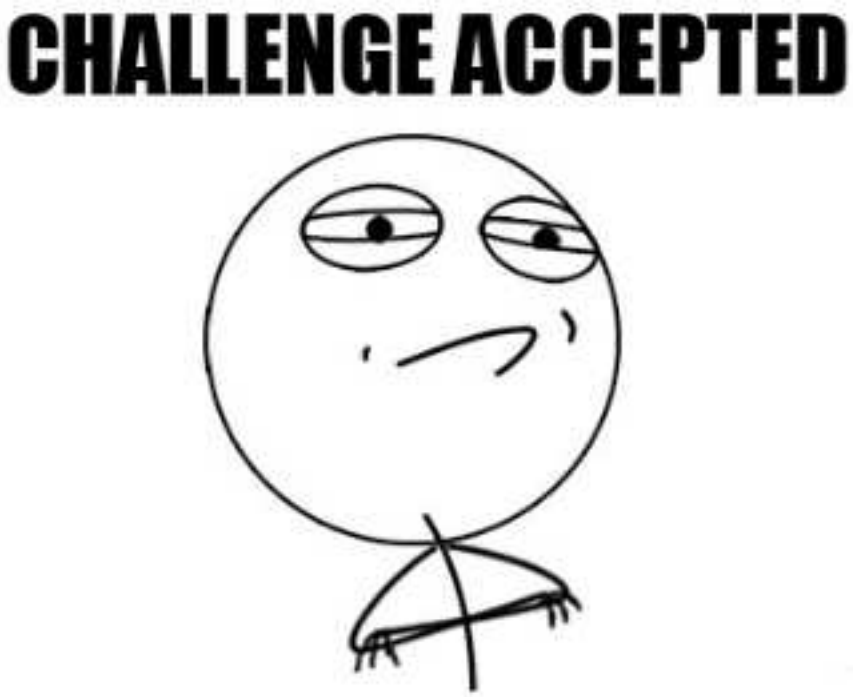
EL ENSAMBLADOR...
PERO SI ES MUY FÁCIL

Guia supuestamente poco dolorosa
a la programación del IA-32 (i386)
(sintaxis AT&T)

Manel Guerrero Zapata

21 de octubre de 2013



Índice general

Índice de códigos fuente

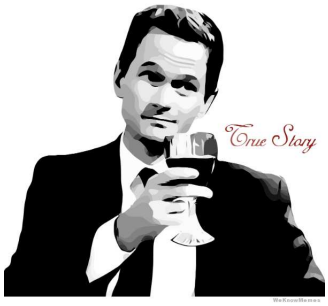
Capítulo 1

Prólogo

En setiembre del 2012 di por primera vez una asignatura que consiste en enseñar a programar en C en ocho semanas y en ensamblador del 386 en seis semanas a estudiantes de primero de telecomunicaciones. El único problema era que, si bien soy un buen programador en C, no tenía ni remota idea de ensamblador.
-Pero bueno -me dije- tampoco puede ser tan complicado, ¿no?. **lol**
-Y seguro que hay un buen libro en inglés en el que me puedo basar. **lol lol**

Al final todo fue muy bien en un cuatrimestre super excitante a la vez que agotador. Y, con las transpas y apuntes de otros profes, Google y un par de libros en inglés que no me terminaron de convencer, aprendí ensamblador y empecé a enseñarlo. X-D

Este libro es el libro que me hubiera gustado tener entonces, y va dedicado a los alumnos a los que tuve el placer de enseñar ensamblador por primera vez: los grupos 70 y 90 del otoño del 2012.



Capítulo 2

Introducción

2.1. Iconos y cajas

A lo largo del libro utilizaremos los siguientes iconos y cajas:

TEORÍA

- En estas cajas pondremos los conceptos teóricos.

EL CÓDIGO

- En estas cajas explicaremos las lineas de código más importantes.

TRAMPAS

- En estas cajas hablaremos de los errores y descuidos típicos que se cometen al programar en ensamblador.

TRUCOS

- En estas cajas daremos trucos varios.

EL BOXEO

- Million Dollar Baby (2004):

  "El boxeo es un acto antinatural porque todo va al revés. Si quieres desplazarte hacia la izquierda, no das un paso a la izquierda, cargas sobre el pie derecho. Para desplazarte hacia la derecha usas el pie izquierdo. En vez de huir del dolor como haría una persona cuerda, das un paso hacia él. En el boxeo todo va al revés". - Eddie 'Scrap-Iron' Dupris (interpretado por Morgan Freeman)

- Al igual que en el boxeo, cuando empecemos a programar en ensamblador puede que también nos parezca que todo va al revés. Y que es imposible de aprender. Pero al final de este libro ya todo nos parecerá normal.
- En estas cajas utilizaremos la percepción de que "el ensamblador es como el boxeo porque todo va al revés" como mnemotécnico.

La terminal (o consola) la mostraremos de esta manera:

```
$ ls
foo       foo.s
$
```

El '$ ' no lo escribes tú. Se llama "prompt" en inglés y aparece sólo. En este ejemplo, el usuario teclea el comando "ls" que hace que el sistema operativo

muestre por pantalla los ficheros existentes en el directorio actual (foo y foo.s). Si no sabes lo que es una consola, consulta:
http://valentux.wordpress.com/2010/01/26/la-consola-o-terminal-en-gnulinux-lo-basico/

2.2. Compilando y depurando

Sistema operativo: Lo mejor es tener instalada una distribución Linux (como por ejemplo Ubuntu) de 32 bits. Si tenéis una de 64 bits puede que os dé algún problema. Todo este libro asume que tenéis un Linux instalado.

Compilar: Los ficheros con código fuente en ensamblador tienen extensión '.s'. Para compilar el codigo 'foo.s' y generar el ejecutable 'foo':

```
$ gcc -g -o foo foo.s
$
```

El '-g' es para que el ejecutable contenga la información necesaria para ser depurado. El '-o' indica que a continuación viene el nombre que tendrá el ejecutable.

Depende de como tengas configurado tu ordenador, es posible que los errores de compilación te salgan en castellano. Muchas veces la traducción es malísima y no se entiende. Para hacer que los errores salgan en inglés asigna 'C' a la variable de entorno 'LANG' tal y como ves en el ejempo:

```
$ gcc -g -o t6_vectors t6_vectors.s
t6_vectors.s: Assembleriviestit:
t6_vectors.s:9: Virhe: "vw(,%cx,2)" ei ole voimassa oleva
 "base/index"-lauseke
$ LANG=C gcc -g -o t6_vectors t6_vectors.s
t6_vectors.s: Assembler messages:
t6_vectors.s:9: Error: 'vw(,%cx,2)' is not a valid
 base/index expression
$
```

En este ejemplo, en mi ordenador los errores de compilación salen en finlandés, pero puedo forzar que salgan en inglés escribiendo 'LANG=C ' antes de 'gcc'.
Si, por defecto los mensajes de error no os salen en castellano, y tenéis instalados los paquetes necesarios podéis ver los mensajes de error en castellano con 'LANG=es_ES.UTF-8'. En algunos ordenadores no funcionará, y tendréis que escribir 'LANGUAGE=es_ES.UTF-8'
Si todo falla y vuestra distribución Linux es una Ubuntu u otra basada en Debian podéis provar lo siguiente:
Primero le decimos que si los paquetes necesarios no están instalados, nos los instale y que reconfigure el idioma:

```
$ sudo apt-get update
$ sudo apt-get install language-pack-es manpages-es
 manpages-es-extra
$ sudo locale-gen es_ES.UTF-8
$ sudo dpkg-reconfigure locales
$
```

Y, a partir de entonces, deberiamos ver los errores de compilacion en castellano con esta invocació:

```
$ LANG=es_ES.UTF-8 LANGUAGE=es_ES.UTF-8 gcc -g
 -o t6_vectors t6_vectors.s
t6_vectors.s: Mensajes del ensamblador:
t6_vectors.s:9: Error: "vw(,%cx,2)" no es una expresion
 base/index valida
$
```

Depurar: Yo para depurar (o debugar) recomiendo el ddd. Si no sabes que es depurar un código o quieres más información sobre el ddd que la que doy aquí consulta la siguiente página web:
http://www.linuxfocus.org/Castellano/January1998/article20.html
Existen alternativas al ddd como Netbeans y Eclipse. Pero, en mi opinión, tienen demasiadas opciones y son auténticos monstruos que van muy lentos.
Para instalar ddd en Ubuntu (o en cualquier distribución que use paquetes Debian) ejecutad:

```
$ sudo apt-get install gdb ddd
$
```

Para depurar con ddd un programa, primero lo compilamos con el '-g' y luego invocamos a ddd

```
$ ddd ./foo &
$
```

Si nada más ejecutar el ddd le damos al icono 'STOP' nos creará un breakpoint en la primera linea que va a ser ejecutada. Para crear un breakpoint en otra linea clicamos primero en la linea y luego en el icono 'STOP'. Luego le podemos dar a 'Run' para que empiece el programa hasta encontrar el primer breakpoint y a partir de entonces irle dando a 'Next' para que vaya ejecutando el programa linea a linea. Si la siguiente linea a ejecutar es una llamada a una función 'Step' sirve para ir "a dentro" de la función. 'Cont' hace que el programa continúe hasta el siguiente breakpoint o hasta terminar. En el menú 'Status' tenéis la opción 'Registers' que os mostrará los registros. Si os ponéis sobre el nombre de una variable os indicará su valor (en caso de problemas id al nombre de la variable en la línea donde está declarada.)
Los usuarios del editor 'vi' o 'vim' pueden usar el cgdb. El cgdb es un editor

modo texto estilo 'vi'. Ubuntu tiene el paquete 'cgdb'. Si no usas el 'vi' de forma habitual ni lo intentes, no tienes el suficiente nivel friki para esto. Más información sobre cgdb en:
http://cgdb.github.io/

Editando el código fuente: Si usáis Ubuntu yo recomiendo que utilicéis gedit o vuestro editor preferido. Para editar el código fuente 'foo.s'

```
$ gedit foo.s &
$
```

Yo utilizo el vim o el gvim, pero es un editor muy poco intuitivo y se tarda mucho en aprender a utilizarlo.

Sintaxis utilizada en este libro: El código ensamblador del IA32 (Arquitectura Intel de 32 bits, también conocida como i386 porque se usó por primera vez en el Intel 386) se puede escribir utilizando dos sintaxis diferentes: la Intel y la AT&T. En este libro utilizaremos la sintaxis AT&T. Tenedlo presente cuando miréis otros libros o ejemplos de códigos en ensamblador.

TRUCOS

- Es **sintaxis AT&T** si en el código aparecen '$' y '%' y probablemente también aparecen '(', ')'.
- Es **sintaxis Intel** si en el código **no** aparecen '$' y '%' y probablemente si que aparecen '[' , ']'.

Que debeis saber antes de empezar:

- Saber utilizar la consola.
- Saber utilizar un editor de texto.
- Saber programar en C.

2.3. Trucos para frikis

Si no eres un friki sáltate esta sección! Si no usas la consola de forma habutial ni lo intentes, no tienes el suficiente nivel friki para esta sección. Luego no digas que no te lo advertí. ;-P

2.3.1. Cómo ver el código ensamblador de un programa en C

```
$ cat foo.c
int a=0;

main(){
    a = a + 2;
}
$ gcc -g -c foo.c
$ LANG=C objdump -d -M suffix foo.o

foo.o:     file format elf32-i386


Disassembly of section .text:

00000000 <main>:
   0:   55                      pushl   %ebp
   1:   89 e5                   movl    %esp,%ebp
   3:   a1 00 00 00 00          movl    0x0,%eax
   8:   83 c0 02                addl    $0x2,%eax
   b:   a3 00 00 00 00          movl    %eax,0x0
  10:   5d                      popl    %ebp
  11:   c3                      retl
$ LANG=C objdump -S -M suffix foo.o

foo.o:     file format elf32-i386


Disassembly of section .text:

00000000 <main>:
int a=0;

main(){
   0:   55                      pushl   %ebp
   1:   89 e5                   movl    %esp,%ebp
    a = a + 2;
   3:   a1 00 00 00 00          movl    0x0,%eax
   8:   83 c0 02                addl    $0x2,%eax
   b:   a3 00 00 00 00          movl    %eax,0x0
}
  10:   5d                      popl    %ebp
  11:   c3                      retl
```

```
$
```

El 'LANG=C' es para obligarle a mostrarme la salida en inglés.

2.3.2. Cómo ver el código ensamblador de sólo una función

```
$ cat foo.c
int a=0;

main(){
    a = a + 2;
}
$ gcc -g -o foo foo.c
$ gdb foo
GNU gdb (GDB) 7.5.91.20130417-cvs-ubuntu
[...]
(gdb) disassemble main
Dump of assembler code for function main:
   0x080483ec <+0>: push   %ebp
   0x080483ed <+1>: mov    %esp,%ebp
   0x080483ef <+3>: mov    0x804a020,%eax
   0x080483f4 <+8>: add    $0x2,%eax
   0x080483f7 <+11>:    mov    %eax,0x804a020
   0x080483fc <+16>:    pop    %ebp
   0x080483fd <+17>:    ret
End of assembler dump.
(gdb) quit
$
```

2.3.3. Otra manera de generar ensamblador a partir de C

```
$ cat foo.c
int a=0;

main(){
    a = a + 2;
}
$ gcc -S foo.c
$ cat foo.s
    .file   "foo.c"
    .globl  a
    .bss
    .align 4
    .type   a, @object
```

```
    .size   a, 4
a:
    .zero   4
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushl   %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl    %esp, %ebp
    .cfi_def_cfa_register 5
    movl    a, %eax
    addl    $2, %eax
    movl    %eax, a
    popl    %ebp
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3"
    .section    .note.GNU-stack,"",@progbits
$
```

2.3.4. Utilizando el gdb a pelo para debugar

```
$ cat flags.s
.bss
.comm a, 4, 1
.comm b, 4, 1
.text
.global main
main:
 movb $1, a
 movb $-1, b
 movb a, %al
 subb b, %al
 movl $0, %ebx
 movl $1, %eax
 int $0x80
$ gcc -g -o flags flags.s
```

```
$ gdb flags
GNU gdb (GDB) 7.5.91.20130417-cvs-ubuntu
[...]
(gdb) break main
Breakpoint 1 at 0x80483ec: file flags.s, line 7.
(gdb) run
Starting program: /???/flags

Breakpoint 1, main () at flags.s:7
7     movb $1, a
(gdb) next
8     movb $-1, b
(gdb) next
9     movb a, %al
(gdb) next
10    subb b, %al
(gdb) next
11    movl $0, %ebx
(gdb) info registers
eax            0x2  2
ecx            0xbffff204    -1073745404
edx            0x80483ec     134513644
ebx            0xb7fc1000    -1208217600
esp            0xbffff16c    0xbffff16c
ebp            0x0  0x0
esi            0x0  0
edi            0x0  0
eip            0x8048405     0x8048405 <main+25>
eflags         0x213     [ CF AF IF ]
cs             0x73 115
ss             0x7b 123
ds             0x7b 123
es             0x7b 123
fs             0x0  0
gs             0x33 51
(gdb) eflags
Undefined command: "eflags".  Try "help".
(gdb) quit
A debugging session is active.

    Inferior 1 [process 5727] will be killed.

Quit anyway? (y or n) y
$
```

Fijaros que con 'info registers' los flags están en 'eflags'.

2.4. Los ejemplos de este libro

El código fuente de todos los ejemplos de este libro, al igual que el libro en formato PDF, se pueden descargar de mi página de la asignatura:

http://people.ac.upc.edu/guerrero/fo.html

En la página tenéis el PDF del libro y los ejemplos del libro en un fichero comprimido llamado 'asm_src.tgz'. Os bajáis el fichero comprimido a un directorio. Y para descomprimir el fichero hacéis:

```
$ cd <nombre_del_directorio_donde_esta_el_fichero_tgz>
$ tar xzvf asm_src.tgz
[...]
$
```

Esto os creará un subdirectorio llamado 'asm_scr' que contendrá todos los ficheros de ejemplo.
Veréis que empiezan por 't5', 't6' y 't7'. Eso, como ya habréis intuido es por que corresponden a los temas 5, 6 y 7 de la asignatura.
En la página de la asignatura hay versiones de estos mismos ejemplos pero pueden ser ligeramente diferentes.

2.5. Otros libros

Otros libros sobre IA-32 con sintaxis AT&T (aunque no me convencen, y no porque estén en inglés):

- "Programming from the Ground Up" por Jonathan Bartlett.
  El PDF está aquí:
  http://mirrors.fe.up.pt/pub/nongnu//pgubook/ProgrammingGroundUp-1-0-lettersize.pdf
  Los códigos fuentes están aquí junto con todos los ficheros fuente para generar el libro:
  http://cvs.savannah.gnu.org/viewvc/pgubook/pgubook/ProgrammingGroundUp/

- "Professional Assembly Language" por Richard Blum.

2.6. Si encuentras algún error

Este libro se ha escrito de unos pocos tirones y en un tiempo récord. Seguro que está plagado de errores.

Si encuentras algún error (ya sea en los programas, faltas de ortografía, etcétera) o tienes alguna idea de cómo mejorar este libro y quieres enviarme un e-mail, puedes hacerlo a: guerrero arroba ac punto upc punto edu. Haz que el subject (asunto?) del e-mail empiece por "[Libro IA32]". Decidme cual es la fecha que aparece en la portada para que sepa a que versión os referís. Muchas gracias. :-)

Capítulo 3

Primeros programas

3.1. El IA32

En esta sección voy a describir sólo lo imprescindible del IA32 para poder programar en ensamblador.

Verás que esta primera sección es super densa y puede que no termines de entenderla del todo. Pero con los ejemplos de las siguientes secciones ya te irá quedando todo más claro.

3.1.1. La memoria

El IA32 (Intel 386 y posteriores) es una arquitectura de 32 bits que tiene la memoria organizada en 2^{32} palabras de 8 bits (1 byte). Cada uno de estos bytes se ocupa una posición de memoria del 0 al $2^{32} - 1$. Como 2^{32} bytes son 4 Gigabytes, el IA32 puede gestionar hasta 4 Gigabytes de memoria. Al ser una arquitectura de 32 bits cuando lee o escribe en memoria siempre lo hace en direcciones múltiples de 4 y siempre lee o escribe 4 bytes (32 bits). Puede ser útil pensar en la memoria como un vector 2^{32} bytes.

Las instrucciones y los datos de los programas se guardan en memoria. Los datos, si son números naturales se guardan en binario, si son caracteres se guarda su valor ASCII, y si son números enteros con signo se guardan en complemento a 2.

A continuación tenéis una tabla con los números naturales, en "Complemento a 1" (Ca1) y "Complemento a 2" (Ca2) que se pueden codificar con tres bytes:

n	bin	n	Ca1	n	Ca2
0	000	3	011	3	011
1	001	2	010	2	010
2	010	1	001	1	001
3	011	0	000	0	000
4	100	-0	111	-1	111
5	101	-1	110	-2	110
6	110	-2	101	-3	101
7	111	-3	100	-4	100

Podéis ver que en Ca1 y Ca2 el signo está en el bit de más peso (1='-';0='+'). Que en Ca1 los bits que no son el signo son la codificación binaria del número sin signo. Y que el Ca2 de un número negativo es igual al Ca1+1. Se usa Ca2 en lugar de Ca1 porqué nos permite operar con números sin preocuparnos de si son con o sin signo y porqué no tiene dos codificaciones diferentes para el cero.

TRUCOS

- Conversión rápida a Ca2 para humanos:
  #1: empezando por la derecha copiar el número original hasta el primer 1 (primer 1 incluido).
  #2: Después, se niegan (complementan) los dígitos restantes (es decir, se substituye 0 por 1 y viceversa).
  Ej: 92 -> -92
  #1 01011100 -> ?????100
  #2 01011100 -> 10100100

En cuanto a la tabla ASCII la podéis consultar en: http://www.asciitable.com/

3.1.2. Los registros

A parte de la memoria, el IA32 puede almacenar datos en lo que se denominan 'registros'. Cada registro puede almacenar un número binario de 32 bits. El acceso (lectura o escritura) a un registro es muchísimo más rápido que el acceso a memoria.

Existen 8 registros denominados "de propósito general":

- EAX (Accumulator). Para operaciones aritméticas.
- ECX (Counter). Contador para bucles (como la variable 'i' en C).

- EDX (Data). Para operaciones aritméticas y de Entrada/Salida.
- EBX (Base). Puntero a datos o a primer elemento del vector (la base del vector).
- ESI (Source Index). Puntero a origen o indice de origen.
- EDI (Destination Index). Puntero a destino o indice a destino.
- ESP (Stack Pointer). Puntero ala cima de a la pila.
- EBP (Stack Base Pointer). Puntero a la base de la pila.

La 'E' al principio de sus nombres viene del inglés "Extended" y nos indica que son las versiones de 32 bits de estos registros. La arquitectura I32 nos permite acceder a la mitad baja de los cuatro primeros. La mitad baja de un registro de 32 bits es un registro de 16 bits. A las mitades bajas de 'eax', 'ecx', 'edx' y 'ebx' se las llama 'ax', 'cx', 'dx' y 'bx'. El I32 también nos permite acceder a la mitad alta y a la mitad baja de estos cuatro registros de 16 bits. La mitad alta del 'ax' es un registro de 8 bits llamado 'ah' y la mitad baja es otro registro de 8 bits llamado 'al' ('h' de 'high' y 'l' de 'low').

Bits:	31..16	15..00
EAX:		AX
EBX:		BX
EDX:		DX
ECX:		CX
ESI:		
EDI:		
ESP:		
EBP:		

Bits:	31..24	23..16	15..08	07..00
EAX:			AH	AL
EBX:			BH	BL
EDX:			DH	DL
ECX:			CH	CL
ESI:				
EDI:				
ESP:				
EBP:				

Los dos últimos registros de propósito general (ESP y EBP) ya veremos exactamente para qué sirven cuando estudiemos llamadas a subrutinas (una subrutina es el equivalente de una función en C). Los 6 primeros los podemos utilizar para lo que queramos pero cuando los he listado os he indicado para qué suelen usarse.

A parte de estos ocho registros de propósito general, existen dos registros especiales:

- Flags: Es un registro donde cada uno de sus 32 bits nos informa de una cosa diferente. Hay cuatro flags especialmente importantes:
  - Flag 0. CF : Carry Flag. El bit de este flag (el bit número 0 del registro de flags) vale 1 si en la última operación aritmética ha sucedido un 'carry' (suma) o un 'borrow' (resta) más allá del último. Considerando que era una operación sin signo.
  - Flag 6. ZF : Zero Flag. 1 si el resultado de la última operación es 0.
  - Flag 7. SF : Sign Flag. 1 si el resultado de la última operación es un número negativo.
  - Flag 11. OF : Overflow Flag. 1 si el valor resultante de la última operación es demasiado grande para caber en un registro, en el caso de que esta fuera una operación con signo.
- EIP (Instruction Pointer) Contiene la dirección de memoria donde empieza la próxima instrucción a ejecutar, a no ser que se produzca un salto o una llamada a subrutina (función).

3.1.3. Tipos de datos, endianismo y alineación

En el IA32 tenemos los siguientes tipos de datos:

- Byte: 8 bits. (b)
- Word: 16 bits. (w)
- Long: 32 bits. (l)
- Posiciones de memoria (también de 32 bits).

En el I32, si un dato tiene más de 1 byte el byte de más peso se guarda en la posición más alta de memoria. A esto se le llama utilizar el formato "little endian".

A parte, por un tema de eficiencia, los datos se guardarán "alineados en memoria". Esto significa que un dato de 4 bits se almacenará empezando en una dirección de memoria divisible por 4 y un dato de 2 bits en una dirección divisible por 2. Si no lo hiciéramos así acceder a estos datos requeriría dos accesos a memoria.

El equivalente de C de un 'int' es un 'long' y de un 'char' es un 'byte'. Los 'word' se usan muy poco (son herencia de las arquitecturas de 16 bits, en las cuales sólo habían registros de 8 y 16 bits).

3.1.4. Modos de direccionamiento y la instrucción 'mov'

Una instrucción típicamente trabaja sobre uno o más operadores. Por ejemplo la instrucción 'mov' copia el valor de un primer operador a un segundo operador (que será un registro o una posición de memoria).

El formato típico de una instrucción es:
INSTRUCCION OPERADOR1 OPERADOR2
Donde al operador2 también se lo conoce como "operador de destino"

También tenemos instrucciones que trabajan con un único operador:
INSTRUCCION OPERADOR1

Los modos de direccionamiento sirven para indicar donde están los operadores de una instrucción. Tenemos los siguientes:

- 1. Inmediato: '$100' (el número 100) (Trampa: el operador de destino nunca puede ser inmediato.)
- 2. Registro: '%eax' (el registro eax)
- 3. Memoria.

Ejemplos de 'mov': Asignar un 0 al registro eax:

```
1 movl $0, %eax
```

Copiar el valor del registro eax al registro ebx:

```
1 movl %eax, %ebx
```

Asignar el valor 1000 al registro ebx:

```
1 movw $1000, %bx
```

Asignar el valor ASCII del carácter 'A' al registro al:

```
1 movb $'A', %al
```

Fíjate que 'mov', al igual que todas las instrucciones que operan con datos termina con 'l', 'w' o 'b' dependiendo de si opera con longs, words o bytes. Y si bien es cierto que compilará sin problemas en la mayoría de los casos aunque no le añadamos la 'l', 'w' o 'b', lo mejor es que nos acostumbremos a ponerla siempre.

En cuanto al modo de direccionamiento a memoria tenemos:

- 3.a. Memoria, Normal: '(%eax)' = valor en la posición de memoria contenida en el registro eax. Ejemplo: 'movl (%ecx), %eax' lee el long contenido en la posición de memoria contenida en el registro ecx y la copia al registro eax.

- 3.b. Memoria, Desplazamiento(Base): '8(%ebp)' = valor en la posición de memoria resultante de sumar 8 al contenido del registro ebp. Ejemplo: 'movl 8(%ebp), %edx' lee el long contenido en la posición de memoria ebp+8 y la copia al registro edx. El desplazamiento tanto puede ser una constante como el nombre de una variable (en este último caso el valor del desplazamiento será la posición de memoria donde esta variable empieza).

- 3. Memoria, Indexado: D(Rb,Ri,Fe) El operando está en la dirección de memoria resultante de calcular: D+Rb+Ri*S. Donde:

  - D: Desplazamiento: Constantie o variable.
  - Rb: Registro base: Cualquier registro.
  - Ri: Registre índice: Cualquier registro menos %esp.
  - S: Factor de escalación: 1, 2, o 4 (ya veremos cuando estudiemos vectores para que es útil esto).
  - Si uno de ellos no aparece, por defecto: D=0, Rb=0, Ri=0 y Fe=1.

Copiar el long que empieza el la posición de memoria 4+ %ebp+ %ebx a %eax:

```
1 movl 4(%ebp,%ebx), %eax
```

Copiar el long que empieza el la posición de memoria tab+ %ebp+ %ebx*2 a %eax:

```
1 movl tab(%ebp,%ebx,2), %eax
```

Copiar el long que empieza el la posición de memoria %eax+ %ebx a %edx:

```
1 movl (%eax, %ebx), %edx
```

TRAMPAS

- Solamente 1 operando puede estar en memoria.
  'movl (%eax), (%ebx)' dará error de compilación!
  'movl a, b' dará error de compilación!
- Esto sucede en todas las instrucciones! No solo en 'mov'.

EL BOXEO

- El ensamblador es como el boxeo porque todo va al revés:
  El 'mov' es el equivalente del '=' en C. En C asignamos el valor de la derecha a la variable de la izquierda 'a=0;'. En ensamblador con sintaxis AT&T asignamos el valor de la izquierda (operador 1) a la variable o registro de la derecha (operador 2) 'movl $0, %eax';

A continuación tenemos una comparativa entre la sintaxis del IA32 y C. Aunque en las dos últimas filas es obvio que C no accede directamente al registro:

IA32	C	Descripción
$1	1	El número '1'
1	&1	La dirección de memoria '1'
i	i	El valor de la variable 'i'
$i	&i	Posición de memoria donde está la variable 'i'
%eax	eax	Valor guardado en el registro eax.
(%eax)	*eax	Valor guardado en la pos de memoria en eax

Debes tener la cabeza a punto de estallar y la sensación de que no entiendes nada. No te preocupes. A partir de ahora empezaremos a ver ejemplos y empezarás a relacionar conceptos.

Confía en mi. :-)

3.2. Mi primer programa

Código fuente 3.1: `t5_p01_add.s`

```
1  #===[Equivalencia en C]=============
2  # int a=2, b=3, r;
3  # r = a + b;
4
5  .data          # initialized global data
6  a: .long 2     # int a = 2;
7  b: .long 3     # int b = 3;
8
9  .bss           # uninitialized global data
10 .comm r,4,4    # int r;
```

```
11
12  .text        # code segment
13  .global main     # So the OS can call 'main'
14
15  main:        # main() {
16   movl a, %eax    # a -> eax
17   addl b, %eax    # b + eax -> eax
18   movl %eax, r    # eax -> r
19
20   # LINUX EXIT
21   #movl $0, %ebx # Return 0 (the usual)
22   movl r, %ebx    # Return r (lowest byte)
23   movl $1, %eax
24   int $0x80
```

```
$ gcc -g -o t5_p01_add t5_p01_add.s
$ ./t5_p01_add
$ echo $?
5
$
```

EL CÓDIGO

- Líneas 1-3: Este es nuestro primer programa que suma 2+3. Aquí tenemos el código equivalente en C. El '#' indica que el resto de la línea es un comentario.
- Líneas 5-7: El bloque 'data' contiene las variables que declaramos e inicializamos. En este caso dos longs (a=2 y b=3).
- Línea 9-10: El bloque 'bss' contiene las variables que declaramos pero no inicializamos. En este caso la variable 'r' de 4 bytes (long) alineada a 4 bytes.
- Línea 12: El bloque 'text' contiene las instrucciones del programa
- Línea 13: Un programa siempre empieza haciendo publica la etiqueta 'main' para que pueda ser invocada por el Sistema Operativo.
- Línea 15: La etiqueta 'main', donde empieza el programa.
- Línea 16: Copiamos 'a' al registro 'eax'.

- Línea 17: Sumamos ('add' significa sumar en inglés) 'b' con el contenido del registro 'eax' y el resultado lo guardamos en 'eax'. Ahora 'eax' contiene 'a+b'.
- Línea 18: Copiamos el contenido de 'eax' a la variable 'r'.
- Línea 20: El programa ya ha acabado y ahora debemos retornar el control al Sistema Operativo. En Linux se hace de la siguiente manera:
- Línea 21: Normalmente se copia un '0' al registro 'ebx'. Aquí está comentado.
- Línea 22: Copiamos el resultado de la ejecución (la variable 'r') al registro 'ebx'.
- Línea 23: Copiamos un '1' al registro 'eax'.
- Línea 24: Llamamos a la interrupción '0x80' (80 hexadecimal). Esta interrupción lo que hace es (en el caso de que el registro 'eax' valga '1' termina la ejecución del programa devolviendo el contenido del registro 'ebx' al sistema operativo.

TRUCOS

- Lo compilamos con el gcc y lo ejecutamos. Justo después de ejecutarlo tecleamos 'echo $?', que hace que el Sistema Operativo nos muestre que es lo que le ha devuelto el último programa ejecutado modulo 256. Recordad que cuando un programa termina su ejecución pasa el contenido de 'ebx' (la parte 'bl' para ser exactos) al Sistema Operativo. Este pequeño truco nos permitirá ver que los programas funcionan correctamente sin necesidad de un depurador hasta que aprendamos a hacer 'printf'.
- Este truco lo saqué del libro: "Programming from the Ground Up" (por Jonathan Bartlett).

3.3. Copiando datos de un sitio a otro

Como vimos anteriormente, para copiar datos de un sitio a otro utilizaremos la instrucción 'mov' (del inglés 'move': mover; aunque más que mover tendría

que ser copiar).

```
1 movl op1, op2
```

Copia un long (4 bytes) del operando 1 al operando 2. Ejemplo: 'movl var, %ebx' copia el contenido de la variable 'var' al registro 'ebx'. Los dos operadores tienen que ser de tipo long.

```
1 movw op1, op2
```

Copia un word (2 bytes) del operando 1 al operando 2. Ejemplo: 'movw $0, %bx' copia la constante '0' al registro 'bx'. Los dos operadores tienen que ser de tipo word.

```
1 movb op1, op2
```

Copia un byte del operando 1 al operando 2. Ejemplo: 'movb %ah, %al' copia el contenido del registro 'ah' al registro 'al'. Los dos operadores tienen que ser de tipo byte.

Si necesitamos copiar de byte o word a un operando de mayor tamaño, podemos utilizar 'movz' o 'movs'.

'movz' mueve el operando 1 al operando 2 "extendiendo ceros" (la 'z' es porqué en inglés cero es "zero"). Esto significa que copia el operando 1 a la parte baja del operando 2 y pone a '0' el resto del operando 2.

```
1 movz[bw|bl|wl] op1, op2
```

Ejemplo: para mover un byte a un long "extendiendo ceros" (rellenando con ceros) haríamos 'movzbl

No obstante si el número es un entero con signo lo que nos interesará es hacer un 'movs'. 'movs' mueve el operando 1 al operando 2 "extendiendo el signo". Esto significa que copia el operando 1 a la parte baja del operando 2 y copia el signo del operando 1 (su bit de más peso, porqué los enteros con signo los codificamos en Complemento a 2) al resto de los bits del operando 2.

En el siguiente ejemplo 'al' vale '-1' y lo copiamos a 'ebx':

```
1 movl $-1, %al
2 movsbl %al, %ebx
```

Usamos 'movs' en lugar de 'movz' ya que '-1' es '0xFF' en hexadecimal y queremos que 'ebx' valga '-1' (0xFFFFFFFF) y no '0x000000FF' (+255).

3.4. Operaciones aritméticas

Instrucción	Operación
add[bwl] op1, op2	op2 = op2 + op1
sub[bwl] op1, op2	op2 = op2 - op1
inc[bwl] op1	op1++
dec[bwl] op1	op1–
neg[bwl] op1	op1 = -op1
imul[bwl] op1, op2	op2 = op1 * op2 (op2 registro)

Aquí vemos las instrucciones que podemos utilizar par sumar, restar, incrementar, decrementar, negar y para hacer multiplicaciones enteras. Es bastante intuitivo. Recordad que en la multiplicación entera el segundo operando tiene que ser un registro (no puede ser una posición de memoria). Los nombres vienen del inglés add, subtract, increment, decrement, negate, e integer multiplication (sumar, restar, incrementar, decrementar, negar y multiplicación entera).

TRAMPAS

- En 'imul' el operador 2 tiene que ser un registro!
  No puede ser ninguna variable ni posición de memoria.
  'imull %eax, var' dará error de compilación!

- Es 'imull', 'imulw' e 'imulb'. No 'imul', 'imuw' e 'imub'!

A continuación vemos un par de ejemplos que utilizan estas instrucciones:

Código fuente 3.2: `t5_p02_inc.s`

```
1  #===[Equivalencia en C]=============
2  # int i=0, j=3, r;
3  # i++; j--;
4  # j = -j;
5  # r = i - j;
6
7  .data          # initialized global data
8  i: .long 0     # int i = 0;
9  j: .long 3     # int j = 3;
10
11 .bss           # uninitialized global data
12 .comm r,4,4    # int r;
```

```
13
14 .text        # code segment
15 .global main
16
17 main:        # main() {
18  incl i      # i++
19  decl j      # j--
20  negl j      # j = -j
21
22  movl i, %eax   # i-> eax
23  subl j, %eax   # i-j -> eax
24
25  movl %eax, r   # i-j -> r
26
27  # LINUX EXIT
28  #movl $0, %ebx # Return 0 (the usual)
29  movl r, %ebx   # Return r (its lowest byte)
30  movl $1, %eax
31  int $0x80
```

```
$ gcc -g -o t5_p02_inc t5_p02_inc.s
$ ./t5_p02_inc
$ echo $?
3
$
```

Código fuente 3.3: `t5_p03_imul.s`

```
1  #===[Equivalencia en C]=============
2  # #define N = 10
3  # int i=2, j=3, k=6, r;
4  # r = (i + j) * ( k + (-2) ) + N;
5
6  N = 10       # define N 10
7
8  .data        # initialized global data
9  i: .long 2   # int i = 2;
10 j: .long 3   # int j = 3;
11 k: .long 4   # int k = 4;
12
13 .bss         # uninitialized global data
14 .comm r,4,4  # int r;
15
16 .text        # code segment
17 .global main    # So the OS can call 'main'
18
```

```
19  main:           # main() {
20    movl i, %eax    # i -> eax
21    addl j, %eax    # i+j -> eax
22    movl %eax, r    # i+j -> r
23
24    movl k, %eax    # k -> eax
25    addl $-2, %eax # k + (-2) -> eax
26    imull r, %eax   # (i+j) * (k+(-2)) -> eax
27  # El operando destino de imul tiene que ser registro
28    addl $N, %eax   # (i+j) * (k+(-2)) + N -> eax
29
30    movl %eax, r    # (i+j) * (k+(-2)) + N -> r
31
32    # LINUX EXIT
33    #movl $0, %ebx # Return 0 (the usual)
34    movl r, %ebx    # Return r (its lowest byte)
35    movl $1, %eax
36    int $0x80
```

```
$ gcc -g -o t5_p03_imul t5_p03_imul.s
$ ./t5_p03_imul; echo $?
20
$
```

EL BOXEO

- El ensamblador es como el boxeo porque todo va al revés:
  'add' es equivalente a '+=' en C.

  (Por si no lo sabéis, en C, 'a += 2;' equivale a 'a = a+2;'. ¡A que mola! Lo mismo con '-=', '*=', '/=', '%='. Por lo tanto 'add' equivale a '+=', 'sub' equivale a '-=' e 'imul' equivale a '*='.)

  Pero, como el ensamblador va al revés, si en C asignamos el resultado de la operación a la variable de la izquierda 'a+=2;'. En ensamblador con sintaxis AT&T asignamos el resultado de la operación a la variable o registro de la derecha (operador 2) 'addl $2, %eax';
  En todas las operaciones aritméticas con dos operadores el resultado de la operación se guarda en el operador 2 (el operador de la derecha).

TRAMPAS

- Recordad: En cualquier instrucción, solamente uno de los operandos puede estar en memoria (ser una variable o una dirección de memoria).
  'addl a, b' dará error de compilación!
  'addl (%eax), b' dará error de compilación!

3.5. Estructuras de control

Las estructuras de control permiten modificar el flujo de ejecución. En C teníamos el 'if/else', el 'while' y las llamadas a funciones. En ensamblador tendremos saltos condicionales, saltos incondicionales y llamadas a subrutinas (el equivalente a funciones en C).

Inmediatamente antes de hacer un salto condicional deberemos realizar una comparación entre dos operandos sobre la que se basará la condición de salto. Para ello utilizaremos la instrucción 'cmp'. 'cmp' resta el operando 1 al operando 2 pero no deja el resultado en ninguna parte. Lo único que hace es "setear" (activar/desactivar) los flags 'Cero', 'Signo' y 'Overflow de acuerdo con la resta.

Muy importante a tener en cuenta es que (técnicamente 'cmp' es también una instrucción aritmética) y al igual que en el resto de instrucciones aritméticas el segundo operador no puede ser de tipo inmediato. Esto era obvio en el resto de instrucciones aritméticas (ya que modifican el segundo operador al guardar allí el resultado de la operación).

Instrucción	Operación
cmp[bwl] op1, op2	op2 - op1 (op2 no puede ser immediato!)

Los saltos incondicionales se hacen con 'jmp etiq'. No usan 'cmp' antes. El ordenador al llegar a ejecutar esta instrucción lo que hace es saltar a la etiqueta 'etiq' y seguir la ejecución desde allí.

Aquí tenéis una tabla con las instrucciones de salto:

Instrucción	Operación	Flags
cmp op1, op2	op2 - op1	ZF, SF i OF
jmp etiq	Salta a etiq	
je etiq	Salta si op2 == op1	ZF == 1
jne etiq	Salta si op2 != op1	ZF == 0
jg etiq	Salta si op2 > op1	ZF=0 y SF=OF
jge etiq	Salta si op2 ≥ op1	SF=OF
jl etiq	Salta si op2 < op1	SF!=OF
jle etiq	Salta si op2 ≤ op1	ZF==1 o SF!=OF
call etiq	Invoca la subrutina etiq	

Los nombres vienen del inglés: 'cmp' compare (comparar), 'jmp' jump (saltar), 'je' jump if equal (saltar si son iguales), 'jne' jump if not equal (saltar si no son iguales), 'jg' jump if greater (saltar si es mayor), 'jge' jump if greater or equal (saltar si es mayor o igual), 'jl' jump if less (saltar si es menor), 'jle' jump if less or equal (saltar si es menor o igual),

Finalmente para invocar una subrutina, utilizaremos 'call' (llamar).

TRAMPAS

- En 'cmp' el operador no puede ser inmediato!
  (Inmediato significa una constante)
  'cmpl %eax, $0' dará error de compilación!
  Tenéis que hacer 'cmpl $0, %eax'.

EL BOXEO

- El ensamblador es como el boxeo porque todo va al revés:
  En la sintaxis AT&T se compara el operador 2 con el operador 1. Por ejemplo 'cmpl op1, op2' seguido de 'jg etiq' salta si op2 es mas grande que op1. Cuando lo intuitivo sería que saltara si op1 fuera más grande que op2.

3.6. ¿Cómo hago un 'if/else' en ensamblador?

```
1 if (a == 0) {
2     r = 1;
3 } else {
4     r = -1;
5 }
```

Para hacer un 'if' en ensamblador va bien pensar el código en C y luego:

1. hacer un 'cmp' de los dos operandos que comparamos pero al revés. Esto es porque en la sintaxis AT&T se compara el segundo operador con el primero. Así que haríamos: línea 1 'cmpl $0, a'. En C si uno de los dos operandos es una constante se suele poner a la derecha. Lo que nos va perfecto porque recordad que el segundo operador no puede ser inmediato (una constante).

2. Hacer el salto condicional con la condición para ir al 'else'. Si la condición del 'if' es 'si son iguales' la condición para ir al 'else' es 'si no son iguales'. Por lo tanto: línea 2 'jne else'.

3. Poner el bloque 'then': línea 3 'movl $1, r'. (En C, el bloque 'then' es el código que se ejecuta si la condición es cierta)

4. Después del bloque 'then' salto incondicional al final del 'if': línea 4 'jmp endif'.

5. Etiqueta del 'else': línea 5 'else:'.

6. El bloque de código 'else': línea 6: 'movl $-1, r'. (En C, el bloque 'else' es el código que se ejecuta si la condición es falsa)

7. Etiqueta de fin de 'if': línea 7: 'endif:'

```
1     cmpl $0, a
2     jne else
3     movl $1, r
4     jmp endif
5 else:
6     movl $-1, r
7 endif:
```

Aquí tenemos un ejemplo completo de un programa simple con un 'if':

Código fuente 3.4: `t5_p04_if.s`

```
1 #===[Equivalencia en C]=============
2 # int a=3, b=3, r;
3 # if (a >= b) r=1; else r=0;
```

```
4
5  .data          # initialized global data
6  a: .long 3     # int a = 3;
7  b: .long 3     # int b = 3;
8
9  .bss           # uninitialized global data
10 .comm r,4,4    # int r;
11
12 .text          # code segment
13 .global main
14
15 main:          # main() {
16  movl a, %eax
17  cmpl b, %eax    # if (a >= b)
18  jl else         # {
19  movl $1, r #    r=1;
20  jmp endif  # }
21 else:          # else {
22  movl $0, r #    r=0;
23 endif:         # }
24
25  # LINUX EXIT
26  #movl $0, %ebx # Return 0 (the usual)
27  movl r, %ebx    # Return r (its lowest byte)
28  movl $1, %eax
29  int $0x80
```

```
$ gcc -g -o t5_p04_if t5_p04_if.s
$ ./t5_p04_if ;echo $?
1
$
```

EL CÓDIGO

- Paso 1) Línea 16-17: Atentos a la trampa! No podemos hacer un 'cmpl b, a'. ¿Porque? Por que en ensamblador del IA32 los dos operandos no pueden estar en memoria. (Si, las variables se guardan en la memoria). Por lo tanto, primero moveremos 'a' a un registro y luego compararemos 'b' con ese registro. Fijaros que intercambio los operadores de 'C' a ensamblador.

- Paso 2) Línea 18: Hago el salto condicional de la condición para saltar al 'else'. Si la condición del 'if' era 'si mayor o igual que' su negación (y por tanto la condición del 'else') será 'si menor que'. Por lo tanto saltaremos a la etiqueta 'else' si 'a es menor que b'.
- Paso 3) Línea 19: Bloque 'then'.
- Paso 4) Línea 20: Salto incondicional al final del 'if'.
- Paso 5) Línea 21: Etiqueta 'else'.
- Paso 6) Línea 22: Bloque 'else'.
- Paso 6) Línea 23: Etiqueta 'endif'.

TRAMPAS

- El salto condicional con la condición opuesta al salto 'jge' es 'jl', no 'jle.
- El salto condicional con la condición opuesta al salto 'jg' es 'jle', no 'jl.
- No puede haber dos etiquetas iguales en un programa. Si hacéis otro 'if' utilizad etiquetas diferentes. Como por ejemplo: 'else1', 'endif1' y 'else2', 'endif2'.
- No podemos hacer un 'cmp' con dos variables o con dos operandos que estén en memoria. De hecho ninguna instrucción puede tener sus dos operandos en memoria. La solución es mover uno de ellos a un registro y comparar la variable con el registro.

EL BOXEO

- El ensamblador es como el boxeo porque todo va al revés:
  En un 'if' los operandos de la comparación están del revés y el salto

condicional es con la condición del 'else' en lugar de la del 'then'.

3.7. ¿Cómo hago un 'while' en ensamblador?

```
1 i = 1;
2 while (i <= 10) {
3       r = r+i;
4       i++;
5 }
```

Para hacer un bucle en ensamblador va bien pensar primero el código en C. En ensamblador a un bucle le llamaremos 'loop'. Es más inmediato traducir un 'while' a ensamblador que un 'for'. Cuando ya tenemos el 'while' en C:

1. La inicialización de la variable contador ya la sabemos hacer: línea 1 'movl $1, %ecx'.

2. Empieza el bucle. Necesitaremos una etiqueta aquí: línea 2 'loop:' ('while:' también estaría bien).

3. hacer un 'cmp' de los dos operandos que comparamos en la condición del 'while' en C pero al revés. Esto es porque en la sintaxis AT&T se compara el segundo operador con el primero. Así que haríamos: línea 3 'cmpl $10, %ecx'. Acordaros, que al igual que en el caso del 'if' en C si uno de los dos operandos es una constante se suele poner a la derecha. Lo que nos va perfecto porque en ensamblador el segundo operador de una instrucción no puede ser inmediato (una constante).

4. Hacer el salto condicional al final del bucle con la condición de salida del bucle (que es la opuesta a la condición de permanencia que aparece en el 'while' en C). Si la condición de permanencia del 'while' era 'i menor o igual que 10' en ensamblador la condición de finalización del 'loop' será ' %exc mayor que 10'. Por lo tanto: línea 4 'jg endloop'.

5. Poner el cuerpo del bucle: línea 5 'addl %ecx, r'.

6. Después del cuerpo del bucle incremento el contador del bucle: línea 6 'incl %ecx'.

7. Salto incondicional al principio del bucle: línea 7 'jmp loop'.

8. Etiqueta de fin de bucle: línea 8: 'endloop:'

```
1       movl $1, %ecx
2   loop:
3       cmpl $10, %ecx
4       jg endloop
5       addl %ecx, r
6       incl %ecx
7       jmp loop
8   endloop:
```

Aquí tenemos un ejemplo completo de un programa simple con un 'while':

Código fuente 3.5: `t5_p05_while.s`

```
1  #===[Equivalencia en C]=============
2  # int f=1, n=3, i;
3  # i=1; while(i<=n) {f=f*i;i++;}
4  # retornar f; // f <- n factorial
5
6  .data          # initialized global data
7  f: .long 1     # int f = 1;
8  n: .long 3     # int n = 3;
9
10 .text          # code segment
11 .global main
12
13 main:          # main() {
14  movl f, %eax
15  movl $1, %ecx       # i = 1; // ecx = i
16 while:              # while (i<=n)
17  cmpl n, %ecx
18  jg endwhile
19  imull %ecx, %eax    # f=f*i; # op2 registro!
20  incl %ecx           # i++;
21  jmp while
22 endwhile:
23  movl %eax, f
24
25  # LINUX EXIT
26  movl f, %ebx        # Return f (its lowest byte)
27  movl $1, %eax
28  int $0x80
```

```
$ gcc -g -o t5_p05_while t5_p05_while.s
$ ./t5_p05_while; echo $?
6
$
```

EL CÓDIGO

- Paso 1) Línea 15: De la misma manera que en C se suele usar la variable 'i' como contador, en ensamblador se suele usar el registro '%ecx'. Lo inicializamos a cero.
- Paso 2) Línea 16: La etiqueta 'while' a la que saltaremos en cada iteración.
- Paso 3) Línea 17: La comparación con los operandos al revés que en C.
- Paso 4) Línea 18: El salto condicional con la condición de salida del bucle a la etiqueta de final de bucle. Si la condición de permanencia del 'while' era que 'i sea menor que n' la de salida será que '%ecx sea mayor que n'.
- Paso 5) Línea 19: El cuerpo del bucle.
- Paso 6) Línea 20: Incremento del contador del bucle.
- Paso 7) Línea 21: Salto incondicional al principio del bucle.
- Paso 8) Línea 22: Etiqueta de final de bucle.

TRAMPAS

- Recordad las trampas de 'if/else' que aquí también están.
- Pequeña trampa: El segundo operando de 'imul' tiene que ser registro. Por eso en la linea 14 movemos 'f' a registro.

EL BOXEO

- El ensamblador es como el boxeo porque todo va al revés:
  En un 'while' los operandos de la comparación están del revés y el salto condicional es con la condición para salir del bucle en lugar de la de permanencia en el bucle.

3.8. ¿Cómo hago un 'while(a AND b)' en ensamblador?

Veamos un programa en C muy simple que implementa el problema de "el tablero de ajedrez y los granos de trigo": Se dice que el rey quiso premiar al inventor del ajedrez y le preguntó que quería como recompensa. Y lo "único" que le pidió el inventor fue trigo, en concreto pidió que el rey le diera un grano de trigo por la primera casilla del ajedrez, el doble por la segunda, el doble por la tercera, y así sucesivamente hasta llegar a la casilla número 64. El rey le dijo que ningún problema. Pero cuando sus matemáticos hicieron el calculo le dijeron que no había suficiente grano en todo el reino.
El programa que vemos a continuación calcula lo que hay en las primeras casillas hasta la primera casilla que contenga 100 granos o más.

Código fuente 3.6: `t5_while_and.c`

```
1  #include <stdio.h>
2
3  main(){
4       int n=1, i=0;
5       while (n<100 && i<64) {
6            n=n*2;
7            i++;
8            printf("i = %d, n = %d\n", i, n);
9       }
10 }
```

```
$ gcc -g -o t5_while_and t5_while_and.c
$ ./t5_while_and_c
i = 1, n = 2
i = 2, n = 4
i = 3, n = 8
i = 4, n = 16
i = 5, n = 32
i = 6, n = 64
i = 7, n = 128
```

```
$
```

A continuación vemos el mismo código pero ahora en ensamblador. Como aún no sabemos hacer 'printf', sólo retornaremos la cantidad de granos de la primera casilla que contiene 100 o más granos.

Código fuente 3.7: `t5_while_and.s`

```
1  #===[C equivalente]==============
2  # int n=1, i=0;
3  # while (n<100 && i<64) {n=n*2; i++;}
4  # retornar n;
5
6  .data          # initialized global data
7  n: .long 1     # int n = 1;
8  i: .long 0     # int i = 0;
9
10 .text          # code segment
11 .global main
12
13 main:
14 movl n, %eax
15 movl i, %ecx
16 while:
17  cmpl $100, %eax
18  jge endwhile    # %eax>=100
19  cmpl $64, %ecx
20  jge endwhile    # %ecx>=10
21  imull $2, %eax
22  incl %ecx
23  jmp while
24 endwhile:
25  movl %eax, n
26
27  # LINUX EXIT
28  movl n, %ebx # Retorna n%256 !
29  movl $1, %eax
30  int $0x80
```

```
$ gcc -g -o t5_while_and t5_while_and.s
$ ./t5_while_and; echo $?
128
$
```

EL CÓDIGO

- Líneas 17-20: El concepto es simple. Para implementar 'while (a AND b)' lo que haremos es, justo después de la etiqueta 'while:' si 'a' no se cumple saltar al final del bucle y, luego, si 'b' no se cumple saltar al final del bucle.
- Línea 21: Después de esto viene el cuerpo del bucle (al que sólo iremos a parar si tanto 'a' como 'b' se cumplían).

3.9. ¿Cómo hago un 'while(a OR b)' en ensamblador?

Este caso es un poco más complejo. Cojamos este ejemplo en C (que no hace nada útil):

Código fuente 3.8: `t5_while_or.c`

```
 1 #include<stdio.h>
 2 main(){
 3       int n=7, m=1;
 4       while (n<100 || m<100) {
 5             n=n*2;
 6             m=m*3;
 7             printf("n=%d\tm=%d\n",n,m);
 8       }
 9       printf("Retornar n=%d\n",n);
10 }
11 /*
12 n=14      m=3
13 n=28      m=9
14 n=56      m=27
15 n=112     m=81
16 n=224     m=243
17 Retornar n=224
18 */
```

```
$ gcc -g -o t5_while_or t5_while_or.c
$ ./t5_while_or
n=14     m=3
```

```
n=28    m=9
n=56    m=27
n=112   m=81
n=224   m=243
Retornar n=224
$
```

El código traducido al ensamblador sería el siguiente:

Código fuente 3.9: `t5_while_or.s`

```
1  #===[C equivalente]=============
2  # int n=7; m=1;
3  # while (n<100||m<100) {n=n*2; m=m*3;}
4  # retornar n;
5
6  .data          # initialized global data
7  n: .long 7     # int n = 7;
8  m: .long 1     # int m = 1;
9
10 .text          # code segment
11 .global main
12
13 main:
14   movl n, %eax
15   movl m, %ebx
16 while:         # while
17   cmpl $100, %eax
18   jl ok        # %eax<100
19   cmpl $100, %ebx
20   jge endwhile   # %ebx>=100
21 ok:      # Cuerpo del bucle
22   imull $2, %eax
23   imull $3, %ebx
24   jmp while
25 endwhile:
26   movl %eax, n
27
28   # LINUX EXIT
29   movl n, %ebx
30   movl $1, %eax
31   int $0x80
```

```
$ gcc -g -o t5_while_or t5_while_or.s
$ ./t5_while_or; echo $?
224
$
```

EL CÓDIGO

- Líneas 17-18: Para implementar 'while (a OR b)' lo que haremos es, justo después de la etiqueta 'while:' si 'a' es cierto 'a OR b' es cierto y, por lo tanto, saltamos al cuerpo del bucle ('ok:').
- Líneas 19-20: Si 'a' era falso, si no 'b' es falso 'a OR b' es falso y por lo tanto saltamos al final del bucle, de lo contrario iremos a la siguiente línea que es el principio del cuerpo del bucle.

3.10. Bucles anidados

Ya para acabar, vamos a ver un ejemplo con un bucle dentro de otro bucle:

Código fuente 3.10: `t5_for_for.c`

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

```
$ gcc -g -o t5_for_for t5_for_for.c
$ ./t5_for_for
i=1, j=1, n=1
i=1, j=2, n=2
i=1, j=3, n=3
i=1, j=4, n=4
i=2, j=2, n=5
i=2, j=3, n=6
i=2, j=4, n=7
i=3, j=3, n=8
i=3, j=4, n=9
i=4, j=4, n=10
$
```

El código traducido al ensamblador, que retornaría 'n' al Sistema Operativo, sería el siguiente:

Código fuente 3.11: `t5_for_for.s`

```
1  #===[C equivalente]==============
2  # n=0;
3  # for (i=1;i<5;i++)
4  #    for (j=i;j<5;j++)
5  #      n++; ret n;
6
7  .text          # code segment
8  .global main
9  main:          # main() {
10  movl $0, %eax
11  movl $1, %esi
12 fori:
13  cmpl $5, %esi
14  jge endfori #  %esi>=5
15  movl %esi, %edi
16 forj:
17  cmpl $5, %edi
18  jge endforj #  %edi>=5
19  incl %eax
20  incl %edi
21  jmp forj
22 endforj:
23  incl %esi
24  jmp fori
25 endfori:
26  # LINUX EXIT
27  movl %eax, %ebx
28  movl $1, %eax
29  int $0x80
```

```
$ gcc -g -o t5_for_for t5_for_for.s
$ ./t5_for_for;echo $?
10
$
```

EL CÓDIGO

- : El concepto es simple. El cuerpo del primer bucle (líneas 11-25) contiene el segundo bucle (líneas 15-22). Y, debemos utilizar etiquetas y registros contador diferentes para cada bucle.

3.11. Estructura d'un programa en IA32

Para acabar este capitulo os dejo un recordatorio de la estructura de un programa en IA32 con sus definiciones de constantes, declaración de variables y bloque del programa con su finalización de ejecución. Más o menos ya lo habíamos visto todo con excepción de como se definen constantes.

```
1  # Definicion de constantes
2  N = 20          # decimal
3  X = 0x4A        # hexadecimal
4  B = 0b010011 # binario
5  C = 'J'         # caracter
6  .data           # declaracion variables initializadas
7  ii: .long 1  # int ii = 1;
8  ss: .word 2  # short ss = 2;
9  cc: .byte 3  # char cc = 3;
10 s1: .ascii "Hola\n" # Chars
11 s2: .asciz "Hola\n" # Chars+\0
12 .bss # declaracion variables no initializadas
13 .global nom_var # variable global
14 #(visible desde ficheros externos)
15
16 # .comm nom_var, tamano, alineacion
17 .comm i, 4, 4  # int i;
18 .comm s, 2, 2  # short s;
19 .comm c, 1, 1  # char c;
20
21 .text # Seccion de codigo
22 .global main # main visible por el SO
23 main:
24 ...
25 # Finalizar el programa en Linux
26     movl $0, %ebx   # Ret %bl al SO
27     movl $1, %eax   # $1 = funcion sys_exit del SO
28     int $0x80   # Ejecuta la interrupcion
```

Capítulo 4

Vectores

4.1. Declaración y uso de vectores

Las declaraciones de vectores inicializados a un determinado valor se hacen en el bloque '.data' de la siguiente manera: "nombre_variable: .tipo valor". Ejemplo de declaración de vectores con su equivalencia en C:

```
1 .data
2 vl: .long 1,2,3      # int   vl[3] = {1,2,3};
3 vw: .word 1,2,3      # short vw[3] = {1,2,3};
4 vb: .byte 1,2,3      # char  vb[3] = {1,2,3};
5 s1: .ascii "abd"     # char  s1[3] = {'a','b','c'};
6 s2: .asciz "abd"     # char  s2[4] = {'a','b','c','\0'};
7 #.asciz guarda una posicion extra con el char '\0'
```

Las declaraciones de vectores no inicializados se hacen en el bloque '.bss' de la siguiente manera: .comm nombre_var,tamaño_vector,alineación. Si es un vector de longs lo alinearemos a 4 bytes, si es de words a 2 bytes y si es de chars a 1 byte.

Ejemplo de declaración de vectores con su equivalencia en C:

```
1 .bss
2 .comm uvl,3*4,4      # int uvl[3]
3 .comm uvw,3*2,2      # short uvw[3]
4 .comm uvb,3,1        # char uvb[3]
```

Si recuerdas, con el modo de direccionamiento indexado: D(Rb,Ri,S) el operando está en la posición de memoria resultante de calcular D+Rb+Ri*S. Por lo tanto si yo quiero acceder a la posición %ecx de un vector lo haré de la siguiente manera:

```
1 # Con long:
2 movl $0, v(,%ecx,4) # v[i] = 0;
```

```
3  # Y si %ebx apunta a v
4  movl $0, (%ebx,%ecx,4)
5  # Con word:
6  movl $0, v(,%ecx,2) # v[i] = 0;
7  # Y si %ebx apunta a v
8  movl $0, (%ebx,%ecx,2)
9  # Con char:
10 movb $'a', c(%ecx)  # c[i]='a';
11 # Y si %ebx apunta a c
12 movl $'a', (%ebx,%ecx)
```

Veamos ahora un pequeño ejemplo:

Código fuente 4.1: `t6_vectors.s`

```
1  .data
2  vw: .word 4,5,6 # short vw[3]={4,5,6};
3  .text
4  .global main
5  main:
6      movw $1, %cx              # i=1;
7      movswl %cx, %ebx          # Trampa!
8      # Rb & Ri tienen que ser de 32 bits!
9      movw %cx, vw(,%ebx,2)     # vw[i]=i;
10     # Ahora vw={4,1,6}
11     movl vw(,%ebx,2), %ebx
12
13     movl $1,%eax
14     int $0x80
```

```
$ gcc -g -o t6_vectors t6_vectors.s
$ ./t6_vectors
$ echo $?
1
```

TRAMPAS

- En direccionamiento de memoria Rb y Ri tienen que ser registros de 32 bits! Si en lugar de extender %cx a %ebx y luego referirnos a 'vw(,%ebx,2)' nos refirieramos a 'vw(,%cx,2)' nos daría un error

de compilación diciéndonos "Error: 'vw(,%cx,2)' no es una expresión base/índice válida":

```
$ gcc -g -o t6_vectors t6_vectors.s
t6_vectors.s: Assembleriviestit:
t6_vectors.s:9: Virhe: "vw(,%cx,2)" ei ole
 voimassa oleva "base/index"-lauseke
$
```

4.2. Recorrido de vector de bytes

Veamos ahora un ejemplo donde recorremos un vector de bytes declarado como cadena de caracteres '.ascii' (finalizada con '.') y retornamos al sistema cuantas 'a's hay en la cadena de caracteres.

Si 'v' es un vector de bytes: El equivalente en C de 'v[i]' es 'v(,%ecx,1)' que equivale a 'v(%ecx)'. Ya que 'v(,%ecx,1)' devuelve 'v+%ecx*1' o 'v+%ecx' (donde 'v' es la dirección de memoria donde empieza el vector).

Código fuente 4.2: `t6_recorrido_a.s`

```
1  # Retorna el numero de 'a's en v.
2  # ./t6_recorrido_a ; echo $?
3  # 7
4
5  .data
6  v: .ascii "Esto es una cadena acabada en punto."
7  n: .long 0        # int n = 0;
8
9  .text             # code segment
10 .global main
11 main:             # main() {
12     movl $0, %ecx  # for (i = 0; // ecx = i
13 for:
14     cmpb $'.', v(%ecx) # for(?;v[i]!='.';?) {
15     je endfor
16     cmpb $'a', v(%ecx) # if (v[i]=='a')
17     jne endif
18     incl n          # n++;
19 endif:
20     incl %ecx       # for (?;?;i++)
21     jmp for
22 endfor:
23
```

```
24      # LINUX EXIT
25      movl n, %ebx    # return n
26      movl $1, %eax
27      int $0x80
```

```
$ gcc -g -o t6_recorrido_a t6_recorrido_a.s
$ ./t6_recorrido_a
$ echo $?
7
$
```

EL CÓDIGO

- Línea 7: Usaremos la variable 'n' para almacenar el número de 'a's encontrado.
- Línea 12: Inicializamos %ecx a cero. %ecx se suele usar como registro para recorrer vectores (al igual que la variable 'i' en C). %esi también es una buena opción.
- Líneas 13-15: Empieza el bucle y saldremos del bucle cuando la posición %ecx del vector sea igual a '.'.
- Líneas 16-19: Si la posición %ecx del vector es igual a 'a' incrementamos el contador de 'a's 'n'.
- Líneas 20-21: Incrementamos %ecx y pasamos a la siguiente iteración del bucle.

4.3. Recorrido de vector de longs

Y aquí tenemos un ejemplo donde recorremos un vector de longs (finalizada en cero) y retornamos al sistema cuantas 'a's hay en la cadena de caracteres. Es muy similar al anterior. Las diferencias principales es que usamos el registro %eax para acumular la suma de los elementos recorridos (%eax es un buen registro para utilizar como acumulador) y que ahora el vector es de longs.

Si 'v' es un vector de longs: El equivalente en C de 'v[i]' es 'v(, %ecx,4)'. Ya que 'v(, %ecx,4)' devuelve 'v+ %ecx*4'(donde 'v' es la dirección de memoria donde empieza el vector).

Código fuente 4.3: t6_recorrido_i.s

```
1  # Retorna la suma de los elementos de un vector
       acabado en 0.
2  # ./t6_recorrido_i ; echo $?
3  # 45
4
5  .data
6  v: .int 1,2,3,4,5,6,7,8,9,0
7
8  .text
9  .global main
10 main:
11      movl $0, %eax
12      movl $0, %ecx   # for (i = 0;
13 for:
14      cmpl $0, v(,%ecx,4)      # for(?;v[i]!=0;?) {
15      je endfor
16      addl v(,%ecx,4), %eax    # a=a+v[i];
17      incl %ecx        # for (?;?;i++)
18      jmp for
19 endfor:
20
21      # LINUX EXIT
22      movl %eax, %ebx # return a
23      movl $1, %eax
24      int $0x80
```

```
$ gcc -g -o t6_recorrido_i t6_recorrido_i.s
$ ./t6_recorrido_i
$ echo $?
45
$
```

4.4. Búsqueda en vector de byte

Veamos ahora un ejemplo donde realizamos una búsqueda en un vector de bytes declarado como cadena de caracteres '.ascii' (finalizada con '.') y retornamos al sistema en que posición está la primera 'a' de la cadena de caracteres. Si no hubiera ninguna 'a' retornaríamos '-1'.

Código fuente 4.4: t6_busqueda_a.s

```
1  # Retorna la posicion de la primera 'a' en v.
2  # ./t6_busqueda_a ; echo $?
3  # 10
```

```
 4 # Si no hi hay ninguna 'a' retorna 255 (-1)
 5
 6 .data
 7 v: .ascii "Esto es una cadena acabada en punto."
 8 n: .long 0       # int n = 0;
 9
10 .text
11 .global main
12 main:
13     movl $0, %ecx    # for (i = 0; // ecx = i
14 for:
15     cmpb $'.', v(%ecx)  # for(?;v[i]!='.'
16     je endfor
17     cmpb $'a', v(%ecx)  # && v[i]!='a';?) {
18     je endfor
19     incl %ecx        # for (?;?;i++)
20     jmp for
21 endfor:
22     cmpb $'a', v(%ecx)  # if (v[i]=='a')
23     jne else
24     movl %ecx, n     # n = i;
25     jmp endif
26 else:
27     movl $-1, n      # n = -1; (255)
28 endif:
29
30     # LINUX EXIT
31     movl n, %ebx     # return n
32     movl $1, %eax
33     int $0x80
```

```
$ gcc -g -o t6_busqueda_a t6_busqueda_a.s
$ ./t6_busqueda_a
$ echo $?
10
$
```

EL CÓDIGO

- Líneas 14-21: Recorremos el vector hasta encontrar o una 'a' o el punto

del final de la cadena de caracteres. Al salir del bucle %ecx contiene la posición del vector donde está esa 'a' o '.'.

- Líneas 22-28: Si la posición %ecx del vector es igual a 'a' retornamos %ecx y sinó retornamos '-1'.

4.5. Búsqueda en vector de longs

Y, finalmente, un ejemplo de búsqueda en un vector de longs que devuelve la posición del primer número negativo en un vector 'v' acabado en cero; o '-1' si no hubiese ningún número negativo en el vector.

Código fuente 4.5: `t6_busqueda_i.s`

```
1  # Retorna la posicion del primer negativo en v.
2  # ./t6_busqueda_i ; echo $?
3  # 5
4  # Si no hay ningun neg retorna 255 (-1)
5
6  .data
7  v: .int 1,2,3,4,5,-6,7,8,9,0
8
9  .text
10 .global main
11 main:
12     movl $0, %ecx   # for (i = 0; // ecx = i
13 for:
14     cmpl $0, v(,%ecx,4)      # for(?;v[i]!=0
15     je endfor
16     cmpl $0, v(,%ecx,4)      # && v[i]>=0;?) {
17     jl endfor
18     incl %ecx        # for (?;?;i++)
19     jmp for
20 endfor:
21     cmpl $0, v(,%ecx,4)      # if (v[i]<0)
22     jge else
23     movl %ecx, %eax # a = i;
24     jmp endif
25 else:
26     movl $-1, %eax  # a = -1; (255)
27 endif:
28
29     # LINUX EXIT
30     movl %eax, %ebx # return a
```

```
31      movl $1, %eax
32      int $0x80
```

```
$ gcc -g -o t6_busqueda_i t6_busqueda_i.s
$ ./t6_busqueda_i
$ echo $?
5
$
```

4.6. Vector de words

Vamos a hacer un ejemplos con un vector de words. Ya habréis adivinado que los vectores de words van iguales que los de longs cambiando el ',4)' por ',2)'. Este ejemplo en concreto forma parte de la colección de problemas de la asignatura (problema 3).

El ejemplo recorre un vector 'v' de words de 'N' posiciones y almacena en las variables 'min' y 'max' el entero mínimo y el entero máximo del vector. Usa la constante 'N' para indicar el tamaño del vector 'v'.

Como aún no sabemos invocar funciones, no podemos hacer un 'printf' con el valor de 'min' y el de 'max'. Así que el código solo puede devolver uno de los dos valores al sistema (en el ejemplo devolvemos 'max'). No os preocupéis en el siguiente capitulo aprenderemos a hacer 'printf' y 'scanf'

Código fuente 4.6: `t6_p03.s`

```
1  N = 10
2  .data
3  v: .word 1,2,3,4,5,6,7,8,9,-1
4  .bss
5  .comm min,2,2
6  .comm max,2,2
7  .text
8  .global main
9  main:
10      movl $0, %ecx
11      movw v(,%ecx,2), %dx
12      movw %dx, min
13      movw %dx, max
14      movl $1, %ecx
15      movw v(,%ecx,2), %dx
16 for:
17      # %ecx>=$N -> endfor
18      cmpl $N, %ecx
19      jge endfor
20
```

```
21      # %dx<=max -> endifmax
22      cmpw max, %dx
23      jle endifmax
24      movw %dx, max
25  endifmax:
26
27      # %dx>=min -> endifmin
28      cmpw min, %dx
29      jge endifmin
30      movw %dx, min
31  endifmin:
32
33      incl %ecx
34      movw v(,%ecx,2), %dx
35      jmp for
36  endfor:
37
38      movswl max, %ebx
39      movl $1, %eax
40      int $0x80
```

```
$ gcc -g -o t6_p03 t6_p03.s
$ ./t6_p03
$ echo $?
9
$
```

EL CÓDIGO

- Línea 1: La constante 'N' nos indica la longitud del vector 'v'.
- Líneas 3-6: El vector 'v' se declara como vector de words y las variables 'min' y 'max' como words.
- Líneas 7-13: Se asigna la primera posición del vector a 'min' y a 'max'.
- Líneas 14-35: Se recorre el resto del vector. Por cada posición del vector, si su valor es menor que 'min' (el valor mínimo de la parte del vector que ya hemos recorrido) copiamos esa posición del vector a 'min'; y si su valor es mayor que 'max' copiamos esa posición a 'max'.

TRAMPAS

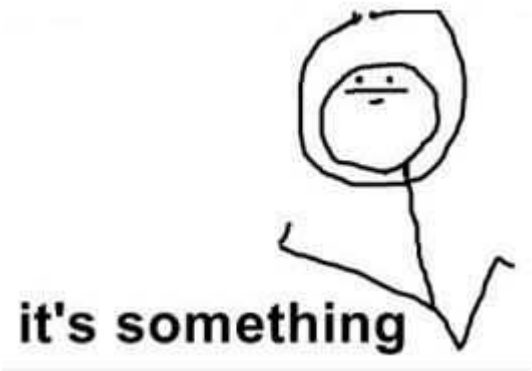
- Líneas 11-13: No podemos escribir 'movw v(,%ecx,2), min' (que es lo que nos pediría el cuerpo) porque una instrucción sólo puede tener un operando en memoria (y las variables están en memoria). Por eso tenemos que usar un registro auxiliar (%dx es un buen registro para almacenar un dato). Si en la línea 11 tuviéramos 'movw v(,%ecx,2), min' nos daría un error de compilación diciéndonos "Error: demasiadas referencias a memoria":

```
gcc -g -o t6_p03 t6_p03.s
t6_p03.s: Assembleriviestit:
t6_p03.s:11: Virhe: liian monta muistiviittausta
 mallinteeseen "mov"
$ LANG=C gcc -g -o t6_p03 t6_p03.s
t6_p03.s: Assembler messages:
t6_p03.s:11: Error: too many memory references
 for 'mov'
$
```

- Líneas 21 y 27: Lo mismo sucede con las comparaciones ('cmpw').

EL BOXEO

- El ensamblador es como el boxeo porque todo va al revés.
  Excepto en el funcionamiento de los vectores donde, como habéis podido ver, todo funciona como esperaríamos.



4.7. Ejemplos más complejos

A continuación cuatro ejemplos de programas con vectores. Sólo os doy el código y lo que hacen. Estos programas de ejemplo tienen una serie de bloques que empiezan con '# BEGIN DEBUG' y terminan con '# END DEBUG'. Estos bloques lo que hacen es 'printf' del resultado final. De momento los podéis ignorar. Los he puesto para que podáis comprobar que los programas funcionan correctamente.

4.7.1. Ejemplo 1

Dado una secuencia de caracteres 'iv' que contiene palabras separadas por espacio (asumiendo que no hay dobles espacios ni signos de puntuación) y una secuencia de caracteres no inicializada, copiar la tercera palabra de 'iv' a 'ov'. Asumimos que como mínimo hay cuatro palabras.

Código fuente 4.7: `t6_foob.s`

```
1 # Copiar la tercera palabra de 'iv' a 'ov'.
```

```
 2 # Asumimos que no hay dobles espacios ni signos de
       puntuacion.
 3 # 'iv' declarada como inicializada. El 'ov' como no
       inicializado.
 4 SIZE = 100
 5 .data
 6 iv: .asciz "Uno dos tres quatro cinco seis siete ocho
       nueve diez onze"
 7 # BEGIN DEBUG
 8 s2: .asciz "[%c]"
 9 s: .asciz "%c"
10 sr: .asciz "\n"
11 # END DEBUG
12 .bss
13 .comm ov,SIZE,1
14 .text
15 .global main
16 main:
17  movl $1, %ebx # numero de palabra
18  movl $0, %esi
19 for0:
20  cmpl $3, %ebx
21  je endfor0 # %ebx == $3
22  cmpb $' ', iv(%esi)
23  jne no_space # iv(%esi) != $' '
24  incl %ebx
25 no_space:
26  incl %esi
27  jmp for0
28 endfor0:
29
30  movl $0, %edi
31 for:
32  cmpb $' ', iv(%esi)
33  je endfor # iv(%esi) == $' '
34  movb iv(%esi), %dl
35  movb %dl, ov(%edi)
36  incl %esi
37  incl %edi
38  jmp for
39 endfor:
40
41 # BEGIN DEBUG
42  movb $0, ov(%edi) # Terminar ov con '\0'
43
44  movl $0, %edi
```

```
45  for_printf:
46   cmpb $0, ov(%edi)
47   je endfor_printf # oc(%edi) == 0
48
49   pushl ov(%edi)
50   pushl $s
51   call printf
52   addl $8, %esp
53
54   incl %edi
55   jmp for_printf
56  endfor_printf:
57
58   pushl $sr
59   call printf
60   addl $4, %esp
61
62  # END DEBUG
63
64   movl $0, %ebx
65   movl $1, %eax
66   int $0x80
```

```
$ gcc -g -o t6_foob t6_foob.s
$ ./t6_foob
tres
$
```

4.7.2. Ejemplo 2

Dado un vector 'v' de enteros y una variable entera 'n' que indica cuantos elementos tiene el vector, contad cuantos ceros, positivos y negativos hay en el vector.

Código fuente 4.8: `t6_fool.s`

```
1  # Cuantos ceros, positivos y negativos hay en el
       vector 'v' de 'n' elementos?
2  # 'v' y 'n' declaradas como inicializadas. el resto
       como no inicializadas.
3  .data
4  n: .long 8
5  v: .long 7,9,2,3,-4,-5,-6,0
6  # BEGIN DEBUG
7  s: .asciz "Ceros:%d. Positivos:%d. Negativos:%d.\n"
8  # END DEBUG
```

```
 9  .bss
10  .comm cero,4,4
11  .comm posi,4,4
12  .comm nega,4,4
13  .text
14  .global main
15  main:
16   movl $0, cero
17   movl $0, posi
18   movl $0, nega
19   movl $0, %ecx
20  etiq_for:
21   cmpl n, %ecx
22   jge etiq_endfor # %ecx >= n
23
24   cmpl $0, v(,%ecx,4)
25  # je etiq_cero # == 0
26   jg etiq_posi # > 0
27   jl etiq_nega # < 0
28  etiq_cero:
29   incl cero
30   jmp etiq_forinc
31  etiq_posi:
32   incl posi
33   jmp etiq_forinc
34  etiq_nega:
35   incl nega
36  # jmp etiq_forinc
37  etiq_forinc:
38   incl %ecx
39   jmp etiq_for
40  etiq_endfor:
41
42  # BEGIN DEBUG
43   pushl %ecx
44   pushl nega
45   pushl posi
46   pushl cero
47   pushl $s
48   call printf
49   addl $16, %esp
50   popl %ecx
51  # END DEBUG
52
53   movl $0, %ebx
54   movl $1, %eax
```

```
55    int $0x80
```

```
$ gcc -g -o t6_fool t6_fool.s
$ ./t6_fool
Ceros:1. Positivos:4. Negativos:3.
$
```

4.7.3. Ejemplo 3

Dado una secuencia de caracteres 'iv' que contiene palabras separadas por espacio y una secuencia de caracteres no inicializada, copiar 'iv' a 'ov' cambiando ' 's por '_'s.

Código fuente 4.9: `t6_barb.s`

```
 1  # Copiar 'iv' a 'ov' cambiando '␣'s por '_'s.
 2  # 'iv' declarada como inicializada. El 'ov' como no
       inicializado.
 3  SIZE = 100
 4  .data
 5  iv: .asciz "Hay␣mucho␣espacio␣en␣blanco␣por␣aqui."
 6  # BEGIN DEBUG
 7  s: .asciz "%c"
 8  sr: .asciz "\n"
 9  # END DEBUG
10  .bss
11  .comm ov,SIZE,1
12  .text
13  .global main
14  main:
15   movl $0, %ecx
16  for:
17
18   cmpb $0, iv(%ecx)
19   je endfor # iv(%ecx) == $0
20
21   cmpb $'␣', iv(%ecx)
22   je else # iv(%ecx) == $'␣'
23   movb iv(%ecx), %al
24   movb %al, ov(%ecx)
25   jmp endif
26  else:
27   movl $'_', ov(%ecx)
28  endif:
29
30   incl %ecx
```

```
31   jmp for
32  endfor:
33
34  # BEGIN DEBUG
35   movl $0, %ecx
36  for_printf:
37   cmpb $0, ov(%ecx)
38   je endfor_printf # oc(%ecx) == 0
39
40   pushl %ecx
41   pushl ov(%ecx)
42   pushl $s
43   call printf
44   addl $8, %esp
45   popl %ecx
46
47   incl %ecx
48   jmp for_printf
49  endfor_printf:
50
51   pushl $sr
52   call printf
53   addl $4, %esp
54
55  # END DEBUG
56
57   movl $0, %ebx
58   movl $1, %eax
59   int $0x80
```

```
$ gcc -g -o t6_barb t6_barb.s
$ ./t6_barb
Hay_mucho_espacio_en_blanco_por_aqui.
$
```

4.7.4. Ejemplo 4

Dada una variable entera 'n' con valor entre '0' y '20' hacer un programa que imprima por pantalla los primeros 'n' elementos de la sucesión de Fibonacci. En el ejemplo 'n' vale '20' pero funciona con cualquier 'n' entre '0' y '20'.

La sucesión de Fibonacci es tal que f[0]=0, f[1]=1, y para i ¿=1, f[i]=f[i-1]+f[i-2].

Código fuente 4.10: `t6_barl.s`

```
1 # Fibonacci en un vector de 'n' elementos.
```

```
2  # 'n' declarada como inicializada. El vector como no
       inicializado.
3  SIZE = 20
4  .data
5  n: .long 20
6  # BEGIN DEBUG
7  s: .asciz "%d\n"
8  # END DEBUG
9  .bss
10 .comm v,SIZE*4,4
11 .text
12 .global main
13 main:
14  cmpl $1, n
15  jl end # n < 1
16  movl $0, %ecx
17  movl $0, v(,%ecx,4)
18
19  cmpl $2, n
20  jl end # n < 2
21  movl $1, %ecx
22  movl $1, v(,%ecx,4)
23
24  movl $0, %esi
25  movl $1, %edi
26  movl $2, %ecx
27 for:
28  cmpl n, %ecx
29  jg endfor # %ecx > n
30
31  movl v(,%esi,4), %eax
32  addl v(,%edi,4), %eax
33  movl %eax, v(,%ecx,4)
34  incl %esi
35  incl %edi
36  incl %ecx
37  jmp for
38 endfor:
39
40 end:
41 # BEGIN DEBUG
42  movl $0, %ecx
43 for_printf:
44  cmpl n, %ecx
45  jge endfor_printf # %ecx >= n
46
```

```
47    pushl %ecx
48    pushl v(,%ecx,4)
49    pushl $s
50    call printf
51    addl $8, %esp
52    popl %ecx
53
54    incl %ecx
55    jmp for_printf
56  endfor_printf:
57  # END DEBUG
58
59    movl $0, %ebx
60    movl $1, %eax
61    int $0x80
```

```
$ gcc -g -o t6_barl t6_barl.s
$ ./t6_barl
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
$
```

Capítulo 5

Subrutinas

5.1. La pila (stack)

La pila (o 'stack' en inglés) es una pila de bytes LIFO: Last In, First Out.
La pila tiene asociado un puntero a su cima (%esp) que se inicializa por el Sistema Operativo cuando carga el programa en memoria.
La pila crece en sentido decreciente de las direcciones de memoria (anti-intuitivo).
Por lo tanto, la cima tiene una posición de memoria inferior a la base.
En las arquitecturas de 32 bits apilaremos y desapilaremos longs.
Podemos pensar en la pila como en una pila de libros que tenemos encima de la mesa. Si apilo un libro siempre lo apilaré en su cima. Y, si desapilo un libro, siempre será el libro que estaba encima de todo (el que estaba en la cima).

Para apilar el long 'op1' haremos:

```
1 pushl op1
```

Equivale a:

```
1 subl $4, %esp
2 movl op1, (%esp)
```

Para desapilar el long que está en la cima de la pila y guardar ese valor en 'op1' haremos:

```
1 popl op1
```

Equivale a:

```
1 movl (%esp), op1
2 addl $4, %esp
```

Aquí vemos una pila ("@-" y "@-" sirven para recordarnos de que las posiciones de la pila de más "arriba" tienen direcciones de memoria menores a las posiciones de más "abajo"). Tiene un único long (que vale 1) y %esp (que siempre apunta a la cima de la pila) apunta a ese elemento. Si la dirección de memoria donde está su único elemento fuera la 1000, la posición inmediatamente superior sería la 996, y la siguiente 992, ya que cada posición de la pila es un long (4 bytes).

@-		
@+	0x00000001	← %esp

Si %eax vale 0x12345678 y lo apilamos con:

```
1 pushl %eax
```

La pila pasará a tener una copia de %eax y %esp se habrá decrementado en 4 porqué un long ocupa 4 bytes:

@-		
	0x12345678	← %esp
@+	0x00000001	4(%esp)

Si queremos referirnos al valor de la cima podemos hacerlo con "(%esp)". Y si queremos referirnos al long que está debajo del long de la cima podemos hacerlo con "4(%esp)" (ya que está en la posición de memoria 4+ %esp).

Si %ebx vale 0xFFFFFFFF y lo apilamos con:

```
1 pushl %ebx
```

La pila pasará a tener una copia de %ebx encima de la copia de %eax y, de nuevo, %esp se actualizará decrementándose en 4:

@-		-4(%esp)
	0xFFFFFFFF	← %esp
	0x12345678	4(%esp)
@+	0x00000001	8(%esp)

Si queremos desapilar el elemento de la cima y guardarlo en %ecx lo haremos con:

```
1 popl %ecx
```

Con esto, se desapilará el elemento que tiene en la cima (0xFFFFFFFF) y se asignará a %ecx y, %esp se actualizará incrementándose en 4:

@-		-8(%esp)
		-4(%esp)
	0x12345678	← %esp
@+	0x00000001	4(%esp)

Si queremos desapilar el elemento de la cima sin guardarlo en ninguna parte podemos hacerlo manipulando directamente el registro %esp. Desapilar un long equivale a sumar 4 a %esp:

```
1 addl $4, %esp
```

Con esto, se desapilará el elemento que tiene en la cima:

@-		-12(%esp)
		-8(%esp)
		-4(%esp)
@+	0x00000001	← %esp4

TEORÍA

- RESUMEN DE LA PILA
- Para apilar el long 'op1':

```
1 pushl op1
```

- Para desapilar un long y guardarlo en 'op1':

```
1 popl op1
```

Desapilar el long de la cima sin guardarlo:

```
1 addl $4, %esp
```

- %esp Siempre apunta al elemento que está en la cima de la pila.

EL BOXEO

- El ensamblador es como el boxeo porque todo va al revés:
  La dirección de memoria de la cima de la pila es más pequeña que la dirección de la base. Y, para reducir el tamaño de la pila le sumas al "stack pointer" el número de bytes que quieres desapilar y descartar.

5.2. Invocar a una subrutina

Para invocar una subrutina que ya existe, (como por ejemplo "printf") si en C hacíamos

```
1 printf("Hello World!\n");
```

En ensamblador haremos:

- push de los parámetros empezando por el de la derecha (y si un parámetro es un vector, pasaremos el puntero al vector).
- call subrutina
- Si la subrutina retorna algún valor lo dejará en %eax.
- Eliminar los parámetros de la pila (equivalente a hacer un pop pero el parámetro no se guarda en ningún registro).

Código fuente 5.1: `t7_ex_printf.s`

```
1 .data
2 s: .asciz "Hello World!\n"
3 .text
4 .global main
5 main:
6     # Pasar param (puntero a s)
7     pushl $s
```

```
 8      # Llamar subrutina
 9      call printf
10      # Eliminar param de la pila
11      addl $4, %esp
12
13      movl $1, %eax
14      movl $0, %ebx
15      int $0x80
```

```
$ ./t7_ex_printf
Hello World!
$
```

TRAMPAS

- El string de printf() tiene que acabar en '\n' o puede que no aparezca por pantalla (a diferencia de en C).

EL BOXEO

- El ensamblador es como el boxeo porque todo va al revés:
  El último parámetro de una llamada a subrutina se apila primero y el primer parámetro se apila el último.

5.3. La diferencia entre call y jmp

Para llamar a una subrutina 'etiqueta' haremos:

```
1 call etiqueta
```

Esto equivale a:

```
1 pushl %eip
2 movl etiqueta, %eip
```

Ya que %eip siempre apunta a la siguiente instrucción a ejecutar. Por lo tanto, lo que hacemos es guardar la dirección de memoria donde está la instrucción de después del 'call' (la instrucción a ejecutar cuando volvamos de la subrutina) y hacer que la siguiente instrucción a ejecutar sea la primera de la subrutina.
Para retornar de la subrutina haremos:

```
1 ret
```

Que equivale a:

```
1 popl %eip
```

Con esto actualizamos %eip con la llamada dirección de retorno.
El registro %eip no se puede modificar directamente. De forma que las llamada a una subrutina son saltos al código de esa subrutina después de la cual podemos volver a la linea siguiente de la que ha llamado a la subrutina.
Por cierto, los parámetros (que se pasan de derecha a izquierda) se guardan en direcciones de memoria alineadas a 4 bytes. Si un parámetro es un char o un word, primero se convierte en long y luego se apila con 'pushl'. Ejemplo de una llamada a "subrutina('a')":

```
1 movb $'a', %al
2 movzbl %al, %eax
3 pushl %eax
4 call subrutina
```

El valor de retorno de una función se deja en %eax, %ax, o %al dependiendo de si la función devuelve un long un word o un byte.

5.4. Registros a salvar antes de invocar una subrutina

Si no queremos que la subrutina que invocamos nos modifique los valores de los registros %eax, %ecx y %edx deberemos guardar una copia en la pila antes de hacer la llamada. Después de la llamada restauramos los valores originales. La secüencia completa sería:

- **push de %eax, %ecx, %edx.**
- push de los parámetros empezando por la derecha (si el parámetro es un vector, se pasa el puntero al vector).
- call de la subrutina.
- Si la subrutina retorna algo se deja en %eax.
- Eliminar los parámetros de la pila (equivalente a hacer un pop peró el parámetro no se guarda en ningún registro).

- **pop de %edx, %ecx, %eax. (fijaos que es en orden inverso que el push)**

Para recordar los nombres de estos tres registros y el orden en el que se suelen apilar, yo los llamo "los registros rocanroleros": ACDC (%eax, %ecx, %edx).

5.5. printf y scanf

A continuación podemos ver el exemplo t7_printf.s que imprime por pantalla todos los elementos del vector i' acabado en '0' (excepto el '0').

Código fuente 5.2: `t7_printf.s`

```
1  .data
2  i: .int 21, 15, 34, 11, 6, 50, 32, 80, 10, 0 # Acabado
       en 0
3  s: .asciz "El numero es: %d\n" # .asciz es como .ascii
       pero acabado en '\0'
4  .text
5  .globl main
6  main:
7      movl $0, %ecx
8  loop:
9      pushl %ecx              # 1 Salvar eax,ecx,edx
10     pushl i(,%ecx,4)        # 2 Pasar params
11     pushl $s                # 2 Pasar params
12     call printf             # 3 Llamar subrutina
13                             # 12 Resultado en eax
14     addl $8, %esp           # 13 Eliminar params
15     popl %ecx               # 14 Restaurar eax,ecx,edx
16     incl %ecx
17     cmpl $0, i(,%ecx,4)
18     jne loop
19
20     movl $1, %eax
21     movl $0, %ebx
22     int $0x80
```

EL CÓDIGO

- En este ejemplo, como de los registros rocanroleros sólo nos importa %ecx (%eax y %edx no los usamos y nos da igual si la

subrutina invocada los modifica) este es el único que apilamos y recuperamos en las lineas 9 y 15.

- La numeración del 1 al 3 y del 12 al 14 corresponde a la numeración de todos los pasos de una llamada a subrutina que utilizamos en clase. Más adelante veremos los pasos que faltan (del 4 al 11, que son los que se ejecutan dentro de la subrutina).

```
$ gcc -g -o t7_printf t7_printf.s
$ ./t7_printf
El numero es: 21
El numero es: 15
El numero es: 34
El numero es: 11
El numero es: 6
El numero es: 50
El numero es: 32
El numero es: 80
El numero es: 10
$
```

Veamos ahora otro ejemplo un poco más completo donde imprimimos el indice del vector junto con el valor contenido en esa posición:

Código fuente 5.3: `t7_printf2.s`

```
 1 .data
 2 i: .int 21, 15, 34, 11, 6, 50, 32, 80, 10, 0 # Acabado
       en 0
 3 s: .asciz "i[%d] = %d\n" # .asciz es como .ascii pero
      acabado en '\0'
 4 .text
 5 .globl main
 6 main:
 7     movl $0, %ecx
 8 loop:
 9     pushl %ecx              # 1 Salvar eax,ecx,edx
10     pushl i(,%ecx,4)        # 2 Pasar params
11     pushl %ecx              # 2 Pasar params
12     pushl $s                # 2 Pasar params
13     call printf             # 3 Llamar subrutina
14                             # 12 Resultado en eax
15     addl $12, %esp          # 13 Eliminar params
16     popl %ecx               # 14 Restaurar eax,ecx,edx
```

```
17      incl %ecx
18      cmpl $0, i(,%ecx,4)
19      jne loop
20
21      movl $1, %eax
22      movl $0, %ebx
23      int $0x80
```

```
$ gcc -g -o t7_printf2 t7_printf2.s
$ ./t7_printf2
i[0] = 21
i[1] = 15
i[2] = 34
i[3] = 11
i[4] = 6
i[5] = 50
i[6] = 32
i[7] = 80
i[8] = 10
$
```

Igual que invocamos printf podemos invocar scanf:

Código fuente 5.4: `t7_scanf.s`

```
 1  .data
 2  s: .asciz "Introduce un num:"
 3  is: .asciz "%d"
 4  os: .asciz "El num es: %d. Scanf retorna %d.\n"
 5  .bss
 6  .comm n,4,4
 7  .text
 8  .global main
 9  main:
10      #printf(s);
11      pushl $s
12      call printf
13      addl $4, %esp
14
15      #scanf(is, &n);
16      pushl $n
17      pushl $is
18      call scanf
19      addl $8, %esp
20
21      #printf(os,n,eax)
22      pushl %eax
```

```
23      pushl n
24      pushl $os
25      call printf
26      addl $12, %esp
27
28      movl $0, %ebx
29      movl $1, %eax
30      int $0x80
```

```
$ gcc -g -o t7_scanf t7_scanf.s
$ ./t7_scanf
Introduce un num :7
El num es: 7. Scanf retorna 1.
$
```

EL CÓDIGO

- Fijaos que en la línea 16 hacemos un pushl de '$n' en lugar de 'n' porqué el parámetro es un puntero a n ('&n' en C), es decir su posición de memoria.
- De la misma manera, en la lína 17 hacemos un pushl de '$is' en lugar de 'is' porqué los strings son vectores de carácteres, y los vectores se pasan como parámetro pasando la dirección de memória donde empiezan.

Veamos ahora otro ejemplo un poco más completo donde leemos dos enteros:

Código fuente 5.5: `t7_scanf_2.s`

```
1  .data
2  sp: .asciz "Intro 2 nums: "
3  ss: .asciz "%d %d"
4  sp2: .asciz "Los nums son: %d y %d. Scanf retorna %d.\
       n"
5  .bss
6  .comm n1,4,4
7  .comm n2,4,4
8  .comm ret,4,4
9  .text
10 .globl main
```

```
11 main:
12     pushl $sp
13     call printf
14     addl $4, %esp
15
16     pushl $n2
17     pushl $n1
18     pushl $ss
19     call scanf
20     addl $12, %esp
21     movl %eax, ret
22
23     pushl ret
24     pushl n2
25     pushl n1
26     pushl $sp2
27     call printf
28     addl $16, %esp
29
30     movl $0, %ebx
31     movl $1, %eax
32     int $0x80
```

```
$ gcc -g -o t7_scanf_2 t7_scanf_2.s
$ ./t7_scanf_2
Intro 2 nums: 3 7
Los nums son: 3 y 7. Scanf retorna 2.
$
```

5.6. Código de una subrutina

Hasta ahora hemos estado invocando subrutinas ya existentes. Si la subrutina la implementamos nosotros el código que tendremos que escribir será:

- La etiqueta con el nombre de la función
- El Establecimiento del enlace dinámico, que consiste en apilar %ebp y mover el valor de %esp a %ebp. Este %ebp indica la base de la pila de esta subrutina y no se modificará hasta que se termine la subrutina (linea 52). %esp se puede modificar si apilo más elementos.

```
1 subrutina:
2     pushl %ebp
3     movl %esp, %ebp
```

De manera que una vez establecido el enlace dinámico el registro %ebp contiene una posición de memoria de referencia de esta subrutina, donde:

- (%ebp) Apunta al %ebp de la subrutina que nos ha invocado
- 4(%ebp) Apunta a la dirección a la que debemos retornar al terminar la subrutina
- 8(%ebp) Apunta al primer parámetro (de existir)
- 12(%ebp) Apunta al segundo parámetro (de existir)
- Etcétera.

- A continuación viene el cuerpo de la subrutina.
- Y, al terminar la ejecución de la subrutina deberemos mover el resultado que devuelve la subrutina a %eax, deshacer el enlace y retornar (recuperar la dirección de retorno y moverla a %eip).

```
1     popl %ebp # deshacer el enlace
2     ret #popl %eip
```

Veamos un ejemplo donde tenemos una función llamada 'minuscula' que cómo parámetro de entrada tiene un carácter y lo que hace es, en caso de que se tratara de una mayúscula nos devuelve el carácter en minúsculas, en caso contrario nos devuelve el carácter original. Esta función la utilizaremos para pasar todos los caracteres de una secuencia de caracteres (o string) a minúsculas.

Código fuente 5.6: t7_minusculas_print_2.s

```
1  SIZE = 100
2  .data
3  is: .asciz "Demasiadas Mayusculas PARA TAN pOcAs
       BalAs.\n"
4  .bss
5  .comm os,SIZE,1
6  .text
7  .global main
8  main:
9       pushl $is
10      call printf
11      addl $4, %esp
12
13      movl $0, %ecx
14 for:
15      # is(%ecx)==$0
16      cmpb $0, is(%ecx)
17      je endfor
18      pushl %ecx
19      movsbl is(%ecx), %edx
```

```
20      pushl %edx
21      call minuscula
22      addl $4, %esp
23      popl %ecx
24      movb %al, os(%ecx)
25      incl %ecx
26      jmp for
27  endfor:
28      movb $0, os(%ecx)
29
30      pushl $os
31      call printf
32      addl $4, %esp
33
34      movl $0, %ebx
35      movl $1, %eax
36      int $0x80
37
38  minuscula:
39      pushl %ebp
40      movl %esp, %ebp
41
42      movb 8(%ebp), %al
43      # %al < $'A'
44      cmpb $'A', %al
45      jl minendif
46      # %al > $'Z'
47      cmpb $'Z', %al
48      jg minendif
49      addb $'a'-'A', %al
50
51  minendif:
52      popl %ebp
53      ret
```

```
$ gcc -g -o t7_minusculas_print_2 t7_minusculas_print_2.s
$ ./t7_minusculas_print_2
Demasiadas Mayusculas PARA TAN pOcAs BalAs.
demasiadas mayusculas para tan pocas balas.
$
```

EL CÓDIGO

- linea 18: Empieza la llamada a la subrutina 'minuscula'. Para ello apilamos los registros %eax, %ecx y %edx si es que no queremos perder el valor que contienen. En este caso sólo %ecx.
- linea 19: El parámetro con el que invocamos la subrutina es de tipo byte. Por ello lo pasamos a long con 'movsbl'.
- linea 20: Apilamos el parámetro ya convertido a long.
- linea 21: Llamamos a la subrutina
- linea 22: Al volver, desapilamos el parámetro.
- linea 23: Recuperamos el valor de %ecx que habíamos guardado en la pila.
- linea 24: El resultado de la función siempre se guarda en %eax. Esta vez como 'minuscula' retorna un byte, el resultado (el carácter pasado a minúsculas) está en %al.
- linea 38: La subrutina 'minuscula'. Las subrutinas siempre empiezan con las siguientes dos lineas y siempre terminan con las dos últimas (lineas 52 y 53).
- linea 39: Apilo (guardo una copia de) %ebp
- linea 39: Muevo el valor de %esp a %ebp.
- lineas 42-51: El cuerpo de la subrutina.
- linea 52: Recupero el valor de %ebp (la base de la pila de la subrutina que me ha llamado).
- linea 53: Retorno.

TRAMPAS

- linea 3: La cadena de caracteres que se pasan a 'printf()' tiene que

acabar en '\n' o puede que no aparezca por pantalla (a diferencia de en C).

- lineas 19 y 20: La manera elegante de hacer un 'pushl' de un byte consiste en primero convertir el byte a long con 'movsbl' o 'movzbl' (dependiendo de si el byte es con o sin signo) y luego hacer el 'pushl' de ese long.
- lineas 23 y 24: Fijaros que tenemos que restaurar los registros rocanroleros cuyo valor nos importa (%ecx en este caso) antes de usar sus valores. Habría sido un error haber intercambiado estas dos líneas de código.
- linea 28: Salimos del 'for' cuando hemos copiado todos los caracteres de 'is' a 'os' y 'is(%ecx)' es cero. Pero ese cero no lo hemos movido a 'os(%ecx)'. Por eso lo hacemos aquí. Recordad que el cero sirve de marca de final de vector de caracteres.
- linea 49: Hay que ir con cuidado con esta suma para que no se produzcan desbordamientos. En este caso 'a'-'A' es 32 y %al es una minúscula con lo cual no hay ningún problema.

TRUCOS

- La función 'minuscula' retorna un byte que dejamos en %al. Es por eso que solo modificamos %al sin preocuparnos de lo que haya en el resto de %eax. No obstante, puede ser una buena costumbre asegurarse de que el resto de %eax esté a cero. Esto es útil de cara a debugar (donde al ver los registros veremos %eax) o para evitarse errores. En este código una opción sería cambiar la linea 48 el 'movb' por un 'movzbl':

```
42  movzbl 8(%ebp), %eax
```

5.7. Salvar registros %ebx, %esi, %edi

De la misma manera que antes de llamar a una subrutina hay que guardar en la pila los valores de los "registros rocanroleros" (%eax, %ecx y %edx) que no queremos que sean modificados porque los estamos usando, dentro de la función,

justo después de establecer el enlace dinámico hay que guardar en la pila los registros que denominaremos "registros base e índices" (%ebx, %esi, %edi) que modifiquemos dentro de la función. A falta de un mnemotécnico mejor para los "registros base e índices" podemos pensar en cómo pronunciarías BDC en inglés (bi, si, di).
El código completo de la subrutina sería el siguiente:

- La etiqueta con el nombre de la función y el Establecimiento del enlace dinámico (que consiste en apilar %ebp y mover el valor de %esp a %ebp).

```
1 subrutina:
2     pushl %ebp
3     movl %esp, %ebp
```

  De manera que una vez establecido el enlace dinámico el registro %ebp contiene una posición de memoria de referencia de esta subrutina, donde:

  - (%ebp) Apunta al %ebp de la subrutina que nos ha invocado
  - 4(%ebp) Apunta a la dirección a la que debemos retornar al terminar la subrutina
  - 8(%ebp) Apunta al primer parámetro (de existir)
  - 12(%ebp) Apunta al segundo parámetro (de existir)
  - Etcétera.

- **Salvar registros %ebx, %esi, %edi**: Guardamos en la pila una copia de los "registros índice y bases" que se modifican en esta subrutina.
- A continuación viene el cuerpo de la subrutina.
- Y, al terminar la ejecución de la subrutina deberemos mover el resultado que devuelve la subrutina a %eax
- **Restaurar registros %edi, %esi, %ebx**: Restauramos de la pila la copia de los "registros índice y bases" que se han modificado en esta subrutina.
- Deshacer el enlace y retornar (recuperar la dirección de retorno y moverla a %eip).

```
1     popl %ebp # deshacer el enlace
2     ret #popl %eip
```

5.8. Subrutinas con variables locales

Finalmente, para terminar de complicarlo todo. Vamos a ver como implementaríamos una subrutina que tuviera variables locales.
Pues bien justo después de establecer el enlace dinámico, y antes de salvar los

registros base e índice, reservaremos espacio en la pila para las variables locales. Esto lo haremos restando a %esp el número de bytes (alineado a 4 bytes) que necesitamos. Si las variables locales son tres longs (12 bytes en total), haremos:

```
1    subl $12, %esp  # 12 bits para variables locales
```

Una vez hecho esto para acceder a las 3 variables locales podemos usar '-4(%ebp)', '-8(%ebp)' y '-12(%ebp)'. Si por ejemplo, quisiéramos inicializarlas a cero haríamos:

```
1    movl $0, -4(%ebp)    # Inicializar VarLoc1 a 0
2    movl $0, -8(%ebp)    # Inicializar VarLoc2 a 0
3    movl $0, -12(%ebp)   # Inicializar VarLoc3 a 0
```

Finalmente, justo después de restaurar los registros base e índices, y antes de deshacer el enlace dinámico restaurando el "base pointer" con 'pop %ebp', desharemos la reserva del espacio para variables locales:

```
1    addl $12, %esp  # 12 bits para variables locales
```

Esto es equivalente a hacer:

```
1    movl %ebp, %esp # Restaurar %esp
```

Son equivalentes por que justo antes de hacer la reserva de espacio para variables locales, hemos establecido el enlace dinámico con 'movl %esp, %ebp'.
Utilizad la alternativa que os parezca más intuitiva.

Vamos a ver un ejemplo para que aclararnos un poco. El problema de intentar crear un ejemplo simple de programa con variables locales, es que intentando hacerlo lo más simple posible uno termina haciendo un ejemplo con una única variable local que se podría haber implementado utilizando un registro en lugar de la variable local. Con el añadido que lo más probable es que el código tenga que ir moviendo el valor de la variable local a registro y viceversa. Pero bueno, entendiendo que sólo es para ver como se implementan variables locales vamos a ver un ejemplos super simple y luego ya veremos uno más complejo.
Veamos primero la versión en C:

Código fuente 5.7: `t7_multiplica.c`

```
1  #include<stdio.h>
2  main() {
3      printf("%d\n",
4          multiplica(2,3) +
5          multiplica(5,2));
6  }
7
8  int multiplica(int a, int b) {
```

```
 9      int m=0;
10      while(b!=0) {
11          m=m+a;
12          b--;
13      }
14      return m;
15  }
```

```
$ gcc -g -o t7_multiplica t7_multiplica.c
tarom@sisu:~/Dropbox/TAROM/FO/0Books/src$ ./t7_multiplica
16
$
```

Y ahora la versión en ensamblador:

Código fuente 5.8: `t7_multiplica.s`

```
 1  .text
 2  .global main
 3  main:
 4      pushl $3
 5      pushl $2
 6      call multiplica
 7      addl $8, %esp
 8      movl %eax, %ebx
 9
10      pushl $2
11      pushl $5
12      call multiplica
13      addl $8, %esp
14      addl %eax, %ebx
15
16      movl $1, %eax
17      int $0x80
18
19  multiplica:
20      pushl %ebp
21      movl %esp, %ebp
22      subl $4, %esp
23      pushl %ebx
24
25      movl $0, -4(%ebp)
26      movl 8(%ebp), %eax
27      movl 12(%ebp), %ebx
28  multi_loop:
29      cmp $0, %ebx
30      je multi_endloop
```

```
31      addl %eax, -4(%ebp)
32      decl %ebx
33      jmp multi_loop
34  multi_endloop:
35      movl -4(%ebp), %eax
36      popl %ebx
37      addl $4, %esp
38      popl %ebp
39      ret
```

EL CÓDIGO

- hasta la linea 22: El 'main' no tiene ningún secreto. Invocamos dos veces a 'multiplica' con los dos números a multiplicar y el resultado de las dos multiplicaciones la guardamos en %ebx y finaliza la ejecución. Y en la subrutina 'multiplica' establecemos el enlace dinámico (lineas 20 y 21).
- linea 22: Reservamos espacio para la variable local.
- linea 23: Estado de la pila después de reservar espacio para la variable local en la primera invocación de la subrutina 'multiplica' y salvar los registros base e índices:

@-	%ebx	← %esp	l23 pushl %ebx
(%ebp-4)	VarLoc		l22 subl $4, %esp
(%ebp)	%ebp 'main'	← %ebp	l20 push %ebp
(%ebp+4)	@ret		l06 call multiplica
(%ebp+8)	$2		l05 pushl $2
(%ebp+12)	$3		l04 pushl $3

  - Antes de invocar a 'multiplica' salvaríamos los registros %eax, %ecx, %ebx que usamos. En este caso ninguno.
  - Luego apilaríamos los parámetros con los que invocamos 'multiplica' empezando por el último (l04 y l05).
  - Hacemos el 'call' (l06) que hace que nos apile la dirección de retorno (dirección a la que tenemos que saltar al hacer un 'ret') y salte a la etiqueta de la subrutina 'multiplica'.
  - Establecemos el enlace dinámico salvando el base pointer del 'main' (l20) y haciendo que el nuevo %ebp (el %ebp de 'multiplica') apunte a la cima de la pila actual (l21).

  - Reservamos espacio para la variable 'long' (l22).
  - Salvamos los registros base e índices que se modifican dentro de 'multiplica', que en este caso, sólo es el %ebx (l23).
  - El contenido de esta pila es lo que se conoce como **bloque de activación de la subrutina** (en inglés "stack frame"). Que contiene, como mínimo: los parámetros con la que la subrutina es invocada, la dirección de retorno y las variables locales.

- Línea 24: Ya puede empezar el cuerpo de la subrutina.
- Líneas 25-27: Inicializamos la variable local a cero y movemos el primer y seguntdo parámetro a %eax y a %ebx. **Siempre el primer parámetro está en 8(%ebp) y la primera variable local en -4(%ebp).** Esta es la belleza de los bloques de activación.
- 28-34: Bucle donde sumamos el número contenido en %eax %ebx veces en la variable local.
- Línea 35: Ponemos el valor contenido en la variable local en el registro %eax, porqué es lo que devuelve la función.
- Línea 36: Restauramos %ebx
- Línea 37: Liberamos el espacio que habíamos reservado para la variable local.
- Líneas 38 y 39: Deshacemos el enlace dinámico y retornamos.

Vamos a ver otro ejemplo donde tiene más sentido el uso de variables locales. En el siguiente programa (t7_signos.s) tenemos una subrutina llamada 'signos' a la cual se le pasan dos parámetros. El primero un entero y el segundo un vector de enteros (vamos a llamarlos 'n' y 'v'). En C esto equivaldría a "void signos(int n, int *v)". Pues bien la subrutina recorre las primeras 'n' posiciones del vector 'v' e imprime por pantalla cuantos de estos enteros son cero, cuantos son enteros positivos y cuantos son enteros negativos. Para ello utiliza tres variables 'long' que usa como contadores de ceros, positivos y negativos. Para que se vea fácilmente que el programa funciona adecuadamente, la subrutina 'signos' irá imprimiendo los enteros del vector a medida que lo vaya recorriendo.
Aquí vemos la compilación y ejecución del programa donde la función 'main' invoca a la subrutina 'signos' con tres vectores de enteros diferentes:

```
$ gcc -g -o t7_multiplica t7_multiplica.s
$ ./t7_multiplica
$ echo $?
16
```

```
$
```

Y aquí tenemos el código fuente. Sigue los comentarios e intenta entender que va haciendo linea a linea:

Código fuente 5.9: `t7_signos.s`

```
1  .data
2  s: .asciz "\nCeros:%d. Positivos:%d. Negativos:%d.\n"
3  sd: .asciz "%d, "
4  n1: .long 8
5  v1: .long 7,9,2,3,-4,-5,-6,0
6  n2: .long 6
7  v2: .long 1,2,3,-4,-5,-6
8  n3: .long 1
9  v3: .long 1
10 .text
11 .global main
12 main:
13     pushl $v1
14     pushl n1
15     call signos          # signos(n1,v1)
16     addl $8, %esp
17
18     pushl $v2
19     pushl n2
20     call signos          # signos(n2,v2)
21     addl $8, %esp
22
23     pushl $v3
24     pushl n3
25     call signos          # signos(n3,v3)
26     addl $8, %esp
27
28     movl $0, %ebx
29     movl $1, %eax
30     int $0x80
31
32 signos:                  # void signos(int n, int *v)
33     pushl %ebp           # Salvar antiguo base pointer
34     movl %esp, %ebp      # Establecer enlace dinamico
35     subl $12, %esp       # Reservar para 3 var long
36     pushl %ebx           # Salvar regs %ebx %esi %edi
37
38     movl $0, -4(%ebp)    # Inicializar VarLoc1 a 0
39     movl $0, -8(%ebp)    # Inicializar VarLoc2 a 0
40     movl $0, -12(%ebp)   # Inicializar VarLoc3 a 0
```

```
41      movl $0, %ecx         # Inicializar indice a 0
42      movl 12(%ebp), %ebx # Parametro2 ('v') a %ebx
43  signos_for:
44      cmpl 8(%ebp), %ecx  # Comp Parametro1('n') y %ecx
45      jge signos_endfor     # Si hemos recorrido 'v' salir
46
47      pushl %ecx
48      pushl (%ebx,%ecx,4) # <- esto es v[i]
49      pushl $sd
50      call printf           # printf("%d, ", v[i]);
51      addl $8, %esp
52      popl %ecx
53
54      cmpl $0, (%ebx,%ecx,4) # cmp v[i] con 0
55  #   je signos_zero # == 0  # si v[i] es 0...
56      jg signos_posi # > 0   # si v[i] es > 0...
57      jl signos_nega # < 0   # si v[i] es < 0...
58  signos_zero:               # v[i] es 0
59      incl -4(%ebp)          # VarLoc1++
60      jmp signos_forinc      # salto a fin iteracion
61  signos_posi:               # v[i] es > 0
62      incl -8(%ebp)          # VarLoc2++
63      jmp signos_forinc      # salto a fin iteracion
64  signos_nega:               # v[i] es < 0
65      incl -12(%ebp)         # VarLoc3++
66  #   jmp signos_forinc      # salto a fin iteracion
67  signos_forinc:             # final de iteracion
68      incl %ecx              # i++ (contador de bucle)
69      jmp signos_for         # salto a principio bucle
70  signos_endfor:             # final de bucle
71
72      pushl %ecx
73      pushl -12(%ebp)     # VarLoc3
74      pushl -8(%ebp)      # VarLoc2
75      pushl -4(%ebp)      # VarLoc1
76      pushl $s
77      call printf         # Imprimo las 3 Var locales
78      addl $16, %esp
79      popl %ecx
80
81      popl %ebx           # Restaurar %edi, %esi, %ebx
82      addl $12, %esp      # Liberar espacio Var Locales
83      popl %ebp           # Deshacer enlace dinamico
84      ret                 # Retornar
```

EL CÓDIGO

- hasta la linea 31: El 'main' no tiene ningún secreto. Invocamos tres veces a 'signos' con los tres vectores y finaliza la ejecución.
- linea 15: Estado de la pila antes de invocar 'signos':

@-		
	n1	← %esp
@+	$v1	

  Al invocar la subrutina 'signos' no salvamos copia de ningún registro "rocanrolero" porque no los usamos.
- linea 37: A continuación tenéis el estado de la pila dentro de 'signos' en su primera invocación. Id siguiendo como se va llenando empezando por abajo.

@-	%ebx	← %esp	l36 pushl %ebx
(%ebp-12)	VarLoc3		l35 subl $12, %esp
(%ebp-8)	VarLoc2		l35 subl $12, %esp
(%ebp-4)	VarLoc1		l35 subl $12, %esp
(%ebp)	%ebp 'main'	← %ebp	l33 push %ebp
(%ebp+4)	@ret		l15 call signos
(%ebp+8)	n1		l14 pushl n1
(%ebp+12)	$v1		l13 pushl $v1

  - Antes de invocar a 'signos' salvaríamos los registros %eax, %ecx, %ebx que usamos. En este caso ninguno.
  - Luego apilaríamos los parámetros con los que invocamos 'signos' empezando por el último (l13 y l14).
  - Hacemos el 'call' (l15) que hace que nos apile la dirección de retorno (dirección a la que tenemos que saltar al hacer un 'ret') y salte a la etiqueta de la subrutina 'signos'.
  - Establecemos el enlace dinámico salvando el base pointer del 'main' (l33) y haciendo que el nuevo %ebp (el %ebp de 'signos') apunte a la cima de la pila actual (l34).
  - Reservamos espacio para las 3 variables 'long' (l35).
  - Salvamos los registros base e índices que se modifican dentro de 'signos', que en este caso, sólo es el %ebx (l36).

- Y ya puede empezar el cuerpo de la subrutina. Recordad que el contenido de esta pila es lo que se conoce como **bloque de activación de la subrutina** y que contiene, como mínimo: los parámetros con la que la subrutina es invocada, la dirección de retorno y las variables locales.

- linea 50: Y aquí tenéis el estado de la pila al llamar a 'printf'

(%ebp_printf+4)	@ret a signos	← %esp	l50 call printf
(%ebp_printf+8)	(%ebx, %ecx,4)		l49 pushl ...
(%ebp_printf+12)	%ecx		l48 pushl %ecx
	%ebx		l36 pushl %ebx
(%ebp_signos-12)	VarLoc3		l35 subl $12, %esp
(%ebp_signos-8)	VarLoc2		l35 subl $12, %esp
(%ebp_signos-4)	VarLoc1		l35 subl $12, %esp
(%ebp_signos)	%ebp 'main'		l33 push %ebp
(%ebp_signos+4)	@ret a main		l15 call signos
(%ebp_signos+8)	n1		l14 pushl n1
(%ebp_signos+12)	$v1		l13 pushl $v1

- Como ejercicio coged lápiz y goma e id ejecutando en vuestra mente el programa linea a linea (como si fueras un debuger) a la vez que vais apilando y desapilado. Al finalizar la ejecución la pila debe estar vacía. Y en todo momento se debe mantener que el primer parámetro con el que han invocado a la subrutina es (%ebp+8) y que la primera variable local de la subrutina es (%ebp-4). Si lo consigues: felicidades! Ya no vamos a complicar las cosas mucho más.

EL BOXEO

- El ensamblador es como el boxeo porque todo va al revés:
  Para reservar espacio en la pila se le resta el tamaño en bytes al %esp.
  Y para liberar ese espacio, se le suma.

TEORÍA

- RESUMEN DE LLAMADA A SUBRUTINA

- Subrutina1 llama a Subrutina2. El número (1..14) índica en que orden suceden los pasos.
- **En el código de Subrutina 1:**
- 1- Salvar los regs
- 2- Pasar los parámetros
- 3- Llamar a la subrutina 2
- 12- Recoger resultado
- 13- Eliminar parámetros
- 14- Restaurar regs
- **En el código de Subrutina2:**
- 4- Establecer enlace dinámico
- 5- Reservar espacio para var locales
- 6- Salvar registros
- (Ejecución de la subrutina 2)
- 7- Devolver resultado
- 8- Restaurar regs
- 9- Liberar el espacio de var locales
- 10- Deshacer enlace dinámico
- 11- Retorno

5.9. Paso de parámetros por referencia

Vamos a aprender como invocar una subrutina pasándole parámetros por referencia. Para ello primero vamos a ver un programa en C que luego traduciremos a ensamblador.

Código fuente 5.10: `t7_complex_swap.c`

```
1  #include<stdio.h>
2
3  void swap(int *a, int *b) {
4      int esi, edi;
5      printf("swap: (@0x%X: %d) (@0x%X: %d)\n", a, *a, b,
           *b);
6      esi = *a;
7      edi = *b;
8      *b = esi;
9      *a = edi;
10     printf("swap: (esi:%d) (edi:%d)\n", esi, edi);
11     printf("swap: (@0x%X: %d) (@0x%X: %d)\n", a, *a, b,
           *b);
12 }
13
14 void func(int a) {
15     int b = -7;
16     printf("func: (%d) (%d)\n", a, b);
17     swap(&a, &b);
18     printf("func: (%d) (%d)\n", a, b);
19 }
20
21 main(){
22     func(7);
23 }
```

El código, que no hace nada demasiado útil, tiene una función 'void func(int a)' con una variable local 'int b=-7'. Y lo que hace es pasar el parámetro 'a' y la variable 'b' por referencia a una función 'swap' (intercambiar en inglés) que intercambia sus valores.
Podéis ver que tienen unos cuantos printf para demostrarnos que funciona correctamente.
Si lo compilamos nos da cuatro 'warnings' de que el formato "%X" en printf espera un argumento de tipo entero y le estamos pasando un puntero a entero. Pero eso es por que quiero imprimir en que dirección de memoria están las variables. (Si veis palabras raras en los mensajes de los 'warnings' es porqué la configuración de mi ordenador hace que los mensajes de error salgan en una mezcla de inglés y finlandés).

```
$ gcc -g -o t7_complex_swap t7_complex_swap.c
t7_complex_swap.c: Funktio "swap":
t7_complex_swap.c:5:2: varoitus: format "%X" expects
 argument of type "unsigned int", but argument 2 has
 type "int *" [-Wformat]
t7_complex_swap.c:5:2: varoitus: format "%X" expects
 argument of type "unsigned int", but argument 4 has
 type "int *" [-Wformat]
t7_complex_swap.c:11:2: varoitus: format "%X" expects
 argument of type "unsigned int", but argument 2 has
 type "int *" [-Wformat]
t7_complex_swap.c:11:2: varoitus: format "%X" expects
 argument of type "unsigned int", but argument 4 has
 type "int *" [-Wformat]
$ ./t7_complex_swap
func: (7) (-7)
swap: (@0xBF850AD0:7) (@0xBF850ABC:-7)
swap: (esi:7) (edi:-7)
swap: (@0xBF850AD0:-7) (@0xBF850ABC:7)
func: (-7) (7)
$
```

Ahora vamos a traducir el código a ensamblador de tal manera que, como ya habréis intuido, las variables 'esi' e 'edi' pasen a ser los registros '%esi' e '%edi'.
El código es muy simple en las funciones 'main' y 'func' hasta que llegamos a la línea 30 donde estamos en la función 'func' y tenemos que apilar los parámetros que tenemos que pasar por referencia.
El problema es, ¿cómo hago un push de un puntero a por ejemplo '-4(%ebp)'?. La solución que utilizamos en este código es calcular en %ebx %ebp-4 (lineas 30 y 31) y hacer un 'push %ebx' y lo mismo para '8(%ebp)'.
Y la parte interesante de la función 'swap' (no os dejéis marear por los 'printf') es, primero, cuando quiero que lo apuntado por el primer parámetro vaya al registro %esi (líneas 66 y 67) y que lo apuntado por el segundo parámetro vaya al registro %edi (líneas 68 y 69). Y, segundo, cuando hago que luego los valores de esos dos registros se asignen de nuevo (ahora intercambiados) a las variables apuntadas por los dos punteros (líneas 70 y 71).
Examinad con calma estas seis lineas. El secreto está en llegara a ver que lo que hacemos es: Mover el primer puntero a %eax y el segundo a %ecx. Mover los valores apuntados por los punteros a %esi y %edi y mover %edi y %esi intercambiados a las posiciones de memoria apuntadas por los dos punteros.

Código fuente 5.11: `t7_complex_swap.s`

```
1  .data
2  sfunc: .asciz "func: (%d) (%d)\n"
```

```
 3 sswap: .asciz "swap: (@0x%X: %d) (@0x%X: %d)\n"
 4 sswap2: .asciz "swap: (esi:%d) (edi:%d)\n"
 5 .text
 6 .global main
 7 main:
 8     pushl $7
 9     call func
10     addl $4, %esp
11
12     movl $0, %ebx
13     movl $1, %eax
14     int $0x80
15
16 func:
17     pushl %ebp
18     movl %esp, %ebp
19     subl $4, %esp
20     pushl %ebx
21
22     movl $-7, -4(%ebp)
23
24     pushl -4(%ebp)
25     pushl 8(%ebp)
26     pushl $sfunc
27     call printf
28     addl $12, %esp
29
30     movl %ebp, %ebx
31     subl $4, %ebx
32     pushl %ebx # puntero a -4(%ebp)
33     movl %ebp, %ebx
34     addl $8, %ebx
35     pushl %ebx # puntero a 8(%ebp)
36     call swap
37     addl $8, %esp
38
39     pushl -4(%ebp)
40     pushl 8(%ebp)
41     pushl $sfunc
42     call printf
43     addl $12, %esp
44
45     popl %ebx
46     addl $4, %esp
47     popl %ebp
48     ret
```

```
49
50 swap:
51      pushl %ebp
52      movl %esp, %ebp
53      pushl %esi
54      pushl %edi
55
56      movl 12(%ebp), %ecx
57      pushl (%ecx)
58      pushl %ecx
59      movl 8(%ebp), %eax
60      pushl (%eax)
61      pushl %eax
62      pushl $sswap
63      call printf
64      addl $20, %esp
65
66      movl 8(%ebp), %eax
67      movl 12(%ebp), %ecx
68      movl (%eax), %esi
69      movl (%ecx), %edi
70      movl %edi, (%eax)
71      movl %esi, (%ecx)
72
73      pushl %edi
74      pushl %esi
75      pushl $sswap2
76      call printf
77      addl $12, %esp
78
79      movl 12(%ebp), %ecx
80      pushl (%ecx)
81      pushl %ecx
82      movl 8(%ebp), %eax
83      pushl (%eax)
84      pushl %eax
85      pushl $sswap
86      call printf
87      addl $20, %esp
88
89      popl %edi
90      popl %esi
91      popl %ebp
92      ret
```

Y la ejecución de la versión en ensamblador nos demuestra que funciona correctamente:

```
$ gcc -g -o t7_complex_swap t7_complex_swap.s
$ ./t7_complex_swap func: (7) (-7)
swap: (@0xBFF870C8:7) (@0xBFF870BC:-7)
swap: (esi:7) (edi:-7)
swap: (@0xBFF870C8:-7) (@0xBFF870BC:7)
func: (-7) (7)
$
```

5.9.1. Load Effective Address (lea)

En realidad, esto de "calcular una dirección de memoria en un registro" no se suele hacer. Existe una instrucción llamada 'leal' que no forma parte del temario de la asignatura que imparto.
Lo que nosotros hacemos complicado se hace un poco más fácil (y en mi opinión más intuitivo) con leal (Load Effective Address [Long]). Lea es como un 'mov' que como 'op1' tiene una dirección de memoria ("D(Rb,Ri,S)") pero que en lugar de mover lo contenido por la dirección de memoria a 'op2' mueve la dirección de memoria (D+Rb+Ri*S) a 'op2'.
Por lo tanto:

```
1       movl %ebp, %e
2       subl $4, %ebx
3       pushl %ebx # puntero a -4(%ebp)
4       movl %ebp, %ebx
5       addl $8, %ebx
6       pushl %ebx # puntero a 8(%ebp)
7       call swap
```

Equivaldría a:

```
1       leal -4(%ebp), %ebx
2       pushl %ebx
3       leal 8(%ebp), %ebx
4       pushl %ebx
5       call swap
```

5.10. Modos de direccionamiento complejos

Y, para acabar, os dejo con ejemplos de programas ensamblador medianos que tienen modos de direccionamiento complejos. Que disfrutéis.

5.10.1. OFFSET_VAR(%ebp, %ecx,4)

Programa que indica cuantas veces aparece cada letra del abecedario en una cadena ascii. En el podemos ver como acceder a posiciones de vectores que se han pasado como parámetro o que son variables locales.
En este ejemplo podemos ver que 'OFFSET_LETRAS' es una constante que como valor tiene el "offset" o desplazamiento relativo de una variable a %ebp. Por lo tanto lo que seria 'v[i]' en C (donde 'v' es una variable local). En ensamblador será OFFSET_LETRAS(%ebp, %ecx,4). El '4' porque es un vector de 'long's.

Código fuente 5.12: `t7_cuento_minusculas_b.s`

```
1  NUM_LETRAS = 26
2  SIZE_LETRAS = +4*NUM_LETRAS
3  OFFSET_LETRAS = -4*NUM_LETRAS
4  .data
5  s_intro: .asciz "Cuento las apariciones de letras
       minusculas.\n\n"
6  s1: .asciz "hola, muchas letras veo yo!"
7  s2: .asciz "murcielago"
8  s3: .asciz "9aaaaaaaaas"
9  s: .asciz "%d "
10 sc: .asciz "%c"
11 sr: .asciz "\n\n"
12 s_letras: .asciz "\na b c d e f g h i j k l m n o p q
       r s t u v w x y z\n"
13 .text
14 .global main
15 main:
16  pushl $s_intro
17  call printf
18  addl $4, %esp
19
20  pushl $s1
21  call letras
22  addl $4, %esp
23
24  pushl $s2
25  call letras
26  addl $4, %esp
27
28  pushl $s3
29  call letras
30  addl $4, %esp
```

```
31
32   movl $0, %ebx
33   movl $1, %eax
34   int $0x80
35
36  letras:
37   pushl %ebp
38   movl %esp, %ebp
39   subl $SIZE_LETRAS, %esp
40   pushl %ebx
41
42   movl $0, %ecx
43  letras_for0:
44   cmpl $NUM_LETRAS, %ecx
45   jge letras_endfor0 # %ecx>=$NUM_LETRAS
46   movl $0, OFFSET_LETRAS(%ebp,%ecx,4)
47   incl %ecx
48   jmp letras_for0
49  letras_endfor0:
50
51   movl 8(%ebp), %ebx
52   movl $0, %ecx
53  letras_for1:
54   cmpb $0, (%ebx,%ecx)
55   je letras_endfor1 # $0==(%ebx,%ecx)
56
57   pushl %ecx
58   pushl (%ebx,%ecx)
59   pushl $sc
60   call printf
61   addl $8, %esp
62   popl %ecx
63
64   # passar de 'a' a 0, de 'b' a 1, ..., 'z' a 25
65   movl $0, %eax
66   movb (%ebx,%ecx), %al
67   # begin controlar que passa si (%al<'a' || %al>'z')
68   cmpb $'a', %al
69   jl letras_incfor1 # %al<'a'
70   cmpb $'z', %al
71   jg letras_incfor1 # %al>'z'
72   # end controlar
73   subb $'a', %al
74
75   incl OFFSET_LETRAS(%ebp,%eax,4)
76
```

```
77  letras_incfor1:
78   incl %ecx
79   jmp letras_for1
80  letras_endfor1:
81
82   pushl %ecx
83   pushl $s_letras
84   call printf
85   addl $4, %esp
86   popl %ecx
87
88   movl $0, %ecx
89  letras_for2:
90   cmpl $NUM_LETRAS, %ecx
91   jge letras_endfor2 # %ecx>=$NUM_LETRAS
92
93   pushl %ecx
94   pushl OFFSET_LETRAS(%ebp,%ecx,4)
95   pushl $s
96   call printf
97   addl $8, %esp
98   popl %ecx
99
100  incl %ecx
101  jmp letras_for2
102 letras_endfor2:
103
104  pushl $sr
105  call printf
106  addl $4, %esp
107
108  popl %ebx
109  movl %ebp, %esp
110  popl %ebp
111  ret
```

```
$ gcc -g -o t7_cuento_minusculas_b t7_cuento_minusculas_b.s
$ ./t7_cuento_minusculas_b
Cuento las apariciones de letras minusculas.

hola, muchas letras veo yo!
a b c d e f g h i j k l m n o p q r s t u v w x y z
3 0 1 0 2 0 0 2 0 0 0 2 1 0 3 0 0 1 2 1 1 1 0 0 1 0

murcielago
```

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
1 0 1 0 1 0 1 0 1 0 0 1 1 0 1 0 0 1 0 0 1 0 0 0 0 0

9aaaaaaaaas
a b c d e f g h i j k l m n o p q r s t u v w x y z
9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0

$
```

5.10.2. (%ebx,%ecx,4)

Variante del mismo programa con direccionamientos del estilo '(%ebx, %ecx,4)'. Lo que hacemos es poner en el registro base (%ebx) la base del vector (que es %ebp + OFFSET_DEL_VECTOR) y a partir de aquí la equivalencia de lo que sería 'v[i]' en C es '(%ebx, %ecx,4)' en ensamblador.

Código fuente 5.13: `t7_cuento_minusculasb.s`

```
1  NUM_LETRAS = 26
2  SIZE_LETRAS = +4*NUM_LETRAS
3  OFFSET_LETRAS = -4*NUM_LETRAS
4  .data
5  s_intro: .asciz "Cuento las apariciones de letras
       minusculas.\n\n"
6  s1: .asciz "hola, muchas letras veo yo!"
7  s2: .asciz "murcielago"
8  s3: .asciz "9aaaaaaaaas"
9  s: .asciz "%d "
10 sc: .asciz "%c"
11 sr: .asciz "\n\n"
12 s_letras: .asciz "\na b c d e f g h i j k l m n o p q
       r s t u v w x y z\n"
13 .text
14 .global main
15 main:
16  pushl $s_intro
17  call printf
18  addl $4, %esp
19
20  pushl $s1
21  call letras
22  addl $4, %esp
23
24  pushl $s2
25  call letras
```

```
26   addl $4, %esp
27
28   pushl $s3
29   call letras
30   addl $4, %esp
31
32   movl $0, %ebx
33   movl $1, %eax
34   int $0x80
35
36  letras:
37   pushl %ebp
38   movl %esp, %ebp
39   subl $SIZE_LETRAS, %esp
40   pushl %ebx
41   pushl %esi
42
43   movl $0, %ecx
44   movl %ebp, %ebx
45   addl $OFFSET_LETRAS, %ebx
46  letras_for0:
47   cmpl $NUM_LETRAS, %ecx
48   jge letras_endfor0 # %ecx>=$NUM_LETRAS
49   movl $0, (%ebx,%ecx,4)
50   incl %ecx
51   jmp letras_for0
52  letras_endfor0:
53
54   movl 8(%ebp), %esi
55   movl $0, %ecx
56  letras_for1:
57   cmpb $0, (%esi,%ecx)
58   je letras_endfor1 # $0==(%esi,%ecx)
59
60   pushl %ecx
61   pushl (%esi,%ecx)
62   pushl $sc
63   call printf
64   addl $8, %esp
65   popl %ecx
66
67   # passar de 'a' a 0, de 'b' a 1, ..., 'z' a 25
68   movl $0, %eax
69   movb (%esi,%ecx), %al
70   # begin controlar que passa si (%al<'a' || %al>'z')
71   cmpb $'a', %al
```

```
72   jl letras_incfor1 # %al<'a'
73   cmpb $'z', %al
74   jg letras_incfor1 # %al>'z'
75   # end controlar
76   subb $'a', %al
77
78   incl (%ebx,%eax,4)
79
80  letras_incfor1:
81   incl %ecx
82   jmp letras_for1
83  letras_endfor1:
84
85   pushl %ecx
86   pushl $s_letras
87   call printf
88   addl $4, %esp
89   popl %ecx
90
91   movl $0, %ecx
92  letras_for2:
93   cmpl $NUM_LETRAS, %ecx
94   jge letras_endfor2 # %ecx>=$NUM_LETRAS
95
96   pushl %ecx
97   pushl (%ebx,%ecx,4)
98   pushl $s
99   call printf
100  addl $8, %esp
101  popl %ecx
102
103  incl %ecx
104  jmp letras_for2
105 letras_endfor2:
106
107  pushl $sr
108  call printf
109  addl $4, %esp
110
111  popl %esi
112  popl %ebx
113  movl %ebp, %esp
114  popl %ebp
115  ret
```

5.10.3. -4(%ebx, %ecx,4)

Programa que calcula la sucesión de Fibonacci.
La sucesión de Fibonacci es tal que f[0]=0, f[1]=1, y para i ¿=1, f[i]=f[i-1]+f[i-2].
Si %ebx apunta a la base del vector de enteros 'v' y %ecx al indice del vector 'i':
'v[i]' es (%ebx, %ecx,4), 'v[i-1]' es -4(%ebx, %ecx,4), i 'v[i-2]' es -8(%ebx, %ecx,4).
Que grande, ¿no?
Mira las lineas 60-62 para ver como hacemos el 'v[i]=v[i-1]+v[i-2];'.

Código fuente 5.14: `t7_fibonacci.s`

```
1  SIZE = 20
2  .data
3  s: .asciz "%d\n"
4  .bss
5  .comm v,4*SIZE,4
6  .text
7  .global main
8  main:
9   # fibonacci[0]=0
10  movl $0, %ecx
11  movl $0, v(, %ecx,4)
12  # fibonacci[1]=1
13  movl $1, %ecx
14  movl $1, v(, %ecx,4)
15  # fibonacci[2..SIZE]
16  movl $2, %ecx
17 loop:
18  # %ecx>=$SIZE
19  cmpl $SIZE, %ecx
20  jge endloop
21
22  push %ecx
23  push $v
24  call fibonacci
25  addl $4, %esp
26  popl %ecx
27
28  incl %ecx
29  jmp loop
30 endloop:
31
32  # printf the array
33  movl $0, %ecx
34 loop_print:
35  cmpl $SIZE, %ecx
```

```
36   jge endloop_print
37   pushl %ecx
38   pushl v(,%ecx,4)
39   pushl $s
40   call printf
41   addl $8, %esp
42   popl %ecx
43   incl %ecx
44   jmp loop_print
45  endloop_print:
46
47   # the end
48   movl $0, %ebx
49   movl $1, %eax
50   int $0x80
51
52  fibonacci:
53   pushl %ebp
54   movl %esp, %ebp
55   pushl %ebx
56
57   movl 12(%ebp), %ecx
58   movl 8(%ebp), %ebx
59
60   movl -4(%ebx,%ecx,4), %edx
61   addl -8(%ebx,%ecx,4), %edx
62   movl %edx, (%ebx,%ecx,4)
63
64   popl %ebx
65   popl %ebp
66   ret
```

```
$ gcc -g -o t7_fibonacci t7_fibonacci.s
$ ./t7_fibonacci
0
1
1
2
3
5
8
13
21
34
55
```

```
89
144
233
377
610
987
1597
2584
4181
$
```

5.10.4. v(%esi, %ecx,4)

Variante del mismo programa que es más simple por que no hay llamada a subrutina y el vector no se pasa como parámetro. Con lo cual podemos hablar de 'v[a+i]' con direccionamientos del estilo 'v(%esi, %ecx,4)'.

Código fuente 5.15: `t7_fibonacci_2.s`

```
1  SIZE = 20
2  .data
3  s: .asciz "%d\n"
4  .bss
5  .comm v,4*SIZE,4
6  .text
7  .global main
8  main:
9   # fibonacci[0]=0
10  movl $0, %ecx
11  movl $0, v(, %ecx,4)
12  # fibonacci[1]=1
13  movl $1, %ecx
14  movl $1, v(, %ecx,4)
15  # fibonacci[2..SIZE]
16  movl $2, %ecx
17  movl $-4, %esi
18  movl $-8, %edi
19 loop:
20  # %ecx>=$SIZE
21  cmpl $SIZE, %ecx
22  jge endloop
23
24  movl v(%esi,%ecx,4), %edx
25  addl v(%edi,%ecx,4), %edx
26  movl %edx, v(,%ecx,4)
```

```
27
28   incl %ecx
29   jmp loop
30  endloop:
31
32   # printf the array
33   movl $0, %ecx
34  loop_print:
35   cmpl $SIZE, %ecx
36   jge endloop_print
37   pushl %ecx
38   pushl v(,%ecx,4)
39   pushl $s
40   call printf
41   addl $8, %esp
42   popl %ecx
43   incl %ecx
44   jmp loop_print
45  endloop_print:
46
47   # the end
48   movl $0, %ebx
49   movl $1, %eax
50   int $0x80
```

Capítulo 6

Epílogo

Aprender siempre es un proceso doloroso y uno debe no sólo aceptar, sino "abrazarse" a ese dolor. Caminar contra esa resistencia que notamos. Como decía un maestro Zen "El obstáculo es el camino!".

En este libro he intentado disminuir el dolor y la resistencia de este proceso de aprendizaje. También he intentado hacer más divertido el proceso. Porqué uno siempre aprende más si se lo pasa bien. Pero ahora falta que vosotros le echéis horas y juguéis (programéis) hasta que esto se os de bien.

Espero que os lo hayáis pasado tan bien leyendo este libro como yo me lo he pasado escribiéndolo. Y que, con un poco de suerte, en breve seáis expertos programadores en el ensamblador del IA32.

HICE UN PROGRAMA
DE MÁS DE CIEN LÍNEAS
EN ENSAMBLADOR Y
COMPILÓ A LA PRIMERA