Práctica de x86 Assembly y ABI

Parte 1: x86 Assembly

Segundo Cuatrimestre 2024

Arquitectura y Organización de Computadores
DC - UBA

Estructura de la clase

Hoy vamos a ver:

- Programación en Assembly x86

- Convención C

- Uso y administración de la pila

- Ejercitación

Introducción

Los manuales de desarrollador de software Intel

1. Intel® 64 and IA-32 Architectures Software Developer's Manual **Volume 1: Basic Architecture**

2. Intel® 64 and IA-32 Architectures Software Developer's Manual **Volume 2: Instruction Set Reference, A-Z**

---

3. Intel® 64 and IA-32 Architectures Software Developer's Manual **Volume 3: System Programming Guide**

Entorno de ejecución

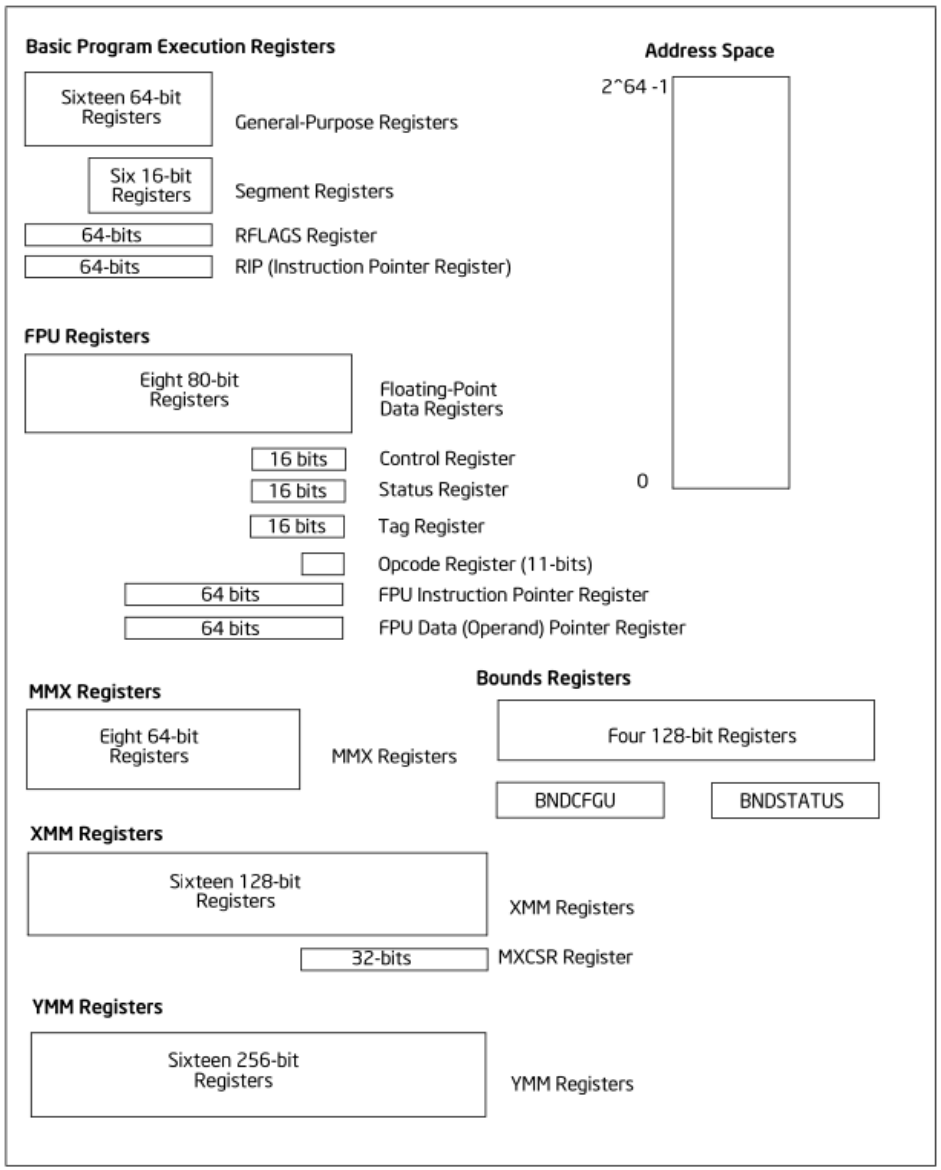
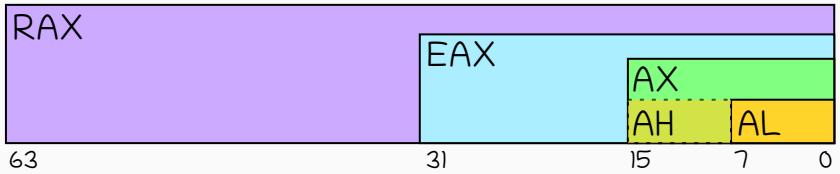
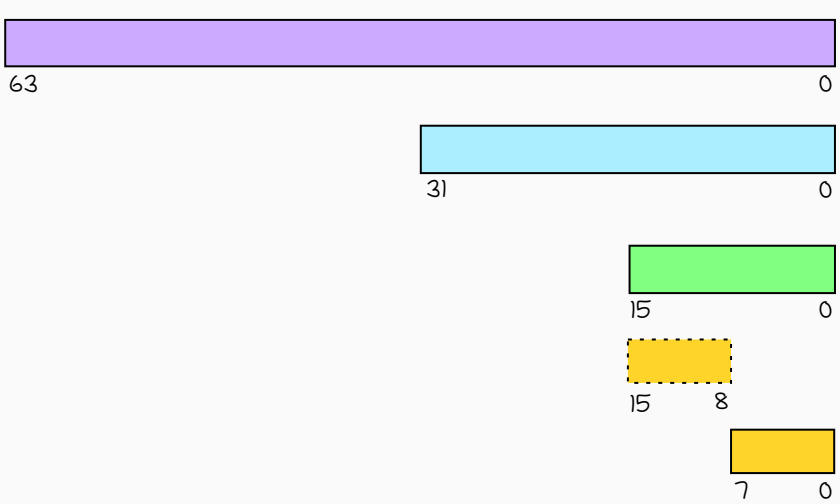
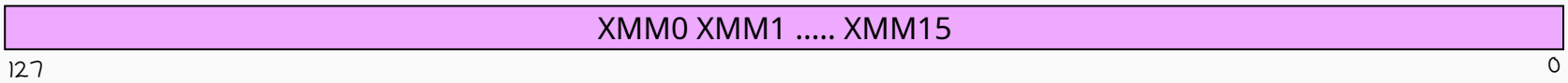


Figure 3-2. 64-Bit Mode Execution Environment

Registros


RAX
EAX
AX
AH
AL
63
31
15
7
0


63
0
RAX RBX RCX RDX RSI RDI RSP RBP
R8 R9 R10 R11 R12 R13 R14 R15
31
0
EAX EBX ECX EDX ESI EDI ESP EBP
R8D R9D R10D R11D R12D R13D R14D R15D
15
0
AX BX CX DX SI DI SP BP
R8W R9W R10W R11W R12W R13W R14W R15W
15
8
AH BH CH DH
7
0
AL BL CL DL SIL DIL SPL BPL
R8B R9B R10B R11B R12B R13B R14B R15B


XMM0 XMM1 ..... XMM15
127
0

Programación en Assembly x86

Estructura de programa en asm


```
1 ; Programa Hola Mundo que usa syscall de x86_64
2 ;
3 ;* Obtenido de man 2 syscall
4 ;* En x86-64
5 ;*
6 ;* System  Ret  Ret  Error
7 ;* call #  val  val2    -
8 ;* rax     rax  rdx
9 ;*
10 ;* Los parámetros de syscall deben pasarse así:
11 ;* arg1  arg2  arg3  arg4  arg5  arg6
12 ;* rdi   rsi   rdx   r10   r8    r9
13
14 %define SYS_WRITE 1
15 %define SYS_EXIT 60
16 %define STDOUT 1
17
18 section .data
19 msg db  '¡Hola Mundo!', 10
20 len EQU $ - msg
21
22 global _start
23 section .text
24 _start:
25     mov     rax, SYS_WRITE  ;ssize_t write(int fd, const void *buf, size_t count);
26     mov     rdi, STDOUT
27     mov     rsi, msg
28     mov     rdx, len
29     syscall                 ; fast system call ; RCX    --> dirección de retorno
30                                                ; RFLAGS --> R11
31     ; en este punto RAX tiene los bytes escritos por sys_write()
32     mov     rax, SYS_EXIT   ; void exit(int status);
33     mov     rdi, 0
34     syscall
```

directivas de preprocesador

directivas de ensamblador

Estructura de programa en asm


```
1  ; Programa Hola Mundo que usa syscall de x86_64
2  ;
3  ;* Obtenido de man 2 syscall
4  ;* En x86-64
5  ;*
6  ;* System  Ret  Ret  Error
7  ;* call #  val  val2     -
8  ;* rax     rax  rdx
9  ;*
10 ;* Los parámetros de syscall deben pasarse así:
11 ;* arg1  arg2  arg3  arg4  arg5  arg6
12 ;* rdi   rsi   rdx   r10   r8    r9
13
14 %define SYS_WRITE 1
15 %define SYS_EXIT 60
16 %define STDOUT 1
17
18 section .data
19 msg db  '¡Hola Mundo!', 10
20 len EQU $ - msg
21
22 global _start
23 section .text
24 _start:
25     mov     rax, SYS_WRITE  ;ssize_t write(int fd, const void *buf, size_t count);
26     mov     rdi, STDOUT
27     mov     rsi, msg
28     mov     rdx, len
29     syscall                 ; fast system call ; RCX    --> dirección de retorno
30                                                ; RFLAGS --> R11
31     ; en este punto RAX tiene los bytes escritos por sys_write()
32     mov     rax, SYS_EXIT   ; void exit(int status);
33     mov     rdi, 0
34     syscall
```

pseudo-instrucciones

Estructura de programa en asm


```
1 ; Programa Hola Mundo que usa syscall de x86_64
2 ;
3 ;* Obtenido de man 2 syscall
4 ;* En x86-64
5 ;*
6 ;* System  Ret  Ret  Error
7 ;* call #  val  val2     -
8 ;* rax     rax  rdx
9 ;*
10 ;* Los parámetros de syscall deben pasarse así:
11 ;* arg1  arg2  arg3  arg4  arg5  arg6
12 ;* rdi   rsi   rdx   r10   r8    r9
13
14 %define SYS_WRITE 1
15 %define SYS_EXIT 60
16 %define STDOUT 1
17
18 section .data                                                    sección .data
19 msg db  '¡Hola Mundo!', 10
20 len EQU $ - msg
21
22 global _start
23 section .text                                                    sección .text
24 _start:
25     mov     rax, SYS_WRITE  ;ssize_t write(int fd, const void *buf, size_t count);
26     mov     rdi, STDOUT
27     mov     rsi, msg
28     mov     rdx, len
29     syscall                 ; fast system call ; RCX    --> dirección de retorno
30                                                ; RFLAGS --> R11
31     ; en este punto RAX tiene los bytes escritos por sys_write()
32     mov     rax, SYS_EXIT   ; void exit(int status);
33     mov     rdi, 0
34     syscall
```

Instrucciones

Los operandos pueden ser:

- Registros:

```
add rdi, rsi
```

- Memoria:

```
mov [LABEL], rax
sub r10,[rdx + 4*rax + OFFSET]
```

- Inmediatos:

```
sub rsp,8
jne fin
```

Memoria

- `[displacement]`
- `[reg]`
- `[reg + reg*scale]` `; scale es 1, 2, 4, u 8`
- `[reg + displacement]`
- `[reg + reg*scale + displacement]`

Forma general

[Base	+	(Index	*	Scale	)	+	Displacement]
RAX		RAX		1			Cte. 32 bits
...		...		2			
R15		R15		4			
		(NO RSP)		8			

Memoria


```
[1003]                  ; sólo displacement
[rbx]                   ; registro base solamente
[rbx + rsi*4]           ; base + index * scale
[rbx + rdx]             ; scale es 1
[rax - 8]               ; displacement es -8
[rax + rdi*8 - 200]     ; todos los componentes
[rbx + counter]         ; dirección de variable 'counter'
                        ; como desplazamiento
```

Inmediatos


```
200             ; decimal
0200            ; sigue siendo decimal - el 0 no lo hace octal
0xc8            ; hexa - el querido 0x
0hc8            ; hexa - 0h también es aceptado
0q310           ; octal - prefijo 0q
11001000b       ; binario - sufijo b
0b1100_1000     ; binario - prefijo 0b, guiones permitidos
```

Instrucciones

La mayoría de las instrucciones con 2 operandos toman la siguiente forma:

- `inst reg, reg`
- `inst reg, mem`
- `inst reg, imm`
- `inst mem, reg`
- `inst mem, imm`

Pseudo-Instrucciones


```
db    0x55                  ; sólo el byte 0x55
db    0x55,0x56,0x57        ; 3 bytes sucesivos
db    'a',0x55              ; 0x97, 0x55
db    'hello',13,10,'$'     ; strings como cadenas de bytes
db    `hola\nmundo\n\0`     ; strings con "C-style \-escapes"
dw    0x1234                ; 0x34 0x12
dd    0x12345678            ; 0x78 0x56 0x34 0x12
dq    0x123456789abcdef0    ; constante de 8 bytes
times 4 db 'ja'             ; "jajajaja"
```

Pseudo-Instrucciones

Para reservar espacio (sin inicializar):

```
buffer:         resb    64              ; reserva 64 bytes
wordvar:        resw    1               ; reserva un word
realarray:      resq    10              ; array de 10 qwords
```

Estas pseudo-instrucciones deben ir en **section .bss**

EQU vs %define


```
1  .data
2  msg1 db 'Hola!', 10
3  len EQU $ - msg1
4  msg2 db ':(', 0
5  ; len EQU $ - msg2 ;ERROR!
6  ...
7  add rax, len
8  ...
9  ; len = 6 siempre!
10 ...
11 mov rcx, len
12 ...
13
```

```
1  %define len(x,y) (x+1-y)
2  .data
3  msg1 db 'Hola!'
4  endmsg1 db 10
5  msg2 db ':)'
6  endmsg2 db 0
7  ...
8  ; len se reemplaza por 6:
9  add rax, len(endmsg1, msg1)
10 ...
11 ; len se reemplaza por 3:
12 mov rcx, len(endmsg2, msg2)
13 ...
```

Ensamblado con NASM y formato de salida ELF

- Usaremos el ensamblador **NASM**.
- NASM esta diseñado para varios sistemas operativos y distintas versiones de ANSI-C por eso, hay varios formatos de salida.
- Dado que vamos a trabajar en **Linux**, vamos a utilizar elfx32 , elf32 y elf64, que generan **formatos de salida ELF32 y ELF64** (Executable and Linkable Format) en los archivos objetos [1].

[1] Más información:
`https://www.nasm.us/xdoc/2.15.05/html/nasmdoc8.html#section-8.9.2`

Ensamblado y linkeo

Para ensamblar:

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

Linking:

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

Ejecutamos:

```
$ ./holamundo
```

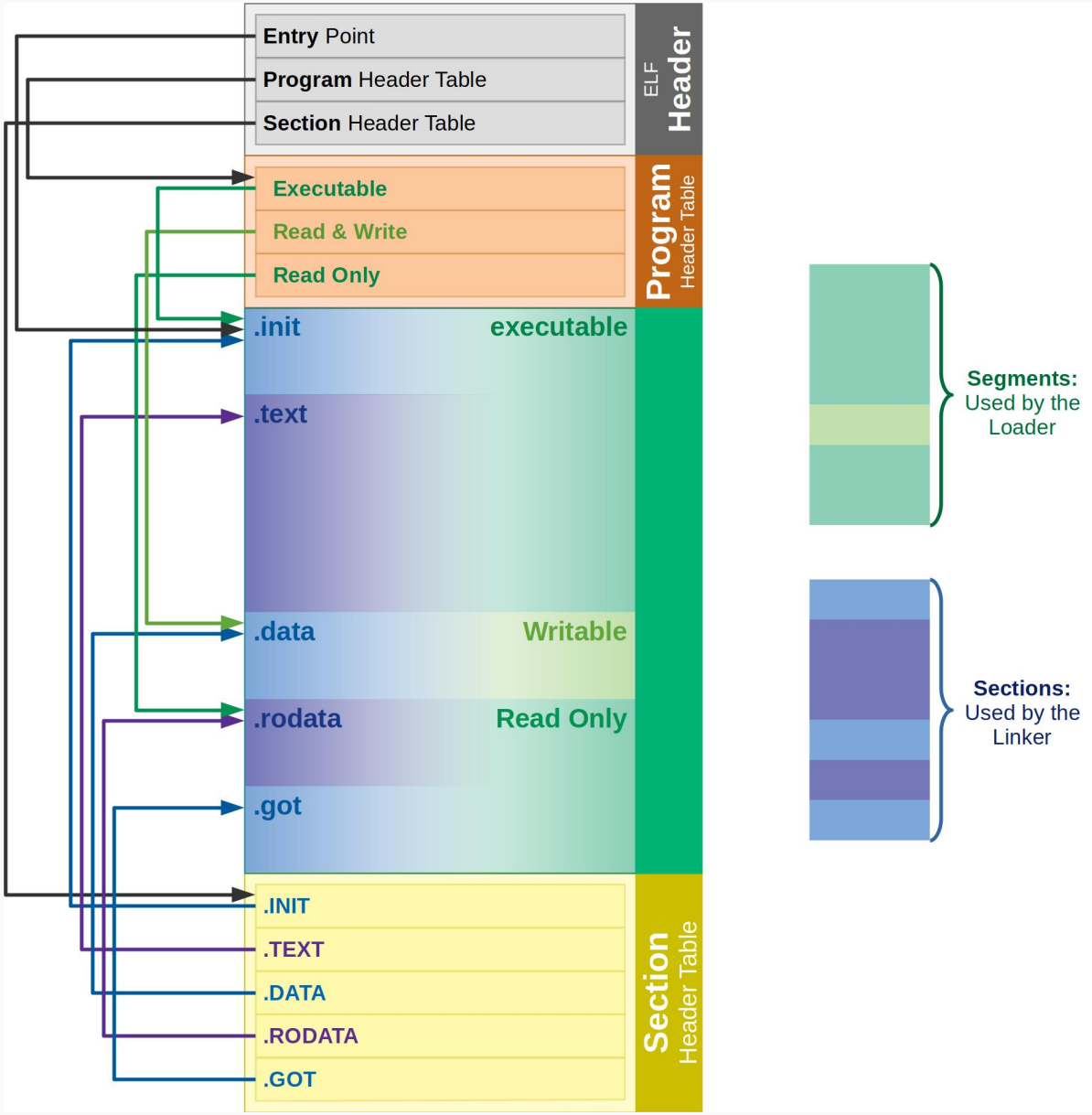
Makefile

Makefile básico para nasm:

```
AS      := nasm
ASFLAGS := -f elf64 -F DWARF -g -Wall
LD      := ld
LDFLAGS := -g
TARGET  := holaorga

.PHONY: all clean
all: $(TARGET)
# assembly
holaorga.o: holaorga.asm
        $(AS) $(ASFLAGS) $<
# linking
$(TARGET): holaorga.o
        $(LD) $(LDFLAGS) $< -o $@
clean:
        rm -rf *.o $(TARGET)
```

Conociendo la plataforma... ELF!



Consultas