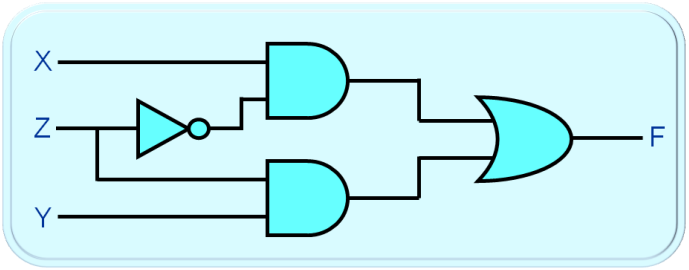


A P O L O S

prerekvizita pro předměty

Architektura počítačů

&

Logické systémy a procesory

Richard Šusta

Katedra řídicí techniky

ČVUT-FEL v Praze

Verze 1.1 ze dne 9. února 2019

Obsah

Seznam obrázků

Seznam tabulek

1 Souhrn prerekvizity

Prerekvizita znamená minimální potřebné znalosti pro absolvování určitého předmětu nebo kurzu, které musí být splněné před umožněním jeho studia. [Slovník cizích slov]

Proč vznikla?

Mnozí se setkali s logickými obvody a binárními čísly už na střední škole nebo si téma sami nastudovali, ale pro ostatní jde o nový pojem. Když jsem přednášky zahájil od úplných základů, znalejší studenti mi psali do ankety hodnocení předmětů, že se v prvních hodinách docela nudili. Sotva jsem v reakci na jejich připomínky zaměřil výklad rychleji na zajímavější otázky, méně obeznámení mi zase naopak vyčítali, že nerozuměli úvodním pasážím.

Tak, abych vyhověl všem, napsal jsem prerekvizitu pro sjednocení vstupních znalostí. Zahrnul jsem do ní pouze lehké pojmy. Věci složitější na pochopení zůstaly v přednáškách.

Jak ji studovat?

Přečtěte si určitě celý text. Budete-li mít dojem, že nějaké pasáži rozumíte, nepřeskakujte ji, ale velmi zběžně si ji prohlédněte, zda přece jenom v ní neobjevíte nějaký nový poznatek. Zpomalte však, pokud narazíte na méně známé pojmy či si někde nebudete stoprocentně jisti, a pečlivě si téma prostudujte včetně postupů v řešených příkladech.

Vzhledem k tomu, že většina odborné literatury bývá převážně v angličtině, budu se snažit u českých technických pojmů uvádět jejich anglické překlady.

Přehled kapitol

Kapitola 2

Kapitola 2 obsahuje minimální znalosti z logických funkcí. Začíná sice hezky vysokoškolsky ☺, a to matematickou definicí nutnou pro další logický popis, ale nezalekněte se vzorců, další text zahrnuje pouze jednoduché pojmy. Předpokládá ovšem, že dovedete převést malé dekadické číslo (stačí v rozsahu 0 až 15) na binární číslo bez znaménka. Neumíte-li, přečtěte si napřed úvod kapitoly 3.

Pokračuje se možnostmi zápisu logické funkce do pravdivostní tabulky. Pouhé otrocké vypisování všech možných kombinací lze mnohdy nahradit stručnějšími metodami. Jednou z nich bude i nakreslení Karnaughovy mapy (KM) logické funkce, což bývá nejčastější způsob, jímž se při ručním zápisu vyjadřují menší logické funkce v technické praxi. Popis KM zůstane na úrovni "kuchařky" a její teoretické pozadí bude vysvětleno na přednáškách.

Nakonec se proberou základní logické funkce NOT, AND, OR a XOR - ty můžete znát i z jazyků C, C# a Java, kde pro ně existují bitové operátory ~, &, | a ^, pro první tři z nich i logické operátory !, && a ||. Na závěr si ukážeme jednoduché převody mezi logickými schématy a logickými výrazy.

Kapitola 3

Obsah kapitoly 3 by všichni mohli "teoreticky" znát z programování. Zopakuje se v ní binární kódování celých čísel bez znaménka a se znaménkem, tedy typy *unsigned integer* a *signed integer*, a jejich přetečení při sčítání nebo odčítání. Dále si připomeneme logické a aritmetické posuny doleva a doprava, které v logice patří mezi velmi důležité operace. V jazycích C, C# a Java je částečně umí bitové operátory << a >>.

Na závěr si zopakujeme hexadecimální notaci, důležitá BCD čísla a ASCII kódování.

2 Logické funkce

Mějme logické proměnné, které nabývají jen hodnot z nějaké konečné množiny B.

Úplně definovanou logickou funkcí n proměnných (*Completely Specified Logic Function*) $y = f(x_1,x_2,x_3,..x_n)$ nazveme zobrazení:

$$B^n \rightarrow B, \text{ kde } (x_1,x_2,x_3,..x_n) \in B^n , x_i \in B, y \in B.$$

- Pokud množina B obsahuje pouze dva prvky, má tedy mohutnost (*cardinality*) |B|=2, pak hovoříme o **dvouhodnotové logice**[1] (*two-valued logic*). Množinu B lze napsat jako B={´0´,´1´}, kde ´0´ a ´1´ značí logickou nulu (*false*) a logickou jedničku (*true*).
- B^n označuje kartézský součin (*Cartesian product*), tj. množinu všech možných n-tic vytvořených z B, a pro |B|=2 je $|B^n|=2^n$.
- Zobrazením (*mapping*) $B^n \rightarrow B$ volíme výstupy, které budou pevně přiřazené jednotlivým prvkům z B^n. Pro n logických proměnných lze definovat 2^{2^n} různých logických funkcí:
  - pro n=1 existují $2^{2^1} = 2^2 = 4$ různé logické funkce,
  - pro n=2 existuje $2^{2^2} = 2^4 = 16$ různých logických funkcí,
  - pro n=3 existuje již $2^{2^3} = 2^8 = 256$ různých logických funkcí.

Příklad: Mějme B={´0´,´1´}. Logickou funkci dvou vstupů zapíšeme jako y=f(x1, x2), zde kartézský součin B^2 má čtyři dvojice, tj. B^2 = { (´0´, ´0´), (´0´, ´1´), (´1´, ´0´), (´1´, ´1´) }. Existuje 16 různých zobrazení. Vybereme jedno z nich. Výstupu přiřadíme logickou ´1´ jen při různých vstupech a tuto logickou funkci nazveme *xor* neboli non-ekvivalence. Naši funkci y=xor(x1, x2) definujeme tedy jako zobrazení:

xor: $B^2 \rightarrow B$ = (´0´, ´0´) → ´0´ zjednodušený zápis 0 0 → 0
(´0´, ´1´) → ´1´ 0 1 → 1
(´1´, ´0´) → ´1´ 1 0 → 1
(´1´, ´1´) → ´0´ 1 1 → 0

2.1 Popis logické funkce pravdivostní tabulkou

Zobrazení přiřazuje právě jednu hodnotu z B ke každé kombinaci hodnot vstupních proměnných. Tabulka je jen jiným zápisem zobrazení. Funkci *xor* z předchozího příkladu zapíšeme třeba jako

x1	x2	xor
0	0	0
0	1	1
1	0	1
1	1	0

nebo i takto:

x1	x2	xor
1	1	0
1	0	1
0	1	1
0	0	0

či ještě jinak:

x1	x2	xor
0	0	0
1	1	0
1	0	1
0	1	1

Všechny tři tabulky popisují totožnou logickou funkci. Na pořadí řádků v tabulce vůbec nezáleží, můžeme je napsat v libovolném sledu za předpokladu, že uvedeme úplně všechny. Definice logické funkce od nás totiž požaduje pouze to, že ke každé možné kombinaci vstupních proměnných je přiřazena právě jedna výstupní hodnota z B.

[1] Při návrhu logických obvodů nevystačíme jen s logickou ´0´ a logickou ´1´. I v tomto textu brzy zavedeme 3hodnotovou logiku přidáním hodnoty X (*don´t care*), protože se bez ní neobejdeme. Pro profesionální práci se pak hodně používá 9hodnotová logika MVL-9, o níž se dozvíte v odborných předmětech.

Vypisování všech možných kombinací je zdlouhavé, a tak se často několik funkcí spojuje do jedné tabulky. Například můžeme spolu s *xor* napsat i další běžné logické funkce:

x1	**x2**	**xor**	**and**	**or**	**nand**	**nor**
0	0	0	0	0	1	1
0	1	1	0	1	1	0
1	0	1	0	1	1	0
1	1	0	1	1	0	0

Někdy se hodí snížení počtu řádků. Například pro přidělení požadavku na přerušení se používají prioritní logické funkce. Jejich výstup p udává číslo nejvyššího vstupu *xi* v logické '1'.

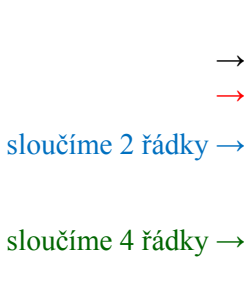
Pro 3 vstupy může prioritní funkce mít například tabulku vpravo.

- Výstup p3 je 00, pokud žádný vstup není v ´1´.
- Výstup p3 přejde do 01, pokud jen vstup x1 je ´1´.
- Pokud bude x3 v ´0´ a x2 v ´1´, pak výstup p3 má hodnotu 10, bez ohledu na vstup x1, protože chceme, aby x2 mělo vyšší prioritu než x1.
- Výstup p3 bude 11 při nejvíce prioritním vstupu x3 v ´1´ bez ohledu na stav ostatních vstupů.

x3	x2	x1	p3	
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

Předchozí tabulku lze zkrátit použitím příznaku, že stejná hodnota výstupu se opakuje pro některý vstupní bit jak v ´1´, tak v ´0´. Ten nahradíme třeba znakem ´-´ (pomlčka) pro reprezentování jakési "*wildcard*" (divoké karty, zástupného znaku).

x3	x2	x1	p	
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1



x3	x2	x1	p3	
0	0	0	0	0
0	0	1	0	1
0	1	-	1	0
1	-	-	1	1

Nová tabulka (vpravo nahoře) má už jen 4 řádky. Aplikací *wildcards* se zkracuje zápis pomocí slučování vstupů (*merged inputs*) a vkládá se jimi jakýsi předpis pro generování řádků tabulek. Snadno teď napíšeme i větší prioritní funkci pro 10 vstupů.

x10	**x9**	**x8**	**x7**	**x6**	**x5**	**x4**	**x3**	**x2**	**x1**	**p10**			
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	0	0	0	1	-	0	0	1	0
0	0	0	0	0	0	0	1	-	-	0	0	1	1
0	0	0	0	0	0	1	-	-	-	0	1	0	0
0	0	0	0	0	1	-	-	-	-	0	1	0	1
0	0	0	0	1	-	-	-	-	-	0	1	1	0
0	0	0	1	-	-	-	-	-	-	0	1	1	1
0	0	1	-	-	-	-	-	-	-	1	0	0	0
0	1	-	-	-	-	-	-	-	-	1	0	0	1
1	-	-	-	-	-	-	-	-	-	1	0	1	0

Místo 2^{10} =1024 řádků, které bychom museli uvést při vypisování celé tabulky, jich stačilo jen 11: Poslední řádek tabulky obsahuje 9 *wildcards*, `1---------`, a ve skutečnosti reprezentuje předpis, který vygeneruje 2^9 =512 řádků, protože každý použitý zástupný *wildcard* nabývá 2 hodnot, jak ´0´, tak ´1´. Všechny vytvořené řádky budou mít stejný výstup p10=1010.

Jiný příklad: Máme-li tabulku vlevo, pak ve skutečnosti jde o zkrácený zápis tabulky vpravo:

c	b	a	y
-	0	-	1
-	1	0	0
0	1	1	1
1	1	1	0

→

c	b	a	y
0	0	0	1
0	0	1	1
1	0	0	1
1	0	1	1
0	1	0	0
1	1	0	0
0	1	1	1
1	1	1	0

U určitých funkcí se bez zástupných *wildcards* neobejdeme, jako například u předchozí prioritní funkce p10 pro deset vstupů. Při ručním zápisu se však jejich nadměrným používáním snižuje názornost, jak je patrné i z levé tabulky nahoře, z níž na první pohled nepoznáme, zda jsme skutečně uvedli všechny možné kombinace vstupů.

Zástupné *wildcards* se však hojně uplatňují v počítačích pro zpracování pravdivostních tabulek, třeba během procesu minimalizace logických funkcí.

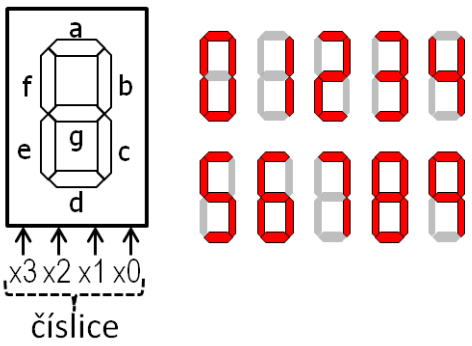
2.2 Hodnota X - don´t care

Pokud bychom třeba měli zapsat pravdivostní tabulku pro dekodér převádějící dekadické číslice na 7segmentový displej, pak ji snadno vytvoříme pro vstupní hodnoty 0 až 9 (binárně *unsigned* jako 0000 až 1001, viz kapitola 3.1 str. 23).

Jaké výstupní hodnoty se mají ale přiřadit vstupům 10 až 15 (binárně *unsigned* 1010 až 1111), které zadání nespecifikuje? Můžeme si pro ně něco vymyslet, ale v době návrhu ještě nevíme, zda námi náhodně vybrané hodnoty neztíží pozdější operace, jako třeba minimalizaci logických funkcí. Moudřejší bude zatím odložit rozhodnutí o jejich hodnotách.

Jako znamení odloženého rozhodnutí použijeme příznak zvaný "***don´t care***", který specifikuje, že nám na výstupní hodnotě nezáleží. Ten se často zapisuje jako X.

Pomocí X a zástupného znaku ´-´ už snadno zapíšeme tabulku dekodéru převádějící dekadickou číslici na její 7segmentový obraz. Předpokládáme, že segmenty svítí při logické ´1´.



Obrázek 1 - 7segmentový displej

Číslice	bity čísla				Segment						
	x3	x2	x1	x0	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
10-11	1	0	1	-	X	X	X	X	X	X	X
12-15	1	1	-	-	X	X	X	X	X	X	X

Tabulka 7segmentového displeje vpravo na Obrázek 1 je téměř profesionální, až na dělení vstupů a výstupů do jednotlivých sloupců. Při stručnějším zápisu se logické hodnoty často spojují do sekvencí, respektive vektorů, což výrazně zmenší tabulku.

Například místo

x3	x2	x1	x0
0	0	0	0

zapíšeme jen 0000 a do záhlaví tabulky uvedeme pořadí logických proměnných v sekvenci. Tabulku, kterou uvádí Obrázek 1, lze zkrátit na stručnější zápis vpravo:

Číslice	bity čísla				Segment						
	x3	x2	x1	x0	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
10-11	1	0	1	-	X	X	X	X	X	X	X
12-15	1	1	-	-	X	X	X	X	X	X	X

Číslice	Binárně x:3210	Segment abcdefg
0	0000	1111110
1	0001	0110000
2	0010	1101101
3	0011	1111001
4	0100	0110011
5	0101	1011011
6	0110	1011111
7	0111	1110000
8	1000	1111111
9	1001	1110011
10-11	101-	XXXXXXX
12-15	11--	XXXXXXX

Sekvence logických ´0´ a ´1´ mají i praktický význam. Dají se snáze využít pro zápisy logické funkce v profesionálních vývojových nástrojích pro návrhy logických obvodů. Logické hodnoty se v nich často zpracovávají ve formě vektorů pro zkrácení kódu. Naproti tomu se tam skoro nepoužívá zdlouhavé definování logické funkce pomocí vyplňování tabulek dělených na jednotlivé sloupce.

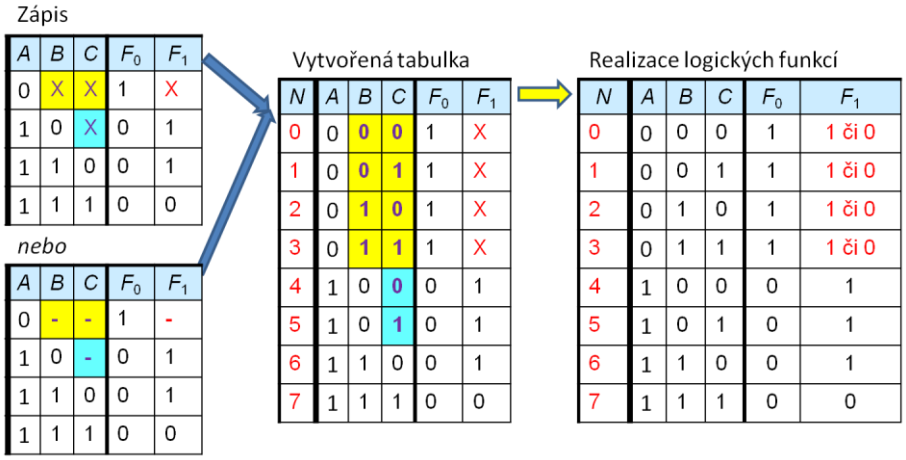
Více o "don't-care"

- "*don't-care*" označuje pouze návrhářovu poznámku, že se o hodnotě výstupu rozhodne během další kroků návrhu, a to podle toho, co se později ukáže výhodnějším. Jedná se tedy o znamení odloženého rozhodnutí (tj. něco jako značka *to-do*).
- "*don't-care*" se dá použít jedině u výstupů. Při návrhu nemůžeme v žádném případě odložit rozhodnutí o hodnotách vstupů, ty musíme dopředu znát.
- "*don't care*" nemá význam neznámého výstupu, ačkoliv se tak někde nesprávně překládá. V češtině odpovídají jeho významu víc pojmy "zatím nezadaný", "dosud nespecifikovaný", "ještě neurčený".
- "*don't-care*" nelze fyzicky realizovat v obvodech, a tak se nakonec musí všechny "*don't-care*" nahradit nějakou konkrétní realizovatelnou logickou hodnota, tedy např. logickou ´0´ či logickou ´1´[2].

Žel v publikacích se často zástupné *wildcards* pro sloučené vstupy a "*don't care*" výstupy označují stejnými symboly, zpravidla znaky X nebo -. Pak musíme rozlišovat jejich konkrétní významy podle jejich umístění v pravdivostní tabulce, zda se nachází v části vstupů nebo výstupů.

[2] Ve snaze o maximální přesnost se zde vyhýbáme tvrzení, že se X (don´t care) se vždy a všude musí dodefinovat buď na logickou ´0´ nebo ´1´. Většinou se tak stane, ale existují i jiné možnosti. Výstup může například přejít i do stavu vysoké impedance, tj. být odpojený, což se hodně používá na počítačových sběrnicích, jak později uvidíte v odborných předmětech.

- Vstup logické funkce: Je-li například napsaný kód "0 - -" nebo "0 X X" (dle autorem použité notace), pak se vygenerují 4 řádky vstupů 000, 001, 010 a 011 s naprosto stejnými výstupními hodnotami, protože znak, ať už ´X´ či ´-´, má zde postavení zástupného *wildcard*, tedy předpisu pro generování hodnot.
- Výstup logické funkce: Například kód "1X" či "1-" bude u výstupu znamenat odložené rozhodnutí o jeho hodnotě. Tady má naopak symbol vždy význam "*don´t care*". U výstupu totiž nemůžeme použít žádné generování zástupnými *wildcard* znaky - každý výstup musí mít v konečné tabulce použité pro realizaci logické funkce vždy jen jednu fixní hodnotu, a lze pouze dočasně odložit rozhodnutí o tom, jaká nakonec bude.



2.3 Zápis pravdivostní tabulky pomocí výčtu hodnot

Tabulka 1 popisuje 8 logických funkcí, jejichž výstupy F0 až F7 nabývají hodnoty ´1´ jen pro jednu logickou kombinaci vstupů, hlásí tak její přítomnost. Společně tvoří *one-hot* dekodér, do češtiny překládaným jako 1 z N, v našem případě 1 z 8. Jde o velmi důležitý logický konstrukční prvek, který tvoří základ mnoha dalších funkcí.

N	C	B	A	F0	F1	F2	F3	F4	F5	F6	F7
0	0	0	0	**1**	0	0	0	0	0	0	0
1	0	0	1	0	**1**	0	0	0	0	0	0
2	0	1	0	0	0	**1**	0	0	0	0	0
3	0	1	1	0	0	0	**1**	0	0	0	0
4	1	0	0	0	0	0	0	**1**	0	0	0
5	1	0	1	0	0	0	0	0	**1**	0	0
6	1	1	0	0	0	0	0	0	0	**1**	0
7	1	1	1	0	0	0	0	0	0	0	**1**

Tabulka 1- Dekodér "One-hot" - 1 z 8

Ani jedna z předchozích metod se nehodí pro elegantní zkrácení popisu podobného dekodéru. Můžeme však výstupy specifikovat množinou vstupních hodnot, v nichž nabývají logické ´1´, protože těch je zde méně než ´0´, což nazveme ***onset***. Kombinace vstupních bitů zakódujeme jako binární číslo *unsigned* (popis kapitola 3.1 str. 23).

Tabulka 1 se redukuje na jeden řádek, na seznam onset-ů.

$F0^{on} = \{ 0 \}$, $F1^{on} = \{ 1 \}$, $F2^{on} = \{ 2 \}$, $F3^{on} = \{ 3 \}$, $F4^{on} = \{ 4 \}$, $F5^{on} = \{ 5 \}$, $F6^{on} = \{ 6 \}$, $F7^{on} = \{ 7 \}$

v ostatních stavech výstupy F0 až F7 nabývají ´0´.

Tabulka 2 popisuje jiný analogický dekodér, který se nazývá *one-cold*, protože výstupy F0 až F7 budou právě pro jednu vstupní kombinaci v ´0´. Česká terminologie žel nerozlišuje mezi dekodéry *one-hot* a *one cold* a oba překládá jako 1 z N, zde tedy 1 z 8.

N	C	B	A	F0	F1	F2	F3	F4	F5	F6	F7
0	0	0	0	**0**	1	1	1	1	1	1	1
1	0	0	1	1	**0**	1	1	1	1	1	1
2	0	1	0	1	1	**0**	1	1	1	1	1
3	0	1	1	1	1	1	**0**	1	1	1	1
4	1	0	0	1	1	1	1	**0**	1	1	1
5	1	0	1	1	1	1	1	1	**0**	1	1
6	1	1	0	1	1	1	1	1	1	**0**	1
7	1	1	1	1	1	1	1	1	1	1	**0**

Tabulka 2- Dekodér "One-cold" - 1 z 8

Zde popis pomocí *onset*-ů není výhodný, lepší je popis pomocí *offset*-ů[3], které znamenají množinu vstupů, pro které je výstup v ´0´. Funkce dekodéru *one-cold* zapíšeme opět snadno:

$F0^{off} = \{ 0 \}$, $F1^{off} = \{ 1 \}$, $F2^{off} = \{ 2 \}$, $F3^{off} = \{ 3 \}$, $F4^{off} = \{ 4 \}$, $F5^{off} = \{ 5 \}$, $F6^{off} = \{ 6 \}$, $F7^{off} = \{ 7 \}$

Opět zde platí, že neuvedené hodnoty jsou v druhé množině, tedy v logické ´1´.

Popisy pomocí *onset*-u a *offset*-u lze použít i u logických funkcí, které používají *don´t care* stavy, akorát zde musíme přidat ještě *don´t care* set.

N	C	B	A	X	Y
0	0	0	0	**0**	0
1	0	0	1	**0**	0
2	0	1	0	**X**	0
3	0	1	1	**X**	0
4	1	0	0	1	0
5	1	0	1	1	**X**
6	1	1	0	1	**1**
7	1	1	1	1	**1**

Logické funkce X(C,B,A) a Y(C,B,A), definované tabulkou nahoře, můžeme zapsat takto:

$$X: X^{off} = \{ 0,1 \}, X^{dc} = \{ 2, 3 \} ; Y: Y^{on} = \{ 6,7 \}, \; Y^{dc} = \{ 5 \} ,$$

Pro každou funkci jsme zvolili metodu, která nám dala nejméně práce. Výstup X jsme popsali pomocí *offsetu* a *don´t care* setu, protože výstupních ´0´ bylo méně než ´1´, zatímco výstup Y jsme vytvořili pomocí *onsetu* a *don´t care* setu.

Poznámka: Matematické označení pro zápisy *onset, offset* a *don´t case set* se hodně liší podle autora. Zde uvedené popisy F^{on} , F^{off} a F^{dc} nejsou ustálené. Uplatňuje se i jiné značení. Například *onset* se často zapíše jako malé m (od *minterm)* a *offset* jako velké M (od *Maxterm).* Výše uvedené zápisy pro X a Y by v jiné literatuře mohly ve stručném zápisu vypadat i takto

$$X: M(0,1), dc(2, 3); \; Y: m(6,7), dc(5)$$

Použité názvy minterm a maxterm vyplývá z procesu minimalizace logických funkcí, jehož vysvětlení přesahuje rámec této publikace. Seznámíte se s ním v odborných předmětech.

[3] Název *offset* je poměrně zavádějící, protože se tak v technice a matematice obvykle označuje odchylka nebo posun, ale v literatuře o logických obvodech se opravdu používá. Česká terminologie pro *onset* a *offset* v logických obvodech je tak různorodá, že ji radši ani neuvádíme.

2.4 Karnaughova mapa

V technické praxi se logické funkce s menším počtem vstupů zapisují Karnaughovou mapou. Její detailní odvození si ukážeme na pravdivostní tabulce funkce Y =f(d,c,b,a), se čtyřmi vstupy (d,c,b,a) a výstupem Y. Hodnoty Y označíme jen logickými konstantami y00 až y15, aby jejich indexy naznačily řazení výstupů. Konstanty mají hodnoty logická ´0´, ´1´ či X. .

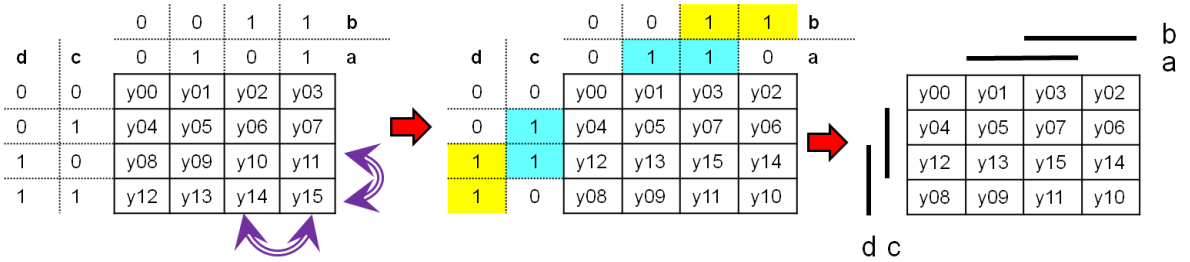
d	c	b	a	Y
0	0	0	0	y00
0	0	0	1	y01
0	0	1	0	y02
0	0	1	1	y03
0	1	0	0	y04
0	1	0	1	y05
0	1	1	0	y06
0	1	1	1	y07
1	0	0	0	y08
1	0	0	1	y09
1	0	1	0	y10
1	0	1	1	y11
1	1	0	0	y12
1	1	0	1	y13
1	1	1	0	y14
1	1	1	1	y15

		0	0	1	1	b
d	c	0	1	0	1	a
0	0	y00	y01	y02	y03	
0	1	y04	y05	y06	y07	
1	0	y08	y09	y10	y11	
1	1	y12	y13	y14	y15	

Obrázek 2 - Pravdivostní tabulka nakreslená v maticovém tvaru

Pravdivostní tabulka vlevo má 16 řádků, ale vstupy d a c mají totožné hodnoty vždy pro čtveřici řádků. Takovou tabulku můžeme s výhodou zkráceně zapsat v maticovém tvaru 4x4, viz vpravo, kde pro každý výstup jsou jeho hodnoty vstupů specifikované sloupcem a řádkou.

Tabulku vpravo na Obrázek 2 upravíme. Prohodíme její dva poslední sloupce a poté i dvě poslední řádky, čímž dostaneme prostřední tabulku na Obrázek 3 — ta je již Karnaughovou mapou logické funkce. Logické ´1´ vstupů jsou v ní vedle sebe, a tak místo vypisování 0 a 1 se často kreslí jen čára symbolizující, kde je hodnota ´1´, viz tabulka vpravo.



Obrázek 3 - Geneze Karnaughovy mapy 4x4

Nejdůležitější vlastností Karnaughovy mapy, nutnou podmínkou k tomu, aby se vůbec jednalo o Karnaughovu mapu, je skutečnost, že při jakémkoliv pohybu v ní o jedno políčko svisle nebo vodorovně se změní jen jedna vstupní proměnná.

Například výstup y00 má vstupy dcba=0000 a sousední výstup y04, o řádek níže, má vstupy dcba = 0100. Při přechodu od y00 k y04 se tedy změnil jen vstup c z ´0´ na ´1´.

Platí to i přes okraj mapy. Například při posunu z prvního řádku na čtvrtý ve stejném sloupci. Zde vezmeme-li na ukázku třeba poslední sloupec - výstup y02 má vstupy dcba=0010 a výstup y10 má vstupy dcba = 1010. Změnilo se jen d z ´0´ na ´1´.

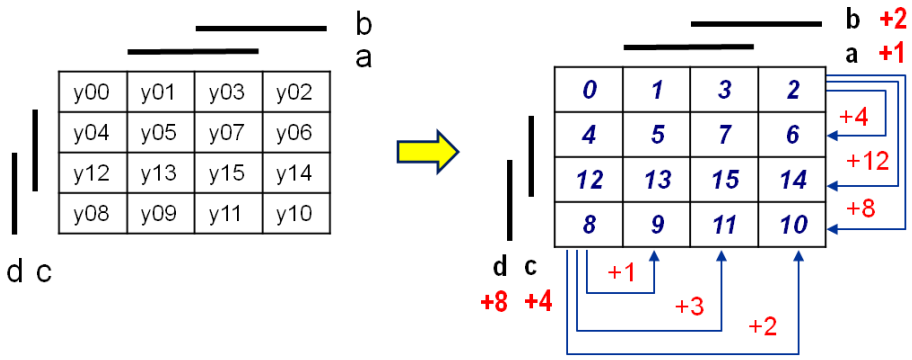
Stejně tak například výstup y14 na konci třetího řádku má hodnoty vstupů dcba=1110 a výstup y12 na začátku stejného řádku má hodnoty dcba = 1100. Změnilo se jen b z ´1´ na ´0´.

Řazení hodnot tak, aby se vždy měnila jen jedna vstupní proměnná, se nazývá Grayovým kódem či Grayovým uspořádáním. Jde o velmi významný pojem, využívaný nejen v minimalizaci logických funkcí, ale ve snímačích pozice a přenosech informace, například pro korekci chyb v digitální televizi.

Indexy i výstupů yi v Karnaughově mapě nejdou za sebou. Porovnáme-li ale jejich čísla v jednom řádku, pak vůči prvku v prvním sloupci řádky je index ve druhém sloupci stejného řádku vždy větší o +1 , ve třetím sloupci o +3 a ve čtvrtém sloupci o +2 .

Vlastnost vyplývá ze vstupních proměnných. Každý bit má váhu danou mocninou řadou 2^n. Uspořádáme-li je jako dcba, pak vstup a má váhu $1=2^0$, vstup b má váhu $2=2^1$, vstup c má váhu $4=2^2$ a vstup d má váhu $8=2^3$. Jejich součtem (řádek+sloupec) je dána hodnota indexu odpovídajícímu příslušnému poli Karnaughovy mapy.

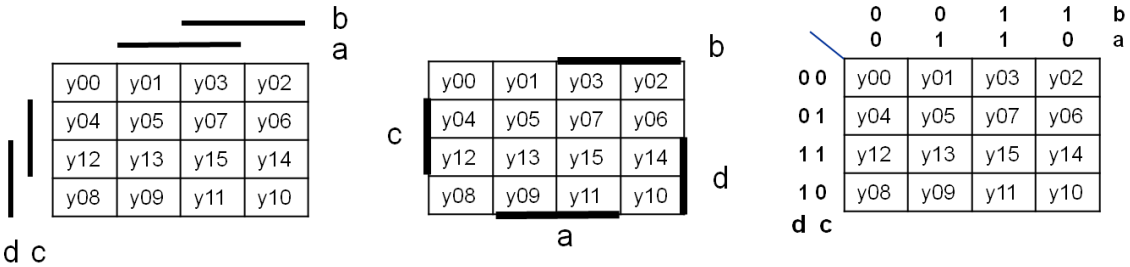
Druhý sloupec má indexy o +1 vyšší než první sloupec, protože se v něm uplatní proměnná a s vahou +1. Třetí sloupec bude mít navíc oproti prvnímu už dvě proměnné a+b, tedy bude mít indexy o +3 vyšší než první sloupec. Analogicky poslední sloupec má navíc oproti prvnímu sloupci jen proměnnou b, a ta má váhu +2.



Obrázek 4 - Závislosti v Karnaughově mapě 4x4

Z uvedené vlastnosti odvodíme i rozdíly mezi indexy v jednom sloupci. Druhý řádek má hodnoty indexů vždy o +4 větší než odpovídající prvek stejného sloupce prvního řádku (+c=4). Třetí řádek má indexy větší o +12 (+d+c) oproti prvnímu řádku a čtvrtý řádek o +8 (+d). Stačí nám tak správně přidělit indexy jen prvnímu řádku a zbylé si už mechanicky odvodíme.

Karnaughova mapa, zkráceně KM, kterou uvádí Obrázek 3, není samozřejmě jedinou možností, jak ji nakreslit či jak uspořádat vstupní proměnné. Používá se několik různých stylů, dle zvyku autora, viz Obrázek 5.



Obrázek 5 - Některé možné způsoby nakreslení Karnaughovy mapy 4x4

Rovněž se případně mohou i jinak seřadit proměnné, takže budou mít jiné váhy, čímž dostaneme i odlišné řazení indexů. Musí se však vždy splnit dříve zmíněná nutná vlastnost KM, a to vstupy měnící se podle Grayova kódu, tedy pro jakýkoliv posun o jeden prvek, ať ve sloupci či řádku, včetně přechodů přes hrany mapy, se nám změní jen jedna vstupní proměnná.

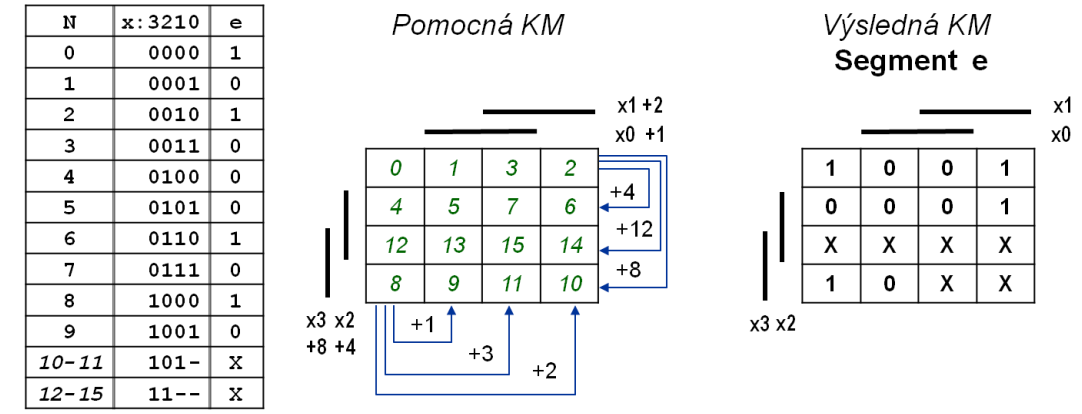
Příklad: Nakreslete Karnaughovu mapu (KM) pro segment e 7segmentového displeje.

Řešení: Obrázek 1 na straně 7 popisuje pravdivostní tabulku 7segmentového displeje. Vezmeme z ní hodnoty pro segment e.

Vzhledem k tomu, že nemáme zatím velké zkušenosti s kreslením KM, tak raději postupujeme přes mezikrok, abychom neudělali zbytečnou chybu ☺.

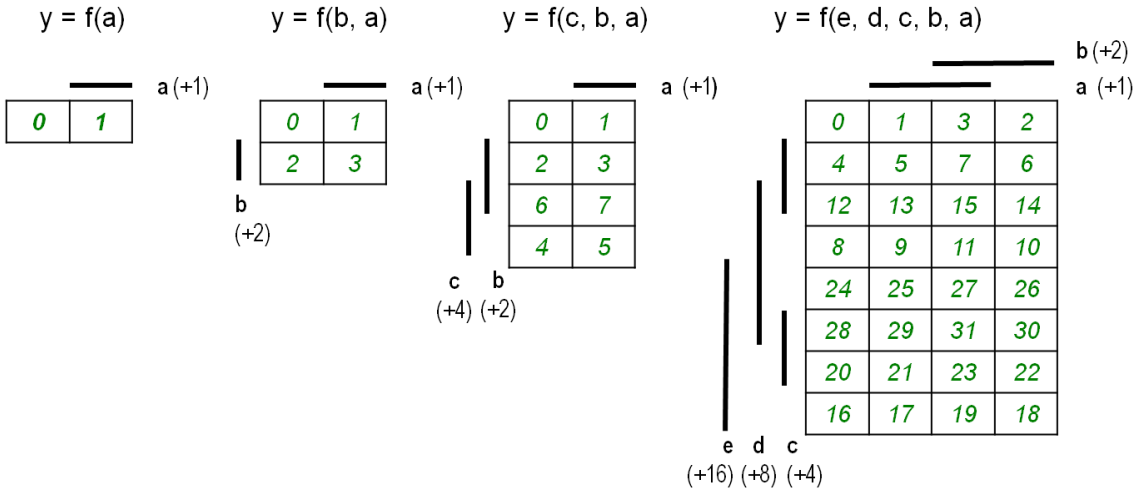
Napřed si nakreslíme pomocnou KM, kde vyplníme čísla indexů jednotlivých polí dle našeho řazení vstupních proměnných. Podle něho pak zapíšeme hodnoty podle pravdivostní tabulky segmentu e.

N	x:3210	e
0	0000	1
1	0001	0
2	0010	1
3	0011	0
4	0100	0
5	0101	0
6	0110	1
7	0111	0
8	1000	1
9	1001	0
10-11	101-	X
12-15	11--	X



2.4.1 Karnaughovy mapy pro jiné velikosti

Karnaughovy mapy se nehodí pro zpracování v počítači a používají výhradně pro ruční minimalizaci či pro zápis logických funkcí s malým počtem vstupů. I když je lze teoreticky sestavit pro libovolně velkou logickou funkci, tak jejich přehlednost klesá exponenciálně s nárůstem počtu proměnných. Demonstruje to i Obrázek 6, kde se prezentují některé vybrané možnosti jak sestavit Karnaughovu mapu pro jiné počty vstupů než 4x4, a to včetně výsledného číslování polí.

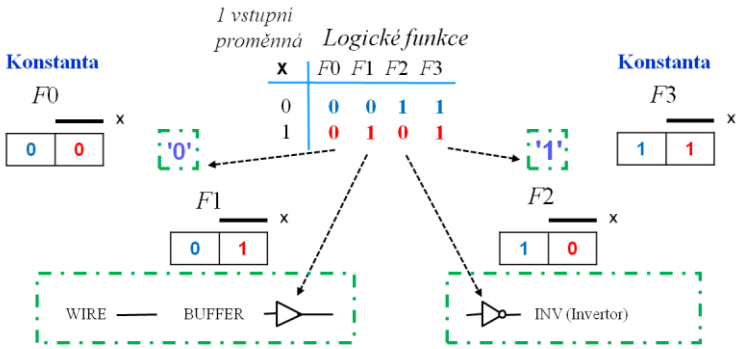


Obrázek 6 - Karnaughovy mapy pro jiné velikosti než 4x4

Naštěstí v logické funkci lze snížit počet proměnných různými dekompozicemi, které budou tématem odborných předmětů, takže pro ruční návrhy můžete vždy vystačit s mapami do velikosti 4x4 ☺.

2.5 Přehled hlavních logických funkcí

Obrázek 7 ukazuje všechny logické funkce **jedné vstupní proměnné**, včetně používaných schematických značek. Dvě z nich jsou konstanty, protože jejich výstup zůstává neměnný bez ohledu na vstupní hodnotu - F0 má na výstupu vždy logickou ´0´ a F3 trvalou logickou ´1´.



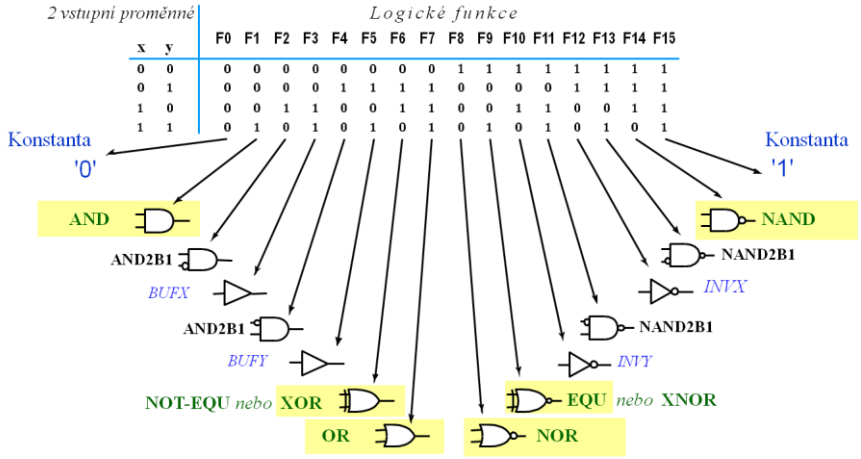
Obrázek 7 - Logické funkce jedné vstupní proměnné

Další F1 má svou výstupní hodnotu rovnou vstupu. Jde-li o pouhé spojení, pak zastupuje vodič či drát, kvůli tomu se nazývá WIRE. Použije-li se pro ni elektronický prvek, buď pro elektrické oddělení nebo získání vyššího výstupního proudu či změny úrovně napětí, tak pro zdůraznění této skutečnosti se v tomto případě nazývá BUFFER (oddělovač, budič).

Logická funkce F2 INV (NOT) mění vstupní logickou ´0´ na ´1´, a ´1´ na ´0´. Nazývá se kvůli tomu invertor nebo negace, anglicky pak *invertor* nebo *negation* či *complement*,

Všechny **logické funkce dvou vstupních proměnných** ukazuje Obrázek 8, opět včetně jejich schematických značek. Když si ho prohlédnete, zjistíte, že je jich sice 16, ale 6 z nich vyznačených modrým písmem, lze nahradit logickými funkcemi jedné proměnné.

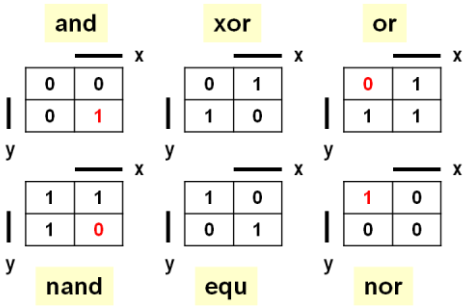
První z nich jsou funkce F0 a F15 nezávisící na vstupech. Jde o konstanty známé z Obrázek 7. Další funkce BUFX, BUFY, INVX a INVY mající výstupní hodnotu závislou jenom na jednom vstupu, a takže je lze nahradit prvky BUFFER nebo invertor (INV) připojený k tomu vstupu, který u příslušné logické funkce ovlivňuje výstup.



Obrázek 8 - Logické funkce dvou vstupních proměnných

Ze zbývajících 10 logických funkcí se prakticky používá jenom 6 z nich - ty jsou v obrázku vyznačené žlutým zvýrazněním, a to AND, XOR, OR, NOR, EQU, NAND. Jejich Karnaughovy mapy ukazuje Obrázek 9. Jak můžeme z něho vidět, vybrané logické funkce dvou proměnných jsou snadno zapamatovatelné pro svou symetrii - jejich výstup nezávisí na pořadí

vstupů, tj. nezmění se, prohodíme-li x a y mezi sebou. Navíc tři logické funkce v dolní řádě na Obrázek 9, *nand*, *equ* a *nor* jsou negované logické funkce první řady, takže ve skutečnosti nám stačí dobře znát jen tři logické funkce dvou proměnných, a to *and*, *xor* a *or*.



Obrázek 9 - Karnaughovy mapy hlavních logických funkcí 2 proměnných

Pro další text zavedeme uspořádání logických hodnot předpisem ´0´ < ´1´, slovy logická ´0´ je menší než logická ´1´.

- **Logická funkce *AND*** dává na výstupu logickou ´**1**´ pouze v případě, že oba její vstupy jsou v logické ´**1**´. Lze ji tedy považovat za výběr minimální hodnoty vstupů, tj. bude-li jakýkoliv vstup v logické ´**0**´, pak minimální hodnota bude ´**0**´.
- **Logická funkce *OR*** je svým způsobem zrcadlově obrácená k *and*. Logická funkce *or* dává na výstupu logickou ´**0**´ pouze v případě, že má oba vstupy v logické ´**0**´. Lze ji tedy považovat za výběr maximální hodnoty vstupů, tj. bude-li jakýkoliv vstup v logické ´**1**´, pak maximální hodnota bude ´**1**´.

Použité analogie výběru minima a maxima nám dovolují snadno rozšířit logické funkce *and* a *or* na funkce s libovolným počtem vstupů.

- **Logická funkce *AND* s n vstupy**, $Z=and(x_{n-1},...,x_1, x_0)$, vybírá minimum z hodnot všech svých n vstupů - Z bude v logické ´**1**´ jen tehdy, budou-li všechny vstupy x_i = ´**1**´. Bude-li na jednom vstupu nebo na více vstupech logická ´**0**´, pak minimum Z= ´**0**´.
- **Logická funkce *OR* s n vstupy**, $Z=and(x_{n-1},...,x_1, x_0)$, vybírá maximum z hodnot všech svých n vstupů - Z bude v logické ´**0**´ jen tehdy, budou-li všechny vstupy x_i = ´**0**´. Bude-li na jednom vstupu nebo na více vstupech logická ´**1**´, pak maximum Z=´**1**´.

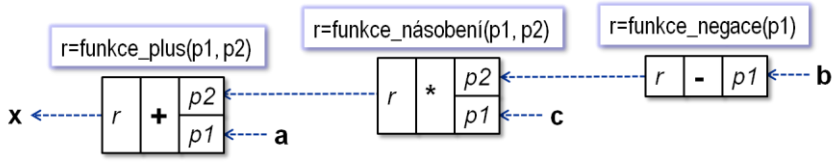
Logické funkce XOR a EQU provádí porovnání vstupů:

- Logická funkce XOR, *eXclusive OR* (vylučující ´nebo´), dává na výstupu ´1´, pokud je pouze jediný její vstup v logické ´1´. Častěji se ale popisuje tak, že *xor* má svůj výstup v logické ´1´, pokud jsou hodnoty vstupů různé. Kvůli této vlastnosti se někdy nazývá NOT-EQU nebo NON-EQU, od *not/non equivalence*, ale název *xor* bývá mnohem běžnější jak v programech, tak i v logických systémech.
- Logická funkce EQU, *EQUivalence*, dává na výstupu logickou ´1´, pokud mají oba vstupy stejnou hodnotu. Občas se používá i její jiný název naznačující, že jde o negované xor, a to XNOR, *eXclusive NOT OR*, ale EQU bývá mnohem častější.

I logickou funkci *xor* lze definovat pro více vstupů, ale podobné rozšíření nemá větší praktický význam. Vícevstupové *xor* má mizivé uplatnění, na rozdíl od opravdu hojně používaných vícevstupových *and* a *or*. Skoro všude se vystačí s dvouvstupovým *xor* — to na druhou stranu tvoří ale mimořádně významný prvek řady obvodů.

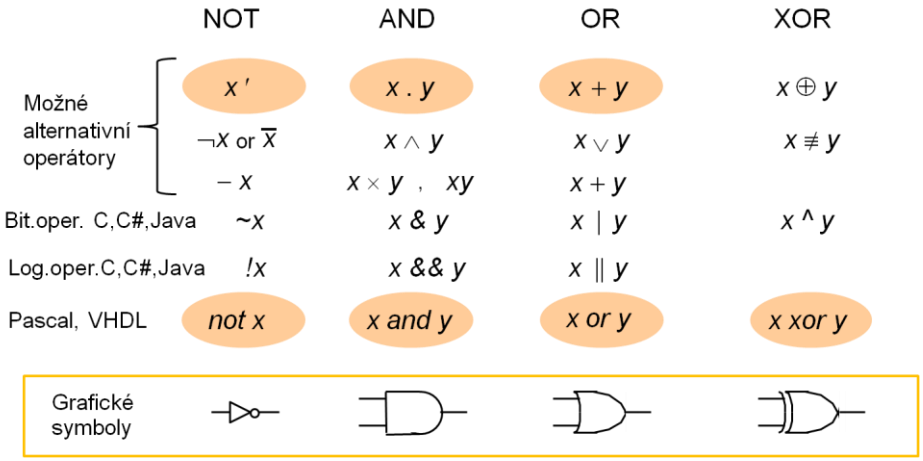
2.6 Operátory a logické funkce

Vezme-li běžný matematický výraz, například x=a+(-b)*c, pak skutečný postup, jak se bude výraz počítat, tvoří zřetězené volání funkcí, které lze vyjádřit výrazovým stromem:



nebo matematicky zapsat: x = funkce_plus(a, funkce_násobeni(c, funkce_negace(b))). Unární funkce funkce_negace má jeden vstupní parametr p1 a vrací výsledek r (*result*), zatímco zbylé funkce jsou binární, tj. mají dva vstupní parametry p1 a p2.

I logické funkce se mnohem častěji zapisují pomocí operátorů než ve funkčním tvaru. Přehled používaných operátorů ukazuje Obrázek 10, v němž zvýraznění zdůrazňuje nejčastější používané symboly, jejichž volba vyplývá především z klávesnic počítačů.



Obrázek 10 - Symboly pro logické operátory

Pro operace AND a OR by byly asi nejvhodnější symboly ∧ a ∨, avšak ty se píší složitěji na běžném počítači, takže se pro ně více používají symboly ´+´ a ´.´ běžně vyhrazené pro sčítání a násobení. Ty mají ale úplně jiné vlastnosti jako logické operátory!

Sčítání není distributivní vůči násobení, ale logické OR je distributivní vůči AND, viz žlutě zvýrazněné políčko v tabulce dole. Navíc sčítání a násobení nejsou idempotentní operace, ale AND a OR ano, viz zeleně zdůrazněná políčka.

	komutativní	***asociativní***	***distributivní***	***idempotentní***
AND ´.´	a.b = b.a	a.(b.c) = (a.b).c	a.(b+c) = (a.b)+(a.c)	a.a = a
OR ´+´	a+b = b+a	a+(b+c) = (a+b)+c	a+(b.c) = (a+b).(a+c)	a+a = a

Násobení má vyšší prioritu (precedenci) před sčítáním, což pak nesprávně indukuje i prioritu AND před OR, pro kterou není žádný skutečný důvod, protože obě logické operace mají rovnocenné postavení. Většinou se však z důvodu snížení počtu nutných závorek zavádí vyšší priorita operátorů AND před OR, ale neplatí vždy. Není definovaná například v jazyce VHDL, který se používá pro návrh obvodů.

Cvičení: Zkuste si vztahy uvedené nahoře dokázat na základě skutečnosti, že AND lze chápat jako výběr minimální hodnoty a OR jako výběr maximální hodnoty, viz kapitola 2.5.

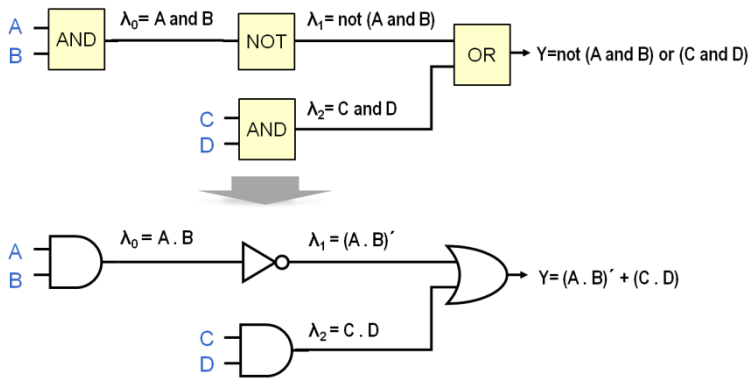
2.7 Logická schémata

Logické schéma jednoduché logické funkce představuje ve skutečnosti postup vyhodnocení výrazu, tedy jakýsi výpočtový strom, který vytvoří syntaktický analyzátor (slangově *parser*).

Vezmeme-li například funkci Y= (not (A and B)) or (C and D). Jelikož unární operace mají obecně vyšší prioritu než binární operace, můžeme vynechat červeně vyznačené závorky a napsat funkci zkráceně jako Y= not (A and B) or (C and D), respektive i pomocí operátorů "+" , "." a "´" jako Y=(A . B)´ + (C . D) .

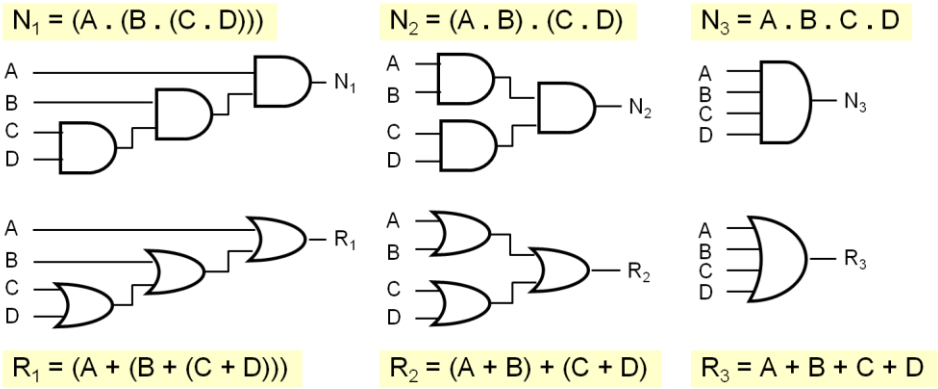
Její vyhodnocení může začít třeba levou operaci AND: λ_0=A.B [λ_0=A and B], kde λ_0 označuje její mezivýsledek. Poté se provede jeho negace λ_1=(A.B)´ [λ_1=not (A and B)]. Dále se spočte další operace AND: λ_2=C.D [λ_2=C and D]. Nakonec se oba mezivýsledky λ_1 a λ_2 spojí pomocí operace OR na Y=λ_1+λ_2 = (A.B)´ + (C.D) [Y= not (A and B) or (C and D)].

Postup vyhodnocení ukazuje Obrázek 11. Nahoře jsou jednotlivé operace zapsané jmény logických funkcí, avšak dole je stejné schéma nakresleno mnohem častějším způsobem pomocí schematických značek pro jednotlivé logické operátory. Grafická značka pro logickou operaci se z historických důvodů často nazývá logické hradlo (ang. *logic gate*).



Obrázek 11 - Logické schéma a jeho logický výraz

Logické schéma, podobně jako strom výrazu, popisuje přesný postup vyhodnocení. Budeme-li mít třeba funkce N = A . B . C . D a R = A + B + C + D, tedy logické funkce s více operacemi AND či OR za sebou, pak je lze počítat několika různými způsoby. Některé z nich ukazuje Obrázek 12.



Obrázek 12 - Některé možnosti vyhodnocení více logických operací AND a OR

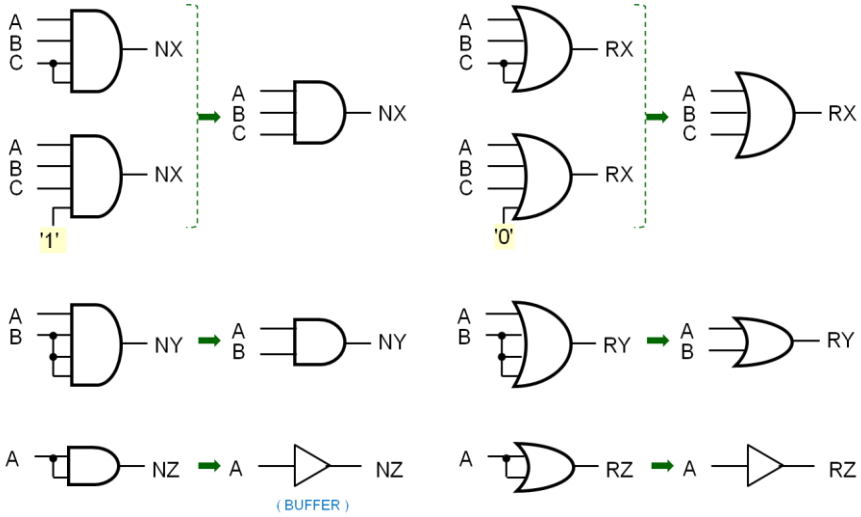
Výsledky N_1 a R_1 se vyhodnocují řetězením binárních operací, což nám dovoluje asociativita AND a OR. Další N_2 a R_2 se počítají jiným zřetězením operací, a to do stromu. Poslední N_3 a

R_3 využívají vícevstupové operace AND a OR zavedené v kapitole 2.5, které spočtou násobné logické operace v jednom kroku.

Z matematického hlediska platí $N=N_1=N_2=N_3$ a $R=R_1=R_2=R_3$, protože logický výsledek zde nezávisí na tom, v jakém pořadí vyhodnotíme logickou funkci. Při její realizaci nám na tom většinou také nezáleží, protože v řadě případů žádáme pouze správný logický výsledek,[4] a tak volíme způsob, který se nám zrovna zamlouvá.

Někdy bývají ve schématech použité značky (hradla) pro logické funkce AND a OR s více vstupy a přitom jenom část jejich vstupů se využije, a to z nějakých z konstrukčních důvodů, nebo třeba i kvůli tomu, že grafický editor zrovna nenabízel vhodnou značku ☺.

U vícevstupových elementů lze snížit počet vstupů několika způsoby. U operací AND a OR můžeme připojit více vstupů logické funkce na jeden signál, jelikož pro obě funkce platí idempotence, viz Kapitola 2.6 na str. 16. Redukujeme tak počet vstupů funkce, jak ukazuje Obrázek 13. Propojíme-li všechny vstupy logického hradla, můžeme dostat až obyčejný budič (*buffer*), který byl dříve popsaný, viz Obrázek 7 na str. 14.



Obrázek 13 - Snížení počtu vstupů u AND a OR

Vstupy, které u hradla AND a OR nepotřebujeme, lze také připojit na konstanty.

- U AND, které představuje výběr minima ze vstupů, můžeme nepoužité vstupy dát na maximum, neboť to neovlivní výsledek, tedy na logickou ´1´, viz funkce NX.
- Zato u OR, jenž představuje výběr maxima, můžeme nepoužité vstupy připojit naopak na minimum, tady na logickou ´0´, viz funkce RX.

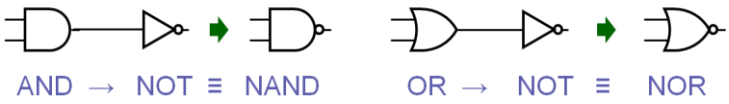
Důležitá poznámka: Necháte-li jakýkoliv nepoužitý vstup nezapojený, pak se pokaždé jedná o hrubou chybu v návrhu[5]. Vývojové nástroje pro obvody se snaží podobná opomenutí korigovat, vypíší přitom záplavu varování a nepoužité vstupy automaticky připojí na '1' nebo '0'. Nemusí se však pokaždé trefit do vhodné veličiny a hledání takové nezapojené chyby bývá pak nenasytně "časožravé" ☺

[4] Výpočty jednotlivých logických funkcí N_x nebo R_x mají odlišné délky maximální cesty mezi vstupy a výstupem, takže jejich výsledky dostáváme s různým časovým zpožděním. Většinou nám to nevadí. Výjimečné situace, kdy se musí brát v úvahu doba vyhodnocení logické funkce, přesahují rámec této publikace a budou tématem pokročilejších přednášek odborných předmětů.

[5] I kdyby nezapojený vstup neměl vliv na činnost obvodu, zvyšuje se tím elektrické rušení v obvodu. Jev bude vysvětlen ve velmi pokročilých přednáškách odborných předmětů.

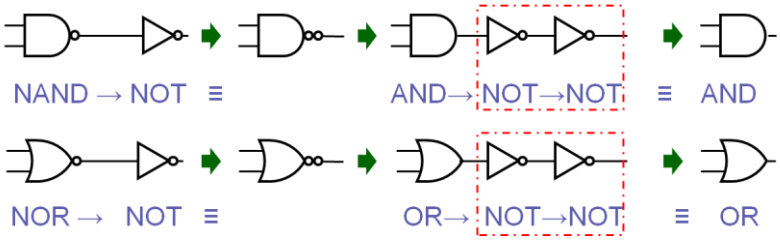
2.7.1 Bublinky negací

Při kreslení schémat se často invertory nepíší celou značkou, ale redukují se na pouhé kolečko či bublinku. Značky pro logické funkce NAND a NOR, které definoval Obrázek 9 na str. 15, se skládají z funkce AND a NOT, respektive OR a NOT, avšak kreslí se zkráceně. Místo nakreslení celého invertoru (NOT) se za výstup značky pro AND, respektive OR, jen doplní bublinka na znamení negace, jak ukazuje Obrázek 14.



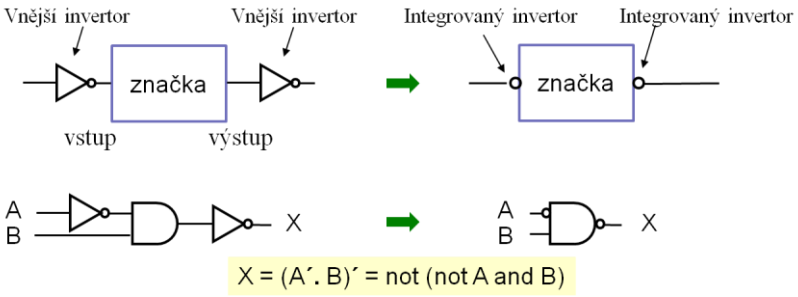
Obrázek 14 - Značky NAND a NOR

Jelikož platí not (not a) = a, slovy dvě negace se navzájem vyruší, pak dvě bublinky negací se navzájem také vymažou, jak ukazuje Obrázek 15.



Obrázek 15 - Dvojitá negace

Bublinky negací se také používají jak na výstupech, tak na vstupech, viz Obrázek 16, kde je nakreslená jedna značka popisující rovnici X = (A´.B)´



Obrázek 16 - Bublinky negace vstupů a výstupů

2.7.2 Realizace logických schémat

V dřívějších dobách logická schémata sloužila i k přímé realizaci logických funkcí. Jednotlivé značky operací se zapojily obvody zvanými logická hradla, které dovedly přímo realizovat operace OR, AND, NOT, NAND, NOR, XOR a další. Schéma tak představovalo jakýsi konstrukční plán celého obvodu.

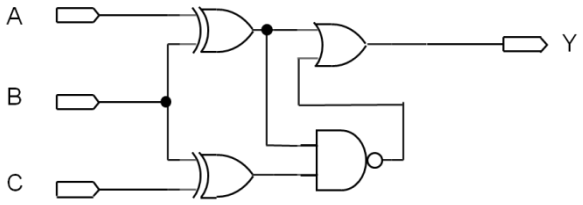
Vývoj moderních prostředků pro logiky ale posouvá tento způsob spíš do raritní oblasti, takže se pomalu stává skoro stejně vzácným jako například obvody s elektronkami. Dnes se logika v drtivé většině realizuje programovatelnými obvody a název hradlo zůstává už jen jako synonymum pro značku logické operace.

Logická schémata se však dále používají hlavně pro jejich názornost. U menších logických funkcí je v nich lépe vidět závislosti výstupů na vstupech a případná spolupráce logických funkcí s pokročilejšími logickými obvody.

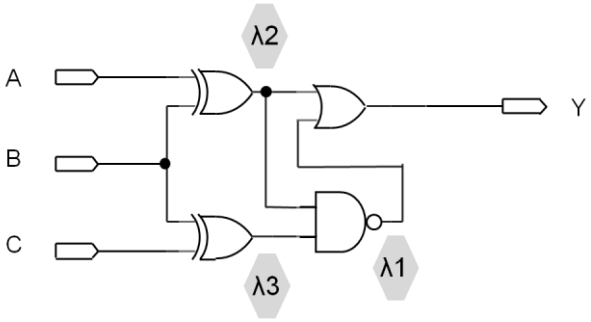
2.7.3 Převod logického schéma na výraz

Na závěr této části si ukážeme převod logického schéma na logický výraz. Jelikož schéma představuje postup vyhodnocení logické funkce, stačí tedy jen jít podle něho a prováděné operace psát jako logický výraz, co demonstrujeme na příkladu.

Příklad - napište logický výraz odpovídající logické funkci.



Řešení: Logický výraz můžeme sestavit buď odleva, tedy ve směru výpočtu, nebo naopak od konce. Ukážeme si druhý způsob, který bývá mnohem univerzálnější, protože umí i hodně složité obvody s vnitřními smyčkami. Označme si napřed výstupy jednotlivých bloků.



Obvod obsahuje xor elementy, a tak zvolíme zápis operátorů slovy. Výstup Y je OR funkcí:

$$Y = \lambda 2 \text{ or } \lambda 1 \qquad \text{(eq1)}$$

λ1 je dáno funkcí AND s bublinkou negace, tedy λ1 = not (λ2 and λ3). Substituujeme vztah pro λ1 do (eq1), čím dostaneme:

$$Y = \lambda 2 \text{ or not } (\lambda 2 \text{ and } \lambda 3) \qquad \text{(eq2)}$$

Dále vypočteme λ2 = A xor B a dosadíme za dva jeho výskyty v (eq2)

$$Y=(A \text{ xor } B) \text{ or not } ((A \text{ xor } B) \text{ and } \lambda 3) \qquad \text{(eq3)}$$

Zůstane nám jen stanovit λ3 = B xor C a to dosadit do rovnice (eq3), čím obdržíme výsledek:

$$Y=(A \text{ xor } B) \text{ or not } ((A \text{ xor } B) \text{ and } (B \text{ xor } C)) \qquad \text{(eq4)}$$

Rovnici (eq4) lze zapsat i pomocí jiných symbolů pro operátory, xor necháme názvem

$$Y=(A \text{ xor } B)+((A \text{ xor } B).(B \text{ xor } C))'$$

~o~

Zde lze uzavřít kapitolu o základech logických funkcí. Pro otestování znalostí si můžete zkusit test v následující části.

Pro porozumění složitějším operacím, jako například sčítačkám nebo čítačům, je nutné znát i základy vnitřního kódování celých čísel, tedy typů *signed* a *unsigned integer*, a dále hexadecimální notaci, BCD čísla a ASCII kódování znaků. Většina studentů se s těmito pojmy už jistě setkala v programovacích předmětech, nicméně raději pojmy zopakujeme v kapitole 0.

2.8 Test ze znalostí Kapitoly 2

Zkuste odpovědět na 4 otázky z hlavy, tj. bez pomůcek. Správné řešení najdete v příloze.

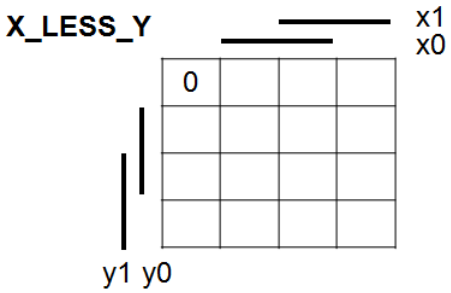
Otázka 1: Doplňte nedokončené pravdivostní tabulky logických funkcí:

x3	x2	x1	AND(x1,x2,x3)	OR(x1,x2,x3)	NAND(x1,x2,x3)	NOR(x1,x2,x3)
0	0	0				
0	0	1				
0	1	0	**0**			
0	1	1		1		
1	0	0			1	
1	0	1				0
1	1	0				
1	1	1				

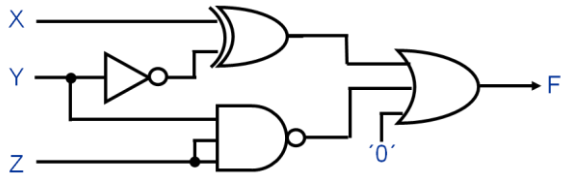
x2	x1	XOR(x1,x2)	EQU(x1,x2)
1	0		
1	1		
0	0	**0**	
0	1		0

Otázka 2: Přepište tabulku vlevo do Karnaughovy mapy.

x1	x0	y1	y0	X_LESS_Y
0	0	0	0	0
0	0	0	1	1
0	0	1	-	1
0	1	0	-	0
0	1	1	-	1
1	0	0	-	0
1	0	1	0	0
1	0	1	1	1
1	1	-	-	0



Otázka 3: Napište logický výraz, který je realizovaný logickým schématem:



F(X,Y,Z)=.....

Otázka 4: Nakreslete k logické funkci její logické schéma.

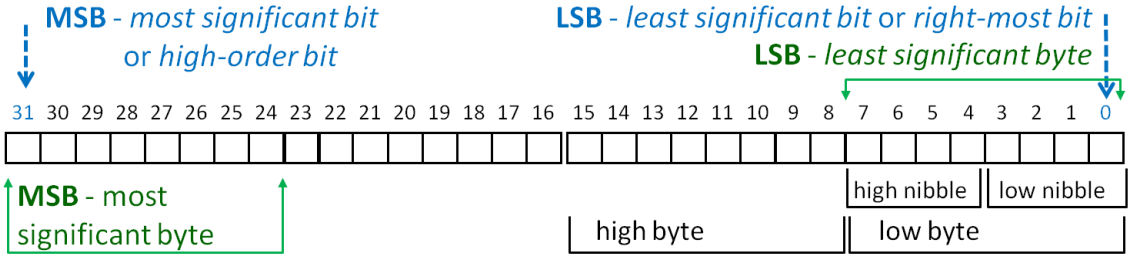
G(X,Y,Z) = not((not X xor Y) and not (not Y or Z))

3 Kódování čísel

Možná jste někde už slyšeli následující vtip, či nějakou jeho modifikaci:

Po autohavárii mi programátor podepsal náhradu škody 1000 Kč.
Zaplatil deset korun s tím, že mi dává dvě koruny navíc.

Pointa se opírá o pojem, že 1000 znamená vždy 8 dekadicky ve dvojkové (binární) soustavě. Může, ale nemusí. Hodnota závisí na způsobu kódování čísel a bitové délce čísla. Programovací jazyky používají různé délky binárních čísel, ale zpravidla jen násobky délky bytu, tedy 8 bitů. Nejnižší bit se v nich nachází vpravo, zatímco nejvyšší vlevo.



Obrázek 17 - Byte, bit, MSB, LSB

Samotné slovo "*bit*" pochází angličtiny a znamená trošku, špetku. Někde se používá i pojem "*nibble*" pro čtveřici bitů, což v angličtině znamená ždibec, kousíček. Samotný byte vznikl pak jazykovou alternací od anglického "*bite*", tedy sousto. V literatuře se někdy pro byte používá výjimečně i jiný pojem, a to "*octet*", neboli osmice.

Binární číslo může mít v logických obvodech libovolnou kladnou délku, tj. délku větší než nula. Není zde omezení na celý počet bytů. Nejnižší bit binárního čísla může ležet jak vpravo (jako v obrázku nahoře), tak vlevo. I když se i v obvodech se upřednostňuje klasické počítačové uspořádání s nejnižším bitem vpravo. Pro označení pořadí se používají zkratky:

MSB ve významu "*most significant bit*" nebo "*high-order bit*", tedy nevýznamnější bit. *MSB* se používá i pro uspořádání bytů jako "*most significant byte*".

LSB má opačný význam "least significant bit" nebo "right-most bit", nejméně významný bit. *LSB* se opět používá i pro uspořádání bytů jako "*least significant byte*".

Rozlišení, zda se *MSB* či *LSB* vztahuje k bitu či bytu musí vyplynout z kontextu popisu.

Pojem *word* - délka "*word*" (slovo) označuje nativní délku daného počítače. Na dnešních 32bitých procesorech je "*word*" 32bitů, na 64bitovém procesoru je 64 bitů. Rozhodně *word* není vždy a všude 16bitové binární číslo, jak se někde mylně uvádí[6]. Například při letu na Měsíc měly "*Apollo Guidance Computers*" 15bitové *word*.

Binární číslo je pouhá posloupnost 1 a 0 a o jeho dekadické hodnotě lze rozhodnout až na základě specifikace způsobu použitého pro zakódování dekadických čísel. Používá se několik různých způsobů. Ty nejčastější rozebereme v následujících kapitolách.

[6] Jistou výjimkou velikosti *word* jsou některé průmyslové programovací jazyky pro PLC (programovatelné automaty), pro něž je šířka *word* definovaná normou norma IEC 1131-3. V té je pojem WORD zavedený jako 16bitový typ a z toho odvozené DWORD (double word) pro 32bitový typ a LWORD (long word) pro 64bitový typ. Norma se ovšem vztahuje jen k PLC, jinde neplatí.

3.1 Binární číslo ´*unsigned*´ - bez znaménka

Binární čísla *unsigned*, celým anglickým názvem: *binary encoded unsigned integers*, představují základní binární čísla. Jejich princip se opírá o matematický fakt, že v řadě 2^N mocnin dvou je součet všech předchozích členů řady vždy o jedničku nižší než následující člen řady. Například součet prvních 4 členů řady je $2^0+2^1+2^2+2^3 = 1+2+4+8 = 15 = 2^4-1$. Obecně platí:

$$2^n - 1 = \sum_{k=0}^{n-1} 2^k \qquad (1)$$

Každé celé nezáporné číslo můžeme vyjádřit jako součet vybraných členů řady 2^N. Bude-li nějaký člen příslušné mocniny použitý, zapíšeme bit 1, jinak 0. Řetězec m bitů $x \approx b_{m-1}\, b_{m-2} \ldots b_1\, b_0$ nazveme binárním číslem bez znaménka, dále jen binární číslo *unsigned*, a přiřadíme mu hodnotu:

$$x = \sum_{k=0}^{m-1} b_k 2^k \qquad (2)$$

Například řetězec 1100100 se jako binární číslo *unsigned* převede na dekadickou hodnotu 100:

1*2^6	+ **1***2^5	+ **0***2^4	+ **0***2^3	+ **1***2^2	+ **0***2^1	+ **0***2^0 =
64	**+ 32**			**+ 4**		**=100**

Vztah (1) nám zaručuje, že ke každému nezápornému celému číslu existuje právě jedna kombinace členů řady, která dává dané číslo jako svůj součet, přičemž každý člen řady se v součtu vyskytne nejvýše jen jednou. Jinými slovy, kódování je zcela jednoznačné.

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
13	8192
14	16384
15	32768
16	65536
17	131072
18	262144
19	524288
20	1048576

3.1.1 Změna bitové délky čísla

Hodnotu binárního čísla *unsigned*, bez znaménka, určují pouze jedničkové bity. Před číslo lze vložit libovolný počet nul, aniž se změní jeho hodnota. Například binární čísla *unsigned*: 1100100, 01100100, 001100100, 0001100100, atd. mají stejnou dekadickou hodnotu 100. Zde předpokládáme neomezenou bitovou délku. V praxi bývá počet bitů binárního čísla omezený, a tak můžeme samozřejmě přidávat jen tolik nul, kolik se nám vejde.

3.1.2 Logické posuny

Operace posunu doleva, (angl. *logical left shift*), je přidání bitu 0 za binární číslo *unsigned*, tj. na jeho pravou stranu. Například pro u binárního čísla 101, odpovídajícího dekadicky číslu 5 (2^2+2^0), po posunu doprava dostaneme 1010, které má dvojnásobnou dekadickou hodnotu, tj. 10. Každý jedničkový bit se totiž posunul pod následující vyšší člen mocninné řady s dvojnásobnou hodnotou. Podobně binární číslo bez znaménka 10100 s přidanou dvojicí bitů má čtyřnásobnou dekadickou hodnotu, tj. 20, a 101000. má osminásobnou hodnotu 40.

		2^5=32	2^4=16	2^3=8	2^2=4	2^1=2	2^0=1
8*5=**40**	=32+8	1	0	1	0	0	0
4*5=**20**	=16+4		1	0	1	0	0
2*5 = **10**	=8+2			1	0	1	0
5	=4+1				1	0	1

Je-li bitová délka čísla omezená, pak hodnota bude dvojnásobkem jen tak dlouho, dokud se nám levý 1 bit nedostane na konec délky pro uložení čísla.

Máme-li například binární číslo 101 uloženo na 8 bitů, tj. jako 00000101, pak jeho hodnota se při prvních 5 posunech vlevo vždy zdvojnásobí, tj. až k hodnotě 10100000, která odpovídá dekadickému číslu 5 * 2^5 = 5*32=160. Další posun vlevo by v 8bitové délce vedl na 01000000, což jako binární číslo *unsigned* je dekadicky 64. Nejvyšší bit 1 se ztratil díky omezené délce čísla. Došlo zde k aritmetickému přetečení, blíže viz kapitola 3.1.5.

Programovací jazyky na bázi jazyka C definují operátor bitového posunu vlevo << [7], za nímž následuje délka posunu v bitech. Máme-li proměnnou byte x = 5; (v jazyce C unsigned *char x=5;*), pak platí: 2 * x == (x << 1) a 4*x == (x << 2) až 32*x == (x << 5).

Opačná operace logického posunu doprava, (angl. *logical right shift)* pro binární čísla *unsigned* odpovídá operaci celočíselného dělení 2, neboť bity 1 se dostávají k předchozím členům mocninné řady s poloviční hodnotou. Při celočíselném dělení dostaneme výsledek a zbytek po dělení. Posuneme-li binární číslo *unsigned* 101 o jeden bit doprava, dostaneme 10, nejnižší bit 1 nám vypadl, je zbytkem po dělení.

		2^5=32	2^4=16	2^3=8	2^2=4	2^1=2	2^0=1	
5	=4+1				**1**	**0**	**1**	
5/2 = **2**	=2					**1**	**0**	-> 1
a zbytek po dělení 1								

V jazyce C existuje částečná implementace posunu operátorem >> , který nedává zbytek po dělení, jen podíl. Pro proměnnou byte x = 5; platí: (x % 2) == 1 a x/2 == (x>>1) a x/2==2.

Překladač programovacích jazyků často využívají posuny k překladu celočíselného násobení a dělení konstantami rovnými mocninám 2. Posuny jsou totiž velmi rychlé operace.

3.1.3 Převod binárního čísla bez znaménka na dekadické číslo

Postup 1: Dekadická hodnota binárního řetězce, při braná jako binární číslo *unsigned*, je určena přímo součtem odpovídajících členů řady 2^N, kde N je číslo bitu.

Máme-li tedy binární řetězec X=10011, který má jedničky na pozicích 4., 1. a 0. bitu, pak můžeme stanovit jeho hodnotu pouhým součtem odpovídajících členů:

$$X=10011 \rightarrow 2^4 + 2^1 + 2^0 = 16 + 2 + 1 = 19$$

Pokud je binární řetězec delší a s více jedničkami, jako například Q=11111110110, pak jeho převod na číslo bez znaménka pomocí sčítání by byl trošku náročnější, protože Q má jedenáct bitů a většina z nich jsou jedničky:

bit	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
Q	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**0**

Výpočet si můžeme zkrátit uplatněním zmíněné vlastností řady mocnin dvou, a to, že hodnota následujícího členu řady 2^N je vždy o jedničku vyšší než součet všech předešlých členů. Víme tedy, že

$$2^{11}-1 = 2048-1 = 2047 = 2^{10}+2^9+2^8+2^7+2^6+2^5+2^4+2^3+2^2+2^1+2^0$$

Číslo Q vede na posloupnost sčítanců z řady 2^N, ve které nejsou jen dva členy 2^3 a 2^0, neboť binární řetězec Q má nulové bity na 3. a 0. pozici. Z toho plyne

$$Q = 11111110110 \rightarrow 2047 - 2^3 - 2^0 = 2047-8-1 = 2038$$

[7] V jazyce C++ bývají operátory << a >> přetížené, např. vložením iostream.h, a používají se pro čtení a zápis z/do datových proudů (angl. streams). Pro číselné operátory se i tak pořád chovají jako posuny.

Postup 2: Pro převod můžeme využít i operaci logického posunu doleva, jímž provádíme výpočet polynomu podle Hornerova schématu (pod tímto názvem k nalezení na Wikipedii).

Postupujeme od nejvyššího bitu. Vezmeme ho a zapíšeme jako výsledek, tedy 1 nebo 0. Pokud za ním existuje další bit, vynásobíme výsledek dvěma a přičteme k němu hodnotu dalšího bitu, tedy 1 nebo 0. Postup opakujeme, dokud se nedostaneme k nejnižšímu bitu.

10011

1 → 1*2+**0**=2 → 2*2+**0**=4 → 4*2+**1**=9 → 9*2+**1**=19

Jiný příklad již ve zkráceném zápisu

1111110110

1 → 2+**1**=3 → 6+**1**=7 → 14+**1**=15 → 30+**1**=31 → 62+**1**=63 → 126+**1**=127 ...

→ 254+**0**=254 → 508+**1** → 1018+**1** = 1019 → 2038+**0**=2038

Postup 2 nedoporučujeme však pro ruční výpočet, a to na základě zkušeností. Střídají se při něm operace násobení a sčítání, takže není zcela mechanický a je snadné udělat početní chybu. Hodí se však pro algoritmizaci převodu, více dále u BCD čísel, kapitola 3.5.2, str. 41.

3.1.4 Převod dekadického čísla na binární číslo bez znaménka

Pro jednoduché převody lze použít buď postupné odčítání, nebo dělení 2.

3.1.4.1 Postupné odčítání

Při postupném odčítání najdeme v řadě mocnin 2 největší člen řady, který je stále menší než převáděné číslo. Například při převodu čísla 35, vybereme 2^5 , tedy 32. Od něho začneme postupné odčítání, dokud nedostaneme nulu.

číslo	odečítaný člen řady	výsledek	bit	
35	-32	3	**1**	MSB
3	-16	*nelze*	**0**	
3	-8	*nelze*	**0**	
3	-4	*nelze*	**0**	
3	-2	1	**1**	
1	-1	0	**1**	LSB

Pokud je člen řady menší, tak zapíšeme bit 1, hodnotu členu odečteme, v opačném případě píšeme 0 a zkusíme další člen. Výsledkem převodu dekadického čísla 35 na binární číslo bez znaménka bude binární číslo 100011.

Postupné odčítání vyžaduje znalost členů řady 2^N. Několik prvních členů řady není těžké si zapamatovat, ale větší dekadická čísla se pohodlněji převedou postupným dělením.

3.1.4.2 Převod dělením 2

Princip převodu vychází z posunu doprava, viz Kapitola 3.1.3 na str. 24. Číslo postupně celočíselně dělíme 2, dokud nedostaneme 0, a zapisujeme zbytky po dělení jako binární číslo. Ty píšeme od nejnižšího bitu k vyššímu, protože při posunu doprava nám vypadávají nejnižší bity. Při převodu čísla většího než 0, dostaneme nakonec vždy 1 celočíselně děleno 2, a to s výsledkem 0, a zbytkem po celočíselném dělení 1.

35 / 2 = 17 *zbytek po celočíselném dělení* **1** - nejnižší bit
17 / 2 = 8 *zbytek po celočíselném dělení* **1**
8 / 2 = 4 *zbytek po celočíselném dělení* **0**
4 / 2 = 2 *zbytek po celočíselném dělení* **0**
2 / 2 = 1 *zbytek po celočíselném dělení* **0**
1 / 2 = 0 *zbytek po celočíselném dělení* **1** - nejvyšší bit

Dělení není nutné tak podrobně rozepisovat, stačí nám pořád celočíselně dělit 2 a zbytky po dělení zpětně stanovit podle toho, které dílčí podíly byly liché - ty měly zbytek po dělení 1.

Převedeme například dekadické číslo 1000 na binární číslo *unsigned*. V následující řádce → značí, že další číslo bylo odvozeno celočíselným dělením 2:

1000 → **500** → **250** → **125** → **62** → **31** → **15** → 7 → 3 → 1 → 0

Zapíšeme-li liché podíly jako bity 1 a ostatní jako 0, pak dostaneme binární číslo 1111101000 . Bity jsme řadili od nejnižšího, čili v opačném pořadí, než je řada podílů dělení. Všimněte si, že by vůbec nevadilo, kdybychom zahrnuly i poslední → 0, protože bychom dostali binární číslo 01111101000, které má stejnou hodnotu, viz odstavec 3.1.1.

3.1.5 Aritmetické přetečení při sčítání a odčítání

V počítačích i logických obvodech se vždy ukládá jen konečný počet bitů. Pokud k číslu přičítáme 1, pak časem dosáhneme maximální hodnoty, při níž číslo bez znaménka je reprezentováno samými bity 1, v počtu zvolené délky pro uložení čísla.

	Carry	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
254		1	1	1	1	1	1	1	0
+1									1
255		1	1	1	1	1	1	1	1
+1									1
0	1	0	0	0	0	0	0	0	0
+1									1
1		0	0	0	0	0	0	0	1

Tabulka 3 - Přičítání 1 k 8bitovému číslu bez znaménka

Maximální 8bitové binární číslo *unsigned* je 11111111, což je dekadicky 255. Když k němu přičteme 1, dostaneme binární číslo *unsigned* =100000000, které správně odpovídá 2^8 tedy 256, ale má devět bitů. Do 8bitového čísla se nám uloží jedině jeho spodních osm nulových bitů. Nejvyšší 9. bit se musí zahodit, takže náš výsledek bude ve skutečnosti 0.

Na 8bitové aritmetice binárních čísel *unsigned* bude tedy 255+1 = 0. Samozřejmě, kdybychom použili delší binární číslo, pak by správně platilo 255+1=256.

Ztracený nejvyšší bit se nazývá Carry, neboli přenos do vyššího řádu. Pro aritmetiku binárních čísel *unsigned* ohlašuje chybu aritmetického přetečení, tedy překročení maximální zobrazitelné hodnoty pro zvolenou bitovou délku.

K přetečení dojde i při odečítání 1, což si můžeme představit jako postup odspodu nahoru v Tabulka 3. Pak operace odečtení 1 od 0 zde dává výsledek 255. V logických obvodech se někdy tento opačný směr, přetečení z 0 na maximum, nazývá *Borrow*, protože si při něm půjčujeme bit z vyššího řádu.[8] V mnohé literatuře se však oba směry nazývají *Carry*.

[8] Většina procesorů nerozlišuje směr přetečení a jejich ALU v obou případech generuje Carry. Zda šlo o Carry či Borrow se stanoví jen dle právě provádění instrukce asembleru. Sčítání - Carry, odčítání - Borrow.

Odečteme-li od 0 číslo 1, pak vždy dojde k přetečení a celé číslo se vyplní bity 1, což je maximální hodnota. Bitová délka výsledku určí jedině to, jakou dekadickou hodnotu bude chybný výsledek operace 0-1 představovat, například:

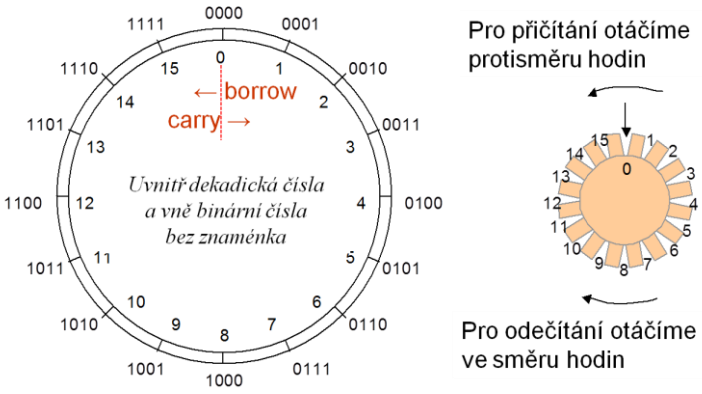
pro 8bitové číslo bez znaménka bude (0-1) = 2^8-1 = 255,
pro 9bitové číslo bez znaménka bude (0-1) = 2^9-1 = 511
pro 16bitové číslo bez znaménka bude (0-1) = 2^{16}-1=65535, atd.

Výsledek odčítání 0-1 je při běžném počítání -1. Hodnota výsledku operace s binárními čísly *unsigned* omezené délky se stanovila pomocí korekce 2^m , kde m je počet bitů zvolený pro uložení čísla. Při výsledku menším než nula připočítáváme 2^m, dokud nedostaneme kladné číslo. Pokud je výsledek naopak větší než 2^m, pak odečítáme 2^m.

Otázka 1: V 4bitové aritmetice čísel *unsigned*, jaká bude dekadická hodnota výsledku operace dvou dekadických čísel při jejich sečtení 14+4 a při odečtení 4-14 ?

Výpočet: Vypočteme **14+4** = 18. Výsledek je větší než 2^4=**16**, a tak korekce je 18-**16**=**2**.
Vypočteme **4-14** = -10. Výsledek je menší než 0, a tak korekce je -10+**16** = **6**.

Předchozí výpočet si můžeme názorně zobrazit, pokud si binární čísla *unsigned* představíme jako kolo opatřené číslicemi, viz Obrázek 18. Operace sčítání odpovídá otáčení kola proti směru hodinových ručiček a odčítání zase otáčení kola ve směru hodinových ručiček. Počet zubů, o něž kolo otočíme, je dán hodnotou čísla, které přičítáme, či odčítáme. Na obrázku vidíme, že číslo 14 leží o 4 pozice proti směru hodinových ručiček od čísla 2 (14+4=2) a číslo 4 leží o 14 pozic ve směru hodinových ručiček od čísla 6 (4-14=6).



Obrázek 18 - Přičítání a odčítání 4bitových čísel bez znaménka

Otázka 2: V 8bitové aritmetice binárních čísel *unsigned*, jaká bude dekadická hodnota výsledku sčítání dvou dekadických čísel 200 a 100?

Výpočet: Sečtu obě dekadická čísla, tedy 200+100 = 300. Výsledek je větší než 2^8 = **256**. Odečtu korekci 2^8: 300-**256**=44. Výsledek operace bude tedy 44.

Otázka 3: V 10bitové aritmetice binárních čísel *unsigned*, jaká bude dekadická hodnota výsledku operace odčítání dvou dekadických čísel 1000-1500?

Výpočet: Vypočtu rozdíl dekadických čísel 1000-1500 = -500. Výsledek je menší než 0, a tak k němu přičtu 2^{10} = **1024.** Tedy -500+**1024**=524. Výsledek bude tedy 524.

Otázka 4: V 5bitové aritmetice binárních čísel *unsigned*, jaká bude dekadická hodnota výsledku sčítání dvou dekadických čísel 10 a 20?

Výpočet: Vypočtu 10+20=30. Výsledek je kladný a menší než 2^5=32. Korekce není třeba.

3.2 Binární číslo ´*signed*´ - se znaménkem ve dvojkovém doplňku

Pro celá čísla se znaménkem existuje několik různých kódování, z nichž se nejvíce používá dvojkový doplněk, založený na aritmetickém přetečení, viz kapitola 3.1.5.

Pokud k binárnímu číslu x uloženému na m bitů vytvoříme **jedničkový doplněk** χ jako negaci všech jeho bitů, tak součet x+ χ bude binární číslo se všemi bity v hodnotě 1, neboť χ má bity 1 jedině tam, kde x má bity 0.

Součet x+ χ = 2^m -1 je maximální binární číslo *unsigned*. Přičteme-li 1 k χ, pak dostaneme x+(χ+1)= 2^m. Výsledek 2^m je binární číslo *unsigned* o délce m+1 bitů. Do m bitů se uloží jen jeho nulové spodní bity. Pro m-bitová čísla *unsigned* tedy platí, že x+(χ +1) = 0.

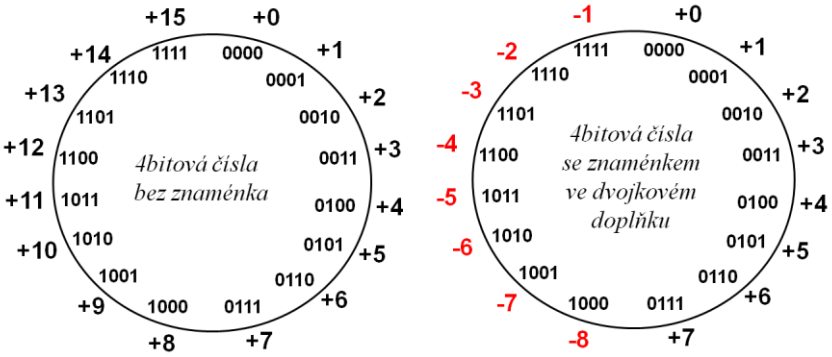
Pro tuto vlastnost se hodnota (χ+1) nazývá **dvojkový doplněk k číslu x *(two's complement of x).***

Například, u binárních čísel délky 4 bity je dekadické číslo 4 uloženo jako binární číslo *unsigned* 0100. Jeho jedničkový doplněk (bitová negace) je 1011 (χ). Přičteme 1 (binárně 0001), dostaneme 1100 (χ+1), což je dvojkový doplněk 4bitového binárního čísla 0100. Součet 0100+1100 =10000. Výsledek 10000 vzatý jako binární číslo *unsigned* se dekadicky rovná 16, avšak 10000 má 5bitvou délku. Do 4bitového binárního čísla se nám uloží jeho spodní 4 bity, takže dostaneme výsledek 0000. Došlo zde k aritmetickému přetečení.

Čísla se znaménkem v dvojkovém doplňku, dále jako typ *signed*, zavedeme takto:

- dvojkový doplněk binárního čísla prohlásíme za negaci původního čísla,
- dále stanovíme, že binární číslo délky m bitů, které má bit 1 ve svém nejvyšším bitu (tj. v bitu s váhou 2^{m-1}), bude obrazem záporného dekadického čísla.

Pro 4bitovou aritmetiku jsou čísla *signed* uvedena na Obrázek 19 vpravo. Binární číslo 1000, mající dekadickou hodnotu -8, má zde výlučné postavení. I k němu existuje dvojkový doplněk, ale jím se binární číslo 1000 samo. Jelikož to má 1 v nejvyšším bitu, reprezentuje dekadické číslo -8, ale už zde nemá kladný dekadický protějšek. Tato nesymetrie je jediným nedostatkem zakódování binárních čísel *signed* (se znaménkem ve dvojkovém doplňku).



Obrázek 19 - 4bitová čísla *unsigned* a *signed*

Zobrazení čísel *signed* jinak nabízí samé výhody. Operace sčítání a odčítání se počítají jako v případě čísel *unsigned* (binárních čísel bez znaménka). Můžeme tedy používat stejnou výpočetní jednotku pro obě reprezentace čísel. Závisí jen na nás, zda interpretujeme výsledek operace jako binární číslo *unsigned* nebo *signed*. Navíc kladná čísla se zobrazí stejně jako čísla bez znaménka, pouze pro záporná čísla se používá dvojkový doplněk.

Pro výše uvedené výhody jsou čísla typu *signed* (se znaménkem ve dvojkovém doplňku) hlavním způsobem pro uložení celých čísla se znaménkem (typ *signed integer*).

3.2.1 Významné vlastnosti

Binární čísla typu *signed* mají významné vlastnosti, které stojí za zapamatování.

- Číslo 0 se zobrazí jako samé bity 0.
- Označme bitovou délku m, pak dekadická čísla v rozsahu -2^{m-1} do 2^{m-1} -1 lze převést, ostatní čísla leží mimo rozsah. Například pro m=4 je to -2^3 až 2^3-1, tedy -8 až 7.
- Binární číslo *signed* mající 1 a za ní m-1 bitů 0 je vždy nejmenším číslem a jeho hodnota se rovná -2^{m-1}. Toto číslo je současně jedinou anomálií — neexistuje k němu kladné číslo. Například, máme-li tedy 8bitová čísla, pak nejmenší binární číslo je 10000000 a jeho dekadická hodnota je $-2^{8-1} = -2^7 = -128$.
- Binární číslo *signed* mající 0 následovanou m-1 jedničkami je vždy největším číslem a jeho hodnota se rovná 2^{m-1} -1. Například, máme-li tedy 8bitová čísla, pak největší binární číslo *signed* je 01111111 a jeho dekadická hodnota je 2^{8-1} -1 = 2^7 -1 = 127. *Poznámka: Dekadické číslo -127 bude převedeno na 10000001 - je o 1 vyšší než -128.*
- Binární číslo *signed* mající samé bity 1, tedy m bitů 1, se vždy rovná dekadickému číslu -1. Například pro 8bitová čísla je to 11111111. *Poznámka: Dekadické číslo -2 bude převedeno na 11111110, je o 1 menší než -1.*

3.2.2 Aritmetická negace pomocí dvojkového doplňku

Mějme binární číslo *signed* (se znaménkem ve dvojkovém doplňku) o známé bitové délce m. K němu najdeme záporné číslo (aritmetická negace) **algoritmem dvojkového doplňku**:

a) logicky negujeme všechny bity binárního čísla, tzv. jedničkový doplněk.
b) k jedničkovému doplňku přičteme binárně 1, čímž obdržíme dvojkový doplněk původního čísla.

Příklad 1: Pro 8bitové binární číslo *signed* 01100100, mající dekadickou hodnotu 100, vypočtěte jeho aritmetickou negaci.
Výpočet: Spočteme jedničkový doplněk negací bitů 01100100→10011011. Přičteme k němu 1, tedy 10011011+00000001 = **10011100**.

Příklad 2: Proveďte aritmetickou negaci pro 10bitové číslo *signed* 1000000000, (dekadicky -512).
Výpočet: Zadání je chyták. Správná odpověď je: **nelze**, viz významné vlastnosti výše.

3.2.3 Převod dekadických čísel na binární

Pro převod dekadických čísel musíme vždy znát bitovou délku požadovaného binárního čísla *signed*. Tu opět označme jako m.

- Kladná čísla vyhovující rozsahu čísel, viz odstavec 3.2.1, převádíme stejně jako čísla bez znaménka.
- Záporná dekadická čísla konvertujeme jedním z následujících postupů, v nichž označíme vstupní záporné dekadické číslo jako -x
    a) převedeme absolutní hodnotu -x, tedy |-x|, jako číslo bez znaménka a k němu vypočteme dvojkový doplněk. Nevýhodou tohoto způsobu je nutnost binárně přičíst 1 při výpočtu dvojkového doplňku.

b) binárnímu sčítání se vyhneme, pokud převedeme absolutní hodnotu dekadického čísla zmenšenou o 1, tedy převedeme |-x|-1 na číslo bez znaménka a pro něj uděláme jen jedničkový doplněk, logickou negaci bitů. Výsledek bude rovný binárnímu číslu -x.

c) alternativně můžeme převést (2^m - x) na číslo bez znaménka. Výsledek bude rovný číslu -x. Tento způsob využívá přímo způsobu, jakým byla čísla se znaménkem ve dvojkovém doplňku zavedena.

Jak si ověříme, že si postup pamatujeme? Stačí znát obraz aspoň jednoho čísla, třeba dekadické -1 převáděné na samé bity 1. Poté si napřed zkuste převést zvoleným postupem ono známé číslo. Pokud dostanete správný výsledek, je i způsob převodu správný.

Příklad: Převeďte číslo -12 na 8bitové binární číslo *signed* (binární číslo se znaménkem ve dvojkovém doplňku).

- Výpočet podle a) Absolutní hodnota |-12| = 12. Převedeme 12 jako 8bitové binární číslo *unsigned* na 00001100. Vypočteme jeho jedničkový doplněk 11110011 a k němu přičteme 1, tedy 11110011+00000001 = **11110100.**
- Výpočet podle b): |-12|-1 = 11. Dekadické 11 číslo bude jako 8bitové binární číslo *unsigned* = 00001011. Od něho jedničkový doplněk (bitová negace) je 00001011 → **11110100.**
- Výpočet podle c): 2^8-12 = 256-12 = 244. Dekadické číslo 244 jako 8bitové binární číslo *signed* je 11110100. A toto binární číslo **11110100** má dekadickou hodnotu, pokud ho bereme jako binární číslo *signed*, rovnou -12.

3.2.4 Převod binárních čísel na dekadická

Pro převod binárních čísel musíme opět znát bitovou délku požadovaného binárního čísla *signed*. Tu si znovu označme jako m.

- Binární čísla mající 0 v nejvyšším bitu se opět převádí na kladná dekadická čísla stejně jako čísla bez znaménka.
- Záporná binární čísla, tj. mající 1 v nejvyšším bitu, konvertujeme dále uvedenými postupy, které jsou opačnými k předchozím metodám v 3.2.3.
  a) vypočteme dvojkový doplněk binárního čísla a ten převedeme jako binární číslo *unsigned* na hodnotu, kterou označíme x. U ní změníme znaménko na mínus, tedy na -x.
  b) můžeme také udělat jen jedničkový doplněk, logickou negaci bitů, a výsledek převést na číslo bez znaménka, které označíme y. Hledané dekadické číslo -x se rovná: -x = -y-1.
  c) alternativně můžeme převést celé binární číslo na číslo jako číslo bez znaménka, výsledek označíme z. Hledané číslo -x = z-2^m.

  Příklad: Převeďte 000111000 jako 9bitové binární číslo *signed*.

  Výpočet: Nejvyšší bit je 0, takže binární číslo konvertujeme jako číslo bez znaménka, třeba sečtením vah jedničkových bitů na $2^5+2^4+2^3$ = 32+16+8=56.

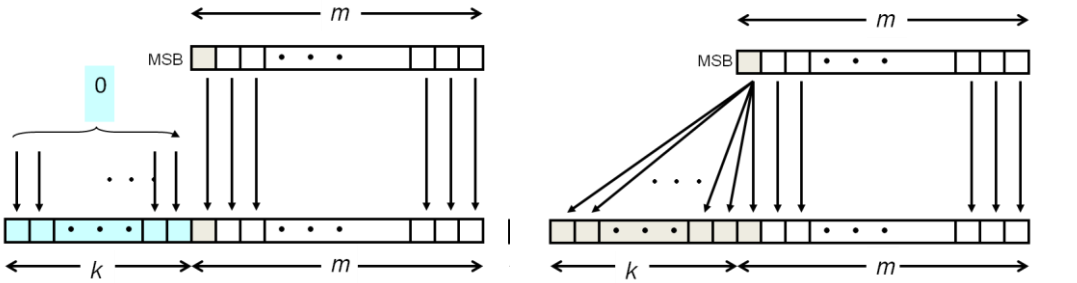
Příklad: Převeďte 8bitové binární číslo 11001100 jako binární číslo *signed*.

Výpočet: Nejvyšší bit je 1, takže převádíme metodami pro záporná čísla.

- Výpočet podle a) K binárnímu číslo 11001100 vypočteme jeho dvojkový doplněk 00110011+00000001=00110100. Ten převedeme jako binární číslo *unsigned* na 52. Hledané číslo je pak jeho negací, tedy **-52.**
- Výpočet podle b): Provedeme bitovou negaci (jedničkový doplněk) čísla 11001100 → 00110011 a výsledek převedeme jako binární číslo *unsigned* 00110011 → 51. Hledaný výsledek je -51-1 = **-52.**
- Výpočet podle c): Číslo 11001100 převedeme jako číslo bez znaménka třeba sčítání vah bitů jako 128+64+8+4=204. Hledané číslo je $204-2^8$= 204-256=**-52**.

3.2.5 Změna délky čísla - sign extension

Před binární číslo *unsigned* lze přidat bity 0, aniž se změní jeho hodnota, viz 3.1.1. Pro zachování hodnoty čísla *signed* se musí použít znaménkové rozšíření (angl. *sign extension*), při němž se zachovává nejvyšší bit určující, zda je číslo kladné nebo záporné.



Obrázek 20 - Znaménkové rozšíření pro binární čísla *signed*

Obrázek 20 ukazuje rozdíly mezi oběma způsoby uložení binárních čísel. Zatímco u neznaménkových čísel se vždy přidávají bity 0, u binárních čísel *signed* se do přidaných bitů kopíruje nejvyšší bit čísla, tj. *MSB*.

Naopak, zmenšujeme-li binární číslo *unsigned*, lze odstranit všechny levé bity 0, ale u binárních čísel *signed* lze rušit jen nejvyšší bity 0 u kladných čísel a bity 1 u záporných, ale výhradně jen v takové délce, aby se nezměnila kladnost/zápornost čísla, tj. po změně délky musí být hodnota nejvyššího bitu nového čísla shodná s nejvyšším bitem původního čísla.

Příklad 1: Rozšiřte 4bitové číslo *signed* 0111 na 8 bitů.
Řešení: Nejvyšší bit, tj. 0, se kopíruje do bitů vlevo, tedy výsledek 0000 0111.

Příklad 2: Rozšiřte 8bitové číslo *signed* 1000 0010 na 16 bitů.
Řešení: Opět kopírujeme MSB=1 do přidaných bitů, tedy výsledek 1111 1111 1000 0010.

Příklad 3: Jaký je nejmenší možný počet bitů pro 8bitové číslo *signed* 1110 0100?
Řešení: Nejmenší možná bitová délka pro jeho uložení je 6 bitů, tedy 10 0100.

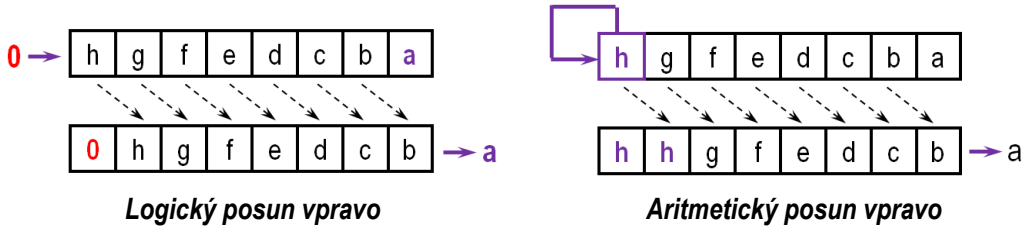
Příklad 4: Jaký je nejmenší možný počet bitů pro 8bitové číslo *signed* 0000 0100?
Řešení: Nejmenší možná bitová délka pro jeho uložení jsou 4 bity, tedy 0100.

Poznámka: Typ použitého čísla určuje, zda se musí provést znaménkové rozšíření. Ve strojovém kódu procesorů kvůli tomu existují odlišné instrukce pro nahrání dat menší velikosti do

větší. Například procesor MIPS nahrává neznaménkový byte do 32bitového registru instrukcí LBU, ale pro byte obsahující číslo se znaménkem ve dvojkovém doplňku má instrukci LB, která provádí znaménkové rozšíření. Procesory typu x86 zas používají instrukce MOV a MOVSX. Překladače vyšších jazyků vybírají strojové instrukce podle typů převáděných čísel.

3.2.6 Logické a aritmetické posuny

Nutnost zachovat znaménkový bit si vyžaduje i různé operace posunů (angl. *shift*) s binárními čísly. Rozlišují se logické posuny doleva a doprava, kdy se obsah posouvá jako binární číslo *unsigned*, a aritmetické posuny doleva a doprava, které respektují znaménkový bit.



Obrázek 21 - Logický a aritmetický posun vpravo

Posuny vpravo jsou naznačené na Obrázek 21 pro 8bitové binární číslo "hgfedcba", kde ´a´ je nejnižším bitem a ´h´ nejvyšší bitem. Pokud máme v binárním čísle formát *unsigned*, pak provádíme posun vždy s přidáváním 0. Bereme-li binární číslo jako *signed*, pak provedeme aritmetický posun, při němž zůstává jeho nejvyšší bit pevně namístě, zde ´h´.

Například řetězec 8 bitů 11101011, s nejvyšším bitem 1, se po logickém posunu změní na 01110101, zatímco po aritmetickém posunu na 11110101. Naproti tomu u 8bitového řetězce 01101010, jehož nejvyšší bit je rovný 0, logický i aritmetický posun dávají stejné výsledky 00110101.

Vstup	Hodnota jako	Posun vpravo	Výsledek posunu	Hodnota jako
11101011	*unsigned*= 235	logický	01110101	*unsigned*= 117
	signed= -21	aritmetický	11110101	*signed*= -11
01101010	*unsigned*= 106	logický	00110101	*unsigned*= 53
	signed= 106	aritmetický	00110101	*signed* = 53

Z tabulky nahoře můžeme odvodit, že logický posun doprava je vhodné použít pro binární čísla *unsigned*, u nichž odpovídá dělení 2 s tím, že nejnižší bit ztracený operací posunu doprava je zbytkem po dělení.

Aritmetický posun doprava je nutné použít u binárních čísel *signed*, aby se nám zachoval znaménkový bit. Pro kladná čísla je výsledek rovný dělení 2, a to opět se zbytkem po dělení, kterým je ztracený nejnižší bit. U záporných čísel je však podíl zaokrouhlený směrem k nižšímu číslu, tedy se provádí operace -21/2 = floor(-10.5) = -11, kde floor() označuje zaokrouhlení na celé číslo směrem dolů.[9] Aritmetické posuny doprava interpretované jako dělení 2 dávají paradoxně třeba pro číslo -1 výsledek -1/2 = -1.

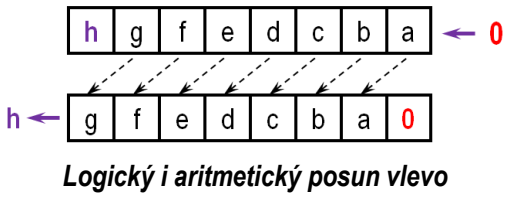
Pokud chceme aritmetické posuny doprava použít místo dělení 2 pro binární čísla *signed*, pak musíme pro záporná čísla provést někdy korekci výsledku, abychom obdrželi ná-

[9] V jazyce C je floor(...) funkce, v Java je floor(...) metoda, v C# zná metodu Math.Floor(...)

mi očekávané číslo. Korekce je však velmi jednoduchá, přičíst 1 ve vybraných případech.[10] Nutno si však pamatovat, že obecně nelze aritmetický posun doprava pro záporná binární čísla *signed* brát automaticky jako přesnou analogii celočíselného dělení 2.

Posuny vlevo binárních čísel jsou shodné — logický posun se provádí stejně jako aritmetický, viz Obrázek 22, kde posun je opět naznačený pro 8bitové binární číslo "hgfedcba", v němž ´a´ je nejnižším bitem a ´h´ nejvyšší bitem, který se při posunu vlevo ztrácí.

Pokud nedojde při posunu k aritmetickému přetečení, pak posun vlevo odpovídá násobení 2, jak pro binární čísla *unsigned*, tak pro binární čísla *signed*.



Obrázek 22 - Posun vlevo 8bitového binárního čísla

Jediný rozdíl u obou reprezentací čísel je podmínka, kdy při posunu vlevo dojde k aritmetickému přetečení, po němž hodnota výsledku už neodpovídá původnímu číslu vynásobenému 2. U binárního čísla *unsigned* nastane aritmetické přetečení v okamžiku, kdy se jeho nejvyšší bit, který se při posunu ztratil, rovnal 1. U binárních čísel *signed* ale nastane aritmetické přetečení v tom okamžiku, kdy se změní hodnota nejvyššího bitu.

Například, máme-li 8bitové binární číslo *signed* 11101011 (dekadicky -21), pak po jeho prvním posunu vlevo dostaneme 11010110 (-42) a po druhém 10101100 (-84). Provedeme-li třetí posun vlevo, pak obdržíme 01011000 (dekadicky 88), neboť pro binární číslo *signed* došlo k přetečení.

Jiný příklad: mějme 8bitové číslo *signed* 00110010 (dekadicky 50). První posun vlevo dá výsledek 01100100 (100). Další posun dá binární číslo 11001000, které má hodnotu jako 8bitové číslo *signed* -56, čili máme přetečení. Všimněte si, že při stejném vstupu, ale značícím tentokrát binární číslo *unsigned*, by výsledek byl stále dvojnásobkem předchozí hodnoty, a to 200. K přetečení by u binárního čísla *unsigned* došlo až při dalším posunu vlevo, jehož výsledek by byl 10010000. Ten má jako binární číslo *unsigned* hodnotu 144.

Poznámka: Všimněte si, že pojem aritmetického přetečení závisí na tom, jak interpretujeme binární číslo. Pokud ho bude brát jako obyčejný řetězec bitů, bez stanovené číselné hodnoty, pak posuny budou pouhou změnou pozic bitů, jakousi analogií dopravníkového pásu, který posune o jednu pozici každý bit a bit ležící na konci pásu přitom vypadne ven.

[10] Obecně možno říct, že binární čísla *unsigned* i *signed* mohou v procesorech používat stejnou aritmetickou jednotku, což je hlavní jejich výhodou. Pouze v některých případech vyžaduje manipulace s binárními čísly *signed* nepatrně odlišné instrukce. Dále u operací násobení nebo dělení, když jsou jejich operandy záporná binární čísla *signed*, se někdy musí provést nenáročná korekce výsledku. Nicméně procesory často obsahují speciální jednotky pro násobení záporných binárních čísel *signed*, a to pro urychlení operací s nimi. Záporná binární čísla *signed* blízká nule mívají totiž hodně bitů v 1 a na běžných násobičkách by operace s nimi trvaly příliš dlouho. Přesné popisy přesahují rámec této publikace. Zájemci mohou hledat třeba " Boothův algoritmus".

3.2.7 Aritmetické přetečení při sčítání a odčítání

V kapitole 3.1.5 na straně 26, kde jsme rozebírali sčítání a odčítání binárních čísel *unsigned*, jsme detekovali aritmetického přetečení pomocí *Carry*, tj. přenosu z nejvyššího řádu. *Carry* ale nemá žádný praktický význam pro binární čísla *signed*, neboť se generuje běžně; čísla sama jsou na něm založená, viz popis dvojkového doplňku v kapitole 3.2 na straně 28.

U binárních čísel *signed* dochází k aritmetickému přetečení při překročení hranice mezi jejich největším a nejmenším čísel. Například, máme-li 8bitová čísla *signed*, pak jejich největší číslo 127 je uloženo 0111 1111 a pokud k němu přičteme 1, tj. 0000 0001, pak dostaneme 1000 0000, jejich nejmenší číslo -128. Obdržíme paradoxně záporný výsledek po součtu dvou kladných čísel. Analogicky bychom při odčítání 1 od -128 dostali zase kladný výsledek 127.

	Overflow	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
126		0	1	1	1	1	1	1	0
+1		0	0	0	0	0	0	0	1
127	0	0	1	1	1	1	1	1	1
+1		0	0	0	0	0	0	0	1
-128	1	1	0	0	0	0	0	0	0
+1		0	0	0	0	0	0	0	1
-127	0	1	0	0	0	0	0	0	1

Tabulka 4 - Aritmetické přetečení při sčítání 8bitových binárních čísel *signed*

ALU procesorů detekují podobné situace na základě tabulky znaménkových bitů operandů a výsledku. Pokud je výsledek evidentně nesprávný generují příznak (*flag*) *Overflow*.

operand 1	operace	operand 2	výsledek
kladný	+	kladný	záporný
záporný	+	záporný	kladný
záporný	-	kladný	kladný
kladný	-	záporný	záporný

Tabulka 5 - Kdy se objeví příznak *overflow* u operace s binárními čísly *signed*

Přesněji ALU nastaví po každém sčítání a odčítání binárních čísel nejméně čtyři základní aritmetické příznaky[11] na 0 nebo na 1, podle toho, zda situace nastala:

- *Carry* - příznak přenosu z nejvyššího řádu binárního čísla, tj. ztracený nejvyšší bit[12];
- *Overflow* - příznak přetečení pro binární čísla *signed;*
- *Sign* - kopie nejvyššího bitu čísla;
- *Zero* - pokud výsledek obsahuje samé bity 0.

Sčítání i odčítání binárních čísel *signed* a *unsigned* se provádí úplně stejně, a tak většina ALU vždy nastaví všechny zmíněné příznaky. Program si sám musí otestovat příznak, na kterém mu záleží, a to podle toho, jaký formát uvažoval u vstupních operandů.

Zmíněné hlavní aritmetické příznaky se samozřejmě generují i pro další aritmeticko-logické operace, ale u nich platí jiné podmínky; ty jsou vždy specifikované v popisu jednotlivých instrukcí procesoru.

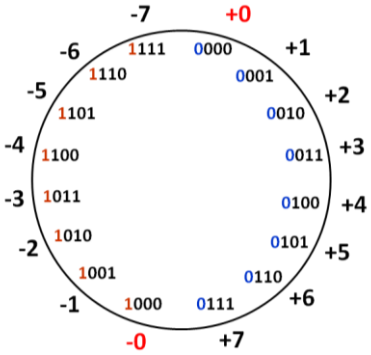
[11] U procesoru bývají příznaky uložené registru zvaném "*flag register*" či "*status register*" spolu s dalšími jinými příznaky.

[12] ALU nastavuje *Carry* i jinde, například i u operací logických posunů (*shift*), zmíněných v předchozí kapitole. U posunu doprava je pak *Carry* nejnižším bitem, který se po posunu ztrácí.

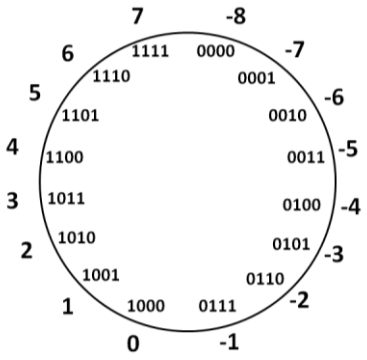
3.3 Celá čísla se znaménkem v přímém a aditivním kódu

Pro čísla se znaménkem existují i další způsoby kódování. Pro úplnost se zmíníme o dvou velmi rozšířených metodách, a to přímém a aditivním kódu.

Číslo se znaménkem můžeme uložit tak, že zakódujeme absolutní hodnotu čísla způsobem binárního čísla *unsigned* a k ní přidáme znaménkový bit. Tento způsob se nazývá přímý binární kód (angl. *straight binary*) nebo kód znaménko a hodnota (angl. *Sign and magnitude representation* či zkráceně jen *Sign–magnitude*).



Obrázek 23 - Znaménko a hodnota



Obrázek 24 - Aditivní kód s K=8

Přímý kód pro 4bitová binární čísla ukazuje Obrázek 23. V kódu existují dvě nuly, záporná a kladná. Pokud se nám například během iterací výsledek limitně přiblížil k nule, známe směr, ze kterého se k ní dostal. Další předností kódu je velmi rychlá aritmetická negace, kterou tvoříme pouze změnou nejvyššího bitu.

Příklad: Zakódujte číslo -15 v 8bitovém přímém kódu.

Řešení: 8bitový přímý kód má jeden bit znaménka a sedm bitů absolutní hodnoty čísla. Zakódujeme tedy |-15| na 7 bitů jako binární číslo *unsigned* 000 1111 a před něj doplníme bit znaménka, zde 1, protože číslo je záporné. Dostaneme výsledek 1000 1111.

Další zakódování čísel se znaménkem se v češtině označuje jako **aditivní kód** nebo jako kód s posunutou nulou. V angličtině pro něj existují názvy *Excess*-K nebo *offset binary* či *biased representation*. V něm se celá čísla znaménkem napřed převedou na čísla nezáporná, a to přičtením pevně dané konstanty K, zvolené tak, aby výsledek byl vždy kladné číslo. To pak zakódujeme jako binární číslo *unsigned*.

Při zvolené bitové délce binárního čísla m a konstantě K dostaneme rozsah zobrazitelných celých čísel od $-K$ do 2^m-1-K.

Například pro 4bitová čísla a hodnotu $K=8$, dostaneme rozsah od -8 do 7, viz Obrázek 24. Rozsah zobrazitelných záporných čísel můžeme nastavit dle potřeby volbou hodnoty K. Zvolíme-li například K=30, pak pro 4bitová binární čísla dostaneme zobrazitelný rozsah dekadických od -30 do -15.

Příklad: Zakódujte na -15 jako 5 bitové binární číslo v aditivním kódu **+16.**

Řešení: -15 **+16** = 1. Dekadické číslo 1 jako 5bitové binární číslo je 00001.

Aditivní kód místo čísel x a y provádí aritmetické operace s čísly $(x+K)$ a $(y+K)$, takže například výsledek součtu čísel $x+y$ je číslo $(x+y)+2*K$, což znamená, že se výsledek musí vždy ko-

rigovat odečtením K. Korekce násobení jsou ještě složitější a s čísly v aditivním vůbec nelze přímo provést dělení.

Přímé a aditivní kódy se obecně nehodí pro bezprostřední výpočty, protože běžné procesory s nimi neumí pracovat. Používají se však pro přenos čísel, jako vnitřní kódy nebo pro u složených čísel, jako například u čísel v pohyblivé řádové čárce zakódovaných podle normy IEEE 754. Tu používají skoro všechny moderní procesory pro typy *float*, *double* i *extended*. IEEE 754 ukládá mantisu čísla v přímém kódu a exponent čísla v aditivním kódu[13].

3.4 Hexadecimální notace

Hexadecimální notace je způsob zkráceného zápisu binárních řetězců. Každou jejich čtveřice bitů zakódujeme jako 4bitové binární číslo *unsigned*. Pro zachování zápisu čtveřic jedním znakem se pro hodnoty 10 až 15 používají písmena A až F.

Příklad 1: Jak je binární řetězec 10100111 v hex. notaci?
Řešení: Řetězec rozdělíme na čtveřice 1010 0111 a každou z nich zakódujeme, takže dostaneme **A7**

Příklad 2: Řetězec 11100110101011 převeďte na hex. notaci:
Řešení: Řetězec rozdělíme opět na čtveřice, a to od jeho pravé strany na 11 1001 1010 1011. Levou skupinu, která má jen 2 bity, doplníme nulami, tedy na 0011 1001 1010 1011, což zakódujeme na **39AB**

Příklad 3: Hexadecimální notaci 1F přepište na 6bitový binární řetězec.
Řešení: Přímým přepisem dostaneme 0001 1111 a vezmeme posledních 6 bitů, tedy **01 1111**

Binární řetězec	Znak	Hodnota *unsigned*
0000	**0**	0
0001	**1**	1
0010	**2**	2
0011	**3**	3
0100	**4**	4
0101	**5**	5
0110	**6**	6
0111	**7**	7
1000	**8**	8
1001	**9**	9
1010	**A**	10
1011	**B**	11
1100	**C**	12
1101	**D**	13
1110	**E**	14
1111	**F**	15

Samotný zápis v hexadecimální notaci vede implicitně na počet bitů dělitelný čtyřmi, avšak je možné jím zaspat i menší délku, pokud přidáme specifikaci bitové délky.

3.4.1 Hexadecimální čísla

Hexadecimální číslo znamená, že bitový řetězec zapsaný v hexadecimální notaci interpretujeme jako binární číslo *unsigned*. Pro zápis čísla existuje několik různých způsobů. Některé z nich si ukážeme na hodnotách A7 a 39AB z příkladu 1 a 2 nahoře.

a) 0A7H , 39ABH Hexadecimální notace se zakončí příponou H a pokud začíná písmenem, přidá se prefix 0 pro zdůraznění číselné hodnoty.
b) 0xA7, 0x39AB Před číslo se připojí prefix 0x.
c) X"A7", X"39AB" Zápis v jazyce VHDL.
d) 16#A7, 16#39AB Zápis v jazyce PostScript.
...a mnoho další formátů, viz Wikipedie, heslo *Hexadecimal*.

[13] S čísly IEEE 754 se většinou nepočítá přímo, ale před aritmetickými operacemi se vždy rozloží na mantisu a exponent, které se zpracují zvlášť, a výsledek se znovu zakóduje. Některé operace lze však provést přímo se zakódovaným číslem. Uložení mantisy v přímém kódu dovoluje rychlou aritmetickou negaci čísla, a to pouhou změnou jednoho bitu. Rovněž lze dvě čísla IEEE 754 vzájemně porovnat přímo v zakódovaném tvaru, tedy stejně tak rychle jako dvě čísla integer.

V programovacích jazycích se však konstanta (*literal*) zapsaná jako hexadecimální číslo může i brát jako binární číslo *signed*, pokud ji přiřazujeme do proměnné typu *signed*. Situaci si ukážeme na programu v jazyce C++, přeloženém tak, že proměnná *int* má délku 4 byty.[14]

```
int intsize = sizeof(int);       // intsize = 4 (byty)
unsigned char uc = 0xFF;         // uc = 255
char sc = 0xFF;                  // sc = -1
unsigned short int usi = 0xFFFF; // usi = 65535
short int ssi = 0xFFFF;          // ssi = -1
unsigned int ui = 0xFFFFFFFF;    // ui = 4294967295
int si = 0xFFFFFFFF;             // si = -1
```

Jak vidíte z ukázky, konstanta 0xFF se nemusí vždy rovnat 255, viz proměnná sc.

3.4.2 Číselné soustavy

Hexadecimální čísla se často zavádějí jako čísla o základu (*radix*) 16. Taková definice všem přímo indukuje, že číslo je typu *unsigned*, a kvůli tomu jsme se jí zatím vyhnuli.

$$x_{16} = \sum_{k=0}^{m-1} a_k 16^k ; \quad 16 > a_k \geq 0 \qquad (3)$$

Hodnotu hexadecimálního čísla x_{16}=0xA7 podle (3) vypočteme jako $x_{16}=10*16^1+7*16^0 = 167$. Z matematického hlediska lze jako základ zvolit libovolné celé číslo větší než jedna. Zvolíme-li například r=10, dostaneme dekadická čísla. Označme-li obecný nenulový základ **r**, pak číslo x_r je v soustavě se základem **r** dáno vzorcem:

$$x_r = \sum_{k=0}^{m-1} a_k \mathbf{r}^k ; \quad \boldsymbol{r} > a_k \geq 0, \ \boldsymbol{r} > 1 \qquad (4)$$

Hodnoty a_k jsou čísla, ale nahrazují se jedním znakem, podobně jako v hexadecimální notaci. V obvodech a počítačích se upřednostňují čísla s radixy $\mathbf{r}=2^m$ dovolující rychlé převody na binární čísla a efektivní uložení v paměti. Exponent m určuje délku skupiny bitů, která se kóduje jedním znakem. Běžné používané soustavy jsou v tabulce dole:

Název čísla	**Základ (radix) r**	**Počet bitů skupiny**
Binární	2	1
Oktalové	8	3
Hexadecimální	16	4
Base32	32	5
Base64 nebo **Radix64**	64	6
Dekadické	*10*	*- nelze převádět po bitových skupinách-*

Tabulka 6 - Běžně používané základy (radixy) pro číselné soustavy

V následujících odstavcích se krátce zmíníme o dosud neprobraných kódech.

3.4.2.1 *Oktalová čísla*

Zvolíme ve vzorci (4) hodnotu r=8, pak dostaneme kódování ve skupinách po třech bitech, které se nazývá oktalová (nebo též oktální) čísla, v angličtině "*octal numeral system*" nebo zkráceně "*oct*",

[14] Velikost int závisí na překladači. V 64bitovém prostředí by sice mohla být i 8 bytů, tj. 64 bitů, ale mnohé překladače z důvodů zpětné kompatibility volí ve výchozím nastavení i zde int jako 4byty, tj. 32 bitů.

Například hexadecimální číslo 0xA7, s dekadickou hodnotou 167, je zakódováno jako binární řetězec 1010 0111. Ten rozdělíme odprava na trojice 10 100 111 a každou zakódujeme jako binární číslo *unsigned*. Výsledek bude oktalové číslo 247. Oktalová čísla se někdy zapisují s příponou Q, aby se zdůraznil jejich charakter, tedy 247Q.

Oktalová čísla se dříve hodně používala v telekomunikační technice, ale dnes se vyskytují pouze výjimečně. Prakticky jediné častější uplatnění jsou příkazy *chown* a *chmod* používané v implementacích Unixu (zkratky pro *change owner* a *change mode*), kde se jejich argumenty zadávají jako oktalová čísla (bez přípony Q).

3.4.2.2 Base32 a Base64

Při volbě základu 32 dostaneme kódování po pěticích a při základu 64 po šesticích bitů. Jde o velmi hojně používané způsoby pro úsporný textový přenos binárních řetězců, například šifrovacích klíčů mezi počítači. Často se v nich zadávají i kódy pro aktivaci programů.

Kódování Base32 a Base64 nejsou tak průhledná jako hexadecimální čísla, protože málokdy reprezentují nějakou číselnou hodnotu podle vzorce (4). Kódované skupiny o délce 5, či respektive 6 bitů, nejsou soudělné s běžnými velikostmi čísel, a to 1, 2, 4 a 8 bytů, takže se nevyplatí kódovat oddělená čísla, protože by se nedosáhla významná úspora. Skupiny čísel se proto často spojují v jeden dlouhý binární řetězec, který se zakóduje jako celek. Při dekódování se zase rozdělí na řadu čísel.

Dělení velmi dlouhých binárních řetězců začíná zpravidla odleva, a lze užít různé přiřazení znaků číslům a_k ze vzorce (4). Existuje několik kódovacích tabulek, zavedených jednotlivými výrobci, podle kterých se převádějí hodnoty od 0 do 31 pro Base32 a od 0 do 63 pro Base64 na alfanumerické znaky. Tabulky pro kódování jsou navržené tak, aby se vyloučila záměna podobných znaků, jako třeba malého písmena l a číslice 1. Base32 využívá jen číslice a písmena a malé i velké písmeno hodnotí stejně. Base64 již rozlišuje mezi malými a velkými písmeny, malému písmenu je přiřazena jiná hodnota než velkému.

Na konec kódů Base32 a Base64 se připojuje sekvence =, což se nazývá *pad* (oddělení), slouží pro specifikaci délky a udává i konec, aby se textový řetězec mohl poslat ve více textových řádcích. Blíže viz Wikipedie, hesla Base32 a Base64.

Srovnání jednoho možného zakódování pro Base32 a Base64 s jinými kódy.:

Dekadické číslo 1234567890

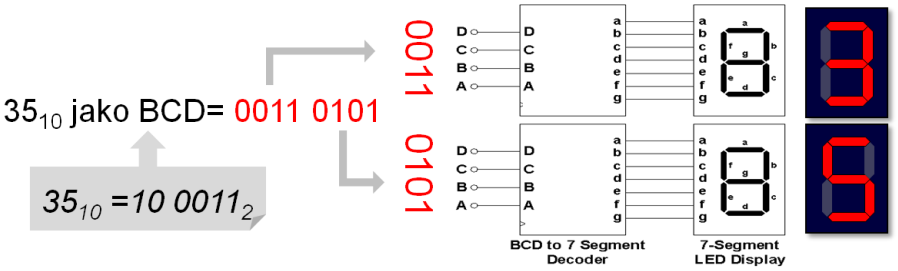
zapsáno jako	zakódované jako řetězec
• dekadické	`1234567890`
• binární (*un*)*signed*	`0100 1001 1001 0110 0000 0010 1101 0010`
• hexadecimální	`499602D2`
• oktalové	`11145401322`
jeho dělení na bity	`01 001 001 100 101 100 000 001 011 010 010`

v bitovém řetězci doplněném na celý počet skupin

• Base32 RFC4648	**`JGLAFUQ=`**
• bity	`01001 00110 01011 00000 00101 10100 10+000`
• Base64 original	**`SZYC0g==`**
• bity	`010010 011001 011000 000010 110100 10+0000`

3.5 BCD - Binary Coded Decimal

BCD kódování čísel patří k nejstaršímu používanému způsobu. Jde o přímý zápis dekadického čísla, protože každou jeho číslici zakódujeme jako binární číslo *unsigned* a uložíme do čtyř bitů. Například dekadické číslo 35 uložíme jako 0011 0101.



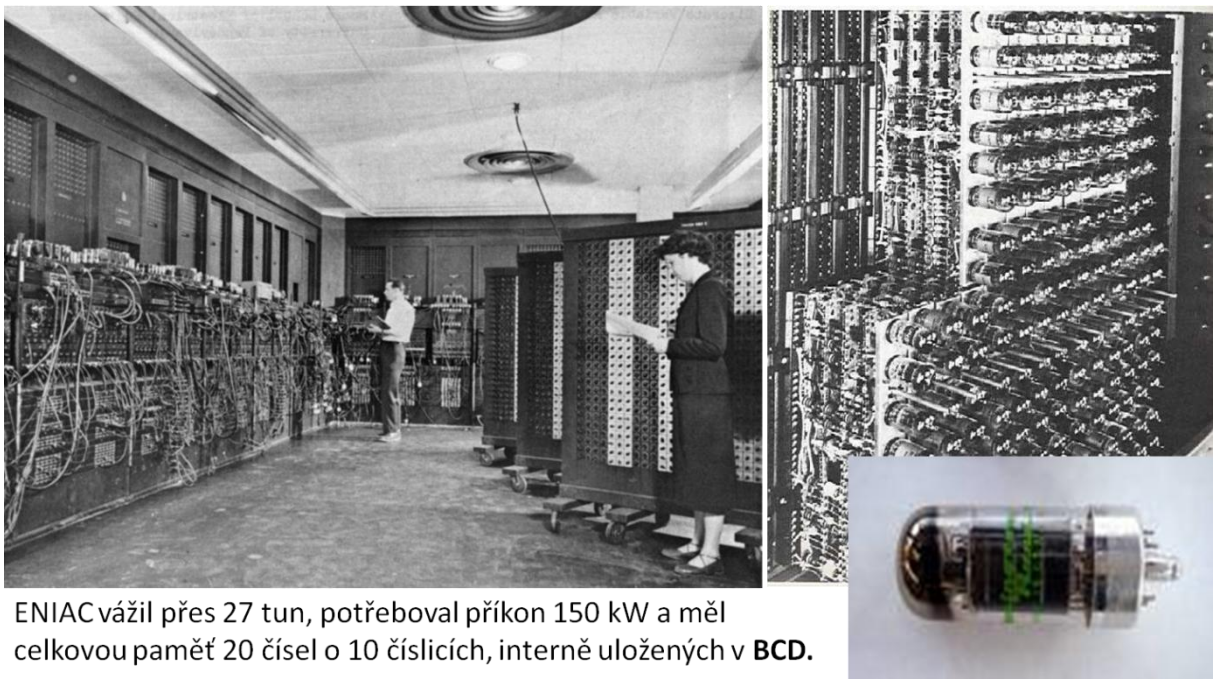
Obrázek 25 - BCD číslo 35

Výhodou BCD kódování je velká čitelnost binárního čísla, protože jsme schopni přímo z jeho binárního kódování vidět dekadické číslo. Převody na BCD tvar se používají s výhodou u zobrazovacích jednotek, např. chceme-li číslo zobrazit na 7segmentovém displeji, musíme ho napřed převést na BCD tvar, viz Obrázek 25. Stejně tak při tisku čísla, například pomocí C funkce printf(), se číslo převede na BCD tvar a ten na znaky číslic.

BCD čísla kódují číslice do čtveřic bitů, stejně jako hexadecimální čísla, však na rozdíl od nich nevyužívají celý rozsah 4bitových čísel *unsigned* od 0 do 15, ale pouze hodnoty 0 až 9. Pokud by se ve čtveřici bitů ocitla hodnota mimo tento rozsah, nejde o platné BCD číslo. Při zakódování dekadického čísla 9876543210 bude výsledné BCD číslo mít 4*10 bitů, tedy 40 bitů:

Dekadické číslo	9	8	7	6	5	4	3	2	1	0
BCD číslo	**1001**	**1000**	**0111**	**0110**	**0101**	**0100**	**0011**	**0010**	**0001**	**0000**

První počítače pracovaly v BCD, a to právě pro snadnou čitelnost hodnoty pro lidské operátory, jako třeba vůbec první elektronický počítač (*Electronic Numerical Integrator and Computer*) vyrobený v roce 1946, viz Obrázek 26 [fotografie z Wikipedie, heslo ENIAC].



Obrázek 26 - ENIAC Electronic Numerical Integrator and Computer

Z hlediska uložení BCD čísel v počítači se rozlišují dva tvary BCD čísel. *Unpacked* BCD ukládá každou BCD číslici v samostatném bytu a hodí se pro numerické operace. *Packed* BCD ukládá v jednom bytu 2 BCD číslice a slouží pro úsporné uložení čísel. Mikroprocesory dovedou většinou provádět aritmetické operace jen s *unpacked* BCD čísly.

Dnes se přímé počítání s BCD čísly provádí pouze výjimečně, protože je zhruba 2krát až 3krát pomalejší než s binárními čísly a navíc vyžaduje více přístupů do paměti. Používá se však převod na BCD, protože ten potřebujeme před každým tiskem čísel.

Binární číslo lze samozřejmě převést na BCD pomocí dělení deseti a počítání zbytků po dělení, ale takový postup je zbytečně pomalý. Existuje mnohem rychlejší metoda, která se dá na úrovni asembleru velmi rychle algoritmizovat, a to převod pomocí posunů binárního doleva pomocí Hornerova schématu, viz Postup 2 v kapitole 3.1.3 začínající na str. 24. Metoda se dá navíc provádět přímo s *packed* BCD čísly a i snadno paralelizovat na úrovni obvodů. Potřebujeme pro ni jedině operaci násobení BCD číslem 2.

3.5.1 Násobení BCD číslem 2

Pokud číslo BCD obsahuje číslici 0 až 4, pak ji snadno vynásobíme 2 stejně jako binární číslo pomocí posunu doleva o jeden bit, protože výsledek bude 0 až 8, což je platný BCD rozsah. Potíž nastane při násobení dvěma BCD číslice v rozsahu 5 až 9, protože výsledek bude 10 až 18, což již není BCD číslice.

Můžeme si však pomoci tak, že k BCD kódu číslice o hodnotě 5 až 9 před posunem připočteme 3.

Proč přičítáme 3? Jde o polovinu délky rozsahu, který chybí v BCD kódování. To používá pouze hodnoty 0 až 9 z celého rozsahu od 0 do 15 pro 4bitová *unsigned binary* čísel Chybí v něm tedy 6 hodnot, a to do 10 do 15. Kvůli tomu se před posunem doleva (násobením 2) připočítává korekce +3, tedy polovina délky chybějícího rozsahu, ke všem číslicím větším než 4. Při posunu doleva (násobení 2) pak dojde u korigovaných BCD číslic automaticky k přeskočení 6 chybějících čísel, a tak dostaneme správný výsledek.

BCD	Korekce	Před posunem doleva	Po posunu
0000 0000 [0 | **0**]		0000 0000 [0 | 0]	0000 0000 [0 | **0**]
0000 0001 [0 | **1**]		0000 0001 [0 | 1]	0000 0010 [0 | **2**]
0000 0010 [0 | **2**]		0000 0010 [0 | 2]	0000 0100 [0 | **4**]
0000 0011 [0 | **3**]		0000 0011 [0 | 3]	0000 0110 [0 | **6**]
0000 0100 [0 | **4**]		0000 0100 [0 | 4]	0001 0100 [0 | **8**]
0000 0101 [0 | **5**]	+[0 | 3]	0000 1000 [0 | 8]	0001 0000 [**1** | **0**]
0000 0110 [0 | **6**]	+[0 | 3]	0000 1001 [0 | 9]	0001 0010 [**1** | **2**]
0000 0110 [0 | **7**]	+[0 | 3]	0000 1010 [0 | 10]	0001 0100 [**1** | **4**]
0000 0110 [0 | **8**]	+[0 | 3]	0000 1011 [0 | 11]	0001 0110 [**1** | **6**]
0000 0110 [0 | **9**]	+[0 | 3]	0000 1100 [0 | 12]	0001 1000 [**1** | **8**]

Korekce může způsobit dočasný vznik neplatných hodnot pro BCD číslice, v tabulce nahoře 1010, 1011 a 1100, viz poslední řádky, ale ty se vzápětí logickým posunem doleva převedou na správné hodnoty.

3.5.2 Převod binárního čísla unsigned na BCD

Algoritmizaci převodu si ukážeme 8bitovém binárním číslu *unsigned* 01110011, hex 0x73, s dekadickou hodnotou 115.

Postup:

a) Na začátku každého kroku se napřed projdou všechny jednotlivé 4bitové BCD číslice. Ke každé, která je větší než číslo 4 (0100), se připočte binárně 3 (0011). Přičítání se provádí izolovaně pro každou BCD číslici, tedy jako operace se 4bitovým číslem *unsigned*. Výsledek může po přičtení být větší než 9 (1001) - to se však ihned zkoriguje v kroku b).

b) Dále se provede logický posun vlevo celého BCD čísla a převáděného binárního čísla jako jednoho celistvého řetězce bitů.

Pro 8bitové číslo se kroky a) a b) opakují v cyklu celkem 8krát.

Operace	*packed* **BCD**		**Binární číslo**
Inicializace	**[0 | 0 | 0]**	0000 0000 0000	**0**1110**0**1**1**
po 1. společném posunu BCD a bin. čísla	**[0 |0 | 0]**	0000 0000 000**0**	**1**1100**1**1**0**
po 2. společném posunu BCD a bin. čísla	**[0 |0 | 1]**	0000 0000 00**0**1	**1**1001**1**00
po 3. společném posunu BCD a bin. čísla	**[0 | 0 | 3]**	0000 0000 0**0**11	**1**0011**1**000
po 4. společném posunu BCD a bin. čísla	**[0 | 0 | 7]**	0000 0000 **0**111	**0**0110**1**0000
Skupina 0111 > 0100	+[0 | 0 | 3]	+0000 0000 0011	
výsledek korekce	[0 | 0 | 10]	0000 0000 1010	**0**0110**1**0000
po 5. společném posunu BCD a bin. čísla	**[0 | 1 | 4]**	0000 0001 0100	**0**1100**1**0000
po 6. společném posunu BCD a bin. čísla	**[0 | 2 | 8]**	0000 0010 1000	**1**1000**1**0000
Skupina 1000 > 0100→ korekce	+[0 | 0 | 3]	+0000 0000 0011	
výsledek korekce	[0 | 2 | 11]	0000 0010 1011	**1**1000**1**0000
po 7. společném posunu BCD a bin. čísla	**[0 | 5 | 7]**	0000 0101 0111	**1**0000**1**0000
Skupiny 0101 a 0111 >= 0100→ korekce	+[0 | 3 | 3]	+0000 0011 0011	
výsledek korekce	**[0 | 8 | 10]**	0000 1000 1010	**1**0000**1**0000
po 8. společném posunu BCD a bin. čísla	**[1 | 1 | 5]**	0001 0001 0101	00000000
konec - BCD udává výsledek			

Algoritmus převodu je relativně jednoduchý a mechanický a lze ho rozšířit i na delší čísla. Operace současného posunu vlevo několika čísel najednou se procesoru provádí snadno na úrovni strojového kódu, protože procesory bit, který se při posunu vypadává ven, dávají obvykle do Carry příznaku, a ten se dá použít jako vstup pro další posun. Není tedy problém v asembleru provést logický posun pro libovolně dlouhý řetězec čísel.

Jazyk C, či jiný obdobný vyšší programovací jazyk, ale nemá ve své syntaxi konstrukce, které by dovolily snadno spojit posuny více proměnných. Procesor je hravě umí, ale programovací jazyk je nedovede elegantně zapsat.

Nejúspornější cesta k jejich naprogramování vede přes společné uložení BCD a binárního čísla do jednoho čísla typu *unsigned int* či *unsigned long*, podle potřebné délky. Máme pak komplikovanější testy skupin 4 bitů, ale pořád kód vyjde efektivněji než při snaze naprogramovat přenos bitů mezi posuny oddělených proměnných.

```
typedef unsigned char byte;
struct BCD_t { byte digit[3]; };

void byte2bcd(byte bin, BCD_t & bcd)
{
   // 3 BCD digits are stored in bits 19 down to 8; binary input bin in bits 7 down to 0
   unsigned int BCD_plus_bin = bin;
   for (int i = 1; i <= 8; i++) // counter of bit shifts
   {
        //For 8bits input bin, the correction is only performed for lower BCD digits [1] and [0],
        // because the upper BCD digit [2] can contain only values from 0 to 2
        if ((BCD_plus_bin & 0xF00) > 0x400) BCD_plus_bin += 0x300;
        if ((BCD_plus_bin & 0xF000) > 0x4000) BCD_plus_bin += 0x3000;
        BCD_plus_bin <<= 1;   // shift left together BDC + bin
   }
   // unpacking BCD digits to obtain result
   bcd.digit[0] = (BCD_plus_bin & 0xF00) >> 8;
   bcd.digit[1] = (BCD_plus_bin & 0xF000) >> 12;
   bcd.digit[2] = (BCD_plus_bin & 0xF0000) >> 16;
}
int main(int argc, char * argv[]) // a test of byte2bcd function
{
        byte VSTUP = 0x73; // bin=01110011
        BCD_t bcd;

        byte2bcd(VSTUP, bcd);

        for (int j = 2; j >= 0; j--) // direct print of characters
              putc((char)(bcd.digit[j] + '0'), stdout);

        putc('\n',stdout); // end of line

        return 0;
}
```

Poznámka 1: Kód by šel urychlit, protože korekce pro nejnižší BCD číslici [0] se může objevit až při i>=4 a v následující BCD číslici [1] až při i>=7.

Poznámka 2: Převod čísla na znak přičtením '0' bude vysvětlen v následující kapitole o kódování znaků ASCII, v odstavci "Důležité vlastnosti ASCII".

3.6 Kódování znaků ASCII

Kódování znaků zvané ASCII (*American Standard Code for Information Interchange*) vzniklo už v roce 1963, poté prošlo revizemi. Po poslední v roce 1986 se ustálila tabulka používaná dodnes a současně tvořící i základ pro novější kódy UTF8 nebo Unicode. Ty pro zpětnou kompatibilitu zachovávají číselnou hodnotu znaků definovaných v ASCII.

ASCII obsahuje 128 platných znaků o dekadické hodnotě 0 až 127, viz Tabulka 7 na str. 43. Tento rozsah lze uložit do 8bitového čísla jak typu *unsigned*, tak i *signed*.

Příklad: Převeďte řetězec "*Hello, Logic!*" na ASCII byty.

Řešení: Vyhledáme symboly v ASCII tabulce a zapíšeme jejich ASCII kódy třeba jako:

Znak	**H**	**e**	**l**	**l**	**o**	**,**		**L**	**o**	**g**	**i**	**c**	**!**
Hexadecimálně	48	65	6c	6c	6f	2c	20	4c	6f	67	69	63	21
Dekadicky	72	101	108	108	111	44	32	76	111	103	105	99	33

Tabulka 7 - ASCII kódování znaků

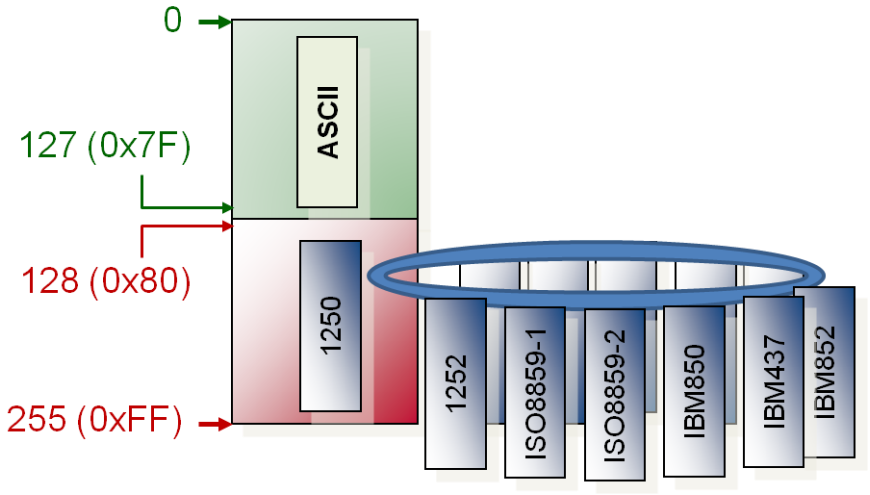
ASCII	Hex	Znak	ASCII	Hex	Znak	ASCII	Hex	Symbol	ASCII	Hex	Symbol
0	0x0	NUL	32	0x20	(mezera)	64	0x40	@	96	0x60	`
1	0x1	SOH	33	0x21	!	65	0x41	A	97	0x61	a
2	0x2	STX	34	0x22	"	66	0x42	B	98	0x62	b
3	0x3	ETX	35	0x23	#	67	0x43	C	99	0x63	c
4	0x4	EOT	36	0x24	$	68	0x44	D	100	0x64	d
5	0x5	ENQ	37	0x25	%	69	0x45	E	101	0x65	e
6	0x6	ACK	38	0x26	&	70	0x46	F	102	0x66	f
7	0x7	\a BEL	39	0x27	'	71	0x47	G	103	0x67	g
8	0x8	\b BS	40	0x28	(	72	0x48	H	104	0x68	h
9	0x9	\t TAB	41	0x29	)	73	0x49	I	105	0x69	i
10	0xA	\n LF	42	0x2A	*	74	0x4A	J	106	0x6A	j
11	0xB	\v VT	43	0x2B	+	75	0x4B	K	107	0x6B	k
12	0xC	\f FF	44	0x2C	,	76	0x4C	L	108	0x6C	l
13	0xD	\r CR	45	0x2D	-	77	0x4D	M	109	0x6D	m
14	0xE	SO	46	0x2E	.	78	0x4E	N	110	0x6E	n
15	0xF	SI	47	0x2F	/	79	0x4F	O	111	0x6F	o
16	0x10	DLE	48	0x30	0	80	0x50	P	112	0x70	p
17	0x11	DC1	49	0x31	1	81	0x51	Q	113	0x71	q
18	0x12	DC2	50	0x32	2	82	0x52	R	114	0x72	r
19	0x13	DC3	51	0x33	3	83	0x53	S	115	0x73	s
20	0x14	DC4	52	0x34	4	84	0x54	T	116	0x74	t
21	0x15	NAK	53	0x35	5	85	0x55	U	117	0x75	u
22	0x16	SYN	54	0x36	6	86	0x56	V	118	0x76	v
23	0x17	ETB	55	0x37	7	87	0x57	W	119	0x77	w
24	0x18	CAN	56	0x38	8	88	0x58	X	120	0x78	x
25	0x19	EM	57	0x39	9	89	0x59	Y	121	0x79	y
26	0x1A	SUB	58	0x3A	:	90	0x5A	Z	122	0x7A	z
27	0x1B	ESC	59	0x3B	;	91	0x5B	[	123	0x7B	{
28	0x1C	FS	60	0x3C	<	92	0x5C	\	124	0x7C	|
29	0x1D	GS	61	0x3D	=	93	0x5D	]	125	0x7D	}
30	0x1E	RS	62	0x3E	>	94	0x5E	^	126	0x7E	~
31	0x1F	US	63	0x3F	?	95	0x5F	_	127	0x7F	

Důležité vlastnosti ASCII

- Řídicí znaky - znaky s kódy 0 až 31 mají význam řídících kódů, které byly původně určené pro synchronizaci dálnopisů. Pro hlavní řídicí kódy, dnes stále používané, s hodnotami 7 až 13, existují i *escape* kódy v jazyce C, psané se zpětným lomítkem. Tabulka 7 je má vyznačené červeně. Zde se zmíníme jen BS - *backspace* ('\b'), TAB - tabulátor ('\t'), LF - *linefeed* - posun o řádek ('\n') a CR - *carriage return* - přesun na začátek řádku ('\r'). Úplný přehled řídicích znaků lze najít na Wikipedii, pod heslem ASCII.
- Čísla 0 až 9 mají kódy dekadicky 48 až 57, hexadecimálně 0x30 až 0x39. Převod mezi číslicemi a jejich číselnou hodnotou je proto snadný, stačí odečíst či přičíst 48 (0x30) od hodnoty znaku číslice, tedy ASCII hodnotu znaku '0'.
- Písmena - písmena jsou uložena v souvislých blocích v abecedním pořadí. Velká písmena od 65 do 90 (0x41 do 0x5A) a malá od 97 až 122 (0x61 do 0x7A), takže se snadno dá testovat, zda znak je písmeno, i abecedně třídit.
- Malé a velké písmeno mají kódy vůči sobě vzájemně posunuté vždy o 32 dekadicky, 0x20 hexadecimálně, takže převody mezi malými a velkými písmeny jsou rovněž velmi rychlé, stačí přičíst, resp. odečíst 32 (0x20) od hodnoty kódu znaku.

3.6.1 Extended ASCII

Základní kód ASCII končil u 127 (0x7f). Nepoužité hodnoty od 0x80 do 0xFF se později začaly využívat pro rozšíření ASCII (*extended* ASCII) o národní znaky. Pro češtinu šlo hlavně o znaky s diakritikou.



Obrázek 27 - Princip extended ASCII

Princip rozšíření ukazuje Obrázek 27. Zatímco v části od znaku 0 do 127 zůstávala kódovací tabulka neměnná, definovaná ASCII normou, tak v horní části od 128 do 255 se tabulka kódů měnila podle národní potřeby.

Zde existovalo mnoho různých kódových možností. Specifikace IBM OEM jich zahrnovala 81, podrobnější IANA (*Internet Assigned Numbers Authority*) jich registrovala 257, a ještě zdaleka neobsahovala všechny používané kódové stránky. Chyběla tam třeba česká kódová stránka 895 (bratří Kameničtí), dříve u nás velmi oblíbená. IANA ale obsahuje kódovou

stránkou 1250 používanou českými Windows a českou kódovou stránku 852, v níž se často ukládalo v MS-DOSu.

Pokud se neznala informace o kódové stránce textu, vznikaly četné problémy s kompatibilitou. Nicméně dodnes se *extended* ASCII používá, protože má nezanedbatelnou výhodu, že každý znak má délku pouze 1 byte.

Pokud pracujeme v programu se znaky v *extended* ASCII, musíme mít na paměti, že rozšiřující kódová stránka používá hodnoty od 128 do 255 (od 0x80 do 0xFF). Ty se v 8bitovém čísle typu *signed* zobrazí jako hodnoty od -128 do -1. A v jazyce C se typ *char* bere právě jako *signed char* (*signed* je zde výchozí), tedy 8bitové binární číslo *signed*.

Velmi častou chybou bývá přeskočení bílých znaků (*whitespaces*), o hodnotách 0x8 až 0x20 pomocí porovnání se znakem mezera ' ' = 0x20.

Následující program byl přeložený v C++, kde sizeof(char)=1 (1 byte):

```
char * line = " \n\t šípek";
// wrong program for skipping of whitespaces
int i=0; while (line[i]!=0 && line[i] <= ' ') i++;
char c = line[i]; // c='p'
```

Program přeskočil nejen mezery, znak nové řádky '\n' (0xA) a tabulátor '\t' (0x9), ale i znaky s diakritikou, protože v rozšířené ASCII tabulce mají záporné hodnoty.

Kód opravíme třeba použitím *unsigned char* pro *line*. Znaky s diakritikou z rozsahu 0x80 až 0xFF v rozšířeném ASCII se pak převedou na hodnoty 128 až 255, takže jedině bílé znaky budou menší nebo rovné mezeře.

```
unsigned char * line = (unsigned char*)" \n\t šípek";
int i=0; while (line[i]!=0 && line[i] <= ' ') i++;
char c = line[i]; // c='š'
```

Kódování *extended* ASCII se dnes při programování nejčastěji nahrazuje Unicodem, který může mít ve verzi Basic 16bitové znaky. V plné verzi má znaky v rozsahu 0 až 0x10FFFF, jimiž se pokryjí všechny světové jazyky a používané značky. Znaky Unicodu s hodnotou 0 až 0x7f jsou shodné s ASCII.

Pro ukládání textů a webové stránky se dnes zase stále častěji volí UTF8 (*Unicode Transformation Format*) mající maximální kompatibilitu s ASCII, protože ukládá text po bytech a jeho kódy 0 až 0x7f jsou shodné s ASCII. Kódy 0x80 až 0xFF se v UTF8 používají pro indikaci více bytových sekvencí pro vložení znaků Unicodu. Jejich délka může být až 6 bytů, avšak v UTF-8 se dnes často uplatňuje norma RFC 3629 omezující sekvence na 4 byty.

Výše uvedený program pro vynechání bílých znaků (*whitespaces*) se však rozhodně nezjednoduší, použijeme-li znaky v Unicodu, například v jazyce C typ *wchar_t*. Pak nám totiž nestačí testovat jen 6 bílých znaků (v ASCII tabulce menších nebo rovných mezeře), ale správně bychom měli doplnit další testy rozeznávající nejméně 19 nových znaků[15] přidaných Unicodem pro různé typografické mezery a řádkování. Obvykle se však na ně v programech pro zpracování textu zapomíná, buď na všechny, či na jejich část, a tiše☺ se předpokládá, že vstupní text je neobsahuje.

V hardwaru a jednoduchých zobrazovacích jednotkách však ASCII stále zůstává hlavním použitým kódování.

[15] Celkem Unicode přidává 25 bílých znaků, ale 6 z nich se používá málokdy. Kompletní seznam najdete na Wikipedii pod heslem "Whitespace character".

3.7 Kolik je 1000?

Kapitolu 3 jsme zahájili vtipem:

Po autohavárii mi programátor podepsal náhradu škody 1000 Kč.
Zaplatil deset korun s tím, že mi dává dvě koruny navíc.

Kolik je tedy 1000 v různých soustavách? Vše závisí na použitém zobrazení :-)

Zápis 1000 se převede na dekadické číslo

- = **-8** ze 4bitového binárního *signed*,
- = **-0** ze *sign-magnitude*,
- = **8** z binárního *unsigned* či binárního *signed* delšího než 4 bity,
- = **512** z oktalového čísla,
- = **1000**, bereme-li zápis přímo dekadicky,
- = **4096** z hexadecimálního čísla,
- = **na libovolné celé číslo** z aditivního kódu, a to v závislosti na offsetu, který se v něm používá.

3.8 Test znalostí z kapitoly 3

Zkuste odpovědět bez pomůcek na otázky. Výsledky najdete v příloze.

Otázka 1 - doplňte chybějící hodnoty v tabulkách

	8 bitové číslo unsigned		BCD číslo	
Dekadické číslo	Binárně	Hexadecimálně	Binárně	Hexadecimálně
100	0110 0100	64	0001 0000 0000	100
150				
50				
300				

	10 bitové číslo signed		Přímý kód - znaménko hodnota	
Dekadické číslo	Binárně	Hexadecimálně	Binárně	Hexadecimálně
-100	11 1001 1100	39C	10 0110 0100	264
-10				
	11 1111 1110			
		200		
			10 0000 0100	
511				

Otázka 2 - Mějme výraz s operandy zapsanými dekadicky. Doplňte dekadickou hodnotu, kterou bude mít výsledek součtu či rozdílu, budou-li čísla i výsledek uložená v daném formátu

Formát čísla	Délka v bitech	Operace s čísly s dekadickými hodnotami	dá výsledek s dekadickou hodnotu
unsigned	8	100+200=	44
unsigned	10	100+200=	300
unsigned	8	200+200=	
unsigned	9	200+200=	
unsigned	8	127+1=	
signed	8	127+1=	
signed:	8	100-150=	
signed:	12	100-150=	

Otázka 3 - Mějme 8-bitové operandy v binárním tvaru. Napište výsledky operací s nimi.

bitový řetězec	posun vlevo o 1 bit		posun vpravo o 1 bit	
	aritmetický	logický	aritmetický	logický
1000 0001				
1111 1111				
0101 0101				
1010 1010				

Otázka 4 - Jaká bude výsledná hodnota proměnných iresult a cresult programu v jazyce C?

```
char c1 = 'A';
char c2 = 'b';
int iresult = c2 - c1;      //  iresult =.............
char cresult = '0' + 5;     //  cresult =..............
```

4 Příloha

4.1 Abecední seznam zkratek a použitých termínů

Aditivní kód - způsob uložení čísel se znaménkem, viz Kapitola 3.5 na str. 39.

ALU *Arithmetic Logic Unit* - aritmeticko-logická jednotka procesoru je základní komponentou procesoru, která provádí všechny aritmetické a logické operace.

ASCII *American Standard Code for Information Interchange* - základní kódování znaků, viz kapitola 3.6 na str. 42.

BCD *Binary Coded Decimal* - způsob zakódování čísel. Všechna čísla se na něj musí převést před jejich zobrazením v dekadickém tvaru, viz Kapitola 3.5 na str. 39.

biased representation - - anglický název pro aditivní kód, viz kapitola 3.3 na straně 35.

Borrow přenos z nejvyššího řádu binárního čísla směrem dolů, ke kterému dochází při odčítání, popsáno v kapitole 3.1.5 na straně 26. V řadě publikací se však směr přenosu nerozlišuje a pro oba směry přetečení se používá *Carry*,

Carry přenos z nejvyššího řádu binárního čísla, viz kapitola 3.1.5 na straně 26. *Carry* je současně jedním z hlavních příznaků (*flags*), které generují ALU procesorů.

Excess-K anglický název pro aditivní kód, vizkapitola 3.3 na straně 35.

Extended *ASCII* - rozšíření ASCII kódu o národní tabulky, viz Kapitola 3.6.1 na str. 44.

hradlo, resp. logické hradlo (logic gate)- označovalo původně konstrukční prvek. Dnes se ale často používá i pro schematickou značku logické operace, viz Kapitola 2.7 na str. 17.

LSB má význam "*least significant bit*" nebo "*right-most bit*", nejméně významný bit. LSB se používá i pro uspořádání bytů jako "*least significant byte*". Zda se LSB vztahuje k bitu či bytu, musí vyplynout z kontextu, viz Obrázek 17 na straně 22.

MSB má význam "*most significant bit*" nebo "*high-order bit*", tedy nevýznamnější bit. MSB se používá i pro uspořádání bytů jako "*most significant byte*". Zda se MSB vztahuje k bitu či bytu záleží na kontextu popisu, viz Obrázek 4 na straně 7.

offset binary - anglický název pro aditivní kód, viz kapitola 3.3 na straně 35.

overflow termín obecně znamená přetečení, ale v počítačové logice se tento pojem nejčastěji používá pro příznak (*flag*) přetečení u aritmetické operace s binárními čísly *signed*. Příznak *overflow* znamená, že výsledek operace je nesmyslný — jako třeba záporný výsledek sčítání dvou kladných čísel. Dojde k tomu kvůli omezené délce binárního čísla. *Overflow* je současně jedním z hlavních příznaků, které generují ALU všech běžnějších procesorů.

přímý kód způsob uložení čísel se znaménkem, viz kapitola 3.3 na straně 35.

Sign–magnitude - anglický název pro přímý kód, viz kapitola 4.3 na straně 20.

signed (binární číslo) - zkrácený název používaný pro zakódování celých čísel se znaménkem ve dvojkovém doplňku, popsáno v kapitole 3.2 na straně 28.

straight binar*y* - anglický název pro přímý kód, viz kapitola 3.3 na straně 35.

unsigned (binární číslo) - zkrácený název používaný pro zakódování celého nezáporného čísla jako binárního čísla bez znaménka, popsáno v kapitole 3.1 na straně 23.

4.2 Řešení testu z Kapitoly 2

Otázka 1: Doplňte nedokončené pravdivostní tabulky logických funkcí:

x3	x2	x1	AND(x1,x2,x3)	OR(x1,x2,x3)	NAND(x1,x2,x3)	NOR(x1,x2,x3)
0	0	0	**0**	**0**	**1**	**1**
0	0	1	**0**	**1**	**1**	**0**
0	1	0	**0**	**1**	**1**	**0**
0	1	1	**0**	**1**	**1**	**0**
1	0	0	**0**	**1**	**1**	**0**
1	0	1	**0**	**1**	**1**	**0**
1	1	0	**0**	**1**	**1**	**0**
1	1	1	**1**	**1**	**0**	**0**

x2	x1	XOR(x1,x2)	EQU(x1,x2)
1	0	**1**	**0**
1	1	**0**	**1**
0	0	**0**	**1**
0	1	**1**	**0**

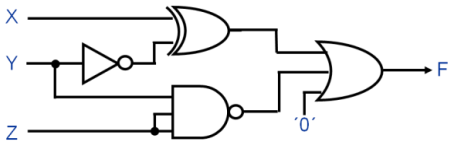
Otázka 2: Přepište tabulku vlevo do Karnaughovy mapy.

x1	x0	y1	y0	X_LESS_Y
0	0	0	0	0
0	0	0	1	1
0	0	1	-	1
0	1	0	-	0
0	1	1	-	1
1	0	0	-	0
1	0	1	0	0
1	0	1	1	1
1	1	-	-	0

X_LESS_Y

y1 \ x1 x0	00	01	11	10
00	0	1	1	0
01	0	0	0	0
11	0	0	1	0
10	0	0	1	0

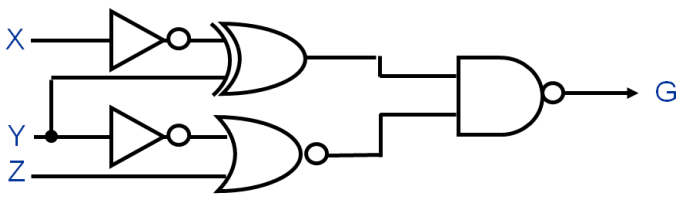
Otázka 3: Napište logický výraz, který je realizovaný logickým schématem:



F(X,Y,Z)= (X xor not Y) or not (Y and Z)

Otázka 4: Nakreslete k logické funkci její logické schéma.

G(X,Y,Z) = not((not X xor Y) and not (not Y or Z))



4.3 Řešení testu z kapitoly 3

Otázka 1 - doplňte chybějící hodnoty v tabulkách

	8 bitové číslo unsigned		BCD číslo	
Dekadické číslo	Binárně	Hexadecimálně	Binárně	Hexadecimálně
100	0110 0100	64	0001 0000 0000	100
150	1001 0110	96	0001 0101 0000	150
50	0011 0010	32	0000 0101 0000	050
300	nelze	nelze	0011 0000 0000	300

	10 bitové číslo signed		Přímý kód - znaménko hodnota	
Dekadické číslo	Binárně	Hexadecimálně	Binárně	Hexadecimálně
-100	11 1001 1100	39C	10 0110 0100	264
-10	11 1111 0110	3F6	10 0000 1010	20A
-2	11 1111 1110	3FE	10 0000 0010	202
-512	10 0000 0000	200	mimo rozsah	mimo rozsah
-4	11 1111 1100	3FC	10 0000 0100	204
511	01 1111 1111	1FF	01 1111 1111	1FF

Otázka 2 - Mějme výraz s operandy zapsanými dekadicky. Doplňte dekadickou hodnotu, kterou bude mít výsledek součtu či rozdílu, budou-li čísla i výsledek uložená v daném formátu

Formát čísla	Délka v bitech	Operace s čísly s dekadickými hodnotami	dá výsledek s dekadickou hodnotu
unsigned	8	100+200=	44
unsigned	10	100+200=	300
unsigned	8	200+200=	144
unsigned	9	200+200=	400
unsigned	8	127+1=	128
signed	8	127+1=	-128
signed:	8	100-150=	nelze, 150 mimo rozsah
signed:	12	100-150=	-50

Otázka 3 - Mějme 8-bitové operandy v binárním tvaru. Napište výsledky operací s nimi

bitový řetězec	posun vlevo o 1 bit		posun vpravo o 1 bit	
	aritmetický	logický	aritmetický	logický
1000 0001	0000 0010	0000 0010	1100 0000	0100 0000
1111 1111	1111 1110	1111 1110	1111 1111	0111 1111
0101 0101	1010 1010	1010 1010	0010 1010	0010 1010
1010 1010	0101 0100	0101 0100	1101 0101	0101 0101

Otázka 4 - Jaká bude výsledná hodnota proměnných iresult a cresult programu v jazyce C?

```
char c1 = 'A';
char c2 = 'b';
int iresult = c2 - c1;      //  iresult = 33 (0x21)
char cresult = '0' + 5;     //  cresult = '5'
```

Autor: Richard Šusta,
http://susta.cz/

Obrázky: * Obrázek 26 - ENIAC Electronic Numerical Integrator and Computer převzatý z Wikipedie,
* ostatní obrázky Richard Šusta

Vydavatel: Katedra řídicí techniky ČVUT-FEL v Praze,
Technická 2, 166 00 Praha 6
http://dce.fel.cvut.cz/

Datum vydání: září, 2016

Rozsah: 50 stran

Domovská stránka dokumentu:

http://dcenet.felk.cvut.cz/edu/fpga/doc/Apolos_V11.pdf