

ASSEMBLER

*a
ZX Spectrum*

1. díl

proxima
Veřejná Obchodní Společnost

ASSEMBLER
ASSEMBLER

A ZX SPECTRUM
A ZX SPECTRUM

STRUČNĚ O ASSEMBLERU

Tato kapitola je určena pro ty, kteří s assemblerem teprve začínají. Zajímavé informace tu však najdou všichni a k tabulkám v této kapitole se budete vracet velmi často.

Nejprve úvod pro úplné laiky. Assembler (strojový kód, machine code) je jazyk, který je počítači vlastní - je doslova zadrátován v procesoru. V assembleru jsou naprogramovány všechny ostatní programy (BASIC je vlastně v assembleru napsaný program, který umožňuje vkládat a vykonávat příkazy - pro každý příkaz obsahuje BASIC podprogram v assembleru, který vykonává to, co jednotlivé příkazy BASICu znamenají).

Každý vyšší programovací jazyk je prostředek, jak napsat požadovaný program bez použití assembleru. Vyšší programovací jazyky vznikly pro usnadnění nelehké práce programátorů - program ve vyšším jazyku je kratší než v assembleru (zdrojový text, nikoliv přeložený kód!), ve vyšších jazycích se nedělají tak snadno chyby a také jejich následky nejsou tak fatální. Program v assembleru nelze snadno přenést na jiný typ počítače. Program ve vyšším jazyku lze provádět dvěma způsoby:

INTERPRET - každý příkaz je prováděn ihned po přečtení.

- vhodné pro ladění
- provádění je pomalejší než v druhém případě

KOMPILÁTOR - program je nejprve přeložen do strojového kódu a pak vykonáván.

- výhodné při opakovaném použití
- provádění je obvykle výrazně rychlejší než u interpretování

Nyní proč používat assembler - zatím vše hovoří v jeho neprospěch. Když chcete využít možnosti počítače naplno, chcete napsat rychlý a krátký program, zjistíte, že to buď nelze vůbec ve vyšším jazyku provést, nebo jen za cenu neúměrných komplikací. Na Spectru k tomu přistupuje také značné omezení velikosti paměti (kompilátor nebo interpret zabírají v paměti místo, které by mohlo být využito programem).

Počítač (Z80) rozumí assembleru ve formě posloupnosti čísel (nul a jedniček) - této formě se obvykle říká **strojový kód**. Pro člověka je mnohem příznivější forma symbolického zápisu instrukcí, které se obvykle říká **assembler**. Slovo **assembler** se také používá pro označení programu pro převod programů ze symbolické formy do formy číselné. Pro další práci můžeme přesné významy uvedených slov nerozlišovat - pochopíte je vždy z kontextu.

symbolická forma	číselná forma	význam
ld a, b	01111000	B → A

Uvedená instrukce přenáší obsah z registru B do registru A. Význam symbolického zápisu (mnemoniky) je následující:

- ld - mnemonika, typ instrukce (říká CO se má dělat)
- a, b - operandy (říkají s ČÍM má být akce provedena)

Mnemonika se v instrukci vyskytuje vždy, operandy se mohou vyskytovat buď dva, jeden nebo se nevyskytují vůbec (v tomto případě plynou přímo z mnemoniky).

REGISTRY Z80 - Mikroprocesor Z80 obsahuje tyto 8-bitové registry:

- a - akumulátor (strádač), nejdůležitější registr
- f - flag registr (stavový registr), zde jsou informace o předchozích operacích
- b, c, d, e, h, l - ostatní obvyčejné 8 bitové registry
- r - refresh registr (oživovací registr), slouží k oběrstvování paměti
- i - interupt registr (registr přerušení), viz dále.

Protože do 8 bitů lze zapsat pouze číslo v rozmezí **0-255**, obsahuje Z80 také registry 16-bitové a umožňuje používat dvojice 8-bitových registrů jako 16-bitové registry. Do takových registrů lze zapsat číslo v rozmezí **0-65535**. K dispozici máte tyto 16-bitové registry:

- PC - čítač instrukcí (program counter), ukazuje vždy na prováděnou instrukci
- SP - ukazatel na zásobník (stack pointer), na zásobníku jsou návratové adresy
- ix, iy - indexové registry

Jako 16-bitové registry lze používat tyto kombinace registrů:

- af, bc, de, hl - nejdůležitější z nich je registr hl

Při používání registrů si uvědomte, že pokud pracujete s registrovými páry, mění se i jednotlivé registry (tedy například s registry h a l můžete pracovat buď jako se dvěma 8-bitovými registry nebo jako s jedním 16-bitovým registrem hl). Všechny základní registry (a, b, c, d, e, f, h, l) jsou v procesoru dvakrát (tzv. záložní registry) a můžete volit, kterou skupinu chcete používat - přepínat lze zvlášť registry a, f a registry b, c, d, e, h, l.

16-bitové indexové registry ix, iy lze používat i rozděleny na 8-bitové části - registr ix lze rozdělit na hx a lx, podobně iy na hy a ly.

Poslední informace o registrech Vám řekne, jak je číslo uloženo v registrovém páru. Zapišete-li do registru hl hodnotu **12345**, bude v registru h hodnota **48** (neboli celá část podílu 12345/256) a v registru l pak **57** (zbytek po dělení 12345/256). Opačně, když naplníte registry h a l nějakými čísly, pak v dvoiregistru hl bude hodnota:

$$hl = 256 * h + l, \text{ kde } h \text{ je hodnota z registru h, a } l \text{ hodnota z registru l.}$$

Zcela stejně se chovají registrové páry bc, de a af, rovněž tak ix a iy pokud je rozdělíte na hx, lx a hy, ly.

Hlavní sada registrů		Alternativní sada registrů		
Akumulátor A	Flagy F	Akumulátor A'	Flagy F'	- registry AF
B	C	B'	C'	} registry obvyčejného použití
D	E	D'	E'	
H	L	H'	L'	

Vektor přerušeni I	Oživování paměti R	
Indexový registr IX (hx, lx)		} registry speciálního použití
Indexový registr IY (hy, ly)		
Ukazatel na zásobník SP		
Čítač instrukcí (Programový ukazatel) PC		

Z tohoto stručného popisu nemůžete pochopit vše, to také není cílem této kapitoly ani této knihy, zde byste se měli dozvědět, jak naprogramovat to nebo ono bez toho, abyste museli ihned chápat, jak to přesně pracuje. Nejprve budete používat naše příklady a později je budete stále více modifikovat a přizpůsobovat k obrazu svému.

FLAGY (příznaky, stavové indikátory). Instrukce Z80 tvoří dvě skupiny:

- a) instrukce **řízení běhu programu** (skoky, volání a návrat z podprogramu)
- b) **pracovní** instrukce (všechny ostatní)

Každá pracovní instrukce mění vnitřní stav procesoru, který může ovlivnit provedení následující instrukce. Pod pojmem vnitřní stav procesoru si můžete představit informaci o tom, jak dopadla poslední operace - jestli u sčítání došlo k přetečení (výsledek není v povoleném rozsahu, jestli je výsledek nulový, kladný nebo záporný. . .

Instrukce pro **řízení běhu programu** mohou testovat platnost zmíněných podmínek a podle toho provést odskok, volání nebo návrat z podprogramu či případně pokračovat na další instrukci.

Z80 má pro uložení vnitřního stavu procesoru vyhrazen jeden osmibitový registr - F. Může si tedy pamatovat 8 nezávislých informací typu 0/1. Ve skutečnosti je použito 6 z 8 možných a při programování budete používat jen 4. Přebývající dva bity je také možno testovat ale nikoliv přímo. Pro praktické programování Vám budou stačit v 99% případů pouze dva příznaky. Následuje seznam příznaků a vysvětlení jejich významu.

SIGN flag (znaménko) - kopíruje do sebe hodnotu nejvyššího bitu výsledku (M.S.B.). Znamená to že čísla větší než 128 u osmibitových a větší než 32768 u šestnáctibitových registrů jsou chápána jako záporná. Flag může nabývat hodnotu 0 (kladné číslo - značení **P** jako **Plus**) nebo 1 (záporné číslo - značení **M** jako **Minus**). Ke způsobu uložení čísel a k možným pohledům na ně se ještě v této kapitole vrátíme.

ZERO flag (příznak nuly) - nabývá hodnoty 1 (výsledek operace je nula - značení **Z** jako **Zero**) nebo 0 (výsledek operace není nula - značení **NZ** jako **Non Zero**). Příznak nuly se obvykle vztahuje na obsah registru **A**, může se však vztahovat i na jiné registry nebo také na jednotlivé bity registrů. Tento příznak je nejdůležitější ze všech.

HALF CARRY flag (příznak polovičního přetečení) - tento příznak nelze přímo testovat a je určen spíše pro procesor než pro programátora. Na tento příznak můžete úplně klidně zapomenout, nebudeme jej potřebovat.

PARITY & OVERFLOW (P/V) flag (příznak parity a přetečení) - logické operace sem ukládají paritu výsledku (počítá stejné bity ve výsledku - sudý počet **PE**, lichý počet **PO**), aritmetické operace pak přetečení, čísla jsou chápána jako čísla se znaménkem - přetečení nastane pokud je součet dvou kladných čísel číslo záporné nebo pokud je součet dvou záporných čísel číslo kladné - nepleťte s **CARRY**. Příznak nabývá hodnotu 0 (značení **PO** jako **Parity Odd**) nebo 1 (značení **PE** jako **Parity Even**). Tento příznak je možné použít pro zjištění stavu přerušení.

N flag (příznak odečítání) - podobně jako **Half Carry** je určen pro procesor, nelze jej přímo testovat. Obsahuje 1 pokud byla předchozí operace odečítání.

CARRY (příznak přetečení) - druhý důležitý příznak. Nabývá hodnotu 0 (**NC** od **Non Carry** - nedošlo k přetečení) a 1 (**C** od **Carry** - došlo k přetečení). K přetečení dojde například tehdy, když budete sčítat čísla 200 a 100 v 8-bitovém registru. Výsledek sčítání by měl být 300 ale to je číslo, které nelze zapsat do 8 bitů, výsledek bude 44 a bude nastaveno **C**. Obdobně se postupuje u odečítání a u operací na dvojeregistrech.

Obsah příznaku přetečení lze nejen nastavit a testovat ale také využít přímo při aritmetických a logických operacích - hodnotu **Carry** flagu lze přičítat a odečítat (výhodně při počítání s čísly většího rozsahu než 8 (16) bitů. Příznak se také používá při porovnávání čísel podle velikosti.

Příznak **CARRY** také používají instrukce rotací a posuvů, bližší objasnění naleznete v části věnované těmto instrukcím.

Interrupt enable FLIP-FLOP (IFF) - není uložen v registru **F**, obsahuje stav přerušení (zakázané/povolené). Jeho hodnotu lze instrukcemi **Id a,i** a **Id a,r** kopírovat do **P/V** flagu.

Následující tabulka ukazuje vliv instrukcí na příznaky. Pokud nějaká instrukce není v tabulce uvedena, znamená to, že ponechává příznaky beze změny.

Mnemonicika	Sign	Zero	P/V	Carry	Poznámka
add a,R1 adc a,R1 sub R1 sbc a,R1 cp R1 neg	! ! ! ! ! !	! ! ! ! ! !	U U U U U U	! ! ! ! ! !	Aritmetické instrukce
and R1 or R1 xor R1	! ! !	! ! !	P P P	! ! !	Logické instrukce
inc R1 dec R1	! !	! !	U U	- -	Inkrementace Dekrementace
add hL,R2 adc hL,R2 sbc hL,R2	- ! !	- ! !	- U U	! ! !	16-ti bitová aritmetika
rla rlca rra rrca	- - - -	- - - -	- - - -	! ! ! !	Rotace akumulátoru
rl R1 rlc R1 rr R1 rrc R1	! ! ! !	! ! ! !	P P P P	! ! ! !	Rotace
sll R1 sra R1 sllia R1 srl R1	! ! ! !	! ! ! !	P P P P	! ! ! !	Posuny
rld rrd	! !	! !	P P	- -	Rotace skupiny bitů
daa	!	!	P	!	Desítková korekce
cpl scf ccf	- - -	- - -	- - -	- 1 !	Binární doplněk Nastavení Carry Převrácení Carry
in R1,(C)	!	!	P	-	Vstup z portu
ini ind outi outd	? ? ? ?	b=0 b=0 b=0 b=0	? ? ? ?	- - - -	Blokový vstup a výstup Z=0 když B<0 Z=1 když B=0
inir indr otir otdr	? ? ? ?	1 1 1 1	? ? ? ?	- - - -	Blokový vstup a výstup s opakováním

Mnemonic	Sign	Zero	P/V	Carry	Poznámka
ldi ldd	? ?	? ?	bc <> 0 bc <> 0	- -	Blokový přenos
ldir lddr	? ?	? ?	0 0	- -	Blokový přenos s opakováním
cp i cp d	? ?	a = (h l) a = (h l)	bc <> 0 bc <> 0	- -	Blokové hledání
cp i r cp d r	? ?	a = (h l) a = (h l)	bc <> 1 bc <> 1	- -	Blokové hledání s opakováním
ld a, i ld a, r	! !	! !	IFF IFF	- -	Do příznaku P/V jde stav přerušení
bit N, R1	? ?	! !	? ?	- -	Stav daného bitu

Vysvětlivky:

- ! - příznak je nastaven podle výsledku operace
- ? - příznak není definován
- - příznak je nezměněn
- 0 - příznak je vynulován
- 1 - příznak je nastaven
- P - příznak je nastaven vzhledem k paritě výsledku
- O - příznak je nastaven vzhledem k přetečení výsledku
- R1 - osmibitový operand (registr nebo adresa v paměti)
- R2 - šestnáctibitový registr

U některých instrukcí si své závěry ohledně chování příznaků raději ještě ověřte tím, že pomocí monitoru (monitor od PROMETHEA, DEVAST, VAST, PIKOMON,...) vybranou instrukci protrasujete. Nejčastější chyby při použití příznaků vznikají, když instrukce daný příznak ponechává nezměněn a Vy si myslíte, že jej nastavuje podle výsledku operace (například Carry u instrukcí **inc R1** a **dec R1**), další poměrně častou chybou je, že zapomenete u instrukce **sbc hl, R2** vynulovat Carry).

Číslo v assembleru. V této části se dozvíte o číselných soustavách, které assembler používá, a něco o aritmetice procesoru Z80.

Assembler používá celkem tři číselné soustavy - dvojkovou (binární), desítkovou (decimální, dekadickou) a šestnáctkovou (hexadecimální). Číslo, podle kterého je číselná soustava pojmenována, je základ dané číselné soustavy. Co je to základ číselné soustavy si ukážeme na příkladu desítkové soustavy:

číslo **34327** lze zapsat jako **3*10000 + 4*1000 + 3*100 + 2*10 + 7*1**,

vidíte, že jednotlivá čísla (řády **10000, 1000, 100, 10, 1**) jsou mocniny čísla **10** - a to je tedy základ desítkové soustavy. Také si můžete všimnout, že desítka je počet číslic, které desítková soustava používá - to platí všeobecně.

Nyní dvojková soustava. Základem je číslo **2** a veškeré číslice, které tato soustava používá jsou **0** a **1**. Jednotlivé řády jsou tedy mocniny čísla **2** - **1, 2, 4, 8, 16, 32, 64, 128...** Naše číslo **34327** ve dvojkové číselné soustavě bude vypadat takto: **1000011000010111**. O tom, že jde o správný výsledek se snadno přesvědčíte tím, že spočítáte hodnotu čísla.

Zápis ve dvojkové soustavě má tento význam: $32768+1024+512+18+4+2+1 = 34327$. Do assembleru se binární čísla zapisují se znakem % před číslem:

```
ld    a, %111001010
```

Číslo do dvojkové soustavy můžete převést tak, že postupně odečítáte mocniny **2** tak dlouho, až dostanete nulu (začínáte od nejvyšších řádů). Potom za každý řád, který byl odečten napíšete **1** a za každý nepoužitý řád **0**. Převod do desítkové soustavy už byl popsán. Převádění z jedné soustavy do druhé však nebudete příliš často potřebovat - na převádění můžete využívat monitor **PROMETHEUS** (v něm můžete používat všechny soustavy bez potřeby převádět z jedné do druhé - použijete takovou soustavu, kterou zrovna potřebujete). Binární soustavu využijete nejčastěji u logických instrukcí a při práci s grafikou.

Šestnáctková soustava, podle názvu, má základ **16** a tedy používá nejen číslice ale také další znaky - písmena od **A (=10)**, **B (=11)**, **C (=12)**, **D (=13)**, **E (=14)** a **F (=15)**. Tato číselná soustava se používá proto, že je velmi výhodná při dělení dvoubytového čísla na dvě jednobytová čísla - stačí totiž rozdělit podle číslic. Naše číslo **34327** se v hexadecimální soustavě píše **#8617** (znak # se používá pro rozlišení): $8*4096+6*256+1*16+7 = 34327$. Zapišeme-li tedy číslo **#8617** do registru **hl** bude v registru **h** hodnota **#86** a v registru **l** pak hodnota **#17**. Pro převádění čísel do hexadecimální soustavy se obvykle používá tabulka.

Občas je potřeba zapsat do registru také kód nějakého znaku - samozřejmě že je možné se podívat do tabulky kódů - je to však zbytečná práce a proto assembler umožňuje vkládat potřebné znaky přímo:

```
ld    a, "A"
```

Tak dostanete do registru **a** kód znaku **A**, tedy číslo **65**. Chcete-li vložit znak " (úvozovka), musíte se podívat do návodu k Vámi používanému assembleru, někdy je nutno úvozovku vložit dvakrát (podobně jako v BASICu):

```
ld    a, "''"
```

Při psaní čísel v assembleru je také možno používat aritmetické výrazy. Můžete používat operátory **+**, **-**, *****, **/** a **%** (operace modulo neboli zbytek po dělení). Výrazy jsou obvykle vyhodnocovány zleva doprava a nebere se ohled na prioritu operátorů - je to jednodušší a potřebná část assembleru je kratší.

Aritmetické operace jsou obvykle prováděny **modulo 65536** - bude-li výsledek překračovat rozsah **0-65535**, bude výsledkem zbytek po dělení číslem **65536**. Budete-li mít nějaké nejasnosti o tom, co bude výsledkem operace, raději si to ověřte dříve než program budete spouštět - mohlo by to být zdrojem nepochopitelných chyb (zvláště je-li program odladěn na jiném assembleru - překladači).

Důležité pro programování v assembleru je možnost získat horní a spodní byte libovolné adresy, lze to provést například takto (některé assembly mají zvláštní funkce):

```
ld    l, ADRESA?256 ; dolní byte adresy
ld    h, ADRESA/256 ; horní byte adresy
```

Na závěr ještě několik příkladů, že stejné číslo lze zapsat mnoha různými způsoby:

```
ld    a, 65
ld    a, #41
ld    a, %10000001
ld    a, "A"
ld    a, 180/3+70/2 ; POZOR - výpočet je prováděn zleva doprava!
ld    a, 25+#28
```


Instrukce assembleru lze podle významu rozdělit do několika skupin a ty si popíšeme:

První skupinu tvoří instrukce přesunu 8 bitových hodnot (8-Bit Load Group). Symbolický zápis u těchto instrukcí je **ld dest, src**, kde **ld** je zkratka anglického slova **LOAD** (naložit), **dest** je místo uložení a **src** je místo, odkud je hodnota čtena. Všechny možné instrukce **ld** jsou popsány v dalším textu, u každého typu instrukce je popsán vliv instrukce na flagy (stavové indikátory, jejich smysl se dozvíte později).

Instrukce **LOAD** může přenášet hodnoty mezi všemi základními registry, možné instrukce jsou vypsány v následující tabulce:

Přesun obsahu z jednoho osmibitového registru do druhého						4 T-cykly
ld b, b	ld c, b	ld d, b	ld e, b	ld h, b	ld l, b	ld a, b
ld b, c	ld c, c	ld d, c	ld e, c	ld h, c	ld l, c	ld a, c
ld b, d	ld c, d	ld d, d	ld e, d	ld h, d	ld l, d	ld a, d
ld b, e	ld c, e	ld d, e	ld e, e	ld h, e	ld l, e	ld a, e
ld b, h	ld c, h	ld d, h	ld e, h	ld h, h	ld l, h	ld a, h
ld b, l	ld c, l	ld d, l	ld e, l	ld h, l	ld l, l	ld a, l
ld b, a	ld c, a	ld d, a	ld e, a	ld h, a	ld l, a	ld a, a

Instrukce na diagonále (zdrojový i cílový registr je stejný) nejsou užitečné vůbec k ničemu - neprovádějí žádnou činnost, byly vytvořeny proto, že k tomu vedly hardwarové důvody (můžete je používat pro zpestření místo instrukce **NOP**).

- - -

Čas od času na programátory přijde potřeba nějaký registr naplnit číslem:

Naplnění osmibitového registru číslem v rozsahu 0-255						7 T-cyklů
ld b, N	ld c, N	ld d, N	ld e, N	ld h, N	ld l, N	ld a, N

Na místě **N** můžete v assembleru psát libovolný výraz, jehož hodnota je v rozsahu **0..255** nebo také v rozsahu **-128..127** pokud se jedná o číslo se znaménkem.

- - -

V procesoru **Z80** se kromě obyčejných registrů nalézají také registry pro speciální použití - **I** a **R** registry. Práci s nimi obstarávají tyto instrukce:

Přesuny mezi registrem A a registry I a R 9 T-cyklů			
ld a, i	ld a, r	ld i, a	ld r, a

U těchto instrukcí se zastavíme podrobněji. Jsou to jediné instrukce pracující se zvláštními registry **I** a **R**. Z toho, celkem zřejmé, plyne, že pokud chcete registry **I** nebo **R** naplnit, musíte nejprve naplnit registr **A** a potom jeho obsah přenést do **I** nebo **R**.

Pokud naopak přenášíte hodnotu z **I** nebo **R** do **A**, dejte si pozor na to, že to jsou dvě jediné **ld** instrukce, které **nastavují flagy!** Jak již bylo zmíněno, můžete těmito instrukcemi také zjistit stav přerušení (povolené - zakázané).

Instrukce **ld** zajišťuje také zápis obsahu registru do paměti a naopak, přečtení obsahu paměti do registru. Paměťové místo je možno adresovat přímo (adresou) nebo nepřímo (obsahem nějaké dvojice registru nebo šestnáctibitovým registrem). Výsadní místo mezi těmito způsoby má adresování pomocí registru **hl**.

Adresace je v mnemonice **Z80** naznačena kulatými závorkami a u registru **hl** tedy bude v instrukci **ld** napsáno **(hl)** - byte paměti, jehož adresa je uložena v registru **hl**. Zápis **(hl)** můžete používat jako libovolný z obyčejných registrů (vyjma instrukce **ld (hl),(hl)**, která jednak neexistuje a kromě toho by stejně k ničemu nebyla.

Přesuny registrů <=> adresa v HL 7 T-cyklů	
<code>ld (hl), b</code>	<code>ld b, (hl)</code>
<code>ld (hl), c</code>	<code>ld c, (hl)</code>
<code>ld (hl), d</code>	<code>ld d, (hl)</code>
<code>ld (hl), e</code>	<code>ld e, (hl)</code>
<code>ld (hl), h</code>	<code>ld h, (hl)</code>
<code>ld (hl), l</code>	<code>ld l, (hl)</code>
<code>ld (hl), a</code>	<code>ld a, (hl)</code>

Do paměti na adresu v registrovém páru **hl** lze zapsat přímo osmibitovou hodnotu:

Zápis čísla na adresu v HL 10 T-cyklů
<code>ld (hl), N</code>

Z přehledu instrukcí je zřejmé, že registrový pár **hl** je předurčen pro použití jako ukazatel (pointer) do paměti - budeme jej tak velmi často používat.

- - -

Pro adresování paměti můžeme také používat registrové páry **bc** a **de**. Použití je však proti použití registrového páru **hl** silně omezeno:

Přesuny mezi registrem A a pamětí adresovanou BC nebo DE 7 T-cyklů			
<code>ld a, (bc)</code>	<code>ld a, (de)</code>	<code>ld (bc), a</code>	<code>ld (de), a</code>

Obsah jiného registru nebo přímou osmibitovou hodnotu lze do paměti adresované registrovými páry **bc** a **de** zapsat jen prostřednictvím registru **a**.

- - -

Přímý přístup do paměti (na zadanou adresu) je také omezen pouze na registr **a**. Hodnoty ostatních registrů je nutno opět zapisovat a číst prostřednictvím **a**.

Přesuny mezi registrem A a přímo adresovanou pamětí 13 T-cyklů	
<code>ld a, (NN)</code>	<code>ld (NN), a</code>

Na místě **NN** může být v assembleru zapsán libovolný aritmetický výraz. Obě uvedené instrukce patří mezi nejpoužívanější.

Poslední způsob adresování paměti je indexování. Jako adresa se používá součet obsahu indexového registru (**ix**, **iy**) a posunutí v rozsahu **-127..128**, neboli (**ix+E**) a (**iy+E**). Tyto instrukce jsou určeny pro práci s tabulkami, umožňují velmi pohodlný přístup k bytům v okolí bytu, na který ukazuje indexový registr.

V **ld** instrukcích můžete (**ix+E**) a (**iy+E**) používat všude tam, kde lze použít (**hl**). Operační kódy těchto instrukcí se vytváří tak, že se před operační kód instrukce přičepí prefix **IX (221)** nebo prefix **IY (253)**.

Přesuny mezi obvyčejnými registry a adresou (ix+E) nebo (iy+E)			19 T-cyklů
ld (ix+E), b	ld b , (ix+E)	ld (iy+E), b	ld b , (iy+E)
ld (ix+E), c	ld c , (ix+E)	ld (iy+E), c	ld c , (iy+E)
ld (ix+E), d	ld d , (ix+E)	ld (iy+E), d	ld d , (iy+E)
ld (ix+E), e	ld e , (ix+E)	ld (iy+E), e	ld e , (iy+E)
ld (ix+E), h	ld h , (ix+E)	ld (iy+E), h	ld h , (iy+E)
ld (ix+E), l	ld l , (ix+E)	ld (iy+E), l	ld l , (iy+E)
ld (ix+E), a	ld a , (ix+E)	ld (iy+E), a	ld a , (iy+E)

Jak si můžete všimnout, trvají tyto instrukce 19 T-cyklů, nehodí se tedy příliš do programů, které vyžadují velkou rychlost (to platí obecně o používání indexových registrů, tedy v rychlých programech je lépe používat jen obvyčejné registry).

Stejně jako u (**hl**), lze i u (**ix+E**) a (**iy+E**) zapsat na adresu osmibitovou hodnotu:

Zápis čísla na adresu v (ix+E) a (iy+E)		19 T-cyklů
ld (ix+E), N	ld (iy+E), N	

Poznámka: registr **iy** je používán systémem ZX Spectra jako ukazatel do oblasti systémových proměnných (má hodnotu **23610**) a pokud budete používat ve svých programech některé služby **ROM** nebo přerušeni v módu **IM 1**, musíte tuto hodnotu zachovat, což znamená to, že nesmíte registr **iy** používat jinak, než jako ukazatel na systémové proměnné BASICu. Pokud ve svém programu hodnotu registru **iy** změníte, tak jej musíte při návratu do BASICu opět nastavit na hodnotu **23610 (#5C3A)**.

- - -

Mezi instrukce **ld**, které přenášejí 8-bitové hodnoty, patří také instrukce pracující s polovinami indexových registrů (**hx**, **lx**, **hy**, **ly**). Tyto instrukce nepatří mezi standardní instrukce (občas jsou označovány jako "tajné" instrukce) a proto s nimi některé assemblyery neumějí pracovat (GENS), v některých jsou označovány jako **xh**, **xl**, **yh**, **yl** (mrs). Pokud ovšem používáte assembler **PROMETHEUS**, nemusíte se tím zatěžovat. Pro práci s polovinami indexových registrů máte k dispozici tyto instrukce:

Přesun obsahu z osmibitového registru do poloviny indexregistru				8 T-cyklů
ld lx , b	ld hx , b	ld ly , b	ld hy , b	
ld lx , c	ld hx , c	ld ly , c	ld hy , c	
ld lx , d	ld hx , d	ld ly , d	ld hy , d	
ld lx , e	ld hx , e	ld ly , e	ld hy , e	
ld lx , hx	ld hx , hx	ld ly , hy	ld hy , hy	
ld lx , lx	ld hx , lx	ld ly , ly	ld hy , ly	
ld lx , a	ld hx , a	ld ly , a	ld hy , a	

V tabulce si můžete všimnout, že neexistuje možnost přenosu z registrů **h** a **l**. To plyne ze způsobu, jakým tyto instrukce vznikají - všechny výskyty **h** a **l** jsou nahrazeny **hx**, **hy** a **lx**, **ly**. Obdobně existují i instrukce opačné:

Přesun obsahu z poloviny indexregistru do obvyčejného registru 8 T-cyklů			
ld b, lx	ld b, hx	ld b, ly	ld b, hy
ld c, lx	ld c, hx	ld c, ly	ld c, hy
ld d, lx	ld d, hx	ld d, ly	ld d, hy
ld e, lx	ld e, hx	ld e, ly	ld e, hy
ld hx, lx	ld hx, hx	ld hy, ly	ld hy, hy
ld lx, lx	ld lx, hx	ld ly, ly	ld ly, hy
ld a, lx	ld a, hx	ld a, ly	ld a, hy

Asi by Vás napadlo, že budou existovat instrukce umožňující přímé naplnění poloviny indexregistru 8-bitovou hodnotou, zde jsou:

Přímé naplnění poloviny indexregistru 8-bitovou hodnotou 11 T-cyklů			
ld lx, N	ld hx, N	ld ly, N	ld hy, N

Při používání polovin indexregistru nezapomeňte, že ovlivňujete samozřejmě také obsah celého indexregistru - nesmíte-li používat **iy**, nesmíte samozřejmě používat ani **hy**, **ly**!

Uvedené instrukce (8-bitový přesun) nemění stavy indikátorů (flagů). Výjimku tvoří pouze instrukce **ld a, i** a **ld a, r**, které nastavují znovu **SIGN**, **ZERO** a **PARITY** flagy.

* * *

Druhou velikou skupinu instrukcí tvoří instrukce přenosu 16-bitové hodnoty (16-Bit Load Group). Patří sem všechny operace přesunu s dvoiregistry a také operace práce na zásobníku.

Začneme přímým plněním dvojregistru 16-bitovou hodnotou - nejprve obvyčejně:

Přímé plnění obvyčejného dvojregistru 16-bitovou hodnotou 10 T-cyklů			
ld bc, NN	ld de, NN	ld hl, NN	ld sp, NN

Dále jsou na řadě indexové registry:

Přímé plnění indexregistru 16-ti bitovou hodnotou 14 T-cyklů	
ld ix, NN	ld iy, NN

- - -

Pro přesun mezi dvoiregistrem a pamětí existuje několik instrukcí, všechny však používají jen přímé adresování.

Přímý přesun 16-bitové hodnoty mezi HL a pamětí		16 T-cyklů
<code>ld hl, (NN)</code>	<code>ld (NN), hl</code>	

Přesun 16-bitové hodnoty mezi dvoiregistrem a pamětí		20 T-cyklů
<code>ld bc, (NN)</code>	<code>ld (NN), bc</code>	
<code>ld de, (NN)</code>	<code>ld (NN), de</code>	
<code>ld hl, (NN)</code>	<code>ld (NN), hl</code>	
<code>ld sp, (NN)</code>	<code>ld (NN), sp</code>	

Zde neškodí menší objasnění skutečnosti, že pro registrový pár **hl** existují dvě instrukce, které dělají totéž, liší se však rychlostí a také délkou. První instrukce (rychlejší) byly obsaženy už v instrukčním souboru procesoru Intel 8080, na který Z80 navazuje. Další instrukce byly přidány až u Z80 a operace s registrem **hl** byla zopakována ze stejných důvodů, z jakých existuje například instrukce **ld a,a** - hardware.

Pro Vás je podstatné, že každý assembler (z těch co znám) překládá instrukce pracující s **hl** prvním (tedy kratším a rychlejším) způsobem a že každý monitor (dtto) umí disasemblovat obě dvě verze.

Zápis obsahu dvoiregistru do paměti a přečtení obsahu paměti do dvoiregistru můžete provádět také s oběma indexovými registry:

Přímý přesun 16-bitové hodnoty mezi IX (IV) a pamětí		20 T-cyklů
<code>ld ix, (NN)</code>	<code>ld (NN), ix</code>	
<code>ld iy, (NN)</code>	<code>ld (NN), iy</code>	

- - -

Následuje malá skupina instrukcí, které přenáší obsah z jednoho dvoiregistru do druhého - jsou pouze tři a všechny pracují s registrem **SP**. Jsou to tyto:

Přenos obsahu z HL do SP	6 T-cyklů
<code>ld sp, hl</code>	

Přenos obsahu z indexregistru do SP registru		10 T-cyklů
<code>ld sp, ix</code>	<code>ld sp, iy</code>	

To jsou všechny instrukce **ld** pracující s 16-bitovými hodnotami. Stejně jako jejich 8-bitové kolegyně nemění stavy indikátorů (flagů) - tentokrát bez výjimky.

Nynější skupina instrukcí nás přivádí k jednomu důležitému programátorskému pojmu a tím je pojem **zásobník**.

Stručně řečeno, zásobník je datová struktura, do níž se data ukládají tak, že je přístup vždy jen k poslednímu záznamu (ve smyslu poslední vloženému - nejčerstvějšímu ze všech, které tam jsou). Dobře si tuto situaci můžete představit takto:

V úřadě sedí úředník, řekněme mu **U**. Jeho práce spočívá v tom, že přijímá a vyřizuje žádosti, dělá to tak, že vyřizuje nejnovější žádost, pokud přijde nová žádost, otevře šuplík, vyřizovanou žádost do něj vloží a začne vyřizovat novou. Když náhodou nějakou žádost vyřídí, otevře šuplík a vytáhne z něj tu žádost, která je navrchu a pokračuje v jejím vyřizování. Pokud však **U** obdrží další žádost, vrátí starou zpět do šuplíku a věnuje se nové. Při práci **U** mohou nastat dva problémové okamžiky - první nastane tehdy, je-li šuplík prázdný a **U** nedostává žádné žádosti. V takovém případě upadá **U** do strnulého stavu a čeká na nějakou žádost. Druhý případ je horší - **U** dostává tolik žádostí, že mu šuplík přeteče.

Zásobník je vlastně svého druhu šuplík a procesor pak úředník.

Na rozdíl od úřadů, kde tento způsob práce zřejmě patří k převládajícím a má tu nevýhodu, že žádosti na dne šuplíku takřka nemají šanci na vyřízení (pokud chcete, aby Vaše žádost byla vyřízena, musíte vystihnout okamžik, kdy je žádostí málo, a tehdy podat žádost, jinak bude "pohřbena"), má v programování tento způsob zacházení z daty velký význam.

Anglicky se zásobníku říká **stack** a způsob práce je výstižně nazván jako **LIFO** (Last In First Out - poslední dovnitř, první ven). Když už jsme u těch zkratek, existuje ještě druhá podobná zkratka - **FIFO**, což není jen jméno jistého časopisu ale název pro další datovou strukturu nazývanou česky **fronta** (anglicky **queue** a First In First Out - první dovnitř, první ven).

Na úřadech se fronty vyskytují také, nikoliv však v kancelářích ale před nimi. Do šuplíku na dno je totiž velice špatný přístup.

Zanechme exkurzi do úřadů a vraťme se k programování. Pro zásobník tedy máme definovány dvě operace - **push** (vlození do šuplíku) a **pop** (vybrání ze šuplíku). Zásobník se nám může vyprázdnit (prázdný šuplík - **empty**) nebo přetéct (plný šuplík - **overflow**). Občas se také hodí podívat se na vrchol zásobníku co tam je (nejvrchnější žádost v našem šuplíku - **top**).

Procesor Z80 umí také se zásobníkem pracovat, má na to instrukce **push** a **pop** ale nejen ty. Na zásobník se odkládají 16-bitové hodnoty. Potíž je v tom, že na zásobník se ukládají také návratové adresy (instrukce **call** a **ret**). Podržíme-li se našeho průměru s úředníkem, je to asi totéž, jako kdyby si náš pan **U** vozil šuplík s sebou a vždy když někam jede si do něj uložil adresu odkud vyjel, aby se mohl vrátit. V cíli si zařaduje (použije šuplík) a při návratu ze šuplíku vytáhne první papír, podívá se na adresu a tam vyrazí - pokud se splete a v šuplíku nahoře je nějaká žádost, pak dojede bůh ví kam, může také vyřídít místo žádosti papír s adresou a výsledek je stejný - žalostný. U procesoru je situace ještě horší protože čísla mohou znamenat cokoliv.

Proto si dávejte na zásobník obzvláštní pozor! Při návratu z podprogramu musí být zásobník ve stejném stavu, v jakém byl při vstupu do podprogramu. Chyby se dělají často při větvení programu, kdy nějaká větev neošetřuje zásobník - to bývá často zdrojem "nevysvětlitelných" chyb, kdy program skoro vždycky chodí, jen občas spadne.

Instrukce **push** odečte od **sp** registru **2** a na (**sp**) uloží obsah určeného dvoiregistru. Naproti tomu instrukce **pop** nejprve přečte do dvoiregistru obsah (**sp**) a potom přičte k **sp** registru **2**. Provedete-li tedy **push** a vzápětí poté **pop** se stejným registrem, nezmění se nic.

Instrukce **push** a **pop** lze také použít pro přenos dat z dvojregistru do dvojregistru. Výhodné je to však jen u indexregistru (u ostatních je lépe použít přenos po částech, to znamená třeba **ld b,d** a **ld c,e** jako "**ld bc,de**", která neexistuje, výjimku tvoří **af**, kde to jinak než přes zásobník udělat nejde). Důvodem je tu rychlost - použití dvou instrukcí **ld** je dlouhé **8 T-cyklů**, naproti tomu **push** a **pop** mají dohromady **21 T-cyklů** (skoro třikrát tolik). Pouze u indexových registrů by použití instrukcí **ld** buď vůbec nešlo nebo by byl delší kód programu.

Uložení obvyčejného dvojregistru na zásobník			11 T-cyklů
<code>push bc</code>	<code>push de</code>	<code>push hl</code>	<code>push af</code>

Přečtení obvyčejného dvojregistru ze zásobníku			10 T-cyklů
<code>pop bc</code>	<code>pop de</code>	<code>pop hl</code>	<code>pop af</code>

Pro indexregistry jsou k dispozici stejné operace:

Uložení indexregistru na zásobník		15 T-cyklů
<code>push ix</code>	<code>push iy</code>	

Přečtení indexregistru ze zásobníku		14 T-cyklů
<code>pop ix</code>	<code>pop iy</code>	

* * *

Na řadě je skupina vyměňovacích instrukcí (Exchange Group). Instrukce této skupiny mohou navzájem vyměnit obsahy dvojregistry nebo mezi dvojregistrem a pamětí.

Nejčastěji používaná instrukce vyměňuje obsah dvojregistru **hl** a **de**. Je užitečná, pokud máte v registru **de** nějakou adresu a chcete s ní provést nějaké operace - prohodit **hl** a **de**, provést operace s **hl** (mnohem více instrukcí k dispozici) a prohodit zpátky. Také se hodí pro dočasné uklizení registru **hl** v případě, že registr **de** není zrovna použit (je to rychlejší než uložení do paměti nebo na zásobník).

Výměna obsahů mezi HL a DE	4 T-cyklů
<code>ex de,hl</code>	

Pro práci se záložními (alternativními) registry slouží dvě instrukce. První prohodí obsah mezi hlavním a záložním dvojregistrem **af** - užitečné pro uložení flagů a obsahu registru **a** (opět značně rychlejší než pomocí paměti nebo zásobníku). Občas je však třeba uchovat buď jen flagy nebo jen obsah registru **a**, v takovém případě si musíte pomoci jinak.

Druhá instrukce prohazuje vzájemně obsahy ostatních obyčejných hlavních a alternativních registrů. Vymění tedy navzájem **bc**, **de** a **hl** s jejich dvojníky - užitečné hlavně když potřebujete uložit všechny registry a záložní nejsou nijak používané (opět značně rychlejší než zásobník o paměti nemluvě - tady musím upozornit na jednu skutečnost, zásobník je také v paměti, zde jsou slovem zásobník myšleny instrukce s ním pracující **push** a **pop** a pod slovem paměť myšleny instrukce **ld**).

Výměna obsahů mezi AF a AF' 4 T-cykly
<code>ex af,af'</code>

Výměna BC, DE a HL za záložní 4 T-cyklů
<code>exx</code>

- - -

Výměna obsahu dvoiregistrů s obsahem paměti je možná s buňkou (**sp**) a jako dvoiregistr můžete použít **hl** a oba indexregistry **ix** a **iy**.

Instrukce tedy vymění obsah zvoleného dvoiregistru s vrcholem zásobníku - pro provedení instrukce je na zásobníku to, co bylo ve dvoiregistru, a ve dvoiregistru to, co bylo na vrcholu zásobníku.

Použijeme-li opět našeho pana U, je to jako kdyby se při vyřizování jedné žádosti rozhodl vyřizovanou žádost vložit do šuplíku a začít vyřizovat první žádost ze šuplíku.

Výměna obsahu (SP) a HL 19 T-cyklů
<code>ex (sp),hl</code>

Budete-li chtít provést operaci, která by odpovídala neexistující instrukci **ex (sp),de**, můžete to snadno docílit touto sekvencí instrukcí:

```
ex de,hl
ex (sp),hl
ex de,hl
```

Instrukce **ex (sp),hl** se také vhodně využívá pro zpomalení běhu programu, dáte-li totiž dvě za sebe, nestane se nic ale bude to trvat **38 T-cyklů**.

Pro indexregistry jsou to tyto dvě instrukce:

Výměna obsahů mezi (SP) a indexregistrem 23 T-cyklů	
<code>ex (sp),ix</code>	<code>ex (sp),iy</code>

Vyměňovací instrukce nemění nastavení indikátorů (flagů). Pozor však na instrukci **ex af,af'**, která sice nemění indikátory ale mění celý registr **f**, v němž jsou uloženy!

Skupina instrukcí pro přenos bloků (Block Transfer Group) je tvořena jen čtyřmi instrukcemi, co do užitečnosti je velmi významná.

Blokový přesun bez opakování		16 T-cyklů
<code>ldi</code>		<code>ldd</code>

Instrukce `ldi` provádí v uvedeném pořadí tyto činnosti: Přenese obsah z bytu, na který ukazuje registr `hl`, do bytu, na který ukazuje registr `de` (symbolicky **(de) ← (hl)**). Oba registry, `hl` i `de`, zvětší o jedničku (**hl ← hl + 1, de ← de + 1**). Odečte jedničku od registru `bc` (**bc ← bc - 1**) a testuje výsledek na nulu. Pokud je v `bc` nula, vynuluje se příznak **P/V** (platí podmínka **PO**).

Instrukce `ldd` provádí téměř totéž, pouze oba pointery (`hl` a `de`) o jedničku **zmenšuje**. Jako pomůcka Vám poslouží poslední písmeno mnemoniky; **ldd(crement)** a **ldi(ncrement)**.

Registr `hl` slouží jako ukazatel na zdrojový blok, registr `de` ukazuje na cílový blok a registr `bc` je použit jako počítadlo přenesených bytů.

Obě instrukce se hodí tam, kde chcete přenášet paměťový blok z jedné adresy na druhou a navíc při každém přesunu provést nějakou akci. Instrukce `ldi` je vhodná při přesunování bloku odpředu (od nižších adres k vyšším) a `ldd` naopak při přesunu odzadu (od vyšších adres k nižším).

Aby bylo možno přenášet bloky najednou, existují instrukce, které nejen testují hodnotu `bc` na nulu ale podle výsledku také buď skončí nebo provádění opakují:

Blokový přesun s opakováním		21 (16) T-cyklů
<code>ldir</code>		<code>lddr</code>

Písmeno `r` v mnemonice znamená anglické slovo **repeat** (opakuj). Stejně jako u `ldi` a `ldd` jsou písmena `i` a `d` od slov **increment** a **decrement**. Obě instrukce opět slouží k přesunu bloků, lze je použít i k vyplnění paměti zvoleným číslem (jak se to udělá se dozvíte v dalších kapitolách).

Budete-li chtít přenášet blok paměti tak, že se bude kus cílové oblasti překrývat se zdrojovou oblastí, musíte si dát pozor na to, jestli použijete `ldir` nebo `lddr`. Je to proto, aby se zdrojová oblast přepisovala z té strany, která už je přečtena. Pokud budete blok posunovat dolů (cílová oblast je pod zdrojovou oblastí) použijte `ldir`, v opačném případě pak `lddr` (Opakují, že toto rozlišení má smysl jen když se obě oblasti překrývají!).

Uvedený počet T-cyklů u instrukcí `ldir` a `lddr` se vztahuje k jednomu přenosu. První číslo je délka přenosu spolu se skokem zpět na začátek instrukce, druhé číslo je délka pouze přenosu (stejná jako u `ldd`, `ldi`). Celkovou časovou náročnost u instrukce `ldir` (**lddr**) spočítáte takto **bc * 21 - 5 = počet T-cyklů**, `bc` je zde hodnota v `bc` registru před provedením instrukce.

* * *

Instrukce pro blokové hledání (Block Search Group) jsou podobně výkonná skupina jako instrukce pro přenášení bloku, s předchozí skupinou mají i mnoho společných vlastností.

Instrukce umožňují vyhledat v daném bloku výskyt daného čísla. Nejprve tabulka:

Blokové hledání bez opakování		16 T-cyklů
cpi		cpd

Instrukce **cpi** porovnává obsah registru **a** s obsahem bytu, na který ukazuje registr **hl**, zvětší **hl** o jedničku a zmenší **bc** o jedničku. Pokud je shoda mezi **a** a **(hl)**, je nastaven příznak **ZERO** (platí **Z**). Pokud je v **bc** po odečtení jedničky nula, vynuluje příznak **P/V** (platí **PE**).

Instrukce **cpd** se od **cpi** liší jen tím, že registr **hl** nezvětšuje ale zmenšuje (decrement).

Obě instrukce existují i s automatickým opakováním dokud není nalezena shoda nebo dokud je **bc** nenulový. Chcete-li tedy najít mezi adresami 30000 a 40000 výskyt bytu s hodnotou 123, provedete to takto:

```
ld hl, 30000 ; začátek oblasti
ld bc, 10000 ; délka oblasti
ld a, 123 ; hledaná hodnota
cpi r ; hledej
jr z, FOUND ; odskok v případě, že byl výskyt
```

Registr **hl** ukazuje po nalezení **za** první výskyt hledaného čísla.

Blokové hledání s opakováním		21 (16) T-cyklů
cpir		cpdr

Prohlédněte si předchozí skupinu instrukcí (**ldi**, **ldd**, **ldir**, **lddr**), dozvíte se tam další podrobnosti o T-cyklech.

* * *

Na řadu přicházejí instrukce 8-bitové aritmetiky. Jde o sčítání a odčítání a o sčítání a odčítání s použitím **CARRY** flagu. Všechny tyto instrukce používají jako první operand akumulátor (registr **a**) a výsledek také ukládají do téhož registru.

Osmibitové sčítání - **add** - instrukce přičte k akumulátoru hodnotu druhého operandu. Flagy se nastavují podle výsledku. Carry je nastaven v případě, že výsledek sčítání překročí číslo 255.

Osmibitové odečítání - **sub** - instrukce odečte od akumulátoru hodnotu druhého operandu. Flagy se nastavují podle výsledku. Carry je nastaven v případě, že výsledek odečítání je záporný.

Osmibitové sčítání s Carry - **adc** - instrukce přičte k akumulátoru hodnotu druhého operandu a hodnotu Carry flagu. Flagy se nastavují stejně jako u sčítání.

Osmibitové odečítání s Carry - **sbc** - instrukce odečte od akumulátoru hodnotu druhého operandu a hodnotu Carry flagu. Flagy se nastavují stejně jako u odečítání.

Možné instrukce 8-bitové aritmetiky naleznete v následujících tabulkách:

Instrukce 8-bitové aritmetiky, druhý operand registr			4 T-cykly
add a , b	sub b	adc a , b	sbc a , b
add a , c	sub c	adc a , c	sbc a , c
add a , d	sub d	adc a , d	sbc a , d
add a , e	sub e	adc a , e	sbc a , e
add a , h	sub h	adc a , h	sbc a , h
add a , l	sub l	adc a , l	sbc a , l
add a , a	sub a	adc a , a	sbc a , a

Instrukce 8-bitové aritmetiky, druhý operand polovina indexregistru			8 T-cyklů
add a , hx	sub hx	adc a , hx	sbc a , hx
add a , lx	sub lx	adc a , lx	sbc a , lx
add a , hy	sub hy	adc a , hy	sbc a , hy
add a , ly	sub ly	adc a , ly	sbc a , ly

Instrukce 8-bitové aritmetiky, druhý operand číslo			7 T-cyklů
add a , N	sub N	adc a , N	sbc a , N

Instrukce 8-bitové aritmetiky, druhý operand (HL)			7 T-cyklů
add a , (hl)	sub (hl)	adc a , (hl)	sbc a , (hl)

Instrukce 8-bitové aritmetiky, druhý operand (IX+E) nebo (IY+E)			19 T-cyklů
add a , (ix+E)	sub (ix+E)	adc a , (ix+E)	sbc a , (ix+E)
add a , (iy+E)	sub (iy+E)	adc a , (iy+E)	sbc a , (iy+E)

U mnemoniky instrukce sub je zajímavá anomálie, není tu uveden první operand ačkoliv u ostatních instrukcí uveden je.

- - -

Mezi osmibitové aritmetické instrukce patří i inkrementy (zvětšení o jedničku) a dekrementy (zmenšení o jedničku).

U instrukcí inc a dec si dejte pozor na to, že nenastavují **CARRY** flag!

Inkrement a dekrement registru						4 T-cykly
inc b	inc c	inc d	inc e	inc h	inc l	inc a
dec b	dec c	dec d	dec e	dec h	dec l	dec a

Inkrement a dekrement poloviny indexregistru				8 T-cyklů
inc hx	inc lx	inc hy	inc ly	
dec hx	dec lx	dec hy	dec ly	

Inkrement a dekrement (HL)	11 T-cyklů
inc (hl)	
dec (hl)	

Inkrement a dekrement (IX+E) nebo (IV+E)		23 T-cyklů
inc (ix+E)	inc (iy+E)	
dec (ix+E)	dec (iy+E)	

* * *

Logické instrukce jsou další rozsáhlá a důležitá skupina instrukcí. Svými vlastnostmi se podobají aritmetickým instrukcím - zcela totožné operandy, první operand je vždy akumulátor a stejná časová náročnost.

Bitový logický součin - **and** - provádí operaci logického součinu po bitech. Logický součin má výsledek **1** právě když oba operandy mají hodnotu **1**, jinak je výsledek **0**. Podle výsledku se nastavují také **SIGN** a **ZERO** flagy. Pro větší názornost uvedu příklad:

```

    11101101
and 10100001
-----
    11101101

```

Bitový logický součet - **or** - provádí operaci logického součtu po bitech. Logický součet má výsledek **1** právě když alespoň jeden z operandů má hodnotu **1**. Jinak má výsledek **0**. Podle výsledku se nastavují také **SIGN** a **ZERO** flagy. Opět, příklad:

```

    11101101
or  10100001
-----
    11101101

```

Bitový exkluzivní součet - **xor** - provádí operaci exkluzivního součtu po bitech. Exkluzivní součet má výsledek **1** právě když mají oba operandy různou hodnotu. Pokud mají hodnotu stejnou, je výsledek **0**. Podle výsledku se nastavují **SIGN** a **ZERO** flagy. Příklad:

```

    11101101          01001100
xor 10100001          xor 10100001
-----
    01001100          11101101

```

U exkluzivního součtu si všimněte, že pokud provedete exkluzivní součet dvakrát týmž číslem, bude výsledek stejný jako předchozí stav (viz druhý sloupec příkladu).

Při práci s grafikou se bez instrukcí **and**, **or**, **xor** neobejdete. Instrukce **and** se používá pro vybrání zvolených bitů (ostatní vynuluje). Instrukce **or** je užitečná při nastavování některých bitů (v grafice přikreslení k již existujícímu). Instrukce **xor** je používána na převrácení vybraných bitů (v grafice tato instrukce umožňuje něco nakreslit tak, aby se to dalo druhým nakreslením smazat).

porovnání - **cp** (compare) - porovná obsah akumulátoru a uvedený operand. Nastaví **ZERO** flag když jsou oba stejné. Nastaví **CARRY** když je druhý operand větší než obsah akumulátoru. Instrukce **cp** je realizována jako neprovedené odečítání, které však podle výsledku nastaví indikátory (flagy) - na toto si vzpomeňte když nebudete vědět jak se nastavuje **CARRY** flag a můžete si to odvodit. Malá tabulka pro možné výsledky porovnání a pro nastavení flagů **CARRY** a **ZERO**.

Nastavení flagů u instrukce			cp b
a = b	z	c	m
a < > b	nz	?	?
a > b	nz	nc	p
a > = b	?	nc	p
a < b	nz	c	m
a < = b	?	c	m

V tabulce vidíte, že příznaky **SIGN** a **CARRY** jsou totožné co do informace, výhodnější je však používat **CARRY** flag neboť relativní skoky s podmínkami testujícími **SIGN** flag neexistují.

Opět celkové tabulky:

Logické instrukce pracující s registrem			4 T-cyklů
and b	or b	xor b	cp b
and c	or c	xor c	cp c
and d	or d	xor d	cp d
and e	or e	xor e	cp e
and h	or h	xor h	cp h
and l	or l	xor l	cp l
and a	or a	xor a	cp a

Logické instrukce pracující s polovinou indexregistru			8 T-cyklů
and hx	or hx	xor hx	cp hx
and lx	or lx	xor lx	cp lx
and hy	or hy	xor hy	cp hy
and ly	or ly	xor ly	cp ly

Logické instrukce pracující s přímým operandem (číslem)			7 T-cyklů
and N	or N	xor N	cp N

Logické instrukce pracující s (HL)			7 T-cyklů
and (hl)	or (hl)	xor (hl)	cp (hl)

Logické instrukce pracující s (IX+E) nebo (IY+E)			19 T-cyklů
and (ix+E)	or (ix+E)	xor (ix+E)	cp (ix+E)
and (iy+E)	or (iy+E)	xor (iy+E)	cp (iy+E)

* * *

Speciální aritmetické instrukce - patří sem instrukce pracující s akumulátorem a také instrukce pro práci s **CARRY** flagem.

Desítková korekce - **daa** - převod hodnoty akumulátoru do pakované BCD formy, musí následovat po sčítání nebo odečítání s pakovaným BCD operandem. Pakovaná BCD forma (binary coded decimal) je způsob uložení dvouciferného desítkového čísla - jednotky jsou uloženy ve spodních čtyřech bitech, desítky v horních čtyřech bitech obvyklým způsobem. Hodnota čísla je přímo vidět při hexadecimálním výpisu hodnoty akumulátoru.

Desítková korekce	4 T-cykly
daa	

Komplement akumulátoru - **cpl** - bitové převrácení hodnoty akumulátoru. Každý bit je nastaven na opačnou hodnotu. Tato instrukce nenastavuje žádný z přímo testovatelných flagů. Instrukce provádí stejnou operaci jako **xor %11111111**.

Komplement akumulátoru	4 T-cykly
cpl	

Negace akumulátoru - **neg** - obsah akumulátoru je odečten od nuly (násobení -1, převrácení znaménka u čísla v akumulátoru).

Negace akumulátoru	8 T-cyklů
neg	

- - -

Nastavení **CARRY** flagu - **scf** (set carry flag) - uloží do **CARRY** flagu **1**, platí tedy podmínka **C**. Instrukce se uplatní u instrukcí rotací a posuvů.

Nastavení CARRY flagu 4 T-cykly
scf

Komplementace **CARRY** flagu - **scf** (complement carry flag) - operace změni hodnotu **CARRY** flagu na opačnou (**C** změni na **NC**, **NC** změni na **C**).

Komplement CARRY flagu 4 T-cykly
ccf

Pokud chcete **CARRY** flag vynulovat, můžete to provést tak, že jej nastavíte a potom změníte. V 99.9% případů však stačí použít některou logickou instrukci pracující s akumulátorem, například **or a**, jediný případ, kdy tento postup nelze použít, je když nepotřebujete změnit žádný flag, pak musíte použít **scf** a **ccf**.

Zvláštní instrukcí v instrukčním souboru je instrukce **nop** (no operation). Tato jakási "neinstrukce" dělá to, že 4 T-cykly nedělá nic. Je vhodná pro programy, které musí trvat přesně určený časový interval jako "časová" vycpávka. Použije se i v programech, které sama modifikují svůj vlastní kód.

Neinstrukce 4 T-cykly
nop

* * *

Procesor Z80 umí pracovat se třemi módy přerušeni. Ve Spectru se používají pouze dva z nich. Přerušeni je akce, která je vyvolána signálem mimo procesor (hodiny, periferie, ..). Po obdrženi žádosti o přerušeni (signál zvenku) zakáže procesor přerušeni, uloži na zásobník adresu následující instrukce a podle módu skočí na určitou adresu, kde musí být program pro obsluhu přerušeni.

V **módu 1** skáče procesor na adresu 56, kde je v ROM Spectra program pro čtení klávesnice a pro obsluhu časového čítače.

V **módu 2** získá procesor ze sběrnice dolní byte adresy, horní byte adresy je uložen v registru **i**, na takto získané adrese si procesor přečte adresu a na ní skočí.

Popsané přerušeni se nazývá **maskovatelné** protože je jej možno zakázat. Z80 zná ještě další typ přerušeni - **nemaskovatelné** - to však nelze na Spectru bez zásahu do hardware použít.

Povolení přerušeni - **ei** (enable interrupt). Instrukce povolí přijeti žádosti o přerušeni.

Povolení přerušeni 4 T-cykly
ei

Zakázání přerušení - **di** (disable interrupt). Instrukce zakáže přijetí žádosti o přerušení.

Zakázání přerušení	4 T-cykly
d i	

Instrukce čekání na přerušení - **halt**. Zastaví procesor do doby, než dojde k přijetí žádosti o přerušení. Pokud je ovšem přerušení zakázáno, bude procesor čekat věčně - dosti častý důvod, proč se program "zadře". Proto raději před každou instrukcí **halt** vložte ještě instrukci **ei**.

Čekání na přerušení	? T-cyklů
h a l t	

Nastavení přerušovacího módu - **im**. Instrukce nastaví přerušovací mód procesoru podle čísla, které je za ní uvedeno.

Nastav přerušovací mód			4 T-cykly
im 0	im 1	im 2	

* * *

Instrukce 16-bitové aritmetiky. Jde o 16-bitové sčítání, sčítání s použitím **CARRY**, odečítání s použitím **CARRY**, inkrement a dekrement.

16-bitové sčítání - **add** - přičtení obsahu vybraného registru k obsahu registru **hl** a indexregistrů **ix** a **iy**. U této instrukce si dejte pozor na skutečnost, že nemění obsahy **ZERO** a **SIGN** flagů, podle výsledku se nastavuje pouze hodnota **CARRY**.

16-bitové sčítání používající obvyklé registry			11 T-cyklů
add hl, bc	add hl, de	add hl, hl	add hl, sp

16-bitové sčítání používající indexregistry			15 T-cyklů
add ix, bc	add ix, de	add ix, ix	add ix, sp
add iy, bc	add iy, de	add iy, iy	add iy, sp

Instrukce **add hl,hl** (**add ix,ix**, **add iy,iy**) provádějí vlastně násobení 2 vybraného registru. Také je možné je použít jako aritmetický posun doleva (zprava vstupuje 0, vlevo vystupuje bit do **CARRY**). Instrukce **add hl,sp** je použitelná pro přenos hodnoty z **sp** do **hl** takto: **ld hl,0 add hl,sp** (nejkratší způsob).

16-bitové sčítání s použitím **CARRY** (s přenosem) - **adc** - přičte k registru **hl** zvolený registr a hodnotu **CARRY** flagu. Na rozdíl od 16-bitového sčítání tato instrukce nastavuje podle výsledku všechny flagy - použijete ji všude, kde je to třeba místo obyčejného sčítání, nesmíte však před použitím zapomenout vynulovat **CARRY** flag. Přenos využijete při práci s celými čísly většími než 65535 (tady čísla, které potřebují pro uložení více než dva byty). U těchto čísel se musí sčítání provádět postupně a ve vyšších řádech přičítat vždy přenos z řádů nižších.

16-bitové sčítání s přenosem v CARRY			15 T-cyklů
<code>adc hl, bc</code>	<code>adc hl, de</code>	<code>adc hl, hl</code>	<code>adc hl, sp</code>

Instrukci **adc hl,hl** můžete použít také jako rotaci doleva registru **hl**.

16-bitové odčítání s použitím **CARRY** je důležitá instrukce - s její pomocí lze napsat porovnání 16-bitových registrů (něco jako **cp**, které však pro dvoiregistry neexistuje). Porovnání napíšete takto (pro **hl** a **bc**):

```

or      a                ; vynulování CARRY
sbc    hl, bc           ; odečtení pro nastavení ZERO a SIGN
add    hl, bc           ; uvedení do původního stavu, CARRY

```

Tady je vidět důvod, proč 16-bitové sčítání nenastavuje ani příznak **ZERO** ani příznak **SIGN**. Pokud víte, že je příznak přenosu vynulován, nemusíte instrukci **or a** uvádět, dejte si však dobrý pozor, aby tomu tak skutečně bylo - opět zdroj "podivných" chyb.

Budete-li odečítání s přenosem používat pro odečítání, nezapomeňte vynulovat příznak přenosu (**CARRY**) - opět možné chyby.

16-bitové odčítání s přenosem v CARRY			15 T-cyklů
<code>sbc hl, bc</code>	<code>sbc hl, de</code>	<code>sbc hl, hl</code>	<code>sbc hl, sp</code>

Instrukci **sbc hl,hl** můžete použít k tomu, aby se do **hl** registru v závislosti na stavu příznaku **CARRY** zapsala buď **0** nebo **65535**. Je to zvláštní, ale občas se to může hodit (stejně tak u osmibitové instrukce **sbc a,a**, to využijeme v kapitole o tisku znaků).

- - -

Mezi 16-bitovou aritmetiku patří také inkrementy a dekrementy dvoiregistrů. Zde si dejte pozor na to, že tyto instrukce nemají žádný vliv na příznaky - obvykle se to hodí, občas by naopak neškodilo, kdyby příznaky nastavovaly.

16-bitový inkrement a dekrement obyčejných dvoiregistrů			6 T-cyklů
<code>inc bc</code>	<code>inc de</code>	<code>inc hl</code>	<code>inc sp</code>
<code>dec bc</code>	<code>dec de</code>	<code>dec hl</code>	<code>dec sp</code>

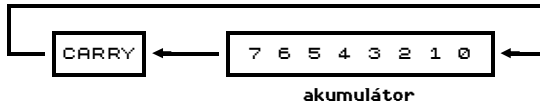
16-bitový inkrement a dekrement indexregistrů			10 T-cyklů
<code>inc ix</code>	<code>inc iy</code>	<code>dec ix</code>	<code>dec iy</code>

Na řadě jsou instrukce rotací a posuvů (Rotate and Shift Group), jejich využití je hlavně při práci s grafikou ale nejen tam. Nejprve projdeme instrukce pracující pouze s akumulátorem (registrem **a**).

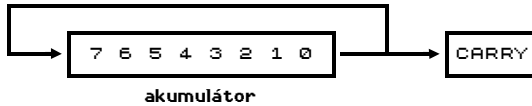
Cyklická rotace akumulátoru doleva - **rlca** (Rotate Left Circular Accumulator) - rotuje bity doleva, co vlevo vystoupí vstoupí vpravo a je také uloženo do **CARRY**. Názornější asi bude schematický obrázek:



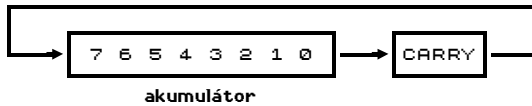
Rotace akumulátoru doleva - **rla** (Rotate Left Accumulator) - rotuje akumulátor doleva, co vlevo vystoupí, jde do **CARRY** a vpravo vstoupí původní hodnota **CARRY**.



Cyklická rotace akumulátoru doprava - **rrca** (Rotate Right Circular Accumulator) - je obdoba instrukce **rlca**, rotuje se opačným směrem.



Rotace akumulátoru doprava - **rra** (Rotate Right Accumulator) - je obdoba instrukce **rla**, rotuje opačným směrem.



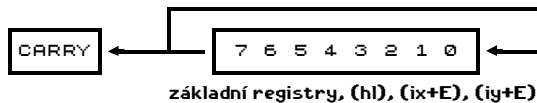
Pro úplnost ještě souhrnná tabulka instrukcí:

Rotace pracující pouze s akumulátorem			4 T-cykly
rlca	rla	rrca	rra

Pozor, instrukce pro rotaci akumulátoru nenastavují žádný jiný flag než **CARRY**. Budete-li chtít příznaky nastavit, musíte použít instrukce z následující podskupiny.

Z80 nabízí ještě další instrukce rotací a také posuvů, ty už dokáží pracovat nejen se všemi základními registry ale i s pamětí adresovanou registrem **hl** a indexovými registry. Oproti již uvedeným instrukcím trvají dvojnásobný čas a jsou delší.

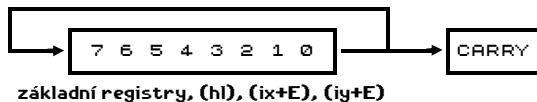
Cyklická rotace doleva - **rlc** (Rotate Left Circular) - rotuje bity doleva, co vlevo vystoupí vstoupí vpravo a je také uloženo do **CARRY**.



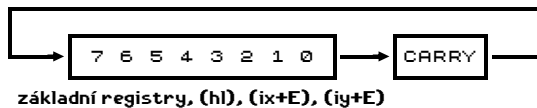
Rotace doleva - **rl** (Rotate Left) - rotuje doleva, co vlevo vystoupí, jde do **CARRY** a vpravo vstoupí původní hodnota **CARRY**.



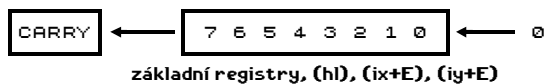
Cyklická rotace doprava - **rrc** (Rotate Right Circular) - je obdoba instrukce **rlc**, rotuje se opačným směrem.



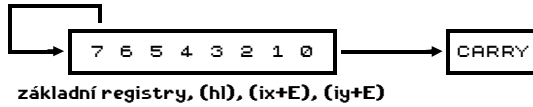
Rotace doprava - **rr** (Rotate Right) - je obdoba instrukce **rl**, rotuje opačným směrem.



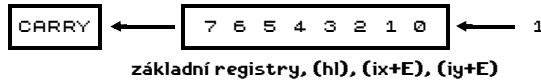
Aritmetický posun doleva - **sll** (Shift Left Arithmetic) - posune bity doleva, vpravo vstupuje **0**, vlevo vystupující bit je přenesen do **CARRY**. Operace odpovídá násobení dvěma.



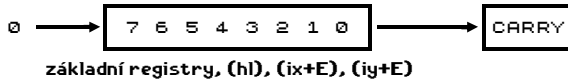
Aritmetický posun doprava - **sra** (Shift Right Arithmetic) - posune bity doprava, vystupující bit je vložen do **CARRY**. Sedmý bit se nemění. Operace odpovídá dělení dvěma pro číslo se znaménkem.



Invertovaný aritmetický posun doleva - **sla** (Shift Left Inverted Arithmetic) - posune bity doleva, zprava vstupuje **1**, vystupující bit jde do **CARRY**. Tato instrukce bývá uváděna mezi "tajnými" instrukcemi Z80, některé assemblyery ji nedokáží překládat. Někdy je tato instrukce označována jako **SLL** (produkt Y.T.R.C).



Logický posun doprava - **srl** (Shift Right Logical) - posune bity doprava, zleva vstupuje **0**, vystupující bit je uložen do **CARRY**.



Všechny uvedené rotace a posuny nastavují příznaky podle výsledku operace.

Instrukce rotací a posunů pro obvyčejné registry						8 T-cyklů	
r l c b	r l b	r r c b	r r b	s l a b	s r a b	s l i a b	s r l b
r l c c	r l c	r r c c	r r c	s l a c	s r a c	s l i a c	s r l c
r l c d	r l d	r r c d	r r d	s l a d	s r a d	s l i a d	s r l d
r l c e	r l e	r r c e	r r e	s l a e	s r a e	s l i a e	s r l e
r l c h	r l h	r r c h	r r h	s l a h	s r a h	s l i a h	s r l h
r l c l	r l l	r r c l	r r l	s l a l	s r a l	s l i a l	s r l l
r l c a	r l a	r r c a	r r a	s l a a	s r a a	s l i a a	s r l a

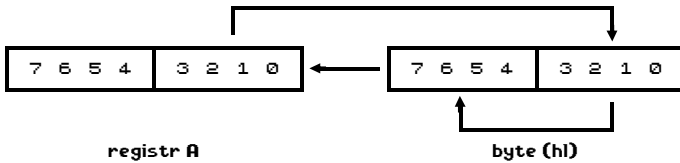
Rotace a posuvy s (HL)	15 T-cyklů
r l c	(hl)
r l	(hl)
r r c	(hl)
r r	(hl)
s l a	(hl)
s r a	(hl)
s l i a	(hl)
s r l	(hl)

Rotace a posuvy s (IX+E), (IY+E) 23 T-cyklů			
r lc	(ix+E)	r lc	(iy+E)
r l	(ix+E)	r l	(iy+E)
r rc	(ix+E)	r rc	(iy+E)
r r	(ix+E)	r r	(iy+E)
s la	(ix+E)	s la	(iy+E)
s ra	(ix+E)	s ra	(iy+E)
s lia	(ix+E)	s lia	(iy+E)
s rl	(ix+E)	s rl	(iy+E)

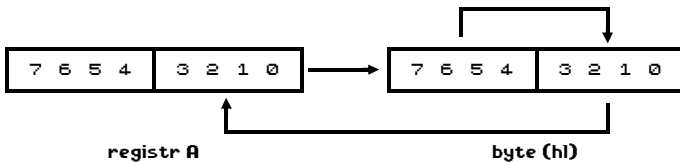
- - -

Následující dvě instrukce jsou rotace určené pro práci s pakovanou BCD formou zápisu čísla (viz instrukce **daa**). Zatím jsem se s nimi setkal jen u několika kódovacích programů (hry ULTIMATE), tyto instrukce se téměř nepoužívají.

Číslicová rotace doleva - **rld** (Rotate Left Digit). U této instrukce se raději nebudu pokoušet o popis, raději si rovnou prohlédněte obrázek. Instrukce pracuje s registrem **a** a s bytem paměti adresovaným pomocí registru **hl**.



Číslicová rotace doprava - **rrd** (Rotate Right Digit). Opět raději obrázek:



Číslicové posuny	18 T-cyklů
r ld	r ld

Podmíněné a nepodmíněné instrukce skoku (Jump Group) jsou instrukce, bez nichž se programovat prostě nedá. Z80 poskytuje skoky absolutní i relativní. Absolutní skok nese informaci o adrese. Relativní skok si nese informaci jak daleko má skočit vzhledem k adrese na niž je instrukce uložena. Toto se obzvlášť využije v programech, u kterých není známo, na jaké adrese budou pracovat.

Absolutní nepodmíněný skok - **jp NN** - provedení této instrukce má za následek přesun provádění na adresu **NN**. Jde vlastně o zápis čísla **NN** do registru **pc** (jakési **ld pc, NN**).

Absolutní nepodmíněný skok	10 T-cyklů
jp	NN

Absolutní podmíněný skok - **jp cc, NN** - kde **cc** je podmínka vztahující se k jednotlivým flagům, instrukce provádí skok jen v případě, že je splněna zvolená podmínka. Možných podmínek je u absolutních skoků celkem 8, zde jsou:

- nz** - non zero (flag **ZERO** je nastaven na nulu)
- z** - zero (flag **ZERO** je nastaven na jedničku)
- nc** - non carry (flag **CARRY** je nastaven na nulu)
- c** - carry (flag **CARRY** je nastaven na jedničku)
- po** - parity odd (flag **PARITY/OVERFLOW** je nastaven na nulu)
- pe** - parity even (flag **PARITY/OVERFLOW** je nastaven na jedničku)
- p** - plus (flag **SIGN** je nastaven na nulu)
- m** - minus (flag **SIGN** je nastaven na jedničku)

Existující instrukce jsou vypsané v následující tabulce:

Absolutní podmíněné skoky	10 T-cyklů
jp nz, NN	
jp z, NN	
jp nc, NN	
jp c, NN	
jp po, NN	
jp pe, NN	
jp p, NN	
jp m, NN	

- - -

Mezi absolutní skoky patří také skoky, jejichž cílová adresa je zapsána v nějakém dvoiregistru. Tyto instrukce jsou nepodmíněné a pracují s dvoiregistrem **hl** a s oběma **index registry**. Velké využití absolutních skoků s cílovou adresou v dvoiregistru mají jedinečně použít při práci s rozeskokovými tabulkami.

Absolutní skok s adresou v	HL 4 T-cykly
jp	(hl)

Skoky s adresou v indexregistru lze použít v kombinaci s přístupem do tabulek - před rutinou mohou být uloženy parametry. Program naplní indexregistr adresou rutiny, do ostatních registrů přečte parametry a provede skok do rutiny. Použití takové konstrukce má smysl v případě, že máte několik podprogramů volaných z jednoho místa.

Absolutní skok s indexregistrem	8 T-cyklů
JP	(iX)
JP	(iY)

- - -

Relativní skoky - opět existují podmíněné a nepodmíněné. Ve srovnání s absolutními skoky jsou sice pomalejší ale také kratší. U relativních podmíněných skoků je rozdíl mezi časovým trváním v případě, že podmínka splněna je (**12 T-cyklů**), a v případě, že podmínka splněna není (**7 T-cyklů**). Podmíněných relativních skoků je méně než skoků absolutních, testovat lze pouze flagy **CARRY** a **ZERO**. Výhodou relativních skoku je, že se nemusí u relokovatelných programů přepočítávat jako skoky absolutní.

Relativní skok obsahuje 8-bitovou hodnotu, která je chápána jako číslo se znaménkem a přičítá se k hodnotě registru **PC**. V okamžiku přičtení relativního posunu ukazuje registr **PC** na následující instrukci. Relativní skoky mohou dosahovat na adresy **adr-126** až **adr+129**, kde **adr** je adresa relativního skoku. V asemblerech se místo relativního posunu píše přímo absolutní adresa a assembler si při překladu potřebný posun dopočítá sám. Pokud je vypočítaná hodnota mimo povolený rozsah, ohlásí překladač chybu.

Relativní skok nepodmíněný	12 (7) T-cyklů
Jr	E

Pro podmíněné relativní skoky máte k dispozici tyto podmínky, testují jen dva flagy ale budou Vám stačit prakticky vždy:

- nz** - non zero (flag **ZERO** je nastaven na nulu)
- z** - zero (flag **ZERO** je nastaven na jedničku)
- nc** - non carry (flag **CARRY** je nastaven na nulu)
- c** - carry (flag **CARRY** je nastaven na jedničku)

Jednotlivé instrukce tedy vypadají takto (první údaj o T-cyklech platí, když se skok provede, druhý platí v případě, že skok proveden není - program pokračuje následující instrukcí strojového kódu):

Relativní skoky podmíněné	12 (7) T-cyklů
Jr nz, E	
Jr z, E	
Jr nc, E	
Jr c, E	

- - -

Poslední instrukcí patřící mezi skokové instrukce je instrukce **djnz**. Tato instrukce je vlastně relativní skok, který se provede jen tehdy, když po odečtení jedničky od registru **b** nevychází nula - **decrement jump non zero**. Instrukce je určena pro programování cyklů, které mají proběhnout nejvýše 256krát. Počet průběhu je před započítáním cyklu nutno vložit do registru **b**, nezapomeňte však, že v těle cyklu se hodnota registru **b** nesmí měnit, potřebujete-li registr **b** uvnitř cyklu, použijte na začátku cyklu instrukci **push bc** a na konci pak **pop bc**.

Instrukce DJNZ	13 (8) T-cyklů
d j n z E	

Instrukce **djnz** se také používá jako zpomalovací smyčka všude, kde je potřeba - naplňte registr **b** požadovaným číslem a přidejte instrukci **WAIT djnz WAIT**.

* * *

Bitové manipulace tvoří nejpobčtbnější skupinu instrukcí. Umožní Vám nastavit, nulovat a testovat hodnotu libovolného bitu v registru nebo paměti.

Nejprve testování hodnoty registru - **bit**. Instrukce testuje hodnotu vybraného bitu a podle výsledku je nastaven **ZERO** flag (je-li bit nulový, platí **Z**, jinak **NZ**).

Pokud potřebujete testovat nastavení několika bitů najednou, vyplatí se použít raději logické operace. Přeneste hodnotu z daného registru do akumulátoru, pomocí instrukce **and N** vyberte požadované bity (**N** bude v požadovaných bitech obsahovat jedničky) a pomocí instrukce **cp N** porovnejte s požadovaným stavem (ve vybraných bitech testované hodnoty a všude jinde samé nuly).

Testování hodnoty jednotlivých bitů v registrech							8 T-cyklů
bit 0, b	bit 0, c	bit 0, d	bit 0, e	bit 0, h	bit 0, l	bit 0, a	
bit 1, b	bit 1, c	bit 1, d	bit 1, e	bit 1, h	bit 1, l	bit 1, a	
bit 2, b	bit 2, c	bit 2, d	bit 2, e	bit 2, h	bit 2, l	bit 2, a	
bit 3, b	bit 3, c	bit 3, d	bit 3, e	bit 3, h	bit 3, l	bit 3, a	
bit 4, b	bit 4, c	bit 4, d	bit 4, e	bit 4, h	bit 4, l	bit 4, a	
bit 5, b	bit 5, c	bit 5, d	bit 5, e	bit 5, h	bit 5, l	bit 5, a	
bit 6, b	bit 6, c	bit 6, d	bit 6, e	bit 6, h	bit 6, l	bit 6, a	
bit 7, b	bit 7, c	bit 7, d	bit 7, e	bit 7, h	bit 7, l	bit 7, a	

Testování hodnoty bitů v (HL)	12 T-cyklů
bit 0, (h l)	
bit 1, (h l)	
bit 2, (h l)	
bit 3, (h l)	
bit 4, (h l)	
bit 5, (h l)	
bit 6, (h l)	
bit 7, (h l)	

Testování hodnoty bitů v (IX+E), (IV+E)		20 T-cyklů
bit 0, (ix+E)		bit 0, (iy+E)
bit 1, (ix+E)		bit 1, (iy+E)
bit 2, (ix+E)		bit 2, (iy+E)
bit 3, (ix+E)		bit 3, (iy+E)
bit 4, (ix+E)		bit 4, (iy+E)
bit 5, (ix+E)		bit 5, (iy+E)
bit 6, (ix+E)		bit 6, (iy+E)
bit 7, (ix+E)		bit 7, (iy+E)

Instrukce **bit** nastavují **ZERO** flag podle hodnoty testovaného bitu, ostatní indikátory zůstávají po instrukci nezměněny.

- - -

Následuje instrukce **set**, která nastaví vybraný bit (zapiše do něj jedničku). Instrukce **set** nemá žádný vliv na indikátory.

Nastavení hodnoty jednotlivých bitů v registrech na 1							8 T-cyklů
set 0, b	set 0, c	set 0, d	set 0, e	set 0, h	set 0, l	set 0, a	
set 1, b	set 1, c	set 1, d	set 1, e	set 1, h	set 1, l	set 1, a	
set 2, b	set 2, c	set 2, d	set 2, e	set 2, h	set 2, l	set 2, a	
set 3, b	set 3, c	set 3, d	set 3, e	set 3, h	set 3, l	set 3, a	
set 4, b	set 4, c	set 4, d	set 4, e	set 4, h	set 4, l	set 4, a	
set 5, b	set 5, c	set 5, d	set 5, e	set 5, h	set 5, l	set 5, a	
set 6, b	set 6, c	set 6, d	set 6, e	set 6, h	set 6, l	set 6, a	
set 7, b	set 7, c	set 7, d	set 7, e	set 7, h	set 7, l	set 7, a	

Nastavení hodnoty bitů v (HL)	15 T-cyklů
	set 0, (hl)
	set 1, (hl)
	set 2, (hl)
	set 3, (hl)
	set 4, (hl)
	set 5, (hl)
	set 6, (hl)
	set 7, (hl)

Nastavení hodnoty bitů v (IX+E), (IV+E)		23 T-cyklů
set 0, (ix+E)		set 0, (iy+E)
set 1, (ix+E)		set 1, (iy+E)
set 2, (ix+E)		set 2, (iy+E)
set 3, (ix+E)		set 3, (iy+E)
set 4, (ix+E)		set 4, (iy+E)
set 5, (ix+E)		set 5, (iy+E)
set 6, (ix+E)		set 6, (iy+E)
set 7, (ix+E)		set 7, (iy+E)

Poslední instrukce - **res**, která vynuluje vybraný bit (zapiše do něj nulu). Instrukce **res** nemá žádný vliv na indikátory.

Nastavení hodnoty jednotlivých bitů v registrech na 1						8 T-cyklů
res 0, b	res 0, c	res 0, d	res 0, e	res 0, h	res 0, l	res 0, a
res 1, b	res 1, c	res 1, d	res 1, e	res 1, h	res 1, l	res 1, a
res 2, b	res 2, c	res 2, d	res 2, e	res 2, h	res 2, l	res 2, a
res 3, b	res 3, c	res 3, d	res 3, e	res 3, h	res 3, l	res 3, a
res 4, b	res 4, c	res 4, d	res 4, e	res 4, h	res 4, l	res 4, a
res 5, b	res 5, c	res 5, d	res 5, e	res 5, h	res 5, l	res 5, a
res 6, b	res 6, c	res 6, d	res 6, e	res 6, h	res 6, l	res 6, a
res 7, b	res 7, c	res 7, d	res 7, e	res 7, h	res 7, l	res 7, a

Nulování hodnoty bitů v (HL)	15 T-cyklů
	res 0, (hl)
	res 1, (hl)
	res 2, (hl)
	res 3, (hl)
	res 4, (hl)
	res 5, (hl)
	res 6, (hl)
	res 7, (hl)

Nulování hodnoty bitů v (IX+E), (IY+E)		23 T-cyklů
res 0, (ix+E)		res 0, (iy+E)
res 1, (ix+E)		res 1, (iy+E)
res 2, (ix+E)		res 2, (iy+E)
res 3, (ix+E)		res 3, (iy+E)
res 4, (ix+E)		res 4, (iy+E)
res 5, (ix+E)		res 5, (iy+E)
res 6, (ix+E)		res 6, (iy+E)
res 7, (ix+E)		res 7, (iy+E)

- - -

Naše exkurze do instrukčního souboru procesoru Z80 se blíží ke konci, zbývá nám poslední skupina instrukcí - vstupní a výstupní instrukce (pro komunikaci s periferiemi). Ve Spectru umožňují tyto instrukce měnit barvu **BORDERu** a vytvářet zvuk (rozšířené verze Spectra - Spectrum 128, +2, +3, Didaktik a další - používají instrukce ke stránkování rozšířené paměti - Z80 může adresovat najednou 64 KB paměti).

Pro komunikaci s periferiemi se používají **porty**. Ve Spectru je jich 65636 a každý má svoji adresu. Ve skutečnosti se však používá jen několik portů, které to jsou se dozvíte vždy na místě, kde je budeme potřebovat.

Mikroprocesor dokáže na vybraný port zapsat (instrukce **out**) obsah libovolného základního registru Z80 a naopak přečíst (**in**) hodnotu libovolného portu do registru.

Pro komunikaci s porty se nejčastěji používají instrukce pracující s registrem **a**, jsou nejkratší a nejrychlejší. Tyto instrukce nemají žádný vliv na indikátory.

Přečtení portu do akumulátoru	11 T-cyklů
in a, (N)	

Zápis obsahu akumulátoru na port	11 T-cyklů
out (N), a	

Adresa (N) je v tomto případě osmibitová a znamená spodní byte možné adresy portu. Chcete-li číst touto instrukcí port s šestnáctibitovou adresou, vložte před instrukci **in a,(N)** ještě instrukci **ld a,N**, kde do akumulátoru vložíte horní byte adresy portu. Totéž lze provést i u instrukce **out**, přitom ztratíte obsah akumulátoru, který má být na datovou sběrnici také poslán a nedocílíte žádaného výsledku, proto musíte port adresovat jinak (viz další instrukce). Zde by asi neškodilo menší objasnění toho, jak periferie komunikují: procesor na adresovou sběrnici zapíše adresu portu a u instrukce **out** také na datovou sběrnici přenášená data, adresový dekodér periferie si zjistí, zda je to "jeho" port a pokud ano, data si přečte (**out**) nebo je tam vloží (**in**). Některé periferie mají dekodér testující pouze spodní byte adresy a na horním bytu pak vůbec nezáleží - proto většinou stačí jen spodní byte.

Druhou možností je adresování portu pomocí registrového páru **bc**. Adresování párem registrů **bc** je značeno jako **(c)**. Na takto adresovaný port je možno zapsat (přečíst) obsahy všech základních registrů s výjimkou registru **f**. Použití těchto instrukcí **in** má vliv na všechny flagy s výjimkou **CARRY, P/V** flag ukazuje paritu.

Zápis a čtení portu s	adresou v BC	12 T-cyklů
out (c), b	in b, (c)	
out (c), c	in c, (c)	
out (c), d	in d, (c)	
out (c), e	in e, (c)	
out (c), h	in h, (c)	
out (c), l	in l, (c)	
out (c), a	in a, (c)	

- - -

Podobně jako u instrukcí **ld** existují i zde blokové instrukce vstupu a výstupu, také jsou s opakováním či bez něj.

Instrukce **ini** - přečte obsah z portu adresovaného **bc** a uloží jej do paměti na místo adresované registrem **hl**, zvětší obsah **hl** o jedničku a zmenší obsah **b** o jedničku. Pokud je po provedení v registru **b** nula, bude nastaven indikátor **ZERO**.

Instrukce **ind** - přečte obsah z portu adresovaného **bc** a uloží jej do paměti na místo adresované registrem **hl**, zmenší obsah **hl** o jedničku a zmenší obsah **b** o jedničku. Pokud je po provedení v registru **b** nula, bude nastaven indikátor **ZERO**.

Obě instrukce - **ini**, **ind** - mají také verze s automatickým opakováním - **inir**, **indr**. Obě běží tak dlouho, dokud není registr b vynulován. Instrukce s opakováním nastavují **ZERO** flag na hodnotu **1**, ostatní flagy nejsou definovány.

Instrukce INI a IND	16 T-cyklů
<code>ini</code>	
<code>ind</code>	

Instrukce INIR a INDR	21 (16) T-cyklů
<code>inir</code>	
<code>indr</code>	

Zmíněné instrukce (**ini**, **ind**, **inir**, **indr**) mají protějšky, které provádějí takřka totéž, pouze opačným směrem - přenášejí obsah z (**hl**) na (**cl**). Všechny ostatní vlastnosti jsou úplně stejné - jsou to instrukce **outi**, **outd**, **otir**, **otdr**.

Instrukce OUTI a OUTD	16 T-cyklů
<code>outi</code>	
<code>outd</code>	

Instrukce OTIR a OTDR	21 (16) T-cyklů
<code>otir</code>	
<code>otdr</code>	

Blokový vstup a výstup je určen pro ovládání disketové mechaniky a jinde se příliš moc neuplatní.

* * *

Po přečtení této kapitoly byste měli mít přehled o tom, co vlastně Z80 umí a co ne. Pokud mezi instrukcemi marně hledáte období některých příkazů BASICu jako **PRINT**, **INKEY\$**, **SAVE**, **LOAD**, **DRAW**, **PLOT**, **CLS**, ..., pak vězte, že všechny tyto operace je nutno buď naprogramovat nebo použít podprogramy z **ROM**. V dalším textu se dozvíte vždy obě možnosti - preferovat budeme programování bez využívání **ROM** protože se tak naučíte mnohem víc. Uvědomte si, že programování v assembleru je vlastně neustálé přenášení obsahu z jedné paměťové buňky do druhé vylepšené občas nějakou aritmetickou nebo logickou operací a teprve význam, který paměti určíte Vy nebo který má v hardware počítače určuje smysl operací.

Kdysi kdosi definoval počítač jako **pilného blbce, který třídí jedničky a nuly**. Tato definice má pravdu - inteligenci počítač získává až programem, který právě vykonává. Program je dílo člověka a může o svém tvůrci říci mnoho - například podle programů snadno rozlišíte lidi, jejichž první gramotnost má povážlivé mezery - hacky a carky neznají.

PÍŠEME ZNAKY

Způsob tisku znaků na každém počítači záleží od režimu, v jakém je paměť zobrazována. Obecně existují dva způsoby - textový a grafický. Textový režim je takový, kdy každý byte přímo odpovídá jednomu znaku na obrazovce - výhody textového režimu jsou v nízké náročnosti na paměť a rychlé práci s obrazovkou, nevýhody pak v nemožnosti měnit počet znaků na řádce, počet řádek na obrazovce a potíže při používání vlastních znaků (ne vždy to jde). V grafickém režimu znamená každý byte několik bodů na obrazovce - záleží to na způsobu práce s barvami - pro černobílé zobrazení je každý bod jeden bit. Výhodou grafického režimu je možnost zobrazit cokoliv v daném rastru, možnost kombinovat texty a grafiku, různé znakové soubory a měnit velikost písma. Nevýhodou je naopak větší náročnost na paměť a delší doba potřebná k práci s obrazovkou - přesunuje se větší úsek paměti.

Po obecném úvodu se podíváme, jak je tomu na Spectru. ZX má pouze jediný obrazovkový režim a to režim grafický. K dispozici máte **256 x 192** bodů, jsou sdruženy do skupin po **64**. Skupinu tvoří čtverec **8x8** bodů - každá skupina bodů může mít dvě barvy - podklad a inkoust. Body, nazývané též pixely, jsou uloženy v paměti od adresy **16384** do adresy **22527** (včetně) a zabírají tedy **6144 (=256*192/8)** bytů. Barvy pro všechny skupiny bodů jsou uloženy na adresách **22528 .. 23295** a zabírají **768 (=32*24)** bodů. Standardně je na Spectru každý znak napsán v rastru **8x8** bodů (po skupinách), což umožňuje, aby každý znak měl svou barvu papíru a inkoustu.

Rozložení pixelových bytů je na první pohled poněkud nesmyslné (ono není příliš smysluplné ani na další pohledy, nicméně je to skutečnost a nám nezbyvá než se s tím smířit). Pokud chcete získat představu, jak je obrazovka pokryta, napište a spusťte následující program v BASICu:

```
10 CLS
20 FOR i=16384 TO 22527
30 POKE i,255
40 PRINT #0; AT 0,0; i,
50 PAUSE 0
60 NEXT i
```

Program nejprve smaže obrazovku a pak bude zapisovat do jednotlivých pixelových bytů obrazovky číslo 255, výsledek bude nakreslení 8 bodů, vypíše adresu bytu a počká na stisk klávesy. Pokud Vám způsob zaplňování obrazovky připomíná nahrávání obrázku pak to není nic neobvyklého protože při nahrávání je obrázek ukládán do paměti ve stejné posloupnosti adres.

Pro rozložení atributů (to už je zcela přirozené) můžete použít tentýž, jen trochu modifikovaný, program:

```
10 LIST
20 FOR i=22528 TO 23295
30 POKE i,RND*255
40 PRINT #0; AT 0,0; i,
50 PAUSE 0
60 NEXT i
```

Do atributové paměti (paměť pro barevné informace) se nezapíše číslo 255 ale náhodná hodnota z rozsahu 0..255 - to pro větší barevnost, před přepisováním se provede listing a můžete se přesvědčit, že každý znak má svůj atribut. Každý byte atributové paměti nese v 7 bitu informaci o tom, jestli čtverec bliká, v 6 bitu informaci o jasu barev (společná pro inkoust i papír), v bitech 5,4,3 je barva papíru a 2,1,0 udávají inkoust.

Barvy (inkoust a papír) jsou uloženy ve třech bitech a je jich tedy celkově 8. Ke každému číslu mezi 0 a 7 je přiřazena barva a to takto:

- 0 ... černá (black)
- 1 ... tmavě modrá (blue)
- 2 ... červená (red)
- 3 ... fialová (magenta)
- 4 ... zelená (green)
- 5 ... světle modrá (cyan)
- 6 ... žlutá (yellow)
- 7 ... bílá (white)

Pro výpočet atributu můžete použít následující jednoduchý matematický vzorec:

$$\text{Atribut} = 128 * \text{blikání} + 64 * \text{jas} + 8 * \text{papír} + \text{inkoust}$$

Nyní k vlastnímu tisku znaku - v grafickém režimu znamená vypočítat adresu znakové předlohy, vypočítat adresu znaku v obrazovce a přenést požadovaný počet bytů a případně obarvit znak požadovaným atributem. Po vytištění znaku je obvykle potřeba zajistit podmínky pro tisk dalšího znaku (posunout tiskovou pozici na další).

Máme dvě možnosti, jak si s tímto problémem poradit - použít podprogramy v ROM nebo napsat program, který to umožňuje. Použití ROM je výhodné tím, že nepotřebuje žádný program (je už v ROM) a umožňuje poměrně mnoho funkcí, nevýhody jsou závislost na existenci systémových proměnných a nižší rychlost tisku.

Nejprve použití ROM. Tisk znaku je zajišťován instrukcí **rst 16** (volání podprogramu na adrese 16). Znak, který má být vytisknut, je uložen v registru **a**. Podprogram pro tisk znaku zachovává obsahy dvoiregistru **hl**, **de**, **bc**. Program umí tisknout všechny ASCII znaky, semigrafiku, UDG, klíčová slova a zpracovávat tyto řídicí kódy:

6 - print COMMA (posune na další pozici - začátek nebo polovina řádku)

8 - cursor left (posune tiskovou pozici doleva)

9 - cursor right (posune tiskovou pozici doprava)

10 - cursor down (posune tiskovou pozici dolů)

11 - cursor up (posune tiskovou pozici nahoru)

13 - ENTER (přesune tiskovou pozici na začátek dalšího řádku)

16 - ink (ovládání barvy inkoustu - pošlete kód **16** a potom číslo **0-7**)

17 - paper (ovládání barvy papíru - pošlete kód **17** a potom číslo **0-7**)

18 - flash (ovládání blikání - pošlete kód **18** a potom číslo **0** nebo **1**)

19 - bright (ovládání jasu - pošlete kód **19** a potom číslo **0** nebo **1**)

20 - inverse (ovládání inverze - pošlete kód **20** a potom číslo **0** nebo **1**)

21 - over (ovládání over - pošlete kód **21** a potom číslo **0** nebo **1**)

22 - at (nastavení tiskové pozice - pošlete kód **22** a potom **řádek** a **slopec**)

23 - tab (tabulátor - pošlete kód **23** a potom číslo rozložené do **dvou** bytů)

Před použitím **rst 16** je nutno nejprve otevřít příslušný tiskový kanál. Pro horní část obrazovky (obvyklý příkaz PRINT) se jedná o kanál **2**. Pro spodní část obrazovky (editační řádek) jde o kanál **0**. Otevření kanálu se provede zavoláním podprogramu na adrese **#1601** v ROM, číslo kanálu musí být v registru **a**.

```
ld    a, 2           ; číslo kanálu do registru a
call  #1601         ; zavolání podprogramu
```

Příklad: chcete vytisknout na 14 řádků ve 20 sloupci blikající červenou hvězdičku na žlutém podkladě s jasem. Můžete to udělat například takto:

```

ld a,2 ; číslo kanálu do registru A
call #1501 ; zavolání podprogramu

ld a,22 ; kód pro funkci AT
rst 16 ; odeslání do tiskového podprogramu
ld a,14 ; číslo řádku
rst 16
ld a,20 ; číslo sloupce
rst 16

ld a,16 ; kód pro nastavení inkoustu
rst 16
ld a,2 ; barva inkoustu
rst 16

ld a,17 ; kód pro nastavení papíru
rst 16
ld a,6 ; barva papíru
rst 16

ld a,18 ; kód pro nastavení blikání
rst 16
ld a,1 ; blikání zapnuto
rst 16

ld a,19 ; kód pro nastavení jasu
rst 16
ld a,1 ; vyšší jas
rst 16

ld a,"*" ; kód znaku hvězdička
rst 16

```

Se znalostí systémových proměnných to můžete docílit také mnohem kratším způsobem:

```

ld a,2 ; číslo kanálu do registru A
call #1601 ; zavolání podprogramu

ld a,242 ; atribut odpovídající žadaným barvám
ld (23695), a ; nastavení ATTR_T

ld a,32 ; kód pro funkci AT
rst 16 ; odeslání
ld a,14 ; číslo řádku
rst 16
ld a,20 ; číslo sloupce
rst 16

ld a,"*" ; kód znaku hvězdička
rst 16

```

Pokud tisknete více různých znaku nebo řídicích kódů, je lepší použít program pro tisk řetězce, který v tomto případě zkrátí program na třetinu (zbudou jen data), k tomu se však dostaneme až v další kapitole.

Na závěr použití tisku znaku pomocí ROM si ukážeme něco složitějšího:

```

ent      #                               ; místo pro spuštění

START    call #D6EB                       ; podprogram pro CLS
         ld  a,2                           ; číslo kanálu do registru A
         call #1601                        ; zavolání podprogramu

         ld  c,8                             ; osm řádků
         ld  b,8                             ; osm sloupců
LOOP     ld  a,22
LOOP2    rst 16
         ld  a,c
         rst 16
         ld  a,b
         rst 16
         } nastavení tiskové pozice

         ld  a,16
         rst 16
         ld  a,b
         dec a
         rst 16
         } nastavení barvy inkoustu

         ld  a,17
         rst 16
         ld  a,c
         dec a
         rst 16
         } nastavení barvy papíru

         ld  a,"@"
         rst 16
         ; vytiskni znak @

         djnz LOOP2                       ; vnitřní cyklus
         dec c                             ; zmenš C o jedničku
         jr  nz,LOOP                       ; vnější cyklus
         ret                               ; po skončení návrat

```

Polohu barevného obrazce na obrazovce můžete ovlivňovat tím, že do části pro nastavení tiskové pozice přidáte navíc přičtení vhodných konstant k obsahu registru **a** před tím, než zavoláte **rst 16** (za instrukce **ld a,c** a **ld a,b** přidejte **add a,N**).

Při používání **ROM** k tisku znaků můžete vhodně využít dvě adresy - 23606, kde je uložena adresa začátku znakového souboru zmenšená o 256 a 23675, kde je uložena adresa začátku UDG. Obě adresy jsou samozřejmě dvojbtytové.

- - -

Další část kapitoly už bude patřit kompletnímu tisku znaku se vším potřebným, z ROM využijeme jen grafické předlohy pro jednotlivé znaky. Ukážeme si nejprve základní tisk podobný tomu, co dokáže program v ROM. Od něj se bude lišit tím, že dokáže psát po celé ploše obrazovky, tím že když tisková pozice dosáhne pravý spodní roh, přesune se do levého horního rohu a tím, že nemá naprosto žádný vliv na atributy - přepisuje jen pixely.

```

ent      #                               ; vstupní bod do programu

START    ld  hl,16384                       ; tisková pozice je nastavena
         ld  (PRINTPOS),hl                 ; na levý horní roh obrazovky

```



```

LOOP      ld      a , 32                ; začínáme mezerou
          push   a f                    ; uschování obsahu akumulátoru
          call  CHAR1                  ; vytisknutí znaku v akumulátoru
          pop    a f                    ; obnovení obsahu akumulátoru
          inc   a                        ; posun na další znak
          cp    128                     ; pokud jsou ještě další znaky
          jr    c , LOOP                ; pokračuj v tisku, jinak skonči
          ret                             ; a vrať se zpět

CHARS     equ    15616-256             ; CHARS ukazuje na znakový soubor

CHAR1     push   a f                    ; uložení tisknutého znaku
          exx                               ; registry B, C, D, E, H, L jsou uloženy

          ld    l , a                    ; kód znaku do registru L
          ld    h , 0                     ; H je nulován, HL obsahuje kód znaku
          add   hl , hl
          add   hl , hl                   } registr HL je násoben osmi (1 znak)
          add   hl , hl
          ld    bc , CHARS                ; do BC adresa znaků zmenšená o 32*8
          add   hl , bc                   ; nyní ukazuje HL na předlohu znaku

          ld    de , (PRINTPOS)          ; do DE tisková pozice
          push  de                         ; uložení pro další použití

CHAR1A    ld    b , 8                     ; znak je uložen v 8 bytech
          ld    a , (hl)                  ; přesun znaku z adresy uložené v HL
          ld    (de) , a                  ; na adresu uloženou v DE
          inc   hl                         ; posun na další byte předlohy
          inc   d                           ; posun na další adresu v obrazovce
          djnz  CHAR1A                    ; zacyklení

          pop   de                         ; obnovení tiskové pozice
          inc   e                           ; posun v rámci třetiny obrazovky
          jr    nz , CHAR1B                ; pokud nepřekročí hranici je hotovo

          ld    a , d
          add   a , 8                       } korekce při přechodu mezi třetinami
          ld    d , a
          cp    88                          ; test opuštění obrazovky
          jr    c , CHAR1B                ; odskok je-li vše v pořádku
          ld    d , 84                       ; pokud ne, nastav levý horní roh

CHAR1B    ld    (PRINTPOS) , de           ; ulož novou tiskovou pozici

          exx                               ; obnov registry B, C, D, E, H, L
          pop   a f                         ; obnov akumulátor
          ret                             ; vrať se

PRINTPOS  defw  0                          ; sem je ukládána tisková pozice

```

Zde je na místě několik vysvětlení. Na adrese 15616 začíná v ROM znakový generátor (předlohy pro jednotlivé znaky), číslo 256 je odečteno proto, že mezera, tedy první znak, má kód nikoliv 0 ale 32 a 32*8=256. Bylo by možno na začátek tiskového podprogramu CHAR1 přidat instrukci `sub 32` (před `ld l,a`), ale program by se prodloužil jak časově, tak také prostorově. Další poznámku si zaslouží skutečnost, že registr `a` je ukládán v hlavním programu (smyčka) i v podprogramu pro tisk znaku - tady se projevil postup při psaní této ukázky. V podprogramu CHAR1 se registr `a` původně neukládá.

Později vznikla hlavní smyčka a potom vznikla potřeba zdvojit či vícenásobit volání podprogramu CHAR1 (kvůli vyzkoušení přechodu přes hranici třetiny) - po návratu však neobsahoval registr a kód znaku a bylo nutné jej obnovit. V tomto případě se tedy vyplatí uchovávat tisknutý znak a při návratu jej vrátit do a, většinou však tato potřeba není a kód tisknutého znaku se neuchovává - tady můžete jednu dvojici **push af** a **pop af** klidně odstranit a nic se nestane. Rozmyslete si, kde se odstranění více vyplatí. Navíc platí, že škodí mnohem více, když se něco neuloží vůbec, než když se to uloží vícekrát byť zbytečně. Ponecháte-li uložení v hlavní smyčce, můžete bez obav měnit jeden program pro tisk znaku za jiný bez toho, abyste museli zjišťovat, jestli obsah akumulátoru zachovává nebo ne. Ponecháte-li uložení v podprogramu, ušetříte uložení při každém volání, kdy je potřebné kód tisknutého znaku v registru ponechat.

Podprogram zachovává hodnoty ostatních obyčejných registrů - v tomto příkladě je to také zbytečné ale při práci se to téměř vždy vyplatí.

Uvedený příklad lze mírně modifikovat a tisknuté znaky se budou výrazně odlišovat od těch původních. Pro ilustraci předchozí věty tu jsou ještě dva tiskové podprogramy CHAR2 a CHAR3. Můžete je připsat k již napsanému textu a volat je místo nebo spolu s podprogramem CHAR1. Zde jsou výpisy, připište je na konec.

```

CHAR2      push  a f          ; uložení tisknutého znaku
           exx              ; registry jsou uloženy

           ld    l , a
           ld    h , 0
           add   hl , hl
           add   hl , hl
           add   hl , hl
           ld    bc , CHAR5
           add   hl , bc
           } nalezení znakové předlohy

           ld    de , (PRINTPOS) ; do DE tisková pozice
           push  de          ; uložení pro další použití

CHAR2A     ld    b , 8
           ld    a , (hl)
           rrca
           or    (hl)
           ld    (de) , a
           inc  hl
           inc  d
           djnz CHAR2A
           ; znak je uložen v 8 bytech
           ; přesun znaku s adresy uložené v HL
           ; rotace řádkem znaku doprava
           ; starý tvar spojen s novým - zdvojení
           ; na adresu uloženou v DE
           ; posun na další byte předlohy
           ; posun na další adresu v obrazovce
           ; zacyklení

           pop  de          ; obnovení tiskové pozice
           inc  e          ; posun v rámci třetiny obrazovky
           jr   nz , CHAR2B ; pokud nepřekročí hranici je hotovo
           ld  a , d
           add  a , 8
           ld  d , a
           cp  88
           jr  c , CHAR2B
           ld  d , 84
           } korekce při přechodu mezi třetinami

CHAR2B     ld    (PRINTPOS) , de ; ulož novou tiskovou pozici

           exx              ; obnov registry B, C, D, E, H, L
           pop  a f          ; obnov akumulátor
           ret              ; vrať se
    
```

A třetí podprogram pro tisk znaku (asi nejzajímavější - cyklus je v něm rozvinut).

```

CHARS    push  a f                ; uložení tisknutého znaku
          exx                    ; registry jsou uloženy

          ld   l, a
          ld   h, a
          add  hl, hl
          add  hl, hl
          add  hl, hl
          ld   bc, CHARS
          add  hl, bc
          } nalezení znakové předlohy

          ld   de, (PRINTPOS)     ; do DE tisková pozice
          push de                 ; uložení pro další použití

          ld   a, (hl)
          rrca
          ld   (de), a
          inc  hl
          inc  d
          } první řádek je posunut doprava

          ld   a, (hl)
          rrca
          ld   (de), a
          inc  hl
          inc  d
          } druhý řádek je posunut doprava

          ld   a, (hl)
          rrca
          ld   (de), a
          inc  hl
          inc  d
          } třetí řádek je posunut doprava

          ld   a, (hl)
          ld   (de), a
          inc  hl
          inc  d
          ld   a, (hl)
          ld   (de), a
          inc  hl
          inc  d
          } čtvrtý a pátý jsou beze změny

          ld   a, (hl)
          rlca
          ld   (de), a
          inc  hl
          inc  d
          ld   a, (hl)
          rlca
          ld   (de), a
          inc  hl
          inc  d
          ld   a, (hl)
          rlca
          ld   (de), a
          inc  hl
          inc  d
          } poslední řádky jsou posunuty doleva

```

```

        pop    de
        inc   e
        jr    nz ,CHAR3B
        ld    a ,d
        add  a ,8
        ld    d ,a
        cp   88
        jr    c ,CHAR3B
        ld    d ,84
CHAR3B  ld    (PRINTPOS) ,de

        exx
        pop  a f
        ret
    
```

} přesun na další tiskovou pozici

} obnovení registrů a návrat zpět

U všech tří podprogramů si můžete všimnout mnoha společných částí, v takovém případě se ze společných částí dělají podprogramy a v jednotlivých programech se vyskytují jen volání - zde by bylo vhodné udělat podprogram z nalezení znakové předlohy (začíná **ld l,a** a končí **add hl,bc**). Napadne Vás asi, že by se daly připojit také dvě instrukce před touto částí (**push af** a **exx**) a také dvě instrukce za touto částí (**ld de,(PRINTPOS)** a **push de**) - bohužel to nelze provést protože se zde pracuje se zásobníkem a ten je potřebný pro návratovou adresu - jako podprogram lze upravit takovou část programu, do které se vstupuje jen na začátku, vystupuje jen na konci (skoky pouze uvnitř) a která má při skončení zásobník ve stejném stavu jako na začátku - tyto podmínky lze občas porušit, pro začátek se jich raději držte. K uvedeně části by tedy bylo možno připojen jen zepředu instrukci **exx** a ze zadu pak **ld de,(PRINTPOS)**, není to vhodné z toho důvodu, že podprogram má být také logický celek - pokud chybí místo, je Vám dovoleno cokoliv. Tím, že vyrobíte z dostatečně dlouhých společných částí podprogramy, zkrátíte program ale také jej zpomalíte, jak moc, to záleží na tom, která část, programu je nahrazena - pokud je to uprostřed cyklu, je zpomalení větší protože dochází navíc k volání (**call**) a k návratu (**ret**) tolikrát, kolik průběhů cyklu program provede.

Druhá stejná část - konec - se dá také nahradit podprogramem, zde je výhodnější raději do podprogramu **CHAR1** přidat před **pop de** návěstí **CHAR1CM** a do ostatních dvou podprogramů vložit na stejné místo instrukci **jr CHAR1CM** a smazat zbytek. Program se tak znatelně zkrátí a neznatelně zpomalí.

Všechny tři podprogramy můžete při tisku navzájem střídat a psát některé části textu odlišně od ostatních (zvýraznění).

- - -

Dosud uvedené tiskové rutiny se nezatěžovaly nastavováním atributů pro znaky, ukážeme si, co je potřeba do podprogramu přidat. Za tímto účelem použijeme program **CHAR1** a upravíme jej přidáním části pro obsluhu atributů.

```

CHARC1  push  a f
        exx
        ; uložení
        ; registrů

        ld    l ,a
        ld    h ,0
        add  hl ,hl
        add  hl ,hl
        add  hl ,hl
        ld    bc ,CHARS
        add  hl ,bc
    
```

} nalezení znakové předlohy

```

ld de, (PRINTPOS) ; tisková pozice do DE
push de ; a na zásobník

CHARC1A ld b,8
ld a,(hl)
ld (de),a
inc hl
inc d
djnz CHARC1A
} tisk znaku - pixely

pop hl ; obnovení tiskové pozice
push hl ; opětne uložení

ld a,h ; posunutí začátku
sub 64 ; obrazovky do nuly
rrca
rrca
rrca
} rotace doprava (dělení osmi)
and 3 ; v A je nyní číslo třetiny
add a,88 ; posun na začátek atributů
ld h,a ; spodní byte (L) je stejný (neměníme)

ld a,r ; do A atribut (R dává "náhodné" číslo)
ld (hl),a ; nastavení daného atributového bytu

pop hl
inc l
jr nz,CHARC1B
ld a,h
add a,8
ld h,a
cp 88
jr c,CHARC1B
ld h,64
CHARC1B ld (PRINTPOS),hl
} přesun na další tiskovou pozici
(verze s HL je kratší a rychlejší)

exx
pop af
ret
} obnovení registrů a návrat zpět

```

Tímto jsme vyřešili problém barevného tisku - do všech tří podprogramů přidejte novou část a máte problém vyřešen.

Další problém je s nastavováním tiskové pozice - zatím ji lze nastavit jen přímo adresou a to není právě nejpohodlnější (většina her to však dělá právě tak). Vyrobit si tedy krátký program **ADRSET**, který dostane v registru **b** číslo řádku a v **c** číslo sloupce (nebo naopak, teď nevím, ale na to jistě přijdete brzy sami). Použijeme při tom jeden užitečný prográmeček z ROM.

```

ADRSET ld a,c ; znaková pozice (řádek nebo sloupec?)
add a,a
add a,a
add a,a
ld c,a
} je vynásobena 8 a převedena

ld a,b ; totéž se děje s druhým parametrem

```

```

add a , a
add a , a
add a , a
    } násobení

call #2B0 ; podprogram v ROM vrátí adresu bytu
ld (PRINTPOS) , hl ; v obrazovce, kde leží daný bod, ta se
ret ; zapíše na svoje místo a návrat

```

Pokud se dobře pamatují, mělo by v registru c být číslo sloupce, hlavu však za to na špalek nedám (mohl bych o ni přijít - jedná se tu o známý problém: ze dvou možností si člověk vždy napoprvé vybere tu špatnou - proto si raději sami vyzkoušejte, jaká je pravda). Pokusně to zjistíte tak, že do jednoho registru dáte třeba desítku, do druhého nulu a potom necháte něco napsat, podle toho, kde se znaky vypíšou, se dozvíte, kam co patří.

Náš program by také mohl umět nastavit jenom inkoust (papír, jas nebo blikání) beze změny ostatních. Něco takového jako dělají příkazy **INK**, **PAPER**, **BRIGHT** a **FLASH** v BASICu. Není nic jednoduššího, nahraďte instrukci **ld a,r** instrukcí **ld a,(ATRIBUT)** a připište tento podprogram.

```

SETINK ld c , %11111000 ; maska pro inkoust
jr STCOMMON ; skok do společné části

SETPAPER ld c , %11000111 ; maska pro papír
rlca
rlca
rlca
jr STCOMMON } posun barvy na bitovou pozici papíru

SETBRGHT ld c , %10111111 ; maska
rrca ; posun
rrca
jr STCOMMON

SETFLASH ld c , %01111111 ; maska
rrca ; posun

STCOMMON ld b , a
ld a , c
and b
ld b , a
ld a , (ATRIBUT)
and c ; ponech ostatní
or b ; přidej nové
ld (ATRIBUT) , a
ret

ATRIBUT defb 56 ; bílý papír, černý inkoust

```

Uvedeným podprogramem nastavujete barvy tak, že do **akumulátoru** vložíte požadované číslo (stejně jako v BASICu) a zavoláte návěští s odpovídajícím významem. Najednou můžete barvy nastavit samozřejmě i přímým zápisem na adresu **ATRIBUT**.

Další úprava, kterou je s tiskem možno provádět, je zvětšení výšky písmen a umožnění psát text na libovolnou pixelovou pozici - zatím jen vzhledem k souřadnici Y protože tatáž úprava i pro X je řádově složitější. U posunu o pixely nahoru a dolů stále stačí pracovat s celými byty, u posunu o pixely doleva nebo doprava je nutno používat rotace, logické instrukce a pracovat přímo s bity.

Pro vertikální posunování na obrazovce si ukážeme dva velice jednoduché ale přímo geniální podprogramy, které mnohokrát výhodně využijeme. Jedná se o rutiny **DOWNHL** a **UPHL**, které Vám vypočítají adresu bytu pod a bytu nad bytem, na který ukazuje dvoiregistr hl. Tyto nebo podobné podprogramy se vyskytují nejméně v polovina všech programů a takřka v každé grafické hře. Podívejte se na výpisy obou rutin.

```

DOWNHL      inc    h                ; posun v rámci textového řádku
            ld     a , h            ; je jednoduchý,
            and   7                ; není-li překročen,
            ret   nz               ; vrať se zpátky

            ld     a , l            ; přechod mezi textovými řádky
            add   a , 32            ; pokud při přičítání dojde k přetečení,
            ld   l , a             ; je to signál, že došlo k přechodu
            ld   a , h             ; mezi třetinami a pak je již vše
            jr   c , DOWNHL2       ; hotovo a tedy odskok

            sub   8                ; ještě zbývá úprava horního bytu
            ld   hl , a            ; při přechodu mezi textovými řádky

DOWNHL2     cp    88              ; na závěr test, jestli nedošlo
            ret   c                ; k opuštění obrazovky, návrat když ne
            ld   h , 64            ; pokud ano, nastav na začátek
            ret                    ; a vrať se také

```

Za podrobnější zmínku stojí chování rutiny **DOWNHL** v případě, že byte adresovaný registrem hl je v úplně nejspodnějším pixelovém řádku. Rutina vrací adresu bytu, který je ve stejném sloupci jako výchozí byte ale v nejvyšším pixelovém řádku. Obdobně, jenže naopak, pracuje také rutina **UPHL** - u bytu v nejvyšším pixelovém řádku vrací byte v řádku nejnižším.

```

UPHL        ld     a , h            ; přechod uvnitř textového řádku
            dec   h                ; spolu s testem,
            and   7                ; zda nejde o přechod mezi řádky
            ret   nz               ; když ne, tak se vrať

            ld     a , l            ; oprava při přechodu
            sub   32                ; mezi textovými řádky
            ld   l , a             ; a test přechodu
            ld   a , h             ; mezi třetinami
            jr   c , UPHL2         ; když ano, odskok

            add   a , 8             ; dokončení přechodu
            ld   hl , a            ; mezi textovými řádky

UPHL2       cp    64              ; test opuštění obrazovky
            ret   nc               ; pokud ne, vrať se
            ld   h , 87            ; pokud ano, oprav
            ret                    ; a vrať se také

```

Pro větší názornost si vyzkoušejte funkci obou programů. V následujícím programu po vyzkoušení zaměňte volání **DOWNHL** za **UPHL** a spusťte ještě jednou.

```

ent $

TEST    im    1                ; nastav přerušovací mód 1 - standardní
        ei                ; povol přerušování
        ld    hl, 190000      ; nějaká adresa uprostřed obrazovky
        ld    b, 192         ; na výšku má obrazovka 192 pixelů
LOOP    ld    (hl), 255       ; zaplní byte - 8 bodů
        halt              ; počkej na přerušování (50x za sekundu)
        call DOWNHL        ; posun na další byte
        djnz LOOP          ; znovu do cyklu
        ret

```

Nyní nějaké použití - tisková rutina, která má znaky vysoké 12 pixelů. Znaky jsou vyrobeny úpravou obyčejných znaků z ROM. Některé řádky vznikly složením dvou sousedních řádků. Rutina může tisknout na libovolnou pixelovou pozici Y.

```

ent $

START   ld    a, 32
LOOP    push  af
        call ZOUT
        pop  af
        inc  a
        cp   128
        jr  c, LOOP
        ret

ZOUT    exx
        push af
        add  a, a
        ld  l, a
        ld  h, 15
        add hl, hl
        add hl, hl
PPOS    ld  de, 16384
        push de

        ex  de, hl
        ld  (hl), a
        ld  bc, 2116
        call DOWNHL
        ld  a, (de)
        ld  hx, a
        ld  (hl), a
        inc de
        rl  c
        jr  nc, ZOUT3
        call DOWNHL
        ld  a, (de)
        or  hx
        ld  (hl), a
        djnz ZOUT2
        call DOWNHL
        ld  (hl), 0

```

testovací smyčka

uložení
registrů

nalezení znakové předlohy v ROM

číst se bude z (DE) a ukládat na (HL)
první řádek znaku je prázdný
B - počet řádků, C - zdvojené řádky
posun

uložení pro případné spojení
s dalším řádkem bodů

rotuj seznamem řádků, pokud
není v seznamu, odskoč
posun pro další řádek
přečti další ale neposunuj se
spoj s předchozím řádkem
zapiš řádek vzniklý ze svých sousedů
zacyklení

posun na poslední řádek
poslední je také prázdný


```

pop    hl                ; obnov tiskovou pozici
inc    l                ; posuň se
ld     a, l             ; a otestuj,
and    31               ; jestli nejsi na dalším řádku
jr     nz, ZOUT4       ; pokud ne, odskoč
dec    l                ; vrať se na výchozí pozici
ld     a, l             ; a vynuluj
and    %11100000       ; spodních pět bitů
ld     l, a             ; jsi na začátku starého řádku
ld     b, 12            ; mezirádková mezera je 12 bodů
ZOUT5  call DOWNHL     ; posuň se dolů
djnz  ZOUT5            ; a opakuj dvanáctkrát
ZOUT4  ld (POS+1), hl  ; ulož tiskovou pozici přímo do instrukce

pop    af
exx
ret
} obnov registry a vrať se

DOWNHL inc h
ld     a, h
and    7
ret    nz
ld     a, l
add    a, 32
ld     l, a
ld     a, h
jr     c, DOWNHL2
sub    8
ld     h, a
DOWNHL2 cp 88
ret    c
ld     h, 64
ret
} posun adresy na obrazovce o pixel dolů

```

Několik poznámek k uvedenému programu - první zajímavá část je výpočet adresy znaku v ROM - toto je nejkratší možný výpočet, lze aplikovat jen v případě, že horní byte adresy znakového souboru zmenšený o jedničku je dělitelný čtyřmi - to je to číslo patnáct, které je zdánlivě zcela bez kontextu ($(\text{int}(15616 / 256) - 1) / 4 = 15$). Druhá "podivnost" je skutečnost, že instrukce, která čte tiskovou pozici (**ld de, 16384**), nečte hodnotu z paměti ale používá tzv. přímý operand. Stačí si uvědomit, že tento program poběží v RAM a tedy číslo **16384** je možno přepsat, musíte však vědět, **kde** a **jak** je zapsáno. Na druhou otázku je jednoduchá odpověď, je zapsáno běžným způsobem ve dvou bytech - nejprve nižší a potom vyšší byte. Druhá otázka, **kde** je číslo uloženo, je trochu složitější - obvykle na adrese o jednu vyšší, než je adresa, kde instrukce začíná (první byte je operační kód), pokud však instrukce pracuje s indexovým registrem (nebo jeho polovinou), pak se uložení nalézá dva byty za adresou počátku instrukce. To platí pro přímé operandy v 95% případech, jsou však výjimky, nejlépe učiníte, když si zpočátku raději přeloženou instrukci prohlédnete jako číselný výpis paměti - jako přímý operand použijte 0, snadno pak zjistíte, kde je operand uložen. Uvedený způsob nelze použít při psaní programu pro ROM (to ale stejně hrozí tak jednomu procentu autorů), výhody jsou dvě - zkracuje se zdrojový text a přeložený kód, dochází k zrychlení programu. Dejte si dobrý pozor na to, aby se nepřepisovalo nic jiného než to, co se přepisovat má (hlavně u indexregistřů) - je to opět zdroj "nepochopitelných" chyb, kdy program někdy pracuje, někdy padá.

Instrukce **ld bc, 2116** plní registr **b** číslem **8** a registr **c** číslem **%01000100** (jednička znamená, že tento a následující řádek budou spojeny a vytvoří další řádek). Bylo by možno použít dvě instrukce a bylo by to přehlednější, takhle je to však kratší a rychlejší.

Za zmínku stojí také způsob posunu na další tiskový řádek v případě, že tisk dojde na konec. Pokud chcete řádky od sebe ještě více vzdálit, stačí místo **12** vložit větší číslo, můžete samozřejmě vložit i menší, pak se řádky budou překrývat, někdy se může hodit i to.

Protože tisk znaků je téma skutečně rozsáhlé a velice potřebné, ukážeme si ještě další dva programy, které umí něco nového. Jde o tisk, který je používán v některých hrách.

```

ent $

START      ld    bc, 12*256+5      ; nastavení pozice, takto je lépe
           call  ADRSET         ; vidět, co je v B (12) nebo C (5)

           ld    a, 32
LOOP       push  a f
           call  CHAR1
           pop   a f
           inc  a
           cp   128
           jr   c, LOOP
           ret

CHAR1      exx
           push  a f

           add  a, a
           ld   l, a
           ld   h, 15
           add  hl, hl
           add  hl, hl
           ex  de, hl
PPOS      ld   hl, 16384
           push  hl

CHAR1A     ld   b, 8
           ld   a, (de)
           ld   (hl), a
           call DOWNHL
           ld   a, (de)
           ld   (hl), a
           call DOWNHL
           inc  de
           djnz CHAR1A

           pop  hl
           inc  l
           ld  a, l
           and  31
           jr  nz, CHAR1B
           dec  l
           ld  a, l
           and  %11100000
           ld  l, a
           ld  b, 15

CHAR1D     call DOWNHL
           djnz CHAR1D
CHAR1B    ld   (PPOS+1), hl
    
```

testovací smyčka

uložení
registrů

nalezení znakové předlohy v ROM

každý řádek předlohy je zdvojen

posun na další tiskovou pozici

```

pop    a f
exx
ret
} obnov registry a vrať se

ADRSET  ld    a, c
        add  a, a
        add  a, a
        add  a, a
        ld   c, a
        ld   a, b
        call #22B0
        ld   (PPOS+1), hl
        ret

DOWNHL  inc   h
        ld   a, h
        and  7
        ret  nz
        ld   a, l
        add  a, 32
        ld   l, a
        ld   a, h
        jr   c, DOWNHL2
        sub  8
        ld   h, a
DOWNHL2 cp   88
        ret  c
        ld   h, 64
        ret
} posun adresy na obrazovce o pixel dolů

```

Poslední program je modifikací svého předchůdce, liší se tím, jak znaky vypadají a tím, že jsou obarveny. U tohoto tisku není možnost nastavovat tiskovou pozici na výšku po pixelech protože by neodpovídaly atributy. Pokud však použijete podprogram **ADRSET** z minulého programu, docílíte s barvami různé efekty - vyzkoušejte si je.

```

ent    $

START  ld    bc, 1*256+5      ; nastavení pozice
        call ADRSET

LOOP   ld    a, 32
        push af
        call CHAR1
        pop  af
        inc  a
        cp   128
        jr   c, LOOP
        ret

CHAR1  exx
        push af
        ; uložení
        ; registrů

```

PPOS	<pre> add a,a ld l,a ld h,15 add hl,hl add hl,hl ex de,hl ld hl,16384 push hl </pre>	} nalezení znakové předlohy v ROM
CHAR1A	<pre> ld b,8 ld a,(de) ld c,a rrca or c ld (hl),a call DOWNHL ld (hl),0 call DOWNHL inc de djnz CHAR1A </pre>	} tisk znaku
	<pre> pop hl push hl </pre>	; obnovení ; pozice
	<pre> ld a,h sub 64 rrca rrca rrca and 3 add a,88 ld h,a </pre>	} výpočet adresy atributů
	<pre> ld (hl),3+64 </pre>	; barva pro horní polovinu znaku
	<pre> ld bc,32 add hl,bc </pre>	; posun adresy ; na spodní atribut
	<pre> ld (hl),6 </pre>	; barva pro dolní polovinu znaku
CHAR1D	<pre> pop hl inc l ld a,l and 31 jr nz,CHAR1B dec l ld a,l and %11100000 ld l,a ld b,16 call DOWNHL djnz CHAR1D </pre>	} posun na další tiskovou pozici
CHAR1B	<pre> ld (PPOS+1),hl </pre>	
	<pre> pop af exx ret </pre>	} obnov registry a vrať se

```

ADRSET      ld    a,c
            add  a,a
            add  a,a
            add  a,a
            ld   c,a
            ld   a,b
            add  a,a
            add  a,a
            add  a,a
            add  a,a
            call #22B0
            ld   (PPOS+1),hl
            ret

DOWNHL      inc   h
            ld   a,h
            and  7
            ret  nz
            ld   a,l
            add  a,32
            ld   l,a
            ld   a,h
            jr   c,DOWNHL2
            sub  8
DOWNHL2     ld   h,a
            cp   88
            ret  c
            ld   h,64
            ret

```

rozsah C je 0-31, rozsah B je 0-11

posun adresy na obrazovce o pixel dolů

Tímto bychom mohli kapitolu o tisku znaků prozatím uzavřít. Neukázali jsme si zatím tisk velkých písmen (rastr 16x16 a větší) ani tisk na libovolnou pixelovou pozici (zatím jen po ose Y) a tím související proporcionální tisk. K tomu se vrátíme v některé z dalších kapitol.

VÝPIS TEXTŮ

Nyní už umíme vytisknout libovolný znak a to hned několika způsoby. Na řadě je vypsání textu - ukážeme si nejpoužívanější způsoby. Nejprve program:

```

ent    $

START   call  #D6B
        ld   a,2
        call #1E01
        }   smaž obrazovku a otevři kanál #2

        ld   hl,TEXT1
        call TEXTOUT
        ;   adresa prvního textu
        ;   vypsání

        call TEXTOUT2
        ;   druhý tiskový podprogram
        defb 22,5,5
        ;   data jsou uložena
        defm 'Text no.2'
        ;   za instrukcí CALL, jejich
        ;   poslední znak je invertován

```

	ld	a, 22	}	nastavení tiskové pozice
	rst	16		
	xor	a		
	rst	16		
	xor	a		
	rst	16		
	ld	a, 3		; číslo textu v tabulce
	call	TEXTOUT3		; najdi a vypiš
	call	TEXTOUT4		; obdoba předchozího způsobu
	defb	5		; parametr je za CALLeM
LOOP	ld	b, 5		; vypiš, celou tabulku (5 textů)
	ld	a, 22	}	nastavení tiskové pozice
	rst	16		
	ld	a, b		
	add	a, a		
	rst	16		
	ld	a, 25		
	sub	b		
	rst	16		
	ld	a, 17	}	nastavení barev
	rst	16		
	ld	a, b		
	rst	16		
	ld	a, 16		
	rst	16		
	ld	a, 9		
	rst	16		
	ld	a, b	}	vytiskni text
	dec	a		
	call	TEXTOUT3		
	djnz	LOOP		
	ret			; zacyklení
TEXTOUT4	pop	hl		; do HL adresu za instrukci CALL
	ld	a, (hl)		; přečti číslo textu
	inc	hl		; posuň adresu za parametr
	push	hl		; vrať adresu na zásobník
TEXTOUT3	ld	hl, TEXTS		; do HL adresu tabulky s texty
	or	a		; test na nulový text a případný
	jr	z, TEXTOUT		; odskok na vlastní tisk
TOUT3A	bit	7, (hl)		; test ukončovacího bitu
	inc	hl		; posuň na další znak (flagy se nemění)
	jr	z, TOUT3A		; ne jedná-li se o koncový znak, opakuji
	dec	a		; dekrementuj číslo textu
	jr	nz, TOUT3A		; a pokud to není hledaný text, opakuji
TEXTOUT	ld	a, (hl)		; přečti kód znaku
	and	127		; odstraň případný příznak konce textu
	rst	16		; vytiskni znak
	bit	7, (hl)		; test koncového příznaku
	inc	hl		; posuň se na další znak (nemění flagy)
	jr	z, TEXTOUT		; nešlo-li o poslední znak, jdi pro další
	ret			

```

TEXT1      defb 22,15,10      ; nastavení tiskové pozice
           defb 17,6         ; nastavení barvy papíru
           defm "Text no.1"  ; text
           defb 32,21,1     ; druhé nastavení tiskové pozice
           defb 17,4        ; barva papíru
           defb 19,1        ; vyšší jas
           defm "This is also" ; konec textu,
           defm ' text no.1' ; poslední znak je invertován

TEXTOUT2   pop  hl           ; odeber adresu textu
TOUT2A    ld  a, (hl)       ; přečti znak,
           and 127          ; odstraň případný koncový příznak
           rst 16           ; vypiš znak
           bit 7, (hl)      ; testuj konec znaku
           inc hl           ; posuň se na další znak
           jr  z,TOUT2A     ; ne jde-li o poslední znak, opakuj čtení
           jp  (hl)         ; skoč za text a pokračuj v programu

TEXTS     defm 'First'      ; tabulka textů,
           defm 'Second'    ; poslední znak
           defm 'Third'     ; každého textu
           defm 'Forth'     ; je invertován
           defm 'Fifth'     ; (7 bit je nastaven na jedničku)
           defm 'text'

```

Upozornění: tyto zdrojové texty byly odladěny na systému PROMETHEUS a pokud pracujete s jiným překladačem assembleru, musíte upravit řádky, na nichž se vyskytují texty v apostrofech takto:

```

defm 'text'      →      defm "tex"
                   defb "t"+128

```

U některých překladačů instrukce **defm** nemusí existovat, v takovém případě ji obvykle plně zastoupí instrukce **defb**.

U výpisu textů máme dva okruhy problémů - jak identifikovat požadovaný text a jak poznat jeho konec. Nejprve identifikaci zvoleného textu:

Text je možno jednoznačně identifikovat adresou prvního znaku. V zásadě existují dvě přenosové cesty - pomocí dvojregistru nebo pomocí zásobníku.

První situace - přenos dvojregistrem - je použit v podprogramu **TEXTOUT**. Výhoda spočívá v jednoduchosti vypisovacího programu (pouze cyklus až do konce textu). Nevýhodou je, že se přenáší zbytečně mnoho informací - textů je obvykle méně než 256, měl by tedy stačit jen jeden byte místo dvou, které se zde používají. Nevýhodnost této skutečnosti se projeví v případech, že se výpis textu volá z mnoha různých míst.

Druhá situace - přenos zásobníkem - využívá skutečnosti, že se při volání podprogramu ukládá na zásobník návratová adresa. Tato adresa by přece mohla být přímo adresou vypisovaného textu. Je však potřeba zařídit, aby se po vypsaní textu pokračovalo až za ním, zařídit posunutí návratové adresy za text. Tento způsob je naprogramován podprogramem **TEXTOUT2**, který si adresu textu odebere ze zásobníku do **hl**, vypíše text a na konci provede **jp (hl)**, což je ekvivalent pro **push hl** (ulož opravenou adresu zpátky na zásobník) a **ret** (vrať se zpět - odeber návratovou adresu a vlož ji do **PC** - **jp (hl)** bez meziuložení na zásobník). Výhoda je zřejmá - přenesení adresy textu se děje jako by mimochodem, nevýhodné je, že se na text nelze při použití téhož podprogramu pro tisk textu odvolávat odjinud. Přenos pomocí zásobníku se používá i pro libovolná jiná data - výhodné je, že veškeré redundantní (nadbytečné) informace jsou eliminovány (jako nadbytečnou informaci lze v případě přenosu registrem chápat operační kód instrukce **ld**, která plní registr - vyplácí se od okamžiku, kdy je počet volání větší než délka rozšíření rutiny).

Adresu textu můžete také přenést pomocí zásobníku takto:

```

    ....
    call TEXTOUT5          ; volání
    defw TEXT1            ; adresa textu
    ....

TEXTOUT5 pop hl          ; do HL adresy (?) adresy textu
          ld e, (hl)     ; přečti do E spodní byte adresy
          inc hl         ; posun
          ld d, (hl)     ; přečti do D horní byte adresy
          inc hl         ; posun
          push hl        ; vrať adresu na zásobník
          ex de, hl      ; přesuň adresu textu do HL
          jr TEXTOUT     ; skoč do vlastního tisku

```

Předvedené rozšíření tiskového podprogramu **TEXTOUT** se začíná vyplácet již při devíti voláních rutiny. Je-li to možné (v našem příkladě ne), můžete odstranit relativní skok a program připsat přímo před **TEXTOUT**, ušetří se další dva byty. Výhodou tohoto přenosu je, že můžete přenášet i další informace o textu - kam se má vypsat, jaké barvy ... - aniž byste byli omezováni počtem registrů.

Přenos pomocí adresy je možno modifikovat například tím, že se přenáší jen spodní byte adresy - to lze samozřejmě jen v případě, že horní byte adresy je pro všechny texty shodný a to je možné pokud celková délka všech textů nepřeroste **256-spodní byte tabulky textů**, z čehož plyne, že pro maximální možnou délku - 256 bytů - musí texty začínat na adrese, jejíž spodní byte je nula. Popisovaná úprava vyžaduje přehled o adresovém umístění textů, je možné brát adresu (její spodní byte) relativně k počátku tabulky s texty:

```

    ....
    call TEXTOUT6          ; volej tisk
    defb T1-T0            ; relativní adresa textu na adrese T1
    call TEXTOUT6          ; volej tisk
    defb T2-T0            ; relativní adresa textu na adrese T2
    ....

TEXTOUT6 pop hl          ; odeber adresu parametru
          ld c, (hl)     ; přečti relativní adresu textu
          inc hl         ; posun
          push hl        ; vrať návratovou adresu
          ld b, 0         ; vynuluj horní byte registru BC
          ld hl, T0       ; do HL adresu počátku textů
          add hl, bc      ; přičti relativní adresu
          jr TEXTOUT     ; a s adresou textu skoč na výpis

T0      defm 'text T0'   ; vlastní tabulka
T1      defm "text T1 & " ; text T1 pokračuje textem T2
T2      defm 'text T2'

```

To je vše o identifikaci textu adresou. Na rozdíl od následující identifikace číslem, umožňuje tisknout jeden text z různých míst, což se může hodit pokud je jeden text koncovou částí jiného (viz příklad).

Druhou možností, jak identifikovat text, je seřadit texty za sebou a očíslovat je. Toto se vyplácí v případě, že je textů méně než 255 a pro přenos čísla stačí jeden byte (kdyby byly potřeba dva byty, pak je lépe použít přímo adresu). Program pro tisk potom prohledá tabulku až nalezne požadovaný text a ten vypíše - rutiny **TEXTOUT3** a **TEXTOUT4**.

Od sebe se tyto dvě liší tím, že u druhé z nich se parametr uvádí jako **defb** přímo za voláním podprogramu, naproti tomu první dostane číslo textu přímo v akumulátoru. Protože nastavba pro přečtení parametru za instrukci call je dlouhá přesně čtyři byty, můžeme snadno zjistit, že se složitější program (**TEXTOUT4**) vyplácí už při pěti voláních.

U identifikace textu číslem je nepříjemné, že u většího počtu textu trvá prohledávání tabulky nějakou dobu - vzhledem k rychlosti strojového kódu to obvykle nevádí.

Dostáváme se k druhému problému - jak poznat konec textu. Všechny zatím uvedené programy používají pro ukončení textu invertovaný znak. Výhoda této varianty je v tom, že text není delší než musí být, nevýhodou je skutečnost, že můžete použít je 128 různých znaků - obvykle bohatě stačí, nemusí však vždy. Takto je například vyrobena tabulka klíčových slov a chybových hlášení v ROM Spectra.

Druhou možností je použití speciálního ukončovacího kódu - například 0, která se dobře testuje. Výhodou je možnost použití ostatních 254 znaků, nevýhodou pak nezbytné prodloužení původního textu.

Třetí možnost je uvést před každým textem jeho délku. Výhodou je rychlejší prohledávací program u identifikace textu číslem, může vypadat třeba takhle,

```

TEXTOUT7  ld  hl,TEXTY          ; první text
           or  a                 ; a jedná-li se o něj
           jr  z,TOUT7A        ; odskoč
           ld  b,0              ; do BC
TOUT7B    ld  c,(hl)           ; délku textu
           inc hl                ; plus jedna za číslo
           add hl,bc            ; přičti
           dec a                 ; test nalezení
           jr  nz,TOUT7B       ; a znovu, když ne
TOUT7A    ld  b,(hl)           ; do B délku textu
TOUT7C    inc hl                ; posun na další znak
           ld  a,(hl)           ; výpis
           rst 16                ; znaku
           djnz TOUT7C         ; opaku j B krát
           ret

TEXTY     defb L1-L0           ; délka textu "franta"
L0        defm "franta"       ; text
L1        defb L3-L2         ; obdobně další
L2        defm "pepa"
L3        defb L5-L4
L4        defm "alouis"
L5
    
```

Nakonec si ukážeme složitější program pro tisk textů, který se hodí třeba pro programování textovek - umožňuje více operací. Tisková rutina bude umět tyto akce s různými kódy:

```

32...127 - obvyklý tisk písmene
0...7 - nastavení barvy inkoustu
8...15 - barva papíru (-8)
16,17 - jas (ON, OFF)
18,19 - kurzíva
20,21 - tučný tisk
128..254 - text z tabulky
255 - konec textu
    
```

Následuje vlastní výpis programu - při opisování budete potřebovat mnoho trpělivosti, je totiž zatím nejdelší. Obzvláště pevné nervy Vám přeji při opisování znakového souboru na konci výpisu.

```

ent $

S      ld  a,4           ;nastav
      out (254),a      ;zelený border
      ld  a,15         ;text č. 15
      call texts      ;vytiskni jeJ
      ret

TEXTABLE defm "la "    ;text 0 (128)
      defb -1         ;koncová značka

      defm " jsem"    ;text 1 (129)
      defb -1

      defm "sta"     ;text 2 (130)
      defb -1

      defm "e "      ;text 3 (131)
      defb -1

      defm "kU"      ;text 4 (132)
      defb -1

      defm "kr#sn'"  ;(apostrof) text 5 (133)
      defb 22,-1

      defm "len"     ;text 6 (134)
      defb 22,-1

      defm " hodiny" ;text 7 (135)
      defb -1

      defm " pohovořil" ;text 8 (136)
      defb 22,-1

      defm " dcer"   ;text 9 (137)
      defb 132,22,-1

      defm "svatba"  ;text 10 (138)
      defb 22,-1

      defm "na stat" ;text 11 (139)
      defb 132,22,-1

      defm " Na_"    ;(podtržítka) text 12 (140)
      defb 131,130
      defm " r'"     ;(apostrof)
      defb 135,22,-1

      defm "Bij[ $tyfi" ;text 13 (141)
      defb 135,22,-1

      defb 140,141,22,-1 ;text 14 (142)

```

```
def b 18 ; text 15-hlavní text (nadpis)
def m "J#ra da Cimrman"
def b 22,22,19,20
def m "Uhl'čtsk' " ; (apostrof)
def m "Janovice"
def b 21,22,22
def m "Miloval" ; první odstavec
def b 129
def m "d@v#"
def b 131,133
def m "M@"
def b 128
def m "o#i tuz"
def b 131,133
def m "Ulasý m@"
def b 128
def m "jako "
def b 134
def m "Na p@l#"
def b 132
def m " ple"
def b 128,134,142

def m "Sotva" ; druhý odstavec
def b 129
def m "s n["
def b 136
def m "ček"
def b 128
def m "abych"
def b 136
def m "Zda mi "
def b 130
def m "čl daj["
def b 137
def m "Ji'" ; (vlnovka)
def b 129
def m "vid@l ven"
def b 132,137,142

def m "A tak dohod" ; třetí odstavec
def b 128,"s",131,138
def m "A brzí#ko by"
def b 128,138
def m "Bydli m"
def b 131
def m "zde"
def b 139
def m "Rodi#"
def b 131
def m "t'" ; (apostrof a vlnovka)
def b 139,142,-1
```

; následující část provádí vyhledání textu v tabulce, tato část zajišťuje rekurzivní volání – texty tak mohou být složeny z většího množství úrovní

```

TEXTS    push  hl           ; uložení registru HL
         push  bc           ; totéž s registrem BC
         ld    hl, TEXTABLE ; adresa tabulky textů
         or    a            ; nulový text už je nalezen
         jr    z, TEXTS2    ; vytiskni je j
TEXTS3   ld    c, (hl)     ; přečti znak do C
         inc  hl           ; posuň se dál
         inc  c            ; testuj konec textu
         jr    nz, TEXTS3   ; a opakuj dokud je j nenalezněš
         dec  a            ; zjisti, jestli se jedná o žádaný text
         jr    nz, TEXTS3   ; když ne tak hledej dál

```

; nyní přichází na řadu vlastní vytištění nalezeného textu

```

TEXTS2   ld    a, (hl)    ; přečti znak
         inc  hl           ; posuň se na další
         cp   -1          ; testuj konec textu
         jr    z, TEXTSEND ; případně odskoč
         call CHAR        ; vytiskni znak
         jr    TEXTS2     ; a jdi pro další
TEXTSEND pop  bc         ; obnov registry BC a HL
         pop  hl         ; pozor!- zde nelze použít EXX
         ret

```

; vstupní bod do tisku znaku, zde jsou obslouženy všechny kódy kromě ukončovacích

```

CHAR     bit   7, a       ; testuj textové kódy
         res  7, a       ; vymaž sedmý bit
         jr   nz, TEXTS  ; s textovými kódy do vyhledání a tisku
         cp   32         ; testuj řídicí kódy
         jr   c, CONTROLS ; a odskoč do jejich zpracování

         exx           ; přehod registry
         ld    l, a
         ld    h, 0
         add  hl, hl
         add  hl, hl
         add  hl, hl
         ld   bc, CHARS-256
         add  hl, bc
         } tradiční výpočet znakové předlohy

PPOS     ld    de, 16384  ; adresa tiskové pozice
         push de         ; ulož pro pozdější použití

RRC A1   call  GET2      ; přečti znakový řádek
RRC A2   nop           ; sem se zapisuje kód RRC A u kurzívy
         call  GET      ; zapiš do obrazovky a přečti další řádek
RRC A3   nop           ; první tři řádky jsou při tisku kurzívou
         call  GET      ; posunuty doprava, jinak beze změny
RRC A4   nop           ; program tedy sám modifikuje
         call  GET      ; kód podle potřeby

```

```

call GET ; prostřední dva řádky
call GET ; jsou vždy bere změny

RLCA1 nop ; poslední tři řádky
call GET ; se podle potřeby
RLCA2 nop ; posouvají
call GET ; doleva
RLCA3 nop ; poslední řádek je zapsán
ld (de),a ; přímo bez čtení dalšího

pop hl ; nyní budeme zpracovávat barvy
push hl ; obnov adresu v HL, nech i na zásobníku

ld a,h ; přepočítání adresy
add a,10 ; z pixelů do atributů
CHAR2 cp 88 ; je možné provádět
jr nc,CHAR1 ; několika způsoby,
add a,7 ; tenhle je delší než
jr CHAR2 ; ty dříve uvedené

CHAR1 ld h,a ; a proto jej nebudeme používat
COLOR ld (hl),56 ; zapiš atribut

CHAR9 pop hl
inc l
jr nz,CHAR3
ld a,h
add a,8
ld h,a
cp 88
jr c,CHAR3
CHAR3 ld h,64
ld (PPOS+1),hl

CHAREND exx ; přehod registry a vrať se
ret

```

} posun na další tiskovou pozici

; zde se budeme zabývat všemi řídicími kódy

```

CONTROLS cp 18 ; kód pro kurzívu
jr nz,CONTR2 ; odskoč na další kódy
defb 1 ; kód instrukce LD BC,NN
rlca ; do C jde kód instrukce RLCA
rrca ; do B jde kód instrukce RRCA
COMMON ld a,b ; do A buď RRCA nebo NOP
ld (RRCA1),a
ld (RRCA2),a
ld (RRCA3),a
ld a,c ; do A buď RLCA nebo NOP
ld (RLCA1),a
ld (RLCA2),a
ld (RLCA3),a
ret ; zapiš a vrať se

CONTR2 cp 19 ; kód vypnutí kurzívy
jr nz,CONTR3 ; odskoč když ne
ld bc,0 ; do B a C instrukce NOP
jr COMMON ; skoč do zápisu, který je společný

```

} zapiš všude, kam patří

} zapiš a vrať se

CONTR3	cp 20 jr nz, CONTR4 defb 62 rrca	; kód pro tučně písmo ; odskoč ; kód instrukce LD A,N ; do A kde kód instrukce RRCA
COMMON2	ld (BOLD), a ret	; zapiš do čtení znakového řádku
CONTR4	cp 21 jr nz, CONTR5B xor a jr COMMON2	; kód pro "odtučnění" ; odskoč ; do A kód instrukce NOP ; skoč do společného zápisu
CONTR5B	cp 22 jr nz, CONTR5 exx ld hl, (PPOS+1) ld a, l or %00011111 ld l, a jr CHAR9	; kód pro odřádkování ; odskoč ; přehoď registry ; přečti tiskovou pozici ; a uprav ji tak, ; jako by šlo ; poslední znak na řádku ; a skoč do posunu na další znak
CONTR5	push hl ld hl, COLOR+1 cp 17 jr nz, CONTR6 res 6, (hl) pop hl ret	; ulož HL, budeme je potřebovat volný ; do HL adresa atributu ; test kódu pro "odjasnění" ; odskoč ; vypni jas ; obnov HL a vrať se
CONTR6	cp 16 jr c, CONTR7 jr nz, POPHLRET set 6, (hl)	; kód pro "zjasnění" ; odskoč s kódy pro INKOUST a PAPÍR ; vrať se - nevyužitě kódy ; zapni jas
POPHLRET	pop hl ret	; obnov HL a vrať se
CONTR7	sub 8 jr c, CONTR8 add a, a add a, a add a, a ld c, a ld a, (hl) and %11000111 or c ld (hl), a pop hl ret	; zde se oddělí PAPÍR a INKOUST ; jedná se o INKOUST } násobení 8 ; vlož výsledek do C ; do A původní barva ; ponech ostatní části atributu ; připoj nový papír ; zapiš zpět ; obnov HL a vrať se
CONTR8	add a, 8 ld c, a ld a, (hl) and %11111000 jr COMMON3	; přičti zpět ; vlož do C ; původní atribut ; ponechej ostatní ; skoč do společné části

```

GET      ld      (de) , a      ; zapiš znakový řádek do obrazovky
         inc      d        ; a posuň se na další řádek
GET2     ld      a , (hl)   ; přečti znakový řádek
BOLD     nop                     ; zde se případně provede posun doprava
         or      (hl)      ; a připojí původní tvar – ztučnění
         inc     hl        ; posun na další část předlohy a návrat
         ret
    
```

; poslední částí je znakový soubor s češtinou. Jeho opisování Vám upřímně nezávidím

```

CHARS   de fw  0,0,0,0,4096,4112 ; právě začínáme
         de fw 16,16,9216,36,0,0 ; per aspera ad astra
         de fw 4104,1080,17468
         de fw 60,4136,17464
         de fw 17472,56,25088
         de fw 2148,9744,70,4096
         de fw 4136,17450,58
         de fw 4104,17464,16504

         de fw 56,2048,4112,4112 ; už máte za sebou 6%
         de fw 8,8192,4112,4112 ; jen tak dál
         de fw 32,0,4136,4220,40
         de fw 0,4112,4220,16,0
         de fw 0,2048,4104,0,0
         de fw 124,0,0,0,6144,24
         de fw 0,2052,8208,64
         de fw 14336,21580,25684

         de fw 56,12288,4176 ; 13% – výborné
         de fw 4112,124,14336 ; už zbývá Jen 87%
         de fw 1092,16440,124
         de fw 14336,2116,17412
         de fw 56,2048,10264
         de fw 31816,8,32256
         de fw 31808,16898,60
         de fw 14336,30784,17476

         de fw 56,31744,2052 ; 20% – jedna pětina
         de fw 4104,16,14336 ; nepolevu,jte
         de fw 14404,17476,56
         de fw 14336,17476,1084
         de fw 56,0,4096,8208,0
         de fw 16,4096,8208,0
         de fw 4104,4128,8,0
         de fw 31744,31744,0,0

         de fw 2064,2052,16 ; 26% – už více než čtvrtina
         de fw 14336,2116,16,16 ; že se na to .....
         de fw 4136,17464,16508
         de fw 56,14336,17476
         de fw 17532,68,30720
         de fw 30788,17476,120
         de fw 14336,16452,17472
         de fw 56,28672,17480
    
```

de f w 18500, 112, 31744 ; **33%** - jedna třetina
de f w 30784, 16448, 124 ; ještě Vás to baví ?
de f w 31744, 30784, 16448
de f w 64, 14336, 16452
de f w 17500, 56, 17408
de f w 31812, 17476, 68
de f w 31744, 4112, 4112
de f w 124, 1024, 1028

de f w 17476, 56, 18432 ; **40%** - brzy budete v polovině
de f w 24656, 18512, 68 ; zítra je také den
de f w 16384, 16448, 16448
de f w 124, 17408, 21612
de f w 17476, 68, 17408
de f w 21604, 17484, 68
de f w 14336, 17476, 17476
de f w 56, 30720, 17476

de f w 16504, 64, 14336 ; **46 %** - polovina je už na dosah
de f w 17476, 18516, 52 ; kolik už jste udělali chyb ?
de f w 30720, 17476, 18552
de f w 68, 14336, 14400
de f w 17412, 56, 31744
de f w 4112, 4112, 16
de f w 17408, 17476, 17476
de f w 56, 17408, 10308

de f w 4136, 16, 17408 ; **53%** - jste za polovinou
de f w 17476, 21588, 40 ; nedělejte si z toho nic, já jsem
de f w 17408, 4136, 10256 ; to musel psát také
de f w 68, 17408, 10308
de f w 4112, 16, 31744
de f w 2052, 8208, 124
de f w 4104, 12288, 4112
de f w 56, 4136, 17528

de f w 17476, 68, 4104 ; **60%** - běžte se na chvilku projít
de f w 17464, 17476, 56 ; nebo z toho dočista zblbnete
de f w 2068, 8220, 8224, 32
de f w 4136, 16440, 1080
de f w 120, 10260, 8304
de f w 10272, 16, 0, 1080
de f w 17468, 60, 16384
de f w 30784, 17476, 120, 0

de f w 17464, 17472, 56 ; **66%** - jaká byla procházka ?
de f w 1024, 15364, 17476 ; s chutí do práce, zazpívejte si:
de f w 60, 0, 17464, 16504 ; Vyhrnem si rukávy ...
de f w 60, 3072, 6160, 4112
de f w 16, 0, 17468, 15428
de f w 14340, 16384, 30784
de f w 17476, 68, 4096
de f w 12288, 4112, 56

de f w 1024 , 1024 , 1028	; 73 % - zpíváte ještě?
de f w 6180 , 8192 , 12328	; raději toho nechte
de f w 10288 , 36 , 8192	
de f w 8224 , 8224 , 24 , 0	
de f w 21508 , 21588 , 84 , 0	
de f w 17528 , 17476 , 68 , 0	
de f w 17464 , 17476 , 56 , 0	
de f w 17528 , 30788 , 16448	
de f w 0 , 17468 , 15428	; 80% - zbývá už jen pětina
de f w 1540 , 0 , 8220 , 8224	; navrhuje tykání
de f w 32 , 0 , 16440 , 1080	
de f w 120 , 4096 , 4152	
de f w 4112 , 12 , 0 , 17476	
de f w 17476 , 56 , 0 , 17476	
de f w 10280 , 16 , 0 , 21572	
de f w 21588 , 40 , 0 , 10308	
de f w 10256 , 68 , 0 , 17476	; 86% - teď už to nemůžeš vzdát!
de f w 15428 , 14340 , 0	; A co jinak?
de f w 2172 , 8208 , 124	
de f w 4104 , 17476 , 17476	
de f w 56 , 10256 , 17492	
de f w 17476 , 56 , 4104	
de f w 17476 , 15428 , 14340	
de f w 4136 , 2172 , 8208	
de f w 124 , 16956 , 41369	; 93% - výborně!
de f w 39329 , 15426	; gratuluji k dosažení cíle

V uvedeném programu je několik nových zajímavostí, postupně si je všechny probereme a vysvětlíme některé méně jasné detaily.

První novinkou je instrukce **out (254).a**. Tato instrukce nastavuje barvu borderu - na portu **254** jsou to spodní tři byty - to je také důvod, proč nelze na borderu nastavit jas, není na to bit. Čtvrtý a pátý bit se používají pro komunikaci s magnetofonem, ostatní bity nejsou nijak využité.

Druhou novinkou je poněkud divoká tabulka, ve které se vyskytují všelijaké kódy a speciální znaky na místě písmen - na místě těchto znaků jsou v použitém znakovém souboru uložena česká písmena. Tabulka obsahuje komprimovaný text, který se dozvíte po spuštění.

Třetí zajímavost spočívá v tom, že z podprogramu **CHAR** se občas vstupuje do **TEXTS** a odtud je opět volán podprogram **CHARS** - tomuto (tedy skutečnosti, že podprogram volá sám sebe přímo nebo přes další podprogramy) se říká rekurze. Jde o velice silný programovací prostředek. Ve strojovém kódu se příliš nepoužívá, mnohem více ve vyšších programovacích jazycích. Zde je použita proto, že každý text se může skládat nejen ze znaků ale i z dalších textů a ty se mohou opět skládat nejen ze znaků ale také z textů a tak dále. Takto tvořené texty mají tu výhodu, že často opakovaná slova nemusí zabírat mnoho prostoru - tento způsob používají všechny lepší cizí textovky a proto mají obvykle mnohem větší rozsah a bohatější popisy lokací. U domácích her jsem něco podobného zatím neviděl, jsou totiž obvykle psány v BASICu - jen občas se některé časté slovo uloží do řetězcové proměnné a ta se používá místo něj - nelze však udělat více než jednu úroveň vnoření.

Pro zajímavost jak je tento způsob ukládání u velkých textů účinný naznačuje hra **SHERLOCK**, ve které je podle celkem věrohodných zpráv 0.5 MB textů a navíc ještě obrázky, a to vše se vejde do 40 KB.

Hry používají také další způsoby komprese - vynechávají mezery mezi slovy a texty upravují tak, že každé slovo začíná velkým písmenem, podle toho poznají, kdy má přijít mezera a program její vytištění už zajišťuje sám. Další občas použitý způsob spočívá v tom, že znaky nejsou ukládány do 8 bitů (256 možných znaků) ale do 5 (32 znaků), 6 (64 znaků) nebo 7 bitů (128 znaků).

Uvedené způsoby komprese textů mají také tu výhodu, že se Vám v textech nebude nikdo vrtat protože by se z toho spíš zbláznil. Nevýhodou je, že cizí hry jsou prakticky nepřeložitelné - pokud je chcete hrát, a stojí to za pokus, nezbyde Vám než se naučit anglicky (to se ovšem nikdy neztratí).

Čtvrtá zajímavost je vlastní tisk znaku - podprogramy **GET**. Tato úprava je zvolena proto, že je kratší - šlo by to ještě lépe - stačí použít instrukci **djnz** - zkuste to ještě zlepšit. Tříkrát se tam opakuje sekvence **RRCA, nop, call GET** a **call GET, RLCA, nop**. Navíc by se odstranilo trojnásobné přepisování za návěštím **COMMON**, úpravu si vyzkoušejte sami.

Sami si zkuste přidat ještě nastavení tiskové pozice - obdobu **AT** a případně další úpravy. Pokud budete přidávat kódy s parametry, musíte vylepšit začátek podprogramu **CHAR** třeba takto:

```

CHAR      push  a f          ; uložení hodnoty akumulátoru
STATUS    ld    a , 0       ; přečti si stav od minula
          cp    1           ; pokud očekáváš první parametr
          jr    z , PARAM1   ; zpracuj je j
          cp    2           ; pokud očekáváš druhý parametr
          jr    z , PARAM2   ; zpracuj je j
          ....

          ....
          pop   a f          ;
          bit   7 , a        ; zda už program
          res  7 , a        ; pokračuje zcela obvykle
          ....

PARAM1    ld    a , 2       ; po prvním parametru
          ld    (STATUS+1) , a ; je očekáván ještě druhý
          pop   a f          ; obnov hodnotu parametru
          ....             ; pokračuj ve zpracování

PARAM2    xor   a           ; po druhém parametru
          ld    (STATUS+1) , a ; už mohou přijít obvyklé znaky
          pop   a f          ; obnov parametr
          ....             ; a zpracuj ho

          ....
          ld    a , 1       ; v části pro zpracování
          ld    (STATUS) , a ; kódu s parametry musí
          ....             ; být zaznamenáno,
                          ; že se očekávají parametry
    
```

Vytvoření takového programu ponechám na Vás, můžete použít **ADRSET** z nějaké vhodné předchozí tiskové rutiny. Dejte si pozor, aby možné parametry nekolidovaly se znakem pro ukončení textu!

Zkuste napsat svoji vlastní obdobu **RST 16** - můžete přidat i další kódy - smazání obrazovky, čekání na stisk klávesy, pípnutí, zavolání určeného podprogramu, změnu barvy borderu, nakreslení kurzoru za tiskovou pozici a další.

VÝPIS ČÍSEL

Vedle textů je samozřejmě občas potřeba vypsat obsah nějakého registru - a právě o tom je tato kapitola. Naučíte se vypisovat číslo v nejběžnějších soustavách.

Začneme neirozšířenější z číselných soustav - soustavou desítkovou. Protože však program pro soustavu šestnáctkovou je obdobný, spojíme je do jednoho programu najednou.

```

ent #

START   ld  a,2           ; otevření kanálu
        call #1601     ; pro tisk

        ld  hl,12345   ; číslo 12345
        call DECIMAL5  ; vytiskni jako pětimístné
        ld  a,13       ; odřádkuj
        rst 16

        ld  hl,1234    ; číslo 1234
        call DECIMAL4  ; vytiskni jako čtyřmístné
        ld  a,13       ; odřádkuj
        rst 16

        ld  hl,123     ; číslo 123
        call DECIMAL3  ; vytiskni jako třímístné
        ld  a,13       ; odřádkuj
        rst 16

        ld  hl,12      ; číslo 12
        call DECIMAL2  ; vytiskni jako dvojmístné
        ld  a,13       ; odřádkuj
        rst 16

        ld  hl,1       ; číslo 1
        call DECIMAL1  ; vytiskni jako jednomístné
        ld  a,13       ; odřádkuj
        rst 16

        ld  a,13       ; odřádkuj
        rst 16

        ld  hl,#ABCD   ; číslo #ABCD
        call HEX4      ; vytiskni jako čtyřmístné
        ld  a,13       ; odřádkuj
        rst 16

        ld  hl,#ABC    ; číslo #ABC
        call HEX3      ; vytiskni jako trojmístné
        ld  a,13       ; odřádkuj
        rst 16

        ld  hl,#AB     ; číslo #AB
        call HEX2      ; jako dvojciferné
        ld  a,13       ; odřádkuj
        rst 16

```

```

ld hl,#A ; číslo #A
call HEX1 ; jako jednociferné
ld a,13 ; odřádkuj
rst 16

ret

DECIMAL5 ld de,10000 ; řád desetitisíců
call DIGIT ; počet desetitisíců
DECIMAL4 ld de,1000 ; řád tisíců
call DIGIT ; a jeho počet
DECIMAL3 ld de,100 ; řád stovek
call DIGIT ; počet
DECIMAL2 ld de,10 ; desítky
call DIGIT ; počet
DECIMAL1 ld de,1 ; jednotky

DIGIT ld a,"0"-1 ; do A kód znaku 0 bez jedné
DIGIT2 inc a ; přičti jedničku
or a ; vynuluj CARRY Flag
sbc hl,de ; pokusně odečti řád
jr nc,DIGIT2 ; pokud není výsledek záporný opakuj
add hl,de ; přičti řád zpátky
cp "9"+1 ; testuj znaky 0 až 9
jr c,DIGIT3 ; odskoč pokud platí
add a,"A"-"9"-1 ; oprava na A až F pro hexa čísla
DIGIT3 rst 16 ; vytiskni číslici
ret

HEX4 ld de,#1000 ; počet
call DIGIT ; hexadecimálních tisíců
HEX3 ld de,#100 ; počet
call DIGIT ; hexadecimálních stovek
HEX2 ld de,#10 ; počet
call DIGIT ; hexadecimálních desítek
HEX1 jr DECIMAL1 ; jednotky jako u desítkových

```

Program postupně tiskne jednotlivé řády čísla. Zvolíte-li program pro tisk čísla menšího rozsahu, než je tisknutá hodnota, nebude program pracovat korektně - první řád bude nesmyslný.

U hexadecimálních čísel je dobré vytisknout před číslem ještě znak # pro rozlišení od desítkových čísel.

U čísel se občas hodí, aby se nevýznamné nuly (před číslem) nevypisovaly nebo nahrazovaly mezerami. Druhou možností zajišťuje následující program:

```

ent #

START ld a,2 ; otevření kanálu
call #1501 ; pro tisk

ld hl,12345 ; číslo 12345
call DECIME ; vytiskni jako pětimístné
ld a,13 ; odřádkuj
rst 16

```

```

ld hl,1234 ; číslo 1234
call DECIMS ; vytiskni jako pětimístné
ld a,13 ; odřádkuj
rst 16

ld hl,123 ; číslo 123
call DECIMS ; vytiskni
ld a,13 ; odřádkuj
rst 16

ld hl,12 ; číslo 12
call DECIMS ; vytiskni
ld a,13 ; odřádkuj
rst 16

ld hl,1 ; číslo 1
call DECIMS ; vytiskni
ld a,13 ; odřádkuj
rst 16

ret

DECIMS ld c," " ; do C kód předznaku (mezera)
ld de,10000
call DIGIT21
ld de,1000
call DIGIT21
ld de,100
call DIGIT21
ld e,10
call DIGIT21
ld e,1
ld c,"0" ; poslední řád se tiskne jako nula

DIGIT21 ld a,"0"-1 ;
DIGIT22 inc a ;
or a ;
sbc hl,de ;
jr nc,DIGIT22 ;
add hl,de ;
cp "0" ;
jr nz,DIGIT23 ;
ld a,c ;
DIGIT24 rst 16 ;
ret

DIGIT23 ld c,"0" ;
jr DIGIT24 ;

```

Pokud budete chtít první nuly netisknout, můžete například před tiskem nastavit (**set**) nějaký bit registru **c** jako signál, že se číslo tisknout nemá, při tisku znaku testovat znak **0** a v případě, že je nastaven signál **netisknout** provést návrat, při nalezení první platné číslice však musíte příznak netisknout vynulovat (**res**).

Do registru **c** samozřejmě nemusíte vkládat pouze kód mezery ale i jiného smysluplného znaku (tečka, mínus, hvězdička,...). Můžete tam vložit také nulu a pak se bude program chovat stejně jako v prvním případě.

KLÁVESNICE NA ZX SPECTRU

Testování klávesnice na ZX Spectru lze provádět mnoha způsoby. Postupně se dozvíte všechny. Ke každému způsobu uvidíte program, který vrací v registru a kód stisknuté klávesy. Pro snazší vyzkoušení klávesnicových podprogramů vložte nejprve jednoduchý obrazkovkový editor:

```

ent #
START
TESTER  ld  a,2                ; otevření kanálu pro tisk
        call #1601          ; do horní obrazovky

        xor  a
        ld  (LINE),a
        ld  (COLUMN),a
        ld  hl,22528
        ld  (CURSOR),hl    } počáteční inicializace

MAINLOOP call 8020          ; otestuj klávesu BREAK
        ret nc              ; vrať se, je-li stíštěna

        ld  a,22
        rst 16
        ld  a,(LINE)
        rst 16
        ld  a,(COLUMN)
        rst 16              } nastav pozici pro tisk

        ld  hl,(CURSOR)
        ld  (hl),160        ; namaluj kurzor na
                             ; obrazovku

        call INKEY1        ; čekej na stisk klávesy

        cp  ""
        jr  c,CONTROLS    ; test typu znaku
                             ; odskok pro kód < 32

        rst 16            ; vytiskni znak

        ld  hl,(CURSOR)
        inc hl
        ld  a,(COLUMN)
        inc a
        cp  32
        jr  c,CHAR1
        ld  a,(LINE)
        inc a
        cp  22
        jr  c,CHAR2
        xor  a
        ld  hl,22528
        ld  (LINE),a
        xor  a
        ld  (COLUMN),a
        ld  (CURSOR),hl
        jr  MAINLOOP      } posun na další pozici

CHAR2
CHAR1

```

```

CONTROLS  ld  b,a
           ld  a,(23695)
           ld  hl,(CURSOR)
           ld  (hl),a
           ld  a,b

```

}-smazání kurzoru

```

           cp  8
           jr  nz,CTRL1
           dec hl
           ld  a,(COLUMN)
           dec a
           cp  255
           jr  nz,CLEFTTRGT
           ld  bc,32
           add hl,bc
           ld  a,31
CLEFTTRGT ld  (COLUMN),a
           ld  (CURSOR),hl
           jr  MAINLOOP

```

}-kurzor doleva

```

CTRL1     cp  10
           jr  nz,CTRL2
           ld  bc,32
           add hl,bc
           ld  a,(LINE)
           inc a
           cp  22
           jr  nz,DOWN
           ld  bc,22*32
           or  a
           sbc hl,bc
           xor a
CDOWN     jr  CUPDOWN

```

}-kurzor dolů

```

CTRL2     cp  11
           jr  nz,CTRL3
           ld  bc,32
           or  a
           sbc hl,bc
           ld  a,(LINE)
           dec a
           cp  255
           jr  nz,CUPDOWN
           ld  bc,22*32
           add hl,bc
           ld  a,21
CUPDOWN   ld  (LINE),a
           ld  (CURSOR),hl
           jr  MAINLOOP

```

}-kurzor nahoru

```

CTRL3      cp      9
           jr      nz, CTRL4
           inc     hl
           ld      a, (COLUMN)
           inc     a
           cp      32
           jr      nz, CRIGHT
           ld      bc, 32
           or      a
           sbc     hl, bc
           xor     a
CRIGHT     jr      CLEFTRGT

CTRL4      jp      MAINLOOP      ; další klávesy ne jsou

LINE       defb  0
COLUMN     defb  0
CURSOR     defw  0
           } uložení pozice kurzoru

INKEY1     ei
           halt
           bit    5, (iy+1)      ; povolení přerušení
           jr    z, INKEY1      ; čekání na přerušeni
           jr    z, INKEY1      ; test na stisk klávesy
           jr    z, INKEY1      ; není-li stisk, vrať se
           res    5, (iy+1)      ; zruš příznak stisku
           ld    a, (23560)      ; přečti kód klávesy
           ret
           ; vrať se s kódem v B

A0LENGTH  equ    $-START      ; v A0LENGTH je délka
    
```

První způsob (podprogram **INKEY1**) plně využívá možnosti, které poskytuje operační systém Spectra. Počítač každou padesátinu sekundy provede otestování klávesnice a pokud zaznamená stisk klávesy nebo klávesy a nějakého shiftu, zapíše na adresu **23560** kód stisknuté klávesy a nastaví **pátý bit** na adrese **23611** (neboli **iy+1**). Při použití tohoto podprogramu musíte mít povolené přerušeni v módu **im 1** nebo **im 2**. V druhém případě musí Váš obslužný program pro přerušeni volat podprogram na adrese **56** - nejjednodušší je na konci místo instrukcí **ei** a **ret** vložit instrukci **jp 56** - přerušeni se obnovuje v tomto podprogramu. Registr **iy** musí obsahovat hodnotu **23610** (**#5C3A**). Na testování mají vliv některé systémové proměnné:

23561 - doba (v padesátinách sekundy), která uplyne, než se začne u stále stisknuté klávesy opakovat vráceni jejího kódu - autorepeat. Hodnota se inicializuje na 35.

23562 - interval, v jakém se bude opakovat kód stále stisknuté klávesy. Na počátku je nastavena na 5 padesátin sekundy.

Význam těchto konstant je následující pokud stisknete a budete držet klávesu, vrátí se její kód v okamžiku stisku, potom 35 padesátin sekundy nevrátí počítač nic a potom bude stejný kód vracet každých 5 padesátin sekundy a to dokud klávesu nepustíte. Obě konstanty můžete nastavit na libovolnou hodnotu v rozmezí 0 až 255.

Systém umí číst klávesnici ve všech módech - **☐ ☐ ☐ ☐ ☐**. Nastavení, v jakém módu bude klávesnice čtena, se provádí na třech adresách - 23611, 23617 a 23658. Jednotlivé módy (jde o módy kurzoru v BASICu) nastavíte takto:

```

klávesový mód ☐ - ld (iy+1), 204
                   ld (iy+7), 0
                   ld (iy+48), 8
    
```



```
klávesový mód [ ] - ld (iy+7),1
klávesový mód [ ] - ld (iy+7),2
klávesový mód [ ] - ld (iy+1),196
                    ld (iy+7),0
klávesový mód [ ] - ld (iy+1),204
                    ld (iy+7),0
                    ld (iy+48),0
```

Registr **iy** musí opět ukazovat na adresu **23610 (#5C3A)**. Používáte-li ve svých programech přerušeni v módu **1**, nepoužívejte raději registr **iy** a nechte jej nastavený na uvedenou hodnotu. Pokud se přece jen ukáže použití **iy** jako vhodné, musíte zakázat přerušeni po dobu, kdy je v registru **iy** jiná hodnota.

Uvedená nastavení lze na pro čtení klávesnice používat i v BASICu, nesmíte se však na kód klávesy dotazovat pomocí **INKEY\$** ale přímo pomocí **PEEK 23560**. Podprogram pro **INKEY\$** totiž povoluje čtení pouze v módech **[]** a **[]**.

Občas se může stát, že Vás nezajímá ani kód klávesy podle módu ani to, jestli je stisknutý **CAPS SHIFT** či **SYMBOL SHIFT**, a chcete pouze zjistit, která klávesa je stisknuta. V tomto případě stačí číst adresu **23556** a na ní je zapsán tzv. hlavní kód klávesy a jeho hodnota je obnovována každou padesátinu sekundy bez ohledu na to, jestli klávesa byla stisknuta nebo ne. Pokud klávesa stisknuta nebyla, je zde uložena hodnota **255**. Jinak je tu uložen odpovídající kód číslice, velkého písmene, **ENTERu (13)**, **SPACE (32)** nebo **EXTEND MODE (14)**.

```
INKEY2      ei                ; povolení přerušeni
            halt            ; čekání na přerušeni
            ld  a, (23556)   ; načtení hodnoty do A
            cp  255         ; test na nestištění
            jr  z, INKEY2   ; skok zpět když platí
            ret            ; návrat z podprogramu
```

Ve zkušebním programu přepište volání **INKEY1** na **INKEY2** a můžete vyzkoušet funkci. Tentokrát Vám nebudou pracovat pohyby kurzoru.

Instrukce **ei** v podprogramu nemusí být pokud máte jistotu, že při volání je přerušeni povoleno. Instrukce **halt** také není pro funkci programu bezpodmínečně nutná.

Tento i předchozí podprogramy jsou udělány tak, že čekají, až nějaká klávesa stisknuta bude. Pokud budete chtít, aby byl signalizován stav, kdy žádná klávesa stisknuta není, stačí provést jednoduché úpravy - místo skoku zpět do testu vložit do registru a třeba nulu (kód, který říká, že žádná klávesa stisknuta nebyla) a vrátit se zpět.

Dosud popsané způsoby testování vyžadují pro práci povolené přerušeni - tato vlastnost však může být občas nežádoucí. Vyzkoušejte tento program:

```
INKEY3      call 654        ; volání KEY-SCAN v ROM
            jr  nz, INKEY3  ; více kláves, skok zpět
            call 798        ; volání K-TEST v ROM
            jr  nc, INKEY3  ; nevyhovuje, skok zpět
            dec  d          ; nastavení [ ] módu
            ld  e, a        ; hlavní kód do E
            jp  819        ; skok do dekódování
```

Tato rutina vrací zpět hodnoty jako **INKEY1** v módu **1**. Na tento program má vliv to, kam ukazuje registr **iy**, pokud je totiž na adrese **iy+48** nastaven **3 bit**, jde o mód **0**, pokud je tento bit nulový, jde o mód **1**. Pokud tedy **iy** ukazuje do systémových **proměnných (23610 - #5C3A)**, jde o takový mód, který byl naposledy nastaven. Chcete-li mít test určitě v módu **1**, nastavte registr **iy** na hodnotu **39** před voláním **KEY-SCAN**. Více podrobností se můžete dozvědět z komentovaného výpisu **ROM**.

Předchozí způsoby vracení kódy kláves stejné, jako jsou při práci v BASICu. Někdy se hodí, aby byly vráceny kódy jiné. Můžete si vyrobit tabulku změn a nevhodné kódy nahradit požadovanými - je vhodné pokud se nejedná o příliš mnoho kláves. Chcete-li klávesnici doslova převrátit naruby, je vhodnější následující program:

```

INKEY4   call 654           ; volání KEY-SCAN v ROM
         jr   z, INKEY4    ; více kláves, opět test
         ld  a,e           ; test jestli byla vůbec
         cp  255          ; nějaká klávesa stišťena
         jr  z, INKEY4    ; pokud ne, testuj znovu

         ld  a,d           ; do A případný SHIFT

         ld  hl, SYMBTAB   } stišťen SYMBOL SHIFT
         cp  #18
         jr  z, INKEY4A

         ld  hl, CAPSTAB   } stišťen CAPS SHIFT
         cp  #27
         jr  z, INKEY4A

         ld  hl, NORMTAB   ; nebylo stišťeno nic

INKEY4A  ld  d, 0
         add hl, de
         ld  a, (hl)
         ret              } přečtení kódu z tabulky

SYMBTAB  defm " *† [ & % > } / "
         defm " , - ] ' # < { ? "
         defm " . ( # "
         defb 143
         defm "\_E"
         defb 0
         defm " = ; ) @ "
         defb 131
         defm " | : "
         defb " " , 13 , 34
         defm " _ ! "
         defb 130
         defm " ~ " , 0
         } tabulka pro SYMBOL

```

```

CAPSTAB  def m "BH Y"
         def b 10,8
         def m "TGV"
         def m "NJU"
         def b 11,5
         def m "RFC"
         def m "MKI"
         def b 9,4
         def m "EDX"
         def b 0
         def m "LO"
         def b 15,6
         def m "WSZ"
         def b " ",13,"P"
         def b 12,7
         def m "QA"

```

} tabulka pro CAPS

```

NORMTAB  def m "bhy65tgv"
         def m "nju74rfc"
         def m "mki83edx"
         def b 0
         def m "lo92wsz"
         def b " ",13
         def m "p01qa"
         def b 0

```

} tabulka bez shiftu

Podprogram **KEY-SCAN** vrací zpět v registru e hodnotu v rozsahu 0 až 39 - pokud byla stisknuta nějaká klávesa, nebo hodnotu 255 - pokud nebyla stisknuta žádná klávesa. Je-li současně stisknut nějaký **SHIFT**, je jeho hodnota uložena v registru d (#18 pro **SYMBOL SHIFT** a #27 pro **CAPS SHIFT**). Při neúspěšném testu (bylo stišťeno více kláves a ani jedna nebyl **SHIFT**) je nastavena podmínka **nz**, v opačném případě pak platí **z**. Stisknete-li současně **CAPS SHIFT** a **SYMBOL SHIFT**, bude kód pro **CAPS SHIFT** v registru d a pro **SYMBOL SHIFT** v e.

Pomocí této rutiny můžete provádět i takové změny, které nelze dosáhnou tabulkou změn - testovat např. **CAPS SHIFT** a **ENTER** nebo **SYMBOL SHIFT** a **SPACE** a podobné kombinace. Stačí na vhodném místě tabulky vložit požadovaný kód. Například klávesa **ENTER** vrací kód 13 ať je stisknuta sama nebo spolu s nějakým **SHIFTEM**. Ve všech tabulkách je na tomtéž místě napsána hodnota 13, obdobně klávesa **SPACE** vrací vždy kód 32.

Pokud budete chtít testovat klávesu bez ohledu na to, jestli je stišťena ještě nějaká další klávesa, musíte číst přímo jednotlivé porty klávesnice. Nejprve ukázkový testovací program:

```

ent     $

MAINLOOP call SCANNER
        call SHOWKEYS

        call 8020
        jr   c,MAINLOOP
        ret

```

} hlavní smyčka

```

SHOWKEYS  ld  hl,KEYBOARD ;buffer s klávesami
           ld  ix,22528+33 ;adresa v attributech
           ld  b,4 ;čtyři řádky
SHOW1     ld  c,10 ;kláves je deset v řadě
SHOW0     ld  a,(hl)
           inc hl
           ld  (ix+0),a
           ld  (ix+1),a
           ld  (ix+32),a
           ld  (ix+33),a }zobraz klávesu a posun
           inc ix
           inc ix
           inc ix
           dec c ;vnitřní cyklus přes
           jr  nz,SHOW0 ;sloupce a skok zpět
           ld  de,66
           add ix,de }posun na další řádek
           djnz SHOW1
           ret

SCANNER   ld  hl,PORTTAB ;tabulka adres portů
           ld  ix,KEYBOARD ;buffer pro klávesy
           ld  e,4 ;čtyři řádky klávesnice
           ld  c,254 ;dolní byte adresy portu

SCAN1     ld  b,(hl) ;horní byte adresy portu
           inc hl ;posun na další položku
           push hl ;uložení pro další použití
           ld  d,5 ;kláves je 5 na portu
           ld  hl,BITTAB ;tabulka masek pro bity

SCAN2     in  a,(c) ;hodnota portu BC do A
           cpl ;komplement registru A
           and (hl) ;ponech pouze žádaný bit
           inc hl ;další položka v tabulce
           jr  z,SCAN2B ;odskok, není stišťeno
           ld  a,255 ;signál - je stišťena
SCAN2B    ld  (ix+0),a ;zapiš výsledek
           inc ix ;posun na další klávesu
           dec d ;opakuji celkem 5 krát
           jr  nz,SCAN2 ;skok na začátek cyklu

           pop hl ;ukazatel na porty
           ld  b,(hl) ;horní byte adresy portu
           inc hl ;posun na další
           push hl ;ulož pro další použití
           ld  d,5 ;znovu pět kláves
           ld  hl,BITTAB+4 ;bity jsou řazeny opačně

SCAN3     in  a,(c)
           cpl
           and (hl)
           dec hl
           jr  z,SCAN3B
           ld  a,255
SCAN3B    ld  (ix+0),a
           inc ix
           dec d
           jr  nz,SCAN3 }test pravé půlky řádku

```

```

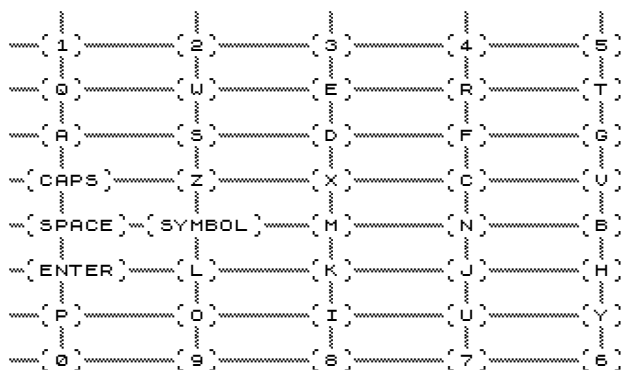
pop hl ; ukazatel na porty
dec e ; klávesnice má 4 řádky
jr nz,SCAN1 ; vrať se na start cyklu
ret

PORTTAB defb 247,239 ; 1 2 3 4 5, 0 9 8 7 6
defb 251,223 ; Q W E R T, P O I U Y
defb 253,191 ; A S D F G, Enter L K J H
defb 254,127 ; C S Z X C V, SP SS M N B

BITTAB defb 1,2,4,8,16 ; bity 0 až 4

KEYBOARD defb 8*5 ; místo pro 40 kláves
    
```

Uvedený program neustále čte klávesnici a do bufferu si zapisuje informace o každé klávese, jestli je nebo není stisknuta. Po každém přečtení klávesnice tyto informace zobrazí na obrazovku - stisknuté klávesy jsou svítivé bílé čtverce, klávesy nestisknuté pak čtverce černé. Program přerušíte stiskem **BREAKu**. Uspořádání klávesnice ZX Spectra neumožňuje úplně nezávislé testování každé klávesy - stisknete-li třeba klávesy 1 až 5, budete je držet a potom stisknete nějakou další klávesu, projeví se to rozsvícením více kláves navíc (celé pětice kláves, do které přidaná klávesa patří). Uspořádání je následující:



Poslední informace, které potřebujete vědět pro testování libovolné klávesy, jsou adresy portů a bity pro jednotlivé klávesy. Všechno naleznete v následující tabulce:

Adresa portu		0	1	2	3	4
254	11111110	CSH	Z	X	C	U
253	11111101	A	S	D	F	G
251	11111011	0	U	R	R	T
247	11110111	1	2	3	4	5
239	11101111	0	9	8	7	6
223	11011111	P	O	I	U	Y
191	10111111	ENT	L	K	J	H
127	01111111	SPC	SSH	M	N	B

První tři sloupce v tabulce jsou adresa portu vypsaná třemi způsoby. První dva obsahují horní byte adresy vypsaný nejprve v desítkové a potom ve dvojkové soustavě. Třetí pak celou adresu (dolní byte je vždy **254**). V pravé části tabulky jsou jednotlivé bity a klávesy, které jim odpovídají. Pokud je vybraná klávesa stisknuta, bude mít odpovídající bit hodnotu **0**, v opačném případě pak hodnotu **1**.

Například: chcete testovat klávesu **Q**. V tabulce je napsáno, že tato klávesa je na portu **64510** (horní byte je **251**) a jedná se o nulový bit tohoto portu. Testovací program může vypadat například takto:

```

ld    bc,64510      ; adresa portu do BC
in    a,(c)         ; přečti port do A
bit   0,a           ; platí Z je-li stisknuta

ld    bc,64510      ; adresa portu do BC
in    a,(c)         ; přečte port do A
rra   ; platí NC je-li stisknuta

ld    a,251         ; horní byte adresy do A
in    a,(254)       ; přečti port 254 do A
bit   0,a           ; platí Z je-li stisknuta

ld    a,251         ; horní byte adresy do A
in    a,(254)       ; přečti port 254 do A
rra   ; platí NC je-li stisknuta

```

Zde máte čtyři způsoby testování klávesy **Q**, nejkratší z nich je poslední program. Pokud budete testovat jiné, než krajní klávesy, nebude použití rotací vhodné - neplatí v případě, že testujete několik kláves na jednom portu, například rozeskok podle klávesy - například rozeskok v menu lze napsat takto:

```

ld    a,247         ; horní byte adresy do A
in    a,(254)       ; přečti port 254 do A
rra   ; zarotuj A (0 bit)
jp    nc,PRESSED1  ; platí NC při stisku 1
rra   ; zarotuj A (1 bit)
jp    nc,PRESSED2  ; platí NC při stisku 2
rra   ; zarotuj A (2 bit)
jp    nc,PRESSED3  ; platí NC při stisku 3
rra   ; zarotuj A (3 bit)
jp    nc,PRESSED4  ; platí NC při stisku 4
jp    NOPRESS      ; nebylo stisknuto nic

```

Při definici ovládání ve hrách se často používá program pro zjištění bitu a portu stisknuté klávesy. Těmito hodnotami se potom modifikuje ta část programu, která je volána pro řízení pohybu.

```

LOOP  ld    bc,#FEFE      ; do BC adresa portu
      in    a,(c)         ; přečti do A hodnotu
      cpl   ; komplement A registru
      and   31            ; ponechej bity 0 až 4
      ret   nz           ; vrať se při stisku
      rlc   b            ; posun na další port
      jr    c,LOOP       ; není-li poslední, cyklusj
      scf   ; nastav C a vrať se
      ret

```

Pokud byla při zavolání programu stisknuta libovolná klávesa, vrátí program v registru **bc** adresu portu a v registru **a** nastavený odpovídající bit - platí podmínka **nc**. Pokud nebude nalezeno nic, vrátí program podmínku **c**. Bude-li stisknuto více kláves, nemusí program pracovat správně.

- - -

Posledním programem bude čekání na stisk libovolné klávesy. Uvedeme zde celkem tři způsoby:

```

WAITKEY  call 654           ; volání KEY-SCAN v ROM
         inc  e             ; pokud je v registru E
         jr  z, WAITKEY     ; číslo 255, testuj znovu

```

Toto je nejkratší způsob, jak napsat čekání na stisk klávesy, má jen tu nevýhodu, že mění registry **af**, **bc**, **de**, **hl**. Tuto nevýhodu nemá následující program (mění pouze **af**):

```

WAITKEY  xor  a             ; vynuluj registr A
         in  a, (254)       ; přečti port 254 do A
         cpl                ; komplement A
         and  31            ; testuj bity 0 až 4
         jr  z, WAITKEY     ; není stisk, opakuj

```

Tento program potřebuje vysvětlení. Pokud je horní adresa portu rovna **0**, testují se najednou všechny klávesové porty. Každý bit vlastně reprezentuje celý sloupec kláves z tabulky. Je-li alespoň jedna klávesa stisknuta, je hodnota bitu rovna **0**, jinak je rovna **1**.

Pro čekání na klávesu lze použít také podprogram příkazu **PAUSE** v ROM. Na rozdíl od předchozích vyžaduje povolené přerušování a tedy i registr **iy** nastavený na **23610 (#5C3A)**, umožňuje však navíc po zadaném časovém intervalu pokračování i bez stisku klávesy. Program mění registry **af** a **bc**.

```

res 5, (iy+1)           ; klávesa není stisknuta
ld  bc, 100             ; čekej maximálně 2 sec.
ei                                     ; povolení přerušování
call #1F3D              ; podprogram PAUSE

```

Doba, po kterou se na klávesu čeká, je uložena v registru **bc** a měří se v padesátinách sekundy. Pokud do **bc** vložíte **0**, bude čekání ukončeno pouze stiskem klávesy.

Poslední skutečnost, na kterou si dávejte pozor, je to, že klávesnice na ZX Spectru je málo kvalitní a má občas tendenci prokmitávat nebo držet stisknuta delší dobu než by měla. Následky jsou pak takové, že se obtížně píšou vstupy do programu - takové nešvary se občas vyskytnou i u zahraničních her (hlavně textovky).

Uvedené potíže odstraní snadno tím, že po přečtení klávesy chvíli počkáte. Po přečtení a přijetí klávesy bývá dobrým zvykem oznámit tuto skutečnost zvukovým signálem (klávesové echo). Tot vše.

16-BITOVÁ ARITMETIKA

V této kapitole si ukážeme základní aritmetické operace s šestnáctibitovými čísly, jsou to sčítání, odečítání, násobení, dělení, zbytek po dělení, porovnání a změna znaménka.

Nejprve sčítání - jedinou instrukcí můžete k registru **hl** přičíst obsahy registrů **bc**, **de**, **hl** a **sp**. Podobně tak s indexregistry - pouze místo registru **hl** je odpovídající indexregistr. Pro **hl** a indexregistry tedy máme instrukce **add** - tyto nastavují pouze příznak **CARRY**. Pokud požadujete nastavení i pro další příznaky, musíte použít instrukci **adc**, před tím však nezapomeňte vynulovat **CARRY** (or **a**) - jinak se přičte k výsledku. Pro indexregistry taková instrukce neexistuje!

Pokud však potřebujete pracovat s jinými registry než s **hl** a indexregistry, musíte sčítání rozložit po bytech - například k **bc** přičteme **de**:

```
ADD_BCDE  ld  a , c           ; obsah z C do A
          add  a , e           ; zde je přičten obsah E
          ld  c , a           ; a výsledek jde zpět do C
          ld  a , b           ; do A horní byte - B
          adc  a , d           ; přičtení obsahu D a přenosu
          ld  b , a           ; z nižšího řádu, zápis zpět do B
```

Tímto programem je podle výsledku nastaveny pouze příznaky **CARRY** a **SIGN**, ostatní příznaky jsou evidentně nastaveny podle instrukce **adc** a nesou tedy informace o vyšším bytu výsledku (**CARRY** a **SIGN** platí samozřejmě pro vyšší byte, nicméně totéž platí i pro celý výsledek).

Zcela obdobně lze přičíst ke dvojregistru obsah libovolného 8-bitového registru. Instrukci **adc** pouze modifikujete nahrazením druhého operandu nulou.

Jako zvláštní případ sčítání lze chápat přičtení jedničky - instrukce **inc**. Zde si dejte pozor na to, že tato instrukce neovlivňuje stavové indikátory.

Další na řadě je odečítání - zde je situace podobná jako u sčítání. Pro **hl** máme k dispozici instrukci **sbc**, tedy odečítání s přenosem. Nezapomeňte před odečtením vynulovat **CARRY**!

U odečítání s jinými registry než **hl** musíte provést rozklad na byty - například odečteme od registru **de** registr **ix**:

```
SUB_DEIX  ld  a , e           ; obsah z C do A
          sub  l , x           ; zde je odečten obsah LX
          ld  e , a           ; a výsledek Jde zpět do E
          ld  a , d           ; do A horní byte - D
          sbc  a , h , x       ; odečtení obsahu HX a přenosu
          ld  d , a           ; z nižšího řádu, zápis zpět do D
```

Nastavení příznaku je stejné jako u sčítání po bytech.

Totéž, co o přičtení jedničky, platí i o odečtení jedničky - **dec**.

Na řadu přichází násobení. Nejprve univerzální program pro vynásobení registru **hl** registrem **de**. Program pracuje jak pro čísla bez znaménka tak pro čísla se znaménkem.


```

MULT      ld    b,16                ; 16 řádů
          ld    a,h                ; do "dvojregistru" AC
          ld    c,l                ; vlož obsah HL
          ld    hl,0               ; začínáme u 0

MULT2     add   hl,hl              ; zdvojnásobení HL - další řád
          rl   c                  ; do CARRY zarotuj
          rla                          ; "dvojregistrem" AC
          jr   nc,MULT3           ; odskoč nejde-li o tento řád
          add  hl,de              ; přičti druhý operand

MULT3     djnz MULT2             ; opakuj se všemi řády
    
```

Na začátku dostane program oba činitele v **hl** a **de**, na konci je výsledek v **hl**, **de** se nemění, **af** a **bc** se mění. Z příznaků je nastaven **CARRY**. Chcete-li znát i ostatní příznaky v závislosti na výsledku, změňte instrukci **add hl,hl** na instrukce **or a** a **adc hl,hl**.

Při programování často narazíte na násobení mocninou dvojky - v takovém případě je nejlépe použít rotaci vlevo. U dvojregistru **hl** to zajistí **add hl,hl**, u **bc** a **de** musíte použít instrukce rotací a posunů - například **sla c, rr b**. Opakováním nebo zacyklením docílíte násobení vyššími mocninami.

Pokud potřebujete násobit konstantou, vyplatí se občas následující možnost - budeme násobit třeba číslem **13**. Číslo **13** lze rozložit na mocniny dvou takto: **13 = 1 + 4 + 8**. Potřebný program pro násobení registru **hl** číslem **13** vypadá takto:

```

MULT13    push  hl                ; uložení 1 násobku
          add   hl,hl              ; dvojnásobek (x2)
          add   hl,hl              ; čtyřnásobek (x4)
          push  hl                ; uložení čtyřnásobku
          add   hl,hl              ; osminásobek (x8)
          pop   de                ; do DE čtyřnásobek
          add   hl,de              ; a jeho přičtení (x12)
          pop   de                ; do DE 1 násobek
          add   hl,de              ; a jeho přičtení (x13)
    
```

Neboli - vytvoříte si největší mocninu a při jejím vytváření si ukládáte ty mocniny, které budou potřeba, potom všechno sečtete a dostanete výsledek.

Násobení malým číslem se občas vyplatí převést na sčítání v cyklu - násobení deseti lze porýdit třeba takhle:

```

MULT10    ld    b,10              ; násobíme deseti
          ex   de,hl              ; budeme přičítat v DE
          ld   hl,0               ; začínáme u nuly
MULT10B   add   hl,de             ; přičti DE
          djnz MULT10B           ; a opakuj B krát
    
```

Nejjednodušší případ je násobení **256**, to stačí přenést obsah nižšího bytu do bytu vyššího a nižší byte vynulovat:

```

MULT256   ld    h,l              ; vyšší byte
          ld    l,0              ; nižší byte
    
```

Pro dělení si také nejprve ukážeme obecný program a potom se zmíníme o některých speciálních případech. Nyní program pro dělení dvoiregistru **hl** dvoiregistrem **de**:

```

DIU      ld    a , h           ; do "AC" dělenec
          ld    c , l           ; (číslo, které je děleno)
          ld    hl , 0          ; vynuluj HL (zbytek po dělení)
          ld    b , 16          ; 16 řádů

DIU2     s l i a c            ; "registr" AC zdvojnásob
          r l a                ; a přičti jedničku
          adc   hl , hl         ; zdvojnásob zbytek po dělení
          sbc   hl , de         ; zkus odečíst řád
          jr    nc , DIU3       ; povedlo se, odskoč
          add   hl , de         ; přičti jej zpátky
          dec   c              ; a odečti jedničku od "AC"

DIU3     d j n z DIU2         ; opakuj B-krát
          ld    h , a           ; do H vyšší byte
          ld    l , c           ; do L nižší byte
    
```

Na uvedené rutině pro dělení je zajímavé to, že "dvoiregistr" **AC** obsahuje současně jak část dělence, tak část výsledku.

Důležité je dělení mocninami dvojky. K tomu stačí posun dvoiregistru doprava. Chcete-li třeba vydělit registr **de** osmi, napište toto:

```

DIU8     ld    b , 3           ; třetí mocnina 2 je 8
DIU8B    s r l d              ; posuň vyšší byte
          r r e                ; posuň nižší byte
          d j n z DIU8B       ; opakuj B-krát
    
```

Dělení 256 pořídíte zadarmo tak, že obsah vyššího bytu přepíšete do nižšího a vyšší byte vynulujete.

Potřebujete-li zjistit zbytek po dělení, odstraňte instrukce **ld h,a** a **ld l,c** a výsledek bude v **hl**, případně přepište obsah z **hl** do jiného registru.

Chcete-li porovnat dva dvoiregistry, můžete to provést takto:

```

CPHLDE   or    a              ; vynuluj CARRY
          sbc   hl , de         ; odečti pro nastavení příznaků
          add   hl , de         ; obnov hodnotu a nastav CARRY
    
```

Uvedená sekvence instrukcí provádí totéž, co by byla prováděla neexistující instrukce **cp hl,de**. Můžete tedy provádět **cp hl,hl**, **cp hl,de** a **cp hl,bc**.

Porovnávání jiných registrů docílíte porovnáním jejich vyšších bytů a v případě rovnosti ještě jejich nižších bytů:

```

CPDEBC   ld    a , d           ; vezmi obsah D
          cp    b              ; a porovnej s B
          jr    nz , CPDEBC2    ; nerovnájí-li se, odskoč na konec
          ld    a , e           ; vezmi obsah E
          cp    c              ; a porovnej s C

CPDEBC2  ; zde jsou příznaky jako po "CP DE,BC"
    
```

Zbývá už jen obrácení znaménka. Existují dva způsoby - odečtení od nuly nebo komplementace a přičtení jedničky - zde jsou:

```

SWAPSIGN  ex   de , hl           ; přesuň do DE
           ld   hl , 0           ; do HL de j 0
           or   a                 ; vynuluj CARRY
           sbc  hl , de          ; a odečti

SWAPSIGN  ld   a , d             ; komplementuj
           cpl                    ; obsah
           ld   d , a            ; registru D
           ld   a , e            ; komplementuj
           cpl                    ; obsah
           ld   e , a            ; registru E
           inc  de                ; v DE je nyní -DE

```

Velmi praktický příklad využití naleznete v kapitole **VSTUP VYHODNOCENÍ TEXTU**.

JEDNODUCHÝ ZVUK

ZX Spectrum má jednoduchý reproduktor, který lze nastavit do dvou poloh. Střídáte-li obě polohy dostatečně rychle, vzniká zvuk téhož kmitočtu, s jakým měníte obě polohy reproduktoru. Teoretický způsob výroby zvuku o určitém kmitočtu (výšce) je tedy jasný. Praktické provedení si ukážeme na několika příkladech.

Podstatná informace je, že reproduktor je ovládán **čtvrtým bitem na portu 254**. Tento bit jde také do vstupu **EAR** (vstup z magnetofonu). Pro komunikaci s magnetofonem slouží také výstup **MIC** (výstup na magnetofon). Oba vstupy (výstupy) se používají také jako zdroje signálu pro zesilovač a proto je při tvorbě zvuku žádoucí měnit oba dva.

Nyní už slíbené příklady - jde o několik jednoduchých zvuků, které můžete použít například jako klávesové echo...

```

SOUND1    ld   b , 128           ; celkem 128 změn
           ld   hl , 10000       ; adresa do ROM
SOUND1A   ld   a , (hl)         ; přečti obsah z (HL)
           and  24                ; ponech bity pro zvuk
           or   7                 ; přidej bílý border
           out  (254) , a        ; a pošli do reproduktoru
           dec  l                 ; změn dolní byte adresy
           djnz SOUND1A         ; a opakuj B-krát
           ret

SOUND2    ld   hl , 92           ; nastav se do ROM
SOUND2A   ld   a , (hl)         ; přečti obsah
           or   a                 ; a v případě,
           ret  z                 ; že jde o nulu skončí
           and  24                ; ponech zvukové bity
           or   7                 ; přidej bílý border
           out  (254) , a        ; pošli na port
           inc  hl                ; posuň ukazatel v paměti
           jr   SOUND2A         ; skoč na začátek

```

```

SOUND3   ld    b,32           ; opakuji celkem 32 krát
          ld    hl,1000      ; adresa do ROM
SOUND3B  ld    a,(hl)        ; přečti obsah
          and   24           ; ponech zvukové bity
          or    7            ; bílý border
          out   (254),a      ; a odešli
SOUND3A  dec   a             ; počkej podle toho,
          jr    nz,SOUND3A   ; co jsi odeslal na port
          dec   l            ; změň adresu do ROM
          djnz SOUND3B       ; hlavní smyčka
          ret

SOUND4   ld    l,200        ; první konstanta
          ld    h,100        ; druhá konstanta
          ld    b,10         ; opakuji 10x
SOUND4A  ld    a,7          ; bílý border + OFF
          out   (254),a      ; odešli
          dec   l            ; zmenší první konstantu
          ld    a,l          ; a čekej podle
SOUND4B  dec   a            ; její hodnoty,
          jr    nz,SOUND4B   ; při OFF
          ld    a,24+7       ; bílý border + ON
          out   (254),a      ; odešli
          ld    a,h          ; druhá konstanta
SOUND4C  dec   a            ; se nemění, tedy doba
          jr    nz,SOUND4C   ; čekání při ON Je stejná
          djnz SOUND4A       ; hlavní smyčka
          ret

```

Vytváření zvuků je z velké části používání metody pokusů a omylů. V uvedených programech můžete měnit všechny konstanty a sledovat, co se bude dít se zvukem - zvlášť vděčná je poslední rutina (**SOUND4**), jejíž úpravami získáte rozličné zvuky.

Zajímavé zvukové efekty můžete najít ve většině her. Jeden takový příjemný zvuk si ukážeme. Pokud budete efekty hledat, pátrejte po instrukci **out (254),a**, bez které se vytváření zvuku neobejde (případně **out (c),R1**, ale ty se příliš často nepoužívají).

```

BEEP     ld    e,1          ; začínáme jedničkou
          ld    a,24         ; do A černý border + ON
BEEP2    ld    b,e          ; do B parametr (1,2,4,8,...)
          out   (254),a      ; odešli na port
          xor   24           ; zaměň ON/OFF
BEEP3    djnz BEEP3         ; opakuji podle parametru
          dec   c            ; prostřední cyklus 256x
          jr    nz,BEEP2     ; opakuje vnitřní cyklus
          sla  e             ; vnější cyklus běží 8x
          jr    nc,BEEP2     ; a mění parametr - 1,2,4,8,16,32,...
          ret

```

Poslední ukážka, co lze na Spectru vytvářet se zvukem, je dvojhlasá hudební rutina. Program umožňuje vytvořit doprovodnou melodii do libovolného programu. Umožňuje současně hrát dva různé tóny libovolné délky.

Program hraje buď do dohrání celé melodie nebo do stisku libovolné klávesy. V textu programu si můžete všimnout několika instrukcí **nop**, které jsou určeny pro časové naladění programových smyček, které vytvářejí tóny.

Nyní už vlastní výpis dvoukanálové hudební rutiny - pevné nervy při opisování.

```

ent #

START di ; vynecháním dojde k zhoršení zvuku
      ld hl, DATA ; začátek dat do HL
      ld (DATSTORE+1), hl ; uloží pro další použití
LOOP4 xor a ; testuj
      in a, (254) ; klávesnici
      cpl ; na stisk
      and 31 ; libovolné klávesy
      jr z, DATSTORE ; není-li stisk odskoč
RETURN ei ; povol přerušení a vrať se
      ret

DATSTORE ld hl, 0 ; do HL adresu dat
         ld a, (hl) ; přečti délku
         or a ; a testuj konec dat
         jr z, RETURN ; vrať se - melodie končí

         inc hl ; posuň se na první hlas
         ld b, (hl) ; a přečti jej do B
         inc hl ; posuň se na druhý hlas
         ld c, (hl) ; a přečti jej do C
         inc hl ; posuň se na další tóny
         ld (DATSTORE+1), hl ; a uloží ukazatel na data

         ld hl, 0
         ld l, b
         ld de, TABULKA
         push de
         add hl, de
         ld d, (hl)
         ld e, 1
         ld b, 0
         pop hl
         add hl, bc
         ld h, (hl)
         ld l, e
         } výpočet adresy
         } v tabulce not pro
         } první kanál (D,E)

         } výpočet adresy
         } tabulce not pro
         } druhý kanál (H,L)

BEEPER ld c, a ; vlastní tóny se tvoří zde
BORDER1 ld a, 7 ; barva borderu pro první kanál
        ex a, a f' ; je vložena do záložního AF
BORDER2 ld a, 7 ; barva borderu pro druhý kanál
        ld hx, d ; další části rutiny nebudou
        ld d, 16 ; popsány příliš podrobně
BEEP3 nop ; protože celá rutina není
BEEP4 ex a, a f' ; mým výtvořem a příliš
        dec e ; dobře ji neznám
        out (254), a ; upozorním jen na zajímavé detaily
        jr nz, BEEP1
        ld e, hx
        xor d
        ex a, a f'
        dec l
        jr nz, BEEP2 ; odskok při Nz a instrukce RET NZ se
        ret nz ; nikdy neprovede - časové ladění

```

```

BEEPS    out    (254), a
         ld     l, h
         xor   d
         nop
         ; časové ladění
LOOP9    djnz  BEEP3
         inc   c
         jr   nz, BEEP4
         jr   LOOP4

BEEP1    jr   z, BEEP1          ; kdyby se na tuto instrukci dostal
         ex   af, af'         ; program s podmínkou Z, zacyklil by se
         dec  l
         jp   z, BEEPS        ; stačit by relativní skok ale jde o čas
BEEP2    out    (254), a
         jr   LOOP9

TABULKA  de fb 255, 0, 0, 0, 240, 270 ; tabulka not, každý tón zde má
         de fb 215, 203, 192, 180    ; informace pro vytvářecí cyklus
         de fb 171, 161, 151, 144
         de fb 136, 128, 121, 114
         de fb 108, 102, 96, 91, 86
         de fb 81, 76, 72, 68, 64, 61
         de fb 57, 54, 51, 48, 45, 43
         de fb 40, 38, 36, 34, 32, 30
         de fb 28, 27, 25, 24, 23, 21
         de fb 20, 19, 18, 17, 16, 15
         de fb 14, 13, 12, 1, 0

DATA     de fb 206, 25, 41          ; data, první je délka not
         de fb 231, 13, 39          ; čím větší, tím kratší noty
         de fb 231, 13, 37          ; pak následují výšky not
         de fb 231, 25, 37          ; v obou kanálech

         de fb 231, 25, 32
         de fb 231, 13, 39
         de fb 231, 13, 32
         de fb 231, 25, 41

         de fb 231, 25, 39
         de fb 206, 13, 37
         de fb 231, 27, 39
         de fb 231, 27, 34

         de fb 231, 15, 41
         de fb 231, 15, 43
         de fb 231, 27, 43
         de fb 231, 27, 39

         de fb 231, 15, 44
         de fb 231, 15, 46
         de fb 231, 27, 46
         de fb 231, 27, 39

         de fb 231, 15, 48
         de fb 231, 15, 49
         de fb 231, 27, 49
         de fb 231, 27, 48

```

```

defb 205,15,46
defb 156,36,44
defb 0

LENGHT equ $-START

```

Data v uvedené hudební rutině se sestávají z trojic - délka, výška prvního kanálu, výška druhého kanálu. Data jsou ukončena 0. Program si přečte délku not a po uvedené dobu hraje dva uvedené tóny - chcete-li napsat takovou melodii, ve které jsou v každém kanálu nesteré dlouhé tóny, musíte ty delší rozložit na několik částí podle kratších. Délka je uváděna jako doplněk do 255.

Pokud Vám uvedená melodie připadá známá, pak není divu, protože pochází ze hry EQUINOX stejně jako použitá rutina. Tatáž rutina je použita i v jiných hrách od firmy MICROGEN - Stainless Steel, Frost Byte, Three Weeks in Paradise,...

Popsaný způsob vytváření zvuků má jednu nevýhodu - zdržuje program při běhu, a to tím víc, čím složitější zvuk se vytváří. Proto se ve občas vyskytne přímá implementace zvukové části do výkonné rutiny. Na vhodné místo (obvykle zjištěno pokusy) se vloží navíc ještě vytvoření zvuku. Jako ukázka nám poslouží následující program - jde o efektivní smazání obrazovky (pixelů).

```

ent $

ld hl,0
ld de,16384
ld bc,6144
ldir } pokusné zaplnění obrazovky

PIXCLS ld de,6000 ; adresa do ROM

PC ld hl,16384 ; první byte obrazovky
ld b,0 ; předpokládáme prázdnou obrazovku
push de ; ulož adresu do ROM

PC2 ld a,(de) ; přečti byte z ROM
and (hl) ; a ponech jen některé bity
inc hl ; podle toho, které jsou
and (hl) ; na obrazovce nastaveny
dec hl ; (použij dva po sobě jdoucí byty)
ld (hl),a ; výsledek zapiš (ublyly některé bity)

or b ; do B přidej obsah bytu
ld b,a ; (při prázdné obrazovce je v B nula)

ld a,(hl) ; přečti z obrazovky byte
and 24 ; ponech zvukové bity
or 7 ; přidej bílý border
out (254),a ; a pošl na port

PC4 inc hl ; posuň se na další byte
inc de ; jak v obrazovce, tak v ROM
ld a,h ; testuj konec pixelů
cp 22528/256 ; a v případě, že není dosažen,
jr nz,PC2 ; skoč znovu na začátek

```

```

pop de ; obnov adresu v ROM
inc d ; a posuň se

ld a, b ; testuj, zda už je obrazovka
or a ; smazána a když ještě není
jr nz, PC ; opakuj celé mazání znovu
ret

```

Obdobně můžete ozvučit třeba tisk znaku - do hlavní smyčky přidejte ovládání reproduktoru, třeba takhle:

```

CHAR2
    . . . . .
    ld b, 8
    ld a, (hl) ; přesun bytu z paměti
    ld (de), a ; na obrazovku
    ld c, a ; do C přenášený byte
CHAR3
    dec c ; čekací cyklus
    jr nz, CHAR3 ; přes registr C
    and 24 ; ponech zvukové bity
    or 7 ; přidej bílý border
    out (254), a ; a odešli
    inc hl ; posuň se dál v paměti
    inc d ; i na obrazovce
    djnz CHAR2 ; proved osmkrát
    . . . . .

```

Poslední informace, kterou občas využijete, je využití podprogramu v ROM. Podprogram pro **BEEP** v ROM leží na adrese **#3B5** a vstupní data jsou v registru **hl** a **de**. V registru **de** je uložena hodnota **f*t**, kde **f** je daná frekvence a **t** je doba trvání v sekundách. V registru **hl** je počet T-cyklů na jeden kmit dělený čtyřmi. Neboli **de** obsahuje počet kmitů a **hl** délku kmitu. Potřebné hodnoty si nejčastěji budete muset najít pokusně - pozor na velké hodnoty, mohlo by trvat skutečně dlouho, než by se cyklus ukončil.

Hodnoty můžete také vypočítat: Pro tón střední C je frekvence 261.63 Hz. K vytvoření tónu tedy musí být reproduktor střídavě zapínán a vypínán každou 1/523.26 sekundy. Systémové hodiny ve Spectru jsou nastaveny na kmitočet 3.5 MHz a tón střední C bude tedy vyžadovat 6689 T-cyklů.

$$\begin{aligned}
 \text{hl} &= 3500000 / 261.63 / 2 = 6689 \\
 \text{de} &= 261.63 * 1 = 262
 \end{aligned}$$

Uvedené střední C po dobu jedné sekundy získáte následujícím programem:

```

ld hl, 6689 ; délka cyklu
ld de, 262 ; počet cyklů
call #3B5 ; zavolání podprogramu

```

Uvedeným podprogramem se provádí také klávesový klik. Potřebný program vypadá takto:

```

ld d, 0 ; do DE délka
ld e, (iy-1) ; klávesového kliku
ld hl, #08 ; výška do HL
call #3B5 ; a vlastní klik

```

O dalších možnostech (dokonalejší hudební rutiny) se zmíníme v dalších dílech.

VSTUP A VYHODNOCENÍ TEXTU

Obsahem kapitoly bude rozsáhlý příklad. Dozvíte se jak naprogramovat čtení textu, čtení čísla (desítkové, šestnáctkové, binární), čtení znaku a zpracování jednoduchého aritmetického výrazu (vyhodnocování zleva doprava, na prioritu se nebere ohled). Plivněte si do dlaní a pusťte se do opisování, pak si povíme víc.

	ent	#		
RUN	ld	hl,16384	; tato část smaže obrazovku	
	ld	de,16385	; jestli chcete	
	ld	bc,6143	; pochopit, jak	
	ld	(hl),l	; to dělá, projděte	
	ldir		; si jednotlivé cykly	
	ld	bc,768	; instrukce LDIR na papíře	
	ld	(hl),56	; (stačí samozřejmě pár prvních)	
	ldir		; podobné atributy	
	MAIN	ld	hx,31	; délka vstupu
		ld	hl,6*32+20480	; adresa na obrazovce
call		INPUT	; volej vstupní podprogram	
cp		7	; testuj EDIT	
ret		z	; vrať se do assembleru	
ld		hl,TEXT1	; vytiskni první	
call		TEXTOUT	; text (Unsigned)	
ld		de,23296	; editační řádek v paměti	
call		COMPUT	; vypočítej zapsaný výraz	
push		hl	; ulož získanou hodnotu	
call		NUMBER	; vytiskni jako číslo bez znaménka	
ld		hl,TEXT3	; za číslem vytiskni	
call		TEXTOUT2	; nějaké mezery	
ld		hl,TEXT2	; vytiskni druhý	
call		TEXTOUT	; text (Signed)	
pop	hl	; obnov výsledek výrazu		
call	NUMSIGN	; a vytiskni jej jako číslo		
ld	hl,TEXT3	; se znaménkem, za číslem		
call	TEXTOUT2	; vytiskni mezery (smaže staré číslo)		
jr	MAIN	; skoč pro další výraz		
INPCLEAR	ld	de,(INPOS+1)	; do DE pixelová pozice	
	ld	c,16	; 16 pixelových řádků na výšku	
	INPC2	ld	b,hx	; do B délka řádku
	inc	b	; plus 1 za kurzor	
INPC3	xor	a	; vynuluj A	
	push	de	; ulož adresu začátku řádku	
	ld	(de),a	; vymaž byty	
INPC3	inc	de	; v pixelovém	
	djnz	INPC3	; řádku	
	pop	de	; obnov adresu počátku	

```

call DOWNDE ; a posuň se na další řádek
dec c ; odečti jedničku
jr nz, INPC2 ; a zbývají-li řádky, opakuji
ret

INPUT ld (INPOS+1),hl ; ulož adresu začátku pro další použití
ld hl,23296 ; do HL adresa editační oblasti
ld b,hx ; do B délka editační oblasti
IN1 ld (hl),32 ; a nyní celou editační
inc hl ; zónu vyplníme mezerami
djnz IN1 ; na konec editační zóny
ld (hl),b ; přijde 0

res 5,(iy+1) ; signál není stisknuta klávesa
xor a ; nastav kurzor
ld (CURSOR+1),a ; na začátek editační zóny

IN2 ld b,hx ; nyní celou editační zónu
INPOS ld hl,0 ; vytiskneme, nastav
ld (PPOS+1),hl ; tiskovou pozici
ld hl,23296 ; začínáme od začátku

CURSOR ld c,0 ; do C polohu kurzoru
IN3 ld a,l ; testuj spodní byte adresy
cp c ; v případě rovnosti
ld a,"C"+128 ; dej do A kód kurzoru
call z,CHAR ; a vytiskni ho

ld a,(hl) ; vytiskni znak
call CHAR ; z editační zóny

inc hl ; a posuň se pro další
djnz IN3 ; opakuji se všemi znaky

ld a,l ; kurzor také může
cp c ; být až za posledním
ld a,"<"+128 ; znakem, pak bude
call z,CHAR ; na řádku vypadat jinak

call INKEY ; přečti si kód klávesy
cp 7 ; testuj EDIT (Caps Shift + 1)
ret z ; a případně se vrať zpátky
cp 13 ; testuj ENTER a případně odskoč
jr z,INPCLEAR ; na smazání řádku z obrazovky

ld hl,IN2 ; na zásobník ulož adresu IN2, sem
push hl ; se bude nyní program vracet
ld hl,CURSOR+1 ; do HL vlož adresu pozice kurzoru
cp 8 ; testuj kurzor doleva
jr z,CURSLEFT ; odskoč
cp 9 ; kurzor doprava
jr z,CURSRGHT
cp 12 ; delete (správně BACKSPACE)
jr z,BCKSPACE
cp 199 ; znak <= (funkce DELETE)
jr z,DELETE

```

	cp	32	; nyní zbývají
	ret	c	; obvyčejné znaky,
	cp	128	; odfiltruj
	ret	nc	; netisknutelné znaky
	ex	a f , a f '	; a kód přesuň do A'
	ld	a , (h l)	; testuj, zda není kurzor
	cp	hx	; na konci řádku,
	ret	nc	; když ano, tak se vrať
	inc	(h l)	; posuň kurzor doprava
	ld	l , (h l)	; do HL vlož adresu,
	dec	l	; na kterou bude znak
	ld	h , 23296 / 256	; uložen
INS	ld	a , (h l)	; přečti původní znak
	or	a	; a testuj konec řádku
	ret	z	; případně se vrať
	ex	a f , a f '	; přehoď původní a nový znak
	ld	(h l) , a	; a nový zapiš, pro další znak
	inc	h l	; bude novým znakem předchozí
	jr	INS	; znak, opakuji posun znaku až do konce
CURSLEFT	ld	a , (h l)	; přečti polohu kurzoru
	or	a	; a v případě, že je na levém okraji
	ret	z	; tak se vrať a nic nedělej
	dec	(h l)	; posuň kurzor doleva a vrať se na IN2
	ret		
CURSRGHT	ld	a , (h l)	; přečti polohu kurzoru
	cp	hx	; a když je na konci řádku
	ret	nc	; tak se vrať
	inc	(h l)	; jinak posuň kurzor doprava a vrať se
	ret		; vrať se na IN2
DELETE	ld	a , (h l)	; na konci
	cp	hx	; editační zóny
	ret	z	; DELETE nepracuje
	inc	a	; jinak uprav polohu
	jr	BCK2	; a pokračuj společnou částí
BACKSPACE	ld	a , (h l)	; BACKSPACE naopak
	or	a	; nepracuje na začátku
	ret	z	; editační zóny
	dec	(h l)	; posuň kurzor vlevo
BCK2	ld	l , a	; společná část,
	ld	h , 23296 / 256	; která zajišťuje
	ld	e , l	; přesunutí následujících
	ld	d , h	; znaků na uvolněné místo
	dec	e	; po smazání znaku

DEL2	ld a, (hl) ldi a or a jr nz, DEL2 ex de, hl dec hl ld (hl), " " ret	; vlastní přesun ; je prováděn instrukcí ; LDI dokud není přenesena 0, ; která signalizuje konec zóny ; na poslední pozici, ; která se nyní uvolnila, ; je zapsána mezera ; návrat na IN2
DOWNDE	inc d ld a, d and 7 ret nz ld a, e add a, 32 ld e, a ld a, d jr c, DOWNDE2 sub 8 ld d, a	} posun adresy v obrazovce dolů o pixel
DOWNDE2	cp 88 ret c ld d, 64 ret	
INKEY	ei halt bit 5, (iy+1) jr z, INKEY res 5, (iy+1)	; raději povol přerušování ; a počkej na něj ; testuj stisk ; a když není, čekej dále ; vynuluj si signál o stisku
	push bc push hl	; ulož ; HL a BC
INKEY2	ld hl, 0 ld b, l ld a, (hl) inc hl and 24 or 4 out (254), a djnz INKEY2	} klávesové echo
	pop hl pop bc	
	ld a, (23560) ret	; nyní přečti kód znaku a vrať se
READSIGN	ld a, (de) cp "-" jr nz, READNUM	; přečti znak ; a zjisti, jestli nejde ; o mínus, když ne, skoč do čtení čísla

SWAPSIGN	inc de call READNUM ld a, l cpl ld l, a ld a, h cpl ld h, a inc hl ret	; posuň se na další znak ; zavolej čtení čísla bez znaménka ; a změň znaménko u absolutní hodnoty
INUCHAR	ld a, (de) inc de inc de add a, 128 ld l, a ret	; přečti znak ; a posuň se za něj, ; přeskoč druhý apostrof ; zvedni kód znaku o 128 (invertování) ; vlož je j do HL (v H je nula) ; a vrať se
CODECHAR	ld a, (de) inc de inc de ld l, a ret	; přečti znak ; a posuň se za něj ; přeskoč uvozovku ; a vlož kód do HL ; pak se vrať
READNUM	ld a, (de) inc de ld hl, 0 cp "" jr z, CODECHAR cp "'" jr z, INUCHAR ld b, 16 cp "#" jr z, READNUM3 ld b, 2 cp "%" jr z, READNUM3 ld b, 10 dec de	; čtení čísla a znaku v"" nebo "" ; přečti čím začínáme a posuň se ; číslo začíná nulové ; testuj uvozovku ; a skoč pro obyčejný znak ; testuj apostrof ; a skoč pro invertovaný znak ; do B základ šestnáctkové soustavy ; test dvojitého křížku ; a skoč do čtení ; do B základ dvojkové soustavy ; test procenta ; a skok do čtení ; zbývá jen desítkové číslo, pak je však ; nutno se vrátit o znak - není předznak
READNUM3	ld a, (de) sub "0" cp 10 jr c, READNUM4 sub "A" - "9" - 1	; přečti číslici ; uprav znaky "0" až "9" na 0 až 9 ; další přípustné znaky ; jsou "A" až "F" ; tak je uprav na rozsah 10 až 15
READNUM4	cp 16 jr c, READNUM6 sub 32	; ještě přicházejí ; v úvahu také znaky "a" až "f", ; úprava na rozsah 10 až 15

READNUM5	cp 16	; pokud ani teď není kód mezi 0 a 15
	ret nc	; nejedná se o číslici a nepatří k číslu
	inc de	; posun na další číslici
	push de	; ulož adresu znaku
	ex de,hl	; přehoď nyní jsi číslo do DE
	ld hl,0	; a vynuluj HL
READNUM5	push bc	; ulož základ pro později
	add hl,de	; opakované sčítání místo násobení
	djnz READNUM5	; násobíme základem soustavy
	ld d,b	; do D dej 0
	pop bc	; obnov základ číselné soustavy
	ld e,a	; do E právě čtený řád
	add hl,de	; a přičti
	pop de	; obnov ukazatel do editační zóny
	jr READNUM5	; a skoč pro další znak
NUMSIGN	bit 7,h	; tisk kladného čísla je stejný
	jr z,NUMBER	; jako tisk čísla bez znaménka, odskoč
	call SWAPSIGN	; u záporného čísla se nejprve spočítá
	push hl	; absolutní hodnota a pak se tiskne jako
	ld a,"-"	; číslo bez znaménka, předtím se ovšem
	call CHAR	; vytiskne ještě znak mínus
	pop hl	
NUMBER	ld de,10000	; tisk je celkem obvyklý,
	ld b,0	; na rozdíl od předchozích rutin
	call N1	; však vynechává neplatné nuly
	ld de,1000	; na začátku čísla
	call N1	
	ld de,100	
	call N1	
	ld e,10	; stačí E, v D už nula je
	call N1	
	ld b,1	; signál - tuto nulu už vytiskni
	ld e,b	; jednička do DE
N1	ld a,"0"-1	} obvyklý výpočet řádu
N2	inc a	
	or a	
	sbc hl,de	
	jr nc,N2	
	add hl,de	
	cp "0"	; test na znak 0
	jr nz,N3	; když nejde o "0" odskoč
	bit 0,b	; test neplatné nuly
	ret z	; a pokud je splněn, návrat
N3	ld b,1	; další nuly už jsou platné
		; rutina pokračuje tiskem znaku

```

CHAR      exx

          add  a,a
          ld   l,a
          sbc  a,a
          ld   c,a
          ld   h,15
          add  hl,hl
          add  hl,hl
          }
          } počítání adresy, u kódů
          } větších než 127 je nastaven
          } inverzní tisk (C=255)

PPOS      ld   de,16384
          push de
          ld   b,8

CHAR2     ld   a,(hl)
          rrc  a
          or   (hl)
          xor  c
          ld   (de),a
          call DOWND
          ld   a,(hl)
          xor  c
          ld   (de),a
          call DOWND
          inc  hl
          djnz CHAR2

          pop  de
          inc  e
          ld   a,e
          and  31
          jr   nz,CHAR3
          dec  e
          ld   a,e
          and  %11100000
          ld   e,a
          ld   b,15

CHAR4     call DOWND
          djnz CHAR4

CHAR3     ld   (PPOS+1),de

          exx
          ret

SEEKCHAR  ld   a,(de)
          cp   " "
          ret  nz
          inc  de
          jr   SEEKCHAR
          }
          } přečti znak
          } a když to není mezera,
          } tak se vrať
          } posuň se na
          } další znak

COMPUT    call SEEKCHAR
          call READSIGN
          push de
          }
          } přeskoč mezery
          } přečti číslo se znaménkem do HL
          } ulož adresu znaku

```

COMPUT2	pop de	; obnov adresu znaku
	call SEEKCHAR	; přeskoč mezery, přečti znaménko
	or a	; a testuj konec řádku
	ret z	; vrať se na konci
	push af	; ulož kód operace
	push hl	; ulož prozatímní výsledek
	inc de	; posuň se za operátor (znak operace)
	call SEEKCHAR	; přeskoč mezery
	call READSIGN	; přečti číslo se znaménkem
	pop bc	; obnov současný výsledek
	pop af	; obnov kód operace
	push de	; ulož adresu znaku
	ld d,b	; přesuň současný
	ld e,c	; výsledek do DE
	ld bc,COMPUT2	; na zásobník ulož
	push bc	; adresu COMPUT2
	ex de,hl	; přehoď první a druhý operand
	cp "?"	; testuj modulo (zbytek po dělení)
	jr z,MOD	
	cp "/"	; operace celočíselného dělení
	jr z,LOM	
	cp "*"	; operace násobení
	jr z,KRAT	
	cp "+"	; sčítání
	jr z,PLUS	
	cp "-"	; odčítání
	jr z,MINUS	
	pop af	; neexistující operace, konec výpočtu
	ret	
MOD	ld a,h	} dělení a modulo
	ld c,l	
	ld hl,0	
	ld b,16	
MOD2	slia c	
	rla	
	adc hl,hl	
	sbc hl,de	
	jr nc,MOD1	
	add hl,de	
MOD1	dec c	
	djnz MOD2	
	ret	
LOM	call MOD	} dělení
	ld h,a	
	ld l,c	
	ret	

KRAT	ld	b,16	}	násobení
	ld	a,h		
	ld	c,l		
	ld	hl,0		
KRAT2	add	hl,hl		
	rl	c		
	rla			
	jr	nc,KRAT1	}	odčítání
	add	hl,de		
KRAT1	djnz	KRAT2		
	ret			
MINUS	or	a	}	odčítání
	sbc	hl,de		
	ret			
PLUS	add	hl,de		;sčítání
	ret			
TEXTOUT	ld	e,(hl)	}	počáteční tisková pozice
	inc	hl		
	ld	d,(hl)		
	inc	hl		
	ld	(PPOS+1),de		
TEXTOUT2	ld	a,(hl)	}	vlastní tisk textu
	and	127		
	call	CHAR		
	bit	7,(hl)		
	inc	hl		
	jr	z,TEXTOUT2		
	ret			
TEXT1	defw	18432+8		;první text s tiskovou pozicí
	defm	'Unsigned: '		
TEXT2	defw	18432+96+8		;druhý text
	defm	' Signed: '		
TEXT3	defm	'		;třetí text

Hotovo? Než program poprvé spustíte, tak si jeho zdrojový text raději nahrajte na kazetu - budou-li v programu nějaké chyby (vzniklé při přepisu), mohl by Vás opustit a psát znovu tak dlouhý text není příjemné.

Asi neuškodí, když si o některých podprogramech povíme něco podrobnějšího. Budeme je probírat ve stejném pořadí, v jakém jsou uvedeny v programu:

INPCLEAR - vyčištění obdélníku na obrazovce, šířka je zadána v osmicích bodů (vždy jeden byte), výška je zadána v bodech. Počáteční adresa je v **de** registru. Podprogram můžete modifikovat a používat ve vlastních programech také k zaplnění dané plochy.

INPUT - vstup textu zadané délky (uložena v **hx**). Poloha vstupního řádku na obrazovce se zadává adresou levého horního rohu (uložena v **hl**). Vstup lze ovládat těmito klávesami:

Caps Shift + 5 - kurzor doleva o jeden znak
Caps Shift + 8 - kurzor doprava o jeden znak
Caps Shift + 0 - smazání znaku před kurzorem
Symbol Shift + Q - smazání znaku za kurzorem
Caps Shift + 1 - ukončení vstupu bez smazání na obrazovce
Enter - ukončení vstupu se smazáním z obrazovky

Program používá jako editační oblast paměť od adresy 23296 - na Spectru je tato část paměti používána jako tiskový buffer. Tuto adresu můžete změnit, musí však jít o číslo beze zbytku dělitelné 256. Pracovní oblast si podprogram čistí sám. Pokud je při návratu v registru **a** číslo **7**, pak k návratu došlo stiskem **EDIT (Caps Shift + 1)**.

DOWNDE - tradiční posun adresy na obrazovce o jeden pixel dolů, tentokrát pro **de**.

INKEY - podprogram vrací hodnotu stisknuté klávesy, pokud není klávesa stisknuta, čeká na ni. Kód klávesy je vrácen v akumulátoru. Podprogram používá přerušeni v módu 1.

READSIGN - očekává, že v registru **de** je adresa řetězce čísel. Po vykonání ukazuje registr **de** za číselný řetězec. V registru **hl** je hodnota přečteného čísla. Program umí číst tyto zápisy čísel se znaménkem:

desítkové číslo - 12345, -9887
šestnáctkové číslo - #A12F, -#B01
binární číslo - %1001011, -%11001
znak - "A", "-z"
invertovaný znak - 'A', -'#

Za ukončení čísla je považován každý nepatřičný znak. Program neprovádí kontrolu správnosti zápisu - např. v binárním čísle mohou být i jiné číslice než **0** a **1** - výsledek je pak ovšem samozřejmě chybný. Mezery se uvnitř čísla vyskytovat nesmí - výskyt mezery je chápán jako ukončení zápisu čísla. Pokud na vhodné místo přidáte volání **SEEKCHAR**, můžete v zápisu čísla mezery mít (místo instrukce **ld a,(de)** dejte **call SEEKCHAR**).

READNUM - obdobný program jako **READSIGN**, liší se tím, že pracuje pro čísla bez znaménka. Výsledek je také v **hl**.

NUMSIGN - vypíše obsah **hl** jako číslo se znaménkem.

NUMBER - vypíše obsah **hl** jako číslo bez znaménka.

CHAR - vytiskne znak v akumulátoru. Tiskne znaky ve dvojnásobné výšce a znaky s kódy o 128 vyššími než ASCII tiskne inverzně.

SEEKCHAR - očekává, že **de** ukazuje na znakový řetězec. Vrací v **de** adresu prvního znaku - přeskočí mezery. Mimo upravené adresy vrací také v akumulátoru kód nalezeného znaku.

COMPUT - v **de** dostane adresu výrazu a v **hl** vrací jeho hodnotu. Registr **de** po skončení ukazuje za výraz - na první znak, který nelze chápat jako část výrazu. Ve výrazu se mohou vyskytovat čísla (viz **READSIGN**) oddělená operátory **+**, **-**, *****, **/** a **?** (zbytek po dělení). Mezi čísla a operátory mohou být také mezery. Vyhodnocování výrazu je prováděno zleva doprava a na prioritu operátorů není brán ohled (vyjma unárního minusu, což je znaménko u čísla).

MOD, LOM, KRAT, PLUS, MINUS - podprogramy pro uvedené operace - první operand je vždy v **hl**, druhý vždy v **de** a výsledek je vrácen v **hl**.

TEXTOUT - obvyklý program pro tisk textu, očekává v **hl** adresu textu v paměti. Text musí na začátku obsahovat adresu tiskové pozice, na konci je ukončen invertovaným znakem.

TEXTOUT2 - část předchozího podprogramu, odtud volejte tisk v případě, že chcete pokračovat v tisku na místě, kde tisk naposledy skončil. Registr **hl** musí obsahovat adresu textu (pokud tedy tisknete tentýž text podprogramy **TEXTOUT1** a **TEXTOUT2**, musíte v druhém případě použít adresu o dvě větší než v případě prvním).

Uvedený program si důkladně prohlédnete a některé části protrasujete. Můžete se pokusit jej vylepšit - část v okolí **CODECHAR** by se dala napsat kratší (společné části). Složitější úprava bude vylepšit program tak, aby v zápisu výrazu dokázal najít možné chyby a upozornit na ně - iniciativě se meze nekladou. Program **INPUT** můžete navíc vylepšit tak, aby mohl nabízet standardní hodnoty u číselných vstupů - znamená to zapsat do editační zóny nabízenou hodnotu a nastavit kurzor za poslední znak. Pro zapsání můžete použít stejný program jako pro tisk, pouze místo tisku budete jednotlivé číslice zapisovat do editační zóny.

Při programování složitějších vstupů, kde přicházejí v úvahu nějaká chybová hlášení, si dejte práci s tím, aby se při chybě vrátil editační řádek ve stavu, v jakém byl odeslán. Rozhodně není šťastné při chybě celý vstup smazat a nutit uživatele aby jej zadal znovu. Důležitá je také co nejlepší detekce a hlavně signalizace chyb - neškodí, když program přímo ukáže a napíše, co se mu nelíbí.

Slušností ale také pudem sebezáchovy lze odůvodnit požadavek na to, aby program pokud možno dostatečně jasně naznačil, co vlastně od uživatele chce. Proč se jedná o pud sebezáchovy poznáte v okamžiku, kdy narazíte na nějaký svůj starší program, kde je tato zásada opomenuta, a sami nevíte, k čemu se program může hodit. Otázka typu **A=** je sice hezká, ale příliš informací neposkytuje, o případech, kde se objeví pouze kurzor bez jakéhokoliv náznaku, co se má vkládat, ani nemluvě (to se týká hlavně programů v BASICu a jiných vyšších programovacích jazycích, kde je možno použít již hotové podprogramy). Uvedený příklad, jak jinak, tuto zásadu porušuje - je na Vás, aby tomu tak nebylo.

Další užitečná maličkost je možnost kdykoliv vstup přerušit a vrátit se do vyšší úrovně - takto je v různých programech umožněno předčasně ukončit prováděnou akci při zadávání parametrů (zvolíte **SAVE** a při otázce na jméno si to rozmyslíte). Pro výskok ze vstupu použijte klávesu **EDIT (CS+1)**, je to celkem standardní a mnohé programy (obzvláště ty mé) to používají.

Poslední odstavce v této kapitole obsahuje některé nápady pro vylepšení vstupu. Zkuste sami přidat tyto funkce: skok na začátek a na konec řádku, posun na předchozí a na následující slovo, smazání editačního řádku, vyvolání minulého obsahu (textu, který byl naposledy odeslán), přepínání mezi vkládacím a přepisovacím módem kurzoru (dobře signalizovat přímo tvarem kurzoru), umožnit **CAPS LOCK**, . . .

KAZETOVÉ OPERACE

Poslední kapitola prvního dílu příručky **Assembler a ZX Spectrum** Vás seznámí se způsobem, jak programovat spolupráci s magnetofonem na úrovni strojového kódu. Zatím si ukážeme jak používat podprogramy **SAVE a LOAD (VERIFY) z ROM**. O tom, jak vyrobit vlastní rutiny, se dozvíte v příštím díle.

Nejprve si vyzkoušejte ukázkový program - umožňuje nahrát do paměti obrázek, odstraní z něj atributy a invertuje pixely. Potom obrázek předvede a po stisku klávesy ho uloží pod novým jménem na kazetu, později umožní nahrávku verifikovat nebo opakovat. Program je tradičně delší, obsahuje však množství nových nápadů a tak se Vám práce s vkládáním vyplatí:

```

ent $

BEGIN      di                ; zakaž přerušení
           call INPCOM       ; nech si zadat jméno
           cp 7              ; pokud byl stížen EDIT
           ret z             ; při zadávání, tak se vrať
BEGIN2     call LDHEAD       ; přečti hlavičku z kazety
           jr nc, BEGIN      ; je-li BREAK, skoč znovu pro jméno
           ld b, 10          ; ngní testuj,
           ld hl, INLIN2     ; zda se zadané
           ld a, 32          ; jméno neskládá
TST        cp (hl)          ; jenom z mezer,
           inc hl            ; pokud ano, můžeš
           jr nz, TST20      ; nahrát
           djnz TST          ; libovolný
           jr LLLLLL         ; blok

TST2       ld de, INLIN2     ; editační zóna
           ld hl, HEAD2+1    ; jméno v hlavičce
           ld b, 10          ; má deset znaků
TST3       ld a, (de)        ; ngní se
           cp (hl)          ; provede
           ret nz           ; porovnání
           inc hl            ; jména
           inc de            ; v hlavičce
           djnz TST3         ; se zadaným
           ret              ; jménem (podprogram)

TST20      call TST2         ; volej test jména
           jr nz, BEGIN2     ; není stejné, hledej dál

LLLLL      ld ix, FREE       ; blok dat se načte za program
           ld de, (HEAD2+11) ; délka bloku
           scf               ; nastav CARRY
           sbc a, a          ; a do A dej 255
           call LOAD         ; nahraj data
           ex af, af'        ; v CARRY je informace o paritě bloku
           ld a, 127         ; testuj
           in a, (254)        ; klávesu
           rra               ; SPACE (BREAK)
           jr nc, BEGIN      ; a vrať se na začátek, je-li stížena
           ex af, af'        ; obnov AF (kvůli CARRY)
           jr c, CONVERT     ; je-li CARRY=1, je nahrávka OK, odskoč

```

```

call TAPERROR ; jinak vypiš chybové hlášení
jr BEGIN2 ; a pokud se blok přečíst znovu

CONVERT ld hl, FREE
ld bc, 6144

CONVERT2 ld a, (hl)
cpl ; invertuj
ld (hl), a ; pixelovou část
inc hl ; obrazovky
dec bc
ld a, b
or c
jr nz, CONVERT2

CONVERT3 ld bc, 768
ld (hl), 56 ; atributy nastav
inc hl ; černý inkoust
dec bc ; a bílý papír
ld a, b
or c
jr nz, CONVERT3

out (254), a ; v A je nula, nastav BORDER

ld hl, FREE
ld de, 16384 ; ukaž co jsi vytvořil
ld bc, 6912
ldir

ld bc, 0
ei ; čekání na klávesu
call #1F3D

call #D6B
ld a, 7 ; mazání obrazovky + bílý BORDER
out (254), a

call INPCOM ; jméno pro SAVE
cp 7
ret z

ld hl, INLIN2 ; přenes
ld de, NAME ; jméno
ld bc, 10 ; do hlavičky
ldir ; pro SAVE

SAVE call TT
defm " Start tape " ; vytiskni text
defm '& Key '

ld bc, 0
ei ; a počkej na stisk klávesy
call #1F3D

call #D6B ; smaž obrazovku

```

```

ld ix,HEAD ; do IX začátek hlavičky
ld de,17 ; hlavička má délku 17 bytů
xor a ; a LEADER vždy 0
call PAUSE ; chvílku počkej a proved SAVE
ld a,127 ; testuj BREAK
in a,(254) ; a pokud je
rra ; stisknut,
jr nc,MENU ; skoč do volby operace

MENU
call START ; nastav registry pro blok dat
call PAUSE ; a ulož jej na kazetu
call TT
defb 20,1,"S",20,0
defm "ave "
defb 20,1,"U",20,0
defm "erify "
defb 20,1,"R",20,0
defm "eturn "
} tiskni hlavní menu

call KEY
cp "s"
jr z,SAVE
cp "r"
ret z
cp "v"
jr nz,MENU
} rozeskok podle klávesy

LOOP
call LDHEAD ; VERIFY - hledej hlavičku
jr nc,MENU ; na BREAK se vrať do menu
call TST2 ; testuj jméno ze SAVE
jr nz,LOOP ; když je jiné, hledej dál

call START ; nastav registry pro blok
cp a ; vynuluj CARRY flag (priznak VERIFY)
call LOAD ; a proved VERIFY
jr c,MENU ; pokud je vše OK, vrať se
ld a,127 ; testuj SPACE (BREAK)
in a,(254) ; a pokud je klávesa
rra ; stisknuta, vrať
jr nc,MENU ; se také do menu
call TAPERROR ; už zbývá jen možnost,
jr MENU ; že došlo k nalezení chyby, ohlaš ji

START
ld ix,FREE ; do IX začátek a do DE délku bloku
ld de,6912 ; do A jde 255 - leader, toto je
ld a,255 ; společné pro všechny kazetové
ret ; operace - SAVE, LOAD i VERIFY

PAUSE
ld b,30 ; počkej
ei ; něco přes
halt ; půl sekundy
djnz HALT ; a potom
jp #4C6 ; skoč do rutiny SAVE

HALT

HEAD
defb 3 ; typ bloku - CODE
NAME defm "0123456789" ; deset znaků pro jméno - přepisují se
LEN defw 6912 ; délka bloku
defw 16384,0 ; počáteční adresa a další paramet

```

```

HEAD2      defb 17      ; vyhrad si místo pro hlavičku

IP4        cp    12
           jr    nz, IP8
           dec  hl
           bit  7, (hl)
           jr    nz, IP2
           inc  hl
           ld   (hl), 32
           } ošetření DELETE
IP7        dec  hl
IP6        ld   (hl), "_"
           ld   (INPOS+1), hl
           jr   IP2

IP8        cp    32
           jr    c, IP2
           and  127
           ld   (hl), a
           inc  hl
           bit  7, (hl)
           jr    nz, IP7
           jr   IP6
           } ošetření znaků

INPCOM     ld   hl, INLIN2
           ld   (INPOS+1), hl
           ld   (hl), "_"
           ld   b, 10
           } inicializace vstupu
IP3        inc  hl
           ld   (hl), 32
           djnz IP3

IP2        call TT
TXT        defm ' Name: '
           call TT2
INLIN2     defm ' 0123456789 '
           } tisk

           call KEY      ; klávesa
INPOS      ld   hl, 0     ; pozice kurzoru
           cp    7       ; návrat při
           ret  z        ; stisku EDIT
           cp    13      ; test ENTER
           jr    nz, IP4 ; odskok pro ostatní

IP5        ld   (hl), 32
           inc  hl
           bit  7, (hl)
           jr   z, IP5   ; vyplnění oblasti od kurzoru
           }             ; do konce řádku mezerami
           }             ; (smaže i kurzor)
           ret

LOAD       inc  d
           ex  af, af'
           dec  d
           ld  a, 4+8
           out (254), a
           call #562     ; volej rutinu z ROM
           ld  a, 15     ; nastav bílý border a
           out (254), a ; a aktivuj EAR (3 bit)
           ret

```

```

LDHEAD      ld      ix,HEAD2          ; místo pro hlavičku
            ld      de,17            ; vždy délka 17 bytů
            xor     a                 ; LEADER u hlavičky je nula
            scf                     ; nastav CARRY - příznak LOAD
            call    LOAD              ; a proved' nahrání
            ex     af,af'            ; ulož CARRY
            ld     a,127             ; testuj
            in     a,(254)           ; klávesu
            rra                     ; SPACE (BREAK)
            ret     nc               ; je-li stisknuta, vrať se
            ex     af,af'            ; obnov CARRY
            jr     nc,LDHEAD         ; při chybě hledej další hlavičku

            call    TT               ; vytiskni
            defm   ' Found: '       ; co jsi přečetl

LDHEAD2     ld      hl,HEAD2+1
            ld      b,10
            ld      a,(hl)
            rst     16
            inc     hl
            djnz   LDHEAD2
            ld     a,32
            rst     16
            scf
            ret
            } vypiš nalezené jméno

TT          ld      a,2
            call   #1601
            } kanál

            ld      bc,71*256+47
            call   #22E5
            } udělej bod
            } na souřadnicích 47,71

            ld      de,256
            ld      bc,26*256
            call   #24BA
            } D=1 (směr Y nahoru) a E=0 (směr X)
            } po ose Y 26 bodů, po ose X žádný
            } nakresli čáru

            ld      de,1
            ld      bc,162
            call   #24BA
            } další čára půjde doprava
            } o 162 bodů
            } čára

            ld      de,255*256
            ld      bc,26*256
            call   #24BA
            } do D přijde -1 ... směr dolů
            } délka 26 bodů
            } čára

            ld      de,255
            ld      bc,162
            call   #24BA
            } do E zapiš -1 ... doleva
            } délka 162 bodu
            } čára

            ld      a,22
            rst     16
            ld      a,11
            rst     16
            ld      a,7
            rst     16
            } nastav tiskovou pozici

```



```

TT2      pop    hl
TT3      ld     a, (hl)
          and   127
          rst   16
          bit   7, (hl)
          inc  hl
          jr   z, TT3
          jp   (hl)
    } vytiskni text za CALLeM

TAPERROR call  TT
          defm "Tape loading"
          defm ' error'
    } nepřijemná zpráva

KEY      ei
          halt
          bit   5, (iy+1)
          jr   z, KEY
    } čekej na klávesu

          res   5, (iy+1)
          ld   a, (23560)
          cp   7
          jr   z, KEY2
          cp   13
          jr   z, KEY2
          cp   12
          jr   z, KEY2
          cp   32
          jr   c, KEY2
          cp   128
          jr   nc, KEY2
    } zruš signál
    ; přečti její kód
    } ponech jen platné kódy

KEY2     push  af
          ld   hl, #C8
          ld   de, #F
          call #3B5
          pop  af
          di
          ret
    } zapípej

A0LEN    equ   #-BEGIN
          ;

FREE     def s 6912
          ; vyhrazení místa

```

Začneme nejprve ukládáním na kazetu - SAVE. Podprogram začíná na adrese **#4C2**. Na začátku se uloží adresa **#53F (SALDRET)**, na které je se testuje stisk **BREAK** a v negativním případě následuje návrat do volání SAVE, v případě pozitivním pak výpis chybového hlášení **Tape loading error** a návrat do BASICu. Pokud pracujete se strojovým kódem, není obvykle tato varianta při stisku BREAKu žádoucí a proto se SAVE volá také od adresy **#4C6** - volaný podprogram se vždy vrátí zpět (viz příklad). Případný stisk BREAKu si samozřejmě musíte ošetřit sami.

Nyní k parametrům, které je nutno při vstupu zadat do registrů:

- IX** - adresa prvního bytu bloku dat, která mají být uložena
- DE** - délka bloku dat
- A** - LEADER - rozlišovací byte

Podprogram pro SAVE lze přerušit stiskem SPACE (BREAK).

Podprogram pro LOAD a VERIFY je společný - zvolená funkce se vybírá na vstupu podle hodnoty CARRY. Podprogram začíná na adrese #556 a podobně jako SAVE ukládá na zásobník adresu #53F (SALDRET). Chcete-li se vyhnout návratu přes test BREAKu (LOAD by bylo možno BREAKnout a tak se dostat do BASICu), musíte volat podprogram na adrese #562, bohužel uložení SALDRET na zásobník není hned na začátku a tak musíte opsat počáteční instrukce:

```
inc d
ex af,af'
dec d
di
ld a,7+8
out (254),a
call #562
```

Na vstupu vyžaduje podprogram pro LOAD a VERIFY stejné parametry ve stejných registrech jako SAVE. Navíc je nutno CARRY nastavit (scf) při LOAD a vynulovat (or a) při VERIFY. Nastavení CARRY při ukládání bloku (leader 255) lze udělat dvěma způsoby:

```
ld a,255          scf
scf              sbc a,a
```

První je běžný, druhý netradiční, je však kratší.

Po návratu obsahuje CARRY informaci o výsledku - je-li nastaven, je vše OK a nahrávání (verifikace) proběhlo bez chyby, je-li vynulován, pak to znamená buď že byla nalezena chyba nebo že byla stisknuta klávesa SPACE (BREAK). Dvě alternativy u návratu s chybou musíte rozlišit dodatečně tím, že ještě jednou otestujete SPACE (BREAK), podívejte se na příklad.

Při chybovém návratu můžete zjistit další informace o tom, kde k chyba došlo - testujte obsah **de** registru. Je-li v **de** původní hodnota, byla chyba v leaderu (jiná hodnota). Pokud tam naleznete nulu, jedná se o chybu v paritě. V ostatních případech došlo k výpadku během nahrávání.

O obou rutinách - SAVE a LOAD (VERIFY) - se ještě zmíníme podrobněji v některém dalším dílu příručky. Dozvíte se, jak vytvořit zcela vlastní loader, který bude kromě nahrávání provádět ještě další akce (psát text, pohybovat obrázkem, atd.).

V uvedeném příkladu jsou použity také podprogramy z ROM pro nakreslení bodu a čáry. Vysvětlíme si způsob, jak se zadávají parametry:

Podprogram PLOT - na adrese #22E5, parametry vstupují v **bc**. V registru **b** je souřadnice Y a v registru **c** souřadnice X. Souřadnice Y musí být v rozmezí 0..191, jinak bude hlášena chyba **Integer out of range**.

Podprogram DRAW - adresa #24BA. Na vstupu předpokládá v registru **bc** absolutní hodnoty posunu a v registru **de** pak znaménka posunů. Obdobně jako u PLOT se **b** a **d** týkají souřadnice Y a **c** a **e** souřadnice X. Znaménko je signalizováno takto; záporná čísla mají -1, nula je signalizována nulou a kladné číslo 1. Chcete-li provést ekvivalent **DRAW 100,-30**, musíte nastavit registry takto: **b=30, c=100, d=255, e=1** nastavovat můžete samozřejmě přímo registry, lepší (a hlavně kratší) je nastavovat dvojregistry najednou - viz příklad.

DOMLUVA

(Utvořeno podle vzoru "předmluva")

Nejprve k původu názvu této kapitoly. Nejprve jsem uvažoval o názvu **Závěr**, pak jsem si však uvědomil, že by mohlo dojít k nedorozumění - toto není závěr, knížka bude mít další díly. Možná Vás napadne, proč ji nevydáme najednou, důvody jsou jednoduché - další části se teprve připravují a chceme abyste měli možnost se s knížkou seznámit (kupujete pouze část zajíce v pytli). Nakonec jsem zvolil slovo "domluva", na které jsem byl přiveden svými přáteli (© 1991 Petr Koudelka, KoZa software), omlouvám se tímto autorovi za porušení jeho domnělých autorských práv na toto slovo.

Nyní už k vlastnímu obsahu této kapitoly:

Mám na Vás několik drobných (až nicotných) požadavků a proseb (nesplnění se trestá smrtí nebo vězením na 24 hodin nepodmíněně):

1) neberte předchozí, tento ani následující text příliš vážně (v rámci této kapitoly)!

2) čistěte si zuby alespoň dvakrát denně nejméně 3 minuty!

3) nepijte a nekuřte, nevysedávejte u počítače dlouho do noci, pořádně se vyspěte!

4) budete-li mít se strojovým kódem nějaké problémy, hledejte pomoc nejprve ve svém okolí, teprve pak se můžete obrátit na linku důvěry a teprve v poslední řadě pište nám. Nemůžeme odpovědět na všechno - nemáme křišťálovou kouli ani dostatek času. Navíc pomoc od kolegů přijde jistě rychleji než dopis od nás.

5) nedloubejte se v nose na veřejnosti, není to estetické! (sledujete to nenápadně a decentní výchovné působení? To je, co?).

6) obsah dalších dílů můžete ovlivnit tím, že nám zašlete nějaké připomínky a návrhy, pokusíme se je do dalších dílů zapracovat - neočekávejte však odpověď. Čtete Amatérský programátor a ZX Magazin, některé problémy se pokusíme ventilovat na jejich stránkách.

7) nechte se pojiistit u České státní (?) pojišťovny, budete mít radost!

8) při praní používejte jediné Ariel (evropská jednička) nebo Persil (Všetky Evropské..)

9) jediné kečupy Spak dodají Vašemu jídlu správný šmak!

10) případné gramatické chyby v této publikaci berte jako naše výchovné působení na Vás, kdo jich najde víc?!

11) pokud se Vám tato kapitola nelíbí, nic si z toho nedělejte, nám se líbí (**Úvodní blábol** z manuálu k DESKTOPU vzbudil u některých uživatelů negativní nálady - myslete si o nás co chcete, my už se nezměníme).

12) používejte ladící systém PROMETHEUS - je v mnoha ohledech nejlepší. Psal jsem jej sám a pro sebe (původně), vím co říkám. Potíže vzniklé použitím jiného assembleru nás nezajímají - max. počet symbolů a binární čísla (MRS), velikost zdrojového textu a "tajné" instrukce (GENS a jeho odrůdy), případně další - v textu je na ně občas upozorněno. Všechny ukázkové programy jsou napsány a vyzkoušeny na PROMÉTHEU, jestli přeci jen narazíte na chybu (velice nepravděpodobně, ale krk na to nedám), musela vzniknout při přepisování.

13) to by pro první díl stačilo (jste pověřiví?). Doufám, že již netrpělivě očekáváte další díly a těším se s Vámi na shledanou v dalším dílu

Obsah 1. dílu příručky ASSEMBLER A ZX SPECTRUM
.....

Stručně o assembleru	1
Příseme znaky	35
Výpis textů	51
Výpis čísel	64
Klávesnice na ZX Spectru	68
16-bitová aritmetika	78
Jednoduchý zvuk	81
Vstup a vyhodnocení textu	87
Kazetové operace	98
Domluva	105
Obsah 1. dílu příručky ASSEMBLER A ZX SPECTRUM	106

Sem si můžete napsat co Vás napadne:

Sem si můžete napsat co Vás napadne:

- Název knihy:** ASSEMBLER A ZX-SPECTRUM 1. díl
- Autor:** Tomáš VILÍM
- Vydal:** PROXIMA - software, post box 24, pošta 2, 400 21 Ústí nad Labem
- Vyšlo:** v lednu 1992
- Vydání:** první