Práctica 1.- Programación en ensamblador x86 Linux

1 Resumen de objetivos

Al finalizar esta práctica, se debería ser capaz de:

- Usar las herramientas `gcc`, `as`, `ld`, `objdump` y `nm` para compilar código C, ensamblar y enlazar código ensamblador, y localizar y examinar el código generado por el compilador.
- Reconocer la estructura del código generado por `gcc` para rutinas aritméticas muy sencillas, relacionando las instrucciones del procesador con la construcción C de la que provienen.
- Describir la estructura general de un programa ensamblador en `gas` (GNU assembler).
- Escribir un programa ensamblador sencillo.
- Usar interrupciones software para hacer llamadas al sistema operativo (*kernel* Linux).
- Enumerar los registros e instrucciones más usuales de los procesadores de la línea x86.
- Usar con efectividad un depurador como `gdb/ddd`, para ver el contenido de los registros, ejecutar paso a paso y con puntos de ruptura, desensamblar el código, y volcar el contenido de la pila y los datos. Los depuradores permiten familiarizarse con la arquitectura del computador.
- Argumentar la utilidad de los depuradores para ahorrar tiempo de depuración.
- Explicar la gestión de pila en procesadores x86.
- Recordar y practicar en una plataforma de 32bits la representación de caracteres, de enteros naturales y enteros con signo en complemento a dos, y las operaciones de suma en doble precisión y división entera.

2 Herramientas de prácticas

Las prácticas se realizarán en Linux usando las herramientas GNU. Para compilar (traducir fuente C a ensamblador, objeto o ejecutable) se usará `gcc`, para ensamblar (traducir fuente ensamblador a objeto) se usará `as`, para enlazar `ld` (permite combinar varios ficheros objeto en un único fichero ejecutable), y para depurar `gdb`, a través del entorno gráfico `ddd`.

En la Figura 1 se pueden ver los programas que se ejecutan para obtener un ejecutable a partir de un fichero fuente en lenguaje C: compilador, ensamblador y enlazador. En la práctica el programador en lenguaje de alto nivel no tiene que ejecutar los tres programas por separado.

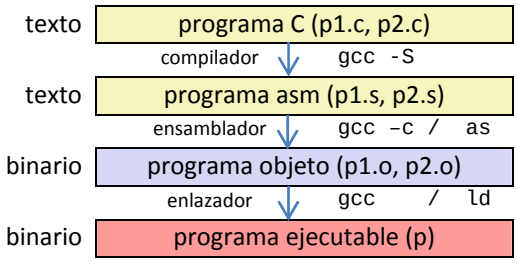


Figura 1: proceso de compilación

El código fuente se puede crear con cualquier editor de texto, como por ejemplo `gedit`, teniendo cuidado de que la extensión coincida con el tipo de lenguaje usado (.c/.s). El procesamiento del compilador `gcc` puede detenerse tras las etapas de traducción a ensamblador o a objeto con los modificadores (*switches*) `-S/-c`. El proceso puede continuarse con el propio `gcc` o con el ensamblador `as` y el enlazador `ld`.

Por ejemplo, con estos dos ficheros fuente en lenguaje C…

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Figura 2: programa p1.c

```
int main()
{
  return sum(1,3);

//printf("%d\n", sum(1,3));
}
```

Figura 3: programa p2.c

…usando `gcc` se podría producir el ejecutable `p` con una sola orden, o generar el código ensamblador u objeto para cada fichero, o retomar esos ficheros ensamblador u objeto para producir el ejecutable. Teniendo los ficheros intermedios, también se podría continuar con las otras herramientas `as/ld`.

Los modificadores necesarios se pueden consultar en los manuales (`man gcc`, `man as`, `man ld`…) aunque los más habituales son:

gcc			
switch	**argumento**	**sentido**	**explicación**
-c		Compile	Compila o ensambla los fuentes, pero no enlaza. Se obtiene un objeto por cada fuente. Por defecto, los objetos se llaman como el fuente.o
-S		aSsemble	Compila los fuentes pero no ensambla. Se obtiene un fuente ensamblador por cada fuente C. Por defecto, se llaman como el fuente.s
-o	fichero.o	Output	Cambiar el nombre del fichero producido. Por defecto, ejecutable a.out, objeto fuente.o, ensamblador fuente.s
-g	1…3	debuG	Genera información de depuración (por defecto, nivel 2)
-O	0…3, s		Optimizar para velocidad (0 no…3 agresivo) o tamaño (s). Poner –O = –O1. [nada] = –O0.
-m32 **-m64**		Machine	Genera código para entorno de 32/64 bits, aunque no sea el configurado por defecto.
-L	dir	LibraryDir	Añadir dir a la lista de búsqueda de librerías (preferible sin espacio: -Ldir)
-l	name	LibraryName	Enlazar contra libname.so / libname.a (preferible usar sin espacio: -lname)
-print-file-name=lib			Imprime el pathname que se usaría para lib a la hora de enlazar
-v		Verbose	Imprime los comandos ejecutados para las diversas etapas de la compilación
-###			Como –v, pero no ejecuta los comandos. Es más fácil de leer.
as			
-g		debuG	mismo sentido
-o	fichero.o	Output	mismo sentido
--32 **--64**			
ld			
-L	dir	LibraryDir	mismo sentido
-l	name	LibraryName	mismo sentido
-m	elf_i386 elf_x86_64	Machine	
-M		printImpMap	Imprimir mapa de enlazado, indicando posición en memoria de objetos, símbolos, etc.

Tabla 1: modificadores para gcc, as, ld

Ejercicio 1: gcc

Crear los ficheros p1.c y p2.c mostrados anteriormente (Figura 2, Figura 3), y reproducir con ellos la siguiente sesión en línea de comandos Linux. Recordar que los modificadores `–m32`, `--32`, `-melf_i386`, son necesarios cuando se desea generar código de 32bit en una plataforma de 64bit.

```
gcc      p1.c p2.c -o p       # compilar de una vez
./p ; echo $?                 # muestra cód. ret. 4
file p                        # ELF 64-bit LSB executable

gcc –m32 p1.c p2.c -o p       # compilar para 32bits
./p ; echo $?                 # muestra cód. ret. 4
file p                        # ELF 32-bit LSB executable

ls                            # existen p1.c, p2.c, p
gcc –m32 -O -S p1.c           # crea p1.s (optimización básica –O1)
    cat p1.s                  # observar aspecto código x86
gcc         -S p2.c           # crea p2.s
    cat p2.s                  # observar aspecto código x86_64
    rm  p2.s                  # no lo necesitaremos

gcc -m32 -c p1.s p2.c         # crea p1.o, p2.o, ambos 32bit
ls; file p1.o p2.o            # comprobar: ELF 32-bit LSB relocatable

gcc -m32 p1.o p2.o –o p       # crear ejecutable p (no a.out) partiendo de objetos
./p; echo $?                  # funciona
```

Figura 4: compilación, ensamblado y enlazado, usando gcc

Observar que el código ensamblador x86 menciona registros de 32bit como EBP (en instrucciones como `pushl %ebp` / `movl %esp, %ebp`), mientras que el código x86_64 menciona registros de 64bit (como en `pushq %rbp` / `movq %rsp, %rbp`). En el shell bash, `$?` representa el código de estado retornado por el último programa ejecutado, y la almohadilla # introduce un comentario hasta final de línea. El punto y coma "`;`" separa dos comandos en la misma línea.

Ejercicio 2: as y ld

El mismo resultado se puede obtener con las distintas herramientas separadamente. Reproducir la siguiente sesión de comandos Linux.

```
rm *.s *.o p; ls              # limpiar
gcc –m32 –O –S p1.c p2.c      # ASM para 32bits, optimización básica –O1
ls

as –32 p1.s –o p1.o           # ensamblar creando objeto p1.o (no a.out)
as –32 p2.s –o p2.o
ls; file *.o

ld p1.o p2.o                  # errores: x86_64, y falta _start
ld –m elf_i386 p1.o p2.o      # _start está definido en crt1.o, verlo con nm

gcc –### -m32 p1.o p2.o       # una forma de ver cómo enlaza gcc
gcc –print-file-name=         # nos interesa reproducir último paso collect2…
LDIR=`gcc –print-file-name=`  # …usando ld, y este command substitution es…
echo $LDIR                    # una forma de ahorrarse teclear el directorio gcc
ld –m elf_i386 p1.o p2.o –o p \
       -dynamic-linker /lib/ld-linux.so.2 \
       /usr/lib32/crt1.o /usr/lib32/crti.o /usr/lib32/crtn.o \
       -lc             $LDIR/32/crtbegin.o $LDIR/32/crtend.o
./p; echo $?                  #funciona
```

Figura 5: compilación usando gcc, ensamblado y enlazado usando as y ld

Dependiendo de la version y configuración del compilador en la distribución que se use, el proceso de enlazado indicado por gcc `-###` será más o menos complicado. El *backslash* "\" continúa un comando que se desea prolongar a la siguiente línea. Las comillas inversas `…` (*backquote*) en bash sirven para reemplazar (*command substitution*) el comando entrecomillado por su salida estándar. En este caso, las hemos usado para definir la variable de entorno `LDIR`, usada posteriormente 2 veces con `ld`.

Si se desea, se puede modificar el programa p2.c para que el comentario "`//`" afecte a la primera sentencia en lugar de a la última. De esta forma, el programa imprime el resultado, en lugar de retornarlo como código de estado. La sintaxis del formato de printf se puede consultar en los manuales (`man 3 printf`).

2.1 Código ensamblador y código máquina

La traducción del ejemplo a lenguaje ensamblador ya la vimos en la sesión de la Figura 4:

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Figura 6: programa p1.c

```
 .file "p1.c"
  .text
 .globl sum
  .type sum, @function
sum:
 pushl %ebp
 movl %esp, %ebp
 movl 12(%ebp), %eax
 addl 8(%ebp), %eax
 leave
 ret
 .size sum, .-sum
 .section .note.GNU-stack,"",@progbits
 .ident "GCC: (GNU) 3.4.6 20060404 (Red Hat 3.4.6-9)"
```

Figura 7: fichero ensamblador p1.s

La versión ensamblador es una representación legible de las instrucciones máquina en que se convierte el programa, como las comentadas anteriormente `pushl %ebp` / `movl %esp, %ebp`. Contiene también directivas, como `.text` (iniciar sección de código) y `.size` (tamaño de un símbolo). En este caso, se define el tamaño de símbolo `sum` (una función global, ver `.global` y `.type`) como "`.-sum`", esto es, la diferencia entre el contador de posiciones "`.`" y la propia etiqueta `sum`. El ensamblador *emite* código máquina conforme traduce ensamblador, ocupando posiciones (bytes) de memoria. El símbolo "`.`" es la posición por donde se va ensamblando, y cada etiqueta toma el valor del contador cuando se emite.

El fichero objeto p1.o no es legible, ya que contiene código máquina, pero se puede desensamblar y consultar los símbolos que define con otras utilidades del paquete GNU *binutils*, como `objdump` y `nm`.

Los modificadores necesarios se pueden consultar en los manuales (`man objdump`, `man nm`) aunque los más habituales son:

objdump	fich.obj.		
switch	**argumento**	**sentido**	**explicación**
-d		Dis-assemble	Muestra los mnemotécnicos ensamblador correspondientes a las instrucciones máquina en las secciones de código del fichero objeto
-S		Source	Intercala código fuente con desensamblado. Implica –d. Requiere compilar con –g.
-h		Headers	Resumen de las cabeceras de sección presentes
-r		Reloc	Muestra las reubicaciones
-t	1…3	table	Muestra las entradas de la tabla de símbolos (similar a nm)
-T	0…3, s	Table	Muestra tabla de símbolos dinámica (similar a nm -D). Para librerías compartidas.
-j / --section=	name	Just	Seleccionar información sólo de la sección mencionada
-s / --full-contents			Mostrar contenidos completos de todas las secciones (o sólo de las indicadas con -j)
nm	fich.obj.		
-D		Dynamic	Mostrar símbolos dinámicos (p.ej. en librerías compartidas)

Tabla 2: modificadores para objdump, nm

Ejercicio 3: objdump y nm

Reproducir la siguiente sesión en línea de comandos Linux

```
objdump –d p1.o  # mostrado al lado->
objdump –t p1.o
nm         p1.o  # sum está en .text

objdump –S p1.o         # no hay debug
gcc –m32 –g –O –c p1.c # añadir -g
objdump –S p1.o         # ahora sí
objdump –t p1.o         # secciones -g
objdump –h p1.o         # ver.text=11B
gcc –m32    –O –c p1.c # quitar -g
objdump –S p1.o         # ahora no
objdump –h p1.o
```

Figura 8: sesión Linux

```
p1.o: formato del fichero elf32-i386

Desensamblado de la seccion .text:

00000000 <sum>:
   0: 55          push   %ebp
   1: 89 e5       mov    %esp,%ebp
   3: 8b 45 0c    mov    0xc(%ebp),%eax
   6: 03 45 08    add    0x8(%ebp),%eax
   9: c9          leave
   a: c3          ret
```

Figura 9: desensamblado de p1.o

Como vemos en la Figura 9, el ensamblador emitió 11 bytes, el contador iba por 0 cuando se definió `sum`, irá por 0xb tras emitir `ret`, y por tanto "`.size sum, .-sum`" calculará el tamaño de sum como 11. La primera instrucción se codifica en lenguaje máquina como 0x55 y ocupa 1B, la posición 0. La segunda instrucción, "`mov %esp,%ebp`", empieza en 0x1, ocupa 2B y acaba por tanto en 0x2, dejando el contador de posiciones en 0x3.

Se puede comprobar (con `nm`) que el símbolo `_start` que nos impedía enlazar nuestros dos objetos con `ld` a secas (Figura 5) está en uno de los objetos añadidos por `gcc`. En el Apéndice 2 hay un resumen de las instrucciones y modos de direccionamiento x86 y de las directivas del ensamblador GNU, que puede resultar útil para entender tanto este desensamblado como el siguiente programa completo.

3 Primer programa completo en ensamblador

Teclear (o copiar-pegar, o descargar del sitio web de la asignatura) el código de la Figura 10 en un fichero llamado `saludo.s`. Si se opta por reescribirlo, tener en cuenta que la almohadilla # indica que el resto de la línea es comentario, con lo cual no es necesario copiarlo.

En el código se pueden distinguir: instrucciones del procesador, directivas del ensamblador, etiquetas, expresiones y comentarios. Las instrucciones usadas en este caso han sido INT y MOV, para realizar las dos llamadas al sistema requeridas (escribir mensaje y terminar programa). Las directivas son comandos que entiende el ensamblador (no instrucciones del procesador), y se han usado para declarar las secciones de datos y código, para emitir un *string* y un entero (en `.data`) y para declarar como global el punto de entrada (en `.text`). Las etiquetas se han usado para nombrar esos tres elementos (`saludo`, `longsaludo` y `_start`), y poder referirse a ellos posteriormente (en *`WRITE`* o en `.global`), ya que representan su dirección de comienzo. Notar el uso del contador de posiciones y aritmética de etiquetas (`.-saludo`) para calcular la longitud del *string*. La otra expresión de inicialización es el valor del *string*. Los comentarios se indican con # ó `/**/`. Los valores inmediatos se prefijan con $, y los registros con %.

```
# saludo.s:    Imprimir por pantalla
#              Hola a todos!
#              Hello, World!
# retorna:     código retorno 0, programado en la penúltima línea
#              comprobar desde línea de comandos bash con echo $?

# SECCIÓN DE DATOS (.data, variables globales inicializadas)
#       datos hex, octal, binario, decimal, char, string: 0x, 0, 0b, díg<>0, ', ""
#       ejemplos: 0x41, 0101, 65, 0b01000001, 'A, "AAA"

.section .data # directivas comienzan por . (en este caso empezamos una sección)
               # no son instrucciones máquina, son indicaciones al ensamblador
               # etiquetas recuerdan el valor del contador de posiciones (bytes)
saludo:
       .ascii "Hola a todos!\nHello, World!\n"     # \n representa salto de línea

longsaludo:
       .int   .-saludo       # . = contador posiciones. Aritmética de etiquetas.

# SECCIÓN DE CÓDIGO (.text, instrucciones máquina)

.section .text               # cambiamos de sección, ahora emitimos en .text
.global _start               # muestra punto de entrada a ld (como main en C)

_start:                      # punto de entrada ASM (como main en C)

       #    Llamada al sistema WRITE, consultar "man 2 write"
       #             ssize_t  write(int fd, const void *buf, size_t count);
       mov $4, %eax          # write: servicio 4 kernel Linux
       mov $1, %ebx          #    fd: descriptor de fichero para stdout
       mov     $saludo, %ecx #   buf: dirección del texto a escribir
       mov  longsaludo, %edx # count: número de bytes a escribir
       int $0x80             # llamar write(stdout, &saludo, longsaludo);

       #      Llamada al sistema EXIT, consultar "man 2 exit"
       #              void      _exit(int status);
       mov $1, %eax          #   exit: servicio 1 kernel Linux
       mov $0, %ebx          # status: código a retornar (0=OK)
       int $0x80             # llamar exit(0);
```

Figura 10: saludo.s: ejemplo de llamadas al sistema WRITE y EXIT

En arquitectura x86 (y x86_64) cada posición de memoria es un byte. Ya vimos en la Figura 7 cómo se usó aritmética de punteros para calcular el tamaño ocupado por la función `sum`. En este caso, podríamos alterar el *string* en el código fuente ensamblador, y la variable `longsaludo` tomaría automáticamente el valor correcto para la posterior llamada a *`WRITE`*.

Un mecanismo frecuentemente usado por los Sistemas Operativos para permitir que se les realicen llamadas al sistema (*syscalls*) son las interrupciones software (que son instrucciones programadas, en oposición a las interrupciones hardware o las excepciones). El mecanismo de interrupción permite el cambio a espacio kernel (en oposición al espacio de usuario). El programador utiliza un vector concreto (`0x80` en el caso de Linux) cuando desea realizar la llamada. La subrutina de servicio espera encontrar el número de servicio en EAX y sus argumentos en sucesivos registros EBX, ECX, EDX, ESI, EDI, EBP (hasta 6) de manera que el programador debe fijar estos valores antes de realizar la interrupción `int 0x80`. Si la llamada al sistema produce un valor de retorno, lo devuelve en EAX (es decir, que se encuentra disponible en EAX tras ejecutarse `int 0x80`).

Los números de servicio (llamada) pueden encontrarse en `/usr/include/asm/unistd_32.h`, y los argumentos de cada llamada pueden conocerse leyendo la correspondiente página de manual de la sección 2 (Llamadas al Sistema). En nuestro caso, nos interesa consultar `man 2 write` y `man 2 exit` (o `info write`, `info exit`) para saber que tienen 3 (EBX…EDX) y 1 argumentos, respectivamente. El argumento de `exit(status)` es un código de retorno (se puede probar a cambiarlo en el fuente y comprobarlo con `echo $?`) mientras que `write(fd, buf, count)` escribe `count` bytes a partir de `buffer` en el descriptor de fichero `fd`. En concreto, el descriptor para la salida estándar (STDOUT_FILENO) está definido en `/usr/include/unistd.h` (también se puede comprobar listando `ls -la /dev/stdout`).

Ejercicio 4: ddd

Ensamblar y enlazar el programa `saludo.s`, incluyendo información de depuración, y reproducir la siguiente sesión ddd. Aunque usemos el front-end ddd en modo gráfico, es conveniente aprender también los comandos gdb en modo texto. Hay una lista de comandos en el enlace [6].

```
as --32 –g     saludo.s –o saludo.o  # ensamblar 32bits / información depuración
ld –m elf_i386 saludo.o –o saludo    # enlazar p/32bits
ddd saludo &                         # sesión ddd en modo gráfico
   list               # localizar la línea “mov $1,%ebx” y ponerle un breakpoint
   break 30           # equivalente en modo gráfico: cursor a izq. línea y stop
   info break         # equiv. gráf: Source->Breakpoints. Notar address 080480xx

   run                # eq.gr: Program->Run / (View->CmdTool->)botonera->Run.
   disassemble        # eq.gr: View->MachCodeWin
   print $eip         # Notar dirección break=EIP
   print $eax         # Notar EAX=4, pero EBX=0 aún
   info registers     # eq.gr: Status->Registers
   stepi              #
   p $eip             # EIP sigue avanzando (p=print)
   p $ebx             # Notar EBX=1 ahora
   si                 #                     (si=stepi)
   p/x $ecx           # Notar ECX=080490xx > EIP

   x/32cb &saludo     # eq.gr: Data->Memory->Examine 32 char bytes from &saludo
   x/32xb &saludo     # Print para probar (cambiar a hex bytes)/Display para fijo
   x/1dw &longsaludo  # Print sólo, para ver dirección de inicio y valor
   x/1xw &longsaludo  # Ver correspondencia tras final saludo
   x/4xb &longsaludo
   si                 # Comprobar registros EAX,EBX,ECX,EDX = 4,1,0x080490…,28
   info reg           # (write,stdout,&saludo,longsaludo)
   disas              # $saludo=$0x0804…(inm), longsl…=0x0804…(dir), %edx=28(reg)
   si                 # Se escribe el mensaje en la pantalla (View->GDB Console)

2x si / cont          # Pulsar cont o clickar 3 stepi para exit(0)
   set $ebx=1         # o parar justo antes del final
   stepi              # y cambiar el código de retorno sobre la marcha

   run                      # volver a empezar
   print      saludo        # interpreta string como entero 4B
   p   (char) saludo        # typecast a char 1B ‘H’
   p    (int) saludo        # comprobar que toma 4B “Hola” (4 códigos ASCII)
   p  /x(int) saludo        # como un entero 0x616c6f48
   p         &saludo        # dirección en memoria
   p  (char*)&saludo        # typecast a char*
   p  (char*)&saludo+13     # saltarse 13 letras, localizar \n
   p*((char*)&saludo+13)    # cambiarlo por ‘-’
   set var *((char*)&saludo+13)=’-’
   print     (char*)&saludo # comprobar cambio en memoria
   cont                     # comprobar cambio en ejecución
```

Figura 11: ensamblado, enlazado, y sesión de depuración usando gbd/ddd

Como vemos, la forma general de usar un depurador es escoger un punto de parada (o varios), lanzar la ejecución, comprobar valores de variables y registros cuando el programa se detenga (al encontrar algún punto de parada), y seguir ejecutando (paso a paso, continuar ejecución normal, o volver a empezar desde el principio). Las siguientes preguntas pueden servir como ejercicio de autocomprobación. Aunque algunas puedan responderse genéricamente, para proporcionar valores y resultados concretos es necesario comprobar las respuestas usando ddd., y tal vez `objdump -s/nm`.

	Sesión de depuración saludo.s
1	¿Qué contiene EDX tras ejecutar **mov longsaludo, %edx**? ¿Para qué necesitamos esa instrucción, o ese valor? Responder no sólo el valor concreto (en decimal y hex) sino también el significado del mismo (¿de dónde sale?) Comprobar que se corresponden los valores hexadecimal y decimal mostrados en la ventana Status->Registers
2	¿Qué contiene ECX tras ejecutar **mov $saludo, %ecx**? Indicar el valor en hexadecimal, y el significado del mismo. Realizar un dibujo a escala de la memoria del programa, indicando dónde empieza el programa (_start, .text), dónde empieza saludo (.data), y dónde está el tope de pila (%esp)
3	¿Qué sucede si se elimina el símbolo de dato inmediato ($) de la instrucción anterior? (**mov saludo, %ecx**) Realizar la modificación, indicar el contenido de ECX en hexadecimal, explicar por qué no es lo mismo en ambos casos Concretar de dónde viene el nuevo valor (obtenido sin usar $)
4	¿Cuántas posiciones de memoria ocupa ocupa la variable **longsaludo**? ¿Y la variable **saludo**? ¿Cuántos bytes ocupa por tanto la sección de datos? Comprobar con un volcado Data->Memory mayor que la zona de datos antes de hacer Run.

5	Añadir dos volcados Data->Memory de la variable **longsaludo**, uno como entero hexadecimal, y otro como 4 bytes hex. Teniendo en cuenta lo mostrado en esos volcados… ¿Qué direcciones de memoria ocupa **longsaludo**? ¿Cuál byte está en la primera posición, el más o el menos significativo? ¿Los procesadores de la línea x86 usan el criterio del extremo mayor (big-endian) o menor (little-endian)? Razonar la respuesta
6	¿Cuántas posiciones de memoria ocupa la instrucción **mov $1, %ebx**? ¿Cómo se ha obtenido esa información? Indicar las posiciones concretas en hexadecimal.
7	¿Qué sucede si se elimina del programa la primera instrucción **int 0x80**? ¿Y si se elimina la segunda? Razonar las respuestas
8	¿Cuál es el número de la llamada al sistema *READ* (en kernel Linux 32bits)? ¿De dónde se ha obtenido esa información?

Tabla 3: preguntas de autocomprobación (saludo.s)

4 Llamadas a funciones

Es conveniente dividir el código de un programa entre varias funciones, de manera que al ser éstas más cortas y estar centradas en una tarea concreta, se facilita su legibilidad y comprensión, además de poder ser reutilizadas si hacen falta en otras partes del programa. En la Figura 12 se muestra un programa con una función que calcula la suma de una lista de enteros de 32bits. La dirección de inicio de la lista y su tamaño se le pasa a la función a través de registros. El resultado se devuelve al programa principal a través del registro EAX.

Conocer el funcionamiento de la pila (*stack*) es fundamental para comprender cómo se implementan a bajo nivel las funciones. La pila se utiliza (en llamadas a subrutinas) para guardar la dirección de retorno, para almacenar las variables locales, y para pasar argumentos (según la convención de llamada). Las instrucciones PUSH y POP (Tabla 8) y las llamadas y retornos de subrutinas (CALL, RET, INT, IRET, Tabla 12) utilizan de forma implícita la pila, que es la zona de memoria adonde apunta el puntero de pila ESP (RSP en x86_64). La pila crece hacia direcciones inferiores de memoria, y ESP apunta al último elemento insertado (tope de pila), de manera que PUSH primero decrementa ESP en el número de posiciones de memoria que ocupe el dato a insertar, y luego escribe ese dato en las posiciones reservadas, a partir de donde apunta ESP ahora. Similarmente POP primero lee del tope de pila, guardando el valor en donde indique su argumento, y luego incrementa ESP. Por su parte, CALL guarda la dirección de retorno en pila antes de saltar a la subrutina indicada como argumento, y RET recupera de pila la dirección de retorno.

Ejecutar el programa de la Figura 12 paso a paso con ddd y responder a las siguientes preguntas de autocomprobación. Aunque algunas puedan responderse genéricamente, para proporcionar valores y resultados concretos es necesario comprobar las respuestas usando ddd.

	Sesión de depuración suma.s
1	¿Cuál es el contenido de EAX justo antes de ejecutar la instrucción RET, para esos componentes de lista concretos? Razonar la respuesta, incluyendo cuánto valen 0b10, 0x10, y (.-lista)/4
2	¿Qué valor en hexadecimal se obtiene en **resultado** si se usa la lista de 3 elementos: .int 0xffffffff, 0xffffffff, 0xffffffff? ¿Por qué es diferente del que se obtiene haciendo la suma a mano? NOTA: Indicar qué valores va tomando EAX en cada iteración del bucle, como los muestra la ventana Status->Registers, en hexadecimal y decimal (con signo). Fijarse también en si se van activando los flags CF y OF o no tras cada suma. Indicar también qué valor muestra **resultado** si se vuelca con Data->Memory como decimal (con signo) o unsigned (sin signo).
3	¿Qué dirección se le ha asignado a la etiqueta **suma**? ¿Y a **bucle**? ¿Cómo se ha obtenido esa información?
4	¿Para qué usa el procesador los registros EIP y ESP?
5	¿Cuál es el valor de ESP antes de ejecutar CALL, y cuál antes de ejecutar RET? ¿En cuánto se diferencian ambos valores? ¿Por qué? ¿Cuál de los dos valores de ESP apunta a algún dato de interés para nosotros? ¿Cuál es ese dato?
6	¿Qué registros modifica la instrucción CALL? Explicar por qué necesita CALL modificar esos registros
7	¿Qué registros modifica la instrucción RET? Explicar por qué necesita RET modificar esos registros
8	Indicar qué valores se introducen en la pila durante la ejecución del programa, y en qué direcciones de memoria queda cada uno. Realizar un dibujo de la pila con dicha información. NOTA: en los volcados Data->Memory se puede usar $esp para referirse a donde apunta el registro ESP
9	¿Cuántas posiciones de memoria ocupa la instrucción **mov $0, %edx**? ¿Y la instrucción **inc %edx**? ¿Cuáles son sus respectivos códigos máquina? Indicar cómo se han obtenido. NOTA: en los volcados Data->Memory se puede usar una dirección hexadecimal 0x… para indicar la dirección del volcado. Recordar la ventana View->Machine Code Window. Recordar también la herramienta objdump.
10	¿Qué ocurriría si se eliminara la instrucción RET? Razonar la respuesta. Comprobarlo usando ddd

Tabla 4: preguntas de autocomprobación (suma.s)

```
# suma.s:       Sumar los elementos de una lista
#               llamando a función, pasando argumentos mediante registros
# retorna:      código retorno 0, comprobar suma en %eax mediante gdb/ddd

# SECCIÓN DE DATOS (.data, variables globales inicializadas)
.section .data
lista:
        .int 1,2,10, 1,2,0b10, 1,2,0x10      # ejemplos binario 0b / hex 0x
longlista:
        .int   (.-lista)/4    # . = contador posiciones. Aritmética de etiquetas.
resultado:
        .int   -1             # 4B a FF para notar cuándo se modifica cada byte

# SECCIÓN DE CÓDIGO (.text, instrucciones máquina)
.section .text
_start: .global _start        # PROGRAMA PRINCIPAL-se puede abreviar de esta forma

        mov     $lista, %ebx  # dirección del array lista
        mov  longlista, %ecx  # número de elementos a sumar
        call suma             # llamar suma(&lista, longlista);
        mov  %eax, resultado  # salvar resultado
                              #   void _exit(int status);
        mov  $1, %eax         #   exit: servicio 1 kernel Linux
        mov  $0, %ebx         # status: código a retornar (0=OK)
        int  $0x80            # llamar _exit(0);

# SUBRUTINA:   suma(int* lista, int longlista);
# entrada:     1) %ebx = dirección inicio array
#              2) %ecx = número de elementos a sumar
# salida:         %eax = resultado de la suma

suma:
        push     %edx         # preservar %edx (se usa aquí como índice)
        mov  $0, %eax         # poner a 0 acumulador
        mov  $0, %edx         # poner a 0 índice
bucle:
        add  (%ebx,%edx,4), %eax      # acumular i-ésimo elemento
        inc        %edx               # incrementar índice
        cmp   %edx,%ecx               # comparar con longitud
        jne    bucle                  # si no iguales, seguir acumulando

        pop    %edx           # recuperar %edx antiguo
        ret
```

Figura 12: suma.s: ejemplo de llamada a subrutina (paso de parámetros por registros)

5 Trabajo a realizar

Se propone mejorar este último programa para calcular la media de una lista de N enteros, evitando la posibilidad de desbordamiento aritmético (*overflow*). Necesitaremos utilizar las instrucciones de división con signo, extensión de signo y suma con acarreo, para realizar la suma en múltiple (doble) precisión.

Según la temporización de cada curso, se procurarán realizar guiadamente (como Seminario Práctico) los Ejercicios 1-4 aproximadamente (incluso `suma.s` si sobrara tiempo). Aunque no diera tiempo a tanto, responder a las preguntas (Tabla 3, Tabla 4), comprender los programas mostrados (Figura 10, Figura 12) y ejercitarse en el uso de las herramientas son competencias que cada uno debe conseguir personalmente. Como ya se sabe crear y depurar un pequeño programa ensamblador, para aprender el funcionamiento de nuevas instrucciones basta con leer el manual y probarlas en un ejemplo.

Al objeto de facilitar el desarrollo progresivo de la práctica, se sugiere realizar en orden las siguientes tareas:

5.1 Contestar las preguntas de autocomprobación (saludo.s, suma.s)

El objetivo es comprender con detalle el funcionamiento de las instrucciones mostradas, la ubicación de código y datos en memoria, el funcionamiento de la pila y llamadas y retornos de subrutinas, y adquirir habilidad en el manejo de las herramientas usadas (compilador, ensamblador, enlazador y depurador).

5.2 Sumar N enteros *sin* signo de 32bits en una plataforma de 32bits sin perder precisión (N≈32)

Realizar un programa ensamblador con una función que sume una lista de N números en binario natural (sin signo) de 32bits, sin perder dígitos. Notar que la suma puede necesitar más de 32bits. Indicar cuántos bits adicionales pueden llegar a necesitarse para almacenarse el resultado, si N≈32. Razonar que si se hace una suma en doble precisión (32+32bits) no habrá desbordamiento para cualquier valor práctico de N.

La suma en doble precisión implica conservar los acarreos que de otra forma se hubieran perdido (instrucción etiquetada `bucle:` en `suma.s`, donde no se comprueba si hay acarreo saliente de EAX). Si los vamos acumulando en otro registro de 32bits (p. ej. EDX), la concatenación de ambos (EDX:EAX) puede almacenar un entero sin signo de 64bits. Para comprobar y/o sumar el acarreo pueden usarse la suma con acarreo ADC o el salto condicional según haya acarreo o no JC/JNC, posiblemente combinado con INC. Probablemente necesitemos usar otro registro como índice (tal vez ESI) en lugar de EDX.

Se puede tomar como punto de partida el programa `suma.s`, añadiendo directivas `.int` adicionales para los N≈32 valores (por ejemplo, 4 líneas de 8 valores) y cambiando el tamaño de `resultado` a 64bits (¿Cuál es la directiva gas para ello?). La subrutina debería devolver el resultado en EDX:EAX, y el programa principal almacenarlo en la variable `resultado` de 64bits, según el criterio del extremo menor (recordar que la familia x86 es *little-endian*). La dirección que empieza 4 posiciones detrás de `resultado` se indica `resultado+4`, naturalmente. Desde ddd debería poder visualizarse el valor correcto de la suma con Data->Memory->*Examine 1 decimal giant from &resultado*.

También se podría redactar el progama en C y observar qué código genera gcc con distintos niveles de optimización (`-O1`, `-O2`). La función se declararía como `unsigned long long suma(unsigned* lista, int longlista)`. La lista se podría inicializar con la sintaxis `unsigned lista[]={1,2 [,…] }`, y se podría imprimir el resultado con `printf("resultado=%lld\n", suma(lista,longlista))`. También se podría usar el formato "`%llx`" para ver el resultado en hexadecimal (o usar ddd).

Para saber si el resultado es correcto, se pueden probar ejemplos pequeños (todos los elementos a 1 suman 32, todos a 2 suman 64, cíclicamente 1,2,3,4 suman 80, etc.) pero es costumbre usar ejemplos que ejerciten todas las posibilidades del código. En este caso interesan ejemplos que produzcan acarreos, incluso varios acarreos, hasta uno por cada suma.

	Cuestiones sobre suma64unsigned.s
1	Para N≈32, ¿cuántos bits adicionales pueden llegar a necesitarse para almacenar el resultado? Dicho resultado se alcanzaría cuando todos los elementos tomaran el valor máximo sin signo. ¿Cómo se escribe ese valor en hexadecimal? ¿Cuántos acarreos se producen? ¿Cuánto vale la suma (indicarla en hexadecimal)? Comprobarlo usando ddd.
2	Si nos proponemos obtener sólo 1 acarreo con una lista de 32 elementos iguales, el objetivo es que la suma alcance 2^{32} (que ya no cabe en 32bits). Cada elemento debe valer por tanto $2^{32}/32 = 2^{32}/2^{?} = ?$. ¿Cómo se escribe ese valor en hexadecimal? Inicializar los 32 elementos de la lista con ese valor y comprobar cuándo se produce el acarreo.
3	Por probar valores intermedios: si la lista se inicializara con los valores 0x10000000, 0x20000000, 0x40000000, 0x80000000, repetidos cíclicamente, ¿qué valor tomaría la suma de los 32 elementos? ¿Cuándo se producirían los acarreos? Comprobarlo con ddd.

Tabla 5: preguntas de autocomprobación (suma64uns.s)

5.3 Sumar N enteros *con* signo de 32bits en una plataforma de 32bits

Modificar el programa anterior para que los números de la lista se consideren como enteros con signo. Correspondientemente, la variable de 64bits en donde se almacene el resultado también será con signo.

En lenguaje C la declaración de la función cambiaría a `long long suma(int* lista, int longlista)`, y la de los datos a `int lista[]={1,2 [,…] }`. Si se redacta el programa C (para observar qué código se genera), comparar el resultado de ambas versiones, con y sin signo, para una lista sencilla como por ejemplo {-1,-1,-1,-1}. El resultado del programa con signo debe ser -4 (0xffffffff fffffffc), mientras que sin signo se obtiene casi 16Giga (=0x00000003 fffffffc). La diferencia está en que no se realiza extensión de signo 32→64bits para sumar números sin signo. Si aún no se ve claro, otro ejemplo más sencillo sería {0,-1}, en donde se obtiene 0x00000000 ffffffff= 2^{32}-1 = 4Giga-1 sin signo, y 0xffffffff ffffffff=-1 con signo.

Tomando como partida el programa ensamblador del apartado anterior (en donde -1+0≠-1 porque -1 se interpreta sin signo), las instrucciones de extensión de signo que nos interesan no son las de tipo MOVZX (porque no queremos usar los registros de 64bits en una plataforma de 32bits), sino las de tipo CBW (Tabla 10). Probablemente querremos *leer (**no** sumar)* el elemento en EAX, extender a 64bits, sumar esos 64bits con otros 2 registros usados como acumulador (se puede usar EBP si fuera necesario), y antes de retornar, mover la suma acumulada de 64bits a EDX:EAX, que es donde la espera el programa principal.

Para saber si el resultado es correcto, se pueden probar ejemplos pequeños (todos los elementos a -1 suman -32, 32 parejas 1,-2 suman -16, cíclicamente 1,2,-3,-4 suman -32, etc.) pero en las preguntas de autocomprobación sugerimos casos con más posibilidades de detectar algún problema en nuestro programa, si lo hubiera.

	Cuestiones sobre suma64signed.s
1	¿Cuál es el máximo entero positivo que puede representarse (escribirlo en hexadecimal)? Si se sumaran los N≈32 elementos de la lista inicializados a ese valor ¿qué resultado se obtendría (en hexadecimal)? ¿Qué valor aproximado tienen el elemento y la suma (indicarlo en múltiplos de potencias de 10)? Comprobarlo usando ddd.
2	Misma pregunta respecto a negativos: menor valor negativo en hexadecimal, suma, valores decimales aprox., usar ddd.
3	Si nos proponemos obtener sólo 1 acarreo con una lista de 32 elementos positivos iguales, *se podría pensar que* el objetivo es que la suma alcance 2^{31} (que ya no cabe en 32bits *como número positivo en complemento a dos*). *Aparentemente*, cada elemento debe valer por tanto $2^{31}/32 = 2^{31}/2^{?} = ?$. ¿Cómo se escribe ese valor en hexadecimal? Inicializar los 32 elementos de la lista con ese valor y comprobar si se produce el acarreo.
4	Repetir el ejercicio anterior de forma que sí se produzca acarreo desde los 32bits inferiores a los superiores. ¿Cuál es el valor de elemento requerido? ¿Por qué es incorrecto el razonamiento anterior? Indicar los valores decimales aproximados (múltiplos de potencias de 10) del elemento y de la suma. Comprobarlo usando ddd.
5	Respecto a negativos, -2^{31} sí cabe en 32bits como número negativo en complemento a dos. Calcular qué valor de elemento se requiere para obtener como suma -2^{31}, y para obtener -2^{32}. Comprobarlo usando ddd.
6	Por probar valores intermedios: si la lista se inicializara con los valores 0xF0000000, 0xE0000000, 0xE0000000, 0xD0000000, repetidos cíclicamente, ¿qué valor tomaría la suma de los 32 elementos (en hex)? Comprobarlo con ddd.

Tabla 6: preguntas de autocomprobación (suma64sgn.s)

5.4 Media de N enteros *con* signo de 32bits, en plataforma de 32bits

Modificar el programa anterior para que calcule la media de la lista de números. La instrucción de división a añadir (Tabla 10) viene obligada, teniendo en cuenta que se trata de números con signo (¿cuál debe ser?). Por tanto también viene obligado en qué registros debe ponerse el dividendo, y en dónde quedará el cociente (¿dónde?). El resultado final ya no es de 64bits.

En lenguaje C la declaración de la función cambiaría a `int media(int* lista, int longlista)`, y la función podría acabar con `return suma/longlista` si se ha acumulado en `long long suma=0`.

Para saber si el resultado es correcto, se pueden probar todos los ejemplos sugeridos anteriormente. En la mayoría de ellos la lista repite 32 veces un mismo elemento, que termina siendo el valor de la media (en el caso cíclico 1,2,-3,-4, la media es -1). En las preguntas de autocomprobación sugerimos esta vez casos más sencillos.

	Cuestiones sobre media.s
1	Rellenando la lista al valor -1, la media es -1. Cambiando un elemento a 0, la media pasa a valer 0. ¿Por qué? Consultar el manual de Intel sobre la instrucción de división. ¿Cuánto vale el resto de la división en ambos casos? Probar con ddd.
2	También se obtiene 0 si se cambia lista[0]=1, 2, 3… Para facilitar el cálculo mental, podemos ajustar lista[1]=-2, y así la suma vale lista[0]-32, resultando más fácil calcular el resto. ¿Para qué rango de valores de lista[0] se obtiene cociente 0? ¿Cuánto vale el resto a lo largo de ese rango? Comprobar que coinciden los signos del dividendo (suma) y del resto. NOTA: Para evitar el ciclo editar-ensamblar-enlazar-depurar, se pueden poner un par de breakpoints antes y después de llamar la subrutina que calcula la media. Tras encontrar el primer breakpoint se puede modificar lista[0] con el comando **set var lista=<valor>**. Pulsar Cont para llegar al segundo breakpoint y ver en EAX el resultado retornado por la subrutina. Dependiendo de si se restauran los valores de los otros registros antes de retornar, es posible que sea ventajoso poner el segundo breakpoint antes de POP EDX para ver el valor del resto al mismo tiempo que el cociente. Para hacer muchas ejecuciones seguidas, puede merecer la pena (re)utilizar la línea de comandos (run/set var…/cont) en lugar del ratón.
3	¿Para qué rango de valores de lista[0] se obtiene media **1**? ¿Cuánto vale el resto en ese rango? Comprobarlo con ddd, y notar que tanto las sumas como los restos son positivos (el cociente se redondea hacia cero).
4	¿Para qué rango de valores de lista[0] se obtiene media **-1**? ¿Cuánto vale el resto en ese rango? Comprobarlo con ddd, y notar que tanto las sumas como los restos son positivos (el cociente se redondea hacia cero).

Tabla 7: preguntas de autocomprobación (media.s)

6 Entrega del trabajo desarrollado

Los distintos profesores de teoría y prácticas acordarán las normas de entrega para cada grupo, incluyendo qué se ha de entregar, cómo, dónde y cuándo. Por ejemplo, puede que en un grupo se deba entregar en un documento PDF las respuestas a las preguntas de autocomprobación (Tabla 3-Tabla 7), los listados de los programas ensamblador (5.2, 5.3, 5.4), y unos breves comentarios sobre por qué se realizó cada modificación (instrucciones añadidas o eliminadas, registros cambiados, etc.) partiendo del programa original `suma.s`, subiéndolo al SWAD hasta 3 días después de la última sesión de prácticas dedicada a esta práctica, con penalización creciente por entrega tardía hasta 1 semana posterior.

Puede que en otro grupo se pueda trabajar y entregar por parejas, pero que el profesor de prácticas visite cada puesto al final de cada sesión comprobando si ambos estudiantes saben responder a las preguntas, programar en ensamblador y utilizar las herramientas, permitiendo que se suba al SWAD el trabajo en caso afirmativo. Puede que en otros grupos se deba subir también el código programado en lenguaje C y explicar si ha servido o no de ayuda ver el código generado por gcc. Los profesores de teoría y prácticas de cada grupo acordarán cómo entregará ese grupo el trabajo desarrollado.

7 Bibliografía

[1] Apuntes y presentaciones de clase

[2] Jonathan Bartlett, “Programming from the Ground Up”. Ed. Dominick Bruno, Jr.
http://savannah.nongnu.org/projects/pgubook/

[3] Manuales de Intel sobre IA-32 e Intel64, en concreto el volumen 2: “Instruction Set Reference”
http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-software-developer-vol-2a-2b-instruction-set-a-z-manual.pdf

[4] Resumen del repertorio de instrucciones básico x86 (mencionado en transparencias clase)
http://www.jegerlehner.ch/intel/IntelCodeTable.pdf
http://www.jegerlehner.ch/intel/IntelCodeTable_es.pdf

[5] GNU binutils, artículo Wikipedia: http://en.wikipedia.org/wiki/GNU_Binutils
Sitio web http://www.gnu.org/software/binutils/
Manuales (incluyendo gas, ld, nm…) http://sourceware.org/binutils/docs/

[6] GNU Debugger, GDB, Wikipedia http://en.wikipedia.org/wiki/Gdb
Sitio web http://www.gnu.org/s/gdb/
Manuales http://sourceware.org/gdb/current/onlinedocs/gdb/
Chuletario http://www.csd.uoc.gr/~hy255/refcards/gdb-refcard.pdf
Resumen comandos http://web.cecs.pdx.edu/~jrb/cs201/lectures/handouts/gdbcomm.txt

[7] Data Display Debugger, DDD, Wiki http://en.wikipedia.org/wiki/Data_Display_Debugger
Sitio web http://www.gnu.org/s/ddd/
Manuales (incluye tutorial) http://www.gnu.org/s/ddd/manual/html_mono/ddd.html
http://www.gnu.org/s/ddd/manual/html_mono/ddd.html#Sample%20Session

[8] Código ASCII, Wikipedia http://en.wikipedia.org/wiki/ASCII
Código UTF-8 http://en.wikipedia.org/wiki/UTF-8
Tabla ASCII http://en.wikipedia.org/wiki/File:ASCII_Code_Chart-Quick_ref_card.png

Apéndice 1. Tabla de caracteres ASCII

En algún momento (Figura 11) se han interpretado 4 bytes de un *string* como un entero. Para comprobar que el resultado obtenido no es en absoluto impredecible, sino que es matemáticamente preciso, se proporciona la tabla de códigos ASCII de 7bits en la que están presentes todos los caracteres utilizados en aquel *string*.

	0	1	2	3	4	5	6	7
0	NUL	DEL	SP	0	@	P	‘	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	“	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	’	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Por ejemplo, si un programa definiera un *string* “Hello”, y posteriormente el *string* se interpretara como entero, el valor entendido por el procesador dependería del tamaño de entero (usualmente la longitud de palabra) y el ordenamiento de bytes (si la memoria está organizada en bytes). Los procesadores IA-32 siguen la convención del extremo menor, así que el *string* interpretado como entero de 4B sería 0x6c6c6548. Observar cómo se toman las primeras cuatro letras “Hell”, y el byte menos significativo es el que está almacenado en la posición de memoria más baja “H”.

Apéndice 2. Resumen de la arquitectura de los repertorios x86 y x86_64, y del ensamblador GNU AS.

Los 8 registros x86 de propósito general se pueden acceder en tamaño byte (AH…BL), palabra de 16bits (AX…BP, para modo 16bits), y doble palabra de 32bits (EAX…EBP). Hay un registro de flags (EFLAGS) y un contador de programa (EIP).

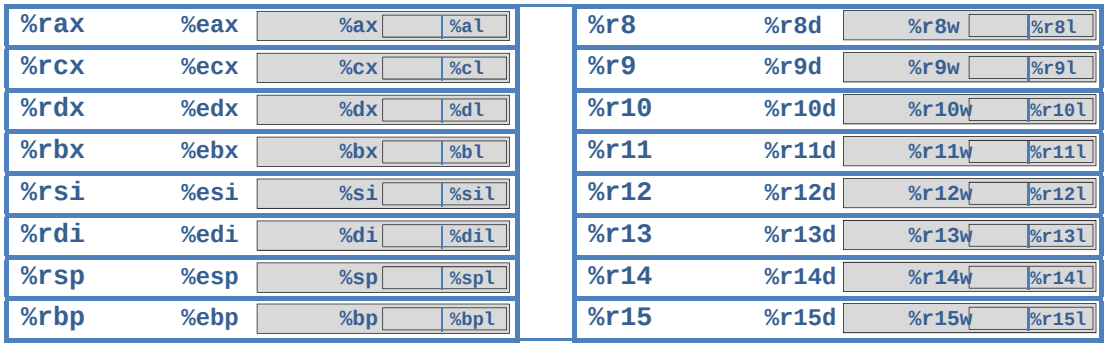
Registros enteros IA-32				nombre
%eax	%ax	%ah	%al	acumulador
%ecx	%cx	%ch	%cl	contador
%edx	%dx	%dh	%dl	datos
%ebx	%bx	%bh	%bl	base
%esi	%si			índice fuente
%edi	%di			índice destino
%esp	%sp			puntero pila
%ebp	%bp			puntero base

También existen registros para datos en punto flotante de 64/80bits (FP0…FP7), registros MMX de 64bits (MM0…MM7), registros SSE de 128 bits (XMM0…XMM7), etc, que se usan con instrucciones específicas del repertorio.

Existen también registros de segmento (CS,SS,DS,ES,FS,GS, aunque en Linux se usa un modelo de memoria plano, no segmentado). Y registros de control (CRi), depuración (DRi), específicos (MSRi), etc.

Los 16 registros x86_64 de propósito general son de tamaño cuádruple palabra (RAX…RBP + R8…R15), y también puede accederse en tamaños menores: 32bits (EAX…EBP + R8D…R15D), 16bits (AX…BP + R8W…R15W), 8 bits (AL…BL + SIL…BPL + R8L…R15L). RFLAGS y RIP también son de 64bits.

Registros enteros x86-64



Se muestra a continuación un brevísimo resumen de las instrucciones más frecuentemente usadas (en sintaxis Intel como aparecen en el manual, y con ejemplos AT&T como suelen utilizarse en Linux). Recordar que, como norma general, en las instrucciones con 2 operandos sólo 1 puede ser memoria. Siempre se pueden consultar los manuales de Intel para comprobar qué modos de direccionamiento son válidos para cada argumento, y qué flags de estado resultan afectados por cada instrucción.

Instrucciones de movimiento de datos

Instrucción		Efecto	Descripción	Ejemplos
MOV	D, S	D ← S	Mover fuente a destino (mismo tamaño). Variantes AT&T: tamaño byte, word, long…	movb %al, %bl movl $0xf02, %ecx
MOVS	D, S	D ← ExtSigno(S)	Mover con extensión de signo (aumentando tamaño) Variantes AT&T: byte-to-word, byte-long, word-long…	movsbw %al, %bx movslq %ecx, %rdx
MOVZ	D, S	D ← ExtCero(S)	Mover rellenando con ceros (aumentando tamaño) movzbw, movzbl, movzwl…	movzbq %al, %rbx movzwl %cx, %edx
PUSH	S	ESP ← ESP-|S| M[ESP] ← S	Meter fuente en pila	pushl $4
POP	D	D ← M[ESP] ESP ←ESP+|S|	Sacar de pila a destino	popl %ebp
LEA	D, S	D ← &S	Cargar dirección efectiva	leal tabla(,%eax,4), %esi

Tabla 8: Instrucciones x86 - transferencia

Observar que la sintaxis AT&T intercambia el orden de los argumentos (S, D) e incorpora un sufijo con el tamaño de operando (`byte`, `word`, `long`) o incluso de la conversión realizada (`byte-to-long`…). Notar que los datos inmediatos se prefijan con $, los registros con %, y la sintaxis para direccionamiento a memoria es `Desplaz(Rbase,Rindice,FactEscala)`, pudiendo omitirse los componentes no deseados.

Instrucciones aritmético-lógicas

Instrucción		Efecto	Descripción	Instrucción		Efecto	Descripción
INC	D	D ← D + 1	Incrementar	NEG	D	D ← -D	Negar (aritmético)
DEC	D	D ← D - 1	Decrementar	NOT	D	D ← ~D	Complementar (lógico)
ADD	D, S	D ← D + S	Sumar	XOR	D, S	D ← D ^ S	O-exclusivo
SUB	D, S	D ← D - S	Restar	OR	D, S	D ← D | S	O lógico
ADC	D, S	D ← D+S+C	Sumar con acarreo	AND	D, S	D ← D & S	Y Lógico
SBB	D, S	D ←D-(S+C)	Restar con débito	IMUL	D, S	D ← D * S	Multiplicar
SAL	D, k	D ← D << k	Desplazamiento a izq.	RCL	D, k	D ← D $\circlearrowright_C$ k	Rotación izq. incl. acarreo
SHL	D, k	D ← D << k	Desplazamiento a izq.	ROL	D, k	D ← D $\circlearrowright$ k	Rotación a izquierda
SAR	D, k	D ← D$>>_A$ k	Desplaz. aritm. derecha	RCR	D, k	D ← D $\circlearrowleft_C$ k	Rotación der. incl. acarreo
SHR	D, k	D ← D $>>_L$ k	Desplaz. lógico derecha	ROR	D, k	D ← D $\circlearrowleft$ k	Rotación a derecha

Tabla 9: Instrucciones x86 - aritmético-lógicas

No confundir la negación aritmética con el complemento lógico. Recordar que los desplazamientos a derecha pueden ser aritméticos o lógicos según se inserten ceros o copias del bit de signo. Las rotaciones pueden incluir o no el acarreo en el conjunto de bits a desplazar. La instrucción `IMUL` es particular, consultar en el manual de Intel sus diversos modos de direccionamiento (1, 2, 3 operandos).

Instrucciones aritméticas especiales

Instrucción		Efecto	Descripción	Operandos / Variantes	
MUL	S	$Ac_{D:A} \leftarrow Ac_A * S$	Multiplicación sin signo	AX ← AL*r/m8	EDX:EAX ← EAX*r/m32
				DX:AX ← AX*r/m16	RDX:RAX ← RAX*r/m64
DIV	S	$Ac_A \leftarrow Ac_{D:A} / S$ $Ac_D \leftarrow Ac_{D:A} \% S$	División sin signo	AL(AH)← AX / r/m8	EAX(EDX)←EDX:EAX / r/m32
				AX(DX)←DX:AX / r/m16	RAX(RDX)←RDX:RAX / r/m64
IMUL	S	$Ac_{D:A} \leftarrow Ac_A * S$	Multiplicación con signo	1 operando: Como MUL, nbit x nbit = 2nbits	
	Dr,S	D ← D * S		2 operandos: Reg * (R/M/Inmediato), n x n = nbits	
	Dr,S,I	D ← D*S*Inm		3 operandos: Reg = Reg* (R/M/Inm), n x n = nbits	
IDIV	S	$Ac_A \leftarrow Ac_{D:A} / S$ $Ac_D \leftarrow Ac_{D:A} \% S$	División con signo	Como DIV	
CBW		$Ac_{2n} \leftarrow ExtSign(Ac_n)$	Extensión de signo acum. word-to-long-to-quad	AX ← ExtSigno(AL)	cbtw
CWDE				EAX ← ExtSigno(AX)	cwtl
CDQE				RAX ← ExtSigno(EAX)	cltq
CWD		$Ac_{D:A} \leftarrow ExtSign(Ac_A)$	Extensión de signo acum. x-to-double	DX:AX ← ExtSigno(AX)	cwtd
CDQ				EDX:EAX ← ExtSigno(EAX)	cltd
CQO				RDX:RAX ← ExtSigno(RAX)	cqto

Tabla 10: Instrucciones x86 - aritmética especial

En general los productos y divisiones usan implícitamente los registros A y D del tamaño deseado para multiplicar A(nbits) x S(nbits) = D:A(2nbits), o dividir D:A(2n) / S(nbits) produciendo cociente y resto en A y D(nbits). Las extensiones de signo en acumulador (registros A y D) pueden redactarse en sintaxis Intel o AT&T, gas entiende ambas versiones. En modo 32bits será `cltd` la que probablemente usemos más.

Comparaciones y saltos/ajustes/movimientos condicionales

Instrucción		Cálculo	Efecto	Instrucción		Efecto	Descripción
CMP	S_1, S_2	S_1 - S_2	Ajustar flags según cálculo	SETcc	D (r/m8)	D ← 0/1 según cc se cumpla o no	Códigos: e,ne,z,nz (Z) s,ns (S) o,no (O), c,nc (C), p,np (P) [n]a[e],[n]b[e] (sin signo), [n]l[e], [n]g[e] (con signo)
TEST	S_1, S_2	S_1 & S_2	Ajustar flags según cálculo	CMOVcc	Dreg, S	D ← S según cc	Mismos códigos
JMP	label		Salto incondicional directo Intel: "relative short/near"	Jcc	label	Saltar label según cc	Mismos códigos
jmp	*Ptr	(AT&T)	Salto incondic. Indirecto "near, absolute, indirect"				
JMP	PtrDst	(Intel)					

Tabla 11: Instrucciones x86 - comparaciones y condicionales

Las comparaciones y tests permiten calcular qué condiciones se cumplen entre los 2 operandos (recordar que AT&T invierte el orden de los operandos), y posteriormente se puede (des)activar un byte (`SETcc`), realizar o no un movimiento (`CMOVcc`), o saltar a una etiqueta del programa (`Jcc`) según alguna de esas condiciones. Las condiciones "cc" pueden referirse a flags sueltos como (Z)ero, (S)ign, (O)verflow, (C)arry, (P)arity, y a combinaciones interpretadas sin signo (Above/Below) y con signo (Less/Greater). En sintaxis AT&T, `jmp *%eax` sería saltar a la dirección indicada en el registro, y `jmp *(%eax)` sería leer la dirección de salto de memoria, donde apunta `%eax`.

Otras instrucciones de control de flujo, y miscelánea

Instrucción		Efecto	Descripción	Instrucción		Efecto	Descripción
CALL	label	PUSH eip eip ← label	Llamada a subrutina Intel: "near relative"	INT	v	PUSH eflags CALL ISR#v	Llamada a ISR Interrupción "Interrupción software"
call	*Ptr	(AT&T)					
CALL	PtrDst	(Intel)	Intel: "near absol. indirect"				
RET		POP eip	Retorno de subrutina	IRET		RET POP eflags	Retorno de ISR
LEAVE		ESP ← EBP POP EBP	Libera marco pila Para usar con ENTER	NOP		No-op	
CLC		C ← 0	Ajustes del flag acarreo	CLI		I ← 0	Ajustes del flag Interrupt (des)habilitar IRQs
STC		C ← 1		STI		I ← 1	
CMC		C ← ~C					

Tabla 12: Instrucciones x86 - control y miscelánea

El registro de flags (EFLAGS en 32bits, RFLAGS en 64bits) contiene (entre muchos otros) los bits aritméticos C(arry), O(verflow), S(ign), Z(ero), un bit P(arity) ajustado a paridad impar, y un bit de habilitación I(nterrupt).

Registro EFLAGS, algunos bits interesantes

31	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					OF	DF	IF	TF	SF	ZF				PF		CF

Recordar también que en Linux se usa preferentemente sintaxis AT&T, distinta a la usada por Intel en sus manuales. Las diferencias más significativas se destacan en el manual de AS en §9.13:

Diferencia	Intel	AT&T

Constantes decimal, hex, octal, binario
Constantes char, string
Prefijo de valor inmediato
	`65,41h,101o,01000001b`
	`'A', "Hello"`
	`-65`
		`65,0x41,0101,0b01000001`
		`'A, "Hello"`
		`$-65`

Prefijo de registro	`eax`	`%eax`
Orden de operandos fuente y destino	`mov ebx, 0f02h`	`movl $0xf02, %ebx`
Dirección de variable	`mov ecx, offset var`	`movl  $var, %ecx`

Sufijos/Directivas de tamaño de operandos
AT&T: b, w, l, q
Intel: (byte/word/dword/qword) ptr
	`mov ah,byte ptr [ebx]`
	`mov cx, dx`
	`mov esi, edi`
	`mov rbp, rsp`
		`movb (%ebx), %ah`
		`movw %dx, %cx`
		`movl %edi, %esi`
		`movq %rsp, %rbp`

Sintaxis modos direccionamiento
Despl=constante o nombre variable (su dirección)
Rb/Ri registros base/índice. S factor escala=1, 2, 4, 8
	`Despl[Rb+Ri*s]`
	`mov edx, array[eax*4]`
	`mov eax,[ebp-4]`
		`Despl(Rb,Ri,s)`
		`movl array(,%eax,4),%edx`
		`movl -4(%ebp), %eax`

Prefijo transferencias control (JMP/CALL) absolutas	`jmp  eax` `call tabla[esi*4]`	`jmp  *%eax` `call *tabla(,%esi,4)`

Mnemotécnicos transferencias control lejanas inm.	`jmp  far sect:offset`
	`call far sect:offset`
	`ret  far 4`
		`ljmp  $sect,$offset`
		`lcall $sect,$offset`
		`lret $4`

Extensiones con y sin signo (MOVSX, MOVZX)
AT&T: Sufijos bw, bl, bq, wl, wq, (lq sólo con movs)
y sin sufijo X
	`movsx  bx,  al`
	`movzx ebx,  al`
	`movsx rbx,  al`
	`movzx ecx,  bx`
	`movsx rcx,  bx`
	`mov   rdx, ecx`
		`movsbw  al,  bx`
		`movzbl  al, ebx`
		`movsbq  al, rbx`
		`movzwl  bx, ecx`
		`movswq  bx, rcx`
		`movsdq ecx, rdx`

Extensiones de signo en acumuladores	`cbw; cwde; cdqe` `cwd; cdq;  cqo`	`cbtw; cwtl; cltq` `cwtd; cltd; cqto`

Tabla 13: diferencias ensambladores Intel - AT&T

Por último, destacar de la sección §7 del manual de AS las directivas más comúnmente utilizadas:

Directiva	Utilidad
.ascii *"string" [, ...]*	Ensambla cada string (pasando a código ASCII cada letra) en direcciones consecutivas
.byte *expr [, ...]*	Ensambla cada expresión en el siguiente byte (dirección)
.word *expr [, ...]* **.short** *expr [, ...]*	Emite los números indicados (enteros de 2bytes) en direcciones consecutivas
.long *expr [, ...]* **.int** *expr [, ...]*	Emite los números indicados (enteros de 4bytes) en direcciones consecutivas
.quad *bignum [, ...]*	Emite los números indicados (enteros de 8bytes) en direcciones consecutivas
.equ *symbol, expr* **.set** *symbol, expr*	Asigna a *symbol* el valor *expr* (numérico o carácter). En sucesivas apariciones, se sustituye el símbolo por el valor.
.global *symbol* **.globl** *symbol*	Hace *symbol* visible desde `ld`. Si *symbol* se define en este módulo, se podrá referenciar desde otros módulos enlazados con éste. Si no, este módulo puede usar la definición de otro módulo.
.p2align *pow [,fill [,max]]*	Salta el contador de posiciones hasta próxima posición múltiplo de 2^{pow}. Las posiciones saltadas se rellenan con *fill*. Si hubiera que saltar más de *max* posiciones, se cancela el alineamiento.
.att_syntax *[[no]prefix]* **.intel_syntax** *[[no]prefix]*	Conmutar a sintaxis AT&T o Intel a partir del punto donde aparece la directiva. Se puede escoger si los registros deben llevar prefijo % o no.
.section *name [,flags[,@.]]* **.text** **.data**	Continúa ensamblando en la sección llamada *name*. Son secciones comunes: `.text` (para código) `.data` (para datos) y `.bss` (datos globales sin inicializar, `.bss` se inicializa a cero). `.text` y `.data` son tan habituales que se puede omitir `.section`.
.type *name, @type*	Marca (para `ld`) el símbolo *name* como objeto o función *(@function, @object)*
.size *name, expr*	Fija el tamaño asociado con el símbolo *name*. Se puede usar aritmética de etiquetas incluyendo el contador de posiciones . en la *expr*.

Tabla 14: directivas GNU AS

Con estas directivas sobra para redactar nuestros programas de prácticas. Si leyendo algún ensamblado gcc nos apareciera alguna directiva que ignoráramos, siempre podríamos consultarla en el manual de as.