Obligatorio 1 - Biblioteca de trabajo con bits

4 de abril de 2018

1 Generalidades

La aprobación de este curso se consigue mediante la correcta implementación de tres pequeños proyectos de programación (que llamaremos obligatorios). Éstos son propuestos en tres momentos del curso, aumentando en complejidad, y que forman parte de un mismo paquete, alimentándose mutuamente. Los programas desarrollados en la primer entrega serán utilizados en la segunda y en la tercera. Algo similar pasa entre lo que se desarrolle en la segunda y tercera entrega. Cada obligatorio será entregado a través de una página web habilitada para tales fines, con fecha límite de entrega señalada en la misma página. Estas entregas se complementan con pruebas parciales escritas cuyo objetivo es evaluar aspectos más teóricos relacionados con el propio obligatorio.

Es importante recalcar que **tanto la prueba escrita como el proyecto entregado son individuales.** El sistema de recepción de entregas, además de almacenar los archivos enviados por los estudiantes, realiza un control de copias contra las entregas de otros estudiantes así como de programas similares que se encuentran en la web. Ese programa es capaz de detectar copias "maquilladas", es decir donde se cambiaron nombres de variables u otras formas de ocultar una copia. Este asunto debe ser bien entendido. No tenemos ningún inconveniente en que discutan soluciones, miren en la web, etc. pero el trabajo entregado debe ser realmente el producto de vuestro trabajo y si el programa de control de copia detecta que hubo copia ello implica una sanción que puede implicar la pérdida del curso e incluso sanciones mayores, tal como está especificado en el reglamento de la Facultad.

En caso de ser posible, el sistema intentará además compilar y ejecutar la entrega de cada estudiante, a fin de dar un mínimo de información respecto de qué tan bien funciona la entrega. Dependiendo del caso, esta evaluación preliminar estará o no disponible.

La evaluación preliminar mencionada anteriormente **no** determina la nota obtenida en la prueba, siendo ésta definida por una evaluación global por parte de los docentes que incluye los obligatorios, los parciales y la participación en clase.

1.1 Formato del archivo a entregar

El archivo entregado debe ser un archivo comprimido en formato **zip** (NO se aceptan archivos en formato rar), de nombre **nombres_separados_por_infraguiones.apellidos_separados_por_infraguiones.zip** y que los fuentes estén en la raíz del zip. El contenido del archivo debe incluir los siguientes elementos (que deben estar en la raíz del mismo y no en un directorio interno):

- Todos los archivos fuente creados por el estudiante (**.h** y **.c**)
- Un archivo **Makefile** para compilar el o los programas requeridos en el trabajo.

Por ejemplo, supongamos que el obligatorio consiste en la generación de un ejecutable de nombre `oblig`, su nombre es Juan Pablo Perez Fernandez, y usted implementó dicho ejecutable en un archivo **bits.c** y su correspondiente archivo de encabezado **bits.h**. Entonces debe subir un archivo de nombre `Juan_Pablo.Perez_Fernandez.zip` con el siguiente contenido:

```
bits.c
bits.h
obligatorio.c
Makefile
```

El **Makefile** en este caso debe ser así:

```
all: libbits.a obligatorio
COPT=-Wall -ansi -ggdb

obligatorio: obligatorio.o libbits.a
	cc $(COPT) -o $@ obligatorio.o -L./ -lbits
.c.o:
	cc $(COPT) -c $<

libbits.a: bits.o
	ar rcs $@ $<
```

En el archivo **Makefile** anterior se puede generar la biblioteca **libbits.a** y también compilar un pequeño programa de prueba que hemos llamado **obligatorio.c** que simplemente llama a las funciones implementadas con ciertos valores como parámetros de entrada e imprime el resultado. Esto les debe servir a ustedes para ver si dan los resultados correctos las funciones que están en la biblioteca. Para ello se debe invocar el **Makefile** de la siguiente manera:

```
make libbits.a
make obligatorio
```

La primera línea genera la biblioteca **libbits.a** que se utiliza en el programa **obligatorio.c**. La segunda línea genera el ejecutable **obligatorio** que es posible invocar de la siguiente manera:

```
./obligatorio
```

Nota: Pueden crear un zip desde la máquina virtual con el comando `zip`; la sintaxis es, desde la carpeta de trabajo:

```
$zip -r nombre_archivo.zip
```

en el ejemplo anterior, sería `zip -r Juan_Pablo.Perez_Fernandez.zip`.

1.2 Metodología de trabajo

Algunas recomendaciones generales sobre cómo trabajar con proyectos como los que se proponen aquí:

- Simplicidad (KISS - Keep It Simple, Stupid). No complicar el código más allá de lo requerido.
- Prolijidad. No importa cuánto aburra, documentar bien lo que se hace es fundamental; es muy fácil olvidarse lo que uno mismo hizo. Esto incluye la inclusión de comentarios y el uso de variables con nombres autoexplicativos, si es posible.
- Incrementalidad. Implementar y probar de a pequeños pasos. "No construir un castillo de entrada". Es muy difícl encontrar las causas de un problema si se prueba todo simultáneamente.

2 Introducción al problema

El problema que se plantea en este obligatorio, la codificación de señales digitales, es de gran importancia en muy diversas áreas de la ingeniería eléctrica, en particular en telecomunicaciones y en electrónica. Si bien desde el punto de vista teórico y formal las herramientas para trabajar con este tipo

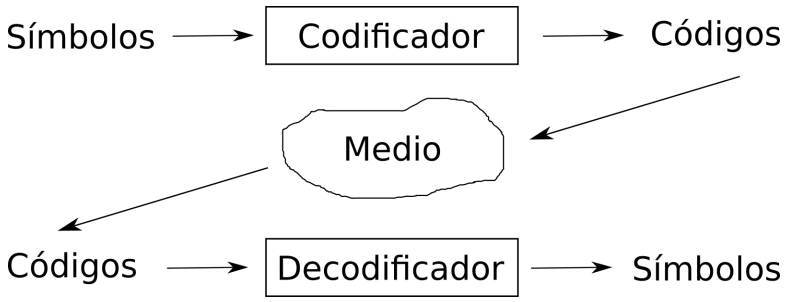

Figura 1: Esquema general de trabajo, un mensaje compuesto por N símbolos es codificado por el *Codificador*, generando k códigos que son trasmitidos a través de un canal o almacenados en memoria. En el esquema estos dos casos están representados por la nube llamada *Medio*. El *Decodificador* convierte los códigos recibidos a través del canal de comunicación o leídos de memoria y los convierte nuevamente en símbolos que pueden ser interpretados por un humano.

de problemas se ven más adelante en la carrera, es posible trabajar con, y comprender informalmente, algunos algoritmos importantes y algunos conceptos asociados.

De lo que se trata en general es de codificar una secuencia de símbolos con un fin específico. Por ejemplo para enviar un mensaje a través de un *canal de comunicación* o para almacenarlo en memoria. En esos casos es necesario asociar a cada símbolo un código binario único, de tal manera que sea posible decodificar el mensaje (en el receptor si estamos hablando de un canal de comunicación o al leer la memoria en el caso de un mensaje alamcenado). Qué código asociar a cada símbolo y cómo asociarlos es todo un mundo.

Por ejemplo puede ser interesante asociar códigos que tengan la propiedad de permitirnos darnos cuenta de si hubo algún error, y eventualmente qué error fue y corregirlo si ocurrió. Esos códigos se llaman *códigos correctores de errores* y son muy utilizados en electrónica. También puede ser interesante buscar códigos que compriman el mensaje, de manera tal que aprovechemos el canal de comunicación de manera óptima en algún sentido. Por ejemplo si ese canal tiene una capacidad limitada de trasmitir información (un cierto *ancho de banda*) podemos buscar la forma de trasmitir la información contenida en nuestro mensaje con la menor cantidad de bits posible. En otras circunstancias podemos estar interesados en ocultar ante terceros el contenido de la información trasmitida, y para ello definir una codificación que asocie códigos a símbolos de una manera que solo nosotros y el destinatario del mensaje conozcamos y que sea muy difícil de descubrir por terceras personas. Esta tercera forma de codificar da lugar a toda un área llamada *criptografía*, que se ocupa de desarrollar métodos específicos de codificación y decodificación con ese fin.

La figura 1 ilustra el esquema general de trabajo. A los efectos de este obligatorio usaremos los conceptos con poca rigurosidad, de manera más bien intuitiva. Para una definición precisa deberán esperar a los cursos específicos. De todas maneras, a fin de trabajar en este obigatorio, definiremos de manera muy general algunos conceptos: Un *símbolo* es un elemento comprensible por un ser humano, por ejemplo una letra en el alfabeto o un número. Un *mensaje* es una serie de símbolos. Llamaremos *código* a la representación de un símbolo en forma binaria, es decir como una secuencia de unos y ceros. En este contexto el *medio* es el lugar en que se almacenan o por el que se trasmiten los datos en forma de códigos, es decir una zona de memoria o un canal de comunicación respectivamente.

En estos obligatorios vamos a trabajar siempre con el conjunto de símbolos definidos en la codificación ASCII (American Standard Code for Information Interchange). Este estandar incluye las letras y cifras, así como una serie de caractéres de control que permiten escribir un texto. La codificación ASCII establece ciertos códigos para esos símbolos[1], con la propiedad de que todos esos códigos tienen un tamaño fijo, 8 bits, y es ampliamente utilizada en las computadoras.

En términos generales es posible leer un archivo de texto, byte por byte, y conociendo la codificación ASCII, sabemos qué símbolo es representado por ese conjunto de 8 bits. Del mismo modo, si tenemos

[1] http://www.asciitable.com/

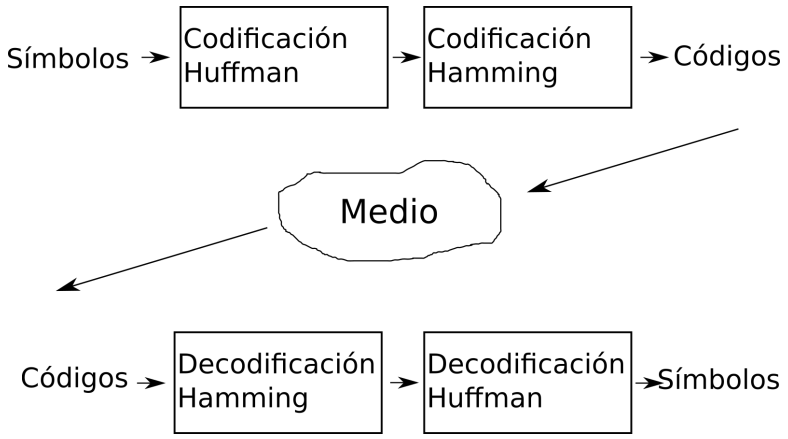

Figura 2: Esquema final donde se pueden probar las dos codificaciones implementadas en el curso, la de Huffman para comprimir y la de Hamming para detectar y corregir errores. Los códigos son trasmitidos a través de un canal o almacenados en memoria. En el esquema estos dos casos están representados por la nube llamada *Medio*.

una serie de símbolos alfanuméricos podemos escribir una sucesión de bytes correspondientes a sus respectivos códigos ASCII.

A lo largo del curso realizaremos tres obligatorios que van a permitirnos implementar la codificación y decodificación Hamming y Huffman de modo que al final del curso podamos implementar una cadena como la que se representa en la figura 2.

Para ello realizaremos tres obligatorios:

- Obligatorio 1: Construcción de una biblioteca para manejo de bits.
- Obligatorio 2: Codificación Hamming.
- Obligatorio 3: Codificación Huffman.

3 Obligatorio 1: Trabajando con bits

En el lenguaje C podemos trabajar de manera natural con datos de tipo entero (`int`), flotante (`float`) o con caracteres (o bytes) mediante el tipo `char`, entre otros. Es un poco más complicado trabajar directamente con bits. Los tipos nativos del lenguaje C no incluyen ninguno que refiera solamente a un bit, el más chico refiere a 8 bits (`char`). De modo que para actuar a nivel de bits debemos utilizar máscaras y operaciones lógicas.

Así por ejemplo, si se quiere poner a uno el tercer bit menos significativo de la palabra *input* se debe aplicar un *or bit a bit (bitwise)* entre la palabra *input* y una máscara que tenga todos los bits a cero menos el tercer bit menos significativo, como se puede observar en la siguiente expresión

$$output = input \mid 0x08$$

Del mismo modo, si queremos ver si el sexto bit de derecha a izquierda de la palabra input vale 1 o 0, podemos usar el *and bit a bit (bitwise)* mediante

$$val = input \ \& \ 0x20$$

Otras operaciones importantes para trabajar con bits son *left shift* (respectivamente *rigth shift*) representado por el operador $input << N$ ($input >> N$) que produce un desplazamiento de N bits hacia la izquierda (derecha) de la variable *input*.

Hay algunas cosas a considerar al trabajar con bits. El tipo de datos utilizados influye de manera significativa en el resultado. Si la palabra es de tipo `char`, su tamaño es de 8 bits y el bit más significativo es el bit de signo. Para considerar de la misma manera los 8 bits deberemos declarar la variable como `unsigned char`. Sucede lo mismo si trabajamos con `int`, salvo que en ese caso la palabra es de una longitud que depende de la implementación y el compliador (puede ser 32 o 64 bits, por ejemplo). En este caso consideremos que un entero es de 32 bits. Para que los 32 bits sean considerados de la misma manera deberemos declarar la variable como `unsigned int`.

Si queremos concatenar varios bits en palabras de tipo `unsigned char`, es decir de 8 bits, podemos utilizar los operadores >> y <<.

En estos obligatorios deberemos convertir símbolos en códigos, que a veces tienen un tamaño menor o mayor a 8 bits y al leer los códigos desde un archivo codificado, obtendremos palabras de 8 bits que podrán incluir varios códigos (si se trata de una sucesión de códigos de longitud pequeña) o eventualmente obtendremos fragmentos de un código si el mismo tiene una longitud mayor a 8 bits.

Por ejemplo. Si tenemos un código 01 que representa a un símbolo β, y en el mensaje hay 4 símbolos seguidos $\beta\beta\beta\beta$, en el archivo codificado aparecerá la secuencia 01010101 que puede ser leída por ejemplo mediante `var = getchar(file);` [2]

En la variable *var* (que debemos declarar como de tipo `unsigned char`) tendremos la secuencia 01010101 que será necesario interpretar, explorando el valor de cada bit y teniendo en cuenta su posición.

A fin de enteder bien este manejo de bits, que nos acerca al HW y encontraremos en diversas partes de la carrera (diseño lógico, microporcesadores, sistemas embebidos, fpga, etc.), empezaremos por construir una pequeña *biblioteca* que nos permita hacer algunas cosas con los bits directamente. Una biblioteca es un conjunto de funciones que tienen claramente especificada la forma de ser llamadas (qué tipo de variables como parámetros y qué tipo de variable devuelve, si es que devuelve algo). Esas funciones las agrupamos en un paquete que tiene un archivo *.h* común, donde están debidamente declaradas. En verdad una biblioteca se parece a un conjunto de programas como el que hacemos habitualmente, con la salvedad de que no tienen la función *main()*, dado que en todo ejecutable hay una sola función *main()* -que es el punto de entrada del programa- y esa será aportada por el programa que hagamos y que invoque a la biblioteca. Todas las funciones de la biblioteca podrán ser llamadas desde otros programas siempre que incluyamos el archivo *.h* correspondiente (para que sus declaraciones permitan al compilador verificar que todo está en orden al invocar dichas funciones) y que en el archivo Makefile demos la indicación de que se junte el programa nuestro con la biblioteca precompilada.

3.1 Descripción de la tarea

La tarea consistirá en crear una pequeña biblioteca que llamaremos `libbits`, compuesta de las funciones que describiremos a continuación.

Como forma de probar que funciona la biblioteca pueden crear un programa ejecutable que les permitirá invocar cada función. Un detalle importante es que el conjunto de funciones de la biblioteca deben estar en el archivo llamado `bits.c` y sus declaraciones en el archivo llamado `bits.h`.

Ustedes pueden escribir un programa, por ejemplo **obligatorio.c**, que contenga solo el `main()` y que invoque las distintas funciones de la biblioteca.

En este obligatorio no pedimos que implementen un programa que haga algo en particular, sino que simplemente prueben todas las funciones de la biblioteca utilizando `printf` para ver los resultados.

Un posible programa para probar la biblioteca es el siguiente:

```
#include <stdio.h>
#include "bits.h"

int main(int argc, char* argv[])
{
```

[2] Esto va sólo a modo informativo en este primer obligatorio ya que no trabajaremos con entrada/salida hasta el siguiente

```
    unsigned char mask,codigo,buffer;
    buffer = mask = 0;
    codigo = 0x5;
    printf("funcion ver_binario de 5 lugares: ");
    ver_binario(codigo,5);
    printf("funcion ver_binario de 8 lugares: ");
    ver_binario(codigo,8);
    printf("concateno los 3 bits mas a la izquierda del codigo anterior: ");
    buffer = concatenar(buffer, codigo, 3);
    ver_binario(buffer,8);
    printf("de nuevo: ");
    buffer = concatenar(buffer, codigo, 3);
    ver_binario(buffer,8);
    printf("bit 0 vale %d \n",bit(buffer,0));
    printf("bit 1 vale %d \n",bit(buffer,1));
    printf("bit 2 vale %d \n",bit(buffer,2));
    printf("creo mascara con todo a cero salvo los bits 1 a 5: ");
    mask = crear_mascara( 5, 1);
    ver_binario(mask,8);
    printf("espejo los 8 bits: ");
    buffer = espejar(mask, 8);
    ver_binario(buffer,8);
    return 0;
}
```

A continuación describimos las funciones que debe incluir la biblioteca:

NOTA: *Siempre que mencionemos el número de un bit empezaremos en 0 (como se suele hacer en el lenguaje C para contar por ejemplo en una instrucción `for`) y contaremos de derecha a izquierda. Eso quiere decir que si tenemos una palabra de 8 bits, el bit menos significativo (el de más a la derecha) será llamado `bit 0` y el bit más significativo lo llamaremos `bit 7`. De modo que cuando digamos que accedemos al 5to. bit debemos acceder al `bit 4`. A la vez, si decimos que queremos leer los 3 bits menos significativos de la palabra, debemos leer `bit 0`, `bit 1` y `bit 2`.*

3.2 bit

Esta función debe testear el valor del bit número `nb` del `buffer` y devolver su valor booleano como un entero.

Parámetros:

`buffer`**:** como carácter sin signo.

`nb`**:** como entero.

3.3 ver_binario

Esta función debe mostrar en pantalla los `nb` bits menos significativos de la palabra de entrada `buffer` en forma binaria. No devuelve nada.

Parámetros:

`buffer`**:** como entero sin signo.

`nb`**:** como entero.

3.4 set_bit

Esta función debe setear el bit `nb` del `buffer` a 0 o a 1 según sea el valor del parámetro `val` y devolver el `buffer` con ese bit seteado al valor correspondiente, como carácter sin signo.

Parámetros:

`buffer`: como carácter sin signo.

`nb`: como entero.

`val`: como entero.

3.5 concatena

Esta función debe concatenar `buffer` con `codigo`. Para ello debe primero correr `nb` a la izquierda el contenido de `buffer` y colocar en su parte baja la palabra `codigo` (de tamaño `nb` bits), y devolver la nueva palabra así creada como carácter sin signo.

Parámetros:

`buffer`: como carácter sin signo.

`codigo`: como carácter sin signo.

`nb`: como entero.

3.6 crear_mascara

Esta función debe crear una máscara cuyos bits valgan 0 con excepción de los que se encuentran entre el bit `min` y el bit `max` (incluyéndolos), que deben valer 1. Devuelve la máscara como carácter sin signo.

Parámetros:

`max`: como entero.

`min`: como entero.

3.7 espejar

Esta función debe tomar los `nb` bits menos significativos de la palabra de entrada `in` y espejarlos, es decir que el bit más significativo de ese conjunto (el bit `nb-1`) debe aparecer ahora en la posición menos significativa (en la posición `0`), el bit siguiente (el bit `nb-2`) ir a la posición `1` y del mismo modo el resto de los bits. Se debe devolver los `nb` bits menos significativos de la palabra espejada como carácter sin signo.

Parámetros:

`in`: como entero sin signo.

`nb`: como entero.

3.8 paridad

Esta función debe evaluar la paridad de la palabra de entrada `in` y retornar el valor 1 si el número de bits que valen 1 de `in` es par y 0 si ese número es impar. El valor de retorno es un entero.

Parámetros:

`in`: como entero sin signo.

3.9 Especificación de requerimientos

1. Implementar la función de nombre `bit`.
2. Implementar la función de nombre `ver_binario`.
3. Implementar la función de nombre `set_bit`.

4. Implementar la función de nombre `concatena`.
5. Implementar la función de nombre `crear_mascara`.
6. Implementar la función de nombre `espejar`.
7. Implementar la función de nombre `paridad`.
8. Crear un archivo **Makefile** que construya la libreria `libbits.a` que debe incluir las funciones anteriores.

Consideraciones y sugerencias

- Cuando una función no devuelve nada debe declararse con valor de retorno `void`.
- Recuerden que pueden utilizar la función `printf` de **stdio.h** para imprimir en pantalla valores de variables.
- Observen que `ver_binario` se puede implementar fácilmente usando `bit`.
- Implementen primero la función `ver_binario` y `bit` para poder verificar el resto fácilmente luego.
- Implementen una función a la vez, verifíquela y siga con la siguiente.