

Proyecto “Zip”

Integrantes:

- Abuzaid Karim – 35.472.429
- González Alan – 37.177.303

Herramientas:

- **GitHub** (Sistema de control colaborativo de revisión y desarrollo de software)
<https://github.com/vercryger/compressor>
- **Nasm** (versión 2.09.10)
- **GCC** (versión 2.09.10)

Aclaraciones:

El código fuente **Assembly** de las subrutinas tanto para la compresión como para la descompresión, se encontrará en los archivos ***encode.asm*** y ***decode.asm*** respectivamente. Dentro de los mismos archivos, se utilizarán subrutinas *externas* declaradas en ***library.asm***.

Instrucciones:

Una vez descomprimido el archivo, en la carpeta ***src*** se encontrará todo el código fuente necesario para ejecutar el programa.

Para correr el programa, simplemente ejecutar el *bash script*:

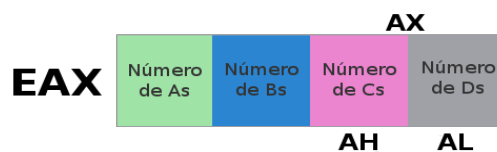
```
src$ ./run.sh
```

Dicho script, primero compilará el código **Assembly**, luego el de **C**, y por último se ejecutará el programa propiamente dicho.

Compresión

```
push dword [ebp + 8] ; pointer to cadeChar[]  
call counter  
add esp, 4
```

Como punto de partida, se procesó la cadena pasada como parámetro desde **C** (*cadeChar*), utilizando la subrutina *counter* que cuenta la cantidad de cada una de las letras (*A*, *B*, *C* y *D*) de forma tal de que en un registro de propósito general (en este caso EAX), quede almacenada dicha información:



Finalmente la información esta lista para ser utilizado por el *llamante*.

```
push dword eax  
call turnToNegative  
add esp, 4
```

A continuación se convirtieron todos los valores de EAX a negativos de la siguiente manera: Al valor más grande se le colocó el valor -1, al segundo más grande -2 y así sucesivamente con los demás números de 8 bits, es decir que el registro EAX luego de haber sido procesado por la subrutina *counter*, contendrá combinaciones de los bytes -1, -2, -3 y -4. Para el ejemplo en cuestión, el resultado en EAX será: FFFFFFFD.

```
mov [aux], eax  
  
push dword eax  
call sortbyte  
add esp, 4
```

Luego para armar la *matriz de codificación* se copió el registro en una variable de 4 bytes *aux*, y al registro EAX se lo ordenó un registro de bytes. Para el ejemplo:

cadeChar = "AAACCBBAACBBBBAADDBB"

counter(*cadeChar*) --> EAX = FFFFFFFD *aux* = FFFFFFFD
sortbyte(*eax*) --> EAX = FFFFFFFD

La subrutina *sortbyte* ordena de a bytes en orden no ascendente las frecuencias de cada una de las letras, pero ahora la cardinalidad no corresponde a la de cada una de las letras, es por esto que antes de ordenarlo se guardó en *aux* el orden original. Ahora sólo bastaría mapear cada byte de EAX con los de *aux* para

saber a *qué* letra corresponde y por ende saber qué letra tiene mayor frecuencia. Dicho cálculo se realiza desde la línea 36 hasta la línea 71. Una vez hecho esto en ECX se tendrá las posiciones de las letras a las cuales pertenecen dichas frecuencias. Por ejemplo:

<i>aux</i>			
0	1	2	3
-2	-1	-3	-4

<i>EAX</i>			
<i>Extended</i>		<i>AX</i>	
0	1	2	3
-1	-2	-3	-4

<i>ECX</i>			
<i>Extended</i>		<i>CX</i>	
0	1	2	3
1	0	2	3

Luego en base a estas posiciones calculadas en ECX, puedo saber en qué orden están en el *cadeChar[]*:

cadeChar + **0** = A
cadeChar + **1** = B
cadeChar + **2** = C
cadeChar + **3** = D

Y ahora sólo falta asignarle la correspondiente codificación a cada una de las letras, y por esto es que se ordenó anteriormente las frecuencias, por que a la letra que le pertenece la primer frecuencia, se la codificará como *01* ya que es la que más veces aparece, luego a la letra que le pertenece la segunda frecuencia se la codificará como *011* porque es la que le sigue en cuanto a apariciones, y así sucesivamente con las demás. Es decir, cada posición de las frecuencias en EAX está asociada a una codificación única:

<i>EAX</i>			
<i>Extended</i>		<i>AX</i>	
0	1	2	3
08	06	03	02
<i>01</i>	<i>011</i>	<i>0111</i>	<i>01111</i>

Finalmente guardo en *matrizCod* (de forma *aleatoria*) las codificaciones correspondientes a las letras. Primero guardo la codificación de *01111*, luego la de *0111* y así sucesivamente, con lo que obtenemos:

Dirección	Contenido	<i>cadeChar</i>
<i>matrizCod</i> + 0	000000011	A
<i>matrizCod</i> + 1	00000001	B
<i>matrizCod</i> + 2	00000111	C
<i>matrizCod</i> + 3	00001111	D

Una vez generada la matriz de codificación, lo que sigue es codificar la cadena de entrada. El pseudocódigo sería el siguiente:

```

i = 0
j = 0
AX = 1 // ¡Importante!*
BX = cadeChar[j] // obtengo la primer letra
while (BL != 0) { // no es el fin de la cadena
    BL = getCodification(BL) // obtengo la codificación en matrizCod y la almaceno en BL
    shl AX, 1 // necesario para marcar el inicio de una nueva letra

    while (BL != 0) { // hay 1's en BL para pasar a AL
        if ( isNotFull(AL) ) { // AX != 1
            shr BL, 1
            rcl AX, 1 // rotación con el carry
        } else { // AL está completo, hay que guardar el contenido en cadeZip
            cadeZip[i] = AL
            i++
            AX = 1 // ¡Importante!*
        }
    }
    j++
    BL = cadeChar[j] // obtengo la siguiente letra
}
// en AL aún pueden haber quedado bits para guardar en cadeZip
if (AX != 1) {
    while (AX == 0) {
        shl AX, 1
    }
    cadeZip[i] = AL
}

```

(*) Es necesario para saber en qué momento AL estará completo y por ende se necesitará pedir nueva letra, pues AH será igual a 1, ya que se shiftearán las codificaciones de BL en AX de derecha a izquierda. Por ejemplo:

Al comenzar el primer ciclo, la primer letra sería una A, entonces BL tendría el valor 65, en binario 1000001:

BX	
BH	BL
00000000	01000001

AX	
AH	AL
00000000	00000001

Luego se debe buscar la codificación del valor 65 y colocarlo en BL, seguido de shiftear con 0 de derecha a izquierda el registro AX:

BX	
BH	BL
00000000	00000011

AX	
AH	AL
00000000	000000010

A continuación todos los bits de BL se pasarán a AX, y esto se repetirá varias veces hasta que AX tenga el siguiente estado:

AX	
AH	AL
00000001	01101101

Es aquí cuando ya no hay mas lugar en AL y por lo tanto se debe mover su contenido al arreglo *cadeZip* y volver AL a su estado original.

En el caso en que aun queden bits en BL, luego de restaurar AX, se continuarán colocando los 1's restantes hasta agotar BL.

Descompresión

Para descomprimir la cadena zipeada, básicamente el problema se reduce en ir pasando de a bytes de *cadeZip* a un registro de propósito general, y luego ir shifteándolo de derecha a izquierda, y rotar o shiftear con carry en otro registro de 1 byte, hasta que el carry sea cero, pues en este punto lo que sigue es la codificación de otra letra, y por lo tanto lo que coloqué en el otro registro es la codificación de una letra, sólo bastaría con obtener la letra cuya codificación se corresponde con la almacenada y luego guardar esta letra (1 byte) en el arreglo *cadeChar*; y este proceso deberá repetirse hasta que se obtenga dos veces consecutivas “carry cero”, ya que será el fin de la cadena zipeada.

El pseudoalgoritmo sería el siguiente:

```
i = 0
j = 0
counter = 7 // ¡Importante!*
carry = 0
EAX = 0
BH = cadeZip[i] // obtengo el primer byte de codificación
BL = cadeZip[i++] // obtengo el segundo byte de codificación
shl BX, 1
while (BH != 0) {
    --counter
    shl BX, 1
    if (carry == 0) { // AL = codificación completa de una letra
        if (AL == 0) break // (carry == 0) 2 veces consecutivas
        AL = getLetter(matrizCod, AL) // obtengo la letra correspondiente a la codificación en AL
        cadeChar[j] = AL
        EAX = 0
        j++;
    }
    rcl AL, 1
    if (counter == 0) {
        i++
        BL = cadeZip[i] // obtengo el próximo byte de codificación
        counter = 8;
    }
}
```

(*) La variable *counter* será un contador de shifteos del registro BX, cuando esta indique que se hayan realizado los 8 shifteos, entonces BL será cero, y por lo tanto debo colocar en él, el próximo byte del arreglo *cadeZip*. Como inicialmente se shiftea una vez a la izquierda el registro BX (para quitar el primer 0 de la cadena, pues de lo contrario el algoritmo terminaría en el segundo *if*), es por esto que a la variable *counter* se la inicializó en 7.