B33POS: Počítačové systémy

Lekce 03. Informace a kódování

Pavel Píša
pisa@fel.cvut.cz

Petr Štěpán
stepan@fel.cvut.cz

4. října, 2025

Obsah

Motivace

Co je informace? Lze ji nějak popsat nebo definovat?

Nejobecnější definice informace je získávání údajů o reálném světě.

Informace snižuje nejistotu – neurčitost znalosti – o svém okolí.

Člověk získává informace svými smysly – nejvíce zrakem, sluchem, hmatem, ale také čichem a chutí.

Motivace

Co je nejistota?
Nejistota o dění ve světě je velmi komplexní a těžko vyčíslitelná. Názorněji si to můžeme představit, na výběru jedné karty z mariášových karet. Pokud máme jednu kartu z balíčku, tak to může být libovolná karta z 32 různých existujících karet. Každá karta má stejnou pravděpodobnost rovnou $\frac{1}{32}$.
Pokud získáme nějakou informaci, jak to ovlivní nejistotu o kartě:

informace	počet možných karet	pravděpodobnost
barva karty je zelená	8	1/8
karta není eso	28	1/28
karta je sedma	4	1/4
karta je červené eso	1	1/1

Která informace nejvíce sníží nejistotu, tedy zvýší pravděpodobnost správného výsledku?

Kvantifikace informace

Jak kvantifikovat informaci?
Například podle počtu ano/ne otázek, které nás povedou k získání praděpodobnosti 1.
Jak se tedy můžeme ptát, jakou kartu držíte?
Můžeme se ptát:

- Je karta srdce? Je karta kule? Je karta žaludy? Je karta zelená?
- Je karta sedmička? Je karta osmička? ... Je karta eso?

Tyto otázky nejsou moc efektivní, pokud jsou všechny karty stejně pravděpodobné. V nejhorším případě se musíme zeptat 12 krát v nejlepším 2 krát.
Potřebujeme otázku Je karta zelená? a Je karta eso?

Kvantifikace informace

Kolik průměrně potřebujeme otázek, pokud jsou všechny karty stejně pravděpodobné?

- Pro srdcovou sedmičku máme 2 otázky, pro srdcovou osmičku 3, ... pro srdcového krále 8 a pro srdcové eso také 8 (odpověď ne na poslední otázku)
- Pro kulovou sedmičku máme 3 otázky, pro kulovou osmičku 4, ... pro kulového krále 9 a pro kulové eso také 9
- Pro žaludovou sedmičku 4 otázky, ... pro žaludové eso 10 otázek.
- Pro zelenou sedmičku 4 otázky, ... pro zelené eso 10 otázek.

Nejhorší případ je tedy 10 otázek a průměrný počet otázek je $\frac{220}{32} = 6.875$

Kvantifikace informace

Lze otázky navrhnout lépe:

- Je karta srdce nebo kule?
- podle výsledku předchozí otázky se ptáme rovnou Je karta srdce? nebo Je karta žaludy?
- Je karta menší než spodek?
- podle výsledku Je karta menší než devítka? nebo Je karta menší než král?
- nyní se již můžeme rovnou ptát na jednu hodnotu ze dvou

Tyto otázky jsou nejefektivnější, protože bez ohledu na vybranou kartu vždy po 5 otázkách známe přesně o jakou kartu jde.

Kódování informace

Abychom zakódovali, jaká karta to je, tak kód karty budou odpovědi na otázky:

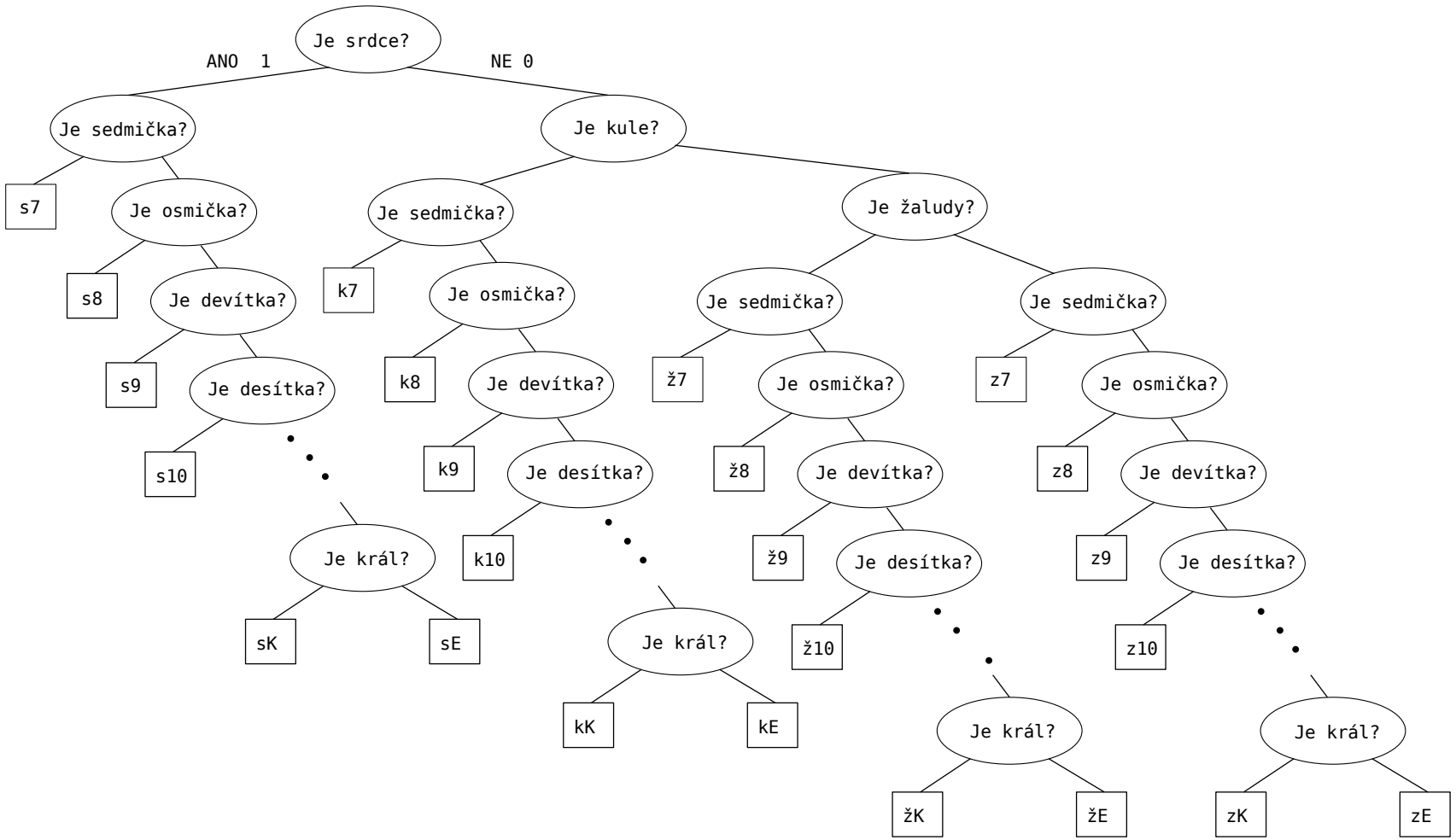
0 - Ne

1 - Ano

Tedy kód třeba žaludové desítky v první sadě otázek je:
0010001
Uměli byste dekódovat v první sadě otázek dvě karty dané tímto kódem:
0000000101000000?
Jak pracovat s takovým kódem, kdy různé karty mají různé délky kódu?
Lze poznat, kde kód končí a začíná druhá, případně další karta?

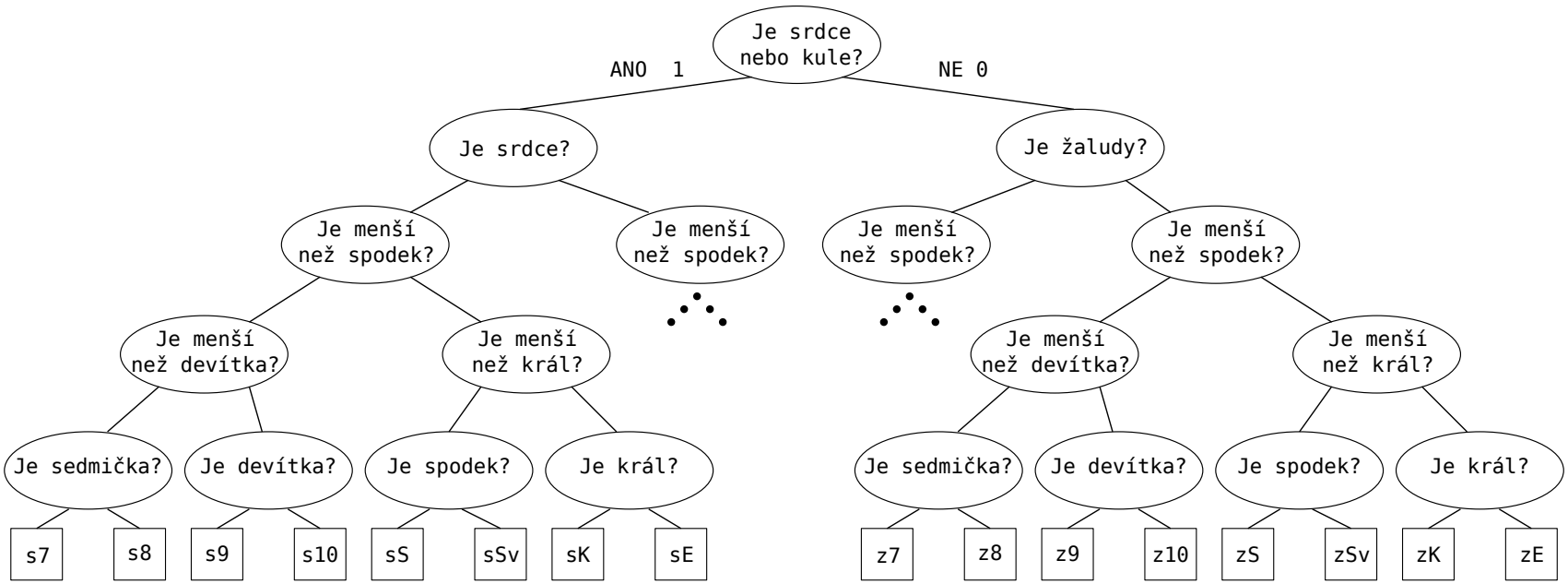
Rozhodovací strom

Můžeme si zkontruovat rozhodovací strom a jakmile se dostaneme do listu stromu, pak víme, že kód skončil a známe kartu:



Rozhodovací strom

Pro optimální sadu otázek bude strom vypadat takto:



Oproti předchozí variantě je tento strom dokonale vyvážen, což znamená že listy jsou pouze na nejnižší úrovni stromu.

Kódování čísel

Pokud chceme zakódovat čísla od 0 do 255 a pokud jsou všechna čísla stejně pravděpodobná je nejlepší kódování s pevnou délkou.
Podobně jako v případě, můžeme navrhnout sadu otázek, která nás dovede nejrychleji k uloženému číslu:

- Je číslo ≥ 128? (pokud ano, odečti pro další otázky 128)
- Je číslo ≥ 64? (pokud ano, odečti pro další otázky 64)
- Je číslo ≥ 32? (pokud ano, odečti pro další otázky 32)
- Je číslo ≥ 16? (pokud ano, odečti pro další otázky 16)
- Je číslo ≥ 8? (pokud ano, odečti pro další otázky 8)
- Je číslo ≥ 4? (pokud ano, odečti pro další otázky 4)
- Je číslo ≥ 2? (pokud ano, odečti pro další otázky 2)
- Je číslo ≥ 1?

Kód daný těmito otázkami je vlastně dvojková soustava.

$v = \sum_{i=0}^{k-1} 2^i b_i$

Velikost bajtu je dána kolik různých čísel můžeme reprezentovat 8 otázkami, 8-mi bity, které reprezentují odpovědi ANO/NE na sadu výše uvedených otázek.

Obsah

Čísla bez znaménka

Reprezentace celých kladných čísel s nulou

V jazyce C jsou tyto typy (podle normy ISO/IEC 9899:TC3):

typ	min	max	počet bajtů
unsigned char	0	255	1
unsigned short	0	65 535	2
unsigned long	0	4 294 967 295	4
unsigned long long	0	18 446 744 073 709 551 615	8

- Norma definuje většinou minimální rozsahy, `unsigned int` nemusí mít jen 2 bajty, ale většinou má 4 bajty (GNU, MS C).
- Pokud chcete zjistit skutečnou velikost, použijte např. příkaz `sizeof(unsigned int)`
- Doporučujeme používat typy `uintX_t` (kde `X` je počet bitů 8, 16, 32, nebo 64), např. `uint8_t`, `uint64_t`
- V některých případech je vhodné použít typy `uint_fastX_t` a `uint_leastX_t`, např. `uint_least8_t`, `uint_fast64_t`

Čísla bez znaménka

Zápis konstant v jazyce C v soustavě:

- desítkové – nesmí začínat 0 kromě 0
- osmičkové – začíná 0
- hexadecimální – začíná 0x
- binární – začíná 0b (pouze GNU překladač)

Příklad: 252 == 0xfc == 0374 == 0b11111100

Poznámka: Podle hexadecimálního zápisu zjistíte velikost čísla v bajtech, např. 0x123456 se vejde do tří bajtů.

Čísla bez znaménka - uložení v paměti

- Paměť počítače pracuje s bajty
- Historicky vzniklo několik možností uložení čísel do paměti.

Jak lze tedy uložit číslo 0x12345678 do paměti:

adresa	Big-endian	Little-endian
400	0x12	0x78
401	0x34	0x56
402	0x56	0x34
403	0x78	0x12

- Procesory Intel zavedly little-endian, procesory Motorola zavedly big-endian.
- V současnosti je 99% procesorů little-endian, dokonce i nové ARM procesory jsou littleendian, nebo umějí little endian nastavit pro práci s čísly
- Je to důležité, když získáte například přes internet data po bajtech, musíte definovat, jaká čísla reprezentují
- RISC V - little-endian, MIPS - big-endian
- Bitcoin - DER signatures big-endian, transaction hash little-endian

Čísla bez znaménka - kvíz

```c
#include <stdio.h>
int main() {
  unsigned char p[] = {0,0,0,0};
  *(unsigned int*)p=10;
  printf("%02x,%02x,%02x,%02x\n", p[0],p[1],p[2],p[3]);
}
```

Co bude výstupem tohoto programu na procesorech Intel?

A nic, program nelze přeložit

B náhodný výstup, p nelze přetypovat na *int

C 0a,00,00,00

D 00,00,00,0a

Obsah

Záporná čísla

Potřebujeme zakódovat znaménko do reprezentace čísla:

- Naivně - vrchní bit bude znaménko
- Výhody
  - Jednoduchá reprezentace
  - Jednoduše spočteme absolutní hodnotu
  - Jednoduše určíme znaménko čísla
- Nevýhody
  - Máme 0 a -0, přitom je to stejné číslo
  - Potřebujeme jiný algoritmus na sčítání, odčítání, násobení a dělení
    - Abychom provedli libovolnou operaci, musíme spočítat absolutní hodnoty obou operandů a pak provést sčítání, odčítání, násobení, dělení
    - Nakonec musíme určit, jaké znaménko bude mít výsledek a toto znaménko nastavit.

Záporná čísla s posunutou nulou

- Zvolíme si o kolik chceme posunout 0.
- Pokud chceme stejný rozsah kladných a záporných čísel, zvolíme pro jeden bajt 127, nebo 128 jako reprezentaci 0.
  - Pro nulu reprezentovanou hodnotou 127 bude rozsah čísel reprezentovaných bajtem $< -127, 128 >$
  - Pro nulu reprezentovanou hodnotou 128 bude rozsah čísel reprezentovaných bajtem $< -128, 127 >$
- Pokud víme, že potřebujeme větší rozsah kladných čísle můžeme zvolit hodnotu 100 jako reprezentaci nuly.
  - Pak bude rozsah čísel reprezentovaných bajtem $< -100, 155 >$

Záporná čísla s posunutou nulou

- pro k-bitovou reprezentaci zvolíme číslo Zero (dále zkráceně Z), které bude reprezentovat nulu
- reprezentace – kód čísla X je definován jako $A(x) = x + Z$
- rozkódování je funkcí $D(a) = a - Z$
- rozsah čísel reprezentovaných je $< -Z, (2^k - Z) - 1 >$

Pro k=8 a Zero=127

Binární hodnota	Hodnota
11111111	$128_{(10)}$
11111110	$127_{(10)}$
...	...
10000001	$2_{(10)}$
10000000	$1_{(10)}$
01111111	$0_{(10)}$
01111110	$-1_{(10)}$
...	...
00000001	$-126_{(10)}$
00000000	$-127_{(10)}$

Záporná čísla s posunutou nulou v jazyce C

```c
#define zero 127

unsigned char code(int i) {
  return (unsigned char)
                    (i+zero);
}

int decode(unsigned char c) {
  return (int)c-zero;
}
```

```c
int main() {
  int a=-20, b=6;
  unsigned char
    code_a = code(a);
  unsigned char
    code_b = code(b);
  printf("a %i zakodovan na %u\
    overeni dekodovani %i\n", a,
    code_a, decode(code_a));
  printf("b %i zakodovan na %u\
    overeni dekodovani %i\n", b,
    code_b, decode(code_b));
  return 0;
}
```

Počítání s čísly s posunutou nulou

- sčítání a odčítání čísel s posunutou nulou se řídí následujícími rovnicemi:
- $A(x+y) = (x+y)+Z = (x+Z)+(y+Z)-Z = A(z)+A(y)-Z$
- $A(x-y) = (x-y)+Z = (x+Z)-(y+Z)+Z = A(x)-A(y)+Z$

```
unsigned code_sum_ab = code_a + code_b - zero;
printf("a+b %i zakodovan na hodnotu %u overeni\
  dekodovani %i\n", a+b, code_sum_ab, decode(code_sum_ab));

unsigned code_sub_ab = code_a - code_b + zero;
printf("a-b %i zakodovan na hodnotu %u overeni\
  dekodovani %i\n", a-b, code_sub_ab, decode(code_sub_ab));
```

Jak lze spočítat násobení?

Počítání s čísly s posunutou nulou

- násobení je ještě složitější:
- $A(x \cdot y) = (x \cdot y) + Z = (x + Z) \cdot (y + Z) - (x + Z + y + Z) \cdot Z + Z^2 + Z =$
  $A(x) \cdot A(y) - (A(x) + A(y)) \cdot Z + Z^2 + Z$

```
unsigned char code_mul_ab = (unsigned char)
  ((unsigned int)code_a * code_b -
  ((unsigned int)code_a+code_b)*zero +
  zero*zero + zero);
printf("a*b %i zakodovan na hodnotu %u overeni\
  dekodovani %i\n", a*b, code_mul_ab, decode(code_mul_ab));
```

A co dělení, můžeme ho spočítat nějak podobně?

Dělení čísel s posunutou nulou

```
unsigned char abs_shifted
        (unsigned char a) {
  if (a>=zero) {
    return a-zero;
  } else {
    return zero-a;
  }
}

unsigned char sign_shifted
        (unsigned char a) {
  if (a>=zero) {
    return 0;
  } else {
    return 1;
  }
}
```

```
unsigned char abs_a =
     abs_shifted(code_a);
unsigned char abs_b =
     abs_shifted(code_b);
unsigned char abs_div_ab =
   (unsigned char)(abs_a/abs_b);

unsigned char code_div_ab;
if (sign_shifted(code_a)==
          sign_shifted(code_b)) {
  code_div_ab = abs_div_ab+zero;
} else {
  code_div_ab = zero-abs_div_ab;
}
printf("a/b %i zakodovan na hodnotu\
  %u overeni dekodovani %i\n",
  (int)(a/b), code_div_ab,
  decode(code_div_ab));
```

Záporná čísla s posunutou nulou

Ukázali jsme si, že záporná čísla s posunutou nulo jsou řešením, ale počítání s nimi není jednoduché.

Metoda použitá pro dělení, tedy počítání s čísly bez znaménka a následné vytvoření reprezentace se znaménkem nebo záporné reprezentace se znaménkem, je ale běžně používanou metodou i pro jiné reprezentace čísel, jak uvidíme později.

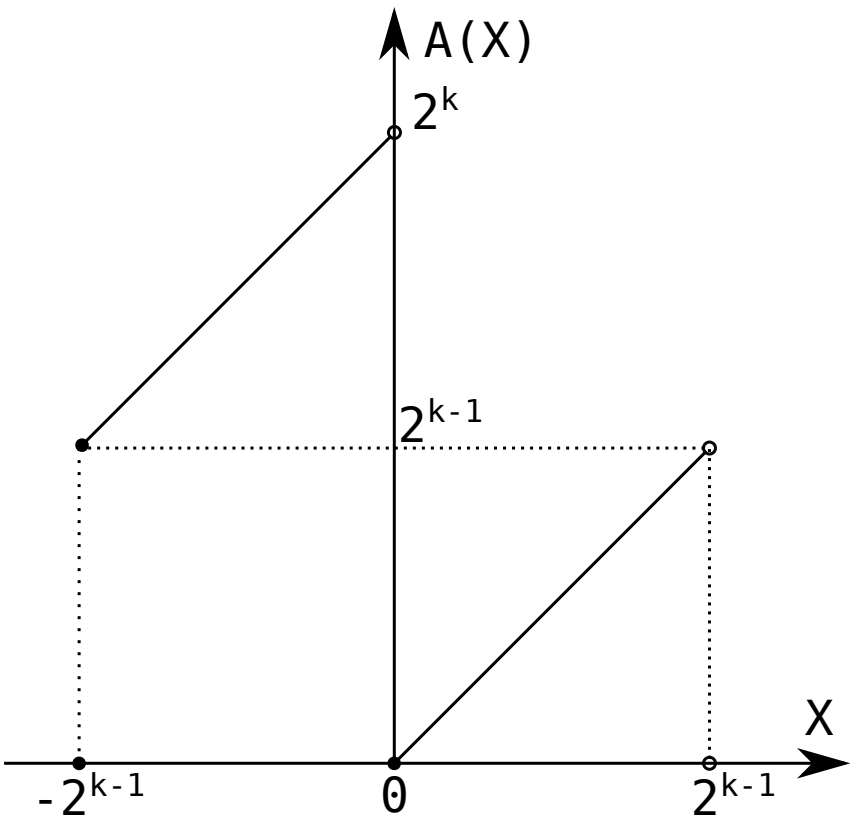
Můžeme tedy vymyslet něco lepšího?

Dvojkový doplněk

- Vybereme si, jaké rozmezí čísel chceme kódovat, podobně jako při posunuté nule, pro k-bitovou reprezentaci čísel např. $< -Z, (2^k - Z) - 1 >$
- Číslo zakódujeme do dvojkového doplňku jako $A(x) = x \,\%2^k$.
  - Výsledkem funkce modulo $(x \,\%y)$ je číslo z rozsahu $< 0, y)$, tedy z rozsahu $< 0, 2^k)$, to jsou všechna čísla, které lze reprezentovat k bity.
  - Funkce modulo vrátí číslo, které je nutné od zadaného čísla odečíst, abychom dostali násobek čísla y.
  - Můžete si to zkusit v Pythonu i v jazyce C/C++, například:
    - hodnota modulo pro všechna kladná čísla x menší než 2^k je rovna x (protože $x - x = 0$ a to je násobek 2^k)
    - hodnota $-1\%2^k$ je vždy $2^k - 1$, protože $-1 - (2^k - 1) = (-1 - 2^k) + 1 = -2^k$ a číslo -2^k je násobek 2^k
- Prakticky se vždy pro k-bitovou reprezentaci čísel volí $Z = 2^{k-1}$ tedy pro 8-mi bitovou reprezentaci čísel máme rozsah $< -128, 127 >$
  - Výhodou volby tohoto intervalu je to, že všechna záporná čísla mají nastevený nejvyšší bit na 1 a všechna kladná čísla a 0 mají nejvyšší bit nastaven na 0

Dvojkový doplněk

- rozsah čísel reprezentovaných k-bity je $< -2^{k-1}, 2^{k-1} - 1 >$
- pro číslo X budeme značit A(X) jeho reprezentaci v dvojkovém doplňku:



Binární hodnota	Dvojkový doplněk
11111111	$-1_{(10)}$
11111110	$-2_{(10)}$
11111101	$-3_{(10)}$
...	...
10000010	$-126_{(10)}$
10000001	$-127_{(10)}$
10000000	$-128_{(10)}$
01111111	$127_{(10)}$
01111110	$126_{(10)}$
...	...
00000001	$1_{(10)}$
00000000	$0_{(10)}$

Početní operace s dvojkovým doplňkem

- Z grafu vlevo jsme viděli, že pokud chceme porovnávat dvě čísla X a Y, tak nestačí porovnat jejich reprezentace, protože záporné reprezentace jsou vštší než kladné
  - Pokud chceme porovnávat čísla X a Y, tak nejdřív zjistíme, zda mají obě čísla stejné znaménko.
  - Pokud mají stejná znaménka pak prostě porovnáme A(X) a A(Y)
  - Pokud mají různá znaménka, tak kladné číslo je větší než záporné
- velká výhoda je, že sčítání i odčítání funguje stejně, jako pro čísla bez znaménka
  - $(X + Y)\%2^k = (X\%2^k + Y\%2^k)\%2^k$
  - důležité je uvědomit si, že operace $\%2^k$ znamená, že výsledek počítáme pouze v k bitech

Příklad:

```
 53= 0b00110101
 -5= 0b11111011
-----------------
304=0b100110000
 48= 0b00110000
```

Příklad:

```
-45=0b11010011
  5=0b00000101
-----------------
-40=0b11011000
```

Opačné číslo – Absolutní hodnota

- Pro výpočet absolutní hodnoty bychom pro záporná čísla potřebovali umět spočítat hodnotu -X.
- Protože sčítání funguje s dvojkovým doplňkem, tak nemusíme navrhovat HW obvod pro odčítání, jednoduše `A-B` vypočteme jako `A+(-B)`
- potřebujeme tedy vymyslet jak získat z `B` hodnotu `-B`
- k výpočtu `-B` bychom chtěli použít jednoduchý obvod, který zneguje každý bit. K tomu pak lze interně použít operaci $XOR\ B, (2^k - 1)$
  - Co je zač číslo $2^k - 1$? To je číslo, které obsahuje samé jedničky, tedy `k` jedniček.
  - pokud znegujeme každý bit, tak je to vlastně matematická operace $-1 - B$, protože $2^k - 1$ je reprezentace čísla -1 a matematická operace $-B$ způsobí negaci všech bitů čísla B.

Opačné číslo

Jak tedy spočítat -x?

$x + (-x) = 0 = 1 + (-1)$

$-x = (-1 - x) + 1$

$-x = neg(x) + 1$

- Výsledný postup je tedy:
  1. znegujte všechny bity čísla B
  2. k výslednému číslu přičti 1

Příklad:

53=0b00110101 znegováním cifer dostaneme

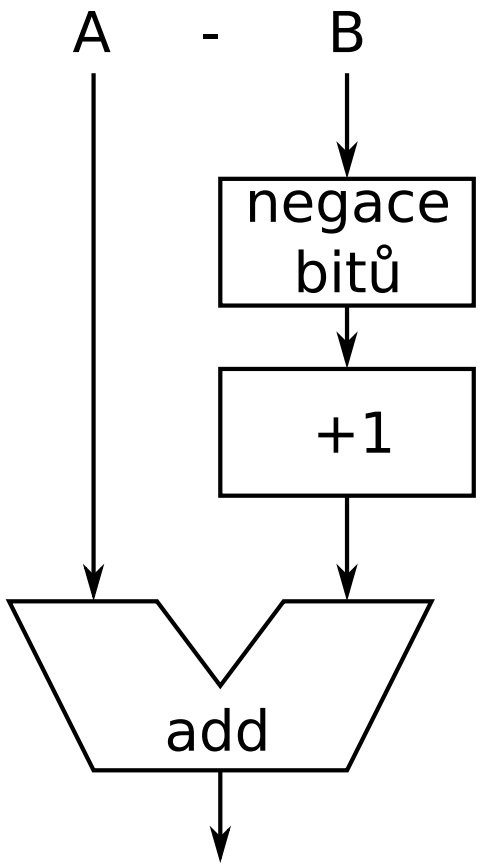
-54=0b11001010 po přičtení 1 pak

-53=0b11001011

Odčítání

Odčítání lze řešit:

- speciálním obvodem podobným sčítačce se všemi možnostmi zrychlení jako byly u sčítání
- nebo z druhého čísla vytvoříme číslo opačné a to pak jednoduše sečteme s tím prvním



Násobení a dělení

- pro násobení lze odvodit, že pokud bychom ignorovali znaménko a spočítali $M \cdot N$, pak pro M,N obě k-bitová čísla musíme výsledek upravit takto:

$$A(M \cdot N) = A(M) \cdot A(N) \begin{cases} -A(M) \cdot 2^k & \text{pokud } M < 0 \\ -A(N) \cdot 2^k & \text{pokud } N < 0 \end{cases}$$

- protože reprezentace záporného čísla dvojkovým doplňkem je vlastně $A(M) = 2^k + M$, pak součin dvou záporných čísel je $(2^k + M) \cdot (2^k + N) = 2^{2 \cdot k} + 2^k \cdot M + 2^k \cdot N + M \cdot N$
- moderní násobení a dělení počítá s absolutními hodnotami a nakonec určíme znaménko výsledku podle znaménka operandů.
  - nejvyšší bit značí znaménko
  - výpočet opačného čísla je rychlý a levný

Čísla se znaménkem

Reprezentace celých čísel

V jazyce C jsou typy:

typ	min	max	počet bajtů
char	-128	127	1
short	-32 768	32 767	2
long	-2 147 483 648	2 147 483 647	4
long long	-9 223 372 036 854 775 808	9 223 372 036 854 775 807	8

- Norma C má ve skutečnosti min o 1 větší , kvůli procesorům s jedničkovým doplňkem – ten se dnes již prakticky nepoužívá
- Stejně tak `int` musíte otestovat, zda je 2 bajtový nebo 4 bajtový
- Doporučujeme používat typy `intX_t` (kde `X` je počet bitů 8, 16, 32, nebo 64), např. `int8_t`, `int64_t`
- V některých případech je vhodné použít typy `int_fastX_t` a `int_leastX_t`, např. `int_least8_t`, `int_fast64_t`

Kvíz

Uvažujte tento program:

```c
#include <stdio.h>
int main() {
  unsigned char a=150u, b=120u, c;
  char sa=-100, sb=-80, sc;

  c=a+b;
  sc=sa+sb;
  printf("c=%u sc=%d\n", c, sc);
}
```

Co program vytiskne:

A c=270 sc=-180

B c=14 sc=-76

C c=14 sc=76

D c=-14 sc=-76

E Numeric error

Přetečení při sčítání čísel bez znaménka

Pokud se podíváme co se stalo, unsigned char je 8-bitová reprezentace čísel:

```
 150 =   1001 0110
+120 =   0111 1000
-----------------
  14 =   0000 1110
 270 =1  0000 1110
```

Protože se výsledek nevejde do 8-bitů, nejvyšší jednička se ztratí a výsledek je pouze 14.

Pokud přetečení chcete detekovat, máte následující možnosti:

- C23 bool ckd_add(type1 *result, type2 a, type3 b) – součet dvou čísel s detekcí přetečení
- GNU GCC 5+, Clang 3.8+ __builtin_add_overflow(a, b, result) – obě verze i pro sub a mul

Přetečení při sčítání čísel se znaménkem

Přetečení při sčítání čísel se znaménkem je složitější. Co je dobře a co je špatně:

```
-112 = 10010000
+ 45 = 00101101
-----------------
 -67 = 10111101
```

DOBŘE

```
  -12 = 11110100
+ -20 = 11101100
-----------------
  -32 =111100000
```

DOBŘE

```
  -90 = 10100110
+ -42 = 11010110
-----------------
  124 =101111100
```

ŠPATNĚ

- Přetečení při sčítání čísel se znaménkem nastane právě tehdy, když dojde k přenosu na posledních cifrách a zároveň nedojde k přenosu na předposledních cifrách:
  - overflow = c_n xor c_{n-1}; c_n je nejvyšší carry, c_{n-1} je druhé nejvyšší carry
- Druhá možnost je sledovat, zda při součtu dvou kladných čísel máme záporná výsledek, nebo při součtu dvou záporných čísel je kladný výsledek:
  - overflow = (a_n and b_n and (not s_n)) or ((not a_n) and (not b_n) and s_n); a_n, b_n jsou nejvyšší bity sčítanců; s_n je největší bit výsledku

Kvíz

Při sčítání dvou čísel s odlišnými znaménky:

A může dojít k přetečení jen v dvojkovém doplňku

B může dojít k přetečení pouze při jiné reprezentaci než je dvojkový doplněk

C nemůže dojít pouze při reprezentaci v dvojkovém doplňku

D nemůže dojít v žádné reprezentaci čísel se znaménkem

Jiné reprezentace záporných čísel

Doplněk jedné:

- záporné číslo je negací bitů opačného čísla
  - pro $X \geq 0$ je reprezentace $A(X) = X$
  - pro $X < 0$ je reprezentace $A(X) = 2^k - 1 - |X|$
- nevýhody: dvě reprezentace 0, složitější sčítání čísel se znaménky

BCD formát

- jiná reprezentace celých čísel, každá cifra jeden nibble (4 bity)
  - číslo 1234 je uloženo v hexadecimálním tvaru jako `0x1234`
- výhody: snadný převod na desítkovou soustavu
- nevýhody: neefektivní reprezentace, složitější počítání

Obsah

Bitové operace

- V jazyce C a ve strojových instrukcích procesoru jsou definovány bitové operace – and, or, xor, negace.
- Pokud máte proměnnou `int a=-10` a chcete zjistit její nejméně významný bajt, pak můžete použít instrukci bitový and `unsigned int bajt = a & 0xff;`
- POZOR neplést s operací `&&`, toto je logický and.
- Jak instrukce pracuje, provede "logický"and mezi sobě odpovídajícími bity čísla:

```
   a = 11111111 11111111 11111111 11110110
       &&&&&&&& &&&&&&&& &&&&&&&& &&&&&&&&
0xff = 00000000 00000000 00000000 11111111
-------------------------------------------
a&0xff = 00000000 00000000 00000000 11110110
```

Bitové operace

Programovací jazyk C umí pracovat s čísly u nichž zadáte jejich bitovou velikost. Struktura s více bitovými položkami může být definována následujcím způsobem:

```
struct pole {
  int nibble:4;
  unsigned int male:2;
  int znam_male:2;
  unsigned int vetsi:7;
  unsigned int jeden_bit:1;
}
```

- položka nibble je číslo se znaménkem a má 4 bity
- položka male je malé číslo bez znaménka o rozsahu 0...3
- položka znam_male je číslo od -2 .. 1
- položka vetsi je číslo od 0 do 127
- položka jeden_bitje buď 0 nebo 1

Bitové operace

Jak bude skutečně definovaná struktura `struct pole` uložena v paměti záleží na překladači.

Struktura bude optimálně uložená ve dvou bajtech, ale může být také uložena ve více bajtech od 5 do 8.

Zarovnávání položek ve strukturách a třídách lze ovlivnit příkazem:
`#pragma pack(N)`,
kdy N definuje na kolik bajtů se jednotlivé položky zarovnávají.

Tento přístup není standardizován, proto je dobré, umět si naprogramovat ukládání do podobné struktury sami.

Bitové operace

```
unsigned short pole=0xffff;

pole &= (~0xf);
pole |= (nibble &0xf);

pole &= (~0x30);
pole |= ((male &0x3)<<4);

pole &= (~0xc0);
pole |= ((znam_male &0x3)<<6);

pole &= (~0x7F00);
pole |= ((vetsi &0x7F)<<8);

pole &= (~0x8000);
pole |= ((jeden_bit &0x1)<<15);
```

```
printf("nibble = %u\n",
        pole&0x0f);

printf("male = %u\n",
        (pole&0x30)>>4);

printf("znam_male = %d\n",
  (((pole&0xc0)>>6)^0x2)-2);

printf("vetsi = %u\n",
        (pole&0x7f00)>>8);

printf("jeden bit = %u\n",
        (pole&0x8000)>>15);
```

Bitové pole

Bitové pole se používá často jako nejefektivnější způsob zjištění, zda jsou některé objekty použité nebo nepoužité.

```
unsigned char array[128];
// 1024 bitů uložených v poli 128 bajtů

void set_bit(int index, unsigned char bit) {
  if (bit==0) {
    array[index>>3] &= (~(1<<(index&7)));
  } else {
    array[index>>3] |= (1<<(index&7));
  }
}

unsigned char get_bit(int index) {
  return ((array[index>>3] & (1<<(index&7)))!=0u)?1u:0u;
}
```

Bitové pole

Použití funkcí je pak již jednoduché:

```
int main() {
  int i;

  for (i=0; i<1024; i++) {
    set_bit(i, (i%5)==0);
  }

  for (i=0; i<1024; i++) {
    printf("bit %i = %u\n", i , get_bit(i));
  }
  return 0;
}
```

Testovací kvíz

Kolik jste toho zatím pochopili?

A Vše bez problému.

B Téměř vše.

C Téměř nic.

D Vůbec nic.