V I R U S

Tutorial: Programación en Assembler bajo Linux
Autor: Diego Echeverri Saldarriaga
Descripcion: Documento dedicado principalmente a novatos en la materia. Tenemos una intro bastante asequible explicando desde lo más básico, con varios ejemplos en código. La cosa se complica en los dos capítulos siguientes si no sabes nada de programación ya que toma ejemplos de otros lenguajes. Plantea algunos problemas para mejorar en la materia, pero desgraciadamente no trae las soluciones.
Dificultad: Novato, apenas requiere conocimientos..
DOWNLOAD: http://mgu7mxp3vxzfj7hs.onion/
Herramientas: El ensamblador as y el enlazador ld, el depurador gdb y un editor hexadecimal de tu elección.
Pasado a formato HTML por: Virus FECHA: ../../....

INTRODUCCION:

La programación bajo Assembler es una parte fundamental del curso de Arquitectura ST-039. La idea de este tutorial es que sea suficientemente fácil y explicito como para poder realizarse sin supervisión, así como también hacer la introducción a dicho lenguaje mediante una aproximación desde el Pseudocódigo.
Capítulo 1

Muestra como esta estructurado un programa en Assembler bajo Intel. Se explican las directivas mas utilizadas del ensamblador as y los comandos del gdb, como también, el manejo de los registros y tamaños de variables utilizadas por el procesador. Esta sección deberá durar como máximo dos sesiones.

Capítulo 2

Este capitulo explicará el control de flujo (If, While, For) a partir de las instrucciones de Loop y saltos que provee Intel. La Duración estimada es de 1 sesión.

Capítulo 3

Este capitulo estará dedicado al manejo de funciones y modularidad de un programa realizado en Assembler. Se utilizará una sesión para esta parte.

AL ATAKE:

CAPITULO 1. Programación, ensamblado, enlazado y depuración.

En esta sección se explicará como está organizado un programa en Intel. A veces se hace referencia

a algunos aspectos de la arquitectura (como es el caso del sistema de registros) con el fin de que aquél que posea ya conocimientos en programación bajo Intel con herramientas como TASM y Visual Basic pueda de una vez adaptarse al entorno de as.

Para aquél que no esté familiarizado, sería útil dar una lectura rápida de la primera parte, con el objetivo de al menos generar dudas que serán solucionadas en la segunda parte, la solución del primer problema en Assembler.

Estructura básica de un programa.

Este es el esqueleto básico de un programa en Assembler de Intel:

```
1.     /*
2.     Este es un tipo de comentario
3.     */
4.     #Otro tipo de comentario
5.     #nombre de archivo (opcional)
6.     .file "Ejemplo1.s"
7.     #secciones
8.     .section .data
9.     # <- datos
10.    # <- datos
11.    #PROGRAMA
12.    .section .text
13.    .global _start
14.    _start:
15.    #FINALIZA PROGRAMA
16.    xorl %eax, %eax
17.    incl %eax
18.    xorl %ebx, %ebx
19.    int $0x80
20.    .end
```

Comentarios

Realizar comentarios en los programas de Assembler pasa de ser una buena costumbre de programación a una necesidad vital. En as se permite realizar comentarios de dos formas:

1. '#' para comentarios de una sola línea

2. '/**/' para comentarios de varias líneas

Secciones

El código de Assembler debe ser relocalizado en la fase de enlazado con el fin de que cada parte ocupe la sección que le corresponde. Existen básicamente 5 secciones de enlazado pero, por razones prácticas, nos ocuparemos únicamente de la de datos **.data** y la de "texto" **.text**.

Las secciones **.text** y **.data** (que aparecen en las líneas 8 y 12) son muy parecidas, la única diferencia es que la sección **.text** es comúnmente compartida entre procesos, además, contiene

instrucciones y constantes (el código es .text). La sección **.data** es usualmente alterable.

Directiva .global

La directiva **.global** hace visible el símbolo que se le pase al **ld**. En el caso de nuestro ejemplo (línea 13), se le está dando el nombre del label donde comienza el programa.

Fin del programa

Las líneas 15 a 20 muestran como debe terminar un programa en Linux, también nos sirven como primer ejemplo de instrucciones de Assembler. Una instrucción de Assembler consta de:

label: código de instrucción parámetro 1 parámetro 2...

Label

La idea de un *label* es "guardar" la ubicación donde se encuentra, ya sea para una instrucción o para hacer referencia al área de datos (lo que se suele llamar el left-hand side de la variable). Un *label* es un identificador seguido de ':'. Cuando se vaya a hacer referencia a este se nombra solo el identificador. Este concepto quedará más claro una vez resolvamos nuestro primer problema. Los *labels* son opcionales.

Código de instrucción y parámetros

Los códigos de instrucción y sus respectivos parámetros, están dados por los manuales específicos del procesador en el que se esté trabajando, como también del Ensamblador que se utilice. Para el caso de **Intel** se puede consultar disponible en línea en (http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html). Las diferencias de sintaxis (Sintaxis AT&T) dadas por el ensamblador son básicamente las siguientes:

- Los registros se escriben de la forma **%registro**

- Los números (valores inmediatos) deben estar acompañados del prefijo **$**.

- Los códigos de instrucción que operen sobre 32 bits se les debe agregar el sufijo **l**.

- Las referencias "source,destination" de Intel se invierten.

Para el caso del fin del programa, la secuencia de instrucciones es la siguiente:

```
xorl %eax, %eax
```

Realiza un or exclusivo de eax con eax. Como eax es igual a eax, el resultado de esta operación deja eax en '0'.

```
incl %eax
```

Esta instrucción incrementa eax.

```
xorl %ebx, %ebx
```

Igual que la primera instrucción.

```
int $0x80
```

Esta es una llamada a una interrupción, le dice al sistema operativo que el programa terminó su ejecución.

Un primer problema. Hallar la sumatoria de 5 números alojados en memoria.

Por más trivial que parezca el problema siempre resulta útil hacer el esquema de ejecución a modo de pseudocódigo. De esta forma garantizamos, en primer lugar, una buena lógica de programa, y en segundo, reducir la labor de programación a una simple traducción de un lenguaje al cual ya estamos familiarizados. Como primer intento podríamos hablar de un pseudocódigo de alto nivel de este tipo:

```
Input: N 1 , N 2 , N 3 , N 4 , N 5

Output: Sumatoria

return N 1 + N 2 + N 3 + N 4 + N 5 ;
```

Ahora bien, si tenemos en cuenta que Assembler es un lenguaje donde las operaciones son predominantemente binarias, valdría la pena convertir nuestro algoritmo a uno que solo requiera operaciones de éste tipo:

```
Input: N 1 , N 2 , N 3 , N 4 , N 5
Output: Sumatoria
Acumulador = 0;
Acumulador += N 1 ;
Acumulador += N 2 ;
Acumulador += N 3 ;
Acumulador += N 4 ;
Acumulador += N 5 ;
return Acumulador
```

Ya con el algoritmo de operaciones binarias, podemos comenzar la solución desde Assembler.

Variables

Algo importante en la programación en Assembler es tener claras las variables que se utilizarán. Para esto, se debe tener en cuenta lo siguiente:

1- Nombre de la variable. Toda variable ha de tener un nombre. Debe ser de fácil recordación ya que el orden es muy importante en la programación en Assembler.

2- Tamaño de la variable. Definir de que tamaño es la variable (64 bits, 32 bits, 16 bits, 8 bits, o un espacio de memoria de N bits).

Según nuestro algoritmo necesitaremos 6 espacios de memoria. Cinco para los números que necesitamos sumar y un espacio para el acumulador.

El procesador Intel posee espacios para almacenar variables, estos espacios se denominan Registros. Las ventajas que nos ofrece la utilización de registros es la velocidad de acceso, por esto es preferible que los datos que se utilicen con mayor frecuencia se alojen en registros. Los registros de Intel que podemos utilizar se organizan de esta forma:

Registros de 32 bits EAX EBX ECX EDX

Registros de 16 bits Corresponden a la mitad derecha de registros de 32 bits y son: AX BX CX DX

Registros de 8 bits Corresponden a cada una de las mitades de los registros de 16 (H para High y L para Low) y son: AH AL BH BL CH CL DH DL

Conociendo esto, alojaremos nuestras variables de la siguiente forma:

```
N 1 , N 2 ...N 5  32 bits  Memoria
Acumulador  32 bits  EAX
```

Las variables en memoria se definen en el área de datos así:

label: tamaño valor_inicial

Aquí el label sirve como nombre de la variable, es decir, el nombre con el cual haremos referencia al valor guardado. El tamaño se define como *.long* para 32 bits, *.word* para 16 bits, y *.byte* para 8 bits.

Los valores iniciales equivalen a los números que tendrá el programa. Por defecto se leen en decimal pero pueden escribirse en binario (inicializando la cadena con "0b"), en hexadecimal (inicializando la cadena con "0x") o en octal inicializándola en "0").

Ya con esto podemos escribir nuestra sección **.data** para alojar las variables a las que hallaremos la sumatoria (alojaremos 100, 100, 100, 100, y -100).

```
.section .data
Numero1: .long 100
Numero2: .long 0144
Numero3: .long 0x64
Numero4: .long 0b1100100
Numero5: .long -100
```

Implementando el Algoritmo

Para la implementación del algorítmo se debe buscar si alguna instrucción del procesador hace lo que nosotros necesitamos, de lo contrario, será necesario descomponer aquella instrucción en una secuencia que ya exista. Para consultar las instrucciones más utilizadas visite http://www.jegerlehner.ch/intel/index_es.html

La primera instrucción pone el acumulador (EAX) en 0. La instrucción *clrl %eax* nos realizará esta tarea.

Para sumar utilizaremos (add origen, destino). Asi nos quedará el código del programa de esta

forma:

```
1. .file "Sumatoria.s"
2. .section .data
3. Numero1: .long 100
4. Numero2: .long 0144
5. Numero3: .long 0x64
6. Numero4: .long 0b1100100
7. Numero5: .long -100
8. .global _start
8.1. .text
9. _start:
10. clrl %eax
11. addl Numero1,%eax
12. addl Numero2,%eax
13. addl Numero3,%eax
14. addl Numero4,%eax
15. addl Numero5,%eax
16. xorl %eax, %eax
17. incl %eax
18. xorl %ebx, %ebx
19. int $0x80
20. .end
```

Ejecutando el algoritmo

Para ejecutar el algoritmo, debemos primero ensamblarlo con **as**. En general, las herramientas GNU siguen el formato *programa opciones archivo_fuente*. Las opciones que nos interesan de **as** son las siguientes:

-**o** Con esta opción se especifica el archivo de salida.

--**gstabs** Con esta opción se genera la información para hacer el debuggin.

Por tanto, nos queda el comando de esta forma:

```
as --gstabs -o Sumatoria.o Sumatoria.s
```

Cuando tenemos el programa objeto (Sumatoria.o), debemos enlazarlo. Para esto usamos el enlazador de GNU, **ld**.

```
ld -o Sumatoria Sumatoria.o
```

Sumatoria es el ejecutable.

Haciendo debuggin

Para hacer el debug del programa, utilizamos la herramienta **gdb**. Para llamarlo utilizamos:

```
gdb ejecutable
```

Lo cual nos abre una ventana de este tipo:

```
$ gdb Ejemplo1
GNU gdb 6.3.50_2004-12-28-cvs (cygwin-special)
Copyright 2004 Free Software Foundation, Inc.
```

```
GDB is free software, covered by the GNU General
Public License, and you are welcome to change it
and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type
"show warranty" for details.
This GDB was configured as "i686-pc-cygwin"...
(gdb)
```

En este punto podemos utilizar los diferentes comandos de **gdb**. Estos son los más importantes:

list: Muestra el código fuente, es útil para tener las líneas exactas donde se ponen los puntos de parada (break).

break linea Pone un punto de parada en la línea indicada.

break *dirección Pone un punto de parada en la dirección indicada.

run Ejecuta el programa.

kill Termina el programa.

quit Sale de gdb.

info registers Muestra el estado de los registros.

info variables Muestra el estado de las variables.

print variable: Muestra el valor de la variable.

step n Avanza *n* instrucciones. Cuando *n* no está definido, avanza un paso.

next n Avanza *n* instrucciones. Cuando *n* no está definido, avanza un paso, pero no entra dentro de los *calls*.

Ejercicios

Manejo de variables

Cambiar el programa anterior para que trabaje con variables de 16 bits y de 8

Invertir

Hacer un programa que dados 5 números alojados en memoria (los números del programa anterior), invierta el orden de los valores asignados en un principio. Es decir, que N1 sea N5, N2 sea N4, N3 sea N3, N4 sea N2, y N5 sea N1.

Operaciones aritméticas

Dados dos números en memoria, A y B, hacer un programa que al final deje almacenado en EAX A+B, en EBX A-B, en ECX A*B, y en EDX A/B.

Operaciones lógicas

Dados dos números en memoria, A y B, hacer un programa que al final deje almacenado en EAX AandB, en EBX AorB, en ECX notA, y en EDX AxorB. **Nota de virus:** para la conjunción,

disyunción, negación y disyunción exclusiva, respectivamente. Estas instrucciones tienen en común que realizan sus operaciones "bit a bit". Es decir, el procesador realiza tantas operaciones lógicas como bits tienen los operandos tomando los bits que ocupan la misma posición y, por tanto, produciendo otros tantos resultados. Analizar con el depurador para comprender.

Capítulo 2. Control de flujo y estructuras indexadas.

Control de flujo

El control de flujo en los programas de alto nivel se realiza con estructuras *If*, *While*, *For*. En Ensamblador dichas estructuras, no existen, en cambio, existen los saltos condicionales, una especie de *goto* condicionados, con los cuales se pueden escribir cada una de las rutinas que utilizan los lenguajes de alto nivel. En esta sección veremos las instrucciones de Assembler de Intel que nos permiten hacer esto.

Instrucción Loop

La instrucción **loop** es muy útil para hacer secuencias un determinado número de veces. Es decir, resulta útil para algoritmos de tipo *for i = n to 100 do*.

El loop funciona de la siguiente manera, primero hay que decrementar ECX:

```
dec %ECX
```

En segundo lugar comprobamos el nuevo valor de ECX, y si es igual a 0 pasamos a la siguiente instrucción, si no es igual a 0 salta a la etiqueta que se le pasa por parámetro. Siendo así, una secuencia de loop será de la forma:

```
1. mov n, %ecx
2. bucle:
3. ##SECUENCIA
4. loop bucle
```

Una cosa importante a tener en cuenta, es que dado que el valor que controla el número de saltos está en ECX, dentro de la secuencia es importante que el valor no se altere, o si se altera, conservarlo en una variable, para cuando se ejecute la instrucción Loop, restaurarlo.

Comparación y saltos no condicionales

En Assembler existen las instrucciones *cmp* y los saltos *jnz*, *jz*, *jmp*, etc. Las cuales son útiles para la implementación del resto de secuencias de control.

CMP: Sirve para comparar. Internamente es una resta entre los dos operandos que se comparan.

JE: Este es un salto que se ejecuta cuando la comparación anterior dio como resultado que los dos operandos son iguales.

JNE: Salta si no es igual.

JG: Salta si es mayor.

JGE: Salta si es mayor o igual.

JL: Salta si es menor.

JLE: Salta si es menor o igual.

JMP: Salta siempre.

Implementando el If Then Else

Una secuencia If Then Else tiene la siguiente forma:

```
If A operador B Then
SECUENCIA DE PASOS SI SE CUMPLE LA CONDICIÓN
Else
SECUENCIA DE PASOS SI NO SE CUMPLE LA CONDICIÓN
End
```

Esto se puede implementar en Ensamblador de la siguiente forma:

```
1. cmp A,B
2. SALTO Else
3. SECUENCIA DE PASOS SI SE CUMPLE LA CONDICIÓN
4. jmp fin
5. Else:
6. SECUENCIA DE PASOS SI NO SE CUMPLE LA CONDICIÓN
7. fin:
```

Donde SALTO, es una instrucción cuya condición de salto sea **contraria** al operador utilizado, es decir, igual JNE, no igual JE, menor o igual JG, mayor o igual JL, menor JGE y mayor JLE.

Implementando el While

Siguiendo exactamente la misma manera que utilizamos para generar el If, podemos implementar el While.

```
While A operador B Do
SECUENCIA DE PASOS DENTRO DEL WHILE
End
```

Que en ensamblador sería algo así:

```
1. While:
2. cmp A,B
3. SALTO salir
4. SECUENCIA DE PASOS DENTRO DEL WHILE
5. jmp While
6. salir:
```

Ejercicios: encontrar el número mayor entre un arreglo (apuntador donde comienza la estructura de datos) no ordenado.

Otra vez usaremos variables de 32 bits. Siendo así, una forma de acomodar las variables es de esta forma:

Usaremos 4 variables de 32 bits. La primera se llamará "Tam" y estará alojada en memoria. La segunda se llamará "Arreglo" y también estará alojada en memoria. La tercera variable se llamará "mayor" y se alojará en EAX. La cuarta y última variable, se llamará "i", y estará alojada en ECX.

Ya con esto podemos definir nuestra sección **.data**. Inicialmente, comenzaremos con un arreglo de tamaño 5. Para definir un arreglo utilizaremos la directiva *.int*:

```
.section .data
Tam: .long 5
Arreglo: .int 0,2,4,8,10
```

Implementando el algoritmo

Para realizar algoritmos "grandes" lo más útil casi siempre es implementar de a pequeños fragmentos. Por esto, primero implementaremos la parte interna del *for*, es decir, el *if* siguiendo lo que vimos antes.

```
1. cmpl %eax, prueba
2. JLE else
3. movl prueba, %eax
4. jmp fin
5. else:
6. EN ESTE CASO NO HAY ELSE
7. fin:
```

Por ahora, utilizaremos una variable en memoria llamada prueba, con el fin de garantizar que el fragmento esté correcto, despues reemplazaremos esto por la correspondiente posición. Una vez hallamos probado la parte del if según lo indicado más arriba, podemos implementar el for tal y como ya se explicó.

```
.file "MayorenArreglo.s"
.section .data
Tam: .long 5
Arreglo: .int 0,2,4,8,10
prueba: .long 4
.text
.global _start
_start:
movl Tam, %ecx
loop1:
cmpl %eax, Arreglo(,%ecx,4)
JLE else
movl Arreglo(,%ecx,4), %eax
jmp fin
else:
#En este caso no hay Else
fin:
loop loop1
xorl %eax, %eax
incl %eax
xorl %ebx, %ebx
int $0x80
.end
```

El manejo indexado funciona de la siguiente manera. Cuando llamamos *Arreglo(,ecx,4)* estamos usando la dirección base del arreglo, estamos indexando con ECX (con lo cual nos iterará todas las posiciones), y con el 4 le decimos que el tamaño de los datos es de 4 bytes (una palabra).

Ejercicios propuestos

Problema del programa anterior: En la solución de la búsqueda del número mayor hay un grave error. ¿Cúal es? ¿Cómo se soluciona?

Indexación de varias dimensiones: Realice el mismo programa, pero que haga la búsqueda en una matriz de 2 dimensiones.

Búsqueda de un elemento: Realice la búsqueda de un elemento en un arreglo, y retorne el ínidice donde se encuentre.

Control de flujo: Realice cualquier tipo de búsqueda en el arreglo, pero utilizando el control de flujo *While* y *Do While*.

Capítulo 3. Funciones y Modularidad

Funciones

A medida que un programa aumenta de complejidad, se detectan ciertas porciones de código que realizan lo mismo basados en 0 o más parámetros. A estos fragmentos, podemos agruparlos en funciones. Independiente de la arquitectura utilizada, una función está constituida por dos componentes: **el llamador** y **el llamado**.

El llamador (caller) y el llamado (called)

El llamador es la parte de código que tiene estos objetivos: colocar los parámetros en el sitio adecuado, guardar la dirección de retorno e invocar el llamado.

El llamado es la parte que se encarga de: garantizar que todo quede como estaba, localizar (si los hay) los parámetros, definir (si las hay) las variables locales, recuperar la dirección de retorno, retornar el control al llamador.

Una aproximación al llamador y al llamado desde C

```
1. int main(void) {
2. int a, b;
3. a = 10;
4. b = 20;
5. sumar(a,b);
6. return 0;
7. }
8. void sumar(int x, int y){
9. int z;
10. z = x + y;
11. }
```

Podemos comenzar a diferenciar que partes del código de la función *sumar* son responsabilidad de

que parte de la función (llamador o llamado). En la línea 5 se puede ver que le estamos pasando los parámetros a y b, es decir, estamos cumpliendo con la primera responsabilidad del Llamador. También podemos decir que después de ejecutarse esta línea, C pasará a la siguiente (línea 6) se cumple también con la segunda responsabilidad del Llamador, es decir, esta instrucción después de ejecutarse, sabe donde debe continuar. En tercer lugar, sabe también que esta línea ejecuta el código que escribimos para *sumar*, lo cual quiere decir que cedimos el control al Llamado. La función sumar reconoce dentro de su ámbito, las variables *x* e *y*. Esto corresponde a Localizar los parámetros. En la línea 9 definimos una variable local, lo cual también definimos como responsabilidad del Llamado.

Una aproximación al Llamador y al Llamado desde ASM

Dado el código asm:

```
1. .section .data
2.
3. a: .long 4
4. b: .long 5
5.
6. .section .text
7. .global sumar
8. sumar:
9. pushl %ebp
10. movl %esp, %ebp
11. movl 8(%ebp), %ebx
12. movl 12(%ebp), %eax
13. addl %ebx, %eax
14. leave
15. ret
16.
17. .global _start
18. _start:
19.
20. # "Main"
21. pushl a
22. pushl b
23. call sumar
24. popl %ebx
25. popl %ebx
26.
27. # Finalizo el programa
28. xorl %eax, %eax
29. incl %eax
30. xorl %ebx, %ebx
31. int $0x80
32. .end
```

Como pudimos ver en el llamador de C, una sola línea tenía a su cargo todas las responsabilidades. En ASM somos nosotros los que debemos pasarle los parámetros. Existen varias formas de pasar parámetros, en este caso los pasaremos por **pila**. Para esto, tenemos simplemente que hacerle PUSH a cada uno como se ve en las líneas 21 y 22. La utilización del CALL realiza varias cosas. En primer lugar, guarda la dirección donde se encuentra, para que cuando se termine de ejecutar la función, esta sepa a que dirección debe retornar. También, funciona como una especie de salto incondicional, es decir, irá a la primera posición de memoria donde se encuentra ubicada la función.

Las líneas 9 y 10 sirven para la creación del marco de pila, que es la forma como Intel maneja funciones. En las líneas 11 y 12 estamos recuperando los parámetros que pasamos por pila, mediante un direccionamiento indexado. Con las instrucciones *leave* y *ret* (líneas 14 y 15), regresamos a la línea posterior donde hicimos el call.

Después de haber realizado el call, ponemos dos *pop* (líneas 24 y 25) para desocupar los parámetros que pasamos dentro de la función. Esto con el objetivo de cumplir con la responsabilidad que tiene el Llamado de "dejar las cosas como estaban".

Resolviendo un problema

Dado un arreglo de tamaño N y el apuntador a éste, hallar la sumatoria utilizando la función suma de la sección anterior. Esta vez llamaremos a la función suma desde un archivo externo.

Solución en pseudocódigo

Ya que utilizaremos la función que anteriormente implementamos (suma), podemos hacer referencia a esta, sin tener que implementarla de nuevo (realmente se trata de una sola línea ASM, así que no hay diferencia, pero para problemas grandes esto resulta muy conveniente).

```
sumatoria = 0
i = 0
While i menor que Tamaño Do
Sumatoria = Sumar(Arreglo[i],Sumatoria)
Incrementar i
end
return Sumatoria
```

Las variables que usaremos, de 32 bits, son Tam y Arreglo[], que estarán alojadas en memoria. Y usaremos las variables Sumatoria e i que estarán alojadas en EAX y ECX respectivamente.

Implementando el algoritmo

Para aumentar el nivel de "orden", utilizaremos un archivo separado para la función *suma* que ya habíamos implementado. De esta forma:

```
1. .file "Suma.s"
2. .section .text
3. .global suma
4. suma:
5. pushl %ebp
7. movl %esp, %ebp
8. movl 8(%ebp), %ebx
9. movl 12(%ebp), %eax
10. addl %ebx, %eax
11. leave
12. ret
```

El archivo principal será de la siguiente manera:

```
.file "Principal.s"
.section .data
Tam: .long 5
Arreglo: .int 0,2,4,8,10
.section .text
.global _start
```

```
.extern suma
_start:
movl $0, %ecx
movl $0, %eax
while1:
cmpl Tam, %ecx
jge finwhile1
pushl Arreglo(,%ecx,4)
pushl %eax
call suma
pop %ebx
pop %ebx
inc %ecx
jmp while1
finwhile1:
# Finalizo el programa
xorl
%eax, %eax
incl %eax
xorl %ebx, %ebx
int $0x80
.end
```

Donde la directiva **.extern** es aquella que le dice al enlazador, que se recurrirá a una función que se encuentra en otro archivo.

Ensamblando y enlazando

Se debe tener en cuenta que en el proceso de ensamblado y enlazado se debe realizar de la siguiente manera. Primero se debe ensamblar cada fuente:

```
$ as -gstabs -o Suma.o Suma.s
$ as -gstabs -o Principal.o Principal.s
```

Despues se enlaza de la siguiente forma:

```
$ ld -o Principal Suma.o Principal.o
```

Ejercicios propuestos

Paso de parámetros: Cambie la función para trabajar con otras formas de paso de parámetros (por registros y área de memoria).

Parámetros por referencia: Cambie el programa de sumatoria a una función que reciba como parámetros un número N y el apuntador (sugerencia, la instrucción LEA puede ser útil).

Funciones recursivas: Implemente una función recursiva que halle factorial.

FIN

Fonte: http://mgu7mxp3vxzfj7hs.onion/