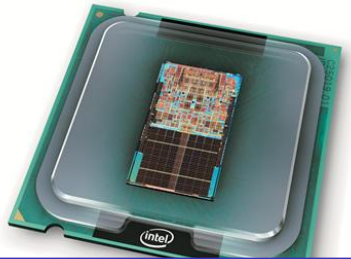



# Procesadores IA-32 e Intel©64

Alejandro Furfaro

21 de agosto de 2025

# Contenido

# Primeras conclusiones de la iAPx86

# Primeras conclusiones de la iAPx86

- Palabra clave: Compatibilidad.

# Primeras conclusiones de la iAPx86

- Palabra clave: Compatibilidad.
- Fue la llave el enorme éxito de esta arquitectura, pero también la ha condicionado en determinados períodos de su evolución.

# Primeras conclusiones de la iAPx86

- Palabra clave: Compatibilidad.
- Fue la llave el enorme éxito de esta arquitectura, pero también la ha condicionado en determinados períodos de su evolución.
- En los años transcurridos desde 1978, han habido períodos en los que otras arquitecturas le han sacado ventaja y le ha costado mas esfuerzo poder competir.

# Primeras conclusiones de la iAPx86

- Palabra clave: Compatibilidad.
- Fue la llave el enorme éxito de esta arquitectura, pero también la ha condicionado en determinados períodos de su evolución.
- En los años transcurridos desde 1978, han habido períodos en los que otras arquitecturas le han sacado ventaja y le ha costado mas esfuerzo poder competir.
- iAPx86 no puede ser analizada sin tener en cuenta estos factores “históricos”.

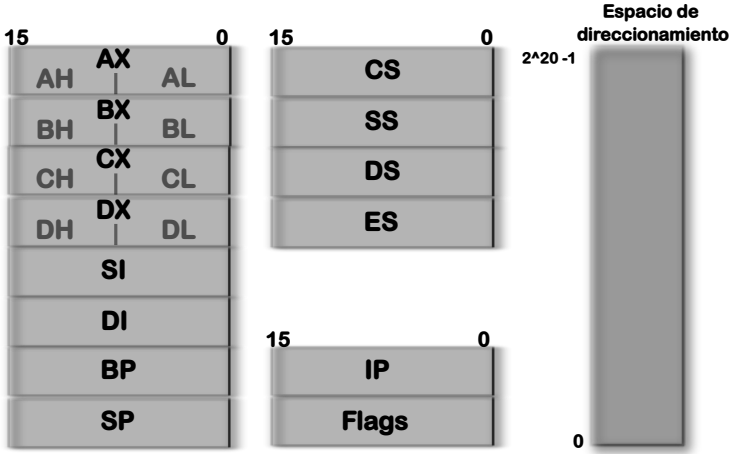
# Primeras conclusiones de la iAPx86

- Palabra clave: Compatibilidad.
- Fue la llave el enorme éxito de esta arquitectura, pero también la ha condicionado en determinados períodos de su evolución.
- En los años transcurridos desde 1978, han habido períodos en los que otras arquitecturas le han sacado ventaja y le ha costado mas esfuerzo poder competir.
- iAPx86 no puede ser analizada sin tener en cuenta estos factores “históricos”.
- El compromiso de Compatibilidad adoptado en 1978 significa que las decisiones de fondo adoptadas inicialmente en el diseño de la arquitectura, inevitablemente condicionarán lo que puede o no puede hacerse a futuro.

# Primeras conclusiones de la iAPx86

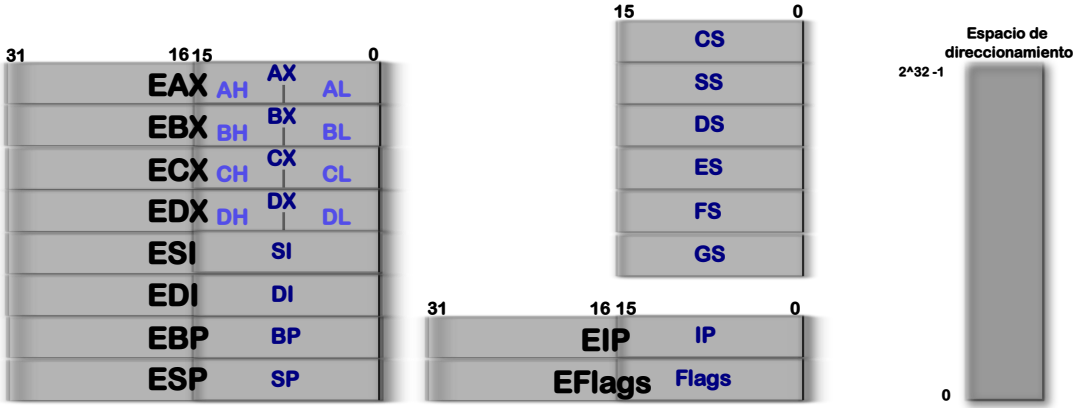
- Palabra clave: Compatibilidad.
- Fue la llave el enorme éxito de esta arquitectura, pero también la ha condicionado en determinados períodos de su evolución.
- En los años transcurridos desde 1978, han habido períodos en los que otras arquitecturas le han sacado ventaja y le ha costado mas esfuerzo poder competir.
- iAPx86 no puede ser analizada sin tener en cuenta estos factores “históricos”.
- El compromiso de Compatibilidad adoptado en 1978 significa que las decisiones de fondo adoptadas inicialmente en el diseño de la arquitectura, inevitablemente condicionarán lo que puede o no puede hacerse a futuro.
- La administración de memoria por segmentación como veremos es un clarísimo ejemplo de ello.

# Registros y espacio de direccionamiento



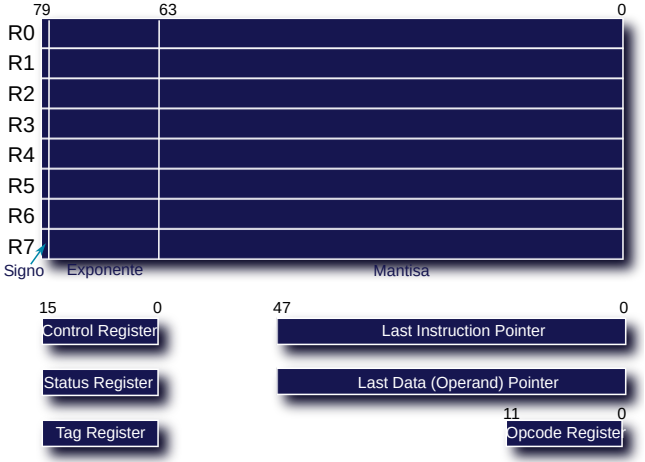

Entorno Básico de ejecución en 16 bits.

# Arquitectura de 32 bits compatible



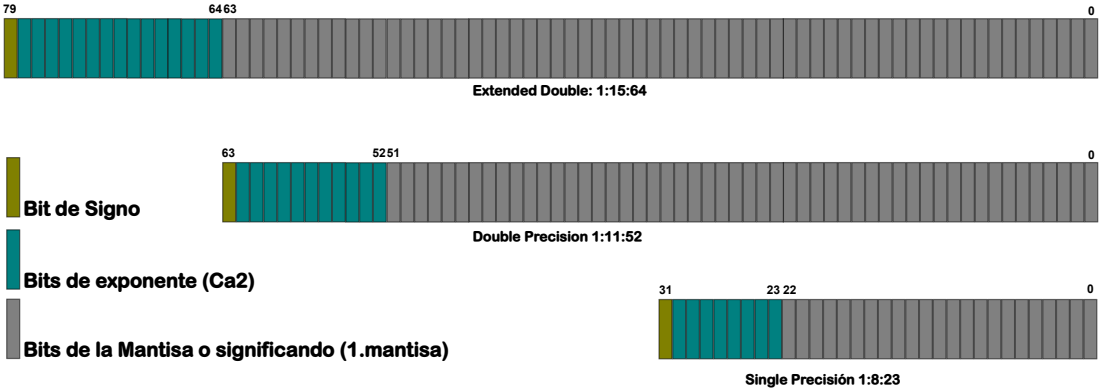

Entorno Básico de ejecución del 80386.

# Floating Point Unit



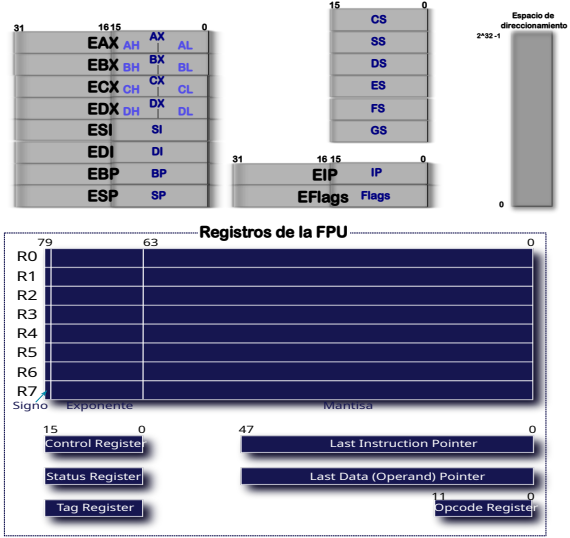

Entorno Básico de ejecución de la FPU.

# Floating Point Unit

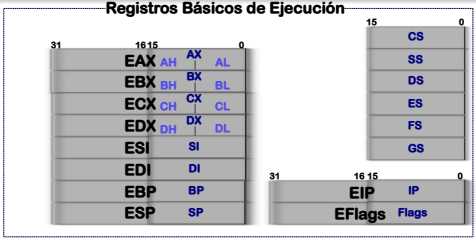


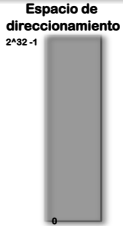

Formatos de los Datos de Punto Flotante de la FPU

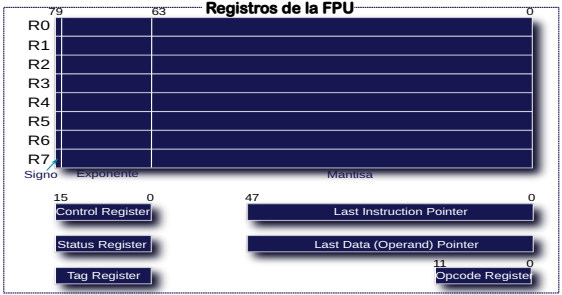
# Floating Point Unit on board



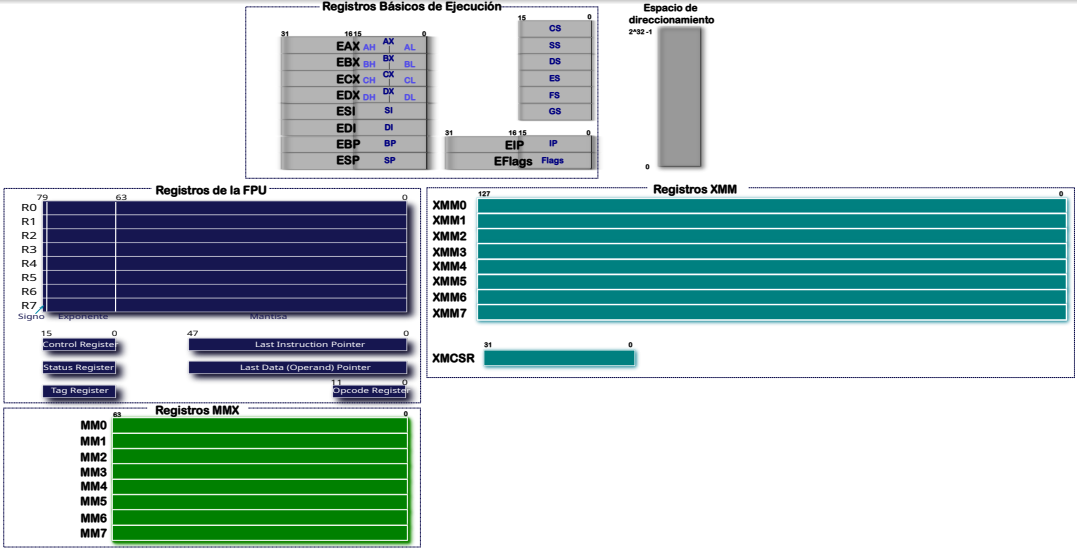
# Extensiones MMX



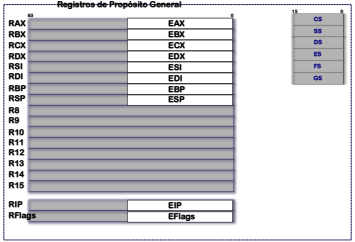



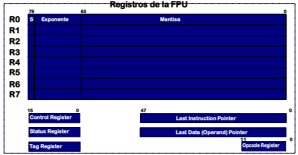

# Llega el Pentium III



# Extensiones de 64 bits









# Extensiones de 64 bits

En este modo los registros de Propósito General se extienden a 64 bits y aparecen 8 registros de 64 bits adicionales.

El procesador a pesar de estar en modo 64 bits puede acceder a diferentes tamaños de operador.

Para ello es necesario que utilice el prefijo REX, antecediendo a cada instrucción que requiera este tipo de operandos.

La Figura muestra los registros disponibles para el acceso a operandos de 32 bits cuando se utiliza el prefijo REX.

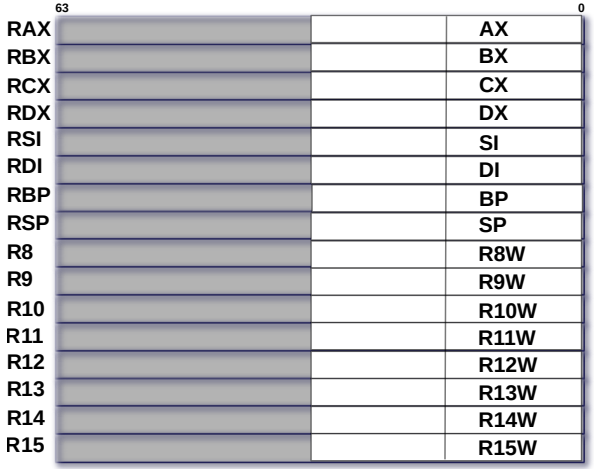
**Registros de Propósito General**

| 63 | | 0 |
|---|---|---|
| RAX | | EAX |
| RBX | | EBX |
| RCX | | ECX |
| RDX | | EDX |
| RSI | | ESI |
| RDI | | EDI |
| RBP | | EBP |
| RSP | | ESP |
| R8 | | R8D |
| R9 | | R9D |
| R10 | | R10D |
| R11 | | R11D |
| R12 | | R12D |
| R13 | | R13D |
| R14 | | R14D |
| R15 | | R15D |

Registros de Propósito general para operandos Double word, utilizando prefijo REX.

# Extensiones de 64 bits

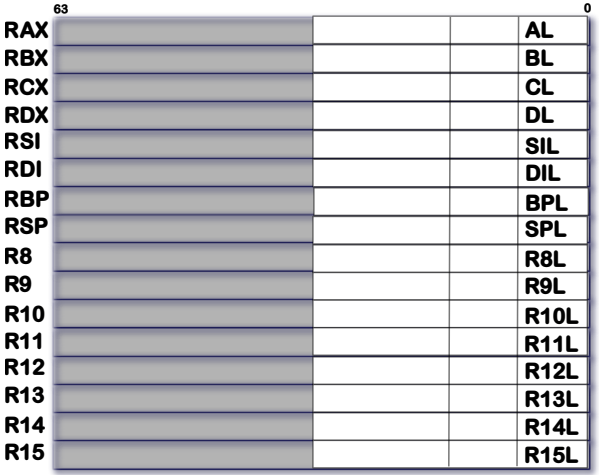
**Registros de Propósito General**




Intel® 64 : Registros de Propósito general para operandos word, utilizando prefijo REX

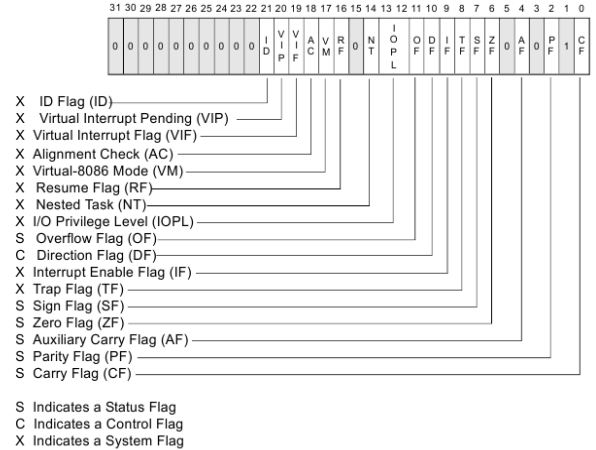
# Extensiones de 64 bits

**Registros de Propósito General**




Intel® 64 : Registros de Propósito general para operandos byte, utilizando prefijo REX

# Flags, EFLAGS, RFLAGS




IA-32: Registro EFLAGS. Cortesía Intel®

# Como Obtener los Operandos

Como reglas generales, de acuerdo a la documentación disponible, las instrucciones de estos procesadores pueden obtener los operandos desde:

- La instrucción en si misma (es decir que el operando está implícito en la instrucción).
- Un registro
- Una posición de memoria
- Un port de E/S

# Como Almacenar los resultados

Del mismo modo el resultado de una instrucción puede tener como destino para su almacenamiento:

- Un registro
- Una posición de memoria
- Un port de E/S

# ¿Que significa Direccionamiento Implícito?

Se trata de instrucciones en las cuales el código de operación es suficiente como para establecer que operación realizar, y cual es el operando.

Son ejemplos de este Modo, las instrucciones que operan sobre los Flags, como por ejemplo:

- CLC: Clear Carry. El operando (el Flag CF), está implícito en la operación. Lo mismo ocurre con STC (Set Carry), CMC (Complement Carry)
- CLD (Clear Direction Flag), STD (Set Direction Flag)
- CLI y STI para limpiar y setear el flag de Interrupciones (IF)
- HLT, o NOP

. . . entre otros.

# ¿Que significa direccionar en Modo Inmediato?

En este modo, el operando fuente viene dentro del código de la instrucción, con lo cual no es necesario ir a buscarlo a memoria luego de decodificada la instrucción.

```
1  ;////////////////////////////////////////////////////////
2  ;ascii recibe en al un byte decimal no empaquetado
3  ;y retorna su ascii en el mismo registro
4  ;////////////////////////////////////////////////////////
5  ascii:
6          add     al,'0'
7          cmp     al,'9'
8          jle     listo
9          add     al,'A'-'9'-1
10 listo:  ret
```

# ¿Que significa direccionar a Registro?

Todos los operandos involucrados son Registros del procesador. No importa si hay uno o dos operandos, son todos Registros.

- Registros de Propósito General de 64, 32, 16 u 8 bits (arquitecturas IA-32 e Intel® 64 ).
- Registros de segmento.
- EFlags (o RFlags).
- Registros de la FPU.
- MM0 a MM7.
- XMM0 a XMM7 (IA-32 )o XMMM0 a XMM15 (Intel® 64 ).
- Registros de Control.
- Registros de Debug.
- MSR's (Model Specific Registers).

```
1 inc      rdx           ; Incrementa en contenido del registro RDX
2 mov      eax,ebp       ; Mueve al registro EAX el contenido del EBP
3 mov      cr0,eax       ; Mueve al registro cr0 el contenido del EAX
4 fmul     st(0),st(3)   ; ST(0) = ST(0) * ST(3)
5 sqrtpd xmm2,xmm6       ; Raíz cuadrada de dos double precision FP empaquetados
6                        ; en xmm6. Resultados en xmm2.
```

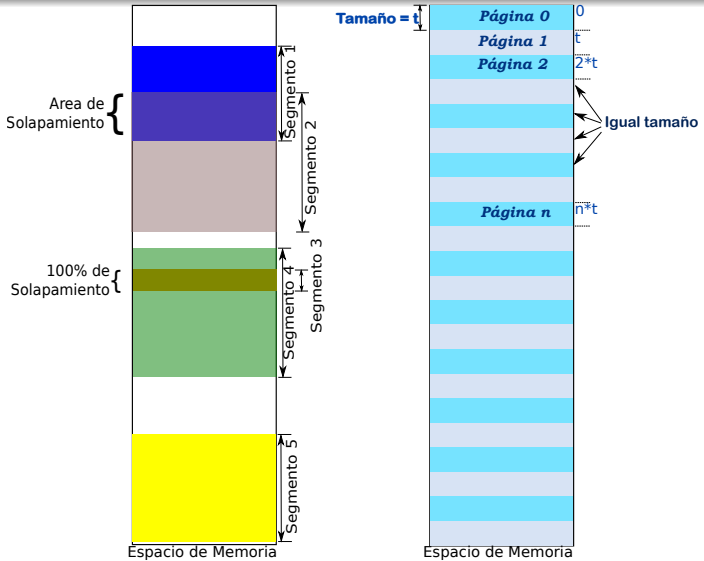
# Espacio Físico

- Los procesadores IA-32 organizan la memoria como una secuencia de bytes, direccionables a través de su Bus de Address.
- La memoria conectada a este bus se denomina ***memoria física*** .
- El espacio de direcciones que pueden volcarse sobre este bus se denomina ***direcciones físicas***.

## Capacidad de direccionamiento de memoria

Los procesadores IA-32 son la continuidad del 8086. Este procesador fue el primer de 16 bits de ancho de palabra, y por ende todos sus registros internos tienen ese tamaño. Su espacio de Direccionamiento es de 1 Mbyte $\therefore$ Address Bus es de 20 líneas. Este espacio de direccionamiento se administra por segmentación.

# Segmentación vs. Paginación



# Espacio Lógico

## Segmentación

Por diversos motivos que en su momento tuvieron sentido, Intel definió organizar el espacio de direccionamiento de la Familia iAPx86 en segmentos.

El compromiso de compatibilidad ató a los siguientes procesadores a mantener este esquema.

# Espacio Lógico

## Condiciones iniciales de segmentación

# Espacio Lógico

## Condiciones iniciales de segmentación

- 4 registros de segmento para almacenar hasta 4 selectores de segmento.

# Espacio Lógico

## Condiciones iniciales de segmentación

- 4 registros de segmento para almacenar hasta 4 selectores de segmento.
- Registros de 16 bits los segmentos tienen a lo sumo 64K de tamaño

# Espacio Lógico

## Condiciones iniciales de segmentación

- 4 registros de segmento para almacenar hasta 4 selectores de segmento.
- Registros de 16 bits los segmentos tienen a lo sumo 64K de tamaño
- Expresión de las direcciones en el modelo de programación mediante dos valores:

# Espacio Lógico

## Condiciones iniciales de segmentación

- 4 registros de segmento para almacenar hasta 4 selectores de segmento.
- Registros de 16 bits los segmentos tienen a lo sumo 64K de tamaño
- Expresión de las direcciones en el modelo de programación mediante dos valores:
  1. Identificador del segmento en el que se encuentra la variable o la instrucción que se desea direccionar,

# Espacio Lógico

## Condiciones iniciales de segmentación

- 4 registros de segmento para almacenar hasta 4 selectores de segmento.
- Registros de 16 bits los segmentos tienen a lo sumo 64K de tamaño
- Expresión de las direcciones en el modelo de programación mediante dos valores:
  1. Identificador del segmento en el que se encuentra la variable o la instrucción que se desea direccionar,
  2. Desplazamiento, offset, o ***dirección efectiva*** a partir del inicio de ese segmento en donde se encuentra efectivamente

# Operando en Memoria... CISC

Para identificar un operando (fuente o destino) de una instrucción en memoria, se utiliza lo que Intel denomina dirección lógica.

Este nombre obedece a que es una dirección abstracta expresada en términos de su arquitectura pero que necesita ser procesada para convertirse finalmente en la dirección física que es la que saldrá hacia el bus del sistema por las líneas de address.

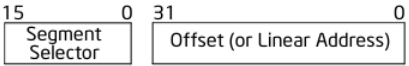



Figura: IA-32: Dirección lógica

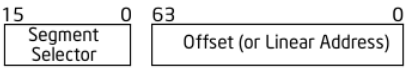



Figura: Intel® 64 : Dirección Lógica

# Segmentos para cada dirección Lógica

- El valor del segmento en una dirección lógica, puede especificarse de manera implícita.
- Por lo general este es el modo en que se hace, aunque es posible explicitar con que segmento se desea direccionar un operando de memoria determinado.
- Si no se especifica explícitamente el segmento como parte de la dirección lógica el procesador lo establecerá automáticamente de acuerdo a la tabla.

| Referencia a: | Reg. | Segmento | Regla de selección por defecto |
|---|---|---|---|
| Instrucciones | CS | Segmento de Código | Cada opcode fetch |
| Pila | SS | Segmento de Pila | Todos los push y pop, cualquier referencia a memoria que utilice como registro base ESP o EBP. |
| Datos Locales | DS | Segmento de datos | Cualquier referencia a un dato, excepto en el stack o un destino de instrucción de string |
| Strings Destino | ES | Segmento de datos extra direccionado por ES | Destino de Instrucciones de manejo de strings |

Reglas de selección de segmento predefinidos

# Desplazamiento para ubicar operandos en Memoria

La riqueza de modos de direccionamiento de operandos en memoria la da el desplazamiento.
Aquí es Intel puso todo el esfuerzo dando una gran cantidad de alternativas para calcular el desplazamiento.
Básicamente un desplazamiento tiene al menos uno de los siguientes componentes, o cualquiera de las combinaciones posibles:

- Desplazamiento directo: Se trata de un valor de 8, 16, o 32 bits, explícitamente incluido en la instrucción.
- Base: Se trata de un valor contenido en un registro de propósito general, que indica una dirección a partir de la cual se calcula el desplazamiento. Es un valor de 32 bits en el modo IA-32 y de 64 bits en IA-32e.
- Índice: Se trata de un valor contenido en un registro de propósito general, que se representa la dirección a la cual nos queremos referir. Típicamente es un valor que al incrementarse permite recorrer por ejemplo un buffer de memoria. Es un valor de 32 bits en el modo IA-32 y de 64 bits en IA-32e.
- Escala: Es un valor por el cual se multiplica el valor del Índice: Puede valer 2, 4, u 8.

# Calculando el desplazamiento

A partir de estos cuatro componentes se obtiene lo que Intel denomina Dirección Efectiva.
Los cuatro componentes anteriores pueden ser positivos o negativos en representación Ca2
(excepto el valor del factor de escala que es siempre positivo).
A continuación se presentan los diferentes registros y valores que pueden tomar los cuatro componentes:

| Base | Índice | Escala | Desplazamiento |
|---|---|---|---|
| EAX | EAX | 1 | Nada |
| EBX | EBX | 2 | 8bits |
| ECX | ECX | 4 | 16bits |
| EDX | EDX | 8 | 32bits |
| ESP | EBP | | |
| EBP | EDI | | |
| EDI | ESI | | |
| ESI | | | |

$$\begin{bmatrix} EAX \\ EBX \\ ECX \\ EDX \\ ESP \\ EBP \\ EDI \\ ESI \end{bmatrix} + \left[ \begin{pmatrix} EAX \\ EBX \\ ECX \\ EDX \\ EBP \\ EDI \\ ESI \end{pmatrix} * \begin{pmatrix} 1 \\ 2 \\ 4 \\ 8 \end{pmatrix} \right] + \begin{bmatrix} Nada \\ 8bits \\ 16bits \\ 32bits \end{bmatrix}$$

En general la expresión general que representa el cálculo interno del procesador es:

$$DireccionEfectiva = Base + (Indice * escala) + Desplazamiento$$

# Casos particulares

Cuestiones a destacar, respecto de la matriz anterior:

- El registro ESP (o RSP), no puede utilizarse de Índice. Esto es bastante lógico ya que su uso privilegiado es como puntero de pila. Por lo tanto modificarlo para recorrer otro array de datos que no sea la pila es poco menos que imprudente.
- Cuando se emplean como registros base ESP y EBP (o RSP y RBP), se utilizan asociados al registro de segmento SS. Para el resto de los usos se asocian al DS.
- El factor de escala solo se puede emplear cuando se utiliza un Registro Índice. En otro caso no se emplea.

# ¿El Offset puede venir directo en la Instrucción?

Se incluye en la instrucción un valor en forma explícita que representa el valor del offset.
En el listado siguiente se muestran diversos ejemplos de instrucciones de este Modo.

```
1  or    ecx, dword [0x300040A0]  ;Calcula la or lógica entre
2                                 ;ECX y la doble word contenida
3                                 ;a partir de la direccion de
4                                 ;memoria 0x300040A0.
5  inc   byte [0xAF007600]        ;Incrementa el byte contenido
6                                 ;por la direccion de memoria
7                                 ;0xAF007600
8  dec   dword [i]                ;El valor de la dirección de la
9                                 ;variable i se calcula directa-
10                                ;mente y el valor se reemplaza
11                                ;en tiempo de compilación
```

# Una de las formas de direccionar en forma indirecta

En este modo el ofsset está contenido directamente en un registro Base.

El procesador simplemente lo toma desde el registro, sin otro cálculo.

```
1 mov    edx, i      ;edx = desplazamiento de i en el segmento
2                    ;de datos
3 inc    [edx]       ;incrementamos i direccionada a través de
4                    ;un registro puntero base.
```

El ejemplo anterior es trivial ya que la variable como vimos puede incrementarse de manera directa.

# Formas mas útiles de direccionar en forma indirecta

Combina el valor contenido en un registro que apunta a la base de un bloque de datos con un valor explícito puesto en la instrucción, que permite calcular la dirección efectiva del operando.

También resulta útil para acceder a una estructura mas compleja de datos, apuntando a la base de la estructura con un registro y utilizando el Desplazamiento para acceder al campo deseado de la estructura.

# Formas mas útiles de direccionar en forma indirecta

```
1  ORG   8000h
2  use16                ;Estamos en Modo Real=>Código de 16 bits
3  start:       jmp main        ;Salto al inicio del programa.
4  ALIGN 8
5  gdt:         resb 8          ;NULL Descriptor. Dejamos 8 bytes sin usar.
6  Data_sel     equ  $-gdt      ;Calcula dinámicamente la posición del slector
7  K_data:      dw   0xffff     ;límite 15.00
8               dw   0x0000     ;base 15.00
9               db   0x00       ;base 23.16
10              db   10010010b  ;Presente Segmento Datos Read Write
11              db   0xCF       ;G = 1, D/B = 1, y límite 0Fh
12              db   0x00       ;base 31.24
13 gdt_size     equ  $-gdt      ;calcula dinámicamente el tama´no de la gdt
14 main:
15 ...
16 ...
17              mov  ebx,K_data ;ebx apunta a la base del descriptor
18 ;////////////////////////////////////////////////////////////////////////
19 ;Lee con direccionamiento base + desplazamiento los atributos del descriptor
20 ;de segmento de datos definido en la tabla anterior
21              mov  al, byte [ebx + 5]
22 ;////////////////////////////////////////////////////////////////////////
```

# Formas mas útiles de direccionar en forma indirecta

Es una forma eficiente de acceder a dato de un array cuyo tamaño es 2, 4, u 8 bytes.

El Desplazamiento puede ubicar el inicio del array y el valor índice se guarda en un registro que al incrementar pasa al siguiente elemento permitiendo con la escala ajustar al tamaño del mismo.

```
1 % define Dir_Tabla     0x2000F000
2 % define mascara       0xFFFFFFE
3       mov   ecx,size_tabla ;ecx = cantidad de elementos de 4 bytes de la tabla.
4       xor   esi,esi        ;esi apunta al inicio de la tabla
5 mas:
6       and   [esi*4 + Dir_Tabla], mascara
7       Origenes;borra bit menos significativo del elemento de la tabla
8       inc   esi            ;esi apunta al siguiente elemento de 4 bytes
9       loop  mas            ;va por el siguiente elemento hasta que ECX llegue a 0
```

# Formas mas útiles de direccionar en forma indirecta

- Este modo es especial para acceder a matrices bidimensionales.
- Un ejemplo obligado es el buffer de video
- Cuando trabaja en modo texto el buffer de video es una matriz de 25 filas por 80 columnas.
- Cada elemento consta de dos bytes:
  el primero contiene el ASCII del caracter a presentar y el segundo los atributos (intensificado, video inverso, y colores de caracter y fondo).
- Vamos a escribir un código que aprovechando este modo de direccionamiento sirva para limpiar el contenido de la pantalla.
- El registro base apunta a cada línea de la pantalla y el índice apunta a cada elemento de la línea.

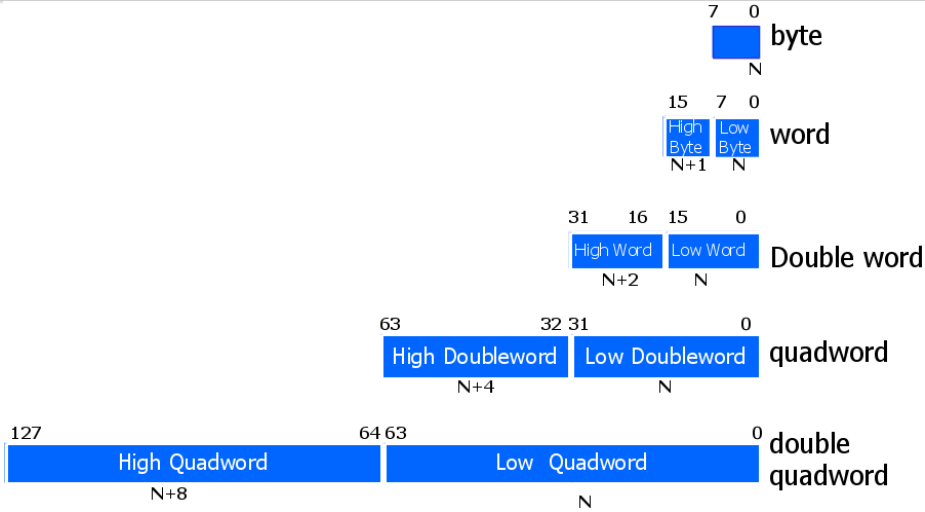
# Formas mas útiles de direccionar en forma indirecta

```
1  ;El siguiente código limpia la pantalla dejándola en modo blanco sobre neg
2  ;El buffer de video comienza en la dirección física 0x000B8000. Utilizamos
3  ;este valor como desplazamiento en el cálculo de la dirección efectiva.
4    xor    ebx,ebx        ;EBX = 0. EBX apunta a la 1er. fila de 80 caracteres
5  col:
6    xor    edi,edi        ;EDI = 0. EDI apunta al primer elemento de la fila
7    mov    ecx,size_row   ;ECX = cantidad de filas para loop
8  row:
9    mov    byte [ebx + edi + 0x000B8000],0x00 ;caracter nulo no imprime nada
10   add    edi,2          ;EDI apunta al porx. elemento de 2 bytes
11   loop   row            ;si ECX = 0 se completó fila
12   add    ebx,160        ;Apunta a la siguiente fila
13   cmp    ebx,0x1000     ;fin del buffer?
14   jle    col
```

# Formas mas útiles de direccionar en forma indirecta

```
1  ;El siguiente código limpia la pantalla dejándola en modo blanco sobre neg
2  ;El buffer de video comienza en la  dirección física 0x000B8000. Utilizamos
3  ;este valor como desplazamiento en el cálculo de la dirección efectiva.
4    xor    ebx,ebx       ;EBX = 0. EBX apunta a la 1er. fila de 80 caracteres
5  col:
6    xor    edi,edi       ;EDI = 0. EDI apunta al primer elemento de la fila
7    mov    ecx,size_row  ;ECX = cantidad de filas para loop
8  row:
9    mov    byte [ebx + edi*2 + 0x000B8000],0x00 ;caracter nulo no imprime nad
10   inc    edi           ;EDI apunta al porx. elemento de 2 bytes
11   loop   row           ;si ECX = 0 se completó fila
12   add    ebx,160       ;Apunta a la siguiente fila
13   cmp    ebx,0x1000    ;fin del buffer?
14   jle    col
```

# Tipos básicos de datos




IA-32 e Intel® 64 : Tipos de datos fundamentales

# Little endian

- Desde el procesador 8086, esta familia maneja el almacenamiento en memoria de las variables en el formato little endian.

# Little endian

- Desde el procesador 8086, esta familia maneja el almacenamiento en memoria de las variables en el formato little endian.
- Dicho de otra forma: una variable de varios bytes de tamaño almacena su byte menos significativo en la dirección con que se referencia la variable y a partir de allí coloca el resto de los bytes en orden de significancia, terminando con el almacenamiento del byte mas significativo, en la dirección de memoria mas alta (es decir termina con el menor, de allí little endian).

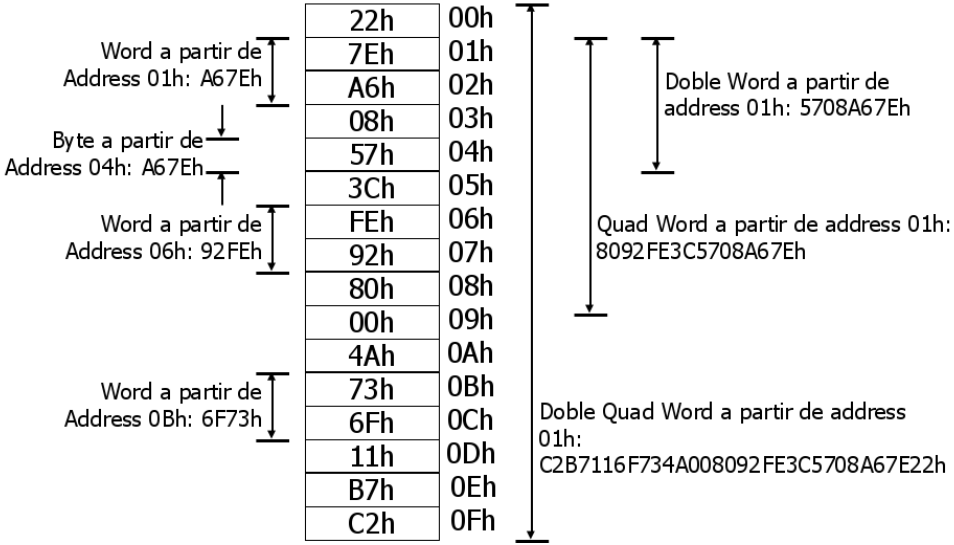
# Little endian

- Desde el procesador 8086, esta familia maneja el almacenamiento en memoria de las variables en el formato little endian.
- Dicho de otra forma: una variable de varios bytes de tamaño almacena su byte menos significativo en la dirección con que se referencia la variable y a partir de allí coloca el resto de los bytes en orden de significancia, terminando con el almacenamiento del byte mas significativo, en la dirección de memoria mas alta (es decir termina con el menor, de allí little endian).
- Esta situación se representa en próximo slide. A simple vista pareciera que están almacenados al revés, ya que si lo miramos en la memoria está de atrás hacia adelante.

# Little endian

- Desde el procesador 8086, esta familia maneja el almacenamiento en memoria de las variables en el formato little endian.
- Dicho de otra forma: una variable de varios bytes de tamaño almacena su byte menos significativo en la dirección con que se referencia la variable y a partir de allí coloca el resto de los bytes en orden de significancia, terminando con el almacenamiento del byte mas significativo, en la dirección de memoria mas alta (es decir termina con el menor, de allí little endian).
- Esta situación se representa en próximo slide. A simple vista pareciera que están almacenados al revés, ya que si lo miramos en la memoria está de atrás hacia adelante.
- Otros procesadores utilizan el formato Big Endian, es decir colocando la información en el orden en el que normalmente esperamos encontrarla.

# IA-32 e Intel® 64 : layout en memoria para diferentes tipos de datos



# Little endian

- La razón por la que Intel adoptó Little Endian, obedece a que el procesador 8086 (y sus sucesores por cuestión de compatibilidad), administra la memoria de a bytes.

# Little endian

- La razón por la que Intel adoptó Little Endian, obedece a que el procesador 8086 (y sus sucesores por cuestión de compatibilidad), administra la memoria de a bytes.
- Esto significa que cada dirección de memoria tiene una capacidad de almacenamiento de 8 bits.

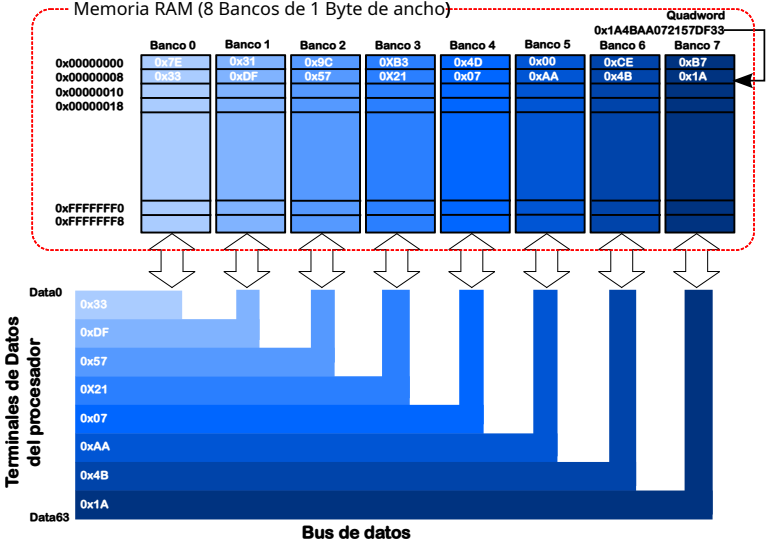
# Little endian

- La razón por la que Intel adoptó Little Endian, obedece a que el procesador 8086 (y sus sucesores por cuestión de compatibilidad), administra la memoria de a bytes.
- Esto significa que cada dirección de memoria tiene una capacidad de almacenamiento de 8 bits.
- Por lo tanto, y debido a esta decisión de diseño, es que un bus de datos de 16 bits primero, 32 más tarde, y 64 actualmente, se conecta a bancos de memoria RAM Dinámica organizados en bytes.

# Little endian

- La razón por la que Intel adoptó Little Endian, obedece a que el procesador 8086 (y sus sucesores por cuestión de compatibilidad), administra la memoria de a bytes.
- Esto significa que cada dirección de memoria tiene una capacidad de almacenamiento de 8 bits.
- Por lo tanto, y debido a esta decisión de diseño, es que un bus de datos de 16 bits primero, 32 más tarde, y 64 actualmente, se conecta a bancos de memoria RAM Dinámica organizados en bytes.
- Por lo tanto, cuando el procesador lee un dato de 64 bits a través del bus de datos, cada byte de las direcciones de memoria que se leen viaja por un byte del bus de datos, de acuerdo al ordenamiento que la información tiene en la memoria, de la manera en que se muestra en el próximo slide.

# Little endian: Relación con el conexionado del Bus de Datos.



# ¿Porque conviene alinear los datos?

# ¿Porque conviene alinear los datos?

- Los procesadores IA-32 e Intel® 64 no ponen restricciones respecto de la alineación en memoria para las diferentes variables de los programas (consecuencia favorable de la administración de memoria de a bytes).

# ¿Porque conviene alinear los datos?

- Los procesadores IA-32 e Intel® 64 no ponen restricciones respecto de la alineación en memoria para las diferentes variables de los programas (consecuencia favorable de la administración de memoria de a bytes).
- Esto otorga gran flexibilidad a la hora de aprovechar al máximo la memoria.

# ¿Porque conviene alinear los datos?

- Los procesadores IA-32 e Intel® 64 no ponen restricciones respecto de la alineación en memoria para las diferentes variables de los programas (consecuencia favorable de la administración de memoria de a bytes).
- Esto otorga gran flexibilidad a la hora de aprovechar al máximo la memoria.
- Pero si una variable queda repartida en dos filas diferentes, se requerirán dos ciclos de lectura para accederla.

# ¿Porque conviene alinear los datos?

- Los procesadores IA-32 e Intel® 64 no ponen restricciones respecto de la alineación en memoria para las diferentes variables de los programas (consecuencia favorable de la administración de memoria de a bytes).
- Esto otorga gran flexibilidad a la hora de aprovechar al máximo la memoria.
- Pero si una variable queda repartida en dos filas diferentes, se requerirán dos ciclos de lectura para accederla.
- La variable se trae al procesador con dos lecturas de memoria

# ¿Porque conviene alinear los datos?

- Los procesadores IA-32 e Intel® 64 no ponen restricciones respecto de la alineación en memoria para las diferentes variables de los programas (consecuencia favorable de la administración de memoria de a bytes).
- Esto otorga gran flexibilidad a la hora de aprovechar al máximo la memoria.
- Pero si una variable queda repartida en dos filas diferentes, se requerirán dos ciclos de lectura para accederla.
- La variable se trae al procesador con dos lecturas de memoria
- Estas dos lecturas de memoria son transparentes a nivel de software (la aplicación no debe ser modificada en absoluto), ya que el procesador autónomamente realiza las dos lecturas.

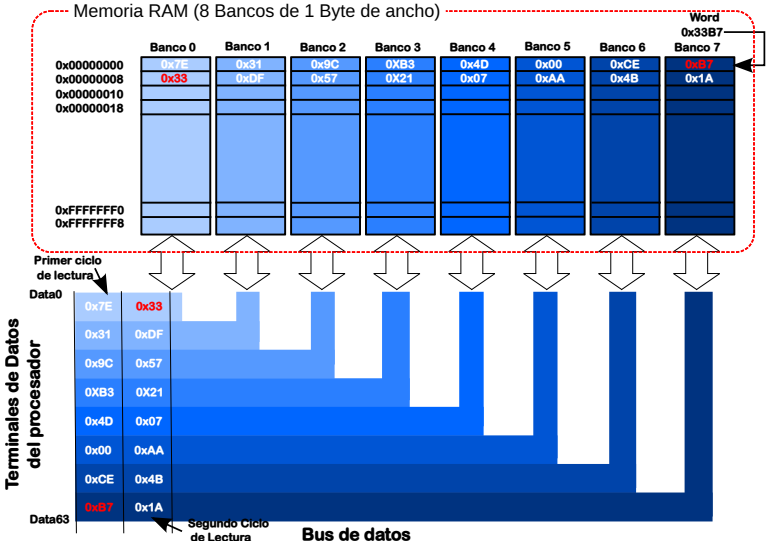
# ¿Porque conviene alinear los datos?

- Los procesadores IA-32 e Intel® 64 no ponen restricciones respecto de la alineación en memoria para las diferentes variables de los programas (consecuencia favorable de la administración de memoria de a bytes).
- Esto otorga gran flexibilidad a la hora de aprovechar al máximo la memoria.
- Pero si una variable queda repartida en dos filas diferentes, se requerirán dos ciclos de lectura para accederla.
- La variable se trae al procesador con dos lecturas de memoria
- Estas dos lecturas de memoria son transparentes a nivel de software (la aplicación no debe ser modificada en absoluto), ya que el procesador autónomamente realiza las dos lecturas.
- Entonces **¿cual es el problema?**. La respuesta es: **performance**. Dos ciclos de lectura en lugar de uno solo tornan mas lento el acceso a la variable. Esto puede evitarse usando las directivas de alineación que todos los lenguajes poseen.

# ¿Porque conviene alinear los datos?

- Los procesadores IA-32 e Intel® 64 no ponen restricciones respecto de la alineación en memoria para las diferentes variables de los programas (consecuencia favorable de la administración de memoria de a bytes).
- Esto otorga gran flexibilidad a la hora de aprovechar al máximo la memoria.
- Pero si una variable queda repartida en dos filas diferentes, se requerirán dos ciclos de lectura para accederla.
- La variable se trae al procesador con dos lecturas de memoria
- Estas dos lecturas de memoria son transparentes a nivel de software (la aplicación no debe ser modificada en absoluto), ya que el procesador autónomamente realiza las dos lecturas.
- Entonces **¿cual es el problema?**. La respuesta es: **performance**. Dos ciclos de lectura en lugar de uno solo tornan mas lento el acceso a la variable. Esto puede evitarse usando las directivas de alineación que todos los lenguajes poseen.
- Esta situación se representa en el siguiente slide.

# Acceso a datos no alineados



# ¿Preguntas?