El Software

3 de mayo de 2018

Índice

El Software

En esta parte de la unidad, **El Software**, nos interesa conocer el proceso de desarrollo de software, desde el punto de vista de la organización de computadoras. Explicaremos cómo se llega desde un programa, en un lenguaje de alto o bajo nivel, a obtener una sucesión de instrucciones de máquina para un procesador.

Lenguajes de bajo nivel

Lenguaje de máquina o código máquina

Hemos visto un conjunto de instrucciones y convenciones sobre cómo se utilizan los datos en el MCBE, que es el llamado **lenguaje de máquina** del MCBE. En este lenguaje, las operaciones y los datos se escriben como secuencias de dígitos binarios.

Por supuesto, escribir un programa para el MCBE y **depurarlo**, es decir, identificar y corregir sus errores, es una tarea muy dificultosa, porque los códigos de operación, las direcciones y los datos, fácilmente terminan confundiéndonos. Para facilitar la programación, se ha definido un lenguaje alternativo llamado el **ensamblador** del MCBE.

Lenguaje ensamblador

Cuando escribimos un programa en el lenguaje **ensamblador** del MCBE, las instrucciones se corresponden una a una con las del programa en lenguaje de máquina. Pero en el lenguaje ensamblador del MCBE:

- En lugar de códigos de tres bits usamos unas abreviaturas un poco más significativas (llamadas los **mnemónicos** de las instrucciones).

- En lugar de direcciones de cinco bits para los datos, usamos unos nombres simbólicos (**rótulos o etiquetas**) que hacen referencia a esas direcciones.
- Para las instrucciones de salto, en lugar de desplazamientos, también usamos rótulos o etiquetas para indicar la instrucción del programa adonde deseamos saltar.

Cada CPU del mundo real tiene su propio lenguaje de máquina, y aunque mucho más poderosos y de instrucciones más complejas, se parecen bastante, en líneas generales, al lenguaje de máquina del MCBE. Igual que ocurre con el lenguaje de máquina, cada CPU del mundo real tiene su propio lenguaje ensamblador, basado en los mismos principios que el que mostramos aquí.

El lenguaje de máquina de cualquier CPU, y su lenguaje ensamblador (o *Assembler*), son llamados en general **lenguajes de bajo nivel**.

Mnemónicos

Los **mnemónicos** o nombres simbólicos de las instrucciones se basan en los nombres en inglés de las operaciones correspondientes. Disponemos de los mnemónicos:

- LD para la operación de cargar el Acumulador con un contenido de memoria (código 010), y ST para la operación inversa (código 011).
- ADD para la operación de suma (código 100) y SUB para la resta (código 101).
- JMP y JZ para los saltos incondicional y condicional (códigos 110 y 111), respectivamente.
- HLT para la instrucción de parada (código 001) y NOP para la operación nula o no operación (código 000).

Rótulos

Cuando necesitamos hacer referencia a una dirección, como en las operaciones de transferencia o en las aritméticas, el ensamblador nos permite independizarnos del valor de esa dirección y simplemente indicar un **nombre simbólico o rótulo** para esa dirección. Así, un rótulo equivale en lenguaje ensamblador a la **dirección de un dato**.

Para que el programa quede completo, ese nombre simbólico debe aparecer en algún lugar del programa, al principio de la instrucción, y separado por un carácter ":" del resto de la línea.

Ejemplo

Dirección	Instrucción	Rótulo	Mnemónico	Argumento
00000	01000111		LD	CANT
00001	11100100	SIGUE:	JZ	FIN
00010	01111111		ST	OUT
00011	10100110		SUB	UNO
00100	11011101		JMP	SIGUE
00101	00100000	FIN:	HLT	
00110	00000001	UNO:	1	
00111	00000011	CANT:	3	

En este ejemplo, SIGUE, FIN, UNO y CANT son rótulos. El rótulo CANT, por ejemplo, nos permite referirnos en la primera instrucción, LD CANT, a un dato declarado más adelante con ese nombre. Del mismo modo, cuando la instrucción es de salto, podemos hacer referencia a la posición de memoria donde se hará el salto usando un rótulo, como en la quinta instrucción, JMP SIGUE.

Rótulos en instrucciones de salto

Es importante recordar que, de todas maneras, en la traducción de ensamblador a lenguaje de máquina **para las instrucciones de salto**, el rótulo se sustituye por un **desplazamiento**, y no por una dirección.

Ejemplo

- En el ejemplo existe un rótulo SIGUE que identifica a la instrucción en la posición 1. La instrucción del ejemplo JMP SIGUE, al ser ejecutada, deriva el control a la instrucción 1. Es decir, almacena un 1 en el registro PC, para que la siguiente iteración del ciclo de instrucción ejecute la instrucción en la dirección 1 de la memoria. Sin embargo, el argumento para la instrucción JMP **no vale 1** sino **-3**, como podemos corroborar en la columna "Instrucción" de la tabla.

Rótulos predefinidos

Los rótulos IN y OUT vienen predefinidos en el lenguaje ensamblador de MCBE y corresponden a las posiciones de memoria 30 (para entrada) y 31 (para salida) respectivamente.

Ejemplo

La instrucción en línea 2, ST OUT, almacena el contenido del acumulador en la posición 31, lo que equivale a escribir ese contenido en la salida del MCBE.

Traductores

Uno puede pensar en un programa cualquiera como si se tratara de una máquina, cuyo funcionamiento es, en principio, desconocido. Todo lo que vemos es que, si introducimos ciertos datos, de alguna forma esta "máquina" devolverá un resultado.

Si pensamos en una clase especial de estos programas, donde los datos que ingresan son a su vez un programa, y donde la salida devuelta por la máquina es, a su vez, un programa, entonces esa clase especial de programas son los **traductores**.

Ensambladores

Como hemos dicho anteriormente, una CPU como el MCBE sólo sabe ejecutar instrucciones de código máquina expresadas con unos y ceros. Cuando vimos el lenguaje ensamblador del MCBE lo propusimos simplemente como una forma de abreviar las instrucciones de máquina, o como una forma de facilitar la escritura, porque los mnemónicos y rótulos eran más fáciles de memorizar y de leer que las instrucciones con unos y ceros.

Sin embargo, un programa escrito en ensamblador del MCBE podría ser traducido automáticamente, por un traductor, a código de máquina MCBE, ahorrándonos mucho trabajo y errores.

Esta clase de traductores, que reciben un programa en lenguaje ensamblador y devuelven un programa en código de máquina, son los llamados **ensambladores** o **assemblers**.

Ensamblador x86

Cada CPU tiene su propio lenguaje ensamblador, y existen programas traductores (ensambladores) para cada una de ellas. Por ejemplo, la familia de procesadores de Intel para computadoras personales comparte el mismo ISA, o arquitectura y conjunto de instrucciones. Cualquiera de estos procesadores puede ser programado usando un ensamblador para la familia **x86**.

Según la tradición, el primer programa que uno debe intentar escribir cuando comienza a aprender un lenguaje de programación nuevo es "Hola mundo". Es un programa que simplemente escribe esas palabras por pantalla. Aquí mostramos el clásico ejemplo de "Hola mundo" en el lenguaje ensamblador de la familia x86.

```
.globl  _start
.text               # seccion de codigo
_start:
        movl    $len, %edx  # carga parametros longitud
        movl    $msg, %ecx  # y direccion del mensaje
        movl    $1, %ebx        # parametro 1: stdout
```

```
        movl    $4, %eax        # servicio 4: write
        int     $0x80       # syscall

        movl    $0, %ebx        # retorna 0
        movl    $1, %eax        # servicio 1: retorno de llamada
        int     $0x80       # syscall
.data               # seccion de datos
msg:
        .ascii   "Hola, mundo!\n"
        len =   . - msg     # longitud del mensaje
```

Los procesadores de la familia x86 se encuentran en casi todas las computadoras personales y notebooks.

Ensamblador ARM

Por supuesto, los procesadores de familias diferentes tienen conjuntos de instrucciones diferentes. Así, un lenguaje y un programa ensamblador están ligados a un procesador determinado. El código máquina producido por un ensamblador no puede ser trasladado sin cambios a otro procesador que no sea aquel para el cual fue ensamblado. Las instrucciones de máquina tendrán sentidos completamente diferentes para uno y otro.

Por eso, el código máquina producido por un ensamblador para x86 no puede ser trasladado directamente a una computadora basada en un procesador como, por ejemplo, ARM; sino que el programa original, en ensamblador, debería ser **portado** o traducido al ensamblador propio de ARM, por un programador, y luego ensamblado con un ensamblador para ARM.

```
.global main
main:
    @ Guarda la direccion de retorno lr
    @ mas 8 bytes para alineacion
    push    {ip, lr}
    @ Carga la direccion de la cadena y llama syscall
    ldr     r0, =hola
    bl      printf
    @ Retorna 0
    mov     r0, #0
    @ Desapila el registro ip y guarda
    @ el siguiente valor desapilado en el pc
    pop     {ip, pc}
hola:
    .asciz "Hola, mundo!\n"
```

El ARM es un procesador que suele encontrarse en plataformas móviles como *tablets* o teléfonos celulares, porque ha sido diseñado para minimizar el consumo de energía, una característica que lo hace ideal para construir esos productos portátiles. Su arquitectura, y por lo tanto, su conjunto de instrucciones, están basados en esos principios de diseño.

Ensamblador PowerPC

Lo mismo ocurre con otras familias de procesadores como el PowerPC, un procesador que fue utilizado para algunas generaciones de consolas de juegos, como la PlayStation 3.

```
.data             # seccion de variables
msg:
    .string "Hola, mundo!\n"
    len = . - msg     # longitud de cadena
.text             # seccion de codigo
    .global _start
_start:
    li  0,4        # syscall sys_write
    li  3,1    # 1er arg: desc archivo (stdout)
               # 2do arg: puntero a mensaje
    lis     4,msg@ha   # carga 16b mas altos de &msg
    addi    4,4, msg@l # carga 16b mas bajos de &msg
    li  5,len      # 3er arg: longitud de mensaje
    sc         # llamada al kernel
#
    li  0,1    # syscall sys_exit
    li  3,1    # 1er arg: exit code
    sc         # llamada al kernel
```

Lenguajes de programación

Lenguajes de bajo nivel

Como vemos, tanto el lenguaje de máquina como el ensamblador o **Assembler** son lenguajes **orientados a la máquina**. Ofrecen control total sobre lo que puede hacerse con un procesador o con el sistema construido alrededor de ese procesador. Por este motivo son elegidos para proyectos de software donde se necesita dialogar estrechamente con el hardware, como ocurre con los sistemas operativos.

Sin embargo, como están ligados a un procesador determinado, requieren conocimiento profundo de dicho procesador y resultan poco **portables**. Escribir un programa para resolver un problema complejo en un lenguaje de bajo nivel suele ser muy costoso en tiempo y esfuerzo.

Lenguajes de alto nivel

Otros lenguajes, los de **alto nivel**, ocultan al usuario los detalles de la arquitectura de las computadoras y le facilitan la programación de problemas de software complejos. Son más **orientados al problema**, lo que quiere decir que nos aíslan de cómo funcionan los procesadores o de cómo se escriben las instrucciones de máquina, y nos permiten especificar las operaciones que necesitamos para resolver nuestro problema en forma más parecida al lenguaje natural, matemático, o humano.

Una ventaja adicional de los lenguajes de alto nivel es que resultan más portables, y su **depuración** (el proceso de corregir errores de programación) es normalmente más fácil.

Niveles de lenguajes

Se han diseñado muchísimos lenguajes de programación. Cada uno de ellos es más apto para alguna clase de tareas de programación y cada uno tiene sus aplicaciones.

- Entre estos lenguajes, que podemos organizar en una jerarquía, encontramos los de bajo nivel u orientados a la máquina, los de alto nivel, u orientados al problema, y algunos en una zona intermedia.
- Pero también pueden clasificarse por el **paradigma de programación** al cual pertenecen.
- Además, algunos son habitualmente lenguajes **compilables**, y otros, **interpretables**.

Paradigmas de programación

La programación en lenguajes de alto nivel puede adoptar varias formas. Existen diferentes modos de diseñar un lenguaje, y varios modos de trabajar para obtener los resultados que necesita el programador. Esos modos de pensar o trabajar se llaman **paradigmas de lenguajes de programación**.

Hay al menos cuatro paradigmas reconocidos, que son, aproximadamente en orden histórico de aparición, **imperativo** o procedural, **lógico o declarativo**, **funcional** y **orientado a objetos**. Los paradigmas lógico y funcional son los más asociados a la disciplina de la Inteligencia Artificial.

Paradigma imperativo o procedural

Bajo el paradigma imperativo, los programas consisten en una sucesión de instrucciones o comandos, como si el programador diera órdenes a alguien que las cumpliera. El ejemplo en lenguaje **C** explica cuáles son las órdenes que deben ejecutarse, una por una.

```
int factorial(int n)
{
        int f = 1;
        while (n > 1) {
                f *= n;
                n--;
        }
        return f;
}
```

Paradigma lógico o declarativo

El lenguaje Prolog representa el paradigma lógico y con frecuencia constituye el corazón de los sistemas de Inteligencia Artificial que realizan **razonamiento**.

La definición de **factorial** en lenguaje Prolog que mostramos se compone de un hecho y dos reglas. El hecho consiste en que el **factorial** de 0 vale 1. La primera regla expresa que el factorial de un número **N** se calcula como el factorial de **N-1** multiplicado por N. Es una definición **recursiva** porque la definición de la regla se utiliza a sí misma.

```
factorial(0,X):- X=1.
factorial(N,X):- N1=N-1, factorial(N1,X1), X=X1*N.
factorial(N):- factorial(N,X), write(X).
```

El usuario de este programa puede usarlo de dos maneras. Podría preguntar el valor del factorial de un número N, o consultar si es cierto que el factorial de N es otro número dado Y.

Paradigma funcional

En el lenguaje Lisp, perteneciente al paradigma funcional, una función es un enunciado entre paréntesis que puede contener a otras funciones. En particular la definición de **factorial** presentada aquí contiene a su vez una invocación de la misma función, volviéndola una función **recursiva**.

```
(defun factorial (n)
  (if (= n 0)
      1
      (* n (factorial (- n 1))) ) )
```

El lenguaje Lisp utiliza notación prefija para los operadores.

Orientación a objetos

En un lenguaje **orientado a objetos**, definimos una **clase** que funciona como un molde para crear múltiples instancias de objetos que se parecen entre sí, ya que tienen los mismos datos que los componen y la misma funcionalidad. Los objetos creados se comunican entre sí por **mensajes**, disparando **métodos** o conductas de otros objetos.

```
class Combinatoria():
    def factorial(self,n):
        num = 1
        while n > 1:
            num = num * n
            n = n - 1
        return num

c = Combinatoria()
a = c.factorial(3)
print a
```

En el ejemplo de programación orientada a objetos en Python, definimos una clase **Combinatoria** que producirá objetos con la conducta **factorial**. El programa crea un objeto, instancia de la clase Combinatoria, llamado **c**, al cual se le envía el mensaje **factorial**, que dispara la conducta correspondiente especificada en el método del mismo nombre. Finalmente se imprime su valor.

Compiladores e intérpretes

Los traductores de lenguajes de alto nivel pueden funcionar de dos maneras: o bien producen una versión en código máquina del programa fuente (**compiladores**) o bien analizan instrucción por instrucción del programa fuente, y además de generar una traducción a código máquina de cada línea, la ejecutan (**intérpretes**).

Luego de la compilación, el programa en código máquina obtenido puede ser ejecutado muchas veces. En cambio, el programa interpretado debe ser traducido cada vez que se ejecute.

Velocidad de ejecución

- Una ventaja comparativa de la compilación respecto de la interpretación es la mayor velocidad de ejecución. Al separar las fases de traducción y ejecución, un compilador alcanza la máxima velocidad de ejecución posible en un procesador dado.
- Por el contrario, un intérprete alterna las fases de traducción y ejecución, por lo cual la ejecución completa del mismo programa tardará algo más de tiempo.

Portabilidad

- El código interpretado presenta la ventaja de ser directamente portable. Dos plataformas diferentes podrán ejecutar el mismo programa interpretable, siempre que cuenten con intérpretes para el mismo lenguaje.
- Por el contrario, un programa compilado está en el código de máquina de alguna arquitectura específica, así que no será compatible con otras.

Ciclo de compilación

Terminología

- Cuando utilizamos un **compilador** para obtener un programa **ejecutable**, el programa que nosotros escribimos, en algún lenguaje, se llama **programa fuente**, y estará generalmente contenido en algún **archivo fuente**.
- El resultado de la traducción será un archivo llamado **objeto** conteniendo las instrucciones de código máquina equivalentes.
  - Sin embargo, este archivo objeto puede no estar completo, ya que el programador puede hacer uso de rutinas o funciones que vienen provistas con el sistema, y no necesita especificar cómo se realizan esas funciones.
  - Al no aparecer en el programa fuente, esas funciones no aparecerán en el archivo objeto.
  - Por ejemplo, cualquier programa "Hola Mundo", en cualquier lenguaje, imprime en pantalla un mensaje; pero la acción de imprimir algo en pantalla no es trivial ni sencilla, y la explicación de cómo se hace esta acción **no está contenida en esos programas**. En su lugar, existe una llamada a una función de impresión cuya definición reside en algún otro lugar.
- Ese otro lugar donde están definidas funciones disponibles para el programador son las **bibliotecas**. Las bibliotecas son archivos conteniendo grupos o familias de funciones.
- El proceso de **vinculación**, que es posterior a la traducción, debe buscar en esas bibliotecas la definición de las funciones faltantes en el archivo objeto.
- Si la vinculación resulta exitosa, el resultado final es un programa **ejecutable**.

Fases del ciclo de compilación

- El desarrollador que necesita producir un archivo ejecutable utilizará varios programas de sistema como editores, traductores, vinculadores, etc.
- En algún momento anterior, alguien habrá creado una biblioteca de funciones para uso futuro. Esa biblioteca consiste en versiones objeto de varias funciones, compiladas, y reunidas con un programa bibliotecario, en un archivo.

- Esa biblioteca es consultada por el vinculador para completar las referencias pendientes del archivo objeto.
- En resumen, la primera fase del ciclo de compilación es necesariamente la **edición del programa fuente**.
- Luego, la traducción para generar un **archivo objeto** con referencias pendientes.
- Luego, la vinculación con **bibliotecas** para resolver esas referencias pendientes.
- El resultado final del ciclo de compilación es un **ejecutable**.

Entornos de desarrollo o IDE

Muchos desarrolladores utilizan algún **ambiente integrado de desarrollo (IDE)**, que es un programa que actúa como intermediario entre el usuario y los componentes del ciclo de compilación (editor, compilador, vinculador, bibliotecas).

El entorno integrado facilita el trabajo al desarrollador automatizando el proceso. Sin embargo, aunque el ambiente integrado lo oculte, el sistema de desarrollo **sigue trabajando como se ha descrito**, con fases separadas y sucesivas para la edición, traducción, vinculación y ejecución de los programas.