Lenguaje Ensamblador

David Alejandro González Márquez

Clase disponible en: https://github.com/fokerman/computingSystemsCourse

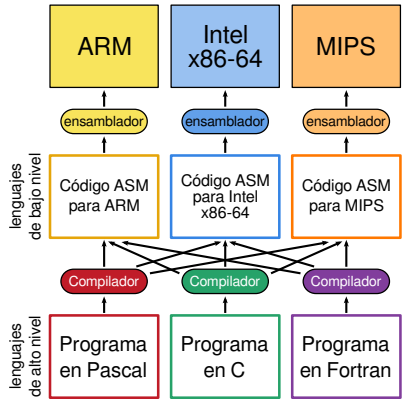
Lenguaje Ensamblador

El lenguaje ensamblador consiste en escribir las instrucciones del procesador como **texto** (mnemónicos).

Estas instrucciones son convertidas a su codificación binaria por medio del **ensamblador**.

Los lenguajes de tipo ensamblador, son **específicos** para cada arquitectura.

Los mnemónicos que pueden ser escritos dependen de las instrucciones que soporte el procesador.



En lo que sigue, vamos a estudiar el lenguaje ensamblador de la arquitectura x86-64 o AMD64.

Arquitectura x86-64 o AMD64

- Es una **extensión** de la arquitectura x86 de Intel.
- Binario **compatible** con toda la línea de procesadores.
- Desarrollado por AMD bajo el nombre **AMD64**.
- Vio la luz en el 2003 bajo la línea de procesadores `Opteron`.
- Actualmente es el estándar en notebooks, computadoras de escritorio y servidores.
- Compite con arquitecturas como ARM y sus extensiones, enfocadas principalmente a dispositivos embebidos*.

Arquitectura x86-64 o AMD64

Características:

- Arquitectura de 64 bits.
- Direcciones de memoria de 64 bits (8 bytes), direccionable a byte.
- Registros de 64 bits (pueden operar también en 32, 16 u 8 bits).
- Opera con datos en *little endian*.
- Múltiples modos de direccionamiento.

Arquitectura x86-64 o AMD64

Características:

- Arquitectura de 64 bits.
- Direcciones de memoria de 64 bits (8 bytes), direccionable a byte.
- Registros de 64 bits (pueden operar también en 32, 16 u 8 bits).
- Opera con datos en *little endian*.
- Múltiples modos de direccionamiento.

Manuales:

- Intel ® 64 and IA-32 Architectures - Software Developer's Manual
  - Volume 1: Basic Architecture.
  - Volume 2: Instruction Set Reference A-Z.
  - Volume 3: System Programming.
  - Volume 4: Model-Specific Registers.

  Total 4778 páginas.

  `https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html`

Arquitectura x86-64 o AMD64

Registros

Conjuntos de Registros:

1. Registros de propósito general
2. Registros de la unidad de Punto Flotante
3. Registros de las extensiones MMX
4. Registros de las extensiones SSE

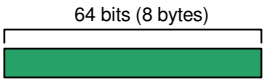
Arquitectura x86-64 o AMD64

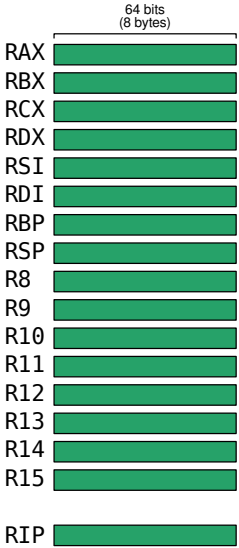
Registros

Conjuntos de Registros:

1. **Registros de propósito general**
2. ~~Registros de la unidad de Punto Flotante~~
3. ~~Registros de las extensiones MMX~~
4. ~~Registros de las extensiones SSE~~

Vamos a hacer foco solo sobre los registros de **propósito general**.





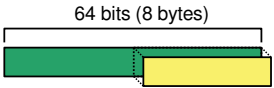
Arquitectura x86-64 o AMD64

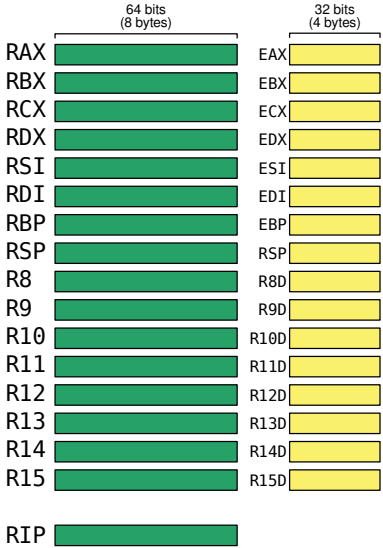
Registros

Conjuntos de Registros:

1. **Registros de propósito general**
2. ~~Registros de la unidad de Punto Flotante~~
3. ~~Registros de las extensiones MMX~~
4. ~~Registros de las extensiones SSE~~

Vamos a hacer foco solo sobre los registros de **propósito general**.





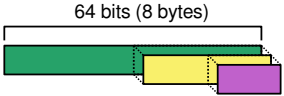
Arquitectura x86-64 o AMD64

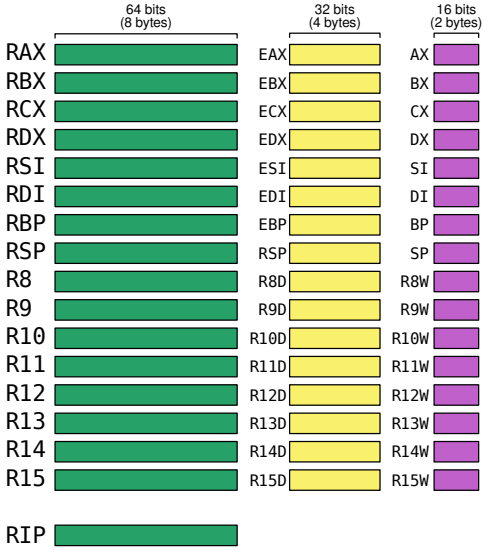
Registros

Conjuntos de Registros:

1. **Registros de propósito general**
2. ~~Registros de la unidad de Punto Flotante~~
3. ~~Registros de las extensiones MMX~~
4. ~~Registros de las extensiones SSE~~

Vamos a hacer foco solo sobre los registros de **propósito general**.





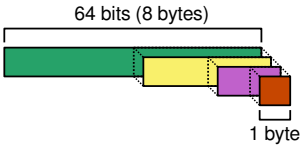
Arquitectura x86-64 o AMD64

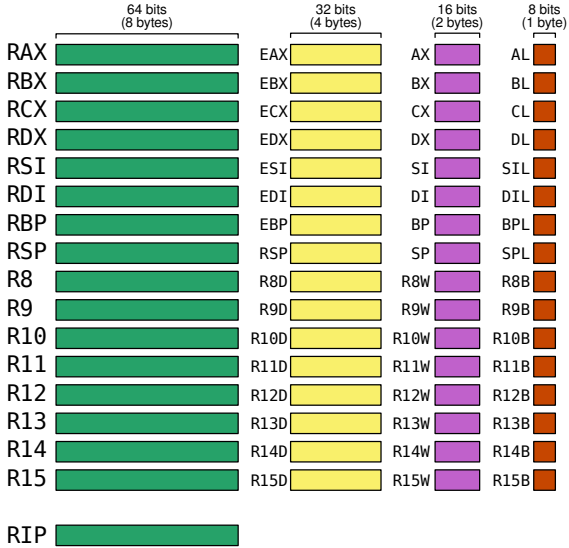
Registros

Conjuntos de Registros:

1. **Registros de propósito general**
2. ~~Registros de la unidad de Punto Flotante~~
3. ~~Registros de las extensiones MMX~~
4. ~~Registros de las extensiones SSE~~

Vamos a hacer foco solo sobre los registros de **propósito general**.





Arquitectura x86-64 o AMD64

Parámetros

Las instrucciones en general pueden tomar entre tres y cero parámetros.
Sus posibilidades son las siguientes:

Arquitectura x86-64 o AMD64

Parámetros

Las instrucciones en general pueden tomar entre tres y cero parámetros.
Sus posibilidades son las siguientes:

Implícito

El operando es parte de la instrucción.

Ej. CLC, clear carry flag.

Arquitectura x86-64 o AMD64

Parámetros

Las instrucciones en general pueden tomar entre tres y cero parámetros.
Sus posibilidades son las siguientes:

Implícito
El operando es parte de la instrucción.

Ej. `CLC`, clear carry flag.

Inmediato
Un valor constante inmediato como parte de la codificación de la instrucción.

Ej. `MOV RAX, 150`

Arquitectura x86-64 o AMD64

Parámetros

Las instrucciones en general pueden tomar entre tres y cero parámetros.
Sus posibilidades son las siguientes:

Implícito
El operando es parte de la instrucción.

Ej. `CLC`, clear carry flag.

Inmediato
Un valor constante inmediato como parte de la codificación de la instrucción.

Ej. `MOV RAX, 150`

Registro
Un registro de propósito general u otros registros según la instrucción.

Ej. `ADD AX, BX`

Arquitectura x86-64 o AMD64

Parámetros

Las instrucciones en general pueden tomar entre tres y cero parámetros.
Sus posibilidades son las siguientes:

Implícito

El operando es parte de la instrucción.

Ej. CLC, clear carry flag.

Inmediato

Un valor constante inmediato como parte de la codificación de la instrucción.

Ej. `MOV RAX, 150`

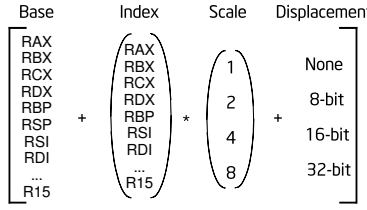
Registro

Un registro de propósito general u otros registros según la instrucción.

Ej. `ADD AX, BX`

Memoria

Una dirección de memoria respetando el formato:



Offset = Base + (Index * Scale) + Displacement

Ej. `ADD AL, [0x123123]`
Ej. `ADD AX, [RDX]`
Ej. `ADD EAX, [RBX+RCX]`
Ej. `ADD RAX, [RBX+RCX*2+5]`

Arquitectura x86-64 o AMD64

Conjunto de Instrucciones (algunas)

Instrucción	Descripción
`ADD DST, SRC`	`DST = DST + SRC`
`ADC DST, SRC`	`DST = DST + SRC + carry`
`SUB DST, SRC`	`DST = DST - SRC`
`SBB DST, SRC`	`DST = DST - SRC - borrow`
`INC DST`	`DST = DST + 1`
`DEC DST`	`DST = DST - 1`
`NEG DST`	`DST = -DST`
`CMP DST, SRC`	Altera los flags resultado de `DST-SRC`
`AND DST, SRC`	`DST = bitwiseAnd(DST, SRC)`
`OR DST, SRC`	`DST = bitwiseOr(DST, SRC)`
`XOR DST, SRC`	`DST = bitwiseXor(DST, SRC)`
`NOT DST`	`DST = bitwiseNot(DST)`

Arquitectura x86-64 o AMD64

Conjunto de Instrucciones (algunas)

Instrucción	Descripción
`SAR DST, imm`	Shift aritmético a derecha
`SHR DST, imm`	Shift lógico a derecha
`SAL DST, imm`	Shift aritmético a izquierda
`SHL DST, imm`	Shift lógico a izquierda
`ROR DST, imm`	Rotación a derecha
`ROL DST, imm`	Rotación a izquierda
`RCR DST, imm`	Rotación con carry a derecha
`RCL DST, imm`	Rotación con carry a izquierda

Instrucción	Descripción
`XCHG DST, SRC`	Intercambia dos datos, ya sea memoria o registros
`MOV DST, SRC`	Copia datos entre registros, memoria o setea valores inmediatos

Arquitectura x86-64 o AMD64

Conjunto de Instrucciones (algunas)

Instrucción	Descripción
`JMP DST`	Salto incondicional
`JE DST`	Salto si es igual
`JA DST`	Salto si es $>$ (sin signo)
`JAE DST`	Salto si es $\geqslant$ (sin signo)
`JB DST`	Salto si es $<$ (sin signo)
`JBE DST`	Salto si es $\leqslant$ (sin signo)
`JG DST`	Salto si es $>$ (con signo)
`JGE DST`	Salto si es $\geqslant$ (con signo)
`JL DST`	Salto si es $<$ (con signo)
`JLE DST`	Salto si es $\leqslant$ (con signo)
`JC DST`	Salto si hay carry
`JO DST`	Salto si hay overflow
`JS DST`	Salto si es negativo
`JZ DST`	Salto si es cero

Hola Mundo ...

Ejemplo

Escribir un programa en lenguaje ensamblador que imprima por pantalla:

```
Hola Mundo
```

Hola Mundo ...

Ejemplo

Escribir un programa en lenguaje ensamblador que imprima por pantalla:

```
Hola Mundo
```

¿Cómo?

Secciones, etiquetas y símbolos

Un programa se separa en secciones:

- **section .data**: donde se declaran las variables globales inicializadas. (`DB`, `DW`, `DD` y `DQ`).
- **section .text**: donde se escribe el código.

Secciones, etiquetas y símbolos

Un programa se separa en secciones:

- **section .data**: donde se declaran las variables globales inicializadas. (`DB`, `DW`, `DD` y `DQ`).
- **section .text**: donde se escribe el código.

Etiquetas y símbolos:

- **_start**: Símbolo utilizando como punto de entrada de un programa.
- **global**: Modificador que define un símbolo que va a ser visto externamente.

Secciones, etiquetas y símbolos

Un programa se separa en secciones:

- **section .data**: donde se declaran las variables globales inicializadas. (DB, DW, DD y DQ).
- **section .text**: donde se escribe el código.

Etiquetas y símbolos:

- **_start**: Símbolo utilizando como punto de entrada de un programa.
- **global**: Modificador que define un símbolo que va a ser visto externamente.

Comandos e instrucciones para el ensamblador:

- **DB** define un byte, **DW** define una word (2 bytes), **DD** define una doble word (4 bytes), **DQ** define una quad word (8 bytes).
- Expresión **$**: toma el valor de la posición en memoria de la línea que contiene la expresión.
- Comando **EQU**: para definir constantes como ecuaciones que evalúa el ensamblador.

Llamadas al sistema operativo (syscalls)

Si vamos a querer imprimir en pantalla dentro de un sistema, tenemos que **solicitar permiso** al sistema operativo para hacerlo.

Llamadas al sistema operativo (syscalls)

Si vamos a querer imprimir en pantalla dentro de un sistema, tenemos que **solicitar permiso** al sistema operativo para hacerlo.

En sistemas POSIX, como Linux, su interfaz es:

1- El número de función que queremos en RAX.

2- Los parámetros en RBX, RCX, RDX, RSI, RDI y RBP; en ese orden.

3- Llamamos a la interrupción del sistema operativo (int 0x80).

4- En general, la respuesta está en RAX.

Llamadas al sistema operativo (syscalls)

Si vamos a querer imprimir en pantalla dentro de un sistema, tenemos que **solicitar permiso** al sistema operativo para hacerlo.

En sistemas POSIX, como Linux, su interfaz es:

1- El número de función que queremos en RAX.

2- Los parámetros en RBX, RCX, RDX, RSI, RDI y RBP; en ese orden.

3- Llamamos a la interrupción del sistema operativo (int 0x80).

4- En general, la respuesta está en RAX.

- Mostrar por pantalla (**sys_write**):
  Función **4**
  Parámetro 1: **¿donde?** (1 = stdout)
  Parámetro 2: **Dirección de memoria del mensaje**
  Parámetro 3: **Longitud del mensaje** (en bytes)

Llamadas al sistema operativo (syscalls)

Si vamos a querer imprimir en pantalla dentro de un sistema, tenemos que **solicitar permiso** al sistema operativo para hacerlo.

En sistemas POSIX, como Linux, su interfaz es:

1- El número de función que queremos en RAX.

2- Los parámetros en RBX, RCX, RDX, RSI, RDI y RBP; en ese orden.

3- Llamamos a la interrupción del sistema operativo (int 0x80).

4- En general, la respuesta está en RAX.

- Mostrar por pantalla (**sys_write**):
  Función **4**
  Parámetro 1: **¿donde?** (1 = stdout)
  Parámetro 2: **Dirección de memoria del mensaje**
  Parámetro 3: **Longitud del mensaje** (en bytes)

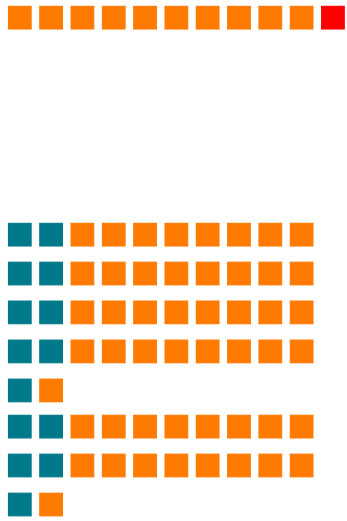
- Terminar programa (**exit**):
  Función **1**
  Parámetro 1: **código de retorno** (0 = sin error)

Hola Mundo...

```
section .data
  msg: DB 'Hola Mundo', 10
  largo EQU $ - msg

  global _start
section .text
  _start:
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, msg    ; mensaje
    mov rdx, largo  ; longitud
    int 0x80
    mov rax, 1      ; funcion 1
    mov rbx, 0      ; codigo
    int 0x80
```

Hola Mundo...

```
section .data
  msg: DB 'Hola Mundo', 10
  largo EQU $ - msg

  global _start
section .text
  _start:
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, msg    ; mensaje
    mov rdx, largo ; longitud
    int 0x80
    mov rax, 1      ; funcion 1
    mov rbx, 0      ; codigo
    int 0x80
```

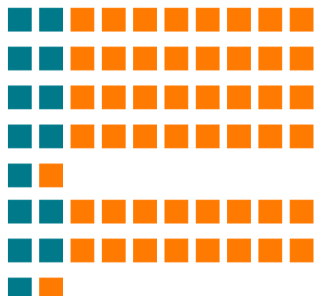


Hola Mundo...

```
section .data
  msg: DB 'Hola Mundo', 10
  largo EQU $ - msg

  global _start
section .text
  _start:
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, msg    ; mensaje
    mov rdx, largo  ; longitud
    int 0x80
    mov rax, 1      ; funcion 1
    mov rbx, 0      ; codigo
    int 0x80
```

¿Cómo funciona **EQU**?

```
largo = 11 - 0
largo = 11
```



Hola Mundo...

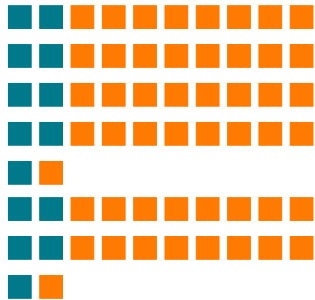
```
section .data
  msgPepe: DB 'Hola Pepe', 10
  msg: DB 'Hola Mundo', 10
  largo EQU $ - msg

  global _start
section .text
  _start:
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, msg    ; mensaje
    mov rdx, largo ; longitud
    int 0x80
    mov rax, 1      ; funcion 1
    mov rbx, 0      ; codigo
    int 0x80
```

Hola Mundo...

```
section .data
  msgPepe: DB 'Hola Pepe', 10
  msg: DB 'Hola Mundo', 10
  largo EQU $ - msg

  global _start
section .text
  _start:
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, msg    ; mensaje
    mov rdx, largo  ; longitud
    int 0x80
    mov rax, 1      ; funcion 1
    mov rbx, 0      ; codigo
    int 0x80
```

¿Cómo funciona **EQU**?

```
largo = 21 - 10
largo = 11
```



Ensamblando y linkeando

Ensamblamos:

```
nasm -f elf64 -g -F DWARF holamundo.asm
```

Linkeamos:

```
ld -o holamundo holamundo.o
```

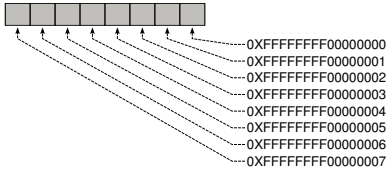
Ejecutamos:

```
./holamundo
```

Pila (Stack)

La Pila o Stack es una **estructura** en memoria administrada por el procesador.

Se utiliza para guardar el **contexto de ejecución** de una función, variables locales y datos.



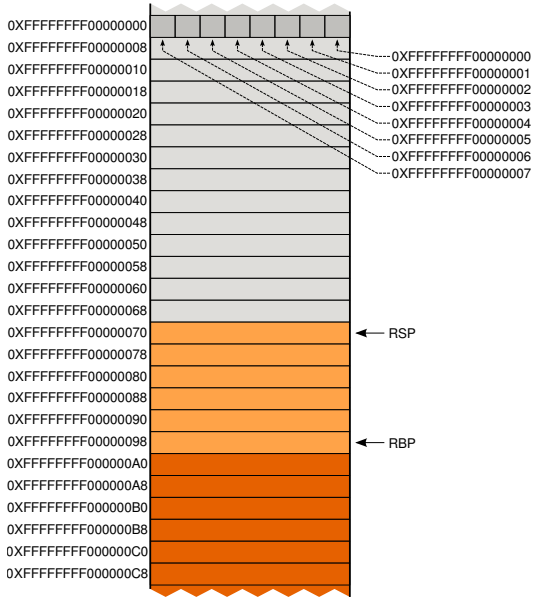
Pila (Stack)

La Pila o Stack es una **estructura** en memoria administrada por el procesador.

Se utiliza para guardar el **contexto de ejecución** de una función, variables locales y datos.

En x86-64, se identifica por dos registros:

- RBP - **Base Pointer**
  Apunta a la base
- RSP - **Stack Pointer**
  Al tope (último elemento válido)



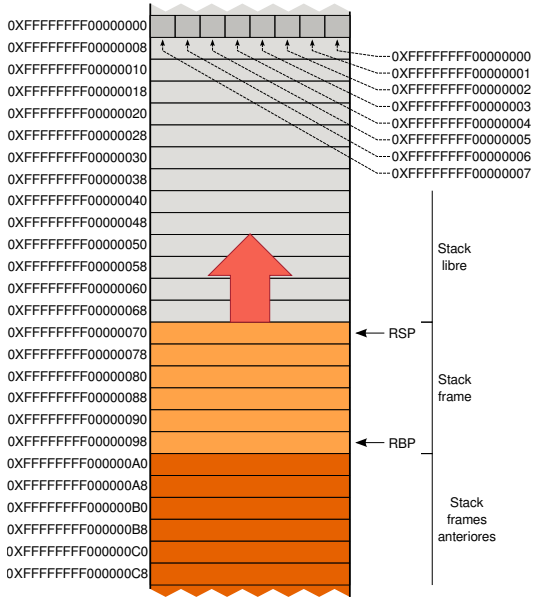
Pila (Stack)

La Pila o Stack es una **estructura** en memoria administrada por el procesador.

Se utiliza para guardar el **contexto de ejecución** de una función, variables locales y datos.

En x86-64, se identifica por dos registros:

- RBP - **Base Pointer**
  Apunta a la base
- RSP - **Stack Pointer**
  Al tope (último elemento válido)



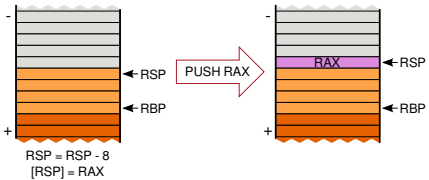
Pila (Stack)

Existen varias instrucciones que utilizan la pila, vamos a hacer foco en 4 de ellas.

Pila (Stack)

Existen varias instrucciones que utilizan la pila, vamos a hacer foco en 4 de ellas.

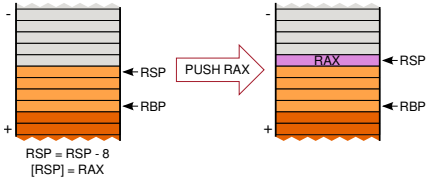
PUSH - Guarda un dato en la pila.



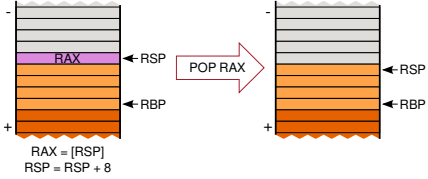
Pila (Stack)

Existen varias instrucciones que utilizan la pila, vamos a hacer foco en 4 de ellas.

PUSH - Guarda un dato en la pila.



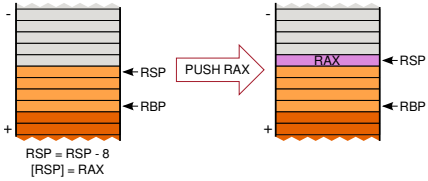
POP - Quita un dato de la pila.



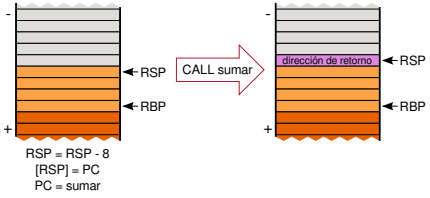
Pila (Stack)

Existen varias instrucciones que utilizan la pila, vamos a hacer foco en 4 de ellas.

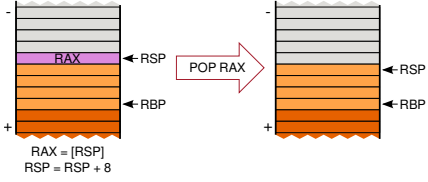
PUSH - Guarda un dato en la pila.



CALL - Salta a una subrutina y guarda la dirección de retorno en la pila.



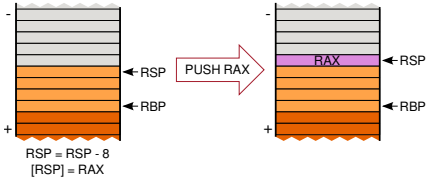
POP - Quita un dato de la pila.



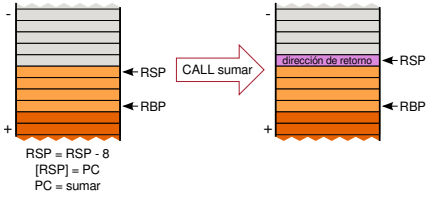
Pila (Stack)

Existen varias instrucciones que utilizan la pila, vamos a hacer foco en 4 de ellas.

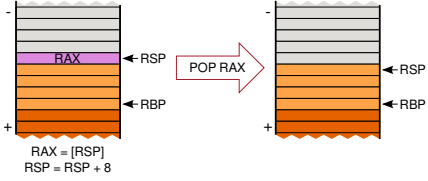
PUSH - Guarda un dato en la pila.



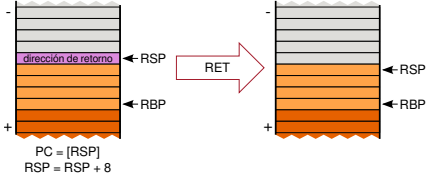
CALL - Salta a una subrutina y guarda la dirección de retorno en la pila.



POP - Quita un dato de la pila.



RET - Toma de la pila la dirección de retorno y regresa a la rutina.



Ejemplo de uso de la pila

Imprimir dos nombres pasando por la pila

```
section .data
  msg1: DB 'James P. Sullivan', 10
  largo1 EQU $ - msg1
  msg2: DB 'Mike Wazowski', 10
  largo2 EQU $ - msg2

  global _start
section .text
  _start:

    push largo1     ; push largo1

    push msg1       ; push msg1

    push largo2     ; push largo2

    push msg2       ; push msg2
```

```
mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg2
pop rdx         ; pop largo2
int 0x80
```

```
mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg1
pop rdx         ; pop largo1
int 0x80

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

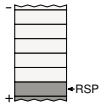
Terminal

```
$ ./swapPushPop.out
```

Ejemplo de uso de la pila

Imprimir dos nombres pasando por la pila

```
section .data
  msg1: DB 'James P. Sullivan', 10
  largo1 EQU $ - msg1
  msg2: DB 'Mike Wazowski', 10
  largo2 EQU $ - msg2

  global _start
section .text
  _start:

    push largo1     ; push largo1

    push msg1       ; push msg1

    push largo2     ; push largo2

    push msg2       ; push msg2
```




```
mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg2
pop rdx         ; pop largo2
int 0x80
```

```
mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg1
pop rdx         ; pop largo1
int 0x80

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

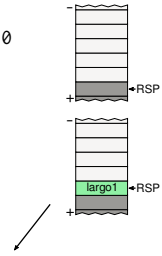
Terminal

```
$ ./swapPushPop.out
```

Ejemplo de uso de la pila

Imprimir dos nombres pasando por la pila

```
section .data
  msg1: DB 'James P. Sullivan', 10
  largo1 EQU $ - msg1
  msg2: DB 'Mike Wazowski', 10
  largo2 EQU $ - msg2

  global _start
section .text
  _start:

    push largo1     ; push largo1


    push msg1       ; push msg1


    push largo2     ; push largo2


    push msg2       ; push msg2
```




```
mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg2
pop rdx         ; pop largo2
int 0x80
```

```
mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg1
pop rdx         ; pop largo1
int 0x80

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

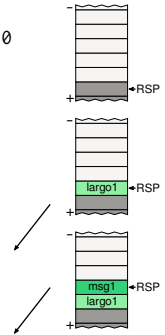
Terminal

```
$ ./swapPushPop.out
```

Ejemplo de uso de la pila

Imprimir dos nombres pasando por la pila

```
section .data
  msg1: DB 'James P. Sullivan', 10
  largo1 EQU $ - msg1
  msg2: DB 'Mike Wazowski', 10
  largo2 EQU $ - msg2

  global _start
section .text
  _start:

    push largo1     ; push largo1


    push msg1       ; push msg1


    push largo2     ; push largo2


    push msg2       ; push msg2
```




```
mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg2
pop rdx         ; pop largo2
int 0x80
```

```
mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg1
pop rdx         ; pop largo1
int 0x80

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

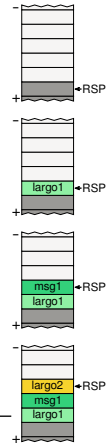
Terminal

```
$ ./swapPushPop.out
```

Ejemplo de uso de la pila

Imprimir dos nombres pasando por la pila

```
section .data
  msg1: DB 'James P. Sullivan', 10
  largo1 EQU $ - msg1
  msg2: DB 'Mike Wazowski', 10
  largo2 EQU $ - msg2

  global _start
section .text
  _start:

    push largo1     ; push largo1


    push msg1       ; push msg1


    push largo2     ; push largo2


    push msg2       ; push msg2
```




```
mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg2
pop rdx         ; pop largo2
int 0x80
```

```
mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg1
pop rdx         ; pop largo1
int 0x80

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

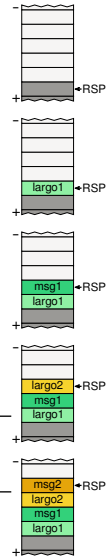
Terminal

```
$ ./swapPushPop.out
```

Ejemplo de uso de la pila

Imprimir dos nombres pasando por la pila

```
section .data
  msg1: DB 'James P. Sullivan', 10
  largo1 EQU $ - msg1
  msg2: DB 'Mike Wazowski', 10
  largo2 EQU $ - msg2

  global _start
section .text
  _start:

    push largo1     ; push largo1


    push msg1       ; push msg1


    push largo2     ; push largo2


    push msg2       ; push msg2
```




```
mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg2
pop rdx         ; pop largo2
int 0x80
```

```
mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg1
pop rdx         ; pop largo1
int 0x80

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

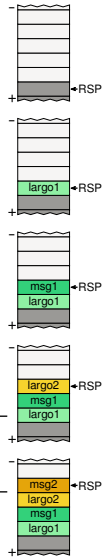
Terminal

```
$ ./swapPushPop.out
```

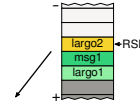
Ejemplo de uso de la pila

Imprimir dos nombres pasando por la pila

```
section .data
  msg1: DB 'James P. Sullivan', 10
  largo1 EQU $ - msg1
  msg2: DB 'Mike Wazowski', 10
  largo2 EQU $ - msg2

  global _start
section .text
  _start:

    push largo1     ; push largo1


    push msg1       ; push msg1


    push largo2     ; push largo2


    push msg2       ; push msg2
```




```
mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg2
pop rdx         ; pop largo2
int 0x80
```




```
mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg1
pop rdx         ; pop largo1
int 0x80

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

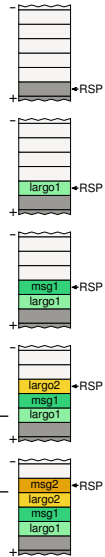
Terminal

```
$ ./swapPushPop.out
```

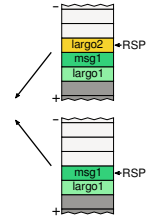
Ejemplo de uso de la pila

Imprimir dos nombres pasando por la pila

```
section .data
  msg1: DB 'James P. Sullivan', 10
  largo1 EQU $ - msg1
  msg2: DB 'Mike Wazowski', 10
  largo2 EQU $ - msg2

  global _start
section .text
  _start:

    push largo1     ; push largo1


    push msg1       ; push msg1


    push largo2     ; push largo2


    push msg2       ; push msg2
```




```
mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg2
pop rdx         ; pop largo2
int 0x80
```




```
mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg1
pop rdx         ; pop largo1
int 0x80

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

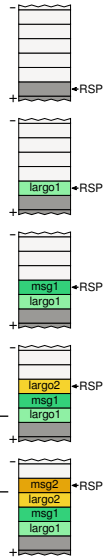
Terminal

```
$ ./swapPushPop.out
```

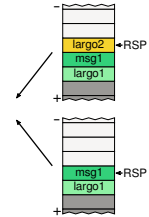
Ejemplo de uso de la pila

Imprimir dos nombres pasando por la pila

```
section .data
  msg1: DB 'James P. Sullivan', 10
  largo1 EQU $ - msg1
  msg2: DB 'Mike Wazowski', 10
  largo2 EQU $ - msg2

  global _start
section .text
  _start:

    push largo1     ; push largo1


    push msg1       ; push msg1


    push largo2     ; push largo2


    push msg2       ; push msg2
```




```
mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg2
pop rdx         ; pop largo2
int 0x80
```




```
mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg1
pop rdx         ; pop largo1
int 0x80

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

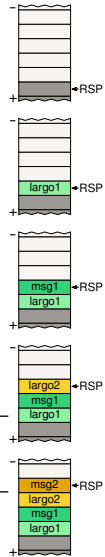
Terminal

```
$ ./swapPushPop.out
Mike Wazowski
```

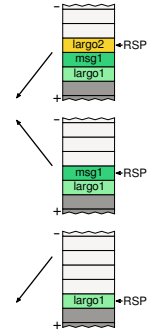
Ejemplo de uso de la pila

Imprimir dos nombres pasando por la pila

```
section .data
  msg1: DB 'James P. Sullivan', 10
  largo1 EQU $ - msg1
  msg2: DB 'Mike Wazowski', 10
  largo2 EQU $ - msg2

  global _start
section .text
  _start:

    push largo1     ; push largo1


    push msg1       ; push msg1


    push largo2     ; push largo2


    push msg2       ; push msg2
```




```
mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg2
pop rdx         ; pop largo2
int 0x80
```




```
mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg1
pop rdx         ; pop largo1
int 0x80

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

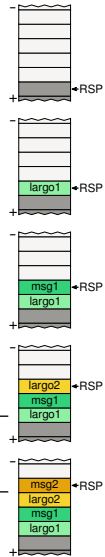
Terminal

```
$ ./swapPushPop.out
Mike Wazowski
```

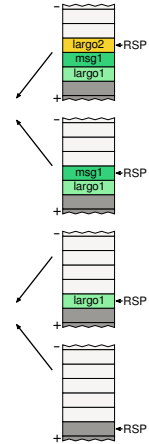
Ejemplo de uso de la pila

Imprimir dos nombres pasando por la pila

```
section .data
  msg1: DB 'James P. Sullivan', 10
  largo1 EQU $ - msg1
  msg2: DB 'Mike Wazowski', 10
  largo2 EQU $ - msg2

  global _start
section .text
  _start:

    push largo1     ; push largo1


    push msg1       ; push msg1


    push largo2     ; push largo2


    push msg2       ; push msg2
```




```
mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg2
pop rdx         ; pop largo2
int 0x80


mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg1
pop rdx         ; pop largo1
int 0x80

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```



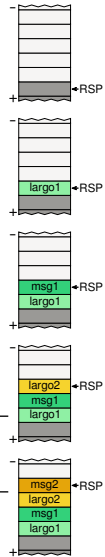
Terminal

```
$ ./swapPushPop.out
Mike Wazowski
```

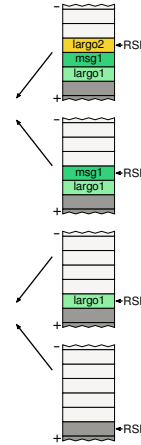
Ejemplo de uso de la pila

Imprimir dos nombres pasando por la pila

```
section .data
  msg1: DB 'James P. Sullivan', 10
  largo1 EQU $ - msg1
  msg2: DB 'Mike Wazowski', 10
  largo2 EQU $ - msg2

  global _start
section .text
  _start:

    push largo1     ; push largo1


    push msg1       ; push msg1


    push largo2     ; push largo2


    push msg2       ; push msg2
```




```
mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg2
pop rdx         ; pop largo2
int 0x80


mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg1
pop rdx         ; pop largo1
int 0x80

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```



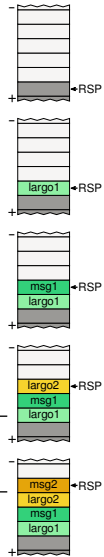
Terminal

```
$ ./swapPushPop.out
Mike Wazowski
James P. Sullivan
```

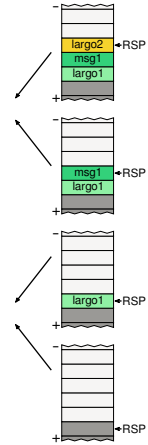
Ejemplo de uso de la pila

Imprimir dos nombres pasando por la pila

```
section .data
  msg1: DB 'James P. Sullivan', 10
  largo1 EQU $ - msg1
  msg2: DB 'Mike Wazowski', 10
  largo2 EQU $ - msg2

  global _start
section .text
  _start:

    push largo1     ; push largo1


    push msg1       ; push msg1


    push largo2     ; push largo2


    push msg2       ; push msg2
```




```
mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg2
pop rdx         ; pop largo2
int 0x80


mov rax, 4      ; funcion 4
mov rbx, 1      ; stdout
pop rcx         ; pop msg1
pop rdx         ; pop largo1
int 0x80

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```



Terminal

```
$ ./swapPushPop.out
Mike Wazowski
James P. Sullivan
$
```

Llamado a subrutinas

Llamado a subrutinas

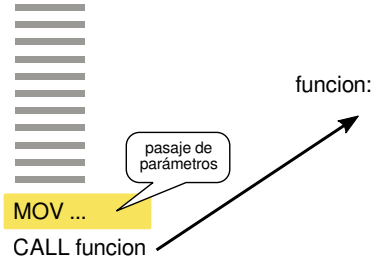
CALL funcion

Llamado a subrutinas

Pasaje de parámetros

Mover a registros los datos que requiere la función.

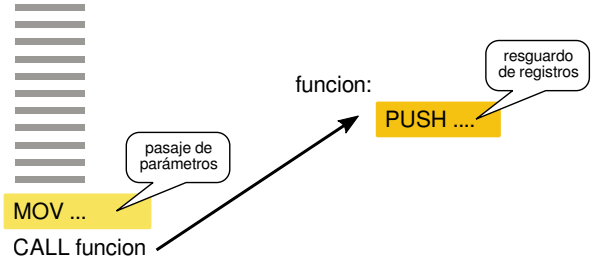
Llamado a subrutinas



Pasaje de parámetros

Mover a registros los datos que requiere la función.

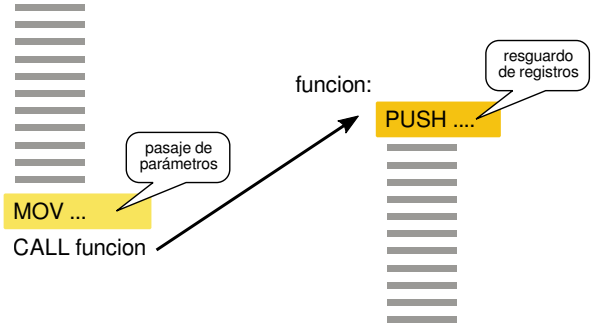
Llamado a subrutinas



Pasaje de parámetros
Mover a registros los datos que requiere la función.

Resguardo de registros
Guardar en la pila los registros que no deben ser modificados.

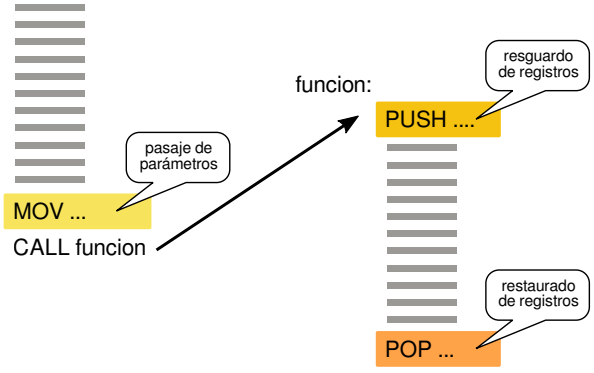
Llamado a subrutinas



Pasaje de parámetros
Mover a registros los datos que requiere la función.

Resguardo de registros
Guardar en la pila los registros que no deben ser modificados.

Llamado a subrutinas

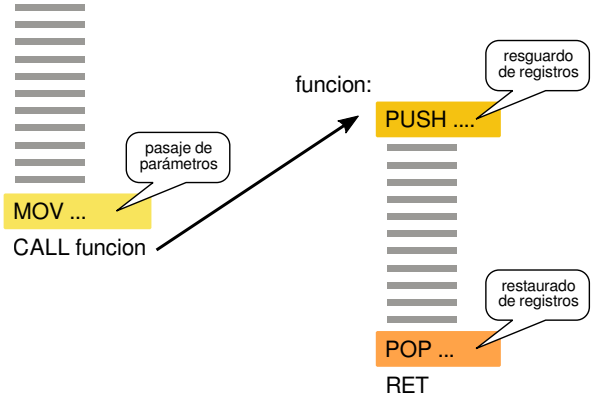


Pasaje de parámetros
Mover a registros los datos que requiere la función.

Resguardo de registros
Guardar en la pila los registros que no deben ser modificados.

Restaurado de registros
Restaurar desde la pila los registros salvados.

Llamado a subrutinas

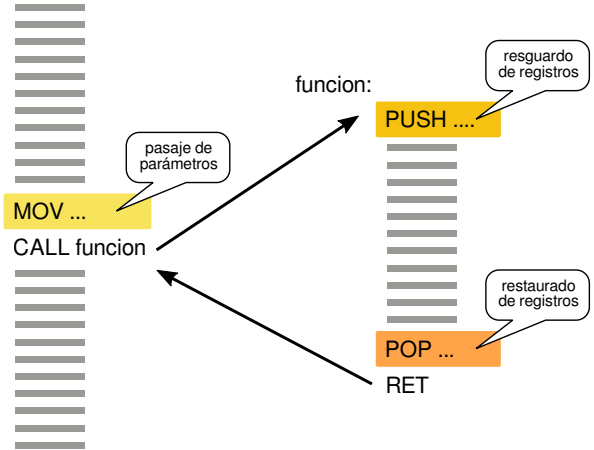


Pasaje de parámetros
Mover a registros los datos que requiere la función.

Resguardo de registros
Guardar en la pila los registros que no deben ser modificados.

Restaurado de registros
Restaurar desde la pila los registros salvados.

Llamado a subrutinas



Pasaje de parámetros
Mover a registros los datos que requiere la función.

Resguardo de registros
Guardar en la pila los registros que no deben ser modificados.

Restaurado de registros
Restaurar desde la pila los registros salvados.

Ejemplo de instrucciones `Call` y `Ret`

Imprimir 10 veces “hola mundo”.

```
section .data
  msg: DB 'Hola Mundo', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, msg    ; mensaje
    mov rdx, largo  ; longitud
    int 0x80
    pop rbx
    ret
```

```
_start:
  mov rbx, 10
  ciclo:
    call printHolaMundo
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```

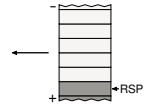
Terminal

```
$ ./printCiclo.out
```

Ejemplo de instrucciones `Call` y `Ret`

Imprimir 10 veces “hola mundo”.

```
section .data
  msg: DB 'Hola Mundo', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, msg    ; mensaje
    mov rdx, largo  ; longitud
    int 0x80
    pop rbx
    ret
```




```
_start:
  mov rbx, 10
  ciclo:
    call printHolaMundo
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```

Terminal

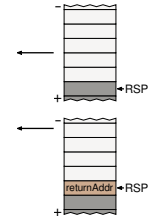
```
$ ./printCiclo.out
```

Ejemplo de instrucciones Call y Ret

Imprimir 10 veces “hola mundo”.

```
section .data
  msg: DB 'Hola Mundo', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, msg    ; mensaje
    mov rdx, largo  ; longitud
    int 0x80
    pop rbx
    ret
```

```
_start:
  mov rbx, 10
  ciclo:
    call printHolaMundo
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```



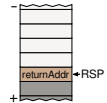
Terminal

```
$ ./printCiclo.out
```

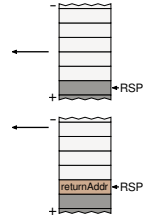
Ejemplo de instrucciones `Call` y `Ret`

Imprimir 10 veces “hola mundo”.

```
section .data
  msg: DB 'Hola Mundo', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, msg    ; mensaje
    mov rdx, largo  ; longitud
    int 0x80
    pop rbx
    ret
```




```
_start:
  mov rbx, 10
  ciclo:
    call printHolaMundo
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```



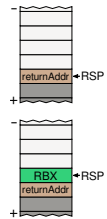
Terminal

```
$ ./printCiclo.out
```

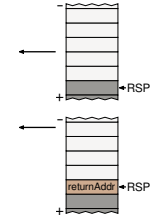
Ejemplo de instrucciones `Call` y `Ret`

Imprimir 10 veces “hola mundo”.

```
section .data
  msg: DB 'Hola Mundo', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, msg    ; mensaje
    mov rdx, largo  ; longitud
    int 0x80
    pop rbx
    ret
```




```
_start:
  mov rbx, 10
  ciclo:
    call printHolaMundo
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```



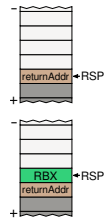
Terminal

```
$ ./printCiclo.out
```

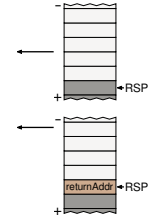
Ejemplo de instrucciones `Call` y `Ret`

Imprimir 10 veces “hola mundo”.

```
section .data
  msg: DB 'Hola Mundo', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, msg    ; mensaje
    mov rdx, largo  ; longitud
    int 0x80
    pop rbx
    ret
```




```
_start:
  mov rbx, 10
  ciclo:
    call printHolaMundo
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```



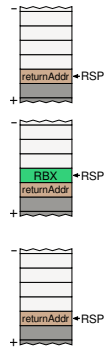
Terminal

```
$ ./printCiclo.out
Hola Mundo
```

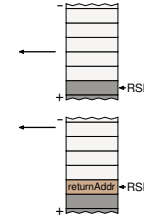
Ejemplo de instrucciones Call y Ret

Imprimir 10 veces “hola mundo”.

```
section .data
  msg: DB 'Hola Mundo', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, msg    ; mensaje
    mov rdx, largo  ; longitud
    int 0x80
    pop rbx
    ret
```




```
_start:
  mov rbx, 10
  ciclo:
    call printHolaMundo
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```



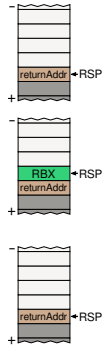
Terminal

```
$ ./printCiclo.out
Hola Mundo
```

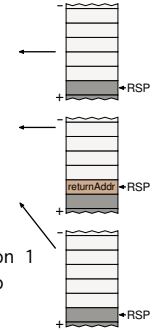
Ejemplo de instrucciones `Call` y `Ret`

Imprimir 10 veces “hola mundo”.

```
section .data
  msg: DB 'Hola Mundo', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, msg    ; mensaje
    mov rdx, largo  ; longitud
    int 0x80
    pop rbx
    ret
```




```
_start:
  mov rbx, 10
  ciclo:
    call printHolaMundo
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```



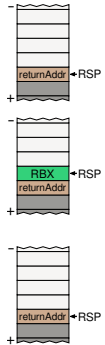
Terminal

```
$ ./printCiclo.out
Hola Mundo
```

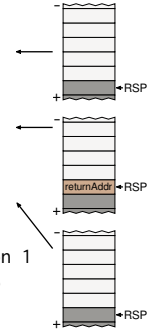
Ejemplo de instrucciones `Call` y `Ret`

Imprimir 10 veces “hola mundo”.

```
section .data
  msg: DB 'Hola Mundo', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, msg    ; mensaje
    mov rdx, largo  ; longitud
    int 0x80
    pop rbx
    ret
```




```
_start:
  mov rbx, 10
  ciclo:
    call printHolaMundo
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```



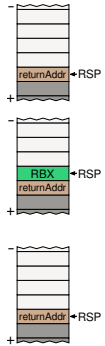
Terminal

```
$ ./printCiclo.out
Hola Mundo
Hola Mundo
```

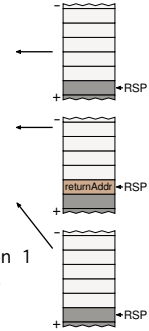
Ejemplo de instrucciones `Call` y `Ret`

Imprimir 10 veces “hola mundo”.

```
section .data
  msg: DB 'Hola Mundo', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, msg    ; mensaje
    mov rdx, largo  ; longitud
    int 0x80
    pop rbx
    ret
```




```
_start:
  mov rbx, 10
  ciclo:
    call printHolaMundo
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```



Terminal

```
$ ./printCiclo.out
Hola Mundo
Hola Mundo
...
Hola Mundo
$
```

Ejemplo de instrucciones Call y Ret

Imprimir 10 veces “hola mundo” con un contador.

```
section .data
 msg: DB 'Hola Mundo 0', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, rdi    ; mensaje
    mov rdx, rsi    ; longitud
    int 0x80
    pop rbx
    ret
```

```
_start:
  mov rbx, 10
  ciclo:
    mov rdi, msg
    mov rsi, largo
    call printHolaMundo
    inc byte [msg+largo-2]
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```

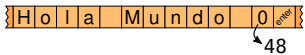
Terminal

```
$ ./printCiclo.out
```

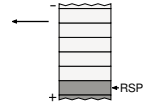
Ejemplo de instrucciones Call y Ret

Imprimir 10 veces “hola mundo” con un contador.

```
section .data
 msg: DB 'Hola Mundo 0', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, rdi    ; mensaje
    mov rdx, rsi    ; longitud
    int 0x80
    pop rbx
    ret
```




```
_start:
  mov rbx, 10
  ciclo:
    mov rdi, msg
    mov rsi, largo
    call printHolaMundo
    inc byte [msg+largo-2]
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```



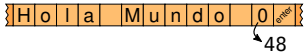
Terminal

```
$ ./printCiclo.out
```

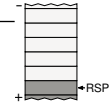
Ejemplo de instrucciones Call y Ret

Imprimir 10 veces “hola mundo” con un contador.

```
section .data
 msg: DB 'Hola Mundo 0', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, rdi    ; mensaje
    mov rdx, rsi    ; longitud
    int 0x80
    pop rbx
    ret
```




```
_start:
  mov rbx, 10
  ciclo:
    mov rdi, msg
    mov rsi, largo
    call printHolaMundo
    inc byte [msg+largo-2]
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```



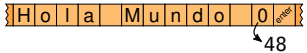
Terminal

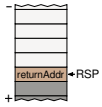
```
$ ./printCiclo.out
```

Ejemplo de instrucciones Call y Ret

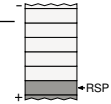
Imprimir 10 veces “hola mundo” con un contador.

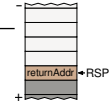
```
section .data
 msg: DB 'Hola Mundo 0', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, rdi    ; mensaje
    mov rdx, rsi    ; longitud
    int 0x80
    pop rbx
    ret
```






```
_start:
  mov rbx, 10
  ciclo:
    mov rdi, msg
    mov rsi, largo
    call printHolaMundo
    inc byte [msg+largo-2]
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```





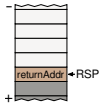
Terminal

```
$ ./printCiclo.out
```

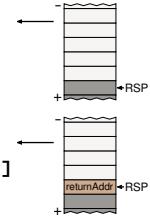
Ejemplo de instrucciones Call y Ret

Imprimir 10 veces “hola mundo” con un contador.

```
section .data
 msg: DB 'Hola Mundo 0', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, rdi    ; mensaje
    mov rdx, rsi    ; longitud
    int 0x80
    pop rbx
    ret
```




```
_start:
  mov rbx, 10
  ciclo:
    mov rdi, msg
    mov rsi, largo
    call printHolaMundo
    inc byte [msg+largo-2]
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```



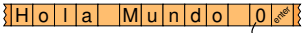
Terminal

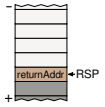
```
$ ./printCiclo.out
```

Ejemplo de instrucciones Call y Ret

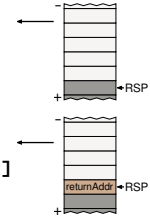
Imprimir 10 veces “hola mundo” con un contador.

```
section .data
 msg: DB 'Hola Mundo 0', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, rdi    ; mensaje
    mov rdx, rsi    ; longitud
    int 0x80
    pop rbx
    ret
```






```
_start:
  mov rbx, 10
  ciclo:
    mov rdi, msg
    mov rsi, largo
    call printHolaMundo
    inc byte [msg+largo-2]
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```



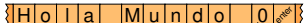
Terminal

```
$ ./printCiclo.out
Hola Mundo 0
```

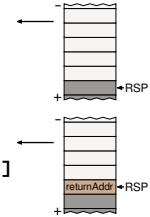
Ejemplo de instrucciones Call y Ret

Imprimir 10 veces “hola mundo” con un contador.

```
section .data
 msg: DB 'Hola Mundo 0', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, rdi    ; mensaje
    mov rdx, rsi    ; longitud
    int 0x80
    pop rbx
    ret
```




```
_start:
  mov rbx, 10
  ciclo:
    mov rdi, msg
    mov rsi, largo
    call printHolaMundo
    inc byte [msg+largo-2]
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```



Terminal

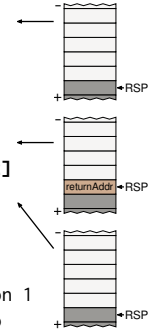
```
$ ./printCiclo.out
Hola Mundo 0
```

Ejemplo de instrucciones Call y Ret

Imprimir 10 veces “hola mundo” con un contador.

```
section .data
 msg: DB 'Hola Mundo 0', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, rdi    ; mensaje
    mov rdx, rsi    ; longitud
    int 0x80
    pop rbx
    ret
```


```
_start:
  mov rbx, 10
  ciclo:
    mov rdi, msg
    mov rsi, largo
    call printHolaMundo
    inc byte [msg+largo-2]
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```



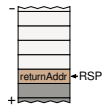
Terminal

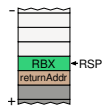
```
$ ./printCiclo.out
Hola Mundo 0
```

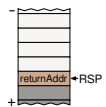
Ejemplo de instrucciones Call y Ret

Imprimir 10 veces “hola mundo” con un contador.

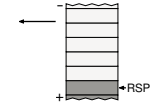
```
section .data
 msg: DB 'Hola Mundo 0', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, rdi    ; mensaje
    mov rdx, rsi    ; longitud
    int 0x80
    pop rbx
    ret
```

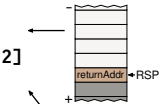


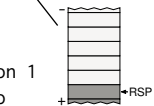





```
_start:
  mov rbx, 10
  ciclo:
    mov rdi, msg
    mov rsi, largo
    call printHolaMundo
    inc byte [msg+largo-2]
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```







Terminal

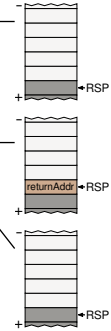
```
$ ./printCiclo.out
Hola Mundo 0
```

Ejemplo de instrucciones Call y Ret

Imprimir 10 veces “hola mundo” con un contador.

```
section .data
 msg: DB 'Hola Mundo 0', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, rdi    ; mensaje
    mov rdx, rsi    ; longitud
    int 0x80
    pop rbx
    ret
```


```
_start:
  mov rbx, 10
  ciclo:
    mov rdi, msg
    mov rsi, largo
    call printHolaMundo
    inc byte [msg+largo-2]
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```



Terminal

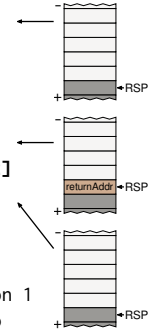
```
$ ./printCiclo.out
Hola Mundo 0
Hola Mundo 1
```

Ejemplo de instrucciones Call y Ret

Imprimir 10 veces “hola mundo” con un contador.

```
section .data
 msg: DB 'Hola Mundo 0', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, rdi    ; mensaje
    mov rdx, rsi    ; longitud
    int 0x80
    pop rbx
    ret
```


```
_start:
  mov rbx, 10
  ciclo:
    mov rdi, msg
    mov rsi, largo
    call printHolaMundo
    inc byte [msg+largo-2]
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```



Terminal

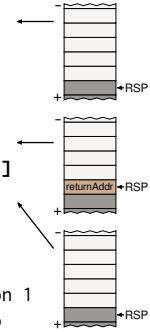
```
$ ./printCiclo.out
Hola Mundo 0
Hola Mundo 1
```

Ejemplo de instrucciones Call y Ret

Imprimir 10 veces “hola mundo” con un contador.

```
section .data
 msg: DB 'Hola Mundo 0', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, rdi    ; mensaje
    mov rdx, rsi    ; longitud
    int 0x80
    pop rbx
    ret
```


```
_start:
  mov rbx, 10
  ciclo:
    mov rdi, msg
    mov rsi, largo
    call printHolaMundo
    inc byte [msg+largo-2]
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```



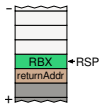
Terminal

```
$ ./printCiclo.out
Hola Mundo 0
Hola Mundo 1
Hola Mundo 2
```

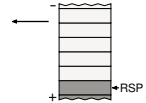
Ejemplo de instrucciones Call y Ret

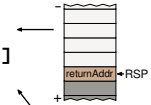
Imprimir 10 veces “hola mundo” con un contador.

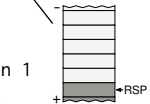
```
section .data
 msg: DB 'Hola Mundo 0', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, rdi    ; mensaje
    mov rdx, rsi    ; longitud
    int 0x80
    pop rbx
    ret
```




```
_start:
  mov rbx, 10
  ciclo:
    mov rdi, msg
    mov rsi, largo
    call printHolaMundo
    inc byte [msg+largo-2]
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```







Terminal

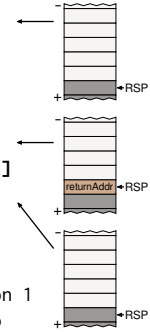
```
$ ./printCiclo.out
Hola Mundo 0
Hola Mundo 1
Hola Mundo 2
```

Ejemplo de instrucciones Call y Ret

Imprimir 10 veces “hola mundo” con un contador.

```
section .data
 msg: DB 'Hola Mundo 0', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, rdi    ; mensaje
    mov rdx, rsi    ; longitud
    int 0x80
    pop rbx
    ret
```


```
_start:
  mov rbx, 10
  ciclo:
    mov rdi, msg
    mov rsi, largo
    call printHolaMundo
    inc byte [msg+largo-2]
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```



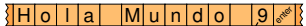
Terminal

```
$ ./printCiclo.out
Hola Mundo 0
Hola Mundo 1
Hola Mundo 2
Hola Mundo 3
```

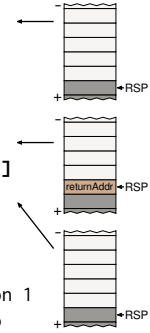
Ejemplo de instrucciones Call y Ret

Imprimir 10 veces “hola mundo” con un contador.

```
section .data
 msg: DB 'Hola Mundo 0', 10
  largo EQU $ - msg

  global _start
section .text

  printHolaMundo:
    push rbx
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, rdi    ; mensaje
    mov rdx, rsi    ; longitud
    int 0x80
    pop rbx
    ret
```




```
_start:
  mov rbx, 10
  ciclo:
    mov rdi, msg
    mov rsi, largo
    call printHolaMundo
    inc byte [msg+largo-2]
    sub rbx, 1
    cmp rbx, 0
    jnz ciclo

  mov rax, 1      ; funcion 1
  mov rbx, 0      ; codigo
  int 0x80
```



Terminal

```
$ ./printCiclo.out
Hola Mundo 0
Hola Mundo 1
Hola Mundo 2
Hola Mundo 3
...
Hola Mundo 9
$
```

Ejemplo de instrucciones `Call` y `Ret`

Recorrer un arreglo de 16 enteros de 32 bits sin signo y contar la cantidad de valores múltiplos de dos.

```
section .data
  data: DD 3,2,43,5,2,43,345,46,24,2,1,24,2,1,13,12
  size: DD 16

  global _start
section .text

  _start:
    mov rax, 0
    mov rcx, 0
    mov rbx, data
    ciclo:
      mov edx, [rbx+rcx*4]
      and edx, 1
      jnz noEsPar
        inc rax
      noEsPar:
      inc ecx
      cmp ecx, [size]
      jnz ciclo
```

```
mov rdi, rax
call printHex

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

.....

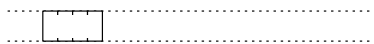
.....

Ejemplo de instrucciones `Call` y `Ret`

Recorrer un arreglo de 16 enteros de 32 bits sin signo y contar la cantidad de valores múltiplos de dos.

```
section .data
  data: DD 3,2,43,5,2,43,345,46,24,2,1,24,2,1,13,12
  size: DD 16

  global _start
section .text

  _start:
    mov rax, 0
    mov rcx, 0
    mov rbx, data
    ciclo:
      mov edx, [rbx+rcx*4]
      and edx, 1
      jnz noEsPar
        inc rax
      noEsPar:
      inc ecx
      cmp ecx, [size]
      jnz ciclo
```

```
mov rdi, rax
call printHex

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```



Ejemplo de instrucciones `Call` y `Ret`

Recorrer un arreglo de 16 enteros de 32 bits sin signo y contar la cantidad de valores múltiplos de dos.

```
section .data
  data: DD 3,2,43,5,2,43,345,46,24,2,1,24,2,1,13,12
  size: DD 16

  global _start
section .text

  _start:
    mov rax, 0
    mov rcx, 0
    mov rbx, data
    ciclo:
      mov edx, [rbx+rcx*4]
      and edx, 1
      jnz noEsPar
        inc rax
      noEsPar:
      inc ecx
      cmp ecx, [size]
      jnz ciclo
```

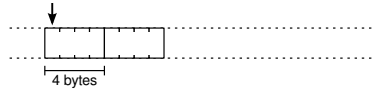
```
mov rdi, rax
call printHex

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

Ejemplo de instrucciones `Call` y `Ret`

Recorrer un arreglo de 16 enteros de 32 bits sin signo y contar la cantidad de valores múltiplos de dos.

```
section .data
  data: DD 3,2,43,5,2,43,345,46,24,2,1,24,2,1,13,12
  size: DD 16

  global _start
section .text

  _start:
    mov rax, 0
    mov rcx, 0
    mov rbx, data
    ciclo:
      mov edx, [rbx+rcx*4]
      and edx, 1
      jnz noEsPar
        inc rax
      noEsPar:
      inc ecx
      cmp ecx, [size]
      jnz ciclo
```

```
mov rdi, rax
call printHex

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

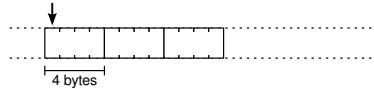


Ejemplo de instrucciones `Call` y `Ret`

Recorrer un arreglo de 16 enteros de 32 bits sin signo y contar la cantidad de valores múltiplos de dos.

```
section .data
  data: DD 3,2,43,5,2,43,345,46,24,2,1,24,2,1,13,12
  size: DD 16

  global _start
section .text

  _start:
    mov rax, 0
    mov rcx, 0
    mov rbx, data
    ciclo:
      mov edx, [rbx+rcx*4]
      and edx, 1
      jnz noEsPar
        inc rax
      noEsPar:
      inc ecx
      cmp ecx, [size]
      jnz ciclo
```

```
mov rdi, rax
call printHex

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

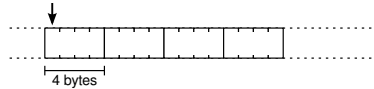


Ejemplo de instrucciones `Call` y `Ret`

Recorrer un arreglo de 16 enteros de 32 bits sin signo y contar la cantidad de valores múltiplos de dos.

```
section .data
  data: DD 3,2,43,5,2,43,345,46,24,2,1,24,2,1,13,12
  size: DD 16

  global _start
section .text

  _start:
    mov rax, 0
    mov rcx, 0
    mov rbx, data
    ciclo:
      mov edx, [rbx+rcx*4]
      and edx, 1
      jnz noEsPar
        inc rax
      noEsPar:
      inc ecx
      cmp ecx, [size]
      jnz ciclo
```

```
mov rdi, rax
call printHex

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

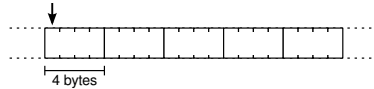


Ejemplo de instrucciones `Call` y `Ret`

Recorrer un arreglo de 16 enteros de 32 bits sin signo y contar la cantidad de valores múltiplos de dos.

```
section .data
  data: DD 3,2,43,5,2,43,345,46,24,2,1,24,2,1,13,12
  size: DD 16

  global _start
section .text

  _start:
    mov rax, 0
    mov rcx, 0
    mov rbx, data
    ciclo:
      mov edx, [rbx+rcx*4]
      and edx, 1
      jnz noEsPar
        inc rax
      noEsPar:
      inc ecx
      cmp ecx, [size]
      jnz ciclo
```

```
mov rdi, rax
call printHex

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

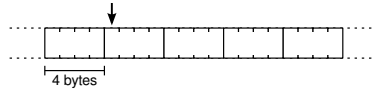


Ejemplo de instrucciones `Call` y `Ret`

Recorrer un arreglo de 16 enteros de 32 bits sin signo y contar la cantidad de valores múltiplos de dos.

```
section .data
  data: DD 3,2,43,5,2,43,345,46,24,2,1,24,2,1,13,12
  size: DD 16

  global _start
section .text

  _start:
    mov rax, 0
    mov rcx, 0
    mov rbx, data
    ciclo:
      mov edx, [rbx+rcx*4]
      and edx, 1
      jnz noEsPar
        inc rax
      noEsPar:
      inc ecx
      cmp ecx, [size]
      jnz ciclo
```

```
mov rdi, rax
call printHex

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

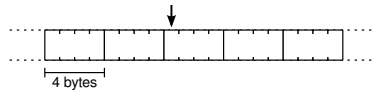


Ejemplo de instrucciones `Call` y `Ret`

Recorrer un arreglo de 16 enteros de 32 bits sin signo y contar la cantidad de valores múltiplos de dos.

```
section .data
  data: DD 3,2,43,5,2,43,345,46,24,2,1,24,2,1,13,12
  size: DD 16

  global _start
section .text

  _start:
    mov rax, 0
    mov rcx, 0
    mov rbx, data
    ciclo:
      mov edx, [rbx+rcx*4]
      and edx, 1
      jnz noEsPar
        inc rax
      noEsPar:
      inc ecx
      cmp ecx, [size]
      jnz ciclo
```

```
mov rdi, rax
call printHex

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

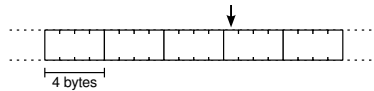


Ejemplo de instrucciones `Call` y `Ret`

Recorrer un arreglo de 16 enteros de 32 bits sin signo y contar la cantidad de valores múltiplos de dos.

```
section .data
  data: DD 3,2,43,5,2,43,345,46,24,2,1,24,2,1,13,12
  size: DD 16

  global _start
section .text

  _start:
    mov rax, 0
    mov rcx, 0
    mov rbx, data
    ciclo:
      mov edx, [rbx+rcx*4]
      and edx, 1
      jnz noEsPar
        inc rax
      noEsPar:
      inc ecx
      cmp ecx, [size]
      jnz ciclo
```

```
mov rdi, rax
call printHex

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

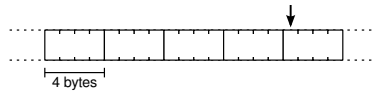


Ejemplo de instrucciones `Call` y `Ret`

Recorrer un arreglo de 16 enteros de 32 bits sin signo y contar la cantidad de valores múltiplos de dos.

```
section .data
  data: DD 3,2,43,5,2,43,345,46,24,2,1,24,2,1,13,12
  size: DD 16

  global _start
section .text

  _start:
    mov rax, 0
    mov rcx, 0
    mov rbx, data
    ciclo:
      mov edx, [rbx+rcx*4]
      and edx, 1
      jnz noEsPar
        inc rax
      noEsPar:
      inc ecx
      cmp ecx, [size]
      jnz ciclo
```

```
mov rdi, rax
call printHex

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

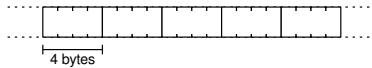


Ejemplo de instrucciones `Call` y `Ret`

Recorrer un arreglo de 16 enteros de 32 bits sin signo y contar la cantidad de valores múltiplos de dos.

```
section .data
  data: DD 3,2,43,5,2,43,345,46,24,2,1,24,2,1,13,12
  size: DD 16

  global _start
section .text

  _start:
    mov rax, 0      ; Inicializo RAX en cero
    mov rcx, 0
    mov rbx, data
    ciclo:
      mov edx, [rbx+rcx*4]
      and edx, 1
      jnz noEsPar
        inc rax
      noEsPar:
      inc ecx
      cmp ecx, [size]
      jnz ciclo
```

```
mov rdi, rax
call printHex

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

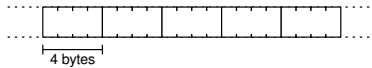


Ejemplo de instrucciones Call y Ret

Recorrer un arreglo de 16 enteros de 32 bits sin signo y contar la cantidad de valores múltiplos de dos.

```
section .data
  data: DD 3,2,43,5,2,43,345,46,24,2,1,24,2,1,13,12
  size: DD 16

  global _start
section .text

  _start:
    mov rax, 0      ; Inicializo RAX en cero
    mov rcx, 0      ; Uso RCX como indice del arreglo
    mov rbx, data
    ciclo:
      mov edx, [rbx+rcx*4]
      and edx, 1
      jnz noEsPar
        inc rax
      noEsPar:
      inc ecx
      cmp ecx, [size]
      jnz ciclo
```

```
mov rdi, rax
call printHex

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

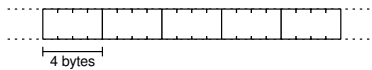


Ejemplo de instrucciones Call y Ret

Recorrer un arreglo de 16 enteros de 32 bits sin signo y contar la cantidad de valores múltiplos de dos.

```
section .data
  data: DD 3,2,43,5,2,43,345,46,24,2,1,24,2,1,13,12
  size: DD 16

  global _start
section .text

  _start:
    mov rax, 0      ; Inicializo RAX en cero
    mov rcx, 0      ; Uso RCX como indice del arreglo
    mov rbx, data   ; Guardo en RBX el comienzo del arreglo
    ciclo:
      mov edx, [rbx+rcx*4]
      and edx, 1
      jnz noEsPar
        inc rax
      noEsPar:
      inc ecx
      cmp ecx, [size]
      jnz ciclo
```

```
mov rdi, rax
call printHex

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

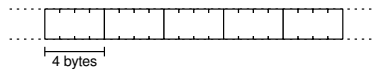


Ejemplo de instrucciones `Call` y `Ret`

Recorrer un arreglo de 16 enteros de 32 bits sin signo y contar la cantidad de valores múltiplos de dos.

```
section .data
  data: DD 3,2,43,5,2,43,345,46,24,2,1,24,2,1,13,12
  size: DD 16

  global _start
section .text

  _start:
    mov rax, 0      ; Inicializo RAX en cero
    mov rcx, 0      ; Uso RCX como indice del arreglo
    mov rbx, data   ; Guardo en RBX el comienzo del arreglo
    ciclo:
      mov edx, [rbx+rcx*4]   ; Comienzo + Indice * tamaño
      and edx, 1
      jnz noEsPar
        inc rax
      noEsPar:
      inc ecx
      cmp ecx, [size]
      jnz ciclo
```

```
mov rdi, rax
call printHex

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

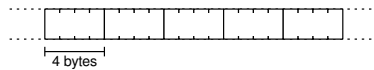


Ejemplo de instrucciones Call y Ret

Recorrer un arreglo de 16 enteros de 32 bits sin signo y contar la cantidad de valores múltiplos de dos.

```
section .data
  data: DD 3,2,43,5,2,43,345,46,24,2,1,24,2,1,13,12
  size: DD 16

  global _start
section .text

  _start:
    mov rax, 0      ; Inicializo RAX en cero
    mov rcx, 0      ; Uso RCX como indice del arreglo
    mov rbx, data   ; Guardo en RBX el comienzo del arreglo
    ciclo:
      mov edx, [rbx+rcx*4]   ; Comienzo + Indice * tamaño
      and edx, 1
      jnz noEsPar
        inc rax   ; Incremento si es par
      noEsPar:
      inc ecx
      cmp ecx, [size]
      jnz ciclo
```

```
mov rdi, rax
call printHex

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

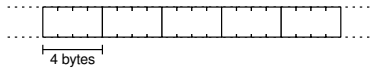


Ejemplo de instrucciones Call y Ret

Recorrer un arreglo de 16 enteros de 32 bits sin signo y contar la cantidad de valores múltiplos de dos.

```
section .data
  data: DD 3,2,43,5,2,43,345,46,24,2,1,24,2,1,13,12
  size: DD 16

  global _start
section .text

  _start:
    mov rax, 0      ; Inicializo RAX en cero
    mov rcx, 0      ; Uso RCX como indice del arreglo
    mov rbx, data   ; Guardo en RBX el comienzo del arreglo
    ciclo:
      mov edx, [rbx+rcx*4]   ; Comienzo + Indice * tamaño
      and edx, 1
      jnz noEsPar
        inc rax   ; Incremento si es par
      noEsPar:
      inc ecx           ; Incremento indice, siguiente valor
      cmp ecx, [size]
      jnz ciclo
```

```
mov rdi, rax
call printHex

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```

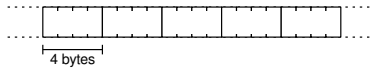


Ejemplo de instrucciones `Call` y `Ret`

Recorrer un arreglo de 16 enteros de 32 bits sin signo y contar la cantidad de valores múltiplos de dos.

```
section .data
  data: DD 3,2,43,5,2,43,345,46,24,2,1,24,2,1,13,12
  size: DD 16

  global _start
section .text

  _start:
    mov rax, 0      ; Inicializo RAX en cero
    mov rcx, 0      ; Uso RCX como indice del arreglo
    mov rbx, data   ; Guardo en RBX el comienzo del arreglo
    ciclo:
      mov edx, [rbx+rcx*4]   ; Comienzo + Indice * tamaño
      and edx, 1
      jnz noEsPar
        inc rax   ; Incremento si es par
      noEsPar:
      inc ecx           ; Incremento indice, siguiente valor
      cmp ecx, [size]   ; ¿Llegue al final?
      jnz ciclo
```

```
mov rdi, rax
call printHex

mov rax, 1      ; funcion 1
mov rbx, 0      ; codigo
int 0x80
```



Convención para preservar registros

Para no tener que preservar todos los registros antes de llamar a una función.
Se respeta una convención.

Reglas:

- Preservar en cualquier función los registros RBX, R12, R13, R14, R15, RBP y RSP.
  Se deben preservar SOLO los registros que se modifican.
- Retornar el resultado a través de RAX si es un valor entero.
  RDX:RAX si ocupa 128bits o XMM0, si es un número de punto flotante.
- Preservar la consistencia de la pila.
- La pila opera alineada a 8 bytes.

En llamados al sistema se deben preservar todos los registros.

Pasaje de parámetros

Los parámetros se pasan por registro, de izquierda a derecha según la firma de la función, clasificados por tipo:

- **Enteros y direcciones de memoria**: RDI, RSI, RDX, RCX, R8 y R9.
- **Punto flotante**: XMM0 a XMM7.
- **Resto** de los parámetros que superen la cantidad de registros se ubican en la pila.

En el contexto de la materia solo vamos a utilizar valores enteros y direcciones de memoria.

Pasaje de parámetros

Los parámetros se pasan por registro, de izquierda a derecha según la firma de la función, clasificados por tipo:

- **Enteros y direcciones de memoria**: RDI, RSI, RDX, RCX, R8 y R9.
- **Punto flotante**: XMM0 a XMM7.
- **Resto** de los parámetros que superen la cantidad de registros se ubican en la pila.

En el contexto de la materia solo vamos a utilizar valores enteros y direcciones de memoria.

Ejemplo

Una función denominada `sumar` que toma un número de 16 bits, otro de 8 bits, los suma y el resultado lo retorna como un número de 32 bits.
Su firma en **C** se escribe como: `int32_t sumar(int16_t a, int8_t b)`
DI es el parámetro `a`.
SIL es el parámetro `b`.
En **EAX** se retorna el resultado.

Bibliografía

- Tanenbaum, “Organización de Computadoras. Un Enfoque Estructurado”, 4ta Edición, 2000.
  - **Capítulo 5 - El nivel de Arquitectura del Conjunto de Instrucciones**
    - 5.5 Tipos de Instrucciones - Páginas 348 - 355
    - 5.6 Control de Flujo - Páginas 370 - 380

- Null, “Essentials of Computer Organization and Architecture”, 5th Edition, 2018.
  - **Chapter 5 - A Closer Look at Instruction Set Architectures**
    - 5.2 Instruction Formats
    - 5.3 Instruction Types
    - 5.4 Addressing

Referencias

- Intel ® 64 and IA-32 Architectures - Software Developer’s Manual
  `https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html`
- System V Application Binary Interface - AMD64 Architecture Processor Supplement
  `https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf`
- Linux System Call Table
  `http://faculty.nps.edu/cseagle/assembly/sys_call.html`

¡Gracias!

Recuerden leer los comentarios adjuntos
en cada clase por aclaraciones.

Notas Complementarias

Ejecución

```
$ ./holamundo
```

Ejecución

```
$ ./holamundo
Hola Mundo
```

Ejecución

```
$ ./holamundo
Hola Mundo
```

¿Dondé quedó el código de error?

Ejecución

```
$ ./holamundo
Hola Mundo
```

¿Dondé quedó el código de error?

```
$ echo "$?"
```

Ejecución

```
$ ./holamundo
Hola Mundo
```

¿Dondé quedó el código de error?

```
$ echo "$?"
0
```

Ejecución

```
$ ./holamundo
Hola Mundo
```

¿Dondé quedó el código de error?

```
$ echo "$?"
0
```

¿Cómo reconozco un binario?

Ejecución

```
$ ./holamundo
Hola Mundo
```

¿Dondé quedó el código de error?

```
$ echo "$?"
0
```

¿Cómo reconozco un binario?

```
$ file holamundo
```

Ejecución

```
$ ./holamundo
Hola Mundo
```

¿Dondé quedó el código de error?

```
$ echo "$?"
0
```

¿Cómo reconozco un binario?

```
$ file holamundo
holamundo: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
 statically linked, with debug_info, not stripped
```

Ejecución

Ya sabemos que es un `elf`. Pero, ¿qué tiene adentro?

Ejecución

Ya sabemos que es un `elf`. Pero, ¿qué tiene adentro?

```
$ objdump -d holamundo
```

Ejecución

Ya sabemos que es un `elf`. Pero, ¿qué tiene adentro?

```
$ objdump -d holamundo

holamundo:     file format elf64-x86-64


Disassembly of section .text:

00000000004000b0 <_start>:
  4000b0: b8 04 00 00 00         mov    $0x4,%eax
  4000b5: bb 01 00 00 00         mov    $0x1,%ebx
  4000ba: 48 b9 d8 00 60 00 00   movabs $0x6000d8,%rcx
  4000c1: 00 00 00
  4000c4: ba 0b 00 00 00         mov    $0xb,%edx
  4000c9: cd 80                  int    $0x80
  4000cb: b8 01 00 00 00         mov    $0x1,%eax
  4000d0: bb 00 00 00 00         mov    $0x0,%ebx
  4000d5: cd 80                  int    $0x80
```

Ejecución

Y la sección de datos:

Ejecución

Y la sección de datos:

```
$ objdump -D holamundo
```

Ejecución

Y la sección de datos:

```
$ objdump -D holamundo

...

Disassembly of section .data:

00000000006000d8 <msg>:
  6000d8: 48 6f                 rex.W outsl %ds:(%rsi),(%dx)
  6000da: 6c                    insb   (%dx),%es:(%rdi)
  6000db: 61                    (bad)
  6000dc: 20 4d 75              and    %cl,0x75(%rbp)
  6000df: 6e                    outsb  %ds:(%rsi),(%dx)
  6000e0: 64 6f                 outsl  %fs:(%rsi),(%dx)
  6000e2: 0a                    .byte 0xa

...
```

Ejecución

Y si quiero encontrar una cadena de texto:

Ejecución

Y si quiero encontrar una cadena de texto:

```
$ strings holamundo
```

Ejecución

Y si quiero encontrar una cadena de texto:

```
$ strings holamundo

Hola Mundo
holamundo.asm
NASM 2.13.02
...
largo
...
.symtab
.strtab
...
.text
.data
.debug_aranges
.debug_pubnames
.debug_info
...
```

Ejecución

¿Cómo ver los símbolos dentro de un binario?

Ejecución

¿Cómo ver los símbolos dentro de un binario?

```
$ nm holamundo
```

Ejecución

¿Cómo ver los símbolos dentro de un binario?

```
$ nm holamundo
```

```
00000000006000e3 D __bss_start
00000000006000e3 D _edata
00000000006000e8 D _end
000000000000000b a largo
00000000006000d8 d msg
00000000004000b0 T _start
```

Ejecución

¿Cómo ver los símbolos dentro de un binario?

```
$ nm holamundo
```

```
00000000006000e3 D __bss_start
00000000006000e3 D _edata
00000000006000e8 D _end
000000000000000b a largo
00000000006000d8 d msg
00000000004000b0 T _start
```

¿Y dónde se aprende cómo usar estos bonitos comandos?

Ejecución

¿Cómo ver los símbolos dentro de un binario?

```
$ nm holamundo
```

```
00000000006000e3 D __bss_start
00000000006000e3 D _edata
00000000006000e8 D _end
000000000000000b a largo
00000000006000d8 d msg
00000000004000b0 T _start
```

¿Y dónde se aprende cómo usar estos bonitos comandos?

En el manual de referencia.

Ejecución

¿Cómo ver los símbolos dentro de un binario?

```
$ nm holamundo
```

```
00000000006000e3 D __bss_start
00000000006000e3 D _edata
00000000006000e8 D _end
000000000000000b a largo
00000000006000d8 d msg
00000000004000b0 T _start
```

¿Y dónde se aprende cómo usar estos bonitos comandos?

En el manual de referencia.

¿Cómo accedo al manual?

Ejecución

¿Cómo ver los símbolos dentro de un binario?

```
$ nm holamundo
```

```
00000000006000e3 D __bss_start
00000000006000e3 D _edata
00000000006000e8 D _end
000000000000000b a largo
00000000006000d8 d msg
00000000004000b0 T _start
```

¿Y dónde se aprende cómo usar estos bonitos comandos?

En el manual de referencia.

¿Cómo accedo al manual?

Mediante el comando `man`.

```
$ man comando
```

Ejecución

```
$ man nm
```

```
NM(1)                                   GNU Development Tools                                   NM(1)

NAME
       nm - list symbols from object files

SYNOPSIS
       nm [-A|-o|--print-file-name] [-a|--debug-syms]
          [-B|--format=bsd] [-C|--demangle[=style]]
          [-D|--dynamic] [-fformat|--format=format]
          [-g|--extern-only] [-h|--help]
          [-l|--line-numbers] [--inlines]
          [-n|-v|--numeric-sort]
          [-P|--portability] [-p|--no-sort]
          [-r|--reverse-sort] [-S|--print-size]
          [-s|--print-armap] [-t radix|--radix=radix]
          [-u|--undefined-only] [-V|--version]
          [-X 32_64] [--defined-only] [--no-demangle]
          [--plugin name] [--size-sort] [--special-syms]
          [--synthetic] [--with-symbol-versions] [--target=bfdname]
          [objfile...]

DESCRIPTION
       GNU nm lists the symbols from object files objfile....  If no object files are listed as
       arguments, nm assumes the file a.out.

       For each symbol, nm shows:

       ·   The symbol value, in the radix selected by options (see below), or hexadecimal by default.
```