Tutorial de Programación en Assembler Bajo Linux

Diego Echeverri Saldarriaga

10 de julio de 2006

Índice general

Introducción

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 explicara 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.

Capítulo 4 En esta parte se explicará **SPARC** mediante la comparación del ya conocido ensamblador de Intel.

Capítulo 1

Programación, Ensamblado, Enlazado y Depuración

En esta sección se explicará como esta 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 fín de que aquel 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 aquel que no esté familiarizado, seria 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 solucion del primer problema en Assembler.

1.1. Estructura básica de un programa

Este es el esqueleto simplificado 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
```

1.1.1. 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 2 formas:

- '#' para comentarios de una sola línea (4).
- "/ ∗ ∗/" para comentarios de varias líneas (1-3).

La sección de comentarios esta en [1] (sección 3.3)

1.1.2. 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`, para ver las otras secciones se puede recurrir a [1] (sección 4).

Las secciones `.text` y `.data` (que aparecen en las líneas 8 y 10) 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[1].

1.1.3. Directiva `.global`

La directiva `.global` hace visible el simbolo que se le pase al **ld**. En el caso de nuestro ejemplo, se le esta pasando el nombre del label (Ver: 3) donde comienza el programa.

1.1.4. Fin del Programa

Las lineas (15-20) 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>  <parametro 1>, <parametro 2> ...
```

[1] Para efectos de lo que se realizará durante el curso, la diferencia no es muy importante.

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 mas 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 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 [2] disponible en linea en (`http://www.intel.com/design/pentium4/manuals/index_new.htm`). Las diferencias de sintaxis (Sintaxis AT&T) dadas por el ensamblador son básicamente las siguientes, y pueden ser consultadas para mayor detalle en [1] (Sección 8.8).

- Los registros se escriben de la forma `%<registro>`.
- Las 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:` Ésta 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.

1.2. Un Primer Problema:

Hallar la sumatoria de 5 números alojados en memoria.

1.2.1. Solución en Pseudocódigo

Por mas trivial que paresca 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 podriamos hablar de un pseudocódigo de alto nivel de este tipo.

Input: N_1, N_2, N_3, N_4, N_5
Output: Sumatoria
return $N_1 + N_2 + N_3 + N_4 + N_5$;

Algoritmo 1: Primera Aproximación

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

Input: N_1, N_2, N_3, N_4, N_5
Output: Sumatoria
$Acumulador \longleftarrow 0$;
$Acumulador \mathrel{+}= N_1$;
$Acumulador \mathrel{+}= N_2$;
$Acumulador \mathrel{+}= N_3$;
$Acumulador \mathrel{+}= N_4$;
$Acumulador \mathrel{+}= N_5$;
return *Acumulador*

Algoritmo 2: Ahora con operaciones binarias

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

1.2.2. 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 debe tener 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 Bytes)

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 los 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, L para Low) y son: AL, AH, BL, BH, CL, CH, DL, DH

Conociendo esto, alojaremos nuestras variables de la siguiente forma.

Nombre	Tamaño	Lugar
$N_1, N_2 ... N_5$	32 bits	Memoria
Acumulador	32 bits	EAX

Las Variables en memoria se definen en el área de datos asi:

```
<label>: <tamaño> <valor inicial>
```

Aqui el label sirve como el 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 inicializandola 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, -100).

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

1.2.3. 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 instruccion en una secuencia que ya exista. Para consultar

las instrucciones se puede buscar en [2] o también existe una hoja de referencia con las instrucciones más utilizadas en `http://www.jegerlehner.ch/intel/IntelCodeTable_es.pdf`.

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
```

[2]

1.2.4. Ejecutando el algorítmo

Para ejecutar el algoritmo, debemos primero ensamblarlo con **as**.

En general, las herramientas de GNU siguen el formato:

```
<Programa> <opciones> <archivo(s) fuente(s)>
```

Las opciones que nos interesan de **as** son las siguientes (para mas información ver [1] Sección 1).

[-o] Con ésta opción se especifica el archivo de salida.

[–gstabs] Con esta opción se genera la información para hacer el debbuging

[2] Nótese la linea extra 8.1, se debe poner esta línea para garantizar que las instrucciones sean tomadas por **gdb**

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.

1.2.5. Haciendo Debugging

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

```
gdb <ejecutable>
```

Lo cual nos abre una ventana de éste 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 mas importantes, para verlos se puede consultar [3].

***l*ist:** muestra el código fuente, es útil para tener las lineas exactas donde se ponen los puntos de parada (break).

***b*reak <linea>:** pone un punto de parada en la línea indicada.

***b*reak *<dir>:** pone un punto de parada en la dirección indicada.

***r*un:** ejecuta el programa

***k*ill:** termina el programa

***q*uit:** sale de **gdb**

***i*nfo *r*egisters:** muestra el estado de los registros.

***i*nfo *va*riables:** muestra el estado de las variables.

***p*rint <Variable>:** muestra el valor de la variable.

***s*tep n:** avanza n "instrucciones". Cuando n no esta definido, avanza un paso

***n*ext n:** avanza n "instrucciones". Cuando n no esta definido, avanza un paso, pero no entra dentro de los *calls*.

1.3. Ejercicios

1.3.1. Manejo de variables

Cambiar el programa de 1.2.3 para que trabaje con variables de 16 bits y de 8 bits.

1.3.2. Invertir

Hacer un programa que dados 5 números alojados en memoria N_1, N_2, N_3, N_4, N_5, invierta el orden de los valores asignados en un principio. es decir: $N_1 \longrightarrow N_5$ $N_2 \longrightarrow N_4$ $N_3 \longrightarrow N_3$ $N_4 \longrightarrow N_2$ $N_5 \longrightarrow N_1$

1.3.3. Operaciones aritméticas

Dados 2 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

1.3.4. Operaciones lógicas

Dados 2 números en memoria A y B, hacer un programa que al final deje almacenado en EAX $A \vee B$, en EBX $A \wedge B$, en ECX $\sim A$ y en EDX $A \otimes B$

Capítulo 2

Control de flujo y Estructuras Indexadas

2.1. Control de Flujo

El control de flujo en los programas de alto nivel se realiza con estructuras *If, While, For*. En Assembler 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 leguajes de alto nivel. En esta sección veremos las instrucciones de Assembler de **Intel** que nos permiten hacer esto.

2.1.1. Instrucción *Loop*

?? La instrucción Loop es muy útil para hacer secuencias un determinado número de veces. Es decir, resulta útil para algoritmos de esta forma

Input: Entrada
Output: Salida
for $i \longleftarrow n$ *to* 1 **do**
 | //Secuencia de pasos;
end

El *Loop* funciona de la siguiente manera:

1. Se hace `dec %ECX`
2. Si ECX es igual a cero, pasa a la siguiente instrucción. Sinó, salta a la etiqueta que se le pasa por parametro

Siendo asi, una secuencia 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.

2.1.2. 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 las secuencias de control.

CMP: Sirve para comparar. Internamente es una resta entre los 2 operandos que se comparan

JE: Éste es un salto que se ejecuta cuando la comparación anterior dio como resultado que los 2 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*

Input: Entrada
Output: Salida
if *A operador B* **then**
  #SECUENCIA DE PASOS SI SE CUMPLE LA CONDICIÓN;
else
  #SECUENCIA DE PASOS SI **NO** SE CUMPLE LA CONDICIÓN;
end

Algoritmo 3: *operador* es cualquiera de $(=, \neq, \leq, \geq, <, >)$

Una secuencia If then else de la forma del Algoritmo 3, se puede implementar de la siguiente manera:

```
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 \textbf{NO} SE CUMPLE LA CONDICIÓN
7. fin:
```

Donde <Salto>, es una instrucción cuya condición de salto sea contraria al operador utilizado, es decir:

Operador	Instrucción de Salto
$=$	JNE
$\neq$	JE
$\leq$	JG
$\geq$	JL
$<$	JGE
$>$	JLE

Implementando el *While*

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

Input: Entrada
Output: Salida
while *A operador B* **do**
| #SECUENCIA DE PASOS DENTRO DEL WHILE;
end

Algoritmo 4: *operador* es cualquiera de $(=, \neq, \leq, \geq, <, >)$

```
1. while:
2. cmp a,b
3. <Salto> salir
4. #SECUENCIA DE PASOS DENTRO DEL WHILE\
5. jmp while
6. salir:
```

2.2. Un Primer Problema

Encontrar el número mayor en un arreglo no ordenado.

Input: $Arreglo[], Tam$
Output: $Mayor$
$Mayor \longleftarrow Arreglo[Tam]$;
for $i \longleftarrow Tam$ *to* 0 **do**
  if $Arreglo[i] > Mayor$ **then**
    $Mayor \longleftarrow Arreglo[i]$
  end
end
return $Mayor$

Algoritmo 5: Número Mayor en Arreglo

2.2.1. Solución en Pseudocódigo

2.2.2. Variables

Otra vez utilizaremos variables de 32. Siendo asi, una forma de acomodar las variables es de esta forma.

Nombre	Tamaño	Lugar
Tam	32 bits	Memoria
$Arreglo$[1]	32 bits*Tam	Memoria
$Mayor$	32 bits	EAX
i	32 bits	ECX[2]

Ya con esto podemos definir nuestra sección `.data`. Iniciamente 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
```

2.2.3. Implementando el Algoritmo

Para realizar algoritmos "grandes"lo mas 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 en la sección 2.1.2.

```
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:
```

[1] Arreglo viene siendo un apuntador a donde comienza la estructura de datos
[2] Se coloca en ECX a propósito, para seguir la estructura de 2.1.1

Por ahora utilizaremos una variable en memoria llamada prueba, con el fin de garantizar que el fragmento este correcto, despues reemplazaremos esto por la correspondiente posición.

Una vez hayamos probado la parte del *If* según lo indicado en 1.2.4, podemos implementar el *for* según lo visto en 2.1.1.

```
.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 (por lo cual nos iterará todas las posiciones), y con el cuatro le decimos que el tamaño de los datos es de 4 Bytes (una palabra).

2.3. Ejercicios

2.3.1. Problema del Programa Anterior

En la solución de la busqueda del número mayor hay un grave error, ¿Cuál és? ¿Cómo se soluciona?

2.3.2. Indexación de varias dimensiones

Realice el mismo programa, pero que haga la busqueda en una matriz de 2 dimensiones.

2.3.3. Búsqueda de un Elemento

Realice una busqueda de un elemento en un arreglo, y retorne el índice donde se encuentre

2.3.4. Control de Flujo

Realice cualquier tipo de busqueda en el arreglo pero utilizando el control de flujo *While* y *Do While*.

Capítulo 3

Funciones y Modularidad

3.1. 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[1] es la parte del código que tiene estos objetivos:

- Colocar Los parámetros en el sitio adecuado
- Guardar la dirección de retorno
- 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.

[1]Para la descripción del *Llamador* y *Llamado* se han utilizado ciertas notas textuales del Curso *Programación Digital* de Jose Luis Montoya Pareja

Una Aproximación al Llamador y el Llamado desde C

Dado un Código de éste tipo en 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 ya que después de ejecutarse esta línea, C pasará a la siguiente (línea 6) se cumple también con la segunda responsabilidad Llamador, es decir, esta instrucción despues de ejecutarse, sabe donde debe continuar. En tercer lugar, sabe tambien que esta linea 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 y 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 el 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 2 *pop* (líneas 24 y 25) para desocupar los parámetros que pasamos dentro de la función. Esto con el objetivo de cumplir la responsabilidad que tiene el llamado de "dejar las cosas como estaban".

3.2. Resolviendo Un Primer 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 la Función *suma* desde un archivo Externo.

3.2.1. 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).

Input: $Arreglo[], Tam$
Output: $Sumatoria$
$Sumatoria \longleftarrow 0;$
$i \longleftarrow 0;$
while $i < Tam$ **do**
  $Sumatoria \longleftarrow sumar(Arreglo[i], Sumatoria);$
  $i++;$
end
return $Sumatoria$

Algoritmo 6: Sumatoria de N números en Arreglo

3.2.2. Variables

Nombre	Tamaño	Lugar
Tam	32 bits	Memoria
$Arreglo[]$	32 bits	Memoria
$Sumatoria$	32 bits	EAX
i	32 bits	ECX

3.2.3. 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.

3.2.4. 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
```

Después se enlaza de la siguiente forma:

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

3.3. Ejercicios

3.3.1. Paso de Parámetros

Cambie la función para trabajar con otras formas de paso de parámetros (por registros y area de memoria)

3.3.2. Parámetros por Referencia

Cambie el programa de sumatoria a una función que reciba como parámetros N y el apuntador (sugerencia: La instrucción LEA puede ser útil)

3.3.3. Funciones Recursivas

Implemente una función recursiva que halle factorial.

Bibliografía

[1] Dean Elsner and Jay Fenlason. *Using as*, Enero 1994.

[2] *Software Developer's Manual IA-32 Intel Architecture*, Marzo 2006.

[3] Richard M. Stallman and Roland H. Pesch. *Debugging with GDB*, Enero 1994.