Obligatorio 1 - Biblioteca de trabajo con bits

19 de marzo de 2025

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 sucesivos del curso, aumentando en complejidad, y forman parte de un mismo paquete, alimentándose mutuamente. Los programas desarrollados en la primer entrega son 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 lo entregado debe ser realmente el producto de vuestro trabajo y si se 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, 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 **nombre1_nombre2_apellido1_appelido2.zip** en el que los fuentes están 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.

La resolución de este obligatorio consiste en la generación de un ejecutable de nombre `obligatorio1`. Supongamos que su nombre es Juan Pablo Perez Fernandez, y usted generó dichos ejecutables a partir de los archivos **obligatorio1.c**, **bits.c**, y **bits.h**, además de la biblioteca **libbits.a** que se genera desde **bits.c** y su correspondientes 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
obligatorio1.c
Makefile
```

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

```
all: libbits.a obligatorio1
COPT=-Wall -ansi -ggdb

obligatorio1: obligatorio1.o libbits.a
	cc $(COPT) -o $@ obligatorio1.o -L./ -lbits

.c.o:
	cc $(COPT) -c $<

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

clean:
	rm -f *.o *.a obligatorio1
```

A partir del archivo **Makefile** anterior se puede generar la biblioteca **libbits.a** y también compilar el programa de prueba que hemos llamado **obligatorio1.c**, que simplemente llama a las funciones de la biblioteca con ciertos valores como parámetros de entrada e imprime el resultado. Esto les debe servir a ustedes para ver si las funciones que están en la biblioteca generan los resultados correctos. Para ello se debe invocar el **Makefile** de la siguiente manera:

```
make libbits.a
make obligatorio1
```

La primera línea genera la biblioteca **libbits.a** que se utiliza en el programa **obligatorio1.c**. La segunda línea genera el ejecutable **obligatorio1**, el cual es posible invocar de la siguiente manera:

```
./obligatorio1
```

En caso que se desée limpiar el directorio con el fin de realizar una nueva compilación se debe utilizar el siguiente comando:

```
make clean
```

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 .`

Nota 2: El **Makefile** presentado anteriormente sirve para generar la biblioteca y compilar los programas, pero el que se utilizará para evaluar el trabajo de los estudiantes será el siguiente:

```
COPT= -fPIC -std=c99 -Wall -Wextra -Winline -ggdb -Iinclude
LDFLAGS= -shared -L./
LDLIBS= -lbits -lm
SOURCES= $(shell echo *.c)
OBJ=$(SOURCES:.c=.o)

all: myprogram link  .a  .so
```

```
%.o: %.c
	gcc -c -o $@ $< $(COPT)  $(LDFLAGS) $(LDLIBS)

myprogram: $(OBJ)
	gcc -o $@ $^ $(CFLAGS)

.a: $(OBJ)
	gcc $(COPT) $(LDFLAGS) -o libbits.a $(OBJ)

.so: $(OBJ)
	gcc $(COPT) $(LDFLAGS) -o libbits.so $(OBJ)

link: $(OBJ)
	@ar rcs libbits.a $(OBJ)

clean:
	rm -f *.o *.a *.so myprogram
```

La diferencia es que este Makefile tiene opciones de compilación más estrictas, como pueden ver si buscan la documentación (https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html).

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 difícil 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 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 almacenado). Qué código asociar a cada símbolo y cómo asociarlos es todo un mundo.

A fin de trabajar en este obligatorio, 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 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 dígitos, así como una serie de caractéres de control que permiten escribir un texto. La codificación

valor decimal	binario	hexadecimal
0	$0b00000000$	$0x00$
8	$0b00001000$	$0x08$
10	$0b00001010$	$0x0A$
15	$0b00001111$	$0x0F$
240	$0b11110000$	$0xF0$
255	$0b11111111$	$0xFF$

Cuadro 1: Algunos ejemplos de números y su representación en formato binario y hexadecimal.

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 para identificar los símbolos asociados a las teclas o para imprimir en pantalla.

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 una serie de símbolos alfanuméricos podemos escribir una sucesión de bytes correspondientes a sus respectivos códigos ASCII. Trabajaremos con la codificación ASCII extendida que contiene 255 códigos distintos.

A lo largo del curso realizaremos tres obligatorios que van a permitirnos implementar una biblioteca de manejo de bits y la compresión y descompresión de mensajes. Dichos obligatorios son:

- Obligatorio 1: Construcción de una biblioteca para manejo de bits.
- Obligatorio 2: Compresión y descompresión de datos usando Huffman con tabla de codificación dada.
- Obligatorio 3: Generación de la tabla de codificación de 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. En todos esos casos el largo de una palabra es fijo, por ejemplo 64, 32 o 8 bits. 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.

NOTA: *Estamos usando la siguiente convención. 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 (el de más a la izquierda) 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`.*

Una segunda convención que utilizaremos refiere a la manera de escribir una palabra. Si comienza por $0x$ *entonces la misma está escrita en formato hexadecimal, es decir que cada cuatro bits se representan por una cifra entre* 0 *(que significa* 0 *en decimal) y* F *(que significa* 15 *en decimal). Si comienza por* $0b$ *quiere decir que estamos escribiendo la palabra en formato binario. El Cuadro 1 muestra algunos ejemplos.*

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

Así por ejemplo, si se quiere poner a uno el cuarto 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 cuarto bit menos significativo (que es el bit 3), como se puede observar en la siguiente expresión

$$output = input \mid 0x08$$

El resultado estará en la variable *output*

Del mismo modo, si queremos ver si el sexto bit (que es el bit 5) de la palabra input vale 1 o 0, podemos usar el *and bit a bit (bitwise)* mediante

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

y el resultado estará en la variable *val*.

Otras operaciones importantes para trabajar con bits son *left shift* (respectivamente *rigth shift*) representado por el operador $input \ll N$ ($input \gg N$) 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 pues en un tipo con signo el bit más significativo se usa para señalar si el valor es positivo o negativo. Para considerar de la misma manera los 8 bits (es decir para no considerar el bit más significativo como bit de signo) 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). Consideremos por ejemplo un entero de 32 bits. Para que los 32 bits sean considerados de la misma manera deberemos declarar la variable como `unsigned int`.

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 hardware (HW) y encontraremos en diversas partes de la carrera (diseño lógico, microprocesadores, 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 son sus 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. Las bibliotecas no tienen la función main(). 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.a`, compuesta de las funciones que describiremos a continuación.

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

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

Para probarlo pueden escribir un programa, por ejemplo **obligatorio1.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.

NOTA: *Para saber el tamaño de un entero en la máquina en que están trabajando pueden utilizar la función* `sizeof(int)` *que devuelve el tamaño de un tipo de dato (en este caso un entero), en bytes. En lo que sigue se asume que un entero ocupa 4 bytes (es decir 32 bits).*

A continuación describimos las funciones que debe incluir la biblioteca:

3.2. bit

Dados los parámetros de entrada `nb` y `buffer`, esta función debe retornar el valor del bit `nb` del entero sin signo `buffer`. Dicho valor será 0 o 1 y deberá retornarse como entero sin signo.

Los parámetros deben respetar el siguiente orden en el encabezado de la función:

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

`nb`**:** como entero.

La función debe quedar definida de la siguiente manera:

```
unsigned int bit(unsigned int buffer, int nb);
```

Si el valor de `nb` está fuera de rango se debe devolver el valor `-1` para indicar que hubo error.

3.3. ver_binario

Dados los parámetros de entrada `nb buffer` y `nombrearchivo`, esta función debe enviar al puntero de archivo `nombrearchivo` (en forma binaria) los `nb` bits menos significativos del entero sin signo `buffer`. Luego de hacerlo, se debe finalizar con un salto de línea. Esta función retorna un entero. Si el valor de `nb` está fuera de rango o hay algún problema con el archivo apuntado por `nombrearchivo`, se debe devolver el valor `-1` para indicar que hubo error. Si todo anduvo bien debe retornar 0.

En el caso que se desee mostrar una variable de tipo char recordar que se debe castear antes de pasarlo como parámetro, por ejemplo, (unsigned int) nombre-variable.

Los parámetros deben respetar el siguiente orden en el encabezado de la función:

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

`nb`**:** como entero.

`nombreArchivo`**:** como puntero a FILE.

Por ejemplo, si buffer = $0xA0000008$ y queremos imprimir en pantalla los 5 bits menos significativos, invocamos la función así:

$$ver\_binario(buffer, 5, stdout);$$

El resultado será 01000

Se entiende que el archivo apuntado por `nombreArchivo` ya está abierto en escritura.

La función debe quedar definida de la siguiente manera:

```
int ver_binario(unsigned int buffer, int nb, FILE* nombreArchivo);
```

3.4. print_binario

Dados los parámetros de entrada `nb` y `buffer`, esta función debe imprimir en pantalla (en forma binaria) los `nb` bits menos significativos del entero sin signo `buffer`. Luego de hacerlo, se debe finalizar con un salto de línea. Esta función no tiene retorno.

En el caso que se desee mostrar una variable de tipo char recordar que se debe castear antes de pasarlo como parámetro, por ejemplo, (unsigned int) nombre-variable.

Los parámetros deben respetar el siguiente orden en el encabezado de la función:

`buffer`: como entero sin signo.

`nb`: como entero.

Por ejemplo, si buffer = $0xA0000008$ y queremos imprimir en pantalla los 5 bits más significativos, invocamos la función así:

$$print\_binario(buffer, 5);$$

El resultado será 01000

La función debe quedar definida de la siguiente manera:

```
void print_binario(unsigned int buffer, int nb);
```

3.5. setbit

Dados los parámetros de entrada `nb`, `buffer` y `val`, esta función debe setear el bit `nb` del entero sin signo `buffer` con el valor de `val` (0 o 1). Esta función retornará el nuevo `buffer` como entero sin signo. Notar que el `buffer` de entrada puede ser igual al `buffer` de salida.

Los parámetros deben respetar el siguiente orden en el encabezado de la función:

`buffer`: como entero sin signo.

`nb`: como entero.

`val`: como entero.

La función debe quedar definida de la siguiente manera:

```
unsigned int setbit(unsigned int buffer, int nb, int val);
```

Si el valor de `nb` está fuera de rango, menor a 0 o mayor a 31, se debe imprimir un mensaje de error y volver al programa que hace la llamada devolviendo el `buffer` incambiado.

3.6. concatena

Esta función debe concatenar lo que había en el entero sin signo `buffer` con los `nb` bits más significativos del entero sin signo `codigo`. Para ello primero se debe correr `nb` bits a la izquierda lo que había en `buffer` y poner los `nb` bits más significativos de `codigo` a continuación.

Ejemplo:

```
buffer =        $00000000000000000000010101010101$

codigo =        $11001100000000000000000000000000$

c = concatena(buffer, codigo, 7);

Resultado:      $00000000000000010101010111001100$
```

Los parámetros deben respetar el siguiente orden en el encabezado de la función:

`buffer`: como entero sin signo.

`codigo`: como entero sin signo.

`nb`: como entero.

Noten que estamos esperando que los bits que nos interesan de `codigo` estén en la parte alta.
La función debe quedar definida de la siguiente manera:
`unsigned int concatena(unsigned int buffer, unsigned int codigo, int nb);`

3.7. crear_mascara

Dados los parámetros de entrada `max` y `min`, 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 entero sin signo. En este caso los parámetros de entrada `max` y `min` deben tener un valor entre 0 y 31, por ejemplo, si se tiene `min` = 1 y `max` = 4, la salida debe ser: 00000000000000000000000000011110.
Los parámetros deben respetar el siguiente orden en el encabezado de la función:

`max`: como entero.

`min`: como entero.

La función debe quedar definida de la siguiente manera:
`unsigned int crear_mascara(int max, int min);`
Si el valor de `min` o `max` están fuera de rango o `min` es mayor que `max`, se debe imprimir un mensaje de error y devolver 0 al programa que hace la llamada.

3.8. espejar

Esta función debe tomar los `nbits` menos significativos de un entero sin signo `in` y espejarlos. Devuelve un entero sin signo donde los `nbits` menos signficativos contienen el espejo (es decir la palabra invertida de tal manera que el bit menos significativo deviene el más significativo) de los `nbits` menos significativos de `in` y el resto de los bits vale 0.
Ejemplo:

```
titi = $00000000000000000111101010101$

toto = espejar(titi, 15);

toto = $00000000000000001010101010111111$

toto = espejar(titi, 32);

toto = $10101010101111100000000000000000$
```

Los parámetros deben respetar el siguiente orden en el encabezado de la función:

`in`: como entero sin signo.

`nbits`: como entero.

La función debe quedar definida de la siguiente manera:
`unsigned int espejar(unsigned int in, int nbits);`
Si el valor de `nbits` está fuera de rango, se debe imprimir un mensaje de error y devolver 0 al programa que hace la llamada.

3.9. extraer

Dados los argumentos `buffer`, `min` y `max`, esta función devuelve un entero sin signo donde los bits menos signficativos contienen los bits entre `min` y `max` de `buffer` (incluyéndolos) y el resto de los bits vale 0.

Ejemplo:

```
buffer = $00000000000000000111110101010101$

toto = extraer(buffer, 8,15);

toto =  $00000000000000000000000001111101$
```

Los parámetros deben respetar el siguiente orden en el encabezado de la función:

`buffer`: como entero sin signo.

`min`: como entero.

`max`: como entero.

La función debe quedar definida de la siguiente manera:
`unsigned int extraer(unsigned int buffer, int min, int max);`

3.10. Especificación de requerimientos

1. Implementar la función de nombre `bit`.
2. Implementar la función de nombre `ver_binario`.
3. Implementar la función de nombre `print_binario`.
4. Implementar la función de nombre `setbit`.
5. Implementar la función de nombre `concatena`.
6. Implementar la función de nombre `crear_mascara`.
7. Implementar la función de nombre `espejar`.
8. Implementar la función de nombre `extraer`.
9. 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.
- Implemente primero la función `bit` y luego la función `print_binario`, para poder verificar el resto fácilmente luego.
- Implementen una función a la vez, verifíquela y siga con la siguiente.
- No se debe utilizar la librería `math` en este obligatorio.