Introducción a Ensamblador

Por: Ing. Pablo Rojas
https://github.com/sthongurdt

SUPERCOMPUTACION Y CALCULO CIENTÍFICO

UNIVERSIDAD INDUSTRIAL DE SANTANDER

BUCARAMANGA, COLOMBIA

Introducción

Ensamblador es un **lenguaje de programación de bajo nivel para un dispositivo especifico, no es portátil** a través de múltiples sistemas. El código de ensamblador se pasa a lenguaje maquina gracias a programas como NASM, MASM, etc. Un conocimiento básico de cualquiera de los lenguajes de programación le ayudará a entender los conceptos de programación en ensamblador.

Cada computadora tiene un procesador que administra las actividades, **cada familia de procesadores tiene su propio conjunto de instrucciones** para manejar varias operaciones denominadas "instrucciones en lenguaje maquina". El lenguaje maquina es oscuro y complejo, por ello, **ensamblador está diseñado para una familia especifica de procesadores** para hacer su programación mas comprensible.

Ventajas

>> Se toma conciencia de

- Como interactúa el programa con el SO, CPU, BIOS y hardware del dispositivo
- Como se representan los datos en la memoria
- Como la CPU accede y ejecuta instrucciones
- Como las instrucciones acceden y procesan los datos

>> Menor uso de memoria y reducción del tiempo de ejecución

>> Es mas sencillo realizar trabajos complejos de hardware especifico

>> Es adecuado para trabajos en los que el tiempo es un factor critico

>> Es más adecuado para escribir rutinas de servicio de interrupción

Agenda

- Sintaxis básica
- Direccionamiento de datos a memoria
- Segmentos de memoria – Registro
- Llamadas de sistema
- Variables – Constantes
- Números, Cadenas
- Instrucciones Aritméticas
- Instrucciones Lógicas
- Condicionales
- Bucle
- Vectores
- Macros

Sintaxis básica

Un programa en ensamblador puede dividirse en tres secciones:

data: se usa para declarar datos inicializados o constantes. No se modifican en tiempo de ejecución y la sintaxis es:

```
section.data o data
```

bss: se usa para declarar variables, su sintaxis es:

```
section.bss o bss
```

text: se usa para mantener el código y le dice al kernel donde comienza la ejecución del programa, su sintaxis es:

```
section.text, global _start o _start:
```

Los comentarios se inician con un punto y coma (;):

```
add eax, ebx          ; suma de eax con ebx
```

Declaraciones en ensamblador

Los programas en ensamblador consisten en tres tipos de declaraciones:

Instrucciones (instrucciones ejecutables): indican al procesador que hacer, genera una instrucción en lenguaje maquina que es un código de operación (opcode).

Pseudo-ops (Directivas del ensamblador): le dicen al ensamblador acerca de aspectos del proceso de ensamblaje. No se ejecuta y no genera opcode.

Macros: son mecanismos de sustitución de texto.

Sintaxis de las declaraciones en ensamblador

Una instrucción básica tiene dos partes, el nombre de la instrucción a ejecutar (mnemonic) y los operandos (parámetros del comando), lo que esta entre [] es opcional.

[etiqueta] mnemonic [operandos] [;comentario]

EJP:

```
INC COUNT          ;Incrementar la variable de memoria COUNT
MOV TOTAL, 48      ;Transferir el valor 48 a la variable de memoria TOTAL
ADD AH, BH         ;Añadir el contenido del registro BH al registro AH
ADD MARKS, 10      ;Suma 10 a la variable MARKS
AND MASK1, 128     ;Realizar la operación AND en las variables MASK1 y 128
```

Direccionamiento de datos en memoria

Hay dos tipos de direcciones de memoria:

- **Dirección absoluta** - una referencia directa de una ubicación específica.
- **Dirección de segmento (offset)** - dirección de inicio de un segmento de memoria con el valor de offset.

Segmentos de memoria

Un modelo de memoria segmentada divide la memoria en grupos independientes referidos por punteros ubicados en el registro de segmentos, podemos especificar varios segmentos como:

- **Segmento de datos**: es donde se almacenan los elementos de datos para el programa, no se amplía después de ser declarada y permanece estática durante todo el programa (representado por section.data, section.bss (es una memoria llena de ceros, estática, que contiene búferes para datos que se declaran más adelante en el programa)).
- **Segmento de código**: es un área fija que define la memoria que almacena los códigos de instrucción.
- **Pila (stack)**: contiene los valores de los datos pasados a funciones y procedimientos dentro del programa.

Registros

Los registros almacenan elementos para su procesamiento sin necesidad de acceder a la memoria.

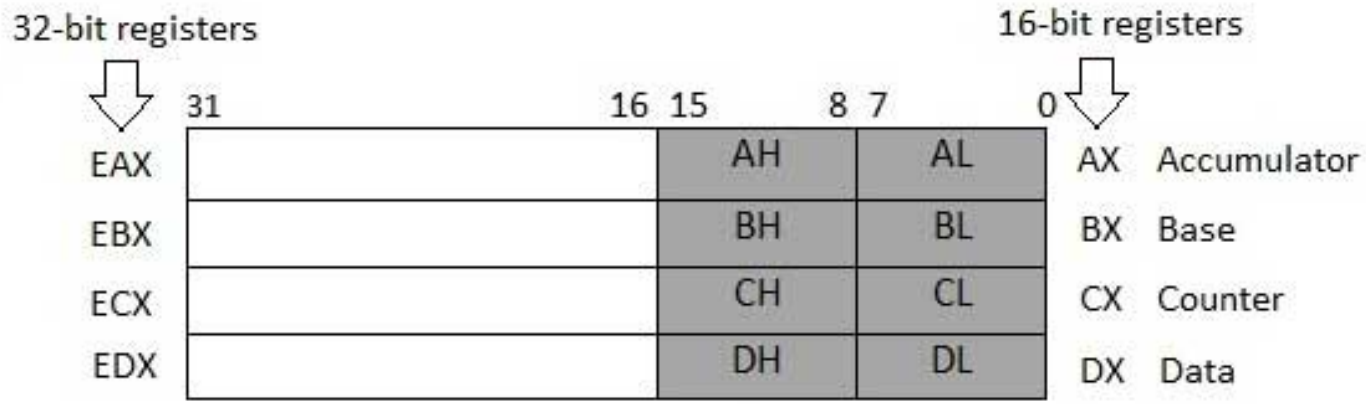
Los registros generalmente se dividen en los siguientes grupos:

- Registro de datos
- Registro de punteros
- Registro de índices

Registro de datos

Se pueden usar como:

- Registro de datos completo de 32 bits >> EAX, EBX, ECX, EDX
- La mitad inferior se puede usar como registro de datos de 16bits >> AX, BX, CX y DX.
- La mitad inferior tambien se puede usar como registro de datos de 8 bits >> AH, AL, BH, BL, CH, CL, DH y DL.

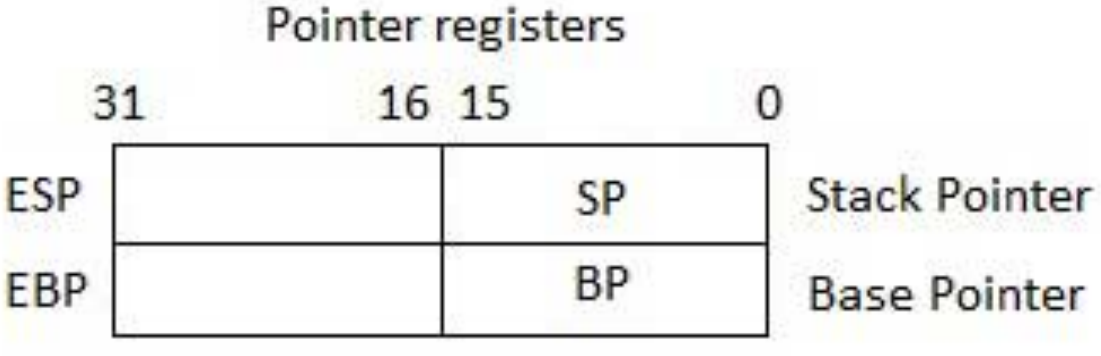


Mas info.

Registro de punteros

Hay tres categorías de registros de punteros:

- **Puntero de instrucción (IP)**: almacena la dirección de desplazamiento de la siguiente instrucción a ejecutar.
- **Puntero Pila (SP)**: proporciona el valor de offset dentro de la pila de programas.
- **Puntero base (BP)**: ayuda en la referenciación de las variables de parámetros pasadas a una subrutina.

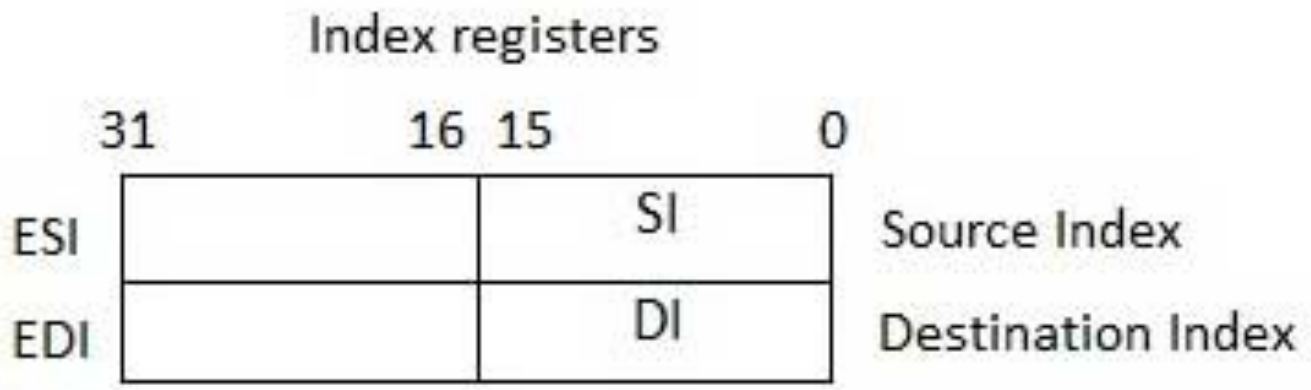


Mas info.

Registro de índices

Se utilizan para el direccionamiento indexado y a veces se utilizan en suma y resta. Hay dos conjuntos de índices:

- **Índice de fuente** (SI): Se utiliza como índice de fuente para operaciones de cadena (string).
- **Índice de destino** (DI): Se utiliza como índice de destino para operaciones de cadena.



Mas info.

Llamadas del sistema

Son APIs para la interfaz entre el usuario y el kernel. Las llamadas al sistema se ponen en el registro EAX. Hay seis registros utilizados que almacenan los argumentos de la llamada de sistema. Estos son el EBX, ECX, EDX, ESI, EDI, y EBP. Se llama la interupcion corespondiente (80), el resultado de muestra en EAX.

EJP :

```
mov edx,4          ;longitud del mensaje
mov ecx,msg        ;mensaje a escribir
mov ebx,1          ;descriptor de archivo (stdout)
mov eax,4          ;número de llamada al sistema (sys_write)
int 0x80           ;llamar al kernel
```

Mas info

Variable

La sintaxis para la declaración de asignación de almacenamiento para datos inicializados es:

[nombre-variable] definir-directiva iniciar-valor [, iniciar valor]

Donde nombre-variable identifica cada espacio de almacenamiento, existen cinco formas básicas:

```
choice          DB      'y'
number          DW      12345
neg_number      DW      -12345
big_number      DQ      123456789
real_number1    DD      1.234
real_number2    DQ      123.456
```

Directive	Purpose	Storage Space
DB	Define Byte	allocates 1 byte
DW	Define Word	allocates 2 bytes
DD	Define Doubleword	allocates 4 bytes
DQ	Define Quadword	allocates 8 bytes
DT	Define Ten Bytes	allocates 10 bytes

Mas info

Constante

Hay varias directivas que definen constantes, estas son tres:

EQU: se utiliza para definir las constantes. La sintaxis es la siguiente:

```
CONSTANT_NAME EQU expression
TOTAL_STUDENTS equ 50
```

%assign: puede usarse para definir constantes numéricas como la directiva EQU. Esta directiva permite una redefinición:

```
%assign TOTAL 10
```

Más adelante en el código, puedes redefinirlo como

```
%assign TOTAL  20
```

%define: permite definir constantes numéricas y de cadena. Esta directiva es similar a la #define en C. Esta directiva también permite la redefinición y distingue entre mayúsculas y minúsculas.

Hola mundo

```
section     .text
  global _start                ;debe ser declarado para el enlazador (ld)

_start:                        ;indica el punto de entrada del enlazador
  mov       edx,len            ;longitud del mensaje
  mov       ecx,msg            ;mensaje a escribir
  mov       ebx,1              ;descriptor de archivo (stdout)
  mov       eax,4              ;número de llamada al sistema (sys_write)
  int       0x80               ;llamada al kernel
  mov       eax,1              ;número de llamada al sistema (sys_exit)
  int       0x80               ;llamada al kernel

section     .data
msg db 'Hello, world!', 0xa    ;cadena a imprimir
len equ $ - msg                ;longitud de la cadena
```

Números

Los números y las instrucciones se representan en binario, cuando se ingresa o se muestra en pantalla un numero, esta en formato ASCII. Estas conversiones generan una sobrecarga, ensamblador permite procesar números de manera mas eficiente, se pueden representar de dos formas:

Formulario ASCII

BCD (Decimal Codificado en Binario)

Forma ASCII

ASCII almacena los decimales como una cadena de caracteres. Ejp: 1234 >> 31 32 33 34H.

Hay cuatro instrucciones para procesar números en representación ASCII:

> AAA - Ajuste ASCII después de la adición

> AAS - Ajuste ASCII después de la resta

> AAM - Ajuste ASCII después de la multiplicación

>AAD - Ajuste ASCII antes de la división.

```
_start:                 ;tell linker entry point
   sub     ah, ah
   mov     al, '9'
   sub     al, '3'
   aas
   or      al, 30h
   mov     [res], ax
```

Forma BCD

Hay dos tipos de representación: Empaquetada y Desempaquetada.

En la desempaquetada, cada byte almacena el equivalente binario de un digito decimal. Ejp:

1234 = 01 02 03 04H

En la empaquetada, cada digito se almacena usando 4 bits, 2 dígitos se almacenan en un byte. Ejp:

1234 = 12 34H

Hay dos instrucciones para procesar estos números:

>AAM - Ajuste ASCII después de la multiplicación

>AAD - Ajuste ASCII antes de la división

Las instrucciones ASCII también se pueden usar.

```
_start:                 ;tell linker entry point

   mov     esi, 4       ;pointing to the rightmost digit
   mov     ecx, 5       ;num of digits
   clc
add_loop:
   mov  al, [num1 + esi]
   adc  al, [num2 + esi]
   aaa
   pushf
   or   al, 30h
   popf

   mov  [sum + esi], al
   dec  esi
   loop add_loop
```

Cadena - Strings

Las cadenas pueden tener tantos caracteres como sean necesarios, su longitud puede especificarse de dos formas:

Almacenamiento explicito: Se usa el símbolo del contador “$”.

```
msg  db  'Hello, world!',0xa ;our dear string
len  equ  $ - msg            ;length of our dear string
```

$ apunta al byte después del último carácter de la variable de cadena msg.

Caracter Sentinel: Se usa al final para delimitar una cadena en lugar de almacenar la cadena.

```
message DB 'I am loving it!', 0
```

Instrucciones para Cadenas

Las instrucciones pueden requerir operando de origen, destino o ambos. Para segmentos de 32 bits, podemos usar los registros ESI y EDI para apuntar al origen y destino, respectivamente. Para los 16 bits, SI y DI apunta al origen y destino.

Hay 5 instrucciones básica para procesar cadena:

MOV: mueve un byte, word o doubleword de una ubicación de memoria a otra.

LOD: carga desde memoria, byte en AL, word en AX y doubleword en EAX.

STO: almacena datos de registro (AL, AX o EAX) a memoria.

CMP: compara elementos en la memoria, pueden ser byte, word, doubleword.

SCA: compara el contenido de un registro con el contenido de un elemento de memoria.

Instrucciones Aritméticas

INC: se utiliza para incrementar un operando en uno. Funciona en un solo operando que puede estar en un registro o en memoria, tiene la siguiente sintaxis:

```
INC EBX        ;Incrementa a registro de 32-bit
INC DL         ;Incrementa a registro de 8-bit
INC [count]    ;Incrementa la variable
```

DEC: igual que INC pero solo disminuye.

```
dec [value]
```

Instrucciones Aritméticas

ADD y SUB: se utilizan para realizar sumas y restas simples de datos, operando de 8 bits, 16 bits o 32 bits, tienen la siguiente sintaxis:

ADD/SUB destino, fuente

MUL/IMUL: Hay dos instrucciones para multiplicar los datos binarios. La instrucción MUL (Multiplicar) maneja datos sin signo y la IMUL (Multiplicar Enteros) maneja datos con signo, tienen la siguiente sintaxis:

MUL/IMUL multiplicador

DIV/IDIV: La operación de división genera dos elementos: un cociente y un resto, puede producirse un desbordamiento. La instrucción DIV maneja datos sin signo y la IDIV maneja datos con signo, tienen la siguiente sintaxis:

DIV/IDIV divisor

Mas info

Instrucciones Lógicas

El conjunto de instrucciones del procesador proporciona las instrucciones lógica AND, OR, XOR, TEST y NOT, que prueba, establece y borra los bits de acuerdo con la necesidad del programa.

Instrucción	Forma
AND	AND op1, op2
OR	OR op1, op2
XOR	XOR op1, op2
TEST	TEST op1, op2
NOT	NOT op1, op2

El primer operando en todos los casos podría estar en el registro o en la memoria. El segundo operando podría estar en registro / memoria o en un valor inmediato (constante). Sin embargo, **las operaciones de memoria a memoria no son posibles.** Estas instrucciones comparan o hacen coincidir los bits de los operandos y establecen los indicadores CF, OF, PF, SF y ZF.

Ejemplos

AND **Operand1: 0101** **Operand2: 0011** **----------** **After AND -> Operand1: 0001**	**OR** **Operand1: 0101** **Operand2: 0011** **----------** **After OR -> Operand1: 0111**
XOR **Operand1: 0101 0011** **After NOT -> Operand1: 1010 1100**	**TEST** La instrucción TEST funciona igual que la operación AND, pero a diferencia de la instrucción AND, no cambia el primer operando. Entonces, si necesitamos verificar si un número en un registro es par o impar, también podemos hacerlo usando la instrucción TEST sin cambiar el número original.

Condicionales

La ejecución condicional en lenguaje ensamblador se logra mediante varias instrucciones de bucle y ramificación. Estas instrucciones pueden cambiar el flujo de control en un programa. La ejecución condicional se observa en dos escenarios.

Salto incondicional: Esto se realiza mediante la instrucción JMP. La transferencia de control puede ser hacia adelante, para ejecutar un nuevo conjunto de instrucciones o hacia atrás, para volver a ejecutar los mismos pasos.

Sintaxis:

```
JMP    etiqueta
```

Salto condicional: Esto se realiza mediante un conjunto de instrucciones de salto j <condición>. Las instrucciones condicionales transfieren el control rompiendo el flujo secuencial y lo hacen cambiando el valor de compensación en IP.

Instrucción	Descripción	Bandera
JE/JZ	Salto igual o salto cero	ZF
JNE/JNZ	Salto no igual o salto no cero	ZF
JG/JNLE	Salto más grande o salto no menor igual	OF, SF, ZF
JGE/JNL	Salto mayor igual o salto no menor	OF, SF
JL/JNGE	Salto menos o salto no mayor igual	OF, SF
JLE/JNG	Salto menor igual o salto no mayor	OF, SF, ZF
Para Operadores Aritméticos:		

Instrucción	Descripción	Bandera
JE/JZ	Salto igual o salto cero	ZF
JNE/JNZ	Salto no igual o salto no cero	ZF
JA/JNBE	Saltar arriba o Saltar no abajo / Igual	CF, ZF
JAE/JNB	Saltar arriba / igual o saltar no debajo de	CF
JB/JNAE	Saltar abajo o saltar no arriba / Igual	CF
JBE/JNA	Saltar abajo / Igual o Saltar no arriba	AF, CF
Para Operadores Lógicos		

Instrucción CMP: La instrucción CMP compara dos operandos. Generalmente se usa en la ejecución condicional. Esta instrucción básicamente resta un operando del otro para comparar si los operandos son iguales o no. No perturba el destino o los operandos de origen. Se usa junto con la instrucción de salto condicional para la toma de decisiones.

Sintaxis:

CMP destino, fuente

Bucles

El conjunto de instrucciones del procesador incluye un grupo de instrucciones de bucle para implementar la iteración. La instrucción LOOP básica tiene la siguiente sintaxis

LOOP etiqueta

La instrucción LOOP supone que el registro ECX contiene el recuento de bucles. Cuando se ejecuta la instrucción de bucle, el registro ECX disminuye y el control salta a la etiqueta de destino, hasta que el valor del registro ECX, es decir, el contador alcanza el valor cero.

El fragmento de código anterior podría escribirse como:

```
mov ECX,10
l1:
<loop body>
loop l1
```

Arrays / Arreglos / Vectores

Podemos inicializar variables de tres maneras, hexadecimal, decimal o binaria. Por ejemplo: **MESES DW 0CH**, **MESES DW 12** o **MESES DW 1100B**.

También se puede definir una matriz unidimensional: **NUMEROS DW 34, 45, 56, 67, 78, 89**

> Esto declara una matriz inicializada con los números 34, 45, 56, 67, 78, 89 y 2x6=12 byte de espacio de memoria consecutiva.

TIMES también se puede usar para múltiples inicializaciones al mismo valor.

> INVENTORY DW 0, 0, 0, 0, 0, 0, 0, 0 >> INVENTORY TIMES 8 DW 0

Macros

Los macros garantizan una programación modular, un macro es una secuencia de instrucciones, asignada por nombre y se puede usar en cualquier parte del programa, la sintaxis es:

```
%macro macro_name  number_of_params
<macro body>
%endmacro
```

Donde, number_of_params especifica los parámetros numéricos, macro_name especifica el nombre de la macro.

Cuando necesite usar alguna secuencia de instrucciones muchas veces en un programa, puede poner esas instrucciones en una macro y usarlas en lugar de escribir las instrucciones todo el tiempo.

EJP:

Una necesidad muy común de programas es escribir una cadena de caracteres en la pantalla. Para mostrar una cadena de caracteres, necesita la siguiente secuencia de instrucciones:

```
mov     edx,len     ;message length
mov     ecx,msg     ;message to write
mov     ebx,1       ;file descriptor (stdout)
mov     eax,4       ;system call number (sys_write)
int     0x80        ;call kernel
```

```
; Un macro con dos parametros
; Implementando la llamada de sistema de escritura en pantalla
  %macro write_string 2
     mov   eax, 4
     mov   ebx, 1
     mov   ecx, %1
     mov   edx, %2
     int   80h
  %endmacro
```

```
section     .text
   global _start          ;must be declared for using gcc

_start:                   ;tell linker entry point
   write_string msg1, len1
   write_string msg2, len2
   write_string msg3, len3

   mov eax,1              ;system call number (sys_exit)
   int 0x80               ;call kernel

section     .data
msg1 db     'Hello, programmers!',0xA,0xD
len1 equ $ - msg1

msg2 db 'Welcome to the world of,', 0xA,0xD
len2 equ $- msg2

msg3 db 'Linux assembly programming! '
len3 equ $- msg3
```

Bibliografia

Para mas información, practicas, etc, visite las siguientes paginas web:

https://www.tutorialspoint.com/assembly_programming/index.htm

https://www.godbolt.org/

https://sites.cs.ucsb.edu/~franklin/64/lectures/mipsassemblytutorial.pdf

https://www.assemblylanguagetuts.com/mips-registers-table/

https://riptutorial.com/es/mips/example/29993/simulador-mars-mips