr)

IRQ

r13 (sp)  
r14 (lr)

SVC

r13 (sp)  
r14 (lr)

Undef

r13 (sp)  
r14 (lr)

spsr

spsr

spsr

spsr

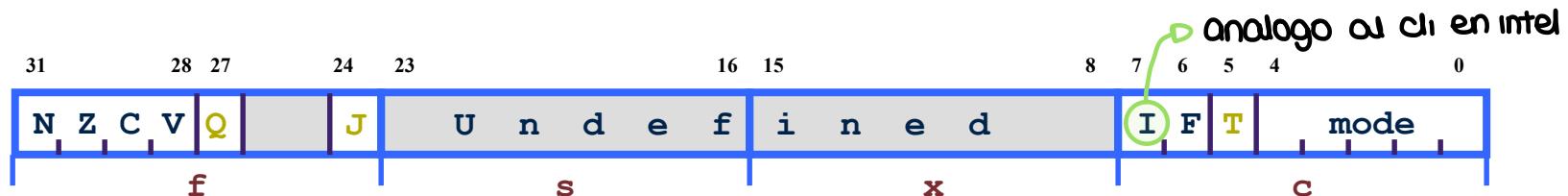
# Set de Registros

- ARM has 37 registers all of which are 32-bits long.
  - 1 dedicated program counter *analogo al IP en intel*
  - 1 dedicated current program status register *analogo a los flags*
  - 5 dedicated saved program status registers
  - 30 general purpose registers
- The current processor mode governs which of several banks is accessible. Each mode can access
  - a particular set of r0-r12 registers
  - a particular r13 (the stack pointer, sp) and r14 (the link register, lr)
  - the program counter, r15 (pc)
  - the current program status register, cpsr

Privileged modes (except System) can also access

- a particular spsr (saved program status register)

# Registros - Flags



## Condition code flags

- N = Negative result from ALU
- Z = Zero result from ALU
- C = ALU operation Carried out
- V = ALU operation oVerflowed

## Sticky Overflow flag - Q flag

- Architecture 5TE/J only
- Indicates if saturation has occurred

## J bit

- Architecture 5TEJ only
- J = 1: Processor in Jazelle state ➡ Operation code de Java

## Interrupt Disable bits.

- I = 1: Disables the IRQ.
- F = 1: Disables the FIQ. (Fast IRQ)

## T Bit

- Architecture xT only
- T = 0: Processor in ARM state
- T = 1: Processor in Thumb state

↳ instrucciones

de 16 bits

## Mode bits

- Specify the processor mode

# Características generales

- Casi todas las instrucciones se ejecutan en un ciclo de clock y tienen tamaño fijo.
- Todas las familias de procesadores comparten el mismo conunto de instrucciones
- Tipo de datos de 8/16/32 bits.
- Pocos modos de direccionamiento
- No se crea fragmentación de memoria

## INTEL :

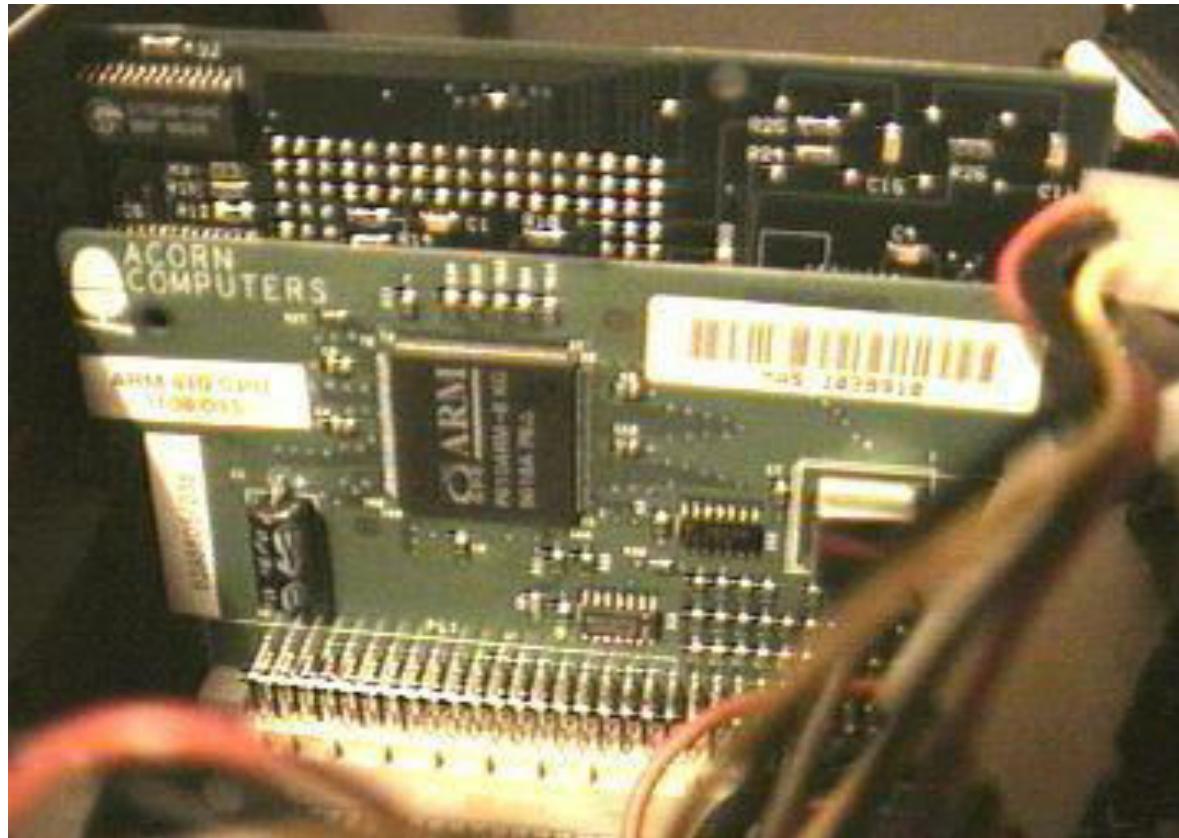
- NOP (no op code) = 1 ciclo de clock
- NOV AX,[100h] = 12 ciclos de clock
- ARM alarga los ciclos de clock ➔ mas facil calcular cuanto tardan los programas ➔ mejora el sincronismo  
lo mismo con el tamaño ➔ mas facil moverse en memoria para el IP  
↳ rellena el tamaño que ocupo una instrucción (Ej. con NOPs) ➔ mas facil correr código multicore

# MIPS

Millones de instrucciones por segundo

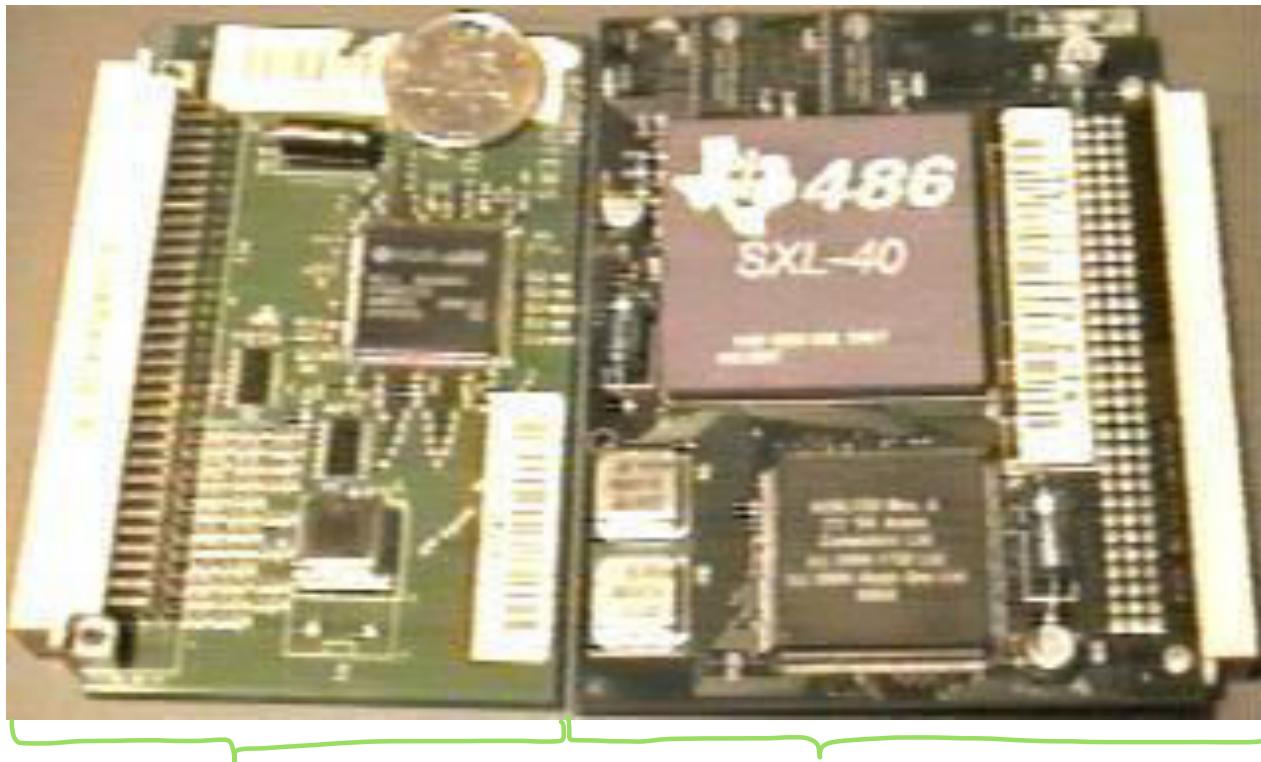
Procesador	Clock	MIPS
80486	100 Mhz	70
Pentium I	60 Mhz	100
ARM720T	60 Mhz	60
ARM9TDMI	180 Mhz	200

# Diferencias de tamaños



**ARM610 (33MHz) y 486SXL-40 (33MHz)**

# Diferencias de tamaños



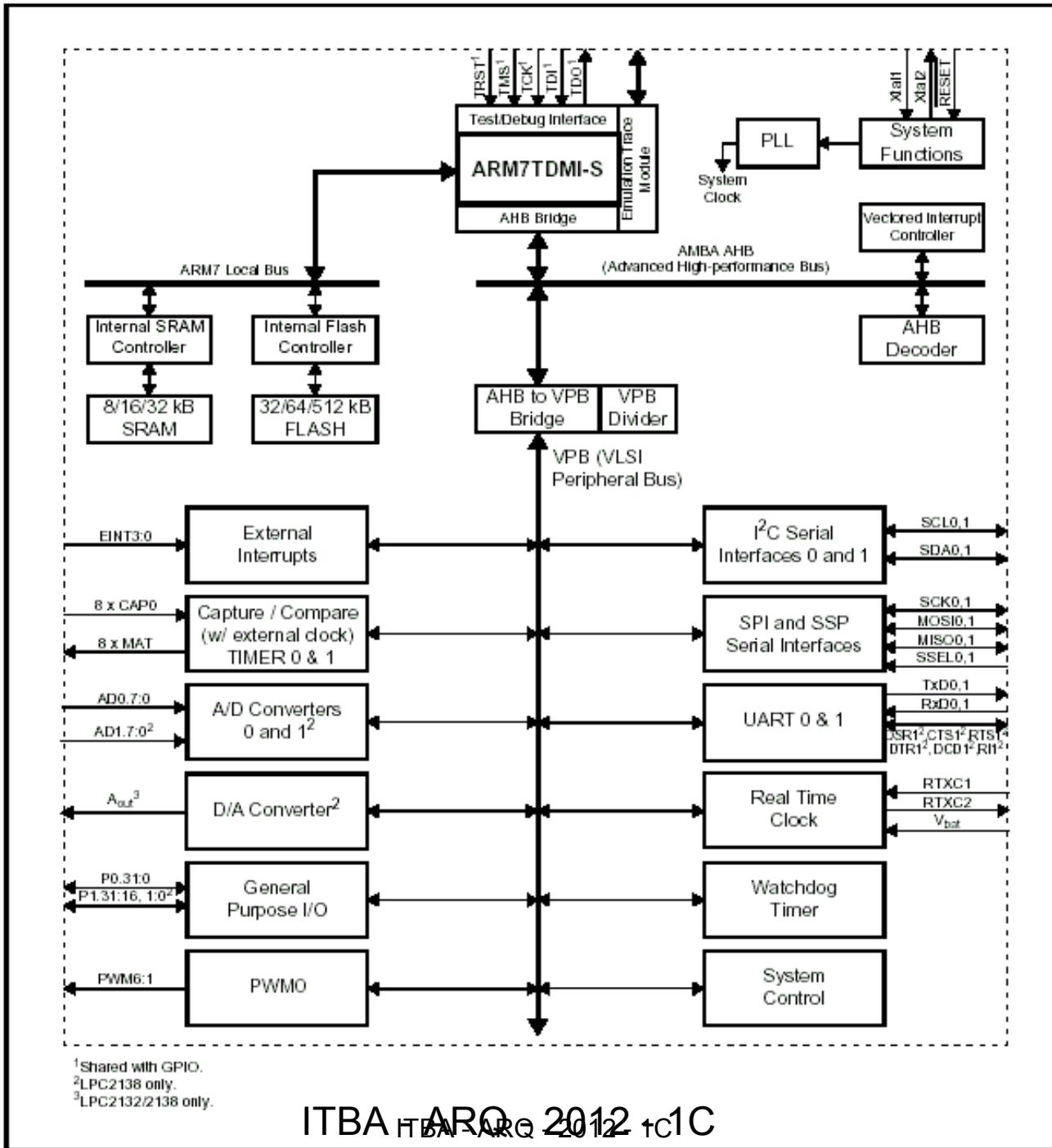
**ARM610 (33MHz) y 486SXL-40 (33MHz)**

# Familia LPC213x

- Microcontrolador 16/32-bit ARM7TDMI-S
- RAM de 8/16/32 kB en chip.
- 32/64/512 kB de memoria Flash.
- 1 ó 2 Conversores A/D con 10 bits de resolución.
- Conversor D/A con 10 bits de resolución.
- 2 contadores/timer de 32 bits.
- Real Time Clock con bateria independiente
- Interfaces serie ( UART, I2C, SPI y SPP )
- Controlador de interrupciones

# Familia LPC213x

- Oscilador en chip. Entre 1 y 30 MHz
- Interrupciones manejadas con vector. ↗ análogo al IOT
- Hasta 9 interrupciones de hardware
- Modo de conservación de energía.
- Encendido por interrupcion de hardware.



# Memorias en LPC213x

## Memoria Flash On-Chip

- En ella se puede almacenar código y dato.
- Se puede programar a través del puerto serie
- Se puede programar o borrar mientras está corriendo el código
- (muy útil para upgrade firmware)
- Soporta 10.000 borrados y/o escrituras
- 10 años puede retener la información guardada.

## RAM

- Se puede utilizar para código y datos.

## Mapa de memoria

No tiene mapa EIS

\* 512 MB para periféricos ↡ mapeados en memoria

Ya tiene pensado donde va la RAM y la memoria no volatil.

Desventaja: uno tiene que buscar que chip le sirve

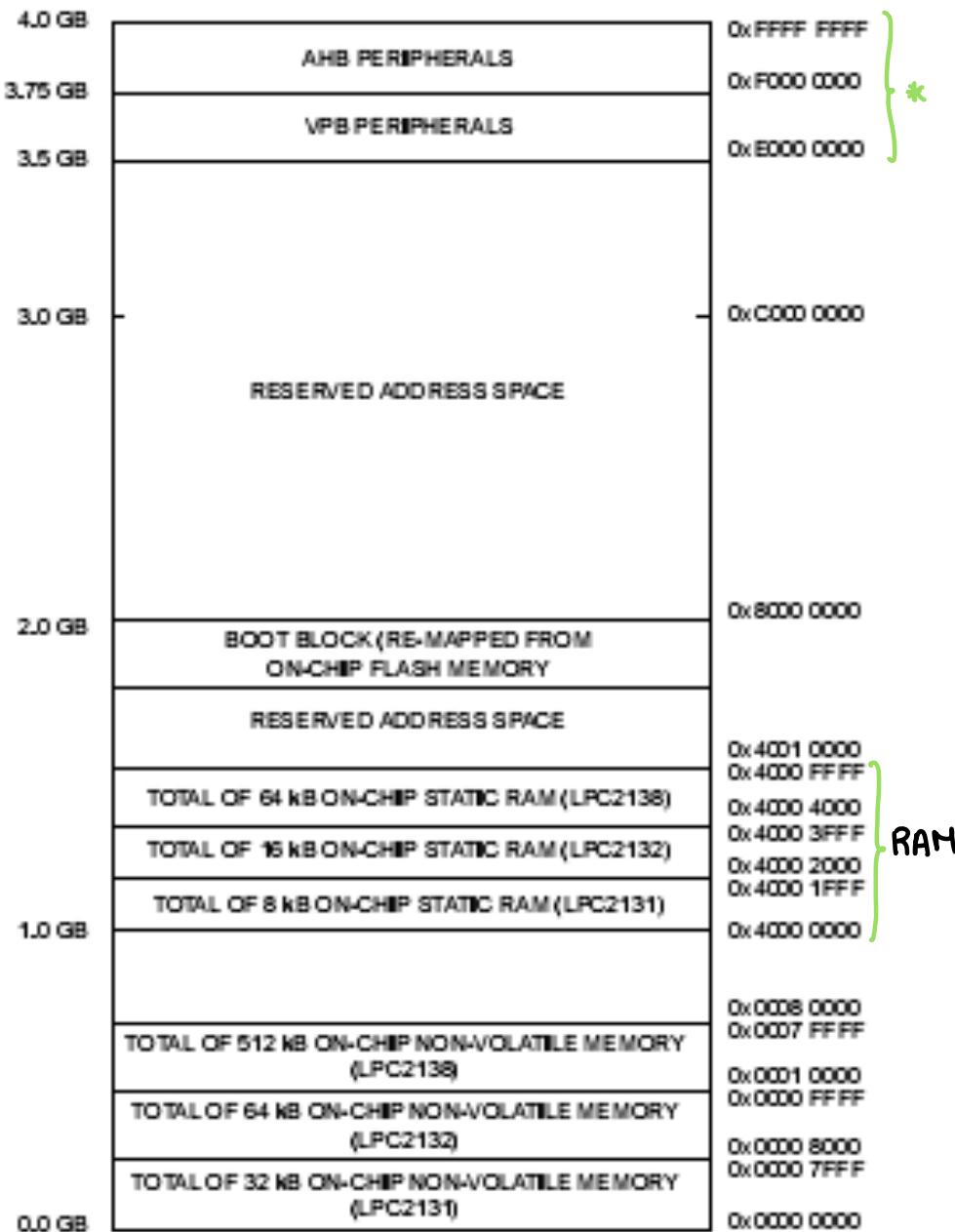
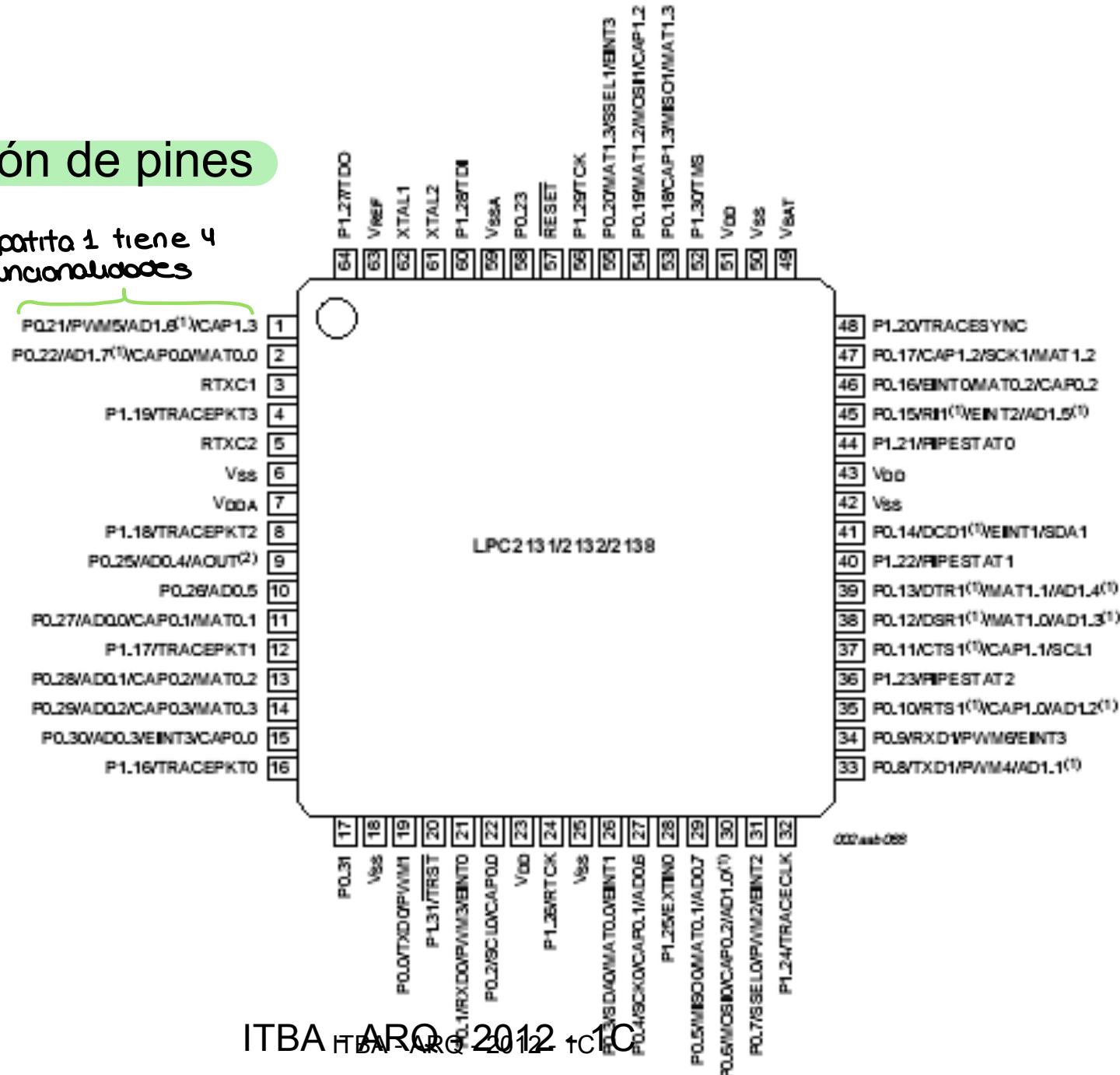


Fig 3. LPC2131/2132/2138 memory map.

# Distribución de pines

parte 1 tiene 4  
funcionalidades



# Distribución de pines

Como vemos en la imagen anterior, varios de todos los pines puede tener diferentes funcionalidades.

Esto permite una mayor flexibilidad en la utilización del procesador.

Para decidir cual de las funcionalidades utilizar existen 3 registros que se permiten seleccionarlas.

Address	Name	Description	Access
0xE002C000	PINSEL0	Pin function select register 0	Read/Write
0xE002C004	PINSEL1	Pin function select register 1	Read/Write
0xE002C014	PINSEL2	Pin function select register 2	Read/Write

Las funcionalidades de cada pin se combinan a traves del procesador  combinando los pin selector.

# Distribución de pines

En la tabla anterior podemos ver que los 3 registros son en realidad 3 direcciones de memoria.

En este tipo de procesadores, muchos registros están mapeados en memoria, a diferencia de la familia Intel.

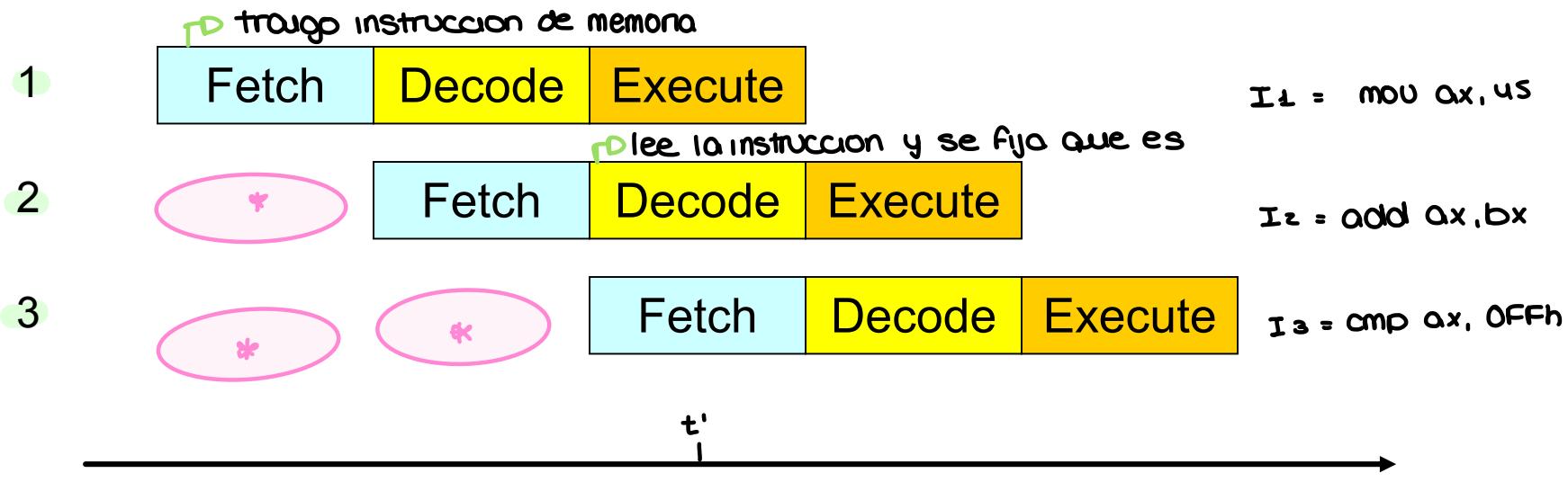
Veremos ahora las hojas de datos de los integrados para comprender mejor su funcionamiento interno.

# Pipeline en ARM7

➡ EN TODOS LOS PROCESADORES

Posee un pipeline de 3 etapas

↳ fetch, decode y execute ➡ etapas independientes



Instrucciones que un procesador va correr = memoria.

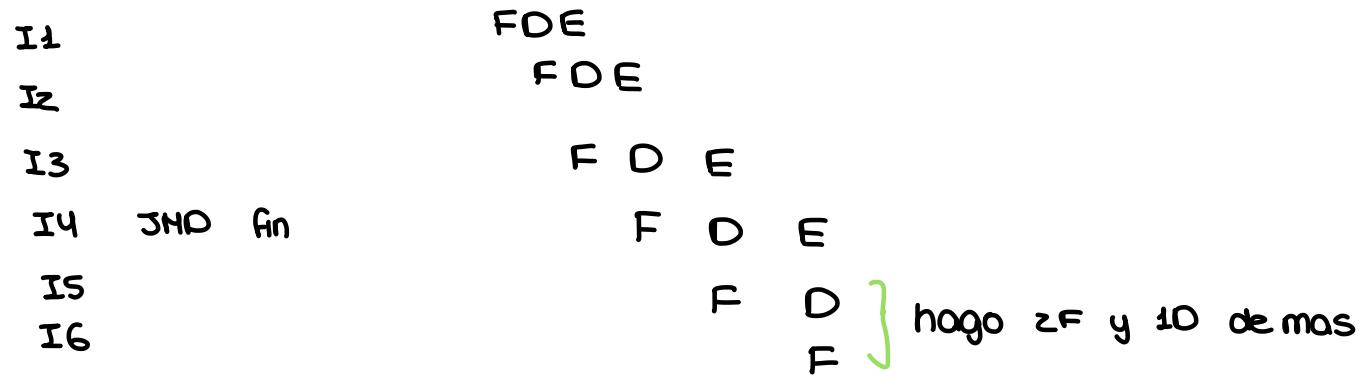
tiempo

Registro que apunta a la sig. instrucción a ejecutarse = IP.

Una instrucción de salto, provoca que se vacie el pipeline

En t', ejecuto I<sub>1</sub>, decodifico I<sub>2</sub> y fetcheo I<sub>3</sub>.

\* CASO en el que arranca la PC PERO normalmente se realizan 3 etapas simultáneamente



fin :

⇒ un JMP provoca un vacío de pipeline = perdida de performance

I<sub>7</sub>

Si cada JMP vacío el pipeline ⇒ hay un problema.

I<sub>8</sub>

Existe Branch Prediction ⇒ predice si hace JMP o no ⇒ % alto de hits.

Obs : El que mas tarde es el fetch

Obs : El execute depende de la instrucción (NOP = rápido, mov a memoria es lento)

⚠ Si I<sub>2</sub> = mov [200h], ah e I<sub>3</sub> = sub ax, bx ⇒ no se puede ejecutar I<sub>2</sub> al mismo tiempo que fetcheo I<sub>3</sub>

⇒ el fetcheo lo hago cuando termina el execute

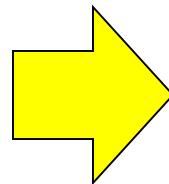
¿Porque? Necesito los buses para el fetch PERO lo estoy usando para el execute y tengo un solo par de buses (arquitectura von Neumann) ⇒ por eso se usa Harvard modificado

# Instrucciones en ARM

## Instrucciones condicionales

➡ para evitar que se vacie el pipeline

```
If ( a==b)  
{ a++; } else { a--; }
```



```
Cmp R1,R2 /* se asume R1 vale a y R2 b */  
Addeq R1,#1 /* suma 1 a R1 */  
Subne R1,#1 /* resta si no es igual */
```

Vemos que con las instrucciones condicionales se evitan los saltos en el código que demoran la ejecución del programa

# Instrucciones en ARM

## Instrucciones condicionales

31                          28  
Condición (4bits)      (Otros campos de la instrucción)  
*lee primero estos bits, si cumple hace el resto*

Antes de la ejecución de cada instrucción se chequea los bits de condición ( 31:28 ) para determinar si se debe ejecutar o no.

Por ejemplo:

La condición '0000' significa EQUAL. Por lo tanto la instrucción sólo podrá ser ejecutada si el flag Z ( zero ) está activo.

**Obs:** no existe en Intel porque las instrucciones tienen tamaños variables

todos los procesadores tienen pipeline **PERO** no todos tienen instrucciones variables.

# Condiciones

- The possible condition codes are listed below:
  - Note AL is the default and does not need to be specified

Suffix	Description	Flags tested
<b>EQ</b>	Equal	<b>Z=1</b>
<b>NE</b>	Not equal	<b>Z=0</b>
<b>CS/HS</b>	Unsigned higher or same	<b>C=1</b>
<b>CC/LO</b>	Unsigned lower	<b>C=0</b>
<b>MI</b>	Minus	<b>N=1</b>
<b>PL</b>	Positive or Zero	<b>N=0</b>
<b>VS</b>	Overflow	<b>V=1</b>
<b>VC</b>	No overflow	<b>V=0</b>
<b>HI</b>	Unsigned higher	<b>C=1 &amp; Z=0</b>
<b>LS</b>	Unsigned lower or same	<b>C=0 or Z=1</b>
<b>GE</b>	Greater or equal	<b>N=V</b>
<b>LT</b>	Less than	<b>N!=V</b>
<b>GT</b>	Greater than	<b>Z=0 &amp; N=V</b>
<b>LE</b>	Less than or equal	<b>Z=1 or N!=V</b>
<b>AL</b>	Always	

# TDMI

T : THUMB ( set de instrucciones )

Set de instrucciones de 16 bits que se suman a las estandar de 32 bits.

D : Debug Interface ( JTAG )

( Permite relaizar un HALT del micro para debug )

M : Multiplicador ( en hardware, de alta resolución )

Componente interno del micro, logra resultados de 64 bits.

I: Interrupt ( interrupciones rápidas )

- Thumb sirve para ahorrar espacio, mantener compatibilidad hacia otras...
- Debuggear en tiempo real lo que se ejecuta.

# THUMB

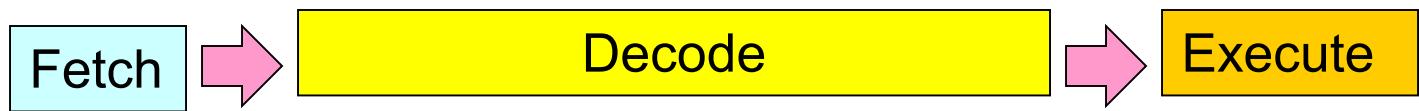
El microprocesador tiene 2 sets de instrucciones:

El ARM clásico de 32 bits y el Thumb de 16 bits.

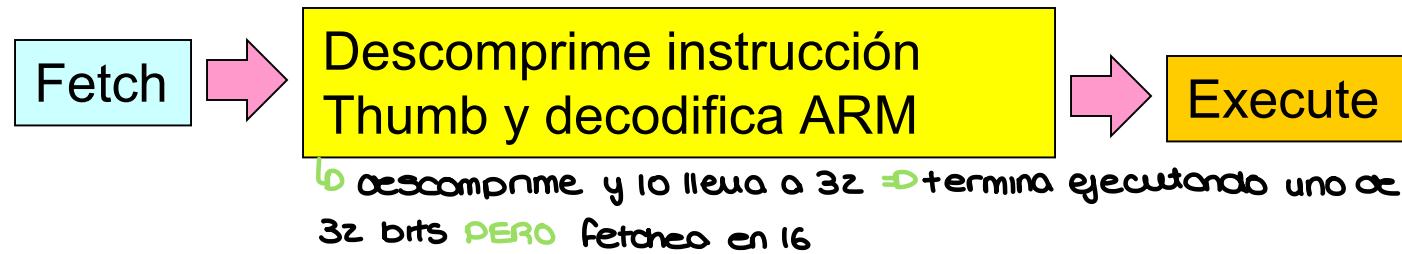
En Thumb:

- Instrucciones de 16 bits.
- Se descomprimen en forma dinámica en el modulo de “decode” del pipeline

Pipeline  
ARM



Pipeline  
Thumb



# THUMB

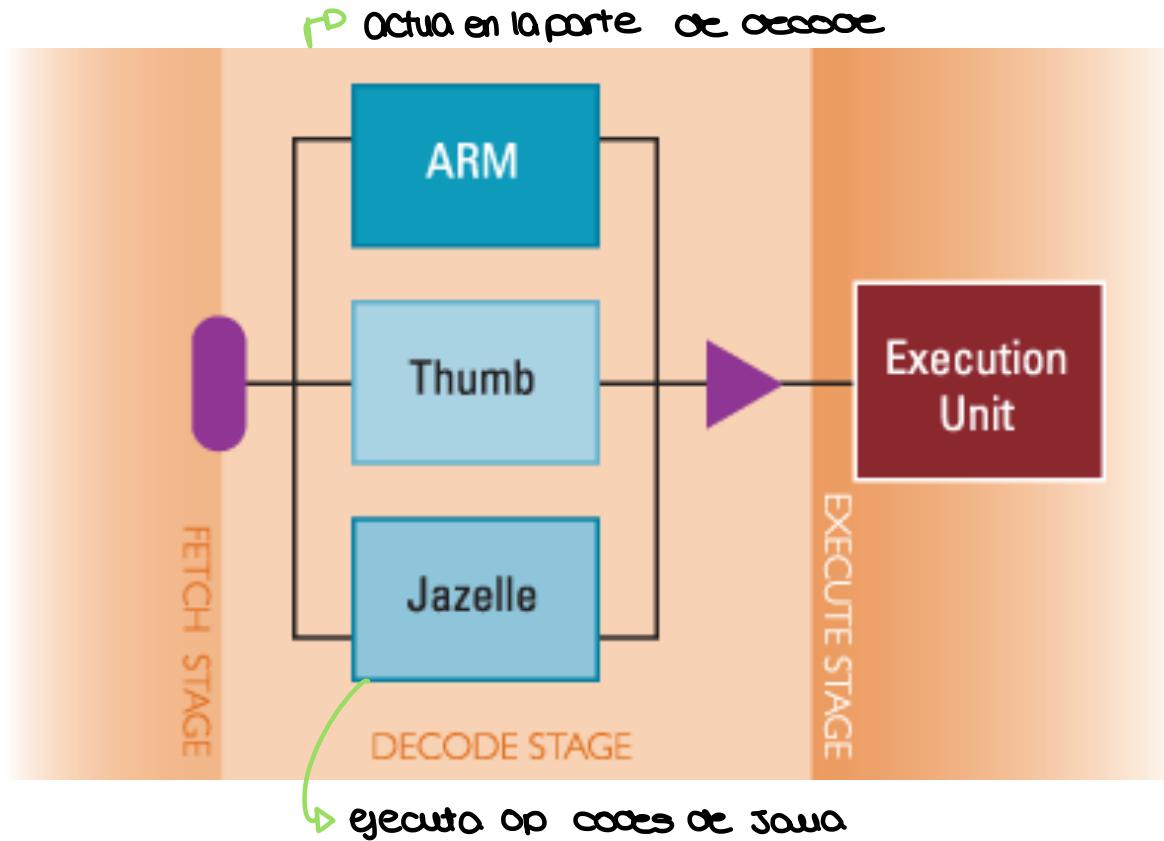
En Thumb:

- Se gana entre un 35% y un 40% de memoria comparado con el set de 32bits
- Cada instrucción de 16 bits tiene se correspondiente en 32 bits.
- Al usar enteros de 32 bits la ventaja la tiene el set ARM de 32 bits.
- Al funciones, por ejemplo, de manejo de caracteres, es mejor Thumb
- Se pude switchear de Thumb a ARM en forma dinamica, para aplicaciones combinadas ( 16 y 32 bits ) y asi no perder performance.
- La mayoría de las instrucciones Thumb no son condicionales.
- La mayoría de las instrucciones utilizan 2 operandos
- En ARM la mayoría de las instrucciones utilizan 3 operandos.
- Debido a la compresión se pierden algunas funcionalidades especiales

# Extensión Jazelle

Los procesadores con extension Jazelle, son capaces de ejecutar bytes code de Java directamente en hardware.

Las instrucciones Java que no posee las emula con las instrucciones ARM.



# ARM9TDMI

- Es el reemplazo de los ARM7
- Es compatible a nivel binario con ARM7 
- Tiene un pipeline de 5 niveles en lugar de 3. Esto permite aumentar la frecuencia del clock.
- Con memoria cache.
- Arquitectura Harvard Modificada. Instrucciones y datos separados en cache pero mismo espacio de memoria.

# Computación paralela

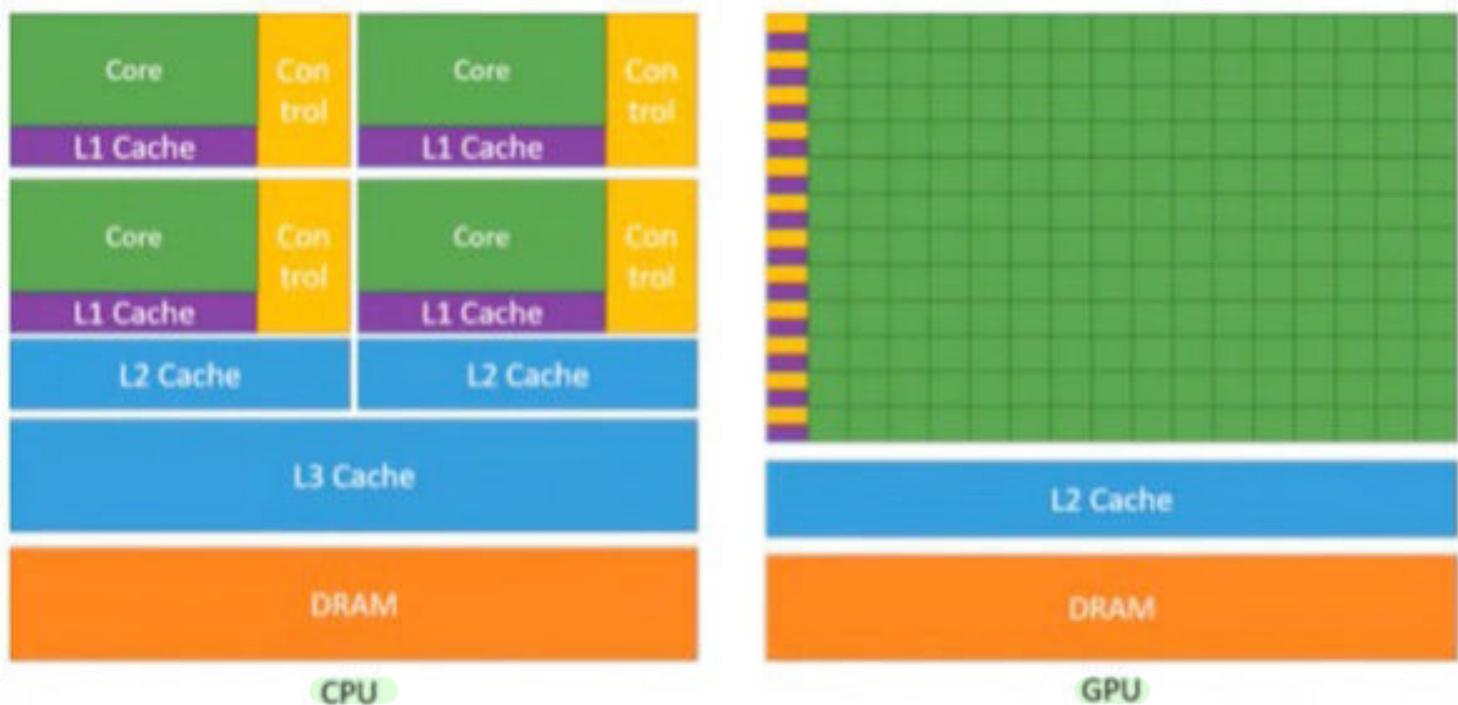
- División de tareas de computo
- Puede encontrarse en diferentes niveles

## EJEMPLOS:

- (1) Pipeline
- (2) Threads
- (3) Multi-cores
- (4) APIs (Ej. MPI - Message Passing Interface)
- (5) Clusters - Grids

## GPU

- ⇒ unidades de procesamiento gráfico
- Proveen mas instrucciones por segundo
- Diseñada para ejecutar miles de threads en paralelo
- Menor velocidad de ejecución de un thread que una CPU
- PERO mas cantidad i.e. mas hilos de ejecución por segundo



## Machine learning

En redes neuronales los nodos pueden realizar su tarea computacional de manera paralela.  
Pega bien con la GPU porque cada nodo ejecuta un código independiente del otro y luego se comunican los resultados de cada nodo  $\Rightarrow$  hilos de ejecución independientes entre sí.

## Usos de GPU

- Graficos
- Machine learning
- Minería de Criptomonedas
- Password cracking

## Ejemplo Titan XP

- 12 Gb de memoria
- 30 Multiprocesadores
- 128 CUDA cores por multiprocesador (= 3840 CUDA cores)  $\Rightarrow$  3840 threads en forma paralela.
- 3Nb de Cache L2
- 2048 de thread máximo por Multiprocesador