Tema 3: Formato de los datos y clases de instrucciones

Objetivos:

- Estudiar los tipos de datos y formatos de representación que utilizan las instrucciones.
- Diferenciar las instrucciones que operan sobre datos, encargadas de su transformación, de las que controlan el flujo de ejecución, encargadas de evaluar las condiciones que determinan la trayectoria computacional del programa.
- Introducir el funcionamiento de las instrucciones SIMD para acelerar las operaciones de tipo matricial, fundamentales para las aplicaciones multimedia.
- Revisar la semántica de algunos repertorios, especialmente el del procesador ARM.
- Estudiar la forma en que las instrucciones de control dan soporte a las construcciones de control de alto nivel (*while*, *do*, etc.)

Contenido:

1. Tipos de datos y formatos de representación.

Cualquier dato, numérico o no-numérico, manipulado por un computador deberá representarse con dígitos binarios siguiendo un sistema de representación. Estudiaremos en primer lugar los métodos de representación de los diferentes tipos de números: naturales, enteros y reales. Después analizaremos la forma de representar información no- numérica.

1.1. Números naturales

Existen dos formas principales de representar números naturales en un computador: utilizando el sistema binario de numeración, o utilizando el decimal pero codificando cada digito en binario. Analizaremos en los apartados siguientes estos sistemas.

1.1.1. Representación binaria

$$N \equiv (x_{n-1}, x_{n-2}, \ldots, x_1, x_0)$$

$$V(N) = \sum_{i=0}^{n-1} 2^i \cdot x_i$$

- rango de representación = $[0, 2^n - 1]$
- resolución = 1

- dificultades:

  - el resultado de la suma puede necesitar n+1 bit

  - no es posible en general la resta: habría que comprobar que el minuendo es mayor que el sustraendo

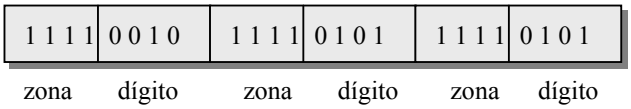
  - para el producto se necesitan 2n bits

1.1.2. Representación BCD (Decimal Codificado en Binario)

- Codifica en binario puro con 4 bits cada uno de los dígitos decimales
- Ejemplo 2 5 5 ---> BCD = 0010 0101 0101 en cambio en binario sería:

  2 5 5 ---> binario = 1111 1111

- Existen 2 formatos de representación:
  - BCD desempaquetado



  - BCD empaquetado

1.2. Números enteros

1.2.1. Signo y magnitud

$$N \equiv (x_{n-1}, x_{n-2}, ..., x_1, x_0)$$

$$V(N) = \sum_{i=0}^{n-2} 2^i x_i \qquad \text{si } Xn\text{-}1 = 0$$

$$V(N) = -\sum_{i=0}^{n-2} 2^i x_i \qquad \text{si } Xn\text{-}1 = 1$$

como

$$1 - 2 \bullet Xn\text{-}1 = 1 \quad si \quad Xn\text{-}1 = 0$$

$$1 - 2 \bullet Xn\text{-}1 = -1 \quad si \quad Xn\text{-}1 = 1$$

podemos poner:

$$V(N) = (1 - 2x_{n-1}) \sum_{i=0}^{n-2} 2^i x_i$$

- rango de representación = $[-(2^{n-1} - 1), 2^{n-1} - 1]$
- resolución = 1
- dificultades:
  - doble representación del cero
  - no es posible en general la resta: habría que comprobar que el minuendo es mayor que el

sustraendo

1.2.2. Complemento a dos

- Los números positivos se representan igual que en signo y magnitud
- Los números negativos se representan como 2^n - magnitud del número

Ejemplo: (para n= 8 bits) X = - 50 se representa por

1 0000 0000 = 2^8

- 0011 0010 = 50 (magnitud de -50)

---

0 1100 1110 = - 50 en c2

- El valor del número *X* se puede expresar en general como:

$$V(N) = -x_{n-1} \bullet 2^{n-1} + \sum_{i=0}^{n-2} 2^i x_i$$

Se puede obtener el valor decimal de un número en c2 sumando los pesos de las posiciones con 1:

-128	64	32	16	8	4	2	1
1	0	0	0	1	0	0	0
-128				+8			= -120

0 bien pesando sólo las posiciones con cero (cuando el número es negativo) y sumando 1:

64 + 32 + 16 + 4 + 2 + 1 + 1 = 120

Podemos calcular el c2 de un número binario complementando bit a bit (c1) y sumado 1:

0 1 0 0 0 0 0 1 = 65

c1 = 1 0 1 1 1 1 1 0

\+ 1

1 0 1 1 1 1 1 1 = -65

- rango de representación = [-2^{n-1} , 2^{n-1} - 1]
- resolución = 1
- ventaja: la resta se convierte en suma del c2 del sustraendo

1.2.3. Complemento a uno

- Los números positivos se representan igual que en signo y magnitud
- Los números negativos se representan como 2^n - magnitud del número - 1
- El valor del número *X* se puede expresar en general como:

$$V(N) = -x_{n-1} \bullet (2^{n-1} + 1) + \sum_{i=0}^{n-2} 2^i x_i$$

- rango de representación = [- 2^{n-1} - 1, 2^{n-1} - 1]
- resolución = 1
- dificultades:
  - doble representación del cero
  - la resta es más compleja que en c2

- Para hallar el c1 de un número binario se complementa bit a bit el número.

Tabla de representación de los tres sistemas para n = 4 bits

Número decimal	Signo y magnitud	Complemento a 1	Complemento a 2
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
0	0000	0000	0000
-0	1000	1111	No existe
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	no existe	No existe	1000

1.3. Números reales

Utilizando una notación exponencial, un número real *R* queda determinado por dos valores, un *exponente* (*e*) y una *mantisa* (*m*) tal que el valor de *R* venga dado por la expresión:

$$V(R) = B^{e} \bullet m$$

Donde *B* es una base conocida. Por ejemplo, utilizando base 10 tenemos:

$$976.000.000.000.000 = 9{,}76 \bullet 10^{14}$$
$$0{,}0000000000000976 = 9{,}76 \bullet 10^{-14}$$

Los números quedarían definidos por:

```
976.000.000.000.000     →    mantisa     m = 9,76;  exponente    e =  14
0,0000000000000976      →    mantisa     m = 9,76;  exponente    e = -14.
```

La representación en coma flotante utiliza esta definición y establece un formato para la mantisa y para el exponente.

1.3.1. Exponente

Utiliza una representación sesgada o *codificación en exceso* con un sesgo = $2^{k-1}-1$; siendo *k* el número de bits reservados para el campo del exponente. Es decir, en lugar de representar directamente *e*, utilizamos su codificación *exceso 127*: *E = e + 127*.

Para *k = 8* bits se codifica el exponente (con signo) con valores que van de *–127* a *+128* utilizando el código exceso $2^{8-1}-1 = 127$. Los correspondientes valores codificados van en decimal de *0* (todos los bits a 0) a *255* (todos los bits a 1). La tabla siguiente muestra el rango de valores del exponente y su codificación en exceso 127:

E (binario)	E (decimal)	e = E-127
0000 0000	0	-127
0000 0001	1	-126
0000 0010	2	-125
.	.	.
.	.	.

.	.	.
0111 1110	126	-1
0111 1111	127	+0
1000 0000	128	+1
.	.	.
.	.	.
.	.	.
1111 1111	255	+128

Una propiedad importante de la codificación sesgada es que se mantiene el orden natural de los vectores binarios en relación con su magnitud: el más grande es el 1111 1111 y el más pequeño 0000 0000, cosa que no ocurre en la representación en signo y magnitud, c1 o c2. Esta propiedad es importante a la hora de comparar números en coma flotante.

1.3.2. Mantisa

La mantisa *m* se representa en coma fija normalizada (signo y magnitud), es decir, haciendo que el primer dígito decimal valga 1, y ajustando convenientemente el exponente:

mantisa *m* → mantisa normalizada $M = \pm 0{,}1bbb...b$

con $b \in \{ 0, 1 \}$

Como el primer bit de la mantisa es siempre *1* no necesita almacenarse en el campo *M*, por lo que se pueden representar mantisas con *24* bits con valores en decimal que van de:

0,5 correspondiente a la mantisa 0,100000000000000000000000

a

1 (en realidad $1- 2^{-24}$) correspondiente a la mantisa 0,111111111111111111111111

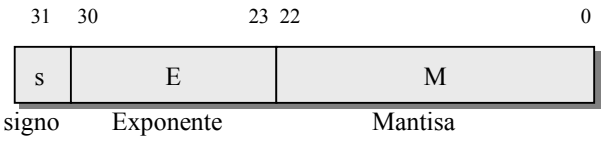
1.3.3. Base

Se suele utilizar la base 2. (la arquitectura IBM S/390 utiliza base 16)

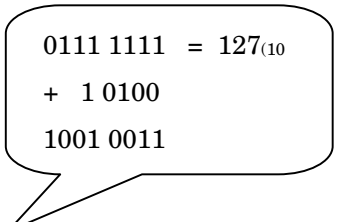
Teniendo en cuenta lo anterior, un número real R vendría representado por:

$$V(R) = (-1)^s \bullet 2^{E - 127} \bullet M$$

con el siguiente formato para una longitud de palabra de 32 bits (1 para el signo de la mantisa, 8 para el exponente, 23 para la magnitud de la mantisa):



Por ejemplo:



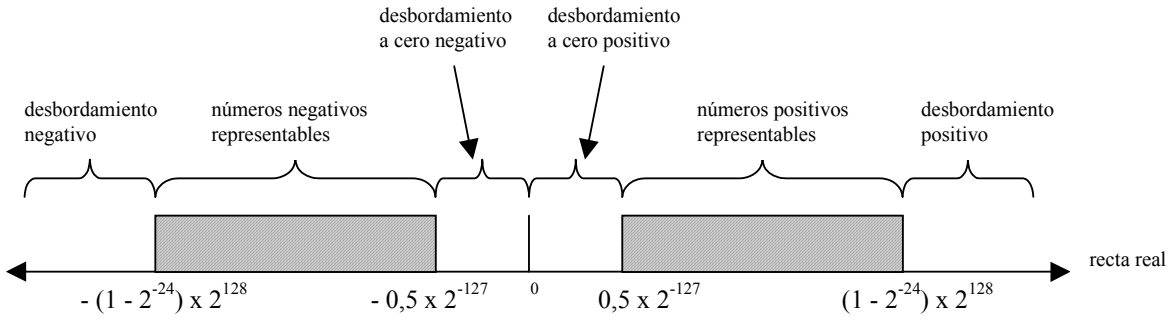
$0{,}11010001 \times 2^{10100}$	=	0	10010011	10100010000000000000000
$-0{,}11010001 \times 2^{10100}$	=	1	10010011	10100010000000000000000
$0{,}11010001 \times 2^{-10100}$	=	0	01101011	10100010000000000000000

$-0{,}11010001 \times 2^{-10100}$ = 1 01101011 10100010000000000000000

corresponden a las expresiones decimales:

$0{,}8164062 \times 2^{20}$ = 856063,95
$-0{,}8164062 \times 2^{20}$ = - 856063,95
$0{,}8164062 \times 2^{-20}$ = 0,000000778585
$-0{,}8164062 \times 2^{-20}$ = - 0,000000778585

Los ***rangos*** de números que pueden representarse en coma flotante con 32 bits y el formato descrito aparecen en la siguiente gráfica:



En esta gráfica se aprecia la existencia de 5 regiones excluidas de dichos rangos:

- región de desbordamiento negativo
- región de desbordamiento a cero negativo
- región de desbordamiento a cero positivo
- región de desbordamiento positivo

1.3.4. Problemas de la representación en coma flotante

La anterior representación no contempla un valor para el cero. Sin embargo, en la practica se destina una configuración de bits especial, como veremos en el estándar del IEEE.

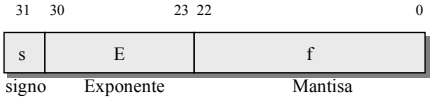
Aparecen dos tipos de desbordamientos, *el desbordamiento propiamente dicho* que ocurre cuando una operación aritmética da lugar a un número cuyo exponente es mayor que 128 (ejemplo 2^{215}), y el *desbordamiento a cero* que ocurre cuando una magnitud fraccionaria es demasiado pequeña (ejemplo 2^{-215})

Los números representados no están espaciados regularmente a lo largo de la recta real. Están más próximos cerca del origen y más separados a medida que nos alejamos de él.

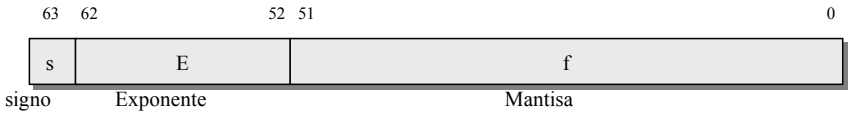
Tenemos que elegir el compromiso entre rango de representación y precisión de la misma repartiendo convenientemente los bits entre mantisa y exponente. Si aumentamos los bits del exponente aumentamos el rango pero disminuimos la densidad de números representados en dicho rango, ya que la cantidad total de valores representables lo determina el número total de bits (32).Hay que notar que con la representación en coma flotante no se representan más valores individuales. Con 32 bits sólo podemos representar 2^{32} valores diferentes. La coma flotante distribuye estos valores entre dos intervalos, uno positivo y otro negativo.

1.3.5. Estándar IEEE 754

Desarrollado para facilitar la portabilidad entre diferentes procesadores de programas numéricos. La base implícita es 2 y utiliza dos formatos, uno simple de 32 bits y otro doble de 64 bits. Además define dos formas ampliadas de estos formatos (dependientes de la implementación).



Formato simple



Formato doble

En el formato simple, los valores de exponente de 1 a 254 representan números en coma flotante normalizados distintos de cero, con exponente sesgado en un rango de -127 a +127 y mantisa con un 1 implícito a la izquierda del punto decimal, lo que significan 24 bits efectivos.

Ejemplo:

R_{IEEE} = 1 10000001 01000000000000000000000

↓ ↓ ↓

- e = 129 - 127 = 2 f = $0.01_{(2} = 0.25_{(10}$ => 1.f = $1.25_{(10}$ => $R_{(10}$ = $-1.25 \bullet 2^{2} = -5$

En este estándar no todos los patrones de bits se interpretan como valores numéricos, algunas se utilizan para representar valores especiales:

1. Un exponente cero con una fracción cero representa al +0 ó -0, dependiendo del bit de signo.
2. Un exponente todo unos con una parte fraccionaria cero representa al $+\infty$ o al $-\infty$, dependiendo del bit de signo.
3. Un exponente de todo unos con una fracción distinta de cero se conoce como NaN (Not a Number) y se emplea para indicar varias condiciones de excepción.
4. Un exponente cero con una parte fraccionaria distinta de cero representa un número denormalizado. En este caso el bit a la izquierda de la coma es cero y el exponente original -126.

En la siguiente tabla resumimos la interpretación de los números en coma flotante y simple precisión:

precisión simple (32 bits)				
	signo	exponente	fracción	valor
positivo normalizado ≠0	0	0<E<255	f	$2^{E-127}(1,f)$
negativo normalizado ≠0	1	0<E<255	f	$-2^{E-127}(1,f)$
cero positivo	0	0	0	0
cero negativo	1	0	0	-0
más infinito	0	255(todos 1s)	0	∞
menos infinito	1	255(todos 1s)	0	$-\infty$
NaN silencioso	0 ó 1	255(todos 1s)	f≠0	NaN
NaN indicador	0 ó 1	255(todos 1s)	f≠0	NaN
positivo denormalizado	0	0	f≠0	$2^{E-126}(0,f)$

negativo denormalizado	1	0	f≠0	$-2^{E-126}(0,f)$

Infinito

n + (+ ∞) = + ∞	n ÷ (+∞) = +0
n - (+ ∞) = - ∞	(+ ∞)+ (+ ∞) = + ∞
n + (- ∞) = - ∞	(- ∞)+ (- ∞) = - ∞
n - (- ∞) = + ∞	(- ∞)- (+ ∞) = - ∞
n x (+ ∞) = + ∞	(+ ∞)- (- ∞) = + ∞

NaN indicadores y silenciosos

Un NaN es un valor simbólico codificado en formato de coma flotante. Existen dos tipos: indicadores y silenciosos. Un NaN indicador señala una condición de operación no válida siempre que aparece como operando. Permiten representar valores de variables no inicializadas y tratamientos aritméticos no contemplados en el estándar.

Los NaN silenciosos se propagan en la mayoría de las operaciones sin producir excepciones. La siguiente tabla indica operaciones que producen un NaN silencioso:

Operación	
Suma o resta	(+ ∞)+ (- ∞) (- ∞)+ (+ ∞) (+ ∞)- (+ ∞) (- ∞)- (- ∞)
Multiplicación	0 x ∞
División	0/0 ∞/∞
Resto	x RE 0 ∞ RE y
Raíz cuadrada	$\sqrt{x}$ con $x < 0$

Números denormalizados

Se incluyen para reducir las situaciones de desbordamiento hacia cero de exponentes. Cuando el exponente del resultado es demasiado pequeño se denormaliza desplazando a la derecha la parte fraccionaria e incrementando el exponente hasta que el exponente esté dentro de un rango representable.

1.4. Caracteres

1.4.1. ASCII (American Standard Code for Information Interchange)

Utiliza un patrón de 7 bits con el que se pueden representar hasta 128 caracteres diferentes, suficientes para codificar los caracteres del alfabeto incluyendo signos especiales y algunas acciones de control. El patrón 011XXXX representa los dígitos decimales del 0 al 9 con los

correspondientes valores binarios para XXXX.. Así, 011 0000 representa al 0; 011 0001 representa al 1; etc.

1.4.2. EBCDIC(Extended Binary-Coded-Decimal Interchange Code)

Se trata de un código de 8 bits utilizado por IBM. En este caso los dígitos decimales del 0 al 9 vienen codificados por los patrones del 1111 0000 al 1111 1001.

Como se puede observar en ambos casos la representación es compatible con la BCD empaquetada de 4 bits. Además, el valor binario (sin signo) de ambos códigos mantiene el orden de los símbolos correspondientes a los dígitos decimales y caracteres, lo que facilita la ordenación de información simbólica en los programas.

1.5. Compresión de datos

La representación de datos comprimidos ha ido adquiriendo cada vez más importancia debido a la utilización de mayores cantidades de información, sobre todo en las aplicaciones multimedia. Comprimiendo datos se optimiza tanto la memoria utilizada para su almacenamiento como el tiempo de transmisión por las redes. Uno de los métodos de compresión más conocido es el llamado de Lempel-Ziv (LZ), que aprovecha de forma bastante simple las repeticiones de ciertos patrones de bits que aparecen en una cadena. Se aplica a cualquier tipo de información representada por una cadena de bits y se realiza en los siguientes pasos:

1) Se descompone la cadena de modo que no se repita el mismo patrón de bits. Para ello:

a) se coloca un separador después del primer dígito

b) el siguiente separador se coloca de modo que el fragmento entre separadores sea el más corto posible que no haya aparecido previamente.

El proceso continúa hasta fragmentar toda la cadena.

Ejemplo: la cadena de bits

```
10110010100110100 1000
```

se fragmentaría de la forma siguiente:

```
1-0-11-00-10-100-110-1001-000
```

2) Se enumeran los fragmentos.

```
 1 2  3   4   5   6    7    8     9
1-0-11-00-10-100-110-1001-000
```

Cada fragmento es siempre la concatenación de un *fragmento prefijo* aparecido con anterioridad, y de un *bit adicional,* 0 ó 1. Por ejemplo, el fragmento número 3, 11, es el número 1 seguido de un 1; el número 6, 100, es el 5 seguido de un 0; y así sucesivamente.

3) Se sustituye cada fragmento por el par *(número de prefijo, bit adicional).* El 0 indicará el *prefijo vacío*. En nuestro ejemplo:

```
(0,1)-(0,0)-(1,1)-(2,0)-(1,0)-(5,0)-(3,0)-(6,1)-(4,0)
```

4) Se codifican en binario los números de los prefijos.

En nuestro ejemplo, como aparecen siete prefijos, se necesitan tres bits para codificarlos:

```
(000,1)-(000,0)-(001,1)-(010,0)-(001,0)-(101,0)-(011,0)-(110,1)-(100,0)
```

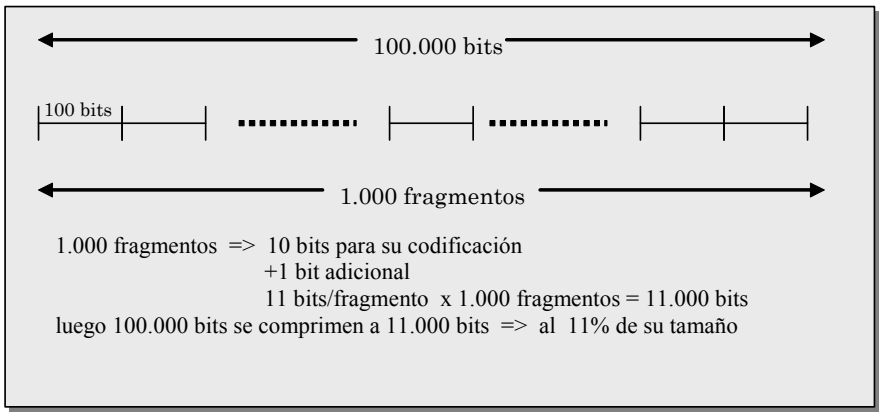
5) Se quitan los paréntesis y los separadores y obtenemos la cadena de bits comprimida:

```
000100000011010000101010011011011000
```

La descompresión se realiza siguiendo el proceso inverso y conociendo el número de bits de la codificación del prefijo, tres en nuestro ejemplo. Sabemos entonces que cada fragmento está codificado por cuatro bits, tres para el prefijo y uno para el bit adicional:

```
   0001000000110100001010100110110110000
1) 0001-0000-0011-0100-0010-1010-0110-1101-1000
2) (000,1)-(000,0)-(001,1)-(010,0)-(001,0)-(101,0)-(011,0)-(110,1)-(100,0)
3) (0,1)-(0,0)-(1,1)-(2,0)-(1,0)-(5,0)-(3,0)-(6,1)-(4,0)
   1 2  3  4 .......
4) 1-0-11-00 .......
```

En el ejemplo anterior la cadena resultante es más larga que la original, en contra del propósito del método. La causa es la poca eficiencia del método para cadenas muy cortas. Cuando se aplica a una cadena muy larga, los fragmentos crecen en longitud más rápidamente que las subcadenas de bits necesarias para su codificación. Por ejemplo, supongamos que una cadena de 100.000 bits se ha dividido en 1.000 fragmentos con un tamaño medio de 100 bits (los primeros serán probablemente más cortos y los últimos más largos). Para codificar el prefijo solo se necesitan 10 bits (2^{10} = 1024). Por tanto cada fragmento estará codificado por 11 bits (los 10 del prefijo más el bit adicional), mientras que su longitud original era de 100 bits. Es decir, el método lograría comprimir el fichero de datos a un 10% de su tamaño original.



Está claro que cuando los fragmentos sean más largos el método será más eficaz. También será más eficaz cuando haya muchas regularidades en la cadena de bits. Esto ocurre si la cadena tiene muchas repeticiones. Un caso extremo, por ejemplo, es una cadena con todo ceros. En este caso la fragmentación es muy simple:

```
0-00-000-0000-00000-000000-0000000-...
```

```
Ejemplo1: cadena de 28 ceros, se comprime en una de igual tamaño
0000000000000000000000000000
1  2  3    4    5      6      7
0-00-000-0000-00000-000000-0000000
(0,0)-(1,0)-(2,0)-(3,0)-(4,0)-(5,0)-(6,0)
000,0-001,0-010,0-011,0-100,0-101,0-110,0
000000100100011010001010110 0
Ejemplo2: cadena de 36 ceros, se comprime en una de 32 bits
```

```
0000000000000000000000000000000000
1  2  3    4    5      6      7        8
0-00-000-0000-00000-000000-0000000-00000000
(0,0)-(1,0)-(2,0)-(3,0)-(4,0)-(5,0)-(6,0)-(7,0)
000,0-001,0-010,0-011,0-100,0-101,0-110,0-111,0
00000010010001101000101011001110
Ejemplo3: cadena de 55 ceros, se comprime en una de 50
0000000000000000000000000000000000000000000000000000000000
1  2  3    4    5      6      7        8        9         10
0-00-000-0000-00000-000000-0000000-00000000-000000000-0000000000
(0,0)-(1,0)-(2,0)-(3,0)-(4,0)-(5,0)-(6,0)-(7,0)-(8,0)-(9,0)
0000,0-0001,0-0010,0-0011,0-0100,0-0101,0-0110,0-0111,0-1000,0-1001,0
00000000100010000110010000101001100011101000010010
```

Una cadena con n ceros se dividiría en un número de fragmentos $m-1$ tal que:

$$n = 1+2+3+...+(m-1) = m(m+1)/2 \qquad \textit{(progresión aritmética)}$$

lo que implica que aproximadamente $m \approx \sqrt{2n}$.

Para 1MB, que son 8.388.608 bits, tendríamos 4.096 fragmentos y necesitaríamos sólo *12* bits para describir el prefijo. Por tanto, necesitaríamos 13 bits para describir cada fragmento o 4.096x13=53.248 bits para describir el fichero entero. El fichero habría quedado reducido a 6,65 KB, es decir, al 0,63 % de su tamaño original.

El caso opuesto es una cadena de bits cuya fragmentación contenga todas las subcadenas posibles. Por ejemplo, hay 128 subcadenas de 8 bits. Si la fragmentación resultara en una concatenación de todas ellas, se necesitarían 8 bits para describir el prefijo y 9 para describir el fragmento, mientras que la longitud original de los fragmentos es de 8 bits. El método resultaría perfectamente inútil en este caso.

El método LZ es más eficaz cuanto más regular sea la cadena, es decir, cuantas más repeticiones contenga. Esta característica está de acuerdo con el teorema de *Shannon*, que afirma que cuanto más regular y menos aleatoria sea una cadena (menor sea su entropía), mayor será su grado de compresión. *Shannon* demostró que para una cadena de n bits aleatorios y completamente independientes unos de otros la entropía vale:

$$H = -n[p \log_2 p + (1 - p) \log_2 (1 - p)]$$

donde p es la probabilidad de que cada bit sea un 1.

Como el programa WinZip de Windows utiliza alguna variante de este método, se puede comprobar empíricamente la fórmula de *Shannon* creando ficheros de cadenas de bits aleatorios con diferentes probabilidades p = 0,01; 0,1; *0,2,* etc. y comprimiendo los ficheros con WinZip. Después se podrá observar que los tamaños comprimidos están en la misma relación que los previstos en la fórmula de *Shannon*.

Ejemplo4: cadena de 55 ceros seguida de un 1, se comprime en una de 55 bits

```
00000000000000000000000000000000000000000000000000000000001
```

```
1  2  3    4    5      6      7        8        9         10     11
0-00-000-0000-00000-000000-0000000-00000000-000000000-0000000000-1
(0,0)-(1,0)-(2,0)-(3,0)-(4,0)-(5,0)-(6,0)-(7,0)-(8,0)-(9,0)-(0,1)
0000,0-0001,0-0010,0-0011,0-0100,0-0101,0-0110,0-0111,0-1000,0-1001,0-0000,1
00000000100010000110010000101001100011101000010010000001
```

Ejemplo5: cadena de 42 bits con máxima variación, se comprime en una de 64

```
01000110110000010100111001011101111
1 2  3  4  5  6  7   8   9   10  11  12  13  14
0-1-00-01-10-11-000-001-010-011-100-101-110-111
(0,0)-(0,1)-(1,0)- (1,1-(2,0)-(2,1)-(3,0)-(3,1)-(4,0)-(4,1)-(5,0)-(5,1)-(6,0)-(6,1)
000,0-000,1-001,0-001,1-010,0-010,1-011,0-011,1-100,0-100,1-101,0-101,1-110,0-110,1
00000001001000110100010101100111100010011010101111001101
```

1.6. Tipos de datos en el 68000

Utiliza la alternativa *big-endian* para ordenar los bytes de las palabras (2 bytes) y dobles palabras (4 bytes), es decir, el byte menos significativo se almacena en la dirección más baja. Dispone de los siguientes tipos:

- Entero con signo en c2: de 1 byte, 2 bytes y 4 bytes.
- Entero sin signo: de 1 byte, 2 bytes y 4 bytes.
- BCD empaquetado (2 dígitos de 4 bits en un byte)

Carece de datos en formato de coma flotante

1.7. Tipos de datos en el Pentium II

Puede trabajar con tipos de datos de 8 (byte), 16 (palabra), 32 (palabra doble) y 64 (palabra cuádruple) bits de longitud. Los tipos mayores de 1 byte no es necesario que estén alineados en memoria, el procesador se encarga de ello si no lo están. Utiliza la alternativa *little-endian* para ordenar los bytes de las palabras, dobles palabras y cuádruples palabras, es decir, el byte menos significativo se almacena en la dirección más baja. Admite una gran variedad de tipos:

- General: byte, palabra, doble palabra y cuádruple palabra con contenido binario arbitrario.
- Entero: valor binario con signo en c2 de una palabra, una doble palabra, o una cuádruple palabra.
- Ordinal: entero sin signo de un byte, una palabra o una doble palabra.
- BCD desempaquetado: un dígito BCD en cada byte
- BCD empaquetado: dos dígitos BCD en un byte.
- Campo de bits: secuencia contigua de bits independientes no alineada de hasta 2^{32} - 1 bits.
- Cadena de bytes: Secuencia contigua de bytes, palabras, o dobles palabras de hasta 2^{32} - 1 bytes.
- Coma flotante:

  - IEEE 754 de simple precisión (32 bits)
  - IEEE 754 de doble precisión (64 bits)
  - IEEE 754 de precisión ampliada(78 bits)

2. Instrucciones que operan sobre datos.

En este apartado revisaremos las instrucciones máquina que operan sobre datos. En el siguiente veremos las que controlan el flujo de ejecución. Podemos clasificar las instrucciones que operan sobre datos en los siguientes grupos:

- Movimiento o transferencia de datos
- Desplazamiento y rotación
- Lógicas y manipulación de bits
- Aritmáticas
- Transformación de datos
- Entrada/salida
- Manipulación de direcciones

En otro apartado estudiaremos las nuevas instrucciones paralelas que operan simultáneamente sobre un conjunto de datos del mismo tipo, conocidas como SIMD (*Single Instructions Multiple Data*) e introducidas por primera vez en el Pentium para acelerar las operaciones enteras de las aplicaciones multimedia (MMX), y ampliadas en el Pentium III a las operaciones paralelas de coma flotante (SSE: *Streaming SIMD Extension*), muy frecuentes en las aplicaciones visuales.

2.1. Instrucciones de movimiento o transferencia de datos

Se trata del tipo más básico de instrucción máquina. Transfieren el contenido de información entre elementos de almacenamiento: registros (REG), memoria (MEM) y PILA. Dependiendo de la fuente y destino de la transferencia estas instrucciones suelen recibir nombres diferentes que resumimos en la siguiente tabla:

REG --> REG: transferencia	MEM --> REG: carga (load)	PILA --> REG: extracción (pop)
REG -->MEM: almacenamiento (store)	MEM --> MEM: movimiento (move)	PILA --> MEM: extracción (pop)
REG --> PILA: inserción (push)	MEM --> PILA: inserción (push)	

Una instrucción de este tipo deberá especificar los siguientes elementos:

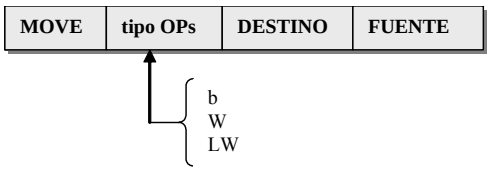
• Tipo de transferencia. Para especificar el tipo existen dos alternativas:

- Una instrucción genérica, con un único CO, siendo el operando el que especifica el tipo de movimiento. Ejemplo MOVE

- Una instrucción diferente para cada tipo de movimiento. Ejemplos TR, STO, LD, PUSH, POP

• Direcciones de la fuente y destino de la transferencia. Son los operandos de la instrucción con sus correspondientes tipos de direccionamiento

• Tipo y tamaño del dato transferido: byte, media palabra, una palabra, doble palabra, etc. Ejemplo MC 68.X

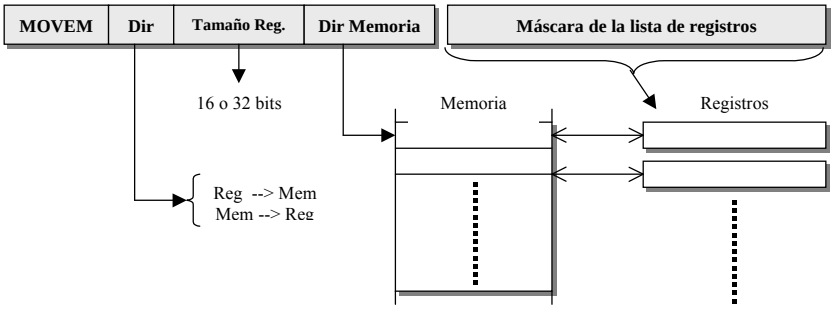


• Número de datos a mover:

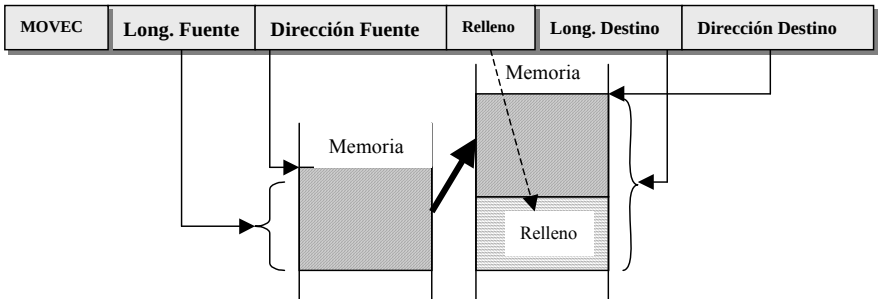
- Movimiento simple. Como el ejemplo anterior

- Movimiento múltiple

Ejemplo: MC 68.X



Ejemplo VAX-11



Long. Fuente = Long. Destino => transferencia total

Long. Fuente > Long. Destino => transferencia parcial: finaliza cuando se agota el destino

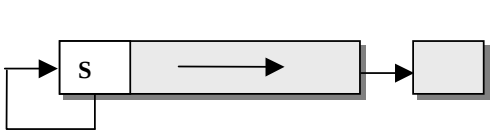
Long. Fuente < Long. Destino => transferencia total: se completa con el carácter de relleno

2.2. Instrucciones de desplazamiento y rotación

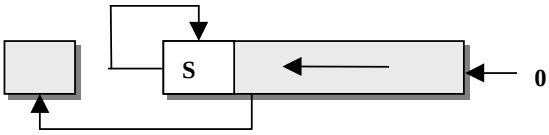
Una instrucción de este tipo deberá especificar los siguientes elementos:

- Tipo de operación: desplazamiento aritmético, lógico, rotación (especificado en CO)
- Cuenta: número de posiciones (normalmente 1 bit, especificado en CO)
- Dirección: izquierda, derecha (especificado en CO o en el signo de Cuenta)
- Tipo de dato: normalmente su longitud

- Desplazamientos aritméticos



desplazamiento aritmético derecha



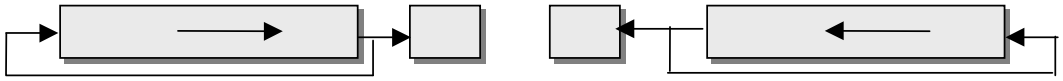
desplazamiento aritmético izquierda

- Desplazamientos lógicos

desplazamiento lógico derecha

desplazamiento lógico izquierda

- Rotaciones



rotación derecha

rotación izquierda

2.3. Instrucciones lógicas y de manipulación de bits

Son instrucciones para la manipulación de bits individuales dentro de una unidad direccionable (byte, palabra, etc.) a través de operaciones booleanas. Las instrucciones lógicas operan bit a bit sobre los vectores de bits de sus operandos, las más frecuentes en los repertorios son las siguientes:

- NOT (complementación)
- AND (conjunción lógica)
- OR (disyunción lógica)
- XOR (disyunción exclusiva)
- Equivalencia

Con la AND podemos realizar funciones de máscara cuando queramos seleccionar un conjunto determinado de bits, y con la XOR podemos complementar bit a bit un segmento de bits. En el siguiente ejemplo hemos creado una máscara sobre los 4 bits menos significativos de R1 haciendo la AND con R2, y hemos complementado los 6 bits centrales de R1 haciendo la XOR con R3:

<R1> = 1010 0101

<R2> = 0000 1111

<R3> = 0111 1110

<R1> AND <R2> = 0000 0101

<R1> XOR <R3> = 1101 1011

Las instrucciones de manipulación de bits permiten poner a 0 ó a 1 determinados bits del operando, en especial algunos bits del *registro de estado*.

2.4. Instrucciones aritméticas

Casi todos los repertorios disponen de las operaciones aritméticas básicas de suma resta multiplicación y división sobre enteros con signo (coma fija). Con frecuencia disponen también de operaciones sobre reales (coma flotante) y decimales (BCD).

- Suma
- Resta
- Multiplicación
- División
- Cambio de signo
- Valor absoluto
- Incremento
- Decremento
- Comparación

2.5. Instrucciones de transformación de datos

- Traducción: traducen valores de una zona de memoria utilizando una tabla de correspondencia.
- Conversión: operan sobre el formato de los datos para cambiar la representación.

Por ejemplo cambio de BCD a binario o cambio de formato EBCDIC a formato ASCII como realiza la instrucción *Translate* (TR) del S/370.

2.6. Instrucciones de entrada/salida

Estas instrucciones coinciden con las de referencia a memoria de carga y almacenamiento (LOAD y STORE) para aquellos procesadores que disponen de un único espacio de direcciones compartido para Memoria y E/S.

Los procesadores con espacio de E/S independiente disponen de instrucciones específicas tales como:

- Inicio de la operación: START I/O
- Prueba del estado: TEST I/O
- Bifurcación sobre el estado: BNR (*Branch Not Ready*)
- Entrada: IN que carga en el procesador un dato de un dispositivo periférico
- Salida: OUT que lo envía a un dispositivo periférico.

2.7. Instrucciones de manipulación de direcciones

Estas instrucciones calculan la dirección efectiva de un operando para manipularla como un dato cargándola en un registro o una pila.

Como ejemplos tenemos las instrucciones LEA y PEA del 68000:

LEA.L opf, An: *dirección fuente --> An*

Lleva la dirección efectiva del operando fuente *opf* al registro de direcciones *An*

PEA.L opf: dirección fuente --> Pila

lleva la dirección efectiva del operando fuente *opf* a la Pila de la máquina a través del registro SP con predecremento: dirección --> -(SP)

2.8. Instrucciones que operan sobre datos del Pentium II

	Movimiento o transferencia de datos
MOV	Transferencia entre registros, o entre registro y memoria
PUSH	Inserción del operando en la pila
PUSHA	Inserción de todos los registros en la pila
MOVSX	Transfiere un byte a una palabra, o una palabra a una doble palabra con extensión del signo
MOVS	Mueven un byte, palabra o doble palabra de una cadena indexada
LODS	Carga un byte, palabra o doble palabra de una cadena

	Desplazamiento y rotación
SHL	Desplazamiento lógico a la izquierda
SHR	Desplazamiento lógico a la derecha
SAL	Desplazamiento aritmético a la izquierda
SAR	Desplazamiento aritmético a la derecha
ROL	Rotación izquierda
ROR	Rotación derecha

	Lógicas y manipulación de bits

AND	Conjunción lógica de los operandos
BTS	BitTes&Set utilizada para primitivas de sincronismo en multiproceso
BSF	Identifica el primer bit a 1 de una palabra o doble palabra y almacena su número de posición en un registro
SETcc	Pone un byte a cero o a uno dependiendo de algún bit del registro de estado

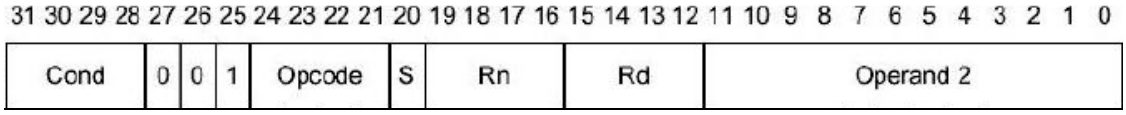
	Transformación de datos
XLAT	Traduce con la ayuda de una tabla

	Entrada/Salida
IN	Entrada de dato desde un dispositivo de E/S
OUT	Salida de dato hacia un dispositivo de E/S

	Manipulación de direcciones
LEA	Carga el desplazamiento del operando fuente (no su valor) en el destino

2.9. Instrucciones que operan sobre datos en el ARM

Formato



- Opcode: Función aritmético lógica
- Rn: Registro fuente.
- Rd: Registro destino.
- Cond: Bits de condición.
- El bit 25 distingue entre registro o un inmediato en los bits del operando2
-

Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

Operaciones aritméticas		
ADD	R0, R1, R2	R0 := R1 + R2
ADC	R0, R1, R2	R0 := R1 + R2 + C
SUM	R0, R1, R2	R0 := R1 - R2
SBC	R0, R1, R2	R0 := R1 - R2 + C - 1
RSB	R0, R1, R2	R0 := R2 – R1
RSC	R0, R1, R2	R0 := R2 – R1 + C - 1

Movimiento entre registros		
MOV	R0, R2	R0 := R2
MVN	R0, R2	R0 := not R2

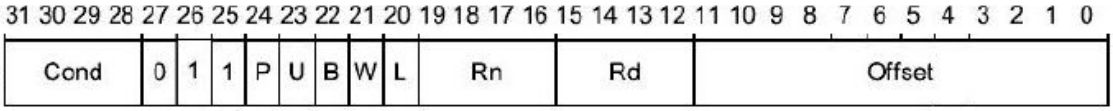
Operaciones lógicas		
AND	R0, R1, R2	R0 := R1 and R2
ORR	R0, R1, R2	R0 := R1 or R2
EOR	R0, R1, R2	R0 := R1 xor R2
BIC	R0, R1, R2	R0 := R1 and not R

Operaciones de comparación		
CMP	R1, R2	set cc con R1- R2
CMN	R1, R2	set cc con R1+ R2
TST	R1, R2	set cc con R1 and R2

TEQ R1, R2	set cc con R1 xor R2

2.10.Instrucciones de carga y almacenamiento

Formato



– STR r2, [r3] Almacena el contenido de r2 en la dirección de memoria que contiene r3.

– LDR r2, [r3] Carga en r2 el contenido de la dirección de memoria que contiene r3.

Desplazamiento(Offset)

LDR r1, [r2, #4] Carga en r1 el contenido de la dirección de memoria que contiene r2 + 4.

Post- indexado

LDR r1,[r2], #4 Carga en r1 el contenido de la dirección de memoria en r2 e incrementa r2 en 4.

Pre-indexado

LDR r1, [r2, #4]! Carga en r1 el contenido de la dirección de memoria en r2 + 4 e incrementa r2 en 4

Carga y almacenamiento de un único registro	
LDR R0, [R1]	R0 := mem_{32}[R1]
LDR R0, [R1, #4]	R0 := mem_{32}[R1+4]
LDR R0, [R1, #4]!	R0 := mem_{32}[R1+4]; R1 := R1 + 4 (pre-indexado)
LDR R0, [R1], #4	R0 := mem_{32}[R1]; R1 := R1 + 4 (post-indexado)
LDRB R0,[R1]	R0 := mem_{8}[R1]
STR R0, [R1]	mem_{32}[R1] := R0

Carga y almacenamiento de múltiples registros	
LDMIA R1, {R0, R2, R4}	R0 := mem_{32}[R1] R2 := mem_{32}[R1+4] R4 := mem_{32}[R1+8]
LDMIA R1!, {R2-R9}	R2 := mem_{32}[R1] R3 := mem_{32}[R1+4] ……. R9 := mem_{32}[R1+28] R1 := R1 + 32

STMIA R1, {R0, R2, R4}	mem_{32}[R1] :=R0 mem_{32}[R1+4] := R1 mem_{32}[R1+8] := R4

3. Instrucciones paralelas SIMD (*Single Instruction Multiple Data*)

Se trata de instrucciones muy optmizadas para realizar operaciones multimedia. Fueron introducidas en el 96 en la microarquitectura P6 de la familia IA-32 como tecnología MMX del Pentium. Estas instrucciones operan sobre los datos siguiendo un modo SIMD(*Single Instruction Multiple Data*), es decir, realizando la misma operación (*single instruction*) en paralelo sobre múltiples secuencias de datos, consiguiendo una mejora de velocidad sobre la alternativa secuencial del orden de 2 a 8 veces. Las ventajas de estas extensiones SIMD se habían probado con éxito en los repertorios de otros procesadores, por ejemplo, en el SPARC de Sun Microsystems con el nombre de VIS (Visual Instruction Set) y PA-RISC de Hewlett-Packard con el nombre de conjunto de instrucciones MAX-2.

Los programas multimedia operan sobre señales muestreadas de vídeo o audio representadas por grandes vectores o matrices de datos escalares relativamente pequeños (8 a 16 bits) o incluso 32 bits para señales 3D

3.1. Tipos de datos

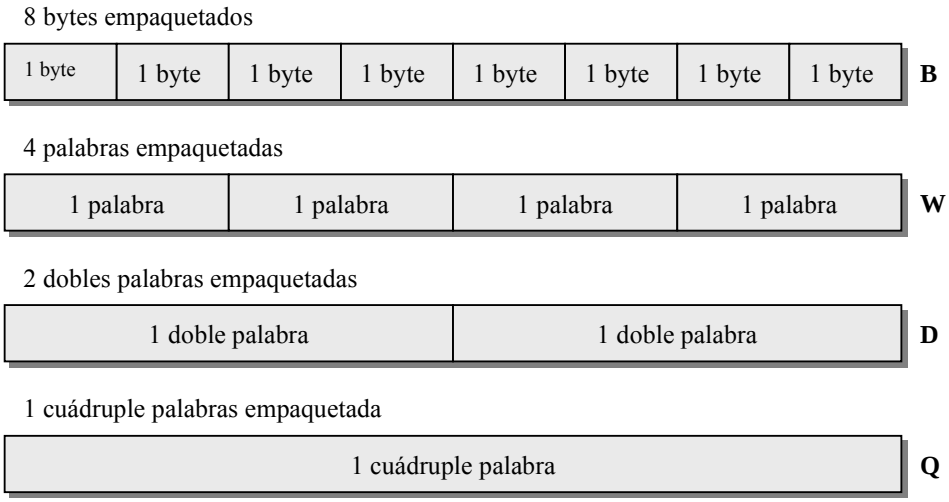
Los diferentes medios de una aplicación multimedia suelen utilizar el siguiente número de bits para codificar sus informaciones básicas:

Gráficos y vídeos: utilizan 8 bits para cada punto (escala de grises) o componente de color.

Audio: utiliza 16 bits para cada valor muestreado de la señal.

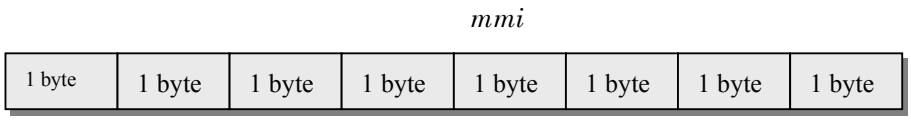
Gráficos 3D: pueden utilizar hasta 32 bits por punto.

Para poder realizar el procesamiento paralelo de estas informaciones básicas, MMX define los tres tipos de datos siguientes:



Registros MMX

Existen 8 registros de propósito general MMX que se designan como *mmi (i = 0,...7)*. Estos registros no tienen una realidad independiente dentro de la organización del procesador, se soportan sobre los 64 bits menos significativos de los registros de punto flotante de 80 bits. El campo de exponente del correspondiente registro de punto flotante (bits 64-78) y el bit de signo (bit 79) se pone a 1's, convirtiendo el contenido en un valor NaN desde el punto de vista de los valores en punto flotante. De esta forma se reduce la posible confusión, asegurando que los valores de datos MMX no se tomen como valores en punto flotante. Las instrucciones MMX acceden sólo a los 64 bits menos significativos de los registros. Este uso dual de los registros de punto flotante no significa que se pueda mezclar en las aplicaciones ambos usos de forma indiscriminada. Dentro de una aplicación el programador deberá asegurar que el código MMX y el código en punto flotante estén encapsulados en rutinas separadas.



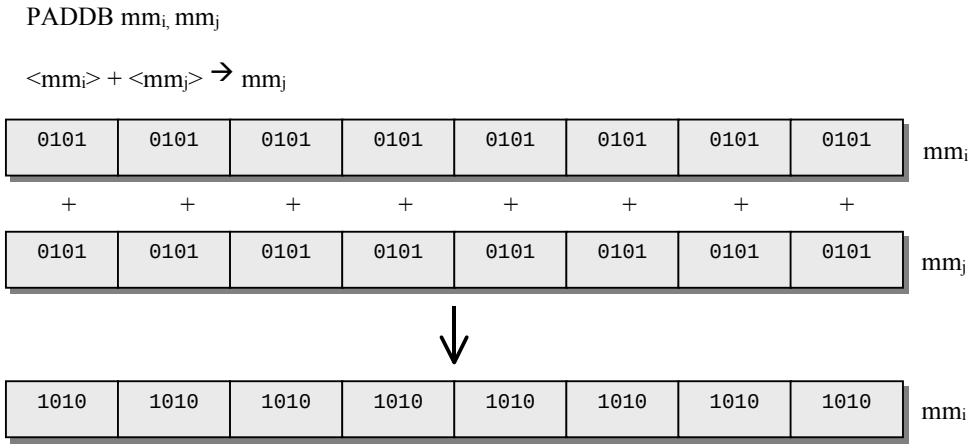
3.2. Repertorio de instrucciones MMX

Formato CSO [<tipo de datos>] <registros MMX>

CSO = Código Simbólico de Operación

Tipo de datos = B, W, D ó Q

Ejemplo: instrucción de suma paralela con truncamiento de dos registros MMX con 8 bytes empaquetados



3.2.1.1. Instrucciones Aritméticas

Este grupo lo forman 9 instrucciones, 3 de suma, 3 de resta y 3 de multiplicación. En ellas se distingue entre la aritmética con saturación y con truncamiento. En la aritmética binaria con truncamiento el resultado de una operación con desbordamiento queda truncado por la pérdida del bit de desbordamiento. Por el contrario, en la aritmética con saturación, cuando la suma produce un desbordamiento, o la resta un desbordamiento negativo, el resultado se fija respectivamente al mayor o menor valor representable.

Ejemplo:

```
    1111 0000 0000 0000
+   0011 0000 0000 0000
  ---------------------
```

rebose --> **1** `0010 0000 0000 0000` resultado en aritmética con truncamiento

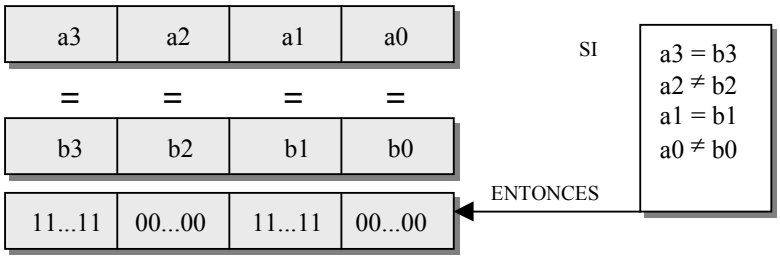
`1111 1111 1111 1111` resultado en aritmética con saturación

La aritmética con saturación se utiliza, por ejemplo, cuando se suman dos números que representan intensidad de imagen (nivel de gris, por ejemplo, o de un color base) y no se quiere que el resultado sea menos negro (menos intenso en el caso de color base) que la suma de las imágenes originales. Las instrucciones de este grupo son las siguientes:

Instrucciones Aritméticas	Semántica
PADD [B, P, D] mm_i, mm_j	Suma paralela con truncamiento de 8 B, 4 W ó 2 D
PADDS [B, W] mm_i, mm_j	Suma con saturación
PADDUS [B, W] mm_i, mm_j	Suma sin signo con saturación
PSUB [B,W, D] mm_i, mm_j	Resta con truncamiento
PSUBS [B,W] mm_i, mm_j	Resta con saturación
PSUDUS [B, W] mm_i, mm_j	Resta sin signo con saturación
PMULHW	Multiplicación paralela con signo de 4 palabras de 16 bits, con elección de los 16 bits más significativos del resultado de 32 bits
PMULLW mm_i, mm_j	Multiplicación paralela con signo de 4 palabras de 16 bits, con elección de los 16 bits menos significativos del resultado de 32 bits
PMADDWD mm_i, mm_j	Multiplicación paralela con signo de 4 palabras de 16 bits, y suma de pares adyacentes del resultado de 32 bits

3.2.1.2. Instrucciones de Comparación

La tecnología MMX dispone de instrucciones paralelas de comparación que generan una máscara de bits como resultado. Estas instrucciones posibilitan la realización de cálculos dependientes de los resultados anteriores eliminando la necesidad de instrucciones de bifurcación condicional. La máscara resultado de una comparación tiene todo 1's en los elementos correspondientes a una comparación con resultado *true*, y todo 0's cuando el resultado de la comparación es *false*:



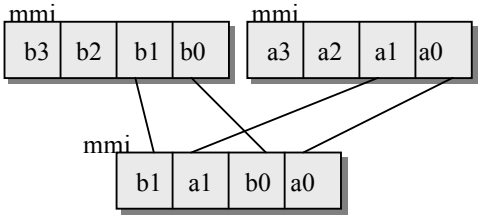
Las instrucciones de este grupo son las siguientes:

Instrucciones de Comparación	Semántica
PCMPEQ [B, W, D] mm_i, mm_j	Comparación paralela de igualdad, el resultado es una máscara de 1´s si True y de 0´s si False
PCMPGT [B, W, D] mm_i, mm_j	Comparación paralela de > (magnitud), el resultado es una máscara de 1´s si True y de 0´s si False

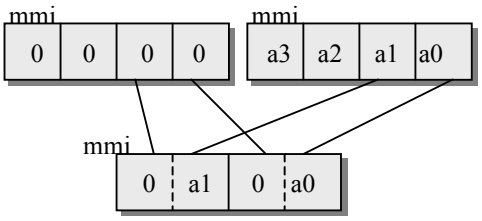
3.2.1.3. Instrucciones de Conversión

La tecnología MMX utiliza dos conjuntos de instrucciones para realizar conversiones entre los tipos de datos MMX (*PACK y UNPACK*). Las instrucciones *UNPACK* operan sobre un tipo de datos MMX para producir otro de mayor capacidad (por ejemplo, sobre uno de 16 bits para producir

otro de 32 bits). En realidad, las instrucciones *UNPACK* toman dos operandos y los entrelazan, como ocurre en el siguiente ejemplo para datos con componentes de 16 bits:



Si sólo se quiere desempaquetar elementos de 16 bits en elementos de 32 bits se ejecuta la instrucción *UNPACK* con un operando de todos 0's. El resultado es 2 palabras de 32 bits con 0's en las posiciones más significativas:



Las instrucciones de este grupo son las siguientes:

Instrucciones de Conversión	Semántica
PACKUSWB mm_i, mm_j	Empaqueta palabras en bytes con saturación sin signo
PACKSS [WB, DW] mm_i, mm_j	Empaqueta palabras en bytes con saturación con signo, o palabras dobles en palabras
PUNPCKH [BW, WD, DQ] mm_i, mm_j	Desempaqueta en paralelo los B, W o D más significativos del registro MMX. Mezcla entrelazada
PUNPCKL [BW, WD, DQ] mm_i, mm_j	Desempaqueta en paralelo los B, W o D menos significativos del registro MMX. Mezcla entrelazada

3.2.1.4. Instrucciones Lógicas

El repertorio MMX dispone de 4 operaciones lógicas que realizan otras tantas operaciones bit a bit de este tipo sobre los 64 bits de los operandos. Combinadas con las máscaras resultantes de las instrucciones de comparación permiten la realización de cálculos alternativos en función de resultados anteriores. Las instrucciones de este grupo son las siguientes:

Instrucciones Lógicas	Semántica
PAND mm_i, mm_j	AND lógico bit a bit (64 bits)
PANDN mm_i, mm_j	AND NOT lógico bit a bit (64 bits)
POR mm_i, mm_j	OR lógico bit a bit (64 bits)
PXOR mm_i, mm_j	XOR lógico bit a bit (64 bits)

3.2.1.5. Instrucciones de Desplazamiento

Este grupo lo constituyen operaciones de desplazamiento y rotación lógicos y aritméticos realizados en paralelo sobre cada uno de los elementos de datos de uno de sus operandos. El otro indica el número de bits de desplazamiento o rotación. Las instrucciones de este grupo son las siguientes:

Instrucciones de Desplazamiento	Semántica
PSLL [W, D, Q] mm_i	Desplazamiento lógico izquierda paralelo de W, D ó Q tantas veces como se indique en el registro MMX o valor inmediato
PSRL [W, D, Q] mm_i	Desplazamiento lógico derecha paralelo de W, D ó Q tantas veces como se indique en el registro MMX o valor inmediato
PSRA [W, D,] mm_i	Desplazamiento aritmético derecha paralelo de W ó D tantas veces como se indique en el registro MMX o valor inmediato

3.2.1.6. Instrucciones de Transferencia

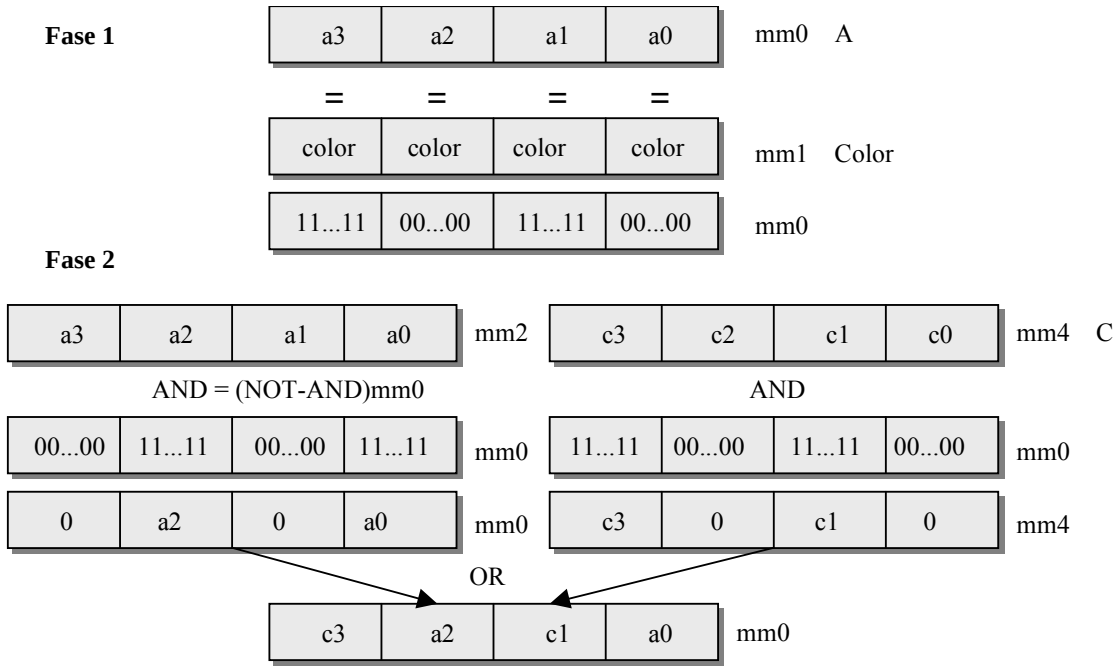
La única instrucción de este grupo permite la transferencia de información entre registros MMX, la carga de registros MMX desde memoria, y el almacenamiento de los contenidos de los registros MMX en memoria:

Instrucción de Transferencia	Semántica
MOV [D, Q] $mm_{i,\ memoria}$	Transfiere una D ó Q a/desde el registro MMX

3.3. Ejemplos de codificación MMX

3.3.1. Ejemplo 1. Codificación MMX de un bucle iterativo

```
for i = 0 to 3
if a[i] = color then
Salida[i] = c[i] else
Salida[i] = a[i]
```



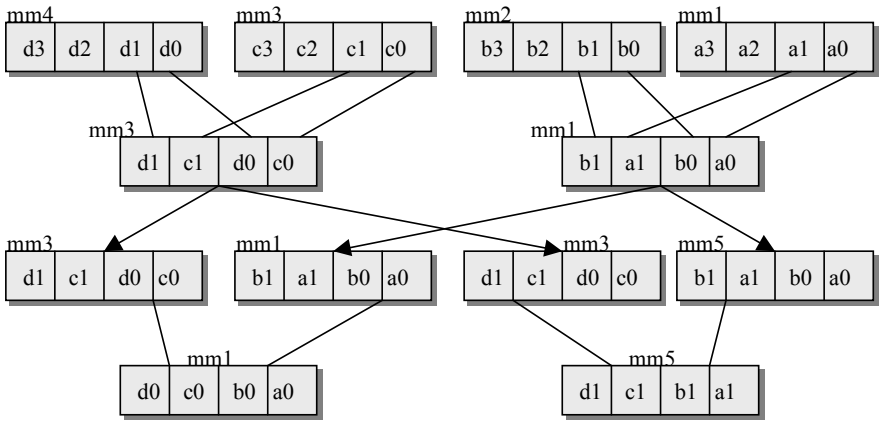
El programa MMX que implementa el proceso iterativo es el siguiente:

```
MOVQ      mm0, A      ;carga el array A
MOVQ      mm2, mm0    ;hace copia del array A
MOVQ      mm4, C      ;carga el array C
MOVQ      mm1, Color  ; carga Color
PCMPEQW   mm0, mm1    ;compara A con Color
PAND      mm4, mm0    ;selecciona elementos de C
PANDN     mm0, mm2    ;selecciona elementos de A
POR       mm0, mm4    ;genera la salida
```

3.3.2. Ejemplo 2. Transposición de matrices

$$\begin{bmatrix} a_3a_2a_1a_0 \\ b_3b_2b_1b_0 \\ c_3c_2c_1c_0 \\ d_3d_2d_1d_0 \end{bmatrix}^T = \begin{bmatrix} d_3c_3b_3a_3 \\ d_2c_2b_2a_2 \\ d_1c_1b_1a_1 \\ d_0c_0b_0a_0 \end{bmatrix}$$

El proceso se realiza en dos pasos y supondremos que los elementos de la matriz son palabras de 16 bits. En la fase 1 se utiliza la instrucción *UNPACK* para entrelazar los elementos de filas adyacentes; y en la fase 2 se desempaquetan los resultados de la primera fase utilizando esta vez instrucciones *UNPACK* de doble palabra (32 bits) para producir la salida deseada. El proceso se resume en la siguiente figura:



El programa MMX que implementa el proceso de transposición es el siguiente:

```
MOVQ        mm1, fila1    ;carga la primera fila de la matriz
MOVQ        mm2, fila2    ;carga la segunda fila de la matriz
MOVQ        mm3, fila3    ;carga la tercera fila de la matriz
MOVQ        mm4, fila4    ;carga la cuarta fila de la matriz
PUNPCKLWD   mm1, mm2      ;desempaqueta palabras orden inf. de filas 1 & 2,  mm1=[b1,a1,b0,a0]
PUNPCKLWD   mm3, mm4      ;desempaqueta palabras orden inf. de filas 3 & 4,  mm3=[d1,c1,d0,c0]
MOVQ        mm5, mm1      ;copia mm1 en mm5
PUNPCKLDQ   mm1, mm3      ;desempaqueta dobles palabras orden inferior ->  mm1=[d0,c0,b0,a0]
PUNPCKHDQ   mm5, mm3      ;desempaqueta dobles palabras orden superior ->  mm1=[d1,c1,b1,a1]
```

las otras 2 filas [d3,c3,b3,a3] y [d4,c4,b4,a4] se generan de forma análoga.

3.3.3. Ejemplo 3:

Función de desvanecimiento gradual de una imagen B y aparición simultanea de otra imagen A

Las dos imágenes A y B se combinan en un proceso que efectúa sobre cada par de *pixels* homólogos la media ponderada por un factor de desvanecimiento *fade* que irá variando de 0 a 1, y su complementario (*1-fade*), que lo hará de 1 a 0

$$Pixel\_resultante = Pixel\_A * fade + Pixel\_B*(1 - fade)$$

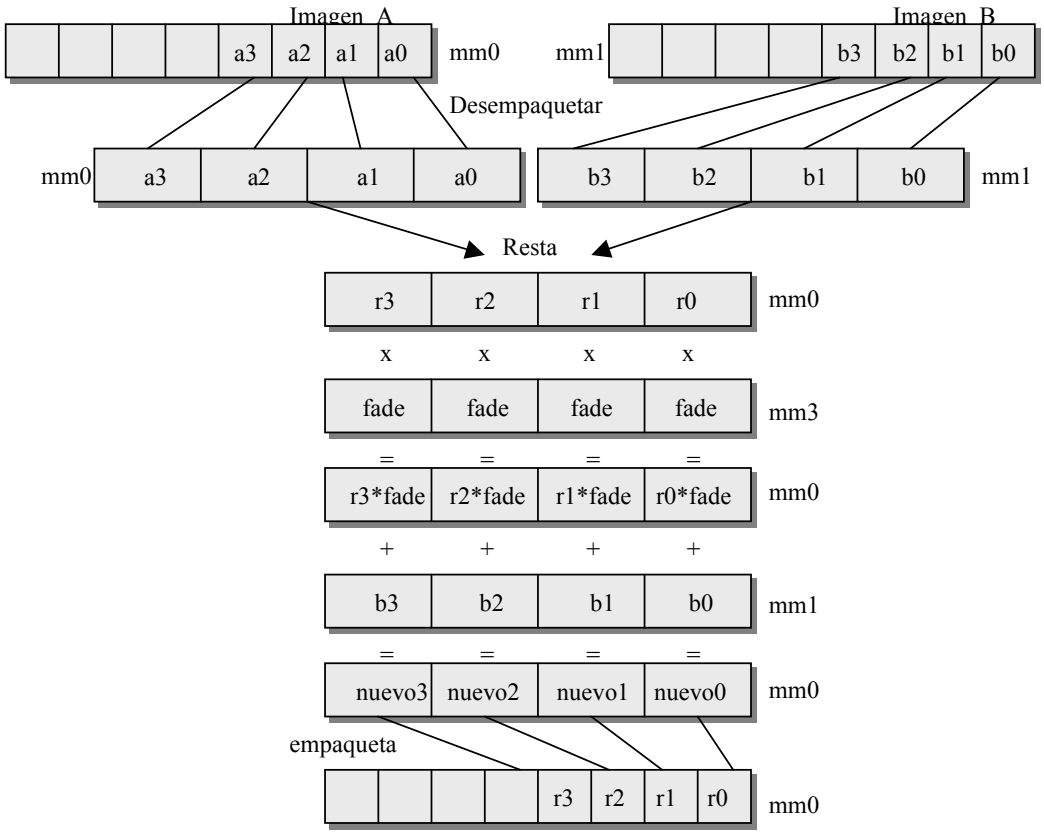
El resultado es una transformación gradual de la imagen B en la imagen A, a medida que *fade* pasa de *0* a *1*. Cada pixel ocupa 1 byte y hay que desempaquetarlo previamente a un valor de 16 bits para adaptarlo al tamaño de las multiplicaciones MMX de 16 bits.

Si una imagen tiene una resolución de 640 X 480 y en el cambio de imagen se emplean 255 valores posibles para *fade* (*fade* codificado con 1 byte), será necesario ejecutar el proceso 640*480*255 = 7.833.600 veces. Eso para el caso de imágenes en blanco y negro, si fuesen en color necesitaríamos esa cantidad multiplicada por 3 (tres componentes básicas del color), es decir 23.500.800. Para secuenciar las operaciones pongamos la expresión anterior de la intensidad del pixel resultante de la siguiente forma:

$$Pixel\_resultante = Pixel\_A * fade + Pixel\_B - Pixel\_B*fade =$$

$$(Pixel\_A - Pixel\_B) * fade + Pixel\_B$$

Luego hay que restar al *Pixel_A* el valor del *Pixel_B*, multiplicar el resultado por *fade* y sumar finalmente el valor del *Pixel_B*. A estas operaciones hay que añadir las previas de movimiento de datos y desempaquetamiento. En la siguiente figura aparece el esquema del proceso:



El programa MMX que implementa este proceso es el siguiente:

PXOR	mm7, mm7	;pone a cero mm7
MOVQ	mm3, valor_fade	;carga en mm3 el valor de fade replicado 4 veces
MOVD	mm0, imagenA	;carga en mm0 la intensidad de 4 pixels de la imagen A
MOVD	mm1, imagenB	;carga en mm1 la intensidad de 4 pixels de la imagen B
PUNPCKLBW	mm0, mm7	;desempaqueta a 16 bits los 4 pixels de mm0
PUNPCKLBW	mm1, mm7	;desempaqueta a 16 bits los 4 pixels de mm1
PSUBW	mm0, mm1	;resta la intensidad de la imagen B de la de A
PMULHW	mm0, mm3	;multiplica el resultado anterior por el valor de fade
PADDDW	mm0, mm1	;suma el resultado anterior a la imagen B
PACKUSWB	mm0, mm7	;empaqueta en bytes los 4 resultados de 16 bits

4. Instrucciones de control del flujo de ejecución.

El orden de procesamiento de las instrucciones viene determinado por el avance del contador de programa, que lo hace de forma secuencial con todas las instrucciones salvo con las de los siguientes grupos:

- Bifurcación condicional (e incondicional)
- Bifurcación a subrutinas

Este grupo de instrucciones juega un triple papel en las posibilidades expresivas y computacionales del repertorio:

- Toma de decisiones en función de resultados previamente calculados. Las instrucciones de bifurcación condicional cumplen este papel en el repertorio.
- Reutilización de parte del código del programa. Los procesos iterativos son el caso más elemental de reutilización de código, que se construyen con instrucciones de bifurcación condicional. El segundo caso es el de las subrutinas, con ellas se construyen los bloques (funciones y procedimientos) de los lenguajes de alto nivel.
- Descomposición funcional del programa. Se trata del segundo papel que cumplen las subrutinas dentro de un programa: facilitar la modularidad del mismo, y con ello su depuración y mantenibilidad.

4.1. Instrucciones de bifurcación condicional

Los tres componentes básicos de las instrucciones de salto condicional son: el código de operación, la condición de salto y la dirección destino.

Semántica:

IF Condición = True THEN

CP <-- Dirección

ELSE

CP <-- <CP> + 1

4.1.1. Gestión de la Condición

- Generación

  a) Implícita: la condición se genera como efecto lateral de la ejecución de una instrucción de manipulación de datos (ADD, SUB, LOAD, etc.)

  b) Explícita: existen instrucciones específicas de comparación o test que sólo afectan a las condiciones (CMP, TST, etc.)

- Selección

  a) Simple: afecta a un sólo bit del registro de condiciones (Z, N, C, etc.)

  b) Compuesta: afecta a una combinación lógica de condiciones simples (C+Z, etc.)

simples		compuestas	
Nemotécnico	Condición	nemotécnico	Condición
CC (carry clear)	¬C	HI (high)	¬C • ¬Z
CS (carry set)	C	LS (low or same)	C + Z
NE (not equal)	¬Z	HS (high or same)	¬C
EQ (equal)	Z	LO (low)	C
VC (overflow clear)	¬V	GE (greater or equal)	N xor ¬V
VS (overflow set)	V	LT (less than)	N xor V
PL (plus)	¬N	GT (greater than)	(N xor ¬V)and¬Z
MI (minus)	N	LE (less or equal)	(N xor V)orZ

- Uso

  a) Con almacenamiento de la condición en un registro de estado o condición: una 1ª

instrucción de gestión de datos o comparación genera la condición que se almacena en un registro de estado de la máquina, y una 2ª instrucción selecciona la condición y bifurca en caso que sea cierta (True)

b) Con almacenamiento de la condición en un registro general: una 1ª instrucción de gestión de datos o comparación genera la condición que se almacena en un registro general de la máquina que debe especificarse en la misma, y una 2ª instrucción selecciona la condición y bifurca en caso que sea cierta (True)

c) Sin almacenamiento de la condición: una única instrucción genera, selecciona la condición y bifurca en caso que sea cierta (BRE R1, R2, DIR : salta a DIR si <R1> = <R2>)

4.1.2. Especificación de la dirección

- Direccionamiento directo (Jump)
- Direccionamiento relativo (Branch)
- Direccionamiento implícito (Skip: salta una instrucción si se cumple la condición)
- Instrucciones de procesamiento de datos condicionales (ARM)

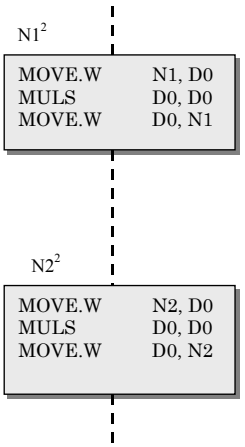
4.2. Subrutinas

Supongamos que en diferentes puntos de un programa se realiza la operación de elevar al cuadrado un número. El código simbólico (MC68000) de la operación sería el siguiente:

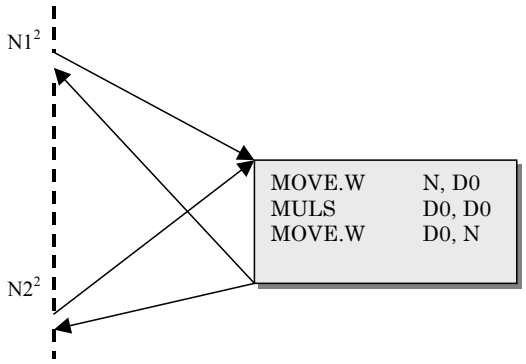
```
MOVE.W    N, D0
MULS      D0, D0
MOVE,W    D0, N
```

El programador tiene dos opciones para repetir el código en los puntos donde aparezca la operación: las macros y las subrutinas.

a) Una MACRO o *subrutina abierta* es una llamada que se realiza en aquellos puntos donde aparece la operación y que el ensamblador sustituye por el código correspondiente en un proceso denominado de expansión de macros.



b) Una SUBRUTINA o *subrutina cerrada* por el contrario no repite el código en cada llamada, bifurca a un único código que una vez ejecutado devuelve control a la instrucción siguiente a la que realizó la bifurcación:



Cuatro problemas hay que resolver en el mecanismo de una subrutina (cerrada):

- Controlar la dirección de vuelta a la instrucción siguiente a la que realizó la llamada
- Organizar el paso de parámetros
- Ubicar las variables locales de la subrutina
- Preservar el contexto del programa

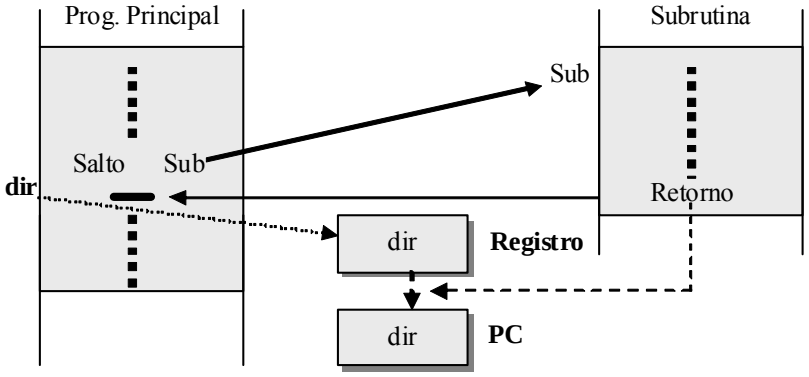
Analizaremos las diferentes alternativas de resolución de estos problemas resaltando las ventajas e inconvenientes de cada una.

4.2.1. Control de la dirección de vuelta

Para hacer que la dirección de vuelta de una subrutina sea siempre la siguiente a la de la instrucción que realiza la llamada se pueden utilizar las siguientes alternativas:

1) Utilización de un registro

Se trata de una alternativa fácil de implementar y rápida en su ejecución. Antes de realizar el salto se almacena la dirección de la siguiente instrucción (**dir** en la figura) en el Registro. La subrutina finaliza con una instrucción de Retorno que lleva el contenido de Registro al contador de programa (PC)



Esta alternativa, sin embargo, impide el anidamiento, es decir, la posibilidad de que se pueda realizar la llamada a una subrutina desde el interior de otra subrutina Tampoco posibilita la recursión, es decir, una nueva llamada a la propia subrutina desde el interior de una llamada previa.

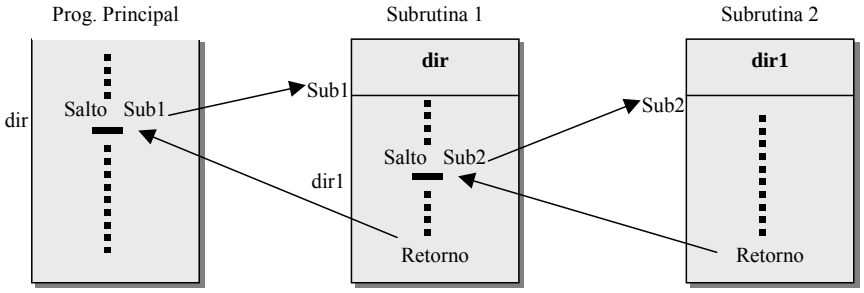
Aunque en el 68000 no es necesario utilizar esta alternativa, su codificación podría ser la siguiente:

```
Cuadrado    MOVE.W    N, D0
            MULS      D0, D0
            MOVE.W    D0, N
```

```
                JMP         (A0)
                .....
                MOVE.W      N1, N
                LEA.L       siguiente1,     A0
                JMP         cuadrado
siguiente1      .....

                MOVE.W      N2, N
                LEA.L       siguiente2,     A0
                JMP         cuadrado
siguiente2      .
```

2) Utilización de una posición de memoria de la subrutina

Esta alternativa exige que antes de bifurcar a la subrutina se ponga la dirección de vuelta en una posición de memoria perteneciente a la subrutina (por ejemplo, la anterior a la de la primera instrucción). El retorno se realiza con un salto incondicional e indirecto sobre esta posición.



Esta alternativa permite un número ilimitado de llamadas anidadas, pero no la recursión.

En el 68000 codificaríamos esta alternativa llevando el contenido de la posición de memoria a un registro de direcciones y realizando un salto indirecto sobre él para implementar la vuelta:

```
vuelta          ---------------------------

cuadrado        MOVE.W N, D0
                MULS            D0, D0
                MOVE.W          D0, N
                LEA.L           vuelta, A0
                JMP             (A0)
                    .......
                LEA.L           siguiente1, A1
                MOVE.L A1, vuelta
                JMP             cuadrado
siguiente1        .......
                LEA.L           siguiente2, A1
                MOVE.L A1, vuelta
                JMP             cuadrado
siguiente2      ......
```

3) Utilización de una pila (*stack*)

Inmediatamente antes de realizar la bifurcación a la subrutina se coloca la dirección de retorno en una pila. Lo último que realiza la subrutina es recuperar esta dirección de retorno y llevarla al contador de programa. Esta alternativa permite el anidamiento y la recursión (sólo por

lo que respecta al ordenamiento de llamadas y vueltas) porque resuelve el orden de los retornos con el orden de inserción y extracción que impone la Pila. Se vuelve a la instrucción siguiente a la última llamada porque de la pila siempre se extrae el último retorno almacenado.

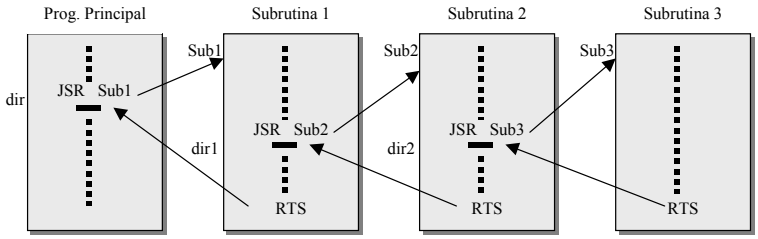
En el 68000 existen dos instrucciones que realizan las dos acciones descritas:

JSR D1

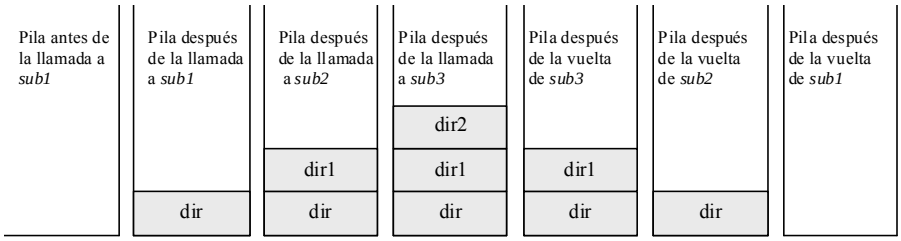
Bifurca a D1 almacenando previamente la dirección de la instrucción siguiente en una pila simulada en memoria a través de un registro puntero de pila (*stack pointer*)

RTS

Lleva el último retorno introducido en la pila al contador de programa.



La evolución de la pila en las secuencias de llamadas y vueltas sería la siguiente:



La estructura del código para el 68000 sería la siguiente:

```
cuadrado    MOVE.W      N, D0
             MULS       D0, D0
             MOVE.W     D0, N
             RTS
             ......
            JSR         cuadrado
siguiente1   .....
            JSR         cuadrado
siguiente2   .....
             ......
```

4.2.2. Paso de parámetros

1) Utilización de un registro

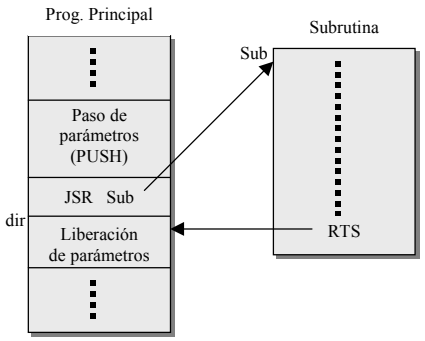
La utilización de registros para el paso de parámetros y devolución de resultados entre el programa principal y la subrutina es el método más rápido (pues no hay que acceder a memoria) pero resulta bastante limitado si el procesador no dispone de un buen número de registros. (Veremos más adelante que en algunos procesadores RISC existen mecanismos especiales para realizar el paso de parámetros a través de registros con solapamiento).

2) Utilización de memoria

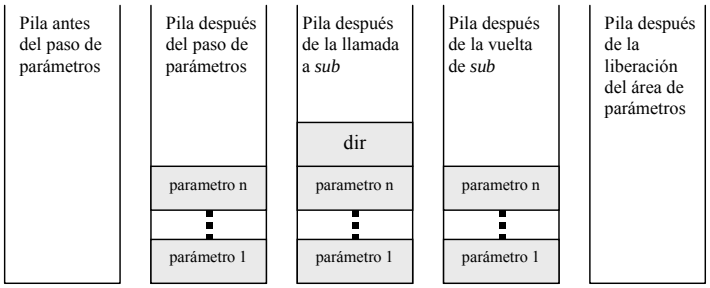
La utilización de memoria para el paso de parámetros aporta mayor capacidad pero lentifica el proceso por la necesidad de acceder a memoria. Tanto esta alternativa como la anterior impiden la recursión de las subrutinas

3) Utilización de la pila

Cuando se realiza la llamada no sólo se introduce en la pila la dirección de retorno, sino que previamente se introducen también los parámetros que deben transferirse. La subrutina accede a los parámetros actuales en la pila.

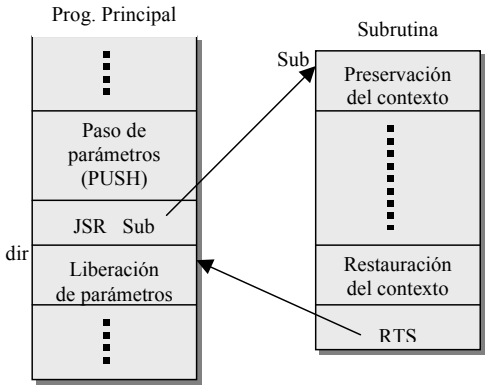


La evolución de la Pila en el proceso de llamada y vuelta de subrutina sería la siguiente:

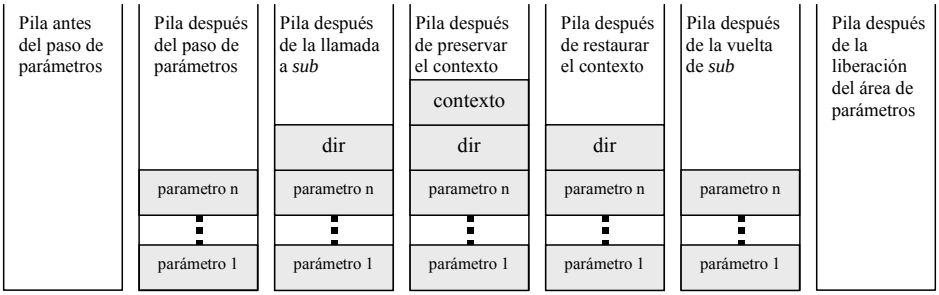


4.2.3. Preservación del contexto

Para que no se modifique el contexto de operación del programa principal, será necesario que la subrutina comience preservando el valor todos los registros que se modifican en su ejecución, y finalice restaurando el valor de dichos registros:

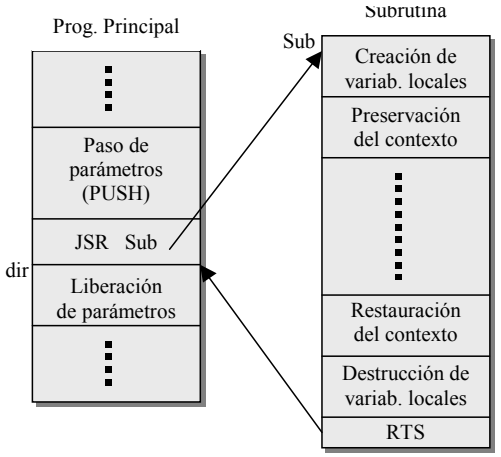


En este caso la evolución de la Pila sería la siguiente:

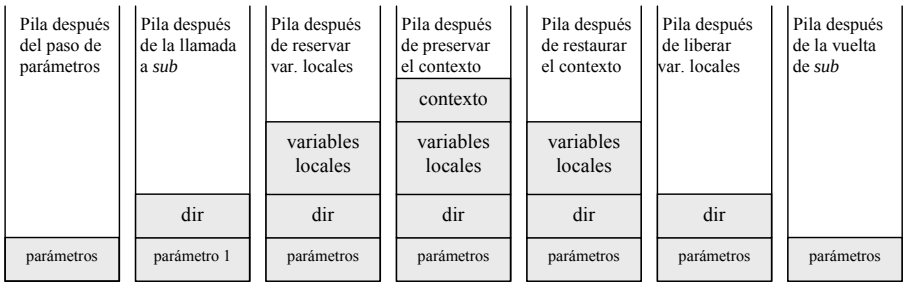


4.2.4. Variables locales de la subrutina

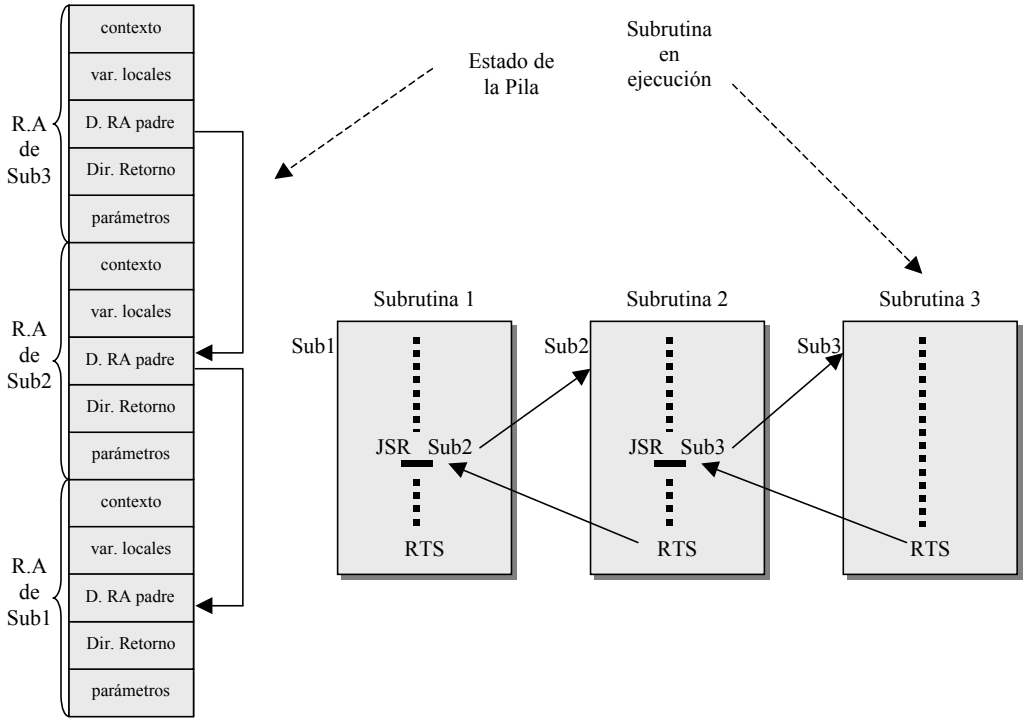
Para que una subrutina con variables locales sea re-entrante y por tanto recursiva es necesario crear un conjunto nuevo de variables locales cada vez que se produce una nueva llamada sin haber finalizado la anterior. Para ello se puede utilizar de nuevo la Pila, que se convierte así en el soporte de la gestión dinámica de la memoria en las llamadas a subrutinas. El conjunto de información que se introduce en la pila como consecuencia de una llamada a subrutina se conoce como Registro de Activación (RA)



La evolución de la Pila en el proceso de llamada, ejecución con variables locales y vuelta de subrutina sería la siguiente:



Finalmente, para que el RA de una subrutina pueda recuperarse cuando se vuelve de una llamada a otra subrutina, es necesario que antes de realizar la llamada se guarde en el nuevo RA (el de la subrutina llamada) la dirección del antiguo RA (el de la subrutina que realiza la llamada, o subrutina padre). De esta manera todos los RAs quedan enlazados a través de este nuevo campo:

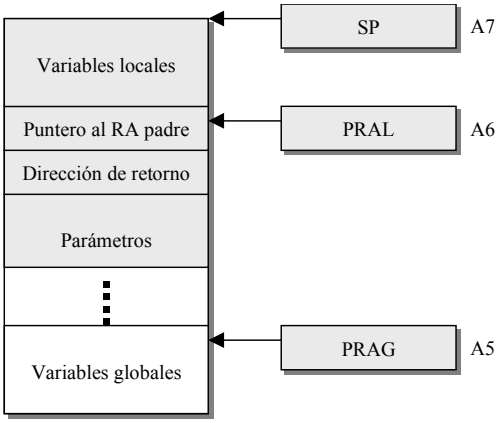


4.3. Ejemplo: MC 68000

En el 68000 los Registros de Activación (RA) se gestionan con las instrucciones LINK y UNLK, y contienen los cuatro campos siguientes:

- parámetros reales de la llamada a la subrutina,
- dirección de retorno de la subrutina,
- puntero del registro de activación del subprograma padre
- zona de variables locales.

El puntero al registro de activación actual se almacena en un registro de direcciones, que normalmente es el registro A6. El puntero de pila (SP) es el registro A7 y el puntero a las variables globales A5.



La instrucción LINK sirve para reservar la memoria correspondiente a un registro de activación y actualizar el puntero al mismo, es decir, para asignar y enlazar el registro de activación. La instrucción UNLIK deshace el enlace y devuelve la memoria.

La instrucción *LINK* tiene el siguiente formato:

LINK An, #desplazamiento

An se usará como puntero al registro de activación, y el desplazamiento es un número entero negativo, cuya magnitud es el número de bytes que ocupan las variables locales. Su semántica es la siguiente:

An --> -(SP); SP --> An; SP + desplazamiento --> SP

Es decir:

1. Guarda el registro en la pila.
2. Copia el contenido del puntero de pila en el registro.
3. Suma el número negativo al puntero de pila, decrementando el puntero de pila un número de bytes igual al que ocupan las variables locales.

Teniendo en cuenta que en la pila están ya los parámetros y la dirección de retorno de la subrutina, la ejecución de esta instrucción al principio de la subrutina tiene el siguiente efecto:

- Guarda en la pila el puntero al registro de activación actual (A6), que contiene el puntero al registro de activación de la subrutina padre.
- Copia el puntero de pila (SP) en el puntero al registro de activación actual (PRAL), que no varía durante la ejecución de la subrutina. Con esto se enlaza el registro de activación de la subrutina con el de la subrutina padre.
- Decrementa el puntero de pila reservando espacio para las variables locales de la subrutina.
-

La instrucción *UNLK* desasigna la zona de memoria local restituyendo el puntero de registro de activación de la subrutina padre. Tiene el siguiente formato:

UNLK An

An es el registro de direcciones que se ha usado como puntero de registro de activación. Su semántica es la siguiente:

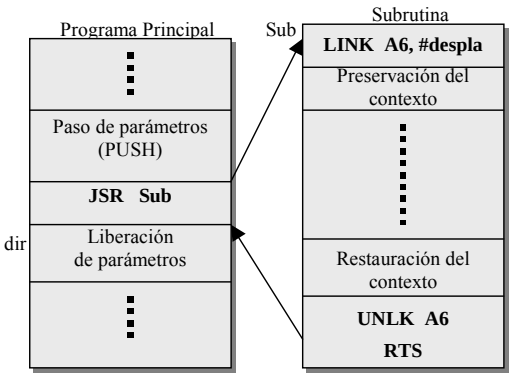
An --> SP; (SP)+ --> An

Es decir:

1. Copia el contenido del registro en el puntero de pila (SP).
2. Saca una palabra larga de la pila y la almacena en el registro.

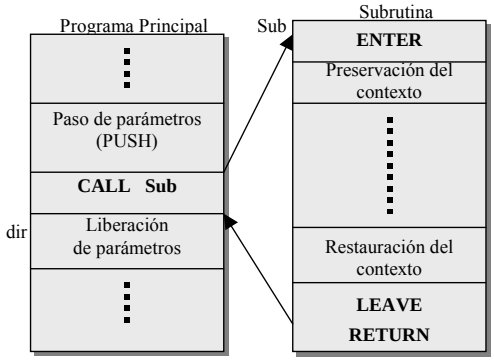
La ejecución de esta instrucción antes del retorno de subrutina tiene el siguiente efecto:

- Copia en el puntero de pila el contenido del puntero al registro de activación actual. Con esto se libera la zona de variables locales de la pila.
-
- Saca de la pila el puntero de registro de activación de la subrutina padre y lo coloca como puntero de registro de activación actual. El puntero de pila queda apuntando a la dirección de retorno.



4.4. Pentium

El procesador Pentium II tiene instrucciones equivalentes a las del 68000 para realizar la gestión de las subrutinas: CALL, ENTER, LEAVE y RETURN equivalentes a JSR, LINK, UNLK y RTS.



Sin embargo, en el 8088 y 8086 la subrutina se iniciaba con la siguiente secuencia de instrucciones:

```
PUSH  EBP
MOV     EBP, ESP
SUB     ESP, espacio_para_variables_locales
```

Donde EBP es el PRAL y ESP el Puntero de pila (SP).

Esta secuencia de instrucciones es más rápida que su equivalente ENTER que se introdujo con el 80286 (10 ciclos ENTER frente a 6 la secuencia anterior). La instrucción ENTER se añadió al repertorio para sustituir a todo el código anterior, aunque su velocidad de ejecución es menor que la de dicho código.

4.5. Instrucciones de bifurcación en el ARM

CO	Condición
B	Incondicional
BAL	Siempre
BEQ	Igual
BNE	No igual
BPL	Más
BMI	Negativo

BCC	Carry cero
BLO	Menor
BCS	Carry uno
BHS	Mayor o igual
BVC	Overflow cero
BVS	Overflow uno
BGT	Mayor que
BGE	Mayor o igual
BLT	Menor que
BLE	Menor o igual
BHI	Mayor
BLS	Menor o igual

1) Instrucción básica de salto incondicional

```
        B    etiqueta    ; Salto incondicional a etiqueta
        …….
        …….
Etiqueta …….
```

2) Instrucciones de salto condicional

```
        MOV    r0, #10     ;Inicializa el contador del bucle
bucle   …….
        …….
        …….
        SUB    r0, r0, #1    ;Decrementa el contador
        CMP    r0, #0        ; ¿es cero?
        BNE    bucle         ; Bifurca si r0 ≠ 0
```

3) Bit S permite cambiar el cc en CPSR

```
        MOV    r0, #10     ;Inicializa el contador del bucle
bucle   …….
        …….
        …….
        SUBS   r0, r0, #1    ;Decrementa el contador
        BNE    bucle         ; Bifurca si r0 ≠ 0
```

4) Instrucciones condicionadas

Código con instrucciones de salto condicional

```
        CMP   r0, #5
        BEQ   Bypass
        ADD   r1, r1, r0
        SUB   r1, r1, r2
Bypass…….
```

Código equivalente con instrucciones condicionadas

```
        CMP     r0, #5
        ADDNE   r1, r1, r0
        SUBNE   r1, r1, r2
Bypass…….
```

5) Instrucción de salto a subrutina

```
BL    sub
```

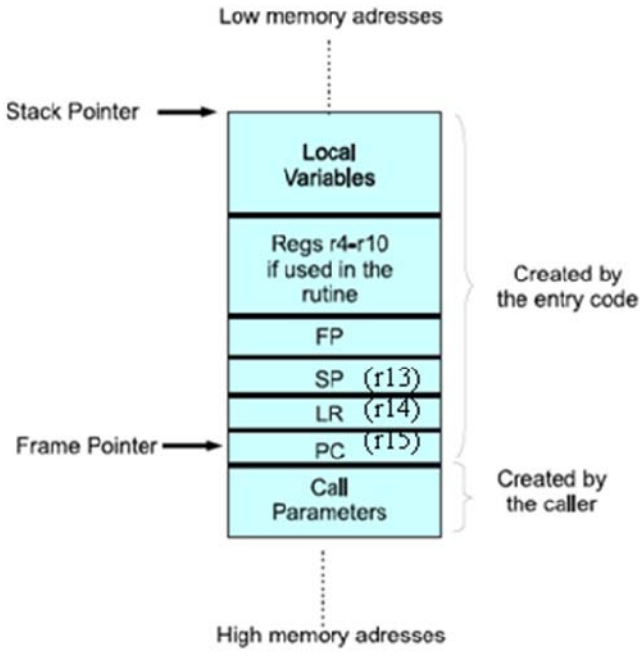
(Branch-and-Link): r14 := r15
r15 := sub (dirección de subrutina)

6) Instrucción de vuelta de subrutina

```
MOV  r15, r14
```

Ejemplo

```
        BL    SUB1
        …….
SUB1    STMED   r13!, {r0-r2, r14}    ;salva registros en la pila
        …….
        LDMED   r13!, {r0-r2, r14}    ;recupera registros de la pila
        MOV     r15, r14
```



5. Soporte de las instrucciones de control a las construcciones de alto nivel

5.1. Construcciones condicionales

IF-THEN

L.A.N.

```
IF  A  ope  B  THEN
        <sentencias>
```

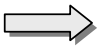
L.M.

```
        CMP  A, B
        B¬ope  L
        <sentencias>
L       ......
```

IF-THEN-ELSE

L.A.N.

```
IF  A  ope  B  THEN
        <sentencias1>
ELSE
        <sentencias2>
```



L.M.

```
        CMP  A, B
        Bope  L1
        <sentencias2>
        BRA  L2
L1      <sentencias1>
L2      ......
```

IF-THEN-ELSE-IF

L.A.N.

```
IF  A1  ope1  B1  THEN
        <sentencias1>
ELSE IF A2  ope2  B2  THEN
        <sentencias2>
......
ELSE IF An  open  Bn  THEN
        <sentenciasn>
ELSE
        <sentencias>
```

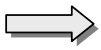
L.M.

```
        CMP  A1, B1
        B¬ope1  L1
        <sentencias1>
        BRA  L
L1      CMP  A2, B2
        B¬ope2  L2
        <sentencias2>
        BRA  L
L2      .......

Ln-1    CMP  An, Bn
        B¬open  Ln
        <sentenciasn>
        BRA  L
Ln      <sentencias>
L       ........
```

CASE

L.A.N.

```
CASE  K      OF
          0: <sentencias1>
          1: <sentencias2>
          .....
          n: <sentenciasn>
   defecto: <sentencias>
```



L.M.

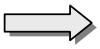
```
TABLA
          L0
          L1
          L2
          ...
          Ln
          Ln+1

          MOVE K, A1
          LEA TABLA(A1), A2
          JMP  (A2)
L0        <sentencias0>
          BRA  FIN
L1        <sentencias1>
          BRA  FIN
                ....
Ln        <sentenciasn>
          BRA  FIN
Ln+1      <sentenciasn+1>
FIN       ........
```

5.2. Construcciones iterativas

WHILE-DO

L.A.N.

```
WHILE  A  ope  B
          BEGIN
          <sentencias>
END DO
```



L.M.

```
INI       CMP  A, B
          B¬ope  L
          <sentencias>
          BRA  INI
L         ......
```

REPEAT

L.A.N.

```
REPEAT
          <sentencias>
UNTIL  A  ope  B
```

L.M.

```
INI       <sentencias>
          CMP  A, B
          B¬ope  INI
          ......
```

FOR

L.A.N.

```
FOR  A = B   TO  C   DO
          <sentencias>
```

L.M.

```
          MOVE B, A
     L1   CMP  C, A
          BGT  L2
          <sentencias>
          INC  A
          BRA  L1
     L2   ......
```

Arquitectura	**Procesador**	**Parámetros**	**Microarquitectura**
IA-32 (extensión de X86)	486		486
	Pentium	3,3M transistores	P5
	Pentium Pro (variante Xeon)	5,5M transistores	P6
	Pentium II (variante Celeron)	MMX	
	Pentium III	SSE (Streaming SIMD Extension)	
	Pentium 4	42M transistores 476 pines 0,18 micras 2 GHz	NetBurst