1. Ensamblador Linux

Aquest tutorial està extret directament de l'adreça web: http://www.publispain.com/supertutoriales/programacion/ensamblador/cursos/1/linasm.htm

1.1. Sintaxis del Ensamblador .AT&T x86

DJGPP usa la sintaxis propia de AT&T, que es un poco diferente de la que estamos acostumbrados a ver. Las diferencias principales son las siguientes:

- En la sintaxis de AT&T el destino de una operación se pone en el segundo parámetro y el origen se pone en el primer parámetro. (Al revés que en los Intel).
- El nombre de los registros es precedido por el carácter porcentaje ("%").
- Los valores inmediatos han de ser precedidos por el carácter "$".
- El tamaño de los operandos se especifica mediante un último carácter de la instrucción:"b" (8-bits) | "w" (16-bits) | "l" (32-bits).

Veamos algunos ejemplos. (Entre paréntesis figura el equivalente en Intel)

```
movw %bx, %ax   (mov ax, bx)
xorl %eax, %eax (xor eax, eax)
movw $1, %ax    (mov ax,1)
movb X, %ah     (mov ah, byte ptr X)
movw X, %ax     (mov ax, word ptr X)
movl X, %eax    (mov eax, X)
```

La mayoría de los opcodes son idénticos entre el formato AT&T y el formato Intel excepto para estos:

```
movsSD (movsx)
movzSD (movzx)
```

Donde S representa el tamaño del origen y D el del destino ("b","w" o "l"). Por ejemplo: "movswl %ax, %ecx (movsx ecx, ax)".

```
cbtw        (cbw)
cwtl        (cwde)
cwtd        (cwd)
cltd        (cdq)
lcall $S,$O (call far S:O)
ljmp $S,$O  (jump far S:O)
lret $V     (ret far V)
```

Los prefijos de instrucción no deberían (mas bien no deben) escribirse en la misma línea de la instrucción sobre la que actúan. Por ejemplo, "rep" y "stosd" deberían escribirse en dos líneas separadas.

Los direccionamientos a memoria se expresan de distinta manera. La sintaxis típica Intel para direccionar la RAM sigue el siguiente modelo:

SECTION:[BASE + INDEX*SCALE + DISP]

se escribe en AT&T como sigue

SECTION:DISP(BASE, INDEX, SCALE).

He aquí algunos ejemplos: (con sus equivalentes en Intel)

```
movl 4(%ebp), %eax            (mov eax, [ebp+4])
addl (%eax,%eax,4), %ecx      (add ecx, [eax + eax*4])
movb $4, %fs:(%eax)           (mov fs:eax, 4)
movl _array(,%eax,4), %eax    (mov eax, [4*eax + array])
movw _array(%ebx,%eax,4), %cx (mov cx, [ebx + 4*eax + array])
```

Las instrucciones de salto nunca se ven precedidas como en Intel de las palabras reservadas "near", "short" o "far" pues el ensamblador elige la instrucción a ensamblar en cada caso tendiendo a optimizar el código eligiendo las instrucciones que desplazan menos el puntero de instrucciones.

Dado que las instrucciones de salto condicionales en la arquitectura del PC solo pueden ser de tipo "short" todas las instrucciones siguientes tienen un único byte para el desplazamiento. Ejemplo de algunas instrucciones: "jcxz", "jecxz", "loop", "loopz", "loope", "loopnz" and "loopne". Remediamos este problema como lo hacíamos en el formato de ensamblador de los Intel, utilizando etiquetas intermediarias para los saltos como se ve en el siguiente código para realizar la lógica "jcxz foo".

```
jcxz cx_zero
jmp cx_nonzero
cx_zero:
jmp foo
cx_nonzero:
```

Hay que tener cuidado cuando utilicemos las instrucciones de multiplicación como "mul" o "imul". En este tipo de instrucciones, si expresamos dos operandos, solo tendrá en cuenta el ensamblador el PRIMER operando. Así, la instrucción "imul $ebx, $ebx" no pondrá el resultado en "edx:eax". Para esto, debemos utilizar la misma instrucción pero con un solo operando, es decir, la instrucción "imul %ebx".

1.2. Ensamblador en linea - (dentro de C o C++)

Vamos a empezar por la macro "asm" pues su funcionamiento es frecuentemente preguntado. Su sintaxis basica es la que se describe a continuación:

```
__asm__(sentencias en ensamblador
: salidas
: entradas
: registros modificados);
```

Los cuatro campos son:

- sentencias en ensamblador en formato AT&T separados por un retorno de carro
- Salidas - indicador seguido del nombre entre paréntesis separado por una coma
- Entradas- indicador seguido del nombre entre paréntesis separado por una coma
- Registros modificados - nombres separados por una coma

El ejemplo más sencillo:

```
__asm__("
      pushl %eax\n
      movl $1, %eax\n
      popl %eax"
);
```

Los 3 últimos campos no son indispensables pues si no se utilizan variables o punteros de entrada o de salida y no quieres informar al compilador de los registros que modificas por si se te olvida alguno no pasa nada.

Veamos un ejemplo más complejo con **variables de entrada**.

```
int i = 0;
__asm__("
      pushl %%eax\n
      movl %0, %%eax\n
      addl $1, %%eax\n
      movl %%eax, %0\n
      popl %%eax"
      :
      : "g" (i)
);
/* i++; */
```

¡No os desesperéis todavía! Tratare de explicarlo. Nuestra variable de entrada "i" queremos incrementarla en 1. No tenemos variables de salida ni registros modificados (pues se restaura eax). Por esta razón el segundo y cuarto parámetro están vacíos.

Si se especifica el campo de entrada es necesario especificar el campo de salida hallan o no hallan variables de salida. Simplemente se deja sin especificar (el campo vacío). Para el ultimo campo esta operación no es necesaria. Se debe poner un espacio o un retorno de carro para separar un campo de otro.

Analicemos el campo de la **entrada**. El indicador del campo no es ni más ni menos que una directiva que indica al compilador como debe gestionar las variables. Toda directiva debe ponerse entre dobles comillas. En este caso la directiva "g" indica al compilador que decida donde desea almacenar el parámetro (pila, registro, memoria) y se utiliza mucho pues generalmente los compiladores optimizan bien el código. Otra directiva útil es la directiva "r" que le permite cargar la variable en cualquier registro que este libre de uso. De igual manera: "a" (ax/eax), "b" (bx/ebx), "c" (cx/ecx), "d" (dx/edx), "D" (di/edi), "S" (si/esi), etc.

El primer parámetro de entrada se simboliza como "%0" dentro de las sentencias de ensamblador. Y así en el mismo orden de especificación en el campo de entrada. Es decir, que para N entradas y ninguna salida "%0" hasta %N-1 simbolizaran las variables de entrada en el orden en el que se listen en el campo 3.

¡Algo muy importante! Si se utiliza el campo de entrada, salida o de modificación de registros los nombres de los registros deben ser precedidos de 2 % ("%%eax") en vez de uno como solíamos hacerlo.

Veamos ahora el significado de la directiva __volatile__ después de __asm__ en un ejemplo con dos entradas.

```
int i=0, j=1;
__asm__ __volatile__("
      pushl %%eax\n
      movl %0, %%eax\n
      addl %1, %%eax\n
      movl %%eax, %0\n
      popl %%eax"
      :
      : "g" (i), "g" (j)
);
/* i = i + j; */
```

Queda claro que "%0" simboliza "i" y que "%1" simboliza "j", verdad? Pero entonces ¿para qué sirve la directiva **volatile**? Simplemente previene al compilador para que modifique nuestras sentencias de ensamblador si le es posible para optimizar el código (reordenación, eliminación de código inútil, recombinación). Es una opción muy recomendada.

Pasemos ahora a como se especifican los parámetros de salida. Veamos un ejemplo:

```
int i=0;
__asm__ __volatile__("
      pushl %%eax\n
      movl $1, %%eax\n
      movl %%eax, %0\n
      popl %%eax"
      : "=g" (i)
);
/* i++; */
```

Todos los indicadores o directivas especificados en el campo de variables de salida deben ser precedidos de "=" y van a ser ordenados y simbolizados como en el caso de las variables de entrada. ¿Cómo entonces se distingue uno de entrada con uno de salida? Veamos un ejemplo.

```
int i=0, j=1, k=0;
__asm__ __volatile__("
      pushl %%eax\n
      movl %1, %%eax\n
      addl %2, %%eax\n
      movl %%eax, %0\n
      popl %%eax"
      : "=g" (k)
      : "g" (i), "g" (j)
);
/* k = i + j; */
```

Si se ha entendido todo lo anterior solo puede no entenderse como distinguir un parámetro de entrada con uno de salida pues he aquí la explicación.

Cuando utilizamos parámetros de **entrada y salida**

%0 ... %K representan las salidas

%K+1 ... %N son las entradas

Así en el ejemplo anterior "%0" se refiere a "k", "%1" a "i" y "%2" a "j". No era tan complicado verdad?

Más adelante puede sernos útil utilizar el tercer campo pues nos evita de utilizar la pila para conservar y restaurar los registros modificados. Veamos el ejemplo anterior con este campo en vez de las instrucciones "push" y "pop".

```
int i=0, j=1, k=0;
__asm__ __volatile__("
      movl %1, %%eax\n
      addl %2, %%eax\n
      movl %%eax, %0"
      : "=g" (k)
      : "g" (i), "g" (j)
      : "ax", "memory"
);
/* k = i + j; */
```

Como puede deducirse el registro de 32 bits "eax" se ve modificado por nuestra rutina en ensamblador y sin embargo especificamos como registro de 16 bits "ax" como modificado, ¿por qué? Se debe simplemente a que los registros de 16 bits indicados en el tercer campo recogen todos los tamaños posibles de los mismos (32,16,8)

Si modificamos la memoria (escribimos en variables) es recomendable especificar la directiva "memory". En todos los ejemplos anteriores deberíamos haber utilizado esta directiva pero por razones de simplicidad no se ha utilizado.

Las etiquetas locales dentro del ensamblador en línea debe terminar por una b o una f según si esta después o antes la etiqueta relacionada con la instrucción de salto. Creo que queda lo suficientemente claro en el siguiente ejemplo.

```
__asm__ __volatile__("
      0:\n
      ...
      jmp 0b\n
      ...
      jmp 1f\n
      ...
      1:\n
      ...
);
```

1.3. Ensamblador Externo

El mejor modo de aprender a utilizar el ensamblador interno es analizar los ficheros en ensamblador generados por el C mediante "gcc -S file.c". Su esquema básico es:

```
.file "myasm.S"

.data
  somedata: .word 0
  ...

.text
  .globl __myasmfunc
  __myasmfunc:
    ...
    ret
```

¡Y las macros! Simplemente es necesario incluir el fichero de librería para definirlas en ensamblador. Simplemente incluimos ese fichero en nuestro fuente en ensamblador y las usamos de la manera adecuada. Veamos un ejemplo, myasm.S:

```
#include <libc/asmdefs.h>
.file "myasm.S"
.data
  .align 2
  somedata: .word  0
  ...
.text
  .align 4
  FUNC(__MyExternalAsmFunc)
  ENTER
    movl   ARG1, %eax
    ...
    jmp    mylabel
    ...
  mylabel:
    ...
  LEAVE
```

Este puede ser un buen esqueleto de fuente para utilizar el ensamblador externo.