Organización del computador II

Segundo Cuatrimestre de 2008

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 1

Compresión y descompresión de imagenes

Grupo: Usain Bolt

Integrante	LU	Correo electrónico
Ya Lun Wang	623/06	aaron.wang.yl@gmail.com
Agustín Olmedo	679/06	agustinolmedo@gmail.com
Carlos Sebastian Torres	723/06	sebatorres1987@hotmail.com

Índice

1. Introducción

El objetivo en este trabajo práctico de la materia es transformar un archivo con formato BMP a un nuevo archivo con formato OC2. La transformación de uno a otro formato estará intrinsecamente relacionado al método de codificación de Huffman, para comprimir la representación binaria de simbolos, a otra codificación más corta, permitiendo así, guardarse en un menor espacio de memoria.

Estas intrucciones previamente descriptas a nivel general se harán a partir de rutinas en lenguaje C, en conjunción con subrutinas escritas en lenguaje ensamblador. Las rutinas en C serán las que están directamente relacionadas a la interfaz con el usuario, mientras que las del lenguaje ensamblador tendrá que ver con aquellas operaciones que operan directamente con la compresión de datos y el armado de estructura de datos que hacen posible dicha compresión.

Las operaciones que consisten en leer los encabezados y crear el archivo al cual se quiere comprimir o descomprimir, están dados por operaciones escritas en C. Éstas se encargan de dar formato y subir al nuevo archivo a devolver la información obtenida en la codificación o decodificación anterior, producto de las rutinas creadas en assembler. Más adelante describiremos éstas últimas en detenimiento.

Primeramente, si lo que deseamos es comprimir un BMP, se lee el formato del archivo a partir del encabezado. Esto último contiene información como el tamaño del archivo, el alto, el ancho, entre otro datos que serán usados por las rutinas que están en lenguaje ensamblador. Los programas en C se encargan de crear el prototipo del archivo OC2 con su respectivo encabezado provisto por la cátedra, mientras que los del lenguaje ensamblador se encargan de la compresión misma de los símbolos. En el archivo OC2, su encabezado tendrá, además de los datos del encabezado del BMP, una tabla de codificación que es aquella que usaremos para comprimir el dato, y que será de utilidad para su descompresión posterior.

En la descompresion, ocurre algo análogo, solamente que se escriben los datos nuevamente sobre su encabezado, dando las propiedades del archivo BMP a partir de la información provista por el archivo comprimido OC2, en su mismo encabezado (debido a que contiene ya la información del encabezado del BMP, en el encabezado de OC2)

Hemos mencionado el papel que cumple la codificación en el pasaje de BMP a OS2, y en su descompresión. Ahora daremos información sobre el método empleado para dicho proceso, a grandes rasgos, para luego, entrar en detalles acerca de la manera en la cual se utilizarán estas ideas a la hora de programarlo en lenguaje ensamblador. Estas ideas darán paso a las estructuras de datos adecuadas para su representación y daremos muestra de las razones por las cuáles los hemos elegido.

1.1. Codificacion de Huffman

La codificación de Huffman permite comprimir el espacio requerido para el almacenamiento de un string con codificacion fija, primeramente mediante una lectura de frecuencias de cada simbolo, y consiguientemente rearlizarse con este último dato un arbol binario que determina la nueva codificación. La primera observación que podemos hacerle a esta nueva codificacion es que será más corto que aquella que emplea la codificación fija. Esto es gracias a que no todo símbolo ocupa la misma cantidad de bits en su representación sino que, en aquellos para los cuales son más frecuentes en un string, tendremos para el mismo una codificación más corta para el caso de emplear la de Huffman

¿Como se logra reducir la longitud de la codificación fija? Veamos lo siguiente. Anteriormente hemos mencionado la codificación fija. Supongamos un ejemplo de codificación fija, como la del ASCII. En ésta, existe 127 caracteres y cada cual está representado por 8 bits (incluiremos en nuestro análisis el bit más significativo de paridad dado que no afectará nuestro seguimiento del presente ejemplo). Dada una cadena de caracteres con 6 caracteres, obtendremos que la longitud total que se requerirá para su representación será un total de 48 bits, es decir, 8 bits por cada caracter.

Esta longitud que vemos, puede ser achicada a partir de la codificación de Huffman, gracias a que éste último asigna a cada caracter en una cadena de caracteres una frecuencia, y a partir de éste se crea una tabla de codificación donde se asignará a cada caracter una nueva codificación. Esta nueva codificación se caracteriza por no ser fija para cada caracter. Entre dos codificacíones cuales quiera, se puede caracterizar que ningun es prefijo de otro. En otras palabras, la codificación nueva son de prefijos únicos entre caracteres. Esta es la clave que permite desprenderse de tener siempre una longitud fija en la codificación, y asi obtener que para cada caracter, el más frecuente tendra una longitud menor que aquellos que son más frecuentes

Para este trabajo práctico, emplearemos este método en la codificación de una imagen en formato BMP y dar una nueva en OC2.

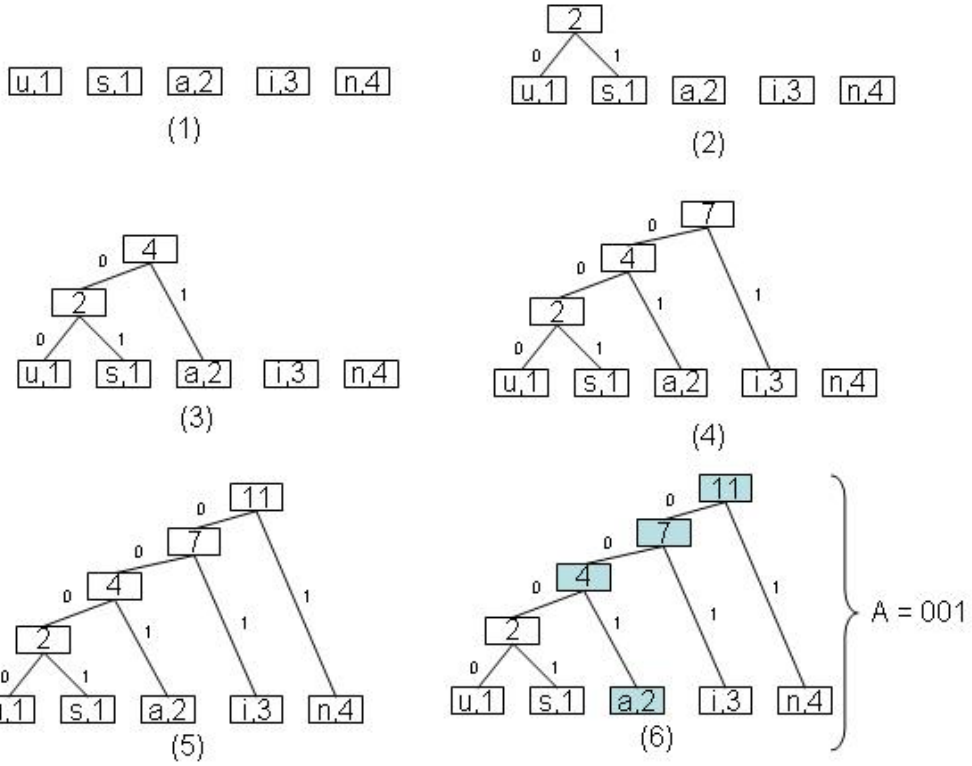
En el formato BMP hay un encabezado que da la pauta de que es efectivamente el tipo mencionado, y luego hay una secuencia de bytes. Tres bytes consecutivos conforman un píxel. En éste, se puede encontrar la imformación necesaria acerca del color que se debe decodificar. En un píxel, cada byte representa el matiz de rojo, verde y azul respectivamente, y dicho trío en conjunto con una paleta de colores informa la manera en la cual se deben combinar los colores para mostrar una imagen determinada.

La parte en la que está la información sobre cada pixel, iremos leyendo de a un byte interpretándolo como un símbolo. Luego haremos la tabla con la codificación de Huffman para estos símbolos.

La creación de la tabla tendrá dos etapas distinguibles: la primera consiste en determinar la frecuencia de cada uno de los simbolos y luego se armará un arbol de Huffman para la asignación de la frecuencia

El arbol de Huffman es lo que nos permitirá determinar la nueva codificación de prefijos unicos para cada simbolo. La construccion de dicha estructura de datos será a partir de las frecuencias de cada símbolo antes calculadas. El arbol empezará a construirse desde las raices, que en este caso son los diferentes caracteres con sus respectivas frecuencias en la tira de caracteres de la que se quiere comprimir. La idea consiste en la iteración de los siguientes pasos:

- Tomo dos nodos A y B que tengan la menor frecuencia y luego creo uno nuevo ubicandolo como el padre de los dos anteriormente tomados, y se le asigna la suma de ambas frecuencias
- Quito A y B en las futuras evaluaciones
- Repito el primer paso hasta que quede un solo nodo sin evaluar (que será la raiz)



Ya construido el arbol de Huffman, solamente nos quedará devolver la codificación para cada uno de los caracteres que nos aparecen en el string. Para esto, tengamos en cuenta que todos estos caracteres que aparecen, están situados como raices del arbol. Obtener la codificación de Huffman para cada uno de ellos es dar el camino desde la raiz hasta su hoja correspondiente que la representa. Esto quiere decir que por cada camino izquierdo que tomemos en el recorrido del arbol, escribiremos un 0, mientras que será un 1 para el camino derecho

Estos datos serán depositados en una tabla especial que contendrá dicha codificación, para que luego pueda usarse en la compresion y descompresión posterior de datos.

2. Desarrollo

2.1. Calculando la cantidad de apariciones de cada símbolo

El procedimiento para encontrar la frecuencia de cada caracter estará basado en una tabla, al que denominaremos tabla de frecuencias. La tabla de frecuencias se inicializa con los 256 posiciones inicializadas en 0. Cada una de esas posiciones representan la aparición de alguna de los caracteres que pueden ser representados en 8 bits. Por cada secuencia representado en 1 byte, se incrementará en uno, de modo que al final del algoritmo que calcula la cantidad de apariciones, tendremos la cantidad concreta de veces que aparecieron las distintas secuencias de bits agrupados de a bytes.

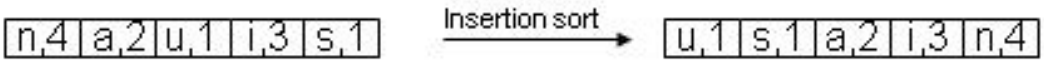
2.2. Representación del arbol de Huffman

La primera idea que se tuvo para la representación fue empleando aquella que se nos fue provista por una de las clases prácticas dada por la cátreda de la materia. En ésta, se usaba la estructura nodo que consistía en un puntero al hijo izquierdo, otro al hijo derecho, un simbolo, y finalmente una frecuencia.

El arbol se iría construyendo nodo por nodo y luego se recorrería el arbol a partir de los punteros a los hijos, mientras se escribe secuencialmente a medida que se acerca al nodo raiz deseado, un 1 por cada hijo izquierdo y un 0 por cada hijo derecho. Si bien dicha representación del arbol es derivado directamente de la idea abstracta que la constituye, nos encontramos con varias dificultades. Una de ellas estaba relacionada a la dificultad en cuanto a la manera de liberar el espeacio requerido por el arbol luego de su utilización. No solamente era costoso a nivel algoritmico sino que también tendría un rendimiento menor a la estructura de representación que presentaremos próximamente, que fue el que quedó confirmado para esta entrega del trabajo.

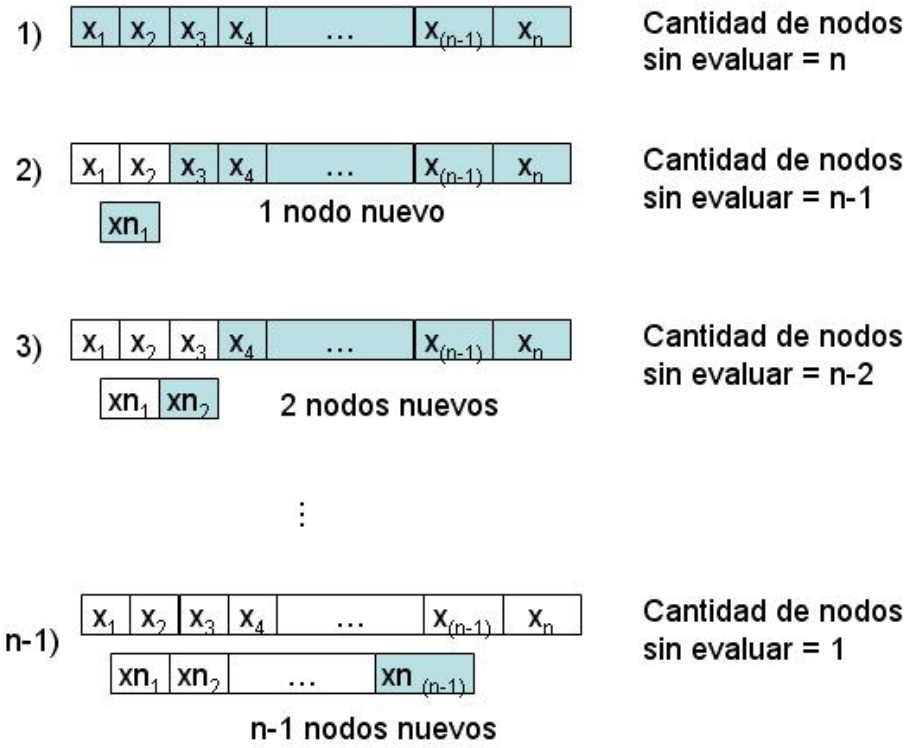
En lugar de interconectar los nodos para formar el arbol, empleamos la estructura de dato de un arreglo para representar los nuevos nodos internos. En principio, pareciera ser que emplear una estructura estática como el arreglo, implicaría un malgasto de memoria. Sin embargo, conociendo formalmente ciertas propiedades del arbol de Huffman, podemos reservar memoria para los nodos internos del árbol, sin desperdiciar nada.

El proceso para crear el árbol de Huffman comienza con un arreglo en el cual cada uno de sus contenidos tiene al caracter y la frecuencia del caracter en el string. A dicho arreglo, se lo ordenará con respecto a la frecuencia de aparición, empleando el algoritmo de insertion sort.



Mencionamos anteriormente que ibamos a reservar memoria para el espacio requerido en los nodos internos del árbol. Aquí explicaremos qué relación guarda ese espacio, con respecto a la cantidad de raices que se tiene disponible. Retomemos los pasos seguidos para la construcción del árbol que fue explicado en la introducción. Si observamos bien, podremos notar que la cantidad de nodos que quedan por evaluar se va decrementando en uno entre iteraciones consecutivas. Como se ha dicho antes, tomamos dos nodos A y B, creando un nodo C que tendrá la suma de las frecuencias de A y B. Luego marcamos A y B para dejarlos fuera de futuras evaluaciones. En la iteración i, tenemos los nodos sin evaluar A,B,x_3,...,x_n los nodos que aún quedan por evaluar, siendo x_3,...,x_n otros nodos restantes que no fueron evaluados. Si avanzamos a la iteración i+1, tendremos que los nodos que faltan evaluar son C,x_3,...,x_n. La diferencia entre la cardinalidad de A,B,x_3,...,x_n y C,x_3,...,x_n es de uno, siendo el segundo conjunto menor. Esta reducción producto de evaluar los nodos raiz restantes, se podrá hacer n-1 veces debido a que nos detendremos cuando obtengamos solamente un nodo sin evaluar. Como en cada iteración aumento en uno la cantidad de nodos internos, tengo entonces en total, n-1 nodos internos para el arbol de Huffman. Este resultado nos permite reservar memoria de antemano, sin desperdiciarla.

Usar un arreglo en lugar de nodos internos reservados de manera dinámica, permite que cuando se deje de usar el árbol, se pueda liberar simplemente con el comando free proporcionado por el lenguaje ensamblador. De otra forma, estaríamos liberando uno por uno, y consecuentemente de la estructura complicada del arbol, se podría generar errores en la programación.



2.3. Codificar

La codificación emplea la tabla con la codificación de prefijos únicos. A partir de la imagen, se irá leyendo de a byte, y para cada uno de estos, se realizará una búsqueda lineal sobre la tabla de codificación. No todos los bytes serán leidos debido a que cierta parte de ellos son bytes basura que no deben interpretarse como codificación válida. La razón que explica la presencia de estos bytes que no se codifican, se debe a que el formato requiere que cada línea sea un´múltiplo de 4 con respecto a la cantidad de bytes. Para salvar este inconveniente, tenemos un contador con la cantidad de bytes efectivos por línea, y otro con la cantidad de basura que existe al final de cada línea. El algoritmo simplemente se encargará de, una vez alcanzado el valor del contador de bytes efectivos, saltear la cantidad de bytes basura, ubicándose así sobre la siguiente linea sobre el que se quiere interpretar. También mencionaremos la presencia de un contador general que tiene la medida en bytes del archivo en general. Esto nos servirá como punto de parada una vez que ya leimos todos los bytes que son parte del archivo BMP, y nos dará la pauta de cuándo se ha terminado el algoritmo.

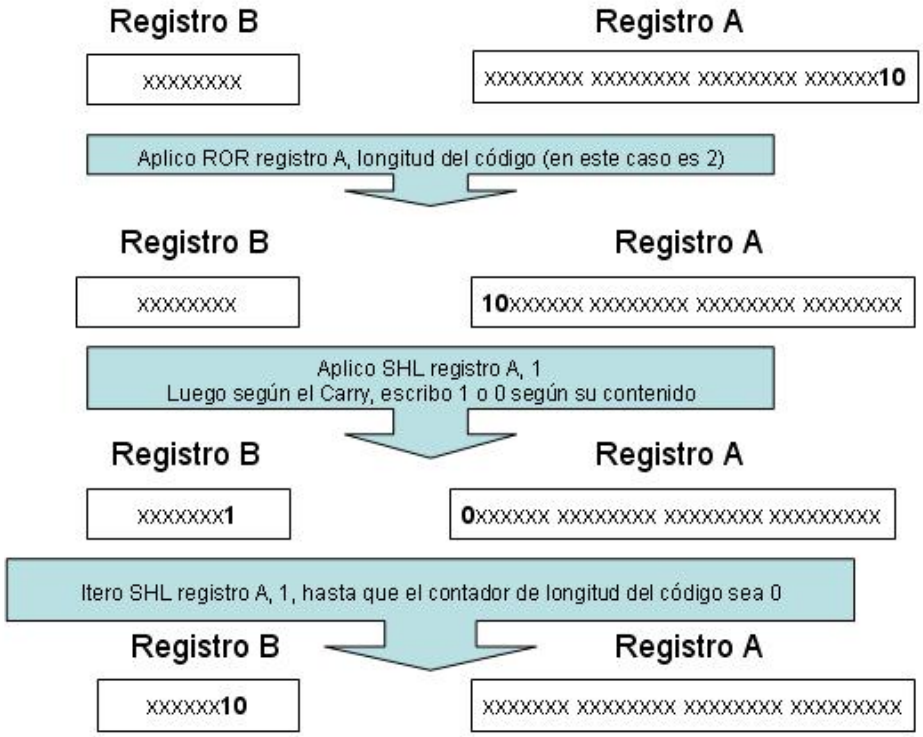
Como mencionamos anteriormente, la codificación de Huffman tiene una longitud variable, y generalmente no excede la longitud de 32 bits. Obtenido esta codificación queremos escribirlo sobre la memoria. Tambien, hay que tener en cuenta que no es posible escribir de a bit sobre la memoria, de modo que se necesita un intermedio en el proceso de escritura. Para esto usaremos el intermedio de un registro de propositos generales para dicho traspaso. El procedimiento escribirá sobre el registro hasta que todos los bits del mismo, estén en uso de alguna representación de algún codigo, y luego se transferirá el dato a memoria. Es por esta razón que tenemos que emplear máscaras para poder realizar la escritura de a bit, con el uso de algun registro de 8 bits.

Al no existir operaciones que transfieren información de a bit, estamos obligados a emplear máscaras sobre el registro que se trasfiere a memoria una vez que se llena.

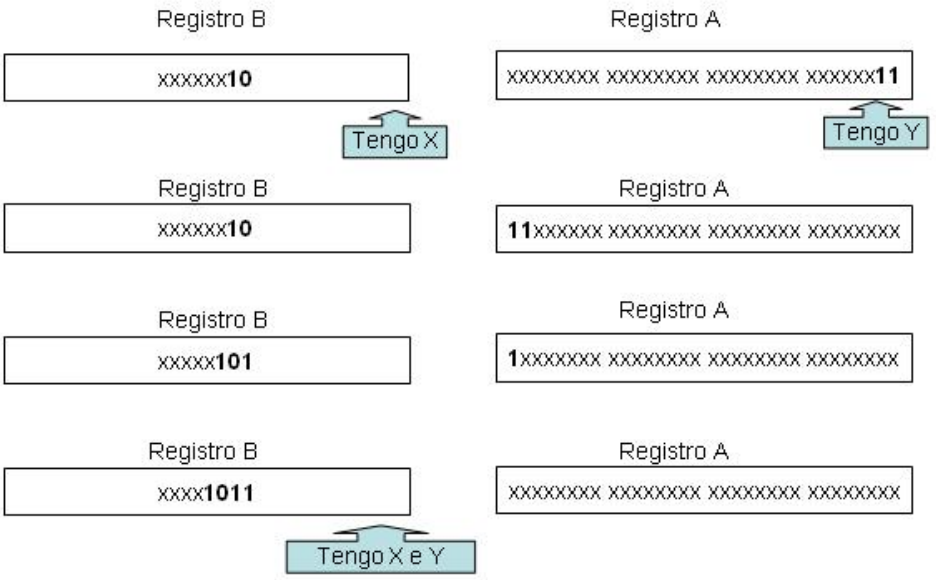
Usaremos un registro de proposito general sobre el cual iremos cargando bit a bit de los caracteres que aparecen hasta que se llenen los 8 bits, y una vez logrado esto último, cargar el registro completo a

memoria. Como hay prefijos únicos entre caracter y caracter, no es posible que a la hora de interpretarlo, se lleguen a confusiones, de modo que es válido colocar dichos códigos de manera consecutiva sobre la memoria.

Supongamos que queremos escribir los caracteres X, Y y Z con su codificación respectiva de 10,11,01. Un registro al que denomiremos A, en el cual cargaremos la codificación de X. La parte más significativa que no están siendo usados será rellenado con ceros. En otro registro auxiliar contCod se tendrá la longitud de la codificación. Gracias a la información provista por dicha longitud, seremos capaces de saber la cantidad de bits que será traspasados al registro que funcianará para la trasnferencia hacia la memoria, al que denominaremos B. Antes de pasar hacia el registro B, vamos a poner la codificación en la parte más significativa mediante la operación ROR (rotate right) tantas veces como sea la longitud indicada por contCod. Para el caso de la letra X, con su codificación 10, pasaremos sus 2 bits hacia el registro B. Para dicho proceso se empleará la operación shl(shift left) que consiste en mover el bit menos significativo al flag de carry, en un ciclo hasta que el contador de la longitud sea igual a 0. El carry flag tendra el valor del bit que se está evaluando, de modo que usaremos dicho bit para escribir un 1 o un 0, segun su valor sobre el registro B.



Para Y y Z se repite el mismo paso análogo, copiando sobre el arreglo B de manera consecutiva los bits a lo dejado en el traspaso del codigo de X



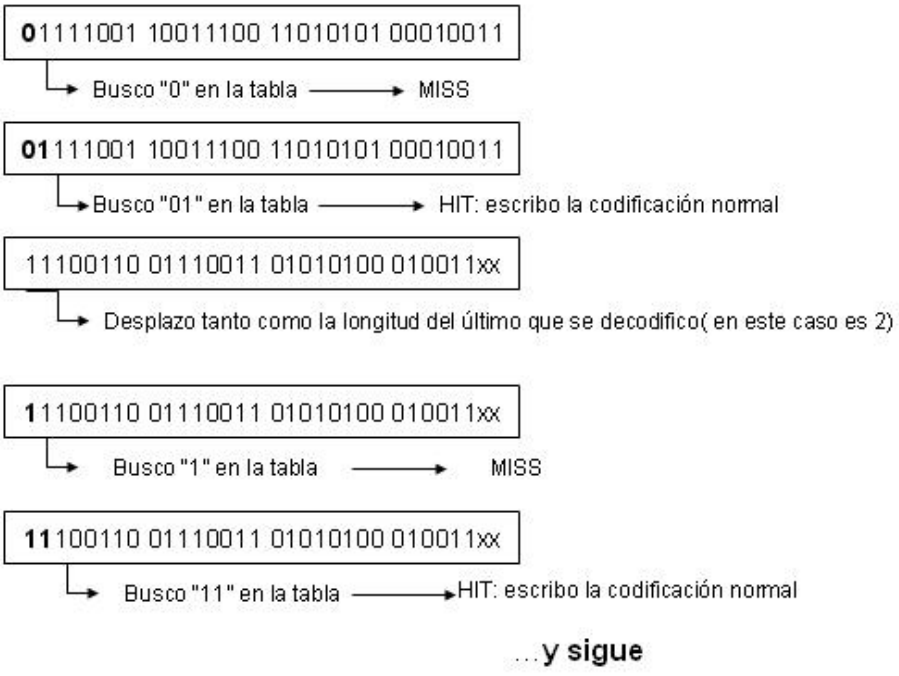
Siempre que se esté pasando un bit hacia el registro B, se pregunta si este mismo se llenó. En caso afirmativo, se procederá a pasar lo que hay en B, hacia la memoria. Cabe destacar que puede ocurrir que el registro B se llene y que a su vez aun no se ha escrito la totalidad del código de un cierto símbolo. Esto no es de mayor importancia porque para la próxima vez en el cual se imprime en memoria, los bits quedaran consecutivos, de modo que no estarán cortados de ninguna manera. Así, si queda interrumpido el traspaso de un bit correspondiente al código de algún simbolo que quedó truncado porque se lleno B, simplemente se pondrá esos bits que faltan sobre el nuevo B que estará limpiado para otros 8 bits de codificación.

Como es de suponer, es muy probable que queden bits de sobra que no representan nada en el último traspaso. Estos bits no serán tenidos en cuenta a la hora de realizarse la decodificación, y se

guardará en una variable para la eventualidad de dicho proceso, a modo de información necesaria para no interpretar bits que no deberian representar ningun símbolo para el BMP. Estps bits pueden ser del 1 al 7. Aquí también se puede ver la razón por la cual se empleó un registro de 8 bits, y no uno más grande para pasar a memoria. Si bien un registro más grande implicaría que el pasaje al bitstream sea mayor en una menor cantidad de veces, la cantidad de basura que puede llegar a generarse hacia el final del algoritmo puede ser desventajosa, tanto en el procesamiento de la decodificación, como en un aumento leve en la cantidad de bits guardados. Es de esta forma, que se decidió emplear un registro de 8 bits para el pasaje a memoria.

2.4. Decodificar

La decodificación emplea el método inverso, pero con la misma forma en la cual se escribe sobre la memoria, es decir, mediante máscaras sobre registros de propósito general y su posterior transferencia a memoria. El algoritmo recibe como parámetro el bitstream de OC2. Posteriormente empezará leyendo un bit y buscará para ese bit en la tabla de codificación, alguna que corresponda a la misma. De encontrar una codificación fija, se traspasrá a un registro que luego se transferirá a memoria directamente. De no ser asi, el algoritmo tomará un bit más que los que había tomado en la iteración anterior, y buscará la nueva codificación en la tabla de código. En otras palabras, el algoritmo tomará incrementalmente de a bit, hasta que logre encontrar una codificación que esté representado en la tabla de con los códigos de Huffman, para luego pasarlos a memoria por intermedio de algún registro de propósitos generales.



Es de observar que como los códigos presentes en el OC2 usan la codificación de Huffman, tienen consecuentemente prefijos únicos, de modo que la interpretación de los mismos llevan a una decodificación única libre de errores. No puede ocurrir que en la lectura de un código, se pase por alto algún simobolo codificado, por aquella propiedad de los prefijos únicos.

Otra cosa que difiere con respecto al algoritmo de codificar, es que no vamos a encontrarnos con truncamientos en el registro que pasa a escribir sobre el BMP, sino que ocurre en aquel que lee que del OC2 que contiene los simbolos comprimidos. Esto no es una verdadera limitación, debido a que se

emplea un registro auxiliar que guarda el código actual para el que se está buscando sobre la tabla de codificación, de modo que el truncamiento no afectará de ninguna manera. Esto permite que si bien en el registro que tiene los códigos esté truncando un código de algun símbolo, en su lectura posterior a la actualización del mismo, se continúe con el bit siguiente, sin afectar de manera alguna el código buscado.

Finalmente, el tratamiento de los bits que no son leidos, está dado como parámetro en la función. Estos bits que deben ser ignorados, son producto de la codificación de BMP a OC2, y como se ha mencionado con anterioridad, pueden ser de 0 a 31 bits aquellos que deben ser ignorados en la lectura.

3. Resultados

ACLARACIÓN: para ver las imágenes y sus correspondientes archivos comprimidos, vea en la carpeta /tporga2/TP1/resultados del disco entregado

A continuación presentaremos los pasos tomados para realizar la compresión de datos y su respectiva descompresión.

En estos pasos, empezamos tomando imágenes que solamente tuvieran los tres colores primarios usados en el RGB (es decir, el rojo, verde y el azul). El tamaño de dicha imagen era de 3 x 2 pixeles y esto nos sirvio para determinar si efectivamente había una compresión y una descompresión en el cual se tomarán correctamente los bytes efectivos, y que se ignorara aquellos que componen los bytes. El resultado de la compresión en base a Huffman no fue alentador, debido a que obtuvimos que el dato comprimido pesaba más que el dato original. Sin embargo, si tenemos en cuenta que el tamaño del header del formato oc2 es significativamente mayor que el de bmp, sumandole el hecho de que el margen de compresión estaba sumamente acotado por el tamaño original de la imagen, verdaderamente era de esperarse este tipo de resultados.

- imagen: hola;
- original: 78 bytes
- comprimido: 82 bytes
- tamaño:3x2

Luego fuimos procesando imágenes con colores simples pero que denotaban formas más complicadas que la que mostramos anteriormente. Es el caso de la siguiente imagen en el que solo aparecen dos colores, pero cuya forma representan a un cubo. Estos nos permitió probar más en detalle si al comprimir y al descomprimir, no había ninguna variación de posiciones de pixeles que generaran que la imagen se vea deformada en cuanto a la figura representada. También, el tamaño de la imagen fue aumentada para poder evaluarse la cantidad de bytes comprimidos, sin que el header de oc2 sea un factor determinante. Los resultados fueron buenos debido a que la cantidad comprimida era significativa del orden de los 80 % con respecto al tamaño original

- imagen: hola;
- original: 4,78 KB
- comprimido: 479 bytes
- tamaño:40x40

Queriamos ver el rendimiento de nuestro compresor para imagenes más complejas con definición fotográfica. Para esto, empleamos una foto de Usain Bolt (velocista olímpico que rompio su propio record y tuvo tiempo de lucir su destreza ante el resto de sus competidores antes de llegar a la meta), teniendo esta imagen un nivel de definición mayor que las que probamos anteriormente. Para este caso, el mejoramiento en cuanto a compresión fue mínimo. La explicación de esto se debe a que en imágenes como

la que acabamos de presentar, la amplia gama de colores se traduce a la presencia de casi todas las codificaciones que se pueden dar en un espacio de 1 byte. Esto quiere decir que si analizamos el buffer de la imagen, es muy probable que encontremos todos los números de 0 a 255. Como se podrá intuir fácilmente, la codificación de huffman ya no es tan eficiente para el caso en el cual existan muchas formas de código dado que este hecho sumado a aquel determinado por el tamaño total de la imagen. La codificación determinado por Huffman, en general no será mas corta que la de la codificación anterior.

Según se pudo apreciar en los ejemplos que se tomó en la compresión de datos, comprimir una imagen de matiz blanco-negro y comprimir alguna que tenga todos los colores no parecen presentar grandes diferencias de tamaño comprimido. La explicación a esto se debe a que en una imagen con matiz blanco-negro, tenemos que los 3 campos de colores primarios en el campo de un pixel son iguales. Sin embargo, el rango posible de números en esos tres campos siguen siendo de 0 a 255, como ocurre con las imágenes de RGB. De esta manera, las imágenes en blanco y negro tienen 3 bytes agrupados cuyos valores son iguales, mientras que en una imagen con multiples colores, el valor de tomado de a byte varía sin un patrón reconocible. De cualquier forma, agrupar de a 3 los bytes de forma tal para que queden iguales, como es el caso de imágenes en blanco y negro, no significan en ninguna mejora a la hora de la codificación ni tampoco en la decodificación. De modo tal, es de esperarse los resultados obtenidos en cuanto a la poca diferencia que se tiene entre la compresión de ambos tipos de imágenes.

Otras imagenes fueron comprimidas y luego descomprimidas. Estas se agrupan en alguna de las categorias anteriormente mencionadas, y no se obtuvieron resultados adversos a la explicación mencionada anteriormente, lo que corrobora la justificación para cada caso en la compresión de datos

4. Conclusiones

El trabajo práctico fue sin dudas, de gran utilidad como proceso de aprendizaje del lenguaje ensamblador por muchos motivos. La aplicación de conceptos como la creación de máscaras para trasnferir de bit, hasta el manejo eficiente de estructuras de datos que representen adecuada y eficazmente representaciones de estructuras abstractas, nos han sido de gran utilidad. Las mayores dificultades en este trabajo estuvieron dados por el manejo del lenguaje de preogramación, debido a la facilidad que se tiene de perder el hilo del razonamiento inherente a los algoritmos creados. Para amortiguar dicha dificultad, hemos empleado de forma exhaustiva el empleo de comentarios al costado de cada operación y también hemos utilizado un itinerario de usos de registros para diferentes partes de un algoritmo.

Finalmente, el trabajo nos fue de utilidad para relacionar lo aprendido en la materia con algunos conceptos básicos de compresión de datos, vistos en otras materias anteriores. Los conceptos adquiridos en cuanto a la manera de codificar nos fueron de gran interés grupal, y creemos que nos ha servido para comprender las bases del pasaje de un formato a otro empleando métodos de codificación de longitud no fija.

Los resultados mostraron que esta codificación es muy buena para aquellas imágenes en las que no se presenten muchos colores con matices diferentes, dado que en el caso contrario la codificación de Huffman no achica la codificación original de manera significativa (o de manera alguna)