

# Práctica 1: Entorno de desarrollo GNU

Gustavo Romero López

Arquitectura y Tecnología de Computadores

29 de septiembre de 2015

# Índice

- 1 Índice
- 2 Objetivos
- 3 Introducción
- 4 Esqueleto
- 5 Ejemplos
  - hola
  - make
  - C++
  - 32 bits
  - 64 bits
  - asm + C
  - Optimización
- 6 Enlaces

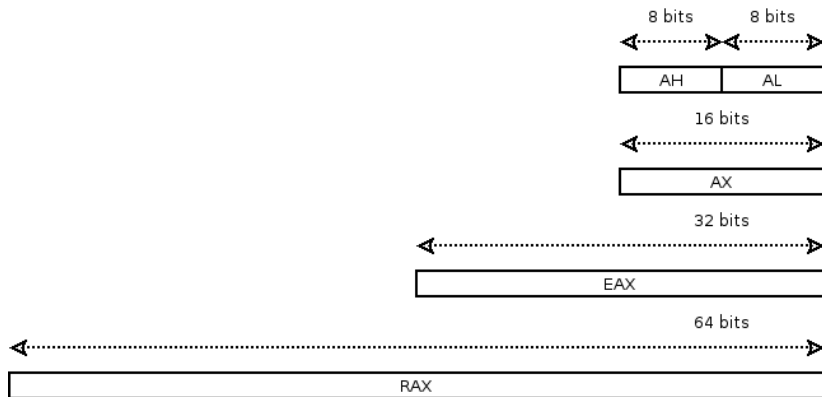
# Objetivos

- Programar en ensamblador.
- Linux es tu amigo: si no sabes algo pregunta (**man**).
- Hoy estudiaremos varias cosas:
  - Esqueleto de un programa básico en ensamblador.
  - Como aprender de un maestro: **gcc**.
  - Herramientas del entorno de programación:
    - **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.

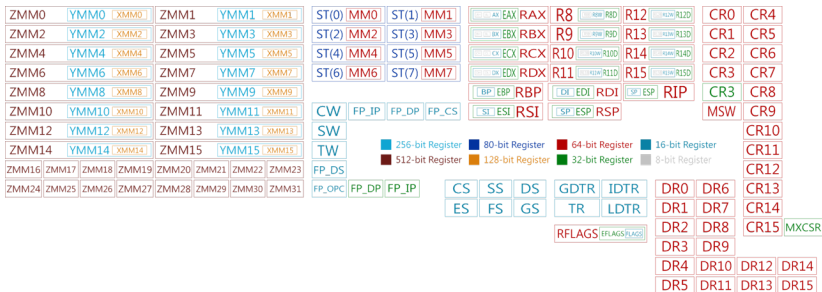
# Ensamblador 80x86

- 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 conseguirse mediante instrucciones sencillas.
- Normalmente es utilizado para poder acceder partes que los lenguajes de alto nivel nos ocultan o hacen de forma que no nos interesa.

# Arquitectura 80x86: registros



# Arquitectura 80x86: registros completos



# Arquitectura 80x86: banderas

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

X ID Flag (ID)

X Virtual Interrupt Pending (VIP)

X Virtual Interrupt Flag (VIF)

X Alignment Check (AC)

X Virtual-8086 Mode (VM)

X Resume Flag (RF)

X Nested Task (NT)

X I/O Privilege Level (IOPL)

S Overflow Flag (OF)

C Direction Flag (DF)

X Interrupt Enable Flag (IF)

X Trap Flag (TF)

S Sign Flag (SF)

S Zero Flag (ZF)

S Auxiliary Carry Flag (AF)

S Parity Flag (PF)

S Carry Flag (CF)

# Ensamblador desde 0: secciones de un programa

```
1  .data                                # sección de datos
2
3  .text                                # sección de código
```



# Ensamblador desde 0: punto de entrada

```
5  .text                                # sección de código
6      .globl _start                    # punto de entrada
7
8  _start:                              # etiqueta de entrada
```

# Ensamblador desde 0: variables

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

# Ensamblador desde 0: código

```
5  .text                                # sección de código
6      .globl _start                    # punto de entrada
7
8  _start:                              # etiqueta de entrada
9      movl    $4, %eax                 # write
10     movl    $1, %ebx                 # salida estándar
11     movl    $msg, %ecx               # cadena
12     movl    tam, %edx                # longitud
13     int     $0x80                    # llamada al sistema
14
15     movl    $1, %eax                 # exit
16     xorl    %ebx, %ebx               # 0
17     int     $0x80                    # llamada al sistema
```

# Ensamblador desde 0: ejemplo básico hola.s

```
1  .data                                # sección de datos
2      msg: .string "hola, mundo!\n"
3      tam: .int  . - msg
4
5  .text                                # sección de código
6      .globl _start                    # punto de entrada
7
8  _start:                              # etiqueta de entrada
9      movl    $4, %eax                 # write
10     movl    $1, %ebx                 # salida estándar
11     movl    $msg, %ecx               # cadena
12     movl    tam, %edx                # longitud
13     int     $0x80                    # llamada al sistema
14
15     movl    $1, %eax                 # exit
16     xorl    %ebx, %ebx               # 0
17     int     $0x80                    # llamada al sistema
```

# ¿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 y objetivos.

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` y modifique la forma de compilación de los ficheros ensamblador.
- 4 Examine el código ejecutable con `objdump -C -D hola`

# Makefile

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

```
3 SRC = $(wildcard *.cc)
4 ESP = $(wildcard cout.s printf*.s)
5 ASM = $(filter-out $(ESP), $(wildcard *.s))
6 OBJ = $(ASM:.s=.o)
7 EXE = $(basename $(ASM) $(ESP) $(SRC))
8
9 %.o: %.s
10     $(AS) $(ASFLAGS) $< -o $@
11
12 %: %.o
13     $(LD) $(LDFLAGS) $< -o $@
14
15 %: %.s
16     $(CC) $(CFLAGS) $< -o $@
```

# Ejemplo en C++: hola2.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 hola2`
  - Sintaxis Intel: `objdump -C -D hola2 -M intel`

## Ejercicios:

- 5 Muestre el código de la función `main()`.

# Depuración: hola32.s

ejemplo de 32 bits

```
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 al sistema
13         ret                     # retorno a _start
14
15  exit:   movl    $1, %eax    # exit
16         xorl    %ebx, %ebx   # 0
17         int     $0x80        # llamada al sistema
18
19  _start: call    write        # llamada a función
20         call    exit          # llamada a función
```

## 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

```

 8  write:  mov    $1, %rax    # write
 9          mov    $1, %rdi    # stdout
10          mov    $msg, %rsi  # texto
11          mov    tam, %rdx   # tamaño
12          syscall           # llamada al sistema
13          ret               # retorno
14
15  exit:   mov    $60, %rax    # exit
16          xor    %rdi, %rdi   # 0
17          syscall           # llamada al sistema
18
19  _start: call    write       # llamada a función
20          call    exit        # llamada a función

```

## Ejercicios:

- 8 Descargue hola64.s. Ejecute el programa instrucción por instrucción con el ddd hasta comprender como funciona.
- 9 Si se siente con fuerzas échele un vistazo al ejemplo: hola64p.s. Sobre este podemos destacar: código de 64 bits, llamada a subrutina, uso de la pila y codificación de caracteres.

# Mezclando lenguajes: ensamblador y C $\longrightarrow$ printf.s

```
6  .data
7  i:      .int 12345          # variable entera
8  f:      .string "i = %d\n" # cadena de formato
9
10 .text
11 main:   pushl %ebp          # preserva ebp
12         movl %esp, %ebp     # copia pila
13
14         pushl (i)           # apila i
15         pushl $f            # apila f
16         call  printf        # llamada a función
17         addl  $8, %esp      # restaura pila
```

## Ejercicios:

- 10 Descargue y compile **printf.s**.
- 11 Modifique **printf.s** para que finalice mediante la función `exit()` de C (man 3 `exit`). Solución: **printf2.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:

- 12 ¿Cómo implementa gcc los bucles for?
- 13 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: sum.cc

## sin optimización (gcc -O0)

```

1      4005b6: 55                push    %rbp
2      4005b7: 48 89 e5         mov     %rsp,%rbp
3      4005ba: c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%rbp)
4      4005c1: c7 45 f8 00 00 00 00 movl    $0x0,-0x8(%rbp)
5      4005c8: eb 0a           jmp     4005d4 <main+0x1e>
6      4005ca: 8b 45 f8         mov     -0x8(%rbp),%eax
7      4005cd: 01 45 fc         add     %eax,-0x4(%rbp)
8      4005d0: 83 45 f8 01      addl    $0x1,-0x8(%rbp)
9      4005d4: 83 7d f8 09      cmpl    $0x9,-0x8(%rbp)
10     4005d8: 7e f0           jle     4005ca <main+0x14>
11     4005da: 8b 45 fc         mov     -0x4(%rbp),%eax
12     4005dd: 5d             pop     %rbp
13     4005de: c3             retq

```

## con optimización (gcc -O3)

```

1      4004c0: b8 2d 00 00 00   mov     $0x2d,%eax
2      4004c5: c3             retq

```

# 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