# Plancha 1:
# C y sistemas de numeración posicional

2018 – Arquitectura de las Computadoras
Licenciatura en Ciencias de la Computación
*Entrega: viernes 5 de octubre*

**1)** A continuación se presentan ciertos números enteros en binario y a su derecha, expresiones en lenguaje C incompletas. Complete estas expresiones de forma que la igualdad valga. Utilice operadores de bits, operadores enteros y constantes de enteros literales según considere necesario.

```
10000000 00000000 00000000 00000000  ==  ... << ...

10000000 00000000 10000000 00000000  ==  (1 << ...) | (1 << ...)

11111111 11111111 11111111 00000000  ==  -1 & ...

10101010 00000000 00000000 10101010  ==  0xAA ... (0xAA << ...)

00000000 00000000 00000101 00000000  ==  5 ... 8

11111111 11111111 11111110 11111111  ==  -1 & (... (1 << 8))
```

**2)** Implemente una función que tome tres parámetros $a$, $b$ y $c$ y que rote los valores de las variables de manera que al finalizar la función el valor de $a$ se encuentre en $b$, el valor de $b$ en $c$ y el de $c$ en $a$. Evite utilizar variables auxiliares.

**3)** Escriba un programa que tome la entrada estándar, la codifique e imprima el resultado en salida estándar. La codificación deberá ser hecha caracter a caracter utilizando XOR y un código que se pase al programa. como argumento de línea de comando.

El código adicional para el XOR se debe pasar como argumento de línea de comandos al programa. Es decir, suponiendo que el ejecutable se llame `prog`, la línea de comando para ejecutar el programa tendría el formato:

```
$ ./prog <código>
```

Por ejemplo, se podría hacer:

```
$ ./prog 12
$ ./prog 4321
```

¿Qué modificaciones se tendrían que hacer al programa para que decodifique? ¿Se gana algo codificando más de una vez?

Pruebe el programa codificando el código fuente del programa y utilizando como código -98. Si el archivo fuente se llama `prog.c`, esto se podría hacer con la siguiente línea de comando:

```
$ ./prog -98 <prog.c
```

**4)** *Algoritmo del campesino ruso.* La multiplicación de enteros positivos puede implementarse con sumas, AND y desplazamientos de bits usando las siguientes identidades:

$$a.b = \begin{cases} 0 & \text{si } b = 0 \\ a & \text{si } b = 1 \\ 2a.k & \text{si } b = 2k \\ 2a.k + a & \text{si } b = 2k + 1 \end{cases}$$

Úselas para implementar una función:

```
unsigned mult(unsigned a, unsigned b);
```

**5)** La arquitectura x86_64 restringe los enteros a 64 bits. ¿Qué sucede si ese rango no nos alcanza? Una solución es extender el rango utilizando más de un entero (en este caso enteros de 16 bits) para representar un valor. Así podemos pensar que:

```
typedef struct {
    unsigned short n[16];
} nro;
```

representa el valor:

$$\begin{aligned} N \quad = \quad & nro.n[0] + \\ & nro.n[1] * 2^{sizeof(short)*8} + \\ & nro.n[2] * 2^{2*sizeof(short)*8} + \\ & ... + \\ & nro.n[15] * 2^{15*sizeof(short)*8} \end{aligned}$$

Podemos pensar en la estructura `nro` como un entero de 256 bits. Lamentablemente la arquitectura no soporta operaciones entre valores de este tipo, por lo cual debemos realizarlas en software.

- Implemente funciones que comparen con 0 y con 1 y determinen paridad para valores de este tipo.
- Realice funciones que corran a izquierda y derecha los valores del tipo `nro`.
- Implemente la suma de valores del tipo `nro`.

Nota: en el repositorio Subversion de la materia hay una función para imprimir valores de este tipo. Esta función utiliza la biblioteca GMP (*GNU Multiple Precision Arithmetic Library*), por lo cual deberá compilar el código agregando la opción `-lgmp`. Puede encontrar la función en el archivo `Public/código/gmp1.c` del Subversion:

```
https://svn.dcc.fceia.unr.edu.ar/svn/lcc/R-222/Public/código/gmp1.c
```

**6)** Implemente el algoritmo del campesino ruso para los números anteriores.