

Práctica 1: Entorno de desarrollo GNU

Estructura de Computadores

Gustavo Romero López

Updated: 29 de septiembre de 2020

Arquitectura y Tecnología de Computadores

Índice

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

Objetivos

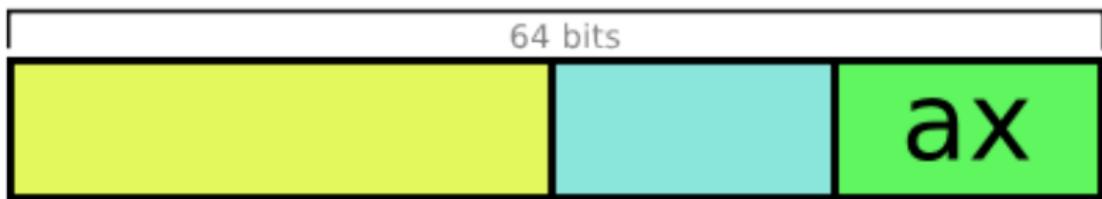
- Nociones de ensamblador 80x86 de 64 bits.
- Linux es tu amigo: si no sabes algo pregunta... **man**.
- Hoy aprenderemos varias cosas:
 - El esqueleto de un programa básico en ensamblador.
 - Como aprender de un maestro: el compilador **gcc**.
 - Herramientas clásicas del entorno de programación UNIX:
 - **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.
 - **gdb** y **ddd** (gdb con cirugía estética): los depuradores.
 - Herramienta web: Compiler Explorer

Ensamblador 80x86

- ④ Los **80x86** son una familia de procesadores.
- ④ El más utilizado junto a los procesadores **ARM**.
- ④ En estas prácticas vamos a centrarnos en su **lenguaje ensamblador** (inglés).
- ④ El lenguaje ensamblador es el más básico, tras el binario, con el que podemos escribir programas utilizando las **instrucciones** que entiende el procesador.
- ④ Cualquier estructura de un lenguajes de alto nivel pueden crearse mediante instrucciones muy sencillas.
- ④ 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

rax



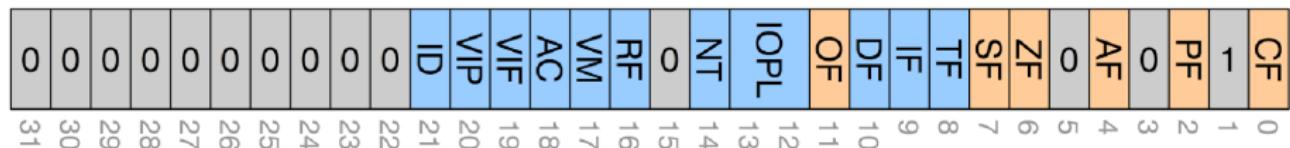
eax

Arquitectura 80x86: registros completos

ZMM0	YMM0	XMM0	ZMM1	YMM1	XMM1	ST(0)	MM0	ST(1)	MM1	ALAH	AXEAX	RAX	RBW	R8D	R8	R12W	R12D	R12	CR0	CR4	
ZMM2	YMM2	XMM2	ZMM3	YMM3	XMM3	ST(2)	MM2	ST(3)	MM3	BLBH	BXBX	RBX	R9W	R9D	R9	R13W	R13D	R13	CR1	CR5	
ZMM4	YMM4	XMM4	ZMM5	YMM5	XMM5	ST(4)	MM4	ST(5)	MM5	CLCH	CXECX	RCX	R10W	R10D	R10	R14W	R14D	R14	CR2	CR6	
ZMM6	YMM6	XMM6	ZMM7	YMM7	XMM7	ST(6)	MM6	ST(7)	MM7	DIBD	DXEDX	RDX	R11W	R11D	R11	R15W	R15D	R15	CR3	CR7	
ZMM8	YMM8	XMM8	ZMM9	YMM9	XMM9					BPL	BPEB	RBP	DIL	DI	EDI	RDI			IP	EIP	RIP
ZMM10	YMM10	XMM10	ZMM11	YMM11	XMM11	CW	FP_IP	FP_DP	FP_CS	SL	SI	ESI	RSI	SPN	SP	ESP	RSP		MSW	CR9	
ZMM12	YMM12	XMM12	ZMM13	YMM13	XMM13	SW													CR10		
ZMM14	YMM14	XMM14	ZMM15	YMM15	XMM15	TW													CR11		
ZMM16	ZMM17	ZMM18	ZMM19	ZMM20	ZMM21	ZMM22	ZMM23	FP_DS											CR12		
ZMM24	ZMM25	ZMM26	ZMM27	ZMM28	ZMM29	ZMM30	ZMM31	FP_OPC	FP_DP	FP_IP	CS	SS	DS	GDTR	IDTR		DRO	DR6	CR13		
											ES	FS	GS	TR	LDT		DR1	DR7	CR14		
														FLAGS	EFLAGS	RFLAGS	DR2	DR8	CR15		
																	DR3	DR9	MXCSR		
																	DR4	DR10	DR12	DR14	
																	DR5	DR11	DR13	DR15	

Arquitectura 80x86: banderas

eflags register



■ Reserved flags

■ System flags

■ Arithmetic flags

TF: Trap
IF: Interrupt
DF: Direction

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

Arquitectura 80x86: paso de parámetros a funciones

%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

Programa mínimo en C... todos ellos equivalentes

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

- ⑤ Compilar: gcc minimo1.c -o minimo1
 - ⑤ ¿Qué he hecho? file ./minimo1
 - ⑤ ¿Qué contiene? nm ./minimo1
 - ⑤ Ejecutar: ./minimo1
 - ⑤ Desensamblar: objdump -d minimo1
 - ⑤ Ver llamadas al sistema: strace ./minimo1
 - ⑤ Ver llamadas de biblioteca: ltrace ./minimo1
 - ⑤ ¿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

```
1 .data  
2  
3 .text
```

Ensamblador desde 0: punto de entrada

```
1 .text  
2     .global _start
```

Ensamblador desde 0: datos

```
1 .data
2 msg:      .string "¡hola, mundo!\n"
3 tam:      .quad . - msg
```

Ensamblador desde 0: código

```
1 write:    mov    $1,    %rax    # write
2             mov    $1,    %rdi    # stdout
3             mov    $msg,   %rsi    # texto
4             mov    tam,   %rdx    # tamaño
5             syscall           # llamada a write
6             ret
7
8 exit:     mov    $60,   %rax    # exit
9             xor    %rdi,   %rdi    # 0
10            syscall          # llamada a exit
11            ret
```

Ensamblador desde 0: ejemplo básico hola.s

```
1 .data
2 msg:     .string "¡hola, mundo!\n"
3 tam:     .quad . - msg
4
5 .text
6         .global _start
7
8 write:   mov    $1,    %rax  # write
9         mov    $1,    %rdi  # stdout
10        mov    $msg,   %rsi  # texto
11        mov    tam,   %rdx  # tamaño
12        syscall          # llamada a write
13        ret
14
15 exit:    mov    $60,   %rax  # exit
16        xor    %rdi,   %rdi  # 0
17        syscall          # llamada a exit
18        ret
19
20 _start:  call   write      # llamada a función
21         call   exit       # llamada a función
22
```

¿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 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`.

makefile

<http://pccito.ugr.es/~gustavo/ec/practicas/1/makefile>

```
ASM = $(wildcard *.s)
SRC = $(wildcard *.c *.cc)
EXE = $(basename $(ASM) $(SRC))
ATT = $(EXE:=.att)
```

```
CFLAGS = -g -std=c11 -Wall
CXXFLAGS = $(CFLAGS:c11=c++11)
```

```
all: $(EXE) $(ATT)
```

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

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

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "¡hola, mundo!" << std::endl;
6 }
```

- ◎ ¿Qué hace gcc 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:

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

```
1  write:  movl  $4, %eax      # write
2          movl  $1, %ebx      # salida estándar
3          movl  $msg, %ecx     # cadena
4          movl  tam, %edx      # longitud
5          int   $0x80      # llamada a write
6          ret               # retorno
7
8  exit:   movl  $1, %eax      # exit
9          xorl  %ebx, %ebx    # 0
10         int   $0x80      # llamada a exit
11
```

Ejercicios:

6. Descargue hola32.s. Ejecute el programa instrucción por instrucción con el ddd hasta comprender como funciona.
7. Si quiere aprender un poco más estudie hola32p.s. Sobre el mismo podemos destacar: código de 32 bits, uso de “*little endian*”, llamada a subrutina, uso de la pila y codificación de caracteres.

Depuración: hola64.s

ejemplo de 64 bits

```
1  write:   mov    $1,    %rax  # write
2          mov    $1,    %rdi  # stdout
3          mov    $msg,  %rsi  # texto
4          mov    tam,   %rdx  # tamaño
5          syscall           # llamada a write
6          ret
7
8  exit:    mov    $60,   %rax  # exit
9          xor    %rdi,  %rdi  # 0
10         syscall          # llamada a exit
11         ret
```

Ejercicios:

8. Descargue hola64.s. Ejecute el programa instrucción por instrucción con el ddd hasta comprender como funciona.
9. 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.

¿Dónde están mis datos?

printf-c-1.c y printf-c-2.c

- ④ ¿Sabes C? \iff ¿Has usado la función printf()?

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int i = 12345;
6     printf("i=%d\n", i);
7     return 0;
8 }
```

```
1 #include <stdio.h>
2
3 int i = 12345;
4 char *formato = "i=%d\n";
5
6 int main()
7 {
8     printf(formato, i);
9     return 0;
10 }
```

Ejercicios:

10. ¿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

```
1 .data
2 i:      .int 12345          # variable entera
3 f:      .string "i = %d\n" # cadena de formato
4
5 .text
6         .extern printf      # printf en otro sitio
7         .globl _start        # función principal
8
9 _start: push (i)            # apila i
10        push $f              # apila f
11        mov  $0, %eax         # n de registros vectoriales
12        call printf          # llamada a printf
13        add  $8, %esp          # restaura pila
14
15        movl $1, %eax         # exit
16        xorl %ebx, %ebx       # 0
17        int   $0x80          # llamada a exit
```

Ejercicios:

11. Descargue y compile printf32.s.
12. 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

```
1 .data
2 i:      .int 12345          # variable entera
3 f:      .string "i = %d\n" # cadena de formato
4
5 .text
6     .globl _start
7
8 _start: mov $f, %rdi        # formato
9         mov (i), %rsi        # i
10        xor %rax, %rax      # n de registros vectoriales
11        call printf         # llamada a función
12
13        xor %rdi, %rdi      # valor de retorno
14        call exit           # llamada a función
```

Ejercicios:

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

Optimización: sum.cc

```
1 int main()
2 {
3     int sum = 0;
4
5     for (int i = 0; i < 10; ++i)
6         sum += i;
7
8     return sum;
9 }
```

Ejercicios:

15. ¿Cómo implementa gcc los bucles **for**?
16. 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>

The screenshot shows the Compiler Explorer interface with two main compiler panes.

Left Compiler Pane: Compiles C++ source code using gcc 10.2. The code defines a concept `Addable` and uses it in a function `main`.

```
template <class T>
concept bool Addable =
    requires (T t) { t + t; };

int main()
{
    int x = 1, y = 2;
    Addable z = x + y;
    return z;
}
```

Right Compiler Pane: Compiles the same C++ code using gcc 10.2 with the flags `-fconcepts -std=c++17`. The assembly output shows the implementation of the `Addable` concept via move instructions.

```
main:
    pushq %rbp
    movq %rsp, %rbp
    movl $1, -4(%rbp)
    movl $2, -8(%rbp)
    movl -4(%rbp), %edx
    movl -8(%rbp), %eax
    addl %edx, %eax
    movl %eax, -12(%rbp)
    movl -12(%rbp), %eax
    popq %rbp
    ret
```

Output: Both compilers produce 0 errors and 0 warnings, with execution times of 335ms and 631ms respectively.

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

Compiler Explorer interface showing C++ source code and assembly output for x86-64 gcc 10.2.

Top Left: C++ source code #1 (Editor #1, Compiler #1) C++ X

```
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 }
```

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

```
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
```

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

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

Bottom Right: Compiler options...

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

The image shows the Compiler Explorer interface with three main panes:

- C++ source #1:** Contains the C++ code for a simple summation program.
- x86-64 gcc 10.2 (Editor #1, Compiler #1) C++:** Shows the generated assembly code for the main function, highlighting memory operations and control flow.
- x86-64 gcc 10.2 (Editor #1, Compiler #2) C++:** Shows the generated assembly code for the main function, highlighting SIMD operations.

Compiler Options: -march=knl -O3

```
int main()
{
    int s = 0;
    for (int i = 0; i < 1000000; ++i)
        s += i;
    return s;
}
```

```
main:
    pushq %rbp
    movq %rsp, %rbp
    movl $0, -4(%rbp)
    movl $0, -8(%rbp)
.L3:
    cmpl $999999, -8(%rbp)
    jg .L2
    movl -8(%rbp), %eax
    addl %eax, -4(%rbp)
    addl $1, -8(%rbp)
    jmp .L3
.L2:
    movl -4(%rbp), %eax
    popq %rbp
    ret
```

```
main:
    vmovdqa32 .LC0(%rip), %zmm0
    xorl %eax, %eax
    vpxor %xmm1, %xmm1, %xmm1
    vmovdqa32 .LC1(%rip), %zmm3
.L2:
    addl $1, %eax
    vmovdqa32 %zmm0, %zmm2
    cmpl $62500, %eax
    vpadd %zmm3, %zmm0, %zmm0
    vpadd %zmm2, %zmm1, %zmm1
    jne .L2
    vmovdqa %ymm1, %ymm0
    vextracti64x4 $0x1, %zmm1, %ymm1
    vpadd %ymm1, %ymm0, %ymm1
    vmovdqa %xmm1, %xmm0
    vextracti128 $0x1, %ymm1, %xmm1
    vpadd %xmm1, %xmm0, %xmm0
    vpsrlqd $8, %xmm0, %xmm1
    vpadd %xmm1, %xmm0, %xmm0
    vpsrlqd $4, %xmm0, %xmm1
    vpadd %xmm1, %xmm0, %xmm0
    vmovd %xmm0, %eax
    ret
.LC0:
    .long 0
```

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