

Práctica 1: Entorno de desarrollo GNU

Estructura de Computadores

Gustavo Romero López

Actualizado: 19 de septiembre de 2018

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

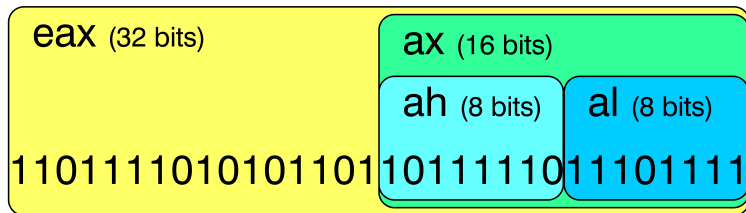
Objetivos

- ⊙ Programar en ensamblador.
- ⊙ 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

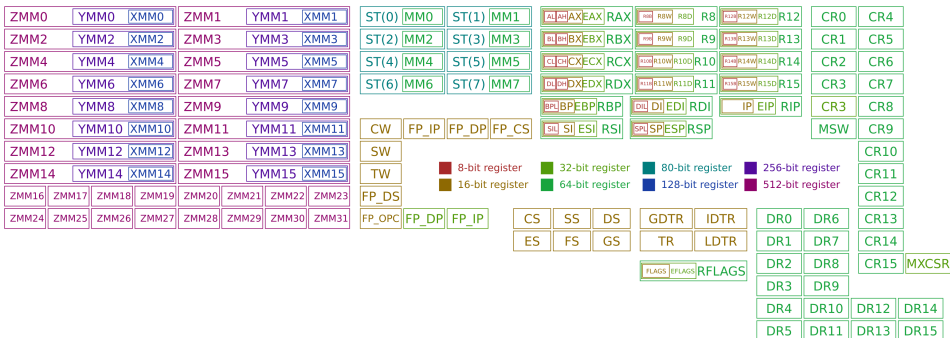
Ensamblador 80x86

- ⊙ Los **80x86** son una familia de procesadores.
- ⊙ Junto con los procesadores tipo **ARM** son los más utilizados.
- ⊙ 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 lenguaje de alto nivel pueden crearse mediante instrucciones sencillas.
- ⊙ Normalmente es utilizado para poder acceder a partes que los lenguajes de alto nivel nos ocultan, complican o hacen de forma que no nos interesa.

Register aliasing / sub-registers




Arquitectura 80x86: registros completos



Arquitectura 80x86: banderas

eflags register

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

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`
- ⊙ 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 (0x00007ffe2dddbc000)
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                                # datos
2
3  .text                                # código
```

Ensamblador desde 0: punto de entrada

```
1  .text                                # código
2      .globl _start                    # empezar aquí
```

Ensamblador desde 0: datos

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

Ensamblador desde 0: código

```
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
```

Ensamblador desde 0: ejemplo básico hola.s

```
1  .data                                # datos
2  msg:      .string "ihola, mundo!\n"
3  tam:      .int . - msg
4
5  .text                                # código
6          .globl _start                # empezar aquí
7
8  write:    movl    $4, %eax            # write
9           movl    $1, %ebx            # salida estándar
10          movl    $msg, %ecx           # cadena
11          movl    tam, %edx            # longitud
12          int     $0x80                # llamada a write
13          ret                          # retorno
14
15  exit:     movl    $1, %eax            # exit
16          xorl    %ebx, %ebx           # 0
17          int     $0x80                # llamada a exit
18
19  _start:
20          call    write                # llamada a función
21          call    exit                # llamada a función
```

¿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`.

```
SRC = $(wildcard *.c *.cc)
```

```
CFLAGS = -g -std=c11 -Wall
```

```
CXXFLAGS = $(CFLAGS:c11=c++11)
```

```
sums: CXXFLAGS+=-Os
```

```
#
```

```
#####
```

```
%.o
```

```
$(LD) $(LDFLAGS) $< -o $@
```

```
%.s
```

```
$(CC) $(CFLAGS) -nostartfiles $< -o $@
```

```
%.c
```

```
$(CC) $(CFLAGS) $< -o $@
```


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

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "¡hola, mundo!"
6                  << std::endl;
7  }
```

- ⦿ ¿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
```

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         call printf           # llamada a printf
12         add $8, %esp          # restaura pila
13
14         movl $1, %eax         # exit
15         xorl %ebx, %ebx       # 0
16         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      # null
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	

Pruebe addable.cc en <https://godbolt.org>

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

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

[Editor](#)[Diff View](#)[More ▾](#)[Support Compiler Explorer on Patreon! ✖](#)[Share ▾](#)[Other ▾](#)[Policies ▾](#)

C++ source #1 ✕

A ▾ Save/Load + Add new... ▾ C++ ▾

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

x86-64 gcc 8.2 (Editor #1, Compiler #1) C++ ✕

x86-64 gcc 8.2 -lconcepts

A ▾ 11010 LX0: .text // 's+ Intel Demangle

Libraries ▾ + Add new... ▾

```
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
```

x86-64 gcc 8.2 (Editor #1, Compiler #2) C++ ✕

x86-64 gcc 8.2 -lconcepts -O3

A ▾ 11010 LX0: .text // 's+ Intel Demangle

Libraries ▾ + Add new... ▾

```
1 main:
2     mov     eax, 3
3     ret
```

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