MODULO II: ARQUITECTURA DEL PROCESADOR

Tema 2: Formato de instrucciones y modos de direccionamiento

Objetivos:

- Entender la arquitectura del repertorio de instrucciones (ISA) de un computador, los formatos de codificación, el lenguaje ensamblador y las máquinas virtuales.
- Describir los modos de direccionamiento de los datos en la memoria desde las instrucciones y las posibilidades que aporta cada modo.
- Estudiar los formatos de instrucciones y modos de direccionamiento de algunos procesadores comerciales, especialmente el ARM que utilizarán los alumnos en el laboratorio.
- Analizar la forma en que los modos de direccionamiento facilitan al compilador la construcción de las diferentes estructuras de datos en los lenguajes de alto nivel.

Contenido:

1. Introducción al repertorio de instrucciones

Las instrucciones máquina son las acciones elementales que puede ejecutar un computador. Una acción compleja deberá codificarse como una secuencia de instrucciones máquina en lo que se denomina un programa.

La arquitectura de un procesador entendida como el conjunto de recursos operativos disponibles por un programador a nivel de lenguaje máquina queda definida por el repertorio de instrucciones (*ISA: Instruction Set Architecture*).

En general, una instrucción codifica una *operación básica* que el computador realiza sobre unos *datos* ubicados en la memoria o en los registros de la máquina y a los que accede utilizando un *modo de direccionamiento*.

Por consiguiente, la arquitectura ISA de un procesador viene determinada por los siguientes factores:

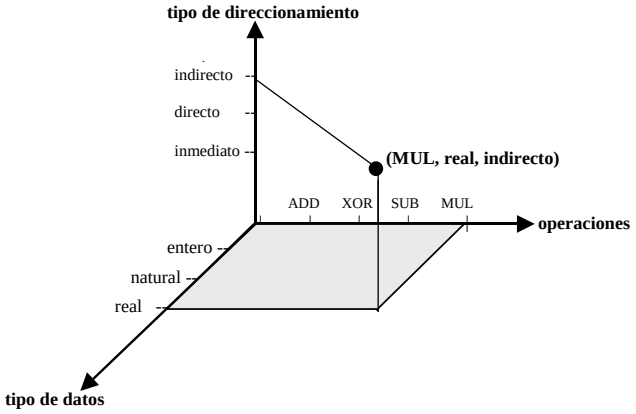
Tipos de datos y formatos que pueden manejar las instrucciones: naturales, enteros, reales, caracteres, etc.

Modos de direccionamiento de los datos en memoria: inmediato, directo, indirecto, etc. Estos dos factores son determinantes para la implementación eficiente de las estructuras complejas de datos de un lenguaje de alto nivel.

Conjunto básico de operaciones que se pueden realizar sobre los datos: suma, resta, etc.

Propiedad de ortogonalidad

Diremos que un repertorio es *ortogonal* cuando las instrucciones puedan combinar los valores de los tres factores anteriores sin ninguna restricción. La ortogonalidad completa no se da en ningún repertorio de máquina real.



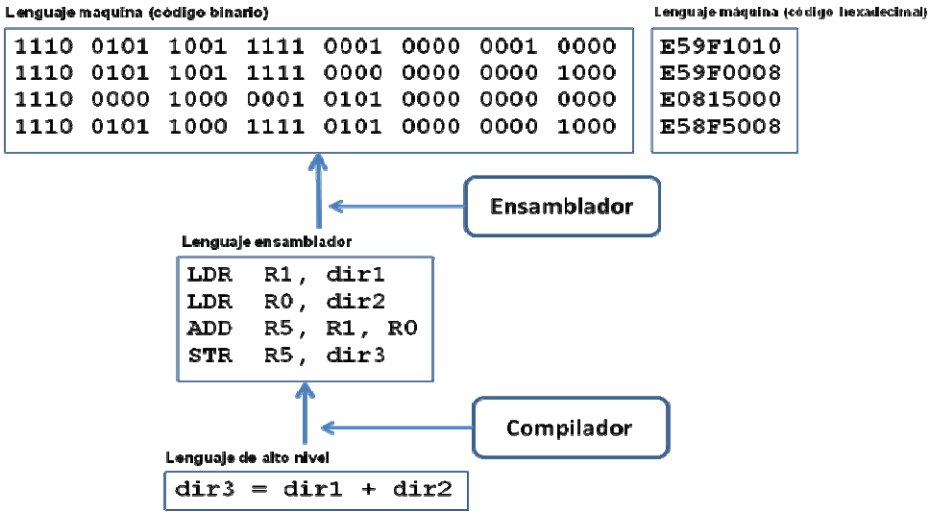
2. Lenguaje ensamblador y máquinas virtuales

Un computador podemos programarlo a diferentes niveles, si bien el programa final ejecutable siempre tiene que estar codificado en binario, en lenguaje máquina. La primera opción, y la más tediosa y propensa a errores, sería codificar el programa directamente en binario, utilizando la tabla binaria de códigos de operación del procesador y la codificación binaria de los operandos. Una segunda opción, que ahorraría sin duda errores de transcripción, consistiría en escribir el código binario en hexadecimal, sustituyendo cada 4 bits por el valor hexadecimal correspondiente. En este caso sería necesario un pequeño programa para traducir los caracteres hexadecimales (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F) a binario antes de la ejecución.

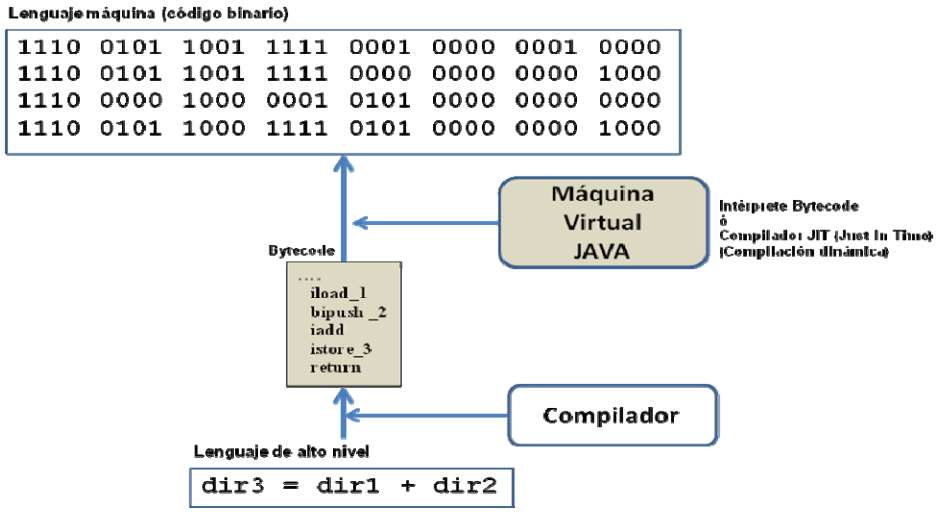
El primer paso hacia el uso de un lenguaje más cómodo y fiable para el programador sería utilizar el lenguaje ensamblador. En este caso podemos utilizar códigos simbólicos de operación que transmiten mejor su significado (por ejemplo, ADD en lugar de 1110, o E, para la suma). Los operandos también se pueden representar con nombres simbólicos (secuencia de caracteres) que aludan a su significado. El programa codificado en ensamblador habrá que traducirlo a lenguaje máquina (binario) antes de su ejecución, de lo cual se encarga el *Ensamblador*. Con esta alternativa se gana sin duda seguridad en la codificación del programa, pero las facilidades que aporta el *Ensamblador* no van más allá del uso de nombres simbólicos para los códigos de operación y los operandos. Sigue existiendo una correspondencia instrucción-a-instrucción entre el programa codificado en ensamblador y su traducción (ensamblado) binario.

Un paso más significativo para el programador consiste en utilizar un lenguaje de Alto Nivel (Pascal, C, etc.), lenguajes que proporcionan una sintaxis y una semántica más próximas al problema a resolver, y más alejadas del lenguaje máquina. En este caso necesitamos un Compilador que nos traduzca nuestro programa a código binario, bien directamente, bien pasando por código ensamblador. En cualquiera de los casos el código final que genera el compilador sólo será válido para el procesador concreto para el que se realice la compilación. Si queremos ejecutarlo en otro procesador tendremos que volverlo a compilar para dicho procesador.

En la siguiente figura hemos esquematizado las anteriores alternativas:



En la actualidad existe sin embargo otro modelo de ejecución muy utilizado en los programas que se diseñan para operar en la red y que aporta mayor portabilidad. Se trata de utilizar una máquina virtual, por ejemplo la Máquina Virtual de Java (JVM), entre el compilador y el leguaje máquina del procesador. Esta alternativa presenta la ventaja de la portabilidad del código generado para esta máquina virtual (Bytecode para JVM)), siempre que el comportamiento de la máquina virtual se defina dentro de un estándar admitido por la comunidad de usuarios. En la siguiente figura se representa el esquema de operación de la ejecución de un programa con máquina virtual:



El Bytecode que genera ahora el compilador no corresponde a ningún procesador concreto, sino a la máquina virtual, y para su ejecución en un procesador se pueden seguir dos alternativas: la interpretación o la compilación. En el primer caso la máquina virtual opera interpretando instrucción-a-instrucción el Bytecode. En el segundo se genera codigo nativo del procesador que después es ejecutado. En este segundo caso se suele utilizar una compilación de tipo JIT (Just In Time), que es una compilación dinámica que genera código a medida que se va necesitando, evitando la compilación completa de una aplicación que puede tener un gran tamaño si dispone de muchos modos de funcionamiento, como es usual en las aplicaciones actuales. Una gran ventaja de la aternativa compilada del Bytecode es que puede generar código nativo adaptado a los recursos particulares disponibles en el procesador concreto donde se realice la ejecución.

3. Formato de las instrucciones: diseño del repertorio

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:

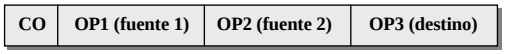
- Código de operación (CO)
- Operandos fuente (OP1, OP2,...)
- Operando destino o Resultado (OPd)
- Instrucción siguiente (IS)

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.

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

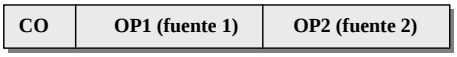
3.1. Primer criterio: número de operandos explícitos por instrucción

3.1.1. 3 operandos explícitos



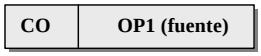
- ejemplo: ADD B,C,A A ← B + C
- Máxima flexibilidad
- Ocupa mucha memoria si los operandos no están en registros

3.1.2. 2 operndos explícitos



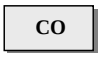
- ejemplo: ADD B, C C ← B + C
- Reduce el tamaño de la instrucción
- Se pierde uno de los operandos

3.1.3. 1 operando explícito



- ejemplo: ADD B Acumulador ← <Acumulador> + B
- Supone que fuente 1 y destino es un registro predeterminado (acumulador)
- Se pierde un operando fuente

3.1.4. 0 operandos explícitos



- ejemplo: ADD cima de pila ← <cima de pila> + <cima de pila - 1>
- Se trata de computadores que trabajan sobre una pila

3.1.5. 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 D, B
	MOV B, 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

3.2. Segundo criterio: forma de almacenar operandos en la CPU

3.2.1. Arquitectura de pila

- Ejemplo: HP 3000/70

3.2.2. Arquitectura de acumulador

- Ejemplo: Motorola 6809

3.2.3. Arquitectura de registros de propósito general

- Ejemplo: IBM 360

Ejemplo de código máquina para 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 PUSH B ADD POP C	LOAD A ADD B STORE C	LOAD A, R1 ADD B, R1 STORE R1, 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

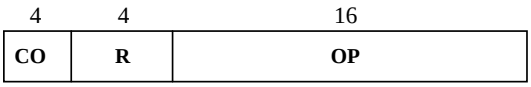
3.3. Códigos de operación de longitud fija y variable

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



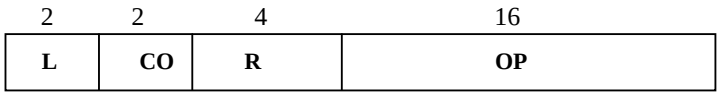
En este caso se pueden codificar 16 instrucciones de 2 operandos: uno en registro y el otro en memoria

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

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



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

3.3.1. 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:

Frecuencia de aparición en el programa ➔ optimización de memoria

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.

Podemos optimizar el CO utilizando la codificación de Huffman, que genera un código de longitud variable con la *propiedad de no superposición* de los CO resultantes. Es decir, garantiza que el CO de una determinada instrucción no coincide con la subcadena inicial de bits del CO de otra instrucción. La decodificación de un código de Huffman deberá realizarse de forma serie de izquierda a derecha.

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 su codificación necesitaría 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 nodo terminal del árbol de decodificación.

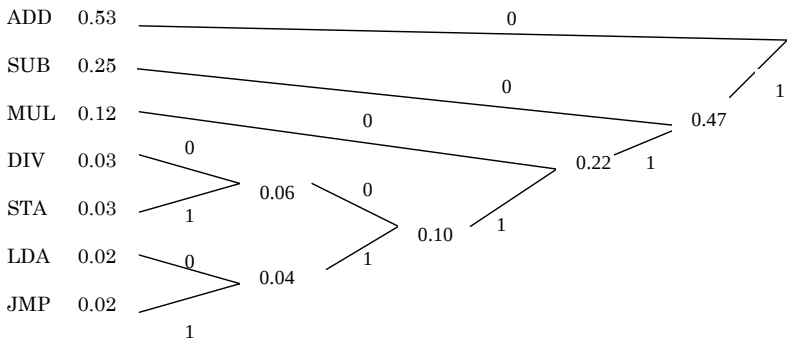
2) Se modifica la columna actual uniendo las dos frecuencias menores de dicha columna con sendos arcos, obteniéndose un nuevo nodo cuyo valor será la suma de los nodos de procedencia.

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

Resulta códigos de 1, 2, 3 y 5 bits con una longitud media l_n dada por la siguiente expresión:

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

$$1.89\_bits < 3\_bits$$

3.4. Propiedades generales del direccionamiento.

3.4.1. Resolución

Es la menor cantidad de información direccionable 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

3.4.2. 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 constituido por más de 1 byte.

3.4.2.1. Modo big-endian

El modo *big-endian*: almacena el byte más significativo del escalar en la dirección más baja de memoria. Lo utilizan los procesadores de Motorota, por ejemplo el MC68000

3.4.2.2. Modo little-endian

El modo *little-endian*: almacena el byte más significativo del escalar en la dirección más alta de memoria. Lo utilizan los procesadores de Intel, por ejemplo el Pentium.

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

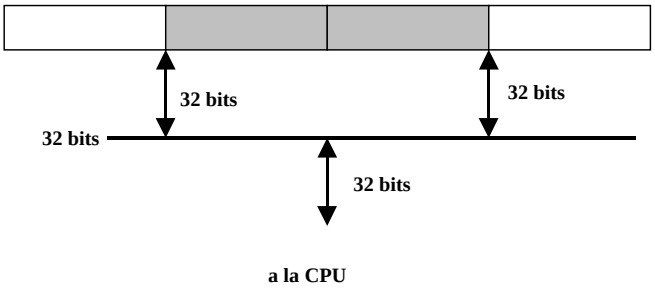
big-endian		little-endian	
184	12	184	78
185	34	185	56
186	56	186	34
187	78	187	12

3.4.3. Alineación

Un objeto de datos de *n* bytes ubicado en la dirección de memoria D se dice que está alineado si ***D mod 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



3.4.4. 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.

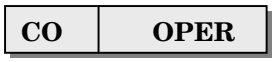
4. Modos de direccionamiento.

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

4.1. Implícito

• El dato operando se supone ubicado en algún lugar específico de la máquina, por ejemplo, una pila.

4.2. Inmediato.

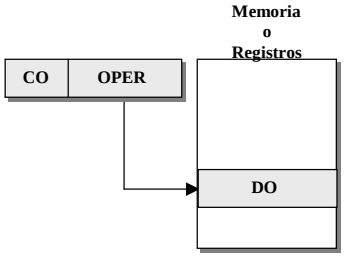


DO = OPER

- 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

4.3. Directo (memoria o registros)

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



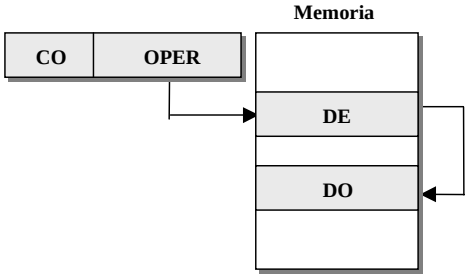
- La especificación de un registro requiere menor número de bits que la de una posición de memoria
- El acceso a los registros es más rápido que a Memoria
- El direccionamiento directo a memoria se conoce como absoluto
- A veces se limita el número de bits de OPER, limitando así el acceso a sólo una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

4.4. Indirecto (memoria)

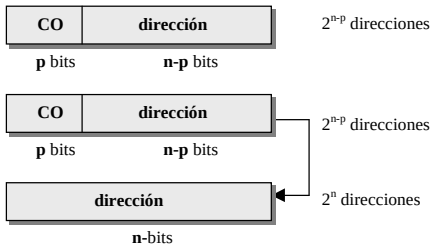
OPER = Dirección de memoria

DE = <OPER>

DO = <<OPER>>

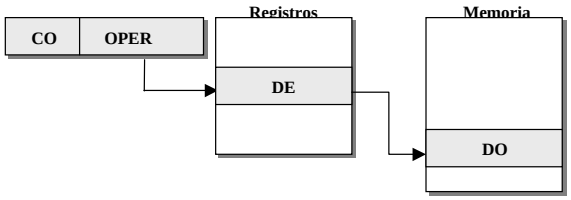


- Permite el tratamiento de una dirección de memoria como un dato
- Permite el paso por referencia de parámetros a subrutinas
- Permite referenciar un espacio mayor de direcciones



4.5. Indirecto (registro)

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



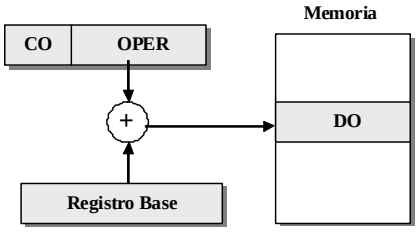
4.6. Modos con desplazamiento.

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.

4.6.1. Direccionamiento base más desplazamiento

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

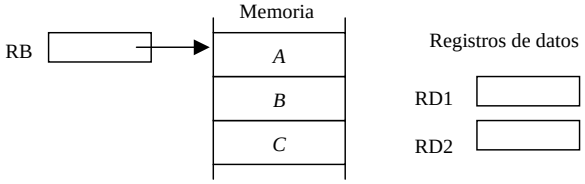
Se utiliza para la reubicación de datos y programas en memoria



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

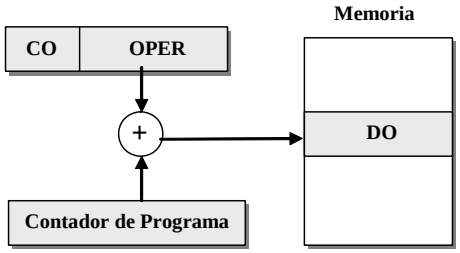


Este programa podremos reubicarlo en memoria y siempre llevará a la dirección apuntada por el registro base RB la suma de los contenidos de las dos direcciones siguientes. Los datos también pueden ser reubicados sin afectar la codificación del programa.

4.6.2. Direccionamiento relativo

DE = <Contador de programa> + OPER; OPER = desplazamiento

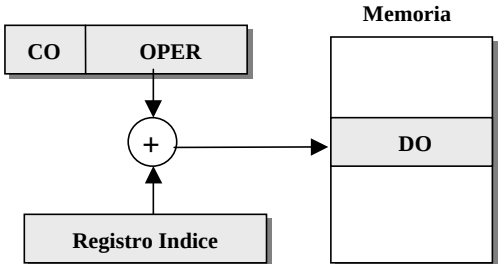
- Se utiliza en las instrucciones de salto para conseguir la reubicación de estas instrucciones
- El desplazamiento en estas instrucciones tiene signo (c2) lo que significa que el salto relativo se puede dar hacia posiciones anteriores o posteriores a la ocupada por la instrucción.



4.6.3. Direccionamiento indexado

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

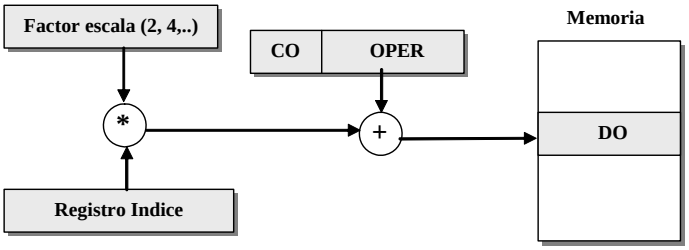
- Se utiliza para recorrer estructuras lineales como los *arrays*
- Par facilitar su uso se le suele añadir el *pre* o *post* incremento o decremento del registro índice



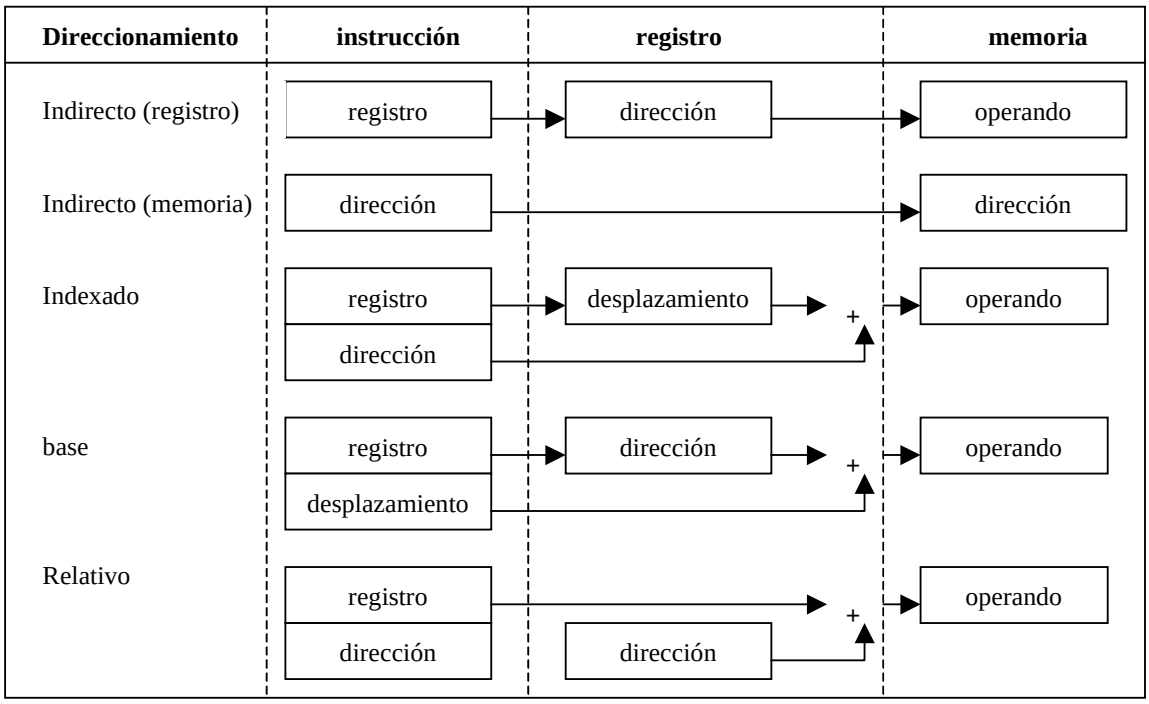
4.6.4. Direccionamiento indexado con factor de escala

DE = <Registro índice> *<Factor de escala>+ OPER; OPER = desplazamiento

- Se utiliza para recorrer estructuras lineales con elementos de 2, 4,.. palabras
- También se puede utilizar el *pre* o *post* incremento o decremento del registro índice



4.7. Resumen de los tipos de direccionamiento



4.8. Modos compuestos (MC 68.X)

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

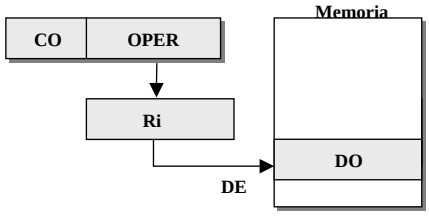
  - Registro

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

- Indirecto registro

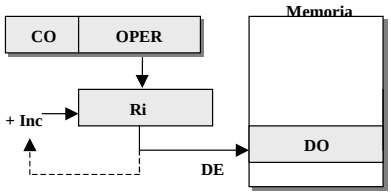
  - puro

    DE = <Ri>, con Ri cualquier registro



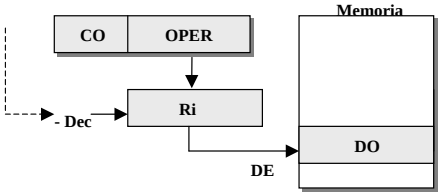
- Indirecto registro con postincremento

DE = <Ri>; Ri ← <Ri> + Inc, con Inc = 1, 2 ó 4 bytes



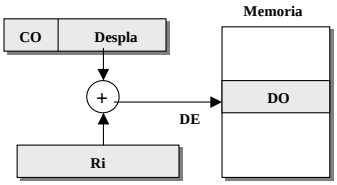
- Indirecto registro con predecremento

Ri ← <Ri> - Dec, DE = <Ri>, con Dec = 1, 2 ó 4 bytes



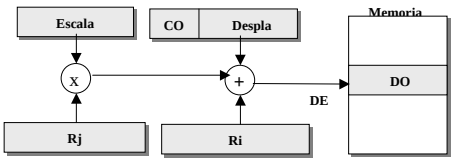
- <u>Indirecto con desplazamiento</u> (= base + desplazamiento)

DE = <Ri> + Despla



- <u>Indirecto registro indexado</u> (= base + desplazamiento indexado)

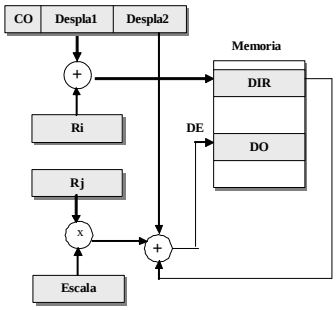
DE = <Ri> + <Rj>xEscala + Despla, con Escala = 1, 2, 4, 8 bytes



• Indirecto memoria

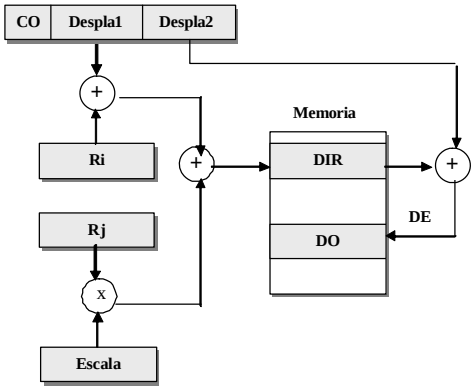
- <u>Postindexado</u> (= base + desplazamiento indirecto indexado + desplazamiento)

DE = <<Ri> + despla1> + <Rj>xEscala + Despla2



- Preindexado (base + desplazamiento indexado indirecto + desplazamiento)

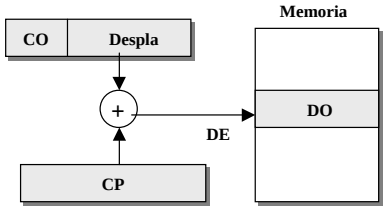
  DE = <<Ri> + Despla1 + <Rj>xEscala> + Despla2



• Relativo

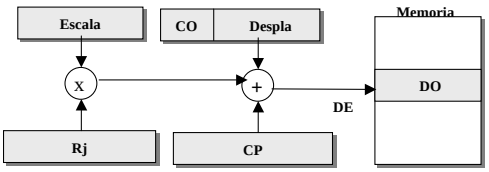
- Básico

  DE = <CP> + Despla



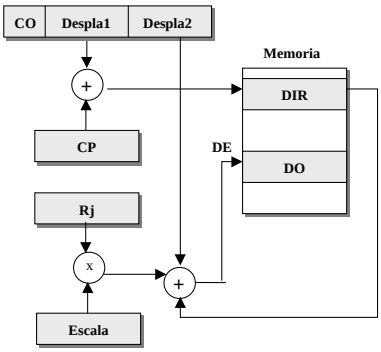
- Indexado con desplazamiento

  DE = <CP> + <Rj>xEscala + Despla



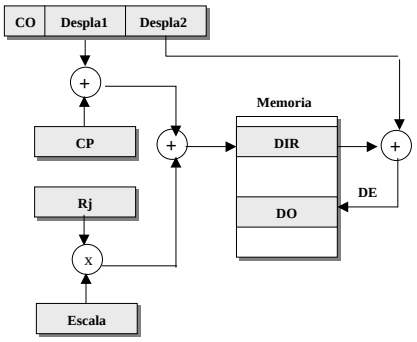
- Indirecto memoria postindexado

  DE = <<CP> + Despla1> + <Rj>xEscala + Despla2



- Indirecto memoria preindexado

  DE = <<CP> + Despla1 + <Rj>xEscala> + despla2



4.9. Modos de direccionamiento del MIPS R-2000

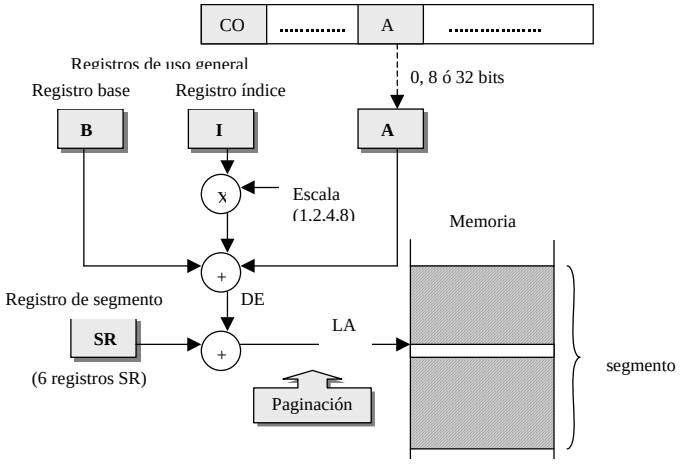
- Inmediato
- Registro

  LA = R
- Relativo

  DE = <PC> + Despla
- Indirecto registro con desplazamiento (= base + desplazamiento)

  DE = <Ri> + Despla

4.10.Modos de direccionamiento del Pentium II



- Inmediato

  DO = A (1,2,4 bytes)
- Registro

  LA = R, DO = <R> (LA = dirección lineal)
- Desplazamiento

  LA = <SR> + A
- Base

  LA = <SR> + <B>

- Base + desplazamiento

  LA = <SR> + <B> + A

- Indexado

  LA = <SR> + <I>xEscala + A

- Base + desplazamiento indexado

  LA = <SR> + <B> + <I> + A

- Base + desplazamiento indexado escalado

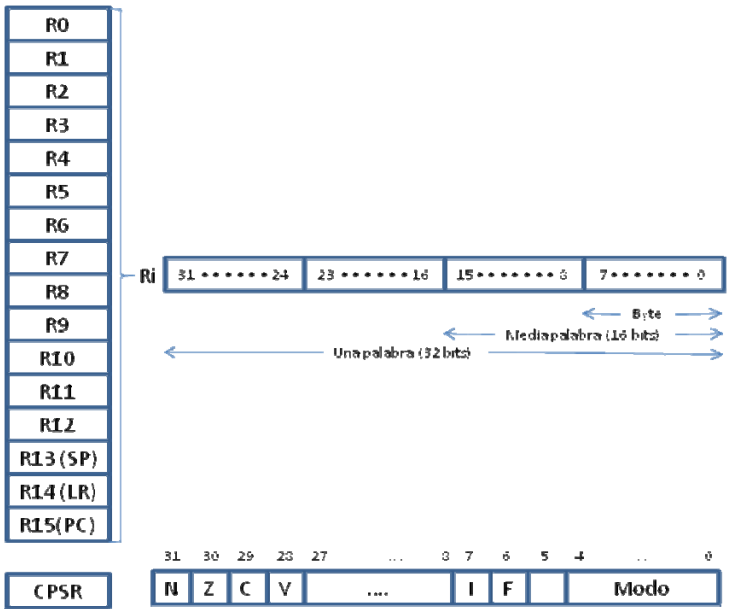
  LA = <SR> + <B> + <I>xEscala + A

- Relativo

  LA = <PC> + A

4.11.Modos de direccionamiento del ARM

El procesador ARM es un RISC con 16 registros de 32 bits, de los cuales 3 son de propósito específico (R13, R14 y R15) y los demás de propósito general. Además dispone de un registro de estado (CPSR) que soporta los bits de condición. Puede manipular datos de 1 byte, 2 bytes (media palabra) y 4 bytes (una palabra). El registro R15 es el contador de programa (PC), el R14 el registro de enlace (LR) y el 13 el stack pointer (SP). La siguiente figura esquematiza los registros del ARM



Distinguiremos entre los modos de direccionamiento asociados a operandos de datos (utilizados por las instrucciones de procesamiento de datos, que como todos los RISC tienen lugar sobre registros) y operandos de acceso a memoria (utilizados por las instrucciones de carga y almacenamiento de los registros desde memoria).

4.11.1. Operandos de procesamiento de datos

A) Valor no modificado

MOV R0, #1234 R0 := $1234_{(10}$

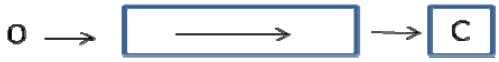
MOV R0, R1 R0 := R1

B) Desplazamiento lógico izquierdo

MOV R0, R1, LSL #2 R0 := R1 << 2 unidades
MOV R0, R1, LSL R2 R0 := R1 << R2 unidades

C) Desplazamiento lógico derecho

MOV R0, R1, LSR #2 R0 := R1 >> 2 unidades
MOV R0, R1, LSR R2 R0 := R1 >> R2 unidades

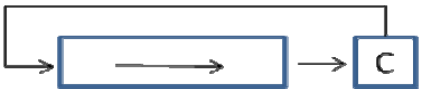


D) Desplazamiento aritmético derecho

MOV R0, R1, ASR #2 R0 := R1 << 2 unidades, manteniendo el signo
MOV R0, R1, ASR R2 R0 := R1 << R2 unidades, manteniendo el signo

E) Rotación derecha

MOV R0, R1, ROR #2 R0 := R1 rotado 2 unidades
MOV R0, R1, ROR R2 R0 := R1 rotado R2 unidades

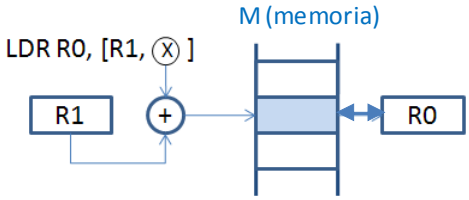


F) Rotación derecha extendida

MOV R0, R1, RRX R0 := R1 rotado 1 unidad

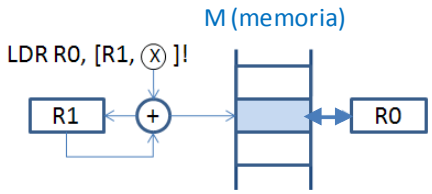
4.11.2. Operandos de acceso a memoria

A) Direccionamiento con desplazamiento (offset)



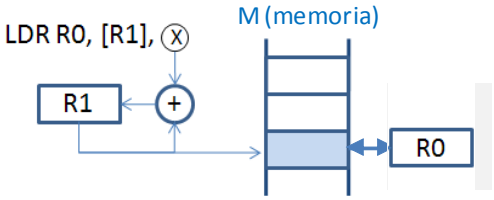

```
LDR     R0, [R1]              @ Carga R0 desde M[R1]
LDR     R0, [R1, #4]          @ Carga R0 desde M[R1 + 4]
LDR     R0, [R1, R2]          @ Carga R0 desde M[R1 + R2]
LDR     R0, [R1, R2, LSL #2]  @ Carga R0 desde M[R1 + R2 * 4]
STR     R0, [R1, #2]          @ Almacena R0 en M[R1 + R2 * 4]
```

B) Direccionamiento pre-indexado




```
LDR     R0, [R1, #4]!          @ Carga R0 desde M[R1+4] y actualiza  R1= R1+4
LDR     R0, [R1, R2]!          @ Carga R0 desde M[R1+R2] y actualiza  R1= R1+R2
LDR     R0, [R1, R2, LSL #2]!  @ Carga R0 desde M[R1+R2*4] y actualiza  R1= R1 + R2*4
STR     R0, [R1, #4]!          @ Almacena R0 en M[R1+4] y actualiza R1 = R1 + 4
```

C) Direccionamiento post-indexado




```
LDR     R0, [R1], #4           @ Carga R0 desde M[R1] y actualiza  R1= R1+4
LDR     R0, [R1], R2           @ Carga R0 desde M[R1] y actualiza  R1= R1+R2
LDR     R0, [R1], R2, LSL #2   @ Carga R0 desde M[R1] y actualiza  R1= R1+R2*4
STR     R2, [R5], #8           @ Almacena R2 en M[R5] y actualiza  R5= R5 + 8
```

5. Soporte de los modos de direccionamiento a los lenguajes de alto nivel

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.

5.1. Visibilidad en C y Pascal

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 las funciones (en la función Main) y pueden ser referenciadas desde cualquiera de ellas.

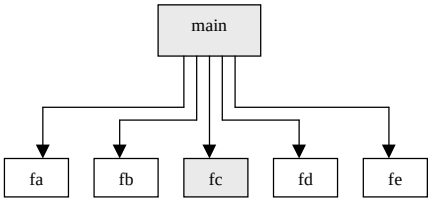
Las variables locales de una función se asignan dinámicamente, es decir, cada vez que se activa la función. La asignación tiene lugar dentro de un registro de activación (RA) que se ubica

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(...) ...
          .                 }                   }                       .
          .                                                             .
   }                                                                    .
                                                                      }
```

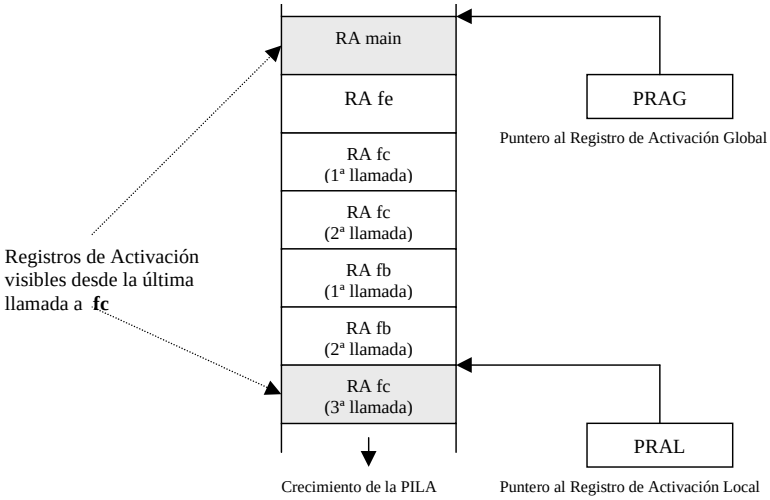
La estructura de este programa sería la siguiente:



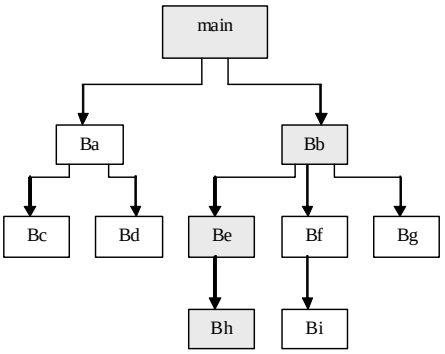
Supongamos que tiene lugar la siguiente secuencia de llamadas:

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)

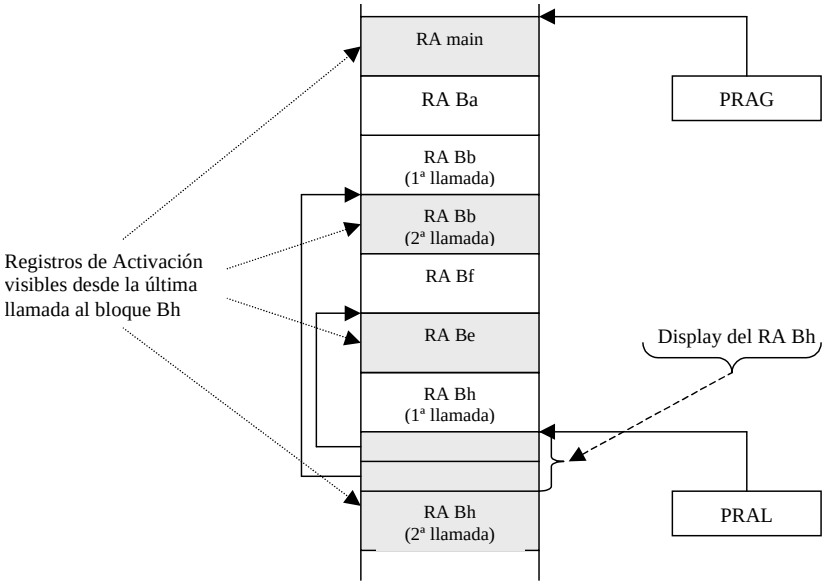


En los lenguajes de tipo Pascal un bloque puede ser un *procedure* o una *function*, y el concepto de visibilidad adquiere una estructura anidada. Los bloques de un programa de este tipo presentan una estructura jerárquica y la visibilidad de variables desde la llamada a un bloque se extiende a todos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



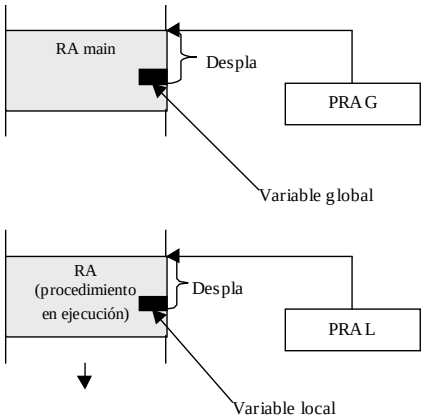

```
program main(..)
        procedure a
        begin
                procedure c
                begin
                ...
                end
                procedure d
                begin
                ...
                end
        end
        procedure b
        begin
                procedure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

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



El *display* de un RA contiene un puntero a cada RAi visible desde RA

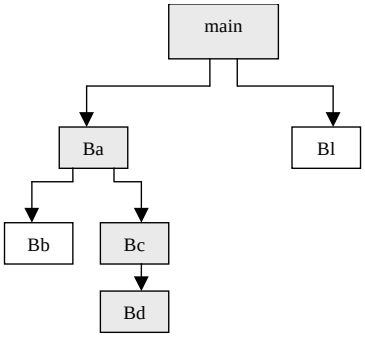
5.2. Acceso a variables escalares locales o globales (contenido)



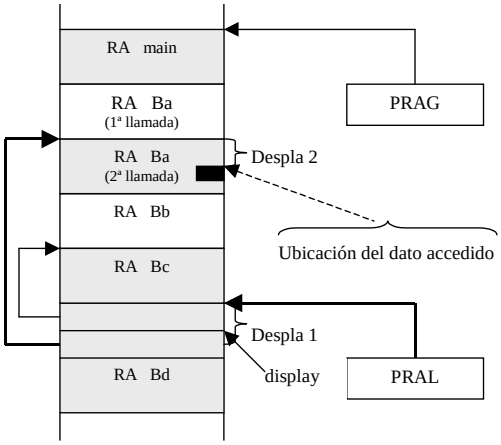
Variables locales: DE = <PRAL> + despla ➔ direccionamiento base + desplazamiento

Variables globales: DE = <PRAG> + despla ➔ direccionamiento base + desplazamiento

5.3. Acceso a variables escalares no locales (contenido)



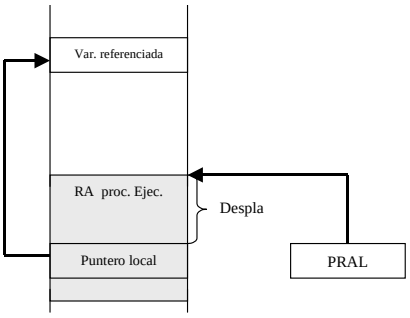
Secuencia de llamada: main → Ba → Ba → Bb → Bc → **Bd**



DE = <<PRAL> + Despla1> + Despla 2 ➔

direccionamiento base + desplazamiento indirecto + desplazamiento

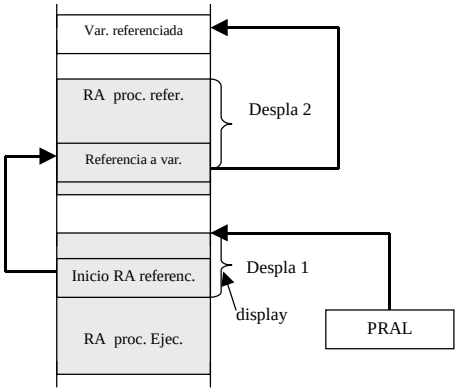
5.4. Acceso a variables escalares locales (dirección)



DE = <<PRAL> + Despla> ➔

direccionamiento base + desplazamiento indirecto

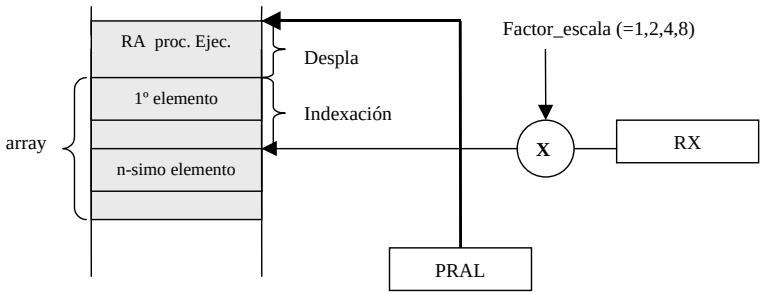
5.5. Acceso a variables escalares no locales (dirección)



DE = <<<PRAL> + Despla1> + Despla 2> ➔

direccionamiento base + desplazamiento indirecto + desplazamiento indirecto

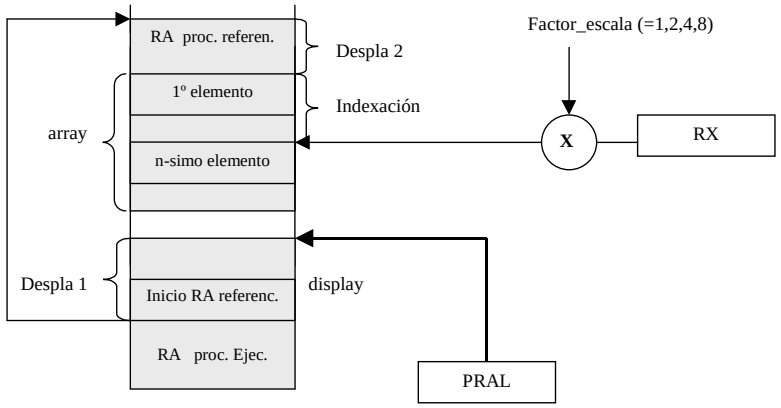
5.6. Acceso a variables de tipo *array* locales (contenido)



DE = <PRAL> + Despla + <RX>factor_escala ➔

direccionamiento base + desplazamiento indexado

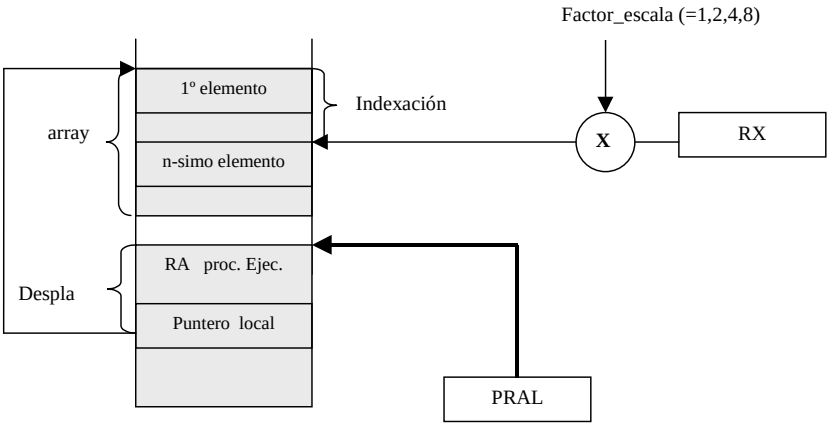
5.7. Acceso a variables de tipo *array* no locales (contenido)



DE = <<PRAL> + Despla 1> + Despla 2 + <RX>factor_escala ➔

direccionamiento base + desplazamiento indirecto + desplazamiento indexado

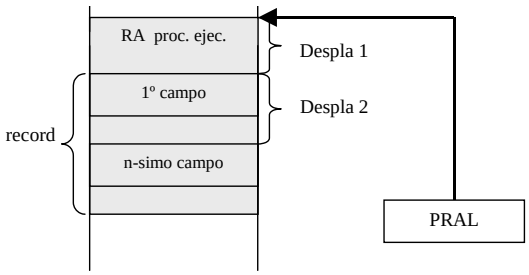
5.8. Acceso a variables de tipo *array* locales (dirección)



DE = <<PRAL> + Despla > + <RX>factor_escala ➔

direccionamiento base + desplazamiento indirecto indexado

5.9. Acceso a variables de tipo record locales (contenido)



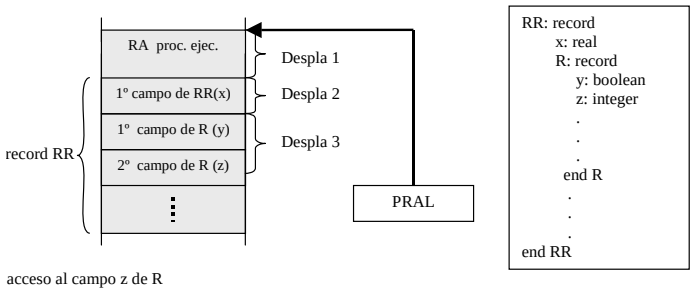
DE = <PRAL> + Despla 1 + Despla 2 ➔

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)

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



DE = <PRAL> + Despla 1 + Despla 2 + Despla 23➔

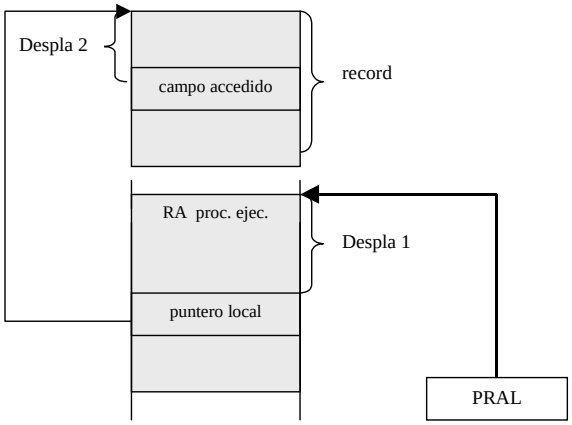
direccionamiento base + desplazamiento

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)

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

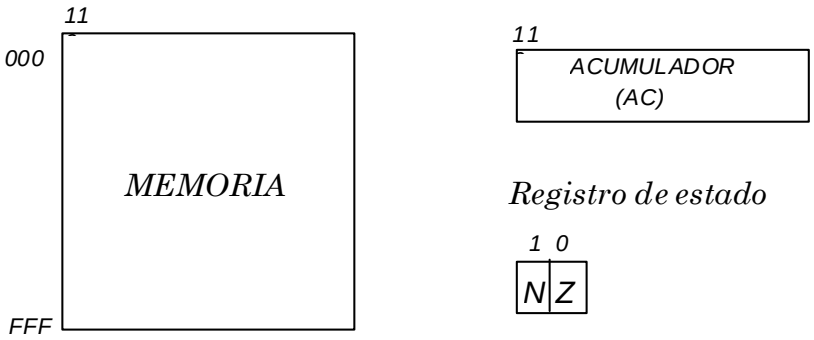


DE = <<PRAL> + Despla 1> + Despla 2 ➔

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 aritmetico-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 -limite* (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