

# Resumen

## Organización del

## Computador

### (TBO23)



Lorenzo Minervino



## Sistemas de numeración

Sistemas de numeración no posicionales: son aquellas en los cuales el valor de un símbolo no depende de su posición. Ejemplo: Sistema en números romanos

Sistemas de numeración posicionales: son aquellos en los cuales el valor de un símbolo depende del lugar que ocupe dentro del número. El más conocido es el sistema decimal

Sistemas posicionales de Base b: un sistema de numeración posicional de base b en donde la base siempre es igual a la cantidad de símbolos que posee el sistema, en el decimal es igual a 10 dado que tenemos un total de 10 dígitos diferentes (0,1,2,3,4,5,6,7,8,9).

La tabla muestra los sistemas: Decimal, Binario, Octal y Hexadecimal

00	0000	00	00
01	0001	01	01
02	0010	02	02
03	0011	03	03
04	0100	04	04
05	0101	05	05
06	0110	06	06
07	0111	07	07
08	1000	10	08
09	1001	11	09
10	1010	12	0A
11	1011	13	0B
12	1100	14	0C
13	1101	15	0D
14	1110	16	0E
15	1111	17	0F
16	10000	20	10

La representación de un número se define a partir del Teorema Fundamental de la Numeración:

$$C = X * B = \sum_{i=-\infty}^{+\infty} X_i * b^i$$

donde C es un número en cualquier base, X es el vector de dígitos, B es el vector de pesos y b es la base del sistema de numeración

### Teorema fundamental de la numeración

$$\begin{array}{rcl} 4 & 3 & 2 & 1 & 0 & -1 & -2 \longrightarrow i \\ 12456,21 [10] & = & 1 \times 10^{-2} + 2 \times 10^{-1} + 6 \times 10^0 + 5 \times 10^1 + 4 \times 10^2 + 2 \times 10^3 + 1 \times 10^4 [10] = \\ 11 & 10 & 1 & 0 & -1 & -10 \longrightarrow i \\ 1101,01 [2] & = & 1 \times 10^{-10} + 0 \times 10^{-9} + 1 \times 10^{-8} + 0 \times 10^{-7} + 1 \times 10^{-6} + 1 \times 10^{-5} [2] = \end{array}$$

### Cambios de base

#### Casos

Número enteros:

- a) Base b a base 10: Teorema fundamental de la numeración (se usa y después se suman los términos)
- b) Base 10 a base b: se realizan divisiones sucesivas hasta que el cociente sea menor que el divisor (base deseada), luego juntar el último cociente obtenido junto con todos los restos de abajo hacia arriba

#### Ejemplos:

$$34_{10} \rightarrow ( )_2 \quad \begin{array}{r} 34 \mid 2 \\ 0 \quad 17 \mid 2 \\ \quad 1 \quad 8 \mid 2 \\ \quad 0 \quad 4 \mid 2 \\ \quad 0 \quad 2 \mid 2 \\ \quad 0 \quad \textcolor{red}{1} \end{array}$$

Hemos llegado al final de las divisiones sucesivas notar que  $1 < 2 \rightarrow$   
El resultado final es  $(100010)_2$

- c) Base b a base c: se combinan los dos métodos anteriormente explicados pivotando con la base 10

Partes fraccionarias:

- a) 0, Base b a Base 10: Teorema fundamental de la numeración
- b) 0, base 10 a base b: se resuelve mediante multiplicaciones sucesivas tomando de cada resultado la parte entera. Se termina cuando la parte fraccionaria es igual a cero. En caso de no llegar a este resultado cuantos

más decimales se toma mayor precisión se alcanza. Los criterios de corte pueden ser: una multiplicación da 0 (no hay más dígitos), que te des cuenta que hay infinitos dígitos o que alguna multiplicación se repita (número periódico)

Ejemplo:

$$0,125_{10} \rightarrow ( )_2$$

0,125 × 2 = 0,250	0
0,250 × 2 = 0,500	0
0,500 × 2 = 1,000	1
<b>0,125<sub>10</sub> = 0,001<sub>2</sub></b>	

c) 0, Base b a base c: nuevamente se pivotea con base 10

3210E

Aclaración: Si se tiene un número que posee parte entera y fraccionaria se realiza el cambio para cada parte por separado y luego se suman los resultados obtenidos

Casos especiales de cambio de base

a)  $b^x = p$

Se irán formando grupos de x dígitos y se hará el cambio para cada uno de estos grupos en forma independiente. Para la parte entera se empiezan a formar los grupos de derecha a izquierda en caso de ser necesario se completa con ceros a izquierda. Para la parte fraccionaria se comienza a formar grupos de izquierda a derecha y de ser necesario se completa con ceros a derecha

b)  $b^{\frac{1}{x}} = p$  (la inversa al método anterior)

Cada dígito en la base b se expandirá en x dígitos en la base p

**Ejemplo:**

$$10110_2 \rightarrow ( )_8$$

$2^3 = 8$  tomo de 3 dígitos

$$(010 | 110)_2 = 26_8$$

$$110_2 = 6_8$$

$$010_2 = 2_8$$

$$AE75_{16} \rightarrow ( )_2$$

$16^{1/4} = 2$  se expande en 4 dígitos binarios

$$5_{16} = 0101_2$$

$$7_{16} = 0111_2$$

$$E_{16} = 1110_2$$

$$A_{16} = 1010_2$$

$$(1010111001110101)_2$$

Números periódicos:

$$0, \widehat{456} [10] = \frac{456 - 4}{990} [10]$$

Expresión: en el numerador va el número detrás de la coma menos los dígitos no periódicos; en el denominador va el numero mayor de la base una vez por cada digito periodico y un cero por cada digito no periodico

### Formatos de representación en una computadora

Formato Binario de Punto Fijo sin signo

Pasos:

- 1) Pasar de la base origen a base 2
- 2) Insertar los dígitos binarios (bits) obtenidos en el espacio de almacenamiento y completar con ceros de ser necesario



El valor absoluto habla sobre un número por sí solo en alguna base determinada pero sin ningún formato de representación, el formato es un formato dado tal como punto fijo sin signo y la configuración tiene que ver con la base en la que está expresada.

Overflow: ocurre cuando una operación aritmética en un formato específico da como resultado un número que para ser representado en este formato necesita más bits de los disponibles

$$\begin{array}{r} 10011010 \\ + 11011011 \\ \hline 10011010 \\ \boxed{1}01110101 \\ \downarrow \\ \text{Overflow!} \end{array}$$

## Metodos representación números negativos

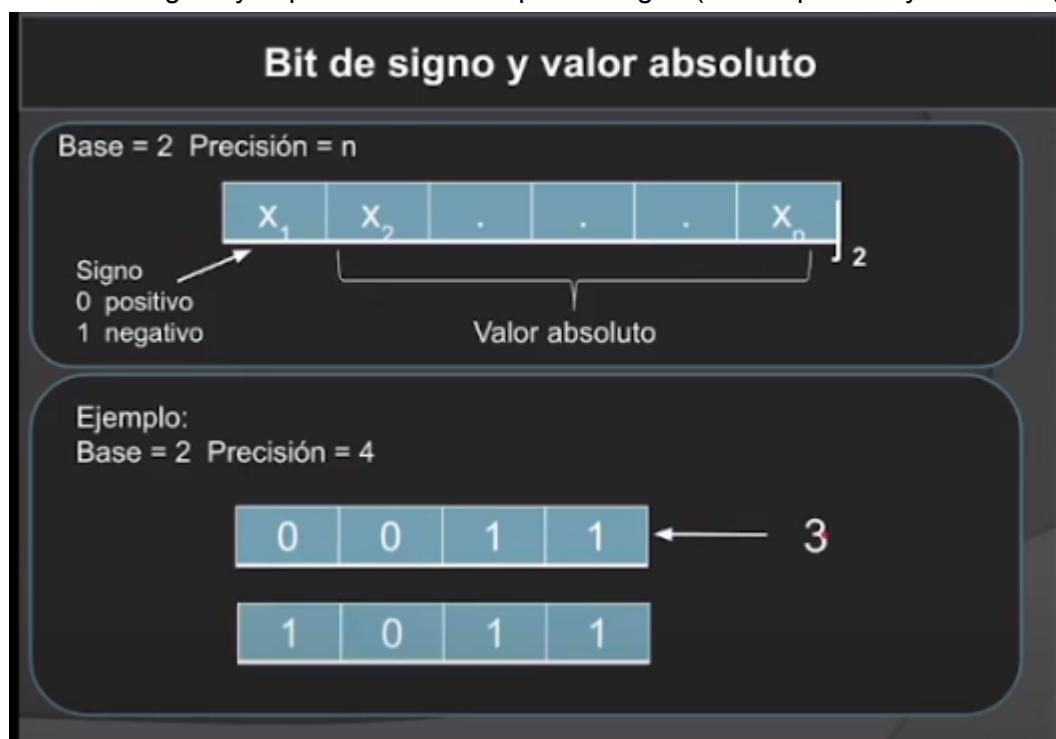
Base: base del sistema de numeración

Precisión = n = cantidad de dígitos a utilizar

$B^n$  = #total de números representables. Slendo B la cantidad de símbolos (base) y n la precisión ( cantidad de dígitos)

### **Bit de signo y valor absoluto:**

se usan n dígitos y el primero se utiliza para el signo (0 si es positivo y 1 si es negativo)



Paso por paso

Representar: -6

- 1) Ver signo para determinar el primer bit: 1 (por ser negativo)
- 2) Pasar valor absoluto a base 2:  $|-6_{10}| = 6_{10} = 110_2$
- 3) Concatenar los bits: 1110

Indicar número almacenado en 1101

- 1) Ver primer bit para determinar el signo: negativo (por ser 1)
- 2) Pasar a base 10 los bits descartando el primero:  $101_2 = 5_{10}$
- 3) Indicar el número según signo y valor obtenidos: -5

Rango de representación:

$$\text{Mínimo: } -(2^{n-1} - 1)$$

$$\text{Máximo: } (2^{n-1} - 1)$$

Ventajas:

Rango simétrico

Desventajas:

Doble representación del 0

No permite operar aritméticamente

### Complemento a 1:

Se usa el primer dígito para el signo (0 si es positivo y 1 si es negativo). Si es positivo se usa el valor absoluto de los dígitos restantes; Si es negativo se aplica la función NOT (reemplazar todos los 1 por 0 y los 0 por 1 en los dígitos restantes)



Paso por paso:

Base = 2 y precisión = 4

Representar -6

- 1) Pasar valor absoluto a base 2:  $|-6_{10}| = 6_{10} = 110_2$
- 2) Completar con 0 a izquierda hasta completar n: 0110
- 3) Si es negativo, complementar (hacer NOT): 1001

Indicar número almacenado en 1101

1. Si primer bit es 1 (es negativo), complementar (hacer NOT): 0010
2. Pasar a base 10:  $0010_2 = 2_{10}$
3. Indicar el número según signo y valor obtenidos: -2

Rango de representación:

$$\text{Mínimo: } -(2^{n-1} - 1)$$

$$\text{Máximo: } (2^{n-1} - 1)$$

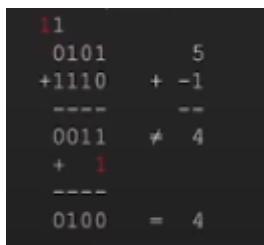
Desventajas:

Doble representación del 0

Ventajas:

Rango simétrico

Permite operar aritméticamente sumando el “end-around carry”


$$\begin{array}{r} 0101 \\ +1110 \\ \hline 0011 \end{array} \quad \begin{array}{l} 5 \\ + -1 \\ \hline 4 \end{array}$$

The diagram shows a binary addition problem. The first row shows the binary numbers 0101 and +1110. The second row shows the decimal equivalents 5 and -1. The third row shows the result 0011 and the decimal equivalent 4. A red '1' is written above the first column of the addition, indicating a carry. A red '+' is written between the two numbers in the second row, and a red '=' is written below the result in the third row.

## Complemento a la Base

$$Rep(x_b) = x_b \text{ si } x \geq 0$$

$$Rep(x_b) = Cb(|x_b|) \text{ si } x < 0$$

¿Qué es Cb?

$$Cb(r) + r = B^n \Rightarrow Cb(r) = B^n - r$$

Notar que si k es complemento de r  $\Rightarrow k + r = B^n \Rightarrow r \text{ es complemento de } k$

**Ejemplos:**

$$B = 10 \quad n = 2 \Rightarrow B^n = 10^2 = 100$$

$$Rep(3) = 03$$

$$Rep(-3) = Cb(3) = 100 - 3 = 97$$

$$Rep(-1) = Cb(1) = 100 - 1 = 99$$

$$Rep(-50) = Cb(50) = 100 - 50 = 50$$

Rep(50) => NO SE PUEDE

=> 49 es el mayor positivo representable

## Rango de representación

Rango de representación:

$$\text{Mínimo: } -(B^n / 2)$$

$$\text{Máximo: } (B^n / 2) - 1$$

Desventajas:

Rango asimétrico (un negativo más)

Ventajas:

Única representación del 0

Permite operar aritméticamente

**Permite operar aritméticamente:**

**Sumas**

$5 + 2 \quad (\text{es } 7)$	$5 + (-2) \quad (\text{es } 3)$	$-5 + 2 \quad (\text{es } -3)$	$-5 + (-2) \quad (\text{es } -7)$
$05 + 02$	$05 + 98$	$95 + 02$	$95 + 98$
$\begin{array}{r} 0 \\ 05 \\ +02 \\ \hline -07 \end{array}$	$\begin{array}{r} 1 \\ 05 \\ +98 \\ \hline +03 \end{array}$	$\begin{array}{r} 0 \\ 95 \\ +02 \\ \hline -97 \end{array}$	$\begin{array}{r} 1 \\ 95 \\ +98 \\ \hline -93 \end{array}$
$(A)$	$(B)$	$(C)$	$(D)$

**Restas: al plantear A-B se transforma en A+Bcomp**

$5 - 2 \quad (\text{es } 3)$	$5 - (-2) \quad (\text{es } 7)$	$-5 - 2 \quad (\text{es } -7)$	$-5 - (-2) \quad (\text{es } -3)$
$05 - 02$	$05 - 98$	$95 - 02$	$95 - 98$
$05 + \text{Cb}(2)$	$05 + \text{Cb}(98)$	$95 + \text{Cb}(2)$	$95 + \text{Cb}(98)$
$= 05 + 98$	$= 05 + 02$	$= 95 + 98$	$= 95 + 02$
$(B)$	$(A)$	$(D)$	$(C)$

Notar el carry que se descarta cuando la suma se va fuera de rango

Conclusión:

$A - B$  se trabaja como  $A + \text{Comp}(B)$  (no importa el signo de  $B$ )

## Complemento a 2

$$Rep(x_b) = x_b \text{ si } x \geq 0$$

$$Rep(x_b) = NOT(|x_2|) + 1 \text{ si } x < 0$$

### Ejemplos:

B = 2 n = 4

$$\begin{aligned} \text{Rep}(3) &= 0011 \\ \text{Rep}(-3) &= \text{NOT}(0011) + 1 \\ &= 1100 + 1 \\ &= 1101 \\ \text{Rep}(-1) &= \text{NOT}(0001) + 1 \\ &= 1110 + 1 \\ &= 1111 \end{aligned}$$

$$\begin{aligned} \text{Rep}(-8) &= \text{NOT}(1000) + 1 \\ &= 0111 + 1 \\ &= 1000 \\ \text{Rep}(8) &\Rightarrow \text{NO SE} \\ &\Rightarrow 7 = 0111_{12} \text{ es el mayor} \\ &\text{positivo representable} \end{aligned}$$

### Veamos...

0000	0001
0001	+1111
0010	----
0011	10000
0100	0011
0101	+1101
0110	----
0111	10000
1000	1000
1001	+1000
1010	----
1011	10000
1100	0010
1101	+1000
1110	----
1111	10000

NOT + 1  
es Cb  
con b=2

Anotación: concluimos que el Complemento a 2 es el caso particular de complemento a la base pero que nos permite resolver de una manera más mecánica

Paso a paso

Base = 2 y precisión = 4

Representar -6

- 1) Pasar valor absoluto a base 2:  $|-6_{10}| = 6_2 = 110_2$
- 2) Completar con 0 a izquierda hasta completar n: 0110
- 3) Si es negativo, complementar (hacer NOT + 1):  $1001 + 1 = 1010$

Indicar número almacenado en 1101

1. Si primer bit es 1 (es negativo), complementar (hacer NOT + 1):  $0010 + 1 = 0011$
2. Pasar a base 10 los bits:  $0011_2 = 3_{10}$
3. Indicar el número según signo y valor obtenido: -3

Rango de representación:

Mínimo:  $- (2^{n-1})$

Máximo:  $(2^{n-1} - 1)$

Desventajas:

Rango asimétrico (un negativo más)

Ventajas

Única representación del 0

Permite operar aritméticamente

Aplica la misma mecánica de resolver  $X-Y$  como  $X + Cb(Y)$

### Exceso de base

$$Rep(x_b) = x_b + \text{exceso } \forall x$$

¿Pero que es el exceso?

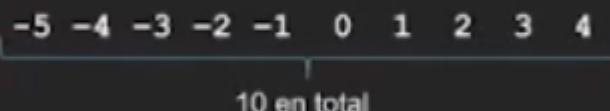
Es  $B^n/2$

Entendamos por qué:

Con  $B=10$  y  $n = 1$

$B^n = 10$  valores posibles

Cuál sería el rango de números a representar "más justo"?



Cuánto hay q sumar como mínimo a cada negativo para que "desaparezca" el signo?

5

Que termina siendo  $B^n/2$

Números a representar

-5 -4 -3 -2 -1 0 1 2 3 4

Como se representan

0 1 2 3 4 5 6 7 8 9

### Ejemplos:

$$B = 10 \quad n = 2$$

$$\Rightarrow B^n/2 = 10^2/2 = 50$$

$$\text{Rep}(3) = 3 + 50 = 53$$

$$\text{Rep}(-3) = -3 + 50 = 47$$

$$\text{Rep}(0) = 0 + 50 = 50$$

$$\text{Rep}(-50) = -50 + 50 = 0$$

Rep(-51) NO SE PUEDE (da -1)

$$\text{Rep}(49) = 49 + 50 = 99$$

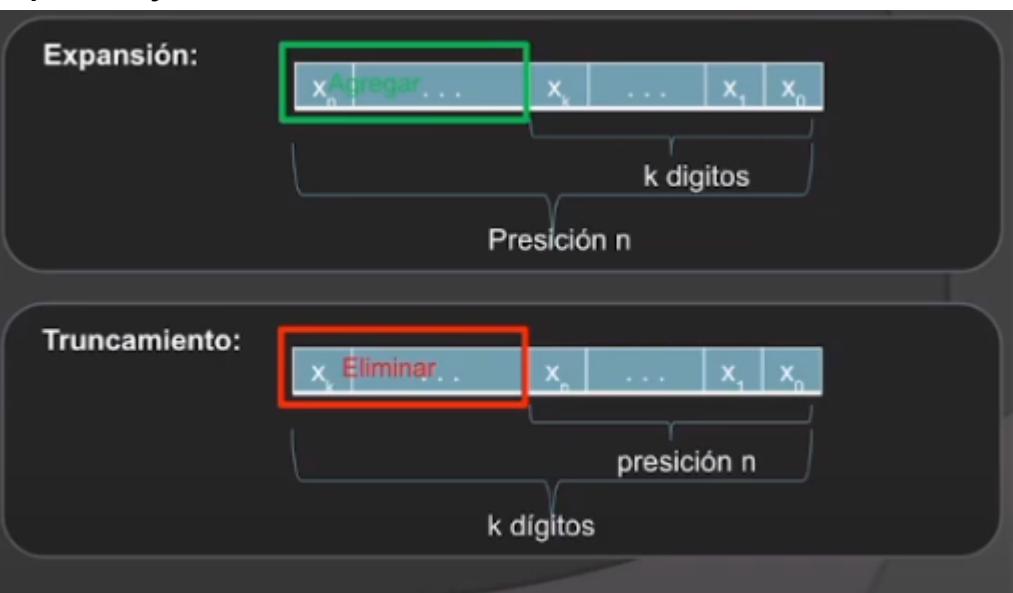
Rep(50) NO SE PUEDE (da 100)

## Formato y Configuración

Formato: Representación computacional de un número

Configuración: Expresión en una determinada base de un número en un formato

## Expansión y Truncamiento



## Formatos de representación de números enteros

### **Binario de punto fijo sin signo**

Cómo almacenar un número

- 1) Pasar el numero a base 2
- 2) Completar con 0 a izquierda hasta alcanzar n dígitos

Cómo recuperar un número almacenado

Pasos anteriores en orden inverso

Rango representación

Mínimo: 0

Máximo:  $2^n - 1$

### Binario de punto fijo con signo

Enteros positivos y negativos

Es la implementación del método complemento a 2

Cómo almacenar un número

- 1) Pasar el numero a base 2
- 2) Completar con 0 a izquierda hasta alcanzar n dígitos
- 3) Si el número es negativo, complementar usando método de “complemento a 2” (Not + 1)

Cómo recuperar un número almacenado

- 1) Si el primer bit es 1 (es negativo), complementar
- 2) Quitar 0 a izquierda
- 3) Pasar a base 10 y colocar el signo que corresponda

Validación Overflow en operaciones aritméticas

B=2      n=4

Resolver  $7 + 1$

$7_{(10)} = 0111_{(2)}$   
 $1_{(10)} = 1_{(2)}$

**0111** Últimos 2 acarreos distintos  
0111 => OVERFLOW

+ 0001

----

1000

Resolver  $7 - 1$

$7_{(10)} = 0111_{(2)}$   
 $1_{(10)} = 1_{(2)}$

Haloo C(1) para hacer  $7+C(1)$

NOT(0001) = 1110

     + 1

-----

1111

Ahora sumo

**1111** Últimos 2 acarreos iguales  
0111 => VALIDO

+ 1111

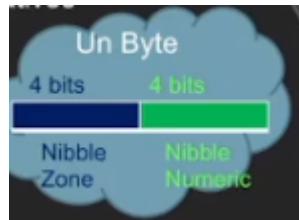
----

0110

28

## BCD Empaquetado

Enteros positivos y negativos



Cómo almacenar un número

- 1) Pasar el numero a base 10
- 2) Colocar con dígito en los nibbles dejando libre el último (el de la derecha)
- 3) Colocar en el último nibble el signo siendo C, A, F o E para positivos y B o D para negativos

Ej. n=3 +123<sub>|10</sub> --> 00123A<sub>|16</sub>      -456<sub>|10</sub> --> 00456B<sub>|16</sub>

Cómo recuperar un número almacenado

- 1) Tomar cada dígito en los nibbles (excepto el último) para armar cadena en base 10
- 2) Colocar el signo según el dígito del último nibble

## Zoneado

Cómo almacenar un número

- 1) Pasar el numero a base 10
- 2) Colocar c/dígito en los nibbles numeric
- 3) Colocar una F en cada nibble zone excepto en el último
- 4) Colocar en el último nibble zone el signo siendo C, A, F o E para positivos B o D para negativos

Ej. n=4 +123<sub>|10</sub> --> F0F1F2A3<sub>|16</sub>      -456<sub>|10</sub> --> F0F4F5B6<sub>|16</sub>

Cómo recuperar un número almacenado

- 1) Tomar cada dígito en los nibbles (excepto el último) para armar la cadena en base 10

## Representación en forma digital/numéricas de los caracteres

- 1) ASCII (American Standard Code for Information Interchange): 7 bits ASCII Básico u 8 bits ASCII Extendido
- 2) EBCDIC (Extended Binary Coded Decimal Interchange Code): 8 bits
- 3) UNICODE Consortium: CODificación Universal / Estándar de los Caracteres

Hex	Dec	EBCDIC	ASCII
23	35		#
30	48		0
31	49		1
42	66		B
62	98		b
7B	123	#	{
82	130	b	
C0	192	{	
C2	194	B	
F0	240	0	
F1	241	1	

## Binario punto fijo flotante IEEE 754

Notación científica  $S M \times B^E$ , ejemplo:  $- / + 7,65987 \times 10^{-1}$

Mantisa Normalizada:  $0 < M < B$  (el dígito de la cifra significativa en Binario es 1)

Precisión: Simple (32) / Doble (64) / Extendida (128)

Representación:

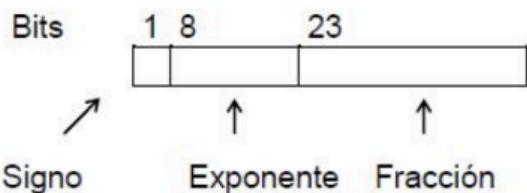
- Signo
- Exponente
- Mantisa

Exponente con Exceso:

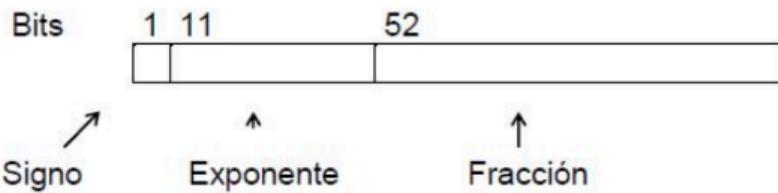
- 127 / 1023 / 16383

Mantisa normalizada

### Simple precisión



### Doble precisión



Ejemplo:

–  $123,456_{10}$

- 1) Paso de base 10 a base 2

$$123_{10} \rightarrow 1111011_2$$

$$0,456_{10} \rightarrow 011101000_2$$

$$- 123,456 \rightarrow 1111011,011101000_2$$

- 2) Normalizar

$$- 1,11101101110100 \times 10^{110}_2$$

- 3) Almaceno

$$\text{Signo} = 1$$

EExceso

$$6 + 127 = 133_{10}$$

$$133_{10} = 10000101_2$$

- 4) Nro.Final

$$1 | 10000101 | 111011011101000000000000_2$$

**Binario punto fijo flotante IEEE 754** puede representar cualquier número de la recta real:

Normalizado	$\pm$	$0 < \text{Exp} < \text{Max}$	Cualquier patrón de bits
Desnormalizado	$\pm$	0	Cualquier patrón de bits $\neq 0$
Cero	$\pm$	0	0
Infinito	$\pm$	111...1	0
NAN	$\pm$	111...1	Cualquier patrón de bits $\neq 0$

## **Máquina elemental**

### **Clasificación de computadoras según su generación**

#### **Generación 0: Computadoras mecánicas (1642-1945)**

- Pascal: máquina mecánica de cálculo (sumas y restas)
- Babbage: máquina de diferencias (algoritmo de diferencias finitas - polinomios)
- Aiken: máquina analítica Mark I (Harvard 1944)

#### **Generación 1: Tubos de vacío (1945-1955)**

- Turing: computadora electrónica
- Mauchly / Eckert: programable por switches
- Von Neumann: programa almacenado

#### **Generación 2: Transistores**

- DEC: PDP 1 (primera minicomputadora) y PDP 8 (primera mini masiva / omnibus)
- IBM: dominio en uso científico
- CDC: primera supercomputadora
- Burroughs: primera computadora diseñada para un lenguaje de alto nivel

#### **Generación 3: Circuitos integrados**

- PDP 11: minicomputadora dominante
- IBM: System/360 (familia de computadoras):
  - Uso comercial y científico
  - distintos modelos compatibles entre sí con distintas capacidades
  - multiprogramación
  - simulaba otras arquitecturas con microarquitectura programada

#### **Generación 4: integración a muy gran escala:**

- IBM: PC (comienzo era computadores personales)
- APPLE: (primera computadora con GUI)
- DEC (primera computadora RISC de 64 bits)
- COMMODORE / ATARI: computadoras hogareñas sin estándar

#### **Generación 5: computadoras “invisibles”:**

- APPLE: Newton (primera palmtop)
- Computadores embebidos: relojes inteligentes, celulares y electrodomésticos

## Principios básicos de Von Neumann:

1. Programa almacenado: tanto las instrucciones como los datos que en ellas se usan residen en una misma memoria
2. Ruptura de secuencia: debe existir una instrucción que permite a la máquina no seguir con la secuencia de ejecución

## Conceptos preliminares

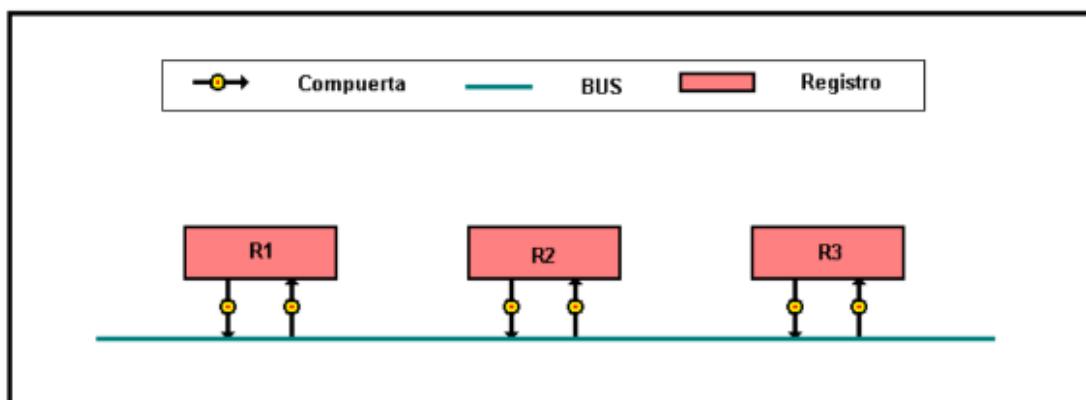
Registro: los registros son considerados los bloques más importantes de un computador. Una definición general los identifica como una “memoria muy rápida” que permite almacenar una cierta cantidad de bits (información).

Compuerta: las compuertas constituyen la base del hardware sobre la cual se construyen los computadores digitales. Son circuitos electrónicos biestables unidireccionales, es decir, permiten el pasaje de información en un sólo sentido y admiten únicamente dos estados (abierto / cerrado, 1 / 0).

Bus de datos: mueve la información por los componentes de hardware internos y externos del sistema tanto de entrada como de salida (teclado, Mouse etc.)

Bus de Direcciones: ubica los datos en la memoria teniendo relación directa con los procesos de la CPU.

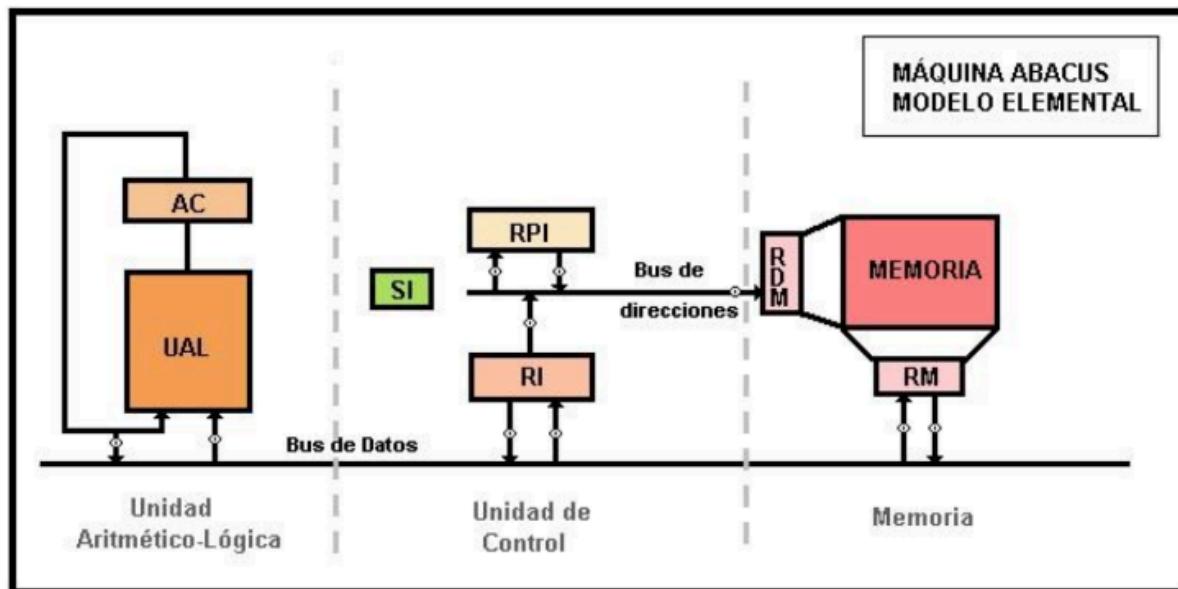
Bus de control: marca el estado de una instrucción que fue dada a la PC.



Para intercambiar información entre dos registros, es necesario que dicha información viaje a través del bus. El traspaso de la misma entre un registro y el bus de datos, por ejemplo, está regido por las compuertas: sólo si la compuerta está abierta se produce el pasaje de datos. Cuando se abre una compuerta la información está disponible en el bus y mientras esté abierta los datos permanecerán en el mismo. Sin embargo, el bus no tiene capacidad de almacenamiento, por ende, al cerrar la compuerta la información desaparecerá. La compuerta que manda información al bus es única, es decir, no pueden existir dos abiertas simultáneamente, esto es importante tenerlo presente a la hora de

indicar la secuencia de apertura de compuertas en el proceso de ejecución de una instrucción.

### ABACUS: MAQUINA ELEMENTAL



### Registros

**AC:** acumulador

**RI:** registro de instrucción

**SI:** secuenciador de instrucciones

**RPI:** registro de próxima instrucción

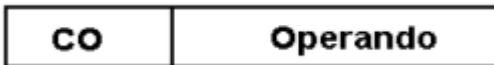
**RI:** registro de instrucción

**RDM:** registro de direcciones de memoria

**RM:** registro de memoria

### Descripción de componentes:

- 1) **UAL:** Abacus es una máquina de una sola dirección, su unidad aritmético-lógica posee un registro particular llamado acumulador (AC) que sirve tanto para albergar el primer operando como para albergar el resultado. Esta característica permite instrucciones de una sola dirección: la del segundo operando. Entre las operaciones que realiza esta: CARGA, ALMACENAMIENTO, SUMA, LÓGICAS, etc. Es importante recordar que **todas las operaciones** se realizan “contra” el acumulador y el resultado siempre queda almacenado en él.
- 2) **UC:** Es la encargada de extraer y analizar las instrucciones de la memoria central. Esta contiene dos registros.  
**RPI:** contiene la dirección de la próxima instrucción a ejecutar, se comunica con la memoria y con el RI a través del bus de direcciones. A medida que se van ejecutando las instrucciones este registro aumenta su contenido en una unidad excepto para las instrucciones de ruptura de secuencia.  
**RI:** contiene la instrucción extraída de la memoria, en ella podemos identificar dos partes fundamentales: el código de operación y la dirección del operando. Este registro se comunica con la memoria mediante el Bus de Datos.



En cuanto al SI cabe destacar que tras analizar el código de operación distribuye las órdenes de la Unidad de Control a la Unidad Aritmético-Lógica y a la Memoria para ejecutar las fases de la instrucción, en otras palabras, es el encargado de administrar la apertura y cierre de las compuertas para el correcto funcionamiento y pasaje de la información.

**3) Memoria:** esta contiene dos registros

RDM: contiene la dirección de la celda de memoria en (de) la cual se escribirá (leerá) la información

RM: contiene el dato que debe ser leído (escrito) desde (en) la memoria.

### Desarrollo de una instrucción

El desarrollo de una instrucción en Abacus puede descomponerse en 2 fases:

- **Fase de búsqueda:** consiste en localizar la instrucción que se va a ejecutar; esta fase es común a todos los tipos de instrucciones y, a su vez, la secuencia de acciones que se lleva a cabo es idéntica a la búsqueda de operandos. También se actualiza en forma secuencial la dirección de la siguiente instrucción a ejecutar.
- **Fase de ejecución:** como su nombre lo indica consiste en ejecutar la instrucción, claramente esta fase es dependiente del tipo de tarea a realizar (suma, almacenamiento, salto, etc.).

### Descripción de fases:

**Búsqueda de una instrucción:** La unidad de control ordena la transferencia del contenido del RPI al RDM y envía a la memoria una orden de lectura. El contenido de la celda de memoria queda almacenado en el RM, luego la Unidad de Control ordena la transferencia del contenido del RM al RI, pudiendo entonces los circuitos especializados analizar el código de operación de la instrucción. Finalmente se prepara el RPI para ejecutar la próxima instrucción.

$$\begin{array}{l}
 \text{RDM} \leftarrow (\text{RPI}) \\
 \text{RM} \leftarrow ((\text{RDM})) \\
 \text{RI} \leftarrow \text{RM} \\
 \text{RPI} \leftarrow (\text{RPI}) + 1
 \end{array}$$

Para realizar la búsqueda de un operando, una vez que la Unidad de control analiza el contenido del código de operación, ordena la transferencia del contenido del campo operando del RI al RDM y luego envía una orden de operación de lectura. El operando buscado queda disponible en el RM.

**Ejecución de una instrucción:** esta fase depende exclusivamente de la tarea a realizar:

- **Suma:** se debe sumar el contenido del RM al contenido del acumulador

$$\begin{array}{l} \text{RDM} \leftarrow (\text{Op}) \\ \text{RM} \leftarrow ((\text{RDM})) \\ \text{AC} \leftarrow \text{AC} + (\text{RM}) \end{array}$$

- **Carga:** se debe almacenar en el acumulador un dato contenido en memoria

$$\begin{array}{l} \text{RDM} \leftarrow (\text{Op}) \\ \text{RM} \leftarrow ((\text{RDM})) \\ \text{AC} \leftarrow (\text{RM}) \end{array}$$

- **Almacenamiento:** se debe "guardar" en memoria el contenido del acumulador

$$\begin{array}{l} \text{RDM} \leftarrow (\text{Op}) \\ \text{RM} \leftarrow (\text{AC}) \\ (\text{RDM}) \leftarrow (\text{RM}) \end{array}$$

- **Bifurcación:** se debe "saltar" a la dirección indicada en la instrucción. La dirección de bifurcación debe ser transferida al RPI (para buscar la próxima instrucción a ejecutar).

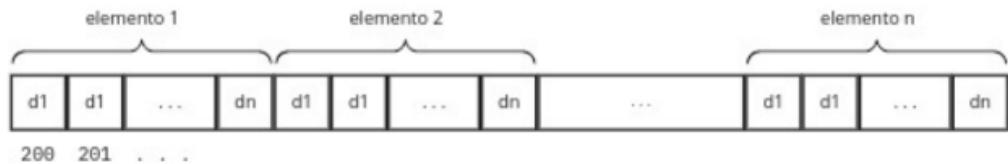
$$\text{RPI} \leftarrow (\text{Op})$$

### Set instrucciones

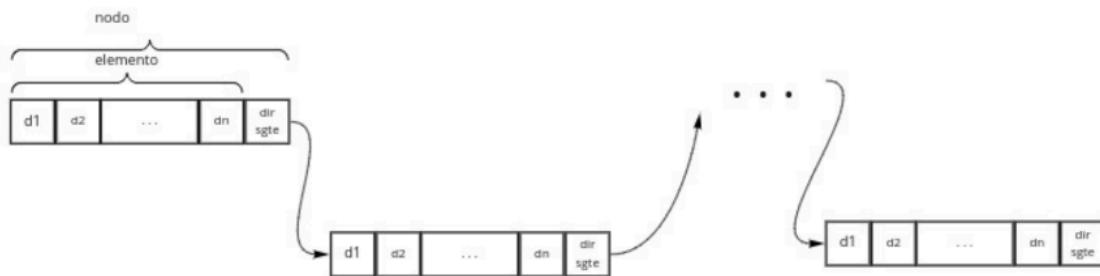
Código	Descripción
0	Carga inmediata
1	Carga
2	Almacenamiento
3	Suma
4	NOT (AC)
7	Bifurcación si AC = 0
8	Bifurcación si AC < 0

<b>9</b>	<b>Bifurcación si AC &gt; 0</b>
<b>F</b>	<b>FIN</b>

## Vectores



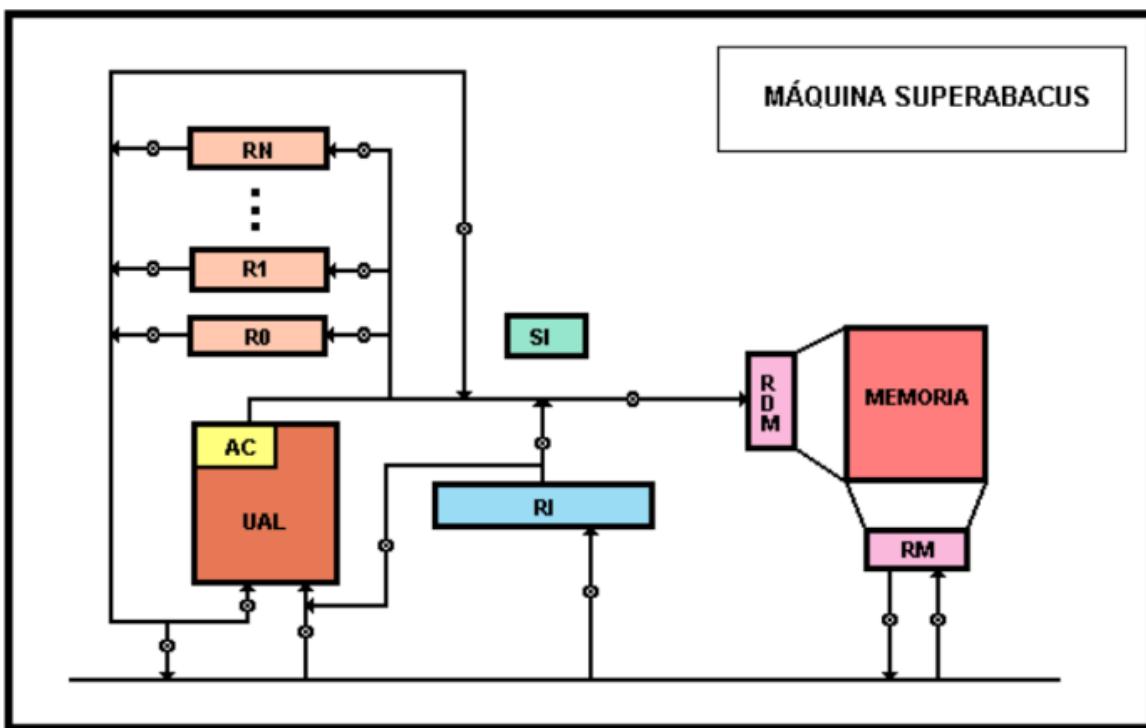
## Listas



En el contexto de *Abacus* si quiero cambiarle el signo a un valor tengo que aplicarle NOT y sumarle 1, manualmente.

### SUPERABACUS: MAQUINA ELEMENTAL

Superabacus es una máquina de tercera generación ya que posee un conjunto de registros banalizados, es decir, utilizables tanto como registros aritméticos o, según las condiciones de direccionamiento, como registro base o como registro de índice para cálculo de direcciones. Todos los cálculos se realizan en un solo sumador que actúa a la vez de unidad aritmético-lógica y de unidad de cálculo de direcciones.

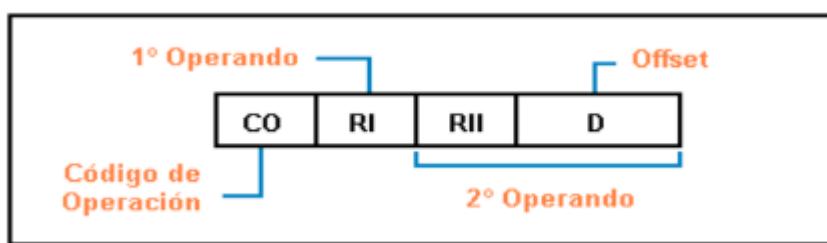


Como características principales podemos enumerar las siguientes:

1. Posee un conjunto de registros generales éstos pueden contener datos o direcciones.
2. No tiene RPI se asigna esa función a R0 (registro cero), cuyo incremento se consigue por transferencia vía el sumador.
3. Es una máquina de dos direcciones (1o y 2o operando).
4. La UAL se utiliza tanto para calcular direcciones como para operar con los datos.

#### Registro de Instrucción:

En Superabacus la estructura del RI es diferente respecto de la Máquina Abacus; es un Registro que alberga instrucciones de dos operandos. Posee Código de Operación, 1o Operando (RI) y 2o Operando (RII - D). La siguiente figura representa la estructura del RI:



RI y RII representan dos registros diferentes (se los indica con su número Ej.: R1, R5, etc.). El segundo operando puede contener la dirección de una celda de memoria, se obtiene sumando el Offset (desplazamiento) al contenido del registro indicado: A (celda) = (RII) + D.

#### **Fase de búsqueda:**

Al igual que en el caso de la máquina Abacus, la fase de búsqueda de las instrucciones de Superabacus es común para todas.

RDM	← ( R0 )
RM	← ( ( RDM ) )
RI	← ( RM )
AC	← ( R0 )
R0	← ( AC ) + 1

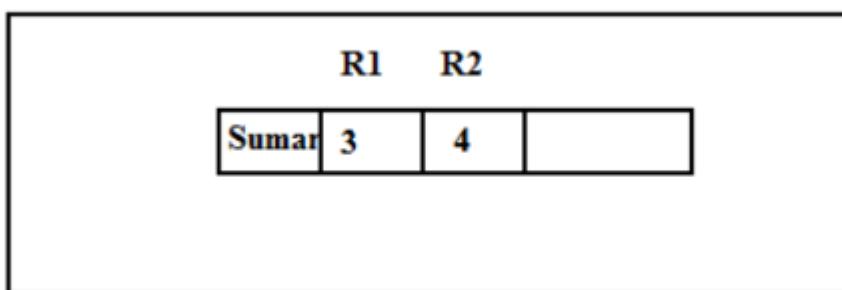
Como se puede apreciar, la gran diferencia con la máquina Abacus se da en la forma en que se incrementa el valor del R0 que hace las veces de RPI. En este caso ese incremento se resuelve usando la UAL en vez del SI.

#### **Instrucciones:**

Las operaciones pueden ser entre registros, entre un registro y un dato inmediato o entre un registro y un operando en memoria. Analizaremos cada caso para una instrucción SUMAR.

1. **Sumar Registro:** se suma el contenido de ambos registros y el resultado se almacena en aquel que se indica en el primer operando.

#### **SUMAR RI, RII (Ejemplo SUMAR 3, 4)**

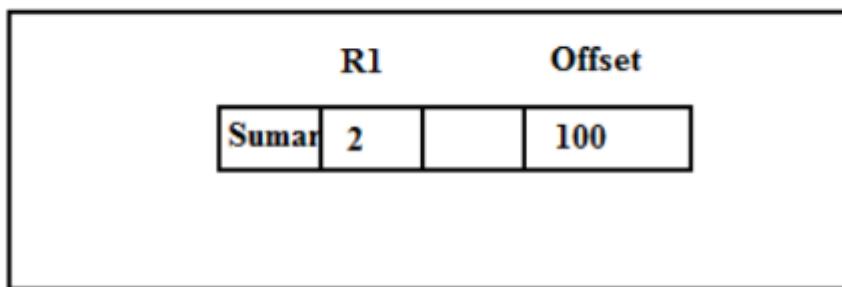


AC	← ( R3 )
AC	← ( R4 ) + ( AC )
R3	← ( AC )

2. **Sumar Inmediato:** se suma al registro indicado en el 1o operando el dato inmediato almacenado en la instrucción. El resultado se almacena en el

registro.

### SUMAR RI, DII (Ejemplo SUMAR 2, 100)



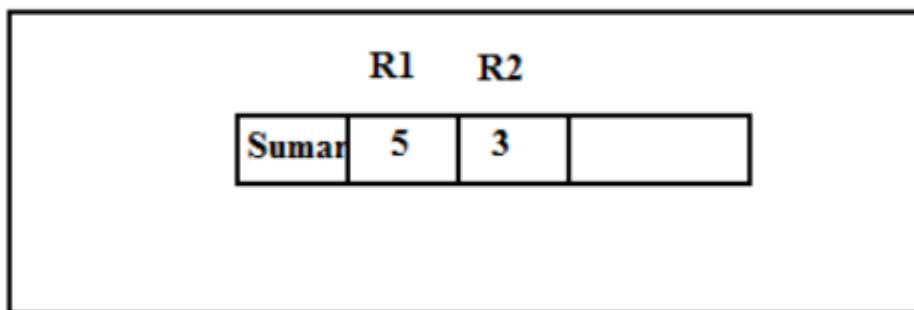
$$AC \leftarrow (R2) + 100$$

$$R2 \leftarrow (AC)$$

3. **Sumar Palabra en memoria (Registro Indirecto):** se suma al registro indicado en el primer operando el contenido de la celda de memoria cuya dirección está dada por el contenido del registro indicado en el segundo operando. El resultado queda almacenado en el registro del primer operando:

### SUMAR RI, (RII) (Ejemplo SUMAR 5, (3) )

### SUMAR RI, (RII) (Ejemplo SUMAR 5, (3) )



$$RDM \leftarrow (R3)$$

$$RM \leftarrow ((RDM))$$

$$AC \leftarrow (R5) + (RM)$$

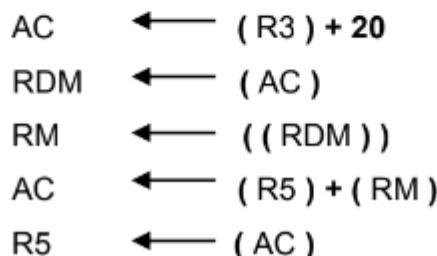
$$R5 \leftarrow (AC)$$

En este caso podemos notar que dado que el acumulador tiene una entrada desde los registros y una desde la memoria es posible realizar simultáneamente la suma del R5 y el RM.

4. **Sumar Palabra en memoria (Desplazamiento):** se suma al registro indicado en el primer operando el contenido de la celda de memoria cuya dirección está dada por el contenido del registro indicado en el segundo operando más el Offset. El resultado queda almacenado en el registro del primer operando:

#### SUMAR RI, DI (RII) (Ejemplo SUMAR 5, 20 (3) )

R1	R2	Offset
Sumar	5	3



En este caso podemos notar que dado que el acumulador tiene una entrada desde los registros y una desde la memoria es posible realizar simultáneamente la suma del R5 y el RM.

### Arquitectura del conjunto de instrucciones

ISA (Instruction Set Architecture) / Arquitectura de Programación

#### Repertorio de instrucciones:

¿Que es una instrucción de máquina?  
Opcode + Operandos (0 a n)

#### Categorías:

- Aritméticas y lógicas (add, subtract, and, or, etc)
- Movimiento de datos (load, store, move)
- Entrada/Salida (start I/O)
- Control de flujo (branch, jump, compare, call, return, etc)

#### Tipos de operandos:

- Registro
- Memoria

- Inmediato

Clasificación según la ubicación de los operandos

- Stack ('60s a '70s)
- Acumulador (antes de '60s)
- Registro-Memoria ('70s hasta ahora)
- Registro-Registro (Load/Store) ('60s hasta ahora)
- Memoria-Memoria ('70s a '80s)

¿Cómo se resuelve  $C = A + B$  según cada arquitectura?

Stack	Accumulator	Register (register-memory)	Register (load-store)	Memory (memory-memory)
Push A	Load A	Load R1,A	Load R1,A	Move C,A
Push B	Add B	Add R3,R1,B	Load R2,B	Add C,B
Add	Store C	Store R3,C	Add R3,R1,R2	
Pop C			Store R3,C	

Clasificación de la ISA según el número de direcciones:

- 0 dirección (Stack) ( $TOS \leftarrow TOS + Next$ )
- 1 dirección (Acumulador) ( $AC \leftarrow AC + Mem[A]$ )
- 2 direcciones (Reg-Mem/Reg-Reg/Mem-Mem) ( $R1 \leftarrow R1 + Mem[A]$ )
- 3 direcciones (Reg/Mem) ( $R1 \leftarrow R2 + R3$ )

### Formato de instrucciones (Encoding)

Definición: define el despliegue de los bits que componen la instrucción

Componentes:

- Opcode (código de operaciones)
- 0 a n operandos
- Modo de direccionamiento
- Flags

Clasificación:

- Fijo: la instrucción siempre mide la misma cantidad de bits
- Variable: la instrucción no es predecible, su cantidad de bits puede variar
- Híbrido: una mezcla entre fijo y variable

### Formatos:

- ARM
- x86
- IBM Mainframe

**Tipos de datos:**

- Numéricos
- Caracteres
- Datos lógicos
- Direcciones

**Modos de direccionamiento:**

- Inmediato
- Memoria directo
- Memoria indirecto
- Registro
- Registro indirecto
- Desplazamiento (relativo, registro base, indexado)
- Stack

**Memoria:**

- Direccionamiento (celda)
- Tamaño de palabra (Word size)
- Big vs Little Endian (ordenamiento de bytes)
- Espacio de direcciones (address space)

**Control de flujo:**

Métodos para evaluar condiciones de bifurcación:

- Condition Code (CC)
- Condition register
- Compare and Branch

## Lenguaje ensamblador

### **Lenguaje de máquina o lenguaje ensamblador**

Lenguaje de máquina / código de máquina: representación binaria de un programa de computadora el cual es leído e interpretado por el computador. Consiste en una secuencia de instrucciones de máquina”

Lenguaje ensamblador: “Representación simbólica del lenguaje de máquina de un procesador específico”

### **Lenguaje ensamblador**

Transición entre lenguaje de máquina y ensamblador

Address		Contents		
101	0010	0010	101	2201
102	0001	0010	102	1202
103	0001	0010	103	1203
104	0011	0010	104	3204
201	0000	0000	201	0002
202	0000	0000	202	0003
203	0000	0000	203	0004
204	0000	0000	204	0000

(a) Binary program

Address	Contents
101	2201
102	1202
103	1203
104	3204
201	0002
202	0003
203	0004
204	0000

(b) Hexadecimal program

Address	Instruction
101	LDA 201
102	ADD 202
103	ADD 203
104	STA 204
201	DAT 2
202	DAT 3
203	DAT 4
204	DAT 0

(c) Symbolic program

Label	Operation	Operand
FORMUL	LDA	I
	ADD	J
	ADD	K
	STA	N
I	DATA	2
J	DATA	3
K	DATA	4
N	DATA	0

(d) Assembly program

**Figure 13.13** Computation of the Formula  $N = I + J + K$

### ¿Para qué sirve el lenguaje ensamblador?

- Debugging y verificación
- Desarrollar compiladores
- Sistemas embebidos
- Drivers de hardware y código de sistema
- Acceder a instrucciones no disponibles de un lenguaje de alto nivel
- Código automodificable
- Optimizar código en tamaño
- Optimizar código en velocidad
- Biblioteca de funciones

### Elementos que lo componen

- Etiquetas
- Mnemónicos
- Operandos
- Comentarios

### Tipo de sentencias

- Instrucciones
- Directivas (pseudoinstrucciones)
- Macroinstrucciones

### Ejemplos

- Intel x86

- Arm
- IBM Mainframe

## Traducción vs Interpretación

Traductor: programa que convierte un programa de usuario escrito en un lenguaje (fuente) en otro lenguaje (destino).

Clasificación:

- Lenguajes compilados (C, C++, Go, Rust)
- Lenguajes ensambladores (Intel 64/IA-32, ARM, SPARC, MIPS, IA-64 (Itanium))

Interprete: programa que ejecuta directamente un programa de usuario escrito en un lenguaje fuente.

Clasificación:

- Lenguajes interpretados (Python, JavaScript, Ruby)
- Bytecode (Java)

## Ensambladores

Definición: programa que traduce un programa escrito en lenguaje ensamblador y produce código objeto como salida)

Traducción 1 a 1 a lenguaje máquina

Hay dos tipos:

- Dos pasadas
- Una pasada

En lenguaje de *dos pasadas* se realizan las siguientes acciones.

Primera pasada:

- Definición de etiquetas -> Tabla de símbolos
- LC: location counter (empieza en 0 con el 1er byte del código objeto ensamblado)
- Se examina cada sentencia de lenguaje ensamblador)
- Determinada la longitud de la instrucción de máquina (reconoce opcode + modo de direccionamiento + operandos) para actualizar el LC
- Revisa directivas del ensamblador (ejemplo: definición de áreas de storage)
- Por cada etiqueta encontrada se fija si está en la tabla de símbolos. Si no lo está la agrega (si es la definición, registra el LC como tal, sino lo registra como referenciado a la etiqueta)

Segunda pasada:

- Traduce el mnemónico en el opcode binario correspondiente
- Usa el opcode para determinar el formato de la instrucción y la posición y tamaño de cada uno de los campos de instrucción
- Traduce cada nombre de operando en el registro o código de memoria apropiado
- Traduce cada valor inmediato en un string binario en la instrucción

- Traduce las referencias a etiquetas en el valor apropiado de LC usando la tabla de símbolos
- Setear otros bits necesario en la codificación de la instrucción (ejemplo: indicadores de modo de direccionamiento, bits de código de condición, etc)

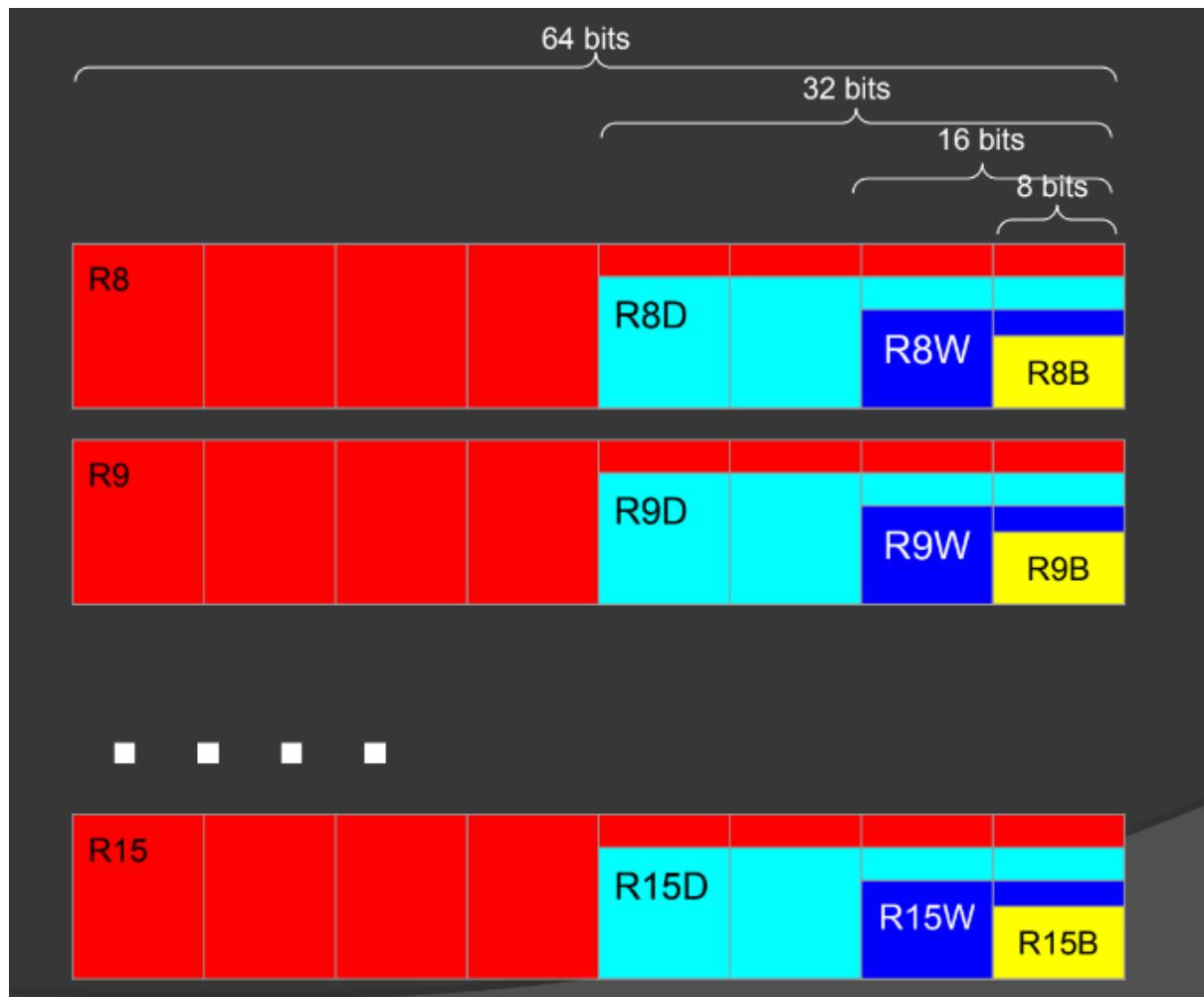
## Caso de estudio INTEL

### ISA (Instruction Set Architecture)

#### **Registros:**

Registros Generales:



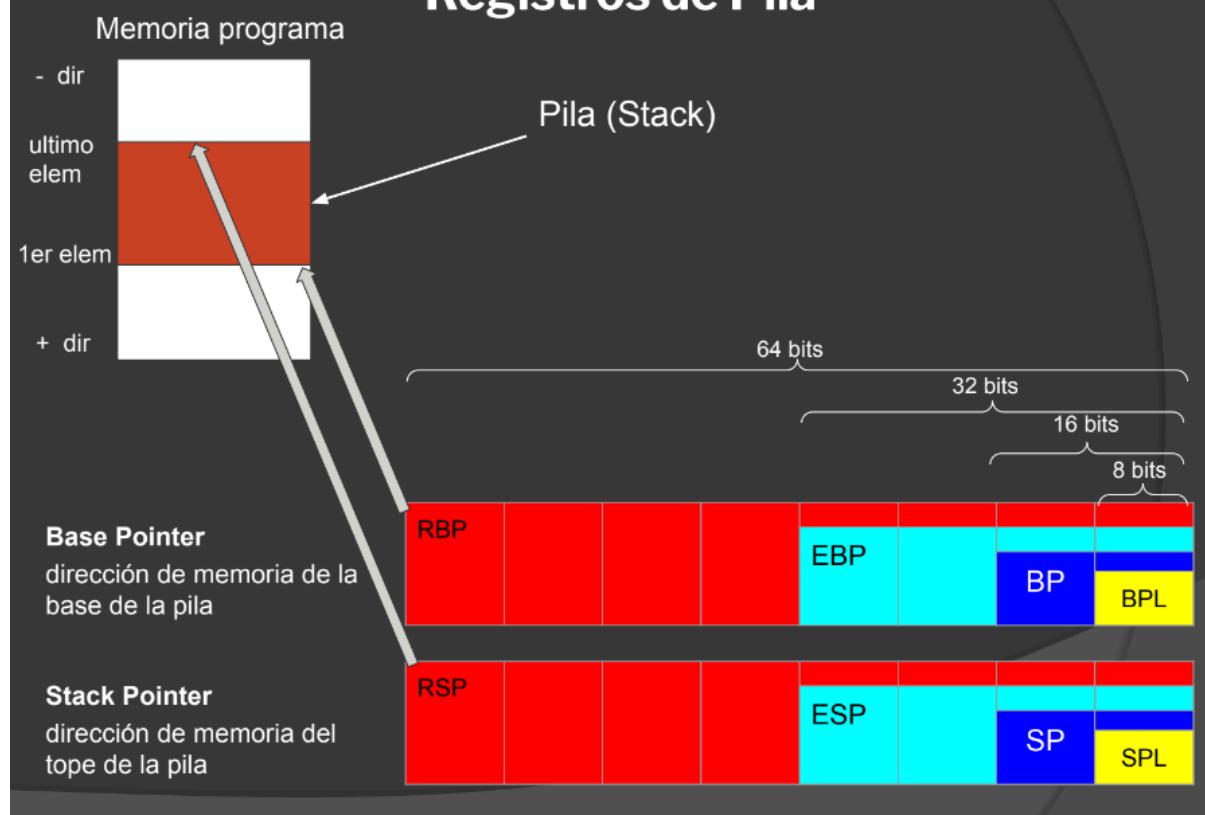


Registros índice:



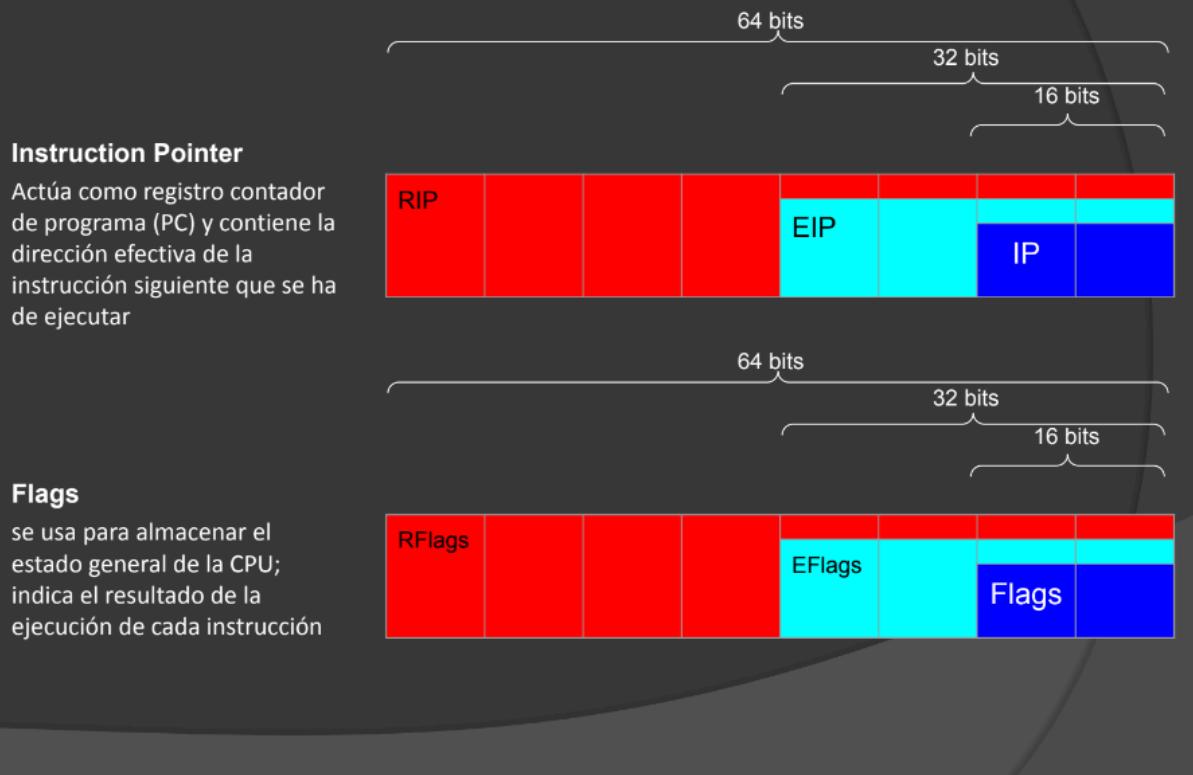
Registros de Pila:

# Registros de Pila



Registro de Instrucción y Control:

# Registros de Instrucción y Control



## Direccionamiento

Modos de direccionamiento:

- Implícito: El dato está implícito en el código de operación (Ej. CBW)
- Registro: El dato está en un registro (Ej. MOV RAX,5)
- Inmediato: El dato está dentro de la instrucción (Ej. MOV RAX,5)
- Directo: El dato está en memoria referenciado por el nombre de un campo (Ej. MOV RAX, [VARIABLE] o MOV RAX, [VARIABLE + 2])
- Registro Indirecto: El dato está en memoria apuntado por un registro base o índice (Ej. MOV EAX, [EBX] o MOV EAX, [ESI])
- Registro Relativo: El dato está en memoria apuntado por un registro base o índice más un desplazamiento (Ej. MOV RAX, [RBX + 4], MOV RAX, [VECTOR + RBX], MOV [RDI + 3], RAX)
- Base + Índice: El dato está en memoria apuntado por un registro más un registro índice. (Ej. MOV [RBX + RDI], CL)
- Base Relativo + Índice: El dato está en memoria apuntado por un registro base más un registro índice más un desplazamiento (Ej. MOV RAX, [RBX + RDI + 4] o MOV RAX, [VECTOR+RBX+RDI]])

## Tipos de dato

- Numérico Entero: Binario de punto fijo con Signo
- Numérico Decimal: Binario de punto Flotante IEEE
- Caracteres: ASCII

## Memoria

- Celda de Memoria: 1 Byte
- Palabra: 2 Bytes
- Doble Palabra: 4 Bytes
- Cuádruple Palabra: 8 bytes

## Endiannes

Definición: Es el método aplicado para almacenar datos mayores a un byte en una computadora respecto a la dirección que se le asigna a cada uno de ellos en la memoria

Existen 2 métodos:

- Big-Endian: determina que el orden en la memoria coincide con el orden lógico del dato (“el dato final en la mayor dirección”)
- Little Endian: es a la inversa, el dato inicial para la lógica se coloca en la mayor dirección y el dato final en la menor.

## Endiannes - Ejemplos

- **Caso 1:** Definición de un área de memoria con contenido inicial definido en formato carácter

```
msg    db  'HOLA'
```

48	4F	4C	41
1A1	1A2	1A3	1A4

Aquí la posición de memoria que toma cada carácter recibido es la que por intuición uno asume, o sea, la letra ‘H’ en la dirección menor

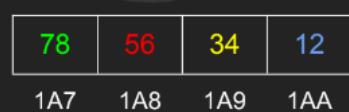
## Endiannes - Ejemplos

- ◎ **Caso 2:** Definición de un área de memoria con contenido inicial definido en formato numérico

```
num      dw 4666 ; es 123A en hexa
```



```
num2     dd 12345678h
```



Aquí es donde se observa la ubicación de los bytes con el método Little-Endian, el byte menos significativo se ubica en la dirección de memoria menor que el byte más significativo

19

## Endiannes - Ejemplos

- ◎ **Caso 3:** Se ejecuta una copia de memoria a registro

```
mov      AX, [msg]
```



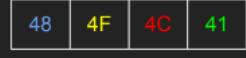
```
mov      BX, [num1]
```



```
mov      ECX, [msg]
```



```
mov      EDX, [num2]
```



La parte alta del registro contiene el byte de orden superior de memoria, y la parte baja del registro contiene el byte de orden inferior.

## Ensamblador NASM (Netwide Assembler)

### **Directivas al ensamblador / Pseudo-instrucciones**

Son instrucciones para que el ensamblador tome alguna acción durante el proceso de ensamblado. No se traducen a código máquina.

- section: indica el comienzo de un segmento
- global: indica que una etiqueta declarada en el programa es visible para un programa externo
- extern: indica que una etiqueta usada en el programa pertenece a un programa externo (donde habrá sido declarada global)
- db, dw, dd, dq, dt: sirven para definir áreas de memoria (variables) con contenido inicial
- resb, resw, resid, resq, rest: sirven para definir áreas de memoria (variables) sin contenido inicial
- times: repite una definición la cantidad de veces que se indica
- %macro %endmacro: indican el inicio y final de un bloque para definir una macro
- %include: permite incluir el contenido de un archivo

### **Estructura de un programa**

```
global main

section .data
;variables con contenido inicial

section .bss ;block starting symbol
;variables sin contenido inicial

section .text
;instrucciones
main:

        ret
```

### **Definición y reserva de campos en memoria**

Con contenido inicial (en *section .data*)

- db define byte (1 byte)
- dw define word (2 bytes)
- dd define double (4 bytes)
- dq define quad (8 bytes)
- dt define ten (10 bytes)

Sin contenido inicial (*section .bss*)

- resb reserve byte (1 byte)
- resw reserve word (2 bytes)
- resd reserve double (4 bytes)
- resq reserve quad (8 bytes)
- rest reserve ten (10 bytes)

Definición	Bytes reservados
campo1 resb 1	1
campo2 resb 2	2
campo3 resw 1	2
campo4 resw 2	4
campo5 resd 1	4
campo6 resd 2	8
campo7 resq 1	8
campo8 resq 2	16
vector1 times 2 resb 2	4
vector2 times 3 resw 2	12

Definicion	Bytes reservados	Contenido memoria
decimal1 db 11	1	0B
decimal2 dw -11	2	F5 FF
decimal3 dd 12345	4	39 30 00 00
decimal4 dq -1	8	FF FF FF FF FF FF FF FF
hexa1 db -0Bh	1	F5
hexa2 dw 0Ch	2	0C 00
hexa3 dd FFFFh	4	FF FF 00 00
hexa4 dq 96B43Fh	8	3F B4 96 00 00 00 00 00

Definicion		Bytes reservados	Contenido memoria
octal1	db 13o	1	0B
octal2	dw 71o	2	39 00
octal3	dd 10o	4	08 00 00 00
binario1	db 1011b	1	0B
binario2	dw 1011b	2	0B 00
binario3	dd -1000b	4	F8 FF FF FF
truncado	db 2571 ;0A0Bh	1	0B
noTruncado	dw 2571 ;0A0Bh	2	0B 0A

Definicion		Bytes reservados	Contenido memoria
letra	db 'A'	1	41
letra2	dw 'A'	2	41 20
letra3	db 'a'	1	61
cadena	db 'hola'	4	68 6F 6C 61
cadena2	dw 'ola'	4	6F 6C 61 20
numero	db '12'	2	31 32
vector1	times 3 db 'A'	3	41 41 41
vector2	times 3 db 'A',0	6	41 00 41 00 41 00
registro	times 0 db 'A'	0	n/a

## Macros

Son secuencias de instrucciones asignadas a un nombre que pueden usarse en cualquier parte del programa

ejemplo:

```
%macro macro_name number_of_params
<macro body>
%endmacro
```

## Sin parámetros

```
1 %macro mPuts 0
2     sub      rsp,8
3     call     puts
4     add      rsp,8
5 %endmacro
6 global main
7 extern puts
8
9 section .data
10    mensaje db "Hola mundo!",0
11    mensaje2 db "Chau!",0
12
13 section .text
14 main:
15     mov     rdi,mensaje
16     mPuts
17     mov     rdi,mensaje2
18     mPuts
19     ret
```

## Con 1 parámetro

```
1 %macro mPuts 1
2     mov     rdi,%1
3     sub      rsp,8
4     call     puts
5     add      rsp,8
6 %endmacro
7 global main
8 extern puts
9
10 section .data
11    mensaje db "Hola mundo!",0
12    mensaje2 db "Chau!",0
13
14 section .text
15 main:
16     mPuts    mensaje
17     mPuts    mensaje2
18     ret
```

## Inclusión de archivos

Mediante el uso de una directiva es posible incluir el contenido de un archivo del código

La sintaxis es:

```
%include "file_name"
```

```
misMacros.asm X
1 %macro mPuts 1
2     mov     rdi,%1
3     sub      rsp,8
4     call     puts
5     add      rsp,8
6 %endmacro
7 extern puts

test-include.asm X
1 %include "misMacrosL.asm"
2 global main
3 section .data
4     mensaje db "Hola mundo!!",0
5
6 section .text
7 main:
8     mPuts    mensaje
9     ret
```

PS C:\Users\Dario\Documents\Orga\include> nasm test-include.asm -fwin64  
PS C:\Users\Dario\Documents\Orga\include> gcc test-include.obj  
PS C:\Users\Dario\Documents\Orga\include> ./a.exe  
Hola mundo!!  
PS C:\Users\Dario\Documents\Orga\include>

## Funciones en C

### Manejo de parámetros

```
    result  funcName(p1, p2, p3, p4, p5, p6, p7,..., pn)

Linux:   rax          rcx rdx r8  r9  stack...
Windows: rax          rdi rsi rdx rcx r8  r9  stack...
```

### Ajuste para llamados a rutinas/funciones externas e internas

LINUX	WINDOWS
...	...
sub    rsp,8	sub    rsp,32
call   rutina_externa	call   rutina_externa
add    rsp,8	add    rsp,32
...	...
sub    rsp,8	call   rutina_interna
call   rutina_interna	
add    rsp,8	

## Calling convention

Caller saved registers

Windows

rax, rcx, rdx, r8-r11

Linux

rax, rcx, rdx, r8-r11, rdi, rsi

## Salida por pantalla

### Función PPUTS

Definición: imprime un string hasta que encuentra un 0(cero binario). Agrega el carácter de fin de línea a la salida

```
int puts(const char *str)

;LINUX
global main
extern puts

section .data
cadena db "Hola",0
section .text
main:
    mov    rdi,cadena
    sub    rsp,8
    call   puts
    add    rsp,8

;WINDOWS
global main
extern puts

section .data
cadena db "Hola",0
section .text
main:
    mov    rcx,cadena
    sub    rsp,32
    call   puts
    add    rsp,32
```

### Función PRINTF

Definición: convierte a string cada uno de los parámetros y los imprime con el formato indicado por pantalla

```

int printf(const char *format, arg-list)

;LINUX
global main
extern printf

section .data
    direccion db "Direccion: %s %lli",0
    calle     db "Paseo Colon",0
    nro      dq 955
section .text
main:
    mov    rdi,direccion
    mov    rsi,calle
    mov    rdx,[nro]
    sub    rsp,8
    call   printf
    add    rsp,8

;WINDOWS
global main
extern printf

section .data
    direccion db "Direccion: %s %lli",0
    calle     db "Paseo Colon",0
    nro      dq 955
section .text
main:
    mov    rcx,direccion
    mov    rdx,calle
    mov    r8,[nro]
    sub    rsp,32
    call   printf
    add    rsp,32

```

## Especificadores de Formato

		Linux	Windows
%hhi	número entero con signo base 10	8 bits	-
%hi	número entero con signo base 10	16 bits	16 bits
%i	número entero con signo base 10	32 bits	32 bits
%d	número entero con signo base 10	32 bits	32 bits
%li	número entero con signo base 10	64 bits	32 bits
%lli	número entero con signo base 10	-	64 bits
%o	número entero sin signo base 8	32 bits	32 bits
%x	número entero sin signo base 16	32 bits	32 bits
%c	caracter		
%s	string		

## Limpieza de pantalla

### Función system

Definición: ejecuta un comando del sistema operativo

Para limpiar la pantalla en unix el comando es “clear” y en windows “cls”

```
int system(const char *command)

;LINUX
global main
extern system
section .data
    cmd_clear db "clear",0
section .text
main:
    mov    rdi,cmd_clear
    sub    rsp,8
    call   system
    add    rsp,8

;WINDOWS
global main
extern system
section .data
    cmd_cls db "cls",0
section .text
main:
    mov    rdi,cmd_cls
    sub    rsp,32
    call   system ;NO FUNCIONA
    add    rsp,32
```

NO FUNCIONA

## Ingreso por teclado

### Función GETS

Definición: lee una serie de caracteres ingresados por teclado hasta que se presione “enter” y los almacena en el campo en memoria indicado por parámetro. Agrega un 0 binario al final.

```
char *gets(char *buffer)

;LINUX
global main
extern gets

section .bss
    cadena resb 100
section .text
main:
    mov    rdi,cadena
    sub    rsp,8
    call   gets
    add    rsp,8

;WINDOWS
global main
extern gets

section .bss
    cadena resb 100
section .text
main:
    mov    rcx,cadena
    sub    rsp,32
    call   gets
    add    rsp,32
```

## Conversión de Formato (string a entero)

### Función sscanf

Definición: lee una serie de datos desde un string y, de ser posible, los guarda en el formato indicado para cada uno. Retorna la cantidad de datos que se convirtieron correctamente.

```
int sscanf(const char *buffer,const char *format, arg-list)

;LINUX
global main
extern sscanf

section .data
nroStr db "955",0
numFormat db "%li",0
section .bss
numero resq 1
section .text
main:
    mov rdi,nroStr
    mov rsi,numFormat
    mov rcx,numero
    sub rsp,8
    call sscanf
    add rsp,8

    cmp rax,1
    jl error

;WINDOWS
global main
extern sscanf

section .data
nroStr db "955",0
numFormat db "%lli",0
section .bss
numero resq 1
section .text
main:
    mov rcx,nroStr
    mov rdx,numFormat
    mov r8,numero
    sub rsp,32
    call sscanf
    add rsp,32

    cmp rax,1
    jl error
```

### Conversión de formato (entero a string)

#### Función sprintf

Convierte a string cada uno de los parámetros y los imprime con el formato indicado en un campo en memoria. Agrega un 0 binario al final del campo

```
int sprintf(char *str, const char *format, arg-list)

LINUX
extern sprintf

section .data
numero dw 185
format db "%i",0
section .bss
numero_str resb 10
section .text
main:
    mov rdi,numero_str
    mov rsi,format
    xor rdx,rdx
    mov dx,[numero]
    sub rsp,8
    call sprintf
    add rsp,8

WINDOWS
extern sprintf

section .data
numero dw 185
format db "%i",0
section .bss
numero_str resb 10
section .text
main:
    mov rcx,numero_str
    mov rdx,format
    xor r8,r8
    mov r8w,[numero]
    sub rsp,32
    call sprintf
    add rsp,32
```

### Instrucciones

## Transferencia y copia

MOV op1,op2

Función: copia el valor del 2do operando en el primer operando

Combinaciones	Ejemplos en NASM
MOV <reg>,<reg>	MOV AH,BL MOV AX,BX MOV ECX, EAX MOV RDX,RCX
MOV <reg>,<mem>	MOV CH,[VARIABLE_8] (*) MOV CX, [VARIABLE_16] (*) MOV ECX, [VARIABLE_32] (*) MOV RDX, [VARIABLE_64] (*)
MOV <reg>,<inm>	MOV DL,7o MOV CX,2450h MOV EAX,0h MOV RDX,28h
Combinaciones	Ejemplos en NASM
MOV <mem>,<reg>	MOV [VARIABLE_8],AH (*) MOV [VARIABLE_16],AX (*) MOV [VARIABLE_32],EAX (*) MOV [VARIABLE_64],RAX (*) <del>MOV VARIABLE_64,RAX</del>
MOV <long><mem>,<inm>	MOV byte[VARIABLE_8],2Ah MOV word[VARIABLE_16],7770 MOV dword[VARIABLE_32],1234 MOV qword[VARIABLE_64],1234 <del>MOV [VARIABLE_64],4321</del>

## Comparación

CMP op1, op2

Definición: compara el contenido del op1 contra el op2 mediante la resta entre los dos operando sin modificarlos

Combinaciones	Ejemplos en NASM
CMP <reg>,<reg>	CMP AH,BL CMP AX,BX CMP EAX,EBX CMP RCX,RAX
CMP <reg>,<mem>	CMP CH,[VARIABLE_8] (*) CMP CX,[VARIABLE_16] (*) CMP EAX,[VARIABLE_32] (*) CMP RBX,[VARIABLE_64] (*)
Combinaciones	Ejemplos en NASM
CMP <reg>,<inm>	CMP CH,10o CMP CX,2Ah CMP EAX,1000 CMP RBX,432h
CMP <mem>,<reg> <b>(*) Considera la long del REG</b>	<del>CMP VARIABLE,AX</del> CMP [VARIABLE_8],RAX (*) CMP [VARIABLE_8],AH CMP [VARIABLE_64],RCX
CMP <long><mem>,<inm>	<del>CMP VARIABLE,245h</del> CMP byte[VARIABLE_8],2Ah CMP word[VARIABLE_16],2Ah CMP dword[VARIABLE_32],2Ah CMP qword[VARIABLE_64],2Ah

## Saltos / Bifurcaciones

JMP op

Definición: bifurca a la dirección indicada del operando

Jcc op

Definición: bifurca a la dirección indicada del operando si se cumple la condición

Combinaciones	Ejemplos en NASM
JMP <etiqueta>	JMP finalizar
Jcc <etiqueta>	JE esIgual

y

### Condiciones generales

<b>JE</b>	op	-> por igual ( $ZF=1$ )
<b>JNE</b>	op	-> por no igual ( $ZF=0$ )
<b>JZ</b>	op	-> por igual a cero ( $ZF=1$ )
<b>JNZ</b>	op	-> por distinto a cero ( $ZF=0$ )
<b>JRCXZ</b>	op	-> por contenido de RCX igual cero
<b>JC</b>	op	-> por carry flag distinto a cero ( $CF=1$ )
<b>JO</b>	op	-> por overflow ( $OF=1$ )

### Condiciones con signo

<b>JG</b>	op	-> por mayor ( $ZF=0$ and $SF=OF$ )
<b>JGE</b>	op	-> por mayor o igual ( $SF=OF$ )
<b>JL</b>	op	-> por menor ( $SF<>OF$ )
<b>JLE</b>	op	-> por menor o igual ( $ZF=1$ or $SF<>OF$ )
<b>JNG</b>	op	-> por no mayor ( $ZF=1$ or $SF<>OF$ )
<b>JNGE</b>	op	-> por no mayor o igual ( $SF<>OF$ )
<b>JNL</b>	op	-> por no menor ( $SF=OF$ )
<b>JNLE</b>	op	-> por no menor o igual ( $ZF=0$ and $SF=OF$ )

Condiciones sin signo

<b>JA</b>	op	-> por mayor (CF=0 and ZF=0)
<b>JAE</b>	op	-> por mayor o igual (CF=0)
<b>JB</b>	op	-> por menor (CF=1)
<b>JBE</b>	op	-> por menor o igual (CF=1 or ZF=1).
<b>JNA</b>	op	-> por no mayor (CF=1 or ZF=1)
<b>JNAE</b>	op	-> por no mayor o igual (CF=1)
<b>JNB</b>	op	-> por no menor (CF=0)
<b>JNBE</b>	op	-> por no menor o igual CF=0 and ZF=0)

## Suma

ADD op1, op2

Combinaciones	Ejemplos en NASM
ADD <reg>,<reg>	ADD AH,BL ADD AX,BX ADD ECX,EAX ADD RDX,RCX
ADD <reg>,<mem>	ADD AL,[VARIABLE_8] (*) ADD BX,[VARIABLE_16] (*) ADD ECX,[VARIABLE_32] (*) ADD RDX,[VARIABLE_64] (*)

Combinaciones	Ejemplos en NASM
ADD <reg>,<inm>	ADD DL,10b ADD AX,10h ADD ECX,1234 ADD RCX,1234h
ADD <mem>,<reg> <b>(* Considera la long del REG)</b>	<del>ADD VARIABLE_16,AX</del> ADD [VARIABLE_8],AL ADD [VARIABLE_8],BX (*) ADD [VARIABLE_32],ECX ADD [VARIABLE_64],RDX
ADD <long><mem>,<inm>	<del>ADD VARIABLE_123</del> ADD byte[RDI],245h ADD word[VARIABLE_16],2Ah ADD dword[VARIABLE_32],123 ADD qword[VARIABLE_64],2Ah

### Resta

SUB op1, op2

Definición: resta los valores de los dos operando (binarios de punto fijo con signo) dejando el resultado en el primero

Combinaciones	Ejemplos en NASM
SUB <reg>,<reg>	SUB AH,BL SUB AX,BX SUB ECX,EAX SUB RDX,RBX
SUB <reg>,<mem>	SUB AL,[VARIABLE_8] (*) SUB BX,[VARIABLE_16] (*) SUB ECX,[VARIABLE_32] (*) SUB RDX,[VARIABLE_64] (*)

Combinaciones	Ejemplos en NASM
SUB <reg>,<inm>	SUB DL,10b SUB AX,10h SUB ECX,1234 SUB RDX,1234h
SUB <mem>,<reg> <b>(*) Considera la long del REG</b>	<del>SUB VARIABLE,AX</del> SUB [VARIABLE_8].AL SUB [VARIABLE_8],BX(*) SUB [VARIABLE_32],ECX SUB [VARIABLE_64],RDX
SUB <long><mem>,<inm>	<del>SUB VARIABLE,123</del> SUB byte[EDI],245h SUB word[VARIABLE],2Ah SUB dword[VARIABLE],123 SUB qword[VARIABLE],123h

### Incremento y decremento

INC op

Definición: suma uno al operando (binarios de punto fijo con signo)

DEC op

Definición: resta uno al operando (binarios de punto fijo con signo)

Combinaciones	Ejemplos en NASM
INC/DEC <reg>	INC/DEC BH <input type="checkbox"/> BH = BH + 1 / BH = BH - 1 INC/DEC CX <input type="checkbox"/> CX = CX + 1 / CX = CX - 1 INC/DEC EAX <input type="checkbox"/> EAX = EAX + 1 / EAX = EAX - 1 INC/DEC RDX <input type="checkbox"/> RDX = RDX + 1 / RDX = RDX - 1
INC/DEC <mem>	INC byte[VAR_8B] INC word[VAR_16B] INC dword[VAR_32B] <input type="checkbox"/> VAR_32B = VAR_32B +1 INC qword[VAR_64B]
	DEC byte[VAR_8B] DEC word[VAR_16B] <input type="checkbox"/> VAR_16B = VAR_16B - 1 DEC dword[VAR_32B] DEC qword[VAR_64B]

## Multiplicación - formato 1 operando

IMUL op

Si longitud de op es 8 bits: multiplica AL \* op y deja el resultado en AX

Si longitud de op es 16 bits: multiplica AX \* op y deja el resultado en DX:AX

Si longitud de op es 32 bits: multiplica EAX \* op y deja el resultado en EDX:EAX

Si longitud de op es 64 bits: multiplica RAX \* op y deja el resultado en RDX:RAX

Los operandos son interpretados como binario de punto fijo CON signo

MUL op

Igual que IMUL pero los operando son interpretados como binario de punto fijo SIN signo

Combinaciones	Ejemplos en NASM
MUL/IMUL <reg>	MUL/IMUL BH $\square$ AX = <b>(AL)</b> *(BH) MUL/IMUL BX $\square$ DX:AX = <b>(AX)</b> *(BX) MUL/IMUL EBX $\square$ EDX:EAX = <b>(EAX)</b> *(EBX) MUL/IMUL RCX $\square$ RDX:RAX = <b>(RAX)</b> *(RCX)
MUL/IMUL <mem>	MUL/IMUL <b>byte</b> [VAR_8] $\square$ AX = <b>(AL)</b> *(VAR_8) MUL/IMUL <b>word</b> [VAR_16] $\square$ DX:AX = <b>(AX)</b> *(VAR_16) MUL/IMUL <b>dword</b> [VAR_32] $\square$ EDX:EAX = <b>(EAX)</b> *(VAR_32) MUL/IMUL <b>qword</b> [VAR_64] $\square$ RDX:RAX = <b>(RAX)</b> *(VAR_64)

## Multiplicación - Formato 2 operandos

IMUL op1, op2

Multiplica el contenido de los operandos y almacena el resultado en el primero.

Ambos operandos deben tener la misma longitud

Si el resultado no entra en el operando 1, se trunca

Los operando son interpretados como binario de punto fijo CON signo

MUL op1, op2

Igual que IMUL pero los operando son interpretados como binario de punto fijo SIN signo

Combinaciones	Ejemplos en NASM
IMUL <reg>, <inm>	IMUL CX,4 $\square$ CX = (CX)*4
IMUL <reg>,<reg>	IMUL EAX,EBX $\square$ EAX = (EAX)*(EBX)
IMUL <reg>,<log><mem>	IMUL RBX, <b>qword</b> [VAR_64] $\square$ RBX = (RBX)*(VAR_64)

**IMUL op1, op2, op3**

Multiplica el contenido de los operandos 2 y 3 y almacena el resultado en el operando 1

El operando 1 es siempre un registro y el operando 3 siempre un valor inmediato

Si el resultado no entra en el operando 1, se trunca

Los operandos son interpretados como binario de punto fijo CON signo

**MUL op1, op2, op3**

Igual que IMUL pero los operandos son interpretados como binario de punto fijo SIN signo

Combinaciones	Ejemplos en NASM
IMUL <reg>, <reg>,<inm>	IMUL CX,BX,10h □ $CX = (BX) * 10h$
IMUL <reg>,<mem>,<inm>	IMUL RBX,qword[VAR_64],4 □ $RBX = (VAR\_64) * 4$

## División

**IDIV op**

Si longitud de op es 8 bits: AX / op, resto en AH y cociente en AL

Si longitud de op es 16 bits: DX:AX / op, resto en DX y cociente en AX

Si longitud de op es 64 bits: RDX: RAX / op resto en RDX y cociente en RAX

Los operandos son interpretados como binario de punto fijo CON signo

**DIV op**

Igual que IDIV pero los operandos son interpretados como binario de punto fijo SIN signo

Combinaciones	Ejemplos en NASM
DIV/IDIV <reg>	<p>DIV/IDIV BX □ DX:AX / BX</p> <ul style="list-style-type: none"> <li>• Cociente = AX</li> <li>• Resto = DX</li> </ul> <p>DIV/IDIV EBX □ EDX:EAX / EBX</p> <ul style="list-style-type: none"> <li>• Cociente = EAX</li> <li>• Resto = EDX</li> </ul> <p>DIV/IDIV RCX □ RDX:RAX / RCX</p> <ul style="list-style-type: none"> <li>• Cociente = RAX</li> <li>• Resto = RDX</li> </ul>
DIV/IDIV <long><mem>	<p>DIV/IDIV byte[VAR_8] □ AX / [VAR_8]</p> <ul style="list-style-type: none"> <li>• Cociente = AL</li> <li>• Resto = AH</li> </ul> <p>DIV/IDIV dword[VAR_32] □ EDX:EAX / [VAR_32]</p> <ul style="list-style-type: none"> <li>• Cociente = EAX</li> <li>• Resto = EDX</li> </ul> <p>DIV/IDIV qword[VAR_64] □ RDX:RAX / [VAR_64]</p> <ul style="list-style-type: none"> <li>• Cociente = RAX</li> <li>• Resto = RDX</li> </ul>

## Conversión

**CBW:** convierte el byte almacenado en AL a una word en AX

**CWD:** convierte la word almacenada en AX a una double-word en DX:AX

**CWDE:** convierte la word almacenada en AX a una double-word en EAX

**CDQE:** convierte la doble-word almacenada en EAX a una quad-word en RAX

Expandan en signo del operando

	Ejemplo en NASM
MOV AL,byte[VAR_8B] CBW CWDE CDQE	<p>VAR_8B db 9Bh □ AL = 9B</p> <p>AX = FF9B</p> <p>EAX = FFFFFFFF9B</p> <p>RAX = FFFFFFFFFFFFFF9B</p>
MOV AX,word[VAR_16B] CWD CWDE CDQE	<p>VAR_16B dw FF9Bh □ AX = FF9B</p> <p>DX:AX = FFFFh:FF9B</p> <p>EAX = FFFFFFFF9B</p> <p>RAX = FFFFFFFFFFFFFF9B</p>

## Complemento

NEG op

Definición: realiza el complemento a 2 del operando, es decir, le cambia el signo

Combinaciones	Ejemplos en NASM
NEG <reg>	NEG BH NEG AX NEG ECX NEG RDX
NEG <mem>	NEG byte[VAR_8] NEG word[VAR_16] NEG dword[VAR_32] NEG qword[VAR_64]

## Loop

LOOP op

Definición: resta 1 al contenido del registro RCX y si el resultado es 0, bifurca al punto indicado por el operando, sino continua la ejecución en la instrucción siguiente.

El desplazamiento al punto indicado debe estar en un rango entre -128 a 127 bytes (*near jump*)

```
    mov    rcx, 5
inicio:
    . . .
    . . .
loop  inicio
    . . .
```

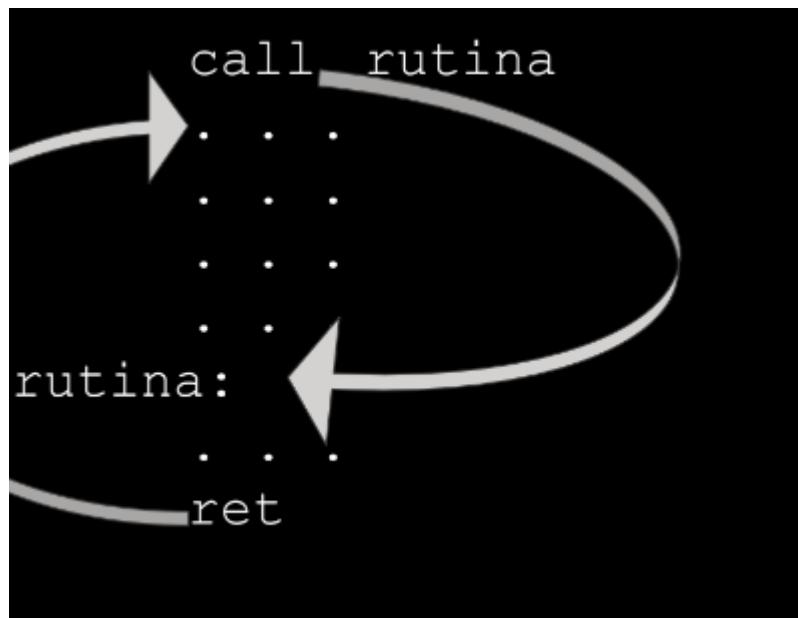
## Llamada y retorno de procedimiento

CALL op

Definición: almacena en la pila la dirección de la instrucción siguiente a la call y bifurca al punto indicado por el operando

RET

Definición: toma el elemento del tope de la pila que debe ser una dirección de memoria (generalmente cargada por una call) y bifurca hacia la misma



## Ejemplos rutinas externas

The screenshot shows two assembly files side-by-side:

```
moduleHello.asm:
1 global sayHello
2 extern puts
3 section .data
4 mensaje db "Hello",0
5 section .text
6 sayHello:
7     mov rdi,mensaje
8     sub rsp,8
9     call puts
10    add rsp,8
11    ret

moduleBye.asm:
1 global sayGoodbye
2 extern puts
3 section .data
4 mensaje db "Goodbye",0
5 section .text
6 sayGoodbye:
7     mov rdi,mensaje
8     sub rsp,8
9     call puts
10    add rsp,8
11    ret
```

Below the files, a terminal window shows the compilation command:

```
dario@dario-VirtualBox:/media/sf_Orga/rutinas externas$ nasm moduleHello.asm -f elf64
dario@dario-VirtualBox:/media/sf_Orga/rutinas externas$ nasm moduleBye.asm -f elf64
dario@dario-VirtualBox:/media/sf_Orga/rutinas externas$
```

A file explorer window at the bottom shows the generated object files:

- moduleBye.o
- moduleHello.o
- main\_external\_routines.asm
- moduleBye.asm
- moduleHello.asm

```

main_external_routines.asm X
1 global main
2 extern sayHello
3 extern sayGoodbye
4 section .data
5 section .text
6 main:
7 sub    rsp,8
8 call   sayHello
9 add    rsp,8
10
11 sub    rsp,8
12 call   sayGoodbye
13 add    rsp,8
14 ret

```

```

dario@dario-VirtualBox:/media/sf_Orga/rutinas externas$ nasm main_external_routines.asm -f elf64
dario@dario-VirtualBox:/media/sf_Orga/rutinas externas$ gcc main_external_routines.o moduleHello.o moduleBye.o -no-pie
dario@dario-VirtualBox:/media/sf_Orga/rutinas externas$ ./a.out
Hello
Goodbye
dario@dario-VirtualBox:/media/sf_Orga/rutinas externas$

```

## AND

AND op1, op2

Definición: Ejecuta la operación lógica AND entre el operando 1 y 2 dejando el resultado en el 1

Combinaciones	Ejemplos en NASM
AND <reg>,<reg>	AND AH,BL / AND AX,BX / AND ECX,EAX
AND <reg>,<long><mem>	AND AL, <b>byte</b> [VAR_8B] AND ECX, <b>dword</b> [VAR_32B]
AND <reg>,<inm>	AND CH,00h / AND CX,250h / AND CX,3456
AND <long><mem>,<inm>	AND <b>word</b> [VAR_16B], 1010b AND <b>dword</b> [VAR_32B],FFCC00AAh
AND <long><mem>,<reg>	AND <b>byte</b> [VAR_8B],BL AND <b>dword</b> [VAR_32B],EAX

## OR

OR op1, op2

Definición: ejecuta la operación lógica OR entre el operando 1 y 2 dejando el resultado en el 1

Combinaciones	Ejemplos en NASM
OR <reg>,<reg>	OR AH,BL / OR AX,BX / OR ECX,EAX
OR <reg>,<long><mem>	OR AL, <b>byte</b> [VAR_8B] OR ECX, <b>dword</b> [VAR_32B]
OR <reg>,<inm>	OR CH,00h / OR CX,250h / OR CX,3456
OR <long><mem>,<inm>	OR <b>word</b> [VAR_16B], 1010b OR <b>dword</b> [VAR_32B],FFCC00AAh
OR <long><mem>,<reg>	OR <b>byte</b> [VAR_8B],BL OR <b>dword</b> [VAR_32B],EAX

### Exclusive or

XOR op1, op2

Definición: ejecuta la operación lógica EXCLUSIVE OR entre el operando 1 y 2 dejando el resultado en el 1

Combinaciones	Ejemplos en NASM
XOR <reg>,<reg>	XOR AH,BL / XOR AX,BX / XOR ECX,EAX
XOR <reg>,<long><mem>	XOR AL, <b>byte</b> [VAR_8B] XOR ECX, <b>dword</b> [VAR_32B]
XOR <reg>,<inm>	XOR CH,00h / XOR CX,250h / XOR CX,3456
XOR <long><mem>,<inm>	XOR <b>word</b> [VAR_16B], 1010b XOR <b>dword</b> [VAR_32B],FFCC00AAh
XOR <long><mem>,<reg>	XOR <b>byte</b> [VAR_8B],BL XOR <b>dword</b> [VAR_32B],EAX

### NOT

NOT op

Definición: Ejecuta la operación lógica NOT en el operando.

Combinaciones	Ejemplos en NASM
NOT <reg>	NOT AH NOT BX NOT ECX
NOT <mem>	NOT byte[VAR_8B] NOT word[VAR_16B] NOT dword[VAR_32B]

### Transferencia y copia

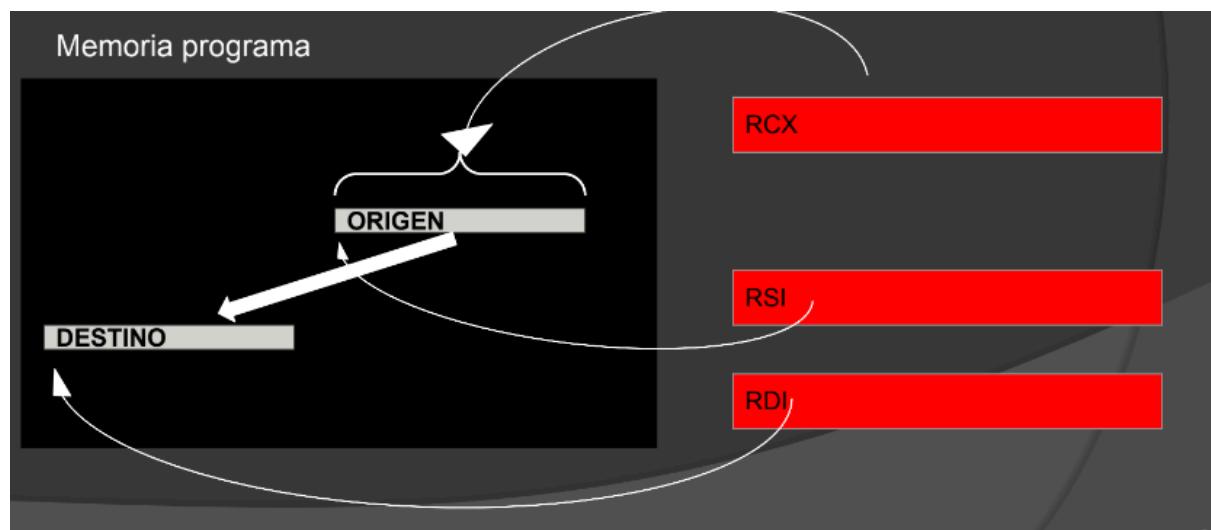
LEA op1, op2

Copia en el operando 1 (un registro) y la dirección de memoria del operando 2

Combinaciones	Ejemplos en NASM
LEA <reg>,<mem>	LEA RAX,[VARIABLE]  es equivalente a hacer MOV RAX,VARIABLE ;notar q aca NO hay corchetes

MOVSB

Definición: copia el contenido de memoria apuntando por RSI (origen/source) al apuntado por RDI (destino/destination). Copia tantos bytes como los indicados en el registro RCX



```
    . . .
    MOV    RCX, 4
    LEA    RSI, [MSGORI]
    LEA    RDI, [MSGDES]
REP    MOVSB
    . . .
```

### Comparación de strings

CMPSB

Definición: compara el contenido de memoria apuntado por RSI (origen/source) con el apuntado por RDI (destino/destination)

Compara tantos bytes como los indicados en el registro RCX

```
    . . .
    MOV    RCX, 4
    LEA    RSI, [MSG1]
    LEA    RDI, [MSG2]
REPE    CMPSB
    JE     IGUALES
    . . .
```

### Manejo de la pila (stack)

PUSH up

Inserta el operando (de 64 bits) en la pila. Decrementa (resta 1) el contenido del registro RSP

POP op

Definición: Elimina el último elemento insertado en la pila (de 64 bits) y lo copia en el operando. Incrementa (suma 1) al contenido del registro RSP

Combinaciones	Ejemplos en NASM
PUSH / POP <reg>	PUSH / POP RDX
PUSH / POP qword<mem>	PUSH / POP qword[VARIABLE]

Manejo de Pila:

- La pila es usada para almacenar transitoriamente datos, direcciones de retornos de subrutinas o pasar parámetros a funciones o subrutinas
- El último que entra es el primero que sale LIFO
- PUSH sirve para poner el dato en la pila mientras que POP se usa para recuperar el dato
- El stack Pointer (SP) se incrementa con el POP y se decrementa con el PUSH
- El operando puede ser un registro o posición de memoria (ambos 64bits)

Combinaciones	Ejemplos en NASM
PUSH / POP <reg>	PUSH / POP RDX
PUSH / POP qword<mem>	PUSH / POP qword[VARIABLE]

## Tablas

Tira de bytes en memoria destinada a usar como una estructura de Vector o Matriz

```
tabla    times 40 resb 1
vector   times 10 resw 1
matriz   times 25 db "*"
```

### **Posicionamiento en el elemento i de un vector**

(i - 1) \* longitudElemento

### **Posicionamiento en el elemento i,j de una matriz**

(i-1)\*longitudFila + (j-1) \* longitudElemento

longitudFila = longitudElemento\*cantidadColumnas

Ejemplo:

## ◦ Tablas (Cont.)

Dada una matriz (4 columnas x 3 filas) del tipo doble → se pide cargar un valor en el elemento de la fila 2 y columna 3

```
posx      dd  02
posy      dd  03
longfil   dd  16
longele   dd  4
matriz    times 12  resd   1
. . . . .
mov rbx,matriz ;pongo el pto al inicio de la matriz
mov rax,dword[posx] ;guardo el valor de la fila
sub rax,1
imul    dword[longfil] ;me desplazo en la fila
add rcx,rax
mov rax,dword[posy] ;guardo el valor de la fila
sub rax,1
imul    dword[longele] ;me desplazo en la columna
add rcx,rax      ;sumo los desplazamientos
add rbx,rcx      ;me posicione en la matriz

mov dword[rbx],XXX ;muevo el valor
```

### **Validación**

Código destinado a verificar que los datos de un programa provenientes del exterior (teclado, archivos) cumplen las condiciones esperadas y/o necesarias.

Clasificación según el contenido del dato:

- Lógica: que se adecúe al significado lógico

Ej: Dia de la semana: lunes, martes, miércoles, etc o Respuesta “S” o “N”

- Física: que el dato esté en un formato particular  
Ej: Campo en formato empaquetado o fecha en formato DD/MM/AA

Clasificación según el mecanismo aplicado

- Por valor: comparar contra uno o varios valores válidos
- Por Rango: comparar que esté dentro de un rango continuo válido
- Por tabla: buscar que exista en una tabla de valores válidos