Práctica 2.- Programación ensamblador x86-64 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.
- Hacer llamadas al sistema operativo (*kernel* Linux) desde ensamblador x86-64.
- Enumerar los registros e instrucciones más usuales de los procesadores de la línea x86-64.
- Usar con efectividad un depurador como `gdb`/`ddd` para ver los registros, ejecutar paso a paso y con puntos de ruptura, desensamblar el código, y volcar el contenido de la pila y los datos.
- Reconocer la utilidad de entornos integrados de desarrollo como Eclipse para editar código fuente, compilar programas C, ensamblar programas ASM y depurar los respectivos ejecutables.
- Argumentar la utilidad de los depuradores para ahorrar tiempo de depuración, y reconocer cómo estas herramientas permiten familiarizarse con la arquitectura del computador.
- Explicar la gestión de pila en procesadores x86-64.
- Recordar y practicar en una plataforma mixta de 32-64 bits la representación de distintos tipos de datos (caracteres, números naturales, enteros en complemento a dos), y el funcionamiento de diversas operaciones (incluyendo suma entera en doble precisión y división entera).

2 Herramientas de prácticas

Las prácticas se realizarán en Linux utilizando las herramientas GNU. Opcionalmente podrán usarse ddd y/o el entorno Eclipse, aunque la configuración avanzada del mismo no es objetivo de esta asignatura, y los guiones asumirán que se usa algún editor (`gedit`, `vi`, …) y el interfaz usuario texto de gdb. Usaremos `gcc` para compilar (traducir fuente C a ensamblador, objeto o ejecutable), `as` para ensamblar (traducir fuente ensamblador a objeto), `ld` para enlazar (combinar varios ficheros objeto en un único fichero ejecutable), y para depurar usaremos `gdb` (con `-tui` o a través del *front-end* ddd o del entorno Eclipse).

En la Figura 1 se pueden ver las herramientas que se deben ejecutar para obtener un fichero 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.

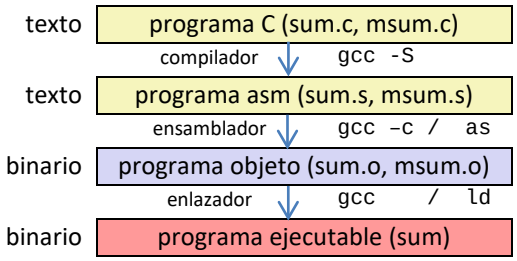


Figura 1: proceso de compilación

El código fuente se puede crear con cualquier editor de texto, como por ejemplo `gedit`, procurando 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, a partir de estos dos ficheros fuente en lenguaje C…

```
long plus(long, long);

void sumstore(long x, long y,
              long *dest) {
  long t = plus(x, y);
  *dest = t;
}
```

```
void sumstore(long x, long y,
              long *dest);
int main() {
    long d;
    sumstore(2, 3, &d);
    return d;
}
```


Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/

Figura 2: fichero fuente sum.c

```
long plus(long a, long b) {
    long s = a + b;
    return s;
}
```

Figura 3: fichero fuente msum.c

…usando gcc se podría producir el ejecutable `sum` 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 ensamblador, 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**	**significado**	**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		aSsembly	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.ext	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; Eclipse usa nivel 3)
-O	0…3, s, g		Optimizar para velocidad (0 no optimización…3 agresivo) o tamaño (s, size). Poner –O = –O1. No poner nada = –O0. Para depurar usar –Og mejor.
-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 compilación
-###			Como –v, pero no ejecuta los comandos. Es más fácil de leer.
-fno-stack-protector **-fno-asynchronous-unwind-tables** **-fno-omit-frame-pointer -no-pie** **-fno-if-conversion -fno-tree-ch** **-fno-reorder-blocks -fno-tree-ter**			Versiones recientes de gcc añaden protecciones o info.debug y dificultan leer el código ASM generado. Estos switches quitan dos de esas opciones. Versiones recientes de gcc optimizan demasiado y generan código distinto al mostrado en transparencias en clase. Estos switches permiten quitar la optimización concreta y reproducir el código explicado en clase.
as			
-g		debuG	mismo sentido
-o	fichero.o	Output	mismo sentido
--32 **--64**			mismo sentido
ld			
-L	dir	LibraryDir	mismo sentido
-l	name	LibraryName	mismo sentido
-m	elf_i386 elf_x86_64	Machine	mismo sentido
-M		printImpMap	Imprimir mapa de enlazado, posición en memoria de objetos, símbolos, etc.

Tabla 1: modificadores para gcc, as, ld

Ejercicio 1: gcc

Crear los ficheros sum.c y msum.c mostrados anteriormente (Figura 2, Figura 3), y reproducir con ellos la siguiente sesión en línea de comandos Linux. Notar cómo los modificadores `–fno-stack-protector` y `-fno-asynchronous-unwind-tables` se anotan en variables de entorno para ahorrar tecleado.

```
gcc         sum.c msum.c -o sum  # compilar de una vez
./sum ; echo $?                  # muestra cód. ret. 5 = 2+3
file sum                         # ELF 64-bit LSB shared object

gcc -no-pie sum.c msum.c -o sum  # position-indep. añadido recientemente
./sum ; echo $?                  # quitar para generar ejecutable normal
file sum                         # ELF 64-bit LSB executable
```


Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

```
FNSP=-fno-stack-protector           # ahorrarse teclear switches tan largos
FNAUT=-fno-asynchronous-unwind-tables

gcc -Og -S  sum.c $FNAUT          # crea  sum.s (-CallFrameInfo)
       cat  sum.s                 # rax res, rbx s-invcado, rdi/si/dx args
gcc -Og -S msum.c $FNAUT $FNSP    # crea msum.s (-StackProtector)
       cat msum.s                 # 8(%rsp) es var.local d

gcc -c      sum.s msum.c          # crea sum.o, msum.o desde ASM/C
ls; file *.o                      # ELF 64-bit LSB relocatable

gcc -no-pie sum.o msum.o -o sum   # crea exe sum(no a.out) desde objetos
file sum; ./sum; echo $?          # ELF 64-bit LSB executable - funciona
```

Figura 4: compilación, ensamblado y enlazado, usando gcc

Observar que el código ensamblador x86-64 menciona tanto registros de 32 bits (p.ej. en `main` se usan ESI, EDI, EAX en la secuencia `movl $3,%esi / movl $2, %edi / call sumstore / movl 8(%rsp), %eax`) como registros de 64 bits (p.ej. en `sumstore` se usan RBX, RDX, RAX en la secuencia `pushq %rbx / movq %rdx, %rbx / call plus / movq %rax, (%rbx)`). En los comandos del *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 (`as` y `ld`, limitando el uso de `gcc` a compilar C→ASM). Reproducir la siguiente sesión de comandos Linux. Notar que el último comando `ld` es tan largo que se ha fraccionado en dos líneas, usando el *escape* \↵ (*backslash*-<Enter>), que indica al *shell bash* que se debe ignorar el salto de línea y considerar la siguiente línea una continuación de la actual. En caso de duda, teclear todo el comando en una sola línea, en lugar de aplicar *escape* al salto de línea.

```
rm    sum *sum.[os]; ls               # limpiar ficheros producidos
gcc -Og -S sum.c msum.c $FNAUT $FNSP  # compilar: ya no necesitamos gcc
ls        *sum.?                      # seguir trabajando con [m]sum.s

as  sum.s -o  sum.o       # ensamblar creando objeto sum.o (no a.out)
as msum.s -o msum.o
ls    *.o; file *.o

ld       sum.o msum.o     # warn: falta _start
                          # _start está definido en crt1.o, verlo con nm
gcc -### sum.o msum.o     # una forma de ver cómo enlaza gcc
gcc -### -no-pie  *.o     # reproducir último paso collect2 usando ld:
ls /lib64                 # : /usr/lib/x86*/crt?.o -lc –dynamic-linker …
ls /usr/lib/x86_64-linux-gnu/*crt*  # otros progs pudieran necesitar crtbegin/end
                          # backslash \ es "escape" para <Enter> en bash
ld sum.o msum.o -o sum    -dynamic-linker /lib64/ld-linux-x86-64.so.2 \↵
                          /usr/lib/x86_64-linux-gnu/crt?.o -lc
file sum; ./sum; 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. En este caso (gcc 7.3.0 Ubuntu 18.04.1) bastó con enlazar contra *libC* (*switches* `-lc` y `–dynamic-linker`) y añadir tres ficheros de *runtime* presentes en `/usr/lib/x86_64-linux-gnu` (`crt1.o`, `crti.o` y `crtn.o`). Otros programas podrían necesitar los *runtime* `crtbegin.o` y `crtend.o` (presentes en donde indica `gcc –print-file-name=`). El *backslash* "\" continúa un comando que se desea prolongar a la siguiente línea. Si se desea, se puede teclear el comando `ld` en una sola línea sin pulsar <Enter>, haciendo innecesario el *backslash*.

Si se desea, se puede modificar el programa msum.c para sustituir la última sentencia `return` de `main` por un `printf("2 * 3 --> %ld\n", d);` (en cuyo caso también convendría empezar con `#include <stdio.h>` para declarar el prototipo de la función `printf`). 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`).

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

2.1 Código ensamblador y código máquina

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

```
// gcc -Og -S sum.c
// … -fno-asynchronous-unwind-tables

long plus(long, long);

void sumstore(long x, long y,
              long *dest) {
  long t = plus(x, y);
  *dest = t;
}
```

Figura 6: fichero fuente C sum.c

```
        .file  "sum.c"
        .text
        .globl sumstore
        .type  sumstore, @function
sumstore:
        pushq  %rbx
        movq   %rdx, %rbx
        call   plus@PLT
        movq   %rax, (%rbx)
        popq   %rbx
        ret
        .size  sumstore, .-sumstore
        .ident "GCC: (Ubuntu 7.3.0-16u3) 7.3.0"
        .section .note.GNU-stack,"",@progbits
```

Figura 7: fichero ensamblador sum.s

La versión ensamblador es una representación legible de las instrucciones máquina en que se convierte el programa, como las comentadas anteriormente `pushq %rbx` / `movq %rdx, %rbx`. Contiene también directivas, como `.text` (iniciar sección de código) y `.size` (tamaño de un objeto). En este caso, se define el tamaño del objeto `sumstore` (función global, ver `.global` y `.type`) como `".-sumstore"`, esto es, la diferencia entre el contador de posiciones `"."` y la propia etiqueta `sumstore`.

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 sum.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**	**significado**	**explicación**
-d		Disassemble	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		table	Muestra las entradas de la tabla de símbolos (similar a nm)
-T		Table	Muestra tabla de símbolos dinámicos (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   sum.o # mostrado al lado->
objdump -t   sum.o
nm           sum.o # sumstore en .text

objdump -S    sum.o      # falta -g
```

```
sum.o:     file format elf64-x86-64


Disassembly of section .text:
```


Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/

BY-NC-ND

```
gcc -g -Og -c sum.c
objdump -S    sum.o      # ahora sí

objdump -t    sum.o      # secciones -g
objdump -h    sum.o      # ver.text=14B
gcc    -Og -c sum.c      # quitar -g
objdump -S    sum.o      # ahora no
objdump -h    sum.o
```

Figura 8: sesión Linux

```
0000000000000000 <sumstore>:
  0: 53          push   %rbx
  1: 48 89 d3    mov    %rdx,%rbx
  4: e8 00 00
        00 00    callq  9 <sumstore+0x9>
  9: 48 89 03    mov    %rax,(%rbx)
  c: 5b          pop    %rbx
  d: c3          retq
```

Figura 9: desensamblado de sum.o

Como vemos en la Figura 9, el ensamblador emitió 14 bytes, el contador iba por 0 cuando se definió `sumstore`, irá por 0xe tras emitir `ret`, y por tanto "`.size sumstore, .-sumstore`" calculará el tamaño de `sumstore` como 14. La primera instrucción, "`push %rbx`", se codifica en lenguaje máquina como 0x53 y ocupa 1B, la posición 0. La segunda instrucción, "`mov %rdx,%rbx`", empieza en 0x1, ocupa 3B y acaba por tanto en 0x3, dejando el contador de posiciones en 0x4.

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 del *runtime* de gcc. En el Apéndice 2 hay un resumen de las instrucciones y modos de direccionamiento x86-64 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 ASM: llamadas al sistema

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 SYSCALL y MOV, para realizar las dos llamadas al sistema requeridas (*`WRITE`* escribir mensaje y *`EXIT`* 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 (`.data` y `.text`), 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,  ‘,     ""
#     ejs: 0x41, 0101, 0b01000001,  65,  ‘A,   "AAA"

.section .data      # directivas comienzan por .
            # no son instrucciones máquina, son indicaciones para as
            # etiquetas recuerdan valor contador posiciones (bytes)
saludo:
      .ascii "Hola a todos!\nHello, World!\n"       # \n salto de línea

longsaludo:
      .quad   .-saludo    # . = contador posic. Aritmética de etiquetas.

# SECCIÓN DE CÓDIGO (.text, instrucciones máquina)

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

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

```
_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 $1, %rax        # write: servicio 1 kernel Linux
       mov         $1,%rdi #    fd: descriptor de fichero para stdout
       mov    $saludo,%rsi #   buf: dirección del texto a escribir
       mov longsaludo,%rdx # count: número de bytes a escribir
       syscall             # llamar write(stdout, &saludo, longsaludo);

#      Llamada al sistema EXIT, consultar "man 2 exit"
#      void _exit(int status);
       mov $60, %rax       #   exit: servicio 60 kernel Linux
       mov  $0, %rdi       # status: código a retornar (0=OK)
       syscall             # llamar exit(0);
```

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

En arquitectura x86-64 cada posición de memoria es un byte. Ya vimos en la Figura 7 cómo se usó aritmética de etiquetas para calcular el tamaño ocupado por la función `sumstore`. En este caso, podríamos modificar (alargar o acortar) el *string* en el código fuente ensamblador, y la variable `longsaludo` tomaría automáticamente el valor correcto (longitud) para la posterior llamada a *`WRITE`*.

Dado que los Sistemas Operativos se ejecutan en un nivel de privilegio elevado (*espacio kernel* vs. *espacio usuario*) se debe utilizar algún mecanismo proporcionado por la arquitectura para elevar el privilegio de un proceso y/o permitirle ejecutar una llamada al sistema (*syscall*). Tradicionalmente se han usado las interrupciones software a tal efecto, especialmente en procesadores donde dicho mecanismo era el único disponible para conmutar entre niveles de privilegio, e incluso en procesadores donde no había espacio *kernel* protegido en oposición al espacio de usuario. El programador utiliza un vector concreto (`0x80` en el caso de Linux i386, ver `man 2 syscall`) cuando desea realizar la llamada. La subrutina de servicio espera encontrar el número de servicio y hasta 6 argumentos en registros del procesador (EAX y EBX, ECX, EDX, ESI, EDI, EBP en Linux i386) 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 otro registro (EAX en Linux i386). Los números de servicio (llamada) pueden encontrarse en `/usr/include/x86_64-linux-gnu/asm/unistd_32.h`.

La demanda de mayores prestaciones ha llevado a los fabricantes a proporcionar interfaces más rápidos para conmutar a espacio *kernel*, y así Linux x86-64 usa la instrucción `syscall` (ver `man 2 syscall`), pasando el número de servicio en RAX y los argumentos en RDI, RSI, RDX, R10, R8, R9 (una variante de la *SystemV AMD64 ABI*). El programador debe fijar dichos valores antes de ejecutar la instrucción `syscall`. Si la llamada al sistema produce un valor de retorno, lo devuelve en RAX. Los números de servicio (llamada) pueden encontrarse en `/usr/include/x86_64-linux-gnu/asm/unistd_64.h`.

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` para saber que tienen 3 (RDI, RSI, RDX) 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: gdb -tui

Ensamblar y enlazar el programa `saludo.s`, incluyendo información de depuración, y reproducir la siguiente sesión gdb. Aunque permitiremos usar el *frontend* ddd o el entorno Eclipse en modo gráfico si estuvieran disponibles, es conveniente aprender también los comandos gdb en modo texto, y de hecho alguna funcionalidad sólo puede accederse en dicho modo. Hay una lista de comandos en el enlace [6].

```
as  -g   saludo.s -o saludo.o    # ensamblar incluyendo info. depuración
ld       saludo.o -o saludo      # enlazar
gdb -tui saludo                  # sesión gdb en modo text-user-interface
```


Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

```
   list              # localizar línea "mov $1,%rdi" y ponerle breakpoint
   break 9           # equivalente gráfico ddd: cursor a izq.línea y stop
   info break        # equiv.gráf: Source->Brkpts. Notar address 0x4000b7

   run               # eq.gr: Program->Run /(View->CmdTool->)botonera->Run
   disassemble       # eq.gr: View->MachCodeWin
   print $rip              # Notar dirección break = RIP=0x4000b7
   print $rax              # Notar RAX=1, pero RDI=0 aún
   info registers          # eq.gr: Status->Registers
   stepi                   #
   p $rip                  # EIP sigue avanzando (p=print)
   p $rdi                  # Notar RDI=1 ahora
   si                      #                   (si=stepi)
   p/x $rsi                # Notar RSI=0x6000df > RIP
   disas _start            # Notar traducción ASM->LM $saludo, longsaludo

   x/32cb &saludo    # eq.gr: Data->Memory->Examine 32 char bytes &saludo
   x/32xb &saludo    # Print p/probar (cambiar a hex bytes)/Display p/fijo
   x/s    &saludo    # eq.gr: Data->Memory->Examine 1 string bytes &saludo
   p (char*) &saludo
   p(long)longsaludo
   x/1dg &longsaludo       # eXamine para ver dirección de inicio y valor
   x/1xg &longsaludo       # comprobar ordenamiento little-endian
   x/8xb &longsaludo
   si                # Comprobar regs EAX,RDI,RSI,RDX = 1,1,0x6000df,28
   info reg          #                (write,stdout,&saludo,longsaludo)
   disas             # $saludo=$0x6000df(inm), longs=0x6000fb(dir), %rdx=28(reg)
   si                # Se escribe mensaje en pantalla (View->GDB Console)
3x si / cont         # eq.gr: Pulsar cont o clickar 3 stepi para exit(0)

   clear 9                 # otra ejecución: parar justo antes del final
   break 16                # localizar la 2a syscall y ponerle bkpt
   info break
   run
   set $rdi=1              # y cambiar código de retorno sobre la marcha
   stepi / cont

   cl 16                   # otra ejecución: parar antes de imprimir
   br 12                   # localizar la primera syscall y ponerle bkpt
   info br
   run
   print      saludo             # 'saludo' has unknown type; cast it to
   p    (int) saludo             # interpreta 4B "Hola" (4 códigos ASCII)
   p /x (int) saludo             # como un entero 0x616c6f48, ver Apénd.1
   p /x(char) saludo             # typecast a char 1B 'H'
   p   (char) saludo             # hex 0x48, decimal 72

   p         &saludo             # dirección de memoria
   p  (char*)&saludo             # typecast a char* = string
   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
   quit
```

Figura 11: ensamblado, enlazado, y sesión de depuración usando gbd –tui

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). Observar qué fácilmente se modifica el *string* para que sea una línea, no 2.

Los *frontends* como ddd o entornos de desarrollo como Eclipse presentan al usuario un interfaz gráfico más intuitivo, como se ve en las siguientes figuras, aunque internamente usen gdb como depurador. Pueden usarse para la asignatura, pero el profesorado no es responsable de su funcionamiento.

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

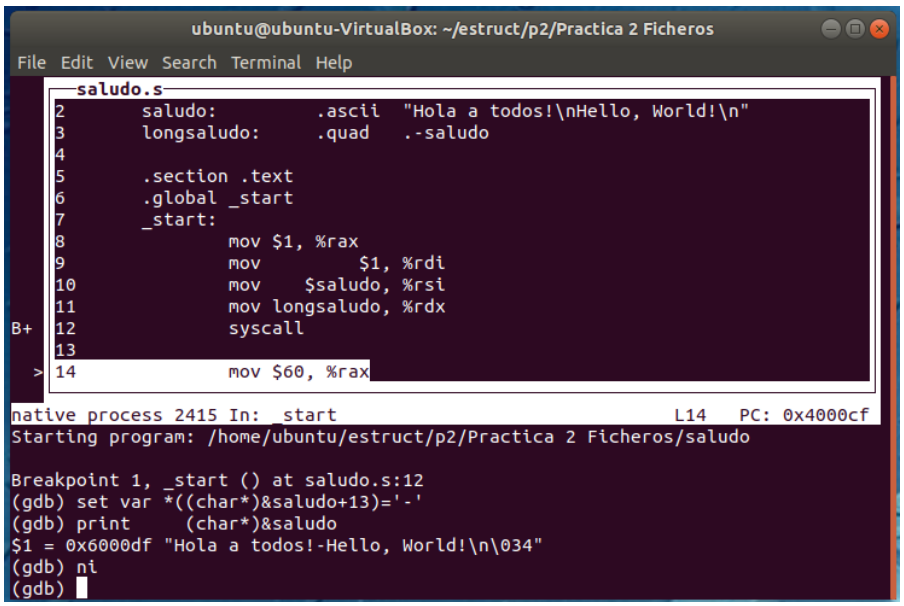


Figura 12: gbd –tui saludo. Observar qué fácilmente se modifica el string para que sea una línea, no 2.

En la Figura 12 se observa que ha sido necesario pulsar <Ctrl>-L para refrescar la pantalla, que quedó un poco trastocada tras ejecutar `syscall`. Al refrescar la pantalla se recupera el marco de decoración del código fuente, pero se pierde el texto escrito por la propia llamada al sistema.

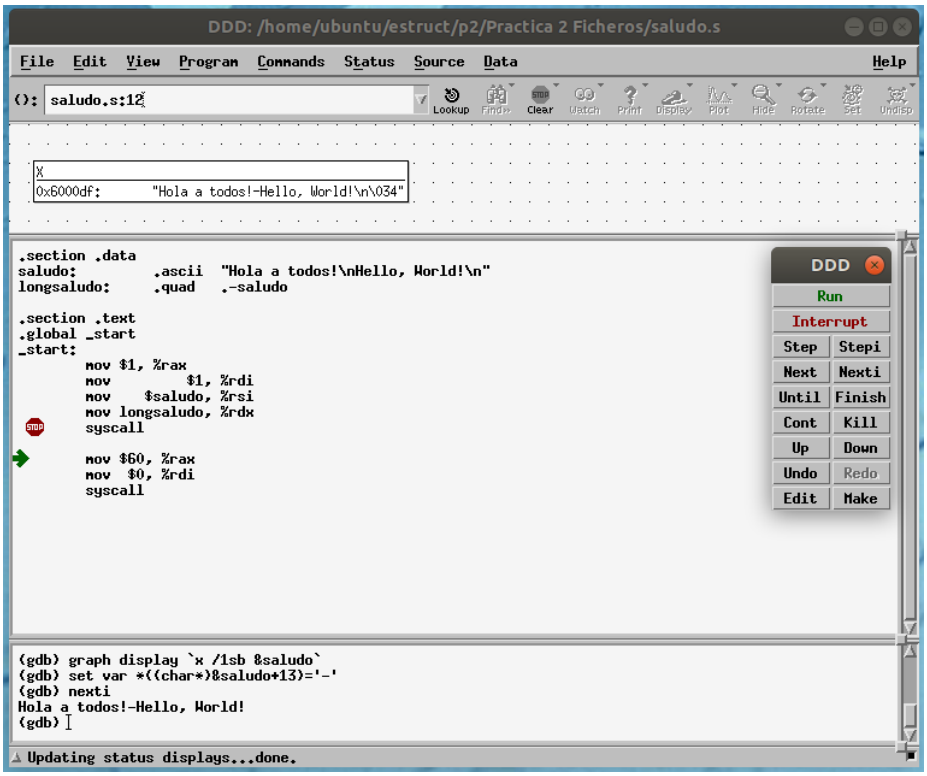


Figura 13: ddd saludo. Sigue siendo necesario usar el mismo comando en modo texto para modificar el string.

En la Figura 13 se ha utilizado *Data>Memory>Examine 1 string bytes from &saludo* para mostrar el *string*, aunque para modificarlo siga siendo necesario usar el mismo comando en modo texto. Para continuar la ejecución se ha usado el botón *Nexti*, que ha sido traducido al correspondiente comando en modo texto. La botonera agrupa los comandos más habituales. También son de uso frecuente las opciones de menú *View>**, *Program>** (*Run in Execution Window* es útil para obtener la salida en un `xterm` separado), *Status>Registers* y *Data>Memory*. Con *Edit>Preferences>Helpers* puede cambiarse el `xterm` a otro programa preferido.

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

Con el entorno Eclipse querremos utilizar *File>Import>C/C++ Executable* para depurar el ejecutable `saludo` que ya habíamos ensamblado con información de depuración usando `as -g`. Conviene configurar el lanzador en la pestaña *Debugger>Stop on startup at:* `main`, cambiándolo a `_start`. El *Wizard* nos permite cambiar automáticamente a la ***Perspectiva*** de *Depuración* (tal vez tengamos que cerrar la perspectiva *Java*), en donde disponemos de los botones habituales (*Debug*, *Run*, *Resume*, *Terminate*, *Step Into*, *Step Over*, etc). El modo *Instruction Stepping Mode* es útil cuando se depura código desensamblado para el cual no se dispone de fuente.

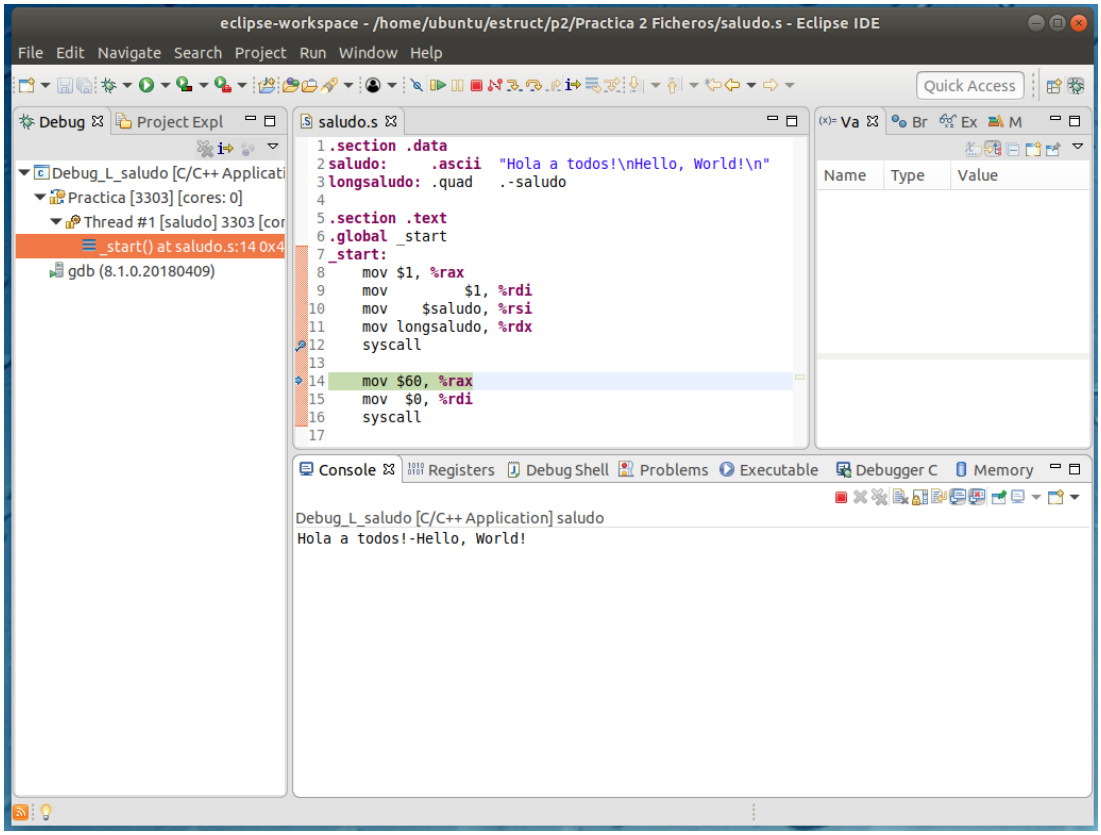


Figura 14: eclipse. Seguimos usando el mismo comando para cambiar el string.

En la Figura 14 se ha añadido un breakpoint en la primera `syscall`, se ha continuado con *Resume*, se ha añadido un *Memory>Memory Monitor* para `&saludo`, se ha vuelto a utilizar el mismo comando texto en la consola *Debugger C*, y al pulsar *Step Over* se ha obtenido el mismo resultado. Seguramente interesa consultar las ***Vistas*** *Console*, *Registers*, *Memory*, *Breakpoints*, etc.

Por experiencia con promociones anteriores (léase bomba digital), y viendo las versiones de los paquetes oficiales de ambos entornos en Ubuntu (quieta en ddd-3.3.12-5, atrasada por eclipse-3.8.1-11, sólo como paquete *snap* eclipse-4.8), el depurador que utilizaremos oficialmente en estas prácticas será `gdb -tui`, aunque no prohibimos que se usen ddd o Eclipse, pero en ningún caso será preocupación de esta asignatura resolver bloqueos del ddd al depurar la bomba digital (si es que los hay), o configurar Eclipse para enlazar o ensamblar o incluso compilar nuestros programas. Editaremos nuestros programas fuente con `vim` o `gedit`, los pasaremos a ejecutable con `as`, `ld` y/o `gcc`, y los depuraremos con `gdb -tui`, *permitiendo* el uso de ddd o Eclipse si el estudiante conoce y desea usar esos *frontends*.

4 Segundo programa ASM: Llamadas a funciones (libC, usuario)

Es conveniente dividir el código de un programa entre varias funciones, de manera que al ser estas 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 15 se muestra un programa ensamblador con una función (subrutina) que calcula la suma de una lista de enteros de 32 bits. La dirección de inicio de la lista y su tamaño se le pasa a la función a través de los registros RBX y ECX. El resultado se devuelve al programa principal a través del registro EAX. Se preservan los demás registros.

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

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 (si el compilador no puede optimizar el uso de registros para todas ellas), y para pasar argumentos (según la convención de llamada: por ejemplo SysV AMD64 usa pila a partir del 7º argumento). Las instrucciones PUSH y POP (Apéndice 2, Tabla 3) y las llamadas y retornos de subrutinas (CALL, RET, INT, IRET, Tabla 7) utilizan de forma implícita la pila, que es la zona de memoria adonde apunta el puntero de pila RSP. La pila crece hacia direcciones inferiores de memoria, y RSP apunta al último elemento insertado (tope de pila), de manera que PUSH primero decrementa RSP en el número de posiciones de memoria que ocupe el dato a insertar (8B o 2B, aunque nosotros sólo apilaremos palabras *quad* de 8B), y luego escribe ese dato en las posiciones reservadas, a partir de donde apunta RSP ahora. Similarmente POP primero lee del tope de pila, guardando el valor en donde indique su argumento, y luego incrementa RSP. 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 15 paso a paso con gdb `-tui` (o ddd o Eclipse) y comprobar que la pila funciona como se ha explicado; en particular, que CALL guarda la dirección de retorno, RET recupera el contador de programa, PUSH almacena temporalmente un valor, y POP lo recupera posteriormente.

```
# suma.s:     Sumar los elementos de una lista
#             llamando a función, pasando argumentos mediante registros
#             comprobar con "./suma; echo $?" o con depurador gdb/ddd

# SECCIÓN DE DATOS (.data, variables globales inicializadas)
.section .data
lista:        .int 1,2,10, 1,2,0b10, 1,2,0x10  # ejs. binario 0b / hex 0x
longlista:    .int (.-lista)/4    # . = contador posiciones. Aritm.etiq.
resultado:    .int  0

# formato:    .asciz "suma = %u = 0x%x hex\n"  # fmt para printf() libC
# el string "formato" sirve como argumento a la llamada printf opcional
# opción: 1) no usar printf, 2)3) usar printf/fmt/exit, 4) usar tb main
# 1) as  suma.s -o suma.o
#    ld  suma.o -o suma                                      1232 B
# 2) as  suma.s -o suma.o                                    6520 B
#    ld  suma.o -o suma -lc -dynamic-linker /lib64/ld-linux-x86-64.so.2
# 3) gcc suma.s -o suma –no-pie –nostartfiles                6544 B
# 4) gcc suma.s -o suma –no-pie                              8664 B

# SECCIÓN DE CÓDIGO (.text, instrucciones máquina)
.section .text                  # PROGRAMA PRINCIPAL
_start: .global _start          # se puede abreviar de esta forma
# main: .global   main          # Programa principal si se usa C runtime

      call  trabajar            # subrutina de usuario
#     call  imprim_C            # printf()  de libC
      call  acabar_L            # exit()   del kernel Linux
#     call  acabar_C            # exit()    de libC
      ret

trabajar:
      mov     $lista, %rbx      # dirección del array lista
      mov  longlista, %ecx      # número de elementos a sumar
      call suma                 # llamar suma(&lista, longlista);
      mov  %eax, resultado      # salvar resultado
      ret

# SUBRUTINA: int suma(int* lista, int longlista);
# entrada:   1) %rbx = dirección inicio array
#            2) %ecx = número de elementos a sumar
# salida:       %eax = resultado de la suma
suma:
      push     %rdx             # preservar %rdx (se usa como índice)
      mov  $0, %eax             # poner a 0 acumulador
      mov  $0, %rdx             # poner a 0 índice
bucle:
```


Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

```
bucle:
       add  (%rbx,%rdx,4), %eax   # acumular i-ésimo elemento
       inc   %edx                 # incrementar índice
       cmp   %edx,%ecx            # comparar con longitud
       jne    bucle               # si no iguales, seguir acumulando

       pop   %rdx                 # recuperar %rdx antiguo
       ret

#imprim_C:
#      si se usa esta subrutina, usar también la línea que define formato
#      se puede linkar con ld -lc -dyn ó gcc -nostartfiles, o usar main
#      mov  $formato,  %rdi       # traduce resultado a decimal/hex
#      mov  resultado, %esi       # versión libC de syscall __NR_write
#      mov  resultado, %edx       # ventaja: printf() con fmt "%u" / "%x"
#      mov         $0, %eax       # varargin sin xmm
#      call printf                # == printf(formato,resultado,resultado)
#      ret

acabar_L:                         #   void _exit(int status);
       mov       $60,  %rax       #   exit: servicio 60 kernel Linux
       mov  resultado, %edi       # status: código a retornar (la suma)
       syscall                    # == _exit(resultado);
       ret

#acabar_C:                        #   void  exit(int status);
#      mov  resultado, %edi       # status: código a retornar (la suma)
#      call exit                  # ==  exit(resultado)
#      ret
```

Figura 15: suma.s: ejemplo de llamada a subrutina (paso de parámetros por registros RBX/RCX)

Con esto concluye la sección de Seminario de esta práctica. Para la parte de trabajo personal se sugerirá mejorar este programa para que no pierda bits al ir sumando números (si son muchos, o grandes). También se pedirá dividir por el tamaño de la lista para calcular la media. Se pedirá realizarlo sobre registros de 32 bits (`.int` normales de 4B como los que hemos usado hasta ahora), usando aritmética multi-precisión, para recordar la diferencia entre pensar que los datos tienen signo o que no lo tienen. Por último, como la plataforma que usamos dispone de registros de 64 bits, podemos repetir el cálculo en uno de ellos y comprobar si nuestro programa en doble precisión produce el mismo resultado.

Desarrollo de las Prácticas en [Windows + VirtualBox +] Ubuntu 18.04.LTS

Como ya se ha comentado en clase de Teoría [1] (Presentación p.34 y Tema 1 p.64), en el laboratorio estamos usando Ubuntu 18.04.LTS 64bit. En un portátil con Windows se puede optar por instalar Ubuntu en una partición separada, o instalar algún software de virtualización como por ejemplo VirtualBox (si no se tenía previamente) y crear una máquina virtual con dicho Ubuntu.

A la instalación por defecto de Ubuntu se le podrían añadir (usando el comando `apt`, o tal vez instalándose el *frontend* gráfico *Synaptic*) los siguientes paquetes cuya presencia asumimos: `g++` (la *suite* del compilador), `ghex` (editor hexadecimal), `make` (para recompilar las *Guest Additions* de VirtualBox). Si se desea, también puede interesar instalar `gcc-multilib` (para recompilar aplicaciones de 32 bits que necesiten estas librerías de compatibilidad), `ddd` (*frontend* depurador gráfico), `eclipse-4.8` (paquete *snap* en *Ubuntu Software*, evitar el paquete ubuntu *Development (universe) eclipse-3.8*). Probablemente Eclipse requerirá instalar previamente `default-jre`, y posteriormente, entrando en el propio entorno Eclipse, *Help>Eclipse MarketPlace>Find CDT>Eclipse C/C++ IDE CDT 9.4 (Oxygen.2)*. Similarmente, para la opción *Run in Execution Window* de `ddd` convendrá instalar `xterm`. Para la práctica de cache también querremos instalar `gnuplot-x11`. El *firewall* se puede comprobar/activar con el comando "`sudo ufw status/enable`".

De esta forma se pueden repetir los tutoriales de prácticas, como este que acabamos de completar, de forma independiente. Al estar las sesiones de tutorial transcritas en su totalidad (incluso se han transcrito los equivalentes en modo texto de los comandos gráficos ejecutados en ddd), se pueden probar antes de venir al laboratorio, se pueden repetir después de haber asistido al tutorial, y se pueden repasar en cualquier momento, independientemente de las sesiones presenciales de laboratorio.

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

5 Trabajo a realizar

Se propone mejorar este último programa (Sección 4, Figura 15, **suma.s**) para calcular la media de una lista de N enteros realizando los cálculos sobre registros enteros. Entendemos por enteros los `.int` normales de la plataforma x86-64 (4B), y por cálculos las operaciones aritmético-lógicas (no las instrucciones de movimiento que obviamente utilizarán direcciones de 64 bits). Se debe considerar la posibilidad de desbordamiento aritmético (*overflow*), respondiendo siempre con el resultado correcto.

Por comodidad usaremos N=16 pero el código no puede asumir que el tamaño es fijo, y debe funcionar correctamente para cualquier tamaño N>0. Por explicar razonadamente la condición de usar registros de 32 bits, que en principio pudiera parecer arbitraria, piénsese en la dificultad de escribir valores de prueba para ejercitar la posibilidad de *overflow* si se usaran enteros `.quad` de 64 bits. También es un ejercicio interesante pensar en cómo se deben tratar adecuadamente los acarreos en aritmética multi-precisión con signo (que requiere extender el signo de los datos al tamaño del resultado), frente al tratamiento sin signo (que puede realizarse por extensión con ceros, o sencillamente acumulando acarreos). Adicionalmente, podemos aprovechar la plataforma de 64 bits para utilizar un registro `.quad` en donde realizar los cálculos directamente (sin aritmética multi-precisión) y comparar este resultado con lo calculado en 32 bits.

Según la temporización de cada curso, se procurarán realizar guiadamente (como Seminario Práctico) los Ejercicios 1-4 (e incluso **suma.s** si sobrara tiempo). Aunque no diera tiempo a tanto, comprender los programas mostrados (Figura 10, Figura 15) y ejercitarse en el uso de las herramientas (Figura 11, Figura 12, Figura 13, Figura 14) son competencias que cada uno debe conseguir personalmente.

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

5.1 Sumar N enteros *sin* signo de 32 bits sobre dos registros de 32 bits usando uno de ellos como acumulador de acarreos (N≈16)

En su forma actual **suma.s** ya permite un tamaño variable de la lista. La primera modificación que podemos hacer es cambiar la lista de nueve elementos a una lista que repita 16 veces el número 1. Podemos salvarlo con el nombre **media.s** y ejecutarlo para comprobar si `./media; echo $?` devuelve 16.

Notar que la suma puede necesitar más de 32 bits, incluso si la lista fuera de 2 números (0xffff ffff y 1, o 0x8000 0000 y 0x8000 0000). *¿Qué valor mínimo* habría que repetir 16 veces para producir acarreo? Es 0x1000 0000. Modificar la lista y comprobar que responde que el resultado es 0. Si no se entiende por qué, conviene ejecutarlo paso a paso con el depurador. Seguramente conviene dividir la definición de la lista en 4 líneas, cada una de ellas de 4 enteros. Sólo la primera necesita etiqueta `lista:`. Si hubiéramos rellenado la lista con 16 valores 0x0fffffff no hubiera llegado a producir acarreo (*¿por qué?*), y el resultado sería correcto. Comprobarlo. Es fácil calcular mentalmente el resultado en hexadecimal.

Quien lea el guión sin realizar las comprobaciones no habrá caído en la cuenta de que el programa retorna 0 y 240 para los dos ejemplos mencionados. La página `man 2 exit` explica *por qué* se retornan valores tan bajos. Se puede depurar el programa para comprobar que todo iba bien hasta la llamada a `EXIT`. Pero si imprimiéramos el resultado por pantalla usando `printf()` de *libC* evitaríamos este problema con `EXIT`, además de facilitar la comprobación de si el programa es correcto tras cada modificación que hagamos (si se imprime el resultado correcto, asumimos que no hay fallos). Para ello podemos descomentar el `formato` sugerido, cambiar la etiqueta `_start` por `main`, borrar el cuerpo de `main` y sustituirlo por los cuerpos de `trabajar`, `imprim_C` y `acabar_C` uno tras otro (se pueden borrar esas etiquetas, los respectivos `ret` y toda la subrutina `acabar_L`, y se puede retornar 0 en `acabar_C`), y dejar para el final la subrutina `suma` con su `bucle` y su `ret`. Tal y como se sugiere en el comentario *4)* del fuente, estas modificaciones nos permiten *"compilar"* el programa ensamblador usando gcc (recordar que con el Ubuntu 18.04 que usamos en prácticas, es necesario usar la opción `-no-pie` para que se genere un ejecutable, en lugar de un objeto compartido). Ahora es más fácil comprobar los dos ejemplos mencionados, 16 elementos a 0x1000 0000 o a 0x0fff ffff. Comprobar si los resultados se imprimen correctamente por pantalla con el formato deseado, sin necesidad de entrar en el depurador. El primer ejemplo pierde un bit de acarreo, y nuestro objetivo es conservar ese bit.

Prácticas de Estructura de Computadores, por Javier Fernández y Mancia Anguita, licencia CC BY-NC-ND.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/

La suma en doble precisión de números *sin signo* puede realizarse conservando los *acarreos* que de otra forma se hubieran perdido (instrucción etiquetada `bucle:` en `suma.s`, Figura 15, tras la cual no se comprueba si hay acarreo saliente de EAX). Si los vamos acumulando en otro registro de 32 bits (p. ej. EDX), la concatenación de ambos (EDX:EAX) puede almacenar el resultado (sin signo, de 64 bits). Para comprobar si hay acarreo y acumularlo sólo cuando lo haya, inmediatamente después de la suma etiquetada `bucle:` puede usarse un salto condicional JNC para saltarse una instrucción INC (*pensarlo*) y llegar a una etiqueta colocada tras ese INC. Necesitaremos usar otro registro como índice (tal vez RSI) en lugar de RDX/EDX, e inicializar los registros apropiadamente (acumuladores e índice a 0). Cuando veamos la instrucción DIV para calcular la media, se entenderá el interés de acumular en EDX:EAX.

Tomando como punto de partida el programa `suma.s`, con directivas `.int` adicionales para los N=16 valores (4 líneas de 4 valores 0x1000 0000), realizamos las modificaciones indicadas (JNC, INC, etiqueta) y cambiamos el tamaño de `resultado` a 64 bits (consultar en Apéndice 2, Tabla 9, la directiva as para ello). La subrutina `suma` debería devolver el resultado en los acumuladores EDX:EAX (deben eliminarse los `push/pop %rdx`), y el programa principal almacenarlo en la variable `resultado` de 64 bits, según el criterio del extremo menor (recordar que la familia x86-64 es *little-endian*). La dirección que empieza 4 posiciones detrás de `resultado` se indica `resultado+4`, naturalmente. Desde gdb debería poder visualizarse el valor correcto de la suma con `p/x (long)resultado`, o con `x/1xg &resultado`. Desde ddd se puede conseguir el mismo volcado de memoria con el menú *Data→Memory→Examine 1 hex giant from &resultado*.

Aunque consigamos calcular correctamente el resultado 0x1 0000 0000 en EDX:EAX, por pantalla seguimos imprimiendo enteros de 4 bytes. Es inmediato cambiar los especificadores de formato a "`%lu`" y "`%lx`", así como los argumentos de `printf` a `%rsi` y `%rdx`, para conseguir imprimir el resultado de 8 bytes en decimal sin signo y en hexadecimal. Tras conseguir imprimir el resultado correcto para el ejemplo más pequeño que produce acarreo, probar con el ejemplo más grande posible. El resultado debe ser 0xf ffff fff0 (*¿por qué?*).

5.2 Sumar N enteros *sin* signo de 32 bits sobre dos registros de 32 bits mediante extensión con ceros (N≈16)

Un número E sin signo de N bits puede ser extendido a N+M bits sin más que añadirle M bits a cero como parte más significativa (a la izquierda). En nuestro caso, deseamos sumar al acumulador EDX:EAX de 32+32 bits un elemento de la lista, E, de 32 bits. Como consideramos que E no tiene signo, podemos extenderlo a 64 bits *imaginándonos* que tuviera otros 32 bits a 0 a la izquierda. Para sumar estos dos números de 64 bits en doble precisión (usando sumas de 32 bits), empezaríamos sumando las partes menos significativas (E y EAX), puesto que puede haber acarreo. Luego sumaríamos las partes más significativas (*0* y EDX… y el acarreo si lo hubiera).

Afortunadamente existe la instrucción de suma con acarreo ADC (consultar Apéndice 2, Tabla 4) que permite sumar dos valores y el flag CF. La modificación necesaria es por tanto eliminar el salto JNC, el INC y la etiqueta posterior, y sustituirlos por una suma con acarreo sobre EDX (*pensarlo*) inmediatamente posterior a la suma de las partes menos significativas. Volver a probar el programa con los cuatro ejemplos usados hasta ahora: 16 veces 1, 16 veces 0x0fff ffff, 16 veces 0x1000 0000, 16 veces 0xffff ffff. A lo mejor conviene imprimir EDX y EAX separadamente para facilitar la lectura.

Un problema que tenemos es que cada vez que cambiamos de ejemplo tenemos que volver a teclear los 16 valores. Una solución hubiera sido dejar comentada la lista completa de cada ejemplo, y descomentar en cada momento sólo el ejemplo deseado. O tal vez, como se trata de cuatro líneas iguales, dejar comentada una línea de cada ejemplo, para poder copiarla rápidamente cuatro veces cuando sea necesario.

Otra recomendación que facilita enormemente incorporar una batería de test al código fuente es la mostrada en la Figura 16, basada en las directivas `.irpc` y `.macro` de as (ver Apéndice 2, Tabla 9). Como la mayoría de los ejemplos consisten en repetir 4 veces una línea que emite 4 enteros, se pueden definir separadamente las líneas deseadas, pudiendo cambiar de ejemplo sin más que descomentar la definición deseada.

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

```
.section .data
        .macro linea
         #     .int 1,1,1,1
         #     .int 0x0fffffff, 0x0fffffff, 0x0fffffff, 0x0fffffff
         ...    ...
         #     .int 5000000000,5000000000,5000000000,5000000000
        .endm
lista: .irpc i,1234
               linea
        .endr
```

Figura 16: uso de las directivas .irpc y .macro para preparar una batería de pruebas

Cambiar de ejemplo se reduce entonces a comentar una línea y descomentar otra. Para aún mayor comodidad, se podría acomodar esta batería para compilación condicional con el preprocesador cpp, aprovechando que de todas formas ya estamos *"compilando"* el programa usando gcc para conseguir el enlazado con *libC* automáticamente. La batería y el formato *printf* podrían quedar entonces de la siguiente manera:

```
.section .data
#ifndef TEST
#define TEST 9
#endif
        .macro linea                        #  Resultado - Comentario
  #if  TEST==1                              // 16 – ejemplo muy sencillo
               .int 1,1,1,1
#elif  TEST==2                              // 0x0 ffff fff0, casi acarreo
               .int 0x0fffffff, 0x0fffffff, 0x0fffffff, 0x0fffffff
...
#elif  TEST==8                              / 11 280 523 264 << 16x5e9= 80e9
               .int 5000000000,5000000000,5000000000,5000000000
#else
        .error "Definir TEST entre 1..8"
#endif
        .endm
...
formato:      .ascii "resultado \t =   %18lu (uns)\n"
              .ascii          "\t\t = 0x%18lx (hex)\n"
              .asciz          "\t\t = 0x %08x %08x \n"
```

Figura 17: batería de pruebas acomodada para compilación condicional con cpp, y formato recomendado

En la Figura 17 se muestran los resultados incorporados al propio código fuente de forma que quede éste autodocumentado, en un formato muy similar a la propia definición de la batería en la página 16. Con esta recomendación se puede ejecutar toda la batería con una única línea de comando en el *shell* (un bucle *for* del *bash*), como el mostrado en la Figura 18.

```
ubuntu@ubuntu-VirtualBox:~/estruct/p2$ for i in $(seq 1 9); do
> rm media
> gcc -x assembler-with-cpp -D TEST=$i –no-pie media.s -o media
> printf "__TEST%02d__%35s\n" $i "" | tr " " "-" ; ./media
> done
rm: cannot remove 'media': No such file or directory
__TEST01__-----------------------------------
resultado        =                    16 (uns)
                 = 0x                 10 (hex)
                 = 0x 00000000 00000010
__TEST02__-----------------------------------
...
ubuntu@ubuntu-VirtualBox:~/estruct/p2$
```

Figura 18: uso de un bucle *for* para comprobar la batería de tests aprovechando compilación condicional

Comprobar el correcto funcionamiento del programa por lo menos para la siguiente batería de 8 tests. Notar que cuando el ejemplo se ofrece en decimal se espera que el resultado también se exprese en decimal, y similarmente para hexadecimal (ejemplo hex ⇒ resultado hex). Notar que para facilitar la lectura de las cantidades, se espera que los dígitos decimales se agrupen de 3 en 3, y los hexadecimales de 4 en 4. Salvo el 5º ejemplo en decimal (que es idéntico al 4º, escrito en hex) y el último (que incluye un error de representación numérica), todos los ejemplos se pueden calcular mentalmente sin

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

necesidad de calculadora, motivo por el cual unos están en decimal y otros en hexadecimal. Usar el accesorio "Calculadora" de Ubuntu en modo programador si surgen dudas sobre algún resultado.

Test #	Ejemplo	Resultado	Comentario
TEST01	**[1, …]**	16	Todos los elementos a 1
	Ejemplos fáciles de calcular en hexadec.		
TEST02	**[0x0FFF FFFF, …]**	0x0 ???? ????	Máximo elem e para que 16e no produzca acarreos
TEST03	**[0x1000 0000, …]**	0x? ???? ????	Mínimo elem e para que 16e sí produzca 1 acarreo
TEST04	**[0xFFFF FFFF, …]**	0x? ???? ????	Notar agrupación de 4 en 4 dígitos hexadec.
TEST05	**[-1, …]**	~~?? ??? ??? ???~~	Notar agrupación de 3 en 3 dígitos decimales
	Ejemplos fáciles de calcular en decimal		
TEST06	**[200 000 000, …]**	? ??? ??? ???	Máx e $A{\cdot}10^B$ p/suma representable uns32b <=4Gi-1
TEST07	**[300 000 000, …]**	? ??? ??? ???	Mín el $A{\cdot}10^B$ p/suma produzca 1CF uns32b (>=4Gi)
TEST08	**[5 000 000 000, …]**	~~?? ??? ??? ???~~	Mín elm tipo $A{\cdot}10^B$ no representable uns32b>=4Gi
	+ TEST05	+ ~~?? ??? ??? ???~~	si al resultado incorrecto del TEST08
	+ skin 16	+ 16	se le suma el TEST05 y const. Skinner 16
	=	80 000 000 000	se obtiene el resultado correcto. ¿Por qué?

Se han tachado los resultados del TEST05 y TEST08 porque son claramente erróneos, debido a la incorrecta interpretación y/o representación de los elementos de la lista. Coincide que sumando esos resultados tachados junto con el valor 16 se obtiene el resultado correcto del TEST08. *¿Por qué?* ¿Funciona el 16 como constante de Skinner, o se puede explicar por qué hace falta?

5.3 Sumar N enteros *con* signo de 32 bits sobre dos registros de 32 bits (mediante extensión de signo, naturalmente) (N≈16)

Se desea ahora interpretar los números de la lista como enteros con signo. Correspondientemente, la variable de 64 bits en donde se almacene el resultado también se interpretará con signo. Eso implica realizar cambios al programa. En concreto, el flag de acarreo (CF) ya no significa que haya error en el resultado, e insistir en acumular los acarreos produciría un resultado sólo comprensible si los datos originales vuelven a interpretarse sin signo (repasar el TEST04 y TEST05 de la anterior batería de tests).

Notar que para una lista tan sencilla como por ejemplo 16 veces -1, el resultado del programa con signo debería ser -16 (0xffffffff fffffff0), mientras que con nuestro programa actual (sin signo) se obtiene casi 64Giga (=0xf ffff fff0). La diferencia radica en que nuestro programa sin signo extiende con ceros (*aunque sea imaginariamente* en la solución de la sección 5.1) los datos originales, mientras que para poder acumular números de 32 bits con un acumulador de 64 bits se debería realizar extensión de signo 32→64 bits al dato original antes de sumarlo al acumulador. Si aún no se ve claro, otro ejemplo más convincente por su sencillez sería `lista[]={0,-1}`, para el cual nuestro programa actual calcula 0x00000000 ffffffff= 2^{32}-1 = 4Giga-1 (tras haber extendido -1 con ceros, interpretándolo por tanto sin signo), cuando lo que deseamos sería 0xffffffff ffffffff=-1 (*extendiendo -1 con unos y 0 con ceros*, interpretando por tanto ambos elementos con signo).

Tomando como partida el programa ensamblador del apartado anterior (en donde 0-1≠-1 porque -1 se interpreta sin signo), nos interesaría añadir alguna instrucción de extensión de signo (Apéndice 2, Tabla 3 y Tabla 5) tras ***leer*** (no *sumar*) cada elemento de la lista, para pasarlo a doble precisión ***antes*** de acumular. Las instrucciones que nos interesan no son las de tipo MOVSX ni CBW/cbtw, porque no queremos usar los registros de 64 bits, sino las de tipo CWD/cwtq, porque queremos pasar de uno a dos registros de 32 bits. Probablemente querremos *leer* ***(no** sumar)* el elemento en EAX, extender a dos registros de 32 bits, sumar esos 64 bits (ADD para los LSB, ADC para los MSB) con otros 2 registros usados como acumulador (se puede usar EBP si fuera necesario), y antes de retornar, mover la suma acumulada de 64 bits a EDX:EAX, que es donde la espera el programa principal. Como ya dijimos, cuando veamos la instrucción de división quedará claro por qué se desea la suma en EDX:EAX.

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

Recordar que estamos recomendando *"compilar"* usando gcc, tanto para enlazar cómodamente *printf* como para aprovechar la compilación condicional de cpp para preparar la batería de tests. Para que *printf* imprima ahora resultados de 64 bits *con* signo hay que modificar el especificador de formato de "`%lu (uns)`" a "`%ld (sgn)`". Recordar también que el formato "`%08x %08x`", apropiado para facilitar la lectura de grandes números hex, requiere introducir las mitades más y menos significativas del resultado en los registros de 32 bits correspondientes a la posición del argumento en la lista de argumentos.

Comprobar el correcto funcionamiento del programa por lo menos para la siguiente batería de 19 tests. Recordar que el resultado se debe expresar en la misma base en la que se ofrece el ejemplo (decimal o hexadecimal), que todos los ejemplos son fáciles de calcular mentalmente (por eso unos están en decimal y otros en hexadecimal), y que para facilitar la lectura de las cantidades, los dígitos decimales deben agruparse de 3 en 3 y los hexadecimales de 4 en 4. Siempre se puede usar el accesorio "Calculadora" de Ubuntu en modo programador si surgen dudas sobre algún resultado.

Test #	Ejemplo	Resultado	Comentario
TEST01	**[-1, …]**	-16	Todos los elementos a -1
	Ejemplos fáciles de calcular en hexadecimal		Rellenar toda la lista con un valor…
TEST02	**[0x0400 0000, …]**	0x0 ???? ????	…positivo pequeño (suma cabría en sgn32b)
TEST03	**[0x0800 0000, …]**	0x0 ???? ????	…positivo intermedio (sm. cabría en uns32b)
TEST04	**[0x1000 0000, …]**	0x? ???? ????	…positivo intermedio (sm. no cabría uns32b)
TEST05	**[0x7FFF FFFF, …]**	0x? ???? ????	…positivo grande (máximo elem. en sgn32b)
			Similarmente con valores negativos…
TEST06	**[0x8000 0000, …]**	0xffff ???? ???? ????	…negativo grande (mínimo elem. en sgn32b)
TEST07	**[0xF000 0000, …]**	0xffff ???? ???? ????	…negativo intermedio (no cabría en sgn32b)
TEST08	**[0xF800 0000, …]**	0xffff ???? ???? ????	…negativo pequeño (suma cabría en sgn32b)
TEST09	**[0xF7FF FFFF, …]**	0xffff ???? ???? ????	…anterior-1 es interm. (no cabría en sgn32b)
	Ejemplos fáciles de calcular en decimal		Rellenar toda la lista con un valor…
TEST10	**[100 000 000, …]**	? ??? ??? ???	…fácil calcular q. suma cabe sgn32b (<=2Gi-1)
TEST11	**[200 000 000, …]**	? ??? ??? ???	…pos+gran $A{\cdot}10^B$ suma cabe uns32b (<=4Gi-1)
TEST12	**[300 000 000, …]**	? ??? ??? ???	…pos+peq $A{\cdot}10^B$ suma no cabe uns32b(>=4Gi)
TEST13	**[2 000 000 000, …]**	?? ??? ??? ???	…pos+gran $A{\cdot}10^B$ reprsntble sgn32b (<=2Gi-1)
TEST14	**[3 000 000 000, …]**	~~-?? ??? ??? ???~~	…pos+peq $A{\cdot}10^B$ no reprsntble sgn32b(>=2Gi)
			Similarmente con valores negativos…
TEST15	**[-100 000 000, …]**	-? ??? ??? ???	…fácil calcular q. suma cabe sgn32b (>=-2Gi)
TEST16	**[-200 000 000, …]**	-? ??? ??? ???	…neg+peq $-A{\cdot}10^B$ suma no cabe sgn32b(<-2Gi)
TEST17	**[-300 000 000, …]**	-? ??? ??? ???	…aún menos hubiera cabido
TEST18	**[-2 000 000 000, …]**	-?? ??? ??? ???	…neg+gran $A{\cdot}10^B$ reprsntble sgn32b (>=-2Gi)
TEST19	**[-3 000 000 000, …]**	~~+?? ??? ??? ???~~	…neg+peq $A{\cdot}10^B$ no reprsntble sgn32b(<-2Gi)
	TEST06	- ~~?? ??? ??? ???~~	si al resultado incorrecto del TEST19
	TEST06	- ~~?? ??? ??? ???~~	se le suma 2 x TEST6, se obtiene
	=	-48 000 000 000	el resultado correcto. ¿Por qué?

Se han tachado los resultados del TEST14 y TEST19 porque son claramente erróneos, debido a la incorrecta interpretación y/o representación de los elementos de la lista. Parece lógico que ambos resultados tengan la misma magnitud con distinto signo, ya que los elementos también tienen el signo

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

cambiado. Coincide que sumándole (2 x TEST06) al TEST19, se obtiene el resultado correcto del TEST19. *¿Por qué?* ¿Funciona (2 x TEST09) como constante de Skinner, o se puede explicar por qué hace falta?

En la Figura 19 se muestran los resultados incorporados al propio código fuente de forma que quede éste autodocumentado, en un formato muy similar a la propia definición de la batería de tests.

```
.section .data
#ifndef TEST
#define TEST 20
#endif
       .macro linea                          #  Resultado: Comentario
  #if  TEST==1                               // -16: ejemplo muy sencillo
              .int -1,-1,-1,-1
#elif  TEST==2                               // 0x0 4000 0000: incluso sgn32b
              .int  0x04000000, 0x04000000, 0x04000000, 0x04000000
...
#elif  TEST==19                              // 20 719 476 736>>16x-3e9= -48e9
              .int -3000000000,-3000000000,-3000000000,-3000000000
#else
       .error "Definir TEST entre 1..19"
#endif
       .endm
...
...
formato:      .ascii "resultado \t =   %18ld (sgn)\n"
              .ascii         "\t\t = 0x%18lx (hex)\n"
              .asciz         "\t\t = 0x %08x %08x \n"
```

Figura 19: batería de pruebas acomodada para compilación condicional con cpp, y formato recomendado

5.4 Media y resto de N enteros *con* signo de 32 bits calculada usando registros de 32 bits (N≈16)

Modificar el programa anterior para que calcule la media y resto de la lista de números. La instrucción de división a añadir (Apéndice 2, Tabla 5) viene obligada, teniendo en cuenta que se trata de números con signo (*¿cuál debe ser?*). También viene obligado en qué registros debe ponerse el dividendo, y en dónde quedarán el cociente y el resto (*¿dónde?*), que ya no son de 64 bits. Añadir la instrucción de división en la subrutina, y salvar el cociente y el resto tras volver de la subrutina, en variables del tamaño (y tipo) apropiado.

Aunque el libro [1] utilizado en las clases no explica la extensión de signo con la instrucción CLTD, sí que explica las instrucciones CLTQ (pág. 221) y CQTO (pág.234), mostrando un ejemplo de su uso antes de IDIV (pág. 236. En la página 221 menciona la posibilidad de extender con MOVSLQ, y también es posible realizar esa operación con MOV y SAR $31 (*pensarlo*), forma usada frecuentemente por gcc. La explicación empieza en la página 233, Sección 3.5.5 *"Special Arithmetic Operations"*, y concluye con el Problema 3.12 en la página 236 (resuelto en la página 365), sugiriendo las modificaciones que habría que realizar al código anterior (pág. 235) para que la división fuera sin signo. El ejemplo anterior (pág.234, desensamblado en la pág.235) explica la multiplicación sin signo en doble precisión.

Los ejemplos que hemos sugerido para las baterías anteriores consistían en repetir siempre el mismo número, buscando casos que producirían *overflow* o acarreo en caso de realizarse sobre un acumulador de 32 bits. Con esos ejemplos no aprenderemos mucho sobre la división, ya que la media debe ser siempre dicho valor repetido. Sugerimos esta vez casos más sencillos para descubrir si la instrucción de división entera IA32 funciona como la división euclidiana "normal" usada en aritmética modular (cociente redondeado hacia ±∞ según se obtenga resto positivo) o truncada (cociente redondeado hacia 0, resto del mismo signo que el dividendo). Como informáticos, deberíamos conocer con precisión las reglas de evaluación aritmética de los lenguajes en los que aprendemos a programar, incluyendo la precedencia de los distintos operadores, el tratamiento del acarreo/*overflow*, el signo del resto en la división entera (¿mismo signo que numerador?), etc.

Recordar que recomendamos *"compilar"* usando gcc para aprovechar *printf* de *libC* y la compilación condicional de cpp para la batería de tests. Para que *printf* imprima ahora resultados de 32 bits con signo se debe modificar de nuevo el formato para ajustarse al tipo de datos del cociente y del resto. Se

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

pueden imprimir ambos en una misma llamada *printf*, de nuevo en decimal y hexadecimal, en dos líneas de pantalla si se desea (usando los formatos `"\n"` y `"\t"` para saltar de línea y alinear con tabulaciones), controlando incluso los dígitos de anchura (y ceros a la izquierda) de cada número (`"%11d"`, `"0x%08x"`).

Comprobar el correcto funcionamiento del programa por lo menos para la siguiente batería de 42 tests. Recordar que el resultado se debe expresar en la misma base en que se ofrece el ejemplo (decimal o hexadecimal), que todos los ejemplos son fáciles de calcular mentalmente (por eso unos están en decimal y otros en hexadecimal), y que para facilitar la lectura de las cantidades, los dígitos decimales deben agruparse de 3 en 3 y los hexadecimales de 4 en 4. Siempre se puede usar el accesorio "Calculadora" de Ubuntu en modo programador si surgen dudas sobre algún resultado.

Test #	Ejemplo	Media	Resto	Comentario
TEST01	**[1, 2, 1, 2, …]**	1	8	Cíclicamente 1,2… euclídea o truncada?
TEST02	**[-1, -2, -1, -2, …]**	-?	-?	Cíclicamente -1,-2… aclarado
	Ejemplos fáciles de calcular en hexadecimal			
TEST03	**[0x7FFF FFFF, …]**	0x???? ????	0x?	…positivo grande (máx. elem. sgn32b)
TEST04	**[0x8000 0000, …]**	0x???? ????	0x?	…negativo grande (mín. elem. sgn32b)
TEST05	**[0xFFFF FFFF, …]**	0xFFFF FFFF	0x0	…negativo más pequeño (equiv. TEST15)
	Ejemplos fáciles de calcular en decimal			
TEST06	**[2 000 000 000, …]**	? ??? ??? ???	??? ???	…fácil ver q. suma cabe sgn32b (<=2Gi-1)
TEST07	**[3 000 000 000, …]**	~~-? ??? ??? ???~~	??? ???	…pos+gran A·10^B suma uns32b (<=4Gi-1)
TEST08	**[-2 000 000 000, …]**	-? ??? ??? ???	??? ???	…pos+peq A·10^B suma no uns32b(>=4Gi)
TEST09	**[-3 000 000 000, …]**	~~+? ??? ??? ???~~	??? ???	…pos+gran A·10^B repr. sgn32b (<=2Gi-1)
	Ejemplos fáciles decimal: división eucl. vs. trunc.			
TEST10	**[0, 2, 1, 1, 1, 1, …]**	?	?	Toda la lista a 1 salvo 2º elemento a 2, y
TEST11	**[1, 2, 1, 1, 1, 1, …]**	?	?	barrido del 1er elemento a 0,1,8,15,16
TEST12	**[8, 2, 1, 1, 1, 1, …]**	?	?	(barrido para ver si resto eucl./trunc.)
TEST13	**[15, 2, 1, 1, 1, 1, …]**	?	??	
TEST14	**[16, 2, 1, 1, 1, 1, …]**	?	?	Ventajoso pasar 1er "1" a 2º elemento
TEST15	**[0,-2,-1,-1, -1,-1, …]**	-1	0	Igual pero en negativo (equiv. a TEST05)
TEST16	**[-1,-2,-1,-1, -1,-1, …]**	?	?	
TEST17	**[-8,-2,-1,-1, -1,-1, …]**	?	?	
TEST18	**[-15,-2,-1,-1, -1,-1, …]**	?	??	
TEST19	**[-16,-2,-1,-1, -1,-1, …]**	?	?	Ventajoso pasar 1er "-1" a 2º elemento

Como *podemos comprobar*, la instrucción de división en Intel 64 se corresponde con lo que la Wikipedia denomina división truncada (buscar *Módulo*), lo cual tiene una implicación respecto al signo del módulo (*¿cuál?*). *¿Coincide IDIV* con los operadores / (división entera) y % (módulo) en los lenguajes C/C++?

En la Figura 20, además del formato *printf* recomendado y la técnica de autodocumentación del código fuente (incorporando los resultados), se muestra también una posible técnica con dos macros para implementar los últimos tests a partir del 10, en donde toda la lista vale un mismo valor (±1) salvo los dos primeros elementos. ¿Por qué se afirma en los comentarios de los tests 14 y 19 que es "ventajoso acumular el 1er elemento al 2º"? ¿Qué significa esa frase?

```
.section .data

#ifndef TEST
#define TEST 20
#endif
       # doble macro: lista definida por una linea0 y 3 lineas normales
       # mayoría ejemplos linea0 = linea => lista tiene 4 lineas normales
```

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

```
        .macro linea                        # Media / Resto - Comentario
  #if   TEST==1                             //    1 / 8
              .int  1, 2, 1, 2
#elif   TEST==2                             //   -1 /-8
              .int -1,-2,-1,-2
...
...
#elif   TEST>=10 && TEST<=14
              .int  1, 1, 1, 1              # linea0 + 3lineas, casi todo a 1
#elif   TEST>=15 && TEST<=19
              .int -1,-1,-1,-1              # linea0 + 3lineas, casi todo  -1
#else
        .error "Definir TEST entre 1..19"
#endif
        .endm

        # mayoría ejemplos linea0 = linea => lista tiene 4 lineas normales

        .macro linea0                       # Media / Resto - Comentario
  #if   TEST>=1 && TEST<=9
              linea                         # casi siempre 4 lineas iguales
#elif   TEST==10                            //   1 / 0   - equiv toda lista 1
              .int   0, 2, 1, 1
...
#elif   TEST==15                            //  -1 / 0   - equiv TEST05
              .int   0,-2,-1,-1
...
#elif   TEST==19                            //    ? / ?
              .int -16,-2,-1,-1
#else
        .error "Definir TEST entre 1..19"
#endif
        .endm

        # mayoría ejemplos linea0 = linea => lista tiene 4 lineas normales

lista:        linea0
        .irpc i,123
              linea
        .endr

longlista:    .int   (.-lista)/4
media:        .int   0
resto:        .int   0

formato:      .ascii "media \t = %11d \t resto \t = %11d   \n"
              .asciz       "\t = 0x %08x \t    \t = 0x %08x\n"
```

Figura 20: batería de pruebas acomodada para compilación condicional con cpp, y formato recomendado

5.5 Media y resto de N enteros calculada en 32 y en 64 bits (N≈16)

Añadir al programa anterior otra subrutina `sumaq:` que repita la acumulación, pero esta vez sobre un registro de 64 bits, no sobre dos de 32 bits. La instrucción de *extensión* de signo es ahora *distinta* (Apéndice 2, Tabla 5). Por hacer mínimos cambios, si se extiende EAX→RAX tal vez convendría acumular sobre RDI (dejando sin uso RBP), y moverlo al dividendo RAX (asegurándose de *extenderlo* a RDX:RAX) antes de dividir por el divisor de 64 bits. Aunque el cociente y el resto (*¿dónde estarán?*) sean de 64 bits, es correcto aprovechar solo las partes menos significativas de 32 bits (*¿por qué?*). Se puede seguir salvando el cociente y el resto, tras volver de la subrutina, en las mismas variables, e imprimirlas con el mismo formato (o tal vez con otra copia `formatq:`, en la que avise de que los cálculos se realizaron en 64 bits). Lo que sí necesitaremos cambiar será la etiqueta `bucle:` (a `bucleq:`, por ejemplo).

El programa llamaría a `suma:` y a *printf* (con el mismo `formato:` del Apartado 5.4), y posteriormente llamaría a `sumaq:` y a *printf* con el nuevo `formatq:`, confiando en que ambos mensajes produzcan el mismo resultado impreso. Se puede reutilizar la misma batería de tests, naturalmente.

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

6 Entrega del trabajo desarrollado

Los distintos profesores de prácticas indicarán las normas de entrega para cada grupo, incluyendo qué se ha de entregar, cómo, dónde y cuándo.

Por ejemplo, puede que en un grupo el profesor habilite que los estudiantes que cumplan el régimen de asistencia puedan entregar en SWAD, hasta medianoche del domingo de la última semana de esta práctica, el fichero ensamblador correspondiente al último Apartado que hayan conseguido implementar (código fuente, listo para "compilar" con `gcc -D TEST=$i` y para reproducir la batería con `for i in $(seq 1 …);` usando la técnica mostrada en la Figura 18) y un fichero .txt con los resultados de la batería de test, volcado íntegramente de la pantalla (incluso redireccionando la salida estándar) en el orden y con el formato indicado en este guión de prácticas. Puede que en otro grupo el profesor de prácticas proporcione otras instrucciones distintas.

En cualquier caso, **el examen de prácticas** a final del curso (Test TP de 4 puntos) **es el mismo para todos los grupos**, lo cual significa que independientemente del profesor de prácticas, cada estudiante es responsable de comprender todo lo explicado en estos **guiones de prácticas (que son comunes a todos los grupos y son materia de examen)** y desarrollar las habilidades que se ejercitan en los mismos (que también son materia de examen).

7 Bibliografía

[1] Apuntes y presentaciones de clase, y particularmente
Programación Máquina II: Control
sección "Saltos condicionales", p.12-21, particularmente
generación de código para if-then-else, p.14-15
Libro CS:APP: Randal E. Bryant, David R. O'Hallaron: "Computer Systems: A Programmer's Perspective", 3rd Ed., Pearson, 2015. http://csapp.cs.cmu.edu/

[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"
https://software.intel.com/en-us/articles/intel-sdm
https://software.intel.com/en-us/articles/intel-sdm#three-volume
https://software.intel.com/sites/default/files/managed/a4/60/325383-sdm-vol-2abcd.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

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

Apéndice 1. Tabla de caracteres ASCII

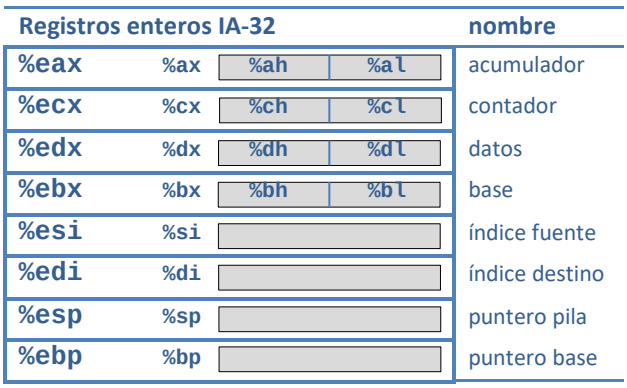
En algún momento (Figura 11) se han interpretado 4 bytes de un *string* como un entero. Para comprobar que el resultado obtenido no es en absoluto impredecible, sino que es matemáticamente preciso, se proporciona la tabla de códigos ASCII de 7 bits en la que están presentes todos los caracteres utilizados 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 x86-64 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 16 bits (AX…BP, para modo 16 bits), y doble palabra de 32 bits (EAX…EBP). Hay un registro de flags (EFLAGS) y un contador de programa (EIP).



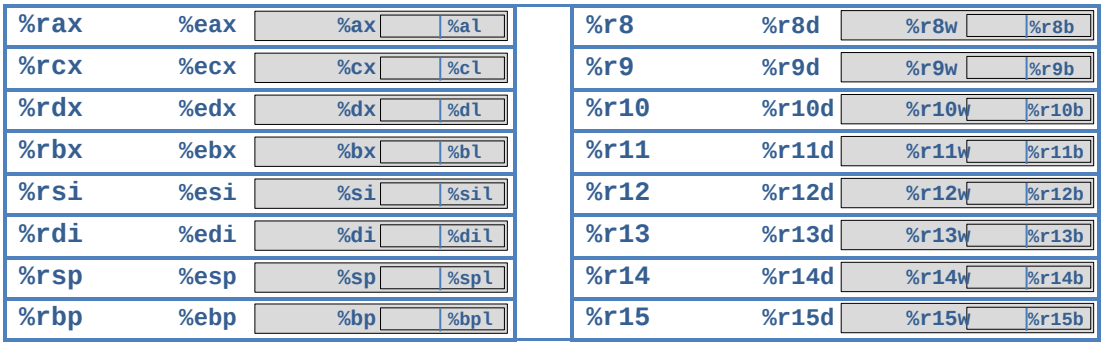
También existen registros para datos en punto flotante de 64/80 bits (FP0…FP7), registros MMX de 64 bits (MM0…MM7), registros SSE de 128 bits (XMM0…XMM7), etc, que se usan con instrucciones específicas del repertorio.

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.

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

Los 16 registros x86-64 de propósito general son de tamaño cuádruple palabra (RAX…RBP + R8…R15), y también puede accederse en tamaños menores: 32 bits (EAX…EBP + R8D…R15D), 16 bits (AX…BP + R8W…R15W), 8 bits (AL…BL + SIL…BPL + R8B…R15B). RFLAGS y RIP también son de 64 bits.

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	RSP ← RSP-|S| M[RSP] ← S	Meter fuente en pila	pushq $8
POP	D	D ← M[RSP] RSP ←RSP+|S|	Sacar de pila a destino	popq %rbp
LEA	D, S	D ← &S	Cargar dirección efectiva	leaq tabla(,%rax,4), %rsi

Tabla 3: 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 4: 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).

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

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 Dr,S Dr,S,I	$Ac_{D:A} \leftarrow Ac_A * S$ $D \leftarrow D * S$ $D \leftarrow D*S*Inm$	Multiplicación con signo	1 operando: Como MUL, nbit x nbit = 2nbits 2 operandos: Reg * (R/M/Inmediato), n x n = nbits 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 CWDE CDQE		$Ac_{2n} \leftarrow ExtSign(Ac_n)$	Extensión de signo acum. word-to-long-to-quad	AX ← ExtSigno(AL) cbtw EAX ← ExtSigno(AX) cwtl RAX ← ExtSigno(EAX) cltq
CWD CDQ CQO		$Ac_{D:A} \leftarrow ExtSign(Ac_A)$	Extensión de signo acum. x-to-double	DX:AX ← ExtSigno(AX) cwtd EDX:EAX ← ExtSigno(EAX) cltd RDX:RAX ← ExtSigno(RAX) cqto

Tabla 5: 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 32 bits 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 JMP	*Ptr PtrDst	(AT&T) (Intel)	Salto incondic. Indirecto "near, absolute, indirect"				

Tabla 6: 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. Los códigos de condición "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 rip rip ← label	Llamada a subrutina Intel: "near relative"	INT	v	PUSH rflags CALL ISR#v	Llamada a ISR Interrupción "Interrupción software"
call CALL	*Ptr PtrDst	(AT&T) (Intel)	Intel: "near absol. indirect"				
RET		POP rip	Retorno de subrutina	IRET		RET POP rflags	Retorno de ISR
LEAVE		RSP ← RBP POP RBP	Libera marco pila Para usar con ENTER	NOP		No-op	
CLC STC CMC		C ← 0 C ← 1 C ← ~C	Ajustes del flag acarreo	CLI STI		I ← 0 I ← 1	Ajustes del flag Interrupt (des)habilitar IRQs

Tabla 7: Instrucciones x86 - control y miscelánea

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

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

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	`rax`	`%rax`
Orden de operandos fuente y destino	`mov rbx, 0f02h`	`movl $0xf02, %rbx`
Dirección de variable	`mov rcx, offset var`	`movl $var, %rcx`

Sufijos/Directivas de tamaño de operandos
AT&T: b, w, l, q
Intel: (byte/word/dword/qword) ptr
	`mov ah,byte ptr [rbx]`
	`mov cx, dx`
	`mov esi, edi`
	`mov rbp, rsp`
		`movb (%rbx), %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, tabla[rax*4]`
	`mov eax,[rbp-4]`
		`Despl(Rb,Ri,s)`
		`movl tabla(,%rax,4),%edx`
		`movl -4(%rbp), %eax`

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

Mnemotécnicos transferencias control lejanas inm.	`jmp far sect:offset`
	`call far sect:offset`
	`ret far 8`
		`ljmp $sect,$offset`
		`lcall $sect,$offset`
		`lret $8`

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`
		`movslq ecx, rdx`

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

Tabla 8: 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" [, ...]* **.asciz** *"string" [, ...]*	Ensambla cada string (pasando a código ASCII cada letra) en direcciones consecutivas Añade al string un byte 0 final al estilo de los strings en lenguaje C ("0-terminated")
.byte *expr [, ...]*	Ensambla cada expresión en el siguiente byte (dirección, posición de memoria)
.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.
.macro *name [,args]* **.endm**	Define una *macro(instrucción)* que genera código ensamblador. En sucesivas llamadas, se sustituye el nombre por el código definido. Los argumentos se referencian como \arg.
.irp[c] *symbol, values* **.endr**	Asigna a *symbol* los valores *values* (numérico o carácter). La directiva *instruction repeat [char]* repite el código definido para los valores indicados, sustituyendo \symbol por cada 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 9: directivas GNU AS

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

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

A petición de un grupo del curso 2019-2020, se añaden versiones de las Tablas 3, 4, 6 (nuevas Tablas 10, 11, 12) evitando usar la sintaxis Intel, aunque recordamos que si se quiere sacar provecho de los manuales del fabricante se debe conocer su sintaxis.

Instrucciones de movimiento de datos (evitando sintaxis Intel)

Instrucción		Efecto	Descripción	Ejemplos
MOV	S, D	D ← S	Mover fuente a destino (mismo tamaño). tamaño byte, word, long…	movb %al, %bl movl $0xf02, %ecx
MOVS	S, D	D ← ExtSigno(S)	Mover con extensión de signo (aumentando tamaño) byte-to-word, byte-long, word-long…	movsbw %al, %bx movslq %ecx, %rdx
MOVZ	S, D	D ← ExtCero(S)	Mover rellenando con ceros (aumentando tamaño) movzbw, movzbl, movzwl…	movzbq %al, %rbx movzwl %cx, %edx
PUSH	S	RSP ← RSP-|S| M[RSP] ← S	Meter fuente en pila	pushq $8
POP	D	D ← M[RSP] RSP ←RSP+|S|	Sacar de pila a destino	popq %rbp
LEA	S, D	D ← &S	Cargar dirección efectiva	leaq tabla(,%rax,4), %rsi

Tabla 10: Instrucciones x86 - transferencia

Instrucciones aritmético-lógicas (evitando sintaxis Intel)

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	S, D	D ← D + S	Sumar	XOR	S, D	D ← D ^ S	O-exclusivo
SUB	S, D	D ← D - S	Restar	OR	S, D	D ← D | S	O lógico
ADC	S, D	D ← D+S+C	Sumar con acarreo	AND	S, D	D ← D & S	Y Lógico
SBB	S, D	D ←D-(S+C)	Restar con débito	IMUL	S, D	D ← D * S	Multiplicar
SAL	k, D	D ← D << k	Desplazamiento a izq.	RCL	k, D	D ← D $\circlearrowright_C$ k	Rotación izq. incl. acarreo
SHL	k, D	D ← D << k	Desplazamiento a izq.	ROL	k, D	D ← D $\circlearrowright$ k	Rotación a izquierda
SAR	k, D	D ← D>>$_A$ k	Desplaz. aritm. derecha	RCR	k, D	D ← D $\circlearrowleft_C$ k	Rotación der. incl. acarreo
SHR	k, D	D ← D >>$_L$ k	Desplaz. lógico derecha	ROR	k, D	D ← D $\circlearrowleft$ k	Rotación a derecha

Tabla 11: Instrucciones x86 - aritmético-lógicas

Comparaciones y saltos/ajustes/movimientos condicionales (evitando sintaxis Intel)

Instrucción		Cálculo	Efecto	Instrucción		Efecto	Descripción
CMP	S_2, S_1	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_2, S_1	S_1 & S_2	Ajustar flags según cálculo	CMOVcc	S, Dreg	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		Salto incondic. Indirecto “near, absolute, indirect”				

Tabla 12: Instrucciones x86 - comparaciones y condicionales

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND