

RPG: Clasificación R/M

■ Para máquinas RPG

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

■ ISA

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

■ RISC

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

■ CISC

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

Modos de Direccionamiento

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

■ Inmediato (ej: \$0, \$variable)

- El número es el valor del operando

■ Registro (ej: %eax, %ebx...)

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

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

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

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

ej: modos IA-32

Código fuente ASM:

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

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

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

a veces puede ser ventajoso
+ instrucciones -tamaño

inmediato ≠ directo

resto modos indirectos

Desensamblado del ejecutable:

Disassembly of section .text:

```
08048074 <_start>:
8048074: b8 00 00 00 00      mov     $0x0, %eax
8048079: 31 db              xor     %ebx, %ebx
804807b: 43                inc     %ebx
804807c: b9 98 90 04 08     mov     $0x8049098, %ecx
8048081: 8b 15 98 90 04 08  mov     0x8049098, %edx
8048087: 8b 11              mov     (%ecx), %edx
8048089: 03 14 99          add     (%ecx,%ebx,4), %edx
804808c: 03 14 9d 98 90 04 08 add     0x8049098(,%ebx,4), %edx
8048093: 8b 55 f8          mov     -0x8(%ebp), %edx
```

Rendimiento

■ Reloj del procesador

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

■ Ecuación básica de rendimiento

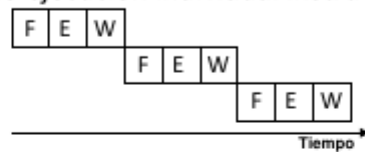
- T tiempo para ejecutar programa benchmark
- **N instrucciones** (recuento dinámico bucles/subrutinas)
 - N no necesariamente igual a #instr. progr. objeto.
- **S ciclos/instr.** ("pasos básicos" de media)

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

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

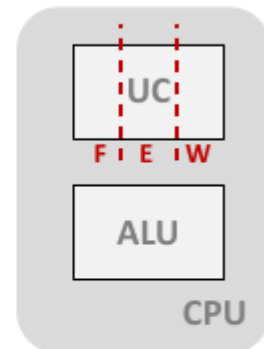
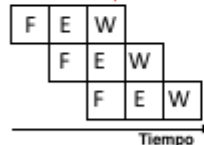
- $N \times S$ es suponiendo ejecución individual instrucciones

ADD R1,R2, R3
MUL R4,R5, R5
SUB R3,R5, R5



- Pero las distintas etapas hacen tareas distintas

- UC puede tener **circuitería separada para cada etapa**:
 - Fetch: captación
 - Exec: ejecución
 - Write: actualización registro

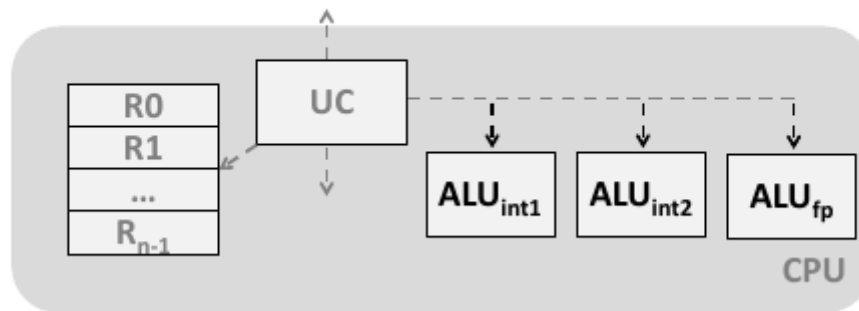


- **una vez lleno el cauce**, valor efectivo $S=1$ ciclo/instr

- dependencias datos (ej: MUL-SUB arriba, dependencia R5)
- competición recursos (ej: almacenar resultado M/fetch instrucción+2)
- saltos (pipeline flush)
- $S \geq 1$, $S \approx 1$

Funcionamiento superescalar (que $S < 1$)

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



Otras formas de reducir T

■ Velocidad del reloj $(R \uparrow, S/R)$

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

■ Repertorio RISC/CISC $(N \cdot S)$

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

Perspectiva histórica

■ 2ª Guerra Mundial

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

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

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



Perspectiva histórica

■ 2ª Generación (55-65): transistores

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



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

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



Perspectiva histórica

■ 4ª Generación (75-...): VLSI

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



■ Actualidad

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



62

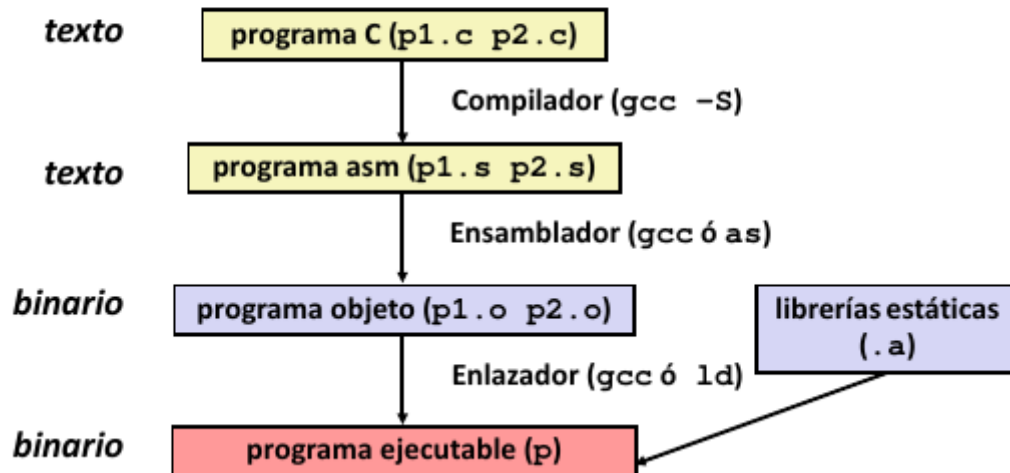
tema2.1

Evolución Intel x86: Hitos significativos

<i>Nombre</i>	<i>Fecha</i>	<i>Transistores</i>	<i>MHz</i>
■ 8086	1978	29K	5-10
▪ Primer procesador Intel 16-bit. Base para el IBM PC & MS-DOS			
▪ Espacio direccionamiento 1MB			
■ 386	1985	275K	16-33
▪ Primer procesador Intel 32-bit de la familia (x86 luego llamada) IA32			
▪ Añadió "direccionamiento plano" [†] , capaz de arrancar Unix			
■ Pentium 4E	2004	125M	2800-3800
▪ 1 ^{er} proc. Intel 64-bit de la familia (x86, llamada x86-64, EM64t) Intel 64			
■ Core 2	2006	291M	1060-3500
▪ Primer procesador Intel multi-core			
■ Core i7	2008	731M	1700-3900
▪ Cuatro cores, hyperthreading (2 vías)			

Convertir C en Código Objeto

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



Representación Datos C, IA32, x86-64

■ Tamaño de Objetos C (en Bytes)

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

— o cualquier otro puntero

Registros enteros x86-64

%rax	%eax	%ax	%al	%r8	%r8d	%r8w	%r8b
%rbx	%ebx			%r9	%r9d		
%rcx	%ecx			%r10	%r10d		
%rdx	%edx			%r11	%r11d		
%rsi	%esi	%si	%sil	%r12	%r12d		
%rdi	%edi			%r13	%r13d		
%rsp	%esp			%r14	%r14d		
%rbp	%ebp			%r15	%r15d		

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



RSP PUNTERO DE PILA NO SE GUARDAN DATOS

Mover Datos

■ Mover Datos

`movq Source, Dest†`

■ Tipo de Operandos

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

%rax
%rcx
%rdx
%rbx
%rsi
%rdi
%rsp
%rbp
%rN

[†] Los tres operandos inmediatos tienen limitaciones de tamaño

Combinaciones de Operandos `movq`

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

Ver resto instrucciones transferencia (incluyendo pila) en el libro

No se puede transferir Mem-Mem con sólo una instrucción

Modos Direccionamiento a memoria sencillos

■ Normal[†] (R) Mem[Reg[R]]

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

```
movq (%rcx), %rax
```

■ Desplazamiento D(R) Mem[Reg[R]+D]

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

```
movq 8(%rbp), %rdx
```

Modos Direcccionamiento a memoria completos

■ Forma más general

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

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

■ Casos Especiales

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

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

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

TEMA 2.2

Consultando Códigos de Condición

■ Instrucciones SetCC Dest

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

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

[†] "CC" = "condition code"

Salto

■ Instrucciones jCC

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

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

Traducción en General Expresión Condicional (usando saltos)

Código C

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

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

Versión Goto

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

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

Malos Casos para Movimientos Condicionales

- Recordar que se calculan ambos valores

Cálculos costosos

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

- Sólo tiene sentido cuando son cálculos muy sencillos

Cálculos arriesgados

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

- Pueden tener efectos no deseables

Cálculos con efectos colaterales

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

- No deberían tener efectos colaterales

Traducción en General de “Do-While”

Código C

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



Versión Goto

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

- Body: {
 Sentencia₁;
 Sentencia₂;
 ...
 Sentencia_n;
}

Traducción en General de “While” (#1)

Código C

```
while (Test)  
    Body
```



Versión Goto

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

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

Traducción en General de “While” (#2)

Versión While

```
while (Test)  
    Body
```



Versión Do-While

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



Versión Goto

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

- Traducción tipo “copia-test”
 - Conversión a “do-while”
- Usada con -O1[†]

Forma del bucle "For"

Forma General

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

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

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

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

"init" = inici
"test" = comp
"update" = actu

Resumen

■ Control C

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

■ Control Ensamblador

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

■ Técnicas estándar

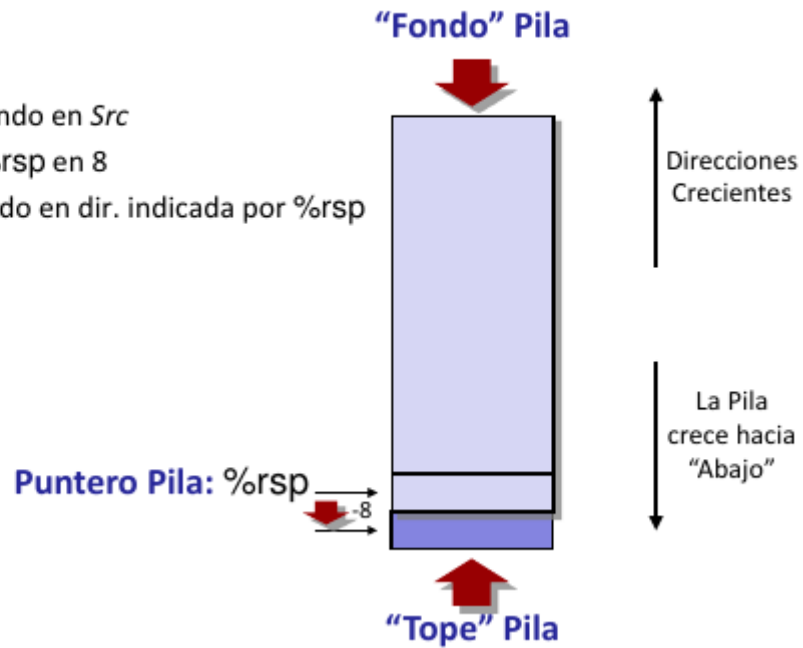
- Bucles convertidos a forma do-while (ó salta-en medio ó copia-test)
- Sentencias switch grandes usan tablas de saltos
- Sentencias switch poco densas → árboles decisión (if-elseif-elseif-else)

TEMA 2.3

Pila x86-64: Push[†]

■ pushq *Src*

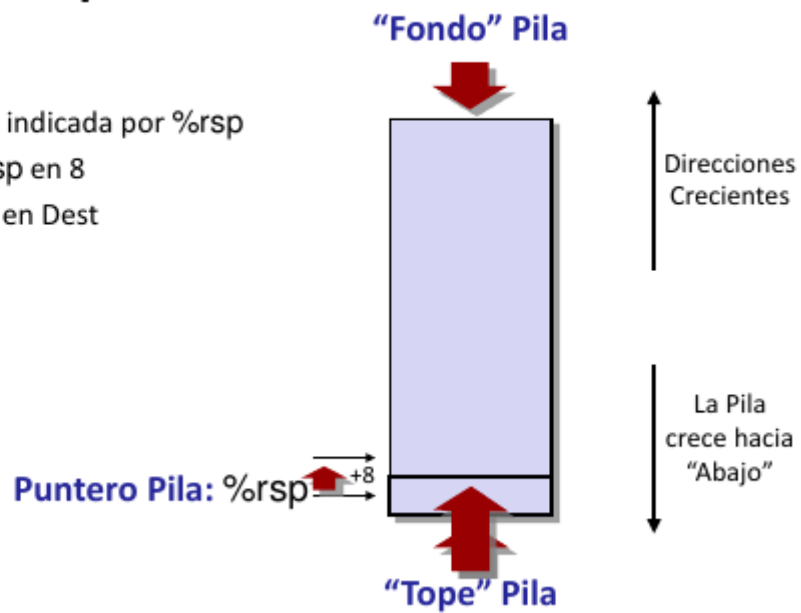
- Capta el operando en *Src*
- Decrementa `%rsp` en 8
- Escribe operando en dir. indicada por `%rsp`



Pila x86-64: Pop[†]

■ popq *Dest*

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



Flujo de Control en Procedimientos

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

Lenguajes basados en pila[†]

- **Lenguajes que soportan recursividad**
 - P.ej., C, Pascal, Java
 - El código debe ser “*Reentrante*”
 - Múltiples instanciaciones[‡] simultáneas de un mismo procedimiento
 - Se necesita algún lugar para guardar el estado de cada instancia
 - Argumentos
 - Variables locales
 - Puntero (dirección) de retorno
- **Disciplina de pila**
 - Estado para un procedimiento dado, necesario por tiempo limitado
 - Desde que se le llama hasta que retorna
 - El invocado[‡] retorna antes de que lo haga el invocante[‡]
- **La pila se reserva en *Marcos*[‡]**
 - estado para una sola instancia de procedimiento

[†] “block structured” en terminología Intel

[‡] “callee/caller” en inglés

[‡] “caller/callee” en español

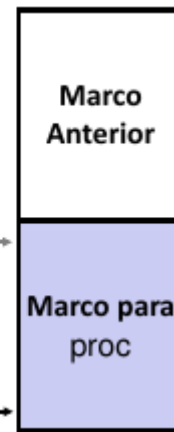
Marcos de Pila

■ Contenido

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

Puntero de Marco: %rbp[‡]
(Opcional)

Puntero de Pila: %rsp



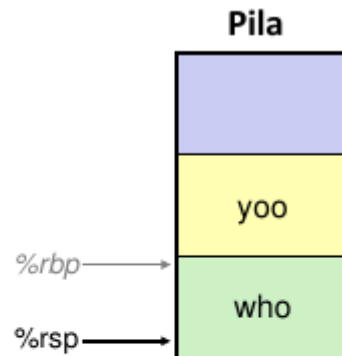
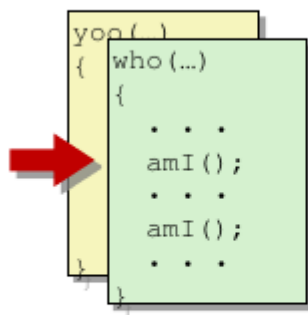
“Tope” Pila

■ Gestión

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

[‡] si se usa `-fno-omit-frame-pointer`
[†] No es obligatorio

Ejemplo



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

■ %rax	Val.retorno S-Invocante	<div><div></div><div>%rax</div></div>
■ Valor de retorno		
■ También salva-invocante		
■ Puede ser modificado [†] por el proc.		
■ %rdi, ..., %r9	Argumentos S-Invocante	<div><div></div><div>%rdi</div></div>
■ Argumentos [†]		
■ También salva-invocante		
■ Pueden ser modificados [†] por proc.		
■ %r10, %r11	Temporales S-Invocante	<div><div></div><div>%rsi</div></div>
■ Salva-invocante		
■ Pueden ser modificados [†] por proc.		
		<div><div></div><div>%rdx</div></div>
		<div><div></div><div>%rcx</div></div>
		<div><div></div><div>%r8</div></div>
		<div><div></div><div>%r9</div></div>
		<div><div></div><div>%r10</div></div>
		<div><div></div><div>%r11</div></div>

Uso de Registros en Linux x86-64 #2

■ %rbx, %r12 - %r15		<div><div></div><div>%r12</div></div>
■ Salva-invocado		
■ Invocado debe preservar y restaurar		
■ %rbp	Temporales S-Invocado	<div><div></div><div>%r13</div></div>
■ Salva-invocado		
■ Invocado debe preservar y restaurar		
■ Puede que se use como marco pila		
■ Puede usarse intermezcladamente [†]		
■ %rsp	Especiales	<div><div></div><div>%r14</div></div>
■ Forma especial de salva-invocado		
■ Restaurado a su valor original a la salida del procedimiento		
		<div><div></div><div>%r15</div></div>
		<div><div></div><div>%rbx</div></div>
		<div><div></div><div>%rbp</div></div>
		<div><div></div><div>%rsp</div></div>

Tipos de Datos Básicos

■ Enteros

- Almacenados y manipulados en registros (enteros) propósito general
- Con/sin signo depende de las instrucciones usadas[†]

Intel	ASM [‡]	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	d	4	[unsigned] int
quad word	q	8	[unsigned] long int (x86-64)

■ Punto Flotante

- Almacenados y manipulados en registros punto flotante

Intel	ASM	Bytes	C
Single	s	4	float
Double	d	8	double
Extended	t	10/12/16	long double

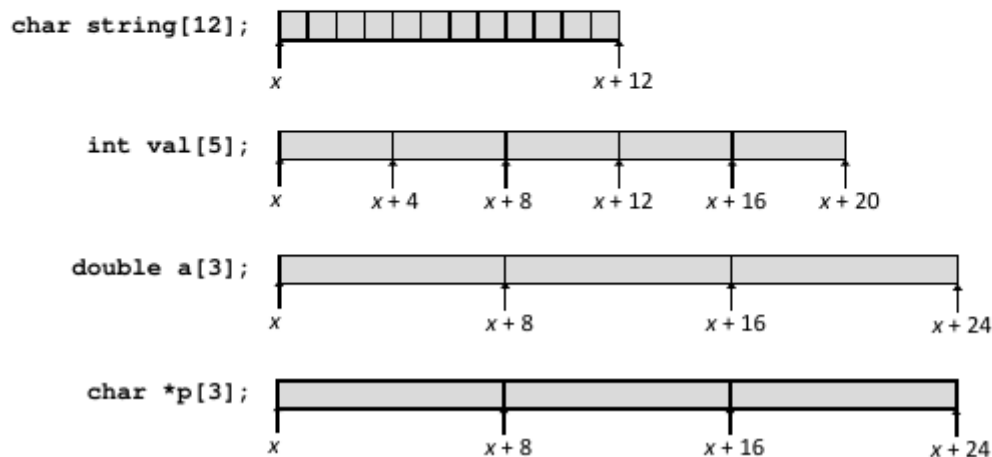
[‡] sufijos en sintaxis AT&T Lin
[†] y del tipo datos indicado en
de los flags ó *condition code

Ubicación[†] de Arrays

■ Principio Básico

$T \ A[L];$

- Array de tipo T y longitud L
- Reservada[†] región contigua en memoria de $L * \text{sizeof}(T)$ bytes

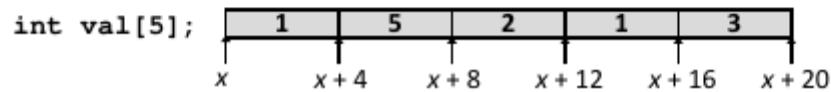


Acceso a Arrays

■ Principio Básico

T $A[L]$;

- Array de tipo T y longitud L
- El identificador A (Tipo T^*) puede usarse como puntero al elemento 0



■ Referencia[†]

Tipo

Valor

`val[4]` `int`
`val` `int *`
`val+1` `int *`
`&val[2]` `int *`
`val[5]` `int`
`*(val+1)` `int`
`val + i` `int *`



[†] otros autores usan "(de)referenciación" en sentido mucho más estricto, para indicar el tipo puntero, o la

```
val[4]  1
val x
val+1  x+4
&val[2] x+8
val[5]  ~~~
*(val+1) 5
val+1  x+4*i
```

Arrays Multidimensionales (Anidados)

■ Declaración

T $A[R][C]$;

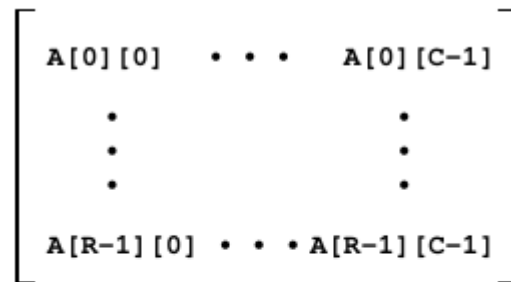
- Array 2D de (elems. de) tipo T
- R filas (rows), C columnas
- Elems. tipo T requieren K bytes

■ Tamaño Array

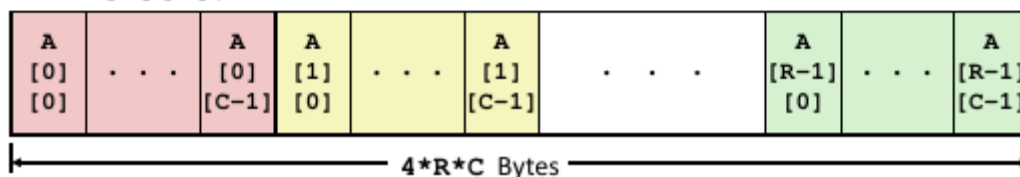
- $R * C * K$ bytes

■ Disposición

- Almacenamiento por filas (row-major-order)[†]



`int A[R][C];`



Acceso a Matriz 16 X 16

■ Elementos del Array

- `int A[16][16];`
- Dirección $A + i * (C * K) + j * K$
- $C = 16, K = 4$

```
/* Get element A[i][j] */  
int fix_ele(fix_matrix A, size_t i, size_t j)  
{  
    return a[i][j];  
}
```

```
# A en %rdi, i en %rsi, j en %rdx  
salq    $6, %rsi          # 64*i  
addq    %rsi, %rdi         # A + 64*i  
movl    (%rdi,%rdx,4), %eax # M[A + 64*i + 4*j]  
ret
```

Principios de Alineamiento

■ Datos Alineados

- El tipo de datos primitivo requiere K bytes
- La dirección debe ser múltiplo de K
- Requisito en algunas máquinas; recomendado en x86-64

■ Motivación para Alinear los Datos

- A la memoria se accede (físicamente) en trozos (alineados) de 4 ú 8 bytes (dependiendo del sistema)
 - Ineficiente cargar o almacenar dato que cruza frontera quad word
 - Mem. virtual muy delicada cuando un dato se extiende a 2 páginas

■ Compilador

- Inserta huecos en estructura para asegurar correcto alineamiento campos

Resumen de Tipos Compuestos en C

- **Arrays**
 - Reserva de memoria contigua para almacenar elementos
 - Se usa aritmética de indexación para localizar elementos individuales
 - Puntero al primer elemento
 - Sin chequeo de límites
- **Estructuras**
 - Reserva de una sola región de memoria, campos van en el orden declarado
 - Se accede usando desplazamientos determinados por el compilador
 - Puede requerir relleno interno y externo para cumplir con el alineamiento
- **Combinaciones**
 - Se pueden anidar representación estructura y array arbitrariamente
 - Relleno externo estructuras garantiza alineamiento en arrays de structs
- **Uniones**
 - Declaraciones superpuestas
 - Forma de soslayar el sistema de promoción de tipos en C

TEMA 3

Unidad de control

- **La unidad de control interpreta y controla la ejecución de las instrucciones leídas de la memoria principal, en dos fases:**
 - secuenciamiento de las instrucciones
 - La UC lee de MP la instrucción apuntada por PC, $IR \leftarrow M[PC]^*$
 - determina la dirección de la instrucción siguiente y la carga en PC*
 - ejecución/interpretación de las instrucción en IR
 - La UC reconoce el tipo de instrucción,
 - manda las señales necesarias para tomar los operandos necesarios y dirigirlos a las unidades funcionales adecuadas de la unidad de proceso,
 - manda las señales necesarias para realizar la operación,
 - manda las señales necesarias para enviar los resultados a su destino.

*Estos dos pasos pueden repetirse si la instrucción consta de varias palabras

Ej. de ejecución Add A, R0*

- $M[POS_A] + R0 \rightarrow R0$
 - Ensamblador traduce p.ej: $POS_A=100$
 - Valor anterior R0 perdido, el de POS_A se conserva
 - Arquitectura R/M

■ Pasos básicos de la UC


- PC apunta a posición donde se almacena instrucción
- **Captación:** $MAR \leftarrow PC, \text{Read}, PC++, T_{acc}, MDR \leftarrow \text{bus}, IR \leftarrow MDR$
- **Decodificación:** se separan campos instrucción
 - Codop: ADD $\text{mem} + \text{reg} \rightarrow \text{reg}$
 - Dato1: 100 direccionamiento directo, habrá que leer $M[100]$
 - Dato2: 0 direccionamiento registro, habrá que llevar R0 a ALU
- CPUs con longitud instrucción variable – dirección (100) en siguiente palabra
- **Operando:** $MAR \leftarrow 100, \text{Read}, T_{acc}, MDR \leftarrow \text{bus}, ALU_{in1} \leftarrow MDR$
- **Ejecución:** $ALU_{in2} \leftarrow R0, \text{add}, T_{alu}$
- **Almacenamiento:** $R0 \leftarrow ALU_{out}$

Unidad de procesamiento con un bus

■ Una instrucción puede ser ejecutada mediante una o más de las siguientes operaciones:

- Transferir de un registro a otro
- Realizar operación aritmética o lógica y almacenar en registro
- Cargar posición de memoria en registro
- Almacenar registro en posición de memoria

■ Ejecución de una instrucción completa

- Ej.: **Add (R3) \rightarrow R1**
 1. Enable PC, Load MAR, Select 4, Sumar,  able Z
 2. Comenzar lectura, Enable Z, Load PC, Load Y*
 3. Esperar fin de ciclo de memoria, Load MDR desde mem.
 4. Enable MDR hacia bus interno, Load IR
 5. Decodificar instrucción
 6. Enable R3, Load MAR
 7. Comenzar lectura, Enable R1, Load Y
 8. Esperar fin de ciclo de memoria, Load MDR desde mem.
 9. Enable MDR hacia bus interno, Select Y, Sumar, Load Z
 10. Enable Z, Load R1, Saltar a captación

} captación

} ejecución

Unidad de control

■ Señales de entrada a la UC:

- Señal de reloj
- Instrucción actual (codop, campos de direccionamiento,...)
- Estado de la unidad de proceso
- Señales externas (por ej. interrupciones)

■ Señales de salida de la UC:

- Señales que gobiernan la unidad de procesamiento:
 - Carga de registros
 - Incremento de registros
 - Desplazamiento de registros
 - Selección de entradas de multiplexores
 - Selección de operaciones de la ALU ...
- Señales externas
 - Por ej. lectura/escritura en memoria

Tipos de unidades de control

■ Existen dos formas de diseñar la UC:

- **Control fijo o cableado** ("hardwired")
 - Se emplean métodos de diseño de circuitos digitales secuenciales a partir de diagramas de estados.
 - El circuito final se obtiene conectando componentes básicos como puertas y biestables, aunque más a menudo se usan PLA.
- **Control microprogramado**
 - Todas las señales que se pueden activar simultáneamente se agrupan para formar palabras de control, que se almacenan en una memoria de control (normalmente ROM).
 - Una instrucción de lenguaje máquina se transforma sistemáticamente en un programa (microprograma) almacenado en la memoria de control.
 - Mayor facilidad de diseño para instrucciones complejas
 - Método estándar en la mayoría de los CISC.

Ejemplo de UC cableada

- Implementación de una unidad de control cableada sencilla (ODE)
- Pasos a seguir para llegar al diseño físico:
 1. Definir una máquina de estados finitos
 2. Describir dicha máquina en un lenguaje de alto nivel
 3. Generar la tabla de verdad para la PLA
 4. Minimizar la tabla de verdad
 5. Diseñar físicamente la PLA partiendo de la tabla de verdad

Concepto de UC microprogramada

- Definiciones:
 - **Microinstrucción**: cada palabra de la memoria de control
 - **Microprograma**: conjunto ordenado de microinstrucciones cuyas señales de control constituyen el cronograma de una (macro)instrucción del lenguaje máquina.
 - Ejecución de un microprograma: lectura en cada pulso de reloj de una de las microinstrucciones que lo forman, enviando las señales leídas a la unidad de proceso como señales de control.
 - Microcódigo: conjunto de los microprogramas de una máquina.

Formato de las microinstrucciones

Micro-programación horizontal:

- Ninguna o escasa codificación
- ✓ Capacidad para expresar un alto grado de paralelismo en las microoperaciones a ejecutar (simultáneamente)
- ✗ Microinstrucciones largas

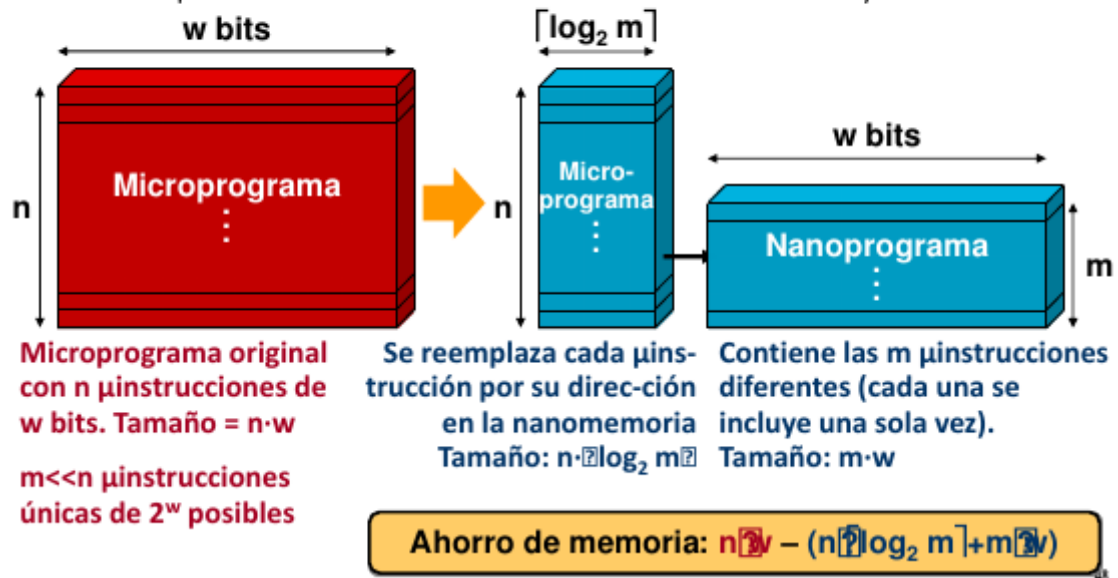
Micro-programación vertical:

- Mucha codificación
- ✓ Microinstrucciones cortas
- ✗ Escasa capacidad para expresar paralelismo (la longitud del programa se ve incrementada)

Nanoprogramación

■ Objetivo: reducir el tamaño de la memoria de control

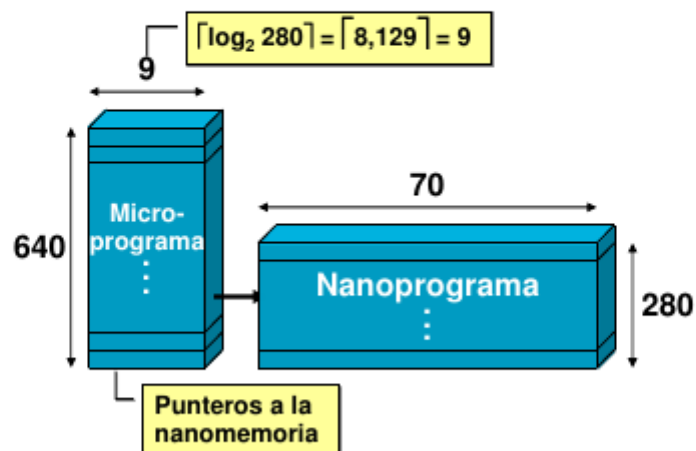
- Implica una memoria a dos niveles: memoria de control y nanomemoria.



Nanoprogramación. Ejemplo 2

■ Estructura de la UC del Motorola 68000

- 640 microinstrucciones, de las cuales 280 son únicas



Ahorro de memoria: $640 \cdot 70 - (640 \cdot 9 + 280 \cdot 70) = 44800 - 25360 = 19440$ bits (43%)

Control residual

■ Control inmediato:

- Hasta aquí, todas las señales de control necesarias para manipular la microarquitectura estaban codificadas en campos de la microinstrucción actual.

■ Control residual:

- En ciertos casos puede ser útil que una microinstrucción pueda almacenar señales de control en un registro (de control residual) para usarlas en ciclos posteriores.
- Objetivo: optimizar el tamaño del microprograma.
- Usos:
 - En microsubrutinas o conjuntos compartidos de μ instrucciones.
 - En caso de que parte de la información de control permanezca invariable durante muchas μ instrucciones.

Diseño horizontal

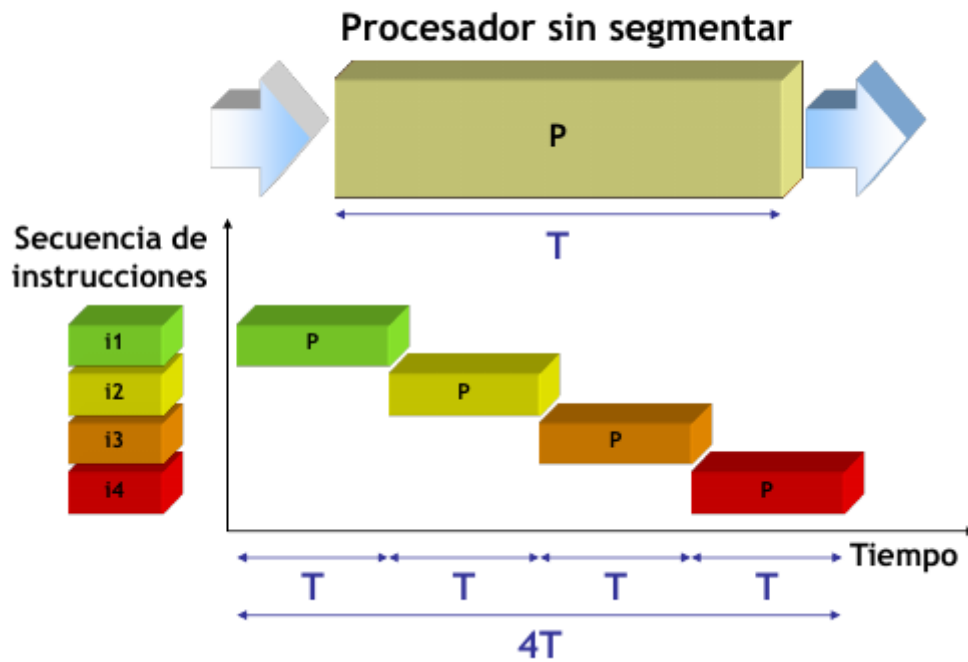
■ Arquitectura (3): repertorio de 23 instrucciones máquina

Binario	Nemotécnico	Instrucción	Significado
0000xxxxxxxxxxxx	LODD	Carga directa	$ac := m[x]$
0001xxxxxxxxxxxx	STOD	Almacenamiento directo	$m[x] := ac$
0010xxxxxxxxxxxx	ADDD	Suma directa	$ac := ac + m[x]$
0011xxxxxxxxxxxx	SUBD	Resta directa	$ac := ac - m[x]$
0100xxxxxxxxxxxx	JPOS	Salto si positivo	if $ac \geq 0$ then $pc := x$
0101xxxxxxxxxxxx	JZER	Salto si cero	if $ac = 0$ then $pc := x$
0110xxxxxxxxxxxx	JUMP	Salto incondicional	$pc := x$
0111xxxxxxxxxxxx	LOCO	Carga de constante	$ac := x$ ($0 \leq x \leq 4095$)
1000xxxxxxxxxxxx	LODL	Carga local	$ac := m[sp + x]$
1001xxxxxxxxxxxx	STOL	Almacenamiento local	$m[sp + x] := ac$
1010xxxxxxxxxxxx	ADDL	Suma local	$ac := ac + m[sp + x]$
1011xxxxxxxxxxxx	SUBL	Resta local	$ac := ac - m[sp + x]$
1100xxxxxxxxxxxx	JNEG	Salto si negativo	if $ac < 0$ then $pc := x$
1101xxxxxxxxxxxx	JNZE	Salto si no cero	if $ac \neq 0$ then $pc := x$
1110xxxxxxxxxxxx	CALL	Llamada a subrutina	$sp := sp - 1; m[sp] := pc; pc := x$
1111000000000000	PSHI	Apilamiento indirecto	$sp := sp - 1; m[sp] := m[ac]$
1111001000000000	POPI	Desapilamiento indirecto	$m[ac] := m[sp]; sp := sp + 1$
1111010000000000	PUSH	Apilamiento	$sp := sp - 1; m[sp] := ac$
1111011000000000	POP	Desapilamiento	$ac := m[sp]; sp := sp + 1$
1111100000000000	RETN	Retorno de subrutina	$pc := m[sp]; sp := sp + 1$
1111101000000000	SWAP	Intercambio de AC y SP	$tmp := ac; ac := sp; sp := tmp$
11111100yyyyyyyy	INSP	Incremento de SP	$sp := sp + y$ ($0 \leq y \leq 255$)
11111110yyyyyyyy	DESP	Decremento de SP	$sp := sp - y$ ($0 \leq y \leq 255$)

1

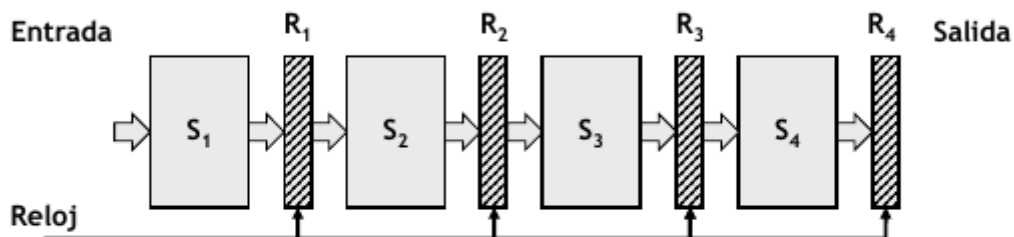
TEMA 4

Concepto de segmentación



Concepto de segmentación

S_i : etapa de segmentación i -ésima
 R_i : registro de segmentación de la etapa i -ésima

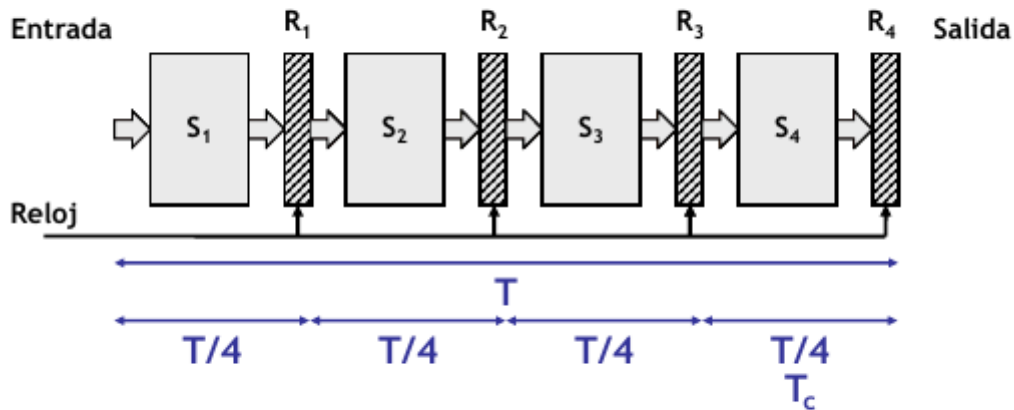


SEGMENTACIÓN EN UN PROCESADOR

Subdividir el procesador en n etapas, permitiendo el solapamiento en la ejecución de instrucciones

Concepto de segmentación

Las instrucciones entran por un extremo del cauce, son procesadas en distintas etapas y salen por el otro extremo.

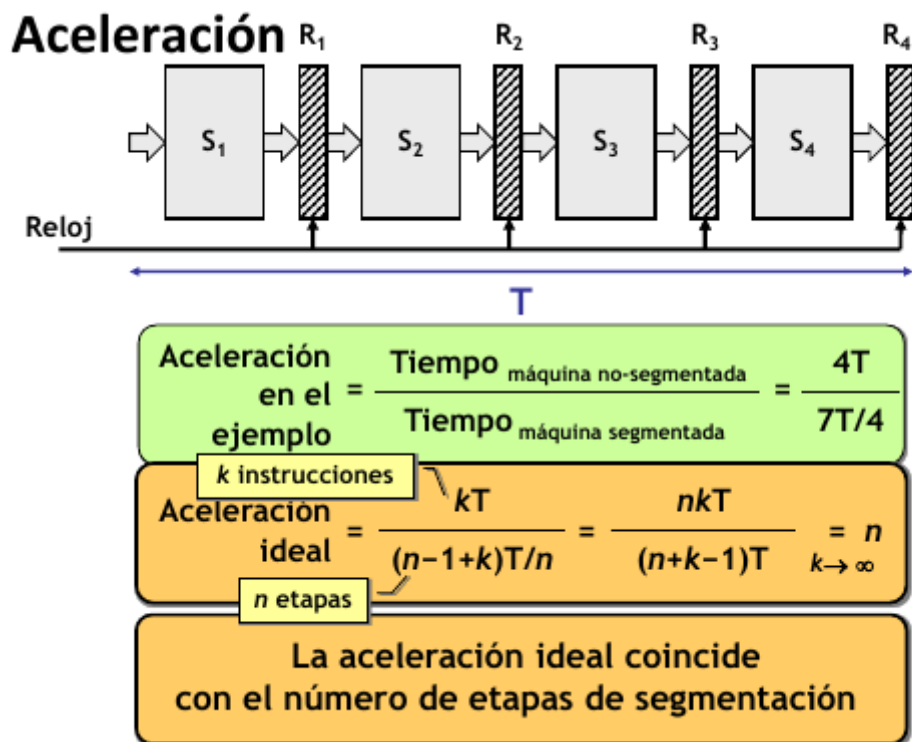


Cada instrucción individual se sigue ejecutando en un tiempo T...

...pero hay varias instrucciones ejecutándose simultáneamente

Ejemplo de segmentación

- Cada etapa del cauce debe completarse en un ciclo de reloj
- Fases de captación y de memoria
 - Si acceden a memoria principal, el acceso es varias veces más lento
 - La caché permite acceso en un único ciclo de reloj
- El periodo de reloj se escoge de acuerdo a la etapa más larga del cauce



Riesgos

Riesgo

Situación que impide la ejecución de la siguiente instrucción del flujo del programa durante el ciclo de reloj designado

Obliga a modificar la forma en la que avanzan las instrucciones hacia las etapas siguientes

Reducción de las prestaciones logradas por la segmentación

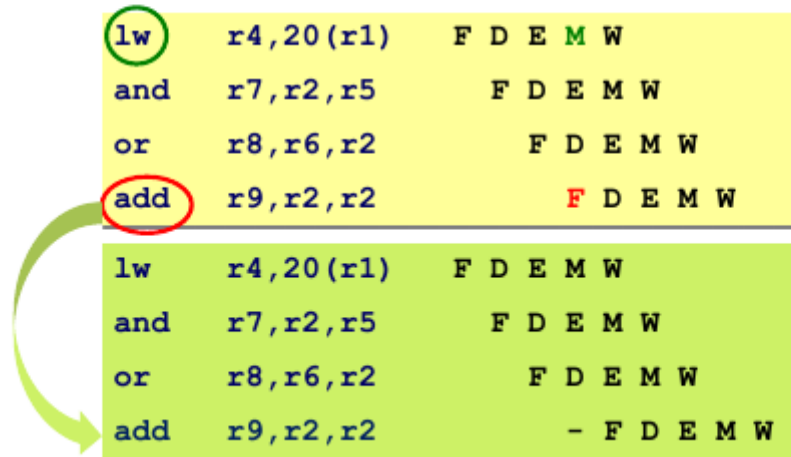
Riesgos

■ Supongamos las siguientes etapas:

- **F**: búsqueda (fetch) de instrucción.
- **D**: decodificación de instrucción / lectura de registros.
- **E**: ejecución / cálculo de direcciones
- **M**: acceso a memoria.
- **W**: escritura (write) de resultados.

Riesgos estructurales

- Conflicto por el **empleo de recursos**, dos instrucciones necesitan un mismo recurso.
- Ej. 1: lectura de dato + captación suponiendo **una única memoria** para datos e instrucciones.

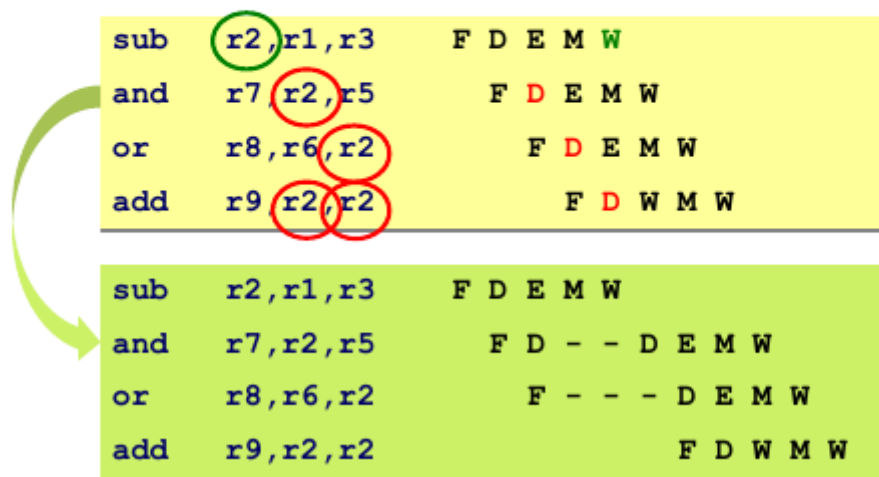


24

- Para reducir el efecto de los fallos de caché se suelen captar instrucciones antes de que sean necesarias (precaptación) y se almacenan en una cola de instrucciones.

Riesgos (por dependencias) de datos

- Acceso a datos cuyo valor actualizado depende de la ejecución de instrucciones precedentes.



Riesgos de control

- Consecuencia de la ejecución de instrucciones de salto.
- Salto **incondicional**:

br	L1	F	D	E	M	W
and	r2,r1,r4	F	D	E	-	-
sub	r5,r6,r7	F	D	-	-	-
or	r8,r1,r6	F	-	-	-	-
L1:add	r6,r1,r4	F	D	E	M	W

- Se pierden 3 ciclos (**huecos de retardo de salto***), ya que tras captar la instrucción br, hasta después del 4º ciclo (es decir, pasados otros 3 ciclos) no se conoce la dirección de salto.
- Importante averiguar la dirección de salto **lo antes posible**, por ej. en la etapa de decodificación:

Riesgos de control

- Salto **condicional**:

beq	r2,r3,L1	F	D	E	M	W
and	r2,r1,r4	F	D	E	M	W
sub	r5,r6,r7	F	D	E	M	W
or	r8,r1,r6	F	D	E	M	W
L1:add	r6,r1,r4	F	D	E	M	W

- Si se produce el salto se pierden 3 ciclos.
- Si no se produce el salto, no se pierden ciclos.

Salto retardado (*delayed branch*)

- En lugar de desperdiciar las etapas posteriores a la de salto, una o más instrucciones parcialmente completadas se completarán antes de que el salto tenga efecto.
- El compilador busca instrucciones **anteriores** lógicamente al salto que pueda colocar tras el salto.
- Si el salto es condicional, las instrucciones colocadas detrás no deben afectar a la condición de salto.

Antes:

```
mov r1,#3
jmp etiq
nop
```

Después:

```
jmp etiq
mov r1,#3
```

Annulling branch

- Un salto de este tipo ejecuta la(s) instrucción(es) siguiente(s) sólo si el salto se produce, pero la(s) ignora si el salto no se produce.
- Con un salto de este tipo, el destino de un salto condicional sí puede colocarse tras el salto.

Antes:

```
    bz else
    nop
    ; código then
    ...
else:
    mov r3,#100
```

Después:

```
    bz,a else+4
    mov r3,#100
    ; código then
    ...
else:
    mov r3,#100
```

Predicción de saltos

- Intentar predecir si una instrucción de salto concreta dará lugar a un salto (**branch taken**) o no (**branch not taken**).
 - Si los resultados de las instrucciones de salto condicional fueran aleatorios, comenzar a ejecutar las instrucciones siguientes al salto desperdiciaría ciclos en la mitad de las ocasiones.
 - Se pueden minimizar las pérdidas de ciclos inútiles si para cada instrucción de salto se puede predecir con un acierto > 50% si el salto se producirá o no.
 - Se pueden hacer predicciones distintas si el salto es hacia direcciones menores o mayores.
- Tipos de predicción:
 - **Estática**: se toma la misma decisión para cada tipo de instrucción
 - **Dinámica**: cambia según la historia de ejecución de un programa

tema 5

Funciones que debe incluir el sistema de E/S (I)

- **Direccionamiento o selección del periférico:**
 - El procesador sitúa en el bus de direcciones la dirección asociada con el dispositivo.
 - Si se conectan varios periféricos debe preverse la forma de que no haya conflictos de acceso al bus.
 - Con p bits \Rightarrow pueden direccionarse 2^p direcciones distintas (mapa de E/S).
 - Cada dirección especifica uno o dos puertos de E/S:
 - El hardware de cada dirección suele ser único (bien entrada o bien salida).
 - Pero a veces los circuitos de E y de S de una única dirección son independientes (misma dirección \Rightarrow dos puertos, uno de entrada y otro de salida).
 - Cada interfaz de periférico emplea varios puertos para comunicarse con el procesador.

Funciones que debe incluir el sistema de E/S (VI)

■ **Sincronización:**

- Acomodación de las velocidades de funcionamiento del procesador/MP y los dispositivos de E/S.
- Hay que establecer un mecanismo para saber cuándo se puede enviar o recibir un dato.
- Deben incluirse:
 - Palabras de memoria temporal en la interfaz que sirvan como **búfer**. La entrada o salida se hace sobre este búfer intermedio. La operación de E/S real se realiza sólo cuando el dispositivo está preparado.
 - **Señales de control de conformidad** para iniciar o terminar la transferencia (listo, petición, reconocimiento).
- La temporización de las transferencias puede ser:
 - Síncrona
 - Asíncrona

¡Ojo! concepto confuso,
depende del autor

Dos acepciones:

la que vamos a ver: síncrono, velocidad de conexión prefijada; asíncrono velocidad variable

la otra: síncrono, señal de reloj en el bus que marca las subidas y bajadas

Funciones que debe incluir el sistema de E/S (XI)

- En la temporización asíncrona o con “handshaking” ...
 - Se establece un **diálogo (handshaking)** para adaptar el cronograma a las necesidades de tiempo del periférico.
 - **Handshaking**: establecimiento de una comunicación sincronizando dos dispositivos mediante acuse de recibo o intercambio de señales de control.
 - Ventajas:
 - ✓ Se pueden **conectar dispositivos con distintos requisitos de tiempo**.
 - ✓ Se tiene una **mayor garantía** de que el **dato sea válido**, puesto que se exige una contestación positiva del periférico.
 - Para evitar que el sistema quede bloqueado si no existe el periférico o éste no contesta, es necesario establecer un **período de espera máximo**, después del cual se considera la transferencia como errónea.

Conceptos a diferenciar (III)

■ E/S aislada o independiente

- El procesador distingue internamente entre espacio de memoria y espacio de E/S.



■ E/S mapeada en memoria

- El procesador no distingue entre accesos a memoria y accesos a los dispositivos de E/S.

E/S independiente frente a E/S en memoria

■ E/S aislada, independiente, o con espacio de E/S

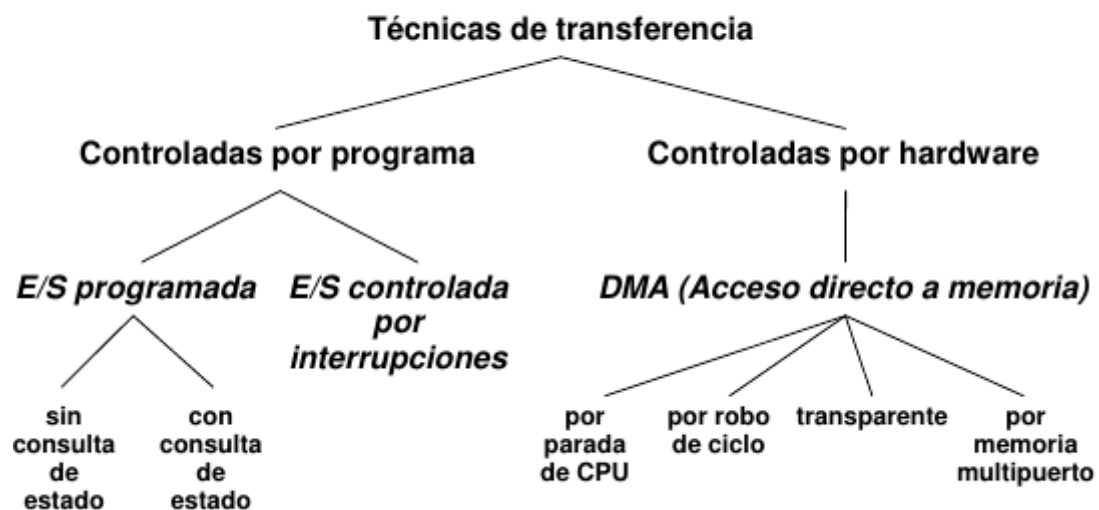
- Emplea la **patilla IO/M#** del procesador
 - **Nivel alto** \Rightarrow Indica a memoria y a dispositivos de E/S que se va a efectuar una operación de **E/S**.
 - Al ejecutar instrucciones específicas de E/S: IN y OUT.
 - **Nivel bajo** \Rightarrow Operación de intercambio de datos con **mem.**
 - Al ejecutar instrucciones de acceso a memoria: LOAD, STORE o MOVE.
- **Instrucciones específicas:** IN y OUT (o READ y WRITE), con poca riqueza de direccionamiento.
- Ejemplo:
 - procesadores de 8 bits empleaban dirección de 8 bits para puertos de E/S \Rightarrow 256 puertos: IN puerto, OUT puerto
 - y disponían de bus de direcciones de 16 bits \Rightarrow 64 K dir. memoria

E/S independiente frente a E/S en memoria

■ E/S mapeada en memoria

- Se usan algunas **direcciones de memoria** para acceder a los puertos de E/S, tras decodificarlas adecuadamente.
- El **procesador no distingue** entre accesos a memoria y accesos a los dispositivos de E/S.
 - **No** se usa la patilla IO/M#
- **No se dispone de instrucciones especiales**, sino que se usan LOAD, STORE o MOVE.
- Para evitar particionar el mapa dedicado a memoria, se agrupa la E/S en una zona bien definida al principio o final del mapa de memoria.

Técnicas de E/S



Técnicas de E/S

- **E/S programada**

-El procesador participa activamente ejecutando instrucciones en todas las fases de una operación de E/S: inicialización, transferencia de datos y terminación.

- **E/S controlada por interrupciones**

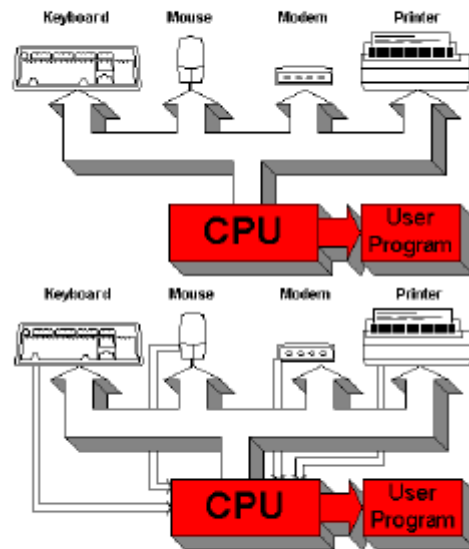
-Los dispositivos de E/S se conectan al procesador a través de líneas de petición de interrupción, que se activan cuando los dispositivos requieren los servicios del procesador.

-En respuesta, el procesador suspende la ejecución del programa en curso y ejecuta un programa de gestión de interrupción para transmitir datos con el dispositivo.

-Como en la E/S programada, los pasos de transferencia de datos están bajo el control directo de programas de control.

- **E/S mediante DMA**

-Requiere la presencia de un controlador DMA, que puede actuar como controlador del bus y supervisar las transferencia de datos entre MP y uno o más dispositivos de E/S, sin intervención directa del procesador salvo en la inicialización.



31

Concepto de interrupción

- **Interrupciones:**

excepto en el caso de las interrupciones software

- **Bifurcaciones normalmente externas al programa en ejecución,** provocadas por muy diversas causas
 - externas (señales que provienen del exterior del procesador), o
 - internas (la interrupción la puede producir el propio procesador)
- ...cuyo objetivo es **reclamar la atención del procesador** sobre algún acontecimiento o hecho importante
- ...pidiendo que se **ejecute un programa específico** para tratar dicho acontecimiento,

(el código que se ejecuta en respuesta a una solicitud de interrupción se conoce como rutina de servicio de interrupción o ISR (*Interrupt Service Routine*)).

- ...de manera que **el programa en ejecución queda temporalmente suspendido.**

(POLLING I)

- Se usa para
 - Identificar el origen de una interrupción
 - Establecer un mecanismo software de asignación de prioridades a los dispositivos
- En esta solución, el ordenador dispone normalmente de una única línea de interrupción INT#, que sirve para que cualquier dispositivo solicite una interrupción.
- La línea INT# se organiza en colector abierto (OR cableado)
 - Cualquier dispositivo puede poner $INT_i=1$ para solicitar la interrupción, lo que hace que INT# se active (se ponga a 0).
- La ISR está en una posición de memoria fija y se encarga de identificar cuál es el dispositivo que interrumpió, comprobando el valor de los biestables de interrupción de los dispositivos, que estarán a 1 para aquellos dispositivos que solicitan interrupción.

(DAISY-CHAIN I)

- Se usa para establecer un mecanismo hardware de asignación de prioridades a los dispositivos.
- Se basa en dos señales comunes a todos los peticionarios y a la CPU:
 - INT#: Petición de interrupción
 - INTA#: Concesión o aceptación de interrupción
- Los peticionarios se conectan a INT# en colector abierto.
 - Esto permite que uno o varios dispositivos soliciten simultáneamente la interrupción, poniendo un 0 en INT#
- La señal INTA# sirve, a modo de testigo, para que uno solo de los peticionarios sea atendido.
 - Es un pulso que recorre en serie, uno tras otro, los dispositivos.
 - Es tomado y eliminado por el primero que desea ser atendido.

tema 6.1

Algunas definiciones

■ Tiempo de acceso:

- tiempo que se requiere para leer (o escribir) un dato (palabra) en la memoria
 - medido en ciclos o en (n-μ-m) s

■ Ancho de banda: (de la memoria de un computador)

- Número de palabras a las que puede acceder el procesador (o que se pueden transferir entre el procesador y la memoria) por unidad de tiempo
 - medido en (K-M-G) B/s

Algunas definiciones

■ Métodos de acceso:

- **Aleatorio (RAM):** tiempo de acceso independiente de posición a acceder
 - SRAM, ROM
- **Secuencial (SAM):** t. acceso depende de posición de los datos a acceder
 - Cinta magnética
- **Directo** (semialeatorio, **DASD** – direct access storage device)
 - tiempo acceso tiene una componente aleatoria y otra secuencial
 - Discos giratorios (época en que lat. rotacional >> t. búsqueda)
- actualmente muchos dispositivos tienen 2 o más componentes acceso
 - DRAM: $CL-T_{RCD}-T_{RP}-T_{RAS}$
 - (CAS latency, Row-Col delay, Row precharge, Row active)
 - No es lo mismo acceder a nueva fila, a otra columna de la misma fila, a ráfaga...

Memoria de Acceso Aleatorio (RAM)

■ Características principales

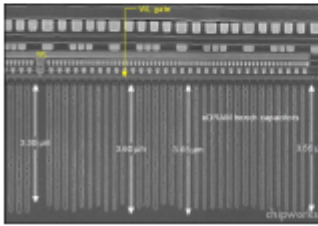
- La **RAM** tradicionalmente se empaqueta como un chip
 - o incluida[†] como parte de un chip procesador
- Unidad básica almacenamiento es normalmente una celda (1 bit/celda)
- Múltiples chips de RAM forman una memoria

■ La RAM tiene dos variedades:

- SRAM (RAM estática)
- DRAM (RAM Dinámica)

Tecnologías RAM

■ DRAM

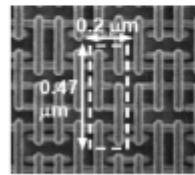


■ (1 Transistor + 1 condensador) / bit

- Condensador orientado verticalmente

■ Debe refrescar estado periódicamente

■ SRAM



■ 6 transistores / bit

- 2 inversores (x 2 tr) + 2 puertas de paso

■ Mantiene el estado indefinidamente

Resumen SRAM vs DRAM

	Trans. por bit	tiempo acceso	necesita refresco?	necesita EDC [†] ?	Coste	Aplicaciones
SRAM	6 ó 8	1x	No	Quizás	100x	memoria cache
DRAM	1	10x	Sí	Sí	1x	memoria principal, frame buffers [†]

EDC[†]: Error detection and correction

■ Tendencias

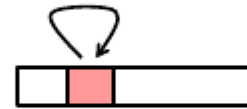
- La SRAM escala con la tecnología de semiconductores
 - está llegando a sus límites
- Escalado DRAM limitado por mínima capacidad necesaria C_(condensador)
 - razón de aspecto limita cómo de profundo se puede hacer el C
 - también llegando a su límite

Localidad

- **Principio de localidad:** Los programas tienden a usar datos e instrucciones con direcciones iguales o cercanas a las que han usado recientemente

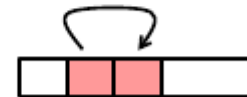
- **Localidad temporal:**

- Elementos referenciados recientemente probablemente serán referenciados de nuevo en un futuro próximo



- **Localidad espacial:**

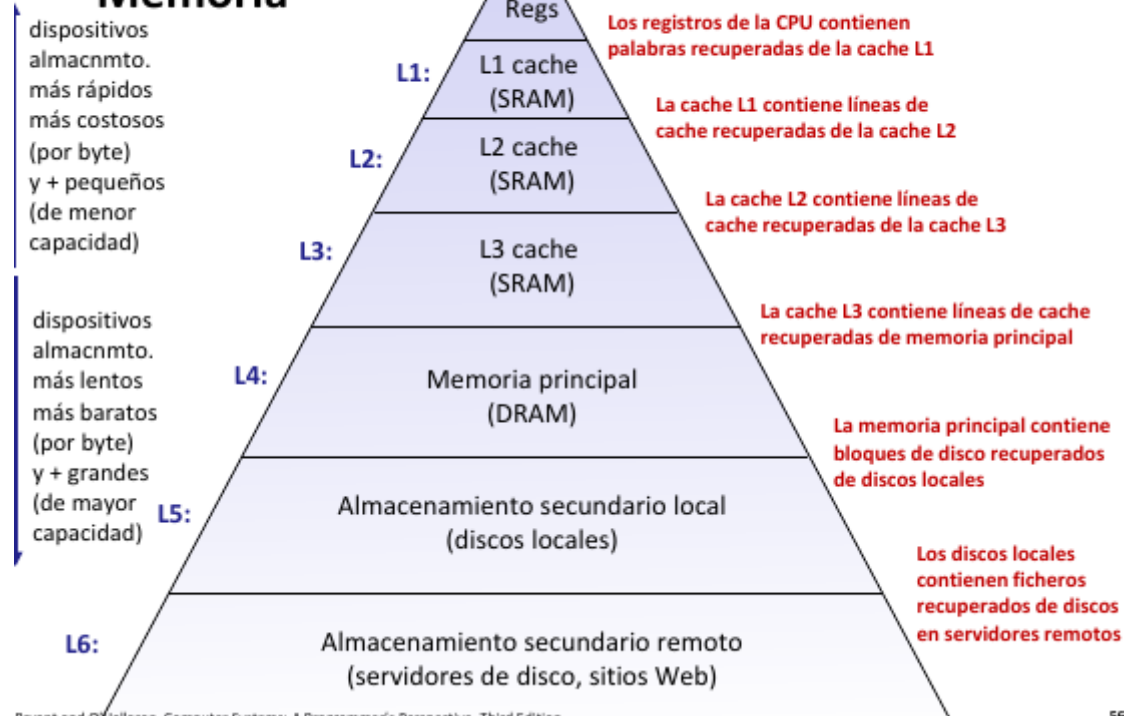
- Elementos con direcciones cercanas tienden a ser referenciados muy juntos en el tiempo



Expresión matemática de localidad

- si en instante tiempo t se accede al dato/posición mem. $d(t)$...
- **Temporal**
 - $d(t + n) = d(t)$ con n pequeño
- **Espacial**
 - $d(t + n) = d(t) + k$ con n, k pequeños

Ejemplo Jerarquía Memoria



Caches

- **Cache:** Un dispositivo de almacenamiento más rápido y pequeño que funciona como zona de trabajo temporal para un subconjunto de los datos de otro dispositivo mayor y más lento
- **Idea fundamental de una jerarquía de memoria:**
 - $\forall k$, el dispositivo a nivel k (+rápido, +pequeño) sirve de cache para el dispositivo a nivel $k+1$ (+lento, +grande)
- **¿Por qué funcionan bien las jerarquías de memoria?**
 - Debido a la localidad, los programas suelen acceder a los datos a nivel k más a menudo que a los datos a nivel $k+1$
 - Así, el almacenamiento a nivel $k+1$ puede ser más lento, y por tanto más barato (por bit) y más grande
- **Idea Brillante (ideal):** La jerarquía de memoria conforma un gran conjunto de almacenamiento que cuesta como el más barato pero que proporciona datos a los programas a la velocidad del más rápido

Conceptos Generales de Cache: 3 Tipos de Fallo de Cache

- **Fallos en frío (obligados)**
 - Los fallos en frío ocurren porque la cache empieza vacía y esta es la primera referencia al bloque
- **Fallos por capacidad**
 - Ocurren cuando el conjunto de bloques activos (**conjunto de trabajo**) es más grande que la cache
- **Fallos por conflicto**
 - Mayoría caches limitan que los bloques a nivel $k+1$ puedan ir a pequeño subconjunto (a veces unitario) de las posiciones de bloque a nivel k
 - P.ej. Bloque i a nivel $k+1$ debe ir a bloque $(i \bmod 4)$ a nivel k (corr. directa)
 - Fallos por conflicto ocurren cuando cache nivel k suficientemente grande pero a varios datos les corresponde ir al mismo bloque a nivel k
 - P.ej. Referenciar bloques 0, 8, 0, 8, 0, 8, ... fallaría continuamente (ejemplo anterior con correspondencia directa)

Tiempo de acceso a disco

- **Tiempo promedio acceso algún sector determinado, aprox:**
 - $T_{\text{acceso}} = T_{\text{prom búsqueda}} + T_{\text{prom rotación}} + T_{\text{prom transferencia}}$
- **Tiempo de búsqueda ($T_{\text{prom búsqueda}}$)**
 - Tiempo para colocar cabezales sobre el cilindro que contiene el sector
 - Valores típicos $T_{\text{prom búsqueda}} = 3\text{—}9\text{ ms}$
- **Latencia rotacional ($T_{\text{prom rotación}}$)**
 - Tiempo esperando a que pase bajo cabezales el primer bit del sector
 - $T_{\text{prom rotación}} = 1/2 \times 1/\text{RPMs} \times 60\text{ s}/1\text{ min}$
 - Velocidad rotacional típica = 7,200 RPMs ($\Rightarrow 4.17\text{ ms}$)
- **Tiempo de transferencia ($T_{\text{prom transferencia}}$)**
 - Tiempo para leer los bits del sector
 - $T_{\text{prom transferencia}} = 1/\text{RPMs} \times 1/(\# \text{ sectores/pista prom}) \times 60\text{ s}/1\text{ min}$

Tiempo para una rotación (minutos) fracción de rotación a leer

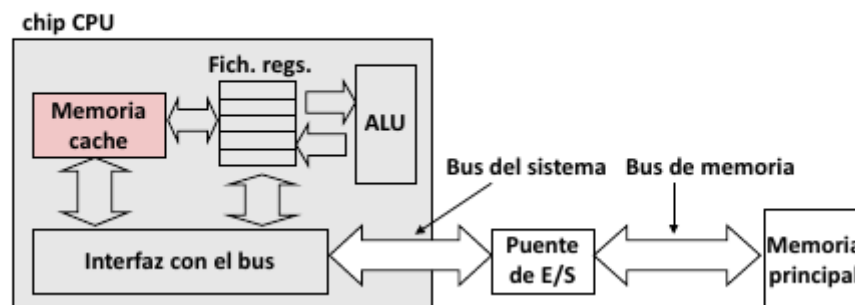
Resumen

- La brecha de velocidad entre CPU, memoria y almacenamiento masivo continúa ampliándose.
- Los programas bien escritos exhiben una propiedad llamada localidad.
- Las jerarquías de memoria, basadas en cacheado, cierran la brecha al explotar la localidad.
- El progreso en memoria flash está sobrepasando a todas las demás tecnologías de memoria y almacenamiento (DRAM, SRAM, disco magnético)
 - Capaz de apilar celdas en tres dimensiones

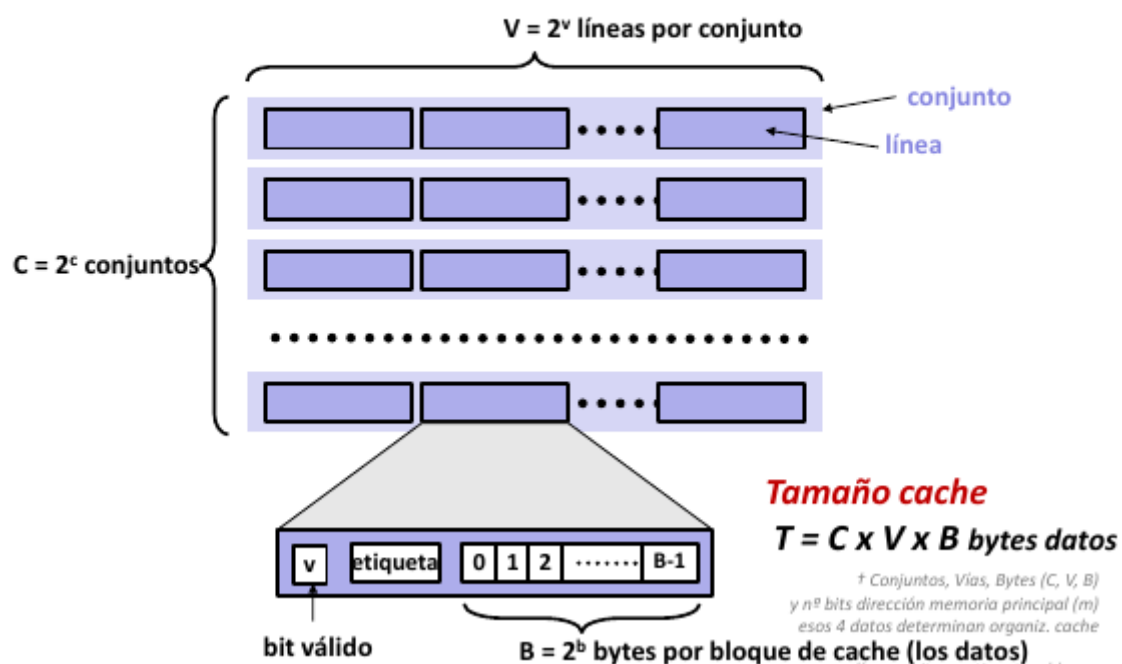
tema 6.2

Memorias Cache

- Las **memorias cache** son memorias pequeñas y rápidas basadas en SRAM gestionadas automáticamente por hardware
 - Retiene bloques de memoria principal accedidos frecuentemente
- La CPU busca los datos primero en caché
- Estructura típica del sistema:



Organización General de Cache (C, V, B, **m**[†])



Cache: resumen de políticas de colocación

- **Organización (C,V,B,m)**
 - La caché tiene $2^c \times 2^v = 2^{c+v}$ líneas. Un bloque tiene 2^b bytes. Una dirección física tiene m bits. La MP tiene 2^m bytes (2^{m-b} bloques).
- **Correspondencia directa**
 - Bloque i de MP \Rightarrow línea $i \bmod 2^l$ de cache (L líneas= $2^l=2^{c+v}=2^c$ con $v=0$)
- **Correspondencia totalmente asociativa**
 - Bloque i de MP \Rightarrow cualquier línea de cache
- **Correspondencia asociativa por conjuntos**
 - Bloque i de MP \Rightarrow conjunto $i \bmod 2^c$ de cache (cualquier línea del conjunto)
- **Consideraremos el “Ejemplo 1”:**
 - Tamaño de caché: 2K bytes. $\Rightarrow c+v+b=11$
 - 16 bytes por bloque. $b=4 \Rightarrow c+v=7$, **128 líneas en cache**
 - Memoria principal máx: 256K bytes. $\Rightarrow m=18$, **16K bloques en MP**
 - (C×V=128, B=16, m=18), $c+v=7$, $b=4$, $c+v+b=11$, C×V×B=2K

Métricas para prestaciones de cache

- **Tasa de Fallo**
 - Fracción de referencias a memoria no encontradas en caché (fallos / accesos) = $1 - \text{tasa de acierto}$
 - Valores típicos (en porcentaje):
 - 3-10% para L1
 - puede ser bastante pequeño (por ejemplo, <1%) para L2, dependiendo del tamaño, etc.
- **Tiempo en Acierto⁺** (tiempo de acceso en caso de acierto)
 - Tiempo para entregar una línea de cache al procesador
 - incluye el tiempo para determinar si la línea está en cache
 - Valores típicos :
 - 4 ciclos reloj para L1
 - 10 ciclos reloj para L2
- **Penalización por Fallo**
 - Tiempo adicional requerido debido a un fallo
 - típicamente 50-200 ciclos para M.principal (Tendencia: ¡aumentando!)

Resumen de cache

- **Las memorias cache pueden tener un impacto significativo en el rendimiento**

- **¡Puedes escribir tus programas para aprovechar esto!**
 - Céntrate en los bucles internos, donde se producen la mayor parte de los cálculos y de los accesos a memoria
 - Intenta maximizar la localidad espacial leyendo los datos secuencialmente con paso 1
 - Intenta maximizar la localidad temporal utilizando un dato con la mayor frecuencia posible una vez que se haya leído de memoria