Gcc Inline Assembly - How to

Martin Candurra (astor) disponible en hackemate.com.ar
martincad@yahoo.com
Lunes, 20 de Octubre de 2003, a las 14:46:00 ART

Con este humilde artículo espero facilitar la comprensión del *Inline Assembly* en Gcc. Si bien existen varios documentos escritos sobre el tema (incluido el manual oficial de Gcc) espero lograr un enfoque diferente con este texto. Mi parte está cumplida si alguien, leyendo este artículo, logra mezclar lenguaje C y ensamblador con facilidad.

Contents

1 Introducción

1.1 Que es y para que sirve el inline assembly ?

Existen muchos casos donde debemos utilizar en nuestro código (en lenguaje C) un poco de assembly debido a que queremos optimizar algunas lineas un poco más "a mano", o sencillamente necesitamos usar instrucciones las cuales no poseen ningún tipo de función o macro asociada en C(por ejemplo LGDT, LTR, IRET, etc).Es en estos casos en los cuales es necesario recurrir al *inline assembly* (o ensamblado en linea). Esta herramienta que nos brinda el Gcc no sólo es muy útil, si no que después de acostumbrarse a su utilización termina siendo realmente cómoda

1.2 Por qué otro texto de Inline Assembly ?

Por la simple razón que cuando necesite aprender inline assembly considero me costo más de lo que hubiera querido. No encontre la cantidad de ejemplos que me hubiera gustado, y sentí que esos textos estaban a otro nivel.

Desde ya que con este humilde texto no pretendo brindar más inforción de la queda el manual de Gcc, si no que intentaré ser lo mas claro posible, utilizando ejemplos sencillos, que cualquier novato entendería. Al igual que a varios conocidos que tengo, me costo bastante entender ese típico ejemplo de la multiplicación de números via *Inline Assembly* que aprovecha la arquitectura super-escalar del microprocesador Pentium. Imaginense, mientras buscaba la palabra *clobber* en el diccionario, intentaba interpretar porque ese "leal" con todos sus agregados tomaba solo 1 ciclo de máquina. (Para los que no sepan de que estoy hablando, no tienen más que ir a otro *how to* de *inline assembly* para entenderme :-). Tengan en cuenta que la información que aqui se brinda es básica es decir, en caso de necesitar algo mas avanzado, deberán recurrir al manual de Stallman.

1.3 Requisitos básicos

Para que este documento puede serle útil usted necesitará

- Contar con el Gnu Compiler Collection (gcc)
- Experiencia programando en Lenguaje C y Assembly
- Conocimientos básicos de sintaxis AT& T

1.4 Aclaración

Tanto en el manual de GCC como en otros documentos van a encontrar una diferenciación entre el *Inline Assembly* y lo denominado *"Extended" Inline Assembly*. El primero es utilizado para insertar código *assembly* sin que este utilice variables o macros del código escrito en C. En cambio, el *"extended"* nos brinda la posibilidad de poner en registros, o utilizar en instrucciones del microprocesador valores que se encuentran en variables, o viceversa. Este texto se concentra en el *"extended inline assembly"* ya que es donde la mayoría de nosotros nos chocamos un poco al comenzar. Vale también aclarar que todos los ejemplos aquí mostrados son aptos sólo para la arquitectura IA32 (386 o superior).

1.5 Copyright y reproducción

2 Inline Assembly

2.1 Ejemplo muy simple

Como coloco una instrucción de "no operation" dentro de mi código ?

```
int main (void)
{
    __asm__ ("nop");
}
```

Procedemos a compilar utilizando el modificador -S, el cual le dice a gcc que compile pero no linkee, dejandonos como resultado un fuente en assembly (sintaxis AT& T). Es decir, ejecutamos `gcc -S ejemplo1.c` con lo que obtendríamos un ejemplo1.s conteniendo algo similar a esto:

```
        .file   "ejemplo1.c"
        .section .text
.globl _main
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
/APP
        nop
/NO_APP
        leave
        ret
        .ident  "GCC: (GNU) 3.2.3"
```

De donde salio todo ese código si lo único que escribí fue un "nop" ? Es aquí donde eran necesarios los conocimientos de *assembly* :-) . Por el momento basta con que sepas que las funciones en C reciben sus argumentos a través del *stack*, y es en ese mismo lugar donde "viven" las variables locales. Si bien nuestro *"main"* no recibe argumentos ni define variables locales, el Gcc arma la estructura como si las usara. Es entre las lineas /APP y /NO_APP que aparece nuestro código.

NOTA: Estas líneas fueron compiladas utilizando el DIGPP El código de salida puede variar ligeramente dependiendo del compilador.

2.2 Otro ejemplo sencillo

Veamos otro ejemplo simple para los que aún se sientan confundidos:

```
int main(void)
{
    __asm__ ("movl $0xdead, %eax");
}
```

Luego de ejecutar `gcc -S ejemplo2.c` obtenemos:

```
        .file   "ejemplo2.c"
        .section .text
.globl _main
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
/APP
        movl $0xdead, %eax
/NO_APP
        leave
        ret
        .ident  "GCC: (GNU) 3.2.3"
```

NOTA: es necesario colocar antes de los datos inmediatos el operando "$" para que el gcc lo intérprete correctamente, mientras que a los registros debe anteponerse un "%". Como verán el *inline assembly* básico no representa gran utilidad, excepto en algunos casos como los que se describen a continuación.

2.3 Ejemplos más útiles

Ahora

```
__asm__ ("sti");
```

El cual habilita las interrupciones enmascarables del microprocesador. Es bastante comun encontrarse con alguna de las siguientes definiciones:

```
#define enable()    __asm__ __volatile__ ("sti")
        o
#define sti()       __asm__ __volatile__ ("sti")
```

las cuales son visualmente mas agradables que su contraparte utilizando inline assembly.

Veamos algún otro ejemplo útil del *inline assembly* común:

```
__asm__ __volatile__ ("pushf ; cli");
        .
        .
    código crítico
        .
        .
__asm__ __volatile__ ("popf");
```

En este caso no hacemos mas que bloquear las interrupciones para un fragmento de código crítico, el cuál no puede ser interrumpido. El uso del PUSHF y POPF en lugar de un simple CLI y STI me aseguran que las interrupciones al finalizar mi código quedarán en el estado que estaban antes de que yo las deshabilite.

3 Extended Inline Assembly

3.1 Introducción

Con lo que vimos anteriormente no logramos ningún tipo de interacción con los operadores definidos por el resto del código. Es esta versión "extendida" la que nos permitirá pasar valores inmediatos a registros, cargar registros con el contenido de variables, cargar variables con el contenido de registros, etc.

La sintaxis genérica es:

```
__asm__ ( "instrucciones" : lista_salida : lista_entrada : lista_destruida);
```

Donde:

lista_salida(o *output list*): contiene los registros, variables donde se guardara un dato.

lista_entrada (o *input list*): contiene los registros, variables o datos inmediatos que se utilizaran como entradas en una instrucción.

lista_destruida (o *clobber list*): contiene los registros que se ven modificados por las "instrucciones". Esta lista es necesaria para que el gcc sepa que registros debe utilizar, cuales debera resguardar, etc.

Si bien esto puede parecer un poco confuso, espero ir evacuando las dudas a través de ejemplos.

3.2 Ejemplo 1

```
int main(void)
{
    int i=100;
    __asm__ ("movl %0, %%eax" : : "g" (i));
}
```

En ese fragmento lo que hacemos no es más que cargar el valor de i en el registro eax. Nótese que al utilizarse la versión "extendida" es necesario agregar otro operador "%" a los registros (Como ya en la versión basica del *Inline Assembly* anteponiamos el operador "%" a los registros, ahora deberemos agregar otro más, con lo que cada registro tendrá un "%%" antes de su nombre.

Volviendo al ejemplo anterior, si observamos, el modificador "g" (i) se encuentra en la lista de entrada, mientras que las lista de salida esta vacía al igual que la clobber list.

Podriamos utilizar otro ejemplo levemente mas avanzado, para quien todavía se encuentra perdido.

3.3 Ejemplo 2

```
#define MAX_VALOR       200
int main(void)
{
    int i=100;
    __asm__ ("movl %0, %%eax; movl %1, %%ebx" : :  "g" (i) , "g" (MAX_VALOR));
}
```

Aca nuevamente no utilizamos ni la lista de salida, ni la lista de registros destruidos. Simplemente cargamos en eax y en ebx el valor de i y 200 respectivamente.

Que función cumple el %0 y el %1 ? Estos son los valores que serán reemplazados dinámicamente por Gcc. En caso de que agregaramos más instrucciones con mas parámetros a cargar, continuarán con %2, %3, etc.

Que representa el "g" ? Este modificador le indica a Gcc que el parametro que se pasa entre paréntesis puede ser tanto una variable en memoria como un valor inmediato. Si bien el modificador "g" es uno de los más utilizados (al menos por mi) hay muchos otros. Sólo intentare explicar algunos, pueden consultar el resto en el manual.

3.4 Modificadores

A continuación se listan los modificadores más usados en el ensamblado extendido. Existen otros más, incluyendo algunos propios de ciertas arquitecturas.

1. "a" eax
2. "b" ebx
3. "c" ecx
4. "d" edx
5. "S" esi
6. "D" edi
7. "q" o "r" cualquier registro de propósito general (a conveniencia de gcc)
8. "I" valor inmediato (entre 0 y 31)

Voy a intentar clarificar un poco esto a través de unos ejemplos. Empecemos por uno sin mucha utilidad práctica, pero que a nuestros fines nos sirve:

3.5 Ejemplo 3

```
int main(void)
{
    int i=100;
    __asm__ ("mov %0, %%ebx" : : "c" (i));
}
```

Aqui el modificador "c" indica al Gcc que el valor de i deberá ser cargado en el registro ECX, el cual será colocado en el lugar de %0. En pocas palabras, en este caso, cargamos el valor de i en EBX, utilizando a ECX como registro de paso.

Pasemos a un ejemplo más real y útil.

3.6 Ejemplo 4

La siguiente es una implementación de la función memcpy utilizando *inline assembly* (Sacada de Routix).

```
void *memcpy( void *dest, const void *src, unsigned int n)
{
    __asm__("cld ; rep ; movsb":  : "c" ((unsigned int) n), "S" (src), "D" (dest));

    return dest;
}
```

Recordemos que la instrucción movsb mueve el contenido de esi a edi, e incrementa o decrementa los valores de ESI y EDI (según el flag de dirección, modificado en este caso por CLD). La instrucción REP hace que se repita esto mientras ECX no sea 0. Tampoco en esta ocasión utilizamos la lista de salida, ni la de registros destruidos. El modificador "c" indica que el registro de destino será ECX, mientras que "S" y "D" lo hacen con ESI y EDI respectivamente.

3.7 Modificador "r"

El modificador "r" como ya comentamos con anterioridad le indica a Gcc que puede utilizar cualquier registro. Esto es útil ya que en muchas oportunidades no es necesario usar un registro en particular, y cualquiera que se utilice puede realizar bien la tarea. Quizá para alguien con no mucha experiencia mezclando código C y assembly le venga bien la siguiente explicación. Como vimos antes, podemos ver el código ensamblador generado por Gcc utilizando la opción -S. Allí pudimos observar que los accesos a variables se realizan utilizando registros de propósito general. Que pasaría si el código en C guardara un valor en EAX para un futuro acceso a una variable y nosotros muy despreocupadamente utilizamos un

```
__asm__ ("movl %0, %%cr3" : : "a" (i) )
```

? Tal cual lo imaginamos, vamos a estar pisando el valor que pensaba utilizar Gcc con lo cual el comportamiento del programa es ahora totalmente impredecible. Es en este (y en otros) casos donde el modificador "r" nos ayuda a evitar esos problemas, ya que deja que Gcc elija un registro que no esta utilizando. Otra forma de evitar este tipo de inconvenientes es usar la denominada "clobber list" (la cual nombre anteriormente como lista de registros destruidos).

Veamos otro ejemplo sacado del código fuente de Routix , el cual muestra el uso del modificador "r"

```
#define load_cr3(x) __asm__ ("movl %0, %%cr3" : : "r" (x) )
```

Esta macro carga un nuevo directorio de paginas. No importa realmente si usted no sabe que es la paginación, simplemente piense en CR3 como un registro más, el cual no puede ser cargado de forma inmediata (es decir, requiere ser cargado a través de otro registro). Este caso es un perfecto ejemplo el cuál deja a Gcc utilizar el registro de propósito general que tenga disponible.

3.8 Clobber List

Ya hemos dado una explicación de porque es importante incluir los registros que modifica nuestro código en la clobber list, por lo que ahora vamos a apoyar esa teoría con algunos ejemplos:

El siguiente ejemplo esta sacado también del código fuente de Routix . Es la interfaz en modo usuario de la llamada al sistema exec (su comportamiento es ligeramente diferente a la del estandar Unix). Elegí este fragmento de código porque aporta información sobre el uso no solo de la clobber list (o lista de registros destruidos) si no también de la lista de salida.

```
int exec (char *tarea)
{
    __asm__ __volatile__ ("movl %0, %%eax ; movl %1, %%ebx" :  : "g" \
                (SYS_PROCESS | SYS_EXEC) , "g" (tarea) : "eax" , "ebx");
    __asm__ __volatile__ ("int $0x50");

     int retorno;
    __asm__ __volatile__ ("movl %%eax, %0" : "=g" (retorno)  );

    return retorno;
}
```

La primera linea no hace más que cargar en el registro EAX el número de llamada al sistema correspondiente, y en EBX la dirección de un string que posee el path del programa a ejecutar. Lo que aporta de nuevo esta función es que al final incluye un: "eax", "ebx" que corresponde a la clobber list. Con este agregado le estoy diciendo explícitamente a gcc que los registros EAX y EBX estan siendo modificados para que él pueda tomar los recaudos necesarios. (Tanto SYS_PROCESS como SYS_EXEC son macros, las cuales son consideradas como valores inmediatos).

La segunda línea hace pasar al modo kernel vía la interrupción 0x50.

El último fragmento agrega el modificador "=g" en la lista de salida. Hasta ahora habíamos pasado valores de variables a registros, pero nunca en sentido inverso, es en este último caso donde la output list (o lista de salida) entra en juego. La linea

```
__asm__ __volatile__ ("movl %%eax, %0" : "=g" (retorno)  )
```

mueve el valor del registro EAX a lo que representa el %0, que en este caso es la variable retorno. Es evidente el agregado del signo "=" antes de la "g", el cual es necesario en el modificador usado en la lista de salida.

Veamos algún ejemplo más de la lista de salida.

Este ejemplo es la implementación de la función inportb en Routix :

```
unsigned char inportb(word puerto )
{
    unsigned char valor;

    __asm__ __volatile__("inb %w1,%b0" : "=a" (valor) : "d" (puerto) );
    return valor;
}
```

Aca podemos ver algunos agregados más. No se si habrán notado que siempre nos manejamos con datos de 4 bytes, es decir con el registro completo. No siempre es necesario esto, en algunos casos alcanza con un word (AX) y en otro incluso con un byte (AL). El caso de la instrucción INB (o IN) es un claro ejemplo. El

número de puerto es un entero de 2 bytes, mientras que el valor leído de él es de tan solo 1. El operador %w le dice al gcc que el dato que recibe es un word (2 bytes), mientras que el %b hace lo respectivo a 1 byte.

Si compilamos este ejemplo con gcc -S obtenemos

```
_inportb:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        movl    8(%ebp), %eax
        movw    %ax, -2(%ebp)
        movw    -2(%ebp), %dx
/APP
        inb %dx,%al
/NO_APP
        movb    %al, -3(%ebp)
        movl    $0, %eax
        movb    -3(%ebp), %al
        leave
        ret
```

donde podemos comprobar que realmente el gcc utiliza los registros parciales DX y AL.

4 Varios

En esta sección intento adjuntar algunos cometarios que considero importantes.

4.1 Modificadores '\n' y '\t'

Probablemente si ven código fuente que usa ensamblado en línea como Linux o Routix (por qué no ?) encuentren entre instrucción e instrucción un modificador '\n' o '\t'. Estos no son más que los famosos modificadores de formato utilizados en C. El caracter '\t' tabula 8 espacios mientras que '\n' avanza hacia una nueva linea. Que función cumplen aquí ? Sencillamente ayudan a generar un código más claro.

Pueden hacer la prueba de compilar

```
__asm__ ("movl %0, %%cr3;nop" : : "r" (puerto) );

        o bien

__asm__ ("movl %0, %%cr3\n\tnop" : : "r" (puerto) );
```

háganlo y comparen los resultados.

4.2 Que es volatile ?

En muchos casos pueden haber encontrado luego de un __asm__ al modificador __volatile__ . Este tiene por objeto decirle al Gcc que sea cual fuera el modo de compilación no debe alterar el código escrito por ustedes en el *inline assembly.* Si no lo hacen, y sólo escriben un __asm__ el Gcc intentará optimizar todo el código con el fin de lograr mayor eficiencia, lo cuál puede llegar a ir contra nuestras intenciones. Miren un poco el código fuente de Linux para ver que tanto Linus Torvalds utilizó a ese modificador.

5 Comentario final

No debe considerarse que este texto es la máxima sobre el *inline assembly*, si no que es un pequeño aporte de su servidor para facilitar la comprensión de la mezcla de C y assembly. La práctica es un recurso fundamental en el proceso de aprendizaje, asi que es necesario que cada uno se siente a hacer sus propios ejemplos y ver cual es el resultado de la compilación con el Gcc. El manual escrito por Richard Stallman es mucho más completo que este, asi que resulta muy provechoso leerlo.

6 Contacto

Me gustaría recibir todo tipo de críticas constructivas con el fin de enriquecer y facilitar la compresión de este texto. Cualquier cosa que parezca redundante, o poco clara con gusto será modificada. Espero que les sea de utilidad.

martincad@yahoo.com