

Análisis de vulnerabilidades

Temario día

1

DIA 1

- 1. Conceptos básicos
- 2. Lenguaje ensamblador

Evaluación

- Examen Práctico 30%
- Examen Teórico 20%
- Tareas (a mano) 20%
- Prácticas 30%

Introduccion

Objetivos

- Exposición de técnicas para realizar:
- análisis de vulnerabilidades
- ingeniería inversa
- desarrollo de exploits

Porque VA y RE

- **No puedes protegerte de algo que no conoces.**
- **Entender cómo funcionan los programas.**
- **Habilidad no muy común**
- **Mercado en crecimiento.**

Therac-25

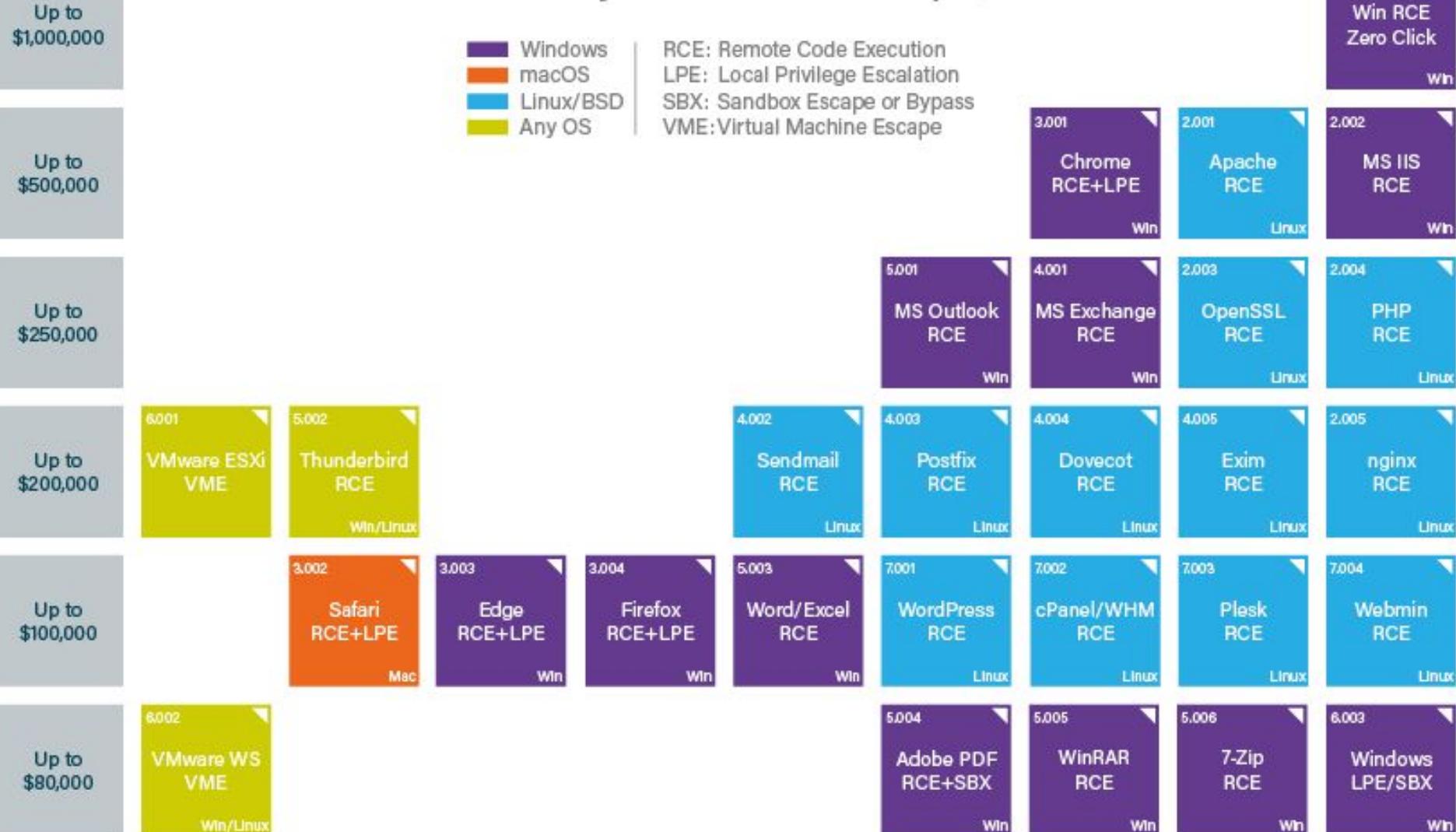
Dates: 1985-1987

Cause: "race condition"

Consequences:
at least 5 patients
died, others were
seriously injured



ZERODIUM Payouts for Desktops/Servers*



<https://www.zerodium.com/program.html>

ZERODIUM Payouts for Mobiles*

Up to
\$2,000,000

Up to
\$1,500,000

Up to
\$1,000,000

Up to
\$500,000

Up to
\$200,000

RJB: Remote Jailbreak with Persistence
 RCE: Remote Code Execution
 LPE: Local Privilege Escalation
 SBX: Sandbox Escape or Bypass

iOS
 Android
 Any OS

1.001
 iPhone RJB
 Zero Click
 iOS

1.002
 iPhone RJB
 iOS

2.001
 WhatsApp
 RCE+LPE
 iOS / Android

2.002
 SMS/MMS
 RCE+LPE
 iOS / Android

2.003
 iMessage
 RCE+LPE
 iOS

2.004
 WeChat
 RCE+LPE
 iOS / Android

2.005
 FB Messenger
 RCE+LPE
 iOS / Android

2.006
 Signal
 RCE+LPE
 iOS / Android

2.007
 Telegram
 RCE+LPE
 iOS / Android

2.008
 Email App
 RCE+LPE
 iOS / Android

3.001
 Chrome
 RCE+LPE
 Android

3.002
 Safari
 RCE+LPE
 iOS

4.001
 Baseband
 RCE+LPE
 iOS / Android

5.001
 LPE to
 Kernel / Root
 iOS / Android

2.009
 Media Files
 RCE+LPE
 iOS / Android

2.010
 Documents
 RCE+LPE
 iOS / Android

3.003
 SBX
 for Chrome
 Android

3.004
 Chrome RCE
 w/o SBX
 Android

3.005
 SBX
 for Safari
 iOS

3.006
 Safari RCE
 w/o SBX
 iOS

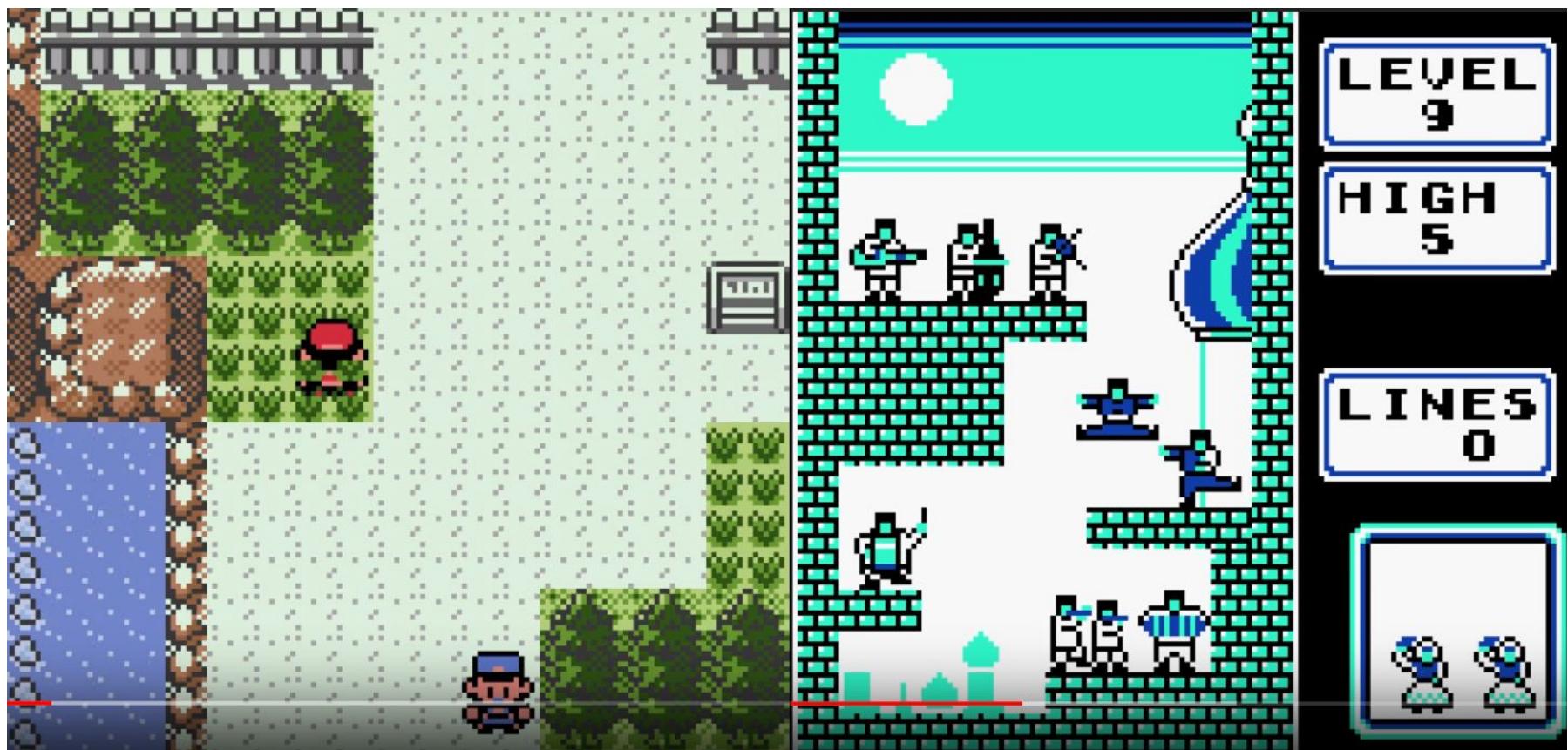


Explotación Binaria

- Los fallos pueden convertirse en:
- Tomar el control del servidor
- Obtener información sensible
- Instalar malware
- Dañar activos de información



Explotación Binaria



Pokémon: Yellow Version "arbitrary code execution" by MrWint
<https://youtu.be/zZCqoHHtovQ?t=108>

Historia

1972: El plan de estudios de la USAF describe los overflows

1985: Primer virus “Brain”. Infecta sector de arranque.

1988: Se propaga morris debido a un overflow

1996: Se publica ‘Smash the stack’ por aleph-1

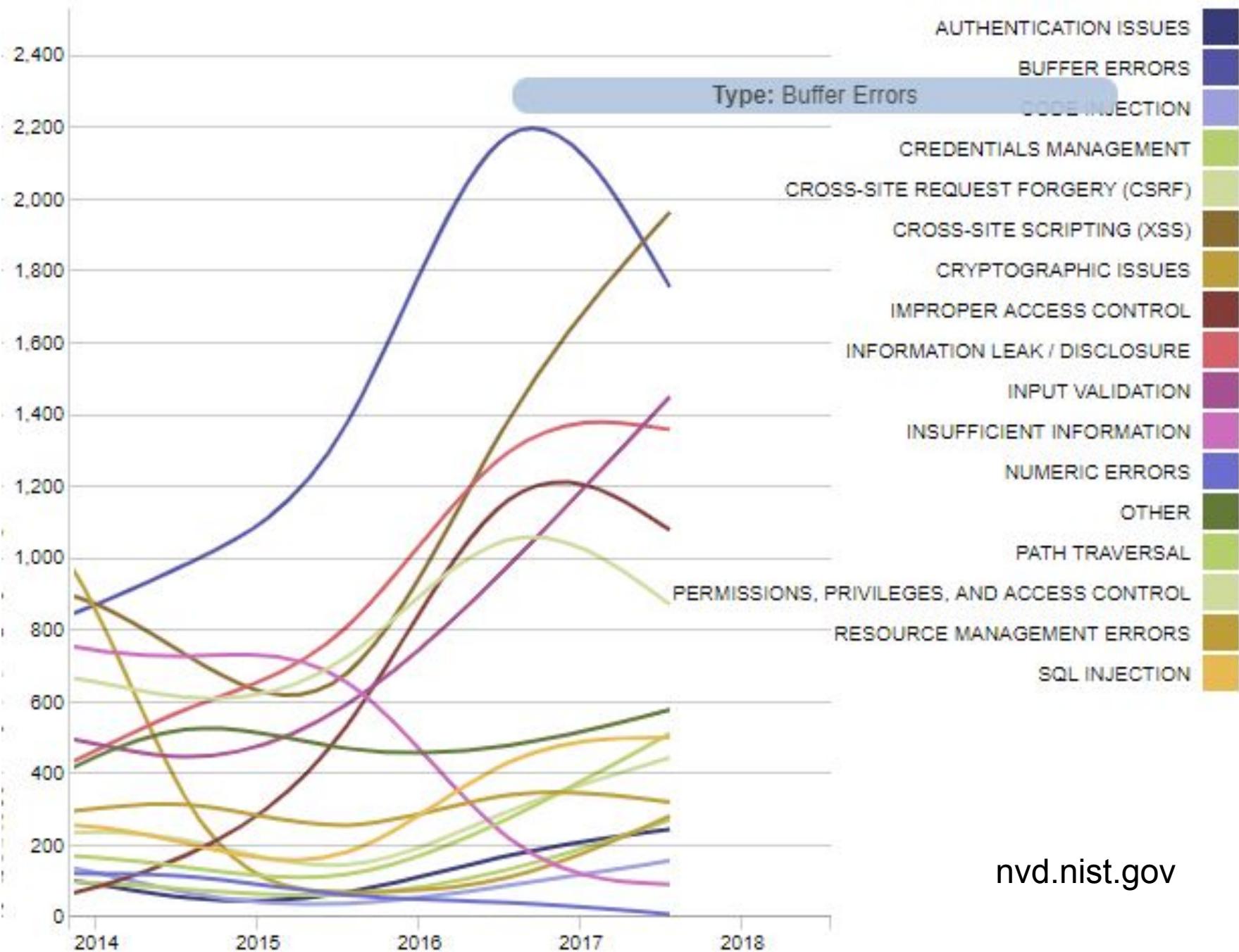
2001: Code red infecta millones de servidores

2004: XP SP2 es liberado

2007: JailBreak al iphone

2008: Stuxnet

Análisis de vulnerabilidades



nvd.nist.gov

1. Conceptos básicos

Conceptos básicos

- **Incidente de seguridad:** Es cualquier evento no esperado que afecte la continuidad del negocio y atente contra la confidencialidad, integridad o disponibilidad de la información
- **Riesgo:** Posibilidad de un incidente, que puede resultar en daños o pérdida de activos de una organización
- **Amenaza:** Causa potencial de un incidente, que puede resultar en daños o pérdida de activos de una organización

Conceptos básicos

- **Vulnerabilidad:** fallo o hueco de seguridad
 - Desbordamiento de buffer
 - Escalamiento de privilegios
- **Exploit:** es un código o técnica que amenaza con tomar ventaja de una vulnerabilidad.
 - Para ejecutar comandos
 - Ganar acceso a un sistema
 - Escalar privilegios

Conceptos básicos

- **Payload:** Son las acciones posteriores a explotar una vulnerabilidad. Generalmente son tareas automatizadas para un determinado objetivo. (Ejemplos: crear usuarios, modificar archivos, obtener una terminal remota)

Conceptos básicos

- **Compilador:** Herramienta que traduce de un lenguaje a otro. El resultado de este código máquina es llamado archivo objeto.

Conceptos básicos

- **Lenguaje de alto nivel:** Lenguaje diseñado para facilitar la comprensión por parte de los humanos. Es convertido a código máquina por un compilador.
- **Lenguaje de bajo nivel:** Es la versión legible del conjunto de instrucciones (mnemónicos) de una arquitectura de computadoras, conocido como lenguaje ensamblador.

Conceptos básicos

- **Código máquina:** Consiste en opcodes (dígitos en hexadecimal) que le indican al procesador lo que debe realizar. Se crea cuando un programa en lenguaje de alto nivel es compilado.

Creación de un ejecutable

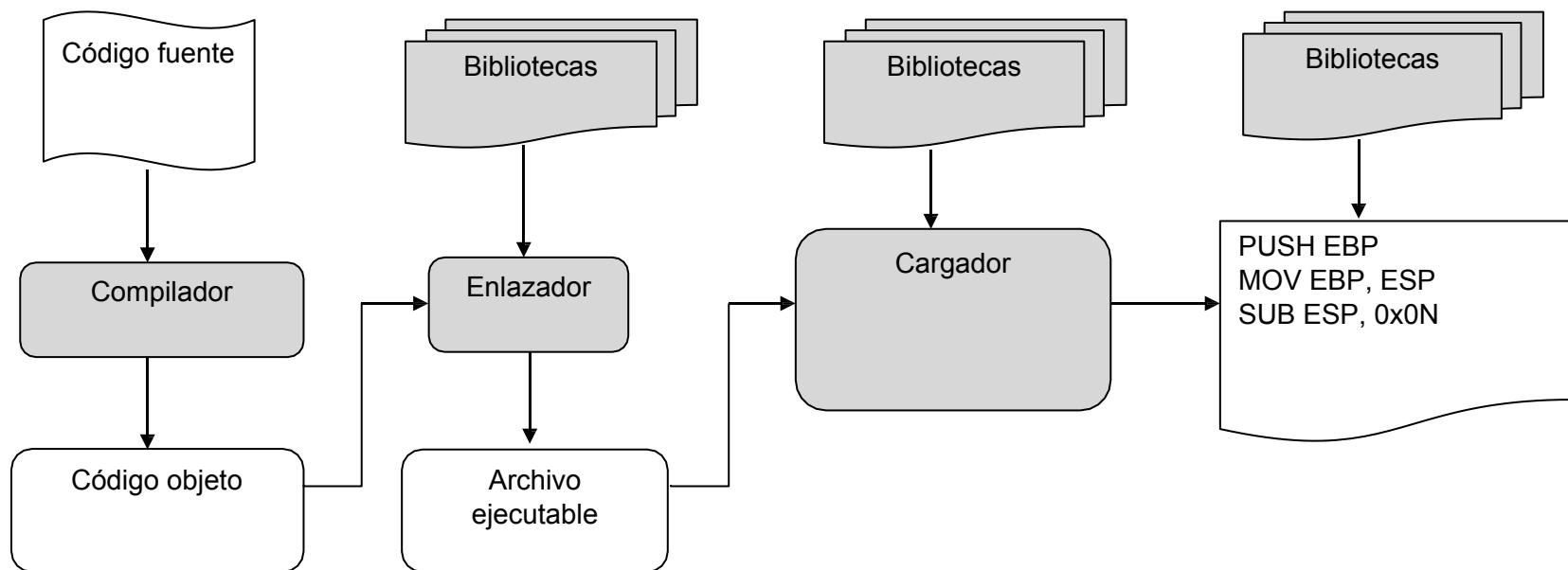
- En primera instancia el código fuente es traducido por un compilador a código objeto.
- Posteriormente el enlazador proporciona las bibliotecas requeridas, se puede ligar de dos maneras: estáticamente o dinámicamente.
 - Ligado estático: las bibliotecas requeridas se colocan como parte del ejecutable.
 - Ligado dinámico: las bibliotecas requeridas se satisfacen al momento de la ejecución.

Creación de un ejecutable

- El resultado del enlazador es un archivo ejecutable.
- Finalmente, cuando el archivo ejecutable es utilizado, éste es colocado en memoria por el cargador, el cual se encarga de reservar memoria y de llamar al enlazador dinámico para resolver las dependencias restantes.
- Como consecuencia se carga el *opcode* necesario para la ejecución del programa.

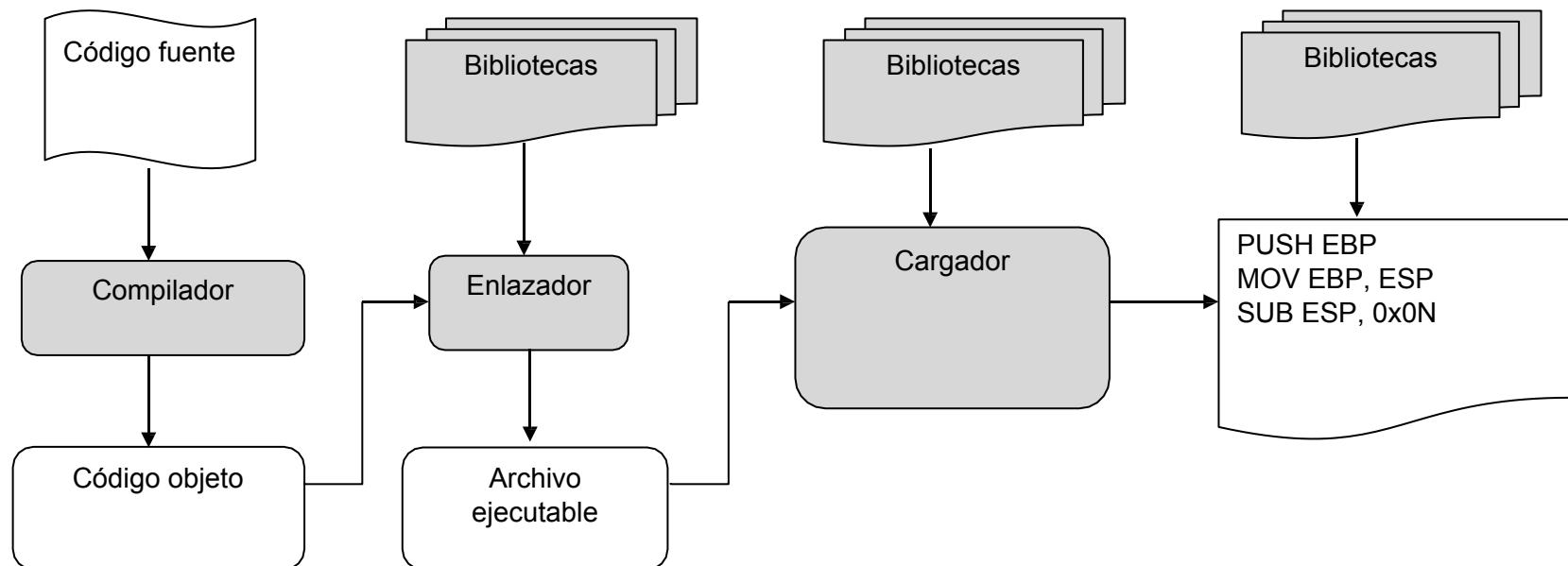
Creación de un ejecutable

- El proceso para convertir código fuente a un ejecutable, tiene que pasar por distintos componentes como se muestra en la siguiente imagen:



Creación de un ejecutable

- nasm -f elf32 -o \$fuente.obj ;
- ld -m elf_i386 -o \$fuente \$fuente.obj
file hello_world.obj
objdump -m Intel -h hello_world.obj



2. Lenguaje ensamblador

Lenguaje ensamblador

- Es el lenguaje de bajo nivel por excelencia.
- Las instrucciones en lenguaje ensamblador son conocidas como mnemónicos.
- Un ensamblador traduce las instrucciones a código máquina.

Sintaxis básica

- Una instrucción en ensamblador está compuesta por los siguientes elementos:

Sintaxis básica			
[Etiqueta:]	Mnemónico	[Operando(s)] (Destino,Origen)	[; Comentario]

Sintaxis básica

- **Mnemónico.** Mnemónico de la instrucción.
- **Operandos.** Contiene los datos requeridos por la instrucción, pueden ser valores constantes, provenir de direcciones en memoria o de registros. El número de operandos pueden ser 0, 1 o 2 (RET; INC EAX; MOV EAX, 1H respectivamente).

Sintaxis básica

- **Etiqueta.** Sirve como punto de control para el flujo del programa. Debe comenzar con un carácter alfabético o un punto, el resto de la etiqueta puede contener letras, números o signos especiales [-\$.@%].
- **Comentario.** Información adicional para propósitos de documentación. Cualquier texto posterior al signo ';' será excluido de la compilación.

Acceso a datos

- Para hacer referencia a los datos se pueden utilizar:
 - Valores inmediatos. Un dato es almacenado en uno de los registros de propósito general. (MOV EAX, 0x01)
 - Registros. El dato contenido en un registro es almacenado en otro registro. (MOV EAX, EBX).

```
section .text
global _start
_start:
    mov eax,0x1234          ; must be declared for linker (Id)
    mov ebx,0x5678          ; entry point
    mov edi,eax             ; movs valores inmediatos
    mov esi,ebx
    mov edi,mem1            ; movs entre registros
    mov esi,mem2            ; movs direcciones
                            ; check x/x &mem1, content x/s &mem1

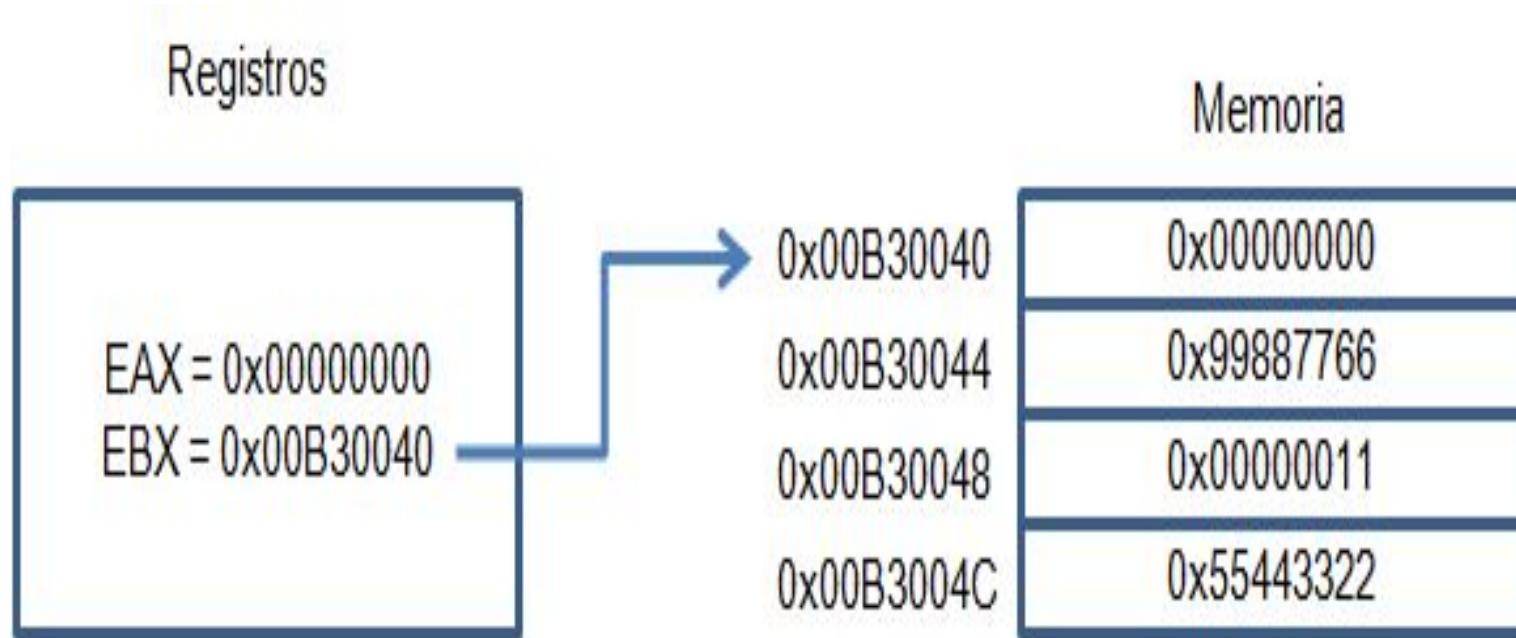
    mov [name],dword 'Done' ; x/s 0x804...
    mov eax,1                ; system call number (sys_exit)
    int 0x80                 ;call kernel

section .data
name db 'Test', 0xa
mem1: db "hello",0xa
mem2: db "world",0xa
```

Acceso a datos

- Localidades de memoria
 - **Direccionamiento directo:** busca y accede a los datos en la dirección de memoria especificada, el registro es el destino. (MOV EAX, [0x08040200]).
 - **Direccionamiento indirecto:** también llamado “direccionamiento por referencia” calcula la dirección del destino, llamada “dirección efectiva” (en el registro reside la dirección del destino). (MOV EAX, [EBX+8])
 - **Implícitos.** El operando está definido dentro de la misma instrucción, dichos operandos pueden ser registros o la pila. (PUSH 0x23, CLC).

Direccionamientos



Registros

- Se tienen 8 registros de propósito general, cada registro tiene un fin específico, dependiendo del tipo de instrucción que se haya ejecutado.
- Estos son: **EAX, EBX, ECX, EDX, EBP, ESP, ESI y EDI**, donde la letra “E” inicial proviene de “extended”.

Registros

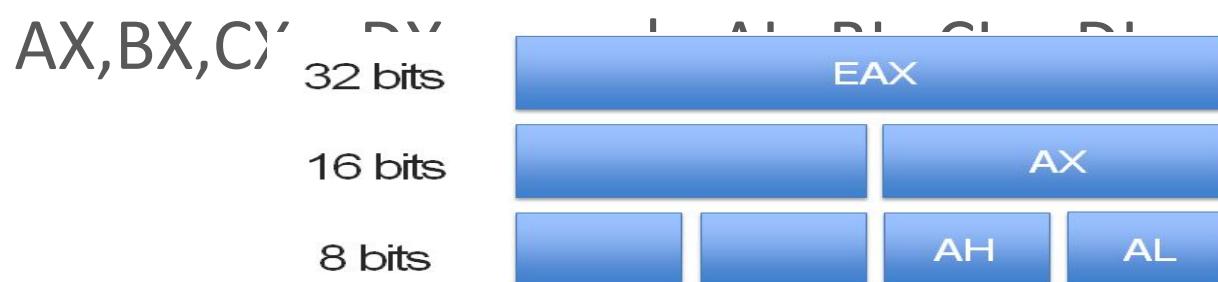
- EAX: Registro acumulador, utilizado en operaciones aritméticas (suma, resta, multiplicación y división), almacenar valores de retorno de funciones y para lectura/escritura en periféricos de I/O.
- EBX: Registro base o apuntador, utilizado para indicar el desplazamiento de direcciones en segmentos de memoria y almacenar datos.

Registros

- ECX: Registro contador, utilizado en operaciones iterativas (ciclos/loops).
- EDX: Registro de datos, utilizado para realizar operaciones aritméticas junto con EAX.

Registros

- Se puede acceder a los primeros 16 bits de EAX, EBX, ECX y EDX, omitiendo la ‘E’, especificandolos de la siguiente forma: AX, BX, CX y DX.
- Se puede acceder a los 8 bits más significativos de AX, BX, CX y DX, usando AH, BH, CH y DH.
- Se puede acceder a los 8 bits menos significativos de AX, BX, CX y DX.



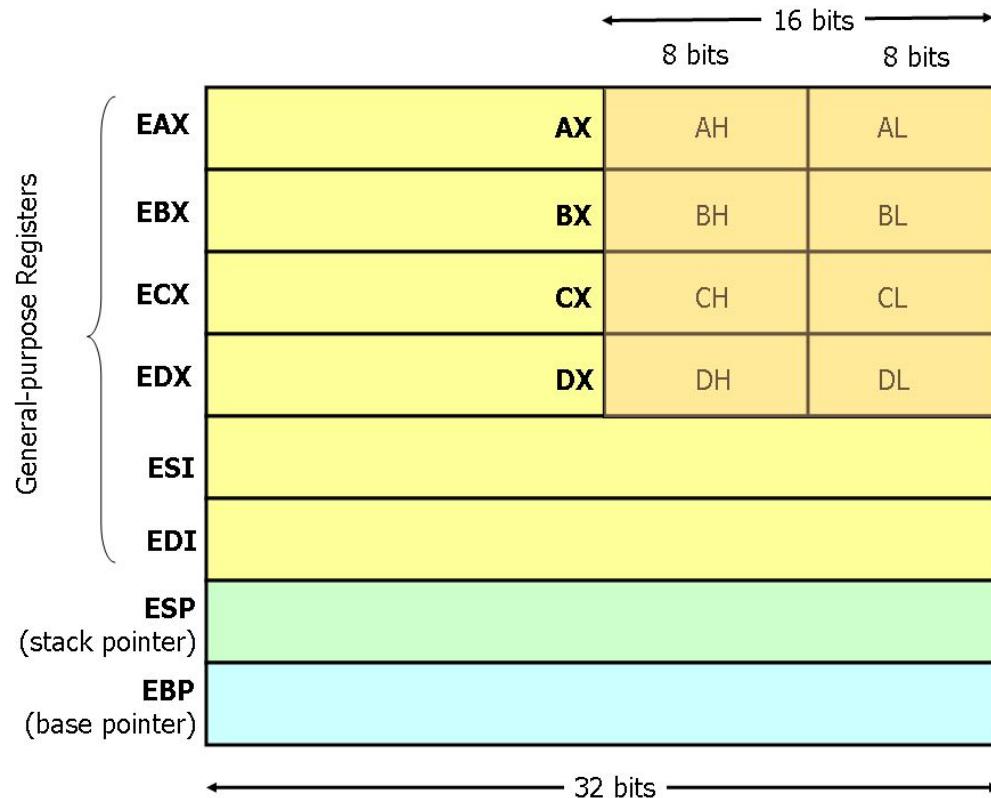
Registros

- ESP (*Stack Pointer*): Apuntador de pila, contiene la dirección de memoria del último valor almacenado en la pila que está usando el un programa en ese momento.
- EBP (*Base Pointer*): Apuntador base, indica la dirección inicial de la pila y es usado para referenciar argumentos y variables locales.

Registros

- ESI (*Source Index*): Índice origen, indica dónde se encuentra el búfer de datos de entrada.
- EDI (*Destination Index*): Índice destino, contiene la dirección donde se copiará el búfer de datos indicado por ESI.

Registros



<http://www.cs.virginia.edu/~evans/cs216/guides/x86.htm>

Registros x64

- Los registros de propósito general en procesadores de **arquitectura x64** (64 bits) son 16: RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14 Y R15.

Registros

- Registro apuntador a instrucciones:

EIP (*Instruction Pointer*): Registro que contiene la dirección de memoria de la siguiente instrucción a ejecutar.

También conocido como contador.

Registros

- Registro de banderas o de estado (EFLAGS).

Las banderas indican el estado actual de la computadora dependiendo del procesamiento de instrucciones, ya que las comparaciones o cálculos aritméticos cambian su estado. El registro está formado por 32 bits, donde cada bandera es un bit.

Registros

- Algunas banderas importantes son:
 - *Sign* (SF - 7): Se establece en 1 si el resultado tiene signo negativo, si es positivo tomará el valor 0.
 - *Zero* (ZF - 6): Se establece en 1 si el resultado de una instrucción fue 0.
 - *Trap* (TF - 8): Ejecución paso a paso.

Registros

- *Overflow* (OF - 11): Se establece si el resultado de una instrucción aritmética con signo genera un número el cual es demasiado largo para el destino.
- *Interruption* (IF - 9): Indica si una interrupción de los dispositivos externos se procesa o no, está reservada para el sistema operativo en modo protegido.
- *Carry* (CF - 0): Se establece si el resultado de una operación aritmética acarrea (adición) o toma prestado (resta) un bit más allá del MSB

Tipos de instrucciones

- En lenguaje ensamblador, existe un número considerable de instrucciones las cuales pueden clasificarse en las siguientes categorías.
 - Manipulación de datos
 - Transferencia de datos
 - Instrucciones condicionales e instrucciones para ramificación

Tipos de instrucciones

- Manipulación de datos
 - Instrucciones aritméticas: ADD, SUB, MUL, IMUL, DIV, IDIV, INC, DEC.
 - Operaciones booleanas: NOT, AND, OR, XOR.
 - Manipulación de bits: SHR, SHL, ROR, ROL.
- Transferencia de datos
 - Incluye instrucciones como MOV, XCHG, PUSH y POP.

Tipos de instrucciones

- Instrucciones condicionales e instrucciones para ramificación
 - Saltos (JMP, JZ, JNZ, JE, JNE, LOOP, etc.)*
 - Llamadas y servicios (CALL, RET, IRET e INT)*
 - Comparaciones (NEG, CMP, TEST, etc.)
 - **Nota:** se modifica el valor de EIP

- **Aritmética**

El formato de la instrucción de la adición es “**ADD destino, valor**” y el de la sustracción es “**SUB destino, valor**”.

lenguaje ensamblador

- Aritmética

La sustracción modifica dos banderas importantes:

- **ZF** (*zero flag*); se establece en 1 si el resultado de la operación es 0.
- **CF** (*carry flag*); se establece si se efectúa un préstamo en la sustracción y replica el valor saliente en corrimientos y rotaciones.

lenguaje ensamblador

- Aritmética

- ADD EAX, EBX ; Asigna el valor de EBX a EAX y almacena el resultado en EAX
- SUB EAX, 0x10 ; Resta 10 en hexadecimal a EAX

Las instrucciones **INC** y **DEC** incrementan o decrementan un registro en uno.

- INC EDX ; Incrementa EDX en 1
- DEC ECX ; Decrementa ECX en 1

lenguaje ensamblador

```
global _start
section .text
_start:
    mov eax,0x1          ; eax=1
    add eax,0x2      ; eax+=2
    ; bl+= (byte en la dir de memoria a la que apunta mem1
    add bl, byte [mem1]
    mov dl,0x5      ; lowest 8 bytes edx = 5
    ; resta a lo que haya en la dirección de memoria mem2 dl
    sub byte [mem2], dl
section .data
mem1: db 0x2
```

lenguaje ensamblador

- Aritmética

El formato de la multiplicación sin signo es “**MUL valor**” y **siempre multiplica al registro EAX**, por lo que este último debe estar configurado apropiadamente antes de que ocurra la operación.

El producto es almacenado como resultado de 64 bits a través los registros EDX y EAX. EDX almacena los 32 bits más significativos de la operación y EAX almacena los 32 bits menos significativos.

Lenguaje ensamblador

- Aritmética

Multiplicador (valor)	Multiplicando	Producto		
8 bits	AL	+	AH:AL	-
16 bits	AX	+	DX:AX	-
32 bits	EAX	+	EDX:EAX	-

lenguaje ensamblador

- Aritmética

- MOV EAX,0x44332211 ; Asigna el valor 0x44332211 a EAX
 - MUL0x50 ; Multiplica EAX (0x44332211) por 0x50 y almacena el resultado (**154FFAA550**) en EDX:EAX

lenguaje ensamblador

- Aritmética

El formato de la división sin signo es “**DIV valor**” y hace lo mismo que la instrucción **MUL** pero de manera opuesta, divide EDX y EAX entre un valor, por lo que estos últimos deben estar configurados apropiadamente antes de que ocurra la operación.

lenguaje ensamblador

- Aritmética

El cociente se almacena en EAX (AX o AL) y el residuo se almacena en EDX (DX o AH).

- MOV EDX,0x0
- MOV EAX,0x150
- DIV 0x75 ; Divide los registros EDX:EAX (0x150) entre 0x75
y
almacena el resultado en el registro EAX y el residuo en EDX

lenguaje ensamblador

- Aritmética

Divisor (valor)	Dividendo	Cociente	Residuo
8 bits	AX	AL	AH
16 bits	DX:AX	AX	DX
32 bits	EDX:EAX	EAX	EDX

lenguaje ensamblador

- Boolean

Instrucción NOT: Lleva a cabo la negación bit a bit, es decir, invierte todos los bits y el resultado se guarda en el mismo operando.

- MOV EAX, 0xF049 ; EAX = 0xF049
- NOT EAX ; EAX = 0xFB6

Lenguaje ensamblador

- Boolean

A	B	A · B
0	0	0
0	1	0
1	0	0
1	1	1

Instrucción AND: Realiza la conjunción bit a bit y el resultado se guarda en el operando destino (AND destino, origen).

- MOV EAX, 0xF049 ; EAX = 0xF049 EBX = 0x05CA
- MOV EBX, 0x05CA; EAX = 0x0048 y EBX = 0x05CA
- AND EAX, EBX ;

Lenguaje ensamblador

- Boolean

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

Instrucción OR: Realiza la disyunción bit a bit y el resultado se guarda en el operando destino (OR destino, origen).

- MOV EAX, 0xF049 ; EAX = 0xF049
- MOV EBX, 0x05CA ; EBX = 0x05CA
- OR EAX, EBX ; EAX = 0xF5CB y EBX = 0x05CA

lenguaje ensamblador

- Boolean

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

Instrucción XOR: Realiza la disyunción exclusiva bit a bit y el resultado se guarda en el operando destino (XOR destino, origen).

En algunos lenguajes de programación se usa el acento circunflejo para efectuar la operación XOR.

```
global _start
section .text
_start:
    mov eax,0x11111111      ; and
    mov edx,0x10101010
    and eax,edx
    mov eax,0x11010101      ; or
    mov edx,0x10101010
    or eax,edx
    mov eax,0x11010011      ; xor
    mov edx,0x11011101
    xor eax,edx
    mov al,0b11001100      ; not
    not al
```

lenguaje ensamblador

- Aritmética

Instrucción NEG: Invierte todos los bits, posteriormente agrega (suma) un 1 y el resultado se guarda en el mismo operando.

- MOV EAX, 0xF049 ; EAX = 0xF049
- NEG EAX ; EAX = 0xFB7 (0xFB6 + 0x0001)

lenguaje ensamblador

- Comparaciones

Instrucción CMP: Realiza la resta implícita (SUB) del origen al destino (CMP destino, origen), el resultado no se guarda en el destino pero cambia el estado de las banderas.

- MOV EAX, 0xF049 ; EAX = 0xF049
- MOV EBX, 0x05CA; EBX = 0x05CA
- CMP EAX, EBX ; EAX = 0xF049 y EBX = 0x05CA

lenguaje ensamblador

- Comparación

Instrucción TEST: Realiza la conjunción implícita (AND) bit a bit entre el origen y el destino (TEST destino, origen), el resultado no se guarda en el destino pero cambia el estado de las banderas.

- MOV EAX, 0xF049 ; EAX = 0xF049
- MOV EBX, 0x05CA; EBX = 0x05CA
- TEST EAX, EBX ; EAX = 0xF049 y EBX = 0x05CA

lenguaje ensamblador

- Manipulación de bits

Las instrucciones **SHR** y **SHL** son usadas para corrimiento de bits en los registros.

El formato para corrimiento a la derecha es “**SHR destino, contador**” y el formato para el corrimiento a la izquierda es “**SHL destino, contador**”.

lenguaje ensamblador

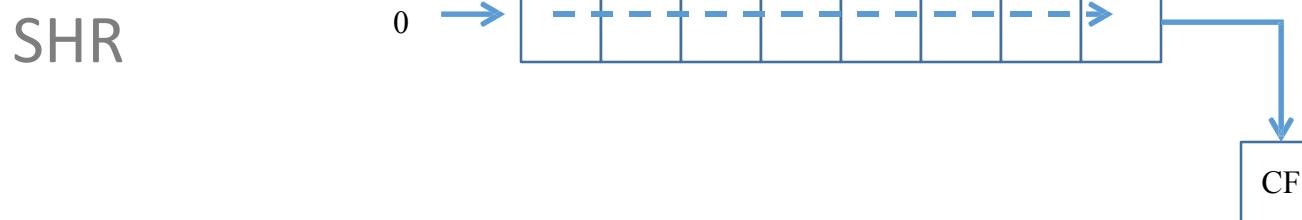
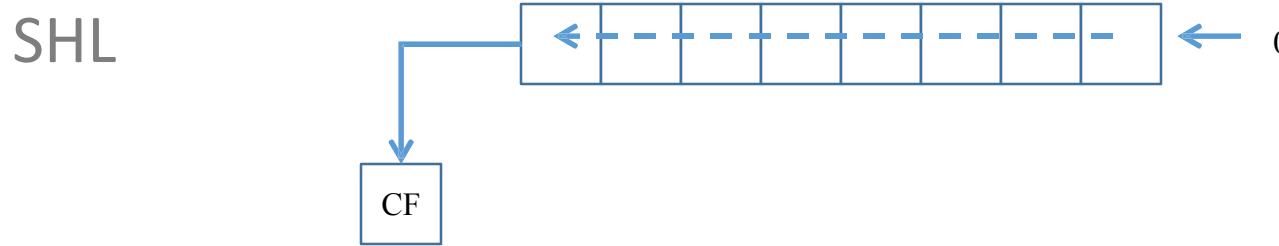
- Corrimientos

Durante dichos corrimientos, el bit procesado se mueve a la bandera de acarreo (CF).

- MOV BL, 0x0A ; Mueve el valor 0x0A (0000 1010 en binario) al registro BL
- SHL BL, 2 ; Corrimiento del registro BL de 2 bits a la izquierda, el resultado es 0010 1000 (40d o 28h)

lenguaje ensamblador

- Aritmética



lenguaje ensamblador

- Manipulación de bits

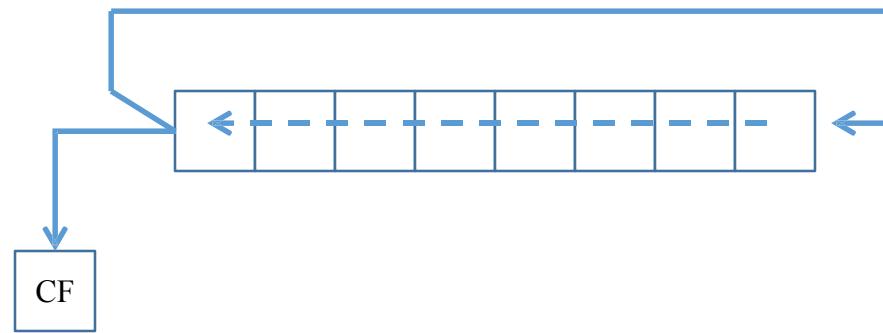
Las instrucciones de rotación **ROR** y **ROL** hacen que los bits **desplazados** regresen al otro extremo. En otras palabras, durante la rotación a la derecha (ROR) los bits menos significativos se rotan a la posición más significativa.

- ❑ MOV BL, 0x0A ; Mueve el valor 0x0A al registro BL .: BL = 0000 10**10**
- ❑ ROR BL, 2 ; Rotación del registro BL de 2 bits a la derecha .: BL = **1000** 0010

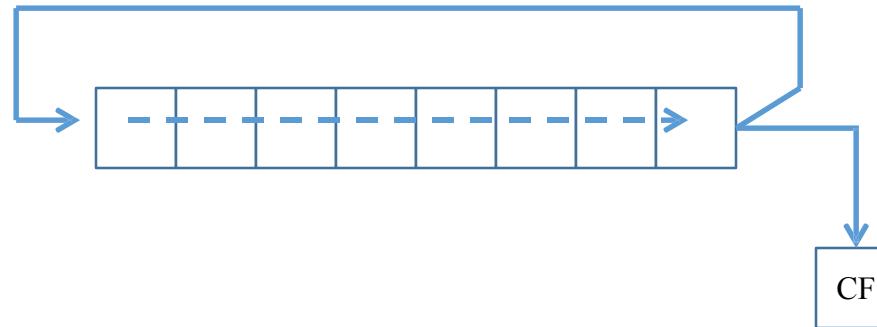
lenguaje ensamblador

- Rotación

ROL



ROR



; shift arithmetic left:

section .text

;agrega un 0 en el bit menos significativo y el bit más significativo se desplaza a CF (si es 1) + ->01111111<- -

;CF = 0 11111110<- nuevo cero ->

la bandera al hacer shl no se activa y el número se duplica

_start:

mov al,0b01111111

shl al,1

; aqui si se activa CF

mov al,0b11111111

shl al,1

; shift right

mov al,0b01111111

shr al,1

lenguaje ensamblador

- Transferencia de datos

Instrucción MOV: Asigna el origen al destino

(MOV destino, origen). Direccionamiento

inmediato.

MOV EAX, 0xF049 ; EAX = 0xF049

MOV EBX, 0x05CA ; EBX = 0x05CA

lenguaje ensamblador

- Transferencia de datos

Instrucción LEA (*Load Effective Address*): Carga la dirección efectiva del operando origen dentro del operando destino (LEA destino, origen).

- LEA EAX, [EBP-0x04] ; dirección de la primera variable local

lenguaje ensamblador

```
global _start
section .text
_start:
; | 1234 | <- celda 0x804a000 <-mem1    # i r $eax
lea eax, [mem1]
; | 1234 | <- celda 0x804a000 <-mem1<-eax
lea ebx, [eax]
; | 1234 | <- celda 0x804a000 <-mem1<-eax<-ebx
mov eax, 1
; | 1234 | <- celda 0x804a000 <-mem1<-ebx
; eax = 1
int 0x80
section .data
mem1: dw 0x1234
```

lenguaje ensamblador

- Transferencia de datos

La instrucción **LEA** no siempre es utilizada para referenciar direcciones de memoria, en ocasiones se usa para calcular valores debido a que requiere pocas instrucciones.

Por ejemplo, al factorizar el número 7 en la instrucción “**LEA EBX, [EAX*7+7]**” se obtiene la expresión “**EBX = (EAX+1) * 7**”.

lenguaje ensamblador

- Transferencia de datos

Instrucción XCHG: Intercambia el contenido del operando destino y con el de origen (XCHG destino, origen). Direccionamiento inmediato.

- MOV EAX, 0xF049 ; EAX = 0xF049
- MOV EBX, 0x05CA; EBX = 0x05CA
- XCHG EAX, EBX ; EAX = 0x05CA y EBX = 0xF049

lenguaje ensamblador

- Transferencia de datos

Instrucción **PUSH**: Almacena el operando en la pila
(PUSH operando). Direccionamiento inmediato.

- MOV EAX, 0xF049 ; EAX = 0xF049
- PUSH EAX ; Ultimo valor pila = 0xF049

lenguaje ensamblador

- Transferencia de datos

Instrucción POP: Extrae y almacena el último valor almacenado en la pila en el operando (POP operando). Direccionamiento inmediato.

- PUSH FOOD
- PUSH BAAD
- MOV EAX, 0xF00D ; EAX = 0xFFFF
- POP EAX ; EAX = BAAD, Ultimo valor en la pila = 0xF00D

Programación en ensamblador

```
; prog1.asm          ; Linea requerida por ld
global _start         ; etiqueta que indica el
section .text        ; inicio de la sección
                      ; ejecutable
                      ; inicio del programa

_start:
    mov eax,0x04      ; eax = 0x04
    mov ebx,0x06      ; ebx = 0x06
    add eax,ebx       ; eax = eax + ebx
```

```
section .text
global _start
_start:
    mov eax,0x1234          ; must be declared for linker (Id)
    mov ebx,0x5678          ; entry point
    mov edi,eax             ; movs valores inmediatos
    mov esi,ebx
    mov edi,mem1            ; movs entre registros
    mov esi,mem2            ; movs direcciones
                            ; check x/x &mem1, content x/s &mem1

    mov [name],dword 'Done' ; x/s 0x804...
    mov eax,1                ; system call number (sys_exit)
    int 0x80                 ;call kernel

section .data
name db 'Test', 0xa
mem1: db "hello",0xa
mem2: db "world",0xa
```

Programación en ensamblador

Ensamblar el código fuente
nasm -f elf -o prog1.o prog1.asm

¿Ejecutar?
./prog1.o

Programación en ensamblador

¿Permission denied?

Enlazar el código objeto con las bibliotecas necesarias para producir el ejecutable.

```
ld -o prog1 prog1.o
```

¿Ejecutar?

```
./prog1
```

Programación en ensamblador

¿Segmentation fault?

Solución: gdb

Programación en ensamblador

Utilizar GDB para ejecutar el programa en un entorno controlado.

```
gdb prog1
```

Dentro de GDB escribir los siguientes comandos:

```
break _start run
```

```
set disassembly-flavor intel
```

```
layout asm
```

```
layout regs
```

```
+--Register group: general--+
leax      0x0      0
lecx      0x0      0
ledx      0x0      0
lebx      0x0      0
lesp      0xbfffffdf0      0xbfffffdf0
lebp      0x0      0x0
lesi      0x0      0
ledi      0x0      0
leip      0x8048060      0x8048060 <_start>
lefags    0x202    [ IF ]
```

```
B+> 0x8048060 <_start>    mov    eax,0x4
0x8048065 <_start+5>    mov    ebx,0x6
0x804806a <_start+10>   add    eax,ebx
0x804806c                 add    BYTE PTR [esi],ch
0x804806e                 jae    0x80480e9
0x8048070                 ins    DWORD PTR es:[edi],dx
0x8048071                 je     0x80480d4
0x8048073                 bound  eax,QWORD PTR [eax]
0x8048075                 cs
0x8048076                 jae    0x80480ec
0x8048078                 jb     0x80480ee
```

child process 30322 In: _start
(gdb)

Line: ?? PC: 0x8048060

Programación en ensamblador

break _start - establece un breakpoint al comienzo del programa

run - ejecuta programa hasta el breakpoint

set disassembly-flavor intel - muestra código ASM en sintaxis intel

layout asm - despliega código asm

layout regs - despliega registros

B+ 0x8048060 <_start> mov eax,0x4
0x8048065 <_start+5> mov ebx,0x6
0x804806a <_start+10> add eax,ebx

- Ejecutar stepi

Registers group: general			
Ieax	0x4	4	

- Ejecutar stepi

Registers group: general			
Ieax	0x0	0	
Iebx	0x6	6	
Iecx	DFFEEEDD		

- Ejecutar stepi

Registers group: general			
Ieax	0xa	10	
Iecx	0x0	0	

Saltos

- Son instrucciones utilizadas para dirigir el flujo de ejecución de un programa hacia una localidad de memoria (relativa o absoluta), comúnmente se especifica una etiqueta.
- Existen dos tipos de saltos: condicionales y no condicionales.

Saltos no condicionales

- Siempre salta hacia otra porción de código en alguna dirección de memoria (JMP destino), por lo que no se necesita revisar el estado de las banderas.

JMP <destino>

Por ejemplo, las instrucciones JMP, CALL y RET.

Saltos condicionales

- Utilizan el valor de las banderas (resultado de las operaciones con registros) para saltar a una nueva rama y transferirle el control si una condición se cumple.

Saltos condicionales

- La estructura de un salto condicional es típicamente de la forma **JXX**, donde las letras “X” (que pueden ir de 1 a 4 letras) describen las siguientes condiciones:
 - Comparación general (Ej. Bandera Zero = 0, Bandera Signo = 1, etc.)
 - Comparación sin signo (ambos operadores)
 - Comparación con signo
- JXX <destino> ; donde XX es una condición a evaluar

lenguaje ensamblador

- Saltos

```
MOV EAX, variable_a    CMP EAX,  
0x08040200  
  
JNE bloque_2
```

bloque_1:

primer bloque de código
JMP fin

bloque_2:

segundo bloque de código

fin:

lenguaje ensamblador

- Saltos

La estructura de un salto condicional es típicamente de la forma **JXX**, donde las letras “X” (que pueden ir de 1 a 4 letras) describen las siguientes condiciones:

- Comparación general
- Comparación sin signo
- Comparación con signo

lenguaje ensamblador

- Saltos

Comparación general

- JZ: Salta si la bandera cero está activa ($ZF = 1$)
- JE: Salta si es igual ($ZF = 1$)
- JNZ: Salta si la bandera cero está deshabilitada ($ZF = 0$)
- JNE: Salta si no es igual ($ZF = 0$)
- JC: Salta si hubo acarreo ($CF = 1$)
- JNC: Salta si no hubo acarreo ($CF = 0$)

lenguaje ensamblador

- Saltos

Comparación general

- JS: Salta si el número tiene signo (SF = 1)
- JNS: Salta si el número no tiene signo (SF = 0)
- JO: Salta si fue un desbordamiento (OF = 1)
- JNO: Salta si no fue un desbordamiento (OF = 0)

lenguaje ensamblador

- **Saltos**

Comparación general

- JP: Salta si hubo paridad, número de bits habilitados par (PF=1)
- JNP: Salta si no hubo paridad, número de bits habilitados impar (PF=0)
- JCXZ: Salta si el registro CX tiene el valor 0.
- JECXZ: Salta si el registro ECX tiene el valor 0.

lenguaje ensamblador

- Saltos

Comparación sin signo (ambos operadores)

- ❑ JA: (above) Salta si es mayor, **op1 > op2** (CF = 0 y ZF = 0)
- ❑ JNBE: Salta si no es menor o igual, **op1 > op2** (CF=0 y ZF=0)
- ❑ JAE: Salta si es mayor o igual, **op1 >= op2** (CF = 0)
- ❑ JNB: Salta si no es menor, **op1 >= op2** (CF = 0)

lenguaje ensamblador

- Saltos

Comparación sin signo (ambos operadores)

- ❑ JB: (below) Salta si es menor, **op1 < op2** (CF = 1)
- ❑ JNAE: Salta si no es mayor o igual, **op1 < op2** (CF = 1)
- ❑ JBE: Salta si es menor o igual, **op1 <= op2** (CF = 1 y ZF = 1)
- ❑ JNA: Salta si no es mayor, **op1 <= op2** (CF = 1 y ZF = 1)

lenguaje ensamblador

- Saltos

Comparación con signo

- JG: (greater) Salta si es mayor, **op1 > op2** (SF = OF y ZF = 0)
- JNLE: Salta si no es menor o igual, **op1 > op2** (SF=OF y ZF=0)
- JGE: Salta si es mayor o igual, **op1 >= op2** (SF = OF)
- JNL: Salta si no es menor, **op1 >= op2** (SF = OF)

lenguaje ensamblador

- Saltos

Comparación con signo

- JL: (less) Salta si es menor, **op1 < op2** (SF != OF)
- JNGE: Salta si no es mayor o igual, **op1 < op2** (SF != OF)
- JLE: Salta si es menor o igual, **op1 <= op2** (SF != OF o ZF = 1)
- JNG: Salta si no es mayor, **op1 <= op2** (SF != OF o ZF = 1)

```
section .text
global _start
_start:
    mov ecx,1 ; valor 1
    mov edx,2 ; valor 2
    cmp ecx,edx ; if ecx == edx:
    je iguales ; iguales ZF=1
    jmp distintos ; else distintos
```

salir:

```
    mov ebx,1
    mov eax,1
    int 0x80
```

distintos:

```
    mov edx,len_dis ; len a escribir
    mov ecx,msg_dis ; buffer
    mov ebx,1 ; salida estandar
    mov eax,4 ; llamada a imprimir
    int 0x80
    jmp salir
```

iguales:

```
; cantidad a escribir
    mov edx,len_equ
    mov ecx,msg_equ ; buffer
    mov ebx,1 ; salida estandar
    mov eax,4 ; num de llamada a
imprimir
    int 0x80
    jmp salir
```

section .data:

```
msg_dis: db 'son distintos',0xa
len_dis: equ $ - msg_dis
msg_equ: db 'son iguales',0xa
len_equ: equ $ - msg_equ
```

Procedimientos

- Un procedimiento es un conjunto de instrucciones delimitadas por una etiqueta indicando el comienzo de la misma y terminando con la instrucción RET.

Inicio_procedimiento:
 <Código>

RET

Procedimientos

- El procedimiento es llamado desde otra función usando la instrucción CALL, teniendo como argumento la etiqueta al inicio del procedimiento.

CALL <Procedimiento>

```
global _start
section .text
; Se declaran antes del punto de
entrada
; se utiliza call para llamarlas
sum:  
    mov eax,ecx
    add eax,edx
    ret  
  
salida:  
    mov eax,1
    int 0x80
```

```
_start:  
    mov ecx,2
    mov edx,2
    call suma
```

Procedimientos

- Nota: un procedimiento con frecuencia puede utilizar un prólogo y un epílogo para gestionar el uso de la pila. No obstante, no es forzosa la presencia de ambos.
- Lenguajes de programación como C, los incluyen como parte de cada función.

Interrupciones y excepciones

- El procesador proporciona dos mecanismos para modificar la ejecución de un programa: interrupciones y excepciones.
 - Una interrupción es evento asíncrono que generalmente se presenta por la acción de un dispositivo de entrada/salida.
 - Una excepción es un evento síncrono que se genera cuando el procesador detecta una o más condiciones predefinidas durante la ejecución de una instrucción.

Interrupciones y excepciones

- Por ejemplo:
 - El mnemónico de interrupción **INT 80h** se encarga de atender las llamadas al sistema (syscall) en Linux.
 - INT 21h atiende las syscall en sistemas DOS.

Interrupciones y excepciones

<https://syscalls.kernelgrok.com/>

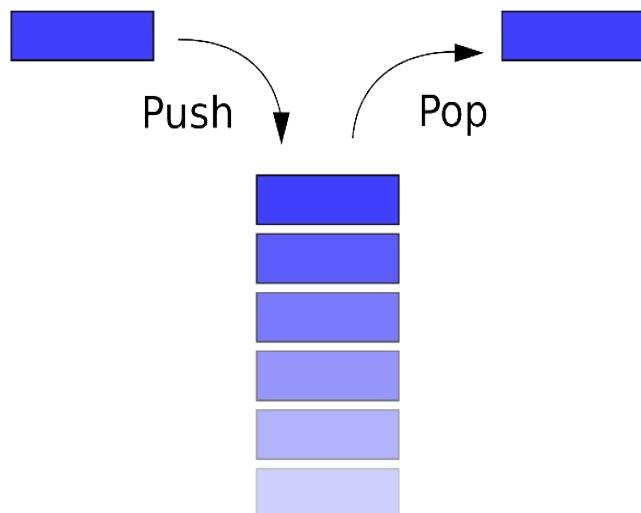
Registers									Definition
#	Name	eax	ebx	ecx	edx	esi	edi		Search:
0	sys_restart_syscall	0x00	-	-	-	-	-	-	kernel/signal.c:2058
1	sys_exit	0x01	int error_code	-	-	-	-	-	kernel/exit.c:1046
2	sys_fork	0x02	struct pt_regs *	-	-	-	-	-	arch/alpha/kernel/entry.S:716
3	sys_read	0x03	unsigned int fd	char __user *buf	size_t count	-	-	-	fs/read_write.c:391
4	sys_write	0x04	unsigned int fd	const char __user *buf	size_t count	-	-	-	fs/read_write.c:408
5	sys_open	0x05	const char __user int flags	*filename	int mode	-	-	-	fs/open.c:900
6	sys_close	0x06	unsigned int fd	-	-	-	-	-	fs/open.c:969
7	sys_waitpid	0x07	pid_t pid	int __user *stat_addr	int options	-	-	-	kernel/exit.c:1771
8	sys_CREAT	0x08	const char __user int mode	*pathname	-	-	-	-	fs/open.c:933
9	sys_link	0x09	const char __user const char __user *oldname	*newname	-	-	-	-	fs/namei.c:2520
10	sys_unlink	0x0a	const char __user * pathname	-	-	-	-	-	fs/namei.c:2352
11	sys_execve	0x0b	char __user * * __user char __user * __user *	struct pt_regs	-	-	-	-	arch/alpha/kernel/entry.S:925
12	sys_chdir	0x0c	const char __user *filename	-	-	-	-	-	fs/open.c:361
13	sys_time	0x0d	time_t __user *tloc	-	-	-	-	-	kernel posix-timers.c:855
14	sys_mknod	0x0e	const char __user int mode	*filename	unsigned dev	-	-	-	fs/namei.c:2067
15	sys_chmod	0x0f	const char __user mode_t mode	*filename	-	-	-	-	fs/open.c:507
16	sys_lchown16	0x10	const char __user old_uid_t user	*filename	old_gid_t group	-	-	-	kernel/uid16.c:27
17	not implemented	0x11	-	-	-	-	-	-	
18	sys_stat	0x12	char __user *filename	struct _old_kernel_stat	__user *statbuf	-	-	-	fs/stat.c:150
19	sys_lseek	0x13	unsigned int fd	off_t offset	unsigned int origin	-	-	-	fs/read_write.c:167
20	sys_getpid	0x14	-	-	-	-	-	-	kernel/timer.c:1337
21	sys_mount	0x15	char __user *dev_name	char __user *dir_name	char __user *type	unsigned long flags	void __user *data	-	fs/namespace.c:2118
22	sys_oldumount	0x16	char __user *name	-	-	-	-	-	fs/namespace.c:1171
23	sys_setuid16	0x17	old_uid_t uid	-	-	-	-	-	kernel/uid16.c:67
24	sys_getuid16	0x18	-	-	-	-	-	-	kernel/uid16.c:212

Showing 1 to 25 of 338 entries

First Previous 1 2 3 4 5 Next Last

Pila

- La pila (Stack) es una estructura de datos tipo LIFO (último en entrar, primero en salir) usada para colocar y remover elementos.



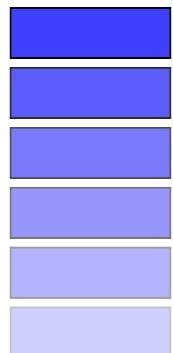
Pila

- Frecuentemente se utiliza para almacenamiento temporal de variables locales, argumentos y direcciones de retorno.
- Su principal uso es la gestión de datos intercambiados entre llamadas a funciones.
- Cada vez que se realiza una llamada se genera un nuevo *stack frame* en la pila.

Pila

- La pila crece hacia las direcciones de memoria más bajas, es decir, cada vez que se insertan valores en la pila se utilizan direcciones de memoria más pequeñas.

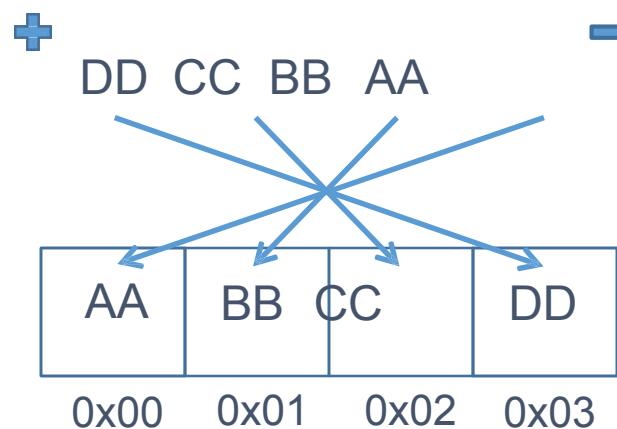
0x00000000



0xFFFFFFFF

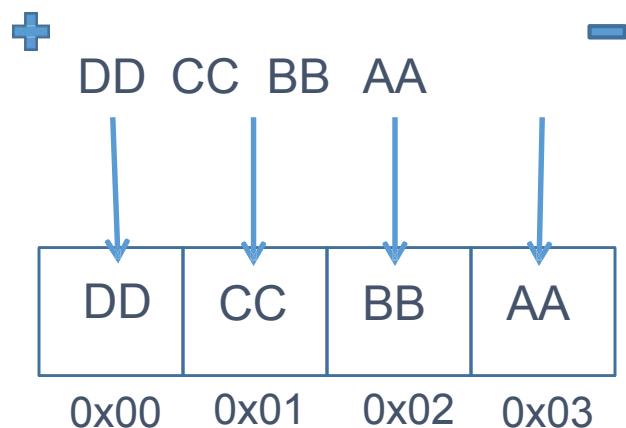
Lenguaje Ensamblador

- *Orden de bytes*
- Little-endian
- Significa que el byte menos significativo se almacena en la dirección más baja de memoria. Usado por Intel.



Lenguaje Ensamblador

- Big-endian
- Representa los bytes en “orden natural”. El byte más significativo se almacena en la dirección de memoria más baja.



Sintaxis AT&T

- La notación es: mnemónico origen, destino
- Los registros se denotan por el signo %
- Las constantes numéricas son precedidas del signo \$
- Los números hexadecimales inician con 0x
- Los números binarios inician con 0b

Sintaxis AT&T

- Cuando se efectúa operaciones con direcciones de memoria, el tamaño para 8, 16, 32 y 64 bits se especifica con los sufijos: b, w, l y q, a diferencia de Intel donde se usa: BYTE PTR, WORD PTR, DWORD PTR y QWORD PTR.

Ejemplo:

```
movl $0x0,0x8(%rsp)
```

```
mov DWORD PTR [rsp+0x8],0x0
```

Directivas

- Son instrucciones propias del ensamblador y no están relacionadas con los mnemónicos definidos para un procesador.
- Generalmente son usadas para almacenar valores en el ejecutable o para señalar secciones .
- Ejemplos de directivas son DB, DW, EQU, SECTION, GLOBAL, etc.

Más programación ...

Generar un programa que compare si el contenido del registro EAX es igual a 0xFACEB00C, si lo es que muestre el mensaje “EAX tiene 0xFACEB00C :)”, en caso contrario “EAX no tiene 0xFACEB00C :V”.

Más programación ...

```
; prog2.asm    global _start
section .text
_start:
    cmp
    eax,0xFACEB00C jne
    bloque1

bloque1:
    mov edx,len_msg1
    mov ecx,msg1
    mov ebx,1
    mov eax,4
    int 0x80 jmp fin

    bloque2:
        mov edx,len_msg2
        mov ecx,msg2
        mov ebx,1
        mov eax,4 int
        0x80

fin:

section .data
msg1 db 'EAX tiene 0xFACEB00C
:)',0xa
len_msg1 equ $ - msg1
msg2 db 'EAX no tiene 0xFACEB00C
:V',0xa
len_msg2 equ $ - msg2
```

Tare

a

- Significado de la “R” en los registros de propósito general de 64 bits(*)
- Generar el código ensamblador de las sentencias switch, y for(comentadas linea por linea, en papel)
- Investigar tipos de datos en ensamblador y su tamaño

Prácticas

- Generar un programa que cifre una cadena con al menos 5 instrucciones de ensamblador distintas, al menos una de las instrucciones debe ser SHR o SHL con un corrimiento mínimo de 3.
- Generar un programa que descifre la cadena previamente cifrada.

<http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf>