APÉNDICE 2

ENSAMBLADOR EN LÍNEA CON EL *gcc*

1. Herramientas

Los programas se desarrollarán en C y en ensamblador. El ensamblador se utilizará básicamente para realizar ciertas optimizaciones en el código C (dentro del programa en C, *"in line"*). Las prácticas se realizarán con software de dominio público, en concreto con el compilador ***gcc*** bajo el sistema Linux, o bien con el paquete ***djgpp*** que implementa gcc en DOS e incluye además el entorno de desarrollo **rhide** para DOS. Este paquete se puede encontrar en http://www.delorie.com/djgpp.

2. Ensamblador de GNU

El compilador gcc (y el djgpp) utiliza para el ensamblador la sintaxis de AT&T (GAS = Gnu ASsembler). Ésta tiene pequeñas diferencias con respecto a la sintaxis estándar de Intel (usada en TASM, MASM, etc). Las principales diferencias se detallan en la siguiente tabla (ver el apéndice 1 para más detalle):

DIFERENCIAS	AT&T (Motorola)	Intel
1. En AT&T, a los nombres de los registros se les añade el prefijo %.	%eax	eax
2. En AT&T, el destino se coloca a la derecha y el fuente a la izquierda (en Intel es al contrario).	movl %eax, %ebx	mov ebx, eax
3. En AT&T, a los valores inmediatos se les añade el prefijo $.	movl $var, %eax movl $0xf02, %ebx	mov eax, offset var mov ebx, 0f02h

4. En AT&T, el tamaño del resultado se especifica con sufijos (b, w o l) en las instrucciones (en Intel cuando hay ambigüedad se utiliza byte ptr, word ptr o dword ptr).	movb var, %ah
	movw %bx, %ax
	movl %ebx, %eax
		mov ah, byte ptr var
		mov ax, bx
		mov eax, ebx

5. Direccionamiento a memoria.	desplazamiento(base,indice,escala)
	movl array (, %eax, 4), %edx
	movl (%ebx), %eax
	movl 3(%ebx), %eax
		[base+indice*escala+desplazamiento]
		mov edx, array[eax*4]
		mov eax , [ebx]
		mov eax , [ebx+3]

6. Salto lejano.	lcall $sección, $offset ljmp $sección, $offset lret $V	call far sección:offset jmp far sección:offset ret far V

7. Varían los nemotécnicos de algunas instrucciones.	movswl %ax, %ecx
	movzbw %ah, %cx
	cbtw
	cwtl
	cwtd
	cltd
		movsx ecx, ax
		movzx cx, ah
		cbw
		cwde
		cwd
		cdq

En la página web del DJGPP podemos encontrar una guía detallada sobre la sintaxis AT&T, y ejemplos de ensamblador en línea:

http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html

2.1. Ensamblador en línea en *gcc*

A continuación detallaremos la sintaxis utilizada para insertar código ensamblador en un programa en C. Para más información sobre la sintaxis utilizada por GAS se

recomienda estudiar la página de manual (*man as* o bien *info as*) o visitar los enlaces que se proponen al final del guión.

2.1.1. Ensamblador en línea básico

El formato básico para incluir instrucciones ensamblador es simple, y similar al utilizado por Borland:

```
asm ( "sentencias" )
```

Ejemplos:

- ejecutar la instrucción NOP
  ```
  asm ("nop")
  ```
- deshabilitar interrupciones:
  ```
  asm ("cli")
  ```
- y definiendo como una macro:
  ```
  #define disable() asm volatile ("cli")
  ```

Se puede utilizar ***__asm__*** en lugar de ***asm*** si esta palabra reservada entra en conflicto con algún identificador del programa C.

Veamos cómo incluir varias líneas de ensamblador en un programa C:

```
__asm__ ("movl %eax, %ebx \n\t"
      "movl $56, %esi \n\t"
      "movl %ecx, $label(%edx , %ebx, $4) \n\t"
      "movl %ah, (%ebx)";
```

Cada línea va encerrada entre comillas dobles (" "); el retorno de carro y el tabulador (\n\t) al final de cada línea se debe a que gcc pasa esas líneas tal cual a GAS, que espera una instrucción ensamblador por línea, y además identada un tabulador.

2.1.2. Ensamblador en línea extendido

En el último ejemplo hemos hecho uso de los registros sin tener en cuenta si en instrucciones previas gcc tenía almacenado algún valor. Puesto que no queremos interferir con gcc en la asignación de registros a variables del programa C, en el bloque de instrucciones ensamblador deberemos indicar a qué variables queremos acceder y si es para lectura o escritura, en qué registros queremos guardarlas (y que gcc tenga esto en cuenta para preservar valores que pudieran perderse), etc.

De cualquier forma, lo más sencillo y seguro para evitar problemas será dejar que gcc elija los registros en los cuales guardar los valores que vayamos generando.

La sintaxis general del ensamblador en línea extendido será:

```
asm ( "sentencias"
      : salidas
      : entradas
      : preservar_uso )
```

donde:

- sentencias: Secuencia de instrucciones ensamblador separadas por caracteres *"new line"* ('\n'). Para mejorar aspecto en la salida como código ensamblador utilizar mejor "\n\t" como separación entre instrucciones.

- salidas: Secuencia de variables gcc de salida entre paréntesis precedidas por las restricciones (y modificadores de éstas) a las que están sujetas (ver más adelante).

- entradas: Secuencia de variables gcc de entrada entre paréntesis precedidas por las restricciones (y modificadores de éstas) a las que están sujetas.

- preservar_uso: Para un correcto funcionamiento se debería informar al compilador de los recursos que se modifican dentro del código ensamblador *"in line"*, con el fin de que tenga cuidado en preservar su valor si resulta necesario. Los recursos a preservar se colocan separados por comas después de las entradas. Los recursos indicados en la línea de salida no se deben colocar en la lista para preservar, ya que el compilador ya sabe, por estar en la lista de salida, que se va a escribir sobre ellos. Se debería preservar los registros que se modifican dentro del código ensamblador (eax, ebx, ...) que no aparecen en la lista de salida. Si alguna instrucción ensamblador modifica memoria (y no se especifica en la línea de salida), se debe incluir para un correcto funcionamiento "**memory**" en la lista. Con "**cc**" se indica que se preserva el registro de estado. No se permite colocar en la lista de preserva registros asignados a entradas, sin embargo puede haber casos en los que el código ensamblador modifique estos registros (hay que tener cuidado con esto).

Ejemplo:

El siguiente código ensamblador en línea asigna el valor de la variable ***a*** a la variable ***b*** (b = a), y asigna el valor 4 a la variable ***c*** (c = 4).

```
asm   ("movl %2, %%eax  \n\t  \
        movl %%eax, %0  \n\t  \
        movl $4, %%eax  \n\t  \
        movl %%eax, %1  \n\t"
        : "=g"(b), "=g"(c)
        : "m"(a)
        : "%eax" );
```

a) Sentencias

```
movl %2, %%eax    \n\t
```

```
movl %%eax, %0    \n\t
movl $4, %%eax    \n\t
movl %%eax, %1    \n\t
```

Dentro de las sentencias se hace referencia a las entradas y a las salidas con un número precedido de % (%0, %1, ..., %i, ..., %N). Cuando sólo se utilizan entradas (salidas) la asignación de número se debe realizar en orden de aparición de éstas en la lista de entradas (salidas) comenzando desde **0**. Cuando se utilizan entradas y salidas, los números correlativos comenzando desde **0** se asignan primero a las ***salidas***, y los siguientes números se asignan a las ***entradas***, también de forma correlativa por orden de aparición en la lista. Por ejemplo si se dispone de ***N*** variables, de las cuales ***K*** son de salida, la asignación por orden de aparición para las salidas sería: %0, %1, ..., %K-1; y la asignación para las entradas por orden de aparición de éstas sería: %K, %K+1, ..., %N-1.

Dentro del código *"in line"* los registros van precedidos de **dos** caracteres % en lugar de **uno** (para evitar confusión con las entradas y las salidas).

b) Salidas

Se especifica que se utilizan como salida las variables b y c. La restricción g le indica al compilador que puede escoger entre cualquier registro de propósito general o memoria para ubicar el resultado que va a pasar a b tras la ejecución de la secuencia de instrucciones *"in line"*. El modificador = indica que se trata de una salida. En el ejemplo %0 se sustituye por la ubicación de b, y %1 por la ubicación de c.

c) Entradas

El contenido de la variable a es entrada al código ensamblador. La restricción m indica que la entrada al código ensamblador en la posición de %2 es memoria.

d) Preservar uso

En la lista aparece %eax. De esta forma se le indica al compilador que de los registros en los que puede asignar a entradas o a salidas (en este caso se lo puede asignar sólo a las salidas dado la restricción) excluya a eax, y además le indica que preserve su contenido si es necesario (puede que se haya modificado el contenido de este registro por una instrucción anterior al *"in line"* para ser utilizado por una instrucción posterior).

Ejemplo:

Veamos un programa C completo en el que se incluyen instrucciones ensamblador en línea:

```
// COMPILAR:
//            gcc -Wall -g ej.c -o ej
#include <stdio.h>
```

```
#include <stdlib.h>

int main(void) {
  int a=7;
  int b=5;
  int c=0;

  __asm__         ("movl %2,%%eax\n\t"
                   "movl %%eax,%0\n\t"

                   "movl $4,%%eax\n\t"
                   "movl %%eax,%1\n\t"

                   : "=g"(a),"=g"(b)     /*salidas*/
                   : "m"(c)              /*entradas*/
                   : "%eax"              /*preservar valor*/
                   );

  printf("\n\n a=%d \n b=%d \n c=%d \n" , a, b, c );
  return 0;
}
```

en este ejemplo, se asigna el valor de la variable **c** a la variable **a** (a=c) utilizando **eax** como almacen intermedio; así mismo, se asigna el valor **4** a la variable **b** (b=4).

2.1.3. Modificadores para asm

volatile: El compilador no borrará o reordenará significativamente una sentencia ***asm volatile()***. Se debe tener en cuenta que el compilador puede borrar bloques ***asm*** si comprueba que su salida no se utiliza, además puede reordenar el código para mejorar prestaciones en la ejecución (por ejemplo sacar código fuera de ciclos, reordenar el código dentro de éstos).

const: El compilador asume que una sentencia ***asm const()*** produce salidas que dependen únicamente de las entradas, y por tanto puede someterse a la optimización propia de sub-expresiones y se puede sacar fuera de ciclos.

2.1.4. Modificadores de las salidas

= Se utiliza para marcar una variable como de salida.
+ Marca una variable como de entrada y salida.
& Indica que un operando de salida es modificado dentro del código ensamblador antes de que una de las entradas se lea. De esta forma gcc sabe que no debe colocar en un mismo registro una entrada y dicha salida.

2.1.5. Restricciones de las entradas y las salidas

- m Operando en memoria.
- r Registro de propósito general eax, ebx, ecx, edx, esi o edi.
- q Registros eax, ebx, ecx, o edx.

- g Registro de propósito general, memoria o operando entero inmediato.
- f Registro en punto flotante.
- t Registro en punto flotante (primero de la pila).
- u Registro en punto flotante (segundo de la pila).
- A Registros eax y edx (dato de 64 bits).
- a Registro eax.
- b Registro ebx.
- c Registro ecx.
- d Registro edx.
- D Registro edi.
- S Registro esi.
- 0, 1,…, N Para indicar a gcc que asigne la variable al mismo recurso al que ha asignado otra variable.
- i Operando entero inmediato. Incluye constantes simbólicas cuyos valores se conocen en tiempo de ensamblar (0, ..., 0xffffffff).
- n Operando entero inmediato con valor conocido.
- I Constante en el rango 0 a 31 (para desplazamientos de 32 bits).
- J Constante en el rango 0 a 63 (para desplazamientos de 64 bits).
- K ‘0xff’.
- L ‘0xffff’.
- M 0, 1, 2, o 3 (desplazamientos para instrucciones lea).
- N Constante en el rango 0 a 255.
- G Constante punto flotante (estándar 80387).

3. Ejemplos de ejecución de ensamblador en línea

Para ir poniendo en práctica lo explicado hasta ahora, es aconsejable generar los programas ejecutables para los ejemplos que se muestran en este apartado e ir comprobando su funcionamiento. Podemos utilizar el último programa en C como base para ir insertando el código en línea y probarlo.

3.1. Restricciones al compilador sobre cómo manejar las variables que comunican C con el código en ensamblador

3.1.1. Preservar el valor de los registros

Correcto

```
int i=1;

asm volatile("                  \
  movl    $2, %%eax        \n\t \
  addl %%eax,    %0        \n\t"

: "+r"(i)
:
: "%eax"
);

printf("i=%d\n", i);
```

Incorrecto

```
int i=1;

asm volatile("                  \
  movl    $2, %%eax        \n\t \
  addl %%eax,    %0        \n\t"

: "+r"(i)
);

printf("i=%d\n", i);
```

NOTA:
Un registro que se utiliza en asm se debe preservar.

```
                ------------

int i=1;

asm volatile("                  \
  movl    $2, %%eax        \n\t \
  addl %%eax,    %0        \n\t"

: "=r"(i)
: "0"(i)
: "%eax"
);

printf("i=%d\n", i);

                ------------

int i=1;

asm volatile("                  \
  movl    $2, %%eax        \n\t \
  addl %%eax,    %0        \n\t"

: "=m"(i)
:
: "%eax"
);

printf("i=%d\n", i);
```

3.1.2. Asignación de recursos a las variables de entrada y de salida

Correcto

```
int i=1, j=2;

asm ("                          \
  movl    $2, %%eax        \n\t \
  addl %%eax,    %0        \n\t \
  movl    $3, %%ebx        \n\t \
  addl %%ebx,    %1        \n\t"

: "+r"(i), "+r"(j)
:
: "%eax", "%ebx"
);

printf("i=%d, j=%d\n", i, j);

                ------------

int i=1, j=2;

asm ("                          \
  movl    $2, %%eax        \n\t \
  addl %%eax,    %0        \n\t \
  movl    $3, %%ebx        \n\t \
  addl %%ebx,    %1        \n\t"

: "=r"(i), "=r"(j)
: "0"(i), "1"(j)
: "%eax", "%ebx"
);

printf("i=%d, j=%d\n", i, j);
```

NOTA:
La entrada i utiliza el mismo registro que la salida i asignada a %0, y la entrada j el mismo que la salida j asignada a %1.

Incorrecto

```
int i=1, j=2;

asm ("                          \
  movl    $2, %%eax        \n\t \
  addl %%eax,    %0        \n\t \
  movl    $3, %%ebx        \n\t \
  addl %%ebx,    %1        \n\t"

: "=r"(i), "=r"(j)
:
: "%eax", "%ebx"
);

printf("i=%d, j=%d\n", i, j);
```

NOTA:
No se asegura que la entrada y la salida para %0 se asigne al mismo registro. Igual ocurre con %1.

3.1.3. Asignación de variables de salida a recursos ya asignados

Correcto

```
int i=1, j=2, k=3;

asm volatile("                  \
  movl    %2, %%eax        \n\t \
  addl %%eax,    %0        \n\t \
  addl %%eax,    %1        \n\t"

: "=m"(i), "=m"(j)
: "m"(k), "0"(i), "1"(j)
: "%eax"
);

printf("i=%d, j=%d\n", i, j);
```

Incorrecto

```
int i=1, j=2, k=3;

asm volatile("                  \
  movl    %2, %%eax        \n\t \
  addl %%eax,    %0        \n\t \
  addl %%eax,    %1        \n\t"

: "=m"(i), "=m"(j)
: "0"(i), "1"(j), "m"(k)
: "%eax"
);

printf("i=%d, j=%d\n", i, j);
```

NOTA:
Las variables de salida que se asignan a un recurso ya asignado previamente deben ir al final en la lista de variables.

3.2. Introducir direcciones de variables C en el código en ensamblador

Se puede acceder a la dirección de una variable C en ensamblador anteponiendo $, o bien mediante el operador & en la lista de variables de entrada.

3.2.1. Utilizando el operador $

```
/* Mover la dirección de a a la variable b */
static unsigned long int a=0, b=1;

asm volatile("                  \
  movl   $%1, %%eax        \n\t \
  movl %%eax,    %0        \n\t"

: "=g"(b)
: "m"(a)
: "%eax"
);

printf("a=%ld, &a=%lx, b=%lx\n", a, (unsigned long int) &a, b);
```

NOTA:
Funciona si la variable de entrada (en este caso a) es **estática**.

3.2.2. Utilizando el operador &

```
/* Mover la dirección de a a la variable b */
unsigned long int a=0, b=1;

asm volatile("                  \
  movl    %1, %%eax        \n\t \
  movl %%eax,    %0        \n\t"
```

```
: "=m"(b)
: "m"(&a)
: "%eax"
);

printf("a=%ld, &a=%lx, b=%lx\n", a, (unsigned long int) &a, b);
```

NOTA:
Si se compila con opción -O (código optimizado) no funciona a no ser que la variable a se declare como **estática**.

3.2.3. Usando un registro para almacenar la entrada

```
/* Mover la dirección de a a la variable b */
unsigned long int a=0, b=1;

asm volatile("                  \
  movl    %1, %%eax        \n\t \
  movl %%eax,    %0        \n\t"

: "=m"(b)
: "r"(&a)
: "%eax"
);

printf("a=%ld, &a=%lx, b=%lx\n", a, (unsigned long int) &a, b);
```

3.3. Manejo de diferentes tipos de direccionamiento

3.3.1. Usando un registro base y un desplazamiento

Ejemplo de acceso a una componente de un vector especificando un desplazamiento, y utilizando como base la dirección del vector. La dirección del vector para poderla utilizar como base se deberá introducir en un registro.

```
/* Direccionamiento: Acceder a diferentes posiciones de
   un vector: el registro de entrada se utiliza como
   base, 4 es el desplazamiento (inmed32) */

long int vector[8]={1,2,3,4,5,6,7,8}, sal=0;

asm volatile("                  \
  movl 4(%1), %%eax        \n\t \
  movl %%eax,    %0        \n\t"

: "=g"(sal)
: "r"(vector)
: "%eax"
);

printf("sal=%ld\n", sal);
```

NOTA:
Debe especificarse que la entrada ha de colocarse en un registro. En el ejemplo se imprime 2 (notar que los componentes del vector son de 4 bytes). Funciona por la manera en la que se definen los vectores (observe el código ensamblador).

3.3.2. Usando un registro base y otro índice

Ejemplo de acceso a una componente de un vector utilizando un índice con el que poder moverse por los N componentes (con tamaño M bytes) del vector. (En el ejemplo N=5 y M=4)

```
/* Direccionamiento: Acceder a diferentes posiciones de
   un vector: el registro de entrada se utiliza como
   base, esi se utiliza como índice, 4 es el factor de
   escala para el índice, y el desplazamiento es 0 */

long int vector[8]={1,2,3,4,5,6,7,8}, sal=0;

asm volatile("                          \
  movl          $2, %%esi        \n\t \
  movl (%1,%%esi,4), %%eax       \n\t \
  movl        %%eax,    %0       \n\t"

: "=g"(sal)
: "r"(vector)
: "%eax", "%esi"
);

printf("sal=%ld\n", sal);
```

NOTAS:

- Debe especificarse que la entrada ha de colocarse en un registro.
- En el ejemplo se imprime 3, ya que está en la posición 0+(vector+(2*4)). El tamaño de un long int es 4 bytes y el cero es porque no se ha indicado ningún desplazamiento.
- Necesariamente el índice debe ser un registro, en este caso se ha utilizado esi.

3.3.3. Usando la dirección del vector como desplazamiento y un registro índice

Ejemplo de acceso a una componente de un vector utilizando como desplazamiento la dirección del vector. Se utiliza un índice para poder moverse por los N componentes del vector. (En el ejemplo N=5 y M=4).

```
/* Direccionamiento: Acceder a diferentes posiciones de
   un vector: la entrada desde gcc (dirección del
   vector) se utiliza como desplazamiento, esi se
   utiliza como índice, 4 es el factor de escala para el
   indice */

static long int vector[8]={1,2,3,4,5,6,7,8};
long int sal=0;

asm volatile("                          \
  movl          $3, %%esi        \n\t \
  movl %1(,%%esi,4), %%eax       \n\t \
  movl        %%eax,    %0       \n\t"

: "=m"(sal)
: "m"(*vector)
: "%eax", "%esi"
```

```
);

printf("sal=%ld\n", sal);
```

NOTA:
El vector se debe declarar como estático.

3.4. Macros

Puede resultar útil definir macros para código ensamblador. Por ejemplo, para instrucciones o secuencias de instrucciones en ensamblador que el compilador no utilice o lo haga en raras ocasiones. Por supuesto interesará su uso siempre que sean de interés porque supongan mejoras en tiempo (por ejemplo instrucciones de movimiento condicional) o porque ofrezcan prestaciones que de otra forma no obtendríamos (por ejemplo xchg).

3.4.1. Desplazamiento

Desplazamiento derecha del primer operando (a, de 32 bits) el número de posiciones especificadas en el tercer operando (desp de 8 bits). La parte más significativa dejada al descubierto por los desplazamientos se rellena con la parte menos significativa del segundo operando (b, de 32 bits).

```
#define despD32_8(a, b, desp)    \
  asm ("shrdl %2, %1, %0" : "=m"(a) : "r"(b), "ic"(desp), "0"(a))
```

Se utiliza la instrucción ensamblador shrd. El sufijo l, indica que los datos son de 32 bits. Para esta instrucción máquina (ver manual de Intel) el dato de salida puede estar en memoria o en un registro, en la macro se ha restringido a m. El dato de entrada b ha de estar en un registro, de ahí que se haya utilizado r como restricción. El desplazamiento ha de ser un dato de 8 bits, bien inmediato o estar en el registro cl, por esto se han utilizado dos restricciones i y c. El compilador seleccionará cl o dato inmediato dependiendo de como se especifique desp, por ejemplo, en la llamada

```
despD32_8(dato_desp, dato_relleno, 8);
```

el desplazamiento a la instrucción se da como dato inmediato, mientras que en

```
despD32_8(dato_desp, dato_relleno, desp8);
```

el desplazamiento a la instrucción se da introducido en cl.

3.4.2. Intercambio de valores

Una posible macro para el intercambio de dos valores puede ser:

```
#define intercambio2(a, b)    \
  asm ("xchgl %0, %1" : "+m"(a), "+r"(b))
```

3.4.3. Obtención del número de ciclos del reloj desde que se inició el ordenador

```
#define ciclos(cont)    \
  asm volatile("rdtsc" : "=A"(cont))
```

La instrucción en ensamblador rdtsc (sólo a partir del Pentium) devuelve el número de ciclos de reloj contados desde que se inició el computador en los registros edx (parte más significativa) y eax (parte menos significativa). Se pueden declarar variables de 64 bits en gcc mediante *long long int*.

3.4.4. Transferencia de datos entre dos zonas de memoria

Una posible macro para transferir datos entre dos zonas de memoria es:

```
#define transfer(fuente, destino, numbytes)       \
  asm volatile("                                  \
    movl %0, %%esi \n\t                           \
    movl %1, %%edi \n\t                           \
    movl %2, %%ecx \n\t                           \
    cld            \n\t                           \
    rep            \n\t                           \
    movsb          \n\t"                          \
  : : "m"(fuente), "m"(destino), "im"(numbytes)    \
  : "%ecx", "%edi", "%esi", "memory")
```

Otra posibilidad es:

```
#define transfer(fuente, destino, numbytes)     \
  asm volatile("                                \
    cld            \n\t                         \
    rep            \n\t                         \
    movsb          \n\t"                        \
  : : "S"(fuente), "D"(destino), "c"(numbytes) \
  : "memory")
```

Estas macros copian una zona de memoria con dirección fuente a otra con dirección destino, los datos son de 8 bits (de ahí el sufijo b en movs), y el total de datos a transferir es numbytes. Para la copia se aprovecha la instrucción de cadenas movs de los procesadores x86, que mueve el dato con dirección en esi a la dirección que contenga edi. Para repetir esta operación de transferencia un número de veces se utiliza el prefijo rep que repite la operación a la que precede el número de veces especificado en el registro ecx. La instrucción cld pone a 0 el bit DF del registro de flags, si este bit está a cero las instrucciones de cadenas (como movs, cmps, scas, lods, stos) incrementan, cuando se ejecutan, el contenido de los registros esi y edi. Se deben preservar los registros que se modifican durante la ejecución (edi, esi y ecx) ya que gcc no sabe que se modifican, y podría, si necesita el valor almacenado al entrar en uno de estos registros, querer acceder a ese valor leyendo del registro. Además ya que el estamento asm modifica memoria que no se especifica en la lista de salida, ésta también se preserva colocando "memory" en la lista.

Un ejemplo de utilización de esta macro podría ser:

```
char *original="Mi mama me mima.";
char *copia="abcdefghijklmnop";
printf("Antes:   Original:%s \tCopia:%s\n", original, copia);
transfer(original, copia, 16);
printf("Despues: Original:%s \tCopia:%s\n", original, copia);
```

4. *Uso de GCC en los ejercicios*

- Una forma de realizar los programas es utilizar un programa base donde se realizan las entradas y salidas del programa. En este mismo programa se realizan las optimizaciones oportunas utilizando el ensamblador en línea. Veamos un ejemplo:

  se presenta un programa en C que define e inicializa tres variables y a continuación las modifica. En un segundo paso, a partir de ese código C, vamos a realizar dichas operaciones utilizando ensamblador en línea (tratando de reducir el tiempo de ejecución):

```
// COMPILAR:
//            gcc -Wall -g ej_c.c -o ej_c
// EJECUTAR:
/*            time ej_c
real    0m0.025s
user    0m0.002s
sys     0m0.000s          */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int var1=7;
  int var2=5;
  int resultado=0;

  printf("\nINICIO:\n var1 = %d \n var2 = %d \n resultado = %d \n\n", var1, var2,
resultado);

  var2++ ;

  printf("\nINCR. var2 :\n var1 = %d \n var2 = %d \n resultado = %d \n\n", var1,
var2, resultado);

  resultado = var1 + var2 ;

  printf("\nSUMA :\n var1 = %d \n var2 = %d \n resultado = %d \n\n", var1, var2,
resultado);

  var1++ ;
  resultado = var1 ;

  printf("\nINCR. var1 / ASIGNAR A resultado :\n var1 = %d \n var2 = %d \n
resultado = %d \n\n", var1, var2, resultado);

  return 0;
}
```

  En el programa anterior se van a sustituir las instrucciones C resaltadas en negrita por bloques de ensamblador en línea:

```
// COMPILAR:
//            gcc -Wall -g ej_inline.c -o ej_i
// EJECUTAR:
/*            time ej_i
real    0m0.003s
user    0m0.000s
sys     0m0.000s      */
#include <stdio.h>
```

```
#include <stdlib.h>

int main(void) {
  int var1=7;
  int var2=5;
  int resultado=0;

  printf("\nINICIO:\n var1 = %d \n var2 = %d \n resultado = %d \n\n", var1, var2,
resultado);

  __asm__ __volatile__ ("incl %0"
                        : "=r"(var2)       // salida
                        : "r"(var2)        // entrada
                        );

  printf("\nINCR. var2 :\n var1 = %d \n var2 = %d \n resultado = %d \n\n", var1,
var2, resultado);

  // la variable de salida es la %0 (resultado)
  // la variable de entrada %1 es var1
  // la variable de entrada %2 es var2
  __asm__ __volatile__ ("movl %2,%0\n\t"
                        "addl %1,%0"
                        : "=r"(resultado)             // salida
                        : "r"(var1) , "r"(var2)       // entradas
                        );

  printf("\nSUMA :\n var1 = %d \n var2 = %d \n resultado = %d \n\n", var1, var2,
resultado);

  __asm__ __volatile__ ("incl %0\n\t"
                        "movl %0,%1"
                        : "=r"(resultado),"=r"(var1)       // salida
                        : "r"(var1)        // entrada
                        );

  printf("\nINCR. var1 / ASIGNAR A resultado :\n var1 = %d \n var2 = %d \n
resultado = %d \n\n", var1, var2, resultado);

  return 0;
}
```

al ejecutarlos, podemos ver que la versión optimizada utilizando ensamblador en línea tarda un tiempo, en media, ligeramente inferior (algunos milisegundos).

- Una opción para realizar optimizaciones (en casos muy concretos) será desarrollar el programa en lenguaje de alto nivel y llamar al compilador para que genere el código ensamblador, que se examinará para reordenar instrucciones u optimizar. Para ello debemos hacer uso de la opción –S del gcc, que nos genera el código ensamblador:

```
// COMPILAR:
//            gcc -S e.c -o e.s
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  int a=7;
  int b=5;
  int c=0;

  c=b;   // swap
  b=a;
  a=c;

  c=a+b;  //suma

  return 0;
}
```

el código ensamblador generado (sin utilizar optimización alguna) es el siguiente:

```
        .file   "e.c"
        .text
        .align 2
.globl main
        .type   main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        movl    $7, -4(%ebp)
        movl    $5, -8(%ebp)
        movl    $0, -12(%ebp)

        movl    -8(%ebp), %eax
        movl    %eax, -12(%ebp)
        movl    -4(%ebp), %eax
        movl    %eax, -8(%ebp)
        movl    -12(%ebp), %eax
        movl    %eax, -4(%ebp)

        movl    -8(%ebp), %eax
        addl    -4(%ebp), %eax
        movl    %eax, -12(%ebp)

        movl    $0, %eax
        leave
        ret
.Lfe1:
        .size   main,.Lfe1-main
        .ident  "GCC: (GNU) 3.2 20020903 (Red Hat Linux 8.0 3.2-7)"
```

el código resaltado corresponde al “swap” y a la suma. Ese código se podría reordenar y optimizar para realizar esas dos operaciones más rápidamente (las seis instrucciones para hacer el swap se pueden cambiar por un ***xchgl***).

5. Trabajo a desarrollar

5.1. Comprensión de los ejemplos de la Sección 3

Genere programas ejecutables para los ejemplos que se muestran en la Sección 3 y ejecútelos.
Además genere el código en ensamblador y observe las asignaciones de variables a registros que realiza el compilador.
Compruebe si el resultado de la ejecución es correcta o no, en caso negativo anote la causa en función de lo que observe en la ejecución y en el código ensamblador generado. Entregue las anotaciones al profesor.

5.2. Optimización de programas C

- Optimizar el código señalado en el siguiente programa C (en el que se mueven datos entre matrices) para mejorar el tiempo de ejecución:

```
// COMPILACIÓN:          gcc -Wall   movim.c  -o movim
/*
  A,B matrices de TxT (500x500) de valores enteros aleatorios
  (tal y como esté inicializada la memoria al ejecutar el programa)

  objetivo: optimizar el movimiento de datos desde la matriz A a la B
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define  TAMANO  500

int main(void) {
  clock_t start, finish;
  int A[TAMANO][TAMANO];
  int B[TAMANO][TAMANO];
  int i,j,tmp;

  printf("\n\n-----moviendo datos : ");
  start =clock();
  /********************** OPTIMIZAR***********************************/
  for(i=0; i<TAMANO; i++) {
    for(j=0; j<TAMANO; j++) {
      tmp = A[i][j] ;
      B[i][j] = tmp ;
    }
  }
  /*******************************************************************/
  finish =clock ();

  printf("\n\n-----TIEMPO (solo en las operaciones) : ");
  printf("\n\t %f miliseg.\n",((double)(finish - start))*1000 / CLOCKS_PER_SEC);

  return 0;
}
```

- Optimizar el código señalado en el siguiente programa C (bucles, operaciones matemáticas, “swaping” entre variables, etc) para mejorar el tiempo de ejecución:

```
// COMPILACIÓN:          gcc -Wall   mat.c  -o mat
/*
  A,B,E matrices de TxT (500x500) de valores enteros aleatorios entre 2 y 5

  tmp = ( A[i,j] + B[i,j] ) ** E[i,j]
  R[i,j] = constante * tmp
  swap ( A[i,j] , B[i,j] )
  suma_total = sumatoria( R[i,j] )

  objetivo: optimizar el cálculo de la potencia y la sumatoria, y el swap
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define  TAMANO  500

unsigned potencia(unsigned base, unsigned exp) {
  unsigned resultado;

  /********************** OPTIMIZAR***********************************/
  int j,tmp;
  resultado=1;
  j=1;
  while( j<=exp ) {
    tmp = resultado * base;
    j = j + 1;
    resultado = tmp;
  }
  /*******************************************************************/
```

```
  return resultado;
}

int main(void) {
  clock_t start, finish;
  int A[TAMANO][TAMANO];
  int B[TAMANO][TAMANO];
  int E[TAMANO][TAMANO];
  int R[TAMANO][TAMANO];
  int suma, tmp1,tmp2, tmp3, punto;
  unsigned i,j,k;
  srand( time(NULL) );

  printf("\n-----valor escalar : ");
  scanf("%i",&punto);

  printf("\n-----rellenando las matrices... ");
  for(i=0; i<TAMANO; i++) {
    for(j=0; j<TAMANO; j++) {
      A[i][j] = 2+(int) (5.0*rand()/(RAND_MAX+1.0));
      B[i][j] = 2+(int) (5.0*rand()/(RAND_MAX+1.0));
      E[i][j] = 1+(int) (3.0*rand()/(RAND_MAX+1.0));
    }
  }

  printf("\n\n-----operaciones : ");
  start =clock();

  /********************** OPTIMIZAR***********************************/
  for(i=0; i<TAMANO; i++) {
    for(j=0; j<TAMANO; j++) {
      tmp1 = A[i][j];
      tmp2 = B[i][j];
      tmp3 = E[i][j];
      tmp1 = tmp1 + tmp2 ;
      tmp2 = potencia( tmp1 , tmp3 );

      k = suma = 0;
      while( k<tmp2 ) {
       suma = suma + punto;
       k = k + 1;
      }
      R[i][j] = suma;
    }
  }

  for(i=0; i<TAMANO; i++) {
    for(j=0; j<TAMANO; j++) {
      tmp1 = A[i][j];
      tmp2 = B[i][j];
      A[i][j] = tmp2;
      B[i][j] = tmp1;
    }
  }

  suma = 0;
  for(i=0; i<TAMANO; i++) {
    for(j=0; j<TAMANO; j++) {
      tmp1 = suma;
      tmp2 = R[i][j];
      tmp3 = tmp1 + tmp2;
      suma = tmp3;
    }
  }
  /*********************************************************************/

  finish =clock ();
  printf("\n\t suma = %d " , suma);

  printf("\n\n-----TIEMPO (solo en las operaciones) : ");
  printf("\n\t %f miliseg.\n",((double)(finish - start))*1000 / CLOCKS_PER_SEC);

  return 0;
}
```

En ambos casos, entregue sus resultados y conclusiones al profesor. Para comprobar las diferencias de velocidad entre la versión optimizada y sin optimizar, se puede aumentar el tamaño de las matrices.

6. Enlaces interesantes

[1] http://linuxassembly.org
[2] http://linuxassembly.org/articles/linasm.html
[3] http://linuxassembly.org/articles/rmiyagi-inline-asm.txt
[4] http://www-106.ibm.com/developerworks/linux/library/l-ia.html?dwzone=linux
[5] http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html