Estudio de un Sistema Operativo
Diseño de Sistemas Operativos

# Problemas de aula 5. Lenguaje ensamblador desde C

Pascual Pérez, Juan Carlos Pérez & Sergio Sáez

Pascual Pérez pperez@disca.upv.es
Sergio Sáez ssaez@disca.upv.es
Juan Carlos Pérez jcperez@disca.upv.es

---

## 5.1 Introducción.

- En esta sesión practicaremos el uso de la directiva `asm`.
- El conocimiento de esta directiva es fundamental para la comprensión de algunas funciones del núcleo de Linux.
- Durante la lección se plantearán ejercicios para demostrar el acceso, desde un fragmento de código escrito en ensamblador, a los diferentes componentes de un programa en lenguaje C: Funciones, variables, parámetros locales, etc.

Si disponemos del lenguaje C, ¿Para qué necesitamos ensamblador? Existen, básicamente, dos razones:

- **Mejora del espacio ocupado y del tiempo de ejecución de un programa:** Los compiladores optimizan muy bien el código, pero en ocasiones, utilizar directamente determinadas instrucciones muy específicas de un procesador mejora de manera considerable estos dos factores. Por ejemplo: stos, lods, etc. en IA32, que permiten copiar una cadena entera con muy pocas instrucciones. Existen diversos ejemplos en `include/asm-i386/string.h`.
- **Programación de sistemas:** Los procesadores actuales disponen de mecanismos que facilitan la gestión de memoria, de procesos y de dispositivos de E/S. Estos mecanismos se utilizan para el desarrollo de los sistemas operativos y el acceso a los mismos desde un lenguaje de alto nivel sería imposible sin herramientas como la directiva `asm`.

## 5.2 Documentación.

- Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture

Descripción general de la arquitectura i386.

- Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual

  Manual de referencia con el juego de intrucciones completo de la arquitectura i386.

- GCC-Inline-Assembly-HOWTO

  Manual de uso de la directiva **asm** en gcc.

NOTA: La versión 4.3 del gcc que está instalada por defecto funciona de una manera diferente y con ella no veréis los efectos de los ejemplos que siguen. Usad la versión 4.1 del gcc. Si no está instalada en la máquina que estáis usando, instaladla haciendo:

```
$ sudo apt-get install gcc-4.1
$ alias gcc="gcc-4.1"
```

## 5.3 Un ejemplo

Copiad este programa simple en un editor. Trabajad en un subdirectorio de vuestro **home** que puede llamarse **dirasm**.

```
#include <stdio.h>
int func1(void){
    register unsigned int a=4;
    register unsigned int b=7;
    register unsigned int c;

    c=a+b;
    asm("nop");             /* separador */
    asm ("movl %ecx,%eax"); /* registro EAX= registro ECX */
    asm("nop");             /* separador */
    return c;
}
int main (int argc, char ** argv){
    printf(" %d \n",func1());
    return 0;
}
```

- Compila y ejecuta el programa indicándole al compilador que no optimice con **-O0** (letra "O" mayúscula seguida de cero). ¿Qué ocurre?

  

- Analiza cuál es el problema, entiende bien lo que ha ocurrido y elimina la línea que lo provoca. Emplea las herramientas que ya estudiaste para desensamblar (**objdump**), para ver el código resultado del preprocesado (**gcc -E**), y para que el compilador no llegue a ensamblar (**gcc -S**). Esto último permite ver todo el programa traducido a ensamblador.

  

- Ahora el programa funciona ilustrando el hecho de que el compilador desconoce que determinados elementos del procesador (registros, *flags*, memoria, etc) han sido modificados desde dentro de un fragmento en lenguaje ensamblador y por lo

tanto no los protege. Existe una directiva `asm` extendida que permite describir los elementos explícitos e implícitos que modifican las instrucciones en ensamblador.

## 5.4 La directiva asm extendida.

La directiva extendida permite:

- Introducir código en ensamblador.
- Acceder a los diferentes elementos de un programa en lenguaje C (funciones, variables, etc.).
- Informar al compilador de qué elementos del sistema, registros, *flags*, memoria, se modificarán durante la ejecución del fragmento de código en ensamblador y de qué forma. Esto le permite al compilador prever que elementos, generalmente registros, tiene que recargar, guardar en pila, etc.

El formato de la directiva es:

```
__asm__ ( plantillas de instrucciones
          : operandos de salida           /* opcional */
          : operandos de entrada          /* opcional */
          : registros y flags modificados /* opcional */
        );
```

### 5.4.1 La Plantilla de instrucciones.

Estudiemos mediante el siguiente ejemplo cómo es una plantilla de instrucción:

```
__asm__ ( "movl %0,%%eax"
          : /* sin operandos de salida */
          : "b" (mi_variable)
          : /* el compilador no sabe qué objetos se modifican */
        );
```

- El operando fuente `%0` no es válido en ensamblador puro, Esto no es ni un registro, ni una dirección de memoria, ni un operando inmediato.
- `%0` identifica a un operando de entrada o de salida. Se permiten hasta 10 operandos, del `%0...%9`.
- En el ejemplo, `%0` se asocia al único operando de entrada `"b"`. `"b"` es la abreviatura del registro `EBX` en IA32.

### 5.4.2 Los operandos de entrada.

El formato de los operandos es el siguiente:

`"modificador" (expresión en lenguaje C) , "modificador" (expre ...`

- El compilador genera código ensamblador necesario para relacionar la expresión en C con el modificador.
- Los modificadores hacen referencia a un elemento del procesador, registros, memoria, *flags*, etc.

- Ejemplos:
  - `"c" (count), "a" (fill_value), "D" (dest)` Asocia la variable `count` al registro `ECX`, `fill_value` a `EAX`, y `dest` a `EDI`.
  - `"r" (count)` Asocia a la variable `count` un registro de propósito general: el que más le convenga al compilador.
  - `"m" (quantity)` `quantity` se tratará como una posición de memoria.
  - `"0" (count)` Asocia la variable `count` al mismo elemento al que se asocia al operando `%0`. Este operando también cuenta.
  - `"r" (3+4)` Asocia la expresión 3+4 (que el preprocesador se encargará de simplificar a 7), a un registro. Con esto la directiva nos permite incluir constantes.

### 5.4.3 Los operandos de salida.

El formato es muy similar al de los operandos de entrada. La diferencia es que estos operandos se reescriben obligatoriamente al final de la ejecución del código en ensamblador.

El formato es el siguiente:

- `"`*`remodificador`*` modificador" (expresión válida en lenguaje C)`, ...
  (sin espacio entre remodificador y modificador).

  El remodificador indica la forma en la que se reescribirá el modificador. Es decir, al final o durante la ejecución, algo similar a lo que ocurre en C con las expresiones: `++p, p++` .

- Ejemplos:
  - `"=c" (count)` Asocia la variable `count` al registro `ECX` descartando el valor previo. Con esto le indicamos al compilador que este operando se modificará al final del código en ensamblador.
  - `"&r" (count)` Asocia la variable `count` a un registro de propósito general. Le indicamos al compilador que este parámetro se modificará durante la ejecución del código antes de acabar de leer todos los de parámetros entrada, y por lo tanto, no puedo usarse un registro que también se use como parámetro de entrada o como parte de una dirección de memoria.

### 5.4.4 Los objetos modificados.

Le indicamos al compilador qué elementos del procesador se modificarán durante la ejecución del fragmento de código en ensamblador. Una operación de suma (`addl %eax,%eax`), cambia los códigos de condición y este efecto no aparece indicado en ningún lugar de la instrucción. Los elementos modificados tienen el siguiente aspecto:

- Registros: `"eax"`, `"ecx"`, etc,
- Posiciones de memoria: `"memory"`. Sólo tiene función informativa.

- Códigos de condición del procesador (*flags*): `"cc"`.

En el siguiente ejemplo se muestra la forma de indicar que la instrucción `btsl` modifica los códigos de condición. En este caso `btls` pone a 1 un bit y deja el antiguo valor en el *flag* `CF`.

```
__asm__ ("btsl %1,%0"
         : "=m" (ADDR)
         : "r" (pos)
         : "cc"
        );
```

## 5.5 Otro ejemplo.

Copia este programa, compílalo sin optimizar, y desensámblalo. Podrás observar cómo el compilador genera más código del que codificas dentro de la directiva `asm`. Genera código de entrada de operandos, codifica las instrucciones especificadas y genera código para escribir en los operandos de salida.

```
#include <stdio.h>
int main (int argc,char ** argv){
    int a,b,c;

    a=3;b=4;
    asm("nop");
    asm ("addl %1,%2;\n"
         "movl %2,%0; "
         : "=r" (c)
         : "r" (a), "r" (b)
        );
    asm("nop");
    printf("El resultado de %d+%d=%d \n",a,b,c);
    return 0;
}
```

- Cambia el modificador del segundo operando (`%1`), asociado a la variable `a`, a memoria `"m"` y observa los cambios.

- Cambia ahora el modificador del primer operando (`%0`), asociado la variable `c`, a memoria `"m"` y observa los cambios.

- Cambia ahora el modificador del tercer operando (`%3`), asociado la variable `b`, a memoria `"m"` y observa los cambios. ¿Qué ocurre?. ¿Por qué?. Busca en el manual de IA32 los posibles formatos de `add`.

- Con los tres operandos en memoria, intenta añadir instrucciones para que funcione. Pista: utiliza instrucciones `movl` y `addl`.

## 5.6 La palabra reservada `register`.

El modificador `register` es una palabra reservada del lenguaje C, como lo son `while`, `if`, `int`, etc. anteponiendo ésta, en la declaración de una variable, le indicamos al compilador que trate la variable como un registro y no como una posición de memoria.

- Compila el primer ejemplo, con las declaraciones `register` y sin ellas y compara el resultado de las compilaciones. ¿Qué cambia?

- Declara la variable `c` con `register`, haz que tenga un ámbito global y después compila el programa (compila siempre con `-Wall`). ¿Qué ocurre?,

- Observa que el compilador limita a ciertos ámbitos el uso de este modificador. ¿Por qué no deja que sea una variable global?


## 5.7 La palabra reservada `volatile`.

El modificador `volatile` es una palabra reservada como `register`. Anteponiendo ésta, en la declaración de un objeto del lenguaje C, le indicamos al compilador que el objeto es volátil ;D. En realidad en C el objeto persiste en todo su ámbito, pero no su valor, que sí puede ser volátil (cambiante) Si se aplica a:

- **A una variable:** Que el valor de la variable puede ser alterado por otro componente del sistema, por ejemplo: un DMA, otro procesador, etc. Cada vez que hagamos referencia a esa variable se leerá/escribirá de/a memoria (realiza la función contraria a `register`), por si ésta ha sido modificada. Por ejemplo: El compilador, que optimiza, sabe cuándo las variables dentro de un bucle se utilizan mucho y decide ponerlas en un registro interno, es decir, copiar su valor antes de entrar en el bucle y trabajar con esta copia. Si en algun momento otro componente del sistema la modifica, dentro del bucle no sé verá reflejado ese cambio. Para evitar esta situación, anteponemos `volatile` al tipo de la variable.
- **A una directiva `asm`:** En este caso se le dice al compilador que no optimice el fragmento, es decir, que allí donde aparezca, se mantenga tal y como lo ha escrito el programador.

Edita el programa, compila (con `-O0`) el programa con `volatile` y compara el resultado de la compilación del mismo sin el modificador.


```
#include <stdio.h>

int main (int argc, char ** argv)
{
        volatile unsigned int a;
        volatile unsigned int b;
        int c;

        a=4;
        b=a+7;
```

```
        c=a+b;

        printf("Valor de c: %d \n",c);
        return 0;
}
```

## 5.8 Llamadas a función

Recuerda lo que viste en una sesión anterior sobre el funcionamiento de la pila y las llamadas a funciones. Edita y compila el siguiente ejemplo:

```
int div (int a, int b){ int c; c=a/b; return c;}
int main (int argc, char ** argv){
        int result;
        asm ("nop");
        __asm__ ("subl $0x08,%%esp; \n"
                 "pushl %1; \n"
                 "pushl %2; \n"
                 "jmp   div; \n"
                 "addl $0x10, %%esp; \n"
                 "movl %%eax, %0;"
                 : "=m" (result)
                 : "r" (12), "r" (4)
                 : "memory" );
        asm ("nop");
        printf("la division de 12/4 es: %d \n",result);
        return 0;
}
```

- ¿Por qué no funciona? Realiza en papel una traza del programa y representa el estado de la pila antes y después de la llamada.

- Cambia las instrucciones necesarias para que la pila tenga una estructura consistente con lo que espera la función.

- El programa ¿realiza la división correctamente? observa de nuevo cómo la función `div` recoge los argumentos. Corrige el problema.


## 5.9 Saltos locales

Edita, compila y ejecuta los dos programas. ¿Por que uno retorna al shell y el otro no? (analiza el código generado con `objdump`)

```
int main (int argc, char ** argv){
        __asm__ ("1: nop; \n"
                 "jmp 1b; \n"
                 "1: nop" : : );
        return 0;
}

int main (int argc, char ** argv){
        __asm__ ("1: nop; \n"
                 "jmp 1f; \n"
                 "1: nop" : : );
```

```
        return 0;
}
```

## 5.10 Ejercicio 1

Ejercicio 1: Interpreta lo que hace esta función:

```
static inline char * func1(char * cad1,const char *cad2)
{
int d0, d1, d2;
__asm__ __volatile__( "1: lodsb ;"
                      "   stosb ;"
                      "   testb %%al,%%al;"
                      "   jne 1b"
: "=&S" (d0), "=&D" (d1), "=&a" (d2)
: "0" (cad2),"1" (cad1)
: "memory");
return cad1;
}
```

Notas:

- `%%al` es un registro de 8 bits del procesador i386.
- Las instrucciones:
  - `lodsb` carga el byte apuntado por el registro ESI en el registro `%al`.
  - `stosb` carga en la posición de memoria apuntada por el registro EDI el valor del registro `%al`
  - `testb` comprueba el valor de un registro modificando adecuadamente los *flags*.
  - `jne` salta si `testb` ha puesto a zero el *flag* `ZF`, en ejemplo. Si `%al` no es cero, salta.

Para analizar la función, ayúdate de las herramientas que ya conoces: la orden `gcc` con la opción `-E` o `-S`, la orden `objdump` con la opción `-d`, etc.

## 5.11 Ejercicio 2

El siguiente fragmento de código ¿qué hace? y ¿cómo?. Ayúdate para averiguarlo de la orden `gcc` con la opción `-E`. Fíjate que no utiliza ninguna función de biblioteca.

```
#define __NR_write 4
#define _syscall_3(type,name,type1,arg1,type2,arg2,type3,arg3) \
type name(type1 arg1,type2 arg2,type3 arg3) \
{ long __res; \
__asm__ volatile ( "int $0x80" \
: "=a" (__res) \
: "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
"d" ((long)(arg3))); \
return (type) (__res); \
}
inline _syscall_3(int,write,int,fd, const char *, buff, int, count)
int main(int argc, char ** argv){
 write(1,"Hola Mundo\n",11);
}
```