

## SEMINARUL 2

### OPERATORI PE BIȚI

**Probleme rezolvate:**

1. **Să se interschimbe valorile a două variabile de tip întreg folosind operatorul  $\wedge$  (XOR/sau exclusiv pe biți).**

**Rezolvare:**

Considerând  $x$  și  $y$  două variabile de tip întreg, proprietățile algebrice ale operatorului  $\wedge$  sunt următoarele:

- a)  $x \wedge y = y \wedge x$  (comutativitate)
- b)  $x \wedge (y \wedge z) = (x \wedge y) \wedge z$  (asociativitate)
- c)  $x \wedge 0 = x$  (numărul 0 este element neutru)
- d)  $x \wedge x = 0$  (inversul oricărui număr întreg  $x$  este tot  $x$ )

Pentru a interschimba valorile memorate în variabilele  $x$  și  $y$  vom proceda astfel:

```
x = x ^ y;  
y = x ^ y;  
x = x ^ y;
```

Folosind proprietățile algebrice enunțate mai sus, demonstrați corectitudinea acestei metode dacă variabilele  $x$  și  $y$  au adrese diferite! Ce se întâmplă în cazul în care variabilele  $x$  și  $y$  au aceeași adresă? Cum se poate evita această problemă? Dați un exemplu în care două variabile  $x$  și  $y$  au aceeași adresă!

2. **Implementarea operațiilor de deplasare circulară (rotire) pe biți pentru numere întregi.**

**Rezolvare:**

Operațiile standard de deplasare pe biți, implementate în limbajul C prin operatorii  $<<$  și  $>>$ , elimină din reprezentarea binară a unui număr întreg o parte dintre primii sau ultimii biți, în funcție de tipul deplasării. De exemplu, pentru  $x = 93_{(10)} = 01011101_{(2)}$  de tip unsigned char și deplasări pe biți cu  $p = 3$  poziții vom obține:

Bit	7	6	5	4	3	2	1	0
$x =$	<u>0</u>	1	<u>0</u>	1	1	<u>1</u>	<u>0</u>	<u>1</u>
$x \ll p =$	1	1	1	0	1	<u>0</u>	<u>0</u>	<u>0</u>
$x \gg p =$	<u>0</u>	<u>0</u>	<u>0</u>	0	1	0	1	1

Se poate observa ușor faptul că deplasările pe biți sunt, în general, niște transformări ireversibile. În exemplul de mai sus se poate observa cum bișii 7, 6 și 5 din  $x$  (subliniați cu o

singură linie) se pierd în cazul deplasării spre stânga cu 3 poziții și se adaugă 3 biți egali cu 0 pe pozițiile 2, 1 și 0.

Totuși, în unele cazuri (de exemplu, în criptografie), avem nevoie ca deplasările pe biți să fie niște operații reversibile. În acest caz, deplasările pe biți standard vor fi înlocuite cu deplasări circulare (rotiri) pe biți. Astfel, primii sau ultimii biți care se pierdeau în cazul deplasărilor standard vor fi adăugați la dreapta sau la stânga reprezentării binare a numărului respectiv, în funcție de tipul deplasării circulare. De exemplu, pentru  $x = 93_{(10)} = 01011101_{(2)}$  de tip `unsigned char` și deplasări circulare pe biți cu  $p = 3$  poziții vom obține (am notat cu `RotL` deplasarea circulară la stânga și cu `RotR` pe cea la dreapta):

Bit	7	6	5	4	3	2	1	0
$x$	<u>0</u>	1	<u>0</u>	1	1	<u>1</u>	<u>0</u>	<u>1</u>
$x \text{ RotL } p$	1	1	1	0	1	<u>0</u>	<u>1</u>	<u>0</u>
$x \text{ RotR } p$	<u>1</u>	<u>0</u>	<u>1</u>	0	1	0	1	1

Se poate observa ușor faptul că deplasările circulare pe biți sunt niște transformări reversibile. În exemplul de mai sus se poate observa cum biții 7, 6 și 5 din  $x$  (subliniați cu o singură linie) nu se pierd în cazul deplasării circulare spre stânga cu 3 poziții, fiind mutați pe pozițiile 2, 1 și 0.

Pentru a implementa deplasarea circulară la stânga a biților lui  $x$  cu  $p$  poziții (operația `RotL`), vom proceda în modul următor (am presupus ca  $x$  este o variabilă de un tip de date întreg `TDI`):

- vom deplasa biții lui  $x$  cu  $p$  poziții la stânga;
- vom deplasa biții lui  $x$  cu  $8 * sizeof(TDI) - p$  poziții la dreapta;
- vom aplica operatorul `|` (sau pe biți) între cele două valori de mai sus.

De exemplu, pentru  $x = 93_{(10)} = 01011101_{(2)}$  de tip `unsigned char` și o deplasare circulară la stânga pe biți cu  $p = 3$  poziții vom obține (într-un mod asemănător se va proceda și în cazul deplasării circulare la dreapta):

Bit	7	6	5	4	3	2	1	0
$x$	0	1	0	1	1	1	0	1
$x \ll p$	1	1	1	0	1	0	0	0
$x \gg (8 * sizeof(unsigned char) - p)$	0	0	0	0	0	0	1	0
$(x \ll p)   (x \gg (8 * sizeof(unsigned char) - p))$	1	1	1	0	1	0	1	0

În concluzie, cele două operații de deplasări circulare pe biți pot fi sintetizate astfel:

- **RotL:**  $x = (x \ll p) | (x \gg (8 * sizeof(TDI) - p))$
- **RotR:**  $x = (x \gg p) | (x \ll (8 * sizeof(TDI) - p))$

**3. Testarea/setarea/resetarea/comutarea valorii unui bit din reprezentarea binară internă a unei valori de tip întreg (cu sau fără semn).**

**Rezolvare:**

În continuare, vom considera cei  $n$  biți din reprezentarea binară internă a unei valori de tip întreg ca fiind numerotati cu  $0, 1, \dots, n - 1$  de la dreapta spre stânga (de la bitul cel mai puțin semnificativ spre bitul cel mai semnificativ). Valoarea  $n$  poate fi egală cu 8, 16, 32 sau 64, în funcție de tipul de date utilizat (char, short int, int/long int sau long long int – cu sau fără semn). În general,  $n = 8 * \text{sizeof}(TDI)$ , unde  $TDI$  este tipul de date întregi utilizat. De asemenea, în toate exemplele de mai jos, vom considera o variabilă  $x = 93_{(10)} = 01011101_{(2)}$  de tip unsigned char (adică memorată pe un octet fără semn) și o variabilă  $b \in \{0, 1, \dots, 7\}$  semnificând poziția unui bit din reprezentarea binară internă a variabilei  $x$ .

Pentru a testa valoarea bitului aflat pe poziția  $b$ , vom utiliza o "mască"  $m$  (tot de tip unsigned char) care va avea bitul de pe poziția  $b$  egal cu 1 și restul biților nuli (deci  $m = 1 << b$ ), după care vom aplica operatorul  $\&$  ("și pe biți") între ea și valoarea  $x$ . De exemplu, considerând  $b = 3$ , vom obține:

Bit	7	6	5	4	3	2	1	0
$x =$	0	1	0	1	1	1	0	1
$m =$	0	0	0	0	1	0	0	0
$x \& m =$	0	0	0	0	1	0	0	0

Se observă faptul că bitul aflat pe poziția  $b$  în  $x \& m$  va fi egal cu 1 dacă bitul aflat pe poziția  $b$  în  $x$  este egal cu 1 sau va fi egal cu 0 în caz contrar. Deoarece restul biților din  $x \& m$  sunt toți 0, datorită faptului că  $0 \& v = 0$  pentru orice valoare  $v \in \{0, 1\}$ , rezultă că  $x \& m$  va fi o valoare nenulă (nu neapărat egală cu 1!) dacă bitul aflat pe poziția  $b$  în  $x$  este egal cu 1 sau 0 în caz contrar.

Pentru a seta valoarea bitului aflat pe poziția  $b$  (adică valoarea să devină 1), vom utiliza aceeași mască  $m$  ca mai sus, dar vom aplica operatorul  $|$  ("sau pe biți"). De exemplu, considerând  $b = 5$ , vom obține:

Bit	7	6	5	4	3	2	1	0
$x =$	0	1	0	1	1	1	0	1
$m =$	0	0	1	0	0	0	0	0
$x   m =$	0	1	1	1	1	1	0	1

Se observă faptul că bitul aflat pe poziția  $b$  în  $x | m$  va fi egal cu 1, indiferent de valoarea bitului aflat pe poziția  $b$  în  $x$ , datorită faptului că  $1 | v = 1$  pentru orice valoare  $v \in \{0, 1\}$ .

Mai mult, restul biților din  $x \mid m$  vor avea aceleasi valori ca în  $x$ , datorită faptului că  $0 \mid v = v$  pentru orice valoare  $v \in \{0,1\}$ .

Pentru a reseta valoarea bitului aflat pe poziția  $b$  (adică valoarea sa să devină 0), vom utiliza o "mască"  $m$  (tot de tip `unsigned char`) care va avea bitul de pe poziția  $b$  egal cu 0 și restul biților egali cu 1 (deci  $m = \sim(1 << b)$ ), după care vom aplica operatorul `&`. De exemplu, considerând  $b = 3$ , vom obține:

Bit	7	6	5	4	3	2	1	0
$x =$	0	1	0	1	1	1	0	1
$m =$	1	1	1	1	0	1	1	1
$x \& m =$	0	1	0	1	0	1	0	1

Se observă faptul că bitul aflat pe poziția  $b$  în  $x \& m$  va fi egal cu 0, indiferent de valoarea bitului aflat pe poziția  $b$  în  $x$ , datorită faptului că  $0 \& v = 0$  pentru orice valoare  $v \in \{0,1\}$ . Mai mult, restul biților din  $x \& m$  vor avea aceleasi valori ca în  $x$ , datorită faptului că  $1 \& v = v$  pentru orice valoare  $v \in \{0,1\}$ .

Pentru a comuta valoarea bitului aflat pe poziția  $b$  (adică valoarea bitului respectiv să devină 0 dacă este 1 sau invers), vom utiliza tot o "mască"  $m$  care va avea bitul de pe poziția  $b$  egal cu 1 și restul biților nuli (deci  $m = 1 << b$ ), după care vom aplica operatorul `^` ("sau exclusiv pe biți" - XOR). De exemplu, considerând  $b = 3$ , vom obține:

Bit	7	6	5	4	3	2	1	0
$x =$	0	1	0	1	1	1	0	1
$m =$	0	0	0	0	1	0	0	0
$x ^ m =$	0	1	0	1	0	1	0	1

Se observă faptul că bitul aflat pe poziția  $b$  în  $x ^ m$  va fi egal cu complementul său față de 1, datorită faptului că  $1 ^ v = \sim v$  pentru orice valoare  $v \in \{0,1\}$ . Mai mult, restul biților din  $x ^ m$  vor avea aceleasi valori ca în  $x$ , datorită faptului că  $0 ^ v = v$  pentru orice valoare  $v \in \{0,1\}$ .

În concluzie, cele 4 operații prezentate pot fi sintetizate astfel:

- **Valoarea unui bit:**  $\text{vb} = (x \& (1 << b)) != 0;$
- **Setarea unui bit:**  $x = x | (1 << b);$
- **Resetarea unui bit:**  $x = x \& (\sim(1 << b));$
- **Comutarea unui bit:**  $x = x ^ (1 << b);$

4. Să se afișeze reprezentarea binară internă a unei valori de tip întreg.

**Rezolvare:**

Pentru a afișa reprezentarea binară internă a unei valori  $x$  de un tip întreg  $TDI$  vom afișa valoarea fiecărui bit al său, de la bitul cel mai semnificativ spre bitul cel mai puțin semnificativ. În acest sens, vom folosi o mască  $m$  în care, inițial, cel mai semnificativ bit va fi egal cu 1, iar restul bițiilor vor fi nuli. După ce vom afișa valoarea bitului cel mai semnificativ din  $x$ , vom deplasa în  $m$  bitul egal cu 1 cu o poziție spre dreapta și vom afișa valoarea bitului corespunzător din  $x$ . Vom repeta acest procedeu până când vom afișa și bitul cel mai puțin semnificativ din reprezentarea binară internă a valorii  $x$ .

```
unsigned long long m = 1ULL << (8*sizeof(TDI)-1);
unsigned char b;

for(b = 0; b < 8*sizeof(TDI); b++)
{
    printf("%u" , (x & m) != 0);
    m = m >> 1;
}
```

5. Să se verifice dacă un număr natural  $n$  este de forma  $2^k$  sau nu și, în caz afirmativ, să se afișeze valoarea  $k$ .

**Rezolvare:**

Un număr natural  $n$  este de forma  $2^k$  dacă și numai dacă reprezentarea sa binară este de forma 0 ... 010 ... 0, unde bitul egal cu 1 se află pe poziția  $k$  (se consideră biții numerotați de la dreapta spre stânga, începând cu 0). Pentru un număr  $n = 2^k$ , reprezentarea binară a numărului  $n - 1$  va fi de forma 0 ... 001 ... 1, ultimul bit egal cu 1 fiind pe poziția  $k - 1$ . Astfel, aplicând operatorul  $\&$  între  $n$  și  $n - 1$  vom obține valoarea 0 dacă și numai dacă  $n = 2^k$ .

De exemplu, pentru  $n = 16$  vom obține:

Bit	7	6	5	4	3	2	1	0
$n$	=	0	0	0	1	0	0	0
$n - 1$	=	0	0	0	0	1	1	1
$n \& (n - 1)$	=	0	0	0	0	0	0	0

Codul sursă este următorul:

```
#include <stdio.h>

int main()
{
    unsigned int n , k , aux;
    printf("n = ");
    scanf("%u" , &n);
    if((n & (n-1)) != 0)
        printf("Numarul %u nu este o putere a lui 2!" , n);
    else
    {
        k = 0;
        aux = n;
        while(aux != 0)
        {
            k++;
            aux = aux >> 1;
        }

        printf("Numarul %u este egal cu 2 la puterea %u" , n , k-1);
    }

    return 0;
}
```

Ce se întâmplă pentru  $n = 0$ ? Cum ați rezolva acest caz?

6. **Să se determine în mod eficient numărul de biți nenuli din reprezentarea binară a unui număr natural.**

**Rezolvare:**

Orice număr natural  $n$  poate fi scris ca o sumă de puteri ale lui 2, iar pozițiile biților nenuli din reprezentarea sa binară reprezintă, de fapt, exponenții puterilor respective. De exemplu, pentru  $n = 93$  vom obține:

Bit	7	6	5	4	3	2	1	0
$n$	=	0	1	0	1	1	1	0
termenii sumei	=		$2^6$		$2^4$	$2^3$	$2^2$	$2^0$

Astfel, numărul biților nenuli din reprezentarea binară a unui număr natural  $n$  este egal cu numărul puterilor lui 2 utilizate pentru scrierea sa în baza 2. Pentru a calcula în mod eficient acest număr vom folosi ideea din problema anterioară, astfel: dacă  $n \& (n-1)$  este o valoare nenulă, înseamnă că mai există puteri ale lui 2 pe care trebuie să le însumăm pentru a-l obține pe  $n$ , adică mai există biți nenuli în reprezentarea binară a lui  $n$ , deci trebuie să reluăm procedeul până când  $n$  devine 0. Practic, problema anterioară este un caz particular al acestei probleme, respectiv cazul în care suma puterilor lui 2 conține un singur termen.

```

#include <stdio.h>

int main()
{
    unsigned int n, k, aux;
    printf("n = ");
    scanf("%u", &n);

    aux = n;
    k = 0;

    while (aux != 0)
    {
        aux = aux & (aux - 1);
        k++;
    }

    printf("Reprezentarea binara a numarului %u contine %u biti
nenuli", n, k);

    return 0;
}

```

7. *Să se genereze toate submulțimile mulțimii  $A = \{1, 2, \dots, n\}$ , unde numărul natural nenul  $n$  se citește de la tastatură.*

**Rezolvare:**

Pentru a genera toate submulțimile mulțimii  $A$  vom utiliza corespondența bijectivă existentă între acestea și numerele scrise în baza 2 folosind cel mult  $n$  cifre. Practic, corespondența bijectivă este asigurată de faptul că un număr  $b = \overline{b_{n-1}b_{n-2} \dots b_2b_1b_0}$  scris în baza 2 poate fi privit ca un vector caracteristic de lungime  $n$  asociat unei submulțimi  $S \subseteq A$ :

$$b_i = \begin{cases} 0, & \text{dacă } i + 1 \notin S \\ 1, & \text{dacă } i + 1 \in S \end{cases}$$

De exemplu, pentru  $n = 3$ , deci  $A = \{1, 2, 3\}$ , vom avea următoarea corespondență bijectivă:

$S$	$b$
$\emptyset$	000
{1}	001
{2}	010
{1,2}	011
{3}	100
{1,3}	101
{2,3}	110
{1,2,3}	111

Se observă cu ușurință faptul că  $b$  ia toate valorile cuprinse posibile pentru un număr natural scris în baza 2 folosind cel mult  $n$  cifre, respectiv 0 și  $2^n - 1$ . Astfel, în program vom

parcurge toate numerele  $b$  cuprinse între 0 și  $2^n - 1$ , iar pentru fiecare număr vom afișa pozițiile pe care se află biții nenuli.

Atenție, deoarece numărul tuturor submulțimilor mulțimii  $A$  este  $2^n$ , algoritmul va avea o complexitate exponențială! Deoarece am utilizat tipul de date `unsigned long long int` pentru variabila  $b$ , rezultă faptul că valoarea maximă permisă pentru  $n$  este 64 (observați modul în care am calculat valoarea maximă posibilă pentru  $b$ , respectiv  $p = 2^n - 1$ ), o valoare mai mult decât suficientă pentru un algoritm cu complexitate exponențială.

```
#include <stdio.h>
int main()
{
    unsigned long long n , k, p, b , m;

    printf("n = ");
    scanf("%u", &n);

    if(n < 64)
        p = (1 << n) - 1;
    else
        p = ~0;

    for(b = 0; b <= p; b++)
    {
        printf("Submultimea %I64u: " , b+1);

        m = 1;
        for(k = 0; k < 8*sizeof(unsigned long long); k++)
        {
            if((b & m) != 0)
                printf("%u " , k + 1);
            m = m << 1;
        }

        printf("\n");
    }

    return 0;
}
```

**Probleme propuse:**

1. Fie  $x$  și  $y$  două numere naturale. Calculați numărul biților din reprezentarea binară internă a numărului  $x$  a căror valoare trebuie comutată pentru a obține numărul  $y$ .
2. Fie  $n$  un număr natural. Calculați numărul obținut prin inversarea ordinii biților săi. De exemplu, prin inversarea ordinii biților numărului (aici de tipul char)  $n = 26_{(10)} = 00011010_{(2)}$  se va obține numărul  $01011000_{(2)} = 88_{(10)}$ .
3. Se citește un sir format din numere naturale cu proprietatea că fiecare valoare distinctă apare de exact două ori în sir, mai puțin una care apare o singură dată. Să se afișeze valoarea care apare o singură dată în sir.
4. Fie  $n$  un număr natural. Să se determine cea mai mică putere a lui 2 mai mare sau egală decât numărul  $n$ .
5. Fie  $n$  un număr natural. Să se determine cea mai mare putere a lui 2 mai mică sau egală decât numărul  $n$ .
6. Să se calculeze numărul obținut prin aplicarea operatorului XOR între toate elementele tuturor submulțimilor unei mulțimi  $A = \{a_1, a_2, \dots, a_n\} \subset \mathbb{N}$ , mai puțin mulțimea vidă. De exemplu, pentru  $n = 3$  și  $A = \{15, 3, 8\}$ , numărul cerut va fi egal cu  $(15)^{(3)}^{(8)}^{(15^3)}^{(15^8)}^{(3^8)}^{(15^3^8)} = 0$  (am folosit parantezele doar pentru a pune în evidență elementele submulțimilor lui A).
7. Fie  $n$  un număr natural și  $i, j$  două poziții din scrierea binară a lui  $n$ . Care este valoarea numărului obținut prin interschimbarea biților aflați pe pozițiile  $i$  și  $j$  din  $n$ ?
8. Fie  $n$  un număr natural. Definim greutatea lui ca fiind numărul de biți setați (au valoarea = 1) în scrierea binară a lui  $n$ . Notăm cu  $S_k$  mulțimea tuturor numerelor care au greutatea  $k$ . Dându-se un număr  $x$  din  $S_k$  se cere care este cel mai mic număr  $y$  din  $S_k$  cu proprietatea că diferența  $|y - x|$  este minimă. De exemplu, dacă  $x = 6 = (00000110)_2$ , alegem  $y = 5 = (00000101)_2$ ; dacă  $x = 7 = (00000111)_2$  alegem  $y = 11 = (00001011)_2$ .
9. Fie  $v$  un vector cu  $n - 2$  componente ce conține primele  $n$  numere naturale ( $1, 2, \dots, n$ ) aranjate la întâmplare din care am extras la întâmplare 2 numere. Cum găsiți cele două numere lipsă folosind, printre altele, operatori pe biți (XOR)?