Tema 2: Interfaz Alto Nivel – Ensamblador

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya

The trouble with programmers is that you can never tell what a programmer is doing until it's too late.

- *Seymour Cray*

As soon as we started programming, we found out to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

- *Maurice Wilkes – 1949*

If debugging is the process of removing bugs, then programming must be the process of putting them in.

- *Edsger W. Dijkstra*

Índice

- **Referencias históricas**
- **Visión del programador en ensamblador x86**
  - Espacio de memoria y registros
  - Tipos de datos básicos
  - Modos de direccionamiento
- **Instrucciones**
- **Traducción de sentencias C a ensamblador**
- **Tipos de datos estructurados: vectores y matrices**
- **ABI (Aplication Binary Interface)**
  - Structs
  - Subrutinas

Referencias históricas

Evolución desde el punto de vista de la Arquitectura (Lenguaje Máquina):

- **1978.** Se anuncia el i8086 como una extensión del i8080. El i8086 es un microprocesador de 16 bits. Se queda a medio camino entre una máquina de acumulador y una máquina de registros de propósito general.
- **1980.** Se anuncia el i8087, coprocesador en coma flotante. El LM del i8086 se amplía con 60 instrucciones. Se abandona el concepto de acumulador y se utiliza un híbrido entre un banco de registros y una pila para las operaciones en coma flotante.
- **1982.** El i80286 aumenta el espacio de direcciones a 24 bits. Se ofrece el *real addressing mode* para seguir ejecutando aplicaciones i8086.
- **1985**. El i80386 extiende la arquitectura a 32 bits. Se añaden nuevos modos de direccionamiento e instrucciones. El nuevo procesador es casi una máquina de registros de propósito general.
- En los siguientes procesadores apenas hay cambios en la Arquitectura (4 nuevas instrucciones en 10 años).
- **1997**. El Pentium MMX incluye instrucciones para aplicaciones multimedia (MMX). El conjunto de instrucciones multimedia se ha ido aumentando en los siguientes procesadores: SSE (Pentium III, 1999), SSE2 (Pentium 4, 2001), SSE3 (Pentium 4 Prescott, 2005; ampliado AMD Athlon, 2005), SSE4 (Core y AMD K10, 2007), …
- **2003**. AMD realiza la extensión de la arquitectura a 64 bits. Los registros aumentan a 64 bits y se aumenta su número a 16. Intel tuvo que copiar esta extensión.

Referencias históricas

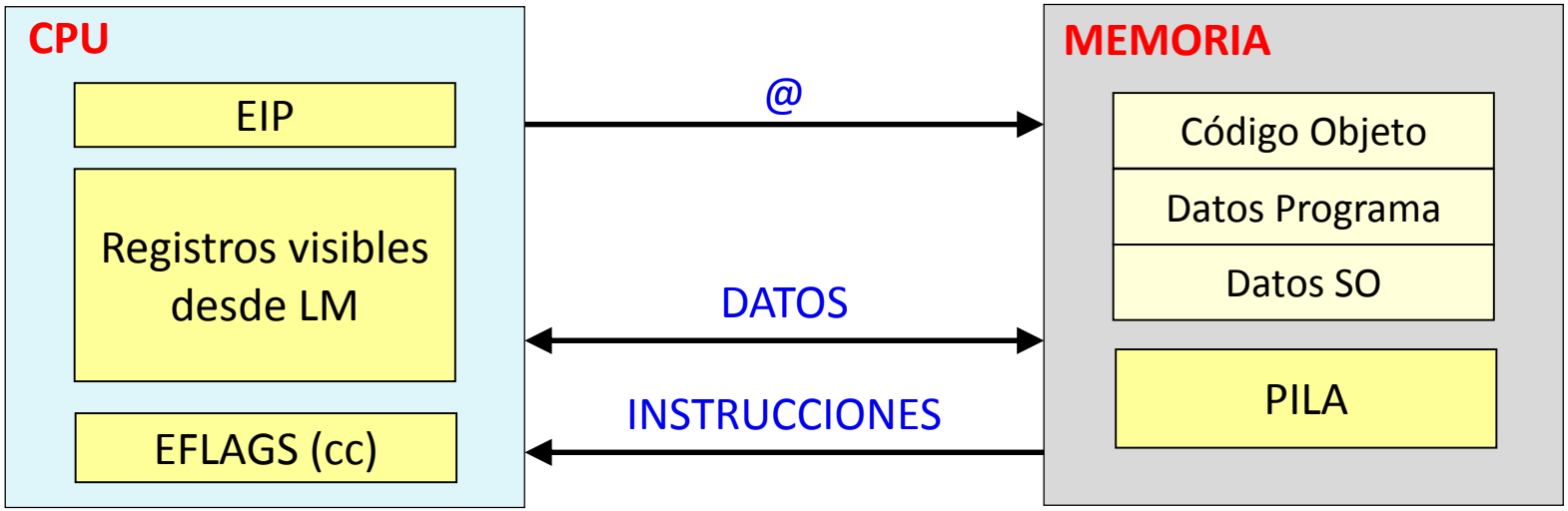
Lenguaje Máquina MIPS (características RISC)

- **Instrucciones aritméticas acceden sólo a registros**
  - En algunos casos un operando puede ser inmediato
- **Solo las instrucciones Load y Store pueden accede a memoria.**
- **Referencias a memoria con modos de direccionamiento simples**
  - **Base + Desplazamiento**
- **Instrucciones de longitud fija**
- **Muchos registros**

Lenguaje Máquina x86 (características CISC)

- **Instrucciones pueden referenciar diferentes tipos de operandos**
  - inmediato, registros, memoria
- **Instrucciones aritméticas pueden leer/escribir en memoria, pero sólo 1 de los 2 operandos puede estar en memoria**
- **Referencias a memoria pueden suponer cálculos complejos**
  - **Rb + S*Ri + D**
- **Instrucciones pueden tener diferente longitud**
- **Pocos registros**

Visión del programador (Estado de la Arqu.)



- **EIP: Contador de programa. Apunta a la siguiente instrucción a ejecutar**
- **Registros: Se usan muy frecuentemente como variables de acceso rápido**
- **Códigos de Condición**
  - Almacenan información respecto al comportamiento de las últimas instrucciones ejecutadas
  - Se usan en los saltos condicionales
- **Memoria**
  - Vector direccionable a nivel de byte, Little Endian
  - Código, datos usuario, datos SO
  - Pila para soportar gestión de subrutinas

Características del ensamblador

- **Tipos de datos básicos**
  - Enteros
    - ✓ dato de 1, 2 ó 4 bytes
    - ✓ datos y direcciones (punteros)
  - Reales (coma flotante): 4, 8 ó 10 bytes
  - No incluye tipos estructurados
    - ✓ Se codifican como datos almacenados de forma contigua
    - ✓ Dispone de modos de direccionamiento para manipularlos
- **Operaciones primitivas**
  - Operaciones aritméticas/lógicas sobre registros y datos en memoria
  - Transferencia de datos entre memoria y banco de registros
  - Saltos condicionales e incondicionales (a/de procedimientos)

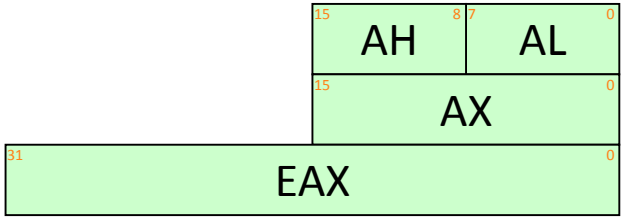
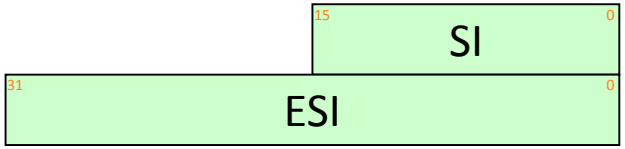
Visión del programador

¿Qué necesitamos estudiar?

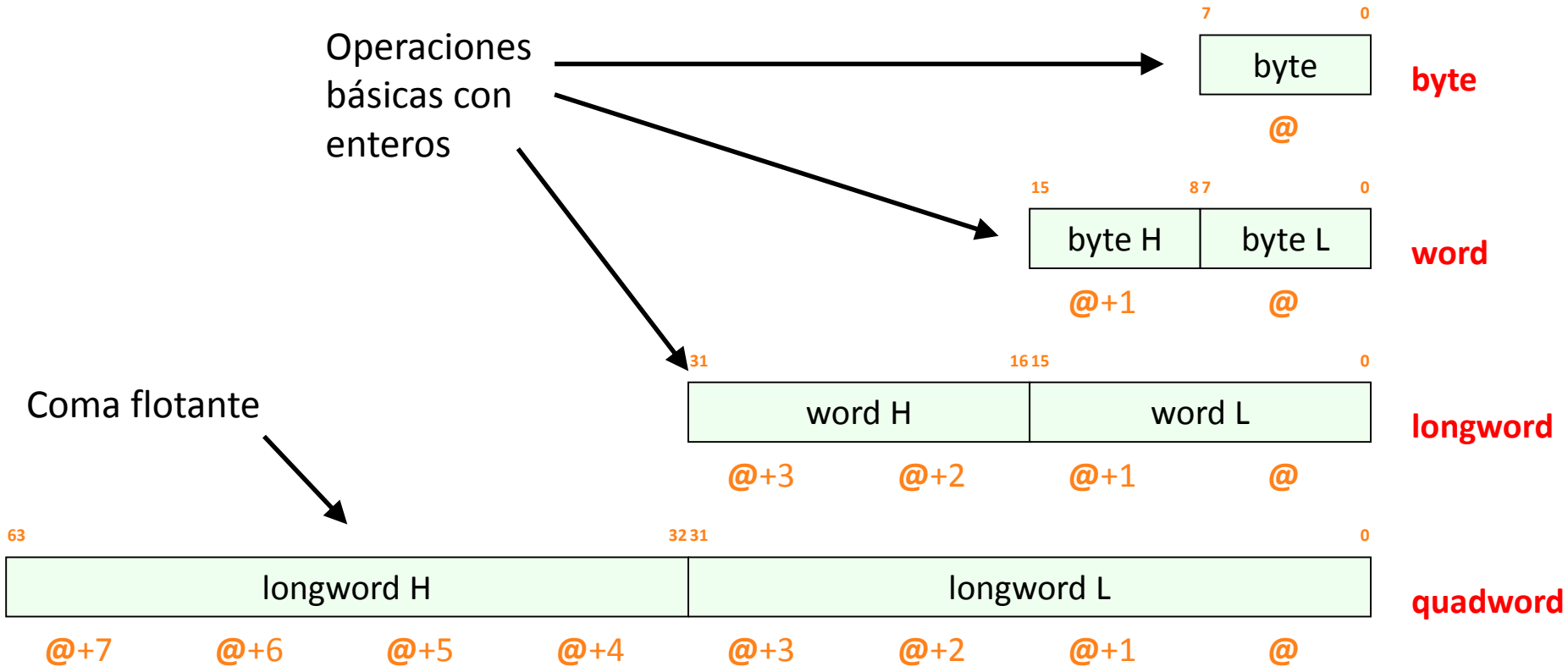
- Espacio de memoria
- Registros disponibles
- Repertorio de instrucciones: qué hacen, cómo se codifican, cuánto tardan
- Tipos y representación de los datos
- Modos de direccionamiento
- Secuenciamiento de instrucciones
- Comunicación con el exterior

Visión del programador

- **Espacio de memoria**
  - Espacio lineal de 2^{32} posiciones de 1 byte: [0 - 2^{32}-1]
  - Modo protegido / Modelo plano de memoria/ Little Endian
- **Registros disponibles**

32 bits	16 bits	8 bits	
%eax	%ax	%ah,%al	
%ebx	%bx	%bh,%bl	
%ecx	%cx	%ch,%cl	
%edx	%dx	%dh,%dl	
%esi	%si		
%edi	%di		
%esp	%sp		Reservados para uso específico de subrutinas
%ebp	%bp		
%eip			Contador programa
%eflags			Palabra de estado

Tipos de datos básicos



Tipos de datos básicos

0	34
1	22
2	5A
3	3B
4	C1
5	45
6	FF
7	00
8	21
9	2A
10	2C
11	7B
12	90
13	43
14	11
15	FF

```
byte 8:  0x21
byte 3:  0x3B
word 8:  0x2A21
word 3:  0xC13B
longword 8 :  0x7B2C2A21
longword 3 :  0xFF45C13B
quadword 8 :  0xFF1143907B2C2A21
quadword 3 :  0x2C2A2100FF45C13B
```

¡ Little Endian !

Tipos de datos básicos

0	34
1	22
2	5A
3	3B
4	C1
5	45
6	FF
7	00
8	21
9	2A
10	2C
11	7B
12	90
13	43
14	11
15	FF

```
byte 8:  0x21
byte 3:  0x3B
word 8:  0x212A
word 3:  0x3BC1
longword 8 :  0x212A2C7B
longword 3 :  0x3BC145FF
quadword 8 :  0x212A2C7B904311FF
quadword 3 :  0x3BC145FF00212A2C
```

¡ Big Endian !

Tipos de datos básicos

Rango Naturales

- **byte**: $0 \leq x \leq 255$
- **word**: $0 \leq x \leq 65.535$
- **longword**: $0 \leq x \leq 4.294.967.215$

Rango Enteros

- **byte**: $-128 \leq x \leq 127$
- **word**: $-32.768 \leq x \leq 32.767$
- **longword**: $-2.147.483.648 \leq x \leq 2.147.483.647$

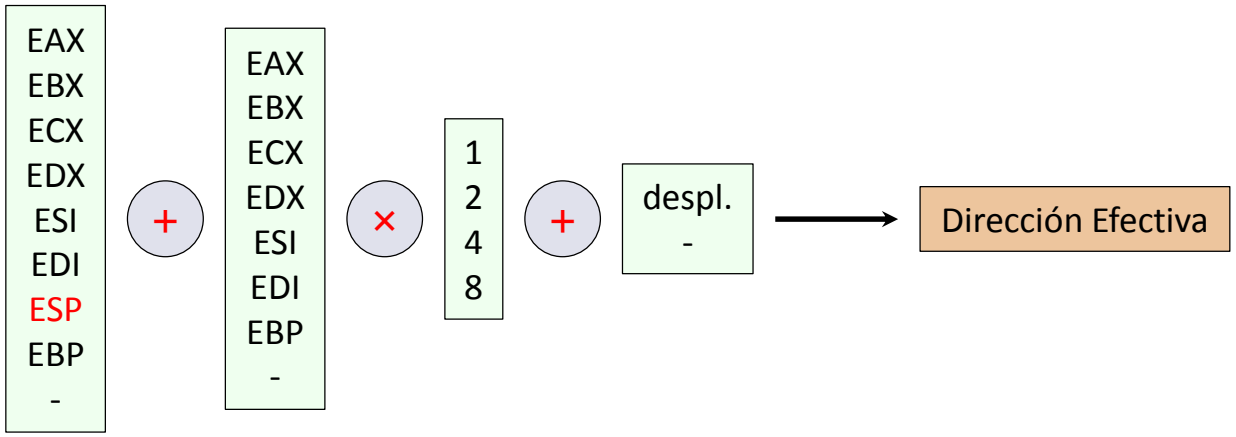
Rango Reales (IEEE 754)

- **Precisión simple** (32 bits, 24 precisión*): $1{,}18 \cdot 10^{-38} \leq x \leq 3{,}40 \cdot 10^{38}$
- **Precisión doble** (64 bits, 53 precisión*): $2{,}23 \cdot 10^{-308} \leq x \leq 1{,}79 \cdot 10^{308}$
- **Precisión doble extendida** (80 bits, 64 precisión*): $3{,}37 \cdot 10^{-4932} \leq x \leq 1{,}18 \cdot 10^{4932}$

* Incluye bit escondido.

Modos de direccionamiento

- **Inmediato**: **$19, $-3, $0x2A, $0x2A45**
  - Codificado con 1, 2 ó 4 bytes
- **Registro**: **%eax, %ah, %esi**
- **Memoria**: **D(Rb, Ri, s)** → **M[Rb+Ri×s+D]**
  - **D**: desplazamiento codificado con 1, 2 ó 4 bytes
  - **Rb**: registro base. Cualquiera de los 8 registros
  - **Ri**: registro índice. Cualquiera excepto %esp
  - **S**: factor escala: 1, 2, 4 u 8

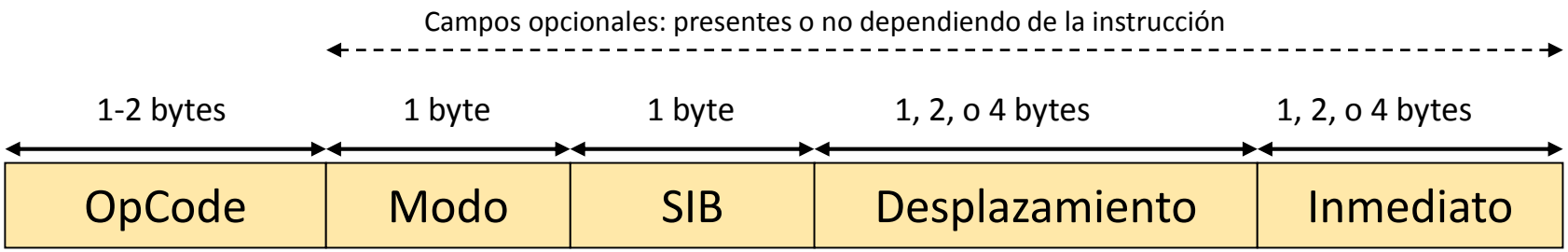


Modos de direccionamiento

- Ejemplos de modos de direccionamiento:

`(%eax,%ebx)`	`M[eax+ebx]`
`-3(%eax,%ebx)`	`M[eax+ebx-3]`
`(%eax,%ebx,4)`	`M[eax+ebx·4]`
`(,%ebx,4)`	`M[ebx·4]`
`12(%eax)`	`M[eax+12]`
`(%eax)`	`M[eax]`
`3(%eax,%esi,2)`	`M[eax+esi·2+3]`
`4`	`M[4]`
`$4`	`4`
`%eax`	`Registro eax`
`%al`	`8 bits de menor peso de eax`

Codificación de las instrucciones

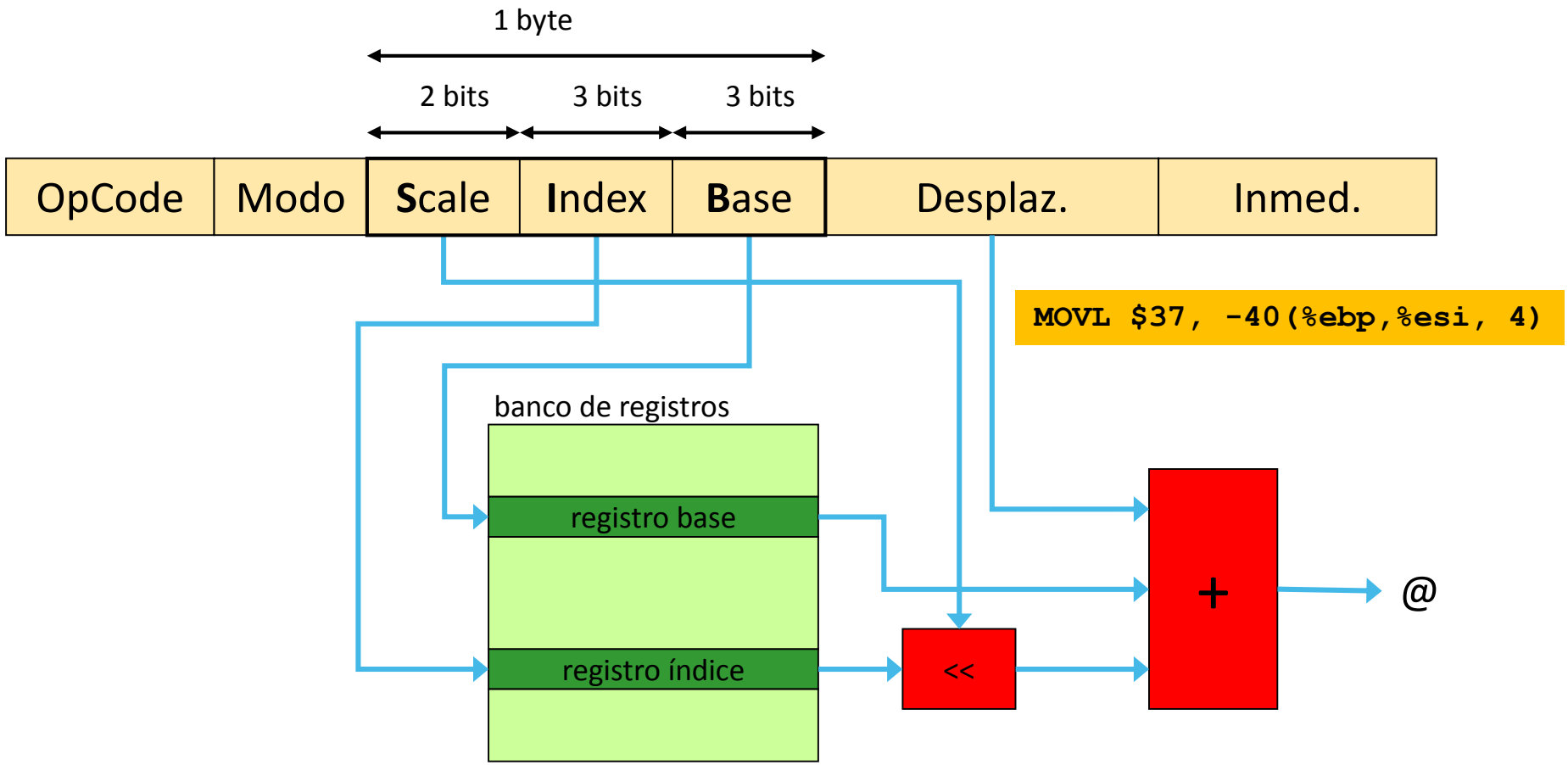


Formato General de las instrucciones

```
MOVL $37, -40(%ebp,%esi, 4)
```

- **OpCode codifica:**
  - la operación a realizar
  - el tamaño de los operandos
  - cuál es el operando fuente y cuál el destino
  - si el operando fuente es un inmediato o registro/memoria
- **Modo codifica:**
  - el modo de direccionamiento del operando memoria si lo hay
  - el registro para los operandos registro
  - indica si hay desplazamiento para el caso de que un operando esté en memoria
- **SIB, en el caso que uno de los operandos esté en memoria, codifica:**
  - el escalado (**S**cale)
  - registro índice (**I**ndex)
  - registro base (**B**ase)

Codificación del modo memoria



Posibles valores de Scale: 0,1,2,3 (equivale a multiplicar por 1,2,4,8 respectivamente)

IA-32 Intel® Architecture Software Developer's Manual Volume 2: Instruction Set Reference

Instrucciones de Movimiento de Datos

Instrucciones	Descripción	Notas	Ejemplo
MOVx op1, op2	op2 ← op1	x = {L, W, B}	MOVB $-1,%AL
MOVSxy op1, op2	op2 ← ExtSign(op1)	xy = {BW, BL, WL}	MOVSBW %CH,%AX
MOVZxy op1, op2	op2 ← ExtZero(op1)	xy = {BW, BL, WL}	MOVZWL %BX,%EDX
PUSHL op1	%ESP ← %ESP – 4; M[%ESP] ← op1		PUSHL 12(%EBP)
POPL op1	op1 ← M[%ESP]; %ESP ← %ESP + 4;		POPL %EAX
LEAL op1, op2	op2 ← &op1	op1: memoria	LEAL (%EBX,%ECX),%EAX

Instrucciones Aritméticas (1/2)

Instrucciones	Descripción	Notas	Ejemplo
ADDx op1, op2	op2 ← op2+op1	x = {L, W, B}	ADDL $13,%EAX
SUBx op1, op2	op2 ← op2-op1	x = {L, W, B}	SUBW %CX,%AX
ADCx op1, op2	op2 ← op2+op1+CF	x = {L, W, B}	ADCL %EDX,%EAX
SBBx op1, op2	op2 ← op2-op1-CF	x = {L, W, B}	SBBL %ECX,%EAX
INCx op1	op1 ← op1+1	x = {L, W, B}	INCL %EAX
DECx op1	op1 ← op1-1	x = {L, W, B}	DECW %BX
NEGx op1	op1 ← -op1	x = {L, W, B}	NEGL %EAX

Instrucciones Aritméticas (2/2)

Instrucciones	Descripción	Notas	Ejemplo
IMUL op1, op2	op2 ← op2·op1	op2: registro	IMUL (%EBX),%EAX
IMUL inm,op1,op2	op2 ← op1·inm	inm: constante	IMUL $3,%EAX,%ECX
IMULL op1	%EDX%EAX ← op1·%EAX	op1: mem. o reg. (**Enteros**)	IMULL (%EBX)
MULL op1	%EDX%EAX ← op1·%EAX	op1: mem. o reg. (**Naturales**)	MULL (%EBX)
CLTD	%EDX%EAX ← ExtSign(%EAX)		CLTD
IDIVL op1	%EAX ← %EDX%EAX / op1 %EDX ← %EDX%EAX % op1	op1: mem. o reg. (**Enteros**)	IDIVL (%EBX)
DIVL op1	%EAX ← %EDX%EAX / op1 %EDX ← %EDX%EAX % op1	op1: mem. o reg. (**Naturales**)	DIVL %ESI

Instrucciones Lógicas

Instrucciones	Descripción	Notas	Ejemplo
ANDx op1, op2	op2 ← op2&op1	x = {L, W, B}	ANDL $13,%EAX
ORx op1, op2	op2 ← op2|op1	x = {L, W, B}	ORW %CX,%AX
XORx op1, op2	op2 ← op2^op1	x = {L, W, B}	XORL %EDX,%EAX
NOTx op1	op1 ← ~op1	x = {L, W, B}	NOTB %AH
SALx k,op1	op1 ← op1<<k (aritm.)	x = {L, W, B}, k: inm. o %CL	SALL $1,%EAX
SHLx k,op1	op1 ← op1<<k (log.)	x = {L, W, B}, k: inm. o %CL	SHLW %CL,%DX
SARx k,op1	op1 ← op1>>k (aritm.)	x = {L, W, B}, k: inm. o %CL	SARL $1,%EAX
SHRx k,op1	op1 ← op1>>k (log.)	x = {L, W, B}, k: inm. o %CL	SHRW %CL,%DX
CMPx op1, op2	op2-op1	x = {L, W, B}, activa flags	CMPL $13,%EAX
TESTx op1, op2	op2&op1	x = {L, W, B}, activa flags	TESTW %CX,%AX

Instrucciones de Secuenciamiento

Instrucciones	Descripción	Notas	Ejemplo
JMP etiq	EIP ← EIP+despl.	EIP ← &etiq	JMP loop
JMP op	EIP ← op	op: reg. o mem.	JMP (%ebx,%esi,4)
Jcc etiq	if (cc) EIP ← EIP+despl.	cc = {E, NE, G, GE, L, LE, …} (**Z**)	JLE else
Jcc etiq	if (cc) EIP ← EIP+despl.	cc = {A, AE, B, BE, …} (**N**)	JA loop
Jcc etiq	if (cc) EIP ← EIP+despl.	cc = {Z, NZ, C, NC, O, …} (flags)	JNC error
CALL etiq	%ESP ← %ESP-4 M[%ESP] ← EIP EIP ← EIP+despl.	Guardar @retorno EIP ← &etiq	CALL sub
CALL op	%ESP ← %ESP-4 M[%ESP] ← EIP EIP ← op	op: reg. o mem.	CALL (%EBX)
RET	EIP ← M[%ESP]; %ESP ← %ESP+4		RET

Códigos de condición (FLAGS)

- **Se activan implícitamente después de ejecutar cualquier instrucción aritmética**
- **Se almacenan en un registro (32 bits) especial del procesador: EFLAGS**

ADDL op1, op2 ; op2 ← op2 + op1

- **CF** (Carry Flag): Carry del bit 31 de la suma. Overflow en unsigned
- **ZF** (Zero Flag): ZF = 1 si t == 0
- **SF** (Sign Flag): SF = 1 si t < 0
- **OF** (Overflow Flag): OF = 1 si (op2>0 && op1>0 && op2+op1<0) || (op2<0 && op1<0 && op2+op1>0)

CMPL op1, op2 ; op2 - op1, y se activan los flags sin guardar el resultado de la resta

- **CF** (Carry Flag): Carry (borrow) de la resta del bit más significativo
- **ZF** (Zero Flag): ZF = 1 si op1 == p2
- **SF** (Sign Flag): SF = 1 si (op2 – op1) < 0 ⇒ op2<op1
- **OF** (Overflow Flag): OF = 1 si (op2>0 && op1<0 && op2-op1<0) || (op2<0 && op1>0 && op2-op1>0)

Flags e Instrucciones de Secuenciamiento

Instrucciones	Flags	Descripción
JE etiq	ZF	Igual / cero
JNE etiq	~ZF	No igual / no cero
JS etiq	SF	Negativo
JNS etiq	~SF	No negativo
JG etiq	~(SF^OF)&~ZF	Mayor (con signo)
JGE etiq	~(SF^OF)	Mayor o igual (con signo)
JL etiq	(SF^OF)	Menor (con signo)
JLE etic	(SF^OF)|ZF	Menor o igual (con signo)
JA etiq	~CF&~ZF	Mayor (sin signo)
JAE etiq	~CF	Mayor o igual (sin signo)
JB etiq	CF	Menor (sin signo)
JBE etiq	CF^ZF	Menor o igual (sin signo)

Ejemplos de Códigos de condición (FLAGS)

Instrucciones	OF	DF	IF	TF	SF	ZF	AF	PF	CF
ADD op1, op2	X				X	X	X	X	X
AND op1, op2	0				X	X	?	X	0
DEC op1	X				X	X	X	X	
NOT op1									
STC									1
MOV op1, op2									
MUL op1	X				?	?	?	?	X
SAL k,op1	i				X	X	?	X	X
LEAL op1, op2									

Significado	
X	Depende resultado
0	cero
1	uno
?	No definido
	No modificado
i	Consultar manual

Para más detalles, ¡consultad el manual!

Ejemplo ensamblador (1/2)

- Convertir ‘abc…’ en ‘ABC…’

```
.data
   .align 4
v: .string "Esto es una frase … salto de linea.\n"
.text
   .align 4
   .globl main
   .type main,@function
main: ...
      xorl %esi,%esi          ; esi ← 0
do:   movb v(, %esi),%al      ; al ← v[i]
      cmpb $'a',%al           ;
      jl cont                 ; ¿v[i] < 'a'?
      cmpb $'z',%al
      jg cont                 ; ¿v[i] > 'z'?
      andb $0xDF, v(, %esi)   ; v[i] ← MAY(v[i])
cont: incl %esi               ; i++
      cmpb $'\n', v(, %esi)   ;
      jne do                  ; ¿v[i] == '\n'?
end:  ...
```

CÓDIGOS ASCII			
A	0100 0001	a	0110 0001
B	0100 0010	b	0110 0010
C	0100 0011	c	0110 00??
…		…	

```
addb $'A'-'a', v(, %esi)
```

Ejemplo ensamblador (2/2)

- $A_i \leftarrow A_i / B_i$, A y B vectores de enteros acabados en 0

```
.data
   .align 4
A: .int 34, 45, 12, …, 56, -67, 0
B: .int -4, 6, 91, …, 12, 4, 0
.text
   .align 4
   .globl main
   .type main,@function
main: ...
      xorl %ecx,%ecx            ; ecx ← 0
      leal A, %edi              ; edi ← @inicio A
      movl $B, %esi             ; esi ← @inicio B
do:   movl (%edi,%ecx,4),%eax   ; eax ← A[i]
      cltd                      ; edx ← ExtSign(eax)
      idivl (%esi,%ecx,4)       ; eax ← edxeax / B[i]
      movl %eax,(%edi,%ecx,4)   ; A[i] ← eax
      incl %ecx                 ; i++
      cmpl $0,(%esi,%ecx,4)       ;
      jne do                    ; ¿B[i] == 0?
end:  ...
```

TRADUCCION DE SENTENCIAS C A ENSAMBLADOR

Sentencia CONDICIONAL (IF-THEN-ELSE)

Modelo:

```
if (cond)
  CUERPO-IF
else
  CUERPO-ELSE
```

Traducción genérica:

```
        evaluar condición
        j(no cumple) else
if:     CUERPO-IF
        jmp endif
else:   CUERPO-ELSE
endif:
```

Ejemplo:

```
int max(int x, int y) {
  int max;
  if (x>y) max = x;
       else max = y;
  return max;
}
```

Ejemplo de IF-THEN-ELSE

Ejemplo:

```
int max(int x, int y) {
  int max;
  if (x>y) max = x;
      else max = y;
  return max;
}
```

Traducción:

```
max:    pushl %ebp
        movl %esp, %ebp
        subl $4, %esp
        movl 8(%ebp),%ecx
        cmpl 12(%ebp),%ecx
        jle else
if:     movl 8(%ebp),%eax
        jmp endif
else:   movl 12(%ebp),%eax
endif:  movl %ebp, %esp
        popl %ebp
        ret
# x → 8[%ebp]
# y → 12[%ebp]
# resultado en %eax
```

Sentencia ITERATIVA (DO-WHILE)

Modelo:

```
do
   CUERPO-DO
while (cond)
```

Traducción genérica:

```
do:   CUERPO-DO
      evaluar condición
      j(cumple) do
end:
```

Ejemplo:

```
int ContA(char v[]) {
  int i, cont;
  cont = 0;
  i = 0;
  do {
    if (v[i] == 'a') cont++;
    i++;
  } while (v[i] != '.');
  return cont;
}
```

Ejemplo de DO-WHILE

Ejemplo:

```
int ContA(char v[]) {
  int i, cont;
  cont = 0;
  i = 0;
  do {
    if (v[i] == 'a') cont++;
    i++;
  } while (v[i] != '.');
  return cont;
}
```

Traducción:

```
ContA:  pushl %ebp
        movl %esp, %ebp
        subl $8, %esp
        movl $0, %eax # cont
        movl $0, %edx # i
do:     movl 8(%ebp),%ecx
        cmpb $'a',(%ecx,%edx)
        jne endif
        incl %eax;
endif:  incl %edx
        cmpb $'.',(%ecx,%edx)
        jne do
end:    movl %ebp, %esp
        popl %ebp
        ret
# @v → 8[%ebp]
```

Sentencia ITERATIVA (WHILE)

Modelo:

```
while (cond) {
  CUERPO-WHILE
}
```

Traducción genérica:

```
while: evaluar condición
       j(no cumple) end
       CUERPO-WHILE
       jmp while
end:
```

Ejemplo:

```
int gcd(int a, int b) {
  while (b!=0) {
    if (a>b) a = a-b;
        else b = b-a;
  }
  return a;
}
```

Ejemplo 1 de WHILE

Ejemplo:

```
int gcd(int a, int b) {
  while (b!=0) {
    if (a>b) a = a-b;
        else b = b-a;
  }
  return a;
}
```

Traducción:

```
gcd:    pushl %ebp
        movl %esp, %ebp
while:  cmpl $0, 12(%ebp)
        je end
        movl 8(%ebp),%eax
        cmpl 12(%ebp),%eax
        jle else
        subl 12(%ebp),%eax
        jmp endif
else:   subl %eax, 12(%ebp)
endif:  jmp while
end:    popl %ebp
        ret
# a → 8[%ebp]
# b → 12[%ebp]
```

Ejemplo 2 de WHILE

Ejemplo:

```
int gcd(int a, int b) {
  int tmp;
  while (b!=0) {
    tmp = b;
    b = a%b;
    a = tmp;
  }
  return a;
}
```

Traducción:

```
gcd:    pushl %ebp
        movl %esp, %ebp
        subl $4, %ebp
while:  cmpl $0,12(%ebp)
        je end
        movl 12(%ebp), %ecx
        movl 8(%ebp), %eax
        cltd
        idivl 12(%ebp)
        movl %edx, 12(%ebp)
        movl %ecx, 8(%ebp)
        jmp while
end:    movl 8(%ebp), %eax
        movl %ebp, %esp
        popl %ebp
        ret
# a → 8[%ebp]
# b → 12[%ebp]
```

Sentencia ITERATIVA (FOR)

Modelo:

```
for(INI; COND; INC) {
  CUERPO-FOR
}
```

Ejemplo:

```
int sumV(int V[], int N) {
  int sum, i;
  sum = 0;
  for (i=0; i<N; i++)
    sum = sum + V[i];
  return sum;
}
```

Traducción genérica:

```
      INI
for:  evaluar condición
      j(no cumple) end
      CUERPO-FOR
      INC
      jmp for
end:
```

Ejemplo de FOR

Ejemplo:

```
int sumV(int V[], int N) {
  int sum, i;
  sum = 0;
  for (i=0; i<N; i++)
    sum = sum + V[i];
  return sum;
}
```

Traducción:

```
sumV: pushl %ebp
      movl %esp, %ebp
      subl $4, %esp
      movl $0, %eax # sum
      movl $0, %ecx # i
for:  cmpl 12(%ebp),%ecx
      jge end
      movl 8(%ebp), %edx
      addl (%edx,%ecx,4),%eax
      incl %ecx
      jmp for
end:  movl %ebp, %esp
      popl %ebp
      ret
# @V → 8[%ebp]
# N → 12[%ebp]
```

Sentencia CONDICIONAL (SWITCH)

Ejemplo:

```
switch_eg(int x)
{
  int result = x;
  switch (x) {
   case 100: result *= 13; break;
   case 102: result += 10;
   case 103: result += 11; break;
   case 104:
   case 106: result *=result; break;
   default: result = 0;
  }
  return result;
}
```

Implementación con una serie de condicionales (tipo if):

- Funciona bien en algunos casos
- Muy lento en la mayoría

Implementación con vector de punteros:

- Más eficiente en general

Ejemplo de SWITCH

Ejemplo:

```
void S(int x)
{
  int result = x;
  switch (x) {
   case 100: result *= 13; break;
   case 102: result += 10;
   case 103: result += 11; break;
   case 104:
   case 106: result *=result; break;
   default: result = 0;
  }
  return result;
}
```

Traducción con IFs:

```
S:    pushl %ebp
      movl %esp, %ebp
      movl 8(%ebp),%eax
      movl %eax, %ebx
      cmpl $100,%ebx
      jne C102
L100: imull $13,%eax
      jmp end
C102: cmpl $102,%ebx
      jne C103
L102: addl $10,%eax
L103: addl $11,%eax
      jmp end
C103: cmpl $103,%ebx
      je L103
C104: cmpl $104,%ebx
      je L106
C106: cmpl $106,%ebx
      jne default
L106: imull %eax,%eax
      jmp end
def:  xorl %eax,%eax
end:  popl %ebp
      ret
```

Otra forma de implementar un SWITCH (1/2)

Ejemplo:

```
void S(int x)
{
  int result = x;
  switch (x) {
   case 100: result *=
   case 102: result +=
   case 103: result +=
   case 104:
   case 106: result *=
   default: result = 0
  }
  return result;
}
```

Traducción con Vector de punteros (pseudoC):

```
code *JT[7] = {L100,LDEF,L102,L103,L104,LDEF,L106};

xi = x - 100;
if ((x<100)||(x>106)) jmp LDEF;
goto JT[x-100];
L100: {código para x==100}; goto DONE;
L102: {código para x==102};
L103: {código para x==103}; goto DONE;
L104:
L106: {código para x==106}; goto DONE;
LDEF: {código para default};
DONE: return result;
```

Otra forma de implementar un SWITCH (2/2)

Ejemplo:

```
void S(int x)
{
  int result = x;
  switch (x) {
   case 100: result *= 13; break
   case 102: result += 10;
   case 103: result += 11; break
   case 104:
   case 106: result *=result; br
   default: result = 0;
  }
  return result;
}
```

Traducción con Vector de punteros :

```
.section .rodata
     .align 4
LT: .long L0, LD, L2, L3, L46, LD, L46
S:   pushl %ebp
     movl %esp, %ebp
     movl 8(%ebp),%eax
     cmpl $100,%eax
     jl LD
     cmpl $106,%eax
     jg LD
     leal -100(%eax),%edx
     jmp LT(,%edx,4)
L0:  imull $13,%eax      # case 100
     jmp end
L2:  addl $10,%eax       # case 102
L3:  addl $11,%eax       # case 103
     jmp end
L46: imull %edx,%edx     # case 104,106
     jmp end
LD:  xorl %eax,%eax      # default
end: popl %ebp
     ret
```

Tipos de datos estructurados

Vectores

- **Declaración en C:**

```
tipo nombre[tamaño];  //indexado a partir de 0
```

- **Almacenamiento en posiciones consecutivas de memoria**
  - Acceso elemento V[i]: **@inicio V + i·tam** (tam: tamaño de los elementos de V)
- **Ejemplos:**

Declaración en C	Tamaño elemento	Tamaño vector	@elemento i
`char A[12];`	**1B**	12B	@inicio A + i
`char *B[80];`	**4B**	320B	@inicio B + **4**·i
`double C[1024];`	**8B**	8KB	@inicio C + **8**·i
`int *D[5];`	**4B**	20B	@inicio D + **4**·i
`int E[100];`	**4B**	400B	@inicio E + **4**·i

Tipos de datos estructurados

- Vectores

Ejemplo:

```
int Vi(int V[100], int i) {
   return V[i];
 }
```

Traducción:

```
Vi: pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp),%ecx      # @V → 8[%ebp]
    movl 12(%ebp),%edx     # i → 12[%ebp]
    movl (%ecx,%edx,4),%eax
    popl %ebp              # resultado en %eax
    ret
```

Tipos de datos estructurados

Matrices

- **Declaración en C:**

```
tipo nombre[NumFilas][NumColumnas]; //indexado a partir de (0,0)
```

- **Almacenamiento por filas en posiciones consecutivas de memoria**
  - Acceso elemento A[i][j]: **@inicio A + (i·NumColumnas + j)·tam** (tam: tamaño de los elementos de A)
- **Ejemplos:**

Declaración en C	Tamaño elemento	Tamaño matriz	@elemento (i, j)
`char A[80][25];`	1B	2000B	@inicio A + i·25 + j
`char *B[80][10];`	4B	3200B	@inicio B + (i·10 + j) · 4
`double C[1024][100];`	8B	800KB	@inicio C + (i·100 + j) · 8
`int *D[5][90];`	4B	1800B	@inicio D + (i·90 + j) · 4
`int E[100][30];`	4B	12000B	@inicio E + (i·30 + j) · 4

Tipos de datos estructurados

Matrices 3-dimensiones

- Ejemplo, matriz de enteros de 3 dimensiones:

```
int M3D[10][64][48]; // cada int ocupa 4 bytes
```

- La matriz se almacena en posiciones consecutivas de memoria: cara a cara y en cada cara por filas.
- Acceso al elemento `M3D[cara][fila][columna]`:
  - `@inicio M3D + (cara·64·48 + fila·48 + columna)·4`
- Es fácil deducir como se almacenan / accede a matrices de N-dimensiones.

Tipos de datos estructurados

- Matrices

Ejemplo:

```
int Mfc(int M[50][80], int fil, int col) {
   return M[fil][col];
 }
```

Traducción:

```
Mfc: pushl %ebp
     movl %esp, %ebp
     movl 8(%ebp),%ecx
     imull $80,12(%ebp),%eax # fil → 12[%ebp]
     addl 16(%ebp),%eax      # col → 16[%ebp]
     movl 8(%ebp),%ecx       # @M → 8[%ebp]
     movl (%ecx,%eax,4),%eax
     popl %ebp               # resultado en %eax
     ret
```

Tipos de datos estructurados

- Matrices

Ejemplo:

```
void Copia(int M[50][80], int X[50][80]) {
   int i, j;
   for (i=0; i<50; i++)
     for (j=0; j<80; j++)
       M[i][j] = X[i][j];
 }
```

Tipos de datos estructurados

- Matrices

Traducción:

```
Copia: pushl %ebp            # i → %ecx
       movl %esp, %ebp       # j → %edx
       salvar reg
       movl 8(%ebp),%edi     # @M → 8[%ebp]
       movl 12(%ebp),%esi    # @X → 12[%ebp]
       xorl %ecx, %ecx
fori:  cmpl $50, %ecx
       jge endi
       cuerpo-FORi
       incl %ecx
       jmp fori
endi:  restaurar reg
       popl %ebp
       ret
```

```
#cuerpo-FORi:
       xorl %edx, %edx
forj:  cmpl $80, %edx
       jge endj
       cuerpo-FORj
       incl %edx
       jmp forj
endj:
```

```
#cuerpo-FORj:
imull $80, %ecx, %eax
addl %edx, %eax
movl (%esi,%eax,4),%ebx
movl %ebx,(%edi,%eax,4)
```

Instrucciones ejecutadas: 15 + 50·(7 + 80·8) = 32.267

Tipos de datos estructurados

- Matrices

Optimización:

```
Copia: pushl %ebp
       movl %esp, %ebp
       salvar reg
       movl 8(%ebp),%edi   # @M → 8[%ebp]
       movl 12(%ebp),%esi  # @X → 12[%ebp]
       xorl %ecx,%ecx
loop:  movl (%esi,%ecx,4),%eax
       movl %eax,(%edi,%ecx,4)
       incl %ecx
       cmpl $4000, %ecx
       jl loop
       restaurar reg
       popl %ebp
       ret
```

Objetivo: reducir el número de instrucciones.

- Se puede ver la matriz como un vector de 4000 posiciones.

Instrucciones ejecutadas: 11 + 4000·5 = 20.011

Tipos de datos estructurados

- Matrices

Optimización: Desenrollar

```
Copia: pushl %ebp
       movl %esp, %ebp
       salvar reg
       movl 8(%ebp),%edi   # @M → 8[%ebp]
       movl 12(%ebp),%esi  # @X → 12[%ebp]
       xorl %ecx,%ecx
loop:  movl (%esi,%ecx,4),%eax
       movl %eax,(%edi,%ecx,4)
       movl 4(%esi,%ecx,4),%eax
       movl %eax,4(%edi,%ecx,4)
       addl $2,%ecx
       cmpl $4000,%ecx
       jl loop
       restaurar reg
       popl %ebp
       ret
```

Objetivo: reducir el número de instrucciones.

- El bucle se ejecuta 1/2 de veces.

Instrucciones ejecutadas: 11 + 2000·7 = 14.011

Tipos de datos estructurados

- Instrucciones SIMD (Single Instruction Multiple Data)

Optimización: Desenrollar 4 + SIMD

```
Copia: pushl %ebp
       movl %esp, %ebp
       salvar reg
       movl 8(%ebp),%edi   # @M → 8[%ebp]
       movl 12(%ebp),%esi  # @X → 12[%ebp]
       xorl %ecx,%ecx
loop:  movdqa (%esi,%ecx,4),%xmm0
       movdqa %xmm0,(%edi,%ecx,4)
       addl $4,%ecx
       cmpl $4000,%ecx
       jl loop
       restaurar reg
       popl %ebp
       ret
```

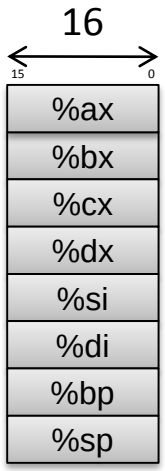
Objetivo: reducir el número de instrucciones.
- El bucle se ejecuta 1/4 de veces y además tiene menos instrucciones

movdqa: mov double quadword (128 bits) aligned
(dirección de inicio de X y M debe ser múltiplo de 16)
(existe **movdqu** u=unaligned pero es menos eficiente)
%xmm0: registro de 128 bits para las extensiones SSE
(en 128 bits podemos almacenar 4 enteros)

Instrucciones ejecutadas: 11 + 1000·5 = 5.011

Instrucciones SIMD

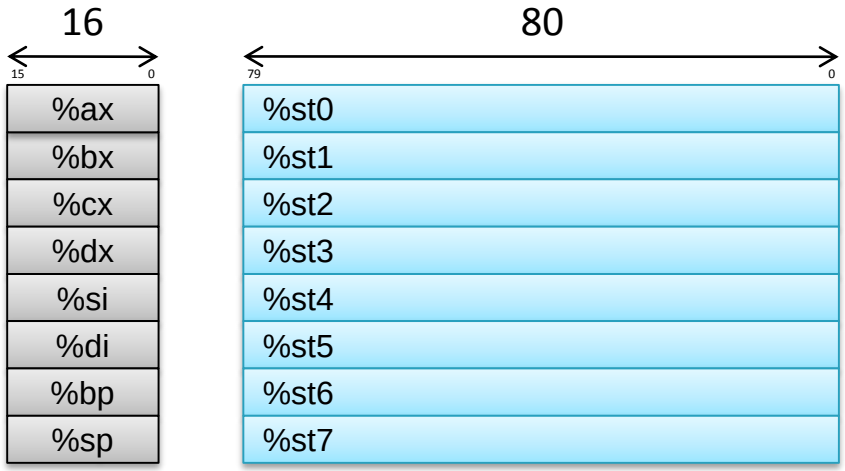
- **Extensiones del lenguaje máquina:**
  - i8086 (1977)



Procesador de 16 bits. A medio camino entre una máquina de acumulador y una máquina de registros de propósito general.

Instrucciones SIMD

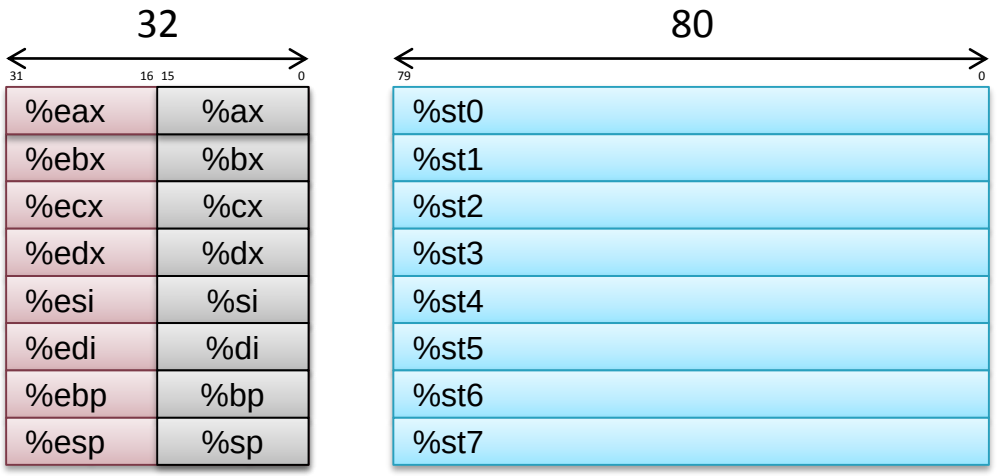
- **Extensiones del lenguaje máquina:**
  - i8086 (1977), i8087 (1980)



Coprocesador de coma flotante: simple precisión (32 bits) doble precisión (64 bits), precisión extendida (80 bits). Añade 60 instrucciones. Híbrido entre un banco de registros de propósito general y máquina de pila (register stack).

Instrucciones SIMD

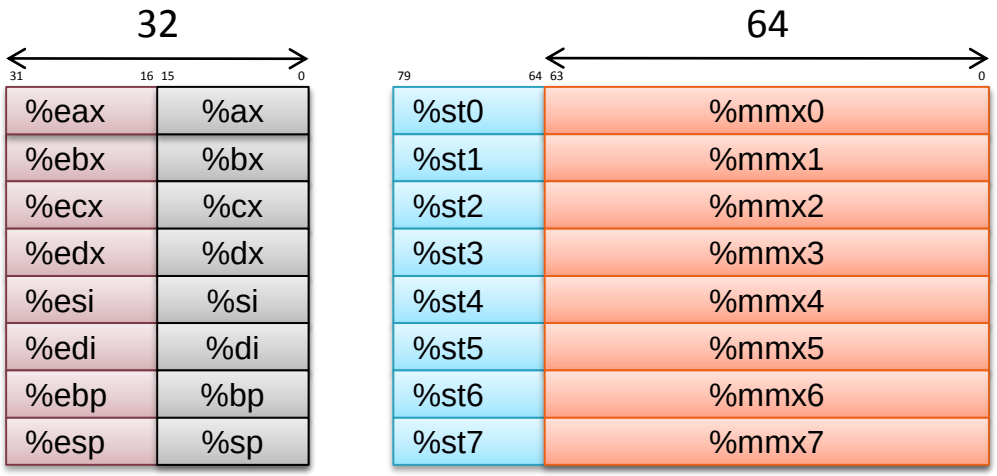
- **Extensiones del lenguaje máquina:**
  - i8086 (1977), i8087 (1980), IA-32 (1985)



El i80386 extiende la arquitectura a 32 bits (IA-32). Se añaden modos de direccionamiento e instrucciones. La extensión IA-32 es una máquina de registros de propósito general (con alguna excepción).

Instrucciones SIMD

- **Extensiones del lenguaje máquina:**
  - i8086 (1977), i8087 (1980), IA-32 (1985), MMX (1997)

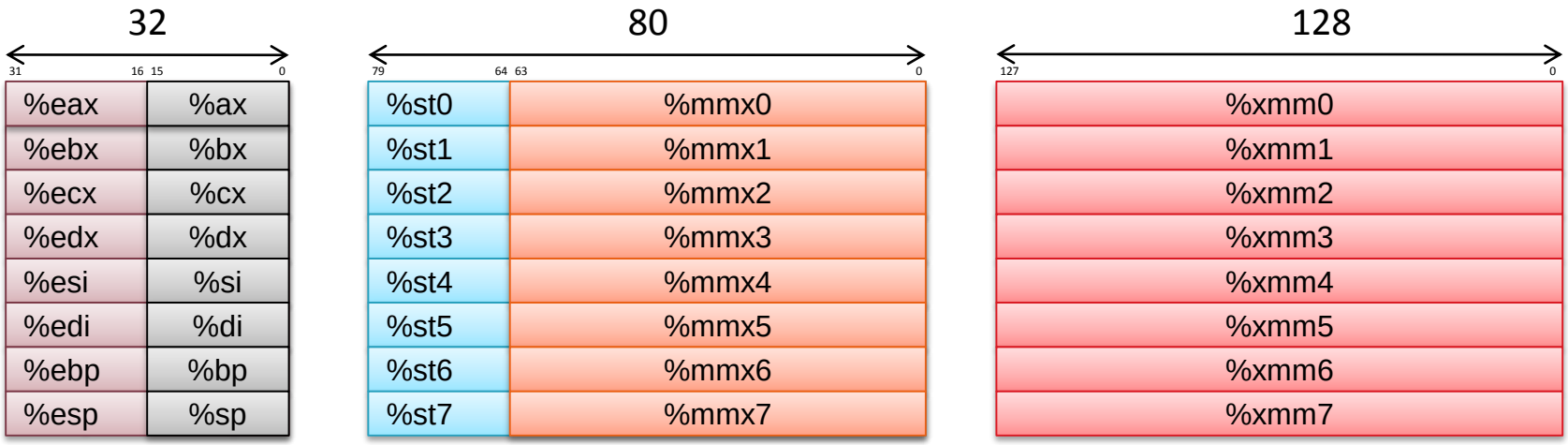


Instrucciones SIMD de 64 bits para enteros (8x8, 4x16, 2x32). Los registros %mmx están mapeados sobre los registros de punto flotante (sobre los 64 bits de mantisa)

Instrucciones SIMD

Extensiones del lenguaje máquina:

- i8086 (1977), i8087 (1980), IA-32 (1985), MMX (1997), SSE (1999)

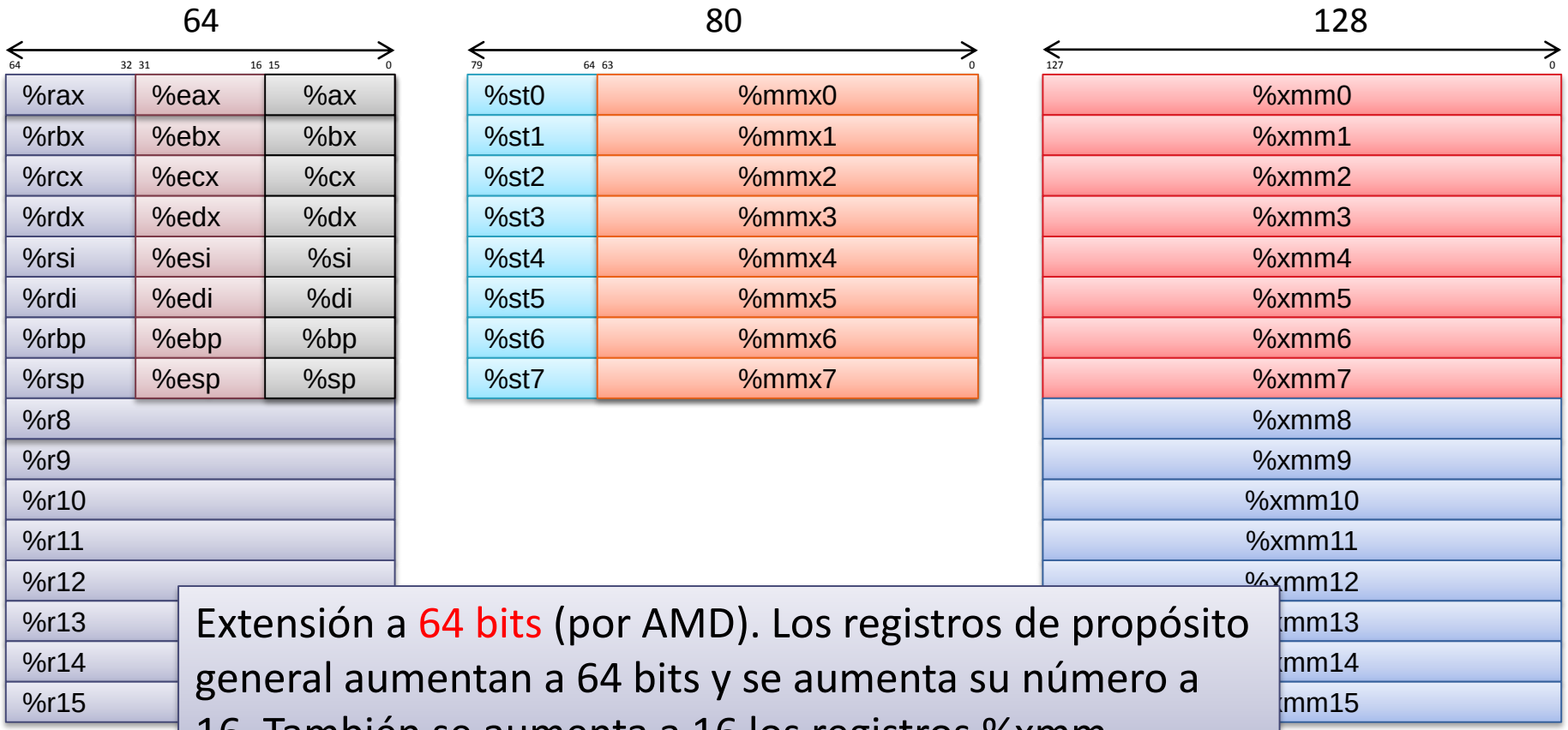


Nuevas instrucciones SIMD de 128 bits para enteros (16x8,8x16,4x32,2x64) y punto flotante (4x32,2x64). Usan un banco de registros separado. Se han ido incorporando nuevas instrucciones en distintas generaciones: SSE, SSE2, SSE3, SSSE3, SSE4 (4.1, 4.2, 4a), …….

Instrucciones SIMD

Extensiones del lenguaje máquina:

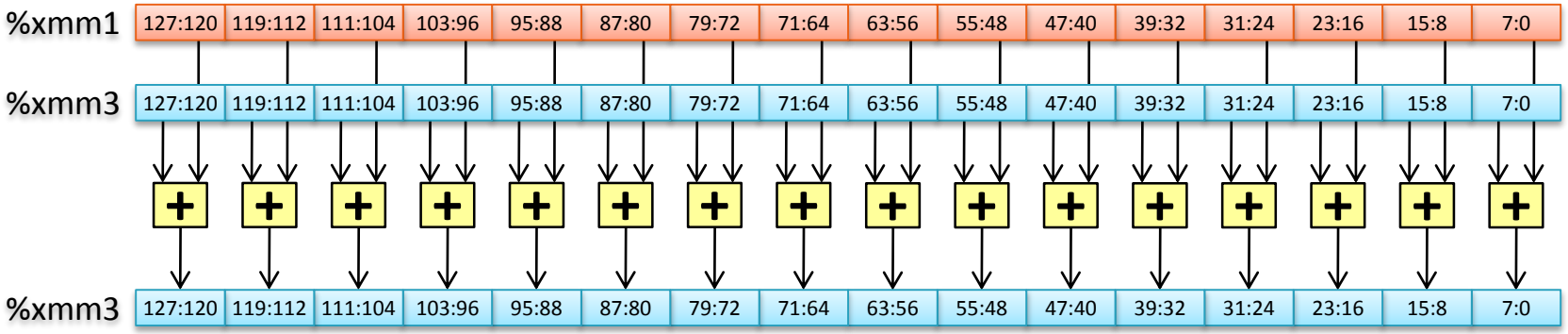
- i8086 (1977), i8087 (1980), IA-32 (1985), MMX (1997), SSE (1999), AMD-64 (2003)



Extensión a 64 bits (por AMD). Los registros de propósito general aumentan a 64 bits y se aumenta su número a 16. También se aumenta a 16 los registros %xmm

Instrucciones SIMD

- Ejemplo: **`paddb %xmm1, %xmm3`** # add packed byte integers



Instrucciones	Descripción	Notas
PADDB	add packed byte integers	16x8 bits
PADDW	add packed word integers	8x16 bits
PADDD	add packed doubleword integers	4x32 bits
PADDQ	add packed quadword integers	2x64 bits
ADDPS	add packed single-precission floating-point values	4x32 bits
ADDPD	add packed double-precission floating-point values	2x64 bits

Instrucciones SIMD

- **Ejemplo: Calcular el vector máximo de 2 vectores de caracteres**

```
void maxv(char a[], b[], max[]) {
  int i;
  for (i=0;i<8000;i++) {
    if (a[i]>b[i]) max[i]=a[i];
    else max[i]=b[i];
  }
}
```

```
for:  cmpl $8000, %esi
      jge fin
      movdqa (%ebx, %esi),%xmm0
      pmaxsb (%ecx, %esi),%xmm0
      movdqa %xmm0, (%edx, %esi)
      addl $16, %esi
      jmp for
```

9 instrucciones cada iteración

vs

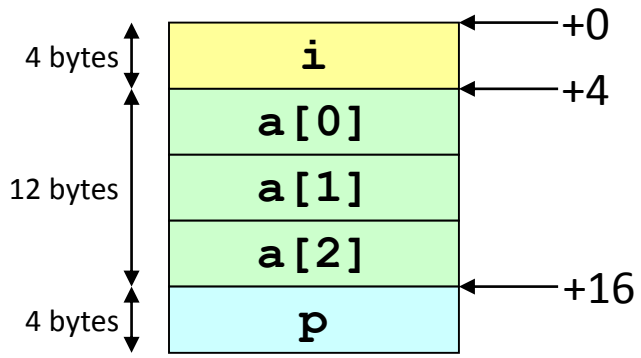
7 instrucciones cada 16 iteraciones

```
maxv: pushl %ebp
      movl %esp, %ebp
      ; Salvar Registros
      movl 8(%ebp),%ebx      ; ebx ← @a
      movl 12(%ebp),%ecx     ; ecx ← @b
      movl 16(%ebp),%edx     ; edx ← @max
      xorl %esi,%esi         ; i ← 0
for:  cmpl $8000, %esi       ; i < 8000
      jge fin
      movb (%ebx, %esi),%al  ; al ← a[i]
      cmpb (%ecx, %esi),%al  ; a[i] > b[i]
      jle else
      movb %al, (%edx, %esi); max[i] ← a[i]
      jmp cont
else: movb (%ecx, %esi), %al;
      movb %al, (%edx, %esi); max[i] ← b[i]
cont: incl %esi              ; i++
      jmp for
fin:  ; Restaurar Registros
      popl %ebp
      ret
```

Tipos de datos estructurados

Estructuras (struct)

- conjunto heterogéneo de datos
  - ✓ almacenados de forma contigua en memoria
  - ✓ referenciados por su nombre



Ejemplo:

```
typedef struct {
  int i;
  int a[3];
  int *p;
} X;
X S;

Init(&S);
```

Ejemplo:

```
void Init (X *S) {
  (*S).i = 1;
  S->a[2] = 0;
  S->p = &(*S).a[0];
}
```

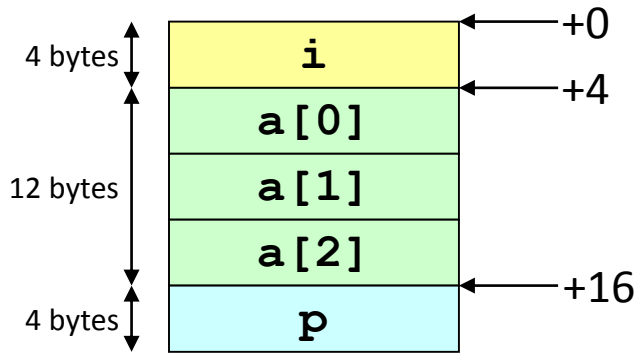
Tipos de datos estructurados

- Estructuras (struct)

Ejemplo:

```
typedef struct {
   int i;
   int a[3];
   int *p;
} X;
X S;

Init(&S);
```



Traducción:

```
Init:  push %ebp
       movl %esp,%ebp
       movl 8(%ebp),%edx
       movl $1,(%edx)
       movl $0,12(%edx)
       leal 4(%edx),%eax
       movl %eax,16(%edx)
       popl %ebp
       ret
```

Ejemplo:

```
void Init (X *S) {
   (*S).i = 1;
   S->a[2] = 0;
   S->p = &(*S).a[0];
}
```

Alineamiento de datos

- **Alineamiento de datos**
  - Un tipo de dato primitivo requiere k bytes
    - ✓ La dirección debe ser múltiplo de k
    - ✓ En algunas máquinas es obligatorio. Aconsejable en x86
    - ✓ Trato distinto en Windows y Linux
- **Motivación para alinear datos**
  - Accesos a memoria por longword o quadwords alineados
  - Accesos no alineados pueden provocar que un mismo dato se encuentre en 2 líneas de cache diferentes.
  - Memoria virtual: problemas si el dato está entre dos páginas
- **Compilador**
  - Inserta “espacios” en la estructura para asegurar que los datos están alineados.

Alineamiento de datos

- **Alineamiento en linux-32 (gcc):**
  - **char** (1 byte): alineado a 1-byte (no hay restricciones en la @)
  - **short** (2 bytes): alineado a 2-bytes (el bit más bajo de la @ debe ser 0)
  - **int** (4 bytes): alineado a 4-bytes (los 2 bits más bajos de la @ deben ser 00)
  - **puntero** (4 bytes): alineado a 4-bytes
  - **double** (8 bytes): alineado a 4-bytes
  - **Long double** (12 bytes): alineado a 4-bytes
- **Offsets dentro de una estructura:**
  - deben satisfacer los requerimientos de alineamiento de sus elementos
- **Dirección de la estructura**
  - Cada estructura tiene un requerimiento de alineamiento k
  - k = el mayor de los alineamientos de cualquier elemento
  - La @ inicial y el tamaño de la estructura debe ser múltiplo de k

Alineamiento de datos

- **Diferencias linux-64:**
  - **double** (8 bytes): alineado a 8-bytes.
  - **long double** (16 bytes): alineado a 16-bytes.
  - **puntero** (8 bytes): alineado a 8-bytes.
- **Diferencias windows-32:**
  - **double** (8 bytes): alineado a 8-bytes.
  - **long double** (10 bytes): alineado a 2-bytes

Alineamiento de datos: Ejemplo

Ejemplo:

```
struct S1 {
    char c;    // alineado a 1
    int i[2]; // alineado a 4
    double v; // alineado a 4  (8 en Linux-64)
}*p;
```

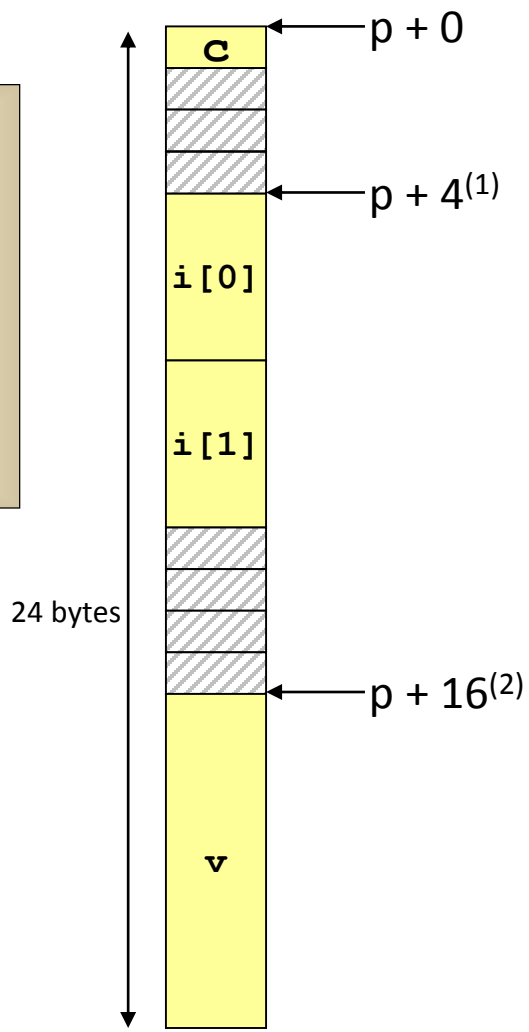
- (Linux-32) k = max (1,4,4) = 4
- (Linux-64) k = max (1,4,8) = 8

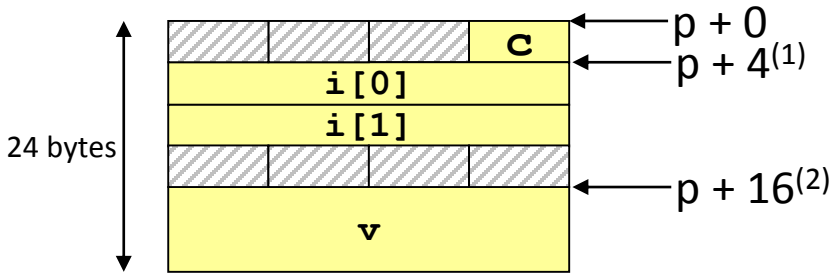
Alineamiento de datos: Ejemplo en Linux-64

- **k = 8 debido al elemento *double***

Ejemplo:

```
struct S1 {
    char c;
    int i[2];
    double v;
}*p;
```





¡La dirección de inicio y el tamaño de la estructura han de ser **múltiplo de 8**!

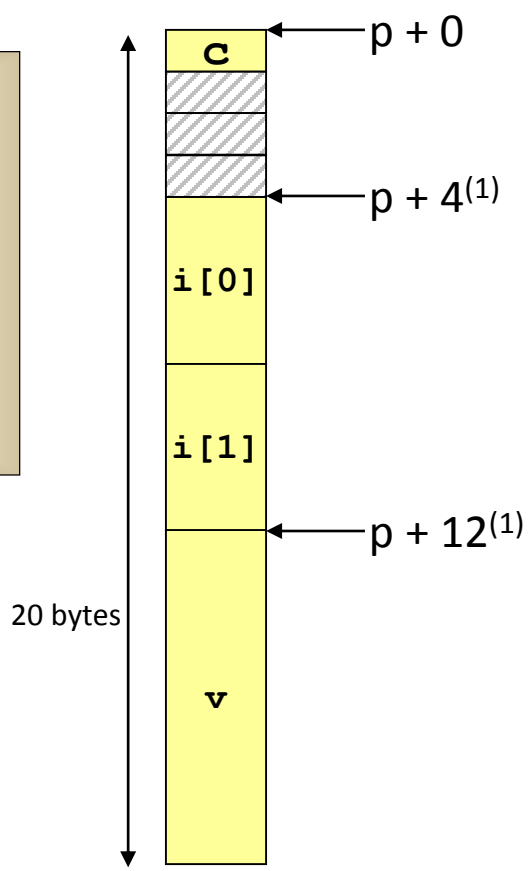
(1) Múltiplo de 4
(2) Múltiplo de 8

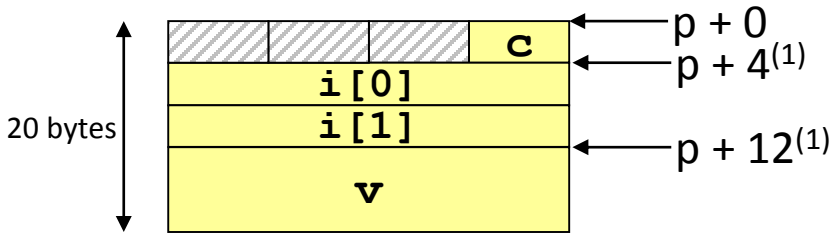
Alineamiento de datos: Ejemplo en Linux-32

- **k = 4 debido a que el elemento *double* se trata a nivel de alineamiento como un elemento de 4 bytes.**

Ejemplo:

```
struct S1 {
    char c;
    int i[2];
    double v;
}*p;
```





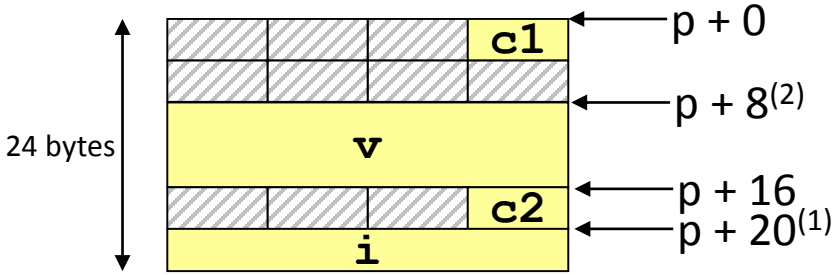
¡La dirección de inicio y el tamaño de la estructura han de ser **múltiplo de 4**!

(1) Múltiplo de 4

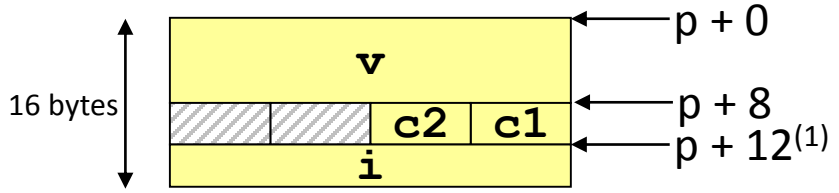
Alineamiento y orden de los elementos

- **El orden de los elementos de una estructura influye en su tamaño.**
  - Ejemplo en Linux-64

```
struct S4 {
    char c1;
    double v;
    char c2;
    int i;
} *p;
```




```
struct S5 {
    double v;
    char c1;
    char c2;
    int i;
} *p;
```



(1) Múltiplo de 4
(2) Múltiplo de 8

¡La dirección de inicio y el tamaño de la estructura han de ser **múltiplo de 8**!

Alineamiento de datos

- **El orden de los elementos de una estructura influye en su tamaño.**

```
struct S4 {
    char c1;
    double v;
    char c2;
    int i;
} *p;
```

```
struct S5 {
    double v;
    char c1;
    char c2;
    int i;
} *p;
```

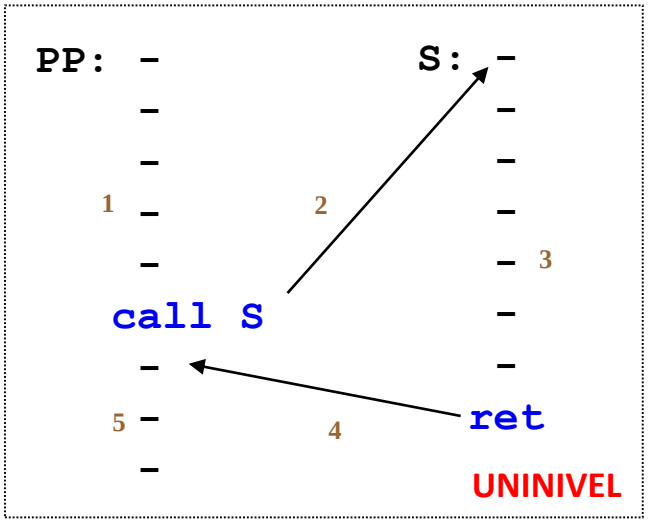
- El programador de C puede reordenar los elementos de la estructura para minimizar el espacio ocupado.
- Sin embargo, el programador de ensamblador **NO** puede realizar esta optimización cuando enlaza ensamblador con C.

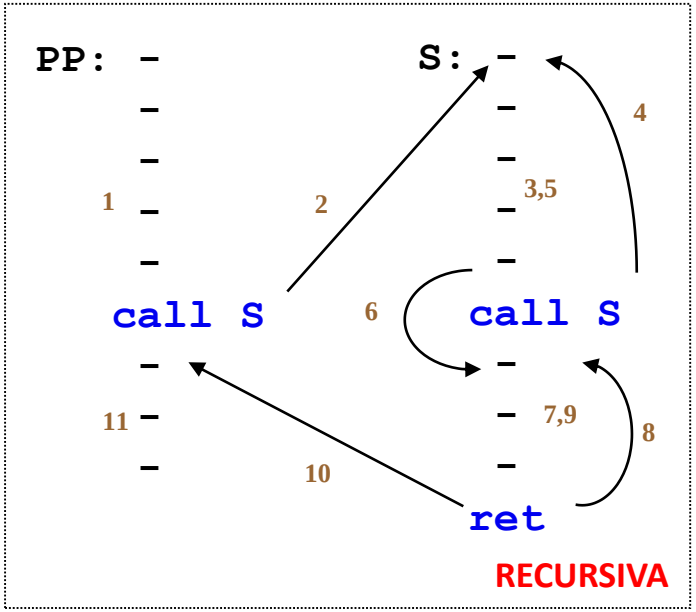
GESTION DE SUBRUTINAS

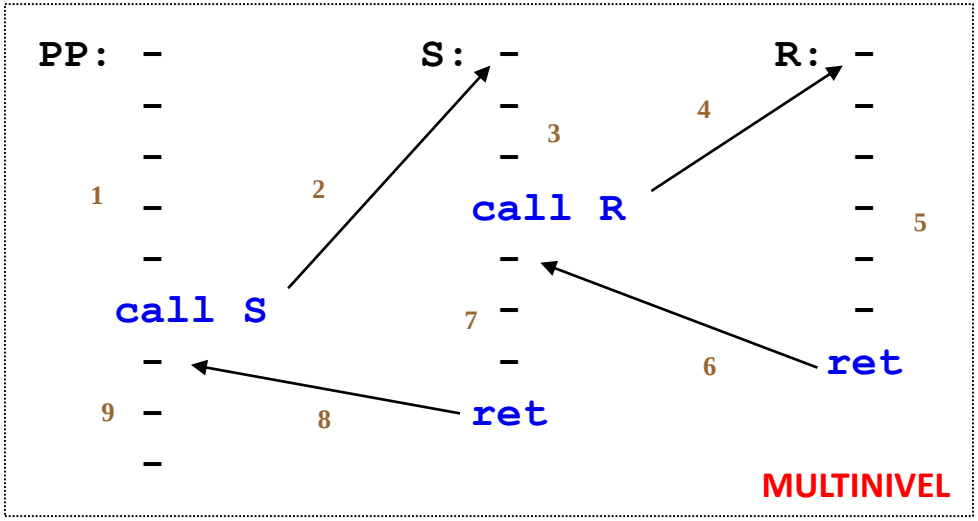
Definiciones

- **Subrutina: Conjunto de instrucciones de LM que realiza una tarea específica y que puede ser activada (llamada) desde cualquier punto de un programa o desde la propia subrutina**
- **Activación interna: la llamada se hace desde la propia subrutina**
- **Activación externa: la llamada se hace desde el programa principal o desde otra subrutina**
- **Entre el 5 y el 10% de las instrucciones que ejecuta un procesador son llamadas o retornos de subrutinas.**
- **Clasificación de las subrutinas**
  - Uninivel
  - Multinivel
  - Recursivas
  - Reentrantes
  - No reentrantes

Tipos de subrutinas







¿Vale la pena usar subrutinas?

Ventajas del uso de subrutinas

- El código ocupa menos espacio en memoria
- El código está más estructurado
  - ✓ facilidad de depuración
  - ✓ facilidad de expansión o modificación
  - ✓ posibilidad de usar librerías públicas
- El LM refleja la idea fundamental de los lenguajes estructurados de alto nivel: la existencia de funciones y procedimientos

Inconvenientes del uso de subrutinas

- El tiempo de ejecución de los programas aumenta debido a:
  - ✓ la ejecución de las instrucciones de llamada y retorno de subrutina
  - ✓ el paso de parámetros
- La complejidad del procesador es mayor porque debe añadirse hardware específico para la gestión eficiente de subrutinas

Terminología

- Parámetros
  - ✓Valor
  - ✓Referencia
- Variables locales
- Invocación
- Retorno resultado
- Cuerpo subrutina

```
int DOT(int v1[], int v2[], int N) {
  int i, sum;
  sum = 0;
  for (i=0; i<N; i++)
    sum += v1[i] * v2[i];
  return sum;
}

void PDOT(int M[10][10], int *p) {
  int i;
  *p = 0;
  for (i=0; i<10; i++)
    *p += DOT(&M[0][0],&M[i][0],10);
}
```

Convenciones en C-Linux 32 bits

- Los **parámetros** se pasan **por la pila de derecha a izquierda**
  - Los vectores y matrices siempre se pasan por referencia
  - Los structs se pasan por valor, no importa el tamaño
  - Los parámetros de tipo caracter (1 byte) ocupan 4 bytes
  - Los parámetros de tipo short (2 bytes) ocupan 4 bytes
- Las **variables locales** están alineadas en la pila con la misma convención que en un struct
  - Char en cualquier dirección
  - Short en direcciones múltiplos de 2
  - Integer en direcciones múltiplos de 4
  - El tamaño del conjunto de variables locales debe ser múltiplo de 4 para que la pila quede bien alineada
- Los registros
  - %ebp, %esp se salvan siempre implícitamente en la gestión de subrutinas
  - %ebx, %esi, %edi se han de salvar si son modificados
  - %eax, %ecx, %edx se pueden modificar en el interior de una subrutina.
    Si es necesario, el LLAMADOR ha de salvarlos
- Los **resultados** se devuelven siempre en **%eax**
- La pila debe quedar siempre alineada a 4

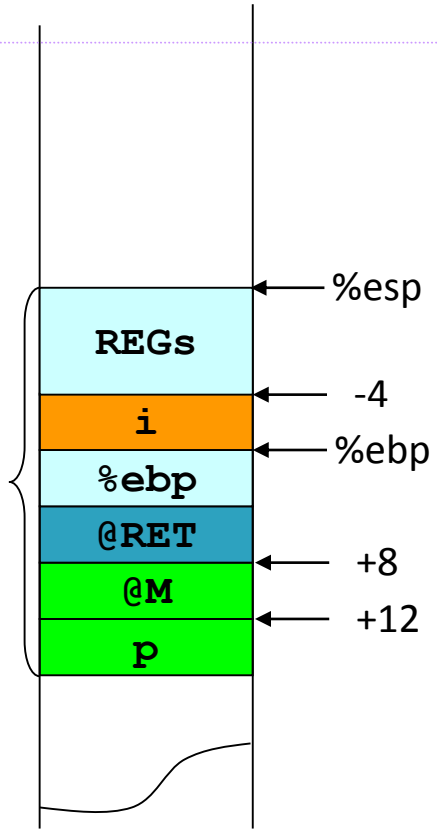
Bloque de activación

PILA

```
void PDOT(int M[10][10], int *p) {
  int i;
  *p = 0;
  for (i=0; i<10; i++)
    *p += DOT(&M[0][0],&M[i][0],10);
}
```

```
        {código llamador PDOT}
        empilar parámetros PDOT
        call PDOT
        ...

PDOT:   pushl %ebp
        movl %esp, %ebp
        subl $4, %esp
        salvar registros
        -
        -
        -
```

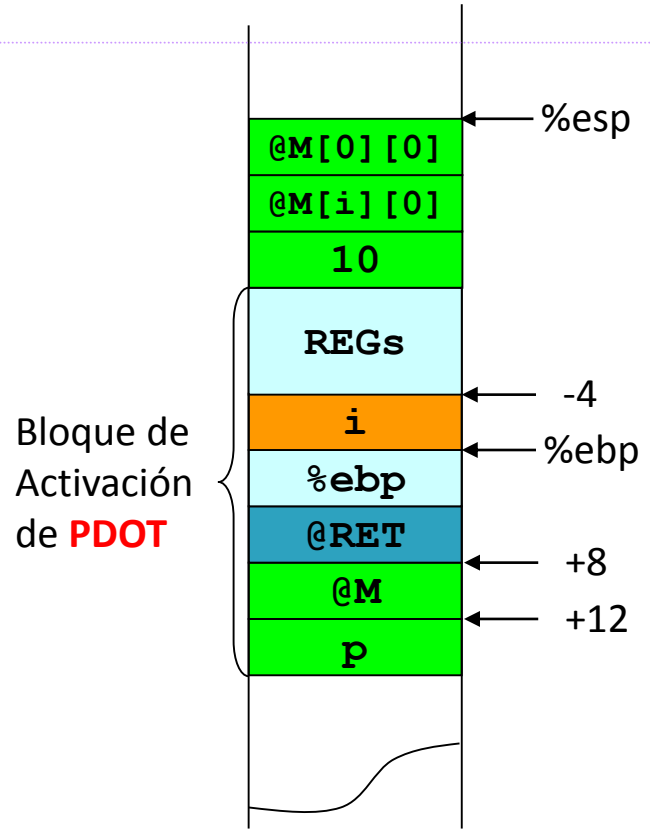


Ejemplo de Subrutinas

1. Paso de parámetros

```
PDOT:   -
        -
        -
        pushl $10
        imull $10,-4(%ebp),%edx
        movl 8(%ebp),%ebx
        leal (%ebx,%edx,4),%eax
        pushl %eax
        pushl %ebx
```

```
void PDOT(int M[10][10], int *p) {
  int i;
  *p = 0;
  for (i=0; i<10; i++)
    *p += DOT(&M[0][0],&M[i][0],10);
}
```

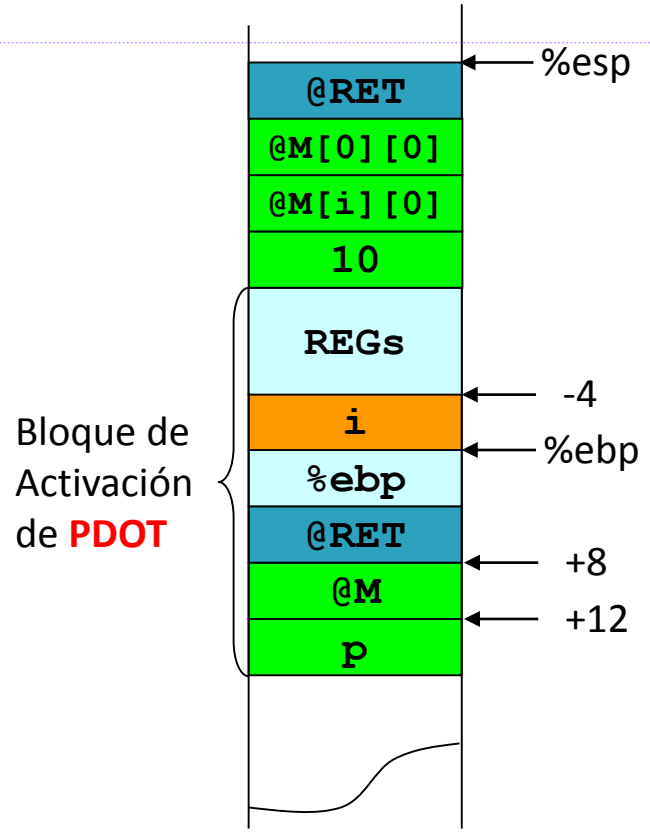


Ejemplo de Subrutinas

2. Llamada a la subrutina

```
PDOT:   -
        -
        -
        pushl $10
        imull $10,-4(%ebp),%edx
        movl 8(%ebp),%ebx
        leal (%ebx,%edx,4),%eax
        pushl %eax
        pushl %ebx
        call DOT
```

```
void PDOT(int M[10][10], int *p) {
  int i;
  *p = 0;
  for (i=0; i<10; i++)
    *p += DOT(&M[0][0],&M[i][0],10);
}
```



Ejemplo de Subrutinas

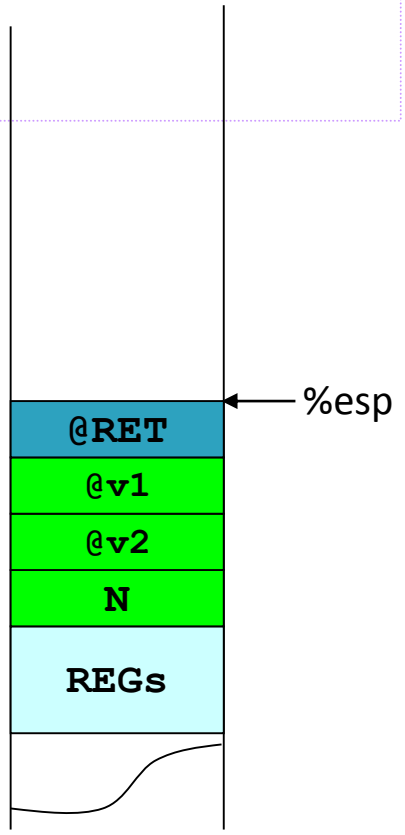
2. Saltamos a la subrutina

DOT:

```
int DOT(int v1[], int v2[], int N) {
  int i, sum;

  sum = 0;
  for (i=0; i<N; i++)
    sum += v1[i] * v2[i];

  return sum;
}
```



Ejemplo de Subrutinas

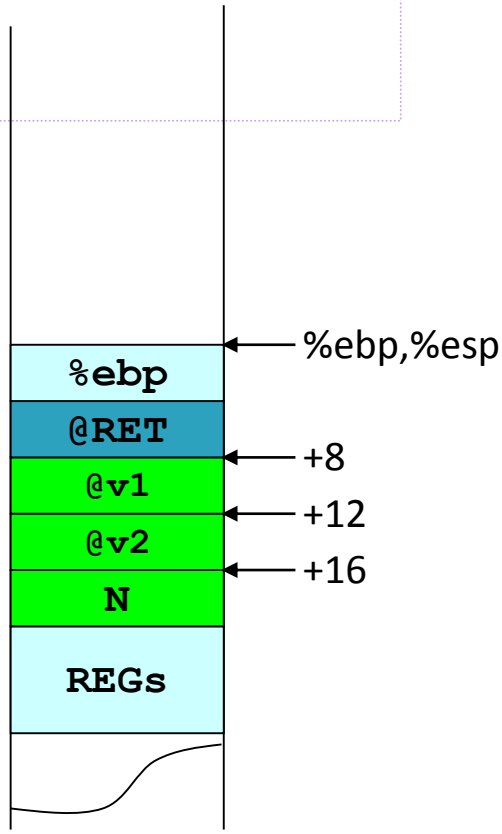
3. Enlace dinámico y puntero al bloque de activación

```
DOT: pushl %ebp
     movl %esp, %ebp
```

```
int DOT(int v1[], int v2[], int N) {
  int i, sum;

  sum = 0;
  for (i=0; i<N; i++)
    sum += v1[i] * v2[i];

  return sum;
}
```



Ejemplo de Subrutinas

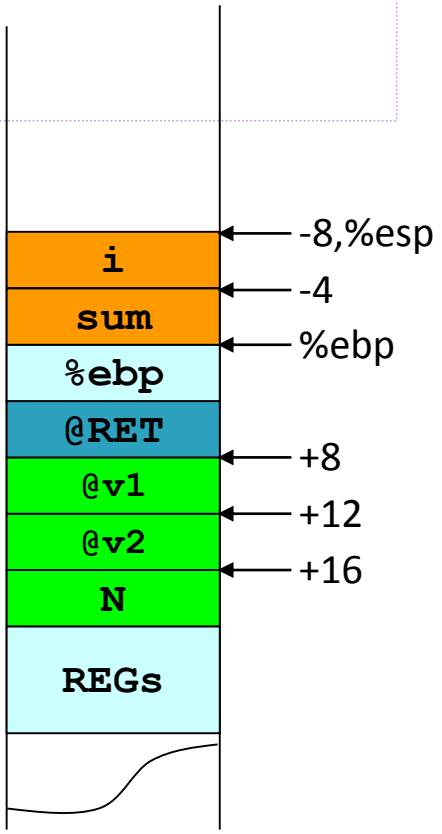
4. Reserva espacio para variables locales

```
DOT: pushl %ebp
     movl %esp, %ebp
     subl $8, %esp
```

```
int DOT(int v1[], int v2[], int N) {
  int i, sum;

  sum = 0;
  for (i=0; i<N; i++)
    sum += v1[i] * v2[i];

  return sum;
}
```



Ejemplo de Subrutinas

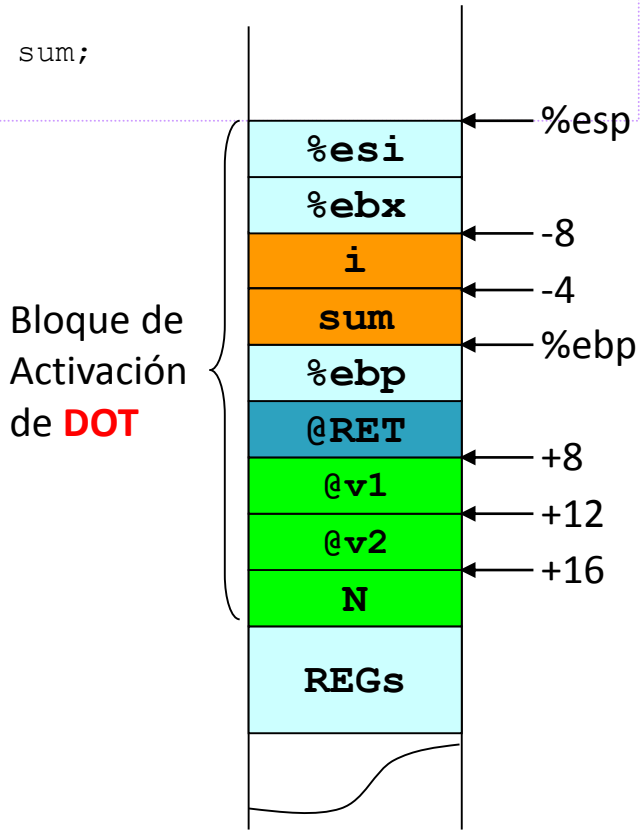
5. Salvar estado del llamador

```
DOT: pushl %ebp
     movl %esp, %ebp
     subl $8, %esp
     pushl %ebx
     pushl %esi
```

```
int DOT(int v1[], int v2[], int N) {
  int i, sum;

  sum = 0;
  for (i=0; i<N; i++)
    sum += v1[i] * v2[i];

  return sum;
}
```



Ejemplo de Subrutinas

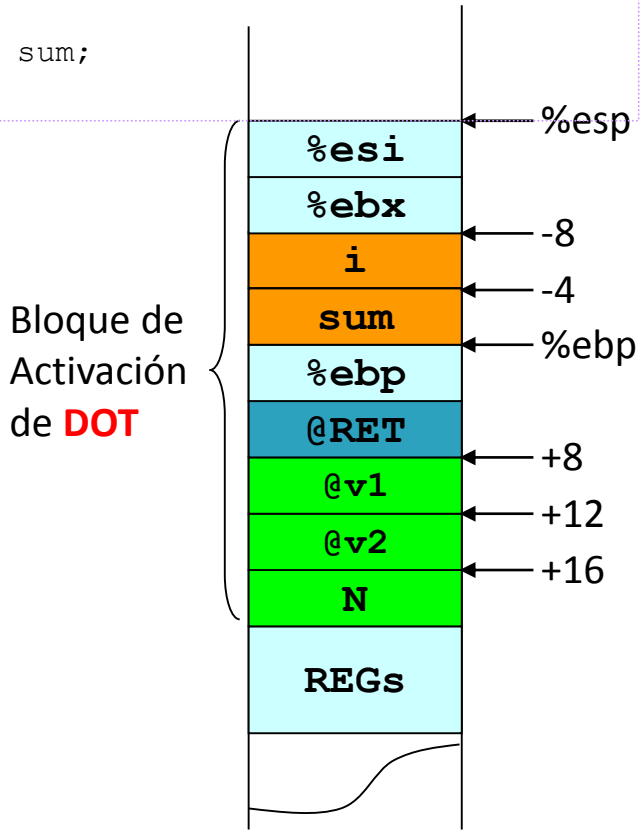
6. Cuerpo subrutina

```
DOT: pushl %ebp
     movl %esp, %ebp
     subl $8, %esp
     pushl %ebx
     pushl %esi
     movl 8(%ebp),%ebx
     movl 12(%ebp),%esi
     movl $0,-4(%ebp)
     xorl %edx,%edx
for: cmpl 16(%ebp),%edx
     jge end
     movl (%esi,%edx,4),%ecx
     imull (%ebx,%edx,4),%ecx
     addl %ecx,-4(%ebp)
     incl %edx
     jmp for
end:
```

```
int DOT(int v1[], int v2[], int N) {
  int i, sum;

  sum = 0;
  for (i=0; i<N; i++)
    sum += v1[i] * v2[i];

  return sum;
}
```



Ejemplo de Subrutinas

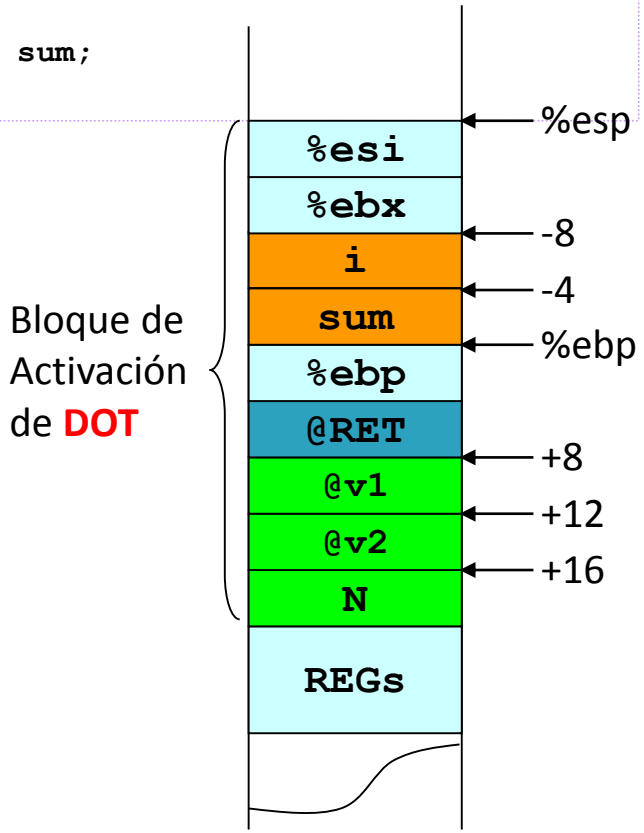
7. Mover resultado a %eax

```
DOT: pushl %ebp
     movl %esp, %ebp
     subl $8, %esp
     pushl %ebx
     pushl %esi
     movl 8(%ebp),%ebx
     movl 12(%ebp),%esi
     movl $0,-4(%ebp)
     xorl %edx,%edx
for: cmpl 16(%ebp),%edx
     jge end
     movl (%esi,%edx,4),%ecx
     imull (%ebx,%edx,4),%ecx
     addl %ecx,-4(%ebp)
     incl %edx
     jmp for
end: movl -4(%ebp),%eax
```

```
int DOT(int v1[], int v2[], int N) {
  int i, sum;

  sum = 0;
  for (i=0; i<N; i++)
    sum += v1[i] * v2[i];

  return sum;
}
```



Ejemplo de Subrutinas

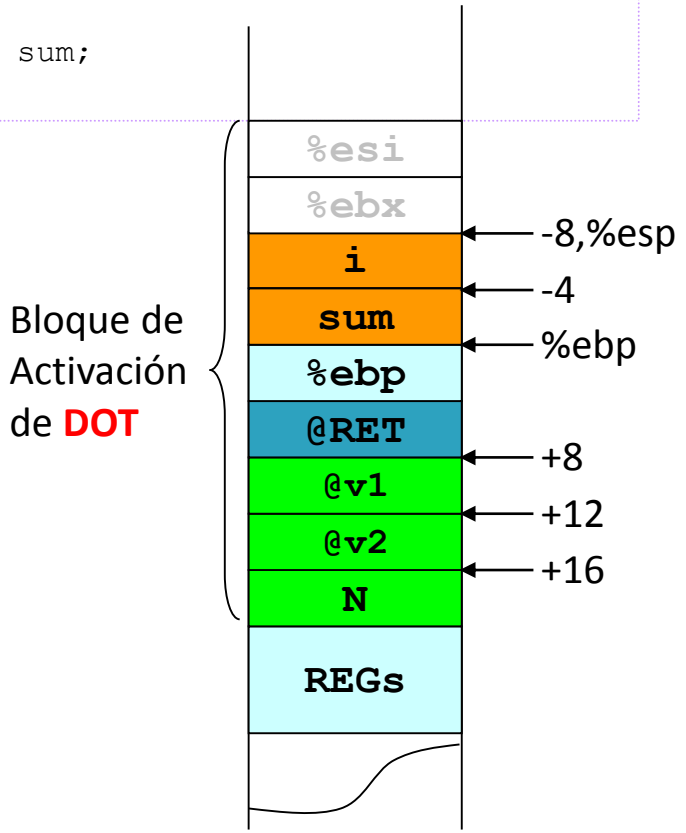
8. Restaurar estado llamador

```
DOT: pushl %ebp
     movl %esp, %ebp
     subl $8, %esp
     pushl %ebx
     pushl %esi
     movl 8(%ebp),%ebx
     movl 12(%ebp),%esi
     movl $0,-4(%ebp)
     xorl %edx,%edx
for: cmpl 16(%ebp),%edx
     jge end
     movl (%esi,%edx,4),%ecx
     imull (%ebx,%edx,4),%ecx
     addl %ecx,-4(%ebp)
     incl %edx
     jmp for
end: movl -4(%ebp),%eax
     popl %esi
     popl %ebx
```

```
int DOT(int v1[], int v2[], int N) {
  int i, sum;

  sum = 0;
  for (i=0; i<N; i++)
    sum += v1[i] * v2[i];

  return sum;
}
```



Ejemplo de Subrutinas

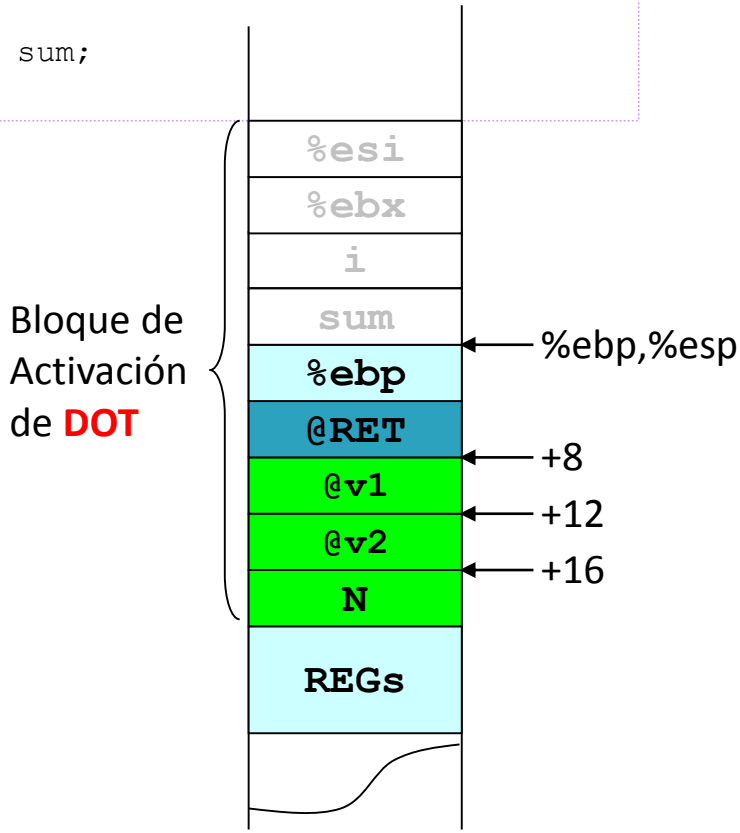
9. Eliminar variables locales

```
DOT: pushl %ebp
     movl %esp, %ebp
     subl $8, %esp
     pushl %ebx
     pushl %esi
     movl 8(%ebp),%ebx
     movl 12(%ebp),%esi
     movl $0,-4(%ebp)
     xorl %edx,%edx
for: cmpl 16(%ebp),%edx
     jge end
     movl (%esi,%edx,4),%ecx
     imull (%ebx,%edx,4),%ecx
     addl %ecx,-4(%ebp)
     incl %edx
     jmp for
end: movl -4(%ebp),%eax
     popl %esi
     popl %ebx
     movl %ebp,%esp
```

```
int DOT(int v1[], int v2[], int N) {
  int i, sum;

  sum = 0;
  for (i=0; i<N; i++)
    sum += v1[i] * v2[i];

  return sum;
}
```



Ejemplo de Subrutinas

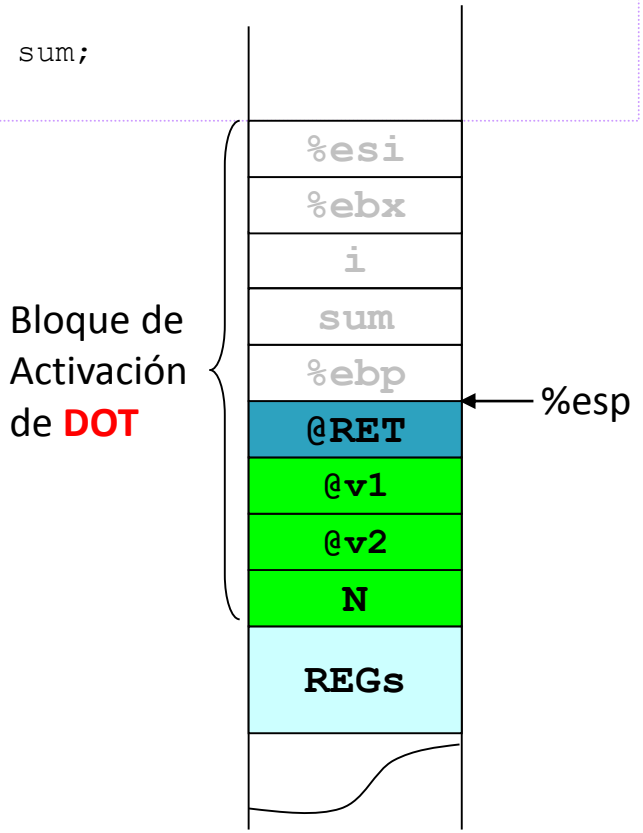
10. Deshacer enlace dinámico

```
DOT: pushl %ebp
     movl %esp, %ebp
     subl $8, %esp
     pushl %ebx
     pushl %esi
     movl 8(%ebp),%ebx
     movl 12(%ebp),%esi
     movl $0,-4(%ebp)
     xorl %edx,%edx
for: cmpl 16(%ebp),%edx
     jge end
     movl (%esi,%edx,4),%ecx
     imull (%ebx,%edx,4),%ecx
     addl %ecx,-4(%ebp)
     incl %edx
     jmp for
end: movl -4(%ebp),%eax
     popl %esi
     popl %ebx
     movl %ebp,%esp
     popl %ebp
```

```
int DOT(int v1[], int v2[], int N) {
  int i, sum;

  sum = 0;
  for (i=0; i<N; i++)
    sum += v1[i] * v2[i];

  return sum;
}
```



Ejemplo de Subrutinas

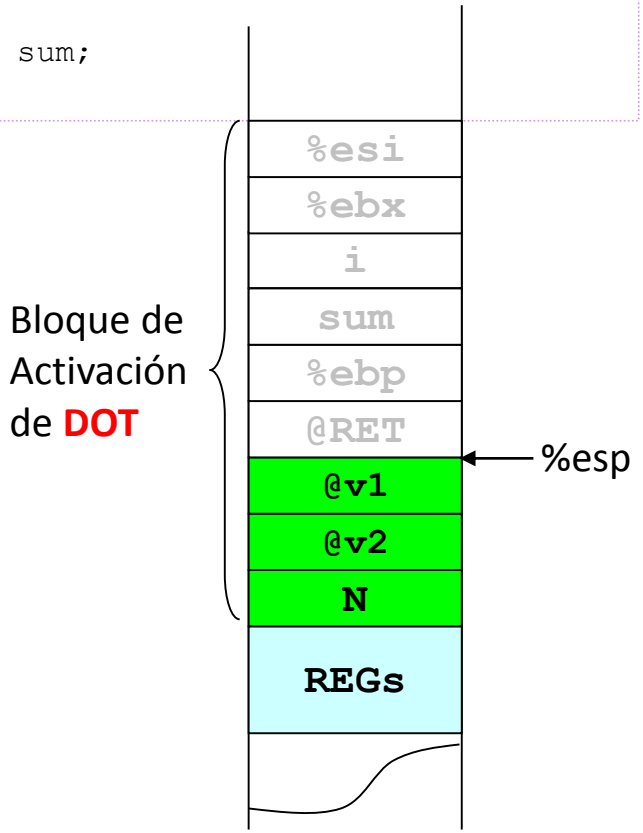
11. Retorno subrutina

```
DOT: pushl %ebp
     movl %esp, %ebp
     subl $8, %esp
     pushl %ebx
     pushl %esi
     movl 8(%ebp),%ebx
     movl 12(%ebp),%esi
     movl $0,-4(%ebp)
     xorl %edx,%edx
for: cmpl 16(%ebp),%edx
     jge end
     movl (%esi,%edx,4),%ecx
     imull (%ebx,%edx,4),%ecx
     addl %ecx,-4(%ebp)
     incl %edx
     jmp for
end: movl -4(%ebp),%eax
     popl %esi
     popl %ebx
     movl %ebp,%esp
     popl %ebp
     ret
```

```
int DOT(int v1[], int v2[], int N) {
  int i, sum;

  sum = 0;
  for (i=0; i<N; i++)
    sum += v1[i] * v2[i];

  return sum;
}
```

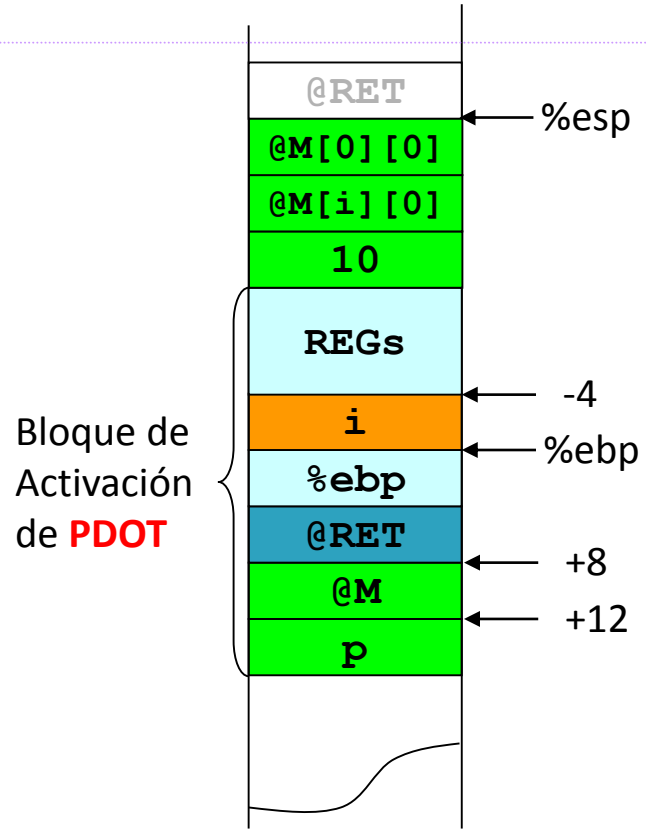


Ejemplo de Subrutinas

11. Volvemos a la subrutina

```
void PDOT(int M[10][10], int *p) {
  int i;
  *p = 0;
  for (i=0; i<10; i++)
    *p += DOT(&M[0][0],&M[i][0],10);
}
```

```
PDOT:   -
        -
        -
        pushl $10
        imull $10,-4(%ebp),%edx
        movl 8(%ebp),%ebx
        leal (%ebx,%edx,4),%eax
        pushl %eax
        pushl %ebx
        call DOT
```

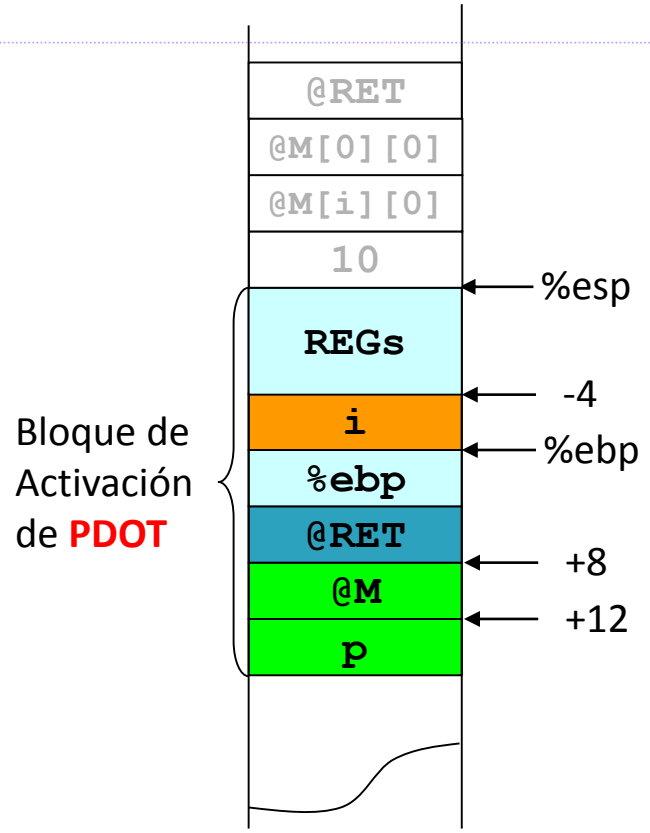


Ejemplo de Subrutinas

12. Eliminar parámetros

```
void PDOT(int M[10][10], int *p) {
  int i;
  *p = 0;
  for (i=0; i<10; i++)
    *p += DOT(&M[0][0],&M[i][0],10);
}
```

```
PDOT:   -
        -
        -
        pushl $10
        imull $10,-4(%ebp),%edx
        movl 8(%ebp),%ebx
        leal (%ebx,%edx,4),%eax
        pushl %eax
        pushl %ebx
        call DOT
        addl $12,%esp
```

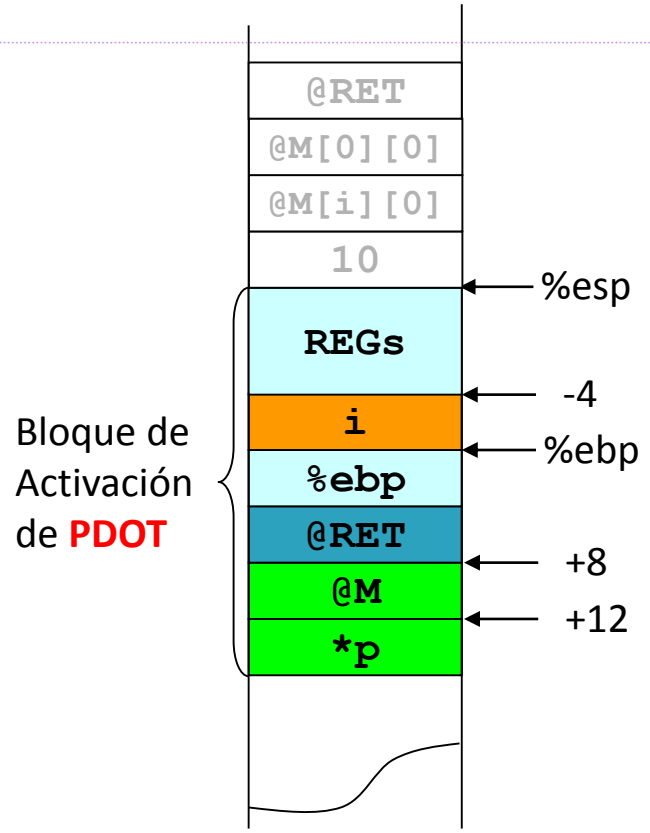


Ejemplo de Subrutinas

13. Recoger/usar resultado

```
void PDOT(int M[10][10], int *p) {
  int i;
  *p = 0;
  for (i=0; i<10; i++)
    *p += DOT(&M[0][0],&M[i][0],10);
}
```

```
PDOT:   -
        -
        -
        pushl $10
        imull $10,-4(%ebp),%edx
        movl 8(%ebp),%ebx
        leal (%ebx,%edx,4),%eax
        pushl %eax
        pushl %ebx
        call DOT
        addl $12,%esp
        movl 12(%ebp),%ebx
        addl %eax,(%ebx)
```



Pasos en la Gestión de subrutinas

PDOT:

-

-

-

1 Paso de parámetros

2 llamada subrutina

12 elimina parámetros

13 Recoger/usar resultado

-

-

DOT:

3 Enlace dinámico, puntero bloque de activación

4 Reserva espacio variables locales

5 Salvar estado llamador

6 Cuerpo subrutina

7 Mover resultado a eax

8 Restaura estado

9 elimina variables locales

10 Deshacer enlace dinámico

11 retorno de subrutina

Gestión de Registros

- Los registros **%eax, %ecx, %edx se pueden modificar** en el interior de una subrutina.
  Si es necesario, el **LLAMADOR** ha de salvarlos

```
PDOT:   -
        -
        movl $0, %ecx
for:    cmpl $10, %ecx
        jge ffor
        pushl $10
        imull $10,%ecx,%edx
        movl 8(%ebp),%ebx
        leal (%ebx,%edx,4),%eax
        pushl %eax
        pushl %ebx
        call DOT ;puede machacar %ecx
        addl $12,%esp
        movl 12(%ebp),%ebx
        addl %eax,(%ebx)
        incl %ecx
        jmp for:
ffor:   -
        -
```

Usamos %ecx como contador de bucle

```
void PDOT(int M[10][10], int *p) {
  int i;  // almacenamos i en %ecx
  *p = 0;
  for (i=0; i<10; i++)
    *p += DOT(&M[0][0],&M[i][0],10);
}
```

```
DOT: pushl %ebp
     movl %esp, %ebp
     subl $8, %esp
     pushl %ebx
     pushl %esi
     …
     movl (%esi,%edx,4),%ecx
     imull (%ebx,%edx,4),%ecx
     …
     popl %esi
     popl %ebx
     movl %ebp,%esp
     popl %ebp
     ret
```

Gestión de Registros

- Los registros **%eax, %ecx, %edx se pueden modificar** en el interior de una subrutina.
  Si es necesario, el **LLAMADOR** ha de salvarlos

```
PDOT:   -
        movl $0, %ecx
for:    cmpl $10, %ecx
        jge ffor
        pushl %ecx
        pushl $10
        imull $10,%ecx,%edx
        movl 8(%ebp),%ebx
        leal (%ebx,%edx,4),%eax
        pushl %eax
        pushl %ebx
        call DOT ;puede machacar %ecx
        addl $12,%esp
        movl 12(%ebp),%ebx
        addl %eax,(%ebx)
        popl %ecx
        incl %ecx
        jmp for:
ffor:
```

Mejor aun usar %ebx, %esi o %edi en lugar de %ecx como contador de bucle

```
void PDOT(int M[10][10], int *p) {
  int i;  // almacenamos i en %ecx
  *p = 0;
  for (i=0; i<10; i++)
    *p += DOT(&M[0][0],&M[i][0],10);
}
```

```
DOT: pushl %ebp
     movl %esp, %ebp
     subl $8, %esp
     pushl %ebx
     pushl %esi
     …
     movl (%esi,%edx,4),%ecx
     imull (%ebx,%edx,4),%ecx
     …
     popl %esi
     popl %ebx
     movl %ebp,%esp
     popl %ebp
     ret
```

Gestión de Registros

PDOT:

-

-

0 Salvar registros llamador

1 Paso de parámetros

2 llamada subrutina

12 elimina parámetros

13 Recoger/usar resultado

14 restaurar registros llamador

-

-

DOT:

3 Enlace dinámico, puntero bloque de activación

4 Reserva espacio variables locales

5 Salvar estado llamador

6 Cuerpo subrutina

7 Mover resultado a eax

8 Restaura estado

9 elimina variables locales

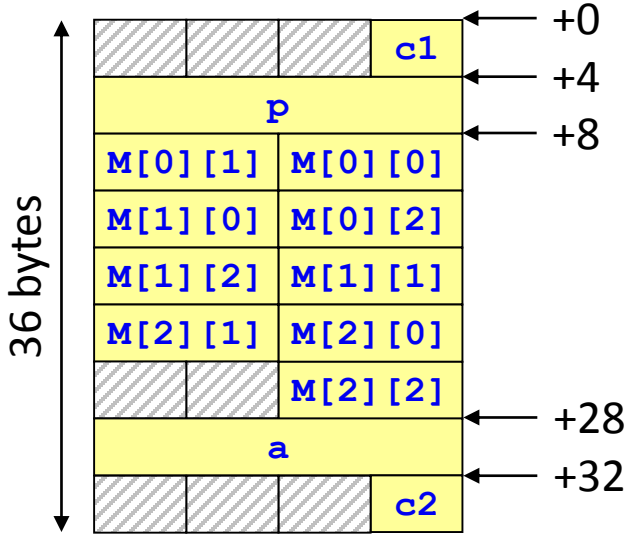
10 Deshacer enlace dinámico

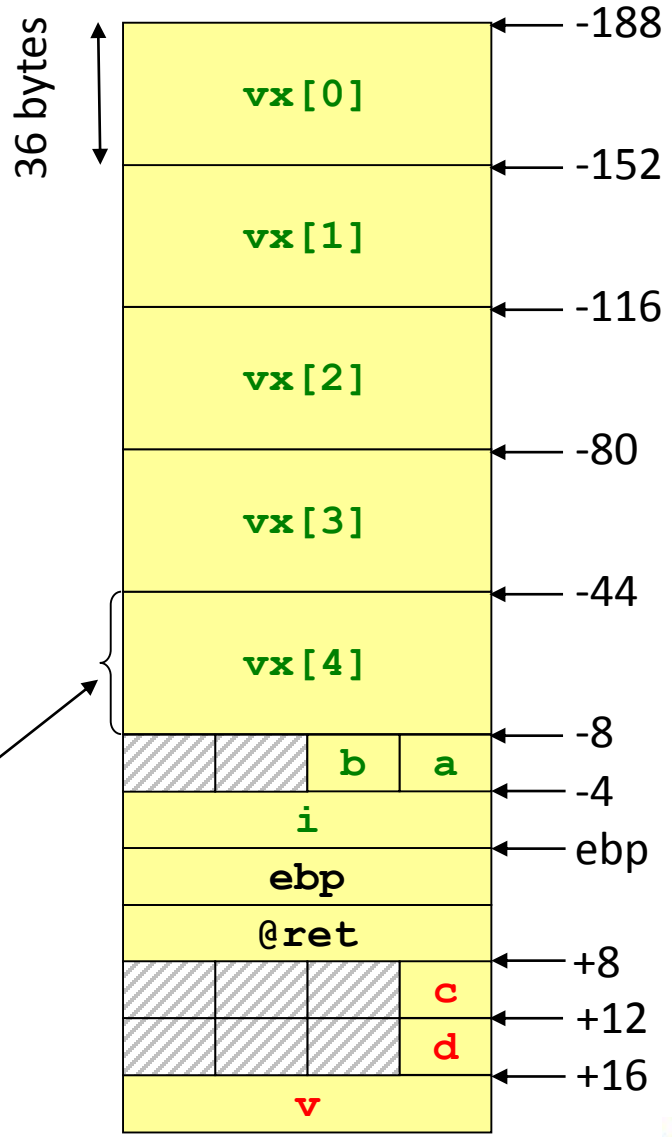
11 retorno de subrutina

Ejemplos de Subrutinas y Structs

```
typedef struct {
  char c1;
  char *p;
  unsigned short M[3][3];
  int a;
  char c2;
} X;

int rut (char c, char d, int v[4])
{
  X vx[5];
  char a;
  char b;
  int i;
  ...
}
```





Ejemplos de Subrutinas y Structs

```
typedef struct {
  char c1;
  char *p;
  unsigned short M[3][3];
  int a;
  char c2;
} X;

int rut2 (X sx, X *px)
{
...
}
```