

72.08 - Arquitectura de Computadoras

Apuntes



1Q - 2024

Índice

Unidad 1 - Binarios.....	4
Arquitecturas.....	4
Programas y Binarios.....	4
Programa en Memoria.....	5
Llamadas a funciones (CALL).....	7
System Calls (syscall).....	8
Unidad 2 - Procesadores y Lenguaje ASM (Assembler).....	10
Registro de un Procesador.....	10
Registros (32 bits).....	12
Lenguaje Assembler (Intel).....	13
Modos de Direccionamiento.....	15
Registro de Flags.....	16
Debugger.....	18
Linkedición con GCC.....	20
Unidad 3 - Assembler y C.....	22
Mezcla de lenguajes en ejecutables.....	22
Repaso de Pila.....	22
Instrucción RET.....	23
Instrucción CALL.....	24
Análisis de C.....	24
Manejo de la pila en C.....	25
Convenciones en C.....	26
Llamada de ASM a C.....	27
Llamada de C a ASM.....	28
Inline Assembler.....	29
Salidas en ASM.....	29
Análisis de Manejo de Pila.....	31
Unidad 4 - Memoria y Entrada y Salida.....	36
Transmisión Digital.....	36
Tipos de Arquitectura.....	37
Sistema de Entrada y Salida.....	37
CPU.....	38
Mapa de Memoria.....	41
Registros de Intel.....	42
Memorias.....	42
Tiempo de Acceso.....	43
Memorias - Estructura.....	43

Memoria Comercial.....	44
Integrados Compuertas y Decodificadores.....	45
Decodificación de Hardware.....	46
Decodificador 3 a 8.....	49
Buses.....	52
Tipos de Decodificación de Hardware.....	54
Periférico Estándar.....	54
Sistema de Entrada y Salida.....	57
Mapeo en memoria.....	57
Interrupciones.....	58
Rutina de Atención de Interrupción.....	59
PIC (Controlador Programable de Interrupciones).....	59
Interrupciones de Software.....	61
Servicio de BIOS.....	61
Interrupciones de Hardware por Default.....	61
Excepciones.....	62
 Unidad 5 - Modo Protegido.....	64
Comutación de Tareas.....	64
Protección de Tareas.....	65
Memory Management Unit (MMU).....	65
GDT y LTD (Unidad de segmentación).....	66
Descriptores de Segmento.....	66
Dirección Lineal.....	68
Descriptores de Sistema.....	69
Interrupciones en PRE TPE.....	70
Privilegios de E/S.....	70
Memoria Virtual.....	71
Elección de Intel para 32 bits.....	73
 Unidad 6 - Memoria Caché.....	78
 Unidad 7 - ARM.....	82
Microprocesadores ARM.....	82
Pentium.....	83
Intel 64 / EM64T.....	83
Microprocesadores ARM.....	84
Diseño de Procesadores.....	84
System on a Chip (SoC).....	85
ARM Cortex A9.....	85
Registros.....	87

Características Generales.....	88
Mapa de Memoria.....	89
Pipeline en ARM7.....	89
Instrucciones en ARM.....	90

Unidad 1 - Binarios

Arquitecturas

Arquitectura PC: Procesador + RAM + ...



La PC es creación de IBM, pero se puede copiar, por eso hay variedad de fabricantes. Se creó con la idea de Open Hardware (el equivalente a Open Source). Lo mismo pasa con los teléfonos entre Android y los Apple.

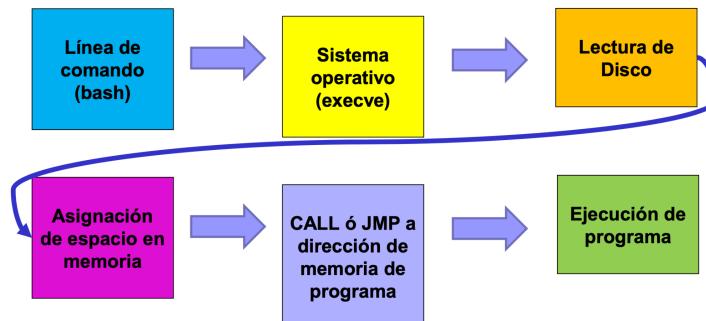


¿Cuál es la diferencia entre una consola y una PC si tienen también un procesador AMD (clon de Intel)? Es por el Sistema Operativo y porque la consola es sólo para juegos.

Programas y Binarios

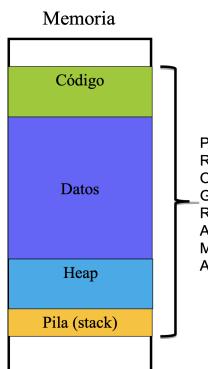
La línea de comandos llama al sistema operativo, que lee nuestro “a.out” en el disco. Luego se fija si hay espacio en memoria y lo “sube”, para luego hacer un CALL o JMP

y leerlo. ¿Por qué lo sube en vez de leerlo de memoria? Porque la memoria es mucho más rápida. Por ejemplo, si el código reutiliza una función o usa ciclos, va a ser mucho más rápido leerlo si está en la memoria que volver al disco para buscar la instrucción.



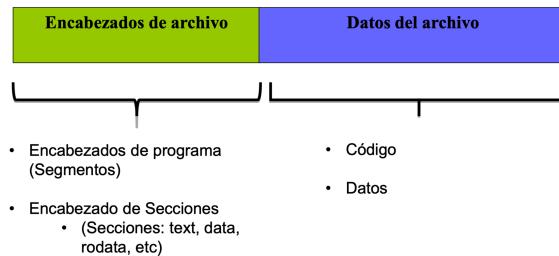
Cuando **compilamos** y **linkeditamos** con gcc, lo que pasa es lo siguiente. El gcc invoca primero al preprocesador cpp, el cual toma el código fuente original y realiza las macro expansiones de directivas como el #include o el #define. Luego, gcc toma la salida y convierte en código C al equivalente en ASM (por default en sintaxis AT&T), generando un .S. El compilador GNU de ASM se encarga de generar el código objeto (es decir en binario - extensión .O) para ser luego procesado por el microprocesador. Por último, el linkeditor crea el ejecutable, validando primero todas las referencias a funciones y uniendo los programas objetos necesarios. El archivo ejecutable es el archivo binario (código + syscalls) + header (propio de cada SO, contiene información para este).

Programa en Memoria



- Código: Instrucciones del programa
- Datos: Variables estáticas y globales que se inician 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.

Para todo esto, tiene que haber punteros, para saber dónde está guardado. Son tan importantes que el procesador los guarda como registros. Van a estar físicamente en el procesador. Van a haber punteros a código, a datos y a pila.



Encabezados de archivo (header) es la información para el sistema operativo que se agrega. En Linux, con el comando **file** podemos ver el encabezado de los archivos (en linux no hay extensiones, entonces puedo guardar archivos ejecutables como .jpg, porque no significa nada).

Otro programa es **strings**, que va a extraer las partes de datos (de lo que vimos en el gráfico de arriba). Extrae todas las cadenas que tengan representación ASCII.

Ejemplo:

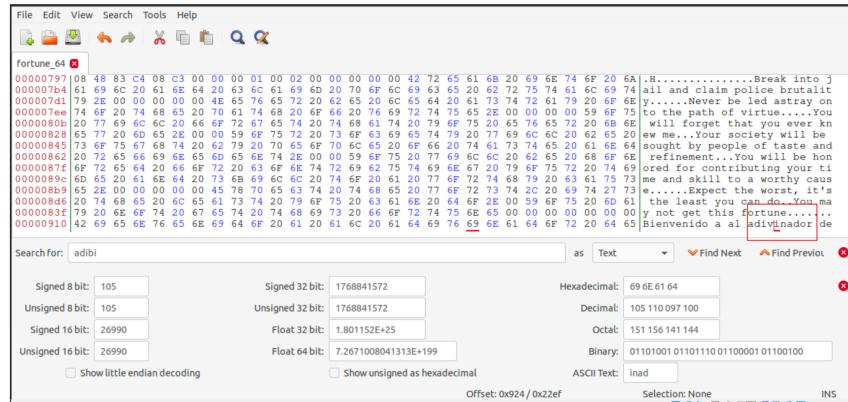
```
osboxes@osboxes:~/arqui/ArquiTP1$ ./fortune_64
Bienvenido a al adibinador de la fortuna! Este mensaje es muy largo, y puede ser
muy molesto al usuario. Tal vez deberíamos acortarlo?
Cual es tu nonbre?: Santiago

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

```
osboxes@osboxes:~/arqui/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 nonbre?:
Tu fortuna es:
;*3$"
```

Vemos todos los strings de los que se elige el de la “galleta de la fortuna”. Además, vemos el string con la consigna.

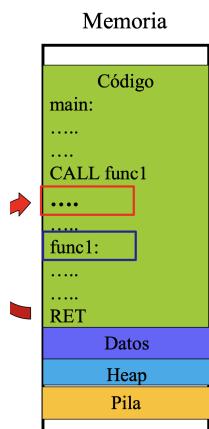
¿Qué pasa si al ejecutable lo abrimos con un editor de texto? Vemos todos caracteres en desorden, porque trata de interpretar todo lo que ve. Si lo abrimos con un editor hexadecimal (**bless**), vemos nuevamente los strings. Lo que está en rojo es la posición relativa dentro del archivo.



Cada código que no tiene representación ASCII, lo pone como un punto. Como es un editor, donde está “adibinador” mal escrito podemos corregirlo. Notemos que esto va a correr, porque cambiamos un byte por otro byte. Como era un ejecutable, tiene un checksum que verifica que no sea modificado. En este caso corre, pero también vamos a poder modificar el checksum.

Con esto, concluimos que podemos editar un ejecutable y cambiar datos. ¿Vamos a poder cambiar el código? Si, lo vamos a poder llevar a assembler y cambiar las instrucciones.

Llamadas a funciones (CALL)



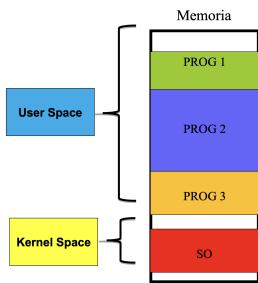
Permite saltar a otra posición de memoria y correr una función extra. Es una instrucción de Assembler. En C lo hacíamos con el nombre de la función.

Guarda en la pila (pushea) la dirección de retorno. La función debe terminar con la instrucción RET de Assembler, que es el equivalente al return de C. Lo que hace es “popear” la dirección de la pila y salta a esa posición.

En cualquier lado donde pongamos RET, va a tomar la primera dirección en la pila y salta a ella, por eso es fundamental manejar bien la dirección guardada en la pila. Vamos a tener que tener mucho cuidado, porque ahora la pila **la vamos a manejar nosotros**.

System Calls (syscall)

En vez de llamar a nuestras funciones, vamos a llamar al sistema operativo. Este es un programa que es siempre el primero que corre, solo antes de la BIOS, que es también un programa.

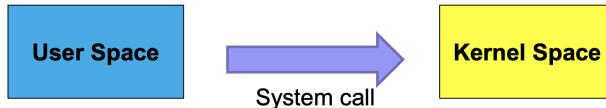


El kernel space es el lugar de memoria que toma el sistema operativo para ejecutarse. Además, los permisos son distintos.

Todo lo que ejecutamos como la barra de tareas, Chrome, etc. se almacena en el User Space.

Supongamos que varios programas quieren acceder al teclado. El sistema operativo pone a disposición un montón de funciones, como acceder al teclado justamente. Estos son los System Calls.

Gráficamente:



En linux, tenemos el strace que permite ver todos los system calls. El siguiente ejemplo es un código en Pampero que es únicamente un print de “hola mundo”.

- “\$ strace ./holamundo”

```

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_dev=makedev(136, 10), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7732000
write(1, "Hola Mundo", 10) = 10
Hola Mundo = ?
exit_group(0) = ?

```

Vemos casi al final el llamado a write con el Hola Mundo. Más arriba, también podemos ver un open con libc, que es la librería de c.

Algunos ejemplos de system calls en Linux (es anecdótico). Cada uno tiene un id que es único, y lo que está luego del Source son los argumentos que debe recibir. Los recibe por registros del procesador.

%eax	Name	Source	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	kernel/exit.c	int	-	-	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
3	sys_read	fs/read_write.c	unsigned int	char *	size_t	-	-
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t	-	-
5	sys_open	fs/open.c	const char *	int	int	-	-
6	sys_close	fs/open.c	unsigned int	-	-	-	-
7	sys_waitpid	kernel/exit.c	pid_t	unsigned int *	int	-	-
8	sys_creat	fs/open.c	const char *	int	-	-	-

Unidad 2 - Procesadores y Lenguaje ASM (Assembler)

Nosotros vamos a ver procesadores Intel, pero si cambiamos de procesador assembler va a ser distinto (no mucho).

Registro de un Procesador

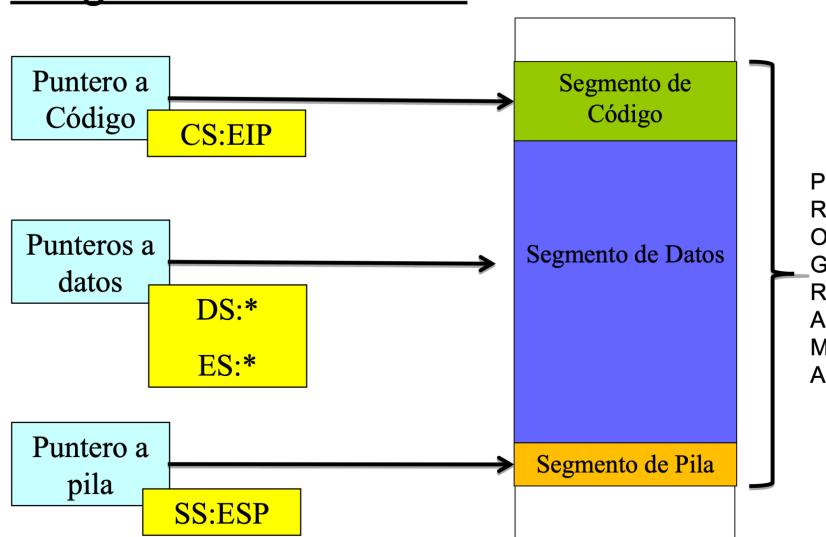
Los registros son pequeños almacenamientos de información que están dentro del procesador. No los guarda en la memoria porque los usa continuamente, y es más rápido que la memoria (que a su vez es más rápido que el disco).

Los registros son:

- **IP:** Instruction Pointer (EIP en 32 bits y RIP en 64 bits)
Puntero a las 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

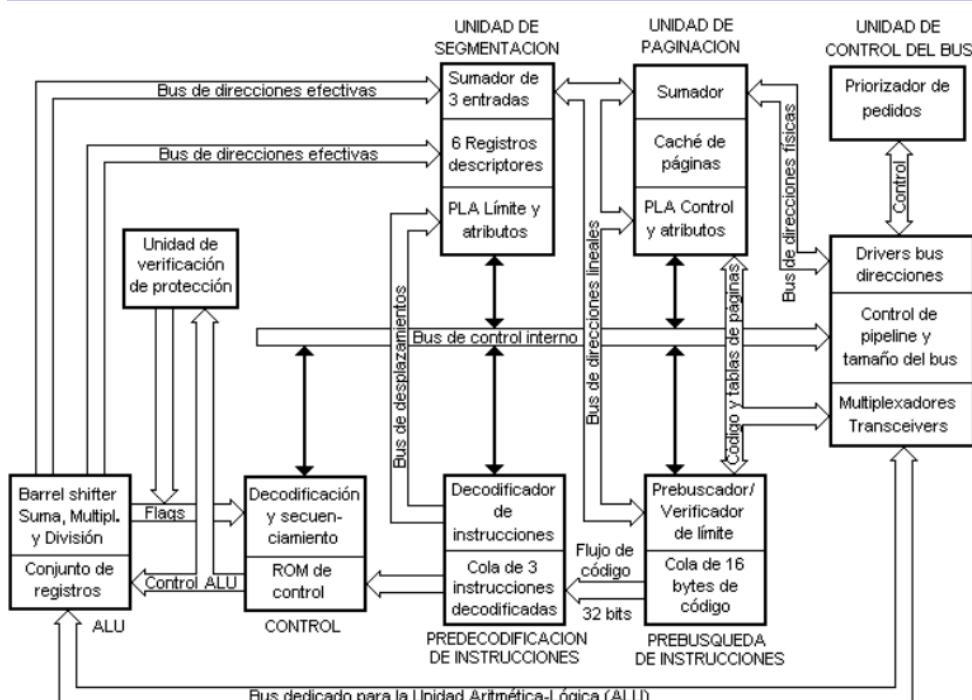


Intel trabaja con pares de registros. El CS apunta al inicio del segmento de código, y el IP es la posición relativa al inicio del segmento. Por ejemplo, necesito ir a Lavardén al 314. El CS es Lavardén 300 (inicio de la cuadra) y 14 “cuánto me tengo que mover”. Lo mismo con el SS y el ESP. La agrupación se hace para facilitar el acceso, como los capítulos de un libro. Entonces, el par es: comienzo - posición relativa. La manera de trabajar de a pares se llama **agrupamiento**.

Cuando buscamos en una carpeta de un disco rígido, es otra manera de agrupamiento en el disco. Otra manera de guardar es la dirección final - **direcciónamiento absoluto**.

El procesador nunca deja de ejecutar, nunca deja de funcionar. Si quisiéramos que pare o espere, hacemos básicamente que entre en un loop. Sólo deja de funcionar cuando lo apagamos. Se enciende cuando le pasamos electricidad.

El procesador que vamos a ver nosotros:



Los procesadores más 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 más 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. Cuando tenemos muchos programas abiertos al mismo tiempo, en realidad no corren al mismo tiempo. Corre un poco de uno, después del otro, etc., pero lo hace tan rápido que nosotros no nos damos cuenta.

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

Tiempo compartido: El S.O asigna un tiempo para cada tarea (time slot). Es análogo a multitarea.

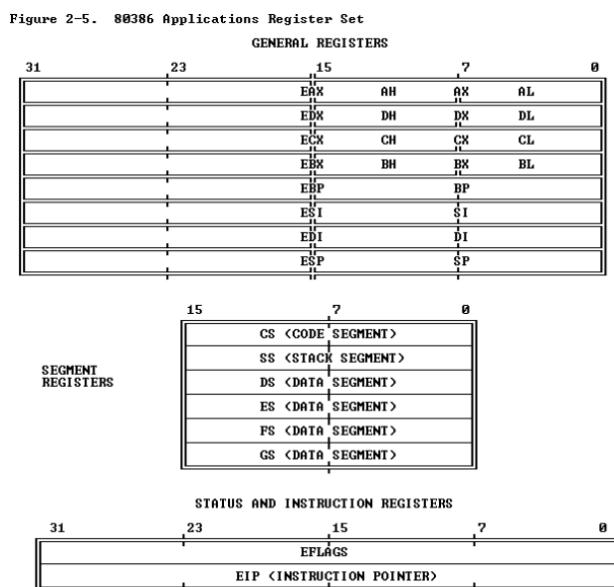
Tiempo Real: La conmutación de tareas viene dada por acontecimientos externos.

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.

El procesador es la caja de herramientas del sistema operativo. Nosotros vamos a ver todo lo que hace el procesador y no el sistema operativo.

Volviendo al caso de multitarea, cuando corremos un programa como Spotify, no corre el sistema operativo. Todo esto se rompe si tenemos un procesador multi core, pero dentro de cada uno pasa lo que ya vimos. Nosotros vamos a trabajar con un único procesador.

Registros (32 bits)



Vemos que tenemos dos “cajas” AH y AL unidos por AX. Esto lo hizo Intel para mantener la compatibilidad. Es como una unión, donde los dos primeros son registros de 8 bits y AX los une en uno sólo de 16 bits.

Lenguaje Assembler (Intel)

Assembler es el primer lenguaje de programación que tiene una relación unívoca con el código máquina. Por cada instrucción del código máquina que puede ejecutar el procesador, existe una instrucción de assembler.

Existen varias sintaxis para ASM, las más conocidas son:

- Intel
- AT & T

Veamos dos sentencias equivalentes con las dos sintaxis:

<i>mov EAX, 1</i>	<i>(sintaxis Intel)</i>
<i>movl \$1, %eax</i>	<i>(sintaxis AT&T)</i>

Esta instrucción convierte el registro de EAX en 1.

No vamos a usar la sintaxis de AT&T.

La siguiente instrucción agrega 1 a lo que haya en EAX y lo guarda en EAX. Lo vemos en código máquina y hexa.

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

Las instrucciones, cuanto más complejas son, más bits necesitan. Intel tiene instrucciones de tamaño variable, y las ARM tienen tamaño fijo, como por ejemplo ARM32 de 32 bits.

Ejemplos de uso con ASM:

```

    mov  ah, 23
    mov  bl, 99h
    mov  ax, 1234h
    mov  eax,12345678h
    mov  rax,12345678ABCDEF00h

```

La primera, como no tiene la h agrega 23 (en decimal). La tercera se realiza porque 1234h “entra” en la dirección que estamos poniendo (16 bits).

mov ax, [100h]	0100h	<table border="1"> <tr><td>D0</td><td>12</td><td>00</td><td>11</td></tr> <tr><td>00</td><td>3A</td><td>07</td><td>FF</td></tr> <tr><td>32</td><td>B8</td><td>C0</td><td>C1</td></tr> <tr><td>74</td><td>7E</td><td>E2</td><td>AE</td></tr> </table>	D0	12	00	11	00	3A	07	FF	32	B8	C0	C1	74	7E	E2	AE
D0	12	00	11															
00	3A	07	FF															
32	B8	C0	C1															
74	7E	E2	AE															
mov ebx,[102h]	0104h																	
mov cl,[109h]	0108h																	
mov ax,[bx]	010Ch																	

Los corchetes se utilizan para tomar el contenido de la dirección de memoria que está entre ellos.

La primera instrucción guarda D0 en ax. En ax tenemos 2 byte (16 bits) pero en 100h hay un solo byte (4 bit por cada carácter - como son dos caracteres D0, 8 bits). Lo que hace es tomar también lo que está en la dirección 0101h, es decir 12. Esto tiene una razón probabilística.

Cada posición de memoria tiene un byte, para mantener la compatibilidad.

Para guardar en memoria, invertimos el orden:

Mov [102h],eax	0100h	<table border="1"> <tr><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> </table>																
Mov [104h],bl	0104h	<table border="1"> <tr><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> </table>																
Mov [108], rcx	0108h	<table border="1"> <tr><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> </table>																

Tomemos el segundo caso. Sabemos que bl es un byte. La dirección 104h se va a guardar con el valor de bl. Como bl es un byte, tiene 8 bits, y por lo tanto puede

tomar valores de 0 a 255. Si por ejemplo bl tiene el valor 10, vamos a guardar 10 en la dirección 104h.

Ahora tomemos la primera instrucción. Como eax tiene 4 bytes y 102h tiene 1 de capacidad, en teoría no lo podría agregar, pero lo que hace es completar las direcciones “siguentes”. Es decir parte en 102h, parte en 103h, etc.

La tercera instrucción está tomando la dirección 108 en decimal, es decir en cualquier lado.

Modos de Direccionamiento

Como en otros procesadores tendremos la sintaxis general que será:

Instrucción destino , fuente

Direccionamiento inmediato:

```
mov ax, 0ffffh
```

Direccionamiento de registro:

```
mov edx,eax  
mov ah,al
```

Direccionamiento directo o absoluto

```
mov ax, [57D1h]  
mov ebx, es:[42c9h]
```

Direccionamiento indirecto

```
mov cx, [bp]  
mov es:[di],ax
```

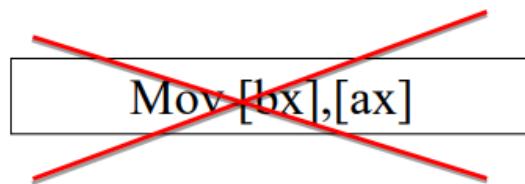
La primera instrucción va a la dirección de memoria del valor que hay en bp y lo copia en cx.

Direccionamiento con índice o indexado

```
mov cx, [bp+4]  
mov es:[di+8],ax
```

Supongamos que bp vale 6. Entonces se le agrega 4, por lo que dentro del corchete nos queda 10 (decimal porque el 4 es decimal). Entonces va a la dirección 10 de memoria y lo trata de guardar en cx.

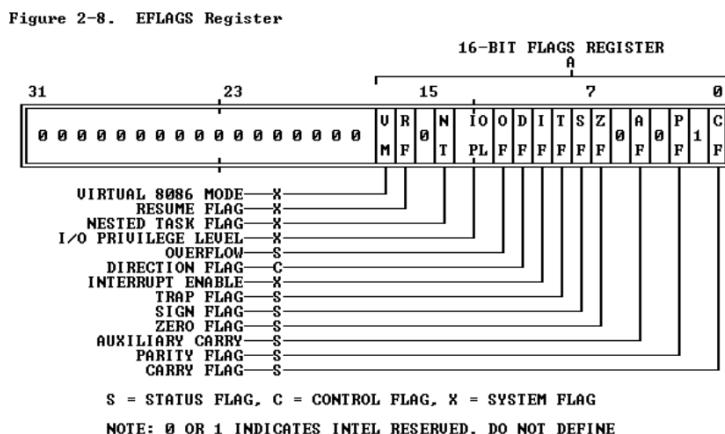
Recordatorio: NO existe el movimiento de datos de memoria a memoria en una sola instrucción:



Lo que tenemos que hacer es pasarlo por un registro.

Registro de Flags

Los flags son “banderitas” de valor 1 o 0 que se encuentran todas dentro de un registro (procesador).



Hay instrucciones que cambian los flags y otras que no. Por cada instrucción que corre, si alteró una flag la pone en 1 o 0. Los flags podemos usarlos, ignorarlos, etc.

Ejercicio 1

```

section .text
global _start

_start:

    mov    dx,0FFh
    mov    bx,20h
    add    dx,bx
    push   dx
    push   4
    pop    cx

    Ciclo:
    inc    bx
    dec    cx
    jnz    Ciclo

    mov    eax,parametros
    mov    AH,[parametros]
    mov    BL,[parametros+1]

```

add	ah,bl
mov	[salida],ah
ret 0	
section .data	
parametros	db 11h,12h,13h
salida	db 0

El assembler “puro y duro” empieza en el primer mov. Todo lo que está por arriba son directivas que pide el compilador con una cierta sintaxis.

section: definen dónde está el código.

global _start: estamos declarando start como una función global, es decir que se puede acceder desde afuera.

En ASM, un rótulo es lo mismo que una función. El código termina con **ret 0** que es lo mismo que **return**. Las funciones en ASM se declaran con los dos puntos, como **_start:**. ¿Cómo le mando parámetros? No está definido, no hay convención. Lo podemos trabajar como queramos. Es muy desorganizado pero más flexible. Lo podemos hacer por registro, por pila, etc.

La instrucción **push dx**, pone el valor de dx en la pila y lo mantiene a su vez.

La instrucción **pop cx**, toma el primer valor de la pila y lo guarda en cx.

Ciclo: es un rótulo, es decir un “puntero” a una parte del código.

inc incrementa

dec decrementa

jnz es un “jump not zero” que va a, en este caso, Ciclo. Vemos que cx quedó con valor 4, por lo que va a “saltar” hasta que se decremente a cero y se levante el flag.

En la zona de datos, tenemos db que es “define byte” que es tamaño de dato. Por ejemplo definimos “parámetros”, un tipo de dato que es de a bytes y 3 direcciones. Estas direcciones van a ser variables según cada vez que lo corramos, porque lo va a guardar donde hay lugar. Estamos declarando y a su vez inicializando. Es un arreglo cuyos valores son 11, 12 y 13 en hexadecimal.

Si queremos que tenga 2 bytes, usamos wb, etc.

Deabajo, definimos salida que tiene un byte (db) y vale 0.

mov eax,parametros guarda en eax la dirección de memoria de parametros.
mov AH,[parametros] lo que hace es guardar en AH el 11h, porque recordemos que parametros apunta al primer elemento.

En mov BL,[parametros + 1] va a guardar en BL el 12h.

Debugger

Programa que toma como entrada otro programa y lo corre línea por línea, parándolo a medida que avanzan las instrucciones.

Ejemplo:

→ 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 teej1!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

Vemos a la derecha el código en Assembler, en el medio la instrucción en código máquina y a la izquierda el CS:IP.

Vemos que las instrucciones son de distintos tamaños, y las direcciones en memoria van de acuerdo a dichos bytes.

Vemos luego:

Data Dump																
0x000000000000600000-0x00000000000601000																
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	00	06	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	00	03	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

En la izquierda tenemos el DS:*(data segment) y en el centro podemos ver 11, 12, 13 que declaramos antes.

Podemos ver que “parametros” vale 006000d8 y su contenido es 11. Si le sumamos 1, vamos a 006000d9, que es el 12. Vemos gráficamente lo que hicimos antes en el código.

Si hacemos parametros + 3, vamos a pisar salida que le asignamos 0. Tenemos que tener cuidado.

Ejercicio 2

```

section .text
GLOBAL _start

_start:
    mov ecx, cadena           ; Puntero a la cadena
    mov edx, longitud          ; Largo de la cadena
    mov ebx, 1                  ; FileDescriptor (STDOUT)
    mov eax, 4                  ; ID del Syscall WRITE
    int 80h                     ; Ejecucion de la llamada

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

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

section .bss
placeholder resb 10

```

Es básicamente un “hola mundo”. La section .bss es un espacio de datos a parte de los datos comunes. Se reserva memoria sin inicializar, a diferencia del section .data.

El int viene de interrupción, y lo que hace es ir a buscar la interrupción que queramos correr, en este caso 80h. Por ejemplo, nosotros queremos ver el hola mundo en la salida estándar (pantalla). Para eso, vamos a llamar a una función del SO que es el que “sabe” cómo imprimir en pantalla.

Entonces, las interrupciones son un vector con un montón de entradas. Si llamamos a tal lugar, leemos del teclado, si llamamos a tal otro, imprimimos en pantalla, etc.

Salida estándar: la estándar la podemos declarar como el monitor, el wifi, etc.; lo que nosotros queramos, pero generalmente es la pantalla. Cuando mandamos un mensaje en wpp, tiene dos salidas, a la pantalla para que lo veamos y al wifi para enviar el paquete de datos.

Tomando el ejercicio, vamos a cargar los argumentos en los registros:

move ecx, cadena → puntero a la cadena

move edx, longitud → largo de la cadena

move ebx, 1 → DONDE vamos a imprimir

move eax, 4 → QUE vamos a hacer (en este caso write)

Vemos en la data section que tenemos el valor de cadena y longitud.

Después se llama al exit con mov eax 1 y se asigna el valor de retorno 0 con el mov ebx, 0.

Linkedición con GCC

La función start se debe llamar main:

```
section .text
GLOBAL main

main:
    mov ecx, cadena      ; Puntero a la cadena
    mov edx, longitud     ; Largo de la cadena
    .......
```

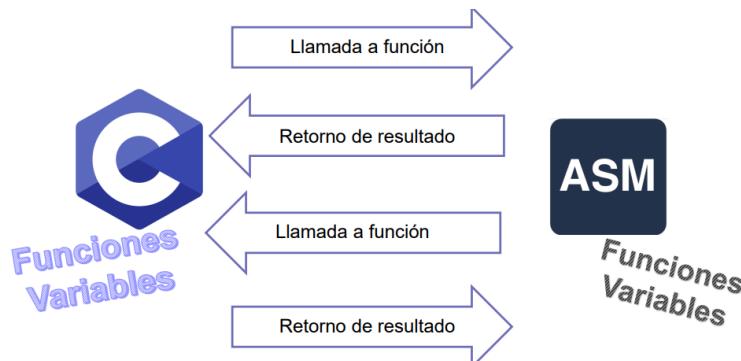
Para compilar: nasm -f elf64 teoej1forc.asm -o teoej1forc.o

Para linkeditar : gcc teoej1forc.o -o teoej1forc

Los archivos ejecutables tienen distintos tamaños. Esto se da porque el gcc agrega un montón de validaciones al comienzo, por eso vemos que los comienzos de ambos códigos son distintos.

Unidad 3 - Assembler y C

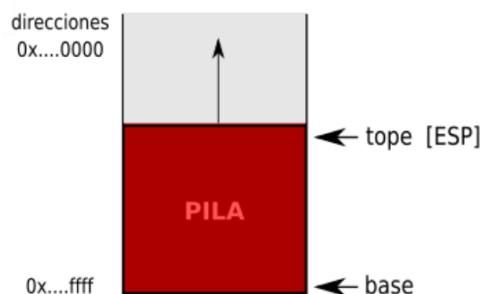
Mezcla de lenguajes en ejecutables



Los valores van a ser retornados en función a cómo fue predefinido para dicho lenguaje. Este problema se soluciona con la pila y con los registros.

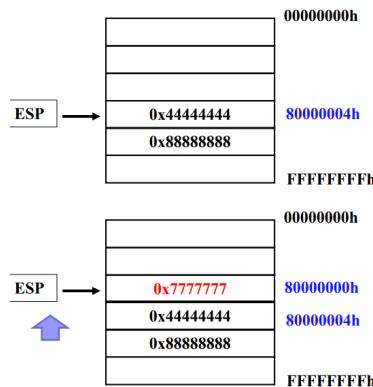
Vemos que C es mucho más “prolijo” que assembler, por lo que este se debería adaptar a él. Por eso vamos a usar las convenciones de C en ambos lenguajes.

Repaso de Pila

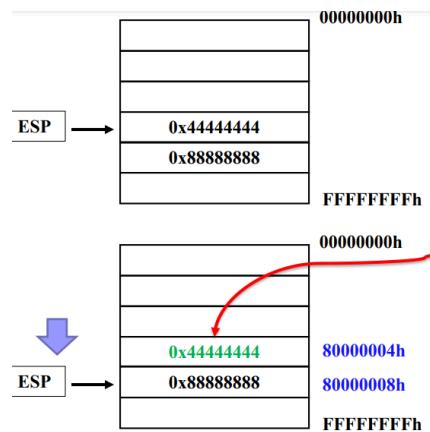


Vemos que la base de la pila está en las direcciones más grandes de memoria, y el tope en las más pequeñas. Las direcciones más bajas las vamos a poner arriba, y las altas abajo.

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.



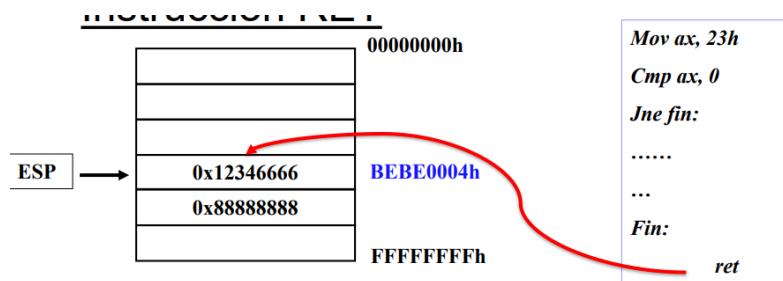
Cuando se ejecuta una instrucción *PUSH*, el procesador decrementa el registro ESP ó RSP y guarda el valor en el stack.



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. Vemos que es una pila LIFO (Last In First Out). ¿Qué pasa con el dato en verde? No se borra, queda libre para ser luego sobreescrito. Lo mismo pasa con los discos, y por eso tarda lo mismo en “borrarse” archivos de 1mb que otros de 1gb. En informática no existe el borrado. La única forma de borrar es sobreescribir.

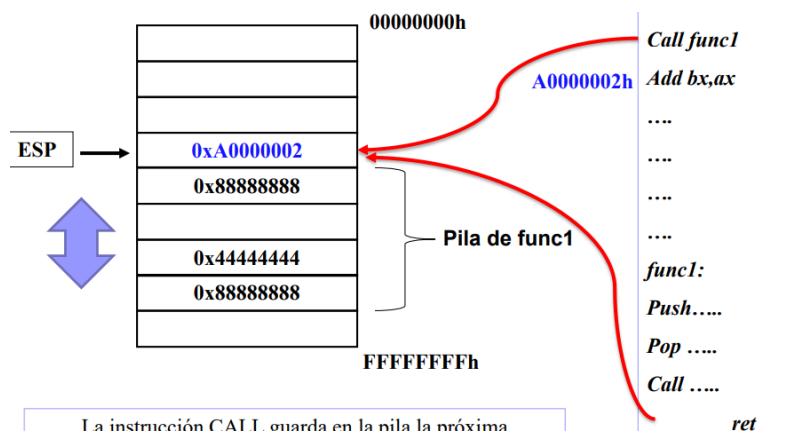
Instrucción RET

Ret va a la pila, agarra el valor que está en el stack pointer y salta a esa posición de memoria. Luego sigue ejecutando el código que haya en esa posición. Lo que hace es cambiar el valor del instruction pointer.



En este caso salta a la dirección de memoria 12346666h. Es equivalente a hacer un JMP a esa dirección. El return de c es el equivalente al ret de assembler.

Instrucción CALL



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. Tiene que dejar el stack pointer tal cual estaba.

Análisis de C

Vamos a entender cómo funciona el compilador de c para poder mezclarlo con ASM.

El pasaje de argumentos cambia según la arquitectura del compilador:

- Arquitectura de 32 bits: Se pasan por la pila
- Arquitectura de 64 bits: Se pasan primero por registros y luego por la pila

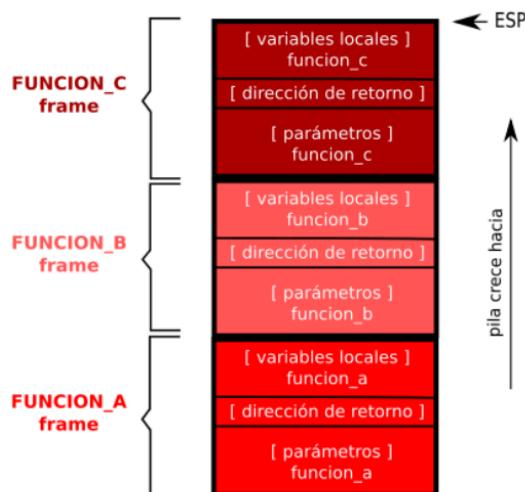
En la de 64 se cambió a registros porque es más seguro, ya que es más fácil modificar la pila y ejecutar código en ella.

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

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.

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. 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.

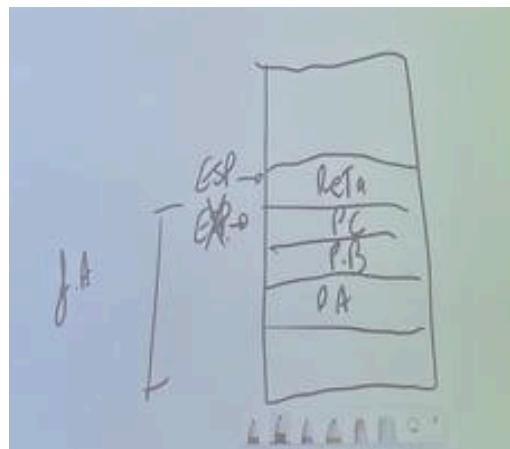


La función A primero pushea sus parámetros y cuando termina pushea la dirección de retorno (a donde pasa a apuntar ESP). Luego, si se llama a la función B, esta hace lo mismo.

Como vemos que están contiguas las pilas, la función B podría hacer muchos pop y modificar la pila de A. Por esto, lo que hace el compilador de C es backupear la dirección de retorno en **EBP**. Básicamente lo que hace es:

```
mov EBP, ESP
```

Todas las funciones que se llaman backapean el ESP. Esto nos salva el retorno, pero si hacemos muchos pop como dijimos antes programando mal, voy a modificar los parámetros.



El EBP se pushea a la pila para luego guardarlo (armado de stack frame). Para recuperarlo, se mueve ESP al EBP, se lo popea de la pila y finalmente se retorna (desarmado de stack frame). Para cada variable que declaramos en C, el compilador guarda espacio “arriba” de la dirección de retorno.

Convenciones en C

Parámetros de una función: Por ejemplo, en el caso de `funcion_en_C` (`param_a`, `param_b`, `param_c`), 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.

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	Push ebp Mov ebp,esp	Desarmado de stack frame	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.
- Si es un dato más complejo (ej. Estructura de datos) retorna un puntero formado por EDX:EAX

Ninguna función que llama a otra debería dejar algo que le interese en eax.

Llamada de ASM a C

<pre>; Puts.asm ; Programa que imprime utilizando puts global main extern puts section .data mensaje db 'Utilizando puts', 0Ah, 0 section .text</pre>	<pre>main: push ebp frame mov ebp,esp ; genera stack push dword mensaje ; parametro para puts call puts ; llamada pop eax ; saca el parametro de la pila mov esp,ebp frame pop ebp ; destruye stack ret</pre>
---	---

Esto no lo vamos a usar mucho, que es llamar de ASM a c. Nosotros vamos a hacer que nuestro main esté en el lenguaje de más alto nivel.

Para llamar desde Assembler, declaramos la función como externa, para que se encargue el linkeditor. Las primeras dos líneas del código ASM generan el stackframe. Luego, pusheamos el mensaje, es decir que hacemos el pasaje de parámetros por stack. Luego llamamos a la función para que imprima, y hacemos un pop eax para sacar el parámetro que le pasamos a la pila. Si no hicieramos esto y entramos en un ciclo, la pila se va a agrandar cada vez más, por eso tenemos que poppear el parámetro. Esto se da porque la convención es que las funciones *dejan la pila como la encuentran*. Otra manera de “limpiar” la pila sería mover el stack pointer en vez del pop.

Los archivos cuando se compilan, son recorridos una sola vez (por ej. por nasm). Tomemos el caso en que leemos un texto. Con la primera leída no podemos resumir porque no sabemos cuáles van a ser highlights. Los compiladores hacen algo parecido. Como leen una única vez, cuando llamamos a puts, la resuelven directamente. No la resuelven en una “segunda pasada”.

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;
}
```

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

Estas son las que más vamos a usar, porque con assembler vamos a poder hacer cosas propias de Intel que C no puede. Ya vimos que C no fue desarrollado para ningún SO, es de alto nivel.

Así, podemos hacer todo nuestro código en C, y cuando necesitamos hacer algo de muy bajo nivel que C no sabe, creamos la función en ASM y la llamamos desde C. Por ejemplo si necesitamos escanear el ingreso de una palabra desde el teclado. En PI usábamos scanf, que llama al SO que tiene las instrucciones de bajo nivel que permiten hacer esto.

Vemos que la función siete la declaramos global para poder “encontrarla” desde afuera. En C la declaramos como externa al igual que antes en ASM.

Sabemos que eax se utiliza para retornar valores (entre otras cosas) por las convenciones que vimos. Entonces, la función guarda el 7 en el registro eax para así ser tomado por C.

Notemos que la declaramos como unsigned int la valor de retorno de la función en C. El que se debe encargar de que efectivamente se devuelva esto, es el que hace la función (nosotros en ASM).

Inline Assembler

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

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

NO lo hacemos. Es escribir una línea assembler dentro de C con sintaxis AT&T.

Salidas en ASM

Tenemos dos formas de ver el código C convertido en ASM

- Compilar con “-S”. Lo que hace es compilar el código en C pero lo deja en ASM, no llega a código máquina.
- Utilizar GDB

Veamos un ejemplo:

```
/* asmyc1.c */
#include <stdio.h>
#include <stdlib.h>

int main ()
{
    int numero=10;
    printf("Numero vale = %d", numero );
    exit(0);
}
```

```
gcc -S -masm=intel asmyc1.c
```

```

.file  "asmyle1.c"
.section .rodata
.LC0:
.string "Numero vale = %d"
.text
.align 2
.globl main
.type  main,@function
main:
    push  ebp
    mov   ebp,esp
    sub   esp,8
    and   esp, -16
    mov   eax,0
    sub   esp,eax
    mov   [ebp-4], 10
    sub   esp,8
    push  [ebp-4]
    push  .LC0
    call  printf
    add   esp,16
    sub   esp,12
    push  0
    call  exit
.Lfe1:
.size  main,.Lfe1-main
.ident "GCC: (GNU) 3.2 20020903 "

```

El `-masm=intel` es para que la sintaxis de ASM sea la de Intel y no la de AT&T.

Vemos que el main arranca con el armado del stack frame. El string se guardó en `.rodata` (`ro = read-only`) porque no es código y además es constante, por eso `ro`. Cuando comienza a ejecutarse el main, en la pila tenemos el ret del main (`retso - so = sistema operativo`). Todo lo que está debajo en la pila corresponde a la pila del SO. Luego, pusheamos `ebp` y el `esp` (stack pointer) pasa a apuntar al valor del `ebp` en la pila. Así, el `ebp` pasa a igualarse con el `esp`. El `ebp` que estamos backueado en la pila en este momento, se corresponde con el `ebp` del SO, porque tenía un valor y nosotros lo vamos a usar (y queremos que no se pierda).

Luego, le restamos 8 al `esp`, por lo que “sube” 2 en la pila (cada dirección en la pila tiene 4 bytes). El `ebp` del main se mantiene apuntando a donde estaba antes. `mov[ebp-4], 10` va a guardar un 10 en la posición de “arriba suyo”. Esto es la **asignación** del valor 10 a la variable número. Es, en C, `int numero = 10`. Vemos con esto que cuando se movió el `esp` antes, lo que hizo fue *guardar lugar*. Con esto, vemos que una variable local de C se guarda en la pila.

Ahora, `sub esp, 8` corre nuevamente el `esp` dos bytes para arriba, como hizo antes. Luego se pushea el 10 que guardamos antes en la pila.

En `push Lc0`, como este es un rótulo, lo que está haciendo es pushear la dirección de memoria donde se encuentra el string “Número vale ...”. NO el string, **pushea la dirección de memoria (MUY IMPORTANTE)**. Notemos que estos son los parámetros para el `printf`; para el “10”, recibió un **VALOR**, no un puntero, pero para el string recibió el **PUNTERO**.

Luego, se llama al printf y se pushea la dirección de retorno en la pila. La dirección de retorno que se pushea es la del main, porque tiene que volver al main y seguir por la instrucción "add esp,16 ...". Entonces, todo lo que quede por "arriba" en la pila va a ser para lo que haga ret.

Una vez que vuelve de printf, suma 16 al esp, por lo que este va para abajo (4 "lugares"). Esto lo que hace es liberar la pila (en lugar de hacer un pop). Vemos que luego vuelve a subir 12, y estas dos instrucciones se pueden hacer con add esp,4. Esto no es óptimo, pero tiene que ver con cómo funciona el compilador (no es perfecto).

Por último, se pushea un 0 y se llama a exit, que se corresponde con el exit(0). Notemos que a lo largo del código, no hizo nada con los #include, porque eso es información para el linkeditor. Con el "int main" no hizo nada pero porque estamos terminando el programa con un exit, no con return.

Si ahora tenemos una estructura de 64 bits, los parámetros no se van a poder pasar por la pila. El código recompilado nos quedaría:

<pre> .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 </pre>	<pre> 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 call exit .LFE2: .size main, .-main .ident "GCC: (Ubuntu 4.8.4)" .section .note.GNU-stack,"",@progbits </pre>
---	---

No tenemos que acordarnos de todos los registros, pero tenemos que saber que como la estructura es de 64, los parámetros se pasan por registros y no por la pila.

Análisis de Manejo de Pila

Tomemos el siguiente programa

```
//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;
}
```

Analizamos la salida con un debugger (gdb):

```
(gdb) set disassembly-flavor intel
(gdb) disasemble main
Dump of assembler code for function main:
0x080483e9 <+0>: push  ebp
0x080483ea <+1>: mov   ebp,esp
0x080483ec <+3>: sub   esp,0x8
0x080483ef <+6>: mov   DWORD PTR [esp+0x4],0x4
0x080483f7 <+14>: mov   DWORD PTR [esp],0x3
0x080483fe <+21>: call  0x080483dc <suma>
0x08048403 <+26>: mov   eax,0x0
0x08048408 <+31>: leave 
0x08048409 <+32>: ret
End of assembler dump.

(gdb) disasemble suma
Dump of assembler code for function suma:
0x080483dc <+0>: push  ebp
0x080483dd <+1>: mov   ebp,esp
0x080483df <+3>: mov   eax,DWORD PTR [ebp+0xc]
0x080483e2 <+6>: mov   edx,DWORD PTR [ebp+0x8]
0x080483e5 <+9>: add   eax,edx
0x080483e7 <+11>: pop   ebp
0x080483e8 <+12>: ret
End of assembler dump.

(gdb)
```

Vemos que tenemos los códigos asm de main y la función suma. Nuevamente, lo primero que vamos a tener en la pila es el ret al SO, con el esp apuntando allí. Armamos el stack frame como ya hicimos antes (mov ebp, esp).

sub esp,0x8 va a restar 8 reservando espacio (2 para arriba) como vimos en el caso anterior. Vemos que main no tiene variables locales, pero igual se mueve.

mov DWORD PTR[esp+0x4],0x4 lo que hace es guardar un 4 en la posición de arriba del ebp.

mov DWORD PTR[esp],0x3 guarda en donde está apuntando esp (arriba del 4) un 3. Luego se llama a la función suma (con la dirección donde se encuentra el rótulo), y se guarda el ret al main (que es la dirección 0x08048403, es decir de la siguiente instrucción al call). Lo primero que hace la suma es armar el stack frame para ella.

Pushea el ebp de main, porque es el que va a usar (para ahora la suma), y los iguala como antes. Entonces ahora tenemos el ebp de suma.

Vemos que guarda en eax el 4 y en edx el 3 para hacer la suma (los toma de sus posiciones en el stack). Luego, por convención el resultado se guarda en eax. Notemos que no se movió el stack pointer, y usó el ebp para buscar los argumentos. Por último, la función suma realiza el desarmado de stack frame con una sola instrucción porque el esp no se movió. En este caso fue eficiente. Luego llama al ret. Finalmente se guarda 0 en eax porque tenemos un return 0 en c.

¿Qué pasa si en el main agregamos la siguiente variable local?

```
int main(void)
{
    int resul;
    resul=suma(3,4);
    return 0;
}
```

Lo recompilamos y vemos que:

```
Dump of assembler code for function main:
0x080483e9 <+0>: push  ebp
0x080483ea <+1>: mov   ebp,esp
0x080483ec <+3>: sub   esp,0x18
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
0x08048406 <+29>: mov   eax,0x0
0x0804840b <+34>: leave 
0x0804840c <+35>: ret
End of assembler dump.
```

La función suma es exactamente igual. Vemos que el código comienza exactamente igual, pero ahora el sub esp, 0x18 (que está en hexa) va a guardar más lugar. Son “6 renglones” para arriba en la pila. Luego, guarda el 3 y 4 la posición esp del stack y por debajo. Luego procede exactamente igual. Cuando “vuelve” de suma, guarda el resultado de la suma (que está en eax) en un renglón arriba de donde está el ebp (que quedó abajo en la pila, por encima del ret al SO. Todo el lugar que queda el medio en la pila es espacio de más. Esto nuevamente es porque el compilador no es perfecto. Lo que hace leave es desarmar el stack frame (es una macro).

Otro caso:

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    int pass = 0;
    char buff[15];

    printf("\n Enter the password : \n");
    gets(buff);

    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;
}
```

Notemos que podemos romperlo si le pasamos algo de mas de 15 chars, que es lo que le asignamos a buff:

```
[svalles@pampero teoria]$ ./detalle3  
  
Enter the password :  
thesecretpass  
  
Wrong Password  
[svalles@pampero teoria]$
```

Vemos que nos tira el Wrong Password y después stack smashing detected. Lo que sucedió es que nos pasamos de lo que reservamos para la contraseña. No hay una función que detecte que me pasé del buff. Acá, el gcc incluyó funciones que se aseguran de que no rompamos la pila. Esto es porque debajo del lugar de reserva, está la dirección de retorno al SO. Si nos pasamos con los caracteres, pisaríamos eso y explotaría todo. El gcc agrega esas funciones para proteger esto.

Lo que hace, es que cuando queremos pisar ese valor, se reemplaza lo que queremos poner por una dirección a otra función para “salvarla”. Esto se llama **inyección de código por pila**. La principal razón por la que esto podría suceder es la falta de validación de datos.

Si en vez de poner un “aaaaaaaaajjj” ponemos código seguido por la dirección al comienzo del mismo, vamos a poder ejecutar ese mismo código (inyectamos código por la pila).

Hay una forma de compilar sin proteger al stack:

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

Si lo compilamos con esto, vemos que:

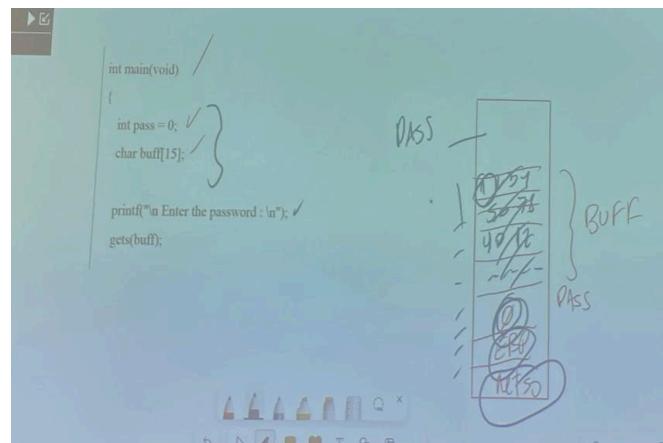
```
[svalles@pampero teoria]$ ./detalle3sinprotect
Enter the password :
123456789012345

Wrong Password
[svalles@pampero teoria]$ ./detalle3sinprotect
Enter the password :
1234567890123451

Wrong Password

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

La primera es de 15, por lo que la toma como wrong. La segunda, de 16 ocupa los 15 que reservamos y una más que es el pass en el stack, y la modifica a 1. Por eso se imprime el string, porque pass queda en 1. Esto sucede únicamente por cómo están declarados buff y pass, si los declarábamos al revés no sucedía.

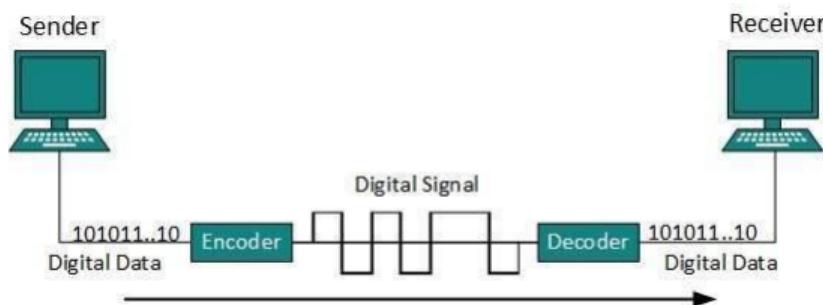


Lo que hace el gcc es inyectar arriba del ebp un número random (llamado canary) y justo antes del ret llama a una función __stack_chk_fail. Esto lo hace siempre antes de que llamemos a un ret. Compara los valores del canary y, si no coinciden, no salta. No corre dicho ret. Si queremos compilar un código sin canary, debemos usar el flag -fno-stack-protector (está más arriba).

(Tomado en parciales el canary, como pregunta teórica y ejercicios prácticos).

Unidad 4 - Memoria y Entrada y Salida

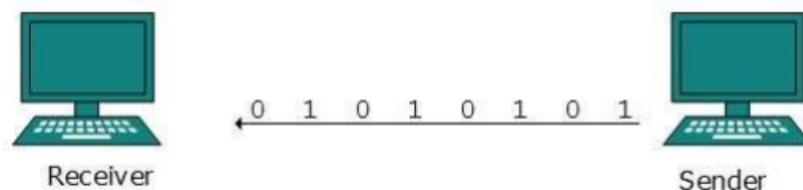
Transmisión Digital



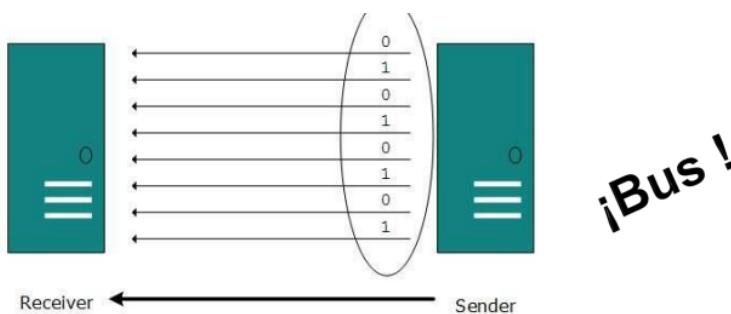
Dentro de la compu, en la motherboard tenemos un montón de componentes, y todos tienen que “hablar” con el procesador. Esto lo hacen mandando unos y ceros por cables de cobre (en realidad láminas) que se llaman pistas. Los ceros y unos se tienen que representar de alguna manera, como por ejemplo (el cero 5V y el uno 2V). Son dos representaciones mecánicas o físicas que luego se le asigna 1 o 0.

Hay dos formas de conectar el procesador con los periféricos:

- Serie:



- Paralelo:

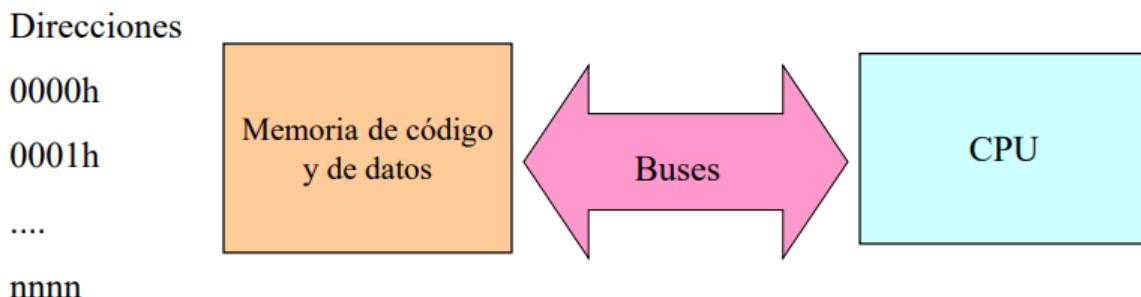


Vemos que son 8 cables, lo que representa un byte. Es más rápido porque en un instante de tiempo llega un sólo uno. El problema es que va a ser más caro. Por ejemplo en el internet de casa, tenemos en serie porque sería muy caro tirar ocho cables. En el caso del mother, vamos a tener en paralelo. El bus es toda esa información que sale en un mismo momento.

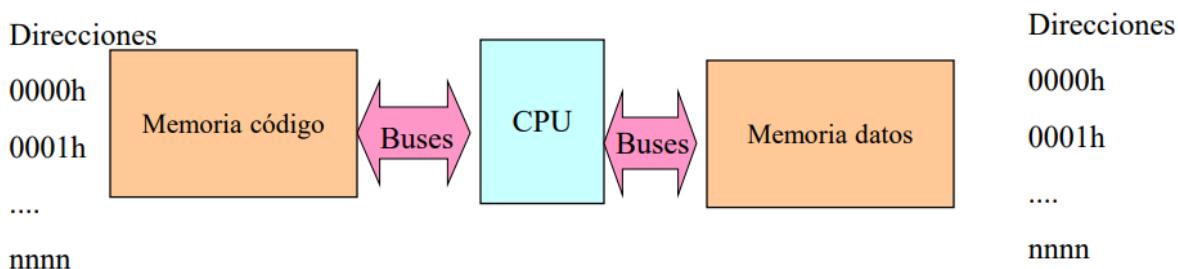
Tipos de Arquitectura

Cuando ejecutamos un código, necesitamos acceder al código y a los datos. Hubo dos filosofías:

Von Neumann: Pensó en un solo bus, es decir que todo pase por el mismo. El problema de este bus compartido, es que hay que esperar que uno lo libere para que otro lo pueda usar.

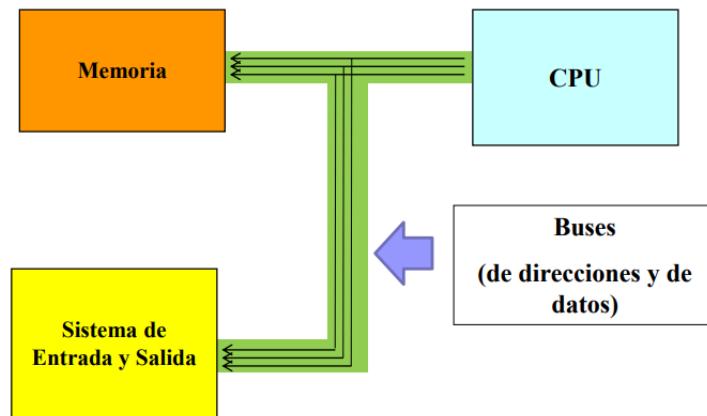


Harvard: Pensó en el doble bus, es decir que tenga memoria de código por un lado y memoria de datos por el otro, para no tener que utilizar un mismo bus.



Si bien la de Harvard parece mejor, nuestras computadoras usan Von Neumann, porque tendríamos dos tipos de memoria, y se complica predecir cuánta memoria va a ser para código y cuánta para datos. Si por ejemplo nos quedamos sin memoria "de código", es mucho más complicado cambiarlo. Los celulares con procesadores ARM vienen con ambas.

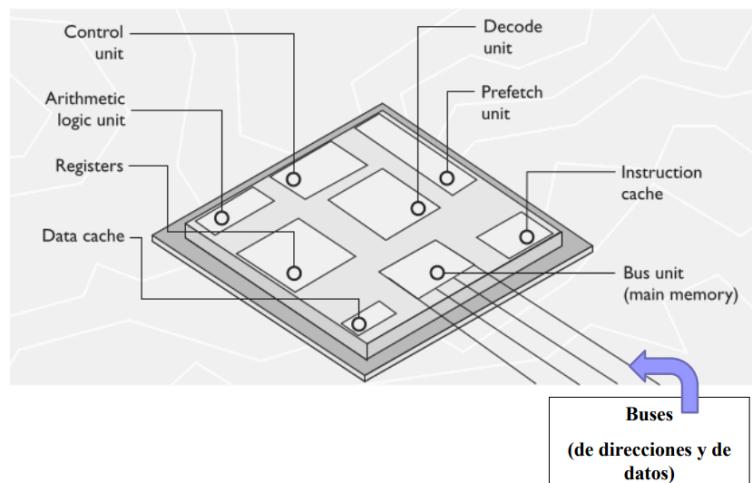
Sistema de Entrada y Salida



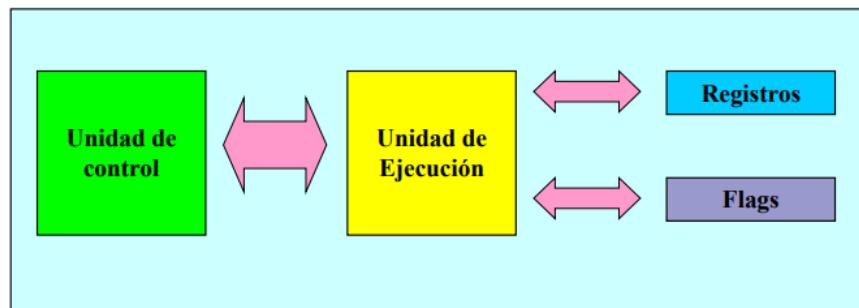
Lo que está en verde es el bus. El sistema de entrada y salida es todo lo que usamos nosotros (teclado, monitor, placa de wifi, etc.). Vemos que no hay un bus por cada periférico. Hay un único bus donde todos ellos se conectan. Por ejemplo, en el aula, si Santiago es el procesador y nosotros los periféricos, el aire es el bus. Estamos todos “conectados” al aire, no hay uno para cada uno. Cuando habla (manda datos), todos los recibimos.

Pero. ¿cómo sabemos a quién le está hablando? En el caso del aula, cuando Santiago le quiere hablar a alguien, dice en un paso previo el nombre y luego envía el dato, pero todos lo escuchamos. El bus que vemos se divide en 3, dos de ellos son bus de direcciones (salen los unos y ceros que van a apuntar a los periféricos, sale el puntero) y el bus de datos (circula la información que va de periférico a periférico). Es decir, un canal para apuntar, y otro para pasar los datos.

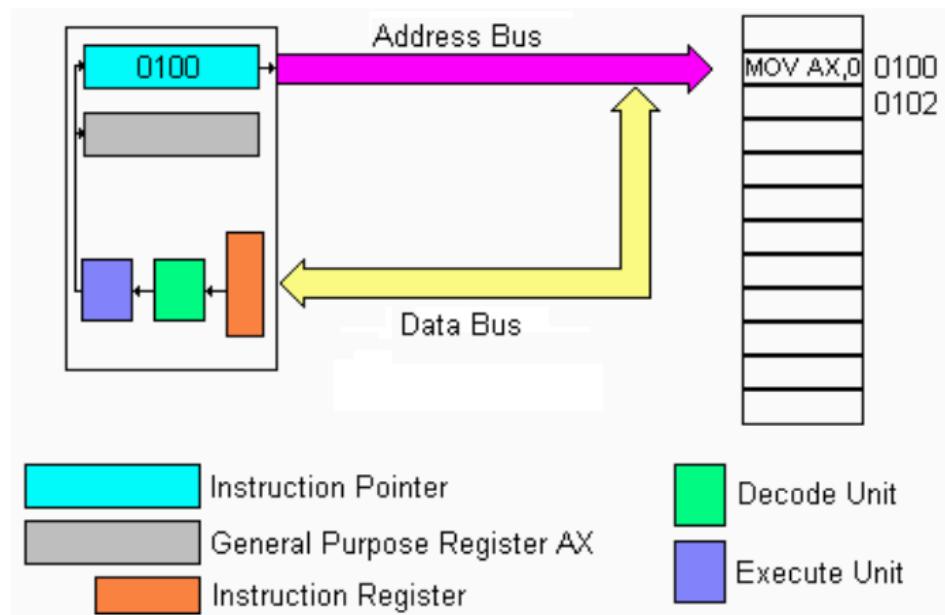
CPU



La unidad de bus es desde donde salen los “cablecitos”. Recupera las instrucciones de memoria, las decodifica y las escribe en memoria.



Luego, la unidad de ejecución lleva a cabo la ejecución de la instrucción. Los registros vimos que son memoria interna utilizada como variable. Los flags indican eventos luego de ejecutar las instrucciones.



Lo que está en color cian es el instruction pointer. Vemos que toma la dirección 100 en hexa. Cuando toma un valor, el 100 en hexa se transforma en binario y cada 1 o 0 sale por el address bus. Como hay 4 dígitos hexa y cada uno ocupa 4 bits, la dirección tiene 16 bits. Entonces, el address bus va a tener 16 “cables”. Como lo que llega por el bus se guarda en los registros, la cantidad de “cablecitos” del bus depende de cuánto sean los registros. Por ejemplo, hoy en día, como los procesadores tienen 64 bits, los buses van a ser también de 64 bits.

Por lo tanto, lo que vemos en la imagen es un procesador de 16 bits, que tiene un bus de también 16 bits. Como el bus de direcciones, le estoy indicamos a la memoria con “quién queremos interactuar”. Vemos que en la posición de memoria

0100 hay una instrucción, que va a viajar por el bus ahora de datos. **IMPORTANTE:** vemos que el bus de direcciones es unidireccional, y el de datos bidireccional. Por eso, si bien se envía la dirección 0100 por el bus de direcciones, la instrucción viene por el de datos. Esto sigue siendo Von Neumann porque hay una sola memoria. La información del 0100 le va a llegar a todos los periféricos, porque recordemos que los buses son compartidos. Sólo el periférico que esté ubicado en la posición 0100 debería responder.

Volviendo a la analogía del aula, es como si se dijera el legajo. El que lo tiene, es el que sabe que le están hablando a él. Pero esto se puede sólo si todos tienen legajos distintos, por lo que en la computadora *todos los periféricos deben tener direcciones distintas*. Si dos tuvieran la misma dirección, cuando “responden” habría colisión de la información.

Una vez que llega la instrucción, se guarda en el registro de instrucción, y luego actúan el Decode Unit y el Execute Unit que los vamos a ver más adelante. Una vez que se ejecutó internamente la instrucción, se guarda el 0 en AX (que lo vemos como gris). La siguiente está en 0102 porque la instrucción “mov AX, 0” ocupó 2 bites.

Pero cada periférico no va a tener una única dirección (como un legajo), va a tener un rango. Por ejemplo, la RAM tiene muchísimas direcciones de memoria comparado con un mouse o teclado. Un disco rígido de un tera, no va a tener un tera de direcciones. Puede ocurrir que la cantidad de direcciones de un periférico sea mayor a la cantidad de direcciones que un CPU puede generar.

Recordar:

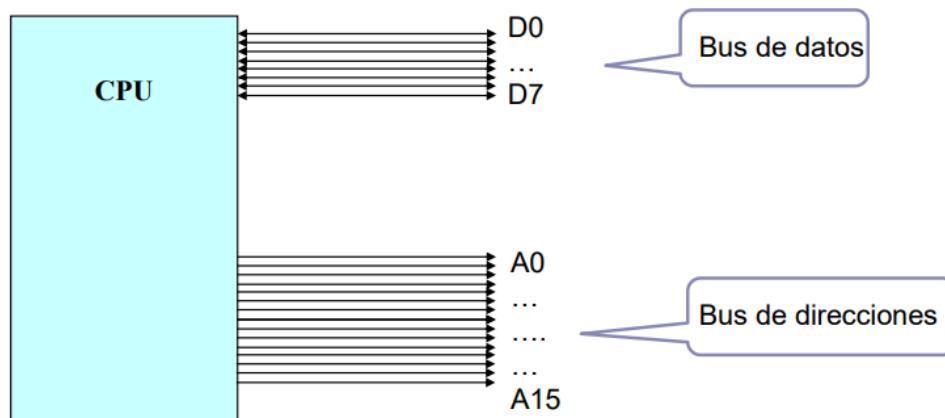
- 2^{10} = K (Kilo)
- 2^{20} = M (Mega)
- 2^{30} = G (Giga)
- 2^{40} = T (Tera)
- 2^{50} = P (Peta)
- 2^{60} = E (Exa)

Hoy tenemos procesadores de 64 bits $\rightarrow 2^{64} = 2^4 * 2^{60} = 16E$. Entonces, hoy en día tenemos procesadores que pueden apuntar a 16 Exa direcciones. Igualmente, lo que

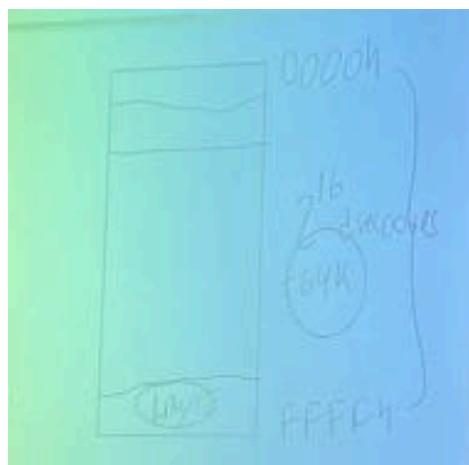
hacen los discos es tener algunas direcciones y proporcionar los archivos. No accedemos a CADA bit del disco.

Mapa de Memoria

Todas las direcciones de memoria que puede ver un procesador.



Vemos en este caso que tenemos 8 bits para el bus de datos y 16 para el de direcciones. Entonces la máxima dirección que podemos representar es 0000h - ffffh (en hexa). Entre ellos, hay 2^{16} direcciones = 64K. En el mapa de memoria hay un byte por cada dirección. Como hay 64K, vamos a tener 64Kb direcciones. Como el bus de datos tiene 8 cables, cada “bloque” de memoria va a tener 1 byte.



Estas direcciones las podemos dividir, por ejemplo, en 32KB para la RAM, 16KB para la ROM, etc. Ahora podemos ver otros casos con distintas combinaciones de cantidades de cables. Ejemplo, si tenemos 32 líneas de datos y 32 líneas de direcciones. Las direcciones van a poder ser: 0000 0000h - ffff ffffh. Cada “bloque” de memoria va a ser de 4 bytes porque tiene 32 líneas de datos (32 bits). Dividimos 32 por 8 y nos da los 4 bytes.

Registros de Intel

Para decidir cómo ubicar los periféricos en las direcciones de memoria, lo primero que tenemos que pensar es cuál es la primera instrucción que ejecuta el micro al encenderse.

Cuando prendemos la computadora, el instruction pointer apunta a la BIOS (ROM). Los 32Gb de RAM pierden la información cuando no recibe energía, mientras que la ROM no. Entonces, cuando armemos nuestra computadora, el instruction pointer tiene que apuntar siempre a la ROM. No a un disco rígido por ejemplo, porque es más lento que la ROM. La dirección a la ROM NO es 00..0.

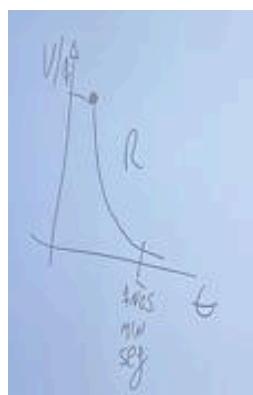
Por ejemplo, en un proyector, tenemos que elegir nuestro procesador y leer el manual del fabricante para ver a qué dirección apunta el Instruction Pointer para guardar ahí las instrucciones de inicio.

Memorias

ROM (Read Only Memory) - su nombre ya quedó viejo.

- PROM (Programmable ROM)
- EPROM (Erasable PROM)
- Flash, EEPROM () - son las que tenemos hoy en día, son las de los pendrives.

Mantienen su información sin energía (no volátil). La información en un pendrive o ROM va a desaparecer luego de un tiempo de no recibir energía, y tiene que ver con la curva entre el tiempo y el voltaje o corriente.



Generalmente está en el orden de los 10 a 40 años. El pendrive va a funcionar si le volvemos a dar corriente, pero no van a estar los datos.

Refresco de memoria: cada dato puede ser un 0 o un 1. Antes de que se pierda la energía, deberíamos “inyectarle” lo necesario para volver al nivel de antes (por ej si 5V es un 1, tiene que volver a esos 5V, porque el número va bajando). Lo que

hacemos en el refresco es leerlo, para saber que era, y volver a su posición. Hay un umbral en el cuál ya no se puede distinguir qué es. Esta leída y escritura no se hace siempre porque tomaría muchos recursos, se hace manualmente. Además, los pendrives son muy dinámicos, se escriben y sobreescriben muchas veces.

La ROM está en orden de años en la pérdida de energía, mientras que la RAM está en el orden de los milisegundos. Por esto es volátil. La escritura en RAM tarda nanosegundos, mientras que en la ROM milisegundos.

La BIOS lo primero que hace cuando arranca la compu es un testeo del procesador con operaciones aritméticas para asegurarse de que funciona bien. Luego prueba la RAM escribiendo y leyendo un valor. Todo esto hace la ROM.

RAM (Random Access Memory):

DRAM: Dinamic Ram - es la que tenemos en la computadora.

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

SRAM: Static RAM - la tenemos en la memoria caché del procesador

- No necesita refresco.
- Más compleja, más costosa.
- Más rápida
- Se suele utilizar para memoria caché.

Tiempo de Acceso

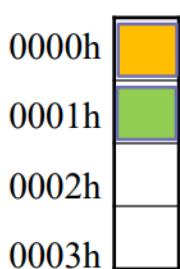
Transferencia: tiempo que tarda un bit en viajar desde el procesador a la memoria.

La tasa es casi la misma en todas las computadoras porque depende del cobre.

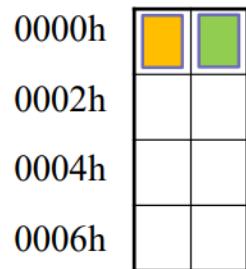
Latencia: tiempo que tarda la memoria en devolver el valor. Es lo que determina la velocidad. Esto quiere decir que una memoria sea más rápida que la otra, qué tan rápido **devuelve**. Las DRAM suelen tener tiempos entre 50 y 150 ns. Las SRAM menores a 10 ns.

Memorias - Estructura

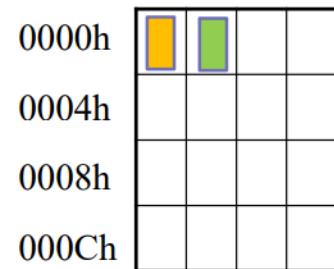
Intel decidió que por más que el bus de datos crezca, en cada posición de memoria de TODOS sus procesadores hubiera un byte. Por lo tanto la decodificación cambia según el tipo de memoria.



Bus de 8 líneas



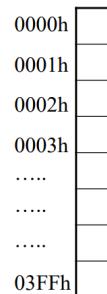
Bus de 16 líneas



Bus de 32 líneas

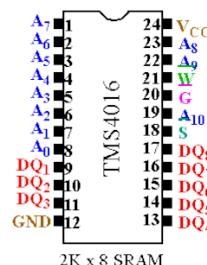
Vemos en el bus de 16 líneas, que ahora traemos dos bytes al mismo tiempo, pero no hay en cada posición de memoria 2 bytes. La misma lógica para el de 32 líneas. Esto lo hicieron por retrocompatibilidad.

Cuando hagamos un mapa de memoria de intel nos va a quedar (en 32 bits):



Donde cada “cuadradito” es un byte. Lo que cambia es que vamos a poder traer de a más bytes de información.

Memoria Comercial



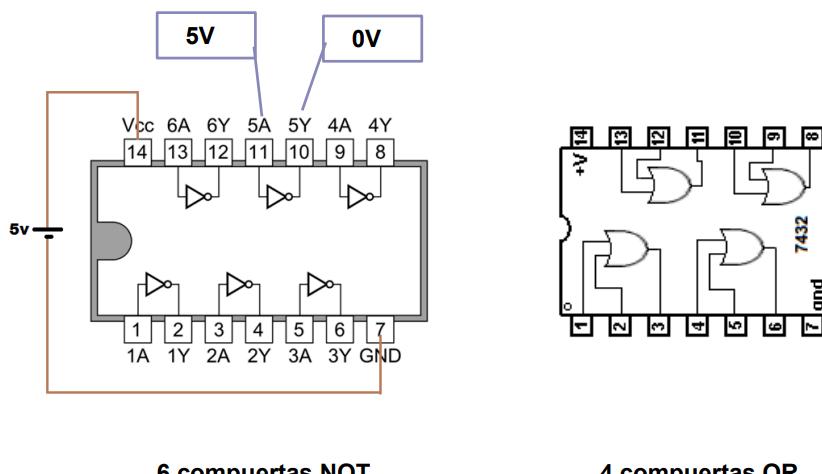
Pin(s)	Function
A ₀ -A ₁₀	Address
DQ ₀ -DQ ₇	Data In/Data Out
S (CS)	Chip Select
G (OE)	Read Enable
W (WE)	Write Enable

Vemos que dice $2K \times 8$, que son 2 kilobyte. El 8 representa bits. Podemos ver en el gráfico y en la tablita los buses. Hay 11 “patitas” porque $2k$ es 2^{11} por 2^{10} , que es 2^{11} . Por eso tenemos las 11 patitas. Además, vemos las 8 correspondientes a los datos.

Integrados Compuertas y Decodificadores

¿Cómo se resuelven las operaciones dentro del procesador? El campo eléctrico del registro se va cuando le sacamos corriente, porque es volátil. Para por ejemplo sumar, se van a sumar las corrientes de los registros que queremos operar.

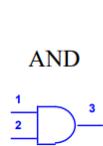
Ejemplo de Integrado (compuerta NOT):



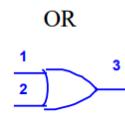
6 compuertas NOT

4 compuertas OR

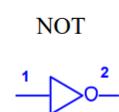
Vemos que tiene un V_{cc} (corriente continua) y un GND (ground), por lo que lo tenemos que alimentar. Los triángulos son compuertas inversoras o compuertas NOT. Si por ejemplo en la 5A ponemos un 1 (5V), lo invierte y lo saca por la salida 5Y. El de 4 compuertas, tiene 1 entrada y 2 salidas, va a hacer la operación lógica del “o” (OR). Repasamos la tabla de verdad:



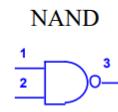
B	A	B.A
0	0	0
0	1	0
1	0	0
1	1	1



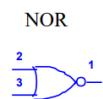
B	A	B+A
0	0	0
0	1	1
1	0	1
1	1	1



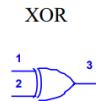
A	\bar{A}
0	1
1	0



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



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



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

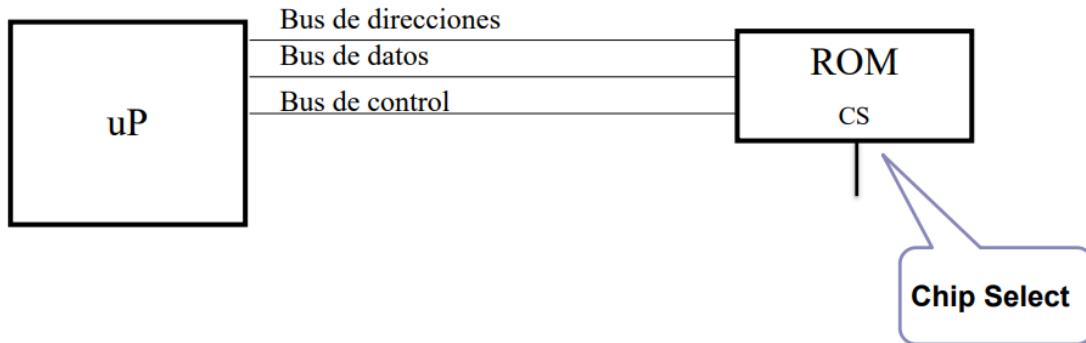


A	A
0	0
1	1

Vemos que el XOR se parece a una suma módulo 2. El buffer no tiene sentido matemático, pero se usa cuando queremos arreglar una señal. Por ejemplo, si bien decimos que el 1 es 5V, en la realidad va bajando. Lo que hace el buffer es “levantarla” nuevamente a 5.

Con todo esto, vamos a poder hacer las operaciones que nos preguntamos antes dentro del procesador. Todo esto se llama microcódigo.

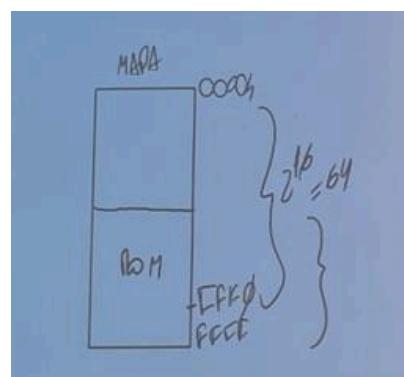
Decodificación de Hardware



Según qué address “salga” por el bus de direcciones, se va a activar el chip select del periférico y este va a tomar lo que sale por el bus de datos. Supongamos que compramos una ROM de 64K y tenemos una memoria de 64K. Todos los punteros que saquemos van a terminar en la ROM. No haría falta hacer nada con el Chip Select, se lo deja conectado a Vcc y funcionando siempre.

El chip select es el encargado de “avisarle” al periférico que es él a quién le “habla” el procesador. Es una “patita” en el periférico. Supongamos que ahora tenemos el mismo procesador de 64K pero una ROM de 32K y una RAM de 32K. Si tenemos $32K = 2^5 (32) * 2^{10} (K) = 2^{15} \rightarrow 15$ patitas.

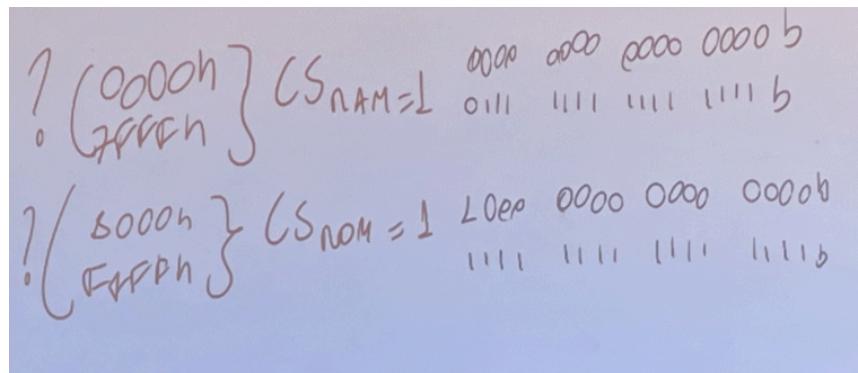
El problema ahora es ver cuál ponemos arriba y cuál abajo en el mapa de memoria. Recordemos que cuando prendemos la compu, tenemos el instruction pointer que debe apuntar a la primera instrucción. Ese dato nos va a decir dónde va cada componente. Si por ejemplo apunta a 0000h, la ROM tiene que ir “arriba” y la RAM abajo.



Cuando el procesador de memoria saque una dirección, queremos que se active el chip select de la ROM en este caso, porque es este el componente que tiene dicha posición y que se apague cuando no se dirige a él. En este caso, como las memorias son de 32K, las direcciones van desde 0000h (16 ceros) hasta 7fffh. Para la ROM van a ser esas direcciones, pero las de la RAM van a ir de 8000h hasta fffffh. Pero

notemos que la 8000h que manda el procesador y “cae” en la RAM, es la 0000h de la misma. Esto se resuelve con los cables y todo el circuito decodificador.

Para todos los circuitos decodificadores que hagamos, tenemos que preguntarnos siempre qué condiciones comunes hay:

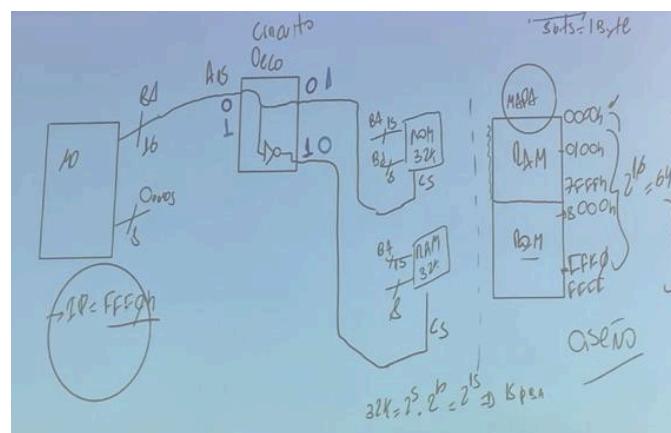


Vemos qué tienen en común. En la ROM (que en la foto está como RAM porque lo hice al revés), todas las direcciones arrancan con cero (bit A15) y en la RAM 1. Por lo tanto, la condición común va a ser el valor del primer dígito (el valor de A15).

$$CS_{ROM} = 1 \rightarrow A15 = 0$$

$$CS_{RAM} = 0 \rightarrow A15 = 1$$

Entonces, ambos Chip Select de la ROM los vamos a conectar con la patita A15 en nuestro circuito decodificador, quedándonos:



Usamos un not para invertir lo que va a la ROM (RAM en la foto porque lo hice al revés). Luego, de la A0 a la A14 van al los Address Bus de la ROM y RAM (recordemos que es compartido porque le llega todo a todos), porque vimos que al ser de 32K, tienen 15 patitas. En general, la que nos sobra va al circuito decodificador. Cuando tenemos que apuntar a algo más chico, más específico tiene que ser el puntero y por lo tanto más “grande”.

Decodificador 3 a 8



Tenemos que saber que cada input tiene un “peso” que se corresponde con:

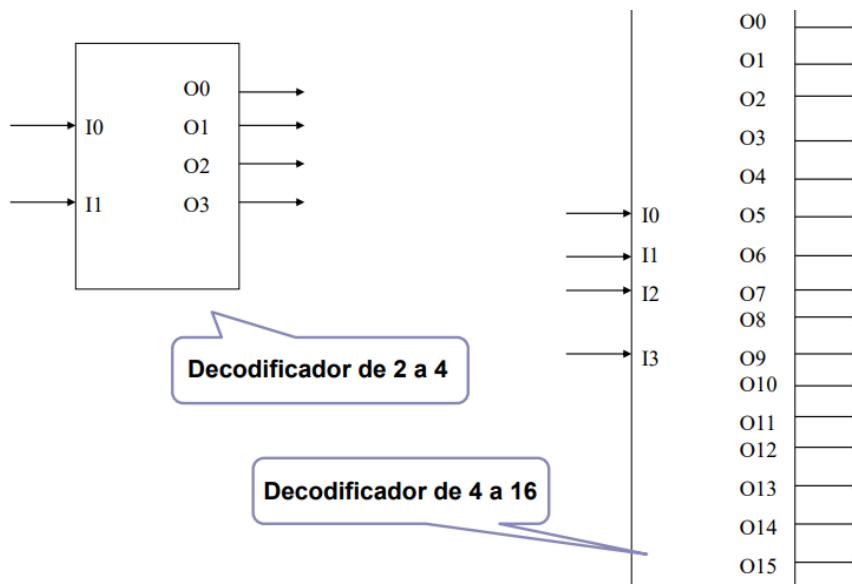
- I₀ tiene peso 2⁰
- I₁ tiene peso 2¹
- I₂ tiene peso 2²

Entonces, según si llega un uno o cero, vamos a multiplicarlo por el peso de su entrada y vamos a sumar todas las entradas. Así, vamos a obtener el output que se “prenderá” para dicha señal. O_x con $x = I_0 * 2^0 + I_1 * 2^1 + I_2 * 2^2$.

Como es importante que todas las entradas estén conectadas correctamente, en caso de no contar con decodificadores más chicos y que sobren inputs, lo que se debe hacer es conectarlos a GND para que esté siempre en cero o VCC para que esté siempre en uno. Por otro lado, los decodificadores tienen chip select, por lo que se pueden colocar a otros decos o se les puede conectar VCC para que esté siempre prendido.

Cuando se necesita más de un decodificador del mismo tipo y se conecta sucesivamente con otros, se llama **decodificación en cascada**. Esto se utiliza por ejemplo para periféricos que ocupan muy pocas direcciones de memoria.

Ayuda a crear los circuitos decodificadores. De 3 a 8 quiere decir que tiene 3 patitas de entrada y 8 de salidas. El objetivo de esto es no usar compuertas. Existen otros:



Tomemos el siguiente ejemplo:

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 (16K de direcciones y 8 de ancho) a partir de la dirección de memoria 8000h.

Lo primero que tenemos que hacer es siempre el mapa de memorias. La primera dirección es 16 ceros y la última 16 unos. Vemos que la dirección 8000h que nos indican, está justo a la mitad. Las dos ROM, como son de 16, van a entrar justo en la parte de abajo. Veamos de dónde a dónde va a ir cada una.

$\text{ROM} \rightarrow 16\text{K} = 2^4 * 2^{10} = 2^{14} \rightarrow$ La dirección mínima va a ser 14 ceros y la máxima 14 unos.

0000h a 3fffh.

Ahora, le tenemos que sumar a la dirección 8000h donde queremos comenzar los 16K de la primer ROM:

$$\begin{array}{r}
 8000h \\
 + \quad 3fffh \\
 \hline
 bffffh
 \end{array}$$

Para la segunda, sabemos que llega hasta el final, fffffh, por lo que podemos restarle 16K:

$$8000h$$

- 3ffffh

c000h

Con esto ya tenemos todo lo que es diseño. Ahora vamos a las condiciones para el Chip Select:

CSrom1 = 1 de 8000h a bffffh

CSrom2 = 1 de c000h a fffffh

Antes, podemos calcular cuántos buses van a ir al circuito decodificador con lo que ya calculamos. Vemos que las ROM tienen 14 patitas, por lo que nos sobran 2 de las 16 del procesador. Esas dos van a ser A15 y A14 y van a ir a:

ROM 1: 1000 0000 0000 0000 8000h

1011 1111 1111 1111 bffffh

ROM 2: 1100 0000 0000 0000 c000h

1111 1111 1111 1111 fffffh

Vemos que la diferencia está en A15 y A14.

Conectamos todos los buses de address y de datos. Luego tomamos un decodificador para el circuito de decodificado. Vemos que vamos a tener 2 entradas y 2 salidas, por lo que tomamos un decodificador de 2 a 4; hay dos salidas que no vamos a usar porque no las necesitamos. Es importante ver que el decodificador tiene entradas I₀ e I₁, y no es lo mismo si mandamos A15 o A14 a cada una, nos modifica toda la tabla de verdad del deco. Tomando las reglas vistas anteriormente: Si A15 es 1 y A14 es cero, va a prender la salida O₂. Esto porque:

A15 * 2¹ (en I₁) → A15 = 1, esa cuenta da 2

A14 * 2⁰ (en I₀) → A14 = 0, esa cuenta da 0

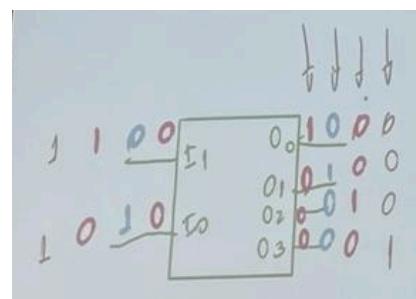
La suma da 2, por lo que se prende el output 2. Como queremos que esto prenda la rom 1, conectamos la misma a la salida O₂.

Para el caso que son los dos un 1, que es lo que se corresponde con la RAM2, vemos que tenemos que conectar el O₃ (output 3):

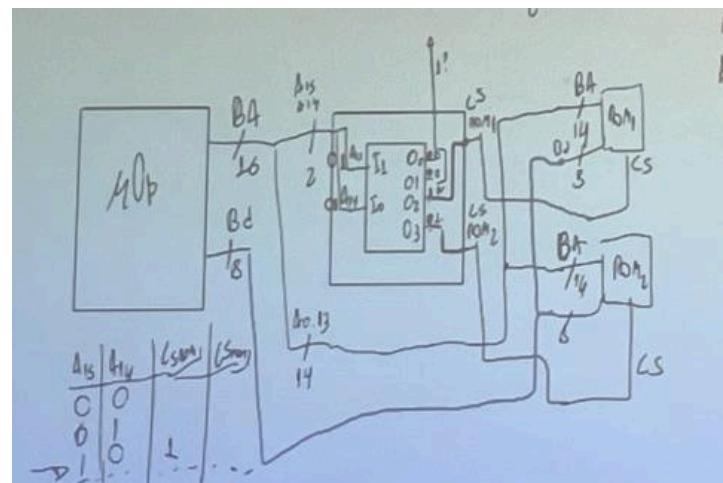
A15 * 2¹ (en I₁) → A15 = 1, esa cuenta da 2

A14 * 2⁰ (en I₀) → A14 = 1, esa cuenta da 1

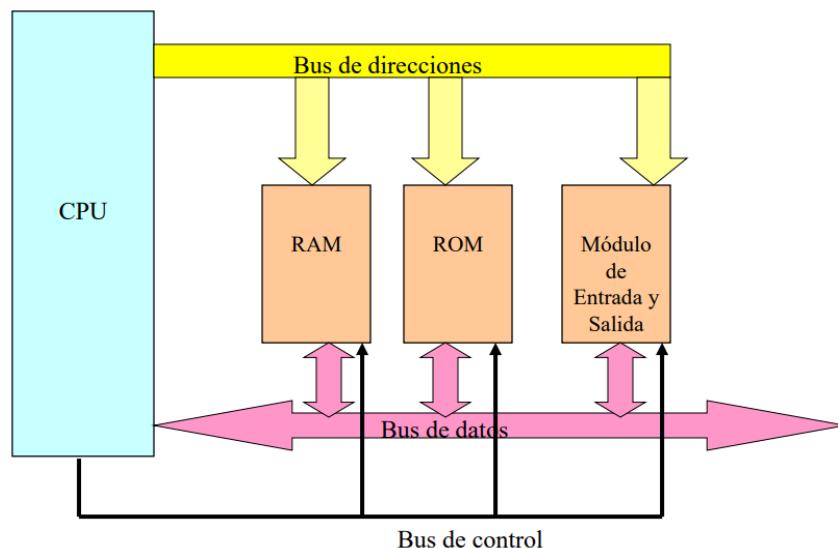
Como la suma da 3, debemos conectar la RAM2 al output 3. Esto lo podemos ver:



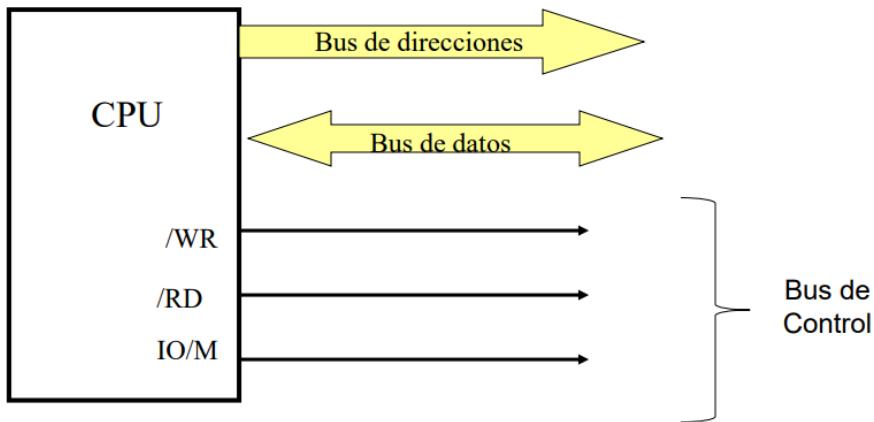
Con esto, todo el circuito decodificador con procesador y periféricos nos queda:



Buses



El bus de control nos va a servir para que la memoria sepa si estamos haciendo un read o write. Nosotros lo sabemos según donde ponemos los corchetes en las instrucciones.



/WR (Cuando vale cero hay una escritura - porque está negado con la /).

/RD (Cuando vale cero hay una lectura - idem).

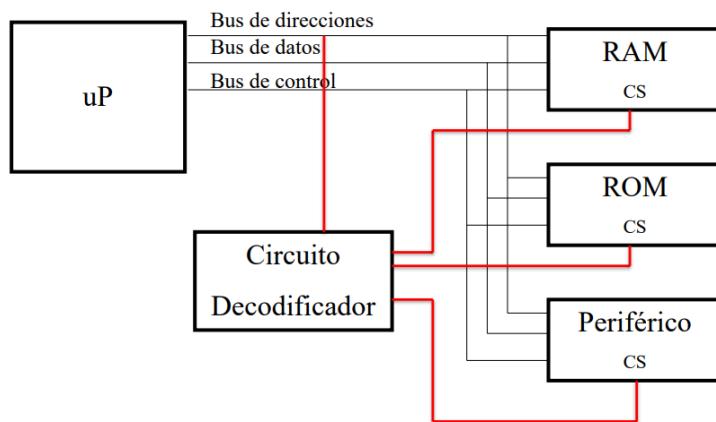
IO/M (Si vale 1: operaciones con ports, si vale 0: operaciones con la memoria - idem). Es como un bit extra para duplicar el mapa de memoria y agregar un **mapa de entrada y salida (E/S)**.

Vamos a tener dos mapas de 64K, uno de memoria y el otro de E/S. Podemos tener por ejemplo, la ROM y RAM en la memoria y el teclado y mouse en el mapa de entrada y salida. Hoy en día, se dejó esta patita porque muchos componentes todavía la requieren para funcionar. Para usarlo, se crearon dos nuevas instrucciones, IN y OUT.

Las memorias, también van a tener sus patitas de read y de write al igual que el procesador, para recibir de él qué necesitan hacer.

Los periféricos como por ejemplo un mouse, van a tener solamente una patita de read (pensado desde el procesador esto, porque se lee lo que indica el procesador). Es un dispositivo de entrada de información. Podría llegar a tener en el caso de un mouse gamer con luces configurables.

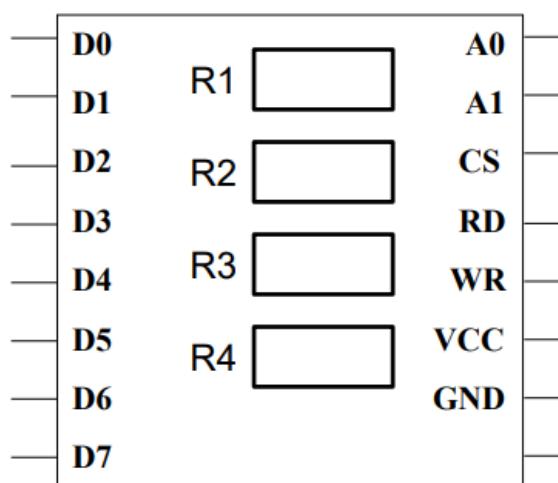
Tipos de Decodificación de Hardware



- *Decodificación completa*: Se dice que una decodificación es completa cuando hay una relación biunívoca (una dirección, un dispositivo) 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".

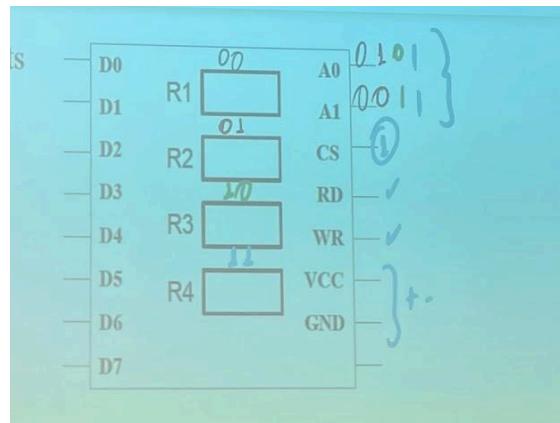
Nunca tenemos que dejar una patita de bus de address sin utilizar. Si por ejemplo, no conectamos la más significativa, se nos rompe el sistema de los chip select, y se puede "llamar" a dos a la misma vez, lo que se llama **imagen**.

Periférico Estándar



Tiene 8 patitas de bus de datos, 2 de bus de address, chip select, read y write, y positivo y negativo. Adentro tiene 4 registros, por lo que está bien que tenga 2 patitas de bus de address.

Si es un mouse, esos 4 registros nos pueden brindar por ejemplo, click derecho, click izquierdo, movimiento en x y movimiento en y. Tiene que tener registros internos porque tiene que almacenar lo que luego va a leer el procesador.



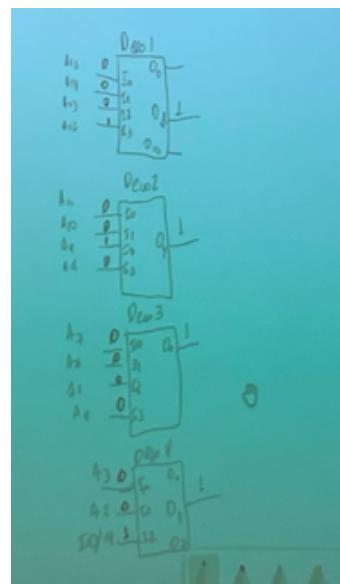
Supongamos que queremos decodificar ese periférico en el mapa de entrada/salida (NO en el mapa de memoria) en la posición 1200h. Dibujamos primero el mapa de entrada y salida. Vemos que va desde las direcciones 0000h hasta fffffh, y buscamos la 1200h. Como tiene dos patitas de address, son cuatro combinaciones y por lo tanto va a ocupar 4 direcciones de memoria, es decir que va a ir de 1200h a 1203h. Para ver cuándo se prende el chip select, tenemos que analizar las 4 direcciones y ver qué tienen en común. Los pasamos a binario:

$$1200h \rightarrow 0001\ 0010\ 0000\ 0000h$$

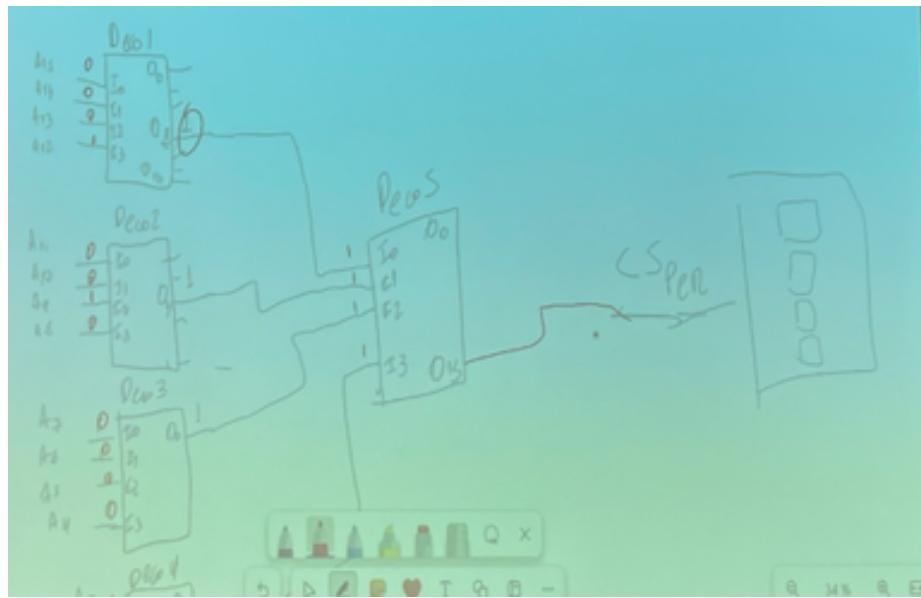
$$1203h \rightarrow 0001\ 0010\ 0000\ 0011h$$

Vemos que tienen en común los 14 bits más significativos. Chequeamos que 14 son las que se lleva el sistema decodificador y 2 las que se lleva el periférico. Vamos a tomar ahora un decodificador. No podemos tomar uno que tenga 15 patitas de entrada porque debería tener 2 a la 15 de salida, que es más de 32mil. Entonces, vamos a tomar varios decodificadores en cascada:

Para los primeros 4 bits, tomamos un deco de 4, lo mismo para los segundos 4 y los terceros, y otro de 3 para los 2 bits que deben coincidir de los binarios y el IO/M. Si se cumplen los 4, se prende el chip select.



Otra opción es poner un deco más conectado a las patitas de los otros decos que nos interesan.



Cómo son pocas direcciones de memoria, nos lleva más, por lo mismo que dijimos antes. En conclusión del ejercicio, vemos que el periférico es lo mismo que una ROM o RAM como las que trabajamos antes pero que sólo tiene 4 direcciones de memoria.

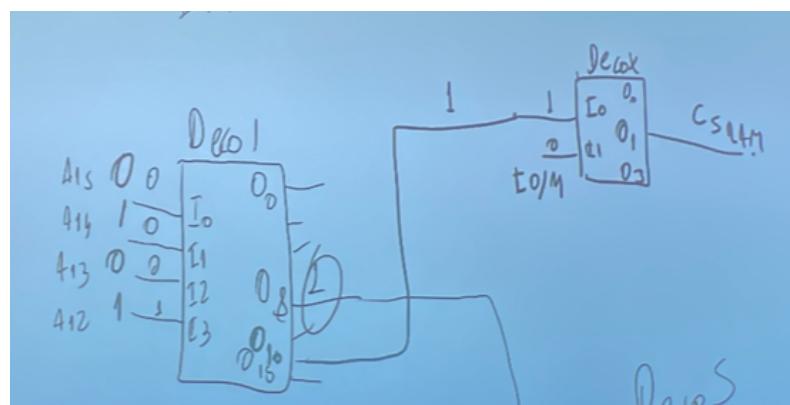
Tomamos este mismo ejercicio, pero ahora vamos a tener una memoria RAM que va a ocupar las direcciones 5000h a 5fffh. Estaría mal empezar el ejercicio de cero, vamos a tomar lo que ya hicimos como punto de partida.

Van a haber 2 a la 12 direcciones, porque va de:

0101 0000 0000 0000

0101 1111 1111 1111

Vemos que coinciden los primeros 4 bits, que son de A15 a A12. Vemos que podemos tomar el Deco 1, y que se nos va a prender el output 10. Pero la RAM necesita que la IO/M esté en 0, por lo que vamos a tomar otro deco, cuya entrada I0 va a estar conectada al output10 y la I1 a la IO/M. Tomamos el output1 (llega 1 por I0 y 0 por I1) y lo conectamos al chip select de la RAM.



Sistema de Entrada y Salida

E/S aislada: Una señal especial del micro indica la ejecución de una operación de E/S. Es lo que vimos de IN y OUT.

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. Algunos periféricos acceden directo a la RAM sin pasar por el procesador. Un ejemplo es un disco rígido. Lo que hace el periférico es activar el chip select de la RAM en algunas “zonas” de memoria predefinidas.

Mapeo en memoria: Se le otorga un sector de memoria principal al dispositivo. Es decir, consiste en poner un periférico en el mapa de memoria.

Mapeo en memoria

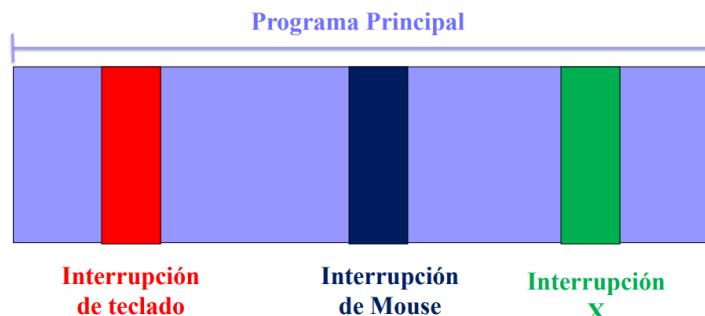
Formato Texto: se divide la pantalla en 80x25 filas y columnas para imprimir caracteres. Podemos verlo cuando se prende la computadora y se inicia el booteo de la BIOS (en las compus de hoy en día está oculto, pero se puede cambiar).



Se toman las 2000 aprox posiciones de memoria (que no son muchas) y se coloca un carácter en cada una. Se tomó como inicio la dirección B8000h. Este fue uno de los primeros mapeos en memoria. Cómo dibujar cada carácter está hardcodeado dentro de la placa de video. Se guarda la letra en el primer byte y las características en el segundo, por lo que son 4000 bytes.

Interrupciones

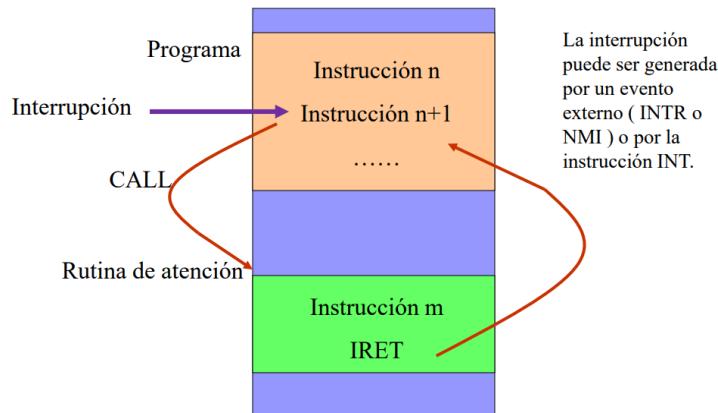
Una señal externa interrumpe al micro para requerir un servicio de atención. Es una señal que interrumpe al procesador para ejecutar lo que se llama rutina de atención de interrupción.



Cuando apretamos una tecla, se interrumpe el programa principal y corre lo que está en rojo, que es una interrupción de teclado x cantidad de segundos, termina y se vuelve a correr el programa principal. Esto pasa porque estamos viendo un monoprocesador (un sólo núcleo), por lo que hay un solo instruction pointer.

Rutina de Atención de Interrupción

En lugar de volver con ret, vuelve con iret, que es lo mismo pero recupera de la memoria otro tipo de información.

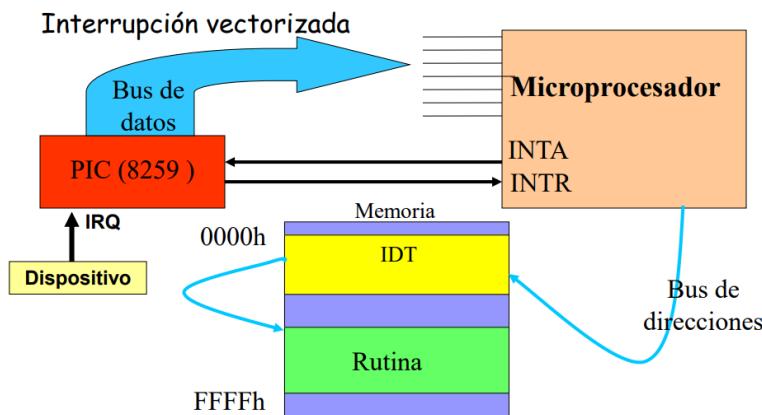


- Interrupción de Hardware: mediante una patita del procesador, le pasamos una cantidad de volts y se interrumpe. Hay dos patitas, INTR y NMI. La primera se puede no prestarle atención (enmascararla). Esto sirve por si el procesador está ejecutando un proceso importante que no debe ser interrumpido (operaciones atómicas). En la otra, NMI (interrupción no enmascarable), lo que pongamos va a sí o sí interrumpir.
- Interrupción de Software: usando la instrucción INT. Es muy similar a call, pero la gran diferencia es que call llama a una función que hicimos nosotros, y el INT a una que hizo otra persona (por lo general el SO).

Para habilitar/deshabilitar las interrupciones en la patita INTR se utiliza el flag IF. Si IF=1 está habilitado y si IF=0 está deshabilitado.

Interrupciones en Multicore: Cada CPU tiene su PIC y afuera existe el APIC donde se conectan los periféricos y dicho PIC decide a qué core le tira las interrupciones.

PIC (Controlador Programable de Interrupciones)



Vemos que en el microprocesador tiene una sola patita de interrupción para periféricos, se incluyó el **PIC** (es un periférico) que permite conectar 8 (pues tiene 8 patitas de interrupción) periféricos. Las patitas tienen el nombre de **IRQ** quiere decir Interrupt Request. El dispositivo interrumpe al PIC, que le “avisa” al procesador y le dice quién es el que quiere interrumpir. Una vez que el procesador sabe qué periférico fue, se fija en la **tabla IDT**, donde hay punteros a todas las rutinas de interacción de los periféricos (los drivers), la va a buscar y ejecuta la misma.

El contenido de la tabla lo programa el que desarrolla el SO. Vemos que la misma está en RAM, por lo que cuando se apaga la compu se borra. Cuando se prende, se recupera del disco cuando se bootea la computadora (al igual que cuando se sube el SO). Esto lo hace la BIOS.

```

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): 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 última 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.

Como no alcanzó con un PIC, se conectaron dos sucesivamente de la siguiente forma:

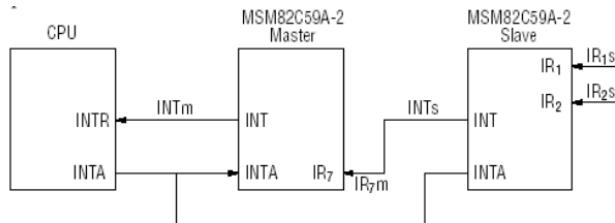


Fig. 1 System Configuration

Interrupciones de Software

```

mov  Ax,0
cmp  Ax,Bx
INT 22h
Add  Cx,Bx
Mov  [Cx],Bx
  
```

El número que ponemos luego de int es la posición en el vector de instrucciones. En este caso, al descriptor ubicado en la posición 22h de la IDT.

Servicio de BIOS

El BIOS al iniciar la PC guarda en memoria rutinas básicas para poder empezar a operar, porque la IDT está vacía en este momento. Estas rutinas básicas están en ROM al igual que la BIOS (que recordemos que al no ser volátil, no se borran cuando se apaga).

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.)
IRQ1	09h	Teclado
IRQ2	0Ah	INT desde 8259A esclavo
IRQ8	70h	Servicio de reloj en tiempo real.
IRQ9	71h	Redirecccionamiento 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

Las IRQ son las patitas en los PIC. Vienen por default y se pueden reprogramar o no.

Excepciones

Son exactamente igual que las interrupciones, pero en vez de ser generadas por un periférico o por un programador que pone INT, son generadas por el mismo procesador, que se interrumpe a sí mismo. Dicho formalmente, es *un evento generado por el procesador cuando detecta una o más condiciones predefinidas al ejecutar una instrucción*. Por ejemplo, cuando explicamos algo y vemos en la cara de otro que no entiende, nos interrumpimos y reformulamos. Entonces, cuando el procesador detecta un error, se autointerrumpe, va a la IDT y busca cuál de los 32 errores fue el detectado para correr la rutina de interrupción de excepción. Las 32 son:

Id	Description
0	Divide error
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
14	Page fault
15	(reserved)
16	Coprocessor error
17-31	(reserved)
32-255	Available for external interrupts via INTR pin

La 13 es la conocida como segmentation fault o el pantallazo azul de windows. Es una protección general. Linux, en vez de poner el pantallazo, es imprimir el Segmentation Fault. Pero el que detectó el error fue el procesador, no el SO, porque cuando corre nuestro programa es lo único que corre. Por lo tanto, como no corre el SO, no lo puede detectar. En conclusión, el SO no supervisa nuestras operaciones porque no está corriendo.

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.

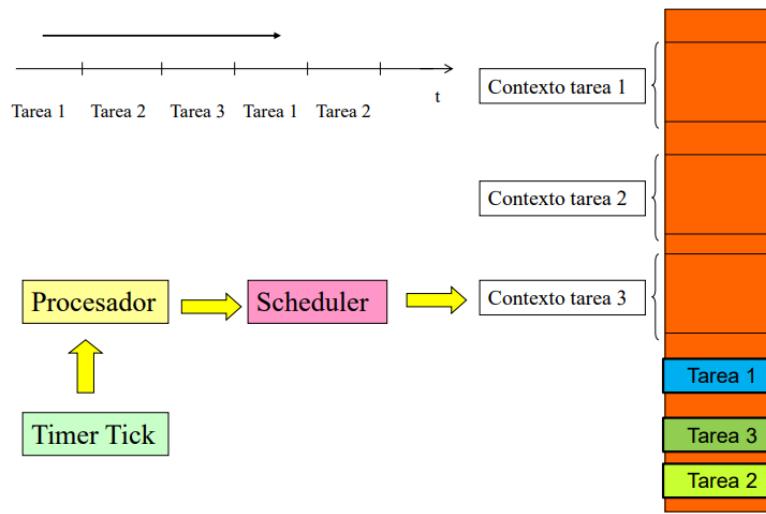
Las excepciones son programadas en el TP por nosotros.

Unidad 5 - Modo Protegido

Los primeros procesadores de Intel no tenían modo protegido, por lo que se podía hacer lo que se quisiera.

Comutación de Tareas

Recordamos que se ejecuta de un proceso a la vez, con una velocidad tan rápida (orden de los milisegundos) que nosotros creemos que se corre todo a la vez. Cada poquito que corre se llama slot time. El Timer Tick es un integrado en la PC que interrumpe al procesador cada 55ms (por default). Esto lo hace siempre, sin importar si el SO es el único proceso. Es enmascarable, porque está conectado a la IRQ0. Es el que permite la comutación de tareas. Lo que hace es interrumpir al PIC como cualquier periférico.



La rutina Scheduler de interrupción lo que hace es decidir a quién se le asigna el procesador en cada momento. Se encuentra en el SO, en el kernel space y fue programada por los desarrolladores del SO.



El instruction pointer no para nunca, por lo que hay un proceso para cuando nadie “quiere” el procesador. En windows se llama “proceso inactivo del sistema”, y su PID es cero.

Cuando se “corta” un proceso que estaba corriendo, tiene que guardar lo que se llama el contexto de tarea (todos los registros de la misma), para que cuando vuelva a recibir el procesador, siga en el mismo estado. Lo que hace es backuppearlo

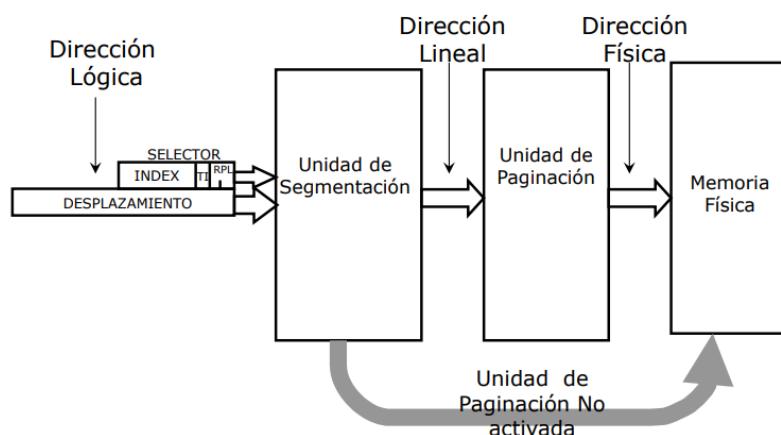
en memoria, y luego cuando vuelve se recupera. Estas tareas, son las que pueden tener errores y mandar el instruction pointer a cualquier lado.

El SO es una tarea igual que el resto, que le “toca” cuando hay una interrupción, un error, etc. El timer tick puede decidir cuánto correr, porque el scheduler lo programó el desarrollador del SO.

Protección de Tareas

Para evitar que una tarea mande un puntero a otra, el único que puede controlar es, como ya vimos, el procesador, porque el SO no está corriendo. Para que el procesador sepa esto, se le dejan unas “tablitas” según desde dónde hasta dónde está cada tarea.

Memory Management Unit (MMU)

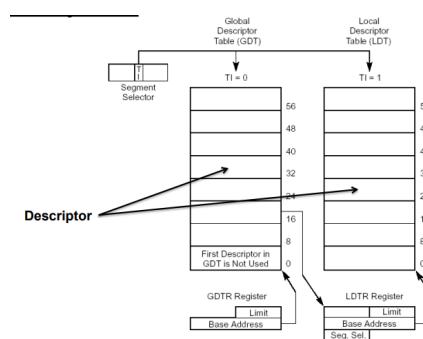


Intel para dividir la memoria creó dos formas. La de segmentación divide la memoria en partes de tamaño variable, y la unidad de paginación en tamaño fijo. La unidad de paginación se puede apagar pero no la de segmentación. Si tenemos un puntero, al usar estas dos unidades, puede sufrir alteraciones. Es decir, si ponemos [100h], no necesariamente sale por el bus de address esa dirección.

En estas dos unidades, no sólo se va a proteger las aplicaciones, sino que también se les va a asignar una dirección virtual de memoria. En RAM tenemos cierta cantidad de memoria pero podemos meter más procesos, porque estas unidades se encargan de mover procesos a disco si no entran en memoria.

Debido a esta limitación de los procesadores Intel de no poder desactivar la unidad de segmentación, algunos sistemas operativos eligen utilizar la memoria en modo “flat” para no utilizar las reglas de segmentación pero sí las de paginación. La página (de donde viene paginación) es el pedacito de memoria que se otorga. La dirección lógica que ingresa el usuario es la misma que la lineal que recibe la unidad de paginación. Se mapea a toda la memoria para “bypassearla”.

GDT y LTD (Unidad de segmentación)



Estas dos tablas indican dónde comienza y dónde termina un segmento de memoria, los permisos, etc. Cada bloque es un *descriptor* del segmento de memoria. En Local Descriptor Table (LDT) van todos los segmentos de memoria de las aplicaciones. Los del SO van en la Global Descriptor Table (GDT). Estas tablas pueden estar en cualquier zona de memoria, en GDTR se guarda la dirección de la GDT (es un registro).

Por cada instrucción de acceso a memoria (`mov [...]`) el procesador se fija si el proceso se corresponde con el espacio asignado en el descriptor correspondiente en las tablas. Sólo se fija si la instrucción es de acceso a memoria, NO si hace por ejemplo un `add` de registros.

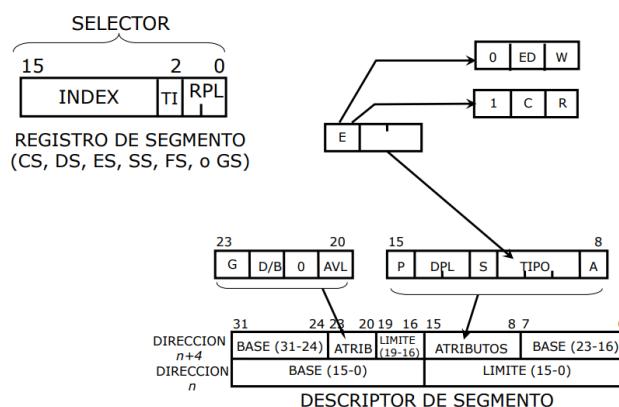
Descriptoros de Segmento

31		16										0		0	
SEGMENT BASE 15...0										SEGMENT LIMIT 15...0					
BASE 31...24		G	0	0	0	LIMIT 19...16	P	DPL	0	TYPE	BASE 23...16				+4
Type	Defines						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=FFFFFH), the lowest 12 bits of the segment base should be zero (ie. segment base 11...000=000H).

Un descriptor es una estructura de datos de 8 bytes. Cada descriptor contiene dónde empieza, dónde termina, qué privilegios tiene, etc. En el segment base se guarda un puntero al inicio del puntero (la base). El segment limit dice el tamaño, no dónde termina. El bit P dice que el segmento que estamos describiendo está en memoria si está en 1 o no si está en cero. Es lo primero que se consulta. Cuando busquemos un segmento en la gdt que no está presente en memoria se dispara una excepción de falta de segmento. Luego se corre una rutina que va a buscar dónde se encuentra. Esto lo guarda en un lugar libre de memoria, no necesariamente el mismo que el anterior, y se modifica el puntero en segment base. Entonces, con esto podemos decir que cuando miramos el instruction pointer, no estamos seguros que realmente esté apuntando allí. DPL es un campo de 2 bits que describe el nivel de privilegio (kernel space y user space - los que se usan hoy en día). 00 son los niveles más altos y 11 los más bajos. Los del medio no se usan.

El type de 4 bits define 16 tipos de descriptor de segmento (nosotros sólo vamos a usar 3).



En el type (o tipo), vemos que el primer bit es una E. Si el segmento no es ejecutable (E en 0), es un segmento de datos. Por ejemplo, para una pila, E tiene que ser cero porque no puede haber código. Si E es 0, vemos que hay una W que permite indicar si se puede escribir o no.

Entonces:

- 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 llamó (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**

El tercer bit dice si se va a la GDT o a la LTD. Nosotros siempre lo vamos a tener en cero porque vamos a trabajar con GDT.

- Accede al Descriptor nro 4 de la GDT.
- Obtiene dirección base (dónde arranca el segmento). (ej) Dirección base = 5600 0000h. Esta es la dirección “verdadera” en memoria física.
- Luego suma offset de 1000 h. Es decir, suma la posición relativa.
- Dirección lineal 5600 1000h. Es la dirección real física a la que vamos a acceder en memoria.

Al sacar de la memoria física y meter el disco, se anota en el P un cero y no se cambia nada más. Si el proceso que vimos se baja a disco y viene otro, se le va a asignar la dirección 5600 1000h pero el DS va a ser 40h por ejemplo.

Cuando nuestro proceso debe volver a memoria, se le asigna una nueva dirección en memoria y se modifica la nueva dirección de memoria física. Lo que se “baja a disco” es todo el segmento, no sólo el proceso. Si por ejemplo tenemos un programa que guarda notas, cuando lo bajamos se tiene que bajar también las notas que se van guardando en memoria a medida que se ingresan.

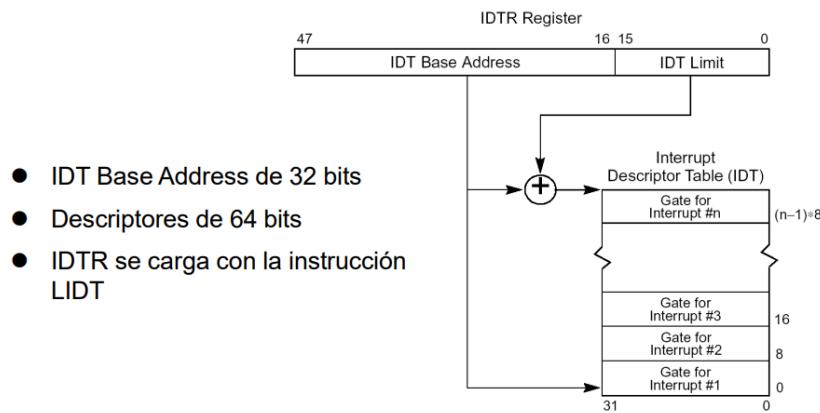
Descriptoros de Sistema

Estos son los 3 tipos que dijimos que íbamos a ver (además de los de código, datos y pila):

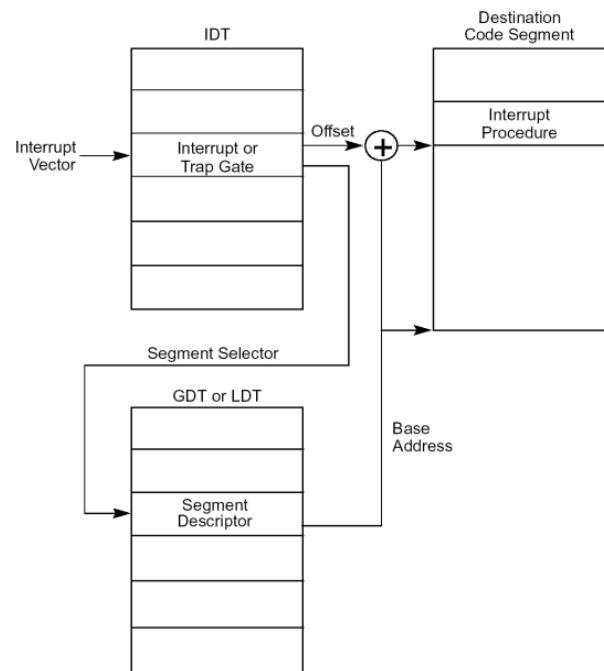
- Trap Gate (Puerta de excepción)
- Interrupt Gate (Puerta de interrupción)
- Task Gate

Estos existen porque Intel tiene una premisa: *todo lo que está en memoria tiene que tener un descriptor*. Estos 3 van a estar en la IDT.

La IDT es la tabla de interrupciones. Va a estar en cualquier posición de memoria, y esto se le indica al procesador mediante el registro IDTR.



Funciona de la siguiente manera:



En destination code segment está la rutina de la interrupción. Como es un segmento, debe tener un descriptor en GDT. Como es una rutina especial por ser una excepción, también se agrega la tabla de IDT. Antes (sin modo protegido) en la IDT estaba directamente el puntero al segmento de código. En la gdt se validan los permisos y se salta para ejecutar la rutina.

El funcionamiento es el siguiente. Los descriptores de sistema tienen guardados los segment selector. Al encontrar la gate deseada en la IDT, el segmento selector indica a qué posición de la GDT (o LDT) se debe acceder. Allí, se accede al descriptor de segmento y se validan los permisos. Si resultan validados, el base address del descriptor de segmento se une con el offset del descriptor de sistema y con eso se obtiene la *dirección lineal*. Si se encuentra apagada la paginación, será la misma que la física. Con esta dirección, se busca la rutina de atención de interrupción deseada.

Interrupciones en PRE TPE

El código necesario para cargar su propia interrupción sólo debería involucrar:

1. Definir la rutina de atención de interrupción.
2. Cargar la interrupción dentro de load_idt invocando a setup_IDT_entry pasandole el número de la interrupción y el puntero a la rutina de atención de interrupción.
3. Si la interrupción es de hardware, dicha rutina debe enviar el EOI y deben habilitar el IRQ con la máscara correspondiente.

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.
- Mapa de bits de E/S en el TSS.

Las instrucciones “sensibles” son:

- IN (Input)
- OUT (Output)
- INS (Input String)
- OUTS (Output String)

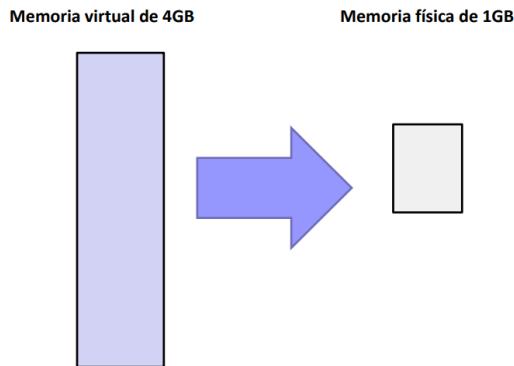
- Cli (Clear Interrupt) - deshabilita todas las interrupciones en el procesador.
- Sti (Set Interrupt)

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

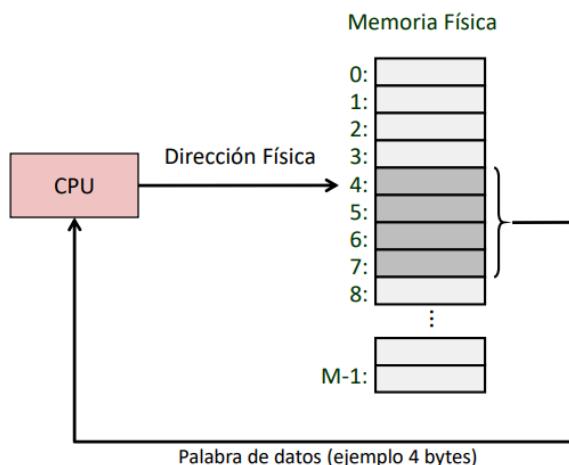
Todas estas instrucciones “sensibles” son privilegiadas, no cualquiera puede ejecutarlas. Solamente las va a poder correr un segmento de código que esté descrito en la GDT con un 00 en el DPL del descriptor.

Memoria Virtual

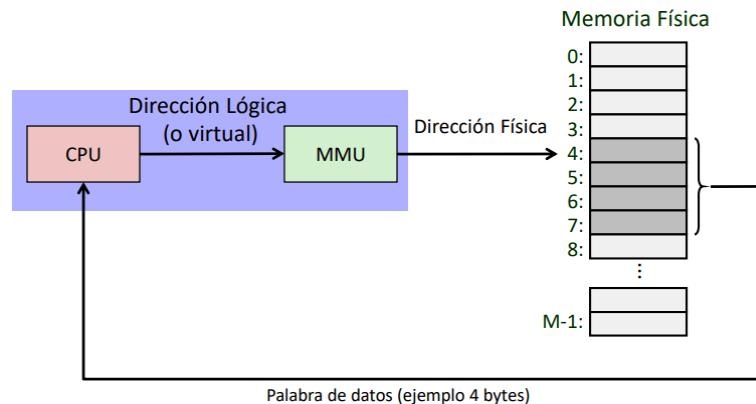
Es tomar una dirección y alterarla, haciendo que los datos reales estén en otro lugar. Es lo que hicimos cuando bajamos programas al disco.



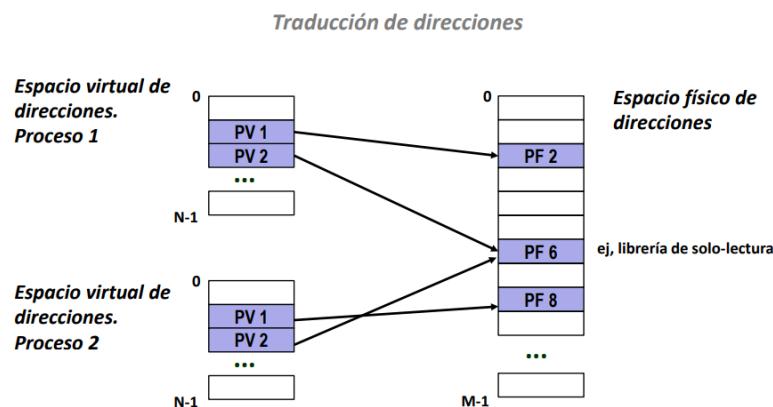
En el direccionamiento físico, sale directamente por el bus de address la dirección física.



Por todo lo que ya vimos, se agregó la MMU (unidad de manejo de memoria) como en la siguiente imagen:



Recordemos que la unidad de paginación divide la memoria en bloques de tamaño constante. Cada bloque se denomina **página**. Cada proceso va a poder ocupar las páginas que necesite (siempre que sea posible) pero nunca va a ocupar menos de una. Es importante destacar que el que maneja las páginas es el procesador, no el SO.



Vemos que hay dos flechas apuntando a un mismo espacio físico, que es una librería de sólo lectura. Esta es una ventaja del mapeo, porque los dos procesos únicamente necesitan leer.

Ejercicio: 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 cuántas páginas quedó dividida la memoria física y la memoria virtual ?
3. ¿Cómo reacciona su sistema para asignar y para liberar memoria?

1) Para elegir el tamaño de página tenemos que pensar cuánto puede llegar a pesar un programa (lo pensamos estadísticamente). Si elegimos un tamaño muy chico,

crece mucho la tabla de descriptores. Entonces, para decidir se debe hacer un análisis de todos los procesos de la computadora (que va a dar una campana de gauss) y sacar la media. Tomamos por ejemplo $1\text{ MB} = 2^{20}$

2) Para saber la cantidad de páginas, dividimos $2^{32} / 2^{20} = 2^{12}$ páginas (son 4096). 32 por 4G.

3) Si tenemos una RAM de 1G, va a soportar 1024 páginas. Cuando se quiera guardar en la página 1025, se va a sacar al disco alguna otra y se va a guardar esta. Es el mismo concepto que vimos antes pero con bloques fijos de memoria.

Elección de Intel para 32 bits

Un procesador Intel como el que venimos trabajando tiene páginas de $4\text{ K} = 2^{12}$. Por lo tanto, para un procesador de 32 bits vamos a tener 2^{20} páginas (dividimos nuevamente $2^{32} / 2^{12}$). Lo que hace intel es que los 12 bits menos significativos sean el offset dentro de la página. Los 20 bits más significativos son “qué página es” de las 2^{20} posibles.

Entonces, por ejemplo:

12345678 h

Índice (en qué página estamos)

Offset

Tomando la dirección podemos saber en qué página va a estar. Esta dirección va a estar entre:

12345 000 h

12345 FFF h

Esto lo vemos sabiendo que 2^{12} son 12 unos que es FFF h. Entonces en cada página vamos sumando de a FFF. Acá vemos por qué se llama offset al 678, porque es la posición relativa.

Como un proceso se puede llevar muchas páginas, sería útil tener un agrupamiento más grande que indique las páginas que se llevó un proceso (por ejemplo facilitaría

liberar cuando ya no se use). Entonces, volvemos a dividir el índice como hicimos antes con la dirección entera. Lo que hizo Intel fue “cortarlo” a la mitad.

Figure 5-8. Format of a Linear Address

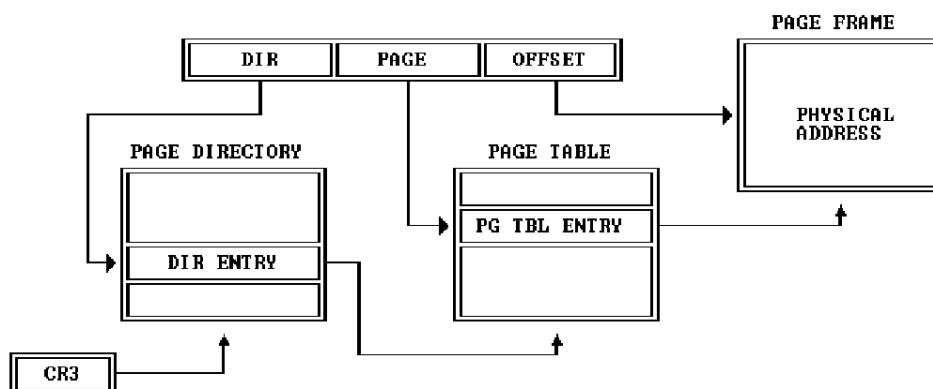


Dir = entrada de directorio.

Page = entrada de página.

Si un proceso requiere 7 páginas por ejemplo, y la primera que le damos es la 0015 5000h. La próxima es 0015 6000 h y la otra 0015 7000h, y así sucesivamente. La última será la 0015 A000h. Al igual que en decodificación, vemos qué tienen en común. Vemos que 001 (10 bits más significativos) se repiten en todas en los bits más significativos. Todo este agrupamiento se realiza para eficiencia, ya que permite acortar tiempos para mover a memoria, asignar permisos, etc.

Figure 5-9. Page Translation



Vemos que tenemos una tabla de direcciones, luego otra de paginación, y finalmente la página de 4k, donde con el offset accedemos al byte específico. Esto se llama **paginación doble**.

Como al directorio de página “entramos” o accedemos con 2^{10} bits, entonces tiene esa misma cantidad de entradas y por lo tanto existe dicha cantidad de tablas de páginas. Como accedemos a estas con 2^{10} direcciones también, van a tener 2^{10} entradas cada una, y por lo tanto esa misma cantidad de páginas. Con esto vemos que tenemos 2^{20} páginas, que es lo mismo que planteamos al principio de todo. Vemos que las mismas están agrupadas de a 2^{10} dos veces. Si miramos cómo va quedando la estructura, notamos que se parece a un árbol.

Si tenemos: mov ah, [12345678h]

678: offset

123 (solo 2 bits del 3): dir

345 (solo 2 bits del 3): pag

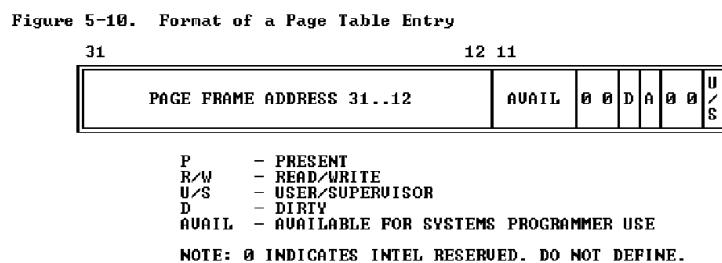
Vemos que:

1	2	3	4	5
0001	0010	00 11	0100	0101
		048h		345h

Entonces la dir va a ser 0x48h para la tabla de direcciones. De allí se obtiene un puntero a la tabla de paginación correspondiente, de la cual se toma la dirección 345h. Este nuevamente es un puntero a la dirección de la página.

Si tenemos dos direcciones de páginas contiguas, no necesariamente las direcciones de memorias físicas también lo están.

La tabla de páginas tiene la forma:



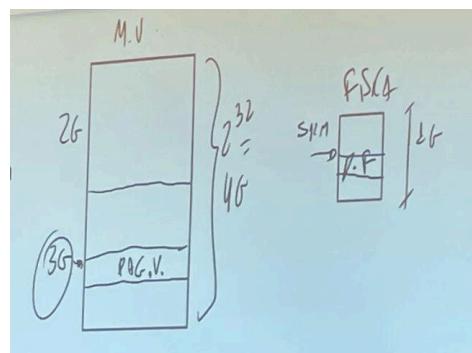
Entonces, va a haber 2^{10} de estas estructuras de datos. La tabla de páginas tiene el mismo formato. El bit *P* de presente funciona exactamente igual que en segmentación. Cuando tiene valor 1, indica que esa tabla se puede utilizar para obtener una dirección de memoria. Si tiene valor 0, no se puede utilizar. Esto en la tabla de páginas. En el directorio de páginas, el 1 representa que la tabla está en memoria y el 0 que no. Si la tabla de página no está en memoria, tampoco están todos los elementos que la integran.

Al encontrar un bit *P* = 0, el procesador genera una excepción. Los sistemas operativos que tienen 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 está ejecutando en niveles 0,1 y 2 es nivel Supervisor, sino ejecuta en nivel 3 es Usuario.

Page frame address: es el puntero que apunta a los 4k de página. Si estamos en el directorio de páginas, apunta a una tabla de páginas.

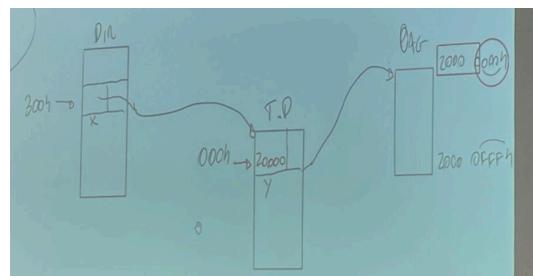
Ejercicio: Se tiene un procesador Intel de 32 bits y 1GB de memoria RAM. Se quiere ubicar un programa en la dirección virtual de 3GB pero que corra en la mitad de la memoria física.



Vamos a tener 4GB de memoria virtual y 1GB de memoria física. Vamos a tener las tablas que vimos antes, que con el bit P van a determinar si se accede a la memoria virtual o a la física. $3GB \rightarrow 3 \times 2^{30}$.

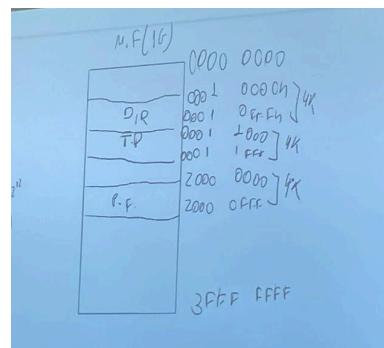
Las direcciones virtuales irán de 0000 0000h a c000 0000h, que se va a tener que convertir en una física que tiene $512M = 2^9 \times 2^{20} \rightarrow$ va a ir de 0FFF FFFF hasta 2000 0000. Por lo tanto, queremos que la c000 0000 se convierta en 2000 0000. Entonces la página nos queda desde 2000 0000h hasta 2000 OFFFh. Entre ambos hay 2^{12} . A esta página apunta la tabla de páginas, con la dirección 20000 (5 bits). Pero primero, veamos en qué dirección va a estar la posición de la tabla de paginación (dirección virtual) y la dirección del directorio de paginación (también virtual):

c	0	0	0	0
1100 0000 00		00 0000 0000		
300h		000h		



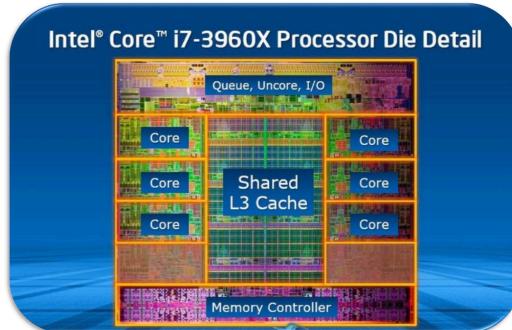
Las direcciones en rojo dentro de las tablas son las reales, mientras que las negras son las virtuales. El CR3 es un registro que va a guardar la dirección donde comienza el directorio. Por ejemplo en este caso tomamos la $0001\ 0000h$. La tabla de página elegimos guardarla en $0001\ 1\dots$

En la memoria física, vamos a tener:



Unidad 6 - Memoria Caché

Casi todos los procesadores tienen memoria caché. Es un espacio de memoria dentro del procesador, pero no es posible programarla. Todo el proceso que hace el procesador con la memoria caché es transparente para nosotros, porque no nos cambia nada en lo que programamos. Es una memoria muy chica (porque es cara) pero muy rápida.

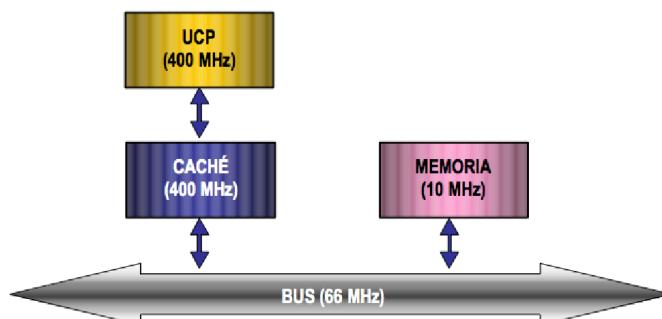


Vemos que ocupa la mayor parte del tamaño del procesador.

Tomemos el ejemplo de un ciclo:

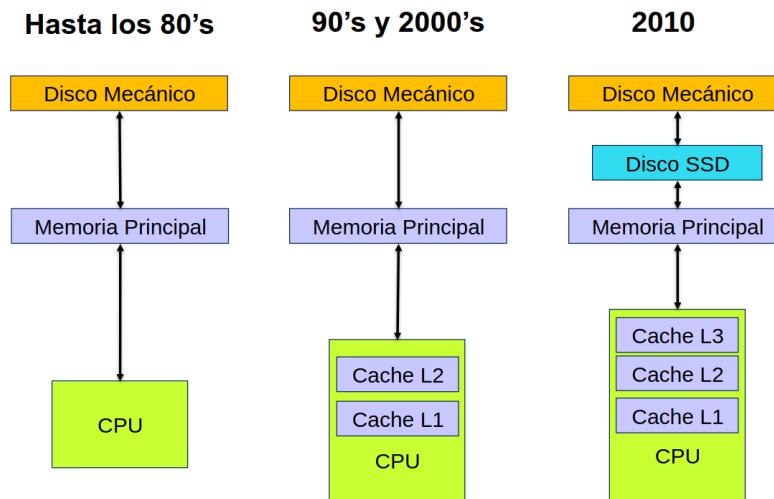
```
ciclo_a:
    mov  EAX,EBP
    mov  [EBP+EAX-24],88
    inc  [EBP-8]
ciclo_b:
    cmp  [EBP-8],15
    jle  ciclo_a
```

Si tiene que ir a buscar la instrucción cada vez que se ejecuta el ciclo, hay una pérdida enorme de eficiencia. Lo que tiene la memoria caché, es que es tan rápida como el procesador. Guarda pequeños bloques de instrucciones que se accede muchas veces, como por ejemplo las instrucciones del ciclo. Se usa nuevamente el concepto de LRU (Least Recently Used) - al igual que en segmentación - para seleccionar lo que se va a guardar en el caché.



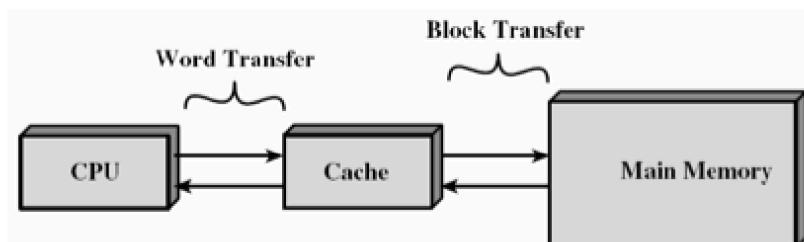
El procesador le pide todas las instrucciones a la memoria caché, no habla más con la memoria. Recibe la información sólo de ella. Si tiene lo que necesita el procesador, accede muy rápido, y si no lo tiene debe ir a buscarlo a memoria y entregárselo. Entonces, si a lo que tiene el caché el procesador accede sólo una vez, no hay ganancia de eficiencia en el tiempo, pero tampoco hay una pérdida. Sí va a estar la ganancia por ejemplo en el caso de los ciclos.

Evolución de las caché:



La L1 es más rápida y la L3 más lenta pero más barata. Lo podemos pensar como una única caché.

La memoria caché divide a la memoria en bloques, sin saber absolutamente nada del contenido, sólo que se lo pidieron. El bloque es el tamaño físico fijo en el que la caché divide la memoria.



Si por ejemplo los bloques son de 32 bytes, si tenemos 1MB de RAM tenemos 32768 bloques. Tomemos además una caché de 4K para datos (sin etiquetas). Entonces:

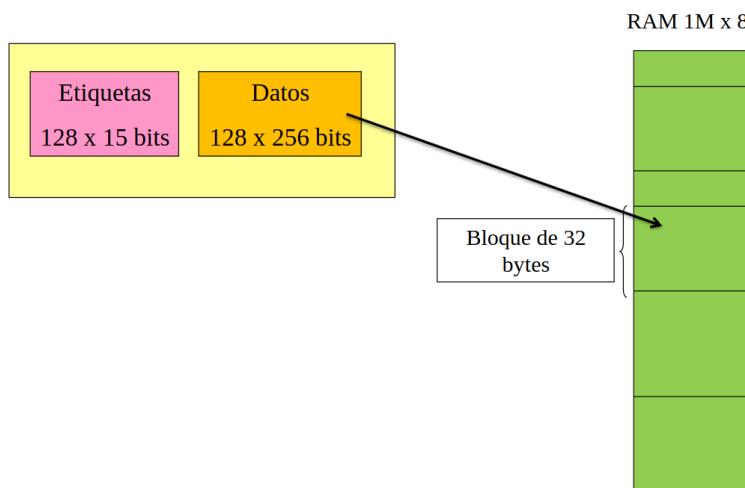
$$4k = 2^{12} \Rightarrow \frac{2^{12}}{2^5} = 128$$

La memoria de datos tendrá 128 entradas de 256 bits (32 Bytes) cada una. Por lo tanto la memoria de etiquetas tendrá 128 entradas de 15 bits cada una porque:

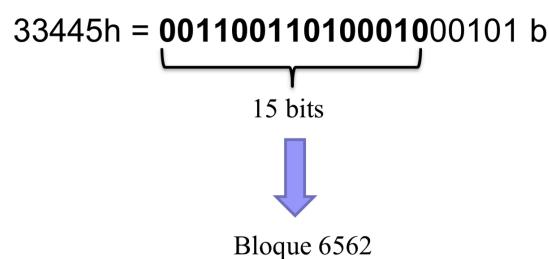
$$\frac{2^{20}}{2^5} = 2^{15}$$

Es decir, como hay 2^{15} bloques, necesitamos 15 bits para poder apuntar a cada uno de ellos.

Etiqueta: puntero a cada bloque existente en la memoria de datos. Entonces, gráficamente la caché será:



Supongamos que nos pasan la siguiente dirección. La caché se fija los 15 bits más significativos para ver si tiene el contenido dentro de las 128 etiquetas.



Trae todo el bloque porque es probable que luego pidamos otro byte del mismo bloque (pues programamos secuencialmente - se basa en la estadística). Luego utiliza los otros bits (offset) para entregar el dato específico.

- Mapeo Directo: Un bloque de memoria solo se puede mapear a una única ranura en el caché. Sencilla, poco utilizada

- Mapeo Asociativo: Un bloque de memoria se puede mapear a cualquier ranura del caché. Compleja, más utilizada hoy en día.

Se actualiza la caché al haber fallo o ausencia de palabra buscada.

- First In First Out (FIFO)
- Least Recently Used (LRU) (necesita flag)
- Random

Actualización de RAM:

- Escritura inmediata (write through)
 - Se actualizan ambas memorias juntas
 - Más lento
 - Más económico
- Escritura obligada (write back)
 - Solo se actualiza lo estrictamente necesario
 - Más rápido
 - Menos económica
 - Problemas con dispositivos DMA
 - Problemas con Multi-cores

Cuando se escribe en un bloque en caché, el que está en memoria se marca como dirty y no se puede utilizar porque está desactualizado. Se actualiza (pisa) cuando baja el bloque a memoria.

Unidad 7 - ARM

Microprocesadores ARM

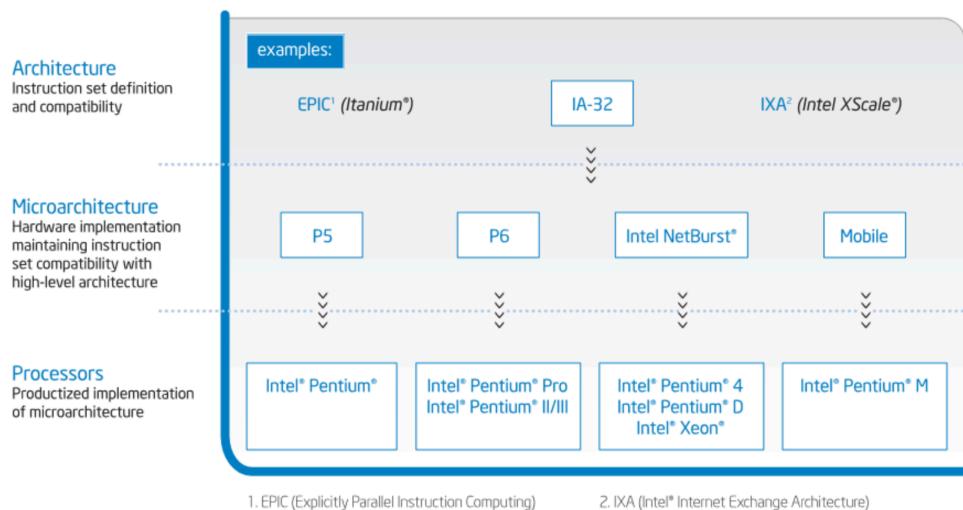
¿Cómo se pasó de 32 bits a 64? No fue de un día para el otro. En el medio de ambos, se intentó “estirar” a los 64. Nombres:

IA-32: Tecnología Intel de 32 bits.

Intel 64/EM64T: Extensión Intel de 64 bits, compatible con 32 bits.

AMD64: Tecnología AMD de extensión 64 bits, compatible con 32 bits.

IA-64: Tecnología Intel de 64 bits (Itanium) no compatible con 32. No lograron hacerlo funcionar, fracasó.



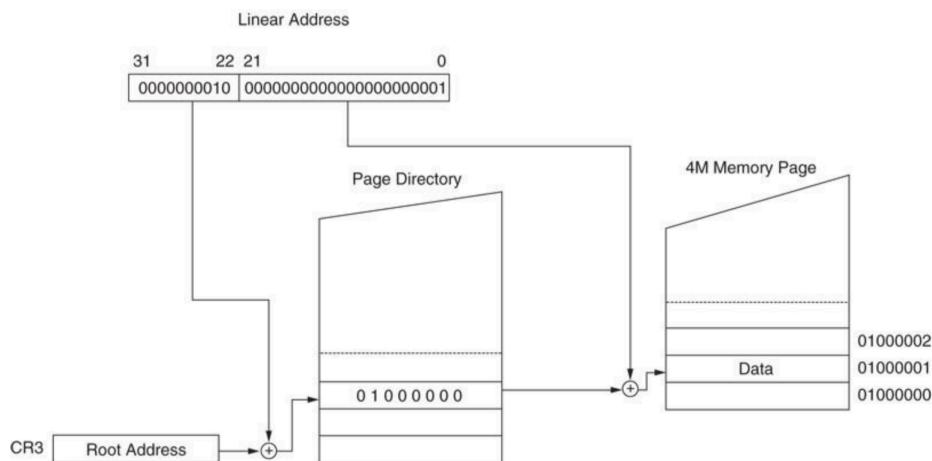
La **arquitectura** es la compatibilidad de un set de instrucciones, es decir, que una arquitectura es la misma cuando comparten un set de instrucciones (por ejemplo, la cartilla de ASM).

La **microarquitectura** es la implementación de hardware de la arquitectura con la que trabajamos. Define dónde va a terminar nuestro procesador, por ejemplo un celular, una computadora o un servidor. Se tiene en cuenta el tamaño que puede tener el procesador, qué temperatura puede levantar, consumo de energía, caché, etc.

Los **procesadores** son la implementación comercial de la microarquitectura.

Pentium

Es un procesador de 32 bits pero con un bus de datos de 64 bits. Se modificaron las páginas para manejar dos niveles y que sean de 4 MB. Vemos que no hay tabla de páginas, por lo que hay un solo nivel de indexación. Esto porque la página se hace más grande.



En el **Pentium Pro**, se pusieron 36 líneas de bus de direcciones y 64 líneas de bus de datos. Se intentó replicar la utilidad de la patita IO/M. Las 4 patitas permitían elegir a cuáles de los 4Gb de memoria se podían acceder. El problema es que no se podía correr un programa de más esos 4Gb. Estas patitas se llaman PAE (Physical Address Extension). El sistema operativo tiene que proveer de un manejo de memoria para lograr utilizar los 64 Gb de direcciones físicas. Las direcciones virtuales siguen siendo de 32 bits.

Intel 64 / EM64T

Se implementan los registros de 64 bits y todo realmente completo en dicha arquitectura (direcciones virtuales de 64, instrucción de direccionamiento a memoria de 64 bits, etc.). Lo importante es la retrocompatibilidad. Si conectamos un disquete de los 80s en la computadora, en teoría el procesador lo puede correr, pero no nos va a dejar el SO porque ya está todo seteado en 64 bits.

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 Address 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

La empresa ARM logró establecer su procesador como el principal utilizado en los teléfonos celulares. Se juntaron 3 empresas (Acorn, Apple y VLSI) para crear una que haga procesadores. Los diseña, NO los fabrica. Hacen todo el proceso que vimos de definir cuántas patitas, qué temperatura puede soportar, etc. Por lo tanto es una empresa de software que hace diseño y simulación. Vende licencias para fabricar procesadores.

Los microprocesadores ARM tienen una alta relación MIPS/watt. Watt es el consumo y MIPS son las millones de instrucciones por segundo. Lo que quieren, son procesadores que consumen muy poco y ejecutan muchas instrucciones. Por esta razón, se hicieron muy útiles para dispositivos móviles.

Diseño de Procesadores

Se utiliza software para simular el microcódigo que realizan las compuertas de silicio

- Simulación lógica
- Simulación eléctrica
- Simulación térmica

Ejemplo de lenguaje : VERILOG. Es un lenguaje de tipo HDL (Hardware Description Language).

Algunos productos con ARM:

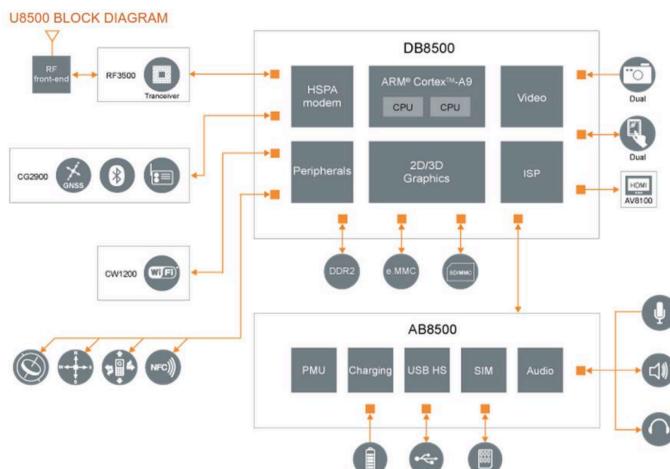
- Apple Newton (Se lanza en 1993. ARM 610 RISC. Muchos bugs, caro, fracasó).
- Nokia 6110 (Texas Instrument fabrica un ARM y lo provee a Nokia. Se lanza en 1997. Arquitectura ARM 7. Primer teléfono GSM (Global System for Mobile communications) con ARM. Exito total).

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

System on a Chip (SoC)

Consiste en pensar los chips para la finalidad que van a terminar siendo usados. Por ejemplo, ya vienen con ROM, RAM, etc. No son upgradeables. Un SoC está integrado por:

- Procesador
- Memorias (ROM, RAM, Flash)
- Osciladores
- Conversores A/D y D/A (A = analógico y D = digital). Permite digitalizar señales analógicas (como por ejemplo el audio captado por el micrófono) y su proceso inverso.
- Interfaces (USB, Ethernet, USART)



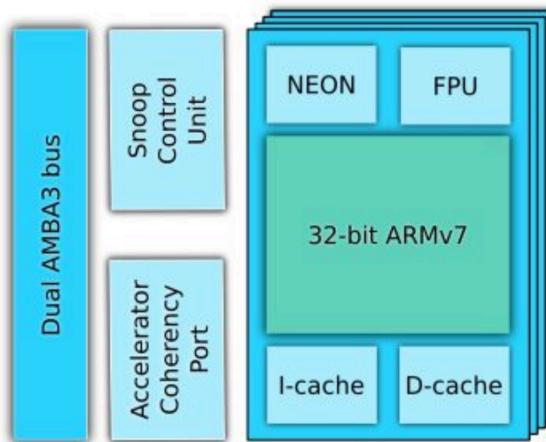
Este es un ejemplo de un SoC llamado Novathor.

ARM Cortex A9

Tiene una arquitectura ARMv7-A. Vemos que es una versión 7. Tiene además:

- Simple ó Multi Procesador (hasta 4)
- Varios niveles de control de consumo
- Basado en RISC
- Instrucciones ARM y Thumb
- 37 registros de 32 bits

Por otro lado, tiene caché de Instrucciones/Datos de 16, 32 o 64 KB y una arquitectura Harvard modificada. Vemos que son dos las caché que tiene.



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
 - L2 Cache 1 MB
- OMAP4 (Motorola Razr, kindle Fire)
- Exynos 4 (Samsung Galaxy S3, Samsung Note)
 - Quad Core ARM-Cortex-A9
 - CPU 1.4-1.6 GHz

Veamos las familias de arquitecturas:

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

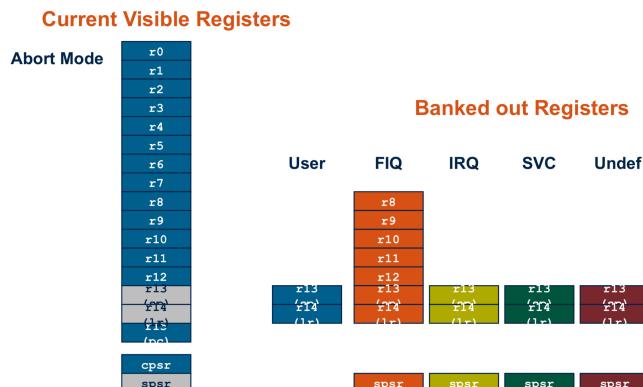
Las arquitecturas no son compatibles entre sí, como es el caso de Intel.

- 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

Vemos que hay varios modos, donde el user space está en modo sin privilegios. Es mucho más detallista en el modo en el cual trabaja el procesador. Nosotros en Intel usamos solo el modo protegido y nada más. Acá hay otros modos que se deben conocer si se quiere desarrollar por ejemplo un sistema operativo.

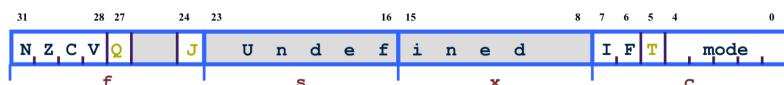
Registros



Vemos que el r15 es el instruction pointer y se llama program counter.

- ARM has 37 registers all of which are 32-bits long.
 - 1 dedicated program counter
 - 1 dedicated current program status register
 - 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)

Tiene al igual que Intel un registro de 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
- **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
- **Mode bits**
 - Specify the processor mode

Características Generales

En el comienzo de la computaciones, existieron dos tipos de procesadores:

- RISC: set de instrucciones reducido. Toma conjuntos de instrucciones pequeñas y simples que toman menor tiempo para ejecutarse. ARM se posicionó en este grupo.
- CISC: set de instrucciones complejo. Hay muchas instrucciones para hacer casi todo lo que queramos, no significa que las instrucciones sean complejas. Se ayuda mucho al programador, porque permite mayor versatilidad en las funciones. Intel se posicionó en este grupo.

Casi todas las instrucciones se ejecutan en un ciclo de clock y tienen tamaño fijo. Esto significa que las instrucciones tardan casi todas lo mismo. Ayuda mucho en la sincronización de códigos. La desventaja es que todas las instrucciones deben tener un tamaño fijo, y por lo tanto hay algunas que les sobra espacio libre y se desperdicia.

El clock es un periférico de una única patita conectado a todos los componentes conectados a la mother. Envía unos y ceros. Cuando los componentes reciben un uno trabajan, y cuando reciben un cero no. Esto permite sincronizar. En arm, con una señal de clock se ejecuta todo por el tamaño fijo. En Intel no pasa, y una instrucción grande puede tardar varias señales de clock en ejecutarse. Si se levanta demasiado la frecuencia, se funde. Levantar la frecuencia se denomina overclocking.

Todas las familias de procesadores comparten el mismo conjunto de instrucciones

- Tipo de datos de 8/16/32 bits.
- Pocos modos de direccionamiento.
- No se crea fragmentación de memoria.

Mapa de Memoria

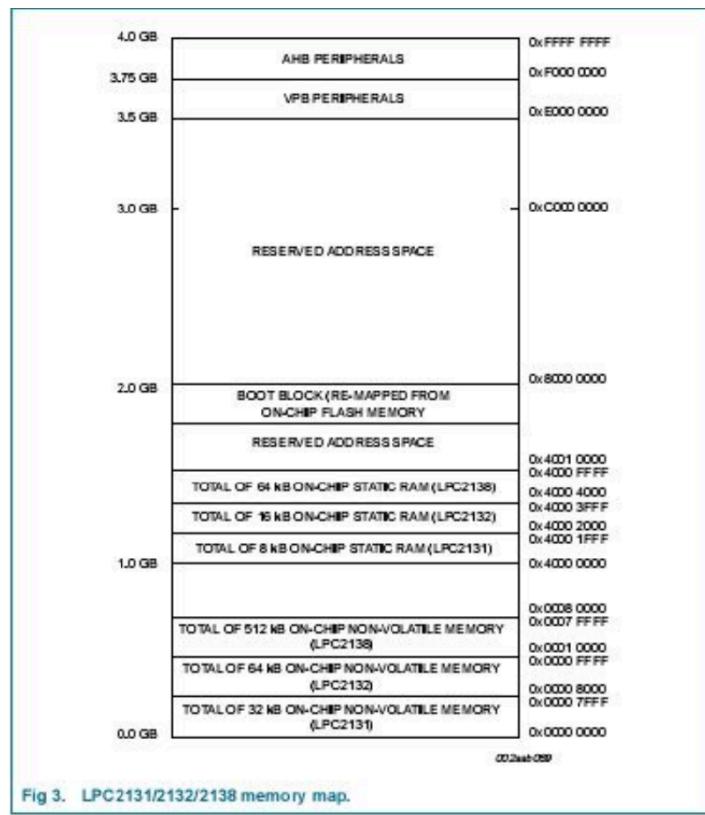
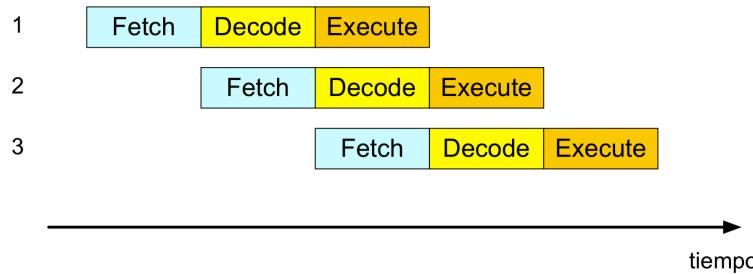


Fig 3. LPC2131/2132/2138 memory map.

Vemos que es al revés de como lo dibujamos nosotros, con las direcciones bajas abajo. En las primeras direcciones se encuentra la rom y ram. Vemos que ya están mapeados los periféricos en la zona alta a partir de 3.5Gb, por lo que no hay chip IO/M.

Pipeline en ARM7

Es un concepto que existe en todos los procesadores (Intel, ARM y todos lo que se nos ocurran). Cuando se quiere ejecutar una instrucción, hay por lo menos 3 etapas como se ve en la imagen.



- Fetch: ir a buscar a memoria la instrucción y traerla al procesador.
- Decode: decodificar la instrucción (ver qué instrucción es).
- Execute: ejecutar.

Se descubrió que mientras se iba a buscar (fetch), no se hacía nada, por lo que se hicieron “paralelizaciones”. Por lo tanto, mientras que se decodifica una instrucción, los buses van buscando la próxima en paralelo. Mientras se decodifica esta nueva instrucción, se ejecuta la primera y se va buscando la tercera.

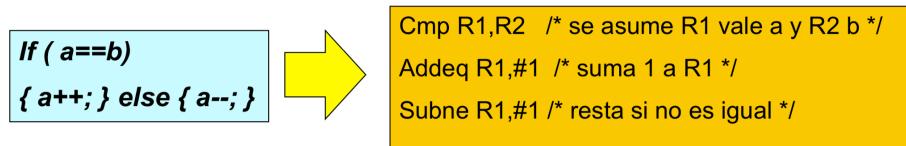
En la imagen, vemos que parece que todas se ejecutan en el mismo lapso de tiempo, pero en realidad fetch es casi seguro la que más tiempo lleva. A lo sumo puede ser comparable con el execute de por ejemplo un mov con corchetes, porque se debe volver a memoria.

Una instrucción de salto provoca que se vacíe el pipeline, pues si hay un jump y vamos haciendo el fetch de la de abajo, esto resulta innecesario y se hace muy ineficiente por ejemplo en un ciclo.

Cuando se detecta que hay un salto condicional (en el decode), se analiza el código en un sandbox para intentar predecir (branch prediction) cuándo va a saltar y no hacer las operaciones innecesarias.

Instrucciones en ARM

ARM agregó condiciones a todo. Como los jne (jump not equals), pero ahora por ejemplo también se puede hacer “sumar si”.



Como las instrucciones tienen todas un tamaño fijo, se tomaron los 4 bits más significativos como la condición (la condición 0000 es equals). El decode lee estos bits, y solo si se cumple la condición decodifica y ejecuta la instrucción. Con esto se logró mejorar el funcionamiento del pipeline, haciéndolo más eficiente.

31

28

Condición (4bits)	(Otros campos de la instrucción)
-------------------	-----------------------------------