Operații pe biți

Capitolul

17

- **Noțiuni de bază**
- **Reprezentarea numerelor în memorie**
- **Operații pe biți**
- **Înmulțire cu operații pe biți**
- **Implementări sugerate**
- **Probleme propuse**
- **Soluțiile problemelor**

17.1. Noțiuni de bază

Până în prezent memoriile utilizate de calculatoare sunt construite din materiale magnetizabile, având două stări cărora prin convenție li se asociază valorile binare 0 şi 1.

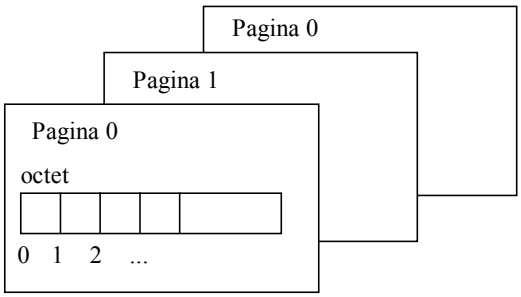
Unitatea de memorie care poate „înmagazina" o valoare de 0 sau 1 se numeşte ***bit*** (***bi****nary digi****t***).

Informațiile reprezentate cu biți sunt grupate în memorie câte 8, formând octeți.

Memoria este privită ca o înşiruire de octeți, aceştia fiind identificați printr-o adresă. Octeții de memorie sunt grupați la rândul lor pe pagini de memorie.

Exemplu

Memoria *RAM* (***R****andom* ***A****ccess* ***M****emory*):



Numărul de octeți ai memoriei interne exprimă capacitatea de memorie a acesteia.

Alte unități de memorare folosite în exprimarea capacității de memorare sunt:

- Kilobyte (notăm KB): 1KB =2^{10} octeți (bytes);
- Megabyte (notăm MB): 1MB =2^{10} KB;
- Gigabyte (notăm GB): 1GB =2^{10} MB.

17.2. Reprezentarea numerelor în memorie

Având în vedere că memoria poate reține doar cifre binare (0 şi 1), numerele vor fi păstrate pe orice suport magnetic în sistemul de numerație cu baza 2.

17.2.1. Reprezentarea numerelor naturale

Pentru a reprezenta un număr trebuie să cunoaştem tipul acestuia, fiecărui tip de dată fiindu-i caracteristic un anumit număr de octeți pentru reprezentare.

În limbajul Pascal:	
tip	număr octeți
`Byte`	1 octet
`Word`	2 octeți

În limbajul C:	
tip	număr octeți
`unsigned short`	1 octet
`unsigned int`	2 octeți
`unsigned long`	4 octeți

Să determinăm acum care numere naturale pot fi reprezentate pe un octet.

Un octet este format din 8 biți, pozițiile binare pe care le ocupă sunt numerotate de la stânga la dreapta. Bitul c_i, are rangul i în componența reprezentării.

Poziții binare:	0	1	2	3	4	5	6	7
	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0

De exemplu, despre primul bit din stânga spunem ca este bitul de pe poziția 0 şi are rangul 7.

Cel mai mic număr natural care poate fi reprezentat pe 8 biți este $0_2 = 0_{10}$:

0	0	0	0	0	0	0	0

Cel mai mare număr natural care poate fi reprezentat pe 8 biți este:

1	1	1	1	1	1	1	1

Pentru a calcula valoarea acestui număr în sistemul zecimal aplicăm algoritmul de conversie din baza 2 în baza 10:

$11111111_2 = 1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1$

sau mai simplu, folosind operațiile din baza 2 numărul scriindu-se ca

$100000000_2 -$ $\quad 256_{10} -$

$\underline{\qquad\quad 1_2}$ $\quad \underline{\quad 1_{10}}$

11111111_2 $\quad 255_{10}$

Putem trage concluzia că pe 8 biți se pot reprezenta numere între 0 şi 255 ($2^8 - 1$).

Prin analogie, pe 16 biți (2 octeți) se pot reprezenta numere naturale cuprinse între 0 şi $2^{16} - 1$.

Pentru a reprezenta un număr natural pe un număr dat de octeți se vor efectua următoarele operații:

1) Se converteşte numărul în baza 2;
2) Se aliniază biții din componența numărului la dreapta; apoi în stânga primei cifre semnificative reprezentarea se completează cu 0 (dacă este cazul).

Exemple

a. Să se reprezinte numărul 480_{10} pe 2 octeți.

Pasul 1: Numărul se reprezintă în baza 2;

Aplicăm algoritmul cunoscut şi obținem $480_{10} = 111100000_2$.

Pasul 2: aliniem cifrele numărului în baza 2 la dreapta pe 16 biți (2 octeți), completând la stânga cu 0-uri:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0

b. Să se reprezinte numărul 480_{10} pe 1 octet:

Pasul 1: Numărul în baza 2: 111110000_2.

Pasul 2: Reprezentăm acest număr pe un octet:

	0	1	2	3	4	5	6	7
1	1	1	1	1	0	0	0	0

Se observă că s-a pierdut cifra cea mai semnificativă a numărului, ceea ce înseamnă că 480 nu poate fi reprezentat pe 1 octet. Tipul de dată trebuie ales în deplină concordanță cu mărimea valorilor (şi deci lungimea în biți a acestora).

17.2.2. Reprezentarea numerelor întregi cu semn

În limbajul Pascal avem:	
tip	număr de octeți
`Shortint`	1 octet
`Integer`	2 octeți
`Longint`	4 octeți

În limbajul C avem:	
tip	număr de octeți
`-`	1 octet
`short`	2 octet
`long, float`	4 octeți

Numerele întregi cu semn se vor reprezenta asemănător numerelor fără semn, cu deosebirea că bitul cel mai semnificativ (bitul 0) va reprezenta de această dată semnul numărului (+ sau –). Prin convenție, dacă bitul semn are valoarea 0, numărul este considerat a fi pozitiv, iar dacă bitul semn este egal cu 1, numărul este considerat a fi negativ.

Reprezentarea numerelor pozitive se realizează la fel ca şi pentru numerele naturale, dar lungimea de reprezentare este mai mică cu 1.

Pentru a reprezenta un număr negativ se aplică următorii paşi:
Pasul 1: Se reprezintă numărul în baza 2 (ca şi cum ar fi pozitiv);
Pasul 2: Numărul obținut se scade din 2^k (unde k este lungimea reprezentării în biți);
Pasul 3: Se reprezintă numărul obținut la pasul 2 pe k biți ca în algoritmul precedent.

Exemplu
Să se reprezinte numărul -40_{10} pe un octet ($k = 8$).
Pasul 1: $40_{10} = 101000_2$
Pasul 2: $100000000_2 -$
$\underline{000101000_2}$
011011000_2
Pasul 3:

1	1	0	1	1	0	0	0

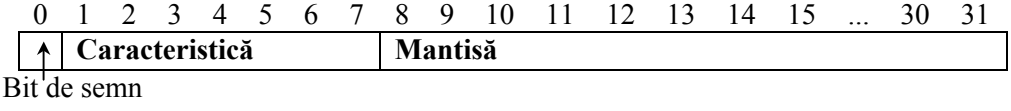
Alternativă practică pentru punctul 2:
Se poate demonstra că dacă se complementează numărul bit cu bit (operație de negare pe bit) şi apoi la ceea ce s-a obținut, se adună 1, vom ajunge la acelaşi rezultat:
00101000_2 negat bit cu bit devine 11010111_2.

$11010111_2 +$
$\underline{\qquad\quad 1_2}$
11011000_2

17.2.3. Reprezentarea numerelor reale

Reprezentarea numerelor reale se realizează în convenția virgulă mobilă.
Reamintim că un număr real se poate scrie sub forma: *c*,*cc...c*E+/–*p* , unde cu *c* am notat cifrele numărului, iar *p* este format din una sau două cifre şi reprezintă puterea la care este ridicat 10 (exponentul).

Exemplu
$2{,}003E{-}5 = 2{,}003 \cdot 10^{-5}$
Dacă spațiul de reprezentare este de 4 octeți (32 biți), acesta are structura:



Pentru a reprezenta în calculator un număr real se respectă următorii paşi:

Pasul 1: Se reprezintă numărul în baza 2;
Pasul 2: Numărul se aduce la formă „normalizată”, adică la forma $0,1... \cdot 10^p$ (partea întreagă este egală cu 0, iar prima cifră de după virgulă este 1; în funcţie de această transformare se stabileşte valoarea exponentului p); mantisa conţine toate cifrele de după virgulă;
Pasul 3: Caracteristica se calculează după formula: $2^7 + p$;
Pasul 4: Caracteristica şi mantisa se introduc în reprezentare, aliniate la dreapta în spaţiile corespunzătoare şi se stabileşte bitul semn al numărului (dacă numărul este pozitiv acesta va fi egal cu 0, dacă nu, va fi 1).

Exemplu
Să se reprezinte numărul 31,5 pe 4 octeţi.
Pasul 1: Trecem numărul 31 din baza 10 în baza 2: $31_{10} = 11111_2$.
Convertim partea fracţionară: $0,5_{10} = 0,1_2$. Deci $31,5_{10} = 11111,1_2$.
Pasul 2: Se normalizează numărul:
$11111,1_2 = 0,111111_2 \cdot 10^5{}_2 \Rightarrow$ Mantisa = 111111_2, exponentul $p = 5$.
Pasul 3: Caracteristica este: $(2^7 + 5)_{10} = 10000000_2 + 101_2 = 10000101_2$.

0	1	2	3	4	5	6	7	8	9	10	...	25	26	27	28	29	30	31
0	1	0	0	0	1	0	1	0	0	0	...	0	1	1	1	1	1	1

17.3. Operaţii pe biţi

După cum bine ştim, în memoria calculatorului toate numerele sunt reprezentate în sistemul de numeraţie binar. Noi le vedem în sistemul de numeraţie zecimal doar pentru că sistemul efectuează permanent conversii de la tastatură spre memorii şi de la memorie înspre monitor pentru ca utilizatorii să nu fie obligaţi să descifreze numere reprezentate în sistemul binar. Ca atare, cu ajutorul operatorilor pe biţi putem acţiona direct asupra reprezentării binare fără a fi nevoie de conversii.

Deoarece operatorii pe biţi se execută mai rapid decât operatorii aritmetici, în implementarea algoritmilor se preferă frecvent utilizarea lor în schimbul celor aritmetici.

17.3.1. Negarea

Negarea este un operator unar care acţionează asupra unui singur operand de tip întreg având ca efect schimbarea fiecărui bit al reprezentării numărului din 1 în 0 şi din 0 în 1.

Notaţii: `not a` în limbajul Pascal şi `~a` în limbajul C++.

Exemplu
Să se calculeze valoarea lui `not a`, pentru $a = 40$.

	Valoare zecimală	*Reprezentare pe 1 octet*
`a`	40	00101000
`not` `a` sau `~a`	215	11010111

17.3.2. Disjuncția

Disjuncția, (numită *sau* logic) este un operator binar care acționează asupra a doi operanzi întregi, bit cu bit, având ca efect setarea bitului rezultat pe 1, dacă cel puțin unul dintre biți este 1 şi pe 0 în caz contrar. Tabla de operare a disjuncției pe bit este:

`a` **`or`** `b`	`0`	`1`
`0`	`0`	`1`
`1`	`1`	`1`

Notații: `a` **`or`** `b` în limbajul Pascal şi `a | b` în limbajul C++.

Exemplu

Să se calculeze valoarea expresiei `a` **`or`** `b`, pentru $a = 40$ şi $b = 38$.

	Valoare zecimală	*Reprezentare pe 1 octet*
`a`	40	00101000
`b`	38	00100110
`a` **`or`** `b` sau `a | b`	46	00101110

17.3.3. Conjuncția

Conjuncția, (numită *şi* logic) este un operator binar care acționează asupra a doi operanzi întregi având ca efect setarea fiecărui bit al rezultatului conform aplicării următoarei table de operare:

`a` **`and`** `b`	`0`	`1`
`0`	`0`	`0`
`1`	`0`	`1`

Notații: `a` **`and`** `b` în limbajul Pascal şi `a & b` în limbajul C++.

Exemplu

Să se calculeze valoarea expresiei `a` **`and`** `b`, pentru $a = 40$ şi $b = 38$.

	Valoare zecimală	*Reprezentare pe 1 octet*
`a`	40	00101000
`b`	38	00100110
`a` **`and`** `b` sau `a & b`	32	00100000

17.3.4. „Sau exclusiv”

Sau exclusiv este un operator binar care acționează asupra a doi operanzi întregi bit cu bit având ca efect setarea bitului rezultat pe 1, dacă cei doi biți nu au aceeaşi valoare şi pe 0 în caz contrar. Tabla de operare a operatorului sau exclusiv pe bit este:

`a` **`xor`** `b`	`0`	`1`
`0`	`0`	`1`
`1`	`1`	`0`

Notații: `a` **`xor`** `b` în limbajul Pascal şi `a ^ b` în limbajul C^{++}

Exemplu

Să se calculeze valoarea lui `a` **`xor`** `b`, pentru $a = 40$ şi $b = 38$.

	Valoare zecimală	*Reprezentare pe 1 octet*
`a`	40	00101000
`b`	38	00100110
`a` **`xor`** `b` sau `a ^ b`	20	00001110

17.3.5. Rotiri pe biți

Există doi operatori, unul pentru rotirea biților spre dreapta, iar celălalt pentru rotirea biților spre stânga. Aceştia sunt operatori binari, primul operand fiind numărul ai cărui biți vor fi rotiți, iar cel de-al doilea reprezintă numărul de rotații. În locul biților deplasați din margine, numărul se completează cu 0-uri.

Notații: **`shl`** pentru rotire la stânga (***Sh**ift **L**eft*) şi **`shr`** pentru rotire la dreapta (***Sh**ift **R**ight*).

Exemplu

Să se calculeze valoarea lui `a` **`shr`** `3` şi a lui `a` **`shl`** `1`, pentru $a = 40$.

	Valoare zecimală	*Reprezentare pe 1 octet*	*Echivalent cu*
	40	00101000	
`shr` `3`	5	00000101	$5 = 40$ **`div`** 2^3
`shl` `1`	80	01010000	$80 = 40 * 2^1$

Observații

Se observă că orice rotire (*translatare*) la stânga cu o poziție este echivalentă cu o înmulțire a numărului cu 2 şi orice rotire la dreapta cu o poziție este echivalentă cu o împărțire a numărului la 2.

17.4. Înmulțire cu operații pe biți

Prezentăm un algoritm care efectuează înmulțirea a două numere întregi ($a,b \leq 10000$) fără ca în implementare să fie nevoie de operatorul înmulțire (*).

```
Algoritm Înmulțire(a,b,rezultat):
  cât timp a ≠ 0 execută:
    dacă bitul cel mai din dreapta = 1 atunci
      rezultat ← rezultat + b
    sfârşit dacă
    a ← a*2                          { îl vom implementa cu shl }
    b  ← [b/2]                       { îl vom implementa cu shr }
  sfârşit cât timp
sfârşit algoritm
```

În instrucțiunea **dacă** se face referire la ultimul bit din primul operand. Acesta se poate accesa simplu, efectuând operația **and** între *a* şi constanta 1 fără să fie nevoie de utilizarea împărțirii întregi la 2. Aici 1 joacă rol de *mască*.

O *mască* este un număr construit în aşa fel încât să permită testarea valorii unuia sau a mai multor biți ai unui număr dat.

Să considerăm un exemplu concret şi să urmărim paşii algoritmului pentru $a = 40$ şi $b = 3$.

$a = 40$	$b = 3$	a **and** 1	rezultat = 120
00101000	00000011	0	0
00010100	00000110	0	0
00001010	00001100	0	0
00000101	00011000	1	$00011000_2 = 24$
00000010	00110000	0	00011000_2
00000001	01100000	1	$00011000_2 + 01100000_2 = 01111000_2 = 120$

De exemplu, să luăm în considerare situația 00101000 **and** 00000001 = 00000001, ($=1_{10}$) sau următoarea: 00101000 **and** 00000001 = 00000000, ($=0_{10}$). Rezultă clar că astfel putem obține informația dorită despre valoarea ultimului bit a lui *a*.

17.5. Implementări sugerate

Pentru a vă familiariza cu utilizarea operatorilor care permit prelucrarea datelor pe biți, propunem rezolvarea următoarelor exerciții.

1. schimbarea semnului unui număr;

2. schimbarea valorii unui bit dintr-un număr;

3. codificarea şi decodificarea unui număr pe baza unei chei;
4. înmulțirea şi împărțirea cu puteri ale lui 2;
5. numărarea biților având valoarea 1 (sau 0) din reprezentarea unui număr în baza 2;
6. descompunerea unui număr întreg în octetul din stânga şi octetul din dreapta;
7. rotirea cifrelor binare ale unui număr întreg;
8. schimbarea cifrelor binare egale cu 1 în 0 şi invers;
9. generarea tuturor submulțimilor mulțimii $\{1, 2, \ldots, N\}$;
10. generarea tuturor şirurilor care conțin numai valorile 0 şi 1.

17.6. Probleme propuse

17.6.1. Afişare în baza 2

Scrieți un program care citeşte un număr natural *n* şi afişează cifrele reprezentării binare a numărului.

Date de intrare
Numărul natural *n* se citeşte din fişierul **`NR.IN`**.

Date de ieşire
Numărul reprezentat în baza 2 se va scrie în fişierul **`NR.OUT`**.

Restricții şi precizări
- $1 \le n \le 1000000000$.

Exemplu

`NR.IN`	**`NR.OUT`**
`40`	`101000`

17.6.2. Sumă de puteri ale lui 2

Să se descompună un număr natural *n* ca sumă de puteri ale lui 2.

Date de intrare
Numărul natural *n* se citeşte din fişierul **`DATE.IN`**.

Date de ieşire
Descompunerea numărului *n* ca sumă de puteri ale lui 2 se va scrie în fişierul de ieşire **`DATE.OUT`**. Operația de ridicare la putere va fi reprezentată prin caracterul `^`.

Restricții şi precizări
- $1 \le n \le 1000000000$.

Exemplu

DATE.IN	**DATE.OUT**
40	2^5+2^3

17.6.3. Inversare

Se consideră două numere întregi *a* şi *b*. Să se inverseze conținutul variabilei *a* cu conținutul variabilei *b* fără utilizarea unei variabile auxiliare.

Date de intrare

Cele două numere se află pe prima linie a fişierului de intrare **INVERS.IN**.

Date de ieşire

Cele două numere se vor scrie, după ce conținuturile lor s-au inversat, în fişierul de ieşire **INVERS.OUT**.

Restricții şi precizări

- $0 \le a, b \le 1000000000$.

Exemplu

INVERS.IN	**INVERS.OUT**
10 4	4 10

17.7. Soluțiile problemelor propuse

17.7.1. Afişare în baza 2

Pentru a afişa reprezentarea binară a numărului va trebui să testăm pe rând biții acestuia. Există mai multe modalități de a face acest lucru. Prezentăm în continuare două dintre ele:

a. Soluție care roteşte masca

Pentru a testa toți biții va trebui să repetăm de 8 ori testarea (dacă numărul este reprezentat pe 8 biți), la fiecare pas deplasând bitul de test cu o poziție. De data aceasta vrem să afişăm valoarea biților, începând cu bitul 0, deci masca se va inițializa cu valoarea:

$10000000_2 = 2^7 = 128_{10}$.

La fiecare pas rotim masca la dreapta cu o poziție.

Exemplu

$40_{10} = 00101000_2$

i	*nr*	*masca*	*nr* **and** *masca* = *rezultat*	*afișăm*
1	**0**0101000	**1**0000000	00000000	0
2	0**0**101000	0**1**000000	00000000	0
3	00**1**01000	00**1**00000	00100000	1
4	001**0**1000	000**1**0000	00000000	0
5	0010**1**000	0000**1**000	00001000	1

Pe următoarele trei poziții afișăm 0. Deci pe parcursul executării programului se afișează: 00101000.

b. Soluție bazată pe rotirea numărului dat

În această abordare, în loc să se modifice poziția bitului de test din mască se va roti numărul dat bit cu bit, testându-se la fiecare pas aceeaşi poziție (primul bit). Rotirile şi testările se vor repeta de atâtea ori câți biți are reprezentarea numărului *n*. Cifrele 0 din fața numărului nu vor fi afișate.

Exemplu

Fie *n* = 40 şi *masca* = 128 reprezentate pe 8 biți.

La fiecare pas se rotește numărul *n* la stânga pentru ca următorul bit (bitul de pe poziția 1) să ajungă în poziția de testare.

În programul cu care implementăm acest algoritm, am introdus o variabilă logică *cifră_semnificativă* care va primi valoarea *adevărat* după găsirea şi afişarea primei cifre semnificative, astfel oferind posibilitatea de a evita afişarea cifrelor egale cu 0 în fața primei cifre semnificative.

Pas	*nr*	*masca*	*nr* **and** *masca* = *rezultat*	*cifră_semnificativă*	*afișăm*
1	00101000	10000000	00000000	*fals*	
2	01010000	10000000	00000000	*fals*	
3	10100000	10000000	10000000	*adevărat*	1
4	01000000	10000000	00000000	*adevărat*	0
5	10000000	10000000	10000000	*adevărat*	1

În următorii trei paşi se va afişa valoarea 0.

```
Algoritm Afişare:
  citeşte n
  masca ← 1              { se construieşte o mască pe 32 de biți care va avea }
  masca ← masca shl 31                    { primul bit 1, restul biților fiind 0}
  cifră_semnificativă ← fals
```

```
  pentru i=1,32 execută:
    dacă n and masca ≠ 0 atunci
      scrie 1
      cifră_semnificativă ← adevărat
    altfel
      dacă cifră_semnificativă atunci
        scrie 0                    { 0-urile nesemnificative nu se vor afişa }
      sfârşit dacă
      n ← n shl 1                  { rotim numărul cu o poziție la stânga }
    sfârşit dacă
  sfârşit pentru
sfârşit algoritm
```

17.7.2. Sumă de puteri ale lui 2

În mod asemănător problemei anterioare se vor testa biții numărului *n*, de data aceasta însă aceştia vor fi parcurşi de la dreapta spre stânga ceea ce ne va permite să păstrăm într-un contor (*pas*) rangul fiecărui bit. Atunci când un bit va avea valoarea 1, contorul va arăta puterea lui 2 care corespunde acelui bit.

Exemplu
$n = 40_{10} = 00101000_2$

Pas	*n*	*Ultimul bit*	*Afişăm*
0	00101000	0	
1	00010100	0	
2	00001010	0	
3	00000101	1	2^3
4	00000010	0	
5	00000001	1	+2^5
6	00000000		

Se observă că la fiecare pas a fost testat ultimul bit, ceea ce ne arată că masca va trebui să aibă ultimul bit egal cu 1, iar ceilalți biți egali cu 0. Deci masca va fi inițializată cu 1.

În algoritmul corespunzător vom folosi variabila booleană *primul* care ne va arată dacă suntem sau nu la primul termen al sumei pentru a şti dacă vom afişa un + sau nu.

```
Algoritm Descompunere_după_puterile_lui_2:
  citeşte n
  masca ← 1
  pas ← 0
```

```
  primul ← adevărat
  cât timp n ≠ 0 execută:        { cât timp mai există biţi egali cu 1 în număr }
    dacă (n şi masca) ≠ 0 atunci          { dacă am găsit un bit egal cu 1}
      dacă nu primul atunci
        scrie '+'
      altfel
        primul ← false
      sfârşit dacă
      scrie 2^pas                  { afişăm puterea lui 2 corespunzătoare }
    sfârşit dacă
    n ← n rotit la dreapta cu 1 poziţie          { trecem la următorul bit }
    pas ← pas + 1
  sfârşit cât timp
sfârşit algoritm
```

17.7.3. Inversare

Vom folosi operatorul **xor**, care, aplicat de două ori consecutiv îşi anulează „acţiunea”. Dacă se aplică între cele două numere întregi date, apoi asupra rezultatului cu primul umăr şi la fel cu al doilea, conţinutul celor două variabile se inversează.

Fie $a = 10_{10}$ şi $b = 11_{10}$.

a	1	0	1	0
b	1	0	1	1
a **xor** b	0	0	0	1

⇒

b	1	0	1	1
a	0	0	0	1
b **xor** a	1	0	1	0

⇒

a	0	0	0	1
b	1	0	1	0
a **xor** b	1	0	1	1

Deci avem în *b* valoarea 10_{10}, iar în *a* valoarea 11_{10}.

```
Algoritm Inversare:
  citeşte a,b
  a ← a xor b
  b ← b xor a                  { valoarea primului număr ajunge în al doilea }
  a ← a xor b                  { valoarea celui de al doilea număr ajunge în primul }
  scrie a
  scrie b
sfârşit algoritm
```