Dvojková pozičná sústava

Dvojková sústava hrá v informatike dôležitú úlohu. Využívajú ju všetky digitálne technológie. Používa ju aj počítač. Aby sme dokázali lepšie porozumieť niektorým princípom práce počítača, budeme sa jej teraz stručne venovať. Jeden z dôvodov, prečo sa dvojková sústava používa**, je jednoduchosť**, s akou sa v nej dajú vykonávať aritmetické operácie a realizovať potrebné elektrické obvody. Dvojková sústava sa od desiatkovej líši len tým, že namiesto 10 cifier používa len dve 0 a 1 a hodnota cifry na pozícii p sa rovná 2^p a nie 10^p.

Prevod medzi číselnými sústavami

Prevod čísel z desiatkovej do dvojkovej sústavy: Jednotlivé cifry čísla v dvojkovej sústave budeme zapisovať odzadu:

1. Zapíšeme zvyšok čísla po delení číslom 2.
2. Číslo vydelíme číslom 2.
3. Pokračujeme 1. krokom, až kým sa číslo nezmenší na nulu.

Na nasledujúcom obrázku je prevod čísla 89 do dvojkovej sústavy. Skúste ho samostatne previesť podľa uvedeného návodu.

2^6 2^5 2^4 2^3 2^2 2^1 2^0

1	**0**	**1**	**1**	**0**	**0**	**1**

Čísla 2 a 10 v dvojkovej , resp. v desiatkovej sústave sa nazývajú **základ**. Keď si pozornejšie všimneme, čo sa v návode na prevod čísla x vlastne deje, vidíme, že nerobíme nič iné, iba zisťujeme, koľko ktorých mocnín základu musíme zobrať, aby sme v súčte dostali x. Aby sme sa vyhli pri zápise čísel nedorozumeniam, akú sústavu používame, základ sústavy zapíšeme k číslu ako pravý dolný index, napr.

$(10)_2 = (2)_{10}$, $(1101)_2 = (13)_2$, $(1011001)_2 = (89)_{10}$.

V prípade, že chceme číslo previesť do inej sústavy, v predchádzajúcom algoritme zameníme číslo 2 za príslušný základ.

Pozrime sa teraz na sčítavanie čísel v dvojkovej pozičnej sústave. Keďže máme iba dve cifry, z ktorých jedna je 0, bude to veľmi jednoduché. Cifra v dvojkovej sústave sa nazýva aj **bit**, používa sa skratka **b.** Osemciferné číslo v dvojkovej sústave sa nazýva **byte**, skratka **B**.

Algoritmus na výpočet súčtu a rozdielu sa podobá na postup, ktorý sa učí na ZŠ. Postupne sa sprava doľava sčítavajú dvojice cifier a prenos zo súčtu predchádzajúcich cifier.

Pri sčítaní čísel v binárnom tvare máme pri prvých cifrách štyri možnosti, ktoré môžu nastať – každá zo sčítavaných cifier môže mať hodnotu 0 alebo 1. Pre všetky vyššie bity máme celkovo osem možností sčítania, pretože potrebujeme rátať aj s prenosom zo sčítania predchádzajúcich cifier. Môžeme si napísať tabuľku všetkých možností pri binárnom sčítavaní.

$prenos_i$	a_i	b	$prenos_i + a_i + b_i$	$prenos_{i+1}$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Vyskúšajme si binárne sčítanie čísel $(118)_{10} = (01110110)_2$ a $(84)_{10} = (01010100)_2$ typu byte. Keď sčítavame dve takéto 8-bitové čísla výsledkom je opäť 8-bitové číslo.

01110110	**118**
+01010100	**+84**
11001010	**202**

Úloha: V programe Lazarus si vyskúšajte súčet dvoch premenných typu byte a overte výsledky sčítaním binárnych reprezentácií týchto čísel na papieri.

Čo sa stane, ak výsledok sčítania presiahne 8 bitov? Ak máme zapnutú kontrolu prekročenia rozsahu (Projekt – Voľby projektu – Generovanie kódu – začiarkneme rozsah), vývojové prostredie nám oznámi chybu. Ak však kontrola prekročenia rozsahu nie je zapnutá, bit, ktorý je mimo rozsah, sa jednoducho ignoruje. Pri 8 bitovom sčítaní sa to deje s 9 bitom t.j. $(100000000)_2 = (2^8)_{10} = (256)_{10}$. To znamená, že ak chceme zistiť, aká bude výsledná hodnota sčítania dvoch 8 bitových čísel v počítači pri vypnutej kontrole prekročenia rozsahu, musíme od matematicky správneho výsledku sčítania odčítať $(256)_{10}$.

1111011O	**246**
+01010100	**+84**
1?01001010	**330**
	-256
	74

Úloha: V programe Lazarus si vyskúšajte súčet dvoch premenných typu byte pri zapnutej kontrole prekročenia rozsahu a pri vypnutej.

Odčítanie čísel v binárnom tvare je veľmi podobné sčítaniu. Opäť máme celkovo 8 možností, ktoré môžu pri odčítaní dvoch cifier nastať, lebo aj pri odčítaní musíme počítať s prenosom do vyššieho

rádu vzniknutom pri odčítaní predchádzajúcich dvoch cifier. Možnosti môžeme opäť vypísať v tabuľke.

$prenos_i$	a_i	b	$prenos_i + a_i + b_i$	$prenos_{i+1}$
0	0	0	0	0
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	1

Vyskúšajme si odčítanie čísel $(118)_{10} = (01110110)_2$ a $(84)_{10} = (01010100)_2$ typu byte.

01110110	**118**
-01010100	**- 84**
00100010	**34**

Čo sa však stane, keď vymeníme poradie, teda od $(84)_{10}$ odčítame $(118)_{10}$? Výsledkom v obore celých čísel je záporné číslo. Ak však pracujeme s nezápornými číslami typu byte, opäť dospejeme k prekročeniu rozsahu. V tomto prípade je výsledkom pri vypnutej kontrole prekročenia rozsahu číslo, ktoré si môžeme vypočítať tak, že k výslednému zápornému výsledku pripočítame $(256)_{10}$.

01010100	**84**
-01110110	**-118**
1?11011110	**-34**
	+256
	222

Úloha: V programe Lazarus si vyskúšajte rozdiel dvoch premenných typu byte pri zapnutej kontrole prekročenia rozsahu a pri vypnutej.

Poznámka:

V Object Pascal-e sa nezáporné čísla reprezentujú ako hodnoty dátových typov byte, word, longword. Premenné dátového typu byte sú schopné uchovávať čísla, ktoré sa dajú reprezentovať v rámci jedného bajtu t.j. 8 bitov. Hodnoty premenných typu word sú reprezentované v dvoch bajtoch (16 bitov) a typu longword v štyroch bajtoch (32 bitov)

Niektoré ďalšie pozičné sústavy

Okrem dvojkovej sústavy sa často používajú v súvislosti s počítačmi šestnástková alebo osmičková sústava. Ich hlavnou výhodou je, že umožňujú rýchly prevod z/do dvojkovej sústavy a poskytujú kompaktnejší zápis. Pretože zápis 8 =2 2·2 = 2^3, 16 =2 2·2·2 = 2^4, ľahko vidíme, že trojice, resp. štvorice dvojkových cifier tvoria jednu cifru v osmičkovej, resp. šestnástkovej sústave. Platí, že

$(000)_2 = (0)_{8.}$ $(001)_2 = (1)_{8.}$ $(010)_2 = (2)_{8.}$ $(011)_2 = (3)_{8.}$

$(100)_2 = (4)_{8.}$ $(101)_2 = (5)_{8.}$ $(110)_2 = (6)_{8.}$ $(111)_2 = (7)_{8.}$

Takže dvojkové číslo stačí rozdeliť na trojbitové kúsky sprava doľava, a keď každý nahradíme zodpovedajúcou cifrou v osmičkovej sústave, máme celé číslo zapísané v osmičkovej sústave. Všimnite si, že na zápis každého čísla nám stačí len tretina symbolov, ako keď je číslo zapísané v dvojkovej sústave. Na druhej strane už nevystačímel en s dvoma symbolmi, ale potrebujeme ich osem. Niekoľko príkladov:

$(10\ 110\ 100)_2 = (264)_8$, $(111\ 011\ 101)_2 = (735)_8$, $(1\ 010\ 011)_2 = (123)_8$

Analogicky postupujeme aj pri šestnastkovej sústave, len teraz dvojkové číslo rozdelíme na štvorbitové kúsky. Platí, že

$(0000)_2 = (0)_{16}$ $(0100)_2 = (4)_{16}$ $(1000)_2 = (8)_{16}$ $(1100)_2 = (C)_{16}$

$(0001)_2 = (1)_{16}$ $(0101)_2 = (5)_{16}$ $(1001)_2 = (9)_{16}$ $(1101)_2 = (D)_{16}$

$(0010)_2 = (2)_{16}$ $(0110)_2 = (6)_{16}$ $(1010)_2 = (A)_{16}$ $(1110)_2 = (E)_{16}$

$(0011)_2 = (3)_{16}$ $(0111)_2 = (7)_{16}$ $(1011)_2 = (B)_{16}$ $(1111)_2 = (F)_{16}$

Dvojkové čísla s predchádzajúceho príkladu zapísané v šestnástkovej sústave

$(1011\ 0100)_2 = (B4)_8$, $(1\ 1101\ 1101)_2 = (1DD)_8$, $(101\ 0011)_2 = (53)_8$

Poznámka:

Existuje podobná funkcia ako **IntToStr**, ktorá prekonvertuje celé číslo do textu. **IntToStr** konvertuje do desiatkovej sústavy, naproti tomu nová funkcia **IntToHex** konvertuje dané číslo do 16-sústavy. Táto funkcia má ešte jeden parameter, ktorý určuje počet cifier výsledku, napr.

IntToHex(97531, 6) = 017CFB,

IntToHex(97531, 4) = 7CFB,

IntToHex(-1, 8) = FFFFFFFF

Reprezentácia záporných čísel v počítači

Aj záporné sú reprezentované ako postupnosť daného počtu bitov. Existuje niekoľko možností:

Znamienkový kód:

Jedna z najprirodzenejších reprezentácií používa najvyšší bit ako znamienkový. Pri 8 bitovom čísle to znamená, že 7 bitov obsahuje absolútnu hodnotu čísla v dvojkovej sústave a najvyšší ôsmy bit (t.j. bit najviac vľavo) reprezentuje znamienko. 0 predstavuje plus a 1 predstavuje mínus. Siedmimi bitmi vieme reprezentovať čísla od $(0)_{10}$ až $(127)_{10}$. Teda rozsah platných čísel v znamienkovom kóde je teda $(-127)_{10}$ s kódom 1111 1111 až $(127)_{10}$ s kódom 0111 1111.

Zaujímavé je, že máme dve možnosti na reprezentáciu nuly: (-0) má kód 1000 0000 a (+0) má kód 0000 0000. Táto vlastnosť komplikuje vyhodnocovanie výsledku porovnania hodnôt dvoch výrazov, ktoré sa robí ako vyhodnotenie ich rozdielu. V prípade nejednoznačnej reprezentácie musíme obvykle testovať dvakrát. Znamienkový kód je výpočtovo náročný pri aritmetických operáciách, kde sa mení znamienko.

Inverzný kód:

Inverzný kód je založený na tom, že záporné číslo reprezentujeme tak, že zapíšeme kladné číslo v dvojkovej sústave a zmeníme v ňom všetky bity na opačné. Táto zmena sa jednoducho vykoná negáciou všetkých bitov reprezentácie čísla v dvojkovej sústave. Napr. číslo $(6)_{10}$ má kód 0000 0110 a číslo $(-6)_{10}$ má kód 1111 1001. Podobne ako v znamienkovom kóde máme dve nuly (-0) s kódom 1111 1111 a (+0) s kódom 0000 0000. Rozsah platných čísel inverzného 8 bitového kódu je $(-127)_{10}$ s kódom 1000 0000 až $(127)_{10}$ s kódom 0111 1111.

Práve existencia dvoch kódov pre nulu spôsobuje problémy pri sčítavaní. Ak sčítanie dvoch čísel spôsobí prekročenie rozsahu, je potrebné k výsledku pripočítať 1 tzv. prenos (carry bit), aby sme mali správny výsledok v inverznom kóde.

1111 1100 t.j. $(-3)_{10}$
+ 0000 0010 t.j. $(2)_{10}$
1111 1110 t.j. $(-1)_{10}$

0000 0011 t.j. $(3)_{10}$
+ 1111 1101 t.j. $(-2)_{10}$
10000 0000 t.j. $(-1)_{10}$
+**1** (prenos)
0000 0001 t.j. $(-1)_{10}$

Rozdiel sa realizuje tak, že druhému z čísel, menšiteľu, zmeníme všetky bity na opačné a namiesto rozdielu urobíme súčet.

Doplnkový kód:

Doplnkový kód sa používa na reprezentáciu celých čísel takmer vo všetkých typoch počítačov. Vychádza z inverzného kódu. Rozdiel oproti inverznému kódu spočíva v tom, že ku kódom záporných čísel v inverznom kóde pripočítame jednotku. Napr. ak chceme reprezentovať číslo $(-6)_{10}$ v doplnkovom 8 bitovom kóde, v binárnej reprezentácií čísla $(6)_{10} = (0000\ 0110)_2$ zmeníme bity na opačné t.j. 1111 1001 a pripočítame 1. Dostaneme doplnkový kód čísla $(-6)_{10}$: 1111 1010.

Doplnkový kód obsahuje iba jeden kód pre nulu a to 0000 0000. Obrovskou výhodou doplnkového kódu, na rozdiel od predchádzajúcich kódov pre celé čísla, je to, že po sčítaní alebo odčítaní máme okamžite správny výsledok v doplnkovom kóde. Pokiaľ pri sčítaní alebo odčítaní došlo k prekročeniu rozsahu dĺžky reprezentácie (napr. ak sčítaním dvoch 8 bitových kódov dostaneme 9 bitové číslo), môžeme prenos (carry bit) ignorovať, pretože bez neho máme správny výsledok. Rozsah platných čísel reprezentovaných doplnkovým 8 bitovým kódom je $(-128)_{10}$ s kódom 1000 0000 až $(127)_{10}$ s kódom 0111 1111.

1111 1101 t.j. $(-3)_{10}$
+ 0000 0010 t.j. $(2)_{10}$
1111 1111 t.j. $(-1)_{10}$

0000 0011 t.j. $(3)_{10}$
+ 1111 1110 t.j. $(-2)_{10}$
10000 0001 t.j. $(-1)_{10}$
- prenos ignorujeme

Ak výsledok aritmetickej operácie sa nedá reprezentovať daným počtom bitov hovoríme o prekročení rozsahu platných čísel. Táto situácia sa dá zistiť tak, že sčítaním dvoch kódov kladných čísel nám vznikne kód záporného čísla, respektíve sčítaním kódov dvoch záporných čísel vznikne kód kladného čísla.

Na záver si zosumarizujeme typy premenných v Object Pascale, ktorých premenné uchovávajú celé čísla.

Typ	Binárny rozsah (doplnkový kód)	Dekadický rozsah
shortint	1000 0000 až 0111 1111	-2^7 až 2^7-1 (-128 až 127)
smallint	1000 0000 0000 0000 až 0111 1111 1111 1111	-2^{15} až $2^{15}-1$
integer	1000 0000 0000 0000 0000 0000 0000 0000 až 0111 1111 1111 1111 1111 1111 1111 1111	-2^{31} až $2^{31}-1$

Reálne čísla

Pri riešení úloh pomocou počítača sa veľmi často stretávame s úlohami, v ktorých potrebujeme pracovať buď s desatinnými číslami, alebo s celými číslami, ktoré presahujú povolený rozsah pre **integer**. Na toto nám pascal ponúka ďalší typ – nazýva sa typ reálne číslo (desatinné číslo) a má meno **Real**. Tento typ má veľmi podobné vlastnosti ako celočíselný typ **Integer**, ale má aj svoje špecifiká. V prvom rade by sme si mali uvedomiť, že niekedy nám môžu vzniknúť nie úplne presné výsledky – hovoríme, že reálna aritmetika má chyby. Preto túto aritmetiku používame obozretne, len ak naozaj musíme a s jej pritom počítame.

Vlastnosti reálnych (desatinných čísel):

- konštanty obsahujú desatinnú bodku a/alebo exponenciálnu časť, napr. 3.14, -0.5, 3000000000.0, 3E9, 4.7E-3
- exponent môže byť približne v rozsahu
- presnosť výpočtov je približne na 15 desatinných miest, ale niektoré operácie nie sú až také presné, ako by sme si mysleli z matematiky
- premenné typu **Real** zaberajú v počítači 8 B (dvojnásobne viac ako **Integer**)
- základné operácie sú +, -, *, /, pričom, ak jeden z operandov je celé číslo, automaticky sa konvertuje na reálne, napr. ak zapíšeme 1 + 1.5 bude sa počítať 1.0 + 1.5, 1/3 – bude sa počítať 1.0/3.0
- automatická konverzia sa robí aj pri priradení: ak do reálnej premennej chceme priradiť celé číslo, toto sa automaticky konvertuje na reálne napr. RCislo := 2*5 sa prekonvertuje na RCislo := 10.0
- opačné priradenie (do celočíselnej premennej reálnu hodnotu) nie je povolené!. Ak také niečo potrebujeme musíme použiť niektorú zaokrúhľovaciu funkciu
- fungujú porovnávania <, <=, >, >=, <>, =
- existuje množstvo štandardných funkcií, napr:
    - Sqrt – odmocnina
    - Sqr – druhá mocnina
    - Sin, Cos, Tan – trigonometrické funkcie
    - Abs – absolútna hodnota
    - Round – zaokrúhľovanie
    - Trunc – odtrhne desatinnú časť – vráti celé číslo
    - FloatToStr – vráti reťazec
    - StrToFloat – z reťazca spraví celé číslo
- reálny typ nemôžeme použiť
    - ako riadiacu premennú for – cyklu (môžeme to zapísať while cyklom)
    - ako hodnotu v podmienenom príkaze case

Úloha: Sčítajte 10 čísel s hodnotou 0,1 a pomocou podmieneného príkazu či sa to rovná jednej vypíšte o tom správu do Mema.

Riešenie: Tento program najprv 10-krát pripočíta k reálnemu číslu hodnotu 0.1. Potom toto číslo vypíše pomocou FloatToStr do textovej plochy: môžeme vidieť správny výsledok 1. Lenže tento výsledok nie je naozaj jedna: ak ho v nasledovnom podmienenom príkaze porovnáme s hodnotou 1, dozvieme sa, že tento test má hodnotu **False**. Ak ale vypíšeme, aký je rozdiel

medzi týmto číslom a 1 dostávame zaujímavé číslo 1.11022302462516E-16, ktoré je veľmi malé (asi 0.0000000000000001), ale stačí na to, aby sa to číslo nerovnalo 1. Keďže programátori vedia o tomto probléme, častejšie ako rovnosť testujú, či je rozdiel dostatočne malý.

Prečo je to tak? Doteraz sme robili prevody do dvojkovej sústavy iba pre celé čísla. V prípade racionálnych čísel a reálnych čísel je potrebné robiť prevod aj časti čísla za desatinnou čiarkou.

Celé časti sa počítali postupným delením dvojkou. Celé číslo v desiatkovej sústave prevedieme do dvojkovej sústavy tak, že toto číslo postupne celočíselne delíme dvomi a zapisujeme si zvyšky po delení sprava doľava až pokiaľ výsledok delenia nie je nula.

Ak potrebujeme spraviť prevod ajv časti čísla za desatinnou čiarkou, namiesto delenia dvojkou potrebujeme dvojkou násobiť – teraz však v obore reálnych čísel. Ak po násobení dostaneme číslo väčšie ako 1, zapíšeme si nakoniec binárneho čísla 1 a túto jednotku od čísla odrátame. Ak po násobení dostaneme číslo menšie ako 1, iba si zapíšeme 0 na koniec binárneho čísla. Výpočet robíme dovtedy, pokiaľ násobením dvojkou nedostaneme výsledok práve 1. V tomto prípade už iba dopíšeme túto jednotku na koniec binárneho čísla.

29 : 2 = 14 zvyšok 1
14 : 2 = 7 zvyšok 0
7 : 2 = 3 zvyšok 1
3 : 2 = 1 zvyšok 1
1 : 2 = 0 zvyšok 1
Z toho vyplýva $(29)_{10} = (11101)_2$
0,15625 · 2= 0,3125
0,3125 · 2= 0,625
0,625 · 2= 1,25
0,25 · 2= 0, 5
0, 5 · 2= 1
Z toho vyplýva $(0, 15625)_{10} = (0, 00101)_2$
Ako je to s číslom 0,1?
Na rozdiel od prevodu zápisu celej časti čísla v desiatkovej sústave do zápisu v dvojkovej sústave, algoritmus prevodu časti za desatinnou čiarkou nemusí vždy skončiť. Napríklad, ak prevádzame do dvojkovej sústavy číslo 0,1 zistíme, že v dvojkovej sústave má toto číslo nekonečný rozvoj. Z toho napr. vyplýva, že číslo 0,1 nevieme v dvojkovej sústave reprezentovať presne.
0,1 · 2 = 0,2
0,2 · 2 = 0,4
0,4 · 2 = 0,8
0,8· 2 = 1,6
0,6 · 2 = 1,2
0,2 · 2 = 0,4
...
Z toho vyplýva $(0,1)_{10} = (0,0001100110011...)_2$
Teda niektoré aj pekne vyzerajúce konštanty, napr. 0,1, sú v počítači reprezentované približnou hodnotou. Preto spočítavanie takýchto približných hodnôt niekedy nedá očakávaný výsledok.