Obligatorio 1 - Biblioteca de trabajo con bits y bytes

22 de marzo de 2022

1. Generalidades

La aprobación de este curso requiere la correcta implementación de dos pequeños proyectos de programación (que llamaremos obligatorios). Éstos son propuestos en dos momentos del curso, y que forman parte de un mismo paquete. 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 la prueba parcial escrita cuyo objetivo es evaluar aspectos más teóricos relacionados con los obligatorios.

Es importante recalcar que **tanto la prueba escrita como los proyectos entregados 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.**[1]

La nota de aprobación de la unidad curricular ésta definida por una evaluación global por parte de los docentes que incluye los obligatorios y el parcial (no hay examen).

Para aprobar la asignatura se debe tener:

1. Alcanzar un mínimo en el total (60 %)
2. Alcanzar un mínimo en cada obligatorio (25 %)
3. Alcanzar un mínimo en el parcial (25 %)

Los puntajes de cada instancia son:

1. Obligatorio 1: 24 puntos (6 puntos mínimo)
2. Obligatorio 2: 40 puntos (10 puntos mínimo)
3. Parcial final: 36 puntos (9 puntos mínimo)
4. **puntos totales: 100 puntos (60 puntos mínimo)**

[1] https://www.fing.edu.uy/es/gestion/normas-y-reglamentos

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), con el siguiente nombre:
nombres_separados_por_infraguiones.apellidos_separados_por_infraguiones.zip.
Por ejemplo, supongamos que su nombre es Juan Pablo Perez Fernandez, entonces debe subir un archivo de nombre `Juan_Pablo.Perez_Fernandez.zip`.

El contenido del archivo zip debe incluir los archivos fuentes y estos deben estar en la raíz del zip (no se deben crear directorios dentro del zip). En particular, en este obligatorio tendrán que implementar y entregar 4 archivos:

```
bits.c
bits.h
obligatorio1.c  (que contiene el main)
Makefile
```

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

```
$ zip nombre_archivo_comprimido.zip archivo_1 [archivo_2 arhivo_3 ...]
```

En el caso de este obligatorio, sería:

```
zip Juan_Pablo.Perez_Fernandez.zip bits.c bits.h obligatorio1.c Makefile
```

1.1.1. bits.h

Debe contener las definiciones de tipo y declaraciones de todas las funciones pedidas.

1.1.2. bits.c

Debe contener las definiciones de las funciones (implementarlas) pedidas. Además puede contener otras funciones auxiliares, macros, tipos auxiliares, etc. que le resulte útil para sus implementaciones.

1.1.3. obligatorio1.c

Debe contener la funcion principal `main`, que se encargue de probar sus librerías y todas sus funciones. Si bien este archivo debe ser entregado no se va a corregir ni a probar con el programa que le vamos a pasar sobre la fecha deadline del entregable. Este programa que llamaremos `autotest` sera el mismo que utilizaran los docentes para realizar la corrección del obligatorio y este revisara si las implementaciones de la bibloteca están bien realizadas.

1.1.4. Makefile

Es el archivo que contiene todo lo necesario para compilar, generar bibliotecas y el ejecutable. El **Makefile** para este obligatorio podría ser así (pero no es la única forma de escribirlo):

```
# ------------------------------------------------------------------
#  Makefile de ejemplo del obligatorio 1.
#  Progrmacion Para Ingenieria Electrica
# ------------------------------------------------------------------

# Define una constante que tiene las banderas de compilacion
COMOPT=-Wall -std=c99 -ggdb -Werror
### -Wall    : muestra todos los warnings
### -std=c99 : utiliza el estandar c99
### -ggbd    : permite utilizar los ejecutables con el debugger
```

```
### -Werror  : todos los warnings son errores

# Genera el ejecutable
obligatorio1: obligatorio1.o libbits.a
                gcc $(COMOPT) -o obligatorio1 obligatorio1.o -L./ -lbits

# Compila bits.c a codigo objeto
bits.o: bits.c
                gcc $(COMOPT) -o bits.o -c bits.c

# Compila obligatorio1.c a codigo objeto
obligatorio1.o: obligatorio1.c
                gcc $(COMOPT) -o obligatorio1.o -c obligatorio1.c

# Crea la biblioteca libbits
libbits.a: bits.o
                ar rcs libbits.a bits.o

# Borra los archivos (si existen) que son producto de la compilacion
clean:
                rm -f *.a *.o obligatorio1
```

Con el archivo **Makefile** anterior se puede generar la biblioteca **libbits.a** y también compilar el `main` que se encuentra en **obligatorio1.c**. Esto les debe servir a ustedes para ver si dan los resultados correctos las funciones que están en la biblioteca. El archivo genera la biblioteca **libbits.a** que se utiliza en el código de **obligatorio1.c** y genera el ejecutable **obligatorio1** que es posible invocar en una terminal de la siguiente manera:

```
./obligatorio1
```

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 auto-explicativos, 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. Obligatorio 1: Fundamentos

2.1. Trabajando con bits y bytes: lógica *bitwise*

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

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

También es posible realizar el xor bit a bit entre dos palabras (a ^ b), y el complemento bit a bit ($\sim$ a). Otras operaciones importantes para trabajar con bits son *left shift* (respectivamente *right 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. 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 $\gg$ y $\ll$.

A fin de entender bien este manejo de bits, que nos acerca al 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 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.

2.2. Cifrado de Vigenère

2.2.1. Introducción

Tanto en este obligatorio como en el siguiente haremos uso de un tipo de cifrado alfabético conocido como cifrado de Vigenère[2]. Sin embargo, no le daremos el uso original de cifrar texto, sino que cifraremos palabras binarias. En este caso tanto la palabra a cifrar como la clave de cifrado tendrá un alfabeto de únicamente dos letras: el caracter '0' y el caracter '1'.

El cifrado de Vigenère es un método de encriptamiento alfabético basado en sustitución poli-alfabética, es decir, en el uso de varios alfabetos de sustitución. El famoso Código Enigma, utilizado en la Segunda Guerra Mundial, también pertenece a la familia de algoritmos de sustitución poli-alfabética.

Incorrectamente atribuido al criptógrafo francés Blaise de Vigenère, su primera descripción data de 1553 por el criptógrafo italiano Giovan Battista Bellaso. La facilidad del algoritmo de cifrado y descifrado, y el desconocimiento hasta el siglo XIX de una manera eficiente de romperlo (descifrar el mensaje sin saber la clave utilizada) llevó a que fuera utilizado como cifrado de guerra, por ejemplo en la guerra civil estadounidense.

2.2.2. Descripción

El cifrado de Vigenère se obtiene aplicando una serie de Cifrados César "entrelazados". El Cifrado César de corrimiento X es el tipo de sustitución alfabética más sencilla. En esta encriptación, cada

[2]Ver artículo de wikipedia.

letra del mensaje se la sustituye por una letra X lugares más adelante. En el caso de que esta suma exceda la letra Z, se continúa circularmente hacia la A, por lo cual la operación se conoce como una suma circular. Por ejemplo, en un Cifrado César de corrimiento 3, la A se sustituye por la D, la B por la E, la Z por la C.

Ejemplo: Cifrado César de corrimiento 3

Mensaje:	`PROGRAMACION PARA INGENIERIA ELECTRICA`
Corrimiento:	3 (se considera el alfabeto de 26 letras, sin la letra Ñ)
Criptograma:	`SURJUDPDFLRQ SDUD LQJHQLHULD HOHFWULFD`

Para descifrar un criptograma de César conociendo el corrimiento, se le resta el corrimiento a cada letra del criptograma.

La idea principal del Cifrado de Vigenère es aplicar un corrimiento diferente para cada letra del mensaje, utilizando una palabra clave que se repite. Cada letra de la palabra clave definirá si aplicar un cifrado César de corrimiento 0 (en el caso de la letra A) a 25 (en el caso de la letra Z, en un alfabeto sin Ñ). La palabra clave se repetirá tantas veces como sea necesario debajo del mensaje a encriptar, como se muestra en el ejemplo:

Ejemplo: Cifrado Vigenère con VIGE como clave

Mensaje:	`PROGRAMACION PARA INGENIERIA ELECTRICA`
Clave repetida:	`VIGEVIGEVIGE VIGE VIGEVIGEVI GEVIGEVIG`
Corrimiento:	21,8,6,4,....
Criptograma:	`KZUKMISEXQUR KIXE DVMIIQKVDI KPZKZVDKG`

Para descifrar un criptograma de Vigenère teniendo la clave, se coloca la clave debajo del mensaje y se le resta a cada letra del criptograma el corrimiento correspondiente.

2.2.3. Caso Binario

En el caso binario, no hay diferencias sustantivas con lo que se describió en la sección anterior, pero vale notar que usaremos un alfabeto de dos caracteres, por lo que tanto la suma circular como la resta circular es equivalente a la operación XOR[3].

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Cuadro 1: Tabla de verdad de la operación XOR binaria.

En el recuadro que sigue se presenta un ejemplo utilizando la palabra binaria de 3 bits (o letras) 110 como clave, y la palabra binaria de 16 bits 1001100011011100 como mensaje:

[3]Si bien la encriptación que utilizamos en esta tarea es muy sencilla y no tiene aplicaciones evidentes, las encriptaciones binarias que se utilizan comercialmente se basan en la operación XOR (y en su propiedad de ser su propio inverso). Ver por ejemplo: este link y este otro link.

Ejemplo: Cifrado Vigenère Binario con 110 como clave

Mensaje:	1001 1000 1101 1100
Clave repetida:	1101 1011 0110 1101
Criptograma:	0100 0011 1011 0001

Ejercicio

Verifique que realizar un XOR bit a bit entre el criptograma y la clave (110) repetida hasta lograr el largo del criptograma, desencripta el mensaje original.

3. 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 deben 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`.

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.

En la siguiente sección describimos los tipos y funciones que debe incluir la biblioteca:

NOTA: *Siempre que mencionemos el número de un bit empezaremos en 0 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`.*

4. Especificación de requerimientos

La biblioteca debe declarar el siguiente tipo de dato en `bits.h`:

- Tipo `Clave_t`, en base a una estructura de nombre `clave`, que contenga:
  - `valor`: de tipo `unsigned int`
  - `largo`: de tipo `int`

Esta estructura se utilizará para representar claves de Vigenère de cualquier tamaño menor o igual a 32. Por otro lado, `clave.valor` contendrá la clave en sus clave.largo bits **menos** significativos, y 0s en el resto de sus bits. Así, para representar la clave 1110001 (de tamaño 7), una variable `clave`, de tipo `Clave_t`, deberá contener `clave.valor==0x00000071` y `clave.largo == 7`. Asimismo, deben declarar las siguientes funciones en `bits.h` y definirlas en `bits.c`:

4.1. bit

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

Testea el valor del bit en la posición `pos_bit` del `buffer` y devuelve su valor booleano como un entero.

- Parámetros:

  `buffer`: entero sin signo al que se le quiere testear el valor.

  `pos_bit`: posición sobre la cual se quiere testear el valor del bit.

- Retorno: valor de bit `pos_bit` de `buffer`. Este valor debe ser 0 o 1.

4.2. ver_binario

```
void ver_binario(unsigned int buffer, int pos_ls, int pos_ms);
```

Muestra en pantalla los bits entre el `pos_ms` bit más significativo (inclusive) y el `pos_ls` bit menos significativo (inclusive) de la palabra de entrada `buffer` en forma binaria. Imprimirá `pos_ms` − `pos_ls`+1 bits sin espacios entre ellos teniendo a la izquierda el bit más significativo y con un salto de linea final (`"\n"`). Se debe suponer que 31 ≥ `pos_ms` ≥ `pos_ls` ≥ 0, en caso contrario el comportamiento es indeterminado.

- Parámetros:

`buffer`: entero sin signo a imprimir en pantalla.

`pos_ls`: posición mínima a mostrar.

`pos_ms`: posición máxima a mostrar.

- Retorno: No hay.

4.3. set_bit

```
unsigned int set_bit(unsigned int buffer, int pos, int valor);
```

Establece el bit `pos` de `buffer` a 0 o a 1 según el valor del parámetro `valor` y devuelve el `buffer` con ese bit seteado al valor correspondiente. Se supone que `valor` es 0 o 1, en caso contrario el comportamiento es indeterminado.

- Parámetros:

  `buffer`: entero sin signo al que se le quiere modificar un bit.

  `pos`: posición del bit que se quiere modificar.

  `valor`: valor que se le quiere fijar al bit `pos`. Se supone que `valor` es 0 o 1.

- Retorno: Entero sin signo igual a buffer, con el bit `pos` modificado con el valor de `valor`.

4.4. crear_mascara

```
unsigned int crear_mascara(int pos_ls, int pos_ms);
```

Esta función debe crear una máscara cuyos bits valgan 0 con excepción de los que se encuentran entre el bit `pos_ls` y el bit `pos_ms` (incluyéndolos), que deben valer 1. Devuelve la máscara creada. Se debe suponer que $31 \geq$ `pos_ms` $\geq$ `pos_ms` ≥ 0, en caso contrario el comportamiento es indeterminado.

- Parámetros:

  `pos_ls`: posición mínima a partir de la cual la máscara tendrá bits 1.

  `pos_ms`: posición máxima en la cual la máscara tendrá bits 1.

- Retorno: La máscara.

4.5. concatena

`unsigned int concatena(unsigned int buffer_ms, unsigned int buffer_ls, int num_bits_ls);`
Concatena los `num_bits_ls` bits menos significativos de `buffer_ls`, en `buffer_ms`. Esto es, se debe crear un `unsigned int` *resultado* que contenga los `num_bits_ls` bits menos significativos de `buffer_ls` en sus `num_bits_ls` bits menos significativos, y a partir del bit `num_bits_ls` contenga los bits de `buffer_ms`, comenzando por el menos significativo. Se debe suponer que `num_bits_ls` ≤ 31, en caso contrario el comportamiento es indeterminado.

Ejemplo: si `buffer_ms`=0x00099999, `buffer_ls`=0x00006666 y `num_bits_ls`=10, `resultado`=0x26666666 como se muestra en la Figura 1.

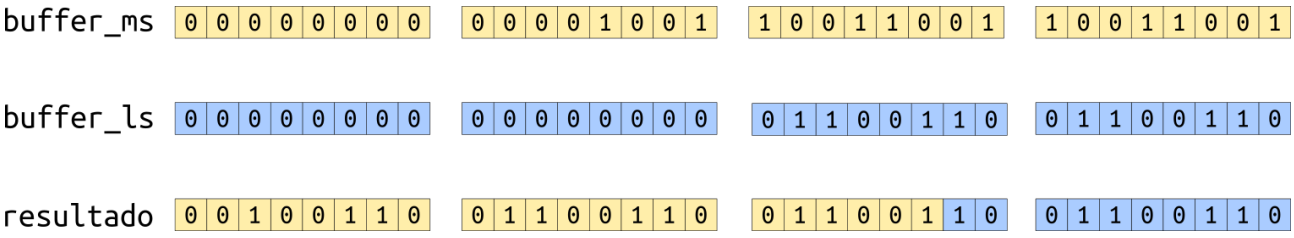

Figura 1: Ejemplo con num_bits_ls = 10.

- Parámetros:

  `buffer_ms`**:** entero sin signo a concatenar que se encontrará en la parte más significativa.

  `buffer_ls`**:** entero sin signo a concatenar que se encontrará en la parte menos significativa.

  `num_bits_ls`**:** largo en bits a concatenar de `buffer_ls`.

- Retorno: Resultado de concatenar los `num_bits_ls` bits menos significativos de `buffer_ls` con `buffer_ms`.

4.6. espejar

`unsigned int espejar(unsigned int buffer, int num_bits);`
Toma los `num_bits` bits menos significativos de la palabra de entrada `buffer` y los espeja, es decir que el bit más significativo de ese conjunto (el bit `num_bits-1`) debe aparecer ahora en la posición menos significativa (en la posición `0`), el bit siguiente (el bit `num_bits-2`) ir a la posición `1` y del mismo modo el resto de los bits. Los bits en posiciones `num_bits` o más significativa deberán quedar a 0. Se debe suponer que `num_bits` ≤ 31, en caso contrario el comportamiento es indeterminado.

- Parámetros:

  `buffer`**:** entero sin signo que se quiere espejar.

  `num_bits`**:** cantidad de bits a espejar.

- Retorno: El `buffer` espejado.

4.7. paridad

`int paridad(unsigned int buffer);`
Esta función debe evaluar la paridad de la palabra de entrada `buffer` y retornar el valor 1 si el número de bits que valen 1 de `buffer` es par y 0 si ese número es impar.

- Parámetros:

`buffer`: entero sin signo del que se desea evaluar paridad.

- Retorno: Entero valiendo 1 o 0 según la paridad de `buffer`.

4.8. ver_clave

```
void ver_clave(Clave_t clave);
```

Muestra en pantalla los bits todos los bits de `clave` sin espacios entre ellos y un salto de linea al final (`"\n"`).

- Parámetros:

  `clave`: La clave que se quiere mostrar en pantalla.

- Retorno: No hay.

Sugerencia: Implementar en una sola linea de código llamando a `ver_binario`.

4.9. rotar_clave

```
Clave_t rotar_clave(Clave_t clave, unsigned int nrot);
```

Esta función debe rotar la clave de manera circular y sin pérdida `nrot` posiciones hacia la izquierda. Por ejemplo, si la estructura `clave` contiene `clave.valor==0x00000071` y `clave.largo == 7`, una rotación de `nrot`=2, implica la siguiente actualización de clave:

$$1110001 \rightarrow 1000111$$

Por lo que se debe retornar una estructura conteniendo `clave.valor==0x00000047` y `clave.largo == 7`.

- Parámetros:

  `clave:` Estructura con los datos de la clave de Vigenère a rotar.

  `nrot:` Cuánto se debe rotar a la izquierda la clave. No hay restricciones sobre la cantidad a rotar.

- Retorno: Clave creada a partir de la rotación de la clave proporcionada.

NOTA: *Recordar que `clave.valor` debe tener 0's en los bits que no son utilizados (los bits en posiciones mayores o iguales `clave.largo`).*

4.10. encriptar

```
unsigned int encriptar(unsigned int buffer, Clave_t clave);
```

Esta función debe aplicar una encriptación Vigenère binaria a `buffer` utilizando como clave los `clave.largo` bits menos significativos de `clave.valor`. La repetición de Vigenère se hace empezando por el bit **más significativo**, como se muestra en la Figura 2:

- Parámetros:

  `buffer`: entero sin signo a encriptar.

  `clave:` estructura con los datos de la clave de Vigenère.

- Retorno: Criptograma generado a partir de `buffer` y la clave en `clave`.

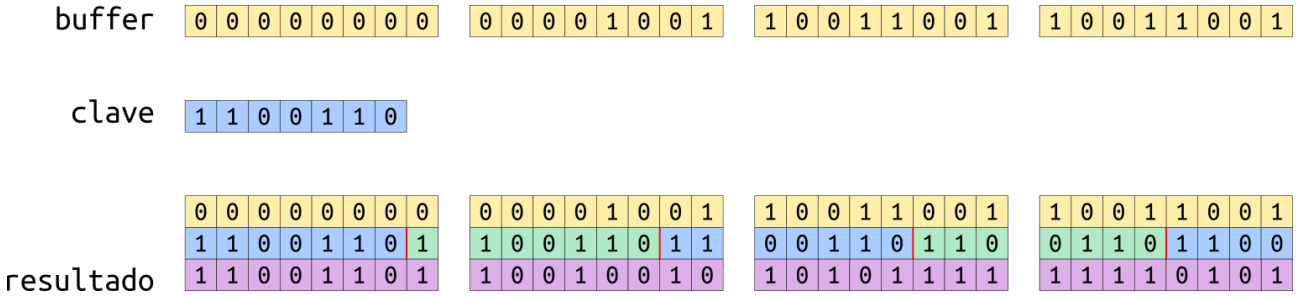

Figura 2: Ejemplo con `clave.largo` = 7.

4.11. 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 `crear_mascara`.
5. Implementar la función de nombre `concatena`.
6. Implementar la función de nombre `espejar`.
7. Implementar la función de nombre `paridad`.
8. Implementar la función de nombre `ver_clave`.
9. Implementar la función de nombre `rotar_clave`.
10. Implementar la función de nombre `encriptar`.
11. Crear un archivo **Makefile** que construya la libreria `libbits.a` que debe incluir las funciones anteriores.

Consideraciones y sugerencias

- Recuerden que pueden utilizar la función `printf` de **stdio.h** para imprimir en pantalla valores de variables.
- Implementen una función a la vez, verifíquela y siga con la siguiente.
- Para verificar una función es muy importante testear casos de borde. En nuestro caso es crucial que testeen cuando los largos son 0 o 32.