



## 1. Formato de las instrucciones

Las informaciones relativas a los cuatro factores anteriores se codifican en cada una de las instrucciones siguiendo un formato preestablecido. El formato determinará la longitud en bits de las instrucciones y los campos que codifican el valor de los factores citados. En general una instrucción se compone de los siguientes campos:

- Código de operación (CO)
- Operandos fuente (OP1, OP2,...)
- Operando destino o Resultado (OPd)
- Instrucción siguiente (IS)

CO	OP1	OP2	.....	OPd	IS
----	-----	-----	-------	-----	----

El CO determina la operación que se realiza sobre OP1,OP2,... El resultado se deja en OPd. Lo normal es que el número de operandos fuente de un repertorio no pase de 2. La dirección de la instrucción siguiente IS queda implícita en todas las instrucciones (se trata de la instrucción siguiente del programa) salvo en las instrucciones de ruptura condicional o incondicional de secuencia.

### 1.1 Diseño del repertorio de instrucciones.

Los repertorios de instrucciones podemos clasificarlos atendiendo a los siguientes criterios:

#### A) Número de operandos explícitos por instrucción

##### a) 3 operandos explícitos

CO	OP1 (fuente 1)	OP2 (fuente 2)	OP3 (destino)
----	----------------	----------------	---------------

ejemplo:      ADD B,C,A       $A \leftarrow B + C$

- Máxima flexibilidad
- Ocupa mucha memoria si los operandos no están en registros

##### b) 2 operandos explícitos

CO	OP1 (fuente 1)	OP2 (fuente)
----	----------------	--------------

ejemplo:      ADD B, C       $B \leftarrow B + C$

- Reduce el tamaño de la instrucción
- Se pierde uno de los operandos

##### c) 1 operando explícito

CO	OP1 (fuente)
----	--------------

ejemplo:      ADD B      Acumulador  $\leftarrow$  <Acumulador> + B

- Supone que fuente 1 y destino es un registro predeterminado (acumulador)
- Se pierde un operando fuente

**d) 0 operandos explícitos**

CO

ejemplo:           ADD                                   cima de pila ← <cima de pila> + <cima de pila - 1>

- Se trata de computadores que trabajan sobre una pila

Ejemplo:

E = (A - B)*(C + D)			
3 operandos	2 operandos	1 operando	0 operandos
ADD C, D, C SUB A, B, A MUL A, C, E	ADD C, D SUB A, B MUL A, C MOV C, E	LOAD A SUB B STORE A LOAD C ADD D MUL A STORE E	(PUSH) LOAD A (PUSH) LOAD B SUB (PUSH) LOAD D (PUSH) LOAD C ADD MUL (PULL) STORE E

**B) Forma de almacenar operandos en la CPU**

- Arquitectura de pila (HP 3000/70)
- Arquitectura de acumulador (Motorola 6809)
- Arquitectura de registros de propósito general (IBM 360)

Ejemplo: código máquina en cada una de las tres alternativas correspondiente a la sentencia de asignación C := A + B

C = A + B		
Pila	Acumulador	Conjunto de registros
PUSH A	LOAD A	LOAD R1, A
PUSH B	ADD B	ADD R1, B
ADD	STORE C	STORE C, R1
POP C		

**Las arquitecturas de registros de propósito general** se clasifican a su vez atendiendo al número máximo de operandos (2 ó 3) que pueden tener las instrucciones de la ALU y cuantos de ellos se pueden ubicar en memoria:

(operandos - en memoria)

- ( 3        -        0 ) Arquitectura registro-registro (también llamada de carga-almacenamiento). Utilizan tres operandos totales y cero en memoria. Formato de longitud fija y codificación simple de las instrucciones que pueden ejecutarse en un número similar de ciclos. Facilitan un modelo simple de generación de código para el compilador. SPARC, MIPS, PowerPC
- ( 2        -        1 ) Arquitectura registro-memoria. Utilizan dos operandos totales con uno ubicado en la memoria. Intel 80X86, Motorola 68000

( 3 - 3 ) Arquitectura memoria-memoria. Utilizan tres operandos totales con la posibilidad de ser ubicados los tres en memoria. VAX

### Códigos de operación de longitud fija y variable

Una máquina con un formato de instrucción que dedica  $n$  bits al CO permitirá  $2^n$  instrucciones diferentes, cada una de las cuales puede tener diferente número de operandos (0, 1, 2, 3, etc.). Los bits del campo OPERANDOS se pueden utilizar para extender el CO de aquellas instrucciones con menor número de operandos.

**Ejemplo:** Partimos de una máquina con instrucciones de longitud fija de 24 bits y consideraremos los siguientes supuestos:

1) La máquina dispone de 16 registros generales

	4	4	16
<b>CO</b>	<b>R</b>	<b>OP</b>	

⇒ En este caso se pueden codificar 16 instrucciones de 2 operandos: uno en registro y el otro en memoria

2) Si queremos extender el CO se puede utilizar una de las 16 combinaciones del CO (quedarían 15 con 2 operandos), por ejemplo CO = 1111, dando la posibilidad de codificar 16 instrucciones de 1 operando en memoria. Si queremos seguir extendiendo el CO podemos utilizar CO = 1111 1111 (quedarían 15 con 1 operando) para definir instrucciones sin operandos ( $2^{16} = 65.536$ )

En la siguiente tabla se resume el proceso descrito.

0 0 0 0	R	OP	15 instrucciones de 2 operandos (CO de 4 bits)
0 0 0 1	R	OP	
.	.	.	
.	.	.	
1 1 1 0	R	.	
1 1 1 1	0 0 0 0	.	15 instrucciones de 1 operando (CO de 8 bits)
1 1 1 1	0 0 0 1	.	
.	.	.	
.	.	.	
1 1 1 1	1 1 1 0	OP	
1 1 1 1	1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	$2^{16} = 65.536$ instrucciones de 0 operandos (CO de 24 bits)
1 1 1 1	1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	
.	.	.	
.	.	.	
1 1 1 1	1 1 1 1	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	

**Otra alternativa:** dedicar 2 bits para indicar si la instrucción tiene 0, 1 o 2 operandos:

	2	2	4	16
<b>L</b>	<b>CO</b>	<b>R</b>	<b>OP</b>	

En este caso podemos codificar los siguientes grupos de instrucciones:

- L = 00 → CO de 2 bits → 4 instrucciones de 2 operandos
- L = 01 → CO de 6 bits → 64 instrucciones de 1 operando
- L = 10 → CO de 22 bits → 4.194.304 instrucciones de 0 operandos

**Optimización del CO variable en función de la frecuencia de las instrucciones.**

Una posibilidad a la hora de codificar las operaciones de un repertorio de instrucciones es utilizar algún criterio de óptimo. En este sentido tenemos dos alternativas:

- a) Frecuencia de aparición en el programa → optimización de memoria
- b) Frecuencia de ejecución en el programa → optimización del tráfico CPU-Memoria

La alternativa b) es la más interesante en la actualidad, pues prima la velocidad de ejecución sobre la memoria necesaria para almacenar el programa.

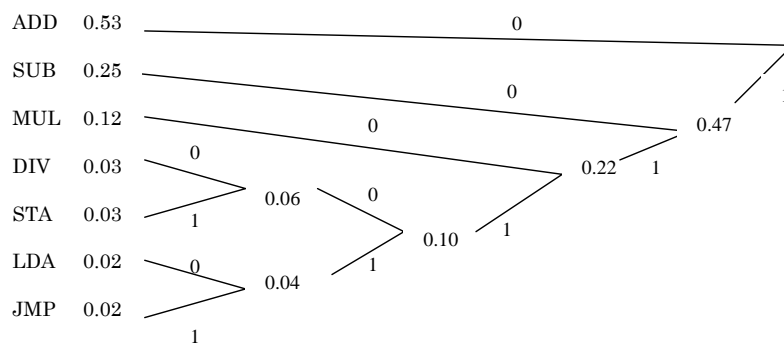
Para optimizar el CO se puede utilizar la codificación de Huffman que veremos con el siguiente Ejemplo: Supongamos las siguientes frecuencias de ejecución de 7 tipos diferentes de instrucciones:

Tipo de instrucciones	Frecuencia de ejecución
ADD	0.53
SUB	0.25
MUL	0.12
DIV	0.03
STA	0.03
LDA	0.02
JMP	0.02

Con CO de longitud fija se necesitarían 3 bits. Para obtener el código de Huffman procedemos de la siguiente manera:

- 1) Se escriben en una columna las instrucciones y a su derecha su frecuencia de ejecución. Cada elemento de la columna será un nodos terminal del árbol de decodificación.
- 2) Se unen las dos frecuencias menores de la columna anterior con sendos arcos, obteniéndose un nuevo nodo cuyo valor será la suma de los nodos de procedencia.
- 3) Se repite el paso 2) hasta llegar a la raíz del árbol que tendrá valor 1
- 4) Comenzando en la raíz, asignamos 0 (1) al arco superior y 1 (0) al inferior hasta llegar a los nodos terminales
- 5) Se obtiene el código de cada instrucción recorriendo el árbol de la raíz a la instrucción y concatenando los valores de los arcos del camino

Para nuestro ejemplo tendremos lo siguiente:



Tipo de instrucciones	Frecuencia de ejecución	Código de Huffman
ADD	0.53	0
SUB	0.25	10
MUL	0.12	110
DIV	0.03	11100
STA	0.03	11101
LDA	0.02	11110
JMP	0.02	11111

Longitud media del código resultante:

$$l_m = \sum_i f_i \times l_i = 0.53 \times 1 + 0.25 \times 2 + 0.12 \times 3 + 0.003 \times 5 + 0.003 \times 5 + 0.02 \times 5 + 0.02 \times 5 =$$

$$1.89\_bits < 3\_bits$$

## 2. Propiedades generales del direccionamiento.

### Resolución

Es la menor cantidad de información direccionada por la arquitectura. El mínimo absoluto es un bit, aunque esta alternativa la utilizan pocos procesadores, por ejemplo, el iAPX432 de Intel (1981) Requiere un gran número de bits para expresar las direcciones de una cierta cantidad de información y mucho tiempo para alinearlos correctamente. Lo más frecuente en los procesadores actuales es utilizar resoluciones de 1 o 2 bytes. La resolución puede ser diferente para instrucciones y datos aunque lo normal es que coincida.

Resolución	MC68020	VAX-11	IBM/370	B1700	B6700	iAPX432
Instrucciones	16	8	16	1	48	1
Datos	8	8	8	1	48	8

### Orden de los bytes en memoria

El concepto de *endian* lo introdujo Cohen para expresar la forma como se ordenan los bytes de un escalar de varios bytes.

- **Modo *big-endian***: almacena el byte más significativo del escalar en la dirección más baja de memoria
- **Modo *little-endian***: almacena el byte más significativo del escalar en la dirección más alta de memoria.

Ejemplo: el hexadecimal 12 34 56 78 almacenado en la dirección de memoria 184 tendrá la siguiente organización en cada uno de los modos:

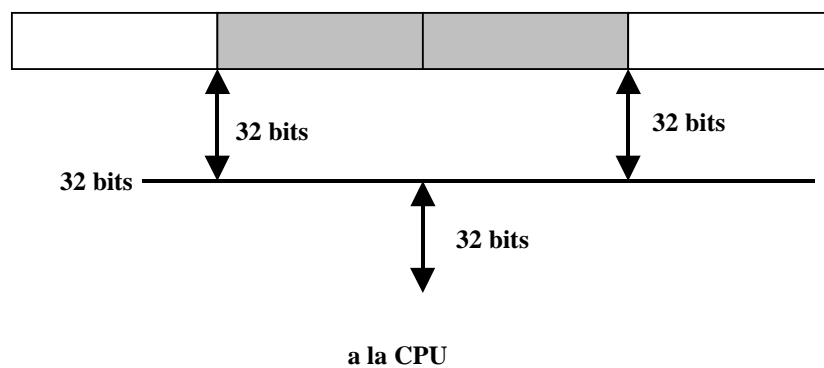
<b>big-endian</b>		<b>little-endian</b>	
184	12	184	78
185	34	185	56
186	56	186	34
187	78	187	12

### Alineación

Un objeto de datos de  $n$  bytes ubicado en la dirección de memoria  $D$  se dice que está alineado si  $D \bmod n = 0$

Objeto de datos direccionado (tamaño)	Alineaciones correctas
byte	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
media palabra (2 bytes)	0, 2, 4, 6, 8, 10
palabra (4 bytes)	0, 4, 8, 12
doble palabra (8 bytes)	0, 8, 16

- Determinadas máquinas sólo permiten accesos alineados
- La falta de alineación implica complicaciones hardware
- Los programas con accesos alineados se ejecutan más rápidamente
- Para alinear datos se utiliza una red de alineación. En el caso de la figura para acceder a una palabra no alineada serán necesarios 2 accesos para obtener la parte alta y baja



### Espacios de direcciones

En un mismo procesador pueden diferenciarse hasta 3 espacios de direcciones diferentes:

- Espacio de direcciones de registros
- Espacio de direcciones de memoria
- Espacio de direcciones de entrada/salida

Los espacios de direcciones de memoria y entrada/salida de algunos procesadores están unificados (un solo espacio), ocupando los puertos de E/S direcciones de ese espacio único. En estos procesadores (ejemplo, 68000) no existen instrucciones específicas de E/S, para esta función se utilizan las de referencia a memoria (carga y almacenamiento) con las direcciones asignadas a los puertos.

### 3. Modos de direccionamiento.

Los modos de direccionamiento determinan la forma como el operando (OPER) presente en las instrucciones especifican la dirección efectiva (DE) del dato operando (DO) sobre el que se realiza la operación indicada por CO.

#### Inmediato.

CO	OPER
----	------

DO = OPER

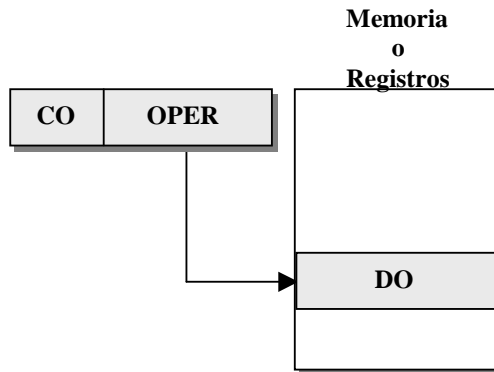
- El dato operando se ubica en la propia instrucción ==> no requiere accesos a memoria.
- Se suele utilizar para datos constantes del programa
- El tamaño está limitado por el número de bits de OPER

#### Implicito

CO
----

- El dato operando se supone ubicado en algún lugar específico de la máquina, por ejemplo, una pila **Directo (memoria o registros)**

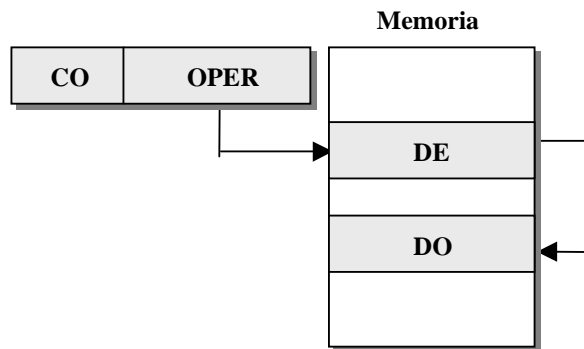
OPER = Dirección de memoria o de un registro  
 DE = OPER  
 DO = <OPER>



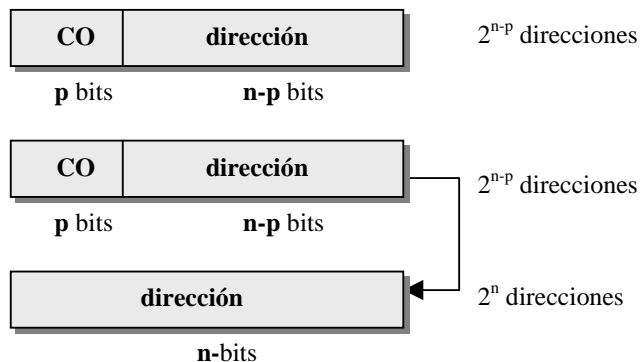
- La especificación de un registro requiere menor número de bits que la de una posición de memoria
- El acceso a los registros es más rápido que a Memoria
- El direccionamiento directo a memoria se conoce como absoluto
- A veces se limita el número de bits de OPER limitando el acceso a sólo una parte de la memoria que suele ser la correspondiente a las direcciones más bajas (página cero)

**Indirecto (memoria)**

OPER = Dirección de memoria  
 DE = <OPER>  
 DO = <<OPER>>



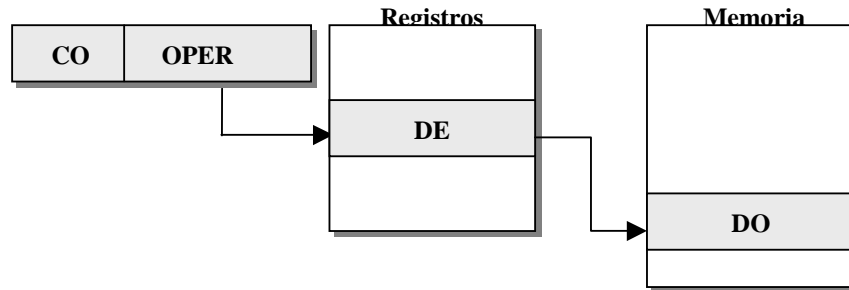
- Permite el tratamiento de una dirección de memoria como un dato
- Permite el paso por referencia de parámetros a subrutinas
- Permite referenciar un espacio mayor de direcciones



**Indirecto (registro)**



OPER = Dirección de un registro  
 DE = <OPER>  
 DO = <<OPER>>



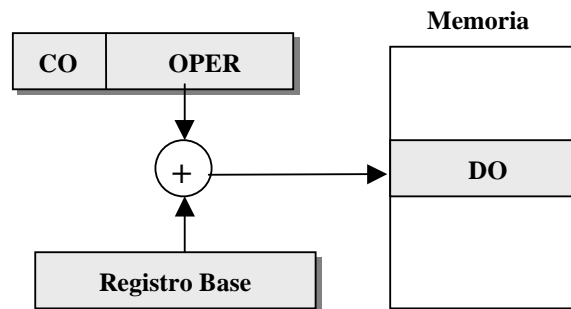
**Modos con desplazamiento.**

Calculan la dirección efectiva (DE) sumando al contenido de un registro el operando declarado en la instrucción (OPER) que se interpreta como un desplazamiento respecto al contenido del registro. La distinta naturaleza del registro hace que se diferencien tres modos con desplazamiento. Todos explotan la proximidad de los datos o instrucciones referenciadas para utilizar menor número de bits en el campo OPER.

**Direccionamiento base más desplazamiento**

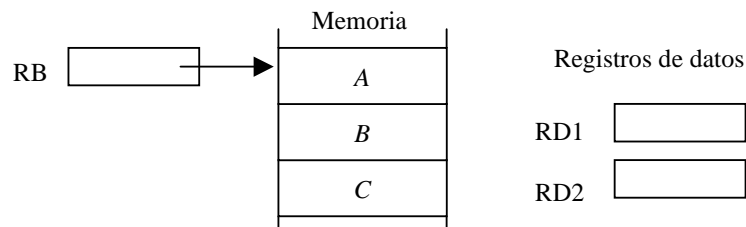
DE = <Registro base> + OPER; OPER = desplazamiento

- Se utiliza para la reubicación de datos y programas en memoria



**Ejemplo: A = B + C**  
 Programa

LOAD RB,1; RD1	(RD1 <-- <RB>+1)	<RB> + 1 = B
LOAD RB, 2; RD2	(RD2 <-- <RB>+2)	<RB> + 2 = C
ADD RD1; RD2	(RD1 <-- <RD1> + <RD2>)	
STORE RD1; RB,0	(A<RB>+0 <-- <RD1>)	<RB> + 0 = A

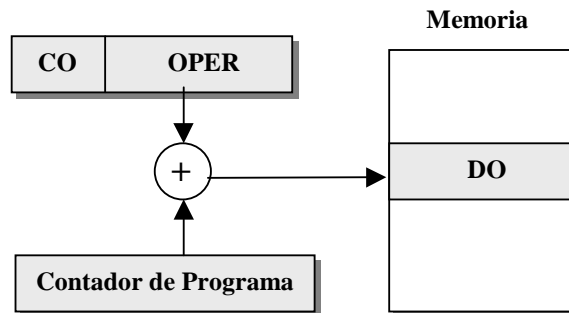


Este programa podremos reubicarlo en memoria y siempre llevará a la dirección apuntada por el registro base RB la suma de los contenidos de las dos direcciones siguientes.

### Direccionamiento relativo

$DE = \langle \text{Contador de programa} \rangle + \text{OPER}$ ; OPER = desplazamiento

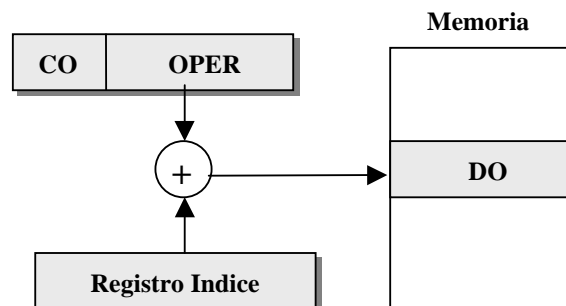
- Se utiliza en las instrucciones de salto para conseguir la reubicación de estas instrucciones
- El desplazamiento en estas instrucciones tiene signo (c2) lo que significa que el salto relativo se puede dar hacia posiciones anteriores o siguientes a la ocupada por la instrucción.



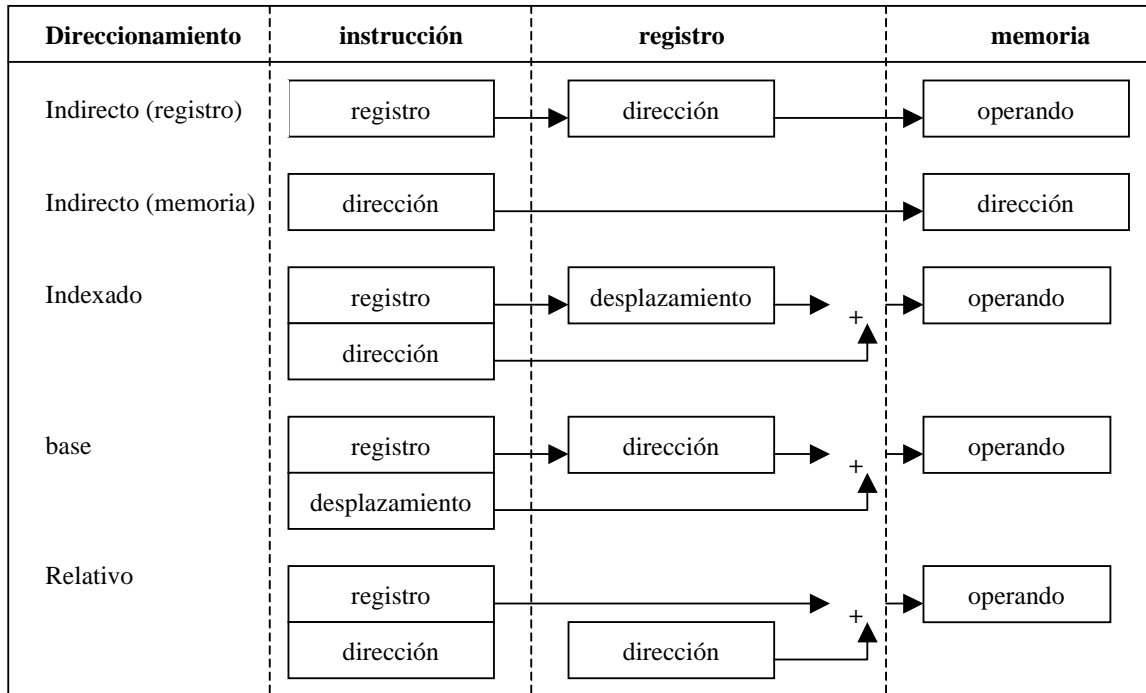
### Direccionamiento indexado

$DE = \langle \text{Registro índice} \rangle + \text{OPER}$ ; OPER = desplazamiento

- Se utiliza para recorrer estructuras lineales como los *arrays*
- Par facilitar su uso se suele complementar con el pre o post incremento o decremento del registro índice



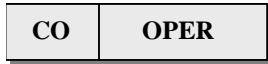
### Resumen de los tipos de direccionamiento



**Modos compuestos**

Veremos los modos compuestos analizando el repertorio de algunos procesadores.

**Modos de direccionamiento del MC 68.X**



• Inmediato

DO = OPER

- enteros: 8, 16 y 32 bits
- reales: 32 bits (simple precisión), 64 bits (doble precisión) y 96 bits (precisión extendida)

• Directo

- Memoria o absoluto

DE = OPER, DO = <DE>, con OPER de 16 y 32 bits

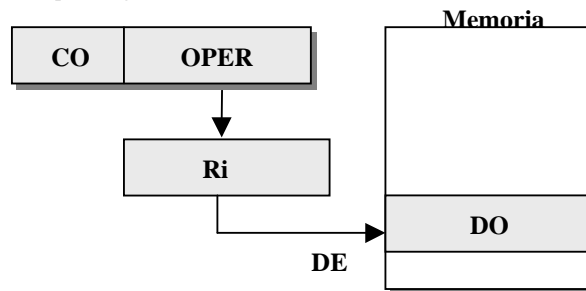
- Registro

Ri = OPER, DO = <Ri>, con Ri cualquier registro

• Indirecto registro

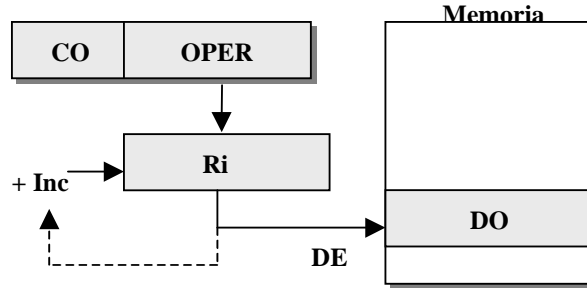
- puro

DE = <Ri>, con Ri cualquier registro



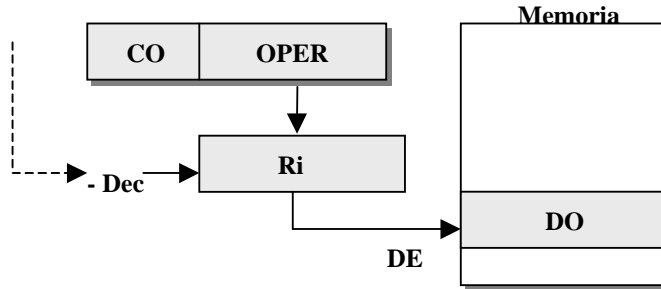
- Indirecto registro con postincremento

$DE = \langle Ri \rangle;$   $Ri \leftarrow \langle Ri \rangle + Inc,$  con  $Inc = 1, 2 \text{ ó } 4 \text{ bytes}$



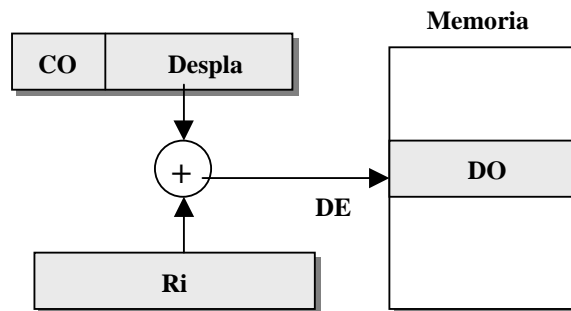
- Indirecto registro con predecremento

$Ri \leftarrow \langle Ri \rangle - Dec,$   $DE = \langle Ri \rangle,$  con  $Dec = 1, 2 \text{ ó } 4 \text{ bytes}$



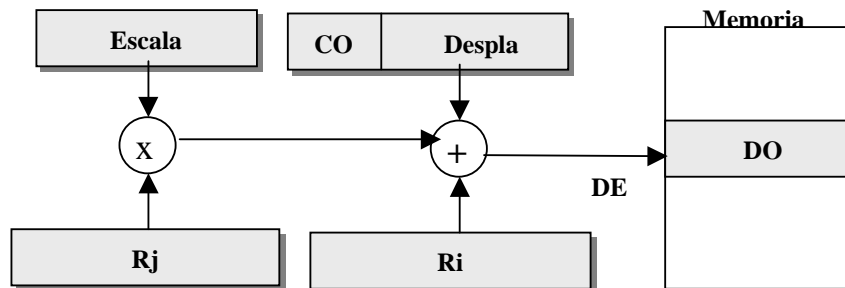
- Indirecto con desplazamiento (= base + desplazamiento)

$DE = \langle Ri \rangle + Despla$



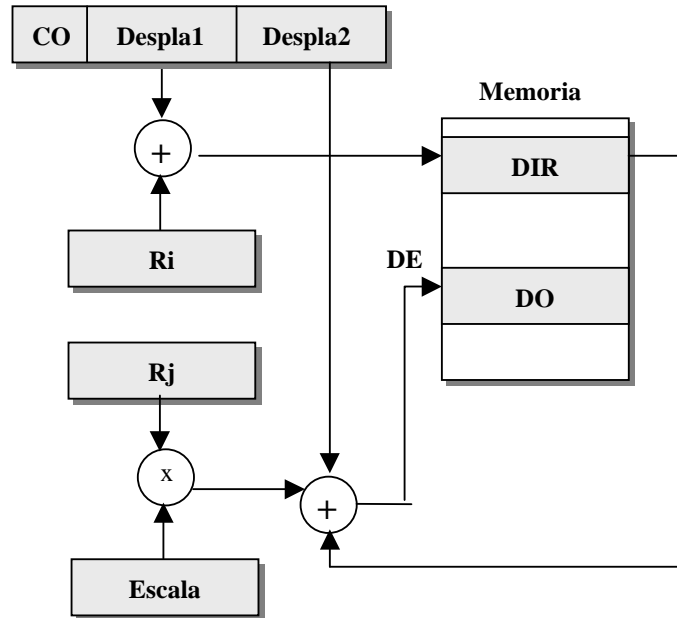
- Indirecto registro indexado (= base + desplazamiento indexado)

$DE = \langle Ri \rangle + \langle Rj \rangle \times Escala + Despla,$  con  $Escala = 1, 2, 4, 8 \text{ bytes}$

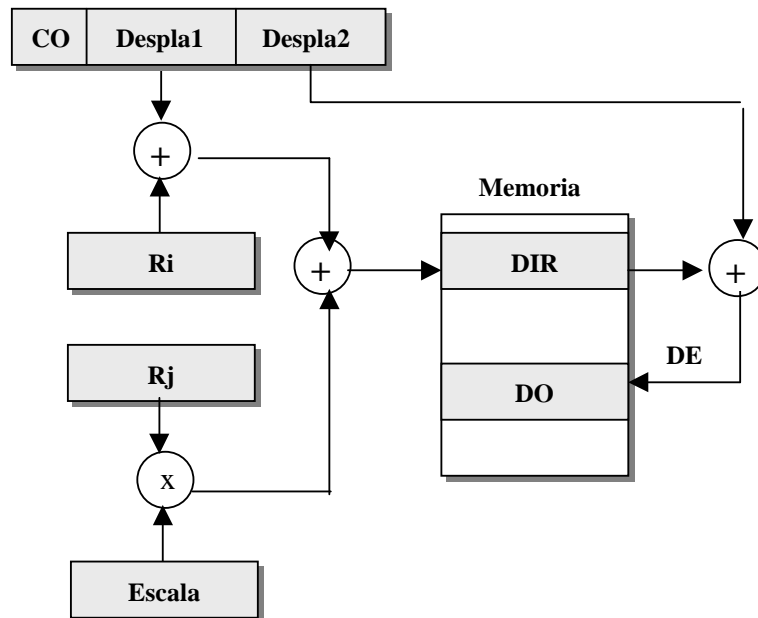


• Indirecto memoria

- Postindexado (= base + desplazamiento indirecto indexado + desplazamiento)  
 $DE = \langle\langle Ri \rangle + despla1 \rangle + \langle Rj \rangle \times Escala + Despla2$



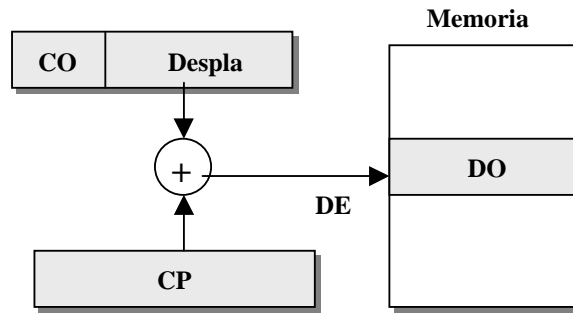
- Preindexado (base + desplazamiento indexado indirecto + desplazamiento)  
 $DE = \langle\langle Ri \rangle + Despla1 + \langle Rj \rangle \times Escala \rangle + Despla2$



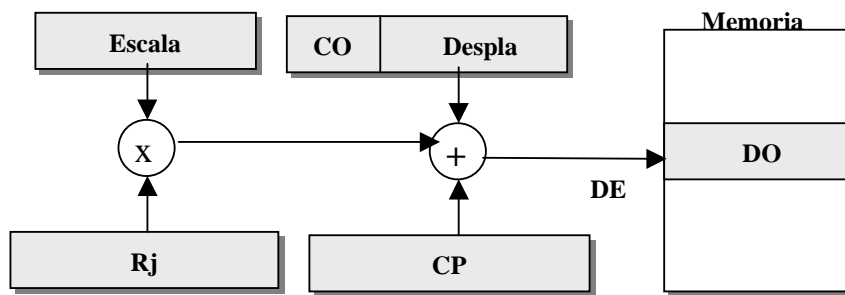
- Relativo

- Básico

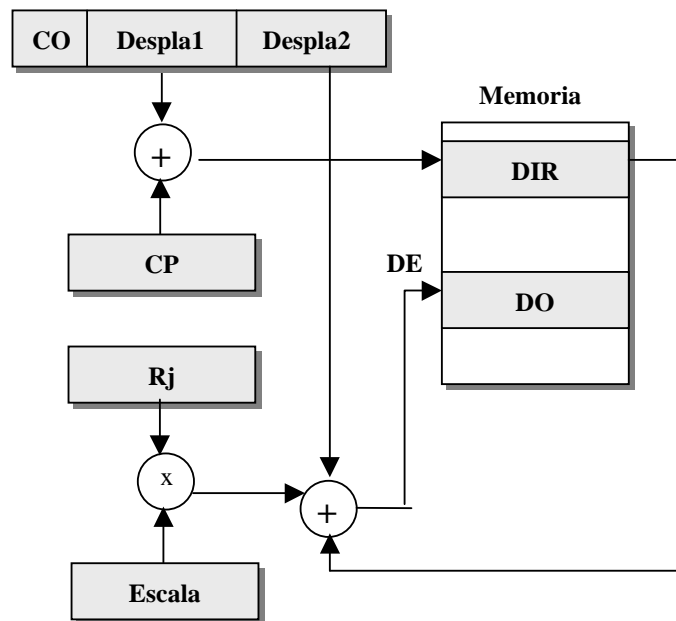
$$DE = \langle CP \rangle + Despla$$



- Indexado con desplazamiento  
 $DE = \langle CP \rangle + \langle R_j \rangle \times Escala + Despla$

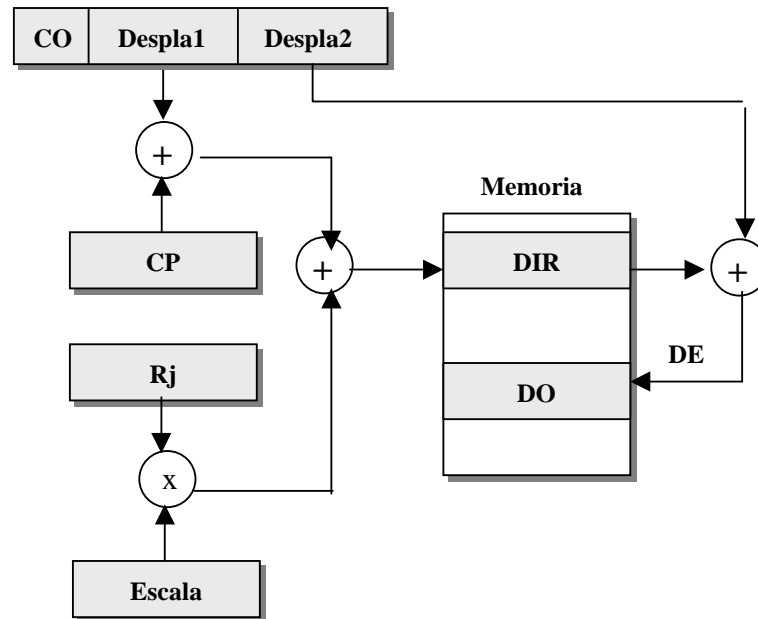


- Indirecto memoria postindexado  
 $DE = \langle \langle CP \rangle + Despla1 \rangle + \langle R_j \rangle \times Escala + Despla2$



- Indirecto memoria preindexado

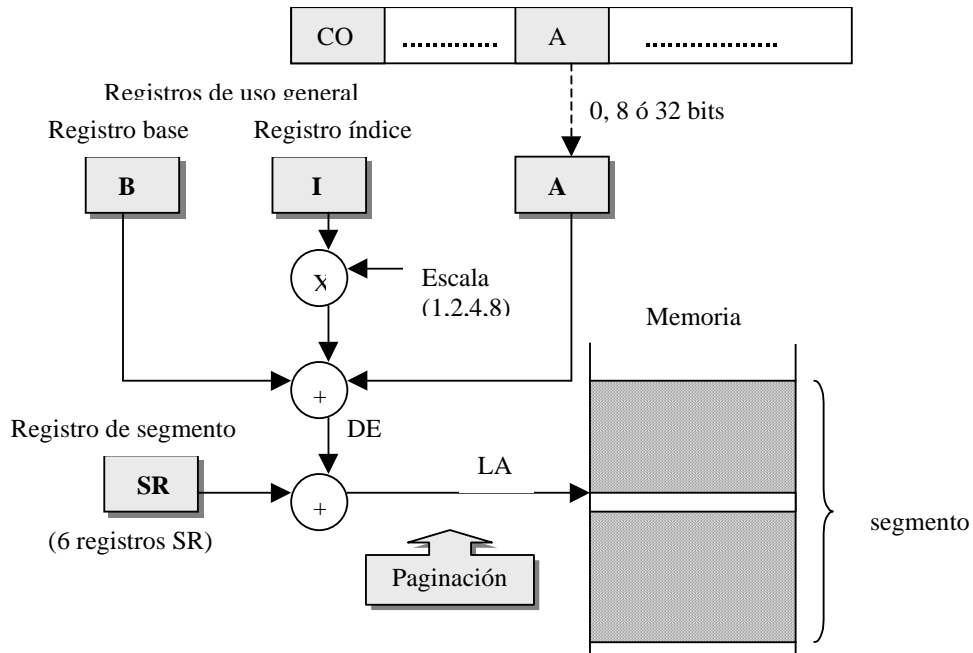
$$DE = \langle\langle CP \rangle\rangle + Despla1 + \langle Rj \rangle \times Escala + despla2$$



### Modos de direccionamiento del MIPS R-2000

- Inmediato
- Registro  
LA = R
- Relativo  
DE =  $\langle PC \rangle + Despla$
- Indirecto registro con desplazamiento (= base + desplazamiento)  
DE =  $\langle Ri \rangle + Despla$

### Modos de direccionamiento del Pentium II



- Inmediato  
 $DO = A$  (1,2,4 bytes)
- Registro  
 $LA = R,$        $DO = \langle R \rangle$  (LA = dirección lineal)
- Desplazamiento  
 $LA = \langle SR \rangle + A$
- Base  
 $LA = \langle SR \rangle + \langle B \rangle$
- Base + desplazamiento  
 $LA = \langle SR \rangle + \langle B \rangle + A$
- Indexado  
 $LA = \langle SR \rangle + \langle I \rangle \times Escala + A$
- Base + desplazamiento indexado  
 $LA = \langle SR \rangle + \langle B \rangle + \langle I \rangle + A$
- Base + desplazamiento indexado escalado  
 $LA = \langle SR \rangle + \langle B \rangle + \langle I \rangle \times Escala + A$
- Relativo  
 $LA = \langle PC \rangle + A$

#### 4. Soporte de los modos de direccionamiento a los lenguajes de alto nivel

##### Visibilidad en C

Un programa escrito en C no es más que una colección de subprogramas (funciones) en idéntico nivel, dentro del programa principal (Main). Estas funciones pueden llamarse entre sí, incluso de forma recursiva. Las variables *locales* definidas en una función son visibles sólo dentro de la función. En cambio las variables *globales* se definen fuera de la funciones (en la función Main) y pueden ser referenciadas desde cualquiera de ellas.

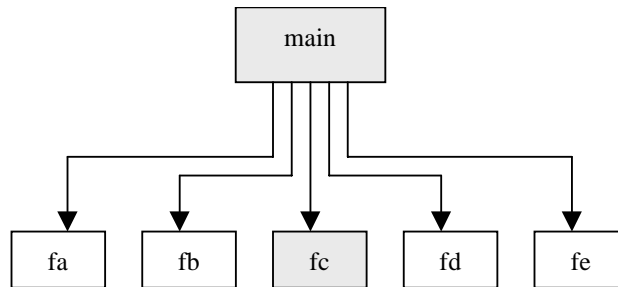


Las variables locales de una función se asignan dinámicamente. Estudiaremos ahora los modos de direccionamiento más adecuados para satisfacer los requerimientos de los lenguajes de alto nivel. Estos modos reducirán al mínimo el número de instrucciones requeridas para acceder a los elementos de las diferentes estructuras de datos (*array*, *record*, etc.) que soportan estos lenguajes, es decir, para calcular sus direcciones efectivas. Estos lenguajes presentan una estructura de bloques e incorporan el concepto de visibilidad de las variables del programa, es decir, las reglas de acceso a las variables de cada uno de los bloques.

, es decir, cada vez que se activa la función. La asignación tiene lugar dentro de un registro de activación (RA) que se ubica en la pila asociada al programa. Dada la naturaleza recursiva de las llamadas, pueden existir en la pila más de un RA para la misma función (tantos como llamadas). Supongamos el siguiente perfil de programa C:

Main()	fb (...)	fc (...)	fd (...)
{	{	{	{
.	.	.	.
.	.	.	.
... = ... fe(...) ...	... = ... fb(...) ...	... = ... fc(...) ...	}
.	.	.	fe (...)
.	.	.	{
.	.	.	.
}	... = ... fc(...) ...	... = ... fb(...) ..	.
fa (...)	.	.	.
{	.	.	... = ... fc(...) ...
.	}	}	.
.			.
.			.
}			}

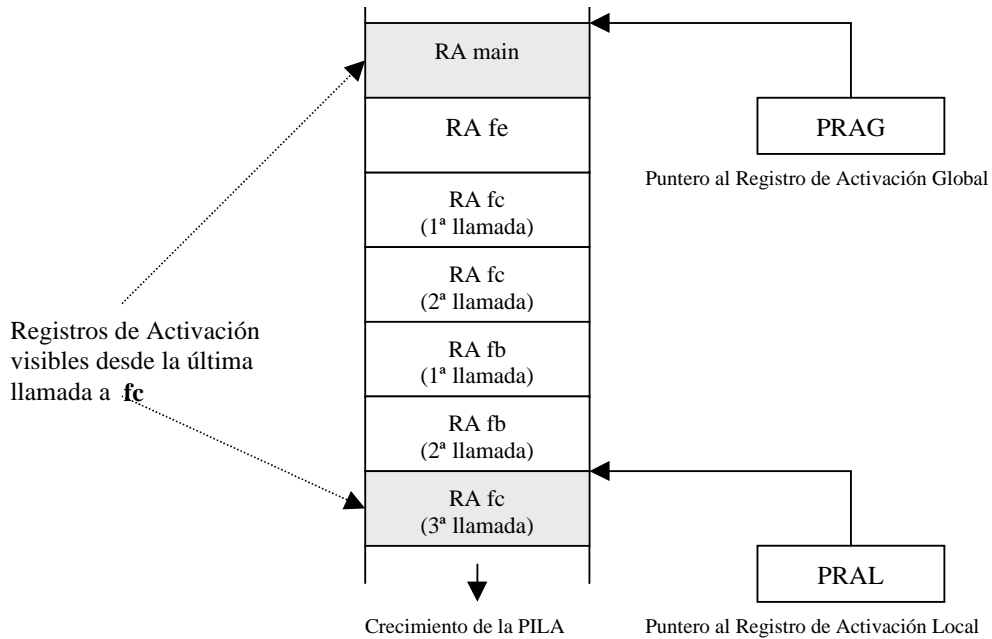
La estructura de este programa sería la siguiente:



Supongamos que tiene lugar la siguiente secuencia de llamadas:

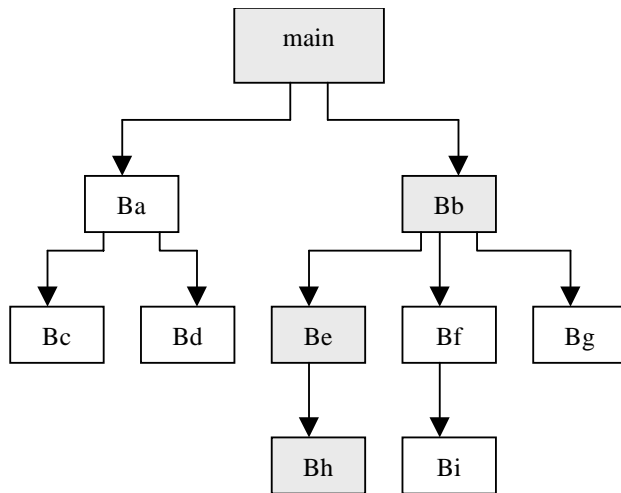
main → fe → fc → fc → fb → fb → fc

En la pila del programa se ubican los registros de activación correspondientes a cada una de estas llamadas, tal como se muestra en la siguiente figura, donde se ha sombreado las variables que pueden ser accedidas desde la tercera activación de la función fc: las globales y las locales a esta tercera llamada. Para realizar el acceso se dispone de sendos registros que apuntan a cada uno de estos RA, el Puntero al Registro de Activación Global (PRAG) y el Puntero al Registro de Activación Local (PRAL)



### Visibilidad en Pascal

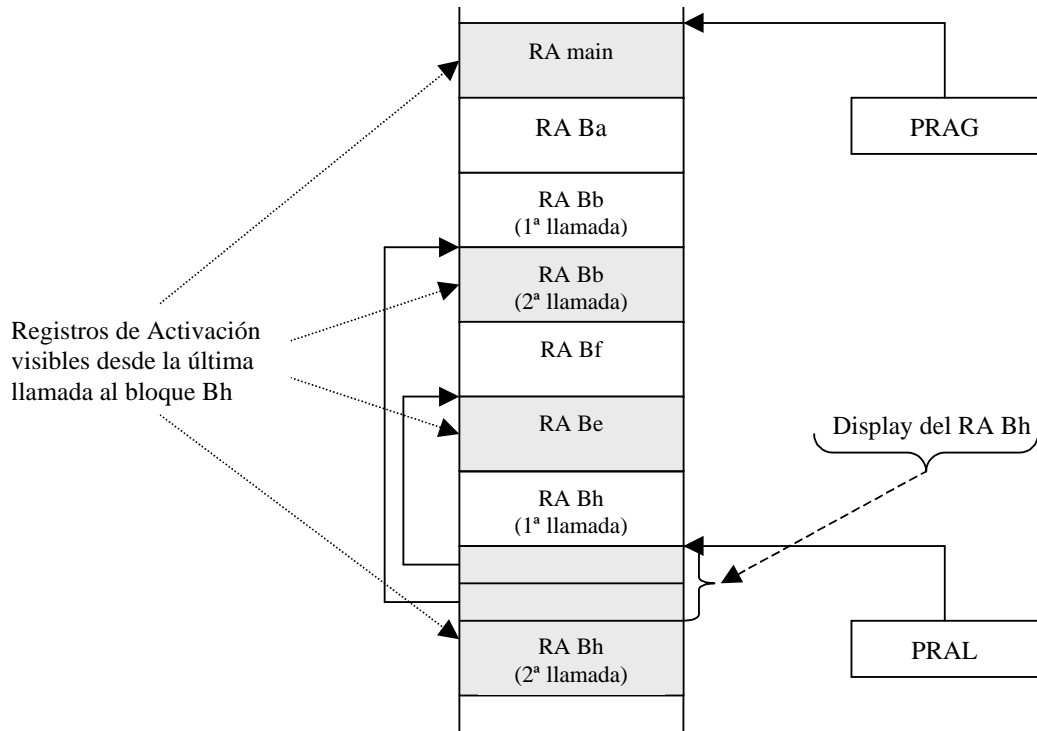
En los lenguajes de tipo Pascal un bloque puede ser un *procedure* o una *function*, y el concepto de visibilidad adquiere una estructura anidada. Los bloques de un programa de este tipo presentan una estructura jerárquica y la visibilidad de variables desde la llamada a un bloque se extiende a todos los bloques (últimas llamadas) en el camino hasta la raíz (*main*)



```

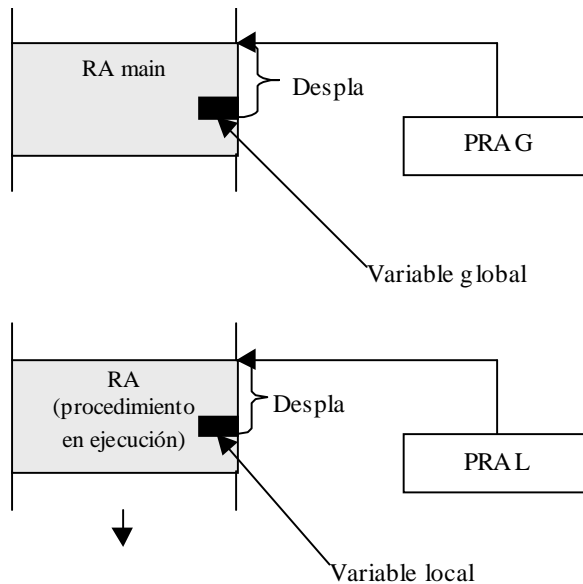
program main(..)
  procedure a
  begin
    procedure c
    begin
      ...
    end
    procedure d
    begin
      ...
    end
  end
end
procedure b
begin
  procedure e
  begin
    ... function h(..)
  end
  procedure f
  begin
    ... function i(..)
  end
  procedure g
  begin
    ...
  end
end
end
end
  
```

Secuencia de llamada: main → Ba → Bb → Bb → Bf → Be → Bh → **Bh**



El *display* de un RA contiene un puntero a cada RA<sub>i</sub> visible desde RA

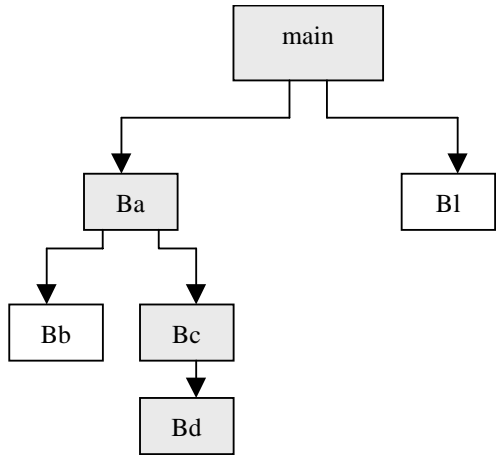
**Acceso a variables escalares locales o globales (contenido)**



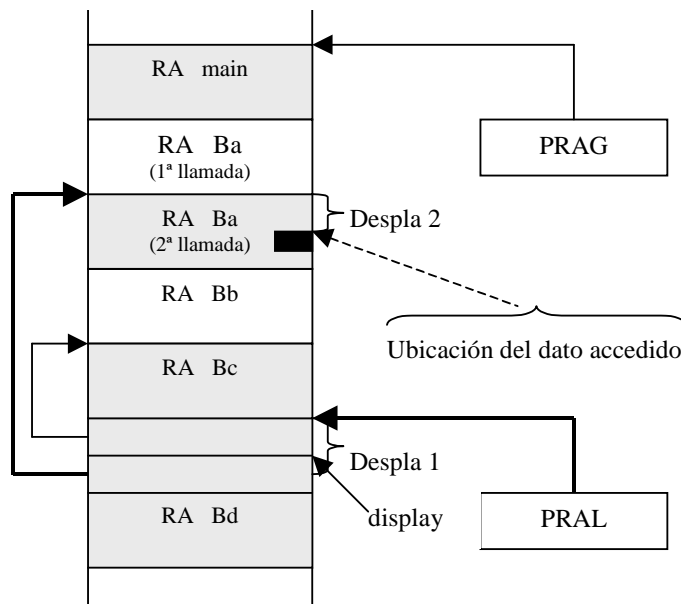
Variables locales:  $DE = \langle PRAL \rangle + despla \rightarrow$  direccionamiento base + desplazamiento

Variables globales:  $DE = \langle PRAG \rangle + despla \rightarrow$  direccionamiento base + desplazamiento

**Acceso a variables escalares no locales (contenido)**

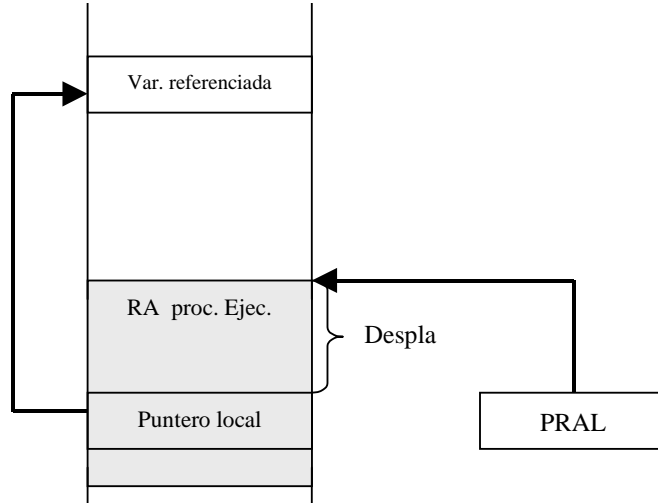


Secuencia de llamada: main → Ba → Ba → Bb → Bc → Bd



DE = <<PRAL> + Despla1> + Despla 2 →  
 direccionamiento base + desplazamiento indirecto + desplazamiento

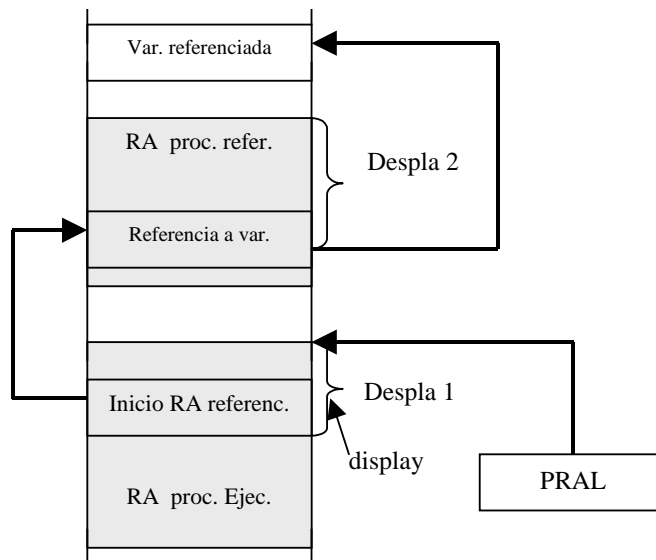
**Acceso a variables escalares locales (dirección)**



$$DE = \langle\langle PRAL \rangle + Despla \rangle \rightarrow$$

direccionamiento base + desplazamiento indirecto

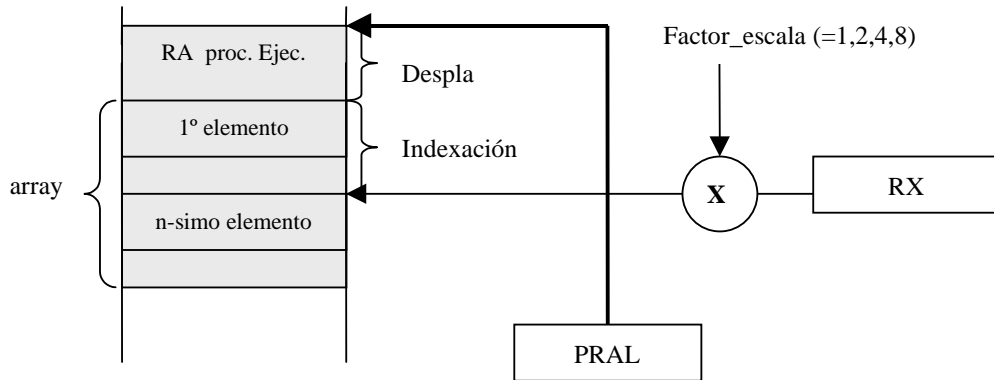
**Acceso a variables escalares no locales (dirección)**



$$DE = \langle\langle\langle PRAL \rangle + Despla1 \rangle + Despla 2 \rangle \rightarrow$$

direccionamiento base + desplazamiento indirecto + desplazamiento indirecto

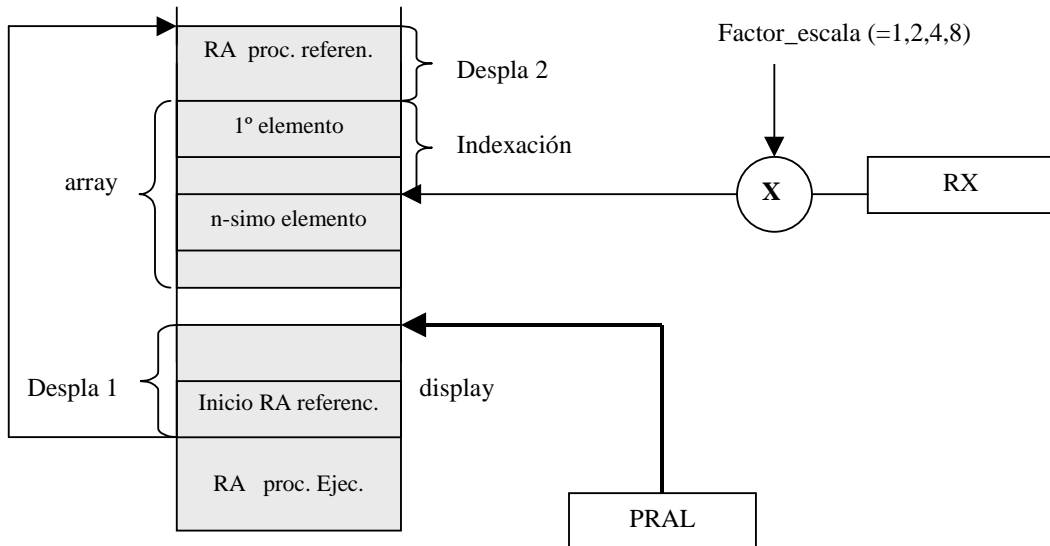
**Acceso a variables de tipo *array* locales (contenido)**



$$DE = \langle PRAL \rangle + Despla + \langle RX \rangle \text{factor\_escala} \rightarrow$$

direccionamiento base + desplazamiento indexado

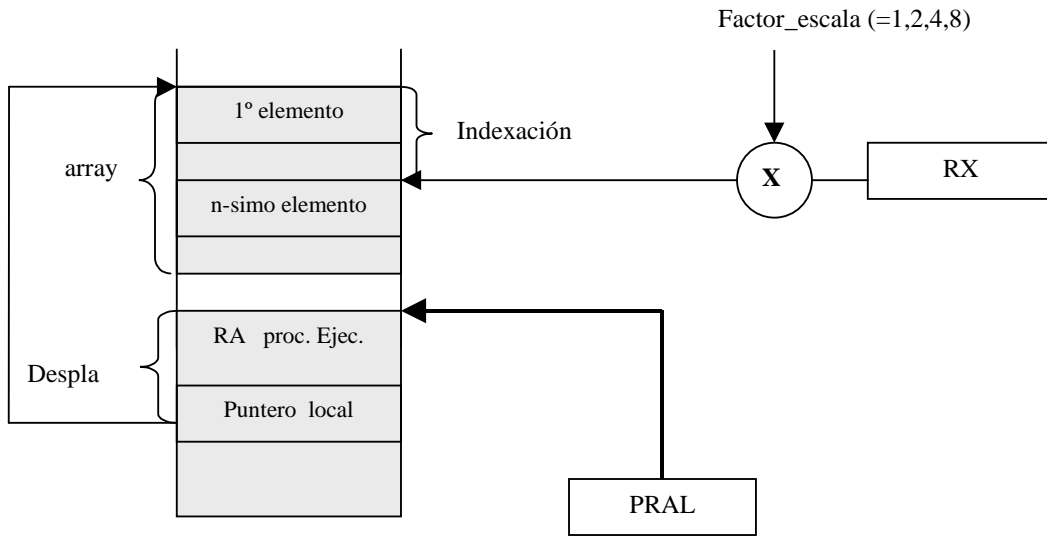
**Acceso a variables de tipo *array* no locales (contenido)**



$$DE = \langle \langle PRAL \rangle + Despla 1 \rangle + Despla 2 + \langle RX \rangle \text{factor\_escala} \rightarrow$$

direccionamiento base + desplazamiento indirecto + desplazamiento indexado

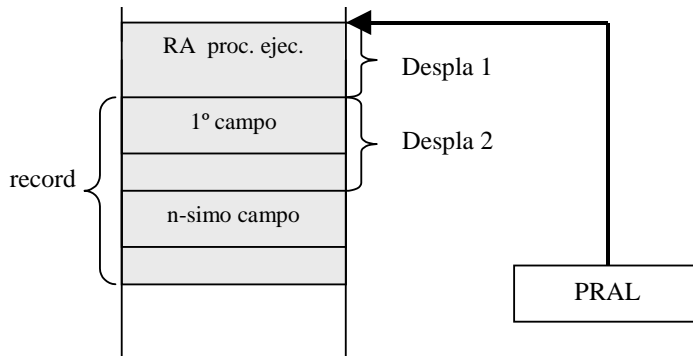
**Acceso a variables de tipo *array* locales (dirección)**



$$DE = \langle\langle PRAL \rangle\rangle + \langle\langle Despla \rangle\rangle + \langle\langle RX \rangle\rangle \times \text{factor\_escala} \rightarrow$$

direccionamiento base + desplazamiento indirecto indexado

**Acceso a variables de tipo *record* locales (contenido)**



$$DE = \langle\langle PRAL \rangle\rangle + \langle\langle Despla 1 \rangle\rangle + \langle\langle Despla 2 \rangle\rangle \rightarrow$$

direccionamiento base + desplazamiento

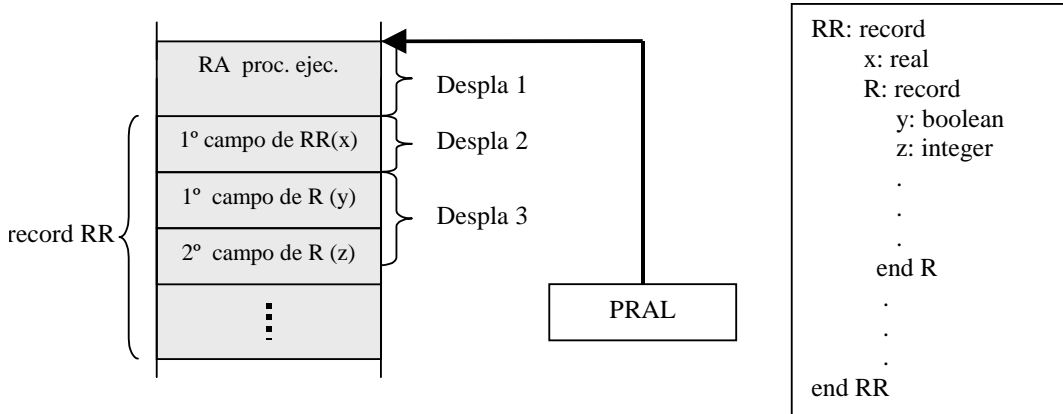
Despla 1 = dirección e inicio del *record*

(conocido en tiempo de compilación)

Despla 2 = posición en *record* del campo accedido

(conocido en tiempo de compilación)

**Acceso a variables de tipo *record* anidados locales (contenido)**



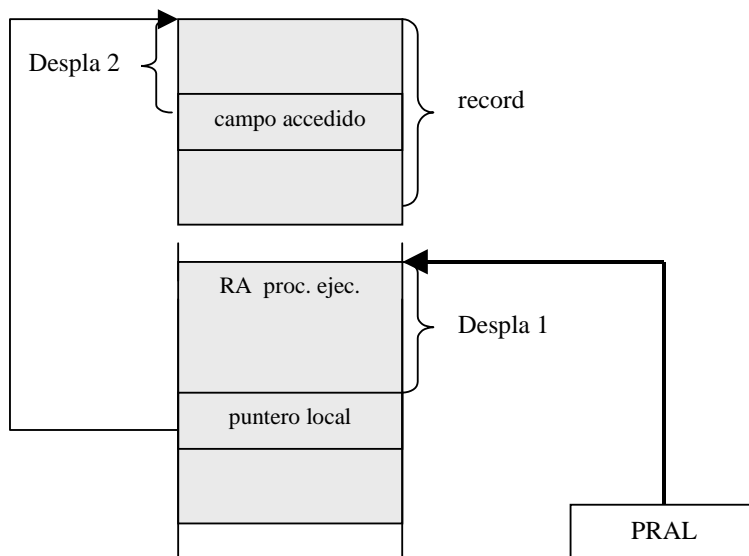
acceso al campo z de R

$$DE = \langle PRAL \rangle + Despla\ 1 + Despla\ 2 + Despla\ 23 \rightarrow$$

direccionamiento base + desplazamiento

- Despla 1 = dirección e inicio RR (conocido en tiempo de compilación)
- Despla 2 = posición en RR del campo R (conocido en tiempo de compilación)
- Despla 3 = posición en R del campo accedido z (conocido en tiempo de compilación)

**Acceso a variables de tipo *record* locales (dirección)**



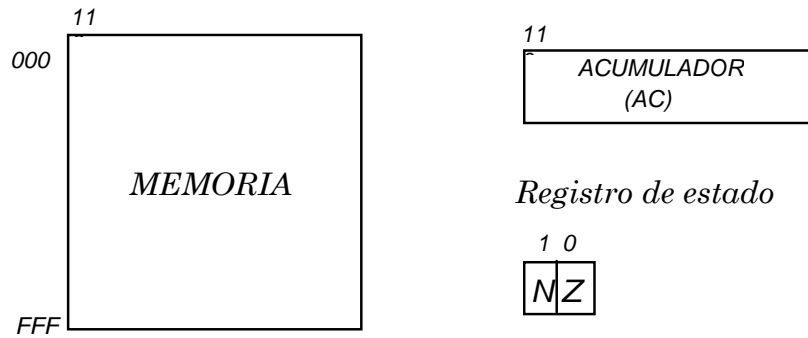
$$DE = \langle \langle PRAL \rangle + Despla\ 1 \rangle + Despla\ 2 \rightarrow$$

direccionamiento base + desplazamiento indirecto + desplazamiento



### Apéndice (motivación del direccionamiento indirecto)

Consideremos un computador con una memoria de 4K palabras de 12 bits, un registro acumulador y dos bits de condición (Z y N)



Todas las instrucciones tienen longitud fija de 12 bits y están compuestas por un código de operación (CO) situado en los 4 bits más significativos (del 8 al 11) y una *dirección/operando* situada en los 8 bits menos significativos (del 0 al 7). Los 8 bits de la *dirección/operando* tienen el significado de dirección en las instrucciones de salto y referencia a memoria, y significado de dato en las instrucciones de operando inmediato.



La máquina dispone de once instrucciones, tres de carga y almacenamiento (LDA, STA, LDAI), tres aritmético-lógicas (SUM, SUMI, NOR), cuatro de salto condicional (JZ, JNZ, JN, JNN) y una instrucción de parada (HALT). En la siguiente tabla se muestran el código simbólico, el tipo de direccionamiento, el código binario (CO) y la semántica de cada una de las instrucciones. El símbolo <-- significa transferencia de la información a su derecha hacia el elemento a su izquierda; MEMORIA(DIRECCION) expresa el contenido de la posición de MEMORIA apuntada por DIRECCION; los paréntesis angulares expresan contenido y el símbolo & concatenación.

Nombre Simbólico	DIR/OPE	COP	Semántica
LDA	DIR	0001	AC <-- MEMORIA(DIR)
STA	DIR	0010	MEMORIA(DIR) <-- <AC>
SUM	DIR	0011	AC <-- <AC> + MEMORIA(DIR)
LDAI	OPE	0100	AC <-- 0000&OPERANDO
SUMI	OPE	0101	AC <-- <AC> + 0000&OPERANDO
NOR	DIR	0110	AC <-- <AC> NOR MEMORIA(DIR)
JZ	DIR	0111	PC <-- DIR SI <Z> = 1
JNZ	DIR	1000	PC <-- DIRE SI <Z> = 0
JN	DIR	1001	PC <-- DIR SI <N> = 1
JNN	DIR	1010	PC <-- DIR SI <N> = 0
HALT	-	0000	parada de la máquina

**Programa**

Inicializa 10 posiciones de memoria (de la 20 a la 29) con un contenido igual a su dirección.

Dirección	Simbólico	Binario	comentario
0	LDA 4	0001 00000100	AC <-- (STA -----)
1	SUMI 1	0101 00000001	AC <-- (STA -----) + 1
2	STA 4	0010 00000100	MEMORIA(4) <-- (STA ----- + 1)
3	LDA 14	0001 00001110	AC <-- índice
4	STA ----	0010 00010011	MEMORIA(índice) <-- índice
5	SUMI 1	0101 00000001	AC <-- índice + 1
6	STA índice	0010 00001110	índice <-- índice + 1
7	LDA 30	0001 00001101	AC <-- límite
8	NOR 30	0110 00001101	AC <-- complemento1(límite)
9	SUMI 1	0101 00000001	AC <-- complemento2(límite)
10	SUM índice	0011 00001110	AC <-- índice - límite
11	JNZ 0	1000 00000000	Vuelve a dirección 0 si resultado ≠ 0
12	HALT	0000 00000000	Parada
13	30	000000011110	límite
14	20	000000010100	índice

Para referenciar posiciones de memoria consecutivas (indexación) hemos utilizado un artificio poco recomendable en programación: modificar instrucciones en tiempo de ejecución, concretamente, sumando un 1 a la instrucción *STA índice* (inicialmente en binario 001000010011) de la posición 4. De esa forma, cada vez que se recorra el cuerpo del ciclo que constituye el programa, la instrucción *STA índice* referenciará la posición de memoria siguiente a la que referenció en el recorrido anterior. El ciclo finalizará cuando el *índice* iguale el *límite*. Para detectarlo se realiza la resta *índice - límite* (complementando a dos *límite*, es decir, complementando a 1 y sumando 1, y sumando el resultado a *índice*) y se bifurca sobre Z.

Esto lo podemos solucionar introduciendo el direccionamiento indirecto (ind) para la instrucción de almacenamiento STA (ind) y modificando las instrucciones 0,1,2 y 4 del programa de la forma siguiente:

0	LDA 14
1	SUMI 1
2	STA 14
4	STA (ind) 14