Tutorial de ensamblador AT&T para procesadores x86

-[0x02]----------
-[Tutorial de ensamblador AT&T para microprocesadores x86]----------
-[by Cemendil]----------SET-29--

Autor : Cemendil <cemendil@hotmail.com>
Fecha : 29/10/2003
Version : 0.91

----------| Contenidos. |----------

Prefacio.

El presente tutorial explica los fundamentos del ensamblador en formato AT&T (el mas empleado en maquinas tipo UNIX), para micros ix86 en modo de 32 bits. Se expondran los fundamentos de este formato (parte 1), como programar subrutinas compatibles con los compiladores de C -- especialmente el gcc -- (parte 2), y como intercalar codigo ensamblador en el interior de un programa en C de manera que gcc sea capaz de compilarlo de manera optima (parte 3).

Esta ultima parte es especialmente complicada, puesto que gcc requiere mucha informacion para poder admitir el ensamblador dentro del codigo en C. Basta pensarlo un momento: si nuestro fragmento de ensamblador modifica el registro %ebx, el compilador debe saberlo. De otro modo podria ocurrir que una variable tipo 'register' almacenada en %ebx por el compilador fuese corrompida al ejecutarse el fragmento de ensamblador, con resultados posiblemente tragicos. Si por el contrario el compilador esta al tanto, el mismo se encargara de hacer el baile de registros necesario para que nada se vaya al garete.

Para entender este tutorial necesitaras ciertos conocimientos previos de ensamblador. No es mi intencion hacer un cursillo de introduccion al ensamblador, si no explicar como sacar partido a la potencia del ensamblado con gcc.

El que el tutorial tenga mas de 100Kb no debe hacerte pensar que el tema es muy largo o complicado; he escrito este documento con abundancia de ejemplos y procurando ser lo mas redundante posible. A menudo un tema se introduce informalmente, luego se repite de manera formal y finalmente se estudia un ejemplo elaborado. Esto hace que el documento sea muy grande, aunque espero que sea tambien mucho mas instructivo. Realmente es necesario que haya mas gente dispuesta a programar en ensamblador de una manera moderna. Los viejos tiempos, en que los programas se hacian 100% en ensamblador, han pasado, pero eso no quiere decir que tengamos que programar en Visual Basic.

Creo que un tutorial como este es necesario, dado que es bastante complicado encontrar documentacion en nuestra lengua sobre ensamblado en UNIX. Por mi experiencia en foros, de vez en cuando aparece alguien totalmente confundido por esos garabatos que vomita UNIX cuando las cosas se tuercen. Bueno, si estas en ese grupo, espero que en este articulo encuentres un alivio a tu confusion. Si tienes experiencia en estos temas, quizas las partes avanzadas del tutorial (sobre todo la Parte 3) te sirvan como minimo de referencia.

Introduccion.

Es muy posible que estes familiarizado con el lenguaje ensamblador para microprocesadores x86, y que hayas escrito algunos programillas usando la notacion Intel, la mas comun para estos menesteres, de la que el siguiente fragmento de codigo es un ejemplo:

```
or [bx], bp
add di, ax
or dh, [bx+si]
add [301ah], di
add [si+0ah], bp
```

```
        xor al, [bx+di]
        int 3
```

Lo que haga este codigo es irrelevante. Lo importante es el formato del codigo: estamos ante ensamblador tipo Intel en 16 bits. Este formato es el que obtendras usando la opcion 'u' del debug de MS-DOS, o cualquier desensamblador generico para Windows. Pero si has estado hozando un poco en UNIX, es posible que te hayas encontrado con desensamblados como el siguiente (usa, por ejemplo, 'gcc -S' con cualquier programa en C, y abre el fichero con extension .s que aparece tras la compilacion):

```
        addl $-8,%esp
        movl -8(%ebp),%eax
        pushl %eax
        movzwl -10(%ebp),%eax
        movl 8(%ebp),%edx
        movsbl (%eax,%edx),%eax
        pushl %eax
        call _putc
        addl $16,%esp
```

Las diferencias respecto al codigo anterior son claras: por un lado, aparecen simbolos curiosos como '%' y '$'. Ademas, instrucciones familiares como 'add' y 'push' adquieren un sufijo, como 'addl' y 'pushl'. Por otro lado, el codigo es de 32 bits, como demuestra el uso de registros extendidos (%eax, %edx, etc.).

El hecho de que el ensamblador sea de 32 bits no es importante. Naturalmente, puedes programar en 32 bits con formato Intel, por ejemplo con nasm. De lo que nos ocuparemos sera de explicar las primeras diferencias, las referentes a la notacion.

Algunos hackers de bastante prestigio detestan el formato AT&T: basta ver la muestra de arriba para darse cuenta de que hay que escribir mucho mas que con formato Intel, con todos esos '%', '$' y sufijos. Sin embargo, la cosa es como el eterno debate entre vi y emacs: puede que el formato AT&T sea poco amigable, pero si te acostumbras acaba siendo casi irreemplazable. Yo emepece programando en formato Intel, pero una vez que me hice a programar a la AT&T, ya casi no puedo ver el ensamblado en modo Intel. Como no es nuestra tarea iniciar 'holy wars' por motivos tan irrelevantes (aunque nos encante ;) sera mejor que juzgues por ti mismo. En ultima instancia, el formato AT&T es inevitable en ciertas ocasiones, por ejemplo al usar gdb.

Si no tienes experiencia en el tema tratado por este tutorial, mi consejo es que lo leas con una consola abierta, dispuesto a compilar y experimentar con cada ejemplo que se propone. Es dificil captar muchos aspectos de estas cuestiones sin dedicar un tiempo a la practica.

Un documento tan largo, y que pretende cubrir un area tan extensa, necesariamente tiene deficiencias. Si observas alguna, por favor enviame un mail. Tambien puedes editar el documento libremente (consulta la licencia).

----------| Licencia. |----------

a mi (Cemendil) del documento, o bien indicaras en lugar bien visible quien eres y que cambios has introducido.

ii) El autor no se hace responsable de cualquier contingencia derivada del uso, correcto o incorrecto, de este documento.

----------| Herramientas. |----------

Lo unico necesario para ensamblar a gusto en el formato AT&T es el ensmablador GNU, el gas. Este viene con todas las distribuciones de UNIX libres, de modo que no tendras problemas en este sentido. Para moverte en las partes 2 y 3 necesitaras tambien el compilador gcc; de nuevo, en UNIX no hay problema.

Para conseguirte las ultimas versiones de gas o gcc, visita www.gnu.org y dirigete a un mirror ftp que contenga las distribuciones en codigo fuente (ftp.aeneas.mit.edu, en /pub/gnu/, por ejemplo). Para conseguir gas, descargate las 'binutils'. La ultima version en fecha de escribir esto es la 2.6. Para conseguir 'gcc', vete al directorio de ese nombre. La ultima version disponible es la 3.3. Ten en cuenta que si quieres montarte toda la suite de compilacion tambien necesitaras el linkador 'ld'.

Ahora, si tienes Windows tambien puedes disfrutar del ensamblado en este formato. Para ello, consiguete el gcc para Windows. Yo conozco dos distribuciones, el djgpp (que mas bien es para MS-DOS, al menos la ultima vez que lo use), y el Bloodshed Dev-C++, que tiene una GUI incorporada y es bastante agradable de usar. Cualquiera de ellos te lo puedes conseguir navegando un poco por la red. Ten en cuenta que el Visual C de Microsoft trabaja solo en formato Intel. De todos modos, el Visual C cuesta un ojo de la cara, asi que si no quieres caer en los oscuros abismos de la pirateria, es bueno saber que tienes esas alternativas que, si no son tan versatiles como el VC, al menos te permiten salvar la candidez de tu alma :).

Hace bastante tiempo vi por la red un programa llamado 'gas2masm'; el programa traducia automaticamente entre ensamblador Intel y AT&T. Si no recuerdo mal, venia una copia con el codigo fuente del Quake. Una herramienta semejante, que debe andar por algun lado en la red, puede resultarte muy util para hacer comparaciones. El programa es sencillo, de modo que puedes imaginarte que las diferencias entre los dos formatos son bastante faciles de aprender.

----------| PARTE 1 : Ensamblador en formato AT&T |----------

En esta parte nos ocuparemos de la estructura del ensamblador en formato AT&T, y de como programar con gas. Ten en cuenta que esto es un tutorial, no un 'white paper'. Los conceptos del ensamblador estaran mezclados con el uso de gas: por ejemplo, '.p2align' es una directiva de gas que no tiene por que ser reconocida por cualquier otro ensamblador en formato AT&T. En fin, dominando gas y usando un poco el sentido comun es posible adaptarse a cualquier otro ensamblador de ese mismo genero (en el dudoso caso de que te encuentres con uno que no sea gas).

---| 1.1 Un codigo de ejemplo.

Lo mejor para familiarizarse con la situacion es trabajar sobre un ejemplo que podamos comprender. Nuestro ejemplo sera la siguiente subrutina (no empleamos un programa principal para ahorrarnos, por ahora, problemas con los valores pasados y devueltos):

```
/* Subrutina hola mundo. */

#include <stdio.h>

static char msg[] = "Hola, mundo!";

void saluda(void)
{
     register short i = 0;

     while (++i < 10)
         if (i == 5) puts(msg);
}
```

Supongamos que tenemos este codigo fuente en el fichero 'hola.c'. En estas condiciones, solo tenemos que hacer desde una shell:

```
demeter# gcc -S hola.c

demeter# ls
hola.c  hola.s
```

Si ahora editamos hola.s, obtenemos:

```
    .file     "test.c"
gcc2_compiled.:
___gnu_compiled_c:
.data
_msg:
      .ascii "Hola, mundo!\0"
.text
      .align 4
.globl _saluda
      .def  _saluda;  .scl 2;  .type 32;  .endef
_saluda:
      pushl %ebp
      movl %esp,%ebp
      subl $20,%esp
      pushl %ebx
      xorl %ebx,%ebx
     .p2align 4,,7
L3:
      incw %bx
      cmpw $9,%bx
      jle L5
      jmp L4
      .p2align 4,,7
L5:
      cmpw $5,%bx
      jne L6
      addl $-12,%esp
```

```
          pushl $_msg
        call _puts
        addl $16,%esp
L6:
        jmp L3
        .p2align 4,,7
L4:
L2:
        movl -24(%ebp),%ebx
        leave
        ret
       .def  _puts;  .scl 2;  .type 32;  .endef
```

Bien, aunque esto parece muy complicado, en realidad tenemos tan solo cuatro tipos de comandos genericos:

1) Directivas como .p2align , .align , .file , .text, etc.
2) Etiquetas como L2: , _msg: , _saluda: , etc.
3) Instrucciones en ensamblador (el resto).
4) Comentarios, que aqui no aparecen. Los comentarios son identicos a los de C, es decir, van entre '/*' y '*/'.

Podemos ver que todos los comandos estan bien delimitados. Las directivas empiezan con un punto, las declaraciones de etiquetas acaban con dos puntos, los comentarios van como en C, y el resto son instrucciones de ensamblador. (Aveces, algunos compiladores ponen un punto antes de las etiquetas, p. ej. '.L0:'. No confundas esto con una directiva).

Por directivas entendemos instrucciones especiales que indican al ensamblador que hacer en ciertas situaciones. Por ejemplo, las directivas .data y .text le indican al ensamblador donde estan los datos y donde esta el codigo. Las directivas tipo .align le dicen al ensamblador que alinee el codigo o datos respecto a una cierta cantidad de bytes (lo tipico es 4, una doble palabra, pero para ciertos micros el optimo puede ser hasta 16, dependiendo de la situacion). La directiva .globl indica al compilador que 'saluda' es el nombre de la subrutina a efectos de linkado. La directiva .ascii marca el inicio de una cadena de caracteres. De la misma manera, una directiva .byte indicaria una cadena de bytes. Existen otras directivas de las que nos ocuparemos en la seccion 1.3.

Nota: ten en cuenta que ciertas directivas solo las emplea en la practica el compilador. No es necesario aprender todas las directivas, si no las mas relevantes. Un buen ejemplo es la directiva .def, que introduce informacion para el debugger.

Entender las etiquetas es mas facil. Podemos ver como los saltos (jmp, jle) se refieren claramente a las etiquetas. Por otro lado, tenemos una instruccion mas que se ocupa de etiquetas. Es la 'pushl $_msg'. Como es facil suponer, lo que hace esta instruccion es meter en la pila la direccion de la cadena asociada a la etiqueta '_msg:', es decir, un puntero a "hola, mundo!\0". Analicemos un poco la instruccion, dado que es un buen ejemplo del formato. Tenemos:

```
pushl $_msg
```

Los elementos raros aqui son el sufijo 'l' a push y el '$' antes de la etiqueta. Es importante que introduzcamos ahora dos nociones fundamentales:

A) En el formato AT&T, toda instruccion que realiza una modificacion de datos _debe_ indicar la magnitud de los datos modificados. Cada magnitud se indica con un sufijo a la instruccion, y hay en total tres magnitudes: byte (sufijo 'b'), palabra (sufijo 'w') y doble palabra (sufijo 'l'). El byte son 8 bits, la palabra 16, y la doble palabra 32. En general, se corresponden con los tipos enteros en C: char, short, long.

Ejemplos (no todos los ejemplos estan en el codigo de arriba):

1) pushl $_msg : Dado que un puntero es de 32 bits, 'push' lleva el sufijo 'l'.

2) cmpw $5, %bx : Dado que %bx es un registro de 16 bits, 'cmp' lleva el sufijo 'w'.

3) inb $20, %al : Para el registro de 8 bits %al, la instruccion 'in' lleva el sufijo 'b'.

4) ret : Esta instruccion no modifica datos, asi que no lleva sufijo. (Ya se, en rigor modifica %eip, pero es que _todas_ las instrucciones modifican %eip, no?)

NOTA: Para la programacion de instrucciones en coma flotante y MMX aparecen otros sufijos, o los mismos con diferente significado. Si vas a programar codigo con la FPU o MMX, deberias leerte la documentacion de gas, donde se detallan todos los sufijos. Algunos ejemplos de la parte 3 de este tutorial emplean la FPU; puedes consultarlos para orientarte.

B) Cuando se quiere indicar una cantidad 'inmediata' en un movimiento de datos, siempre se indica el prefijo '$' antes de esa cantidad. Igualmente, cuando un '$' precede a una etiqueta, se hace referencia a la direccion de memoria a la que apunta la etiqueta. Si no le precede un '$', se entiende que se hace referencia al contenido de la posicion apuntada por la etiqueta. (Aqui el '$' juega un papel analogo al del '&' en C: sin '&' se da el contenido, con '&' se da la direccion).

Ejemplos:

1) pushl $_msg : Mete el puntero de 32 bits que apunta a "Hola, mundo!\0" en la pila.

2) pushb $0x5a : Mete en la pila el byte 0x5a.

3) pushw $0xbb5a : Mete en la pila la palabra 0xbb5a.

4) pushl _msg : Mete en la pila la dpalabra formada por los caracteres 'H', 'o', 'l' y 'a'.

Observa como los numeros hexadecimales se indican exactamente igual que en C. Por defecto, si un numero no lleva el prefijo '0x', se asume que es decimal. Exactamente igual que en C.

ADVERTENCIA: Si una cantidad inmediata no va precedida por un
========== '$', el ensamblador entiende que se trata de una referencia a la memoria.

Ejemplo:

```
pushl 0x5a  : Mete en la pila la doble palabra que se
              encuentra en la direccion de memoria 0x5a
              en el segmento de datos.
```

Ten muy en cuenta esta advertencia, dado que el olvidarse de un '$' es un motivo tipico para obtener un desesperante 'segmentation fault - core dumped' tras otro, al acceder el programa a la memoria arbitrariamente en vez de almacenar datos inmediatos.

Salvo que estes programando un sistema operativo, un sector de arranque o algo asi, nunca querras direccionar memoria directamente, de manera que sospecha de toda cantidad inmediata que vaya sin su correspondiente '$'.

Bueno, una vez que tenemos cierta idea de las directivas, etiquetas y que quieren decir los '$', vamos a ver que quieren decir los '%'. Un simple vistazo nos muestra que siempre que aparece un '%' es como prefijo a un nombre de registro: %ebp, %ebx, %bx, etc. Precisamente la funcion del signo '%' es la de hacer que los nombres de los registros no se puedan confundir con el de alguna variable o subrutina escrita en C. Ten en cuenta que gas es el ensamblador que usa gcc, asi que si en un programa en C usas una variable a la que llamas 'eax', o una subrutina 'esp' (por 'espera', por ejemplo) el ensamblador debe saber que no te estas refiriendo a un registro. Obviamente, no es tarea del gcc anunciar al programador que esta usando nombres de registros: se supone que C es un lenguaje de alto nivel. Asi pues, al ensamblar, antes de un nombre de registro hay que escribir siempre el signo '%'. Una vez que te acostumbras a ello, es una ayuda el poder diferenciar de un vistazo donde se usan variables y donde registros.

NOTA: De todas maneras, algunas versiones de 'gcc' tienen otro mecanismo de seguridad para no confundir simbolos del ensamblador con simbolos del codigo en C. El mecanismo consiste en prefijar todo nombre declarado en C con un subrayado '_'. Si observas el codigo mas arriba, podras ver como la variable 'msg' es ensamblada como '_msg', y la subrutina 'saluda' es asociada a la etiqueta '_saluda'. Este mecanismo de seguridad se explicara con mas detalle en la seccion 2.1.2. Ten en cuenta que no todas las versiones de gcc implementan este mecanismo.

Con esto tenemos casi todo lo relativo al formato AT&T. Bueno, falta un par de cosillas... que suelen disuadir a mucha gente de usar este formato. Observa esta linea, al principio del codigo:

```
subl $20, %esp
```

Bien, ahora podemos entender la cosa facilmente. Es una instruccion 'sub' (restar), que actua sobre magnitudes de 32 bits (sufijo 'l'), y se refiere a una cantidad inmediata de valor 20 decimal ($20), y al registro esp (%esp). Pero un momento ... segun el convenio Intel, lo que haria esta instruccion seria restar a una cantidad inmediata el contenido de un registro! En otras palabras, equivaldria a 20 = 20 - %esp en pseudocodigo, lo que no tiene sentido.

Lo que sucede, naturalmente, es que en formato AT&T el orden de los operadores va _al_reves_ que en el formato Intel. Si en el formato Intel tenemos la forma general:

```
     instruccion  destino, fuente   ( p. ej.  sub esp, 20 )
```

en el formato AT&T tenemos la forma general:

```
     instruccion  fuente, destino   ( p. ej. subl $20, %esp )
```

Esto echa para atras a montones de gente, que de pronto ven que tienen que 'volverse zurdos' (o diestros ;) para manejar este nuevo formato. Mi consejo para cambiar rapido el chip es el siguiente: IMAGINATE QUE ESTAS LEYENDO EL CODIGO. Si, como si estuviese escrito en espaÃ±ol o ingles. Veamos: como escribirias en una frase lo que hace la instruccion 'subl $20, %esp'. Pues mas o menos: "resta 20 a esp". Si te fijas, el orden de los operadores va exactamente igual que en el formato AT&T.

Asi, las lineas anteriores a la que hemos visto son:

```
   pushl %ebp              " %ebp a la pila    "
   movl %esp,%ebp          " mete %esp en %ebp "
   subl $20,%esp           " resta $20 a %esp  "
         ...                     ...
```

Vaya, no se lo que pensaras tu, pero una vez que te acostumbras la cosa es bastante natural. Lo unico que tienes que hacer es leer de izquierda a derecha.

<holy_wars>
Compara eso con 'sub esp, 20'. Para entender esta instruccion debes leerte el mnemonico 'sub' en la izquierda, luego irte al extremo derecho para ver la fuente y entonces al centro para ver el destino. En esta instruccion quizas no sea muy molesto, pero hay codigos con instrucciones mas cargadas que una pizza especial. Ademas, leer estas instrucciones es como leer "restale a esp 20", lo cual es sintacticamente correcto, pero a mi me parece definitivamente incomodo. Para que la notacion Intel tuviera sentido deberia ser, creo yo, 'esp sub 20', que se podria traducir por "esp menos 20", lo cual tiene mas sentido.
</holy_wars>

Con tener en cuenta esto, ya puedes leer practicamente todo el listado en ensamblador del ejemplo, con una excepcion. La linea

```
    movl -24(%ebp), %ebx
```

Esta es la ultima peculiaridad del formato AT&T. La expresion (%ebp) indica al ensamblador que se emplee la direccion de memoria a la que apunta %ebp; en esencia, indica que %ebp actua como puntero. El -24 es un desplazamiento respecto a ese puntero. La expresion -24(%ebp) indica que la fuente del 'movl' es la doble palabra almacenada en la direccion 24 bytes por debajo de la apuntada por %ebp.

En su forma mas general, el direccionamiento en un i386 tiene un puntero base, un desplazamiento inmediato, un indice y un factor de desplazamiento del indice. Por ejemplo,

```
    movl -120(%eax, %ebx, 4), %ecx
```

almacena en %ecx el contenido de la direccion de memoria indicado

por 120 + %eax + 4*%ebx. La forma general del direccionamiento es por tanto: desp(base, ind , fact) .

Ejemplos:

AT&T	Intel
(%eax)	[eax]
-5(%eax)	[eax-5]
(%eax,%ebx)	[eax + ebx]
0x543(%ebp,%eax,2)	[ebp + eax*2 + 543h]
13(,%eax,2)	[eax*2 + 13]

Lo interesante de esta notacion de direccionamiento es que es muy homogenea. Cada cosa esta en su lugar, y es facil de localizar. Hacer un direccionamiento es como llenar una ficha.

NOTA: Adicionalmente, puedes indicar un registro de segmento respecto al cual direccionar la memoria, por ejemplo, %ds:20(%esi, %ebx, 2). Sin embargo, en la practica es muy poco comun usar esta opcion.

Ahora si que tenemos todo lo necesario para entender el codigo en ensamblador que proponiamos al principio. Vamos a ver algunas partes concretas.

Salvando las directivas iniciales, las primeras lineas son:

```
pushl %ebp
movl %esp,%ebp
subl $20,%esp
pushl %ebx
```

estas lineas son muy comunes en subrutinas. Lo que se esta haciendo es conservar la referencia de pila (%ebp) de la subrutina que nos llama (pushl %ebp), definir una nueva referencia (movl %esp,%ebp), definir una total de 20 bytes para almacenamiento de variables locales (subl $20, %esp) y finalmente se conserva el valor de %ebx en la pila (pushl %ebx), dado que se va a usar como variable registro. Veremos mas de esto en la Parte 2.

NOTA: El compilador gcc intenta por todos los medios mantener la pila alineada en 16 bytes al comienzo de las variables locales de cada subrutina, por motivos de eficiencia. Por eso se reservan en la pila 20 bytes que no se van a emplear para nada. Observa tambien los ajustes en la pila antes y despues de la llamada a puts.

Ahora vamos con el cuerpo del programa:

```
L3:
      incw %bx
      cmpw $9,%bx
      jle L5
      jmp L4
     .p2align 4,,7
L5:
      cmpw $5,%bx
      jne L6
      addl $-12,%esp
      pushl $_msg
      call _puts
```

```
              addl $16,%esp
        L6:
                jmp L3
                .p2align 4,,7
        L4:
        L2:
```

Una primera observacion es que gcc alinea siempre los saltos (directiva '.p2align', que comentaremos en la seccion 1.3.2), lo cual es una politica muy sensata. Los saltos mal alineados son tremendamente ineficientes.

Este codigo tiene un fallo: hay muchos mas saltos de los necesarios, pero compilando con -O la cosa cambia mucho (haz la prueba). Por lo demas, es sencillo reconocer lo que pasa: desde L3 a L5 tienes la condicion del bucle, observa que %bx se corresponde con el entero corto 'i' del programa en C. Naturalmente, es tratado con el sufijo 'w'. Desde L5 a L6 tenemos el if, incluyendo la llamada a puts. De L6 a L4 se cierra el bucle.

Finalmente queda por ver la salida de la subrutina:

```
              movl -24(%ebp),%ebx
              leave
              ret
```

el par leave, ret es el usual en estos casos. El movl recupera el valor de %ebx que se metio en la pila al comienzo de la subrutina. Lo mismo habria dado hacer popl %ebx.

---| 1.2 Reglas generales.

Pasemos a resumir todas las reglas que hemos observado, para tener una referencia rapida.

REGLAS FUNDAMENTALES:

1) El orden de los operadores es:

   OPERACION FUENTE, DESTINO   (ej. adcl $26140, %eax)

2) La magnitud de los operadores esta determinada por el sufijo del 'opcode'. Los sufijos de la ALU son 'b', 'w', 'l', que corresponden respectivamente a 8, 16 y 32 bits.

   (ej. incb %al ; pushw $0x5a ; xorl %eax, %eax)

   NOTA: gas es capaz de interpretar correctamente las instrucciones que no lleven sufijo pero cuya magnitud sea obvia, como 'inc %al', pero dara error cuando no sea asi, como en 'push $0x5a'.

3) Cantidades inmediatas y referencias.

   3.a) Las cantidades inmediatas deben ir precedidas de un '$', o se consideraran referencias absolutas de memoria.

   (movl $0x5abb, %eax != movl 0x5abb, %eax)

3.b) Si una etiqueta _no_ va precedida de un '$' se esta haciendo referencia al contenido de la zona de memoria a la que apunta la etiqueta.

```
(ej. movl variable, %eax   /* Almacena contenido de */ )
(                          /* 'variable' en %eax    */ )
```

Si escribes un '$' antes de la etiqueta, se hace referencia a la etiqueta como puntero.

```
(ej. movl $variable, %eax  /* Estas instrucciones   */ )
(    movl %eax,  ptr       /* equivalen en C a:     */ )
(                          /* ptr = &variable;      */ )
```

4) La forma general de direccionamiento es:

```
SEG:DESP(BASE,IND,FACT)
```

donde:
- SEG == registro de segmento a emplear.
- DESP == desplazamiento inmediato a usar.
- BASE == registro base.
- IND == registro indice.
- FACT == factor de multiplicacion del indice.

por ejemplo, leal 120(%eax, %eax, 2), %eax equivale a %eax = 120 + %eax + 2*%eax en pseudocodigo.

5) Los nombres de registros van siempre precedidos de '%'.

(ej. %aex, %ebx, %ecx, %edx, %esi, %edi, %esp, %ebp)

UNA CONVENCION IMPORTANTE:

En el codigo de la seccion 1.1 habras observado que los nombres de las etiquetas de los bucles son L2, L3, L4, etc. En cualquier programa en ensamblador existen etiquetas que no quieres que sean visibles para el linkador; un caso especialmente importante es el de las etiquetas contenidas en las macros.

Cualquier etiqueta que comience por 'L' (ele mayuscula), es considerada por gas como una _etiqueta_local_ y por tanto es traducida directamente a una direccion en memoria sin dejar ninguna referencia en el fichero objeto generado.

Ejemplos de etiquetas locales:

L0, L1, ... , L9 (estas son especiales, ver mas abajo)

Lsalto, Lfalsa, Lloop7, ...

Las etiquetas L0, ... , L9 son especiales porque puedes redefinirlas tantas veces como quieras (gas se encarga de interpretar a cual te refieres en cada momento). Esto las hace especialmente bien dotadas para las macros (tienes un ejemplo de este uso en la seccion 1.4, fichero 'arch.h'). Por ejemplo, puedes definir L0 noventa veces en el mismo programa. En esos casos, si quieres referirte a la etiqueta L0 inmediatamente anterior usa 'L0b' (L0 'backwards'), y si quieres referirte a la inmediatamente posterior usa 'L0f' (L0 'forward').

Por lo tanto, puedes considerar a las etiquetas locales como 'etiquetas basura', que usas en una parte del programa y a continuacion olvidas.

Para mas datos sobre el uso de las L0...L9 consulta la documentacion de gas, donde se precisa como se almacenan internamente.

Una detalle que merece explicacion: en la siguiente seccion veremos que para exportar al linkador un simbolo es necesario usar la directiva .globl (o .global). Entonces, dado que cualquier etiqueta no deberia ser exportada a menos que se use .global, ?por que molestarse en usar etiquetas locales? Bueno, la realidad es que aunque no exportes una etiqueta, cierta informacion de la misma si se exporta, y eso causa que 'gdb' se haga un lio a la hora de desensamblar las subrutinas que has escrito. Una etiqueta no local pero no declarada .global no es lo bastante global como para llamarla desde el linkador, pero es lo suficientemente no local como para que el debugger la trate como una subrutina aparte. Mi consejo es que emplees etiquetas locales dentro de lo posible, y que recurras a las no locales solo en situaciones importantes. Si esto te parece un lio, ten en cuenta que el uso de etiquetas no locales no afecta para nada a la eficiencia del codigo; puedes pasar completamente de las etiquetas locales.

En la seccion 1.4 tienes un ejemplo en el que 'gdb' es capaz de desensamblar el codigo asociado a una etiqueta no local que sin embargo no ha sido declarada .global. Aunque te parezca una ventaja poder desensamblar el codigo en cada etiqueta que usas, esto tiene un inconveniente: 'gdb' solo desensamblara de una vez el codigo que haya entre dos etiquetas no locales dadas. Esto puede resultar incomodo si abusas de las etiquetas no locales.

---| 1.3 Algunas directivas interesantes.

Por lo general, un programa en ensamblador sin directivas es completamente inutil. Veamos algunas directivas interesantes; la referencia completa la puedes encontrar en la documentacion de gas.

1.3.1. Directivas de almacenamiento de datos.

.ascii : almacena en memoria una o varias cadenas de caracteres. Ten en cuenta que esta cadena no es null-terminada (cf. la directiva .asciz)

(ej: .ascii "Hola, mundo!", "\n\0")

.asciz : identico a .ascii, pero se incluye '\0' al final de cada cadena.

.byte : almacena en esa posicion de memoria uno o mas bytes. Esta instruccion es interesante para definir instrucciones no soportadas por el ensamblador.

```
(ej: .byte 0x0f, 0x31        /* Incluir estos datos en */ )
(                            /* el codigo equivale a   */ )
(                            /* la instruccion  rdtsc  */ )
```

.double : almacena una o varias cantidades en coma flotante,
de doble precision (64 bits).

(ej: .double 0.31415926535897932e+1)

.float : almacena una o varias cantidades en coma flotante,
de precision simple (32 bits). (Sinonimo de .single)

(ej: .float 2.718281, 0.69314718055)

.hword : almacena una o varias palabras de 16 bits. Es
sinonimo de .short.

.int : almacena una o varias dpalabras (32 bits). Es
sinonimos de .long.

.octa : almacena uno o varios numeros de 16 bytes (128 bits)
de longitud.

.quad : almacena una o varias cuadruples palabras (64 bits
de longitud). Puede ser muy util.

(ej: .quad 0x0000000003040ffff)

1.3.2. Directivas de alineamiento.

.align : alinea la siguiente etiqueta/instruccion en la
cantidad de bytes indicada. Esta directiva puede
tomar hasta tres parametros:

.align AL, PAD, SKIP

AL : bytes respecto a los que alinear.
PAD : byte con el que rellenar.
SKIP : la directiva sera ignorada si hay que
aÃ±adir mas de SKIP bytes para alinear.

```
(ej:  .align 16,0,7  /* Alinea por 16 bytes a cero, pero */ )
(                    /* que eso no suponga rellenar mas  */ )
(                    /* de 7 posiciones.                 */ )
```

Si en esta directiva no indicas el campo PAD,
el ensamblador rellena con instrucciones tipo
NOP, lo cual es conveniente en el codigo, claro.

.p2align : identico a .align, pero el alineamiento se hace
segun el numero de bits indicado por el primer
parametro de la directiva. Asi, .p2align 2
equivale a .align 4 y .p2align 4 equivale a
.align 16. Para ejemplos del uso de .p2align,
observa el codigo de la seccion 1.1.

Existen otras directivas de alineamiento; si te interesa emplearlas, consulta la documentacion de gas.

1.3.3. Otras directivas importantes.

.data : indica a gas que los siguientes comandos deben
ensamblarse en la seccion de datos. Esta directiva
admite un comando, el numero de seccion, pero es

dudoso que lo encuentres util. Consulta la documentacion, si crees que te puede servir de algo.

.equ : un clasico de los ensambladores. Iguala un simbolo a una expresion, como '.equ STDIN 0'.

.equiv : identico a .equ, pero si el simbolo ya ha sido definido, gas aborta con un error. Puede resultar util.

.file : indica a gas que el codigo que sigue corresponde a cierto fichero. Puedes ver un ejemplo en el codigo de la seccion 1.1.

.global : declara cierto simbolo como global, de manera que 'ld' y 'gdb' pueden reconocerlo. Esto es fundamental a la hora de programar subrutinas, y puede ser muy util para definir breakpoints particulares para 'gdb'.

```
(ej:  .global subrut            /* Hace visible para */ )
(     subrut:                   /* 'ld' el nombre    */ )
(     ...                       /* 'subrut'.         */ )
```

.globl : sinonimo de .global.

.include : permite cargar un fichero dentro de otro. Esto puede emplearse, por ejemplo, para cargar cabeceras o subrutinas interesantes dentro de un programa.

```
(ej:  .include "comun.s" )
```

.rept : repite una cierta seccion de codigo tantas veces como las especificadas en el parametro. El final de la zona a repetir se indica con '.endr'.

```
(ej:  .rept 10       )
(         movsd      )
(     .endr          )
```

.text : indica a gas que debe ensamblar los siguientes comandos en la seccion de codigo. En realidad, .data y .text significan algo mas generico que 'datos' y 'codigo', pero esta distincion funciona perfectamente en todos los casos.

Existen muchas otras directivas, algunas de las cuales pueden resultarte de utilidad en ciertos casos. Recuerda que una de las mayores ventajas de gas es que si necesitas un ejemplo de como codificar algo, siempre puedes recurrir a un programa en C que haga algo parecido, y compilar con -S. Esta tecnica te proveera de infinidad de ejemplos interesantes que funcionan de verdad.

Unas directivas interesantes son las dedicadas a definir macros. Las he omitido del documento por dos motivos: en primer lugar, es algo pesado explicarlas (consulta la documentacion si realmente estas interesado; no es complicado). En segundo lugar, el ensamblador ya es bastante ilegible sin necesidad de macros. Francamente, si la cosa esta tan mal, podrias considerar el uso de un lenguaje de alto nivel, y afinar las zonas criticas en vez de escribir un tochoprograma en ensamblador. De todos modos, en la siguiente seccion explicaremos como usar el preprocesador de gcc en el codigo ensamblado, lo que permite usar un monton de

facilidades de gcc (#define, #ifdef, etc.) dentro de tu programa ensamblado, lo que hace la vida mas facil y ahorra la necesidad de macros en bastantes casos. La idea es usar la mayor cantidad de mecanismos conocidos antes que aprender otros nuevos. ?Pereza mental? Yo prefiero llamarlo economia.

---| 1.4 Usando el preprocesador.

Una de las mejores caracteristicas de gas es su compenetracion con gcc. El primer ejemplo que veremos, antes del plato fuerte de la parte tercera, sera el uso del preprocesador.

En primer lugar, recuerda que los ficheros que continen codigo ensamblador AT&T siempre tienen la extension '.s'. A los ficheros que contengan ensamblador AT&T que requieren preprocesamiento se les pone siempre la extension '.S'.

Ejemplo: hola.s <-- Fichero con codigo ensamblador normal.
macro.S <-- Fichero ensamblador con preprocesamiento.

Si quieres algun ejemplo de fichero .S, echale un vistazo al codigo fuente del nucleo de cualquier UNIX. Podras ver una alegre mezcla de ensamblador y directivas del 'cpp'.

Por ejemplo, al observar el codigo fuente de mi FreeBSD, localizo rapidamente varios ficheros .S en /sys/i386/boot/biosboot. Estos ficheros incluyen un fichero de cabecera llamado 'asm.h' empleando #include "asm.h", una instruccion del preprocesador. Este asm.h contiene varias macros bastante utiles, por ejemplo:

(fragmento de asm.h de FreeBSD)

```
...

#define String       .ascii
#define Value        .word
#define Times(a,b)   (a*b)
#define Divide(a,b)  (a/b)

#define INB          inb    %dx, %al
#define OUTB         outb   %al, %dx
#define INL          inl    %dx, %eax
#define OUTL         outl   %eax, %dx

...
```

La ventaja de usar el preprocesador es evidente. Otro ejemplo mas mundano de ensamblador con preprocesamiento podria ser el siguiente (hazlo tu mismo):

1) Crea un fichero 'arch.h' con las siguientes lineas:

```
#ifdef i686
#define ALGN 16,,8
#define IFR(C1, C2, FROM, TO)   cmpl C1, C2      ; \
                                cmovel FROM, TO
#else /* i686 */
#define ALGN 4
#define IFR(C1, C2, FROM, TO)   cmpl C1, C2      ; \
                                jnz L8f          ; \
                                movl FROM, TO    ; \
                                .align 4         ; \
```

```
                                        L8:
       #endif /* i686 */
```

NOTA: En gas, el separador de varias instrucciones dentro de la misma linea es el simbolo ';', exactamente igual que en C. Mira las macros de arriba: cada vez que hay que separar dos instrucciones en la misma linea empleamos el ';'. Aunque el preprocesador expanda la macro IFR en una sola linea, gas sera perfectamente capaz de diferenciar los comandos individuales.

2) Ahora crea un fichero 'dumb.S' con las siguientes lineas:

```
       #define i686
       #include "arch.h"

           .file "dumb.S"
           .text

       nada:
           .align ALGN      /* Alineamiento dado por arch.h */
           nop
           IFR(-5(%edx), %eax, %esi, %ebx)
           nop
```

3) Desde la shell, ejecuta el preprocesador:

```
          demeter# cpp dumb.S > dumb.i

          demeter# ls
          dumb.i dumb.S
```

4) Edita dumb.i, y obtendras:

```
           ( algunos datos del preprocesador al principio )

           .file "dumb.S"
           .text

         nada:
           .align 16,,8
           nop
           cmpl  -5(%edx),   %eax    ; cmovel  %esi,   %ebx
           nop
```

Este programa en ensamblador no hace nada (util), pero es un buen ejemplo de preprocesamiento.

Ahora veamos como se compila un fichero '.S'. No puede ser mas sencillo. Para compilar 'dumb.S' simplemente debes recurrir a gcc, como si de un programa en C se tratara. Dado que nuestro programa en ensamblador no es un programa principal (le falta el _main), lo compilaremos como un modulo:

```
          demeter# gcc -c dumb.S

          demeter# ls
          dumb.S  dumb.i  dumb.o
```

Ya que hemos llegado hasta aqui, confirmemos que esta todo

bien compilado:

```
demeter# gdb dumb.o

(gdb saluda y suelta la retahila)

(gdb) disassemble nada

0x0 <nada>:     nop
0x1 <nada+1>:   cmp 0xfffffffb(%edx), %eax
0x4 <nada+4>:   cmove %esi,%ebx
0x7 <nada+7>:   nop
0xa <nada+10>:  add %al,(%eax)
...
End of assembler dump.
(gdb)
```

Bueno, aqui esta nuestro programa, bien compilado. Recuerda lo que indicamos al final de la seccion 1.2: aunque no hemos declarado como .global a la etiqueta 'nada', el gdb la reconoce perfectamente. Esto sucedera con todas las etiquetas no locales que empleemos. De paso, esta version de 'gdb' tiene la mala costumbre de comerse los sufijos de magnitud cuando no son necesarios.

Resumiendo, puedes usar 'gcc' para compilar un programa '.S' exactamente de la misma manera que con un programa en C. Esto ayuda bastante a la hora de definir makefiles, aunque por lo general deberas separar los ficheros '.S' de los '.c' en el makefile, puesto que gcc suele llevar opciones de compilacion diferentes para ambos. Hacer 'gcc -O3' a un fichero en ensamblador no es incorrecto, pero no tendra ningun efecto y es inelegante.

----------| PARTE 2 : Ensamblando subrutinas |----------

La manera mas sencilla de programar en ensamblador consiste en hacer el esqueleto del programa en C y desarrollar las subrutinas mas importantes en ensamblador. Esta estrategia es muy eficente, dado que descarga todo el trabajo pesado en el compilador, y todo el trabajo delicado en el programador.

Sin embargo, existe un desventaja. Si la subrutina es muy corta, la perdida de tiempo asociada a las manipulaciones de pila y los saltos ('call' y 'ret') pueden contrarrestar las ventajas de usar ensamblador. En estos casos es mejor emplear el ensamblado 'inline' (ver la parte 3).

Estudiaremos dos modos fundamentales de programar subrutinas. El modo 'canonico', que consiste en emplear el puntero base de pila %ebp para referirse a las variables locales. Cualquier subrutina compilada con 'gcc -S' usara este modo. El modo 'no canonico' consiste en emplear el puntero de pila %esp como referencia para las variables locales, lo que nos permite ratear el registro %ebp para nuestro propio uso. Puedes generar ejemplos de este tipo a partir de ficheros '.c' compilando con 'gcc -fomit-frame-pointer -S'.

---| 2.1 Normas de llamada a subrutinas.

Para entender lo que sigue, debes de tener bien claro lo que es la pila de tu microprocesador, y como funciona. Debes saber como se altera la pila al hacer una instruccion tipo 'call', 'ret', 'push', 'pop', 'enter', 'leave', etc. Si no tienes mucha idea, consulta cualquier buen libro de ensamblador (si eres muy novato, mejor evita los 'white papers' al principio y consiguete una buena introduccion).

Es tal la ola de ideologia RISC que nos invade, que posiblemente no necesites conocer algunas instrucciones 'vectoriales', tales como 'enter' y 'leave'. Algunos manuales de microprocesadores modernos desaconsejan su uso, pero como puedes ver (consulta el codigo en la seccion 1.1) el compilador 'gcc' sigue siendo adepto a ellas.

Ademas, en todos los siguientes codigos de ejemplo supondremos que se esta trabajando con ensamblador con preprocesador (extension '.S', consulta la seccion 1.4). Esto no es esencial, y es muy facil conseguir el codigo preprocesado (usa 'gcc -save-temps'), pero nos valdra como practica.

Las normas que vamos a exponer son validas para casi cualquier compilador de C, pero sobre estas cosas nunca puedes estar del todo seguro. Cuando menos, valen para gcc, y _deberian_ valer para los demas. Tambien son validas con pocas modificaciones para FORTRAN y Pascal.

2.1.1. Como llama el compilador de C a una subrutina.

Conceptualmente, las subrutinas son subprogramas a los que se llama desde un cierto programa, mandandoles una cierta cantidad de datos de manera que la subrutina devuelva uno o ningun dato.

Asi pues, la llamada a la subrutina tiene tres componentes:

a) El programa principal le envia ciertos datos.
b) El programa principal le cede el control.
c) La subrutina devuelve uno o ningun datos.

Estas componentes se gestionan como sigue:

-> Puesto que los datos que se envian a una subrutina en C son desechables, lo mas logico es introducir esos datos en la pila _antes_ de llamar a la subrutina. El propio modulo que llama a la subrutina se encarga luego de desechar los valores de la pila.

-> Puesto que la subrutina puede ser llamada desde varios modulos diferentes, la cesion de control (componente (b)) debe realizarse mediante una instruccion 'call'. La direccion de retorno quedara entonces contenida en la pila hasta que la subrutina devuelva el control al llamador, mediante un 'ret'.

-> Dado que el valor devuelto es tan solo de un tipo de dato (caracter, puntero, en coma flotante, etc.) es logico devolver ese dato en un registro, dado que la pila de la subrutina es desechada al salir de la misma. Tambien pueden devolverse datos de tipo mas complejo, como estructuras, pero no estudiaremos esto en el tutorial.

Por lo tanto, ya podemos hacer un esquema en ensamblador de como un cierto modulo llama a una subrutina. Los pasos a

seguir deben ser:

1) Se meten en la pila los argumentos que se mandan a la subrutina. Por convenio, los argumentos se empujan de derecha a izquierda (es decir, en orden inverso a como es declarada la subrutina). Mas abajo tienes un ejemplo.

2) A continuacion se hace un 'call' a la subrutina. Esto cede el control a la misma.

3) Una vez que el modulo retoma el control, se eliminan los argumentos empujados a la pila, usualmente con un 'subl CANTIDAD, %esp', donde CANTIDAD es el numero de bytes reservados previamente (p. ej., CANTIDAD == $16, o lo que sea).

4) Por convenio, el dato devuelto esta almacenado, segun su tipo, en los siguientes registros:

```
Puntero        :  %eax
Caracter       :  %al
Palabra        :  %ax
Dpalabra       :  %eax
Qpalabra       :  %edx y %eax
Coma flotante  :  %st(0) (el TOS de la FPU)
```

5) Opcionalmente, el compilador de C hace algunos chanchullos para asegurarse de que la pila queda bien alineada en la subrutina. No nos preocuparemos por esto (si tienes curiosidad por esto, mira los comentarios que hicimos al respecto en 1.1).

Con esto tenemos suficiente para dar un ejemplo: supongamos que un programa en C desea llamar a la siguiente subrutina,

```
extern struct barf * llama(long , char , short );
```

de manera que la subrutina devuelve un puntero, y tiene como argumentos formales una palabra, un caracter y una dpalabra. En nuestro codigo C podemos hacer la llamada con la instruccion:

```
static long largo;
static char corto;
static short medio;
static struct barf *ptr;

main()
{
    ...

    ptr = llama(largo, corto, medio);

    ...
}
```

la llamada se compilaria como:

```
...

pushw _medio
pushb _corto
pushl _largo         /* Hasta aqui 1) */
```

```
        call _llama          /* Esto es 2)    */

        addl $7, %esp        /* Esto es 3)    */

        movl %eax, _ptr      /* Esto es 4)    */

        ...
```

Observa que hemos hecho que todas las variables enviadas sean estaticas para facilitar la notacion. Si las variables largo, medio, corto y _ptr fueran automaticas, las instrucciones 'push' estarian referidas al puntero de pila, que es donde se almacenan las variables automaticas en un programa en C.

Nota que la cantidad de bytes metidos en la pila es de 7 en total, que no es multiplo de 16. El compilador de C, para mantener el alineamiento, posiblemente incluira una instruccion 'subl $9, %esp' antes del codigo que hemos descrito, y un 'addl $16, %esp' en vez del 'addl $7, %esp', todo ello con la noble intencion de acelerar el codigo a costa de unos cuantos bytes de pila.

2.1.2. Como se comporta una subrutina en C.

Ahora que ya sabemos lo que el compilador espera de nosotros, es relativamente sencillo emular una subrutina C en ensamblador.

En primer lugar, deberemos darle un nombre a la subrutina y hacer que ese nombre sea visible para el linkador. Esto se logra haciendo:

```
         .text

         .global _llama
        _llama:
```

al principio de la subrutina. Ahora el linkador ya sabe a donde tiene que saltar cuando se encuentre con el 'call _llama' en el modulo en C.

Una vez que el modulo nos ha cedido el control, tenemos que ocuparnos de lo que pasa en la pila, suponiendo que nuestra subrutina maneje algunos datos. Como suponemos que nuestra subrutina es la 'llama' definida mas arriba, sabemos que nos estan mandando 3 variables en la pila.

Justo en el momento en que el control se cede a nuestra subrutina, la estructura de la pila es la siguiente:

```
        ...          ...            (tamaÃ±o)

      9(%esp) --->  medio           (16 bits)
      8(%esp) --->  corto           ( 8 bits)
      4(%esp) --->  largo           (32 bits)
      0(%esp) --->  %eip de retorno (32 bits)
```

Para darse cuenta no hay mas que mirar como ingresaron los datos en la pila, en el codigo de ejemplo de la seccion 2.1.1.

Es con estos datos con los que tenemos que trabajar para desarrollar el codigo de la subrutina. Como veremos, hay dos estrategias: sacrificando un registro para acceder a la pila ordenadamente (modo 'canonico'), o accediendo a ella desordenadamente, pero ahorrando un registro (modo sin 'frame pointer').

UNA REGLA FUNDAMENTAL:

Como puedes ver, el modulo en C demuestra una extraordinaria confianza en las subrutinas a las que llama. El modulo llamador depende muy probablemente del contenido de registros como %ebp y %ebx (y quizas de muchos otros) para funcionar correctamente. Si al volver de la subrutina has modificado %ebp, %ebx u otro registro que use tu llamador, habras saboteado con toda seguridad su funcionamiento.

La norma acerca de registros para gcc es la siguiente:

NORMA: Los registros %esp, %ebp, %ebx, %esi, %edi _deben_ ser devueltos intactos por toda subrutina. Lo mismo se aplica, por supuesto, a los registros de segmento.

COROLARIO: Puedes emplear libremente los registros %eax, %ecx, %edx en el interior de tu subrutina.

Olvidar esta regla es dejar una puerta abierta al caos, la desesperacion, la locura y lo peor de todo, el mensajillo 'segmentation fault - core dumped'.

UNA CONVENCION IMPORTANTE:

El compilador 'gcc' se emplea en muchos sistemas opertivos, como FreeBSD, NetBSD, linux, windows, etc. Lamentablemente, los que implementaron gcc en esos sistemas hicieron convenios distintos acerca de como se llama a las rutinas en ensamblador desde C.

El siguiente convenio es necesario para cygwin (implementacion de 'gcc' sobre windows), y posiblemente para otros sistemas:

Cuando programes una subrutina en ensamblador con la intencion de que se acceda a ella desde C, debes tener en cuenta que todos los simbolos que exportes con .globl o .global deben ir precedidos por un signo de subrayado '_'.

Por ejemplo, para definir una subrutina en ensamblador llamada 'procesa', que pueda ser llamada desde C, debes usar en tu programa ensamblador la directiva:

```
        .text
        .global _procesa
        _procesa:

         ...   (el resto de la subrutina)
```

Lo mismo vale para una variable estatica que quieras que sea visible desde C:

```
        .data
        .global _cadena
```

```
_cadena:
.asciz "Hola, mundo!\n"
```

Cualquier programa escrito en C podra acceder a 'procesa' o 'cadena' por su nombre, sin el subrayado, mientras que cualquier programa en ensamblador accedera a ellos con el nombre completo, con subrayado. Esto se ha pensado para evitar colisiones accidentales de nombres entre el compilador y el ensamblador.

En otras implementaciones de 'gcc' el subrayado puede no ser necesario. Para comprobarlo, basta que compiles con -S algun programa en C y veas como se declaran los nombres de las subrutinas.

Para curarnos en salud, en este tutorial hemos empleado el subrayado en todos los ejemplos.

---| 2.2 Subrutinas 'canonicas'.

Con la informacion obtenida en 2.1 ya tenemos suficentes datos como para empezar a programar subrutinas. Veamos el primer tipo.

Llamamos subrutinas 'canonicas' a las que se comportan de acuerdo con la tradicion de codificacion en C. Esta tradicion indica que hay que utilizar el registro %ebp para referirse a las variables locales, que estan en la pila. Se dice entonces que %ebp actua como 'frame pointer'. Si no se respeta este convenio, el debugger 'gdb' puede mostrarse algo confuso acerca del contenido de la pila de una subrutina.

La definicion de un nuevo 'frame pointer' se consigue con el siguiente mecanismo (siempre tomamos como ejemplo la subrutina 'llama' de la seccion anterior):

```
.text

.global _llama
_llama:

pushl %ebp       /* Almacenamos el viejo frame pointer. */
movl %esp, %ebp  /* Nuevo frame pointer.                */
subl $12, %esp   /* Espacio para variables automaticas. */

...              /* Resto de la subrutina.              */

addl $12, %esp   /* Se liberan las variables automat.   */
popl %ebp        /* Se recupera el viejo frame pointer. */
ret              /* Retorno a modulo llamador.          */
```

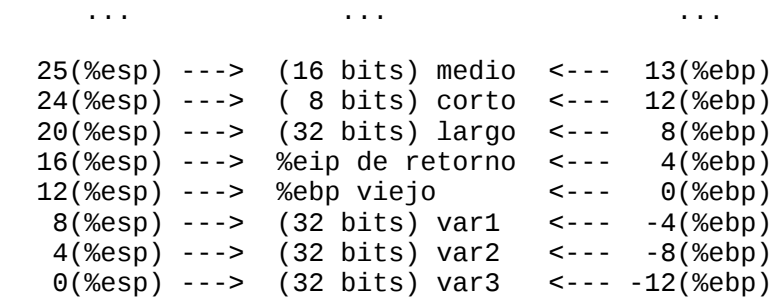
las tres instrucciones finales pueden condensarse en dos:

```
leave
ret
```

esto es mas compacto, pero puede ser levemente ineficiente en algunas maquinas.

La idea que se persigue con el 'frame pointer' es tener un acceso facil a las variables automaticas de la subrutina. Si

observamos la estructura de la pila despues de la instruccion 'subl $12, %esp', tenemos:



Hemos referido la pila a los dos punteros, %esp y %ebp. A las variables reservadas (tres enteros largos, 12 bytes en total) las hemos llamado var1, var2 y var3.

La ventaja de usar un 'frame pointer' para nuestra subrutina es evidente si suponemos que la pila se altera durante la ejecucion de la subrutina. Por ejemplo, imaginemos que el codigo de nuestra subrutina es:

```
.text

.global _llama
_llama:

pushl %ebp       /* Almacenamos el viejo frame pointer. */
movl %esp, %ebp  /* Nuevo frame pointer.                */
subl $12, %esp   /* Espacio para variables automaticas. */

pushl %ebx       /* Conservamos para no corromperlo.    */
xorl %ebx, %ebx  /* Resto de la subrutina ...           */
...
```

Despues de la instruccion 'pushl %ebx' el puntero de pila %esp ha disminuido en 4 unidades, por lo que ahora la variable 'var1' ha pasado a ser 12(%esp), ya no 8(%esp). Y el argumento 'corto' ha pasado a ser 28(%esp), cuando antes era 24(%esp).

Esto demuestra que encontrar las variables locales, o los argumentos de la subrutina, es bastante complicado si usas %esp, dado que tienes que tener una idea exacta del contenido de la pila _en_el_mismo_instante_de_acceder_a_ella_.

Sin embargo, dado que el puntero %ebp no ha cambiado durante el 'pushl %ebx', las variables y argumentos siguen teniendo la misma direccion que al principio. Es muy facil referirse a variables y argumentos a traves de %ebp.

NORMA: Si usas el 'frame pointer' pierdes un registro de proposito general pero ganas legibilidad y comodidad a la hora de acceder a variables automaticas y argumentos de la subrutina. Cualquier modificacion que hagas a un programa ensamblador _sin_ 'frame pointer' podria obligarte a revisar todo el codigo, si es que se produce una alteracion en el orden de la pila.

Debes calibrar cuidadosamente si quieres usar %ebp como 'frame pointer' o no. Las unicas ventajas son de legibilidad, mantenimiento

del codigo, y debugging. No se obtienen ventajas evidentes en terminos de velocidad o espacio.

---| 2.3 Un ejemplo de subrutina 'canonica'.

Vamos a ver un ejemplo formado por un programa principal escrito en C y una subrutina en ensamblador. Lo que hara el programa sera pasar una cadena de caracteres a mayusculas, y devolver el numero de caracteres procesados en una variable tipo short.

Aqui tenemos el programa principal, main.c:

```
/******** Aqui empieza el fichero main.c **********/

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

extern short procesa (char *);

main()
{
   char *c;
   short l;

   c = (char *) malloc(256 * sizeof(char));
   if (c == NULL) exit(1);

   printf("Introduce cadena (>255 caracteres): ");
   scanf("%255s", c);

   l = procesa(c);

   printf("\n\nCadena: %s\n", c);

   printf("Caracteres: %d\n", l);

   free(c);

   exit(0);
}

/*********** Fin de main.c **************/
```

y ahora el segundo fichero, sub.S:

```
 /********** Aqui empieza sub.S *********/


 #define ALINEA 16,,8

 .file "sub.S"

 .text

 .globl _procesa
 .align ALINEA
 _procesa:

 pushl %ebp
```

```
        movl %esp, %ebp
        subl $2, %esp             /* 2 bytes de variables,      */
                                  /* ver abajo.                 */

        pushl %ebx                /* Usaremos este registro.    */

        /* En este momento:
         *    -2(%ebp) == variable a devolver (16 bits).
         *     8(%ebp) == puntero a caracteres.
         */

         xorw %ax, %ax
         movw %ax, -2(%ebp)       /* Contador a cero.           */

         movl 8(%ebp), %eax       /* %eax apunta a la cadena.   */

         .align ALINEA            /* Alinea salto.              */
         L1:

         movb (%eax), %bl         /* Caracter en %bl.           */
         testb %bl, %bl           /* Fin de la cadena ?         */
         jz Lfin
         incw -2(%ebp)            /* Aumenta contador.          */
         andb $0xdf, (%eax)       /* Pasa a mayuscula.          */
         incl %eax                /* Siguiente elemento.        */
         jmp L1                   /* Cierra el lazo.            */

         .align ALINEA
         Lfin:

         movw -2(%ebp), %ax       /* Valor de retorno.          */

         popl %ebx                /* Recupera registro.         */
         addl $2, %esp            /* Libera vars. locales.      */
         popl %ebp                /* Libera stack frame.        */
         ret                      /* Hecho.                     */

        /************** Fin de sub.S  **************/
```

El programa es extraordinariamente tonto, pero consigue expresar la mayor parte de los conceptos que hemos visto. Observa en particular el uso de etiquetas locales, el uso de sufijos de todos los tipos en los operadores, y el uso del preprocesador. Tambien hemos usado un registro de mas, %ebx, que hemos tenido el cuidado de conservar.

Para compilar este programa, y ejecutarlo:

```
demeter# gcc -o cap main.c sub.S

demeter# ./cap
Introduce cadena (>256 caracteres):
HoLaYaDiOs

Cadena: HOLAYADIOS
Caracteres: 10

demeter#
```

Usando este ejemplo como punto de partida, puedes desarrollar casos mucho mas perfeccionados.

Ten en cuenta que el anterior ejemplo esta concebido como codigo de ejemplo, no como un codigo eficiente. Un par de correciones al mismo serian muy pertinentes. Por ejemplo, para mantener la pila alineada respecto a 4 bytes, se deberia usar 'subl $4, %esp' en vez de 'subl $2, %esp' al principio de la subrutina, incluso si de esos 4 bytes solo vamos a usar 2. Ademas, en vez de %ebx podriamos haber usado %edx, lo que nos habria ahorrado el andar conservando el registro en la pila. Mas aun, usando %ecx podriamos habernos ahorrado el uso de la variable local. Pero con estas correciones creo que el codigo habria sido mucho menos instructivo.

---| 2.4 Subrutinas sin 'frame pointer'.

Entendiendo correctamente la seccion 2.2, es facil darse cuenta de en que consiste este tipo de subrutinas. Sencillamente, no empleamos el puntero %ebp como 'frame pointer', lo que exige usar algo mas el coco.

Como primer paso, reescribamos la subrutina del ejemplo anterior sin 'frame pointer'. Ademas de eliminar el uso de %ebp, cambiaremos de posicion el par de comandos 'pushl %ebx', 'popl %ebx' para que puedas apreciar como varia la manera de referirse a la variable local, que en unos momentos es (%esp) y en otros 4(%esp).

Para usar esta subrutina, guardala en un fichero ('sub2.S', por ejemplo), y compilala como la anterior.

```
/*************** Aqui empieza sub2.S *************/


#define ALINEA 16,,8

.file "sub2.S"

.text

.globl _procesa
.align ALINEA
_procesa:

subl $2, %esp           /* 2 bytes de variables,      */
                        /* ver abajo.                 */

/* En este momento, la estructura de la pila es:
 *
 *  6(%esp) --> puntero que nos han pasado.
 *  2(%esp) --> %eip de retorno.
 *  0(%esp) --> variable local (16 bits)
 */

xorw %ax, %ax
movw %ax, (%esp)        /* Contador a cero.           */

movl 6(%esp), %eax      /* %eax apunta a la cadena.   */

pushl %ebx              /* Usaremos este registro.    */

/* La pila ha cambiado a:
 *
 * 10(%esp) -->  puntero que nos han pasado.
 *  6(%esp) -->  %eip de retorno
```

```
         *  4(%esp) -->  variable local (16 bits)
         *  0(%esp) -->  antiguo %ebx
         */

        .align ALINEA            /* Alinea salto.              */
        L1:

        movb (%eax), %bl         /* Caracter en %bl.           */
        testb %bl, %bl           /* Fin de la cadena ?         */
        jz Lfin
        incw 4(%esp)             /* Aumenta contador.          */
        andb $0xdf, (%eax)       /* Pasa a mayuscula.          */
        incl %eax                /* Siguiente elemento.        */
        jmp L1                   /* Cierra el lazo.            */

        .align ALINEA
        Lfin:

        popl %ebx                /* Recupera registro.         */

        /*  La pila ha vuelto a cambiar:
         *
         *  6(%esp) --> puntero que nos han pasado.
         *  2(%esp) --> %eip de retorno.
         *  0(%esp) --> variable local (16 bits)
         */

        movw (%esp), %ax         /* Valor de retorno.          */

        addl $2, %esp            /* Libera vars. locales.      */
        ret                      /* Hecho.                     */

       /************** Fin de sub2.S  **************/
```

Este programa apenas tiene un tamaÃ±o insignificantemente menor que el anterior, pero da una idea del metodo. Ahora todas las referencias de memoria son relativas a %esp, y dispondriamos del registro %ebp si estuvieramos realmente cortos de registros libres.

Cuantos mas 'push', 'pop', 'call' o cualquier otra instruccion que modifique %esp en medio del codigo, mas liante se vuelve la subrutina.

----------| PARTE 3 : Ensamblando 'inline' |----------

El ensamblado 'inline' consiste en la introduccion de codigo ensamblador en el interior de un codigo fuente en C. Esta tecnica es tan vieja como el lenguaje C, y el ensamblador 'gcc' posee unas capacidades extraordinarias para incluir ensamblador 'inline' en los programas en C, llegando a optimar la interaccion entre ambos lenguajes.

Todo esto viene a un precio. Hay que dar al compilador una informacion muy completa de los datos que nuestro codigo ensamblador usa como entrada, de los que usa como salida, y de los que usa como almacenamiento temporal. Sin esta informacion 'gcc' no podra optimar nuestro codigo y, lo que es peor, hasta puede ser inducido a error.

Cuando tiras una piedra en una laguna, no solo la zona de

impacto es afectada por la piedra, si no que el efecto se expande por toda el agua. De la misma manera, introducir un codigo extraÃ±o en un programa C puede causar fallos donde menos lo esperas, si no avisas al compilador de lo que tu codigo hace exactamente.

En esta parte veremos dos tipos fundamentales de ensamblado dentro de C. El primero, al que llamaremos 'metodo Frankenstein', consiste en introducir el ensamblador a lo bruto, en plan protesis, usando algunas tecnicas elementales para asegurarte de que las cosas van a funcionar. El metodo Frankenstein es el que se usaba en los viejos tiempos. Su ventaja es que casa bien con la ley universal "cuando tengas dudas, usa la fuerza bruta"; es un metodo rapido y directo. Su desventaja es que nada te asegura que un cambio en la version del compilador, en las opciones de compilacion o en el codigo en C no vayan a desestabilizar completamente el programa.

El segundo metodo es el feten. Consiste en decirle a gcc en su propio lenguaje todo lo que necesita saber para que el trabajo quede bien hecho. Esta opcion tiene todas las ventajas, excepto que es un poco latosa de aprender. Bueno, tambien es menos divertida ;).

---| 3.1 El metodo Frankenstein.

La idea es simple, brutal y victoriana. Tomas los diferentes pedazos (brazos, piernas, placas de metal...), lo coses todo y lo echas a correr. El resultado suele ser feo y demoniaco, pero contundente. Como hemos aprendido en innumerables peliculas de serie B, eso de jugar a ser Dios conlleva el pago de un alto precio... asi que preparate para enfrentarte a los monstruos que vas a crear.

Bromas aparte, el metodo Frankenstein consiste en los siguientes pasos:

1) Escribe el codigo en C que quieres modificar.

2) Compilalo con 'gcc -S' y localiza la zona en la que quieres injertar el ensamblador.

3) Observa bien la zona de implantacion para asegurarte de que el injerto sera compatible con el codigo.

4) Injerta el ensamblador en el codigo C. Esto se hara usualmente en forma monolitica (tus instrucciones en ensamblador no deberan ser procesadas por el compilador, que se limitara a pasarlas al ensamblador 'tal cual').

5) Compila el codigo injertado con 'gcc -S' y comprueba que la cosa marcha como tu quieres.

6) Ahora compila y linka el programa injertado.

7) Ejecutalo. Si funciona, debes lanzar el grito ritual: "Estaaaaa viiiiivooo! Muhahahahah!" Ya estas listo para castigar al mundo con tu creacion infernal.

3.1.1 Insercion de ensamblador.

Para insertar ensamblador en un codigo en C hay que utilizar el comando 'asm'. Sin embargo, para evitar advertencias por parte del compilador (por ejemplo, compilando con comaptibilidad ANSI),

es mejor usar '__asm__', con dos subrayados antes y despues de
la directiva.

    Como primer ejemplo, imaginemos que queremos introducir una
etiqueta inocua en el codigo en C, de manera que al ejecutar 'gdb'
podamos saltar rapidamente a esa parte de codigo. Si tenemos un
programa como el siguiente (guardalo en un fichero 'memo.c'):

```
#include <stdio.h>

main (int argc, char *argv[])
{
    int i;

    puts("Este programa ha sido invocado como:");

    for (i = 0 ; i < argc ; i++)
    {
        printf("%s", argv[i]);
        putchar(' ');
    }

    putchar('\n');

    exit(0);
}
```

otro de nuestros programas tontos patentados. Lo que hace esta
claro:

```
demeter# gcc -o memo memo.c

demeter# ./memo jarl de peich morl!
Este programa ha sido invocado como:
/root/tut/memo jarl de peich morl!

demeter# ./memo *.c
Este programa ha sido invocado como:
/root/tut/memo bitm.c main.c memo.c test.c

demeter#
```

Etcetera. Ahora supongamos que queremos definir un 'breakpoint'
para gdb entre el 'printf' y el 'putchar' del bucle. Para ello
introducimos entre ambas instrucciones un fragmento de ensamblador:

```
...
printf("%s", argv[i]);
__asm__(" .global _brk ; _brk: ");
putchar(' ');
...
```

Bien, ahora volvemos a compilar y ejecutamos gdb:

```
demeter# gcc -o memo memo.c

demeter# gdb memo
(gdb saluda y se identifica)

(gdb) disassemble brk
Dump of assembler code for function brk:
<brk>:    add $0xfffffff4, %esp
```

```
        <brk+3>:  call <putchar>
        ... (mas lineas en ensamblador)
        End of assembler dump.
        (gdb)
```

Este diminuto (e inocuo) parche en ensamblador nos permite fijar la atencion en la parte del fichero que nos interesa. Como primer ejemplo de inoculacion de codigo, es bastante util.

    En general, para insertar lineas mas largas de ensamblador todo lo que necesitas es usar habilmente el separador de comandos ';' de gas y el simbolo de union de lineas '\' del compilador. Un ejemplo hipotetico seria:

```
       ... (codigo en C)

       __asm__(" .globl _franky       ;" \
               " _franky:             ;" \
               "      addl $4, %esp   ;" \
               "      incl %eax       ;" \
               "      leave           ;" \
               "      ret              ");

       ... (codigo en C)
```

Las normas son:

      1) Cada comando en ensamblador va entre comillas.
      2) Usa un ';' para separar los comandos.

Si compilas con 'gcc -S' veras que este codigo injertado se escribe en una sola linea, con los ';' como separadores. Si quieres que al hacer 'gcc -S' el codigo en ensamblador se vea mas agradable, en lineas separadas, en vez del ';' usa el par '\n\t'. Por ejemplo:

```
       __asm__(" .globl _franky       \n\t" \
               " _franky:             \n\t" \
               "      addl $4, %esp   \n\t" \
               "      incl %eax       \n\t" \
               "      leave           \n\t" \
               "      ret              ");
```

Aunque es algo molesto de escribir, el resultado estetico vale sobradamente la pena.

    Si has compilado con '-S' alguno de los ejemplos anteriores, habras notado que el codigo injertado aparece en el ensamblador entre los comandos '/APP' y '/NO_APP', por ejemplo:

```
       /APP
           .globl _brk ; _brk:
       /NO_APP
```

Estos comandos le dicen al ensamblador que las lineas que has injertado han sido escritas por un humano, por lo que el ensamblador debe tener especial cuidado al leerlas. A donde hemos ido a parar. Los ordenadores nos tratan con condescendencia! En todo caso, gracias a esto podemos localizar rapidamente los comandos injertados en el fichero '.s'.

    Resumiendo:

      1) Para injertar ensamblador en gcc, basta con usar

la directiva __asm__(" ... "); .

2) La cadena entre parentesis se puede romper en varias, que se consideraran parte de una sola linea del ensamblador. En forma general: __asm__(" ... " " ... " ... " ... "); .

3) Para separar comandos de ensablador, o bien usas ';', lo cual es sencillo pero queda feo en el fichero '.s', o bien usas '\n\t', que te hace escribir algo mas pero resulta mas bonito en el '.s'.

4) Tu codigo injertado va en el fichero '.s' entre los comandos '\APP' y '\NO_APP'.

3.1.2 Injertos monoliticos.

El compilador 'gcc' es lo bastante inteligente como para darse cuenta de que ciertas lineas en ensamblador que introduces no son necesarias, lo cual le puede llevar a eliminarlas. Tambien puede ocurrir que decida mezclar tus lineas en ensamblador con las que el genera, para acelerar el codigo.

Si estas haciendo un injerto a lo bruto, esto puede ser de lo mas indeseable. Para decirle al compilador que nuestro codigo es peligroso se emplea el comando '__volatile__', justo despues del comando '__asm__'. Es decir:

__asm__ __volatile__(" ... "),

donde la (o las) cadenas en el interior de los parentesis van exactamente igual que las descritas en la seccion anterior.

El compilador entiende el '__volatile__' como un signo de peligro y no tocara lo que haya dentro, en la medida de lo posible.

3.1.3 Un ejemplo retorcido.

Naturalmente. Los ejemplos de injertos son siempre retorcidos.

Para apartarnos un poco de nuestra politica de programas idiotas, consideremos un ejemplo un poco mas pintoresco. El siguiente programa genera un fichero llamado 'sal.bmp', que contiene el dibujo de una espiral de Arquimedes, en blanco y negro y resolucion de 300x300 pixels.

Almacena el siguiente programa en un fichero ('bitm.c'):

```
/************ Aqui comienza 'bitm.c' ***************/

#include <stdio.h>
#include <math.h>

/* Cabecera de un fichero BMP 300x300 monocromo. */

static char head[] = { \
0x42,0x4d,0x1e,0x2f,0,0,0,0,0,0,0x3e,0,0,0,0x28,0, \
0,0,0x2c,0x01,0,0,0x2c,0x01,0,0,0x01,0,0x01,0,0,0, \
0,0,0xe0,0x2e,0,0,0xc4,0x0e,0,0,0xc4,0x0e,0,0,0,0, \
0,0,0,0,0,0,0,0,0,0,0xff,0xff,0xff,0};
```

```
/* Mascara que localiza un bit en un byte. */

static char masc[] = {128,64,32,16,8,4,2,1};

/* Constantes del BMP. */

#define TAM (40*300)
#define LIN 40
#define HDSIZ 62

#define RADIO 3.0   /* Escala del radio de la espiral. */

main()
{
     char *imag, *alias;
     FILE *fich;
     unsigned long i, j;
     double x,y,t;

     if ((fich = fopen("sal.bmp","w")) == 0)
         { puts("Error: fopen"); exit(1); }

     imag = (char *) malloc(TAM * sizeof(char));

     if (imag == NULL) { puts("Error: malloc"); exit(1); }

     /* Escribe cabecera. */

     for (i = 0 ; i < HDSIZ ; i++)
         putc(head[i], fich);

     /* Inicializa imagen */

     alias = imag;

     for (i = 0 ; i < 300 ; i++)
     {
         for (j = 0 ; j < (LIN-3) ; j++)
             *(alias++) = 0xff;

         *(alias++) = 0xf0;
         *(alias++) = 0;
         *(alias++) = 0;
     }

     /* Genera imagen.                    */

     for (t = 0 ; t < 36 ; t += 0.001)
     {
     /* Calculamos coords. de la espiral. */

        x = 150.0 + RADIO * t * cos(t);
        y = 150.0 + RADIO * t * sin(t);

     /* Eliminamos puntos fuera del BMP.  */

        if ((x > 300) || (x < 0)) continue;
        if ((y > 300) || (y < 0)) continue;

     /* Incluimos el punto en la imagen.  */

        j = (unsigned long) y;
        i = (unsigned long) (x/8);
```

```
            imag[(LIN*j)+i] &= ~masc[((unsigned long) x)%8];
        }

        /* Escribe imagen en fichero.        */

        for (i = 0 ; i < TAM ; i++)
            putc(imag[i], fich);

        free(imag);

        fclose(fich);

        exit(0);
    }

    /***************** Fin de 'bitm.c ***************/
```

Supongamos que queremos sustituir el calculo de las coordenadas de la espiral por una version en ensamblador. Nuestra primera tarea es localizar la zona de injerto, lo cual se puede lograr sencillamente intercalando un par de etiquetas que pasaran al ensamblador:

```
    ...
    __asm__ __volatile__("comienzo:");
    x = 150.0 + RADIO * t * cos(t);
    y = 150.0 + RADIO * t * sin(t);
    __asm__ __volatile__("final:");
    ...
```

Ahora compilamos con 'gcc -ffast-math -S' y hacemos una busqueda de la zona entre las etiquetas. El uso de la opcion '-ffast-math' ayuda a simplificar el codigo que queremos localizar. Veamos que es lo que encontramos:

```
     /APP
           comienzo:
     /NO_APP
           addl $-8,%esp
           fldl -48(%ebp)
           subl $8,%esp
           fstpl (%esp)
           call _cos
           addl $16,%esp
           fldl LC5
           fmulp %st,%st(1)
           fmull -48(%ebp)
           fldl LC6
           faddp %st,%st(1)
           fstpl -32(%ebp)
           addl $-8,%esp
           fldl -48(%ebp)
           subl $8,%esp
           fstpl (%esp)
           call _sin
           addl $16,%esp
           fldl LC5
           fmulp %st,%st(1)
           fmull -48(%ebp)
           fldl LC6
           faddp %st,%st(1)
           fstpl -40(%ebp)
     /APP
```

```
                final:
        /NO_APP
```

Este codigo nos permite formular algunas conjeturas:

1) -32(%ebp) almacena la variable x.
2) -40(%ebp) almacena la variable y.
3) -48(%ebp) almacena la variable t.
4) LC5 almacena la constante RADIO (que vale 3.0)
5) LC6 almacena la constante 150.0

que es mas o menos todo lo que necesitamos para desarrollar nuestro codigo. Pero antes de comenzar, observemos algo que puede causarnos muchos problemas:

-> Las constantes LC5 y LC6 han sido definidas por el
-> compilador precisamente porque las hemos usado en las
-> lineas de codigo que queremos sustituir. Asi que si ahora
-> eliminamos esas lineas, LC5 y LC6 ya no existiran, o bien
-> estaran asociados a otras constantes. En consecuencia,
-> nuestro codigo ensamblador debe definir estas dos
-> constantes (con otro nombre que no interfiera con C).

En esencia, tendremos que definir dos cantidades de doble precision en nuestro codigo ensamblador. Una sera 3.0 y la otra 150.0. Este tipo de cosas son las que te hacen amar los chanchullos con ensamblador.

Ahora que ya nos hemos salvado de este diabolico efecto colateral, pasemos a construir un codigo que haga el calculo de las cantidades que nos interesan. Por ejemplo, esto podria servir:

```
    .data
    Lradio:
        .double 3.0
    Lcentro:
        .double 150.0

    .text
                          /*       Pila de la FPU          */
        fldl -48(%ebp)    /* t                             */
        fld %st(0)        /* t : t                         */
        fldl Lradio       /* 3 : t : t                     */
        fmulp %st(2)      /* t : 3t                        */
        fsincos           /* Cos(t) : Sen(t) : 3t          */
        fmul %st(2)       /* 3tCos(t) : Sen(t) :3t         */
        fxch %st(2)       /* 3t : Sen(t) : 3tCos(t)        */
        fmulp %st(1)      /* 3tSen(t) : 3tCos(t)           */
        fldl Lcentro      /* 150 : 3tSen(t) : 3tCos(t)     */
        fadd %st(0),%st(2) /* 150 : 3tSen(t) : 150+3tCos(t) */
        faddp %st(1)      /* 150+3tSen(t) : 150+3tCos(t)   */
        fstpl -40(%ebp)   /* 150+3tCos(t)                  */
        fstpl -32(%ebp)
```

Lo cual, insertado en modo ensamblador seria:

```
    __asm__ __volatile__(" .data                    \n\t" \
                         " Lradio:                  \n\t" \
                         "     .double 3.0          \n\t" \
                         " Lcentro:                 \n\t" \
```

```
                        "     .double 150.0        \n\t" \
                        " .text                    \n\t" \
                        "     fldl  -48(%ebp)      \n\t" \
                        "     fld  %st(0)          \n\t" \
                        "     fldl  Lradio         \n\t" \
                        "     fmulp  %st(2)        \n\t" \
                        "     fsincos              \n\t" \
                        "     fmul  %st(2)         \n\t" \
                        "     fxch  %st(2)         \n\t" \
                        "     fmulp  %st(1)        \n\t" \
                        "     fldl  Lcentro        \n\t" \
                        "     fadd  %st(0), %st(2) \n\t" \
                        "     faddp  %st(1)        \n\t" \
                        "     fstpl  -40(%ebp)     \n\t" \
                        "     fstpl  -32(%ebp)     \n\t");
```

Prueba a sustituir las dos lineas en C que definen 'x' e 'y' por este injerto en ensamblador, y compila el resultado. Ten en cuenta que tu compilador podria almacenar 'x', 'y' y 't' en otras posiciones. Salvo esto, el codigo deberia funcionar de manera general.

Por si te has perdido, he aqui la forma final del codigo mixto para el programa en C anterior:

```
/***************** Aqui comienza bitm2.c *********************/

#include <stdio.h>
#include <math.h>

/* Cabecera de un fichero BMP 300x300 monocromo. */

static char head[] = { \
0x42,0x4d,0x1e,0x2f,0,0,0,0,0,0,0x3e,0,0,0,0x28,0, \
0,0,0x2c,0x01,0,0,0x2c,0x01,0,0,0x01,0,0x01,0,0,0, \
0,0,0xe0,0x2e,0,0,0xc4,0x0e,0,0,0xc4,0x0e,0,0,0,0, \
0,0,0,0,0,0,0,0,0,0,0xff,0xff,0xff,0};

/* Mascara que localiza un bit en un byte. */

static char masc[] = {128,64,32,16,8,4,2,1};

/* Constantes del BMP. */

#define TAM (40*300)
#define LIN 40
#define HDSIZ 62

#define RADIO 3.0   /* Escala del radio de la espiral. */

main()
{
    char *imag, *alias;
    FILE *fich;
    unsigned long i, j;
    double x,y,t;

    if ((fich = fopen("sal.bmp","w")) == 0)
        { puts("Error: fopen"); exit(1); }

    imag = (char *) malloc(TAM * sizeof(char));
```

```
if (imag == NULL) { puts("Error: malloc"); exit(1); }

/* Escribe cabecera. */

for (i = 0 ; i < HDSIZ ; i++)
    putc(head[i], fich);

/* Inicializa imagen */

alias = imag;

for (i = 0 ; i < 300 ; i++)
{
    for (j = 0 ; j < (LIN-3) ; j++)
        *(alias++) = 0xff;

    *(alias++) = 0xf0;
    *(alias++) = 0;
    *(alias++) = 0;
}

/* Genera imagen.                    */

for (t = 0 ; t < 36 ; t += 0.001)
{
/* Calculamos coords. de la espiral. */

__asm__ __volatile__(" .data                    \n\t" \
                     " Lradio:                  \n\t" \
                     "     .double 3.0          \n\t" \
                     " Lcentro:                 \n\t" \
                     "     .double 150.0        \n\t" \
                     " .text                    \n\t" \
                     "     fldl  -48(%ebp)      \n\t" \
                     "     fld  %st(0)          \n\t" \
                     "     fldl  Lradio         \n\t" \
                     "     fmulp  %st(2)        \n\t" \
                     "     fsincos              \n\t" \
                     "     fmul  %st(2)         \n\t" \
                     "     fxch  %st(2)         \n\t" \
                     "     fmulp  %st(1)        \n\t" \
                     "     fldl  Lcentro        \n\t" \
                     "     fadd  %st(0), %st(2) \n\t" \
                     "     faddp  %st(1)        \n\t" \
                     "     fstpl  -40(%ebp)     \n\t" \
                     "     fstpl  -32(%ebp)     \n\t");

/* Eliminamos puntos fuera del BMP.  */

   if ((x > 300) || (x < 0)) continue;
   if ((y > 300) || (y < 0)) continue;

/* Incluimos el punto en la imagen.  */

   j = (unsigned long) y;
   i = (unsigned long) (x/8);

   imag[(LIN*j)+i] &= ~masc[((unsigned long) x)%8];
}

/* Escribe imagen en fichero.        */

for (i = 0 ; i < TAM ; i++)
    putc(imag[i], fich);
```

```
    free(imag);

    fclose(fich);

    exit(0);
}

/********************* Fin de bitm2.c ************************/
```

---| 3.2 Introduccion al metodo ortodoxo.

El metodo Frankenstein esta sujeto a muchos fallos y es muy dificil de mantener. Si recordamos el ejemplo de la seccion 3.1.3, los problemas que afrontamos fueron:

1) Localizar las variables de entrada para nuestro codigo.
2) Localizar las variables de salida para nuestro codigo.
3) Asegurarnos de que no modificamos accidentalmente ningun dato necesario para el compilador.

El metodo 'ortodoxo' de ensamblado inline nos permite instruir a 'gcc' para que resuelva automaticamente estos tres problemas.

Intuitivamente, puedes suponer que se produce la siguiente situacion: tu codigo, con su informacion sobre 1), 2) y 3), es como una burbuja de codigo ensamblador metida entre el codigo ensamblador generado por 'gcc'. Dado que se tiene toda la informacion sobre como interactua ese codigo, en una fase avanzada de la compilacion el gcc rompe la burbuja de tu codigo y lo mezcla todo, optimando el resultado total.

La desventaja del metodo Frankestein es que la burbuja injertada es irrompible y, si entra en conflicto con el compilador, puede llegar a convertirse en un cancer para el codigo.

La anterior metafora puede hacernos suponer que nos vamos a comunicar con el compilador a un nivel bastante bajo. Tendremos que darle los datos bastante masticados para que los entienda. La cosa puede ser complicada si queremos injertar fragmentos de codigo muy largos; pero recuerda que estos ultimos casos los puedes meter en subrutinas (ver Parte 2).

El tema es delicado, asi que comencemos con algunos casos simples antes de pasar a la situacion general. Lee esta seccion de una vez, intentando captar la idea general; mas adelante se iran perfilando los detalles.

En primer lugar, recordemos el codigo que usamos en la seccion 3.1 para incluir etiquetas en un programa C:

```
__asm__(" .globl _brk ; _brk: ");
```

Este injerto no tiene datos de entrada, no tiene datos de salida y no modifica el contenido de ninguna variable ni registro: sencillamente es una directiva y una etiqueta. Toda la informacion generada son algunos simbolos para el linkador, asi que no nos tenemos por que preocupar de que el compilador se indigeste. Este codigo no necesita modificacion para ser 'ortodoxo'.

Vamos a ver un ejemplo mucho mas ambicioso. Recuerda el ejemplo de 3.1.3, el codigo en coma flotante. Alli nuestro problema era que el codigo necesitaba incorporar ciertas variables de C, y no sabiamos muy bien como hacerlo. Observa que diferente es la aproximacion ortodoxa de la Frankestein:

Sustituyamos el calculo de las coordenadas de la espiral en 'bitm.c' por las tres lineas (guardalo en 'bitm3.c'):

```
__asm__("fsincos" : "=t" (x), "=u" (y) : "0" (t));

x =  (x * t * RADIO) + 150.0;
y =  (y * t * RADIO) + 150.0;
```

Lo que hemos hecho es:

a) Ejecutamos 'fsincos', con dato de entrada 't' y datos de salida 'x' e 'y'. Indicar al compilador estas entradas y salidas se logra con los comandos que van con los ':' al final de la primera linea (veremos la sintaxis de estos comandos algo mas abajo; por ahora olvidate de ello).

b) Ahora que x == Sen(t) e y == Cos(t), hacemos el calculo normal de las coordenadas, en las lineas restantes.

La ventaja de las tres lineas que tenemos arriba es que podemos compilar el codigo con cualquier nivel de optimacion, dado que el compilador se encargara de hacer los arreglos. Por ejemplo, usando 'gcc -O6' el codigo marcha a la perfeccion. Dado que a este nivel de optimacion el compilador opera perfectamente con sumas y productos, la unica optimacion relevante es la que le hemos pedido, es decir, que use 'fsincos'. Compila este programa con 'gcc -O6 -S' y observa hasta que punto optima el compilador el resto del codigo. Se ha logrado un resultado al menos tan bueno como nuestro Frankestein sin necesidad de escribir mas que una linea en ensamblador.

La idea del ensamblador inline bien hecho se reduce muchas veces a insinuar a 'gcc' que debe usar un comando complicado (como 'fsincos'), dejandole que haga el resto del trabajo pesado. Tambien puedes introducir fragmentos largos de ensamblador en el codigo, pero puede resultar una mala politica. Lograr una eficiencia como la de gcc con instrucciones de proposito general puede llevarte un tiempo que puedes emplear en otras cosas mejores.

Veamos otro ejemplo, que personalmente siempre he echado de menos en los lenguajes de alto nivel. A menudo, cuando operamos con C queremos saber si cierta instruccion de suma ha producido un acarreo, por ejemplo cuando se opera con enteros en precision arbitraria. Veamos una solucion al problema del acarreo:

```
#include <stdio.h>

main()
{
    unsigned short i, j, sum, acarreo;

    i = 40000;
    j = 50000;
```

```
            printf("Sumando %d y %d:\n",i,j);

            __asm__(" xorw %0, %0     \n\t" \
                    " xorw %1, %1     \n\t" \
                    " movw %2, %1     \n\t" \
                    " addw %3, %1     \n\t" \
                    " adcw %0, %0     "
                    : "=r" (acarreo) , "=r" (sum)
                    : "rm" (i) , "rm" (j) );

            printf("\nSuma   : %d\n", sum);
            printf("Acarreo: %d\n", acarreo);

            exit(0);
        }
```

Introduce este codigo en un fichero ('aca.c') y compilalo:

```
        demeter# gcc -o aca aca.c

        demeter# ./aca
        Sumando 40000 y 50000

        Suma   : 24464
        Acarreo: 1

        demeter#
```

Una vez mas es un ejemplo tonto, pero que implementa una funcionalidad que a C le resulta dificil imitar. Observa que los comandos en ensamblador operan sobre %0, %1, %2 y %3. Esto es natural, dado que no sabemos como va a llamar el compilador a los argumentos de las instrucciones en ensamblador: la solucion es darles los nombres simbolicos %0, %1, %2 y %3. La informacion que se incluye al final se encargara de decirle al ensamblador como sustituir esos simbolos.

Veamos brevemente como compila esto gcc. La zona interesante es la siguiente (haz un 'gcc -S'):

```
        /APP
             xorw %dx, %dx
             xorw %ax, %ax
             movw -2(%ebp), %ax
             addw -4(%ebp), %ax
             adcw %dx, %dx
        /NO_APP
```

De esto podemos deducir que el compilador ha hecho las sustituciones:

```
             %0  ==  %dx
             %1  ==  %ax
             %2  ==  -2(%ebp)  ( la variable 'i' )
             %3  ==  -4(%ebp)  ( la variable 'j' )
```

Siguiendo el codigo en ensamblador un poco mas, podemos deducir que %ax va a parar a la variable 'sum' y que %dx va a parar a 'acarreo'.

?Como se corresponde esto con las sugerencias que hemos hecho en el __asm__? Veamos la sintaxis y el significado de las mismas. Tenemos:

```
            : "=r" (acarreo) , "=r" (sum)
            : "rm" (i) , "rm" (j)
```

a) La primera linea corresponde a los argumentos de salida del injerto, en tanto que la segunda linea corresponde a las entradas. Cada linea va precedida siempre por un ':'. Si no hubiera salidas, aun asi habria que poner el ':', para que el compilador no se confunda.

b) Los argumentos dentro de cada linea (sean entradas o salidas) son separados por comas, en caso de haber varios.

c) Cada argumento consta de dos partes:

c1) La primera, entre comillas, indica como se escribe el argumento dentro del injerto en ensamblador; es decir, cuando se sustituyen %0, %1, etc., el compilador debe saber si cada uno de esos simbolos tiene que sustituirse por un registro, por una referencia a memoria, por un registro de la FPU, por una constante, etc.

c2) La segunda, entre parentesis, indica a que variable del programa en C se corresponde esa entrada o salida.

Ejemplos:

"rm" (i) --- esto indica que la variable 'i' debe meterse en el injerto como una referencia a memoria o como un registro, que es lo significa la cadena "rm".

"=r" (sum) --- esto indica que la variable 'sum' debe meterse en el injerto como un registro. Por convenio, toda salida lleva siempre un signo '=' metido en la primera parte. Asi, 'registro y salida' se escribe "=r".

Mas adelante veremos las posibilidades mas comunes para la parte c1).

d) El compilador asocia los simbolos %0, %1, %2, %3... a los argumentos de entrada y salida por orden de aparicion. Asi, tenemos que:

Simbolo:	--> Corresponde a:	-> En el injerto:	--> En el .s:
%0	"=r" (acarreo)	sera un registro	%dx
%1	"=r" (sum)	sera un registro	%ax
%2	"rm" (i)	registro o memoria	-2(%ebp)
%3	"rm" (j)	registro o memoria	-4(%ebp)

dado que es en este orden en el que hemos introducido los datos en las lineas de entrada y salida.

Como puedes ver, el compilador tiene a veces opciones para elegir, como en el caso "rm": es gcc quien decide si el simbolo %2 debe ser rellenado con un registro o con una referencia a memoria, segun convenga. En nuestro ejemplo se ha inclinado por una referencia a memoria. Es aconsejable dejar esta flexibilidad al compilador, puesto que los registros no son muy abundantes

en los ix86.

A veces es necesario restringir estas opciones. Por ejemplo, en las entradas solo permitimos referencias a registros, dado que la instruccion 'xor %0, %0', por ejemplo, no podria aceptar que se sustituyese %0 por una referencia a memoria. Este tipo de instrucciones con argumento repetido solo funcionan con registros (por ejemplo, 'xor (%ebp), (%ebp)' es ilegal para los microprocesadores ix86).

De acuerdo. La primera vez que uno lee esto resulta muy complicado. Mi sugerencia es que recuerdes que en realidad le estamos comunicando al compilador una informacion muy sencilla: entradas y salidas. Ponte en el lugar del compilador e imagina que es lo que necesitas saber para cada entrada o salida:

1) A que variable de C corresponde esa entrada o salida.
2) Que simbolo asocio a esa entrada o salida.
3) Como sustituyo ese simbolo en el codigo ensamblador.

Nada mas. Ahora hay que hacerse a la curiosa notacion que han escogido los desarrolladores de gcc para especificar esta informacion. Pero esto no es mas que la parte burocratica; echale un poco de tiempo y basta.

Antes de entrar en una descripcion mas detallada del formato, veamos un ultimo ejemplo. En la primera parte de este tutorial hablamos de la manera de introducir instrucciones no soportadas por el ensamblador, tales como 'rdtsc'. Esta instruccion, presente en la mayor parte de los micros de la serie i686, carga en el par de registros %edx:%eax la cantidad de ciclos de reloj transcurridos desde el arranque del microprocesador (modulo 2 elevado a 64). Esto quiere decir que, por ejemplo, el contador de rdtsc (llamado TSC, 'time stamp counter') se incrementa en 700 millones de unidades cada segundo en mi K7-700. La instruccion rdtsc permite hacer mediciones de tiempo muy precisas, o calcular el numero de ciclos de reloj transcurridos, aproximadamente, entre dos eventos.

Para implementar 'rdtsc' en ensamblador inline, tengamos en cuenta que esta instruccion no tiene argumentos de entrada, no corrompe ningun registro, y como registros de salida tiene especificamente a %edx y %eax. Si miramos la documentacion de 'gcc' (www.gnu.org), en la seccion de 'Extensiones al lenguaje C', subseccion de 'Restricciones para maquinas concretas' (en ingles), podemos encontrar que la manera de indicar al gcc que una entrada o salida corresponde al par %edx:%eax es usando "A". Por tanto, el comando ensamblador sera:

```
                /* rdtsc */       /* salidas */
     __asm__(".byte 0x0f, 0x31" : "=A" (lectura));
```

Donde 'lectura' sera la variable que nos interese. Veamos un ejemplo:

```
        #include <stdio.h>
        #include <math.h>

        main()
        {
            unsigned long long lectura1, lectura2;
            double x,y;
```

```
            __asm__(".byte 0x0f, 0x31" : "=A" (lectura1));

            for (x = 0.0 ; x < 10.0 ; x += 0.01)
                y = exp(x);

            __asm__(".byte 0x0f, 0x31" : "=A" (lectura2));

            lectura2 = (lectdura2 - lectura1) / 1000;

            printf("Numero de ciclos: %d\n", lectura2);

            exit(0);
         }
```

Tras guardar esto en el fichero 'rdtsc.c', lo compilamos y ejecutamos:

```
        demeter# gcc -lm -o rdtsc rdtsc.c

        demeter# ./rdtsc
        Numero de ciclos: 230

        demeter#
```

Es decir, que al micro le lleva en torno a 230 ciclos el ejecutar cada exponencial en el bucle, lo cual es razonable teniendo en cuenta que cada llamada a 'exp' es canalizada por la libreria matematica.

NOTA : Como curiosidad, he compilado y ejecutado este programa en linux, FreeBSD y Windows 98. En los dos primeros los tiempos de ejecucion varian entre 210 y 240 ciclos, en tanto que en Win98 el tiempo ronda los 310 ciclos. La libreria matematica de Win98 no es todo lo eficiente que se podria desear.

Veamos que sucede si calculamos la exponencial directamente en ensamblador. Si sustituimos la linea 'y = exp(x)' por un equivalente en ensamblador, obtenemos:

```
        #include <stdio.h>

        main()
        {
             unsigned long long lectura1, lectura2;
             double x,y;

             __asm__(".byte 0x0f, 0x31" : "=A" (lectura1));

             for (x = 0.0 ; x < 10.0 ; x += 0.01)
             {
                __asm__(" fldl2e        \n\t" \
                        " fmulp         \n\t" \
                        " fld %0        \n\t" \
                        " fld %0        \n\t" \
                        " frndint       \n\t" \
                        " fsubrp        \n\t" \
                        " f2xm1         \n\t" \
                        " fld1          \n\t" \
                        " faddp         \n\t" \
                        " fscale        \n\t" \
                        " fxch          \n\t" \
```

```
                        " fstp %0       \n\t"
                        : "=t" (y)
                        : "0" (x) );
                }

                __asm__(".byte 0x0f, 0x31" : "=A" (lectura2));

                lectura2 = (lectura2 - lectura1) / 1000;

                printf("Numero de ciclos: %d\n", lectura2);

                exit(0);
        }
```

Guardando el resultado en 'rdtsc2.c' y compilando, obtenemos:

```
        demeter# ./rdtsc2
        Numero de ciclos: 119
```

Lo que ciertamente es una mejora. Como es natural, la libreria matematica da ciertas seguridades (comprobacion de errores sobre todo) que no se deben despreciar a la ligera. Pero si tus calculos no son de caracter cientifico, siempre te puedes apoyar en el ensamblador para acelerar enormemente tus programas de calculo en coma flotante.

Si observas el ejemplo de arriba, lo que tenemos es en la practica un hibrido entre ensamblador y C. El C se encarga de gestionar variables y estructuras reiterativas mientras nosotros implementamos lo interesante en ensamblador. Vaaale, quizas este tipo de programas no son muy portables, pero a fin de cuentas, como dice el Fortune File, "Portable == Inutil en cualquier maquina".

---| 3.3 Implementacion del metodo ortodoxo.

Ocupemonos pues de la implementacion del ensamblado 'inline' como Dios manda. El formato general del comando __asm__ es el siguiente:

```
   __asm__ (" ... " : [entradas] : [salidas] : [corrupto] );
```

Donde:

- [entradas] : Se refiere a las entradas en el codigo ensamblador.
- [salidas] : Se refiere a las salidas del codigo ensamblador.
- [corrupto] : Se refiere a los datos/registros corrompidos por el codigo ensamblador. En 'jargon', corrupto se traduce como 'clobbered' (vapuleado).

Cada uno de estos tres campos puede constar de cero o mas partes, cada una de ellas denotando una correspondencia entre un simbolo en el codigo ensamblado y un dato en el programa en C. En caso de haber varias partes en un campo, se separan mediante comas.

Cada parte en los dos primeros campos (entradas y salidas) tiene la forma general:

"fmt" (var)

donde:

(var) : indica a que corresponde esta parte en el programa en C. No tiene por que ser una variable. Tambien puede ser, por ejemplo, una constante. Lo importante es que la expresion entre parentesis se refiere a la expresion de C que queremos meter en el codigo en ensamblador.

"fmt" : indica como se mete en el ensamblador la expresion de C que hemos indicado con (var). Hay muchas maneras de meter simbolos: como constantes, como referencias absolutas de memoria, como referencias de memoria indexadas, como un registro arbitrario, como un registro concreto, como un par de registros, como un registro de la FPU, y algunas mas.

La forma del tercer campo (corrupto) es una lista de nombres de registros (entre comillas y sin el signo '%') que son corrompidos como efecto colateral de nuestro codigo. Veremos ejemplos de corrupcion de registros en la seccion 3.3.5.

Asi pues, la forma mas general del comando __asm__ es:

```
__asm__("..." ["..." ... ]
        [ : ["fmt" (var) [, ...] ] ]
        [ : ["fmt" (var) [, ...] ] ]
        [ : ["reg" [, ...] ] ] );
```

Observa la logica y la simplicidad de la notacion. Como ya hemos indicado antes, hay que decirle a 'gcc' que pasa con las entradas, las salidas y la informacion corrupta. Pues bien, de esto se encarga cada uno de los tres campos. Ahora bien, para un dato de entrada, salida o corrupto, siempre tenemos que indicar al menos dos cosas: que forma tiene ese dato en el ensamblador y que forma tiene ese dato en el C. Una vez que el compilador sabe ambas cosas, no tiene mas que sustituir simbolos. De hacer esa doble identificacion se encargan los pares '"fmt" (var)'.

Bien, ahora sabemos la estructura de los tres campos, pero queda una cuestion por resolver. Dado que el ensamblador debe tener una idea clara de _absolutamente_todo_ dato que usamos en nuestro codigo ensamblador, no tiene sentido usar datos especificos en el codigo ensamblador.

Por ejemplo. Supongamos que queremos meter en la variable 'dato' (tipo long) la cantidad decimal 7. Si sabemos que el codigo almacena 'dato' en -4(%ebp), podriamos hacer:

```
__asm__("  movl $7, -4(%ebp)" : "=m" (dato) );
```

y considerar que esto esta bien hecho. A fin de cuentas, hemos indicado al compilador que nuestro codigo tiene como dato de salida una referencia de memoria indexada (de ahi el "=m"), que va a caer en la variable 'dato'.

Pero esto es una metedura de pata. Lo que necesitamos es que el compilador _llene_ nuestro codigo con la posicion correcta de 'dato', no con la que nosotros le impongamos.

Esto implica que el codigo ensamblador debe ir escrito en forma de 'plantilla', de modo que el compilador lo llene con los datos correctos. Para esto se emplean los simbolos %0, %1, ... hasta %10 (hasta %30 en las versiones avanzadas de gcc).

Por ejemplo:

```
__asm__(" movl $7, %0 " : "=m" (dato) );
```

es correcto, y le dice al compilador que coja la variable 'dato', la incruste en el codigo ensamblado en forma de acceso indexado a memoria en el lugar donde esta '%0', y que tenga en cuenta que 'dato' es variable de salida para este codigo.

Resumiendo: el formato __asm__ bien hecho esta compuesto de:

1) Tres campos en el formato arriba indicado, que indican los datos de entrada, salida y corruptos.

2) Un codigo ensamblador en forma de plantilla, que va entre comillas al principio del comando __asm__. Los 'huecos' de la plantilla se indican %0, %1, etc.

Una pregunta importante es como reconoce 'gcc' a que dato corresponden %0, %1, %2, etc. La cosa es facil: gcc va nombrando los datos por orden segun los va leyendo. Por ejemplo, en el codigo (que ya vimos en la seccion anterior):

```
__asm__(" xorw %0, %0     \n\t" \
        " xorw %1, %1     \n\t" \
        " movw %2, %1     \n\t" \
        " addw %3, %1     \n\t" \
        " adcw %0, %0     \n\t"
        : "=r" (acarreo) , "=r" (sum)
        : "rm" (i) , "rm" (j) );
```

tenemos 4 datos entre entradas y salidas, es decir, que habra que definir simbolos desde %0 hasta %3. Entonces, tendremos que %0 se corresponde con "=r" (acarreo), %1 con "=r" (sum), %2 con "rm" (i) y %3 con "rm" (j). Se leen de izquierda a derecha, segun van apareciendo.

Si se entiende lo que hemos visto hasta ahora en esta seccion, se tiene buena parte del camino andado. La idea general, resumida una vez mas:

1) Escribe el ensamblador usando simbolos, de manera que sea el compilador el que los rellene.

2) Una vez que tienes tu ensamblador escrito con simbolos,

escribe las lineas de entradas, salidas y corruptos
de manera que esos simbolos sean sustituidos adecuada-
mente (por registros, refs. a memoria, etc.) y se
correspondan con los datos en C que te interesan.

Y eso es todo.

Observa el ejemplo que acabamos de ver arriba. Para construirlo, en primer lugar compuse el codigo en ensamblador usando simbolos. Me quedo:

```
xorw %0, %0
xorw %1, %1
movw %2, %1
addw %3, %1
adcw %0, %0
```

Ahora, yo sabia que una instruccion tipo 'xor <algo>, <algo>' solo puede ser sustituida correctamente si <algo> es un registro. De manera que introduje en la primera linea de las salidas "=r", que obliga a 'gcc' a sustituir %0 por un registro de su eleccion. Ademas, como %0 debia ser al final del ensamblador el valor del acarreo, le pedi a gcc que almacenase %0 en la variable 'acarreo', lo cual se logra con poner (acarreo) despues del "=r". Con esto ya podia estar seguro de que no tenia que preocuparme mas por %0. Con %1, %2 y %3 procedi exactamente igual. El resultado final fue el codigo completo, tal y como fue empleado en la seccion anterior.

3.3.1 Simbolos solapados.

Aunque el formato expuesto hasta ahora es conceptualmente muy sencillo, cuando uno echa las cosas a correr siempre aparecen algunas complicaciones muy naturales que a primera vista no se toman en cuenta.

Piensa en el siguiente ejemplo: queremos hacer una diminuta linea de ensamblador que calcule el 'not' (complemento a 1) de una palabra. En principio podriamos hacer la construccion:

```
__asm__("notl %0" : "=r" (var1) : "r" (var2));
```

Pero el problema que aparece es el siguiente: como el registro de entrada es el mismo que el de salida, el compilador se va a hacer un lio. Tal y como hemos escrito el codigo, 'gcc' va a definir los simbolos %0 y %1, y se mostrara muy confundido cuando vea que el codigo solo emplea %0.

Una solucion aceptable seria hacer lo siguiente:

```
__asm__("movl %0, %1 ; notl %1" : "=r" (var1) : "r" (var2));
```

Esto es perfectamente correcto, pero no siempre es aplicable (por ejemplo, al usar la FPU no se tiene tanta facilidad para echar mano de registros adicionales).

--> El problema es por tanto: como decirle a 'gcc' que un simbolo
--> de salida coincide con un simbolo de entrada.

Naturalmente, esto tiene solucion; de hecho, ya hemos visto algun ejemplo en las secciones anteriores. La solucion es simplemente

indicar en el argumento repetido el numero de la etiqueta que se repite (sin el '%'). Por ejemplo:

```
__asm__("notl %0" : "=r" (var1) : "0" (var2));
```

Con esto el compilador se da cuenta de que '%0' es compartido por la entrada y la salida, dado que en la entrada aparece "0" como formato, haciendo referencia a %0. Veamos algunos ejemplos que empleamos anteriormente:

```
__asm__("fsincos" : "=t" (x), "=u" (y) : "0" (t));
```

En este caso, los formatos "t" y "u" indican respectivamente la primera y la segunda posiciones en la pila de la FPU. El fragmento en ensamblador espera la entrada de 't' en la primera posicion de la pila y, tras la instruccion 'fsincos', se pone Sen(t) en la segunda y Cos(t) en la primera posiciones de la FPU. De manera que la variable de salida 'x' y la de entrada 't' comparten la primera posicion en la pila de la FPU. Por este motivo, se usa el "0" para indicar el solapamiento entre la entrada y la salida. (Este tipo de complicaciones es tipico de las operaciones con la FPU; como ejemplo es un tanto enrevesado).

Otro ejemplo de la misma naturaleza es la verion en ensamblador de 'y = exp(x)':

```
__asm__(" fldl2e        \n\t" \
        " fmulp         \n\t" \
        " fld %0        \n\t" \
        " fld %0        \n\t" \
        " frndint       \n\t" \
        " fsubrp        \n\t" \
        " f2xm1         \n\t" \
        " fld1          \n\t" \
        " faddp         \n\t" \
        " fscale        \n\t" \
        " fxch          \n\t" \
        " fstp %0       \n\t"
        : "=t" (y)
        : "0" (x) );
```

Una vez mas, la entrada y la salida comparten el registro %st(0) de la FPU, de manera que se usa el solapamiento.

Veamos un ultimo ejemplo, tonto pero instructivo: un fragmento de ensamblador que calcula el complemento a 1 de una variable y el complemento a 2 de otra:

```
__asm__(" notl %0    \n\t" \
        " negl %1        "
        : "=r" (var1) , "=r" (var2)
        : "0"  (var3) , "1"  (var4) );
```

En este caso tenemos dos solapamientos, el de %0 y el de %1.

Como puedes ver, esta situacion ha sido cubierta con facilidad y elegancia por 'gcc'. Ningun problema por esta parte.

3.3.2 Formatos mas comunes.

Hasta ahora hemos recurrido a los ejemplos para aprender los formatos "=" (salida), "r" (registro), "m" (memoria indexada), "f" (registro de la FPU), "t" (primer registro de FPU), "u" (segundo registro de FPU), y los de solapamiento "0", "1", etc.

Tambien hemos observado (en 3.2) que se pueden combinar varias sugerencias en la misma cadena, como "rm" (registro o memoria indexada). Pueden hacerse muchas otras combinaciones, siempre que tengan sentido, como "=rm" o "mf", etc.

En la documentacion en ingles nuestros 'formatos' se traducen como 'constraints' (restricciones). Para informarse sobre todos los tipos importantes, incluyendo algunos a medio documentar (como los registros SSE), consulta la documentacion de referencia del 'gcc'.

FORMATOS GENERICOS DE LA CPU:

"r" : cualquier registro de la CPU (%eax, %ebx, etc.)

"m" : una referencia a memoria (p. ej. -8(%eax, %ebx, 2)).

"q" : un registro que que puede operar con bytes.
Es decir, %eax, %ebx, %ecx o %edx.

"A" : el par de 64 bits %edx:%eax. Tambien vale para indicar uno cualquiera de los dos, %edx o %eax.

REGISTROS ESPECIFICOS DE LA CPU:

"a" : %eax, %ax, %ah, %al.

"b" : %ebx, %bx, %bh, %bl.

"c" : %ecx, %cx, %ch, %cl.

"d" : %edx, %dx, %dh, %dl.

"D" : %edi, %di.

"S" : %esi, %si.

CONSTANTES ENTERAS:

"i" : una cantidad inmediata de 32 bits.

"n" : una cantidad inmediata conocida en tiempo de compilacion.

"I" : una constante entera entre 0 y 31. Se usa para indicar desplazamientos (shl, ror, etc.)

"J" : una constante entera entre 0 y 64. Se usa para desplazamientos de 64 bits.

"K" : equivale a 0xff.

"L" : equivale a 0xffff.

"M" : 0, 1, 2 o 3. Util para el factor de escala en 'lea', o en cualquier direccionamiento indexado.

"N" : constante entera entre 0 y 255.

REGISTROS DE LA FPU:

"f" : cualquier registro de la FPU.

"t" : el TOS ('top of stack') de la FPU: %st(0).

"u" : %st(1).

"G" : una constante en doble precision estandar para el 387.

"F" : una constante en doble precision.

COSAS CURIOSAS:

"x" : un registro MMX.

"y" : un registro SSE.

Gracias a estos formatos podemos especificar mediante simbolos practicamente cualquier cosa. En la siguiente seccion veremos ejemplos que se corresponden con los mas interesantes de estos formatos, pero antes veamos otro tipo de formatos, de los que forma parte el "=" que vemos tan a menudo en los argumentos de salida:

MODIFICADORES:

"=" : cuando se indica con un formato, se da a entender que el simbolo es escrito pero no leido por el codigo en ensamblador. Es comun usar "=" como formato para cualquier salida, sobre todo en versiones mas viejas de gcc.

"+" : esto indica que el argumento es leido y escrito por el codigo. Es mas restrictivo que "=", dado que implica que 'gcc' no puede utilizar este simbolo como almacenamiento intermedio antes de ser escrito de manera definitiva. Me parece que el compilador se porta de manera caprichosa con esta opcion; es casi mejor pasar de ella. Aun asi, en ocasiones funciona.

Existen otros modificadores, pero corresponden a casos de mucha mas sutileza, y no son criticos. Observa la documentacion para mas detalles.

3.3.3 Ejemplos de formatos de la CPU.

Ya hemos visto ejemplos del uso de "r", "m", "A", "=", "f", "t" y "u". Tambien de algunas combinaciones, como "=r" y "rm". Acerca de la FPU hablaremos con mas detalle en 3.3.4, asi que por ahora nos concentraremos en la CPU.

En primer lugar, veamos que pasa con las operaciones a nivel de byte. Si queremos usar registros que funcionen con bytes es importante que nos limitemos a "q", "a", "b", "c" o "d". Por ejemplo:

```
__asm__("negb %0" : "=q" (var1) : "0" (var2));

__asm__("negb %0" : "=a" (var1) : "0" (var2));
```

curiosamente, he intentado compilar infringiendo la regla, con:

```
__asm__("negb %0" : "=S" (var1) : "0" (var2));
```

y el compilador, sabiamente, ha cambiado %esi por %al de manera automatica. Aun asi es mejor no tentar a la suerte.

Veamos un uso de constantes raras. Mete esto en un codigo en C:

```
#define FACT 2

main()
{
    long var1, var2;

    __asm__(" shll %1, %0 "
            : "=a" (var1)
            :  "I" (FACT), "0" (var2) );
}
```

Si lo compilas con '-S' y te fijas en la zona injertada, tenemos:

```
/APP
        shll $2, %eax
/NO_APP
```

que es justo lo que le pedimos. Esto es un buen ejemplo del uso de un formato raro como "I". De paso te permite ver como se pueden meter definiciones del preprocesador en el ensamblado inline.

Vamos a echar una mirada a las constantes inmediatas. Prueba a compilar con 'gcc -S' el siguiente codigo:

```
#define BASE 23144

main()
{

    long var1;

    __asm__("movl %1, %0" : "=m" (var1) : "i" (BASE + 69));

}
```

compliando y editando el ensamblador obtenemos:

```
        /APP
                movl $23213, -21(%ebp)
        /NO_APP
```

Esto da un ejemplo de cantidad inmediata de 32 bits, y de operaciones dentro de los valores introducidos en el ensamblador, mezclado con preprocesamiento. Puedes desarrollar ejemplos mucho mas complejos.

Llegados a este punto, supongo que te haces una idea de como se manipula el resto de los formatos. Sencillamente, siempre hay uno (o quizas varios) formatos adecuados para una cierta situacion. Uno debe ser lo menos restrictivo posible con el tipo de formato, para no contrarrestar la capacidad de optimacion del 'gcc'.

3.3.4 Ejemplos de formatos de la FPU.

La FPU de los x86 es bastante curiosa. Si llevas algun tiempo en el mundo de la informatica, es posible que recuerdes FORTH, un genial lenguaje interpretado que empleaba varias pilas en sus accesos a la memoria, en vez de accesos aleatorios a traves de variables. La FPU de los x86 fue concebida de la misma manera, lo que la hace francamente entretenida de programar. Quizas un fallo de los lenguajes tipo FORTH es que son poco concurrentes, dado que es muy dificil paralelizar instrucciones que acceden a una pila.

Lo que nos ocupa ahora es que la estructura de pila de la FPU hace muy dificil el andar jugando con registros especificos de la FPU. Mientras que con la CPU podiamos especificar alegremente que registro de proposito general queriamos usar, esto no es factible con la FPU (salvando %st(0) y %st(1) que, como vimos, se pueden indicar con los formatos "t" y "u"). Sin embargo, en muchas situaciones es suficiente con poder usar %st(0) y %st(1), si se programa con cuidado.

Ensamblar inline con la FPU se parece un poco a lo que haciamos en la Parte 2 con las subrutinas. En los argumentos de entrada le decimos al 'gcc' que datos queremos que nos meta en la pila, mientras que en las salidas le decimos que datos hemos dejado en la pila para que los recoja.

Un ejemplo lo tenemos en el caso que ya hemos visto:

```
        __asm__("fsincos" : "=t" (x), "=u" (y) : "0" (t));
```

El dato de entrada, "0" (t), le dice al compilador que nuestro codigo en ensamblador asume que en %st(0) se encuentra la variable 't'. Entonces, 'fsincos' toma de la pila este valor y devuelve Cos(t) en %st(0) y Sen(t) en %st(1). Los argumentos de salida le dicen al compilador que puede recoger de esas dos posiciones de la pila los valores de 'x' e 'y'.

El otro caso que hemos visto, el de la exponencial, es tambien interesante:

```
            __asm__(" fldl2e        \n\t" \
                    " fmulp         \n\t" \
                    " fld %0        \n\t" \
```

```
                " fld %0        \n\t" \
                " frndint       \n\t" \
                " fsubrp        \n\t" \
                " f2xm1         \n\t" \
                " fld1          \n\t" \
                " faddp         \n\t" \
                " fscale        \n\t" \
                " fxch          \n\t" \
                " fstp %0       \n\t"
                : "=t" (y)
                : "0" (x) );
```

Una vez mas, el dato de entrada es 'pusheado' por el compilador, que recoge de la pila el dato de salida una vez que hemos terminado. Lo interesante de este ejemplo es que durante su ejecucion se introducen y se sacan muchos otros datos de la pila (como el logaritmo de 'e' en base 2) pero, debido a que todos los datos adicionales que se introducen son sacados antes del final del injerto, el 'gcc' no tiene por que preocuparse de eso.

Esto es un principio fundamental. Si te ocupas de mantener la coherencia de la pila, y te limitas a decirle al 'gcc' que meta o saque los datos que necesitas, escribir codigo para la FPU es algo realmente facil.

De todos modos, hay sutilezas que debes tener en cuenta para que el compilador se entere de que manipulaciones has operado en la pila. Si estas interesado en escribir ejemplos complicados de ensamblador inline con manipulaciones en la pila un tanto exoticas (por ejemplo, cuando hay mas valores de entrada que de salida), debes consultar la documentacion de 'gcc' (parte 'Assembler Instructions with C Expression Operands') que ofrece una lista de pasos a seguir para manejar esos casos curiosos. El problema esta en que el compilador puede tener problemas para decidir si los datos que se introdujeron inicialmente en la pila fueron expulsados de ella o no por tu codigo en ensamblador. Ese tipo de circunstancias es facil de decidir, aunque introducir las reglas en un tutorial como este podria ser un tanto molesto para los que no pretenden dedicarse especificamente a programar la FPU.

Mi consejo es que uses ensambles inline solo instrucciones de la FPU relativamente cortas pero exoticas, como las que hemos visto, en vez de escribir partes largas que te llevaran a complicaciones innecesarias (aunque no insalvables, en absoluto).

3.3.5 Registros corruptos.

Hemos estado obviando este campo de la instruccion __asm__ hasta ahora, no porque sea complicado (no lo es en absoluto), si no porque no es muy usual encontrarse con este tipo de casos.

Veamos sin embargo un primer ejemplo: supongamos que queremos hacer la instruccion 'rdtsc', pero almacenando solo los 32 bits menos significativos del par %edx:%eax, en tanto que nos importa poco que sucede con los bits mas significativos. En otras palabras, nos quedaremos con el contenido de %eax, despreciando lo que haya en %edx (esta situacion puede aparecerte generando numeros aleatorios mediante temporizadores de mucha precision).

En este caso, tenemos que advertir a 'gcc' que el registro %edx, aunque no forma parte de nuestras entradas ni salidas, ha sido alterado. Esto se indicaria como:

```
__asm__(".byte 0x0f, 0x31"
        : "=a" (tiempo)
        :
        : "edx" );
```

El compilador asume facilmente la situacion.

Observa que los registros corruptos van, como ya indicamos en 3.3, con su nombre completo y sin el '%' de prefijo.

Vamos a ver un caso mucho mas divertido. Ejecutemos la instruccion 'cpuid' y extraigamos la familia, modelo y 'stepping' del micro que estamos empleando. Para ello hay que ejecutar las instrucciones en ensamblador:

```
movl $1, %eax
cpuid
```

Despues de estos comandos se tiene que %eax almacena los datos que queremos, en tanto que %ebx y %ecx quedan en estado indefinido, y %edx contiene informacion sobre las facilidades que ofrece el microprocesador. Aqui tenemos casos sobrados de corrupcion. Vamos a hacer un programilla que nos diga algo sobre el micro. Guarda el siguiente codigo en un fichero ('cpuid.c'):

```
#include <stdio.h>

main()
{

  unsigned long id, carac;

  /* Esto hace el cpuid */

  __asm__(" movl %2, %0        \n\t" \
          " .byte 0x0f, 0xa2   "
          : "=a" (id) , "=d" (carac)
          : "i" (1)
          : "ebx" , "ecx" );

  /* Ahora escribimos algo de informacion. */

  printf("Informacion del microprocesador:\n");
  printf("    Familia : %d\n", ((id & 0xf00)>>8));
  printf("    Modelo  : %d\n", ((id & 0xf0)>>4));
  printf("    Stepping: %d\n\n", (id &0xf));

  /* Ahora deducimos alguna facilidad del micro. */

  if (carac & (1<<23))
      printf("El micro soporta MMX\n");
  else
      printf("El micro no soporta MMX\n");

  if (carac & (1<<15))
      printf("El micro soporta CMOV\n");
  else
      printf("El micro no soporta CMOV\n");

  if (carac & (1<<5))
```

```
            printf("El micro soporta MSRs\n");
          else
            printf("El micro no soporta MSRs\n");

          if (carac & (1<<25))
            printf("El micro soporta XMM\n");
          else
            printf("El micro no soporta XMM\n");

          exit(0);

        }
```

Compilamos y ejecutamos:

```
        demeter# gcc -o cpuid cpuid.c

        demeter# ./cpuid
        Informacion del microprocesador:
            Familia : 6
            Modelo  : 2
            Stepping: 1

        El micro soporta MMX
        El micro soporta CMOV
        El micro soporta MSRs
        El micro no soporta XMM

        demeter#
```

Lo cual es correcto. Mi micro es uno de los primeros AMD K7, de ahi que tenga un Modelo y Stepping tan bajos. Naturalmente, soporta MMX, CMOV y MSRs (Model Specific Registers). Pero, al ser un AMD, no soporta la basurilla multimedia de Intel; tiene su propia basurilla multimedia (3DNow! y todo eso). Desarrollando un poco este codigo se podria lograr un buen identificador de micros.

---| 3.4 Que se nos queda en el tintero.

Intentar cubrir todas las opciones del ensamblado inline es un proyecto enormemente ambicioso, que no se podria conseguir sin complicar mucho este tutorial. Creo que con lo que hemos visto hasta ahora tienes, como minimo, para cubrir todos los casos importantes con los que te puedes topar en la practica.

Al menos, espero haber podido dar en esta parte los fundamentos para entender otros textos que circulan en la red (todos los que he visto estaban en ingles salvo uno, en italiano) sobre ensamblado inline. Es sorprendente lo facil que es escribir ensamblador inline una vez que uno se familiariza un minimo con la materia.

Como ya he indicado, nos falta por detallar algunas opciones especiales en los formatos de entrada y salida (cosas como "%", "&", "#", etc.) Tambien nos hemos dejado en el tintero el mecanismo para tratar el ensamblado inline de la FPU en sus formas mas complicadas, pero eso es algo que puedes consultar en la documentacion 'gnu' una vez que tienes una experiencia minima.

Seguro que quedan otros muchos agujeros por tapar, algunos de ellos imperdonables. Si encuentras alguno, y crees que debe taparse, enviame un mail. Si crees que puedes hacerlo tu mismo, adelante, este documento es _tu_ documento. Disfrutalo.

----------| Referencias |----------

He aqui lo que puedes consultar, antes, despues o durante la lectura de este tutorial. La bibliografia no es completa, pero puede ayudarte a conseguir una buena cultura elemental en el tema del ensamblador AT&T con 'gcc'. Me he tomado la libertad de comentar algunas de las referencias.

---| Jon Beltran de Heredia, "Lenguaje Ensamblador de los 80x86"
Anaya Multimedia, ISBN 84-7614-622-1

Desde mi punto de vista, la mejor introduccion al ensamblador x86 en nuestra lengua. Aunque solo cubre el ensamblado en 16 bits en formato Intel (para el anticuado 'masm'), es insustituible si quieres aprender los fundamentos del ensamblador en estas maquinas.

---| Intel Corp., "Intel Architecture Software Developer's Manual.
Volume 2: Instruction Set Reference".
Order number 243191.

El 'white paper' sobre la arquitectura x86 por excelencia. Los volumenes 1 y 3 (Order # 243190 y 243192 respectivamente) son tambien muy interesantes. Si quieres conseguirtelos en .pdf vete a la web de Intel y busca el documento. Te recomiendo que lo hagas por el 'Order number', que es la manera mas rapida de dar con el. La pagina web de AMD tambien tiene 'white papers' muy interesantes.

---| GNU, "Using the GNU Compiler Collection"

Este documento cubre todos los detalles relevantes del compilador 'gcc', y puedes descargartelo desde www.gnu.org en varios formatos. Mucha de la informacion que hemos omitido por ser latoso o complicado incluirla aqui puedes encontrarla en este documento.

---| GNU, "The GNU Assembler"

Otro documento de www.gnu.org. Este documento es insustituible para enterarse de las directivas y principales convenios del funcionamiento del compilador 'gas'. Siempre es bueno tenerlo a mano.

---| Colin Plumb, "A Brief Tutorial on GCC inline asm (x86 biased)"
20 April 1998.

Una de esas joyas con las que te encuentras en la red. Escrito por el hacker Colin Plumb, es una fantastica introduccion al ensamblado inline, publicado originalmente en una lista de correo y actualmente disponible en varias URLs (una busqueda en google deberia localizartelo). Yo encontre una copia en www.opennet.ru,

en una coleccion de documentos sobre ensamblador. Absolutamente imprescindible.

---| Aparte de estos documentos puedes encontrar muchas otras referencias, buena parte de ellas obras amateur con mas buena intencion que rigor. De todos modos, un google con 'inline assembler' arroja muchos resultados, algunos de ellos de primera linea.

EOF

Fonte: http://mgu7mxp3vxzfj7hs.onion/