edice C 64/128

Strojový jazyk

OBSAH

1. Tři cesty ke strojovému kódu

Každý vyšší programovací jazyk jako Basic či Pascal má své kořeny ve strojovém kódu. Tyto kořeny jsou tvořeny podprogramy. Dílčí podprogramy, které se starají o spolupráci mezi obrazovkou, klávesnicí, dalšími periferiemi a počítačem jsou ve strojovém kódu. Vlastní provádění např. programu v Basicu je pak záležitostí interpretu. Ten po startu programu vyhledává ke každému příkazu Basicu příslušný podprogram ve strojovém kódu, provádí ho a tak pracuje po celou dobu činnosti programu. Tato metoda je poněkud pomalá, protože každý pro každý příkaz Basicu musí být vždy volána příslušná rutina. Interpret i operační systém jsou programy ve strojovém kódu, které jsou uloženy v paměti počítače.

Další cesta ke strojovému kódu vede přes kompilátory. Kompilátor je strojový program, který překládá zdrojový program (i z Basicu) do strojového kódu. Ve srovnání s interpretem přináší výrazné zvýšení rychlosti, ale má i své záporné stránky. Zabírá v paměti místo a obvykle omezuje použití všech některých příkazů Basicu.

Třetí cesta je ta pravá - programování v assembleru, t.j. jazyku symbolických kódů, což jsou jen jinak vyjádřené instrukce strojového kódu. Následující kurs Vám pomůže vniknout do tajů programování v assembleru co nejrychleji, musíte ale přitom dodržovat alespoň tyto dvě základní pravidla:

1. porozumět teorii do detailů - každou kapitolu pochopíte jen tehdy, ovládáte-li předchozí.
2. všechny příklady programů napsat a provést - bez praxe je teorie k ničemu!

Příručka je napsána i pro začátečníky, předpokládá se pouze elementární znalost techniky mikropočitačů. Mohou ji použít i uživatelé jiných mikropočitačů osazených mikroprocesory 6502, 6510 nebo 7501 (Atari, BBC, C4, C16, C20, Oric, Laser), i když to budou mít trochu těžší - i když je soubor instrukcí stejný, tyto

počítače mají jiný operační systém a některé příklady bude nutné modifikovat.

Základ počítače - paměťová buňka

Každý počítač se skládá z několika základních částí. K nim patří: mikroprocesor - CPU (Central Processing Unit) a pracovní a pevná paměť. Pracovní paměť - RAM (Random Access Memory) - lze použít pro čtení i zápis, její obsah se po vypnutí napájení ztrácí. Pevná paměť - ROM (Read Only Memory) - je naproti tomu určena jen pro čtení, její obsah zůstává zachován i po vypnutí. V ROM C64 jsou uloženy programy operačního systému a interpretu Basicu.

Srdcem počítače je ale CPU - u C64 je to typ 6510, který má stejný soubor instrukcí jako 6502 a 7501. CPU provádí všechny početní operace a také řídící úkoly. Při početních operacích se ovšem provádí prakticky pouze několik základních operací s obsahem paměťovóch buněk. Buňky paměti obsahují buď data nebo programy. Každá buňka nese dvě informace - jedna je její obsah (číslomezi 0 až 255), a další je adresa, pod kterou je buňka volána. Adresa je u C64 dána číslem mezi 0 a 65535. Obsah je tedy dán jedním bajtem (8 bitů) a adresa dvěma bajty (16 bitů).

2. Uložení dat v paměti

Tato kapitola je určena pro začátečníky a popisuje systém uložení dat v paměti. Číselný obsah paměťové buňky lze vyjádřit více způsoby - dekadicky, hexadecimálně, binárně - podle toho, jak to zrovna potřebujeme. Jedna paměťová buňka u C64 má velikost 8 bitů, t.j. 1 byte a maximální číslo, které lze do ní uložit je 255 = (2) na 8. Každý bit totiž představuje jedno binární místo, které může nabývat jen dvou hodnot - 0 a 1. Když chceme znát dekadicky hodnotu binárního čísla, musíme přiřadit jednotlivým bitům váhové faktory - jsou to mocniny čísla 2 v závislosti na pořadí bitu zprava, počínaje 0. Tyto faktory pak sečteme pro bity s obsahem 1 a dostaneme číslo dekadicky.

Máme např. binární číslo 00100110:

bit è.:	7.	6.	5.	4.	3.	2.	1.	0.
bin.è.:	0	0	1	0	0	1	1	0
faktor:	128	64	32	16	8	4	2	1
dek.è.:	32		+		4 + 2		= 38	

Aby se binární čísla nepletla s dekadickými, označujeme je na začátku znakem %: %00100110.

Hexadecimální čísla mají za základ číslo 16 (podobně dekadická 10. binární 2). Poněvadž potřebujeme vyjádřit číslici ale pouze jedním znakem, a známe jenom 10 číslic, musíme si v tomto případě vypomoci ještě písmeny A, B, C, D, E a F. Hodnota odpovídající písmenu A je pak 10 a postupně až F=15. Hexadecimální číslice tedy nabývají hodnot 0 až 15, tedy 0 až F. Když si to dáme do souvislosti s jedním bajtem, který rozdělíme na poloviny (tzv. nibble), a uvědomíme si, že 4 bity nabývají hodnot 0 až 15, vidíme, že je to právě naše hexadecimální číslice 0 až F! Obsah 1 nibble (půlbajtu) lze tedy popsat jednou hexadecimální číslicí a obsah celého bajtu dvoumístným hexadecimálním číslem! A protože vše v počítači probíhá s čísly mocniny dvou, je snadnější používat pro většinu označování obsahu i adres paměťových buněk hexadecimálních čísel, než dekadických a binárních. Hexadecimální čísla se označují znakem $: $FF.

Převodní tabulka mezi dekadickými, binárnímia hexadecimálními čísly včetně jejich znakové podoby v ASCII, obrazovkovém kódu je v příloze této příručky.

3. Pomocné programy

Pro práci se strojovým kódem potřebujeme program, který by nám alespoň ukládal strojové instrukce na námi určená místa. Toto ukládání lze provádět i příkazy Basicu PEEK a POKE, kdy instrukce strojového kódu jsou v příkazech DATA. Takovému způsobu se říká basicovský zavaděč, je to ale pomalý a nevhodný způsob. Vhodnější způsob je použití programu typu Monitor, kterých existuje celá řada (MICROMON, SMON, XMON,

HEXAMAT). Jsou to pomocné programy (ve strojovém kódu), které slouží pro práci se strojovým kódem - a to nejen pro čtení a zápis, ale také pro převod z assembleru a do assembleru, různé přesuny, převody, hledání a i Load/Save vybrané části paměti. Běžné monitory mají základní instrukce shodné a liší se jen ve speciálních možnostech. Programy se volají příkazem SYS XXXXX, kde XXXXX je jejich startovací adresa.

Základní příkazy programů typu MONITOR :

A - Assembler - zápis instrukcí assembleru:

```
.A C000    LDA #$00
   adresa  instrukce
```

C - Compare - porovnání obsahu paměti:

```
.C 1000 2000 3000
   od   do   s čím
```

D - Disassembler - převod do assembleru:

```
.D C000 C020
   od   do
```

F - Fill - zaplnění paměti hodnotou:

```
.F 1000 2000 FF
   od   do   čím
```

G - Go - spuštění programu (jako SYS):

```
.G C000
   startadresa
```

H - Hunt - hledání v paměti:

```
.H 1000 2000 A9 12
   od   do   co
```

L - Load - nahrání do paměti:

```
.L"JMENO",01,C000
```

(Pozn.:tento příkaz mívá různé tvary) (č.zař.,adr)

M -Memory - výpis obsahu paměti:

.M 1000 2000

od do

S - Save - uchování paměti:

.S"JMENO", 01, C000, C300

č.zař. od do

(různé formáty - jako u Load)

T - Transfer - přesun obsahu paměti:

.T 1000 2000 4000

od do kam

X - Exit - ukončení a návrat do Basicu

- Převod čísla dekadického na hexadecimální

$ - Převod čísla hexadecimálního na dekadické

Pokud někdo chce použít program HEXAMAT, ten je odlišný a funkce uvedené v menu po jeho spuštění SYS49246 mají tento význam:

END - ukončení programu a návrat do Basicu

READ - výpis obsahu paměti od zadané 4-místné hexadresy

WRITE - zápis hexčísel od zadané hexadresy

START - spuštění programu od zadané hexadresy

SAVE#1 - uchování obsahu paměti na kazetu od-do hexadresy

SAVE#8 - uchování obsahu paměti na disk od-do hexadresy

@ - návrat do menu

Pro testování příkladů v této příručce si zvolíme dostupný monitor, který umožní použití oblasti RAM např. od adresy C300 - musí to být monitor, který sám leží v jiné oblasti paměti (např.: SMON8000 od adresy 32768, SMON9000 a PROFIMON od adresy 36774, MICROMON8192 od adresy 8192, a j.). Všechny příklady lze také psát pomocí vyššího typu programu pro práci s assemblerem, který umožňuje zápis instrukcí assembleru jako zdrojový program a mnoho dalších možností. Většina takovýchto programů

je ale diskově orientována. Jsou to programy: MERLIN64, PROFIASS, PAL a j. Programy, které pracují i s kazetou jsou např.: LADS, MES ASSEMBLER.

4. Střadač - Accumulator

Nyní se seznámíme s první instrukcí assembleru - je to instrukce RTS (Return from Subroutine), jejíž strojový kód je $60. Tato instrukce je určena pro zakončení programů ve strojovém kódu a odpovídá Basicovému příkazu RETURN.

Uložíme tuto instrukci na adresu $C300 pomocí monitorovského příkazu AC300 RTS a odstartujeme příkazem GC300. Co se stane? Monitor odstaruje program od adresy $C300, kde nalezne instrukci návratu, takže se hned vrátí zpět do menu. Formát celého zápisu programu na papír je následující:

adresa	obsah	návěští	instr.assembleru	poznámka
C300	60		RTS	,návrat

Adresa obsahuje vlastní adresu, na které začíná instrukce. Dále jsou uvedeny strojové kódy instrukce a jejího operandu. Strojové kódy instrukcí znát nemusíme, ty za nás dosadí monitor. Instrukce sama může mít různou délku - 1 až 3 bajty a skládá se z vlastního kódu instrukce a 1-2 bajtového operandu, t.j. označení dat, se kterými instrukce pracuje. Kolonka návěští slouží pro označování míst pro skoky v programu - pro zápis v monitoru se nedá použít, ale pomáhá nám při orientaci v programu. Pak následuje vlastní instrukce assembleru, což je symbolické označení strojové instrukce. Nakonec je možno uvést poznámku, co instrukce provádí.

Převody dat ve strojovém kódu

Mimo paměťové buňky v paměti RAM a ROM existují ještě i paměťové buňky přímo v procesoru - ty se nazývají registry. Nejuniverzálnější takový registr se nazývá ACCUMULATOR neboli

střadač, také registr A. Má rozsah jednoho bajtu, tedy 8 bitů. Základní instrukce pro práci se střadačem je **LDA** (Load Accumulator), která naplní střadač obsahem operandem určeného bajtu. Např. LDA #$F7 pošle do střadače $F7.

Další instrukcí pro střadač je **STA** (Store Accumulator), která uloží obsah střadače do bajtu opět určeného operandem instrukce. Například STA $C500 - operandem zde není konstanta ale adresa, na kterou se má uložit obsah střadače. Kódování instrukce je takové, že první bajt je opět kód instrukce a další dva bajty obsahují operand - adresu $C500, která je rozdělená do dvou bajtů - v prvním je nižší část adresy ($00) a v druhém vyšší část adresy ($C5). Tento systém zápisu adresy, nižší - vyšší se používá u všech instrukcí. Těmto adresovým bajtům se také říká Low-Byte a High-Byte. V dalším textu budou jejich obsah označován jako $LL a $HH. Celá adresa pak jako $HHLL. Náš první malý program ve strojovém kódu bude mít za úkol změnit barvu pozadí a okraje obrazovky na černou. Informace o těchto barvách jsou u C64 uloženy na adresách 53280 a 53281 ($D020 a $D021). Mají-li být obě tyto barvy černé, stačí v Basicu napsat:

```
POKE 53280,0:POKE 53281,0
```

Program v assembleru by vypadal takto:

```
C300  A9 00     LDA #$00
C302  8D 20 D0  STA $D020
C305  8D 21 D0  STA $D021
C308  60        RTS
```

Nejdříve se uloží příkazem LDA #$00 kód 0 pro černou barvu do střadače (registru A). Pak se příkazem STA přenese tento kód barvy na adresy D020 a D021. Na konci programu je instrukce RTS, která se postará o návrat ze strojového programu. napsání a vyvolání tohoto programu od adresy $C300 pak provede změnu barev. Druhy adresace

Podle toho, jaký druh operandu následuje za kódem instrukce, mluvíme o druhu adresování, poněvadž operand nám vlastně určuje adresu, se kterou instrukce pracuje. Pro každý druh adresace je stanoven přesný formát zápisu operandu. K

jednotlivým druhům adresace se dostaneme postupně, nyní jen jejich přehled a způsoby zápisu:

Přímé adresování - #$NN - operand přímo obsahuje konstantu se kterou instrukce pracuje

Absolutní adresování - $HHLL - operandem je adresa bajtu, se kterým se bude pracovat $LL - pro nulovou stránku (ZP,HH=00)

Indexované adresování - $HHLL,X - bajt, se kterým se bude pracovat, je na adrese $HHLL+X (platí i pro indexregistr Y a ZP)

Nepřímé absolutní adresování - ($HHLL) - bajt, se kterým se bude pracovat je na adrese, jejíž lowbyte je obsahem adresy v operandu

Nepřímé předindexové adresování - ($LL,X) - viz kap.13

Nepřímé poindexové adresování - ($LL),Y - viz kap.13

5. Změna obsahu buněk a větvení

Obsah paměťové buňky lze změnit pomocí instrukcí **INC** (Increment Location) a **DEC** (Decrement Location). INC $HHLL zvětší obsah adresy $HHLL o 1, INC $HHLL zmenší obsah adresy $HHLL o 1. Pokud je obsah roven $FF (resp. $00), pak po zvýšení (resp. snížení) bude obsah roven $00 (resp. $FF).

Větvení pomocí podmíněných skokových příkazů

V programech se používá instrukcí DEC a INC jako čítačů. Při určité hodnotě čitače pak provádíme odskok na jiné místo v programu (IF... THEN GOTO). K tomu nám slouží instrukce podmíněných skoků. Dvě základní instrukce jsou **BEQ** (Branch on Equal) - skok, pokuď výsledek předchozí operace byl = 0, a **BNE** (Branch on Not Equal) - skok, pokuď výsledek nebyl = 0. Za operačním kódem instrukce je operand, kterým ale není adresa na kterou se má skákat, ale počet bajtů, které se mají přeskočit. A to jak dopředu ($00 až $7F), tak i dozadu ($80 až $FF). Je to vlastně

relativní adresování - vzdálenost skoku je ale omezena na +127 a -128 bajtů.

Hexadecimální kruhový čitač

Při každém průchodu programem se zvýší stav čitače o 1, přičemž čítá do hodnoty $FFFF. Skládá se tedy ze dvou paměťových buněk, a to na adresách $C300 a $C301. Jsou označeny jako LOW (čitač nižšího řádu) a HIGH (čitač vyššího řádu). Počáteční hodnota čitače bude $00FD. Po každém vyvolání programu od adresy $C302 se zvýší obsah dvoubajtového čitače o 1:

```
C300  00 HIGH
C301  FD          LOW
C302  EE 01 C3    INC LOW
C305  F0 01       BEQ ZERO
C307  60          RTS
C308  EE 00 C3    ZERO INC HIGH
C30B  60          RTS
```

Vlastní program začíná na adrese $C302 - tam INC zvyšuje hodnotu nižšího řádu čitače o 1. Pokud se přitom jeho hodnota změní na $00, provede se instrukce BEQ, která skáče na adresu označenou ZERO ($C308). Protože je třeba přeskočit jeden bajt, je hodnota operandu za BEQ rovna $01. Pak program zvýší hodnotu vyššího řádu čitače a nastane návrat. K návratu dojde i pokud nenastane odskok na ZERO. Větvení programu lze provést i instrukcí BNE. Program se pak změní takto:

```
C300  00          HIGH
C301  FD          LOW
C302  EE 01 C3    INC LOW
C305  D0 03       BNE NOZERO
C307  EE 00 C3    INC HIGH
C30A  60          NO ZERO RTS
```

Uvedené zápisy programu obsahují důležitou pomůcku - návěští. Tyto návěští nahrazují nepřehledné adresy a označují místa skoků. Při použití assembleru MERLIN se takto označené relativní adresy nahradí při překladu do strojového kódu skutečnými adresami.

6. Vstup a výstup dat

Nyní se seznámíme s programy, které čtou stav klávesnice a zobrazují data na obrazovce. Domácí počítače používají většinou pro zobrazení znaků ASCII kód. Tento kód předepisuje binární kódování jednotlivých znaků. Standartní tabulka ASCII obsahuje 128 znaků, tabulka pro C64 obsahuje ještě další znaky (viz příloha).

Klávesnice jako zdroj dat

Po stisku klávesy je odpovídající ASCII kód uložen operačním systémem do vyrovnávací paměti klávesnice (bafru). Tento bafr má kapacitu 10 bajtů a lze jej využít pro vstup dat v programech v assembleru. Pokud chceme přenést obsah prvního bajtu z bafru klávesnice, použijeme k tomu rutinu z operačního systému. Operační systém obsahuje řadu užitečných rutin, které můžeme využívat ve vlastních programech. Nyní použijeme rutinu, začínající na adrese $F13E, která přenáší první znak z bafru klávesnice do střadače. V případě, že v bafru není žádný znak, uloží se do střadače hodnota $00.

Rutiny se volají instrukcemi **JSR** (Jump to Subroutine) nebo **JMP** (Jump). Příkaz JMP $HHLL provede skok na adresu $HHLL a to tak, že tuto adresu do zvláštního registru, zv. programový čítač (Program Counter - PC registr). Tento registr má délku 2 bajty a obsahuje vždy adresu právě prováděné instrukce. Při relativních skocích (BNE, BEQ a j.) se k obsahu PC registru přičítá hodnota relativního skoku.

Instrukce JSR $HHLL provede mimo skoku na adresu $HHLL také návrat zpět do hlavního programu hned za instrukci JSR. Procesor si totiž zapamatuje adresu návratu - uloží si ji do zásobníku (Stack). Příklad uvádí program, který čeká na stisk klávesy a pak zapíše její kód do paměťové buňky MEMORY:

```
INCHAR EQU $F13E
C300  00          MEMORY
C301  20 3E F1    WAIT JSR INCHAR
C304  0D 00 C3    STA MEMORY
C307  AD 00 C3    LDA MEMORY
C30A  F0 F5       BEQ WAIT
C30C  60          RTS
```

Po startu od adresy $C301 nastane ukončení programu až po stisku libovolné klávesy, jejíž kód pak bude uložen na adrese $C300. Systémová rutina je zde označena návěštím INCHAR. Po návratu z této rutiny je v A uložen kód klávesy, který se dále uloží do buňky MEMORY. Tento kód se znovu přesune zpět do střadače, a to pro následný test na nulu (BEQ). Nevíme totiž, jaký výsledek měla poslední operace systémové rutiny INCHAR. Má-li kód klávesy hodnotu $00, je touto instrukcí LDA zaručeno splnění podmínky pro skok BEQ.

Výstup dat na obrazovku

Aby se nějaký znak objevil na obrazovce, musí být jeho obrazovkový kód (není totožný s ASCII kódem! viz tabulka) uložen v paměti obrazovky. Tato paměť je dlouhá 1000 bajtů a standartně leží od adresy $0400 do $07E7. Napsání písmene na začátek 1.řádku obrazovky v Basicu lze provést příkazem POKE 1024,1. Podobně to lze udělat prostým uložením ve strojovém kódu, ale přitom nelze použít řídící znaky a navíc musíme znát obrazovkové kódy znaků. Proto je pro tento účel vhodnější použít podobně jako pro vstup znaku z klávesnice rutinu z operačního systému. Tato rutina leží od adresy $E716 a píše na obrazovku znak, jehož ASCII kód je před vyvoláním rutiny obsažen ve střadači.

Celý program spolu s předchozím vypadá takto:

```
                OUTCHR EQU $E716
                INCHAR EQU $F13E
C300  00 MEMORY
C301  A9 93       LDA #$93
C303  20 16 E7    JSR OUTCHR
C306  A9 41       LDA #$41
C308  20 16 E7    JSR OUTCHR
C30B  A9 42       LDA #$42
```

```
C30D  20 16 E7     JSR OUTCHR
C310  A9 43        LDA #$43
C312  20 16 E7  JSR OUTCHR
C315  20 3E F1     WAIT JSR INCHAR
C318  8D 00 C3     STA MEMORY
C31B  AD 00 C3     LDA MEMORY
C31E  F0 F5        BEQ WAIT
C320  60           RTS
```

7. Dvě speciální stránky paměti

V minulé kapitole jsme se seznámili s instrukcí JSR. Uvedli jsme si také, že při jejím provedení si procesor ukládá adresu návratu. Co se ale stane, když i podprogram bude volat další podprogram? Jak to bude s uložením další adresy? Odpověď zní: celkem je možné do sebe začlenit volání až 128 podprogramů, k tomu je zapotřebí si uložit celkem 128 adres, což představuje 256 bajtů. Protože sám procesor žádnou takovou paměť v sobě nemá, používá se pro tento účel zvláštní oblast paměti RAM, která se nazývá zásobník (Stack). S touto pamětí se pracuje tak, že z ní jde jako první ven ten bajt, který se uložil jako poslední (opak zásobníku klávesnice!). Je to tzv. metoda LIFO (Last In - First OUT), u klávesnice to byla metoda FIFO (First In - First Out). Tím je zaručena správná posloupnost adres návratů z jednotlivých podprogramů.

Zásobník procesoru má vyhrazenou určitou část paměti v délce 256 bajtů, t.j. právě jedné stránky (termín jedna stránka vychází z hexčísla vyššího bajtu adresy - ten když se zvýší o 1, tak vzroste adresa o 256 bajtů). Zásobník má k dispozici 1.stránku paměti - leží tedy od adresy $0100 do $01FF (256 až 511 dek.). Aby procesor věděl, kam uložil poslední bajt do zásobníku, má k tomu speciální registr - ukazatel zásobníku (Stack Pointer - SP registr). Tento jednobajtový registr obsahuje nižší část adresy vrcholu zásobníku (vyšší část adresy je totiž konstanta - $01 - takže ji nemusíme ukládat). Při prázdném zásobníku je to $FF

(odpovídá adrese $01FF), zásobník se zaplňuje směrem dolů - při plném zásobníku je to $00 (odpovídá adrese $0100).

Volání podprogramu probíhá takto: JSR $HHLL vytvoří nejdříve adresu návratu tak, aby se procesor vrátil za tuto instrukci. Zvětší tedy obsah PC registru o 2 a uloží tento jeho nový obsah na vrchol zásobníku. Pak do PC registru vloží adresu $HHLL a začne provádět podprogram. Jakmile narazí na instrukci RTS, vyzvedne posledně uloženou adresu ze zásobníku, uloží ji do PC registru a tím se vlastně vrátí za původní volání instrukcí JSR. Současně se také zvýší obsah ukazatel zásobníku.

Proč nás ale zajímá, co dělá sám procesor bez našeho vlivu? Protože i my můžeme používat zásobník a to pro krátkodobé uložení dat nebo obsahu registrů. Máme k tomu instrukce **PHA** (Push Accumulator), která uloží obsah střadače do zásobníku, a **PLA** (Pull Accumulator), která obráceně zase naplní střadač ze zásobníku.Obě instrukce změní správně i stav SP registru. Obě tyto instrukce musí být ale používány s nejvyšší opatrností! Programátor se musí postarat o to, aby si procesor vždy vyzvedl ze zásobníku bajty, které zrovna potřebuje (např. pro návrat - jinak dojde ke skoku do neznáma). Platí jedno pravidlo, že každá instrukce PHA musí mít k sobě instrukci PLA (pokud se jedná o ukládání dat do zásobníku).

Mimo stránky 1 je v paměti ještě jedna zvláštní stránka - stránka 0 neboli Zero Page (ZP) - adresy $0000 až $00FF. Tato stránka je vyhrazena pro důležitá data operačního systému. Navíc je v ní možné adresovat pouze jedním bajtem a proto existuje spousta instrukcí s tímto druhem adresování - adresování Zero Page. Tyto instrukce jsou totiž kratší, protože operandem je jednobajtová adresa. Př.: LDA $LL. Popis obsahu stránky 0 je uveden v mapě paměti C64 (dodatek Q základní příručky). Pro programátora je důležité vědět, které adresy může použít ve svých programech. Volné adresy pro uživatele jsou $02, $FB až $FE a někdy i další.

8. Příznakové signály pro CPU

Již v předchozích kapitolách jsme využívali instrukce pro větvení BEQ a BNE, které v závislosti na poslední operaci provádějí větvení (nebo ne). Jak pozná procesor, že ve střadači je hodnota $00? Případ, kdy je ve střadači hodnota 0, je oznámen tzv. příznakem. CPU nastaví hodnotu příznakového bitu Z (Zero-Flag) na "1". Při výsledku různém od nuly je hodnota Z bitu "0". Pojem flag-praporek je třeba chápat jen obrazně. Ve skutečnosti jsou všechny praporky jen bity, které jsou umístěny přímo v mikroprocesoru a to v tzv. statusregistru. Každý bit představuje jeden příznak. Má-li bit hodnotu "1", znamená to nastavený příznak. Bit se signálním stavem "0" znamená sesazený příznak. Ve statusregistru 6502/6510 je místo pro 8 příznaků. 5. bit je ale nevyužitý a stále nastavený (hodnota "1"). 1. bit je již zmíněný nulový příznak (Zero-flag). Instrukce LDA ,INC , DEC s hodnotou 0 automaticky nastavují Zero - flag. Jiné operace jako STA, JSR nebo RTS neovlivňují zase žádný příznak ve statusregistru. Je-li potřeba, je možné statusregistr (neboli registr příznaků) uložit do zásobníku (STACK) instrukcí **PHP** (Push Processor status). Obrácená instrukce pro přesun za zásobníku do statusregistru je **PLP**.

Srovnávání - compare

Až doposud jsme mohli ve strojovém programu provádět větvení jen když byl výsledek "0", nebo když byl různý od nuly. Co se ale musí udělat, když má procesor provést větvení při výsledku $E4? Neexistují totiž instrukce typu "Branch on $E4". Přesto lze i podle takového výsledku provést větvení. Instrukce CMP #$NN porovná obsah střadače s bajtem o obsahu $NN. Souhlasí-li obsahy obou položek, nastavuje procesor příznakový bit Zero-Flag na "1". Při nerovnosti tento příznak sesadí ("0"). Bezprostředně poté může proběhnout větvení instrukcemi BNE nebo BEQ. Instrukce **CMP** jednoduše vytvoří rozdíl obsahu střadače s datovým bajtem.

Souhlasí-li obě hodnoty, potom je jejich rozdíl roven "0". Proto procesor nastaví nulový příznak (Zero-Flag) na "1". Je-li rozdíl obou položek různý od nuly, je tento příznak sesazen. Srovnávání lze provádět s absolutní adresací i adresací v nulové stránce. Jak lze instrukci CMP využít, ukazuje následující program:

```
                    OUTCHR EQU $E716
                    INCHAR EQU $F13F
C300  A9 93          LDA #$93
C302  20 16 E7       NEXT JSR OUTCHAR
C305  20 3F F1       JSR INCHAR
C308  C9 BD          CMP #$8D
C30A  D0 F6          BNE NEXT
C30C  60             RTS
```

Počitač zde pracuje jako psací stroj tak dlouho, až přijde současný stisk tlačítek SHIFT a RETURN. Pak se program ukončí. Program začíná jako obvykle na adrese $C300.

Nejprve je naplněn střadač hodnotou $93 pro smazání obrazovky. Pak proběhne rutina tisku znaku na obrazovce OUTCHR vyvolaná z adresy $C302 a potom se provede rutina INCHAR, která přenese znak bafru klávesnice do střadače. Tento znak je porovnán instrukcí CMP s kódem $8D pro současný stisk tlačítek SHIFT a RETURN. Jestliže jsou oba znaky shodné, nastaví procesor Zero-Flag na "1" a větvení neproběhne a je jako další provedena instrukce RTS a tím i ukončení programu. V případě nerovnosti sesadí procesor Zero-Flag a následující instrukce BNE provede větvení - odskok na návěští NEXT - procedura začíná znovu. Co se ale stane, když není stisknuto žádné tlačítko? V bafru bafru klávesnice není žádný znak a INCHAR vrací do střadače hodnotu $00. Zero-flag bude 1, nastane skok na návěští NEXT, ale OUTCHR rutina nevypíše na obrazovku žádný znak (=00).

9. Indexregistr - pseudostřadač

Registr není nic jiného než 8-mi nebo 16-bitová paměťová buňka, která je umístěna přímo v procesoru, nikoliv v RAM. Kromě střadače (accumulator) známe již registr-ukazatel zásobníku (stack pointer), čítač instrukcí (program counter) a registr příznaků (statusregistr). Navíc máme k dispozici ještě dva registry - indexregistry X a Y. Nyní je sada registrů 6502/6510 kompletní.

Vlastnosti indexregistrů

S indexregistry lze provádět běžné operace typu Load-, Store-, Compare-, inkrementu i dekrementu. Pak existují instrukce přenosu dat mezi střadačem a indexregistry. Tak se plní např. instrukcí **TAX** obsah indexregistru X obsahem střadače a **TYA** přesune obsah indexregistru Y do střadače. Právě tak se může stav ukazatele zásobníku přenést do X registru a opačně.

Proti jiným procesorům je 6502 se svými šesti registry méně vyzbrojen, ale tuto nevýhodu snadno vyrovnává velmi učinným systémem adresace. A přitom hrají právě indexregistry důležitou roli.

Zvláštní vlastnosti indexregistrů

Obsah indexregistrů může být využit podobně jako velikost skoku u relativních skoků. U relativního skoku přičte procesor známý rozdíl ke známému stavu čítače instrukcí a získá tak absolutní cílovou adresu. Při indexované adresaci přičte procesor obsah indexregistru k předem zadané absolutní adrese a získá tak konečnou adresu. K tomu hned příklady. Instrukce LDA $C300, X plní střadač obsahem paměti, jejíž adresa je sumou čísla $C300 a obsahu indexregistru X. Analogicky k tomu uloží instrukce STA $C300,Y obsah střadače do paměťové buňky, jejíž adresa vznikne sumou $C300 a obsahem indexregistru Y. Kdyby se v době

provádění instrukce STA $C300,Y nacházela v Y-registru hodnota $04, pak by se vlastně prováděla kvaziinstrukce STA $C304.

Tím jsme si ukázali, proč se X a Y registry nazývají indexregistry. V závislosti na jejich obsahu lze získat rozdílné adresy v okolí zadané absolutní adresy. Load instrukce platí také pro indexregistry samotné a to i s jiným indexregistrem pro indexovanou adresaci. Pak instrukce LDX $C300,Y plní indexregistr X obsahem paměťové buňky, jejíž adresa je dána sumou čísla $C300 a obsahu registru Y. Ale tímto způsobem adresace nelze přenášet obsah indexregistru do paměti. Naproti tomu lze indexovanou adresaci aplikovat také na aritmetické instrukce CMP, INC a DEC.

Snadné přesuny textů

Následující program řeší úkol napsat pomocí indexovaného adresování na obrazovku slovo COMMODORE:

```
                    OUTCHR EQU $E716
                    INCHAR EQU $F13E
C300  93 43 4F       TEXT DFB 147,"COMMODORE"
C303  4D 4D 4F
C306  44 4F 52
C309  45
C30A  A2 00          LDX #$00
C30C  BD 00 C3  TEXOUT LDA TEXT,X
C30F  20 16 E7  JSR OUTCHR
C312  E8             INX
C313  E0 0A          CPX #$0A
C315  D0 F5          BNE TEXOUT
C317  20 3E F1       WAIT JSR INCHAR
C31A  C9 00          CMP #$00
C31C  F0 F9          BEQ WAIT
C31E  60             RTS
```

Nejprve jsou přiřazeny našim starým známým rutinám z ROM názvy INCHAR, OUTCHR. ASCII kódy slova COMMODORE včetně úvodního kódu pro mazání obrazovky (147=CLR/HOME) uložíme do desíti paměťových buněk od adresy $C300 do $C309 (označeno TEXT). Potom naplníme instrukcí LDX registr X hodnotou $00 a smyčka pro výpis textu na obrazovce TEXTOUT může začít.

Nejprve se naplní střadač hodnotou z adresy TEXT+X (LDA TEXT,X). Protože obsah registru X je $00, obdrží střadač kód pro mazání obrazovky, který ROM rutina OUTCHR uvede ve skutečnost. Hned potom se zvýší obsah indexregistru X o 1 a porovná se obsah indexregistru X s hodnotou $0A (10 dek) a tím se testuje, zda již byl zpracován poslední z deseti ASCII kódů. Pokud ještě nenastal konec, následuje skok na návěští TEXOUT. To znamená, že vystoupí na obrazovce první znak. Podobně vystoupí všechny znaky slova COMMODORE. Až zpracuje počítač poslední znak, nenastane skok na návěští TEXOUT, ale počítač čeká v čekací smyčce s návěštím WAIT na vstup jakéhokoliv znaku z klávesnice a tím zajistí návrat zpět.

Samozřejmě lze programově přivést na obrazovku také jiné, delší texty. Musí se ale místo adresy prvního znaku uložené v buňkách $C30D (LB-nižší bajt adresy) a $C30E (HB-vyšší bajt adresy) zadat adresa, od které jsou uloženy kódy zobrazovaného textu. Tak jako se musí odpovídajícím způsobem změnit délka zobrazovaného textu na adrese $C314.

Mazání řádků obrazovky

Následující program způsobí vymazání horního řádku obrazovky:

```
                    SCREEN EQU $0400
C300  A0 00         LDY #$00
C302  A9 20         LDA #$20
C304  99 00 04      NEXT STA SCREEN,Y
C307  C8            INY
C308  C0 28         CPY #$28
C30A  D0 F8         BNE NEXT
C30C  60            RTS
```

Adresa první buňky paměti pro obrazovku má název SCREEN ($400=1024 dek.). Od adresy $C300 začíná pak vlastní program s využitím indexregistru Y jako počítadla pro určení buňky v obrazové paměti. paměti. Ve střadač je kód mezery - $20. Pro smazání znaku naplní počítač paměťovou buňku obrazovkové RAM SCREEN+Y kódem mezery a zvýší čítač (registr Y) o 1. Pokud není

všech 40 míst v obrazovkové RAM zaplněno kódem mezery následuje skok na návěští NEXT, jinak je provedena instrukce RTS - návrat z programu.

Aby byla vidět názorně funkce programu, je lépe jej odstartovat z Basicu pomocí povelu SYS49920 (49920=$C300), např. takto:

```
10 PRINT CHR$(147)
20 FOR N=0 TO 79:PRINT "A":NEXT
30 FOR V=0 TO 1000:NEXT:SYS49920
```

Velmi užitečným vedlejším účinkem povelu SYS je to, že se při jeho provádění automaticky ukládá do registrů obsah určité části RAM. Tak ukládá počítač při vyvolání SYS - povelu obsah adresy 780 do střadače, 781 do X - registru, 782 do Y - registru a obsah adresy 783 do statusregistru. Když se předem tato část RAM naplní konkrétními hodnotami, procesor si je pak převezme.

Příklad: POKE 780,160:POKE 782,0:SYS49924 - smaže se vrchní řádek obrazovky (= je naplněn bílými políčky). Zde se vyvolává program až od adresy $C304.

10. Logické operace

Každý lepší procesor má ve svém souboru instrukcí logické operace mezi obsahy paměti a registru procesoru. 6502/6510 tak logicky má takové instrukce pro bity střadače a bity adresovatelné paměťové buňky. Například instrukce ORA #$NN logicky spojuje střadač přes logické NEBO s dtovým bajtem $NN. Další instrukce jsou pro logické operace AND a EOR (exclusive-or). Jak tyto operace s jednotlivými bity probíhají, ukazuje logická tabulka:

OR			AND			EOR		
vstup		výstup	vstup		výstup	vstup		výstup
0	0	0	0	0	0	0	0	0
0	1	1	0	1	0	0	1	1
1	0	1	1	0	0	1	0	1
1	1	1	1	1	1	1	1	0

Manipulace s bity ve střadači

Na první pohled lze sotva zpozorovat význam logických operací. Jejich prvořadým úkolem je dosáhnout toho, aby bylo možné nastavit jednotlivé bity střadače na "1", na "0", nebo je invertovat. Povel ORA nastaví např. příslušné bity ve střadači na "1", pokud byly na "1" nastaveny bity operandu. Ke snazšímu pochopení hned jeden příklad: ORA #%00001111 nastaví ve střadači bity 0-3 na "1" a ponechává bity 4-7 v jejich původním stavu. Operace NEBO nastaví výsledný bit na "1", pokud aspoň jeden vstupní bit je "1". Tím jsou ve střadači na "1" nastaveny všechny bity, které jsou v datovém bajtu nastaveny na "1", zatímco nenastavené bity ("0") v datovem bajtu (operandu) nevyvolají žádnou změnu.

ORA #11111111 by nastavilo všechny bity střadače na "1", zatímco ORA #%00000000 by zůstalo bez účinku na nastavení bitu střadače. Naproti tomu operace AND sesadí všechny bity střadače (z "1" na "0"), které byly v operandu nastaveny na "0". Příklad: AND #%00001111 sesadí bity 4-7 ve střadači a všechny ostatní (0-3) ponechá v původním stavu. Učinek AND spočívá v tom, že výsledný bit je jen tehdy nastaven na "1", když jsou oba vstupní bity "1". Proto zapříčiní všechny sesazené bity v datovém bajtu sesazení všech bitů ve střadači, zatímco bity nastavené na "1" nic nezmění: AND #%00000000 sesadí všechny bity ve střadači, AND #%11111111 zůstává bez účinku na obsah střadače.

Třetí a poslední logické spojení je EOR (exclusive-or). Všechny nastavené bity v operandu invertují odpovídající bity ve ve ve střadači: EOR #%00001111 invertuje bity 0-3 ve střadači. Neboť, jak ukazuje tabulka, nastaví EOR výsledný bit vždy, když oba vstupní bity mají rozdílné stavy. Nastavený bit v operandu tedy způsobí změnu tehdy, když je odpovídající bit střadače "1" na "0" a když je bit střadače "0" nastaven na "1". Nenastavené bity operandu nemají žádný vliv na střadač.

ORA	AND	EOR

støadaè:	11000011	11000011	11000011
operand:	00001111	00001111	00001111
výsledek:	11001111	00000011	11001100

Hra s písmeny

Logické povely se používají např. k vytváření různých efektů na obrazovce. Vyzkoušíme si nyní zobrazení textu v normálním a reverzním módu. Rozdíl mezi těmito módy spočívá jen v jediném bitu č.7 v kódu příslušného znaku. Je-li tento bit nastaven ("1"), pak se objeví na obrazovce písmeno v reverzním módu, jinak, je-li bit 7 sesazen, pak se jedná o normální mód. Strojový program pracuje jen s tímto jediným bitem:

```
                    SCREEN EQU $0400
C300  A2 00         LDX #$00
C302  BD 00 04      REPEAT LDA SCREEN,X
C305  09 80         ORA #%10000000
C307  9D 00 04      STA SCREEN,X
C30A  E8            INX
C30B  E0 28         CPX #$28
C30D  D0 F3         BNE REPEAT
C30F  60            RTS
```

PRINT CHR$(147) "SLOVO 1",CHR$(18)"SLOVO 2"

Strojový program po spuštění přes SYS49920 zobrazí znaky, zapsané normálně, v reverzním módu.

Ve strojovém programu se první adresa textu bude jmenovat SCREEN. Registru X (počítadlu) přiřadíme hodnotu $00. Abychom změnili znak na adrese SCREEN+X, obdrží střadač kopii obsahu této buňky ($C302) a ORA #%10000000 nastaví bit 7. Dále se tato hodnota ze střadače přenese zpět do obrazovkové paměti ($C307), počítadlo (X-registr) se zvýší o "1" a porovná se, zda bylo dosaženo poslední buňky v obrazovkové RAM ($C30B). Následující větvení se provede tehdy, když tento případ nenastal, jinak se vrátí zpět do Basicu.

Samozřejmě lze použít i dvou zbylých logických operací AND a EOR. Vyzkoušejte si sami, co se potom s textem stane.

11. Spojovací článek - Carry-Flag

Je-li výsledkem operace hodnota $00, nastaví CPU Zero-Flag ve statusregistru. Dalším příznakem je příznak přenosu - Carry- Přenos vzniká v matematice vždy tehdy, když nestačí počet platných cifer k úplnému zobrazení čísla. Když se sečítají čísla 5 a 7, dostaneme 2 a přenos do desítkového místa, výsledek je 12. Přesně tak to probíhá i u hexadecimálních čísel - $3A plus $E7 dává $21 a přenos do vyššího řádu.

Jak signalizuje Carry-Flag

Přenos u C64 se neztratí, je signalizován příznakovým bitem č.0 ve statusregistru. V případě přenosu (např. při sčítání) nastaví procesor příznak (Carry-Flag) na "1", jinak je bit sesazen na "0". Právě opačně to vypadá s odečítáním: $3A minus $E7 dává výsledek $53 a jeden vypůjčený přenos z nejvyššího řádu. V takových případech signalizuje procesor příznakem "0" případný přenos a příznakem "0" opak.

Carry-Flag se používá převážně pro aritmetické operace s čísly libovolných velikostí (řádů). Přesně pro tento účel jsou k dispozici obě aritmetické operace (instrukce): sečítání ADC (Add with Carry neboli sečítání s přenosem) a odečítání SBC (Subtract with Carry neboli odečítání s přenosem).

Instrukce ADC #$NN sečítá např. datový bajt $NN plus Carry-Flag s obsahem střadače, výsledek ponechá ve střadači a případný přenos ukládá zase do Carry-Flag. Naproti tomu instrukce SBC #$NN odečítá bajt $NN, jakož i invertovaný Carry-Flag do obsahu střadače a zapisuje případný přenos do Carry-Flag. Pokud je přenos nežádoucí, pomohou dvě instrukce s implicitní adresací SEC (Set Carry neboli nastavení Carry-Flag), která nastaví Carry-Flag na "1". A instrukce CLC (Clear Carry neboli sesazení přenosu) sesadí příznak přenosu na "0". Máme i instrukce větvení podle Carry-Flag. Prostřednictvím BCS (Branch on Carry Set) větví

procesor při nastaveném Carry-Fiag, při BCC (Branch on Carry Ciear) následuje větvení při sesazeném Carry- Flag.

Dodatky k programům

Komfortní modifikace jazyku Basic umožňují uživatelům povelem APPEND přihrávání dalších Basic-programů bez toho, aby se smazal dosavadní program v paměti. Také uživatel C64 může prostřednictvím několika instrukcí ve strojovém kódu využít této možnosti. Princip je velmi jednoduchý. Dva ukazatele v nulté stránce obsahují adresy začátku a konce Basic-paměti. Jako ukazatel se označují dvě za sebou jdoucí paměťové buňky, které podle hesla nižší bajt napřed obsahují adresu. Když zaměníme konec Basic-programu za začátek, musí C64 vlastně povelem LOAD nahrát nový program za konec starého programu. Ukazatel na začátek RAM pro Basic a konec této oblasti paměti (= začátek paměti proměnných) jsou k nalezení pod adresami $2B až $2E (viz dodatek Q manuálu C64). Adresa začátku paměti proměnných udává potom (o dvě snížena) skutečný konec paměti pro Basic, tedy příslušnou adresu, od které se může přihrát další program. Hlavním úkolem dalšího programu je tedy zjistit tuto počáteční adresu Basic RAM, uložit ji do nulté stránky a připojit další program v Basicu:

```
                BASIC EQU $002B
                VARIA EQU $002D
                LOAD EQU $E168
C300  38          APPEND SEC
C301  A5 2D       LDA VARIA
C303  E9 02       SBC #$02
C305  85 2B       STA BASIC
C307  A5 2E       LDA VARIA+1
C309  E9 00       SBC #$00
C30B  85 2C       STA BASIC+1
C30D  4C 68 E1    JMP LOAD
```

V programu je nastaven Carry - Flag na "1" instrukcí SEC, což znamená žádný přenos pro následující instrukci SBC. Následující instrukce (na adrese $C301) přenese do střadače nižší bajt adresy začátku proměnných. Instrukce SBC #$02 sníží obsah

střadače o 2. Když by si přitom musel procesor půjčit přenos, sesadí tím Carry-Flag!!! Po uložení obsahu střadače nižšího bajtu začátku Basic-RAM ($C305) obdrží střadač vyšší bajt adresy začátku paměti proměnných. Odečítání SBC #$00 je potom jen formalita ale důležitá, protože je tím respektován možný přenos při odečítání 2 od nižší části adresy. Když je uložen tento vyšší bajt jako vyšší bajt adresy začátku Basic- RAM, následuje skok na rutinu LOAD v ROM.

Jak lze šikovně tento program využít? Principielně stejně jako LOAD, přesto je toto klíčové slovo nahrazeno povelem SYS 49920. Povelem SYS 49920 "PROGRAM",8 přihraje z diskety nový program za program starý. Nakonec musíme ještě přesunout zpět začátek Basic-RAM na původní místo povelem POKE 43,1:POKE 44,8.

12. Matematický aparát pro CPU

Kdo si již lámal hlavu s tím, jak řeší CPU složitější matematické operace, smekne před instrukcemi ADC a SBC. Jak je možné násobit, když instrukce pro násobení CPU 6502 nezná?

Sečítání místo násobení

Již v základní škole se každý naučí, že násobení je jen řada sečítání: 3 * 5 neni nic jiného, než 5+5+5 - že by v tom spočívalo řešení? Ve skutečnosti je tato úvaha správná. Klíčovým je pro nás ale násobící faktor 2. Nahradíme-li jej sečítáním, pak případně pro 2*26 v binárním kódu obdržíme:

```
% 0 0 0 1 1 0 1 0   (26)
% 0 0 0 1 1 0 1 0   (26)
% 0 0 1 1 0 1 0 0   (52)
```

Zde se nabízí jednoduché pravidlo, a to, že násobení 2 je jen posun binárního čísla o jedno místo vlevo. A pro tento účel je instrukční soubor vybaven instrukcemi rotace a posunu. O něco komplikovanější je násobení 8. Je to trojnásobné posunutí

binárního čísla vlevo. Také násobení libovolných reálných čísel (která nejsou mocninou dvou) je možné, ale proces výpočtu je složitější. Takové rutiny jsou uloženy v ROM.

Dalších výkladů na téma matematika se raději ušetříme, ale přesto zcela nepovšimnuty instrukce rotace a posunu nenecháme. Procesor posune při těchto operacích všechny bity v bajtu o jedno binární místo. Směr posunu, operand, jakož i způsob operace je zakotven v operačním kódu instrukce.

ASL (Arithmetic Shift Left - aritmetický posun vlevo) posune vždy bity operandu doleva, plní uvolněný bit operandu nulou a přesune nejvyšší bit do Carry-Flag. **LSR** (Logical Shift Right - logický posun vpravo) provádí posun opačným směrem doprava.

Podle téměř stejného vzoru pracují instrukce rotací **ROL** (Rotate Left - rotace vlevo) i **ROR** (Rotate Right - rotace vpravo), ale uvolněný bit není zaplněn nulou, ale obsahem Carry-Flag.

Snadné rozhodování

Nyní již známe nejdůležitější instrukce, které provádějí matematické operace. Stále nám ale chybí rozhodovací instrukce. Pro tento účel slouží Negativ-Flag t.j. 7.bit statusregistru. Tento příznak je kopií nejvyššího bitu výsledku operace. Jak známo, dává tento bit (příznak) v dvojkovém doplňku informaci o tom, zda je bajt záporný (nejvyšší bit je "1") nebo kladný (nejvyšší bit je "0").

Dva speciální podmíněné skoky mají v podmínce tento příznak (Negativ- Flag). Při větvení **BPL** (Branch on Plus - větvení při kladném znaménku) dochází ke skoku jen při kladném znaménku (t.zn. (N)= "0"). Naproti tomu u **BMI** (Branch on Minus-větvení při záporném znaménku) proběhne větvení jen při záporném znaménku (t.zn. (N)="1"). Spolu s Carry-Flag je k dispozici široké spektrum podmíněných skoků. Nakonec jeden příklad: je-li obsah střadače větší nebo rovno 8, má následovat větvení (skok). Basic program by to vykonal podmínkou IF A)=8 THEN

GOTO. Ve strojovém kódu to bude povel CMP #$08 následován instrukcí BPL. Všechny možné relace jsou v tabulce:

podmínka	řešení v Basicu	řešení v assembleru
=	IF A=NN THEN LABEL	CMP #$NN BEQ LABEL
<>	IF A<>NN THEN LABEL	CMP #$NN BNE LABEL
>=	IF A>=NN THEN LABEL	CMP #$NN BPL LABEL
>	IF A>NN THEN LABEL	CMP #$NN BEQ $02 BPL LABEL
<=	IF A<=NN THEN LABEL	CMP #$NN BEQ LABEL BPL LABEL
<	IF A<NN THEN LABEL	CMP #$NN BMI LABEL

Hexadecimalní čítač událostí

S právě získanými znalostmi lze nyní snadno vyřešit přesná matematická rozhodování. Jeden sice delší, ale přesto jednoduchý program je např. čítač událostí. Náš čítač je založen na myšlence zvyšovat obsah paměťové buňky (která je zvolena jako čítač) vždy, když bude stisknuto tlačítko a stav čítače se zobrazí na obrazovce. Po vyvolání programu povelem z monitoru G C301 popř. SYS49921 vypíše počítač hexadecimálně stav čítače a zvýší jej dalším stiskem tlačítka.

```
                COUNT EQU $C301
                INCHAR EQU $FFE4
                OUTCHR EQU $FFD2
C300  00          COUNTER
C301  A9 93       COUNT LDA #$93
```

```
C303  20 D2 FF     JSR OUTCHR
C306  AD 00 C3 LDA COUNTER
C309  20 19 C3 JSR HEXOUT
C30C  20 E4 FF     WAIT JSR INCHAR
C30F  C9 00        CMP #$00
C311  F0 F9        BEQ WAIT
C313  EE 00 0C     INC COUNTER
C316  4C 01 0C     JMP COUNT
C319  48           HEXOUT PHA
C31A  4A           LSR A
C31B  4A           LSR A
C31C  4A           LSR A
C31D  4A           LSR A
C31E  20 28 C3     JSR CODOUT
C321  68           PLA
C322  29 0F        AND #$0F
C324  20 28 C3     JSR CODOUT
C327  60           RTS
C328  18           CODOUT CLC
C329  69 30        ADC #$30
C32B  C9 3A        CMP #$3A
C32D  30 02        BMI NUMBER
C32F  69 06        ADC #$06
C331  20 D2 FF     NUMBER JSR OUTCHR
C334  60           RTS
```

Hlavní program vysvětlíme jen krátce: nejprve je oběma ROM rutinám (pro vstup a výstup znaků) přiděleno návěští INCHAR a OUTCHR. Jako čítač slouží paměťová buňka COUNTER - adresa $C300. Oba první programové řádky smažou obrazovku a nastaví kurzor do levého horního rohu. Potom získá střadač kopii stavu čítače ($C306) a rutina HEXOUT tuto hodnotu zobrazí ($C309). Následující část programu, až do adresy $C312 čeká na stisk tlačítka a zvýší stav čítače ($C313). Program končí absolutním skokem na začátek programu. Podprogram HEXOUT má za úkol zobrazit hexadecimálně obsah střadače. K tomuto účelu se musí nejprve vytisknout obrazovkový kód levé a pravé cifry hex.čísla. Po uložení obsahu střadače do zásobníku se 4x přesune instrukcí LSR obsah střadače doprava. Potom se ve střadači nachází cifra, jejíž ASCII kód vytvoří rutina CODOUT. CODOUT nejprve sesadí Carry-Flag a přičte $30 k obsahu střadače a tak získá ASCII kód. Jedná-li se o písmeno (A až F), musí se dodatečně přičíst hodnota $06 (viz

dodatek F manuálu C64). Nakonec vytiskne ROM-rutina OUTCHR příslušný znak na obrazovce. Povel RTS způsobí návrat k HEXOUT rutině. K výstupu pravé cifry instrukce PLA získá kopii obsahu střadače ze zásobníku a AND #$0F odstraní tentokrát levou cifru. Příslušný znak vypočte a "vytiskne" zase rutina CODOUT a RTS zajistí skok nazpět do hlavního programu čitače.

13. Zamotané adresování

Od jisté kapitoly se nám prohání v programech absolutní indexované adresování. Charakteristické na něm je to, že operand není omezen na jednu adresu, nýbrž přičtením obsahu indexregistru získal určitou "svobodu pohybu". Vzorovým příkladem je instrukce LDA $AAAA,X, ale indexovat lze i aritmetické instrukce.

To ještě není dost! Doposud jsme mohli poznat absolutní adresování na strance 0. Ale instrukční bajt můžeme ušetřit i při indexovaném adresování na stránce 0. Je zřejmé, že instrukční soubor určuje i použitý indexregistr. Většinou se používá registr X, indexregistr Y se používá jako index jen ve zvláštních případech.

Adresování ukazatelem

Je to nepřímé adresování, při kterém není operand uložen na adrese, která je součástí instrukce - ale součástí instrukce je adresa, na níž je uložena adresa už skutečného operandu. Tento princip lze objasnit na instrukci JMP. Jak víme, způsobí instrukce JMP skok na adresu $HHLL. A protože je cíl skoku jednoznačný a adresa je přímo v instrukci uvedena, nazývá se tento způsob adresace prostý nebo absolutní. Při nepřímém adresování naproti tomu není v instrukci uvedena adresa HHLL, ale adresa paměťové buňky, na které se nachází skutečná adresa.

Nepřímý skok JMP ($HHLL) s kódem 6C LLHH použije obsah paměťové buňky jako nižší bajt cílové adresy skoku a obsah uvedené buňky HHLL+1 jako vyšší bajt cílové adresy skoku. Proto

adresa, která je uvedena v instrukci $HHLL je označována jako ukazatel, protože ukazuje na cílovou adresu. Závorkovaná hodnota tedy v assembleru označuje nepřímé adresování.

Nepřímé adresování

S otázkou nepřímého adresování jsme se dostali bezpochyby do obtížnější oblasti strojového jazyku. Navíc lze toto adresování adresování ještě indexovat, naštěstí jen v nulové stránce. Protože máme dva indexregistry, máme také dva způsoby tohoto adresování. První skupina instrukcí využívá indexregistru X a nazývá se nepřímé předindexované adresování. Operand má přitom formát ($00LL,X).

Skutečnou adresu získáme z obsahu adresy $00LL v nulté stránce a obsahu indexregistru X. Příklad: instrukce LDA ($00LL,X) plní střadač obsahem paměťové buňky, kterou označuje ukazatel $00LL+X. Nebo pro ty, kteří se ještě neseznámili důvěrně s pojmem ukazatel. Tato instrukce plní střadač obsahem paměťové buňky, jejíž nižší adresový bajt se nachází na adrese $00LL+X a jejíž vyšší adresový bajt se nachází na adrese $00LL+X+1.

Druhou skupinou jsou instrukce tzv. nepřímého poindexového adresování s indexregistrem Y. Operand má formát ($00LL),Y. Ukazatel ukazuje přitom na adresu, jejíž obsah se musí přičíst k registru Y, abychom získali konečnou adresu. Příklad: LDA($0011),Y plní obsah střadače obsahem paměťové buňky, jejíž adresa se získá sečtením obsahu indexregistru s obsahem adresy, která se nachází na adrese $00LL (nižší bajt).

Některé počítače -jako ZX Spectrum - znají povel FLASH, který smaže celý obraz. V podstatě zamění strojová rutina (z ROM) barvu písma s barvou pozadí v celé obrazovkové RAM. Podobný program, který mazal ovšem jen jednotlivé řádky jsme si již ukázali. Díky nepřímému indexování lze toto elegantně provést na celé obrazovce. Princip přitom zůstává stejný - 7.bit každé buňky

obrazovkové RAM se musí invertovat. Přibližně 1kB RAM obrazovky se rozdělí na 4 bloky po 256 bajtech, které se zpracovávají cyklicky:

```
                    BLOCK EQU $00FB
                    SCREEN EQU $0288
C300  A9 00           LDA #$00
C302  85 FB           STA BLOCK
C304  AD 88 02        LDA SCREEN
C307  85 FC           STA BLOCK+1
C309  A2 00           LDX #$00
C30B  A0 00           LDY #$00
C30D  B1 FB           INVERS LDA (BLOCK),Y
C30F  49 80           EOR #%10000000
C311  91 FB           STA (BLOCK),Y
C313  C8              INY
C314  D0 F7           BNE INVERS
C316  E6 FC           INC BLOCK+1
C318  E8              INX
C319  E0 04           CPX #$04
C31B  D0 F0           BNE INVERS
C31D  60              RTS
```

První volná adresa na stránce 0 má návěští BLOCK. Ukazatel sestává z buněk BLOCK a BLOCK+1, která znamená základní adresy bloku ($0400,$0500,$0600,$0700), na začátku má hodnotu $0400. Pod návěštím SCREEN je vyšší bajt adresy začátku obrazovkové RAM. První čtyři programové řádky ukládají adresu prvního bajtu obrazovkové RAM, která má návěští BLOCK. Oba následující řádky nulují, a to: čitače X- registru (čítá ve velké smyčce) a Y-registru (index bajt a zároveň čitač v malé smyčce). Adresou $C30D začíná smyčka INVERS, ve které je invertován 7.bit. K tomu je zapotřebí do střadače přenést bajt z adresy BLOCK+Y, na začátku $0400+Y=$0400. Pak proběhne EOR se sedmým bitem nastaveným a uložení obsahu střadače na původní místo. Po zvýšení čítače v malé smyčce Y proběhne proces znovu od návěští INVERS a to celkem 256krát. Teprve pak se zvýší zákl. adresa bloku a čitač ve velké smyčce. Po 4.průchodu velkou smyčkou následuje RTS. Program můžeme otestovat např. takto:

```
10 SYS49920:FOR N =1 TO 30
20 NEXT N: GOTO 10
```

14. Nejprve myslet, potom jednat!

Programy sestavené ve spěchu přivedou občas nejen počítač k beznaději, ale přivádějí k zoufalství toho, který by se snažil být z příslušného listingu chytrý. Když by chtěl později programátor provést některé změny, dostane se potom nejsnáze k problému dokonce odzadu. Neboť Spaghetti-programy s kupou šíleností jakými jsou: použití jednoduchých proměnných na místo indexovaných, neukončené smyčky, skoky do nejzazšího rohu RAM nevedou vůbec k přehlednosti a vůbec ne k funkčnosti programu.

Basic je ještě milostiv

V Basicu ještě C64 strpí šílenství programátora, neboť chybové hlášení jej upozorní na jeho nesprávný postup. V assembleru ale reaguje počítač velmi citlivě i na to nejjednodušší uklouznutí a to vede k odstavení systému (počítač mlčí).

Proto vyžaduje assembler již daleko více přemýšlení před sestavením vlastního programu. Postup při vývoji programu by měl běžet v tak pevných drahách, aby předem vyloučil zásadní chyby. K osvědčeným typům na spolehlivý návrh programu patří strukturované programování.

Analýza řešení problému

V rámci této kapitoly bychom se nechtěli zavrtat do datových struktur, kterými jsou tvořeny organizace dat bufferu neboli zásobníku, nýbrž se budeme zabývat pravidly pro návrh programu. Na začátku každého návrhu stojí vždy myšlenka. Předpokládejme, že chceme navrhnout program v assembleru, který zkopíruje oblast paměti od adresy ZAC po adresu KON, do oblasti začínající adresou CIL (nejnižší adresa cílové oblasti). Nejdříve musíme rozebrat možné kombinace, které mohou nastat. V našem případě chceme

přenést oblast paměti do jiné oblasti pomocí instrukcí typu LOAD a STORE. Přitom mohou nastat tyto případy:

1. chceme přenést obsah oblasti paměti od adresy $0400 do oblasti od adresy $0407 v délce 5 bajtů. Postup je jednoduchý: zkopírujeme postupně všechny bajty.
2. chceme přenést blok o délce pěti bajtů z oblasti od $0400 do oblasti od $0403. Zde je to horší, protože se obě oblasti překrývají. Začneme-li s přenosem od adresy $0400 a zkopírujeme je do 0403, pak přepíšeme 4.bajt zdrojové oblasti paměti. Musíme tedy tedy začít s kopírováním od horní adresy - $0404 do $0407 atd. Obecně se dá vyvodit, že všechny procesy kopírování ve směru k vyšší adrese se musí provádět z vrcholu zdrojového bloku do vrcholu cílového bloku. Opačně to platí ale také a všechny procesy kopírování do oblasti s nižší adresou musí začínat kopírováním nejnižší adresy zdroje do nejnižší adresy cíle. Jen tak je zaručeno, že překrývající se bloky ani částečně neporuší obsah zdrojového bloku.

Ale teď zpět ke strukturovanému programování. Po rozboru možností se může začít s diagramem programu. Nejlépe se při tom osvědčily tzv. struktogramy, které se skládají z jednotlivých strukturních elementů. Tyto elementy lze dělit na:

PŘÍKAZ - je to povel nebo skupinu povelů bez větvení. Příkladem je akce: přenes kurzor na 7. pozici třetího řádku.

ROZHODOVÁNÍ - větvení chodu programu do dvou směrů, podle toho, zda byla podmínka splněna nebo ne.

VOLBA - širší větvení, které se uskutečňuje v závislosti na hodnotě proměnné. Obdobně jako příkaz Basicu ON X GOTO A,B,C.

OPAKOVÁNÍ - provádění příkazu nebo skupiny příkazů vícekrát. Podmínka konce cyklu může být umístěna na začátku nebo konci elementu.

Od myšlenky ke struktogramu

Nyní známe všechny elementy a můžeme naskicovat první hrubý postup přenosu bloku paměti. Skica je vlastně jen nejhrubší obrys programu:

Rozdíl - počet přenášených bajtů - R

Směr kopírování nahoru?

ne	ano
čitač X=0	čitač X=R
čitač zvýšit	čitač snížit
kopie ZAC+X	kopie ZAC+X
do CIL+X	do CIL+X
opakovat až čitac=R	opakovat až čitač=0
skok zpět	skok zpět

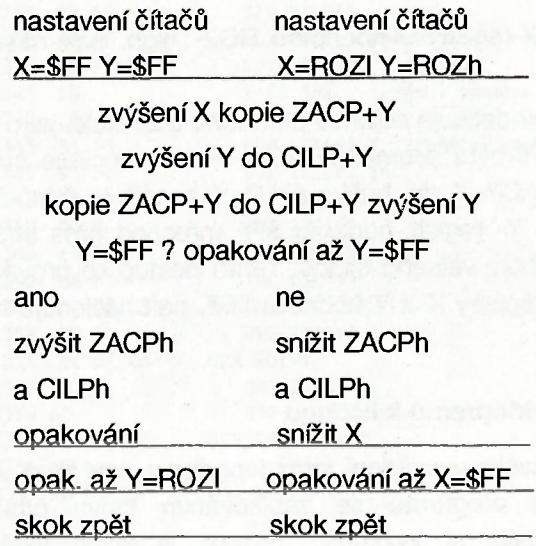
Program prvně spočítá počet kopírovaných bajtů. Následující rozhodování se postará o větvení v závislosti na směru přenosu. Když je umístěna cílová adresa směrem k nižší adrese (od adresy zdroje), čitač je vynulován a smyčka kopíruje (se zvyšováním adresy) tak dlouho, až je čitač roven počtu kopírovaných bajtů. Ve směru k vyšší adrese obdrží čitač hodnotu danou počtem kopírovaných bajtů a kopírování se děje spolu se snižováním hodnoty čitače tak dlouho, až se čitač vynuluje. V obou případech končí program zpětným odskokem (neboli návratem).

Dalším krokem je další zjemňování struktogramu:

Počet přenášených bajtů ROZ = KON-ZAC

CIL-ZAC > 0 ?

NE	ANO
definice pom.ukaz.	definice pom.ukaz.
ZACP=ZAC	ZACP=ZAC+ROZh
CILP=CIL	CILP=CIL+ROZh



Nejprve je spočítán rozdíl (počet kopírovaných bajtů) mezi počáteční adresou ZAC a kon.adresou KON. Směr kopírování se určí srovnáním poč.adres zdrojové a cílové oblasti. Je-li ZAC (zdrojová) adresa větší, pak následuje přenos ve směru k nižší adrese - je zvolena levá větev. V opačném případě je zvolena pravá větev. Obě větve vypadaji velmi podobně, neboť se liší jen sledem kopií. Levá větev definuje nejdříve pomocné ukazatele ZACP a CILP z hodnot poč.adres obou oblastí paměti. Jako čitače počtu kopírovaných bajtů slouží oba indexregistry X (čitač velké smyčky) a Y (čitač malé smyčky), jejichž počáteční hodnota bude -1 = $FF.

Následující smyčka zvýší jak čitač velké, tak i malé smyčky tak, že skutečnými počátečními hodnotami jsou $0000. Potom je buňka s adresou ZACP+Y zkopírována do CILP+Y. Když získá Y-registr hodnotu $FF, znamená to stav těsně před překročením stránky paměti a dále musí být zvýšena hodnota obou čitačů. Vnitřní smyčka opakuje kopírování tak dlouho, až je sesouhlasena hodnota malého čitače (Y-registr) s hodnotou ROZ-low a hodnota

velkého čitače (X-registr) s hodnotou ROZ- high, pak následuje skok do Basicu.

Druhá větev definuje nejdříve pomocné ukazatele, jako sumy nízké adresy a HI-byte proměnné ROZ. Potom kopíruje smyčka obsah adresy ZACP+Y do buňky CILP+Y a snižuje čitač Y tak dlouho, až získá Y- registr hodnotu $FF (přechod přes stránku). Pak se sníží hodnota velkého čitače. Tento postup se provádí tak dlouho, až mají registry X a Y hodnotu $FF, pak následuje návrat do Basicu.

Od struktogramu k listingu

Komu nestačilo vysvětlení tohoto postupu, ten ať si vezme papír a průběh programu se zapisováním stavu čítačů a přemísťováním bajtů si vyzkouší. Z toho je jasně vidět, že kopírování bajtů skutečně funguje. Konečnou verze tohoto programu je tato:

```
                ZAC EQU $C300
                KON EQU $C302
                CIL EQU $C304
                ROZ EQU $C306
                ZACP EQU $FB
                CILP EQU $FD
C308  38          SEC , výpoèet hodnoty ROZ
C309  AD 02 C3    LDA KON
.C30C ED 00 C3    SBC ZAC
C30F  8D 06 C3 STA ROZ
C312  AD 03 C3 LDA KON+1
C315  ED 01 C3       SBC ZAC+1
C318  8D 07 C3       STA ROZ+1
C31B  AD 04 C3       LDA CIL , stanovení smìru
C31E  CD 00 C3    CMP ZAC
C321  AD 05 C3       LDA CIL+1
C324  ED 01 C3       SBC ZAC+1
C327  10 31          BPL DIRTOP , smìr nahoru
C329  AD 00 C3       LDA ZAC , smìr dolù
C32C  85 FB          STA ZACP
C32E  AD 01 C3       LDA ZAC+1
C331  85 FC          STA ZACP+1
C333  AD 04 C3    LDA CIL
C336  85 FD          STA CILP
C338  AD 05 C3    LDA CIL+1
```

```
C33B  85 FE          STA CILP+1
C33D  A2 FF          LDX #$FF , poè.hodnota
C33F  A0 FF          LDY #$FF
C341  E8             ZACS INX , zvý▒ení obsahu èitaèe
C342  C8             INY
C343  B1 FB          LDA (ZACP),Y , zaèátek pøenosu
C345  91 FD          STA (CILP),Y
C347  C0 FF          CPY #$FF
C349  D0 04          BNE INBLOK
C34B  E6 FC          INC ZACP+1
C34D  E6 FE          INC CILP+1
C34F  CC 06 C3    INBLOK CPY ROZ
C352  D0 EE          BNE ZACS+1
C354  EC 07 C3    CPX ROZ+1
C357 D0 E8           BNE ZACS
C359 60              RTS
C35A AD 00 C3     DIRTOP LDA ZAC
C35D 85 FB           STA ZACP
C35F AD 03 C3     LDA KON+1
C362 85 FC           STA ZACP+1
C364 AD 04 C3     LDA CIL
C367 85 FD           STA CILP
C369 18              CLC
C36A AD 05 C3     LDA CIL+1
C36D 6D 07 C3     ADC ROZ+1
C370 85 FE           STA CILP+1
C372 AE 07 C3        LDX ROZ+1
C375 AC 06 C3     LDY ROZ
C378 B1 FB           TOPS LDA (ZACP),Y
C37A 91 FD           STA (CILP),Y
C37C 88              DEY
C37D C0 FF           CPY #$FF
C37F D0 F7           BNE TOPS
C381 C6 FC           DEC ZACP+1
C383 C6 FE           DEC CILP+1
C385 CA              DEX
C386 E0 FF           CPX #$FF
C388 D0 EE           BNE TOPS
C38A 60              RTS
```

Další vysvětlení vypouštíme, protože listing přesně odpovídá struktogramu. K ověření funkce programu přesuneme oblast obrazovkové RAM. Pokud program svým výpisem zaplní obrazovku, roluje počítač obsah obrazovky o jeden řádek nahoru. Za pomoci několika POKE a naši transferrutiny může obsah obrazovky rolovat

také dolů. Následující program nastaví počáteční parametry (viz úvod naší rutiny) a vyvolá transferrutinu.

```
10 SH = PEEK(648)
20 POKE 49920,0:POKE 49921,SH
30 POKE 49922,152:POKE 49923,SH + 3
40 POKE 49924,40:POKE 49925,SH
50 SYS 49928
60 GOTO 50
RUN
```

Nejlépe je trochu si s programem pohrát a poznat tak jeho silné a slabé stránky. Když přitom narazíte na to, že se vám textové řádky nějak divně množí, spočívá to v tom, že při každém vyvolání rutiny se kopíruje obsah nejhořejšího řádku do řádku následujícího.

15. Další příznakové bity

Dekadický příznak - D

Je to 3.bit statusregistru a slouží vlastně k přepínání druhu provozu. V normálním módu je D-flag vypnut a počítač provádí veškeré operace hexadecimálně. Je-li D-flag nastaven ("1"), pak procesor pracuje v dekadickém módu, t.zn. že pracuje s dekadickými čísly. Změna příznaku D se provádí instrukcemi SED (Set Decimal) - nastaví D na "1", a CLD (Clear Decimal) - sesadí D na "0".

V příkladě provedeme změnu hexadecimálního čitače z kapitoly 12 na čitač dekadický. Příkaz INC nahradíme příkazem ADC #$01, protože INC a DEC nepracují dekadicky (to platí i pro INX, INY, DEX, DEY). Výstupní rutina HEXOUT programu z kapitoly 12 zůstává nezměněna:

```
OUTCHR EQU $FFD2
INCHAR EQU $FFE4
HEXOUT EQU $C31D
```

```
                    CITAC EQU $00FA
C300 A9 93          COUNT LDA #$93
C302 20 D2 FF         JSR OUTCHR
C305 A5 FA            LDA CITAC
C307 20 1D C3         JSR HEXOUT
C30A 20 E4 FF         WAIT JSR INCHAR
C30D C9 00            CMP #$00
C30F F0 F9            BEQ WAIT
C311 F8               SED
C312 18               CLC
C313 A5 FA            LDA CITAC
C315 69 01            ADC #$01
C317 85 FA            STA CITAC
C319 D8               CLD
C31A 4C 00 C3       JMP COUNT
C31D 4B               HEXOUT PHA
C31E 4A               LSR A
C31F 4A               LSR A
C320 4A               LSR A
C321 4A               LSR A
C322 20 2C C3         JSR CODOUT
C325 68               PLA
C326 29 0F            AND #$0F
C328 20 2C C3         JSR CODOUT
C32B 60               RTS
C32C 18 C0          DOUT CLC
C32D 69 30            ADC #$30
C32F C9 3A            CMP #$3A
C331 30 02            BMI NUMBER
C333 69 06            ADC #$06
C335 20 D2 FF         NUMBER JSR OUTCHR
C338 60               RTS
```

Příznak přetečení - V

Přetečení je stav, při kterém vŕsledek operace překračuje kapacitu určené paměti. K signalizaci takového stavu slouží 6.bit statusregistru - V-flag.

Nyní se vysvětlíme, jak přetečení vzniká. Zobrazení čísla ve dvojkovém doplňku používá 7.bit jako bit znaménkový. Pokud by se tento bit měl změnit přenosem do vyššího řádu z vedle ležícího bitu a ne změnou znaménka, dochází k přetečení a V-flag se nastaví na 1. Příklad: $4E + $7A = $C8 (%11001000). Pokud ale pracujeme s čísly se znaménky, pak je výsledek chybný, poněvadž

indikuje číslo -56. Přenos do znaménkového bitu způsobil chybu přetečení.

Abychom mohli provést ošetření takové chyby, máme k dispozici instrukce BVS (Branch on Overflow Set) - podmíněný skok při V=1, a BVC (Branch on Overflow Clear) - podmíněný skok při V=0. Pro mazání příznaku V máme instrukci CLV (Clear Overflow).

Operace s jednotlivými bity

Většina procesorů umožňuje přímé operace s bity. CPU 6502 to dovoluje pouze nepřímo, pomocí instrukcí pro nastavení, mazání nebo invertování bitů. Chceme-li zjistit stav některého bitu, je to ještě složitější. Např. test 6.bitu na adrese $HHLL:

```
LDA #%01000000
AND $HHLL
BEQ NOTSET
BNE SET
```

Do A vložíme hodnotu, ve které je 6.bit roven 1. Pak provedeme logický součet AND s obsahem adresy $HHLL. Pokud je výsledek logického součtu roven nule (testovaný bit = 0), provede se skok na NOTSET, v opačném případě se provede skok na SET.

Tato metoda má jeden zápor - operace AND mění obsah střadače. Tento nedostatek nám odstraňuje instrukce BIT. Předešlý test by pak vypadal takto:

```
LDA #%01000000
BIT $HHLL
BEQ NOTSET
BNE SET
```

Další výhodou instrukce BIT je to, že kopíruje 6.bit do V-flagu a 7.bit do N-flagu. Po provedení instrukce BIT pak lze využít následující podmíněné skoky:

BVC je-li 6.bit = 0

BVS je-li 6.bit = 1

BPL je-li 7.bit = 0

BMI je-li 7.bit = 1

Výstup binárních hodnot

Naprogramovat převod binárních čísel je v Basicu relativně obtížné. Program v assembleru je proti tomu jednoduchý:

```
                BIN EQU $FA
                COMP EQU $FB
                COMMA EQU $AEFD
                BYTE EQU $B79E
                OUTCHR EQU $FFD2
C300 20 FD AE     JSR COMMA
C303 20 9E B7     JSR BYTE
C306 86 FA        STX BIN
C308 A9 01        LDA #%00000001
C30A 85 FB        STA COMP
C30C 66 FB        TEST ROR COMP
C30E A5 FA        LDA BIN
C310 24 FB        BIT COMP
C312 F0 04        BEQ NOTSET
C314 A9 31        LDA #$31
C316 D0 02        BNE BINOUT
C318 A9 30        NOTSET LDA #$30
C31A 20 D2 FF     BINOUT JSR OUTCHR
C31D A9 01        LDA #%00000001
C31F C5 FB        CMP COMP
C321 D0 E9        BNE TEST
C323 A9 0D        LDA #$0D
C325 20 D2 FF     JSR OUTCHR
C328 60           RTS
```

Program se volá zvláštním způsobem - SYS49920,N - kde N je číslo, které chceme převést do binárního tvaru. Na začátku programu se volá systémová rutina na adrese $AEFD, která testuje, zda na dalším místě programu (po volání SYS) je čárka. Pokud není, vyvolá se chybové hlášení. Za čárkou má být uvedeno číslo, které chceme převést. Toto číslo se do strojového programu přenáší další systémovou rutinou na adrese $B79E. Tato rutinu uloží toto číslo do registru X. V programu používáme dvě adresy z nulté stránky - $FA s označením BIN jako mezipaměť pro výstupní bajt (uchovává X-registr - C306), a $FB (COMP) pro uchování hodnoty, v níž je nastaven právě aktuální bit (adresy C308 až C30A).

Vlastní výstupní smyčka začíná na adrese C30C s rotací vpravo obsahu COMP. Pak se do A uloží hodnota výstupního bajtu (C30E) a proběhne test na nastavení příslušného bitu v COMP. Je-li tento bit nastaven i v BIN, objeví se na obrazovce dvojková číslice 1 (C314), jinak 0 (C318). Výstup pokračuje tak dlouho, než bude v COMP opět počáteční hodnota %00000001 (C31D až C321). Právě tehdy bude hotov 8.bit a dojde k ukončení programu instrukcí RTS. Předtím se ještě nastaví na obrazovce nový řádek.

16. Programová přerušení

Zastavení při běhu programu nelze vždy svádět na chybu počítače. Bývají to spíše důsledky přerušení programu (Interrupt). Pokud nám dojde vlivem chyby v programu ke zhroucení systému, nemusíme vždy vypínat počítač, stačí obvykle provést RESET. Reset posílá impuls na reset vstup CPU, ten vyvolá přerušení programu a nepřímý skok JMP ($FFFC). Tím se uvede v činnost inicializační rutinu.

Procesor 6502 zná dva druhy přerušení: nemaskované přerušení NMI a maskované přerušení IRQ. Obě tato přerušení se vyvolávají impulzem na určitém vstupu procesoru pro dané přerušení.

NMI přerušení

Při tomto přerušení (pin 4 CPU) přeruší procesor právě prováděný program, uloží aktuální stav PC registru a statusregistru do zásobníku a provede nepřímý skok JMP ($FFFA). Tento skok vede na rutinu pro obsluhu přerušení na adrese $FE43. Tato rutina končí příkazem RTI (Return from Interrupt), který zavede procesor zpět na místo v hlavním programu, kde přišlo k přerušení. K tomu si vyzvedne procesor ze zásobníku stav PC registru a statusregistru.

Statusregistr se ukládá proto, aby byl i po návratu zachován jeho původní stav a nenarušil se běh hlavního programu. NMI přerušení vyvolává stisk klávesy RESTORE, je-li současně stisknuta i klávesa RUN/STOP (testuje rutina přerušení), dojde k inicializaci op.systému.

Programátor může využít NMI přerušení ve svých programech, poněvadž i když ukazetel na adrese $FFFA ukazuje na rutinu $FE43, tak ta provádí nepřímý skok přes vektor na adrese $0318. Tento vektor je v RAM a lze jej proto změnit tak, aby ukazoval na startadresu naší vlastní rutiny (ta musí ale končit RTI).

Přerušení typu NMI nelze již dále přerušit ani ignorovat - je nemaskované.

IRQ přerušení

Je to maskované přerušení (a lze dále přerušit vyšším druhem přerušení), které se vyvolává impulzem na pinu 3 CPU. Při jeho vyvolání se uchová stav PC registru a statusregistru a provede se nepřímý skok JMP($FFFE). Až dosud se neliší od přerušení NMI. Nyní ale nastává rozdíl, protože CPU může toto přerušení ignorovat v závislosti na příznaku I - 2.bit statusregistru. Je-li tento příznak nastaven na 1, pak se všechny impulzy IRQ ignorují a pokračuje se dál v hlavním programu. Je-li I=0, pak dojde k přerušení a provede se rutina pro obsluhu přerušení.

Pro změnu nastavení I-flagu slouží instrukce SEI (Set Interrupt), která nastaví I na "1" a CLI (Clear Interrupt), která sesadí I na "0". Příznak I je nastven na "1" také automaticky při prvním impulzu IRQ, takže další IRQ jsou ignorovány. Při návratu do hlavního programu instrukce RTI nastaví I na "0".

BRK přerušení

Je to přerušení IRQ, které je vyvoláno programově (ne hardwarově jako pravé IRQ) instrukcí BRK. Dojde k němu, pokud při instrukci je I-flag roven 0.

Aby se BRK IRQ odlišilo od hardwarového IRQ, nastavuje příkaz BRK příznakový bit B ve statusregistru (bit č.4). Rutina pro obsluhu IRQ přerušení na adrese $FF48 testuje tento bit a podle jeho obsahu pozná, jestli impulz IRQ přišel zvenku nebo ne. Pak skáče nepřímo přes vektor $0316 (BRK IRQ) nebo $0314 (IRQ).

17. Přehled instrukcí CPU 6502/6510

Dosud jsme poznali samotné instrukce, flagy a způsoby adresování procesoru 6502. Přesto si ale účelné a efektivni programování žádá více, než jen znalost všech instrukci. K mistrovskému programování patří právě tak přehled o všech prostředcích, které jsou k disposici pro řešení nějakého problému. Výpis všech instrukcí předkládá tabulka. Instrukce jsou seřazeny podle určení. Když instrukce ovlivňuje flag (příznak), je v kolonce ovlivnění příznaku pro příslušný příznak znak "!":

Význam jednotlivých opkódů

Přenos

LDA - Load Accumulator - naplň střadač
LDX - Load X register - naplň registr X
LDY - LOAD Y register - naplň registr Y
STA - Store Accumulator - ulož střadač
STX - Store X register - ulož registr X
STY - Store Y register - ulož registr Y
PHA - Push Accumulator - ulož střadač do zásobníku
PLA - Pull Accumulator - naplň střadač ze zásobníku
PHP - Push Proc.status - ulož statusreg. do zásobníku
PLP - Pull Proc.status - naplň statusreg. ze zásobníku
TAX - Transfer A to X - přenes střadač do registru X
TXA - Transfer X to A - přenes registr X do střadače
TAY - Transfer A to Y - přenes střadač do registru Y

TYA - Transfer Y to A - přenes registr Y do střadače
TSX - Transfer ST to X - přenes stackpointer do X
TXS - Transfer X to ST - přenes X do stackpointru

Skoky

JMP - Jump - skoč na adresu
JSR - Jump to Subroutine - skoč na podprogram
RTS - Return from Subr. - návrat z podprogramu
RTI- Return from Inter. - návrat z přeřušení
BEQ - Branch on Equal - skoč, je-li Z=1
BNE - Branch on Not Equal - skoč, je-li Z=0
BMI - Branch on Minus - skoč, je-li N=1
BPL - Branch on Plus - skoč, je-li N=0
BCS - Branch on Carry Set - skoč, je-li C=1
BCC - Branch on Carry Clr - skoč, je-li C=0
BVS - Branch Overflow Set - skoč, je-li V=1
BVC - Branch Overflow Clr - skoč, je-li V=0

Statusregistr

SEC - Set Carry - nastav C=1
SED - Set Decimal - nastav D=1
SEI- Set Interrupt - nastav I=1
CLC - Clear Carry - nastav C=0
CLD - Clear Decimal - nastav D=0
CLI- Clear Interrupt - nastav I=0
CLV - Clear Overflow - nastav V=0

Aritmetika a logika

CMP - Compare Mem and Acc - porovnej obsah paměti a A
CPX - Compare Mem and X - porovnej obsah paměti a X
PCY - Compare Mem and Y - porovnej obsah paměti a Y
ADC - Add Mem to A (Carry)- přičti obsah paměti k A

SBC - Subtract Mem from A - odečti obsah paměti od A
DEC - Decrement Mem by 1 - odečti od obsahu paměti 1
INC - Increment Mem by 1 - přičti k obsahu paměti 1
DEX - Decrement X by 1 - odečti od obsahu X 1
INX - Increment X by 1 - přičti k obsahu X 1
DEY - Decrement Y by 1 - odečti od obsahu Y 1
INY - Increment Y by 1 - přičti k obsahu Y 1
AND - AND Mem with A - proveď AND obsahu paměti a A
EOR - EXOR Mem with A - proveď EXOR obs. paměti a A
ORA - OR Mem with A - proveď OR obsahu paměti a A
BIT- Bit Test Mem with A - test bitů paměti s bity střadače

Posun a rotace

ASL - Shift Left One Bit - posun vlevo o jeden bit
LSR - Shift Right One Bit - posun vpravo o jeden bit
ROL - Rotate One Bit Left - rotace o jeden bit vlevo
ROR - Rotate One Bit Right- rotace o jeden bit vpravo

Jiné

NOP - No Operation - nedělej nic
BRK - Break - přerušení BRK

Přenos dat

operand: #$NN $HHLL $LL $HHLL,X $HHLL,Y ($LL,X) ($LL),Y $LL,X $LL,Y

```
LDA A9 AD A5 BD B9 A1 B1 B5 -
LDX A2 AE A6 - BE - - - B6
LDY A0 AC A4 BC - - - B4 -
STA - 8D 85 9D 99 81 91 95 -
STX - 8E 86 - - - - - 96
STY - 8C 84 - - - - 94 -
PHA 48 PHP 08 TAX AA TAY A8 TSX BA
PLA 68 PLP 28 TXA 8A TYA 98 TXS 9A
```

Skokové příkazy

```
JMP 4C JMP() 6C JSR 20 RTS 60 RTI 40
BEQ F0 BCS B0 BMI 30 BVS 70
```

```
BNE D0 BCC 90 BPL 10 BVC 50
```

Příkazy pro flagy

```
SEC 38 SED F8 SEI 78 CLV B8
CLC 18 CLD D8 CLI 58
```

Příkazy posunu a rotace

Operand: A $HHLL $LL $HHLL,X $LL,X

```
ASL 0A 0E 06 1E 16
LSR 4A 4E 46 5E 56
ROL 2A 2E 26 3E 36
ROR 6A 6E 66 7E 76
```

Aritmetické a logické příkazy

Operand: #$NN $HHLL $LL $HHLL,X $HHLL,Y ($LL,X) ($LL),Y $LL,X $LL,Y

```
ADC 69 6D 65 7D 79 61 71 75 -
SBC E9 ED E5 FD F9 E1 F1 F5 -
INC - EE E6 FE - - - F6 -
DEC - CE C6 DE - - - D6 -
AND 29 2D 25 3D 39 21 31 35 -
EOR 49 4D 45 5D 59 41 51 55 -
ORA 09 0D 05 1D 19 01 11 15 -
BIT - 2C 24 - - - - - -
CMP C9 CD C5 DD D9 C1 D1 D5 -
CPX E0 EC E4 - - - - - -
CPY C0 CC C4 - - - - - -
INX E8 DEX CA INY C8 DEY 88
NOP EA BRK 00
```

Vliv instrukcí na flagy statusregistru

Instrukce	Flag:	N	V	B	D	I	Z	C
LDA LDX LDY PLA TAX TXA TSX TAY TYA		!	-	-	-	-	!	-
PLP RTI		!	!	!	!	!	!	!
ASL LSR ROL ROR CMP CPX CPY		!	-	-	-	-	!	!
BRK		-	-	!	-	!	-	-
SEC CLC		-	-	-	-	-	-	!
SED CLD		-	-	-	!	-	-	-
SEI CLI		-	-	-	-	!	-	-

```
CLV                                  - ! - - - - - -
ADC SBC                              ! ! - - - - ! !
INC DEC INX DEX INY DEY AND EOR ORA  ! - - - - - ! -
BIT                                  ! ! - - - - ! -
```

Nyní jsme si uvedli oficiální seznam instrukcí procesoru 6502/6510. Mimo ně ale ještě existují další instrukce, které mohou zkušení programátoři využít. Je třeba si ale uvědomit, že ne každý monitor nebo assembler s nimi umí pracovat. Jejich význam a kódy jsou v následujících tabulkách:

Aritmetika a logika

ALR - Acc AND Data, LSR result - logické AND střadače a čísla, s výsledkem provede LSR

ARR - Acc AND Data, ROR result - logické AND střadače a čísla, s výsledkem provede ROR

ASO - ASL then ORA with Acc - ASL a pak s výsledkem provede ORA se střadačem

AXS - Store result A AND X - uloží výsledek A AND X

DCM - DEC Mem then CMP with Acc - sníží obsah paměti o 1, pak provede CMP se střadačem

INS - INC Mem then SBC with Acc - zvýší obsah paměti o 1, pak provede SBC se střadačem

LSE - LSR then EOR result with A - provede LSR a s výsledkem EOR se střadačem

MKA - Acc AND #$04 to Acc - provede AND obsahu střadače a čísla #$04

MKX - X AND #$04 to X - provede AND obsahu reg. X a čísla #$04

OAL - ORA Acc with #$EE, then - provede ORA střadače a #$EE, AND with data, then TAX pak AND s pamětí, nakonec TAX

RLA - ROL Mem, then AND with A - provede ROL s pamětí, pak AND se střadačem

RRA - ROR Mem, then ADC to Acc - provede ROR s pamětí, pak ADC se střadačem

SAX - A AND X, then SBC Mem, - provede A AND X, pak odečte store to X data a výsledek uloží do X

XAA - X AND Mem to Acc - provede X AND obsah paměti, výsledek uloží do A

Přesun, skoky, jiné

LAX - Load Mem to A and X - přesune obsah paměti do A a X

SKB - Skip Byte - přeskočí jeden bajt

SKW - Skip Word - přeskočí dva bajty

CIM - Crash Intermediate - rozpad systému

Instrukce kódy

```
NOP 1A 3A 5A 7A DA FA
SKB 80 82 C2 E2 04 14 34 44 54 64 74 D4 F4
SKW 0C 1C 3C 5C 7C DC FC
CIM 02 12 22 32 42 52 62 72 92 B2 D2 F2
```

Operand: #$NN $HHLL $HHLL,X $HHLL,Y $LL $LL,X ($LL,X) ($LL),Y

```
ALR 4B - - - - - - -
ARR 6B - - - - - - -
ASO 0B 0F 1F 1B 07 17 03 13
AXS - 8F - - 87 97(Y) 83 93
DCM - CF DF DB C7 D7 C3 D3
INS - EF FF FB E7 F7 E3 F3
LAX - AF - BF A7 B7 A3 B3
LSE - 4F 5F 5B 47 57 43 53
MKA - 9F - - - - - -
MKX - 9E - - - - - -
OAL AB - - - - - - -
RLA 2B 2F 3F 3B 27 37 23 33
RRA - 6F 7F 7B 67 77 63 73
SAX CB - - - - - - -
XAA 8B - - 9B - - - -
```

Když jsme předložili kompletní softwarovou specifikaci procesoru, musíme osvětlit i hardwarovou stránku. K tomu slouží obrázek, na němž vidíme jednotlivé vývody (piny) procesoru.

Aby byl pořádek v datovém toku uvnitř mikropočítačového systému, potřebuje systém tzv. datovou sběrnici, která sestává u 8-bitového mikroprocesoru z 8 vodičů. Pro každý přenásený bit je tedy k dispozici jeden vodič. Jednoduchý příklad: U instrukce STA

$HHLL vloží CPU (mikroprocesor) obsah střadače v binární formě na datovou sběrnici (dále jen databus), odkud jsou data uloženy do paměťové buňky $HHLL. Pro cílové adresování t.z. je známa v jednom okamžiku konkrétní celá adresa paměťové buňky, je ale nutných 16 adresových vodičů. Tyto vodiče tvoří adresovou sběrnici (adresbus). Ve výše uvedeném příkladu procesor při vykonavání instrukce STA musí dodatečně vložit na adresový bus adresu $HHLL v binární podobě. Jen tak zjistí konkrétní paměťový obvod, o kterou paměťovou buňku se vlastně jedná a na kterou budou uloženy data z databusu. Samozřejmě, že platí právě popsaný postup také pro data, která jsou předavána ve směru k procesoru. Z paměťové buňky jsou data vyslána na databus a procesor hraje roli příjemce. Až dosud se lišily 6510 a 6502 jen v rozmístění pinu určitého významu. 6510 má ale šest přídavných vodičů vstupní a výstupní brány. P0 až P5 jsou určeny k řízení a testování externích zařízení. CPU by mohl např. (samozřejmě přes speciální spínače) zapínat a vypínat 6 žárovek popř. kontrolovat činnost 6-ti spínačů. Kdyby se toto mělo provádět s 6502, byl by zapotřebí navíc další obvod (např. 6520-PIO). Takový obvod obsahuje registr směru a přenosu dat, který určuje, je-li brána připojena jako vstupní nebo jako výstupní. Log. "0" v tomto registru připojí bránu (port) jako vstupní, zatímco "1" znamená výstupní bránu.

Jeden bit dá systému mat

U 6510 je adresa $0000 registrem směru přenosu dat. Je-li např. 3.bit ve stavu log. "1", je pin P3 výstupním pinem brány. Protože máme jen 6 bitů brány, není 6. a 7.bit využitý. Adresa $0001 obsahuje potom tzv. datový registr se stavy příslušných příslušných vodičů brány. Dotazem na bity buňky $0001 lze zjistit logickou uroveň na vstupním portu. A nastavením popř. sesazením

bitu lze přepínat vystupní úrovně na výstupním portu. C64 využívá ovšem celý port, takže nezůstávají žádné volné bity brány živateli. Porty 3 až 6 jsou k dispozici datasetu. Pomocí portu 0 lze ochránit interpret Basicu ($A000 až $BFFF). Nastavíte-li tento port na 0, je k disposici RAM, která jinak patří interpretu. Analogicky port 1 vyřadí obslužný systém ($E000 až $FFFF). Port 2 umožňuje volbu mezi znakovým generátorem (0) a I/O (1)

18. Cesty pro tok dat

Pomocí tzv. čísel zařízení lze C64 oznámit, co je zdrojem a co cílem toku dat. Navíc udávají kterým kanálem data potečou. U našich známých ROM rutin INCHAR a OUTCHR byla tato čísla utajena: 0 otevírá rutině INCHAR cestu ke klávesnici a 3 říká rutině OUTCHR, že cílem datového toku je obrazovka. Tok dat však může jít i jinými cestami: 1 otvírá kanál k dattassetu, 4 k tiskárně, 8 ukazuje cestu k floppy a 2 signalizuje, že je osloven user-port. Tato čísla potřebujeme proto, abychom v MC programu usměrňovali tok dat ke správnému cíli.

Simulace povelu OPEN

Aby byl vyslán jeden datový bajt na tiskárnu nebo floppy, je nutné otevřít sekvenční datový soubor. Sekvenční datový soubor je jen zvláštní způsob vyjádření řetězce znaků. Lze jej měnit tak, že budeme přivěšovat další znaky k již existujícímu souboru, který může být z počátku i prázdný. Prohlížení souboru je možné opakovaným čtením znaků, přičemž se začíná vždy od začátku.

V Basicu se sekvenční datový soubor otevírá příkazem OPEN. Např.: OPEN 1,8,4,"TEXT,S,W" otvírá soubor se jménem TEXT na disku. Přitom "W" (WRITE) definuje, že soubor otvíráme jako výstupní, že do něj budeme zapisovat. Systémová rutina pro OPEN je na adrese $FFC0. Jaké parametry budou v jakých buňkách předávány lze vyčíst z tabulky systémových rutin. Otevření ještě nic neznamená, protože pouze definuje směr toku

dat. Pro vlastní přenos dat je zapotřebí další rutiny. Ta se liší pro vstup (čtení) a výstup (zápis) podle zařízení. Rutina OUTCMD na adrese $FFC9 zajistí výstup do souboru, jehož číslo je uloženo v X registru. Analogicky k tomu rutina INCMD na adrese $FFC6 definuje vstupní soubor. Teprve pak může následovat vstup a výstup dat s rutinami OUTCHR a INCHAR na nově vytvořené soubory. K uzavření vstupu a výstupu pro klávesnici a obrazovku slouží rutina na adrese $FFCC. Když má být uzavřen datový soubor, musí být vyvolána rutina na adrese $FFC3 s číslem souboru ve střadači.

Adresa návěští funkce parametry

$FFC0 OPEN	otvírá soubor	$B7 - délka jména souboru
		$B8 - číslo souboru
		$B9 - sekundární adresa
		$BA - číslo zařízení
		$BB - vektor jména soub.
$FFC3 CLOSE	zavírá soubor	A - číslo souboru
$FFC6 CHKIN	příprava vstupu	X - číslo souboru
$FFC9 CHKOUT	příprava výstupu	X - číslo souboru
$FFCC CLRCHN	končí I/O	-
$FFCF CHRIN	vstup znaku do A	rutina na adr $F157
$FFD2 CHROUT	výstup znaku z A	rutina na adr $F1CA

Přenos RAM obrazovky

Nyní navrhneme rutiny, které přenášejí obsah obrazovky na disketu a zpět. Tento úkol se dá vyřešit pomocí datových souborů:

```
                BLOCK EQU $FB
                SCREEN EQU $0288
                OPEN EQU $FFC0
                OUTCMD EQU $FFC9
                OUTCHR EQU $F1CA
                CMDOFF EQU $FFCC
                CLOSE EQU $FFC3
C300 56 49 44
C303 45 4F 2C
```

```
C306 53 2C 57
C309 A9 09 LDA #$09
C30B 85 B7 STA $B7
C30D A9 01 LDA #$01
C30F 85 B8 STA $B8
C311 A9 04 LDA #$04
C313 85 B9 STA $B9
C315 A9 08 LDA #$08
C317 85 BB STA $BA
C319 A9 00 LDA #$00
C31B 85 BB STA $BB
C31D A9 C3 LDA #$C3
C31F 85 BC STA $BC
C321 20 C0 FF JSR OPEN
C324 A2 01 LDX #$01
C326 20 C9 FF JSR OUTCMD
C329 A9 00 LDA #$00
C32B 85 FB STA BLOCK
C32D AD 88 02 LDA SCREEN
C330 85 FC STA BLOCK+1
C332 A2 00 LDX #$00
C334 A0 00 LDY #$00
C336 B1 FB STORE LDA (BLOCK),Y
C338 20 CA F1 JSR OUTCHR
C33B C8 INY
C33C D0 F8 BNE STORE
C33E E0 03 CPX #$03
C340 F0 06 BEQ END
C342 E8 INX
C343 E6 FC INC BLOCK+1
C345 4C 36 C3 JMP STORE
C348 20 CC FF END JSR CMDOFF
C34B A9 01 LDA #$01
C34D 20 C3 FF JSR CLOSE
C350 40 RTI
```

Tento program po stisku klávesy RESTORE uloží obsah obrazovku na disk.

```
C400 56 49 44
C403 45 4F 2C
C406 53 2C 52
C409 A9 09 LDA #$09
C40B 85 B7 STA $B7
C40D A9 02 LDA #$02
C40F 85 B8 STA $B8
C411 A9 04 LDA #$04
C413 85 B9 STA $B9
C415 A9 08 LDA #$08
```

```
C417 85 BA STA $BA
C419 A9 00 LDA #$00
C41B 85 BB STA $BB
C41D A9 C4 LDA #$C4
C41F 85 BC STA $BC
C421 20 C0 FF JSR OPEN
C424 A2 02 LDX #$02
C426 20 C6 FF JSR INCMD
C429 A9 00 LDA #$00
C42B 85 FB STA BLOCK
C42D AD 88 02 LDA SCREEN
C430 85 FC STA BLOCK+1
C432 A2 00 LDX #$00
C434 A0 00 LDY #$00
C436 20 3E F1 STORE JSR INCHAR
C439 91 FB STA (BLOCK),Y
C43B C8 INY
C43C D0 F8 BNE STORE
C43E E0 03 CPX #$03
C440 F0 06 BEQ END
C442 E8 INX
C443 E6 FC INC BLOCK+1
C445 4C 36 C4 JMP STORE
C448 20 CC FF END JSR CMDOFF
C44B A9 02 LDA #$02
C44D 20 C3 FF JSR CLOSE
C450 60 RTS
```

Tato část programu po vyvolání SYS 50185 zkopíruje uložený obsah obrazovky z disku zpět do paměti obrazovky. Blíže si objasníme jen přenos na disk, protože program pro vstup je velmi podobný. Nejdříve musíme simulovat povel OPEN. V listingu se rozprostírá tato část programu do adresy $C326, přičemž prvních devět bajtů je rezervováno pro název dat.souboru "VIDEO" doplněný o "S" pro sekvenční datový soubor a "W" pro zápis. Na adrese $C309 začíná vlastní program s určením předávaných parametrů pro rutinu BOPEN ($C309 až $C31F). V řadě za sebou jsou umístěny: v buňce $B7 až $BC - délka názvu datového souboru, $09 - číslo datového souboru, $04 - sekundární adresa, $08 - číslo zařízení, adresa začátku názvu datového souboru $00 (LB), a $C3 (HB) adresy začátku názvu datového souboru. Hned za to umístí rutina OPEN datový soubor ($C321). Registr X obsahuje číslo souboru přidělené na adrese v programu $C324 a ROM rutina

OUTCMD fixuje jako výstupní zařízení floppy místo obrazovky. Od tohoto okamžiku je datový soubor VIDEO schopen přijímat data. Následující část programu přepisuje bajt po bajtu obsah obrazovkové RAM. Začátek obrazovkové RAM je zkopírován do ukazatele BLOCK ($C329 až $C330). Potom jsou vynulovány X-registr (vyšší část čitače) a Y-registr (nižší část čitače a indexregistr). V průběhu smyčky STORE ($C336 až $C33C) je 256 bajtů (první blok) zkopírováno rutinou OUTCHR. Kdyby měl být přenášen již 4.blok, následoval by skok na konec programu ($C340), jinak by byl zvýšen obsah vyšší části čitače v X-registru ($C342), nastavena nová bázovací adresa na další blok ($C343) a je zkopírováno dalších 256 bajtů. Tento program se vyvolává NMI impulzem (viz kap.17).

POKE 792,9:POKE 793,195 nastaví ukazatel přerušení na startovací adresu $C39 naší právě vytvořené rutiny. Jakmile naplníme obrazovku libovolným libovolným obsahem stisk RESTORE obsah zaznamená na disk. Potom lze takto uložené obrázky za pomoci druhé rutiny zase vyvolat zpět na obrazovku. Každý takový obrázek ale musí mít také vlastní název datového souboru.

19. Výpis obsahu diskety

Oproti datasetu nabízí floppy nejen vyšší přenosovou rychlost, ale i rychlejší přístup k určitému datovému souboru. K tomu obsahuje každá disketa výpis obsahu označený jako $ v názvu souboru.

Povelem LOAD "$",8 se přehraje do RAM počítače obsah diskety. Na zlost je jen to, že Basic program, který byl v RAM uložen je smazán. Je na nás, abychom si napsali rutinu, která vypíše obsah diskety, aniž by se přemazala RAM pro Basic. Potřebné softwarové prostředky jsou nám již dávno známy a chybí jen návrh struktury programu pro výpis obsahu diskety. Výpis obsahu diskety otevřeme jako programový datový soubor (sek.

adresa 0) tak, aby neohrozil Basic-program. Všimněme si teď jednotlivých bajtů výpisu obshu disku. První dva bajty udávají startovací adresu $0401, která udává kam je umístěn soubor nahrávaný povelem LOAD. Pak následuje tzv. Link-adresa, která v normálním Basic-programu ukazuje na začátek dalšího programového řádku. Ve výpisu obsahu floppy nemá žádný význam, a obsahuje jen pro formu adresu $101. Teprve potom následují dva důležité bajty s číslem driveru a nejméně 27 bajtů s názvem diskety, včetně ID a formátovací značkou. Na konci je přidán nulový bajt udávající konec řádku. Za tímto řádkem, udávajícím název diskety následují zápisy jednotlivých datových souborů, které jsou sestaveny podle stejného schematu. Každý datový zápis začíná link-adresou ($0101) nasledovanou počtem obsazených bloků a názvem datového souboru včetně jeho typu. Každý datový zápis končí nulovým bajtem. Konec výpisu obsahu diskety tvoří zadání volných bloků následované třemi nulovými byty. Teď, když známe organizaci obsahu diskety, navrhneme si struktogram programu:

mazání obrazovky C301-C303
obsah otevřít jako program.soubor C306-C31E
vstup dat ze souboru C321-C323
přeskočení 2 bajtů (startadr) C326-C329
přeskočení 1 bajtu (low linkadr) C32C
    přeskočení 1 bajtu (hi linkadr) C32F
    čtení 2 bajtů (drive, poč.bloků) C332-C330
        čtení znaku názvu souboru C340-C343
    opakování až znak = $00 C346-C348
    přejít na další řádek obrazovky C34A-C34C
    přečíst 1 bajt C34F
opakovat až bajt = $00 C352-C354
vstup dat nastavit zpět na klávesnici C356
otevřený soubor uzavřít C359-C35B

Adresové údaje na pravé straně struktogramu označují programové části listingu. Nejdříve je otevřen a naplněn datový soubor. První čtené bajty udávají start-adresu v Basic RAM a jsou ignorovány tak jako další dva bajty link-adresy. Další dva bajty udávají číslo driveru a od druhého řádku počet obsažených bloků.Ten musí být zadán jako 16- bitové číslo. Potom jsou čteny bajty názvu a typu souboru až do nulového bajtu na konci. Pokud je kursor nastaven na začátek následujícího řádku obrazovky, čte počítač další bajty. Když nebude další bajt roven $00, je brán jako lowbyte (nižší bajt) link-adresy a napíše se další datový zápis. Když je roven tento bajt $00, pak se datový soubor uzavře.

Výstup 16-ti bitového čísla se děje rutinou na adrese $BDCD, která toto číslo interpretuje jako dekadické. Před vyvoláním této rutiny musí být dané číslo v A a X registru (v X je nižší bajt). Procesor uloží nižší bajt do zásobníku, čte potom vyšší bajt znaku a zkopíruje jej do Y-registru. Pak je ze zásobníku zkopírován jeho vrchol do X- registru a tím je volný střadač pro přijetí vyššího bajtu. Rutina OUTADR obstará výstup čísla na obrazovku. Vyvolání rutiny je pomocí SYS49921. Listing programu vypadá takto:

```
                OPEN EQU $FFC0
                CLOSE EQU $FFC3
                INCMD EQU $FFC6
                CMDOFF EQU $FFCC
                INCHAR EQU $F13E
                OUTCHR EQU $F1CA
                OUTADR EQU $BDCD
C300 24
C301 A9 93 LDA #$93
C303 20 CA F1 JSR OUTCHR
C306 A9 01 LDA #$01
C308 85 B7 STA $B7
C30A A9 01 LDA #$01
C30C 85 B8 STA $B8
C30E A9 00 LDA #$00
C310 85 B9 STA $B9
C312 A9 08 LDA #$08
C314 85 BA STA $BA
C316 A9 00 LDA #$00
C318 85 BB STA $BB
C31A A9 C3 LDA #$C3
```

```
C31C 85 BC STA $BC
C31E 20 CO FF JSR OPEN
C321 A2 01 LDX #$01
C323 20 C6 FF JSR INCMD
C326 20 3E F1 JSR INCHAR
C329 20 3E F1 JSR INCHAR
C32C 20 3E F1 JSR INCHAR
C32F 20 3E F1 NEXTLI JSR INCHAR
C332 20 3E F1 JSR INCHAR
C335 48 PHA
C336 20 3E F1 JSR INCHAR
C339 A8 TAY
C33A 68 PLA
C33B AA TAX
C33C 98 TYA
C33D 20 CD BD JSR OUTADR
C340 20 3E F1 NEXTCH JSR INCHAR
C343 20 CA F1 JSR OUTCHR
C346 C9 00 CMP #$00
C348 DO F6 BNE NEXTCH
C34A A9 OD LDA #$OD
C34C 20 CA F1 JSR OUTCHR
C34F 20 3E F1 JSR INCHAR
C352 C9 00 CMP #$00
C354 DO D9 BNE NEXTLI
C356 20 CC FF JSR CMDOFF
C359 A9 01 LDA #$01
C35B 20 C3 FF JSR CLOSE
C35E 60 RTS
```

20. Zobrazení reálných čísel

V jedné z prvních kapitol jsme si rozebrali strukturu paměťové buňky a přitom jsme narazili na 8 bitů, které mají přidělené určité váhové faktory. Součet všech váhových faktorů bitů s obsahem 1 dává dekadickou hodnotu bajtu. Trochu komplikovanější je to u 16-ti bitových čísel, u kterých rozlišujeme zda se jedná o adresu nebo celočíselnou konstantu (číslo integer). Jedná-li se o adresu (hodnota 0 až 65535), její hodnotu dostaneme sečtením všech váhových faktorů bitů o obsahu 1. Celočíselné hodnoty leží v rozsahu -32767 až +32767. Hodnota je daná sečtením váhových faktorů ale jen u kladných čísel. Kladná celá

čísla lze poznat podle 15.bitu, který je roven 0. Je to tzv. znamenkový bit. Záporné číslo je označeno 1 v 15.bitu. Hodnota záporného čísla je zakódována ve formě dvojkového doplňku a získáme ji tak, že invertujeme všechny bity, sečteme jejich váhové faktory a nakonec k tomuto číslu přičteme 1.

Ještě složitější je to u reálných čísel. K jejich uložení v paměti potřebujeme 5 bajtů. Reálné číslo se ukládá jako mantisa a exponent. Mantisa zabírá 4 bajty a exponent 1 bajt. Celočíselná hodnota se pak udává mantisa krát mocnina (exponent) se základem 2. Je-li např. M=1 a E=0, pak je hodnota tohoto reálného čísla 1*(2 na 0) = 1*1 = 1. Váhové faktory bitů v exponentu odpovídají faktorům normálního bajtu. Abychom dostali hodnotu exponentu, musíme od takto dosažené hodnoty odečíst 129 ($81). Konečná hodnota mantisy je sumou váhových faktorů bitů s obsahem 1 (přičemž váhové faktory jsou záporné mocniny čísla 2 počínaje od 6.bitu 1.bajtu mantisy -1 až -31) a čísla 1. Např. jsou v mantise nasazeny bity faktoru 2 na -3 a 2 na -5. Pak je dekadická hodnota:

1+(2 na-3)+(2 na-5) = 1+0,125+0,03125=1,15625

Reálné číslo má také znaménkový bit, takže můžeme vyjádřit i zápornou mantisu (7.bit 1.bajtu mantisy = 1). Nakonec ještě jeden příklad. V ROM C64 se nachází na adrese $BAF9 pět hexadec.čísel 84 20 00 00 00. Dekadickou hodnotu zjistíme ve třech krocích:

Exponent $84 - $81 = 3

Mantisa $20000000 =%00100000000 = 1+0,25 = 1,25

Hodnota = Mantisa * 2 na Exponent =1,25*2 na3 =10

Pět bajtů s hodnotou $8420000000 je tedy dekadicky 10.

FAC - střadač pro reálná čísla

Výpočty v celé šíři typu real nelze provádět najednou v registrech procesoru. K tomu musíme využít pomoci interpretu Basicu. Interpret má ve stránce 0 dva speciální aritmetické registry pro pohyblivou řádovou čárku - FAC1 a FAC2. Oba registry mají

délku 5 bajtů - FAC1 od adresy $61 do $65 a FAC2 od $69 do $6D. Dříve, než začneme s výpočty pomocí FAC1 a FAC2, musíme se seznámit s přemísťovacími rutinami. V tabulce jsou uvedeny rutiny, které můžeme zabudovat do svých programů:

Instrukce	název	funkce
JSR $AEF7	CHKCLS	test na levou závorku
JSR $AEFA	CHKOPN	test na pravou závorku
JSR $AEFD	CHKCOM	test, jestli bajt programu je čárka
JSR $B79E	BYTE	přesune bajt z programu do registru X
JSR $AD8A	REAL	převede bajt z prog. jako real do FAC1
JSR $AD9E	FRMEVL	test na string
JSR $B08B	PTRGET	hledá proměnnou podle jména pak:$0D=typ (FF=string,00=číslo) A=lowadr, Y=hiadr
JSR $B47D	STRING	rezervuje místo pro string, A=počet znaků pak $61=délka, $62=lowadr, $63=hiadr
JSR $B1AA	FC1INT	přesune FAC1 jako celé číslo do Y (low) a A (high)
JSR $B7F7	FC1ADR	přesune FAC1 jako adresu do Y (low) a A (high)
JSR $B391	INTFC1	přesune obsah Y-registru (low) a A (high) jako celé číslo do FAC1
LDX #$90	ADRFC1	obsah $63 (low) a $62 (high) přesune SEC do FAC1 jako adresu JSR $BC49

Nejprve stačí naplnění FAC1 hodnotou, která je oddělena čárkou od příkazu SYS (rutina REAL). Program, který začíná na adrese 49920 začne instrukcí JSR COMMA a JSR REAL. Příkaz SYS 49920, 1.43E-10 je přenese hodnotu 1.43E-10 do FAC1.

Jak snadno změnit ukazatel

V jedné z minulých kapitol jsme startovali program stiskem tlačítka RESTORE. K tomu ale musel být nastaven NMI ukazatel na startovací adresu programu. Adresa 792 obsahovala nižší část a adresa 793 vyšší část startovací adresy. K tomu slouží tento program:

```
                COMMA EQU $AEFD
                REAL EQU $AD8A
                FC1ADR EQU $B7F7
C300 20 FD AE JSR COMMA
C303 20 8A AD JSR REAL
C306 20 F7 B7 JSR FC1ADR
C309 84 FB STY $FB
C30B 85 FC STA $FC
C30D 20 FD AE JSR COMMA
C310 20 8A AD JSR REAL
C313 20 F7 B7 JSR FC1ADR
C316 48 PHA
C317 98 TYA
C318 A0 00 LDY #$00
C31A 91 FB STA ($FB),Y
C31C C8 INY
C31D 68 PLA
C31E 91 FB STA ($FB),Y
C320 60 RTS
```

Vyvolání SYS 49920,792,50000 má nastavit NMI ukazatel na hodnotu 50000 a rozložit ji do dvou bajtů 792,793. První část programu (do adresy $C306) přenese adresu ukazatele, která je za čárkou za příkazem SYS (v našem případě 792) do FAC1. Pak se tato adresa rozloží na nižší a vyšší část a přenese do Y (low) a A (high). Tato adresa je hned uložena do $FB a $FC ($C309, C30B). Pak je stejným způsobem zpracována startadresa (zde 50000). Hi-byte adresy je uložen do zásobníku ($C316) a Lo-byte do A ($C317), řádky $C318 a $C31A naplní Lo-byte ukazatele na adrese ($FB). Potom se index zvýší o 1 a je naplněna hodnota Hi-byte ukazatele. Nakonec test:

SYS 49920,49980,22222 obsadí ukazatel na adrese 49980 hodnotou 22222. Ověříme si to povelem PRINT PEEK

(49980)+256*PEEK (49981), který musí zobrazit námi změněný obsah ukazatele 22222.

21. Aritmetika s čísly typu REAL

V minulé kapitole jsme si ukázali, jak lze naplnit registr FAC hodnotou, která je při volání programu uvedena za příkazem SYS. Pro aritmetické podprogramy se ale lépe hodí volání Basic funkcí USR. Tato funkce předá argument ve FAC1 a provede podprogram, jehož startovací adresa je na adresách 785 a 786. Po ukončení podprogramu instrukcí RTS se přenese obsah FAC1 jako výsledek funkce do Basicu. Příklad: POKE 785,0:POKE 786,195 zapíše adresu 49920=256*195 do ukazatele startadresy. PRINT USR (-4.789) vyvolá podprogram na předem zadané adrese a uloží číslo -4.789 do FAC1. Po návratu je obsah registru FAC1 vypsán na obrazovku.

Jednoduché čtení ukazatele

Následující rutina vypočítá ze dvou po sobě jdoucích bajtů dekadickou adresu. Před prvním vyvoláním podprogramu se musí zadat startovací adresa: POKE785,0:POKE 786,195. Příkaz PRINT USR(adresa) pak vytiskne obsah bajtů adresa a adresa+1 jako jedno dek.číslo:

```
                FC1ADR EQU $B7F7
                ADRFC1 EQU $BC49
C300 20 F7 B7 JSR FC1ADR
C303 84 FB STY $FB
C305 85 FC STA $FC
C307 A0 00 LDY #$00
C309 B1 FB LDA ($FB),Y
C30B 85 63 STA $63
C30D C8 INY
C30E B1 FB LDA ($FB),Y
C310 85 62 STA $62
C312 A2 90 LDX #$90
C314 38 SEC
C315 20 49 BC JSR ADRFC1
C318 60 RTS
```

Program převádí předaný argument do 2-bajtové adresy ($C300) a ukládá ji na adresy $FB a $FC ($C303 a $C305). Obsah takto uložené adresy se musí uložit do buňky $63 jako Lo-byte ($C307 až $C30B). Buňka $62 obsahuje Hi-byte, který je obsahem adresy ($FB/FC)+1 - ($C30D až $C310)). Zbytek programu převádí obsah $62 a $63 na číslo typu real (viz kap.20), které je použito při návratu do Basicu jako výsledek funkce USR.

Speciální aritmetické rutiny

Tyto rutiny můžeme používat pro výpočty ve vlastních programech ve ve strojovém kódu. Ukážeme si to na příkladu: U statistických výpočtů se často objevuje tzv. faktoriál, jehož výpočet je dán vztahem:

$$n! = n * (n-1) * (n-2) * .....*1$$

tedy např.: $4! = 4 * 3 * 2 * 1 = 24$

Následujícím podprogramem se povelem PRINT USR (4) spočítá 4! :

```
                FC1INT EQU $B1AA
                INTFC1 EQU $B391
                FC1FC2 EQU $BC0C
                RESULT EQU $BA2B
                ILEGAL EQU $B248
C300 20 AA B1 JSR FC1INT
C303 84 FB STY $FB
C305 C9 00 CMP #$00
C307 F0 03 BEQ LEGAL
C309 4C 48 B2 JMP ILEGAL
C30C 20 91 B3 LEGAL JSR INTFC1
C30F 20 0C BC NEXT JSR FC1FC2
C312 C6 FB DEC $FB
C314 F0 0D BEQ END
C316 A4 FB LDY $FB
C318 A9 00 LDA #$00
C31A 20 91 B3 JSR INTFC1
C31D 20 2B BA JSR RESULT
C320 4C 0F C3 JMP NEXT
C323 60 END RTS
```

Rutina FC1INT přenese argument funkce USR do registru Y (L-byte) a do střadače (H-byte). Do $FB bude uložen L-byte

($C303). Obsah střadače musí být stále 0, neboť faktoriál z čísla většího než 256 by vedl k přetečení. Je-li ve střadači jiná hodnota než 0, vypíše ROM rutina na adrese $B248 chybové hlášení "ILLEGAL QUANTITY" ($C309). Je-li vše v pořádku, provede rutina INTFC1 přenos zpět do FAC1. Potom začíná vlastní výpočet na návěští NEXT. Tato smyčka používá (v FAC1 uložený) argument funkce USR jako násobenec a o 1 zmenšený argument jako násobitel. Při každém dalším průchodu smyčkou se stane doposud získaný výsledek násobencem a minulý násobitel zmenšený o 1 se stane novým násobitelem a to tak dlouho, až je násobitel roven 0. Poslední výsledek (příp. poč.hodnota) se kopíruje do FAC2 ($C30F) rutinou FC1FC2. Jednoduchý dekrement ($C312) sníží násobitel v buňce $FB. Pokud je hodnota v buňce $FB rovna 0, končí program odkazem do Basicu, přičemž je výsledek přenesen do FAC1 ($C314). Pro všechny ostatní hodnoty násobitele různé od nuly přenáší rutina INTFC1 obsah střadače a Y-registru do FAC1 ($C316,$C318,$C31A). Nakonec je obsah FAC1 přenesen do FAC2 a absolutní skok $C320 uzavírá výpočet.

Další ROM rutiny pro zpracování čísel real jsou uvedeny v tabulce:

Adresa	symbol	funkce
$B86A	+ souèet:	FAC1 = FAC2 + FAC1
$B853	- rozdíl:	FAC1 = FAC2 - FAC1
$BA2B	* násobení:	FAC1 = FAC2 * FAC1
$BB12	/ dìlení:	FAC1 = FAC2 / FAC1
$BF7B	mocnina:	FAC1 = FAC2 na FAC1
$BFED	EXP exponent:	FAC1 = EXP(FAC1)
$B9EA	LOG logaritmus:	FAC1 = LOG(FAC1)
$BF71	SQR odmocnina:	FAC1 = SQR(FAC1)
$E26B	SIN sinus:	FAC1 = SIN(FAC1)
$E264	COS kosinus:	FAC1 = COS(FAC1)
$E2B4	TAN tangens:	FAC1 = TAN(FAC1)
$E30E	ATN arcustg:	FAC1 = ATN(FAC1)
$BC58	ABS absol.h.:	FAC1 = ABS(FAC1)
$BCCC	INT celá èást:	FAC1 = INT(FAC1)
$BC39	SGN znaménko:	FAC1 = SGN(FAC1)

22. Modifikace Basicu

Interpret Basicu C64 lze modifikovat. Ukazatel na další adresu programu se nachází v RAM a lze jej tedy měnit. Předtím, než začne počítač zpracovávat příkaz uvedený na začátku řádku, nebo příkaz uvedený za dvojtečkou, provede počítač ve smyčce interpretu instrukci:

```
A7E1 6C 08 03 JMP ($0308)
A7E4 ...
```

Přitom ukazatel na adrese $0308 obsahuje cílovou adresu skoku $A7E4, takže jde v podstatě jen o přeskočení skokové instrukce. Na adrese $A7E4 převezme počítač jeden znak z programového textu - podobně jako rutina REAL a interpretuje ho jako zkrácený Basic příkaz. Změníme-li ukazatel na adrese $0308, pak můžeme provést jakoukoliv naši rutinu. Nazveme tyto rutiny USD (User - Defined - neboli definované uzivatelem).

Můžeme uvést příklad z kapitoly 19 k vypsání obsahu diskety prostřednictvím USD povelu. Vycházejme z toho, že každý USD povel začneme zavináčem. Povel zavináč D (Directory) má vypsat obsah diskety. K tomu potřebujeme program, který do ukazatele $0308 při vyhodnocení povelu vloží startovací adresu naší rutiny:

```
C300 A9 3F LDA #$3F
C302 8D 08 03 STA $0308
C305 A9 C3 LDA #$C3
C307 8D 09 03 STA $0309
C30A 60 RTS
```

Tento program (adresa $C300 až $C30A) aktivuje povelem SYS 49920 naši dodatečnou sadu příkazů se zavináčem. Toto se může ale udělat teprve tehdy, až se na adrese $C33F skutečně nová vyhodnocovací rutina nachází. Dále inicializujeme zároveň tabulku skoků, ve které jsou umístěny startovací adresy našich USD povelů zmenšené o 1. Pokud není daný povel k dispozici, vypíše rutina z adresy $AF08 SYNTAX ERROR. Každému USD povelu lze tak přiřadit startovací adresu.:

```
C30B 07 AF TAB
```

```
C30D 07 AF
      .    .     .33D 07 AF
C33F 20 73 00 JSR $0073 , pøesune bajt z Basic prg
C342 C9 40 CMP #$40 , test na èárku
C344 D0 17 BNE IGNOR , není-li, pak zpìt
C346 20 73 00 JSR $0073 , dal▒í bajt
C349 20 13 B1 JSR $B113 , test na písmeno
C34C 90 15 BCC SYNTAX , není-li pak SYNTAX ERROR
C34E E9 41 SBC #$41 , odeète hodnotu 65
C350 0A ASL , zdvojnásobí èíslo znaku
C351 A8 TAY , A do Y
C352 B9 0C C3 LDA TAB+1,Y , adresa povelu (high)
C355 48 PHA , do zásobníku
C356 B9 0B C3 LDA TAB,Y , adresa povelu (low)
C359 48 PHA , do zásobníku
C35A 4C 73 00 IGNOR JMP $0073 , dal▒í bajt a provedení pøíkazu
C360 4C E7 A7 JSR $0079 , obnova posl.znaku
C363 4C 08 AF SYNTAX JMP $AF08 , chybové hlá▒ení
```

Vlastní program pro vyhodnocení povelů začíná na adrese $C33F a ukládá nejprve znak z programového textu do střadače. Nejedná-li se přitom o USD povel, který by byl signalizován, jde počítač na IGNOR a pak se vrací přes JMP $A7E7 k vyhodnocení běžného povelu nebo příkazu ve smyčce interpretu. Interpret tedy pokračuje normálně dál. Kdyby ale rutina $0073 zjistila zavináč, pak by ta samá rutina vložila do střadače další znak ($C346), který by byl testován na $C349. Jsou dovoleny jen znaky A až Z, všechny ostatní by vedly k SYNTAX ERROR ($C34C).

Příslušný znak máme ve střadači, ale jak zajistit patřičný vstup do tabulky skoků, která obsahuje startovací adresu USD rutiny? Od střadače odečteme číslo $41 a dostaneme tak pořadí v abecedě ($C34E). Toto číslo, násobené dvěma udává relativní polohu adresy v tabulce. Adresa povelu pro písmeno D se tedy nachází jako šestý a sedmý bajt v tabulce, počítáme-li od nuly. TAY přenese právě vypočtenou hodnotu do Y-registru. Následující 4 instrukce přenesou potom adresu z tabulky do zásobníku. Instrukce JMP $0073 zajistí vyzvednutí dalšího znaku z Basic programu. Protože ROM rutina není volána instrukcí JSR, RTS instrukce odstartuje USD povel.

Všechny položky tabulky adres pro rutiny nových příkazů jsou v našem případě zaplněny adresou $AF07 (RTS), aby nezpůsobily neexistující USD povely chaos v programu.

Nyní si ukážeme příklad nového Basic povelu zavináčL pro mazání skupiny řádků. L 20-50 maže řádky 20-50 v Basic programu. Není-li takové číslo řádku, je automaticky vzato další číslo. Samozřejmě je možné i zadání L-20 nebo L40- pro mazání od začátku nebo konce programu. Celá definice má 4 kroky:

1. Uložíme do paměti program pro testování povelů z předchozího příkladu.
2. Spustíme jej příkazem SYS49920
3. Zapíšeme program pro provedení nových povelů:

```
c400 90 09 f0 04 c9 ab f0 03
c408 4c 08 af 20 6b a9 20 13
c410 a6 20 79 00 f0 0c c9 ab
c418 d0 ee 20 73 00 20 6b a9
c420 d0 e6 a5 5f 85 fb a5 60
c428 85 fc a5 14 05 15 f0 39
c430 20 13 a6 a0 01 b1 5f f0
c438 30 85 fe 88 b1 5f 85 fd
c440 38 a5 2d e5 fd 85 14 a5
c448 2e e5 fe 85 15 a2 ff a0
c450 ff e8 c8 b1 fd 91 fb c0
c458 ff d0 04 e6 fe e6 fc c4
c460 14 d0 ef e4 15 d0 ea f0
c468 15 18 a5 fb 69 02 85 fd
c470 a5 fc 69 00 85 2e a9 00
c478 a8 91 fb c8 91 fb 20 59
c480 a6 20 33 a5 4c 86 e3 ae
```

Poprvé za pomoci MONITORU, později přímo z kazety nebo diskety.

4. Zapíšeme startovací adresy rutin pro nový příkaz Basicu do tabulky (startovací adresa - 1). Pro naši mazací rutinu to zařídí: POKE 49953,255:POKE 49954,195.

Všechny rutiny pro nové povely by měly končit instrukcí JMP $A7AE, zajišťující skok na začátek smyčky interpretu. Povely provádí instrukci JMP $E386 k zajištění skoku do READY. Autor

přeje závěrem mnoho úspěchů při programování, na které se určitě s nově získanými znalostmi vrhnete.

Mýlit se je lidské,

ale má-li se člověk

správně mýlit,

potřebuje počítač!

Přehled instrukcí CPU 6502/6510

Přenos

LDA - Load Accumulator - naplň střadač
LDX - Load X register - naplň registr X
LDY - LOAD Y register - naplň registr Y
STA - Store Accumulator - ulož střadač
STX - Store X register - ulož registr X
STY - Store Y register - ulož registr Y
PHA - Push Accumulator - ulož střadač do zásobníku
PLA - Pull Accumulator - naplň střadač ze zásobníku
PHP - Push Proc.status - ulož statusreg. do zásobníku
PLP - Pull Proc.status - naplň statusreg. ze zásobníku
TAX - Transfer A to X - přenes střadač do registru X
TXA - Transfer X to A - přenes registr X do střadače
TAY - Transfer A to Y - přenes střadač do registru Y
TYA - Transfer Y to A - přenes registr Y do střadače
TSX - Transfer ST to X - přenes stackpointer do X
TXS - Transfer X to ST - přenes X do stackpointru

Skoky

JMP - Jump - skoč na adresu
JSR - Jump to Subroutine - skoč na podprogram
RTS - Return from Subr. - návrat z podprogramu
RTI - Return from Inter. - návrat z přeřušení
BEQ - Branch on Equal - skoč, je-li Z=1
BNE - Branch on Not Equal - skoč, je-li Z=0
BMI - Branch on Minus - skoč, je-li N=1
BPL - Branch on Plus - skoč, je-li N=0

BCS - Branch on Carry Set - skoč, je-li C=1
BCC - Branch on Carry Clr - skoč, je-li C=0
BVS - Branch Overflow Set - skoč, je-li V=1
BVC - Branch Overflow Clr - skoč, je-li V=0

Statusregistr

SEC - Set Carry - nastav C=1
SED - Set Decimal - nastav D=1
SEI - Set Interrupt - nastav I=1
CLC - Clear Carry - nastav C=0
CLD - Clear Decimal - nastav D=0
CLI - Clear Interrupt - nastav I=0
CLV - Clear Overflow - nastav V=0

Aritmetika a logika

CMP - Compare Mem and Acc - porovnej obsah paměti a A
CPX - Compare Mem and X - porovnej obsah paměti a X
CPY - Compare Mem and Y - porovnej obsah paměti a Y
ADC - Add Mem to A (Carry)- přičti obsah paměti k A
SBC - Subtract Mem from A - odečti obsah paměti od A
DEC - Decrement Mem by 1 - odečti od obsahu paměti 1
INC - Increment Mem by 1 - přičti k obsahu paměti 1
DEX - Decrement X by 1 - odečti od obsahu X 1
INX - Increment X by 1 - přičti k obsahu X 1
DEY - Decrement Y by 1 - odečti od obsahu Y 1
INY - Increment Y by 1 - přičti k obsahu Y 1
AND - AND Mem with A - proveď AND obsahu paměti a A
EOR - EXOR Mem with A - proveď EXOR obsahu paměti a A
ORA - OR Mem with A - proveď OR obsahu paměti a A
BIT - Bit Test Mem with A - test bitů paměti s bity střadače

Posun a rotace

ASL - Shift Left One Bit - posun vlevo o jeden bit
LSR - Shift Right One Bit - posun vpravo o jeden bit
ROL - Rotate One Bit Left - rotace o jeden bit vlevo
ROR - Rotate One Bit Right- rotace o jeden bit vpravo

Jiné

NOP - No Operation - nedělej nic
BRK - Break - přerušení BRK

Přenos dat Operand

```
#$NN $HHLL $LL $HHLL,X $HHLL,Y ($LL,X) ($LL),Y $LL,X $LL,Y
LDA A9 AD A5 BD B9 A1 B1 B5 -
LDX A2 AE A6 -  BE -  -  -  B6
LDY A0 AC A4 BC -  -  -  B4 -
STA -  8D 85 9D 99 81 91 95 -
STX -  8E 86 -  -  -  -  -  96
STY -  8C 84 -  -  -  -  94 -
-----------------------------------------------------------------
PHA 48 PHP 08 TAX AA TAY A8 TSX BA
PLA 68 PLP 28 TXA 8A TYA 98 TXS 9A
```

Skokové příkazy

```
JMP 4C JMP() 6C JSR 20 RTS 60 RTI 40
BEQ F0 BCS B0 BMI 30 BVS 70
BNE D0 BCC 90 BPL 10 BVC 50
```

Příkazy pro flagy

```
SEC 38 SED F8 SEI 78 CLV B8
CLC 18 CLD D8 CLI 58
```

Příkazy posunu a rotace

```
Operand: A $HHLL $LL $HHLL,X $LL,X
ASL 0A 0E 06 1E 16
LSR 4A 4E 46 5E 56
ROL 2A 2E 26 3E 36
ROR 6A 6E 66 7E 76
```

Aritmetické a logické příkazy Operand

```
#$NN |HHLL $LL $HHLL,X $HHLL,Y ($LL,X) ($LL),Y $LL,X $LL,Y
ADC 69 6D 65 7D 79 61 71 75 -
SBC E9 ED E5 FD F9 E1 F1 F5 -
INC -  EE E6 FE -  -  -  F6 -
DEC -  CE C6 DE -  -  -  D6 -
AND 29 2D 25 3D 39 21 31 35 -
EOR 49 4D 45 5D 59 41 51 55 -
ORA 09 0D 05 1D 19 01 11 15 -
BIT -  2C 24 -  -  -  -  -  -
CMP C9 CD C5 DD D9 C1 D1 D5 -
CPX E0 EC E4 -  -  -  -  -  -
CPY C0 CC C4 -  -  -  -  -  -
-----------------------------------------------------------------

INX E8 DEX CA INY C8 DEY 88
NOP EA BRK 00
```

Vliv instrukcí na flagy statusregistru

```
Instrukce                                Flag: N V B D I Z C
LDA LDX LDY PLA TAX TXA TSX TAY TYA            ! - - - - ! -
PLP RTI                                        ! ! ! ! ! ! !
ASL LSR ROL ROR CMP CPX CPY                    ! - - - - ! !
BRK                                            - - ! - ! - -
SEC CLC                                        - - - - - - !
SED CLD                                        - - - ! - - -
SEI CLI                                        - - - - ! - -
CLV                                            - ! - - - - -
ADC SBC                                        ! ! - - - ! !
INC DEC INX.DEX INY DEY AND EOR ORA            ! - - - - ! -
BIT                                            ! ! - - - ! -
```

Neoficiální instrukce CPU 6502/6510

Aritmetika a logika

ALR - Acc AND Data, LSR result - logické AND střadače a čísla, s výsledkem provede LSR

ARR - Acc AND Data, ROR result - logické AND střadače a čísla, s výsledkem provede ROR

ASO - ASL then ORA with Acc - ASL a pak s výsledkem provede ORA se střadačem

AXS - Store result A AND X - uloží výsledek A AND X

DCM - DEC Mem then CMP with Acc - sníží obsah paměti o 1, pak provede CMP se střadačem

INS - INC Mem then SBC with Acc - zvýší obsah paměti o 1, pak provede SBC se střadačem

LSE - LSR then EOR result with A - provede LSR a s výsledkem EOR se střadačem

MKA - Acc AND #$04 to Acc - provede AND obsahu střadače a čísla #$04

MKX - X AND #$04 to X - provede AND obsahu registru X a čísla #$04

OAL - ORA Acc with #$EE, then - provede ORA střadače a #$EE, AND with data, then TAX pak AND s pamětí, nakonec TAX

RLA - ROL Mem, then AND with A - provede ROL s pamětí, pak AND se střadačem

RRA - ROR Mem, then ADC to Acc - provede ROR s pamětí, pak ADC se střadačem

SAX - A AND X, then SBC Mem, - provede A AND X, pak odečte store to X data a výsledek uloží do X

XAA - X AND Mem to Acc - provede X AND obsah paměti, výsledek uloží do A

Přesun, skoky, jiné

LAX - Load Mem to A and X - přesune obsah paměti do A a X

SKB - Skip Byte - přeskočí jeden bajt

SKW - Skip Word - přeskočí dva bajty

CIM - Crash Intermediate - rozpad systému

Instrukce kódy

```
NOP 1A 3A 5A 7A DA FA
SKB 80 82 C2 E2 04 14 34 44 54 64 74 D4 F4
SKW 0C 1C 3C 5C 7C DC FC
CIM 02 12 22 32 42 52 62 72 92 B2 D2 F2
```

```
Operand: #$NN $HHLL $HHLL,X $HHLL,Y $LL $LL,X ($LL,X) ($LL),Y
ALR 4B -  -  -  -  -  -  -
ARR 6B -  -  -  -  -  -  -
ASO 0B 0F 1F 1B 07 17 03 13
AXS -  8F -  -  87 97(Y) 83 93
DCM -  CF DF DB C7 D7 C3 D3
INS -  EF FF FB E7 F7 E3 F3
LAX -  AF -  BF A7 B7 A3 B3
LSE -  4F 5F 5B 47 57 43 53
MKA -  9F -  -  -  -  -  -
MKX -  9E -  -  -  -  -  -
OAL AB -  -  -  -  -  -  -
RLA 2B 2F 3F 3B 27 37 23 33
RRA -  6F 7F 7B 67 77 63 73
SAX CB -  -  -  -  -  -  -
XAA 8B -  -  9B -  -  -  -
```

Scanned by oldsoft 2013