Lenguaje Máquina

Agustín Fernández, Josep Llosa, Fermín Sánchez

Estructura de Computadors II
Departament d'Arquitectura de Computadors
Facultat d'Informàtica de Barcelona

Índice

Introducción

- Una instrucción de Lenguaje Máquina es una tira de bits que especifica:
  - Código de operación:
    - Operación a realizar
  - Modos de direccionamiento:
    - Dónde localizar los operandos
    - Dónde dejar el resultado
  - Secuenciamiento:
    - Cuál es la siguiente instrucción a ejecutar
    - Generalmente es implícito
    - Instrucciones de Salto

Introducción

- Criterios de diseño de las instrucciones:
  - Instrucciones cortas mejor
  - Longitud suficiente
  - Longitud múltiplo de la unidad mínima de secuenciamiento/direccionamiento (byte)
- Formato de las instrucciones:
  - Fijo (RISC)
    - Decodificación rápida y fácil
    - Desperdicio de memoria
  - Variable (IA32)
    - Aprovechamiento de memoria
    - Decodificación compleja

Introducción

- Tipos de arquitecturas:

#operandos	#operandos memoria	Tipos de Arquitectura		Ejemplos
0	0	Pila		HP3000
1	1	Acumulador		I8080
2	1	General Purpose Registers (GPR)	Registro/Memoria	IA32
3	0		Load/Store	IBM Power
3	3		Memoria/Memoria	VAX-11

  - Arquitecturas GPR:
    - #ops en memoria = 0 — Load / Store
    - #ops en memoria < #operandos — Registro / Memoria
    - #ops en memoria = #operandos — Memoria / Memoria
    - 2 operandos — op1 <- op1 (*op*) op2
    - 3 operandos — op1 <- op2 (*op*) op3

Introducción

- Ejemplo: d = (a + b*c) /a

Pila	**Acumulador**	**Registro Memoria**	**Memoria Memoria**	**Load Store**
push b	load b	mov b, r0	mul b, c, r0	load b, r0
push c	mul c	mul c, r0	...	load c, r1
mul	...	...		mul r0, r1, r2
...				...

Acabadlo vosotros

Introducción

- Calculad "A = (A-B*C)/(D+E)" en las 5 máquinas descritas:

PILA

```
add/sub/mul/div    # pila[sp+1]=pila[sp] op pila[sp+1]; sp=sp+1;
push @             # sp=sp-1; pila[sp]=M[@]
pop @              # M[@]=pila[sp]; sp=sp+1
```

ACUMULADOR

```
add/sub/mul/div @  # ACC=ACC op M[@]
load @             # ACC=M[@]
store @            # M[@]=ACC
```

Se pueden almacenar resultados temporales en la variable tmp

REGISTRO / MEMORIA

```
add/sub/mul/div @,Ri        # Ri = Ri op M[@]
add/sub/mul/div Ri,@        # M[@] = M[@] op Ri
mov @, Ri/ mov Ri, @        # mov fuente, destino; destino = fuente
```

MEMORIA / MEMORIA

```
add/sub/mul/div op1, op2, p3 # op3 = op1 op op2 ; opi = Ri or @
```

LOAD / STORE

```
add/sub/mul/div Ri, Rj, Rk  # Rk = Ri op Rj
load @, Ri                  # Ri = M[@]
store Ri, @                 # M[@] = Ri
```


Introducción

- Niveles de un computador:
  - Lenguajes de alto nivel (LAN)
  - Lenguaje máquina (LM)
  - Implementacion hardware (HARD)

  LAN --compilación--> **LM** --interpretación--> **HARD**

- Como salvar el desnivel LAN ---> HARD:
  - CISC: Complex Instruction Set Computer
    - Instrucciones LM de alto contenido semántico
    - Esfuerzo en interpretación (Microcódigo)
  - RISC: Reduced (complexity) Instruction Set Computer
    - Instrucciones LM de bajo contenido semántico
    - Esfuerzo en compilación

Introducción

- Estilos de diseño: RISC versus CISC

RISC	CISC
Instrucciones sencillas	Instrucciones complejas
Instrucciones de tamaño fijo	Instrucciones de tamaño variable
Pocos formatos de instrucciones	Muchos formatos dependiendo de los operandos
Operandos siempre en registros ADD Ri,Rj,Rk #Rk ← Ri + Rj	Operandos en registro o memoria ADDL %eax, (%ebx)
Load / Store	Registro – Memoria Memoria – Memoria
Banco de registros grande (≥32)	Históricamente pocos registros
Modos de direccionamiento simples (Ri+despl.)	Modos de direccionamiento complejos

Introducción

- Estilos de diseño: RISC versus CISC

	M68000 (CISC)	RISC II
Año	1980	1983
#Instrucciones	56	39
#Registros	15	137
@Modos @	14	3
#Transistores	68000	41000
%Control on die	50%	6%
Tiempo diseño	30 meses	19 meses

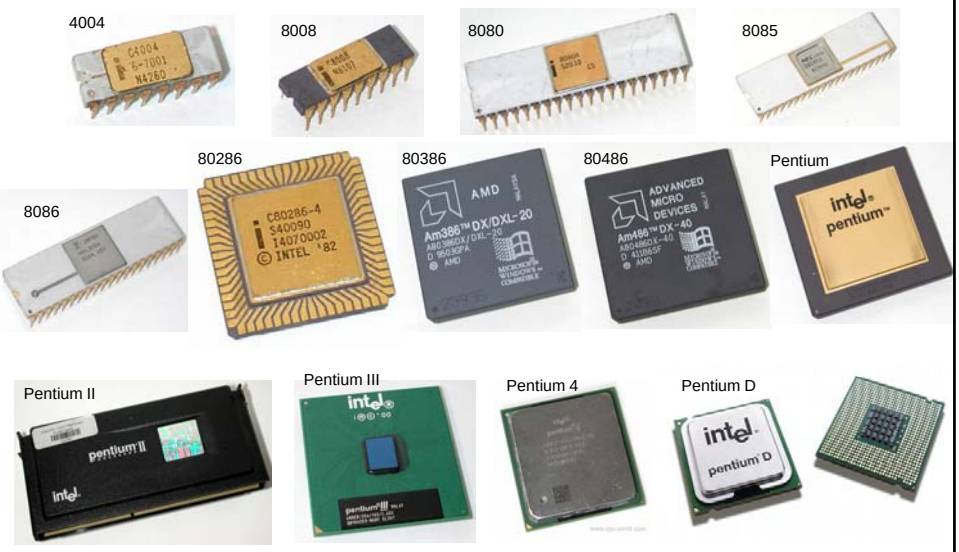
Arquitectura básica del IA32

- Referencias históricas
- Visión del programador en ensamblador IA32
  - Espacio de memoria y registros
  - Tipos de datos básicos
  - Modos de direccionamiento
- Instrucciones de movimiento de datos
- Instrucciones aritméticas
- Instrucciones lógicas
  - Códigos de condición
  - Instrucciones SetX
- Instrucciones de salto

Referencias históricas

Año	Modelo	Tam. Palabra	Trans.	Freq.	Tamaño Memoria		
1971	i4004	4 bits	2.300	0,75 MHz	640 B		
1972	i8008	8 bits	3.500	0,5 MHz	16 KB		
1974	i8080	8 bits	6.000	2 MHz	64 KB		
1978	i8086	16 bits	29K	5-10 MHz	1 MB		
1979	i8088	16 bits	29K	5-8 MHz	1 MB	MS-DOS	IBM PC
1982	i80286	16 bits	134K	6-12,5 MHz	16 MB	Windows	IBM AT
1985	i80386	32 bits	275K	16-33 MHz	4 GB	Linux	
1989	i80486	32 bits	1,2M	25-100 MHz			8 KB
1993	Pentium	32 bits	3,2M	60-200 MHz			16 KB
1995	Pentium PRO	32 bits	5,5M	150-200 MHz			16 KB
1997	Pentium MMX	32 bits	4,5M	166-300 MHz		MMX	32 KB
1997	Pentium II	32 bits	7,5M	233-450 MHz			32 KB
1999	Pentium III	32 bits	28M	0,45-1,4 GHz		SIMD	256 KB
2001	Pentium 4	32 bits	55M	1,4-3,8 GHz			512 KB
2005	Pentium D	64 bits	230M	2,8-3,6 GHz	1 TB	Dual Core	1-2 MB
2006	Core 2	64 bits	291M	1,6-3,2 GHz	1 TB	Dual Core	2-4 MB

Referencias históricas



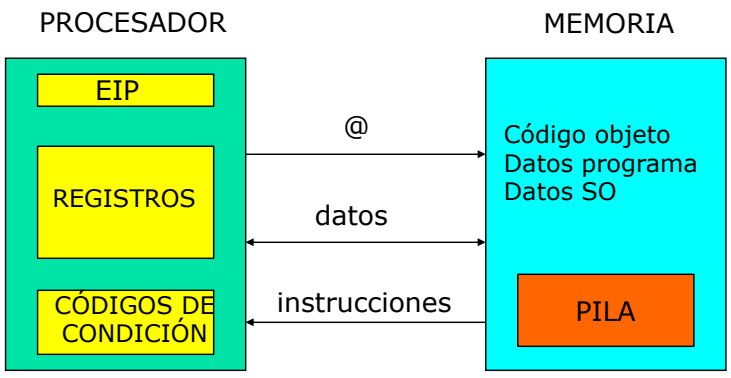
Referencias históricas

- Diseño evolucionario
  - Desde 1978 con el i8086
  - Se añaden + y mejores prestaciones con el tiempo
  - Todavía soporta características del i8086 (incluso obsoletas)
- Otros fabricantes de IA32: AMD (Advanced Micro Devices)
  - Competencia desde el 80386 (am386)
  - En general un poco más lento, mucho más barato
    - En los últimos años la competencia es feroz: AMD e Intel se alternan en prestaciones/precio
  - Versión propia de 64 bits "copiado por Intel"
- CISC
  - Sólo un pequeño subconjunto de instrucciones se encuentran en programas Linux
  - ¿Se pueden conseguir la prestaciones de un RISC?

Referencias históricas

- Características de un CISC
  - Instrucciones pueden referenciar diferentes tipos de operando
    - inmediato, registros, memoria
  - Instrucciones aritméticas pueden leer/escribir en memoria
  - Referencias a memoria pueden suponer cálculos complejos
    - Rb + S*Ri + D
  - Instrucciones pueden tener diferente longitud
- Primer "RISC" x86: Pentium Pro (P6)
  - Anunciado en Febrero 95
  - Base del Pentium II, Pentium III y primer Celeron
  - Pentium 4 basado en una idea similar, pero diferentes detalles
  - Traduce dinámicamente instrucciones x86 a un formato más ancho y simple (MicroOperaciones)
  - Ejecuta hasta 5 instrucciones en paralelo
  - Pipeline profundo (latencia 12-18 ciclos)

Visión del programador



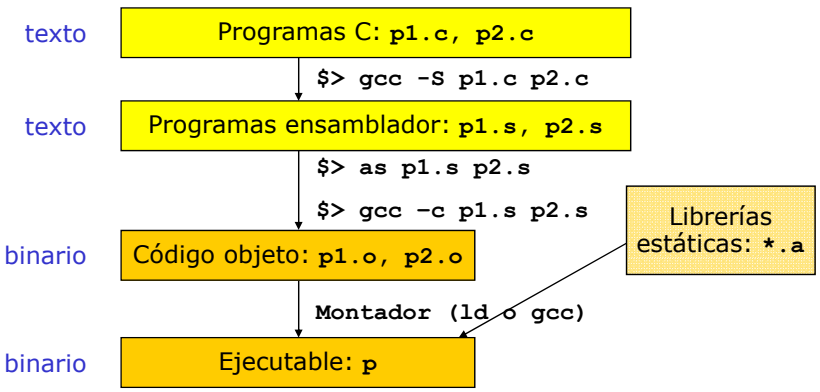
Visión del programador

- 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
  - Código, datos usuario, datos SO
  - Pila para soportar gestión de subrutinas

Compilación: C - código objeto

Programas C: `p1.c, p2.c`

`$> gcc -O p1.c p2.c -o p` (usa optimizaciones -O)



Características del ensamblador

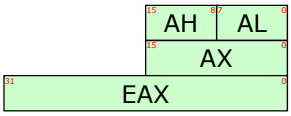
- Tipos de datos básicos
  - Enteros
    - dato de 1, 2 ó 4 bytes
    - datos y direcciones (punteros)
  - Reales (coma flotante): 4, 8 ó 10 bytes
  - No incluye tipos estructurados
    - Se codifican como datos almacenados de forma contigua
- Operaciones primitivas
  - Operaciones aritméticas sobre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

Visión del programador

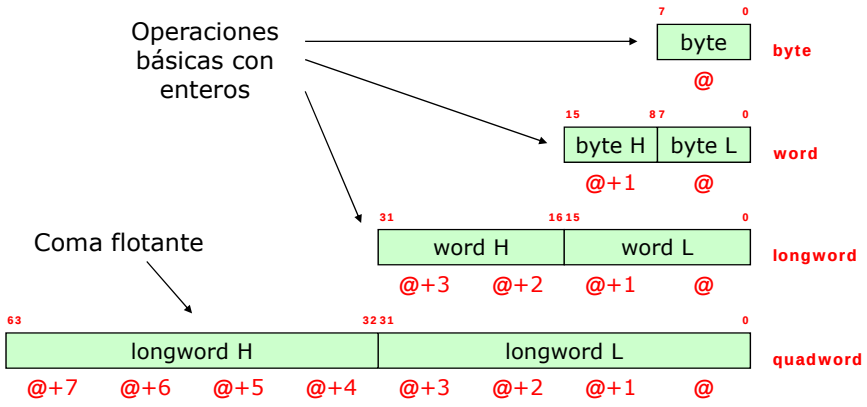
- ¿Qué necesitamos estudiar?
  - Espacio de memoria
  - Registros disponibles
  - Repertorio de instrucciones: qué hacen, cómo se codifican, cuánto tardan
  - Tipos y representación de los datos
  - Modos de direccionamiento
  - Secuenciamiento de instrucciones
  - Comunicación con el exterior

Visión del programador

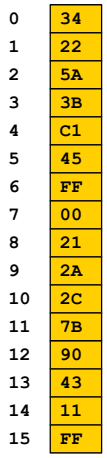
- Espacio de memoria
  - Espacio lineal de 2^{32} posiciones de 1 byte
  - Modo protegido / Modelo plano de memoria/ Little endian
- Registros disponibles

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
%flags			Palabra de estado

Tipos de datos básicos



Tipos de datos básicos



byte 8: 0x21

byte 3:

word 8:

word 3:

longword 8:

longword 3:

quadword 8:

quadword 3:

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: $-2.147.483.648 \leq x \leq 2.147.483.647$

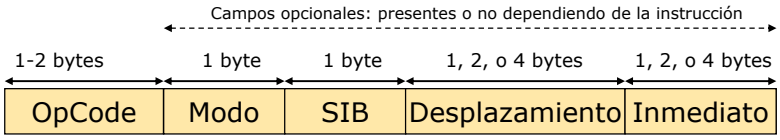
Modos de direccionamiento

- 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) :
  (%eax, %ebx,4) :
  (, %ebx, 4) :
  12(%eax) :
  (%eax) :
  3(%eax, %esi,2) :
  4:
  $4:
  %eax:
  %al:

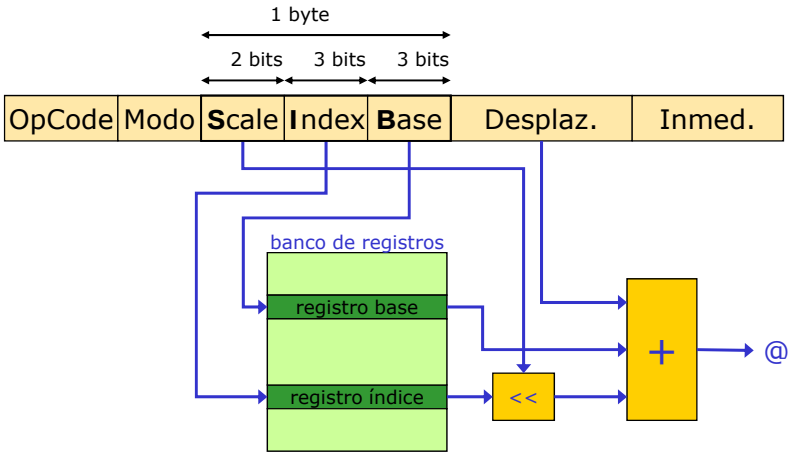
Codificación de las instrucciones



Formato General de las instrucciones

- **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

- movl op1, op2 op2 ← op1
- movw op1, op2
- movb op1, op2
  - ¡No se pueden realizar transferencias memoria-memoria!

- ejemplos
  - movl $0x3040, %eax
  - movl %esp, %ebp
  - movb (%ebx, %ecx), %ah
  - movw $17, %ax

Instrucciones de movimiento de datos

- movsbl op1, op2 op2 ← ExtSign(op1), op1 es byte
- movswl op1, op2 op2 ← ExtSign(op1), op1 es word
- movzbl op1, op2 op2 ← ExtZero(op1), op1 es byte
- movzwl op1, op2 op2 ← ExtZero(op1), op1 es word

Instrucciones de movimiento de datos

- Ejemplos:

  %edx = 0x2312ABF1
  %eax = 0x32573144

- movb %dh, %al: %eax = 0x325731AB
- movsbl %dh, %eax:
- movzbl %dh, %eax:
- movswl %dx, %eax:

Instrucciones de movimiento de datos

- Instrucciones de manejo de la pila
  - pushl op1 %esp ← %esp - 4; M[%esp] ← op1
  - pushw op1 %esp ← %esp - 2; M[%esp] ← op1
  - popl op1 op1 ← M[%esp]; %esp ← %esp + 4
  - popw op1 op1 ← M[%esp]; %esp ← %esp + 2

Instrucciones de movimiento de datos

- leal op1, op2 op2 ← &op1

- Usos
  - Cálculo de direcciones sin efectuar referencias a memoria.
    Ej: traducción de p = &x[i]
  - Cálculo de expresiones aritméticas de la forma x+k·y, con k=1,2,4,8
- Ejemplos
  - leal 6(%eax), %edx
  - leal (%eax, %ecx), %edx
  - leal 9(%eax, %ecx, 4), %edx
  - leal 0xA(, %ecx, 4), %edx

Punteros en C

- Un gran porcentaje de errores de programación en C están relacionados con los punteros
- Un puntero es una dirección a un elemento del tipo indicado
- Todos los punteros tienen el mismo tamaño: 4 bytes

```
int x;                  int *px;
short int y;            short int *py;
unsigned i;             unsigned *pi;
unsigned short j;       unsigned short *pj;
```


Punteros en C

- Ejemplos de errores

```
int x;
int *xp;

x=5;
*xp=5; ¡ERROR! Puntero no inicializado

xp=&x;
*xp=5; ¡CORRECTO!
```


Traducción C - ensamblador

```
int exchange (int *xp, int y)
{
   int x = *xp;
   *xp = y;
   return x
}

// Llamada:
b = exchange(&a, b)
```

```
# y → 12[%ebp]
# xp → 8[%ebp]

movl 8(%ebp),%eax  # leer xp en eax
movl (%eax),%ecx   # x = *xp
movl 12(%ebp),%edx # leer y en edx
movl %edx, (%eax)  # *xp = y
movl %ecx, %eax    # return x

Las funciones en C devuelven el
resultado en %eax
```


Traducción C - ensamblador

```
void swap (int *xp, int *yp)
{
   int t0 = *xp;
   int t1 = *yp;
   *xp = t1;
   *yp = t0;
}

// Llamada:
swap(&a, &b)
```

```
# yp → 12[%ebp]
# xp → 8[%ebp]

movl 12(%ebp),%ecx # leer yp en ecx
movl 8(%ebp),%edx  # leer xp en edx
movl (%ecx),%eax   # t1 = *yp
movl (%edx),%ebx   # t0 = *xp
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0

# eax es t1
# ebx es t0
```


Instrucciones aritméticas

- addl op1, op2 — op2 ← op2 + op1
- subl op1, op2 — op2 ← op2 - op1
- adcl op1, op2 — op2 ← op2 + op1 + CF
- sbbl op1, op2 — op2 ← op2 - op1 - CF

```
addb %dl, %al
adcb %dh, %ah
```

Es lo mismo que

```
addw %dx, %ax
```

- incl op1 — op1 ← op1 + 1
- decl op1 — op1 ← op1 - 1
- negl op1 — op1 ← -op1

Instrucciones aritméticas

- imul op1, op2 — op2 ← op1·op2
  - op2 siempre ha de ser un registro
- imul inm, op1, op2 — op2 ← inm·op1
  - inm es un operando inmediato
- imul op1 — %edx|%eax ← op1·%eax
  - op1 puede ser memoria o registro
- mull op1 — %edx|%eax ← op1·%eax
  - unsigned

Instrucciones aritméticas

- cltd — %edx |%eax ← ExtSign(%eax)
- idivl op1 — %eax ← (%edx | %eax) div op1
  %edx ← (%edx | %eax) %op1
- divl op1 — %eax ← (%edx | %eax) div op1
  %edx ← (%edx | %eax) %op1
  unsigned

¡CONSULTAR MANUAL!

Instrucciones lógicas

- Operaciones lógicas

andl op1, op2	op2 ← op2 & op1
orl op1, op2	op2 ← op2 | op1
xorl op1, op2	op2 ← op2 ^ op1
notl op1	op1 ← ~ op1

- Desplazamientos (k = %cl o inmediato)

sall k, op1	op1 ← op1 << k (aritmético)
shll k, op1	op1 ← op1 << k (lógico)
sarl k, op1	op1 ← op1 >> k (aritmético)
shrl k, op1	op1 ← op1 >> k (lógico)

Instrucciones lógicas

- Utilidades
  - Poner a 1 ó 0 bits determinados
  - complementar bits
- Ejemplos
  - Poner a 1 bit 8: orl $0x00000100, %eax
  - Poner a 0 bit 8: andl $0xFFFFFEFF, %eax
  - ExOr de bit 8: xorl $0x00000100, %eax

Traducción C - ensamblador

```
int logical (int x, int y)
{
   int t1, t2, mask, rval;
   t1 = x ^ y;
   t2 = t1 >> 17;
   mask = (1 << 13) - 7;
   rval = t2 & mask;
   return rval;
}

# x → 8[%ebp]
# y → 12[%ebp]
```

- Optimizad el código usando el mínimo número de registros.
- El valor de la máscara se calcula en tiempo de compilación.
- Se puede hacer en 4 instrucciones.

Traducción C - ensamblador

```
int arith (int x, int y, int z)
{
   int t1,t2,t3,t4,t5,rval;
   t1 = x + y;
   t2 = z + t1;
   t3 = x + 4;
   t4 = y * 48;
   t5 = t3 + t4;
   rval = t2 * t5;
   return rval;
}

# x → 8[%ebp]
# y → 12[%ebp]
# z → 16[%ebp]
```

```
movl 8(%ebp),%eax        # eax = x
movl 12(%ebp),%edx       # edx = y
leal (%eax,%edx),%ecx    # t1 = x+y
leal (%edx,%edx,2),%edx  # edx = 3*y
sall $4,%edx             # t4 = 48*y
addl 16(%ebp),%ecx       # t2 = z+t1
leal 4(%eax,%edx),%eax   # t5 = x+4+t4
imull %ecx,%eax          # rval = t5*t2
```

- rval es %eax
- ¡Sólo se hace 1 producto!

Códigos de condición

- Se activan implícitamente después de ejecutar una instrucción aritmética (p.e.: addl op1, op2 # t = op1+op2)
- ¡No se activan con leal!

- CF Carry Flag
  - Carry de la suma del bit 31. 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 (a>0 && b>0 && t<0) || (a<0 && b<0 && t>0)

Códigos de condición

- cmpl b, a
  - a-b : activa 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 a == b
- SF Sign Flag
  - SF = 1 si (a - b) < 0 (⇒ a<b)
- OF Overflow Flag
  - OF = 1 si (a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)

Instrucciones SetX

SetX	Condición	Descripción
sete	ZF	Igual / cero
setne	~ZF	No igual / no cero
sets	SF	Negativo
setns	~SF	No negativo
setg	~(SF^OF)&~ZF	Mayor (con signo)
setge	~(SF^OF)	Mayor o igual (con signo)
setl	(SF^OF)	Menor (con signo)
setle	(SF^OF)|ZF	Menor o igual (con signo)
seta	~CF&~ZF	Mayor (sin signo)
setb	CF	Menor (sin signo)

- Activan un byte en función de una combinación de códigos de condición
- No suelen usarse. Normalmente se usan directamente las instrucciones de salto condicional

Traducción C - ensamblador

```
int gt (int x, int y)
{
   return x > y;
}

// retorna 0 si x ≤ y
// retorna 1 o ≠0 si x > y

// Llamada:
gt(a, b);
gt(a, 45);
```

```
# x → 8[%ebp]
# y → 12[%ebp]

movl 8(%ebp),%eax   # eax = x
cmpl 12(%ebp),%eax  # x-y
setg %al            # al = (x>y)
movzbl %al, %eax    # return x > y
```


Instrucciones de salto

Jx	Condición	Descripción
jmp	1	Incondicional
je	ZF	Igual / cero
jne	~ZF	No igual / no cero
js	SF	Negativo
jns	~SF	No negativo
jg	~(SF^OF)&~ZF	Mayor (con signo)
jge	~(SF^OF)	Mayor o igual (con signo)
jl	(SF^OF)	Menor (con signo)
jle	(SF^OF)|ZF	Menor o igual (con signo)
ja	~CF&~ZF	Mayor (sin signo)
jb	CF	Menor (sin signo)

- Saltan a diferentes partes del código en función del valor de los códigos de condición

Traducción de sentencias C a ensamblador

- Sentencia condicional (IF-THEN-ELSE)

Ejemplo:

```
if (x>y)
  max = x;
else
  max = y;
```

Traducción genérica:

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

Suponiendo que:

```
x es %eax
y es %edx
max es %ecx
```

Se traduce como:

```
        cmpl %edx,%eax #eval. cond
        jle else
if:     movl %eax,%ecx #cuerpo if
        jmp endif
else:   movl %edx,%ecx #cuerpo else
endif:
```


Traducción de sentencias C a ensamblador

- Sentencia iterativa (DO-WHILE)

Ejemplo:

```
res=1;
do {
  res = res*x;
  x = x-1;
} while (x>1);
```

Traducción genérica:

```
do:
    CUERPO-DO
    evaluar cond
    jtrue do
enddo:
```

Suponiendo que:

```
x es %eax
res es %edx
```

Se traduce como:

```
    movl $1,%edx      #res = 1
do: imull %eax,%edx   #res = res*x
    decl %eax         #x = x-1
    cmpl $1,%eax      #eval cond
    jg do             #jtrue do
enddo:
```


Traducción de sentencias C a ensamblador

- Sentencia iterativa (WHILE)

Ejemplo:

```
res=1;
while (x>1){
  res = res*x;
  x = x-1;
}
```

Traducción genérica:

```
while: eval cond
       j(no cumple) end
       CUERPO-WHILE
       jmp while
end:
```

Suponiendo que:

```
x es %eax
res es %edx
```

Se traduce como:

```
       movl $1,%edx     #res = 1
while: cmpl $1,%eax     #eval cond
       jle end          #jfalse end
       imull %eax,%edx  #res *= x
       decl %eax        #x--
       jmp while
end:
```


Traducción de sentencias C a ensamblador

- Sentencia iterativa (FOR)

Ejemplo:

```
val = 1;
nval = 1;
for(i=1;i<N;i++){
  t = val+nval;
  val = nval;
  nval = t;
}
```

```
for (INI; COND; INC)
  cuerpo-FOR
```

Traducción genérica:

```
     INI
for: eval COND
     jfalse end
     cuerpo-FOR
     INC
     jmp for
end:
```


Traducción de sentencias C a ensamblador

- Sentencia iterativa (FOR)

Ejemplo:

```
val = 1;
nval = 1;
for(i=1;i<N;i++){
  t = val+nval;
  val = nval;
  nval = t;
}
```

Suponiendo que:

```
val es %eax
nval es %edx
t es %ecx
i es %edi
```

Se traduce como:

```
     movl $1,%eax          #val = 1
     movl $1,%edx          #nval = 1
     movl $1,%edi          #i = 1
for: cmpl $N,%edi          #eval cond
     jge end               #jfalse end
     leal (%eax,%edx),%ecx    #???
     movl %edx,%eax        #val = nval
     movl %ecx,%edx        #nval = t
     incl %edi             #i++
     jmp for
end:
```


Traducción de sentencias C a ensamblador

- Sentencia condicional (SWITCH)

```
int 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

Traducción de sentencias C a ensamblador

- Implementación con vector de punteros: pseudocódigo

```
        code *JT[7] = {L100, LDEF, L102, L103, L104, LDEF, L106};
        xi = x - 100;
        if ((xi<0)||(x>6)) jmp LDEF;
        goto JT[xi];
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;
```


Traducción de sentencias C a ensamblador

```
. section   . rodata
   . align 4
. L10:
   . long . L4
   . long . L9
   . long . L5
   . long . L6
   . long . L8
   . long . L9
   . long . L8
```

```
        leal -100(%edx),%eax  #eax=x-100
        cmpl $6,%eax
        ja . L9               #¿Es correcto?
        jmp *. L10(, %eax, 4)
. L4:   imull $13,%edx        # case 100
        jmp . L3
. L5:   addl $10,%edx         # case 102
. L6:   addl $11,%edx         # case 103
        jmp . L3
. L8:   imull %edx,%edx       # case 104,106
        jmp . L3
. L9:   xorl %edx,%edx         # default
. L3:   movl %edx,%eax
```

¡Completar con libro!

Tipos de datos estructurados

- Vectores
  - Definición en C: tipo nombre[tamaño (0..N-1)]
  - Almacenamiento en posiciones consecutivas de memoria
  - Ejemplos:

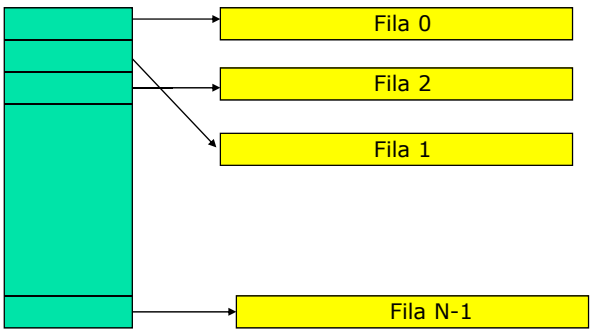
definición	tamaño elementos	tamaño total (bytes)	@ elemento i
char A[12]	1	12	@A + i
char *B[8]	4	32	@B + 4i
double C[6]	8	48	@C + 8i
int *D[5]	4	20	@D + 4i

Tipos de datos estructurados

- Matrices
  - Definición en C: tipo nombre[filas][columnas]
  - ejemplo: int A [4][3]
  - almacenamiento por filas en posiciones consecutivas de memoria
  - @ A[i][j] = @ A[0][0]+((i*col.)+j)*tam. elem.

Tipos de datos estructurados

- Matrices: otras posibilidades de almacenamiento
  - almacenamiento con punteros a filas



Tipos de datos estructurados

- Matrices: otras posibilidades de almacenamiento
  - ejemplo almacenamiento con punteros a filas

```
int fila0 = {1, 2, 3, 4, 5};
int fila1 = {2, 3, 4, 5, 6};
int fila2 = {3, 4, 5, 6, 7};
int *matriz[3] = {fila0, fila1, fila2};
// fila0 puede usarse como puntero a fila0[0]
```

  - Acceso al elemento univ[index][dig]
    - Mem [Mem [univ+4*index]+4*dig]
    - Se requieren dos accesos a memoria

Tipos de datos estructurados

- Matrices: ejemplos

```
int M[50][75];      // i => ecx
…                   // j => edx
M[i][j]=3;          // @ inicio M => ebx
```

- Escribir el trozo de código para realizar M[i][j]=3
- Se puede hacer con 3 instrucciones

Tipos de datos estructurados

- Matrices: ejemplos

```
int M[50][75];
int X[50][75];          // i => ecx
for (i=0;i<50;i++)      // j => edx
  for (j=0;j<75;j++)    // @ inicio M => ebx
   M[i][j]=X[i][j];     // @ inicio X => ebx + 15000
```


Tipos de datos estructurados

```
#for i
      xorl %ecx, %ecx
fori: cmpl $50, %ecx
      jge endfori
      cuerpo-FORi
      incl %ecx
      jmp fori
endfori:
```

```
#for j
      xorl %edx, %edx
forj: cmpl $75, %edx
      jge endforj
      cuerpo-FORj
      incl %edx
      jmp forj
endforj:
```


Tipos de datos estructurados

```
      xorl %ecx, %ecx
fori: cmpl $50, %ecx
      jge endfori
            xorl %edx, %edx
      forj: cmpl $75, %edx
            jge endforj
                  imull $75, %ecx, %eax
                  addl %edx, %eax
                  movl 15000(%ebx, %eax, 4), %esi
                  movl %esi, (%ebx, %eax, 4)
            incl %edx
            jmp forj
   endforj:
      incl %ecx
      jmp fori
endfori:
```

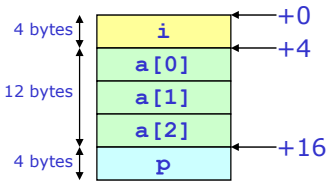

Tipos de datos estructurados

- Optimizar el código del programa anterior para reducir el número de instrucciones
- Se puede ver la matriz como un vector de 3750 posiciones

Tipos de datos estructurados

- Estructuras (struct)
  - conjunto heterogéneo de datos
    - almacenados de forma contigua en memoria
    - referenciados por su nombre

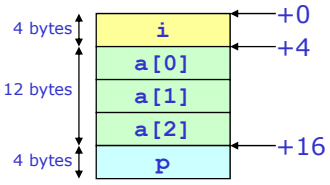
```
struct rec {
  int i;
  int a[3];
  int *p;
}S;

S.i = 1;
S.a[2] = 0;
S.p = &S.a[0];
```



Tipos de datos estructurados

- Estructuras (struct)
  - Ejemplo

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




```
S.i = 1;
S.a[2] = 0;
S.p = &S.a[0];
```

```
#%ebx <- @inicio S
movl $1, (%ebx)
movl $0, 12(%ebx)
leal 4(%ebx),%eax
movl %eax,16(%ebx)
```

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 IA32
    - Trato distinto en Windows y Linux
- Motivación para alinear datos
  - Accesos a memoria por longword o quadwords alineados
  - 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

- Tamaño de tipo de dato primitivo
  - 1 byte: (ej. char): No hay restricciones en la @
  - 2 bytes: (ej. Short): El bit más bajo de la @ debe ser 0
  - 4 bytes: (ej. int): Los 2 bits más bajos de la @ deben ser 00
  - 8 bytes: (ej. double):
    - En MS-Windows y otros SO los 3 bits más bajos de la @ deben ser 000
    - En Linux los 2 bits más bajos de la @ deben ser 00
  - 12 bytes: (ej. long double):
    - En Linux los 2 bits más bajos de la @ deben ser 00

Alineamiento de datos

- 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 (el mayor de los alineamientos de cualquier elemento)
  - La @ inicial de la estructura debe ser múltiplo de k
- Ejemplo

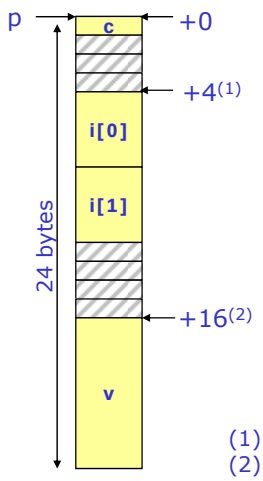
```
struct S1 {
    char c;
    int i[2];
    double v;
}*p;
```

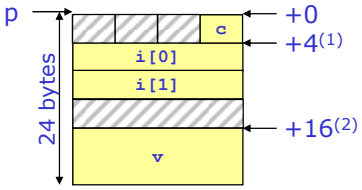
  - (Linux) k = 4
  - (Windows) k = 8 debido al elemento *double*

Alineamiento de datos

- Ejemplo (Windows) k = 8 debido al elemento *double*

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





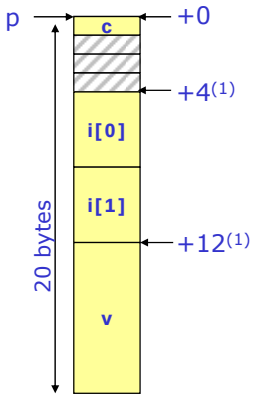
¡La dirección de inicio de la estructura ha de ser Múltiplo de 8!

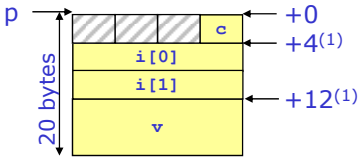
(1) Múltiplo de 4
(2) Múltiplo de 8

Alineamiento de datos

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

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





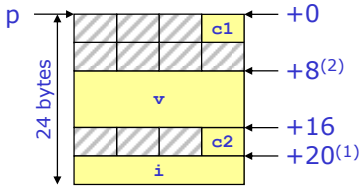
¡La dirección de inicio de la estructura ha de ser Múltiplo de 4!

(1) Múltiplo de 4

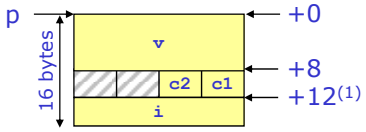
Alineamiento de datos

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

```
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 de la estructura ha 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;
```

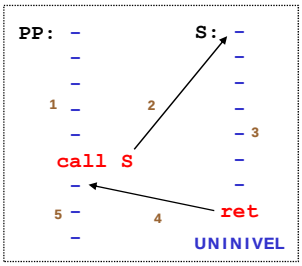
  - El programador de C puede reordenar los elementos de la estructura para minimizar el espacio ocupado.
  - Sin embargo, el programador de ensamblador NO puede realizar esta optimización.

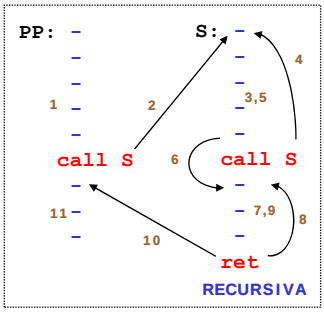
Gestión de subrutinas

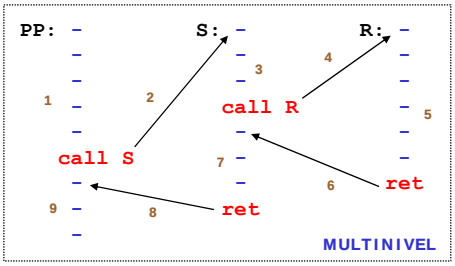
- Subrutina: Conjunto de instrucciones de LM que realiza una tarea específica y que puede ser activada (llamada) desde cualquier punto de un programa o desde la propia subrutina
- Activación interna: la llamada se hace desde la propia subrutina
- Activación externa: la llamada se hace desde el programa 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

Gestión de subrutinas

- Clasificación de las subrutinas







Gestión de 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

Gestión de subrutinas

- Terminología
  - Parámetros
    - Valor
    - Referencia
  - Variables locales
  - Invocación
  - Retorno resultado

```
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);
}
```


Gestión de subrutinas

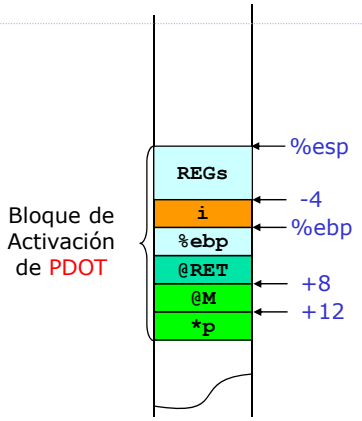
- Convenciones en C-Linux
  - Los parámetros se pasan por la pila de derecha a izquierda
  - Los vectores y matrices siempre se pasan por referencia
  - Los structs siempre se pasan por valor, no importa el tamaño
  - Los parámetros de tipo caracter (1 byte) ocupan 4 bytes
  - Los parámetros de tipo short (2 bytes) ocupan 4 bytes
  - Las variables locales están alineadas en la pila con la misma convención que dentro de un struc
    - Char en cualquier dirección
    - Short en direcciones múltiplos de 2
    - Integer en direcciones múltiplos de 4
  - Los registros %ebx, %esi, %edi se han de salvar si son modificados
  - Los registros %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

Gestión de Subrutinas

- Bloque 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
        -
        -
        -
```

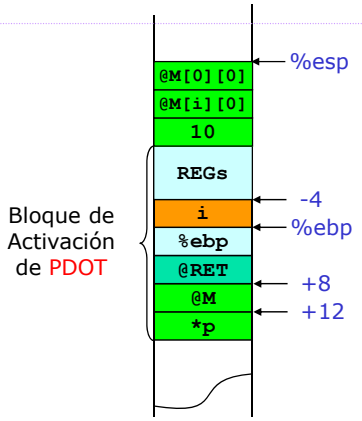


Gestión de Subrutinas

1. Paso de 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
```

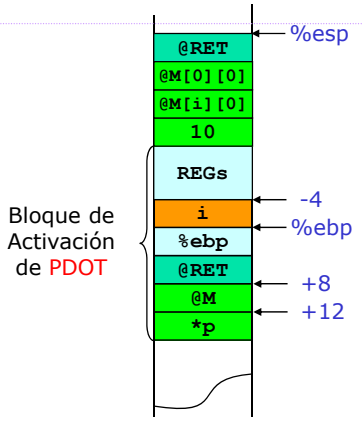


Gestión de Subrutinas

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);
}
```

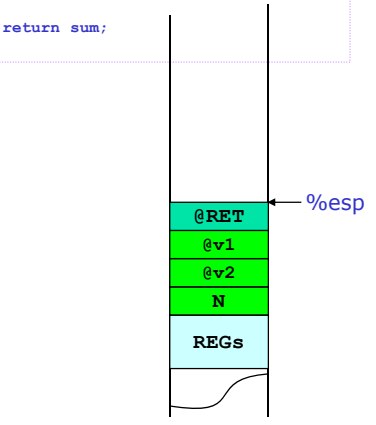


Gestión de Subrutinas

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;
}
```

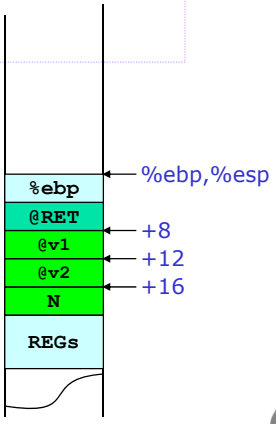


Gestión de Subrutinas

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;
}
```

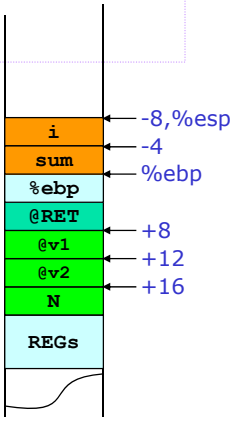


Gestión de Subrutinas

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;
}
```

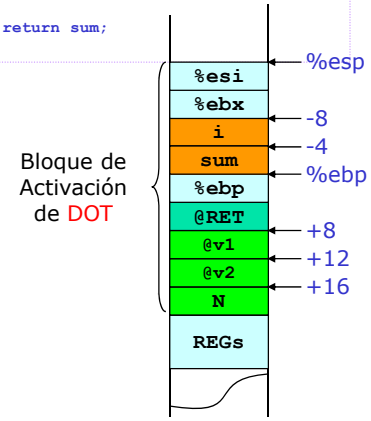


Gestión de Subrutinas

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;
}
```

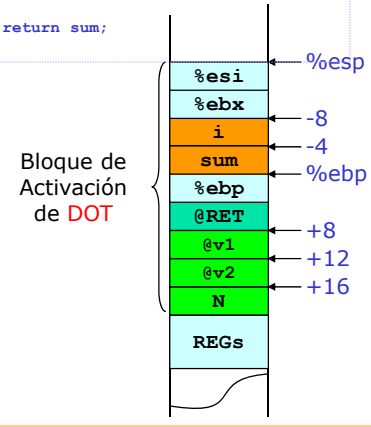


Gestión de Subrutinas

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;
}
```

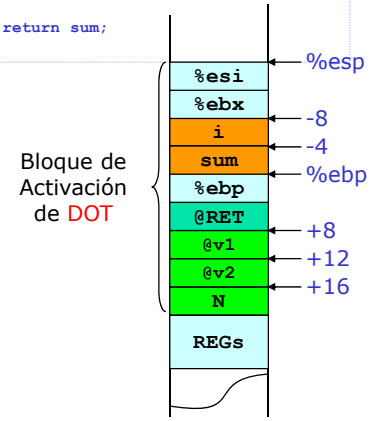


Gestión de Subrutinas

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;
}
```

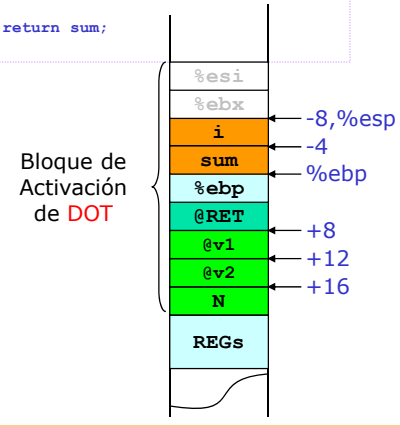


Gestión de Subrutinas

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;
}
```

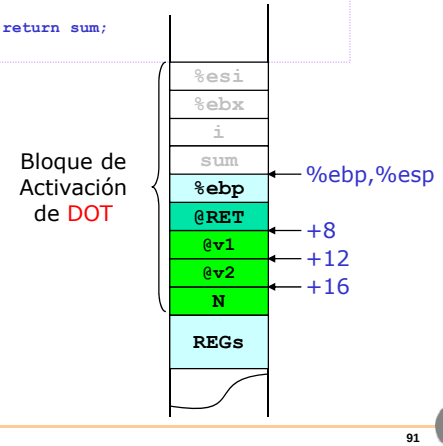


Gestión de Subrutinas

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;
}
```

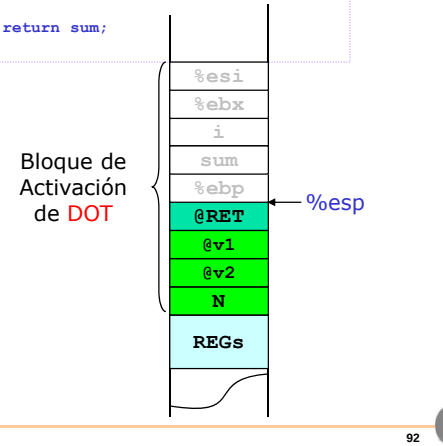


Gestión de Subrutinas

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;
}
```

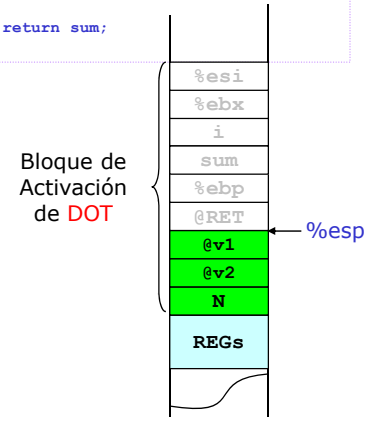


Gestión de Subrutinas

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;
}
```

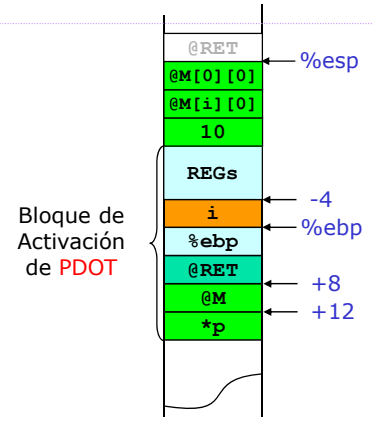


Gestión de Subrutinas

11. Volvemos 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);
}
```

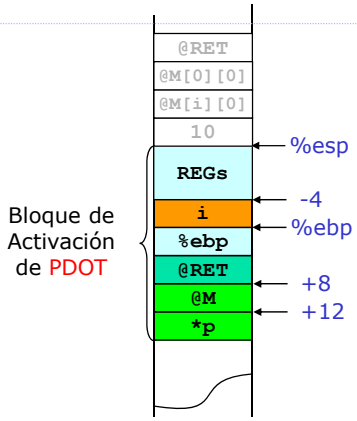


Gestión de Subrutinas

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
```

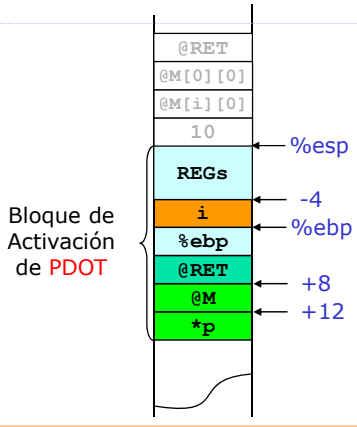


Gestión de Subrutinas

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)
```



Gestión de subrutinas

PDOT:

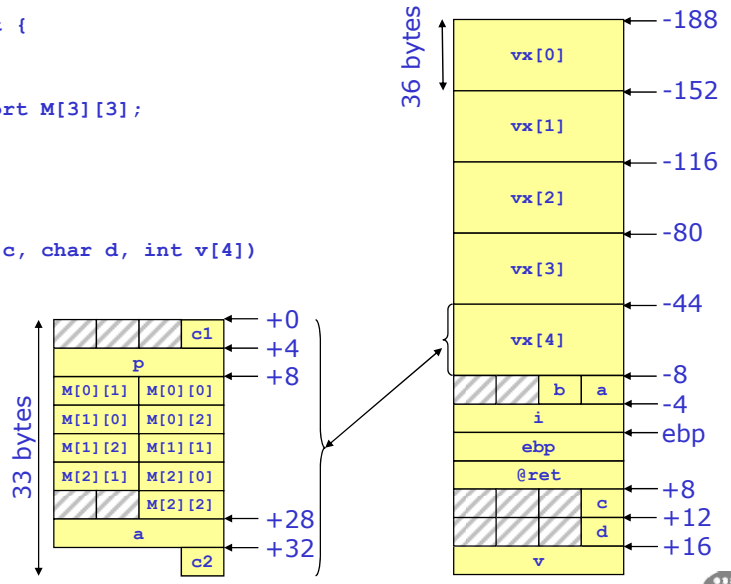
-
-
-
1 Paso de parámetros
2 llamada subrutina
12 elimina parámetros
13 Recoger/usar resultado
-
-

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 Subrutinas

```
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;
  ...
}
```



Gestión de Subrutinas

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

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