LABORATORIO DE PROGRAMACIÓN EN LENGUAJE ENSAMBLADOR x86-16bits

Conversión binario-ASCII

Objetivo

El objetivo de esta práctica es la programación del código necesario para convertir un número entero binario almacenado en memoria a la cadena ASCII correspondiente a su codificación en una variedad de bases (2, 10 ó 16).

El cambio de base se realiza mediante una sucesión de divisiones. Dado que los enteros a convertir pueden tener un tamaño arbitrariamente grande, utilizaremos un algoritmo de división para números enteros de tamaño arbitrario.

Algoritmo de cambio de base

Para convertir un número representado en base b_1 (A_{b1}) a su representación en base b_2 (A_{b2}), existen varios algoritmos. El más común consiste en dividir el número en base b_1 entre b_2, tomar el cociente y repetir sucesivamente hasta que el cociente sea cero[1]. El número en base b_2 es el formado por los restos, leídos del último (dígito de mayor peso) al primero (dígito de menor peso).

$$A_{b_1} = a_{n-1}b_1^{n-1} + \cdots + a_2 b_1^2 + a_1 b_1^1 + a_0 b_1^0$$

$$\begin{array}{cccc} Q_{0b_1} & Q_{1b_1} & Q_{2b_1} & 0 \\ b_2\overline{)A_{b_1}} & b_2\overline{)Q_{0b_1}} & b_2\overline{)Q_{1b_1}} & b_2\overline{)Q_{n-2b_1}} \\ r_{0b_2} & r_{1b_2} & r_{2b_2} & r_{n-1b_2} \end{array}$$

$$A_{b_2} = r_{n-1}b_2^{n-1} + \cdots + r_2 b_2^2 + r_1 b_2^1 + r_0 b_2^0$$

Figura 1. Algoritmo de cambio de base.

División de números arbitrariamente grandes

Exigimos que el número arbitrariamente grande (dividendo) tenga un tamaño múltiplo (*m*) de la palabra que maneje el *hardware* (palabra de *n* dígitos). Asumimos que el divisor tiene tamaño palabra (*n*). En consecuencia, el resto tiene tamaño palabra (*n*) mientras que cociente será un número arbitrariamente grande de *m* palabras.

Si suponemos que la representación es binaria, tendremos:

$$A = (A_{m-1}, A_{m-2}, \cdots, A_1, A_0) \qquad A = \sum_{j=0}^{mn-1} a_j \cdot 2^j$$

$$A_i = (a_{in+n-1}, a_{in+n-2}, \cdots, a_{in+1}, a_{in}) \qquad A_i = \sum_{k=0}^{n-1} a_{in+k} \cdot 2^{in+k}$$

$$A = \sum_{i=0}^{m-1} A_i = \sum_{i=0}^{m-1}\sum_{k=0}^{n-1} a_{in+k} \cdot 2^{in+k} = \sum_{i=0}^{m-1}\left(\sum_{k=0}^{n-1} a_{in+k} \cdot 2^k\right) \cdot 2^{in} = \sum_{i=0}^{m-1} A(i) \cdot 2^{in} \qquad A(i) = \sum_{k=0}^{n-1} a_{in+k} \cdot 2^k$$

Figura 2. Número binario arbitrariamente grande.

[1] A veces se dice que se repiten las divisiones hasta que el cociente sea menor que b_2. En ese caso, el dígito de mayor peso es precisamente dicho cociente.

La operación se realizará tomando dividendos parciales de tamaño doble palabra. En la primera iteración, la parte alta de la doble palabra será cero y la parte baja será la palabra de mayor peso del dividendo (número de tamaño arbitrario). El cociente obtenido será la palabra de mayor peso del cociente final mientas que con el resto obtenido formamos un dividendo parcial nuevo haciendo que el resto sea la palabra de mayor peso y la siguiente palabra del dividendo original la palabra de menor peso. El dividendo parcial así formado se dividirá en la siguiente iteración. El algoritmo se repite hasta que hayamos "bajado" todas las palabras que conforman el dividendo original. El último cociente será la palabra de menor peso del cociente final y el último resto será el resto final de la operación.

Podemos expresar la doble palabra utilizando la notación de *Intel* según la cual separamos la parte alta de la parte baja con dos puntos (':'), tal y como se usa en la operación de multiplicación para el resultado de 32 bits o en la operación de división para el dividendo de 32 bits (DX:AX). Suponiendo esto y que el número arbitrariamente grande se compone de *m* palabras *A(i)* de *n* bits cada una, podemos escribir el algoritmo de división en pseudocódigo de la siguiente manera:

```
DX = 0

for (i = m-1; i = 0; i--)
{
   AX = A(i)
   DX:AX / divisor  ;DX->resto parcial; AX->cociente parcial
   Q(i) = AX
}
                    ;DX->resto final; Q->cociente final
```

Figura 3. Pseudocódigo del algoritmo de división.

Comparación de números arbitrariamente grandes

El algoritmo de cambio de base requiere de una comparación en cada iteración buscando o bien un cero o bien un número menor que el divisor, dependiendo de la implementación. Dado que el tamaño del cociente es también un número arbitrariamente grande (*m* palabras de *n* dígitos en general, o bits en binario), se necesita realizar una comparación para números arbitrariamente grandes. Sin embargo, ya que el valor a comparar es pequeño, podemos escribirlo en un registro de tamaño palabra (para un máquina *Intel*, el registro acumulador AX) y luego hacer una extensión.

La comparación consiste en restar palabra a palabra desde el menor peso al mayor sin salvar el resultado pero salvando el registro de estado de cada operación para que se evalúe al comienzo de la siguiente. Finalmente, el último registro de estado se devuelve como valor en un registro (para una máquina *Intel*, el registro acumulador AX). Hay que tener en cuenta que cualquier resta parcial distinta de cero provoca que el *flag* de cero (ZF) del estado final indique "no nulo".

```
nonulo = 0

DX = A(0) - AX
pila <- estado

for (i = 1; i < m-1; i++)
{
   estado <- pila(estado)
   if (ZF=0) then nonulo = 1
   DX = A(i) - 0                  ;el '0' es una extensión implícita
   ff(CF = 1 then
   {
      DX = DX - 1
   }
   pila <- estado
}
AX = pila(estado)
if (nonulo=1) AX(ZF)=0
```

Figura 4. Pseudocódigo del algoritmo de comparación.

Creación de un módulo de librería

Las dos funciones necesarias para trabajar con números arbitrariamente grandes, se van a escribir en un módulo aparte que invocaremos desde el módulo principal como funciones de librería. Para ello, basta con declarar los procedimientos como públicos con la directiva PUBLIC. Para usarlos desde otro módulo, se declararan como externos con la directiva EXTRN y se invocarán como se hace siempre.

El módulo con el código de las funciones de librería se muestra a continuación. La función que realiza la división de números arbitrariamente grandes se llama ***bigdiv*** mientras que la función que realiza la comparación de un número de tamaño arbitrariamente grande con un valor pasado en AX es ***bigAXcmp***.

```
;-------------------------------------------------
; bigsizop.asm
;-------------------------------------------------
; - Declaración de funciones que realizan
;   operaciones sobre números de tamaño arbitrario
;-------------------------------------------------

public bigAXcmp
public bigdiv

dosseg
.model small

.code

;-------------------------------------------------
; bigAXcmp   Compara un número de
;            tamaño arbitrario con AX
;-------------------------------------------------
; Argumentos:
;      1.- Val. (word) = tamaño en bytes
;      2.- Ref. (near) = operando1
;      3.- AX = operando2
;      Nota: AX contiene un número sin signo
;-------------------------------------------------
; Retorno:
;      AX = registro de estado
;-------------------------------------------------

bigAXcmp proc
   push bp
   mov bp, sp

   push cx          ;salvo los registros que voy a usar
   push dx
   push si

   mov cx, [bp+ 4]  ;copio el tamaño en CX
   mov si, [bp+ 6]  ;copio la dirección del operando1

   mov dx, [si]
   sub dx, ax
   pushf

   add si, 2
   sub cx, 2

   xor al, al       ;indico en AL si alguna resta no es nula

%22restar:
   popf
   jz %22seguir
   mov al, 1        ;si ZF=0 lo indico en AL
%22seguir:
   mov dx, [si]
   sbb dx, 0
   pushf
```

```
   add si, 2
   sub cx, 2
   jnz %22restar

   pop cx                  ;recupero el estado en CX
   cmp al, 1
   jnz %22salir
   and cx, 0FFBFh          ;si alguna resta no fue nula ahora ZF=0

%22salir:
   mov ax, cx              ;devuelve en AX el último registro de estado

   pop si                  ;recupero los registros que he usado
   pop dx
   pop cx

   mov sp, bp
   pop bp
   ret 4
bigAXcmp endp


;-------------------------------------------------
; bigdiv     Divide un número de
;            tamaño arbitrario entre un
;            divisor de tamaño word
;-------------------------------------------------
; Argumentos:
;      1.- Val. (word) = tamaño en bytes
;      2.- Ref. (near) = dividendo
;      3.- Val. (word) = divisor
;      4.- Ref. (near) = cociente
;      5.- Ref. (near) = resto
;      Nota: el cociente está inicialmente a 0
;      Nota: el tamaño del cociente es igual que
;            el del dividendo
;      Nota: el tamaño del resto es igual que el
;            del divisor
;-------------------------------------------------
; Retorno:
;      el resultado en el cociente y resto
;-------------------------------------------------

bigdiv proc
   push bp
   mov bp, sp

   push ax          ;salvo los registros que voy a usar
   push bx
   push cx
   push dx
   push si
   push di

   sub sp, 2        ;declaro una variable local de tamaño word

   mov cx, [bp+ 4]  ;copio el tamaño en CX
   mov si, [bp+ 6]  ;copio la dirección del dividendo
   mov bx, [bp+ 8]  ;copio el valor del divisor
   mov di, [bp+10]  ;copio la dirección del cociente
   mov dx, [bp+12]  ;copio la dirección del resto
   mov [bp-2], dx   ;salvo la dirección del resto en la v. local

   add si, cx       ;apunto a la palabra de mayor peso
   add di, cx

   mov dx, 0

%4bucle:
   mov ax, [si-2]
```

```
   div bx           ;operación de tamaño word (DX:AX(i)/BX)

   mov [di-2], ax   ;salvo el cociente

   sub si, 2
   sub di, 2
   sub cx, 2
   jnz %4bucle

   mov bx, [bp-2]   ;recupero la dirección del resto
   mov [bx], dx     ;salvo el resto

   add sp, 2        ;soslayo la variable local

   pop di           ;recupero los registros que he usado
   pop si
   pop dx
   pop cx
   pop bx
   pop ax

   mov sp, bp
   pop bp
   ret 10
bigdiv endp

end
```

Para crear un ejecutable a partir de varios módulos, generamos los ficheros .obj compilando cada módulo por separado (`masm modulo1.asm`; `masm modulo2.asm`…) y luego enlazamos el módulo principal con el módulo librería (`link principal.obj libreria.obj`).

Paso de argumentos

Recordamos que el paso de argumentos se realiza en orden inverso. Concretamente, para la función ***bigdiv*** tendremos:

```
;DIVISIÓN
;paso argumentos a la función 'bigdiv' (orden inverso)

mov bx, divisor
mov cx, tamano

lea dx, Rparcial
push dx                    ;argumento 5 = dirección del resto

lea dx, Qparcial
push dx                    ;argumento 4 = dirección del cociente
push bx                    ;argumento 3 = divisor
lea dx, Dividendo
push dx                    ;argumento 2 = dirección del dividendo
push cx                    ;argumento 1 = tamaño del entero

call bigdiv
```

Prácticas

A) Escribir un procedimiento que convierta de binario a ASCII.

Utilizando las funciones de librería mostradas anteriormente y aplicando el algoritmo de cambio de base, escribir un procedimiento que genere la cadena ASCII correspondiente a la conversión a base 2, 10 ó 16 de un entero binario de tamaño arbitrario expresado en bytes.

Los argumentos que requiere la función serán los siguientes:

```
;-------------------------------------------------
; entero2ascii      Convierte un entero sin signo de
;                   tamaño arbitrario a cadena ASCII
;-------------------------------------------------
; Argumentos:
;      1.- Val. (word) - tamaño en bytes
;      2.- Ref. (near) - entero binario
;      3.- Ref. (near) - cadena ASCII
;      4.- Val. (word) - base (2, 10 o 16)
;-------------------------------------------------
; Retorno:
;      en la cadena ASCII
;      AX = número de digitos ASCII
;-------------------------------------------------
```

La figura siguiente sugiere un diagrama de flujo que permite implementar el procedimiento pedido. Puesto que los dígitos de salida se obtienen del de mayor peso al de menor peso, se propone salvarlos en la pila para recuperarlos posteriormente en su orden.

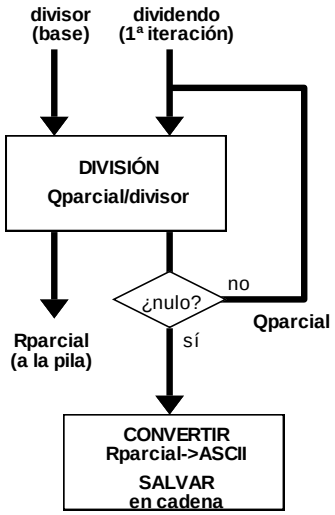

Figura 5. Diagrama de flujo propuesto para la conversión binario-ASCII.

B) Escribir un programa que convierta un número entero binario de 8 bytes a la cadena ASCII correspondiente a binario, decimal y hexadecimal.

Los tamaños de las cadenas ASCII dependen de la base a la que se conviertan. En binario, necesitamos 1 carácter por cada bit de representación del número entero. En decimal necesitamos $\log_{10} 2^n$ caracteres, es decir, $n \log_{10} 2$ que es $n \times 0.3010 \approx n/3$ caracteres. En hexadecimal necesitamos $n/4$ caracteres ($\log_{16} 2^n = \log_{16} 16^{n/4} = n/4$).