Código Objeto

INFORME ACADEMICO

Alumnos: Jesús Ernesto Zazueta Borbón - 16130362

Fecha: 25/11/2021

Mtro. Manuel Ramírez López

Materia: Lenguajes y Autómatas 2

Los ensambladores son programas que traducen código ensamblador a lenguaje de máquina, es decir, código binario. Trabajan muy de cerca con la arquitectura del procesador y, por lo tanto, son particularmente eficientes y económicos.

Un ensamblador traduce el código escrito en lenguaje ensamblador directamente a código binario, cuyo código se puede crear manualmente o por máquina. Por ejemplo, algunos compiladores primero convierten el código del programa en código ensamblador y luego llaman a un ensamblador. Esto, a su vez, funciona como un compilador en sí mismo y, como paso final, crea el código máquina.

Los programas de ensamblador pueden usar el conjunto de instrucciones completo de un procesador, porque para cada instrucción de ensamblador hay exactamente una contraparte en el nivel de la máquina. Los lenguajes modernos de alto nivel, por otro lado, se limitan a una selección del conjunto de instrucciones. Por este motivo, algunos lenguajes ofrecen la posibilidad de integrar lenguaje ensamblador si es necesario.

Cada procesador tiene su propia arquitectura y su propio conjunto de instrucciones con las que se puede abordar. Por tanto, cada procesador necesita su propio ensamblador personalizado, incluido su propio lenguaje ensamblador.

El ensamblador asociado solo puede entender y traducir el código escrito en este lenguaje. Un programa para el procesador A no puede ser utilizado por el procesador B sin cambios; los programas en ensamblador dependen en gran medida de la plataforma. En algunos casos, los lenguajes ensambladores individuales difieren solo mínimamente.

En algunos casos, por lo tanto, es relativamente fácil adaptar un programa a un nuevo procesador. Las diferencias también pueden ser tan grandes que los programas deben volver a desarrollarse por completo antes de que puedan transferirse a otra arquitectura.

Como ya se mencionó, cada ensamblador tiene su propio lenguaje que se adapta a la arquitectura de destino respectiva. Con el lenguaje ensamblador (abreviado: ensamblador), un programador puede escribir código que puede ser procesado

directamente por un ensamblador. Un ensamblador puede convertir programas en ensamblador directamente en código de máquina.

El lenguaje de programación utiliza abreviaturas mnemotécnicas para las instrucciones internas del procesador, con las que se pueden controlar sencillas operaciones lógicas y aritméticas, así como el acceso a los registros y el flujo del programa. Dependiendo de la arquitectura del procesador, se pueden agregar más operaciones.

La ventaja del código ensamblador es que la conversión a código binario es muy eficiente: el lenguaje ensamblador trabaja muy de cerca con la arquitectura respectiva y, por lo tanto, se puede traducir de manera extremadamente rápida y con requisitos mínimos de memoria. Los comandos se pueden traducir prácticamente 1: 1 y no es necesario analizarlos ni transferirlos a lenguajes intermedios.

En muchos casos, sin embargo, los compiladores de lenguajes modernos de alto nivel logran un rendimiento comparable o mejor. Las arquitecturas más complejas, en particular, requieren un alto nivel de conocimiento contextual para que todos los recursos se puedan usar de manera óptima en todo momento; si el programador carece de este conocimiento, el rendimiento se ve afectado.

La programación en lenguaje ensamblador puro es bastante rara en estos días

La memoria y la potencia informática son tan económicas hoy en día que el esfuerzo principalmente vale la pena para optimizar sistemas extremadamente críticos en cuanto al tiempo, por ejemplo, para ganar fracciones de segundo en cálculos científicos. Además, el lenguaje se sigue utilizando a menudo con fines didácticos, precisamente porque está muy ligado a la arquitectura del procesador y así, por ejemplo, aclara cómo funcionan los procesadores.

En general, el lenguaje ensamblador parece muy limitado y engorroso para los estándares actuales. Por ejemplo, casi se garantiza que los programas serán más largos que el código comparable en un lenguaje de alto nivel, porque las operaciones complejas no forman parte del conjunto de instrucciones y, por lo

tanto, debe reprogramarlas usted mismo. Además, los programas largos en lenguaje ensamblador en particular son difíciles de mantener porque el código minimalista a menudo no es fácil de entender.

```
Hola mundo en ensamblador.
SYS_SALIDA equ 1
section .data
    msg db "Hola, Mundo!!!",0x0a
    len equ $ - msg ;longitud de msg
section .text
  global _start ;para el linker
_start: ;marca la entrada
    mov eax, 4 ;llamada al sistema (sys_write)
    mov ebx, 1 ;descripción de archivo (stdout)
    mov ecx, msg ;msg a escribir
    mov edx, len ;longitud del mensaje
    int 0x80 ;llama al sistema de interrupciones
fin: mov eax, SYS_SALIDA ;llamada al sistema (sys_exit)
    int 0x80
```

Las directivas son comandos que interpreta el programa de ensamblado, este programa se encuentra constituido por las directivas que se mencionan a continuación.

Segmentos de memoria

En el lenguaje ensamblador es necesario especificar que instrucciones se guardaran en la memoria correspondiente a los datos y que otros se almacenaran en la memoria de código, para esto se utilizan los segmentos. En NASM los segmentos se especifican mediante la directiva section, de esta forma:

- section .data
    - Define el grupo de declaraciones inicializadas, ubicadas en el segmento de datos de la memoria principal.
- section .text
    - Define el grupo de instrucciones a ser ejecutadas, ubicadas en el segmento de código de la memoria principal.
- section .bss
    - Define el grupo de declaraciones no inicializadas, ubicadas de forma adyacente al segmento de datos de la memoria principal. bss es una abreviatura de ’block started by simbol’

Etiquetas

Su función es facilitar al programador la tarea de hacer referencia a una dirección de memoria, ya sea del segmento de datos o de código, mediante un símbolo o nombre. De esta manera se tienen dos tipos de etiquetas:

Para hacer referencia a una posición de código dentro del programa.

```
SYS_SALIDA equ 1
section .data
    msg db "Hola, Mundo!!!",0x0a
    len equ $ - msg ;longitud de msg
section .text
  global _start ;para el linker
_start: ;marca la entrada
    mov eax, 4 ;llamada al sistema (sys_write)
    mov ebx, 1 ;descripción de archivo (stdout)
    mov ecx, msg ;msg a escribir
    mov edx, len ;longitud del mensaje
    int 0x80 ;llama al sistema de interrupciones
fin: mov eax, SYS_SALIDA ;llamada al sistema (sys_exit)
```

```
    int 0x80
```

Para hacer referencia a las variables y constantes del programa.

```
SYS_SALIDA equ 1
section .data
    msg db "Hola, Mundo!!!",0x0a
    len equ $ - msg ;longitud de msg
section .text
  global _start ;para el linker
_start: ;marca la entrada
    mov eax, 4 ;llamada al sistema (sys_write)
    mov ebx, 1 ;descripción de archivo (stdout)
    mov ecx, msg ;msg a escribir
    mov edx, len ;longitud del mensaje
    int 0x80 ;llama al sistema de interrupciones
fin: mov eax, SYS_SALIDA ;llamada al sistema (sys_exit)
    int 0x80
```

La etiqueta _start: dentro del segmento de código es una etiqueta especial y sirve para indicarle al ensamblador donde es que comienza el programa, por lo que es necesaria la directiva global _start para que esta etiqueta presente un alcance externo; es similar a la función que cumple el main en los lenguajes de alto nivel.

Definición de datos

Las directivas usadas para especificar los datos del programa, se declaran mediante:

· Un nombre, el cual sigue las mismas reglas que la etiqueta de una posición de código con la excepción del uso del carácter dos puntos (:).

· Una instrucción de asignación de espacio en memoria, por ejemplo: DB (Define Byte): Se utiliza para asignar un byte (8 bits) de almacenamiento a cada uno de los datos declarados con esta directiva. En la mayoría de los lenguajes de programación de alto nivel, esta misma cantidad de memoria es utilizada para

almacenar los tipos de datos char, debido a que los caracteres en su representación decimal solo llegan hasta el 255 y el valor o valores a almacenar en la memoria separados por comas.

```
SYS_SALIDA equ 1
section .data
    msg db "Hola, Mundo!!!",0x0a
    len equ $ - msg ;longitud de msg
section .text
  global _start ;para el linker
_start: ;marca la entrada
    mov eax, 4 ;llamada al sistema (sys_write)
    mov ebx, 1 ;descripción de archivo (stdout)
    mov ecx, msg ;msg a escribir
    mov edx, len ;longitud del mensaje
    int 0x80 ;llama al sistema de interrupciones
fin: mov eax, SYS_SALIDA ;llamada al sistema (sys_exit)
    int 0x80
```

En combinación a la declaración de constantes, NASM permite realizar un par de pseudo operaciones para el cálculo de direcciones de memoria mediante dos constantes especiales $ y $$.

$: Hace referencia a la dirección de memoria actual.

```
 letrero db “texto”
 tamanio EQU $- letrero
 ;al restar la dirección de memoria ubicada al final
 ;del mensaje “texto” ($) y la dirección de memoria donde este
 ;mismo comienza mediante su nombre ‘letrero’, la variable
 ;tamanio almacena la longitud del contenido de la variable
 ;’letrero’ en bytes (5 bytes)
```

Interrupciones

En ensamblador para realizar cualquier acción con el hardware , ya sea imprimir en pantalla o leer datos desde el teclado se necesita la intervención del sistema operativo ,al menos en los sistemas modernos asi ocurre, para lograr esto es necesario invocar la acción del sistema con un valor de interrupción el cual variara dependiendo del sistema operativo donde el programa se ejecutara, además de esto los datos que se le pasaran serán distintos, por lo que este ejemplo funciona en el sistema operativo Gnu/Linux, pero no en cualquier otro como por ejemplo Windows.

```
SYS_SALIDA equ 1
section .data
    msg db "Hola, Mundo!!!",0x0a
    len equ $ - msg ;longitud de msg
section .text
  global _start ;para el linker
_start: ;marca la entrada
    mov eax, 4 ;llamada al sistema (sys_write)
    mov ebx, 1 ;descripción de archivo (stdout)
    mov ecx, msg ;msg a escribir
    mov edx, len ;longitud del mensaje
    int 0x80 ;llama al sistema de interrupciones
fin: mov eax, SYS_SALIDA ;llamada al sistema (sys_exit)
    int 0x80
```

Para finalizar ensamblaremos, ligaremos y ejecutaremos el programa.

```
nasm -f elf hola_mundo.asm
#crea el archivo hola_mundo.o
ld -m -o elf_i386 hola_mundo hola_mundo.o
#crea el archivo ejecutable hola_mundo
./hola_mundo
#ejecuta nuestro programa
```