

# 1 Úvod

V úvodní kapitole se seznámíme se základními pojmy mikroprocesorové techniky. Dále vysvětlíme způsob reprezentace údajů v počítači, sběrnice a paměti.

## 1.1 Základní pojmy

**Počítač** lze zjednodušeně označit jako stroj na automatizované zpracování informace. Struktura počítače nemusí nutně souviset s typem zpracovávaných úloh, ovšem některé architektury jsou pro určité problémy výhodnější.

**Program** je sled přesně stanovených dílčích úkonů vedoucích k vyřešení požadovaného úkolu. Zjednodušeně lze program označit jako návod na zpracování údajů (dat).

**Paměť** je obvod, který dokáže uchovávat informace v číslicové podobě. Rozlišujeme **paměť programu** (zde je uložen program, počítač si tedy „pamatuje“ jaké úkony má provádět) a **paměť dat** (zde jsou uložena vstupní data, tedy údaje pro zpracování; mezivýsledky operací; výstupní data, tedy výsledky získané během programu).

**Periferie** je zařízení, kterým komunikuje počítač s vnějším okolím. Jedná se o vstupní nebo výstupní prostředek. Vstupní prostředky slouží pro ovládání počítače. Pomocí výstupních prostředků sděluje počítač výsledky výpočtu. Vstupním prostředkem na úrovni klasického počítače může být například klávesnice, výstupním prostředkem pak monitor.

Jednotlivé části počítače jsou složeny z integrovaných obvodů. **Integrovaný obvod** umožňuje umístit na jediném polovodičovém čipu velké množství tranzistorů případně dalších součástek pro vytvoření složitějšího obvodového celku. Dnešní hustota integrace se pohybuje až ve stovkách miliónů tranzistorů na jednom čipu.

**Procesor** je patrně nejdůležitější částí počítače. Jedná se o výkonnou jednotku počítače. Procesor totiž odpovídá za provádění jednotlivých úkonů předepsaných programem a v souvislosti s tím „přetváří“ vstupní data na výstupní data. Procesor sestává z řadiče, aritmeticko-logické jednotky (ALU) a podpůrných obvodů.

**Řadič** odpovídá za dekódování (rozpoznání) jednotlivých úkonů programu. Na základě požadovaného úkonu dává pokyny pro zpracování dat pomocí ALU a produkuje výstupní data.

**ALU** (Arithmetic Logic Unit, tedy aritmeticko-logická jednotka) je část procesoru odpovědná za provádění aritmetických (početních) a logických operací (například: součet, rozdíl, součin, podíl, logický součet, logický součin, negace atp.). Operace, které ALU přímo nedokáže realizovat (u některých procesorů například násobení nebo dělení) je pak nutné realizovat „opisem“, tedy pomocí sledu náhradních operací, které ALU již dokáže realizovat.

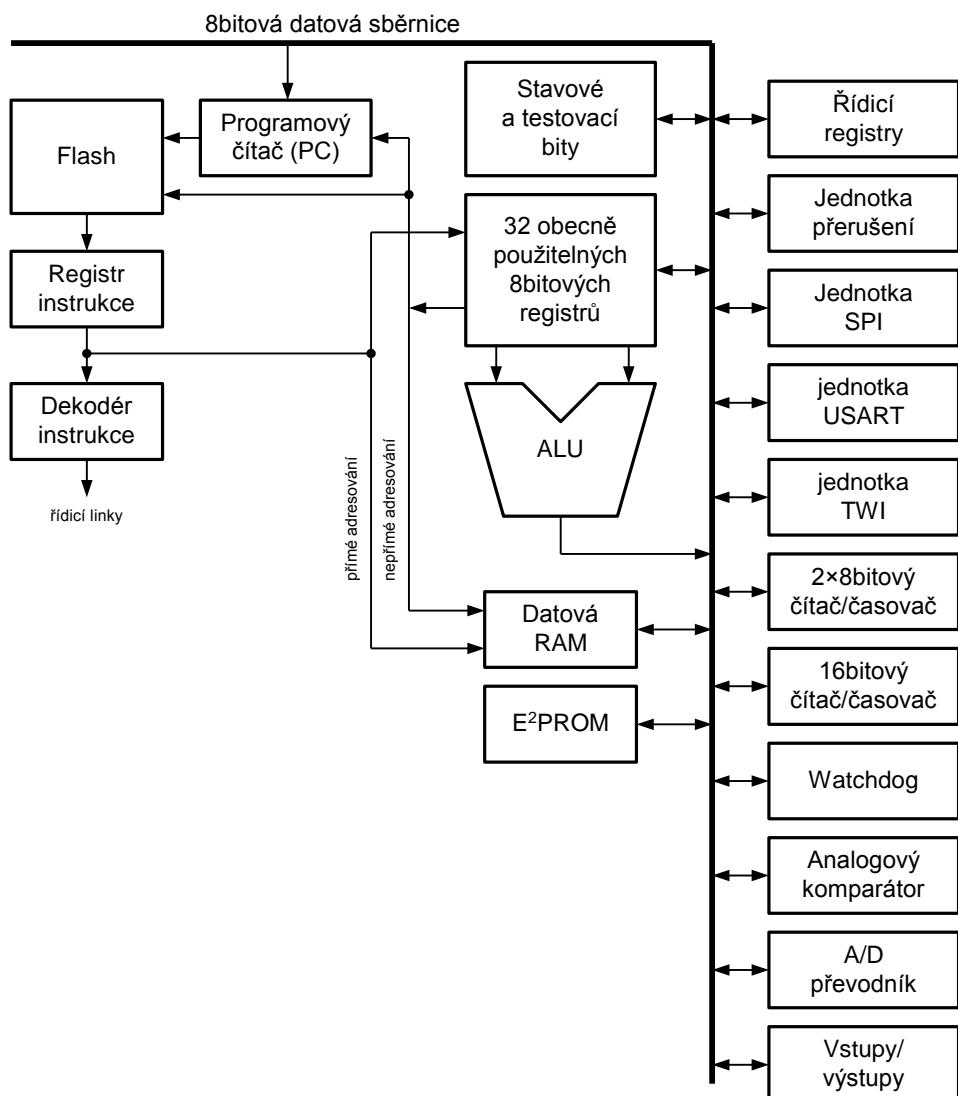
**Instrukce** je nejmenší jednotka, kterou lze v programu provést. Je to tedy již dále nedělitelný (elementární) úkon programu. Skupina všech instrukcí, které dokáže řadič rozpoznat, označujeme jako **instrukční soubor**.

**Registr** je používán jako dočasná buňka pro uložení operandu, který se zúčastní instrukce nebo k uchovávání výsledků instrukcí.

Dříve byl procesor tvořen několika dílčími integrovanými obvody, vysoká hustota integrace dovolila realizovat celý procesor na jediném polovodičovém čipu. Tento celek označujeme jako **mikroprocesor** nebo **CPU** (Central Processing Unit).

Samotný mikroprocesor by dokázal bez přídavných obvodů provést jen velmi málo operací. Kdyby nebyl mikroprocesor doplněn pamětí programu, nemohl by provádět program z toho důvodu, že by jej vlastně „neznal“. Podobně by mohl těžko ovládat připojená zařízení, když by nebyl vybaven obvody rozhraní. Samostatně pracujícím blokem je **mikropočítáč**, který představuje spojení mikroprocesoru, paměti programu a dat a obvodů rozhraní. Je-li tento celek integrován v rámci jediného integrovaného obvodu, označujeme jej jako **jednočipový mikropočítáč**.

V některých případech doplňujeme mikropočítáč dalšími poměrně komplexními periferiemi jako: čítače/časovače, obvody pro komunikaci po sběrnicích různého standardu, řadiče přímého přístupu do paměti (DMA) apod. Takovýto celek již dokáže (pouze doplněním přizpůsobovacích obvodů) ovládat i poměrně složité zařízení. Tento celek tedy označujeme jako **jednočipový mikrořadič**. Uvedený pojem je přejat z anglického označení **microcontroller**, takže se velmi často můžeme setkat se vžitým (bohužel nepřesným) označením **mikrokontrolér**.



Obr. 1.1 Příklad vnitřní stavby mikrokontroléru (AVR architektura)

## 1.2 Reprezentace čísel v počítaci

Číselné údaje lze reprezentovat v různých číselných soustavách.

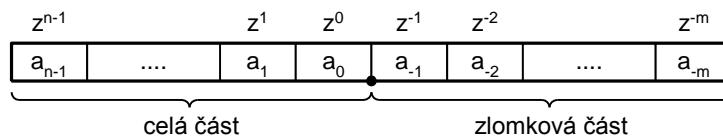
Nejběžnější je **desítková (dekadická) soustava**, ovšem vytvoření číslicových obvodů, které by dokázaly rozlišit 10 různých stavů, je obtížné. Mnohem snazší je realizace číslicových obvodů, které používají **dvojkovou (binární) soustavu**.

Obecný vztah pro vyjádření čísla  $A$  v soustavě se základem  $z$  je:

$$A_{(z)} = \sum_{i=-m}^{n-1} a_i \cdot z^i \quad (1-1)$$

kde (viz obr. 1.2):

- $A_{(z)}$  – číslo vyjádřené v soustavě se základem  $z$ ,
- $z$  – číselný základ (celé číslo větší než 1, tedy: 2, 3, 4, ...),
- $a_i$  – číslice (cifra),
- $z^i$  – váha číslice v daném řádu; pro celou část platí:  $i \geq 0$ , pro zlomkovou část platí  $i < 0$ .



Obr. 1.2 Vysvětlení zápisu čísla v soustavě se základem  $z$

1. **Příklad:** zápis čísla v desítkové soustavě:

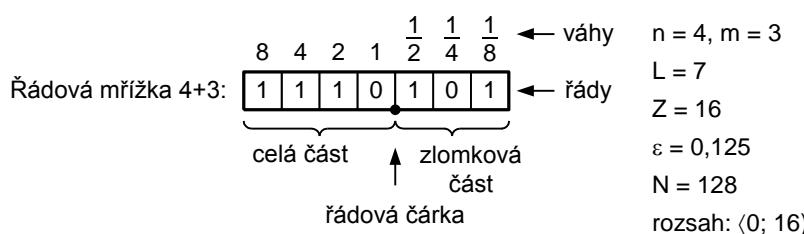
$$2504,36_{(10)} = 2 \cdot 10^3 + 5 \cdot 10^2 + 0 \cdot 10^1 + 4 \cdot 10^0 + 3 \cdot 10^{-1} + 6 \cdot 10^{-2}$$

2. **Příklad:** zápis čísla ve dvojkové soustavě:

$$1011101,101_{(2)} = 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}$$

### Řádová mřížka

Řádová mřížka je schéma pro zápis čísla a výpočet jeho hodnoty.



Obr. 1.3 Příklad řádové mřížky

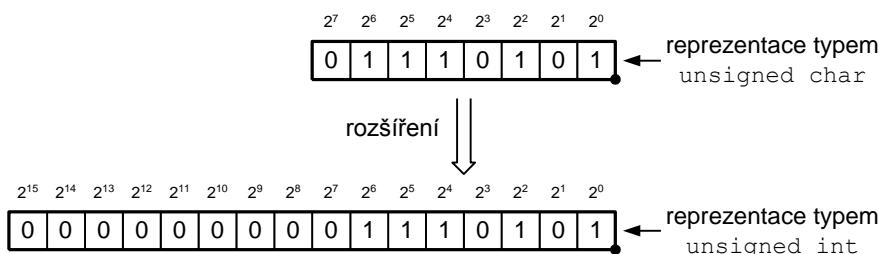
V souvislosti se řádovou mřížkou používáme níže uvedené pojmy (viz obr. 1.3):

- **řád** – pozice, na které je umístěna číslice (například pro desítkovou soustavu mluvíme o řádu jednotek, desítek, stovek,... nebo desetin, setin,...),
- **řádová čárka** – odděluje celou a zlomkovou část čísla,
- **délka řádové mřížky** – odpovídá celkovému počtu číslic, které lze vložit do řádové mřížky (tedy počtu číslic pro celou a zlomkovou část):  $L = n + m$ ,
- **modul řádové mřížky** – představuje velikost čísla zapsaného v řádové mřížce:  $Z = z^n$  (viz též *zobrazitelný rozsah*),

- **jednotka řádové mřížky** – odpovídá nejmenší možné nenulové hodnotě zapsané v řádové mřížce (tedy nejnižšímu rádu zlomkové části):  $\varepsilon = z^{-m}$ ,
- **počet všech možných čísel** – odpovídá počtu vzájemně odlišných kombinací číslic, které lze do řádové mřížky zapsat:  $N = z^{n+m}$ ,
- **zobrazitelný rozsah** – odpovídá skutečnosti, že nejnižší zobrazitelná hodnota je nula (včetně) a nejvyšší zobrazitelná hodnota se blíží zleva modulu, tedy:  $\langle 0; Z \rangle$ , pokud neuvažujeme čísla se znaménkem.

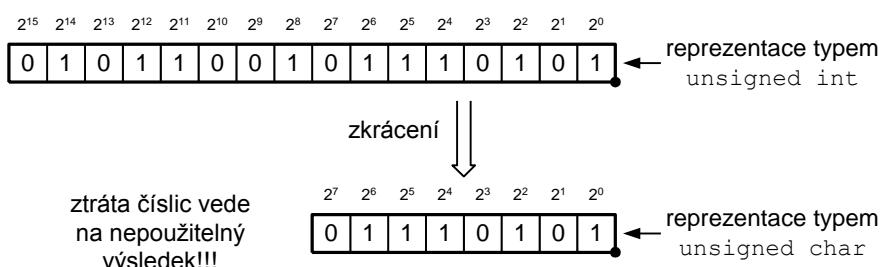
Je zřejmé, že číslo uložené v paměti počítače má vždy omezený rozsah. Dojde-li při výpočtu k situaci, že výsledek přesahuje nejvyšší rád mřížky, mluvíme o **přetečení** (overflow). Výsledek je dále nepoužitelný a ALU indikuje tento stav nastavením příznaku **carry** (přenos – značí přenos do vyššího rádu).

Při používání čísel s různou šírkou řádové mřížky je třeba takto vyjádřená čísla vzájemně převádět. Pokud ukládáme hodnotu vyjádřenou užší řádovou mřížkou do paměti vyhrazené na větší číselné údaje, dochází k **přidávání řádů vlevo**. Příkladem může být konverze hodnoty typu `unsigned char` na hodnotu typu `unsigned int`. Situace je vysvětlena formou obr. 11.4 (předpokládáme, že šíře řádové mřížky datového typu `unsigned int` je 16).



Obr. 1.4 Příklad rozšíření hodnoty čísla

V opačném případě (číslo vyjádřené pomocí širší řádové mřížky ukládáme do paměti vyhrazené pro menší číselné údaje) dochází k **ubírání řádů vlevo**, tedy zkracování řádové mřížky. Tato operace nemusí být bezeztrátová, jak dokumentuje obr. 11.5.



Obr. 1.5 Příklad zkrácení hodnoty čísla

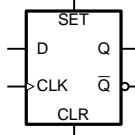
**Čísla s pevnou řádovou čárkou** jsou reprezentována tak, že se poloha řádové čárky během výpočtu nemění. U čísel s **plovoucí řádovou čárkou** může při výpočtu docházet ke změně polohy řádové čárky.

### 1.3 Registr

Označení registr jsme použili již v předchozím textu v souvislosti s vnitřní stavbou mikroprocesoru. Nyní budeme uvažovat jiný případ.

Jako **registr** se obvykle označuje **klopný obvod typu D řízený hranou**. Schématická značka je uvedena formou obr. 1.6.

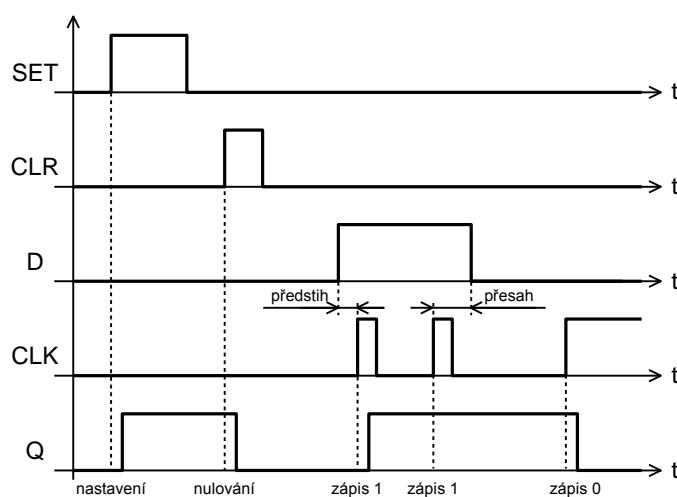
Níže uvedená schematická značka registru odpovídá obvodu, který má dva prioritní vstupy pro nastavení (SET, výstup je nastaven na 1) a nulování (CLR, výstup je nastaven na 0). Jsou-li vstupy **SET** a **CLR** neaktivní, přepíše se stav datového vstupu **D** na výstup **Q** v okamžiku náběžné hrany hodinového signálu **CLK**. V ostatních případech registr udržuje předchozí stav.



SET	CLR	D	CLK	$Q_n$	$\bar{Q}_n$	poznámka
0	1	X	X	0	1	nulování
1	0	X	X	1	0	nastavení
0	0	X	0	$Q_{n-1}$	$\bar{Q}_{n-1}$	pamatuje
0	0	X	1	$Q_{n-1}$	$\bar{Q}_{n-1}$	pamatuje
0	0	X	$1 \rightarrow 0$	$Q_{n-1}$	$\bar{Q}_{n-1}$	pamatuje
0	0	0	$0 \rightarrow 1$	0	1	zápis 0
0	0	1	$0 \rightarrow 1$	1	0	zápis 1

Obr. 1.6 Schématická značka registru a jeho pravdivostní tabulka

Jiný způsob výkladu funkce registru uvádí obr. 1.7. Jedná se o časové průběhy signálů operací s registrtem. V první části vidíme aktivitu signálu **SET**, který vede k nastavení registru ( $Q = 1$ ). V obrázku je rovněž naznačeno, že u skutečného registru není reakce okamžitá, ale zpožděná. Dále je naznačena aktivita signálu **CLR**, dochází tedy k nulování ( $Q = 0$ ).

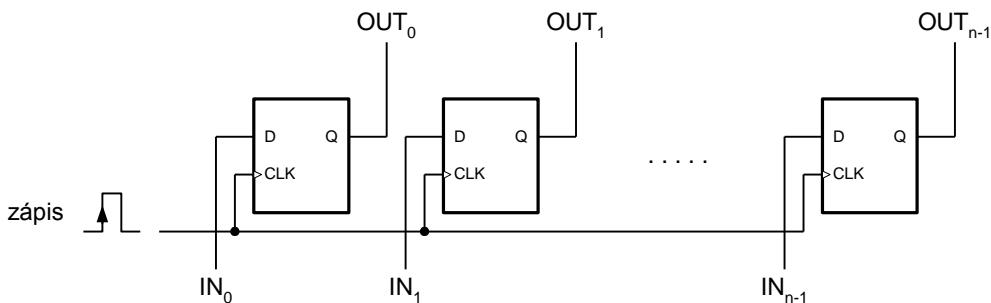


Obr. 1.7 Časový diagram operací s registrtem

V druhé části obrázku je naznačen zápis pomocí náběžné hrany hodinového signálu. Fyzicky je nutné, aby signál na datovém vstupu **D** byl ustálen před příchodem zapisovací hrany **CLK**. Jedná se o tzv. **předstih dat**. Podobně musí být

zaručeno, že v průběhu zápisu nedojde ke změně vstupního údaje, je tedy vyžadován tzv. **přesah dat**.

Pro uložení vícebitových údajů lze používat **N-bitový registr**, jehož vnitřní schéma je naznačeno formou obr. 1.8. Jedná se vlastně o N registrů, které jsou řízeny společným hodinovým signálem. Pro každý bit vstupního údaje je potřeba použít jeden jednobitový registr.



Obr. 1.8 N-bitový registr

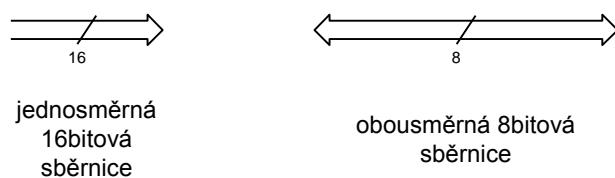
Hlavní výhodou registrů je velmi krátký vybavovací interval. Registry se proto používají všude, kde je potřeba rychle přistupovat k datům. Proto jsou registry integrovány přímo v CPU.

## 1.4 Sběrnice

Sběrnice je soustava vodičů, která vzájemně propojuje jednotlivé části počítače. Jedná se o vodiče, které spolu souvisí. Každý vodič „nese“ část (bit) téže informace. Proto se pro značení vodičů používá společný název a vodiče jsou rozlišeny indexem (počítaným od 0), který obvykle vyjadřuje váhu bitu. Například pro označení datové sběrnice používáme značení:  $D_{N-1} \dots D_0$  nebo  $D[N-1:0]$ .

Některé sběrnice jsou pouze jednosměrné (přenáší data pouze v jednom směru). Jiné sběrnice musí být realizovány jako obousměrné.

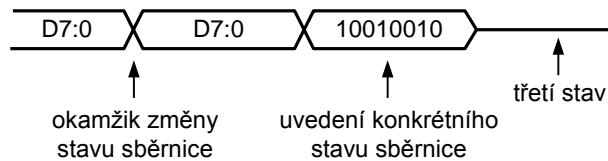
Pro kreslení sběrnice ve schématech používáme jednostranně (pro označení jednosměrné sběrnice) nebo oboustranně (pro označení obousměrné sběrnice) orientovanou šipku. Počet bitů sběrnice pak uvádíme pod symbolem přeškrtnutí. Viz obr. 1.9.



Obr. 1.9 Značení sběrnic ve schématech

Připomeňme, že pro označení aktivity signálu používáme v případě signálu aktivního v log. 1 symbol ve tvaru X a v případě signálu aktivního v log. 0 symbol ve tvaru  $\bar{X}$  případně /X (čteme X non).

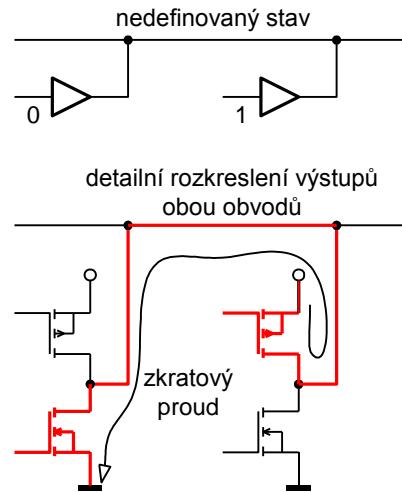
Při kreslení stavu sběrnice v časových digramech nerozkreslujeme obvykle jednotlivé bity, ale používáme „rámeček“, který sdružuje jednotlivé vodiče. Obecný stav sběrnice vyznačujeme názvem krajních vodičů. Změna stavu je vyznačena překřížením. Konkrétní stav sběrnice uvádíme číselně. Třetí stav (viz níže) vyznačujeme rovnou čarou procházející v polovině výšky rámečku. Viz obr. 1.10.



Obr. 1.10 Značení sběrnice v časových diagramech

### Třístavová (3stavová) logika

Na obr. 1.11 je ukázán modelový případ, pro který je vhodné využít možností poskytovaných třístavovou logikou. Je zde nakreslen jeden vodič sběrnice, na který jsou napojeny výstupy dvou obvodů.

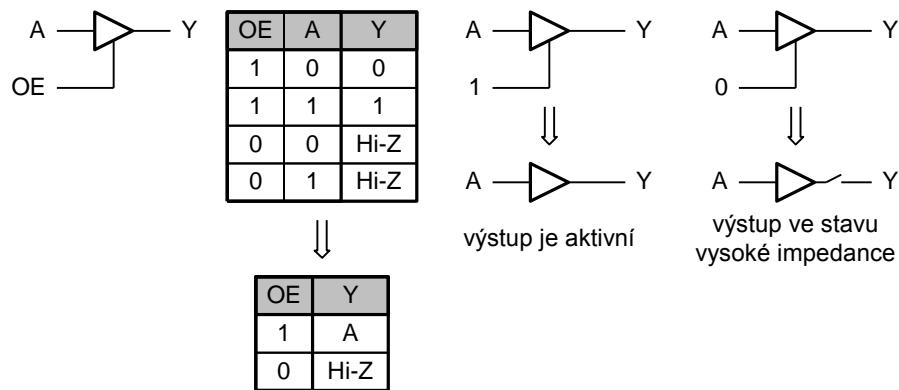


Obr. 1.11 Problém vzájemného propojení výstupů několika obvodů

Je zřejmé, že při použití klasického výstupu může nastat situace, kdy jeden výstup je buzen do log. 0 a jiný do log. 1. Výstupy začne procházet zkratový proud. Tato situace nemusí nutně znamenat poškození obvodů (výstupní odpor je vždy omezen), ale představuje také riziko ne definované logické úrovně. Každopádně je takový stav nepřijatelný.

Připomeňme, že v číslicové technice rozlišujeme stavy 0 a 1. Zmíněný třetí stav znamená, že se signál zcela odpojí. Proto se třetí stav také často označuje jako **stav vysoké impedance** (Hi-Z).

Uvedený problém lze tedy vyřešit tím, že použijeme speciální tzv. **třístavový budič**. Tento budič musí mít navíc jeden řídicí signál, který povoluje nebo blokuje činnost výstupu. Označujeme jej jako **OE** (Output Enable – povolení výstupu).



Obr. 1.12 Výklad funkce třístavového výstupu

Na obr. 1.12 je vlevo uvedena schématická značka třístavového budiče. Následuje pravdivostní tabulka a výklad funkce výstupu pro případ OE = 1 (výstup je aktivní) a OE = 0 (výstup je ve třetím stavu, tedy neaktivní).

## 1.5 Paměť

**Paměť** je obvod, který dokáže uchovávat informace v číslicové formě.

**Bit** je nejmenší informační jednotka, která je schopna rozlišit dva stavy: log. 1 nebo log. 0. Z toho také plyne, že pro kódování informací na úrovni hardwaru je používána dvojková soustava.

Většinou není paměť přístupná přímo po bitech. **Paměťové místo (položka paměti)** je část paměti, která je přímo přístupná. Je to část paměti, do které lze uložit definované množství informace. Významově nejvyšší bit se označuje zkratkou **MSB** (Most Significant Bit) a významově nejnižší pak zkratkou **LSB** (Least Significant Bit). Viz obr. 1.13.

Do paměti lze uložit rozličné údaje (čísla, text, instrukce programu), data jsou však vnitřně vždy uložena jako čísla.

MSB	b <sub>n-1</sub>	b <sub>n-2</sub>	...	b <sub>2</sub>	b <sub>1</sub>	LSB	b <sub>0</sub>

Obr. 1.13 Paměťové místo

Pro označení některých skupin bitů se používají zvláštní termíny. Skupinu osmi bitů ( $n = 8$ ) označujeme jako **bajt** (byte, slabika), skupinu šestnácti bitů ( $n = 16$ ) označujeme jako **slovo** (word). Podobně lze používat označení dvojslovo (double word pro  $n = 32$ ) a čtyřslovo (quad word pro  $n = 64$ ).

Nejčastěji používanou velikostí paměťového místa je jeden bajt. Pro snazší orientaci se pro skupinu 4 bitů zavádí pojem **nibble**. Takže bajt lze rozdělit na dva nibble: horní a dolní, viz obr 1.14.

horní nibble				dolní nibble			
b <sub>7</sub>	b <sub>6</sub>	b <sub>5</sub>	b <sub>4</sub>	b <sub>3</sub>	b <sub>2</sub>	b <sub>1</sub>	b <sub>0</sub>

Obr. 1.14 Horní a dolní nibble v bajtu

Nibble má ještě jeden důležitý praktický význam. Jelikož je to skupina 4 bitů a platí  $2^4 = 16^1$ , jedná se o prostor pro uložení jedné číslice v šestnáctkové soustavě.

**Adresa** je pořadové číslo, které používáme k výběru jednoho paměťového místa. **Kapacita paměti** určuje množství informace, které lze do paměti uložit. Kapacita paměti je určena počtem použitelných adres.

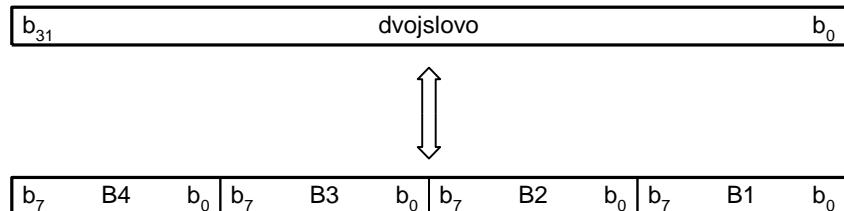
Adresa je kódovaná ve dvojkové soustavě, takže šíři adresy udáváme v bitech. Počet adres lze určit pomocí mocninné funkce  $2^N$ . Jednotlivá paměťová místa mají tedy adresy: 0, 1, 2, ...,  $2^N-1$ .

Pro paměť pracující s 10bitovou adresou platí, že obsahuje  $2^{10}$  tedy 1024 buněk (adresy jsou: 0, 1, 2, ..., 1023). Číslo 1024 označujeme jako K, což je analogie symbolu k – kilo. Pozor! Symbol k představuje hodnotu **1000**. Kdežto **K** odpovídá hodnotě **1024**. Důvod této odchylky spočívá v použité dvojkové soustavě.

**Organizace paměti** určuje nejmenší přímo přístupnou (adresovatelnou) jednotku paměti. Údaj o organizaci paměti uvádíme nejčastěji ve formě počtu adres násobených šířkou položky. Například 128 K × 8 (128 K položek šířky 8 bitů).

## Uložení dat v paměti

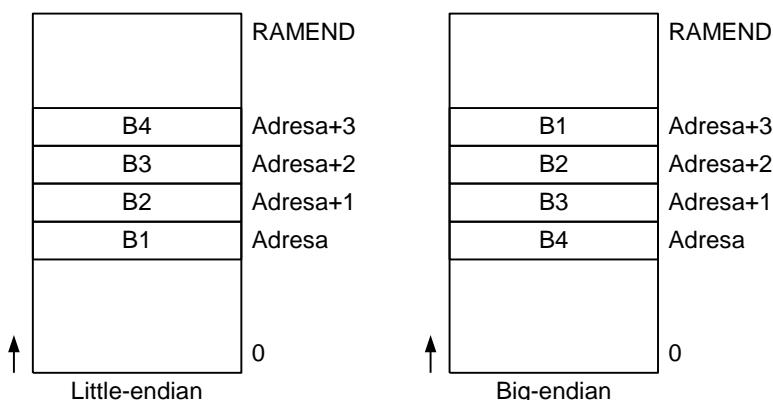
Při ukládání údajů, jejichž velikost je větší než odpovídá paměťovému místu, musí být jednotlivé části údaje uloženy na po sobě jdoucích adresách. Uvažujeme-li údaj velikosti dvojslova (32 bitů), musí se v případě paměťového místa velikosti jednoho bajtu (8 bitů) uložit do paměti na 4 po sobě jdoucí adresy. Viz obr. 1.15.



Obr. 1.15 Představa rozložení dvojslova na 4 bajty

Historicky byly vytvořeny dvě verze uložení vícebajtových údajů v paměti:

- **Little-endian** (používají procesory Intel, AVR a mnoho dalších) ukládá vícebajtové číslo tak, že nejméně významné bajty jsou na nižších adresách a nejvýznamnější bajt je na nejvyšší adrese.
- **Big-endian** (například procesory SPARC) ukládá vícebajtové číslo tak, že nejméně významné bajty jsou na vyšších adresách a nejvýznamnější bajt je na nejnižší adrese.



Obr. 1.16 Varianty uložení vícebajtového čísla v paměti

## Dělení paměti

Existuje více variant pamětí, níže si uvedeme a vysvětlíme termíny a zkratky, které se pro jejich označování používají.

**Volatilní** (prchavá) paměť se vyznačuje tím, že pro udržení uložené informace je nutná přítomnost napájecího napětí. Po zapnutí napájení je obsah paměti ne definovaný. Uložené údaje budou po odpojení napájení ztraceny.

**Nonvolatilní** (neprchavá) paměť udržuje údaje i po odpojení napájení.

Je-li paměť určena pouze pro čtení (tedy její obsah není možné normálním způsobem změnit), označujeme ji zkratkou **ROM** (Read-Only Memory). Takový typ paměti je pochopitelně **nonvolatilní**. Variantami jsou:

- **PROM** (Programmable ROM) – paměť lze jednorázově a nevratně naprogramovat (uložit do ní data) ve speciálním programátoru.
- **EPROM** (Erasable PROM) – paměť lze přeprogramovat opakovaně, před novým programováním se musí obsah vymazat působením ultrafialového záření.

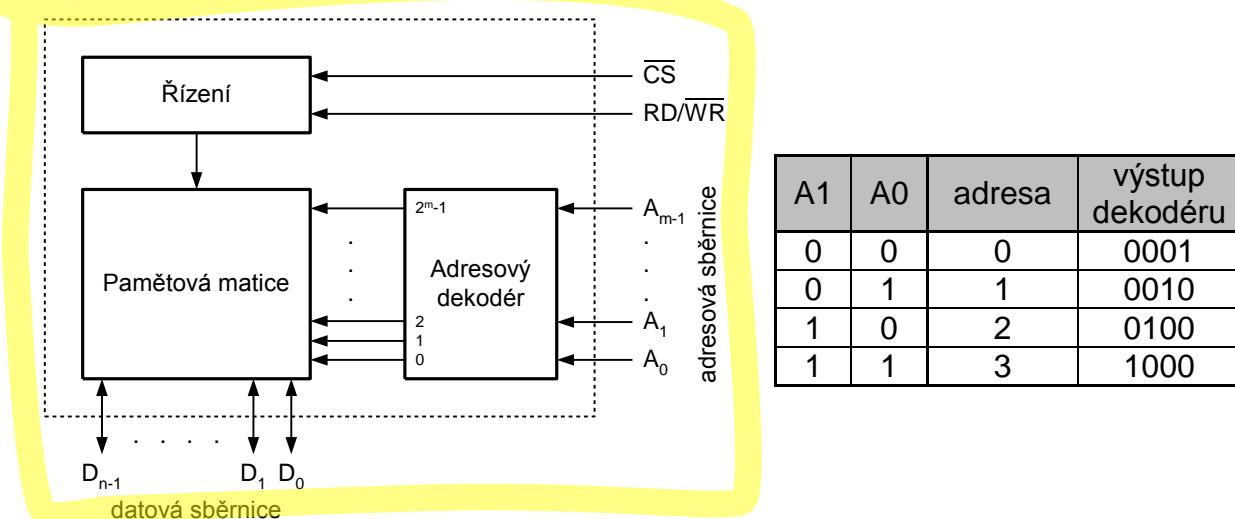
- **EEPROM** též **E<sup>2</sup>PROM** (Electrically EPROM) – paměť lze přeprogramovat elektricky, tedy přímo v systému, ve kterém je zapojena!
- **Flash** je speciálním případem EEPROM. Zásadní rozdíl je ve způsobu mazání paměti. Klasická EEPROM podporuje mazání jednotlivých bajtů. Tedy při přepisu údajů v paměti stačí smazat příslušný bajt a zapsat novou hodnotu. Flash dovoluje smazat pouze celý blok paměti, výrobně je jednodušší. Paměť Flash se tedy většinou používá jako paměť programu. Paměť EEPROM je vhodná pro uložení dat.

Paměti pro uložení dat jsou typu **RWM** (Read-Write Memory). V některých případech dochází k určitému „posunu“ termínů a záměně se zkratkou **RAM** (Random-Access Memory). RAM označuje paměť s libovolným přístupem. Přístup (vybavovací doba) k libovolnému paměťovému místu je stále stejná nezávisle na jeho adrese. Variantami **volatilních** pamětí jsou:

- **SRAM** (Static RAM) – paměťová buňka je tvořena klopným obvodem. Pro jeho realizaci je třeba typicky 6 tranzistorů. Tyto paměti jsou extrémně rychlé ale z důvodu složitější integrace nákladné. Používají pro realizaci paměti typu Cache a pro vytvoření registrů procesoru.
- **DRAM** (Dynamic RAM) – paměťová buňka je tvořena výběrovým tranzistorem a paměťovým kondenzátorem (vytvořeným opět pomocí MOS tranzistoru). Tyto paměťové buňky se používají především pro vytváření pamětí vyšších kapacit. Náboj kondenzátoru se po čase vybije a je nutné zajistit tzv. refresh. Přístupové doby jsou delší než u architektury SRAM.

### Princip činnosti RAM

Blokové schéma paměti RAM je uvedeno na obr. 1.17.



Obr. 1.17 Blokové schéma paměti RAM a pravidlostní tabulka adresového dekodéru pro 2 bity

**Paměťová matice** představuje vlastní paměťový blok. Tento blok má kapacitu  $2^m$  paměťových míst v šíři  $2^n$  bitů. Zprava vstupují adresové vodiče pro výběr paměťového místa, se kterým se má pracovat. Shora vstupují řídicí signály, které určují, zda bude prováděn zápis nebo čtení. Zespodu je pak napojena obousměrná **datová sběrnice**, kterou se přivádí zapisovaná data nebo se jedná o výstup přečtených dat.

**Adresová sběrnice** přivádí žádanou adresu kódovanou v binárním kódu na **adresový dekodér**. Jedná se o dekodér typu 1 z N. Tedy každé vstupní

kombinaci odpovídá aktivace právě jednoho adresního vodiče. Tím se tedy zajistí jednoznačný výběr paměťového místa, se kterým se pracuje.

Řídicí signál  $\overline{CS}$  – **Chip Select** slouží pro výběr paměťového čipu. Ve stavu  $\overline{CS}=1$  je čip deaktivován a nereaguje na žádné další povely. Datová sběrnice je ve třetím stavu. Ve stavu  $\overline{CS}=0$  je paměťový čip připraven k činnosti.

Řídicí signál  $\overline{RD/WR}$  – **Read/Write** slouží pro výběr prováděné operace. V případě  $\overline{RD/WR}=1$  se jedná o čtení dříve uložených dat z paměti, datová sběrnice bude nyní pracovat ve výstupním režimu. V případě  $\overline{RD/WR}=0$  se jedná o zápis dat. Datová sběrnice je nastavena do vstupního režimu.

### Komunikace s RAM

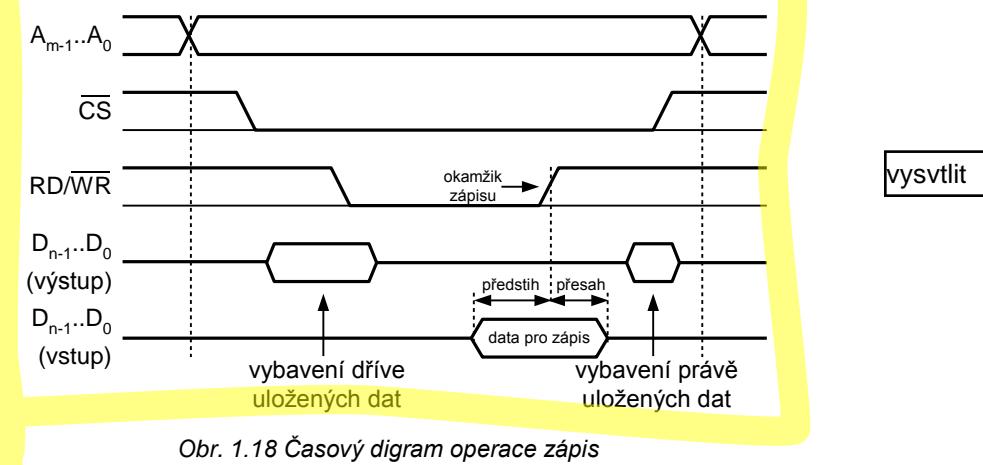
Časový diagram operace zápisu je uveden formou obr. 1.18. Po celou dobu operace musí být ustálen stav adresové sběrnice, adresový dekodér má připravenu adresu paměťového místa, se kterým chceme pracovat.

Signálem  $\overline{CS}=0$  aktivujeme paměť. Na začátku je signál řízení zápisu neaktivní ( $\overline{RD/WR}=1$ ), tedy paměť je připravena přejít do výstupního režimu a vybavit na datové sběrnici údaj, který je aktuálně v paměti uložen.

V okamžiku aktivace zápisu ( $\overline{RD/WR}=0$ ) přechází datová sběrnice do třetího stavu. V této chvíli může vnější obvod vložit na datovou sběrnici data, která se mají uložit do paměti. Tato data musí být ustálena s jistým předstihem a zůstat ustálena s jistým přesahem po náběžné hraně signálu  $\overline{RD/WR}$ . Tato hrana definuje konečný stav dat pro zápis.

V koncové části časového diagramu pak vidíme, že se řídicí signál zápisu vrací do režimu čtení ( $\overline{RD/WR}=1$ ), tedy posledně zapsaná data se na chvíli vybaví z paměti a objeví se na datové sběrnici, která je nyní ve výstupním směru.

Jakmile je paměťový čip deaktivován ( $\overline{CS}=1$ ), přechází datová sběrnice do třetího stavu.



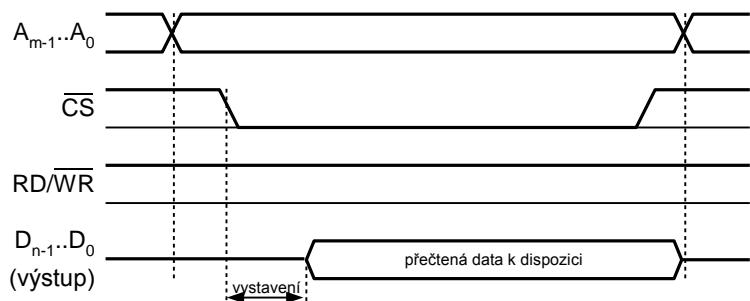
Obr. 1.18 Časový diagram operace zápis

Časový diagram operace čtení je uveden formou obr. 1.19.

Abychom zabránili přepisu dříve uložených údajů, nastavíme nejdříve řídicí signál do režimu čtení  $\overline{RD/WR}=1$ . Signálem  $\overline{CS}=0$  aktivujeme paměť.

Jakmile je údaj vyhledán v paměti, přechází datová sběrnice do výstupního směru a vystavuje přečtená data.

Po deaktivaci paměťového čipu ( $\overline{CS}=1$ ), přechází datová sběrnice do třetího stavu.



Obr. 1.19 Časový digram operace čtení

## 2 Celá čísla se znaménkem

V této kapitole vysvětlíme jednotlivé možnosti reprezentace celých čísel se znaménkem a realizaci základních aritmetických operací.

### 2.1 Reprezentace celých čísel se znaménkem

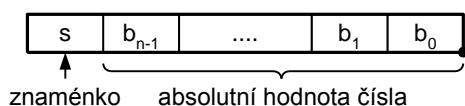
Polyadická soustava neumožňuje zobrazit čísla se znaménkem, proto je nutno použít kód. Existuje několik variant reprezentace čísel se znaménkem.

#### Přímý kód (Signed magnitude)

Přímý kód používá řádovou mřížku dle obr. 2.1.

Nejvyšší bit je vyhrazen na **znaménko**,  $s = 0$  pro kladná čísla a  $s = 1$  pro záporná čísla. **Absolutní hodnota – velikost čísla** je určena zbývajícími bity.

Je zřejmé, že kladné hodnoty se zobrazují stejným způsobem jako čísla bez znaménka. Záporné hodnoty mají nejvyšší bit hodnoty 1 a spodní bity určují absolutní hodnotu čísla. Například číslo +1 v rozměru bajtu má kód 00000001. Číslo -1 má pak kód 10000001.



Obr. 2.1 Řádová mřížka pro přímý kód

Obraz čísla  $A$  v přímém kódu  $P(A)$  je dán:

pro  $A \geq 0$ :

$$P(A) = |A| = A$$

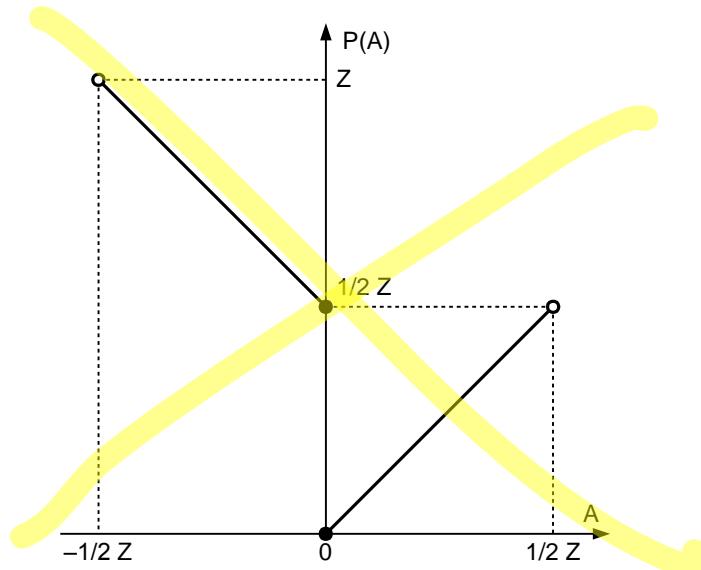
(2-1)

pro  $A \leq 0$ :

$$P(A) = 1/2 \cdot Z + |A|$$

kde:

- $A$  – číslo,
- $P(A)$  – obraz čísla v přímém kódu,
- $|A|$  – absolutní hodnota čísla,
- $Z$  – modul řádové mřížky.



Obr. 2.2 Obraz čísla  $A$  v přímém kódu  $P(A)$

Výhody:

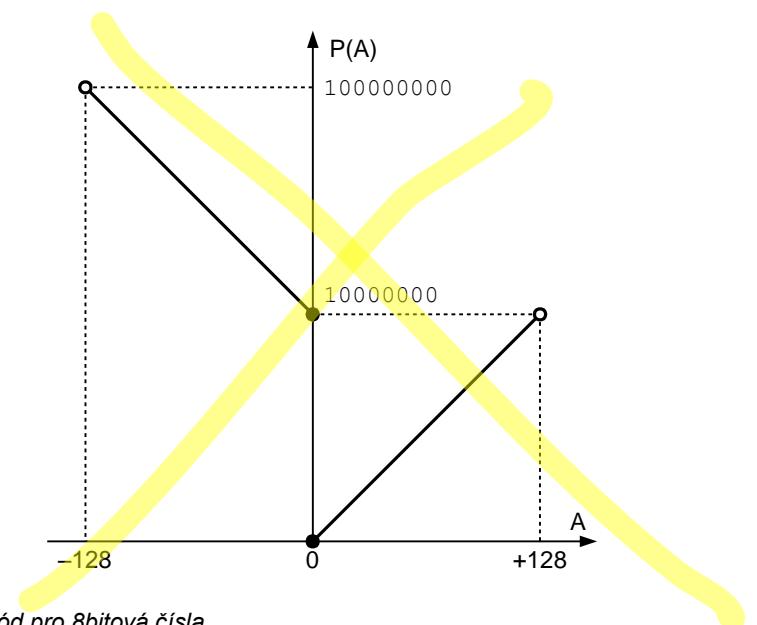
- rychlé zjištění znaménka a absolutní hodnoty čísla,
- rychlá změna znaménka.

Nevýhody:

- nula má dva kódy („kladná nula“: 00000000 a „záporná nula“: 10000000),
- komplikované algoritmy, například operace součtu nebo rozdílu vyžadují testovat stav znaménkového bitu a podle toho upravit výpočet,
- kód je nespojitý a nemonotónní (pro kladné hodnoty je rostoucí, pro záporné hodnoty klesací).

Pro lepší přehled uvedeme konkrétní příklad pro velikost řádové mřížky  $L = 8$ . Pro reprezentaci absolutní hodnoty je tedy k dispozici 7 bitů, rozsah absolutní hodnoty je 0 až  $2^7 - 1$ , tedy 0 až 127. Řádová mřížka velikosti jednoho bajtu tedy dovoluje zobrazit čísla se znaménkem v rozsahu  $-127$  až  $+127$ .

A	P(A)
-127	11111111
-126	11111110
...	.
...	.
...	.
-1	10000001
-0	10000000
+0	00000000
+1	00000001
...	.
...	.
...	.
+126	01111110
+127	01111111



Obr. 2.3 Přímý kód pro 8bitová čísla

### Doplňkový kód

V doplňkovém kódu indikuje nejvyšší bit opět znaménko čísla. Ve shodě s přímým kódem platí:  $s = 0$  pro kladná čísla a  $s = 1$  pro záporná čísla.

Kladná čísla jsou zobrazena normálním způsobem. Záporná čísla jsou zobrazena v tzv. dvojkovém doplňku.

**Jednotkový doplněk** (one's complement) představuje prostou negaci všech bitů. Tedy hodnota bitu se změní z 0 na 1 a obráceně. Tuto operaci označujeme symbolem **NOT** nebo **COM**.

**Dvojkový doplněk** (two's complement) představuje jednotkový doplněk zvýšený o 1. Tuto operaci označujeme symbolem **NEG**.

1. Příklad: Stanovme dvojkový doplněk čísla 13 (00001101). Jednotkový doplněk má hodnotu 11110010, dvojkový doplněk lze stanovit takto:

$$\begin{array}{r} 11110010 \\ +00000001 \\ \hline 11110011 \end{array}$$

hodnotu 11110011, dvojkový doplněk lze stanovit takto: +00000001.

$$\begin{array}{r} 11110011 \\ +00000001 \\ \hline 11110010 \end{array}$$

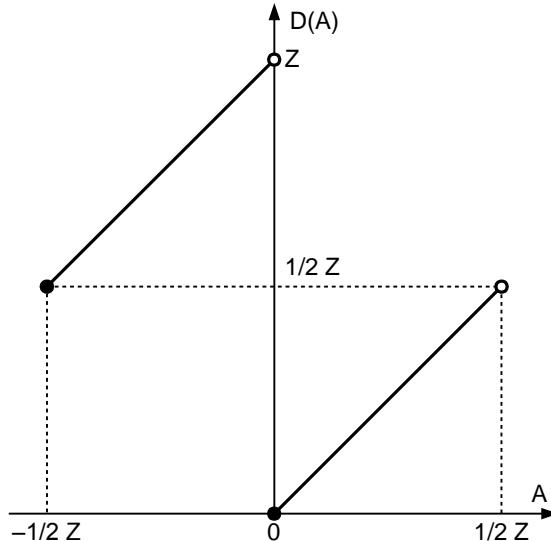
2. Příklad: Stanovme dvojkový doplněk čísla -13 (11110011). Jednotkový doplněk: 00001100, dvojkový doplněk: 00001101, tedy 13.

Z výše uvedených případů plyne, že dvojkový doplněk má vlastnost změny znaménka čísla.

Obraz čísla A v doplňkovém kódu D(A) je dán:

pro  $A \geq 0$ :  $D(A) = A$  (2-2)

pro  $A < 0$ :  $D(A) = Z + A = \text{NEG}(A)$



Obr. 2.4 Obraz čísla A v doplňkovém kódu D(A)

Výhody:

- rychlé zjištění znaménka,
- hodnotě nula odpovídá pouze jeden kód,
- pro sčítání a odčítání stačí jediná sčítáčka,
- je zachována komutativnost (výsledek součtu libovolného počtu čísel v doplňkovém kódu je stejný bez ohledu na jejich pořadí, i když dochází k přetečení),
- snadné rozšiřování a zkracování řádové mřížky (viz níže).
- kód je sice nespojitý, ale monotónní (rostoucí posloupnost).

Nevýhody:

- při přetečení může vzniknout neplatný výsledek, tento stav musí ALU indikovat zvláštním příznakem. Uvažujme například:  $127+1$ , tedy  $\begin{array}{r} +00000001 \\ 10000000 \end{array}$ . Výsledek je evidentně záporný a sice  $-128$ .

Rozšiřování řádové mřížky (znaménkové rozšíření)

Postup: Do přidávaných bitů zkopírujeme hodnotu nejvyššího (znaménkového) bitu rozšiřovaného čísla.

## 1. Příklad:

Číselná hodnota +5 v různé šíři:

Šíře	Kód	Číslo bez znaménka
8 bitů	0000 0101	5
16 bitů	0000 0000 0000 0101	5
32 bitů	0000 0000 0000 0000 0000 0000 0101	5

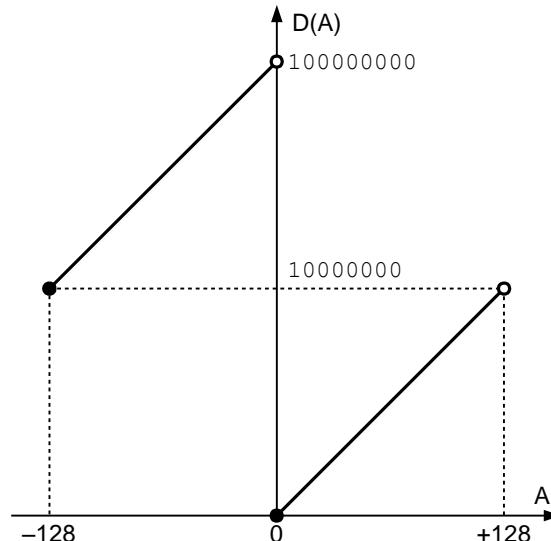
## 2. Příklad:

Číselná hodnota -5 v různé šíři:

Šíře	Kód	Číslo bez znaménka
8 bitů	1111 1011	251
16 bitů	1111 1111 1111 1011	65 531
32 bitů	1111 1111 1111 1111 1111 1111 1011	4 294 967 291

Pro lepší přehled uvedeme konkrétní příklad pro velikost řádové mřížky  $L = 8$ . Řádová mřížka velikosti jednoho bajtu tedy dovoluje zobrazit čísla se znaménkem v rozsahu -128 až +127.

A	D(A)
-128	10000000
-127	10000001
...	.
...	.
...	.
-1	11111111
0	00000000
+1	00000001
...	.
...	.
...	.
+126	01111110
+127	01111111



Obr. 2.5 Doplňkový kód pro 8bitová čísla

## Inverzní kód

Inverzní kód je podobný doplňkovému kódu.

Liší se kódováním záporných čísel. Záporné hodnoty jsou reprezentovány inverzí bitů, tedy jednotkovým doplňkem.

Tento kód má opět dvě reprezentace čísla 0. Tento kód nemá žádné zvláštní výhody, jeho použití není proto příliš časté.

## Kód s posunutou nulou (aditivní kód)

Poslední používanou možností kódování čísel se znaménkem je k číslu připočítat nějakou známou konstantu.

Obraz čísla A v posunutém kódu E(A) je dán (K je posunutí neboli offset):

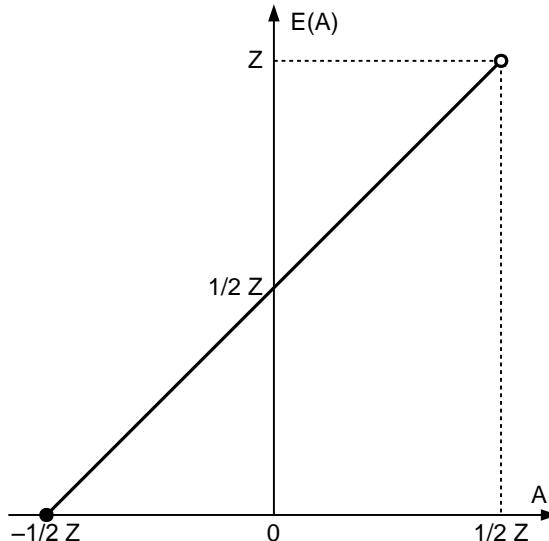
$$E(A) = A + K \quad (2-3)$$

Offset se obvykle volí v polovině rozsahu, tedy:  $K = 1/2 \cdot Z$ . Pro tento případ platí:

Nejmenší záporné číslo pak odpovídá  $E(A) = 0$ . Nula se nachází v polovině rozsahu  $E(0) = 1/2 \cdot Z$ .

Nejvyšší bit určuje znaménko čísla. Pro kladná čísla platí  $s = 1$ , pro záporná pak  $s = 0$  (tedy obráceně než v předchozích případech).

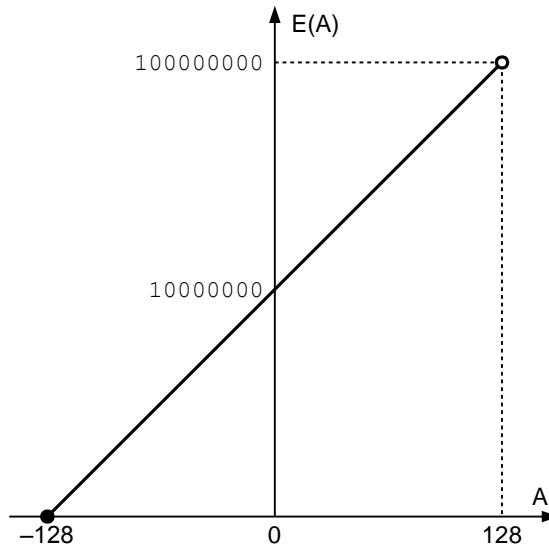
Doplňkový a posunutý kód lze vzájemně snadno převádět, když se liší pouze hodnotou nejvyššího bitu.



Obr. 2.6 Obraz čísla A v posunutém kódu  $E(A)$

Pro lepší přehled uvedeme konkrétní příklad pro velikost řádové mřížky  $L = 8$ . Řádová mřížka velikosti jednoho bajtu tedy dovoluje zobrazit čísla se znaménkem v rozsahu  $-128$  až  $+127$ .

A	$E(A)$
-128	00000000
-127	00000001
...	.
...	.
...	.
-1	01111111
0	10000000
+1	10000001
...	.
...	.
...	.
+126	11111110
+127	11111111



Obr. 2.7 Posunutý kód pro 8bitová čísla

Výhody:

- rychlé zjištění znaménka,
- hodnotě nula odpovídá pouze jeden kód,
- větší číselná hodnota má obraz vyšší hodnoty (kód je proto vhodný pro rychlé porovnávání čísel),
- používá se jako exponent u čísel v plovoucí řádové čárce,
- kód je spojitý a monotónní (rostoucí posloupnost).

Nevýhody:

- kladná čísla se liší od bezznaménkové reprezentace čísel. Operace sčítání nepotřebuje úpravy, ale pro operaci násobení je nutné od operandů odečíst známou konstantu.

## 2.2 Základní aritmetické operace

Níže budeme diskutovat vlastnosti základních aritmetických operací pro celá čísla. Budeme tedy uvažovat řádovou mřížku velikosti  $L = n$ .

### Sčítání a odčítání čísel bez znaménka

#### a) Sčítání $S = X+Y$

Výsledek součtu dvou čísel šíře  $n$  bitů má maximální šířku  $n+1$  bitů. Rozšíření šíře výstupního operandu souvisí s množným přetečením.

Omezíme-li výsledek na původní řádovou mřížku ( $n$  bitů), probíhá součet modulo  $Z$  (tedy  $2^n$ ). Viz obr. 2.8.

Na obr. 2.8 je uveden tzv. **kruh modulo aritmetiky** pro  $n = 3$ . Přesáhne-li součet maximální hodnotu 7, pokračují kódy další otáčkou v kruhu. Tedy hodnotě 8 odpovídá 000, hodnotě 9 odpovídá 001, atd.

Výsledek po přetečení „oříznutý“ na rozsah původní řádové mřížky je chybný. ALU indikuje tuto skutečnost pomocí příznaku C (Carry – přetečení). Tento příznak lze rovněž chápat jako nejvyšší bit celého výsledku.

Pravdivostní tabulka součtů pro  $n = 2$

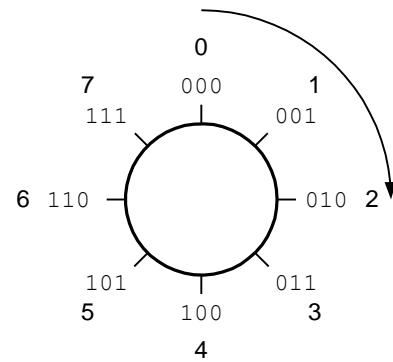
X	Y	S	C	C:S
00	00	00	0	0
00	01	01	0	1
00	10	10	0	2
00	11	11	0	3
01	00	01	0	1
01	01	10	0	2
01	10	11	0	3
01	11	00	1	4
10	00	10	0	2
10	01	11	0	3
10	10	00	1	4
10	11	01	1	5
11	00	11	0	3
11	01	00	1	4
11	10	01	1	5
11	11	10	1	6

#### b) Odčítání $R = X-Y$

Výsledek rozdílu dvou čísel šíře  $n$  bitů má maximální šířku  $n+1$  bitů. Rozšíření šíře výstupního operandu souvisí s možnou výpůjčkou.

Pro případ  $X < Y$  je výsledek chybný. ALU indikuje tuto skutečnost pomocí příznaku B (Borrow – výpůjčka).

V procesorech se místo dvou podobných příznaků B (Borrow) a C (Carry) používá pouze jeden příznak C. Je-li nastaven po operaci součtu, došlo k přetečení. Je-li nastaven po operaci rozdílu, došlo k podtečení.



Obr. 2.8 Kruh modulo aritmetiky pro  $n = 3$

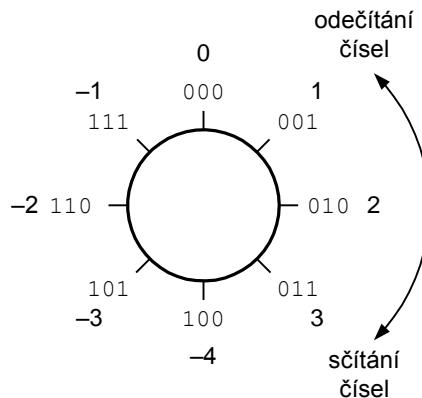
$$\begin{array}{r}
 146 \\
 -3 \\
 \hline
 143
 \end{array}
 \quad
 \begin{array}{r}
 10010010 \\
 -00000011 \\
 \hline
 10001111
 \end{array}
 \quad
 \begin{array}{r}
 3 \\
 -146 \\
 \hline
 113
 \end{array}
 \quad
 \begin{array}{r}
 00000011 \\
 -10010010 \\
 \hline
 01110001
 \end{array}$$

Obr. 2.9 Příklady výpočtu rozdílu čísel bez znaménka

### Sčítání a odčítání čísel v doplňkovém kódu

Sčítání a odčítání čísel v doplňkovém kódu probíhá podobně. V případě odečítání se hodnota menšítele ( $Y$ ) převede do dvojkového doplňku ( $-Y$ ) a seče se s menšencem ( $X$ ). Tedy:  $X - Y = X + (-Y)$ .

Při sčítání se opět uplatní modulo aritmetika, pro  $n = 3$  je kruh modulo aritmetiky uveden formou obr. 2.10.



Obr. 2.10 Kruh modulo aritmetiky dvojkového doplňku pro  $n = 3$

Pro rozeznání přetečení nelze nyní uvažovat příznak  $C$ , tak jak tomu bylo při použití přímého kódu. Jeden ze způsobů stanovení příznaku **V – přetečení aritmetiky dvojkového doplňku** je vztah:

$$V = C_{n-1 \rightarrow n} \oplus C_{n-2 \rightarrow n-1} \quad (2-4)$$

kde:

- $V$  – příznak přetečení dvojkového doplňku,
- $C_{n-1 \rightarrow n}$  – příznak přetečení z řádu  $n-1$  do řádu  $n$  (z nejvyššího řádu),
- $C_{n-2 \rightarrow n-1}$  – příznak přetečení z řádu  $n-2$  do řádu  $n-1$  (z předposledního řádu),
- $\oplus$  – operace xor (výlučný logický součet:  $0 \oplus 0 = 0$ ,  $0 \oplus 1 = 1$ ,  $1 \oplus 0 = 1$ ,  $1 \oplus 1 = 0$ ).

Níže je uvedeno několik příkladů používání aritmetiky dvojkového doplňku při sčítání čísel v šířce  $n = 3$ . Pro stanovení příznaků přetečení si uvědomíme, že nyní je první příznak  $C_{2 \rightarrow 3}$  a druhý  $C_{1 \rightarrow 2}$ .

#### 1. Příklad: Součet čísel s různými znaménky (nepřeteče).

$$\begin{array}{r}
 \begin{array}{c}
 2 & 1 & 0 \\
 \boxed{1} & \boxed{1} & \boxed{1} \\
 \hline
 \end{array} \\
 -1 \\
 +2 \\
 \hline
 +1 \quad \boxed{\pm} \quad \boxed{0} \quad \boxed{0} \quad \boxed{1}
 \end{array}
 \quad
 \begin{array}{l}
 C_{n-2 \rightarrow n-1} = 1 \\
 C_{n-1 \rightarrow n} = 1 \\
 V = 0
 \end{array}$$

Součet čísel s různými znaménky nemůže přeteknout. Uvažujme příklad:  $-1+2 = 1$ . Oba dílčí příznaky přetečení se nastaví, výsledek je ale v pořádku ( $V = 0$ ). Výstupní přetečení se neuvažuje.

2. Příklad: Součet čísel se stejným znaménkem (výsledek nepřekročí rozsah).

$$\begin{array}{r}
 +1 \quad \boxed{\begin{smallmatrix} 2 & 1 & 0 \\ 0 & 0 & 1 \end{smallmatrix}} \\
 +2 \quad + \quad \boxed{\begin{smallmatrix} 0 & 1 & 0 \end{smallmatrix}} \\
 \hline
 +3 \quad \boxed{\begin{smallmatrix} 0 & 0 & 1 & 1 \end{smallmatrix}}
 \end{array}$$

$C_{n-2 \rightarrow n-1} = 0$   
 $C_{n-1 \rightarrow n} = 0$   
 $V = 0$

$$\begin{array}{r}
 -1 \quad \boxed{\begin{smallmatrix} 2 & 1 & 0 \\ 1 & 1 & 1 \end{smallmatrix}} \\
 -2 \quad + \quad \boxed{\begin{smallmatrix} 1 & 1 & 0 \end{smallmatrix}} \\
 \hline
 -3 \quad \boxed{\begin{smallmatrix} \pm & 1 & 0 & 1 \end{smallmatrix}}
 \end{array}$$

$C_{n-2 \rightarrow n-1} = 1$   
 $C_{n-1 \rightarrow n} = 1$   
 $V = 0$

Uvažujme příklad:  $1+2=3$ . Oba dílčí příznaky přetečení se nenastaví, výsledek je v pořádku ( $V=0$ ).

Uvažujme příklad:  $-1-2=-3$ . Oba dílčí příznaky přetečení se nastaví, výsledek je ale v pořádku ( $V=0$ ). Výstupní přetečení se neuvažuje.

3. Příklad: Součet čísel se stejným znaménkem (výsledek překročí rozsah).

$$\begin{array}{r}
 +1 \quad \boxed{\begin{smallmatrix} 2 & 1 & 0 \\ 0 & 0 & 1 \end{smallmatrix}} \\
 +3 \quad + \quad \boxed{\begin{smallmatrix} 0 & 1 & 1 \end{smallmatrix}} \\
 \hline
 -4 \quad \boxed{\begin{smallmatrix} 0 & 1 & 0 & 0 \end{smallmatrix}}
 \end{array}$$

$C_{n-2 \rightarrow n-1} = 1$   
 $C_{n-1 \rightarrow n} = 0$   
 $V = 1$

$$\begin{array}{r}
 -2 \quad \boxed{\begin{smallmatrix} 2 & 1 & 0 \\ 1 & 1 & 0 \end{smallmatrix}} \\
 -4 \quad + \quad \boxed{\begin{smallmatrix} 1 & 0 & 0 \end{smallmatrix}} \\
 \hline
 +2 \quad \boxed{\begin{smallmatrix} \pm & 0 & 1 & 0 \end{smallmatrix}}
 \end{array}$$

$C_{n-2 \rightarrow n-1} = 0$   
 $C_{n-1 \rightarrow n} = 1$   
 $V = 1$

Uvažujme příklad:  $1+3=?$ . Nastaví se pouze příznak přetečení z předposledního řádu, došlo k překročení rozsahu aritmetiky dvojkového doplňku ( $V=1$ ). Výsledek je chybný.

Uvažujme příklad:  $-2-4=?$ . Nastaví se pouze příznak přetečení z posledního řádu, došlo k překročení rozsahu aritmetiky dvojkového doplňku ( $V=1$ ). Výsledek je chybný.

### Násobení a dělení v přímém kódu

Přímý kód je vhodný na provádění operací násobení a dělení. Postup výpočtu je následující:

1. stanovíme absolutní hodnotu obou operandů X a Y (vynulováním nejvyššího bitu),
2. provedeme výpočet součinu nebo podílu:  $|X| * / |Y|$ ,
3. stanovíme znaménko výsledku:  $S(\text{výsledek}) = S(X) \oplus S(Y)$ ,
4. výsledek je dvojnásobně šíře než šíře původní absolutní hodnoty.

Příklad:

$$\begin{array}{r}
 \boxed{\begin{smallmatrix} 2 & 1 & 0 \\ 1 & 1 & 1 \end{smallmatrix}} \quad 7 \\
 * \quad \boxed{\begin{smallmatrix} 2 & 1 & 0 \\ 1 & 1 & 0 \end{smallmatrix}} \quad *6 \\
 \hline
 \boxed{\begin{smallmatrix} 0 & 0 & 0 \end{smallmatrix}}
 \end{array}$$

$\boxed{\begin{smallmatrix} 1 & 1 & 1 \end{smallmatrix}}$   
 $\boxed{\begin{smallmatrix} 1 & 1 & 1 \end{smallmatrix}}$   
 $\hline$ 
 $\boxed{\begin{smallmatrix} 1 & 0 & 1 & 0 & 1 & 0 \end{smallmatrix}}$

42

### 3 Základní bloky CPU

V této kapitole nejdříve zopakujeme základní kombinační a sekvenční obvody. Dále se seznámíme s realizací sčítáčky, ALU a GPR.

#### 3.1 Základní kombinační a sekvenční obvody – opakování

##### Invertor

schematická značka	pravidlostní tabulka	logická rovnice						
	<table border="1"> <thead> <tr> <th>A</th> <th>Y</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> </tr> </tbody> </table>	A	Y	0	1	1	0	$Y = \bar{A}$
A	Y							
0	1							
1	0							

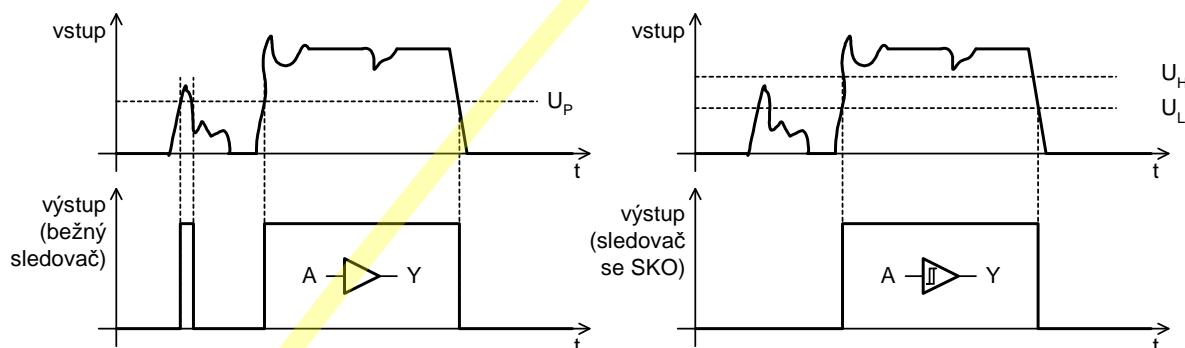
Obr. 3.1 Invertor (NOT)

**Buffer (budič, sledovač)** – výkonové posílení výstupu.

schematická značka	pravidlostní tabulka	logická rovnice						
	<table border="1"> <thead> <tr> <th>A</th> <th>Y</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> </tr> </tbody> </table>	A	Y	0	0	1	1	$Y = A$
A	Y							
0	0							
1	1							

Obr. 3.2 Buffer

**Budič se Schmittovým klopným obvodem na vstupu** – hystereze dovoluje redukovat vstupní rušení a snížit spotřebu obvodu při připojení vstupu na signál s pomalými náběžnými a sestupnými hranami.



Obr. 3.3 Buffer se Schmittovým klopným obvodem na vstupu

**Třístavový budič** – používá se pro propojování výstupů na sběrnicích.

schematická značka	pravidlostní tabulka	logická rovnice															
	<table border="1"> <thead> <tr> <th>OE</th> <th>A</th> <th>Y</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>Hi-Z</td> </tr> <tr> <td>0</td> <td>1</td> <td>Hi-Z</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </tbody> </table>	OE	A	Y	0	0	Hi-Z	0	1	Hi-Z	1	0	0	1	1	1	pro $OE = 0$ , $Y = \text{Hi-Z}$ pro $OE = 1$ , $Y = A$
OE	A	Y															
0	0	Hi-Z															
0	1	Hi-Z															
1	0	0															
1	1	1															

Obr. 3.4 Třístavový budič

**Hradlo AND** – lze použít pro řízení průchodu signálu. Bereme-li B jako řídicí signál, platí: pro  $B = 0$  je  $Y = 0$ , pro  $B = 1$  je  $Y = A$ .

schematická značka	pravidlostní tabulka	logická rovnice															
	<table border="1"> <thead> <tr> <th>B</th><th>A</th><th>Y</th></tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	B	A	Y	0	0	0	0	1	0	1	0	0	1	1	1	$Y = A \cdot B$
B	A	Y															
0	0	0															
0	1	0															
1	0	0															
1	1	1															

Obr. 3.5 Hradlo AND

**Hradlo NAND** – realizace libovolné logické funkce variantou „součet součinů“.

schematická značka	pravidlostní tabulka	logická rovnice															
	<table border="1"> <thead> <tr> <th>B</th><th>A</th><th>Y</th></tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	B	A	Y	0	0	1	0	1	1	1	0	1	1	1	0	$Y = \overline{A \cdot B}$
B	A	Y															
0	0	1															
0	1	1															
1	0	1															
1	1	0															

Obr. 3.6 Hradlo NAND

**Hradlo OR** – používá se například pro sloučení několika signálů.

schematická značka	pravidlostní tabulka	logická rovnice															
	<table border="1"> <thead> <tr> <th>B</th><th>A</th><th>Y</th></tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	B	A	Y	0	0	0	0	1	1	1	0	1	1	1	1	$Y = A + B$
B	A	Y															
0	0	0															
0	1	1															
1	0	1															
1	1	1															

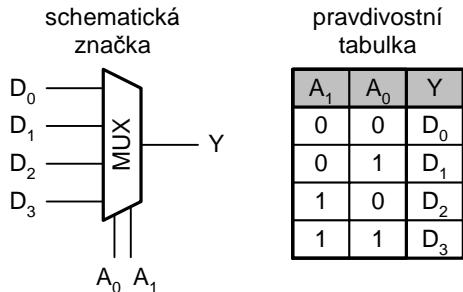
Obr. 3.7 Hradlo OR

**Hradlo XOR** – hradlo XOR nachází uplatnění také jako **komparátor** (pokud jsou stavy obou vstupů shodné, je výstup log. 0) nebo **řízený invertor** (pro  $B = 0$  platí  $Y = A$ , pro  $B = 1$  platí  $Y = \overline{A}$ ; tedy vstup B představuje řídicí vstup určující, zda signál ze vstupu A dorazí na výstup v přímé nebo negované podobě).

schematická značka	pravidlostní tabulka	logická rovnice															
	<table border="1"> <thead> <tr> <th>B</th><th>A</th><th>Y</th></tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	B	A	Y	0	0	0	0	1	1	1	0	1	1	1	0	$Y = A \oplus B = \overline{A}B + A\overline{B}$
B	A	Y															
0	0	0															
0	1	1															
1	0	1															
1	1	0															

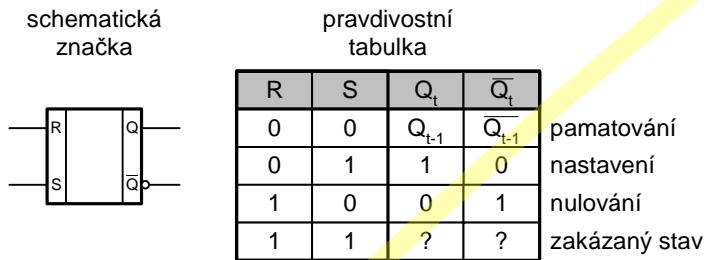
Obr. 3.8 Hradlo XOR

**Multiplexer** – elektronicky řízený přepínač, pomocí adresovacích vstupů (zde  $A_0, A_1$ ) se rozhodne, který datový vstup (zde  $D_0$  až  $D_3$ ) je přepnut na výstup. Výstup pak sleduje jeho stav.



Obr. 3.9 Multiplexer

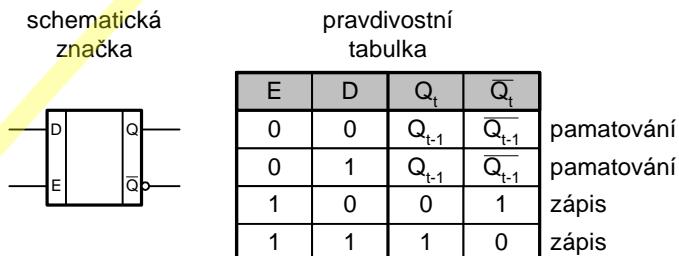
**Klopný obvod R-S** – řídicí vstupy: **S** – Set (nastavení) a **R** – Reset (nulování), výstupy: **Q** a  $\bar{Q}$ .



Obr. 3.10 Klopný obvod R-S

**Klopný obvod D řízený úrovní** – řídicí vstup: **E** – Enable (povolení zápisu), datový vstup: **D** – data (vstupní data), výstupy: **Q**.

**Citlivost na úroveň** znamená, že při **E = 1** reagují výstupy **Q** a  $\bar{Q}$  okamžitě na změnu stavu vstupu **D**. Lze říci, že cesta ze vstupu na výstup je stále průchozí.

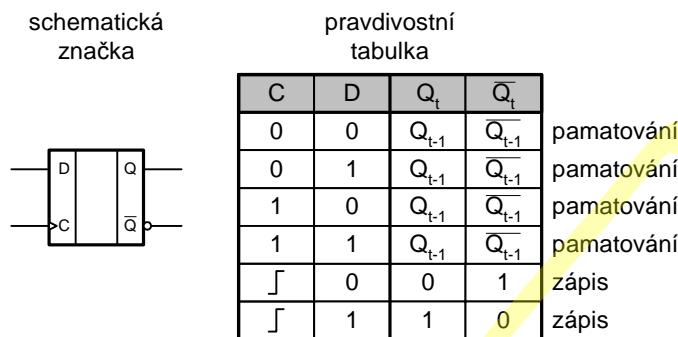


Obr. 3.11 Klopný obvod D řízený úrovní

**Klopný obvod D řízený hranou** – dvoutaktní spojení dvou klopných obvodů D řízených úrovní. Hodinový vstup: **C** – Clock, datový vstup: **D** – Data, výstupy: **Q** a  $\bar{Q}$ .

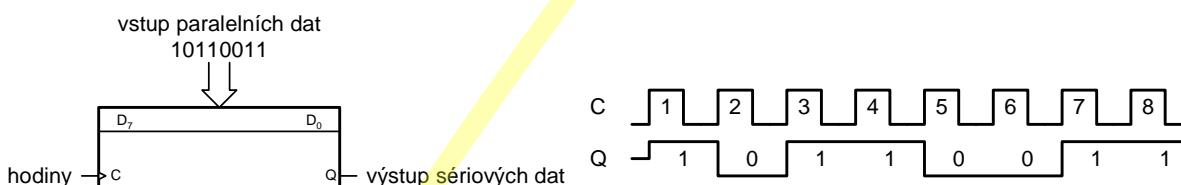
**Citlivost na hranu** hodinového signálu znamená, že okamžik vzorkování vstupu je velmi krátký a synchronizovaný s hodinovým signálem.

Sériovým spojením několika klopných obvodů D lze vytvářet posuvné registry, čítače nebo jiné složitější obvody.



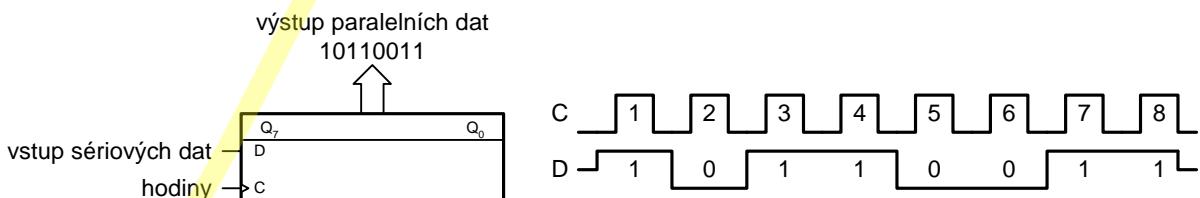
Obr. 3.12 Klopný obvod D řízený hranou

**Posuvný registr PISO** – vstupní data v jednom okamžiku zapíšeme v paralelní formě do klopních obvodů D a potom je pomocí hodinových impulzů lze po jednom bitu „vysouvat“ v sériové podobě. Lze využít pro sériové vysílání dat.



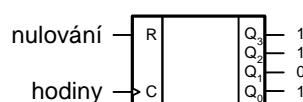
Obr. 3.13 Posuvný registr PISO

**Posuvný registr SIPO** – vstupní data přijímá sériově po jednom bitu a po příjmu všech bitů je poskytuje v paralelní podobě. Lze použít pro sériové přijímání dat.



Obr. 3.14 Posuvný registr SIPO

**Čítač** – výstupní stav odpovídá počtu vstupních hodinových impulzů. Pro čítač čítající nahoru, vede každý příchozí impulz na zvýšení obsahu o 1. Čítače se používají k registraci hodinových impulzů. Přeneseně mohou tedy čítat kmitočet nebo odměřovat časové intervaly.



Obr. 3.15 Čítač

## 3.2 Sčítačka (Adder)

Sčítačka je obvod provádějící sčítání čísel. Sčítačka se používá nejen v rámci ALU, objevuje i v jiných blocích procesoru.

Sčítačka obvykle pracuje s binární reprezentací čísel, protože při použití doplňkového kódu lze sčítačku snadno modifikovat na odčítačku. Jiné varianty čísel se znaménkem vyžadují mnohem komplikovanější realizaci sčítačky.

## 1bitová poloviční sčítačka (1-bit Half Adder)

Poloviční sčítačka sčítá dvě vstupní dvojková čísla **X** a **Y**. Výsledkem je součet **S** a přenos **C**. Přenos reprezentuje přetečení, které je třeba uvážit ve vyšším řádu součtu v případě sčítání vícebitových čísel. Výstupy **S** a **C** lze rovněž chápout jako 2bitový výsledek ve tvaru **C:S**.

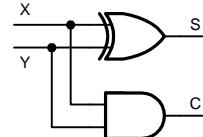
Tab. 3.1 Pravdivostní tabulka poloviční sčítačky

X	Y	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Logické rovnice:

$$S = \overline{XY} + X\bar{Y} = X \oplus Y$$

$$C = XY$$



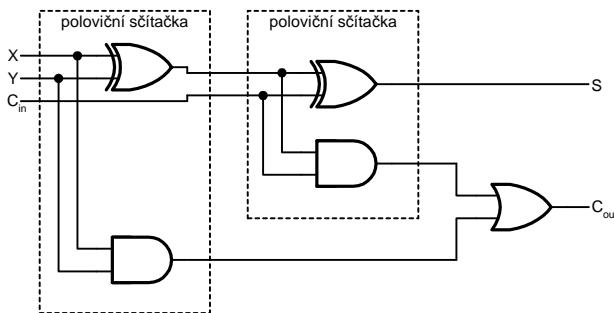
Obr. 3.16 Schéma zapojení 1bitové poloviční sčítačky

## 1bitová úplná sčítačka (1-bit Full Adder)

Úplná sčítačka uvažuje nejen přenos do vyššího řádu ( $C_{out}$ ), ale také přenos z nižšího řádu ( $C_{in}$ ). Na vstupu jsou tedy tři operandy: **X**, **Y** a **C<sub>in</sub>**. Výstup je opět ve formě součtu **S** a přenosu do vyššího řádu **C<sub>out</sub>**. Z obr. 3.17 je zřejmé, že úplná sčítačka vznikne spojením dvou polovičních sčítaček.

Tab. 3.2 Pravdivostní tabulka úplné sčítačky

C <sub>in</sub>	X	Y	S	C <sub>out</sub>
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

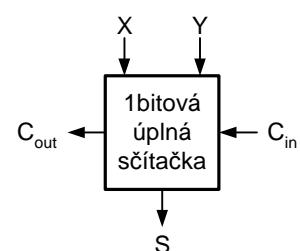


Obr. 3.17 Schéma zapojení 1bitové úplné sčítačky

Logické rovnice:

$$S = \overline{C_{in}} \overline{XY} + \overline{C_{in}} X\bar{Y} + C_{in} \overline{X}\bar{Y} + C_{in} XY = \\ = \overline{C_{in}} (\overline{XY} + X\bar{Y}) + C_{in} (\overline{X}\bar{Y} + XY) = \\ = C_{in} \oplus (X \oplus Y)$$

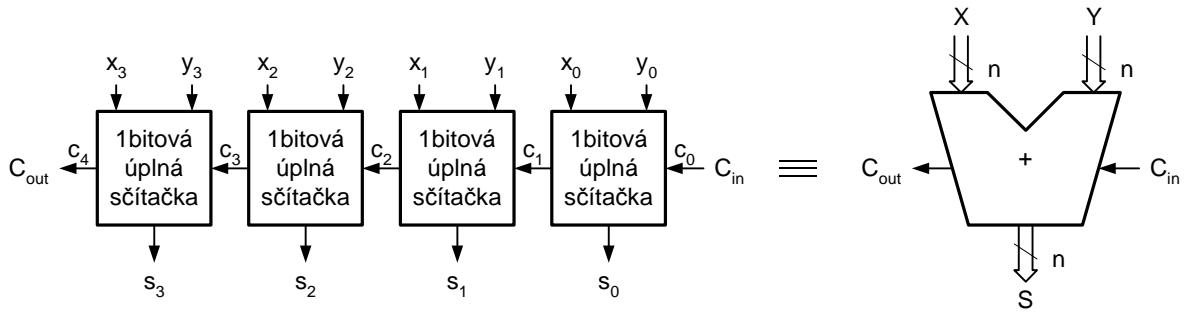
$$C_{out} = \overline{C_{in}} XY + C_{in} \overline{X}\bar{Y} + C_{in} X\bar{Y} + C_{in} XY = \\ = (\overline{C_{in}} + C_{in}) XY + C_{in} (\overline{X}\bar{Y} + X\bar{Y}) = \\ = XY + C_{in} (X \oplus Y)$$



Obr. 3.18 Bloková značka

## N-bitová sčítačka

Jednou z možností vytvoření n-bitové sčítačky je kaskádně zapojit n 1bitových úplných sčítaček. Tato varianta se označuje jako **sčítačka s kaskádním přenosem** (ripple-carry adder). Viz obr. 3.19.



Obr. 3.19 Blokové schéma 4bitové sčítačky s kaskádním přenosem, bloková značka  $n$ -bitové sčítačky

Hlavní nevýhodou uvedené architektury je skutečnost, že je výpočet v každém řádu závislý na výpočtech ve všech předchozích řádech. Vzniká tak znatelné zpoždění výpočtu.

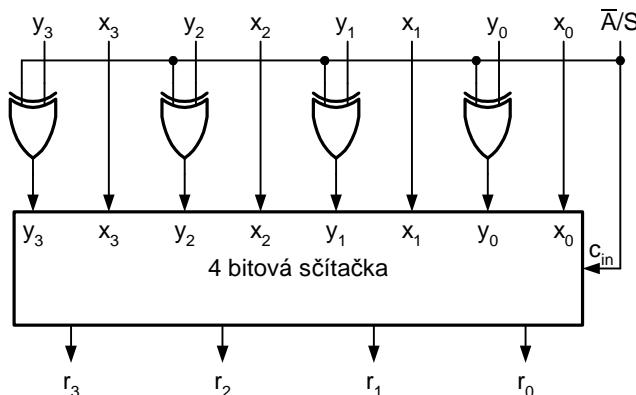
Pro zrychlení výpočtu součtu lze použít upravená zapojení označovaná jako **sčítačka se zrychlením přenosu** nebo **sčítačka s předvídaním přenosu**.

### Sčítačka/odčítačka pro čísla v doplňkovém kódu

Použití doplňkového kódu umožňuje snadnou realizaci kombinované sčítačky/odčítačky. Tedy obvodu, který podle stavu řídicího signálu stanoví součet nebo rozdíl.

Rozdíl dvou čísel  $X, Y$  lze totiž realizovat tak, že sčítáme  $X$  a druhý doplněk  $Y$ . Připomeňme také, že druhý doplněk lze stanovit jako negaci bitů  $Y$  doplněnou o přičtení 1. Přičtení 1 lze realizovat pomocí vstupu  $C_{\text{in}}$ . Pro řízenou negaci čísla použijeme hradlo XOR.

Výsledek těchto úvah je znázorněn na obr. 3.20.



Obr. 3.20 Blokové schéma sčítačky/odčítačky

Vstupní operandy jsou **X3:0, Y3:0**. Řídicí vstup  **$\bar{A}/S$**  určuje, zda bude vypočítán součet ( $\bar{A}/S = 0$ ) nebo rozdíl ( $\bar{A}/S = 1$ ). Výstup je **R3:0**.

Pokud platí  $\bar{A}/S = 0$ , je vstupní přenos 0 a operand  $Y$  vstupuje do sčítačky bez negace. Jedná se tedy skutečně o součet.

Pokud platí  $\bar{A}/S = 1$ , je vstupní přenos 1 a operand  $Y$  vstupuje do sčítačky negovaný (negace zvýšená o 1 odpovídá dvojkovému doplňku). Jedná se tedy skutečně o rozdíl.

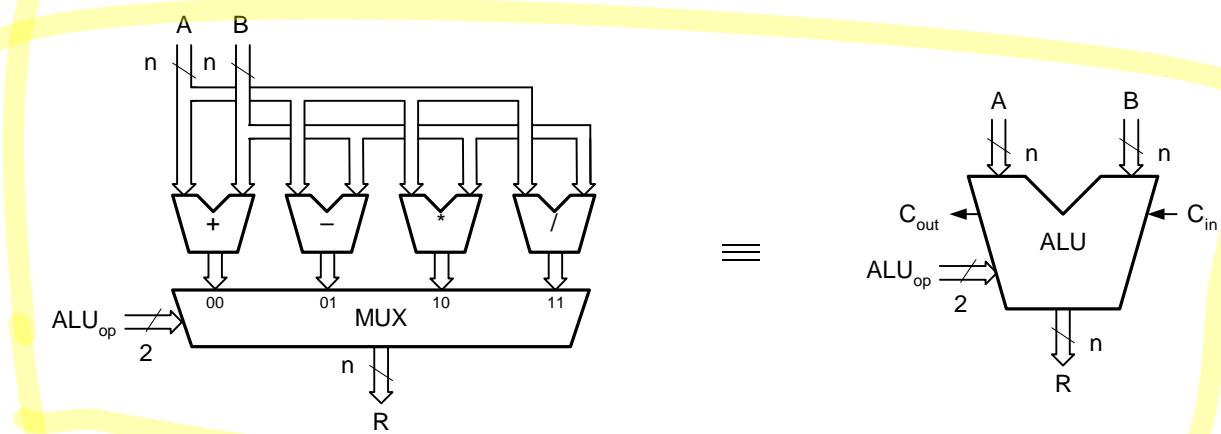
### 3.3 Aritmeticko-logická jednotka (ALU)

Aritmeticko-logická jednotka je kombinační logický obvod, který provádí aritmetické, logické a bitové operace. Do sady nejběžnějších operací patří:

- **aritmetické operace:**
  - součet ( $A+B$ ),
  - rozdíl ( $A-B$ ),
  - součin ( $A*B$ ),
  - podíl ( $A/B$ ),
  - inkrementace (INC),
  - dekrementace (DEC),
  - dvojkový doplněk (NEG),...
- **logické operace:**
  - negace (NOT resp. COM),
  - logický součin (AND),
  - logický součet (OR),
  - výlučný logický součet (XOR),...
- **bitové operace:**
  - logický posuv (LSL resp. LSR),
  - aritmetický posuv (ASR),
  - rotace (ROL resp. ROR),...

Njaký uvést jako příklad

Blokové schéma zjednodušené ALU je uvedeno na obr. 3.21. Tato ALU realizuje pouze aritmetické operace: součet, rozdíl, součin a podíl.



Obr. 3.21 Blokové schéma zjednodušené ALU

Z obr. 3.21 je zřejmé, že ALU je ředitelná 2bitovým signálem  $ALU_{op}$ . Tímto způsobem se vybere výsledek. Říkáme, že **ALU je programovatelná**. Operace ALU a jejich kódy uvádí tab. 3.3. Kód operace spolu s oběma operandy tvoří celou „instrukci“, kterou má ALU vykonat, viz obr. 3.22.

Tab. 3.3 Operace ALU

$ALU_{op}$	výsledek
00	$A+B$
01	$A-B$
10	$A*B$
11	$A/B$

operační kód	1. operand	2. operand
$ALU_{op}$	A	B

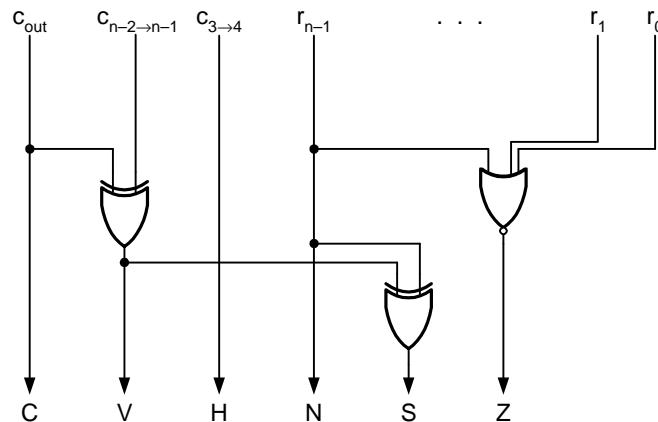
Obr. 3.22 Formát „instrukce“ pro ALU

**Šířka operandu**, který dokáže ALU přímo zpracovat, omezuje výpočetní výkon samotného procesoru. Procesory s 8bitovou ALU se tedy označují jako 8bitové, protože veškeré výpočty probíhají v šíři 8 bitů. Je-li třeba zpracovat „širší“ operandy, musí se tyto operace rozdělit do více dílčích kroků.

V průběhu výpočtu může dojít k přetečení výsledku, k přeplnění rozsahu čísel v doplňkovém kódu apod. Tyto stavy indikuje ALU pomocí tzv. **příznaků** (flags). Obvykle se jedná o tyto příznaky:

- **C** (Carry Flag) – přetečení z nejvyššího bitu výsledku,
- **V** (Two's Complement Overflow Flag) – přeplnění rozsahu čísel v doplňkovém kódu,
- **H** (Half Carry Flag) – přetečení dolního do horního nibble (používá aritmetika BCD),
- **N** (Negative Flag) – indikace záporného výsledku v aritmetice doplňkového kódu,
- **S** (Sign Flag) – indikace znaménka výsledku,
- **Z** (Zero Flag) – indikace nulového výsledku.

Jeden ze způsobu stanovení uvedených příznaků naznačuje obr. 3.23.



Obr. 3.23 Výpočet stavových příznaků

### Operace posuvů a rotací

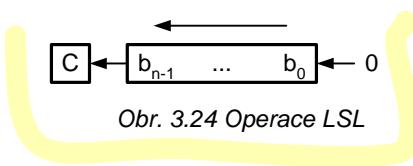
ALU realizuje kromě jiného operace posuvů a rotací, tyto operace jsou důležité pro implementaci mnoha algoritmů.

#### Logický posuv

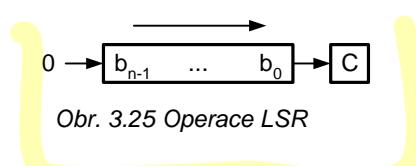
Logický posuv může probíhat doleva nebo doprava.

Operace **LSL** (Logical Shift Left) posouvá bity zprava doleva a na uvolněné místo zprava (do nejnižšího bitu) vloží 0. Nejvyšší bit, který vystoupí zleva, přepíše obsah příznaku C. Viz obr. 3.24.

Podobně operace **LSR** (Logical Shift Right) posouvá bity zleva doprava a na uvolněné místo (do nejvyššího bitu) vloží 0. Nejnižší bit, který vystoupí zprava, přepíše stav příznaku C. Viz obr. 3.25.



Obr. 3.24 Operace LSL



Obr. 3.25 Operace LSR

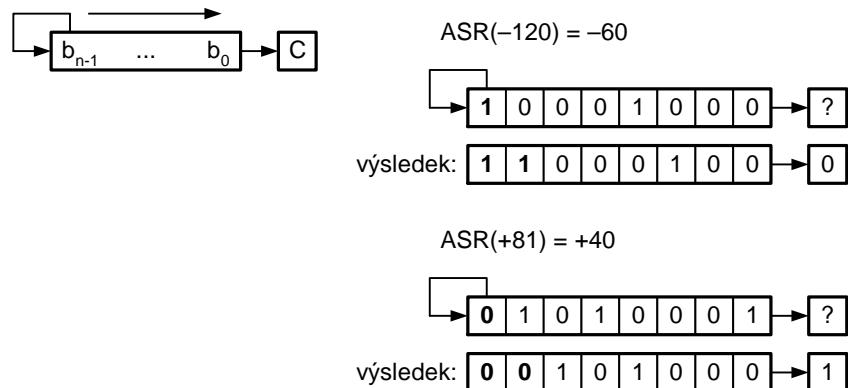
Operace LSL odpovídá **násobení** číslem 2, operace LSR odpovídá **dělení** číslem 2 v aritmetice celých čísel bez znaménka. Například:  
 $LSL(01010001_{(2)}) = 10100010_{(2)}$ , desítkově:  $LSL(81) = 162$ ,  
 $LSR(01010001_{(2)}) = 00101000_{(2)}$ , desítkově:  $LSR(81) = 40$ .

### Aritmetický posuv

Aritmetický posuv může opět probíhat doleva nebo doprava. Tento typ posuvu zohledňuje aritmetiku doplňkového kódu.

Operace **aritmetického posuvu doleva** odpovídá v zásadě logickému posuvu doleva, tedy operaci LSL. Jedná se tedy o násobení 2. Například:  $LSL(11011000_{(2)}) = 10110000_{(2)}$ , tedy ve dvojkovém doplňku:  $LSL(-40) = -80$ . Přetečení aritmetiky dvojkového doplňku lze indikovat testem hodnoty MSB před a po provedení posuvu (pokud se hodnoty liší, došlo k přeplnění aritmetiky druhého doplňku).

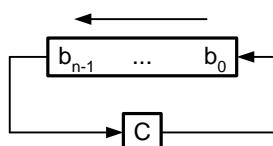
Operace **ASR** (Arithmetic Shift Right) provádí aritmetický posuv doprava, tedy dělení 2. Nesmí však dojít ke ztrátě znaménka. Proto se zleva nevkládá vždy 0, ale původní hodnota nejvyššího bitu. Viz obr. 3.26.



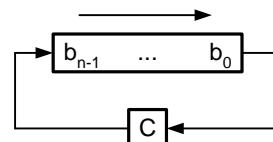
Obr. 3.26 Operace ASR a příklady výpočtu

### Rotace (cyklický posuv)

Při rotaci se vysouvané bity vrací zpět do výsledku z opačné strany. Níže uvažujeme rotace prováděné skrz příznak C. Operace **ROL** (Rotate Left through Carry), **ROR** (Rotate Right through Carry) tedy značí rotaci doleva nebo doprava skrz příznak C. Viz obr. 3.27 a 3.28.



Obr. 3.27 Operace ROL

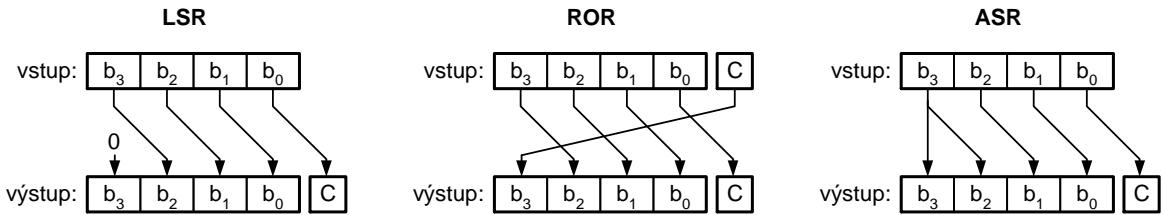


Obr. 3.28 Operace ROR

### Shifter

Operaci posuvu si přirozeně představujeme realizovanou pomocí posuvného registru. Ovšem pro takový případ by bylo nutné vytvořit časování, které by v průběhu výpočtu zajistilo taktování posuvného registru. To však není z principu možné, protože samotná ALU je kombinační obvod.

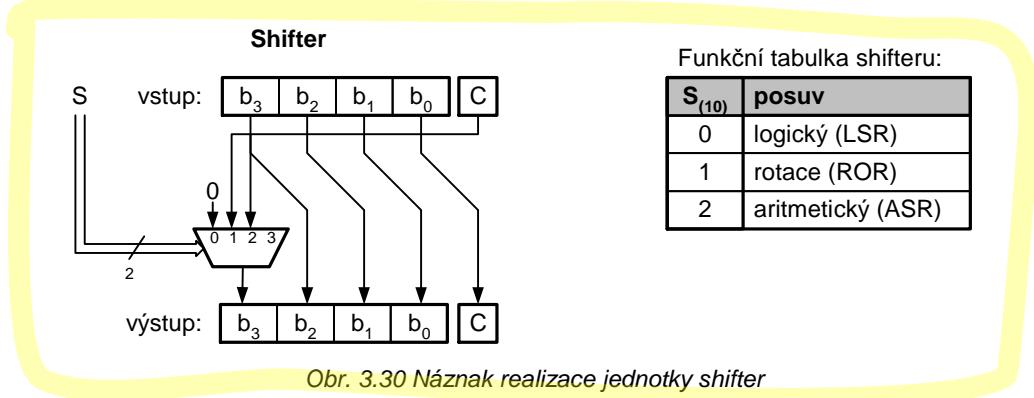
Místo toho se tedy používá čistě kombinační obvod, označovaný jako **shifter**. Náznak realizace pro posuvy prováděné doprava je proveden na obr. 3.29.



Obr. 3.29 Náznak realizace posuvů doprava pomocí jednotky shifter pro 4bitové operandy

Z obr. 3.29 je zřejmé, že jednotlivé posuvy se liší pouze v získání hodnoty nejvyššího bitu. Pro logický posuv je vložena 0, pro rotaci stav příznaku C před operací a pro aritmetický posuv pak původní hodnota nejvyššího bitu.

Výběr hodnoty nejvyššího bitu lze tedy snadno provést například pomocí multiplexeru, náznak výsledné realizace je uveden na obr. 3.30. Symbol **S** označuje 2bitový kód zvoleného posuvu.

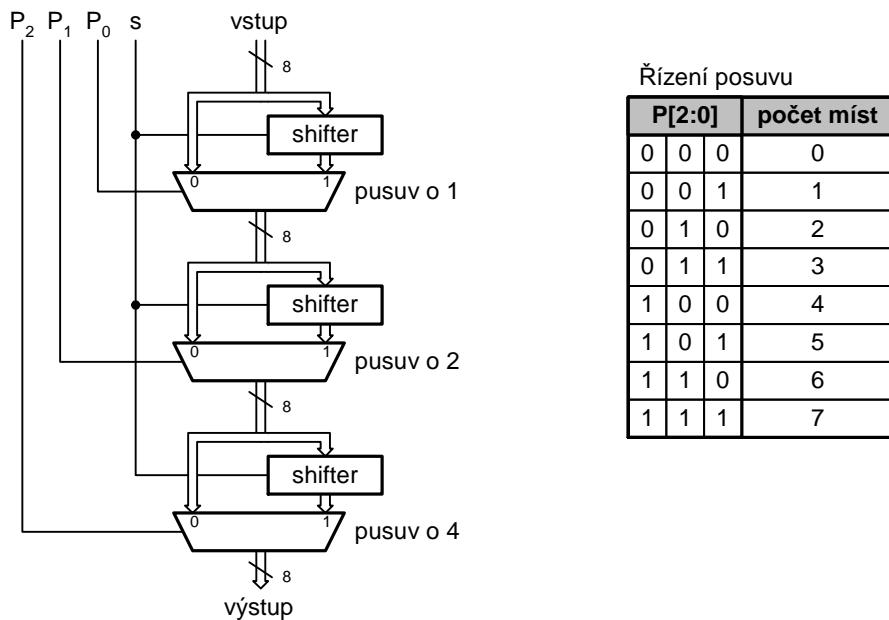


Obr. 3.30 Náznak realizace jednotky shifter

### Barrel Shifter

Jak bylo uvedeno výše, dokáže jednotka shifter realizovat posuv o jeden bit rýze využitím kombinační logiky.

V případě požadavku posuvu o více míst, lze posuvy provádět opakováně. Jinou možností (pokud jí tedy ALU disponuje) je jednotka **barrel shifter**.



Obr. 3.31 Náznak realizace jednotky barrel shifter

Tato jednotka je schopna provést posuv o stanovený počet bitů (prakticky má smysl pouze rozsah 0 až n-1) doleva nebo doprava. Jedná se opět o kombinační obvod, který je tvořen kaskádním spojením  $\log_2(n)$  základních shifterů.

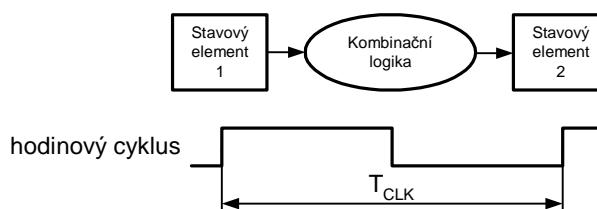
Příklad realizace jednotky barrel shifter pro logické posuvy pro 8bitová čísla ( $n = 8$ ) je uveden na obr. 3.31. Symbol **P** odpovídá počtu míst posuvu, symbol **s** určuje směr posuvu (doleva nebo doprava).

### 3.4 GPR – základ datové cesty

Procesor je tvořen spojením kombinační a sekvenční logiky.

Paměťový element (registr) udržuje stav, zápis je řízen hodinovým signálem. Naproti tomu kombinační část určuje následující stav.

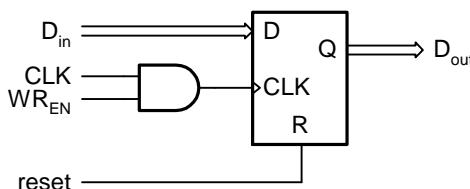
Hodinový signál řídí zápis do registrů. Tento signál má kmitočet  $f_{CLK}$  a periodu  $T_{CLK}$ . Má-li systém správně fungovat, je nezbytné, aby platilo, že zpoždění kombinační logiky je kratší než perioda hodinového signálu, tedy:  $T_{CLK} > t_{PD}$ .



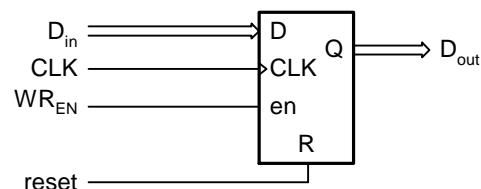
Obr. 3.32 Úloha hodinového signálu

Zápis dat do stavového elementu neprobíhá v každém taktu. Pro řízení zápisu je třeba doplnit výběrový signál **WRen** (Write Enable, tedy povolení zápisu).

Stavový element tedy lze vytvořit z klopného obvodu typu D, který doplníme hradlem typu AND pro řízení zápisu, viz obr. 3.33 a 3.34.



Obr. 3.33 Registr doplněný o řízení zápisu



Obr. 3.34 Registr s řízením zápisu

### GPR (General Purpose Registers – obecně použitelné registry)

Blok GPR tvoří základ datové cesty.

Příklad GPR složeného ze 4 registrů je zakreslen na obr. 3.35.

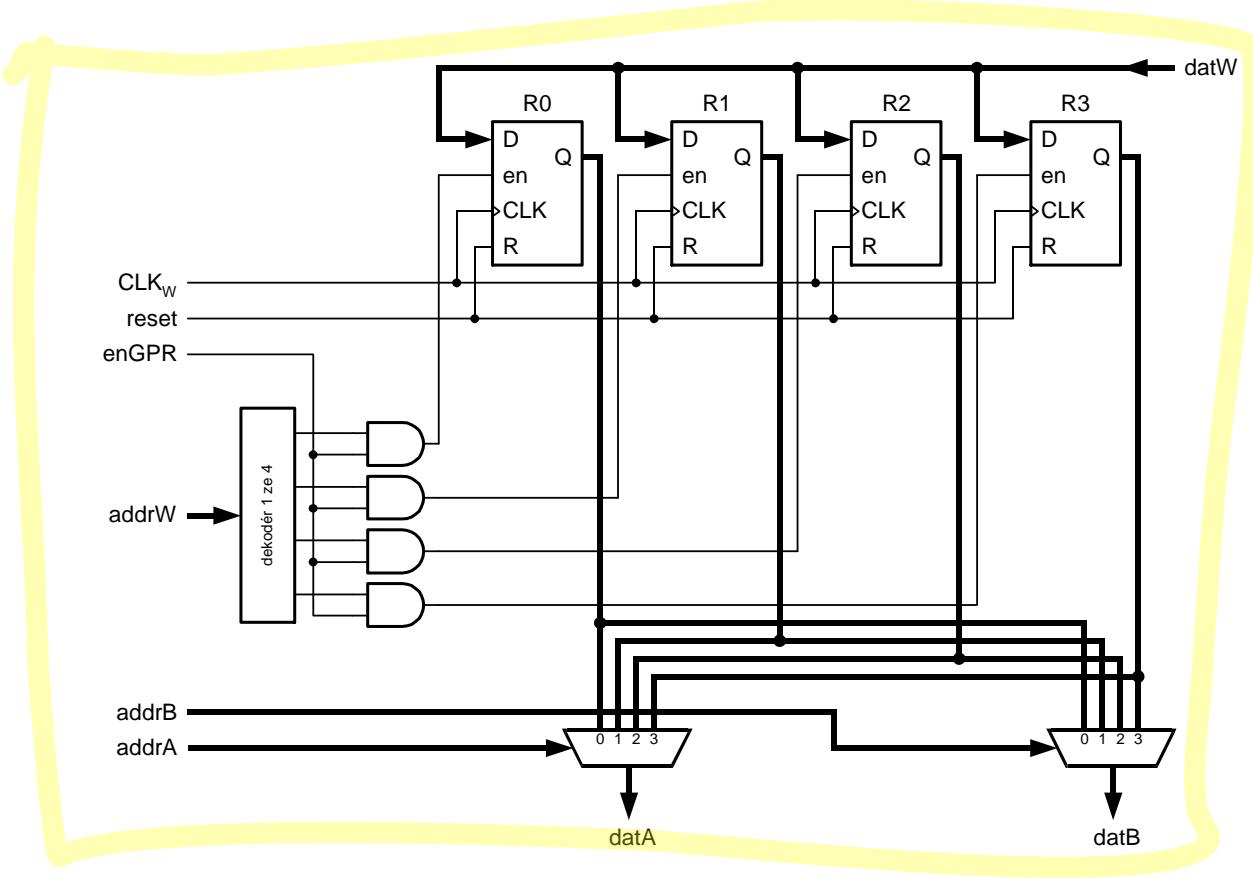
Jedná se o blok registrů, které mají společný vstup dat pro zápis (**datW**) a dva možné datové výstupy (**datA**, **datB**).

Registr pro zápis se vybírá pomocí adresy **addrW**. Dekodér 1 ze 4 zajistí výběr právě jednoho registru ze skupiny **R0** až **R3**. Pokud je aktivní signál **enGPR**, je daný registr připraven k zápisu.

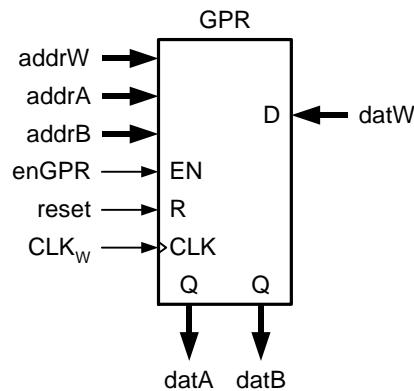
Registry mají dále společný hodinový vstup **CLK<sub>w</sub>** (hodinový signál pro zápis) a **reset** (nulování).

Pro čtení dříve zapsaných údajů nejsou obvykle k dispozici všechny dílčí výstupy. Výstupy se pomocí multiplexerů „koncentrují“ do signálů **datA** a **datB**. Výběr registru pro čtení je určen adresami **addrA** a **addrB**.

Přítomnost dvou výstupů vlastně naznačuje napojení GPR na vstupy ALU.



Obr. 3.35 Blokové schéma bloku GPR se čtyřmi registry



Obr. 3.36 Bloková značka GPR

## 4 Architektury počítačů

V této kapitole se nejdříve seznámíme s vlastnostmi architektur Von Neumann a Harvard. Dále provedeme rozbor konstrukce počítače s harvardskou architekturou, včetně podrobnějšího výkladu pojmu instrukce a důležitých bloků.

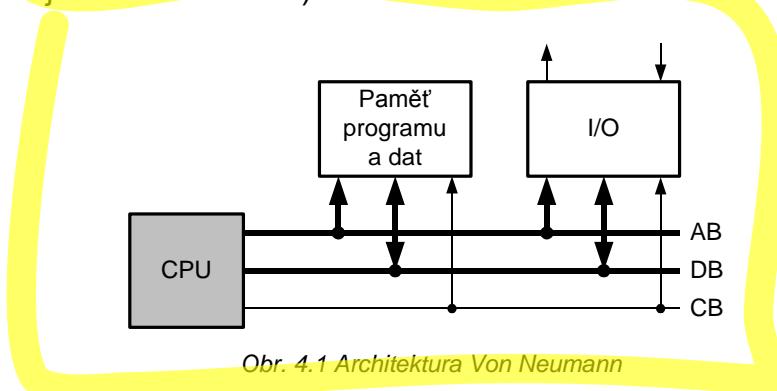
### 4.1 Přehled architektur počítačů

Pro vytváření počítačů se používají dvě architektury: Von Neumann a Harvard.

#### Architektura Von Neumann

Na obr. 4.1 je uvedena architektura **Von Neumann**, pojmenovaná dle svého tvůrce Johna von Neumanna, který navrhl již v roce 1946 schéma univerzálního počítacího stroje. Ve schématu jsou použity tyto zkratky:

- **AB** (Address Bus – adresová sběrnice) – určuje adresu paměťového místa nebo vstupu/výstupu, se kterým pracujeme,
- **DB** (Data Bus – datová sběrnice) – přenáší data pro paměť dat, vstupy/výstupy nebo instrukce z paměti programu,
- **CB** (Control Bus – řídicí sběrnice) – představuje řídicí signály pro řízení komunikace (čtení, zápis, ...),
- **I/O** (Inputs/Outpus – vstupy/výstupy, česká zkratka V/V) – obsluhují připojené periferie, přístup se z hlediska procesoru provádí pomocí speciálních instrukcí (obvykle mají název IN a OUT).



Základem koncepce Von Neumann je **společná paměť pro program i data**.

Vidíme, že adresová sběrnice je z hlediska paměti a bloku vstupu/výstupu vždy vstupní. Paměťové místo nebo periferii, se kterou se pracuje, vybírá vždy procesor. Podobně řídicí signály produkuje také procesor.

Datová sběrnice bloku vstupu/výstupu je pochopitelně obousměrná. Jelikož je paměť používána jak pro čtení programu, tak pro čtení/zápis dat, musí být datová sběrnice obousměrná rovněž u paměti.

#### Nevýhody

- nebezpečí nechtěného přepsání programu daty,
- tato koncepce tvoří brzdu výkonu, protože řadič procesoru je poměrně komplikovaný a tím klesá jeho výkon.

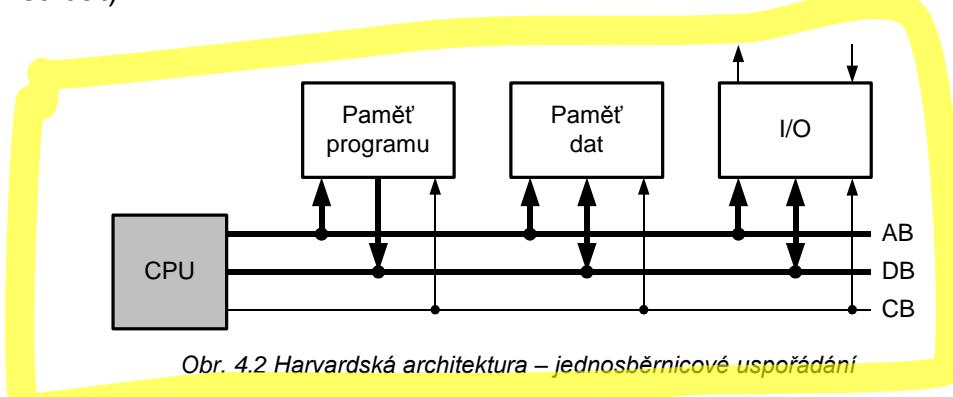
#### Harvardská architektura

Základem koncepce Harvardské architektury jsou **oddělené paměťové prostory pro program i data**.

Základní jednosběrnicové uspořádání je uvedeno na obr. 4.2.

Je zřejmé, že z paměti programu se údaje (instrukce) pouze čtou. Datová sběrnice pro paměť programu je tedy jednosměrná.

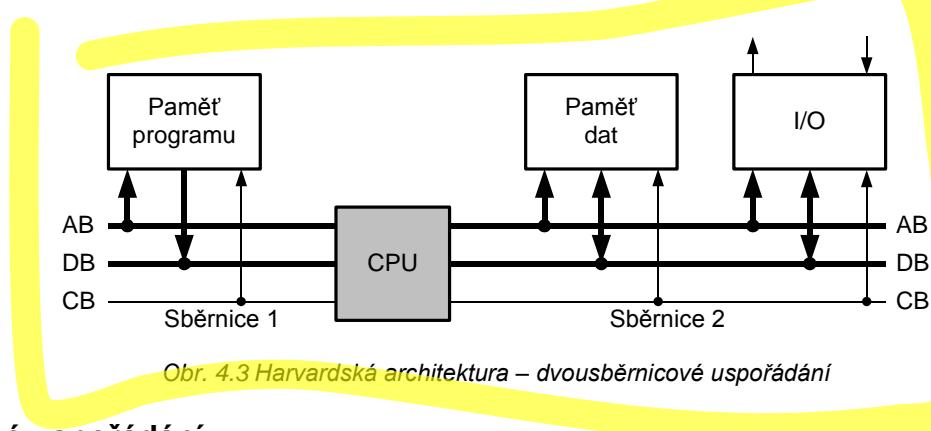
Dále si můžeme uvědomit, že stejnou adresu lze používat jak v rámci paměti programu, tak i pro paměť dat. Proto musí řídicí sběrnice vytvářet zvláštní signály pro programovou a datovou paměť (čtení z paměti programu, čtení z paměti dat, zápis do paměti dat).



Obr. 4.2 Harvardská architektura – jednosběrnicové uspořádání

Jednosběrnicové uspořádání představuje rychlostní omezení. Při provádění instrukce pracující s datovou pamětí se totiž instrukce z paměti programu a data z paměti dat načítají postupně.

Níže je uvedena **koncepce dvousběrnicové architektury**, která dovoluje současné načítání instrukce z paměti programu a údaje z paměti dat.



Obr. 4.3 Harvardská architektura – dvousběrnicové uspořádání

### Smíšené uspořádání

V současné době se můžeme setkat s tzv. smíšeným uspořádáním. Z pohledu programátora se počítač jeví dle architektury Von Neumann. Samotný procesor však pracuje s oddělenými paměti, které jsou připojeny na zvláštních sběrnicích. Což odpovídá harvardské architektuře.

## 4.2 Konstrukce jednoduchého 8bitového harvardského počítače

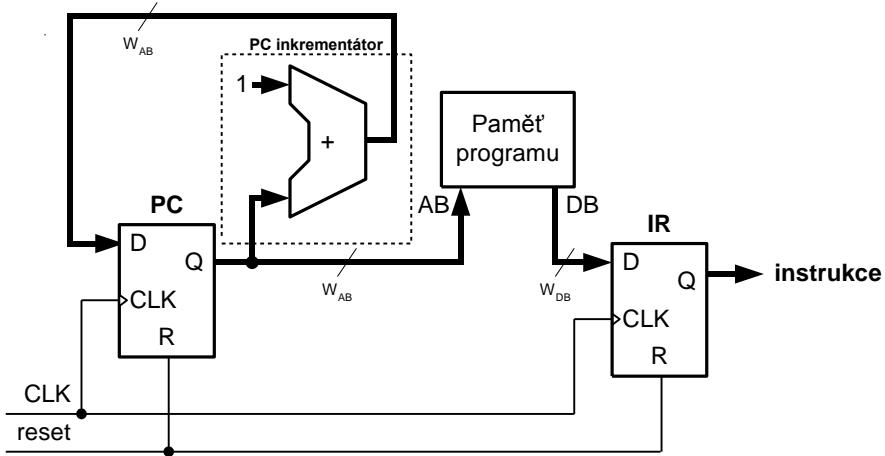
Níže uvedeme konstrukci jednoduchého počítače dle harvardské architektury. Vysvětlíme také úlohu jednotlivých částí řadiče procesoru a jejich návaznost na ALU.

### Základní pojmy

- **Instrukce** – slovo, jehož obsah pro procesor znamená provedení dané operace.
- **Instrukční cyklus IC** – interval, za který je provedena instrukce nebo její část. Interval provedení instrukce tedy odpovídá K-násobku periody hodin  $T_{CLK}$ , kde  $K \geq 1$ .
- **Reset** – uvedení počítače/procesoru do výchozího stavu. Dochází především k nastavení registrů na výchozí hodnoty. Program se „rozeběhne“ od začátku.

## Registry PC, IR

Procesor vykonává program tak, že provádí posloupnost instrukcí uložených v paměti programu. Základ této funkcionality je uveden formou obr. 4.4.



Obr. 4.4 Programový čítač, PC inkrementátor a registr instrukce

Registr **PC – Program Counter** (programový čítač) obsahuje adresu právě prováděné instrukce. Instrukce je načtena z paměti programu a připravena pro zápis do registru **IR – Instruction Register** (registr instrukce).

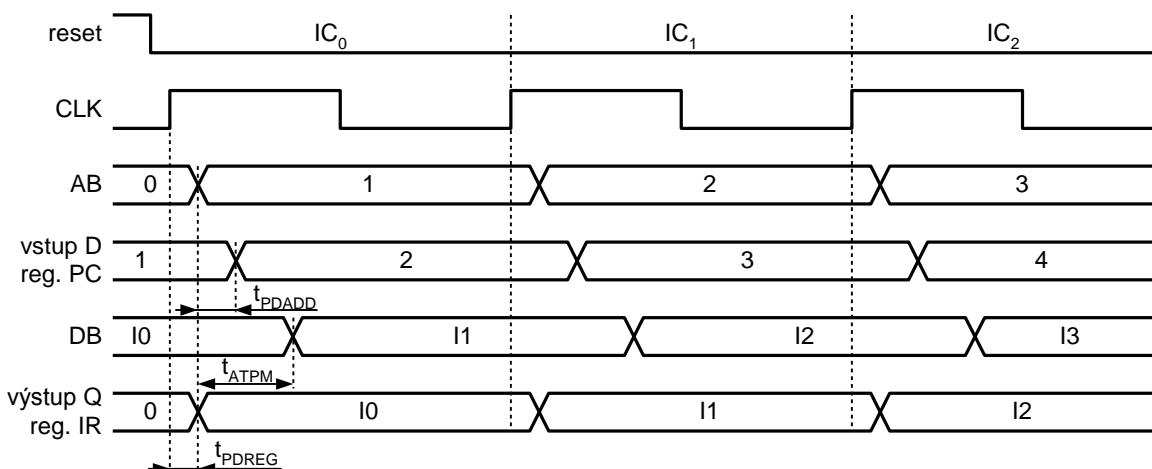
Blok označený jako **PC inkrementátor** představuje sčítáku, která k aktuální hodnotě uložené v registru PC přičítá konstantu 1. Tím vypočítává adresu následující instrukce.

Po **resetu** jsou oba registry vynulovány:  $PC = 0$ ,  $IR = 0$ . Program se tedy „rozbíhá“ od adresy 0.

První instrukce z adresy 0 v paměti programu je připravena na vstupu registru instrukce IR. Podobně je na vstupu registru PC připravena adresa následující instrukce získaná přičtením konstanty 1 k předchozímu obsahu PC (tedy 1).

První hodinový impulz po resetu uloží instrukci do registru IR a tím dá povel k jejímu vykonání. Současně je nová adresa uložena do registru PC ( $PC = 1$ ). Z paměti se přečte další instrukce (z adresy 1), sčítáka vypočítá následující adresu (2) a vše pokračuje dále. Tím se „prochází“ paměť a procesor vykonává instrukce v pořadí, jak jsou v paměti uloženy.

Symboly  $W_{AB}$  a  $W_{DB}$  odpovídají šířce adresové a datové sběrnice.



Obr. 4.5 Diagram časování

Na obr. 4.5 je ještě uveden časový diagram signálů a sběrnic, který odpovídá předchozímu popisu. Interval  $t_{PDADD}$  odpovídá průchozímu zpoždění sčítáčky,  $t_{ATPM}$  odpovídá přístupové době do paměti programu a  $t_{PDREG}$  pak průchozímu zpožděnímu registru.

V předchozím textu jsme uvažovali, že se hodnota registru PC zvyšuje (inkrementuje) vždy o 1. Neuvažovali jsme **instrukce různé délky**. Dokonalejší verze PC inkrementátoru počítá s délkou instrukce. Uvažujme například, že jsou na začátku paměti uloženy instrukce délky: jedno slovo, dvě slova a jedno slovo. Správný „pohyb“ v paměti programu je uveden níže:

<i>adresa v paměti programu</i>	<i>instrukce</i>
0	instrukce délky 1 slovo
1	instrukce délky 2 slova
3	instrukce délky 1 slovo

Manipulace s registrem PC je však mnohem náročnější.

Zatím jsme totiž uvažovali, že se programovou pamětí prochází pouze jedním směrem. Že tedy nedochází k **větvení, opakovanému provádění bloku instrukcí, ani volání podprogramů**. Tyto akce vedou k přímé změně obsahu PC.

Například větvení programu znamená, že pokud je splněna určitá podmínka, přejde provádění programu do jiného místa. Tedy adresa instrukce, kterou se má pokračovat při splnění podmínky, se přímo zapíše do registru PC. Říkáme, že se provede skok na novou adresu. Pokud podmínka splněna není, pokračuje program následující instrukcí.

### Instrukce

**Instrukce** je kódovaný příkaz, tedy číslo strojové operace, která se má vykonat.

Obsah instrukce:

- operační kód (Operation Code, opcode) – určuje, co se má provést,
- operandy – nad čím se má operace provést,
- cíl – místo pro uložení výsledku,
- změna registru PC – určuje, kde bude program pokračovat.

Základní údaje o instrukci:

- kód instrukce – číslo vyjadřující jednoznačně prováděnou operaci,
- doba provádění instrukce – určuje se v násobcích instrukčních cyklů IC,
- délka instrukce – udává, kolik slov zabere instrukce v paměti programu,
- ovlivnění příznaků a registru PC.

### Procesor

Z předchozích úvah již můžeme přistoupit k nakreslení blokového schématu procesoru. Budeme uvažovat, že mikroprocesor má implementovány instrukce:

- NOP (No OPeration): instrukce, která neprovede „nic“, kód: obvykle 0,
- ALU instrukce – viz kapitolu 3,
- LDI: “nahraj do registru konstantu”.

Formát těchto uvažovaných instrukcí je uveden níže.

## Formáty instrukcí

Název	operands	instr[15:12]	instr[11:8]	instr[7:4]	instr[3:0]
NOP		0000	0000	0000	0000
ALUinstr	Rd, Rr	opc	0000	dddd	rrrr
LDI	Rd, K8	opc	kkkk	dddd	kkkk

Instrukce prováděné nad ALU mají jako operandy 4bitová čísla zdrojových a cílových registrů (rrrr a dddd), lze tedy vybírat ze 16 registrů uložených v GPR. Operační kód pro součet (ADD) zvolíme například 0001 a pro rozdíl (SUB) 0010:

- mnemokód: ADD Rd, Rr
  - obecný strojový kód: 0001 0000 dddd rrrr
  - příklad: ADD R3, R1
  - strojový kód: 0001 0000 0011 0001
- 
- mnemokód: SUB Rd, Rr
  - obecný strojový kód: 0010 0000 dddd rrrr
  - příklad: SUB R2, R5
  - strojový kód: 0010 0000 0010 0101

Pro instrukci LDI zvolíme operační kód například 0101:

- mnemokód: LDI Rd, K8
- obecný strojový kód: 0101 kkkk dddd kkkk
- příklad: LDI R4, 0x83
- strojový kód: 0101 1000 0100 0011

Této úvaze pak odpovídá blokové schéma procesoru dle obr. 4.6. Nejvyšší 4 byty (označeno **inst[15:12]**) odpovídají operačnímu kódu, proto vstupují do jednotky **ID** (Instruction Decoder – dekodér instrukce). Dekodér instrukce vytváří řídicí signály pro ostatní bloky.

Dále je zřejmé, že nejnižší 4 byty instrukce (označeno **inst[3:0]**) odpovídají zdrojovému operandu instrukce (Rr). Určují adresu registru, který vstoupí do ALU jako operand B.

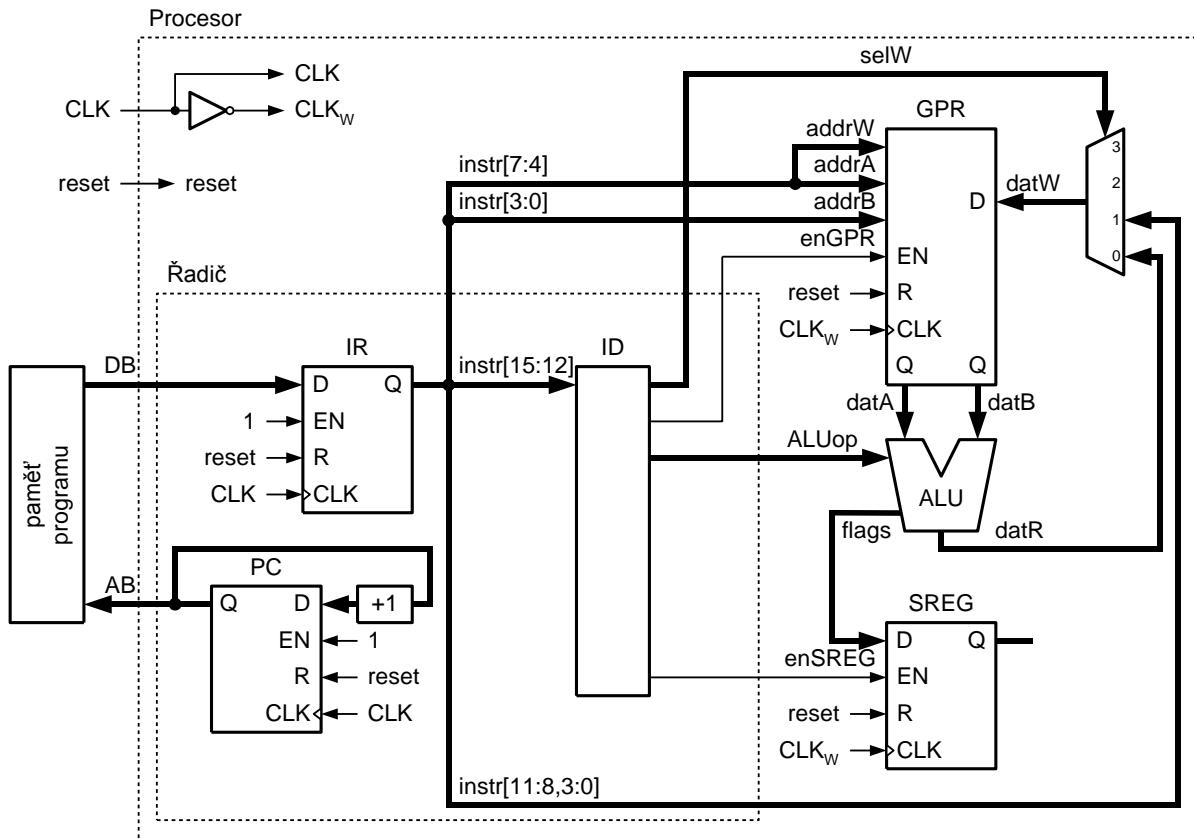
Podobně další 4 byty instrukce (označeno **inst[7:4]**) odpovídají cílovému operandu instrukce (Rd). Určují jednak adresu registru, který vstoupí do ALU jako operand A. Dále určují adresu cílového registru. Takže instrukce typu ALUinstr Rd, Rr provede operaci nad oběma registry a výsledek uloží zpět do registru Rd.

Bity 11 až 8 a 3 až 0 používá instrukce **LDI** pro načtení konstanty, která má být uložena do příslušného cílového registru. Signál **inst[7:4, 3:0]** tedy směruje do GPR ve formě dat pro zápis (po příslušném nastavení multiplexeru).

Generování řídicích signálů dekodérem instrukci ID pak jasně vychází z tohoto popisu a je uvedeno formou tabulky níže.

### Řídicí signály

instrukce	selW	enGPR	ALUop	enSREG
NOP	00	0	000	0
ADD	00	1	001	1
SUB	00	1	010	1
LDI	01	1	000	0

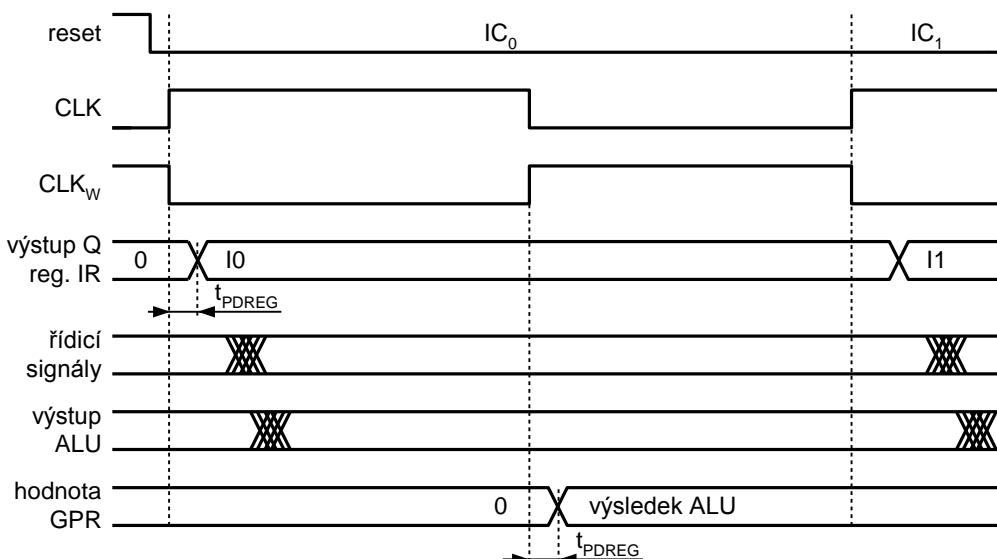


Obr. 4.6 Blokové schéma procesoru

Instrukce **LDI** tedy požaduje vložit jako zapisovaná data bity 11 až 8 a 3 až 0 instrukce. Proto je selW = 1. Ostatní instrukce ale mají selW = 0.

Instrukce **NOP** představuje prázdnou operaci, obsah GPR nesmí být změněn. Proto je zápis odstaven (enGPR = 0). Ostatní instrukce zapisují svoje výsledky do GPR, proto enGPR = 1.

ALU operace musí specifikovat svůj typ pomocí ALUop a dále zapisují příznaky do registru **SREG** (Status REGister – Stavový registr), proto enSREG = 1. Ostatní operace stav příznaků nemění a pro ně tedy platí enSREG = 0.



Obr. 4.7 Časování ALU operací

Z obr. 4.6 je zřejmé, že procesor používá hodinové signály **CLK** a **CLK<sub>w</sub>**. Tyto signály jsou vzájemnou negací.

Signál CLK je použit pro taktování bloků řadiče. Signál CLK<sub>w</sub> pak odpovídá zápisu do GPR a SREG. Důvodem je zpoždění kombinační logiky v ALU. Pokud má být každá instrukce provedena během jediného instrukčního cyklu, který začíná náběžnou hranou hodin CLK a zpoždění ALU odpovídá maximálně jedné polovině T<sub>CLK</sub>, lze výsledek zapsat při sestupné hraně CLK. Jelikož uvažujeme všechny registry citlivé na náběžnou hranu, byl tento problém vyřešen pomocným signálem CLK<sub>w</sub>, který je v protifázi vůči CLK.

### Instrukční soubor

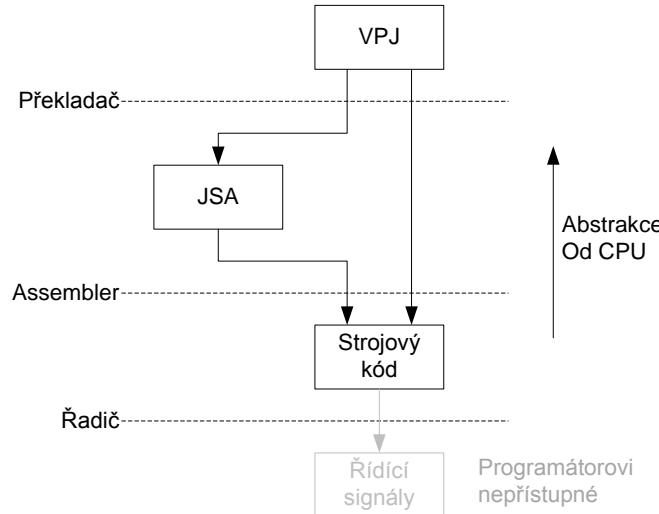
Instrukční soubor je množina všech instrukcí daného procesoru. Jednotlivé instrukce se obvykle dělí do skupin:

- aritmetické operace (součet, rozdíl, inkrementace, dekrementace,...),
- bitové (logické) operace (or, and, xor, posuvy),
- instrukce přesunu dat (registrový → RAM, registrový → registrový, ...),
- větvení,
- skoky, volání podprogramů,
- NOP – prázdná operace, kód 0.

Každý procesor má svůj vlastní instrukční soubor, procesory se odlišují počtem a typem instrukcí. Není možná přenositelnost programů mezi různými procesory na úrovni instrukcí. Přenositelnost mezi různými procesory je možná pouze na úrovni vyšších programovacích jazyků.

### 4.3 Možnosti programování procesoru

**Vyšší programovací jazyk** (VPJ) disponuje příkazy, které nutně neodpovídají instrukcím procesoru. Je zajištěn vysoký stupeň nezávislosti (abstrakce) na konkrétním procesoru, syntaxe je vhodná pro člověka. Zdrojový kód je překládán do strojového kódu (tedy toku instrukcí) překladačem.



Obr. 4.8 Porovnání vyššího programovacího jazyka a assembleru

**Jazyk symbolických adres/instrukcí** (JSA/JSI) umožňuje zápis programu na úrovni strojového kódu. Instrukce jsou vyjádřeny mnemotechnickými zkratkami

(proto jazyk symbolických instrukcí). Místo konkrétních adres používáme symboly návěští (proto jazyk symbolických adres).

Příklad:

```
ldi r15, 30
dec r15
skok: jmp skok
```

**Assembler** je označení pro překladač z jazyka symbolických adres do strojového kódu. Často dochází k posunu tohoto termínu, jako assembler můžeme pak označovat i samotný jazyk symbolických adres.

Technika označovaná jako **disassembling** umožňuje ze strojového kódu získat program v jazyce symbolických adres. Nástroj, který tuto operaci provádí, označujeme jako **disassembler**.

## 5 Architektura instrukčního souboru AVR I

Instrukční sada neboli architektura instrukčního souboru (ISA) zahrnuje datové typy, instrukce, adresovací režimy, architekturu paměti, zpracování přerušení a výjimek a vnější vstupy/výstupy.

V této kapitole se seznámíme s adresovacími režimy, klíčovými vlastnostmi mikrokontrolérů AVR a některými instrukcemi.

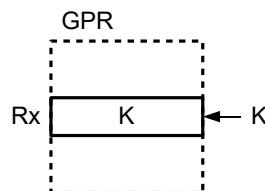
### 5.1 Adresovací režimy (módy adresování)

Adresovací režimy (módy) popisují způsob získání operandu resp. možné zdroje operandů. Pro mnemokódy instrukcí používáme symboly:

- **LD** (LoaD) – odpovídá situaci, kdy do registru nahrajeme určený operand,
- **ST** (STore) – odpovídá situaci, kdy operand (z registru) uložíme do paměti.

#### Operand typu konstanta

Příkladem instrukce, která používá operand typu konstanta, může být LDI Rx,K (LoaD Immediate). Tato instrukce „nahraje“ do registru Rx hodnotu konstanty K. Hodnota konstanty je bezprostřední součástí instrukce.

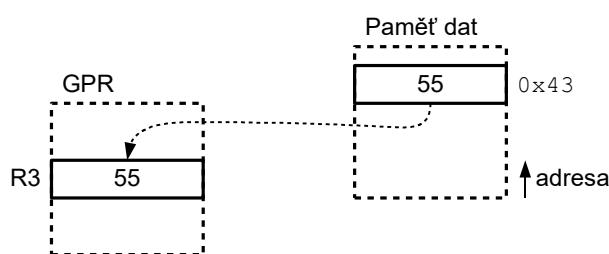


Obr. 5.1 Instrukce **LDI Rx,K**

#### Přímé adresování (Direct Addressing)

Přímé adresování znamená, že operandem je adresa. Může se jednat o adresu registru nebo paměťového místa. Adresa je přímou součástí instrukce.

Příkladem může být instrukce: LDS R3, 0x43 (Load Direct from Data Space). Viz obr. 5.2.

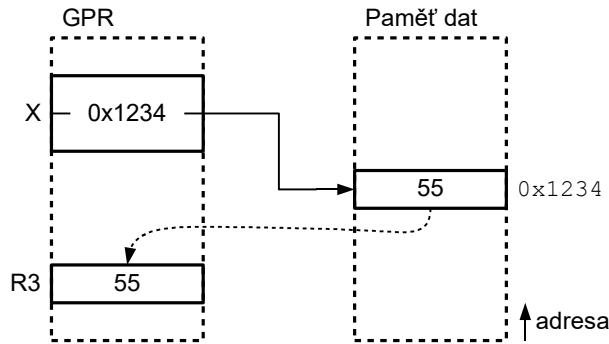


Obr. 5.2 Instrukce **LDS R3,0x43**

#### Nepřímé adresování (Indirect Addressing)

Nepřímé adresování znamená, že operandem je adresa. Může se jednat o adresu registru nebo paměťového místa. Adresa nyní není součástí instrukce, ale je předávána v určitém registru. Nepřímé adresování vlastně odpovídá použití ukazatele.

Příkladem může být instrukce: LD R3,X (Load Indirect from Data Space to Register using Index). Tato instrukce použije obsah registru X (uvažujeme 0x1234) jako adresu do paměti dat. Z tohoto paměťového místa nahraje údaj a uloží jej do registru R3. Viz obr. 5.3.



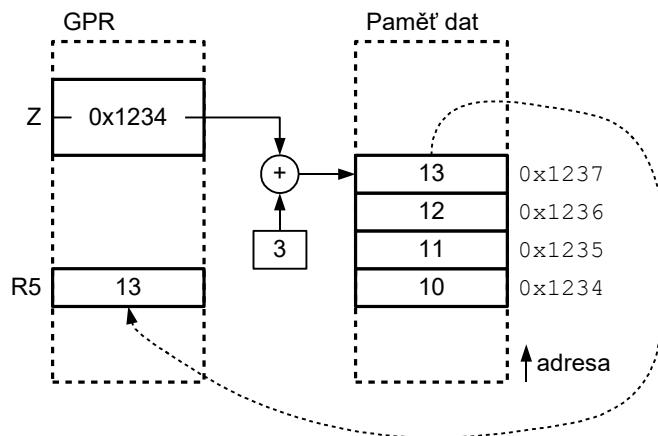
Obr. 5.3 Instrukce **LD R3,X**

### Nepřímé adresování se změnou ukazatele

Doplněním předchozího režimu je spojení operace čtení údaje z paměti se změnou ukazatele, například (inkrementace nebo dekrementace ukazatele X): LD R3,X+ nebo LD R3,-X.

### Nepřímé adresování s posunutím (Indirect Addressing with Displacement)

Nepřímé adresování s posunutím sestavuje adresu pro přístup do paměti ze dvou složek: nepřímé adresy uložené v registru a přímého posunutí, které je součástí instrukce. Příkladem může být instrukce: LDD R5,Z+3.



Obr. 5.4 Instrukce **LD R5,Z+3**

### Implicitní adresování

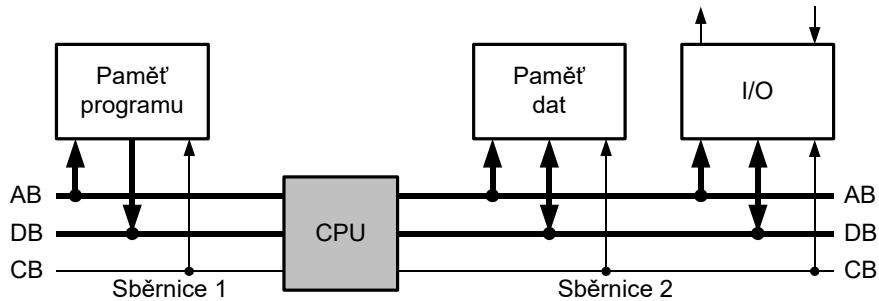
Implicitní adresování znamená, že cílový operand je přímo stanoven instrukcí. Jedná se vlastně o „zamlčený“ parametr. Příkladem je instrukce: LPM (Load Program Memory). Instrukce čte obsah paměťového místa v paměti programu (adresa je určena obsahem registru Z) a nahraje tento údaj do registru R0.

## 5.2 Základní vlastnosti mikrokontrolérů AVR

Připomeňme, že označení mikrokontrolér znamená, že samotný procesor (CPU) je doplněn bloky paměti dat a programu a vstupně/výstupními obvody, které umožní samostatnou činnost tohoto celku.

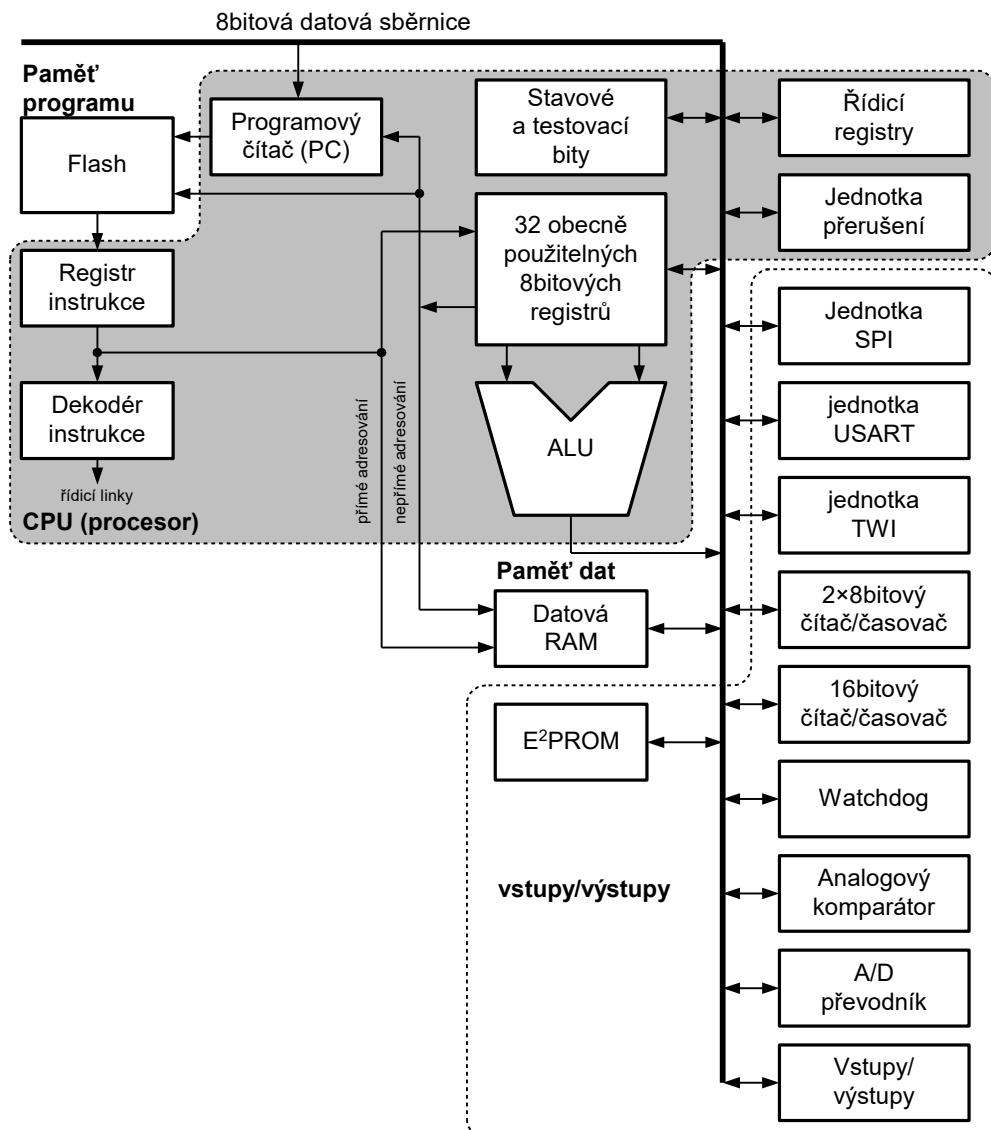
## Architektura

Mikrokontroléry (správněji mikrořadiče) AVR jsou založeny na 8bitové harvardské architektuře s dvousběrnicovým uspořádáním. Viz obr. 5.5.



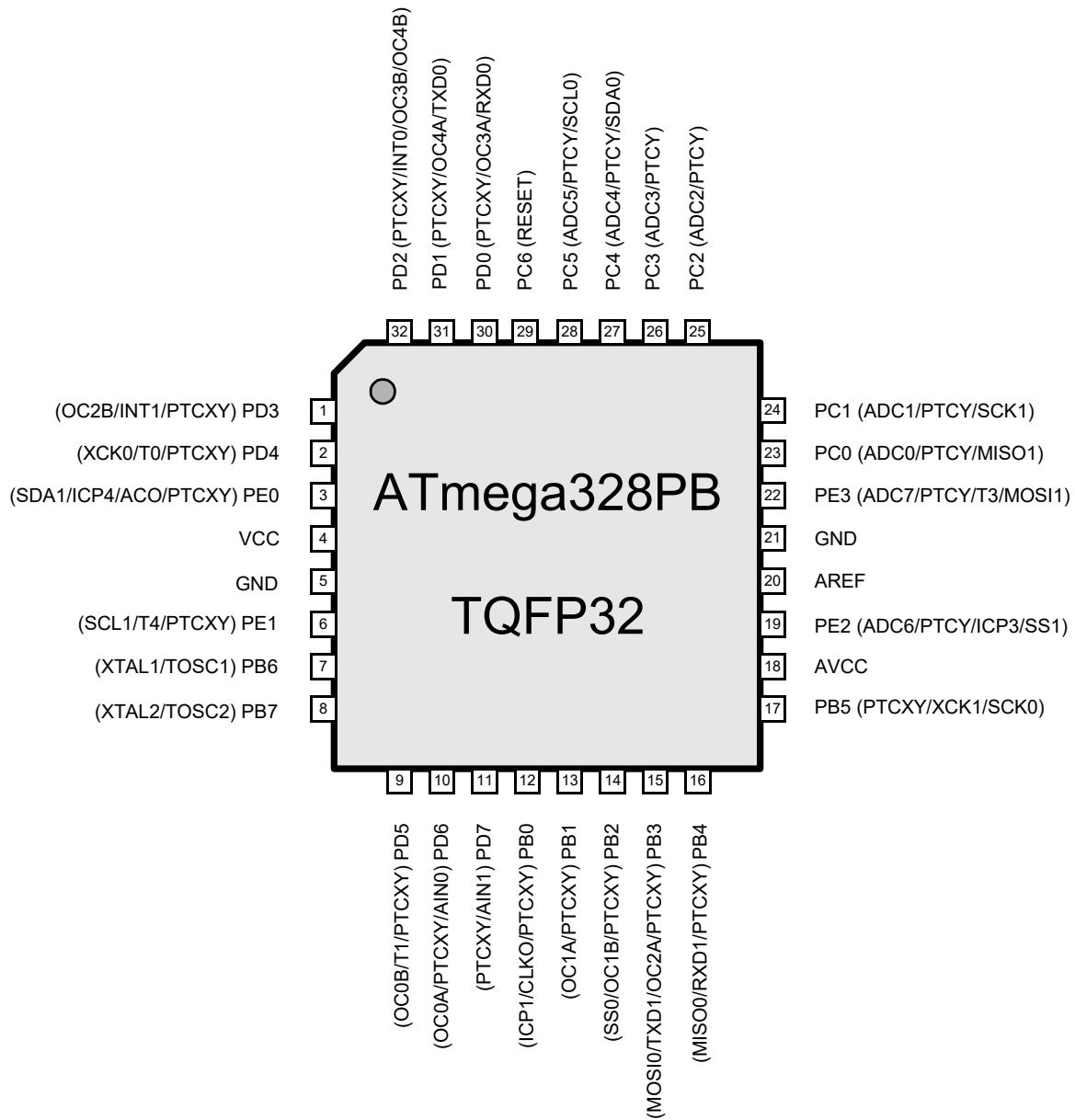
Obr. 5.5 Harvardská architektura s dvousběrnicovým uspořádáním

Blokové schéma mikrokontroléru řady AVR ATmega je uvedeno formou obr. 5.6.



Obr. 5.6 Blokové schéma ATmega

Jednotlivé varianty řady ATmega se odlišují především velikostí pamětí a také vybavením periferiemi. Pojem periferie je v souvislosti s mikrokontrolérem třeba chápat s určitým posunutím. Jedná se o nedílnou součást mikrokontroléru, která však není součástí samotného procesoru (CPU).



Obr. 5.7 Pouzdro mikrokontroléru ATmega328PB (skutečná velikost 7×7 cm) s popisem vývodů

Společné vlastnosti řady ATmega jsou shrnuty níže:

- instrukční soubor obsahuje až **135 instrukcí** (některé varianty nemají implementovány všechny tyto instrukce),
- GPR** o velikosti **32 registrů** délky 8 bitů (R0 až R31),
- hodinový kmitočet 0 až 16 MHz (též varianty s kmitočtem 20 MHz),
- paměť programu je tvořena zabudovanou **Flash** (viz tab. 5.1),
- datová paměť **RAM** (viz tab. 5.1),
- datová paměť **E<sup>2</sup>PROM** (viz tab. 5.1),

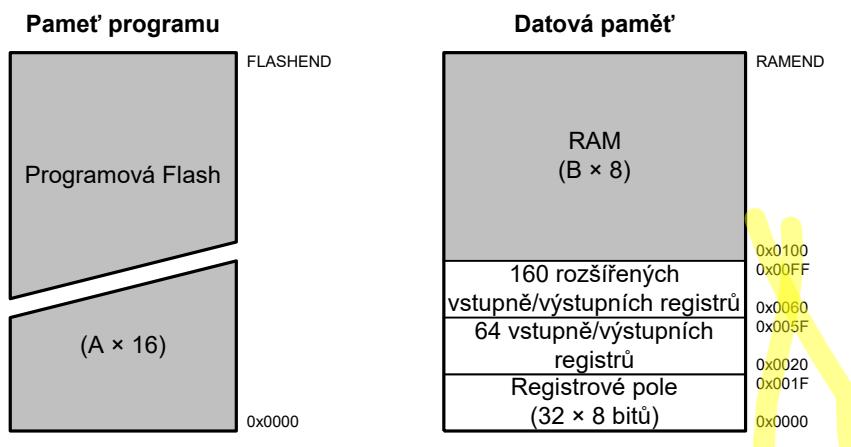
- paměti Flash a E<sup>2</sup>PROM jsou programovatelné přímo v systému pomocí rozhraní SPI nebo JTAG,
- dva 8bitové čítače/časovače, jeden 16bitový čítač/časovač (též varianty s vyšším počtem čítačů/časovačů),
- PWM kanály napojené na čítače/časovače (počet se liší podle konkrétního mikrokontroléru),
- analogový komparátor, 10bitový A/D převodník,
- jednotky USART, SPI, TWI (podpora I<sup>2</sup>C),
- jednotky WDT, Power-on reset.

Tab. 5.1 Klíčové vlastnosti vybraných typů mikrokontrolérů řady ATmega

Typ	Flash [KB]	RAM [KB]	E <sup>2</sup> PROM [B]	Kmitočet [MHz]
ATmega16	16	1	512	0 až 16
ATmega32	32	2	1024	0 až 16
ATmega644	64	4	2048	0 až 20

### Paměti programu a dat

Vnitřní **programová paměť** je představována pamětí **Flash**. Tato paměť je z hlediska probíhajícího programu adresována po slovech. Kapacita paměti je dána typem mikrokontroléru dle tab. 5.1. Počet adres tedy je: ATmega16 (8 K), ATmega32 (16 K), ATmega644 (32 K).



Obr. 5.8 Mapa paměti programu a dat

Vnitřní **datová paměť** je tvořena statickými buňkami architektury SRAM (Static RAM), kapacita paměti je určena tab. 5.1. Z obr. 5.8 je zřejmé, že datová paměť začíná blokem 32 bajtů, které odpovídají GPR (R0 až R31). Následují vstupně/výstupní registry (řídí periferie). Posledním blokem je libovolně použitelná paměť.

Z obr. 5.8 je také zřejmé, že každý GPR má svoji adresu v rozsahu 0 až 31, tedy se chová jako paměťové místo (přístup přes instrukce pro práci s pamětí je pomalejší, než používání stejných buněk jako registry). Nebudeme-li tedy používat všechny registry GPR, můžeme zbytek chápat jako paměťové buňky.

Kromě výše uvedené datové a programové paměti je k dispozici **datová E<sup>2</sup>PROM**, kapacita paměti je určena tab. 5.1. Připomeňme, že se jedná o paměť, která je nonvolatilní (udržuje svůj obsah i při odpojení napájení). K paměti se přistupuje jako k periferii, pomocí určitých vstupně/výstupních registrů.

## GPR (též registrové pole)

Mikrokontroléry ATmega disponují blokem GPR o velikosti 32 registrů šíře 8 bitů. Označují se jako **R0 až R31**. Viz obr. 5.9.

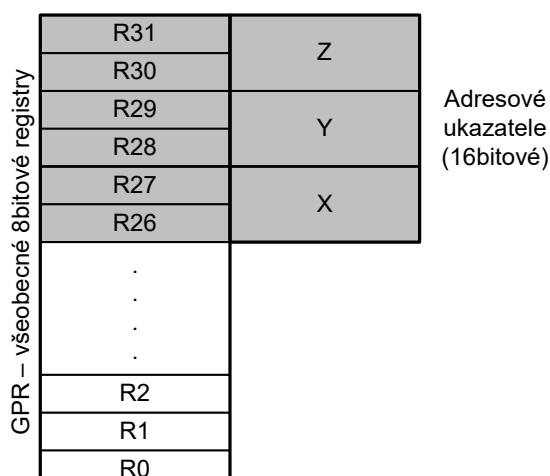
Posledních 6 registrů registrového pole může být použito jako **16bitové ukazatele** při nepřímém adresování. Pomocí nepřímého adresování lze obsáhnout adresní prostor až 64 KB. Označení těchto registrů je: **X, Y a Z** (registrově páry R27:R26, R29:R28 a R31:R30). Viz obr. 5.9.

**Ukazatele** slouží k nepřímému adresování paměti. Adresu paměťového místa, se kterým chceme pracovat, uložíme do registru. Využijeme je například při **blokovém zpracování dat** (přesunu bloku paměti, hledání údaje s určitou hodnotou, apod.). Do ukazatele uložíme adresu prvního paměťového místa, se kterým chceme pracovat. Po každém dílčím kroku je třeba ukazatel posunout na další paměťové místo.

Pro zjednodušení blokových operací je k dispozici rovněž **post-inkrement**. Jedná se o speciální variantu instrukce pracující s ukazatelem, která zajistí automatické zvýšení ukazatele po jejím provedení.

Též lze použít instrukci provádějící **pre-dekrement**, která zajistí snížení ukazatele před přístupem do paměti.

Obrovskou výhodou post-inkrementu a pre-dekrementu je skutečnost, že modifikace ukazatele proběhne v rámci instrukce nepřímého adresování. Není tedy potřebný další instrukční cyklus na vykonání instrukce, která by obsah ukazatele měnila „ručně“.



Obr. 5.9 Mapa GPR (resp. registrového pole)

## ALU a stavový registr SREG

Mikrokontroléry řady ATmega disponují 8 bitovou ALU (aritmeticko-logickou jednotkou), která zajišťuje realizaci operací nad celými čísly bez znaménka nebo se znaménkem (v doplňkovém kódu). Kromě jiného disponuje i instrukcemi pro násobení a podporuje aritmetiku **fractional** (počítání s čísly se zlomkovou částí v pevné řádové čárce).

Výsledek operace ALU indikují příznaky uložené ve stavovém registru **SREG**, viz obr. 5.10.

Bit	7	6	5	4	3	2	1	0
Čtení/zápis Výchozí hodnota	I 0	T 0	H 0	S 0	V 0	N 0	Z 0	C 0

Obr. 5.10 Stavový registr SREG

Význam jednotlivých bitů stavového registru **SREG**:

- **C – příznak přetečení.**
- **Z – příznak nulového výsledku.**
- **N – příznak negativního výsledku.** Indikuje záporný výsledek po aritmetické nebo logické operaci.
- **V – příznak přeplnění čísla v druhém doplňku.** Podporuje aritmetiku dvojkového doplňku.
- **S – znaménkový bit.** Indikuje znaménko čísla v dvojkovém doplňku ( $S = N \text{ xor } V$ ).
- **H – pomocný příznak přetečení.** Indikuje přenos mezi dolní a horní polovinou výsledku (důležité zejména při práci s BCD čísly).
- **T – kopírovací bit.** Instrukce **BLD** a **BST** používají bit T jako zdrojový nebo cílový bit. Bit z registru GPR může být kopirován do bitu T instrukcí **BST** a zase uložen do bitu registru GPR instrukcí **BLD**.
- **I – globální povolení přerušení.** Pro příjem přerušení musí být tento bit nastaven ( $I = 1$ ). Individuální přerušení lze řídit dalšími registry.

### 5.3 Instrukční soubor AVR – úvod

Než přistoupíme k popisu instrukcí, je třeba uvést symboly, které budeme používat. Tab. 5.2 shrnuje zavedené symboly pro zápis instrukcí. Tab. 5.3 zase ukazuje symboly pro vysvětlení funkce instrukcí.

Tab. 5.2 Zavedené symboly pro zápis operandů instrukcí

Symbol	Výraz
K6	6bitová konstanta (0 až 63)
K8	8bitová konstanta (0 až 255)
off6	6bitový offset (0 až 63)
s	bit registru SREG (0 až 7)
b	bit registru registrového pole nebo vstupu/výstupu (0 až 7)
Rr	zdrojový registr registrového pole (není-li uvedeno jinak, je r v rozsahu 0 až 31)
Rd	cílový registr registrového pole (není-li uvedeno jinak, je d v rozsahu 0 až 31)
P	vstupně/výstupní registr (není-li uvedeno jinak, je P v rozsahu 0 až 63)
addr16	16bitová adresa buňky v datové paměti
rel17	relativní 7bitová adresa (-64 až +63) v paměti programu
rel12	relativní 12bitová adresa (-2048 až +2047) v paměti programu
abs16	absolutní 16bitová adresa (0 až 65535) v paměti programu

Tab. 5.3 Symboly zavedené pro výklad instrukcí

Symbol	Výraz
$\leftarrow$	přiřazení
$\leftrightarrow$	výměna
[ ]	obsah paměťového místa (adresa je uvedena v závorce)
.	bit registru (číslo bitu je uvedeno za tečkou)
low	nižší polovina
high	vyšší polovina
and	logický součin
or	logický součet
xor	výlučný logický součet

## Instrukce přesunů dat

Výklad instrukcí zahájíme nejobjemnější skupinou instrukcí přesunů dat. Jedná se však o nejjednodušší instrukce. Tyto instrukce **neovlivňují žádný z příznaků registru SREG** (vyjma případu zápisu do registru SREG nebo práce s bity tohoto registru).

Tab. 5.4 Instrukce přesunů dat (MOV, LDI, LD, LDD, LDS)

Kód	Operandy	Popis	Operace	Příznaky	Slova /IC
MOV	Rd, Rr	přesun mezi registry	Rd $\leftarrow$ Rr	–	1/1
LDI	Rd, data8	nahrání přímé hodnoty, <b>d je v rozsahu 16 až 31</b>	Rd $\leftarrow$ K8	–	1/1
LD	Rd, X	nahrání dat adresovaných nepřímo přes X	Rd $\leftarrow$ [X]	–	1/2
LD	Rd, X+	nahrání dat adresovaných nepřímo přes X, post-inkrement	Rd $\leftarrow$ [X], X $\leftarrow$ X+1	–	1/2
LD	Rd, -X	nahrání dat adresovaných nepřímo přes X, pre-dekrement	X $\leftarrow$ X-1, Rd $\leftarrow$ [X]	–	1/2
LD	Rd, Y	nahrání dat adresovaných nepřímo přes Y	Rd $\leftarrow$ [Y]	–	1/2
LD	Rd, Y+	nahrání dat adresovaných nepřímo přes Y, post-inkrement	Rd $\leftarrow$ [Y], Y $\leftarrow$ Y+1	–	1/2
LD	Rd, -Y	nahrání dat adresovaných nepřímo přes Y, pre-dekrement	Y $\leftarrow$ Y-1, Rd $\leftarrow$ [Y]	–	1/2
LD	Rd, Z	nahrání dat adresovaných nepřímo přes Z	Rd $\leftarrow$ [Z]	–	1/2
LD	Rd, Z+	nahrání dat adresovaných nepřímo přes Z, post-inkrement	Rd $\leftarrow$ [Z], Z $\leftarrow$ Z+1	–	1/2
LD	Rd, -Z	nahrání dat adresovaných nepřímo přes Z, pre-dekrement	Z $\leftarrow$ Z-1, Rd $\leftarrow$ [Z]	–	1/2
LDD	Rd, Y+off6	nahrání dat adresovaných nepřímo přes Y a posunutí	Rd $\leftarrow$ [Y+off6]	–	1/2
LDD	Rd, Z+off6	nahrání dat adresovaných nepřímo přes Z a posunutí	Rd $\leftarrow$ [Z+off6]	–	1/2
LDS	Rd, addr16	nahrání přímo adresovaných dat	Rd $\leftarrow$ [addr16]	–	2/2

Nejjednodušší formou je instrukce MOV Rd, Rr, která **kopíruje data mezi dvěma registry registrového pole**. Hodnota uložená v registru Rr (r jako source) se překopíruje do registru Rd (d jako destination). Obsah registru Rr se pochopitelně nezmění.

Velmi často je používána instrukce LDI Rd, K8. Slouží k **nastavení registru Rd přímou hodnotou (konstantou) K8**. Do strojového kódu instrukce musí kromě operačního kódu „vejít“ i číslo registru a 8 bitů představujících konstantu. Proto nelze adresovat celé registrové pole, ale pouze jeho horní část. Pro d tedy platí, že je v rozsahu pouze **16 až 31**.

Větší skupinu tvoří různé formy instrukce LD. Používají se pro **nepřímé adresování datové paměti**. Adresa je uložena v jednom z ukazatelů X, Y, Z. Ukazatel se může po operaci resp. před operací posunout na následující resp. předchozí adresu. Hodnota přečtená z paměti se uloží do registru Rd.

Podobná je i instrukce LDD. Opět se jedná o **nepřímé adresování pomocí ukazatelů Y, Z**. Tato nepřímá adresa je doplněna přímým 6bitovým posunutím off6 (tedy v rozsahu 0 až 63). Tímto způsobem lze adresovat data z tabulky, která

v paměti začíná na adrese uložené v ukazateli. Posunutí pak určuje položku tabulky, kterou chceme nahrát. Hodnota přečtená z paměti se uloží do registru Rd.

Instrukce `LDS Rd,addr16` se používá pro **přímé adresování celého adresního prostoru**. Operand `addr16` uvádí plnou adresu paměťového místa v paměti dat, jejíž hodnota se načte do registru Rd. Tato instrukce má délku dvou slov.

*Tab. 5.5 Instrukce přesunů dat (ST, STD, STS, LPM, IN, OUT)*

Kód	Operandy	Popis	Operace	Příznaky	Slova /IC
ST	X, Rr	uložení dat adresovaných nepřímo přes X	[X] ← Rr	–	1/2
ST	X+, Rr	uložení dat adresovaných nepřímo přes X, post-inkrement	[X] ← Rr, X ← X+1	–	1/2
ST	-X, Rr	uložení dat adresovaných nepřímo přes X, pre-dekrement	X ← X-1, [X] ← Rr	–	1/2
ST	Y, Rr	uložení dat adresovaných nepřímo přes Y	[Y] ← Rr	–	1/2
ST	Y+, Rr	uložení dat adresovaných nepřímo přes Y, post-inkrement	[Y] ← Rr, Y ← Y+1	–	1/2
ST	-Y, Rr	uložení dat adresovaných nepřímo přes Y, pre-dekrement	Y ← Y-1, [Y] ← Rr	–	1/2
ST	Z, Rr	uložení dat adresovaných nepřímo přes Z	[Z] ← Rr	–	1/2
ST	Z+, Rr	uložení dat adresovaných nepřímo přes Z, post-inkrement	[Z] ← Rr, Z ← Z+1	–	1/2
ST	-Z, Rr	uložení dat adresovaných nepřímo přes Z, pre-dekrement	Z ← Z-1, [Z] ← Rr	–	1/2
STD	Y+off6, Rr	uložení dat adresovaných nepřímo přes Y a posunutí	[Y+off6] ← Rr	–	1/2
STD	Z+off6, Rr	uložení dat adresovaných nepřímo přes Z a posunutí	[Z+off6] ← Rr	–	1/2
STS	addr16, Rr	uložení přímo adresovaných dat	[addr16] ← Rr	–	2/2
LPM		nahrání konstanty z paměti programu	R0 ← [Flash:Z]	–	1/3
LPM	Rd, Z	nahrání konstanty z paměti programu	Rd ← [Flash:Z]	–	1/3
LPM	Rd, Z+	nahrání konstanty z paměti programu s postinkrementem ukazatele Z	Rd ← [Flash:Z] Z ← Z+1	–	1/3
IN	Rd, P	čtení ze vstupně/výstupního registru (portu)	Rd ← P	–	1/1
OUT	P, Rr	zápis do vstupně/výstupního registru (portu)	P ← Rr	–	1/1

Instrukce ST a STD jsou opakem instrukcí LD a LDD. Hodnota uložená v registru Rr se uloží do datové paměti na adresu, která je obsažena v ukazatelích X, Y, Z. Opět lze používat post-inkrement, pre-dekrement nebo přímé posunutí.

Instrukce LPM **nahrává údaj z programové paměti** (Flash). Adresa je opět určena nepřímo, obsahem registru Z. Připomeňme, že Flash se adresuje po slovech, obsah registru Z ale odpovídá adrese paměťového místa v šíři jednoho bajtu. Takže nejnižší bit registru Z určuje, který z bajtů tohoto slova uvažujeme ( $Z_0 = 0$  – dolní bajt,  $Z_0 = 1$  – horní bajt). Tato možnost je využívána pro vytváření různých tabulek konstant (například pro převody zobrazení apod.).

Velmi důležité jsou instrukce IN a OUT. Ty pracují se vstupně/výstupními registry mikrokontroléru. Přesouvají data mezi vstupně/výstupním registrem a

registrem registrového pole. P je vstupně/výstupní registr (port), Rr je hodnota pro zápis, do Rd je uložena přečtená hodnota.

Poznámky:

Časování instrukcí přesunů je závislé na tom, zda se pracuje s registry nebo s pamětí. Instrukce pracující výhradně nad registry GPR trvají pouze jeden instrukční cyklus. Práce s datovou pamětí je 2cyklová a práce s programovou pamětí pak 3cyklová.

Délka instrukcí je jedno slovo, výjimkou jsou instrukce LDS a STS, které mají délku dvě slova.

### Instrukce bitových operací

Bitové operace pracují nad příznakem registru SREG nebo nad bitem registru registrového pole nebo vstupně/výstupního registru určeným jeho číslem. Dále sem patří instrukce posuvů a rotací.

Instrukce **SBI** a **CBI** nastavují/nuluje bit b (0 až 7) vstupně/výstupního registru (portu) P. Důležité je připomenout, že **P** je v rozsahu **0 až 31**. Tedy, že takto lze ovládat pouze dolní polovinu vstupně/výstupních registrů (6bitové číslo portu, které by adresovalo všechny porty, by se „nevešlo“ do strojového kódu instrukce).

Instrukce **LSL** a **LSR** provádí **logický posuv** registru Rd o jedno místo doleva/doprava. Logickým posuvem se myslí, že vsouvá nula zprava/zleva. Bit, který „vypadne“, je uložen do příznaku C. Viz kapitolu 3.3.

Instrukce **ROL** a **ROR** provádí **rotaci** registru Rd o jedno místo doleva/doprava. Při této operaci se registr Rd spojí s příznakem C do 9bitového registru a všechny bity se posunou o jedno místo doleva/doprava. Viz kapitolu 3.3.

Instrukce **ASR** Rd provádí tzv. **aritmetický posuv** registru Rd doprava (připomeňme, že logický a aritmetický posuv jsou stejné, takže pro realizaci aritmetického posuvu doleva se použije instrukce LSL Rd). Při aritmetickém posuvu doprava se bity posunou o jedno místo doprava, nejnižší bit se vloží do příznaku C. Na místo uvolněného nejvyššího bitu se vloží předchozí hodnota. Takže aritmetický posuv doprava čísla  $01001101_{(2)}$  je  $00100110_{(2)}$ , ale čísla  $11001101_{(2)}$  je  $11100110_{(2)}$ . Je to kvůli aritmetice dvojkového doplňku, viz kapitolu 3.3.

Instrukce **SWAP** Rd provede výměnu horní a dolní čtverice bitů (horního a dolního nibble) registru Rd. Tato operace je výhodná například při rozkladech 8bitových čísel na dvě hexadecimální (šestnáctkové) číslice.

Instrukce **BSET** a **BCLR** nastavují/nuluje zvolený bit stavového registru SREG. Jinou možností je použít instrukce SEC, CLC, SEN, CLN, SEZ, CLZ, SEI, CLI, SES, CLS, SEV, CLV, SET, CLT, SEH a CLH.

Instrukce **BST** a **BLD** slouží pro **přenos bitu** b z registru Rr/příznaku T do příznaku T/registru Rr. Instrukce BST se obvykle použije pro rozklad čísla obsaženého v registru Rr na jednotlivé bity. Podobně instrukcí BLD lze bity postupně složit do podoby čísla, které se objeví v registru Rd.

Tab. 5.6 Instrukce bitových operací

Kód	Operandy	Popis	Operace	Příznaky	Slova /IC
SBI	P, b	nastaví bit <b>b</b> v/v registru <b>P</b> , <b>P je v rozsahu 0 až 31</b>	P.b $\leftarrow$ 1	–	1/2
CBI	P, b	nuluje bit <b>b</b> v/v registru <b>P</b> , <b>P je v rozsahu 0 až 31</b>	P.b $\leftarrow$ 0	–	1/2
LSL	Rd	logický posuv registru <b>Rd</b> doleva	viz kapitola 3.3	Z,C,N,V	1/1
LSR	Rd	logický posuv registru <b>Rd</b> doprava		Z,C,N,V	1/1
ROL	Rd	rotace registru <b>Rd</b> doleva přes <b>C</b>		Z,C,N,V	1/1
ROR	Rd	rotace registru <b>Rd</b> doprava přes <b>C</b>		Z,C,N,V	1/1
ASR	Rd	aritmetický posuv registru <b>Rd</b> doprava		Z,C,N,V	1/1
SWAP	Rd	výměna dolní a horní čtveřice <b>Rd</b>	low(Rd) $\leftarrow$ high(Rd) high(Rd) $\leftarrow$ low(Rd)	–	1/1
BSET	s	nastavení příznaku <b>s</b> registru <b>SREG</b>	SREG.s $\leftarrow$ 1	SREG.s	1/1
BCLR	s	nulování příznaku <b>s</b> registru <b>SREG</b>	SREG.s $\leftarrow$ 0	SREG.s	1/1
BST	Rr, b	přenes bit <b>b</b> registru <b>Rr</b> do příznaku <b>T</b>	T $\leftarrow$ Rr.b	T	1/1
BLD	Rd, b	přenes příznak <b>T</b> do bitu <b>b</b> registru <b>Rd</b>	Rd.b $\leftarrow$ T	–	1/1
SEC		nastavení příznaku <b>C</b>	C $\leftarrow$ 1	C	1/1
CLC		nulování příznaku <b>C</b>	C $\leftarrow$ 0	C	1/1
SEN		nastavení příznaku <b>N</b>	N $\leftarrow$ 1	N	1/1
CLN		nulování příznaku <b>N</b>	N $\leftarrow$ 0	N	1/1
SEZ		nastavení příznaku <b>Z</b>	Z $\leftarrow$ 1	Z	1/1
CLZ		nulování příznaku <b>Z</b>	Z $\leftarrow$ 0	Z	1/1
SEI		nastavení příznaku <b>I</b> , povolení přerušení	I $\leftarrow$ 1	I	1/1
CLI		nulování příznaku <b>I</b> , zákaz přerušení	I $\leftarrow$ 0	I	1/1
SES		nastavení příznaku <b>S</b>	S $\leftarrow$ 1	S	1/1
CLS		nulování příznaku <b>S</b>	S $\leftarrow$ 0	S	1/1
SEV		nastavení příznaku <b>V</b>	V $\leftarrow$ 1	V	1/1
CLV		nulování příznaku <b>V</b>	V $\leftarrow$ 0	V	1/1
SET		nastavení příznaku <b>T</b>	T $\leftarrow$ 1	T	1/1
CLT		nulování příznaku <b>T</b>	T $\leftarrow$ 0	T	1/1
SEH		nastavení příznaku <b>H</b>	H $\leftarrow$ 1	H	1/1
CLH		nulování příznaku <b>H</b>	H $\leftarrow$ 0	H	1/1

### Instrukce logických operací

Logickými operacemi jsou logický součin, součet, výlučný součet a negace (sestavení jednotkového doplňku), nastavení a nulování bitů.

Instrukce AND/ANDI představuje **logický součin** dvou operandů. Výsledek míří do levého operandu. Při logickém součinu jsou oba operandy brány jako jednotlivé bity, takže logický součin proběhne pro všech 8 bitů najednou. Logický součin se nejčastěji používá pro **vymaskování** jednoho bitu nebo celé skupiny bitů. To znamená, že některé bity chceme zamaskovat, tedy vlastně vynulovat (také lze použít instrukci CBR).

Instrukce OR/ORI představuje **logický součet** dvou operandů. Výsledek míří do levého operandu. Při logickém součtu jsou oba operandy brány jako jednotlivé bity, takže logický součin proběhne pro všech 8 bitů najednou. Logický součet se

nejčastěji používá pro nastavení jednoho bitu nebo celé skupiny bitů (také lze použít instrukci SBR).

Instrukce EOR představuje **výlučný logický součet** dvou operandů. Výsledek míří do levého operandu. Při výlučném logickém součtu jsou oba operandy brány jako jednotlivé bity, takže výlučný logický součin proběhne pro všech 8 bitů najednou. Výlučný logický součet se nejčastěji používá pro negování jednoho bitu nebo celé skupiny bitů.

10111010 and 00001111	11100001 or 00100100	11101101 xor 00000001
00001010	11100101	11101100

Obr. 5.11 Výklad logických operací

Instrukce COM provede **negaci všech bitů** zvoleného registru registrového pole. Všechny bity tohoto registru tedy změní své hodnoty na opačné. Tato operace stanoví tzv. **jednotkový doplněk**.

Instrukce SBR a CBR slouží k nastavení nebo nulování několika bitů zvoleného registru registrového pole najednou. Bit, který je v pravém operandu (konstantě) nastaven (roven 1), bude ve výsledku nastaven (SBR) nebo vynulován (CBR). Například konstanta 0x03 nastaví/vynuluje dolní dva bity.

Instrukce TST **stanoví hodnotu příznaků Z a N** tak, aby bylo možné otestovat hodnotu v registru Rd. Pak se tedy dá testovat nulovost nebo znaménko čísla v registru Rd (připomeňme, že tyto příznaky se obvykle nastaví automaticky po provedení aritmetické operace).

Instrukce CLR a SER **zajišťují rychlé vynulování nebo nastavení všech bitů** registru Rd. To je velmi častá operace (není třeba používat instrukci LDI, která pracuje pouze nad horní polovinou registrového pole).

Tab. 5.7 Instrukce logických operací

Kód	Operandy	Popis	Operace	Příznaky	Slova /IC
AND	Rd, Rr	logický součin registrů <b>Rd</b> a <b>Rr</b>	Rd $\leftarrow$ Rd and Rr	Z,N,V	1/1
ANDI	Rd, K8	logický součin registru <b>Rd</b> a konstanty, <b>d je v rozsahu 16 až 31</b>	Rd $\leftarrow$ Rd and K8	Z,N,V	1/1
OR	Rd, Rr	logický součet registrů <b>Rd</b> a <b>Rr</b>	Rd $\leftarrow$ Rd or Rr	Z,N,V	1/1
ORI	Rd, K8	logický součet registru <b>Rd</b> a konstanty, <b>d je v rozsahu 16 až 31</b>	Rd $\leftarrow$ Rd or K8	Z,N,V	1/1
EOR	Rd, Rr	výlučný logický součet registrů <b>Rd</b> a <b>Rr</b>	Rd $\leftarrow$ Rd xor Rr	Z,N,V	1/1
COM	Rd	negace (1. doplněk) registru <b>Rd</b>	Rd $\leftarrow$ 0xFF-Rd	Z,C,N,V	1/1
SBR	Rd, K8	nastaví bity registru <b>Rd</b> podle konstanty, <b>d je v rozsahu 16 až 31</b>	Rd $\leftarrow$ Rd or K8	Z,N,V	1/1
CBR	Rd, K8	nuluje bity registru <b>Rd</b> podle konstanty, <b>d je v rozsahu 16 až 31</b>	Rd $\leftarrow$ Rd and (/K8*)	Z,N,V	1/1
TST	Rd	test nuly nebo záporného čísla v <b>Rd</b>	Rd $\leftarrow$ Rd and Rd	Z,N,V	1/1
CLR	Rd	vynulování registru <b>Rd</b>	Rd $\leftarrow$ Rd xor Rd	Z,N,V	1/1
SER	Rd	nastavení všech bitů registru <b>Rd</b>	Rd $\leftarrow$ 0xFF	-	1/1

\* /K8 značí negaci K8

## 6 Architektura instrukčního souboru AVR II

V této kapitole budeme pokračovat ve výkladu instrukčního souboru, zaměříme se na aritmetické instrukce, instrukce porovnávání, instrukce skoků a přeskoků. Rovněž vysvětlíme manipulaci se zásobníkem.

### 6.1 Instrukce (pokračování)

Po probrání jednodušších instrukcí budeme pokračovat ve výkladu instrukčního souboru.

#### Instrukce aritmetických operací

Aritmetické instrukce umožňují provádění celočíselného součtu, rozdílu, součinu a dalších operací. Při aritmetických operacích se mění velký počet příznaků.

Tab. 6.1 Instrukce aritmetických operací

Kód	Operandy	Popis	Operace	Příznaky	Slova /IC
ADD	Rd, Rr	seče registry <b>Rd</b> a <b>Rr</b>	$Rd \leftarrow Rd + Rr$	Z,C,N,V,H	1/1
ADC	Rd, Rr	seče registry <b>Rd</b> a <b>Rr</b> a příznak <b>C</b>	$Rd \leftarrow Rd + Rr + C$	Z,C,N,V,H	1/1
ADIW	Rd, K6	přičte konstantu ke slovu <b>Rd+1:Rd</b> <b>d jsou hodnoty 24, 26, 28, 30</b>	$Rd+1 : Rd \leftarrow Rd+1 : Rd + K6$	Z,C,N,V,S	1/2
SUB	Rd, Rr	odečte registry <b>Rd</b> a <b>Rr</b>	$Rd \leftarrow Rd - Rr$	Z,C,N,V,H	1/1
SUBI	Rd, K8	odečte od registru <b>Rd</b> konstantu, <b>d je v rozsahu 16 až 31</b>	$Rd \leftarrow Rd - K8$	Z,C,N,V,H	1/1
SBIW	Rd, K6	odečte konstantu od slova <b>Rd+1:Rd</b> <b>d jsou hodnoty 24, 26, 28, 30</b>	$Rd+1 : Rd \leftarrow Rd+1 : Rd - K6$	Z,C,N,V,S	1/2
SBC	Rd, Rr	odečte registry <b>Rd</b> a <b>Rr</b> a příznak <b>C</b>	$Rd \leftarrow Rd - Rr - C$	Z,C,N,V,H	1/1
SBCI	Rd, K8	odečte od registru <b>Rd</b> konstantu a <b>C</b> , <b>d je v rozsahu 16 až 31</b>	$Rd \leftarrow Rd - K8 - C$	Z,C,N,V,H	1/1
INC	Rd	inkrementuje registr <b>Rd</b>	$Rd \leftarrow Rd + 1$	Z,N,V	1/1
DEC	Rd	dekrementuje registr <b>Rd</b>	$Rd \leftarrow Rd - 1$	Z,N,V	1/1
NEG	Rd	dvojkový doplněk registru <b>Rd</b>	$Rd \leftarrow 0x00 - Rd$	Z,C,N,V	1/1
MUL	Rd, Rr	násobení čísel bez znamének	$R1 : R0 \leftarrow Rd * Rr$	Z,C	1/2
MULS	Rd, Rr	násobení čísel se znaménkem	$R1 : R0 \leftarrow Rd * Rr$	Z,C	1/2

Instrukce ADD **Rd, Rr** je **celočíselný součet** dvou registrů registrového pole, výsledek je uložen do levého registru (Rd). Při výpočtu může dojít ke změně všech příznaků (viz obr. 6.1):

- **Z** bude nastaven, je-li výsledek součtu nula.
- **C** bude nastaven, pokud se bude generovat přenos ze sedmého bitu (výsledek přeteče a „nevejde“ se do Rd). V opačném případě je C vynulován.
- **H** bude nastaven, pokud se bude generovat přenos ze třetího bitu (přenos mezi dolní a horní čtveřicí bitů výsledku). V opačném případě je H vynulován.
- **N** je nastaven, je-li výsledek záporný (to nastane, když je nejvyšší bit výsledku nastaven). Tento bit má význam pouze při práci s čísly se znaménkem!
- **V** bude nastaven, pokud se přeplní rozsah zobrazení čísel ve dvojkovém doplňku (čísel se znaménkem). To se projeví změnou znaménka výsledku. Pokud nepracujeme s čísly se znaménkem, nemá V význam!

dvojkově	desítkově bez znaménka	desítkově se znaménkem	C = 0	(nepřeteklo)
01111111	127	+127	H = 1	(přenos mezi 3. a 4. bitem)
+ 00000001	+ 1	+ 1	V = 1	(přeplnění)
<b>10000000</b>	<b>128</b>	<b>-128</b>	N = 1	(záporný výsledek)
			Z = 0	(není nula)

dvojkově	desítkově bez znaménka	desítkově se znaménkem	C = 1	(přeteklo)
10000000	128	-128	H = 0	(nebyl přenos mezi 3. a 4. bitem)
+ 11111111	+ 255	+ -1	V = 1	(přeplnění)
<b>01111111</b>	<b>127</b>	<b>+127</b>	N = 0	(nezáporný výsledek)
			Z = 0	(není nula)

dvojkově	desítkově bez znaménka	desítkově se znaménkem	C = 0	(nepřeteklo)
00100101	37	+37	H = 1	(přenos mezi 3. a 4. bitem)
+ 00011101	+ 29	+ 29	V = 0	(nebylo přeplnění)
<b>01000010</b>	<b>66</b>	<b>+66</b>	N = 0	(nezáporný výsledek)
			Z = 0	(není nula)

Obr. 6.1 Výklad instrukce ADD na příkladech

Instrukce ADC Rd,Rr má **podobnou funkci jako ADD**. V tomto případě je obsah registru Rd sčítán s registrem Rr a s příznakem C. Instrukce je vhodná například na sčítání čísel delších než 8 bitů. Níže uvedený příklad předvádí součet dvou 16bitových čísel (první číslo je v registrech R1 – vyšší bajt a R0 – nižší bajt; druhé číslo je v registrech R3 – vyšší bajt a R2 – nižší bajt; výsledek je uložen do R1 – vyšší bajt a R0 – nižší bajt):

```
ADD R0,R2 ;sečti dolní bajty
ADC R1,R3 ;sečti horní bajty včetně přenosu
```

Instrukce ADIW **přičítá 6bitovou konstantu** (0 až 63) ke slovu tvořenému párem registrů R25:R24, R27:R26, R29:R28 nebo R31:R30. Tato instrukce je tedy speciálně použitelná pro posuv ukazatelových registrů X, Y, Z (připomeňme, že například registr Z odpovídá R31:R30). Takto lze tedy snadno realizovat posunutí ukazatele o relativní vzdálenost až +63 bajtů.

Instrukce SUB Rd,Rr **odečte dva registry** registrového pole od sebe. Výsledek je uložen do levého registru (Rd). Příznaky (viz obr. 6.2):

- **Z** bude nastaven, je-li výsledek rozdílu nula.
- **C** bude nastaven, pokud bude požadována výpůjčka ze sedmého bitu (výsledek podteče a nevezde se do registru). V opačném případě je C vynulován.
- **H** bude nastaven, pokud bude požadována výpůjčka ze třetího bitu (výpůjčka mezi dolní a horní čtvericí bitů). V opačném případě je H vynulován.
- **N** je nastaven, je-li výsledek záporný (to nastane, když je nejvyšší bit výsledku nastaven). Tento bit má význam pouze při práci s čísly se znaménkem!
- **V** bude nastaven, pokud se přeplní rozsah zobrazení čísel ve dvojkovém doplňku (čísel se znaménkem). To se projeví změnou znaménka výsledku. Pokud nepracujeme s čísly se znaménkem, nemá V význam!

dvojkově	desítkově bez znaménka	desítkově se znaménkem	C = 0	(bez výpůjčky)
10000000	128	-128	H = 1	(výpůjčka mezi 3. a 4. bitem)
- 00000001	- 1	- +1	V = 1	(přeplnění)
<b>01111111</b>	<b>127</b>	<b>+127</b>	N = 0	(nezáporný výsledek)
			Z = 0	(není nula)

dvojkově	desítkově bez znaménka	desítkově se znaménkem	C = 1	(výpůjčka)
01111111	127	+127	H = 0	(nebyla výpůjčka mezi 3. a 4. bitem)
- 11111111	- 255	- -1	V = 1	(přeplnění)
<b>10000000</b>	<b>128</b>	<b>-128</b>	N = 1	(záporný výsledek)
			Z = 0	(není nula)

dvojkově	desítkově bez znaménka	desítkově se znaménkem	C = 0	(nebyla výpůjčka)
00100101	37	+37	H = 1	(výpůjčka mezi 3. a 4. bitem)
- 00011101	- 29	- +29	V = 0	(nebylo přeplnění)
<b>00001000</b>	<b>8</b>	<b>+8</b>	N = 0	(nezáporný výsledek)
			Z = 0	(není nula)

Obr. 6.2 Výklad instrukce **SUB** na příkladech

Instrukce **SUBI Rd, Rr** je **podobná instrukci SUB**. Pravý operand rozdílu je nyní představován konstantou (tedy ne obsahem registru). Číslo cílového registru Rd je v rozsahu 16 až 31!

Instrukce **SBIW** **odečítá 6bitovou konstantu** (0 až 63) od slova tvořeného párem registrů R25:R24, R27:R26, R29:R28 nebo R31:R30. Tato instrukce je tedy speciálně použitelná pro posuv ukazatelových registrů X, Y, Z (připomeňme, že například registr Z odpovídá R31:R30). Takto lze tedy snadno realizovat posunutí ukazatele o relativní vzdálenost až -63 bajtů.

Instrukce **SBC** a **SBCI** jsou podobné instrukcím **SUB** a **SUBI**. V rozdílu se navíc objevuje příznak C jako výpůjčka.

Instrukce **INC Rd** a **DEC Rd** slouží k rychlému zvýšení (INC jako inkrement) nebo snížení (DEC jako dekrement) registru o 1. To je programech častá operace. Pozor, instrukce nemění příznaky C, H.

Instrukce **NEG Rd** provádí **dvojkový doplněk** registru Rd. Dvojkový doplněk je vlastně stanovení záporné hodnoty čísla se znaménkem.

Instrukce **MUL Rd, Rr** a **MULS Rd, Rr** stanoví **celočíselný součin** obsahů dvou registrů uvažovaných jako číslo bez znaménka (MUL) nebo číslo se znaménkem (MULS). Výsledek je uložen do registrového páru R1:R0. R1 tedy obsahuje horních 8 bitů součinu a R0 obsahuje dolních 8bitů součinu.

### Instrukce porovnávání

Instrukce porovnávání dovolují porovnat obsahy dvou registrů případně registru a konstanty. Podle výsledku porovnání lze poté program rozvětvit.

Tab. 6.2 Instrukce porovnávání

Kód	Operandy	Popis	Operace	Příznaky	Slova /IC
CP	Rd, Rr	porovnání registrů <b>Rd a Rr</b>	Rd-Rr	Z,C,N,V,H	1/1
CPC	Rd, Rr	porovnání registrů <b>Rd a Rr včetně C</b>	Rd-Rr-C	Z,C,N,V,H	1/1
CPI	Rd, K8	porovnání registru <b>Rd s konstantou, d je v rozsahu 16 až 31</b>	Rd-K8	Z,C,N,V,H	1/1

Instrukce CP Rd, Rr **porovnává registry** Rd a Rr tak, že jejich obsahy vzájemně odečte. Výsledek ale není uložen (což je rozdíl proti instrukci SUB). Takže porovnání nezpůsobí změnu žádného z porovnávaných registrů. Testováním příznaků lze například zjistit, zda jsou čísla stejná (Z = 1) apod.

Podobná je instrukce CPC Rd, Rr. Rozdíl je stanoven jako Rd – Rr – C (uvažuje se výpůjčka).

Instrukce CPI Rd, K8 porovnává obsah registru Rd s konstantou (číslo registru je v rozsahu 16 až 31).

### Instrukce skoků a přeskoků

Instrukce uvedené níže se používají pro změnu programového řízení (přechodu do jiné části programu), dělí se do tří skupin:

- **nepodmíněné skoky (jumps)** – provedou se vždy (jejich provedení není podmíněno splněním nějaké podmínky),
- **podmíněné skoky (branches)** – provedou se jen v tom případě, že je splněna určitá podmínka (například nastavený příznak C), jinak se pokračuje následující instrukcí,
- **přeskoky (skips)** – při splnění určité podmínky přeskočí následující instrukci, jinak se pokračuje následující instrukcí.

Instrukce JMP abs16 je **nepodmíněným skok v absolutním** rozsahu adres 0 až 65535.

Instrukce RJMP rel12 je **relativní nepodmíněný skok**. Provede se vždy (nemusí být splněna žádná podmínka), dosah skoku je –2048 až +2047 pozic vůči aktuálnímu místu programu (což je PC + 1).

Instrukce IJMP je **nepodmíněný nepřímý skok v absolutním** rozsahu adres 0 až 65535. Instrukce nemá operandy, místo skoku je určeno obsahem registru Z.

Instrukce podmíněných skoků jsou **relativní 7bitové skoky**, dosah skoku je –64 až +63 pozic vůči aktuálnímu místu programu (PC). Časová náročnost instrukce závisí na tom, zda se skok zrealizuje (podmínka je splněna) nebo ne. Pokud je podmínka splněna (když se skok provede), trvá 2 cykly. Pokud ne, trvá jeden cyklus.

Podmíněné skoky se tedy provedou pouze v případě splnění určité podmínky. Situace je poměrně složitá, proto je uvedena tab. 6.4, která má zlepšit přehled. Jsou zde uvedeny jednak instrukce porovnávající čísla bez znaménka, čísla se znaménkem a jednoduché testy příznaků z registru SREG. Mezi nejčastěji používané podmíněné skoky patří:

- BREQ (provede skok při Z = 1) – používá se pro indikaci nulového výsledku předchozí operace (například po instrukci CP, která zjistila shodu obou operandů),
- BRNE (provede skok při Z = 0) – používá se pro indikaci nenulového výsledku předchozí operace,

- BRCS (provede skok při C = 1) – používá se pro indikaci přetečení (například po instrukci CP, když platilo Rd < Rr pro čísla bez znaménka),
- BRCC (provede skok při C = 0) – používá se pro indikaci, že nedošlo přetečení.

Tab. 6.3 Instrukce skoků

Kód	Operandy	Popis	Operace	Příznaky	Slova /IC
JMP	abs16	nepodmíněný absolutní přímý skok	PC ← abs16	–	2/3
RJMP	rel12	nepodmíněný relativní skok	PC ← PC + rel12 + 1	–	1/2
IJMP		nepodmíněný absolutní nepřímý skok	PC ← Z	–	1/2
BRBS	s, rel7	provede rel. skok, je-li SREG.s = 1	je-li SREG.s=1, PC ← PC + rel7 + 1	–	1/1 až 2
BRBC	s, rel7	provede rel. skok, je-li SREG.s = 0	je-li SREG.s=0, PC ← PC + rel7 + 1	–	1/1 až 2
BREQ	rel7	provede rel. skok, je-li Z = 1 (výsledek je shoda)	je-li Z=1, PC ← PC + rel7 + 1	–	1/1 až 2
BRNE	rel7	provede rel. skok, je-li Z = 0 (výsledek je rozdíl)	je-li Z=0, PC ← PC + rel7 + 1	–	1/1 až 2
BRCS	rel7	provede rel. skok, je-li C = 1	je-li C=1, PC ← PC + rel7 + 1	–	1/1 až 2
BRCC	rel7	provede rel. skok, je-li C = 0	je-li C=0, PC ← PC + rel7 + 1	–	1/1 až 2
BRSH	rel7	provede rel. skok, je-li C = 1 (větší, rovno)	je-li C=0, PC ← PC + rel7 + 1	–	1/1 až 2
BRLO	rel7	provede rel. skok, je-li C = 0 (menší)	je-li C=1, PC ← PC + rel7 + 1	–	1/1 až 2
BRMI	rel7	provede rel. skok, je-li N = 1 (záporné)	je-li N=1, PC ← PC + rel7 + 1	–	1/1 až 2
BRPL	rel7	provede rel. skok, je-li N = 0 (kladné)	je-li N=0, PC ← PC + rel7 + 1	–	1/1 až 2
BRGE	rel7	provede rel. skok, je-li větší rovno (znaménkově)	je-li N=1, PC ← PC + rel7 + 1	–	1/1 až 2
BRLT	rel7	provede rel. skok, je-li menší než 0 (znaménkově)	je-li N=0, PC ← PC + rel7 + 1	–	1/1 až 2
BRHS	rel7	provede rel. skok, je-li H = 1	je-li H=1, PC ← PC + rel7 + 1	–	1/1 až 2
BRHC	rel7	provede rel. skok, je-li H = 0	je-li H=0, PC ← PC + rel7 + 1	–	1/1 až 2
BRTS	rel7	provede rel. skok, je-li T = 1	je-li T=1, PC ← PC + rel7 + 1	–	1/1 až 2
BRTC	rel7	provede rel. skok, je-li T = 0	je-li T=0, PC ← PC + rel7 + 1	–	1/1 až 2
BRVS	rel7	provede rel. skok, je-li V = 1	je-li V=1, PC ← PC + rel7 + 1	–	1/1 až 2
BRVC	rel7	provede rel. skok, je-li V = 0	je-li V=0, PC ← PC + rel7 + 1	–	1/1 až 2
BRIE	rel7	provede rel. skok, je-li povoleno přerušení	je-li I=1, PC ← PC + rel7 + 1	–	1/1 až 2
BRID	rel7	provede rel. skok, je-li zakázáno přerušení	je-li I=0, PC ← PC + rel7 + 1	–	1/1 až 2

Tab. 6.4 Přehled podmíněných skoků

Čísla bez znaménka					
Test	Log. výraz	Instrukce	Opačný test	Log. výraz	Instrukce
Rd=Rr	Z=1	BREQ	Rd≠Rr	Z=0	BRNE
Rd<Rr	C=1	BRLO/BRCS	Rd≥Rr	C=0	BRSH/BRCC
Rd≤Rr	C or Z=1	BRSH*	Rd>Rr	C or Z=0	BRLO*
Rd>Rr	C or Z=0	BRLO*	Rd≤Rr	C or Z=1	BRSH*
Rd≥Rr	C=0	BRSH/BRCC	Rd<Rr	C=1	BRLO/BRCC

Čísla se znaménkem					
Test	Log. výraz	Instrukce	Opačný test	Log. výraz	Instrukce
Rd=Rr	Z=1	BREQ	Rd≠Rr	Z=0	BRNE
Rd<Rr	(N xor V)=1	BRLT	Rd≥Rr	(N xor V)=0	BRGE
Rd≤Rr	Z or (N xor V)=1	BRGE*	Rd>Rr	Z and (N xor V)=0	BRLT*
Rd>Rr	Z and (N xor V)=1	BRLT*	Rd≤Rr	Z or (N xor V)=0	BRGE*
Rd≥Rr	(N xor V)=0	BRGE	Rd<Rr	(N xor V)=1	BRLT

Jednoduché testy					
Test	Log. výraz	Instrukce	Opačný test	Log. výraz	Instrukce
C	C=1	BRCS	Ā	C=0	BRCC
N	N=1	BRMI	Ā	N=0	BRPL
V	V=1	BRVS	Ā	V=0	BRVC
Z	Z=1	BREQ	Ā	Z=0	BRNE

\* značí, že se operandy (registry Rd a Rr) musí vzájemně vyměnit

Velmi zajímavou skupinou instrukcí jsou tzv. **přeskoky (skips)**. Jedná se o skoky, které při splnění podmínky přeskocí následující instrukci. Pokud podmínka splněna není, je následující instrukce vykonána.

Tab. 6.5 Instrukce přeskoků

Kód	Operandy	Popis	Operace	Příznaky	Slova /IC
CPSE	Rd, Rr	přeskoc při shodě ( <b>Rd = Rr</b> )	je-li Rd=Rr, PC←PC+2/3	–	1/1 až 3
SBRC	Rr, b	přeskoc, je-li bit <b>b</b> registru <b>Rr</b> vynulován	je-li Rr.b=0, PC←PC+2/3	–	1/1 až 3
SBRS	Rr, b	přeskoc, je-li bit <b>b</b> registru <b>Rr</b> nastaven	je-li Rr.b=1, PC←PC+2/3	–	1/1 až 3
SBIC	P, b	přeskoc, je-li bit <b>b</b> v/v <b>P</b> vynulován, <b>P je v rozsahu 0 až 31</b>	je-li P.b=0, PC←PC+2/3	–	1/1 až 3
SBIS	P, b	přeskoc, je-li bit <b>b</b> v/v <b>P</b> nastaven, <b>P je v rozsahu 0 až 31</b>	je-li P.b=1, PC←PC+2/3	–	1/1 až 3

Časové provedení přeskoku je závislé jednak na splnění podmínky, ale také na délce následující instrukce. Pokud není podmínka splněna, trvá zpracování přeskoku jeden instrukční cyklus. Při splnění podmínky (přeskovení následující instrukce), trvá přeskok 2 nebo 3 instrukční cykly (závisí totiž na tom, zda je následující instrukce 1cyklová nebo 2cyklová).

Instrukční soubor mikrokontrolérů AVR obsahuje 5 přeskoků:

- CPSE Rd,Rr porovnává obsahy registrů Rd a Rr. Jsou-li jejich hodnoty stejné, přeskočí se následující instrukce,
- SBRC a SBRS testují určený bit (0/1) registru Rr registrového pole,
- SBIC a SBIS testují určený bit (0/1) vstupně/výstupního registru P (port musí být v rozsahu 0 až 31).

### Speciální instrukce

Instrukce NOP neprovádí žádnou operaci. Pouze zpozdí vykonávání programu o jeden hodinový cyklus. Proto se často používá k časování. Pokud uvážíme hodinový kmitočet mikrokontroléru 10 MHz, bude vykonání této instrukce trvat 100 ns.

Instrukce SLEEP způsobí přechod mikrokontroléru do režimu snížené spotřeby.

Instrukce WDR nuluje obvod WatchDog (speciální obvod dohlížející na korektní vykonávání programu, brání například nechtěnému zacyklení programu).

Tab. 6.6 Speciální instrukce

Kód	Operandy	Popis	Operace	Příznaky	Slova /IC
NOP		prázdná operace, časování	viz popis	–	1/1
SLEEP		přejde do režimu snížené spotřeby	viz popis	–	1/1
WDR		nuluje WDT	viz popis	–	1/1

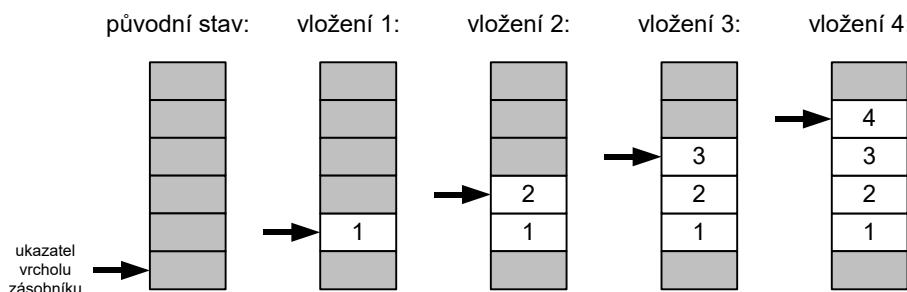
## 6.2 Zásobník (Stack)

Zásobník je obecné označení pro datovou strukturu, která se používá v mnoha algoritmech (například při řazení QuickSort). Je též důležitý pro samotný procesor.

Zopakujme nejdříve klíčové vlastnosti zásobníku zatím bez ohledu na jeho konkrétní implementaci:

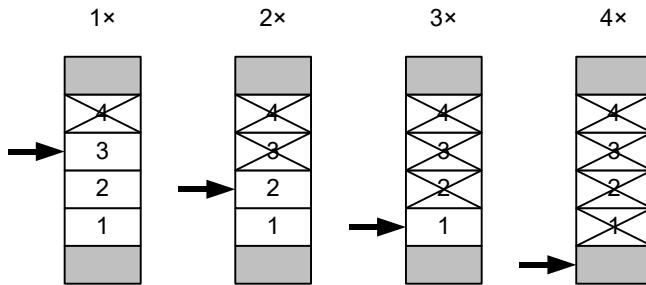
- Prvky se přidávají a odebírají na stejném místě: na **vrcholu zásobníku**.
- Opačný konec zásobníku, se kterým se přímo nepracuje, se nazývá **dno zásobníku**.
- Posledně vložený prvek odebereme jako první. Anglicky se toto chování označuje zkratkou **LIFO** (Last In – First Out, poslední dovnitř – první ven).

Pro lepší představu uvažujme příklad, že do zásobníku budeme postupně vkládat hodnoty: 1, 2, 3, 4. Viz obr. 6.3. Údaje v zásobníku se nepřesunují, posunuje se pouze ukazatel vrcholu zásobníku. V naznačené implementaci je ukazatel nastaven vždy na posledně uložený údaj. Operace vkládání se označuje anglickým slovem **push**.



Obr. 6.3 Příklad vkládání údajů do zásobníku

Víme, že jsme do zásobníku uložili rostoucí posloupnost (1, 2, 3, 4). Při odebírání dříve uložených údajů je zřejmé, že údaje dostáváme v opačném pořadí (4, 3, 2, 1). Otočení pořadí při odebírání údajů je základní vlastností zásobníku. Viz obr. 6.4. Operace odebírání se označuje anglickým slovem **pop**.



Obr. 6.4 Příklad odebírání údajů ze zásobníku

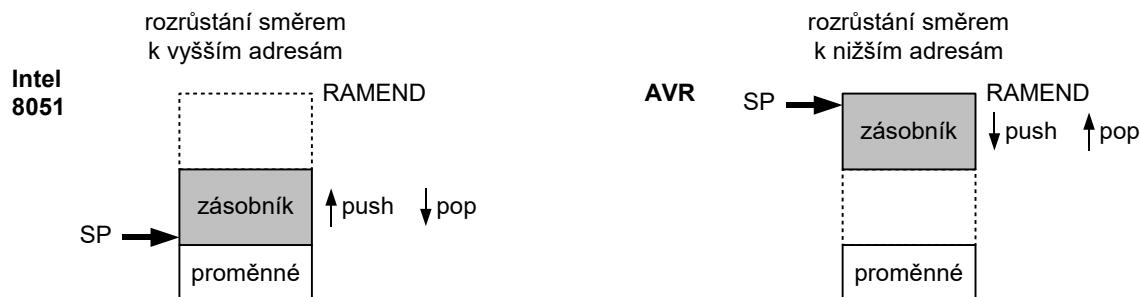
V procesoru je zásobník implementován jako jeho nedílná součást, je nezbytný pro realizaci podprogramů. Obsluha zásobníku je prováděna na hardwarové úrovni.

Zásobník je realizován ve vyhrazené oblasti datové paměti. Vyhrazení prostoru pro zásobník provede při programování v assembleru přímo programátor, ve vyšším programovacím jazyce pak překladač.

**Ukazatel vrcholu zásobníku** je zvláštní registr, obvykle označený jako **SP** (Stack Pointer). Inkrementace resp. dekrementace probíhá hardwarově (automaticky) při práci se zásobníkem.

Skutečná manipulace s registrem SP je dána vnitřní konstrukcí mikroprocesoru, příklady:

- Intel 8051 – zásobník se rozrůstá směrem k vyšším adresám (po vložení údaje se SP inkrementuje):
  - při vkládání se SP nejdříve inkrementuje, údaj se následně uloží na adresu určenou novým obsahem SP,
  - při odebírání se údaj přečte z adresy určené obsahem SP, SP se následně dekrementuje,
  - pro zásobník vyhradíme prostor počínaje první volnou adresou za proměnnými, viz obr. 6.5 vlevo.
- AVR – zásobník se rozrůstá směrem k nižším adresám (po vložení údaje se SP dekrementuje),
  - při vkládání se údaj uloží na adresu určenou obsahem SP, SP se následně dekrementuje,
  - při odebírání se SP nejdříve inkrementuje, údaj se následně přečte z adresy určené novým obsahem SP,
  - pro zásobník vyhradíme prostor na konci paměti, viz obr. 6.5 vpravo.



Obr. 6.5 Vyhrazení prostoru pro zásobník

Využití zásobníku:

- uložení pomocných dat a lokálních proměnných,
- volání podprogramů (ukládání návratové adresy, předávání parametrů pro podprogram přes zásobník),
- obsluha přerušení.

Časté chyby při práci se zásobníkem:

- **Ne inicializovaný zásobník** – u některých procesorů není automaticky zaručeno, že po resetu je SP nastaven na „vhodnou“ adresu. Jednou z prvních akcí po resetu by tedy mělo být správné nastavení SP.
- **Nekonzistence push/pop** – počet operací push a pop musí vždy odpovídat. Pokud například odebereme větší počet údajů, než jsme uložili, budeme čist v podstatě náhodné údaje. Velký pozor je třeba dát při používání podprogramů (viz později).
- **Přetečení/podtečení zásobníku** – v assembleru musí hlídat programátor.

### Implementace zásobníku u AVR

Jak již bylo řečeno, u procesorů řady AVR je zásobník implementován tak, že se rozrůstá směrem k nižším adresám.

V řadě AVR ATmega je registr SP implementován jako 16bitový. Horních 8 bitů představuje registr **SPH**, dolních 8 bitů registr **SPL**. Jedná se o vstupní/výstupní registry.

**Po resetu** je ukazatel vrcholu zásobníku nastaven (obvykle) zcela nevhodně na **SP = 0x0000**. Tedy při vložení prvního údaje dojde k dekrementaci a nový údaj je **SP = 0xFFFF** (podtečení aritmetiky). Dostáváme zcela nevhodnou adresu v paměti dat.

**Proto je nezbytné SP inicializovat**, nejlépe na hodnotu **RAMEND**, která odpovídá poslední platné adrese v paměti dat. Odpovídající kód s použitím operátorů assembleru (HIGH získá horních 8 bitů, LOW získá spodních 8 bitů) je uveden níže:

```
LDI R16,LOW(RAMEND)      ;nahraj dolní část RAMEND do R16
OUT SPL,R16                ;nastav SPL
LDI R16,HIGH(RAMEND)     ;nahraj horní část RAMEND do R16
OUT SPH,R16                ;nastav SPH
```

### Instrukce pro manipulaci se zásobníkem

Základní instrukce pro přímou manipulaci se zásobníkem jsou uvedeny formou tab. 6.7. Kromě toho je se zásobníkem manipulováno při volání podprogramů, vyvolání obsluhy přerušení a návratu z podprogramu resp. obsluhy přerušení. Tyto případy budou vysvětleny později, kapitolách 7.3 a 8.5.

Tab. 6.7 Instrukce pro manipulaci se zásobníkem

Kód	Operandy	Popis	Operace	Příznaky	Slova /IC
PUSH	Rr	uložení do zásobníku	[SP] ← Rr SP ← SP - 1	-	1/2
POP	Rd	vyzvednutí ze zásobníku	SP ← SP + 1 Rd ← [SP]	-	1/2

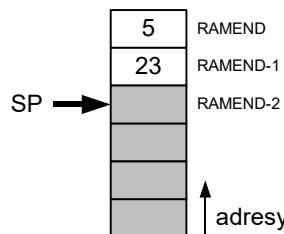
Instrukce PUSH Rr uloží obsah registru Rr na vrchol zásobníku, následně se ukazatel SP dekrementuje (posune na nižší adresu).

Instrukce `POP Rd` vyzvedne ze zásobníku údaj a nahraje jej do registru `Rd`. Před vyzvednutím údaje se ukazatel `SP` inkrementuje (posune se na vyšší adresu).

Pro lepší pochopení doplňujeme krátký příklad použití instrukcí `PUSH` a `POP`. Předpokládejme, že `SP` byl nastaven na hodnotu `RAMEND`. Dále uvažujme: **`R0 = 5`**, **`R10 = 23`**. Nejdříve provedeme uložení obsahů registrů `R0` a `R10` do zásobníku:

```
PUSH R0
PUSH R10
```

Stav zásobníku po vložení údajů je zřejmý z obr. 6.6, `SP` zůstal nastaven na adresu `RAMEND-2`.

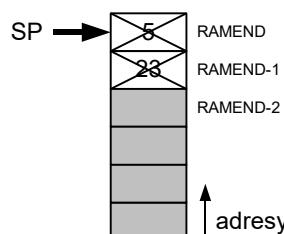


Obr. 6.6 Stav zásobníku po vložení dvou bajtů

Nyní hodnoty uložené v zásobníku vyjmeme zpět do registrů `R0` a `R10`:

```
POP R0
POP R10
```

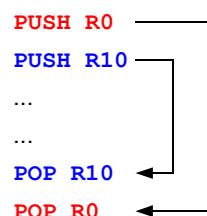
Stav zásobníku po vyzvednutí údajů je zřejmý z obr. 6.7, `SP` zůstal nastaven na adresu `RAMEND`.



Obr. 6.7 Stav zásobníku po vyzvednutí dvou bajtů

Je zřejmé, že po provedení této operace platí: **`R0 = 23`**, **`R10 = 5`**. Tím jsme dosáhli výměny obsahu obou registrů (pokud jsme takovou operaci opravdu zamýšleli).

V případě, že výměnu obsahů registrů nežádáme, mělo by být pořadí instrukcí při vyzvedávání ze zásobníku obrácené. Správné párování ukazuje obr. 6.8.



Obr. 6.8 Párování instrukcí `PUSH` a `POP`

## 7 Architektura instrukčního souboru AVR III

V této kapitole se seznámíme se základními vlastnostmi AVR assembleru a jeho direktivami. Dále se dozvímeme, jak se v assembleru používají podprogramy.

### 7.1 AVR assembler

AVR assembler je překladač jazyka symbolických adres určený pro mikrokontroléry řady AVR.

#### Základní vlastnosti

AVR assembler pracuje jako **dvouprůchodový assembler**. V prvním průchodu proběhne náhrada všech direktiv, konstant, návěstí za čísla a vyhodnocení výrazů. Ve druhém průchodu proběhne překlad do strojového kódu.

Existují dvě verze: základní AVR Assembler, rozšířenou syntaxi umožňuje AVR Assembler 2.

AVR assembler je **case insensitive** (nerozlišuje velikost písmen). Tedy například mnemokódy instrukcí lze psát malými i velkými písmeny.

#### Symboly

Pro zápis symbolu lze použít malé a velké znaky anglické abecedy (a až z, A až Z), číslice (0 až 9) a speciální znak (\_). Pro vzájemné odlišení zápisu čísla od symbolu je nutné, aby symbol nezačínal číslicí!

Některé symboly jsou vyhrazeny a proto nemohou být použity uživatelem. Jedná se o mnemokódy instrukcí (například MOV), direktivy assembleru, funkce assembleru (například LOW) a rezervované operandy (například R0 až R31).

Příklady platných symbolů:

```
PI  
Serial_Port_Buffer  
LOC_4096
```

Příklady neplatných symbolů:

1.PROMENNA	; začíná číslicí
ALFA#	# není platným znakem
MOV	; mnemokód
LOW	; funkce assembleru
DSEG	; direktiva assembleru

#### Návěstí (Label)

Návěstí se obvykle používá k označení určitého místa v programu (kam je třeba provést skok) nebo pro „pojmenování“ proměnné.

Návěstí je speciální případ symbolu, řídí se tedy stejnými pravidly. Navíc je však každé návěstí zakončeno dvojtečkou (:). Příklad:

```
CEKEJ:           DEC R19          ; sniž R19  
                 BRNE CEKEJ      ; čekej do vynulování R19
```

#### Komentář

Komentář je libovolná poznámka zapsaná do zdrojového textu. Komentář začíná středníkem (;) a končí koncem řádku. Komentáře assembler ignoruje, slouží pouze pro lepší orientaci v programu.

AVR Assembler 2 povoluje podobný zápis komentáře jako překladače pro C++. Tedy řádkový komentář může být uvozen dvěma lomítky (//). Blokový komentář začíná znaky /\* a končí znaky \*/.

## Zápis literálů

Literál je přímé uvedení hodnoty. Například 56 je celočíselný literál.

**Znakové ASCII literály** se uzavírají mezi apostrofy (''). **Řetězcové ASCII literály** se uzavírají mezi uvozovky (""). Příklad:

LDI R19, 'a' ; do R19 nahraj malé a

**Číselné literály** lze zapisovat ve čtyřech soustavách (se základy: 2, 8, 10 a 16). Výchozí je desítková soustava. Pro určení soustavy, v níž je číslo zadáno, se používají předpony, viz tab. 7.1.

Tab. 7.1 Číselné literály

Číselná soustava	Zápis	Číslice	Příklad zápisu
dvojková (binární)	předchází 0b	0, 1	0b11111010
osmičková (oktalová)	předchází 0	0 až 7	0372
desítková (decimální)	bez vodící 0	0 až 9	250
šestnáctková (hexadecimální)	předchází \$ nebo 0x	0 až 9, A až F	\$fa, 0xfa

## Operátory

Pro sestavení výrazu, který lze vyhodnotit v době překladu, se používají symboly uvedené v tab. 7.2. Prioritu operandů lze měnit pomocí závorek.

Tab. 7.2 Operátory

Symbol	Název	Popis	Priorita
!	logická negace	vrací 1, pokud je výraz 0; vrací 0 pokud je výraz různý od 0	14
~	bitová negace	vrací invertované bity výrazu	14
-	záporné znaménko	otočí znaménko čísla (pro 1 vrátí -1, pro -25 vrátí 25)	14
*	násobení	vynásobí dva operandy	13
/	dělení	podělí celočíselně dva operandy	13
+	sčítání	sečte dva operandy	12
-	odčítání	odečte dva operandy	12
<<	posuv vlevo	posune levý operand o kolik je určeno pravým operandem	11
>>	posuv vpravo	posune levý operand o kolik je určeno pravým operandem	11
<	menší než	je-li levý operand menší než pravý, vrátí 1 (jinak 0)	10
<=	menší nebo rovno	je-li levý operand menší nebo roven pravému, vrátí 1 (jinak 0)	10
>	větší než	je-li levý operand větší než pravý, vrátí 1 (jinak 0)	10
>=	větší nebo rovno	je-li levý operand větší nebo roven pravému, vrátí 1 (jinak 0)	10
==	rovno	jsou-li oba operandy shodné, vrátí 1 (jinak 0)	9
!=	nerovno	jsou-li oba operandy různé, vrátí 1 (jinak 0)	9
&	bitový součin	vrátí bitový součin obou operandů	8
	bitový součet	vrátí bitový součet obou operandů	7
^	výlučný bitový součet	vrátí výlučný bitový součet obou operandů	6
&&	logický součin	vrátí logický součin obou operandů (jsou-li oba operandy různé od 0, vrátí 1)	5
	logický součet	vrátí logický součet obou operandů (jsou-li oba operandy rovny 0, vrátí 0)	4

## Funkce

Pro sestavení výrazu, který lze vyhodnotit v době překladu, lze používat ještě funkce uvedené v tab. 7.3. Výraz se zapisuje do závorek.

Tab. 7.3 Funkce

Funkce	Popis
LOW	vrátí dolní bajt výrazu
HIGH	vrátí druhý bajt výrazu
BYTE1	vrátí první bajt výrazu (jako LOW)
BYTE2	vrátí druhý bajt výrazu (jako HIGH)
BYTE3	vrátí třetí bajt výrazu
BYTE4	vrátí čtvrtý bajt výrazu
LWRD	vrátí bity 0 až 15 výrazu
HWRD	vrátí bity 16 až 31 výrazu
PAGE	vrátí bity 16 až 21 výrazu
EXP2	vrátí mocninu 2 výrazu ( $2^X$ )
LOG2	vrátí celočíselnou část výrazu log2 (logaritmus při základu 2)

Příklady použití operátorů a funkcí:

```
HIGH ($ABCD)           ;výsledek: $AB
LOW ($ABCD)            ;výsledek: $CD
7*4                   ;výsledek: 28
7/4                   ;výsledek: 1
0b1000>>2           ;výsledek: 0b0010
0b1010<<2            ;výsledek: 0b101000
25-17                 ;výsledek: 8
-1                     ;výsledek: 0b11111111111111
                      ;(zobrazí se ve dvojkovém doplňku)
7==4                  ;výsledek: 0
7>4                  ;výsledek: 1
0b1101&0b0111         ;výsledek: 0b0101
0b1001|0b0101         ;výsledek: 0b1101
0b1101^0b0101         ;výsledek: 0b1000
```

## Formáty zápisu řádku ve zdrojovém textu

[návěští:] direktiva [operandy] [komentář]

[návěští:] instrukce [operandy] [komentář]

komentář

prázdný řádek

## 7.2 Direktivy assembleru

Direktivy (pseudoinstrukce, příkazy assembleru) slouží pro vyhrazení paměťového prostoru, uložení konstant do paměti programu, výběr paměťového prostoru, nastavení lokačního čítače, výběr instrukční sady, vložení zdrojového textu z externího souboru.

### .EQU – definice symbolu

.EQU slouží pro symbolické označení literálu nebo pro zavedení nového jména pro stávající symbol. Formát:

.EQU Symbol=výraz

Příklad:

```
.EQU N=10  
LDI R16,N ;R16=10
```

### .DEF – definice symbolického jména registru

.DEF umožňuje odkazovat se na registr pomocí nově zavedeného symbolu.

Formát:

```
.DEF Symbol=registrová hodnota
```

Příklad:

```
.DEF TEMP=R16 ;R16 jako TEMP  
LDI TEMP,10 ;TEMP=10
```

### Výběr segmentu

Jsou definovány 3 segmentové direktivy: CSEG, DSEG a ESEG pro výběr jednoho ze tří paměťových prostorů, viz tab. 7.4. Výchozím segmentem je CSEG.

Tab 7.4 Výběr segmentu

Direktiva	Segment
.CSEG	programový segment (paměť programu)
.DSEG	datový segment (RAM)
.ESEG	segment E <sup>2</sup> PROM

Formát:

```
.CSEG | .DSEG | .ESEG
```

Tyto direktivy definují počátek programového, datového nebo E<sup>2</sup>PROM segmentu. Zdrojový soubor může obsahovat více dílčích segmentů, které se při překladu spojí do jediného cílového segmentu.

Všechny segmenty mají vlastní lokační čítače (slouží pro určení adresy definovaných objektů). V programovém segmentu se data umísťují po slovech, v datovém a E<sup>2</sup>PROM segmentu se data umísťují po bajtech.

### Lokační čítač programového segmentu PC

Hodnotu lokačního čítače pro CSEG udává symbol PC. Typický příklad použití lokačního čítače je při instrukcích skoků:

```
RJMP PC ;nekonečná smyčka
```

### .BYTE – vyhrazení prostoru v bajtech

.BYTE slouží k vyhrazení prostoru v datovém segmentu (RAM). Výraz udává počet bajtů, které chceme pro proměnnou vyhradit. Formát:

```
[Návěští:] .BYTE výraz
```

Lokační čítač se zvýší o hodnotu udanou výrazem (při překročení rozsahu segmentu je hlášena chyba).

Příklad:

```
.EQU TAB_SIZE=10  
.DSEG  
VAR1: .BYTE 1  
TAB: .BYTE TAB_SIZE ; datový segment  
; vyhraď jeden bajt  
; vyhraď TAB_SIZE bajtů
```

### .DB – uložení konstanty do paměti programu

.DB slouží k uložení konstanty (v rozměru bajtu) do Flash nebo E<sup>2</sup>PROM. Proto je tato direktiva použitelná pouze v segmentech CSEG a ESEG.

V případě programové paměti (Flash) se paměť přiděluje po celých slovech. Proto jsou dva po sobě následující bajty uloženy do jednoho slova. Je-li na jednom řádku uveden lichý počet bajtů, je dolní bajt posledního slova naplněn uvedenou hodnotou a horní bajt je vynulován!

Číselné hodnoty lze uvádět v rozsahu: -128 až 255.

Formát:

```
[Návěstí:] .DB výraz [,výraz...]
```

Příklad:

```
.CSEG  
KONST1: .DB 255, 0B01010101, -128, $AA
```

### .DW – uložení konstanty do paměti programu

.DW slouží k uložení konstanty (v rozměru slova) do Flash nebo E<sup>2</sup>PROM. Proto je tato direktiva použitelná pouze v segmentech CSEG a ESEG.

Číselné hodnoty lze uvádět v rozsahu: -32768 až 65535.

Formát:

```
[Návěstí:] .DW výraz [,výraz...]
```

Příklad:

```
.CSEG  
KONST2: .DW 0,255,-128, $FA0C
```

### .ORG – nastavení počátku segmentu

.ORG slouží pro určení hodnoty lokačního čítače v aktuálním segmentu.

Výchozí pozice lokačního čítače je pro segmenty CSEG a ESEG nula (0). Pro segment DSEG je výchozí hodnotou \$60 (přeskočí se registrové pole a vstupní/výstupní registry).

Formát:

```
.ORG výraz
```

Příklady:

```
.DSEG ; start datového seg.  
.ORG $80 ; nastaví lok. čítač  
VAR: .BYTE 1 ; v RAM na adresu $80  
; vyhradí bajt  
; na adrese $80  
.CSEG  
.ORG $10 ; PC=$10  
MOV R0,R1 ; nějaká operace
```

## .INCLUDE – vložení obsahu externího souboru

.INCLUDE vloží do zdrojového souboru obsah diskového souboru s udaným jménem. Tato technika je vhodná pro rozdělení programu do kratších částí nebo pro uložení často používaných podprogramů do zvláštního souboru. Také se takto vkládají soubory s definicí registrů mikrokontroléru. Formát:

```
.INCLUDE "jméno_souboru"
```

Příklady (vložení definice registrů pro určený typ mikrokontroléru):

```
.INCLUDE "m16def.inc" ;ATmega16
.INCLUDE "m16Adef.inc" ;ATmega16A
.INCLUDE "m32def.inc" ;ATmega32
.INCLUDE "m644def.inc" ;ATmega644
.INCLUDE "m644PAdef.inc" ;ATmega644PA
.INCLUDE "m328PBdef.inc" ;ATmega328PB
```

## 7.3 Podprogramy

Podprogramy slouží pro rozdělení programu na menší části, které lze zapsat samostatně. Tím lze zpřehlednit zápis programu. Hlavní část programu se pak obrací na dílčí podprogramy, ve kterých řešíme příslušné detaily. Celkem jako takový je pak přehlednější.

Podprogramy také dovolují opakovaně vyvolávat předepsané sekvence instrukcí, čímž se zkracuje celková délka zdrojového textu.

Rovněž obsluha přerušení je zvláštním příkladem podprogramu.

### Instrukce pro podporu podprogramů

Podprogram se od ostatních řádků programu příliš neliší. Pouze je třeba jej zavolat (pomocí CALL, RCALL nebo ICALL) a potom se z něj vrátit zpět (pomocí RET).

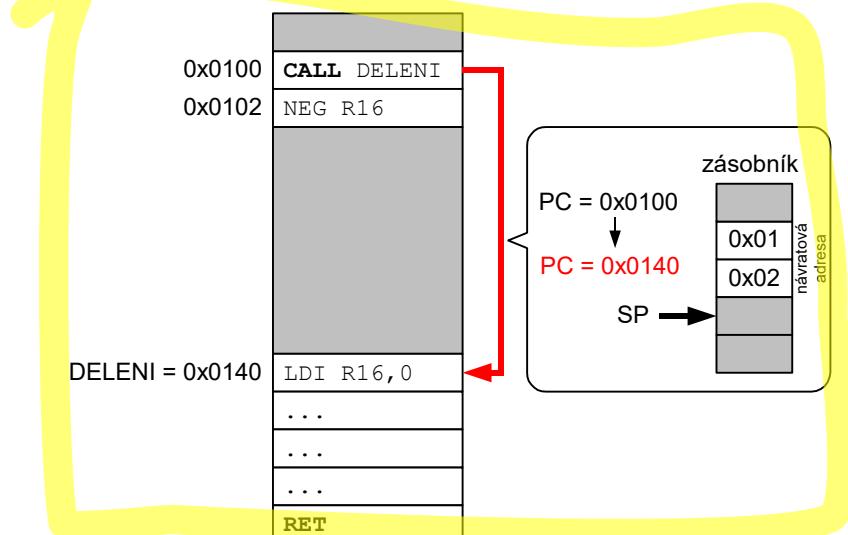
Tab. 7.5 Instrukce pro podporu podprogramů

Kód	Operandy	Popis	Operace	Příznaky	Slova /IC
CALL	abs16	volání podprogramu v rozsahu 64 KS	[SP] ← PC+2 SP ← SP-2 PC ← abs16	–	2/4
RCALL	rel12	relativní volání podprogramu	[SP] ← PC+1 SP ← SP-2 PC ← PC+rel12+1	–	1/3
ICALL		nepřímé volání podprogramu	[SP] ← PC+1 SP ← SP-2 PC ← [Z]	–	1/3
RET		návrat z podprogramu	SP ← SP+2 PC ← [SP]	–	1/4
RETI		návrat z rutiny obsluhy přerušení	SP ← SP+2 PC ← [SP] I=1	I = 1	1/4

Instrukce CALL, RCALL nebo ICALL provádí tzv. **volání podprogramu** a chovají se podobně jako instrukce skoků JMP, RJMP nebo IJMP. Aby se však podprogram mohl vrátit zpět do místa, odkud byl zavolán, musí se nejdříve uložit tzv. **návratová adresa** do zásobníku. Návratová adresa je vlastně adresa instrukce

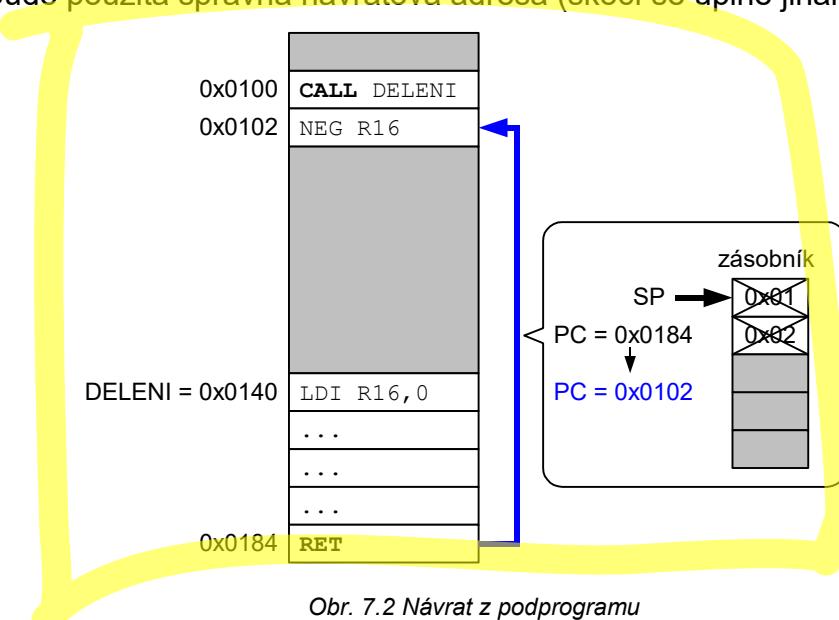
následující za CALL, RCALL nebo ICALL (tuto adresu obsahuje registr PC). Do zásobníku se nejdříve uloží 16bitová návratová adresa, následně se SP sníží o 2. Poté programové řízení přechází na cílovou adresu podprogramu. Viz obr. 7.1.

Rozdíl mezi instrukcemi CALL a RCALL spočívá v „délce“ skoku. CALL je absolutní skok, který obsahne adresy v rozsahu až 64 KS. RCALL provede skok pouze v rozsahu relativní 12bitové adresy. Instrukce ICALL odpovídá nepřímému skoku, cílová adresa je určena obsahem registru Z. Instrukce se rovněž liší délkom v paměti programu a počtem instrukčních cyklů pro její provedení.



Obr. 7.1 Volání podprogramu

Instrukce RET slouží pro **návrat z podprogramu**. Režie spojená s vykonáním této instrukce je nepatrná, protože vše připravila již instrukce CALL, RCALL resp. ICALL. Nejdříve se **SP** zvýší o 2, ze zásobníku se potom vyzvedne návratová adresa a nastaví se do registru PC (tím je proveden návrat). Pozor na správnou manipulaci se zásobníkem! Pokud si do něj na začátku podprogramu uložíte nějakou další hodnotu (kromě návratové adresy) a zapomenete ji vyjmout před použitím instrukce RET, nebude použita správná návratová adresa (skočí se úplně jinam)!



Obr. 7.2 Návrat z podprogramu

Instrukce RETI slouží pro návrat z tzv. obsluhy přerušení. **Obsluha přerušení** je vlastně také podprogram, který vyvolá sám procesor, pokud dojde k přerušení (více o přerušení je uvedeno v kapitole 8). Procesor v tomto případě automaticky zakáže všechna přerušení. Takže jedinou věcí, kterou musí RETI zajistit navíc oproti RET, je povolit přerušení (nastavit bit I z registru SREG).

### Podprogramy s parametry a návratovou hodnotou

Ve vyšším programovacím jazyce mohou mít podprogramy parametry a návratovou hodnotu. Procesor však obvykle nedisponuje instrukcí pro volání podprogramu s parametry ani instrukcí pro návrat z podprogramu pro předávání návratové hodnoty. Tyto detaily musí řešit programátor. Doporučený postup je uveden níže:

1. Dohodnutou oblast paměti naplníme parametry.
2. Vyvoláme podprogram pomocí CALL.
3. Podprogram si hodnoty vyzvedne a zpracuje.
4. Návratovou hodnotu uložíme opět na dohodnuté místo.
5. Provedeme návrat z podprogramu pomocí RET.
6. Po návratu na místo volání vyzvedneme výsledky a zpracujeme.

Jako **oblast pro výměnu dat** mezi volajícím („hlavní program“) a volaným („podprogram“) se obvykle používá:

- zásobník (používá překladač vyššího programovacího jazyka),
- GPR (používá programátor v assembleru).

### 1. Příklad

Uvažujme podprogram **SECTI**, který provede sečtení dvou 16 bitových čísel. První číslo je uloženo vregistrech R1:R0 a druhé číslo vregistrech R3:R2. Výsledek budeme předávat vregistrech R11:R10.

Z hlediska „hlavního programu“ může volání podprogramu SECTI vypadat například takto:

```
... ;naplň R1:R0  
... ;naplň R3:R2  
CALL SECTI ;provedení součtu  
... ;zpracování výsledku
```

Vlastní podprogram by měl být opatřen jakousi hlavičkou, která formou komentářů sděluje důležité informace: jaké registry se používají pro předání vstupních parametrů, v jakých registrech je uložen výsledek, které registry jsou změněny při provádění podprogramu:

```
;SECTI: sečte dvě 16bitová čísla  
;vstup: 1. číslo R1:R0,  
;        2. číslo R3:R2  
;výstup: součet v R11:R10  
;mění:   SREG  
  
SECTI:    MOV R11,R1      ;R11=R1  
              MOV R10,R0      ;R10=R0  
              ADD R10,R2      ;přičte k R10 obsah R2  
              ADC R11,R3      ;přičte k R11 obsah R3 včetně C  
              RET             ;konec
```

## 2. Příklad

Uvažujme podprogram **CEKEJ**, který zajistí vytvoření čekacího intervalu. Podprogram nemá parametry ani návratovou hodnotu, nesmí měnit žádné registry.

Z hlediska „hlavního programu“ může volání podprogramu CEKEJ vypadat například takto:

```
...  
CALL CEKEJ      ;vyvolání čekací rutiny (1)  
...
```

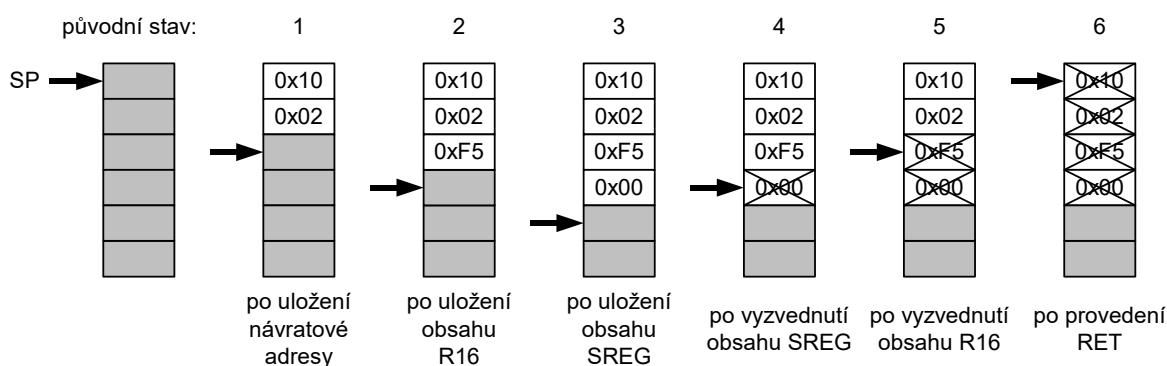
Vlastní podprogram je opět opatřen hlavičkou, obsahy používaných registrů se nejprve uchovají v zásobníku a před koncem se podprogramu se z něj obnoví. Používáme instrukce PUSH a POP. Ty jsou použitelné pro GPR. Registr SREG, jehož obsah se pochopitelně také mění, je vstupně/výstupní registr. Pro něj nejsou instrukce PUSH a POP k dispozici. Hodnotu SREG lze však přesunout pomocí IN do libovolného GPR a podobně z GPR zase přesunout zpět pomocí instrukce OUT:

```
;CEKEJ: zajistí čekání délky (2+1+2+1+100+199+2+1+2+4) × IC  
;vstup: není  
;výstup: není  
;mění: nic (používá zásobník)
```

<b>CEKEJ:</b>	PUSH R16	;uchovej hodnotu R16 (2)
	IN R16, SREG	;R16=SREG
	PUSH R16	;uchovej hodnotu SREG (3)
SMYCKA:		
	LDI R16, 100	;počet průchodů smyčkou
	DEC R16	;sníží R16
	BRNE SMYCKA	;testuje na nulu, při 0 konec
	POP R16	;vyzvedni SREG (4)
	OUT SREG, R16	;SREG=R16
	POP R16	;vyzvedni hodnotu R16 (5)
	<b>RET</b>	;konec (6)

Na obr. 7.3 je komentována manipulace se zásobníkem. Čísla uvedená v závorkách v komentářích zdrojového textu odpovídají jednotlivým okamžikům, kdy se mění obsah zásobníku a ukazatele SP.

Nejdříve (1) je do zásobníku uložena návratová adresa, uvažujeme hodnotu 0x1002. Následně je vyvolán podprogram a obsah registru R16 je uložen (2) do zásobníku (uvažujeme hodnotu 0xF5). Nakonec (3) je uložen obsah registru SREG (uvažujeme hodnotu 0x00).



Obr. 7.3 Manipulace se zásobníkem

Po provedení vlastní čekací smyčky se ze zásobníku obnoví hodnota SREG (4) a také R16 (5). Instrukce RET vyzvedne návratovou adresu (6) a řízení běhu programu přechází zpět do „hlavního programu“.

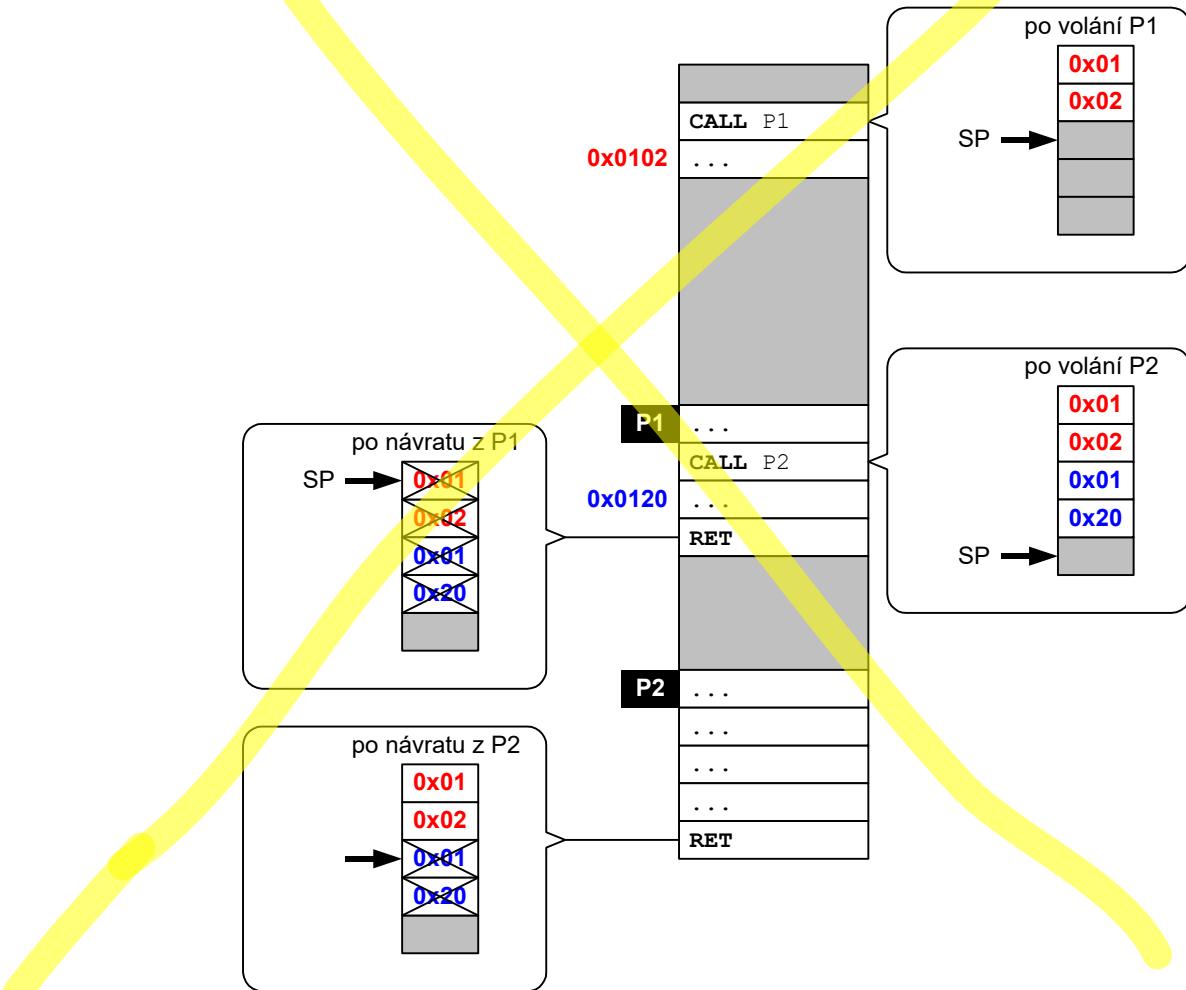
### Vícenásobné vnoření při volání podprogramů

Jelikož je zásobník datová struktura architektury LIFO, podporuje tím vlastně vícenásobné vnoření při používání podprogramů.

Uvažujme, že podprogram označený P1 volá „ze sebe“ jiný podprogram, který je označen jako P2. Situace je zřejmá z obr. 7.4.

Nejdříve voláme z „hlavního programu“ podprogram P1. Návratová adresa do „hlavního programu“ je uložena do zásobníku.

Když v podprogramu P1 vyvoláme jiný podprogram P2, bude návratová adresa do P1 opět uložena do zásobníku.



Obr. 7.4 Manipulace se zásobníkem při vícenásobném vnoření

Při návratu z P2 vyzvedne instrukce RET návratovou adresu do P1 a při návratu z P1 vyzvedne instrukce RET návratovou adresu do „hlavního programu“.

Architektura LIFO tedy zohledňuje skutečnost, že nejdříve se vracíme z posledně vyvolaného podprogramu.

## 8 Základní periferie AVR I

V této kapitole nejdříve vysvětlíme pojem periferní zařízení. Dále popíšeme funkci vstupně/výstupních portů a přerušovacího systému.

### 8.1 Periferní zařízení obecně

Pojmem periferie označujeme zařízení, které není přímou součástí samotného procesoru. Jedná se o vstupní nebo výstupní zařízení, kterým procesor komunikuje s okolím. Můžeme uvést tyto typické příklady periferních zařízení:

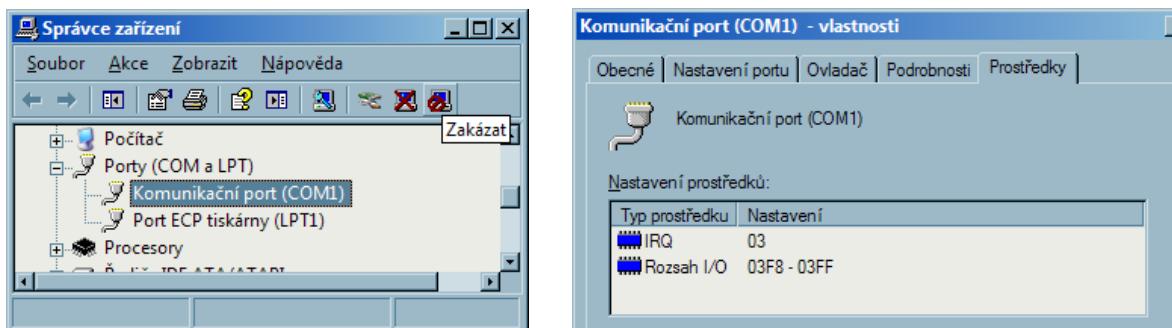
- Klasický počítač:
  - paralelní port (LPTn), sériový port (COMn),
  - zvuková karta, síťová karta,
  - klávesnice, myš,
  - grafická karta.
- Vestavěné (embedded) systémy:
  - vstupně/výstupní porty – digitální vstupy/výstupy,
  - čítače/časovače,
  - A/D převodníky,
  - sériové sběrnice – USART, SPI, I<sup>2</sup>C, atd.

Počítač lze zkonstruovat buď ve formě základní desky, která obsahuje samotný procesor (CPU). Další zařízení se připojují například formou zásuvných karet. Druhou možností je použití mikrokontroléru (MCU), který kromě procesoru již obsahuje periferie integrované v jediném čipu.

### Paměťově mapovaná zařízení

Paměťově mapované zařízení znamená, že je zařízení připojeno na sběrnici procesoru. Obsadí tedy část paměťového prostoru.

Příkladem může být sériový port klasického počítače, který pro své ovládání potřebuje 8 vstupně/výstupních adres (viz obr. 8.1).



Obr. 8.1 Sériový port PC – nastavení prostředků (číslo přerušení a rozsah vstupně/výstupních adres)

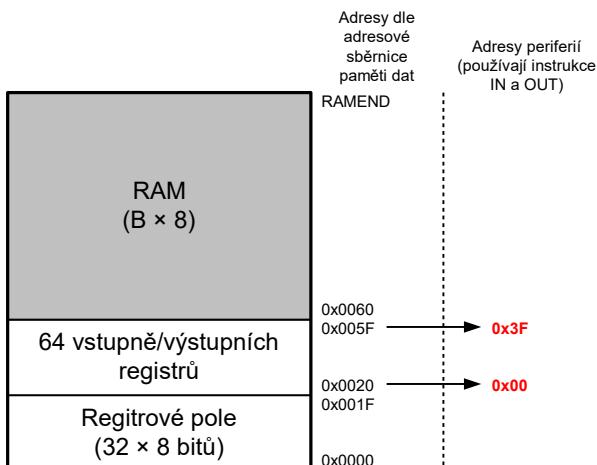
### Základní programovací techniky pro v/v operace

- **vzorkování** – periferie poskytuje na svém výstupu vždy platná data, lze je kdykoli číst,
- **polling** – čekání na stav zařízení v programové smyčce,
- **přerušení** – periferie informuje o změnách stavu a vyžaduje si obsluhu,
- **DMA** – přesuny bloků dat z/do paměti dat na pozadí bez účasti procesoru, upozornění po dokončení celé operace.

## 8.2 Obsluha periferních zařízení AVR

Periferní zařízení se v mikrokontrolérech AVR mapují do bloku 64 vstupně/výstupních registrů na adresách 0x0020 až 0x005F. Viz obr. 8.2.

Pro urychlení přístupu lze v této oblasti používat také instrukce **IN** a **OUT**. Pro zkrácení formátu instrukce je používána posunutá adresa **0x00 až 0x3F**, která odpovídá desítkovému rozsahu 0 až 63 a pro její uložení v instrukci postačí pouze 6 bitů. Vzájemný posuv mezi adresou v paměťovém prostoru datové paměti a adresou periferie je tedy 0x20, což odpovídá délce bloku GPR (32 bajtů).



Obr. 8.2 Prostor adres datové paměti, adresy periferií

V předchozích kapitolách byly probrány instrukce **IN**, **OUT**, **SBI**, **CBI**, **SBIC** a **SBIS**, které pracují se vstupně/výstupními registry. Operand P představuje adresu příslušného vstupně/výstupního registru v rozsahu 0 až 63 (tedy 0x00 až 0x3F). Pro instrukce SBIC a SBIS je rozsah omezen pouze na spodní polovinu (používá se 5bitová adresa), tedy 0 až 31 (0x00 až 1F). Tyto instrukce jsou připomenuty formou tab. 8.1.

Tab. 8.1 Instrukce pro práci s periferiemi

Kód	Operandy	Popis	Operace	Příznaky	Slova /IC
IN	Rd, P	čtení z v/v registru	Rd ← P	–	1/1
OUT	P, Rr	zápis do v/v registru	P ← Rr	–	1/1
SBI	P, b	nastaví bit <b>b</b> v/v registru <b>P</b> , <b>P je v rozsahu 0 až 31</b>	P.b ← 1	–	1/2
CBI	P, b	nuluje bit <b>b</b> v/v registru <b>P</b> , <b>P je v rozsahu 0 až 31</b>	P.b ← 0	–	1/2
SBIC	P, b	přeskoč, je-li bit <b>b</b> portu <b>P</b> vynulován, <b>P je v rozsahu 0 až 31</b>	je-li P.b=0, PC ← PC+2/3	–	1/1 až 3
SBIS	P, b	přeskoč, je-li bit <b>b</b> portu <b>P</b> nastaven, <b>P je v rozsahu 0 až 31</b>	je-li P.b=1, PC ← PC+2/3	–	1/1 až 3

Přístup k příslušnému vstupně/výstupnímu registru je tedy možné provést buď pomocí instrukcí dle tab. 8.1 nebo lze využít faktu, že zařízení jsou mapována v paměti a použít například instrukce **LDS** a **STS**. Vzájemným porovnáním tab. 8.1, 5.4 a 5.5 však zjistíme, že například vykonání instrukce IN nebo OUT zabere jeden instrukční cyklus, ale vykonání instrukce LDS nebo STS zabere dva instrukční cykly (navíc je instrukce dvojnásobné délky).

Instrukce LDS a STS používáme v případě, že jsou některé periferie mikrokontroléra mapovány do prostoru mimo standardní oblast 64 hlavních vstupně/výstupních registrů. Což je případ novějších variant mikrokontrolérů, které jsou vybaveny vyšším počtem periferií. Tehdy nelze instrukce dle tab. 8.1 používat.

Příklad 1: Zápis hodnoty 0x45 do výstupní periferie pomocí OUT:

```
LDI R16, 0x45
OUT IOADRESA, R16
```

Příklad 2: Zápis hodnoty 0x45 do výstupní periferie pomocí STS (adresa musí být zvýšena o 0x20):

```
LDI R16, 0x45
STS IOADRESA+0x20, R16
```

Příklad 3: Čtení ze vstupní periferie pomocí IN do registru R0:

```
IN R0, IOADRESA
```

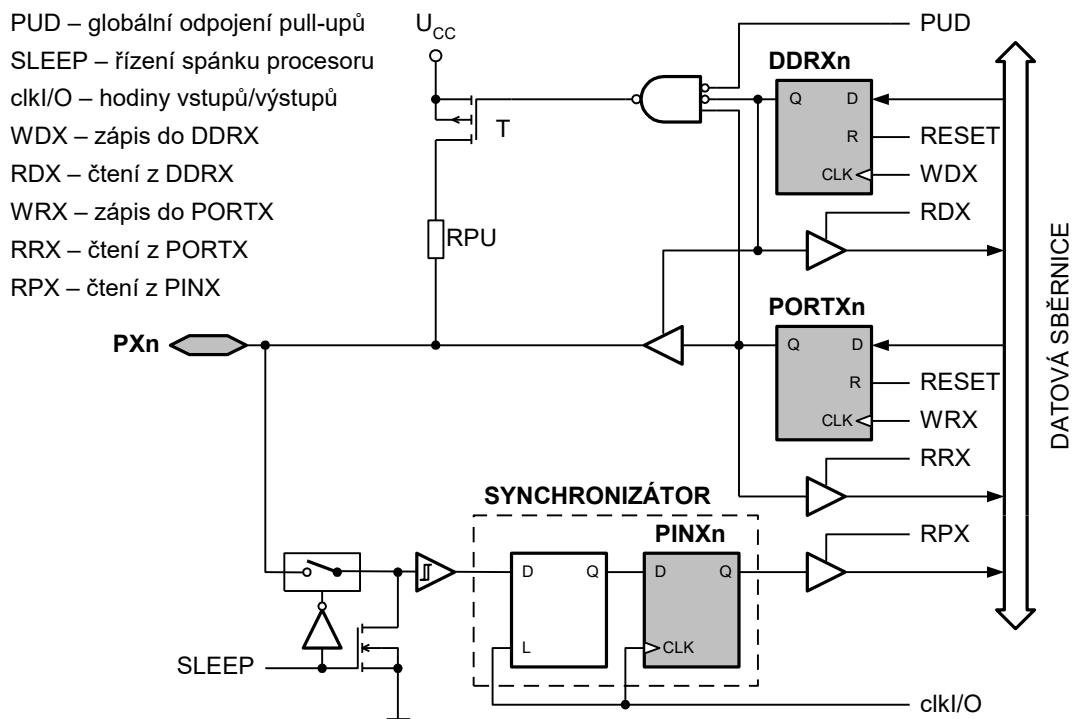
Příklad 4: Čtení ze vstupní periferie pomocí LDS do registru R0 (adresa musí být zvýšena o 0x20):

```
LDS R0, IOADRESA+0x20
```

### 8.3 Vstupně/výstupní porty u AVR ATmega

Mikrokontroléry AVR ATmega disponují obvykle čtyřmi vstupně/výstupními porty, které jsou označeny jako PA až PD. Každý vývod portu lze nakonfigurovat buď jako vstup nebo jako výstup. Vnitřní zapojení jednoho vývodu portu je uvedeno na obr. 8.3.

Po resetu jsou všechny vývody portů konfigurovány jako vstupní (vynuluje se registr **DDRX**). Dále je vynulován registr **PORTX**, takže tyto vstupy mají odpojené pull-up rezistory. Vstupy jsou proti rušení chráněny Schmittovými klopnými obvody.



Obr. 8.3 Zapojení vývodu n portu X

Popis bude proveden na příkladu portu PA:

- První registr ovládá **směr jednotlivých vývodů portu**. Označení je **DDRA** (pro port A). Každý vývod může být tímto registrem nezávisle na ostatních konfigurován jako vstup nebo výstup.
- Druhý registr umožňuje nastavovat logickou hodnotu vývodů, které jsou konfigurovány jako výstupní. Pokud jsou dané vývody konfigurovány jako vstupy, lze připojovat či odpojovat zabudovaný **pull-up rezistor**. Tento rezistor „vytáhne“ výstupní vývod směrem k log. 1 v případě, že není připojen vnější signál. Tímto způsobem se tedy zajišťuje, že je na vývodu definovaná log. 1 i pro případ, že vývod není připojen na vnější obvod. Označení je **PORTA** (pro port A).
- Třetí registr slouží pro čtení stavu vstupních vývodů. Označení je **PINA** (pro port A).

Vliv bitů DDRAn a PORTAn na chování vývodu PAn:

- Je-li **DDRAn = 0** a **PORTAn = 0** je vývod **PAn** konfigurován jako vstup bez rezistoru pull-up (tentot stav odpovídá situaci po resetu mikrokontroléru).
- Je-li **DDRAn = 0** a **PORTAn = 1** je vývod **PAn** konfigurován jako vstup s rezistorem pull-up.
- Je-li **DDRAn = 1** a **PORTAn = 0** je vývod **PAn** výstupní a má log. hodnotu 0.
- Je-li **DDRAn = 1** a **PORTAn = 1** je vývod **PAn** výstupní a má log. hodnotu 1.

**Výstupy jsou typu totem**, to znamená při vybuzení do log. 0 nebo do log. 1 je možno odebírat poměrně značný proud. Výrobce udává, že **maximální proud výstupu je 40 mA a odběr celého obvodu nesmí přesáhnout 200 mA**.

Tab. 8.2 Vliv bitů **DDRXn** a **PORTXn** na chování vývodu **PXn**

<b>DDRXn</b>	<b>PORTXn</b>	<b>bit PUD</b>	<b>vstup/výstup</b>	<b>pull-up</b>	<b>Funkce</b>
0	0	X	vstup	ne	vysokoimpedanční vstup
0	1	0	vstup	ano	vstup s pull-upem
0	1	1	vstup	ne	vysokoimpedanční vstup
1	0	X	výstup	ne	výstup v log. 0
1	1	X	výstup	ne	výstup v log. 1

Obr. 8.3 ukazuje jednotlivé byty výše popsaných vstupně/výstupních registrů pro ovládání portu PA.

Bit	7	6	5	4	3	2	1	0
<b>DDRA</b>	DDRA7	DDRA6	DDRA5	DDRA4	DDRA3	DDRA2	DDRA1	DDRA0
Čtení/zápis	R/W							
Výchozí hodnota	0	0	0	0	0	0	0	0
Bit	7	6	5	4	3	2	1	0
<b>PORTA</b>	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0
Čtení/zápis	R/W							
Výchozí hodnota	0	0	0	0	0	0	0	0
Bit	7	6	5	4	3	2	1	0
<b>PINA</b>	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0
Čtení/zápis	R	R	R	R	R	R	R	R
Výchozí hodnota	?	?	?	?	?	?	?	?

Obr. 8.4 Vstupně/výstupní registry pro řízení portu PA

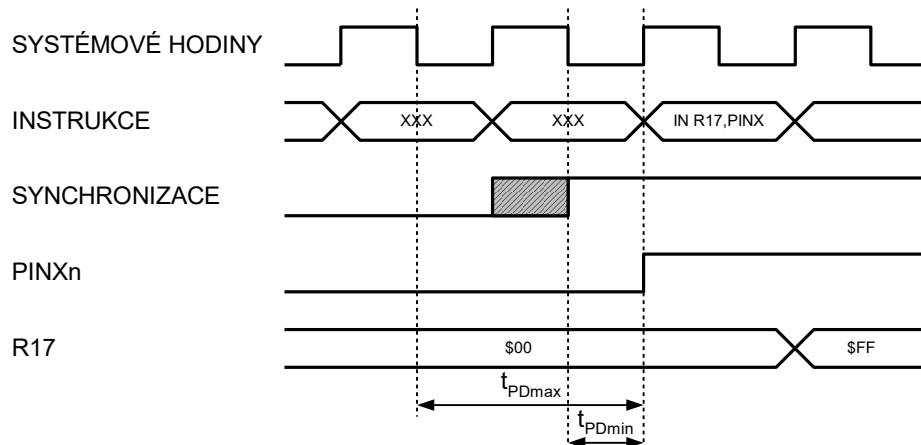
Symbol **R** označuje, že daný bit lze číst. Symbol **W** označuje, že do daného bitu lze zapisovat. **Výchozí hodnota** určuje obsah registru po resetu mikrokontroléru (tedy 0 nebo 1). Označení **?** říká, že výchozí hodnotu nelze předem stanovit.

### Poznámka:

Zpětné čtení obsahu registru **PORTA** vede ke zjištění dříve uložených hodnot. Pro čtení stavu vývodů použijte registr **PINA**.

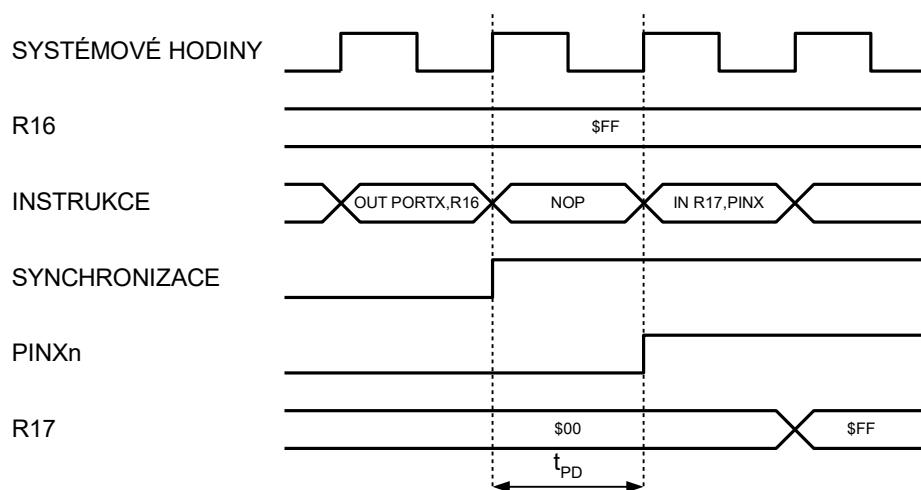
### Synchronizátor

**Synchronizátor** uvedený v obr. 8.3 je důležitý proto, aby se zabránilo poryvům vznikajícím při změně hodnoty vývodu v okamžiku těsně před hranou vnitřních hodin. Na druhou stranu je však nositelem zpoždění. Časový diagram čtení stavu vnějšího signálu připojeného na vývod portu je uveden na obr. 8.5. Maximální a minimální průchozí zpoždění je označeno jako  $t_{PDmax}$  a  $t_{PDmin}$ .



Obr. 8.5 Synchronizace čtení stavu pro případ připojení vnějšího signálu

Uvažujme, že perioda hodin začíná krátce **po** první sestupné hraně systémových hodin. Když jsou hodiny v log. 0, je LATCH zavřen a stane se průchozím pro hodiny v log. 1, jak je vyznačeno šedým obdélníkem **SYNCHRONIZACE**. Hodnota signálu je držena až do doby, než systémové hodiny přejdou do log. 0. Tak je přenesena do registru **PINX** následnou náběžnou hranou hodin. Jak vyznačují kóty  $t_{PDmax}$  a  $t_{PDmin}$ , bude první změna stavu vývodu zpožděna o 0,5 až 1,5 periody systémových hodin.



Obr. 8.6 Synchronizace čtení stavu pro případ programového připojení signálu

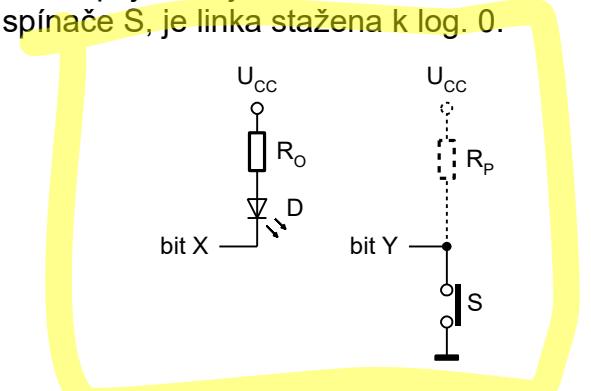
Při zpětném čtení dříve zapsané hodnoty (instrukcí OUT), je nutno vložit instrukci **NOP**. Instrukce **OUT** totiž nastaví signál **SYNCHRONIZACE** náběžnou

hranou hodin. V tomto případě je průchozí zpoždění  $t_{PD}$  rovno jednomu hodinovému cyklu (právě tak dlouho trvá vykonání instrukce NOP).

### Základní vnější obvody

**Nejjednodušší výstupní zařízení je LED.** Obvyklé je buzení LED proti napájecímu napětí. Rozsvícení je tedy možné při log. 0. Toto řešení odpovídá výstupům typu otevřený kolektor.

**Nejjednodušší vstupní zařízení je spínač.** Zapojení vychází z vlastností TTL. Je-li tedy spínač rozpojen, zajistí zdvihací rezistor  $R_P$  vytážení vývodu portu Y do log. 1. Při stisku spínače S, je linka stažena k log. 0.



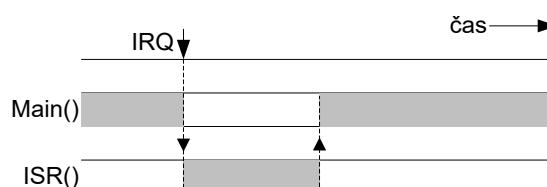
Obr. 8.7 Obvyklé varianty připojení LED a spínače

## 8.4 Přerušení obecně

**Přerušení** (interrupt) je prostředek, kterým může procesor reagovat na stavu periferií. Hlavním důvodem implementace přerušení je požadavek na rychlosť. Procesor je mnohem rychlejší než periferie. Neustále dotazování typu: „jsou k dispozici nová data“ formou pollingu, by procesor zbytečně zatěžovalo.

Vznikne-li požadavek na přerušení (**IRQ** – Interrupt ReQuest), přechází procesor do obsluhy přerušení (**ISR** – Interrupt Service Routine). Přerušení je vlastně asynchronně volaný podprogram (volání provede přímo procesor, po dokončení obsluhy přerušení se vykonávání programu vrací na místo, kde k přerušení došlo).

Z obr. 8.8 je vidět, že běh programu (Main) je přerušen, procesor vykonává obsluhu přerušení (ISR) a po jejím provedení pokračuje na původním místě.



Obr. 8.8 Obsluha přerušení

Obvod pro řízení přerušení se nazývá **řadič přerušení** (Interrupt Controller) u mikrokontrolérů je součástí čipu. Obsluhy přerušení mohou být:

- **Nevektorovatelné** (statické adresy) – při vzniku přerušení přechází programové řízení na pevnou adresu v programové paměti (každý zdroj přerušení může mít zvláštní adresu). Tento způsob používají mikrokontroléry Intel 8051 nebo AVR.
- **Vektorovatelné** – používá se tabulka vektorů přerušení. Při vzniku přerušení se z této tabulky zjistí adresa, na které se nachází obsluha přerušení. Tento způsob používají například procesory Intel 80x86.

## Priorita přerušení

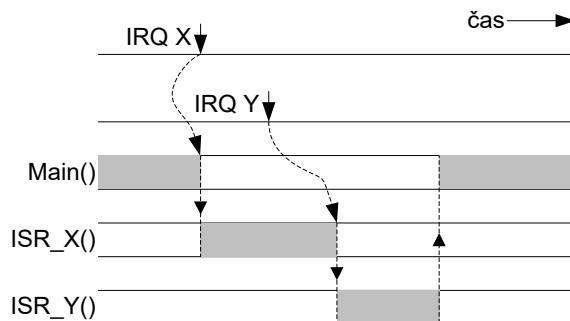
Prioritu přerušení uvažujeme v případě, že je možný souběžný vznik více požadavků na přerušení nebo musí být určitá periferie obslužena přednostně. Priorita přerušení je řešena dvěma způsoby:

- **Pevná priorita** – priorita je určena pořadím adres obsluh přerušení v paměti programu (obvykle přerušení s nejnižší adresou má nejvyšší prioritu).
- **Volitelná priorita** – pomocí speciálního registru lze prioritu volit. Základní variantou je dvouúrovňová priorita (nižší, vyšší). Některé procesory (například ATxmega) podporují prioritu vícestupňovou.

## Vnořené přerušení

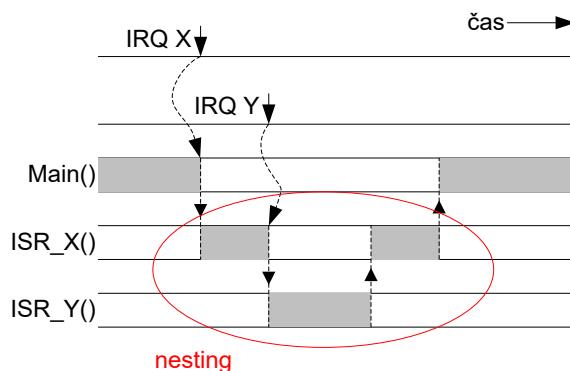
Uvažujme zdroj přerušení X s vysokou prioritou a zdroj přerušení Y s nízkou prioritou. Pokud nastanou požadavky **IRQX** a **IRQY** současně, bude nejdříve obsluženo přerušení X. IRQY se zapamatuje, obsluha proběhne později.

Pokud již probíhá obsluha **ISR<sub>X</sub>**, nebude přerušena při vzniku požadavku **IRQY**. IRQY se zapamatuje a přerušení bude obsluženo později. Viz obr. 8.9.



Obr. 8.9 Obsluha dvou přerušení X a Y (X má vyšší prioritu)

V případě, že je priorita nastavena obráceně (zdroj X má nižší prioritu než zdroj Y), bude situace následující: Pokud probíhá obsluha **ISR<sub>X</sub>**, bude při vzniku požadavku **IRQY** přerušena. Po vykonání **ISR<sub>Y</sub>** se dokončí obsluha **ISR<sub>X</sub>**. Viz obr. 8.10. Tento stav označujeme jako **Interrupt Nesting – vnořené přerušení**.



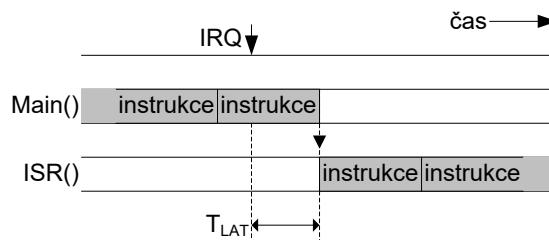
Obr. 8.10 Obsluha dvou přerušení X a Y (X má nižší prioritu)

## Latence (zpoždění) přerušení

Interval od vzniku požadavku na přerušení do vstupu do obsluhy přerušení a vykonání první instrukce obsluhy se označuje jako doba latence  $T_{LAT}$ , viz obr. 8.11. Tento interval není konstantní, je určen jako:  $T_{LAT} = T_{INST} + T_{ZPRAC}$ .

kde:

- $T_{INST}$  – doba provedení instrukce (instrukce se musí dokončit, různé instrukce trvají různou dobu),
- $T_{ZPRAC}$  – doba zpracování požadavku (u mikrokontroléru nehraje roli, u PC ano).



Obr. 8.11 Latence přerušení

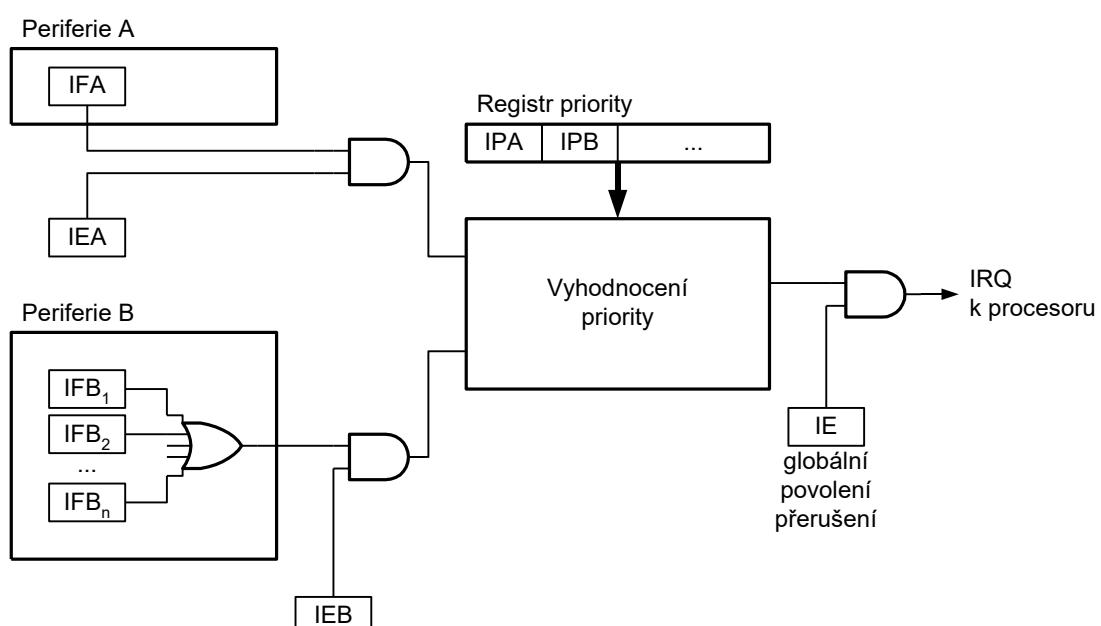
## Typy přerušení

Přerušení lze rozdělit do dvou skupin:

- a) **Maskovatelná přerušení** – programátor může příjem takového přerušení povolit/zakázat nastavením bitu (Interrupt Enable) v určitém registru. Dále je k dispozici bit, který umožňuje najednou zakázat všechna přerušení (Global Interrupt Enable). Požadavek přerušení je v okamžiku zákazu zapamatován pomocí příznaku.
- b) **Nemaskovatelná přerušení** – NMI (Non-maskable Interrupt) – přerušení nelze zakázat.

## Logika přerušení

Obr. 8.12 vysvětluje zpracování přerušení s ohledem na možné maskování a prioritu.



Obr. 8.12 Logika přerušení

Vznikne-li u **periferie A** požadavek přerušení, zapamatuje se v příznaku **IFA** (Interrupt Flag A). Je-li přerušení povoleno pomocí bitu **IEA** (Interrupt Enable A), prochází požadavek přerušení do bloku vyhodnocení priority.

**Periferie B** pracuje podobně. Pouze používá více příznaků přerušení **IFB<sub>1</sub>**, **IFB<sub>2</sub>** až **IFB<sub>n</sub>**, které se pomocí hradla OR „koncentrují“ do jediného požadavku. Tento požadavek může opět blokovat povolovací bit **IEB**. Pokud je přerušení povoleno, přichází požadavek do bloku vyhodnocení priority.

Pomocí bitů určujících prioritu **IPA**, **IPB**, ... (Interrupt Priority) se vyhodnotí priorita přerušení a generuje se požadavek vedený na hradlo AND. Další šíření požadavku na přerušení pak může blokovat globální příznak přerušení **IE**. Pokud je přerušení globálně povoleno, pokračuje požadavek přerušení již přímo na procesor.

Jelikož byly příznaky směřující z periferie B sloučeny do jediného signálu, musí obsluha přerušení pomocí stavu bitů **IFB<sub>1</sub>**, **IFB<sub>2</sub>** až **IFB<sub>n</sub>** teprve testovat příčinu požadavku přerušení.

## 8.5 Přerušení u AVR

Mikrokontroléry AVR disponují výhradně **maskovatelnými přerušeními**.

Každá periferie resp. skupina periferií podobného typu, používá dva registry pro řízení přerušení. Jedná se o registr obsahující byty pro **povolení přerušení** a registr s **příznaky**, ve kterých se udržují požadavky přerušení, která dosud nebyla obsloužena. Všechna přerušení lze tedy **maskovat**. Kromě toho lze všechna přerušení blokovat centrálně pomocí **příznakového bitu I** (viz registr SREG).

Obsluhy přerušení jsou umístěny na **statických adresách** v paměti programu. Pro každou obsluhu je vyhrazeno místo velikosti dvou slov.

**Každý zdroj přerušení používá vlastní obsluhu přerušení.**

**Priorita přerušení je stanovena pevně** pořadím adres obsluh přerušení. Přerušení na nejnižší adrese má nejvyšší prioritu.

Jednotlivé zdroje přerušení pro případ mikrokontroléru ATmega328PB jsou uvedeny formou tab. 8.3.

Sloupec **P** popisuje prioritu přerušení, kde priorita 1 je maximální a priorita 45 je minimální.

Sloupec **Adresa** určuje adresu obsluhy přerušení. Vidíme, že adresa 0x0000 odpovídá první adrese v paměti programu a je chápána jako adresa pro uložení první instrukce programu. Dojde-li k resetu mikrokontroléru, startuje program znova od této adresy.

Sloupec **Symbol** určuje symbolické označení adresy v paměti programu, na kterou je třeba umístit obsluhu daného přerušení. Tyto symboly lze používat pro usnadnění zápisu obsluhy přerušení a snadnou přenositelnost kódu mezi různými typy mikrokontrolérů AVR.

Tab. 8.3 Přerušení u ATmega328PB

P	Adresa	Symbol	Zdroj přerušení
1	0x0000		RESET
2	0x0002	INT0addr	vnější vstup INT0
3	0x0004	INT1addr	vnější vstup INT1
4	0x0006	PCIINT0addr	změna stavu vývodů 0
5	0x0008	PCIINT1addr	změna stavu vývodů 1
6	0x000A	PCIINT2addr	změna stavu vývodů 2
7	0x000C	WDTaddr	přerušení jednotky WDT
8	0x000E	TIMER2_COMPAaddr	čítač/časovač 2, jednotka Output Compare A
9	0x0010	TIMER2_COMPBaddr	čítač/časovač 2, jednotka Output Compare B
10	0x0012	TIMER2_OVFaddr	čítač/časovač 2, přetečení obsahu
11	0x0014	TIMER1_CAPTaddr	čítač/časovač 1, jednotka Input Capture
12	0x0016	TIMER1_COMPAaddr	čítač/časovač 1, jednotka Output Compare A
13	0x0018	TIMER1_COMPBaddr	čítač/časovač 1, jednotka Output Compare B
14	0x001A	TIMER1_OVFaddr	čítač/časovač 1, přetečení obsahu
15	0x001C	TIMER0_COMPAaddr	čítač/časovač 0, jednotka Output Compare A
16	0x001E	TIMER0_COMPBaddr	čítač/časovač 0, jednotka Output Compare B
17	0x0020	TIMER0_OVFaddr	čítač/časovač 0, přetečení obsahu
18	0x0022	SPI0_STCaddr	SPI jednotka 0, přenos dokončen
19	0x0024	USART0_RXaddr	USART 0, dokončen příjem znaku
20	0x0026	USART0_UDREaddr	USART 0, prázdný vysílací registr
21	0x0028	USART0_TXaddr	USART 0, vysílání dokončeno
22	0x002A	ADCCaddr	A/D převodník
23	0x002C	EE_READYaddr	EEPROM připravena
24	0x002E	ANALOG_COMPAaddr	analogový komparátor
25	0x0030	TWI0addr	TWI jednotka 0
26	0x0032	SPM_Readyaddr	připravena jednotka pro zápis do Flash
27	0x0034	USART0_STARTaddr	detekce startu rámce USART 0
28	0x0036	PCIINT3addr	změna stavu vývodů 3
29	0x0038	USART1_RXaddr	USART 1, dokončen příjem znaku
30	0x003A	USART1_UDREaddr	USART 1, prázdný vysílací registr
31	0x003C	USART1_TXaddr	USART 1, vysílání dokončeno
32	0x003E	USART1_STARTaddr	detekce startu rámce USART 1
33	0x0040	TIMER3_CAPTaddr	čítač/časovač 3, jednotka Input Capture
34	0x0042	TIMER3_COMPAaddr	čítač/časovač 3, jednotka Output Compare A
35	0x0044	TIMER3_COMPBaddr	čítač/časovač 3, jednotka Output Compare B
36	0x0046	TIMER3_OVFaddr	čítač/časovač 3, přetečení obsahu
37	0x0048	CFDaddr	detekce selhání hodin (CFD)
38	0x004A	PTC_EOCaddr	konec převodu jednotky PTC
39	0x004C	PTC_WCOMPaddr	okénkový komparátor jednotky PTC
40	0x004E	SPI1_STCaddr	SPI jednotka 1, přenos dokončen
41	0x0050	TWI1addr	TWI jednotka 1
42	0x0052	TIMER4_CAPTaddr	čítač/časovač 4, jednotka Input Capture
43	0x0054	TIMER4_COMPAaddr	čítač/časovač 4, jednotka Output Compare A
44	0x0056	TIMER4_COMPBaddr	čítač/časovač 4, jednotka Output Compare B
45	0x0058	TIMER4_OVFaddr	čítač/časovač 4, přetečení obsahu

## Zápis obsluhy přerušení

Vzhledem k tomu, že pro každou obsluhu přerušení je vyhrazeno místo dvou slov a s ohledem na skutečnost, že obsluhy přerušení mají statické adresy, je nezbytné používat pro zápis obsluhy přerušení níže doporučený postup. Jiné varianty nejsou možné.

Prostor dvou slov neumožňuje zapsat celou obsluhu přerušení. Situace se řeší tak, že na příslušné adresu zapíšeme instrukci skoku na jiné místo v paměti programu a sem zapíšeme prakticky libovolně dlouhou obsluhu přerušení (jsme omezeni pouze kapacitou paměti programu).

Při používání přerušení musí být jako první ze skoků umístěn skok na adresu 0, který odpovídá inicializaci mikrokontroléru při resetu. V rámci takové inicializace musí být provedeno nastavení ukazatele vrcholu zásobníku **SP**, případně konfigurovány periferie a přerušovací systém.

```
.CSEG

.ORG 0
RJMP RESET ;obsluha resetu

.ORG TIMER0_OVFaddr ;obsluha přetečení č/č 0
RJMP CAS0

RESET: LDI R16,LOW(RAMEND) ;nastavení SP
        OUT SPL,R16
        LDI R16,HIGH(RAMEND)
        OUT SPH,R16
        ...
        ...

CAS0:   ... ;uložení registrů do zásobníku
        ...
        ... ;obnova registrů ze zásobníku
        RETI ;návrat z přerušení
```

Uvedený příklad používá jedinou obsluhu přerušení (přetečení čítače/časovače 0).

Obsluha přerušení musí uchovat obsah používaných registrů v zásobníku a před koncem provést jejich obnovení. Obsluha přerušení musí končit instrukcí **RETI**.

## Instrukce pro podporu přerušení

Pro snazší přehled uvádíme souhrnnou tabulkou instrukcí, které se obvykle používají v souvislosti s přerušením.

Tab. 8.4 Instrukce pro podporu přerušení

Kód	Operandy	Popis	Operace	Příznaky	Slova /IC
SEI		nastavení příznaku I, povolení přerušení	I←1	I = 1	1/1
CLI		nulování příznaku I, zákaz přerušení	I←0	I = 0	1/1
RETI		návrat z rutiny obsluhy přerušení	SP←SP+2 PC←[SP] I=1	I = 1	1/4

Připomeňme, že instrukce **SEI** nastaví bit I z registru SREG. Tím tedy povolí příjem přerušení globálně. Podobně instrukce **CLI** tento bit nuluje, tedy slouží pro rychlé zakázání všech přerušení.

Instrukce **RETI** je poslední instrukcí každé obsluhy přerušení. Připomeňme, že přerušení je vyvoláno podobně jako podprogram. Procesor před přechodem na adresu obsluhy přerušení uloží do zásobníku návratovou adresu. Když je obsluha přerušení dokončena, dojde k vyzvednutí této adresy a návratu do místa, kde byl hlavní program přerušen.

Instrukce RET a RETI se odlišují jednou zásadní skutečností. Po vstupu do obsluhy přerušení dochází automaticky k zakázání přerušení ( $I = 0$ ). Mikrokontroléry AVR tedy nepodporují přímo vnořená přerušení (je to však možné „obejít“, pokud obsluha přerušení sama nastaví  $I = 1$ ). Na konci obsluhy přerušení je tedy třeba přerušení povolit.

Pokud použijeme v obsluze přerušení instrukci RET namísto instrukce RETI, bude přerušení vyvoláno pouze jednou a poté budou všechna přerušení zakázána.

Pokud naopak použijeme v podprogramu instrukci RETI místo RET, dojde při návratu z podprogramu k nechtěnému povolení přerušení.

## 9 Základní periferie AVR II

V této kapitole se seznámíme se základními vlastnostmi a režimy čítače/časovače 0 a také s funkcí vstupu vnějšího přerušení.

### 9.1 Čítač/časovač 0

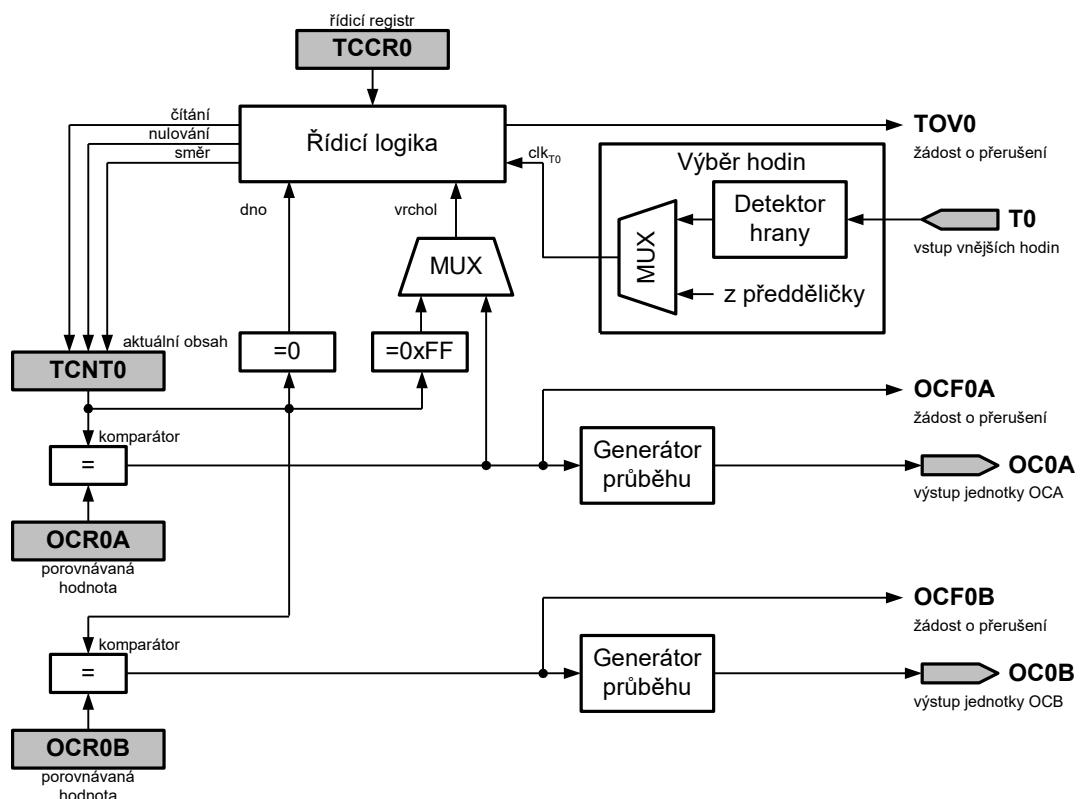
Čítač/časovač je kombinovaná jednotka, která může pracovat jako čítač nebo časovač. **Čítač** je obvod, který počítá impulzy vnějšího signálu. **Časovač** čítá pevný kmitočet, který je odvozen od hodinového signálu mikrokontroléru. Napočítáním určitého počtu impulzů se zajistí odměření časového intervalu.

Jednotka **Output Compare** představuje tzv. komparační registr, jehož obsah je neustále porovnáván s obsahem čítače. Pokud dojde ke shodě (čítač obsahuje stejnou hodnotu jako komparační registr), může být vyvoláno přerušení nebo může nastat určená událost na výstupním vývodu (vývod se například uvede do log. 0 nebo log. 1, případně se jeho stav zneguje).

Čítač/časovač 0 umožňuje čítání impulzů, měření kmitočtu, generování signálu pulzně-šířkové modulace, odměřování časových intervalů atp..

### 9.2 Stručný popis čítače/časovače 0

Schéma zapojení čítače/časovače 0 je uvedeno na obr. 9.1.



Obr. 9.1 Schéma zapojení čítače/časovače 0

Jako zdroj hodin lze použít buď hodinový signál mikrokontroléru **CLK** (lze aktivovat předděličku, hodnoty tedy jsou  $f_0$ ,  $f_0/8$ ,  $f_0/64$ ,  $f_0/256$ ,  $f_0/1024$ ) nebo vnější signál přivedený na vývod **T0** (lze volit polaritu hodin – čítání na náběžnou nebo na sestupnou hranu). Také lze hodiny odpojit, tedy zablokovat čítání (viz tab. 9.1).

Předdělička předřazená hodinovému signálu mikrokontroléru dovoluje realizovat časování v rozsahu zhruba od desítek ms po jednotky  $\mu$ s.

Pokud je použit vnější zdroj hodin (signál přivedený na vývod **T0**), dochází k jeho synchronizaci s hodinovým signálem mikrokontroléru (**CLK**). Protože je vnější hodinový signál vzorkován náběžnou hranou **CLK**, musí být minimální interval mezi dvěma změnami vnějšího hodinového signálu větší než perioda **CLK**. Takže maximální hodnota kmitočtu přivedeného na vstup **T0** je polovinou hodinového kmitočtu mikrokontroléru.

Čítač/časovač 0 má k dispozici dva komparační registry označené jako **OCR0A** a **OCR0B**. Tyto registry lze používat pro generování intervalů zadанé délky, jako generátor kmitočtu nebo PWM jednotku.

### 9.3 Registry čítače/časovače 0

V této kapitole jsou uvedeny řídicí registry čítače/časovače 0.

#### Řídicí registry čítače/časovače 0 – TCCR0A, TCCR0B

Registry **TCCR0A** a **TCCR0B** slouží pro výběr hodinového zdroje čítače/časovače 0, nastavení režimu práce a konfiguraci jednotek Output Compare (viz obr. 9.2).

Bit	7	6	5	4	3	2	1	0
<b>TCCR0A</b>	COM0A1	COM0A0	COM0B1	COM0B0	–	–	WGM01	WGM00
Čtení/zápis	R/W	R/W	R/W	R/W	R	R	R/W	R/W
Výchozí hodnota	0	0	0	0	0	0	0	0
Bit	7	6	5	4	3	2	1	0
<b>TCCR0B</b>	FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00
Čtení/zápis	W	W	R	R	R/W	R/W	R/W	R/W
Výchozí hodnota	0	0	0	0	0	0	0	0

Obr. 9.2 Registry TCCR0A a TCCR0B

Tabulka 9.1 ukazuje, jak bity **CS02**, **CS01** a **CS00** řídí výběr hodinového signálu čítače/časovače 0. Výchozí hodnota (000) odpovídá odstavenému čítači/časovači.

Tab. 9.1 Výběr hodinového signálu pro čítač/časovač 0 registrem TCCR0B

CS02	CS01	CS00	Popis
0	0	0	stop, čítač/časovač 0 je odstaven
0	0	1	$f_0$ (hodinový signál mikrokontroléru CLK)
0	1	0	$f_0/8$ (1/8 CLK)
0	1	1	$f_0/64$ (1/64 CLK)
1	0	0	$f_0/256$ (1/256 CLK)
1	0	1	$f_0/1024$ (1/1024 CLK)
1	1	0	sestupná hrana T0
1	1	1	náběžná hrana T0

Bity **WGM02**, **WGM01** a **WGM00** volí režim práce čítače/časovače 0. Tab. 9.2 obsahuje seznam všech možných režimů. První sloupec (WGM) je desítkově vyjádřené tříbitové číslo dané obsahem bitů **WGM02**, **WGM01** a **WGM00** (například desítková hodnota 4 odpovídá: WGM02 = 1, WGM01 = 0, WGM00 = 0).

Tab. 9.2 Jednotlivé režimy čítače/časovače 0

WGM	WGM02	WGM01	WGM00	Režim	Vrchol	Aktualizace OCR0x	Nastavení příznaku TOV0
0	0	0	0	normální	0xFF	okamžitě	při maximu
1	0	0	1	fázově korigovaný PWM	0xFF	na vrcholu	na dnu
2	0	1	0	CTC	OCR0A	okamžitě	při maximu
3	0	1	1	rychlý PWM	0xFF	na vrcholu	při maximu
4	1	0	0	vyhrazeno (nepoužívat)	—	—	—
5	1	0	1	fázově korigovaný PWM	OCR0A	na vrcholu	na dnu
6	1	1	0	vyhrazeno (nepoužívat)	—	—	—
7	1	1	1	rychlý PWM	OCR0A	na vrcholu	na vrcholu

Dvě dvojice bitů **COM0A0**, **COM0A1** a **COM0B0**, **COM0B1** konfigurují činnost jednotek Output Compare, které jsou vázány na komparační registry **OCR0A** a **OCR0B**. Vysvětlení jejich funkce je uvedeno dále spolu s popisem jednotlivých režimů.

Nastavení bitů **FOC0A**, **FOC0B** umožňuje používat výstupy **OC0A**, **OC0B** mimo PWM režimy.

### Obsah čítače/časovače 0 – TCNT0

Registr **TCNT0** poskytuje přístup k obsahu čítače/časovače 0.

Je třeba poznamenat, že **čítač/časovač 0 čítá nahoru** (ve všech režimech vyjma fázově korigovaného PWM). To znamená, že po každém hodinovém impulzu přivedeném na jeho vstup se obsah zvýší o 1 (například pro první impulz je to změna z 0 na 1).

Pokud registr **TCNT0** obsahuje hodnotu 0xFF = 255 (0b11111111) a přijde-li další hodinový impulz, obsah čítače přeteče (0b00000000) a počítá se od začátku. Při přetečení se nastaví příznak přetečení a může se generovat přerušení (viz dále).

Bit	7	6	5	4	3	2	1	0	LSB
<b>TCNT0</b>	MSB								LSB
Čtení/zápis	R/W								
Výchozí hodnota	0	0	0	0	0	0	0	0	0
Bit	7	6	5	4	3	2	1	0	LSB
<b>OCR0A</b>	MSB								LSB
Čtení/zápis	R/W								
Výchozí hodnota	0	0	0	0	0	0	0	0	0
Bit	7	6	5	4	3	2	1	0	LSB
<b>OCR0B</b>	MSB								LSB
Čtení/zápis	R/W								
Výchozí hodnota	0	0	0	0	0	0	0	0	0

Obr. 9.3 Registry **TCNT0**, **OCR0A**, **OCR0B**

### Komparační registry – OCR0A, OCR0B

Registry **OCR0A**, **OCR0B** jsou komparačními registry čítače/časovače 0. Jejich obsah se nestále porovnává s aktuálním obsahem čítače (**TCNT0**). Při shodě může být vyvolána operace na vývodech **OC0A**, **OC0B**.

### Podpora čítače/časovače 0 v přerušovacím systému

Čítač/časovač 0 může generovat celkem tři přerušení.

Registr **TIMSK0** obsahuje masky těchto přerušení: **TOIE0** (povolení přerušení při přetečení obsahu čítač/časovače 0), **OCIE0A** (povolení přerušení při shodě **TCNT0** = **OCR0A**) a **OCIE0B** (povolení přerušení při shodě **TCNT0** = **OCR0B**).

Registr **TIFR0** obsahuje příznaky těchto přerušení: **TOV0** (indikuje přetečení čítače/časovače 0), **OCF0A** (indikuje shodu TCNT0 = OCR0A) a **OCF0B** (indikuje shodu TCNT0 = OCR0B). Příznaky se nulují automaticky po vstupu do obslužné rutiny přerušení, programové **nulování** je možné **zápisem 1** do příslušného bitu.

Bit	7	6	5	4	3	2	1	0
<b>TIMSK0</b>	—	—	—	—	—	OCIE0B	OCIE0A	TOIE0
Čtení/zápis	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W
Výchozí hodnota	0	0	0	0	0	0	0	0

Obr. 9.4 Registr **TIMSK0**

Bit	7	6	5	4	3	2	1	0
<b>TIFR0</b>	—	—	—	—	—	OCF0B	OCF0A	TOV0
Čtení/zápis	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W
Výchozí hodnota	0	0	0	0	0	0	0	0

Obr. 9.5 Registr **TIFR0**

## 9.4 Popis jednotlivých režimů

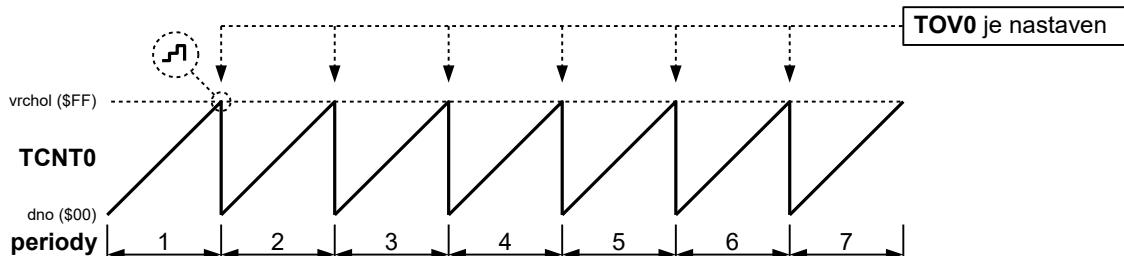
Nyní si vysvětlíme jednotlivé pracovní režimy čítače/časovače 0.

### Normální režim (WGM = 0)

Nejjednoduším režimem je tzv. normální režim.

V tomto režimu se vždy **čítá nahoru** (inkrementace obsahu) a nedochází k žádnému nulování. Čítač jednoduše přeteče, když dosáhne svého 8bitového maxima (vrchol = 0xFF) a restartuje se ze dna (0x00).

Příznak přetečení **TOV0** se nastaví ve stejném hodinovém cyklu čítače, kdy obsah **TCNT0** dosáhl nuly.



Obr. 9.6 Normální režim

### CTC režim (WGM = 2)

V CTC režimu se čítač vynuluje, když **TCNT0** dosáhne hodnoty **OCR0A**. Hodnota OCR0A vlastně definuje hodnotu vrcholu, tedy rozlišení čítače. Tento režim poskytuje větší možnost řízení kmitočtu.

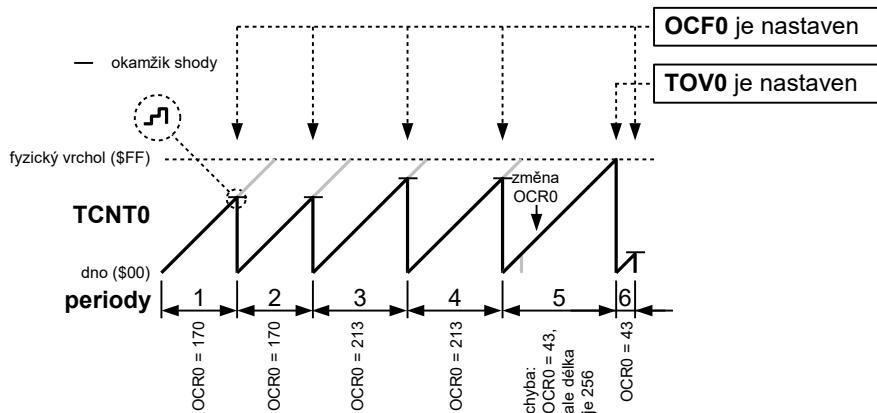
Přerušení se může generovat, když TCNT0 dosáhne vrcholu, který je dán obsahem OCR0A. V tomto případě se nastaví příznak **OCF0**. Je-li přerušení povoleno, vyvolá se.

Na obr. 9.7 jsou zakresleny periody 1 a 2 pro OCR0 = 170, perioda tedy odpovídá 171 cyklům hodin.

Po zvýšení na OCR0 = 213 jsou periody 3 a 4 prodlouženy na 214 cyklů hodin. Naopak snížení OCR0 = 43 by mělo generovat periody 5 a 6 délky 44 cyklů.

Perioda 5 je však generována chybně, změna hodnoty OCR0 proběhla příliš pozdě. Tedy okamžik shody byl minut, čítač přeteče a nastaví příznak TOV0. Tepřve perioda 6 bude generována správně.

Příčinou tohoto chování je skutečnost, že registr OCR0 není dvojitě bufferovaný. Zápis do registru znamená okamžitou aktualizaci.



Obr. 9.7 Režim CTC

Pro generování výstupního průběhu na vývodu OC0X (X je A nebo B) lze zvolit režim **toggle** ( $\text{COM}0\text{X}1 = 0$ ,  $\text{COM}0\text{X}0 = 1$ ). Pak každá shoda  $\text{TCNT}0$  s  $\text{OCR}0\text{X}$  vede k negaci vývodu OC0X, takže se generuje signál střídy 1 : 1. Nejvyšší generovaný kmitočet dostaneme pro případ, že je  $\text{OCR}0\text{X} = 0x00$ . Pro maximální kmitočet platí  $f_{\text{OC}0\text{MAX}} = f_{\text{clk\_IO}}/2$ . Pro ostatní případy:

$$f_{\text{OC}0\text{X}} = \frac{f_{\text{clk\_IO}}}{2 \cdot N \cdot (1 + \text{OCR}0\text{X})} \quad (9-1)$$

kde **N** představuje dělicí poměr (1, 8, 64, 256 nebo 1024).

Podobně jako v normálním režimu je příznak **TOV0** nastaven ve stejném hodinovém cyklu čítače, kdy dojde k přetečení z 0xFF na 0x00.

Tab. 9.3 Význam bitů **COM0X1**, **COM0X0** v režimech bez PWM

<b>COM0X1</b>	<b>COM0X0</b>	Popis
0	0	čítač/časovač 1 odpojen od vývodu OC0X
0	1	negace stavu vývodu OC0X při shodě (toggle)
1	0	vynulování vývodu OC0X při shodě (log. 0)
1	1	nastavení vývodu OC0X při shodě (log. 1)

### Rychlý PWM režim (WGM = 3 nebo 7)

Rychlý PWM režim poskytuje vysokorychlostní generaci PWM průběhu. Rychlý PWM režim se odlišuje od ostatních PWM režimů svou jednofázovou realizací. Čítač čítá ode dna do maxima a po přetečení se vrací opět ke dni.

Hodnota vrcholu je pro WGM = 3 pevná (0xFF) a pro WGM = 7 nastavitelná obsahem registru **OCR0A**.

Pro **neinvertující režim** ( $\text{COM}0\text{X}1 = 1$ ,  $\text{COM}0\text{X}0 = 0$ ) je vývod **OC0X** nastaven do log. 1 vždy po přetečení obsahu čítače. Po dosažení shody  $\text{TCNT}0 = \text{OCR}0\text{X}$  dojde k vynulování tohoto vývodu. Takže **hodnota obsažená v registru OCR0X vlastně přímoúměrně určuje dobu trvání log. 1 na vývodu OC0X**.

V invertujícím režimu ( $\text{COM}0\text{X}1 = 1$ ,  $\text{COM}0\text{X}0 = 1$ ) je vše naopak.

Pro výstupní kmitočet PWM signálu na vývodu **OC0X** platí:

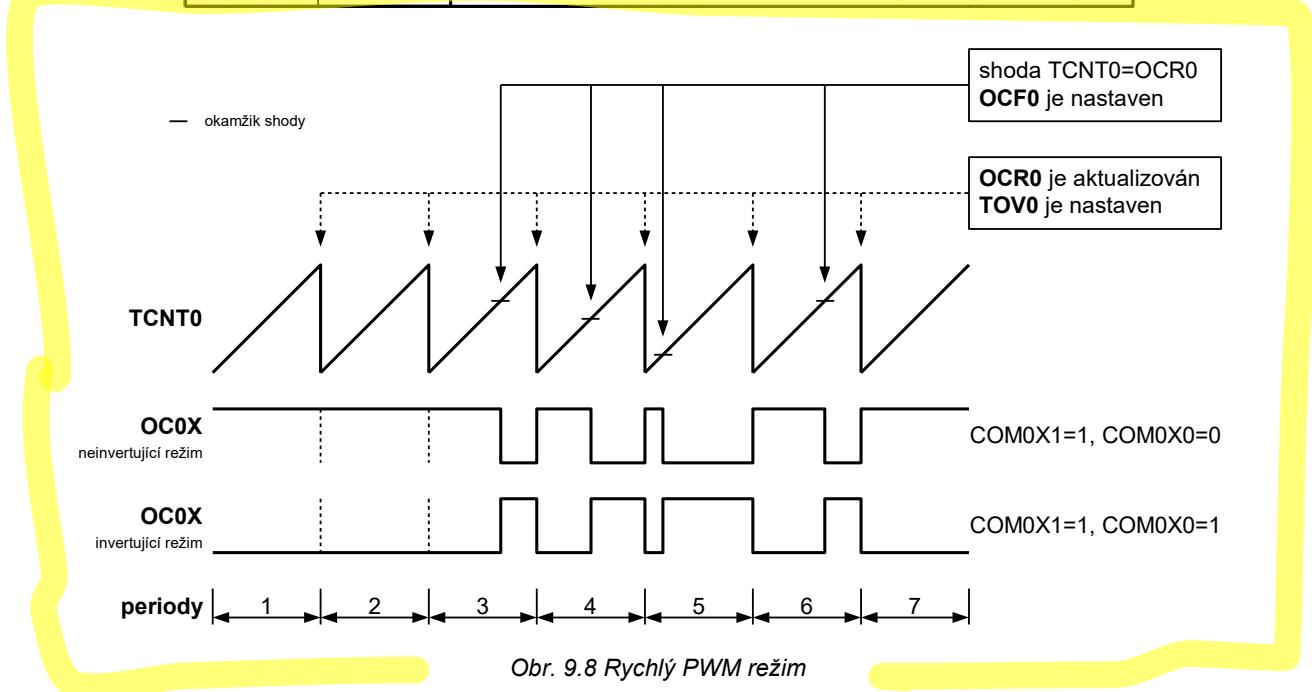
$$f_{\text{OC0X PWM}} = \frac{f_{\text{clk IO}}}{N \cdot 256} \quad (9-2)$$

kde **N** představuje dělicí poměr (1, 8, 64, 256 nebo 1024).

Mezní hodnoty OCR0X registru představují zvláštní případy generování PWM průběhu. Pokud je **OCR0X = 0x00**, vytváří se krátký zákmit v okamžiku přetečení čítače. Pokud je **OCR0X = 0xFF**, bude výstup stabilně v log. 1 (neinvertující režim) nebo log. 0 (invertující režim).

Tab. 9.4 Význam bitů **COM0X1**, **COM0X0** v rychlém PWM režimu

<b>COM0X1</b>	<b>COM0X0</b>	Popis
0	0	čítač/časovač 1 odpojen od vývodu OC0X
0	1	vyhrazeno
1	0	OC0X = 0 po shodě, OC0X = 1 při 0xFF (neinvertující režim)
1	1	OC0X = 1 po shodě, OC0X = 0 při 0xFF (invertující režim)



### Fázově korigovaný PWM režim (WGM = 1 nebo 5)

Fázově korigovaný PWM režim poskytuje PWM průběh s velkým rozlišením, používá dvojfázovou realizaci. Čítač čítá opakovaně z 0x00 do vrcholu a potom z vrcholu do 0x00.

Hodnota vrcholu je pro WGM = 1 pevná (0xFF) a pro WGM = 5 nastavitelná obsahem registru **OCR0A**.

V neinvertujícím režimu je vývod **OC0X** vynulován při shodě TCNT0 = OCR0X při **čítání nahoru**. K nastavení dojde při shodě TCNT0 = OCR0X při **čítání dolů**. Pro invertující režim je činnost opačná.

Dvojfázové provedení má sice nižší pracovní kmitočet než provedení jednofázové (rychlý PWM režim), ale pro některé aplikace se mu dává přednost. Dvojfázová činnost spočívá v tom, že čítač se nejprve inkrementuje až do okamžiku dosažení maxima. Pak se přepne a čítá dolů.

Příznak **TOV0** je nastaven pokaždé, když čítač dosáhne dna (0x00). Nastavení COM0X1 = 1, COM0X0 = 0 poskytuje na vývodu OC0X **neinvertující PWM signál**, pro COM0X1 = 1, COM0X0 = 1 dostáváme **invertující PWM signál**.

Pro výstupní kmitočet PWM signálu na vývodu **OC0X** platí:

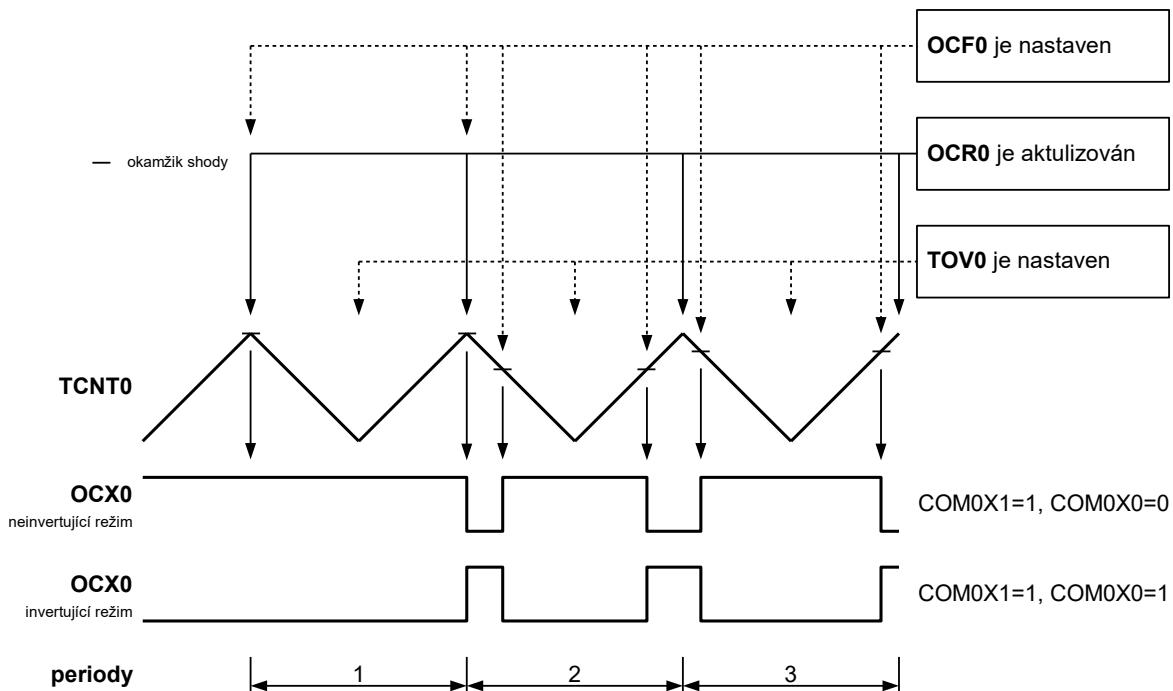
$$f_{\text{OC0X PWM}} = \frac{f_{\text{clk\_IO}}}{N \cdot 510} \quad (9-3)$$

kde **N** představuje dělicí poměr (1, 8, 64, 256 nebo 1024).

Mezní hodnoty **OCR0X** představují zvláštní případy generace PWM signálu. Pro neinvertující režim platí: Je-li **OCR0X = 0x00**, je výstup OC0X stabilně v log. 0. Při **OCR0X = 0xFF** je výstup OC0X stabilně v log. 1. Pro invertující režim platí vše opačně.

Tab. 9.5 Význam bitů **COM0X1**, **COM0X0** ve fázově korigovaném PWM režimu

<b>COM0X1</b>	<b>COM0X0</b>	Popis
0	0	čítač/časovač 1 odpojen od vývodu OC0X
0	1	vyhrazeno
1	0	OC0X = 0 po shodě při čítání nahoru, OC0X = 1 po shodě při čítání dolů (neinvertující režim)
1	1	OC0X = 1 po shodě při čítání nahoru, OC0X = 0 po shodě při čítání dolů (invertující režim)



Obr. 9.9 Fázově korigovaný PWM režim

### Závěrečné poznámky

Pro správnou funkci vývodu **OC0A** je nutné bit nastavit DDRD6 (konfigurovat vývod OC0A jako výstupní), podobně pro vývod **OC0B** je nutné nastavit bit DDRD5.

## 9.5 Vstupy vnějšího přerušení

Nejjednodušší periferii, která může generovat požadavek přerušení, je vstup portu. Vývody **PD2**, **PD3** mají alternativní funkci jako vstupy přerušení **INT0**, **INT1**. Vývody **PB0** až **PB7**, **PC0** až **PC6**, **PD0** až **PD7**, **PE0** až **PE3** mají alternativní funkci jako vstupy přerušení **PCINT0** až **PCINT27**.

Vstupy **INT0** a **INT1** jsou konfigurovatelné s ohledem na spouštěcí podmínu, vstupy **PCINT0** až **PCINT27** reagují na změnu svého stavu (funkce Pin Change).

Vstupy vnějšího přerušení jsou vzorkovány asynchronně (pro aktivaci musí spouštěcí podmínka trvat alespoň 50 ns), mohou tedy mikrokontrolér probudit z režimu snížené spotřeby.

### Vstupy vnějšího přerušení PCINT0 až PCINT27

Bit	7	6	5	4	3	2	1	0
Čtení/zápis	—	—	—	—	PCIE3	PCIE2	PCIE1	PCIE0
Výchozí hodnota	R 0	R 0	R 0	R 0	R/W 0	R/W 0	R/W 0	R/W 0

Obr. 9.10 Registr **PCICR** – povolení přerušení portů PB až PE

- PCIE0 až PCIE3 – povolení přerušení při změně stavu vývodů portu PB až PE.** Je-li příslušný bit nastaven a současně platí  $I = 1$ , vede změna stavu vývodů portu k vyvolání přerušení. Jednotlivé vývody portů lze individuálně povolovat pomocí registrů **PCMSK0** až **PCMSK3**.

Bit	7	6	5	4	3	2	1	0
Čtení/zápis	—	—	—	—	PCIF3	PCIF2	PCIF1	PCIF0
Výchozí hodnota	R 0	R 0	R 0	R 0	R/W 0	R/W 0	R/W 0	R/W 0

Obr. 9.11 Registr **PCIFR** – příznaky přerušení portů PB až PE

- PCIF0 až PCIF3 – příznak přerušení při změně stavu vývodů portu PB až PE.** Příznak pamatuje požadavek přerušení ve chvíli, kdy je přerušení zakázáno. K nulování příznaku dojde aktivací obslužné rutiny přerušení nebo zápisem log. 1 do příslušného bitu.

Bit	7	6	5	4	3	2	1	0
<b>PCMSK3</b>	—	—	—	—	PCINT27	PCINT26	PCINT25	PCINT24
Čtení/zápis	R/W 0							
Bit	7	6	5	4	3	2	1	0
<b>PCMSK2</b>	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16
Čtení/zápis	R/W 0							
Bit	7	6	5	4	3	2	1	0
<b>PCMSK1</b>	—	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8
Čtení/zápis	R/W 0							
Bit	7	6	5	4	3	2	1	0
<b>PCMSK0</b>	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0
Čtení/zápis	R/W 0							

Obr. 9.12 Registry **PCMSK0** až **PCMSK3** – masky vývodů portů PB až PE

- PCINT0 až PCINT27 – masky vývodů portů PB až PE.** Každý z registrů určuje, které vývody portů mohou spouštět přerušení. Jsou-li všechny bity registru vynulovány, daný port negeneruje přerušení.

## Vstupy vnějšího přerušení INT0 a INT1

Bit	7	6	5	4	3	2	1	0
Čtení/zápis	—	—	—	—	—	—	INT1	INT0
Výchozí hodnota	R 0	R 0	R 0	R 0	R 0	R 0	R/W 0	R/W 0

Obr. 9.13 Registr **EIMSK** – povolení přerušení INT0 a INT1

- INT0 a INT1 – povolení přerušení INT0 a INT1.** Je-li příslušný bit nastaven a současně platí  $I = 1$ , vede splnění spouštěcí podmínky k vyvolání přerušení. Spouštěcí podmínka je určena bity registru **EICRA**.

Bit	7	6	5	4	3	2	1	0
Čtení/zápis	—	—	—	—	—	—	INTF1	INTF0
Výchozí hodnota	R 0	R 0	R 0	R 0	R 0	R 0	R/W 0	R/W 0

Obr. 9.14 Registr **EIFR** – příznaky přerušení INT0 a INT1

- INTF0 a INTF1 – příznak přerušení INT0 a INT1.** Příznak pamatuje požadavek přerušení ve chvíli, kdy je přerušení zakázáno. K nulování příznaku dojde aktivací obslužné rutiny přerušení nebo zápisem log. 1 do příslušného bitu.

Bit	7	6	5	4	3	2	1	0
Čtení/zápis	—	—	—	—	ISC11	ISC10	ISC01	ISC00
Výchozí hodnota	R 0	R 0	R 0	R 0	R/W 0	R/W 0	R/W 0	R/W 0

Obr. 9.15 Registr **EICRA** – konfigurace spouštěcí podmínky přerušení INT0 a INT1

- ISC00, ISC01 a ISC10, ISC11 – citlivost přerušení INT0 a INT1.** Dvojice bitů ISCn0, ISCn1 určuje citlivost vstupu INTn pro spuštění přerušení, viz tab. 9.6.

Tab. 9.6 Vliv bitů ISCn0 a ISCn1 na citlivost vstupu INTn

ISCn1	ISCn0	Popis
0	0	vstup je úrovňově citlivý, přerušení vyvolá log. 0
0	1	přerušení je spouštěno libovolnou hranou
1	0	přerušení je spouštěno sestupnou hranou
1	1	přerušení je spouštěno náběžnou hranou

# 10 Realizace aritmetických operací pro celá čísla

Nejdříve se budeme věnovat realizaci aritmetických celých čísel bez znaménka. Následně se zaměříme na BCD čísla a převod do desítkové soustavy.

## 10.1 Aritmetika celých čísel bez znaménka

Procesor je vybaven ALU, která vykonává operace nad čísly bez znaménka přímo. Problém vznikne, pokud je požadována aritmetická operace, která není implementována (například operace dělení). Podobná situace nastane, když požadujeme zpracovat operand větší šíře, než který ALU podporuje přímo.

### Sčítání

Operace sčítání patří mezi samozřejmě operace poskytované ALU. Uvažujme případ, že na 8bitové ALU požadujeme provést součet dvou 32 bitových čísel.

**Realizace takové operace probíhá s využitím příznaku C** (Carry – přenos do vyššího řádu). Pokud součet v dolních bitech přeteče, musíme jej nechat „šířit“ do dalších 8 bitů.

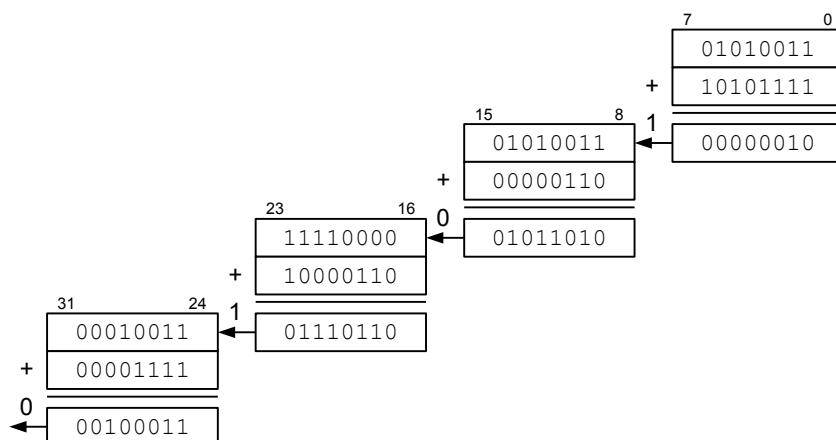
Náznak realizace je uveden formou obr. 10.1.

Uvažujeme operandy součtu:

$$\begin{array}{r} 0001\ 0011\ 1111\ 0000\ 0101\ 0011\ 0101\ 0011_2 = 334517075 \text{ (10)}, \\ 0000\ 1111\ 1000\ 0110\ 0000\ 0110\ 1010\ 1111_2 = 260441775 \text{ (10)}, \end{array}$$

Výsledek:

$$0010\ 0011\ 0111\ 0110\ 0101\ 1010\ 0000\ 0010_2 = 594958850 \text{ (10)}.$$



Obr. 10.1 Náznak realizace 32bitového součtu pomocí 8bitové ALU

Sčítání větší šíře provedeme tedy tak, že sčítáme postupně od nejnižších bajtů směrem k vyšším bajtům a do vyššího součtu vždy zahrnujeme přenos z předchozího součtu. Odpovídající zápis v assembleru AVR za předpokladu, že jsou operandy uloženy ve čtveřicích registrů: R3:R2:R1:R0 a R13:R12:R11:R10, je uveden níže. První součet je proveden instrukcí **ADD** (součet bez přenosu z nižšího řádu), všechny následující součty musí přenos uvažovat a používají tedy instrukci **ADC**:

```
ADD R0,R10  
ADC R1,R11  
ADC R2,R12  
ADC R3,R13
```

Výsledek je v tomto případě uložen ve čtveřici: R3:R2:R1:R0. Samotný součet je 4× časově náročnější než prostý 8bitový součet.

### Odčítání

Odčítání při 32bitové šíři operantu probíhá podobně, jako výše uvedené 32bitové scítání. **Opět využíváme příznaku C**, který má však nyní význam výpůjčky.

Náznak realizace je uveden formou obr. 10.2.

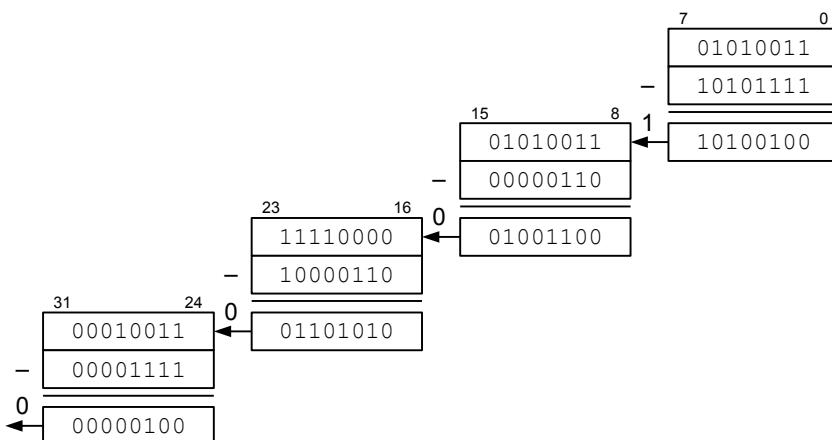
Uvažujeme operandy rozdílu:

$$0001\ 0011\ 1111\ 0000\ 0101\ 0011\ 0101\ 0011_{(2)} = 334517075 \text{ (10)},$$

$$0000\ 1111\ 1000\ 0110\ 0000\ 0110\ 1010\ 1111_{(2)} = 260441775 \text{ (10)},$$

Výsledek:

$$0000\ 0100\ 0110\ 1010\ 0100\ 1100\ 1010\ 0100_{(2)} = 74075300 \text{ (10)}.$$



Obr. 10.2 Náznak realizace 32bitového rozdílu pomocí 8bitové ALU

Odpovídající zápis v assembleru AVR za předpokladu, že jsou operandy uloženy ve čtveřicích registrů: R3:R2:R1:R0 a R13:R12:R11:R10, je uveden níže. První rozdíl je proveden instrukcí **SUB** (rozdíl bez výpůjčky z nižšího řádu), všechny následující rozdíly musí výpůjčku uvažovat a používají tedy instrukci **SBC**:

```
SUB R0,R10
SBC R1,R11
SBC R2,R12
SBC R3,R13
```

Výsledek je v tomto případě uložen ve čtveřici: R3:R2:R1:R0. Samotný rozdíl je 4× časově náročnější než prostý 8bitový rozdíl.

### Násobení

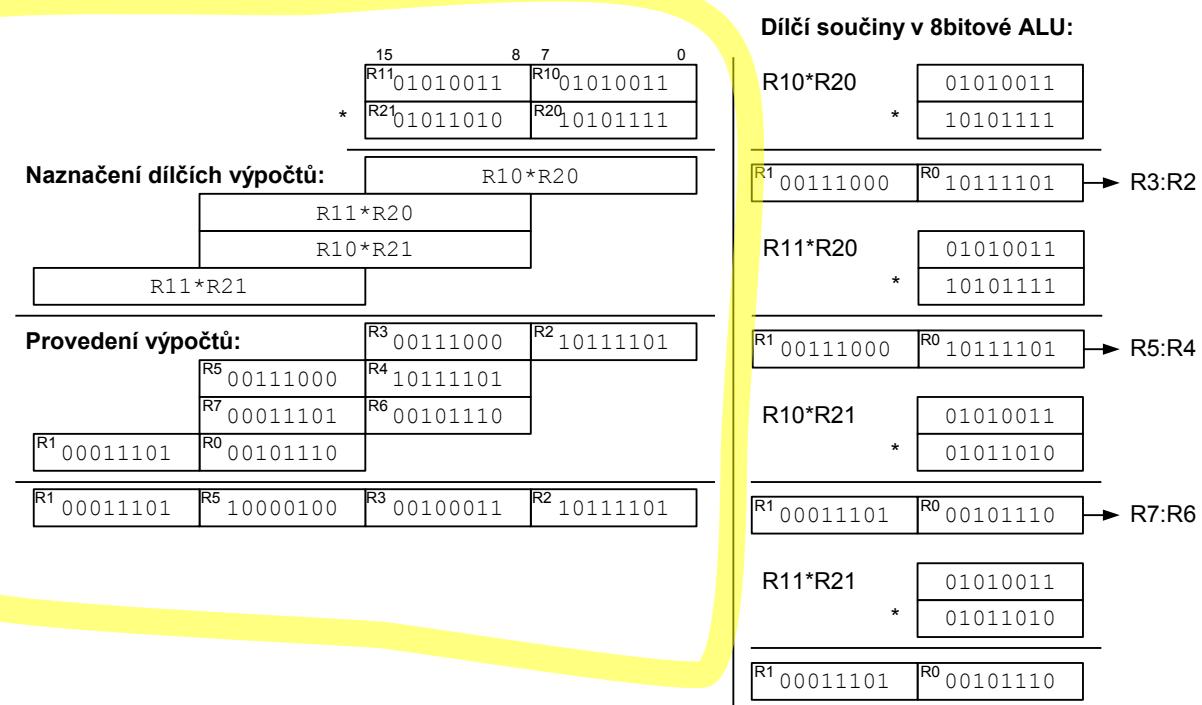
Uvažujme nyní potřebu realizovat 16bitové násobení pomocí 8bitového procesoru.

Uvažujme nejprve případ, že **procesor je vybaven instrukcí pro násobení 8bitových čísel**. Oba činitele součinu lze rozdělit po 8bitech a provést násobení ve více krocích.

V podstatě lze na obsah bajtu nahlížet jako na jednu číslici zapsanou v soustavě se základem 256. Uvažujeme, že operandy součinu jsou uloženy v párech registrů R11:R10 a R21:R20. Je tedy třeba provést tyto dílčí 8bitové součiny: R10\*R20, R11\*R20, R10\*R21 a R11\*R21.

Jednotlivé dílčí součiny nakonec sečteme, při vlastním součtu je třeba uvážit vzájemnou pozici jednotlivých bajtů.

Konkrétní realizace je zřejmá z obr. 10.3. Vlastní 8bitový součin celých čísel bez znaménka provedeme pomocí instrukce **MUL**. Tato instrukce ukládá výsledný součin do páru registrů R1:R0. Proto musí být použity další páry registrů pro zachycení průběžných součinů, konkrétně: R3:R2, R5:R4, R7:R6. Poslední součin ponecháme v R1:R0.



Obr. 10.3 Náznak realizace 16bitového součinu pomocí 8bitové ALU

Takto naznačený postup lze pochopitelně přepsat do assembleru AVR:

```

MUL R10, R20      ; součin R10*R20
MOV R3, R1
MOV R2, R0          ; výsledek do R3:R2


---


MUL R11, R20      ; součin R11*R20
MOV R5, R1
MOV R4, R0          ; výsledek do R5:R4


---


MUL R10, R21      ; součin R10*R21
MOV R7, R1
MOV R6, R0          ; součin do R7:R6


---


MUL R11, R21      ; součin R11*R21, ponecháme v R1:R0
ADD R3, R4          ; součet R3 = R3+R4+R6
ADC R3, R6


---


ADC R5, R7          ; součet R5 = R5+R7+R0+C
ADC R5, R0


---


CLR R7              ; nulování R7 pro snadné přičtení C
ADC R1, R7          ; součet R1 = 0+C

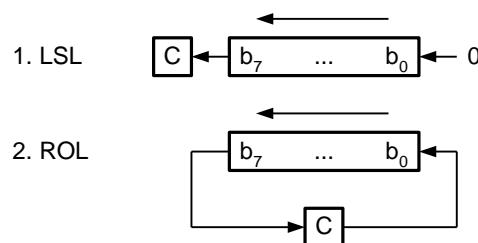
```

Zápis v podstatě odpovídá situaci dle obr. 10.3. Pro započítání přenosu (C) z předposledního bajtu do posledního bajtu byl použit jednoduchý trik. K obsahu R1 byl připočítán dříve vynulovaný registr R7 včetně C.

Výsledek je ve shodě s obr. 10.2 uložen ve čtveřici R1:R5:R3:R2. Pochopitelně je možné zvolit jiné registry pro uložení dílčích součinů nebo konečný výsledek přesunout do jiné čtveřice registrů.

V případě, že procesor není instrukcí pro násobení vybaven, provádíme násobení na úrovni dvojkových čísel s tím, že lze využít vlastností posuvu resp. rotace doleva. **Posuv o jedno místo doleva odpovídá násobení 2.**

Násobení dvou 8bitových čísel vede na 16bitový výsledek. Proto je zřejmé, že musíme realizovat vícebitový posuv. Řešení je naznačeno formou obr. 10.4. Nejdříve (1) provedeme posuv nižšího bajtu doleva pomocí instrukce **LSL**, kdy nejvyšší bit vystoupí do příznaku C. Hodnotu tohoto bitu pak ve druhém kroku (2) rotujeme doleva spolu s ostatními bity vyššího bajtu instrukcí **ROL**.



Obr. 10.4 Náznak realizace násobení pomocí posuvu a rotace doleva

Posuv o 1 místo doleva znamená násobení 2, o dvě místa odpovídá násobení 4, o tři místa násobení 8, atd.

Výpočet obecného součinu (pokud se nejedná přímo o mocninu čísla 2) lze provést postupným posouváním a následným sčítáním. Například při násobení číslem 10 je zřejmé, že platí  $10 = 8+2$ . Výsledek tedy získáme tak, že první činitel posuneme nejdříve o jedno místo doleva. K tomuto mezivýsledku přičteme hodnotu prvního činitele posunutou o 3 místa doleva ( $8 = 2^3$ ). Jak ukazuje obr. 10.5.

$10101110 \quad * \quad 00001010 = 174_{(10)} * 10_{(10)}$		
$  \begin{array}{r}  0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\  + 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1 \\  \hline  0\ 0\ 0\ 0\ 0\ 1\ 1\ 0  \end{array}  $	posuv 1x	
$  \begin{array}{r}  0\ 0\ 0\ 0\ 0\ 0\ 1\ 1 \\  + 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\  \hline  0\ 0\ 0\ 0\ 0\ 0\ 0\ 0  \end{array}  $	posuv 3x	
$  \begin{array}{r}  0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 0  \end{array}  $	výsledek	
$0000011011001100 = 1740_{(10)}$		

Obr. 10.5 Náznak realizace násobení 10

Obecně lze tedy dojít k následujícímu **algoritmu pro 8bitové násobení** dvou registrů R2 a R3 s výsledkem uloženým v páru R1:R0 a s použitím pomocných registrů R4 (pro realizaci posuvu) a R5 (počítadlo bitů):

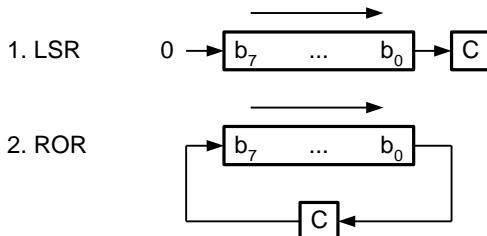
1. Výchozí nastavení registrů: R1 = 0, R0 = 0, R4 = 0, R5 = 8,
2. Posuň R2 doprava (do C vystoupí nejnižší bit).
3. Je-li C rovno 0, jdi na bod 5.
4. Přičti obsah páru R4:R3 k výsledku R1:R0.
5. Posuň páry R4:R3 o jedno místo doleva.
6. Sniž R5 o 1.
7. Je-li R5 rovno 0, jdi na bod 9.
8. Jdi na bod 2.
9. Konec výpočtu.

## Dělení

Na rozdíl od násobení, nemá v případě dělení smysl uvažovat variantu, že by byl procesor vybaven instrukcí dělení v menší šířce operandů. Případnou instrukci dělení nelze nyní použít.

Pro realizaci dělení lze opět použít posuv, v tomto případě doprava. Protože **posuv o jeden bit doprava odpovídá dělení 2**.

Budeme-li uvažovat 16bitové dělení, musíme tedy realizovat vícebitový posuv dle obr. 10.6. Nejdříve (1) provedeme posuv vyššího bajtu doprava pomocí instrukce **LSR**, kdy nejnižší bit vystoupí do příznaku C. Hodnotu tohoto bitu pak ve druhém kroku (2) rotujeme doprava spolu s ostatními bity nižšího bajtu instrukcí **ROR**.



Obr. 10.6 Náznak realizace dělení pomocí posuvu a rotace doprava

Dělení probíhá ve dvojkové soustavě podobně, jako v desítkové soustavě. Z dělence vybíráme po jednom bitu do vícebitového čísla:

- pokud je dělenec menší než dělitel, zapíšeme do výsledku 0,
- pokud je dělenec vyšší nebo roven děliteli, stanovíme zbytek (bity pak budeme připojovat k němu), zapíšeme do výsledku 1.

Obecně lze tedy dojít k následujícímu **algoritmu pro 8bitové dělení** dvou registrů R3 (dělenec), R2 (dělitel) s výsledkem uloženým v registru R0 a zbytkem v registru R1, s použitím pomocného registru R5 (počítadlo bitů):

1. Výchozí nastavení registrů: R1 = 0, R0 = 0, R5 = 8,
2. Posuň R3 doleva (nejvyšší bit vystoupí do C).
3. Rotuj R1 doleva (přes C do něj vystoupí bit dříve vysunutý z R3).
4. Je-li R1 < R2, jdi na bod 6.
5. Stanov: R1 = R1 - R2, jdi na bod 8.
6. C = 0.
7. Jdi na bod 9.
8. C = 1.
9. Rotuj R0 doleva (vsuň C zprava).
10. Sniž R5 o 1.
11. Je-li R5 rovno 0, jdi na bod 13.
12. Jdi na bod 2.
13. Konec výpočtu.

## 10.2 BCD kód

BCD je zkratka označení **Binary Coded Decimal**, jedná se tedy o možnost uložit desítková čísla v binárním kódu.

### Princip

Obvyklý způsob BCD kódování používá pro uložení jedné desítkové číslice jeden nibble (tedy čtveřici bitů) s váhami 8, 4, 2, 1. Takový způsob odpovídá

normálnímu vyjádření číslic 0 až 9 pomocí binárních kódů 0000 až 1001. Kombinace 1010 až 1111 jsou pak zakázané.

Desítkové číslo uložené v BCD kódu zabírá větší počet míst, než uložení v přímém binárním kódu. Například do jednoho bajtu lze uložit pouze dvě číslice BCD a tedy má rozsah pro BCD kódování pouze 0 až 99.

## Sčítání

Procesor nepodporuje sčítání BCD čísel přímo, sčítáčka pracuje v binárním kódu. Provedeme-li například součet  $0x58 + 0x02$ , bude výsledek  $0x5A$ . ALU provedla normální sčítání. Potřebujeme výsledek  $0x60$ . Ten dostaneme použitím tzv. desítkové korekce těsně po instrukci součtu. **Desítková korekce** pracuje takto:

- V případě, že je dolní nibble hodnoty vyšší než 1001 nebo pokud vznikl přenos mezi dolním a horním nibble ( $H = 1$ ), přičte se k výsledku hodnota  $0x06$ .
- Podobně, pokud je horní nibble hodnoty vyšší než 1001 nebo vznikl přenos z nejvyššího bitu ( $C = 1$ ), přičte se k výsledku číslo  $0x60$ .

Některé procesory mají k dispozici instrukci **DA** (Decimal Addition), která zajistí provedení desítkové korekce po operaci součtu. Instrukční soubor procesorů AVR tuto instrukci neobsahuje.

## Odčítání

Odečítání realizujeme převedením na sčítání čísla v desítkovém doplňku.

Například pro výpočet  $164 - 44$  je třeba čísla reprezentovat řádovou mřížkou šířky tří desítkových řádů:  $L = 3$ . Modul této řádové mřížky je tedy  $Z = 10^3 = 1000$ . Desítkový doplněk pro  $-44$  tedy bude:  $D(A) = Z + A = 1000 - 44 = 956$ .

$$\begin{array}{r} 164 \\ - 44 \\ \hline 120 \end{array}$$

Nyní již provedeme součet (zapsáno v šestnáctkové soustavě): + 956 .

ABA

ABA

Po provedení desítkové korekce dostaneme: + 666 , v každém nibble jsme přičítali 6,  $120$

protože všude byly uloženy kódy vyšší než 1001, výsledný přenos jsme ignorovali.

## Převod z binárního kódu do BCD

Převod z binárního kódu do BCD nám umožní snadný rozklad čísla na desítkové číslice například v případě, že je třeba zobrazit hodnotu čísla v desítkové soustavě. Metody provedení:

- postupné dělení binárního čísla základem,
- postupné odečítání základu,
- převod přes Hornerovo schéma.

### 1. Postupné dělení binárního čísla základem

Tato metoda je založena na postupném dělení mocninami čísla 10.

Příklad:

$$\begin{aligned} 5274 / 1000 &= 5, \text{ zbytek } 274 \\ 274 / 100 &= 2, \text{ zbytek } 74 \\ 74 / 10 &= 7, \text{ zbytek } 4 \end{aligned}$$

Výhoda:

- u jednobajtových čísel je jasný a jednoduchý algoritmus.

Nevýhody:

- nutnost dělení,
- pro čísla delší než šířka ALU je komplikovaná realizace (náročné vícebajtové dělení).

## 2. Postupné odečítání základu

Tato metoda je založena na postupném odčítání mocnin čísla 10. Po každém kroku se zkoumá, je-li výsledek záporný:

- pokud ne, zvýšíme hodnotu počítadla na dané pozici,
- pokud ano, daný řád byl eliminován; počítadlo obsahuje počet základů pro daný řád (tedy hledanou číslici v daném řádu).

Příklad:

tisíce = 0

$$1234 - 1000 = 234 < 0 \text{ -- ne, tisíce} = 1$$

**234 - 1000 = -766 < 0 -- ano, hledané tisíce = 1,**

234 je základ pro eliminaci v řádu stovek

stovky = 0

$$234 - 100 = 134 < 0 \text{ -- ne, stovky} = 1$$

**134 - 100 = 34 < 0 -- ne, stovky = 2**

**34 - 100 = -66 < 0 -- ano, hledané stovky = 2,**

34 je základ pro eliminaci v řádu desítek

atd.

Výhoda:

- poměrně jednoduchá realizace i pro delší čísla (vícebajtové odčítání je jednoduché).

Nevýhoda:

- nekonstantní doba převodu (závisí na hodnotě převáděného čísla).

## 3. Převod přes Hornerovo schéma

Každé celé binární číslo **A** šířky **n** bitů lze vyjádřit pomocí vztahu:

$$A_{(2)} = \sum_{i=0}^{n-1} a_i \cdot 2^i = a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \dots + a_12 + a_0$$

Tento vztah lze pomocí postupného vytýkání základu ve vyšších řádech snadno upravit na tzv. **Hornerovo schéma**:

$$A_{(2)} = (\dots(a_{n-1} \cdot 2 + a_{n-2}) \cdot 2 + a_{n-3}) \cdot 2 \dots + a_1) \cdot 2 + a_0$$

Příklad pro 4bitové číslo:

$$A_{(2)} = a_32^3 + a_22^2 + a_12 + a_0 = (a_32^2 + a_22 + a_1) \cdot 2 + a_0 = ((a_32 + a_2) \cdot 2 + a_1) \cdot 2 + a_0$$

Postup:

Hledané dekadické číslo A lze vytvářet od nejvyššího bitu postupným násobením mezičísel dvěma a přičítáním dalšího následujícího nižšího bitu. Budeme-li toto číslo od začátku chápout jako dekadické (v BCD kódu), bude

dekadickým i po každé operaci (násobení 2 a součet), a pokud ne, lze na něj aplikovat dekadickou korekci (DA), po které opět získáme dekadické číslo.

Příklad:  $52_{(10)} = 0011\ 0100_{(2)}$

legenda

BCD	BIN	operace
<b>1. krok</b>		
00	0000 0000	$0 * 2$
00	0000 0000	$0 * 2 + a_7 = a_7$
00	0000 0000	DA
<b>2. krok</b>		
00	0000 0000	$a_7 * 2$
00	0000 0000	$a_7 * 2 + a_6$
00	0000 0000	DA
<b>3. krok</b>		
00	0000 0000	$(a_7 * 2 + a_6) * 2$
01	0000 0001	$(a_7 * 2 + a_6) * 2 + a_5$
01	0000 0001	DA
<b>4. krok</b>		
02	0000 0010	$((a_7 * 2 + a_6) * 2 + a_5) * 2$
03	0000 0011	$((a_7 * 2 + a_6) * 2 + a_5) * 2 + a_4$
03	0000 0011	DA
<b>5. krok</b>		
06	0000 0110	$((((a_7 * 2 + a_6) * 2 + a_5) * 2 + a_4) * 2 + a_3) * 2$
06	0000 0110	$((((a_7 * 2 + a_6) * 2 + a_5) * 2 + a_4) * 2 + a_3) * 2 + a_3$
06	0000 0110	DA
<b>6. krok</b>		
0 (12)	0000 1100	$(((((a_7 * 2 + a_6) * 2 + a_5) * 2 + a_4) * 2 + a_3) * 2 + a_3) * 2$
0 (13)	0000 1101	$(((((a_7 * 2 + a_6) * 2 + a_5) * 2 + a_4) * 2 + a_3) * 2 + a_3) * 2 + a_2$
13	0001 0011	DA
<b>7. krok</b>		
26	0010 0110	$((((((a_7 * 2 + a_6) * 2 + a_5) * 2 + a_4) * 2 + a_3) * 2 + a_2) * 2$
26	0010 0110	$((((((a_7 * 2 + a_6) * 2 + a_5) * 2 + a_4) * 2 + a_3) * 2 + a_2) * 2 + a_1$
26	0010 0110	DA
<b>8. krok</b>		
4 (12)	0100 1100	$((((((a_7 * 2 + a_6) * 2 + a_5) * 2 + a_4) * 2 + a_3) * 2 + a_2) * 2 + a_1) * 2$
4 (12)	0100 1100	$((((((a_7 * 2 + a_6) * 2 + a_5) * 2 + a_4) * 2 + a_3) * 2 + a_2) * 2 + a_1) * 2 + a_0$
52	0101 0010	DA

Výhoda:

- lze realizovat ve více smyčkách pro číslo libovolné délky.

Nevýhoda:

- žádná není.

### Převody BCD do binárního kódu

Nejvýhodnější variantou realizace tohoto převodu je opět rozklad do formy Hornerova schématu. Každý nibble představuje jednu desítkovou číslici.

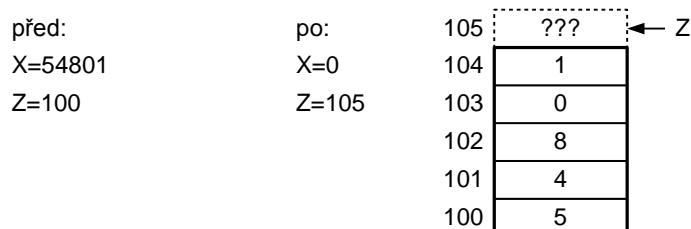
Například čtyřmístné číslo představované nibbly: d c b a převedeme do dvojkové soustavy:  $A_{(2)} = 10 \cdot (10 \cdot (10 \cdot d + c) + b) + a$ .

Připomeňme, že násobení  $\times 10$  lze realizovat jako násobení  $\times 2$  a  $\times 8$  (pomocí posuvů doleva o 1 a o 3 místa). Tedy velmi jednoduše.

## Praktická ukázka realizace rozkladu čísla na desítkové číslice

Níže je uvedena jedna z možností realizace rozkladu čísla na desítkové číslice. Podprogram je označen jako **BNDC** (BiNary to DeCimal).

Převáděné číslo v rozsahu 16 bitů vložíme do registru X. Výsledek převodu bude uložen do datové paměti od adresy určené výchozí hodnotou ukazatele Z. Po provedení převodu bude ukazatel Z obsahovat adresu o 1 vyšší než je adresa poslední převedené číslice. Vedlejším efektem převodu je vynulování registru X. Na obr. 10.7 je ukázán příklad výsledku převodu.



Obr. 10.7 Příklad výsledku převodu pomocí podprogramu BNDC

Podprogram používá registry **NUM** (převedená číslice), **CNTC** (počítadlo číslic) a **CNTB** (počítadlo bitů).

Pro převod každé číslice se nejdříve vynuluje registr NUM a počítadlo CNTB se nastaví na 16. Při převodu se trojice registrů NUM:XH:XL násobí 2. Pokud nejvyšší bajt (odpovídá NUM) dosáhne hodnoty 10, je NUM snížen o 10 a do XL je vrácena 1. Po 16 průchodech je počítadlo CNTB na nule a NUM obsahuje platnou desítkovou číslici, která je uložena do zásobníku.

Po převodu číslice se počítadlo CNTC zvýší o 1. Pokud není registr X nulový, pokračuje se převodem další číslice.

Dekódované číslice se ukládají do zásobníku proto, že převod probíhá od jednotek směrem k vyšším řádům. Závěrečná fáze podprogramu postupně „rozebírá“ zásobník a ukládá jednotlivé číslice do datové paměti počínaje výchozí adresou v ukazateli Z. Dojde tak ke kýzenému „převrácení“ pořadí číslic. Na nejnižší adrese je uložena číslice z nejvyššího řádu výsledku.

```
;BNDC-rozloží číslo na desítkové číslice
;vstup:   X-převáděné číslo
;          Z-adresa prvního bajtu pro uložení číslic
;výstup:  X=0
;          Z-adresa za poslední číslicí
;mění:    NUM, CNTC, CNTB, SREG

        .DEF NUM=R16    ;převedená číslice
        .DEF CNTC=R17   ;počítadlo číslic
        .DEF CNTB=R18   ;počítadlo bitů
```

<b>BNDC:</b>	CLR CNTC      ;počítadlo číslic=0
BNDC1:	CLR NUM       ;nulování číslice
	LDI CNTB,16   ;počet bitů 16
BNDC2:	LSL XL
	ROL XH
	ROL NUM       ;NUM:X=2*(NUM:X)
	CPI NUM,10    ;NUM>=10?

```
        BRCS BNDC3
        SUBI NUM,10      ;vrácení desítky
        ADIW XL,1        ;z vyššího řádu zpět
BNDC3:   DEC CNTB      ;snížení počítadla bitů po násobení
        BRNE BNDC2      ;již 16 bitů?
        PUSH NUM        ;uložení číslice do zásobníku
        INC CNTC        ;zvýšení počítadla převedených číslic
        MOV NUM,XL
        OR NUM,XH        ;test nulového X
        BRNE BNDC1      ;pro nenulové X pokračujeme
;uložení výsledků do RAM:
```

```
BNDC4:   POP NUM        ;vyjmout číslice ze zásobníku
        ST Z+,NUM       ;uložení číslice do SRAM
        DEC CNTC        ;snížení počítadla číslic
        BRNE BNDC4      ;test konce
        RET
```

# 11 Pokročilejší aritmetické operace

V této kapitole se seznámíme se standardním způsobem reprezentace čísel v plovoucí řádové čárce. Dále doplníme informace k realizaci matematických funkcí.

## 11.1 Čísla v plovoucí řádové čárce

Níže je vysvětlen způsob reprezentace čísel v plovoucí řádové čárce nejdříve obecně a poté na základě standardu IEEE 754.

### Úvod

Základní problém použití číselných formátů s pevnou řádovou čárkou (fixed point) spočívá v tom, že pro velký rozsah čísla je nutný velký počet bitů. Což se odrazí na náročné konstrukci procesoru.

Například pro uložení hodnoty Avogadrovy konstanty  $6,023 \cdot 10^{23}$  je třeba minimálně 79bitové číslo ( $2^{79} = 6,0446291 \cdot 10^{23}$ ).

Ovšem pro velká čísla není obvykle třeba tak vysoká přesnost, která odpovídá jednotce řádové mřížky ( $\varepsilon$ ). Takže řešením je použít plovoucí řádovou čárku, tedy zajistit změnu měřítka během výpočtu.

Zobrazení čísla A je pak představováno uspořádanou dvojicí (M; E):

$$A = M \cdot z^E \quad (11-1)$$

kde:

- A – číslo vyjádřené v soustavě se základem **z**,
- M – **mantisa**,
- E – **exponent**.

Počet řádů mantisy určuje přesnost, počet řádů exponentu určuje zobrazitelný rozsah.

### Normalizovaný tvar

Určitým problémem reprezentace čísla dle vztahu (11-1) je několik možných variant, což by vedlo na komplikovanou realizaci aritmetických operací. Například číslo 3584,1 lze vyjádřit například takto:

$$3584,1 \cdot 10^0 = 3,5841 \cdot 10^3 = 0,35841 \cdot 10^4$$

Řešením je tzv. **normalizovaný tvar** zápisu čísla (normalized form).

Pro případ základu 10 (z = 10) provedeme převod na normalizovaný tvar takto: Číslo posouváme doprava/doleva (čímž provedeme změnu polohy řádové čárky) a současně upravujeme exponent tak, aby byla poloha řádové čárky vždy vlevo do prvního nenulového řádu. Například pro výše uvedené číslo je normalizovaný tvar:  $0,35841 \cdot 10^4$ .

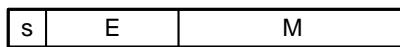
Pro případ základu 2 (z = 2) je třeba, aby byla řádová čárka umístěna vpravo od první 1. Využívá se přitom tzv. **skrytá jednička**. Jelikož vlevo od řádové čárky bude vždy číslice 1, není třeba ji ani ukládat. Příklad normalizace čísla 0,001011 je uveden níže:

nenormalizovaná mantisa	normalizovaná mantisa	uložená mantisa
0.001011	1.011	011

Určitým problémem je skutečnost, že nelze přesně reprezentovat hodnotu 0.

## Konstrukce řádové mřížky

Typická konstrukce řádové mřížky je uvedena formou obr. 11.1. Důvod zvoleného uspořádání spočívá v možnosti rychlého porovnání čísel.

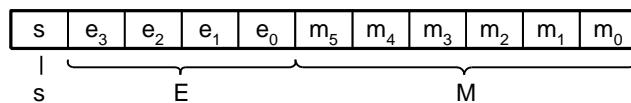


Obr. 11.1 Řádová mřížka pro čísla v plovoucí řádové čárce

kde:

- s – znaménko mantisy,
- M – mantisa,
- E – exponent.

Jako konkrétní případ můžeme uvést obr. 11.2, kde je exponent vyjádřen pomocí 4 bitů a mantisa pomocí 6 bitů.

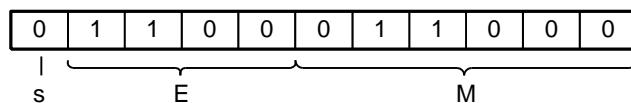


Obr. 11.2 Detailní rozkreslení řádové mřížky

**Exponent reprezentujeme kódem s posunutou nulou** dle vztahu (2-3), kde posunutí volíme  $K = 1/2 \cdot Z = 8$ . **Mantisu reprezentujeme v přímém kódu** se skrytou jedničkou.

Příklad: reprezentujme číslo  $22_{(10)}$

- nejdříve číslo převedeme do dvojkové soustavy:  $10110_{(2)}$ ,
- provedeme normalizaci:  $10110 \times 2^0 = 1.0110 \times 2^4$ ,
- pomocí posunutí  $K = 8$  určíme obraz exponentu:  $4 + 8 = 12_{(10)} = 1100_{(2)}$ ,
- výsledek je uveden formou obr. 11.3.



Obr. 11.3 Reprezentace čísla  $22_{(10)}$

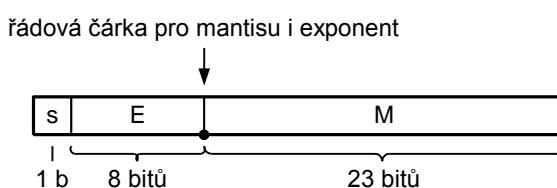
## Standard IEEE 754

Tento standard byl vyvinut roku 1985, stal se obecným standardem pro čísla v plovoucí řádové čárce. Jsou definovány dva základní formáty:

- **single precision** (jednoduchá přesnost – odpovídá typu **float** v jazyce C),
- **double precision** (dvojnásobná přesnost – odpovídá typu **double** v jazyce C).

### Single precision (jednoduchá přesnost) – 32 bitů

Rádová mřížka pro čísla s jednoduchou přesností je uvedena na obr. 11.4.



Obr. 11.4 Řádová mřížka pro čísla s jednoduchou přesností

Pro **exponent** se používá kód s posunutou nulou s tím, že posunutí není zvoleno do poloviny rozsahu, ale  $K = 127$ . Tím je umožněno vyhradit kombinace 0x00 a 0xFF pro zvláštní účely.

Pro **mantisu** se používá přímý kód v normalizovaném tvaru se skrytou jedničkou. Mantisa využívá celkem 24 bitů.

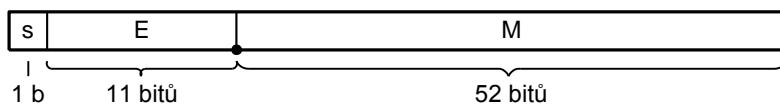
Hodnotu čísla lze určit pomocí vztahu:

$$A = (-1)^S \cdot 1.M \cdot 2^{E-127} \quad (11-2)$$

Tento formát odpovídá přesnosti 6 až 7 desítkových číslic. Rozsah je v absolutní hodnotě:  $10^{-45}$  až  $10^{38}$ .

#### Double precision (dvojnásobná přesnost) – 64 bitů

Řádová mřížka pro čísla s dvojnásobnou přesností je uvedena formou obr. 11.5.



Obr. 11.5 Řádová mřížka pro čísla s dvojnásobnou přesností

**Exponent** je opět uložen v kódu s posunutou nulou,  $K = 1023$ . Kombinace 0x000 a 0xFFFF jsou opět vyhrazeny pro zvláštní účely.

Hodnotu čísla lze určit:

$$A = (-1)^S \cdot 1.M \cdot 2^{E-1023} \quad (11-3)$$

#### Vyhrazené kombinace (viz tab. 11.1)

Uvedený formát dovoluje uložit 5 různých typů čísel. Pro případ jednoduché přesnosti jsou to typy:

- normální – normalizovaná mantisa (odpovídá výše uvedenému popisu),
- čistá 0 –  $E = 0x00$ ,  $M = \text{samé } 0$ ,  $s = 0/1$  (kladná i záporná nula),
- špinavá 0 – pro zmírnění chyb v okolí 0, uloženo bez normalizace (příklad níže),
- $\infty$  –  $E = 0xFF$ ,  $M = \text{samé } 0$ ,  $s = 0/1$  (kladné i záporné nekonečno),
- NaN (Not A Number) –  $E = 0xFF$ ,  $M = \text{nenulové číslo}$ , používá se pro uložení výsledku pro případy operací:  $0/0$ ,  $\infty/\infty$ ,  $SQRT(-A)$ , ...

#### Násobení

Násobení probíhá jako součet exponentů a součin mantis, tedy:

$$M_1 \cdot z^{E_1} * M_2 \cdot z^{E_2} = (M_1 * M_2) \cdot z^{E_1+E_2}$$

Příklad:

$$\begin{aligned} 1.101 \times 2^2 * (-1.110 \times 2^{-3}) &= \\ = -(1.101 * 1.110) \times 2^{2-3} &= \\ = -10.110110 \times 2^{-1} &= \\ = -1.011 \times 2^0 & \end{aligned}$$

#### Dělení

Dělení probíhá jako rozdíl exponentů a podíl mantis, tedy:

$$M_1 \cdot z^{E_1} / M_2 \cdot z^{E_2} = (M_1 / M_2) \cdot z^{E_1-E_2}$$

Tab. 11.1 Příklady zápisů čísel v jednoduché přesnosti

pozn.	$A_{(10)}$	$S_{(2)}$	$E_{(10)}$	$M_{(10)}$	$M_{(2)}$
a	1.0	0	127	0	000 0000 0000 0000 0000 0000
a	-1.0	1	127	0	000 0000 0000 0000 0000 0000
b	0.1	0	123	5 033 165	100 1100 1100 1100 1100 1101
c	$3.13 \times 10^{38}$	0	254	8 388 607	111 1111 1111 1111 1111 1111
	+0	0	0	0	000 0000 0000 0000 0000 0000
	-0	1	0	0	000 0000 0000 0000 0000 0000
	$+\infty$	0	255	0	000 0000 0000 0000 0000 0000
	$+\text{NaN}$	0	255	$M \neq 0$	$M \neq 0$
d	$2^{-128}$	0	0		010 0000 0000 0000 0000 0000

poznámky:

- a) – normální formát,
- b) – nejmenší záporný exponent,
- c) – největší kladný exponent,
- d) – nenormalizované číslo  $0.010 \times 2^{-126} = 2^{-128}$ .

### Sčítání a odčítání

Základní podmínkou výpočtu je, aby obě čísla měla stejný exponent. V případě splnění této podmínky provedeme součet/rozdíl mantis a exponent se nezmění.

V případě, že se exponenty liší, provedeme pro čísla v normalizovaném stavu úpravu na větší z exponentů:

- a) pokud  $E_1 \geq E_2$ , provedeme úpravu na tvar:

$$M_1 \cdot z^{E_1} \pm M_2 \cdot z^{E_2} = (M_1 \pm M_2 \cdot z^{E_2-E_1}) \cdot z^{E_1},$$

- b) pokud  $E_1 < E_2$ , provedeme úpravu na tvar:

$$M_1 \cdot z^{E_1} \pm M_2 \cdot z^{E_2} = (M_1 \cdot z^{E_1-E_2} \pm M_2) \cdot z^{E_2}.$$

Násobení mocninou základu provedeme posuvem. Při této operaci může vzniknout **ztráta přesnosti**. Viz níže uvedený příklad.

Příklad (uvažujme mantisu s řádovou mřížkou  $\varepsilon = 0.001_{(2)}$ ):

$$1.011 \times 2^2 + 1.110 \times 2^3 = (1.011 \times 2^{-1} + 1.110) \times 2^3 =$$

ztráta přesnosti

$$= (0.101 + 1.110) \times 2^3 = 10.011 \times 2^3 =$$

$$= 1.001 \times 2^4 \quad (\text{oseknutí do } \varepsilon)$$

ztráta přesnosti

Správný výsledek měl být je  $1.0011 \times 2^4$ .

## 11.2 Realizace matematických funkcí

Instrukční soubor ve většině případů neobsahuje instrukce pro výpočet matematických funkcí typu:  $\sqrt{x}$ ,  $\cos(x)$ , ... Tento problém je tedy nutné řešit programově.

### Funkce dané tabulkou (lookup table)

Nejjednodušší variantou je sestavit tabulku funkčních hodnot, kterou uložíme do pole. Tyto předpočítané funkční hodnoty jsou konstanty a pole tedy lze umístit do paměti programu.

indexy	hodnoty
$x_0$	$f(x_0)$
$x_1$	$f(x_1)$
$x_{N-1}$	$f(x_{N-1})$

Určení funkční hodnoty odpovídá indexování pole:  $f(x_i) = \text{pole}[x_i]$ .

### Numerický výpočet odmocniny

Pro numerický výpočet nejčastěji využíváme Newtonovu iterační metodu:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad (11-4)$$

---

Pro případ výpočtu  $\sqrt{A}$  hledáme takové  $x$ , pro které platí:  $x = \sqrt{A}$ .

Zápis upravíme tak, aby bylo možno do výpočtu dosadit hodnotu  $A$ , tedy:  $x^2 = A$ .

Funkce musí být v implicitním tvaru:  $f(x) = 0$ , po úpravě:  $x^2 - A = 0$ .

Tedy:  $f(x) = x^2 - A$ , derivace:  $f'(x) = 2x$ .

$$\text{Po dosazení do (11-4): } x_{i+1} = x_i - \frac{x_i^2 - A}{2x_i} = x_i - \left( \frac{x_i}{2} - \frac{A}{2x_i} \right) = \frac{x_i}{2} + \frac{A}{2x_i} = \frac{1}{2} \left( x_i + \frac{A}{x_i} \right).$$

Uvažujme  $A = 9$ , pro jednoduchost volíme  $x_0 = A$ , průběh výpočtu:

$$x_1 = 5; \quad x_2 = 3,4; \quad x_3 = 3,023529412; \quad x_4 = 3,000091554; \quad x_5 = 3.$$

Výsledek:  $\sqrt{9} = 3$ .

---

Nevýhodou uvedeného postupu je požadované dělení. Výpočet je však vhodný pro provádění na procesorech s celočíselnou aritmetikou.

Výpočet lze provést i jinak. Abychom se vyhnuli potřebě dělení, budeme hledat převrácenou hodnotu odmocniny, tedy:  $\frac{1}{y} = \sqrt{A}$ .

$$\text{Zápis upravíme tak, aby bylo možno dosadit hodnotu } A, \text{ tedy: } \frac{1}{y^2} = A.$$

Funkce musí být v implicitním tvaru:  $f(y) = 0$ , po úpravě:  $\frac{1}{y^2} - A = 0$ .

Tedy:  $f(y) = \frac{1}{y^2} - A$ , derivace:  $f'(y) = \frac{-2}{y^3}$ .

Po dosazení do (11-4):  $y_{i+1} = y_i - \frac{\frac{1}{y^2} - A}{\frac{-2}{y^3}} = y_i + \frac{1}{2}(y_i - Ay_i^3) = \frac{3y_i}{2} - \frac{Ay_i^3}{2} = \frac{1}{2}y_i(3 - Ay_i^2)$ .

Uvažujme  $A = 9$ , pro jednoduchost volíme  $y_0 = \frac{1}{A} = 0,111111$ , průběh výpočtu:

$$\begin{aligned} y_1 &= 0,160493827, & y_2 &= 0,222137547, & y_3 &= 0,283880033, & y_4 &= 0,322872253, \\ y_5 &= 0,332846031, & y_6 &= 0,333332265, & y_7 &= 0,333333333. \end{aligned}$$

Výsledek:  $\sqrt{9} = \frac{1}{y} = 3$ .

Výsledek lze též stanovit bez nutnosti dělení tak, že využijeme vlastností odmocniny:

$$\frac{1}{\sqrt{A}} = \frac{\sqrt{A}}{A} = \frac{1}{A \cdot y} \Rightarrow \sqrt{A} = A \cdot y = 9 \cdot 0,333333333 = 3$$

Výhodou uvedeného algoritmu je skutečnost, že není třeba provádět dělení.

## Goniometrické funkce

Pro výpočet hodnot goniometrických funkcí používáme tyto základní postupy:

1. rozvoj do Taylorovy řady,
2. algoritmus CORDIC,
3. min-max polynomiální approximace.

### 1. Rozvoj do Taylorovy řady

Funkci  $f(x)$  lze v okolí bodu A approximovat polynomem:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(A)}{n!} (x - A)^n \quad (11-5)$$

Například pro funkci  $\sin(x)$  v okolí nuly platí:

$$\sin(x) = \sum_{n=0}^{\infty} \frac{-1^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

Při praktické realizaci nahradíme dělení násobením. Tedy nebudeme dělit například  $3!$ , ale násobit  $0,1666667$ .

Výpočet funkční hodnoty na základě Taylorovy řady používají běžně překladače vyšších programovacích jazyků.

Počet členů rozvoje určuje cílovou přesnost výsledku. Například pro 8 platných cifer je třeba sečít prvních 5 členů:  $\sin(x) \doteq x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!}$ .

Počet provedených matematických operací lze ještě snížit úpravou na Hornerovo schéma. Výraz vlevo představuje 9 operací násobení a 4 operace součtu. Vpravo je pouze 6 násobení a 4 součty:

$$x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} = \frac{x}{9!} \left( 9! - x^2 \left( \frac{9!}{3!} - x^2 \left( \frac{9!}{5!} - x^2 \left( \frac{9!}{7!} - x^2 \right) \right) \right) \right)$$

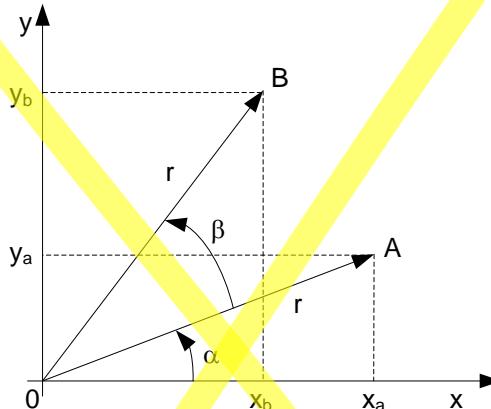
## 2. Algoritmus CORDIC (COordinate Rotation Digital Computer)

Princip algoritmu CORDIC spočívá v rotaci vektoru v rovině dle obr. 11.6. Pro hodnoty úseků  $x_a, y_a$  platí:

$$\begin{aligned} x_a &= r \cdot \cos(\alpha) \\ y_a &= r \cdot \sin(\alpha) \end{aligned} \quad (11-6)$$

Podobně pro úseky  $x_b, y_b$  platí:

$$\begin{aligned} x_b &= r \cdot \cos(\alpha + \beta) = r \cdot \cos(\alpha) \cdot \cos(\beta) - r \cdot \sin(\alpha) \cdot \sin(\beta) \\ y_b &= r \cdot \sin(\alpha + \beta) = r \cdot \sin(\alpha) \cdot \cos(\beta) + r \cdot \cos(\alpha) \cdot \sin(\beta) \end{aligned} \quad (11-7)$$



Obr. 11.6 K algoritmu CORDIC

Je-li  $\cos(\alpha) > 0$  a  $r = 1$ , lze dosadit do rovnic (11-6) a (11-7):

$$\begin{aligned} x_b &= \cos(\beta) \cdot (x_a - y_a \cdot \operatorname{tg}(\beta)) \\ y_b &= \cos(\beta) \cdot (y_a + x_a \cdot \operatorname{tg}(\beta)) \end{aligned} \quad (11-8)$$

Pokud dosadíme:  $x_a = 1$  a  $y_a = 0$  do rovnic (11-8), tedy jednotkový vektor ve směru osy x otočíme o  $\beta$ , dostaneme:

$$\begin{aligned} x_b &= \cos(\beta) \\ y_b &= \sin(\beta) \end{aligned} \quad (11-9)$$

Z toho je zřejmé, že pro výpočet funkcí cos a sin stačí znát hodnotu funkce  $\operatorname{tg}(\beta)$ . Otočení o  $\beta$  lze provést také po částech:

$$\beta = \varphi_1 + \varphi_2 + \dots + \varphi_N \quad (11-10)$$

$$x_i = \cos(\varphi_i) \cdot (x_{i-1} - y_{i-1} \cdot \operatorname{tg}(\varphi_i)) \quad (11-11)$$

$$y_i = \cos(\varphi_i) \cdot (y_{i-1} + x_{i-1} \cdot \operatorname{tg}(\varphi_i))$$

Při vlastní implementaci v procesoru bereme jako výhodnou volbu  $\operatorname{tg}(\varphi_i) = 2^{-i}$  pro  $i = 0, 1, \dots, N$ . Tomu odpovídá tabulka předpočítaných konstant uložených v paměti programu:

$\operatorname{tg}(\varphi_i)$	$\varphi_i [^\circ]$
$2^0$	45
$2^{-1}$	26,57
$2^{-2}$	14,04
$2^{-3}$	7,13
$2^{-4}$	3,58
$2^{-5}$	1,79
$2^{-6}$	0,89

atd.

Libovolný úhel  $\beta$  lze potom vytvořit sčítáním resp. odčítáním úhlů z tabulky. Počet řádků tabulky volíme dle požadované přesnosti. Úhel 45° je třeba připočítat vždy. Příklady:

$$21^\circ: 45 - 26,57 + 14,04 - 7,13 - 3,58 - 1,17 + 0,89 = 21,48^\circ$$

$$49^\circ: 45 + 26,57 - 14,04 - 7,13 - 3,58 + 1,17 + 0,89 = 48,88^\circ$$

$$75^\circ: 45 + 26,57 + 14,04 - 7,13 - 3,58 + 1,17 - 0,89 = 75,18^\circ$$

Při praktickém výpočtu vyjdeme z (11-11), pro případ přičítání  $\varphi_i$ :

$$\begin{aligned} K &= \cos(\varphi_1) \cdot \cos(\varphi_2) \cdots \cos(\varphi_N) \\ x_i &= x_{i-1} - y_{i-1} \cdot \operatorname{tg}(\varphi_i) \\ y_i &= y_{i-1} + x_{i-1} \cdot \operatorname{tg}(\varphi_i) \end{aligned} \tag{11-12}$$

Pro případ odčítání  $\varphi_i$  (otočíme znaménka):

$$\begin{aligned} x_i &= x_{i-1} + y_{i-1} \cdot \operatorname{tg}(\varphi_i) \\ y_i &= y_{i-1} - x_{i-1} \cdot \operatorname{tg}(\varphi_i) \end{aligned} \tag{11-13}$$

Vzhledem k tomu, že platí:  $\cos(\varphi_i) = \cos(-\varphi_i)$ , je hodnota  $\cos(\varphi_i)$  vždy kladná. Výsledné hodnoty jsou:

$$\begin{aligned} x_n &= \frac{\cos(\beta)}{K} \\ y_n &= \frac{\sin(\beta)}{K} \\ \text{neboli:} \end{aligned} \tag{11-14}$$

$$Kx_n = \cos(\beta)$$

$$Ky_n = \sin(\beta)$$

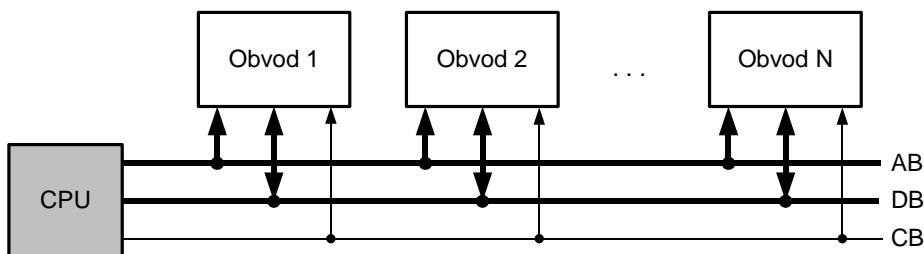
Výhodou algoritmu je skutečnost, že pro pevnou řádovou čárku není třeba násobička. Násobení  $2^{-i}$  totiž odpovídá posuvu doprava!

## 12 Sběrnice

V této kapitole doplníme informace ke sběrnicím. Dále se seznámíme se sériovými sběrnicemi, které se používají v souvislosti s mikroprocesory.

### 12.1 Základní vlastnosti

**Sběrnice** je soustava vodičů propojujících jednotlivé části počítače. Komunikace probíhá různými směry v různých časech.



Obr. 12.1 Sběrnice

Výhody:

- snadné přidávání dalších obvodů (paměti, zařízení),
- zařízení lze přidat do různých počítačů vybavených stejnou sběrnicí,
- snížení nákladů.

Nevýhoda:

- nižší rychlosť komunikace oproti přímému připojení.

Jednotlivé části:

- **DB** (Data Bus – datová sběrnice) – nese užitečnou informaci,
- **AB** (Address Bus – adresová sběrnice) – adresuje zařízení,
- **CB** (Control Bus – řídicí sběrnice) – obsahuje řídicí signály: čtení/zápis, požadavky na přenos, potvrzení přenosu, indikace chyb.

Master a Slave

**Master** je obvod, který řídí sběrnici a zahajuje komunikaci na sběrnici. Rozlišujeme sběrnice typu **Single master** (funkce mastera je pevně přidělena) a **Multimaster**.

V případě multimaster sběrnice musí systém obsahovat obvod označovaný jako **arbitr** (dohlíží na přidělení sběrnice mezi několika mastery). Na sběrnici musí být generovány signály typu: „požadavek na přenos“ (master si vyžaduje pozornost a hodlá převzít řízení sběrnice), „sběrnice přidělena“ (arbitr potvrzuje přidělení sběrnice pro master).

**Slave** je podřízený obvod, tedy zařízení, které je na adresování a lze nad ním uskutečňovat přenos dat.

Komunikační protokol, transakce

**Komunikační protokol** je specifikace posloupnosti kroků a jejich časování nutné pro přenos informace. **Sběrnicový cyklus** neboli **transakce** je komunikace mezi dvěma zařízeními.

Typická transakce probíhá takto:

1. Několik zařízení má požadavek přístupu na sběrnici, jedno je vybráno, stane se z něj master.
2. Master umístí na sběrnici adresu a určí typ transakce:
  - **zápis dat**: master → slave,
  - **čtení dat**: slave → master,
3. Proběhne přenos dat.
4. Potvrzení dokončení transakce a uvolnění sběrnice.

**Přenosová rychlosť** určuje množství přenesené informace za jednotku času.

Například u sběrnice PCI: 133 MB/s, pro UART (sériové porty) např. 9600 b/s.

### Typy sběrnic

Sběrnice používané v **klasickém PC** rozdělujeme podle účelu:

- **systémové** – například pro vzájemné propojení procesoru a pamětí,
- **periferní** – pomocí konektorů umístěných na základní desce připojujeme zařízení dle sběrnice: ISA, PCI, PCIe, PATA, SATA,
- **externí** – zařízení jsou připojena mimo systém (USB, RS232, Ethernet...).

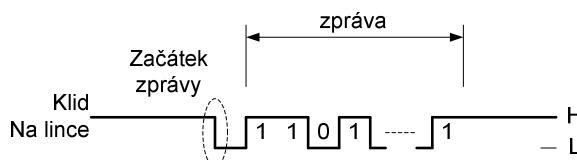
**Jednoúčelové systémy** obvykle používají jedinou single master systémovou sběrnici, kde procesor zastává funkci mastera.

Sběrnice rozdělujeme podle množství přenášených bitů:

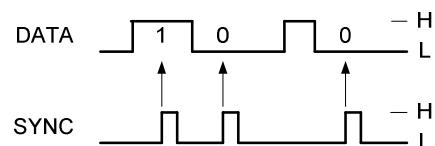
- **paralelní** – počet datových vodičů odpovídá počtu přenášených bitů; sběrnice dosahují velkých rychlostí, ale jsou drahé,
- **sériové** – přenos se provádí po bitech; přenosová rychlosť je nižší, toto řešení je však cenově výhodnější.

Sériové sběrnice dále rozdělujeme podle synchronizace přenosu:

- **asynchronní** (viz obr. 12.2):
  - synchronizační (hodinový) signál se nepřenáší,
  - provádí se pouze „rámcová“ synchronizace po přenesení jedné zprávy,
  - komunikující zařízení musí mít domluven formát a rychlosť přenosu,
  - příklady: UART, USB, SATA.



Obr. 12.2 Asynchronní sériová komunikace na UART



Obr. 12.3 Příklad synchronní sériové komunikace

- **synchronní** (viz obr. 12.3):
  - přenáší se synchronizační signál,
  - synchronizace probíhá na úrovni každého bitu,
  - příklady: SPI, I<sup>2</sup>C.

Zvláštní skupinu tvoří tzv. **průmyslové sběrnice**. Tyto sběrnice jsou určeny pro propojování zařízení na velké vzdálenosti, z tohoto důvodu se konstruují jako sériové (levná kabeláž). Musí být zajištěna odolnost proti rušení a provedeno zabezpečení přenosu. Nejznámější varianty jsou: RS232, RS485, CAN.

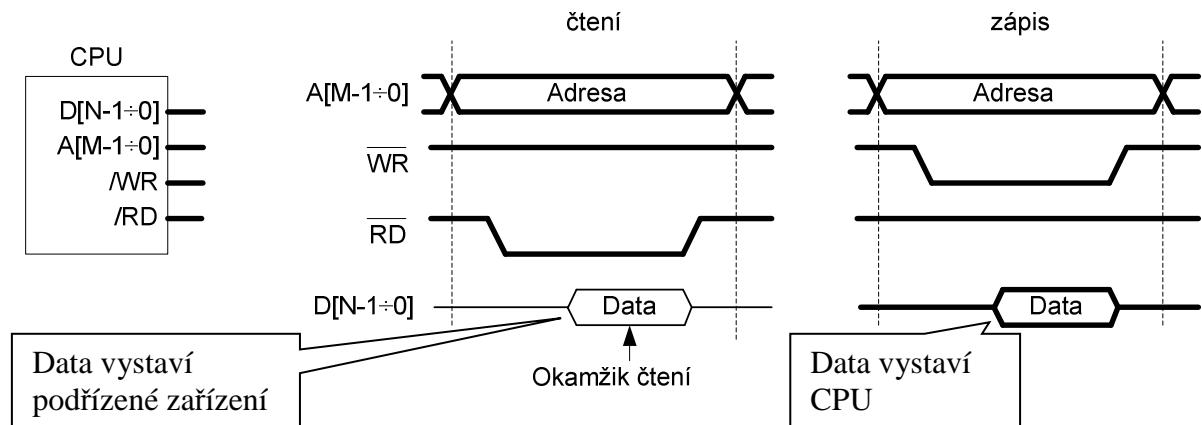
## Zařízení připojitelná k mikroprocesoru

Dále se seznámíme se zařízeními, která se nejčastěji připojují pomocí sběrnice do mikroprocesorového systému.

### Procesor

Je-li procesor vybaven adresovou, datovou a řídicí sběrnicí, můžeme k němu snadno připojit další zařízení.

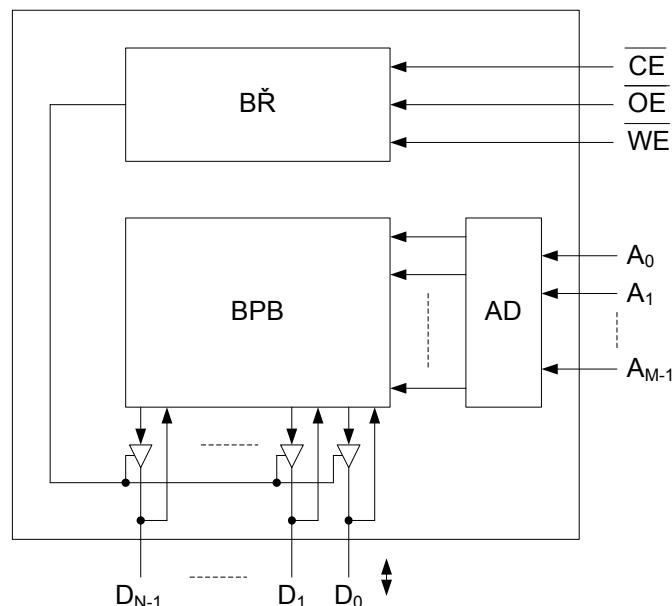
Na obr. 12.4 je uveden procesor s adresovou sběrnicí šíře M bitů, datovou sběrnicí šíře N bitů a řídicí sběrnicí představovanou signály  $\overline{RD}$  (požadavek na čtení) a  $\overline{WR}$  (požadavek na zápis).



Obr. 12.4 Procesor vybavený adresovou, datovou a řídicí sběrnicí pro připojení zařízení

### SRAM (Statická RAM)

SRAM používáme například pro zvýšení velikosti datové paměti procesoru.



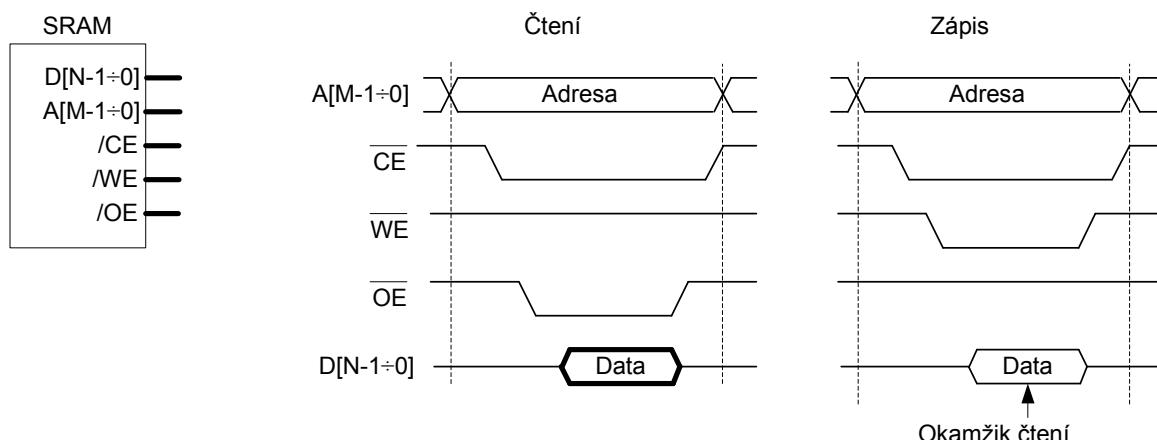
Obr. 12.5 Blokové schéma SRAM

Blokové schéma dle obr. 12.5 je velmi podobné jako dříve uvedené schéma dle obr. 1.17. Pouze používá více řídících signálů:

- $\overline{CE}$  též  $\overline{CS}$  (Chip Enable, Chip Select) – slouží pro výběr obvodu,
- $\overline{OE}$  (Output Enable) – povoluje třístavové budiče datové sběrnice,
- $\overline{WE}$  (Write Enable) – povolení zápisu:
  - 0 = zápis do paměti,  $D[N-1:0]$  = vstupy,
  - 1 = čtení z paměti,  $D[N-1:0]$  = výstupy (ale až po  $\overline{OE} = 0$ ).

Z obr. 12.6 je zřejmé, že při **čtení** z paměti musí být nejprve přivedena adresa, je vybrána operace čtení ( $\overline{WE} = 1$ ) dále se paměť aktivuje ( $\overline{CE} = 0$ ) a jsou aktivovány třístavové budiče ( $\overline{OE} = 0$ ). Paměť vystaví přečtená data na datovou sběrnici.

**Zápis** začíná opět přivedením adresy, aktivací obvodu ( $\overline{CE} = 0$ ) a výběrem operace zápisu ( $\overline{WE} = 0$ ). Třístavové oddělovače zůstanou vypnuty ( $\overline{OE} = 1$ ) a vnější obvod (procesor) přiveze data pro zápis. Zápis fakticky proběhne při náběžné hraně signálu  $\overline{WE}$ .



Obr. 12.6 Bloková značka SRAM, signály při čtení a zápisu

### Adresace

Je-li na sběrnici připojen pouze jeden obvod, může být jeho výběrový signál trvale aktivní ( $\overline{CS} = 0$ ).

V případě připojení více obvodů se provádí **adresace**. Pro zajištění této techniky je třeba:

- každý obvod (slave) musí mít vlastní výběrový signál  $\overline{CS}$ ,
- na základě požadavku procesoru generuje adresový dekodér jednotlivé signály  $\overline{CS}$  tak, aby byl aktivní vždy pouze jeden obvod,
- dochází k tomu, že se adresový prostor rozdělí mezi jednotlivé obvody.

Příklad konstrukce adresovací logiky je uveden na obr. 12.7.

Dekodér typu **74138** pracuje jako dekodér 1 z 8 (pro 3bitové číslo CBA, které odpovídá osmi adresám, aktivuje jeden z výstupů označených jako 0 až 7). Výstupy jsou aktivní v log. 0.

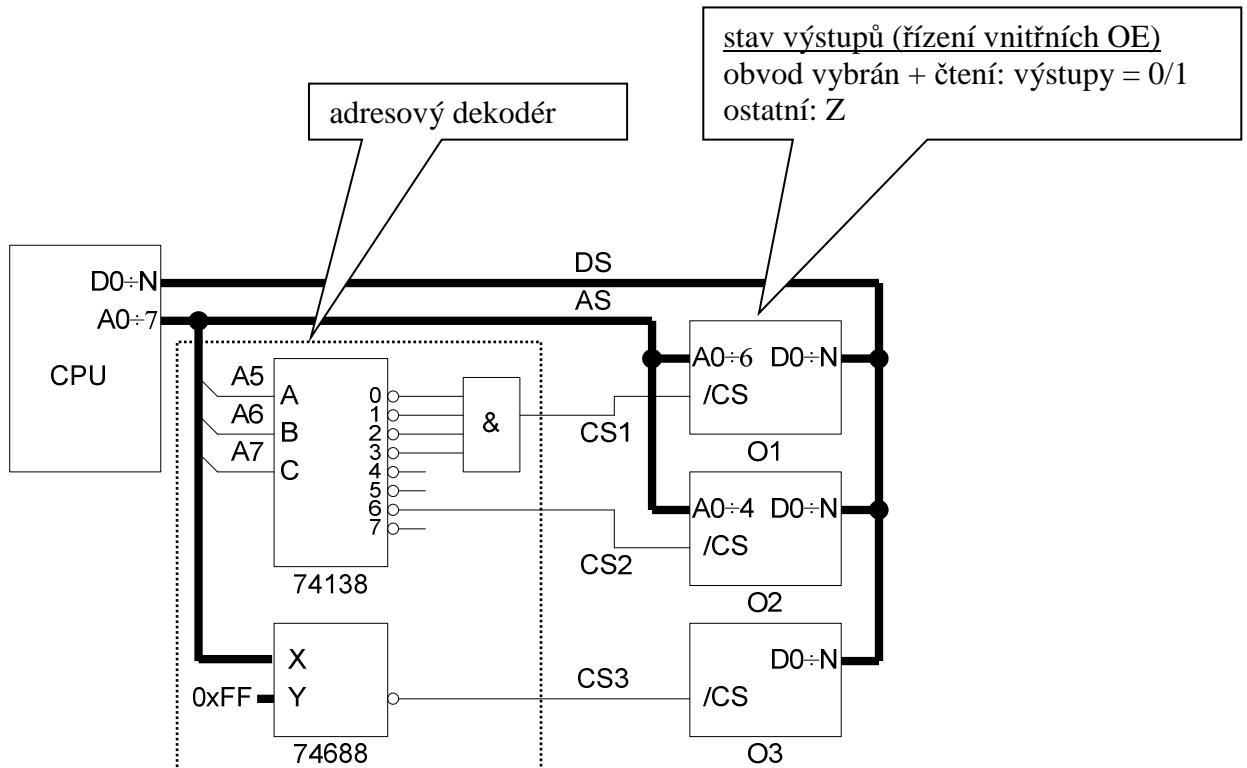
**Obvod O1** má aktivní signál CS1 (log. 0) v případě, že je aktivován alespoň jeden z výstupů 0 až 3 (při použití „obrácené“ logiky funguje hradlo AND jako koncentrátor signálů). Aktivace výstupů 0 až 3 nastane pro kombinace

$A7:A6:A5 = 0XX$ . Tedy pro adresy 0 až 127. Spodních 7 bitů adresové sběrnice je pak připojeno přímo na vývody A6..0.

**Obvod O2** používá pouze 5bitovou adresu A4..0. Signál CS2 je aktivní (log. 0) pro kombinaci 6, neboli pro  $A7:A6:A5 = 110$ . Adresy obvodu O2 jsou tedy  $11000000_{(2)}$  až  $11011111_{(2)}$ , neboli 192 až 223.

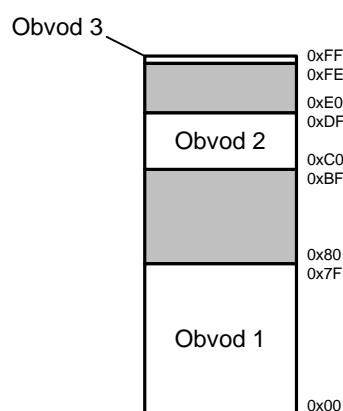
Dekodér typu **74688** pracuje jako komparátor 8bitových čísel. Použitý negovaný výstup je aktivní v případě, že adresa odpovídá  $A7:0 = 255$ .

**Obvod O3** má tedy signál CS3 aktivní (log. 0) pouze pro adresu 255.



Obr. 12.7 Příklad konstrukce adresovací logiky

Pro úplnost ještě doplňujeme mapu adresového prostoru, která uvádí využití adres jednotlivými obvody a dále volný adresní prostor (šedou barvou). Viz obr. 12.8.



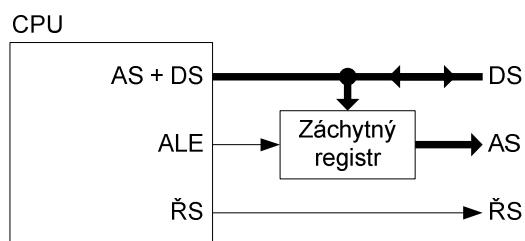
Obr. 12.8 Mapa adres pro systém dle obr. 12.7

## Multiplex

Pro snížení počtu vývodů procesoru a tedy i snížení konečné ceny čipu se používá technika **multiplexování adresové a datové sběrnice**.

Vývody v určitém čase představují část adresové sběrnice, v jiném čase zase datovou sběrnici. Pro praktické použití musí být signály adresové sběrnice „zachyceny“ pomocí vnějšího záchytného registru, procesor indikuje požadavek zachycení signálem označeným obvykle **ALE** (Address Latch Enable).

Řešení multiplexované sběrnice je uvedeno na obr. 12.9. Sloučená adresová a datová sběrnice vystupuje z procesoru. V okamžiku, kdy je na vodičích vystavena adresová sběrnice, aktivuje procesor signál ALE. Tím dojde k zachycení adresy v záchytném registru. Adresa je „podržena“ na výstupu záchytného registru a spolu s daty směřuje na periferii. Řídící signály jsou pak vytvářeny tak, aby se provedla operace čtení nebo zápisu.

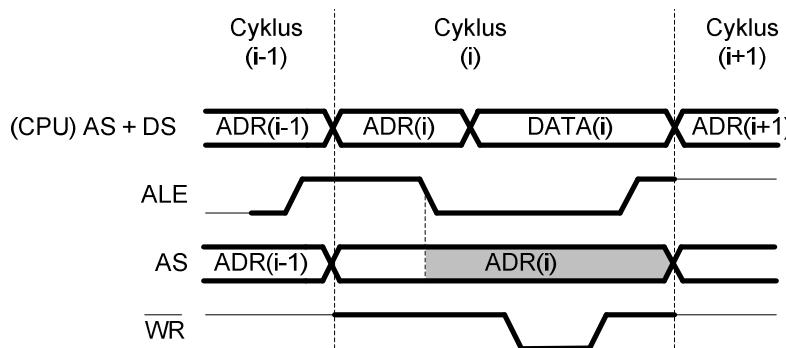


Obr. 12.9 Princip multiplexované adresové a datové sběrnice

Pro lepší informaci je doplněn obr. 12.10, zde je uveden příklad transakce při zápisu na periferii.

Vývody AS+DS obsahují na začátku cyklu adresu. Signál ALE je aktivní a adresa tedy prochází záchytným registrem. V okamžiku sestupné hrany signálu ALE je hodnota adresy již trvale uložena.

Nyní může být na vývody AS+DS „přepnuta“ datová sběrnice s daty pro zápis. Pomocí aktivního signálu WR se periferii sdělí, že má být proveden zápis.



Obr. 12.10 Příklad transakce na multiplexované sběrnici (zápis)

Na obr. 12.11 je uvedeno konkrétní řešení multiplexu sběrnice pro mikrokontrolér **ATmega128**, který dovoluje připojení vnější datové paměti.

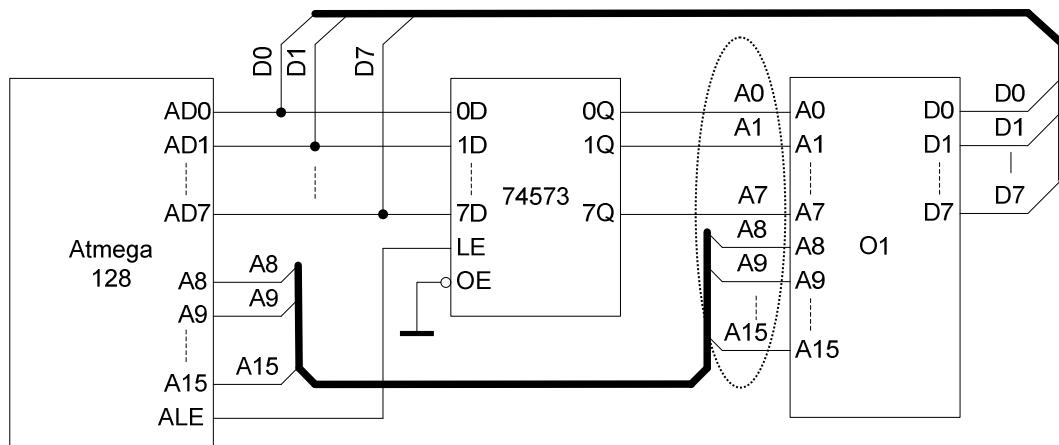
Vývody A15..8 představují vyšších 8bitů adresové sběrnice. Vývody AD7..0 představují multiplexovanou datovou sběrnici D7..0 a adresové vodiče A7..0 (tedy spodní část adresové sběrnice). Signál ALE slouží pro záchyt adresy.

Vidíme, že jako záchytný registr byl použit obvod typu **74573**, což je 8násobný klopný obvod typu D řízený úrovní. Pro LE = 1 je cesta z datových vstupů D

na výstupy Q volná. Pro  $LE = 0$  obvod „pamatuje“ předchozí stav. Výstupy jsou tedy připojeny na spodních 8 bitů adresové sběrnice. Horních 8 bitů adresové sběrnice je připojeno přímo na vývody A15..8 mikrokontroléru.

Datová sběrnice obvodu D7..0 je pak přivedena přímo na vývody AD7..0 mikrokontroléru.

Pro zjednodušení nejsou ve schématu zakresleny další řídicí signály.



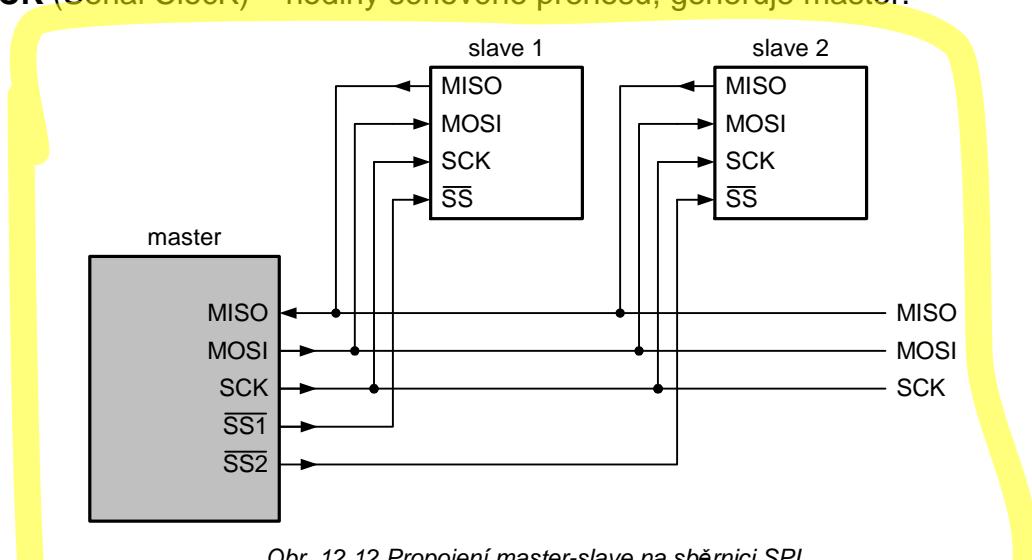
Obr. 12.11 Schéma připojení periferie na multiplexovanou sběrnici mikrokontroléru ATmega128

## 12.2 SPI (Serial Peripheral Interface)

SPI zajišťuje vysokorychlostní sériový synchronní přenos dat mezi mikrokontrolérem a periferním zařízením nebo jinými mikrokontroléry, které jsou vybaveny SPI sběrnicí.

SPI je 3vodičová single master sériová synchronní sběrnice. V případě, že je připojen více než jeden obvod typu slave, je nutná adresace. Potom má každý obvod slave navíc výběrový vodič  $\bar{SS}$  (Slave Select). Vodiče sběrnice mají význam:

- **MISO** (Master In/Slave Out) – vstup sériových dat pro master/výstup sériových dat od slave,
- **MOSI** (Master Out/Slave In) – výstup sériových dat od master/vstup sériových dat pro slave,
- **SCK** (Serial Clock) – hodiny sériového přenosu, generuje master.



Obr. 12.12 Propojení master-slave na sběrnici SPI

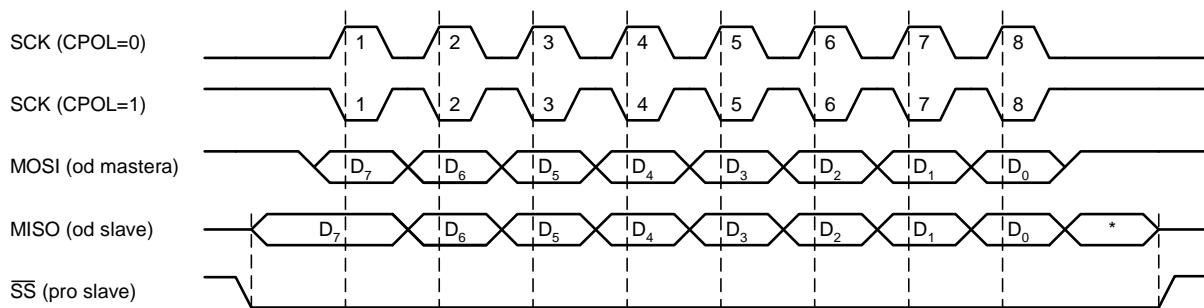
Existují čtyři kombinace fáze a polarity **SCK** hodin, které určují bity **CPHA** (Clock PHAse) a **CPOL** (Clock POLarity) dle tab. 12.1, viz obr. 12.13, 12.14.

Tab. 12.1 SPI režimy

CPOL	CPHA	Vzorkovací hraná	Vystavovací hraná	SPI režim
0	0	náběžná	sestupná	0
0	1	sestupná	náběžná	1
1	0	sestupná	náběžná	2
1	1	náběžná	sestupná	3

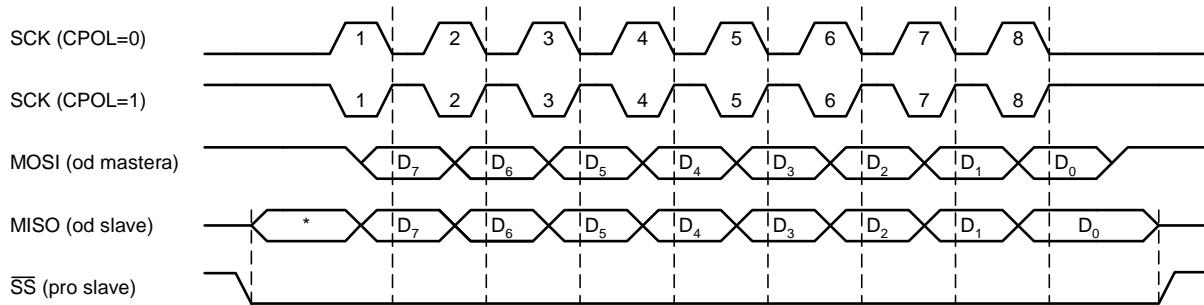
Dále je možné určit pořadí bitů pomocí **DORD** (Data ORDer) buď LSB..MSB (od nejnižšího po nejvyšší) nebo MSB..LSB (od nejvyššího po nejnižší). Níže uvedené obrázky jsou kresleny pro případ **DORD = 0** (začíná se nejvyšším bitem).

Pro **CPHA = 0** jsou při CPOL = 0 data vzorkována náběžnou hranou SCK. Pro CPOL = 1 je vzorkování prováděno sestupnou hranou SCK. CPOL dále určuje stav SCK při neaktivním přenosu (SCK = 0 pro CPOL = 0, SCK = 1 pro CPOL = 1).



Obr. 12.13 Formát SPI přenosu pro CPHA = 0 (\* nedefinováno, obvykle se jedná o právě přijatý nejvyšší bit)

Pro **CPHA = 1** jsou při CPOL = 0 data vzorkována sestupnou hranou SCK. Pro CPOL = 1 je vzorkování prováděno náběžnou hranou SCK. CPOL dále určuje stav SCK při neaktivním přenosu (SCK = 0 pro CPOL = 0, SCK = 1 pro CPOL = 1).



Obr. 12.14 Formát SPI přenosu pro CPHA = 1 (\* nedefinováno, obvykle se jedná o dříve vyslaný nejnižší bit)

### Podpora SPI v mikrokontrolérech

Vytvoření signálů sběrnice SPI je možné provést bez velkých komplikací programově. Moderní mikrokontroléry mají obvykle zabudován vlastní SPI kanál, který po příslušné konfiguraci provádí komunikaci dle povelů programu.

Například mikrokontroléry ATmega mají SPI sběrnici řízenou pomocí trojice registrů: **SPCR** (konfigurační registr), **SPDR** (datový registr pro odeslání nebo příjem jednoho bajtu) a **SPSR** (stavový registr – indikuje případné chyby komunikace).

## Příklady SPI obvodů

Variabilnost a jednoduchost SPI dovoluje připojovat rozličné obvody původně určené pro jiné účely (například posuvné registry SIPO – umožní zvýšit počet výstupů nebo posuvné registry PISO – umožní zvýšit počet vstupů) případně specializované obvody navržené přímo pro sběrnici SPI. Přehled několika vybraných obvodů je uveden formou tab. 12.2.

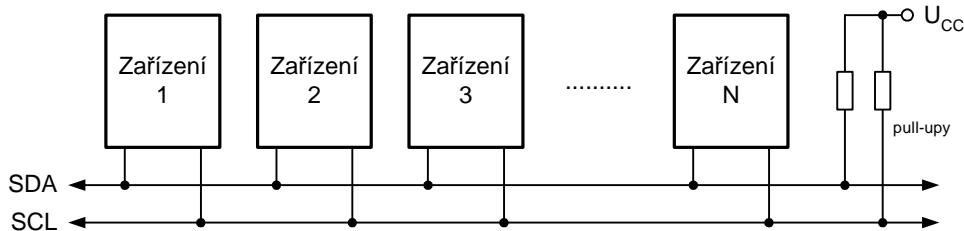
Tab. 12.2 Vybrané obvody kompatibilní se sběrnicí SPI

Obvod	Fukce
<b>74HCT595</b>	8bitový SIPO
<b>74HCT589</b>	8bitový PISO
<b>93C66</b>	E <sup>2</sup> PROM 256×16bitů nebo 512×8bitů
<b>ADC0831</b>	jednokanálový 8bitový A/D převodník
<b>ADC0834</b>	4kanálový 8bitový A/D převodník
<b>MCP3002</b>	2kanálový 10bitový A/D převodník
<b>M5451B7</b>	budič pro 35 segmentů LED
<b>TLC549</b>	jednokanálový 8bitový A/D převodník

## 12.3 I<sup>2</sup>C (Inter-Integrated Circuit)

I<sup>2</sup>C je dvouvodičová sériová synchronní multimaster sběrnice používaná pro připojování nízkorychlostních periferií. Sběrnici vyvinula firma Philips.

Linka **SDA** (Serial DAta) slouží pro přenos dat. Linka označená **SCL** (Serial CLock) představuje hodinový signál přenosu.



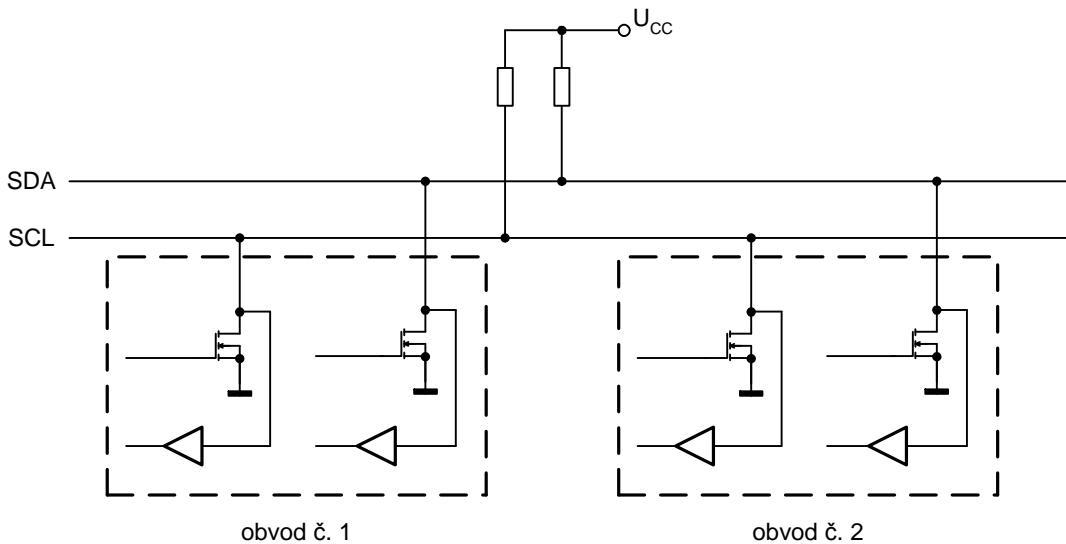
Obr. 12.15 Připojení zařízení na sběrnici I<sup>2</sup>C

Obě linky musí být připojeny na kladný pól napájecího napětí prostřednictvím tzv. zdvihačích rezistorů (pull-up, jedná se vlastně o výstup typu **otevřený kolektor**). Tím je zajištěna práce linek SDA a SCL v obou směrech. Pokud by došlo ke kolizi (chtělo by vysílat více obvodů), poškodí se pouze úrovně signálů a nikoli vysílající obvody. Zpětnou vazbou je zajištěno, že každý obvod může pracovat jako vysílač i přijímač. Viz obr. 12.16.

## Vzájemné elektrické propojení

Jak je zřejmé z obr. 12.16, jsou obě linky připojeny na napájecí napětí přes pull-up rezistory. Sběrnicové budiče všech zařízení mají provedení typu otevřený kolektor. Tato schopnost implementuje funkci montážního součinu, která je pro práci tohoto rozhraní nezbytná. Log. 0 na lince je generována, když jedno nebo více zařízení vybaví výstup do log. 0. Log. 1 se na lince objeví jen v tom případě, když mají všechna zařízení své výstupy ve třetím stavu. Pak se totiž uplatní pull-up rezistory, které „visící“ linky „vytáhnou“ směrem k log. 1.

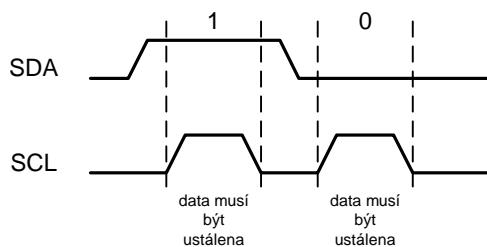
Počet zařízení, která lze připojit ke sběrnici, je omezen pouze maximální povolenou parazitní kapacitou 400 pF. Existují dvě základní rychlosti: SLOW (100 kHz) a FAST (400 kHz).



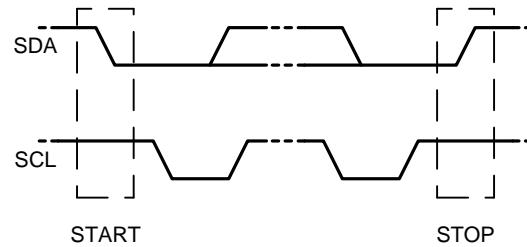
Obr. 12.16 Detail připojení vývodů obvodů na sběrnici  $I^2C$

### Přenos bitů

Každý datový bit přenášený po  $I^2C$  je spojen s jedním pulzem hodin. Úrovně na datové lince (SDA) musí být stabilní v okamžiku, kdy  $SCL = 1$ , viz obr. 12.17. Toto pravidlo platí vyjma případu, že se generuje tzv. START nebo STOP stav (viz níže).



Obr. 12.17 Přenos bitů



Obr. 12.18 START a STOP stavů

### START a STOP stavů

Přenos dat je zahájen, když master vloží tzv. START stav. START stav odpovídá sestupné hraně SDA v okamžiku, kdy je  $SCL = 1$ .

Přenos je ukončen v okamžiku, když master vloží tzv. STOP stav. STOP stav odpovídá náběžné hraně SDA v okamžiku, kdy je  $SCL = 1$ .

### Formát adresního paketu

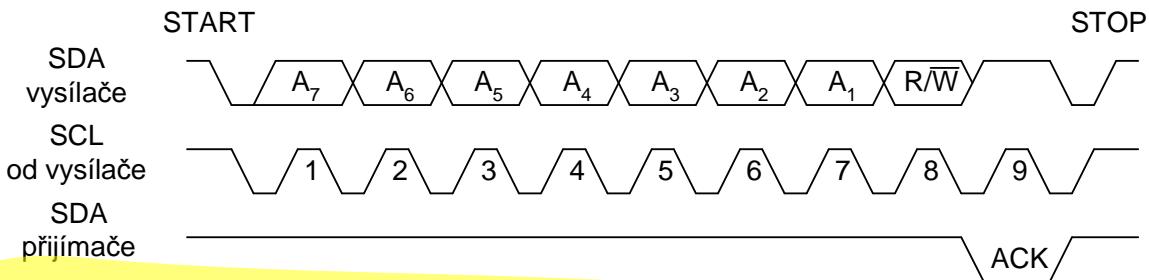
Pro adresování zařízení se nepoužívají výběrové vodiče, adresa slave se posílá podobně jako ostatní data. Všechny adresní pakety jsou na  $I^2C$  sběrnici vysílány v délce 9 bitů. Viz obr. 12.19.

Skládají se ze: 7bitové adresy, řídicího bitu  $R/W$  a potvrzovacího bitu ACK.

Je-li bit  $R/W = 1$ , jedná se o čtecí operaci (Read). Je-li bit  $R/W = 0$ , jedná se o zápis (Write).

Když slave rozpozná, že je adresován, měl by stáhnout SDA linku v okamžiku potvrzovacího bitu ACK (tedy v 9. cyklu hodin). Je-li adresovaný slave zaneprázdněn nebo když žádné zařízení neodpovídá, je linka SDA ponechána v log. 1. Master pak může vyslat STOP stav nebo opakovat START stav pro inicializaci nového přenosu.

Použitá 7bitová adresa dovoluje rozlišit až 128 různých obvodů.



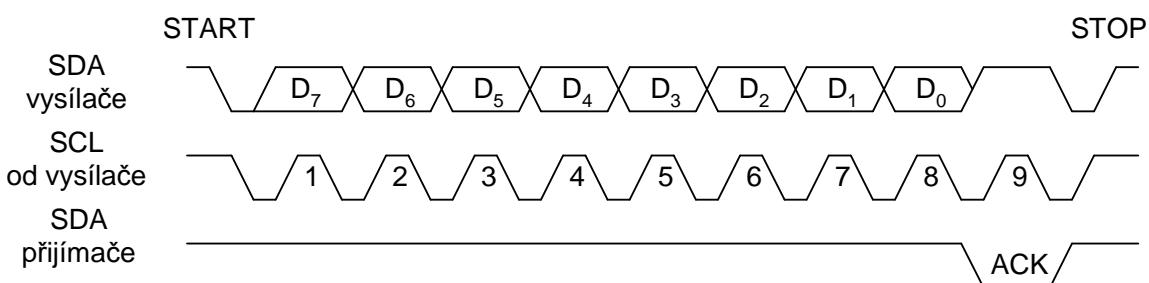
Obr. 12.19 Formát adresního paketu

### Formát datového paketu

Datové pakety se po TWI sběrnici vysílají rovněž v délce 9 bitů. Viz obr. 12.20.

Skládají se z: 8bitového datového údaje a potvrzovacího bitu ACK.

V 8bitových datech se jako první vysílá nejvíce významný bit (MSB). Každý úspěšně přijatý bajt je ze strany přijímače potvrzen stažením linky SDA v 9. cyklu hodin (ACK).



Obr. 12.20 Formát datového paketu

### Adresy I<sup>2</sup>C zařízení

Adresy I<sup>2</sup>C zařízení sestávají z *pevné části*, která je hardwarově zabudovaná v obvodu. Například obvod PCF8591 má čtyři nejvyšší bity adresy vždy rovny 1001.

	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	R/W
PCF8574	0	1	0	0	v	v	v	0/1
PCF8591	1	0	0	1	v	v	v	0/1
TDA8444	0	1	0	0	v	v	v	0

Obr. 12.21 Adresování obvodů PCF8574, PCF8591 a TDA8444

Protože ke sběrnici I<sup>2</sup>C můžeme chtít připojit několik obvodů stejného typu, je část adresy *volitelná*. Obvykle se jedná o 3 bity. Pak lze připojit až osm obvodů stejného typu na sdílenou sběrnici. Tato část adresy je v obr. 12.21 označena symbolem v (jako volitelná). Volitelná část adresy se nastavuje připojením subadresovacích vstupů na log. 1 nebo 0.

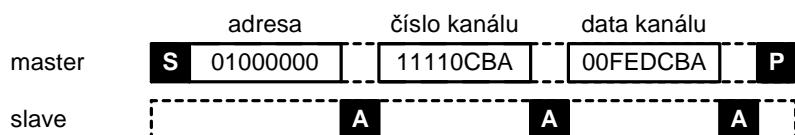
Nejnižší bit R/W určuje *směr přenosu*. Pro R/W=1 je obvod vysílačem (čteme z něj data), pro R/W=0 pracuje obvod jako přijímač (zapisujeme do něj data). Obvody, které pracují v obou směrech, tedy rozlišují dvě adresy (adresa pro vysílání je o jedničku vyšší než pro příjem). Obvod TDA8444 může data pouze přijímat, proto je na obr. 12.21 ve sloupci R/W jen hodnota 0.

## Příklad komunikace s TDA8444

Obvod TDA8444 obsahuje 8kanálový D/A převodník s rozlišením 6 bitů (tedy v rozsahu 0 až 63).

Při komunikaci s obvodem se nejdříve vyšle adresa (uvažujme, že subadresovací vodiče jsou připojeny na log. 0), dále číslo kanálu a nakonec data pro zvolený kanál, viz obr. 12.22.

Adresa pro zápis je 01000000 (viz obr. 12.21). Při zadávání čísla kanálu je nejvyšších pět bitů 11110 (číslo kanálu tvoří spodní 3 byty). Data kanálu jsou 6 bitová (tedy nejvýznamnější 2 byty jsou nulové).



Obr. 12.22 Příklad komunikace s TDA8444

## Podpora I<sup>2</sup>C v mikrokontrolérech

Vytvoření signálů sběrnice I<sup>2</sup>C je možné provést programově, hlavním omezením je nízká rychlosť komunikace po této sběrnici, takže strojový čas procesoru je z velké části tvořen prodlevami komunikace. Proto jsou moderní mikrokontroléry vybaveny jednotkou kompatibilní s I<sup>2</sup>C. Z licenčních důvodů (licence firmy Philips) se však obvykle označuje jinou zkratkou.

Například mikrokontroléry ATmega mají zabudovánu jednotku **TWI** (Two-wire Serial Interface), která je kompatibilní s I<sup>2</sup>C. Jednotka je řízena registry: **TWCR** (konfigurační registr), **TWBR** (registr pro nastavení přenosové rychlosti), **TWDR** (datový registr pro odeslání nebo příjem jednoho bajtu adresy/dat), **TWSR** (stavový registr – indikuje případné chyby komunikace), **TWAR** (adresový registr – používá se v případě, že mikrokontrolér pracuje jako slave). Obsluhu lze řešit pomocí přerušení a tím zužitkovat plně výpočetního výkonu mikrokontroléru.

## Příklady I<sup>2</sup>C obvodů

Tab. 12.3 uvádí vybrané příklady obvodů pracujících s I<sup>2</sup>C sběrnicí.

Tab. 12.3 Přehled vybraných obvodů vybavených sběrnicí I<sup>2</sup>C

Obvod	Funkce
<b>PCF8574</b>	8bitový vstup/výstup
<b>PCF8577</b>	budič 64segmentového LCD
<b>24C00</b>	E <sup>2</sup> PROM 128 bitů
<b>PCF8583</b>	hodiny/kalendář a paměť RAM
<b>PCF8591</b>	8bitový A/D, D/A převodník
<b>SAA1064</b>	budič 4místného LED displeje
<b>MCP23016</b>	16bitový expandér vstupů/výstupů
<b>TDA8444</b>	8kanálový 6bitový D/A převodník

# 13 Vývoj a současné trendy mikroprocesorové techniky

V této kapitole stručně shrneme historii mikroprocesorové techniky. Dále doplníme informace k vnitřní stavbě CPU, architekturám CISC a RISC, orientaci instrukčního souboru, paměťovému subsystému a zvyšování výkonu procesoru.

## 13.1 Historie

Připomeňme generace procesorů tak, jak se postupně vyvíjely:

### I. generace

- u firmy Intel vznikl nápad převést řešení úloh z jednoúčelových obvodů na programovatelné integrované obvody.
- 1971 – Intel I4004 – 108 kHz, 4bitový, 2 300 tranzistorů, pro kalkulačky,
- 1972 – Intel I8008 – 8bitové provedení Intel I4004, 45 instrukcí.

### II. generace

- 8bitové procesory, 10násobně rychlejší než v I. generaci díky použití nové výrobní technologie,
- 1974 – Intel 8080 – první použitelný CPU, světový standard, adresace 64 kB paměti, pracovní kmitočet až 2 MHz,
- 1975 – Motorola MC6800.

### III. generace

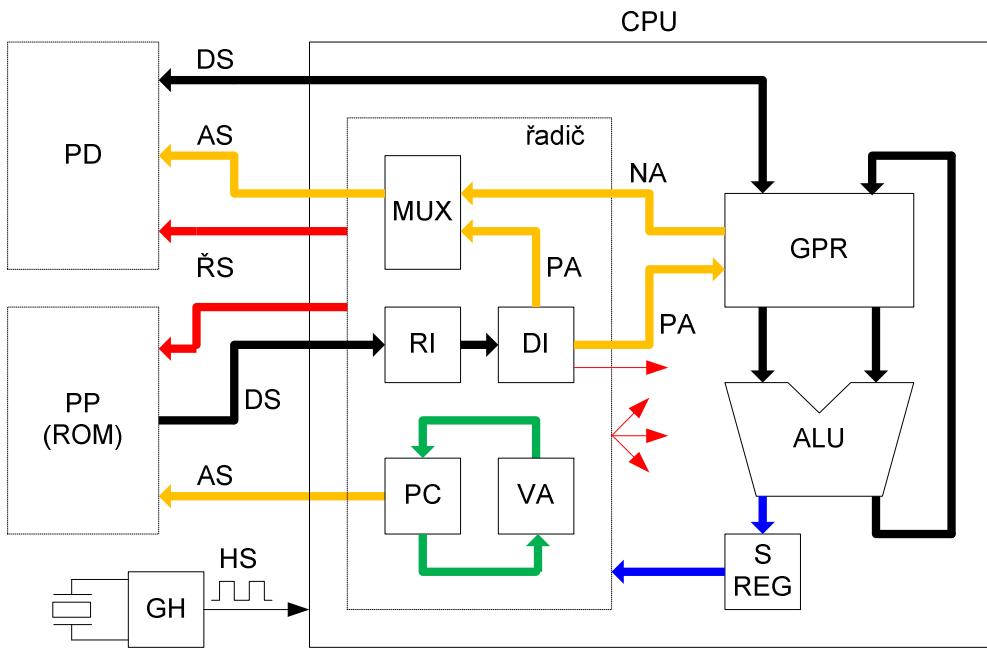
- nové instrukce, módy adresování, integrace některých podpůrných funkcí na čip mikroprocesoru (hodiny, vstupy/výstupy),
- 1976 – Zilog Z80 – stále 8bitový, rozvíjí architekturu I8080.

### IV. generace – diferenciace dle aplikací

- mikroprocesory pro platformu PC:
  - přechod na 16bitové procesory,
  - 1978 – Intel 8086, 29 000 tranzistorů, adresace 1MB,
  - dále: 80386, 80486, Pentium...
- mikroprocesory pro řídicí aplikace – Embedded systems:
  - rozvoj 8bitové architektury (integrace všeho na jeden čip: vznikají mikrořadiče resp. jednočipové mikrokontroléry) – I8048 a z něj vznikl I8051, řada PIC od Microchip,
  - i zde další diferenciace a zlepšování:
    - 8bitové – Atmel AVR
    - 16bitové řídicí mikroprocesory – pro výpočetně náročnější aplikace I80196, 80C166,
    - 32bitové řídicí mikroprocesory – ARM7, ARM9 (mobilní telefony),
  - DSP – digital signal processing (zvuk, video, GSM, radary, lékařství apod.), například Texas Instruments řada TMS, Analog Devices SHARC,
  - mobilní multimédia (ipod, atd.) – ARM11.

## 13.2 Blokové schéma procesoru

Obecné blokové schéma procesoru odpovídající Harvardské architektuře se dvěma sběrnicemi je připomenuto na obr. 13.1.



Obr. 13.1 Blokové schéma CPU

Systém je řízen hodinovým signálem (HS) z generátoru hodin (GH). Řídicí signály jsou signály požadující provedení operace. Stavové signály nesou informace o stavu jednotky. ALU provádí aritmetické a logické operace nad vstupními daty.

**Řadič** obsahuje:

- **registr instrukce (RI):**
  - obsahuje aktuální, právě vykonávanou instrukci, podle jejího obsahu řídí dekodér instrukce (DI) činnost procesoru,
- **dekodér instrukce (DI)** – dekóduje obsah RI, dle konstrukce rozlišujeme:
  - **klasický**: sekvenční log. obvod (stavový automat) sestavený z hradel; rychlejší, komplikovanější (= dražší) – u RISC CPU,
  - **mikroprogramovaný**: instrukce rozkládány na mikrooperace (mikroinstrukce); levnější, pomalejší – u CISC CPU.

**Všeobecné registry** (General Purpose Register) představují soubor registrů (register file – registrové pole) pro mezivýsledky a lokální proměnné. Slouží jako vstupy do ALU pro realizaci aritmetických a přesunových operací. ALU výsledek uloží zpět do jednoho GPR.

**Stavový registr SREG**: ALU ukládá informace o výsledcích operací.

**Programový čítač** (Program Counter PC) – obsahuje adresu právě vykonávané resp. následující instrukce (jedná se o ukazatel do paměti programu). Změna pořadí provádění instrukcí je zajištěna manipulací s PC (přes VA – obvod pro výpočet adresy). Takto se realizují skoky, větvení, volání podprogramů, ...

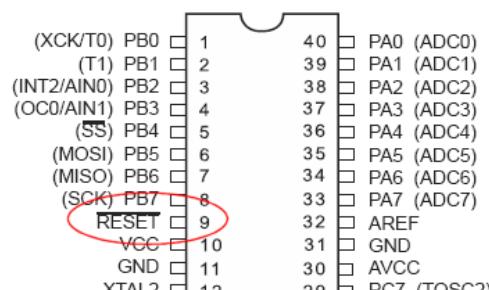
### 13.3 Činnost procesoru

Nyní si popíšeme činnost procesoru po resetu, vysvětlíme pojmy instrukční a strojové cykly a pipelining.

## Reset

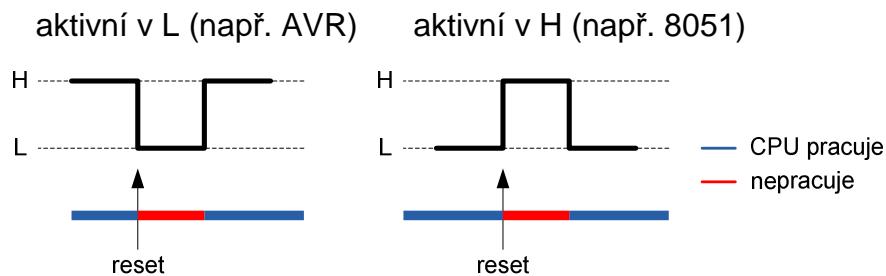
Reset je uvedení procesoru do výchozího stavu. U každého CPU probíhá odlišně. Typicky: PC = 0, SP = ??, GPR = 0, zákaz přerušení.

Procesor má zvláštní vstup **RESET**, viz příklad dle obr. 13.2. Reset je třeba vyvolat automaticky po připojení napájecího napětí nebo ručně (např. tlačítkem) např. při chybě v programu.



Obr. 13.2 Pouzdro mikrokontroléru ATmega16 resp. ATmega32

Polarita signálu je buď aktivní při log. 0 (AVR) nebo při log. 1 (Intel 8051), viz obr. 13.3.



Obr. 13.3 Polarita signálu RESET u různých typů mikrokontrolérů

## Instrukční a strojové cykly

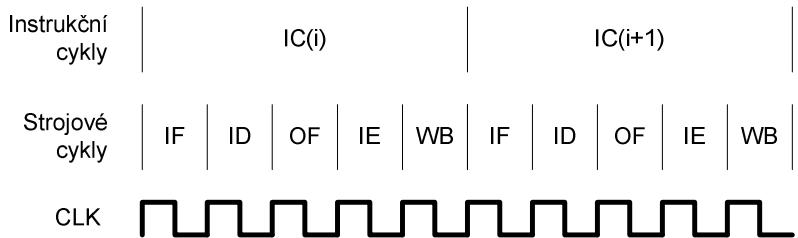
Procesor je sekvenční číslicový obvod a jako takový musí být řízen synchronizačním tzv. **hodinovým signálem**. Tento kmitočet určuje rychlosť provádění instrukcí tedy i výkon procesoru.

Výkonné procesory pracují s kmitočty v řádu stovek MHz až jednotek GHz. Mikrokontroléry jsou obvykle taktovány kmitočty v rozsahu jednotek až stovek MHz. Každý procesor má definováno rozmezí kmitočtů, pro které spolehlivě pracuje (u ATmega obvykle 0 až 16 MHz).

**Strojový cyklus** (machine cycle MC) odpovídá jedné periodě hodin. Procesor obvykle provede část instrukce. **Instrukční cyklus** se skládá z těchto typických strojových cyklů (viz obr. 13.4):

- čtení instrukce (Instruction Fetch),
- dekódování instrukce (Instruction Decode),
- čtení operandů (Operand Fetch),
- vykonání instrukce (Instruction Execution),
- zápis výsledku (Write Back).

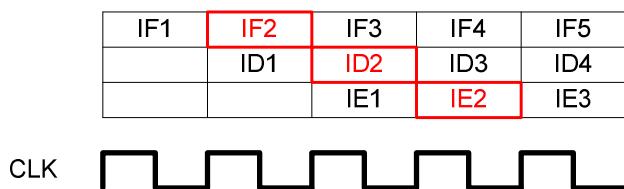
Různé procesory používají různý počet a různé typy strojových cyklů. Například Intel 8051 má 12 strojových cyklů na jeden instrukční cyklus. Atmel AVR používá pipelining, čím je dovoleno vykonat instrukci během jediného taktu hodin.



Obr. 13.4 Rozfázování instrukce při nepřekrývaném zpracování

### Pipelining

Pipelining je technika, která umožňuje současně (paralelní) provádění více instrukcí. Obvykle se jedná o to, že při souběžném provádění se každá instrukce nachází v jiné fázi zpracování. Tato technika se též označuje jako: překrývané, zřetězené nebo proudové zpracování instrukcí.



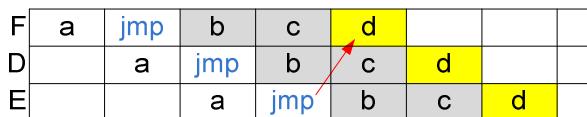
Obr. 13.5 Třístupňový pipelining

Počet stupňů pipeline je dán architekturou procesoru a rozsahem pracovních kmitočtů. Mikrokontroléry mají obvykle 2 až 5 stupňů. Procesory pro PC mají běžně desítky stupňů.

Používání této techniky omezují operace skoků nebo volání podprogramů. Pokud se v programu vyskytne instrukce skoku, je nutné pipeline začít plnit znova tedy zahodit předchozí obsah. Viz tento příklad programu a obr. 13.6:

```
instr. a
jmp nav
instr. b
instr. c
```

nav: instr. d



Obr. 13.6 Příklad „zahodení“ pipeline v případě skoku

### 13.4 Architektury CISC a RISC

Nyní provedeme porovnání architektur CISC a RISC.

#### CISC (Complex Instruction Set Computer)

CISC představuje vývojově starší variantu. Základním rysem byly komplexní instrukce, které svým formátem připomínaly zápis příkazů ve vyšším programovacím jazyku. Tedy smyslem bylo zjednodušit realizaci překladače, například:

```
add M1, M2, M3 // M1 = M2 + M3
```

Základní znaky:

- rozsáhlý instrukční soubor (stovky instrukcí),
- instrukce jsou komplexní operace:
  - složitý návrh čipu → vysoká cena,
  - zpracování instrukce → nutno více kroků řadiče → dlouhý instrukční cyklus,
  - kratší program (použije se méně instrukcí) → méně přístupů k paměti, úspora nákladů na mikroprocesorový systém (výhodné v době vzniku, tedy v 60. letech 20. stol.),
- příklady: x86 (do 80386), 8051.

### RISC (Reduced Instruction Set Computer)

V roce 1970 proběhl u IBM výzkum, který měl odpovědět na otázku, jak lze zvýšit výkon procesoru.

Zjistilo se, že bohatý instrukční soubor vede na složité optimalizace. Ani programátor ani překladač nejsou schopni využít všechny instrukce. Dále bylo zjištěno, že až 50% instrukcí nelze využít optimálně. V běžných programech má většina „výkonných instrukcí“ (aritmetika, cykly), které jsou obvodově nejsložitější, jen velmi malé zastoupení. Mezi nejčastější instrukce patří: přesuny, skoky, porovnání. Řešením tedy je:

- použít nízký počet jednoduchých instrukcí (desítky),
- zajistit kratší dobu provedení instrukce než u CISC,
- použít pipelining,
- mít k dispozici velký počet GPR.

Pro procesory s redukovanou instrukční sadou je tedy typický nízký počet tranzistorů na čipu a s tím spojená nižší spotřeba a nižší cena. Dnes je to nejrozšířenější varianta procesoru:

- 8bitové mikrokontroléry – Atmel AVR, Microchip PIC,
- 32bitové mikrokontroléry – ARM (Acorn Risc Machine, až 75% trhu: PALM, PDA, MP3, Smart Phones...), Motorola Freescale,
- Procesory pro PC – architektura x86 od Pentia dále, Power PC.

### CISC vs. RISC

Na závěr porovnejme nejnámější 8bitové mikrokontroléry 8051 a AVR.

8051 je architektury CISC a instrukční cyklus je rozložen do 12 strojových cyklů. Nepoužívá pipelining. Většina instrukcí zabírá 2 instrukční cykly. Instrukční soubor obsahuje i instrukce pro násobení a dělení, které zabírají 4 instrukční cykly.

AVR je architektury RISC, většina instrukcí trvá jeden instrukční cyklus. V některých případech je třeba doplnit chybějící operace (například realizovat dělení programově). Zjednodušeně lze říci, že výkon AVR odpovídá výkonu procesoru 80386 (CISC) při stejném hodinovém kmitočtu.

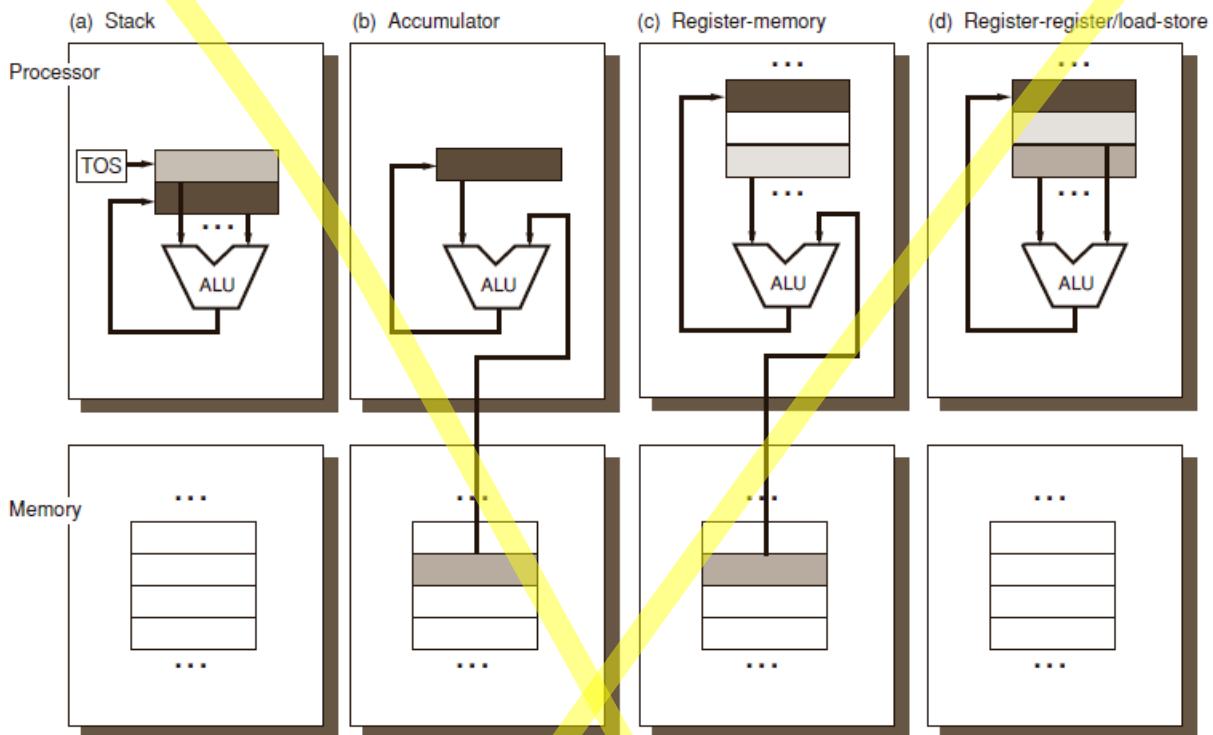
### 13.5 Orientace instrukčního souboru

V závislosti na zvolené architektuře CISC nebo RISC může být instrukční soubor orientován určitým způsobem. Orientací rozumíme způsob, jakým ALU načítá operandy operací nebo jakým směrem většinou probíhají přesuny dat.

Uvažujme výpočet výrazu:  $Z = X + Y$ . Podle orientace instrukčního souboru může být tato operace realizována několika rozličnými způsoby. Varianty jsou uvedeny níže.

Tab. 13.1 Orientace instrukčního souboru

zásobníkový	Aku	Reg-mem	mem-mem	L-S
<code>push X</code>	<code>mov Acc,X</code>	<code>mov r0,X</code>	<code>add Z,X,Y</code>	<code>lds r0,X</code>
<code>push Y</code>	<code>add Acc,Y</code>	<code>add r3,r0,Y</code>		<code>lds r1,Y</code>
<code>add</code>	<code>mov Z,Acc</code>	<code>mov Z,r3</code>		<code>add r0,r1</code>
<code>pop Z</code>				<code>sts Z,r0</code>



Obr. 13.7 Vysvětlení variant operandů instrukcí

### Zásobníkové CPU

- princip: vstupní a výstupní operandy se předávají přes zásobník,
- výhody: krátké instrukce, jednoduchý HW, efektivní adresace operandů,
- nevýhoda: nelze zajistit libovolný přístup k datům,
- příklady: x87 (koprocesor pro řadu x86), .NET a Java Virtual Machines.

### Akumulátorové CPU

- princip: jeden registr (tzv. akumulátor) má výsadní postavení, protože většina operací probíhá přes něj (je to vždy jeden z operandů instrukce, výsledek je také uložen v akumulátoru),
- výhody: krátké instrukce, jednoduchý HW, rychlé přepínání kontextu,
- nevýhody: častá komunikace s pamětí,
- příklady: 8080, 8051, 68HC1x, x86 počínaje 80386 (k dispozici je více akumulátorů).

### Reg-Mem/Mem-mem

- princip: základní operace jsou přesuny dat mezi registrtem a paměťovým místem nebo mezi dvěma paměťovými místy, typicky CISC CPU (x86, Motorola 68k),
- výhoda: kompaktní kód,
- nevýhoda: instrukce jsou variabilní délky.

## Reg-Reg/Load-Store

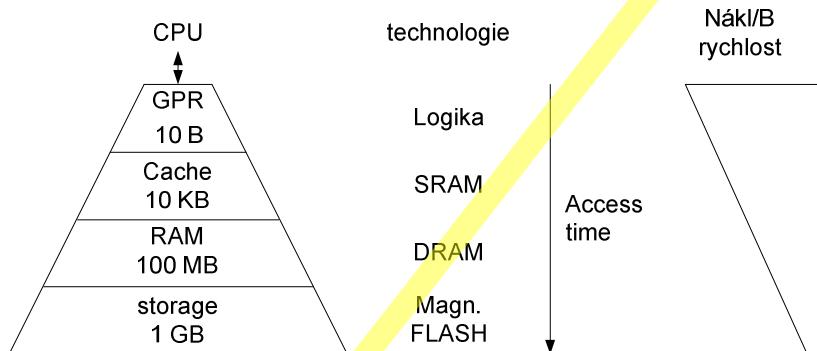
- princip: základní operace jsou přesuny dat mezi dvěma registry nebo mezi paměťovým místem a registrem, používají všechny RISC,
- **využívají** všechny moderní CPU (některé „virtuálně“, například x86 používá na úrovni mikroarchitektury),
- **výhody**: krátké, jednoduché instrukce, GPR = nejrychlejší paměť,
- **nevýhody**: delší kód, delší přepnutí kontextu, do GPR se nevejdou delší datové struktury (struktury, pole).

## 13.6 Paměťový substituční systém

Ideálem je **velká**, rychlá paměť. Taková paměť je technicky nerealizovatelná, uvedené chování však lze napodobit hierarchií.

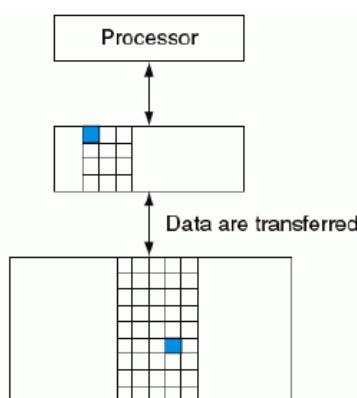
### Princip lokality:

- pravidlo 90/10: „90% času stráví program během 10% instrukcí“,
- v čase – požadujeme položku, v budoucnu bude požadována znova,
- v prostoru – požadujeme položku, položky okolo budou požadovány také.



Obr. 13.8 Hierarchické uspořádání paměti

Problém lze řešit pomocí cache (rychlá vyrovnávací paměť) a virtuální paměti (paměť zvýšená nad fyzické možnosti například použitím pevného disku).



Obr. 13.9 Dvouúrovňová hierarchie paměti

**Blok** označuje minimální množství informace přítomné/nepřítomné ve dvouúrovňové hierarchii. Velikost bloku může odpovídat velikosti adresovatelné jednotky, často se však jedná o násobky (jednotky až desítky slov).

Podle toho, zda se data v blížším bloku paměti nachází (hit) nebo nenachází (miss), definujeme pak úspěšnost resp. neúspěšnost „zásahu“:

$$hit\ rate = \frac{\text{uspesne pristupy}}{\text{vsechny pristupy}} \quad miss\ rate = \frac{\text{neuspesne pristupy}}{\text{vsechny pristupy}},$$

Pochopitelně platí:  $hit\ rate + miss\ rate = 1$ .

**Miss penalty** označuje čas nutný pro kopii bloku do dané úrovni.

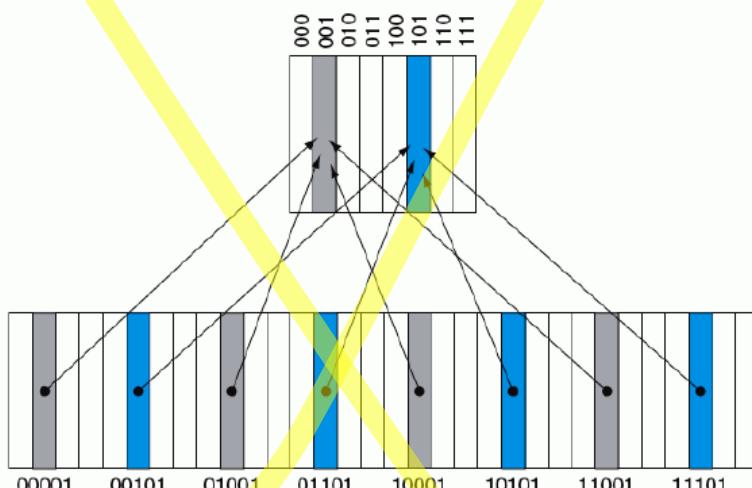
### Přímo mapovaná cache (direct mapped cache)

Principem tohoto typu architektury cache je, že každé místo v paměti odpovídá jednomu místu v cache. Jelikož není kapacita cache neomezená, platí pro přepočet adresy v paměti na adresu v cache vztah:

$$\text{addr}_{\text{CACHE}} = \text{addr}_{\text{MEMORY}} \% \text{size}_{\text{CACHE}} \quad (13-1)$$

kde:

- $\text{addr}_{\text{CACHE}}$  – adresa údaje v cache,
- $\text{addr}_{\text{MEMORY}}$  – adresa údaje v paměti,
- $\text{size}_{\text{CACHE}}$  – velikost cache.



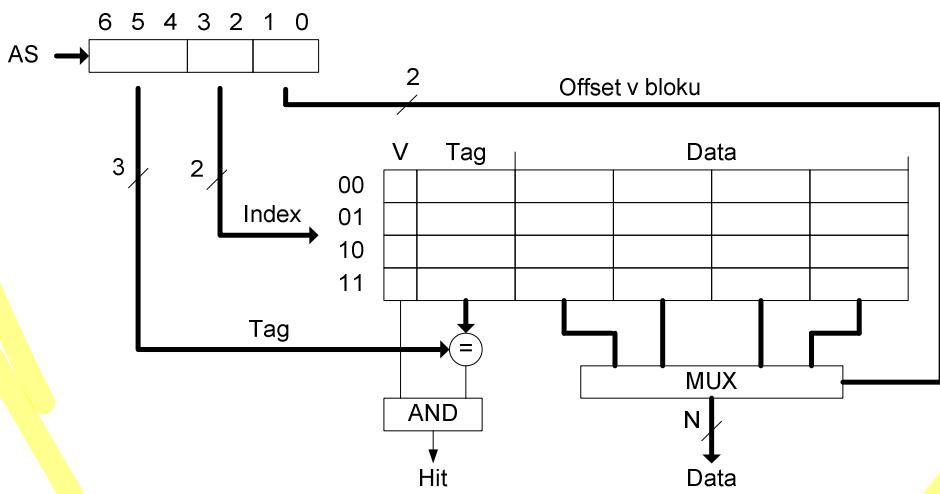
Obr. 13.10 Přepočet adresy paměti na adresu v cache

Na obr. 13.11 je uveden příklad konstrukce cache, která má 7bitovou adresovou sběrnici, blok 4 slov, 4 záznamy:

- **V** (valid bit) – indikuje, že na dané adrese cache je platný záznam (například po resetu je prázdná cache, takže  $V = 0$ ),
- **Tag** – identifikace, zda blok v cache odpovídá požadovanému slovu,
- **Offset** – odpovídá údaji z bloku, ke kterému chceme přistoupit,

Dle stavu adresové sběrnice se porovnají horní 3 byty, zda souhlasí štítek. Pokud ano, vyšle se k hradlu AND log. 1. Dále se testuje příznak platnosti záznamu, pokud je platný, vyšle se k hradlu AND log. 1. V případě platnosti obou signálů je údaj k dispozici v cache a platí Hit = 1.

Index odpovídá adrese jednoho ze čtyř záznamů (00, 01, 10, 11), žádaný jeden ze 4 bajtů je pak určen pomocí offsetu a vystupuje přes koncový multiplexer jako signál označený Data.



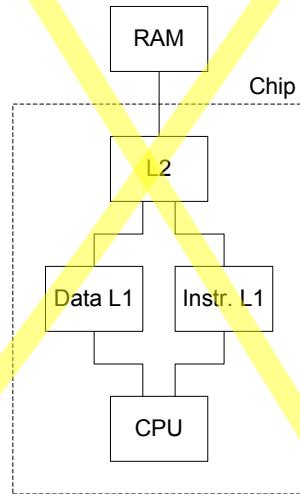
Obr. 13.11 Příklad konstrukce cache

### Možnosti provádění zápisu

- write-through: zápis probíhá současně do cache + paměti,
- write-back: zápis do paměti se provede pouze tehdy, je-li přepisován blok v cache; rychlejší, ale komplikovanější implementace

### Moderní

Moderní CPU pro PC používají například cache dvou úrovní viz obr. 13.13. L2 cache je napojena přímo na paměť pomocí L1 cache pro data a instrukce. Z hlediska procesoru se systém jeví tak, jako by měl Harvardskou architekturu. Pro programátora má však architekturu von Neumann.



Obr. 13.12 Cache L1 a L2

## 13.7 Zvyšování výkonu procesoru

Moderní procesory využívají různé techniky pro zvýšení výpočetního výkonu. Hlavně pipelining a parallelismus (na úrovni dat, na úrovni instrukcí, na úrovni CPU – vícejádrové procesory).

## Paralelizmus na úrovni instrukcí

Jedná se o zpracování více instrukcí během instrukčního cyklu. Příklad:

```

1. e = a + b
2. f = c + d           // 1+2 lze společně
3. g = e * f           // závisí na 1 a 2

```

Pro zajištění této techniky je nutná duplikace vnitřních součástí CPU (např. ALU), způsoby:

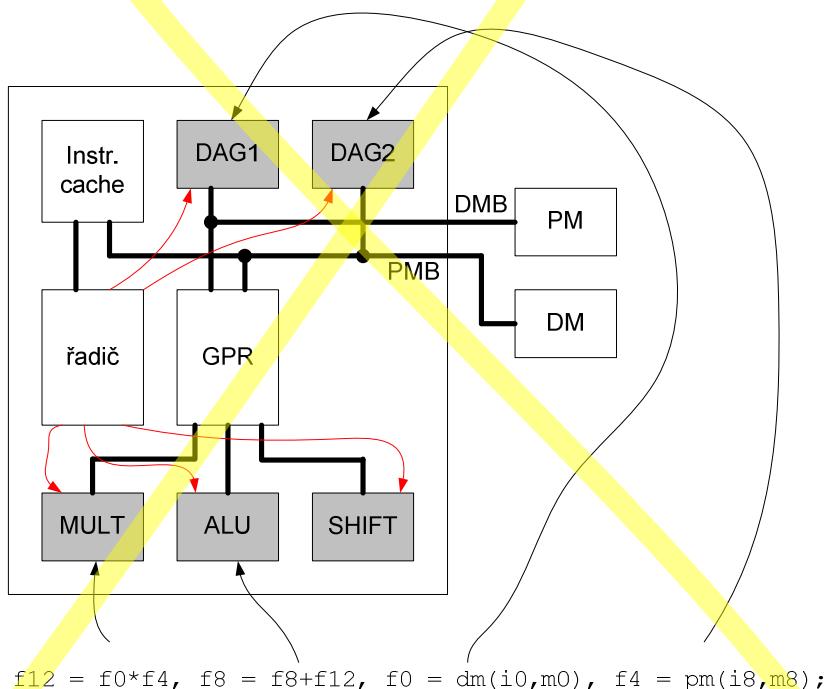
- staticky – VLIW CPU,
- dynamicky – superskalární CPU.

### VLIW CPU (Very Long Instruction Word)

Tuto techniku používá většina DSP nebo například Intel IA-64 (Itanium II).

Sdružování instrukcí musí na úrovni vyššího programovacího jazyka provádět překladač, na úrovni assembleru pak programátor.

Příkladem může být DSP SHARC od firmy Analog Device, který obsahuje 3 výpočetní jednotky: MUL (násobička), ALU a SHIFT (posuvy) a dále zdvojené jednotky DAG pro výpočet a generování adres. Viz obr. 13.13.



Obr. 13.13 DSP SHARC od Analog Devices

Ekvivaletní kód pro AVR:

```

ld    r0, X+
ld    r4, Y+
mul  r0, r4
add  r8, r12

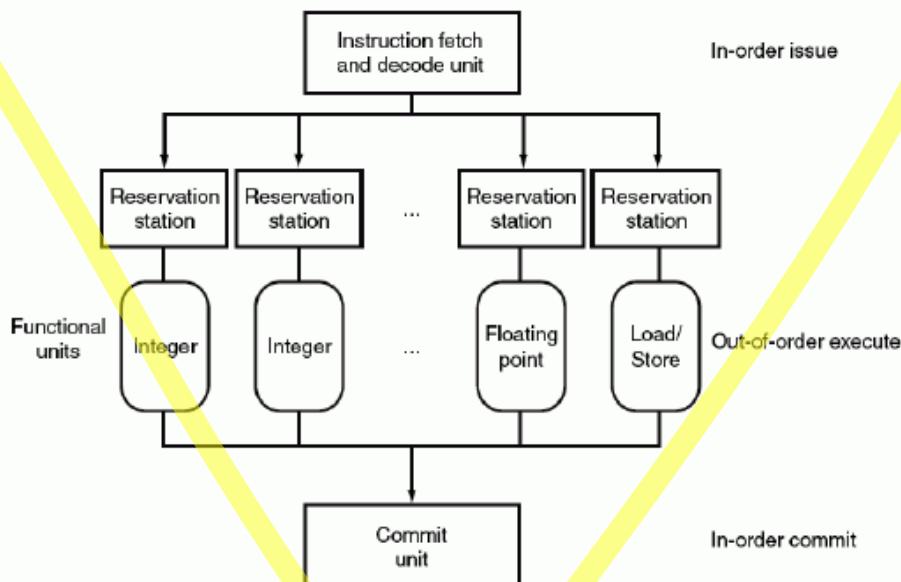
```

## Superskalární CPU

Při použití této techniky je běžný instrukční tok rozkládán dynamicky přímo uvnitř CPU. Rozklad tedy neovlivňuje programátor. Typické uspořádání CPU a tok instrukcí naznačuje obr. 13.14.

Jednotka IF+DU (Instruction fetch and decode unit) zasílá instrukce jednotlivým podřízeným jednotkám RS (Reservation station). Instrukce jsou vykonávány mimo pořadí.

Jednotka CU (Commit unit) provádí úpravu výsledků do původního pořadí, zapisuje výsledky do GPR, popř. RAM.



Obr. 13.14 Princip superskalární architektury

S touto technikou je spojeno **dynamické přejmenování registrů**. Uvažujme tento program v jazyce C:

```
vysl1 = prom1 + prom2;  
vysl2 = prom3 + prom4;
```

Odpovídající program v assembleru:

```
mov r1, prom1  
add r1, prom2  
mov vysl1, r1  
mov r1, prom3  
mul r1, prom4  
mov vysl2, r1
```

Superskalární CPU vykonává instrukce současně:

```
mov r1, prom1, r2, prom3  
add r1, prom2, mul r2, prom4  
mov vysl1, r1, mov vysl2, r2
```

Další variantou je **spekulativní vykonávání instrukcí**. CPU nebo překladač předpovídá běh programu. Pokud je odhad špatný na úrovni CPU je nutno zahodit obsah pipeline. Špatný odhad na úrovni překladače se řeší kontrolním kódem a případnou korekční rutinou.

## Paralelizmus na úrovni dat

Rozlišujeme několik variant dle Flynnovy klasifikace (Single – Multiple, Instruction – Data):

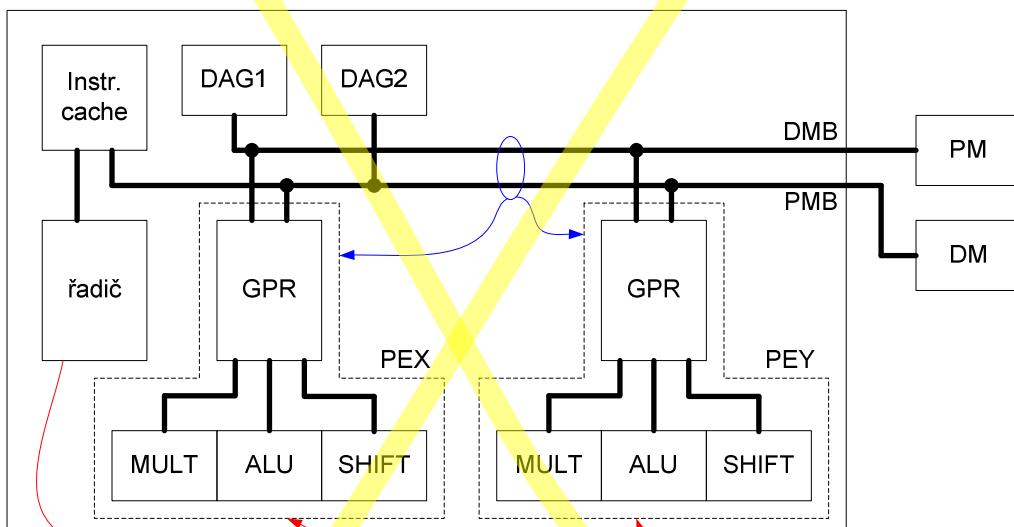
- SISD – používá například AVR,
- SIMD – viz níže,
- MISD – speciální redundantní řídicí systémy (železniční zabezpečovací technika, řídicí systémy letadel),
- MIMD – distribuované výpočetní systémy, některé DSP (ADI TigerSHARC).

### SIMD (Single Instruction Multiple Data)

Jedná se o to, že více dat zpracováno stejnou instrukcí:

$$\text{adresaPEY} = \text{adresaPEX} + 1$$

Například: násobení vektoru skalárem – PEX sudé členy, PEY liché.  
Výpočetní výkon se zvýší dvojnásobně.



Obr. 13.15 SIMD SHARC

Jiným příkladem je SSE (Streaming SIMD Extensions). Jedná se o doplněk instrukční sady procesorů x86 od P III (70 nových instrukcí). Jako ekvivalent lze uvést technologii 3Dnow! od AMD. K dispozici je 8 registrů šíře 128 bitů registry pro uložení: 4 čísel typu float, 2 čísel typu double, ... 16 čísel typu char. Jsou implementovány vektorové operace.

Jako příklad uvádíme instrukci ADDPD (Packed Single-Precision Floating-Point Add), která odpovídá 4násobnému float součtu:

$$\begin{aligned}
 \text{DEST[31-0]} &= \text{DEST[31-0]} + \text{SRC[31-0]}; \\
 \text{DEST[63-32]} &= \text{DEST[63-32]} + \text{SRC[63-32]}; \\
 \text{DEST[95-64]} &= \text{DEST[95-64]} + \text{SRC[95-64]}; \\
 \text{DEST[127-96]} &= \text{DEST[127-96]} + \text{SRC[127-96]};
 \end{aligned}$$