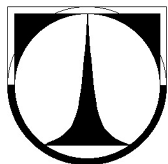


Technická Univerzita v Liberci, Fakulta Mechatroniky

Učební text k předmětu

Číslicové počítače

Poslední oprava 14.10.2010

Obsah

OBSAH......2

ČÍSELNÉ SOUSTAVY......3

DVOJKOVÁ SOUSTAVA......3

Převod z dvojkové soustavy do dekadické......3

Převod z dekadické soustavy do dvojkové......4

HEXADECIMÁLNÍ SOUSTAVA......6

DÉLKA ČÍSLA V RŮZNÝCH SOUSTAVÁCH......7

KÓDOVÁNÍ ZÁPORNÝCH ČÍSEL......7

Dvojkový doplněk......7

Číselná osa......8

STANDARDNÍ DATOVÉ TYPY......9

DATOVÉ TYPY S PEVNOU ŘÁDOVOU TEČKOU......9

Ukládání dat do paměti......10

Datový typ Char......10

ASCII (American Code for Information Interchange) tabulka......10

DATOVÉ TYPY S PLOVOUCÍ ŘÁDOVOU TEČKOU......11

Datový typ Single......11

Rozsah a přesnost......13

Další datové typy pro kódování reálných čísel......15

Číselná soustava s faktoriály......15

Strukturované datové typy......15

PROCESOR......16

STRUČNÝ ÚVOD......16

ČASOVÁ OSA......17

PROCESOR......18

PROGRAMÁTORSKÝ MODEL PROCESORU......19

INSTRUKCE......20

Přesun dat......20

Vstup a výstup......21

Logické operace......21

Rotace......23

Aritmetické operace......23

Skoky......24

Práce se zásobníkem......32

Podprogramy......33

Přehled instrukcí......35

PŘERUŠENÍ......36

POZNÁMKY......37

Zápočet......37

Zkouška......37

Náplň cvičení:......37

Číselné soustavy

Každou hodnotu lze reprezentovat různými způsoby. V historii se používali jako počítadla ruce a proto první číselné soustavy měli základ pět a deset. Dnes se v běžném životě nejvíce používá soustava o základu deset, takzvaná Arabská (vymyšlena ale byla v Indii). Tato soustava používá číslice 0..9, přičemž poloha číslice určuje jeho váhu - tím je možné vyjádřit i čísla větší než deset.

Číslo v desítkové soustavě můžeme obecně rozepsat takto:

$N_{10} = a_n \times 10^n + a_{n-1} \times 10^{n-1} + \ldots + a_1 \times 10^1 + a_0 \times 10^0 + , + a_{-1} \times 10^{-1} + a_{-2} \times 10^{-2} + \ldots a_{-m} \times 10^{-m}$, $a = 0..9$

Příklad:

156,2 = 1 x 100 + 5 x 10 + 6 x 1 + 2 x 0,1

Z historických důvodů lidé používají v některých případech i číselné soustavy o jiném základu, např. čas počítáme v soustavě o základu šedesát (každá hodina má šedesát minut a každá minuta šedesát vteřin).

Počítače mohou z principu pracovat s libovolnou číselnou soustavou. Protože je však nejjednodušší a nejspolehlivější rozlišovat pouze dva stavy nějaké fyzikální veličiny, používá soustava o základu dva. Tyto dva stavy se mohou nazývat různě:

0	–	1
lež	–	pravda
logická 0	–	logická 1

V reálném světě se pracuje nejčastěji s napětím. Např. log. 0 = 0V, log. 1 = 5V. Buňce, která nese takovou informaci, říkáme jeden bit.

Protože zápis čísel v soustavě o základu dva je pro člověka nepřehledný, používají se ve výpočetní technice ještě soustavy o základu osm a šestnáct. Mezi těmito soustavami a dvojkovou soustavou lze totiž snadno převádět a pro člověka jsou čísla v těchto soustavách mnohem „představitelnější“.

Pro některé speciální případy lze používat i jiné typy číselných systémů. Např. pomocí číselné soustavy s faktoriály, kde každá pozice má váhu n!, lze velmi přesně pracovat s racionálními čísly (viz dále).

Dvojková soustava

Převod z dvojkové soustavy do dekadické

Obecně vypadá číslo ve dvojkové soustavě takto:

$N_2 = a_n \times 2^n + a_{n-1} \times 2^{n-1} + \ldots + a_1 \times 2^1 + a_0 \times 2^0 + , + a_{-1} \times 2^{-1} + a_{-2} \times 2^{-2} + \ldots a_{-m} \times 2^{-m}$, $a = 0..1$

Příklad:

$101101,01_2$ = 1 x 32 + 0 x 16 + 1 x 8 + 1 x 4 + 0 x 2 + 1 x 1 + 0 x 0,5 + 1 x 0,25

Z tohoto zápisu již lze číslo přímo přepsat do desítkové soustavy, stačí jen vynásobit a sečíst:

$101101,01_2$ = 45,25

Tento postup je velmi rychlý a lze ho provádět pro malý počet míst i přímo z hlavy. Stačí si pamatovat mocniny dvojky (= váhy pozic).

Převod z dekadické soustavy do dvojkové

Matematické odvození postupu:
Napíšeme si rovnici, kde na levé straně je zadaná hodnota v desítkové soustavě a na pravé straně je obecně rozepsané číslo ve dvojkové soustavě.

Příklad:
$132 = \ldots + a_4 \times 2^4 + a_3 \times 2^3 + a_2 \times 2^2 + a_1 \times 2^1 + a_0 \times 2^0$

Toto je jedna rovnice o mnoha neznámých. Abychom jí mohli vyřešit, musíme jí rozdělit na rovnice pro jednotlivé proměnné. To provedeme tak, že celou rovnici vydělíme číslem 2:

$132 = \ldots + a_4 \times 2^4 + a_3 \times 2^3 + a_2 \times 2^2 + a_1 \times 2^1 + a_0 \times 2^0 \quad : 2$
$66{,}0 = \ldots + a_4 \times 2^3 + a_3 \times 2^2 + a_2 \times 2^1 + a_1 \times 2^0 + a_0 \times 2^{-1}$

Výsledek poté rozdělíme na celou část a část za desetinou tečkou.

Desetinná část:
$0 = a_0 \times 2^{-1} = a_0 \times 0{,}5$
V tomto okamžiku již můžeme přímo napsat, že koeficient $a_0 = 0$.

Celá část:
$66 = \ldots + a_4 \times 2^3 + a_3 \times 2^2 + a_2 \times 2^1 + a_1 \times 2^0$

Se zbytkem rovnice opakujeme tento postup až do doby, kdy na levé straně zůstane 0. V tomto okamžiku máme vypočteny všechny koeficienty.

$66 = \ldots + a_4 \times 2^3 + a_3 \times 2^2 + a_2 \times 2^1 + a_1 \times 2^0 \quad : 2$
$33{,}0 = \ldots + a_4 \times 2^2 + a_3 \times 2^1 + a_2 \times 2^0 + a_1 \times 2^{-1}$
$0 = a_1 \times 2^{-1} \quad \Rightarrow \quad a_1 = 0$

$33{,}0 = \ldots + a_4 \times 2^2 + a_3 \times 2^1 + a_2 \times 2^0 \quad : 2$
$16{,}5 = \ldots + a_4 \times 2^1 + a_3 \times 2^0 + a_2 \times 2^{-1}$
$0{,}5 = a_2 \times 2^{-1} = a_2 \times 0{,}5 \quad \Rightarrow \quad a_2 = 1$

$16 = \ldots + a_4 \times 2^1 + a_3 \times 2^0 \quad : 2$
$8{,}0 = \ldots + a_4 \times 2^0 + a_3 \times 2^{-1}$
$0 = a_3 \times 2^{-1} \quad \Rightarrow \quad a_3 = 0$
…

Výsledek získáme po posledním kroku výpočtu opsáním spočtených koeficientů: $132 = 10000100_2$

Tento postup je poněkud zdlouhavý, proto se používá jeho rychlejší varianta. Dělíme desítkové číslo celočíselně dvěma a opisujeme zbytky.
Příklad:

132 : 2 =	66 : 2 =	33 : 2 =	16 : 2 =	8 : 2 =	4 : 2 =	2 : 2 =	1 : 2 = 0
zbytek: 0	0	1	0	0	0	0	1
a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7

Dělíme do té doby, dokud výsledek není nula a zbytky opisujeme křížem (od a_7 do a_0). Výsledek je tedy:

$132 = 10000100_2$

Desetinou část čísla převádíme obdobně. Pouze používáme pro řešení rovnice místo dělení násobení.

Příklad:

$0{,}22 = a_{-1} \times 2^{-1} + a_{-2} \times 2^{-2} + a_{-3} \times 2^{-3} + a_{-4} \times 2^{-4} + a_{-5} \times 2^{-5} + \dots \quad \times 2$
$0{,}44 = a_{-1} \times 2^{0} + a_{-2} \times 2^{-1} + a_{-3} \times 2^{-2} + a_{-4} \times 2^{-3} + a_{-5} \times 2^{-4} + \dots$
$0 = a_{-1} \times 2^{0} \quad \Rightarrow \quad a_{-1} = 0$

$0{,}44 = a_{-2} \times 2^{-1} + a_{-3} \times 2^{-2} + a_{-4} \times 2^{-3} + a_{-5} \times 2^{-4} + \dots \quad \times 2$
$0{,}88 = a_{-2} \times 2^{0} + a_{-3} \times 2^{-1} + a_{-4} \times 2^{-2} + a_{-5} \times 2^{-3} + \dots$
$0 = a_{-2} \times 2^{0} \quad \Rightarrow \quad a_{-2} = 0$
$0{,}88 = a_{-3} \times 2^{-1} + a_{-4} \times 2^{-2} + a_{-5} \times 2^{-3} + \dots \quad \times 2$
$1{,}76 = a_{-3} \times 2^{0} + a_{-4} \times 2^{-1} + a_{-5} \times 2^{-2} + \dots$
$1 = a_{-3} \times 2^{0} \quad \Rightarrow \quad a_{-3} = 1$

…

Stejně jako u převodu celé části lze postup zrychlit.

Příklad:

0,25 x 2 = 0,5 x 2 = 1 (-1) 0 x 2 = 0
0 1
a_{-1} a_{-2}

Jako výsledek opisujeme celé části výsledku po násobení dvěma. V našem případě je tedy výsledek po zaokrouhlení:

$0{,}25 = 0{,}01_2$

Rozvoj necelé části může být i neukončený (periodický nebo neperiodický). V tomto případě musíme rozvoj zkrátit (zaokrouhlit) a dochází tak k zaokrouhlovací chybě.

Příklad:

0,22 x 2 = 0,44 x 2 = 0,88 x 2 = 1,76 (-1) 0,76 x 2 = 1,52 (-1) 0,52 x 2 = 1,04 (-1) 0,04 x 2 = 0,08 …
0 0 1 1 1 0

$0{,}22 = 0{,}00111000_2$

S čísly ve dvojkové soustavě lze přímo počítat obdobně, jako s čísly v desítkové soustavě. Např. zkusíme sečíst dvě čísla:

Příklad:
Sečtěte dvě čísla 100. Pro výpočet použijte dvojkovou soustavu.
100 = $1\,1\,0\,0\,1\,0\,0_2$

$1\,1\,0\,0\,1\,0\,0_2$
\+ $1\,1\,0\,0\,1\,0\,0_2$
1 1 1 - přenos do vyššího řádu

$1\,1\,0\,0\,1\,0\,0\,0_2 = 128 + 64 + 8 = 200$

Hexadecimální soustava

V této soustavě může každá číslice nabývat šestnácti různých hodnot. Pro reprezentaci hodnot 0..9 používá tato soustava číslice, pro hodnoty 10..15 se používají znaky A až F. Převod z a do desítkové soustavy se provádí stejným postupem, jako u dvojkové soustavy.

Příklad:
Převeďte číslo $2B_{16}$ do desítkové soustavy.

Řešení:
Stejně jako u dvojkové soustavy provedeme vynásobení každé číslice její váhou a sečteme výsledky. Váhy jednotlivých míst jsou mocniny šestnácti.

$2B_{16} = 2 \times 16^1 + B \times 16^0 = 2 \times 16 + 11 \times 1 = 43$

Příklad:
Převeďte číslo 82 do hexadecimální soustavy.

Řešení:
Stejně jako u dvojkové soustavy budeme provádět dělení a budeme opisovat zbytky. Protože základ soustavy je šestnáct, dělit budeme tentokrát šestnácti.

82 : 16 = 5 : 16 = 0
Zbytek: 2 5

Výsledek: $82 = 52_{16}$

Hlavním důvodem pro používání této soustavy je snadný převod z a do dvojkové soustavy. Každou číslici šestnáctkového čísla můžeme totiž nahradit čtyřbitovým binárním vyjádřením a máme okamžitě požadované binární číslo. Podobně lze postupovat při opačném převodu. Nejprve si rozdělíme binární číslo na čtveřice bitů (v případě potřeby doplníme nulami zleva) a poté každou čtveřici převedeme do šestnáctkové soustavy.

Situaci si ještě můžeme zjednodušit použitím tabulky:

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

Příklad:
Převeďte číslo 52_{16} do binární soustavy.

Řešení:
Rozdělíme si číslo na jednotlivé číslice a každou jednotlivě podle tabulky nahradíme jejím binárním čtyřbitovým vyjádřením.

$5_{16} = 0101_2$, $2_{16} = 0010_2$
$52_{16} = 01010010_2$

Příklad:
Převeďte číslo 10101_2 do hexadecimální soustavy.

Řešení:
Binárně vyjádřené číslo doplníme nevýznamnými nulami zleva tak, aby počet míst byl dělitelný čtyřmi. Poté rozdělíme toto číslo na čtveřice bitů a každou čtveřici převedeme jednotlivě podle tabulky na její hexadecimální reprezentaci.

$10101_2 = 0001\ 0101_2 = 15_{16}$

Délka čísla v různých soustavách

V určitých případech potřebujeme předem vědět, kolik platných míst (N) budeme potřebovat k vyjádření nějaké hodnoty (X) v různých číselných soustavách. Výpočet provedeme pomocí logaritmu o základu, který odpovídá základu soustavy:

Pro dekadickou soustavu: $N = [\log_{10} X] + 1$
Pro binární soustavu: $N = [\log_2 X] + 1$
Pro hexadecimální soustavu: $N = [\log_{16} X] + 1$

Výsledek zaokrouhlujeme na celé číslo směrem dolů.

Kódování záporných čísel

Pro zápis záporných čísel ve dvojkové soustavě můžeme použít znaménko mínus stejně, jako v desítkové soustavě. Problém ovšem nastává v okamžiku, kdy budeme chtít záporné číslo vložit do počítače. Ten totiž pracuje ve dvojkové soustavě a mínus musíme vyjádřit pomocí jedniček a nul.

Pro kódování znaménka v počítačích se proto používají tyto metody:

- Vyhrazení jednoho bitu pro znaménko, další bity zůstávají pro binární váhový kód absolutní hodnoty (např. mantisa u typů s pohyblivou řádovou čárkou)
- Přičtení konstanty (např. exponent u typů s pohyblivou řádovou čárkou)
- Pomocí dvojkového doplňku (např. celočíselné typy ve vyšších programovacích jazycích)

Dvojkový doplněk

Doplňkem se rozumí rozdíl kapacity soustavy (tj. 2^n, kde n je počet bitů v binárním vyjádření) a absolutní hodnoty čísla. Rozsah čísel, které tímto způsobem lze vyjádřit, je pak $(- 2^{n-1})$ až $(2^{n-1} – 1)$. Např. pomocí osmi bitů tak lze vyjádřit čísla v rozsahu –128 .. +127 .
Interpretace záporných čísel pomocí dvojkového doplňku umožňuje, na rozdíl od ostatních způsobů, přímo provádět sčítání a odečítání. Zároveň lze také přímo zjistit znaménko, protože nejvyšší bit u záporných čísel je vždy jedna.

Postup pro převod absolutní hodnoty na záporné číslo pomocí dvojkového doplňku je následující:

- doplníme dvojkové vyjádření absolutní hodnoty zleva nulami na požadovaný počet bitů
- provedeme negaci všech bitů
- k výsledné hodnotě přičteme binárně hodnotu jedna

Alternativní postup:

- doplníme dvojkové vyjádření absolutní hodnoty zleva nulami na požadovaný počet bitů
- zprava opíšeme všechny nuly až k první jedničce (včetně), další bity negujeme

Příklad:
Převeďte číslo –46 do dvojkové soustavy jako osmibitové číslo pomocí dvojkového doplňku.

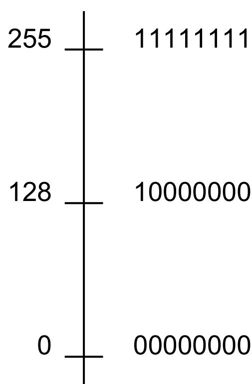
$46 = 00101110_2$ Nejprve převedeme desítkovou absolutní hodnotu a doplníme nulami zleva na osm bitů
11010001_2 Znegujeme výsledek
11010010_2 Přičteme k výsledku jedničku

Výsledek: $-46 = 11010010_2$

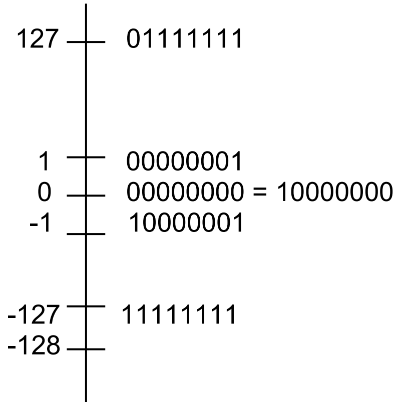
Číselná osa

Každý z uvedených způsobů má vliv na rozmístění čísel na číselné ose. Například pro čísla kódovaná pomocí osmibitových proměnných vypadá takto:

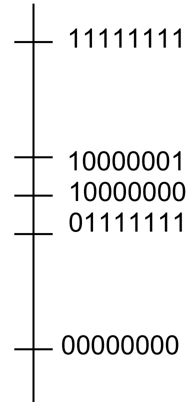
Celá kladná čísla



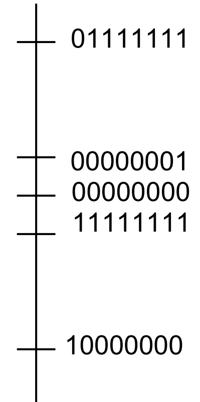
Vyhrazení jednoho bitu pro znaménko



Přičtení konstanty



Dvojkový doplněk

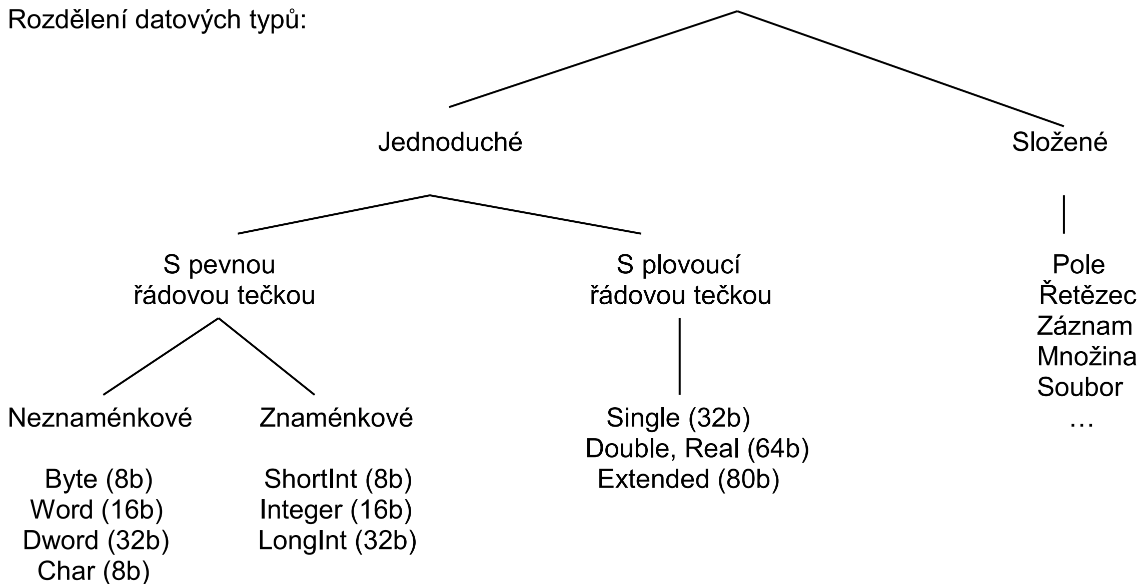


Standardní datové typy

Počítače používají pro vnitřní reprezentaci dat dvojkovou soustavu. To ale ještě není úplná informace pro představu o vnitřním fungování počítače.
Pro zpracování dat v procesoru slouží tzv. ALU (aritmeticko-logická jednotka). Tato jednotka používá pro vstup i výstup dat sběrnici s pevnou šířkou (s pevným počtem bitů). Zpracovávaná data, se kterými pracuje, jsou uložena v paměti, která musí mít odpovídající velikost jedné buňky. To se týká i ostatních součástí procesoru. Proto je šířka datová sběrnice velmi důležitá.
Obvykle je šířka datové sběrnice 2^n. První procesor firmy Intel byl čtyřbitový, první osobní počítače používali osmibitový procesor a dnešní nejvýkonnější osobní počítače mají datovou sběrnici 64 bitů.

Aby bylo možné počítače snadno programovat, museli vzniknout nějaké standardní proměnné, které by byly ve všech programovacích jazycích stejné. U prvních osobních počítačů se přirozeně začaly používat osmibitové proměnné a ve strojovém kódu procesoru obvykle lze provádět matematické operace pouze s nimi. Pro osmibitovou proměnnou se vžil název byte (bajt).
Ve vyšších programovacích jazycích bylo třeba zbavit se této bariéry. Proto existuje mnoho různých typů proměnných pro různé účely. Vzhledem k efektivitě výsledného programu však všechny vycházejí ze základní šířky slova daného procesoru. Na úrovni strojového kódu se totiž implementují jen jako jednoduché matematické operace s daty paměti.

Rozdělení datových typů:



Datové typy s pevnou řádovou tečkou

Jsou to nejjednodušší celočíselné typy, pomocí kterých lze vyjadřovat celá čísla. Můžeme je dále rozdělit na znaménkové a neznaménkové. Rozsah hodnot, které pomocí nich lze vyjádřit je dán jejich velikostí v pamětí (počtem bitů).

Typ	Rozsah
Byte	0 .. 255
Word	0 .. 65535
Dword	0 .. $2^{32}-1$
ShortInt	-128 .. 127
Integer	-32768 .. 32767
LongInt	-2^{32} .. $2^{32}-1$
Char	0..255 – odkaz do ASCII tabulky

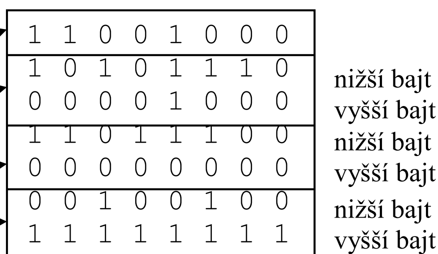
Ukládání dat do paměti

Pro zjednodušení budeme o paměti uvažovat jako tabulce, která má na každém řádku osmibitové číslo. Adresa 0 je nahoře a každý další řádek má adresu o 1 vyšší.
Překladače vyšších programovacích jazyků ukládají proměnně postupně do paměti v pořadí, jak jsou deklarovány.
Proměnné, které mají velikost jeden bajt (Byte, Char) je situace jednoduchá. Vícebajtové proměnné se ukládají do paměti postupně od nejméně významného bajtu.

Příklad:

```
Program Pamet;
Var
  B : Byte;
  I,J : Integer;
  W : Word;

Begin
  B := 200;
 W := 2222;
  I  := 220;
  J := -220;
End.
```



Datový typ Char

V podstatě jde o celé číslo, které slouží jako odkaz do ASCII tabulky znaků. Hodnota této proměnné se interpretuje jako znak.
ASCII tabulka vznikla v šedesátých letech pro přenos textu na velké vzdálenosti. Původně byla používána jako sedmibitová, proto mají všechny důležité znaky hodnotu menší, než 128. Znaky s hodnotou nad 128 jsou závislé na aktuální znakové sadě = problémy se zobrazováním znaků v jiné sadě.
Pro zvýšení kompatibility se začíná používat kódování UNICODE. Znaky jsou kódovány šestnáctibitové, a proto se do tabulky znaků vejdou všechny znakové sady.

ASCII (American Code for Information Interchange) tabulka

Řídící znaky:

9 = Tabulátor
10 = Line Feed (kurzor na další řádek)
13 = Carriage Return (kurzor na začátek řádku)
27 = Escape

32 = Mezera

Čísla:	Velká písmena:	Malá písmena:
48 = „0“	65 = „A“	97 = „a“
…	…	…
57 = „9“	90 = „Z“	122 = „z“

Datové typy s plovoucí řádovou tečkou

Abychom mohli pracovat s reálnými čísly ve dvojkové soustavě, můžeme využít stejný přístup, jako v desítkové. To znamená, že si číslo převedeme do dvojkové soustavy a případně ještě do normalizovaného tvaru.

Příklad:
Převeďte číslo $2{,}345 \times 10^{2}$ do dvojkové soustavy v normalizovaném tvaru.

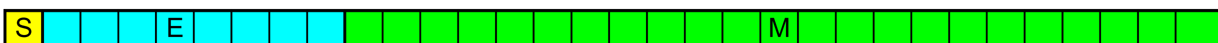
Řešení:
Nejprve si převedeme desítkové číslo z normalizovaného tvaru do tvaru, který je vhodný pro převod do dvojkové soustavy (to je tvar bez exponentu). Provedeme převod celé i desetinné části a nakonec přepíšeme výsledek do normalizovaného tvaru ve dvojkové soustavě (přidáme exponent tak, aby před desetinou čárkou bylo pouze číslo 1).

$2{,}1234 \times 10^{3} = 2123{,}4 = 100001001011{,}011001100_2 = 1{,}00001001011011001100_2 \times 2^{11}$

Pokud už máme reálné číslo ve dvojkové soustavě v normalizovaném tvaru, můžeme se zabývat problémem jak ho uložit do paměti počítače. Ve skutečnosti totiž musíme uložit tři různé informace – znaménko, absolutní hodnotu a exponent. Musíme si zvolit způsob kódování znaménka u exponentu, počet bitů u exponentu a počet bitů na vlastní číslo (mantisu). Tím určíme rozsah a přesnost čísel, která můžeme takto kódovat.
Je zřejmé, že možností je velmi mnoho. Ve vyšších programovacích jazycích se používají hlavně tyto tři datové typy: Single, Double (někdy se označuje jako Real), Extended.

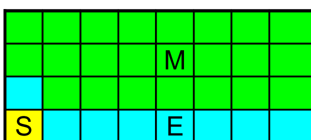
Datový typ Single

Pro zápis reálného čísla v tomto datovém typu se využívá celkem 32 bitů. Jeden bit je znaménko, osm bitů je vyhrazeno pro exponent a 23 bitů pro mantisu. Mantisa je absolutní hodnota reálného čísla v normalizovaném tvaru, ale zapisuje se bez první jedničky. V normalizovaném tvaru totiž binární číslo vždy začíná jedničkou, pak je desetinná čárka a pak následuje zbytek absolutní hodnoty. Proto je zbytečné první jedničku před desetinou čárkou zapisovat.



S – znaménko, 1=mínus
E – exponent, kóduje se přičtením konstanty 127, takže rozsah exponentů je –127 až +128
M – mantisa, absolutní hodnota čísla, zapisuje se bez první jedničky

Tato získané bitové pole poté rozdělíme po osmi bitech a ukládáme tyto osmice bitů do paměti postupně zprava doleva (pořadí bitů v osmicích se zachovává).



Zpětně pak dostaneme reálné číslo dosazením do vzorce:

$(-1)^{S} \times 2^{E-127} \times (1{,}M)$

Postup pro vytvoření 32-bitové konstrukce:

1. Vyjádřit absolutní hodnotu daného čísla *X* v binární soustavě (odděleně zjistit binární vyjádření celé a necelé části, potom obě části oddělit řádovou tečkou)

2. Řádovou tečku posunout za první jedničku zleva. Z počtu pozic, o které se tečka v zápisu posouvá, určit hodnotu EXPONENTU:
   Žádný posun *e* = 0
   Posun vpravo *e* <0
   Posun vlevo *e* > 0

3. Určit obsah pole S:
   X >=0 S=0
   X < 0 S=1

4. Určit obsah pole E: Vyjádřit hodnotu *e+127* binárním váhovým kódováním (jako typ Byte)

5. Určit obsah pole M: bity, které zůstaly po posunutí řádové tečky vpravo od ní.

Příklad:
Uložte do paměti číslo –0,0625 ve formátu datového typu Single.

Řešení:
Nejprve převedeme absolutní hodnotu do dvojkové soustavy.
$0{,}0625 = 0{,}0001_2$

Poté přepíšeme výsledek do normalizovaného tvaru posunem desetinné čárky.
$0{,}0001_2 = 1{,}0_2 \times 2^{-4}$

Obsah pole S je 1, protože číslo je záporné.
Obsah pole E je binárně vyjádřený exponent, ke kterému přičteme konstantu 127.
$E = -4 + 127 = 124 = 01111011_2$
Mantisu získáme opsáním absolutní hodnoty bez první jedničky, takže to jsou v tomto případě samé nuly.

1	0	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

V paměti pak bude číslo uloženo takto:

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	1	0	1

Rekonstrukce číselné hodnoty:

1. 32-bitovou strukturu rozdělit na pole S (1 bit), E (8), M (23)

2. Sestavit zápis . **1.** ...., ve kterém za řádovou tečkou jsou bity z pole M.

3. Číselně interpretovat obsah pole E (binární váhový kód). Zmenšením získané hodnoty o 127 určit hodnotu *e*. V zápisu posunout řádovou tečku o *e* pozic. (Pro *e*>0 se tečka posouvá směrem vlevo).

4. Zápis v binární soustavě převést do dekadické soustavy (binární váhový kód)

5. Doplnit znaménko podle bitu v poli S. (S=1 . číslo je záporné)

Příklad:
V paměti je uloženo číslo ve formátu datového typu Single:

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	1	0	0	1	0	0	1
0	1	0	0	0	0	0	1

Vyjádřete toto číslo dekadicky.

Řešení:

S = 0
E = 10000010_2 = 129
M = 1001001_2

Výsledek je pak:

$2^{130 - 127} \times 1,1001001_2 = 2^{3} \times 1,1001001_2 = 1100,1001_2 = 12,5625$

Z těchto pravidel pro datový typ Single existuje několik vyjímek. Pokud bychom např. chtěli vyjádřit v tomto typu nulu, museli bychom použít nejmenší vyjádřitelné číslo. To spočteme jednoduše:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

S = 0
E = 0
M = 0

$2^{0 - 127} \times 1,0_2 = 2^{-127} \times 1,0_2 = 5,877 \times 10^{-39}$

Takto to ale ve skutečnosti není !!!

Pokud bychom např. sčítali samé nuly, vyšlo by nám po nějaké době i poměrně velké číslo. Stejně tak násobení nulou by výsledek nebyl nula!

Proto se zavádí několik vyjímek, které umožní zvýšit rozsah a přesnost. Tyto vyjímky jsou:

1. Pokud S = 0, E = 0 a M = 0 : hodnota čísla je nula
2. Pokud E = -127 : EXP se bere jako –126 a absolutní hodnota se nebere ve tvaru (1,M) ale (0,M)
3. Pokud E = 128 : hodnota čísla je nekonečno, M musí být 0. Jiné hodnoty M jsou neplatné.

Rozsah a přesnost

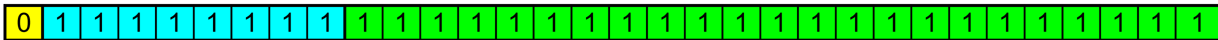
Pro určení rozsahu si spočteme nejmenší a největší zobrazitelné číslo.

Příklad – nejmenší číslo:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

$2^{0 - 127} \times 0,00000000000000000000001_2 = 2^{-127} \times 2^{-23} \times 1_2 = 1,4 \times 10^{-45}$

Příklad – největší číslo:



$2^{255-127} \times 1,11111111111111111111111_2 = 2^{127} \times 2 = 3,4 \times 10^{38}$

Tímto jsme si zjistili rozsah datového typu Single, který je od $1,4 \times 10^{-45}$ do $3,4 \times 10^{38}$ a to jak na kladné tak na záporné části číselné osy, přičemž nula je vyjádřená speciální kombinací.

Pokud nás bude zajímat přesnost, spočteme si rozdíl mezi dvěma za sebou následujícími čísly. Ten je daný váhou nejméně významného bitu v mantise, tj.:
$2 \times 2^{-23} = 1,2 \times 10^{-7}$
Z tohoto můžeme říct, že přesnost takto zakódovaných reálných čísel je zhruba 6 až 7 desetinných míst v desítkové soustavě.

Na první pohled se může zdát, že je to dostatečná přesnost. Je ale nutné si uvědomit, že co je dostatečná přesnost pro jednu aplikace, nemusí v jiné aplikaci stačit. Pokud se nějaký výpočet provádí ve větším množství iteračních kroků, může i při relativně velké přesnosti výpočtu nakonec vyjít nesmyslný výsledek.

Příklad:
Napište program v jazyce Paskal, který bude počítat výraz S = S x 11 – 1. Počáteční hodnota S je 0,1 . Proveďte deset kroků tohoto výpočtu.

Řešení:

```
program Presnost;
 var
  S : Single;
  I : Integer;

begin
  S := 0.1;
  For I := 0 To 10 do
    Begin
      WriteLn(I,' - ',S);
      S := S * 11 - 1;
    End;
end.
```

Výsledek každého kroku by měl při dané počáteční hodnotě být opět 0,1 . Když totiž 0,1 vynásobíme jedenácti, je výsledek 1,1. Po odečtení jedničky by měl být výsledek 0,1 .

Když však tento program spustíme, napíše nám na obrazovku zhruba toto:

```
0 - 0.100000001490116
1 - 0.100000016391277
…
7 - 0.129038155078888
8 - 0.419419705867767
9 - 3.6136167049408
10 - 38.7497825622559
```

Zde je vidět, že i malá chyba výpočtu, způsobená použitým kódováním reálných čísel, může i po poměrně málo krocích vést k naprosto nesmyslnému výsledku.

Další datové typy pro kódování reálných čísel

Aby bylo možno pracovat i s větší přesností reálných čísel, jsou definovány ještě další dva datové typy a to typ Double a Extended. Pracuje se s nimi naprosto stejným způsobem jako s typem Single. Liší se pouze velikostí mantisy a exponentu a tím samozřejmě i rozsahem a přesností.

	Exponent	*Mantisa*	*Velikost*	*Minimální hodnota*	*Maximální hodnota*	*Přesnost*
Single	8 bitů	23 bitů	32 bitů	1,4E-45	3,4E38	6–7 míst
Double (Real)	11 bitů	52 bitů	64 bitů	5,0E-324	1,7E308	15-16 míst
Extended	15 bitů	63 bitů	80 bitů	3,4E-4932	1,1E4932	18-19 míst

Číselná soustava s faktoriály

I když použití většího datového typu (Double nebo Extended) náš předchozí příklad neřeší (chyba nastane jen po větším počtu kroků), v praxi si těmito typy většinou vystačíme.
Pokud by nám ale tyto datové typy nestačily, musíme použít nějaký úplně jiný způsob reprezentace reálných čísel. Takovým způsobem může být například číselná soustava na bázi faktoriálů.

Celá část čísla: i -tá pozice má váhu i! a_i<=|i|
Necelá část čísla: (-i) -tá pozice má váhu 1/i! a_{-i}<=|i|

Např. zápis čísla **4221.002** představuje hodnotu

4*4!+2*3!+2*2!+1*1! + 2/3! = 4*24+2*6+2*2+1*1 + 2/6 = 123.333

Tato soustava má schopnost přesně pojmout všechna racionální čísla
Používá se však jen ve speciálních případech, z důvodu pomalé implementace.

Strukturované datové typy

V praxi potřebujeme často pracovat s mnoha proměnnými najednou. Proto se mimo jednoduchých datových typů definují i typy strukturované (složené). Jejich základním stavebním kamenem jsou typy jednoduché.

Nejvýznamnější strukturované typy:

- Pole – homogenní struktura (všechny prvky pole jsou stejného typu), jednotlivé prvky pole se do paměti ukládají za sebe, můžeme se na ně odkazovat přes indexy. Pole mohou být i vícerozměrná (pole polí).
- Záznam – každá položka záznamu může mít jiný typ, přístup k položkám je přes jejich identifikátory
- Řetězec – je to vlastně pole znaků, má některé speciální vlastnosti, vhodné pro zpracování textů.

Procesor

Stručný úvod

Když se v roce 1801 objevil první tkalcovský stroj programovatelný pomocí děrných štítků, nikdo si asi v té době nedokázal představit, kam se během následujících dvou století posunou možnosti programovatelných strojů. Asi nejmasovějším nasazením strojů na děrné štítky bylo v roce 1980 sčítání lidu v USA. Ani po 80ti letech vývoje však nelze mluvit o tom, že by byly tyto stroje univerzálně programovatelné. Pomocí štítků se jen vkládala data, algoritmus se neměnil. Na první opravdu programovatelný počítač bylo nutné počkat do začátku druhé světové války.
Až do roku 1940 se většina simulací a jednoduchých výpočtů prováděla na analogových počítačích. Ty sice umožňovali poměrně přesné simulace různých dějů, ale jejich „programování“ se dělo změnou zapojení, takže nebyly příliš pružné. O nějakém hromadném zpracování dat nemohla být řeč.
První programově řízený mechanický počítač vytvořil v roce 1938 německý matematik Konrád Zuse a nazval ho Z1. Když roce 1941 vytvářel již třetí generaci svého počítacího stroje (nazval jej překvapivě Z3), programoval jej stále pomocí děrných štítků. Mechanické prvky však již nahradil pomocí relé. Tento počítač dokázal provést násobení za 3-5 sekund a používal se za druhé světové války pro výpočty drah německých raket V1 a V2.
Ani další válečné velmoci však nezaháleli, a tak vznikl v USA v roce 1942 první elektronický (skládal se z 18ti tisíc elektronek) počítač Eniac pro výpočty balistických tabulek a ve Velké Británii v roce 1943 matematik Alan Turing vytvořil počítač Colossus na luštění německých šifer.
Všechny tyto počítače, které vznikly v době války, měli některé společné vlastnosti. Nebyli příliš spolehlivé (např. u počítače Eniac se uvádí střední doba mezi poruchami elektronek kolem sedmi minut), programovali se poměrně složitě pomocí děrných štítků nebo přepínačů, byly to obrovské stroje (např. Eniac zabíral celý blok domů), na dnešní poměry měli zanedbatelný výkon a jejich existence byla poměrně dlouho tajena (např. Colossus byl odtajněn až v roce 1970). Přesto významně přispěli k poválečnému rozvoji počítačů.
Dalším velkým pokrokem byl v roce 1948 počítač Manchester Mark I, který místo mechanické paměti používal paměťovou obrazovku. Navíc jako první používal převratné myšlenky amerického matematika maďarského původu, Johna Von Neumanna (narozen jako János Neumann). Ten v roce 1945 zveřejnil dva základní koncepty, kterými se počítače řídí dodnes. První z nich byl, že počítače by měli být jednoduché stroje a bez nutnosti „předrátovávání“ pro každou úlohu. Jejich sada instrukcí by měla umožňovat plně ovládat jejich jednoduchý hardware a rychle ho přeprogramovat. Druhý koncept byl důležitý pro vývoj vyšších programovacích jazyků. Von Neumann ho nazval Conditional control transfer (podmínkami řízený přesun). V tomto konceptu pokládá základy vyšších programovacích jazyků. V první části mluví o malých podprogramech, mezi kterými lze libovolně skákat, místo jejich postupného vykonávání odshora dolů. Ve druhé části mluví o skocích, založených na logických příkazech jako je IF [výraz] THEN a smyčkách jako je např. příkaz FOR. Také se zmiňuje o knihovnách s bloky kódu, které lze opakovaně používat. To byli v té době velmi revoluční myšlenky.
Zejména první z těchto konceptů byl důležitý pro vytvoření prvního skutečného procesoru. Dřívější stroje byli totiž specializované svým zapojením. Jejich funkce se měnila jen velmi obtížně. Na rozdíl od toho je procesor je univerzální součástka, vyráběná hromadně. Specializace se prování pomocí zadaného programu, který procesor vykonává. To umožňuje vytvoření opravdu univerzálních programovatelných strojů (počítačů).
Poměrně dlouho však byl přístup k programovatelným počítačům (vzhledem k jejich vysoké ceně a obrovským rozměrům) výsadou hrstky vyvolených. To se ale změnilo v 80tých letech.
V roce 1971 uvedla na trh firma Intel na trh první sériově vyráběný procesor jako součástku. Byl to čtyřbitový procesor 4004, určený pro kalkulačky. Pro jeho univerzálnost se rozšířil do jiných oborů, např. dodnes řídí vesmírnou sondu Pioneer 10. Jen pro představu - tento procesor měl stejný výpočetní výkon jako Eniac.
V roce 1974 uvedla firma Intel na trh osmibitový procesor 8080. Tento procesor se stal velmi oblíbeným zejména proto, že v roce 1975 vyšla v časopise Popular Elektronics stavebnice osobního počítače Altair. Tento počítač se stal zároveň prvním, pro který psala sofware firma Microsoft, konkrétně se jednalo o překladač jazyka Basic.
Ve stejné době začali podobné procesory vyrábět i další firmy, např. Motorola nebo Sinclair a začali se objevovat sériově vyráběné osobní počítače. Tyto počítače neměli žádný operační systém a pro jejich programování se většinou používal interpret programovacího jazyka Basic, který byl umístěn přímo z výroby v paměti ROM.
V roce 1981 představila firma IBM počítač IBM PC s operačním systémem MS-DOS. Tato platforma se začala masově šířit hlavně díky tomu, že firma IBM otevřela její specifikace i pro ostatní výrobce hardware.

Časová osa

1600 kolem roku 1600 vznikaly v Číně první mechanické stroje které mohli sčítat a odčítat
 ?   v Evropě se objevil flašinet „programovatelný“ pomocí děrných štítků
1801 první tkalcovský stroj, programovatelný pomocí děrných štítků
1890 sčítání lidu v USA pomocí děrných štítků, Herman Hollerit, později založil IBM
    před rokem 1940 se pro simulace používali zejména analogové počítače
1938 první mechanický počítač Z1, Konrád Zuse, německo
1940 Z2, mechanická paměť, pro výpočty používal relé
1941 Z3, děrná páska, 2600 relé, násobení čísel trvalo 3-5 sekund, používal se pro výpočty drah raket V
1943 první elektronický počítač Eniac, pracoval pro balistickou laboratoř americké armády, používal desítkovou soustava, 18000 elektronek, 160KW,chlazení dvěma leteckými motory, 5000 operací za sekundu, data z děrných štítků, program se zadával přepínači
1943 počítač Colossus, Alan Turing, Velká Británie, používal se pro luštění německých šifer
1948 Manchester Mark I, paměťová obrazovka, první počítač podle Von Neumannovi koncepce
1957 první komerčně používaný programovací jazyk - Fortran od IBM
1971 první sériově vyráběný procesor Intel 4004, 4 bity, jen pár instrukcí, určen pro kalkulačky, stejný výkon jako ENIAC, řídí i vesmírnou sondu Pioneer 10. Pro jeho univerzálnost se rozšířil i do jiných oborů a stal se na dlouhou dobu nejrozšířenějším procesorem v USA.
1972 osmibitový procesor 8008, frekvence 200Khz, 3500 tranzistorů.
1974 8080 další generace, další procesory např. Motorola, první osobní počítače Sinclair ZX81, Sinclair Spectrum, Apple a další
1978 8086, 8088, 29000 tranz. , 10Mhz, první 16ti bit od Intelu
1981 IBM PC, MSDos
1985 80386, podpora multitaskingu, první 32ti bitový procesor od Intelu, 275 tisíc tranzistorů, 33MHz
2000 Pentium 4, 1,5Ghz, 42 mil. Tranz., 2GHz
2003 první 64ti bitový procesor od AMD

Souběžně s vývojem procesorů pro osobní počítače probíhal i vývoj procesorů pro tzv. embedded (vestavěné) aplikace. Tyto procesory jsou určené do aplikací, jako jsou např. kalkulačky, řídící jednotky ve spotřební elektronice (pračkách, mikrovlnných troubách, televizích atd.), řídící jednotky v autech, řízení jednoduchých strojů v průmyslu apod.
Jejich hlavní charakteristikou je relativně malý výkon (ale dostačující pro danou aplikaci), malá spotřeba, malé rozměry a hlavně nízká cena. Mají většinou zabudovanou paměť pro data, paměť pro program a některé potřebné periferie, jako jsou např. časovače, A/D a D/A převodníky apod.. Tím minimalizují potřebu externích součástek.
Tato vývojová větev vychází z procesoru 8080 firmy Intel. Prvním takovým procesorem byl 8048, který měl integrovanou paměť pro program i data. Měl 6000 tranzistorů a pracoval na frekvenci 2Mhz. Následoval procesor 8021, ten již byl masivně používán ve spotřební elektronice.
V roce 1980 byl poprvé představen procesor 8051, který se stal asi nejrozšířenějším procesorem pro vestavěné aplikace. Tento procesor se používá dodnes, i když už ho nevyrábí firma Intel. Pro svou jednoduchost a univerzálnost se stal vzorem pro mnoho firem, které vyrábějí jeho klony. Tyto firmy vyrábějí mnoho různých procesorů s různou rychlostí, různými integrovanými periferiemi a v různých pouzdrech, které jsou vnitřně kompatibilní s původním procesorem 8051.
V dnešní době se v embedded aplikacích běžně používají i jiné typy procesorů, které nejsou kompatibilní s 8051. Mezi nejrozšířenější patří procesory od firem Microchip, Zilog, Motorola, Texas Instruments atd..

Procesor

Procesor je obecně součástka, která vykonává velmi jednoduché operace s datovou pamětí. Jedná se např. o přesun hodnoty z jedné paměťové buňky do jiné nebo sčítání obsahu dvou buněk.
To co však dělá procesor procesorem nejsou tyto jednoduché operace (říká se jim instrukce procesoru) sami o sobě ale to, že můžeme naprogramovat které a v jakém pořadí se mají vykonávat. Tím můžeme ovlivňovat to, jak se procesor bude chovat navenek.

Jednotlivé instrukce si procesor bere z programové paměti a vykonává je jednu po druhé. Tato programová paměť může být zcela oddělená od datové paměti a dokonce může být určena jen pro čtení, protože obvykle není potřeba měnit program za běhu (tzv. Harvardská architektura). Druhou možností je, že programová paměť je součástí datové paměti (tzv. Von Neumannova architektura). Oba dva tyto přístupy mají své výhody a nevýhody.
Harvardská architektura se používá zejména v malých systémech (například tzv. jednočipové mikroprocesory), kde je paměť RAM velmi cenná. Program je totiž uložen v paměti typu ROM (Read Only Memory) nebo EPROM (Erasable Programable Read Only Memory) a z této paměti ho procesor za běhu čte a hned ho vykonává. Pokud chceme program změnit, je potřeba tuto paměť přeprogramovat. Není ale nutné po každém spuštění program kopírovat do cenné datové paměti RAM (Random Access Memory), která tak zůstává volná pro data.
Oproti tomu Von Neumannova architektura je mnohem univerzálnější. Program je uložen v datové paměti a procesor tak na něj pohlíží stejně jako na data. Je možné ho za běhu modifikovat, což přináší mnohé nové možnosti. Nevýhodou je to, že tato architektura potřebuje nějaké paměťové médium, ve kterém by byl program uložen a po zapnutí se nahrál do datové paměti. Datová paměť je totiž obvykle typu RAM a po vypnutí napájení se její obsah ztratí. Proto se tato architektura používá zejména tam, kde je k procesoru připojeno jednak dostatečné množství paměti typu RAM a také nějaké paměťové médium, kde je program uložen před jeho nahráním do paměti a spuštěním. Proto asi nikoho nepřekvapí, že počítače IBM-PC používají právě tuto architekturu.

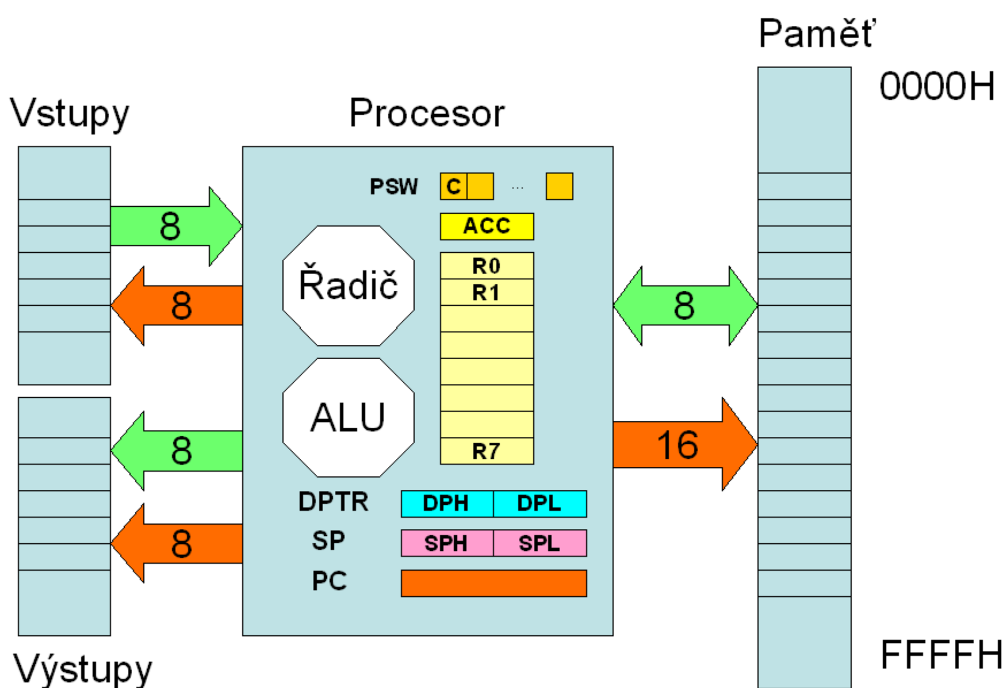
Další možné rozdělení procesorů je např. podle doby vykonávání jednotlivých instrukcí. Procesor je totiž ve své podstatě synchronní stroj = jeho chod je řízen hodinovým signálem, přivedeným zvenčí. Většinou je tento signál generován krystalovým oscilátorem a jedné jeho periodě říkáme hodinový cyklus. Doba trvání jednoho hodinového cyklu se pohybuje v řádu ns až µs.
Starší generace procesorů potřebovali pro vykonání jedné instrukce a všech souvisejících operací (tzv. instrukční cyklus) několik hodinových cyklů. Procesor totiž musí nejprve dekódovat instrukci, aby zjistil co bude pro její vykonání potřebovat. Poté si sáhne do paměti pro požadovaná data, provede požadovanou operaci a uloží výsledek do paměti. Takže např. procesorům Intel 8051 trval jeden instrukční cyklus dvanáct hodinových cyklů. Některé instrukce vyžadovali dokonce pro svoje vykonání více instrukčních cyklů, např. násobení trvalo 6 instrukčních cyklů, tzn. 72 hodinových. Procesory tohoto typu se nazývají obecně CISC (Complete Instruction Set Computer). Mají velké množství instrukcí (což komplikuje fázi dekódování instrukce) a některé instrukce jsou i poměrně složité. Jejich vykonávání je časově náročné a každá instrukce může zabrat jiné množství času.
Proto se v poslední době velmi prosazují procesory typu RISC (Reduced Instruction Set Computer). Tyto procesory mají menší množství instrukcí a jejich instrukce jsou jednodušší. To umožňuje jejich vykonávání během jednoho hodinového cyklu. Proto procesory RISC jsou mnohem výkonnější než procesory CISC na stejné hodinové frekvenci. Na druhou stranu je to vykoupeno tím, že na některé složitější operace (jako je např. násobení) nemají přímo instrukce a tak je potřeba tyto operace naprogramovat softwarově.

Kromě datové a programové paměti ještě procesor ke své práci potřebuje jeden druh paměti, a to vnitřní registry procesoru. Tato sada registrů se používá pro ukládání stavu procesoru (adresa právě vykonávané instrukce, ukazatel do datové paměti, ukazatel na konec zásobníku apod.) a obsahuje i univerzální registry pro ukládání mezivýsledků výpočtů.
Registry jsou vždy součástí procesoru a proto je přístup k nim velmi rychlý (rychlejší než k normální datové paměti). Některé procesory dokonce ani neumožňují provádět některé operace s daty jinde, než v těchto registrech. Data je potřeba do nich nahrát, provést požadovanou operaci a uložit výsledek z registru do datové paměti.

Programátorský model procesoru

V rámci tohoto předmětu budeme programovat procesor, který je velmi jednoduchý a je do velké míry inspirován procesory řady x51. Abychom ho mohli začít programovat, musíme si představit jeho model.

Tento náš virtuální procesor bude používat harvardskou architekturu, tzn. bude mít oddělenou paměť pro program a paměť pro data. Programovou paměť v našem modelu zanedbáme. Budeme předpokládat, že v ní program který se vykonává. Pro naše účely o ní nic dalšího vědět nepotřebujeme.
Pro zjednodušení budeme předpokládat, že je to procesor typu RISC a tak vykonání každé instrukce zabere právě jeden hodinový cyklus. Tento cyklus bude mít periodu 1 µs, tzn. že tento náš procesor vykoná milión instrukcí za vteřinu.
Celý procesor je osmibitový a proto všechny datové sběrnice a registry budou mít osm bitů. Adresová sběrnice k datové paměti bude šestnáctibitová. To znamená, že do paměti se vejde 2^{16} = 65536 bytů dat.



Aby procesor mohl komunikovat s okolním světem, potřebuje i nějaké vstupy a výstupy. Ty budou také osmibitové a pracovat se s nimi bude obdobně jako s datovou pamětí. Adresová sběrnice pro vstupy i výstupy bude osmibitová. To znamená, že vstupů a výstupů může být až 2^{8} = 256 bytů.

Uvnitř procesoru se nachází:

- Řadič instrukcí – dekóduje instrukci na vstupu a provede potřebné operace
- ALU – Aritmeticko-logická jednotka, ta provádí vlastní výpočty
- Univerzálně použitelné registry R0..R7 – slouží jako rychlá paměť pro výpočty
- Registr PSW (Program State Word) – tento registr obsahuje stav procesoru, v našem případě budeme používat z tohoto registru pouze dva bity, a to bit C (Carry) který je v jedničce pokud poslední matematická operace skončila přetečením a bit Z (Zero) který je v jedničce pokud výsledek poslední matematické operace byl 0
- Registr DPTR (Data Pointer) – slouží pro nepřímé adresování v datové paměti, je šestnáctibitový ale přistupujeme k němu osmibitově pomocí registrů DPL a DPH
- Registr SP (Stack Pointer) – slouží jako ukazatel na vrchol zásobníku, viz kapitola o zásobníku, podobně jako DPTR je šestnáctibitový ale pracujeme s ním osmibitově pomocí registrů SPL a SPH
- Registr PC (Program Counter) – tento registr obsahuje adresu aktuálně vykonávané instrukce v programové paměti. Je šestnáctibitový, takže program může mít maximálně 65536 instrukcí

Instrukce

Jednotlivé instrukce jsou uloženy za sebou v programové paměti. Každá instrukce se v paměti skládá z operačního kódu a operandů.
V reálném mikroprocesoru může zápis instrukce v paměti zabírat různé množství paměťových buněk. Instrukce která nemá žádné operandy zabere pouze jednu buňku, kdežto např. instrukce která ukládá konstantu do nějakého registru zabere tři buňky. Registr PC (Program Counter) obsahuje adresu aktuálně vykonávané instrukce a po startu procesoru je v něm hodnota 0. Po každém vykonání instrukce přičte řadič instrukcí do registru PC velikost instrukce v paměti a tak PC ukazuje na další instrukci, která se má vykonat. Jedinou vyjímkou jsou instrukce skoku, po kterých se může vykonávat i jiná instrukce, než která následuje za touto instrukcí.

Operační kód instrukce je v paměti samozřejmě uložen jako číslo. Instrukcí sice procesor nemá mnoho (jednočipové mikroprocesory mají řádově okolo 100 různých instrukcí), ale stejně je pro člověka nepředstavitelné, že by znal všechny jejich operační kódy zpaměti a pamatoval si jejich význam. Navíc by si musel pamatovat i kódy jednotlivých operandů a v případě skoků by musel při každé změně v programu přepočítat adresy instrukcí v paměti.
Takový program by se nejen obtížně psal, ale i četl a ladil. Proto se procesory neprogramují přímo ve strojovém kódu ale v takzvaném jazyce symbolických adres (JSA), který se někdy označuje jako assembler. V tomto jazyce je každé instrukci přiřazena tzv. mnemonická zkratka. Stejně tak jsou zavedeny jednoduchá jména pro registry procesoru místo jejich čísel. Také je možné používat v takovém programu symbolická návěstí pro označení místa, kam ukazuje instrukce skoku.
Program napsaný v assembleru je pak nutné přeložit pomocí překladače assembleru do strojového kódu. Překladač nahradí jména instrukcí, registrů a návěstí konkrétními čísly a poskládá instrukce za sebe do paměti procesoru. Toto je triviální úloha, která nám ale přináší obrovský komfort při psaní programů na úrovni jednotlivých instrukcí.

V následujícím textu si budeme postupně představovat jednotlivé instrukce našeho virtuálního procesoru. Nebudeme se ale již zabývat jejich číselnou interpretací a ukládáním v paměti, ale budeme pracovat na úrovni JSA.

Taková instrukce pak vypadá např. takto:

```
mov A, R1
```

Zápis začíná vždy mnemotechnickou zkratkou instrukce a následuje seznam operandů, které jsou odděleny čárkou. Operandem může být:

- registr procesoru
- konstanta
- místo v paměti určené přímou adresou
- místo v paměti určené nepřímou adresou

```
mov A, R1
mov A, #123
mov A, 100
mov A, @DPTR
```

Číselné hodnoty je možné uvádět zcela rovnocenně v dekadické, hexadecimální nebo binární soustavě. Zápis číselné hodnoty musí začínat číslem 0..9 a pokud není v dekadické soustavě, musí končit písmenem, které určuje soustavu (B=binární, H=hexadecimální).

Přesun dat

```
mov X, Y
```

Instrukce pro přesun dat má mnemotechnickou zkratku MOV, což je zkratka z anglického slova „move“. Tato instrukce provede překopírování osmibitových dat ze zdroje Y do cíle X.

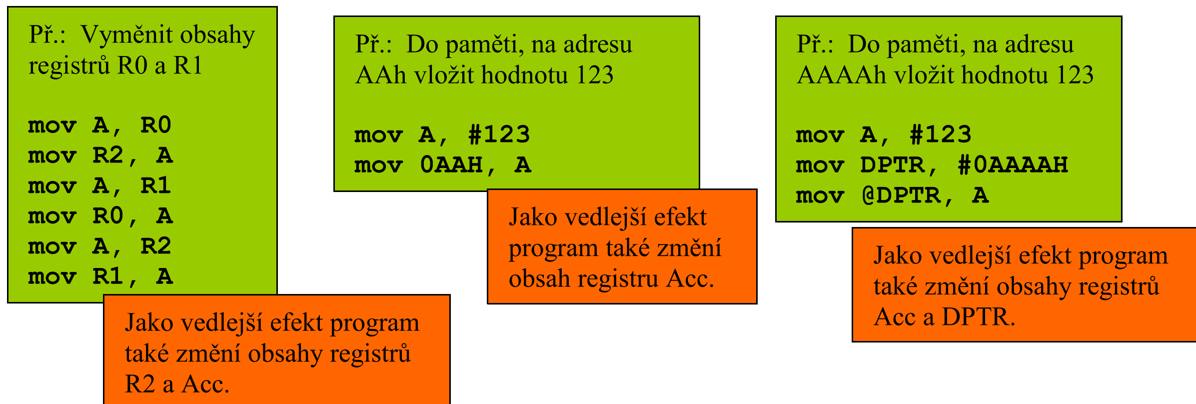
Seznam možných kombinací operandů u této instrukce:

mov A, R_n	mov R_n, A	mov A, #Data8	mov A, @R_n
mov A, DPH	mov DPH, A	mov R_n, #Data8	mov @R_n, A
mov A, DPL	mov DPL, A	mov DPTR, #Data16	mov A, @DPTR
mov A, SPH	mov SPH, A	mov SP, #Data16	mov @DPTR, A
mov A, SPL	mov SPL, A	mov A, Adresa8	mov Adresa8, A

Z ět o er i když tato instrukce kopíruje data osmibitově, existují i vyjímky. Do registrů SP a DPTR je možné vložit přímo šestnáctibitovou konstantu.

Operand R_n znamená, že lze použít libovolný z univerzálních registrů R0.. R7. Pokud jako operand chceme použít číselnou konstantu, pak před ní musíme napsat znak #. Pokud to neuděláme, bude tato konstanta chápána jako adresa v paměti. Takže např. instrukce „mov A , #1" uloží do registru A konstantu 1, kdežto instrukce „mov A , 1" uloží do registru A hodnotu, která je uložena v paměti na adrese 1.
Znak @ slouží pro nepřímé adresování. Takže např. pokud do registru R0 nahrajeme hodnotu 1, pak instrukce „mov A , @R0" uloží do registru A hodnotu, která je v paměti na adrese, která je uložená v registru R0. V tomto případě se do A opět nahraje hodnota z paměti na adrese 1.
Podobně je to i s registrem DPTR. Tento registr je potřeba použít v případě, že chceme pracovat s adresou nad 255. To je totiž maximální hodnota, kterou je možné nahrát do registrů R0..R7, protože jsou osmibitové. Proto místo těchto registrů lze pro nepřímé adresování použít registr DPTR, který je šestnáctibitový a tak umožňuje pracovat s pamětí až do adresy 65535.

Příklady použití instrukce MOV:



Vstup a výstup

```
in A, Adresa8
```

Instrukce IN zkopíruje data z vybraného vstupního portu se zadanou adresou do akumulátoru – registru A. Vstupní porty nemají funkci registru. Jejich obsah je v každém okamžiku dán děním mimo počítač. Instrukce IN zkopíruje obsah portu do registru A jen jednorázově, v okamžiku vykonání instrukce.

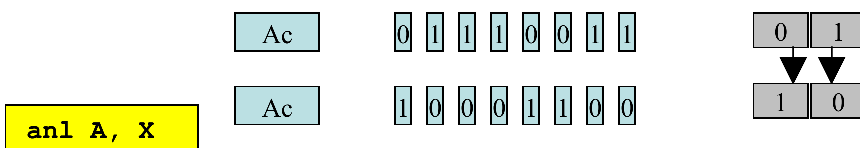
```
out Adresa8, A
```

Instrukce OUT zkopíruje data z akumulátoru na adresou vybraný výstupní port.
Výstupní porty mají funkci registru, jejich obsah je dán posledním provedením instrukce OUT.

Logické operace

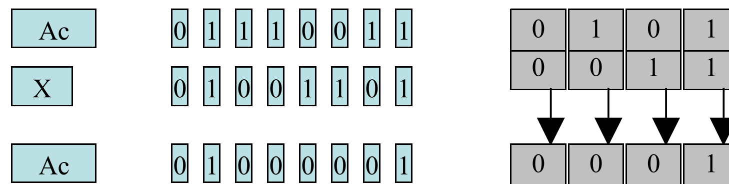
```
cpl A
```

Instrukce CPL (z anglického slova complement=doplněk) provede negaci všech osmi bitů v akumulátoru. To znamená, že každý bit, který byl v hodnotě 0 změní na 1 a naopak.




```
anl A, X
```

Instrukce ANL provede po bitech logický součin obsahu akumulátoru a druhého operandu X. Výsledek uloží do akumulátoru A. Operace součin se provede na dvojicích stejnolehlých bitů obou operandů.

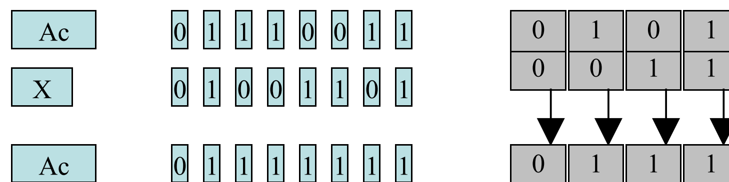


Seznam možných kombinací operandů:

```
anl A, Rn
anl A, #Data8
anl A, @Rn
anl A, Adresa8
```

orl A, X

Instrukce ORL provede po bitech logický součet obsahu akumulátoru a druhého operandu X. Výsledek uloží do akumulátoru A.

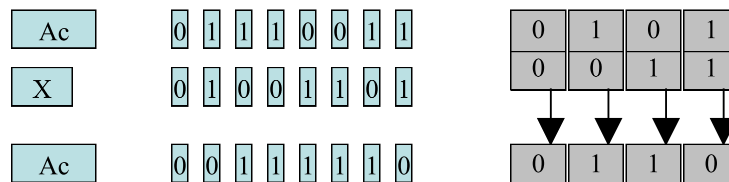


Seznam možných kombinací operandů:

```
orl A, Rn
orl A, #Data8
orl A, @Rn
orl A, Adresa8
```

xrl A, X

Instrukce ORL provede po bitech exkluzivní logický součet obsahu akumulátoru a druhého operandu X. Výsledek uloží do akumulátoru A.



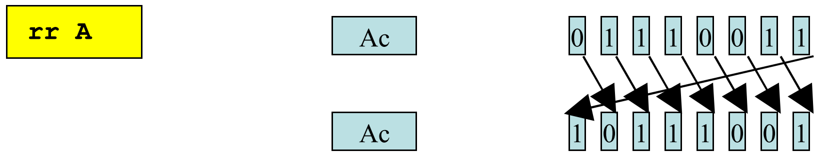
Seznam možných kombinací operandů:

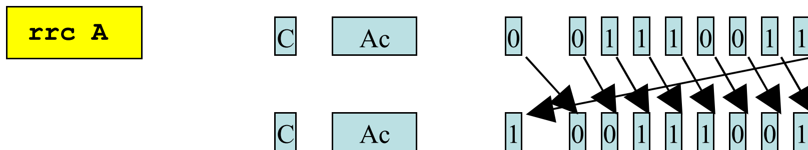
```
xrl A, Rn
xrl A, #Data8
xrl A, @Rn
xrl A, Adresa8
```

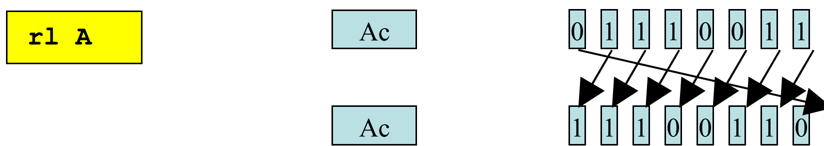
Všechny logické instrukce mají vliv na jednobitový registr Z. Pokud je výsledek v registru A nulový, pak nastaví tento registr na hodnotu 1. Pokud je výsledek nenulový, pak ho nastaví na hodnotu 0.

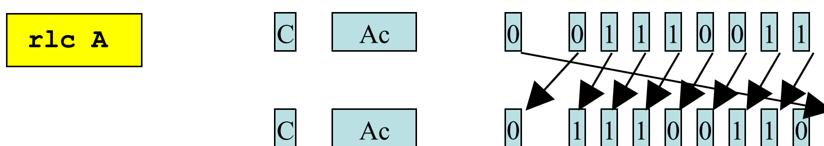
Rotace

Následující instrukce rotují jednotlivé bity doprava (RR a RRC) nebo doleva (RL a RLC). Pracují vždy s akumulátorem nebo s akumulátorem a jednobitovým registrem C (ten je součástí registru PSW).









Aritmetické operace

inc X

Instrukce INC (incrementation) provede zvýšení hodnoty operandu X o jedničku.
Seznam možných operandů:

inc A
inc R_n
inc DPTR

dec X

Instrukce DEC (decrementation) provede snížení hodnoty operandu X o jedničku.
Seznam možných operandů:

dec A
dec R_n

Instrukce INC a DEC nemají vliv na jednobitový registr C.

add A, X

Instrukce ADD provede aritmetický součet obsahu akumulátoru a druhého operandu X. Výsledek uloží do akumulátoru A.
Seznam možných operandů:

```
add A, Rn
add A, #Data8
add A, @Rn
add A, Adresa16
```

```
addc A, X
```

Instrukce ADD provede aritmetický součet obsahu akumulátoru, druhého operandu X a jednobitového registru C. Výsledek uloží do akumulátoru A.
Seznam možných operandů:

```
add A, Rn
add A, #Data8
add A, @Rn
add A, Adresa16
```

Instrukce ADD a ADDC počítají výsledek na 9 bitů. Devátý bit výsledku se ukládá do jednobitového registru C.

```
subb A, X
```

Instrukce SUBB provede aritmetický rozdíl obsahu akumulátoru, druhého operandu X a jednobitového registru C. Výsledek uloží do akumulátoru A.
Seznam možných operandů:

```
add A, Rn
add A, #Data8
add A, @Rn
add A, Adresa16
```

Vykonání instrukce SUBB ovlivňuje obsah jednobitového registru C. Je-li výsledek výpočtu nezáporný, je C=0. Je-li výsledek záporný, je C=1.

Skoky

```
jmp Adresa16
```

Instrukce JMP (jump) je instrukce nepodmíněného skoku. To znamená, že vykonání této instrukce způsobí nahrání nové adresy do registru PC a následně se vykonávají instrukce od této adresy. Adresu lze zadat přímo jako číselnou konstantu, ale to je velmi nepraktické. Místo toho se používají pro skoky symbolická návěstí. Takové návěstí si definujeme pomocí identifikátoru zakončeného dvojtečkou a danou instrukci můžeme odkazovat tímto jménem. Každé návěstí může být v programu definováno pouze jednou.
Pokud chceme provést skok na aktuální instrukci, můžeme místo návěstí použít znak $. V tomto případě provede instrukce JMP skok na sebe a vytvoří tím nekonečnou smyčku, tzv. dynamický stop.

Př.: nekonečná smyčka

```
      mov A, #0
LOOP: inc A
      out 0AAH
      jmp LOOP
```

Př.: dynamický STOP

```
      mov A, #55H
LOOP: jmp LOOP
```

Př.: dynamický STOP

```
      mov A, #55H
      jmp $
```

Následující instrukce jsou tzv. podmíněné skoky. Tyto instrukce provedou skok na jiné místo programu pouze tehdy, je-li splněna nějaká podmínka. Používají se pro větvení programu.

`jz Adresa16`

Jump if Zero: Pokud je jednobitový registr Z = 1 (tzn. výsledek poslední matematické nebo logické operace byl nula) pak se tato instrukce chová jako instrukce JMP. Jinak se chová jako instrukce NOP (No Operation).

`jnz Adresa16`

Jump if Not Zero: Pokud je jednobitový registr Z = 0 (tzn. výsledek poslední matematické nebo logické operace byl nenulový) pak se tato instrukce chová jako instrukce JMP. Jinak se chová jako instrukce NOP (No Operation).

`jc Adresa16`

Jump if Carry: Pokud je jednobitový registr C = 1 (tzn. při poslední matematické operaci došlo k přetečení) pak se tato instrukce chová jako instrukce JMP. Jinak se chová jako instrukce NOP (No Operation).

`jnc Adresa16`

Jump if Not Carry: Pokud je jednobitový registr C = 0 (tzn. při poslední matematické operaci nedošlo k přetečení) pak se tato instrukce chová jako instrukce JMP. Jinak se chová jako instrukce NOP (No Operation).

`cjne A, #data8, Adresa16`

Compare and Jump if Not Equal: Tato instrukce provede porovnání obsahu akumulátoru s daty. Pokud A<>data8, pak se provede skok na zadanou adresu. Jinak se pokračuje další instrukcí. Navíc pokud A<data8, pak se nastaví jednobitový registr C na 1, jinak na 0.

cjne R_n, #data8, Adresa16

Tato instrukce je naprosto stejná, jen místo akumulátoru pracuje s registry R0..R7.

djnz R_n, Adresa16

Decrement and Jump if Not Zero: Instrukce nejprve sníží o jedničku obsah registru a pokud je výsledek různý od nuly, pak se provede skok na zadanou adresu. Jinak se pokračuje následující instrukcí.

Instrukce JZ, JNZ, JC a JNC se používají zejména pro větvení programu na základě nějaké podmínky nebo pro tvorbu podmíněných cyklů.

Instrukce CJNE a DJNZ jsou vhodné pro vytváření nepodmíněných cyklů.

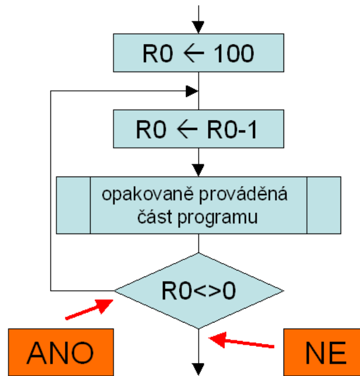
Příklady použití instrukcí skoků:

Cykly řízené proměnnou

Př.: ... počet průchodů smyčkou

```
        ...
        mov R0, #100
LOOP:
        dec R0
        mov A, #1
        out 01H, A
        nop
        nop
        nop
        mov A, #0
        out 01H, A
        mov A, R0
        jnz LOOP
        ...
```

Registr R0 použit jako počitadlo průchodů

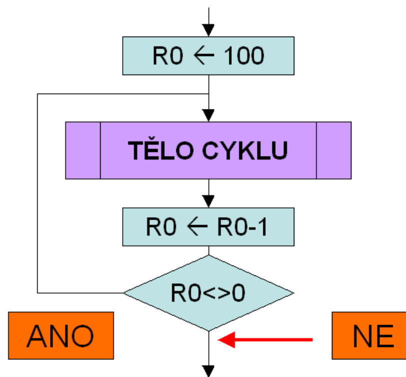


Cykly řízené proměnnou

Př.: ... počet průchodů smyčkou

```
        ...
        mov R0, #100
LOOP:
        mov A, #1
        out 01H, A
        nop
        nop
        nop
        mov A, #0
        out 01H, A
        djnz R0, LOOP
        ...
```

Registr R0 použit jako počitadlo průchodů



Uvniř těla cyklu s instrukcí DJNZ neměnit hodnotu proměnné, která cyklus řídí !!!

Neskákat do a z těla cyklu

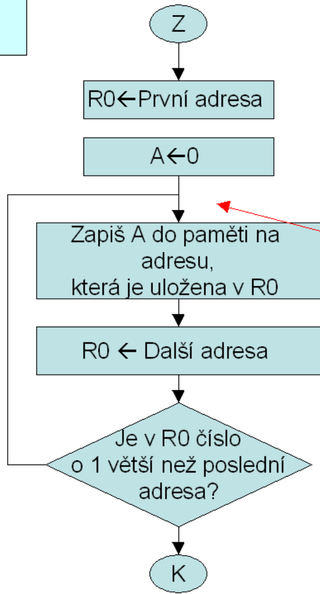
Příklad (paměť 1)

Cyklem je to lepší

Sestavte program, který, vynuluje oblast paměti mezi adresami AAh a BBh (včetně).

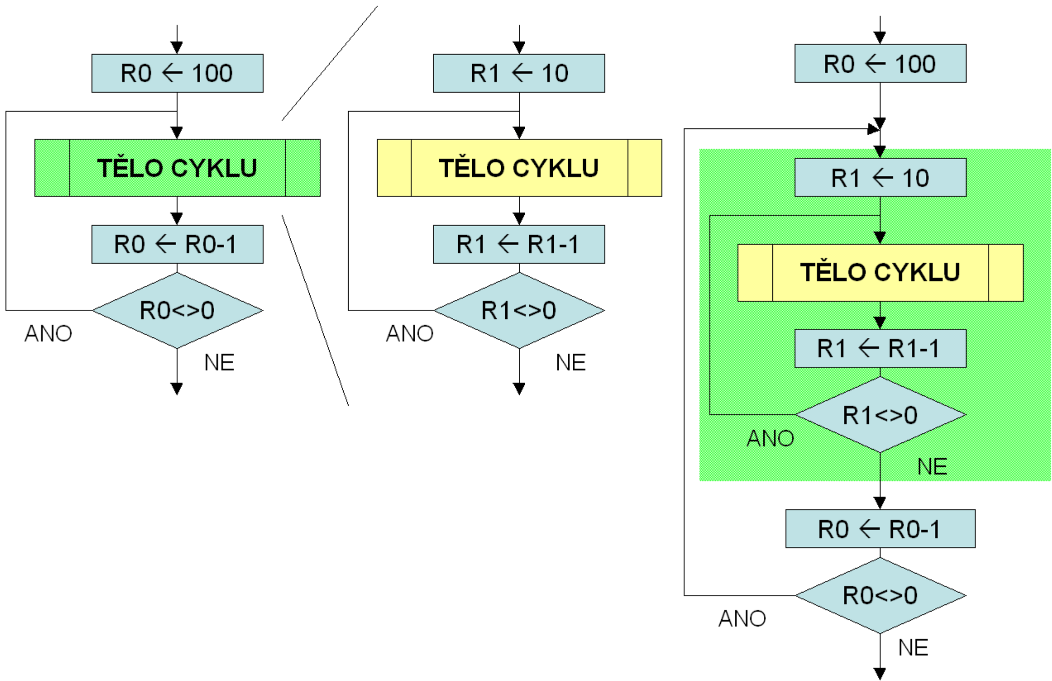
Má se nulovat jen 17 bytů. Šlo by to i „ručně“:

```
mov A, #0,
mov 0AAh, A
mov 0ABh, A
mov 0ACh, A
…
mov 0BBh, A
```




```
    mov R0,#0AAh
    mov A, #0
L1: mov @R0, A
    inc R0
    cjne R0,#0BCh,L1
    jmp $
```

Vnořené cykly

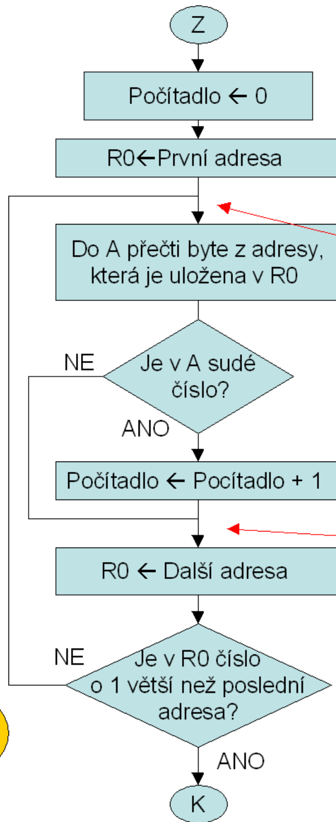


Příklad (paměť 2)

Sestavte program, který zjistí, kolik bytů uložených v paměti Mezi adresami AAh a BBh splňuje tu podmínku, že představuje ve standardním binárním váhovém kódování sudé číslo. Výsledek uložit do registrů procesoru.

Jak se pozná sudé číslo?

Jak veliký může být výsledek ? Vejde se mi do jednoho registru? Co když ne?

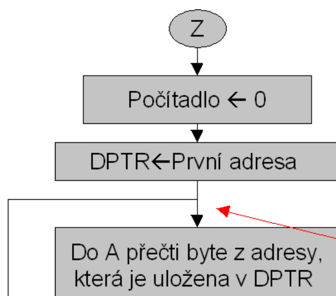



```
        mov  R7,#0
        mov  R0,#0AAh
L1:     mov  A,@R0
        rrc  A
        jc  L2
        inc  R7
L2:     inc  R0
        cjne  R0,#0BCh,L1
        jmp  $
```

Výsledek je v R7 !

Příklad (paměť 3)

Sestavte program, který zjistí, kolik bytů uložených v paměti Mezi adresami AAEEh a ABBBh splňuje tu podmínku, že



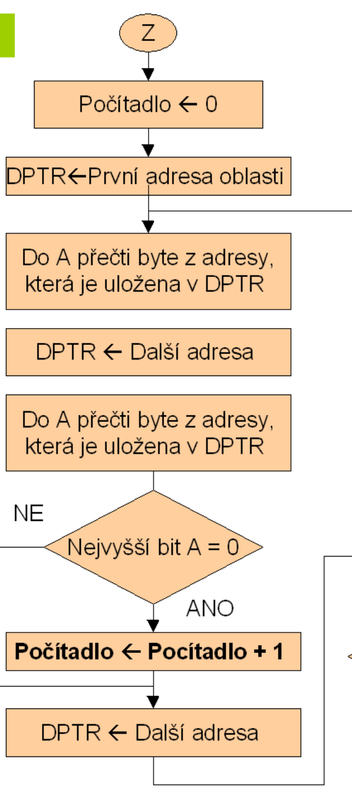

```
        mov  R7,#0
        mov  DPTR,#0AAEEh
L1:     mov  A,@DPTR
```

Příklad (paměť 5)

Sestavte program, který zjistí, kolik čísel ve formátu Integer uložených v paměti mezi adresami 0A000h a 0AFFFh je záporných.

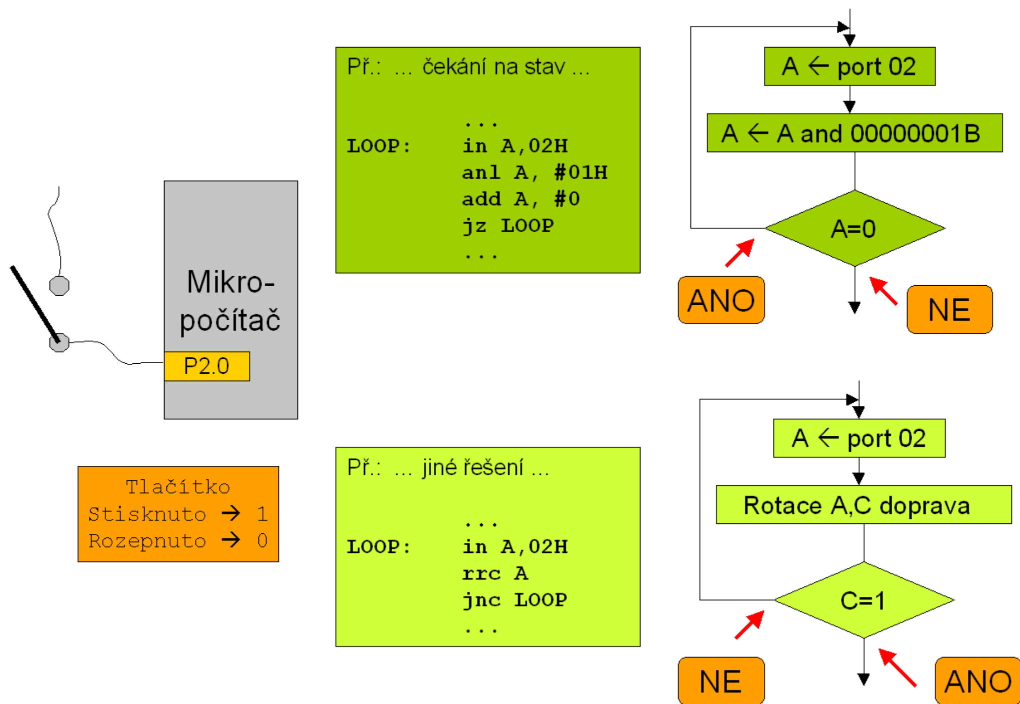
Každé číslo je uloženo na dvou po sobě jdoucích bytech.

O znaménku čísla rozhoduje nejvyšší bit druhého bytu.

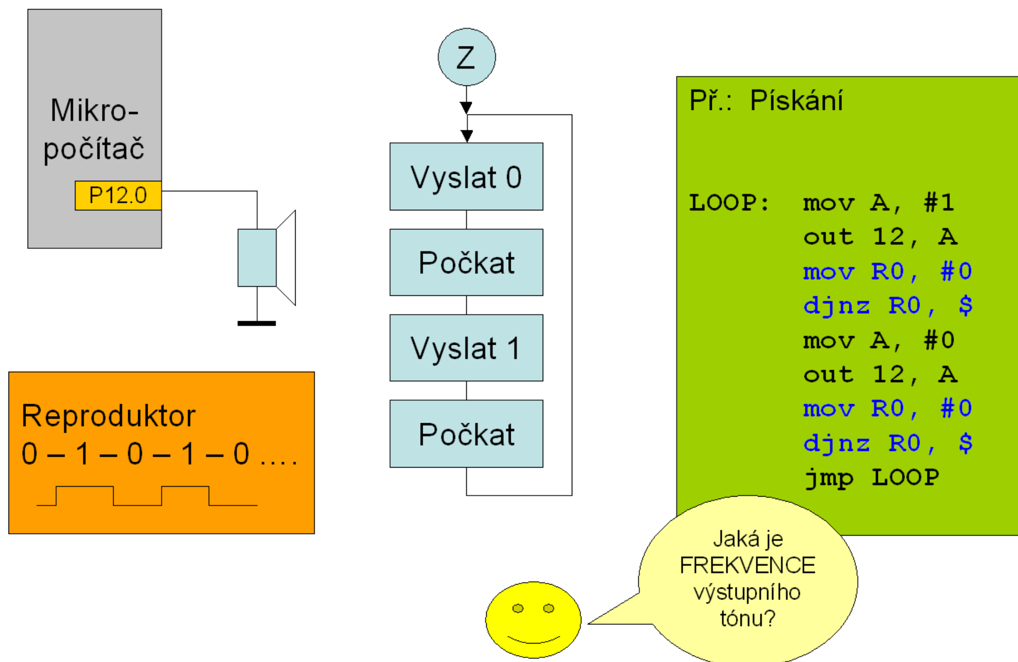



```
        mov  R7,#0
        mov  R6,#0
        mov  DPTR,#0A000h
L1:     mov  A,@DPTR
        inc  DPTR
        mov  A,@DPTR
        rlc  A
        jnc  L2
        inc R7
        mov A, R7
        jnz L2
        inc R6
L2:     inc  DPTR
        mov  A, DPH
        cjne  A,#0B0h,L1
        mov  A, DPL
        cjne  A,#000h,L1
        jmp  $
```

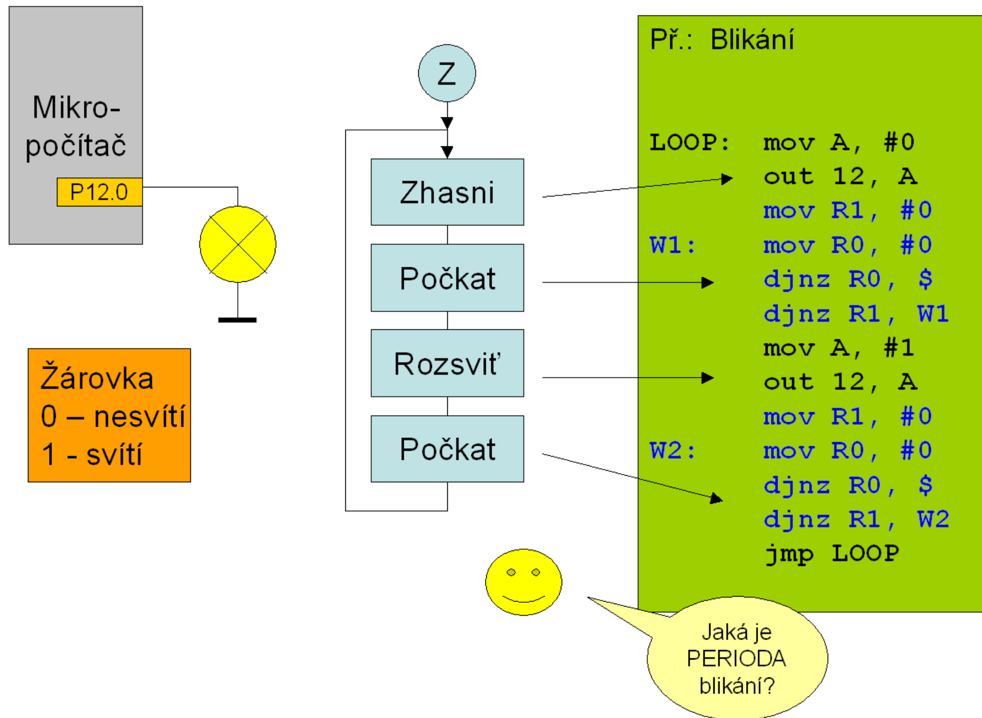
Cykly řízené podmínkou



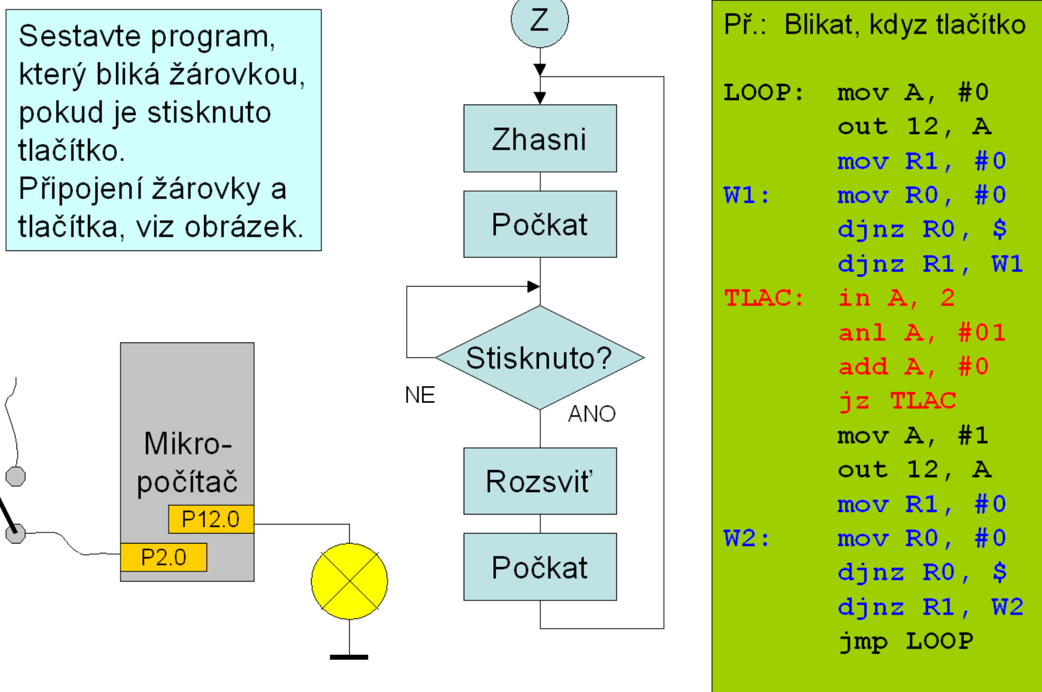
Příklady na cykly



Příklady na cykly

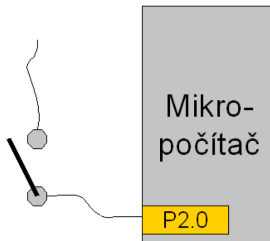


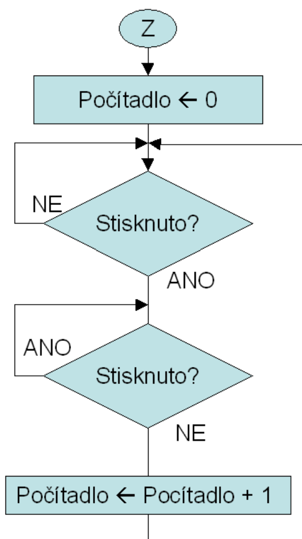
Příklad



Příklad (tlačítko 1)

Sestavte program, který v registru R7 udržuje počet stisků tlačítka.



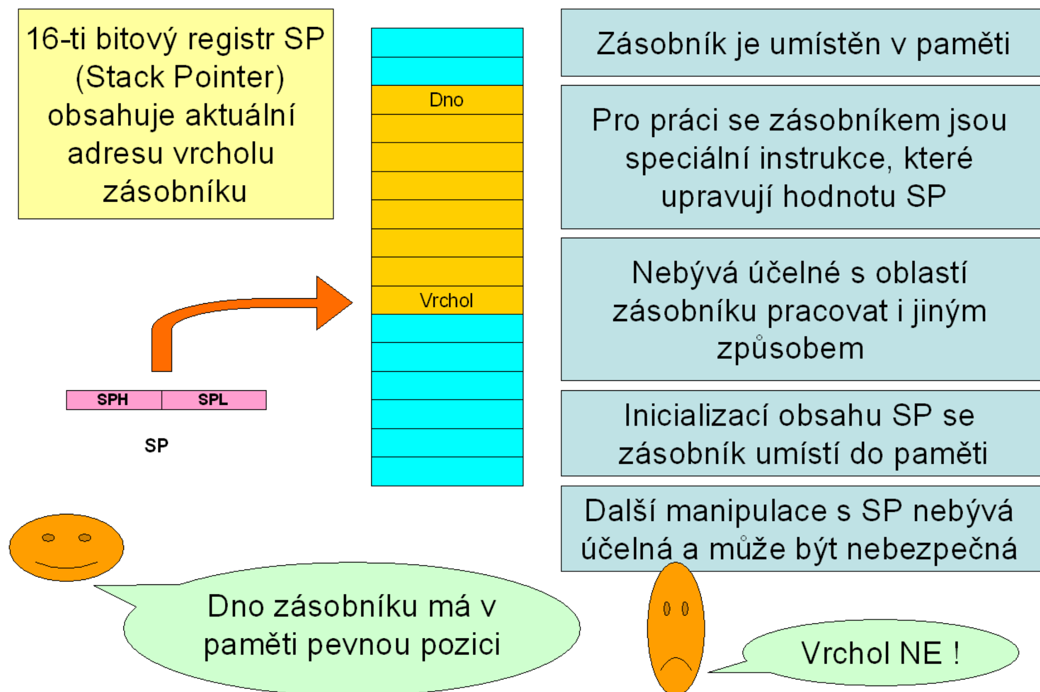



```
       mov R7, #0
B1:    in A, 2
       rrc A
       jnc B1
B2:    in A, 2
       rrc A
       jc B2
       inc R7
       jmp B1
```

Po 256 stiscích je registr vynulován a čítání probíhá znovu od nuly

Práce se zásobníkem

Zásobník je prostor v paměti, který slouží pro dočasné ukládání dat. Na aktuální vrchol zásobníku ukazuje registr SP (Stack Pointer). Pro práci se zásobníkem jsou určené speciální instrukce – PUSH a POP.



push X

Instrukce PUSH slouží pro vkládání dat do zásobníku. Provede nejprve zvýšení obsahu registru SP o jedničku a poté uloží obsah operandu X do paměti na adresu, která je uložena v registru SP.
Možné operandy:

```
push A
push Rn
push PSW
```

pop X

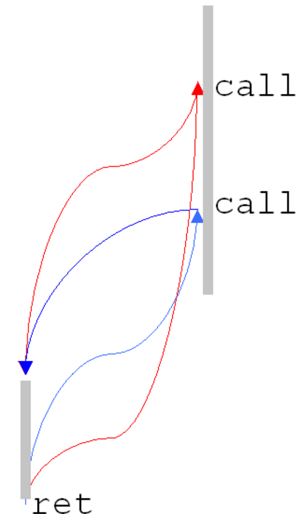
Instrukce POP slouží pro vybírání dat ze zásobníku. Nejprve zkopíruje data z paměti na adrese, která je uložena v SP do operandu X a poté sníží obsah registru SP o jedničku.
Možné operandy:

```
pop  A
pop  Rn
pop  PSW
```

Počet provedených instrukcí POP by měl být stejný, jako počet provedených instrukcí PUSH. V opačném případě by mohlo dojít k tomu, že se bude zásobník neustále zvětšovat nebo zmenšovat až dojde k přetečení nebo podtečení paměti.

Podprogramy

Podobně jako při skoku se při volání podprogramu předává řízení instrukci na jiném místě programu. Na rozdíl od skoku se ale předpokládá návrat do místa, odkud byl podprogram volán. Proto je nutné nejprve uložit adresu aktuální instrukce (registr PC – Program Counter) a pak teprve provést skok. Na konci podprogramu se tato adresa opět vyzvedne a program pokračuje na původním místě, odkud byl podprogram volán. K tomuto slouží instrukce CALL a RET.



call Adresa16

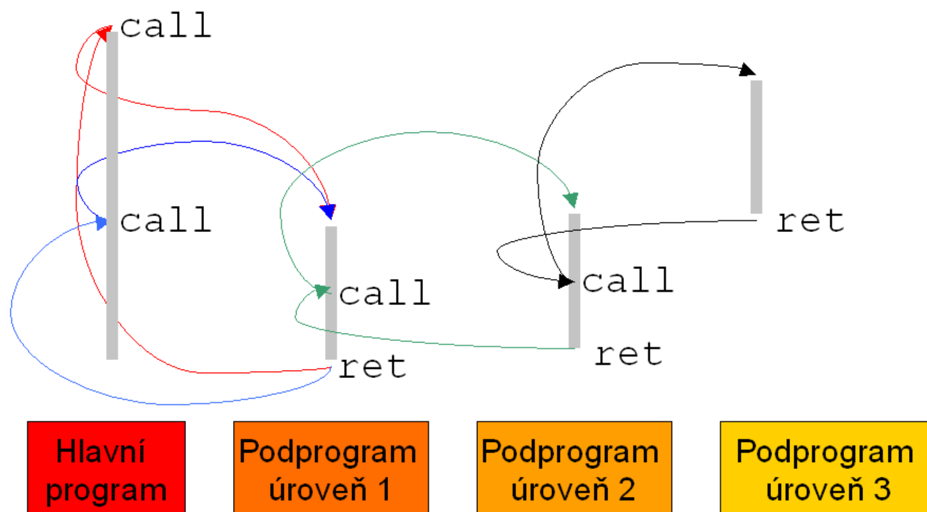
Instrukce CALL uloží do zásobníku (stejným způsobem jako PUSH) nejprve nižší a poté vyšší byte adresy té instrukce, která následuje za instrukcí CALL. Potom se provede skok na zadanou adresu.

ret

Instrukce RET odebere ze zásobníku nejprve vyšší a poté nižší byte adresy instrukce, které poté předá řízení.

Každým voláním podprogramu se nejprve velikost zásobníku zvětší o dva byte (uloží se návratová adresa) a po návratu z podprogramu se opět zmenší o dva byte (odebere se návratová adresa). Proto opět platí, že počet provedených instrukcí CALL musí být stejný jako instrukcí RET.
Dále by se v rámci podprogramu neměli používat instrukce skoku, které míří mimo aktuální podprogram. Stejně tak mimo podprogram by se neměli používat instrukce skoku, které míří do podprogramu.

Z podprogramu je možné provádět volání dalších podprogramů a to do téměř libovolného množství úrovní. Omezením je pouze velikost volné paměti pro zásobník (pro ukládání návratových adres).



Příklady na podprogramy

Př.: Pískání

```
LOOP:   mov A, #1
        out 12, A
        mov R0, #0
        djnz R0, $
        mov A, #0
        out 12, A
        mov R0, #0
        djnz R0, $
        jmp LOOP
```

Př.: Pískání

```
LOOP:   mov A, #1
        out 12, A
        call WAIT
        mov A, #0
        out 12, A
        call WAIT
        jmp LOOP

WAIT:   mov R0, #0
        djnz R0, $
        ret
```

Příklady na podprogramy

Př.: Blikání

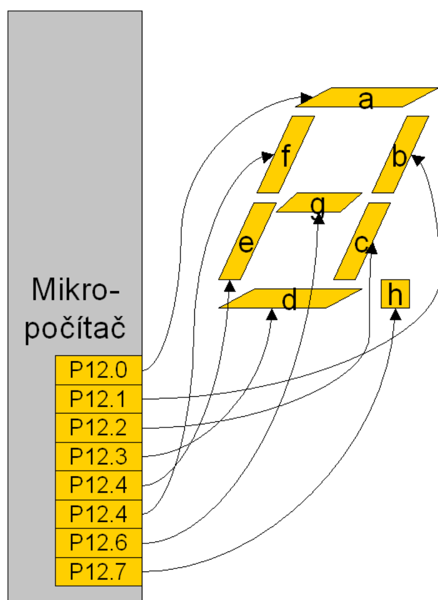
```
LOOP:   mov A, #0
        out 12, A
        mov R1, #0
W1:     mov R0, #0
        djnz R0, $
        djnz R1, W1
        mov A, #1
        out 12, A
        mov R1, #0
W2:     mov R0, #0
        djnz R0, $
        djnz R1, W2
        jmp LOOP
```

Př.: Blikání

```
LOOP:   mov A, #0
        out 12, A
        call W
        mov A, #1
        out 12, A
        call W
        jmp LOOP

W:      mov R1, #0
W1:     mov R0, #0
        djnz R0, $
        djnz R1, W1
        ret
```

Příklad podprogramu



Př.: Podprogram-Displej

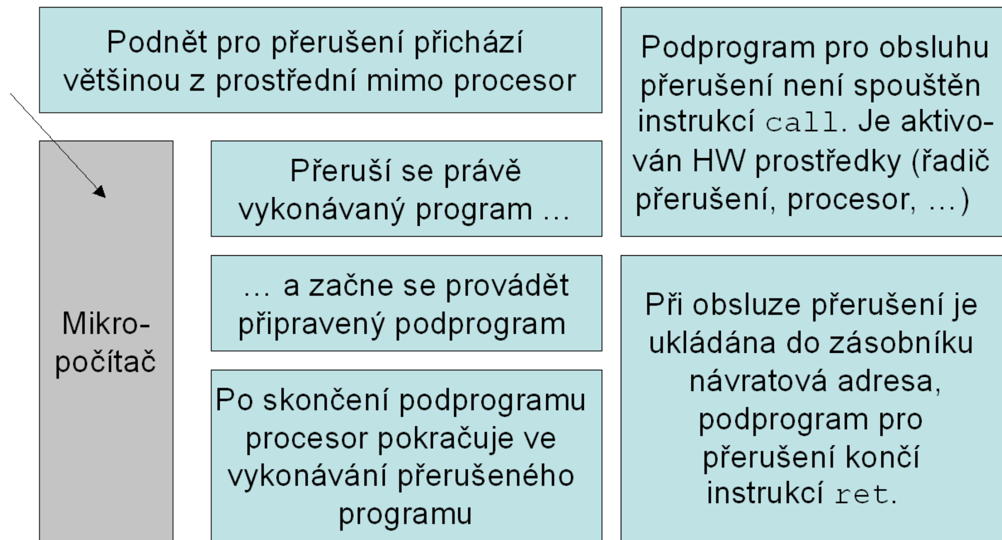
```
DISPL:     cjne R7, #0, D1
           mov A, #00111111B
           jmp DISEND
D1:        cjne R7, #1, D2
           mov A, #00000110B
           jmp DISEND
D2:        cjne R7, #2, D3
           mov A, #01011011B
           jmp DISEND
D3:        cjne R7, #3, D4
           mov A, #01001111B
           jmp DISEND
D4:        cjne R7, #4, D5
           mov A, #01100100B
           jmp DISEND
D5:        cjne R7, #5, D6
           mov A, #01101101B
           jmp DISEND
D6:        cjne R7, #6, D7
           mov A, #01111101B
           jmp DISEND
D7:        cjne R7, #7, D8
           mov A, #00000111B
           jmp DISEND
D8:        cjne R7, #8, D9
           mov A, #01111111B
           jmp DISEND
D9:        cjne R7, #9, ERR
           mov A, #01101111B
DISEND:    out 12H, A
ERR:       ret
```

Přehled instrukcí

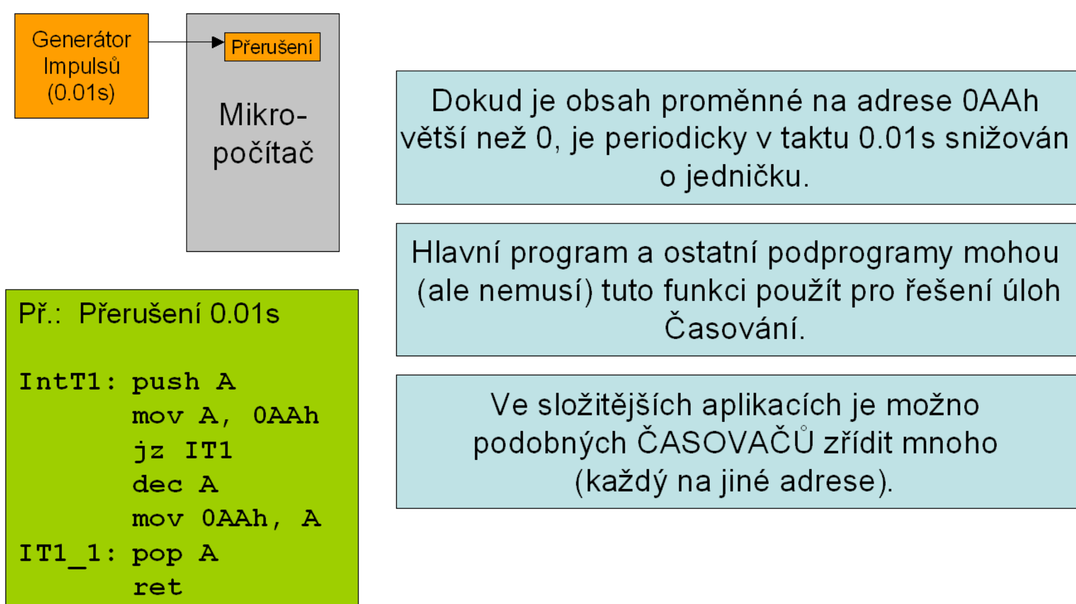
Skupina	Instrukce
Instrukce přesunu dat	`mov X, Y`
Instrukce vstupu a výstupu	`in A, Adresa8` · `out Adresa8, A`
Logické instrukce	`cpl A` · `anl A, X` · `orl A, X` · `xrl A, X`
Instrukce rotací	`rr A` · `rrc A` · `rl A` · `rlc A`
Aritmetické instrukce	`inc X` · `dec X`
	`add A, X` · `addc A, X` · `subb A, X`
Instrukce skoků	`jmp Adresa16` · djnz R_n,Adresa16
	`cjne A,#data8,Adresa16` · cjne R_n,#data8,Adresa16
	`jz Adresa16` · `jnz Adresa16`
	`jc Adresa16` · `jnc Adresa16`
Práce se zásobníkem	`push X` · `pop X`
Podprogramy	`call Adresa16` · `ret`
Prázdná instrukce	`nop`

Přerušení

Systém přerušení procesoru je velmi důležitý zejména pro obsluhu zařízení, připojených k procesoru. Pokud je přerušení správně nastaveno, může se procesor věnovat vykonávání hlavního programu a nemusí se zdržovat testováním, jestli není potřeba vykonat obslužnou rutinu. Pokud tato potřeba nastane, pak se aktivuje žádost o přerušení procesoru a procesor provede automaticky skok na předem nastavenou obslužnou rutinu. Po jejím vykonání se procesor opět vrátí k vykonávání hlavního programu. Zdrojem přerušení může být podnět vně procesoru (např. stisk tlačítka) nebo uvnitř procesoru (např. vnitřní časovač).



Příklad na přerušení



Př.: Přerušení 0.01s

```
IntT1: push A
       mov A, 0AAh
       jz IT1
       dec A
       mov 0AAh, A
IT1_1: pop A
       ret
```

Poznámky

Obsah předmětu

Číselné soustavy
Standardní formáty dat
Procesor, instrukční soubor
Periferní zařízení
Modelování činnosti počítače

Přednášky jsou nepovinné, cvičení povinná

Zápočet

Na základě práce při cvičeních, v půlce semestru test na převody, v zápočtovém týdnu zápočtový test, detaily určují cvičící

Zkouška

Zkoušky se budou konat v průběhu zkouškového období, zápis na zkoušku se provádí předem přes STAG.
Zkouška je písemná, 3-4 příklady/otázky

Náplň cvičení:

1. převody desítková - dvojková soustavy, celá čísla, desetinná část
2. převody desítková - hexadecimální – dvojková soustava, celá čísla
3. datové typy s pevnou řádovou čárkou a jejich ukládání do paměti - byte, word, dword, shortint, integer, longint
4. datové typy s plovoucí řádovou čárkou a jejich ukládání do paměti, single, real, double
5. test na převody soustav
6. seznámení s modelem procesoru, instrukční sada
7. vytváření sekvencí instrukcí
8. jednoduché programové konstrukce
9. cykly, časování programů
10. komunikace s periferiemi
11. automaty
12. zápočtový test
13. zápočtový týden – zápis zápočtů

Doporučená literatura:
WWW.FM.VSLIB.CZ / ~KSI / CZ / MATER / KSI_MAT.HTM
Ladislav Zajíček : Bity do bytu
Jan Rychlík : Programovací techniky
Boris Dědina, Pavel Valášek: Mikroprocesory a mikropočítače, SNTL 1981