

Estructura de Computadores

Práctica 1: Entorno de desarrollo GNU

Gustavo Romero López

Updated: 11 de septiembre de 2023

Ingeniería de Computadores, Automática y Robótica

- 1. Índice
- 2. Objetivos
- 3. Introducción
- 4. C
- 5. Ensamblador
- 6. Ejemplos
 - 6.1 hola
 - 6.2 make
 - 6.3 Ejemplo en C
 - 6.4 Ejemplo en C++
 - 6.5 Ejemplo en 32 bits
 - 6.6 Ejemplo en 64 bits
 - 6.7 ASM + C
 - 6.8 Optimización
- 7. Compiler Explorer
- 8. Cutter
- 9. Enlaces

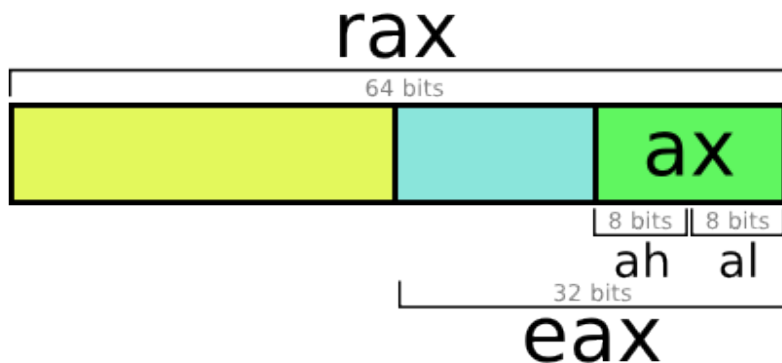
Objetivos

- ⦿ Nociones de ensamblador 80x86 de 64 bits.
- ⦿ Linux es tu amigo: si no sabes algo pregunta... **man**.
- ⦿ Hoy aprenderemos varias cosas:
 - El esqueleto de un programa básico en ensamblador.
 - Como aprender de un maestro: el compilador **gcc**.
 - Herramientas clásicas del entorno de programación UNIX:
 - **make**: hará el trabajo sucio y rutinario por nosotros.
 - **as**: el ensamblador.
 - **ld**: el enlazador.
 - **gcc**: el compilador.
 - **nm**: lista los símbolos de un fichero.
 - **objdump**: el desensamblador.
 - Herramienta web: Compiler Explorer
 - Ingeniería inversa: Cutter

Ensamblador 80x86

- ⦿ Los **80x86** son una familia de procesadores.
- ⦿ El más utilizado junto a los procesadores **ARM**.
- ⦿ En estas prácticas vamos a centrarnos en su **lenguaje ensamblador** (inglés).
- ⦿ El lenguaje ensamblador es el más básico, tras el binario, con el que podemos escribir programas utilizando las **instrucciones** que entiende el procesador.
- ⦿ Cualquier estructura de un lenguajes de alto nivel pueden crearse mediante instrucciones muy sencillas.
- ⦿ Normalmente es utilizado para poder acceder a partes que los lenguajes de alto nivel nos ocultan, complican o hacen de forma inconveniente.

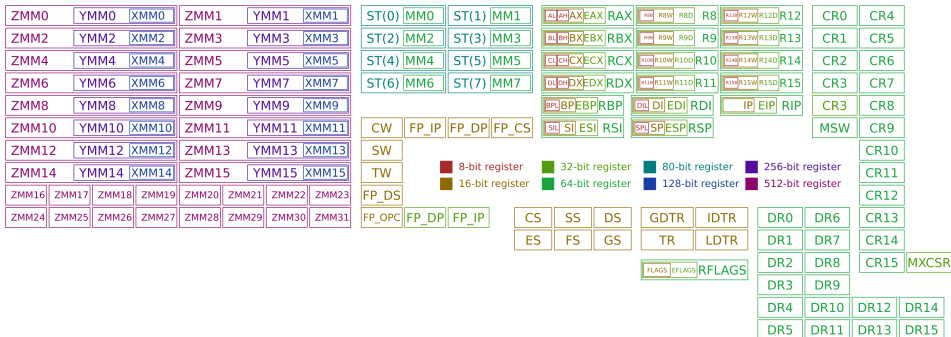
Arquitectura 80x86: el registro A



Arquitectura 80x86: registros de propósito general

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

Arquitectura 80x86: registros completos



Arquitectura 80x86: banderas

eflags register

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
CF	1	PF	0	AF	0	ZF	SF	TF	IF	DF	OF	IOPL	NT	0	RF	VM	AC	VIF	VIP	ID	0	0	0	0	0	0	0	0	0	0	0



Reserved flags



System flags



Arithmetic flags

TF: Trap
IF: Interrupt
DF: Direction

CF: Carry
PF: Parity
AF: Adjust
ZF: Zero
SF: Sign
OF: Overflow

Paso de parámetros a funciones en 64 bits

%rax	Return value	%r8	Argument #5
%rbx	Callee saved	%r9	Argument #6
%rcx	Argument #4	%r10	Caller saved
%rdx	Argument #3	%r11	Caller Saved
%rsi	Argument #2	%r12	Callee saved
%rdi	Argument #1	%r13	Callee saved
%rsp	Stack pointer	%r14	Callee saved
%rbp	Callee saved	%r15	Callee saved

Arquitectura 80x86 en Linux: llamadas al sistema

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			
2	sys_open	const char *filename	int flags	int mode			
3	sys_close	unsigned int fd					
4	sys_stat	const char *filename	struct stat *statbuf				
5	sys_fstat	unsigned int fd	struct stat *statbuf				
6	sys_lstat	const char *filename	struct stat *statbuf				
7	sys_poll	struct poll_fd *ufds	unsigned int nfds	long timeout_msecs			
8	sys_lseek	unsigned int fd	off_t offset	unsigned int origin			
9	sys_mmap	unsigned long addr	unsigned long len	unsigned long prot	unsigned long flags	unsigned long fd	unsigned long off

Programa mínimo en C...

minimo1.c

```
int main() {}
```

minimo2.c

```
int main() { return 0; }
```

minimo3.c

```
#include <stdlib.h>  
int main() { exit(0); }
```

Trasteando el programa mínimo en C

- ⊙ Compilar: `gcc minimo1.c -o minimo1`
- ⊙ ¿Qué he hecho? `file ./minimo1`
- ⊙ ¿Qué contiene? `nm ./minimo1`
- ⊙ Ejecutar: `./minimo1`
- ⊙ ¿Qué devuelve? `./minimo1; echo $?`
- ⊙ Desensamblar: `objdump -d minimo1`
- ⊙ Ver llamadas al sistema: `strace ./minimo1`
- ⊙ Ver llamadas de biblioteca: `ltrace ./minimo1`
- ⊙ ¿Qué bibliotecas usa? `ldd minimo1`

`linux-vdso.so.1 (0x00007ffe2ddbc000)`
`libc.so.6 => /lib64/libc.so.6 (0x00007fbc5043a000)`
`/lib64/ld-linux-x86-64.so.2 (0x0000558dbe5aa000)`
- ⊙ Examinar biblioteca: `objdump -d /lib64/libc.so.6`

Ensamblador desde 0: secciones básicas de un programa

`.data`

`.text`

Ensamblador desde 0: punto de entrada

Si no necesita la biblioteca de C puede evitarla así:

- ⦿ La etiqueta `_start` identifica el programa principal.
- ⦿ Ensamble/Compile con la opción `-nostartfiles`.

```
.text  
_start: .globl _start
```

Si requiere de la biblioteca de C...

- ⦿ La etiqueta `main` identifica el programa principal.
- ⦿ Alinee correctamente la pila al inicio de `main`.

```
.text  
main: .globl main
```

Ensamblador desde 0: datos

.data

msg: .string "¡hola, mundo!\n"

tam: .quad . - msg

Ensamblador desde 0: código

```
write:  mov    $1,    %rax    # write
        mov    $1,    %rdi    # stdout
        mov    $msg, %rsi    # texto
        mov    tam,   %rdx    # tamaño
        syscall                # llamada a write
        ret

exit:   mov    $60,   %rax    # exit
        xor    %rdi, %rdi    # 0
        syscall                # llamada a exit
        ret
```


Ensamblador desde 0: ejemplo básico hola.s

```
1  .data
2  msg:  .string "¡hola, mundo!\n"
3  tam:  .quad . - msg
4
5  .text
6  _start: .globl _start
7          call  write      # llamada a función
8          call  exit       # llamada a función
9
10 write:  mov    $1,  %rax  # write
11         mov    $1,  %rdi  # stdout
12         mov    $msg, %rsi  # texto
13         mov    tam, %rdx  # tamaño
14         syscall          # llamada a write
15         ret
16
17 exit:   mov    $60, %rax  # exit
18         xor    %rdi, %rdi # 0
19         syscall          # llamada a exit
20         ret
```

¿Cómo hacer ejecutable mi programa?

¿Cómo hacer ejecutable el código anterior?

- ⦿ opción a: ensamblar + enlazar
 - `as hola.s -o hola.o`
 - `ld hola.o -o hola`
- ⦿ opción b: compilar = ensamblar + enlazar
 - `gcc -nostdlib -no-pie hola.s -o hola`
- ⦿ opción c: que lo haga alguien por mi → `make`
 - `makefile`: fichero con definiciones, objetivos y recetas.

Ejercicios:

1. Cree un ejecutable a partir de `hola.s`.
2. Use `file` para ver el tipo de cada fichero.
3. Descargue el fichero `makefile`, pruébelo e intente hacer alguna modificación.
4. Examine el código ensamblador con `objdump -d hola`.

```
all: att
```

```
att: $(ATT)
```

```
clean:
```

```
    -rm -fv $(ATT) $(EXE) *~
```

```
exe: $(EXE)
```

```
.PHONY: all att clean exe
```

Ejemplo en C: hola-c.c

```
#include <stdio.h>

int main()
{
    printf("¡hola, mundo!\n");
    return 0;
}
```

- ⊙ ¿Qué hace gcc con mi programa?
- ⊙ La única forma de saberlo es desensamblarlo:
 - Sintaxis AT&T: `objdump -d hola-c`
 - Sintaxis Intel: `objdump -d hola-c -M intel`

Ejercicios:

5. ¿Cómo se imprime el mensaje “hola mundo”?

Ejemplo en C++: hola-c++.cc

```
#include <iostream>
```

```
int main()
```

```
{
```

```
    std::cout << "¡hola, mundo!" << std::endl;
```

```
}
```

- ⦿ ¿Qué hace g++ con mi programa?
- ⦿ La única forma de saberlo es desensamblarlo:
 - Sintaxis AT&T: `objdump -C -d hola-c++`
 - Sintaxis Intel: `objdump -C -d hola-c++ -M intel`

Ejercicios:

6. ¿Qué hace ahora diferente la función `main()` respecto a C?

```
write:  movl  $4, %eax    # write
        movl  $1, %ebx    # salida estándar
        movl  $msg, %ecx  # cadena
        movl  tam, %edx   # longitud
        int   $0x80       # llamada a write
        ret              # retorno

exit:   movl  $1, %eax    # exit
        xorl  %ebx, %ebx  # 0
        int   $0x80       # llamada a exit
```

Puede ser necesario instalar algún paquete especial...

- ⦿ fedora: `sudo dnf -y install glibc-devel.i686`
- ⦿ ubuntu: `sudo apt-get install -y gcc-multilib`

Ejercicios:

7. Si siente curiosidad mire `hola32p.s`. Cabe destacar código de 32 bits, uso de *“little endian”*, llamada a subrutina, uso de la pila y codificación de caracteres.

```
write:  mov    $1,    %rax  # write
        mov    $1,    %rdi  # stdout
        mov    $msg, %rsi  # texto
        mov    tam,   %rdx  # tamaño
        syscall           # llamada a write
        ret
```

```
exit:   mov    $60,   %rax  # exit
        xor    %rdi, %rdi  # 0
        syscall           # llamada a exit
        ret
```

Ejercicios:

8. Compare hola64.s con hola64p.s. Sobre este podemos destacar: código de 64 bits, llamada a subrutina, uso de la pila y codificación de caracteres.

⊙ ¿Sabes C? \iff ¿Has usado la función printf()?

```
int main()
```

```
{
```

```
    int i = 0x12345678;
```

```
    printf("i = %i =  
    ↪ 0x%08x\n", i, i);
```

```
    return 0;
```

```
}
```

```
int i = 0x12345678;
```

```
char *formato = "i = %i =  
    ↪ 0x%08x\n";
```

```
int main()
```

```
{
```

```
    printf(formato, i, i);
```

```
    return 0;
```

```
}
```

Ejercicios:

9. ¿En qué se parecen y en qué se diferencian printf-c-1.c y printf-c-2.c? nm, objdump y kdiff3 serán muy útiles...

Mezclando lenguajes: ensamblador y C (32 bits) printf32.s

```
.data
i:      .int 12345      # variable entera
f:      .string "i = %d\n" # cadena de formato

.text
main:   push    i        # apila i
        push    $f        # apila f
        xor     %eax, %eax # n de registros vectoriales
        call    printf    # llamada a printf
        add     $8, %esp   # restaura pila

        movl    $1, %eax   # exit
        xorl    %ebx, %ebx  # 0
        int     $0x80      # llamada a exit
```

Ejercicios:

10. Descargue y compile printf32.s.
11. Modifique printf32.s para que finalice mediante la función exit() de C (man 3 exit). Solución: printf32e.s.

Mezclando lenguajes: ensamblador y C (64 bits) printf64.s

```
.data
i:      .int 12345          # variable entera
f:      .string "i = %d\n" # cadena de formato

.text
main:   mov  $f, %rdi        # formato
        mov  i, %rsi        # i
        xor  %rax, %rax     # registros vectoriales
        call printf         # llamada a función

        xor  %rdi, %rdi     # valor de retorno
        call exit           # llamada a función
```

Ejercicios:

12. Descargue y compile printf64.s.
13. Busque las diferencias entre printf32.s y printf64.s.

Optimización: sum.cc

```
int main()
{
    int sum = 0;

    for (int i = 0; i < 10; ++i)
        sum += i;

    return sum;
}
```

Ejercicios:

14. ¿Cómo implementa gcc los bucles **for**?
15. Observe el código de la función `main()` al compilarlo...
 - sin optimización: `g++ -O0 sum.cc -o sum`
 - con optimización: `g++ -O3 sum.cc -o sum`

Optimización: función main() de sum.cc


sin optimización (gcc -O0)

4005b6:	55	push	%rbp
4005b7:	48 89 e5	mov	%rsp,%rbp
4005ba:	c7 45 fc 00 00 00 00	movl	\$0x0,-0x4(%rbp)
4005c1:	c7 45 f8 00 00 00 00	movl	\$0x0,-0x8(%rbp)
4005c8:	eb 0a	jmp	4005d4 <main+0x1e>
4005ca:	8b 45 f8	mov	-0x8(%rbp),%eax
4005cd:	01 45 fc	add	%eax,-0x4(%rbp)
4005d0:	83 45 f8 01	addl	\$0x1,-0x8(%rbp)
4005d4:	83 7d f8 09	cmpl	\$0x9,-0x8(%rbp)
4005d8:	7e f0	jle	4005ca <main+0x14>
4005da:	8b 45 fc	mov	-0x4(%rbp),%eax
4005dd:	5d	pop	%rbp
4005de:	c3	retq	


con optimización (gcc -O3)

4004c0:	b8 2d 00 00 00	mov	\$0x2d,%eax
4004c5:	c3	retq	

Compiler Explorer: <https://godbolt.org/z/9bT7sb>



Add... More

Sponsors *PC-lint* 

Share Other Policies


C++ source #1

A B + - v 🔍 🚫

C++

```
1 template <class T>
2 concept bool Addable =
3     requires (T t) { t + t; };
4
5 int main()
6 {
7     int x = 1, y = 2;
8     Addable z = x + y;
9     return z;
10 }
```

x86-64 gcc 10.2 (Editor #1, Compiler #1) C++


x86-64 gcc 10.2  -fconcepts -std=c++17

A ⚙️ 🔍 📄 + - ✎

```
1 main:
2     pushq %rbp
3     movq %rsp, %rbp
4     movl $1, -4(%rbp)
5     movl $2, -8(%rbp)
6     movl -4(%rbp), %edx
7     movl -8(%rbp), %eax
8     addl %edx, %eax
9     movl %eax, -12(%rbp)
10    movl -12(%rbp), %eax
11    popq %rbp
12    ret
```

🔄 📄 Output (0/0) x86-64 gcc 10.2 i - 335ms (2918B)

x86-64 gcc 10.2 (Editor #1, Compiler #2) C++

x86-64 gcc 10.2  -fconcepts -std=c++17 -O3

A ⚙️ 🔍 📄 + - ✎

```
1 main:
2     movl $3, %eax
3     ret
```

🔄 📄 Output (0/0) x86-64 gcc 10.2 i - 631ms (2780B)

Compiler Explorer: <https://godbolt.org/z/ahhqs9>

The image shows the Compiler Explorer interface with two tabs. The left tab, titled 'C++ source #1', contains the following C++ code:

```
1 template<typename T> T adder(T v)
2 {
3     return v;
4 }
5
6 template<typename T, typename... Args>
7 T adder(T first, Args... args)
8 {
9     return first + adder(args...);
10 }
11
12 int main()
13 {
14     return adder(1, 2, 3, 4, 5);
15 }
```

The right tab, titled 'x86-64 gcc 10.2 (Editor #1, Compiler #1) C++', shows the assembly output for the first function:

```
50 int adder<int, int, int>(int, int, int):
51     pushq %rbp
52     movq %rsp, %rbp
53     subq $16, %rsp
54     movl %edi, -4(%rbp)
55     movl %esi, -8(%rbp)
56     movl %edx, -12(%rbp)
57     movl -12(%rbp), %edx
58     movl -8(%rbp), %eax
59     movl %edx, %esi
60     movl %eax, %edi
61     call _int_adder<int, int>(int, int)
62     movl -4(%rbp), %edx
63     addl %edx, %eax
64     leave
65     ret
66
67 int adder<int, int>(int, int):
68     pushq %rbp
69     movq %rsp, %rbp
70     subq $16, %rsp
71     movl %edi, -4(%rbp)
72     movl %esi, -8(%rbp)
73     movl -8(%rbp), %eax
74     movl %eax, %edi
75     call _int_adder<int>(int)
76     movl -4(%rbp), %edx
77     addl %edx, %eax
78     leave
79     ret
80
81 int adder<int>(int):
82     pushq %rbp
83     movq %rsp, %rbp
84     movl %edi, -4(%rbp)
85     movl -4(%rbp), %eax
86     popq %rbp
87     ret
```

The bottom tab, titled 'x86-64 gcc 10.2 (Editor #1, Compiler #2) C++', shows the assembly output for the main function:

```
1 main:
2     movl $15, %eax
3     ret
```

Compiler Explorer: <https://godbolt.org/z/1hWeWM>

C++ source #1 X

A- + - v

C++

```
1 int main()
2 {
3     int s = 0;
4     for (int i = 0; i < 1000000; ++i)
5         s += i;
6     return s;
7 }
```

x86-64 gcc 10.2 (Editor #1, Compiler #1) C++ X

x86-64 gcc 10.2

Compiler options..

A- - + -

```
1 main:
2     pushq %rbp
3     movq %rsp, %rbp
4     movl $0, -4(%rbp)
5     movl $0, -8(%rbp)
6     .L3:
7     cmpl $999999, -8(%rbp)
8     jg .L2
9     movl -8(%rbp), %eax
10    addl %eax, -4(%rbp)
11    addl $1, -8(%rbp)
12    jmp .L3
13    .L2:
14    movl -4(%rbp), %eax
15    popq %rbp
16    ret
```

x86-64 gcc 10.2 (Editor #1, Compiler #2) C++ X

x86-64 gcc 10.2

-march=knl -O3

A- - + -

```
1 main:
2     vmovdqa32 .LC0(%rip), %zmm0
3     xorl %eax, %eax
4     vpxor %xmm1, %xmm1, %xmm1
5     vmovdqa32 .LC1(%rip), %zmm3
6     .L2:
7     addl $1, %eax
8     vmovdqa32 %zmm0, %zmm2
9     cmpl $62500, %eax
10    vpaddq %zmm3, %zmm0, %zmm0
11    vpaddq %zmm2, %zmm1, %zmm1
12    jne .L2
13    vmovdqa %ymm1, %ymm0
14    vextracti64x4 $0x1, %zmm1, %ymm1
15    vpaddq %ymm1, %ymm0, %ymm1
16    vmovdqa %xmm1, %xmm0
17    vextracti128 $0x1, %ymm1, %xmm1
18    vpaddq %xmm1, %xmm0, %xmm0
19    vpsrlq $8, %xmm0, %xmm1
20    vpaddq %xmm1, %xmm0, %xmm0
21    vpsrlq $4, %xmm0, %xmm1
22    vpaddq %xmm1, %xmm0, %xmm0
23    vmovd %xmm0, %eax
24    ret
25    .LC0:
26    .long 0
```

Cutter: <https://github.com/rizinorg/cutter>

The screenshot displays the Cutter application window titled "Cutter - sumg". The interface includes a menu bar (Archivo, Edit, Ver, Ventanas, Debug, Ayuda), a toolbar, and a search bar. The left sidebar shows the "Functions" list with "main" selected. Below it, the "main" function's metadata is displayed: Offset: 0x00401106, Tamaño: 0x19, Importar: falso, Nargs: 0x0, Nbbs: 0x4, Nlocals: 0x0, Tipo de llamada: amd64, Bordas: 4, StackFrame: 0, and Comment: . The "Quick Filter" and "Graph Overview" sections are also visible. The main pane shows the assembly code for the "main" function, with the following instructions highlighted in a blue box:

```
[0x00401106]
int main (int argc, char **argv, char **envp);
b800000000    movl    $0, %eax
ba00000000    movl    $0, %edx
eb05          jmp     0x401117
```

The control flow graph (CFG) is shown below the assembly code, with nodes representing basic blocks and arrows indicating the flow of execution. The nodes are:

- Block 1 (0x00401106):

```
[0x00401106]
83f809    cmpl    $9, %eax    ; 9
7ef6      jle     0x401112
```
- Block 2 (0x0040111c):

```
[0x0040111c]
89d0      movl    %edx, %eax
c3        retq
```
- Block 3 (0x00401112):

```
[0x00401112]
01c2      addl    %eax, %edx
83c001    addl    $1, %eax
```

The flow starts at Block 1, which branches to Block 2 if the condition is less than or equal (jle) and to Block 3 otherwise. Block 2 then returns (retq), and Block 3 increments the register %edx by 1 before looping back to Block 1.

The bottom status bar shows various tabs: Dashboard, Strin..., Imports, Typ..., Search, Disassembly, Gráfico (mai..., Hexdump, Decompiler (ma..., Heap, Signatur..., and Yara.

Enlaces de interés

Manuales:

- ⦿ Hardware:

- AMD
- Intel

- ⦿ Software:

- AS
- NASM

Programación:

- ⦿ Programming from the ground up
- ⦿ Linux Assembly

Chuletas:

- ⦿ Chuleta del 8086
- ⦿ Chuleta del GDB