

Práctica 1: Entorno de desarrollo GNU

Estructura de Computadores

Gustavo Romero López

Updated: 29 de septiembre de 2020

Arquitectura y Tecnología de Computadores

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

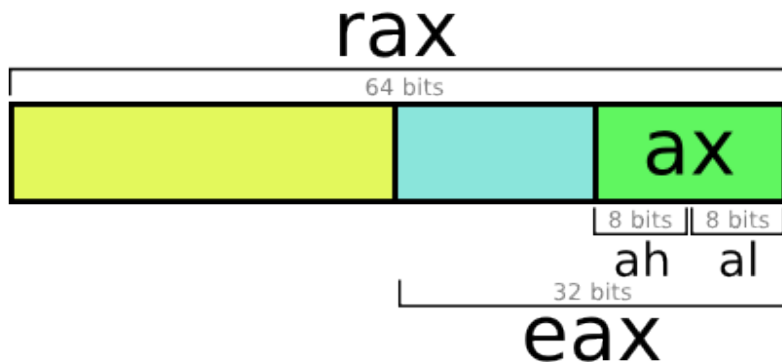
Objetivos

- ⦿ 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.
 - **gdb** y **ddd** (gdb con cirugía estética): los depuradores.
 - Herramienta web: Compiler Explorer

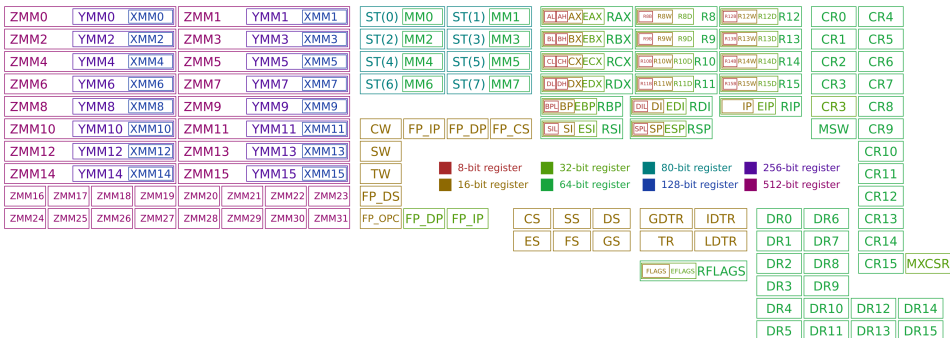
Ensamblador 80x86

- ⊙ 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 completos



Arquitectura 80x86: banderas

eflags register

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
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

 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

- ⊙ Compilar: `gcc minimo1.c -o minimo1`
- ⊙ ¿Qué he hecho? `file ./minimo1`
- ⊙ ¿Qué contiene? `nm ./minimo1`
- ⊙ Ejecutar: `./minimo1`
- ⊙ 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

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`, Pruébalo e intente hacer alguna modificación.
4. Examine el código ensamblador con `objdump -d hola`.


```
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.

```
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.

⊙ ¿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>



Add... More

Sponsors *PC-lint* 

Share Other Policies

C++ source #1 X

A- + - 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++ X

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 ⓘ - 335ms (2918B)

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

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 ⓘ - 631ms (2780B)

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

C++ source #1

A- B +> v

C++

```
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 }
```

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

x86-64 gcc 10.2

Compiler options...

A- G- T- B- +> v

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

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

x86-64 gcc 10.2

Compiler options...

A- G- T- B- +> v

```
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
```

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

```
C++ source #1 X
A- B + v
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- G Y B + /
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- G Y B + /
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
```

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