Programación en Lenguaje Ensamblador bajo Linux

Amelia Ferreira
Vicente Robles

Caracas, 2007

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Introducción *6*

Capítulo 1. La plataforma IA-32 .... *8*

Descripción general de la máquina **8**

Tipos de datos 8

Registros 10

Registros de propósito general 10

Registros de segmento 12

Registro apuntador de instrucción 17

Registros de punto flotante 17

Banderas 17

Capítulo 2. El entorno de programación *19*

El compilador gcc **19**

Binutils 24

El depurador gdb 30

Ejecución de programas con el depurador 30

Comandos básicos de control 34

Comandos básicos de inspección 38

Alteración de la ejecución del programa 41

Uso de la opción –g de gcc 46

Conclusiones y detalles finales 48

Capítulo 3. El programa en lenguaje ensamblador *50*

Definiciones generales **50**

Secciones **50**

Punto de inicio **51**

Finalización del programa **51**

Estructura general **52**

Capítulo 4. Definición de datos *53*

Definición de datos inicializados **53**

Definición de constantes **54**

Definición de datos sin inicializar **55**

Capítulo 5. Instrucciones *56*

Instrucciones de movimiento de datos **57**

La instrucción mov 57

Movimiento de datos inmediatos a registro o a memoria 58

Movimiento de datos entre registros 58

Movimiento de datos entre memoria y registros 58

Movimiento de datos con extensión 59

Carga dirección efectiva (instrucción leal) 59

Uso de la pila 60

Instrucción push (apilar) 60

Instrucción pop (desapilar) 61

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Instrucciones aritméticas y lógicas 62
add (suma) 62
sub (resta) 63
inc (incremento) 63
dec (decremento) 63
neg (negación aritmética) 64
not (negación lógica) 64
and (y lógico) 64
or (o lógico) 65
xor (or exclusivo) 66
shl (desplazamiento lógico a la izquierda) 66
shr (desplazamiento lógico a la derecha) 67
sar (desplazamiento aritmético a la derecha) 67
mul (multiplicación de enteros sin signo) 68
imul (multiplicación de enteros con signo) 68
imul con un operando 68
imul con dos operandos 68
imul con tres operandos 69
cwd (convierte palabra en palabra doble) 69
cltd (convierte palabra doble a palabra cuádruple) 69
div (división de enteros sin signo) 69
idiv (división de enteros con signo) 70

Instrucciones de comparación 71
cmp (compara) 71
test (examina bits) 71

Instrucciones de salto 72

Actualización de un registro con valor de una bandera 74

La instrucción loop 75

Ejercicios 77

Capítulo 6. Entrada / Salida 79

Funciones de alto nivel 79
Función printf 80
Función scanf 81
Consideraciones adicionales 83

Llamadas al sistema 87
Servicio read 88
Servicio write 88

Instrucciones de bajo nivel 93
El subsistema de E/S 93
Interfaz de E/S: 94
Funciones de una Interfaz de E/S: 94
Técnicas de transferencia de E/S: 95
Entrada/Salida programada: 95
Entrada/Salida basada en interrupciones: 96
Acceso directo a memoria (DMA): 96
Direccionamiento de los dispositivos de E/S 97
E/S aislada y E/S mapeada a memoria 98
Instrucciones “in” y “out” 98
Dispositivos de E/S: El teclado 101
Ejemplos prácticos: 105

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Ejercicios **116**

Capítulo 7. Control de flujo ***118***

Traducción de estructuras condicionales **118**

Ciclos **120**

Do while 120
While 121
For 122

Ejercicios **124**

Capítulo 8. Arreglos ***125***

Declaración de arreglos 125
Lectura de datos en un arreglo 126
Escritura de datos en un arreglo 128

Ejercicios **130**

Capítulo 9. Procedimientos ***131***

Instrucciones para llamadas a procedimientos 131
Convenciones para preservar los contenidos de los registros 132
Estructura del programa con llamada a procedimientos 132
Pase de parámetros 134
Ejemplo de programa con pase de parámetro por valor 134
Estado de la pila 135
Ejemplos de programas con pase de parámetro por referencia 136
Ejemplo de programa con una llamada a una función dentro de un ciclo 140

Ejercicios **143**

Capítulo 10. Cadenas de caracteres ***144***

Instrucción movs 144
El prefijo rep 146
La instrucción lods 147
La instrucción stos 147
La instrucción cmps 148
La instrucción scas 150

Ejercicios **153**

Capítulo 11. Operaciones de punto flotante ***154***

Instrucciones de movimiento de datos 157
fld (apila un dato punto flotante en la pila FPU) 157
fild (apila un entero) 158
fst (movimiento de datos desde el tope de la pila) 158
fst (copia el tope de la pila) 158
fstp (lee el tope de la pila y desapila) 158
fist (lee número entero del tope de la pila) 159
fxch (intercambio de valores entre el tope de la pila y otro registro) 159
Instrucciones de control 159
finit (inicialización) 159
fstcw (copia el contenido del registro de control) 159
fstsw (copia el contenido del registro de estado) 159
Instrucciones aritméticas básicas 160
fadd (suma) 160
fsub (resta) 160
fmul (multiplicación) 161

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

fdiv (división) 161
Instrucciones de comparación 163

Ejercicios 165

Capítulo 12. Manejo de archivos 166

Tipos de archivo 167
Archivos de texto 167
Archivos binarios 168

Tipos de acceso 168
Acceso secuencial 168
Acceso aleatorio 168

Funciones de alto nivel 169
fopen....................170
fclose 171
fread 171
fwrite 171
fgets 172
fputs 172
fscanf 172
fprintf 173
fseek 173
feof 173
ferror....................174
fflush 174
remove....................174

Llamadas al sistema 186
creat 187
open 187
close 188
read 189
write 189
lseek 189
unlink 190
mkdir 190
rmdir 190
rename 191
chmod 191
chdir 191

Ejercicios 196

Soluciones a los ejercicios 197

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Introducción

Este libro está destinado a aquellas personas que deseen aprender a programar en lenguaje ensamblador para la plataforma IA-32 bajo ambiente Linux.
Los ejemplos se refieren al compilador gcc (GNU compiler collection). El ensamblador de gcc se llama *as* y por formar parte de gcc comúnmente se conoce como *gas*. Este libro no pretende cubrir todas y cada una de las instrucciones del lenguaje ensamblador sino las más comunes y presenta una serie de ejemplos y ejercicios para ayudar al lector en la programación. Con el conjunto de herramientas presentadas es posible desarrollar programas complejos que solucionen una gran variedad de problemas.

La programación en lenguaje ensamblador ofrece diferentes ventajas. Un programa escrito en lenguaje ensamblador requiere considerablemente menos memoria y tiempo de ejecución que un programa escrito en un lenguaje de alto nivel. La programación en lenguaje ensamblador depende de la arquitectura del computador sobre el cual se trabaja, ésto es importante para entender el funcionamiento interno de la máquina, por ello al programar en ensamblador se llega a comprender cómo funciona el computador y cómo es su estructura básica. La capacidad de poder escribir programas en lenguaje ensamblador es muy importante para los profesionales del área de Sistemas Operativos debido a que los programas residentes y rutinas de servicio de interrupción casi siempre son desarrollados en lenguaje ensamblador.
Además, aún cuando la mayoría de los especialistas en programación desarrolla aplicaciones en lenguajes de alto nivel, que son más fáciles de escribir y de mantener, una práctica común es codificar en lenguaje ensamblador aquellas rutinas que han causado cuellos de botella en el procesamiento.

La organización del libro corresponde al proceso de aprendizaje de la programación en lenguaje ensamblador, primero se describen las características básicas de la arquitectura de la máquina sobre la cual se va a trabajar, luego el entorno de programación es decir, los programas y herramientas necesarios

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

para realizar la programación. Este capítulo sirve de introducción al uso de estas herramientas y luego de consulta a medida que van aprendiendo nuevos conceptos ya que provee la información necesaria para la detección y corrección de errores del programa escrito en lenguaje ensamblador. Los siguientes capítulos describen la estructura del programa, cómo declarar datos y cómo utilizar las instrucciones básicas. Luego se explica cómo realizar operaciones de lectura y escritura de información para posteriormente pasar a la explicación de la programación de estrcuturas condicionales, ciclos y el uso de arreglos. Más adelante se describen las convenciones para el desarrollo de procedimientos. Hasta este punto se cubren los conceptos básicos de la programación en lenguaje ensamblador, los capítulos siguientes proveen una visión un poco más detallada de ciertos aspectos como son el manejo de cadenas de caracteres lo cual es posible realizar de manera más eficiente con un conjunto de instrucciones dedicadas. Estas instrucciones se presentan en el capítulo 10. Luego se describe cómo programar utilizando operaciones de punto flotante y finalmente el último capítulo está dedicado al manejo de archivos.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Capítulo 1. La plataforma IA-32

Escribir programas en lenguaje ensamblador requiere el conocimiento de la estructura básica de la máquina sobre la cual se va a trabajar. Este capítulo describe, por tanto, las características más relevantes de la arquitectura IA-32 (Intel Architecture 32 bits), su repertorio de instrucciones y las consideraciones de diseño que afectan directamente la programación en lenguaje ensamblador, esta descripción se hace usando la nomenclatura de *gas*.

Descripción general de la máquina

El computador se compone de varios bloques funcionales: el procesador, la memoria, los dispositivos de entrada/salida y los buses. El procesador se encarga del control y ejecución de los programas; los programas y los datos que los mismos utilizan se guardan en memoria; la comunicación con el exterior se logra a través de los dispositivos de entrada/salida y los buses son los encargados de transferir la información entre todos los componentes del sistema antes mencionados.

Tipos de datos

Los datos se pueden accesar de diversas maneras. Se puede leer un sólo byte (8 bits) o un conjunto de bytes. En esta máquina en particular se denomina palabra a dos bytes y doble palabra a cuatro bytes. Los datos pueden representarse en notación binaria, octal, decimal o hexadecimal:

Binaria	Octal	Decimal	Hexadecimal
0b1000	08	8	0x8
0b1100	014	12	0xC
0b1101011011	01533	859	0x35B

De esta forma, todo dato precedido por el prefijo '0b' será interpretado por la máquina como un número binario. De forma similar, los prefijos '0' y '0x' denotarán datos octales o hexadecimales respectivamente. Por último, todo dato que no contenga uno de estos prefijos será interpretado por defecto como decimal.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Esta sintaxis es válida tanto para el ensamblador *gas* como para el compilador *gcc*, a excepción del prefijo '0b' el cual no está soportado por este último al momento de escribir estas líneas (se tiene previsto incluir soporte para el mismo en el futuro).

Los tamaños de los diferentes tipos de datos del lenguaje de programación C son los siguientes:

Declaración en C	Tipo de datos en IA-32	Sufijo de gas	Tamaño (en bytes)
char	byte	b	1
short	palabra	w	2
int	doble palabra	l	4
unsigned	doble palabra	l	4
long int	doble palabra	l	4
unsigned long	doble palabra	l	4
char *	doble palabra	l	4
float	precisión simple	s	4
double	precisión doble	l	8
long double	precisión extendida	t	12

En gas las instrucciones utilizan un sufijo para indicar el tamaño de los datos sobre los cuales operan. El mismo puede ser omitido en aquellas instrucciones en las cuales el tamaño de al menos uno de los operandos pueda ser deducido por gas de manera implícita. Sin embargo, debe ser incluido obligatoriamente en casos en que los operandos no tengan un tamaño implícito (por ejemplo una operación entre inmediato y memoria).
Los sufijos disponibles en gas son: 'b' (byte), para denotar operandos de un byte, 'w' (word) para denotar operandos de dos bytes o una palabra, y 'l' (long) para denotar operandos de cuatro bytes o una palabra doble.

Por último, es importante tener presente que el sistema guarda los datos en memoria en secuencia inversa de bytes (little endian) lo cual trae como consecuencia que el byte menos significativo se ubique en la posición de menor orden y el byte más significativo en la posición de memoria de mayor orden.
Por ejemplo si se transfiere el dato 0x457A a las posiciones consecutivas de memoria 0x100 y 0x101 se ubica el byte 0x7A en la posición 0x100 y el byte 0x45 en la posición 0x101.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Registros

Tipo de registro	Número y tamaño	Descripción
De propósito general	8 registros de 32 bits	Almacenan datos. Para uso del programador
De segmento	6 registros de 16 bits	Permiten direccionar la memoria
Apuntador de instrucción	1 registro de 32 bits	Apunta a la siguiente instrucción a ejecutar
De punto flotante	8 registros de 80 bits	Se usan para aritmética punto flotante

Registros de propósito general

Los registros de propósito general se utilizan para almacenar datos temporalmente. Debido a que estos registros han evolucionado desde una máquina de 8 bits (el 8080) un grupo de registros aún se puede utilizar con un tamaño de 8 bits para mantener compatibilidad con toda la línea de procesadores.

Aún cuando estos registros pueden mantener cualquier tipo de datos, algunos tienen cierta funcionalidad específica o son usados de manera especial por algunas instrucciones.

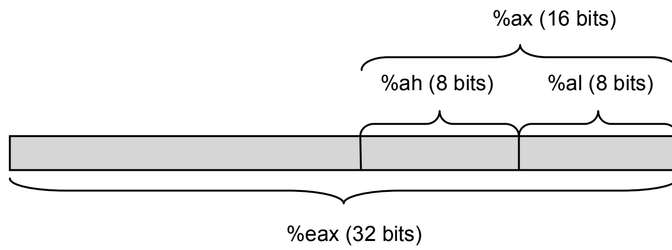
La siguiente tabla muestra los nombres de los registros y sus usos más comunes:

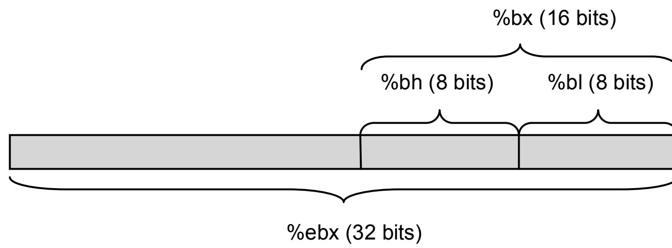
Registro	Descripción
eax	Acumulador para operaciones aritmetico lógicas
ebx	Registro base para acceder a memoria
ecx	Contador para algunas instrucciones
edx	Registro de datos usado para algunas operaciones de entrada/salida
edi	Apuntador a destino para operaciones con cadenas de caracteres
esi	Apuntador a origen para operaciones con cadenas de caracteres
esp	Apuntador de pila
ebp	Apuntador de marco de pila

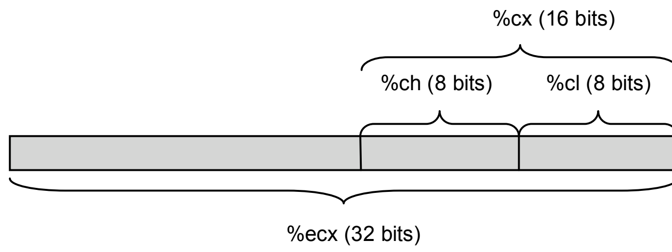
En *gas* los registros se denotan anteponiendo el símbolo de % al nombre del registro. Los registros %eax, %ebx, %ecx y %edx pueden ser direccionados como registros con tamaños de 8, 16 o 32 bits cambiando su nomenclatura de acuerdo al tamaño. De forma similar, los registros %edi, %esi, %ebp y %esp se pueden direccionar como registros de 16 o 32 bits.

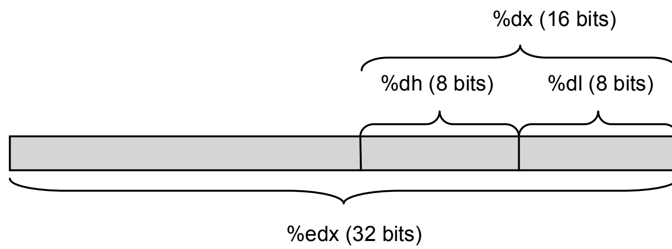
Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Los tamaños de los registros %eax, %ebx, %ecx y %edx son los siguientes:



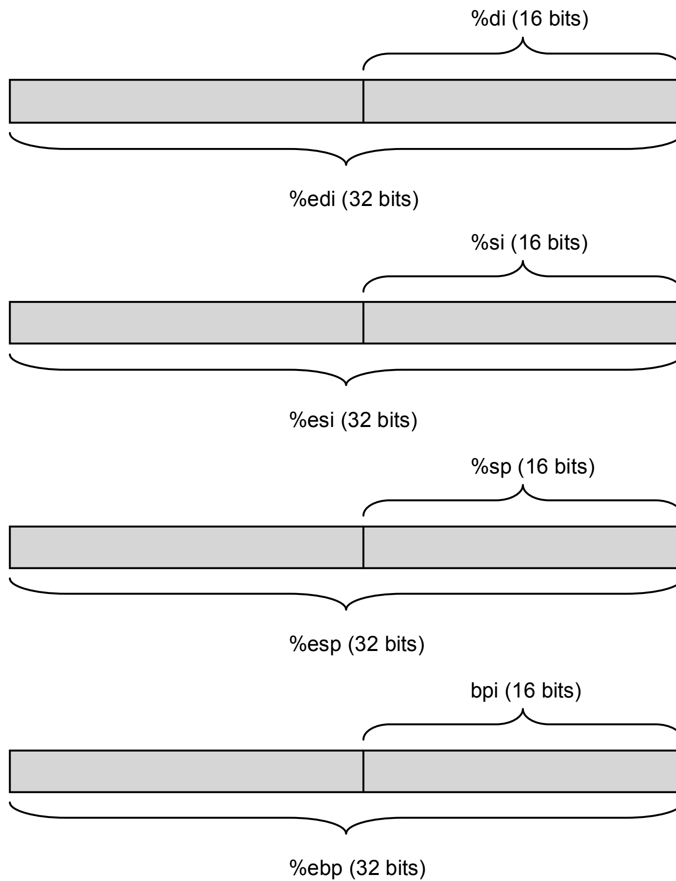






Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Los tamaños de los registros %edi, %esi, %esp y %ebp son los siguientes:



Registros de segmento

Los registros de segmento se utilizan para referenciar áreas de memoria. La plataforma IA-32 permite direccionar la memoria según el modelo de memoria lineal o el modelo de memoria segmentada.

El modelo de memoria lineal presenta todo el espacio de direcciones de la memoria como un espacio contiguo. Todas las instrucciones, los datos y la pila se encuentran en el mismo espacio de direcciones de memoria. Cada posición de memoria se referencia mediante una dirección específica llamada "dirección lineal".

El problema del enfoque anterior consiste en que todos los datos se encuentran mezclados entre si y a la vez distribuidos a lo largo de todo el espacio de direcciones memoria, lo cual hace que su manejo sea engorroso e ineficiente. El

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

modelo de memoria segmentada resuelve este problema dividendo el espacio de direcciones en segmentos independientes entre si. Cada segmento contiene un tipo específico de información, es decir el código (las instrucciones) se ubica en un segmento, los datos en otro y la pila en un tercer segmento. Las posiciones de memoria en los segmentos se definen por direcciones lógicas. Una dirección lógica está compuesta por una dirección de segmento y un desplazamiento. El procesador traduce una dirección lógica a una dirección lineal.

Los procesadores de la familia Intel poseen un grupo de registros creados con el fin de soportar el modelo de memoria segmentada, los cuales son conocidos como *registros de segmento*:

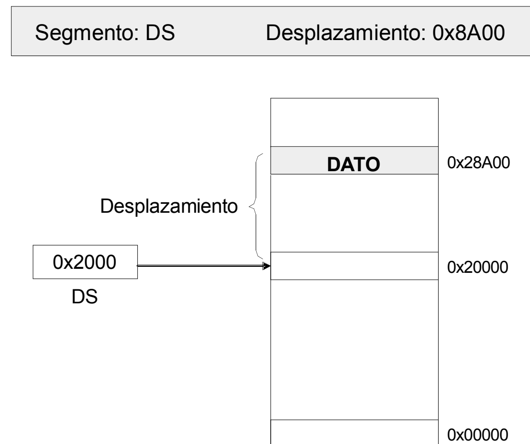
Registro de segmento	Descripción
cs	Segmento de código
ds	Segmento de datos
ss	Segmento de pila
es	Apuntador de segmento extra
fs	Apuntador de segmento extra
gs	Apuntador de segmento extra

El contenido de los mismos es interpretado de diferente forma dependiendo si el procesador se encuentra trabajando en *modo real* o en *modo protegido*.

Los procesadores más antiguos de esta familia trabajaban en modo real, en el cual un registro de segmento simplemente contenía una dirección lineal que señalaba el comienzo de un segmento. De esta forma, si un programa intentaba acceder a un dato ubicado en el desplazamiento ‘D’ del segmento apuntado por el registro ds, la traducción a una dirección lineal consistía en tomar el contenido de dicho registro, multiplicarlo por 0x10 (esto es la dirección de inicio del segmento de datos en uso) y sumarle el desplazamiento ‘D’.

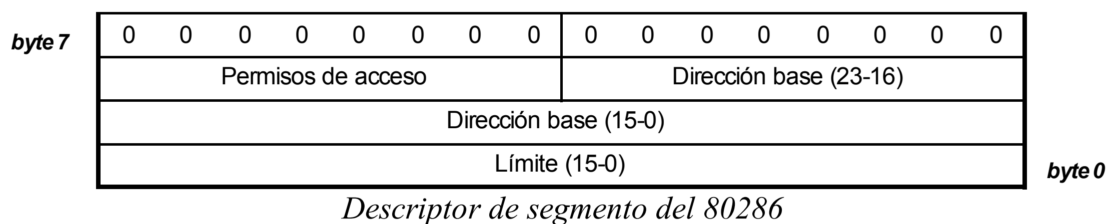
Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Por ejemplo:



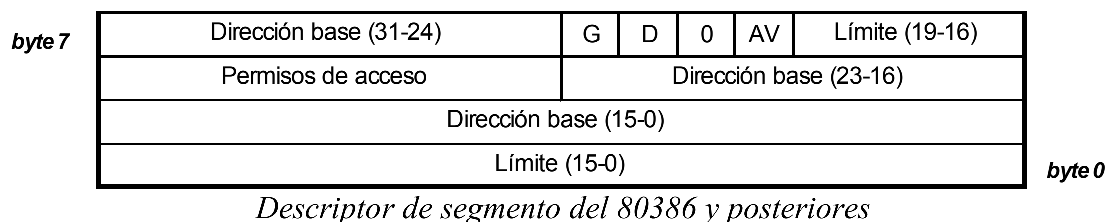
Sin embargo, con el paso de los años este enfoque resultó ser insuficiente, ya que por ejemplo, no había forma de determinar si un desplazamiento causaba que la dirección lineal cayera fuera del segmento. Para solventar éste y otros problemas de seguridad, se implementó a partir del procesador 80286 el modo protegido.

En el modo protegido, el registro de segmento ya no contiene la dirección base del segmento, sino un *selector de segmento*, el cual permite seleccionar un *descriptor de segmento*. Este último no es más que una estructura de ocho bytes, la cual posee un formato definido y contiene entre otras cosas la dirección base del segmento en cuestión:



byte 7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
	Permisos de acceso								Dirección base (23-16)								
	Dirección base (15-0)																
	Límite (15-0)																***byte 0***

Descriptor de segmento del 80286



byte 7	Dirección base (31-24)	G	D	0	AV	Límite (19-16)	
	Permisos de acceso	Dirección base (23-16)					
	Dirección base (15-0)						
	Límite (15-0)						***byte 0***

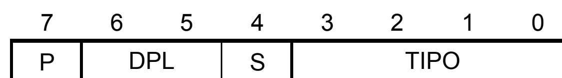
Descriptor de segmento del 80386 y posteriores

De esta manera, la dirección base del segmento pasa ahora a ocupar 24 ó 32 bits (dependiendo del procesador), en contraposición a los 16 bits que pueden

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

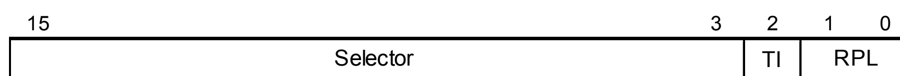
almacenarse en el registro de segmento en modo real. Adicionalmente se cuenta con un campo de límite de segmento que puede ocupar 16 ó 20 bits, lo que determina el tamaño máximo de los segmentos en 64KB o 1MB. El descriptor de segmento del 80386 incluye una bandera (G) que indica que el campo límite debe ser multiplicado por 4K, permitiendo entonces la existencia de segmentos entre 4KB y 4 GB.

El descriptor de 80386 incluye dos banderas adicionales. La primera de ellas (D/B), permite determinar si se está trabajando en modo de 16 ó 32 bits. La segunda (AVL) puede ser utilizada por los sistemas operativos para marcar el segmento descrito como disponible o no (*Available*). El campo de permisos de acceso posee los siguientes subcampos:



En donde P (*present*) es un bit que indica si el descriptor contiene información válida de un segmento o no. El bit DPL (*Descriptor Privilege Level*) indica el nivel de privilegio mínimo para poder acceder al segmento. El bit S (*System*) indica si se trata de un descriptor de sistema o no (en cuyo caso es de código o datos). Por último los cuatro bits correspondientes al campo tipo indican si se trata de un segmento de código o datos, el sentido de expansión del mismo (hacia arriba para datos, hacia abajo para pila), si el segmento es de lectura y/o escritura, y si el mismo ha sido accedido o no.

No todos los bits del registro de segmento están destinados a seleccionar un descriptor:

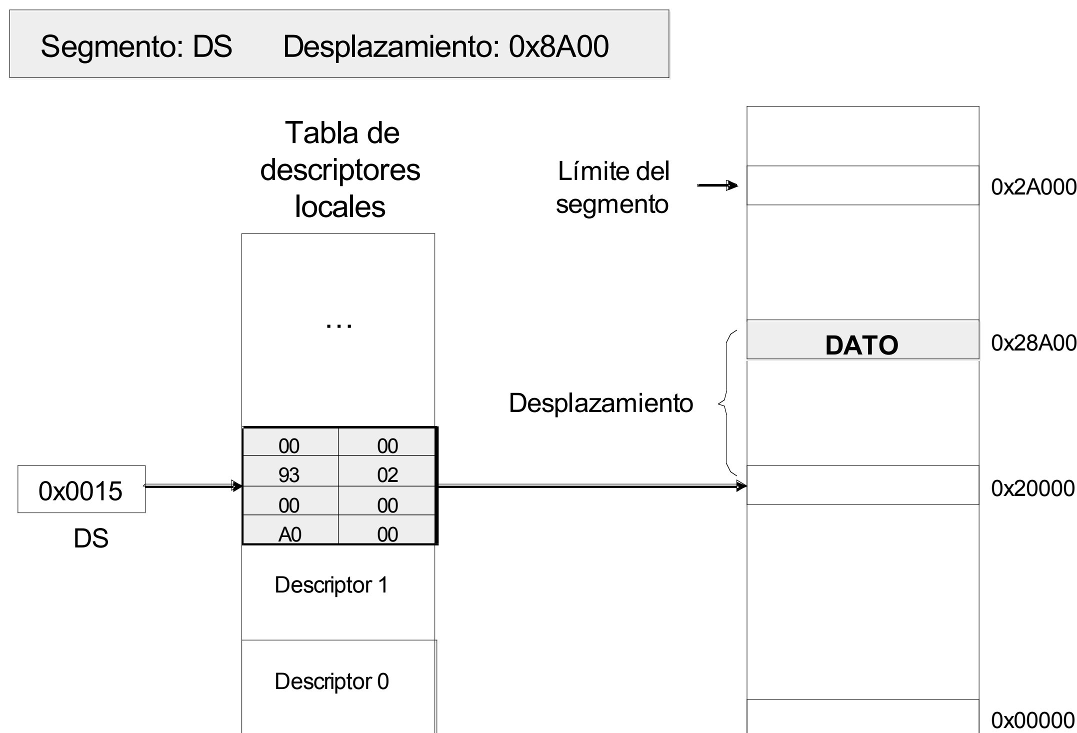


El campo RPL (*Request Privilege* Level) permite indicar el nivel de privilegio con el cual se está accediendo al segmento, siendo 00 el nivel más bajo y 11 el más alto. Este valor debe ser mayor o igual al del campo DPL del descriptor correspondiente. Adicionalmente el campo TI (*Table Indicador*) es una bandera que permite escoger la tabla en la cual será seleccionado el descriptor de segmento con los bits restantes (0 = tabla de descriptores globales, 1 = tabla de descriptores locales). Los descriptores globales son utilizados por todos los

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

programas y a menudo se les conoce como *descriptores de sistema*, mientras que los descriptores locales son exclusivos de un programa particular y a menudo se les conoce como *descriptores de aplicación*.

De esta forma, el mismo acceso a memoria que ejemplificamos anteriormente, se llevaría a cabo de la siguiente manera en modo protegido:



En primer lugar, el descriptor de segmento indica que debe seleccionarse el descriptor número 2 (Selector = 0000000000010) de la tabla de descriptores locales (TI = 1) con un nivel de privilegio de 1 (RPL = 01). De esta manera, se procede a interpretar los ocho bytes correspondientes al mismo en la TDL.

Dicho descriptor indica que la dirección base del segmento en cuestión es la 0x00020000, y que el tamaño del mismo es de 0x0A000, lo que quiere decir que el límite del mismo corresponde a la dirección 0x2A000. Dado que el RPL es mayor que el DPL (00) se permite el acceso al segmento. De manera similar, se verifica que la dirección lineal resultante (0x20000 + 0x8A00) se encuentra dentro de los límites del segmento.

El proceso de traducción de direcciones es un poco más complejo de lo que puede intuirse a partir de esta breve explicación, involucrando el uso de una caché para agilizar el acceso a las tablas de descriptores. Es importante tener en

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

mente también que la memoria puede estar paginada, en cuyo caso será necesario transformar las direcciones lineales (virtuales) a direcciones físicas o reales. Por último, cabe acotar que los registros ES, FS y GS apuntan a segmentos de datos (o seleccionan descriptores de segmentos de datos).

Registro apuntador de instrucción

El registro apuntador de instrucción (eip) o contador de programa contiene la dirección de la próxima instrucción a ejecutarse.

Registros de punto flotante

Son ocho registros los cuales son tratados como una pila. Se denominan %st(0), %st(1), %st(2), etc. %st(0) se ubica en el tope de la pila.

Banderas

Las banderas proveen una manera de obtener información acerca de del estado actual de la máquina y el resultado de procesamiento de una instrucción. La plataforma IA-32 utiliza un registro de 32 bits llamado EFLAGS que contiene las banderas. Las banderas más comunmente usadas son las siguientes:

Bandera	Bit	Nombre
CF	0	Bandera de acarreo (carry flag)
PF	2	Bandera de paridad (parity flag)
AF	4	Bandera de acarreo auxiliar (adjust flag)
ZF	6	Bandera de cero (zero flag)
SF	7	Bandera de signo (sign flag)
DF	10	Bandera de dirección (direction flag)
OF	11	Bandera de desbordamiento (overflow flag)

La bandera de acarreo se activa cuando se produce acarreo en una suma o multiplicación, o un 'préstamo' en una resta entre números sin signo.
La bandera de paridad se usa para indicar si el resultado, en un registro, de una operación matemática es válido.
La bandera de acarreo auxiliar se utiliza en operaciones matemáticas con números decimales codificados en binario (BCD). Se activa si hay acarreo presente.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

La bandera de cero se activa si el resultado de una operación aritmético lógica es cero.

La bandera de signo muestra el bit más significativo del resultado de una operación, el cual denota el signo del número.

La bandera de dirección controla la selección de autoincremento (D=0) o autodecremento (D=1) de los registros %edi o %esi durante las operaciones con cadenas de caracteres. La bandera de dirección sólo se utiliza con las instrucciones para el manejo de cadenas de caracteres.

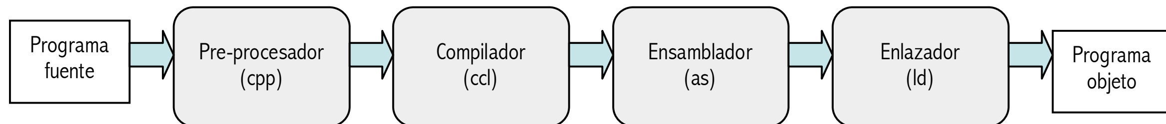
La bandera de desbordamiento se utiliza en la aritmética de enteros con signo cuando un número sobrepasa la capacidad de representación del registro.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Capítulo 2. El entorno de programación

El compilador gcc

El proceso de traducción de un programa escrito en un lenguaje de alto nivel a un archivo objeto ejecutable consta de varias etapas. A continuación se describen las etapas correspondientes al conjunto de herramientas de compilación gcc que permite traducir programas escritos en el lenguaje de programación C.



Para ejemplificar los pasos usaremos un programa fuente denominado ejemplo escrito en lenguaje de programación C, creado en un editor de texto y guardado como ejemplo.c:

```
#include <stdio.h>
int main()
{
        printf("programa de ejemplo\n");
        return 0;
}
```

Con gcc, la traducción de este programa fuente a un archivo objeto ejecutable se realiza de la siguiente manera:

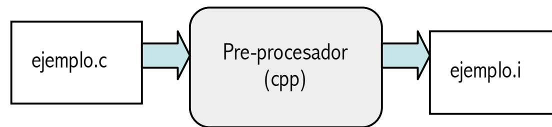
```
linux> gcc –o ejemplo ejemplo.c
```

El compilador lee el archivo ejemplo.c y lo traduce al archivo objeto ejecutable ejemplo. Esto se realiza a través de las cuatro etapas mencionadas.

En la primera etapa el pre-procesador (cpp) modifica el programa original de acuerdo a lo indicado por las directivas que comienzan con el caracter #. En este caso la línea #include <stdio.h> le indica al pre-procesador que debe leer el archivo de encabezado stdio.h e insertar su contenido en el texto del programa.

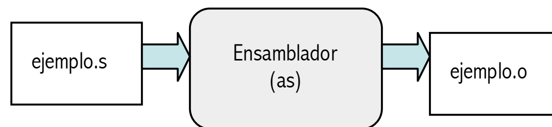
Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

La salida de esta etapa es el archivo intermedio ejemplo.i.

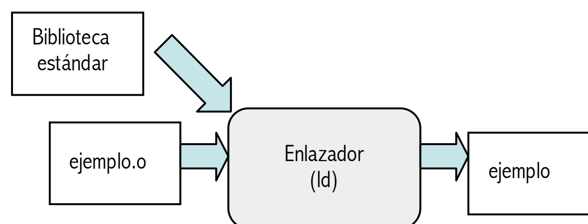


Luego, ejemplo.i entra en la siguiente etapa, el compilador (ccl) el cual traduce el programa a lenguaje ensamblador. La salida de esta etapa es el archivo ejemplo.s.

En la siguiente etapa, el ensamblador (as) lee el archivo ejemplo.s y lo traduce en instrucciones de lenguaje de máquina generando un archivo objeto relocalizable el cual se guarda como ejemplo.o.



Luego el enlazador (ld) enlaza ese archivo objeto con otros archivos asociados a él, en este caso ya que el programa llama una biblioteca (la función printf forma parte de la biblioteca estándar de entrada/salida) el enlazador se encarga de enlazar el programa con esta biblioteca en particular. Su salida es un archivo objeto ejecutable llamado ejemplo.



Para ejecutar el programa se escribe:

linux> ./ejemplo

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Lo cual producirá como salida:

programa de ejemplo
linux>

El compilador gcc se puede ejecutar con varias opciones. Cabe destacar que Linux diferencia entre mayúsculas y minúsculas por lo cual un comando (u opción) en mayúscula será diferente a un comando (u opción) con el mismo nombre en minúscula. Entre las opciones más importantes tenemos:

-o	indica el nombre del archivo de salida, cuando no se usa, el compilador genera un archivo llamado a.out por defecto.
-S	no ensambla, la salida es un archivo en lenguaje ensamblador con el sufijo .s.
-c	no enlaza, la salida será un archivo en código objeto con el sufijo .o su contenido se encuentra en binario. Se puede utilizar la herramienta objdump para desensamblar este archivo y visualizar su contenido.
-O	opción de optimización, se acompaña con un número que indica el nivel de optimización deseado. Se puede escoger entre O0, O1, O2 u O3.
-g	produce información para la depuración del programa.

Usando el programa ejemplo anterior, el comando:

linux> gcc -O2 -S -o ejemplo.s ejemplo.c

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Produce como salida el programa en lenguaje ensamblador:

```
    .file   "ejemplo.c"
    .section    .rodata.str1.1,"aMS",@progbits,1
.LC0:
    .string "programa de ejemplo"
    .text
    .p2align 2,,3
.globl main
    .type   main,@function
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    andl    $-16, %esp
    subl    $12, %esp
    pushl   $.LC0
    call    puts
    xorl    %eax, %eax
    leave
    ret
.Lfe1:
    .size   main,.Lfe1-main
    .ident  "GCC: (GNU) 3.2.2 20030222 (Red Hat Linux 3.2.2-5)"
```

Existe otra opción de (–v, *verbose*), la cual nos permite observar la manera en la que gcc invoca a cada una de las herramientas mencionadas (el compilador, el ensamblador y el enlazador) para generar el archivo ejecutable. Por ejemplo, si repetimos la compilación del archivo ejemplo.s utilizando esta opción, podemos distinguir cada uno de los pasos descritos anteriormente:

linux> gcc -o ejemplo.c ejemplo.c -v

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

En este caso, se pueden observar las siguientes líneas dentro de la salida producida por gcc:

```
...
/usr/libexec/gcc/i386-redhat-linux/4.0.2/cc1 -quiet -v ejemplo.c -
quiet -dumpbase ejemplo.c -auxbase ejemplo -version -o
/tmp/ccVK44SE.s
...
as -V -Qy -o /tmp/cc4ziThv.o /tmp/ccVK44SE.s
...
/usr/libexec/gcc/i386-redhat-linux/4.0.2/collect2 --eh-frame-hdr -m
elf_i386 -dynamic-linker /lib/ld-linux.so.2 -o ejemplo
/usr/lib/gcc/i386-redhat-linux/4.0.2/../../../crt1.o /usr/lib/gcc/i386-
redhat-linux/4.0.2/../../../crti.o /usr/lib/gcc/i386-redhat-
linux/4.0.2/crtbegin.o -L/usr/lib/gcc/i386-redhat-linux/4.0.2 -
L/usr/lib/gcc/i386-redhat-linux/4.0.2 -L/usr/lib/gcc/i386-redhat-
linux/4.0.2/../../.. /tmp/cc4ziThv.o -lgcc --as-needed -lgcc_s --no-
as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed
/usr/lib/gcc/i386-redhat-linux/4.0.2/crtend.o /usr/lib/gcc/i386-redhat-
linux/4.0.2/../../../crtn.o
...
```

Se observa que el gcc llevó a cabo tres pasos:

Llamó al compilador (cc1) pasando el archivo "ejemplo.c" como entrada, y generando el archivo "ccVK44SE.s" como salida.
Llamó al ensamblador (as), pasando la salida del compilador como entrada, y generando el archivo objeto relocalizable "cc4ziThv.o" como salida.
Llamó al enlazador (collect2) pasando como entrada la salida del ensamblador y un conjunto de bibliotecas estándar, y generando como salida el archivo objeto ejecutable "ejemplo"

Luego de observar este resultado, es importante aclarar algunos puntos:

El archivo fuente original puede utilizarse directamente como entrada al compilador. En caso de que la entrada haya sido preprocesada previamente, debe utilizarse la opción –fpreprocessed para indicarle este hecho al cc1.
El enlazador real es ld pero gcc hace la llamada a collect2 debido al funcionamiento interno de esta herramienta.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Cada una de las herramientas llamadas por gcc cuenta con su propio conjunto de opciones que permiten un mayor control sobre todo el proceso de traducción en caso de ser necesario.

Finalmente, es posible realizar todos los pasos del proceso de traducción de manera manual, llamando a cada una de las herramientas con las opciones más adecuadas de acuerdo a lo que se quiere lograr. Por ejemplo, para generar el archivo ejecutable "ejemplo" podemos realizar los siguientes pasos:

```
linux> cpp -o ejemplo.i ejemplo.c
linux> /usr/libexec/gcc/i386-redhat-linux/4.0.2/cc1 -o ejemplo.s ejemplo.i -quiet –
fpreprocessed
linux> as -o ejemplo.o ejemplo.s
linux> ld --eh-frame-hdr -m elf_i386 -dynamic-linker /lib/ld-linux.so.2 -o ejemplo
/usr/lib/gcc/i386-redhat-linux/4.0.2/../../../crt1.o /usr/lib/gcc/i386-redhat-
linux/4.0.2/../../../crti.o /usr/lib/gcc/i386-redhat-linux/4.0.2/crtbegin.o -L/usr/lib/gcc/i386-
redhat-linux/4.0.2 -L/usr/lib/gcc/i386-redhat-linux/4.0.2 -L/usr/lib/gcc/i386-redhat-
linux/4.0.2/../../.. ejemplo.o -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-
needed -lgcc_s --no-as-needed /usr/lib/gcc/i386-redhat-linux/4.0.2/crtend.o
/usr/lib/gcc/i386-redhat-linux/4.0.2/../../../crtn.o
```

Binutils

Como mencionamos anteriormente, si utilizamos la opción –c se produce un archivo en código de máquina (binario). De igual forma podemos obtener este archivo invocando manualmente a las herramientas correspondientes. Sin embargo, existen casos en que necesitamos llevar a cabo el proceso inverso, es decir, obtener la representación en lenguaje ensamblador de un programa en lenguaje de máquina.

En la vida real se dan situaciones en que se nos proporciona una biblioteca previamente compilada (.o, .a, .so) sin ningún tipo de documentación, y debemos averiguar la forma correcta de utilizarla. (Por ejemplo, imagine que se le proporciona la biblioteca para interactuar con un dispositivo de hardware, pero no se le entrega el código fuente ni una documentación que permita conocer las funciones disponibles para ello).

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

En este caso será necesario identificarlas de alguna manera, bien sea obteniendo una lista de los símbolos globales que identifican a dichas funciones (*entry points*) o bien sea desensamblando el código binario y analizando el mismo.

Es posible llevar a cabo estas tareas utilizando un grupo de herramientas provistas por GNU para el manejo de archivos binarios, conocidas como *binutils*, entre las cuales se encuentran:

ld	El enlazador GNU
as	El ensamblador GNU
addr2line	Permite convertir direcciones a nombres de archivo y números de línea
ar	Herramienta para el manejo de bibliotecas estáticas
nm	Lista los símbolos contenidos en un archivo objeto
objcopy	Permite convertir entre formatos de archivo objeto
objdump	Muestra la información contenida en un archivo objeto
readelf	Extrae la información contenida en un archivo objeto en formato ELF
size	Muestra el tamaño de los segmentos de un archivo objeto o una biblioteca estática
strings	Lista todas las cadenas imprimibles de un archivo
strip	Elimina símbolos y secciones de un archivo objeto

De esta forma, podemos utilizar la herramienta objdump para desensamblar el segmento de código del archivo binario ejemplo.o y analizar su contenido:

```
linux> gcc -O2 -c -o ejemplo.o ejemplo.c
linux> objdump -d ejemplo.o
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Lo cual produce la salida:

```
00000000 <main>:
   0: 55                     push   %ebp
   1: 89 e5                  mov    %esp,%ebp
   3: 83 ec 08               sub    $0x8,%esp
   6: 83 e4 f0               and    $0xfffffff0,%esp
   9: 83 ec 0c               sub    $0xc,%esp
   c: 68 00 00 00 00         push   $0x0
  11: e8 fc ff ff ff         call   12 <main+0x12>
  16: 31 c0                  xor    %eax,%eax
  18: c9                     leave
  19: c3                     ret
```

En la cual se identifica claramente el procedimiento main y las instrucciones que conforman el cuerpo del mismo.

La herramienta objdump tiene varias opciones entre las que vale la pena destacar:

-d	desensambla el código del segmento de texto.
-D	desensambla todo el programa incluyendo los datos (el programa trata de traducir toda la información como instrucciones así que en la parte de datos la traducción no se corresponde con la información almacenada).
-G	muestra la información de depuración de programas.
-l	muestra los números de línea.
-r	muestra las entradas de relocalización.
-R	muestra las entradas de relocalización dinámica.
-t	muestra la tabla de símbolos.

Si se incluye información de depuración del archivo binario (utilizando la opción -g), es posible relacionar cada dirección del programa desensamblado con las líneas originales del programa en lenguaje de alto nivel, mediante la utilización de *addr2line*:

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
linux> gcc ejemplo.c -o ejemplo.o -c -g -O2
linux> addr2line -e ejemplo.o -s
0
ejemplo.c:4
11
ejemplo.c:5
19
ejemplo.c:7
```

```
main()
{  <------  pushl %ebp
…

…
printf("programa de ejemplo\n");  <------  call <main+0x12>
…

…
}  <------  ret
```

De esta forma, se observa que la primera instrucción en lenguaje ensamblador (ubicada en la dirección 0x00) se corresponde con el inicio del procedimiento main (línea 4, la llave de apertura). De forma similar, la instrucción *call* (dirección 0x11) corresponde a la llamada a printf en lenguaje C (línea 5). Por último, la instrucción *ret (*dirección 0x19*)*, corresponde a la finalización del procedimiento en lenguaje C (línea 7, la llave de cierre). Es importante tener en cuenta que una instrucción en lenguaje de alto nivel es traducida en una o más instrucciones de bajo nivel, por lo cual dos direcciones proporcionadas a *addr2line* pueden arrojar como salida la misma línea de código fuente original. En capítulos posteriores se dará una explicación más detallada de este tema.

Algunas opciones importantes de esta herramienta son:

-e	permite especificar el archivo de entrada a analizar.
-s	evita que se imprima la ruta completa del archivo en lenguaje de alto nivel al consultar una dirección.
-f	Imprime el nombre de la función a la cual corresponde la línea del programa.

Para archivos binarios que utilicen el formato *ELF*, podemos utilizar la herramienta *readelf*. La misma permite analizar toda la información adicional almacenada en el archivo, como por ejemplo la tabla de símbolos o la información de relocalización:

```
linux> readelf ejemplo.o -s
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

La tabla de símbolos '.symtab' contiene 10 entradas:

Num	Valor	Tam	Tipo	Union	Vis	Ind	Nombre
0	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1	00000000	0	FILE	LOCAL	DEFAULT	ABS	
2	00000000	0	SECTION	LOCAL	DEFAULT	1	ejemplo.c
3	00000000	0	SECTION	LOCAL	DEFAULT	3	
4	00000000	0	SECTION	LOCAL	DEFAULT	4	
5	00000000	0	SECTION	LOCAL	DEFAULT	5	
6	00000000	0	SECTION	LOCAL	DEFAULT	7	
7	00000000	0	SECTION	LOCAL	DEFAULT	6	
8	00000000	26	FUNC	GLOBAL	DEFAULT	1	main
9	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	puts

linux> readelf ejemplo.o –r

La sección de reubicación '.rel.text' en el desplazamiento 0x35c contiene 2 entradas:

Desplazamiento	Info	Tipo	Val. Simbolo	Nom. Simbolo
0000000D	00000501	R_386_32	00000000	.rodata.str1.1
00000012	00000902	R_386_PC32	00000000	puts

Esta herramienta también permite examinar el contenido de cualquier sección o encabezado del programa. Por ejemplo, podemos visualizar las características de todas las secciones, y observar el contenido de la sección de datos de solo lectura, la cual contiene la cadena “programa de ejemplo”:

linux> readelf ejemplo.o –S

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Hay 11 encabezados de sección, comenzando en el desplazamiento: 0xec:

Encabezados de Sección:

[Nr]	Nombre	Tipo	Direc	Desp	Tam	ES	Opt	En	Inf	Al
[0]	.text	NULL	00000000	000000	000000	00		0	0	0
[1]	.text	PROGBITS	00000000	000034	00001a	00	AX	0	0	4
[2]	.rel.text	REL	00000000	00035c	000010	08		9	1	4
[3]	.data	PROGBITS	00000000	000050	000000	00	WA	0	0	4
[4]	.bss	NOBITS	00000000	000050	000000	00	WA	0	0	4
[5]	.rodata.str1.1	PROGBITS	00000000	000050	000014	01	AMS	0	0	1
[6]	.comment	PROGBITS	00000000	000064	00002d	00		0	0	1
[7]	.note.GNU-stack	PROGBITS	00000000	000091	000000	00		0	0	1
[8]	.shstrtab	STRTAB	00000000	000091	000058	00		0	0	1
[9]	.symtab	SYMTAB	00000000	0002a4	0002a4	10		10	8	4
[10]	.strtab	SYMTAB	00000000	000344	000015	00		0	0	1

Clave para Opciones:

W (escribir), A (asignar), X (ejecutar), M (mezclar), S (cadenas)

I (info), L (orden enlazado), G (grupo), x (desconocido)

O (se requiere procesamiento extra del SO) o (específico del SO)

p (específico del procesador)

linux> readelf ejemplo.o –x 5

Volcado hexadecimal de la sección '.rodata.str1.1':

Dirección	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	ASCII
0x00000000	6D	65	6A	65	20	65	64	20	67	6F	72	70	67	6F	72	70	programa de ejem
0x00000010													0	6F	6C	70	plo

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Las principales opciones de *readelf* son las siguientes:

-a	muestra toda la información contenida en el archivo
-h	muestra la información de encabezado del archivo ELF
-S	muestra la información de los encabezados de sección
-s	muestra la tabla de símbolos
-r	muestra la información de relocalización
-x	vacía el contenido de la sección especificada

El depurador gdb

Un depurador es una aplicación que permite correr otros programas, permitiendo al usuario ejercer cierto control sobre los mismos a medida que los estos se ejecutan, y examinar el estado del sistema (variables, registros, banderas, etc.) en el momento en que se presente algún problema. El propósito final de un depurador consiste en permitir al usuario observar y comprender lo que ocurre “dentro” de un programa mientras el mismo es ejecutado.

En los sistemas operativos UNIX/LINUX, el depurador más comúnmente utilizado es gdb, es decir el depurador de GNU. Éste ofrece una cantidad muy extensa y especializada de opciones. Es muy importante entender el hecho de que un depurador trabaja sobre archivos ejecutables. Esto quiere decir que el mismo funciona de forma independiente al lenguaje en que se escribió el programa original, sea éste lenguaje ensamblador o un lenguaje de medio o alto nivel como C.

Ejecución de programas con el depurador

Para iniciar el depurador, se utiliza el comando “gdb”. Esto ocasionará que aparezca un nuevo *prompt* simbolizado por la cadena “(gdb)”, lo que indica que el depurador está listo para recibir una orden (de forma similar al símbolo $ o # en una cónsola de Linux):

```
linux> gdb
GNU gdb Red Hat Linux (6.3.0.0-1.21rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu".
(gdb)
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Una vez dentro del depurador, existen diferentes comandos para analizar el comportamiento de un programa. Un hecho a resaltar es la gran semejanza que existe entre la cónsola proporcionada por GDB y la consola que comúnmente utilizamos para llevar a cabo todas las tareas en Linux. En particular, es importante mencionar que la utilización del tabulador para completar los comandos automáticamente también funciona dentro de gdb. También es posible utilizar las teclas “Flecha Arriba” y “Flecha Abajo” para navegar por los últimos comandos que se han utilizado.

El comando *help* permite obtener ayuda sobre cada uno de los comandos. Esta ayuda funciona de una forma jerárquica. Por ejemplo, si se utiliza el comando *help* sin argumentos, se desplegará una lista con las principales categorías sobre las cuales es posible obtener ayuda:

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
```

Sin embargo, es posible proporcionar información más detallada para obtener la ayuda sobre el comando preciso que se necesite. Por ejemplo, utilizando el comando *help info*, se obtiene una lista de todos los comandos que comienzan con la palabra info, tales como *info registers*, *info variables* o *info stack*. Una vez que se pide ayuda sobre un comando completo, gdb muestra información más detallada acerca del mismo:

```
(gdb) help info registers
List of integer registers and their contents, for selected stack frame.
Register name as argument means describe only that register.
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Dentro del depurador, el primer paso es cargar el programa ejecutable que se quiere depurar. Para ello se utiliza el comando *file*. Por ejemplo, supongamos que se desea depurar el programa de ejemplo que utilizamos en secciones anteriores:

```
(gdb) file ejemplo
Reading symbols from /root/ejemplo...(no debugging symbols found)...done.
Using host libthread_db library "/lib/libthread_db.so.1".
```

El comando “file” no suele utilizarse, ya que (como veremos más adelante) existe una forma más directa para indicar el archivo de entrada. Para ejecutar el programa se utiliza el comando *run*:

```
(gdb) run
Starting program: /root/ejemplo
Reading symbols from shared object read from target memory...(no debugging symbols
found)...done.
Loaded system supplied DSO at 0x531000
(no debugging symbols found)
(no debugging symbols found)
programa de ejemplo
Program exited normally.
```

Observe que el depurador intercala mensajes de depuración con la salida del programa (en este caso “programa de ejemplo”). En especial note que el depurador indica que el programa terminó normalmente. En caso de un error que ocasione una terminación prematura, el depurador mostrará la información apropiada. Finalmente, para salir del depurador se utiliza el commando *quit*.

Ahora bien, supongamos que se tiene un programa llamado *suma.s*, que toma un arreglo de N posiciones, calcula la suma de los elementos que se encuentran en las posiciones impares (A[1], A[3], etc.), e indica si el resultado fue positivo, negativo o cero:

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
        .section .data
RES0:   .asciz "El resultado de la suma es cero (%d)\n"
RESP:   .asciz "El resultado de la suma es positivo (%d)\n"
RESN:   .asciz "El resultado de la suma es negativo (%d)\n"
A:      .long 1,10,-8,7,14,-3,23,-52
N:      .long 8
        .section .text
        .globl _start

_start: xorl %eax, %eax                 # suma = 0
        xorl %ecx, %ecx                 # i = 0

ciclo:  cmpl N, %ecx                    # Mientras i<N
        jge fciclo
        movl A(,%ecx,4), %edx           # Carga A[i]
        addl %edx, %eax                 # suma += A[i]
        addl $1, %ecx                   # i++
        jmp ciclo                       # Repite el ciclo
fciclo: pushl %eax                      # Primer argumento del printf
        cmpl $0, %eax                   # Seleccionar el mensaje
        je R0
        jl RN

RP:     pushl $RESP                     # Segundo argumento del printf
        jmp fin
RN:     pushl $RESN
        jmp fin
R0:     pushl $RES0
        jmp fin

fin:    call printf                     # Muestra el resultado
        addl $8, %esp
        movl $1, %eax                   # Termina el programa
        xorl %ebx, %ebx
        int $0x80
```

En teoría, el programa calcula la suma de los elementos en las posiciones impares, esto es:

A[1] + A[3] + A[5] + A[7] = 10 + 7 + (−3) + (−52) = −38.

Sin embargo, al ensamblar y ejecutar el programa el resultado fue el siguiente:

```
linux> gcc suma.s -o suma -nostartfiles
linux> ./suma
El resultado de la suma es negativo (-8)
```

Si se analiza el programa, es muy fácil descubrir los errores que llevaron a este resultado, ya que los mismos son evidentes. Sin embargo, recomendamos no examinar el código fuente, sino proseguir con la lectura. En las siguientes

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

páginas se utilizará el depurador para determinar lo que está ocurriendo, tal y como se haría en un caso real en el que no se tenga acceso al código fuente, o los errores no sean tan evidentes como para detectarse por simple inspección.

Comandos básicos de control

Todo depurador permite al usuario controlar la ejecución del programa analizado, indicando básicamente cuando detener dicha ejecución, o que fragmentos del programa ejecutar. A continuación se muestra una breve descripción de los comandos más comúnmente utilizados en el depurador gdb para controlar la ejecución de un programa:

break	Permite insertar un punto de parada (*breakpoint*), para detener la ejecución del programa cuando se alcance una instrucción específica. La misma puede ser especificada mediante una etiqueta o una dirección (con un * delante).
watch	Permite insertar un punto de parada, para detener la ejecución del programa cuando una variable o registro sean modificados. El argumento puede ser una etiqueta, una dirección (con un *) o un registro (con un $).
delete	Permite eliminar un punto de parada indicando su número como argumento. Utilizando el argumento "breakpoints" se pueden eliminar todos los puntos de parada.
info breakpoints	Muestra los puntos de parada insertados en el programa, así como información referente a los mismos (número, tipo, dirección, función, etc.). En particular es importante resaltar que muestra las veces que el mismo ha sido alcanzado.
step	Ejecuta el programa hasta alcanzar la siguiente línea de código fuente (si el mismo se encuentra disponible). Básicamente permite ejecutar la próxima instrucción del programa original escrito en lenguaje de alto nivel. Si se especifica un entero N como argumento se ejecutan N instrucciones.
stepi	Ejecuta la próxima instrucción de lenguaje ensamblador del programa. Si se especifica un entero N como argumento se ejecutan N instrucciones.
next	Similar al comando step, pero en caso de encontrarse una llamada a un procedimiento, el mismo es tratado como si fuera una instrucción atómica (es decir, no se lleva a cabo el recorrido interno del procedimiento invocado).
finish	Si el programa se encuentra ejecutando un procedimiento, se procesan todas las instrucciones siguientes hasta que el mismo retorne el control a su invocador.
continue	Permite continuar ejecutando todas las instrucciones restantes del programa, hasta que el mismo finalice, se produzca una excepción o se encuentre un punto de parada.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Para ejemplificar el uso de estos comandos, se procederá a depurar el programa ejecutado en la sección anterior. Esta vez se invocará al depurador pasando como argumento el nombre del archivo ejecutable que se desea depurar, en lugar de utilizar el comando *file*:

```
linux> gdb suma
GNU gdb Red Hat Linux (6.3.0.0-1.21rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...(no debugging symbols found)
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb)
```

A continuación se insertará un punto de parada justo antes de entrar en el ciclo, con el fin de determinar cuantas veces se ejecuta el mismo. Esto se hace de la siguiente forma:

```
(gdb) break ciclo
Breakpoint 1 at 0x8048188
```

El depurador indica la dirección de memoria en la que se encuentra la instrucción en la que se colocó el punto de parada, así como el número asociado a éste. Estos números no son reutilizados. Es posible insertar un punto de parada utilizando una dirección en lugar de una etiqueta (debe añadirse un *). Por ejemplo, con los siguientes comandos se eliminará el punto de parada que acabamos de crear, y se insertará otro en su lugar,esta vez indicando la dirección de la instrucción en la que se desea detener la ejecución y no una etiqueta:

```
(gdb) delete 1
(gdb) break *0x8048188
Breakpoint 2 at 0x8048188
```

Es posible comprobar que efectivamente se ha eliminado el primer punto de parada e insertado el segundo utilizando el commando *info breakpoints*:

```
(gdb) info breakpoints
Num Type           Disp Enb Address    What
2   breakpoint     keep y   0x08048188 <ciclo>
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Dado que se está recorriendo un arreglo de 8 elementos, y solo se van a sumar aquellos almacenados en las posiciones impares, el ciclo debería ejecutarse tan solo cuatro veces. A continuación se procederá a comenzar la ejecución del programa con el comando *run*. Cada vez que se alcance la etiqueta ciclo, el depurador detendrá la ejecución:

```
(gdb) run
Starting program: /root/suma
Reading symbols from shared object read from target memory...(no debugging symbols
found)...done.
Loaded system supplied DSO at 0x355000
(no debugging symbols found)
(no debugging symbols found)

Breakpoint 1, 0x08048188 in ciclo ()
(gdb)
```

Se utilizará el comando *continue* las veces que sea necesario para continuar ejecutando el programa hasta alcanzar el final del mismo. Si se cuentan las veces que se ejecutó el ciclo, nos damos cuenta de que el mismo se está ejecutando ocho veces (el punto de parada es alcanzado nueve veces, pero en esta última la comparación se evalúa como falsa y no se entra al ciclo), es decir, que al parecer el programa está recorriendo todos los elementos del arreglo y sumándolos.

Para verificar esto, se insertará un punto de parada del tipo *watch*, utilizando el comando del mismo nombre, para verificar cuando (y como) es modificada la variable “suma” (que se encuentra en el registro %eax. También se eliminará el breakpoint anterior ya que ha dejado de ser útil. Note que antes de realizar estas acciones reiniciaremos la ejecución el programa (en este caso es posible utilizar el comando *run*, pues sabemos que se detendrá la primera vez que alcance la etiqueta ciclo):

```
(gdb) run
Starting program: /root/suma
……
Breakpoint 2, 0x08048188 in ciclo ()
(gdb) delete 2
(gdb) watch $eax
Watchpoint 3: $eax
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

A partir de este momento se procederá de forma similar a como se hizo con el punto de parada anterior, es decir, se utilizará el comando *continue* para avanzar en la ejecución y dejar que el depurador indique cuando el registro es modificado. Si observa el programa original, notará que esto solo ocurre al momento de sumar cada elemento del arreglo con el acumulador. Si nuestra suposición es correcta, y el problema del programa consiste en que se están sumando todos los elementos, el programa debería detenerse ocho veces, pues el registro %eax será modificado la misma cantidad de veces (una por cada elemento sumado). Efectivamente:

```
(gdb) continue
Continuing.
Watchpoint 3: $eax
Old value = 0
New value = 1
0x08048199 in ciclo ()

(gdb) continue
Continuing.
Watchpoint 3: $eax
Old value = 1
New value = 11
0x08048199 in ciclo ()

(gdb) continue
Continuing.
Watchpoint 3: $eax
Old value = 11
New value = 3
0x08048199 in ciclo ()
```

Después de utilizar tres veces el comando *continue*, comprobamos que efectivamente en las tres primeras iteraciones se han sumado los tres primeros elementos del arreglo (0+1=1; 1+10=11; 11-8=3;). Si se utiliza el comando *continue* seis veces más se verifica que efectivamente en cada iteración se suma un elemento del arreglo, sin importar que el mismo se encuentre en una posición par o no. Esto comprueba la conclusión a la que habíamos llegado en la sección anterior. Esto posiblemente se deba a que el índice "i" no se está actualizando como debería. En la siguiente sección se buscará comprobar este hecho. Si realiza esta experiencia por su cuenta, notará que el programa no finaliza, ya que como el registro %eax se modifica en el procedimiento printf, la

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

ejecución se detiene en este punto. En este caso deje el depurador tal cual, en la próxima sección continuaremos desde aquí.

Comandos básicos de inspección

En la sección anterior se ejecutó dos veces el programa. En la primera se utilizó un *breakpoint*, mientras que en la segunda se utilizó un *watchpoint*. Si algo quedó claro de esta pequeña experiencia, es que el segundo método fue mucho más útil a la hora de sacar conclusiones y comprobar hechos, simplemente porque proporcionó más información que el primero (en este caso, el valor del registro %eax).

Una vez que la ejecución del programa se encuentra detenida (sin importar el tipo de punto de parada utilizado o el motivo por el que se detuvo la ejecución) es posible inspeccionar el estado del programa. Entre las cosas que pueden ser inspeccionadas se encuentran los registros de la máquina (no solo los de propósito general sino también los de propósito específico como el eflags o el eip), el contenido de la memoria, y la pila actual de llamadas.

A continuación se muestra una breve descripción de los comandos más comúnmente utilizados en el depurador gdb para inspeccionar el estado de un programa en ejecución:

info registers	Muestra el contenido de todos los registros. La primera columna indica su valor hexadecimal (binario), mientras que la segunda el contenido del mismo en decimal interpretado como un entero con signo representado en complemento auténtico.
print	Permite mostrar el contenido de un registro o posición de memoria escribiendo como argumento una expresión válida en lenguaje C. Se permite el uso de los operadores * y & para trabajar con apuntadores.
display	Similar al comando print. La diferencia consiste en que el resultado de evaluar la expresión proporcionada se mostrará cada vez que la ejecución se detenga.
disassemble	Desensambla la fracción del segmento de código correspondiente a la función que se está ejecutando actualmente (note que el depurador considera una función a las instrucciones que se encuentran entre una etiqueta y la siguiente).
info variables	Muestra todas las variables globales definidas.
info locals	Muestra todas las variables locales definidas.
backtrace	Muestra la pila activa de llamadas a función.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

A continuación se utilizarán algunos de estos comandos para comprobar que efectivamente el índice utilizado para recorrer el arreglo no se está actualizando debidamente. Eliminamos el *watchpoint* anterior e insertamos un *breakpoint* en la misma instrucción de suma (cuya dirección obtuvimos cada vez que la ejecución se detuvo al modificarse el registro %eax). La razón de hacer este cambio es evitar que la ejecución se detenga en cualquier otra parte del programa en que se modifique %eax, como sucedió con el printf en la ejecución anterior:

```
(gdb) delete 3
(gdb) break *0x08048199
Breakpoint 4 at 0x8048199
```

Ahora se procederá a ejecutar de nuevo el programa desde el comienzo. Dado que la ejecución actual no ha finalizado el depurador solicitará una confirmación. Una vez dada, el programa se detendrá la primera vez que se alcance la instrucción de suma:

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
warning: cannot close "shared object read from target memory": Fichero en formato errÃ³neo
Starting program: /root/suma
…..
Breakpoint 4, 0x08048199 in ciclo ()
(gdb)
```

Una vez alcanzado el punto de parada es posible verificar el estado de los registros mediante el comando *info registers*. Preste especial atención al índice utilizado para recorrer el arreglo durante el ciclo, el mismo se encuentra en el registro %ecx:

```
(gdb) info registers
eax            0x1              1
ecx            0x0              0
edx            0x1              1
ebx            0x1a2fc4         1716164
esp            0xbfeaa1e0       0xbfeaa1e0
ebp            0x0              0x0
esi            0xbfeaa1ec       -1075142164
edi            0x8048184        134513028
eip            0x8048199        0x8048199
eflags         0x202            514
…..
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Aquí se hace evidente el primer error: El índice debería comenzar en uno, ya que éste corresponde al primer elemento impar del arreglo. Sin embargo se observa que el primer elemento que se carga es el que se encuentra en la posición cero. En este caso es el contenido del registro %edx (1), que es en donde se cargó previamente. Es posible utilizar el comando *print* para comprobar efectivamente quienes son los elementos A[0], A[1] y A[%ecx] (este último cargado en %edx)

```
(gdb) print *(&A+0)
$1 = 1
(gdb) print *(&A+1)
$2 = 10
(gdb) print *(&A+$ecx)
$3 = 1
```

Ahora bien, suponga que se desea visualizar exactamente la misma información (el valor de i, y el valor de A[i]) cada vez que la ejecución se detenga en la instrucción de suma. Es posible automatizar este proceso mediante el uso del comando *display*:

```
(gdb) display $ecx
10: $ecx = 0
(gdb) display *(&A+$ecx)
11: *(&{<data variable, no debug info>} 134517372 + $ecx) = 1
```

Utilizando el comando *continue* tres o cuatro veces, se observa la forma en la que varía el índice en cada iteración del ciclo, y el elemento que es sumado a acumulador en cada caso:

```
Breakpoint 4, 0x08048199 in ciclo ()
11: *(&{<data variable, no debug info>} 134517372 + $ecx) = 10
10: $ecx = 1
(gdb) continue
Continuing.

Breakpoint 4, 0x08048199 in ciclo ()
11: *(&{<data variable, no debug info>} 134517372 + $ecx) = -8
10: $ecx = 2
(gdb) continue
Continuing.

Breakpoint 4, 0x08048199 in ciclo ()
11: *(&{<data variable, no debug info>} 134517372 + $ecx) = 7
10: $ecx = 3
(gdb) continue
Continuing.
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
Breakpoint 4, 0x08048199 in ciclo ()
11: *(&{<data variable, no debug info>} 134517372 + $ecx) = 14
10: $ecx = 4
```

Esta pequeña experiencia permite determinar el segundo error del programa: el índice se está actualizando de uno en uno, mientras que en cada iteración debería moverse dos posiciones (ya que se quiere sumar solo los elementos impares). De esta forma, se puede concluir que las instrucciones a corregir en el programa original son las siguientes:

```
xorl %ecx, %ecx    ➔    movl $1, %ecx    # El índice se inicializa en uno.
addl $1, %ecx      ➔    addl $2, %ecx    # Obviar los elementos pares.
```

Sin embargo, no es necesario corregir las instrucciones y volver a compilar el programa original. En la siguiente sección se mostrará como probar las correcciones sugeridas desde el depurador.

Alteración de la ejecución del programa

Mientras se está depurando un programa, es posible modificar el estado del mismo (cambiando el valor de los registros o la memoria), y alternado en consecuencia de forma indirecta la ejecución del mismo. Esto es muy útil cuando se ha determinado un problema y se quiere probar la solución de forma rápida. En otras palabras, es un mecanismo que permite contestar preguntas del tipo "¿Qué pasaría si este registro tuviera el valor 0?" o "¿Qué pasaría si el apuntador de la pila se encontrara una posición más arriba?".

A continuación se muestra una breve descripción de los comandos más comúnmente utilizados para alterar el estado de un programa en ejecución:

set	El comando set permite evaluar una expresión y asignar su resultado a un registro o una posición de memoria mediante el operador de asignación. Se diferencia de print en que el comando set no imprime el resultado en la cónsola.
print	Dado que el comando print evalua la expresión que se pasa como argumento (la cual puede contener el operador de asignación '=') el mismo puede ser utilizado para alterar el contenido de un registro o posición de memoria.
display	Similar al comando print. La diferencia consiste en que la asignación será llevada a cabo cada vez que la ejecución se detenga.
jump	Este comando permite continuar la ejecución en un punto del programa distinto a la instrucción en que se detuvo. La instrucción a la que se saltará puede ser indicada mediante una etiqueta o mediante una dirección (utilizando *).

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Para ejemplificar el uso de estos comandos, se procederá a modificar los valores del índice en el programa anterior, con el fin de comprobar si efectivamente el resultado será correcto al cambiar las dos instrucciones determinadas en la sección anterior. El primer cambio (inicializar el índice en uno y no en cero) se realizará de forma manual mediante el uso del comando *set*. Para ello se debe insertar un nuevo punto de parada en la primera instrucción del programa, ubicada en la etiqueta _start:

```
(gdb) break _start
Breakpoint 5 at 0x8048184
```

La segunda modificación (incrementar el índice de dos en dos), se simulará mediante el uso del comando *display*. Dado que existe un punto de parada dentro del ciclo (en la instrucción de suma), y sabemos que en la versión actual el índice se incrementa en uno con cada iteración, se procederá a asignarle al índice su valor más uno (%ecx = %ecx + 1) cada vez que la ejecución se detenga en dicho punto de parada:

```
(gdb) display $ecx=$ecx+1
12: $ecx = $ecx + 1 = 5
```

Finalmente se debe reiniciar la ejecución del programa. Como en el caso anterior, será necesario confirmar. Observe que la ejecución se detiene en la primera instrucción:

```
(gdb) run
……
Breakpoint 5, 0x08048184 in _start ()
12: $ecx = $ecx + 1 = -1080108287
11: *(&{<data variable, no debug info>} 134517372 + $ecx) = Cannot access memory at
address 0x67ffe80
Disabling display 11 to avoid infinite recursion.
```

Note que el *display* que se había insertado anteriormente (A[%ecx]) no pudo ser procesado debido a que el valor original de %ecx ocasiona que la referencia a memoria no sea válida. El depurador automáticamente deshabilitará este *display* para los sucesivos puntos de parada que sean alcanzados. Es posible utilizar el comando *info display* para comprobar cuales están habilitados y cuales no (observe la letra ‘y’ o ‘n’ que refleja este hecho):

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
(gdb) info display
Auto-display expressions now in effect:
Num Enb Expression
12:   y  $ecx = $ecx + 1
11:   n  *(&{<data variable, no debug info>} 134517372 + $ecx)
10:   y  $ecx
```

Hasta que la ejecución no alcance el ciclo, no tiene sentido habilitar ningún *display*. En consecuencia, conviene deshabilitarlos todos mediante el comando *disable display*:

```
(gdb) disable display
(gdb) info display
Auto-display expressions now in effect:
Num Enb Expression
12:   n  $ecx = $ecx + 1
11:   n  *(&{<data variable, no debug info>} 134517372 + $ecx)
10:   n  $ecx
```

Ahora bien, sabemos que la ejecución se encuentra detenida en la primera instrucción del programa. Puede utilizarse el comando *disassemble* para comprobar este hecho:

```
(gdb) disassemble
Dump of assembler code for function _start:
0x08048184 <_start+0>:  xor    %eax,%eax
0x08048186 <_start+2>:  xor    %ecx,%ecx
End of assembler dump.
```

Este comando desensambla por defecto la “función” en la que se encuentra la próxima instrucción a ejecutar. Sin embargo, es posible especificar un rango de direcciones a desensamblar, por ejemplo:

```
(gdb) disassemble 0x08048184 0x080481A0
Dump of assembler code from 0x8048184 to 0x80481a0:
0x08048184 <_start+0>:  xor    %eax,%eax
0x08048186 <_start+2>:  xor    %ecx,%ecx
0x08048188 <ciclo+0>:   cmp    0x804929c,%ecx
0x0804818e <ciclo+6>:   jge    0x804819e <fciclo>
0x08048190 <ciclo+8>:   mov    0x804927c(,%ecx,4),%edx
0x08048197 <ciclo+15>:  add    %edx,%eax
0x08048199 <ciclo+17>:  add    $0x1,%ecx
0x0804819c <ciclo+20>:  jmp    0x8048188 <ciclo>
0x0804819e <fciclo+0>:  push   %eax
0x0804819f <fciclo+1>:  cmp    $0x0,%eax
End of assembler dump.
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

En cualquier caso, lo importante es que luego de ejecutar las próximas dos instrucciones el registro %ecx quedará inicializado en cero. En ese momento corregiremos esto manualmente utilizando el comando *set* para inicializarlo en uno:

```
(gdb) stepi 2
0x08048188 in ciclo ()
```

Note que la ejecución se detuvo en la primera instrucción del ciclo (dirección 0x08048188). En este momento es posible modificar manualmente el índice (usando *set*), habilitar de nuevo los *display* (usando *enable display*), y proseguir la ejecución del programa (usando *continue*):

```
(gdb) set $ecx=1
(gdb) enable display
(gdb) continue
Continuing.
Breakpoint 4, 0x08048199 in ciclo ()
12: $ecx = $ecx + 1 = 2
11: *(&{<data variable, no debug info>} 134517372 + $ecx) = -8
10: $ecx = 2
```

Observe que el registro %ecx comenzó el ciclo con un valor de uno. Ahora vale dos, después de efectuarse la suma ocasionada por el primer *display*. Posteriormente se ejecutará la instrucción que lo incrementaba en uno en el programa original, con lo cual en la próxima iteración del ciclo el registro contendrá el valor tres, luego cinco y así sucesivamente. Es decir, que los cambios han funcionado.

Sin embargo, note que el *display* 11 (este número puede variar) indica que el elemento cargado fue -8, que en realidad no es el elemento A[1] sino A[2]. No se deje engañar, esto simplemente es consecuencia de que la expresión del *display* 11 es evaluada luego de que el *display* 12 ha actualizado %ecx incrementándolo en uno. El valor que realmente se cargó se encuentra en el registro %edx:

```
(gdb) print $edx
$5 = 10
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Efectivamente, el elemento A[1] es 10. Es posible dejar el display tal y como está, y el resultado será correcto ya que el mismo no afecta a la ejecución del programa. Sin embargo, conviene modificarlo para poder observar de forma correcta el elemento cargado y el índice utilizado en las iteraciones restantes:

```
(gdb) display $edx
13: $edx = 10
(gdb) display $ecx
14: $ecx = 2
(gdb) delete display 10
(gdb) delete display 11
(gdb) info display
Auto-display expressions now in effect:
Num Enb Expression
14:   y  $ecx
13:   y  $edx
12:   y  $ecx = $ecx + 1
(gdb)
```

Utilizando cuatro veces más el comando *continue*, se completarán las tres iteraciones restantes, mostrando en cada una de ellas los valores correctos de i (%ecx) y A[i] (%edx). De igual forma compruebe que el resultado es correcto (-38), y coincide con lo que habíamos pronosticado al analizar el comportamiento del programa original:

```
(gdb) continue
Continuing.
Breakpoint 4, 0x08048199 in ciclo ()
16: $ecx = 3
15: $edx = 7
14: $ecx = $ecx + 1 = 4

(gdb) continue
Continuing.
Breakpoint 4, 0x08048199 in ciclo ()
16: $ecx = 5
15: $edx = -3
14: $ecx = $ecx + 1 = 6

(gdb) continue
Continuing.
Breakpoint 4, 0x08048199 in ciclo ()
16: $ecx = 7
15: $edx = -52
14: $ecx = $ecx + 1 = 8

(gdb) continue
Continuing.
El resultado de la suma es negativo (-38)
Program exited normally.
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Una vez que estamos seguros de los errores y de la efectividad de las soluciones propuestas, podemos modificar las instrucciones adecuadas en el programa original, no solo con la certeza de que al compilarlo de nuevo el programa funcionará de forma adecuada, sino con el conocimiento de cual era el problema y con una justificación clara de porqué las modificaciones hechas han sido exitosas.

Uso de la opción –g de gcc

En caso de que contar con el código fuente del programa que se desea depurar, es posible compilarlo de una forma especial en la que se agregan nuevas secciones (.debug, .line) que contienen información para facilitar el proceso de depuración, tales como la correspondencia entre las líneas del programa original en lenguaje C y las instrucciones en lenguaje ensamblador.

Para incluir estas secciones en un archivo ejecutable, el mismo debe ser compilarlo utilizando la opción –g. Por ejemplo vamos a compilar el archivo “ejemplo.c” utilizando la opción (ejemplo1) y sin utilizarla (ejemplo2). Compare la diferencia de tamaño entre los archivos:

```
linux>  gcc ejemplo.c -o ejemplo1 -g
linux>  gcc ejemplo.c -o ejemplo2
linux>  ll example?
-rwxr-xr-x  1 root root 5805 jul 26 11:28 ejemplo1
-rwxr-xr-x  1 root root 4693 jul 26 11:29 ejemplo2
```

Esta diferencia de tamaño (aproximadamente un 25%) se debe a que el primer archivo contiene una cantidad de información adicional con fines de depuración. En general es recomendable utilizar esta opción mientras se están realizando pruebas con el programa, pero una vez que se vaya a compilar la versión final, nunca utilice esta opción. Esto en primer lugar para generar un archivo más compacto, pero la razón de mayor peso es proporcionar la menor cantidad de información posible que pueda ser utilizada para llevar a cabo procesos de ingeniería reversa con su programa.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Veamos algunos ejemplos de la información adicional incluida mediante el uso del comando –g. En primer lugar, observe lo que ocurre al intentar utilizar los comandos mostrados a continuación con el programa que no contiene información adicional:

```
[root@localhost ~]# gdb ejemplo2
GNU gdb Red Hat Linux (6.3.0.0-1.21rh)
…..
This GDB was configured as "i386-redhat-linux-gnu"...(no debugging symbols found)
Using host libthread_db library "/lib/libthread_db.so.1".

(gdb) list 4,7
No symbol table is loaded.  Use the "file" command.

(gdb) info line 5
No symbol table is loaded.  Use the "file" command.

(gdb) break main
Breakpoint 1 at 0x8048382
(gdb) run
Starting program: /root//ejemplo2
Reading symbols from shared object read from target memory...
(no debugging symbols found)...done.
Loaded system supplied DSO at 0x320000
(no debugging symbols found)
(no debugging symbols found)
Breakpoint 1, 0x08048382 in main ()
```

Note las secciones resaltadas en negrita. Al cargar el programa, gdb no encontró información de depuración, y en consecuencia ninguno de los comandos especializados que se probaron funcionaron. En el caso del archivo compilado con la información de depuración si tienen efecto:

```
[root@localhost ~]# gdb ejemplo1
GNU gdb Red Hat Linux (6.3.0.0-1.21rh)
This GDB was configured as "i386-redhat-linux-gnu"...
Using host libthread_db library "/lib/libthread_db.so.1".

(gdb) list 4,7
4      {
5              printf("programa de ejemplo\n");
6              return 0;
7      }

(gdb) info line 5
Line 5 of "ejemplo.c" starts at address 0x8048398 <main+28>  and ends at 0x80483a8
<main+44>.

(gdb) break main
Breakpoint 1 at 0x8048398: file ejemplo.c, line 5.
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
(gdb) run
Starting program: /root/ejemplo1
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0x488000
Breakpoint 1, main () at ejemplo.c:5
5            printf("programa de ejemplo\n");
```

Note lo fácil que resulta relacionar las instrucciones de alto nivel con sus correspondientes instrucciones en lenguaje de máquina utilizando el depurador. Además cada vez que el programa alcance un punto de parada, mostrará información adicional referida al código fuente original. Se deja al lector la tarea de investigar que otros comandos adicionales provee gdb para ser utilizados con aquellos archivos que contienen información de depuración.

Conclusiones y detalles finales

El depurador es una herramienta extremadamente útil para todo programador, no importa el nivel en que escriba el código, el lenguaje que utilice o el área a la que se dedique. En el mundo real se presenta una gran cantidad de situaciones en las que tenemos que depurar programas hechos por otras personas, bien sea compañeros de trabajo o empresas de desarrollo de software. En la mayor parte de estos casos no contamos con el código fuente o la documentación adecuada.

Aún cuando se trata de código propio, el uso de un depurador no sólo permite corregir los errores en menos tiempo sino que a la hora de reportar cual era la falla o la forma en la que se solucionó se tiene información sólida que respalda los cambios.

Por todo esto y mucho más es importante el aprender a utilizar un depurador. Obviamente es necesario tener una idea lo suficientemente clara del lenguaje ensamblador y sus principios básicos de funcionamiento de un computador moderno. No es necesario ser un experto en determinado lenguaje ensamblador o en depurador específico, ya que cualquiera de estos puede cambiar. Lo que es realmente importante es comprender el funcionamiento general del depurador, con el fin de poder aprender a utilizar de forma efectiva cualquier implementación del mismo.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Para finalizar, se enumeran algunos detalles del depurador gdb que son muy útiles y permiten agilizar mucho el proceso de depuración una vez que se ha adquirido la práctica necesaria:

- Es posible presionar simplemente la tecla ENTER sin escribir nada en la consola del depurador para repetir el último comando. Esto es sumamente útil para realizar una ejecución paso a paso (se ejecuta una instrucción con el comando *stepi* y luego se presiona ENTER sucesivamente para avanzar de una instrucción a otra).

- Es posible escribir un comando de forma incompleta si el mismo puede ser interpretado sin ambigüedad por el depurador. Por ejemplo, se pueden utilizar los comandos *del 1* o *dele 1* para eliminar el punto de parada número 1 (en lugar de escribir *delete 1*). Sin embargo no es válido utilizar el comando *"de 1"*, ya que existen otros dos comandos que comienzan con esas dos letras (*"define"* y *"detach"*).

- Si se desea depurar un programa que recibe argumentos desde la línea de comandos, es posible hacerlo, especificando los mismos como parámetros del comando *run*. Por ejemplo, suponga que se tiene un programa llamado "cuadrado", el cual toma como argumento un entero e imprime el cuadrado del mismo. En condiciones normales se podría ejecutar como "./cuadrado 2". Para depurar esta ejecución, primero se carga el programa en el depurador (gdb cuadrado), y a continuación se ejecuta pasando el argumento como parámetro al comando *run* (*run 2*).

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Capítulo 3. El programa en lenguaje ensamblador

Definiciones generales

Todo programa en lenguaje ensamblador tiene una estructura similar y utiliza elementos comunes. Los conceptos más importantes que se deben manejar son los siguientes:

Directiva- una directiva es una línea de programa que provee información para el ensamblador. Las directivas comienzan con un punto (.).

Identificador: Es una letra seguida de cualquier cantidad de letras o dígitos (y en algunos casos caracteres especiales).

Etiqueta- Es un símbolo (identificador) que identifica una línea de programa, va seguida de dos puntos (:).

Instrucción- Las instrucciones son los enunciados que el ensamblador traduce a un programa objeto.

Comentario- Texto que aparece después de un caracter de inicio de comentario (#), el ensamblador ignora los comentarios. Son muy importantes para desarrollar programas legibles.

Secciones

Un programa escrito en lenguaje ensamblador se compone de varias secciones. Las secciones más comunes son: la sección de texto, la sección de datos y la sección bss.

En la sección de texto se escriben las instrucciones, en la sección de datos los datos inicializados y en la sección bss las variables sin inicializar.

Cada una de las secciones se declara por medio de una directiva. Para declarar las secciones mencionadas se usan las siguientes directivas:

.section .text para la sección de texto (instrucciones)

.section .data para la sección de datos (datos inicializados)

.section .bss para la sección bss (datos sin inicializar)

Comúnmente las secciones se colocan en la siguiente secuencia:

```
.section .data
.section .bss
.section .text
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Las siglas bss corresponden a "block storage start", que significa inicio de bloque de almacenamiento.

La ventaja de declarar variables en la sección .bss es que esos datos no se incluyen en el programa ejecutable y por lo tanto el tamaño total del programa es menor al tamaño generado por la declaración equivalente en la sección .data.

A diferencia de las secciones de datos y bss, la sección de texto es de solo lectura. Esto quiere decir que la misma no puede ser modificada en tiempo de ejecución. El propósito de esto consiste en que si existen múltiples instancias del programa ejecutándose, todas comparten la misma sección de texto (se carga una sola vez en memoria), mientras que cada una tiene su propia sección de datos y/o bss.

Sin embargo, existen datos dentro de un programa que nunca son modificados, tales como las constantes, o los literales no asociados a una variable (como la cadena “Hola Mundo” utilizada en la llamada a printf en el capítulo anterior). Con el fin de compartir estos datos de la misma forma que ocurre con las instrucciones, los mismos pueden ser almacenados dentro de una sección especial llamada rodata (Read Only Data, Datos de solo lectura). La misma se comporta de la misma forma que la sección de datos, pero no permite operaciones de escritura sobre los datos declarados dentro de la misma.

Punto de inicio

Cuando se realiza la traducción del programa en lenguaje ensamblador a un archivo ejecutable, el enlazador necesita saber el punto de inicio en el código. Para ello se declara una etiqueta: _start la cual indica a partir de qué instrucción se comienza a ejecutar el código.

Esta etiqueta debe ser declarada como global, es decir, que esté disponible para aplicaciones externas; esto se logra utilizando la directiva .globl.

Finalización del programa

El lenguaje ensamblador gas no provee una instrucción de fin de ejecución, esto se logra mediante una llamada al sistema. Para realizar esta llamada se pasan dos parámetros: el valor 1 en el registro %eax indica el código de llamada a exit (salida); el valor 0 en el registro %ebx indica la salida normal del programa.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

La finalización se realiza de la siguiente manera:

```
movl $1, %eax      # código de llamada a exit
movl $0, %ebx      # salida normal
int  $0x80         # llamada al sistema
```

Otra manera de finalizar el programa es llamando a la función "exit", para ello se debe pasar el valor cero como parámetro y luego invocar la función de la siguiente manera:

```
pushl $0           # pasa 0 como parámetro (salida normal)
call exit          # llamada a la función de salida
```

Estructura general

En general la estructura de un programa en lenguaje ensamblador tiene la siguiente forma:

```
.section .data
            # aqui se declaran variables inicializadas
.section .bss
            # aqui van las variables declaradas pero sin inicializar
.section .text
.globl _start
_start:     # esta etiqueta indica el inicio del programa principal

            # aqui van las instrucciones


      movl $1, %eax      # estas tres instrucciones corresponden a
      movl $0, %ebx      # la finalización del programa
      int  $0x80
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Capítulo 4. Definición de datos

En las secciones de datos se definen diferentes tipos de datos tales como variables inicializadas o no, formatos de lectura y escritura, mensajes a mostrar por pantalla, etc.

Definición de datos inicializados

En la sección .data se definen los datos inicializados, para ello se pueden utilizar las siguientes directivas:

Directiva	Tipo de dato
.ascii	cadena de caracteres
.asciz	cadena de caracteres con caracter de culminación (null)
.string	equivalente a .asciz
.byte	valor de un byte (8 bits)
.double	número de punto flotante precisión doble (64 bits)
.float	número de punto flotante precisión simple (32 bits)
.long	número entero de 4 bytes (32 bits)
.int	equivalente a .long
.octa	número entero de 16 bytes
.quad	número entero de 8 bytes (64 bits)
.short	número entero de 2 bytes (16 bits)

El formato para estas directivas es el siguiente:

etiqueta: directiva valor

Ejemplo: declaración de variables inicializadas

```
.section .data
cadena:      .ascii “abcdefghijk”
var1:        .long 0
var2:        .float 3.14
var3:        .double 2345.7865
```

En este ejemplo se declara una cadena de caracteres llamada *cadena* cuyo contenido es: `abcdefghijk;` una variable entera de 32 bits llamada `var1` que se inicializa en cero, una variable punto flotante de 32 bits llamada `var2` que se inicializa con el valor 3.14 y una variable punto flotante de 64 bits llamada `var3` que se inicializa con el valor 2345.7865, nótese que el lenguaje utiliza el punto para los decimales.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Se pueden definir múltiples valores en la misma línea. Cada uno de ellos será guardado en memoria en el orden en el cual fueron declarados.

Ejemplo: declaración de múltiples valores con una misma etiqueta

```
.section .data
var: .long 10, 20, 30, 40, 50
```

En este caso cuando se lee la variable *var* arroja el valor 10, para poder leer el siguiente valor se debe incrementar la dirección de *var* en 4 bytes (debido a que la variable está declarada como long, es decir de 32 bits) de esta manera se usa la etiqueta *var* como la dirección inicial de estos valores y su tratamiento es el de un arreglo donde cada acceso se realiza tomando *var* como posición inicial lo cual sería equivalente a decir *var*[0] y las posiciones siguientes como un desplazamiento de 4 bytes cada uno. Para leer por ejemplo el valor 30 se accede como *var*+8.

Cuando se definen las variables el sistema las guarda en forma consecutiva en memoria. Por ejemplo si se definen variables de 16 bits y luego se leen usando instrucciones de 32 bits el sistema no produce un mensaje de error y accede los bytes consecutivos leyendo datos no válidos.
Estas directivas también se pueden utilizar en la sección .rodata la cual es de solo lectura tomando en consideración que los valores de las variables no podrán ser modificados.

Definición de constantes

Las constantes se pueden definir en cualquier parte del programa, sin embargo se recomienda hacerlo en la sección de definición de datos (.data o .rodata) por claridad del programa. La definición de constantes se hace usando la directiva .equ, el formato para esta directiva es:

directiva   símbolo, valor

Ejemplo: definición de constantes

```
.section .data
.equ escala, 32
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Cuando se declara una constante su valor no puede ser modificado por el programa. Esta declaración "nombra" el valor 32 como `escala`. La constante se puede utilizar como un inmediato, como desplazamiento o como dirección de memoria. Cuando se usa como un inmediato se debe anteponer un símbolo de dólar, en este caso para usar `escala` como `inmediato` se coloca: `$escala`.

Definición de datos sin inicializar

En la sección .bss se definen variables sin inicializar, es decir se reserva espacio en memoria y se nombra con una etiqueta. Para definir datos en la sección .bss se usan las directivas:

Directiva	Descripción
.comm	declara un área de memoria para datos sin inicializar
.space	equivalente a .comm
.lcomm	declara un área local de memoria para datos sin inicializar

La directiva .lcomm se usa para datos locales, que no serán usados fuera del código local.

El formato, para estas directivas es el siguiente:

etiqueta directiva, tamaño en bytes

Ejemplo: declaración de un área de memoria sin inicializar

```
.section .bss
area:  .space, 100
```

Se declara una variable llamada `area` de `100` bytes de tamaño.

La ventaja de declarar variables en la sección .bss es que esos datos no se incluyen en el programa ejecutable y por lo tanto el tamaño total del programa es menor al tamaño generado por la declaración equivalente en la sección .data.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Capítulo 5. Instrucciones

Las instrucciones en gas tienen un sufijo que indica el tamaño del dato sobre el cual actúa la instrucción.

Sufijo	Tamaño
b	byte
w	word (2 bytes)
l	long (4 bytes)

El número de operandos de una instrucción varía dependiendo de la instrucción en particular. En general las instrucciones tienen la forma:

instrucción operando fuente, operando destino

Los operandos se pueden clasificar en tres tipos:

- Inmediato: para valores constantes.
- Registro: denota el contenido de uno de los registros.
- Referencia a memoria: denota el contenido de una posición de memoria direccionada por la referencia.

Hay varias maneras de obtener la información las cuales se pueden resumir en la siguiente tabla:

Tipo de direcionamiento	Expresión	Valor del operando
Inmediato	$inm	Inm
Por registro	reg	R[reg]
Absoluto	Inm	M[inm]
Indirecto	(reg)	M[R[reg]]
Base+desplazamiento	Inm(reg)	M[Inm+R[reg]]
Indexado	(regb, regi)	M[R[regb]+R[regi]]
Indexado	Inm(regb, regi)	M[Inm+R[regb]+R[regi]]
Escalado	(, regi, e)	M[R[regi]*e]
Escalado	Inm(, regi, e)	M[Inm+R[regi]*e]
Escalado	(regb, regi, e)	M[R[regb]+R[regi]*e]
Escalado	Inm(regb, regi, e)	M[Inm+R[regb]+R[regi]*e]

`inm` denota un inmediato.

`reg` denota un registro, `regb` un registro base y `regi` un registro índice.

e es la escala la cual puede ser 1, 2, 4 ó 8.

`R[reg]` significa el contenido del registro `reg`.

`M[x]` significa el contenido de la posición de memoria con dirección x.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Ejemplo: valores para cada modo de direccionamiento, asumiendo los contenidos de: `%eax= 0x100` y `%ecx= 0x10`

Expresión	Ejemplo	Valor del operando	Valor
$inm	$0x65	Inm	0x65
reg	%eax	R[reg]	0x100
Inm	0x120	M[inm]	M[0x120]
(reg)	(%eax)	M[R[reg]]	M[0x100]
Inm(reg)	8(%eax)	M[Inm+R[reg]]	M[0x108]
(regb, regi)	(%eax, %ecx)	M[R[regb]+R[regi]]	M[0x110]
Inm(regb, regi)	8(%eax, %ecx)	M[Inm+R[regb]+R[regi]]	M[0x118]
(, regi, e)	(, %eax, 4)	M[R[regi]*e]	M[0x400]
Inm(, regi, e)	8(, %eax, 4)	M[Inm+R[regi]*e]	M[0x408]
(regb, regi, e)	(%eax, %ecx, 2)	M[R[regb]+R[regi]*e]	M[0x120]
Inm(regb, regi, e)	12(%eax, %ecx, 2)	M[Inm+R[regb]+R[regi]*e]	M[0x12C]

El valor inmediato se puede expresar en decimal o en hexadecimal como se puede observar en el siguiente ejemplo:

Asumiendo el contenido de `%eax=0x100`

Operando	Expresión	Valor
12(%eax)	M[12+contenido de %eax]	M[0x10C]
0xC(%eax)	M[0xC+contenido de %eax]	M[0x10C]

Instrucciones de movimiento de datos

La instrucción mov

La instrucción mov permite el movimiento de datos, ya que *gas* utiliza un prefijo para señalar el tamaño de los datos podemos tener tres opciones al momento de realizar una transferencia de datos:

El formato de la instrucción es:

Instrucción	Efecto	Descripción
movb Fuente, Destino	Destino←Fuente	mueve 1 byte
movw Fuente, Destino	Destino←Fuente	mueve 2 bytes
movl Fuente, Destino	Destino←Fuente	mueve 4 bytes

Operandos Válidos		Ejemplo
inmediato	registro	movl $25, %eax
inmediato	memoria	movb $10, etiqueta
registro	registro	movw %bx, %ax
registro	memoria	movl %eax, etiqueta
memoria	registro	movl etiqueta, %eax

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Movimiento de datos inmediatos a registro o a memoria

Los datos inmediatos se especifican directamente en la instrucción. Deben estar precedidos por el símbolo dólar para indicar que son datos inmediatos. Pueden estar expresados en decimal o hexadecimal.

Ejemplos:

```
movb  $45, %ah          # mueve 45 al registro de 8 bits %ah
movw  $123, %ax         # mueve el valor 123 al registro %ax
movl  $85, %eax         # mueve el valor 85 al registro %eax
movl  $5678, %eax       # mueve el valor 5678 al registro %eax
movl  $0x100, %ebx      # mueve el valor 0x100 en hexadecimal al
                          registro %ebx
movl  $12, 0x100        # mueve 12 a la posición de memoria 0x100
movl  $12, a            # mueve 12 a la posición de memoria etiquetada
                        # a, esta variable debe estar previamente
                        # definida
movl $27, (%ecx)        # mueve 27 a la posición de memoria cuya
                        # dirección está en %ecx
```

Movimiento de datos entre registros

Esta es la transferencia de datos que toma menor tiempo dentro del sistema, es buena práctica de programación utilizar este tipo de transferencia en vez de accesos a memoria ya que ello redunda en una mayor eficiencia.

Ejemplos:

```
movb  %ah, %bl          # mueve el contenido de %ah a %bl (1 byte)
movw  %ax, %bx          # mueve el contenido de %ax a %bx (2 bytes)
movl  %eax, %ebx        # mueve el contenido de %eax a %ebx (4 bytes)
```

Movimiento de datos entre memoria y registros

Las direcciones de memoria usualmente se expresan con etiquetas, cuando por ejemplo se escribe:

```
movl  %eax, a           # mueve el contenido de %eax a memoria
                          comenzando en la dirección a
```

Como se están transfiriendo 4 bytes éstos serán guardados en memoria de manera consecutiva a partir de la posición *a*.

Para mover la información de memoria a un registro se escribe:

```
movl  a, %eax           # mueve el contenido de la memoria desde la
                          posición a, transfiere 4 bytes a %eax
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Movimiento de datos con extensión

Hay dos instrucciones adicionales que permiten mover datos extendiéndolos.

Instrucción	Efecto	Descripción
movsbl Fuente, Destino	Destino← Fuente (signo extendido)	mueve un byte signo extendido
movzbl Fuente, Destino	Destino← Fuente (cero extendido)	mueve un byte cero extendido

Operandos Válidos		Ejemplo
registro	registro	movsbl %bx, %eax
registro	memoria	movzbl %eax, etiqueta
memoria	registro	movsbl etiqueta, %eax

La instrucción movsbl toma un operando fuente de 1 o 2 bytes, ejecuta una extensión de signo a 4 bytes y lo copia al destino de 4 bytes.

La instrucción movzbl hace un procedimiento similar pero extiende con ceros.

Ejemplo de movsbl:

```
movsbl %bl, %eax        # extiende el valor de 8 bits en %bl a 32 bits
                          usando el bit de signo
```

Ejemplo de movzbl:

```
movzbl %bl, %eax        # extiende el valor de 8 bits en %bl a 32 bits
                          rellenando con ceros
```

Carga dirección efectiva (instrucción leal)

La instrucción leal (load effective address) permite obtener la dirección de un operando en vez de su valor.

Instrucción	Efecto	Descripción
leal Fuente, Destino	Destino<- dirección de Fuente	Carga la dirección efectiva

Operandos Válidos		Ejemplo
memoria	registro	leal etiqueta, %eax

Ejemplo:

```
leal a, %eax      # actualiza el registro %eax con la dirección de a
```

Existe otra manera de realizar esta operación utilizando la instrucción movl y anteponiendo el símbolo de dólar a la etiqueta que representa la posición de memoria, esta solución se basa en el uso de direccionamiento indirecto. Para cargar %eax con la dirección de *a* se puede, entonces, escribir:

```
movl $a, %eax
```

Lo cual tendrá un resultado equivalente a la instrucción `leal` anterior.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Programa de ejemplo con varios movimientos de datos.

```
.section .data
a: .long 25                    # a=25
b: .long 42                    # b=42
c: .long 5                     # c=5

.section .text
.globl _start
_start:

      movl a, %eax             # %eax=a=25
      movl b, %ebx             # %ebx=b=42
      movl c, %ecx             # %ecx=c=5
      movl %ecx, %eax          # %eax=%ecx=5
      movl %eax, a             # a=%eax=5
      leal a, %ecx             # %ecx=dirección de a
      movl %ebx, (%ecx)        # a=%ebx=42
      movl $a, %ecx            # %ecx=dirección de a
      movl $1, %eax            # finalización del programa
      movl $0, %ebx
      int $0x80
```

Uso de la pila

La pila es un área de memoria que crece desde una dirección inicial hacia direcciones menores. El último elemento colocado en la pila es el que está disponible para ser retirado.

Las instrucciones para el manejo de la pila son dos, una para apilar un operando fuente y una para desapilar el valor que está en el tope de la pila y colocarlo en un operando destino.

Instrucción push (apilar)

Instrucción	Efecto	Descripción
pushw Fuente	R[%esp] ← R[%esp]-2 M[R[%esp]] ← Fuente	Actualiza el apuntador %esp y luego coloca Fuente en el tope de la pila (2 bytes)
pushl Fuente	R[%esp] ← R[%esp]-4 M[R[%esp]] ← Fuente	Actualiza el apuntador %esp y luego coloca Fuente en el tope de la pila (4 bytes)

Operandos Válidos	Ejemplo
inmediato	pushw $7
registro	pushl %eax
memoria	pushl etiqueta

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Instrucción pop (desapilar)

Instrucción	Efecto	Descripción
popw Destino	Destino ← M[R[%esp]] R[%esp] ← R[%esp]+2	lee el valor del tope de la pila, lo guarda en Destino y luego actualiza %esp (dato de 2 bytes)
popl Destino	Destino ← M[R[%esp]] R[%esp] ← R[%esp]+4	lee el valor del tope de la pila, lo guarda en Destino y luego actualiza %esp (dato de 4 bytes)

Operandos Válidos	Ejemplo
registro	popw %ax
memoria	popl etiqueta

Ejemplo de uso de las instrucciones pushl y popl:

Dados los valores iniciales:

%esp=0x108 %eax=15 %ebx=43 %ecx=28

La dirección se encuentra expresada en hexadecimal y los contenidos de los registros en decimal.

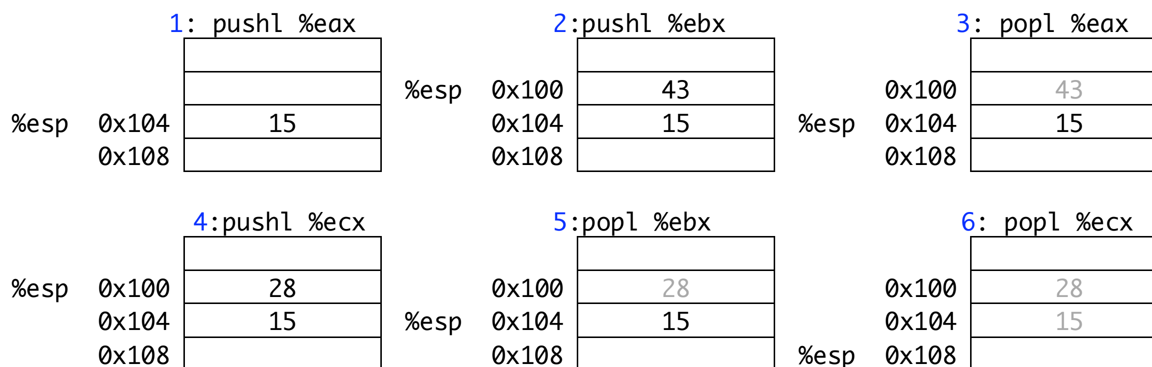
Al ejecutarse las instrucciones:

```
1       pushl %eax
2       pushl %ebx
3       popl %eax
4       pushl %ecx
5       popl %ebx
6       popl %ecx
```

Ocurre lo siguiente:

1	pushl %eax	%esp=%esp-4	%esp=0x104	M[%esp]=%eax	M[0x104]=15	
2	pushl %ebx	%esp=%esp-4	%esp=0x100	M[%esp]=%ebx	M[0x100]=43	
3	popl %eax	%eax= M[%esp]	%eax=M[0x100]	%eax=43	%esp=%esp+4	%esp=104
4	pushl %ecx	%esp=%esp-4	%esp=0x100	M[%esp]=%ecx	M[0x100]=28	
5	popl %ebx	%ebx= M[%esp]	%ebx=M[0x100]	%ebx=28	%esp=%esp+4	%esp=104
6	popl %ecx	%ecx= M[%esp]	%ecx=M[0x104]	%ecx=15	%esp=%esp+4	%esp=108

Y en la pila las direcciones y los contenidos serán:



Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

1: Al apilar el contenido de %eax se decrementa %esp en 4 bytes y se coloca el valor contenido en %eax en la pila, en este caso 15.

2: Al apilar %ebx, se decrementa %esp otros 4 bytes y se coloca el valor, 43 en la pila.

3: Cuando se lee el contenido de la pila y se coloca el valor en %eax, la máquina lee el valor en el tope de la pila, en este caso 43 y lo actualiza en el registro %eax, por lo tanto %eax ahora tendrá 43 como contenido. Luego se suman 4 bytes al apuntador %esp.

4: Al ejecutar de nuevo una instrucción de apilar, como en este caso apilar %ecx, se vuelve a decrementar %esp y se coloca el nuevo valor en el tope de la pila el cual es el número 28.

5: Con la nueva instrucción de desapilar se lee el tope de la pila, que contiene 28 y se actualiza el registro operando de la instrucción, en este caso %ebx; a partir de este momento %ebx tendrá el nuevo valor 28. Luego se incrementa %esp.

6: Con esta instrucción ocurre algo similar a la anterior, se lee el tope de la pila, el valor 15 y se actualiza en el registro operando, es decir %ecx, luego se suman 4 bytes a %esp y éste vuelve a la dirección inicial.

Instrucciones aritméticas y lógicas

Cuando se ejecutan estas instrucciones las banderas son actualizadas automáticamente.

add (suma)

Instrucción	Efecto	Descripción
addb Fuente, Destino	Destino←Destino + Fuente	suma operandos de 1 byte
addw Fuente, Destino	Destino←Destino + Fuente	suma operandos de 2 bytes
addl Fuente, Destino	Destino←Destino + Fuente	suma operandos de 4 bytes

Operandos Válidos		Ejemplo
inmediato	registro	addl $15, %eax
inmediato	memoria	addb $10, etiqueta
registro	registro	addw %bx, %ax
registro	memoria	addl %eax, etiqueta
memoria	registro	addl etiqueta, %eax

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

sub (resta)

Instrucción	Efecto	Descripción
subb Fuente, Destino	Destino←Destino - Fuente	resta operandos de 1 byte
subw Fuente, Destino	Destino←Destino - Fuente	resta operandos de 2 bytes
subl Fuente, Destino	Destino←Destino - Fuente	resta operandos de 4 bytes

Operandos Válidos		Ejemplo
inmediato	registro	subl $15, %eax
inmediato	memoria	subb $10, etiqueta
registro	registro	subw %bx, %ax
registro	memoria	subl %eax, etiqueta
memoria	registro	subl etiqueta, %eax

inc (incremento)

Instrucción	Efecto	Descripción
incb Destino	Destino←Destino + 1	incrementa operando de 1 byte
incw Destino	Destino←Destino + 1	incrementa operando de 2 bytes
incl Destino	Destino←Destino + 1	incrementa operando de 4 bytes

Operandos Válidos	Ejemplo
registro	incw %ax
memoria	incl etiqueta

dec (decremento)

Instrucción	Efecto	Descripción
decb Destino	Destino←Destino - 1	decrementa operando de 1 byte
decw Destino	Destino←Destino - 1	decrementa operando de 2 bytes
decl Destino	Destino←Destino - 1	decrementa operando de 4 bytes

Operandos Válidos	Ejemplo
registro	decw %ax
memoria	decl etiqueta

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

neg (negación aritmética)

Esta instrucción cambia el signo del operando para ello realiza el complemento a 2 del número.

Instrucción	Efecto	Descripción
negb Operando	Operando← - Operando	niega operando de 1 byte
negw Operando	Operando← - Operando	niega operando de 2 bytes
negl Operando	Operando← - Operando	niega operando de 4 bytes

Operandos Válidos	Ejemplo
registro	negl %eax
memoria	negl etiqueta

not (negación lógica)

Esta instrucción ejecuta la operación lógica NOT sobre el operando, es decir invierte los bits. Cambia los unos por ceros y los ceros por unos.

Instrucción	Efecto	Descripción
notb Operando	Operando← ~ Operando	niega operando de 1 byte
notw Operando	Operando← ~ Operando	niega operando de 2 bytes
notl Operando	Operando← ~ Operando	niega operando de 4 bytes

Operandos Válidos	Ejemplo
registro	notl %eax
memoria	notl etiqueta

and (y lógico)

Esta instrucción ejecuta un and bit a bit entre los operandos. La operación lógica and se define como:

Operando 1	Operando 2	AND
0	0	0
0	1	0
1	0	0
1	1	1

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Instrucción	Efecto	Descripción
andb Fuente, Destino	Destino←Destino AND Fuente	and entre operandos de 1 byte
andw Fuente, Destino	Destino←Destino AND Fuente	and entre operandos de 2 bytes
andl Fuente, Destino	Destino←Destino AND Fuente	and entre operandos de 4 bytes

Operandos Válidos		Ejemplo
inmediato	registro	andl $15, %eax
inmediato	memoria	andb $10, etiqueta
registro	registro	andw %bx, %ax
registro	memoria	andl %eax, etiqueta
memoria	registro	andl etiqueta, %eax

or (o lógico)

Esta instrucción realiza un or bit a bit entre los operandos. La operación lógica or se define como:

Operando 1	Operando 2	OR
0	0	0
0	1	1
1	0	1
1	1	1

Instrucción	Efecto	Descripción
orb Fuente, Destino	Destino←Destino OR Fuente	or entre operandos de 1 byte
orw Fuente, Destino	Destino←Destino OR Fuente	or entre operandos de 2 bytes
orl Fuente, Destino	Destino←Destino OR Fuente	or entre operandos de 4 bytes

Operandos Válidos		Ejemplo
inmediato	registro	orl $15, %eax
inmediato	memoria	orb $10, etiqueta
registro	registro	orw %bx, %ax
registro	memoria	orl %eax, etiqueta
memoria	registro	orl etiqueta, %eax

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

xor (or exclusivo)

Esta instrucción realiza un or exclusivo bit a bit entre los operandos. La operación lógica xor se define como:

Operando 1	Operando 2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Instrucción	Efecto	Descripción
xorb Fuente, Destino	Destino←Destino XOR Fuente	xor entre operandos de 1 byte
xorw Fuente, Destino	Destino←Destino XOR Fuente	xor entre operandos de 2 bytes
xorl Fuente, Destino	Destino←Destino XOR Fuente	xor entre operandos de 4 bytes

Operandos Válidos		Ejemplo
inmediato	registro	xorl $15, %eax
inmediato	memoria	xorb $10, etiqueta
registro	registro	xorw %bx, %ax
registro	memoria	xorl %eax, etiqueta
memoria	registro	xorl etiqueta, %eax

La instrucción xor se utiliza a menudo para inicializar registros en cero colocando el mismo registro como ambos operandos, de la siguiente manera:

```
xorl %eax, %eax         # %eax=0
```

Ya que el xor produce un cero cuando los bits son iguales, esta operación siempre producirá un registro con contenido cero.

shl (desplazamiento lógico a la izquierda)

La instrucción sal (desplazamiento aritmético a la izquierda) funciona de manera idéntica a shll. Ambas desplazan el contenido del operando a la izquierda y llenan con ceros. Esta operación produce la multiplicación por potencias de dos, el desplazamiento de una posición corresponde a la multiplicación por 2^1, dos posiciones por 2^2 y así sucesivamente. Esta instrucción tarda menos que el uso de la instrucción de multiplicación.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Instrucción	Efecto	Descripción
shlb Posiciones, Destino	Destino←Destino << posiciones	desplazamiento a la izquierda, operando de 1 byte
shlw Posiciones, Destino	Destino←Destino << posiciones	desplazamiento a la izquierda, operando de 2 bytes
shll Posiciones, Destino	Destino←Destino << posiciones	desplazamiento a la izquierda, operando de 4 bytes

Operandos Válidos		Ejemplo
inmediato	registro	shll $4, %eax
inmediato	memoria	shlb $4, etiqueta
%cl	registro	shll %cl, %eax
%cl	memoria	shll %cl, etiqueta

shr (desplazamiento lógico a la derecha)

La instrucción shr desplaza el contenido del operando a la derecha y llena con ceros. Permite dividir el número entre potencias de 2, para números sin signo.

Instrucción	Efecto	Descripción
shrb Posiciones, Destino	Destino←Destino >> posiciones	desplazamiento lógico a la derecha, operando de 1 byte
shrw Posiciones, Destino	Destino←Destino >> posiciones	desplazamiento lógico a la derecha, operando de 2 bytes
shrl Posiciones, Destino	Destino←Destino >> posiciones	desplazamiento lógico a la derecha, operando de 4 bytes

Operandos Válidos		Ejemplo
inmediato	registro	shrl $4, %eax
inmediato	memoria	shrb $4, etiqueta
%cl	registro	shrl %cl, %eax
%cl	memoria	shrl %cl, etiqueta

sar (desplazamiento aritmético a la derecha)

La instrucción sar desplaza el contenido del operando a la derecha y llena con el bit de signo. Permite dividir el número entre potencias de 2, para números con signo.

Instrucción	Efecto	Descripción
sarb Posiciones, Destino	Destino←Destino >> posiciones	desplazamiento aritmético a la derecha, operando de 1 byte
sarw Posiciones, Destino	Destino←Destino >> posiciones	desplazamiento aritmético a la derecha, operando de 2 bytes
sarl Posiciones, Destino	Destino←Destino >> posiciones	desplazamiento aritmético a la derecha, operando de 4 bytes

Operandos Válidos		Ejemplo
inmediato	registro	sarl $4, %eax
inmediato	memoria	sarb $4, etiqueta
%cl	registro	sarl %cl, %eax
%cl	memoria	sarl %cl, etiqueta

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

mul (multiplicación de enteros sin signo)

Esta instrucción opera sobre números sin signo.

Instrucción	Efecto	Descripción
mulb Fuente	%ax ← Fuente * %al	multiplicación sin signo, %al implícito
mulw Fuente	%dx:%ax ← Fuente * %ax	multiplicación sin signo, %ax implícito
mull Fuente	%edx:%eax ← Fuente * %eax	multiplicación sin signo, %eax implícito

Operandos Válidos	Ejemplo
registro	mulw %cx En este caso el resultado queda en %dx:%ax
memoria	mull etiqueta En este caso el resultado queda en %edx:%eax

imul (multiplicación de enteros con signo)

Hay tres versiones de la instrucción de multiplicación para operandos con signo, una con un operando, una con dos operandos, y otra con tres operandos.

imul con un operando

Instrucción	Efecto	Descripción
imulb Fuente	%ax ← Fuente * %al	multiplicación con signo, 1 byte. %al implícito
imulw Fuente	%dx:%ax ← Fuente * %ax	multiplicación con signo, un operando. %ax implícito
imull Fuente	%edx:%eax ← Fuente * %eax	multiplicación con signo, un operando. %eax implícito

Operandos Válidos	Ejemplo
registro	imulw %cx En este caso el resultado queda en %dx:%ax
memoria	imull etiqueta En este caso el resultado queda en %edx:%eax

imul con dos operandos

Instrucción	Efecto	Descripción
imulw Fuente, Destino	Destino ← Destino * Fuente	multiplicación con signo, dos operandos de 2 bytes
imull Fuente, Destino	Destino ← Destino * Fuente	multiplicación con signo, dos operandos de 4 bytes

Operandos Válidos		Ejemplo
registro	registro	imulw %cx, %bx
inmediato	registro	imull $4, %eax
memoria	registro	imull etiqueta, %ebx

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

imul con tres operandos

En esta instrucción el primer operando es un inmediato y el destino debe ser un registro.

Instrucción	Efecto	Descripción
imulw Multiplicador, Fuente, Destino	Destino ← Fuente * Multiplicador	multiplicación con un inmediato, 2 bytes
imull Multiplicador, Fuente, Destino	Destino ← Fuente * Multiplicador	multiplicación con un inmediato, 4 bytes

Operandos Válidos			Ejemplo
inmediato	registro	registro	imulw $12, %ax, %bx
inmediato	memoria	registro	imull $12, etiqueta, %ecx

cwd (convierte palabra en palabra doble)

Esta instrucción no tiene operandos, convierte un número con signo de una palabra a una palabra doble en el %dx:%ax duplicando el signo para generar un dividendo de 32 bits.

Instrucción	Efecto	Descripción
cwd	%dx:%ax ← %ax con signo extendido	extiende palabra a doble palabra

cltd (convierte palabra doble a palabra cuádruple)

Esta instrucción extiende un número con signo de 32 bits a uno de 64 bits duplicando el signo. Genera el dividendo %edx:%eax.

Instrucción	Efecto	Descripción
cltd	%edx:%eax ← %eax con signo extendido	extiende palabra doble a palabra cuádruple

Estas instrucciones se utilizan antes de la división para asegurar que el dividendo, el cual es implícito en la división, sea del tamaño adecuado.

div (división de enteros sin signo)

Instrucción	Efecto	Descripción
divb Fuente	%al ← %ax ÷ Fuente %ah ← resto	división sin signo, 1 byte , %ax implícito
divw Fuente	%ax ← %dx:%ax ÷ Fuente %dx ← resto	división sin signo, 2 bytes, %dx:%ax implícito
divl Fuente	%eax ← %edx:eax ÷ Fuente %edx ← resto	división sin signo, 4 bytes, %edx:%eax implícito

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Operandos Válidos	Ejemplo
registro	divw %cx En este caso el resultado queda en %ax %dx:%ax implícito
memoria	divl etiqueta En este caso el resultado queda en %eax %edx:%eax implícito

idiv (división de enteros con signo)

Instrucción	Efecto	Descripción
idivb Fuente	%al ← %ax ÷ Fuente %ah ← resto	división con signo, 1 byte, %ax implícito
idivw Fuente	%ax ← %dx:%ax ÷ Fuente %dx ← resto	división con signo, 2 bytes, %dx:%ax implícito
idivl Fuente	%eax ← %edx:eax ÷ Fuente %edx ← resto	división con signo, 4 bytes, %edx:%eax implícito

Operandos Válidos	Ejemplo
registro	idivw %cx En este caso el resultado queda en %ax %dx:%ax implícito
memoria	idivl etiqueta En este caso el resultado queda en %eax %edx:%eax implícito

Ejemplo de un programa que realiza algunas operaciones aritméticas:

```
.section .data
a: .long 5
b: .long 45
c: .long 6

.section .text
.globl _start
_start:
      movl a, %eax            # %eax=a=5
      movl b, %ebx            # %ebx=b=45
      addl %ebx, %eax         # %eax=%eax+%ebx=5+45=50
      decl %eax               # %eax=%eax-1=49
      movl c, %ecx            # %ecx=c=6
      cltd                    # extiende %eax a %edx:%eax
      idivl %ecx              # %eax=%eax/%ecx=49/6=8  %edx=resto=1
      subl %edx, %ebx         # %ebx=%ebx-%edx=45-1=44
      imull $5, %ebx          # %ebx=%ebx*5=44*5=220
      imull %ebx              # %edx:%eax=%eax*%ebx  %eax=8*220=1760
      imull $3,%eax,%ecx      # %ecx=%eax*3=1760*3=5280
      decl %eax               # %eax=%eax-1=1759
      subl $1755, %eax        # %eax=%eax-1755=4
      shll $2, %eax           # %eax=%eax<<2 = %eax*2^2 %eax=16
      movl $32, %ebx          # %ebx=32
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    sarl $4, %ebx            # %ebx=%ebx>>4 = %ebx/2^4 %ebx=2

    movl $1, %eax            # finalizacion del programa
    xorl %ebx, %ebx
    int $0x80
```

Instrucciones de comparación

cmp (compara)

Compara el contenido de dos operandos. Resta los operandos pero no actualiza el resultado, solo actualiza las banderas.

Instrucción	Efecto	Descripción
cmpb Operando2, Operando1	Operando1 - Operando2	Compara los operandos de 1 byte
cmpw Operando2, Operando1	Operando1 - Operando2	Compara los operandos de 2 bytes
cmpl Operando2, Operando1	Operando1 - Operando2	Compara los operandos de 4 bytes

Operandos Válidos		Ejemplo
inmediato	registro	cmpl $5, %eax
inmediato	memoria	cmpb $4, etiqueta
registro	registro	cmpw %bx, %ax
registro	memoria	cmpl %eax, etiqueta
memoria	registro	cmpl etiqueta, %eax

Esta instrucción compara el operando 1 con el operando 2, por ejemplo:

```
cmpl %ebx, %eax         # compara el contenido de %eax con el contenido
                        # de %ebx
```

test (examina bits)

Realiza la operación AND entre los operandos pero sin actualizar el resultado. Solo actualiza las banderas.

Instrucción	Efecto	Descripción
testb Operando2, Operando1	Operando1 AND Operando2	Examina los operandos de 1 byte
testw Operando2, Operando1	Operando1 AND Operando2	Examina los operandos de 2 bytes
testl Operando2, Operando1	Operando1 AND Operando2	Examina los operandos de 4 bytes

Operandos Válidos		Ejemplo
inmediato	registro	testl $5, %eax
inmediato	memoria	testb $4, etiqueta
registro	registro	testw %bx, %ax
registro	memoria	testl %eax, etiqueta
memoria	registro	testl etiqueta, %eax

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Es común usar la instrucción test repitiendo el operando para saber si este es cero, positivo o negativo.

Ejemplo:

```
testl  %eax, %eax         # %eax & %eax
```

Instrucciones de salto

Una instrucción de salto produce un cambio en la ejecución del programa pasando a una nueva posición, no secuencial. La dirección del salto se representa con una etiqueta.

Instrucción	Descripción
jmp etiqueta	Salto incondicional directo
jmp *etiqueta	Salto incondicional indirecto
ja etiqueta	Salto por superior (números sin signo)
jna etiqueta	Salto por no superior (números sin signo)
jae etiqueta	Salto por superior o igual (números sin signo)
jnae etiqueta	Salto por no superior o igual (números sin signo)
jb etiqueta	Salto por inferior (números sin signo)
jnb etiqueta	Salto por no inferior (números sin signo)
jbe etiqueta	Salto por inferior o igual (números sin signo)
jnbe etiqueta	Salto por no inferior o igual (números sin signo)
jc etiqueta	Salto por acarreo (números sin signo)
jnc etiqueta	Salto por no acarreo (números sin signo)
jcxz etiqueta	Salto por %cx=0
jecxz etiqueta	Salto por %ecx=0
je etiqueta	Salto por igual
jne etiqueta	Salto por no igual
jg etiqueta	Salto por mayor (>)
jng etiqueta	Salto por no mayor
jge etiqueta	Salto por mayor o igual (>=)
jnge etiqueta	Salto por no mayor o igual
jl etiqueta	Salto por menor (<)
jnl etiqueta	Salto por no menor
jle etiqueta	Salto por menor o igual (<=)
jnle etiqueta	Salto por no menor o igual
js etiqueta	Salto por negativo
jns etiqueta	Salto por no negativo
jo etiqueta	Salto por desbordamiento
jno etiqueta	Salto por no desbordamiento
jp etiqueta	Salto por paridad
jnp etiqueta	Salto por no paridad
jpe etiqueta	Salto por paridad par
jpo etiqueta	Salto por paridad impar

La instrucción jmp salta de manera incondicional, es decir, no chequea ningún código de condición. El salto puede ser directo a la dirección representada por

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

la etiqueta o indirecto donde la dirección de destino se lee de un registro o de una posición de memoria.

Ejemplo de salto incondicional directo:

```
1           addl %eax, %ebx
2           jmp  etiq1        # salto incondicional a etiq1
3           subl %edx, %ebx
4  etiq1:   addl %edx, %ebx
```

En este caso la tercera instrucción no se ejecuta ya que al ejecutarse el salto el programa pasa a ejecutar la instrucción 4.

Ejemplo de salto incondicional indirecto: dados los valores %eax = 0x120 y el contenido de la posición de memoria 0x120 = 0x180

```
jmp  *%eax        # salta a la dirección 0x120 (contenido de %eax)

jmp  *(%eax)      # salta a la dirección 0x180 (contenido de la
                    posición de memoria 0x120 direccionada por %eax)
```

Las otras instrucciones de salto son condicionales lo cual significa que la máquina revisa los códigos de condición antes de realizar el salto, si la condición se cumple realiza el salto a la etiqueta especificada, si la condición no se cumple continúa la ejecución de manera secuencial.

Ejemplo de salto condicional:

```
1           movl valor1, %eax       # %eax=valor1
2           movl valor2, %ebx       # %ebx=valor2
3           cmpl %ebx, %eax         # compara valor1 con valor2
4           jl   menor              # si valor1<valor2 salta a etiq1
5           subl $5,%eax            # si valor1>=valor2 resta 5 a %eax
6           jmp  fin                # salta a fin
7  menor:   addl $5,%eax            # si valor1<valor2 suma 5 a %eax
8  fin:
```

En el ejemplo anterior que si se cumple la condición de comparación valor1<valor2 el programa salta a la instrucción 7 sin pasar por las instrucciones 5 y 6. En caso de no cumplirse la condición, es decir valor1>=valor2, entonces continúa la ejecución en la instrucción siguiente, en este caso, la instrucción 5. Es importante destacar la necesidad de introducir la instrucción 6, un salto incondicional a la etiqueta *fin* ya que de no estar presente la máquina seguiría el orden secuencial y luego de realizar la resta

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

ejecutaría la suma lo cual arrojaría un resultado erróneo. En este programa sólo se ejecuta una de las dos operaciones aritméticas, la resta en caso de cumplirse la condición o la suma en caso de que no se cumpla.

Actualización de un registro con valor de una bandera

Es posible transferir el contenido de una bandera o una condición que dependa de la combinación de varias banderas a un registro. Esto es útil para poder usar la misma condición en varias partes del programa.

Las instrucciones que realizan esta actualización se denominan set y hay una por cada condición (tal como están expresadas en los saltos). Estas instrucciones sólo mueven un byte y el destino debe ser un registro.

Ejemplo:

```
cmpl  %ebx, %edx        # compara el contenido de %edx con %ebx
setg  %cl               # si %edx>%ebx entonces %cl se actualiza con 1,
                        # en caso contrario en 0
test %eax,%eax          # examina los bits de %eax
setz %bl                # si %eax=0 entonces actualiza %bl en 1,
                        # en caso contrario lo coloca en 0
```

Ejemplo de uso de varias instrucciones

Dado el siguiente programa:

```
.section .text
.globl _start
_start:
l1:   movl $14, %eax
l2:   movl $4, %ecx
l3:   xorl %ebx, %ebx
l4:   incl %ebx
l5:   addl %ecx, %ebx
l6:   imull %ebx, %eax
l7:   decl %ebx
l8:   addl %ebx, %ecx
l9:   cltd
l10:  idivl %ebx
l11:  movl $1, %eax
l12:  movl $0, %ebx
l13:  int $0x80
```

Se pueden observar los cambios en los registros:

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

	%eax	%ebx	%ecx	%edx
I1	14			
I2			4	
I3		0		
I4		1		
I5		5		
I6	70			
I7		4		
I8			8	
I9				0
I10	17			2
I11	1			
I12		0		

La instrucción loop

La instrucción loop permite programar ciclos de manera eficiente.

Instrucción	Descripción
loop etiqueta	Ciclo hasta que %ecx=0
loope etiqueta	Ciclo hasta que %ecx=0 o la bandera de cero (ZF)=0
loopz etiqueta	Equivalente a loope
loopne etiqueta	Ciclo hasta que %ecx=0 o la bandera de cero (ZF)=1
loopnz etiqueta	Equivalente a a loopne

La instrucción loop utiliza el registro %ecx como contador y decrementa su valor automáticamente cada vez que la instrucción se ejecuta. Sólo permite un desplazamiento de 8 bits, por lo tanto, sólo se puede usar para ciclos pequeños. Antes de usar la instrucción hay que actualizar el registro %ecx con el número de iteraciones que se necesitan en el ciclo, la instrucción lo utiliza de manera implícita, es importante recordar que este registro no se puede utilizar, entonces, dentro del ciclo ya que ello afectaría la operación del mismo. Cuando se usa esta instrucción y el registro contador %ecx llega a cero, ésto no afecta la bandera de cero (ZF).

Ejemplo del uso de la instrucción loop, programa que suma 10 veces 5.

```
.section .data
resultado:  .long 0

.section .text
.globl _start
_start:
            movl $0, %ebx           # %ebx=0, donde se va a sumar
            movl $5, %eax           # %eax=5, el valor a sumar
            movl $10, %ecx          # %ecx=10, número de iteraciones
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
ciclo1:     addl %eax, %ebx         # suma 5
            loop ciclo1             # regresa al inicio del ciclo

            movl %ebx, resultado    # actualiza resultado con el valor
                                    # final de la suma

            movl $1, %eax           # fin del programa
            xorl %ebx, %ebx
            int $0x80
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Ejercicios

5.1- Muestre los cambios ocurridos en los contenidos de los registros para el siguiente programa:

```
.section .text
.globl _start
_start:
      movl $5, %eax
      xorl %ecx, %ecx
      addl $6, %ecx
      addl %ecx, %eax
      movl %ecx, %ebx
      movl $2, %edx
      incl  %ebx
      imull %edx, %ebx
      shll $2,%edx
      sarl $1,%ecx
      addl %ebx, %eax
      cltd
      idivl %ecx
      movl $1, %eax
      movl $0, %ebx
      int $0x80
```

5.2- Para los valores a=6, b=8, c=12, d=4 declarados en la sección de datos, escriba un programa que calcule x=a-3b+25(c-d).

5.3- Escriba un fragmento de código que calcule x=16*y sin utilizar la instrucción de multiplicación.

5.4- Escriba un fragmento de código que calcule x=y/32 sin usar la instrucción de división.

5.5- Dados los valores a=4, b=5, c=2, d=7 declarados en la sección de datos; escriba un programa que calcule la expresión x=(a+(b-c))/(c*d).

5.6- Escriba un programa que dados tres valores enteros declarados en la sección de datos, determine el mayor y lo guarde en una variable denominada *mayor*.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

5.7- Escriba un programa que dados dos valores enteros a y b declarados en la sección de datos produzca como salida la actualización de la variable resultado según las siguientes condiciones:

- Si a=b: resultado=a
- Si a>b: resultado=a-b
- Si a<b: resultado=b-a

5.8- Suponiendo que el contenido inicial de %ebp=%esp=0x100 y dadas las siguientes instrucciones, muestre los cambios ocurridos en la pila y en los registros involucrados.

```
movl $25, %eax
addl $5, %eax
pushl %eax
incl %eax
pushl %eax
movl %eax, %ebx
subl $15, %ebx
cltd
idivl %ebx
pushl %edx
popl %eax
popl %ecx
popl %ebx
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Capítulo 6. Entrada / Salida

Un ser humano puede tener las ideas más innovadoras o los mejores sentimientos, sin embargo esto de nada vale si los mismos no pueden ser comunicados a los demás. De la misma forma, un programa de computadora no tiene ninguna utilidad si el mismo no puede de una u otra forma comunicarse con el usuario.

El conjunto de actividades que permiten a un programa interactuar con los usuarios (y el resto de su entorno, como por ejemplo otras computadoras), así como los mecanismos de hardware y software que soportan estas actividades son comúnmente conocidos como sistema de entrada/salida, frecuentemente abreviado E/S.

Aunque en los inicios de la computación moderna el llevar a cabo este tipo de actividades era una tarea bastante compleja y limitada, con la aparición y evolución de los lenguajes de alto nivel esto ha cambiado drásticamente, al punto en que hoy en día basta una simple línea de código para comunicarle un mensaje al usuario o recibir una indicación por parte de este.

A lo largo de este capítulo serán descritas diferentes formas de realizar las actividades de E/S. Se comenzará con el enfoque más sencillo, provisto por un lenguaje de alto nivel como C, para luego abordar enfoques más avanzados como las llamadas al sistema o la utilización de instrucciones bajo nivel diseñadas para este propósito.

Funciones de alto nivel

Actualmente todos los lenguajes de alto nivel proporcionan alguna instrucción que de una u otra forma permite interactuar con los usuarios. En el caso del lenguaje C, las instrucciones básicas de entrada salida son printf y scanf, las cuales permiten respectivamente enviar (print, imprimir) o recibir (scan, leer) información. La 'f' con que finalizan ambas instrucciones hace referencia a que las mismas proporcionan E/S formateada. Más adelante se explicará en que consiste esto.

El sistema de E/S del lenguaje C define los conceptos de flujo y archivo. Un archivo representa a cualquier dispositivo de E/S como puede ser la pantalla o el teclado, y permite que el programador realice operaciones de E/S sobre los mismos sin necesidad de conocer los detalles especificos de un determinado dispositivo.

Sin embargo considere que el lenguaje C permite la utilización de una amplía variedad de dispositivos diferentes. Esto quiere decir que no todos los archivos soportan las mismas operaciones. Para solucionar esto, y facilitar las tareas del programador se define un mecanismo que permite tratar a todos los dispositivos de una manera estándar. Para ello, el sistema de E/S de C trata cada dispositivo físico como un flujo, lo cual no es más que una abstracción lógica de dicho dispositivo.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Los conceptos de archivo y flujo están relacionados de la siguiente manera: Un archivo es asociado a un flujo determinado mediante una operación denominada apertura. A partir de este momento, el programador envía o recibe datos hacia o desde el flujo (que puede ser binario o de texto), y es el subsistema de E/S quien realiza la correspondencia necesaria entre dicho flujo y su archivo asociado, de forma que los datos sean efectivamente intercambiados con el dispositivo físico que dicho archivo representa. Finalmente es posible desasociar el flujo del archivo mediante una operación de cierre.

En un programa que no involucre operaciones avanzadas de E/S, (como la mayor parte de los presentados en esta obra) el programador no necesita realizar operaciones de apertura o cierre, ya que al momento de comenzar la ejecución del programa existen ciertos flujos asociados automáticamente a los dispositivos de E/S utilizados con mayor frecuencia. El estándar ANSI de C define tres flujos predefinidos:

Flujo	Descripción	Dispositivo asociado por defecto
stdin	Entrada estándar	Teclado
stdout	Salida estándar	Monitor
stderr	Flujo de errores	Monitor

A continuación se describirán formalmente las funciones printf y scanf, para luego pasar a ilustrar su utilización mediante algunos ejemplos.

Función printf

La función *printf* permite escribir en la salida estándar, proporcionando un formato específico para los datos a escribir. De allí que las escrituras realizadas por esta función y las lecturas realizadas por scanf sean conocidas como e/s formateada. Esto implica que ambas funciones pueden trabajar con los tipos de datos especificados por el estandar ANSI de C tales como números, caracteres y cadenas.

Esta función recibe una cantidad variable de argumentos, en los que solo uno de ellos es obligatorio: la cadena de formato. Este parámetro es una cadena que permite indicar el formato que tendrá la impresión que se quiere realizar, y en base a la cual se determina la cantidad y tipo de los parámetros restantes.

Dicha cadena puede contenter caracteres comunes y corrientes, los cuales se mostrarán en pantalla exactamente como se encuentran en la cadena, y un segundo tipo de caracteres conocidos como *especificadores de formato*, los cuales están formados por un símbolo de porcentaje (%) y un código de formato (‘c’ para caracteres, ‘d’ para números decimales, etc). Los especificadores de formato compatibles con la función *printf* son los siguientes:

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Especificador	Formato
%c	Caracter simple
%d	Entero con signo en base decimal
%i	Entero con signo en base decimal (*integer*)
%e	Notación científica (letra 'e' minúscula)
%E	Real en notación científica (letra 'E' mayúscula)
%f	Real (punto flotante)
%g	Equivalente a la representación real más corta (%e o %f)
%G	Equivalente a la representación real más corta (%E o %f)
%o	Entero con signo en base octal
%p	Apuntador o dirección de memoria
%s	Cadena de caracteres
%u	Entero sin signo en base decimal
%x	Entero con signo en base hexadecimal (letras minúsculas)
%X	Entero con signo en base hexadecimal (letras mayúsculas)

Además de la cadena de formato, deben existir tantos parámetros adicionales como especificadores de formato contenga la cadena, ya que cada uno de los mismos proporciona el valor por el cual será sustituido cada especificador al momento de mostrar la cadena por la pantalla. Dicho de otra manera, la cadena de formato puede ser vista como una plantilla, mientras que el resto de los parámetros son los valores que completan los campos vacíos de dicha plantilla. De esta forma, las siguientes llamadas a *printf* son válidas (asumiendo que las variables sexo y edad han sido declaradas previamente y son de tipo caracter y entero respectivamente):

```
printf("Hola mundo");
printf("Su edad es %d", edad);
printf("Sexo:%c - Edad:%d", sexo, edad);
```

Por ejemplo, si la variable 'edad' (de tipo entero) tiene un valor decimal de veinte, la ejecución de la segunda llamada a *printf* ocasionará que se muestre el mensaje "Su edad es 20" a través de la pantalla.

Función scanf

La función *scanf* permite leer desde la entrada estándar, proporcionando un formato específico para los datos a recibir. De forma análoga a la función *printf*, esta función permite reconocer los tipos de datos especificados por el estándar ANSI de C tales como números, caracteres y cadenas.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Los argumentos recibidos por la función scanf son similares a los de la función *printf*, siendo de nuevo el primero de ellos la cadena de formato. Sin embargo en este caso la cadena de formato es utilizada para especificar una 'plantilla' que deberá ser completada con la entrada proporcionada por el usuario.

En este caso, por cada especificador de formato que se encuentre en la cadena será necesario proporcionar un apuntador a la dirección de memoria en la que será almacenado el elemento reconocido. Los especificadores de formato compatibles con la función *scanf* son los siguientes:

Especificador	Formato
%c	Caracter simple
%d	Entero con signo en base decimal
%i	Entero con signo en base decimal (*integer*)
%e	Notación científica (letra 'e' minúscula)
%E	Real en notación científica (letra 'E' mayúscula)
%f	Real (punto flotante)
%g	Equivalente a la representación real que coincida (%e o %f)
%G	Equivalente a la representación real que coincida (%E o %f)
%o	Entero con signo en base octal
%p	Apuntador o dirección de memoria
%s	Cadena de caracteres
%u	Entero sin signo en base decimal
%x	Entero con signo en base hexadecimal (letras minúsculas)
%X	Entero con signo en base hexadecimal (letras mayúsculas)

De esta forma, las siguientes llamadas a *scanf* son válidas (asumiendo que la variable edad ha sido declarada como entero, y las variables nombre y apellido como arreglo de caracteres):

```
scanf("%d", &edad);
scanf("%s,%s", nombre, apellido);
```

Por ejemplo, la ejecución de la primera llamada a scanf ocasionará que se detenga la ejecución del programa hasta que el usuario proporcione un número entero decimal mediante la utilización del teclado. Si el usuario introduce, digamos, el valor 20, el mismo será asignado a la variable edad, o en otras palabras, dicho valor será almacenado en la dirección de memoria ocupada por dicha variable (&edad).

Es válido preguntarse que ocurre si la entrada del usuario no coincide al 100% con la especificada por la cadena de formato al hacer una llamada a la función *scanf*. De suceder esto, solo se reconoce la entrada hasta el primer caracter que

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

no coincida con la cadena de formato, y solo se le asigna un valor a las variables correspondientes a los especificadotes de formato que preceden a dicho carácter. En cualquier caso la función retorna la cantidad de elementos de la entrada que fueron asignados a las variables. Por ejemplo, si se hace la siguiente llamada a *scanf*:

```
scanf("%s, %s. V-%d", apellido, nombre, &cedula);
```

Y el usuario introduce la cadena “Perez, Pedro. 16247521”, la función *scanf* devolverá el valor 2, ya que la cadena coincide con la plantilla solo hasta el espacio que se encuentra después del punto. Al no encontrar la letra ‘V’, la función no continúa examinando la cadena de entrada, y retorna el valor dos ya que solo las variables apellido y nombre recibieron un valor (Perez y Pedro, respectivamente). La variable cédula no es modificada.

Consideraciones adicionales

Una diferencia importante entre las funciones *printf* y *scanf* consiste en la manera en que se les pasan los argumentos. Todos los argumentos de una llamada a la función *scanf* (a excepción de la cadena de formato) son pasados *por referencia*, es decir, se proporciona la dirección de la variable y no su valor. Esto se debe a que la función utiliza estas variables para almacenar un valor leido de la entrada estándar y en consecuencia necesita saber la posición de memoria en la que el mismo será almacenado.

En contraparte, la función *printf* no necesita conocer la dirección de las variables, sino el valor que contienen para poder escribirlo en la salida estándar. Por esta razón, todos los tipos de dato simple (enteros, caracteres, etc.) son pasados *por valor*. Algunas otras variables como los arreglos y las estructuras (esto incluye las cadenas ya que son arreglos de caracteres) son pasados por referencia, pero esto se debe a limitaciones del pase de parámetros por valor (ya que los parámetros se almacenan en la pila, véase el capítulo de procedimientos) y no a que la función printf necesite conocer la dirección de dichos argumentos per se.

Para ilustrar la utilización de ambas funciones desde un programa en lenguaje ensamblador, se presentan dos ejemplos sencillos. Sin embargo es importante hacer una acotación en este punto. Cuando se hace una llamada a las funciones printf y scanf (o a cualquier otro procedimiento), el contenido de los registros %eax, %ecx y %edx corre el riesgo de ser modificado. El motivo de esto será explicado en profundidad en el capitulo relativo a la llamada a procedimientos, sin embargo, es importante tener en cuenta este hecho al momento de programar, tomando las precauciones necesarias para no perder información importante al llevar a cabo las operaciones de lectura o escritura.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

El primer ejemplo consiste en un programa que pide al usuario una cadena de caracteres mediante la utilización de scanf, y luego la muestra de nuevo mediante la utilización de printf.

Programa en C:

```
#include <stdio.h>

char cadena[80];
main()
 {
  printf("Por favor introduzca una cadena: ");
  scanf("%s", cadena);
  printf("La cadena introducida es: %s\n", cadena);
}
```

Programa en ensamblador:

```
.section .rodata
cad1:   .asciz "Por favor introduzca una cadena: "
cad2:   .asciz "%s"
cad3:   .asciz "La cadena introducida es: %s\n"

.section .bss
cadena: .space 50   # Reserva espacio para la cadena introducida

.section .text
.globl _start
_start:

    # Muestra un mensaje solicitando que se escriba una cadena
    leal cad1, %eax     # Carga la dirección de la cadena de formato
    pushl %eax          # Pasa el primer argumento (cad1)
    call printf         # Hace la llamada a printf
    addl $4, %esp       # Libera el espacio de la pila ocupado por
                          el único argumento

    # Lee la cadena introducida por el usuario
    leal cadena, %eax   # Carga la dirección donde se almacenará
                          la cadena
    pushl %eax          # Pasa el segundo argumento (cadena)
    leal cad2, %eax     # Carga la dirección de la cadena de formato
    pushl %eax          # Pasa el primer argumento (cad2)
    call scanf          # Hace la llamada a scanf
    addl $8, %esp       # Libera los 8 bytes de pila ocupados por
                          los argumentos
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    # Muestra el resultado
    leal cadena, %eax    # Carga la dirección de la cadena leida
    pushl %eax           # Pasa el segundo argumento (cadena)
    leal cad3, %eax      # Carga la dirección de la cadena de formato
    pushl %eax           # Pasa el primer argumento (cad3)
    call printf          # Hace la llamada a printf
    addl $8, %esp        # Libera los 8 bytes de pila ocupados por
                           los argumentos

    # Termina el programa
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

En este segundo ejemplo se pedirán al usuario dos números enteros en notación decimal, y se mostrará la suma de ambos. Si bien el programa puede parecer extremadamente sencillo, más adelante observaremos que esto se debe a las facilidades provistas por las funciones *printf* y *scanf*.

Programa en C:

```
#include <stdio.h>

int A,B,C;
main()
 {
  printf("Valor de A: ");
  scanf("%d", &A);
  printf("Valor de B: ");
  scanf("%d", &B);
  C = A+B;
  printf("El valor de A+B es: %d\n", C);
}
```

Programa en ensamblador:

```
.section .rodata
cad1:   .asciz "Valor de A: "
cad2:   .asciz "Valor de B: "
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
cad3:   .asciz "El valor de A+B es: %d\n"
cad4:   .asciz "%d"

.section .bss
A:  .space 4            # Reserva espacio para un entero
B:  .space 4            # Reserva espacio para un entero
C:  .space 4            # Reserva espacio para un entero

.section .text
.globl _start
_start:

    # Muestra un mensaje solicitando el valor de A
    leal cad1, %eax     # Carga la dirección de la cadena de formato
    pushl %eax          # Pasamel primer argumento (cad1)
    call printf         # Hace la llamada a printf
    addl $4, %esp       # Libera los 4 bytes de pila ocupados por el
                          único argumento

    # Lee el valor de A
    leal A, %eax        # Carga la dirección donde se almacenará el
                          entero
    pushl %eax          # Pasa el segundo argumento (&A)
    leal cad4, %eax     # Carga la dirección de la cadena de formato
    pushl %eax          # Pasa el primer argumento (cad4)
    call scanf          # Hace la llamada a scanf
    addl $8, %esp       # Libera los 8 bytes de pila ocupados los
                          argumentos

    # Muestra un mensaje solicitando el valor de B
    leal cad2, %eax     # Carga la dirección de la cadena de formato
    pushl %eax          # Pasa el primer argumento (cad2)
    call printf         # Hace la llamada a printf
    addl $4, %esp       # Libera los 4 bytes de pila ocupados por el
                          único argumento


    # Lee el valor de B
    leal B, %eax        # Carga la dirección donde se almacenará el
                          entero
    pushl %eax          # Pasam el segundo argumento (&B)
    leal cad4, %eax     # Carga la dirección de la cadena de formato
    pushl %eax          # Pasa el primer argumento (cad4)
    call scanf          # Hace la llamada a scanf
    addl $8, %esp       # Libera los 8 bytes de pila ocupados los
                          argumentos
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    # Suma los números y deja el resultado en C
    movl A, %eax
    addl B, %eax
    movl %eax, C

    # Muestra el resultado
    pushl C             # Pasamos el segundo argumento (C)
    leal cad3, %eax     # Cargamos la dirección de la cadena de formato
    pushl %eax          # Pasamos el primer argumento (cad3)
    call printf         # Hacemos la llamada a printf
    addl $8, %esp       # Liberamos los 8 bytes de pila ocupados los
                          argumentos

    # Termina el programa
    movl $1, %eax
    movl $0, %ebx
    int $0x80.rodata
```

Llamadas al sistema

Las llamadas al sistema son servicios que provee el sistema operativo a los programas de usuario para llevar a cabo diversas tareas de manera controlada y coordinada. En Linux, la lista de las distintas llamadas al sistema se puede encontrar en el archivo "*unistd.h*", generalmente ubicado en el directorio "*/usr/include/asm/*". De esta forma, es posible conocer el número que identifica a cada servicio mediante el siguiente comando:

```
linux> more /usr/include/asm/unistd.h
....
#define __NR_restart_syscall     0
#define __NR_exit                1
#define __NR_fork                2
#define __NR_read                3
#define __NR_write               4
#define __NR_open                5
#define __NR_close               6
....
```

Para obtener información referente a un servicio específico, es posible consultar la segunda sección del manual (*man*), la cual contiene la información referente a las llamadas al sistema. Por ejemplo, para conocer más acerca del servicio de escritura (write, servicio #4), es necesario utilizar el siguiente comando:

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

linux> man 2 write

Son de particular interés los servicios *read* (lectura) y *write* (escritura), los cuales permiten realizar las mismas operaciones básicas que las instrucciones de alto nivel *printf* y *scanf*, pero a un nivel más elemental. Si bien las llamadas al sistema no son invocadas como funciones de C, a menudo la descripción de las mismas se da en la notación de dicho lenguaje.

Servicio read

El prototipo del servicio *read* es el siguiente:

```
ssize_t read(int fd, void *buf, size_t count);

fd:     Descriptor de archivo desde el que se lee (stdin, stdout, sterr)
buf:    Apuntador al área de memoria donde se almacenarán los datos.
count:  Número máximo de bytes a leer.
```

Servicio write

De forma similar, el prototipo del servicio *write* es el siguiente:

```
ssize_t write(int fd, const void *buf, size_t num);

fd:     Descriptor de archivo en el que se escribe (stdin, stdout, sterr)
buf:    Apuntador al área de memoria que contiene los datos a escribir.
num:    Número máximo de bytes a escribir.
```

Ambas llamadas al sistema serán tratadas con un mayor grado de detalle en el capitulo 12, junto con otras llamadas que permiten el manejo de archivos.

Ahora bien, una vez identificados los servicios de interés, y haber consultado el manual para identificar los parámetros que reciben o los resultados que devuelven, surge la pregunta de cómo hacer uso de los mismos desde un programa en lenguaje ensamblador.

La petición de un servicio al sistema operativo se hace a través de la interrupción 0x80 (esta interrupción es similar a la 23H de DOS). Simplemente se coloca el número que identifica al servicio en el registro %eax, y se levanta la interrupción mediante la instrucción “int $0x80”.

Sin embargo, aún cabe preguntarse ¿Cómo se pasan los argumentos de la llamada al sistema para que la misma se ejecute de manera exitosa? La respuesta es sencilla. Si la llamada a un servicio recibe menos de seis argumentos, los mismos deben ser depositados en los registros %ebx, %ecx, %edx, %esi y %edi, en este mismo orden. Por ejemplo, para el caso de los

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

servicios *read* y *write*, es necesario colocar el descriptor de archivo desde el cual se va a leer o al cual se va a escribir (*fd*) en el registro %ebx.

En caso de existir seis o más argumentos, todos deben ser almacenados en un área de memoria contigua, y debe colocarse en %ebx un apuntador al primero de los mismos.

Para ilustrar la utilización de las llamadas al sistema *read* y *write*, se presentan de nuevo los mismos dos ejemplos que fueron implementados con las funciones *printf* y *scanf* en la sección anterior. El primero de ellos es bastante similar a su contraparte con funciones de alto nivel, y esto se debe a que el mismo solo lee e imprime cadenas y no otro tipo de datos. Sin embargo, note que debido a la ausencia de un mecanismo como el proporcionado por el especificador de formato "%s", es necesario mostrar el resultado final con dos llamadas al sistema independientes.

Programa en ensamblador:

```
.section .rodata
cad1:   .asciz "Por favor introduzca una cadena: "
cad2:   .asciz "La cadena introducida es: "
fcad2:

.section .bss
cadena: .space 50   # Reserva espacio para la cadena introducida


.section .text
.globl _start
_start:

    # Muestra un mensaje solicitando que se escriba una cadena
    movl $4, %eax           # Servicio #4 (write)
    movl $1, %ebx           # File Descriptor = 1 (stdout)
    leal cad1, %ecx         # Apuntador a la cadena a imprimir
    movl $cad2-cad1, %edx   # Número máximo de caracteres a imprimir
    int $0x80               # Llamada al SO

    # Lee la cadena introducida por el usuario
    movl $3, %eax           # Servicio #3, (servicio de lectura)
    movl $0, %ebx           # File Descriptor = 0 (stdin)
    leal cadena, %ecx       # Dirección donde se almacenará la cadena
    movl $49, %edx          # Número máximo de caracteres a leer
    int $0x80               # Llamada al SO

    # Muestra el resultado (parte 1)
    movl $4, %eax           # Servicio #4 (write)
    movl $1, %ebx           # File Descriptor = 1 (stdout)
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    leal cad2, %ecx         # Apuntador a la cadena a imprimir
    movl $fcad2-cad2, %edx  # Número máximo de caracteres a imprimir
    int $0x80               # Llamada al SO

    # Muestra el resultado (parte 2)
    movl $4, %eax           # Servicio #4 (write)
    movl $1, %ebx           # File Descriptor = 1 (stdout)
    leal cadena, %ecx       # Apuntador a la cadena a imprimir
    movl $49, %edx          # Número máximo de caracteres a imprimir
    int $0x80               # Llamada al SO

    # Termina el programa
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

En el ejemplo anterior se observa que a excepción de la llamada adicional, el programa es bastante similar a la versión que utiliza las funciones *printf* y *scanf*. Sin embargo, repetir el segundo ejemplo utilizando llamadas al sistema no es una tarea tan sencilla, ya que los servicios utilizados permiten la lectura y escritura de cadenas de caracteres, mientras que en este ejemplo se necesita leer y escribir otro tipo de dato, como lo son enteros en base decimal.

Para hacer una versión del mismo programa que utilice llamadas al sistema será necesario llevar a cabo la conversión entre un entero decimal y la cadena de caracteres que lo representa de manera manual, es decir, escribiendo código adicional que realice dicha tarea. A continuación se presenta la solución:

Programa en ensamblador:

```
.section .rodata
cad1:   .asciz "Valor de A: "
cad2:   .asciz "Valor de B: "
cad3:   .asciz "El valor de A+B es: "
fcad3:

.section .bss
A:  .space 4        # Reserva espacio para un entero
B:  .space 4        # Reserva espacio para un entero
C:  .space 4        # Reserva espacio para un entero
cadena: .space 11   # Reserva espacio para la cadena introducida


.section .text
.globl _start
_start:
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    # Muestra un mensaje solicitando que proporcione el valor de A
    movl $4, %eax           # Servicio #4, (write)
    movl $1, %ebx           # File Descriptor = 1 (stdout)
    leal cad1, %ecx         # Apuntador a la cadena a imprimir
    movl $cad2-cad1, %edx   # Número máximo de caracteres a imprimir
    int $0x80               # Llamada al SO

    # Lee la cadena introducida por el usuario
    movl $3, %eax           # Servicio #3, (read)
    movl $0, %ebx           # File Descriptor = 0 (stdin)
    leal cadena, %ecx       # Dirección donde se almacenará la cadena
    movl $49, %edx          # Número máximo de caracteres a leer
    int $0x80               # Llamada al SO

    # Obtiene el valor entero representado por la cadena
    xorl %ecx, %ecx         # i=0
    xorl %eax, %eax         # A=0
    movl $10, %esi          # base decimal
for1:
    movb cadena(%ecx), %bl  # Carga un caracter. (cadena[i])
    cmpb $0, %bl            # Si llega al final de la cadena
    je ffor1                # Sale del ciclo
    cmpb $10, %bl           # Si llega al final de la cadena
    je ffor1                # Sale del ciclo
    subb $'0', %bl          # Obtiene el dígito entero que representa
    movzx %bl, %ebx         # Expande el entero a 32 bits (D)
    mull %esi               # A=A*10
    addl %ebx, %eax         # A=A+D
    incl %ecx               # Se repite para el siguiente caracter
    jmp for1
ffor1:
    movl %eax, A            # Guarda el valor de A

    # Muestra un mensaje solicitando que proporcione el valor de B
    movl $4, %eax           # Servicio #4, (write)
    movl $1, %ebx           # File Descriptor = 1 (stdout)
    leal cad2, %ecx         # Apuntador a la cadena a imprimir
    movl $cad3-cad2, %edx   # Número máximo de caracteres a imprimir
    int $0x80               # Llamada al SO

    # Lee la cadena introducida por el usuario
    movl $3, %eax           # Servicio #3, (read)
    movl $0, %ebx           # File Descriptor = 0 (stdin)
    leal cadena, %ecx       # Dirección donde se almacenará la cadena
    movl $49, %edx          # Número máximo de caracteres a leer
    int $0x80               # Llamada al SO
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    # Obtiene el valor entero representado por la cadena
    xorl %ecx, %ecx         # i=0
    xorl %eax, %eax         # B=0
    movl $10, %esi          # base decimal

for2:
    movb cadena(%ecx), %bl  # Carga un caracter. (cadena[i])
    cmpb $0, %bl            # Si llega al final de la cadena
    je ffor2                # Sale del ciclo
    cmpb $10, %bl           # Si llega al final de la cadena
    je ffor2                # Sale del ciclo
    subb $'0', %bl          # Obtiene el dígito entero que representa
    movzx %bl, %ebx         # Expande el entero a 32 bits (D)
    mull %esi               # B=B*10
    addl %ebx, %eax         # B=B+D
    incl %ecx               # Se repite para el siguiente caracter
    jmp for2
ffor2:
    movl %eax, B            # Guarda el valor de B

    # Calcula el valor de A+B
    movl A, %eax
    addl B, %eax
    movl %eax, C

    # Obtiene la cadena que representa al entero C
    movb $0, cadena+10      # Termina la cadena ('\0')
    movl $9, %ecx           # i=9
    movl $10, %esi          # base decimal
for3:
    cltd                    # Expande C a 64 bits (edx:eax)
    idivl %esi              # Lo divide entre la base
    addb $'0', %dl          # Obtiene la representación del dígito
                              decimal
    movb %dl, cadena(%ecx)  # Lo inserta en la cadena
    decl %ecx
    cmpl $0, %eax           # Se repite mientras queden dígitos
    jne for3
    movl %ecx, %esi         # Guarda el desplazamiento

    # Muestra el resultado (parte 1)
    movl $4, %eax           # Servicio #4, (write)
    movl $1, %ebx           # File Descriptor = 1 (stdout)
    leal cad3, %ecx         # Apuntador a la cadena a imprimir
    movl $fcad3-cad3, %edx  # Número máximo de caracteres a imprimir
    int $0x80               # Llamada al SO
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    # Muestra el resultado (parte 2)
    movl $4, %eax           # Servicio #4, (write)
    movl $1, %ebx           # File Descriptor = 1 (stdout)
    leal cadena, %ecx       # Dirección donde se almacenará la cadena
    addl %esi, %ecx         # Ajustamos la dirección de la cadena
    movl $10, %edx          # Número máximo de caracteres a imprimir
    int $0x80               # Llamada al SO

    # Termina el programa
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

Observe que la cantidad de líneas de código se incrementó de manera importante. Si consideramos además que este programa no contempla la utilización de números negativos, o la posibilidad de que el usuario se equivoque e introduzca un carácter distinto a uno de los diez dígitos decimales, se hace evidente lo útil que resulta el contar con funciones como *printf* y *scanf*. Sin embargo, es importante tener siempre en cuenta que dichas funciones de una u otra forma llevan a cabo un procedimiento similar al que hemos realizado aquí manualmente: una función de alto nivel no realiza ninguna tarea que no pueda ser llevada a cabo mediante un conjunto de instrucciones de bajo nivel, ya que en el fondo son estas últimas las únicas que el procesador de una computadora es capaz de ejecutar.

Instrucciones de bajo nivel

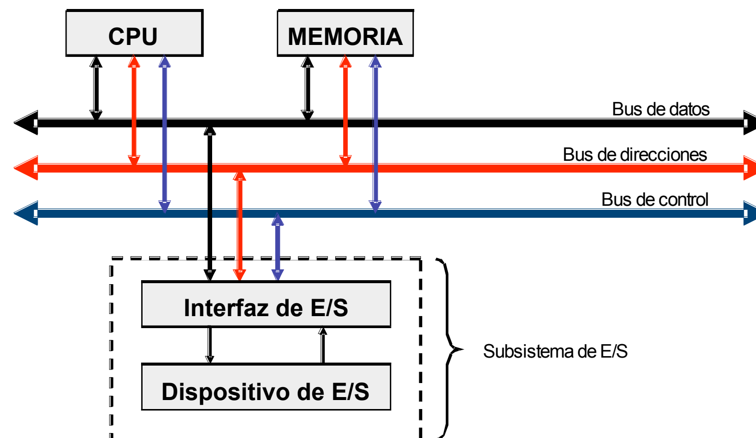
La implementación de las funciones de alto nivel y las llamadas al sistema que han sido descritas anteriormente es posible, gracias a la existencia de funciones de bajo nivel (instrucciones de máquina) que permiten interactuar con los diferentes dispositivos de E/S.

En la arquitectura IA-32 éstas son la instrucción “in”, que permite recibir datos desde el dispositivo, y la instrucción “out”, que permite enviar datos hacia el mismo. Antes de describir en mayor profundidad el funcionamiento de ambas, es necesario conocer el funcionamiento básico de un subsistema de E/S moderno.

El subsistema de E/S

El subsistema de E/S de una computadora proporciona un modo de comunicación entre el sistema central y el ambiente externo. Los dispositivos de E/S conectados a la computadora también se llaman periféricos. Un subsistema de E/S consiste en interfases de E/S y dispositivos periféricos. La estructura general de un susbsistema de E/S es la siguiente:

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.



Interfaz de E/S:

La interfaz de E/S proporciona un método para transferir información entre dispositivos de almacenamiento interno y dispositivos externos. El propósito del enlace de comunicación provisto por la interfaz es resolver las diferencias que existen entre la computadora central y cada periférico. Estas diferencias son:

- Los periféricos son dispositivos electromecánicos y electromagnéticos, por lo tanto se necesita una conversión de valores de señales.
- La velocidad de transferencia de datos es diferente para cada dispositivo y a su vez es diferente con respecto a la velocidad de transferencia de la CPU. Por ello se necesita un mecanismo de sincronización.
- Los códigos de datos y los formatos en los periféricos son diferentes del formato de palabra del sistema central.
- Los modos de operación de cada dispositivo son diferentes entre sí.

Funciones de una Interfaz de E/S:

- Control y Temporización: para coordinar el tráfico entre los recursos internos y los dispositivos externos. Además se realiza arbitraje de buses comunes.
- Comunicación con la CPU: Decodificación de órdenes (ejemplo READ SECTOR), Intercambio de Datos, Información de estado (ejemplo BUSY, READY), reconocimiento de dirección.
- Comunicación con los dispositivos: implica intercambiar órdenes, información de estado y datos.
- Almacenamiento temporal de los datos.
- Detección de errores: por defectos mecánicos y eléctricos, por ejemplo

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

papel atascado en una impresora. También errores en la información almacenada o transmitida, lo cual se logra por ejemplo mediante el uso de un bit de paridad.

Técnicas de transferencia de E/S:

Los subsistemas de E/S suelen clasificarse en base a qué tanto se involucra la CPU en la transacción de E/S. Se entiende por transacción el transferir un único bit, byte, palabra, o bloque de bytes de información entre el dispositivo de E/S y la CPU, o entre el dispositivo de E/S y la memoria principal.

Existen diversas técnicas para el manejo de esta transferencia de datos entre el computador (CPU o memoria) y los dispositivos externos. Algunas usan la CPU como una trayectoria intermedia; otras transfieren los datos directamente desde o hacia la unidad de memoria. Desde este punto de vista, la transferencia de datos puede ser:

- Programada
- Basada en interrupciones
- Por acceso directo a memoria (DMA, Direct Memory Access)
- Con un procesador de E/S (IOP, Input/Output Processor)

A continuación se describe brevemente cada una de estas técnicas.

Entrada/Salida programada:

En este caso los datos se intercambian entre la CPU y la interfaz de E/S. La CPU ejecuta un programa que le da el control directo de la operación de entrada o salida, donde se incluye la detección del estado del dispositivo, el envío de un comando de lectura o escritura y la transferencia de datos.

Transferir datos bajo el control del programa requiere que la CPU realice un monitoreo constante de periféricos. Cuando la CPU envía una orden a la interfaz de E/S, debe esperar hasta que la operación concluya. Una vez que se inicia una transferencia de datos, es necesario que la CPU monitoree la interfaz para detectar cuando puede volverse a realizar una transferencia, ya que la interfaz no realiza ninguna acción para notificar a la CPU.

Este proceso toma mucho tiempo y mantiene ocupado el procesador mientras espera por la disponibilidad del dispositivo. Además, si la CPU es más rápida que la interfaz de E/S éste es un tiempo que la CPU desperdicia. Si bien es un método sumamente ineficiente, tiene la ventaja de ser el que requiriere la menor cantidad de hardware para ser implementado.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Entrada/Salida basada en interrupciones:

Esta técnica permite que la CPU emita un comando de E/S hacia una interfaz y luego ejecute otro trabajo. La interfaz se encargará de producir una interrupción para solicitar atención de la CPU cuando esté lista para intercambiar datos. Cuando la CPU detecta la señal de interrupción detiene la tarea que está procesando, transfiere el control a una rutina de servicio (conocido como manejador de interrupción) para procesar la transferencia de entrada o salida y luego regresa a la tarea que ejecutaba originalmente.

Tanto en la E/S programada como en la de interrupciones, la CPU es la responsable de extraer los datos de la memoria principal en una salida y de almacenar los datos en la memoria principal en una entrada. Si bien esta técnica es más eficiente que la E/S programada, ya que permite que el CPU atienda otras tareas mientras se espera por el dispositivo periférico, la misma requiere de la existencia de un mecanismo de interrupciones lo que incrementa los requerimientos de hardware.

Acceso directo a memoria (DMA):

La técnica de acceso directo a memoria permite mover grandes volúmenes de datos sin involucrar directamente la CPU. Necesita un componente adicional: un controlador de DMA, el cual es capaz de controlar la transferencia de datos.

Cuando la CPU desea leer o escribir un bloque de datos, emite un comando hacia el controlador de DMA y le envía la siguiente información:

1. Si se trata de una lectura o una escritura.
2. La dirección del dispositivo de E/S implicado.
3. La localidad inicial de memoria desde la cual se leerá o en la cual se escribirá.
4. El número de palabras a leer o escribir.

La CPU continuará entonces con otras tareas delegando la operación de transferencia de información al controlador de DMA. Dicho controlador transfiere la información directamente desde o hacia la memoria sin atravesar la CPU, cuando la transferencia se completa envía una señal de interrupción a la CPU para indicar el final de la transferencia.

El controlador de DMA transfiere la información a través del bus de datos, por ello es necesario que la CPU le ceda el bus para que se pueda llevar a cabo la transferencia. Cuando el controlador de DMA toma el control del bus puede hacer la transferencia por ráfagas lo cual consiste en transferir varias palabras en una ráfaga continua mientras el controlador de DMA domina los canales de memoria; o por robo de ciclo donde se transfiere una palabra a la vez, después de lo cual la CPU retoma el control de los canales.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Esta técnica es sumamente efectiva cuando se transmiten bloques de datos. Aunque a simple vista pueda no parecerlo, el mecanismo de robo de ciclos es sumamente efectivo, ya quelos dispositivos son muy lentos en comparación a la CPU. Por esta razón, cuando se produce un conflicto entre el dispositivo y la CPU se le da prioridad al dispositivo.

Direccionamiento de los dispositivos de E/S

Debido a que los buses no son más que líneas de comunicación compartidas por los diferentes dispositivos, es necesario definir un mecanismo por medio del cual uno de ellos pueda determinar si los datos transmitidos en un momento dado a través del bus van dirigidos a él o a otro dispositivo.

Esto se resuelve de la siguiente manera: un dispositivo conectado al bus de datos opera de tal forma, que aunque pueda recibir todos los datos que circulan por él, solo reaccionan cuando la dirección existente en ese momento en el bus de direcciones tiene un valor concreto, específicamente, la "dirección" que identifica a dicho dispositivo. Por ejemplo, la interfaz de E/S del teclado ignora cualquier dato o comando presente en los diferentes buses, si la dirección presente en el bus de direcciones no se encuentra en el rango 0x60-0x6F.

Estas direcciones son conocidas como "puertos de E/S", y deben identificar de manera unívoca a los distintos dispositivos (es decir, un puerto solo debe estar asociado a un único dispositivo). Sin embargo es posible que varios puertos estén asociados al mismo dispositivo. Cuando esto sucede (como en el caso del teclado), cada puerto tiene un uso distinto dentro del mismo dispositivo.

Bajo este esquema, los puertos identifican a los diferentes dispositivos de forma análoga a como una dirección identifica a las diferentes localidades de memoria. El procedimiento para intercambiar entre el procesador y un dispositivo de E/S es entonces similar al utilizado para intercambiar información entre el procesador y la memoria. La principal diferencia radica en que en lugar de utilizar la instrucción "mov" con direcciones de memoria, se utilizan las instrucciones "in" y "out" con números de puerto, tal y como será descrito en la siguiente sección. A continuación se presentan algunos ejemplos de los puertos más comunes en la arquitectura IA-32:

Dispositivo	Rango de puertos
Teclado	0x60 – 0x6F
Controlador de disco duro	0x1F0 – 0x1F7
Puerto de juegos	0x200 – 0x207
Puerto paralelo (LPT1)	0x378 – 0x37F
Puerto serial (COM1)	0x3F8 – 0x3FF

En Linux, es posible visualizar los puertos de E/S en uso, de la siguiente forma:

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

linux> more /proc/ioports

E/S aislada y E/S mapeada a memoria

Actualmente existen dos técnicas diferentes para permitir la interacción entre el CPU y un dispositivo periférico: La E/S aislada y la E/S mapeada a memoria. La diferencia entre ambas técnicas radica en la forma en la que los dispositivos de E/S son direccionados y las instrucciones utilizadas para intercambiar con los mismos:

E/S aislada
En la técnica de E/S aislada, las direcciones de los dispositivos de E/S están aisladas del espacio de direcciones de memoria principal, es decir, que una misma dirección puede ser utilizada para identificar un dispositivo y una localidad de memoria a la vez. No existe ambigüedad alguna, ya que las instrucciones utilizadas para acceder a los dispositivos y la memoria son distintas. Por ejemplo, la dirección 0x64 utilizada en una instrucción "mov" hac e referencia a la localidad de memoria 0x64, mientras que la misma dirección utilizada en una instrucción "in" hace referencia a uno de los puertos asociados al teclado. Este es el esquema descrito en la sección anterior.

E/S mapeada a memoria
En la técnica de E/S mapeada a memoria, una parte del rango de direcciones del subsistema de memoria es destinado para "mapear" los diferentes dispositivos de E/S. La ventaja de esta técnica consiste en que el acceso a los dispositivos ya no se hace con las instrucciones especiales "in" y "out", sino con cualquier instrucción que acceda a memoria como "mov". La desventaja consiste en que se reduce la cantidad de memoria que puede ser direccionada por las aplicaciones.

Instrucciones "in" y "out"

La arquitectura IA-32 proporciona dos instrucciones que permiten enviar o recibir datos hacia o desde un dispositivo periférico utilizando E/S aislada. Ambas instrucciones permiten la utilización de sufijos para indicar la longitud de los datos a transferir. Cuando se utiliza la instrucción in, el destino debe ser siempre el registro al, ax, o eax, dependiendo del tamaño de los datos a recibir desde el dispositivo. De forma similar, en el caso de la instrucción out, el origen debe ser el registro al, ax o eax, dependiendo del tamaño de los datos a enviar al dispositivo.

El número de puerto (el origen en caso de una instrucción in, o el destino en el caso de una instrucción out) puede especificarse como un inmediato si el mismo puede ser representado en ocho bits. Por ejemplo, las siguientes instrucciones permiten leer o escribir un byte desde o hacia el puerto 0x60:

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
inb $0x60, %al
outb %al, %0x60
```

Sin embargo, existen casos en los que el número de puerto no puede ser representado en ocho bits. En estos casos, el mismo debe ser especificado obligatoriamente mediante el contenido del registro dx. Por ejemplo, las siguientes instrucciones permiten leer o escribir un byte desde o hacia el puerto 0x3F8:

```
movx $0x3F8, %dx
inb %bx, %al
outb %al, %bx
```

Existe una variación de ambas instrucciones que permite transferir datos directamente desde la memoria hacia el dispositivo de E/S, en lugar de hacerlo desde el registro al, ax o eax. Los mnemónicos de dichas instrucciones son ins y outs. En estas instrucciones, no es necesario especificar ningún operando ya que ambos están implícitos: El número de puerto se encuentra en el registro dx, mientras que la dirección de memoria se encuentra especificada por el par de registros [es:edi] (para la instrucción ins) o [ds:esi] (para la instruccion outs), a partir del segmento de datos.

Adicionalmente, una vez ejecutadas estas instrucciones, el registro que contiene el desplazamiento a partir del cual se están almacenando (edi) o desde el cual se están tomando (esi) los datos transmitidos es incrementado en 1, 2 o 4 unidades, de acuerdo al tamaño del dato transferido (byte, palabra o doble palabra). Esto permite intercambiar bloques de datos entre la memoria y el dispositivo de una forma sumamente sencilla.

Por ejemplo, supongamos que se desea recibir un arreglo de diez enteros desde un dispositivo direccionado con el puerto 0x60, y almacenar el mismo en el segmento bss identificado con la etiqueta “A”. Esto es posible utilizando la instrucción “ins”:

```
        leal A, %edi
        movx $0x60, %edx
ciclo:
        cmpl %edi, $A + 40
        jge fciclo
        insl
        jmp ciclo
fciclo:
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

A continuación se muestran todos los formatos válidos para las instrucciones de E/S aislada de la arquitectura IA-32:

Instrucción	Efecto	Descripción
inb Puerto, %al	%al Puerto	mueve 1 byte
inw Puerto, %ax	%axl Puerto	mueve 2 bytes
inl Puerto, %eax	%eax Puerto	mueve 4 bytes

Operandos válidos		Ejemplo
Inmediato (8 bits)	registro al	inb $0x60, %al
registro dx	registro al	inl %bx, %eax

Instrucción	Efecto	Descripción
outb %al, Puerto	Puerto←%al	mueve 1 byte
outw %ax, Puerto	Puerto←%ax	mueve 2 bytes
outl %eax, Puerto	Puerto←%eax	mueve 4 bytes

Operandos válidos		Ejemplo
registro al	Inmediato (8 bits)	outb %al, $0x60
registro al	registro dx	outw %ax, %bx

Instrucción	Efecto	Descripción
insb	MEM[es:edi] ←Puerto edi edi□1	mueve 1 byte desde el puerto indicado por el registro bx
insw	MEM[es:edi] Puerto edi edi□2	mueve 2 bytes desde el puerto indicado por el registro bx
insl	MEM[es:edi] Puerto edi edi□4	mueve 4 bytes desde el puerto indicado por el registro bx

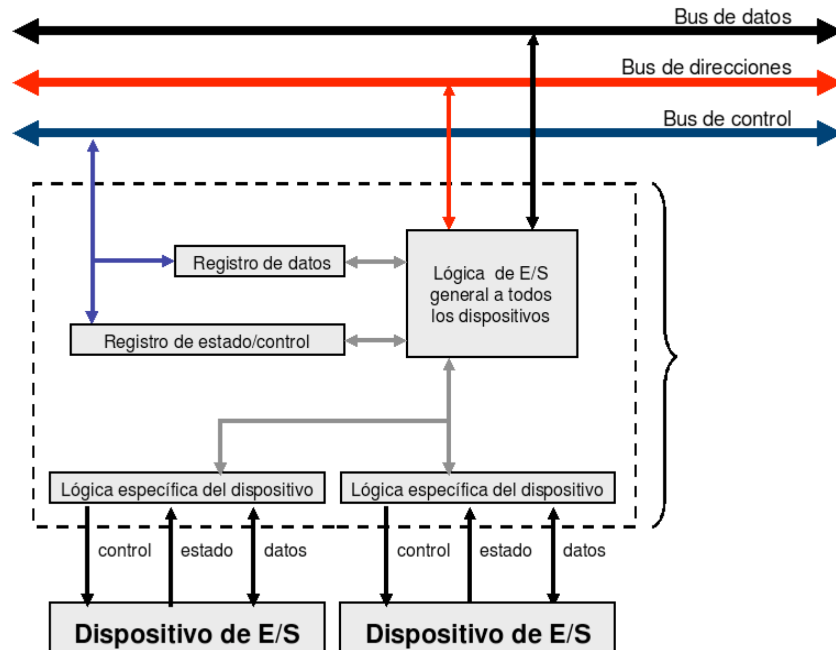
Instrucción	Efecto	Descripción
outsb	Puerto MEM[ds:esi] esi esi□1	mueve 1 byte al puerto indicado por el registro bx
outsw	Puerto MEM[ds:esi] esi esi□2	mueve 2 bytes al puerto indicado por el registro bx
outsl	Puerto MEM[ds:esi] esi esi□4	mueve 4 bytes al puerto indicado por el registro bx

Antes de comenzar con algunos ejemplos prácticos, se presenta una breve introducción a las características generales de un dispositivo de E/S, y el caso particular del teclado, el cual será utilizado en dichos ejemplos.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Dispositivos de E/S: El teclado

Anteriormente en este capítulo introdujo el concepto de interfaz de E/S. El propósito de la misma es permitir que la CPU controle una amplia variedad de dispositivos con características distintas, de una forma relativamente estandarizada. A continuación se explota el diagrama de conexión de la interfaz de E/S presentado anteriormente, con el fin de describir el diagrama de bloques básico de una interfaz de E/S:



El registro de datos actúa como buffer para los datos intercambiados entre el dispositivo y la CPU. Dicho registro puede ser escrito o leído por la misma, de acuerdo al tipo de operación que se esté realizando. Por ejemplo, en el caso de un teclado, cada vez que se presiona una tecla, el código que identifica a la misma es colocado en este registro, para ser leído posteriormente por la CPU. En el caso de un puerto serial, este registro puede ser escrito por la CPU para enviar datos al dispositivo conectado a dicho puerto.

El registro de estado refleja el estado actual del dispositivo manejado a traés de la interfaz. Por ejemplo, en el caso de un módem, dicho registro permite determinar cuando el dispositivo está listo para transmitir datos, o cuando se ha recibido uno desde el otro extremo de la línea.

Por último, el registro de control permite configurar la diferentes características de funcionamiento del dispositivo controlado. Por ejemplo, al conectarse a un router a través de un puerto serial, es necesario configurar la interfaz para que funcione a una velocidad de transmisión determinada. Dado que sobre el registro de control solo se realizan operaciones de escritura, y sobre el registro de estado solo se realizan operaciones de lectura, los mismos comparten el mismo número de puerto sin que se exista ningún tipo de ambigüedad.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

El teclado es el dispositivo de entrada más utilizado en una computadora moderna. Su registro de datos está identificado por el puerto 0x60, mientras que el puerto 0x64 se utiliza para identificar al registro de estado (en una lectura) o al registro de control (en una escritura).

Los datos proporcionados por el mismo son transmitidos a la computadora en forma de códigos de rastreo (scancodes). Los mismos no son otra cosa que un número que identifica cada una de las diferentes teclas y sus estados. Los mismos son producidos cuando el usuario presiona (o levanta) una (o varias) teclas.

Cada modelo de teclado utiliza un código interno (definido en el firmware), el cual es traducido a la forma de un código de rastreo al ser transmitido a la computadora, con la intención de estandarizar el uso de estos dispositivos. Aún así, existen diferentes conjuntos de códigos de rastreo. La mayoría de los teclados permiten seleccionar cual de dichos conjuntos se desea utilizar.

Para efectos de los ejemplos que serán presentados en la siguiente sección, tan solo es necesario comprender que el dispositivo identifica las diferentes teclas mediante códigos de rastreo, los cuales deben ser convertidos de alguna forma, generalmente utilizando una tabla de conversión (conocida como keymap) para obtener el caracter ASCII correspondiente.

Por lo general los códigos de rastreo son asignados en base a la posición que ocupa cada tecla, comenzando de arriba a abajo y de izquierda a derecha. Algunas excepciones a esta regla se hacen para mantener la compatibilidad con modelos más antiguos, en los que algunas teclas han sido añadidas y/o eliminadas.

A continuación se presentan los códigos de rastreo que serán utilizados en los ejemplos de la siguiente sección. Los mismos pertenecen al conjunto de códigos #1. Si bien el conjunto de códigos #2 es el más utilizado actualmente, el mismo es convertido al conjunto de códigos #1 por un microprocesador en la tarjeta madre de la computadora (esto es debido a razones históricas). Se recomienda consultar bibliografía especializada para obtener información más detallada al respecto.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Tecla	Código de rastreo (al presionarla)	Código de rastreo (al levantarla)
1	0x02	0x82
2	0x03	0x83
3	0x04	0x84
4	0x05	0x85
5	0x06	0x86
6	0x07	0x87
7	0x08	0x88
8	0x09	0x89
9	0x0A	0x8A
0	0x0B	0x8B
Q	0x10	0x90
W	0x11	0x91
E	0x12	0x92
R	0x13	0x93
T	0x14	0x94
Y	0x15	0x95
U	0x16	0x96
I	0x17	0x97
O	0x18	0x98
P	0x19	0x99
A	0x1E	0x9E
S	0x1F	0x9F
D	0x20	0xA0
F	0x21	0xA1
G	0x22	0xA2
H	0x23	0xA3
J	0x24	0xA4
K	0x25	0xA5
L	0x26	0xA6
Z	0x2C	0xAC
X	0x2D	0xAD
C	0x2E	0xAE
V	0x2F	0xAF
B	0x30	0xB0
N	0x31	0xB1
M	0x32	0xB2
ENTER	0x1C	0x9C

Observe que la diferencia entre el código generado al presionar una tecla, y el generado al levantarla, consiste únicamente en el séptimo bit, que en el primer caso se encuentra apagado y en el segundo encendido. Es posible visualizar la

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

tabla de traducción utilizada actualmente por su computadora, utilizando el comando *dumpkeys* en la consola de comandos de Linux.

Ahora bien, cuando el usuario presiona (o levanta) una tecla, el código de rastreo correspondiente es colocado en el registro de datos del teclado. Adicionalmente se enciende el bit 0 del registro de estado, para indicar que un código de rastreo válido se encuentra en el registro de datos.

Un programa que utilice la técnica de transferencia de E/S programada, deberá monitorear continuamente el estado de dicho bit para detectar que una tecla ha sido presionada o levantada (en las otras técnicas el teclado activa una interrupción con el fin de notificar este hecho).

De esta forma, el esqueleto básico de un programa que desea recibir el código de rastreo de una tecla que ha sido presionada o levantada es el siguiente:

```
espera: inb $0x64, %al          # Lee el registro de estado
        andb $1, %al            # Enmascara el bit 0
        cmpb $0, %al            # Si está apagado espera
        je espera
        inb $0x60,%al           # Lee el scancode del registro de datos
```

Adicionalmente, es posible enviar comandos al teclado (para lo cual existe un protocolo definido). De manera similar a lo que ocurre en el caso de la lectura de un código de rastreo, es necesario esperar a que el teclado este listo para recibir un comando antes de enviarlo. Esto se determina examinando el bit 1 del registro de estado. Si el mismo se encuentra encendido, indica que el dispositivo se encuentra ocupado. En consecuencia el programa debe esperar a que dicho bit se encuentre apagado para enviar un comando.

Los comandos (y sus parámetros, en caso de necesitarlos) se envían a través del registro de datos. Solo en casos muy específicos se escribe directamente el registro de control del teclado. Por ejemplo, para enviar el comando CMD con parametro DATA al teclado, debe seguirse el siguiente esquema:

```
espera: inb $0x64, %al          # Lee el registro de estado
        andb $2, %al            # Enmascara el bit 1
        jne espera              # Si no está listo para recibir espera

        movb CMD, %al           # CMD: Byte que identifica el comando
        outb %al,$0x60          # Envía el comando
        movb DATA, %al          # DATA: Parámetro del comando
        outb %al,$0x60          # Envía el parámetro
```

En la siguiente sección se hará un uso exhaustivo de los dos fragmentos de código aquí expuestos, con el fin de demostrar los principios de la técnica de transferencia de E/S programada en el caso específico de un teclado.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Ejemplos prácticos:

En esta sección se presentan algunos ejemplos prácticos de utilización de las instrucciones “in” y “out”. Al estudiar los mismos, debe tener en cuenta que al interactuar en lenguaje de máquina con un dispositivo de E/S se están rozando los límites que separan el software del hardware, por lo cual todo comienza a ser mucho más dependiente de este último. En otras palabras: en este nivel, el programa es 100% dependiente del hardware con el cual se esté interactuando.

En este caso se utilizará un dispositivo sencillo y al alcance de todos: el teclado. Se utilizará direccionamiento mediante E/S aislada utilizando las instrucciones “in”y “out”. Además en todos los ejemplos se utilizará la técnica de E/S programada, ya que el uso de cualquier otra técnica involucraría modificar el Sistema Operativo (por ejemplo para reemplazar el manejador de interrupciones del teclado), tarea que se encuentra fuera del alcance de este libro.

Antes de ejecutar cualquiera de estos ejemplos es necesario tener en cuenta los siguientes puntos:

- Aún cuando el programa está interactuando con el teclado mediante E/S programada, el manejador de interrupciones del teclado del Sistema Operativo se mantiene activo, por lo cual el mismo captura y envía a la consola de comandos cada una de las teclas presionadas.
- Los puertos del teclado aquí utilizados son compartidos también por el ratón. El ejecutar estos programas con el mismo conectado a la PC puede causar conflictos y derivar en un comportamiento inesperado, en especial en el último ejemplo en el que se envían comandos al teclado.
- Al estar utilizando E/S programada es posible que se pierdan algunas de las teclas presionadas. La utilización de una computadora con un CPU sumamente rápido ayuda a mejorar este aspecto, más nunca puede corregirlo del todo. Esto se nota en especial en el último ejemplo, en el que se debe esperar por un el teclado no solo para recibir datos sino también para enviar comandos.
- Dado que el programa solicita permiso para acceder directamente a los puertos del teclado, el mismo no puede ser ejecutado sin permisos de superusuario.

Con la intención de minimizar los efectos causados por los problemas anteriormente mencionados, se proponen las siguientes acciones:

- Las cadenas de salida de los programas comienzan con un caracter '\b', esto con el fin de borrar el caracter escrito por el Sistema Operativo en la consola cuando el manejador de interrupciones del teclado detecta una tecla presionada. La solución óptima sería deshabilitar dicho manejador de interrupciones, sin embargo el realizar una acción como esta escapa del alcance de este libro.
- Desconecte el ratón antes de ejecutar el programa o preferiblemente

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

antes de encender la computadora para evitar cualquier tipo de conflicto con dicho dispositivo.

- Ejecute los programas en modo texto. Si no sabe o no puede iniciar su computadora en modo texto, ejecútelos desde un terminal virtual.
- Inicie sesión como superusuario, o bien ejecute los programas con permisos de superusuario utilizando el comando sudo.
- Reduzca la frecuencia con la que el teclado detecta que una tecla es presionada mediante el uso del comando kbdrate. Esto con el fin de poder visualizar con mayor detenimiento el funcionamiento de cada uno de los programas.

Dicho todo esto, se recomienda seguir los siguientes pasos a la hora de ejecutar los programas de ejemplo. En caso de no seguir dichos pasos el comportamiento de los programas puede no ser el esperado:

- Desconecte el ratón de la computadora antes de encender la misma.
- Inicie la computadora.
- Si la misma se inicia en modo gráfico, presione las teclas CTRL+ALT+F1 para cambiar al primer terminal virtual.
- Inicie sesión como superusuario en dicho terminal.
- Cambie la frecuencia de reconocimiento y el retardo de repetición del teclado mediante el comando: “kbdrate -r 2.0 -d 1000”.
- Compile y ejecute los programas de ejemplo.
- Termine la sesión en el terminal virtual mediante el comando “exit”.
- De ser necesario, regrese al entorno gráfico presionando las teclas CTRL+ALT+F7.
- Apague la computadora.
- Conecte de nuevo el ratón.

En este primer ejemplo, simplemente se muestra al usuario el scancode recibido (en formato hexadecimal) cada vez que se presiona (o se levanta) una tecla. El programa finaliza cuando se detecta el scancode 0x1C, correspondiente a la tecla ENTER presionada:

```
        .section .data
CAD:    .asciz "\bScancode leido: 0x%2x\n"
ERROR:  .asciz "Se ha denegado el permiso sobre los puertos del
teclado\n"

        .bss
SC:     .space 1                # Espacio para almacenar el scancode
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
        .section .text
        .globl _start

_start:
        # Solicita permiso sobre los puertos del teclado
        movl $101, %eax         # Servicio 101 (ioperm)
        movl $0x60, %ebx        # Desde 0x60 (Registro de datos)
        movl $5, %ecx           # Hasta 0x64 (Registro de estado)
        movl $1, %edx           # Obtener permiso
        int $0x80               # Llamada al SO

        testl   %eax,%eax       # No se obtuvo el permiso
        jne err


espera: inb $0x64, %al          # Lee el registro de estado
        andb $1, %al            # Enmascara el bit 0
        cmpb $0, %al            # Si está apagado espera
        je espera

        xorl %eax, %eax         # Limpia eax
        inb $0x60,%al           # Lee el scancode del registro de datos
        movb %al, SC            # Salvaguarda el scancode leido

        pushl %eax              # Imprime el scancode (hex)
        pushl $CAD
        call printf
        addl $8, %esp

        cmpb $0x1C,SC           # Si es ENTER (presionada) terminar
        je fin
        jmp espera

err:    pushl $ERROR            # Informa al usuario sobre el error
        call printf
        addl $4, %esp

fin:    movl $101, %eax         # Servicio 101 (ioperm)
        movl $0x60, %ebx        # Desde 0x60 (Registro de datos)
        movl $5, %ecx           # Hasta 0x64 (Registro de estado)
        movl $0, %edx           # Liberar permiso
        int $0x80               # Llamada al SO

        movl $1, %eax           # Termina el programa
        movl $0, %ebx
        int $0x80
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Observe que el corazón del programa está compuesto de dos ciclos anidados: en el más interno se lee el registro de estado del teclado (puerto 0x64), y si verifica si el bit 0 se encuentra encendido. Mientras esto no sea así, el programa se mantiene esperando. Una vez que el usuario presiona una tecla, y el bit 0 del registro de estado se enciende, el programa lee el código de rastreo almacenado en el registro de datos del teclado (puerto 0x60) e imprime el mismo en formato hexadecimal mediante una llamada a printf.

Si dicho código de rastreo coincide con el generado al presionar la tecla ENTER (0x1C), el programa sale del ciclo externo y finaliza su ejecución. Observe además que para poder acceder a los puertos mencionados, el programa solicita permiso al sistema operativo mediante una llamada al servicio ioperm. De manera similar los permisos son liberados al finalizar la ejecución del programa.

Si se observa con detenimiento la salida de este programa, se verifica que tal y como se describió anteriormente, el presionar y levantar una tecla genera dos códigos de rastreo exactamente iguales, con la diferencia de que el primero de ellos tiene el bit 7 apagado, mientras que el segundo lo tiene encendido.

El siguiente programa es una versión modificada del anterior, en el cual en lugar de mostrar el código de rastreo recibido, se indica al usuario si la tecla (cualquiera que esta sea) ha sido presionada o levantada. Nuevamente, el programa finaliza una vez que el usuario presiona la tecla ENTER:

```
        .section .data
CAD1:   .asciz "\bTecla presionada\n"
CAD2:   .asciz "\bTecla levantada\n"
ERROR:  .asciz "Se ha denegado el permiso sobre los puertos del
teclado\n"

        .bss
SC:     .space 1                # Espacio para almacenar el scancode

        .section .text
        .globl _start

_start:
        # Solicita permiso sobre los puertos del teclado
        movl $101, %eax         # Servicio 101 (ioperm)
        movl $0x60, %ebx        # Desde 0x60 (Registro de datos)
        movl $5, %ecx           # Hasta 0x64 (Registro de estado)
        movl $1, %edx           # Obtener permiso
        int $0x80               # Llamada al SO

        testl   %eax,%eax       # No se obtuvo el permiso
        jne err
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
espera: inb $0x64, %al          # Lee el registro de estado
        andb $1, %al            # Enmascara el bit 0
        cmpb $0, %al            # Si está apagado espera
        je espera

        xorl %eax, %eax         # Limpia eax
        inb $0x60,%al           # Lee el scancode del registro de datos
        movb %al, SC            # Salvaguarda el scancode leido

        andb $0x80, %al         # Verifica el estado del bit 7
        jne sino                # Si está encendido fue levantada

si:     pushl SC                # La tecla fue presionada
        pushl $CAD1
        call printf
        addl $8, %esp
        jmp fsi

sino:   pushl SC                # La tecla fue levantada
        pushl $CAD2
        call printf
        addl $8, %esp

fsi:    cmpb $0x1C,SC           # Si es ENTER (presionado) terminar
        je fin
        jmp espera

err:    pushl $ERROR            # Informa al usuario sobre el error
        call printf
        addl $4, %esp

fin:    movl $101, %eax         # Servicio 101 (ioperm)
        movl $0x60, %ebx        # Desde 0x60 (Registro de datos)
        movl $5, %ecx           # Hasta 0x64 (Registro de estado)
        movl $0, %edx           # Liberar permiso
        int $0x80               # Llamada al SO

        movl $1, %eax           # Termina el programa
        movl $0, %ebx
        int $0x80
```

El funcionamiento de este programa es básicamente idéntico al anterior, con la diferencia de que ahora se aplica una máscara (0x80) para comprobar el estado del bit 7 en el código de rastreo recibido, y en base a esto se imprime el mensaje correspondiente.

Ahora bien, suponga que desea detallar aún más este programa, indicando al usuario la tecla que presionó (no su código de rastreo). Dado que los código de

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

rastreo no tienen una correspondencia directa con el código ASCII, no existe una forma sencilla de convertir el código de rastreo al ASCII que le corresponde para poder imprimirlo mediante la función printf.

Una instrucción sumamente poderosa que se utiliza generalmente en este tipo de programas es la instrucción “xlat”. Como su nombre lo indica, la misma sirve para llevar a cabo una “traducción”. En realidad lo que la misma realiza es una búsqueda dentro de una tabla. La instrucción no recibe parámetros de forma explícita, sino que asigna al registro %al, el byte ubicado en la posición N de la tabla, donde N es el valor que contenía dicho registro antes de la ejecución de la instrucción xlat. La tabla se ubica en memoria, y su dirección de inicio debe estar contenida en el registro ebx.

Por ejemplo, suponga que se desea traducir un dígito decimal a su código ASCII correspondiente. Esto es posible mediante el uso de una “tabla de traducción”:

```
TABLA:      .ascii “0123456789”
```

y la instrucción xlat:

```
movb NUM, %al
leal TABLA, %ebx
xlat                        # %al = MEM[%ebx+%al]
```

En el tercer ejemplo, el programa indica al usuario que tecla ha presionado, si la misma es una letra. En caso de ser cualquier otra tecla, se muestra un asterisco. Las teclas levantadas son ignoradas. Como en los casos anteriores, el programa finaliza una vez que el usuario presiona la tecla ENTER:

```
        .section .data
CAD:    .asciz "\b%c\n"
ERROR:  .asciz "Se ha denegado el permiso sobre los puertos del
teclado\n"

table:  .asciz "QWERTYUIOP****ASDFGHJKL*****ZXCVBNM"

        .bss
SC:     .space 1                # Espacio para almacenar el scancode

        .section .text
        .globl _start
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
_start:
        # Solicita permiso sobre los puertos del teclado
        movl $101, %eax         # Servicio 101 (ioperm)
        movl $0x60, %ebx        # Desde 0x60 (Registro de datos)
        movl $5, %ecx           # Hasta 0x64 (Registro de estado)
        movl $1, %edx           # Obtener permiso
        int $0x80               # Llamada al SO

        testl   %eax,%eax       # No se obtuvo el permiso
        jne err

espera: inb $0x64, %al          # Lee el registro de estado
        andb $1, %al            # Enmascara el bit 0
        cmpb $0, %al            # Si está apagado espera
        je espera

        xorl %eax, %eax         # Limpia eax
        inb $0x60,%al           # Lee el scancode del registro de datos
        cmpb $0x1C,%al          # Si es ENTER (presionada) terminar
        je fin

        movb %al, SC
        andb $0x80, %al         # Ignora las teclas levantadas
        jne espera

        cmpb $0x10, SC          # Si el scancode es menor al de la Q
        jl noletra              # No es una letra
        cmpb $0x32, SC          # Si el scancode es mayor al de la M
        jg noletra              # No es una letra

        leal table, %ebx        # Traduce el scancode a la letra
        movb SC, %al
        subb $0x10,%al
        xlat
        jmp mostrar

noletra:
        movb $'*', %al          # Si no es una letra imprime *

mostrar:
        pushl %eax              # Muestra la tecla presionada
        pushl $CAD
        call printf
        addl $8, %esp
        jmp espera
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
err:    pushl $ERROR           # Informa al usuario sobre el error
        call printf
        addl $4, %esp

fin:        movl $101, %eax    # Servicio 101 (ioperm)
        movl $0x60, %ebx       # Desde 0x60 (Registro de datos)
        movl $5, %ecx          # Hasta 0x64 (Registro de estado)
        movl $0, %edx          # Liberar permiso
        int $0x80              # Llamada al SO

        movl $1, %eax          # Termina el programa
        movl $0, %ebx
        int $0x80
```

El funcionamiento de este programa es prácticamente idéntico al del primer ejemplo, con la salvedad de que el código de rastreo recibido es “traducido” a su correspondiente ASCII (en el caso de ser una letra) o a un asterisco (en caso de ser otra tecla). Para ello se define en el segmento de datos la tabla de traducción adecuada. Observe que como los códigos de rastreo de las teclas correspondientes a las letras no ocupan un rango continuo, sino que existen algunos códigos de rastreo correspondientes a otras teclas intercalados entre estas, se han incluido algunos asteriscos dentro de la tabla.

En el caso de que se quisiera traducir el códigos de rastreo a una cadena en lugar de un caracter (por ejemplo para identificar las teclas ENTER o ESC), la tabla debería contener apuntadores a dichas cadenas. Sin embargo, dado que los elementos de la tabla son de 1 byte de longitud, tendría más sentido que la tabla reflejara la posición que ocupa el apuntador a la cadena en cuestión dentro de un arreglo cuyos elementos ocupen 4 bytes. La solución más apropiada depende del problema específico que se quiera resolver, pero en cualquier caso la instrucción xlat es una herramienta poderosa que siempre debe ser considerada.

Todos los ejemplos anteriores han hecho uso de la instrucción “in”, lo cual es natural debido a que estamos interactuando con un dispositivo de entrada como lo es el teclado. Sin embargo, es posible utilizar dicho dispositivo como elemento de salida, valiéndonos de los LEDs indicadores del mismo (NUM, CAPS y SCROLL).

Para ello se envía un comando de control que permite especificar cuales LEDs deben estar encendidos y cuales no. El programa detecta cuando el usuario presiona cualquiera de las teclas correspondientes a los números del 0 al 7 (del teclado básico, no el numérico), y muestra su representación binaria utilizando los LEDs anteriormente mencionados. En otras palabras, el programa transforma el teclado en un convertidor octal a binario de un dígito:

```
        .section .data
ERROR:  .asciz "Se ha denegado el permiso sobre los puertos del
teclado\n"
table:  .byte 0b000, 0b001, 0b100, 0b101, 0b010,0b011,0b110,0b111
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
        .section .bss
SC:     .space 1                # Espacio para almacenar el scancode

        .section .text
        .globl _start

_start:
        # Solicita permiso sobre los puertos del teclado
        movl $101, %eax         # Servicio 101 (ioperm)
        movl $0x60, %ebx        # Desde 0x60 (Registro de datos)
        movl $5, %ecx           # Hasta 0x64 (Registro de estado)
        movl $1, %edx           # Obtener permiso
        int $0x80               # Llamada al SO

        testl   %eax,%eax       # No se obtuvo el permiso
        jne err


espera: inb $0x64, %al          # Lee el registro de estado
        andb $1, %al            # Enmascara el bit 0
        je espera               # Si está apagado espera

        xorl %eax, %eax         # Limpia eax
        inb $0x60,%al           # Lee el scancode del registro de datos
        cmpb $0x1C,%al          # Si es ENTER (presionada) terminar
        je fin

        movb %al, SC
        andb $0x80, %al         # Ignora las teclas levantadas
        jne espera

        cmpb $0x0B, SC          # Si el scancode es el del 0
        je cero                 # Salta a cero

        cmpb $0x02, SC          # Si el scancode es menor al del 1
        jl espera               # No es un dígito octal
        cmpb $0x08, SC          # Si el scancode es mayor al del 7
        jg espera               # No es un dígito octal

        decb SC                 # Resta 1 al scancode
        jmp mostrar

cero:   movb $0, SC             # El dígito es 0
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
mostrar:
        inb $0x64, %al          # Lee el registro de control
        andb $2, %al            # Enmascara el bit 1
        jne mostrar             # Si no está listo para recibir espera

        movb $0xED, %al         # Comando 0xED: Luces del teclado
        outb %al,$0x60          # Envía el comando
        movb SC, %al            # Carga el digito octal
        leal table, %ebx        # Dirección de la tabla de traducción
        xlat                    # Traduce al patrón de las luces
        outb %al,$0x60          # Envía el patrón
        jmp espera


err:    pushl $ERROR            # Informa al usuario sobre el error
        call printf
        addl $4, %esp

fin:    movl $101, %eax         # Servicio 101 (ioperm)
        movl $0x60, %ebx        # Desde 0x60 (Registro de datos)
        movl $5, %ecx           # Hasta 0x64 (Registro de estado)
        movl $0, %edx           # Liberar permiso
        int $0x80               # Llamada al SO

        movl $1, %eax           # Termina el programa
        movl $0, %ebx
        int $0x80
```

En este ejemplo se utilizan las mismas técnicas que en el ejemplo anterior para esperar que una tecla sea presionada y asegurarse que sea uno de los dígitos del 0 al 7 en el teclado básico. Luego el mismo es traducido a un “patrón” de bits que representa los estados de los tres LEDs del teclado (1 encendido, 0 apagado). La idea de dicho patrón consiste en que los LEDs representen el equivalente binario del dígito octal presionado por el usuario. La tabla fue construida de la siguiente forma:

Octal	Binario	Patrón de bits a enviar		
		CAPS	NUM	SCROLL
0	000	0	0	0
1	001	0	0	1
2	010	1	0	0
3	011	1	0	1
4	100	0	1	0
5	101	0	1	1
6	110	1	1	0
7	111	1	1	1

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

El patrón que se envía al teclado debe indicar el estado de cada uno de los tres LED's en el orden CAPS-NUM-SCROLL. Sin embargo, la disposición física de los mismos en el teclado es NUM-CAPS-SCROLL. De allí que en la tabla los bits 1 y 2 se encuentren invertidos con respecto al número binario.

Una vez obtenido el patrón a enviar, el programa debe esperar nuevamente por el teclado, sin embargo en este caso no se espera a que se presione o levante una tecla, sino a que el dispositivo esté listo para recibir un comando. En este caso, el bit del registro de estado utilizado para reflejar esta condición es el bit 2 (en lugar del 0). Una vez que el teclado está dispuesto a recibir un comando, se envía el mismo (0xED, utilizado para modificar el estado de los LEDs) seguido del patrón anteriormente obtenido utilizando la instrucción outb dos veces.

Si bien los protocolos, registros y puertos varían de un dispositivo a otro, los principios de funcionamiento ilustrados en estos cuatro ejemplos son los mismos para cualquiera de ellos. Vale la pena comentar que en el lenguaje de programación C existen macros equivalentes a las instrucciones in y out, así como una función equivalente a la llamada al sistema ioperm. Los manejadores de interrupciones, por ejemplo, suelen ser escritos en lenguaje C y utilizar las macros y funciones anteriormente mencionadas.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Ejercicios

6.1- Escriba un programa que lea seis valores y los muestre por pantalla según el formato escogido por el usuario. Los formatos disponibles son:

[1] En una línea horizontal en el orden de entrada
[2] En una línea horizontal en orden inverso a la entrada
[3] De manera vertical en el orden de entrada
[4] De manera vertical en orden inverso a la entrada

Utilice las funciones *printf* y *scanf*.

6.2- Escriba un programa que lea cuatro valores enteros y calcule su suma. Debe mostrar el resultado por pantalla. Utilice las funciones *printf* y *scanf*.

6.3- Escriba un programa que lea cuatro valores enteros y calcule su promedio. Debe mostrar el resultado por pantalla. Utilice las funciones *printf* y *scanf*.

6.4- Escriba un programa que lea dos valores enteros a y b y produzca como salida un resultado según las siguientes condiciones:

- Si a=b: resultado=a
- Si a>b: resultado=a-b
- Si a<b: resultado=b-a

Utilice las funciones *printf* y *scanf*.

6.5- Escriba un programa que lea la cantidad de minutos de duración de una llamada telefónica y que calcule el monto a pagar por dicha llamada de acuerdo a la siguiente tarifa:

- Si la llamada dura hasta 3 minutos el monto a pagar será Bs. 200
- Cada minuto adicional cuesta Bs. 100

Utilice las funciones *printf* y *scanf*.

6.6- Escriba un programa que solicite al usuario introducir una cadena y dos caracteres. El programa debe reemplazar las apariciones del primer caracter con el segundo caracter. Utilice llamadas al sistema.

6.7- Modifique el primer ejemplo de la sección “Llamadas al sistema” de forma que el resultado pueda ser presentado al usuario mediante una única llamada al sistema en lugar de dos. Ayuda: Piense en la utilidad del carácter nulo al final de una cadena.

6.8- ¿Qué cambios es necesario realizar al segundo ejemplo de la sección “Llamadas al sistema” de forma que trabaje en base binaria en lugar decimal? ¿Para que rango de bases es esto válido? ¿Por qué?.

6.9- Escriba un programa que pida al usuario un entero decimal con signo. Si el número es negativo, el mismo estará precedido por el símbolo ‘-’, mientras que si el número es positivo el mismo puede o no estar precedido por el símbolo ‘+’. La lectura debe ser hecha mediante una llamada al sistema, mientras que el resultado se mostrará al usuario mediante una llamada a la función *printf*.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

6.10- Escriba un programa que permita emular el concepto de la cadena de formato de la función *scanf* utilizando únicamente llamadas al sistema. Asuma que la cadena de formato se encuentra en el segmento de datos en una etiqueta llamada “format”. De igual forma asuma que la dirección de cada uno de los argumentos adicionales se encuentran en un arreglo llamado “params”, en el que cada elemento es una dirección de memoria de 4 bytes. Los especificadores de formato válido son únicamente “%c” y “%s”.

6.11- Escriba un programa que notifique al usuario cuando una de las teclas de función (F1-F12) sea presionada o levantada. Utilice E/S programada.

6.12- Modifique el programa anterior, de manera que omita las repeticiones generadas al mantener presionada durante cierto tiempo la tecla en cuestión.

6.13- Escriba un programa que notifique al usuario cuando se detecte que se ha presionado la combinación de teclas CTRL+ALT+Q. Utilice E/S programada

6.14- Escriba un programa que convierta el teclado en una calculadora binaria de 3 bits. Para ello deberá reconocer comandos de la forma [+*][XXX], donde OP es una de las teclas '+' o '-', y X es una de las teclas '0' o '1'. El mismo deberá calcular la operación especificada (+: OR a nivel de bit, *: AND a nivel de bit) entre el acumulador (inicializado en cero) y el número binario introducido. El resultado debe ser mostrado utilizando los LEDs del teclado. El programa finaliza al presionar la tecla ENTER.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Capítulo 7. Control de flujo

Estas estructuras son las que dirigen el flujo de ejecución de un programa. Aún cuando es posible programar directamente en lenguaje ensamblador, presentamos las estructuras de control de flujo más comunes en los lenguajes de alto nivel y su traducción a ensamblador. Para los programas de alto nivel utilizamos el lenguaje de programación C. Para la programación tanto de las estructuras condicionales como de los ciclos, en lenguaje ensamblador se utilizan las instrucciones de comparación y los saltos condicionales.

Traducción de estructuras condicionales

Por ejemplo para realizar un extracto de programa que haga lo siguiente:

```
if (x>y)
        result=x-y;
else
        result=y-x;
```

Se puede traducir como:

```
1           movl x, %ebx            # %ebx = x
2           movl y, %eax            # %eax = y
3           cmpl %eax, %ebx         # compara %ebx con %eax
4           jg mayor                # si x>y salta a mayor
5           subl %ebx, %eax         # sino %eax=y-x
6           movl %eax, result       # guarda el resultado
7           jmp fin                 # salta a fin
8  mayor:   subl %eax, %ebx         # %ebx=x-y
9           movl %ebx, result       # guarda el resultado
10 fin:
```

Es importante destacar que al culminar la ejecución de las instrucciones 5 y 6, que corresponden a las acciones en caso que no se cumpla la condición, hay que incluir un salto incondicional que salte al final del programa para evitar que se ejecuten las instrucciones 8 y 9 seguidamente lo cual provocaría un error ya que se ejecutarían ambos cálculos del resultado y no sólo uno de ellos dependiendo de la condición.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

El siguiente ejemplo es equivalente al anterior:

```
1           movl x, %ebx            # %ebx = x
2           movl y, %eax            # %eax = y
3           cmpl %eax, %ebx         # compara %ebx con %eax
4           jle menorig             # si x<=y salta a menorig
5           subl %eax, %ebx         # sino %ebx=x-y
6           movl %ebx, result       # guarda el resultado
7           jmp fin                 # salta a fin
8  menorig: subl %ebx, %eax         # %eax=y-x
9           movl %ebx, result       # guarda el resultado
10 fin:
```

El siguiente programa revisa un número declarado en la sección de datos y verifica si es mayor, menor o igual a 5. Produce un mensaje de texto.

Programa en C:

```
#include <stdio.h>
int x=2;
main()
{
      if(x>5)
        printf("El número es mayor que 5\n");
      else
        if (x==5)
          printf("El número es igual a 5\n");
        else
          printf("El número es menor que 5\n");
}
```

Programa en ensamblador:

```
.section .data
x:     .long 9
mensaje1:   .asciz"El numero es igual a 5\n"
mensaje2:   .asciz"El numero es mayor que 5\n"
mensaje3:   .asciz"El numero es menor que 5\n"

.section .text
.globl _start
_start:
            movl x, %eax        # carga %eax con el valor de x
            cmpl $5, %eax       # compara x con 5
            jg mayor            # si x>5 salta a la etiqueta "mayor"
            jl menor            # si x<5 salta a la etiqueta "menor"
            pushl $mensaje1     # si x=5 imprime el mensaje 1
            call printf
            jmp fin             # salta a fin de programa
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
mayor:      pushl $mensaje2   # imprime el mensaje 2 cuando x>5
            call printf
            jmp fin           # salta a fin de programa
menor:      pushl $mensaje3   # imprime el mensaje 3 cuando x<5
            call printf
fin:        movl $1, %eax     # fin de programa
            xorl %ebx, %ebx
            int $0x80
```

Cabe destacar que la instrucción pushl $mensaje1 es equivalente a las instrucciones:

```
            leal mensaje1, %eax
            pushl %eax
```

Las cuales fueron presentadas en el capítulo 6.

Ciclos

Los ciclos permiten ejecutar un conjunto de instrucciones varias veces hasta que se cumpla una condición. El chequeo de la condición puede estar al inicio o al final del ciclo, en caso de estar al final hay que tener en cuenta que la ejecución se va a efectuar al menos una vez.

A modo de ejemplo se presentan cuatro versiones de un programa que imprime los números del 1 al 10.

Do while

Este ciclo revisa la condición al final.

```
#include <stdio.h>
int x=0;
main()
{
  do
   {
        x++;
        printf("%d ",x);
   }
  while(x<10);
  printf("\n");
}
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
# Programa que muestra por pantalla los numeros del 1 al 10
# Traduccion del do  while
.section .data
formato:    .asciz"%d "
salto:      .asciz"\n"

.section .text
.globl _start
_start:
      movl $0, %ebx     # %ebx=0, contador
do:   incl %ebx         # incrementa %ebx
      pushl %ebx        # pasa el valor como parametro
      pushl $formato    # pasa el formato como parametro
      call printf       # muestra el valor por pantalla
      addl $8, %esp     # limpia la pila de parametros
      cmpl $10, %ebx    # compara %ebx con 10
      jl do             # salta al inicio del ciclo si %ebx < 10
      pushl $salto      # imprime salto de linea
      call printf
      movl $1, %eax     # termina el programa
      movl $0, %ebx
      int $0x80
```

While

En este caso la condición se revisa al inicio del ciclo.

```
#include <stdio.h>
int x=1;
main()
{
  while(x<=10)
    {
      printf("%d ",x);
      x++;
    }
  printf("\n");
}
```

```
# Programa que muestra por pantalla los numeros del 1 al 10
# Traduccion del while
.section .data
formato:    .asciz"%d "
salto:      .asciz"\n"

.section .text
.globl _start
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
_start:
            movl $1, %ebx     # %ebx=1, contador
while:      cmpl $10,%ebx     # compara %ebx con 10
            jg finwhile       # salta a finwhile si %ebx > 10
            pushl %ebx        # pasa el valor como parametro
            pushl $formato    # pasa el formato como parametro
            call printf       # muestra el valor por pantalla
            addl $8, %esp     # limpia la pila de parametros
            incl %ebx         # incrementa %ebx
            jmp while         # salta al inicio del ciclo
finwhile:   pushl $salto      # imprime salto de linea
            call printf
            movl $1, %eax     # termina el programa
            movl $0, %ebx
            int $0x80
```

For

Esta estructura permite expresar en una sola línea el estado inicial del contador, la condición y cómo se realiza la cuenta.

```
#include <stdio.h>
int x;
main()
{
  for(x=1;x<=10;x++) printf("%d ",x);
  printf("\n");

}
```

```
# Programa que muestra por pantalla los numeros del 1 al 10
# Traduccion del for
.section .data
formato:    .asciz"%d "
salto:      .asciz"\n"

.section .text
.globl _start
_start:
            movl $1, %ebx     # %ebx=1, contador
for:        cmpl $10, %ebx    # compara %ebx con 10
            jg finfor         # si %ebx>10 salta a fin de ciclo
            pushl %ebx        # pasa el valor como parametro
            pushl $formato    # pasa el formato como parametro
            call printf       # muestra el valor por pantalla
            addl $8, %esp     # limpia la pila de parametros
            incl %ebx         # incrementa %ebx
            jmp for           # salta al inicio del ciclo
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
finfor:     pushl $salto
            call printf
            movl $1, %eax     # termina el programa
            movl $0, %ebx
            int $0x80
```

Como se puede observar, este programa, en lenguaje ensamblador, es idéntico a la implementación del while.

Los ciclos también se pueden programar utilizando la instrucción loop. A continuación se muestra una solución al programa anterior usando loop.

```
# Programa que muestra por pantalla los numeros del 1 al 10
# Uso de la instruccion loop
.section .data
formato:    .asciz"%d "
salto:      .asciz"\n"

.section .text
.globl _start
_start:
            movl $10, %ecx    # %ecx es el contador de iteraciones
            movl $1, %edi     # %edi tiene el numero a mostrar

muestra:    pushl %ecx        # guarda valor de %ecx
            pushl %edi        # apila valor a mostrar
            pushl $formato    # apila formato de salida
            call printf       # muestra numero por pantalla
            addl $8, %esp     # limpia la pila de parametros
            popl %ecx         # restaura valor de %ecx
            incl %edi         # calcula nuevo valor a mostrar
            loop muestra      # salta al ciclo para mostrar el
                              # siguiente valor

            pushl $salto      # apila formato de salto de linea
            call printf       # realiza el salto de linea

            movl $1, %eax     # fin del programa
            xorl %ebx, %ebx
            int $0x80
```

En este caso hay que salvaguardar el valor de %ecx en la pila antes de la llamada a la función printf y luego restaurar ese registro debido a que las llamadas a funciones o procedimientos pueden modificar algunos registros, entre ellos %ecx. Esto se explica en detalle en el capítulo 9.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Ejercicios

7.1- Escriba un programa que lea un entero y que imprima el mensaje "Hola" tantas veces como el número de entrada. El mensaje debe aparecer en pantalla llenando líneas horizontales.

7.2- Escriba un programa que lea un entero y que imprima el mensaje "Hola" tantas veces como el número de entrada. El mensaje debe aparecer en pantalla una sóla vez por cada línea horizontal.

7.3- Escriba un programa que calcule xy.

7.4- Escriba un programa que determine el monto a pagar a un trabajador semanal según la siguiente estructura de pagos:

El pago por hora de trabajo (sueldo básico) puede ser:

- Bs. 3.250,oo para un trabajador no calificado
- Bs. 4.500,oo para un trabajador calificado
- Bs. 4.850,oo para un trabajador calificado con experiencia

El monto a pagar se calcula de la siguiente manera:

- Hasta 40 horas de trabajo se pagan con sueldo básico
- De 40 a 60 horas se pagan con sueldo básico y medio
- Más de 60 horas se pagan con doble sueldo básico

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Capítulo 8. Arreglos

Cuando se declara un arreglo en C se escribe una expresión con la siguiente estructura: T A[N] esta declaración designa el nombre del arreglo como A de N posiciones del tipo de dato T. Esto produce la asignación de un espacio de memoria de N*T, es decir el número de elementos del arreglo por el número de bytes del tipo de dato. Los elementos del arreglo se acceden con un índice cuyo rango es { 0, N-1}.

Ejemplo de declaración y parámetros de arreglos en C:

```
char A[12];
int    B[8];
double C[6];
```

Sus parámetros son los siguientes:

Arreglo	Tamaño de un dato	Tamaño total del arreglo	Dirección inicial	Elemento i
A	1 byte	12 bytes	X_A	$X_A + i$
B	4 bytes	32 bytes	X_B	$X_B + 4i$
C	8 bytes	48 bytes	X_C	$X_C + 8i$

En lenguaje ensamblador se utiliza el tipo de direccionamiento escalado para acceder a las posiciones de un arreglo. Hay que tomar en cuenta la dirección inicial del arreglo y utilizar un índice para ir recorriendo cada uno de sus elementos. El tamaño de los datos representa el desplazamiento entre dos posiciones del arreglo.

Declaración de arreglos

En lenguaje ensamblador los arreglos se declaran como una variable con varios valores, por ejemplo si se declara el arreglo *a* como:

```
.section .data
a:   .long 25,48,32,87
```

Los elementos del arreglo se guardan en memoria en posiciones consecutivas y el identificador *a* apunta a la primera posición del arreglo, es decir la posición cuyo índice es cero. Cada elemento del arreglo tiene el tamaño del tipo de dato asociado, en este caso cada elemento ocupa 4 bytes debido a que el arreglo está declarado como tipo *long* y el tamaño total del arreglo es de 16 bytes.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

En este otro ejemplo:

```
.section .data
arr:  .byte 5,4,3,7
```

Cada elemento del arreglo ocupa 1 byte. El tamaño total de este arreglo es de 4 bytes.

Lectura de datos en un arreglo

La forma general de accesar un arreglo se puede expresar como:

desplazamiento (base, índice, escala)

La dirección se calcula como:

desplazamiento + dirección base + índice *escala

Donde escala refleja el tamaño del tipo de dato del arreglo.

Para leer un elemento del arreglo *a* se puede utilizar la siguiente instrucción:

```
movl a(,%ecx,4), %eax
```

donde *a* es la dirección inicial del arreglo y se utiliza un registro, en este caso %ecx como índice. La escala (4) viene dada por el tamaño del tipo de dato del arreglo.

Hay varias maneras de leer información de un arreglo. A continuación se presentan dos soluciones para leer los valores de la primera y tercera posiciones de un arreglo de enteros:

```
.section .data
      arr:  long 10,20,30,40,50,60,70,80        #arreglo arr de enteros

.section .text
.globl _start
_start:
    # En ambas soluciones se coloca el valor de la primera posición
    # del arreglo (10) en %eax y el valor de la tercera posición del
    # arreglo (30) en %ecx; arr denota la dirección inicial del
    # arreglo. La escala es 4 porque el tipo de dato (entero) ocupa
    # 4 bytes de memoria.

# solución 1
      movl $0, %ebx                     # actualiza %ebx en 0 (índice=0)
      movl arr(, %ebx, 4), %eax         # dirección= arr+(0*4),transfiere
                                        #  el valor 10 a %eax
      movl $2, %ebx                     # actualiza %ebx en 2 (índice=2)
      movl arr(, %ebx, 4),%ecx          # dirección= arr+(2*4),transfiere
                                        # el valor 30 a %ecx
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
# solución 2
      leal arr, %edx                    # %edx contiene la dirección
                                        # inicial del arreglo arr
      movl $0, %ebx                     # actualiza %ebx en 0 (índice=0)
      movl (%edx, %ebx, 4), %eax        # dirección=arr+(0*4),transfiere
                                        #  el valor 10 a %eax
      movl $2, %ebx                     # actualiza %ebx en 2 (índice=2)
      movl (%edx, %ebx, 4), %ecx        # dirección= arr+(2*4),transfiere
                                        # el valor 30 a %ecx
```

Ejemplo: programa que dado un arreglo de enteros de 10 posiciones, inicializado, suma los valores del arreglo y guarda el resultado en la variable "result"

```
.section .data
arr: .long 34, 5, 8, 12, 43, 21, 9, 6, 92, 10
result: .long 0

.section .text
.globl _start
_start:
            movl $0, %eax               # inicializa %eax en 0
            movl $0, %ecx               # inicializa el índice
suma:       addl arr(,%ecx,4), %eax     # suma un nuevo valor
            incl %ecx                   # incrementa el índice
            cmpl $10, %ecx              # compara i con 10
            jl  suma                    # si i<10 salta a suma
            movl %eax, result           # actualiza result
            movl $1, %eax               # fin del programa
            xorl %ebx, %ebx
            int $0x80
```

Ejemplo: programa que dado un arreglo de 10 caracteres, indica la cantidad de ocurrencias de la letra "a".

```
section .data
mensaje:    .asciz"Programa que dado un arreglo de 10 caracteres\n"
mensaj2:    .asciz"indica la cantidad de ocurrencias de la letra a\n"
salida:     .asciz"En el arreglo hay %d veces la letra a\n"
salto:      .asciz"\n\n"

arr:        .byte 'a','b','c','d','a','s','a','w','f','a'

.section .text
.globl _start
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
_start:
            pushl $mensaje          # mensaje inicial
            call printf
            pushl $mensaj2
            call printf
            addl $8, %esp

            xorl %eax, %eax         # acumulador de ocurrencias
            xorl %edx, %edx         # indice para el arreglo
verif:      cmpl $10, %edx
            je finverif
            movb arr(%edx), %bl     # lee caracter
            cmpb $'a', %bl          # compara con a
            jne noa
            incl %edx               # si = a incrementa el indice
            incl %eax               # y el acumulador
            jmp verif               # lee siguiente caracter
noa:        incl %edx               # si dif a solo incrementa indice
            jmp verif

finverif:   pushl %eax              # muestra resultado por pantalla
            pushl $salida
            call printf

            movl $1, %eax           # fin del programa
            xorl %ebx, %ebx
            int $0x80
```

Escritura de datos en un arreglo

La escritura de datos en un arreglo se hace de manera similar, por ejemplo el siguiente programa escribe los números del 1 al 10 en un arreglo de 10 posiciones:

```
.section .bss
arr: .space 40    # se declara el espacio de 40 bytes para generar
                  # el arreglo de 10 posiciones de enteros

.section .text
.globl _start
_start:
            leal arr, %ebx          # %ebx=dir inicial del arreglo
            xorl %ecx, %ecx         # %ecx=0 indice
            movl $1, %eax           # valor a guardar en el arreglo
genera:     cmpl $9, %ecx           # compara el indice con 9
            jg fin                  # salta a fin cuando indice=10
            movl %eax, (%ebx, %ecx, 4)    # escribe valor en arr
            incl %eax               # incrementa el valor
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
            incl %ecx               # incrementa el indice
fin:        movl $1, %eax           # fin del programa
            movl $0, %ebx
            int $0x80
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Ejercicios

8.1- Escriba un programa que lea un arreglo de n enteros y sume los elementos del arreglo.

8.2- Escriba un programa que lea un arreglo de 20 enteros y calcule el promedio de los números del arreglo.

8.3- Escriba un programa que lea un arreglo de n enteros y calcule la cantidad de números pares.

8.4- Escriba un programa que lea un arreglo de n enteros y calcule la sumatoria de los números pares.

8.5- Escriba un programa que lea un arreglo de 12 enteros determine el mayor e indique en que posición se encuentra.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Capítulo 9. Procedimientos

Los procedimeintos son segmentos de programa sumamente útiles para evitar la repetición de código, un programa con procedimientos es además mucho más legible.

Una llamada a un procedimiento involucra el pase de información tanto hacia el procedimiento como de retorno y el pase del control de flujo de ejecución de una parte del código hacia otra. El pase de información desde y hacia el procedimiento se realiza a través de la pila. La porción de la pila asignada a una llamada a un procedimiento se conoce como marco de pila. Para utilizar la pila se usan dos registros que sirven de apuntadores: %ebp es el apuntador de marco de pila y %esp es el apuntador de pila, éste es el registro cuyo contenido cambia mientras se ejecuta el procedimiento.

Instrucciones para llamadas a procedimientos

Las instrucciones que se utilizan para transferir el control en los procedimientos son las siguientes:

Instrucción	Descripción
call etiqueta	llamada a procedimiento
leave	preparación para el retorno
ret	retorno de procedimiento

Cuando se ejecuta una llamada a procedimiento (call) el sistema apila la dirección de retorno y transfiere el control al procedimiento. La dirección de retorno es la dirección de la instrucción siguiente a la llamada. La instrucción de retorno (ret) retira la dirección de retorno de la pila y pasa el control a esta dirección. Dentro del procedimiento hay que preparar la pila para su uso, esto se hace al inicio del procedimiento con las instrucciones:

```
pushl %ebp        # preserva marco de pila anterior
movl %esp, %ebp   # actualiza marco de pila
```

Y luego se prepara la pila para el retorno al programa invocador, lo cual se realiza con la instrucción leave que se ejecuta antes de ret. La instrucción leave reemplaza las siguientes dos instrucciones:

```
movl %ebp, %esp   # mueve el apuntador de pila al marco de
                  # pila anterior
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
        popl %ebp           # restaura el marco de pila anterior
```

Hay procedimientos que retornan un valor como resultado, a este tipo de procedimiento se les conocen como función, en este caso se usa el registro %eax para retornar el resultado.

Convenciones para preservar los contenidos de los registros

Sólo hay un conjunto de registros de usuario y éstos pueden ser usados tanto por el programa principal como por el procedimiento invocado. Para evitar que se pierda información de los registros la arquitectura IA-32 adopta una serie de convenciones las cuales deben ser respectadas al momento de escribir código que incluya procedimientos. Los registros %eax, %edx y %ecx son preservados por el programa invocador. Los registros %ebx, %esi y %edi son preservados por el procedimiento invocado. El procedimiento debe guardar estos valores en la pila antes de utilizarlos y restaurar sus valores antes de regresar al programa invocador.

Estructura del programa con llamada a procedimientos

Cuando se escribe un programa con llamada a procedimiento se realizan los siguientes pasos:

Programa invocador:

- Preservar los registros %eax, %edx y %ecx
- Apilar los parámetros en orden inverso
- Realizar la llamada al procedimiento

Procedimiento:

Prólogo:

- Preservar el marco de pila anterior
- Actualizar el marco de pila
- Preservar los registros %ebx, %esi, %edi

Cuerpo del procedimiento:

- Código del procedimiento

Epílogo:

- Mover el valor de retorno a %eax (si es el caso)
- Restaurar los registros %ebx, %esi, %edi

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

- Actualizar el apuntador de pila al marco de pila
- Restaurar el marco de pila anterior
- Retornar al programa invocador

Programa invocador:

- Limpiar la pila (liberar el espacio ocupado por los prámetros de la llamada)
- El resultado (de existir) está en %eax y debe ser movido a otra localidad
- Restaurar los registros %ecx, %edx y %eax

Los pasos listados anteriormente se realizan con las siguientes instrucciones:

```
# programa invocador:
      pushl %eax        # se preservan los registros del invocador
      pushl %ecx
      pushl %edx
      pushl arg2        # se apilan los parámetros
      pushl arg1
      call procedimiento
      addl $x,%esp      # el valor x depende del número de bytes que
                        # ocupen los parámetros, esta instrucción
                        # limpia la pila eliminando los parámetros
      movl %eax, valor  # si el procedimiento retorna un valor este
                        # estará en %eax
      popl %edx         # se restauran los registros
      popl %ecx
      popl %eax

# procedimiento:
# prólogo:
      pushl %ebp        # se preserva marco de pila anterior
      movl %esp, %ebp   # se actualiza marco de pila
      pushl %ebx        # se preservan los registros del invocado
      pushl %esi
      pushl %edi
# cuerpo del programa
      # aqui van las instrucciones del procedimiento
# epílogo:
      movl resul, %eax  # si hay un valor de retorno se guarda en %eax
      popl %edi         # se restauran los registros
      popl %esi
      popl %ebx
      leave             # se prepara la pila para el retorno
      ret               # regresa el control al programa invocador
```

En algunos casos no es necesario preservar todos los registros.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Pase de parámetros

Los parámetros a pasar hacia el procedimiento pueden ser de dos tipos: por valor o por referencia. Cuando se pasa un parámetro por valor se genera una copia del valor del parámetro y ese dato es el que está disponible en el procedimiento, el procedimiento o función no puede cambiar el parámetro. Cuando se pasa un parámetro por referencia realmente se pasa la dirección del parámetro, no su valor, y al estar la dirección disponible durante la ejecución del procedimiento, el valor allí guardado puede ser modificado.
Para pasar información acerca de un arreglo, se pasa la dirección inicial del arreglo.

Ejemplo de programa con pase de parámetro por valor

El siguiente es un programa con una función que calcula el cuadrado de un número entero. Paso de un parámetro por valor. Retorna el resultado como un entero.

Programa en C:

```
#include <stdio.h>
int cuadrado(int x);

int t=10;
int s;
main()
{
	s=cuadrado(t);
	printf("cuadrado: %d\n", s);
}

int cuadrado(int x)
{
	x=x*x;
	return x;
}
```

Programa en ensamblador:

```
.section .data
t: .long 10
formato: .asciz "cuadrado: %d\n"
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

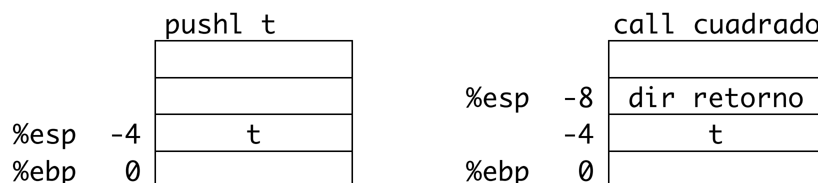
```
.section .text
.globl _start
_start:
    pushl t                          # se apila el parámetro t
    call cuadrado                    # llamada al procedimiento
    addl $4, %esp                    # limpieza de la pila (4 bytes)
    pushl %eax                       # mostrar el resultado por pantalla
    pushl $formato
    call printf
    movl $1, %eax                    # termina el programa
    movl $0, %ebx
    int $0x80

cuadrado:   pushl %ebp               # preserva marco de pila anterior
            movl %esp, %ebp          # actualiza marco de pila
            movl 8(%ebp), %eax       # obtiene el parámetro
            imull %eax, %eax         # calcula x^2. Resultado en %eax
            leave
            ret                      # retorno al programa principal
```

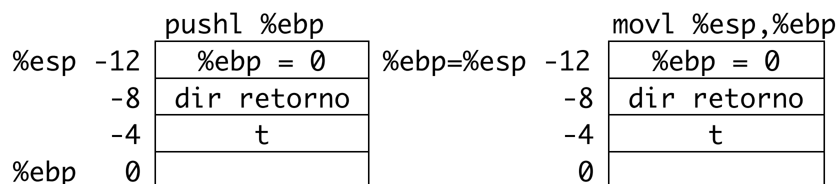
Estado de la pila

Se denotan los cambios en las direcciones sólo con un desplazamiento a partir de la dirección inicial de %ebp.

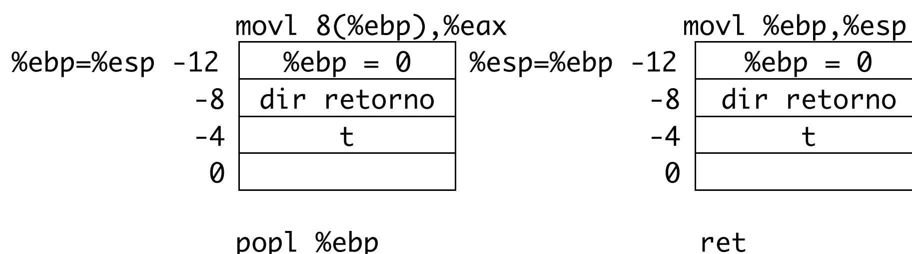
Cuando se apila el parámetro y se realiza la llamada al procedimiento:



En el procedimiento:



leave es equivalente a las instrucciones movl %ebp, %esp y popl %ebp



Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
         -12  |   %ebp = 0    |            -12  |   %ebp = 0    |
    %esp -8   | dir retorno   |             -8  | dir retorno   |
         -4   |       t       |        %esp -4  |       t       |
    %ebp  0   |               |        %ebp  0  |               |
```

La instrucción ret produce la lectura de la pila la cual tiene la dirección de retorno. Luego en el programa principal se limpia la pila incrementando %esp para eliminar el parámetro:

```
                addl $4, %esp
            -12 |   %ebp +0     |
             -8 | dir retorno   |
             -4 |       t       |
%ebp=%esp     0 |               |
```

Ejemplos de programas con pase de parámetro por referencia

El siguiente programa contiene un procedimiento que duplica el valor de los números de un arreglo de 4 posiciones. Paso de parámetro por referencia (la dirección inicial del arreglo) y por valor (el número de posiciones del arreglo).

Programa en lenguaje C:

```
#include <stdio.h>

main()
{
  int a[4];
  int n=4;
  int i;
  a[0]=10;
  a[1]=20;
  a[2]=30;
  a[3]=40;
  duplica(a,n);
  for(i=0;i<n;i++)
    printf("%d\t",a[i]);
  printf("\n");
}
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```c
void duplica(int arr[4],int y)
{
  int i;
  int temp;
  for(i=0;i<y;i++)
    {
      temp=arr[i]*2;
      arr[i]=temp;
    }
}
```

Programa en ensamblador:

```
.section .data
a: .long 10, 20, 30, 40
n: .long 4
formato: .asciz "%d\n"

.section .text
.globl _start
_start:
        leal a, %eax            # obtiene dirección del arreglo a
        pushl %eax              # apila la dir de a
        pushl n                 # apila el valor de n=4
        call duplica            # llamada al procedimiento duplica
        addl $8, %esp           # limpieza de la pila (2 parámetros)
        xorl %ebx, %ebx         # %ebx=0
        xorl %edi, %edi         # %edi=0, índice para el arreglo
ciclo:  movl a(,%ecx,4), %ebx   # lee un número del arreglo
        pushl %ebx              # apila el número
        pushl $formato
        call printf             # imprime el número
        incl %edi
        cmpl n, %edi
        jl ciclo                # imprime mientras el indice<n
        movl $1, %eax           # culmina el programa
        movl $0, %ebx
        int $0x80

duplica:
        pushl %ebp              # preserva marco de pila anterior
        movl %esp, %ebp         # actualiza marco de pila
        movl 8(%ebp), %esi      # %esi=número de elementos de a
        movl 12(%ebp), %edx     # %edx= la dirección de a
        xorl %ecx, %ecx         # %ecx se usa como índice
for:    movl (%edx,%ecx,4), %eax  # lee número del arreglo
        addl %eax, %eax         # duplica el valor leído
        movl %eax, (%edx,%ecx,4)  # reeemplaza el valor
        incl %ecx               # incrementa el índice
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
        cmpl %esi, %ecx          # compara el índice con n
        jl for                   # salta por menor al ciclo del for
        leave                    # prepara la pila para el retorno
        ret                      # retorno al programa principal
```

El siguiente programa tiene una función que calcula la suma de los elementos de un arreglo de enteros de 10 posiciones. Se leen los datos del arreglo y se muestra por pantalla el resultado. Paso de parámetro por referencia. La función suma retorna un valor entero.

```
#include <stdio.h>

int suma(int arreglo[10]);

void leer(int arreglo[10]);

main()
{
  int a[10];
  int resultado;
  leer(a);
  resultado=suma(a);
  printf("El resultado de sumar los elementos del arreglo es: %d\n", resultado);
}

void leer(int arreglo[10])
{
  int x,i;
  printf("Introduzca 10 enteros:\n");
  for(i=0;i<10;i++)
    {
    scanf("%d",&x);
    arreglo[i]=x;
    }

}

int suma(int arreglo[10])
{
  int i;
  int res=0;
  for(i=0;i<10;i++)
    res=res+arreglo[i];
  return res;
}
```

```
# Programa que lee un arreglo de enteros de 10 posiciones
# y suma los elementos del arreglo
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
.section .data
leearr:     .asciz"Introduzca los 10 numeros del arreglo:\n"
dato:       .asciz"%d"
salida:     .asciz"La suma de los elementos del arreglo es: %d\n"

.section .bss
a:    .space 40

.section .text
.globl _start
_start:
      pushl $leearr
      call printf             # imprime mensaje inicial
      leal dato, %edx
      pushl %edx
      leal a, %eax
      pushl %eax
      call leer               # invoca el proc para leer el arreglo
      addl $8, %esp
      leal a, %eax
      pushl %eax
      call suma               # invoca la funcion para sumar contenidos
      addl $4, %esp
      pushl %eax
      pushl $salida
      call printf             # imprime el resultado
      movl $1, %eax
      movl $0, %ebx
      int $0x80

leer:       pushl %ebp
            movl %esp, %ebp
            movl 8(%ebp), %ebx      # %ebx=dir inicial del arreglo
            movl 12(%ebp), %esi     # %esi=dir formato de lectura
            xorl %edi, %edi         # %edi-contador de elementos
leenum:     pushl %ebx
            pushl %esi
            call scanf              # lee un nuevo elemento del arreglo
            addl $4,%ebx            # actualiza dir para siguiente num.
            incl %edi               # incrementa contador
            cmpl $10,%edi           # compara contador con 10
            jl leenum               # mientras contador<10 lee numero
            leave
            ret

suma:       pushl %ebp
            movl %esp, %ebp
            movl 8(%ebp), %ebx      # %ebx=dir inicial del arreglo
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
            xorl %eax, %eax         # %eax se usa para acumular la suma
            xorl %ecx, %ecx         # %ecx indice para el arreglo
sumar:      addl (%ebx,%ecx,4), %eax      # suma nuevo elemento
            incl %ecx               # incrementa el indice
            cmpl $10,%ecx           # compara el indice con 10
            jl sumar                # mientras indice<10 vuelve a sumar
            leave
            ret
```

Ejemplo de programa con una llamada a una función dentro de un ciclo

Es importante destacar que en el caso de llamadas a procedimiento dentro de un ciclo hay que recordar que el programa invocador es responsable de salvaguardar los registros %eax, %ecx y %edx si alguno de ellos controla el funcionamiento del ciclo o su contenido debe permanecer inalterado luego de la ejecución del procedimiento ya que estos registros pueden ser modificados por el procedimiento. Por ejemplo el siguiente programa multiplica dos números usando el método de sumas sucesivas y mostrando por pantalla el número de cada iteración:

```
# Programa que calcula x multiplicado por y por sumas sucesivas
# Muestra el numero de iteracion cada vez que se realiza una suma
# y el subtotal de la suma
.section .data
x:     .long 25
y:     .long 4
iterac:     .asciz"Iteracion numero: %d, subtotal: %d\n"
salida:     .asciz"%d multiplicado por %d es: %d\n"

.section .text
.globl _start
_start:
            movl x, %ebx
            movl $1, %edi
            movl $0, %esi
suma:       cmpl y, %edi
            jg finsum
            addl %ebx, %esi
            pushl %esi
            pushl %edi
            pushl $iterac
            call printf
            incl %edi
            jmp suma
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
finsum:     pushl %esi
            pushl y
            pushl x
            pushl $salida
            call printf

            movl $1, %eax
            movl $0, %ebx
            int $0x80
```

Este programa funciona bien, pero si cambiamos el registro %edi por %edx, entonces el programa arroja un resultado inválido.

```
# Programa que calcula x multiplicado por y por sumas sucesivas
# Muestra el numero de iteracion cada vez que se realiza una suma
# y el subtotal de la suma
.section .data
x:     .long 25
y:     .long 4
iterac:     .asciz"Iteracion numero: %d, subtotal: %d\n"
salida:     .asciz"%d multiplicado por %d es: %d\n"

.section .text
.globl _start
_start:
            movl x, %ebx
            movl $1, %edx
            movl $0, %esi
suma:       cmpl y, %edx
            jg finsum
            addl %ebx, %esi
            pushl %esi
            pushl %edx
            pushl $iterac
            call printf
            incl %edx
            jmp suma

finsum:     pushl %esi
            pushl y
            pushl x
            pushl $salida
            call printf

            movl $1, %eax
            movl $0, %ebx
            int $0x80
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Esto se debe a que el contenido de %edx es modificado durante la ejecución de la función printf y ello afecta el funcionamiento del ciclo.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Ejercicios

9.1- Escriba un programa que lea un arreglo de 20 enteros y calcule el promedio de los números pares del arreglo. Debe invocar un procedimiento que indique cuántos números son pares y calcule su suma.

9.2- Escriba un programa que lea un arreglo de n enteros y calcule la sumatoria de los números mayores que 10 y menores que 100 presentes en el arreglo.

9.3- Escriba un programa que dado un número de entrada "n", muestre por pantalla el factorial de n (n!).

9.4- Escriba un programa que dado un número de entrada "n", siendo n mayor que 2, muestre por pantalla la serie de Fibonacci de los primeros n números. La serie de Fibonacci se genera con la siguiente expresión:

$$Fn=Fn-1+Fn-2 \quad \text{con } F1=F2=1$$

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Capítulo 10. Cadenas de caracteres

El uso de las cadenas de caracteres es imprescindible para el programador debido a que es el lenguaje natural de comunicación entre las personas. Aún cuando se pueden desarrollar programas que manejen cadenas de caracteres utilizando las instrucciones básicas presentadas en el Capítulo 5 existen una serie de instrucciones dedicadas al manejo de este tipo de datos que tienen como objetivo el desarrollo de programas más eficientes. Esta sección introduce estas instrucciones.

Instrucción movs

Esta instrucción permite el movimiento de datos entre dos localidades de memoria. Los operandos son implícitos, el operando fuente es el registro %esi el cual apunta a la cadena a mover y el operando destino es el registro %edi el cual apunta al área de memoria destino.

Instrucción	Descripción
movsb	Mueve un byte
movsw	Mueve 2 bytes
movsl	Mueve 4 bytes

Para poder realizar el movimiento de una cadena es necesario primero actualizar los registros %esi y %edi con las direcciones de la fuente y del destino, respectivamente. Esto se puede hacer utilizando la instrucción leal o direccionamiento indirecto (ver capítulo 4):

```
      leal cadena1, %esi       # primera opción, usando leal
      movl $cadena1, %esi      # segunda opción usando direccionamiento
                               # indirecto
```

Cada vez que se invoca la instrucción movs los registros %esi y %edi se actualizan automáticamente incrementándose o decrementándose el número de bytes de la instrucción ejecutada. El incremento o decremento depende del valor de la bandera DF ubicada en el registro EFLAGS. Si la bandera está en cero, los registros serán incrementados, si está en uno, serán decrementados. Se puede actualizar la bandera usando las instrucciones `cld` para colocar la bandera en 0 y `std` para colocarla en 1.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

El siguiente es un programa que mueve una cadena declarada en la sección de datos. El primer programa mueve una cadena de un caracter, el siguiente mueve una cadena de cuatro caracteres.

```
# Programa que mueve una cadena de una posicion de memoria a otra
# Cadena de un caracter
.section .data
fuente:     .asciz"a"
salida:     .asciz"%s\n"

.section .bss
destino: .space 2

.section .text
.globl _start
_start:
      leal fuente, %esi       # %esi=dir de inicio de la cadena fuente
      leal destino, %edi      # %edi=dir de inicio de la cadena destino
      movsb                   # mueve un byte, es decir, un caracter

      pushl $destino          # muestra la cadena por pantalla
      pushl $salida
      call printf

      movl $1, %eax
      movl $0, %ebx
      int $0x80

# Programa que mueve una cadena de una posicion de memoria a otra
# Cadena de 4 caracteres
.section .data
fuente:     .asciz"abcd"
salida:     .asciz"%s\n"
.section .bss
destino: .space 5

.section .text
.globl _start
_start:
      leal fuente, %esi       # %esi=dire de inicio de la cadena fuente
      leal destino, %edi      # %edi=dir de inicio de la cadena destino
      movsl                   # mueve 4 bytes, es decir, 4 caracteres

      pushl $destino          # muestra la cadena por pantalla
      pushl $salida
      call printf
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
      movl $1, %eax
      movl $0, %ebx
      int $0x80
```

El prefijo rep

Este prefijo se puede utilizar para repetir la ejecución de una instrucción de manejo de cadena de caracteres. Es controlado implícitamente por el registro %ecx, repite la instrucción hasta que %ecx sea igual a 0.

Hay que actualizar el registro %ecx con la cantidad de movimientos de datos que se van a realizar, por ejemplo si queremos mover una cadena de 8 bytes y usamos la instrucción movsb entonces hay que colocar %ecx en 8, si usamos movsw, %ecx debe estar en 4 y si usamos movsl entoces %ecx debe ser actualizado en 2.

Ejemplo de un programa que usa el prefijo rep para mover una cadena un byte a la vez, es decir, un caracter a la vez.

```
# Programa que mueve una cadena de una posicion de memoria a otra
.section .data
fuente:     .asciz"abcdefghijklmnopqrstuvwxyz"
salida:     .asciz"%s\n"
tamano:     .long 26          # numero de caracteres de la cadena

.section .bss
destino: .space 27

.section .text
.globl _start
_start:
      leal fuente, %esi       # %esi=dir de inicio de la cadena fuente
      leal destino, %edi      # %edi=dir de inicio de la cadena destino
      movl tamano, %ecx       # %ecx se inicializa en 26
      rep movsb               # mueve un byte a la vez durante 26 veces

      pushl $destino          # muestra la cadena por pantalla
      pushl $salida
      call printf

      movl $1, %eax
      movl $0, %ebx
      int $0x80
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Existen cuatro prefijos adicionales que además de monitoriar el estado del registro %ecx dependen del valor de la bandera de cero (ZF), estos son:

```
repe       repite mientras igual
repne      repite mientras no igual
repnz      repite mientras no cero    (equivalente a repne)
repz       repite mientras cero       (equivalente a repe)
```

La instrucción lods

La instrucción lods se utiliza para mover una cadena de caracteres desde memoria hacia el registro %eax, usa el registro implícito %esi para indicar la dirección de la cadena.

Instrucción	Descripción
lodsb	Carga un byte en %al
lodsw	Carga 2 bytes en %ax
lodsl	Carga 4 bytes en %eax

La instrucción stos

La instrucción stos se usa para transferir una cadena que está en el registro %eax a una posición de memoria. Usa como operando destino implícito el registro %edi.

Instrucción	Descripción
stosb	Transfiere un byte de %al a memoria
stosw	Transfiere dos bytes de %ax a memoria
stosl	Transfiere cuato bytes de %eax a memoria

Programa que modifica una cadena de caracteres cambiando su representación en minúsculas a mayúsculas.

```
# Programa que cambia una cadena de caracteres en minusculas
# a la misma cadena en mayusculas
.section .data
fuente:     .asciz"abcdefghijklmnopqrstuvwxyz"
salida:     .asciz"%s\n"
tamano:     .long 26    # numero de caracteres de la cadena

.section .bss
destino: .space 27

.section .text
.globl _start
_start:
      leal fuente, %esi       # %esi=dir de inicio de la cadena fuente
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
      leal destino, %edi      # %edi=dir de inicio de la cadena destino
      movl tamano, %ecx       # inicia %ecx con  tamano de la cadena

codif:
      lodsb              # carga caracter en %al
      subb $32, %al      # resta 32 a la representacion del caracter
      stosb              # guarda el caracter en la cadena destino
      loop codif         # vuelve a entrar en el ciclo mientras %ecx>0

      pushl $destino     # muestra la cadena por pantalla
      pushl $salida
      call printf

      movl $1, %eax      # fin del programa
      movl $0, %ebx
      int $0x80
```

La instrucción cmps

La instrucción cmps se usa para comparar cadenas de caracteres, los operandos están implícitos en los registros %esi y %edi como en los casos anteriores. Esta instrucción resta el valor de la cadena de destino del valor de la cadena fuente y esta operación modifica las banderas en EFLAGS.

Instrucción	Descripción
cmpsb	Compara el valor de un byte
cmpsw	Compara el valor de 2 bytes
cmpsl	Compara el valor de 4 bytes

El siguiente programa compara dos cadenas de caracteres.

```
# Programa que solicita dos cadenas por teclado, las compara  e indica
si son iguales
.section .data
solicita1:  .asciz"Introduzca la primera cadena: "
solicita2:  .asciz"Introduzca la segunda cadena: "
salto:      .asciz"\n"
formato:    .asciz"%s"
iguales:    .asciz"Las cadenas son iguales\n"
diferent:   .asciz"Las cadenas son diferentes\n"

.section .bss
cad1: .space 100
cad2: .space 100

.section .text
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
.globl _start
_start:
            pushl $solicita1
            call printf
            pushl $cad1
            pushl $formato
            call scanf

            pushl $salto
            call printf

            pushl $solicita2
            call printf
            pushl $cad2
            pushl $formato
            call scanf

            leal cad1, %esi   # %esi=direccion de inicio de la cadena
cad1
            leal cad2, %edi   # %edi=direccion de inicio de la cadena
cad2
            movl $100, %ecx
            rep cmpsb         # compara las cadenas cad1 y cad2
            jne noigual
            pushl $iguales
            call printf
            jmp fin
noigual:    pushl $diferent
            call printf

fin:        movl $1, %eax
            movl $0, %ebx
            int $0x80
```

Luego de comparar dos cadenas también se pueden realizar saltos condicionales por mayor o menor. La reglas para determinar si una cadena de caracteres es mayor o menor que otra son las siguientes:

- El peso depende del orden alfabético. Las primeras letras del alfabeto tienen mayor peso, es decir la "a" es mayor que la "b" que a su vez es mayor que la "c" y así sucesivamente.
- Las minúsculas tienen mayor peso que las mayúsculas.

Por ejemplo la cadena "cola" es mayor que la cadena "hola" que a su vez es mayor que "Hola".

Estas reglas siguen los valores de codificación de caracteres del código ASCII.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Cuando se comparan cadenas de diferente tamaño la situación es más complicada, las reglas son las siguientes:

- La comparación se basa en el números de caracteres de la cadena más corta.
- Si la cadena corta es mayor que el mismo número de caracteres en la cadena larga, entonces la cadena corta es mayor.
- Si la cadena corta es menor que el mismo número de caracteres en la cadena larga, entonces la cadena corta es menor.
- Si la cadena corta es igual que el mismo número de caracteres en la cadena larga, entonces la cadena larga es mayor.

La instrucción scas

La instrucción scas permite buscar caracteres dentro de una cadena.

Instrucción	Descripción
scasb	Compara un byte en memoria con el valor en %al
scasw	Compara dos bytes en memoria con el valorde %ax
scasl	Compara cuatro bytes en memoria con el valor de %eax

El registro %edi debe tener la dirección de la cadena de caracteres sobre la cual va a actuar la instrucción scas. Cuando se ejecuta la instrucción el valor de %edi es incrementado o decrementado (dependiendo del valor de la bandera DF) la cantidad de bytes correspondientes a la instrucción. Esta instrucción modifica las banderas en EFLAGS por lo tanto se pueden usar los saltos condicionales para determinar el resultado de la comparación. Usualmente se utiliza con el prefijo repe para detectar un caracter diferente al ubicado en el registro o con repne para detectar un caracter igual.

Esta instrucción también es útil para determinar el tamaño de una cadena de caracteres.

El siguiente programa muestra un ejemplo de búsqueda de un caracter dentro de una cadena, puesto que la cadena es solicitada al usuario es necesario determinar su tamaño.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
# Programa que busca un caracter en una cadena de caracteres
.section .data
mensaje1:   .asciz"Este programa solicita una cadena de caracteres y un
caracter\n"
mensaje2:   .asciz"a buscar dentro de la cadena, indica si el caracter
fue encontrado\n"
solict1:    .asciz"Introduzca  la  cadena  de  caracteres  (maximo  100
caracteres): "
solict2:    .asciz"Introduzca el caracter a buscar: "
format1:    .asciz"%s"
format2:    .asciz"La cadena tiene %d caracteres\n"
noesta:     .asciz"El caracter %s no se encuentra en la cadena\n"
siesta:     .asciz"El caracter %s se encuentra en la cadena\n"
tam:        .long 0

.section .bss
cadena:     .space 100  # tamano maximo de la cadena de entrada
caracter:   .space 1    # tamano de un caracter

.section .text
.globl _start
_start:
      pushl $mensaje1   # mensaje inicial
      call printf
      pushl $mensaje2
      call printf
      addl $8, %esp
      pushl $solict1    # mensaje que solicita la cadena
      call printf
      addl $4, %esp
      leal cadena, %edx
      pushl %edx
      pushl $format1
      call scanf        # lectura de la cadena
      addl $8, %esp
      pushl $solict2    # mensaje que solicita el caracter a buscar
      call printf
      addl $4, %esp
      leal caracter, %edx
      pushl %edx
      pushl $format1
      call scanf        # lectura del caracter

      cld               # limpia la bandera CF, asi %ecx se decrementa
      leal cadena, %edi # %edi=dir de inicio de la cadena
      movl $100, %ecx   # tamano maximo de la cadena
      movb $0, %al            # caracter de fin de cadena
      repne scasb       # busca el fin de la cadena
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
      jne fin
      subl$100, %ecx    # calcula el numero de caracteres
      neg %ecx
      decl %ecx         # %ecx tiene el largo de la cadena
      movl %ecx, tam    # el largo de la cadena se guarda en tam

      pushl %ecx
      pushl $format2
      call printf       # muestra el numero de caracteres
      addl $8, %esp

      movl tam, %ecx          # vuelve a cargar %ecx con el largo
                              # de la cadena
      leal cadena, %edi       # %edi= inicio de la cadena
      leal caracter, %esi     # %esi=direccion del caracter
      lodsb                   # carga el caracter en %al
      repne scasb             # realiza la busqueda
      jne nohay

      pushl $caracter         # si el caracter esta en la cadena,
      pushl $siesta           # muestra el mensaje respectivo
      call printf
      jmp fin

nohay:
      pushl $caracter         # si el caracter no esta en la cadena,
      pushl $noesta           # muestra el mensaje que lo indica
      call printf

fin:  movl $1, %eax           # fin del programa
      movl $0, %ebx
      int $0x80
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Ejercicios

10.1.- Escriba un programa que realice la revisión de una contraseña, debe inidicar si la contraseña introducida es correcta.

10.2.- Escriba un programa que detecte la presencia del caracter "-" en una cadena de caracteres, el programa debe leer la cadena e indicar la posición de la primera aparición del caracter.

10.3.- Escriba un programa que cuente el número de ocurrencias del caracter "y" en una cadena.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Capítulo 11. Operaciones de punto flotante

La manera de representar fracciones se realiza usando números de punto flotante. Esta representación es similar a la notación científica, sólo que en las computadoras cada parte de la notación, coeficiente y exponente, se representan en binario. Debido a que existen infinitos números reales y que la máquina sólo puede representar un subconjunto de los mismos, se han establecido estándares para la representación de números fraccionarios los cuales son utilizados de manera universal para representar números de punto flotante. El estándar adoptado se denomina Estándar IEEE 754. IEEE son las siglas del Instituto de Ingeniería Eléctrica y Electrónica, por sus siglas en inglés. El estándar 754 define dos tamaños para números de punto flotante: 32 bits para precisión simple y 64 bits para precisión doble. El formato de precisión simple permite almacenar valores dentro del rango: $1.18x10^{-38}$ a $3.40x10^{38}$. El formato de precisión doble tiene un rango de : $2.23x10^{-308}$ a $1.79x10^{308}$.
La plataforma IA-32 provee ambos formatos y un formato de 80 bits llamado formato de punto flotante de precisión doble extendida. Este último es usado en los registros de la Unidad de Punto Flotante y tiene un rango de $3.37x10^{-4932}$ a $1.18x10^{4932}$.

Para definir valores de punto flotante se utilizan las directivas .float para definir un valor de precisión simple (32 bits) y .double para definir un valor de precisión doble (64 bits). Hasta el momento, el ensamblador gas no provee una directiva para definir valores de precisión doble extendida.

Las operaciones de punto flotante se ejecutan en la unidad de punto flotante (FPU- Floating Point Unit), la cual contiene registros adicionales para el manejo de la información en punto flotante. La FPU permite el procesamiento de funciones matemáticas complejas de manera rápida y eficiente.

Los registros de datos de la FPU se denominan %st(0), %st(1), ..., %st(7) y están relacionados entre si como una pila circular. El tope de la pila es %st(0). Cuando se cargan datos en la FPU, realmente se cargan en la pila comenzando en la posición %st(0), luego cada vez que se apila un nuevo valor se corren los valores presentes en la pila hacia los registros siguientes. Sólo hay ocho espacios

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

disponibles, si se apila un noveno valor éste reemplaza el contenido de %st(0), por ello se dice que la pila es circular.

La FPU tiene su propio registro de estado el cual indica las condiciones de operación de la unidad. Es un registro de 16 bits que contiene las banderas siguientes:

Bit de estado	Descripción
0	Bandera de excepción de operación inválida
1	Bandera de excepción de operando denormalizado
2	Bandera de excepción de división entre cero
3	Bandera de excepción de desbordamiento (valores muy grandes)
4	Bandera de excepción de desbordamiento (valores muy pequeños)
5	Bandera de excepción de precisión
6	Falla de pila
7	Resumen de estado de errores
8	Código de condición (bit 0)
9	Código de condición (bit 1)
10	Código de condición (bit 2)
11-13	Apuntador al tope de la pila
14	Código de condición (bit 3)
15	Bandera de PFU ocupada

Los seis primeros bits son banderas de excepción se activan cuando ocurre una excepción durante el procesamiento de instrucciones en la FPU, se mantienen activas hasta que un programa las desactive.
Este registro puede ser leído transfiriendo su contenido a una doble palabra de memoria o hacia el registro %ax. Esto se logra mediante el uso de la instrucción `fstsw`.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Asimismo la FPU provee un registro de control el cual controla su operación. Sus bits se especifican como:

Bit de control	Descripción
0	Máscara de excepción de operación inválida
1	Máscara de excepción de operando denormalizado
2	Máscara de excepción de división entre cero
3	Máscara de excepción de desbordamiento (valores muy grandes)
4	Máscara de excepción de desbordamiento (valores muy pequeños)
5	Máscara de excepción de precisión
6-7	Reservado
8-9	Control de precisión
10-11	Control de redondeo
12	Control de infinito
13-15	Reservado

Los primeros seis bits se usan para controlar qué banderas del registro de estado son usadas, cuando uno de estos bits está activo impide que la bandera correspondiente en el registro de estado pueda ser activada. Estos bits se encuentran activos por defecto, enmascarando así todas las excepciones. Los bits siguientes proveen la capacidad de escoger ciertas condiciones de funcionamiento de la FPU.

Los bits 8 y 9 definen la precisión con la cual se trabaja, las opciones son las siguientes:

- 00– precisión simple
- 01– no se usa
- 10– precisión doble
- 11– precisión doble extendida

Los bits 10 y 11 definen las opciones de redondeo, las cuales son:

- 00– redondeo hacia el más cercano
- 01– redondeo hacia abajo
- 10– redondeo hacia arriba
- 11– redondeo hacia cero

Por defecto la FPU trabaja con precisión doble extendida y con redondeo hacia el valor más cercano. El valor del registro de control, por defecto, es: 0x037F.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

				Redondeo		Precisión									
bit 15	bit 14	bit 13	bit 12	bit 11	bit 10	bit 9	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0	0	0	0	0	0	1	1	0	1	1	1	1	1	1	1
0				3				7				F			

Para modificar las condiciones hay que modificar el contenido del registro de control. Esto se hace con la instrucción `fldcw` la cual carga una doble palabra en el registro de control.

Ejemplo de un programa que cambia la precisión a simple, notése que es necesario cambiar los bits 8 y 9 a la combinación 00, por lo cual el nuevo valor del registro de control debe ser: 0x007F.

```
.section .data
simple:     .byte 0x7F, 0x00  # el orden se debe a que es little endian

.section .text
.globl _start
_start:
      fldcw simple      # modifica contenido del registro de control

      movl $1, %eax     # fin del programa
      movl $0, %ebx
      int $0x80
```

Instrucciones de movimiento de datos

fld (apila un dato punto flotante en la pila FPU)

Instrucción	Descripción	Ejemplo
flds etiqueta	Apila un valor de precisión simple (32 bits)	flds valor1 valor1 declarado como .float
fldl etiqueta	Apila un valor de precisión doble (64 bits)	fldl valor2 valor2 declarado como .double

Esta instrucción tiene varios formatos con valores predeterminados, ellos son los siguientes:

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Instrucción	Descripción
fld1	Apila +1.0
fldl2t	Apila log(base 2) 10
fldl2e	Apila log(base 2) e
fldpi	Apila el valor de pi (∏)
fldlg2	Apila log(base10) 2
fldln2	Apila log(base e) 2
fldz	Apila +0.0

fild (apila un entero)

Instrucción	Descripción	Ejemplo
filds etiqueta	Apila un valor entero de 32 bits	flds entero1 entero1 declarado como .long
fildl etiqueta	Apila un valor entero de 64 bits	fldl entero2 entero2 declarado como .quad

Cuando se realizan las instrucciones de apilar valores, cada valor adicional se va apilando en el tope st(0) y los demás valores se van corriendo hacia los registros restantes.

fst (movimiento de datos desde el tope de la pila)

Esta instrucción tiene dos variantes:
fst- lee el tope de la pila y mantiene su valor intacto
fstp- lee el tope de la pila y luego desapila ese valor, moviendo los contenidos de los demás registros hacia el tope.

fst (copia el tope de la pila)

Instrucción	Descripción	Ejemplo
fst %st(x)	Copia el dato del tope de la pila a otro registro FPU	fst %st(4) copia el contenido de %st(0) y lo guarda en %st(4)
fsts etiqueta	Copia el dato del tope de la pila a memoria (32 bits)	fsts variable1 copia el contenido de %st(0) y lo guarda en variable1 (32 bits)
fstl etiqueta	Copia el dato del tope de la pila a memoria (64 bits)	fstl variable2 copia el contenido de %st(0) y lo guarda en variable2 (64 bits)

fstp (lee el tope de la pila y desapila)

Instrucción	Descripción	Ejemplo
fstp %st(x)	Desapila el dato del tope de la pila y lo mueve a otro registro FPU	fstp %st(4) desapila contenido de %st(0) y lo mueve a %st(4)
fstps etiqueta	Desapila el dato del tope de la pila y lo mueve a memoria (32 bits)	fstps variable1 desapila el contenido de %st(0) y lo mueve a variable1 (32 bits)
fstpl etiqueta	Desapila el dato del tope de la pila y lo mueve a memoria (64 bits)	fstpl variable2 desapila el contenido de %st(0) y lo mueve a variable2 (64 bits)

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

fist (lee número entero del tope de la pila)

Instrucción	Descripción	Ejemplo
fists etiqueta	Desapila el valor presente en el tope de la pila y lo guarda en memoria (32 bits)	fists valor1 valor1 entero (32 bits)
fistl etiqueta	Desapila el valor presente en el tope de la pila y lo guarda en memoria (64 bits)	fistl valor2 valor2 entero (64 bits)

fxch (intercambio de valores entre el tope de la pila y otro registro)

Instrucción	Descripción	Ejemplo
fxch %st(x)	Intercambia el dato del tope de la pila y otro registro de la pila	fxch %st(1) intercambia el contenido de %st(0) y %st(1)

Instrucciones de control

finit (inicialización)

Esta instrucción inicializa la FPU. No modifica los contenidos de los registros de datos, coloca el registro de control en sus valores por defecto.

fstcw (copia el contenido del registro de control)

Con esta instrucción se crea una copia del contenido del registro de control en una posición de memoria.

fstsw (copia el contenido del registro de estado)

Con esta instrucción se crea una copia del contenido del registro de estado en una posición de memoria.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Instrucciones aritméticas básicas

fadd (suma)

Instrucción	Descripción	Ejemplo
fadds etiqueta	Suma valor de 32 bits al tope de la pila	fadds valor1
faddl etiqueta	Suma valor de 64 bits al tope de la pila	faddl valor2
fadd %st(x), %st(0)	Suma contenido de %st(x) con %st(0), guarda el resultado en %st(0)	fadd %st(4), %st(0)
fadd %st(0), %st(x)	Suma %st(0) con %st(x), guarda el resultado en %st(x)	fadd %st(0), %st(3)
faddp %st(0), %st(x)	Suma %st(0) con %st(x), guarda el resultado en %st(x) y desapila %st(0)	faddp %st(0), %st(4)
faddp	Suma %st(0) con %st(1), guarda el resultado en %st(1), desapila %st(0)	faddp
fiadds etiqueta	Suma valor entero de 32 bits a %st(0), guarda el resultado en %st(0)	fiadds valor1
fiaddl etiqueta	Suma valor entero de 64 bits a %st(0), guarda el resultado en %st(0)	fiaddl valor2

fsub (resta)

Instrucción	Descripción	Ejemplo
fsubs etiqueta	Resta tope de la pila menos valor de 32 bits	fsubs valor1
fsubl etiqueta	Resta tope de la pila menos valor de 64 bits	fsubl valor2
fadd %st(x), %st(0)	Resta el contenido de %st(x) menos %st(0), guarda el resultado en %st(0)	fsub %st(4), %st(0)
fsub %st(0), %st(x)	Resta %st(0) menos %st(x), guarda el resultado en %st(x)	fsub %st(0), %st(3)
fsubp %st(0), %st(x)	Resta %st(0) menos %st(x), guarda el resultado en %st(x) y desapila %st(0)	fsubp %st(0), %st(4)
fsubp	Resta %st(0) menos %st(1), guarda el resultado en %st(1), desapila %st(0)	fsubp
fisubs etiqueta	Resta %st(0) menos valor entero de 32 bits, guarda el resultado en %st(0)	fisubs valor1
fisubl etiqueta	Resta %st(0) menos valor entero de 64 bits, guarda el resultado en %st(0)	fisubl valor2

La instrucción **fsubr** realiza la resta inversa, es decir toma en cuenta los operandos de manera inversa a la tabla presentada para fsub. Resta el segundo operando del primero y guarda el resultado en el segundo lo cual coincide con fsub.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

fmul (multiplicación)

Instrucción	Descripción	Ejemplo
fmuls etiqueta	Multiplica tope de la pila por valor de 32 bits	fmuls valor1
fmull etiqueta	Multiplica tope de la pila por valor de 64 bits	fmull valor2
fmul %st(x), %st(0)	Multiplica el contenido de %st(x) por %st(0), guarda el resultado en %st(0)	fmul %st(4), %st(0)
fmul %st(0), %st(x)	Multiplica %st(0) por %st(x), guarda el resultado en %st(x)	fmul %st(0), %st(3)
fmulp %st(0), %st(x)	Multiplica %st(0) por %st(x), guarda el resultado en %st(x) y desapila %st(0)	fmulp %st(0), %st(4)
fmulp	Multiplica %st(0) por %st(1), guarda el resultado en %st(1), desapila %st(0)	fmulp
fimuls etiqueta	Multiplica %st(0) por valor entero de 32 bits, guarda el resultado en %st(0)	fimuls valor1
fimull etiqueta	Multiplica %st(0) por valor entero de 64 bits, guarda el resultado en %st(0)	fimull valor2

fdiv (división)

Instrucción	Descripción	Ejemplo
fdivs etiqueta	Divide el tope de la pila entre valor de 32 bits	fdivs valor1
fdivl etiqueta	Divide el tope de la pila entre valor de 64 bits	fdivl valor2
divl %st(x), %st(0)	Divide el contenido de %st(x) entre %st(0), guarda el resultado en %st(0)	fdiv %st(4), %st(0)
fdiv %st(0), %st(x)	Divide %st(0) entre %st(x), guarda el resultado en %st(x)	fdiv %st(0), %st(3)
fdivp %st(0), %st(x)	Divide %st(0) entre %st(x), guarda el resultado en %st(x) y desapila %st(0)	fdivp %st(0), %st(4)
fdivp	Divide %st(0) entre %st(1), guarda el resultado en %st(1), desapila %st(0)	fdivp
fidivs etiqueta	Divide %st(0) entre valor entero de 32 bits, guarda el resultado en %st(0)	fidivs valor1
fidivl etiqueta	Divide %st(0) entre valor entero de 64 bits, guarda el resultado en %st(0)	fidivl valor2

La instrucción **fdivr** realiza la división inversa, es decir toma en cuenta los operandos de manera inversa a la tabla presentada para fdiv. Resta el segundo operando del primero y guarda el resultado en el segundo.

Algunas funciones matemáticas avanzadas

Instrucción	Descripción
f2xm1	calcula 2 elevado al contenido de %st(0) menos 1
fabs	calcula el valor absoluto de %st(0)
fchs	cambia el signo del valor en %st(0)
fcos	calcula el coseno del valor en %st(0) en radianes, guarda el resultado en %st(0)
fprem1	calcula el resto de dividir el valor en %st(0) entre el valor en %st(1)
frndint	redondea el valor en %st(0) al entero más cercano
fscale	calcula %st(0) elevado al valor en %st(1)
fsin	calcula el seno del valor en %st(0) en radianes, guarda el resultado en %st(0)
fsqrt	calcula la raiz cuadrada del valor en %st(0)

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

La instrucción **frndint** redondea el valor en %st(0) al entero más cercano usando el método de redondeo que se encuentre en el registro de control.

La instrucción **fprem1** calcula el resto de la división de manera iterativa. Se puede saber cuando la operación culmina revisando el bit 2 del registro de estado, cuando ese bit se encuentra en cero, las iteraciones están completas. Este proceso se debe realizar dentro de un ciclo, tal como se ejemplifica en el siguiente programa:

```
.section .data
valor1:     .float 25.45
valor2:     .float 2.24

.section .bss
.space resultado, 4

.section .text
.globl _start
_start:
            finit             # inicializacion de FPU
            flds valor2       # carga valor2
            flds valor1       # carga valor1
ciclo:      fprem1            # calcula resto de div valor1/valor2
            fstsw %ax         # copia registro de estado en %ax
            testb $4, %ah     # chequea bit 2 del registro de estado
            jnz ciclo         # mientras diferente de 0 vuelva a ciclo
            fsts resultado    # actualiza resultado con el resto
            movl $1, %eax     # fin del programa
            movl $0, %ebx
            int $0x80
```

Las operaciones **fcos** y **fsin** calculan el coseno y el seno respectivamente. Asumen que el valor en %st(0) está expresado en radianes. La fórmula para convertir grados en radianes es la siguiente:

radianes= (grados*pi)/180

Esto se puede realizar con la siguiente secuencia de instrucciones:

```
fsts grados             # carga la cantidad de grados
fidivs val180           # divide entre 180 (valor guardado en memoria)
fldpi                   # carga pi en %st(0), ahora %st(1)=180
fmul %st(1), %st(0)     # multiplica grados/180 por pi
                        # el resultado queda en %st(0)
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Instrucciones de comparación

En los primeros modelos de máquinas Intel era necesario leer el registro de estado y actualizar, por medio de un programa, el contenido del registro de banderas (EFLAGS) para poder realizar saltos de acuerdo a las comparaciones realizadas. A partir del procesador Pentium Pro se introduce un conjunto de instrucciones que actualizan automáticamente el registro de banderas. Estas son las instrucciones presentadas a continuación:

Instrucción	Descripción	Ejemplo
fcomi %st(x)	compara %st(0) con %st(x)	fcomi %st(4)
fcomip %st(x)	compara %st(0) con %st(x) y desapila	fcomip %st(3)
fucomi %st(0), %st(x)	revisa si los valores en %st(0) y en %st(x) son válidos	fucomi %st(0), %st(2)
fucomip %st(0), %st(x)	revisa si los valores en %st(0) y en %st(x) son válidos y desapila	fucomip %st(0), %st(3)

La comparación actualiza las banderas según la siguiente tabla:

Condición	ZF (Bandera de cero)	PF (Bandera de paridad)	CF (Bandera de acarreo)
%st(0)> %st(x)	0	0	0
%st(0)< %st(x)	0	0	1
%st(0) = %st(x)	1	0	0

Ejemplo de un programa que calcula una expresión aritmética con cuatro valores fraccionarios declarados en la sección de datos.

```
# Programa que calcula (a+b)*(c-d)
.section .data
mensaje:    .asciz"Programa que calcula la expresion (a+b)*(c-d)\n"
salto:      .asciz"\n"
a:          .float 2.65
b:          .float 4.85
c:          .float 25.234
d:          .float 12.82
formato:    .asciz"resultado: %f\n"

.section .text
.globl _start
_start:
            pushl $mensaje    # mensaje inicial
            call printf
            pushl $salto
            call printf
            addl $8, %esp

            finit                   # inicializacion de FPU
            flds c                  # carga c en %st(0)
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
        fsubs d                 # %st(0)=%st(0)-d   (c-d)
        flds a                  # carga a en %st(0), %st(1)=c-d
        fadds b                 # %st(0)=%st(0)+b   (a+b)
        fmul %st(1), %st(0)     # %st(0)=%st(0)*%st(1)

        subl $8, %esp           # reserva 8 bytes en la pila
        fstpl (%esp)            # coloca resultado en la pila
        pushl $formato          # apila el formato de salida
        call printf             # muestra resultado por pantalla

        movl $1, %eax           # fin del programa
        xorl %ebx, %ebx
        int $0x80
```

Para mostrar el resultado por pantalla se puede usar la función printf pero hay que tener en cuenta que el formato de salida debe ser %f para mostrar un número fraccionario. Asimismo cuando se desapila el resultado se debe usar la instrucción `fstpl` la cual desapila el valor de 8 bytes y lo coloca en memoria. Para hacer esto primero se resta 8 al apuntador %esp (para reservar 8 bytes) y luego con la instrucción `fstpl (%esp)` se "apila" el valor como parámetro para la llamada a la función printf.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Ejercicios

11.1- Escriba un programa que calcule la nota definitiva tomando en cuenta 6 notas con los siguientes porcentajes:

- Nota 1: 20%
- Nota 2: 15%
- Nota 3: 25%
- Nota 4: 10%
- Nota 5: 25%
- Nota 6: 5%

El programa debe mostrar la nota final con dos decimales y la nota definitiva redondeada, el redondeo debe ser hacia arriba.

11.2.- Escriba un programa que permita convertir grados Centígrados a Fahrenheit.

11.3.- Escriba un programa que acepte como entradas 10 valores de temperatura en grados Centígrados y muestre su promedio.

11.4.- Escriba un programa que solicite el valor del diámetro de un círculo y calcule su área.

11.5.- Escriba un programa que permita realizar la conversión de kilómetros a millas y de millas a kilómetros.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Capítulo 12. Manejo de archivos

En los capitulos anteriores se han descrito mecanismos que permiten recibir datos del usuario mediante los dispositivos de E/S, procesarlos de manera sofisticada mediante el uso de las diferentes estructuras de control, realizar cálculos complejos sobre dichos datos utilizando bien sea registros enteros o de punto flotante, y mostrar los resultados obtenidos haciendo uso de diferentes funciones e instrucciones.

Sin embargo, ¿qué ocurre una vez que el programa finaliza su ejecución, o cuando es retirada la energía eléctrica que hace funcionar al computador?. La respuesta es simple: los datos proporcionados por el usuario, los resultados calculados y cualquier otra información importante se pierde. Esto se debe a que tanto el código como los datos de un programa en ejecución se mantienen en un tipo de memoria volátil (la RAM), la cual pierde cualquier requiere energía eléctrica para mantener la información almacenada en ella.

Existen innumerables ocasiones en que es necesario que la información almacenada se mantenga intacta aún cuando el computador se encuentre apagado. El ejemplo más claro de esto es una base de datos de cualquier tipo. Otros ejemplos de la vida cotidiana son el correo electrónico, imágenes y música digitalizada, archivos de texto, documentos, hojas de cálculo, presentaciones, archivos de configuración de los programas, e inclusive los programas en si mismos.

Un archivo no es más que un conjunto de datos que son almacenados en un dispositivo de almacenamiento permanente, tal como el disco duro, un diskette o una memoria portátil USB. El manejo de archivos involucra básicamente tres acciones por parte del usuario: apertura, manipulación y cierre. La manipulación del archivo consiste en realizar operaciones de lectura y/o escritura sobre el mismo.

Por ejemplo, imagine que posee un archivo de texto en el cual escribe regularmente las actividades que tiene que realizar. Cuando utiliza un editor de texto dentro de un entorno gráfico para modificar este archivo, insconscientemente está realizando las tres acciones anteriormente descritas:

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Apertura: Al hacer doble clic en el archivo.

Lectura: Al ejecutarse el editor se cargan el contenido del archivo: las actividades anotadas previamente.

Escritura: Al seleccionar la opción que permite guardar las modificaciones realizadas al archivo.

Cierre: Al finalizar la ejecución del editor.

Si bien esta explicación puede parecer trivial, a lo largo de este capítulo se mostrará que el manejo de archivos desde el punto de vista del programador siempre está compuesto por estas tres acciones básicas, sin importar que el lenguaje utilizado sea de alto o bajo nivel.

Para realizar una operación de lectura o escritura sobre un archivo, es necesario haber abierto el mismo previamente. Adicionalmente existen otras operaciones básicas que pueden ser realizadas sobre un archivo, las cuales no necesariamente involucran la apertura del mismo. Entre ellas se encuentran la creación de un nuevo archivo, el cambio de atributos (nombre, visibilidad, permisos de acceso, etc.) de uno existente, así como la eliminación del mismo.

Tipos de archivo

Un archivo puede ser clasificado de acuerdo a la forma en la que es almacenada la información dentro del mismo. En este sentido, existen dos tipos de archivo: archivos de texto y archivos binarios. Esta distinción es sumamente importante, ya que es necesario conocer el tipo de archivo que se está manejando para manipularlo adecuadamente.

Archivos de texto

Un archivo de texto es una secuencia de caracteres (generalmente imprimibles), en el cual la información es almacenada de forma que pueda ser comprendida a simple vista por el usuario. Por esta razón, también son llamados archivos de texto plano. Ejemplos de este tipo de archivo son un documentos HTML, código fuente de programas en lenguajes de alto y bajo nivel, documentos de texto sin formato (txt) y los archivos de configuración de algunas aplicaciones.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Archivos binarios

Un archivo binario es una secuencia de bytes, en el cual la información es almacenada en la forma más elemental que puede manejar un computador: ceros y unos. Esto ocasiona que la misma no pueda ser comprendida a simple vista por el usuario. Ejemplos de este tipo de archivo son imágenes y música digitalizada, programas en lenguaje de máquina, documentos con formato (doc, pdf) y los archivos de configuración de algunas aplicaciones.

Observe que los archivos de configuración de las aplicaciones han sido mencionados como ejemplo para ambos tipos de archivo. Esto lo que nos indica es que en el fondo cualquier archivo puede ser almacenado en una u otra forma, la decisión la toma el programador que diseña la aplicación que hará uso del archivo en cuestión, evalúando previamente que tipo de archivo se adapta mejor a sus necesidades. Por ejemplo, si el archivo de configuración es binario será necesario crear una herramienta que permita al usuario final modificarlo, mientras que si es un archivo de texto plano, el mismo podrá ser modificado mediante la utilización de un editor de texto convencional.

Tipos de acceso

Otro criterio utilizado para clasificar los diferentes tipos de archivo, se refiere a la forma en la que se accesa a la información almacenada en los mismos. Desde este punto de vista, un archivo puede ser de acceso secuencial o de acceso aleatorio.

Acceso secuencial

En un archivo de acceso secuencial, para tener acceso al dato almacenado en la posición N, se deben haber accesado los datos almacenados en las N-1 posiciones previas, en un orden secuencial.

Acceso aleatorio

Un archivo de acceso aleatorio permite acceder a cualquier parte del mismo en cualquier momento, como si se tratara de un arreglo almacenado en la memoria. Por esta razón este tipo de acceso es conocido también como acceso

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

directo, ya que permite ir directamente a los datos de interés sin tener que recorrer los que lo preceden dentro del archivo.

Como siempre, la decisión de utilizar un determinado tipo de acceso recae en el programador. El criterio de mayor peso a la hora de elegir uno u otro es el tiempo de acceso. Si existen pocos datos almacenados en un archivo, la diferencia entre los tiempos de acceso de forma secuencial y directa puede ser imperceptible para el usuario. Sin embargo, la diferencia comienza a apreciarse a medida que se manejan archivos con grandes cantidades de información.

Sin embargo existen casos en los que el acceso secuencial de un archivo es recomendable. El caso más representativo es el manejo de una nómina de empleados, clientes o estudiantes, en la cual se se deben procesar todos (o la mayoría) de los registros.

Funciones de alto nivel

En la actualidad la mayor parte de los lenguajes de alto nivel proporcionan un conjunto de funciones para permitir al programador realizar el manejo de archivos de una forma simple y poderosa a la vez. En particular el lenguaje C es bastante rico en este sentido, ya que pose una gran variedad de funciones orientadas a la manipulación de archivos. La siguiente es una lista de las más relevantes, junto con una breve descripción de la tarea que realiza cada una de ellas:

Función	Descripción
fopen	Abre un archivo
fclose	Cierra un archivo
fread	Lee bloques de bytes desde un archivo
fwrite	Escribe bloques de bytes a un archivo
fgets	Lee una cadena de caracteres desde un archivo
fputs	Escribe una cadena de caracters a un archivo
fscanf	Realiza lectura formateada desde un archivo
fprintf	Realiza escritura formateada a un archivo
fseek	Cambia la posición del apuntador de lectura y escritura
feof	Determina si se alcanzó el final del archivo
ferror	Determina si ocurrió un error en la última operación sobre el archivo
fflush	Vacía el contenido del buffer al archivo
remove	Elimina un archivo o directorio

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

A continuación se presenta una descripción más detallada de cada una de estas llamadas al sistema:

fopen

La función *fopen*, permite abrir un archivo existente. El prototipo de la misma es el siguiente:

*FILE *fopen(const char *camino, const char *modo);*

En donde *camino* es una cadena que contiene la ruta (absoluta o relativa) del archivo que se desea abrir y *modo* es una cadena que describe la forma en la que se desea abrir el archivo. La llamada a *fopen devuelve* el descriptor asociado al archivo abierto. En caso de no poderse abrir el archivo, la llamada devuelve NULL.
La cadena modo debe comenzar por una de las siguientes opciones:

Opción	Descripción
r	Abre el archivo solo para lectura. El apuntador se posiciona al comienzo.
r+	Abre el archivo para lectura y escritura. El apuntador se posiciona al comienzo.
w	Trunca el fichero a una longitud de cero bytes o crea uno nuevo si no existe, y lo abre para para escritura. El apuntador se posiciona al principio.
w+	Trunca el fichero a una longitud de cero bytes o crea uno nuevo si no existe, y lo abre para para lectura y escritura. El apuntador se posiciona al principio.
a	Abre el archivo para escritura o crea uno nuevo si no existe. El apuntador se posiciona al final.
a+	Abre el archivo para lectura y escritura o crea uno nuevo si no existe. El apuntador se posiciona al final

Adicionalmente la cadena modo puede incluir el carácter ‘b’ al final de cualquiera de las opciones mencionadas (rb, r+b, etc.) o entre los caracteres de aquellas que ocupan más de un carácter (rb+, ab+, etc.) para indicar que el archivo que se desea abrir es binario (la ausencia de la ‘b’ indica que el archivo es de texto plano). Sin embargo esta opción es ignorada en Linux (y todos los sistemas que siguen el estandar POSIX) ya que el mismo trata de la misma forma ambos tipos de archivo. Sin embargo es común colocar dicha opción para mantener la compatibilidad con otros sistemas.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

fclose

La función *fclose*, permite cerrar un archivo, es decir, elimina la relación existente entre el descriptor y el archivo, de forma que el primero deje de hacer referencia al último (note por lo tanto que el cerrar el descriptor no necesariamente implica la liberación de los recursos asociados al archivo, ya que puede existir otro descriptor haciendo referencia al mismo). El prototipo de la misma es el siguiente:

*int fclose(FILE *flujo);*

En donde *flujo* es el descriptor de archivo que se desea cerrar. La llamada a *fclose* devuelve 0 en caso de éxito y la constante EOF en caso de ocurrir un error.

fread

La función *fread*, permite leer un flujo de datos binarios desde un archivo. El prototipo de la misma es el siguiente:

*size_t fread(void *ptr, size_t tam, size_t nmiemb, FILE *flujo);*

En donde *prt* es un apuntador a la zona de memoria en la que se almacenarán los datos leídos, *tam* es la cantidad de elementos a leer, *nmiemb* es el tamaño en bytes de cada uno de dichos elementos y *flujo* es el descriptor de archivo desde el cual se desea leer. La llamada a *fread* devuelve el número de elementos leídos correctamente.

fwrite

La función *fwrite*, permite escribir un flujo de datos binarios en un archivo. El prototipo de la misma es el siguiente:

*size_t fwrite(const void *ptr, size_t tam, size_t nmiemb, FILE *flujo);*

En donde *prt* es un apuntador a la zona de memoria en la que se encuentran los datos a escribir, *tam* es la cantidad de elementos a escribir, *nmiemb* es el tamaño en bytes de cada uno de dichos elementos y *flujo* es el descriptor de archivo desde el cual se desea escribir. La llamada a *fwrite* devuelve el número de elementos escritos correctamente.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

fgets

La función *fgets*, permite leer una cadena de caracteres desde un archivo. El prototipo de la misma es el siguiente:

*char *fgets(char *s, int tam, FILE *flujo);*

En donde *s* es un apuntador a la zona de memoria en la que se almacenarán los datos leídos, *tam* es la cantidad de caracteres a leer, y *flujo* es el descriptor de archivo desde el cual se desea leer. La llamada a *fgets* devuelve *s* en caso de éxito y NULL en caso de error.

fputs

La función *fputs*, permite escribir una cadena de caracteres en un archivo. El prototipo de la misma es el siguiente:

*int fputs(const char *s, FILE *flujo);*

En donde *s* es un apuntador a la zona de memoria en la que se encuentran los datos a escribir, *tam* es la cantidad de caracteres a escribir, y *flujo* es el descriptor de archivo en el cual se desea escribir. La llamada a *fputs* devuelve un número no negativo en caso de éxito y la constante EOF en caso de error.

fscanf

La función *fscanf*, permite realizar lectura formateda desde un archivo. El prototipo de la misma es el siguiente:

*int fscanf(FILE *flujo, const char *formato, ...);*

En donde *flujo* es el descriptor de archivo desde el cual se desea leer, *formato* es la cadena que define el formato de los datos a leer y los parámetros adicionales son apuntadores a las zonas de memoria en las que serán almacenados los datos reconocidos. Esta función se comporta de la misma manera que la función *scanf* pero toma la entrada desde el archivo especificado en lugar de hacerlo desde la entrada estándar (el teclado). La llamada a *fscanf* devuelve la cantidad de elementos reconocidos y asignados a las correspondientes variables.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

fprintf

La función *fprintf*, permite realizar escritura formateda en un archivo. El prototipo de la misma es el siguiente:

*int fprintf(FILE *flujo, const char *formato, ...);*

En donde *flujo* es el descriptor de archivo en el cual se desea escribir, *formato* es la cadena que define el formato de los datos a escribir y los parámetros adicionales son las variables con las cuales se rellenará la cadena de formato. Esta función se comporta de la misma manera que la función *printf* pero escribe la salida en el archivo especificado en lugar de hacerlo en la salida estándar (la pantalla). La llamada a *fprintf* devuelve la cantidad de caracteres escritos.

fseek

La función *fseek*, permite cambiar la ubicación del apuntador de lectura/escritura de un descriptor de archivo. El prototipo de la misma es el siguiente:

*int fseek(FILE *flujo, long despIto, int origen);*

En donde *flujo* es el descriptor del archivo al cual se desea cambiar la posición del apuntador, *desplto* es la cantidad de bytes que se quiere desplazar el mismo y *origen* es la posición a partir de la cual se hará efectivo dicho desplazamiento. La llamada a *fseek* devuelve 0 en caso de tener éxito y -1 en caso de ocurrir un error.

feof

La función *feof*, permite determinar si se ha alcanzado el final de un archivo abierto. El prototipo de la misma es el siguiente:

*int feof(FILE *flujo);*

En donde *flujo* es el descriptor del archivo en cuestión. La llamada a *feof* devuelve un valor distinto de cero (verdadero) en caso que el apuntador de lectura y escritura se encuentre al final del archivo y 0 en caso contrario.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

ferror

La función *ferror*, permite determinar si ha ocurrido un error en al realizar alguna operación sobre un descriptor de archivo. El prototipo de la misma es el siguiente:

*int ferror(FILE *flujo);*

En donde *flujo* es el descriptor del archivo en cuestión. La llamada a *ferror* devuelve un valor distinto de cero si ha ocurrido un error y 0 en caso contrario.

fflush

La función *fflush*, fuerza a que se vacíe el contenido del buffer asociado a un descriptor de archivo. El prototipo de la misma es el siguiente:

*int fflush(FILE *flujo);*

En donde *flujo* es el descriptor del archivo en cuestión. La llamada a *fflush* devuelve cero en caso de concluir con éxito. En caso de ocurrir un error retorna la constante EOF.

remove

La función *remove*, permite eliminar una entrada del sistema de archivos, y de ser la única referencia al archivo en cuestión, eliminarlo. El prototipo de la misma es el siguiente:

*int remove(const char *pathname);*

En donde *pathname* es una cadena que contiene la ruta (absoluta o relativa) del archivo que se desea eliminar. La llamada a *remove* devuelve 0 en caso de tener éxito y -1 en caso de ocurrir un error.

Para ilustrar la utilización de las funciones anteriormente descritas, se presenta un programa que sirve como agenda telefónica con una cantidad de opciones bastante limitada (solo permite insertar y buscar un contacto en base a la posición).

Programa en C:

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
#include <stdio.h>

int main()
 {
  int opcion = 0;
  FILE* archivo;

  printf("1 Insertar contacto\n");
  printf("2 Buscar contacto\n");

  while(opcion!=1 && opcion !=2)
   {
    printf("Su opción: ");
    scanf("%d", &opcion);
   }

  char nombre[80];
  int telefono;
  int posicion;

  if(opcion==1)
   {
    // Abrir el archivo de texto para escritura al final
    archivo = fopen("agenda.txt","a");

    // Si no se pudo abrir el archivo
    if(archivo==NULL)
     {
      printf("Error al abrir el archivo\n");
      return -1;
     }
    else
     {
      printf("Nombre del contacto: ");
      scanf("%s", nombre);
      printf("Teléfono del contacto: ");
      scanf("%d", &telefono);
      fprintf(archivo, "%s | %d\n", nombre, telefono);
     }
   }

  if(opcion==2)
   {
    // Abrir el archivo de texto para lectura desde el comienzo
    archivo = fopen("agenda.txt","r");

    // Si no se pudo abrir el archivo
    if(archivo==NULL)
     {
      printf("Error al abrir el archivo\n");
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
      return -1;
     }
    else
     {
      printf("Posición: ");
      scanf("%d", &posicion);

      int i=0;
      while(i!=posicion && i!=-1)
       {
        if(feof(archivo)) i=-1;
        else
         {
          fscanf(archivo, "%s | %d\n", nombre, &telefono);
          i++;
         }
       }

      if(i==-1)
       {
        printf("La posición indicada está vacía\n");
       }
      else
       {
        printf("Nombre: %s\n", nombre);
        printf("Telefono: %d\n", telefono);
       }
     }

    // Cerrar el archivo
    fclose(archivo);
    return 0;
   }
  }
```

Programa en ensamblador:

```
.section .rodata
cad1:   .asciz "1 Insertar contacto\n2 Buscar contacto\n"
cad2:   .asciz "Su opción: "
cad3:   .asciz "Ocurrió un error al intentar abrir el archivo\n"
cad4:   .asciz "Nombre del contacto: "
cad5:   .asciz "Teléfono del contacto: "
cad6:   .asciz "Posición: "
cad7:   .asciz "Nombre: %s\nTelefono: %d\n"
cad8:   .asciz "La posición indicada está vacía\n"
file:   .asciz "agenda.txt"
format1:.asciz "%d"
format2:.asciz "%s"
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
format3:.asciz "%s | %d\n"
modo1:  .asciz "a"
modo2:  .asciz "r"


.section .bss
nombre: .space 80
tlf:    .space 4
opcion: .space 4
pos:    .space 4
desp:   .space 4
fd: .space 4


.section .text
.globl _start
_start:
    pushl $cad1         # Muestra las opciones
    call printf
    addl $4, %esp

while:  pushl $cad2     # Pide una opción
    call printf
    addl $4, %esp

    pushl $opcion       # Lee la respuesta del usuario
    pushl $format1
    call scanf
    addl $8, %esp

    cmpl $1, opcion     # Si es 1 salta a insertar
    je insertar
    cmpl $2, opcion     # Si es 2 salta a buscar
    je buscar
    jmp while           # En otro caso pregunta de nuevo


    # Abre el archivo para escritura al final
insertar:

    pushl $modo1        # Modo: ab
    pushl $file         # Archivo: agenda2.dat
    call fopen          # Llama a fopen
    addl $8, %esp

    mov %eax, fd        # Apuntador al archivo abierto

    cmpl $0, fd         # Si fd==NULL (0) ocurrió un error
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    je error

    pushl $cad4         # Pregunta por el nombre del contacto
    call printf
    addl $4, %esp

    pushl $nombre       # Lee la respuesta del usuario
    pushl $format2
    call scanf
    addl $8, %esp

    pushl $cad5         # Pregunta por el telefono del contacto
    call printf
    addl $4, %esp

    pushl $tlf          # Lee la respuesta del usuario
    pushl $format1
    call scanf
    addl $8, %esp

    # Inserta el contacto en el archivo

    pushl tlf           # Telefono
    pushl $nombre       # &Nombre
    pushl $format3      # Cadena de formato
    pushl fd            # Apuntador al archivo abierto
    call fprintf        # Llama a fprintf
    addl $16, %esp

    jmp fin             # Salta al final del programa


    # Abre el archivo para lectura desde el comienzo
buscar:
    pushl $modo2        # Modo: rb
    pushl $file         # Archivo: agenda2.dat
    call fopen          # Llama a fopen
    addl $8, %esp

    mov %eax, fd        # Apuntador al archivo abierto

    cmpl $0, fd         # Si fd==NULL (0) ocurrió un error
    je error

    pushl $cad6         # Pide la posición del contacto
    call printf
    addl $4, %esp
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    pushl $pos          # Lee la respuesta del usuario
    pushl $format1
    call scanf
    addl $8, %esp

    xorl %esi, %esi     # i=0
rep:
    cmpl %esi, pos      # Si i=pos
    je mostrar          # Salta a mostrar

    # Si se alcanzó el final del archivo la posición no existe
    pushl fd            # Apuntador al archivo abierto
    call feof           # Llama a feof
    addl $4, %esp
    cmpl $0, %eax
    jne nopos

    # Lee el contacto desde el archivo
    pushl $tlf          # &Telefono
    pushl $nombre       # &Nombre
    pushl $format3      # Cadena de formato
    pushl fd            # Apuntador al archivo abierto
    call fscanf         # Llama a fscanf
    addl $16, %esp

    incl %esi           # i++
    jmp rep             # Repite la búsqueda

mostrar:
    pushl tlf           # Muestra la información del contacto
    pushl $nombre
    pushl $cad7
    call printf
    addl $12, %esp

    jmp fin             # Salta al final del programa


    # La posición solicitada no existe
nopos:
    pushl $cad8         # Informa al usuario del problema
    call printf
    addl $4, %esp
    jmp fin             # Salta al final del programa


    # Ocurrió un error al abrir el archivo
error:
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    pushl $cad3         # Informa al usuario del error
    call printf
    addl $4, %esp
    jmp fin2

fin:
    # Cierra el archivo si estaba abierto
    pushl fd            # Descriptor del archivo a cerrar
    call fclose         # Llama a fclose
    addl $4, %esp

fin2:
    # Termina el programa
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

Observe que esta versión de la agenda realiza un acceso secuencial sobre un archivo de texto plano para almacenar y recuperar la información de los contactos. Si examina el contenido del archivo *agenda.txt* en un editor de texto encontrá que la información contenida en el mismo puede ser leida por un ser humano. Observe también que para ubicar un contacto es necesario recorrer todos los que lo preceden dentro del archivo.

A continuación se presenta una segunda versión de la agenda, que contempla exactamente las mismas opciones, pero utiliza el acceso directo para almacenar y recuperar la información de los contactos en un archivo binario.

Programa en C:

```
#include <stdio.h>

int main()
 {
  int opcion = 0;
  FILE* archivo;

  printf("1 Insertar contacto\n");
  printf("2 Buscar contacto\n");

  while(opcion!=1 && opcion !=2)
   {
    printf("Su opción: ");
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
  scanf("%d", &opcion);
 }

char nombre[80];
int telefono;
int posicion;

if(opcion==1)
 {
  // Abrir el archivo binario para escritura al final
  archivo = fopen("agenda2.dat","ab");

  // Si no se pudo abrir el archivo
  if(archivo==NULL)
   {
    printf("Error al abrir el archivo\n");
    return -1;
   }
  else
   {
    printf("Nombre del contacto: ");
    scanf("%s", nombre);
    printf("Teléfono del contacto: ");
    scanf("%d", &telefono);
    fwrite(nombre, 80, 1, archivo);
    fwrite(&telefono, 4, 1, archivo);
   }
 }

if(opcion==2)
 {
  // Abrir el archivo binario para lectura desde el comienzo
  archivo = fopen("agenda2.dat","rb");

  // Si no se pudo abrir el archivo
  if(archivo==NULL)
   {
    printf("Error al abrir el archivo\n");
    return -1;
   }
  else
   {
    printf("Posición: ");
    scanf("%d", &posicion);

    fseek(archivo,84*(posicion-1),SEEK_SET);
    fread(nombre, 80, 1, archivo);
    fread(&telefono, 4, 1, archivo);

    if(feof(archivo))
     {
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
       printf("La posición indicada está vacía\n");
      }
     else
      {
       printf("Nombre: %s\n", nombre);
       printf("Telefono: %d\n", telefono);
      }
    }

   // Cerrar el archivo
   fclose(archivo);
   return 0;
  }
}
```

Programa en ensamblador:

```
.section .rodata
cad1:   .asciz "1 Insertar contacto\n2 Buscar contacto\n"
cad2:   .asciz "Su opción: "
cad3:   .asciz "Ocurrió un error al intentar abrir el archivo\n"
cad4:   .asciz "Nombre del contacto: "
cad5:   .asciz "Teléfono del contacto: "
cad6:   .asciz "Posición: "
cad7:   .asciz "Nombre: %s\nTelefono: %d\n"
cad8:   .asciz "La posición indicada está vacía\n"
file:   .asciz "agenda2.dat"
format1:.asciz "%d"
format2:.asciz "%s"
modo1:  .asciz "ab"
modo2:  .asciz "rb"

.section .bss
nombre: .space 80
tlf:    .space 4
opcion: .space 4
pos:    .space 4
desp:   .space 4
fd: .space 4

.section .text
.globl _start
_start:
    pushl $cad1         # Muestra las opciones
    call printf
    addl $4, %esp
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
while:  pushl $cad2     # Pide una opción
    call printf
    addl $4, %esp

    pushl $opcion       # Lee la respuesta del usuario
    pushl $format1
    call scanf
    addl $8, %esp

    cmpl $1, opcion     # Si es 1 salta a insertar
    je insertar
    cmpl $2, opcion     # Si es 2 salta a buscar
    je buscar
    jmp while           # En otro caso pregunta de nuevo


    # Abre el archivo para escritura al final
insertar:

    pushl $modo1        # Modo: ab
    pushl $file         # Archivo: agenda2.dat
    call fopen          # Llama a fopen
    addl $8, %esp

    mov %eax, fd        # Apuntador al archivo abierto

    cmpl $0, fd         # Si fd==NULL (0) ocurrió un error
    je error

    pushl $cad4         # Pregunta por el nombre del contacto
    call printf
    addl $4, %esp

    pushl $nombre       # Lee la respuesta del usuario
    pushl $format2
    call scanf
    addl $8, %esp

    pushl $cad5         # Pregunta por el telefono del contacto
    call printf
    addl $4, %esp

    pushl $tlf          # Lee la respuesta del usuario
    pushl $format1
    call scanf
    addl $8, %esp
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    # Inserta el contacto en el archivo

    pushl fd            # Apuntador al archivo abierto
    pushl $1            # Cantidad de elementos a escribir
    pushl $80           # Tamaño del elemento
    pushl $nombre       # Dirección fuente de los datos
    call fwrite         # Llama a fwrite
    addl $16, %esp

    pushl fd            # Apuntador al archivo abierto
    pushl $1            # Cantidad de elementos a escribir
    pushl $4            # Tamaño del elemento
    pushl $tlf          # Dirección fuente de los datos
    call fwrite         # Llama a fwrite
    addl $16, %esp

    jmp fin             # Salta al final del programa


    # Abre el archivo para lectura desde el comienzo
buscar:
    pushl $modo2        # Modo: rb
    pushl $file         # Archivo: agenda2.dat
    call fopen          # Llama a fopen
    addl $8, %esp

    mov %eax, fd        # Apuntador al archivo abierto

    cmpl $0, fd         # Si fd==NULL (0) ocurrió un error
    je error

    pushl $cad6         # Pide la posición del contacto
    call printf
    addl $4, %esp

    pushl $pos          # Lee la respuesta del usuario
    pushl $format1
    call scanf
    addl $8, %esp

    decl pos            # desplazamiento = (posición-1)*84
    movl $84, %eax
    imull pos
    movl %eax, desp

    # Localiza el contacto dentro del archivo
    pushl $0            # A partir del comienzo (SEEK_SET : 0)
    pushl %eax          # Se desplaza "desplazamiento" bytes
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    pushl fd            # Apuntador al archivo abierto
    call fseek          # Llama a fseek
    addl $12, %esp

    # Lee el contacto desde el archivo
    pushl fd            # Apuntador al archivo abierto
    pushl $1            # Cantidad de elementos a leer
    pushl $80           # Tamaño del elemento
    pushl $nombre       # Dirección destino de los datos
    call fread          # Llama a fread
    addl $16, %esp

    pushl fd            # Apuntador al archivo abierto
    pushl $1            # Cantidad de elementos a leer
    pushl $4            # Tamaño del elemento
    pushl $tlf          # Dirección destino de los datos
    call fread          # Llama a fread
    addl $16, %esp

    # Si se alcanzó el final del archivo la posición no existe

    pushl fd            # Apuntador al archivo abierto
    call feof           # Llama a feof
    addl $4, %esp
    cmpl $0, %eax
    jne nopos

    pushl tlf           # Muestra la información del contacto
    pushl $nombre
    pushl $cad7
    call printf
    addl $12, %esp

    jmp fin             # Salta al final del programa


    # La posición solicitada no existe
nopos:
    pushl $cad8         # Informa al usuario del problema
    call printf
    addl $4, %esp
    jmp fin             # Salta al final del programa


    # Ocurrió un error al abrir el archivo
error:
    pushl $cad3         # Informa al usuario del error
    call printf
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    addl $4, %esp
    jmp fin2

fin:
    # Cierra el archivo si estaba abierto
    pushl fd            # Descriptor del archivo a cerrar
    call fclose         # Llama a fclose
    addl $4, %esp

fin2:
    # Termina el programa
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

Observe que en esta nueva versión no es necesario recorrer todo el archivo para recuperar la información de un contacto, ya que al conocer *a priori* el tamaño en bytes que ocupa cada registro dentro del archivo (84 bytes) es posible posicionar el apuntador de lectura en el desplazamiento adecuado para leer únicamente la información de interés. Note también que si intenta examinar el contenido del archivo *agenda.dat* en un editor de texto, encontrará imposible determinar el número telefónico de los contactos. Esto se debe a que los mismos están almacenados en formato binario como enteros de 4 bytes.

Llamadas al sistema

Todas las funciones de alto nivel descritas en la sección anterior se apoyan en las llamadas al sistema operativo para llevar a cabo cada una de las operaciones involucradas con el manejo de archivos. Las mismas pueden ser utilizadas directamente por un programa en lenguaje ensamblador, siguiendo los pasos descritos en el capitulo 6.

El sistema operativo Linux presenta una gran variedad de llamadas relacionadas con el manejo de archivos. La siguiente es una lista de las más relevantes, junto con una breve descripción de la tarea que realiza cada una de ellas:

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Llamada	Número	Descripción
creat	8	Crea un nuevo archivo
open	5	Abre un archivo
close	6	Cierra un archivo
read	3	Lee datos desde un archivo abierto
write	4	Escribe datos en un archivo abierto
lseek	19	Cambia la posición dentro del archivo desde la que se lee o escribe
unlink	10	Elimina una entrada del sistema de archivos (si es la única referencia al archivo, borra el archivo)
mkdir	39	Crea un directorio
rmdir	40	Elimina un directorio (debe estar vacío)
rename	38	Cambia el nombre de un archivo o directorio
chmod	15	Cambia los permisos de acceso a un archivo o directorio
chdir	12	Cambia el directorio de trabajo

A continuación se presenta una descripción más detallada de cada una de estas llamadas al sistema:

creat

La llamada al sistema *creat* (8), permite crear un nuevo archivo. El prototipo de la misma es el siguiente:

*int creat(const char *camino, mode_t modo);*

En donde *camino* es una cadena que contiene la ruta (absoluta o relativa) del archivo que se desea crear, mientras que *modo* es un número octal que representa los permisos de acceso que serán asignados a archivo creado. La llamada devuelve el descriptor de archivo asociado al mismo. En caso de no poderse crear el archivo, la llamada devuelve un valor negativo que identifica al error ocurrido.

open

La llamada al sistema *open* (5), permite abrir un archivo existente. Los prototipos de la misma son los siguientes:

*int open(const char *camino, int flags);*

*int open(const char *camino, int flags, mode_t modo);*

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

En donde *camino* es una cadena que contiene la ruta (absoluta o relativa) del archivo que se desea abrir y *flags* es un entero que representa las características con las que se desea operar el archivo a abrir (por ejemplo solo lectura, solo escritura, etc.) En este caso el atributo *modo* es opcional, y tiene la misma utilidad que en el caso de la llamada *creat* en caso de que el archivo a abrir no exista y la bandera O_CREAT esté encendida. La llamada devuelve el descriptor asociado al archivo abierto. En caso de no poderse abrir el archivo, la llamada devuelve un valor negativo que identifica al error ocurrido.

Las diferentes banderas que componen el atributo *flags* se encuentran definidas en el archivo "*fcntl.h*", generalmente ubicado en el directorio "*/usr/include/bits/*". A continuación se presentan las más relevantes:

Bandera	Valor (octal)	Descripción
O_RDONLY	00	Solo lectura
O_WRONLY	01	Solo escritura
O_RDWR	02	Lectura y escritura
O_CREAT	0100	Crea el archivo si no existe
O_TRUNK	01000	Trunca el archivo si existe
O_APPEND	02000	Comienza a escribir al final del archivo

close

La llamada al sistema *close* (6), permite cerrar un archivo, es decir, elimina la relación existente entre el descriptor y el archivo, de forma que el primero deje de hacer referencia al último (note por lo tanto que el cerrar el descriptor no necesariamente implica la liberación de los recursos asociados al archivo, ya que puede existir otro descriptor haciendo referencia al mismo). El prototipo de la llamada *close* es el siguiente:

int close(int fd);

En donde *fd* es el descriptor del archivo que se desea cerrar. La llamada devuelve 0 en caso de tener éxito y -1 en caso de ocurrir un error.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

read

La llamada al sistema *read* (4), permite leer datos desde un archivo. El prototipo de la llamada *read* es el siguiente:

*ssize_t read(int fd, void *buf, size_t nbytes);*

En donde *fd* es el descriptor del archivo desde el cual se desea leer, *buf* es un apuntador a la zona de memoria en la que se almacenarán los datos leidos y *nbytes* es la cantidad máxima de bytes que se leerán. La llamada devuelve el número de bytes leídos en caso de tener éxito (puede ser menor a *nbytes*) y -1 en caso de ocurrir un error.

write

La llamada al sistema *write* (5), permite escribir datos en un archivo. El prototipo de la llamada *write* es el siguiente:

*ssize_t write(int fd, const void *buf, size_t num);*

En donde *fd* es el descriptor del archivo en el cual se desea escribir, *buf* es un apuntador a la zona de memoria que contiene los datos a escribir y *num* es la cantidad de bytes que se escribirán. La llamada devuelve el número de bytes escritos en caso de tener éxito y -1 en caso de ocurrir un error.

lseek

La llamada al sistema *lseek* (19), permite cambiar la ubicación del apuntador de lectura/escritura de un descriptor de archivo. El prototipo de la llamada *lseek* es el siguiente:

off_t lseek(int fildes, off_t offset, int whence);

En donde *fildes* es el descriptor del archivo al cual se desea cambiar la posición del apuntador, *offset* es la cantidad de bytes que se quiere desplazar el mismo y *whence* es la posición a partir de la cual se hará efectivo dicho desplazamiento. La llamada devuelve la posición final del apuntar medida en bytes a partir del comienzo del archivo en caso de tener éxito y -1 en caso de ocurrir un error. Es importante tener en cuenta que esta llamada permite desplazarse más allá del final del archivo sin que esto produzca un error. Si una vez hecho esto se realiza

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

una operación de escritura, los bytes ubicados entre el antiguo final del archivo y la posición actual son rellenados con cero.

unlink

La llamada al sistema *unlink* (10), permite eliminar una entrada del sistema de archivos, y de ser la única referencia al archivo en cuestión, eliminarlo. El prototipo de la llamada *unlink* es el siguiente:

*int unlink(const char *pathname);*

En donde *pathname* es una cadena que contiene la ruta (absoluta o relativa) del archivo que se desea eliminar. La llamada devuelve 0 en caso de tener éxito y -1 en caso de ocurrir un error.

mkdir

La llamada al sistema *mkdir* (39), permite crear un nuevo directorio. El prototipo de la llamada *mkdir* es el siguiente:

*int mkdir(const char *pathname, mode_t mode);*

En donde *pathname* es una cadena que contiene la ruta (absoluta o relativa) del directorio que se desea crear, mientras que *modo* es un número octal que representa los permisos de acceso que serán asignados al directorio creado. La llamada devuelve 0 en caso de tener éxito y -1 en caso de ocurrir un error.

rmdir

La llamada al sistema *rmdir* (40), permite eliminar un directorio existente. El prototipo de la llamada *rmdir* es el siguiente:

*int rmdir(const char *pathname);*

En donde *pathname* es una cadena que contiene la ruta (absoluta o relativa) del directorio que se desea eliminar. La llamada devuelve 0 en caso de tener éxito y -1 en caso de ocurrir un error.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

rename

La llamada al sistema *rename* (38), permite cambiar el nombre (y en consecuencia mover) un archivo o directorio existente. El prototipo de la llamada *rename* es el siguiente:

*int rename(const char *oldpath, const char *newpath);*

En donde *oldpath* es una cadena que contiene la ruta (absoluta o relativa) del directorio que se desea eliminar, y *newpath* es una cadena que contiene la nueva ruta (absoluta o relativa) que se desea asignar al archivo. La llamada devuelve 0 en caso de tener éxito y -1 en caso de ocurrir un error.

chmod

La llamada al sistema *chmod* (15), permite cambiar los permisos de acceso de archivo o directorio existente. El prototipo de la llamada *chmod* es el siguiente:

*int chmod(const char *path, mode_t mode);*

En donde *path* es una cadena que contiene la ruta (absoluta o relativa) del directorio en cuestión, y *mode* es un número octal que representa los nuevos permisos de acceso que serán asignados al mismo. La llamada devuelve 0 en caso de tener éxito y -1 en caso de ocurrir un error.

chdir

La llamada al sistema *chdir* (12), permite cambiar el directorio de trabajo actual. El prototipo de la llamada *chdir* es el siguiente:

*int chdir(const char *path);*

En donde *path* es una cadena que contiene la ruta (absoluta o relativa) del nuevo directorio de trabajo. La llamada devuelve 0 en caso de tener éxito y -1 en caso de ocurrir un error.

Para ilustrar la utilización de las llamadas al sistema operativo anteriormente descritas, se presenta una nueva versión de la agenda telefónica, utilizando nuevamente un archivo binario para almacenar la información de los contactos y acceso directo para manipular su contenido.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Programa en ensamblador:

```
.section .rodata
cad1:   .asciz "1 Insertar contacto\n2 Buscar contacto\n"
cad2:   .asciz "Su opción: "
cad3:   .asciz "Ocurrió un error al intentar abrir el archivo\n"
cad4:   .asciz "Nombre del contacto: "
cad5:   .asciz "Teléfono del contacto: "
cad6:   .asciz "Posición: "
cad7:   .asciz "Nombre: %s\nTelefono: %d\n"
cad8:   .asciz "La posición indicada está vacía\n"
file:   .asciz "agenda3.dat"
format1:.asciz "%d"
format2:.asciz "%s"

.section .bss
nombre: .space 80
tlf:    .space 4
opcion: .space 4
pos:    .space 4
desp:   .space 4
fd:     .space 4


.section .text
.globl _start
_start:

    pushl $cad1         # Muestra las opciones
    call printf
    addl $4, %esp

while:
    pushl $cad2         # Pide una opción
    call printf
    addl $4, %esp

    pushl $opcion       # Lee la respuesta del usuario
    pushl $format1
    call scanf
    addl $8, %esp

    cmpl $1, opcion     # Si es 1 salta a insertar
    je insertar
    cmpl $2, opcion     # Si es 2 salta a buscar
    je buscar
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    jmp while           # En otro caso pregunta de nuevo


    # Abre el archivo para escritura al final
insertar:
    movl $5, %eax       # Servicio #5 (open)
    movl $file, %ebx    # Ruta del archivo a abrir
    movl $02101, %ecx   # O_CREAT | O_APPEND | O_WRONLY
                        # 0100|02000|01:02101
    int $0x80           # Llamada al SO
    mov %eax, fd        # Apuntador al archivo abierto

    cmpl $0, fd         # Si fd<0 ocurrió un error
    jl error

    pushl $cad4         # Pregunta por el nombre del contacto
    call printf
    addl $4, %esp

    pushl $nombre       # Lee la respuesta del usuario
    pushl $format2
    call scanf
    addl $8, %esp

    pushl $cad5         # Pregunta por el telefono del contacto
    call printf
    addl $4, %esp

    pushl $tlf          # Lee la respuesta del usuario
    pushl $format1
    call scanf
    addl $8, %esp

    # Inserta el contacto en el archivo
    movl $4, %eax       # Servicio #4, (write)
    movl fd, %ebx       # File Descriptor del archivo abierto
    movl $nombre, %ecx  # Apuntador a los datos a escribir
    movl $84, %edx      # Número de bytes a escribir
    int $0x80           # Llamada al SO

    jmp fin             # Salta al final del programa


    # Abre el archivo para lectura desde el comienzo
buscar:
    movl $5, %eax       # Servicio #5 (open)
    movl $file, %ebx    # Ruta del archivo a abrir
    movl $0, %ecx       # ORDONLY : 0
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    int $0x80           # Llamada al SO
    mov %eax, fd        # Apuntador al archivo abierto

    cmpl $0, fd         # Si fd<0 ocurrió un error
    jl error

    pushl $cad6         # Pide la posición del contacto
    call printf
    addl $4, %esp

    pushl $pos          # Lee la respuesta del usuario
    pushl $format1
    call scanf
    addl $8, %esp

    decl pos            # desplazamiento = (posición-1)*84
    movl $84, %eax
    imull pos
    movl %eax, desp

    # Localiza el contacto dentro del archivo
    movl $19, %eax      # Servicio #19, (lseek)
    movl fd, %ebx       # File Descriptor del archivo abierto
    movl desp, %ecx     # Desplazamiento en bytes
    movl $0, %edx       # A partir del comienzo (SEEK_SET : 0)
    int $0x80           # Llamada al SO

    # Lee el contacto desde el archivo
    movl $3, %eax       # Servicio #3, (read)
    movl fd, %ebx       # File Descriptor del archivo abierto
    movl $nombre, %ecx  # Dirección donde se almacenarán los datos
    movl $84, %edx      # Número de bytes a leer
    int $0x80           # Llamada al SO

    cmpl $0, %eax       # Si se leyeron cero bytes
    je nopos            # la posición no existe

    pushl tlf           # Muestra la información del contacto
    pushl $nombre
    pushl $cad7
    call printf
    addl $12, %esp

    jmp fin             # Salta al final del programa


    # La posición solicitada no existe
nopos:
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    pushl $cad8         # Informa al usuario del problema
    call printf
    addl $4, %esp
    jmp fin             # Salta al final del programa


    # Ocurrió un error al abrir el archivo
error:
    pushl $cad3         # Informa al usuario del error
    call printf
    addl $4, %esp

fin:
    # Cierra el archivo si estaba abierto
    movl $6, %eax       # Servicio #5 (close)
    movl fd, %ebx       # File Descriptor del archivo abierto
    int $0x80           # Llamada al SO

    # Termina el programa
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

Note que el formato de archivo es idéntico al utilizado por la segunda versión del programa, y en consecuencia los archivos generados por una versión pueden ser leídos por la otra y viceversa, es decir, son compatibles. Observe también que las banderas pasadas como argumento a la llamada *open* deben ser combinadas mediante un OR a nivel de bit.

Si bien este programa puede parecer a simple vista más largo y complejo que la versión que utiliza funciones de alto nivel, también es cierto que el mismo no necesita de bibliotecas externas lo que se traduce en una ejecución más rápida. Adicionalmente considere que utilizando llamadas al sistema el programador tiene un mayor control y libertad sobre las acciones que puede llevar a cabo, y no se encuentra limitado por las restricciones impuestas por los autores de las funciones de alto nivel.

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Ejercicios

12.1- El comando *more* muestra el contenido de un archivo de texto, permitiendo al usuario avanzar una línea presionando la tecla ENTER o a la siguiente página presionando la barra espaciadora. Escriba un programa que simule este comportamiento. Utilice las funciones de alto nivel provistas por el lenguaje C.

12.2- Repita el ejercicio anterior utilizando únicamente llamadas al sistema operativo.

12.3- Modifique la primera versión de la agenda telefónica (acceso secuencial, texto plano, funciones de C), de forma que incluya una opción para eliminar un contacto.

12.4- Repita el ejercicio anterior para la tercera versión de la agenda (acceso directo, archivo binario, llamadas al sistema).

12.5- Escriba un programa que dados dos archivos de texto, produzca como salida un tercer archivo que sea la concatenación de los contenidos de ambos. Utilice las funciones de alto nivel provistas por el lenguaje C.

12.6- Escriba un programa que dados dos archivos binarios con el formato utilizado por la agenda telefónica desarrollada en este capítulo, produzca como salida un nuevo archivo binario que contenga la mezcla de los contactos contenidos en ambos. Asuma que dentro de un mismo archivo no existen contactos repetidos. Utilice únicamente llamadas al sistema operativo. **Ayuda**: Al comparar los nombres de los contactos solo tenga en cuenta la parte de la cadena que precede al carácter nulo (/0).

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

Soluciones a los ejercicios

Capítulo 5

5.1- Cambios ocurridos en los registros para el programa:

```
.section .text
.globl _start
_start:
1      movl $5, %eax
2      xorl %ecx, %ecx
3      addl $6, %ecx
4      addl %ecx, %eax
5      movl %ecx, %ebx
6      movl $2, %edx
7      incl  %ebx
8      imull %edx, %ebx
9      shll $2,%edx
10     sarl $1,%ecx
11     addl %ebx, %eax
12     cltd
13     idivl %ecx
14     movl $1, %eax
15     movl $0, %ebx
       int $0x80
```

	%eax	%ebx	%ecx	%edx
1	5			
2			0	
3			6	
4	11			
5		6		
6				2
7		7		
8		14		
9				8
10			3	
11	25			
12	25			0
13	8			1
14	1			
15		0		

5.2-

```
# Programa que calcula x=a-3b+25(c-d)
.section .data
a:     .long 6
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
b:      .long 8
c:      .long 12
d:      .long 4
x:      .long 0

.section .text
.globl _start
_start:
        movl b, %ebx        # %ebx=b
        imull $3, %ebx      # %ebx=3b
        movl a, %eax        # %eax=a
        subl %ebx, %eax     # %eax=a-3b
        movl c, %ecx        # %ecx=c
        subl d, %ecx        # %ecx=c-d
        imull $25,%ecx      # %ecx=(c-d)*25
        addl %ecx, %eax     # %eax= (a-3b)+((c-d)*25)
        movl %eax, x        # guarda el resultado en x

        movl $1, %eax       # fin del programa
        movl $0, %ebx
        int $0x80
```

5.3-

```
# Calculo de x=16*y
        shll $4, y    # desplaza 4 posiciones a la izquierda lo cual es
                      # equivalente a multiplicar por 2^4
```

5.4-

```
# Calculo de x=y/32
        sarl $5,y     # desplazamiento aritmético de 5 posiciones a la
                      # derecha, equivalente a dividir entre 2^5
```

5.5-

```
# Programa que calcula x=(a+(b-c))/(c*d)
.section .data
a:      .long 4
b:      .long 5
c:      .long 2
d:      .long 7
x:      .long 0

.section .text
.globl _start
_start:
        movl b, %eax        # %eax=b
        subl c, %eax        # %eax=b-c
        addl a, %eax        %eax=(b-c)+a
        movl c, %ecx        # %ecx=c
        imull d, %ecx       # %ecx=c*d
        cltd                # %edx=0
        idivl %ecx          # %eax=((b-c)+a)/(c*d)  %edx=resto
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
      movl %eax, x        # guarda el resultado en x

      movl $1, %eax
      movl $0, %ebx
      int $0x80
```

5.6-

```
# Programa que revisa tres numeros y escoge el mayor
.section .data
a:           .long 4
b:           .long 2
c:           .long 3
mayor: .long 0

.section .text
.globl _start
_start:
              movl a, %eax
              movl b, %ebx
              movl c, %ecx
              cmpl %ebx, %eax          # compara a con b
              jg amayorb               # si a>b salata a amayorb
              cmpl %ecx, %ebx          # compara b con c
              jl bmenorc               # si b<c salta a mnenorc
              movl %ebx, mayor         # cuando b es el mayor
              jmp fin                  # salta a fin

bmenorc:     movl %ecx, mayor          # cuando c es el mayor
             jmp fin                   # salta a fin

amayorb:     cmpl %ecx, %eax           # compara a con c
             jl amenorc                # si a<c salta a amenorc
             movl %eax,mayor           # cuando a es el mayor
             jmp fin                   # salta a fin

amenorc:     movl %ecx, mayor          # cuando c es el mayor

fin:         movl $1, %eax             # fin del programa
             movl $0, %ebx
             int $0x80
```

5.7-

```
# Programa que produce como salida a si a =b; a-b si a>b o b-a si a<b
.section .data
a:           .long 6      # declaración de las variables a y b
b:           .long 8
resultado:   .long 0      # variable resultado para guardar la salida

.section .text
.globl _start
_start:
             movl a, %eax        # %eax= a
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
            movl b, %ebx        # %ebx= b
            cmpl %ebx, %eax            # compara a con b
            je igual                   # si a=b salta a igual
            jg mayor                   # si a>b salta a mayor
            subl %eax, %ebx            # cuando a<b, resta b-a
            movl %ebx, resultado       # actualiza el resultado
            jmp fin                    # salta a fin de programa
igual:      movl %eax, resultado       # cuando a=b resultado=a
            jmp fin                    # salta a fin de programa
mayor:      subl %ebx, %eax            # cuando a>b, resta a-b
            movl %eax, resultado       # actualiza el resultado
fin:        movl $1, %eax       # culmina el programa
            movl $0, %ebx
            int $0x80
```

5.8-

```
1       movl $25, %eax
2       addl $5, %eax
3       pushl %eax
4       incl %eax
5       pushl %eax
6       movl %eax, %ebx
7       subl $16, %ebx
8       cltd
9       idivl %ebx
10      pushl %edx
11      popl %eax
12      popl %ecx
13      popl %ebx
```

	%eax	%ebx	%ecx	%edx	%esp
1	25				
2	30				
3					0x0FC
4	31				
5					0xF08
6		31			
7		15			
8				0	
9	2			1	
10					0xF04
11	1				0xF08
12			31		0xF0C
13		30			0x100

Pila:

Dirección	Contenido
0xF04	1

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

0xF08	31
0x0FC	30
0x100	

Capítulo 6

6.1.-

```
# Programa que muestra valores enteros en diferentes formatos
.section .data
mensaje:      .asciz"Programa que lee 6 valores y los muestra en uno de cuatro
formatos\n"
mensaj2:      .asciz"Los formatos disponibles para mostrar los numeros son:\n"
f1:           .asciz"Horizontal en el mismo orden de entrada [1]\n"
f2:           .asciz"Horizontal en orden inverso             [2]\n"
f3:           .asciz"Vertical en el mismo orden de entrada   [3]\n"
f4:           .asciz"Vertical en orden inverso               [4]\n"
escoge:       .asciz"Escoja un formato, introduzca el numero correspondiente:\t"
leer:         .asciz"Introduzca los 6 valores:\n"
v1:           .asciz"valor 1:\t"
v2:           .asciz"valor 2:\t"
v3:           .asciz"valor 3:\t"
v4:           .asciz"valor 4:\t"
v5:           .asciz"valor 5:\t"
v6:           .asciz"valor 6:\t"
salta:        .asciz"\n\n"
valor:        .asciz"%d"
salf12:       .asciz"Los valores leidos son: \n %d\t %d\t %d\t %d\t %d\t %d\n"
salf34:       .asciz"Los valores leidos son: \n %d\n %d\n %d\n %d\n %d\n %d\n"
forma:        .long 0
val1:         .long 0
val2:         .long 0
val3:         .long 0
val4:         .long 0
val5:         .long 0
val6:         .long 0

.section .text
.globl _start
_start:
              pushl $mensaje      # mensaje inicial
              call printf
              pushl $mensaj2      #  formatos disponibles
              call printf
              pushl $f1
              call printf
              pushl $f2
              call printf
              pushl $f3
              call printf
              pushl $f4
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
        call printf
        addl $24, %esp
        pushl $escoge
        call printf

        pushl $forma
        pushl $valor        # lee formato escogido, lo guarda en
        call scanf          # forma

        pushl $leer
        call printf
        addl $16, %esp

        pushl $v1           # lee valor 1
        call printf
        pushl $val1
        pushl $valor
        call scanf
        addl $12, %esp

        pushl $v2           # lee valor 2
        call printf
        pushl $val2
        pushl $valor
        call scanf
        addl $12, %esp

        pushl $v3           # lee valor 3
        call printf
        pushl $val3
        pushl $valor
        call scanf
        addl $12, %esp

        pushl $v4           # lee valor 4
        call printf
        pushl $val4
        pushl $valor
        call scanf
        addl $12, %esp

        pushl $v5           # lee valor 5
        call printf
        pushl $val5
        pushl $valor
        call scanf
        addl $12, %esp

        pushl $v6           # lee valor 6
        call printf
        pushl $val6
        pushl $valor
        call scanf
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
            addl $12, %esp

            pushl $salta # salto de línea
            call printf
            addl $4, %esp

            cmpl $1, forma
            je uno
            cmpl $2, forma
            je dos
            cmpl $3, forma
            je tres

            pushl val1          # muestra valores segun formato 4
            pushl val2
            pushl val3
            pushl val4
            pushl val5
            pushl val6
            pushl $salf34
            call printf
            jmp fin

uno:        pushl val6          # muestra valores segun formato 1
            pushl val5
            pushl val4
            pushl val3
            pushl val2
            pushl val1
            pushl $salf12
            call printf
            jmp fin

dos:        pushl val1          # muestra valores segun formato 2
            pushl val2
            pushl val3
            pushl val4
            pushl val5
            pushl val6
            pushl $salf12
            call printf
            jmp fin

tres:       pushl val6          # muestra valores segun formato 3
            pushl val5
            pushl val4
            pushl val3
            pushl val2
            pushl val1
            pushl $salf34
            call printf

fin:        movl $1, %eax       # fin del programa
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
            xorl %ebx, %ebx
            int $0x80
```

6.2.-

```
# Programa que suma 4 valores
.section .data
mensaje:     .asciz"Programa que lee 4 valores y los suma\n"
leer:        .asciz"Introduzca los 4 valores:\n"
v1:          .asciz"valor 1:\t"
v2:          .asciz"valor 2:\t"
v3:          .asciz"valor 3:\t"
v4:          .asciz"valor 4:\t"
salta:       .asciz"\n"
valor:       .asciz"%d"
salida:      .asciz"La suma de: %d + %d + %d + %d = %d\n"

val1:        .long 0
val2:        .long 0
val3:        .long 0
val4:        .long 0

.section .text
.globl _start
_start:
             pushl $mensaje     # mensaje inicial
             call printf
             pushl $leer
             call printf
             addl $8, %esp

             pushl $v1          # lee valor 1
             call printf
             pushl $val1
             pushl $valor
             call scanf
             addl $12, %esp

             pushl $v2          # lee valor 2
             call printf
             pushl $val2
             pushl $valor
             call scanf
             addl $12, %esp

             pushl $v3          # lee valor 3
             call printf
             pushl $val3
             pushl $valor
             call scanf
             addl $12, %esp

             pushl $v4          # lee valor 4
             call printf
             pushl $val4
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
            pushl $valor
            call scanf
            addl $12, %esp

            pushl $salta # salto de línea
            call printf
            addl $4, %esp

            movl val1, %eax     # suma los 4 valores
            addl val2, %eax
            addl val3, %eax
            addl val4, %eax

            pushl %eax          # muestra los valores y su suma
            pushl val4
            pushl val3
            pushl val2
            pushl val1
            pushl $salida
            call printf

            movl $1, %eax# fin del programa
            xorl %ebx, %ebx
            int $0x80
```

6.3.-

```
.section .data
mensaje:    .asciz"Programa que lee 4 valores y calcula el promedio\n"
leer:       .asciz"Introduzca los 4 valores:\n"
v1:         .asciz"valor 1:\t"
v2:         .asciz"valor 2:\t"
v3:         .asciz"valor 3:\t"
v4:         .asciz"valor 4:\t"
salta: .asciz"\n"
valor: .asciz"%d"
salida:     .asciz"El promedio de: %d , %d , %d , %d es: %d\n"

val1:       .long 0
val2:       .long 0
val3:       .long 0
val4:       .long 0

.section .text
.globl _start
_start:
            pushl $mensaje      # mensaje inicial
            call printf
            pushl $leer
            call printf
            addl $8, %esp

            pushl $v1           # lee valor 1
            call printf
            pushl $val1
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
            pushl $valor
            call scanf
            addl $12, %esp

            pushl $v2           # lee valor 2
            call printf
            pushl $val2
            pushl $valor
            call scanf
            addl $12, %esp

            pushl $v3           # lee valor 3
            call printf
            pushl $val3
            pushl $valor
            call scanf
            addl $12, %esp

            pushl $v4           # lee valor 4
            call printf
            pushl $val4
            pushl $valor
            call scanf
            addl $12, %esp

            pushl $salta # salto de línea
            call printf
            addl $4, %esp

            movl val1, %eax     # suma los 4 valores
            addl val2, %eax
            addl val3, %eax
            addl val4, %eax
            movl $4, %ecx
            cltd
            idivl %ecx          # divide la suma entre 4

            pushl %eax          # muestra los valores y el promedio
            pushl val4
            pushl val3
            pushl val2
            pushl val1
            pushl $salida
            call printf

            movl $1, %eax # fin del programa
            xorl %ebx, %ebx
            int $0x80
```

6.4-

```
# Programa que lee dos valores a y b y produce como salida, por
# pantalla a si a =b; a-b si a>b o b-a si a<b
.section .data
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
leea:        .asciz"Introduzca el dato a:     \n"
leeb:        .asciz"Introduzca el dato b:     \n"
valor: .asciz"%d"
a:           .long 0
b:           .long 0
salida:      .asciz"resultado:   %d\n"

.section .text
.globl _start
_start:
             leal a, %eax        # %eax=dir inicial de a
             pushl %eax          # pasa la dir de a como parametro
             pushl $valor        # pasa el formato valor como parametro
             call scanf          # llamada a la función de lectura (lee a)
             leal b, %eax        # %eax=dir inicial de b
             pushl %eax          # pasa la dir de b como parametro
             pushl $valor        # pasa el formato valor como parametro
             call scanf          # llamada a la función de lectura (lee b)
             movl a, %eax        # %eax=a
             movl b, %ebx        # %ebx=b
             xorl %ecx, %ecx     # %ecx=0 temporal para el resultado
             cmpl %ebx, %eax     # compara a con b
             je igual            # si a=b salta a igual
             jg mayor            # si a>b salta a mayor
             subl %eax, %ebx     # resta b-a
             movl %ebx, %ecx     # guarda resultado en %ecx
             jmp fin             # salta a fin de programa
igual: movl %eax, %ecx           # cuando a=b, %ecx=a
             jmp fin             # salta a fin de programa
mayor: subl %ebx, %eax           # cuando a>b, resta a-b
             movl %eax, %ecx     # guarda resultado en %ecx
fin:         pushl %ecx          # pasa %ecx como parametro
             pushl $salida       # apila formato de salida
             call printf         # muestra resultado por pantalla
             movl $1, %eax       # fin del programa
             movl $0, %ebx
             int $0x80
```

6.5-

```
# Programa que calcula el monto a pagar para una llamada telefonica
.section .data
leea:        .asciz"Introduzca los minutos de la llamada:   \n"
valor:       .asciz"%d"
min:         .long 0
monto:       .asciz"Monto a pagar:     %d\n"

.section .text
.globl _start
_start:
             leal min, %eax      # %eax= dir de min (indica los minutos)
             pushl %eax          # se apila dir de min
             pushl $valor        # se apila el formato de lectura
             call scanf          # lee min
             xorl %ebx, %ebx     # %ebx=0
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
              movl min, %eax      # %eax=min (minutos de la llamada)
              cmpl $3, min        # compara min con 3
              jle basico          # si min<=3 entonces tarifa=basica
              subl $3, %eax       # si min>3 %eax=min-3 (min adicionales)
              imull $100, %eax, %ebx     # %ebx=min adicionales por 100
basico:       addl $200, %ebx     # %ebx=monto a pagar (si la tarifa es basica
                                  # %ebx=0+200, si hay min adicionales
                                  # %ebx=monto adicional+200)

fin:          pushl %ebx          # apila %ebx=monto a pagar
              pushl $monto # apila formato de salida
              call printf         # muestra por pantalla el monto

              movl $1, %eax# fin del programa
              movl $0, %ebx
              int $0x80
```

6.6.-

```
.section .rodata
cadinic:      .asciz "Introduzca la cadena (max. 20 caracteres): "
caracter1:    .asciz "Introduzca el caracter a reemplazar: "
caracter2:    .asciz "Introduzca el caracter a utilizar como reeemplazo: "
cadmod:       .asciz "La cadena modificada es: "
fcadmod:

.section .bss
cadena:  .space 20    # espacio para la cadena
car1:    .space 1     # espacio para el caracter a reemplazar
car2:    .space 1     # espacio para el nuevo caracter
trash:   .space 10    # espacio para volcar los caracteres que no son de
                      # nuestro interés

.section .text
.globl _start
_start:

    # Mensaje solicitando que se escriba la cadena
    movl $4, %eax                  # Servicio #4 (write)
    movl $1, %ebx                  # File Descriptor = 1 (stdout)
    leal cadinic, %ecx             # Apuntador a la cadena a imprimir
    movl $caracter1-cadinic, %edx  # Número de caracteres a imprimir
    int $0x80                      # Llamada al SO

    # Lectura de la cadena introducida por el usuario
    movl $3, %eax                  # Servicio #3, (servicio de lectura)
    movl $0, %ebx                  # File Descriptor = 0 (stdin)
    leal cadena, %ecx              # Dir donde se almacenará la cadena
    movl $20, %edx                 # Número máximo de caracteres a leer
    int $0x80                      # Llamada al SO

    # Mensaje que solicita caracter a reemplazar
    movl $4, %eax                  # Servicio #4 (write)
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    movl $1, %ebx                   # File Descriptor = 1 (stdout)
    leal caracter1, %ecx            # Apuntador a la cadena a imprimir
    movl $caracter2-caracter1,%edx # Número de caracteres a imprimir
    int $0x80                       # Llamada al SO

    # Lectura del caracter a reemplazar
    movl $3, %eax                   # Servicio #3, (servicio de lectura)
    movl $0, %ebx                   # File Descriptor = 0 (stdin)
    leal car1, %ecx                 # Dir donde se almacenará el caracter
    movl $1, %edx                   # Número máximo de caracteres a leer
    int $0x80                       # Llamada al SO

    # Vaciado del buffer de entrada
    movl $3, %eax                   # Servicio #3, (servicio de lectura)
    movl $0, %ebx                   # File Descriptor = 0 (stdin)
    leal trash, %ecx                # Dir donde se almacenará el caracter
    movl $10, %edx                  # Número máximo de caracteres a leer
    int $0x80                       # Llamada al SO

    # Mensaje que solicita caracter a utilizar como reemplazo
    movl $4, %eax                   # Servicio #4 (write)
    movl $1, %ebx                   # File Descriptor = 1 (stdout)
    leal caracter2, %ecx            # Apuntador a la cadena a imprimir
    movl $cadmod-caracter2, %edx   # Número de caracteres a imprimir
    int $0x80                       # Llamada al SO

    # Lectura del caracter a utilizar como reemplazo
    movl $3, %eax                   # Servicio #3, (servicio de lectura)
    movl $0, %ebx                   # File Descriptor = 0 (stdin)
    leal car2, %ecx                 # Dir donde se almacenará el caracter
    movl $1, %edx                   # Número máximo de caracteres a leer
    int $0x80                       # Llamada al SO

    # Vaciado del buffer de entrada
    movl $3, %eax                   # Servicio #3, (servicio de lectura)
    movl $0, %ebx                   # File Descriptor = 0 (stdin)
    leal trash, %ecx                # Dir donde se almacenará el caracter
    movl $10, %edx                  # Número máximo de caracteres a leer
    int $0x80                       # Llamada al SO

    # Reemplazo del caracter
    movb car2, %bh                  # Lee el caracter de reemplazo
    leal cadena, %ecx               # Dirección de inicio de la cadena
leecar:
    movb (%ecx), %ah                # Carga un caracter de la cadena
    cmpb $0, %ah                    # Si es un caracter nulo
    je fincomp                      # La cadena se terminó
    cmpb car1, %ah                  # Si no es el caracter a reemplazar
    jne noc                         # Lo ignora
    movb %bh, (%ecx)                # De lo contario lo reemplaza
noc: incl %ecx                      # Contínua con el siguiente caracter
    jmp leecar                      # de la cadena
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
fincomp:
    # Imprimir resultado (Mensaje)
    movl $4, %eax                  # Servicio #4 (write)
    movl $1, %ebx                  # File Descriptor = 1 (stdout)
    leal cadmod, %ecx              # Apuntador a la cadena a imprimir
    movl $fcadmod-cadmod, %edx     # Número de caracteres a imprimir
    int $0x80                      # Llamada al SO

    # Imprimir resultado (Cadena modificada)
    movl $4, %eax                  # Servicio #4 (write)
    movl $1, %ebx                  # File Descriptor = 1 (stdout)
    leal cadena, %ecx              # Apuntador a la cadena a imprimir
    movl $20, %edx                 # Número de caracteres a imprimir
    int $0x80                     # Llamada al SO

    # Fin del programa
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

6.7.-

```
.section .data
cad1:    .asciz "Por favor introduzca una cadena: "
cad2:    .asciz "La cadena introducida es: "
cadena:  .space 50    # Reserva espacio para la cadena introducida
fcadena:

.section .text
.globl _start
_start:

    # Muestra un mensaje solicitando que se escriba una cadena
    movl $4, %eax              # Servicio #4 (write)
    movl $1, %ebx              # File Descriptor = 1 (stdout)
    leal cad1, %ecx            # Apuntador a la cadena a imprimir
    movl $cad2-cad1, %edx      # Número máximo de caracteres a imprimir
    int $0x80                  # Llamada al SO

    # Lee la cadena introducida por el usuario
    movl $3, %eax              # Servicio #3, (servicio de lectura)
    movl $0, %ebx              # File Descriptor = 0 (stdin)
    leal cadena, %ecx          # Dirección donde se almacenará la cadena
    movl $49, %edx             # Número máximo de caracteres a leer
    int $0x80                  # Llamada al SO

    # Muestra el resultado (una sola llamada)
    movl $4, %eax              # Servicio #4 (write)
    movl $1, %ebx              # File Descriptor = 1 (stdout)
    leal cad2, %ecx            # Apuntador a la cadena a imprimir
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    movl $fcadena-cad2, %edx   # Número máximo de caracteres a imprimir
    int $0x80                  # Llamada al SO

    # Termina el programa
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

6.8.-

Es necesario reservar espacio para representar 32 bits en lugar de 10 dígitos decimales. Esto es:

```
cadena:  .space 10         # Reserva espacio para la cadena introducida
```

debe cambiarse por

```
cadena:  .space 32         # Reserva espacio para la cadena introducida
```

En segundo lugar, es necesario realizar divisiones sucesivas entre 2 en lugar de 10 para convertir las cadenas de entrada a enteros, y realizar multiplicaciones sucesivas por 2 en lugar de 10 para convertir el entero resultante en una cadena que lo represente. Para ello, deben sustituirse las tres apariciones de:

```
    movl $10, %esi         # base decimal
```

Por:

```
    movl $2, %esi          # base binaria
```

Para facilitar este último cambio, y permitir que el programa funcione de forma genérica para otras bases, es posible definir una constante simbólica:

```
.equ BASE, 2               # Se utiliza base 2
```

Y reemplazar las instrucciones anteriores por:

```
    movl $BASE, %esi           # base binaria
```

Esta solución es válida para cualquier base comprendida entre 2 y 10. Para bases superiores a 10 es necesario introducir lógica adicional que contemple el hecho de que los dígitos superiores al 9 son representados mediante las letras del alfabeto. A continuación se presenta el programa con las modificaciones sugeridas anteriormente:

```
.equ BASE, 2               # Se utiliza base 2

.section .rodata
cad1:    .asciz "Valor de A: "
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
cad2:    .asciz "Valor de B: "
cad3:    .asciz "El valor de A+B es: "
salto:   .asciz "\n"

.section .bss
A:       .space 4          # Reserva espacio para un entero
B:       .space 4          # Reserva espacio para un entero
C:       .space 4          # Reserva espacio para un entero
cadena:  .space 32         # Reserva espacio para la cadena introducida

.section .text
.globl _start
_start:

    # Muestra un mensaje solicitando que proporcione el valor de A
    movl $4, %eax              # Servicio #4, (write)
    movl $1, %ebx              # File Descriptor = 1 (stdout)
    leal cad1, %ecx            # Apuntador a la cadena a imprimir
    movl $cad2-cad1, %edx      # Número máximo de caracteres a imprimir
    int $0x80                  # Llamada al SO

    # Lee la cadena introducida por el usuario
    movl $3, %eax              # Servicio #3, (read)
    movl $0, %ebx              # File Descriptor = 0 (stdin)
    leal cadena, %ecx          # Dirección donde se almacenará la cadena
    movl $49, %edx             # Número máximo de caracteres a leer
    int $0x80                  # Llamada al SO

    # Obtiene el valor entero representado por la cadena
    xorl %ecx, %ecx            # i=0
    xorl %eax, %eax            # A=0
    movl $BASE, %esi           # base
for1:
    movb cadena(%ecx), %bl# Carga un caracter. (cadena[i])
    cmpb $0, %bl               # Si llega al final de la cadena
    je ffor1                   # Sale del ciclo
    cmpb $10, %bl              # Si llega al final de la cadena
    je ffor1                   # Sale del ciclo
    subb $'0', %bl             # Obtiene el dígito entero que representa
    movzx %bl, %ebx            # Expande el entero a 32 bits (D)
    mull %esi                  # A=A*10
    addl %ebx, %eax            # A=A+D
    incl %ecx                  # Se repite para el siguiente caracter
    jmp for1
ffor1:
    movl %eax, A               # Guarda el valor de A

    # Muestra un mensaje solicitando que proporcione el valor de A
    movl $4, %eax              # Servicio #4, (write)
    movl $1, %ebx              # File Descriptor = 1 (stdout)
    leal cad2, %ecx            # Apuntador a la cadena a imprimir
    movl $cad3-cad2, %edx      # Número máximo de caracteres a imprimir
    int $0x80                  # Llamada al SO
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    # Lee la cadena introducida por el usuario
    movl $3, %eax              # Servicio #3, (read)
    movl $0, %ebx              # File Descriptor = 0 (stdin)
    leal cadena, %ecx          # Dirección donde se almacenará la cadena
    movl $49, %edx             # Número máximo de caracteres a leer
    int $0x80                  # Llamada al SO

    # Obtiene el valor entero representado por la cadena
    xorl %ecx, %ecx            # i=0
    xorl %eax, %eax            # A=0
    movl $BASE, %esi           # base

for2:
    movb cadena(%ecx), %bl# Carga un caracter. (cadena[i])
    cmpb $0, %bl               # Si llega al final de la cadena
    je ffor2                   # Sale del ciclo
    cmpb $10, %bl              # Si llega al final de la cadena
    je ffor2                   # Sale del ciclo
    subb $'0', %bl             # Obtiene el dígito entero que representa
    movzx %bl, %ebx            # Expande el entero a 32 bits (D)
    mull %esi                  # A=A*10
    addl %ebx, %eax            # A=A+D
    incl %ecx                  # Se repite para el siguiente caracter
    jmp for2
ffor2:
    movl %eax, B               # Guarda el valor de B

    # Calcula el valor de A+B
    movl A, %eax
    addl B, %eax
    movl %eax, C

    # Obtiene la cadena que representa al entero C
    movb $0, cadena+10         # Termina la cadena (‘\0’)
    movl $9, %ecx              # i=9
    movl $BASE, %esi           # base
for3:
    cltd                       # Expande C a 64 bits (edx:eax)
    idivl %esi                 # Lo divide entre la base
    addb $'0', %dl             # Obtiene la representación del dígito   decimal
    movb %dl, cadena(%ecx)     # Lo inserta en la cadena
    decl %ecx
    cmpl $0, %eax              # Se repite mientras queden dígitos
    jne for3
    movl %ecx, %esi            # Guarda el desplazamiento
    incl %esi

    # Muestra el resultado (parte 1)
    movl $4, %eax              # Servicio #4, (write)
    movl $1, %ebx              # File Descriptor = 1 (stdout)
    leal cad3, %ecx            # Apuntador a la cadena a imprimir
    movl $salto-cad3, %edx     # Número máximo de caracteres a imprimir
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    int $0x80                 # Llamada al SO

    # Muestra el resultado (parte 2)
    movl $4, %eax             # Servicio #4, (write)
    movl $1, %ebx             # File Descriptor = 1 (stdout)
    leal cadena, %ecx         # Apuntador a la cadena a imprimir
    addl %esi, %ecx           # Ajustamos la dirección de la cadena
    movl $10, %edx            # Número máximo de caracteres a imprimir
    int $0x80                 # Llamada al SO

    # Imprime un salto de línea
    movl $4, %eax             # Servicio #4, (write)
    movl $1, %ebx             # File Descriptor = 1 (stdout)
    leal salto, %ecx          # Apuntador a la cadena a imprimir
    movl $1, %edx             # Número máximo de caracteres a imprimir
    int $0x80                 # Llamada al SO

    # Termina el programa
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

6.9.-

```
.section .rodata
cad1:    .asciz "Por favor introduzca un entero decimal con signo: "
cad2:    .asciz "El valor del entero introducido es: %d\n"

.section .bss
NUM:     .space 4         # Reserva espacio para almacena el entero
SIGNO:   .space 4         # Reserva espacio para almacena el signo
cadena:  .space 12        # Reserva espacio para la cadena introducida

.section .text
.globl _start
_start:

    # Muestra un mensaje solicitando que proporcione el valor del entero
    movl $4, %eax             # Servicio #4, (write)
    movl $1, %ebx             # File Descriptor = 1 (stdout)
    leal cad1, %ecx           # Apuntador a la cadena a imprimir
    movl $cad2-cad1, %edx     # Número máximo de caracteres a imprimir
    int $0x80                 # Llamada al SO

    # Lee la cadena introducida por el usuario
    movl $3, %eax             # Servicio #3, (read)
    movl $0, %ebx             # File Descriptor = 0 (stdin)
    leal cadena, %ecx         # Dirección donde se almacenará la cadena
    movl $49, %edx            # Número máximo de caracteres a leer
    int $0x80                 # Llamada al SO
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    # Obtiene el valor entero representado por la cadena

    xorl %ecx, %ecx             # i=0
    xorl %eax, %eax             # A=0
    movl $10, %esi              # base decimal

    movb cadena(%ecx), %bl      # Lee el primer caracter de la cadena
    cmpb $'-', %bl              # Si es '-' el número es negativo
    je negativo
    movl $1, SIGNO              # El número se debe multiplicar por 1
    cmpb $'+', %bl              # Si no tiene el símbolo '+'
    jne for1                    # Comienza desde i=0
    incl %ecx                   # Si lo tiene, omite el primer caracter
    jmp for1

negativo:
    movl $-1, SIGNO             # El número se debe multiplicar por -1
    incl %ecx                   # Omitir el primer caracter

for1:
    movb cadena(%ecx), %bl      # Carga un caracter. (cadena[i])
    cmpb $0, %bl                # Si llega al final de la cadena
    je ffor1                    # Sale del ciclo
    cmpb $10, %bl               # Si llega al final de la cadena
    je ffor1                    # Sale del ciclo
    subb $'0', %bl              # Obtiene el dígito entero que representa
    movzx %bl, %ebx             # Expande el entero a 32 bits (D)
    mull %esi                   # A=A*10
    addl %ebx, %eax             # A=A+D
    incl %ecx                   # Se repite para el siguiente caracter
    jmp for1
ffor1:
    imull SIGNO                 # Multiplica por el signo
    movl %eax, NUM              # Guarda el valor del entero

    # Muestra el entero obtenido al usuario
    pushl NUM
    pushl $cad2
    call printf
    addl $8, %esp

    # Termina el programa
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

6.10.-

```
# Datos de prueba definidos en el segmento de datos equivalentes a
# scanf("Soy %s y me gusta la letra %c, &string, &char)
# A diferencia de scanf, toma en cuenta los espacios
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
.section .rodata
params:  .long string, char
format:  .asciz "Soy %s y me gusta la letra %c"
msj:     .asciz "Cadena de entrada: "
msj2:    .asciz "Eres %s y te gusta la letra %c\n"

.section .bss
char:    .space 10         # Variable char
string:  .space 20         # Variable string
cadena:  .space 100        # Reserva espacio para la cadena introducida

.section .text
.globl _start
_start:

    # Muestra un mensaje solicitando la cadena de entrada
    movl $4, %eax              # Servicio #4, (write)
    movl $1, %ebx              # File Descriptor = 1 (stdout)
    leal msj, %ecx             # Apuntador a la cadena a imprimir
    movl $msj2-msj, %edx       # Número máximo de caracteres a imprimir
    int $0x80                  # Llamada al SO

    # Lee la cadena introducida por el usuario
    movl $3, %eax              # Servicio #3, (read)
    movl $0, %ebx              # File Descriptor = 0 (stdin)
    leal cadena, %ecx          # Dirección donde se almacenará la cadena
    movl $49, %edx             # Número máximo de caracteres a leer
    int $0x80                  # Llamada al SO

    # Ciclo para buscar especificadores de formato
    xorl %esi, %esi            # i=0
    xorl %edi, %edi            # j=0
    xorl %ecx, %ecx            # reconocidos=0
buscar:
    movb format(%esi), %al     # Carga un caracter de la cadena de formato
    cmpb $'\0', %al            # Si es un caracter nulo
    je fbuscar                 # Alcanzó el final de la cadena de formato
    cmpb $'%', %al             # Si es un símbolo de porcentaje
    je reconocer               # Debe reconocer una variable
    cmpb cadena(%edi), %al     # De lo contrario, debe coincidir con
    jne fbuscar                # la cadena proporcionada por el usuario

    incl %esi                  # Continúa procesando la cadena de formato
    incl %edi
    jmp buscar

reconocer:
    incl %esi                  # Carga el caracter que especifica
    movb format(%esi), %al     # el tipo de variable a reconocer
    cmpb $'c', %al             # Reconoce un caracter
    je recon_c
    cmpb $'s', %al             # Reconoce una cadena
    je recon_s
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    jmp error                 # Especificador inválido

recon_c:
    movb cadena(%edi), %al    # Caracter reconocido
    movl params(,%ecx,4),%edx  # Dirección destino
    movb %al, (%edx)          # Almacena el caracter
    incl %edx

    incl %ecx                 # reconocidos++
    incl %esi                 # Continúa procesando la cadena de formato
    incl %edi
    jmp buscar

recon_s:
    incl %esi
    movb format(%esi), %bl    # Siguiente caracter de la cadena de formato
    movl params(,%ecx,4),%edx  # Dirección destino
ciclo:
    movb cadena(%edi), %al    # Caracter reconocido
    cmpb %al, %bl             # Si coincide con la cadena de formato
    je fciclo
    movb %al, (%edx)          # Almacena el caracter
    incl %edi                 # Siguiente caracter
    incl %edx
    jmp ciclo
fciclo:
    movb $0, (%edx)           # Cierra la cadena
    incl %ecx                 # reconocidos++
    incl %esi                 # Continúa procesando la cadena de formato
    incl %edi
    jmp buscar

fbuscar:
    # Muestra las variables reconocidas por el 'scanf'
    pushl char
    pushl $string
    pushl $msj2
    call printf
    addl $12, %esp

    # Termina el programa
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

6.11.-

```
.section .data
fkeys:   .byte 59,60,61,62,63,0,1,2,3,4,23,24
CADP:    .asciz "\rSe ha presionado la tecla F%d\n"
CADL:    .asciz "\rSe ha levantado la tecla F%d\n"
ERROR:   .asciz "Se ha denegado el permiso sobre los puertos del teclado\n"
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
.section .bss
SC:      .space 1          # Espacio para almacenar el scancode

.section .text
.globl _start

_start:
    # Solicita permiso sobre los puertos del teclado
    movl $101, %eax            # Servicio 101 (ioperm)
    movl $0x60, %ebx           # Desde 0x60 (Registro de datos)
    movl $5, %ecx              # Hasta 0x64 (Registro de estado)
    movl $1, %edx              # Obtener permiso
    int $0x80                  # Llamada al SO

    testl    %eax,%eax         # No se obtuvo el permiso
    jne err

espera:
    inb $0x64, %al        # Lee el registro de estado
    andb $1, %al               # Enmascara el bit 0
    cmpb $0, %al               # Si está apagado espera
    je espera

    xorl %eax, %eax       # Limpia eax
    inb $0x60,%al         # Lee el scancode del registro de datos
    cmpb $0x1C,%al        # Si es ENTER (presionada) terminar
    je fin

    movb %al, SC          # Determina si la tecla fue
    andb $0x80, %al       # presionada o levantada
    je pres
lev:                      # Carga la dirección de la cadena apropiada
    leal CADL, %ebx
    jmp continua
pres:
    leal CADP, %ebx

continua:
    andl $0x3F, SC        # Limpia el bit 7 del scancode
    movb SC, %al

    xorl %ecx, %ecx
ciclo:
    cmpl $12, %ecx        # Si no es una tecla de función
    jge espera            # la ignora

    cmpb fkeys(%ecx), %al # De lo contario la muestra
    je mostrar

    incl %ecx             # Continúa la búsqueda
    jmp ciclo
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
mostrar:
    incl %ecx             # Muestra la tecla
    pushl %ecx
    pushl %ebx
    call printf
    addl $8, %esp
    jmp espera

err: pushl $ERROR         # Informa al usuario sobre el error
    call printf
    addl $4, %esp

    # Libera los permisos sobre los puertos del teclado
fin: movl $101, %eax      # Servicio 101 (ioperm)
    movl $0x60, %ebx      # Desde 0x60 (Registro de datos)
    movl $5, %ecx         # Hasta 0x64 (Registro de estado)
    movl $0, %edx         # Liberar permiso
    int $0x80             # Llamada al SO

    movl $1, %eax         # Termina el programa
    movl $0, %ebx
    int $0x80
```

6.12.-

```
.section .data
fkeys:   .byte 59,60,61,62,63,0,1,2,3,4,23,24
CAD:     .asciz "\r                              "
CADP:    .asciz "\rSe ha presionado la tecla F%d\n"
CADL:    .asciz "\rSe ha levantado la tecla F%d\n"
ERROR:   .asciz "Se ha denegado el permiso sobre los puertos del teclado\n"

.section .bss
SC:      .space 1         # Espacio para almacenar el scancode
LASTSC:  .space 1         # Espacio para almacenar el scancode anterior

    .section .text
    .globl _start

_start:
    # Solicita permiso sobre los puertos del teclado
    movl $101, %eax       # Servicio 101 (ioperm)
    movl $0x60, %ebx      # Desde 0x60 (Registro de datos)
    movl $5, %ecx         # Hasta 0x64 (Registro de estado)
    movl $1, %edx         # Obtener permiso
    int $0x80             # Llamada al SO

    testl   %eax,%eax     # No se obtuvo el permiso
    jne err

espera:
    inb $0x64, %al        # Lee el registro de estado
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    andb $1, %al           # Enmascara el bit 0
    cmpb $0, %al           # Si está apagado espera
    je espera

    xorl %eax, %eax        # Limpia eax
    inb $0x60,%al          # Lee el scancode del registro de datos
    cmpb $0x1C,%al         # Si es ENTER (presionada) terminar
    je fin

    cmpb %al, LASTSC       # Si el scancode es igual al anterior
    je limpia              # ignorar la tecla

    movb %al, LASTSC       # Salvaguarda el scancode recibido
    movb %al, SC           # Determina si la tecla fue
    andb $0x80, %al        # presionada o levantada
    je pres

# Carga la dirección de la cadena apropiada
lev:
    leal CADL, %ebx
    jmp continua
pres:
    leal CADP, %ebx

continua:
    andb $0x3F, SC         # Limpia el bit 7 del scancode
    movb SC, %al

    xorl %ecx, %ecx
ciclo:
    cmpl $12, %ecx         # Si no es una tecla de función
    jge espera             # ignorarla

    cmpb fkeys(%ecx), %al # De lo contrario la muestra
    je mostrar

    incl %ecx              # Continúa la búsqueda
    jmp ciclo

mostrar:
    incl %ecx              # Muestra la tecla
    pushl %ecx
    pushl %ebx
    call printf
    addl $8, %esp
    jmp espera

limpia:
    pushl $CAD             # Limpia la pantalla (la línea actual)
    call printf
    addl $4, %esp
    jmp espera
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
err: pushl $ERROR          # Informa al usuario sobre el error
     call printf
     addl $4, %esp

     # Libera los permisos sobre los puertos del teclado
fin: movl $101, %eax       # Servicio 101 (ioperm)
     movl $0x60, %ebx      # Desde 0x60 (Registro de datos)
     movl $5, %ecx         # Hasta 0x64 (Registro de estado)
     movl $0, %edx         # Liberar permiso
     int $0x80             # Llamada al SO

     movl $1, %eax         # Termina el programa
     movl $0, %ebx
     int $0x80
```

6.13.-

```
.section .data
ALT:     .long 0
CTRL:    .long 0
Q:       .long 0

BLANCO:  .asciz "\r                                \r"
MSJ:     .asciz "\rSe ha presionado la combinación ALT+CTRL+Q\n"
ERROR:   .asciz "Se ha denegado el permiso sobre los puertos del teclado\n"

.section .bss
SC:      .space 1            # Espacio para almacenar el scancode

.section .text
.globl _start

_start:
     # Solicita permiso sobre los puertos del teclado
     movl $101, %eax         # Servicio 101 (ioperm)
     movl $0x60, %ebx        # Desde 0x60 (Registro de datos)
     movl $5, %ecx           # Hasta 0x64 (Registro de estado)
     movl $1, %edx           # Obtener permiso
     int $0x80               # Llamada al SO

     testl   %eax,%eax       # No se obtuvo el permiso
     jne err

espera:
     inb $0x64, %al          # Lee el registro de estado
     andb $1, %al            # Enmascara el bit 0
     cmpb $0, %al            # Si está apagado espera
     je espera

     xorl %eax, %eax         # Limpia eax
     inb $0x60,%al           # Lee el scancode del registro de datos
     cmpb $0x1C,%al          # Si es ENTER (presionada) terminar
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    je fin

    cmpb $0x1D,%al              # Si se presionó la tecla ALT
    je presiono_alt
    cmpb $0x38,%al              # Si se presionó la tecla CTRL
    je presiono_ctrl
    cmpb $0x10,%al              # Si se presionó la tecla Q
    je presiono_q

    cmpb $0x9D,%al              # Si se levantó la tecla ALT
    je levanto_alt
    cmpb $0xb8,%al              # Si se levantó la tecla CTRL
    je levanto_ctrl
    cmpb $0x90,%al              # Si se levantó la tecla Q
    je levanto_q

    # Si es cualquier otra tecla la misma es ignorada
    jmp limpiar

presiono_alt:
    movl $1, ALT                # Enciende la bandera ALT
    jmp comprueba               # Comprueba el estado de las banderas
presiono_ctrl:
    movl $1, CTRL               # Enciende la bandera CTRL
    jmp comprueba               # Comprueba el estado de las banderas
presiono_q:
    movl $1, Q                  # Enciende la bandera Q

comprueba:
    movl ALT, %eax              # Hace un AND entre las tres banderas
    andl CTRL, %eax
    andl Q, %eax
    cmpl $0, %eax               # Si no se cumple la combinación
    je limpiar                  # Espera por otra tecla

    # Si se cumple la combinación, notifica al usuario
    pushl $MSJ
    call printf
    addl $4, %esp
    jmp espera

levanto_alt:
    movl $0, ALT                # Apaga la bandera ALT
    jmp espera                  # Espera por otra tecla
levanto_ctrl:
    movl $0, CTRL               # Apaga la bandera CTRL
    jmp espera                  # Espera por otra tecla
levanto_q:
    movl $0, Q                  # Apaga la bandera Q
    jmp espera                  # Espera por otra tecla

limpiar:
    movl $4, %eax               # Servicio #4, (write)
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    movl $1, %ebx               # File Descriptor = 1 (stdout)
    leal BLANCO, %ecx           # Apuntador a la cadena a imprimir
    movl $MSJ-BLANCO, %edx      # Número máximo de caracteres a imprimir
    int $0x80                   # Llamada al SO
    jmp espera

err: pushl $ERROR               # Informa al usuario sobre el error
    call printf
    addl $4, %esp

    # Libera los permisos sobre los puertos del teclado
fin: movl $101, %eax            # Servicio 101 (ioperm)
    movl $0x60, %ebx            # Desde 0x60 (Registro de datos)
    movl $5, %ecx               # Hasta 0x64 (Registro de estado)
    movl $0, %edx               # Liberar permiso
    int $0x80                   # Llamada al SO

    movl $1, %eax               # Termina el programa
    movl $0, %ebx
    int $0x80
```

6.14.-

```
.section .data
TABLE:   .byte 0b000, 0b001, 0b100, 0b101, 0b010,0b011,0b110,0b111
ACUM:    .byte 0, '\n'          # Acumulador de la calculadora binaria
AUX:     .byte 0, '\n'          # Operando proporcionado por el usuario
MSJ:     .asciz "\r           \r"
OPER:    .ascii "    \r"
SALTO:   .byte '\n'
ERROR:   .asciz "Se ha denegado el permiso sobre los puertos del teclado\n"

    .bss
SC: .space 1                    # Espacio para almacenar el scancode

    .section .text
    .globl _start

_start:
    # Solicita permiso sobre los puertos del teclado
    movl $101, %eax             # Servicio 101 (ioperm)
    movl $0x60, %ebx            # Desde 0x60 (Registro de datos)
    movl $5, %ecx               # Hasta 0x64 (Registro de estado)
    movl $1, %edx               # Obtener permiso
    int $0x80                   # Llamada al SO

    testl    %eax,%eax          # No se obtuvo el permiso
    jne err

    xorl %esi, %esi             # Caracteres reconocidos

    jmp mostrar                 # Coloca las luces del teclado en 000
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
espera:
    inb $0x64, %al            # Lee el registro de estado
    andb $1, %al              # Enmascara el bit 0
    cmpb $0, %al              # Si está apagado espera
    je espera

    xorl %eax, %eax           # Limpia eax
    inb $0x60,%al             # Lee el scancode del registro de datos
    cmpb $0x1C,%al            # Si es ENTER (presionada) terminar
    je fin

    movb %al, SC
    andb $0x80, SC            # Ignora las teclas levantadas
    jne espera

    cmpl $0, %esi             # Las posiciones 1, 2 y 3 son bits
    jne bit

operacion:
    cmpb $78,%al              # Si se presionó la tecla +
    je presiono_or
    cmpb $55,%al              # Si se presionó la tecla *
    je presiono_and

    # Si es cualquier otra tecla la misma es ignorada
    jmp imprime

bit:
    cmpb $82,%al              # Si se presionó la tecla 0
    je presiono_0
    cmpb $79,%al              # Si se presionó la tecla 1
    je presiono_1

    # Si es cualquier otra tecla la misma es ignorada
    jmp imprime

presiono_0:
    movb $'0', OPER(%esi)
    shlb $1, AUX              # AUX = AUX << 1
    incl %esi
    jmp comprueba             # Comprueba el estado de las banderas
presiono_1:
    movb $'1', OPER(%esi)
    shlb $1, AUX              # AUX = AUX << 1
    incb AUX                  # AUX = AUX + 1
    incl %esi
    jmp comprueba             # Comprueba el estado de las banderas
presiono_or:
    movb $'+', OPER
    incl %esi
    jmp comprueba             # Comprueba el estado de las banderas
presiono_and:
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    movb $'*', OPER
    incl %esi

comprueba:
    cmpl $4, %esi              # Si se leyeron los 3 bits
    je opera                   # Efectúa la operación

imprime:
    movl $4, %eax              # Servicio #4, (write)
    movl $1, %ebx              # File Descriptor = 1 (stdout)
    leal MSJ, %ecx             # Apuntador a la cadena a imprimir
    movl $SALTO-MSJ, %edx      # Número máximo de caracteres a imprimir
    int $0x80                  # Llamada al SO
    jmp espera

opera:
    movb AUX, %al              # Carga el operador (+ o *)
    cmpb $'+', OPER            # Si es +
    je ejecuta_or              # Se ejecuta un OR

ejecuta_and:
    andb %al, ACUM             # ACUM = ACUM and AUX
    jmp seguir

ejecuta_or:
    orb %al, ACUM              # ACUM = ACUM or AUX

seguir:
    movl $4, %eax              # Servicio #4, (write)
    movl $1, %ebx              # File Descriptor = 1 (stdout)
    leal MSJ, %ecx             # Apuntador a la cadena a imprimir
    movl $ERROR-MSJ, %edx      # Número máximo de caracteres a imprimir
    int $0x80                  # Llamada al SO

mostrar:
    inb $0x64, %al             # Lee el registro de control
    andb $2, %al               # Enmascara el bit 1
    jne mostrar                # Si no está listo para recibir espera

    movb $0xED, %al            # Comando 0xED: Luces del teclado
    outb %al,$0x60             # Envía el comando
    movb ACUM, %al             # Carga el digito octal
    leal TABLE, %ebx           # Dirección de la tabla de traducción
    xlat                       # Traduce al patrón de las luces
    outb %al,$0x60             # Envía el patrón

    # Limpia el mensaje mostrado al usuario y reinicia las variables
    xorl %esi, %esi
    movb $0, AUX
    movb $' ', OPER
    movb $' ', OPER+1
    movb $' ', OPER+2
    movb $' ', OPER+3
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    jmp espera

err: pushl $ERROR               # Informa al usuario sobre el error
     call printf
     addl $4, %esp

     # Libera los permisos sobre los puertos del teclado
fin: movl $101, %eax            # Servicio 101 (ioperm)
     movl $0x60, %ebx           # Desde 0x60 (Registro de datos)
     movl $5, %ecx              # Hasta 0x64 (Registro de estado)
     movl $0, %edx              # Liberar permiso
     int $0x80                  # Llamada al SO

     movl $1, %eax              # Termina el programa
     movl $0, %ebx
     int $0x80
```

Capítulo 7

7.1-

```
# Programa que imprime Hola horizontal
.section .data
salida:      .asciz"Hola\t"
leenum:      .asciz"%d"
salta: .asciz"\n"
numero:      .long 0

.section .text
.globl _start
_start:
             leal numero, %eax
             pushl %eax
             pushl $leenum
             call scanf          # lee numero
ciclo:       cmpl $0, numero
             je fin
             pushl $salida
             call printf
             decl numero
             jmp ciclo
fin:         pushl $salta
             call printf
             movl $1, %eax
             movl $0, %ebx
             int $0x80
```

7.2-

```
# Programa que imprime Hola vertical
.section .data
salida:      .asciz"Hola\n"
leenum:      .asciz"%d"
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
numero:       .long 0

.section .text
.globl _start
_start:
             leal numero, %eax
             pushl %eax
             pushl $leenum
             call scanf
ciclo:       cmpl $0, numero
             je fin
             pushl $salida
             call printf
             decl numero
             jmp ciclo
fin:         movl $1, %eax
             movl $0, %ebx
             int $0x80
```

7.3-

```
# Programa que calcula x elevado a la y
.section .data
mensaje0:    .asciz"Programa que calcula x (base) elevado a la y (potencia)\n"
leedato:     .asciz"%d"
mensaje1:    .asciz"introduzca la base: \t"
mensaje2:    .asciz"introduzca la potencia: \t"
base:        .asciz"%d\t"
result:             .asciz"%d\t elevado a la %d\t es: %d\n"
x:                  .long 0
y:                  .long 0

.section .text
.globl _start
_start:
             pushl $mensaje0           # Mensaje inicial
             call printf
             pushl $mensaje1           # Introducir dato de la base
             call printf
             leal x, %eax       # Leer base
             pushl %eax
             pushl $leedato
             call scanf
             pushl $mensaje2           # Introducir la potencia
             call printf
             leal y, %eax       # Leer potencia
             pushl %eax
             pushl $leedato
             call scanf
                                              # %eax se usa para calcular el
resultado
             xorl %eax, %eax           # %eax = 0
             cmpl $0, x                # compara %eax con 0
             jz fin                    # si x=0 resultado=0
             movl $1, %eax      # %eax=1
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
            movl y, %ecx        # %ecx=y tiene la potencia
mult:  cmpl $0, %ecx       # compara y con 0
            je fin                    # si y=0 termina
            imull x, %eax       # multiplica la base y veces
            decl %ecx                 # decrementa potencia
            jmp mult                  # regresa al inicio del ciclo
fin:   pushl %eax                # Muestra el resultado
            pushl y
            pushl x
            pushl $result
            call printf
            movl $1, %eax       # termina el programa
            movl $0, %ebx
            int $0x80
```

7.4-

```
# Programa que calcula el monto a pagar a un trabajador semanal
.section .data
mensaje:     .asciz"Programa que calcula el monto a pagar a un trabajador\n"
mensaj2:     .asciz"Los trabajadores se clasifican como:\n"
clas1:       .asciz"Trabajador no calificado               [1]\n"
clas2:       .asciz"Trabajador calificado                  [2]\n"
clas3:       .asciz"Trabajador calificado con experiencia   [3]\n\n"
trabaj:      .asciz"Introduzca la clasificacion del trabajador:\t"
ttrab:       .long 0
leer:        .asciz"Introduzca la cantidad de horas trabajadas en la semana:\t"
salta:       .asciz"\n"
entrada:     .asciz"%d"
salida:      .asciz"El monto a pagar es: %d\n"
horas:       .long 0
basico:      .long 0
medio:       .long 0
doble:       .long 0
.equ         sueldo1, 3250
.equ         sueldo2, 4500
.equ         sueldo3, 4850

.section .text
.globl _start
_start:
             pushl $mensaje            # muestra mensajes iniciales
             call printf
             pushl $mensaj2
             call printf
             pushl $clas1
             call printf
             pushl $clas2
             call printf
             pushl $clas3
             call printf
             pushl $trabaj
             call printf
             pushl $ttrab
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
                pushl $entrada
                call scanf                # lee tipo de trabajador
                addl $28, %esp

                pushl $leer
                call printf
                pushl $horas
                pushl $entrada
                call scanf                # lee cantidad de horas trabajadas
                addl $16, %esp

                pushl $salta
                call printf
                addl $4, %esp

                movl horas, %eax
                cmpl $40, %eax            # compara horas con 40
                jle hasta40               # si <= salta a hasta 40
                movl $40, basico          # si > basico=40
                subl $40, %eax            # y resta 40 de horas
                cmpl $20, %eax            # compara resto con 20
                jle hasta60               # si <= salta a hasta60
                movl $20, medio           # si > medio=20
                subl $20, %eax            # resta las 20 horas
                movl %eax, doble          # doble=resto de las horas (mas de 60)
                jmp seguir
hasta40:        movl %eax, basico         # cuando <40 horas, basico=cant.horas
                       jmp seguir
hasta60:        movl %eax, medio          # cuando horas esta entre 40 y 60
                                          # medio=cant horas>40
seguir:         cmpl $1, ttrab            # revisa si es trabajador no calificado
                jne califica              # si no, salta a califica
                movl $sueldo1, %edx       # trab no calificado %edx=3250
                jmp calcula

califica:       cmpl $2, ttrab            # revisa si es trab. calificado
                jne califica2             # si es trabajador calificado con
                                          # experiencia salta a califica2
                movl $sueldo2, %edx # trab. calificado %edx=4500
                jmp calcula

califica2:      movl $sueldo3, %edx # trab. calificado con exp. %edx=4850

calcula:        movl %edx, %edi           # %edi para calcular sueldo y medio
                movl %edx, %esi           # %esi para calcular doble sueldo
                sarl $1, %edi             # sueldo/2
                addl %edx, %edi           # %edi=sueldo*1.5
                shll $1, %esi             # sueldo *2
                imull basico, %edx        # horas a sueldo basico por sueldo
                imull medio, %edi         # horas a sueldo medio por sueldo y 1/2
                imull doble, %esi         # horas a doble sueldo por doble sueldo
                addl %edi, %edx           # suma de las tres partes del calculo
                addl %esi, %edx
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
            pushl %edx                # muestra el monto por pantalla
            pushl $salida
            call printf

            movl $1, %eax
            xorl %ebx, %ebx
            int $0x80
```

Capítulo 8

8.1-

```
# Programa que suma los elementos de un arreglo de n enteros
.section .data
lee:        .asciz"Introduzca el numero de elementos del arreglo:\n"
n:          .long 0
leenum:     .asciz"Introduzca a[ %d ]: \t"
leen:       .asciz"%d"
x:          .long 0
resul:      .asciz"La suma de los elementos del arreglo es:      %d\n"

.section .bss
arreglo:    .space 400

.section .text
.globl _start
_start:
            pushl $lee          # Imprime mensaje inicial
            call printf
            leal n, %eax
            pushl %eax
            pushl $leen
            call scanf          # lee n (numero de elementos del arreglo)
            xorl %esi, %esi     # %esi es el indice del arreglo (i)
            xorl %edi, %edi     # %edi es el acumulador
leesum:     cmpl n, %esi        # compara i con n
            je fin              # cuando i=n salta a fin
            pushl %esi
            pushl $leenum
            call printf         # imprime mensaje para lectura de a[i]
            leal x, %ebx
            pushl %ebx
            pushl $leen
            call scanf          # lee a[i]
            addl x, %edi        # suma a[i] a %edi
            incl %esi           # incrementa %esi (i)
            jmp leesum          # vuelve a leer siguiente elemento
fin:        pushl %edi
            pushl $resul
            call printf         # muestra por pantalla la suma
            movl $1, %eax       # fin del programa
            movl $0, %ebx
            int $0x80
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

8.2-

```
# Programa que calcula el promedio de los numeros de un arreglo de
# 20 enteros
.section .data
leenum:      .asciz"Introduzca a[ %d ]: \t"
leen:        .asciz"%d"
x:           .long 0
resul:       .asciz"El promedio de los elementos del arreglo es:  %d\n"

.section .bss
arreglo:     .space 400

.section .text
.globl _start
_start:

             xorl %esi, %esi     # %esi es el indice (i)
             xorl %edi, %edi     # %edi suma elementos del arreglo
leearr:      cmpl $20, %esi      # compara indice con 20
             je fin              # si indice=20 salta a fin
             pushl %esi
             pushl $leenum
             call printf         # imprime mensaje para leer a[i]
             leal x, %ebx
             pushl %ebx
             pushl $leen
             call scanf          # lee a[i]
             addl x, %edi        # suma a[i] en %edi
             incl %esi           # incrementa el indice (i)
             jmp leearr          # vuelve a leer siguiente a[i]
fin:         movl $20, %ebx      # %ebx=20 (tamano del arreglo)
             movl %edi, %eax     # %eax=suma de los elementos del arreglo
             cltd                # expande %eax
             idivl %ebx          # divide %eax/20
             pushl %eax
             pushl $resul
             call printf         # muestra el promedio por pantalla
             movl $1, %eax       # fin del programa
             movl $0, %ebx
             int $0x80
```

8.3-

```
# Programa que calcula la cantidad de numeros pares en un arreglo de n enteros
.section .data
lee:         .asciz"Introduzca el numero de elementos del arreglo: \n"
n:           .long 0
leenum:      .asciz"Introduzca a[ %d ]: \t"
leen:        .asciz"%d"
x:           .long 0
resul:       .asciz"La cantidad de pares en el arreglo es:  %d\n"

.section .bss
arreglo:     .space 400
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
.section .text
.globl _start
_start:
            pushl $lee
            call printf
            leal n, %eax
            pushl %eax
            pushl $leen
            call scanf          # lee n (numeros de elementos del arreglo)
            xorl %esi, %esi     # %esi es el indice (i)
            xorl %edi, %edi     # %edi acumula cantidad de pares
leearr:     cmpl n, %esi        # compara i con n
            je fin              # cuando i=n salta a fin
            pushl %esi
            pushl $leenum
            call printf
            leal x, %ebx
            pushl %ebx
            pushl $leen
            call scanf          # lee a[i]
            movl x, %eax
            cltd
            movl $2, %ecx       # verifica si el numero es par
            idivl %ecx          # divide a[i]/2
            cmpl $0, %edx       # compara el resto con 0
            jne nopar           # si resto es diferente de 0 salta a nopar
            incl %edi           # si el resto es 0 incrementa %edi
nopar:      incl %esi           # incrementa i
            jmp leearr          # vuelve a leer siguiente elemento
fin:        pushl %edi
            pushl $resul
            call printf         # muestra el resultado por pantalla
            movl $1, %eax       # fin del programa
            movl $0, %ebx
            int $0x80
```

8.4-

```
# Programa que calcula la suma de los pares de un arreglo de n enteros
.section .data
lee:        .asciz"Introduzca el numero de elementos del arreglo:\n"
n:          .long 0
leenum:     .asciz"Introduzca a[ %d ]: \t"
leen:       .asciz"%d"
x:          .long 0
resul: .asciz"La suma de los numeros pares del arreglo es:  %d\n"

.section .bss
arreglo:    .space 400

.section .text
.globl _start
_start:
            pushl $lee
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
              call printf
              leal n, %eax
              pushl %eax
              pushl $leen
              call scanf          # lee n (numeros de elementos del arreglo)
              xorl %esi, %esi     # %esi es el indice (i)
              xorl %edi, %edi     # %edi suma los pares
leearr:       cmpl n, %esi        # compara i con n
              je fin              # cuando i=n salta a fin
              pushl %esi
              pushl $leenum
              call printf
              leal x, %ebx
              pushl %ebx
              pushl $leen
              call scanf          # lee a[i]
              movl x, %eax
              cltd
              movl $2, %ecx       # verifica si el numero es par
              idivl %ecx          # divide a[i]/2
              cmpl $0, %edx       # compara el resto con 0
              jne nopar           # si resto es diferente de 0 salta a nopar
              addl x,%edi         # si el resto es 0 suma a[i] a %edi
nopar:        incl %esi           # incrementa i
              jmp leearr          # vuelve a leer siguiente elemento
fin:          pushl %edi
              pushl $resul
              call printf         # muestra el resultado por pantalla
              movl $1, %eax       # fin del programa
              movl $0, %ebx
              int $0x80
```

8.5-

```
# Programa que lee un arreglo de 12 enteros, determina cuál es el mayor e
indica en qué posición se encuentra
.section .data
encab:        .asciz"Este programa lee un arreglo de enteros, determina el mayor
e indica en que posicion se encuentra\n"
pidenum:      .asciz"Introduzca a[ %d ]: \t"
leenum:       .asciz"%d"
num:          .long 0
impmayor:     .asciz"El numero mayor presente en el arreglo es:    %d\n"
impposic:     .asciz"Y se encuentra en la posicion:   %d\n"
tamano:       .long 12

.section .bss
arreglo:      .space 48

.section .text
.globl _start
_start:
              pushl $encab        # imprime el ancabezado
              call printf
              addl $4, %esp
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
# Leemos el arreglo
             xorl %esi, %esi     # %esi es el indice (i)
leearr:      cmpl tamano, %esi   # compara indice con tamano (12)
             je finlee           # si indice=12 salta a fin de lectura
             pushl %esi
             pushl $pidenum
             call printf         # imprime mensaje para leer arreglo[i]
             leal num, %ebx
             pushl %ebx
             pushl $leenum
             call scanf          # lee arreglo[i]
             movl num, %eax      # transfiere num a %eax temporalmente
             movl %eax, arreglo(,%esi,4)      # guarda num en arreglo
             incl %esi           # incrementa el indice (i)
             jmp leearr          # vuelve a leer siguiente a[i]

# Ahora determinamos el mayor y su posicion, %eax va a contener el mayor y %ebx
la posicion
finlee:      xorl %esi, %esi                  # Inicia el indice
             movl arreglo(,%esi,4), %ecx      # lee primer num
             movl %ecx, %eax                  # lo guarda como mayor
             movl %esi, %ebx                  # guarda su posicion
             incl %esi                        # incrementa indice
leesig:      movl arreglo(,%esi,4), %ecx      # lee siguiente numero
             cmpl %eax, %ecx           # compara nuevo num con mayor anterior
             jle noreemplz             # si no es mayor va a noreemplz
             movl %ecx, %eax           # si es mayor reemplaza num
             movl %esi, %ebx           # y la posicion
noreemplz:   incl %esi                 # incrementa el indice
             cmpl tamano,%esi          # verifica fin del arreglo
             je fincomp                # si indice igual a 12 termina
             jmp leesig                # si indice menor que 12 lee siguiente
fincomp:     pushl %eax                # imprime el mayor
             pushl $impmayor
             call printf
             pushl %ebx                # imprime la posicion
             pushl $impposic
             call printf
             movl $1, %eax             # fin del programa
             movl $0, %ebx
             int $0x80
```

Capítulo 9

9.1-

```
# Programa que calcula el promedio de los pares
.section .data
mensaje:     .asciz"Programa que lee un arreglo a de 20 enteros y calcula\n"
mensaj2:     .asciz"el promedio de los numeros pares presentes en el arreglo\n"
leernum:     .asciz"Introduzca a[%d]:\t"
entrada:     .asciz"%d"
salida:      .asciz"Hay %d numeros pares y su promedio es: %d\n"
salto:       .asciz"\n\n"
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
num:          .long 0
pares:        .long 0, 0          # guarda numeros de pares y su sumatoria
                                  # en parea[0]=cantidad de pares y en
                                  # pares[1]=suma de los pares
.section .bss
a:            .space 80           # arreglo a de 20 numeros enteros

.section .text
.globl _start
_start:
              pushl $mensaje      # muestra mensaje inicial
              call printf
              pushl $mensaj2
              call printf
              xorl %esi, %esi
leearr:       cmpl $20, %esi      # ciclo que lee los elementos del arreglo
              je finlee
              pushl %esi
              pushl $leernum
              call printf
              pushl $num
              pushl $entrada
              call scanf
              movl num, %eax
              movl %eax, a(,%esi,4)
              incl %esi
              jmp leearr

finlee:       pushl $a            # dir inicial del arreglo
              pushl $pares        # dir pares
              call par            # retorna cantidad de pares y la
                                  # sumatoria de los pares
              addl $8, %esp

imprime:      pushl pares         # muestra por pantalla cantidad de pares
              leal pares, %eax    # y su sumatoria
              addl $4, %eax
              movl (%eax), %ecx
              pushl %ecx
              pushl $salida
              call printf

              movl $1, %eax # fin del programa
              xorl %ebx, %ebx
              int $0x80

# procedimiento que verifica cuantos numeros son pares y calcula su suma
par:          pushl %ebp
              movl %esp, %ebp
              movl 8(%ebp), %ebx  # dir pares
              movl 12(%ebp),%esi  # dir a
              xorl %ecx, %ecx
              movl $2, %edi
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
verifica:    cmpl $20, %ecx           # compara el indice con tamano de a
             je finverif
             movl (%esi, %ecx, 4), %eax
             incl %ecx
             pushl %eax               # guarda valor leido
             cltd
             idivl %edi               # divide valor entre 2
             cmpl $0, %edx            # compara el resto con 0
             jne verifica             # si resto no es 0 no es par
             incl (%ebx)              # si es par incrementa pares[0]
             popl %eax                # recupera valor leído
             addl %eax, 4(%ebx)       # lo suma a pares[1]
             jmp verifica
finverif:    leave
             ret
```

9.2-

```
# Programa que calcula la suma de los numeros mayores que 10 y menores que 100
presentes en un arreglo de enteros
.section .data
mensaje:     .asciz"Programa que lee un arreglo de n enteros y calcula\n"
mensaj2:     .asciz"la suma de los numeros entre 10 y 100\n"
leertam:     .asciz"Introduzca el numero de elementos del arreglo:\t"
entrada:     .asciz"%d"
leenum:      .asciz"Introduzca los valores del arreglo\n"
salida:      .asciz"La suma de los numeros entre 10 y 100 es: %d\n"
salto: .asciz"\n\n"

n:           .long 0          # tamano del arreglo
num:         .long 0

.section .bss
a:           .space 400       # arreglo a

.section .text
.globl _start
_start:
             pushl $mensaje   # mensaje inicial
             call printf
             pushl $mensaj2
             call printf
             pushl $leertam   # solicita tamano del arreglo
             call printf
             pushl $n
             pushl $entrada
             call scanf       # lee n (tamano del arreglo)
             pushl $salto
             call printf
             pushl $leenum    # solicita al usuario que introduzca los
             call printf      # valores del arreglo
             addl $24, %esp

             xorl %esi, %esi
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
leearr:      cmpl n, %esi        # ciclo que lee los valores del arreglo
             je finlee
             pushl $num
             pushl $entrada
             call scanf
             addl $8, %esp
             movl num, %eax
             movl %eax, a(,%esi,4)
             incl %esi
             jmp leearr

finlee:      pushl $a            # dir inicial del arreglo
             pushl n             # tamano del arreglo
             call suma           # la funcion suma retorna la suma de
             addl $8, %esp # los numeros entre 10 y 100

imprime:     pushl %eax          # muestra el resultado por pantalla
             pushl $salida
             call printf

             movl $1, %eax       # fin del programa
             xorl %ebx, %ebx
             int $0x80

# funcion que retorna la sumatoria de los numeros mayores que 10 y
# menores que 100
suma:        pushl %ebp
             movl %esp, %ebp
             movl 8(%ebp), %ebx         # tamano del arreglo
             movl 12(%ebp),%esi         # dir arreglo
             xorl %eax, %eax            # acumulador para la suma
             xorl %ecx, %ecx            # indice para arreglo
acumul:      cmpl %ebx, %ecx            # compara indice con tamano
             je finacum                 # al leer todos termina
             movl (%esi, %ecx, 4), %edx # lee valor
             incl %ecx                  # incrementa indice
             cmpl $10, %edx             # compara valor leido con 10
             jl acumul                  # si <10 lee siguiente
             cmpl $100, %edx            # si >=10 compara con 100
             jg acumul                  # si >100 lee siguiente
             addl %edx, %eax            # si >=10 y <=100 lo suma
             jmp acumul                 # lee siguiente
finacum:     leave
             ret
```

9.3-

```
# Solución 1:
# Programa que calcula el factorial de un número de manera iterativa

.section .rodata
CAD:   .asciz "Por favor indique un número: "
CADF:  .asciz "%d"
CADR:  .asciz "El factorial del número es: %d\n"
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
.section .bss
NUM:   .space 4

.section .text
.globl _start
_start:
       pushl $CAD          # Pedimos el número
       call printf
       addl $4, %esp

       pushl $NUM          # Lo almacenamos en NUM
       pushl $CADF
       call scanf
       addl $8, %esp

       pushl NUM           # Calculamos el factorial
       call fact
       addl $4, %esp

       pushl %eax          # Mostramos el resultado
       pushl $CADR
       call printf
       addl $8, %esp

       movl $1, %eax              # Terminamos el programa
       movl $0, %ebx
       int $0x80

# Procedimiento iterativo para calcular el factorial de un número

.globl fact
fact:
       pushl %ebp          # Salvaguardamos el marco de pila anterior
       movl %esp, %ebp      # Actualizamos el marco de pila

       pushl %ebx   # Salvaguardar los registros responsabilidad del invocado
       pushl %esi
       pushl %edi

       movl $1, %eax       # El factorial de 0 es 1

for:   cmpl $1, 8(%ebp)    # Si N<=1 terminamos el ciclo
       jle epilogo
       imull 8(%ebp)       # Sino, FACT = FACT * N
       decl 8(%ebp)        # N--
       jmp for                    # Repetimos el ciclo

epilogo:
       popl %edi           # Restauramos los registros salvaguardados
       popl %esi
       popl %ebx

       leave               # se prepara la pila para el retorno
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
      ret                 # Devolver el control al invocador


# Solución 2:
# Programa que calcula el factorial de un número de manera recursiva

.section .rodata
CAD:   .asciz "Por favor indique un número: "
CADF:  .asciz "%d"
CADR:  .asciz "El factorial del número es: %d\n"

.section .bss
NUM:   .space 4

.section .text
.globl _start
_start:
      pushl $CAD          # Pedimos el número
      call printf
      addl $4, %esp

      pushl $NUM          # Lo almacenamos en NUM
      pushl $CADF
      call scanf
      addl $8, %esp

      pushl NUM           # Calculamos el factorial
      call fact
      addl $4, %esp

      pushl %eax          # Mostramos el resultado
      pushl $CADR
      call printf
      addl $8, %esp

      movl $1, %eax             # Terminamos el programa
      movl $0, %ebx
      int $0x80

# Procedimiento recursivo para calcular el factorial de un número

.globl fact
fact:
      pushl %ebp          # Salvaguardamos el marco de pila anterior
      movl %esp, %ebp      # Actualizamos el marco de pila

      pushl %ebx   # Salvaguardar los registros responsabilidad del invocado
      pushl %esi
      pushl %edi

      cmpl $1, 8(%ebp)    # Si N<=1 caso base, FACT = 1
      jle base
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
      movl 8(%ebp), %edx  # Sino, FACT = N * FACT(N-1)
      decl %edx
      pushl %edx
      call fact
      addl $4, %esp
      imull 8(%ebp)
      jmp epilogo

base:  movl $1, %eax

epilogo:
      popl %edi           # Restauramos los registros salvaguardados
      popl %esi
      popl %ebx

      leave               # se prepara la pila para el retorno
      ret                 # Devolver el control al invocador
```

9.4.-

```
# Programa que genera la serie de Fibonacci
.section .data
mensaje:      .asciz"Programa que muestra por pantalla la serie de Fibonacci
hasta n numeros\n"
leern:        .asciz"Introduzca n:\t"
error:        .asciz"El n es cero, por lo tanto no se muestra la serie\n"
entrada:      .asciz"%d"
salida:       .asciz"La serie de Fibonacci para los %d primeros numeros es:\n"
salid2:       .asciz"%d\t"
salto:        .asciz"\n\n"

n:            .long 0

.section .bss
arrfib:       .space 400          # arrfib arreglo para guardar la serie

.section .text
.globl _start
_start:
              pushl $mensaje      # muestra mensaje inicial
              call printf
              pushl $leern
              call printf
              pushl $n
              pushl $entrada
              call scanf          # lee n, cantidad de elementos de la
                                  # serie a mostrar
              addl $16, %esp
              cmpl $0, n          # compara n con 0
              jg mayor0
              pushl $error # si n=0 se muestra el mensaje de error
              call printf
              jmp fin
mayor0:       pushl n             # pasa el numero n por parametro
              pushl $arrfib       # pasa dir del arreglo arrfib por parametro
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
            call fib            # llama a fib para generar la serie
            addl $8, %esp
            pushl n
            pushl $salida
            call printf         # muestra mensaje de salida
            addl $8, %esp
            xorl %edi, %edi
            movl n, %esi
serie:      cmpl %esi, %edi             # compara indice con n
            je fin
            movl arrfib(,%edi,4), %eax
            incl %edi
            pushl %eax                  # muestra la serie por pantalla
            pushl $salid2
            call printf
            addl $8, %esp
            jmp serie
fin:        pushl $salto                # imprime un salto de linea
            call printf
            movl $1, %eax               # fin del programa
            xorl %ebx, %ebx
            int $0x80

# Procedimiento que genera la serie de Fibonacci

fib:        pushl %ebp
            movl %esp, %ebp
            movl 8(%ebp), %edx          # %edx=dir arreglo
            movl 12(%ebp), %ebx         # %ebx=n (n>0)
            xorl %ecx, %ecx             # %ecx=0 indice del arreglo
            movl $1, (%edx, %ecx, 4)    # arrfib[0]=1
            cmpl $1, %ebx
            je finfib                   # si n=1 salta a finfib
            incl %ecx
            movl $1, (%edx, %ecx, 4)    # arrfib[1]=1
            incl %ecx
            movl $1, %edi               # %edi=1 (arrfib[0])
            movl $1, %esi               # %esi=1 (arrfib[1])
ciclo:      cmpl %ebx, %ecx             # compara indice con n
            je finfib
            movl %edi, %eax
            addl %esi, %eax             # suma dos valores anteriores
            movl %eax, (%edx, %ecx, 4) # agrega siguiente elemento
            incl %ecx                   # incrementa indice
            movl %esi, %edi             # actualiza registro temporal
            movl %eax, %esi             # actualiza registro temporal
            jmp ciclo
finfib:     leave
            ret
```

Capitulo 10

10.1.-

```
# Este programa solicita al usuario una contrasena y la compara
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
# con la cadena de caracteres declarada en la seccion de datos,
# indica si la contrasena es valida o no
.section .data
contras:            .asciz"clave1234"
largo:       .long 9
escriba:            .asciz"Por favor introduzca la contrasena: "
format1:            .asciz"%s"
resultig:           .asciz"La contrasena es correcta\n"
resultno:           .asciz"La contrasena es incorrecta\n"

.section .bss
clave:       .space 9

.section .text
.globl _start
_start:
       pushl $escriba
       call printf
       addl $8, %esp
       leal clave, %edx
       pushl %edx
       pushl $format1
       call scanf
       cld                          # limpia la bandera DF
       leal contras, %edi           # actualiza %edi con inicio de la contrasena
       leal clave, %esi             # actualiza %esi con inicio de la clave leida
       movl largo, %ecx             # actualiza %ecx con el numero de caracteres
       repe cmpsb                   # compara 1 caracter a la vez
       jne error             # si es incorrecta va a error
       pushl $resultig       # si es correcta imprime el mensaje correspondiente
       call printf
       jmp fin
error: pushl $resultno              # imprime mensaje de error
       call printf
fin:   movl $1, %eax                # fin del programa
       movl $0, %ebx
       int $0x80
```

10.2.-

```
# Este programa solicita una cadena de caracteres y revisa la
# existencia del caracter "-" . Si esta presenta indica la ,
# posicion de la primera aparicion del caracter
.section .data
mensaj1:     .asciz"Este programa revisa la existencia del caracter  '-'  en
una cadena\n"
caracter:    .asciz"-"
escriba:            .asciz"Por favor introduzca la cadena: "
format1:            .asciz"%s"
salida1:            .asciz"El caracter  '-' no esta presente en la cadena\n"
salida2:            .asciz"El caracter  '-' se encuentra en la posicion %d de
la cadena\n"
largo:       .long 0

.section .bss
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
cadena:             .space 100

.section .text
.globl _start
_start:
            pushl $mensaj1
            call printf
            pushl $escriba
            call printf
            addl $8, %esp
            leal cadena, %edx
            pushl %edx
            pushl $format1
            call scanf          # lee la cadena

            cld                 # limpia la bandera DF
            leal cadena, %edi   # actualiza %edi con inicio de la cadena leida
# determinar el tamano de la cadena leida
            movl $100, %ecx
            movb $0, %al
            repne scasb
            jne fin             # caso de excepcion
            subl $100, %ecx
            neg %ecx
            decl %ecx
            movl %ecx, largo
# comparar la cadena leida con el caracter
            leal cadena, %edi
            leal caracter, %esi    # actualiza %esi con el inicio del caracter
            lodsb
            repne scasb
            jne noesta
            subl largo, %ecx
            neg %ecx
            pushl %ecx
            pushl $salida2
            call printf
            jmp fin
noesta:     pushl $salida1
            call printf

fin:        movl $1, %eax       # fin del programa
            movl $0, %ebx
            int $0x80
```

10.3.-

```
# Este programa cuenta el numero de ocurrencias del caracter 'y'
# dentro de una cadena de caracteres
.section .data
mensaj1:    .asciz"Este programa cuenta el numero de ocurrencias del caracter
'y'\n"
leecad:             .asciz"Por favor introduzca la cadena: "
y:                  .asciz"y"
format1:            .asciz"%s"
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
salida:             .asciz"El caracter  'y'  esta presente %d veces en la
cadena\n"

.section .bss
cadena:             .space 100

.section .text
.globl _start
_start:
             pushl $mensaj1
             call printf
             pushl $leecad
             call printf
             addl $8, %esp
             leal cadena, %edx
             pushl %edx
             pushl $format1
             call scanf                  # lee la cadena
             addl $8, %esp
             cld
# calcula largo de la cadena
             leal cadena, %edi
             movl $100, %ecx
             movb $0, %al
             repne scasb
             jne fin                     # caso de excepcion
             subl $100, %ecx
             neg %ecx
             decl %ecx                   # %ecx tiene el largo de la cadena
             xorl %ebx, %ebx
             leal cadena, %edi
             leal y, %esi
             lodsb                       # carga y en %al
revisa:      repne scasb
             jne termina
             incl %ebx
             jmp revisa
termina:
             pushl %ebx
             pushl $salida
             call printf

fin:         movl $1, %eax               # fin del programa
             movl $0, %ebx
             int $0x80
```

Capitulo 11

11.1.-

```
# Programa que calcula la nota
.section .data
mensaje: .asciz"Programa que calcula la nota definitiva, teniendo como entrada 6
notas parciales.\n"
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
mensaje2:     .asciz"Muestra la nota final con decimales y la definitiva
redondeada\n"
entrada1:     .asciz"Introduzca las notas:\n"
nota1:        .asciz"Nota 1 (valor: 20%): "
nota2:        .asciz"Nota 2 (valor: 15%): "
nota3:        .asciz"Nota 3 (valor: 25%): "
nota4:        .asciz"Nota 4 (valor: 10%): "
nota5:        .asciz"Nota 5 (valor: 25%): "
nota6:        .asciz"Nota 6 (valor:  5%): "
leenota: .asciz"%d"
salto:        .asciz"\n"
n1:           .long 0
n2:           .long 0
n3:           .long 0
n4:           .long 0
n5:           .long 0
n6:           .long 0
porc20:       .float .20
porc15:       .float .15
porc25:       .float .25
porc10:       .float .10
porc5:        .float .05
result2: .quad 0
forma1:       .asciz"Nota final: %.2f\n"     # formato de salida con dos
                                                 # decimales
forma2:       .asciz"Nota definitiva: %d\n"  # formato de salida entero
redondeo:     .byte 0x7f, 0x0b               # para actualizar el
                                                 # redondeo hacia arriba

.section .text
.globl _start
_start:
              pushl $mensaje        # mensaje inicial
              call printf
              pushl $mensaje2
              call printf
              pushl $salto
              call printf
              pushl $entrada1
              call printf
              addl $16, %esp

              pushl $nota1      # lee nota 1
              call printf
              pushl $n1
              pushl $leenota
              call scanf
              pushl $salto
              call printf
              addl $16, %esp

              pushl $nota2      # lee nota 2
              call printf
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
pushl $n2
pushl $leenota
call scanf
pushl $salto
call printf
addl $16, %esp

pushl $nota3      # lee nota 3
call printf
pushl $n3
pushl $leenota
call scanf
pushl $salto
call printf
addl $16, %esp

pushl $nota4      lee nota 4
call printf
pushl $n4
pushl $leenota
call scanf
pushl $salto
call printf
addl $16, %esp

pushl $nota5      lee nota 5
call printf
pushl $n5
pushl $leenota
call scanf
pushl $salto
call printf
addl $16, %esp

pushl $nota6      # lee nota 6
call printf
pushl $n6
pushl $leenota
call scanf
pushl $salto
call printf
addl $16, %esp

finit                 # inicializa FPU
fldcw redondeo        # cambia a redondeo hacia arriba
filds n1          # carga nota 1
fmuls porc20      # multiplica nota 1 * 20%
filds n2          # carga nota 2
fmuls porc15      # multiplica nota 2 * 15%
filds n3          # carga nota 3
fmuls porc25      # multiplica nota 3 * 25%
filds n4          # carga nota 4
fmuls porc10      # multiplica nota 4 * 10%
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
            filds n5         # carga nota 5
            fmuls porc25     # multiplica nota 5 * 25%
            filds n6         # carga nota 6
            fmuls porc5          # multiplica nota 6 * 5%
            faddp                # suma los valores de las notas
            faddp
            faddp
            faddp
            faddp

            fistl result2    # guarda el resultado

            subl $8, %esp    # muestra el resultado con 2 decimales
            fstpl (%esp)
            pushl $forma1
            call printf
            addl $12, %esp

            pushl result2    # muestra el resultado redondeado
            pushl $forma2
            call printf
            addl $8, %esp


            movl $1, %eax    # fin del programa
            xorl %ebx, %ebx
            int $0x80
```

11.2.-

```
.section .data
mensaje: .asciz"Programa que convierte grados Centigrados a Fahrenheit\n"
introd:     .asciz"Introduzca los grados Centigrados: "
leegrado:   .asciz"%f"
salto:      .asciz"\n"
grados:     .float 0.0
nueve:      .float 9.0
cinco:      .float 5.0
treinta2:   .float 32.0
salida:     .asciz"Grados Fahrenheit= %.2f\n"


.section .text
.globl _start
_start:
            pushl $mensaje
            call printf
            pushl $introd
            call printf
            addl $8, %esp

            pushl $grados
            pushl $leegrado
            call scanf
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
            pushl $salto
            call printf

            finit
            flds grados
            fdivs cinco
            fmuls nueve
            fadds treinta2

            subl $8, %esp
            fstpl (%esp)
            pushl $salida
            call printf

            movl $1, %eax
            xorl %ebx, %ebx
            int $0x80
```

11.3.-

```
.section .data
mensaje: .asciz"Programa que calcula el promedio de temperatura\n"
mensaje2:    .asciz"dados 10 valores en grados Centigrados\n"
pidevalor:   .asciz"Introduzca el valor %d: "

leegrado:    .asciz"%f"
salto:       .asciz"\n"
grado:       .float 0.0
cero:        .float 0.0
diez:        .float 10.0
salida:      .asciz"El promedio de temperatura es: %.4f\n"


.section .text
.globl _start
_start:
            pushl $mensaje
            call printf
            pushl $mensaje2
            call printf
            addl $8, %esp

            finit
            flds cero
            xorl %edi, %edi
            incl %edi
leeysuma:   cmpl $10, %edi
            jg termina
            pushl %edi
            pushl $pidevalor
            call printf
            addl $8, %esp
            incl %edi
            pushl $grado
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
              pushl $leegrado
              call scanf
              addl $8, %esp
              fadds grado
              jmp leeysuma

termina: fdivs diez

              subl $8, %esp
              fstpl (%esp)
              pushl $salida
              call printf

              movl $1, %eax
              xorl %ebx, %ebx
              int $0x80
```

11.4.-

```
.section .data
mensaje: .asciz"Programa que calcula el area de un circulo dado su diametro\n"
introd:       .asciz"Introduzca el diametro: "
leediam: .asciz"%f"
salto:        .asciz"\n"
diam:         .float 0.0
dos:     .float 2.0
salida:       .asciz"Area del circulo= %.4f\n"


.section .text
.globl _start
_start:
              pushl $mensaje
              call printf
              pushl $introd
              call printf
              addl $8, %esp

              pushl $diam
              pushl $leediam
              call scanf

              pushl $salto
              call printf

              finit
              flds diam              # apila diametro
              fdivs dos              # divide diam/2
              fst %st(1)
              fmul %st(1), %st(0)
              fldpi                  # apila pi
              fmul %st(1), %st(0)
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
            subl $8, %esp
            fstpl (%esp)
            pushl $salida
            call printf

            movl $1, %eax
            xorl %ebx, %ebx
            int $0x80
```

11.5.-

```
.section .data
mensaje: .asciz"Programa que convierte kilometros en millas o millas en
kilometros\n"

tipoconv:   .asciz"Indique el tipo de conversion que desea realizar:\n"
tipo1:      .asciz"Convertir de kilometros a millas [k] o de millas a
kilometros [m]: "
leetipo: .asciz"%c"
salto:      .asciz"\n"
kiloms:     .asciz"Introduzca la cantidad de kilometros: "
millas:     .asciz"Introduzca la cantidad de millas: "
formatof:   .asciz"%f"
km:         .float 0.0
milla:      .float 0.0
convers: .float 0.62
kmmilla: .asciz"Son %.2f millas\n"
millakm: .asciz"Son %.2f kilometros\n"

.section .bss
tipo:       .space 1

.section .text
.globl _start
_start:
            pushl $mensaje
            call printf
            pushl $tipoconv
            call printf
pidetipo:   pushl $tipo1
            call printf
            addl $12, %esp

            pushl $tipo
            pushl $leetipo
            call scanf
            addl $8, %esp

            cmpl $'k', tipo
            je kilo
            cmpl $'K', tipo
            je kilo
            cmpl $'m', tipo
            je mill
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
            cmpl $'M', tipo
            je mill
            jmp pidetipo

            pushl $salto
            call printf
            addl $4, %esp

            finit

kilo:       pushl $kiloms
            call printf
            addl $4, %esp
            pushl $km
            pushl $formatof
            call scanf
            addl $8, %esp
            flds km
            fmuls convers

            subl $8, %esp
            fstpl (%esp)
            pushl $kmmilla
            call printf
            jmp fin

mill:       pushl $millas
            call printf
            addl $4, %esp
            pushl $milla
            pushl $formatof
            call scanf
            addl $8, %esp
            flds milla
            fdivs convers

            subl $8, %esp
            fstpl (%esp)
            pushl $millakm
            call printf


fin:    movl $1, %eax
            xorl %ebx, %ebx
            int $0x80
```

Capitulo 12

12.1.-

```
# Utilice el siguiente script para desactivar el buffering y echo del terminal
# more.sh:
# stty -icanon -echo
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
# ./12-1 $1
# stty icanon echo

.section .rodata
cad:     .asciz "Ocurrió un error al intentar abrir el archivo especificado\n"
formato: .asciz "%s"
modor:   .asciz "r"

.section .bss
fd:      .space 4
buffer:  .space 80
char:    .space 1

.section .text
.globl _start
_start:

    movl 8(%esp), %eax    # Argumento del programa: Nombre del archivo

    # Abre el archivo en modo lectura
    pushl $modor          # Modo: r
    pushl %eax            # Archivo fuente
    call fopen            # Llama a fopen
    addl $8, %esp
    mov %eax, fd          # Apuntador al archivo abierto

    cmpl $0, fd           # Si fd==NULL (0) ocurrió un error
    je error

    xorl %edi, %edi       # lineas = 0
rep:
    # Lee una linea del archivo
    pushl fd              # Apuntador al archivo abierto
    pushl $80             # Cantidad de caracteres a leer (máximo)
    pushl $buffer         # Apuntador al buffer
    call fgets            # Llama a fscanf
    addl $12, %esp

    # Se muestra por la pantalla
    pushl $buffer         # Apuntador al buffer
    pushl $formato        # Cadena de formato
    call printf
    addl $8, %esp

    # Si se alcanzó el final del archivo sale del ciclo
    pushl fd              # Apuntador al archivo abierto
    call feof             # Llama a feof
    addl $4, %esp
    cmpl $0, %eax         # Si se alcanzó el final del archivo
    jne fin               # Termina el programa

    incl %edi             # lineas = lineas + 1
    cmpl $25,%edi         # Si se han mostrado 25 lineas
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    je espera               # Espera por el usuario
    jmp rep                 # Siguiente línea del archivo


espera:
    call getchar            # Obtiene la entrada del usuario
    movb %al, char

    cmpb $'\n', char        # Si pisó ENTER
    je avanzar1             # Avanza una línea
    cmpb $' ', char         # Si pisó la barra espaciadora
    je avanzar25            # Avanza una página (25 líneas)
    jmp espera

avanzar1:
    decl %edi               # lineas = lineas - 1
    jmp rep

avanzar25:
    xorl %edi, %edi         # lineas = 0
    jmp rep


error:   pushl $cad         # Informa al usuario del error
    call printf
    addl $4, %esp
    jmp fin2

fin: # Cierra el archivo si está abierto

    cmpl $0, fd
    je fin2
    pushl fd                # Descriptor del archivo a cerrar
    call fclose             # Llama a fclose
    addl $4, %esp

    # Termina el programa
fin2:
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

12.2.-

```
# Utilice el siguiente script para desactivar el buffering y echo del terminal
# more.sh:
# stty -icanon -echo
# ./12-2 $1
# stty icanon echo
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
.section .rodata
cad:     .asciz "Ocurrió un error al intentar abrir el archivo especificado\n"
formato: .asciz "%s"
modor:   .asciz "r"

.section .bss
fd:      .space 4
buffer:  .space 80
char:    .space 1

.section .text
         .globl _start
_start:

    # Abre el archivo en modo lectura
    movl $5, %eax              # Servicio #5 (open)
    movl 8(%esp), %ebx         # Ruta del archivo a abrir
    movl $0, %ecx              # ORDONLY : 0
    int $0x80                  # Llamada al SO
    mov %eax, fd               # Apuntador al archivo abierto

    cmpl $0, fd                # Si fd<0 ocurrió un error
    jl error

    xorl %edi, %edi            # lineas = 0
rep:
    # Lee una linea del archivo
    xorl %esi, %esi
leer:
    movl $3, %eax              # Servicio #3, (read)
    movl fd, %ebx              # File Descriptor del archivo abierto
    leal buffer(%esi), %ecx    # Dirección donde se almacenarán los datos
    movl $1, %edx              # Número de bytes a leer
    int $0x80                  # Llamada al SO

    cmpl $0, %eax              # Si se leyeron cero bytes
    je fin                     # Se alcanzó el final del archivo
    cmpb $'\n',buffer(%esi)    # Si se leyó un salto de linea
    je mostrar                 # Muestra la línea almacenada

    incl %esi                  # lineas = lineas + 1
    jmp leer

mostrar:
    movb $0,buffer+1(%esi)     # Cerrar la cadena leída

    # Se muestra por la pantalla
    pushl $buffer              # Apuntador al buffer
    pushl $formato             # Cadena de formato
    call printf                # Llama a printf
    addl $8, %esp

    incl %edi                  # lineas = lineas + 1
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    cmpl $25,%edi               # Si se han mostrado 25 lineas
    je espera                   # Espera por el usuario
    jmp rep                     # Siguiente línea del archivo


espera:
    call getchar                # Obtiene la entrada del usuario
    movb %al, char

    cmpb $'\n', char            # Si pisó ENTER
    je avanzar1                 # Avanza una linea
    cmpb $' ', char             # Si pisó la barra espaciadora
    je avanzar25                # Avanza una página (25 líneas)
    jmp espera

avanzar1:
    decl %edi                   # lineas = lineas - 1
    jmp rep

avanzar25:
    xorl %edi, %edi             # lineas = 0
    jmp rep

error:
    pushl $cad              # Informa al usuario del error
    call printf
    addl $4, %esp
    jmp fin2

fin: # Cierra el archivo si está abierto

    cmpl $0, fd
    je fin2
    movl $6, %eax               # Servicio #6 (close)
    movl fd, %ebx               # File Descriptor del archivo abierto
    int $0x80                   # Llamada al SO

    # Termina el programa
fin2:
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

12.3.-

```
.section .rodata
cad1:    .asciz "1 Insertar contacto\n2 Buscar contacto\n3 Eliminar contacto\n"
cad2:    .asciz "Su opción: "
cad3:    .asciz "Ocurrió un error al intentar abrir el archivo\n"
cad4:    .asciz "Nombre del contacto: "
cad5:    .asciz "Teléfono del contacto: "
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
cad6:    .asciz "Posición: "
cad7:    .asciz "Nombre: %s\nTelefono: %d\n"
cad8:    .asciz "La posición indicada está vacía\n"
cad9:    .asciz "Ocurrió un error al intentar abrir el archivo temporal\n"
file:    .asciz "agenda.txt"
file2:   .asciz "agenda.tmp"
format1: .asciz "%d"
format2: .asciz "%s"
format3: .asciz "%s | %d\n"
modo1:   .asciz "a"
modo2:   .asciz "r"
modo3:   .asciz "w"


.section .bss
nombre:  .space 80
tlf:     .space 4
opcion:  .space 4
pos:     .space 4
desp:    .space 4
fd:      .space 4
fd2:     .space 4


.section .text
.globl _start
_start:
    pushl $cad1               # Muestra las opciones
    call printf
    addl $4, %esp

while:
    pushl $cad2               # Pide una opción
    call printf
    addl $4, %esp

    pushl $opcion             # Lee la respuesta del usuario
    pushl $format1
    call scanf
    addl $8, %esp

    cmpl $1, opcion           # Si es 1 salta a insertar
    je insertar
    cmpl $2, opcion           # Si es 2 salta a buscar
    je buscar
    cmpl $3, opcion           # Si es 3 salta a eliminar
    je eliminar
    jmp while                 # En otro caso pregunta de nuevo


    # Abre el archivo para escritura al final
insertar:
    pushl $modo1              # Modo: a
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    pushl $file                 # Archivo: agenda.txt
    call fopen                  # Llama a fopen
    addl $8, %esp

    mov %eax, fd                # Apuntador al archivo abierto

    cmpl $0, fd                 # Si fd==NULL (0) ocurrió un error
    je error

    pushl $cad4                 # Pregunta por el nombre del contacto
    call printf
    addl $4, %esp

    pushl $nombre               # Lee la respuesta del usuario
    pushl $format2
    call scanf
    addl $8, %esp

    pushl $cad5                 # Pregunta por el telefono del contacto
    call printf
    addl $4, %esp

    pushl $tlf                  # Lee la respuesta del usuario
    pushl $format1
    call scanf
    addl $8, %esp

    # Inserta el contacto en el archivo

    pushl tlf                   # Telefono
    pushl $nombre               # &Nombre
    pushl $format3              # Cadena de formato
    pushl fd                    # Apuntador al archivo abierto
    call fprintf                # Llama a fprintf
    addl $16, %esp

    jmp fin                     # Salta al final del programa


    # Abre el archivo para lectura desde el comienzo
buscar:
    pushl $modo2                # Modo: r
    pushl $file                 # Archivo: agenda.txt
    call fopen                  # Llama a fopen
    addl $8, %esp

    mov %eax, fd                # Apuntador al archivo abierto

    cmpl $0, fd                 # Si fd==NULL (0) ocurrió un error
    je error

    pushl $cad6                 # Pide la posición del contacto
    call printf
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    addl $4, %esp

    pushl $pos                  # Lee la respuesta del usuario
    pushl $format1
    call scanf
    addl $8, %esp

    xorl %esi, %esi             # i=0
rep:
    cmpl %esi, pos              # Si i=pos
    je mostrar                  # Salta a mostrar

    # Si se alcanzó el final del archivo la posición no existe
    pushl fd                    # Apuntador al archivo abierto
    call feof                   # Llama a feof
    addl $4, %esp
    cmpl $0, %eax
    jne nopos

    # Lee el contacto desde el archivo
    pushl $tlf                  # &Telefono
    pushl $nombre               # &Nombre
    pushl $format3              # Cadena de formato
    pushl fd                    # Apuntador al archivo abierto
    call fscanf                 # Llama a fscanf
    addl $16, %esp

    incl %esi                   # i++
    jmp rep                     # Repite la búsqueda

mostrar:
    pushl tlf                   # Muestra la información del contacto
    pushl $nombre
    pushl $cad7
    call printf
    addl $12, %esp

    jmp fin                     # Salta al final del programa


    # Abre el archivo para lectura desde el comienzo
    # Y un archivo secundario para escritura de los contactos
eliminar:
    pushl $modo2                # Modo: r
    pushl $file                 # Archivo: agenda.txt
    call fopen                  # Llama a fopen
    addl $8, %esp
    mov %eax, fd                # Apuntador al archivo original
    cmpl $0, fd                 # Si fd==NULL (0) ocurrió un error
    je error

    pushl $modo3                # Modo: w
    pushl $file2                # Archivo: agenda.tmp
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    call fopen                  # Llama a fopen
    addl $8, %esp
    mov %eax, fd2               # Apuntador al archivo temporal
    cmpl $0, fd2                # Si fd2==NULL (0) ocurrió un error
    je error2

    pushl $cad6                 # Pide la posición del contacto
    call printf
    addl $4, %esp

    pushl $pos                  # Lee la respuesta del usuario
    pushl $format1
    call scanf
    addl $8, %esp

    movl $1, %esi               # i=1
copiar:
    # Si se alcanzó el final del archivo finalizó la copia
    pushl fd                    # Apuntador al archivo abierto
    call feof                   # Llama a feof
    addl $4, %esp
    cmpl $0, %eax
    jne fcopiar

    # Lee un contacto desde el archivo original
    pushl $tlf                  # &Telefono
    pushl $nombre               # &Nombre
    pushl $format3              # Cadena de formato
    pushl fd                    # Apuntador al archivo original
    call fscanf                 # Llama a fscanf
    addl $16, %esp

    cmpl %esi, pos              # Si i=pos
    je saltar                   # No copia el contacto

    # Se inserta en el archivo temporal
    pushl tlf                   # Telefono
    pushl $nombre               # &Nombre
    pushl $format3              # Cadena de formato
    pushl fd2                   # Apuntador al archivo temporal
    call fprintf                # Llama a fprintf
    addl $16, %esp

saltar:
    incl %esi                   # i++
    jmp copiar

fcopiar:
    # Cierra ambos archivos
    pushl fd                    # Descriptor del archivo a cerrar
    call fclose                 # Llama a fclose
    addl $4, %esp
    pushl fd2                   # Descriptor del archivo a cerrar
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    call fclose                # Llama a fclose
    addl $4, %esp

    # Elimina el archivo original
    pushl $file                # Ruta del archivo a eliminar
    call remove                # Llama a remove
    addl $4, %esp

    # Lo sustituye por el archivo temporal
    pushl $file                # Ruta nueva
    pushl $file2               # Ruta antigüa
    call rename                # Llama a rename
    addl $8, %esp

    jmp fin2                   # Salta al final del programa


    # La posición solicitada no existe
nopos:
    pushl $cad8                # Informa al usuario del problema
    call printf
    addl $4, %esp
    jmp fin                    # Salta al final del programa


    # Ocurrió un error al abrir el archivo
error:
    pushl $cad3                # Informa al usuario del error
    call printf
    addl $4, %esp
    jmp fin2

error2:
    pushl $cad9                # Informa al usuario del error
    call printf
    addl $4, %esp

fin:
    # Cierra el archivo si estaba abierto
    pushl fd                   # Descriptor del archivo a cerrar
    call fclose                # Llama a fclose
    addl $4, %esp

    # Termina el programa
fin2:
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

12.4.-

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
.section .rodata
cad1:    .asciz "1 Insertar contacto\n2 Buscar contacto\n3 Eliminar contacto\n"
cad2:    .asciz "Su opción: "
cad3:    .asciz "Ocurrió un error al intentar abrir el archivo\n"
cad4:    .asciz "Nombre del contacto: "
cad5:    .asciz "Teléfono del contacto: "
cad6:    .asciz "Posición: "
cad7:    .asciz "Nombre: %s\nTelefono: %d\n"
cad8:    .asciz "La posición indicada está vacía\n"
cad9:    .asciz "Ocurrió un error al intentar abrir el archivo temporal\n"
file:    .asciz "agenda3.dat"
file2:   .asciz "agenda3.tmp"
format1:.asciz "%d"
format2:.asciz "%s"

.section .bss
nombre:  .space 80
tlf: .space 4
opcion:  .space 4
pos: .space 4
desp:    .space 4
fd:  .space 4
fd2: .space 4


.section .text
.globl _start
_start:

    pushl $cad1                # Muestra las opciones
    call printf
    addl $4, %esp

while:
    pushl $cad2                # Pide una opción
    call printf
    addl $4, %esp

    pushl $opcion              # Lee la respuesta del usuario
    pushl $format1
    call scanf
    addl $8, %esp

    cmpl $1, opcion            # Si es 1 salta a insertar
    je insertar
    cmpl $2, opcion            # Si es 2 salta a buscar
    je buscar
    cmpl $3, opcion            # Si es 3 salta a eliminar
    je eliminar
    jmp while                  # En otro caso pregunta de nuevo


    # Abre el archivo para escritura al final
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
insertar:
    movl $5, %eax              # Servicio #5 (open)
    movl $file, %ebx           # Ruta del archivo a abrir
    movl $02101, %ecx          # O_CREAT | O_APPEND | O_WRONLY
    movl $0644, %edx           # Permisos del archivo
    int $0x80                  # Llamada al SO
    mov %eax, fd               # Apuntador al archivo abierto

    cmpl $0, fd                # Si fd<0 ocurrió un error
    jl error

    pushl $cad4                # Pregunta por el nombre del contacto
    call printf
    addl $4, %esp

    pushl $nombre              # Lee la respuesta del usuario
    pushl $format2
    call scanf
    addl $8, %esp

    pushl $cad5                # Pregunta por el telefono del contacto
    call printf
    addl $4, %esp

    pushl $tlf                 # Lee la respuesta del usuario
    pushl $format1
    call scanf
    addl $8, %esp

    # Inserta el contacto en el archivo
    movl $4, %eax              # Servicio #4, (write)
    movl fd, %ebx              # File Descriptor del archivo abierto
    movl $nombre, %ecx         # Apuntador a los datos a escribir
    movl $84, %edx             # Número de bytes a escribir
    int $0x80                  # Llamada al SO

    jmp fin                    # Salta al final del programa


    # Abre el archivo para lectura desde el comienzo
buscar:
    movl $5, %eax              # Servicio #5 (open)
    movl $file, %ebx           # Ruta del archivo a abrir
    movl $0, %ecx              # ORDONLY : 0
    int $0x80                  # Llamada al SO
    mov %eax, fd               # Apuntador al archivo abierto

    cmpl $0, fd                # Si fd<0 ocurrió un error
    jl error

    pushl $cad6                # Pide la posición del contacto
    call printf
    addl $4, %esp
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    pushl $pos                 # Lee la respuesta del usuario
    pushl $format1
    call scanf
    addl $8, %esp

    decl pos                   # desplazamiento = (posición-1)*84
    movl $84, %eax
    imull pos
    movl %eax, desp

    # Localiza el contacto dentro del archivo
    movl $19, %eax             # Servicio #19, (lseek)
    movl fd, %ebx              # File Descriptor del archivo abierto
    movl desp, %ecx            # Desplazamiento en bytes
    movl $0, %edx              # A partir del comienzo (SEEK_SET : 0)
    int $0x80                  # Llamada al SO

    # Lee el contacto desde el archivo
    movl $3, %eax              # Servicio #3, (read)
    movl fd, %ebx              # File Descriptor del archivo abierto
    movl $nombre, %ecx         # Dirección donde se almacenarán los datos
    movl $84, %edx             # Número de bytes a leer
    int $0x80                  # Llamada al SO

    cmpl $0, %eax              # Si se leyeron cero bytes
    je nopos                   # la posición no existe

    pushl tlf                  # Muestra la información del contacto
    pushl $nombre
    pushl $cad7
    call printf
    addl $12, %esp

    jmp fin                    # Salta al final del programa

    # Abre el archivo para lectura desde el comienzo
    # Y un archivo secundario para escritura de los contactos
eliminar:
    movl $5, %eax              # Servicio #5 (open)
    movl $file, %ebx           # Ruta del archivo a abrir
    movl $0, %ecx              # ORDONLY : 0
    int $0x80                  # Llamada al SO
    mov %eax, fd               # Apuntador al archivo abierto
    cmpl $0, fd                # Si fd<0 ocurrió un error
    jl error

    movl $5, %eax              # Servicio #5 (open)
    movl $file2, %ebx          # Ruta del archivo a abrir
    movl $02101, %ecx          # O_CREAT | O_APPEND | O_WRONLY
    movl $0644, %edx           # Permisos del archivo
    int $0x80                  # Llamada al SO
    mov %eax, fd2              # Apuntador al archivo abierto
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    cmpl $0, fd2                # Si fd2<0 ocurrió un error
    jl error2

    pushl $cad6                 # Pide la posición del contacto
    call printf
    addl $4, %esp

    pushl $pos                  # Lee la respuesta del usuario
    pushl $format1
    call scanf
    addl $8, %esp

    movl $1, %esi               # i=1
copiar:
    # Lee un contacto desde el archivo original
    movl $3, %eax               # Servicio #3, (read)
    movl fd, %ebx               # File Descriptor del archivo abierto
    movl $nombre, %ecx          # Dirección donde se almacenarán los datos
    movl $84, %edx              # Número de bytes a leer
    int $0x80                   # Llamada al SO

    # Si se alcanzó el final del archivo finalizó la copia
    cmpl $0, %eax
    je fcopiar

    cmpl %esi, pos              # Si i=pos
    je saltar                   # No copia el contacto

    # Se inserta en el archivo temporal
    movl $4, %eax               # Servicio #4, (write)
    movl fd2, %ebx              # File Descriptor del archivo abierto
    movl $nombre, %ecx          # Apuntador a los datos a escribir
    movl $84, %edx              # Número de bytes a escribir
    int $0x80                   # Llamada al SO

saltar:
    incl %esi                   # i++
    jmp copiar

fcopiar:
    # Cierra ambos archivos
    movl $6, %eax               # Servicio #6 (close)
    movl fd, %ebx               # File Descriptor del archivo abierto
    int $0x80                   # Llamada al SO

    movl $6, %eax               # Servicio #6 (close)
    movl fd2, %ebx              # File Descriptor del archivo abierto
    int $0x80                   # Llamada al SO

    # Elimina el archivo original
    movl $10, %eax              # Servicio #10 (unlink)
    movl $file, %ebx            # Ruta del archivo a eliminar
    int $0x80                   # Llamada al SO
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    # Lo sustituye por el archivo temporal
    movl $38, %eax             # Servicio #38 (rename)
    movl $file2, %ebx          # Ruta antigüa
    movl $file, %ecx           # Ruta nueva
    int $0x80                  # Llamada al SO

    jmp fin2                   # Salta al final del programa


    # La posición solicitada no existe
nopos:
    pushl $cad8                # Informa al usuario del problema
    call printf
    addl $4, %esp
    jmp fin                    # Salta al final del programa


    # Ocurrió un error al abrir el archivo
error:
    pushl $cad3                # Informa al usuario del error
    call printf
    addl $4, %esp
    jmp fin2

error2:
    pushl $cad9                # Informa al usuario del error
    call printf
    addl $4, %esp

fin:
    # Cierra el archivo si estaba abierto
    movl $6, %eax              # Servicio #6 (close)
    movl fd, %ebx              # File Descriptor del archivo abierto
    int $0x80                  # Llamada al SO

    # Termina el programa
fin2:
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

12.5.-

```
.section .rodata
cad1:    .asciz "Archivo fuente #1: "
cad2:    .asciz "Archivo fuente #2: "
cad3:    .asciz "Archivo destino: "
cad4:    .asciz "Ocurrió un error al intentar abrir el archivo fuente #1\n"
cad5:    .asciz "Ocurrió un error al intentar abrir el archivo fuente #2\n"
cad6:    .asciz "Ocurrió un error al intentar abrir el archivo destino\n"
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
formato:.asciz "%s"
char:    .asciz "%c"
modor:   .asciz "r"
modow:   .asciz "w"


.section .bss
fuente1:.space 80
fuente2:.space 80
destino:.space 80
fd1: .space 4
fd2: .space 4
fd3: .space 4
buffer:  .space 1


.section .text
.globl _start
_start:

    pushl $cad1              # Solicita la ruta del primer archivo fuente
    call printf
    addl $4, %esp

    pushl $fuente1           # Lee la respuesta del usuario
    pushl $formato
    call scanf
    addl $8, %esp

    pushl $cad2              # Solicita la ruta del segundo archivo fuente
    call printf
    addl $4, %esp

    pushl $fuente2           # Lee la respuesta del usuario
    pushl $formato
    call scanf
    addl $8, %esp

    pushl $cad3              # Solicita la ruta del archivo destino
    call printf
    addl $4, %esp

    pushl $destino           # Lee la respuesta del usuario
    pushl $formato
    call scanf
    addl $8, %esp

    # Abre los dos archivos fuentes para lectura y el destino para escritura

    pushl $modor             # Modo: r
    pushl $fuente1           # Archivo fuente 1
    call fopen               # Llama a fopen
    addl $8, %esp
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    mov %eax, fd1               # Apuntador al archivo abierto

    cmpl $0, fd1                # Si fd==NULL (0) ocurrió un error
    je error1

    pushl $modor                # Modo: r
    pushl $fuente2              # Archivo fuente 2
    call fopen                  # Llama a fopen
    addl $8, %esp
    mov %eax, fd2               # Apuntador al archivo abierto

    cmpl $0, fd2                # Si fd2==NULL (0) ocurrió un error
    je error2

    pushl $modow                # Modo: w
    pushl $destino              # Archivo destino
    call fopen                  # Llama a fopen
    addl $8, %esp
    mov %eax, fd3               # Apuntador al archivo abierto

    cmpl $0, fd3                # Si fd3==NULL (0) ocurrió un error
    je error3

    # Copia el contenido del archivo fuente1 a al archivo destino
rep1:
    # Lee un caracter del primer archivo fuente
    pushl $buffer               # Apuntador al buffer
    pushl $char                 # Cadena de formato
    pushl fd1                   # Apuntador al archivo abierto
    call fscanf                 # Llama a fscanf
    addl $8, %esp

    # Si se alcanzó el final del archivo sale del ciclo
    pushl fd1                   # Apuntador al archivo abierto
    call feof                   # Llama a feof
    addl $4, %esp
    cmpl $0, %eax
    jne rep2

    # Lo guarda en el archivo destino
    pushl buffer                # Contenido del buffer
    pushl $char                 # Cadena de formato
    pushl fd3                   # Apuntador al archivo abierto
    call fprintf                # Llama a fprintf
    addl $8, %esp

    jmp rep1                    # Siguiente línea del archivo

rep2:
    # Lee un caracter del segundo archivo fuente
    pushl $buffer               # Apuntador al buffer
    pushl $char                 # Cadena de formato
    pushl fd2                   # Apuntador al archivo abierto
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    call fscanf                 # Llama a fscanf
    addl $8, %esp

    # Si se alcanzó el final del archivo sale del ciclo
    pushl fd2                   # Apuntador al archivo abierto
    call feof                   # Llama a feof
    addl $4, %esp
    cmpl $0, %eax
    jne fin

    # Lo guarda en el archivo destino
    pushl buffer                # Contenido del buffer
    pushl $char                 # Cadena de formato
    pushl fd3                   # Apuntador al archivo abierto
    call fprintf                # Llama a fprintf
    addl $8, %esp

    jmp rep2                    # Siguiente línea del archivo


error3:  pushl $cad5            # Informa al usuario del error
    call printf
    addl $4, %esp
    jmp fin

error2:  pushl $cad5            # Informa al usuario del error
    call printf
    addl $4, %esp
    jmp fin

error1:  pushl $cad4            # Informa al usuario del error
    call printf
    addl $4, %esp
    jmp fin

fin: # Cierra cualquier archivo abierto previamente

cierra1:cmpl $0, fd1
    je cierra2
    pushl fd1                   # Descriptor del archivo a cerrar
    call fclose                 # Llama a fclose
    addl $4, %esp

cierra2:cmpl $0, fd2
    je cierra3
    pushl fd2                   # Descriptor del archivo a cerrar
    call fclose                 # Llama a fclose
    addl $4, %esp

cierra3:cmpl $0, fd3
    je fin2
    pushl fd3                   # Descriptor del archivo a cerrar
    call fclose                 # Llama a fclose
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    addl $4, %esp

    # Termina el programa
fin2:
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

12.6.-

```
.section .rodata
cad1:    .asciz "Archivo fuente #1: "
cad2:    .asciz "Archivo fuente #2: "
cad3:    .asciz "Archivo destino: "
cad4:    .asciz "Ocurrió un error al intentar abrir el archivo fuente #1\n"
cad5:    .asciz "Ocurrió un error al intentar abrir el archivo fuente #2\n"
cad6:    .asciz "Ocurrió un error al intentar abrir el archivo destino\n"
formato: .asciz "%s"
char:    .asciz "%c"
modor:   .asciz "r"
modow:   .asciz "w"


.section .bss
fuente1: .space 80
fuente2: .space 80
destino: .space 80
fd1:     .space 4
fd2:     .space 4
fd3:     .space 4
buffer:  .space 84*500
contactos:.space 4


.section .text
.globl _start
_start:

    pushl $cad1              # Solicita la ruta del primer archivo fuente
    call printf
    addl $4, %esp

    pushl $fuente1           # Lee la respuesta del usuario
    pushl $formato
    call scanf
    addl $8, %esp

    pushl $cad2              # Solicita la ruta del segundo archivo fuente
    call printf
    addl $4, %esp
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    pushl $fuente2             # Lee la respuesta del usuario
    pushl $formato
    call scanf
    addl $8, %esp

    pushl $cad3                # Solicita la ruta del archivo destino
    call printf
    addl $4, %esp

    pushl $destino             # Lee la respuesta del usuario
    pushl $formato
    call scanf
    addl $8, %esp

    # Abre los dos archivos fuentes para lectura y el destino para escritura

    movl $5, %eax              # Servicio #5 (open)
    movl $fuente1, %ebx        # Ruta del archivo a abrir
    movl $0, %ecx              # ORDONLY : 0
    int $0x80                  # Llamada al SO
    mov %eax, fd1              # Apuntador al archivo abierto
    cmpl $0, fd1               # Si fd<0 ocurrió un error
    jl error1

    movl $5, %eax              # Servicio #5 (open)
    movl $fuente2, %ebx        # Ruta del archivo a abrir
    movl $0, %ecx              # ORDONLY : 0
    int $0x80                  # Llamada al SO
    mov %eax, fd2              # Apuntador al archivo abierto
    cmpl $0, fd2               # Si fd<0 ocurrió un error
    jl error2

    movl $5, %eax              # Servicio #5 (open)
    movl $destino, %ebx        # Ruta del archivo a abrir
    movl $01101, %ecx          # O_CREAT | O_TRUNC | O_WRONLY
    movl $0644, %edx           # Permisos del archivo
    int $0x80                  # Llamada al SO
    mov %eax, fd3              # Apuntador al archivo abierto
    cmpl $0, fd3               # Si fd<0 ocurrió un error
    jl error3

    movl $0, contactos         # contactos = 0
copiar1:
    movl $3, %eax              # Servicio #3, (read)
    movl fd1, %ebx             # File Descriptor del archivo abierto
    leal buffer, %ecx          # Dirección donde se almacenarán los datos
    addl contactos,%ecx
    movl $84, %edx             # Número de bytes a leer
    int $0x80                  # Llamada al SO

    cmpl $0, %eax              # Si se leyeron cero bytes
    je copiar2                 # Terminó con este archivo
    addl $84,contactos
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    jmp copiar1

copiar2:
    movl $3, %eax              # Servicio #3, (read)
    movl fd2, %ebx             # File Descriptor del archivo abierto
    leal buffer, %ecx          # Dirección donde se almacenarán los datos
    addl contactos,%ecx
    movl $84, %edx             # Número de bytes a leer
    int $0x80                  # Llamada al SO

    cmpl $0, %eax              # Si se leyeron cero bytes
    je seguir                  # Terminó con este archivo

    xorl %ebx, %ebx            # i=0
revisar:cmpl contactos,%ebx    # mientras i<contactos
    jge noesta

    leal buffer(%ebx), %esi    # Nombre del contacto en la posición i
    leal buffer, %edi          # Nombre del contacto leído
    addl contactos, %edi
    movl $80, %ecx             # Comparar hasta 80 caracteres
    rep cmpsb                  # Compara el nombre de los contactos
    je copiar2                 # Si el contacto ya existe se ignora
    addl $84, %ebx             # i++
    jmp revisar

noesta:
    addl $84,contactos         # Si el contacto no está en la lista
    jmp copiar2                # Se agrega

seguir:
    xorl %edi, %edi            # i=0
vaciar: cmpl contactos, %edi   # mientras i<contactos
    jge fin
    movl $4, %eax              # Servicio #4, (write)
    movl fd3, %ebx             # File Descriptor del archivo abierto
    leal buffer(%edi), %ecx    # Apuntador a los datos a escribir
    movl $84, %edx             # Número de bytes a escribir
    int $0x80                  # Llamada al SO
    addl $84, %edi
    jmp vaciar

error3:  pushl $cad5           # Informa al usuario del error
    call printf
    addl $4, %esp
    jmp fin

error2:  pushl $cad5           # Informa al usuario del error
    call printf
    addl $4, %esp
    jmp fin

error1:  pushl $cad4           # Informa al usuario del error
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.

```
    call printf
    addl $4, %esp
    jmp fin

fin: # Cierra cualquier archivo abierto previamente

cierra1:cmpl $0, fd1
    jl cierra2
    movl $6, %eax                 # Servicio #5 (close)
    movl fd1, %ebx                # File Descriptor del archivo abierto
    int $0x80                     # Llamada al SO

cierra2:cmpl $0, fd2
    jl cierra3
    movl $6, %eax                 # Servicio #5 (close)
    movl fd2, %ebx                # File Descriptor del archivo abierto
    int $0x80                     # Llamada al SO

cierra3:cmpl $0, fd3
    jl fin2
    movl $6, %eax                 # Servicio #5 (close)
    movl fd3, %ebx                # File Descriptor del archivo abierto
    int $0x80                     # Llamada al SO

    # Termina el programa
fin2:
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

Este documento es propiedad intelectual de los autores, Amelia Ferreira y Vicente Robles.
Su reproducción total o parcial requiere la autorización previa de sus autores. Caracas 2007.