Plancha 1

C y sistemas de numeración posicional

2022 - Arquitectura del Computador

Licenciatura en Ciencias de la Computación

1

Introducción

Esta plancha trata los sistemas de representación de números enteros en el lenguaje de programación C y los operadores de bits para manipular estos números a bajo nivel.

2

Procedimiento

Resuelva cada ejercicio en computadora. Cree un subdirectorio dedicado para cada ejercicio, que contenga todos los archivos del mismo. Para todo ejercicio que pida escribir código, genere un programa completo, con su función `main` correspondiente; evite dejar fragmentos sueltos de programas.

Asegúrese de que todos los programas que escriba compilen correctamente con `gcc`. Se recomienda además pasar a este las opciones `-Wall` y `-Wextra` para habilitar advertencias sobre construcciones cuestionables en el código.

3

Ejercicios

1. A continuación se presentan ciertos números enteros expresados en binario utilizando 32 bits y a su derecha, expresiones en lenguaje C incompletas. Complete estas expresiones de forma que la igualdad sea cierta. Utilice operadores de bits, operadores enteros y constantes de enteros literales según considere necesario.

   a) `10000000 00000000 00000000 00000000 == ... << ...`

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

   c) `11111111 11111111 11111111 00000000 == -1 & ...`

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

   e) `00000000 00000000 00000101 00000000 == 5 ... 8`

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

   g) `11111111 11111111 11111111 11111111 == 0 ... 1`

   h) `00000000 00000000 00000000 00000000 == 0x80000000 + ............`

   ### Soluciones

   a) `10000000 00000000 00000000 00000000 == 1 << 31`

   b) `10000000 00000000 10000000 00000000 == (1 << 31) | (1 << 15)`

   c) `11111111 11111111 11111111 00000000 == -1 & (-1 << 8)`

   d) `10101010 00000000 00000000 10101010 == 0xAA | (0xAA << 24)`

   e) `00000000 00000000 00000101 00000000 == 5 << 8`

   f) `11111111 11111111 11111110 11111111 == -1 & (~ (1 << 8))`

   g) `11111111 11111111 11111111 11111111 == 0 - 1`

   h) `00000000 00000000 00000000 00000000 == 0x80000000 + 0x80000000`

2. Implemente una función `int is_one(long n, int b);` que indique si el bit `b` del entero `n` es 1 o 0.

Solución

```
int is_one(long n, int b) {
    return n & (1 << b);
}
```

3. Implemente una función `void printbin(unsigned long n);` que tome un entero de 32 bits y lo imprima en binario.

Solución

```
void printbin(unsigned long n) {
    for (int i = 31; i >= 0; i--)
        printf("%c", is_one(n, i) ? '1' : '0');
}
```

4. 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. Evitar utilizar variables auxiliares. Ayuda: Tener en cuenta las propiedades del operador `XOR`.

Solución

```
void swap(unsigned long* a, unsigned long* b, unsigned long* c) {
    *a = *a ^ *b ^ *c;
    *b = *a ^ *b ^ *c;
    *c = *a ^ *b ^ *c;
    *a = *a ^ *b ^ *c;
}
```

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

   El código adicional para el operador `XOR` también 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> <cadena a codificar>`.

   Por ejemplo, se podría hacer `./prog 12 Mensaje` para codificar la cadena «Mensaje» con el código 12. Pruebe el programa codificando con diferentes códigos, por ejemplo, utilizando el código -98.

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

SOLUCIÓN

```c
#include <stdio.h>  // putchar
#include <stdlib.h> // atoi

int main(int argc, char** argv) {
    for (int i = 0; argv[2][i]; i++)
        putchar(atoi(argv[1]) ^ argv[2][i]);
}
```

No es necesario modificar el programa para decodificar pues

$$(m \oplus a) \oplus a = m \oplus (a \oplus a) = m \oplus 0 = m$$

No se gana nada codificando varias veces pues

$$(m \oplus a) \oplus b = m \oplus (a \oplus b)$$

6. Algoritmo del campesino ruso: La multiplicación de enteros positivos puede implementarse con sumas, el operador `AND` y desplazamientos de bits usando las siguientes identidades:

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

Úselas para implementar una función `unsigned mult(unsigned a, unsigned b);`.

SOLUCIÓN

```c
unsigned mult(unsigned a, unsigned b) {
    if (!b) return 0;                              // b == 0
    if (b == 1) return a;                          // b == 1
    if (!(b & 1)) return mult((a << 1), (b >> 1)); // b par
    return mult((a << 1), (b >> 1)) + a;           // b impar
}
```

7. Muchas arquitecturas de CPU restringen los enteros a un máximo de 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:

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

representa el valor:

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

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.

a) Implemente funciones que comparen con 0 y con 1 y determinen paridad para valores de este tipo.

b) Realice funciones que corran a izquierda y derecha los valores del tipo `nro`.

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

SOLUCIONES

a)

```
int es_cero(nro n) {
    for (int i = 0; i < 16; i++)
        if (n.n[i]) return 0;

    return 1;
}
int es_uno(nro n) {
    for (int i = 1; i < 16; i++)
        if (n.n[i]) return 0;

    return n.n[0] == 1;
}
int es_par(nro n) {
    return (!(n.n[0] & 1));
}
```

b) COMPLETAR.

c)

```
nro suc(nro n) {
    nro s = n;

    for (int i = 0; i < 16; i++) {
        s.n[i] += 1;
        if ((int) n.n[i] + 1 == s.n[i]) return s;
    }

    return s;
}

nro pred(nro n) {
    nro p = n;

    for (int i = 0; i < 16; i++) {
        p.n[i] -= 1;
        if ((int) n.n[i] - 1 == p.n[i]) return p;
    }

    return p;
}

nro add(nro a, nro b) {
    if (es_cero(b)) return a;
    return add(suc(a), pred(b));
}
```

8. Implemente el algoritmo del campesino ruso para los números anteriores.

SOLUCIÓN

```
nro mult(nro a, nro b) {
        if (es_cero(b)) return b;
        if (es_uno(b)) return a;
        if (es_par(b)) return mult(izq(a, 1), der(b, 1));
        return add(mult(izq(a), der(b)), a);
}
```