

# Estructura de Computadores

Antonio Cañas

Despacho 2a Dept. ATC

M, X 13:30 - 14:30

J 8:30 - 11:30 , 13:30-14:30

Mensajería preferible por SWAD

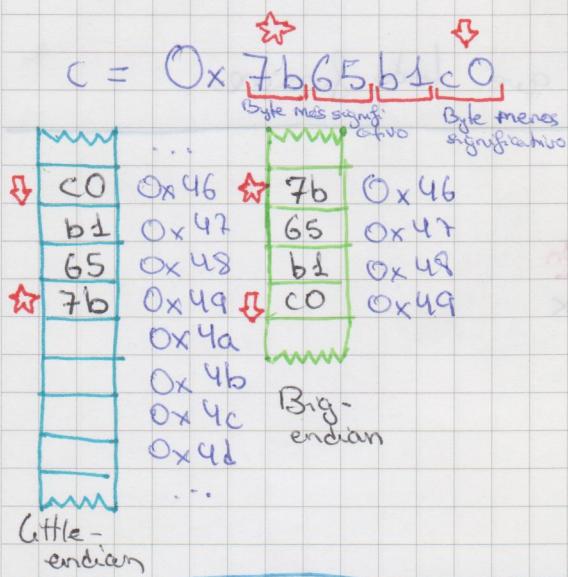
¡Hacer problemas!

Ojalá cada problema, pero si vive  
para poder aprender.

## 0. Introducción

/\* Leer TAC, clase con Sheila \*/

### Organización en bytes



El tamaño en posición de memoria no es el mismo que el del registro de la CPU.

(as posiciones suelen ser de 1 B, ya que no necesitan empaginar strings, pero sus problemas son el alineamiento y el ordenamiento)

### Clasificación Registro / Memoria

Clasificación  $M/n$  ( $M$ emoria /  $n^{\circ}$  de operandos)

Ej:  $mov \$0x14, @ordi$

op.memoria → operando

0/0: Maquinas pila, sólo sacan y meten cosas en la pila.

1/1: las operaciones ALU admiten 0 operandos de memoria de los 0 operandos totales.

1/2: Maquinas acumulador, sacan y meten datos de memoria, pero lo normal es operar entre la pila y la memoria. Solo tiene 1 operando que es de memoria.

x/2, x/3: Registros de propósito general; las operaciones se realizan entre registros (R/R) y entre registro y memoria (R/M). Tienen 2 o 3 operandos, de los cuales x son de memoria. x/2 es lo más común.

### La arquitectura de Von Neumann

Consta de 5 componentes

Entrada / Salida Transmisión y digitalización de dispositivos que introducen información (o la extraen) al / del ordenador.

Memoria Almacena los datos.

CPU (Unidad de procesamiento central) Procesa la información de las entradas y la memoria ejecutando programas.

Unidad Aritmético-Lógica Se encarga de operaciones

Unidad de Control Se encarga de que todo funcione.

Lenguaje ASM Cod. op. Campos de dirección  
mov %rax, %rbx, %rax

↓

Lenguaje máquina OD 14 2A

↓

Binario 0011 ...

Sobre la memoria.

La memoria se divide en palabras (<sup>pequeñas</sup> grandes) y posiciones (<sup>grandes</sup> pequeñas).

La memoria es accesible de forma aleatoria (RAM), lo que significa que el tiempo de acceso a cualquier posición es el mismo.

! Un disco duro **no** forma parte de la memoria principal.

Repetitorios

ISA (Instruction Set Architecture) se ocupa de instrucciones a bajo nivel.

RISC Repertorio de instrucciones reducidas,

RPG tipo 0/2, 0/3, pocas instrucciones y modos,

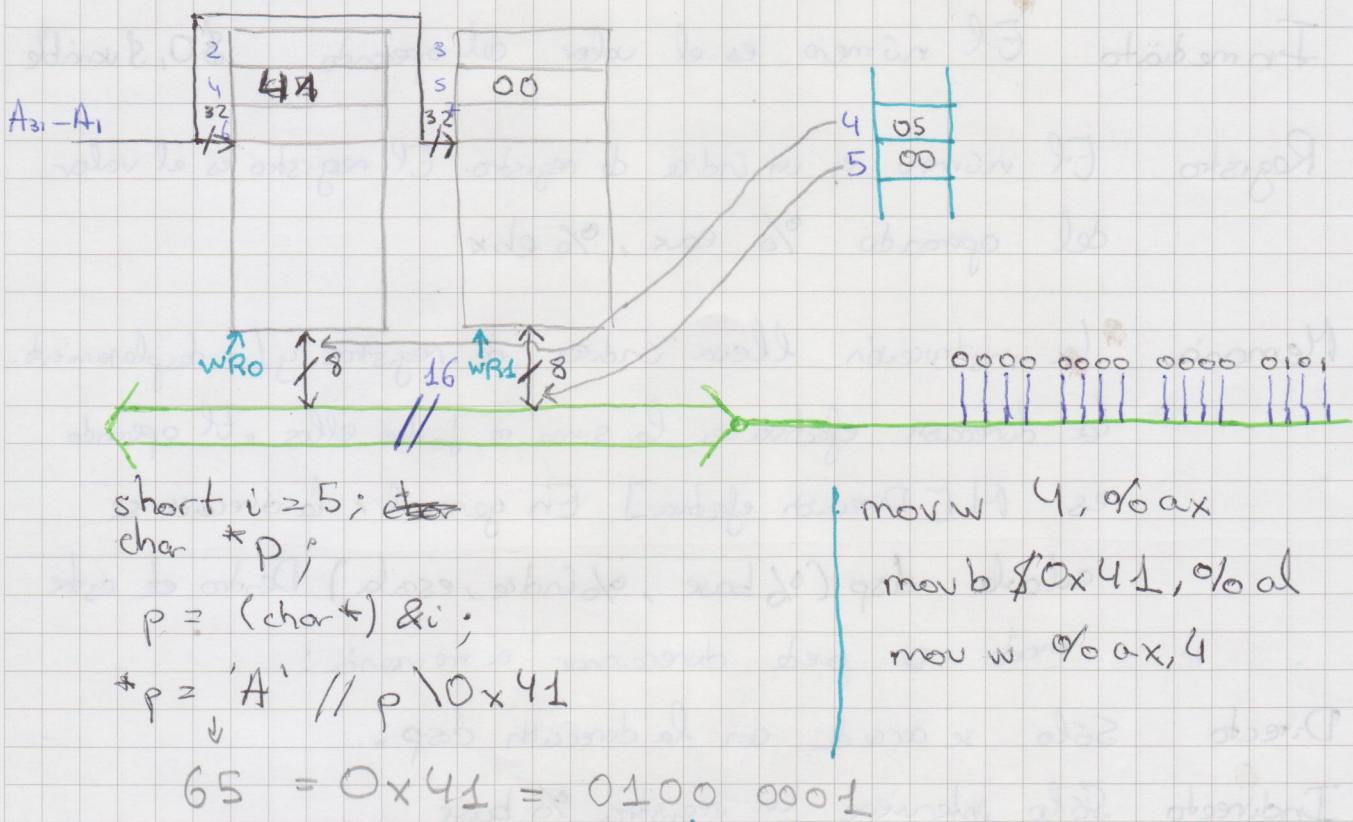
instrucciones sencillas. Tiene una Unidad de Control simple.

CISC Repertorio de instrucciones complejo.

RPG 1/2, 1/3, más próximas a lenguajes de alto nivel.

Tienen instrucciones de tratamiento de memoria.

## Alineamiento en memoria de Bytes



d'Qué queremos escribir?	WRO	WR1
Byte - posición por	1	0
Byte - posición impor	0	1
Short - posición por	1	1
Short - posición impor	No se puede	

El hecho de no poder acceder en 1 solo ciclo de reloj a un short que comienza en posición impor hace que se tarde más.

Por ello, los datos deben estar alineados de n en n bytes. Algunos CPU requieren que los datos estén alineados, o hacen que tarden menos.

Si el tamaño de palabra es de...	La dirección es múltiplo de...
16 bits	2
32 bits	4
64 bits	8

## Modos de direcionamento

Inmediato El número es el valor del operando \$0, \$variable

**Registro** El número es un índice de registro. El registro es el valor del operando %eax, %ebx

Memoria la instrucción lleva índices de registros y/o desplazamiento.  
La dirección efectiva es la suma de todos ellos. El operando  
es  $M[\text{Dirección efectiva}]$  En general, la dirección se  
calcula  $\text{disp}(\% \text{base}, \% \text{índice}, \text{escala})$  Dentro de este  
modo se puede direccionar la memoria:

**Directo** Sólo se accede con la dirección disp.

Indirecto Solo interviene un registro % base

Relativo a base: Intervene un registro %base y un desplazamiento disp.

**Indiceado** Usa el registro %base (a veces), el registro %index y la escala (que multiplica por índice)

## Combinado

! Direcccionamiento de un byte.

8049079 : b9 98 90 04 08 mov 0x049098,%eax  
7c 7d 7e 7f 80 Dirección

8048081 : . . .

## Rendimiento

$$T = \frac{E \times S}{D}$$

## 2. Nivel máquina 1. Operaciones básicas.

### Calcular la posición de memoria

$D(Rb, Ri, S)$

↓      ↓      ↓ Escala  
Desplazamiento | Reg. base | Reg. índice

Rb y Ri son registros

D y S son enteros.

Cálculo pos. mem:  $D + Rb + Ri \cdot S$

### Instrucciones

lea SRC, DEST (Load Effective Address) Calcula la dirección de memoria de SRC y lo guarda en DEST.

a ddq SRC, DEST

Suma: Dest = Dest + Src

mult: SRC, DEST

Multiplicación: Dest = Dest · Src

xor SRC, DEST

Suma lógica exclusiva: Dest = Dest  $\oplus$  Src

and SRC, DEST

Producto lógico: Dest = Dest & Src

or SRC, DEST

Suma lógica: Dest = Dest  $\mid$  Src

inc DEST

Sumar 1: Dest++

dec DEST

Restar 1: Dest --

not DEST

Negación: Dest

### Sufijos

Byte (char) b -1 Quad word (long, char\*) q -8 To das los punteros ocupan

word (short) w -2 Punto flotante (float) s -4 8 Bytes

Double word (int) l -4 (double) l -8

z - Rellena con ceros s - Rellena con bit de signo

### ▷ Ejemplo (la función swap())

void swap(long \* a, long \* b){

long f0 = a

long f1 = b

\*f0 = f1  
\*f1 = f0

swap: movg (%rdi), %rax

movg(%rsi), %rdx

movg(%rax, (%rsi))

movg %rdx, (%rdi)

ret.

▷ ¿Se podría usar un solo registro para hacer swap()?

No, porque se estaría efectuando una instrucción memoria-memoria, algo que no se puede hacer

swap: movq (%rdi), %rax  
movq (%rsi), (%rdi)  
movq %rax, (%rsi)

## Instrucciones shift.

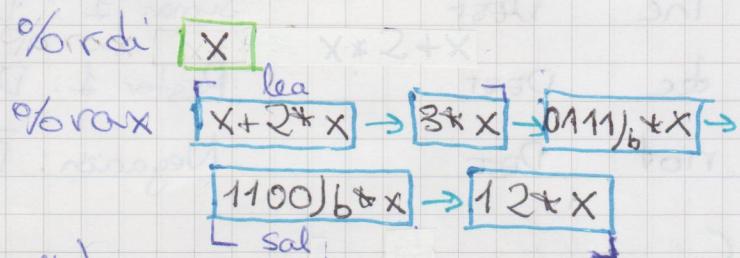
Permiten añadir ceros por la derecha o por la izquierda.

▷ Ejemplo Multiplicando por 12.

long m12 (long x) {  
 return x \* 12;

m12: leaq (%rdi,%rdi,2),%rax  
 salq \$2,%rax

MIGUEL RUS



shl Dest (Shift Left)

sal Dest (Shift Arithmetic Left)

shr Dest (Shift Right) (introduce un 0, uso en productos s.n.s.gra.)

sar Dest (Shift Arithmetic Right) Se usa en multiplicaciones con signo, repite el signo

▷ ¿Qué pasa si queremos dividir un número negativo? !

-5 1111 1011

$$101 \overline{)} 010(+1 = 0 \text{ } \begin{smallmatrix} 1 \\ 2 \end{smallmatrix}$$

Dividir entre 2 (sacar el signo)

$$0111 \overline{)} 1101 = \cancel{5} + 125 ?$$

En ese caso, añadiremos bits de signo en vez de ceros.

! shl y sal hacen lo mismo.

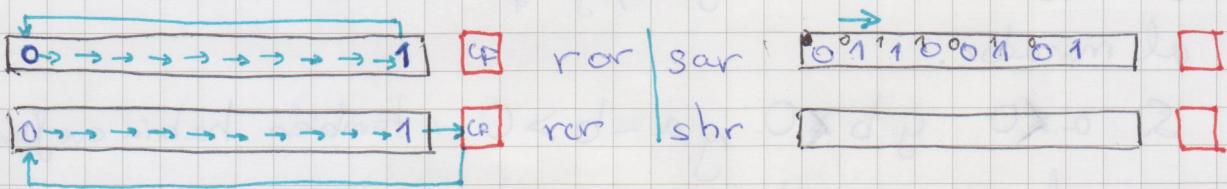
## Rotaciones

ror Dest Rotate Right

rol Dest Rotate Left

ror Dest Rotate with Carry Right

rcl Dest Rotate with Carry Left



## 2. Nivel máquina 2. Control

### Flags

CF Carry Flag. Tiene sentido en números sin signo.

ZF Zero Flag. Tiene sentido en ambos. Mira si un registro da 0.

SF Sign Flag: Dice si el neg. está en pos. o negativo.  
Para Ops con signo.

OF Overflow Flag: Si  $a > 0$  y  $b < 0$ , entonces  $a - b < 0$ .  
En este caso habría overflow, ya que se daña la vuelta  
al marcador.

Si  $a < 0$  y  $b > 0$ , ya que  $a - b > 0$ , también habría overflow  
por el mismo motivo.

Propiedad Al hacer  $a - b$ , si  $C_n \wedge C_{n-1} = 1$ , OF  
será 1.

$$\begin{array}{r} SF=0 \\ OF=1 \\ \hline 1000\ 0000 \rightarrow -128 \\ 1111\ 1110 +128 \\ \hline 0111\ 1111 -2 \end{array}$$

!

Las operaciones aritméticas afectan a los 4 flags mientras que las  
lógicas afectarán a SF y ZF.  
mov y lea no afectan a los flags.

### Compare y Test

cmp a,b realiza la resta  $b - a$ , salvo que no  
guarda el resultado en ningún registro ni variable, sino que  
sólo modifica los flags.

test a,b realiza el and lógico de a y b salvo que  
no guarda el resultado. Sólo cambia los flags.

## Seteando los flags

set CC ajusta el byte destino a 0/1 según el sufijo del código de condición

### Códigos de condición

-e if equal / zero	-g if greater	<u>seta</u> if above
-ne if not equal / zero	-ge if greater or equal	<u>setae</u> if above or equal
-s if signed	-l if less	-b if below
-ns if not signed	-le if less or equal	-be if below or equal

## Saltos condicionales

jmp salta al punto o etiqueta que se pone como parámetro  
j - es un salto condicional en base a códigos de condición.

COND	Números sin signo		Números con signo	
	Instrucción	Condición a comprobar	Instrucción	Condición a comprobar
=	je=jz sete=setz	ZF	je=jz sete=setz	ZF
≠	jne=jnz setne=setnz	~ZF	jne=jnz setne=setnz	~ZF
<	jb=jnae=jc setb=setnae=setc	CF	jl=jnge setl=setnge	SF^OF
≥	jae=jnb=jnc setae=setnb=setnc	~CF	jge=jnl setge=setnl	~(SF^OF)
>	ja=jnbe seta=setnbe	~CF • ~ZF ~(CF   ZF)	jg=jnle setg=setnle	~(SF^OF) • ~ZF ~((SF^OF)   ZF)
≤	jbe=jna setbe=setna	CF   ZF	jle=jng setle=setng	(SF^OF)   ZF

## Movimientos condicionales

mov CC hace un movimiento si se cumple una condición.

COND	Números sin signo		Números con signo	
	Instrucción	Condición a comprobar	Instrucción	Condición a comprobar
=	cmove=cmovz	ZF	cmove=cmovz	ZF
$\neq$	cmovne=cmovnz	$\sim$ ZF	cmovne=cmovnz	$\sim$ ZF
<	cmovb=cmovnae=cmovc	CF	cmovl=cmovnge	SF $\wedge$ OF
$\geq$	cmove=cmovnb=cmovnc	$\sim$ CF	cmovge=cmovnl	$\sim$ (SF $\wedge$ OF)
>	cmove=cmovnbe	$\sim$ CF $\cdot$ $\sim$ ZF	cmovg=cmovnle	$\sim$ (SF $\wedge$ OF) $\cdot$ $\sim$ ZF
		$\sim$ (CF $\mid$ ZF)		$\sim$ ((SF $\wedge$ OF) $\mid$ ZF)
$\leq$	cmovbe=cmovna	CF $\mid$ ZF	cmovle=cmovng	(SF $\wedge$ OF) $\mid$ ZF

## ! Instrucciones de salto / movimiento condicional EXTRA

Condición	Instrucción	Descripción	Indicadores
Overflow	jo seto	jump/set if overflow	OF
No overflow	jno setno	jump/set if not overflow	$\sim$ OF
$< 0$	js sets	jump/set if sign	SF
$\geq 0$	jns setns	jump/set if not sign	$\sim$ SF
Paridad par	jp=jpe setp=setpe	jump/set if parity / jump/set if parity even	PF (8 bits menos signif. del resultado contienen un n° par de unos)
Paridad impar	jnp=jpo setnp=setpo	jump/set if not parity / jump/set if parity odd	$\sim$ PF (8 bits menos signif. del resultado contienen un n° impar de unos)
Registro *CX = 0	jcxz, jecxz, jrcxz	jump if cx/ecx/rcx is zero	$\sim$ CX, $\sim$ ECX, $\sim$ RCX

Condición	Instrucción	Indicadores
Overflow	cmovo	OF
No overflow	cmovno	$\sim$ OF
$< 0$	cmovs	SF
$\geq 0$	cmovns	$\sim$ SF
Paridad par	cmovp=cmovpe	PF
Paridad impar	cmovnp=cmovpo	$\sim$ PF

## Secuencias condicionales.

If

if ( $<$ Condición $>$ )

$<$  Secuencia Then $>$

Repetirse

Así

$<$  Comprueba condición  
 $\sim$ Condición Continuar  
 $<$  Secuencia Then $>$

Continuar:  
 $<\dots>$

## If / Else

C | if (<condición>)  
     | <secuencia then>  
     | else  
     |    <secuencia else>

ASM

suele implementarse

<comprobar condición>

|<no-condición> else

<secuencia then>

|jmp continuar

else:

<secuencia else>

continuar!

...

## Formato goto

En C se permite usar goto para ir a una etiqueta que se llame en este. Da cierto parecido al código en ensamblador.

C | goto

1 | if (x > y) return x;  
 2 | else return y;

1 | if (x <= y) goto else;  
 2 | return x;  
 3 | else: return y;

## Operador condicional ?

C-Pseudo

1 | <Cond>? <secuencia then>;  
    | <secuencia else>

ASM-Pseudo

1 | <t> = <secuencia then>  
 2 | <resultado> = <secuencia else>  
 3 | Si (<conds>) resultado=t>

## Composición de condiciones

AND if (c1 && c2) return x;  
     | else return y;

<comprobar c1>, j<no cond> else  
   <comprobar c2>, j<no cond> else  
     | <secuencia then>, jmp fin  
   else: <secuencia else>  
   fin: <...>

OR if (c1 || c2) return x;  
     | else return y;

<comprobar c1>, j<cond> then  
   <comprobar c2>, j<no cond> else  
   then: <secuencia then>, jmp fin  
   else: <secuencia else>  
   fin: <...>

## 2. Nivel máquina

## 3. Procedimientos.

### La pila

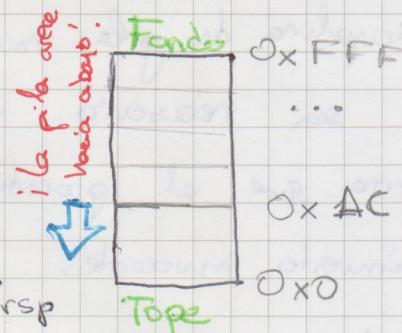
La pila es una región de memoria en forma de pila, que crece hacia abajo (posiciones más bajas).

%rsp Contiene el último elemento de la pila

push Introducir en la pila, <sup>Resta 8 a %rsp</sup>

pop Sacar de la pila, <sup>Suma 8 a %rsp</sup>

%rbp Marco de pila, se usaba en x86-32.



### Flujo de control.

Se usa la pila para soportar llamadas y retornos de procedimientos.

call etiqueta: Guarda la dirección de retorno en la pila y salta a etiqueta. La dirección de retorno es la siguiente linea a call.

ret Recupera la dirección de la pila y salta hacia allá.

Los datos se pasan usando los registros y la pila, en este orden

%rdi → %rsi → %rdx → %rcx → %r8 → %r9 → push <Datos>

Sólo se reserva espacio en pila cuando se necesita.

### Lenguajes basados en pila

Son lenguajes que soportan recursividad. Por ello el código debe ser reentrante, es decir, que debe tener varias instancias simultáneas de un mismo procedimiento. Pero esto se necesita un lugar para guardar el estado de cada instancia.

Cerro los argumentos, variables locales y un punto de retorno.

La disciplina de pila hace que el estado para un procedimiento de los sea necesario desde que se le llama hasta que retorna, de forma que el procedimiento invocado retorna dentro que el procedimiento invocante.

La pila se reserva en Marcos o estados para una instrucción del procedimiento.

### Marcos de pila

Contienen la información de retorno, y el almacenamiento local y el espacio temporal.

Para gestionarlo, primero se reserva el espacio al entrar en el procedimiento, con su código de inicio y añadiendo el push de call. Este se libera al retornar, con su código de finalización y añadiendo el pop de ret.

En Linux el marco contiene:

Lista de argumentos (a partir del 7º)

Variables locales (si no se pueden mantener en registros)

Contexto registros preservados

Y el marco de pila del invocante además incluye la dirección de retorno y los argumentos (a partir del 7º) para su llamada.

## Preservación de registros

Hay registros que se pueden usar para almacenar variables temporalmente. Según convención, hay dos tipos de registros:

Salva invocados: Estos registros tienen que guardar sus valores antes de llamar al procedimiento.

%rdi, %rsi, %rdx, %rbx,%r8,%r9, %rax, %rdo,%r11

Salva invocados: Estos registros tienen que guardar sus valores en el procedimiento que se ha llamado. Es decir, si estamos dentro y queremos usarlo, hay que guardarlo antes de modificar su valor. Al acabar el procedimiento y antes de volver, se saca de pila el valor del registro.

%r12, %r13,%r14, %r15, %rbx, %rbp

! %rsp es un salva invocado especial, el cual se restaura a su valor original al salir del procedimiento.

## Reversibilidad

Se maneja sin consideraciones especiales. Los marcos de pila implican que cada llamada a función tiene almacenamiento privado. (Variables locales y dev. retorno)

Las convenciones de pres. reg. preverían que una llamada a función corrompa los datos de otra a menos que se haya explicitamente.

La disciplina de pila sigue el patrón de llamadas/retornos. Es un buffer FIFO (Primeros que entrar, primero que sale).

Tipos Arrays

Se ubican contiguos en memoria, sea un array o una matriz.

Todos los punteros ocupan 8B (64 bits)

(la memoria ocupa  $\langle \text{num elementos} \rangle \cdot \text{size of}(\text{tipo})$ )

▷ Ejercicio ¿Cuál es el valor de...?

Referencia	Tipo	Valor
val[4]	int	3

val	int*	X
-----	------	---

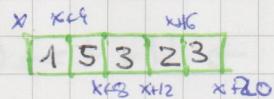
val + 1	int*	X + 4
---------	------	-------

&val[2]	int*	X + 8
---------	------	-------

val[5]	int	Indefinido
--------	-----	------------

*(&val + 8)	int	5
-------------	-----	---

val + i	int*	X + 4 · i
---------	------	-----------



$$\text{! } \text{val}[2] = \text{val} + 2$$

**! size\_t devuelve el tamaño de en long de 32 o 64 bits**

Arrays multidimensionales

Un array que contiene arrays, como una matriz.

A[i][j] es un array de C elementos, si cada elemento es de K bytes.

Si queremos llegar a un elemento, calcularemos  $A + (C \cdot k) \cdot i$

para acceder a su posición de memoria en la fila.

Si queremos llegar a un elemento, dado A[i][j][k], su posición

en memoria sería  $A + i \cdot (C \cdot k) + j \cdot k + k = A + (i \cdot C + j) \cdot k$

## Estructuras

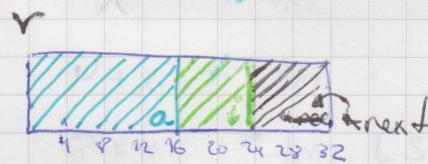
struct rec {

int a[4];  $\rightarrow 4 \times 4 = 16$

size\_t i;  $\rightarrow + 8$

struct rec \* next;  $\rightarrow + 8$

}



Para generar un puntero a un miembro de la estructura, calcularemos

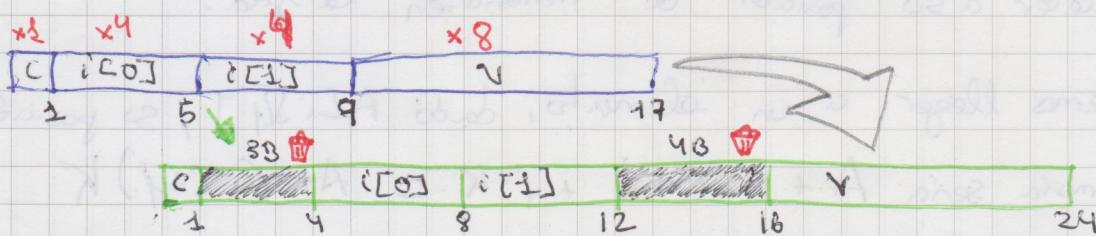
$r + 4 \times idx //$  Índice de elemento

```
C int get_ap (struct rec * r, size_t idx) {
    return &r->a[idx];
}
```

ASM  $\log (ordi,orsi,4)$ ,  
ret

## Alineamiento

La dirección debería ser múltiplo del tipo de datos primitivo.



char se alinea de 1 en 1 B      double, long / punteros se alinean de 8 en 8 B

short se alinea de 2 en 2 B

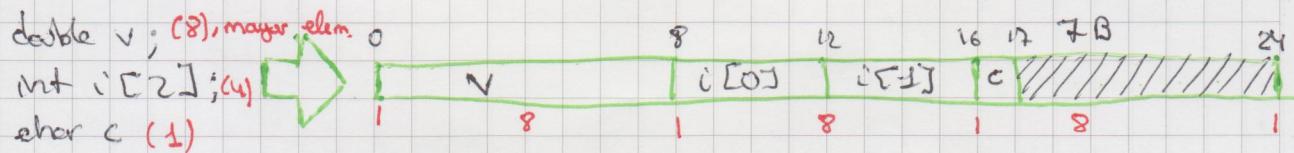
long double se alinea de 16 en 16 B.

int / float se alinea de 4 en 4 B

! los structs se alinean en base al elemento de mayor alcance.

### ▷ Ejemplo

struct S2 {  
 double v; (8), mayor elem.  
 int i[2]; (4)  
 char c (1)  
}; a[3]



Para acceder a un array, usaremos tam\_struct \* struct + index\_mayor.

Para ahorrar espacio, podríamos ir poniendo los mismos tipos en orden de espacio.

### ▷ Ejemplo

struct foo {  
 char a;  
 long b; (4)  
 float c;  
 char d[3];  
 int \*e;  
 short \*f;  
}; struct {

a b c d e f  
-----  
ccc ccc b b b b  
----  
fff fff fff fff

### struct bar {

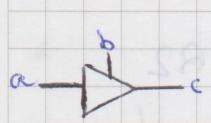
char a;  
char d[3];  
float c;  
long b;  
int \*e;  
short \*f;  
}; struct 2

## Uniones

Se reservan de acuerdo al elemento más grande, y sólo puede usarse un campo a la vez.

### 3. Unidad de control

#### Buffers triestado



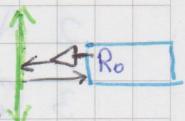
a	b	c
0	0	z
0	1	0
1	0	z
1	1	1

z: Alta impedancia (como si hubiera un cable cortado)

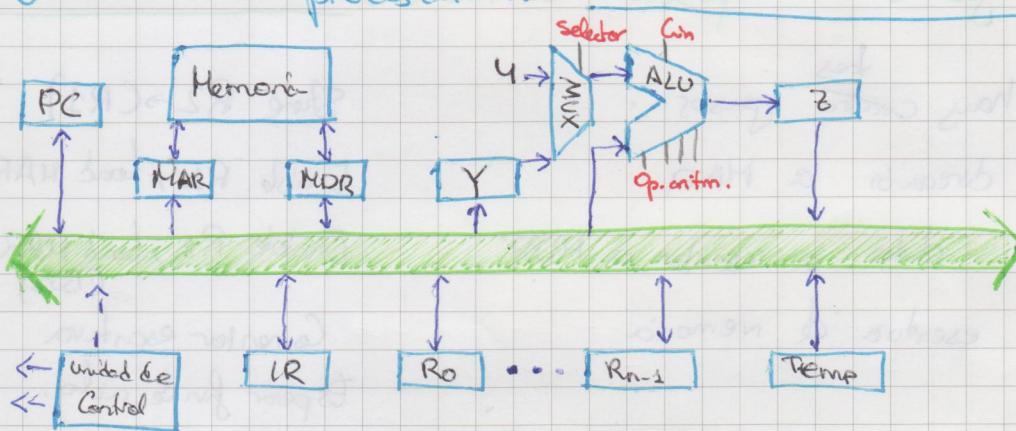
¿Los registros deberían tener buffers triestado?

Sí, ya que así se evitan cortocircuitos o sobreescrituras de bits.

Al menos, en la lectura del registro.



#### Unidad de procesamiento

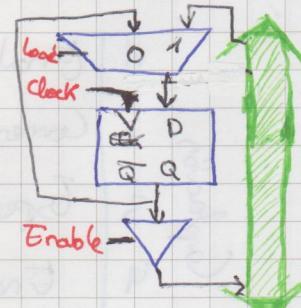


#### Transferencia entre registros

Cada registro usa dos señales de control:

Load Carga en paralelo

Enable Habilitación de salida



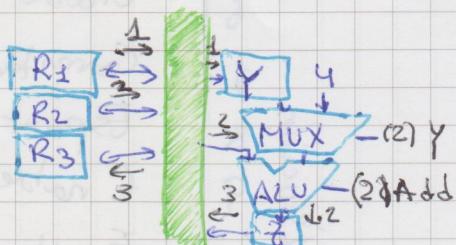
#### Operación aritmética o lógica a registros

Example  $R_3 \leftarrow R_1 + R_2$

1 Enable  $R_1$ , Load Y

2 Enable  $R_2$ , Select Y, Add, Load Z

3 Enable Z, Load  $R_3$



## Cargar posición de memoria a registro

Para ello hay <sup>cuatro</sup> tres pasos:

1 Transferir dirección al MAR

2 Activar lectura de memoria

3 Cargar el MDR desde memoria

4 Almacenar el dato en la MDR

Load (R2)  $\rightarrow$  R2

Enable R2, load MAR

Comenzar lectura

Esperar fin de ciclo, load MDR  
(Memoria)

Enable MDR hacia bus  
internos, load R2

## Almacenar registro en posición de memoria

Para hacerlo hay <sup>tres</sup> cuatro pasos:

Transferir dirección a MAR

Transferir dato a escribir a MDR

Activar escritura de memoria

Store R2  $\rightarrow$  (R1)

Enable R2, load MAR

Enable R2, load MDR  
(Bus)

Comenzar escritura  
Esperar fin de ciclo.

## D) Ejemplo Ejecución de una instrucción completa:

Add (R3)  $\rightarrow$  R1

- |                |                                                        |
|----------------|--------------------------------------------------------|
| Carga          | 1   Enable PC, Load MAR, Select Y, Sumar, Enable Z     |
|                | 2   Comenzar lectura, Enable Z, Load PC, Load Y        |
|                | 3   Esperar ciclo de memoria, Load MDR (Memoria)       |
|                | 4   Enable MDR (Bus), Load R                           |
| Decodificación | 5 > Decodificar instrucción                            |
|                | 6   Enable R3, Load MAR                                |
|                | 7   Comenzar lectura, Enable R <sub>s</sub> , Load Y   |
|                | 8   Esperar ciclo de memoria, Load MDR (Memoria)       |
| Ejecución      | 9   Enable MDR (Bus), Select Y, Sumar, Load Z          |
|                | 10   Enable Z, Load R <sub>s</sub> , Salir a captación |

► Ejemplo Ejecución de una instrucción de salto

## Tmp desplazamiento

Enable PC, Load MAP, Select 4, Swap, Enable Z

Correter lectura, enable Z, load PC, load Y

Especiar för de acto, Local HDR (Memória)

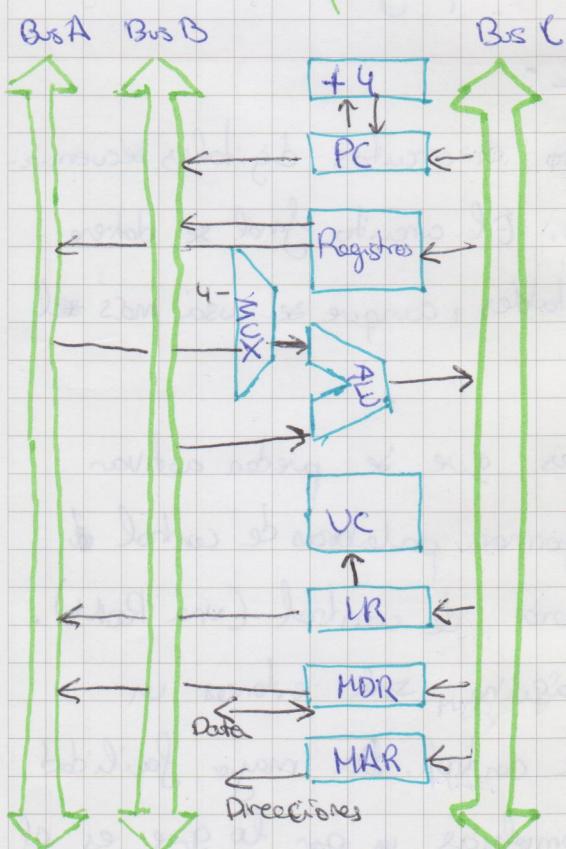
treble MDR ( Bus ), load ( R )

> Decodificar instrucciones

Treble Cmp Desplazamiento en 1R, Suner; load 2

Enable 2, Load PC, Set bit as capture.

## Unidad de procesamiento con buses múltiples



Es un banco de registros con tres puertos, lo que permite en el mismo ciclo poner los contenidos de dos registros en los buses A y B y cargar un registro desde el bus C.

La ALU no necesitará los registros Y y Z y hacer que uno de los operandos se vaya directamente a R a través del bus C, la constante +4

según siendo útil para manejar otras direcciones en instrucciones de movimiento múltiple.

**Ejemplo** Ejecución de una instrucción con 3 buses

$$R6 \leftarrow R4 + R5$$

**Descodificación** | Enable PC,  $R=3$ , Load MAR

Convertir lectura, Incrementar PC

Esperar fin de ciclo, Load MDR (Mem)

Enable MDR (Bus a B),  $R=3$ , Load IR

↓ Descodificar instrucción

**Ejecución** | Enable R4 a A, Enable RS a B, Select A, Sumar,

~~Enable R6~~ <sup>Load</sup> R6, Salir a captación

## Unidades de control cableadas y microprogrammadas

Hay dos formas de diseñar la UC:

**Cableada** Emplean métodos de diseño de circuitos digitales secuenciales a partir de diagramas de estados. El circuito final se obtiene conectando componentes básicos y bistables, aunque se usa más ~~se~~ el PLA

**Microprogrammado.** Todas las señales que se pueden activar simultáneamente se agrupan para formar palabras de control que se almacenan en una memoria de control (una ROM).

Una instrucción de lenguaje máquina se entonces microprograma en la memoria de control. Da mayor facilidad de diseño para instrucciones complejas y por lo que es el método estandar en la mayoría de los CISC

## Diseño de una UC cableada.

Se diseña mediante puestos lógicos y bisestables. Es tedioso y difícil de modificar por la complejidad de los circuitos, pero se ve ser más rápida que las UC microprogrammadas. Se usan PLAs para su implementación.

Sin embargo, las técnicas CAD resuelven las dificultades de diseño de lógica cableada. Esto, junto con tecnologías como RISC,

llevan a un auge de UC cableadas. Esto genera directamente las máscaras de fabricación de circuitos VLSI a partir de descripciones del comportamiento del circuito en lenguaje de alto nivel.

### Pasos

Definir una máquina de estados finitos

Desembalar en un lenguaje de alto nivel

Generar la tabla de verdad para la PLA

Minimizar la tabla de verdad

Diseñar físicamente la PLA partiendo de la tabla de verdad.

## UC microprogrammada.

La idea básica es emplear una memoria de control para almacenar las señales de control de los períodos de cada instrucción.

Microinstrucción Cada palabra de la memoria de control

Microprograma Conjunto de microinstrucciones cuyas señales de control constituyen el cronograma de una instrucción máquina.

### + Ventajas

Simplicidad conceptual, Modificación sencilla

Inclusión de instrucciones complejas, Varias tipos de instrucciones

### - Desventaja -

+ Lento de ejecución

## Formato de microinstrucciones

las señales de control del datapath se agrupan en campos.

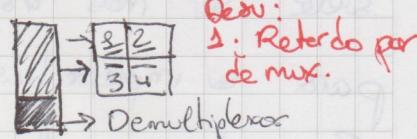
No codificado (un bit) por señal.

1 2 3 4

Codificado Se puede acortar codificando Recorridos  
Campos dific. 1 → 1 2  
3 → 3 4

Desv: Inclusión  
de decodificadores

Solapando Si los señales son excluyentes se gasta un bit más para ver a qué campo pertenece.



2. Es incompatible

una operación si está en un punto campo

solapado y se quiere usar la operación del mismo sitio.

### Microprogramación

horizontal

- + Ninguna o poca codificación
- + Alto grado de paralelismo en las microoperaciones a ejecutar
- Microinstrucciones largas

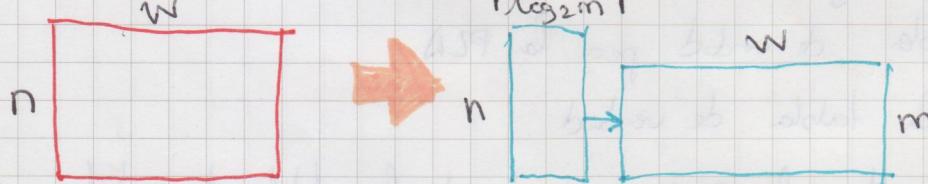
### Microprogramación vertical

- + Microinstrucciones cortas

- Mucha codificación
- Escasa capacidad para expresar paralelismo.

## Nano programación

Objetivo: reducir el tamaño de la memoria de control.



Ahorro en bits  $n \cdot w - (n \log_2 m + mw)$

## Diseño horizontal vs vertical

La microinstrucción vertical se basa en  $n$  campos de  $m$  bits.

La microinstrucción horizontal tiene  $K$  señales de control.

$$\text{Máximo no de } = K + n \cdot (2^m - 1)$$

Señales a usar

-

-

-

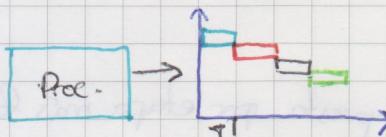
-

-

## 4. Segmentación de cierre

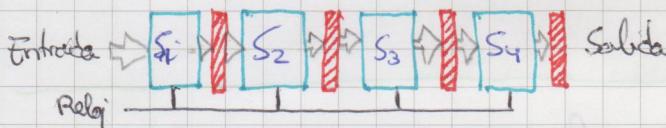
### Segmentación

Sin segmentar



Para poder mejorar el rendimiento sin mucha tecnología nueva podemos usar la segmentación de cierre.

Si tenemos  $n$  procesadores, y los unimos con una pipeline, podemos hacer la tarea en  $n$  etapas. Pero si una etapa tarda mucho, lo mejor es subdividir la tarea, en  $n$  etapas de duración similar, etapa y así enter que las fases se solapen.



Suelen tener estas etapas

IF Entrada de instrucción

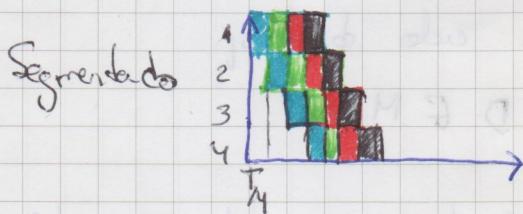
ID Decodificación de instrucción

EX Ejecución

MEM Memoria (lectura)

WB Escritura

Todo esto hace una mejora en la velocidad.



$$N^{\circ} \text{ et. sg} = N^{\circ} \text{ inst} + N^{\circ} \text{ etapas} + 1$$

$$\rightarrow \text{Aceleración} = \frac{N^{\circ} \text{ inst} \cdot \text{Tiempo}}{\left( \frac{N^{\circ} \text{ etapas}}{\text{Upr. lento}} \right) \left( \frac{(N^{\circ} \text{ etapas} + 1 + N^{\circ} \text{ inst}) \cdot \text{Tiempo}}{N^{\circ} \text{ etapas}} \right)}$$

$$= \frac{N^{\circ} \text{ etapas} \cdot N^{\circ} \text{ inst} \cdot \text{Tiempo}}{(N^{\circ} \text{ etapas} + N^{\circ} \text{ inst} + 1) \cdot \text{Tiempo}}$$

La aceleración ideal coincide con el  $N^{\circ}$  de etapas de segmentación.

$$\text{Ganancia} = \frac{\text{Tiempo segm}}{\text{Tiempo s.sgm.}}$$

Q d) Por qué disminuye la aceleración? \_\_\_\_\_

Coste de la segmentación

Duración del ciclo de reloj impuesto por etapa más lenta ] Esto puede hacer que  $T_c > T/n$

Riesgos (charcos) que bloquean el avance de instrucciones

## Riesgos

Problemas que pueden alterar el resultado por la lógica del programa, o bloquean el avance de las instrucciones del programa.

### Riesgos estructurales

Suceden por el conflicto al emplear los recursos, ya que dos instrucciones necesitan el mismo recurso.

► Supongamos que necesitamos tenemos un mismo circuito para IF y M<sub>1</sub> y vamos el siguiente programa:

lw r4,r0(r5) F D E M W

and r7,r2,r5 F D E M W

or r8,r6,r2 F D E M W

add r9,r2,r2 F D E M W

- En ese caso, las dos instrucciones podrían bloquearse, ya que no se pueden gestionar dos accesos a la vez.

Solución Penalizar un acceso con un ciclo de reloj

add r9,r2,r2

- F D E M W

► Si hubiera un fallo de caché al captar una instrucción, podemos penalizar los fetch mientras no se corrija el error, a costa de perder velocidad. Además, para evitar fallos de caché se hace uso de una cola de instrucciones, para captar instrucciones antes de ser necesarias, y almacenarlas ahí. Si hubiera un fallo, se puede seguir tirando de la cola.

## Riesgos de datos

Son riesgos de acceso de datos cuyo valor autorizado depende de la ejecución de instrucciones precedentes.

Sub  $r2, r3, r2$  FDE MW

and  $r7, r2, r5$  FDE MW

or  $r8, r6, r2$  FDE MW

add  $r9, r2, r2$  FDE MW

Solución Retrasar la ejecución y/o decodificación.

sub	F D E M W
and	F D - - - G M W
or	F - - - - D G M W
add	F - - - - - + D E M W

Otra opción sería compilar el hardware de la UC de forma que la salida de la ALU se conecte directamente a la parte de decodificación. (register forwarding)

Los compiladores también pueden resolverlo añadiendo instrucciones de no operando (nop)

sub	F D E M W
nop	F O E M W
nop	F D E M W
nop	F D F M W
and	F D E M W

## Riesgos de control

Suceden por adelantar la segmentación de instrucciones en una de salto.

jmp / br(1) F O E M W

and

F D E + -

sub

F D - -

or

F - -

$\rightarrow$  C3: add

F O B M W

Así perdemos 3 ciclos de reloj,

ya que hasta la fase de escritura de memoria, no se realiza el salto.

Para entregar se trata de predecir el salto lo antes posible para perder el mínimo de ciclos posible.



## Degradación de prestaciones debida a saltos.

b N° de ciclos desperdiciados por saltar

P<sub>b</sub> Probabilidad de saltar

CPI Ciclos por instrucción

P<sub>f</sub> Probabilidad real de producir salto

p<sub>e</sub> Probabilidad efectiva: P<sub>b</sub> · P<sub>f</sub>

$$F_b = \frac{1}{1 + p_e b}$$

La degradación  
crece al  
aumentar p<sub>e</sub>  
o b

$$(CPI = (1 - p_e) + p_e [P_f (1 + b) + (1 - P_f)]) = 1 + p_e b = 1 + p_e b$$

## Salto retardado

En vez de desperdiciar las etapas posteriores a la de Salto, las instrucciones <sup>ya puestas</sup> se ejecutan y después se salta. Lo que hace el compilador es buscar instrucciones anteriores lógicamente al salto.

Ej.

Se podrían añadir instrucciones nsp antes del salto o,

si el salto es incondicional, poner instrucciones add del salto.

br	F DEMW
add	F D FMW
or	F D EMW
and	F D GMW

(20 más) (se salta)

add F D GMW

## Salto condic.

Un salto de ese tipo ejecuta las instrucciones sólo si el salto se produce, y las ignora si no. Así, el destino de un salto condicional si puede colocarse tras el salto.

## Predicción de saltos

Intentar predecir si una instrucción de salto concreta dará una respuesta de salto producido (o no producido)

Esta predicción puede ser

Estatística se forma la misma predicción.

Dinámica se ve el historial de predicciones para formar una predicción.

Esta predicción puede tener una máquina de estados para saber si saltar o no. (ver apuntes AC)

## 6. Memoria

La memoria se clasifica dependiendo del tiempo de acceso. Puede ser:

RAM De acceso aleatorio: Tiempo de acceso independiente de la pos. memoria

De acceso secuencial Tiempo de acceso dependiente de la pos. memoria

DASD De acceso secuencial/directo Como HDD / SSD

Ancho de banda Número de palabras a las que puede acceder por unidad de tiempo Ej.: 1 GB/s

El problema con ello es que el ancho de banda crece a menor ritmo que la velocidad y prestaciones de la CPU.

Para aumentar el ancho de banda podemos:

User tecnología de alta velocidad

Organizar la memoria jerárquicamente

Incrementar el ancho de memoria

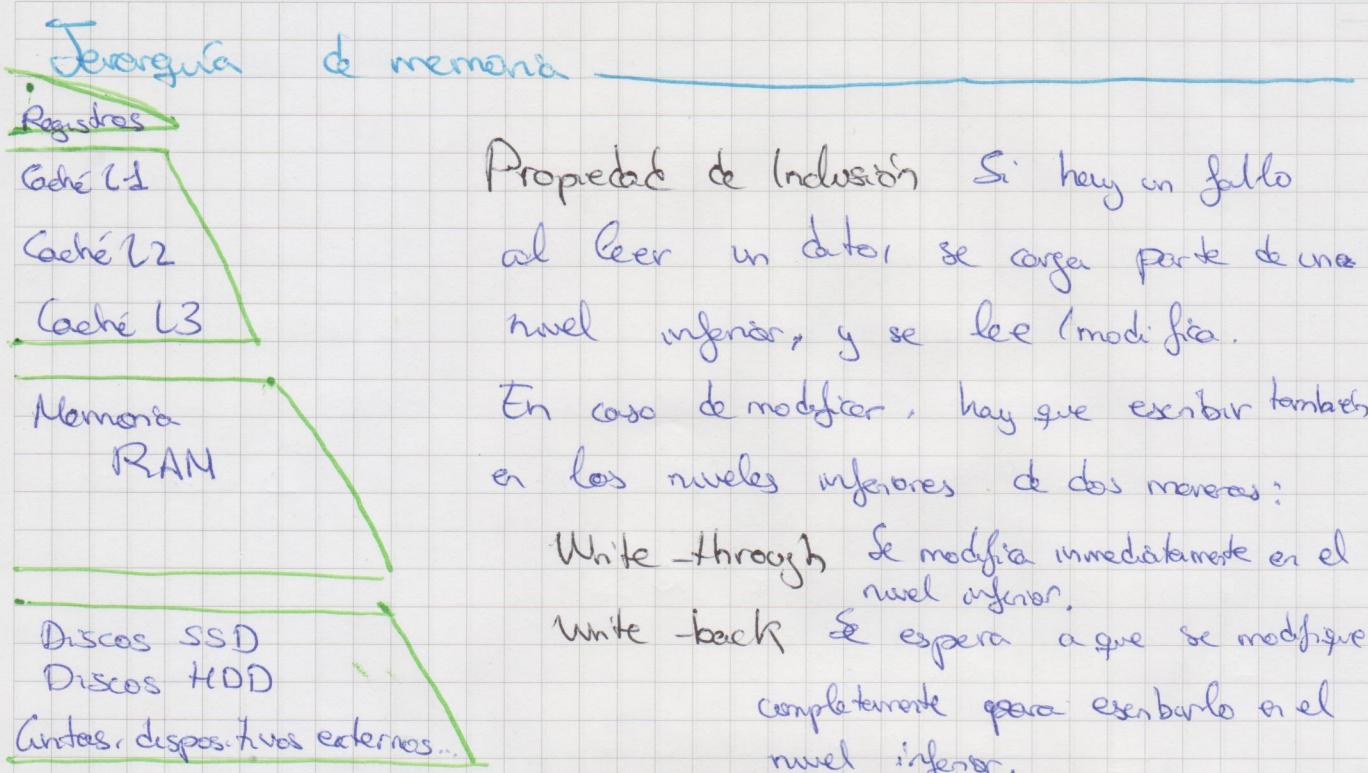
### Localidad de las referencias

Regla 90/10 Una regla El 90% del código depende del 10% de los datos

Localidad espacial Si se referencia un elemento, los elementos cercanos podrían volver referenciarse pronto.

Localidad temporal Si se referencia un elemento, se volverá a referenciar pronto

Localidad secuencial Las direcciones de memoria suelen estar contiguas



## Modelo de evaluación de la jerarquía

Tasa de aciertos: Es el porcentaje de información buscada en un acceso de memoria en el que está presente el nivel  $i$ .

En una jerarquía con  $n$  niveles, donde  $A_0=0$  y  $A_n=1$ ,

Así depende de

Capacidad del nivel  $i$

Granularidad de transferencia de información  $V$

Estrategia de administración de memoria

$$\text{Tasa de fallos} = (1 - \text{Tasa aciertos})$$

Frecuencia de accesos: Probabilidad de acceder con éxito en el nivel  $i$  y que no se encuentre en los niveles anteriores.  
(la suma de todos esos frecuencias debe dar 1 (100%))

Es acumulativa.

## ▷ Ejemplo

$$L_1 A_1 0'9$$

$$L_2 A_2 0'99$$

$$L_3 A_3 0'99999$$

$$\text{RAM} A_4 0'999999$$

$$\text{Disco } A_5 1$$

$$a_1 = 0'9$$

$$a_2 = A_2 - A_1 = 0'99$$

$$a_3 = A_3 - A_2 = 0'0099$$

$$a_4 = A_4 - A_3 = 0'000099$$

$$a_5 = A_5 - A_4 = 0'000001$$

$$\sum_{i=1}^5 a_i = 0'9 + 0'09 + 0'0099 +$$

$$0'000099 + 0'000001$$

$$= 0'9999 + 0'0001 = \frac{1}{2}$$

Los objetivos al diseñar la jerarquía son (1) Obtener un rendimiento cercano al de la memoria  $M_1$ , y (2) Obtener un coste por bit cercano al de la memoria  $M_n$  (la más barata)

1 Rendimiento Se puede medir el tiempo medio de acceso

Tiempo de la jerarquía para cada referencia a memoria.

Tiempo de acceso efectivo a nivel  $i$

$$T_i = \sum_{j=1}^i t_j \rightarrow \text{Tiempo de acceso medio del nivel } i$$

Tiempo medio

$$\bar{T} = \sum_{i=1}^n a_i T_i$$

$$\bar{T} = \sum_{i=1}^n F_{i-1} t_i$$

## ▷ Ejemplo

$$T_1 = t_1 = 1 \text{ ns}$$

$$T_2 = t_1 + t_2 = 1 + 2 = 3 \text{ ns}$$

$$T_3 = t_1 + t_2 + t_3 = 1 + 2 + 4 = 7 \text{ ns}$$

$$T_4 = t_1 + t_2 + t_3 + t_4 = 1 + 2 + 4 + 20 = 27 \text{ ns}$$

$$T_5 = t_1 + \dots + t_5 = 1 + 2 + 4 + 20 + 1000000 = 1000027 \text{ ns}$$

$$\begin{aligned} \bar{T} &= 0'9 \cdot 1 + 0'09 \cdot 3 + 0'0099 \cdot 7 \\ &+ 0'000099 \cdot 27 + 0'000001 \cdot 1000027 \\ &= 0'9 + 0'27 + 0'00693 + \\ &0'002673 + 1'000027 \\ &= 2'242 \text{ ns} \end{aligned}$$

## Memoria de acceso aleatorio

Tipos

Memoria de acceso solo lectura (ROM)

Esta memoria se hacen en tamaños grandes y pueden ser no borrables (ROM), programables pero no borrables (PROM), programables y borrables por ultravioleta (EPROM) o programables y borrables eléctricamente (EEPROM).

También puede ser programable por bloques (FLASH)

Memoria de acceso aleatorio (RAM)

Dentro de la RAM está la SRAM y la DRAM

SRAM : RAM estática. La memoria se almacena mientras circula corriente. Se usa un par de inversores. La lectura mantiene el bit en su sitio.

DRAM : RAM dinámica. A través de un condensador, se carga el dato. La lectura es destructiva, es decir, en cuanto se lee no hay nada en ese sitio. Además, tras un tiempo tiene que borrarse y volver a cargar. Así que hay que refresh.

Los circuitos de memoria DRAM se compone de una matriz

con filas y columnas. Haciendo que de una palabra se maneje la posición con un multiplexor y así sacar el dato.

