

Tema 6: Arquitectura Intel

- 6.1- Evolución histórica
- 6.2- Parámetros de la arquitectura
- 6.3- Modelos de organización del espacio de memoria
- 6.4- Tipos de datos
- 6.5- Registros
- 6.6- Formato de las instrucciones
- 6.7- Modos de direccionamiento
- 6.8- Rudimentos de ensamblador
- 6.9- Juego de instrucciones
- 6.10- Gestión de la pila
- 6.11- Procedimientos y paso de parámetros
- 6.12- Traducción de lenguaje de alto nivel a código máquina
- 6.13- Introducción a la arquitectura x86-64

Evolución histórica: microprocesadores de 16 bits

Arquitectura: Definición del nivel de máquina convencional

- 8086 (año 1978)
 - Evolucionada del procesador Intel 8080 de 8 bits
 - Bus de datos: 16 líneas
 - Bus de direcciones: 20 líneas
- 8088 (año 1979)
 - Versión barata del 8086
 - Bus de datos: 8 líneas
 - Es utilizado por IBM para fabricar el PC (Personal Computer)
- 80186 (año 1982)
 - Versión mejorada del 8086
 - Poco utilizado
- 80286 (año 1982)
 - Permite usar 24 líneas de dirección en un modo especial

Evolución histórica: microprocesadores de 32 bits

- 80386 (año 1985)
 - Bus de datos: 32 líneas
 - Bus de direcciones: 32 líneas
 - Dos modos:
 - Modo real: compatible con 8086
 - Modo protegido: aprovecha todas las características del 80386
- Principales sucesores:
 - 80486 (año 1989)
(Problemas con el registro de marcas con números)
 - Pentium (año 1993)
 - Pentium Pro (año 1995)
 - Pentium II (año 1997)
 - Pentium III (año 1999)
 - Pentium 4 (año 2000)
 - Pentium M (año 2003)
 - Evoluciona a partir del Pentium III
 - Para portátiles (baja frecuencia, bajo consumo)
 - Core (año 2006)
 - Core 2 (año 2006)
 - Core i7 (año 2008)



Evolución histórica: clónicos

- Las especificaciones del IBM PC eran abiertas
 - Distintos fabricantes se introdujeron en el negocio de producir componentes para el IBM PC
- En el caso del microprocesador, otros fabricantes han producido microprocesadores clónicos a Intel (soportan el mismo juego de instrucciones):
 - AMD, Cyrix, IBM, NEC, VIA, etc.
- De entre ellos el que más éxito ha tenido ha sido AMD:
 - Am386
 - Am486
 - K5
 - K6
 - K7

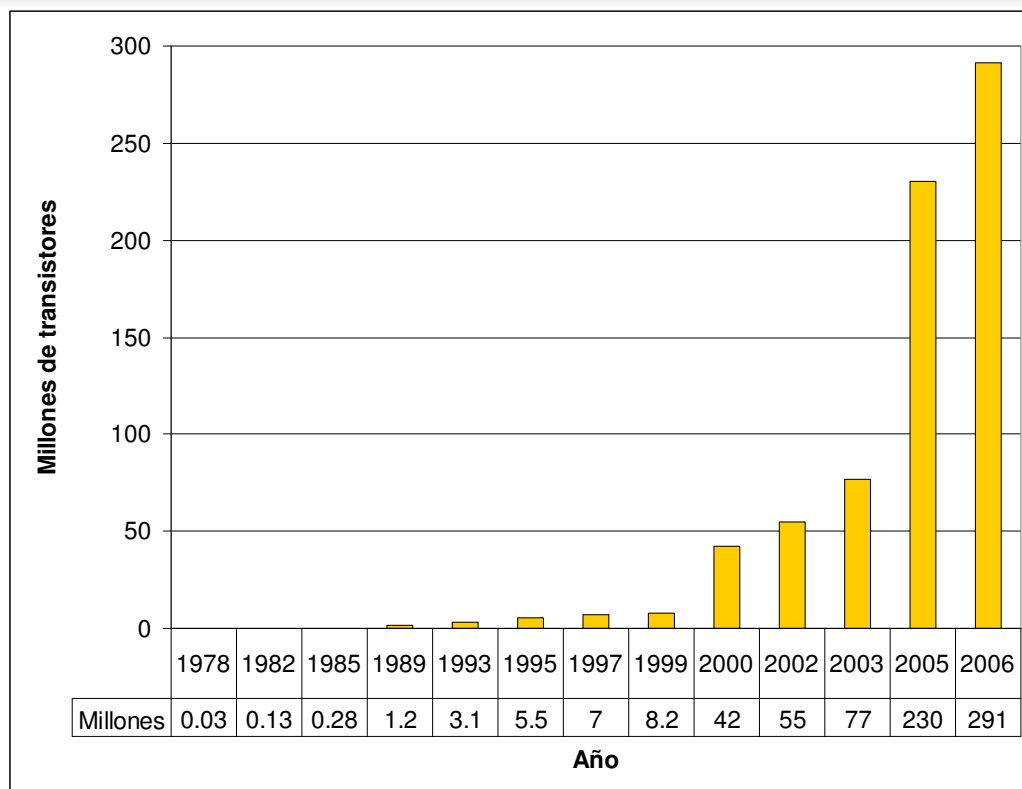


Evolución histórica: Llegan los 64 bits

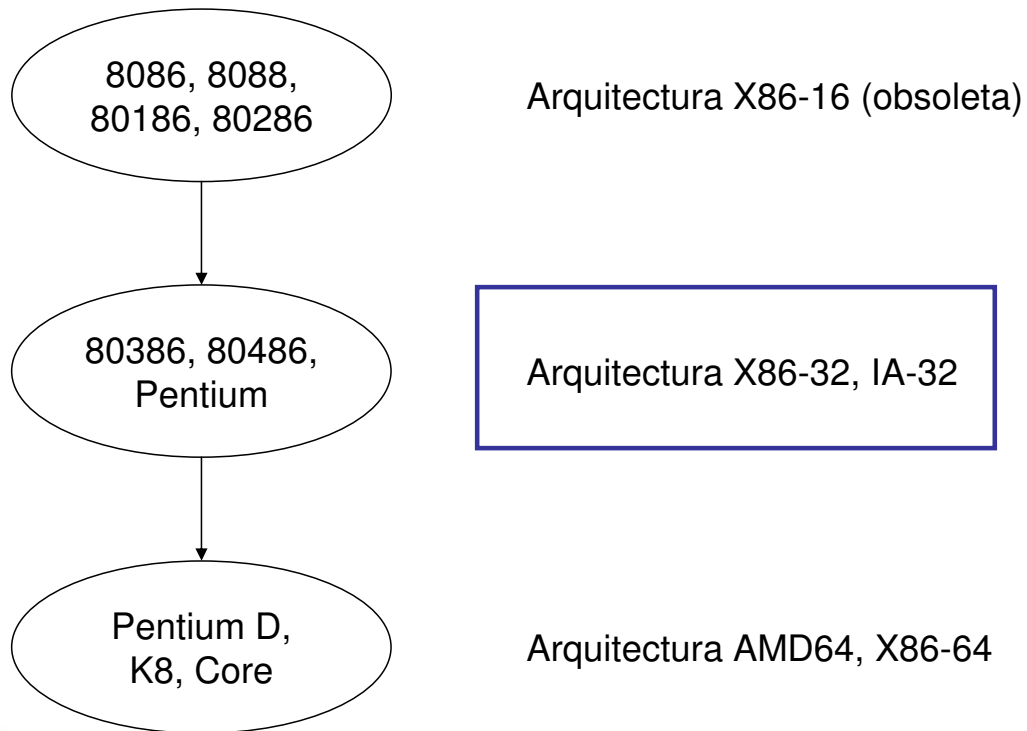
- Arquitectura de 64 bits diseñada por Intel (IA-64):
 - No es compatible con la arquitectura de 32 bits
 - Itanium I (año 2001)
 - Itanium II (año 2002)
 - La arquitectura fracasa, sólo usada en servidores
- Arquitectura de 64 bits diseñada por AMD (AMD64, x86-64, x64):
 - Compatible con la arquitectura de 32 bits
 - K8
- Intel rectifica y “copia” la arquitectura propuesta por AMD (EM64T):
 - Pentium D (año 2005): dos Pentium 4 de 64 bits
 - Core y Core 2 (año 2006) evolucionan a partir del Pentium M
- Tendencia actual: reducir la frecuencia y con ello el consumo y la disipación de calor, aumentar el número de núcleos
 - En la actualidad dos y cuatro: Duo, Quad



Evolución histórica: complejidad



Evolución histórica: resumen



Evolución histórica: más

- Sobre la historia:
 - <http://www.wikipedia.org/>
- Comparativas de rendimiento:
 - <http://www.tomshardware.com/>

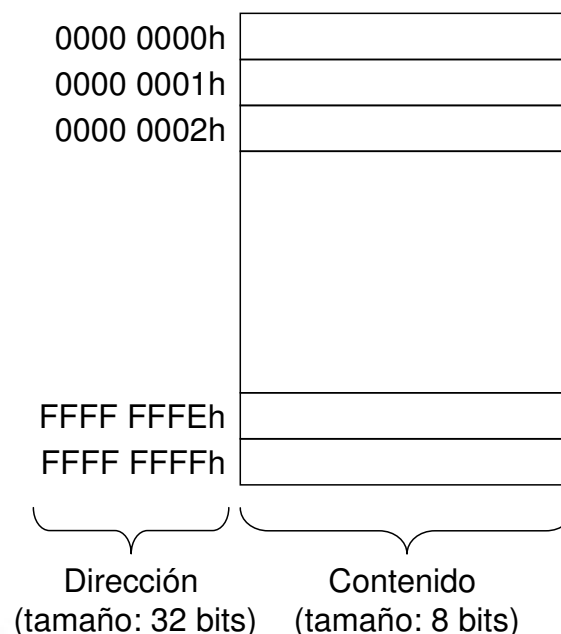
Parámetros de la arquitectura

	CPU Teórica	IA-32
Ancho de la arquitectura (tamaño de los operandos)	16	32
Ancho de las posiciones de memoria	16 (16 bits = 1 palabra = 1 word)	8 (8 bits = 1 byte)
Ancho de las direcciones de memoria	16	32
Dimensiones del espacio de direcciones	64 KW	4 GB



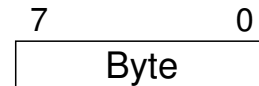
Modelos de organización del espacio de memoria

- Modelo segmentado (no se estudia)
- Modelo plano (flat):



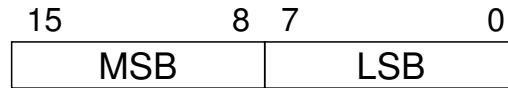
Tipos de datos

- Byte (B) (8 bits)

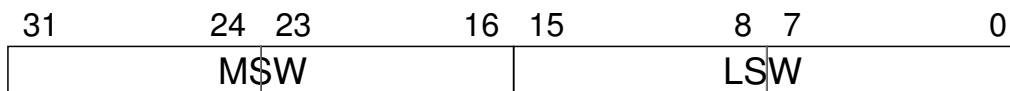


- Palabra (Word) (W) (16 bits)

LSB: Least Significant Byte
MSB: Most Significant Byte



- Doble palabra (Double word) (D) (32 bits)



LSW: Least Significant Word
MSW: Most Significant Word



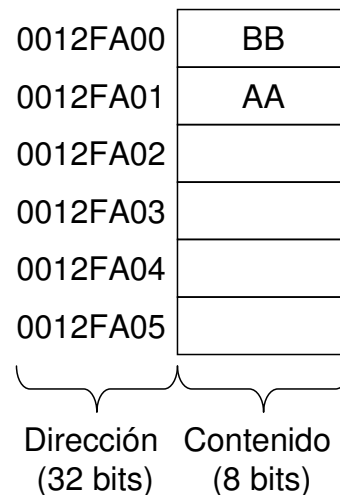
Almacenamiento de datos

- Ejemplo: almacenar el valor AAB Bh en memoria
 - Recordatorio: El ancho de una posición de memoria en Intel es un byte

Alternativa 1:



Alternativa 2:



Es necesario establecer un criterio



Almacenamiento de datos: criterio Little Endian

- Criterio Little Endian:

Las palabras y doubles palabras se organizan en memoria ocupando un conjunto consecutivo de posiciones contiguas, de modo que **el byte menos significativo se sitúa en la posición de memoria más baja** y así, ordenadamente, hasta el byte más significativo que se sitúa en la posición más alta.



Ejercicios de almacenamiento de datos

- Ejercicio:

- Almacenar el valor AABBCDDh a partir de la dirección 0012FA03h

0012FA00	
0012FA01	
0012FA02	
0012FA03	
0012FA04	
0012FA05	
0012FA06	



Ejercicios de almacenamiento de datos

0012FA00	02
0012FA01	04
0012FA02	01
0012FA03	20
0012FA04	21
0012FA05	40
0012FA06	10

- Ejercicios:

- Se ha almacenado un dato de 16 bits a partir de la dirección 0012FA01, determinar cual es su valor en decimal interpretado como entero
- Se ha almacenado un dato de 32 bits a partir de la dirección 0012FA03, determinar cual es su valor en hexadecimal



Registros

- Registros generales:

- Se usan para generar direcciones de memoria y para contener los operandos de las instrucciones aritmético-lógicas

	31	16	15	8	7	0
EAX				AH	AX	AL
EBX				BH	BX	BL
ECX				CH	CX	CL
EDX				DH	DX	DL
ESI				SI		
EDI				DI		
EBP				BP		
ESP				SP		

Compatible con el X86-16



Registros

- Registros de segmento:
 - Para el acceso a la memoria en formato segmentado
- Registro puntero de instrucción (EIP):
 - Contiene la dirección de la siguiente instrucción a ejecutar. Equivalente al PC de la CPU elemental. ¿Cuál será su tamaño?
- Registro de estado (EFLAGS):
 - Registro de 32 bits, los que se van a utilizar son:
 - Bit de peso 0: CF, Carry Flag
 - Bit de peso 6: ZF, Zero Flag
 - Bit de peso 7: SF, Sign Flag
 - Bit de peso 11: OF, Overflow Flag



Formato de las instrucciones

- Las instrucciones, al igual que los datos, necesitan ser codificadas mediante secuencias de ceros y unos

Codificación

MOV ESP, EBP —————> 8B EC

PUSH ECX —————> 51

- En la CPU elemental ...
 - se usa la hoja de codificación
- En Intel ...
 - se usa un documento de cientos de páginas

En Intel no se codifica manualmente



Formato de las instrucciones

- Sin embargo, es necesario conocer algunos aspectos de la codificación:
 - La codificación de una instrucción para el 8086 (año 1978) es válida hoy en día

Se mantiene la compatibilidad

Ventajas: Un programa desarrollado para el 8086 (año 1978) se puede ejecutar en un Intel Core i7 (año 2008)

- La codificación de una instrucción puede necesitar de 1 a 8 bytes

El tamaño de la codificación es variable

Ventajas: Es posible utilizar en las instrucciones modos de direccionamiento más complejos y datos inmediatos de distintos tamaños

Codificación

MOV EAX, 0AABBCCDDh —————> B8 DD CC BB AA

MOV AL, 0DDh —————> B0 DD



Formato de las instrucciones

- Implicaciones de la codificación de tamaño variable:
 - Una instrucción se almacena en un conjunto de posiciones de memoria consecutivas
 - El número de palabras que ocupa la codificación de una instrucción en memoria es desconocido si no se conoce la codificación de la instrucción
 - Resulta en una complejidad mucho mayor que la CPU elemental a la hora de determinar desplazamientos relativos en los saltos

Ventajas: en Intel se pueden usar instrucciones más complejas

Desventajas: todo lo demás



Modos de direccionamiento

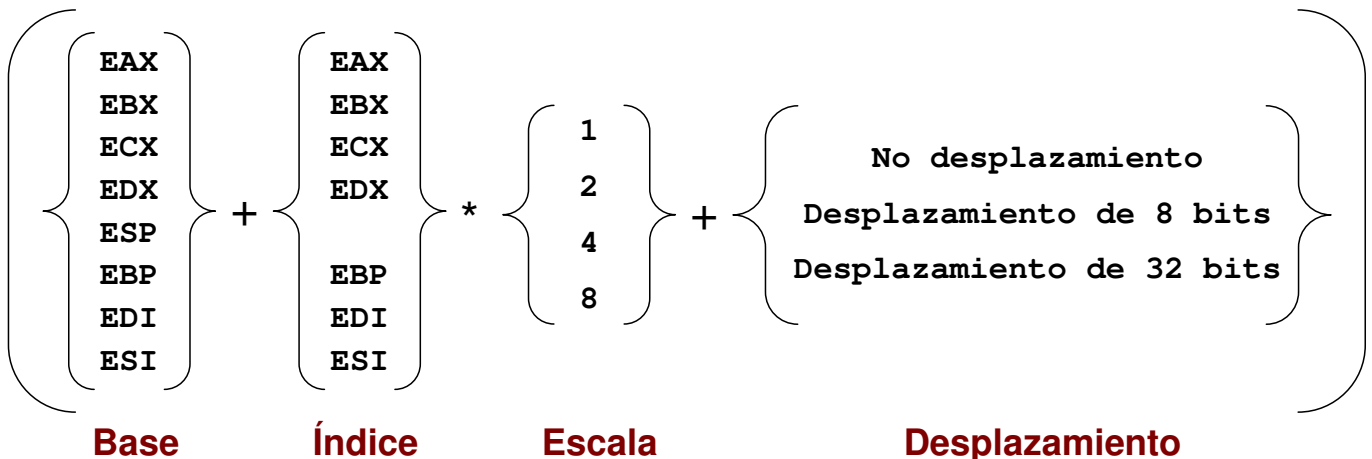
- Modo 1: direccionamiento a registro
 - El operando se encuentra en un registro
 - Ejemplos: `MOV AL, BL`
`MOV EAX, EBX`
`MOV AL, EBX` (Error: registros de distintos tamaños)
- Modo 2: direccionamiento inmediato
 - El operando se encuentra codificado en el código de la instrucción
 - Los inmediatos se codifican en Little Endian al final de la instrucción
 - Ejemplos: `MOV BL, 34h` Codificación: B3 34
`MOV EBX, 34h` Codificación: BB 34 00 00 00

Código de instrucción Inmediato



Modos de direccionamiento

- Modo 3: direccionamiento a memoria
 - El operando se encuentra en una posición de memoria
 - La posición de memoria se indica entre corchetes
 - Los elementos que pueden participar en el calculo de la dirección son:



$$\text{Dirección} = \text{Base} + \text{Índice} * \text{Escala} + \text{Desplazamiento}$$



Modos de direccionamiento

- Modo 3: direccionamiento a memoria (continuación)

- El desplazamiento se codifica en la propia instrucción (al final). Puede ser una constante numérica, una etiqueta, una combinación de ambos, o puede no existir.

- Ejemplos:

MOV AL, [EBX]

MOV AL, [EBX + 127]

MOV AL, [EBX + ESI*2 + 127] Codificación: 8A 44 73 **7F**

MOV AL, [vector + EBX + ESI*2 + 127]



Ejercicios de modos de direccionamiento

EBX	→	0012FA00	EF
0012FA00		0012FA01	BE
		0012FA02	AD
		0012FA03	DE
EAX		0012FA04	DA
9A6C56A3		0012FA05	CA
		0012FA06	FE

1. **MOV AL, [EBX]**

¿EAX?

--	--	--	--

2. **MOV AX, [EBX]**

¿EAX?

--	--	--	--

3. **MOV EAX, [EBX]**

¿EAX?

--	--	--	--



Indeterminación en modos de direccionamiento

- Determinar el tamaño de los operandos:

1. **MOV AL, [EBX]**

2. **MOV [EBX], EAX**

3. **MOV CL, [EBX + ESI*4 +45]**

4. **MOV DI, 12h**

5. **MOV [EAX], 12h** →

Indeterminación:
¿12h? ¿0012h? ¿00000012h?

6. **INC [EBX]** →

Indeterminación:
incrementa ¿byte? ¿palabra? ¿doble palabra?



Indeterminación en modos de direccionamiento

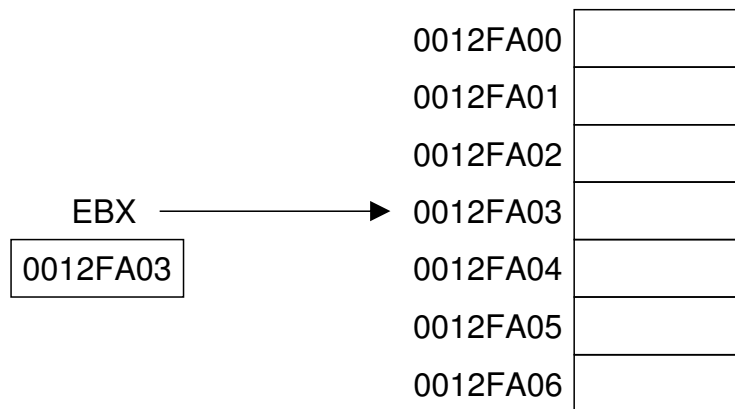
- Ciertas combinación de modos de direccionamiento producen indeterminación
 - Para evitarlo se usa el operador PTR
 - BYTE PTR: especifica un dato inmediato de 8 bits o acceso a memoria de tipo byte
 - WORD PTR: especifica un dato inmediato de 16 bits o acceso a memoria de tipo palabra
 - DWORD PTR: especifica un dato inmediato de 32 bits o acceso a memoria de tipo doble palabra
 - Ejemplos sin indeterminación:
 - MOV [EAX], WORD PTR 12h** → Codifica el inmediato como 0012h
 - INC BYTE PTR [EBX]** → Incrementa un byte



Indeterminación en modos de direccionamiento

- Indicar el contenido de la memoria tras ejecutar la instrucción

MOV [EBX + 2], WORD PTR 12h



Rudimentos de ensamblador para la arquitectura Intel x86

DIRECTIVAS			
TIPO de directiva	CPU elemental	INTEL	Comentarios
Definición de la estructura en secciones	.DATOS	.DATA	
	.CODIGO	.CODE	
	.PILA	—	
Control de carga	ORIGEN	—	
	INICIO	—	
Definición del modelo de programa	—	.386	
	—	.MODEL flat, stdcall	
Finalización de código	FIN	END etiqueta	“etiqueta” señala la primera instrucción a ejecutar
Definición de procedimientos	PROCEDIMIENTO	PROC	
	FINP	ENDP	
Definición de datos	—	DB	Define byte
	VALOR	DW	Define palabra
	—	DD	Define doble palabra
	VECES	DUP	Define un array
Declaración de procedimientos externos al fichero	—	proc PROTO	Permite usar un procedimiento no definido en el fichero fuente



- Constantes
 - Igual al ensamblador de la CPU elemental
- Etiquetas
 - Igual al ensamblador de la CPU elemental
- Operadores

CPU elemental	INTEL	Comentarios
DIRECCION etiqueta	OFFSET etiqueta	Calcula la dirección de la etiqueta
BYTEALTO	—	
BYTEBAJO	—	
—	BYTE PTR	Especifica tamaño BYTE
—	WORD PTR	Especifica tamaño PALABRA
—	DWORD PTR	Especifica tamaño DOBLE PALABRA



Ejemplo de programa

```
.386
.MODEL FLAT, STDCALL
ExitProcess PROTO, dwExitCode:DWORD
```

.DATA

```
    lista      DB 3, 5, -1
    resultado DB 0
```

.CODE

Inicio:

```
    MOV ESI, OFFSET lista
    MOV AL, [ESI]
```

```
    INC ESI
    MOV BL, [ESI]
    ADD AL, BL
```

```
    INC ESI
    MOV BL, [ESI]
    ADD AL, BL
```

```
    INC ESI
    MOV [ESI], AL
```

```
    PUSH 0
    CALL ExitProcess
```

```
END Inicio
```

Calcular la suma
acumulada de una
lista de números



Juego de instrucciones

- Instrucción de movimiento

- **MOV** (transferencia)

- Sintaxis: **MOV {R|M}, {R|M|I}**
 - Ejemplo: **MOV AL, BL; AL ← BL**
 - Objetivo: transferir información

Una instrucción no puede contener dos operandos en memoria, en el destino y en la fuente, a la vez

- Instrucciones de conversión de tipo

- **MOVSX** (transferencia con extensión de signo)

- Sintaxis: **MOVSX {R16|M16}, {R8|M8}**
MOVSX {R32|M32}, {R8|M8}
MOVSX {R32|M32}, {R16|M16}
 - Ejemplo: **MOVSX BX, AL; BX ← AL**
(AL interpretado como un número CON signo)
 - Objetivo: convertir datos interpretados **con** signo (enteros). Extiende el signo de un valor de 8 bits a un valor de 16 bits, o bien el de un valor de 8 ó 16 bits a uno de 32



Juego de instrucciones

- **MOVZX** (transferencia con extensión de cero)

- Sintaxis: **MOVZX {R16|M16}, {R8|M8}**
MOVZX {R32|M32}, {R8|M8}
MOVZX {R32|M32}, {R16|M16}
 - Ejemplo: **MOVZX BX, AL; BX ← AL**
(AL interpretado como un número SIN signo)
 - Objetivo: convertir datos interpretados **sin** signo (naturales). Extiende un valor de 8 bits a un valor de 16 bits, o bien el de un valor de 8 ó 16 bits a uno de 32. La extensión se realiza insertando ceros.

- Ejercicio:

- **AL** contiene el valor **FCh**. Determinar el valor de **EBX** tras ejecutar las siguientes instrucciones. Responder en hexadecimal:

- **MOV BH, AL** ¿EBX?

--	--	--	--
 - **MOVSX BX, AL** ¿EBX?

--	--	--	--
 - **MOVZX BX, AL** ¿EBX?

--	--	--	--



Juego de instrucciones

- Instrucciones aritméticas
 - **ADD** (suma)
 - Sintaxis: **ADD {R|M}, {R|M|I}**
 - Ejemplo: **ADD AL, BL; AL <- AL + BL**
 - **SUB** (resta)
 - Sintaxis: **SUB {R|M}, {R|M|I}**
 - Ejemplo: **SUB EAX, ECX; EAX <- EAX - ECX**
 - **DEC** (decremento)
 - Sintaxis: **DEC {R|M}**
 - Ejemplo: **DEC AX; AX <- AX - 1**
 - **INC** (incremento)
 - Sintaxis: **INC {R|M}**
 - Ejemplo: **INC ESI; ESI <- ESI + 1**



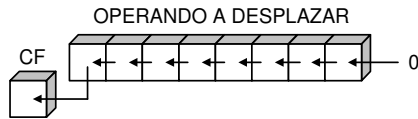
Juego de instrucciones

- Instrucciones lógicas
 - **AND** (operación Y lógica)
 - Sintaxis: **AND {R|M}, {R|M|I}**
 - Ejemplo: **AND AL, BL; AL <- AL AND BL**
 - **OR** (operación O lógica)
 - Sintaxis: **OR {R|M}, {R|M|I}**
 - Ejemplo: **OR EAX, ECX; EAX <- EAX OR ECX**
 - **XOR** (operación XOR lógica)
 - Sintaxis: **XOR {R|M}, {R|M|I}**
 - Ejemplo: **XOR AX, DX; AX <- AX XOR DX**
 - **NOT** (operación de negación lógica)
 - Sintaxis: **NOT {R|M}**
 - Ejemplo: **NOT ESI; ESI <- ~ESI**



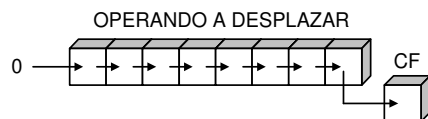
Juego de instrucciones

- Instrucciones de desplazamiento para magnitudes SIN signo
 - **SHL (SHift Left: desplazamiento de bits a la izquierda)**
 - Sintaxis: **SHL {R|M}, {CL|I8}**
 - Ejemplo: **SHL AL, 1**



Desplazar n bits a la izquierda es lo mismo que multiplicar por 2^n

- **SHR (SHift Right: desplazamiento de bits a la derecha)**
 - Sintaxis: **SHR {R|M}, {CL|I8}**
 - Ejemplo: **SHR EAX, 4**

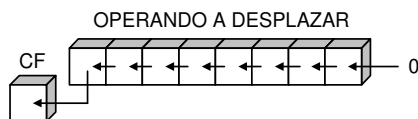


Desplazar n bits a la derecha es lo mismo que dividir entre 2^n



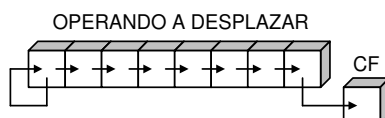
Juego de instrucciones

- Instrucciones de desplazamiento para magnitudes CON signo
 - **SAL (Shift Arithmetic Left: desplazamiento aritmético de bits a la izquierda)**
 - Es igual al **SHL**
 - Sintaxis: **SAL {R|M}, {CL|I8}**
 - Ejemplo: **SAL ECX, 2**



Desplazar n bits a la izquierda es lo mismo que multiplicar por 2^n

- **SAR (Shift Arithmetic Right: desplazamiento aritmético de bits a la derecha)**
 - Sintaxis: **SAR {R|M}, {CL|I8}**
 - Ejemplo: **SAR AL, CL**



Desplazar n bits a la derecha es lo mismo que dividir entre 2^n



Juego de instrucciones

- Ejercicios

- Sabiendo que el valor del **EAX** es 4, ¿cuál será su valor tras ejecutar la instrucción **SHL EAX, 2**?

EAX:

- Sabiendo que el valor del **EAX** es -8, ¿cuál será su valor tras ejecutar la instrucción **SAR EAX, 3**?

EAX:



Juego de instrucciones

- Instrucciones de control de flujo

- **JMP** (salto incondicional)

- Sintaxis: **JMP etiqueta**
JMP {R32|M32}

- Ejemplo:
JMP Salto
MOV EBX, EAX
MOV [EBX], DWORD PTR 34h

Salto:
INC EDX

- Objetivo:
 - **JMP etiqueta** realiza un salto incondicional a la instrucción que hace referencia la etiqueta (salto relativo)
 - **JMP {R32|M32}** realiza un salto incondicional al valor contenido en el registro o a la dirección de memoria indicada (salto absoluto)
- Codificación: la instrucción **JMP etiqueta** se codifica como **JMP Inmediato**. El valor inmediato se codifica con 8 bits si el desplazamiento relativo se encuentra en el intervalo [-128, 127]; si no, se codifica con 32 bits



Juego de instrucciones

- Ejercicio

- Dada la siguiente secuencia de instrucciones:

```
JMP Salto
MOV EBX, EAX
MOV [EBX], DWORD PTR 34h
Salto:
INC EDX
```

- Se sabe:

- El código de instrucción del **JMP Salto** es **EBh** más la codificación del salto relativo.
- La codificación de **MOV EBX, EAX** ocupa 2 bytes
- La codificación de **MOV [EBX], DWORD PTR 34h** ocupa 6 bytes
- La codificación de **INC EDX** ocupa 1 byte

- Determinar la codificación de **JMP Salto**



Juego de instrucciones

- Instrucciones de control de flujo

- **Jcondición** (salto condicional)

- Sintaxis: **Jcondición etiqueta**
Jcondición {R32|M32}

- Ejemplo: **Salto1:**

```
JZ Salto2
MOV EBX, EAX
MOV [EBX], DWORD PTR 34h
Salto2:
JAE Salto1
```

- Objetivo:

- **Jcondición etiqueta** realiza un salto condicional a la instrucción que hace referencia la etiqueta (salto relativo)
- **Jcondición {R32|M32}** realiza un salto condicional al valor contenido en el registro o a la dirección de memoria indicada (salto absoluto)

- Codificación: la instrucción **Jcondición etiqueta** se codifica como **Jcondición Inmediato**. El valor inmediato se codifica con 8 bits si el desplazamiento relativo se encuentra en el intervalo [-128, 127]; si no, se codifica con 32 bits



Juego de instrucciones

- Saltos condicionales basados directamente en el registro de estado:

Salto	Salta si:	Salto	Salta si:
JO	OF = 1	JNO	OF = 0
JC	CF = 1	JNC	CF = 0
JZ	ZF = 1	JNZ	ZF = 0
JS	SF = 1	JNS	SF = 0

- Saltos condicionales NO basados en el registro de estado:

JCXZ \Rightarrow Salta si CX = 0

JECXZ \Rightarrow Salta si ECX = 0



Juego de instrucciones

- Símbolos empleados en instrucciones de salto condicional para relaciones aritméticas:

Letra	Significado		Observaciones
J	Jump	(Saltar)	
A	Above	(mayor que)	Para números sin signo
B	Below	(menor que)	Para números sin signo
G	Greater than	(mayor que)	Para números con signo
L	Less than	(menor que)	Para números con signo
E	Equal	(igual)	
N	Not	(no)	



Juego de instrucciones

- Saltos condicionales para relaciones aritméticas:

Comparación sin signo	Salta si:	Condición de salto	Comparación con signo	Salta si:
JE	$ZF = 1$	=	JE	$ZF = 1$
JNE	$ZF = 0$	≠	JNE	$ZF = 0$
JA JNBE	$ZF = 0$ y $CF = 0$	>	JG JNLE	$ZF = 0$ y $SF = OF$
JAE JNB	$CF = 0$	≥	JGE JNL	$SF = OF$
JB JNAE	$CF = 1$	<	JL JNGE	$SF \neq OF$
JBE JNA	$ZF = 1$ ó $CF = 1$	≤	JLE JNG	$ZF = 1$ ó $SF \neq OF$



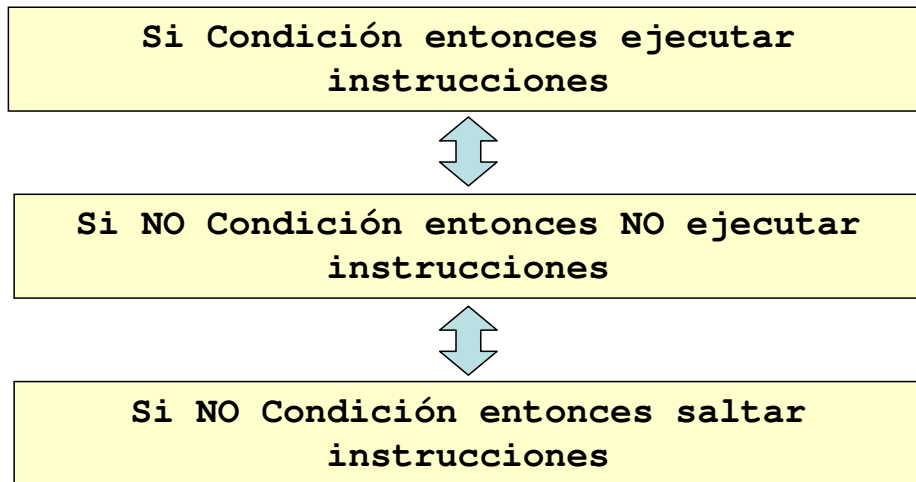
Juego de instrucciones

- Instrucción de comparación
 - **CMP** (comparación)
 - Sintaxis: **CMP** {R|M}, {R|M|I}
 - Ejemplo: **CMP AL, BL**
 - Objetivo: compara dos valores restándolos. No almacena el resultado pero modifica los bits del registro de estado
- Implementación de sentencias condicionales:
 1. Se ejecuta una instrucción de comparación sobre los operandos a comparar (**CMP**)
 2. Se ejecuta la instrucción de salto condicional para relaciones aritméticas según el tipo de comparación deseada (**JA**, **JAE**, **JGE**, etc.)



Juego de instrucciones

- Implementación de sentencias condicionales
 - Se realiza la siguiente transformación para minimizar el número de instrucciones



Juego de instrucciones

- Ejercicio
 - **AL** y **BL** contienen dos número enteros. Implementar el código necesario para incrementar **AX** cuando el número almacenado en **AL** sea mayor que el almacenado el **BL**

INC AX

- Interpretar la comparación si el valor almacenado en AL es FFh y el valor almacenado en BL es 1
- ¿Qué hubiera ocurrido si la instrucción de salto hubiera sido JBE?



Juego de instrucciones

- Ejercicio
 - **EAX** contiene un número natural. Implementar el código necesario para sumar 2 al registro **ECX** cuando el número almacenado en **EAX** sea menor o igual a 25

```
ADD ECX, 2
```



Juego de instrucciones

- Instrucción para la implementación de bucles
 - **LOOP** (iteración en función de **ECX**)
 - Sintaxis: **LOOP etiqueta**
 - Ejemplo: **MOV ECX, 10**
bucle:
INC EAX
LOOP bucle
 - Objetivo: implementar bucles. Realiza dos acciones: decrementa el registro **ECX** (**DEC ECX**) y salta a la etiqueta si el bit de Zero está a cero (**JNZ etiqueta**).

```
MOV ECX, NumeroDeIteraciones
```

Etiqueta:

```
INS1
```

```
INS2
```

```
...
```

```
LOOP Etiqueta
```

Se ejecutarán tantas veces como indique el valor con el que se ha cargado **ECX**



Ejercicio de procesamiento de listas

- Ejercicio

- Realizar un programa que procese una lista de números enteros de tipo byte definidos en la sección de datos del programa.
- Cada número se debe convertir a 32 bits para luego determinar si es positivo o negativo. Si es positivo se debe almacenar en una lista de positivos. Si es negativo se debe almacenar en una lista de negativos.
- Se debe contar cuántos números positivos y negativos hay. Estos dos valores se almacenarán en dos posiciones de memoria



Ejercicio de procesamiento de listas


```
.386
.MODEL FLAT, STDCALL
ExitProcess PROTO, dwExitCode:DWORD

.DATA
    Lista      DB 1, -1, 4, 12
    ListaPos   DD 4 DUP(0)
    ListaNeg   DD 4 DUP(0)
    NumPos     DB 0
    NumNeg     DB 0

.CODE

Inicio:
    XOR EBX, EBX
    XOR ESI, ESI
    XOR EDI, EDI
    MOV ECX, 4

Bucle:
    MOVSX EAX, [Lista + EBX]
```



```
    CMP EAX, 0
    JGE Positivo
    ; procesar negativo
    MOV [ListaNeg + EDI*4], EAX
    INC [NumNeg]
    INC EDI
    JMP Sigue

Positivo:
    ; procesar positivo
    MOV [ListaPos + ESI*4], EAX
    INC [NumPos]
    INC ESI

Sigue:
    ; procesar siguiente elem.
    INC EBX
    LOOP Bucle

    PUSH 0
    CALL ExitProcess

END Inicio
```



Lista ->

Indicar el contenido de la memoria tras la ejecución del programa

- Definición:
 - Almacén de información temporal
- Reserva de memoria:
 - Es reservada automáticamente. No se usa ninguna directiva para reservar espacio
- Registro puntero de pila:
 - **ESP**: apunta al último dato introducido en la pila
- Tamaño de los datos en la pila:
 - 16 ó 32 bits. Los datos de 8 bits no están permitidos
- Operaciones sobre la pila:
 - Apilar (PUSH)
 - Desapilar (POP)

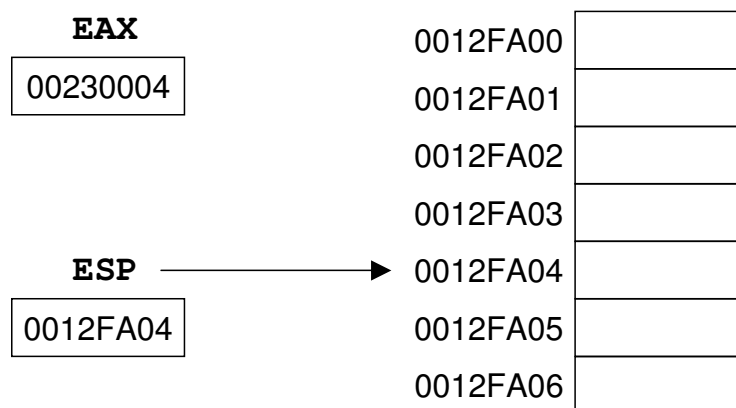
Gestión de la pila

- Instrucciones para el manejo de la pila
 - **PUSH** (apilar)
 - Sintaxis: **PUSH {R|M|I}**
 - Ejemplo:
PUSH AX
PUSH ECX
PUSH 12h
PUSH DWORD PTR [EAX]
 - Objetivo: apila datos en la pila. Realiza dos acciones: resta al registro **ESP** el tamaño del dato que se va a apilar y lo almacena en la posición de memoria apuntada por **ESP**
 - **POP** (desapilar)
 - Sintaxis: **POP {R|M}**
 - Ejemplo:
POP AX
POP ECX
POP WORD PTR [ESI]
 - Objetivo: desapila datos. Realiza dos acciones: mueve el dato almacenado en la posición de memoria apuntada por el registro **ESP** al operando destino y suma al registro **ESP** el tamaño del dato desapilado

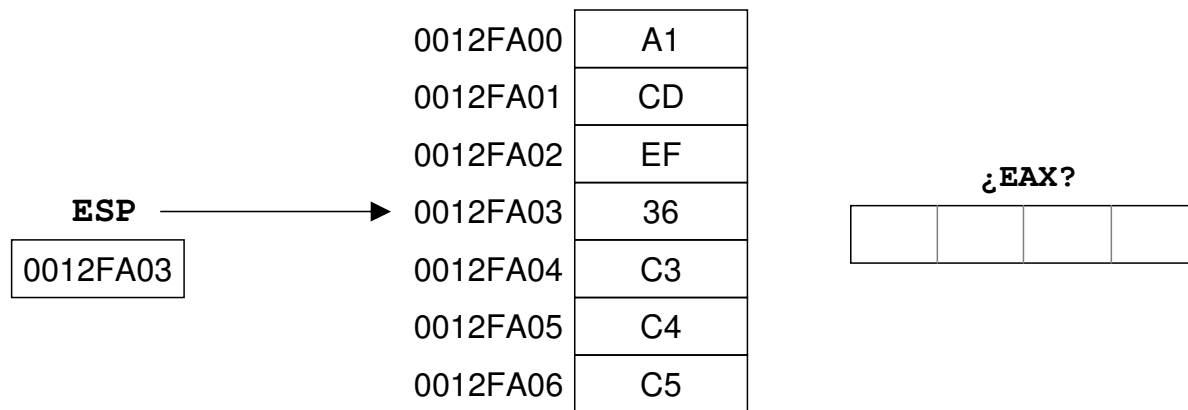


Gestión de la pila

- Ejercicio: indicar el estado de la pila tras la ejecución de la instrucción **PUSH EAX**



- Ejercicio: indicar el estado del registro **EAX** tras la ejecución de la instrucción **POP EAX**



Procedimientos y paso de parámetros

- Instrucciones para el manejo de procedimientos
 - CALL** (llamada a un procedimiento)
 - Sintaxis: **CALL {etiqueta|R32|M32}**
 - Ejemplo: **CALL Multiplica**
 - Objetivo: Realiza la llamada a un procedimiento. Realiza dos acciones: apila la dirección de retorno y salta a la primera instrucción del procedimiento
 - RET** (retorno de un procedimiento)
 - Sintaxis: **RET {I8}**
 - Ejemplo: **RET**
RET 8
 - Objetivo: Retorna de un procedimiento desapilando la dirección de retorno. Opcionalmente elimina los parámetros de un procedimiento de la pila sumándole al registro **ESP** el valor inmediato. El valor inmediato debe coincidir con el número de **bytes** que ocupan los parámetros del procedimiento en la pila

Procedimientos y paso de parámetros

- Definición de un procedimiento:

```
.386
.MODEL FLAT, STDCALL
ExitProcess PROTO, dwExitCode:DWORD
.CODE

Inicio:
    CALL ProcedimientoVacio

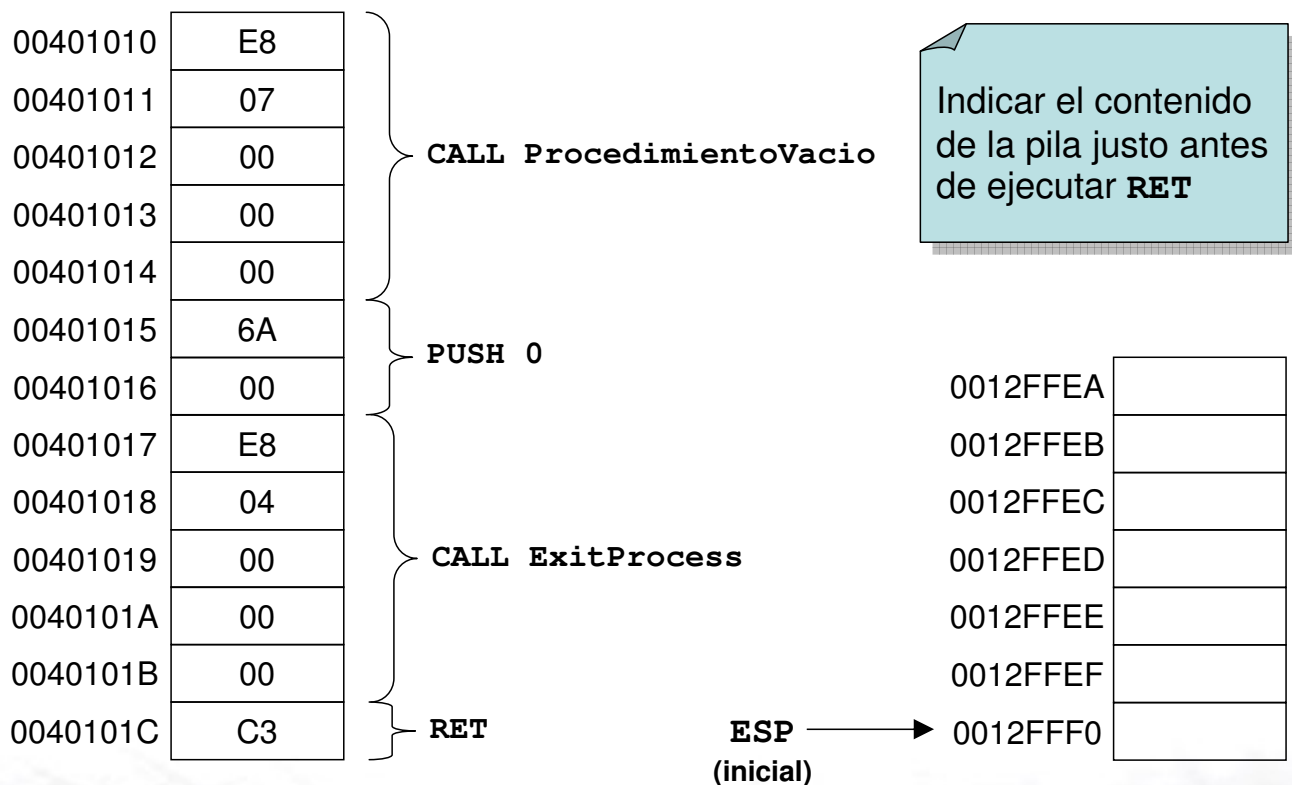
    PUSH    0
    CALL    ExitProcess

ProcedimientoVacio PROC
    RET
ProcedimientoVacio ENDP

END Inicio
```

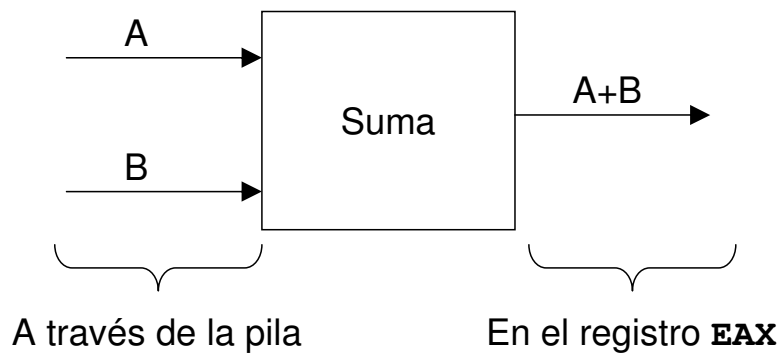


Procedimientos y paso de parámetros



Procedimientos y paso de parámetros

- Ejercicio:
 - Realizar un programa que sume dos números
 - La suma se realizará en un procedimiento que recibirá los parámetros a través de la pila y retornará el resultado en EAX



Procedimientos y paso de parámetros

```
.386
.MODEL FLAT, STDCALL
ExitProcess PROTO, dwExitCode:DWORD
.DATA
    dato1 DD 4
    dato2 DD 6
    Resultado DD 0
.CODE
Inicio:
    PUSH [dato1]
    PUSH [dato2]
    CALL Suma
    MOV [Resultado], EAX

    PUSH 0
    CALL ExitProcess

...
```

ESP →
(inicial)

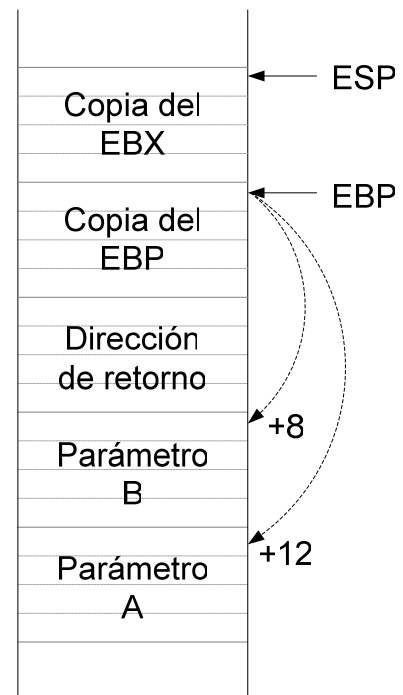
0012FFE4	
0012FFE5	
0012FFE6	
0012FFE7	
0012FFE8	
0012FFE9	
0012FFEA	
0012FFEB	
0012FFEC	
0012FFED	
0012FFEE	
0012FFEF	
0012FFF0	

Indicar el contenido de la pila justo después de ejecutar **CALL Suma**. Sabiendo que el **MOV** se almacena en las direcciones 00401028h, 00401029h, 0040102Ah, 0040102Bh, 0040102Ch.



Procedimientos y paso de parámetros

```
...  
  
Suma PROC  
    PUSH EBP  
    MOV EBP, ESP  
  
    PUSH EBX  
  
    MOV EAX, [EBP + 12]  
    MOV EBX, [EBP + 8]  
  
    ADD EAX, EBX  
  
    POP EBX  
  
    POP EBP  
    RET 8  
Suma ENDP  
  
END Inicio
```



Procedimientos y paso de parámetros

```
...  
  
Suma PROC  
    PUSH EBP  
    MOV EBP, ESP  
    PUSH EBX  
    MOV EAX, [EBP + 12]  
    MOV EBX, [EBP + 8]  
    ADD EAX, EBX  
    POP EBX  
    POP EBP  
    RET 8  
Suma ENDP  
  
END Inicio
```

Diagrama de etiquetado de las instrucciones:

- Prólogo:** Incluye las instrucciones `PUSH EBP` y `MOV EBP, ESP`.
- Salvaguarda de registros:** Incluye la instrucción `PUSH EBX`.
- Acceso a los parámetros:** Incluye las instrucciones `MOV EAX, [EBP + 12]` y `MOV EBX, [EBP + 8]`.
- Cuerpo del procedimiento:** Incluye la instrucción `ADD EAX, EBX`.
- Restauración de registros:** Incluye las instrucciones `POP EBX` y `POP EBP`.
- Epílogo:** Incluye la instrucción `RET 8`.



Ejercicio de procedimiento con paso de parámetros

- Ejercicio
 - Escribir un programa que realice la suma acumulada de una lista de números
 - Desde el programa principal se llamará a un procedimiento que reciba los siguientes parámetros a través de la pila en el siguiente orden:
 1. Dirección de la lista de números a procesar
 2. Número de elementos de la lista
 3. Dirección de la posición de memoria en la que almacenar el resultado
 - El procedimiento será llamado dos veces para procesar dos listas diferentes definidas en la sección de datos



Ejercicio de procedimiento con paso de parámetros

```
.386
.MODEL FLAT, STDCALL
ExitProcess PROTO, dwExitCode:DWORD
.DATA
    Lista1 DB 1, -1, 7
    Lista2 DB -4, 9, 15, 10
    SumaLista1 DD 0
    SumaLista2 DD 0
.CODE
Inicio:
    PUSH OFFSET Lista1
    PUSH 3
    PUSH OFFSET SumaLista1
    CALL SumaLista
```



```
PUSH OFFSET Lista2
PUSH 4
PUSH OFFSET SumaLista2
CALL SumaLista
```

```
PUSH 0
CALL ExitProcess
```

...



Ejercicio de procedimiento con paso de parámetros

...

SumaLista PROC

```
; Prólogo
PUSH EBP
MOV EBP, ESP
```

```
; Salvaguarda
PUSH ESI
PUSH EDI
PUSH EDX
PUSH EAX
PUSH ECX
```

```
; Acceso a parámetros
MOV EDI, [EBP + 16]
MOV ECX, [EBP + 12]
MOV ESI, [EBP + 8]
```

```
; Cuerpo
XOR EAX, EAX
```

```
Bucle:  MOV SX EDX, BYTE PTR [EDI]
        ADD EAX, EDX
        INC EDI
        LOOP Bucle
        MOV [ESI], EAX
```

```
; Restauración
POP ECX
POP EAX
POP EDX
POP EDI
POP ESI
```

```
; Epílogo
POP EBP
RET 12
```

SumaLista ENDP

END Inicio

itadores

Fundamentos de Computadores

Tema 6: Arquitectura Intel

65

Traducción de lenguaje de alto nivel a código máquina

- La CPU de un computador entiende un conjunto de instrucciones reducido y muy básico
- Para resolver un problema se deberá crear un programa a partir del conjunto de instrucciones que una CPU entiende
- Realizar programas utilizando el conjunto de instrucciones de una CPU tiene tres inconvenientes:
 - El programa no es portable debido a que el conjunto de instrucciones varía entre distintas CPUs
 - El número de instrucciones necesario para realizar un programa es muy elevado debido al carácter básico del conjunto de instrucciones de una CPU
 - Es mucho más difícil para los humanos
- Solución: desarrollar el programa en un lenguaje de alto nivel más cercano al usuario que a la máquina y utilizar un compilador para traducir ese programa al lenguaje que entiende la CPU
 - Es necesario un compilador de ese lenguaje para la CPU

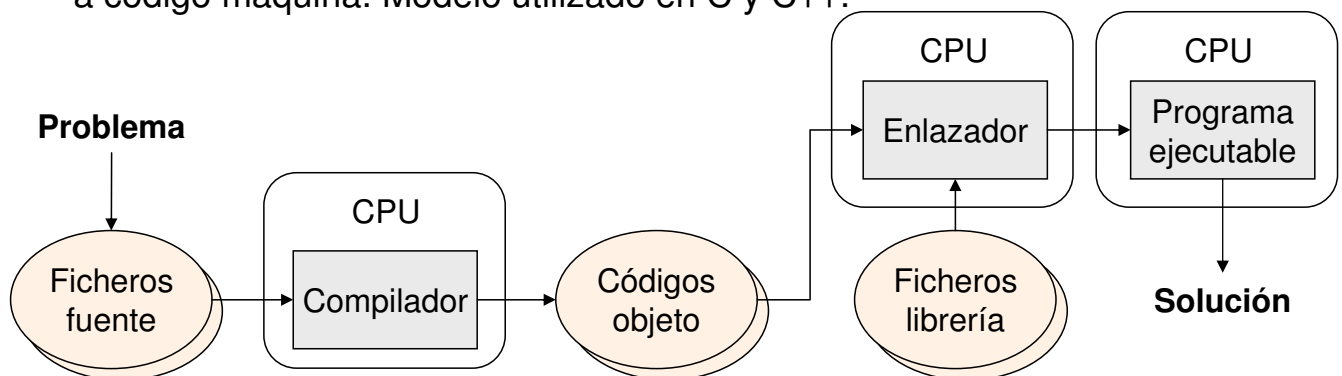
Ejemplos de lenguajes de alto nivel

- Algunos de los lenguajes de propósito general más utilizados en la actualidad son:
 - C (1970): desarrollado originalmente para implementar el sistema operativo UNIX.
 - C++ (1983): amplía el lenguaje C para poder utilizar la metodología orientada a objetos.
 - Java (1995): similar a C++ pero basado en una máquina virtual
 - C# (2000): creado por Microsoft con características similares a Java
- Otros lenguajes ganando popularidad:
 - Perl (1987)
 - PHP (1994)
 - Python (1991)
 - Ruby (1990)
- Lenguajes específicos:
 - SQL (1970): utilizado para manejar información en una base de datos



Modelo de ejecución: compilado

- **Modelo compilado:** Se traduce el programa desde el lenguaje de alto nivel a código máquina. Modelo utilizado en C y C++.

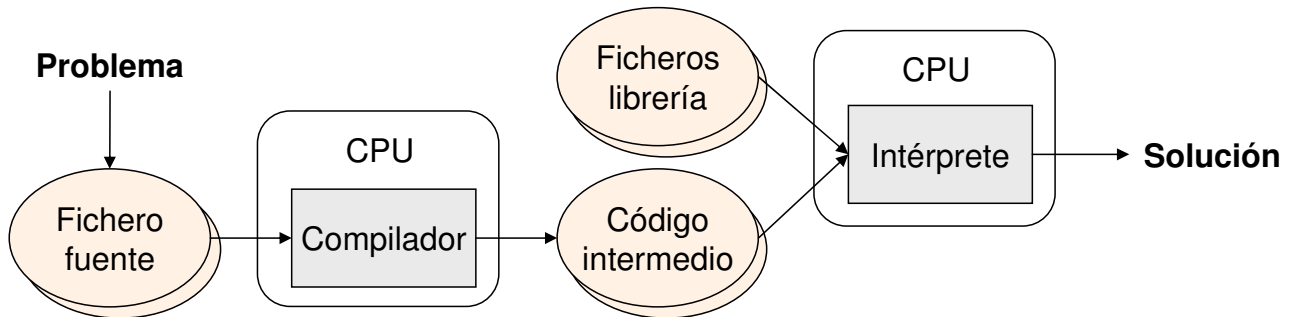


- El fichero fuente contiene un programa en lenguaje de alto nivel
- El compilador es un programa ejecutado por la CPU que realiza la traducción desde el lenguaje de alto nivel al código máquina
- El enlazador es un programa ejecutado por la CPU que combina los distintos ficheros con código máquina en un programa ejecutable
- Un programa ejecutable contiene el código máquina dentro de un formato definido por el sistema operativo sobre el que se ejecuta



Modelo de ejecución: interpretado

- **Modelo interpretado:** El programa en lenguaje de alto nivel no es traducido a código máquina sino que es interpretado por otro programa. Modelo utilizado en la mayoría de los lenguajes modernos.



- El fichero fuente contiene un programa en lenguaje de alto nivel
- El compilador es un programa ejecutado por la CPU que realiza la traducción desde el lenguaje de alto nivel a código en un lenguaje intermedio (independiente de la CPU)
- El intérprete es un programa ejecutado por la CPU que realiza las operaciones indicadas en el fichero con el código intermedio
- A veces compilador e intérprete consisten en un solo programa y el fichero intermedio sólo se almacena en memoria

Compilación de C++ en Intel

- **Compiladores más utilizados:**
 - GCC: compilador gratuito (viene incluido con el Dev-C++). Disponible en los sistemas operativos más comunes.
 - Visual C/C++: compilador desarrollado por Microsoft. Disponible sólo para Windows.
 - Intel C/C++: compilador desarrollado por Intel. Disponible para Windows y Linux
- La tarea fundamental del compilador de C++ es traducir el programa a ensamblador, lo cual incluye, entre otras, las siguientes tareas:
 - Inicialización y finalización del programa
 - Llamada a las funciones
 - Acceso a los parámetros y variables locales
 - Optimización del código generado

- Los programas en C++ comienzan por la función *main*
- Sin embargo, el compilador genera código antes (inicialización) y después (finalización) de la función *main*
- El código de inicialización se encarga principalmente:
 - Inicializar el proceso de gestión de memoria
 - Obtener los parámetros recibidos por el programa
 - Llamar a la función *main*
- El código de finalización se encarga principalmente:
 - Devolver el control al sistema operativo
- ¿Por qué en un programa en lenguaje ensamblador hay que llamar a la función **ExitProcess** al final del programa y no en uno en C++?

Compilación de C++ en Intel: llamadas a funciones

```
int Suma(int a, int b, int c)      int main()
{                                  {
    int ResSuma;                  int Res;

    ResSuma = a + b + c;          Res = Suma(4, 5, 6);
    return ResSuma;              return 0;
}
```

- Para traducir este código es necesario definir cómo se van a pasar los parámetros a la función y cómo se va a retornar el resultado
- Algunas posibles soluciones:
 - Pasar los parámetros en EAX, EBX y ECX y retornar el resultado en EDX
 - Pasar los parámetros por la pila de derecha a izquierda y retornar el resultado en EAX
 - Pasar los parámetros por la pila de izquierda a derecha y retornar el resultado en EDX
- Se han definido varios convenios: cdecl, fastcall, stdcall, thiscall

- Convención **cdecl** de llamadas a funciones en Visual C/C++
 - Los parámetros se pasan a las funciones a través de la pila de derecha a izquierda (el this se pasa el último en caso de métodos de objetos)
 - El resultado se retorna en el registro EAX
 - El código que llama a la función es responsable de limpiar la pila

```

:
:
: resultado = Suma(4, 5, 6);
:
:
:

```

Traducción

```

push    6
push    5
push    4
call    Suma
add     esp, 0Ch
mov     [ebp-4], eax

```

- Es el método utilizado por defecto en lenguaje C en Intel
- El hecho de pasar los parámetros de derecha a izquierda y de que el código que llama a la función sea responsable de limpiar la pila está relacionado con la capacidad del lenguaje C de definir funciones con un número de parámetros variable



Compilación de C++ en Intel: llamadas a funciones (fastcall)

- Convención **fastcall** de llamadas a funciones en Visual C/C++
 - Los dos primeros parámetros se pasan en ECX y EDX
 - El resto se pasa a través de la pila de derecha a izquierda
 - El resultado se retorna en el registro EAX
 - El código de la función es responsable de limpiar la pila

```

:
:
: resultado = Suma(4, 5, 6);
:
:
:

```

Traducción

```

mov     ecx, 4
mov     edx, 5
push    6
call    Suma
mov     [ebp-4], eax

```

- Es más rápido que cdecl ya que requiere menos accesos a memoria
- Una función que utilice esta convención de llamada no puede tener un número variable de parámetros



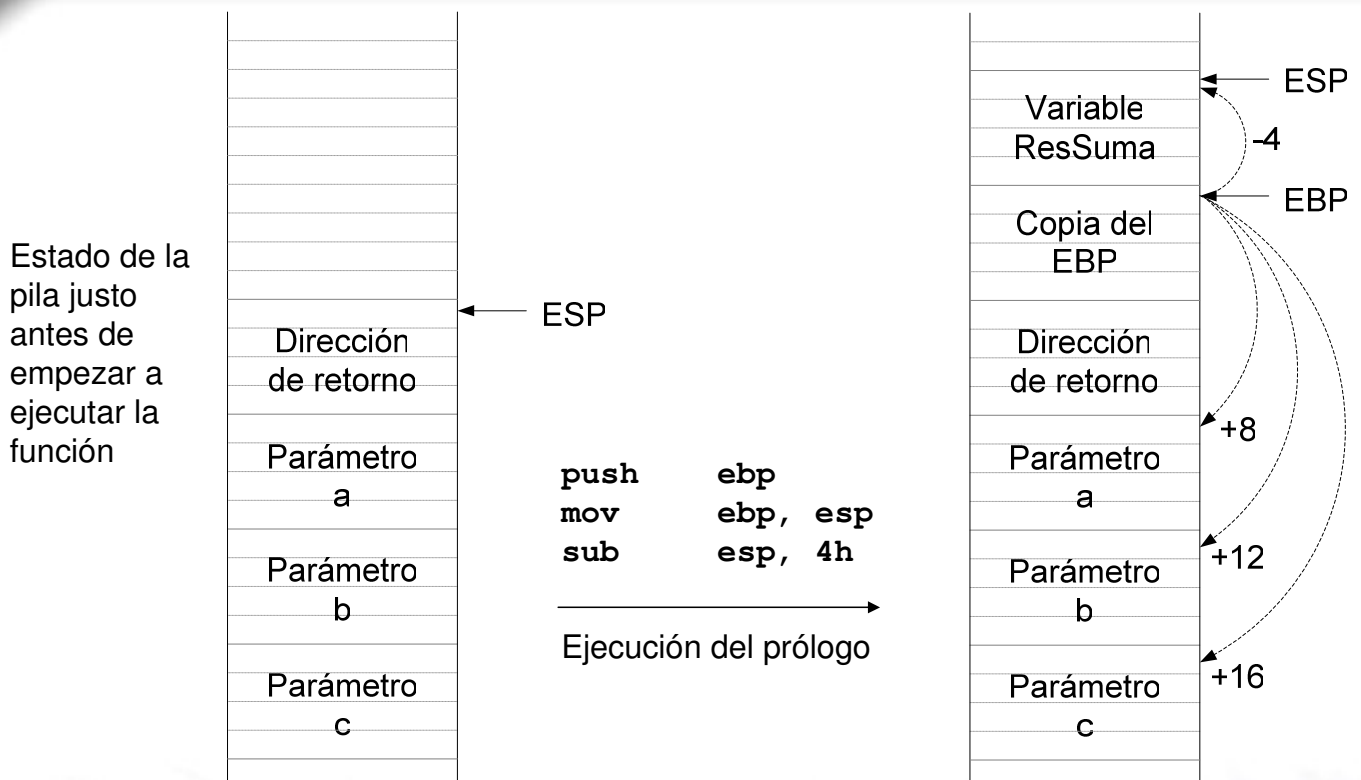
Compilación de C++ en Intel: traducción de funciones

```
int Suma(int a, int b, int c)
{
    int ResSuma;

    ResSuma = a + b + c;
    return ResSuma;
}
```

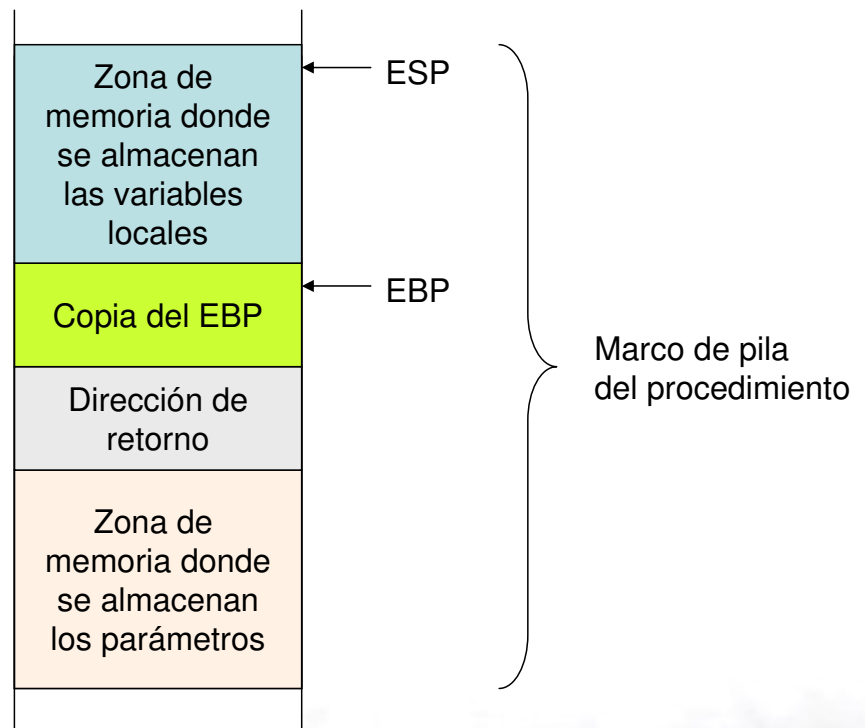
Traducción (cdecl)	{	push	ebp	}	Reserva de espacio para variables locales	}	Prólogo
		mov	ebp, esp				
		sub	esp, 4h				
		mov	eax, [ebp + 8]	}	Acceso a parámetros y cuerpo		
		add	eax, [ebp + 12]				
		add	eax, [ebp + 16]				
		mov	[ebp - 4], eax				
		mov	eax, [ebp - 4]				
		mov	esp, ebp	}	Epílogo		
		pop	ebp				
		ret					

Compilación de C++ en Intel: la pila en las funciones

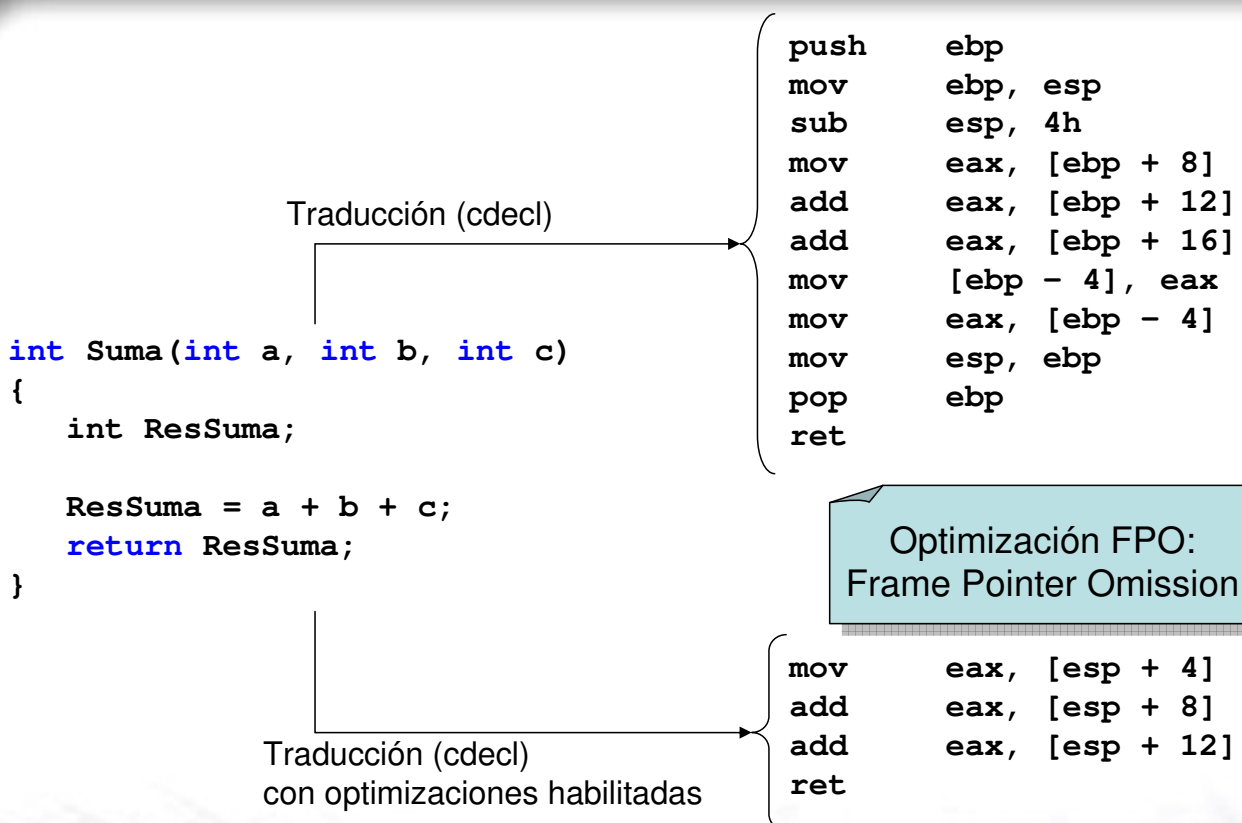


Compilación de C++ en Intel: la pila en las funciones

- En general durante la ejecución de una función la pila tiene esta estructura:



Compilación de C++ en Intel: optimizaciones



Compilación de C++ en Intel: otro ejemplo

```
int SumaVeces(int a, int b,
              int Veces)
{
    int i, Resultado;
    ① Resultado = 0;    ③    ②
    for (i = 0; i < Veces; i++)
        ④ Resultado += a + b;
    ⑤ return Resultado;
}
```

Parámetros:

a: [ebp + 8]
b: [ebp + 12]
Veces: [ebp + 16]

Variables locales:

i: [ebp - 4]
Resultado: [ebp - 8]

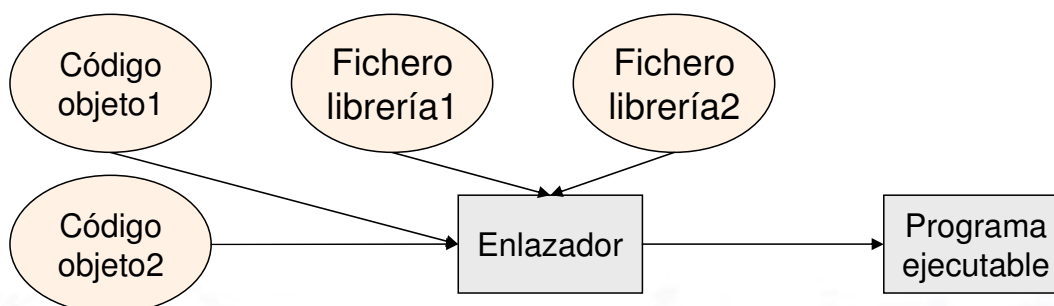
Traducción
(cdecl)

```
push    ebp
mov     ebp, esp
sub     esp, 8h
① mov     [ebp - 8], 0
mov     [ebp - 4], 0
jmp     short 4115B2
4115A9:
mov     eax, [ebp - 4]
② add     eax, 1
mov     [ebp - 4], eax
4115B2:
mov     eax, [ebp - 4]
③ cmp     eax, [ebp + 16]
jge     short 4115C8
mov     eax, [ebp + 8]
add     eax, [ebp + 12]
④ add     eax, [ebp - 8]
mov     [ebp - 8], eax
jmp     short 4115A9
4115C8:
⑤ mov     eax, [ebp - 8]
mov     esp, ebp
pop     ebp
ret
```



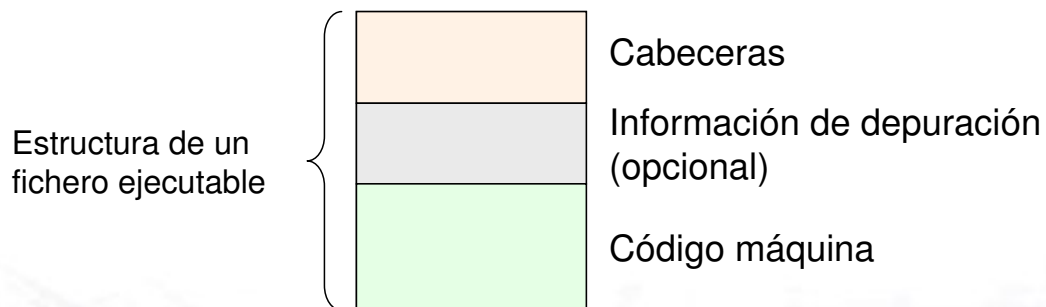
Compilación de C++ en Intel: proceso de enlazado

- Habitualmente para crear un programa se divide el código fuente en múltiples ficheros.
- El proceso de compilación genera un fichero con código objeto por cada fichero con código fuente
- El enlazador es un programa que crea, a partir de varios ficheros con código objeto, un programa ejecutable. Su principal labor es ajustar las referencias, por ejemplo la dirección de las funciones
- Una librería no es más que un fichero con una colección de código objeto de múltiples ficheros fuente.



Compilación de C++ en Intel: fichero ejecutable

- Un fichero ejecutable contiene código máquina en un formato especificado por el sistema operativo en el que se vaya a ejecutar el programa. Ejemplos:
 - Windows: formato PE
 - Linux: formato ELF
- El fichero ejecutable contiene una serie de cabeceras que indican cómo se debe cargar el programa en memoria
- A un fichero ejecutable se le puede desensamblar, es decir, determinar el conjunto de instrucciones que contiene



Compilación de C++ en Intel: ejemplo de fichero ejecutable

Fichero fuente

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int x;

    cin >> x;

    if (x > 10)
        cout << "Mayor" << endl;
    else
        cout << "Menor o igual" << endl;

    return EXIT_SUCCESS;
}
```

Compilación y enlazado

Fichero ejecutable

```
...
cc cc cc cc f3 ab 8d 45 f8 50
b9 c0 b7 45 00 e8 6a e7 ff ff
83 7d f8 0a 7e 20 68 d3 a4 41
00 68 d0 20 45 00 68 08 b7 45
00 e8 09 e8 ff ff 83 c4 08 8b
c8 e8 2c e8 ff ff eb 1e 68 d3
a4 41 00 68 c8 20 45 00 68 08
b7 45 00 e8 e9 e7 ff ff 83 c4
08 8b c8 e8 0c e8 ff ff 33 c0
52 8b cd 50 8d 15 47 c3 41 00
...
```

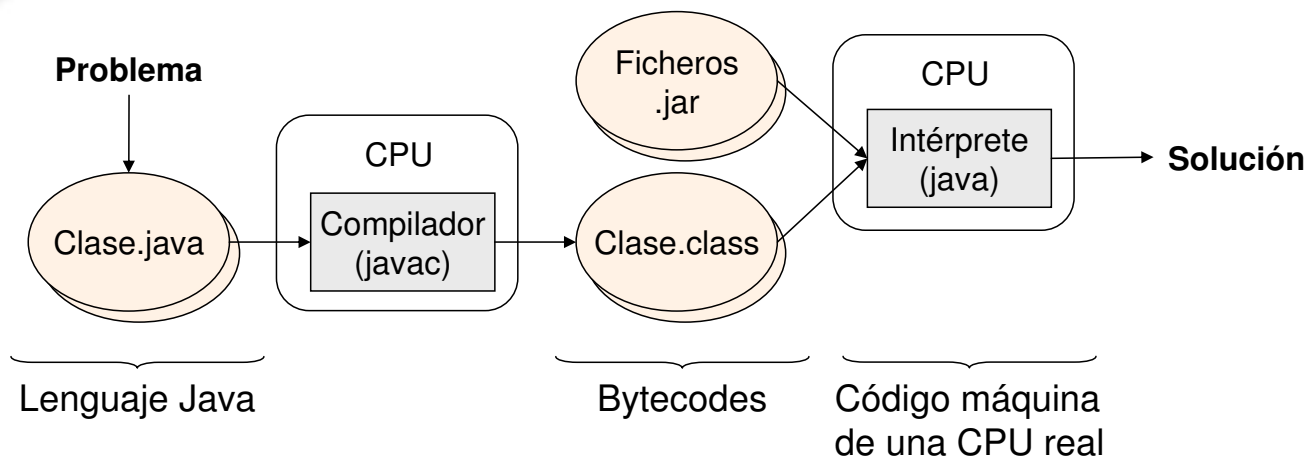
```
cmp dword ptr [ebp-4], 0A
jle short 0041C302
```

Codificación

```
83 7D F8 0A
7E 20
```



Ejecución de programas Java



- El código fuente está en lenguaje Java
- El compilador lo traduce al código máquina de una máquina virtual, la JVM (Java Virtual Machine)
- A las instrucciones de la JVM se las denomina bytecodes
- Los intérpretes de Java permiten emular una JVM en una máquina (CPU) real
- Al final, se ejecutan instrucciones de la máquina real (por ejemplo, de Intel) que tienen el mismo resultado que ejecutar instrucciones de la JVM



Ejecución de programas Java

Fichero fuente .java

```
import java.util.Scanner;

public class prueba {

    public static void main(String[] args) {
        int x;
        Scanner in = new Scanner(System.in);
        x = in.nextInt();

        if (x > 10)
            System.out.println("Mayor");
        else
            System.out.println("Menor o igual");
    }
}
```

.class (desensamblado)

ILOAD_1	mete x en la pila
BIPUSH 10	mete 10 en la pila
IF_ICMPLE L3	compara x y 10 y salta ; si $10 \leq x$

compilación

Fichero .class
(bytecodes)

```
...
bb 00 10 59 b2 00 12 b7 00 18 4
1b 10 0a a4 00 0e b2 00 1f 12 2
...
```

No es código
máquina de Intel



Introducción a la arquitectura x86-64

- Origen:
 - Desarrollada por AMD (Advanced Micro Devices)
 - Se crea para competir con la arquitectura IA-64 desarrollada por Intel
 - IA-64 fracasa y x86-64 es adoptada mayoritariamente
 - Finalmente es copiada por Intel (el original se convierte en el clon)
- Principales características:
 - Es compatible con la arquitectura IA-32
 - Los registros de propósito general se expanden a 64 bits
 - Permite realizar operaciones aritméticas y lógicas entre datos de 64 bits
 - El tamaño de una dirección se expande a 64 bits, lo cual implica un espacio de direcciones de 2^{64}
 - Además añade 8 nuevos registros de propósito general



Introducción a la arquitectura x86-64

- Ventajas (la falsa):
 - 64 es el doble de 32 por tanto un procesador de 64 bits es dos veces mejor que uno de 32
- Ventajas (las verdaderas):
 1. Se puede direccionar más memoria
 2. Se puede operar con datos de mayor tamaño
 3. El incremento del número de registros de propósito general permite que los programas reduzcan el número de accesos a memoria, siendo por tanto su ejecución más rápida
- Aplicabilidad de las ventajas:
 - La 1 para programas que necesiten grandes cantidades de memoria tales como sistemas de gestión de bases de datos o juegos
 - La 2 para aplicaciones que necesiten operar con datos grandes tales como sistemas multimedia
 - **La 3 para todos los programas**



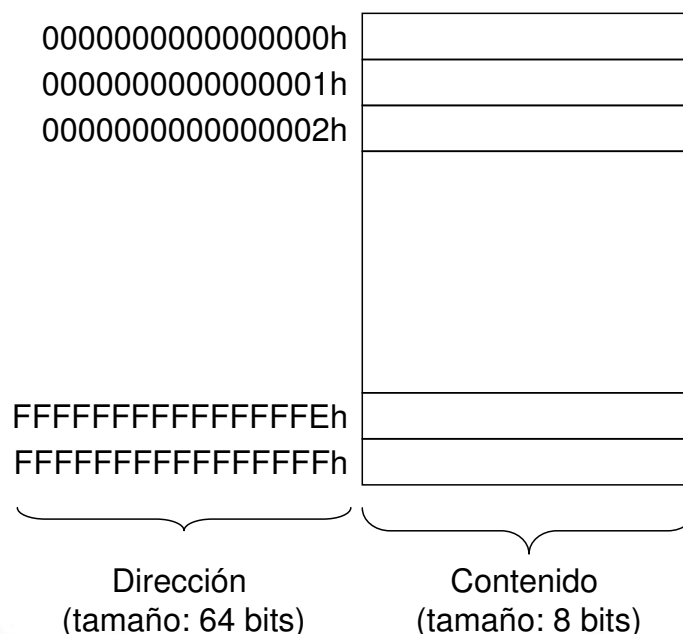
Introducción a la arquitectura x86-64

	IA-32	X86-64
Ancho de la arquitectura (tamaño de los operandos)	32	64
Ancho de las posiciones de memoria	8 (8 bits = 1 byte)	8 (8 bits = 1 byte)
Ancho de las direcciones de memoria	32	64
Dimensiones del espacio de direcciones	4 GB	16 EB (EB = ExaByte = 2^{60})



Introducción a la arquitectura x86-64

- Modelo de memoria:
 - El modelo segmentado ya no existe, sólo el modelo plano



Introducción a la arquitectura x86-64

- Tipos de datos:
 - Aparece el tipo de dato de 64 bits (quad word)
- Registros:
 - Los registros se expanden a 64 bits

	63	32	31	0
RAX				EAX
RBX				EBX
RCX				ECX
RDX				EDX
RSI				ESI
RDI				EDI
RBP				EBP
RSP				ESP

Compatible con IA-32



Área de Arquitectura y Tecnología de Computadores
Departamento de Informática de la Universidad de Oviedo

Fundamentos de Computadores
Tema 6: Arquitectura Intel

89

Introducción a la arquitectura x86-64

- Se añaden 8 nuevos registros de propósito general

	63	32	31	0
R8				R8D
R9				R9D
R10				R10D
R11				R11D
R12				R12D
R13				R13D
R14				R14D
R15				R15D

- Se puede acceder al byte menos significativo de los nuevos registros como **R?B** ó a la palabra menos significativa como **R?W**

- El registro puntero de instrucción también se expande

RIP		EIP
------------	--	------------

- El registro de estado (EFLAGS) permanece con 32 bits



Área de Arquitectura y Tecnología de Computadores
Departamento de Informática de la Universidad de Oviedo

Fundamentos de Computadores
Tema 6: Arquitectura Intel

90

- Convención de llamada a las funciones:
 - Los primeros cuatro parámetros se pasan en los registros **RCX**, **RDX**, **R8** y **R9**
 - El resto de parámetros se pasan a través de la pila
 - El valor de retorno se pone en el registro **RAX**
 - El código que llama a la función es responsable de limpiar la pila

```
.  
.  
.  
resultado = Suma(4, 5, 6);  
.  
.  
.
```

Traducción →

```
{  
  mov    rcx, 4  
  mov    rdx, 5  
  mov    r8,  6  
  call   Suma  
}
```