Considerații teoretice

Formatori:
Tutor: Stângaciu Valentin
Tutor: Belu Claudiu-Marcel
+3

Data de începere a cursului:
25.09.2023
Utilizatori înscriși
Calendar
Note

Considerații teoretice

Operații în baza de numerație 2

Pentru a înțelege mai ușor cum se execută operațiile în diverse baze de numerație, vom analiza prima oară cum se fac aceste operații în baza 10, cu care suntem mai familiarizați.

Adunarea

În baza 10 considerăm adunarea a două numere, a=748 și b=459:

Transport de la rangul inferior (carry)	1 1 1
a	0 7 4 8
b	0 4 5 9
a+b	1 2 0 7

Algoritmul este următorul:

începem de la dreapta la stânga, adunând toate rangurile corespunzătoare din cele două numere

dacă rezultatul este mai mare sau egal cu baza (10), scădem baza din rezultat și pentru rangul respectiv reținem rezultatul scăderii. În această situație, la rangul imediat superior vom mai avea de adunat 1, deoarece 10 unități de la un rang fac cât o unitate la rangul imediat superior. Acest 1 se numește **transport** (carry) și apare când rezultatul depășește domeniul numeric al bazei respective (0...9 pentru baza 10).

Dacă un număr are mai multe cifre decât celălalt, sau dacă apare transport după ce s-au epuizat toate cifrele unui termen, se mai adaugă la dreapta zerouri, până când se termină de adunat toate cifrele, inclusiv transportul

Conform aceluiași algoritm, vom aduna două numere în baza 2:

$$
\begin{array}{ll}
011010011 + & = 2^7+2^6+2^4+2^1+2^0 = 211_{10} \\
010010000 & = 2^7+2^4 = (211)_{10} = 144_{10} \\
\hline
101100011 & = 2^8+2^6+2^5+2^1+2^0 = 355_{10}
\end{array}
$$

Deoarece cifrele disponibile în baza 2 sunt 0 și 1, când s-au adunat cifrele (digiții) rangului 4 (1+1=2), valoarea 2 a depășit acest interval de valori. În această situație s-a scăzut baza și a rămas 0, rezultând în același timp un transport la rangul superior.

Atenție: după cum se observă, pentru a memora rezultatul avem nevoie de 9 cifre. Pe hârtie putem să mai adăugăm o cifră, dar în calculator cea de-a 9-a cifră va fi pierdută dacă numărul nostru ocupă doar 1 octet. De aceea trebuie avut grijă ca rezultatul unei operații să se încadreze în memoria disponibilă pentru acel tip de date. Acesta este doar un exemplu de depășire a memoriei alocate. O sa vedem mai încolo că depășire putem avea și în alte cazuri.

Scăderea

În baza 10 considerăm scăderea numerelor a=748 și b=459:

Împrumut (barrow) de la rangul superior		+10	+10
Reducere datorată împrumutului către rangul inferior	-1	-1	
a (descăzut)	7	4	8
b (scăzător)	4	5	9

a-b	2 8 9

Algoritmul este următorul:

începem de la dreapta la stânga, scăzând rangurile corespunzătoare din cele două numere

Dacă scăzătorul este mai mare decât descăzutul, vom împrumuta (barrow) o unitate de la rangul imediat superior al descăzutului. În această situație, din rangul imediat superior vom scădea 1 (valoarea împrumutată) și la rangul curent vom aduna 10 (baza), deoarece o unitate din rangul imediat superior face cât 10 unități (baza) ale rangului curent.

Dacă descăzutul are mai puține cifre decât scăzătorul, îi vom mai adăuga zerouri la stânga, astfel încât fiecare cifră a scăzătorului să aibă un zero corespondent în descăzut, iar apoi vom mai adăuga și un 1 după toate aceste zerouri. Acest 1 va dispărea fiind împrumutat către rangurile inferioare și rezultatul va fi negativ.

La scăderea în baza 2 se aplică același algoritm, cu diferența că un 1 împrumutat de la rangul imediat superior este echivalent cu 2 unități (baza) la rangul curent.

Înmulțirea

În baza 10 considerăm înmulțirea numerelor a=748 și b=459:

a	**7 4 8**	*Nr. deplasări*
b=4*10^2+5*10^1+9*10^0	**4 5 9**	
a*9*10^0	6 7 3 2	*0*
a***5*10^1**	3 7 4 0 **0**	*1*
a*4*10^2	2 9 9 2 **0 0**	*2*
a*b = a*9*10^0 + a***5*10^1 + + a*4*10^2**	3 4 3 3 3 2	

Algoritmul este următorul:

îl considerăm pe *b* ca fiind scris pozițional, fiecare cifră a sa fiind înmulțită cu puterea lui 10 (baza) corespunzătoare rangului său; *b* devine astfel o sumă de termeni.

Înmulțim pe *a* cu fiecare cifră din *b*, de la dreapta la stânga și deplasăm la stânga rezultatul cu un număr de poziții corespunzătoare rangului său. Efectuăm această deplasare deoarece înmulțirea cu 10 (baza) este echivalentă cu o deplasare la stânga a numărului cu o poziție și inserarea unui 0 în dreapta.

Adunăm toate rezultatele parțiale rezultate din înmulțirile anterioare

În baza 2 înmulțirea este chiar mai simplă, deoarece singurele înmulțiri cu câte o cifră sunt fie cu 0 și atunci rezultatul este 0, fie cu 1 și atunci rezultatul este chiar numărul inițial:

1001 *
1101

1001 +
0000**0**
1001**00**
1001**000**

1110101

Împărțirea

Împărțirea se definește ca fiind operația aritmetică prin care se determină de câte ori un număr (împărțitorul) este cuprins în altul (deîmpărțitul). Pentru a menține lucrurile simple, împărțirea o vom efectua prin scăderi repetate. Pentru aceasta, atâta timp cât deîmpărțitul este mai mare decât împărțitorul, vom scădea împărțitorul din deîmpărțit și vom număra de câte ori am efectuat această scădere. Numărul de scăderi efectuate este rezultatul împărțirii (câtul), iar valoarea rămasă din deîmpărțit este restul.

Exemplu în baza 2: 1011 / 0101

valoare_rămasă_din_deîmpărțit - împărțitor	**Rezultat scădere**	**Nr. operații**
1011-0101	0110	1
0110-0101	0001	2

Rezultatul (câtul) împărțirii este numărul de operații de scădere executate: $2_{10} = 0010_2$

Restul împărțirii este ultima valoare rămasă din deîmpărțit: 0001_2

Reprezentarea numerelor cu semn în complement de 2

Există mai multe metode de reprezentare pe biți a numerelor întregi cu semn, dar cea mai des întâlnită este cea în complement de 2. Pentru a determina biții unui număr negativ în complement de 2, ne folosim de egalitatea **x+(-x)=0**, adică suma unui număr cu opusul său trebuie să fie 0. Dacă rezolvăm această egalitate în funcție de **-x**, obținem **-x=0-x**, adică pentru a obține opusul unui număr pozitiv, trebuie să scădem acel număr pozitiv din 0, operație pe care deja am discutat-o.

Pentru numere cu semn, întotdeauna se va specifica numărul de biți N pe care reprezentăm numerele. Implicit, considerăm numere de 8 biți (un octet).

Exemplu: să se reprezinte în binar numărul -47_{10}

Pentru aceasta va trebui să efectuăm în binar, pe 8 biți, scăderea 0-47:
1 0000 0000 -
0010 1111 47_{10}

1101 0001 => complementul de 2 al lui 47 (209_{10})

Proba: 00101111+11010001=100000000 (9 biți). Deoarece numerele se stochează pe câte 8 biți, considerăm doar primii 8 biți cei mai puțin semnificativi => 00000000, deci numerele sunt complementare

Conform celor discutate la operația de scădere, pentru a scădea un număr din 0 a trebuit să mai adăugăm la stânga lui 0 un 1, ca să avem din ce împrumuta. Acest bit de 1 nu există fizic în memorie, deoarece avem doar 8 biți disponibili, ci el este un auxiliar care ne permite să facem scăderea.

În baza 10 numărul 100000000 este 2^8=256. Astfel, putem determina complementul de 2 al unui număr direct din baza 10, fără să mai trebuiască să-l transformăm în binar. Pentru aceasta scădem numărul respectiv din 256. Reciproc, pentru ca două numere pe un octet să fie opuse în complement de 2, suma lor trebuie să fie 256.

Pentru exemplul anterior, -47 în complement de 2 este: 256-47=209, exact cât ne-a rezultat și în exemplu.

Dacă avem un număr negativ în complement de 2, obținerea numărului pozitiv opus este simetrică, deci mai facem încă o dată complement de 2 din acel număr.

Pentru a determina ce interval de valori putem reprezenta pe un octet în complement de 2, putem să reprezentăm într-un tabel fiecare număr posibil și complementul său. Ne vom opri cu tabelarea atunci când complementul devine mai mic decât numărul original, când de fapt se vor repeta în oglindă numerele din liniile anterioare:

Număr$_{10}$	**Număr$_2$**	**Opus$_{10}$**	**Opus$_2$**
0	0000 0000	0	0000 0000
1	0000 0001	255	1111 1111
2	0000 0010	254	1111 1110
		...	
127	0111 1111	129	1000 0001
128	1000 0000	128	1000 0000
129	1000 0001	127	0111 1111

Constatăm că linia cu numărul 129, care l-ar avea ca și opus pe numărul 127, este de fapt oglindirea liniei anterioare 127 cu 129. Astfel, numărul 129 deja este prea mare pentru a se încadra în numerele pozitive pe un octet (dacă în octet memorăm numere cu semn).

Întrebarea este cum considerăm numărul 128, deoarece și opusul său este tot 128...pozitiv sau negativ? Prin convenție, s-a decis ca toate numerele care au bitul cel mai semnificativ (MSB) 0 să fie considerate pozitive și toate numerele care au MSB 1 să fie considerate negative. Astfel, $128_{10}=10000000_2$, va fi considerat ca fiind negativ. Rezultă că cel mai mare număr pozitiv în complement de 2 reprezentabil pe 1 octet este 127.

Generalizând, dacă avem N biți, intervalul de valori cu semn reprezentabile pe acești biți este: $-2^{N-1}...2^{N-1}-1$

Folosind complementul de 2, putem reduce operația de scădere la una de adunare. Această transformare este utilă când avem de scăzut un număr mai mare (b) din unul mai mic (a). Astfel, a-b devine a+(-b), deci vom face complementul de 2 al lui b, vom face adunarea și în final rezultatul (număr negativ în complement de 2) îl vom transforma în număr pozitiv.

Operatori pe biți

Operatorii pe biți oferă acces direct la biții unui număr. Acești operatori implementează operațiile logice obișnuite NOT (**~**), ȘI (**&**), SAU (**|**), SAU EXCLUSIV (**^**) și deplasări (shift) pe biți, la dreapta (**>>**) și la stânga (**<<**).

Operațiile logice pe biți sunt prezentate în tabelul de mai jos. Pentru a fi mai simplu să se rețină efectul acestora, se poate memora la fiecare dintre ele regula care arată când anume rezultatul este adevărat (1, true).

a b	**~ a** **NOT returnează opusul bitului**	**a & b** **ȘI (AND)** **1 când ambii operanzi sunt 1**	**a | b** **SAU (OR)** **1 când cel puțin un operand este 1**	**a ^ b** **SAU EXCLUSIV (EXOR)** **1 când operanzii sunt diferiți**
0 0	1	0	0	0
0 1	1	0	1	1
1 0	0	0	1	1
1 1	0	1	1	0

Atunci când sunt aplicați unor numere, acești operatori se execută în paralel, pe fiecare dintre biții corespondenți din cele două numere.

Exemplu: Fie numerele **fără semn** a=1001 0101_2 (149_{10}) și b=0101 0110_2 (86_{10}). Cu aceste numere, rezultatele operațiilor pe biți sunt:

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

~ a	0	1	1	0	1	0	1	0
a & b	0	0	0	1	0	1	0	0
a | b	1	1	0	1	0	1	1	1
a ^ b	1	1	0	0	0	0	1	1

Pentru a vizualiza pe calculator biții unui număr, putem folosi funcția *showBits* ca în exemplul de mai jos.

```
#include <stdio.h>

void showBits(unsigned a){
    int i;
    for(i=sizeof(a)*8-1;i>=0;i--)
        printf("%d",(a>>i)&1);
    printf("\n");
}

int main(){
    unsigned a,b;
    printf("a=");scanf("%u",&a);
    printf("b=");scanf("%u",&b);
    showBits(a);
    showBits(b);
    printf("a&b=");showBits(a&b);
    printf("a|b=");showBits(a|b);
    printf("a^b=");showBits(a^b);
    return 0;
}
```

Operatorul **sizeof** evaluează dimensiunea *în octeți* a unei expresii (ex: *sizeof(3.14)* → 8) sau a unui tip (ex: *sizeof(char)* → 1).

Deplasările pe biți au forma **a<<n** pentru deplasare la stânga și **a>>n** pentru deplasare la dreapta. Întotdeauna valoarea a cărei biți se deplasează (*a*) este în partea stângă, iar numărul de biți cu care se face deplasarea (*n*) este în partea dreaptă.

La deplasarea la stânga, **a<<n**, întotdeauna se introduc în locațiile libere rămase în dreapta biți de 0. La deplasarea la dreapta, **a>>n**, dacă numărul de deplasat este fără semn, se introduc biți de 0 în stânga, iar dacă numărul este cu semn, se duplică bitul de semn (MSB) pe biții rămași liberi. **Observație:** A se evita utilizarea operatorului **>>** pe numere cu semn!

```
unsigned char a=0b11010111, b;
b=a<<3;// b=10111000
b=a>>3;// b=00011010

signed char c=0b10110101, d;
d=c<<2;// d=11010100
d=c>>2;// d=11101101
```

La fel ca și în cazul operațiilor aritmetice, operațiile pe biți nu își modifică operanzii (nu acționează asupra lor), ci îi lasă neschimbați. Aceste operații doar returnează rezultatul lor. De aceea, o operație singulară de genul *a<<3;* sau *b^c;* nu are niciun efect în C, ci rezultatul lor trebuie preluat și folosit: *k=a<<3;* sau *printf ("%u", b^c);*

Deplasarea la stânga cu o unitate echivalează cu înmulțirea cu 2 (la fel cum în baza 10, deplasarea la stânga echivalează cu înmulțirea cu 10), iar deplasarea la dreapta echivalează cu împărțirea la 2: 01010011>>3 = 00001010 (83>>3=83/2^3=10). În anumite situații, programatorii, pentru a optimiza operațiile de înmulțire/împărțire cu puteri ale lui 2, le înlocuiesc cu deplasări pe biți, ceea ce pe anumite arhitecturi hardware duce la un spor semnificativ de viteză. Totuși, compilatoarele moderne de C știu să facă automat acest gen de optimizări, așa că în general programatorul nu trebuie să facă în mod explicit această substituție.

Operații fundamentale pe biți

Există o serie de operații fundamentale pe biți, la care pot fi reduse cele mai multe dintre aplicațiile uzuale. Dacă este cunoscut modul de implementare al acestor operații, atunci multe dintre aplicațiile pe biți vor fi mult mai simplu de implementat.

Pentru început vom prezenta câteva dintre proprietățile operațiilor pe biți, proprietăți pe care ne vom baza în continuare. Considerăm că avem un bit *x*, care poate avea orice valoare (0 sau 1). Din tabelul de adevăr prezentat anterior, dacă îl considerăm de exemplu pe *a* ca fiind *x* (toate operațiile sunt comutative), deducem următoarele:

Identitate	Domenii de utilizare

x&0 → 0 x&1 → x	Deoarece ȘI cu 0 dă rezultatul 0, iar ȘI cu 1 păstrează bitul inițial intact, putem folosi ȘI pentru a pune pe 0 anumiți biți, iar pe ceilalți să-i păstrăm neschimbați.
x|0 → x x|1 → 1	Deoarece SAU cu 0 dă ca rezultat întotdeauna bitul inițial, iar SAU cu 1 dă rezultatul 1, putem folosi SAU pentru a pune pe 1 anumiți biți, iar pe ceilalți să-i păstrăm neschimbați.
x^0 → x x^1 → ~x	Deoarece SAU EXCLUSIV cu 0 dă ca rezultat întotdeauna bitul inițial, iar SAU EXCLUSIV cu 1 dă ca rezultat complementul bitului inițial, putem folosi SAU EXCLUSIV pentru a complementa anumiți biți, iar pe ceilalți să-i păstrăm neschimbați.

În operațiile pe biți, dacă avem un anumit număr *a* asupra căruia operăm și folosim un număr *k* pentru a modifica biții lui *a*, *k* se numește *mască (mask)*, iar operația pe biți se numește generic *mascarea* lui a. De exemplu, dacă vrem să păstrăm primii 4 biți LSB din *a* neschimbați, iar pe ceilalți biți să-i punem pe 0, putem să *mascăm* pe *a* folosind *masca* 0b1111: *a=a&0b1111;*

Folosind identitățile de mai sus, vom putea în continuare să implementăm operațiile fundamentale pe biți. Fie un număr *a* și fie *n* indexul unui bit din *a*, asupra căruia dorim să operăm. Indexul *n* se consideră pornind de la LSB (de la dreapta la stânga), cu bitul LSB având indexul 0.

Operație	Formulă	Explicații
testare	**a&(1<<n)** sau **(a>>n)&1**	Pentru a testa dacă bitul *n* este 0 sau 1, deplasăm la stânga un bit de 1 cu *n* poziții, astfel încât 1 să ajungă pe poziția cerută. Apoi, folosind ȘI, păstrăm valoarea bitului cerut și punem pe 0 pe toți ceilalți biți. Astfel, dacă bitul de testat este 1, vom obține o valoare nenulă, iar dacă el este 0, rezultatul este 0. *A doua metodă*: aducem bitul cerut pe prima poziție, după care cu ȘI punem pe toți ceilalți biți pe 0, păstrându-l doar pe el neschimbat. Astfel, rezultatul final este chiar valoarea bitului de testat.
setare	**a|(1<<n)**	Pentru a seta un bit (punerea sa pe 1), deplasăm la stânga un bit de 1 cu *n* poziții, astfel încât 1 să ajungă pe poziția cerută. Apoi, folosind SAU, punem bitul cerut pe 1, păstrând pe toți ceilalți biți neschimbați.
resetare	**a&~(1<<n)**	Pentru a reseta un bit (punerea sa pe 0), deplasăm la stânga un bit de 1 cu *n* poziții, astfel încât 1 să ajungă pe poziția cerută. Apoi complementăm (NOT) valoarea obținută, astfel încât pe poziția dată să avem 0 și pe toate celelalte poziții 1. În final, prin operația ȘI, pe poziția cerută vom avea 0, iar pe celelalte poziții biții vor rămâne neschimbați.
complementare	**a^(1<<n)**	Pentru a complementa un bit, deplasăm la stânga un bit de 1 cu n poziții, astfel încât 1 să ajungă pe poziția cerută. Apoi, folosind SAU EXCLUSIV, complementăm bitul cerut, păstrând pe toți ceilalți biți neschimbați.

Atenție: deoarece operațiile pe biți nu își modifică operanzii, formulele de mai sus în sine nu îl modifică pe *a*. Pentru a folosi rezultatul operației, putem de exemplu să-l atribuim unei variabile: *a=a|(1<<n);*

Forme compuse de atribuire

În programe apare de multe ori situația de a actualiza valoarea unei variabile pe baza vechii ei valori, de exemplu *a=a+b/2*. Deoarece acest gen de operații este des întâlnit, în C avem o formă mai concisă de a le scrie, combinând atribuirea cu operația de actualizare a variabilei: *a+=b/2*. Există asemenea forme compuse de atribuire pentru aproape toți operatorii binari: **+= -= *= /= %= <<= >>= &= ^= |=**.

Aceste forme compuse de atribuire se pot folosi doar atunci când operația de actualizare a rezultatului este ultima executată în partea dreaptă a atribuirii, altfel nu se mai respectă ordinea operațiilor. De exemplu *a=a/2+b* nu se poate scrie în formă compusă, cum ar fi *a/=2+b*, deoarece această din urmă expresie este echivalentă de fapt cu *a=a/(2+b)*.

Ordinea (precedența) și asociativitatea operatorilor

La fel ca și în matematică, operatorii din C se execută într-o anumită ordine. De exemplu, în expresia *a+b*7*, înmulțirea se execută prima, deși apare după adunare. Această ordine de execuție se numește **precedența** operatorilor. Întotdeauna se vor executa primii operatorii cu precedența cea mai mare. Pentru a se modifica ordinea de evaluare a operatorilor, se folosesc paranteze.

Totodată, pentru operatori se definește și **asociativitatea** acestora. Această proprietate se referă la ordinea de execuție (de la stânga la dreapta → asociativitate *stângă*, sau de la dreapta la stânga → asociativitate *dreaptă*) atunci când avem o secvență de mai mulți operatori cu aceeași precedență. De exemplu, expresia 18/3/2 dacă împărțirea ar avea asociativitate stângă s-ar scrie (18/3)/2→3, iar dacă împărțirea ar avea asociativitate dreaptă, s-ar scrie 18/(3/2)→18.

În tabelul de mai jos se află precedența și asociativitatea operatorilor din C, de la precedența cea mai mare la cea mai mică. Toți operatorii dintr-o celulă au aceeași precedență. Unii dintre acești operatori vor fi predați ulterior, în următoarele laboratoare.

Operator	Nume	Asociativitate

++ --	incrementare/decrementare postfix	stângă
( )	apel de funcție	
[ ]	indexare în vector	
.	accesul membrului unei structuri	
->	accesul membrului unei structuri folosind un pointer	
++ --	incrementare/decrementare prefix	dreaptă
+ -	plus sau minus unar	
! ~	NOT logic și pe biți	
(*tip*)	conversie de tip (cast)	
*	indirectare (dereferențiere)	
&	adresa	
sizeof	dimensiunea în octeți	
* / %	înmulțire, împărțire, modulo	stângă
+ -	adunare, scădere	stângă
<< >>	deplasare pe biți la stânga și la dreapta	stângă
< <= > >=	mai mic, mai mic sau egal, mai mare, mai mare sau egal	stângă
== !=	egal, inegal	stângă
&	ȘI pe biți	stângă
^	SAU EXCLUSIV pe biți	stângă
|	SAU pe biți	stângă
&&	ȘI logic	stângă
||	SAU logic	stângă
?:	operatorul condițional (ternar)	dreaptă
=	atribuire	dreaptă
+= -=	atribuire cu sumă, diferență	
*= /= %=	atribuire cu produs, împărțire, modulo	
<<= >>=	atribuire cu deplasare pe biți la stânga, la dreapta	
&= ^= |=	atribuire cu ȘI, SAU EXCLUSIV, SAU pe biți	
,	virgulă (secvență)	stângă

În general este bine, mai ales dacă lucrăm în echipă cu alți programatori, să punem paranteze pentru a se evidenția ordinea operațiilor mai puțin folosite, chiar dacă aceste paranteze nu sunt necesare. Ele măresc în schimb lizibilitatea programului: *('a'<=c&&c<='z')||('A'<=c&&c<='Z')*

În special când se combină operații pe biți cu operații aritmetice, trebuie avută multă atenție la ordinea operațiilor, altfel putând rezulta erori greu de depistat. De exemplu, dacă un programator rescrie expresia *a+b*2* ca *a+b<<1*, pentru a-i optimiza viteza, programul rezultat va fi greșit, deoarece această din urmă expresie de fapt este echivalentă cu *(a+b)<<1*. Corect ar fi trebuit să se scrie *a+(b<<1)*.

◄ Teme și aplicații

Sari la...

Teme și aplicații ►

Contactați serviciul de asistență

Sunteți conectat în calitate d[redacted]
S1-L-AC-CTIRO1-PC

Meniul meu
Profil