klub elektroniky Tlmače

ASSEMBLER PRE ZAČIATOČNÍKOV

1987

Dostává se Vám do ruky další publikace, která je svým obsahem zaměřena výhradně na mikropočítače ATARI. Je určena nejširším vrstvám zájemců o programování na počítačích ATARI v jazyce symbolických adres neboli assembleru. Svým obsahem bude mít co říci jak úplným začátečníkům, tak osobám v programování v assembleru již kovaným, neboť obsahuje řadu zajímavých aplikací.

Dílko sestává z několika tématicky oddělených kapitol a přílohy. Právě začátečníkům bych po přečtení úvodních pěti kapitol doporučil velice pozorně si prostudovat právě onu přílohu a takto získané vědomosti až potom uplatnit v praxi.

Na tomto místě, bych také rád uvedl, že publikace vznikla překladem knihy Marka Chasina „Programming in assembly language on ATARI computers for beginners.“ V mém případě se jedná o první pokus na překladatelském poli. Mnohé nedostatky mohly vzniknout také tím, že rukopis byl přepisován osobou programování nerozumějící (časté příkazy REM ve výpisech BASICovských programů bez dalšího komentáře). Kdykoliv na nějakou chybu narazíte, prosím o tolerantní poshovění. Za pochopení děkuji.

Svůj úvodní vstup bych zakončil úslovím: programování zdar a na ATARI zvlášť!

překladatel

PRVNÍ ČÁST

PŘEDSLOV

Protože jste si koupil tuto knihu s začal ji pročítat, pravděpodobně vlastníte nebo máte přístup k počítači ATARI a zajímáte se o programovaní v jiném jazyce než je BASIC. Jak už jistě víte, počítače ATARI patří mezi nejimpozantnější domácí počítače, ale mnohé z jejich speciálních vlastností nejsou pomocí BASICu přístupné.

Tato kniha je určena k výučbě programovaní v assembleru každému, kdo rozumí ATARI BASICu. Ano, každému! Pravděpodobně jste už četl i jiné knihy a články, které vytvářejí o assembleru jakousi aureolu, nebo jiné užívají označení strojový kód, jako by to byl tajný kód k odemčení dveří k bagdádskému pokladu pokladů. Samozřejmě jenom těm privilegovaným je dáno nahlédnout do komnaty s pokladem.

Ale kdeže! Každý, kdo programoval v BASICu nebo v jiném podobném jazyce, se může naučit programovat v assembleru, jestliže si to skutečně přeje a má-li o jazyku správné instrukce. Tato kniha poskytuje to, co potřebujete. Každý programovací jazyk, i BASIC nebo PILOT nebo FORTH nebo dokonce assembler, má vlastní slova, která slouží jisté operaci. Jedním z příkladů je PRINT v BASICu, který posílá informace na vaši obrazovku. Kombinace těchto slov a způsob, jakým musí být seřazeny, aby počítač pracoval, jak chcete, je nazýván syntaxí jazyka.

V této knize se naučíte syntaxi assembleru a častým uváděním příkladů se také naučíte, jak užívat assembler, aby váš ATARI vykonával úlohy, které jsou v BASICu buď nemožné nebo jsou dvěstěkrát pomalejší. Tam, kde je to vhodné, jsou příklady cele doloženy jak častými poznámkami, tak důkladnými rozbory o účelu, programovacích technikách a teorii každého programu. Tento rozbor vám dovolí předčit příklady a psát své vlastní podprogramy nebo dokonce celé programy v assembleru. Dále, rutiny v této knize se drží pravidel, ustanovených firmou ATARI, pro programátory programující v assembleru. Takže mohou pracovat s každým počítačem ATARI od typu 400 až po ten nejpokročileji typ 1450 XLD.

Příklady jsou uvedeny jak v assembleru, tak tam, kde je to možné, také v BASICu, jehož programy zahrnují tyto rutiny assembleru. Tyto rutiny mohou být ihned použity ve vašich vlastních programech. Vskutku můžete používat přiložený uspořádaný tvar, abyste získali na disku všechny assemblery a programy v BASICu v této knize. Disk je připravený pracovat

nebo být upravován pro použití, o jaké máte zájem. V knize jsou také uvedeny techniky jako práce s joystickem, pohyb hráčů a střel, vstup a výstup pro všechna možná zařízení jako tiskárny, disketové jednotky, magnetofony, obrazovku a další VBI rutiny, DLI, horizontální a vertikální rolování, zvuk, grafika – zkrátka vše, o čem jste už tolikrát slyšeli, že počítač ATARI umí, ale nevěděli jste, jak na to.

Jedna celá kapitola této knihy je věnována užití assembleru a tomu, jak používat tuto knihu s tolika assemblery, vhodnými pro počítače ATARI. Assembler budete potřebovat právě tak, jako potřebujete BASIC k programování v BASICu, a tato kniha se bude všemi zabývat.

Jestliže jste dosáhli stupně, kdy vám BASIC již nestačí, a rádi byste pokročili k jazyku, který vám dá absolutní kontrolu nad všemi funkcemi vašeho pozoruhodného počítače, pak začněte kapitolu první a zjistíte, jak je to snadné.

Kdoví! Možná to budete právě vy, kdo napíše pokračování ke hře STAR RAIDERS!

KAPITOLA PRVNÍ

Úvod

Vítejte ve světě programování v assembleru pro počítače ATARI. Zatím jste se bezpochyby pokusili o programovaní v BASICu a shledali jej mocným a snadno použitelným jazykem. Ale asi jste se také střetli s věcmi, které v BASICu udělat nelze. A asi taky víte, že všechny ty znamenité hry probíhající v reálném čase nebo rychlé třídění jsou všechna programována v jakémsi nadpřirozeném jazyku, který se nazývá strojový kód. Účelem této knihy je naučit vás programovat rychlým, mocným a mnohostranně užitelným jazykem, assemblerem. Studováním této knihy se naučíte užívat všechny rafinované a mocné prostředky jednoho z nejimpozantnějších domácích počítačů ATARI.

Většina příkladů v této knize bude uvedena v BASICu, takže porozumění BASICu bude důležité pro její pochopení. Avšak mnohé programy, které lze jednoduše napsat v assembleru, své protějšky v BASICu mít nebudou. Úlohy budou uváděny v celé knize a vřele doporučuji, abyste vyzkoušeli, jak pracují. V každém případě budou uváděny i komentáře, které vám pomohou, budete-li mít nějaké problémy.

Rozmanitost programovacích jazyků

Váš počítač vlastně rozumí jen jednomu programovacímu jazyku, který se jmenuje strojový kód, jazyk počítače. Typicky program ve strojovém kódu může vypadat třeba takto:

```
1011010110100101...
```

Dříve než odhodíte knihu a vrátíte se zpět k BASICu, vyjasněme si jednu věc: vlastně nikdo neprogramuje přímo ve strojovém kódu. Dokonce mnoho programů, inzerovaných jako stoprocentně napsané ve strojovém kódu, takovými nejsou. Byly napsány v assembleru a pak přeloženy do strojového kódu. Všechny počítačové jazyky, tedy i BASIC, mají-li být vykonány, musí být někdy přeloženy do strojového kódu. Ano, je to tak. Centrální „mozek“ vašeho počítače ATARI nerozumí BASICu.

BASIC: interpretovaný jazyk

Pojďme se chvíli zamyslet nad tím, jak je vykonán program v BASICu, ve snaze lépe porozumět tomu, co jsou a v čem se liší assembler a strojový kód.

Nejdříve si v BASICu napíšeme velmi jednoduchý program:

```
10 PRINT „HELLO“
20 FOR I=1 TO 200
30 NEXT I
40 PRINT „GOODBYE“
50 END
```

Jestliže nyní „naťukáme“ RUN a stlačíme klávesu RETURN, víme, že na obrazovce se objeví slovo HELLO a po krátké pauze se za ním objeví slovo GOODBYE, následováno slovem READY, umístěným o několik řádků níže. Ale jak to vlastně proběhlo?

Kartridž obsahující ATARI BASIC je přesněji nazýván ATARI BASIC INTERPRETER. Interpreter, stejně jako nepočítačový význam slova, je někdo nebo něco, překládající informaci z jedné formy do jiné; např. z angličtiny do ruštiny nebo z BASICu do jiného jazyka. V našem případě kartridž BASICu obsahuje program, který umí převádět klíčové slova v BASICu do tvaru, který je počítačovému „mozku“ srozumitelný. Podívejme se, jak.

Když na klávesnici napíšeme řádek 10, slovo PRINT je přeloženo do kódu, který mu zodpovídá a nazývá ae token. Tento proces se nazývá tokenizace vašeho programu v BASICu a je vykonán po „naťukání“ každého řádku do vašeho počítače a stlačení klávesy RETURN. Tento proces souběžně kontroluje syntax nebo gramatická pravidla, aby byla jistota, že řádek byl napsaný správně. Jestliže nebyl, uvidíte ihned po napsaní řádku známý údaj ERROR a ještě předtím, než budete pokračovat, můžete chyby opravit. Když kartridž BASICu začne interpretování vašeho programu, tento může obsahovat i logickou chybu, ale tokenizace alespoň zajišťuje, že vnitřně je každý řádek správný.

Máme-li kompletně napsaný výše uvedený program, můžeme napsat RUN a stisknout RETURN, čímž začne interpretování programu. První věcí, kterou interpreter ví, je, že začátek programu, místo, kde musí začít, je-li napsáno slovo RUN, je řádek s nejnižším číslem. Ve skutečnosti dříve, než se tam vůbec dostane, vykoná některé nezbytné práce, jako nastavení všech v programu užitých proměnných na nulu, rušení každého

dříve použitého řetězce nebo pole, a mnoho jiných funkcí. Pak obrátí svou pozornost na řádek 10, který je přeložen do strojového kódu prostřednictvím něčeho, co nazýváme tabulka odskoků, kterou se budeme zabývat detailněji v deváté kapitole. Takže nejdříve je řádek 10 přeložen, potom vykonán a pak je strojový kód odložen, aby uvolnil místo pro následující řádek, řádek 20. Proces překladu, vykonání a odložení je opakován pro řádky 20, 30 a další.

Jestliže nyní máme vykonaný celý program o vidíme hbité READY, které nám říká, že BASIC je připraven pro nové instrukce, co myslíte, že se stene, jestliže napíšeme znovu RUN? Správně! Celý proces překladu, vykonaní a odložení každého řádku bude celý znovu zopakován. Pak opět uvidíme hbité READY. Vskutku, cely proces bude opakován tolikrát, kolikrát si usmyslíme a napíšeme slovo RUN. Jak si asi všimnete, je to velice nehospodárný proces. BASIC pokračuje opět a opět v opakování dvou ze tří kroků, překladu a odložení informace, které vlastně nejsou potřebné k tomu, aby program fungoval. Představte si, jak rychle by program byl vykonán, kdybychom mohli z toho, co potřebujeme, tyto dva kroky odsunout. Nakonec, zbavíme-li se těchto dvou kroků, zbývá jeden - vykonání.

Assembler: asemblovaný jazyk

Nyní už znáte smysl programování v assembleru!

Programujeme-li v assembleru, používáme překladač, známý jako assembler. Můžeme produkovat výkonný strojový kód, který můžeme uchovat a který může počítač přímo vykonat. Překládáme ho pouze jedenkrát a ani neodkládáme, takže dosáhneme maximální účinnosti, a tedy i maximální rychlosti. Rychlost, to je ten největší vklad, programujeme-li v assembleru. Je možné napsat program, který bude v assembleru vykonán až tisíckrát rychleji než v BASICu. Pro hry a procesy, které jsou časově náročné, jako přesun bloků paměti, třídění, prohledávání a podobné další procedury, je programování v assembleru absolutně nepostradatelné.

Další důležitou výhodou assembleru je, že programátorovi dává nad počítačem absolutní kontrolu. V BASICu je programátor často oddělován od NUTS- AND BOLTS hardwaru počítače a ztrácí tak podrobnou kontrolu nad mnoha funkcemi.

Tato kontrola je k dispozici jen díky programování v assembleru.

Interpretovaný versus asemblovaný jazyk

Rychlost a možnost kontroly – to jsou dvě výhody assembleru. Ale co nevýhody? Za prvé je to samozřejmě nutnost naučit se nový počítačový jazyk. K tomu vám má posloužit tato kniha. Za druhé ATARI BASIC je interpretovaný jazyk, zatímco assembler ne. To získává význam, když potřebujete dělat v programu změny. V BASICu jednoduše provedete změnu a znovu spustíte program. Např. chceme-li změnit výše uvedený program, můžeme psát:

```
40 PRINT „GOODBYE“;
50 PRINT „Y,ALL“
60 END
```

Nyní když spustíme program, zapíše GODDBYE Y,ALL namísto pouhého GOODBYE, jak tomu bylo dříve. Celé změna programu může velmi pomalým písařům zabrat asi 15 sekund. Tato pružnost je velkou výhodou interpretovaných jazyků. Kdybychom chtěli udělat podobnou změnu v assembleru, program by vyžadoval mnohem více psaní a pak by program musel být znovu převeden do strojového kódu. Tento proces přeměny programu z assembleru do strojového kódu trvá někdy i 15 minut a více, v závislosti na velikosti programu a užitého assembleru. Samozřejmě, náš program je krátký a nezabral by tolik času. Ale je důležité si uvědomit, že jednoduché změny v programu v assembleru nemusí trvat dlouho, ale jestliže uděláte chybu, budete muset zopakovat celý proces znovu.

Třetí nevýhodou assembleru je množství programování, které musíte udělat, chcete-li dokončit třeba i jednoduché úlohy. Například příkaz PRINT v BASICu vyžaduje, abyste napsali pouze jedno slovo, ale v assembleru může programátor potřebovat 20 – 30 řádků. Z tohoto důvodu jsou obvykle programy v assembleru velmi dlouhé.

Čtvrtou a poslední nevýhodou assembleru je obtížná čitelnost programu. Určitě je daleko srozumitelnější příkaz PRINT v BASICu, než řada instrukcí jako:

```
LDA #$01
STA CRSINH
```

nebo něco podobně hloupého. Tento problém může a měl by být zcela překonán tím, že programátoři programující v assembleru, budou na každý řádek vpisovat poznámky. Poznámky v assembleru jsou ekvivalenty příkazů REM v BASICu: pomáhají

programátorovi vzpomenout si, čeho chtěl na daném řádku docílit. Uvedeme-li výše uvedený příklad s těmito poznámkami, získá samozřejmě o něco větší smysl i pro toho, kdo assembler vůbec neovládá.

```
LDA #$01  ; k potlačení kurzoru
STA CRSINH; vložte sem 1
```

Nyní je asi srozumitelnější, co znamená, vidíme-li program inzerovaný jako „100% napsaný ve strojovém kódu“. Byl napsaný v assembleru a potom přeložen do strojového kódu. A jako takový je potom prodáván. Vykonání takových programů je všeobecně mnohem rychlejší než programů v BASICu a dodatečná kontrola programátora nad počítačem mu dovoluje dělat speciální efekty, v BASICu nemožné.

Nu a dostáváme se k dalšímu rozdílu mezi BASICem a assemblerem. BASIC patří do rodiny programovacích jazyků, které jsou označované jako vyšší jazyky. Toto názvosloví poukazuje na schopnost toho, že jeden jednoduchý příkaz umí vykonat poměrně komplikované úlohy, jako např. výše uvedený příklad PRINT. Na druhé straně ale tato jednoduchost programování izoluje programátora od hardwaru a, abych tak řekl, drží si ho „od těla“. Termín vyšší jazyky vznikl s ohledem na jazyky podobné BASICu. Mezi tisíce dalších vyšších jazyků patří PASCAL, FORTRAN, PILOT a ADA. Naopak, jazyky jako assembler nebo strojový kód jsou označovány jako jazyky nižší, protože jejich použití při programování vyžaduje pochopení hardwaru a schopnost dostat se do skutečného nitra stroje, na kterém programujete.

Práce s assemblerem

Abychom mohli assembler převést do strojového kódu, musíme použít další program, nazývaný assembler. Existují spousty výborných assemblerů, vhodných pro počítače ATARI, a techniky použité v této knize budou fungovat na všech. Kapitola šestá je věnována syntaxi a speciálním funkcím každého assembleru, ale programy v assemblerovém jazyce, které jsou uvedeny v této knize, byly vytvořeny za použití kartridže Assembler/Editor od firmy ATARI. Kapitola šestá výslovně uvádí všechny změny potřebné k použití těchto programů s každým z dalších assemblerů.

Kompilátory

Nyní uvedeme další způsob překladu programu do strojového kódu. Kompilátor je program, který převádí programy ve vyšších jazycích (např. BASIC) do strojového kódu. Tyto kompilátory všeobecně převádějí hned celý program, oproti tomu interpreter překládá postupně jednotlivé řádky. Přepsaný program, vytvořený kompilátorem, je schopen pracovat bez instalovaného BASICového kartridže a vždy bude 5 - lOkrát rychlejší než originální program v BASICu. Proč jenom 5 - lOkrát? Tyto kompilátory jsou velmi složité programy, které musí brát do úvahy každou možnou kombinací příkazů v BASICu. Proto tedy tvoří strojový kód, který vykoná v původním programu všechny správné kroky, ale vytvořené kódy optimalizovat neumí. Všeobecně tedy programy napsané v assembleru a převedeny do strojového kódu budou provedeny mnohem rychleji než programy, které byly napsány v BASICu a kompilovány. Druhou velkou nevýhodou kompilovaného kódu je jeho velikost. Např. některé podprogramy v kapitole sedmé jsou dlouhé až 100 bytů. Stejné rutiny, napsané v BASICu a kompilované, by byly dlouhé až 8000 bytů. Takovéto bychom mohli jen stěží použít jako podprogramy v programech v BASICu, jak činíme v kapitole sedmé.

Terminologie

Dříve než budeme pokračovat, popovídame si o několika termínech, kterých programátoři často používají. Je to hantýrka jejich řemesla. I když mluvíme všichni stejným jazykem, pojďme si krátce některá probrat. Hovoříme-li o paměti počítače, často slyšíme termíny ROM a RAM. ROM vzniklo z Read-Only Memory a z paměti tohoto typu lze číst, ale nelze do ní psát. Např. v ATARI jsou ROM všechny paměťové lokace vyšší než 49152, a ačkoliv v BASICu můžeme s využitím příkazu PEEK zjistit, co je tam uloženo, příkazem POKE už do ní nové hodnoty uložit nemůžeme. Ale možná se zeptáte: A co třeba PLAYER-MISSILE GRAPHICS? Vždyť celou dobu pomocí POKE do těchto lokací ukládáme!

To je pravda. Ale jestliže jste v těchto místech používali PEEK, zjistíte, že vlastně jste vůbec nic nezměnili. Hodnoty uložené v těchto lokacích není možné pomocí POKE změnit. Je to vlastně akt napsání na tu adresu, která způsobí změny, které vidíte při PMG nebo jiných aplikacích, které vyžadují zápis do paměťových lokací nad 49152.

Je to v přímém kontrastu a RAM, který byl vytvořen z Random-Access Memory. Vlastně ale jak ROM, tak RAM jsou náhodně přístupné, a RAM by měl být přesněji nazýván Read-Write Memory. Ale protože RWM je těžce vyslovitelné, přijatým termínem se stala RAM. Termín „náhodně přístupné“ poukazuje na metodu, jakou je informace přístupná a je v rozporu s postupným přístupem, což je významná metoda ukládání. Postupný přístup lépe pochopíme, když si představíme magnetofonový pásek. Abychom mohli poslouchat hudbu uprostřed pásku, musíme se dostat přes celou část, jež jí předchází, buď přehráním nebo přetočením pásku. Pro srovnání uvažujme fonografický záznam, kde stačí jednoduše zvednout přenosku a položit ji na místo, odkud chceme hudbu slyšet. Magnetofonový pásek je prostředek postupného přístupu a fonografický záznam přístupu náhodného.

Další termíny, se kterými můžete, ale nemusíte být seznámeni, jsou OS a DOS. OS znamená operační systém (Operating System), a vaše ATARI má jeden z nejlepších mikropočítačových OS. OS je celý obsažen v ROM a má za úkol řízení téměř všeho, co se uvnitř ATARI odehrává. Bez něho by se po zapnutí počítače nedělo nic. Jak bude kniha pokračovat, dovíme se, jak s tímto OS pracovat.

Snad by nebylo od věcí zmínit se, že ATARI má několik z nejlepších mikropočítačových systémů, a to proto, že OS v ATARI 400 a 800 se trochu liší od OS v 1200 XL, který je zas odlišný od OS ATARI 800 XL a 1450 XLD. Vlastně i ATARI 400 a 800 nemají tytéž OS, ale liší se verzemi paměti ROM, tzv. ROM A a B! Z kterého konce tedy začít programování pro tak mnoho OS-ů?

Teď se ukáže to nejkrásnější z OS firmy ATARI. Ta se totiž zaručila, že určité vektory v OS se nikdy nezmění. VEKTOR je ukazovátko, přesný indikátor, který říká, jak a kde najít určité rutiny v OS. Tím se vysvětluje, jak je možné psát programy, která budou fungovat nejen na ATARI 400, 800 či 800 XL, ale i na dalších generacích počítačů ATARI, o kterých se firmě dnes ještě ani nesní.

Existuje nespočetně způsobů, jak tyto vektory obejít, ale není nikde psáno, že programy, jež jich využívají, budou fungovat na všech ATARI. Proto se vřele doporučuje je nepoužívat. Samozřejmě, pokud si píšete rutiny či programy jen pro sebe, obejití vektorů je užitečné. Ovšem pokud zamýšlíte své programy prodávat, držte se v každém případě pravidla použití vektorů.

Příbuzná zkratka DOS znamená disketový operační systém, což je program, který řídí libovolné disketové jednotky připojitelné k počítači ATARI. Sestává ze dvou částí: DOS.SYS a DUP.SYS. Část DOS.SYS je po zapnuti disketové stanice nahrána do počítače a je vždy přítomna. Část DUP.SYS se nahraje, pouze když naťukáte DOS na klávesnici. DUP.SYS obsahuje ono známé DOS menu, které nám umožňuje přístup k obvyklým operacím se soubory jako je např. kopírování disku, formátování apod. Všimněte si, že v DOSu nejsou žádné zaručené vektory, i když tolik softwarových produktů je závislých na určitých konkrétních lokacích, že se o jejich změně pravděpodobně nebude uvažovat. Ale člověk nikdy neví.

Nyní, když znáte rozdíly mezi jazyky a mezi interpretery, kompilátory a assemblery, vysvětlíme si různé číslicová systémy, jichž váš počítač používá.

KAPITOLA DRUHÁ

Číslicové soustavy všeobecně

Dříve než se dostaneme k výučbě programování v assembleru, podívejme se na některé odlišné číslicové soustavy. Nejdříve to bude desítkové soustava, ta která je nám nejbližší. Desítkové soustava je založena na deseti, možná proto, že máme deset prstů, a pro naše předky bylo počítaní využívající prsty tou nejjednodušší formou aritmetiky.

Mysleme na chvíli na číslo 123. Co přesně toto číslo znázorňuje? Ze vzpomínek ze školy si vzpomeneme, že: 1 sto, 2 desítky a 3 jednotky, které nám po sečtení vytvoří 123. Existuje však i jiný způsob pohledu no toto číslo. Znamená, že každá číslice v desítkové soustavě je o jednu mocninu deseti vyšší než číslice po jeho bezprostřední pravici. Pro ty, kteří si již přesně nepamatují, co je to mocnina. Tento termín nám říká, kolikrát je základ násoben sém sebou. Např. 10 umocněno na třetí je 10*10*10=1000 nebo 10 násobeno samo sebou třikrát.

Nyní se vrátíme k číslu 123. V každé poziční číslicové soustavě číslice stojící nejvíce vpravo představuje jednotky. Proč je tomu tak? Protože tato číslice je vždy základ (v našem případě 10) umocněný na nultou. Cokoli umocněno na nultou je vždy 1, a tak toto číslo vždy představuje jednotky. V našem případě dostaneme 3*1=3. Další číslice, v našem případě 2, představuje základ 10 umocněný na první, neboli 10. Protože 2*10=20, dostaneme v našem čísle správnou prostřední číslici. Nu a zbývá číslice, která je nejvíce vlevo, 1, která představuje základ 10 umocněný na druhou, neboli sto. Proto tedy číslice představuje 1*100 neboli 100, a celé číslo pak 100+20+3 neboli 123. Znázorníme-li číslo schématicky, bude jednodušší proces sledovat. V níže uvedených příkladech se vždy pohybujeme zprava doleva. Desítkové číslo 121 např. můžeme znázornit:

Základ	Umocněn na	=	Vynásoben číslicí	=
10	0	1	3	3
10	1	10	2	20
10	2	100	1	100
			Celkem =	123

Použijme trochu komplikovanější číslo – 53798:

Základ	Umocněn na	=	Vynásoben číslicí	=
10	0	1	8	8
10	1	10	9	90
10	2	100	7	700
10	3	1000	3	3000
10	4	10000	5	50000
			Celkem =	53798

Dvojková číselná soustava

Nyní, když už umíme pracovat s desítkovou soustavou, přejdeme k odlišné číselné soustavě, k dvojkové. Proč dvojková? Počítače jsou relativně jednoduchá zařízení. Základní díl informace, který je v něm uložený, se nazývá bit, neboli Binary digIT (dvojkové číslo). Bit je nejmenší částí informace, může buď být nebo ne, 1 nebo 0. Bit je vlastně něco jako světelný vypínač. Světlo může být buď zapnuto nebo vypnuto. Neexistuje nic mezi tím. Bit v počítači se chová naprosto stejně. To vysvětluje, proč je dvojková soustava pro počítače tak přirozená.

Dvojková soustava sestává pouze ze dvou číslic, 0 a 1. Proto všemu, co je v dvojkové soustavě, počítač hned rozumí. Přestože by se naučení nové číselné soustavy mohlo zdát těžké, my to zvládneme snadno. Ve skutečnosti pracují všechny číselné soustavy stejně. Jediným rozdílem je použitý základ. Jak jsme viděli, desítková soustava používá jako základ 10. Dvojkové soustava pak číslici 2. Všimněte si, že největší číslice každé číselné soustavy je o 1 menší než základ. Např. 9 je největší číslice při základu 10, 1 je největší při základu 2. Proč? Protože musíme brát do úvahy nulu, a v každé číselné soustavě konečný počet rozdílných číslic je rovný základu daného číselného základu.

Jako ukázka toho, jak jednoduché je dvojkovou soustavu pochopit, uvedeme zvlášť příklad. Použijeme dvojkové číslo 10110110.

Základ	Umocněn na	=	Vynásoben číslicí	=
10	0	1	0	0
10	1	2	1	2
10	2	4	1	4
10	3	8	0	0

```
  10        4       16          1          16
  10        5       32          1          32
  10        6       64          0           0
  10        7      128          1         128
                                 Celkem = 182
```

Takže vidíme, že 10110110 v dvojkové soustavě je rovno 182 v desítkové soustavě. Zkusme to ještě jednou, jenže tentokrát to zkusíte sami a teprve pak to zkusíme spolu. Dvojkové číslo, které máte převést, je 01011101.

```
Základ  Umocněn na   =  Vynásoben číslicí   =
  10        0        1          1           1
  10        1        2          0           0
  10        2        4          1           4
  10        3        8          1           8
  10        4       16          1          16
  10        5       32          0           0
  10        6       64          1          64
  10        7      128          0           0
                                   Celkem = 93
```

Zvládli jste to sami? Dvojkové číslo 01011101 je stejné jako 93 v desítkové soustavě. Nyní už umíme převádět čísla z dvojkové do desítkové soustavy. Ale jak to udělat, abychom uměli převádět čísla z desítkové do dvojkové soustavy? Je to dokonce jednodušší. Vezměte své desítkové číslo a dělte je mocninami 2. Začnete s nejvyšší mocninou, která dá při dělení vašeho desítkového čísla výsledek l. Pak vždy dělte zbytek další nižší mocninou 2. Pojďme převést např. 124 do dvojkové soustavy.

```
124/128 = 0   zbytek 124
 124/64 = 1   zbytek 60
  60/32 = 1   zbytek 28
  28/16 = 1   zbytek 12
   12/8 = 1   zbytek 4
    4/4 = 1   zbytek 0
    0/2 = 0   zbytek 0
    0/1 = 0   zbytek 0
```

Číslo 124 je rovno 01111100 v dvojkové soustavě.

Hexadecimální číselná soustava

Již jsme téměř u konce s našimi číselnými soustavami. Zbývá udělat jeden krok a skončíme. Existuje ještě mnohem jednodušší způsob representace čísel, než je dvojková soustava. Nazývá se hexadecimální soustava. Hexadecimální? Hexadecimální znamená 16, a základem této soustavy je 16. Již také víte, že největší jednoduchá číslice soustavy musí být 15. Ale moment! 15 není jednoduchá číslice. Proto potřebujeme najít nějaký nový způsob, jak znázornit číslice od 10 do 15. Nejjednoduššími symboly k zapamatováni je prvních šest písmen abecedy. Proto je v šestnáctkové soustavě číslo 10 reprezentováno písmenem A, 11 – B atd. až po 15, které je reprezentováno písmenem F. Protože základem šestnáctkové soustavy je 16, hodnoty čísel se zvětšují mocninami 16ti. Uvedeme příklad. Převeďme číslo 6FC ze šestnáctkové soustavy do desítkové.

```
Základ  Umocněn na   =  Vynásoben číslicí    =
  10        0        1          C           12
  10        1       16          F          240
  10        2      256          6         1536
                                  Celkem = 1788
```

Takže 6FC v hexadecimální soustavě je rovno 1788 v desítkové soustavě. Ve většině počítačových článků a textů je šestnáctková soustava znázorňována znakem dolaru, takže správný způsob, jak znázornit desítkové číslo 1788 v šestnáctkové soustavě, je $6FC. Při převádění z desítkové do šestnáctkové soustavy budeme dělit, jak bylo uvedeno výše. Namísto děleni mocninami 2 budeme dělit mocninami 16.

```
1788/256 = 6 zbytek 252
  252/16 = 15 (F) zbytek 12
    12/1 = 12 (C) zbytek 0
```

Uvedeme si ještě jeden jednoduchý převod. Vezměme dvojkové číslo 10110111. Již víme, že je rovno desítkovému 183 a mohli bychom najít i šestnáctkový ekvivalent. Ale toto je příliš zdlouhavý proces. Budeme často potřebovat převádět z dvojkové do šestnáctkové, takže se naučíme, jak jedním jednoduchým krokem provedeme převod přímo.

Nejdříve vezmeme dvojkové číslo 10110111 a rozdělíme je na dvě části, přesně napůl. Jestliže 8 bitů se nazývá byte, potom každá sada 4 bitů by se měla nazývat… NIBBLE. A nazývá se! Vyšší nibble je 1011 a nižší je 0111. Každý z těchto

niblů můžeme jednoduše převést v jednoduchou šestnáctkovou číslici, protože 4 bity představují číslo od 0 do 15.

```
1011 = 1 osmička  = 8       0111 = 0 osmiček  = 0
       0 čtyřek   = 0              1 čtyřka   = 4
       1 dvojka   = 2              1 dvojka   = 2
       1 jednička = 1              1 jednička = 1
           Celkem = 11 (B)             Celkem = 7
```

Takže šestnáctkový ekvivalent 10110111 je $B7, který jsme získali bez převádění pomoci desítkové soustavy. Můžete si vyzkoušet, že čísla jsou shodná bez ohledu na způsob převodu.

Při převodu z šestnáctkové do dvojkové jenom převrátíme výše uvedený proces. Zde je uveden příklad převodu šestnáctkového čísla $FC do dvojkové soustavy.

```
F = 1 osmička        C = 1 osmička
    1 čtyřka             1 čtyřka
    1 dvojka             0 dvojek
    1 jednička           1 jednička
    1111                 1101

            11111101
```

Organizace dat

Teď, když už snadno umíte převádět čísla z jedné soustavy do druhé, pojďme si popovídat o tom, jak jsou ve vašem počítači uspořádána data. Jak jste si už asi všimli, ve všech výše uvedených příkladech byla dvojková čísla seřazena do skupin o osmi číslicích, v počítačové hantýrce 8 bitů tvoří byte. Každá paměťová lokace ve vašem ATARI uskladňuje 1 byte informací. Mělo by být zřejmé, že v největším možném bytu jsou všechny bity rovny 1. Můžete si spočítat, že největší jednoduchý byte, který může uskladnit jakýkoliv počítač, je 255 v desítkové soustavě. Stejně zjistíme, že je pouze 256 možných různých bytů (nezapomeň na 0). Tak jak počítač zpracovává větší čísla a jak zpracovává více než 256 různých čísel?

Počítače mohou zpracovávat větší čísla dvěma různými způsoby. Jedním je párovat několik bytů dohromady pro vyjádření jednoduchého čísla. Použijeme-li tuto techniku, vidíme, že dvoubytové číslo může byt velké až 256*256 neboli 65536.

Ačkoliv trojbytové čísla obvykle ve vašich ATARI nejsou používána, tento systém dovoluje čísla velké až 256*256*256, tedy 16777216. Jak vidíte, tato technika vám dovoluje uchování velmi velkých čísel. Druhou metodou pamatování si velkých čísel je využití čísel s pohyblivou řádovou čárkou. Dosud používaná čísla byla čísla celá. Žádné zlomky nebo desetiny se nemohou v celých číslech objevit. Avšak u čísel s pohyblivou řádovou čárkou žádná taková omezení nejsou. Čísla jako 1,237 nebo 153,2 jsou zcela právoplatná čísla s pohyblivou řádovou čárkou, avšak nejsou to platná celá čísla. Termín „pohyblivá řádová čárka“ vychází z představy, že řádová čárka se může pohybovat z místa na místo, jak je tomu ve dvou výše popsaných číslech. V obou případech byly 4 číslice v číslech, ale v prvním případě byly 3 napravo od desetinné čárky a v druhém případě pouze 1. Jak znázorňujeme taková čísla v počítači?

Všeobecně jsou čísla kódována tak, že 1 byte představuje mocninu deseti, kterou je číslo násobeno. 1 Byte představuje znaménko této mocniny (jestliže je číslo větší než jedna nebo v intervalu 0,1), a několik bytů představuje mantisu nebo číslo samo. Jinými slovy bychom mohli zakódovat 153,2 jako následující posloupnost bytů:

1,2,1,5,3,2

V použitém kódovacím schématu první číslo představuje znaménko exponentu: 1, je-li kladné, a 0, je-li záporné. Druhé číslo představuje mocninu deseti, kterým je násobena mantisa, neboli 100. Zbývající číslice představují samotné číslo, s desetinnou čárkou za první číslicí. Takže dekódujeme-li toto číslo dle uvedeného pravidla, dostaneme:

100 * 1,532 = 153,2

Samozřejmě jsou možné i jiné kódovací systémy. Důležité je uvědomit si, že kódováním čísel můžeme v počítači znázornit i velmi velká čísla, dokonce bez použití bytů větších než 255.

Dosud jsme se zabývali jenom kladnými čísly. Jak budeme manipulovat se zápornými? Použitím znaménkové dvojkové aritmetiky. V tomto systému se bit umístěný nejvíce vlevo nazývá „nejvýznamnější bit“ a vůbec nepředstavuje mocninu 2, ale představuje znaménko čísla. To znamená, že jestliže nejvýznamnější bit je 1, číslo je záporné, a v případě 0 je číslo kladné. Průvodním jevem tohoto systému je, že největší ozna-

čené číslo, které můžeme znázornit v 1 bytu, je +127 nebo −128, protože máme jenom 7 početních bitů, s kterými můžeme pracovat.

```
Binární čísla bez znaménka
1 0 1 1 0 1 0 1      0 0 1 1 0 1 0 1
      181                  53

Znaménkové binární čísla
1 0 1 1 0 1 0 1      0 0 1 1 0 1 0 1
      −53                 +53
```

Na tomto místě by bylo třeba upozornit na jednu věc. Jestliže použijete dvoubytovou znaménkovou aritmetiku (a ATARI ji používá často), pouze nejvýznamnější bit nejvýznamnějšího bytu (byte představující nejvyšší číslice čísla) je znaménkový bit. To znamená, že dvoubytové číslo se znaménkem obsahuje 15 numerických bitů a jenom 1 znaménkový bit. Tato skutečnost bude pro nás důležité, až se dostaneme k dvoubytové matematice. Samozřejmě používáme-li pohyblivou desetinnou čárku, potřebujeme k našemu kódu přidat 1 byte, který bude představovat znaménko čísla – tzn., zda je konečné číslo kladné nebo záporné. Všechna použitá kódovací schémata to berou do úvahy.

Techniky adresování paměti

V BASICu je odvolávaní se na specifickou paměťovou lokaci dosti jednoduché a přímočaré. Např.

```
POKE 752,1
```

je přímý příkaz k umístění „hodnoty 1 do paměťové lokace 752. Navíc víme, že maximální náhodné přístupná paměť užitelná ve standardním počítači ATARI je 48k RAM. U operačního systému 10k ROM a s jinými prostory používanými pro jiné speciální účely je maximální konečná paměť přístupná v normálním ATARI 64k nebo 65536 paměťových lokací. Toto číslo by pro vás nemělo být ničím novým, protože jsme se s ním setkali již dříve. Je to největší číslo, které může být zakódováno 2 byty.

ATARI adresuje paměť používáním dvoubytového systému, který umožňuje adresovat 65536 různých paměťových lokací. Každý počítač, který je založený na čipu 6502, má stejné ve-

stavěné omezení na počet adresovatelných paměťových lokací. Ale jak mohou některé ATARI obsáhnout více než toto množství paměti? Jak se mohou některé počítače 6502 chlubit více než 64k celkové paměti?

Tajemstvím, jak zvýšit adresování paměti, je využití techniky zvané přidělování paměti z banku. Pří používání této procedury je využíván další byte, který kontroluje, na který bank paměti bude brát dvoubytový vektor zřetel. Představte si počítač s 16 banky nebo řadami, z nichž každá obsahuje 64k celková paměti. V daném okamžiku lze použít pouze 1 z těchto banků, ale střídavě lze použít všechny. Jestliže byte s funkcí výběru banku je rovný 0, je vybrán první bank, který obsahuje normálních 48k RAMu a operační systém ATARI. Za těchto podmínek jestliže vektor řekne, že chce informace z lokace 752, dostane informace z normální lokace 752, stejně jako nepozměněný ATARI. Jestliže je však byte s funkcí výběru banku rovný 1, je vybrán první 64k bank RAM-ky. Další PEEK lokace 752 by nyní vybral lokaci v tomto banku paměti, která by pravděpodobně obsahovala informaci zcela odlišnou, než jak tomu bylo v předchozím příkladě.

Jak si můžete všimnout, některé aspekty výběru paměti a banku jsou při používání mimořádně choulostivé. Představte si např. běžící program v BASICu, uložený v obvyklých 48k. Uprostřed programu se odvoláme na nový bank paměti. Počítač náhle vypadne z programu, protože program už není v adresovatelné části počítače, a to aspoň tak dlouho, dokud znovu nevybereme příslušný bank. To by mohlo mít za následek zhrouceni systému, a asi bychom ztratili obojí – náš program i informaci, kterou jsme se chtěli dovědět. Tyto problémy mohou být do jisté míry překonány novými výrobky, které jsou již na trhu a které rozšíří vaše ATARI za obvyklé maximum 48k RAM. Není žádný teoretický důvod, proč byste nemohli doma mít ATARI s maximální adresovatelnou pamětí přes 16 miliónů bytů (ano – miliónů!).

Opravdu nebuďte překvapeni, jestliže brzy uvidíte pro ATARI rozšířené systémy, které budou dávat možnost adresovat značně více než 192k, které jsou současně k dostání.

KAPITOLA TŘETÍ

ATARI hardware

Nyní se naučíme něco o „pracovním koni“ našeho počítače, o mikroprocesoru 6502. Veškeré počítání je v našem ATARI vykonáváno v čipu, který známe jako CPU (Central Processing Unit), někdy nazývaný i MPU (Micro Processing Unit). Různé počítače mají různá CPU – Z-80, 8080 a také 6502. Počítače ATARI spolu s jinými používají 6502 a modifikace 6502A nebo 6502B. Když hovoříme o programovaní ve strojovém kódu nebo v assembleru, máme opravdu na mysli přímé nebo nepřímé programování 6502-ky.

Mimo mikroprocesor 6502 existují v ATARI tři další specializované čipy, které v žádném jiném mikropočítači nenajdete. Nazývají se ANTIC, POKEY a GTIA (nebo dříve CTIA). Pracují spolu s 6502 a vytvářejí spektakulární grafiku a zvuky, na jaké jsme u počítačů ATARI uvyklí. To, že se vyskytují jen u ATARI vysvětluje, proč jeden a tentýž program vypadá a zní na ATARI mnohem lépe než na některém jiném mikropočítači. Použití každého z těchto čipů a způsobů jejich přístupu pomoci programu bude probráno v knize později.

Podnikneme krátkou průzkumnou cestu 6502 a naučíme se něco o tom, jak pracuje. První věcí, která nás při učení možná překvapí je, že nás výkonný počítač pouze umí porovnávat dvě čísla nebo je sečíst či odečíst! A co třeba odmocniny? Dělení a násobení? Všechna složitá matematika, se kterou v BASICu tak snadno pracujeme? Protože tyto funkce jsou ve skutečnosti jenom kombinacemi porovnávání, sčítání a odčítání, můžeme, jak později uvidíme, náš ATARI naučit, jak tuto složitou matematiku provádět. Mezitím se podívejme, jak náš počítač pracuje. Existuje šest částí každého členu rodiny 6502 a my probereme každý zvlášť a pak se podíváme, jak dohromady pracují.

Aukumulátor

První částí tohoto složitého čipu je akumulátor, v assemblerovském těsnopisu obvykle nazývaný A. Je tou částí, která vlastně vykonává počítání – porovnávání, sčítání nebo odčítání. Jedním způsobem, jak si akumulátor představit, je vykreslit si jej jako velké tiskací Y. Můžete čísla napěcho-

vat do každého z vrcholů ramen a pracovat s čísly tak (sčítat nebo odčítat), abyste dostali výsledek, který může být vytažen ze dna. Z této stránky je akumulátor jedinečný, protože je v počítači jediným místem, kde je možno najednou pracovat se dvěma informacemi. Budeme se chvíli zabývat jednoduchou obdobou v BASICu.

```
Jestliže provedeme příkaz
POKE 752,1

a ihned následuje
POKE 752,0
```

víme, že lokace 752 bude mít nyní hodnotu 0. To znamená, že nemůže mít současně obě hodnoty. Také víme, že k hodnotě uložená v lokaci paměti 752 nemůžeme přímo přidat 12. Kdybychom to chtěli v BASICu udělat, potřebovali bychom následující program:

```
10 I = PEEK(752)
20 I = I + 12
30 POKE 752,I
```

Takže jsme vyňali hodnotu uloženou v lokaci paměti 752, zvýšili ji o 12 a tuto novou hodnotu jsme uložili zpět do lokace paměti 752. Jak jsme zvýšili hodnotu o 12?

Ke zvýšení o 12 jsme na řádku 20 použili akumulátor. Přesné instrukce potřebné pro tuto manipulaci budou probrány později. V současné době bude dostačující, uvědomíme-li si, že jedinou částí našeho ATARI, která umí vykonávat matematické operace, je akumulátor. Kdykoliv potřebujeme vykonat jakoukoliv matematiku, musíme použít výše uvedený model příkladu v BASICu, tzn. že musíme přemístit hodnotu, kterou chceme změnit, do akumulátoru, změnit ji a uložit ji zpít tam, kde ji potřebujeme. To je, jak brzy uvidíme, ta základní operace v programování v assembleru.

Registry X, Y

Naše cesta 6502-kou pokračuje dalšími dvěma částmi čipu, registry X a Y. Tyto dvě ukládací lokace jsou na rozdíl od mnohých jiných paměťových lokací našeho ATARI ukryty přímo v 6502 CPU. Není k nim možný přístup přímo z BASICu, ale jsou často adresovány pomocí assembleru. Registry je možno používat dvěma způsoby. První je celkem jednoduchý a je totožný

a BASICovským příkazem POKE. To znamená, že můžeme tyto dva registry použít jako jednoduché ukládací lokace těch informací, o kterých víme, že je budeme zakrátko potřebovat. To je to zcela jednoduché použití těchto dvou výkonných registrů. Druhým užitím jsou čítače relativního umístění neboli indexové registry. Předpokládejme např., že chceme nejdříve přístup k lokaci 752, pak 753 a pak 754. V BASICu bychom to mohli udělat dvěma způsoby:

```
10 A(1) = PEEK(752)
20 A(2) = PEEK(753)
30 A(3) = FEEK(754)

nebo

10 FOR X=0 TO 2
20 A(X) = PEEK(752+X)
30 NEXT X
```

První způsob je obvykle označován jako „hrubě násilný“ přístup. Pracuje dobře tak dlouho, pokud počet položek, k nímž požadujeme přístup, je nízký. Jestliže ale náš problém bude vyžadovat přistup k 30 lokacím namísto ke 3, náš program vzroste na 30 řádků. V druhém případě dojde pouze ke změně dvojky na řádku 10. Je to všeobecnější a mnohostrannější řešení předloženého problému. Jak vidíte, používáme X jako rozdíl od lokace 752. V první části cyklu přistoupíme do lokace 752+0 neboli 752. V druhé části cyklu do lokace 752+1 neboli 753. Používáme hodnotu X jako rozdíl od základní adresy 752. Užijeme-li takto registry X a Y 6502, poskytnou nám rychlý a snadný způsob přístupu k informacím uloženým v za sebou jdoucích lokacích paměti. Protože takto může být nahromaděno mnoho informací – pole, řetězce, tabulky, obrazovky atp. – máme jeden velmi snadný způsob tvořeni informace a zjišťování toho, co jsme vytvořili.

Programový čítač

Další zastávkou na naší cestě bude setkání s programovým čítačem neboli PC. První věcí, které si na PC všimneme je, že je dvakrát větší než ostatní registry 6502, je široký 2 byty namísto obvyklého jednoho. Je to jediné místo v celém počítači, které může pracovat jako Šestnáctibitový (dvoubytový) registr. PC je zodpovědný za pamatování si, co bude ve vašem programu následovat. Např. jsme se všichni naučili, že v programu v BASICu je řádek 10 vykonán před řádkem 20, a ten zase

před řádkem 30, atd., a že před provedením je každý řádek v BASICu převeden do strojového kódu. Jak si počítač pamatuje, kde je a jaká instrukce ve strojovém kódu následuje? Napoví to PC. Programový čítač je šestnáctibitový registr, protože musí být schopen ukazovat na každou paměťovou lokaci v počítači. Jak jsme se učili v kapitole druhé, 16 bitů je třeba pro adresování 65536 lokací paměti, takže PC musí být široký 16 bitů. Zapamatujte si, že PC vždy ukazuje, která další informace má byt provedena.

Ukazatel zásobníku

Což kdybychom před delší cestou 6502-kou trochu posvačili? Hle, jídelna – zastavme se v ni na jeden rychlý bit (nebo byte?). Nejdříve si vezměme podnos a stříbrné příbory. Nechybí nám něco? Ó ano. Talíř. Tak hrábneme po jednom tam z té kupy. Všimněte si, že když jsme jej vzali z vrcholu hromady, tak se celá snížila o jeden talíř, takže další se ocitl na miste toho našeho.

Mohli bychom brát z vrcholu talíře, ale vždy by na vrcholu byl jiný. Hromada by byla o talíř nižší a všechny talíře by byly o pozici výše, ale talíř na vrcholu by byl vždy na stejné pozici, dokud bychom ovsem nevzali všechny talíře.

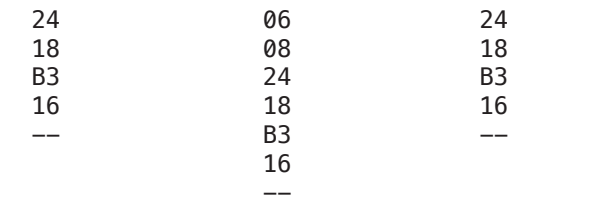
Teď, když jsme se najedli, vraťme se zpět k našemu putování. Dalším registrem na řadě je ukazatel zásobníku. Chová se stejně jako hromada talířů, kterou jsme před chvílí viděli v jídelně. Použijme opět příklad v BASICu. Prohlédněte si následující BASICovský program:

```
10 GOSUB 40
20 PRINT „GOODBYE“
30 END
40 PRINT „HELLO“
50 RETURN
```

Jestliže spustíme program, uvidíme nejdříve na obrazovce natištěné slovo HELLO a pod ním se objeví slovo GOODBYE. Potom program skončí. Jak to proběhlo?

Nejdříve se dostaneme na podprogram na řádku 40, čímž vytiskneme první slovo. Následuje RETURN na řádku 50, který zabezpečí vykonáni řádku 20. Jak ale počítač ví, že po návratu z podprogramu by měl být vykonaný řádek 20? Aha! Tady přichází na řadu pojem zásobník. BASIC používá zásobníky,

které jsou stejné jako ty, o nichž jsme již hovořili. Po vykonání příkazu GOSUB na řádku 10 BASIC vložil číslo řádku a jeho délku do tzv. pracovního zásobníku. Tento zásobník se liší od zásobníku 6502, protože ve všech počítačích základu 6502 je l. stránka paměti, tj. lokace paměti 256 - 511 včetně, používána jako zásobník. Oba tyto zásobníky pracují jako jídelna; jestliže vložíme další adresy do zásobníku, první adresa posune zásobník dolů tak, jak jsou dodávána další čísla (obr. 3-2).



Zásobník před odskokem do podprogramu | Zásobník v průběhu podprogramu | Zásobník po návratu z podprogramu

Ukazatel zásobníku je ta část 6502, která nese informaci o tom, co je uloženo na dně zásobníku, nemusíme se starat o počet hodnot, které jsou v zásobníku, nebo o to, jak do zásobníku dodat nové hodnoty. 6502 ovládá vše, co je pro nás tak vzdálené, stejně jako se BASIC bez nás stará o svůj vlastní zásobník. Jedinou věcí, o kterou musíme dbát je, že nesmíme do zásobníku napěchovat více než 256 čísel. Jestliže to uděláme, první čísla, která jsme tam nastrkali, ze dna vypadnou a ztratíme je. Takže když se je budeme pokoušet dostat zpět a použít, nebudou přístupná a ztroskotáme.

Teď už víme, jak dostat čísla do zásobníku. Ale jak je dostat zpět? V našem BASICovském příkladu zakončujeme podprogram na řádku 50 příkazem RETURN. Tento příkaz zabezpečí, že BASIC vytáhne z vrcholu zásobníku číslo a délku řádku a příkazem RETURN se na tento řádek vrátí. Takto se dostaneme zpět na řádek 20, kam jsme se chtěli z podprogramu dostat. Všimněte si, že jestliže v BASICu používáme podprogramy, nepotřebujeme o zásobnících vědět vůbec nic. Obdobná je situace i v assembleru, ačkoli, jak uvidíme, znalost práce zásobníku je důležitá z hlediska jeho dalších použití v assembleru.

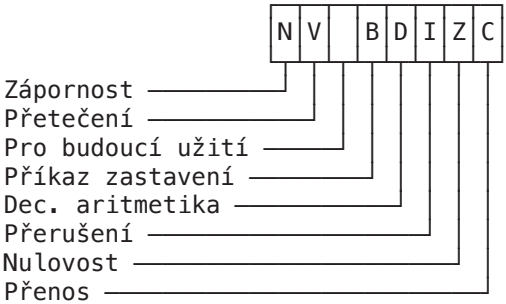
Stavový registr procesoru

Prohlídku 6502-kou doplníme návštěvou stavového registru procesoru, který je vlastně jednobytovou sbírkou různých vlajek, které používá 6502 pro jisté podmínky. Pro ty z vás, kteří se ještě nesetkali s pojmem vlajka: je to proměnná, jejíž hodnota indikuje jisté podmínky. Použijme BASICovský příklad.

```
10 I = 0
20 IF FILE = 33 THEN I = 1
30…
```

V tomto příkladě bychom mohli zkontrolovat, zda FILE = 33, zjištěním hodnoty I. Jestliže I=0, pak víme, že FILE se nerovná 33; ale když I=1, pak víme, že FILE=33. V tomto příkladě je I vlajka, která nám poskytuje informaci o hodnotě FILE. Stejným způsobem nám 7 vlajek ve stavovém registru 6502 dává důležitou informaci o tom, co se děje během programu. Každá vlajka je jednoduchý bit v jednoduchém bytu registru (obr, 3-3). Vlajky jsou obvykle známy jako jednopísmenové zkratky:

Písmeno	Indikační registr	Význam
C	přenosu	nastal
Z	nulovosti	nulový
I	přerušení	nepovoleno
D	decimální aritmetiky	povolena
B	příkaz zastavení	
V	přetečení	nastalo
N	zápornosti	záporný



Indikační registr přenosu

Indikační registr přenosu C nám říká, zda předchozí operace nestavila indikační registr přenosu, tedy, zda je součet větší než 255. Jako jednoduchý příklad sečtěme 250+250. Všichni snadno spočítané (např. s malou pomocí našich ATARI), že výsledek je 500. Avšak toto představuje v programováni v assembleru menší problém. Protože víme, že 255 je největší jednobytové číslo, které můžeme mít, jak asi znázorníme výsledek tohoto jednoduchého příkladu? Nuže, výsledek lze chápat jako 500-255 s přenosem. A protože výsledek je větší než 255, přeneseme 1, a výsledek je 245 s přenosem 1. Ale jak vyjádříme rozdíl mezi výsledkem 245 a výsledkem 245 s přenosem? Protože nejdříve nastavíme bit přenosu ve stavovém registru procesoru na nulu a v akumulátoru sečteme 250 plus 250, skončíme v akumulátoru se zbytkem 245. Bit přenosu je nyní 1 namísto nuly a dovoluje nám počítat skutečný součet. Jestliže sečteme 240+5, výsledek v akumulátoru by měl opět být 245, ale bit přenosu zůstává nulový a umožňuje nám rozlišovat mezi těmito dvěma situacemi. Uvědomíme si, že se nikdy nedostaneme do situace, kdy přenesení bude muset být 2, jestliže každá dvě čísla, která sčítáme dohromady, budou menší než 256. Jednobitový indikační registr přenosu je tedy pro naše potřeby dostačující. Jak dále uvidíme, v assembleru je bit přenosu skutečně používán ve všech matematických operacích.

Indikační registr nulovosti

Indikační registr nulovosti nás zpravuje o tom, zda předchozí operace vedla k výsledku nula. Jestliže ano, Z je rovný 1. Jestliže tedy registr Z je na začátku roven nule a odčítáme 2 - 2, akumulátor bude obsahovat hodnotu nula a registr Z bude 1. Aby bylo možno určit, zde se něco rovná nule, musíme zvážit každý z mnoha možných způsobů a potom se podívat na registr Z. V dalších kapitolách uvidíme, jak je tento registr užitečný.

Indikační registr žádosti o přerušení

IRQ znamená žádost o přerušení. Jestliže jste přečetli jakýkoliv z článků pro pokročilejší techniky - třeba o počítačích ATARI, pravděpodobně znáte termín přerušení jako přerušení display listu nebo snímkové zatemnění. Dříve než dočtete tuto knihu, bude pro vás jednoduché přidat tyto

techniky do vašich vlastních programů. 6502 může být přerušena při obvyklých operacích, pouze je-li vlajka I rovna 0. Jestliže je rovna 1, potom není možné normální přerušení. Tato skutečnost bude důležitá při pozdějších diskusích o různých přerušeních, používaných v ATARI. Prozatím si jenom zapamatujte, že pro přerušení normálním způsobem musí být ve stavovém registru procesoru vlajka I rovna 0. Jestliže nastavíme vlajku I na 1, přerušení nebude možné. Nazýváme to maskování přerušení.

Indikační registr decimální aritmetiky

6502 má dva režimy, ve kterých může pracovat – dvojkový a decimální. Hodnota indikačního registru decimální aritmetiky, D, ve stavovém registru procesoru určuje, v jakém režimu je procesor. Jestliže je hodnota rovna 1, všechny operace budou v decimálním režimu, a jestliže je O, budou v dvojkovém. Všeobecně v assembleru užívá většina operací dvojkovou matematiku, ale máte možnost přehodit výhybku přepínáním tohoto registru.

Indikační registr zastavení

Indikační registr zastavení neboli B registr může být nastaven a vynulován jen samotnou 6502. Programátor B registr nemůže měnit. Používá se pro určení toho, zda přerušeni bylo způsobeno instrukci 6502 BRK, což je zkratka pro BReaK. Protože programátor jej nemůže nastavit. V normálním programu má registr B malou funkci a všeobecně je používán jen k determinování toku programu.

Indikační registr přetečení

Ačkoli každý byte obsahuje 8 bitů (probráno v kapitole druhé), ve znaménkové binární matematice je nejvýznamnější bit používán pro označení znaménka čísla; proto největší znaménkové číslo, které můžeme znázornit v 1 bytu, je 128. Toto je situace obdobná té, která vyžaduje indikační registr přenosu, o němž jsme před chvílí hovořili. Co se stane, jestliže se pokusíme sečíst 120 + 120? Odpověď by měla být +240, ale vyjádření tohoto čísla vyžaduje použití nejvýznamnějšího bitu, který ve znaménkové matematice znázorňuje znaménko, ne část čísla. Proto pracujeme-li ve znaménkové matematice, po-

třebujeme nějakým způsoben určit, zda došlo k přetečení do znaménkového bitu. Indikační registr přetečení, V, používáme k určení tohoto. Jestliže registr V je 1, došlo k přetečení do znaménkového bitu, a jestliže je 0, k přetečení nedošlo. Můžeme tedy tento registr listovat, abychom si byli jisti, že námi vytvořené číslo může být s jistotou interpretováno jako znaménkové binární číslo. Je důležité věnovat tomuto bitu při práci se znaménkovou matematikou pozornost a vzít do úvahy druh programu, abychom s takovýmito přetečeními správní jednali tak, aby správně představovala znaménková čísla bez ohledu na přetečení.

Indikační registr znaménka

Posledním registrem ve stavovém registru procesoru je indikační registr znaménka, N. Jestliže je rovný 1, předchozí operace vedla k zápornému výsledku, a jestliže N je 0, výsledek byl buď kladný nebo rovný 0. Zapamatujeme si, že pak můžeme určit, zda číslo je kladné nebo nula, testováním registru Z. Testování N, C a Z registrů představuje hlavní metody, vezmeme-li do úvahy větvení programu v assembleru, podobně jako struktura IF…THEN… v BASICovských programech.

Tímto končí naše krátká cesta 6502, srdcem našeho počítače ATARI. Teď, když známe strukturu hardwaru, můžeme se začít učit instrukce, nezbytné k jeho programování.

Systém organizace paměti

Za pomoci 6502 jsme již probrali jeden aspekt organizace paměti v počítačích; zásobník tedy zaujímá v paměti specifické místo. Paměť v počítači 6502 je rozdělena na stránky, z nichž každá je dlouhá 256 bytů. Pravděpodobně jste se již s termínem stránka setkali, zejména ve spojitosti s oblastí paměti pro vás, programátory, rezervované počítačem ATARI: stránka 6. Stránka 6 je oblast paměti od $600 do $6FF, nebo-li v decimální terminologii od 1536 do 1791, a firma ATARI uvádí, že žádný z jejich softwarových výrobků tento prostor nepoužije, takže je volný pro vaše použití. Ve skutečnosti to není tak docela pravda. Při využívání tohoto prostoru buďte tedy opatrní. Stránka 6 je umístěna nad stránkou 5; horní půlka stránky 5 je počítači ATARI využívána pro několik účelů. Existují určité podmínky, při kterých může dojít k přetečení této oblasti a toto přetečení bude uloženo na začátku stránky 6, přesně na vrcholu vámi pečlivě ochraňované

informace. Ponaučení: stránku 6 používejte opatrně a se zřetelem na možné léčky.

I jiné stránky paměti v počítačích s 6502 mají své specifická užití. Jako nejzajímavější bych jmenoval stránku 0, prvních 256 bytů paměti v počítači. Stránka nula má pro programátory v assembleru zvláštní význam, protože každý přístup k této stránce je rychlejší než přístup kamkoli jinam v počítači, a protože určité operace lze vykonat jen za pomoci oblastí stránky 0. Avšak celá věc má jeden háček. Asi budete očekávat, že tím, že oblast je tak důležitá, budete ji moci celou používat. Chyba lávky! Právě proto, že je tak důležitá, téměř celou stránku nula využívá ATARI. Ať máte BASICovou či Assembler/Editor kartridž, můžete pracovat jen se 6 byty této stránky. Ano, se šesti! Abychom zpřístupnili více než těchto 6 bytů, naučíme se teď několik triků a zároveň i rozumně využívat oblasti, které máme k dispozici.

Naučíme se také hodně o stránkách 2 až 5 ($200 až $5FF nebo desítkově 512 až 1535), které obsahují informace, které potřebuje operační systém. Stránky nad stránkou 6 jsou obyčejně vyhrazeny pro DOS. Paměť, kterou může programátor assembleru bezpečně použít bez riskování přepsání programů DUP-SYSem, obyčejně začíná na $3200 neboli desítkově na 12800. Každá kartridž, kterou je možno použít, obyčejně začíná na $A000 a pokračuje do $BFFF; potom následuje po adresu $FFFF paměť celá rezervovaná pro operační systém. Mnohé z těchto oblastí budou detailně probrány v pozdějších kapitolách, ale tento náčrt slouží jako úvod do systému organizace paměti ve vašem ATARI a v hrubých rysech udává plán toho, co se kde děje. Budeme-li pokračovat, dozvíme se i detaily.

DRUHÁ ČÁST

VÝUČBA ASSEMBLERU

KAPITOLA ČTVRTÁ

NÁZVOSLOVÍ A INSTRUKČNÍ MNOŽINA

Pár slov o číslech

Dříve, než probereme instrukční soubor 6502-ky, potřebujeme se stručně seznámit s některými značkami, které jsou používané ve všech assemblerech 6502-ky. To nám umožní správně psát čísla a zkratky a navzájem se dorozumět.

Kdykoli je v assemblerových instrukcích použito číslo, musí mu předcházet číselná značka #. Odvoláváme-li se např. na číslo 2, musíme napsat #2. Teprve potom může assembler rozlišovat číslo a adresu v počítači. Když číslu předchází značka #, assembler ví, že máte na mysli číslo. Když se číslo objeví samotné, je pochopeno jako adresa. Vezměme si např. následovný příklad:

```
přičti 2 k hodnotě SUM                              SUM + #2
přičti k hodnotě SUM obsah paměťové lokace 2        SUM + 2
```

Začínající programátoři v assembleru se dopouštějí velké chyby tím, že zaměňují čísla za adresy a adresy za čísla. Tak zcela zničí každý program. Jestliže nejste dostatečně sběhlí v programování v assembleru, celé dny se můžete dívat na výpis programu a chyby si nevšimnete.

Druhou dohodou, používanou při programování v assembleru, je číselná soustava. Kdykoli se objeví buď samotné číslo, nebo mu předchází značka #, assembler ví, že míníte decimální soustavu. Např.:

```
SUM + #11
```

Assembler tím tlumočí mínění, že decimální číslo 11 by mělo být přičteno k hodnotě SUM.

```
SUM + 11
```

Assembler tím tlumočí mínění, že obsah paměťové lokace 11 (v decimálním číselném systému) by měl byt přičten k hodnotě SUM.

Chceme-li pracovat s šestnáctkovou soustavou, číslu musí předcházet znak dolaru. Např.:

```
SUM + $11
```

Tato instrukce znamená přičtení hodnoty SUM k obsahu paměťové lokace $11 (což je lokace 17 v decimální soustavě). Odvoláváme-li se na šestnáctkové číslo, věci se stávají složitějšími. Nejdříve musíme assembleru sdělit, že číslo přichází, a pak mu říci, že číslo je v šestnáctkové soustavě. Náš příklad nyní vypadá následovně:

```
SUM + #$11
```

Instrukce znamená přičtení šestnáctkového čísla $11 (decimálně 17) k hodnotě SUM. Assembler si to nemůže Spatně vyložit. Samozřejmě to může být novici v programování v assembleru špatně napsáno rozličnými formami. Takže už včasné varování je určitou výstrahou. Takovéto chyby při psaní programů v assembleru jsou velmi běžné, a jestliže vaše první programy nebudou fungovat, měli byste nejdříve hledat tyto typy chyb, než se pustíte do složitého hledání logických chyb.

Třetí typ čísla, který většina assemblerů uznává, je používán jen zřídka, ale když bude potřebný, budete rádi, že je dostupný. Je to dvojkové soustava, které obvykle předchází znak procenta, %. Např.:

```
#%11010110
```

Při interpretaci tohoto čísla se nemohou vyskytnout žádné zmatky, protože znak % je označuje jako dvojkové číslo. A dále, decimální číslo 11 010 110 je příliš velké, aby bylo přímo adresováno počítači s 6502-kou.

Abychom si tedy vše shrnuli. Znak # identifikuje následující člen jako číslo, aby ho odlišil od adresy. Znak $ identifikuje následující člen jako šestnáctkový člen a následuje po znaku #, máme-li na mysli šestnáctkové číslo. Znak % identifikuje další člen jako dvojkový a následuje-li po znaku #, jde o dvojkové číslo. Nepředchází-li členu znak $ nebo %, máme na mysli desítkovou soustavu.

Instrukční soubor 6502-ky

Každá instrukce instrukčního souboru 6502-ky je detailně popsaná v příloze 1. Instrukce zde krátce probereme, abyste se blíže seznámili s jejich názvoslovím a s využíváním. Každá instrukce je třípísmenová zkratka plného názvu instrukce. Tato zkratka se nazývá MNEMONIKA, a jestliže se jí jednou naučíte, poměrně snadno si ji zapamatujete. Způsob, jakým tyto instrukce adresují paměť, probereme v kapitole páté.

V tomto oddíle probereme instrukce po skupinách a zaměříme se na jejich využití při programování.

Instrukce uložení

Do této skupiny zahrnujeme tři instrukce:

LDA Ulož hodnotu do akumulátoru,
LDX Ulož hodnotu do registru X,
LDY Ulož hodnotu do registru Y.

Tyto instrukce jsou v určitém směru podobné instrukci PEEK v BASICu. Pomocí instrukci PEEK se získává hodnota, uschovaná ve specifické paměťové lokaci. Každou z instrukcí LOAD můžeme použít i pro získáni hodnoty z paměti, jak ukazuje následující příklad:

```
LDA $0243
```

Tento příkaz odebírá dříve uschované hodnoty v paměťové lokaci a adresou $0243 a ukládá kopii této hodnoty do akumulátoru pro další manipulace. Všimněte si ve větě zejména slova kopie. Stejně jako příkaz PEEK v BASICu, tak instrukce LOAD při programování v assembleru nemění hodnoty uschované v lokaci, ze které se vybírání provádí. Lokace $0243 obsahuje stejnou hodnotu před i po vykonání instrukce LDA; avšak v akumulátoru obsažená hodnota se mění v závislosti na provedené instrukci. Mohli jsme se rozhodnout přemístit tuto hodnotu z lokace $0243 buď do registru X nebo Y; předchozí řádek by se pak četl LDX $0243, respektive LDY $0243.

Protože již víme, že v akumulátoru jsou provedeny úkony jako sčítaní a odčítání, jedno použití LDA je zřejmé. Samozřejmě existují mnohé další použití této instrukce. Další dvě instrukce LOAD, LDX a LDY, jsou používány pro uloženi jednoho z registrů se speciální hodnotou obvykle

dříve, než je tento použit pro jiné operace, jako třeba počítadlo. Mnohé příklady instrukce LOAD budou v knize úplné probrány.

Instrukce uschování

Jak jsme již podotkli, BASICovský příkaz PEEK a assemblerové instrukce LOAD jsou si v mnohém podobné. V assembleru máme rovněž příkazy analogické BASICovskému příkazu POKE, příkazy STORE. Protože existují tři příkazy LOAD, není nic překvapujícího na tom, že nalézáme i tři příkazy STORE.

```
STA Ulož hodnotu do akumulátoru
STX Ulož hodnotu do registru X
STY Ulož hodnotu do registru T.
```

Typický řádek kódu v assembleru, který používá tyto instrukce, může vypadat následovně:

```
STX $0243
```

Tato instrukce kopíruje jakoukoliv hodnotu, dříve uloženou v registru X, do paměťové lokace $0243. Je zřejmá analogie s BASICovským příkazem POKE. Podobně jako při příkazu LOAD, hodnota uložená buď v akumulátoru nebo v registrech, což závisí na použité instrukci, se provedením instrukce nemění. Proto chcete-li uložit číslo 8 do čtyř různých paměťových lokací, měl by být použit následující kód:

```
LDX #8
STX $CC
STX $CD
STX $12
STX $D5
```

Všimněte si zejména, že před každým příkazem uschování nemusíme do registru X znovu ukládat číslo 8. Hodnota v něm zůstává, dokud ji nezměníme. Samozřejmě bychom mohli stejně snadno použít jak akumulátor, tak registr Y pro dokončení předchozího cíle stejným způsobem.

Jedno společné využiti instrukcí LOAD a STORE je přesun hodnot uložených v jedné nebo v několika paměťových lokacích do odlišných lokací. Např.:

```
LDA $5982
```

```
STA $0243
LDA $4903
STA $82
```

V kapitole sedmé uvidíme, jak využít tento typ rutinní práce k napsání podprogramů, které úžasně urychlí vaše BASICovské programy.

Instrukce řízení

Dva typy instrukcí způsobují skok v řízení programu z jednoho místa v programu na jiné. Jsou to instrukce SKOKU a PODMÍNĚNÉHO SKOKU.

Instrukce skoku

pro účely této diskuse jsme dvě instrukce seskupili do této kategorie:

```
JMP Skok na zadanou adresu
JSR Skok do podprogramu
```

Tyto dvě instrukce jsou analogické BASICovským příkazům GOTO,respektive GOSUB. Obě instrukce mají za následek nepodmíněný přesun řízení programu. Zde je uveden příklad instrukce JMP:

```
      JMP SUB1  ; GOTO SUB1
SUB0  LDA #1
      STA 752
SUB1  LDA #0
      STA 752
```

V tomto případě bude kursor stále neviditelný, protože kdykoli se program dostane k instrukci SKOKU, bude jako další vykonán řádek s návěstím SUB1. Tento přesun řízení je NEPODMÍNĚNÝ; tzn., že bude pokaždé proveden. Dvouřádková rutina s nevěstím SUB0 nebude nikdy vykonána. Pro srovnání se podívejme ne příklad instrukce JSR:

```
      JSR SUB1  ; GOSUB SUB1
SUB0  LDA #1
      STA 752
      JMP SUB2
SUB1  LDA #0
```

```
        STA 752
        RTS
SUB2 ....
```

V této rutině skáčeme do podprogramu označeného SUB1. Program potom popořádku vykonává řádky, dokud nenarazí na instrukci RTS (ReTurn from Subroutine). Řízení programů se pak vrátí na řádek následující po instrukci JSR, která předala řízení programu podprogramu. Instrukce RTS je assemblerovským protějškem BASICovskému příkazu RETURN, který také určuje konec podprogramu. V našem příkladě bude vykonán nejdříve SUB1 a pak SUB0. Instrukce JMP, která následuje za SUB0, zabraňuje, aby byl SUB1 vykonán dvakrát. V assembleru existuje ještě další instrukce, která je podobná instrukci RTS, a to RTI (ReTurn from Interrupt). Tato instrukce je používána na konci rutiny přerušeni, aby řízení bylo vráceno do hlavního programu. Přerušení probereme detailněji v pozdějších kapitolách.

Instrukce podmíněného skoku (větvení)

Jako kontrast ke dvěma právě probraným instrukcím nepodmíněného přesunu řízení má 6502-ka množinu instrukcí podmíněného přesunu řízení. Tyto mohou byt přirovnány ke struktuře IF… THEN v BASICu.

```
IF X=5 THEN GOTO 710
```

Tento příkaz pošle program na řádek 710 pouze v případě, kdy je X rovno 5. Rovná-li se jiné hodnotě, řízení programu se posune na další řádek, jehož označení následuje řádek s výrazem IF. Jinými slovy jsme kódováním tohoto řádku počítači umožnili rozhodnout co dělat, v závislosti na podmínkách, které jsme ustanovili a nastavili jsme podmíněný přesun řízení. Instrukcemi podmíněného skoku instrukčního souboru 6502-ky jsou:

```
BCC Větvení, jestliže je v carry 0
BCS Větvení, jestliže je v carry 1
BEQ Větvení, jestliže je výsledek rovný 0
BMI Větvení, jestliže je výsledek záporný
BNE Větvení, jestliže je výsledek nenulový
BPL Větvení, jestliže je výsledek kladný
BVC Větvení, jestliže nedošlo k přetečení
BVS Větvení, jestliže došlo k přetečení
```

Každá z těchto instrukcí je závislá na hodnotě jednoho z indikačních registrů ve stavovém registru procesoru. To, zda bude uskutečněn podmíněný skok, závisí na hodnotě indikačního registru, kterou v tom okamžiku má. Jsou to tedy instrukce podmíněného přesunu řízení programu. Abychom lépe pochopili, jak tyto instrukce pracují, uvedeme jednoduchý příklad.

```
        LDA #0
        BCC SUB4
        LDA #1
SUB4    STA $0243
```

V této rutině je hodnota uložená do paměťové lokace $0243 závislá na stavu indikačního registru přenosu ve stavovém registru procesoru v době, kdy je instrukce podmíněného skoku vykonávána. Jestliže je indikační registr přenosu nastaven (rovná se 1), podmíněný skok není uskutečněn a do akumulátoru je před příkazem STA uložena hodnota 1. Je-li indikační registr přenosu vynulován (roven 0), podmíněný skok je uskutečněn, akumulátor se nemění a do paměťové lokace $0243 je uložena hodnota 0. Instrukce BCS je opakem instrukce BCC: podmíněný skok je vykonán, jestliže je nastaven indikační registr přenosu, a vykonán není, je-li indikační registr přenosu nulový.

Instrukce BEQ a BNE závisí na hodnotě indikačního registru nulovosti ve stavovém registru procesoru, a ne na hodnotě indikačního registru přenosu. Jestliže je indikační registr nulovosti nulový, není vykonán podmíněný skok BEQ, ale BNE. Je-li indikační registr nulovosti nastaven, je vykonán podmíněny skok BEQ, ale BNE již ne. V uvedeném příkladě nebude vykonán podmíněný skok:

```
LDA #0
BNE SUB4
```

ale kdybychom napsali následující, byli bychom vykonali podmíněny skok na SUB4:

```
LDA #1
BNE SUB4
```

Indikační registr přetečení je používán pro určení výsledků instrukcí BVC a BVS analogickým způsobem. Podobně stanovuje indikační registr znaménka výsledek instrukcí BMI a BPL. Jestliže výsledkem předchozích instrukcí je negativní

hodnota, potom podmíněný skok, založený na instrukci BMI, je vykonán. Jestliže výsledek je buď kladný nebo rovný nule, je vykonána instrukce BPL. Vhodné použití těchto osmi instrukcí, které jsou závislé na hodnotách čtyř indikačních registrů ve stavovém registru procesoru, nám může poskytovat velmi dobré řízení toku programu.

Instrukce stavového registru procesoru

Tyto instrukce přímo nastavují indikační registry ve stavovém registru procesoru:

CLC Vynuluj indikační registr přenosu
CLD Vynuluj indikační registr decimální aritmetiky
CLI Vynuluj indikační registr přerušení
CLV Vynuluj indikační registr přetečení
SEC Nastav indikační registr přenosu
SED Nastav indikační registr decimální aritmetiky
SEI Nastav indikační registr přerušení

Tyto instrukce vykonávají naznačené operace přímo s indikačními registry stavového registru procesoru a jejich operace jsou déle popsány v dodatku 1.

Aritmetické a logické instrukce

Do této skupiny instrukcí zahrneme všechny výpočtové instrukce 6502-ky.

ADC Sečti spolu s indikačním registrem přenosu
AND Logický součin
ASL Aritmetický posuv vlevo
BIT Testuje bity paměti s akumulátorem
EOR Exklusivní logicky součet
LSR Logický posuv vpravo
ORA Logický součet
ROL Rotace vlevo
ROR Rotace vpravo
SBC Odečti spolu s indikačním registrem přenosu

Všechny instrukce jsou natolik složité, že na jejich detailní vysvětlení se podívejte do dodatku 1; vystačíte-li s krátkým probráním, čtěte dál.

Instrukce ADC je základní instrukcí sčítání 6502-ky. Sčítá hodnotu uloženou v akumulátoru a bitem přenosu s číslem adresovaným samotnou instrukcí ADC. Přičtěme např. obsah paměťové lokace $0434 k obsahu paměťové lokace $0435 a výsledek uložíme do lokace $0436:

```
CLC
LDA $0434
ADC $0435
STA $0436
```

ADC budeme často používat i ve zbývající části této knihy. Jejím protějškem je instrukce odčítání, SBC. SBC odčítá adresovanou hodnotu od hodnoty uložené v akumulátoru, přičemž používá bit přenosu, je-li pro vykonání odčítání potřebné půjčit si jedničku z vyššího řádu. Pro odečtení stejných hodnot, které jsme dříve sčítali, bychom napsali:

```
SEC
LDA $0434
SBC $0435
STA $0436
```

Do této skupiny patří i čtyři instrukce posuvu: ASL, LSR, ROL a ROR. Všechny tyto instrukce posouvají bity čísla, i když různými způsoby. Dvě instrukce rotace používají bit přenosu a doslova rotují osmi bity adresovaného čísla přes 9 pozic (8 v samotném čísle a devátou tvoří bit přenosu). Názorně si to můžeme ukázat na příkladu:

```
ROR $0434
        obsah adresy $0434   obsah C
START        10110100           1
KONEC        11011010           0
```

Jak si můžete povšimnout, každý bit zarotuje jednu pozici doprava s tím, že nulový bit skončil v bitu přenosu a původní bit přenosu se přemístil do bitu 7 lokace $0434.

Instrukce ROL obrací rotaci namísto doprava doleva. Dvě instrukce posuvu ASL a LSR pracují téměř stejně, až na to, že ačkoli koncový bit přejde jako předtím do bitu přenosu, do čísla je vždy zarotována nula bez ohledu na to, co bylo v bitu přenosu.

Tyto čtyři instrukce posuvu se také užívají k násobení nebo děleni mocninami dvou, protože rotováním bitů doleva

číslo zdvojujeme a rotováním bitů doprava číslo efektivně dělíme dvěma. Avšak existují i zádrhely, které je třeba mít při používání těchto instrukcí na zřeteli. Budou popsány v PŘÍLOZE 1.

Tři logické instrukce AND, EOR a ORA jsou vlastně tři způsoby porovnávání dvou čísel bit vedle bitu. Uvažují binární tvary dvou porovnávaných čísel a produkují různé výsledky v závislosti na tom, zda obě čísla v každém bitu obsahují jedničku nebo nulu. Instrukce AND říká: „Jestliže jsou oba bity 1, výsledek bude rovněž 1. Jestliže ne, pak výsledkem bude nula.“ Instrukce EOR říká: „Jestliže právě jeden z bitů bude 1, pak výsledek bude také 1. Když budou oba bity 1 nebo budou oba 0, výsledkem bude nula.“ Instrukce ORA říká: „Jestliže jedno nebo obě čísla budou 1, výsledek bude tvořit také 1.“ Tyto tři logické instrukce lze použít mnoha způsoby. ORA je nejčastěji používána k nastaveni specifického bitu v čísle, EOR na vytváření doplňku čísla a AND k vynulování specifického bitu čísla. Jestliže si nejste jisti znalostí těchto tří logických operací, podívejte se na další detaily do PŘÍLOHY 1.

Poslední instrukcí této skupiny je BIT, která je instrukcí kontrolní. BIT nastavuje indikační registr znaménka stavového registru procesoru podle 7. bitu testovaného čísla a indikační registr přetečení podle 6.bitu testovaného čísla. BIT také v závislosti na logickém součinu testovaného čísla a hodnoty uložené v akumulátoru nastavuje indikační registr nulovosti. Tato instrukce testuje několik aspektů celého čísla najednou. Všimněte si, že číslo v akumulátoru je instrukci BIT nezměněno. Jestliže za instrukci BIT umístíme jednu z již probraných instrukcí větvení, můžeme způsobit příslušné větvení ve vykonávání programu.

Manipulační instrukce 6502-ky

Stejně jako dříve probrané instrukce LOAD a STORE, zahrnuji následující instrukce výměnu informací z jedné části počítače do jiné:

PHA Vlož obsah akumulátoru do zásobníku
PHP Vlož stavový registr procesoru do zásobníku
PLA Vyber ze zásobníku a vlož do akumulátoru
PLP Vyber za zásobníku a vlož do stavového reg. procesoru
TAX Přesuň obsah akumulátoru do registru X
TAY Přesun obsah akumulátoru do registru Y

TSX Přesuň obsah ukazatele zásobníku do registru X
TXA Přesuň obsah registru X do akumulátoru
TXS Přesuň obsah registru X do ukazatele zásobníku
TYA Přesuň obsah registru Y do akumulátoru

Tyto instrukce jsou používány pro výměnu informací mezi různými registry 6502-ky nebo pro uloženi informací v zásobníku pro pozdější použiti. Instrukce PHA a PLA, jak později uvidíme, se často používají k předávání informací mezi programem v BASICu a podprogramem ve strojovém kódu.

Manipulační instrukce 6502-ky

Instrukce v této skupině mohou zvýšit nebo snížit hodnotu obsaženou buď ve specifické paměťové lokaci nebo v jednom z registrů 6502-ky o 1.

DEC Dekrementuj paměťovou lokaci
DEX Dekrementuj obsah registru X
DEY Dekrementuj obsah registru Y
INC Inkrementuj paměťovou lokaci
INX Inkrementuj obsah registru X
INY Inkrementuj obsah registru Y

Význam těchto instrukcí je jasný, následuje příklad jejich použití:

```
LDA #3
STA $0243
INC $0243
INC $0243
DEC $0243
```

Všimněte si, že žádná instrukce inkrementace nebo dekrementace nepracuje s akumulátorem. Pro zvětšení nebo zmenšení čísla v akumulátoru musíme použít instrukci ADC nebo SBC. Proto pro jednoduchost inkrementování nebo dekrementování je lépe číslo uložit buď do registru X nebo Y, raději než do akumulátoru, a potom jednoduše použít vhodnou instrukci inkrementace nebo dekrementace.

Instrukce porovnávání

Tyto tři instrukce umožňují porovnávat dvě hodnoty. Ovlivňují různé indikační registry ve stavovém registru procesoru v závislosti na výstupu porovnávání:

```
CMP Porovnej obsah akumulátoru s obsahem paměťové lokace
CPX Porovnej obsah registru X a obsahem paměťové lokace
CPY Porovnej obsah registru Y a obsahem paměťové lokace
```

Způsob, jakým každá z nich ovlivňuje stavový registr procesoru, je detailně popsán v dodatku 1. Pro demonstraci využití instrukce porovnávaní je zde však uveden jednoduchý příklad:

```
LDA $0243
CMP $0244
BNE SUB6 ; jdi na SUB6, jestliže $0244 = $0243
LDA #1
```

Zbývající instrukce

Dvě poslední instrukce nelze zařadit do žádné skupiny. Jsou to instrukce BRK a NOP. Instrukce BRK se používá hlavně při odlaďování vašich programů, poté co je napíšete. Způsobí, že program, který se právě vykonává, se zastaví. Svým způsobem je podobná BASICovské instrukci STOP. Instrukce NOP nedělá nic, její hlavní funkcí je rezervovat v programu místo pro budoucí změny, které se možná vyskytnou. Rezervování místa může být nezbytné, protože při programování v assembleru je často určitá paměťová lokace obsazena instrukcí, které může být kritická, a instrukce NOP mohou být ve výpisu nahrazeny funkčními příkazy bez změny paměťových lokací instrukcí, které následují. Tímto ukončíme náš stručný úvod k instrukční množině 6502-ky. Jak jsme již dříve uvedli, detailní rozbor těchto instrukcí lze nalézt v dodatku 1. Vřele doporučujeme, aby si jej důkladně pročetli hlavně začátečníci. Jestliže instrukční soubor již znáte, tento krátký rozbor vám ho měl osvěžit v paměti a můžete pokročit dále.

KAPITOLA PÁTÁ

V této kapitole si blíže objasníme způsoby adresování, kterých jednotlivé instrukce instrukčního souboru používají. Co vlastně pojem adresování označuje?

Jen tak pro ilustraci si představte výpis strojového programu řekněme v decimálních číslech (třebaže dobře víme, že běžnější je hexadecimální).

169, 15, 133, 206, 173, 48, 2, 133, 208, 173, 49, 2,
133, 209, 160, 3, 169, 6, 145, 208, 162, 23, 200, …

Tímto způsobem by se dalo popsat několik stran, a člověku neznalému instrukčního souboru 6502-ky by to nic neřeklo. Ovšem ani vaše ATARI nepozná, zda se jedná o konstantu nebo o kód instrukce, a řekli-li bychom mu, aby začal program vykonávat od druhého čísla (15), bude ho tvrdošíjně považovat za instrukci a ne za konstantu, jak bylo původně zamýšleno. Dlužno ovšem dodat, že mikroprocesor zná svůj instrukční soubor velmi dobře a ví také, jaký způsob adresace tomu či onomu kódu instrukce odpovídá. A opět jsme u toho adresování.

Adresou budeme rozumět údaj, který nám řekne, kde v paměti se nachází nějaký jiný údaj v paměti. Protože paměť představuje 65536 bytů, stačí na adresování celého paměťového prostoru dva byty. Znamená to, že žádná instrukce nemůže být více než tříbytová - jeden byte pro kód instrukce a nanejvýš dva pro operand nebo adresu. Všimněte si slovíčka „nanejvýš“. Je zde proto. Se adresa nebo operand mohou zabírat i méně než dva byty - jeden, ale i žádný (pak je celá instrukce jednobytová).

Nyní si tedy uvedeme adresovací způsoby mikroprocesoru 6502 a blíže si je vysvětlíme na instrukci LDA.

(V dalším textu budou uvedeny i anglické názvy způsobů adresování. Např. v brožurce ing. J. Sklenáře - Mikroprocesor 6502 - jsou v seznamech instrukcí ať již abecedním nebo číselném uvedeny právě tyto anglické názvy. - Poznámka překladatele.)

1. IMMEDIATE - PŘÍMÝ
2. ABSOLUTE - ABSOLUTNÍ
3. ZERO PAGE - NULTÁ STRÍNKA
4. ACCUMULATOR - AKUMULÁTOR

5. ZERO PAGE INDEXED – NULTÁ STRÁNKA INDEXOVANÁ
6. ABSOLUTE INDEXED – ABSOLUTNÍ INDEXOVANÝ
7. INDEXED INDIRECT – INDEXOVANÝ NEPŘÍMÝ
8. INDIRECT INDEXED – NEPŘÍMÝ INDEXOVANÝ
9. RELATIVE – RELATIVNÍ
10. INDIRECT – NEPŘÍMÍ
11. IMPLIED – IMPLICITNÍ

Jednotlivé způsoby adresování nebudeme probírat ve výše uvedeném pořadí, ale nejprve si objasníme ty, kterých instrukce LDA nepoužívá.

Implicitní

Prostě řečeno, tento způsob adresování říká, že určení operandů je součástí instrukce. Jak tomu rozumět? Znamená to, že již sám název, popř. kód instrukce říká, co se bude dít a se kterým místem paměti. Instrukce tohoto typu jsou vždy jednobytové – nevyžadují žádné uvedení adresy nebo operandu. Nyní by úvodní věta tohoto odstavce měla být jasná.

Uveďme si příklady:
TNX – zvětší obsah registru X o 1
SEC – nastav hodnotu stavového registru přenosu na 1
DET – zmenši obsah registru Y o 1
a jiné.

Akumulátor

Instrukce takto adresované pracují – jaké překvapení – s akumulátorem. Všechny instrukce tohoto typu jsou jednobytové.

Uveďme si příklady:
ROR A – rotuj obsahem akumulátoru doprava
ASL A – aritmetický posuv obsahu A doleva
a jiné.

Relativní

Všechny instrukce tohoto typu jsou dvoubytové. První byte představuje kód instrukce, druhý pak hodnotu, která je při-

čtena k obsahu programového čítače. Výsledkem je nová adresa, odkud pokračuje tok programu.

Nejprve si uveďme příklady, které si pak vysvětlíme. Jak uvidíme, přichází do úvahy pouze instrukce větvení:

```
BCS +7
BPL LOOP
```

nebo také výpis pouze v kódech:

```
173, 240, 2, 15, 243
```

Nyní si představme, že instrukce BCS +7 je na adrese $600=1536 decimálně. V okamžiku, kdy se do programového čítače dostane tato adresa, mikroprocesor pozná, že má co do činění s instrukcí BCS a zvětší obsah programového čítače o dvě, tj. na $602 = 1538, protože ví, že instrukce BCS je dvoubytová a že tedy další instrukce bude na adrese 1538. Až teď dojde k vykonání instrukce, tj. k obsahu programového čítače se přičte hodnota druhého bytu (v našem případě na adrese 1537). Jeho hodnota se tedy změní,a program bude pokračovat od takto určeného místa. Jestliže je však číslo v druhém bytu instrukce větší než 127, je chápáno jako dvojkový doplněk do čísla 255, a dojde ke skoku dozadu. Podívejme se na kratičký výpis strojového programu. Přeložený do assembleru by vypadal takto:

```
LDA 752
BPL -12
```

Odkud se nabralo těch -12? Dostaneme je jako 255-243 - což je dvojkový doplněk čísla 243 do 255. Při výpočtu, odkud bude program pokračovat, opět nezapomeňte na změnu hodnoty programového čítače.

Přímý

Na vysvětlení dalších osmi způsobů adresování si již vezmeme na pomoc instrukci LDA. Stejně tak bychom mohli uvažovat instrukci STA, prostě kteroukoliv, která využívá všech osmi způsobů adresace.

Řekněme si nejprve, že instrukce adresované přímým způsobem jsou dvoubytové. První byte představuje kód instrukce, druhý pak jistou konstantu. Tím se dostáváme k významu a po-

užití tohoto adresovacího způsobu. Pracujeme (jádro činnosti je dáno kódem instrukce; LDA, STA aj.) s konstantou 0 až 255. Uveďme si příklad:

```
LDA #15
LDA #$AC
```

Výsledkem této instrukce je uložení konkrétní uvedené hodnoty ve druhém bytu do akumulátoru. Kód LDA je v tomto případě 169. Použijeme-li jiný způsob adresování pro tutéž instrukci, bude její kód jiný.

Absolutní

Instrukce tohoto typu adresování jsou tříbytové. Adresa operandu je uvedena ve druhém a třetím bytu, a to v uspořádání nižší-vyšší ve výsledném strojovém kódu, nicméně v assembleru se adresa zapisuje v přirozené formě vyšší-nižší. Znamená to, že v případě

```
LDA $A000 (ve strojovém kódu potom AD00A0)
```

se do akumulátoru uloží číslo, které se v okamžiku vykonávání instrukce nachází na adrese $A000, což je 160*256+0 (160=$A0). Číslo takto uložené do akumulátoru z dané paměťové lokace nezmizí, ale zůstává tam!

Nultá stránka

Jedná se o adresovací způsob analogický s absolutním s tím, že adresa operandu je jednobytové, celá instrukce je tedy dvoubytová. Je zřejmé, že na adresování instrukce musí stačit jeden byte, protože do úvahy přichází pouze paměťové lokace z nulté stránky, což je, jak známo, úvodních 256 bytů paměťového prostoru, tj. 0 až 255. V souvislosti s nultou stránkou je třeba se zmínit o dvou skutečnostech: přístup k paměťovým lokacím v nulté stránce je rychlejší než kterékoliv jiné buňce paměti; a uživatel má z nulté stránky k dispozici pouze šest lokací, a to 200 až 205. Všechny ostatní využívá OS, což je zřejmé, uvědomíme-li si první skutečnost.

Nultá stránka indexovaná

I v tomto případě vystačíme pouze se dvěma byty. Výsledné adresa operandu je však dána jak obsahem druhého bytu, tak obsahem registru X. Získáme ji tak, že sečteme obsah druhého bytu a obsahem registru (přičemž přenos do vyššího bytu, pokud nastane, zanedbáme). Uveďme si konkrétní příklad:

```
      LDX #0
LOOP  LDA 200,X
      INX
      STA 202,X
      CPX #3
      BNE LOOP
```

Nás bude především zajímat řádek označený návěstím LOOP. Co se tam děje? Do akumulátoru se ukládá číslo, ale odkud, z jaké adresy? Postupujme podle návodu: sečteme obsah druhého bytu, tj. 200, s obsahem registru X. Je zřejmé, že při prvním průchodu cyklem bude v X nula. Výsledné adresa: 200+0=200. O dva řádky dále to však po předchozím INX bude 202+1=203.

Absolutní indexovaný

Jak zjistíte, v dalších způsobech adresování už půjde vždy jen o to, jat vypočítat výslednou adresu operandu. V případě absolutního indexovaného způsobu dostaneme výslednou adresu operandu jako součet obsahu druhého a třetího bytu s registrem X nebo Y.

```
LDA 57344,X
```

Jestliže v okamžiku vykonávání instrukce bude v X číslo 158, bude výsledná adresa 57344 + 158 = 57502.

Indexovaný nepřímý

Na rozdíl od předcházejícího způsobu adresování, který zabíral tři byty, tento zabírá pouze dva. Výpočet výsledné adresy je trošičku složitější, o to zajímavěji se dá využít. Při výpočtu adresy operandu postupujte takto: obsah druhého bytu instrukce sečtěte s obsahem registru X (POZOR! Registr Y nepřichází do úvahy). Dostanete tak číslo, které představuje nižší byte adresy v nulté stránce, kde je umístěna konečná adresa operandu. Představme si, že v X máme číslo 30. Potom

```
LDA (58,X)
```

znamená, že konečná adresa operandu je dána obsahem buněk 88, 89, protože 30+58=88, což je onen zmíněný nižší byte.

Nepřímý indexovaný

Jeho zápis je LDA (88),Y.

Představme si, že v buňce 88 je číslo 64 a v buňce 89 pak číslo 156. Druhý byte instrukce udává adresu v nulté stránce, kde je umístěna další adresa, ze které po přičtení obsahu registru Y (X ne!) vznikne výsledná adresa. Máme-li v Y 40, víme, že druhý byte = 88, což je ona adresa v nulté stránce. Na adrese 88, 89 jsou čísla 64 a 156, které tvoří adresu 256*156+64 = 40000. Výsledná adresa se pak rovná 40000+40=40040.

Nepřímý

Tohoto adresovacího způsobu instrukce LDA nevyužívá. Byl uveden až na konci, protože jeho specifikum spočívá v tom, že s ním pracuje pouze jedna instrukce – JMP. Výslednou adresu spočítáme takto: druhý a třetí byte instrukce udává adresu (její nižší byte), kde je umístěna konečná adresa operandu. Příklad:

```
JMP ($0345)
```

Znali-li bychom obsah paměťových buněk $0345 a $0346, lehce bychom spočítali výslednou adresu. Použití tohoto adresovacího způsobu nám bude zřejmé, přečteme-li si něco o odskokových tabulkách.

Na závěr si ještě uveďme přehled adresovacích způsobů instrukce LDA:

```
LDA Absolutní
LDA Absolutní,X
LDA Absolutní,Y
LDA Přímý
LDA (Nepřímý,X)
LDA (Nepřímý),Y
LDA Nultá stránka
LDA Nultá stránka,X
```

KAPITOLA ŠESTÁ

ASSEMBLERY PRO ATARI

Když hovoříme o assemblerech, zpravidla máme na mysli softwarové výrobky, které nám dovolují psát a spouštět programy v assemblerech. Tyto výrobky sestávají ze tři částí:

editor – k vlastnímu napsání zdrojového kódu

assembler – k překladu zdrojového kódu do strojového kódu

ladič (debugger) – k nalezení chyb a jejich opravě.

V současné době existují na ATARI tyto druhy assemblerů:

1. Assembler/Editor Cartridge
2. ATARI Macro Assembler (AMAC), MEDIT (editor) a DDT (debuger)
3. MAC/65
4. SYNASSEMBLER
5. Macro Assembler/Text Editor (MAE)
6. Edit 6502

Z důvodů, že je nejčastěji vlastněn uživateli, ačkoliv není nejvýkonnější, budou všechny příklady v této knize napsány pomocí kartridže ATARI A/E. V této kapitole si vysvětlíme syntax této kartridže a dále vysvětlíme, jak a v čem se ostatních pět assemblerů liší.

Není účelem této publikace některý z výrobků ať už přímo či nepřímo vychvalovat. Následující popis odlišností je zde uveden jen proto, aby vám umožnil pracovat s příklady v knize, popřípadě si upravit rutiny pro vlastní assembler, který jste zakoupili.

Kartridž ATARI ASSEMBLER/EDITOR

Nejprve si proberme syntax. Kartridž A/E vyžaduje, aby každému řádku předcházelo číslo řádku jako je tomu u BASICu. Musí to být číslo celé od 0 do 65535. Za tímto číslem musí následovat aspoň jeden prázdný znak, tedy mezera. Jednotlivá pole v řádku mají tyto funkce:

ČÍSLO ŘÁDKU NÁVĚSTÍ OP. KÓD OPERAND KOMENTÁŘ

Uveďme si takový typický řádek assemblerovského programu:

```
10 LOOP LDA $0342; ulož vyšší byte
```

Proberme si jednotlivé části tohoto řádku. První pole, návěstí, může a nemusí být, tzn. lze jej vynechat. V případě, že ho použijeme, lze tento řádek adresovat pomocí něho, v tomto případě LOOP. Proto, jestliže použijeme instrukci větvení a za ní LOOP, assembler ví, kam má skočit. Návěstí se zpravidla používá tehdy, jestliže víme, že v další části programu se na tento řádek budeme odvolávat. Prvním znakem návěstí musí být písmeno A - Z, další znaky mohou být i číslice 0-9. Návěstí může sestávat minimálně z jednoho a maximálně z (106 - počet číslic v čísle řádku) znaků. Protože některé assemblery pro ATARI limitují počet znaků v návěstí, nebude v žádném z příkladů v této knize použito návěští o více než šesti znacích.

Operační kód je instrukce 6502-ky, kterou chceme aby počítač vykonal v daném místě programu. Ve výše uvedeném příkladě chceme, aby byla do akumulátoru uložena nějaká hodnota. Pro tuto operaci máme kód LDA. V případě, že použijeme návěstí, musí být mezi poslední číslicí čísla řádku a operačním kódem dvě mezery. Příklad:

```
10␣SST␣LDA␣#6 10␣␣LDA␣#6
```

Důvod je zřejmý: kdyby totiž byla mezi číslem řádku a operačním kódem pouze jedna mezera, assembler by to chápal jako návěstí.

Operand specifikuje adresu nebo číslo, se kterým bychom operaci rádi provedli. V příkladě bylo použito #6, což znamená, že se jedná o konstantu. Operand začíná aspoň s jednou mezerou za operačním kódem, avšak je dovoleno i více mezer.

Mnohé operace pracují s akumulátorem. Zápis těchto instrukcí je následovný:

```
130 ROR A; všimni si A
```

Komentář by měl popisovat operaci uskutečňovanou řádkem, na kterém se komentář nachází. Komentář by neměl popisovat to, že LDA #6 ukládá do akumulátoru číslo 6, ale to, co tím autor myslel, jaký byl účel nebo záměr této činnosti.

Komentář slouží k tomu, abyste, až se k programu vrátíte po půl roce, nestrávili půlden koukáním na výpis a divením se, jakých rafinovaných triků jste před půlrokem použili.

Komentáře lze psát dvěma způsoby. Buď za operandem uděláme mezeru, a všechno, co je za touto mezerou, bude chápáno jako komentář, nebo bude komentáři věnován celý řádek. Jak později uvidíte, zvyšuje se tak čitelnost a srozumitelnost programu. Vynecháme tedy za číslem řádku mezeru a napíšeme středník (;).

```
100 ; Celý tento řádek je komentář.
100   LDA $340B ; I toto je komentář.
```

V celé této knize bude komentářům, ať už napsaným prvním nebo druhým způsobem, vždy předcházet středník.

Nyní známe strukturu assemblerovského řádku a musíme si ještě vysvětlit pár dalších úmluv.

Pseudoinstrukce

Většina assemblerů dává programátorovi k dispozici celou sérii dalších instrukcí, které je assembler schopen interpretovat a které tak v podstatě rozšiřují instrukční soubor 6502-ky. Nazýváme je pseudoinstrukcemi, protože se používají tak jako instrukce, ale nejsou částí instrukčního souboru procesoru.

Jednou z nejdůležitějších je instrukce počátku programu. Jelikož assembler vytváří strojový kód, který se nalézá na určitém místě v paměti, musíme assembleru říci, kde to místo je. Pracujeme-li s kartridží, vypadá instrukce následovně:

```
10␣␣*=␣$0600; začátek
```

Všimněte si, že mezi číslem řádku a hvězdičkou jsou dvě mezery a že mezi znakem rovnítka a prvním znakem adresy je jedna mezera. Tento řádek assembleru říká, že chceme, aby náš program začínal na adrese 600 hexa, což je šestá stránka. Když assembler narazí na tuto instrukci, uloží do programového čítače hodnotu, jež následuje za pseudoinstrukcí. Samozřejmě, že se tato instrukce může v průběhu programu vyskytovat i vícekrát.

Další pseudoinstrukce:

BYTE rezervuje v pamětí aspoň jedno místo pro budoucí použíti. Operand může do tohoto místa uložit informaci. Např. instrukce

```
110 .BYTE 34
```

rezervuje jednu paměťovou lokaci na stávající pozici programového čítače a ukládá do ní číslo $22 (dec. 34). Pomocí jedné instrukce.BYTE lze rezervovat také celou sérii bytů, jak to ilustruje další příklad:

```
125 .BYTE „HELLO“,$9B
```

Tímto zápisem uložíme čísla $48, $45, $4C, $4C, $4F, $9B. Tato čísla jsou tzv. ATASCII (Atari ASCII) kódy pro písmena slova HELLO.

.DBYTE rezervuje pro každou hodnotu operandu dvě paměťová místa. Používá se tedy pro data, která jsou větší než 255 a vyžadují tedy dva byty. Nejprve se ukládá vyšší, pak nižší byte. Uveďme si příklad:

```
115 .DBYTE 300
```

ukládá dvě hexadecimální čísla v pořadí $01 a $2C, protože 300 je v hexa zápisu $012C.

.WORD funguje stejné jako.DBYTE s tím rozdílem, ze nižší byte se uloží dřív.

LABEL = se používá k přiřazení. Např. píšeme program, který často vyžaduje používání adresy $9F. Můžeme tuto hodnotu přiřadit proměnné asi takto:

```
112 FREKU=$9F
```

Protože návěstí v pseudoinstrukci LABEL = je opravdové návěstí, musí začínat přesně za mezerou mezi poslední cifrou řádku a prvním znakem návěstí. Nyní, kdykoliv potřebujeme tuto adresu, použijeme např.:

```
173 LDA FREKU
```

END říká assembleru, že ukončil překlad a má se zastavit právě zde. Je nasnadě, že tato instrukce by měla být poslední

instrukcí vašeho programu. Není ovšem, podobně jako v ATARI BASICu, povinná a assembler i bez ní pozná, že je program u konce.

Existuje ještě celá řada pseudoinstrukcí pro Modul, ale my jsme probrali jen ty nejdůležitější. Další informace naleznete v manuálu k tomuto modulu.

Matematika v poli operandu

Další naše poznámka o modulu ATARI A/E se bude týkat operací sčítání, odečítáni, násobeni a dělení v poli operandu. Chceme-li např. rozložit adresu LOOP do struktury vyšší - nižší byte, můžeme tak učinit takto:

```
135 LDA #LOOP & 255
140 STA DEST
145 LDA #LOOP / 256
150 STA DEST + 1
```

V další části se jen velmi stručně zmiňme o jiných druzích assemblerů, popřípadě makroassemblerů pro ATARI s tím, že se zaměříme především na odlišnosti od modulu ATARI A/E.

ATARI MAKROASSEMBLER

- umožňuje vytvoření „makra“, což je krátká sekvence, jež je programem často volána. Např. v instrukčním souboru 6502 není příkaz jako JMI - lze nahradit makrem.
- používaní oddělených souborů nazývaných SYSTEXT
- nevyžaduje číslování řádků
- konstanty v osmičkové soustavě lze použít s označením @
- vedle značek matematických operací (+, -, /, *) lze provádět i některé logické operace.

MAC/65

- provádí kontrolu syntaxe okamžitě po stlačení klávesy RETURN
- čísla řádků 0 - 65535
- zachovává prioritu matematických operací
- dovoluje číst (příkazem ENTER) i soubory vytvořené jiným assemblerem

- čísla řádků 0 – 63999
- automatické číslování řádků vyžaduje tabelizování zápisu řádku
- návěstí mohou být dlouhá až 32 znaků
- adresovací kód akumulátor nepoužívá A za operačním kódem
- dovoluje pouze sčítání a odečítání
- pseudoinstrukce jsou zcela odlišné:

```
.AS  ....  .BYTE  pouze pro ASCII literály
.BS  ....  .BYTE
.DA  ....  .WORD
.EN  ....  .END
.EQ  ....  =
.OR  ....  *=
```

- překládá 50 až 100× rychleji než Modul.

KAPITOLA SEDMÁ

RUTINY VE STROJOVÉM KÓDU PRO POUŽITÍ S ATARI BASICEM

Uložení programu ve strojovém kódu v paměti

Když začínáme psát podprogramy, musíme se rozhodnout, kde je chceme umístit. Existují dva typy programů ve strojovém kódu, přemístitelné a pevně umístěné. Pevně umístěné jsou ty programy, které ve svém průběhu používají specifické adresy, které se nemohou změnit. Např. předpokládejme, že náš program bude vypadat takto:

```
30         *= $600
45         LDA ADDRI
50         BNE NOZERO
60 NOZERO  RTS
70 ZERO    SBC 1
80         RTS
90 ADDRI   BYTE 4
```

V tomto krátkém příkladě jsme použili několik odkazů na pevně dané adresy. Nelze je změnit, aniž bychom nevyrobili dokonalý nepřehled. Lépe to lze vidět na výpisu po překladu assembleru do strojového kódu:

```
ADDR    ML    LN   LABEL   OP    OPRND
--------------------------------------
0000          30           *=    $600
0600 AD0C06   45           LDA   ADDRI
0603 D003     50           BNE   NOZERO
0605 60       60   NOZERO  RTS
0609 E901     70   ZERO    SBC   1
060B 60       80           RTS
060C 04       50   ADDRI   BYTE  4
```

Výpis je pěkně upraven do sloupců. V prvním sloupci jsou hexadecimální adresy, na kterých je strojová rutina uložena. V druhém sloupci jsou strojové kódy jako výsledek překladu. Např. instrukce RTS na řádku 80 má kód 60 a je na adrese $060B. Ve třetím sloupci vidíme čísla řádků programu v assembleru. Čtvrtý sloupec obsahuje všechna návěstí obsažená v programu, pátý operační kódy, šestý operandy. V tomto pří-

padě zde není sedmý sloupec, který by obsahoval případné komentáře.

Abychom se však vrátili k problému pevných adres. První adresou tohoto druhu je ADDRI na ř. 45. Podívejme se na strojový přepis tohoto řádku. Vidíme tři byty: AD, 0C, 06. AD je kód instrukce LDA adresované v absolutním módu. A protože víme, že v tomto módu je instrukce tříbytová, je jasné, proč assembler vyprodukoval tři byty. Další dva byty v pořadí udávají adresu, ze které je třeba „naložit“ akumulátor – $060C.

Nyní lze lehce pochopit, proč by snaha umístit tento program někde jinde v paměti a spustit ho vedla ke krachu.

Druhou pevnou adresou, na kterou se v tomto programu odvoláváme, je možno nalézt na řádku 55. Každá instrukce JMP má v operandu vždy pevnou adresu. Z toho lze vidět, že kdykoliv se vykonává řádek 55, program vždy skočí na adresu $0609 bez ohledu na to, kde v paměti se program nachází. Ovšem po přemístění programu by se na adrese $0609 sotva nacházelo něco, co by mělo nějakou návaznost na náš program, a tudíž by nutně muselo dojít k zhrouceni programu.

Podívejte se na řádek 50. Pamatujte si, že všechny instrukce větvení používají relativního adresování. Ve výpisu překladu pro tento řádek najdeme D0, 03. D0 je kód BNE (k větvení dojde, jestliže se v testovaném registru nachází nenulová hodnota), ale trojka jako adresa nevypadá. Taky není. Ta pouze říká, aby se program přesunul o tři byty dál vzhledem na stávající hodnotu v čítači. V době vykonávání řádku 50 je čítač nastaven na začátek dalšího řádku. V našem případě na kód 4C instrukce JMP na řádku 55. Zvětšením o tři byty bude ukazovat na 60, kód RTS na řádek 60. Protože tedy tato instrukce jednoduše říká „přičti tři byty“ a na „jdi na $060B“, lze ji umístit kdekoliv v paměti a program se nám vždy přesune tam, kam to potřebujeme.

PROSÍM POZOR! V přemístitelném programu lze použít pouze větvení, ale ne instrukci JMP a přesně určených adres.

Proč tolik řečí o přemístitelnosti programu? Z jednoho prostého důvodu: pokud program není přemístitelný, potřebujeme v počítači vyšetřit nějaké bezpečná místo v paměti, kam bychom program umístili. A to nemusí být vždy jednoduché. Jestliže je tedy program přemístitelný, můžeme ho uložit kamkoliv. Ale kam?

Trochu si objasněme, jak BASIC pracuje s řetězci. Pokud chcete v ATARI BASICu použít řetězec, musíte ho dimenzovat. Takto rezervujete pro řetězec v paměti místo. Jestliže z nějakého důvodu BASIC toto místo potřebuje, jednoduše řetězec přemísti, ale pak je BASIC odpovědný ze to, že si pamatuje, kde řetězec je, a že ho nějakým způsobem ochrání. Aha! Můžeme tedy náš strojový program uložit jako řetězec v BASICu a přistup k němu najít příkazem USR (ADR (náš řetězec $)).

Pro krátké rutiny je obvykle vymezena stránka 6. Pamatujte si, že stránka 6 je vždy volná pro programy v assembleru. Tedy, téměř vždy. Měli byste vědět, že za jistých okolností se může stát, že stránka 6 není tak bezpečná. Jak jsme si již zmínili ve třetí kapitole, postav adresovaný $580 - $5FF (horní polovina páté stránky) je počítačem ATARI využíván jako bafr (místo pro dočasné uchování informací). Při vkládáni informace z klávesnice může dojít k přetečení bafru do dna šesté stránky. Tím se přepíše vše, co bylo uloženo na adresách $600 - $6FF podle toho, o jak velké přetečení se jedná. Pro účely této publikace budeme předpokládat, že k přetečení nedojde a programy, které nelze zapsat v přemístitelném tvaru, budou začínat na adrese $600.

Další možností umístění nepřemístitelných programů je v horní části paměti nebo pod LOMEM. Oba prostory jsou celkem bezpečné, jestliže je jejich použití věnována jistá pozornost. Navíc, pokud jste si jisti, že vaše aplikace nebude nikdy využívat magnetofonu, můžete použít jeho bafr ($430 - $4FF). Velmi krátké rutiny lze umístit na spodní konec zásobníku (asi $100 - $160), protože lze oprávněně předpokládat, že takto hluboko se zásobníku nebude používat. Ovšem to riziko tu je.

Na závěr jednu poznámku o organizaci této knihy. Všechny zde uvedené programy předpokládají přítomnost diskety a DOSu. Jestliže používáte magnetofon, musíte si prostudovat manuál k vašemu assembleru, abyste věděli, jak uskutečnit některé operace. Např. chceme-li natáhnout soubor ve strojovém kódu z disku, stiskneme L, jak nám to umožňuje DOS. Tatáž operace může mít pro různé assemblery různé příkazy, používáme-li magnetofon (třeba BLOAD apod.).

Jednoduchý podprogram vymazání paměti

Výstavbu vlastní knihovny podprogramů začneme velmi jednoduchou rutinou. V BASICu často potřebujeme vymazat jistou oblast paměti (PMG aj.). Pamatujte si, že jestliže chcete uchovat nějaká data blízko topu paměti, musíte přemístit display list pod tuto oblast paměti, jinak nám rutina zničí obrázek na obrazovce. Tohoto přemístění lze velmi jednoduše dosáhnout v BASICu:

```
10 ORIG=PEEK(106)
20 POKE 106,ORIG-8
30 GRAPHICS 0
40 POKE 106,ORIG
```

Samozřejmě lze vymazání paměti nulami dosáhnout i v BASICu. Příklad nám ukáže, jak vymazat osm stran paměti:

```
10 TOP=PEEK(106)
20 START=(TOP-8)*256
30 FOR I=START TO START+2048
40 POKE I,0
50 NEXT I
```

Tento program dělá přesně to, co chceme, ale trvá mu to 13,5 sekundy. To je dost dlouho, pokud jej chceme v průběhu aplikace několikrát opakovat. Dejme tedy šanci strojáku.

Nejprve si musíme rozmyslet, kam program umístíme. Stránka 6 je stejně dobré místo jako kterékoliv jiné. Dále víme, že paměťová lokace na adrese 106 nese informaci o tom, kde je v ATARI vrchol paměti ve stránkách. Konečně tento program je vhodný pro nepřímé adresování, takže budeme potřebovat dvě lokace v nulté stránce k zapamatování naší nepřímé adresy. Napišme si v assembleru, co jsme doposud vykonali:

```
100 ; .........
110 ; nastavení iniciačních podmínek
120 ; .........
130 *= 2600
140 TOP = 106
150 CURPAG = $CD
```

Všimněte si, že jsou pouze čtyři lokace v nulté stránce ($CC - $CF), kde máme jistotu, že ani BASIC ani Ass/Ed. Modul do nich nezasahují. Našli bychom jich snad i více, ale firma Atari se zaručila, že tyto jsou bezpečné.

Nyní přemýšlejme o tom, co chceme, aby rutina dělala. První instrukcí musí byt PLA, abychom ze zásobníku vybrali počet parametrů, jenž do zásobníku ukládá USR. Další je již naprosto zřejmé z následujícího výpisu.

```
160  ; ........
170  ; začni s výpočty
180  ; ........
190  PLA
200  LDA TOP
210  SEC
220  SBC #8
230  STA CURPAG
240  LDA #0
250  STA CURPAG-1
```

Nastavili jsme nepřímou adresu prvního místa v paměti, kterou budeme vymazávat, a uložili ji do CURPAG-1 (nižší byte) a CURPAG (vyšší byte). Do akumulátoru jsme uložili nulu a jsme tedy připraveni vkládat nuly do těch paměťových lokací, do kterých chceme. Dále potřebujeme čítač, který bude informovat o tom, kolik lokací jsme na dané straně vynulovali. Použijeme-li registr Y, může fungovat jednak jako čítač, jednak jako relativní rozdíl adresovacího režimu, kterého používáme. Zbývá tedy pouze nastavit, čítač, uložit nulu v akumulátoru do první lokace, dekrementovat Y o 1 a celý postup opakovat, dokud není v Y opět nula. Cyklus tak proběhne přesně 256× (což je počet bytů v jedné stránce), protože původně byl čítač nastaven na 0, dekrementaci o 1 v něm bude hodnota 255 atd.:

```
260  LDY #0
270  ; .........
280  ;
290  ; .........
300 LOOP STA (CURPAG-1),Y
310  DEY
320  BNE LOOP
```

Takto jsme tedy vynulovali jednu stránku. Jak zabezpečíme vynulováni dalších sedmi? Uvědomte si, že nepřímou adresu uchovanou v nulté stránce máme ve tvaru nižší / vyšší byte. Jestliže jednoduše zvýšíme vyšší byte o 1, bude nepřímá adresa ukazovat na další vyšší stránku:

```
330 INC CURPAG
```

Hle, jak prosté! Nyní už jen musíme zjistit, kdy skončit. Víme, že hotovi jsme tehdy, kdy stránka, kterou nulujeme, je vyšší než vrchol paměti /top/. Je celkem snadná určit, zda je tato podmínka splněna, např. takto:

```
340  LDA CURPAG
350  CMP TOP
360  BEQ LOOP
370  BCC LOOP
380  RTS
```

Jestliže se CURPAG = TOP, ještě musíme vynulovat poslední stránku. Skončení je indikováno podmínkou CURPAG > TOP!

Nyní po napsaní programu je zapotřebí přeložit ho z assembleru do strojového kódu. Učiníme tak vyťukáním ASM a RETURN. Spustíme tak překládací proces a po krátké pauze se na obrazovce objeví:

```
ADDR  ML   LN    LA8EL   OP   OPRND
----------------------------------
           0100 ; ........
           0110 ;
           0120 ; ........
0000       0130          *=  $600
006A       0140  TOP     =    106
00CD       0150  CURPAG  =   $CD
           0160 ; ........
           0170 ; začátek výpočtů
           0180 ; ........
0600 68    0190          PLA
0601 A56A  0200          LDA  TOP
0603 38    0210          SEC
0604 E908  0220          SBC  #8
0606 85CD  0230          STA  CURPAG
0608 A900  0240          LOA  #0
060A 85CC  0250          STA  CURPAG-1
060C AOOO  0260          LDY  #0
           0270 ; .........
           0280 ;
           0290 ; .........
060E 91CC  0300  LOOP    STA  (CURPAG-1),Y
0610 88    0310          DEY
0611 D0FB  0320          BNE  LOOP
0613 E6CD  0330          INC  CURPAG
0615 A5CD  0340          LDA  CURPAG
0617 C56A  0350          CMP  TOP
```

```
0619 F0F3  0360          BEQ  LOOP
061B 90F1  0370          BCC  LOOP
061D 60    0380          RTS
```

Nyní je program přeložen a uložen v paměti. Dalším úkolem je uchovat jej na disku, abychom mohli použít v našem BASICovském programu. Lze tak učinit dvěma způsoby. První je přímo z modulu pomocí příkazu:

```
SAVE D:PROGRAM 0600,061F
```

Na disku se vytvoří soubor zvaný PROGRAM. Všimněte si, že jsme uchovali několik bytů navíc – vyplatí se to. Druhý způsob, jak program uchovat, je jít do DOSu a stlačit K a napsat

```
PROGRAM, 0600, 061F
```

Obě z metod jsou stejně vhodné.

Program se nyní nachází na šesté stránce a máme k němu přístup, kdy se nám libí. Následující krok by však mělo být převedení ho do tvaru, kdy není zapotřebí DOSu k jeho nahrání. Lze jednoduše napsat jeden BASICovský řádek v přímém režimu

```
FOR I=1 TO 30:? PEEK(1531+I);“U“:NEXT I
```

avšak když už máme počítač, proč nenapsat program, který by vytáhl data z paměti a sestavil je do tvaru, ve kterém je bez problému převedeme do DATA příkazů? Tento program je níže:

```
10 FOR J=1 TO 30 STEP 10
20 FOR I=J TO J+9
30 PRINT PEEK(I+1535);“,“;
40 NEXT I
50 PRINT:PRINT
60 NEXT J
```

Po spuštění programu a menší úpravě máme na obrazovce:

```
104,165,106,56,233,8,133,205,169,0
133,204,160,0,145,205,136,208,251,230
205,165,205,197,106,240,243,144,241,96
```

Tyto řádky nyní spolu s naší rutinou začleníme do BASICovského programu:

```
10 FOR I=1 TO 30
20 REA0 A
30 POKE 1535+1,A
40 NEXT I
50 ORIG=PEEK(106)
60 POKE 106,ORIG-8
70 GRAPHICS 0
80 POKE 106,ORIG
90 POKE 20,0
100 X=USR(1536)
110 ? PEEK(20)/50
120 END
10000 DATA 104,165,106,56,233,8,133,205,169,0
10010 DATA 133,204,160,0,145,205,136,208,251,230
10020 DATA 205,165,205,197,106,240,243,144,241,96
```

Na řádku 90 jsme vynulovali vnitřní hodiny a na řádku 110 pak čteme hodnotu v tzv. „mžicích“, t.j. padesátinách sekundy. Vidíme, že výsledná hodnota je 0,0333 sekundy. Takže strojová rutina, jež se zdá tak dlouhá a časově náročná, je více než 4OOkrát rychlejší než BASICovský program, který dělal totéž. To úsilí za to stojí, ne?

Program, který jsme vytvořili, má ještě jednu vadu – je umístěný v šesté stránce, a proto ho nelze použít v programu, který tuto stránku potřebuje pro vlastní užití. Prohlédněme si ještě jednou assemblerovský výpis. Nepoužívali jsme ani skoky, ani jsme nedělali žádné odkazy na adresy v rámci programu vyjma instrukcí větvení. Program není pevně vázán na žádnou specifickou paměťovou lokaci, lze jej umístit kdekoliv v paměti a stále funguje. Využijeme tuto výhodu a převeďme jej do řetězce. Vyžaduje to přidáni řádku 5 a změny řádků 30 a 100:

```
5 DIM CLEAR$(30)
30 CLEAR$(I,I)=CHR$(A)
100 X=USR(ADR(CLEAR$))
```

Po spuštění programu si můžeme nechat CLEAR$ vypsat na obrazovce a pak psát jednořádkovou rutinu

```
20000 E=USR(ADR(„h./. j 8 i  M L M P f M./.MEjPS q
“)):RETURN
```

Podprogram přemístění znakové množiny

Jednou z velkých vymožeností počítačů ATARI je jednoduchost, se kterou lze modifikovat standardní znakovou množinu (normálně velká, malá písmena, inverzní znaky, číslice a symboly každodenního použití). Avšak jak víme standardní znaková množina je uložena v ROMce od adresy 57344 ($E000). Abychom ji mohli modifikovat, musíme ji celou přemístit do RAMky. Samozřejmě tak lze učinit v BASICu:

```
10 ORIGINAL=57344
20 ORIG=PEEK(106)
30 CHSET=(ORIG-4)*256
40 POKE 106,ORIG-8
50 GRAPHICS 0
60 FOR I=0 TO 1023
70 POKE CHSET+I,PEEK(ORIGINAL+I)
80 NEXT I
90 END
```

Tento program rezervuje blízko vrcholu paměti RAMky 8 stránek podobně jako v minulém příkladě, ovšem není nutné tuto plochu vynulovat, protože 4 stránky zaplníme znakovou množinou z ROMky. Vlastní přesun je uskutečněn na řádcích 60 až 80 (128 znaků po osmi bytech = 1024 bytů). Program tedy dělá to, co od něj vyžadujeme, a pokud nám nevadí, že trvá 14,7 sekund, není třeba ho převádět do assembleru.

V opačném případě použijeme techniky jako v minulém příkladě doplněné o nové dva rysy. První dovolí BASICu, aby předal našemu podprogramu lokaci v RAMce, od které bychom rádi umístili znakovou množinu. (Využijeme předávání parametrů, jak o tom pojednáváme v příloze 1.) Druhý rys spočívá v tom, že do každé lokace budeme ukládat jinou hodnotu. K tomu budeme potřebovat dvě nepřímé adresy ve stránce nula. Navíc budeme používat běžnější způsob označování dvojice nižší/vyšší byte: návěstím označíme nižší byte a vyšší pak bude návěstí +1 a ne naopak, kdy jsme návěstím označili vyšší byte a nižší byl návěstí -1. Nuže tedy, začněme:

```
        0100 ; ..........
        0110 ;
        0120 ; ..........
0000 0130     *= $600
00CC 0140 FROM = $CC
00CE 0150 TO   = $CE
```

Od tohoto okamžiku budeme jako ukázky uvádět výpisy assembleru. Pokud byste si chtěli ukázkový program sami naťukat, pak se vám v tomto tvaru objeví až po zasemblování. Takto uvedené ukázky nám umožňují odvolávat se jak na assembler, tak na strojový kód.

Při rezervování paměťových lokací v nulté stránce pro nepřímé adresy jsme vždy nadefinovali nižší byty, takže nepřímá adresa pro místo, odkud budeme brát znakovou množinu, bude uložena na adresách $CC a $CD. Podobně bude na adresách $CE a $CF nepřímá adresa místa, kam chceme množinu přemístit. Prozíravě jsme tyto lokace pojmenovali FROM a TO (z anglického OD a DO).

Nyní, když jsme si takto rezervovali místo pro nepřímé adresy, bude naším úkolem naplnit je správnými adresami. Pamatujte si, že adresu TO nám předá BASICovský příkaz USR, kdežto FROM je pevně daná $E000 operačním systémem. Uveďme si tuto část programu:

```
                0160 ; ..........
                0170 ;
                0180 ; ..........
0600 68         0190   PLA
0601 68         0200   PLA
0602 85CF       0210   STA TO+1
0604 68         0220   PLA
0605 85CE       0230   STA TO
0607 A900       0240   LDA #0
0609 85CC       0250   STA FROM
060B A9E0       0260   LDA #$E0
0600 85CD       0270   STA FROM+1
```

Rozeberme si řádky 190 až 230. Řádek 190 je náš starý známý – vytáhne nám ze zásobníku počet parametrů, který tam byl uložen BASICem. Všimněte si, že jak řádek 200, tak 220 jsou PLA. Takto se pracuje s parametry předávanými BASICem. Číslo, jež se má předat jako parametr, je BASICem upraveno do tvaru nižší/vyšší byte. Nižší byte se ukládá jako první a vyšší byte jako druhý. Proto první číslo, které ze zásobníku vytáhneme, bude vyšší byte (LIFO!) a uložíme ho do TO + 1. Potom přijde na řadu nižší byte a adresa TO. Nastavení adres FROM a FROM+1 je jednodušší, protože se jedná o konstantu $E000, jejíž vyšší byte je $E0 (do FROM+1), nižší $00 (do FROM).

A teď už nám jen zbývá napsat smyčku, která uskuteční vlastní přesun. Víme, že budeme přemísťovat 1024 bytů, což jsou čtyři stránky. Budeme tedy potřebovat čítač, který nám řekne, jak daleko jsme pokročili. Čítačem bude registr X. Další čítač, registr Y, budeme potřebovat k tomu, aby nesl informaci, jak daleko na té či oné stránce jsme. Prohlédněme si zbytek programu:

```
            0280 ; ..........
            0290 ;
            0300 ; ..........
060F A204   0310       LDX  #4
0611 A000   0320       LDY  #0
0613 B1CC   0330 LOOP  LDA  (FROM),Y
0613 91CE   0340       STA  (TO),Y
0617 88     0350       DEY
0618 D0F9   0360       BNE  LOOP
061A E6CD   0370       INC  FROM+1
061C E6CF   0380       INC  TO+1
061E CA     0390       DEX
061F D0F2   0400       BNE  LOOP
0621 60     0410       RTS
```

Mezi touto částí programu a odpovídající částí předcházejícího programu jsou jenom dva rozdíly. První spočívá v použití registru X k určení toho, kdy jsme hotovi. Na řádku 310 uložíme do registru X hodnotu udávající počet stran, které chceme přemístit. Na řádku 390 se dekrementuje X a na řádku 400 testujeme, zda jsme již dosáhli nuly a popřípadě se vracíme na začátek cyklu.

Druhý rozdíl je v tom, že se nechystáme ukládat do každé lokace vždy tutéž hodnotu, takže k uložení (nabrání) hodnoty do akumulátoru používáme tutéž techniku jako pro její uložení do příslušné lokace, tj. indexování nepřímé adresy v nulté stránce registrem Y. Všimněte si, že když se Y=1, vybíráme z druhé lokace znakové množiny vROMce (řádek 330) a ukládáme do druhé lokace v RAMce (řádek 340). Pamatujte si, že jak na řádku 370, tak na řádku 330 zvyšujeme obě nepřímé adresy o jednu stránku.

A nyní už tedy jen zbývá převést tento program do strojového podprogramu pro BASIC. Nejjednodušeji, jak to jde, se dopracujeme ke tvaru:

```
10 GOSUB 20000
20 ORIG=PEEK(106)
```

```
30 CHSET=(ORIG-4)*256
40 POKE 106,ORIG-8
50 GRAPHICS 0
60 POKE 20,0
70 X=USR(ADR(RELOCATE$),CHSET)
80 ? PEEK(20)/50
90 END
20000 DIM RELOCATE$(34)
20010 FOR I=1 TO 34
20020 READ A
20030 RELOCATE$(I,I)=CHR$(A)
20040 NEXT I
20050 RETURN
20060 DATA 104,104,133,207,104,133,206,169,0,133
20070 DATA 204,169,224,133,205,162,4,160,0,177
20080 DATA 204,145,206,136,208,249,230,205,230,207
20090 DATA 202,208,242,96
```

Rutina na řádcích 20000 až 20070 ukládá každý byte strojové rutiny na jeho příslušné místo v řetězci RELOCATE$. Rutinu vyvoláváme na řádku 70, kde jí předáváme parametr CHSET, což je adresa, od které bychom rádi umístili znakovou množinu v RAMce. Rutina je asi 500× rychlejší než sebelepší program v BASICu.

Podprogram přemístění libovolně oblasti paměti

Pomocí několika málo modifikací můžeme program, který jsme právě napsali, učinit mnohem lépe využitelným. Napíšeme ho tak, aby dovoloval přemístit libovolnou oblast paměti do jiné libovolné oblasti. Vidíme, že je třeba změnit pouze dvě části ve výpisu. Musíme totiž změnit absolutní adresu FROM, která byla nastavena pevně na 57344, a počet stránek, jenž byl nastaven na 4. Kdybychom na tomto místě mohli místo konstant použít proměnných, program by byl mnohem pružnější. Udělejme to tedy takto: předejme naší rutině adresu FROM a počet stránek, který chceme přemístit, jako parametry při volání rutiny z BASICu. Zde je výpis celého, již pozměněného programu:

```
            0100 ; ..........
            0110 ;
            0120 ; ..........
0000        0130      * = $600
00CC        0140   FROM = $CC
00CE        0150   TO   = $CE
            0160 ; ..........
```

```
           0170 ;
           0180 ; ..........
0600 68    0190      PLA
0601 68    0200      PLA
0602 85CD  0210      STA FROM+1
0604 68    0220      PLA
0605 85CC  0230      STA FROM
0607 68    0240      PLA
0608 85CF  0250      STA TO+1
060A 68    0260      PLA
060B 85CE  0270      STA TO
060D 68    0280      PLA
060E 68    0290      PLA
060F AA    0300      TAX
           0310 ; ..........
           0320 ;
           0330 ; ..........
0610 A000  0340      LDY #0
0612 B1CC  0350 LOOP LDA (FROM),Y
0614 91CE  0360      STA (TO),Y
0616 88    0370      DEY
0617 D0F9  0380      BNE LOOP
0619 E60D  0390      INC FR0M+1
061B E6CF  0400      INC T0+1
061D CA    0410      DEX
061E D0F2  0420      BNE LOOP
0620 60    0430      RTS
```

Rutinu jsme sestavili tak, že jsme ze zásobníku nejprve vybrali adresu FROM jako dvoubytovou hodnotu a potom týmž způsobem adresu TO a hodnotu počtu stránek k přemístění. Uvědomte si, že ATARI má pouze 256 stránek paměti, takže na poslední parametr nikdy nebudeme potřebovat více než jeden byte.

Až na tyto výjimky se tedy jedná o program totožný s dříve uvedeným programem. A vlastně i tuto modifikovanou verzi lze použít pro stejný účel. Pro úplnost ještě BASICovský program:

```
10 GOSUB 20000
20 ORIG=PEEK(106)
30 CHSET=(ORIG-4)*256
40 POKE 106,ORIG-8
50 GRAPHICS 0
60 X=USR(ADR(TRANSFER$),57344,CHSET,4)
70 END
20000 DIM TRANSFER$(33)
```

```
20010 FOR I=1 TO 33
20020 READ A
20030 TRANSFER$(I,I)=CHR$(A)
20040 NEXT I
20050 RETURN
20060 DATA 104,104,133,205,104,133,204,104,133,207
20070 DATA 104,133,206,104,104,170,160,0,177,204
20080 DATA 145,206,136,208,249,230,205,230,207,202
20090 DATA 208,242,96
```

Cvičení pro čtenáře

Použitím již uvedené techniky lze zaplnit daný počet stránek paměti jiným číslem než nula. Aby vám cvičení dalo co nejvíc, zkuste se nedívat dozadu a prostě sednout a tvořit.

Čtení joysticku

Všem je nám známo, jak složitý je program pro čtení joysticku v BASICu. Ačkoliv existují jisté způsoby, jak tento proces v BASICu urychlit, ve všeobecnosti se ke změně X-ové a Y-ové souřadnice, např. hráče, používá následující rutiny:

```
10000 IF STICK(0)=15 THEN 10050
10010 IF STICK(0)=10 OR STICK(0)=14 OR STICK(0)=6 THEN Y=Y-1
10020 IF STICK(0)=9 OR STICK(0)=13 OR STICK(0)=5 THEN Y=Y+1
10030 IF STICK(0)=10 OR STICK(0)=11 OR STICK(0)=9 THEN X=X-1
10040 IF STICK(0)=6 OR STICK(0)=7 OR STICK(0)=5 THEN X=X+1
10050 RETURN
```

Rutina zde byla zařazena pro svoji jednoduchost. Celkem bez problémů se dá vysledovat její logika. V každém případě by ani nejlepší programátor s touto rutinou, jakkoliv vylepšenou, soutěž v rychlosti nevyhrál. Podívejme se, jak dalece rychleji tato rutina pracuje v assembleru.

Pro účely našeho příkladu budeme předpokládat, že rutina, kterou napíšeme, bude jediný způsob, jak se hráč může pohybovat. Budeme pohybovat pouze jedním hráčem. Protože se hráč může pohybovat pouze ve dvou dimenzích, potřebujeme na zapamatování jeho polohy pouze dvě souřadnice. Pro uchování X-ové souřadnice potřebujeme jeden byte, ale dva byty pro nepřímou adresu Y-ové souřadnice. Rutina čtení joysticku je velmi přímočará:

```
0100 ; ..........
0110
0120 ; ..........
0130     * = %600
0140 YLOC  = $CC
0150 XLOC  = $CE
0160 STICK = $D300
0180 ; ..........
0190 ; nyní čti joystick 1
0200 ; ..........
0210      PLA
0220      LDA STICK
0230      AND #1
0240      BEQ UP
0250      LDA STICK
0260      ANO #2
0270      BEQ DOWN
0280 SIDE LDA STICK
0290      AND #4
0300      BEQ LEFT
0310      LDA STICK
0320      ANO #8
0330      BEQ RIGHT
0340      RTS
```

Jak můžete vidět, po již nudném PLA je čtení joysticku vlastně záležitost ukládání do akumulátoru obsahu hardwarové lokace STICK ($D300) a jeho demaskování čísly 1, 2, 4 a 8. ATARI joystick nastavuje jeden nebo více ze spodních čtyř bitů v lokaci $D000 na nulu dle následujícího předpisu: bit 0 pro pohyb nahoru, bit 1 pro dolů, bit 2 pro pohyb doleva a bit 3 doprava. Jestliže ani jeden těchto bitů není nastaven na nulu, je joystick ve výchozí poloze. Všimněte si, že není možné, aby byl joystick najednou v poloze nahoru a dolů, ale je možné, aby byl zároveň v poloze dolů a doleva nebo nahoru a doleva apod.

Uvedený program nebude pracovat, jak jste si pravděpodobné všimli, protože se v nem vyskytuji čtyři nedefinovaná návěstí UP, DOWN, LEFT a RIGHT (v překladu postupně NAHORU, DOLŮ, DOLEVA a DOPRAVA). Přidáme tedy rutiny, které v závislosti na směru pohybu joysticku ošetří pohyb hráče po obrazovce.

Nejdříve si všimněte, že každý z odkazů na návěstí pro daný směr joysticku používá instrukci BEQ. Efekt této instrukce je, že říká, zda výsledek logického součinu bitu s nanucenou jedničkou a hodnoty nalezené v lokaci STICK je

nula nebo jedna. Jestliže je to nula, joystick byl vychýlen do daného směru. Popřemýšlejte o tom. Víme, že jestliže má být výsledek nula, musí být v jednom nebo druhém čísle všechny bity nastaveny na nulu. V případě čísel 1, 2, 4 a 8 jsou všechny bity až na jeden nulové; takže v čísle z lokace STICK musí být právě příslušný bit roven nule, jestliže výsledek odmaskováni operaci AND je roven nule. Uveďme si konkrétní příklady vychýlení joysticku v jednom ze směrů a jejich odmaskování číslem 4:

```
Joystick   STICK   AND   76543210 (čísla bitů)
doprava     248     -    11110111
             -      4    00000100
              Výsledek = 00000100
```

Vidíme, že výsledek je nenulový. Protože jsme joystickem pohli doprava a testovali pohyb doleva, je výsledek správný. Jiný příklad:

```
Joystick   STICK   AND   76543210 (čísla bitů)
doleva      244     -    11111011
             -      4    00000100
              Výsledek = 00000000
```

Výsledek je roven nule a test tedy pracuje spolehlivě. Je třeba zdůraznit, že všechny tři polohy joysticku doleva (doleva nahoru, doleva, doleva dolů) by byly odmaskovány správně, protože druhý bit je v těchto případech nastaven na nulu. Za zmínku taky stojí, že horní čtyři bity zcela stejným způsobem odrážejí vychýlení joysticku připojeného do druhé zdířky.

Samozřejmě existují i jiné způsoby, jak napsat výše napsaný program. Třeba vás již napadlo použít ošetřující rutinu pro každý směr jako v následujícím výňatku:

```
0210    PLA
0220    LDA STICK
0230    AND #1
0240    BNE D1
0250    JSR UP
0260 D1 LDA STICK
0270    AND #2
0280    BNE D2
0290    JSR DOWN
```

Tento typ konstrukce by pracoval bez jakýchkoliv potíží s jedním ale: rutina je nepřemístitelná. Lokace UP, DOWN, LEFT a RIGHT musí být zahrnuty v programu, a jestliže se na ně odvoláme instrukcí JSR, znamená to, že jde o nepřemístitelný program. Jestliže o přemístitelnost programu stojíte, můžete psát rutiny s JSR. Protože však cílem této publikace je psát co nejvíc relokovatelných programů, budeme dále pracovat s rutinou bez JSR.

Jak se pohybuje hráčem v PMG (player-missile graphics)? Horizontální pohyb je jednoduchý - do příslušného registru horizontální polohy příslušného hráče uložíme jeho horizontální souřadnici. V případě prvního hráče, hráče 0, je registr umístěn na adrese $D000. Budeme ho nazývat HPOS0 a do programu, kterým se zabýváme, musíme přidat řádek

```
170 HPOS0 = $D000
```

jenž nám umožní se na něj odvolávat pomocí návěstí!

Ale co s pohybem vertikálním? Pohyb vertikální se uskutečňuje přesunem každého bytu na novou pozici, což je také důvod, proč jsme museli zavést nepřímou adresu pro Y-ovou souřadnici. Vystačíme s pouze jednou nepřímou adresou, protože hráčem pohybujeme pouze o jeden byte 7 jeho původní polohy. Budeme předpokládat, že hráč je vysoký osm bytů a že se jedná o prázdný čtverec. Chceme-li hráče přemístit na obrazovce o jeden byte výše, musíme začít přemisťovat shora dolů. Kdybychom začali přemisťovat zespodu, přepsali bychom si vyšší byte a tím ho ztratili. Názorně to bude vypadat takto:

```
před pohybem    po pohybu
..........      ..........
..........      .XXXXXXXX.
.XXXXXXXX.      .X......X.
.X......X.      .X......X.
.X......X.      .X......X.
.X......X.      .X......X.
.X......X.      .X......X.
.X......X.      .X......X.
.X......X.      .XXXXXXXX.
.XXXXXXXX.      ..........
..........      ..........
```

Analogicky při pohybu dolů začneme přemisťovat shora.

Nakonec je tu ještě jeden problém. Spodní hrana našeho hráče – čtverce by po pohybu byla dvojitá, protože ačkoliv jsme ji zkopírovali na správnou pozici o jeden byte výše, na její původní pozici jsme neumístili nic a ona tam zůstala. Pokud bychom toto neopravili, obrázek by vypadal takto:

```
XXXXXXXX
X......X
X......X
X......X
X......X
X......X
X......X
XXXXXXXX
XXXXXXXX
```

Ve skutečnosti, pokud tento nedostatek neodstraníme, bude hráč za sebou zanechávat ohon, což je efekt jistě zajímavý, ale není to to, co jsme chtěli! Naštěstí není těžké tento problém vyřešit: prostě budeme přemisťovat o jeden byte více, než kolika je hráč tvořen. Všimněte si, že jelikož je hráč tvořen osmi byty a my přemístíme devět s tím, že do buňky, kde byla spodní hrana, vložíme nulu. Je zřejmé, že tentýž problém nastane při pohybu hráčem dolů. Řešení je nasnadě! Nyní, kdy víte, jak pohybovat hráčem oběma směry, se podívejte na celý program a pak si ho popíšeme detailněji.

```
                0100 ; ..........
                0110 ;
                0120 ; ..........
0000            0130      * = $600
00CC            0140 YLOC   = $CC
00CE            0130 XLOC   = $CE
D300            0160 STICK  = $D300
D000            0170 HPOSP0 = $D000
                0180 ; ..........
                0190 ; čti joystick
                0200 ; ..........
0600 68         0210      PLA
0601 AD00D3     0220      LDA STICK
0604 2901       0230      AND #1
0606 F016       0240      BEQ UP
0608 A000D3     0250      LDA STICK
0608 2902       0260      AND #2
0600 F020       0270      BEQ DOWN
060F AD00D3     0280 SIDE LDA STICK
0612 2904       0290      AND *4
```

```
0614 F02E    0300      BEQ LEFT
0616 AD00D3  0310      LDA STICK
0619 2908    0320      AND *8
061B F02F    0330      BEQ RIGHT
0610 60      0340      RTS
             0350 ; ..........
             0360 ;
             0370 ;
             0380 ; ..........
061E A001    0390 UP   LOY #1
0620 C6CC    0400      DEC YLOC
0622 B1CC    0410 UP1  LDA (YLOC), Y
0624 88      0420      DEY
0625 91CC    0430      STA (YLOC), Y
0627 C8      0440      INY
0628 C8      0450      INY
0629 C00A    0460      CPY #10
062B 90F5    0470      BCC UP1
062D B0E0    0480      BCS SIDE
             0490 ; ..........
             0500 ;
             0510 ; ..........
062F A007    0520 DOWN LDY #7
0631 B1CC    0530 DOWN1
                       LDA (YLOC),Y
0633 C8      0540      INY
0634 91CC    0550      STA (YLOC),Y
0636 88      0560      DEY
0637 88      0570      DEY
0638 10F7    0580      BPL D0WN1
063A C8      0590      INY
063B A900    0600      LOA #0
063D 91CC    0610      STA (YLOC),Y-
063F E6CC    0620      INC YLOC
0641 18      0630      CLC
0642 90CB    0640      BCC SIDE
             0650 ; ..........
             0660 ;
             0670 ; ..........
0644 C6CE    0660 LEFT DEC XLOC
0646 A5CE    0690      LDA XLOC
0648 8D0000  0700      STA HPOSP0
064B 60      0710      RTS
             0720 ; ..........
             0730 ;
             0740 ; ..........
064C E6CE    0750 RIGHT
```

```
                        INC XLOC
064E A5CE    0760       LDA XLOC
0650 8D0000  0770       STA HPOSP0
0653 60      0760       RTS
```

Podívejme se na stavbu programu jako celku. Nejprve zjišťujeme, zda byl joystick vychýlen, nahoru. Jestliže ano, program pokračuje od návěstí UP. Jestliže ne, zjišťujeme, zda nastalo vychýleni dolů a popřípadě pokračujeme od DOWN. V obou případech se po pohybu hráče vrátíme zpět a zkontrolujeme pohyb do stran, protože je možné pohybovat hráčem vertikálně i horizontálně zároveň. Tato zpětná kontrola je zajištěna nuceným větvením programu na řádcích 480 a 640. Totiž na řádku 480 musí být přenosový bit (carry bit) nastaven, protože kdyby nebyl, řádek 470 by program poslal pryč od řádku 480.

Na řádku 640 je nucené větvení ještě zřejmější, protože o řádek dřív vynulujeme přenosový bit a pak klademe podmínku větvení BCC. Ptáte se, proč ne prostě skočit zpět na adresu SIDE? Protože chceme zachovat přemístitelnost programu. Tímto způsobem se tedy vyhneme skoku na absolutní adresu.

Jakmile jsme otestovali horizontální i vertikální pohyb, jsme hotovi a můžeme se vrátit zpět do BASICu. Všimněte si, že v programu jsou až tři instrukce RTS. Nikde není psáno, že rutina může mít pouze jedno RTS! Z našeho programu lze vyskočit, jestliže je joystick vertikálně (řádek 340) nebo jestliže jsme hráče přemístili doleva (řádek 710) nebo doprava (780), protože v kterémkoliv z těchto případů jsme vyčerpali všechny možnosti pohybu.

Krátká rutina ošetřující horizontální pohyb čte stávající X-ovou souřadnici z její uchovávací lokace, buď ji dekrementuje nebo inkrementuje a uloží novou hodnotu jak do HPOSP0, tak do uchovávací lokace.

Nyní si vysvětleme vertikální pohyb. Protože pohyb nahoru předpokládá dekrementování Y-ové souřadnice o jedna (čím nižší Y, tím výše na obrazovce hráč je), budeme vlastně dekrementovat YLOC. S výhodou tak můžeme učinit na začátku rutiny - po dekrementaci totiž YLOC ukazuje na buňku, kam patří horní hrana hráče - čtverce. Tím, že nastavíme Y (registr) na jedna, umožníme, aby příkaz s návěstím UP1 ukazoval na původní buňku horní hrany. Poté Y dekrementujeme a následující instrukce STA uloží byte na jeho novou pozici na obrazovce. Potom musíme Y dvakrát inkrementovat - jed-

nou kvůli předchozí dekrementaci, podruhé abychom se dostali k dalšímu bytu hráče, protože začínáme s Y = 1 a přemisťujeme devět bytů, budeme končit, když Y = 10. Pokud jsme skončili s přemísťováním, nuceným větvením se dostaneme ke kontrole horizontálního pohybu. Technika nepřímého adresováni nám bez zbytečných zmatků umožňuje vybírat z jedné a ukládat do druhé buňky.

Pro pohyb hráče dolů po obrazovce jsme použili nepatrně odlišný algoritmus. Jak již bylo řečeno dříve, začneme s nejspodnějším bytem, tzv. dnem. Nastavíme Y=7 (v tomto případě se jedná o byty 0 až 7). Další postup je zřejmý: postupné vybírání a ukládáni jednotlivých bytů za současného dekrementování, popřípadě inkrementování registru Y opakujeme tak dlouho, dokud hodnota v Y je větší nebo rovna nule. Jestliže tomu tak není, uložíme do původně nejnižšího bytu nulu. K tomu je zapotřebí inkrementovat registr Y, protože k tomuto bodu programu se vlastně dostaneme, až když bude hodnota v registru Y záporná, tj. $FF hexadecimálně. Poté inkrementujeme nepřímou adresu YLOC, protože jsme hráčem pohli o jednu pozici dolů na obrazovce. Nuceným větvením se vrátíme zpět zkontrolovat horizontální pohyb. Všimněte si, že při pohybu nahoru po obrazovce jsme vlastně přemísťovali devět bytů, ale při pohybu dolů pouze osm a potom jsme vymazali závoj vložením nuly do příslušné paměťové buňky. Použili jsme obě metody, abychom ukázali, jak každá z nich pracuje.

Mimochodem, mohli byste vznést námitku, že čtení joysticku probíhá ve čtyřech oddělených částech, a co se tedy stane, jestliže se poloha joysticku mezitím změní? Spočítejme si čas, za který se čteni joysticku vykoná. Vidíme, že celé čtení netrvá déle než 25 mikrosekund! Nemějte tedy žádné obavy, že byste mezitím stihli pákou pohnout!

A opět se dostáváme k závěrečné části našeho snažení, a to převedení strojové rutiny do BASICu:

```
10 TOP=PEEK(106)-8
20 POKE 106,TOP
30 GRAPHICS 0
40 PMBASE=TOP*256
50 POKE 54279,TOP
60 INITX=120
70 INITY=50
80 POKE 559,46
90 POKE 53277,3
100 GOSUB 20000
```

```
110 FOR I=PMBASE+512 TO PMBASE+640
120 POKE I,0
130 NEXT I
140 RESTORE 25000
150 Q=PMBASE+512+INITY
160 FOR I=Q TO Q+7
170 READ A
180 POKE I,A
190 NEXT I
200 POKE 53248,INITX
210 YHI=INT(Q/256)
220 YLO=(PMBASE+512+INITY)-YHI*256
230 POKE 204,YLO
240 POKE 205,YHI
250 POKE 206,INITX
260 POKE 704,68
270 Q=USR(ADR(JOYSTICK$))
280 GOTO 270
20000 DIM JOYSTICK$(87)
20010 FOR I=1 TO 87
20020 READ A
20030 JOYSTICK$(I,I)=CHR$(A)
20040 NEXT I:RETURN
20050 DATA 104,173,0,211,41,1,240,22,173,0
20060 DATA 211,41,2,240,32,173,0,211,41,4
20070 DATA 240,46,173,0,211,41,8,240,47,96
20080 DATA 160,1,198,204,177,204,136,145,204,200
20090 DATA 200,192,10,144,245,176,224,160,7,177
20100 DATA 204,200,145,204,136,136,16,247,200,169
20110 DATA 0,145,204,230,204,24,144,203,198,206
20120 DATA 165,206,141,0,208,96,230,206,165,206
20130 DATA 141,0,208,96,0,208,96
25000 DATA 255,129,129,129,129,129,129,255
```

Na řádku 100 je odskok, do podprogramu na řádku 20000, kde se celá strojová rutina uloží do řetězce JOYSTICK$. Voláni rutiny je na řádku 270. Všimněte si, že řádek 280 vrací řízení programu stále na řádek 270, tedy se stále opakuje čtení joysticku. Samozřejmě lze program rozšířit s tím, že po určitém čase je třeba se vždy vrátit na řádek 270. Všimněte si, jak je pohyb hráče plynulý. Zkuste vytvořit rutinu ošetřující pohyb hráče v BASICu. Vidíte, je vertikální pohyb kostrbatý a neohrabaný.

Vymožeností tohoto programu je, že vlastně všechny parametry PM grafiky (barva hráče, jeho tvar a velikost a další) jsou nastaveny v BASICu, takže rutinu čtení joysticku lze

připojit k téměř všem programům, které ji vyžadují. Pomocí jednoduchých modifikací, které si můžete sami vyzkoušet, bude rutina ošetřovat pohyb i jiného hráče než 0, čtení i druhého joysticku apod. Můžete si přidat i sticky po stlačení spouště (paměťová lokace $D010)! Nejlepší způsob, jak zvládnout programování v assembleru, je programovat. Což takhle tedy začít hned?

ČÁST TŘETÍ

APLIKACE

KAPITOLA OSMÁ

Čip ANTIC

Z jednoho velmi důležitého hlediska je váš počítač ATARI, pokud jej srovnáte s ostatními dostupnými počítači, unikátní. Většina mikropočítačů obsahuje jediný mikroprocesor, a to převážně 6502 nebo Z80. Vaše ATARI má však čtyři mikroprocesory, z nichž tři byly navrženy speciálně pro ATARI. V této kapitole se budeme zabývat jedním z nich, ANTICem.

Tento čip ANTIC obstarává zobrazování na obrazovce, což je velmi důležitý rys počítačů ATARI. Ve většině mikropočítačů je mikroprocesor zodpovědný nejen za výpočty a řízení toku programu, ale i za práci s obrazovkovým zobrazováním. Firma ATARI vyvinula ANTIC, aby byl mikroprocesor 6502 této starosti zbaven. Tím, že se ANTIC stará o obrazovku, může se 6502-ka starat o běžící program.

VIDEO RAM

V počítačích ATARI je v RAM-ce vymezena specifická oblast, která uchovává informace, které má program zobrazit na obrazovce. Tuto oblast budeme nazývat video RAM - VRAM. Tak jako většina specifických částí RAM-ky, má i VRAM svůj specifický vektor, který nám vždy říká, kde se VRAM nachází. Protože v normálních ATARI může být až 48 k RAM, potřebujeme na adresování VRAM-ky dva byty. Konkrétně se jedná o paměťové lokace 88, 89. Všeobecně platí, že kdykoliv použijeme příkaz GRAPHICS n, OS vymezí VRAM pro příslušný grafický režim hned pod vrcholem paměti. Protože se velikost požadované VRAM může pro jednotlivé grafické režimy značně lišit, je vždy velmi důležité vědět, kde v paměti VRAM začíná. K tomu nám slouží již zmíněná paměťové lokace 88, 89. Abychom v BASICu určili začátky VRAMky, stačí jednoduchý řádek:

```
10 BEGIN=PEEK(88)+256*PEEK(89)
```

Tímto příkazem převedeme dvojbytový vektor začátku VRAM-ky na jednoduchou adresu. Řekněme si, jak lze této informace využít.

Víme, že příkazem GRAPHICS 0 v BASICu vymažeme obrazovku. Ve skutečnosti se stane asi toto: OS se podívá do paměťové lokace 106, kterou jsme používali k určení vrcholu paměti RAM. OS potom určí, jak velkou VRAM je příslušnému grafickému režimu třeba vymezit a automaticky tuto oblast vynuluje. Nakonec je sestaven tzv. display list, o kterém si povíme krátce později, a řízení je předáno zpátky BASICu.

Jestliže tedy máme obrazovku v grafickém režimu 0, vytiskneme písmeno A do levého horního rohu obrazovky jednoduchým příkazem

```
PRINT „A“
```

Existuje však ještě jeden způsob, jak dosáhnout téhož záměru. Nyní, kdy víme, kde je v paměti umístěna VRAM, můžeme jednoduše příkazem POKE uložit do příslušné paměťové buňky správné číslo a písmeno se objeví na obrazovce.

```
POKE BEGIN+2,33
```

V příkazu je uvedeno + 2, abychom dodrželi předdefinovaný levý okraj (zkuste si POKE BEGIN,33). Písmeno A má v zobrazovacím kódu číslo 33, odtud tedy ta třicettrojka. Nezapomeňte, že váš počítač ATARI rozlišuje až tři základní kódy. Prvním je ATASCII (ATARI ASCII) kód, který používáme v BASICu. např.:

```
PRINT CHR$(65)
```

čímž opět vytiskneme písmeno A. Druhou množinou kódů je tzv. zobrazovací množina, ve které písmeno A koresponduje s kódem 33. Tohoto kódu používáme, chceme-li uložit informaci přímo do VRAM-ky. Třetí kódový systém se nazývá interní znaková množina. Používá se ke čtení kódu kláves klávesnice. Lokace 764 je jednobytový bafr, který obsahuje interní kód poslední stlačené klávesy. Pokud je v táto lokaci číslo 255, nebyla stlačena žádná klávesa. Programék ošetřující čekání na stlačenou klávesu by vypadal takto:

```
100 POKE 764,255
110 IF PEEK(764)=255 THEN 110
```

Jestliže chceme vědět, která klávesa byla naposledy stlačena, musíme se odvolat na interní znakovou množinu. Například, jestliže PEEK(764)=127, byla stlačena klávesa s velkým A (tj. SHIFT + A). Všechny tři znakové množiny va-

šeho ATARI jsou uvedeny v příloze 2. Všechny příkazy PRINT bychom mohli nahradit příkazy POKE do příslušné paměťové buňky s odvoláním na výpis v příloze 2, jako tomu bylo u písmene A.

Udělejme malý experiment. Pomocí příkazu POKE budeme do VRAM-ky ukládat vždy týž kód 33, ale budeme měnit grafické režimy namísto GRAPHICS 0.

```
10 FOR MODE=0 TO 8
20 GRAPHICS MODE
30 BEGIN=PEEK(88)+256*PEEK(89)
40 POKE BEGIN+2,33
50 FOR DELAY=1 TO 700:NEXT DELAY
60 NEXT MODE
```

Když tento program spustíme, vidíme, že se děje něco velmi zajímavého. Nejdřív se v grafickém režimu 0 v levém horním rohu objeví očekávané A. Podobně je tomu v grafických režimech 1 a 2. Avšak ve všech ostatních grafických režimech se neobjeví žádné A, ale pouze body různých barev!

DISPLAY LIST

Důvodem těchto rozdílů je různý způsob interpretace VRAM-ky ANTICem. Kdyby ANTIC prostě vzal cokoliv bylo ve VRAM-ce a vložil to na obrazovku, moc práce by tím 6502-ce neušetřil. 6502-ka by neustále musela dohlížet na to, jak má zobrazení vypadat a podle toho by musela příslušně upravovat VRAM, což by zabralo spoustu času. Proto tuto práci vykonává ANTIC. Vše, co 6502-ka musí udělat, je nastavit krátký program, kterému čip ANTIC rozumí a který ANTICu řekne, jak si 6502-ka přeje, aby byla VRAM interpretována. Zbytak dělá ANTIC. Tento krátký program sa nazývá display list. Abychom plně porozuměli všem možnostem, které nám display list jako programátorům dává, budeme se muset naučit nový programovací jazyk. Naštěstí tento jazyk nemá mnoho instrukcí, takže se velmi dobře učí. Nejprve uvedeme výpis těchto instrukcí jak v decimálním, tak v hexadecimálním tvaru a potom každou z nich detailněji popíšeme.

HEX.	DEC.	Instrukce
0	0	1 prázdný zobrazovací řádek
10	16	2 prázdné zobrazovací řádky
20	32	3 prázdné zobrazovací řádky

30	48	4 prázdné zobrazovací řádky
40	64	5 prázdných zobrazovacích řádků
50	80	6 prázdných zobrazovacích řádků
60	96	7 prázdných zobrazovacích řádků
70	112	8 prázdných zobrazovacích řádků
2	2	GR. 0 – textový režim
3	3	Speciální textový režim
4	4	Čtyřbarevný textový režim
5	5	Čtyřbarevný textový režim velký
6	6	GR. 1 – textový režim
7	7	GR. 2 – textový režim
8	8	GR. 3 – grafický režim
9	9	GR. 4 – grafický režim
A	10	GR. 5 – grafický režim
B	11	GR. 6 – grafický režim
C	12	Speciální dvoubarevný grafický režim 160x20
D	13	GR. 7 – grafický režim
E	14	Speciální čtyřbarevný grafický režim 160x40
F	15	GR. 8 – grafický režim
1	1	Skok na lokaci určenou následujícími 2 byty
41	65	Skok na lokaci určenou následujícími 2 byty s čekáním na zatemňovací impulz

Z každé instrukce lze vyrobit nové 4 instrukce tak, že nastavíme jeden až čtyři bity na 1. Jedná se o tyto bity:

BIT	INSTRUKCE
4	Povolení jemného vertikálního rolování
5	Povoleni jemného horizontálního rolováni
6	Další dva byty budou adresou VRAMu
7	Nastaví přerušeni display listu (DLI) pro další řádek

Fuj! Zdá se, že toho je najednou moc, ale rozebereme-li to krok za krokem, bude to celkem jednoduché. Začneme tím, že si prohlédneme jednoduchý display list. Vypíšeme si ho velice jednoduše, protože tak jako VRAM má i display list svůj vektor (lokace 560, 561), který nám v kterémkoliv okamžiku řekne, kde je display list umístěn. Není tedy problém si pomocí BASICu nechat vypsat ukázkový display list.

```
10 GRAPHICS 0
20 DL=PEEK(560)+256*PEEK(561)
30 FOR I=DL TO DL+31
40 PRINT PEEK(I);“ “;
50 NEXT I
```

Spustíme program a na obrazovce by se měla objevit následující čísla:

```
112 112 112 66 64 156 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 65 32 156
```

Pokud máte počítač se 48 kB RAM, mohou se pátá, šestá a poslední dvě čísla lišit od námi uvedených. Rozsekejme tedy display list byte po bytu, přičemž mějme stále na paměti, že display list je počítačový program, kde počítačem je v tomto případě ANTIC. Podíváme-li se na náš výpis instrukcí, vidíme, že kód 112 znamená vynechat 8 prázdných řádků. Jelikož máme v programu až tři 112, zdálo by se, že na začátku programu je ANTICu řečeno, aby vynechal 24 řádků. Je to správné?

Možná jste si všimli, že na různých televizních přijímačích je plocha vymezena vaším počítačem různě vpravo nebo více vlevo, nahoře nebo dole. Aby se tento rozdíl pokud možno smyl, většina display listů začíná 24 prázdnými televizními řádky. Samozřejmě si musíme pamatovat, že v GRAPHICS 0 je každý znak vysoký 8 bytů. Proto 24 prázdných televizních řádků představuje právě prostor, který by zabraly 3 řádky grafického režimu 0. Podobně tedy normální obrazovkový prostor GRAPHICS 0 představuje 192 obrazovkových řádků, tj. 8*24. Nic vám tedy nebrání, abyste si přidáním jednoho nebo dvou řádků vytvořili zákaznický display list s tím vědomím, že na vašem monitoru může fungovat, ale na jiném ne.

Další tři byty display listu byly 66, 64 a 156. Když se podíváme na výpis možných instrukcí display listu, zjistíme, že 66 tam není. Jedná se o součet 64+2. 64 je odvozeno nastavením šestého bitu instrukce ANTICu 2.

Dvojka říká, že se jedná o řádek grafického režimu 0; nastavený šestý bit indikuje, že se současně jedná o instrukci Load Memory Scan (LMS, čti: loud memory sken). Tato instrukce ANTICu říká, že následující dva byty display listu jsou vektorem na VRAM pro další v pořadí řádek a pak všechny další, dokud není zaznamenána nová instrukce LMS. Hodnoty 64 a 156 jsou v známém uspořádání nižší-vyšší, jsou tedy adresou VRAM-ky. Dekódujeme je takto: 64+255*156=40000. VRAM se tedy nachází od adresy 40000 a výše. Dalších 23 bytů samých dvojek ANTICu říká, že se jedná o řádky GRAPHICS 0 obsahující 40 bytů textu na řádek, kde každý byte je vysoký 8 televizních řádek. Spolu s instrukcí LMS to dohromady dělá 24 řádků. Další instrukcí je 65, která znamená skok na adresu určenou dvěma byty, které následují, a počkání na vertikální zatemňovací impulz.

Snímkové zatemnění

Abychom pochopili instrukci snímkového zatemnění, musíme si nejprve vysvětlit metodu, jakou je na obrazovce generován obrázek. Z pohledu osoby pozorující obrazovku začíná elektronový paprsek v levém horním rohu, vykresluje řádek, dokud nedosáhne pravého horního řádku. Pak přeskočí na druhý řádek a činnost se opakuje. Intenzita paprsku se mění podle toho, jak generuje tmavší či světlejší místa a vytváří obrázek. Zařízení tohoto typu se nazývají rastrovací zařízení a jsou daleko nejpoužívanějším systémem ke generování elektronického obrázku. Druhým častým typem je vektorové zařízení. Toto zařízení na rozdíl od rastrovacího vykresluje pouze čáru samotnou. Rastrovací zařízení vykreslí celou obrazovku, na které se čára nachází.

Pokud paprsek vykreslil celou obrazovku, vrací se do levého horního rohu a čeká na synchronizační signál, který dá pokyn opětovného vykreslování obrazovky. Tento interval, kdy paprsek čeká a nevykresluje, se nazývá snímkové zatemnění (vertical blank).

Váš počítač ATARI generuje 256 TV řádků na obrazovce a obrazovka je překreslována 50x za sekundu. Zdá se to velmi rychlé, ale vzhledem k rychlosti počítače se vykreslování obrazovky děje tempem vskutku hlemýždím. Vykreslení jedné obrazovky trvá 16,684 mikrosekund, interval snímkového zatemnění 1400 mikrosekund. Když uvážíme, že jeden takt strojového cyklu trvá méně než jednu mikrosekundu (560 nanosekund), je rozdíl mezi rychlostí počítače a televizoru zřejmý.

Poslední instrukce display listu je vlastně tříbytová. Následující dva byty (32 a 156) jsou ve skutečnosti nižší a vyšší byty adresy display listu, tedy téže adresy, kterou najdeme v buňkách 560 a 561, o kterých jsme se již zmínili. Instrukce skoku a čekání na snímkové zatemněni ANTICu dále říká, aby nezačínal vykonávání programu začínajícího na adrese dané posledními dvěma byty dřív, než skončí interval snímkového zatemnění. Čekání je nutné ze dvou důvodů. Za prvé kvůli synchronizaci počítače a televizního přístroje, aby obrázek byl stabilní; za druhé je tímto počítači 50x za sekundu dáno 1400 mikrosekund, kdy se nic neděje. ATARI tohoto času využívá k vnitřním ošetřujícím rutinám, jako např. obnovováni hodnot všech vnitřních časovačů apod. Využití tohoto času si probereme později v této kapitole.

Přímý přístup do paměti

Nyní už pomalu začínáme chápat, jak ATARI „vyrábí“ na obrazovce obrázek. Shrňme si to! Část paměti je vyhrazena k uchování informace, jež má být zobrazena (VRAM). Je interpretována ANTICem, který používá program nazývaný display list. K tomuto procesu ještě jednu poznámku: ANTIC a 6502-ka sdílejí vlastně oblast RAM-ky, která se nazývá VIDEO-RAM. 6502-ka produkuje a mění informaci uchovávanou v této části paměti, ANTIC ji čte, interpretuje a ukládá na obrazovku. Mělo by být zřejmé, že oba mikroprocesory nemohou mít simultánní přístup do téže paměti. Když ji potřebuje 6502-ka, ANTIC k ní nemůže mít přístup a zase naopak, když ANTIC z této oblasti čte, 6502 „má volno“. ANTIC má do paměti přístup pomocí procesu nazývanému Přímý přístup do paměti – DMA (z anglických slov Direct Memory Access).

Takto vlastně ANTIC okrádá 6503-ku. Až ANTIC dočte v paměti, 6502 opět začne pracovat. Tento DMA proces vlastně zpomaluje vykonáni vlastního programu. BASICovské programy lze urychlit asi o 30 % tím, že zakážeme DMA. Zákaz DMA v BASICu se vykoná jednoduchým příkazem

```
POKE 559,0
```

Opětovné povolení se provede příkazem

```
POKE 559,34
```

Ovšemže to má jeden háček. Obrazovka se vypne a zůstane „mimo provoz“ tak dlouho, dokud se nevykoná povoleni DMA. Cokoli však vytisknete na obrazovku příkazem PRINT v době zakázaného DMA, se po povoleni DMA na ní objeví.

Bylo napsáno mnoho článků o úpravě display listu. Zbytek této kapitoly bude věnován programům, které nelze v BASICu napsat, ale ke kterým je z BASICu přístup pomocí strojových rutin. Uvidíme, že tyto rutiny budou vykonávat celkem zajímavé úlohy.

Přerušení

Ve významu, v jakém je tento termín používán v této knize, budeme přerušení chápat jako vzkaz 6502-ce, aby zastavila vykonáváni činnosti, ať už to bylo cokoliv, a dělala to, co nadefinoval programátor. Po skončení ošetření přerušení může

6502 pokračovat v tom, co dělala před přerušením. Normálně se používají dva druhy přerušení – přerušení display listu (DLI – z anglických slov Display List Interrupt) a přerušení snímkového zatemnění (VBI – Vertical Blank Interrupt). Pro účely obou nelze právě kvůli pomalosti použít BASIC, ale strojový kód.

Přerušení Display Listu

Nejprve se budeme zabývat přerušením display listu. Když jsme probírali display list, zmínili jsme se, že nastavení sedmého bitu kterékoliv instrukce indikuje přerušení pro následující TV řádek. Co to znamená?

Ve druhé stránce je v lokacích $200 a $201 (decimálně 512, 513) vektor DLI. Vektor je, jak již víme, cosi jako ukazovátko, která někam ukazuje. Za normální situace buňka na adrese $200 obsahuje $B3, na adrese $201 je $E7. Vektor tedy ukazuje na adresu $E7B3. Tato paměťová buňka obsahuje hodnotu $40, což je kód instrukce RTI – návrat z přerušení. Jinými slovy vektor DLI normálně ukazuje na konec rutiny přerušení. Tento způsob zabezpečení má zabránit tomu, abyste nastavili DLI a poslali počítač někam pryč na náhodnou adresu a tam ho nutili zpracovat kód, který se na příslušných bytech nachází.

Při používáni DLI je třeba mít na paměti i časová omezení. Klasická DLI sestává ze tři částí. Část 1 je čas, který uplyne, než elektronový paprsek dokreslí řádek, jehož sedmý bit byl nastaven. Část 2 je doba mezi časem, kdy paprsek začíná vykreslovat řádek, na kterém je patrný efekt přerušení, po čas, kdy paprsek vstoupí do viditelné části řádku. Část 3 je od tohoto okamžiku až do konce rutiny DLI.

Vykreslení jednoho horizontálního řádku trvá 114 strojových cyklů. Ačkoliv sedmý bit je nastaven na začátku řádku, 6502 je o tom informována až po 36 cyklech. Z toho je zřejmé, že pomocí DLI se nedají implementovat dlouhé strojové programy – není na to dost času.

Jednoduchý příklad

DLI se velice často používá ke změně barvy pozadí obrazovky uprostřed obrazovky. Napišme si takovou rutinu a implementujme ji. Protože budeme přerušovat 6502-ku, musíme si být jisti, že jestliže plánujeme použití některého

z registrů nebo akumulátoru, na začátku rutiny musíme jejich původní hodnoty uchovat a před koncem rutiny je opět instalovat zpět. Podívejme se na program a proberme si ho:

```
             0100 ; ..........
             0110 ;
             0120 ; ..........
0000         0130      * = $600
D40A         0140 WSYNC  = $D40A
D018         0150 COLPF2 = $D018
             0160 ; ..........
             0170 ;
             0180 ; ..........
0600 48      0190       PHA
0601 A942    0200       LDA #$42
0603 8D0AD4  0210       STA WSYNC
0606 8D18D0  0220       STA COLPF2
             0230 ; ..........
             0240 ;
             0250 ; ..........
0609 68      0260       PLA
060A 40      0270       RTI
```

První věci, které si všimneme, je, že program nezačíná instrukcí PLA. Jediná PLA instrukce v programu byla použita k instalování původní hodnoty ze zásobníku do akumulátoru, kterou jsme si na řádku 190 schovali. Přesto má tato rutina probíhat v interakci s BASICem, a my víme, že každý příkaz USR v BASICu vyžaduje PLA k přemístění počtu parametrů ze zásobníku.

Tato zjevná chyba nám nezpůsobí žádně potíže, protože tuto rutinu nebudeme vyvolávat pomocí USR, ale přístup k ní bude zabezpečen přímo rutinou přerušení, kterou v BASICu zakrátko sestavíme. Přerušení nevyžadují žádné instrukce PLA, protože strojovému kódu nepředávají žádné informace a zásobník tudíž zůstává čistý.

Proberme si rutinu detailněji. Nejprve do akumulátoru vložíme hodnotu $42, což jako hexa číslo specifikuje v ATARI systému barev tmavě červenou. Toto číslo získáme jako šestnáctinásobek kódu příslušné barvy plus hodnota jasu. V šestnáctkové soustavě tedy máme 4*16+2, tj. barva, jejíž kód je 4 a hodnota jasu je 2 (tmavá). Tuto hodnotu (myslí se $42) vložíme do hardwarového registru pro barvu pozadí v GRAPHICS 0, který se nachází na adrese $D019 a v ATARI systému označení se nazývá COLPF2. Je důležité, abychom po-

chopili, proč používáme hardwarového registru a ne normálního barvicího registru, který se nachází na adrese 710 decimálně.

Pokud bychom vložili nějakou hodnotu reprezentující barvu do normálního barvicího registru na adrese 710, obrazovka by zčervenala a zůstala červená, dokud bychom hodnotu v tomto registru nezměnili. Avšak my chceme, aby pouze spodní část obrazovky byla červená a horní část aby zůstala modrá. K tomu musíme vědět, že hardwarový registr, $D018, je 50x za sekundu přepisován ze svého stínového registru, 710. Během každého intervalu vertikálního přerušení počítač ATARI čte hodnotu uloženou v registru na adrese 710 a ukládá ji do registru na adrese $D018. Takže obrazovka vlastně zmodrá 50x za sekundu, protože hodnota v registru 710 je 148. Nyní si objasněme, co dělá naše rutina.

Počítači ATARI je tedy 50x za sekundu řečeno, aby při vykreslováni obrázku zachoval modré pozadí. Naše rutina říká hardwarovému registru, že po zobrazení určitého počtu řádků by pozadí mělo být červené. Nyní se podívejme, co se stane, když je obrazovka dokreslena a začíná další interval vertikálního přerušení. ATARI vybere hodnotu 148 a vloží ji do hardwarového registru, čímž vrchní část obrazovky opět zmodrá, a naše rutina spodní část obrazovky „začervení“ atd. atd. Výsledný efekt je ten, že vrchní část je stále modrá a spodní stále červená. Kdybychom bývali místo registru na adrese $D018 na řádku 220 použili registr na adrese 710, byla by červená celá obrazovka.

Mezi uložením hodnoty barvy do akumulátoru a vložením této hodnoty do hardwarového registru barvy je ještě řádek 210, kde je odkaz na lokaci WSYNC na adrese $D40A. Tato lokace je po DLI velmi důležitá.

Musíme si představit, jak elektronový paprsek vykresluje obrazovku zleva doprava po řádcích. Pokaždé, když se dostane na pravý okraj, skočí zpět o jeden řádek níže a začíná opět vykreslovat další řádek. Pokud děláme něco takového, jako je změna barvy pozadí, musíme a chceme si být jistí, že změna nastává na začátku řádku a ne někde uprostřed. Pokud jen tak bez patřičného ošetření vložíme novou hodnotu barvy do hardwarového registru, změna barvy nastane bez ohledu na to, kde se zrovna paprsek nachází, právě v okamžiku vložení nové hodnoty na adresu $D018. Abychom se tomu vyhnuli, vkládáme na řádku 210 do lokace WSYNC jakoukoli hodnotu, a protože je po ruce zrovna číslo barvy, vložíme jej tam. Vůbec nezáleží na tom, o jakou hodnotu se jedná; jde jen o akt vložení.

Kdykoliv do WSYNC vložíme nějakou hodnotu, počítač jednoduše čeká na horizontální synchronizaci a pak až pokračuje dál. Tato synchronizace se provádí v době, kdy paprsek je mimo obrazovku a čeká na začátek vykreslování dalšího řádku. Po provedeni synchronizace počítač vykoná řádek 220.

Zbytek programu už pouze vrací do akumulátoru hodnotu, která tam byla před započetím rutiny přerušení a instrukce RTI zabezpečí návrat z přerušení k činnosti, která byla prováděna před ním.

Instalování DLI vyžaduje trošku programování v BASICu, protože rutina sama o sobě ještě změnu barvy nezaručí. Prohlédněme si tedy BASICovský program:

```
10 GOSUB 20000
20 HIBYTE=INT(ADR(SIPDLI$)/256)
30 LOBYTE=ADR(SIMPDLI$)-256*HIBYTE
40 POKE 512,LOBYTE
50 POKE 513,HIBYTE
60 DL=PEEK(560)+256*PEEK(561)
70 POKE DL+12,PEEK(DL+12)+128
80 POKE 54286,192
90 END
20000 DIM SIMPDLI$(13)
20010 FOR I=1 TO 13
20020 READ A
20030 SIMPDLI$(I,I)=CHR$(A)
20040 NEXT I: RETURN
20050 DATA 72,169,66,141,10,212,141,24,208,104
20060 DATA 64,246,243
```

Jak můžete vidět, nejprve na řádcích 20000 až 20060 sestavuje DLI rutinu, kterou jsme právě napsali, do řetězce. Potom musíme vypočítat, kam BASIC tento řetězec umístil, a jeho adresu rozdělit na vyšší a nižší byte. Poté můžeme počítači říci, kde se rutina nachází, takže když počítač narazí na instrukci DLI, ví, kam se obrátit a kde najde program, který má právě v tom čase vykonat. Tyto informace lze vždy nalézt na lokacích 512 a 513. Proto na řádcích 40 a 50 umístíme do těchto lokací 512 a 513 dva byty vypočítané adresy.

Řádek 60 nám najde display list, a protože jsme tyto instrukce používali již dříve, nebudeme se jimi více zabývat. Na řádku 70 nastavíme sedmý bit dvanáctého bytu display listu. Klidně bychom mohli, kdybychom chtěli změnu barvy pozadí níže na obrazovce, místo DL+12 použít DL+20 apod.

Vyzkoušejte si to a uvidíte. Nezapomeňte, že sedmý bit lze nastavit pouze u instrukce display listu; nesnažte se nastavit sedmý bit jednoho ze dvou bytů adresy ukazující na VRAM (DL+4 nebo DL+5), nebo sedmý bit jednoho ze dvou bytů adresy ukazující na začátek display listu (poslední dva byty display listu).

Řádek 80 je závažný! I kdybychom už učinili všechno, co se vyžaduje k povolení DLI, stále jsme ještě nesdělili počítači ATARI, že bychom je rádi povolili. Činíme tak na řádku 80. Aby přerušeni DL pracovalo, je tato instrukce nezbytná!

Složitější příklad: DLI rutina tabulkou

Přerušeni DL lze použít pro mnoho účelů. Uveďme si aspoň některé:

1. Změna barvy pozadí.
2. Změna barvy znaků.
3. Úplná změna znakové množiny (tím, že vložíme adresu, stránkovou, do příslušného hardwarového registru, $D409, ne do 756!).
4. Převrácení znakové množiny (dá se využít, při programování nějaké hry s kartami; hardwarový registr = $D401).
5. Simulace pohybu horizontu pomocí pohybováni DL přerušení.

Samozřejmě, že při použití DLI se meze vaší fantazii nekladou. My v této knize uvedeme jeden složitější příklad, abychom implementovali komplikovanější rutinu DLI. Mějte jen na paměti, že čas nám není příznivě nakloněn, takže vaše rutiny by měly být co nejstručnější.

Tento příklad nás seznámí s technikou vyhledávání tabulkou. Umožníme přerušení na každém řádku grafického modu 0 a na každém z nich změníme barvu pozadí. Za tímto účelem si vytvoříme tabulku barev, z nichž každá bude použita pro jeden řádek obrazovky. Bude tedy zapotřebí z tabulky postupně číst kódy barev a ukládat je v pravou chvíli do hardwarového barvicího registru. Tabulku zkonstruujeme ve čtvrté stránce, ale klidně by mohla být i v šesté nebo kdekoliv v chráněné oblasti paměti, jak jsme si to vysvětlili již dříve. Rutina DLI v assembleru vypadá takto:

```
0100 ; ..........
0110 ;
0120 ; ..........
```

```
0000        0130        * = $600
D018        0140 COLPF2   = $D018
D40A        0150 WSYNC    = $D40A
0400        0160 OFFSET   = $0400
            0170 ; ..........
            0180 ;
            0190 ; ..........
0600 48     0200         PHA
0601 98     0210         TYA
0602 48     0220         PHA
            0230 ; ..........
            0240 ;
            0250 ; ..........
0603 AC0004 0260         LDY  OFFSET
0606 B90204 0270         LDA  OFFSET+2,Y
0609 8D0AD4 0280         STA  WSYNC
060C 8D18D0 0290         STA  COLPF2
060F EE0004 0300         INC  OFFSET
0612 AD0004 0310         LDA  OFFSET
0615 CD0104 0320         CMP  OFFSET+1
0618 9005   0330         BCC  SKIP
061A A900   0340         LDA  #0
061C 8D0004 0350         STA  OFFSET
            0360 ; ..........
            0370 ;
            0380 ; ..........
061F 68     0390 SKIP    PLA
0620 A8     0400         TAY
0621 68     0410         PLA
0622 40     0420         RTI
```

Zaměřte se především na rozdíly mezi touto a předchozí rutinou. Protože tato rutina pracuje jak s akumulátorem, tak s registrem Y, musíme jejich původní hodnoty před započetím hlavního těla rutiny uchovat do zásobníku. Činíme tak posloupností instrukcí PHA, TYA, PHA.

Hlavní odlišnost mezi touto a předchozí DLI rutinou je patrna z řádků 260, 270 a 300 až 350. Na řádku 260 do registru Y uložíme hodnotu paměťové lokace OFFSET. Toto číslo znamená relativní adresu umístění kódu barvy v tabulce, která začíná na adrese $402. Jestliže se OFFSET rovná 5, vybereme z tabulky 6. barvu, protože první barva má OFFSET = 0. Takto vybrané číslo se stává hodnotou, kterou vložíme do hardwarového registru. Na řádcích 300 až 350 se provede inkrementace paměťové lokace OFFSET a zjistí se, zda už byly použity všechny barvy. Jestliže je tomu tak, vyuzluje se obsah OFFSET

a vyskočí se z přerušení. Jestliže ne, provede se pouze příslušně ošetřen návrat z přerušení. Všimněte si, že v lokaci $401 (OFFSET+1) je informace o počtu barev v tabulce, takže není problém určit, zdali jsme už použili všechny.

Nyní si prohlédněte program v BASICu, který nám obstará přístup k naší DLI rutině:

```
10 GOSUB 20000
20 HI=INT(ADR(TABLEDLI$)/256):LO=ADR(TABLEDLI$)-HI*256
30 GRAPHICS 0:SETCOLOR 1,0,0
40 RESTORE 270
50 FOR I=0 TO 27
60 READ A
70 POKE 1026+I,A
80 NEXT I
90 POKE 1024,0
100 POKE 1025,27
140 DL=PEEK(560)+256*PEEK(561)+6
150 DLBEG=DL-6
160 FOR I=0 TO 2
170 POKE DLBEG+I,240
180 NEXT I
190 POKE DLBEG+I,194
200 FOR I=DL TO DL+22
210 POKE I,130
220 NEXT I
230 POKE 512,LO:POKE 513,HI
240 POKE 54286,192
250 LIST
260 END
270 DATA 6,22,38,54,70,86,102,118,134,150,166,182,198,214
280 DATA 230,246,246,230,214,198,182,166,150,134,118,102,86,70
290 DATA 54,38,22,6
20000 DIM TABLEDLI$(35)
20010 RESTORE 20060
20020 FOR I=1 TO 35
20030 READ A
20040 TABLEDLI$(I,I)=CHR$(A)
20050 NEXT I:RETURN
20060 DATA 72,152,72,172,0,4,185,2,4,141
20070 DATA 10,212,141,24,208,238,0,4,173,0
20080 DATA 4,205,1,4,144,5,169,0,141,0
20090 DATA 4,104,168,104,64
```

Podprogram na řádku 20000 sestaví naši rutinu do řetězce. Poté zjistíme, kde se řetězec nalézá v paměti a rozložíme

adresu jeho počátku do známé konfigurace nižší/vyšší byte. Příkaz GRAPHICS 0 nám zajistí, že budeme pracovat s display listem, se kterým chceme pracovat. Budeme začínat s černým pozadím. Potom napoukujeme kódy barev, které si budeme přát vidět na obrazovce, do tabulky ve čtvrté stránce. Změnou dat na řádku 270 docílíme jiné kombinace barev. Do paměťové lokace $400 (decimálně 1024) vložíme nulu, protože chceme začínat první barvou v tabulce. Kdybychom zde vložili jinou hodnotu, řekněme 10, celé spektrum barev by se na obrazovce posunulo, tzn. že bychom začínali jedenáctou barvou a končili desátou, potom si zjistíme začátek display listu a začátek instrukci grafického režimu 0 (tj. najdeme adresu, kde začíná série dvojek) a nastavíme nejvyšší bit každé instrukce a tím tedy povolíme přerušení na každém řádku. Všimněte si, že přerušení display listu lze nastavit dokonce i pro první tři instrukce, které ANTICu pouze říkají, že má vynechat 8 obrazovkových řádků. Takže naše rutina vlastně zabarví každou z osmic jinou barvou! Na řádku 230 počítači řekneme, kde se DLI rutina nachází a potom už dovolíme přerušení display listu. Příkaz LIST na řádku 250 na obrazovce pouze zobrazí jakýsi text a jeho rolováním vytvoří zajímavý efekt.

Na závěr ještě jednu důležitou poznámku: počítače ATARI používají paměťové lokace WSYNC pro vytvoření zvuku, který doprovází stlačení klávesy na klávesnici, proto programy, která ve velkém používají přerušení display listu, třeba jako tento, se mohou stlačením klávesy zhroutit. Nejjednodušším řešením se ukazuje klávesnici prostě nepoužívat. Pokud váš program vyžaduje předání informace z vnějšku, např. výběr z nějakého menu apod., lze za tímto účelem použít joysticku nebo kláves START, SELECT nebo OPTION. Rozumí se samo sebou, že klávesa RESET eliminuje veškerá přerušení display listu, protože tento příkaz vždy nastaví display list pro grafický režim 0.

Přerušení snímkového zatemnění

Druhým běžným typem přerušení, používaným ve vašem ATARI, je tzv. přerušení snímkového zatemnění (dále také VBI; zkratka z anglických slov Vertical Blank Interrupt). Pomocí tohoto systému lze na ATARI uskutečnit zpracování více programů najednou. Ačkoliv se ve skutečnosti nejedná o multizpracování v pravém slova smyslu, je možno sestavit dva programy tak, že jeden se vykonává v normálním čase a druhý v intervalu snímkového zatemnění. V konečném efektu se zdá, že oba programy běží současně.

Krásným příkladem využiti tohoto multizpracování je program od Chrise Crawforda Východní fronta. Tato hra je simulací německé operace Barbarossa ve druhé světové válce, kde Sovětskou armádu řídí počítač. Počítač o svých tazích „přemýšlí“ v době snímkového zatemnění a vaše tahy vykonává v reálném čase. Znamená to, že čím déle o svém tahu přemýšlíte, tím více vertikálních přerušení se uskuteční a tedy i tím více času má počítač na určení svého tahu.

Všimli jste si již hudby, která často zní, když hrajete nějakou hru? Tato hudba vlastní program vůbec nezpomaluje, protože je hrána pouze v době snímkového zatemnění. Náš vzorový program vám ukáže, jak tohoto efektu docílit. Ačkoliv i tato rutina je přemístitelná, umístíme ji do šesté stránky paměti. Konečně, sami už víte, jak ji převést do řetězte, a tak ji učinit přemístitelnou. Při práci se snímkovým zatemněním se stává každá VBI rutina ze dvou částí. První částí je rutina samotná, druhou částí je krátká rutina, která tuto VBI rutinu instaluje.

Operační systém vašeho počítače ATARI za normálních okolností ukazuje na specifickou rutinu, která se vykonává při každém snímkovém zatemnění. Rutina je tvořena dvěma částmi. První se nazývá přímá a druhá nepřímá VBI rutina. Vektor přímé rutiny se nachází na adrese $0222. Jedná se o dvoubytovou adresu, na kterou počítač skočí, aby vykonal rutinu přímého VBI (normálně ukazuje na obslužnou rutinu začínající na adrese $E45F). Tato rutina končí nepřímou rutinou VBI, která se normálně nachází na adrese $E462 a ukazuje na ní vektor na adrese $0224. Schématicky si to můžeme znázornit takto:

VBI -> $0222 -> $E45F -> $0224 -> $E462 -> RTI

Abychom tedy umístili naši vlastní rutinu místo rutiny počítače ATARI, musíme pouze změnit obsah vektoru tak, aby ukazoval na naši rutinu. Nejprve se však musíme rozhodnout, kterou rutinu chceme nahradit. Dlouhé VBI rutiny musí nahradit normální rutiny, protože není dost času na to, aby se v průběhu jednoho snímkového zatemnění vykonaly obě. Protože v přímé rutině, na niž ukazuje vektor $0222, se vykonává hodně systémových prací (obnova systémových čítačů, kopírování stínových registrů, čtení joysticků aj.), proto je bezpečnější nahradit nepřímou rutinu.

Kdykoliv měníme vektor, stojíme před zásadním problémem. Zřejmost tohoto problému se zvětšuje právě u vektoru snímko-

vého zatemnění, který je používán 50x za sekundu bez ohledu na to, v jakém je stadiu vykonávání programu. Néjlépe to pochopíte na jednoduchém příkladu. Víme, že na adrese $0222 je hodnota $5F a na adrese $0223 je $E4. Řekněme, že bychom chtěli, aby tento vektor ukazoval místo na $E45F na adresu $0620. Změníme tedy nejprve lokaci $0222 na %$0 a do lokace $0223 vložíme $06. Snadné, že? Ale co když v době mezi vložením hodnoty $20 a $06 dojde k VBI. Protože jsme stihli změnit pouze jeden byte, hodnota vektoru bude $E420. Počítač tím pádem odskočí do země nikoho, začne vykonávat program na adrese $E420, což nutně povede ke krachu. Abychom se tomuto problému vyhnuli, pomůže nám samo ATARI. ATARI je vybaveno vlastní krátkou rutinou, která mění vektor snímkového zatemnění. Nejlépe ji opět pochopíme, když si ji napíšeme a prohlédneme:

```
LDY #$20
LDX #$06
LDA #07
JSR SETVEV
RTS
```

Kdyby naše rutina nahrazovala přímou rutinu snímkového zatemnění, uložili bychom ještě před skokem JSR SETVBV ($E45C). A to je vlastně vše. Mějte jen na paměti, že v okamžiku vykonávání táto instalační rutiny se vlastní rutina již musí nacházet na svém místě. Jestliže tomu tak nebude, program se v padesátině sekundy zhroutí.

Samozřejmě, že obě části snímkového zatemnění jsou časově omezené. Nepřímá část rutiny může být dlouhá maximálně 20000 strojových cyklů, přímá rutina asi 2000 strojových cyklů. Jestliže by vaše rutina tento časový limit překročila, došlo by opět k zhroucení, protože TV display a počítač nebudou schopny se sesynchronizovat.

Takže teď, kdy jsme se rozhodli nahradit nepřímou část rutiny a víme, jak ji instalovat, podívejme se na vlastní rutinu, která v intervalu snímkového zatemnění zahraje hudbu.

```
                0100 ; ..........
                0110 ;
                0120 ; ..........
0000            0130       * = $0600
00C0            0140 COUNT1  = $00C0
0224            0150 VVBLKD  = $0224
00C2            0160 COUNT2  = $00C2
E45C            0170 SETVBV  = $E45C
```

```
0660         0100 MUSIC   = $0660
E462         0190 RETURN  = $E462
D200         0200 SND     = $D200
D201         0210 VOL     = $D201
             0220 ; ..........
             0230
             0240 ; ..........
0600 68      0250 ;       PLA
             0260 ; ..........
             0270
             0280 ; ..........
0601 A900    0290         LDA  #0
0603 85C0    0300         STA  COUNT1
0605 85C2    0310         STA  COUNT2
             0320 ; ..........
             0330 ;
             0340 ; ..........
0607 A020    0350         LDY  #$20
0609 A206    0360         LDX  #$06
060B A907    0370         LDA  #07
060D 205CE4  0380         JSR  SETVBV
0610 60      0390         RTS
             0400 ; ..........
             0410 ;
             0420 ; ..........
0611         0430        * = 30620
0620         0440         INC  COUNT1
0622 A6C0    0450         LDX  COUNT1
0624 E00C    0460         CPX  #12
0626 9005    0470         BCC  NO
0628 A900    0480         LDA  #0
062A 8D01D2  0490         STA  VOL
062D E00F    0500 NO      CPX  #15
062F B003    0510         BCS  PLAY
0631 4C62E4  0520         JNP  RETURN
0634 A900    0530 PLAY    LDA  #0
0636 8500    0540         STA  COUNT1
0638 A6C2    0550         LDX  COUNT2
063A BD6006  0560         LDA  MUSIC,X
063D 8D00D2  0570         STA  SND
0640 A9A6    0580         LDA  #$A6
0642 8D01D2  0590         STA  VOL
0645 E6C2    0600         INC  COUNT2
0647 A6C2    0610         LDX  COUNT2
0649 E008    0620         CPX  #8
064B 9004    0630         BCC  DONE
064D A900    0640         LDA  #0
```

```
064F 85C2   0650        STA  COUNT2
0651 4C62E4 0660 DONE   JMP  RETURN
            0670 ; ..........
            0680 ;
            0690 ; ..........
0654        0700       * = $0660
0660 F3     0710         .BYTE 243,243,217,243,204,243,217,243
0661 F3
0662 D9
0663 F3
0664 CC
0665 F3
0666 D9
0667 F3
```

V inicializační části rutiny jsou vynulovány dva čítače: jeden nese informaci o tom, která nota je hrána; druhý zaznamenává délku hrané noty. Dále následuje nám již známá krátká rutina, která nastaví vektor na tu naši hudbu hrající rutinu. Ta začíná na adrese $0620 (řádek 430). Nejprve inkrementujeme čítač délky noty. Pokud se hodnota v čítači rovná 12, zvuk vypneme, jinak necháme notu hrát déle. Zvuk se dá vypnout tak, že do příslušného hardwarového registru vložíme 0, nebo do registru řídicího hlasitost zvuku vložíme také 0 (řádek 490).

Aby mezi notami byla krátká mezera, počkáme, dokud v čítači nebude 15. Aby byla zahrána následující nota, vložíme do časového čítače nulu a z čítače not vybereme číslo noty, která bude hrána. Tohoto čísla využijeme jako relativního umístění noty v tabulce not začínající na adrese $0660 (řádek 710). Proto tedy, jestliže COUNT2=2, bude hrát třetí nota. Vyhledávání noty v tabulce a její zahrání se realizuje na řádcích 560 až 590. Práce s čítači na řádcích 610 až 630 je zřejmá.

Z rutiny vyskakujeme instrukcí JMP RETURN, RETURN=$E462, kde se rutina ukončí normální nepřímou rutinou. Kdybychom byli bývali nahradili přímou část VBI, výskok z rutiny by se byl uskutečnil přes adresu $E45F a v případě velmi dlouhé rutiny přes $462, čímž by se vyeliminovala celá normální ATARI VBI, ale získali bychom tím spoustu času.

Sestavení VBI rutiny v BASICu je celkem jednoduché:

```
10 GOSUB 19000
20 GOSUB 20000
30 GOSUB 21000
40 X=USR(1536)
50 END
19000 RESTORE 19050
19010 FOR I=1536 TO 1552
19020 READ A
19030 POKE I,A
19040 NEXT I:RETURN
19050 DATA 104,169,0,133,192,133,194,160,32,162
19060 DATA 6,169,7,32,92,228,96
20000 RESTORE 20050
20010 FOR I=1568 TO 1619
20020 READ A
20030 POKE I,A
20040 NEXT I: RETURN
20050 DATA 230,192,166,192,224,12,144,5,169,0
20060 DATA 141,1,210,224,15,176,3,76,98,228
20070 DATA 169,0,133,192,166,194,189,96,6,141
20080 DATA 0,210,169,166,141,1,210,230,194,166
20090 DATA 194,224,8,144,4,169,0,133,194,76,98,228
21000 RESTORE 21050
21010 FOR I = 1632 TO 1639
21020 READ A
21030 POKE I,A
21040 NEXT I:RETURN
21050 DATA 243,243,217,243,204,243,217,243
```

V programu jsou nejprve tři odskoky do podprogramů, kde jsou rutiny pomocí příkazů POKE uloženy do paměti a pak příkazem USR je odstartována rutina, která nastaví VBI vektor. První série POKEů uloží inicializační rutinu na stránku 6, druhá série uloží do paměti VBI rutinu samotnou a v třetí sérii je instalována na příslušné místo tabulka barev. Řádek 40 aktivuje rutinu – hudbu a hurá – máte hudební doprovod k dlouhým programovacím sezením. Hudba bude hrát tak dlouho, dokud nestisknete RESET nebo dokud nevrátíte do vektoru nepřímé rutiny původní hodnotu.

Hudba hrána touto rutinou je svým způsobem omezena. Všechny noty musí být stejně dlouhé. Navíc sestává jen z jednoho hlasu. Samozřejmě, že na ATARI lze zkomponovat i daleko libozvučnější melodie. Nyní si už takové programy můžete napsat sami.

Na závěr ještě jedna poznámka týkající se VBI. S velkou výhodou se snímkového zatemnění dá využít pro čtení joysticku a pohyb hráčů po obrazovce. Jestliže umístíme rutinu čtení joysticku do intervalu snímkového zatemnění, zbavíme se tak jedná z časově nejnáročnějších části BASIC-ovskáho programu a umožníme počítači, aby mohl změnit polohu hráče 50x za sekundu, aniž by se tím zpomalil vlastní program. Jako cvičeni si můžete rutinu ze 7. kapitoly převést na VBI rutinu.

Jemné rolování

Zbývají nám již pouze dva poslední bity instrukcí display listu: povolení horizontálního a vertikálního jemného rolování (čtvrtý a pátý bit). Jemné rolování je prostředek, který programátorovi umožní uskutečnit jeden z nejzajímavějších a nejpozoruhodnějších efektů počítačů ATARI - rolováni zdánlivě nekonečné obrazovky. Vlastně jedním z nejkrásnějších příkladů jemného rolováni je již zmíněný program Východní fronta, kde po obrazovce roluje celá mapa východní Evropy.

Nyní se zmíníme o jemném horizontálním rolování a probereme si jemné vertikální rolování, *byste byli schopni napsat vlastní programy využívající tohoto efektu. Jemné horizontální rolováni má jednu obtíž, se kterou se musíme vypořádat. Jak již víte, display list normálního GRAPHICS 0 obsahuje pro každý řádek kód ANTICu 2, jenž ANTICu říká, že si přejeme VRAM interpretovanou 40 byty na řádek jako text. Avšak chceme-li rolovat informací doleva, nastává problém. Prohlédněte si obrázek, abyste problém viděli graficky:

```
Obrazovka  Číslo sloupce
                    1111111111222222222333333333
           0123456769012345678901234567090123456789
.
.
Řádek 5    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Řádek 6    bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
Řádek 7    ccccccccccccccccccccccccccccccccccccccccc
```

Pokud se zobrazuje pouze 40 znaků, není problém žádný. Avšak jak můžeme rolovat oknem obrazovky? Např. zarolujeme-li obrazovkou doprava (tj. informace roluje doleva), jaký bude poslední znak každého řádku? Řádek 5 bude končit znakem b, řádek 6 znakem c, atd. Není to tedy opravdové horizontální rolováni, ale vlastně horizontální a vertikální rolováni zároveň. Abychom zabezpečili čisté horizontální rolování,

potřebujeme zvláštní tvar display listu. Musíme vytvořit takový display list, který má více místa než 40 znaků na řádek, takže když informaci rolujeme, objevuje se na obrazovce to, co prve bylo schováno mimo obrazovku. Naštěstí techniky pro vytvořeni takovéhoto display listu již známe. Pro každý řádek potřebujeme zvláštní instrukci LMS (Load Memory Scan) a každému řádku musíme vyhradit daleko více než 40 bytů. Navrhněme tedy display list, kde každému řádku bude vyhrazeno 250 bytů, takže VRAM bude 6x větší než u obrazovky GRAPHICS 0. Získáme tím spoustu prostoru k rolováni. Zobrazení bude vypadat takto:

```
Obrazovka  Číslo sloupce
                  1111111111222222222333333333
         01234567890123456789012345678901234567890
.
.
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
ccccccccccccccccccccccccccccccccccccccccccccccccccc
.
.
```

Ze zobrazení vyplývá, že paměť vyhrazená jednomu každému řádku zobrazení bude větší než obrazovka sama. Takto vyřešíme problém míchaného rolování.

Druhým rysem našeho display listu je, že každé instrukce LMS má nastavený čtvrtý bit, takže ke kódu LMS 64 musíme přičíst 16. Samozřejmě nesmíme zapomenout na instrukci, jež ANTICu řekne, jak řádek interpretovat, tedy 2, protože použijeme GRAPHICS 0 (ANTIC mode 2). Součet tedy bude 64+16+2=82, což je výsledný kód každého řádku našeho nového display listu.

Ovšemže bychom mohli display list vytvořit v BASICu a bylo by to téměř totéž, co jsme již dělali, proto si trochu zaexperimentujeme a k vytvoření display listu použijeme assembleru. Display list umístíme na šestou stránku, kde bude bezpečný. Zopakujte si, že každý display list začíná 24 prázdnými televizními řádky, pak instrukcemi reprezentace řádků a končí instrukcí skoku s počkáním na snímkové zatemnění. Prohlédněte si program konstrukce display listu:

```
0100 ; ..........
0110 ; init
0120 ; ..........
```

```
0000        0130       * = $0600
0600        0140 DLIST   = $0600
0058        0150 SAVMSC  = $58
0230        0160 SDLSTL  = $0230
E54C        0170 SETVBV  = $E45C
            0180 ; ..........
            0190 ;
            0200 ;
            0210 ;
            0220 ;
            0230 ;
            0240 ; ..........
0600 68     0250 INIT   PLA
0601 A970   0260        LDA  #$70
0603 8D0006 0270        STA  DLIST
0606 8D010a 0280        STA  DLIST+1
0609 8D0206 0290        STA  DLIST+2
060C A018   0300        LDY  #24
060E A203   0310        LDX  #3
0610 A952   0320        LDA  #82
0612 9D0006 0330        STA  DLIST,X
0615 E8     0340        INX
0616 A556   0350        LDA  SAVMSC
0618 9D0006 0360        STA  DLIST,X
061B E8     0370        INX
061C A559   0380        LDA  SAVMSC+1
061E 38     0390        SEC
061F E918   0400        SBC  #24
0621 9D0006 0410        STA  DLIST,X
0624 E8     0420        INX
0625 88     0430        DBY
0626 A952   0440 LOOP   LDA  #82
0628 9D0006 0450        STA  DLIST,X
062B E8     0460        INX
062C BDFD05 0470        LDA  DLIST-3,X
062F 18     0480        CLC
0630 69FA   0490        ADC  #250
0632 9D0006 0500        STA  DLIST,X
0635 E8     0510        INX
0636 BDFD05 0520        LDA  DLIST-3,X
0639 6900   0530        ADC  #0
063B 9D0006 0540        STA  DLIST,X
063E E8     0550        INX
063F 88     0560        KEY
0640 D0E4   0570        BUB  LOOP
0642 A941   0580        LDA  #65
0644 9D0006 0590        STA  DLIST,X
```

```
0647 E8     0600        INX
0648 A900   0610        LDA  #0
064A 9D0006 0620        STA  DLIST,X
064D 8D3002 0630        STA  SDLSTL
0650 E8     0640        INX
0651 A906   0650        LDA  #6
0653 9D0006 0660        STA  DLIST,X
0656 8D3102 0670        STA  SDLSTL+1
            0680 ; ..........
            0690 ;
            0700 ;
            0710 ; ..........
0659 68     0720 PLA
065A AA     0730 TAX
065B 68     0740 PLA
065C A8     0750 TAY
065D A907   0760 LDA  #$07
065F 205CE4 0770 JSR  SETVBV
0662 60     0780 RTS
```

Začneme tím, že na vrchol display listu uložíme 3x $70, což jsou právě ony prázdné televizní řádky. Do registru Y uložíme počet řádků, které budeme konstruovat. Do registru X jsme uložili číslo 3, protože tři řádky display listu jsme již vytvořili. Hodnotu charakterizující každý řádek uložíme do akumulátoru a vložíme ji na příslušné místo do display listu. S každým vytvořením nové instrukce display listu se inkrementuje registr X, protože představuje relativní adresu instrukce od počátku display listu.

Protože každá instrukce je zároveň i instrukcí LMS, znamená to, že po každé instrukci musí následovat dva byty adresy, kde ANTIC najde informaci, co má zobrazovat. Začátek display listu by měl ukazovat na začátek VRAM. Vektor začátku VRAM-ky se vždy nachází na adrese $58, $59. Avšak náš značně rozšířený display list vyžaduje více VRAM, a ta zase více místa na umístění. Toto místo pro umístění VRAM-ky vytvoříme tak, že od vyššího bytu vektoru odečteme 24 stránek. Přesun informace z lokace $58 je na řádcích 350 až 370, přesun informace z vyššího bytu vektoru spolu s odečtením stránek pro rozšíření je vykonán na řádcích 300 až 420. Protože jsme tímto zkompletovali jeden řádek nového display listu (LMS + adresa), dekrementujeme registr Y na řádku 430.

Řádky 440 až 570 vytváří velkou smyčku, která bude vykonána 23x. Každým průchodem cyklem se vytvoří 1 řádek display listu. První instrukcí je už známá 82. Potom si vytáhneme

nižší byte předcházející adresy a přičteme k němu 250 (řádky 470 až 510). Před sčítáním nezapomeňte uplatnit instrukci CLC! Tímto způsobem bude každý následující řádek o 250 bytů výše než předcházející, takže každý řádek bude místo 40 bytů dlouhý 250 bytů.

Zdá se, že na řádcích 520 až 540 se neděje nic, že? K jakémusi číslu se přičítá 0 a ukládá se do paměti. Nezapomínejte však, že každé sčítání se děje spolu s indikačním registrem přenosu (carry bit)! Od poslední instrukce nebyl indikační registr přenosu nulován. Proto, jestliže výsledek předchozího sčítáni byl větší než 255, bude nižší byte adresy uložený do display listu na řádku 500 vlastně hodnota součtu minus 256. Tím by se však nastavil i indikační registr přenosu a přičtení nuly instrukci ADC 0 znamená přičtení hodnoty 1! Adresa tak bude ukazovat na správná místo. Tuto operaci lze zapsat i jiným způsobem:

```
LDA ADDR1
CLC
ADC #250
STA ADDR1
BCC PASS
INC ADDR2
PASS…
```

Cyklus ukončíme opět dekrementací registru Y. Pokud je obsah registru nenulový, opakuje se celý cyklus znovu. Pokud je v registru Y nula, již známým způsobem nastavíme instrukci skoku s čekáním na snímková zatemnění tak, aby ukazovala na začátek display listu na šesté stránce. Činíme tak na řádcích 580 až 670. Všimněte si řádků 530 a 670. Tyto vkládají adresu našeho nového display listu do lokací $230 a $231, což je interní vektor počátku display listu, kterého ATARI (i ANTIC) používá.

Aby bylo rolování rychlé a plynulé, umístíme naši rutinu do intervalu VBI. Adresu rutiny rolování předá sestavovací rutině BASICovský program. Řádky 720 až 770 tuto adresu vyberou ze zásobníku a provede se instalace rolovací rutiny do nepřímé části snímkového zatemnění. Nakonec se řádkem 780 vrátíme do BASICu.

Nyní, kdy máme zkonstruovaný display list, nám už pouze zbývá napsat krátký strojový program, který bude manipulovat se samotnou rolovací rutinou. Abychom tomuto programu lépe porozuměli, proberme si nejdřív mechanismus rolování.

Znak v grafickém režimu 0 je široký 8 bitů. Hrubého rolování se dosáhne tak, že se v jednom kroku rolování přemístí jeden znak buď doprava nebo doleva. My však chceme jemné rolování, ve kterém se znak přemístí v jednom kroku doleva nebo doprava o šířku jednoho bitu. ATARI nám za tímto účelem dává k dispozici registr nazývaný HSCROL ($D404). Odpovídající registr vertikálního rolování, který pracuje úplně stejným způsobem, se jmenuje VSCROL ($D405).

HSCROL nám zabezpečí rolování osmibitového znaku o šířku jednoho bitu, ale pak musí být vynulován. Pokud se do HSCROL zapíše nula, je pozice znaku normální. Jestliže do HSCROL zapíšeme 1, znak se posune o jeden znak doleva. Zapisování postupně hodnot 2,3 až 7 znak posunujeme. Po zapsání sedmičky zapíšeme nulu a celý znak posuneme o celou šířku znaku doleva změnou adresy v instrukci LMS na každém řádku. Názorně celé situace vypadá takto:

```
Číslo zapsané do HSCROL
--------------------------
    0       1       2
--------------------------
.......I ......I. ......I..
.......I ......I. ......I..
.......I ......I. ......I..
.......I ......I. ......I..
.......I ......I. ......I..
.......I ......I. ......I..
.......I ......I. ......I..
.......I ......I. ......I..
```

Po ukončení celého cyklu od 0 do 7 můžeme celý postup znovu opakovat. Takto můžeme prorolovat celou šířku VRAM-ky. Rutina uvedená níže dokonce ani netestuje šířku paměti, takže pokud ji necháte běžet dostatečně dlouho, proroluje se až na vrchol paměti. Získáte zcela nový a svým způsobem i unikátní náhled na operační systém vašeho ATARI!

Nyní, kdy víme, co budeme dělat, si prohlédněme program:

```
                0100 ; ..........
                0110 ;
                0120 ; ..........
0000            0130       * = $600
0600            0140 DLIST   = $600
D404            0150 HSCROL  = $D404
E462            0160 XITVBV  = $E462
```

```
            0170 ; ..........
            0180 ;
            0190 ; ..........
0600 48     0200        PHA
0601 8A     0210        TXA
0602 48     0220        PHA
            0230 ; ..........
            0240 ;
            0250 ; ..........
0603 A207   0260        LDX  $7
0605 8E04D4 0270 LOOP   STX  HSCROL
0608 CA     0280        DEX
0609 10FA   0290        BPL LOOP
060B A207   0300        LDX #7
060D 8E04D4 0310        STX HSCROL
            0320 ; ..........
            0330 ;
            0340 ; ..........
0610 A200   0350        LDX #0
0612 BD0406 0360 LOOP2  LDA DLIST+4,X
0615 18     0370        CLC
0616 6901   0380        ADC #1
0618 9D0406 0390        STA DLIST+4,X
061B BD0506 0400        LDA DLIST+5,X
061E 6900   0410        ADC #0
0620 9D0506 0420        STA DLIST+5,X
0623 E8     0430        INX
0624 E8     0440        INX
0625 E8     0450        INX
0626 E048   0460        CPX #72
0626 90B8   0470        BCC LOOP2
            0480 ; ..........
            0490 ;
            0500 ; ..........
062A 68     0510        PLA
062B AA     0520        TAX
062C 68     0530        PLA
062D 4062E4 0540        JMP XITVBV
```

Protože se tato rutina bude odbývat v intervalu snímkového zatemnění, je na řádcích 200 až 220 provedena úschova obou registrů, akumulátoru a X, kterých budeme používat. Dále na řádcích 260 až 310 rychle proběhneme cyklem přes všech 8 bitů registru HSCROL s tím, že před vyskočením z cyklu nastavíme HSCROL na 7. V dalším cyklu (řádky 350 až 470) jednoduše projdeme celý display list a zvětšíme každou adresu o 1 byte, čímž docílíme hrubého rolování. Kdybychom rolovali

vertikálně, museli bychom ke každé adrese přičíst 250, abychom dosáhli hrubého rolování o jeden řádek nahoru (při užití normální VRAM bychom přičetli 40). V tomto cyklu používáme registr X jako čítač bytů a ne jako čítač řádků, musíme ho tedy při každém průchodu zvětšovat 3x.

Nakonec na řádcích 510 až 530 znovu obnovíme registry, jejichž hodnoty jsme uchovali, a na řádku 540 opustíme rutinu odskokem na rutinu výstupu z nepřímé rutiny snímkového zatemnění.

Nyní můžeme napsat velmi jednoduchý program v BASICu, kterým obě rutiny zprovozníme:

```
10 GOSUB 20000
20 GOSUB 30000
30 FOR I=34000 TO 40000 STEP 5:P0KE I,86:NEXT I
40 DUMMY=USR(ADR(DLSCROLL$),ADR(SCROLL$))
50 GOTO 50
20000 DIM DLSCHOLL$(99)
20010 FOR I=1 TO 99
20020 READ A
20030 DLSCROLL$(I,I)=CHR$(A)
20040 NEXT I: RETURN
20050 DATA 104,169,112,141,0,6,141,1,6,141
20060 DATA 2,6,160,24,162,3,169,82,157,0
20070 DATA 6,232,165,88,157,0,6,232,165,89
20080 DATA 56,233,24,157,0,6,232,136,169,82
20090 DATA 157,0,0,232,189,253,5,24,105,250
20100 DATA 157,0,6,232,189,253,5,105,0,157
20110 DATA 0,6,232,136,208,228,169,65,157,0
20120 DATA 6,232,169,0,157,0,6,141,48,2
20130 DATA 232,169,6,157,0,6,141,49,2,104
20140 DATA 170,104,168,169,7,32,92,228,96
30000 DIM SCROLL$(48)
30010 FOR I=1 TO 48
30020 READ A
30030 SCROLL$(I)=CHR$(A)
30040 NEXT I:RETURN
30050 DATA 72,138,72,162,7,142,4,212,202,16
30060 DATA 250,162,7,142,4,212,162,0,189,4
30070 DATA 6,24,105,1,157,4,6,189,5,6
30080 DATA 105,0,157,5,6,232,232,232,224,72
30090 DATA 144,232,104,170,104,76,98,228
```

Tento program nejprve pomocí podprogramů na řádcích 20000 a 30000 uloží do řetězců program vytvářející dis-

play list a rutinu rolování. Na řádku 30 do naší rozšířené VRAM-ky pouze vložíme několik vertikálních řádků, abychom měli čím rolovat. Na řádku 40 je pomocí DLSCROLL$ sestaven nový display list; této rutině je také předána adresa řetězce SCROLL$, takže může být vložena do snímkového zatemnění (VBI). Protože se nechystáme dělat nic jiného, pouze pozorovat rolováni, je na řádku 50 smyčka, které udržuje program v chodu, zatímco program přerušení snímkovým zatemněním (naše rolovací rutina) si dělá své. Pokud se budete na tento program dívat příliš dlouho, neneseme odpovědnost za vaše činy - je to totiž hypnotické!

Tímto uzavíráme náš přehled display listu, zobrazovací paměti (VRAM), ošetření přerušení a jemného rolováni. Nyní byste měli být schopni vytvořit celkem složité a náročné rutiny v assembleru a bez větších potíži je použít ve vašich BASICovských programech.

KAPITOLA DEVÁTÁ

CENTRÁLNÍ V/V SYSTÉM V ATARI POČÍTAČÍCH

V kterémkoliv počítačovém systému pojmy VSTUP a VÝSTUP označují komunikaci mezi mikroprocesorem a libovolným vnějším zařízením – klávesnicí, obrazovkovým editorem, tiskárnou, magneťákem, disketovou jednotkou apod. OS ATARI, tak jak mnoho dalších počítačů, obsahuje rutiny, pro komunikaci s libovolným z těchto zařízení na několika úrovních. To, co činí ATARI systém tak jednoduchým (z hlediska programátorského použití) a zároveň unikátním, je fakt, že všechna vnější zařízení jsou obsluhována týmž způsobem. Odlišena jsou pouze nepatrnými změnami ve V/V rutinách.

Vstupem rozumíme informaci mikroprocesoru z vnějšího světa, např. klávesnice. Výstupem rozumíme činnost opačnou, jelikož informace vystupuje z počítače ven, např. na tiskárnu. V dalším pokračování budeme centrální V/V systém označovat CIO (zkratka je dána anglickými slovy Central Input-Output).

Vektory v počítačích ATARI

Již dříve jsme se zmínili, že techniky a rutiny používané v této knize budou „chodit“ na všech druzích ATARI, protože firma ATARI se zaručila, že vektory rutin v OS se nezmění. V OS vašeho ATARI je tzv. TABULKA ODSKOKŮ, jež obsahuje adresy všech klíčových rutin nezbytných k programování v assembleru. Tabulka je uložena na paměťových lokacích od adresy $E450 po $E47D v počítači ATARI 800 s rezivizí B vypadá takto:

```
Adresa  Obsahuje instrukci
--------------------------
F450    JMP $EDEA
E453    JMP $EDF0
E456    JMP $E4C4
E459    JMP $E959
E45C    JMP $E8ED
E45F    JMP $E7AE
E462    JMP $E905
E465    JMP $E944
```

```
E468    JMP $EBF2
E46B    JMP $E6D5
E46E    JMP $E4A6
E471    JMP $F223
E474    JMP $F11B
E477    JMP $F125
E47A    JMP $EFE9
E47D    JMP $EF5D
```

Tabulka sestává z adres, kam je předáno řízení programu, pokud se odvoláme na příslušnou instrukci skoku. Můžete se však zeptat, proč tam neskočit přímo. Odpovědí je právě možnost psát programy, která budou fungovat na všech počítačích ATARI. Představme si, že místo skoku na $E456 zvoláme přímo skok na $E4C4. Všechno je v pořádku a program běží. ALE, nyní si představme, že ATARI vyrobí nějaký nový počítač, řekněme 24800 XLTUB, a bude nezbytné provést nějaká změny v OS tohoto úžasného stroje. Dostáváme se do těžkostí. Firma ATARI se totiž nikdy nezaručila, že buňka $B4C4 zůstane navždy stejná; zaručila se pouze, že tabulka odskoků bude vždy ukazovat na správné adresy. Podívejme se nyní na tabulku vektorů s označením, jež používá firma ATARI (a i my ho budeme v dalším textu používat), a s použitím:

Zkratka	Adresa	Použití
DISKIV	$E450	Disk handler initiation routine
DSKINV	$E453	Disk handler vector
CIOV	$E456	Central Input/Output vector
SIOV	$E459	Serial Input/Output vector
SETVBV	$E45C	Set system timers routine vector
SYSVBV	$E45F	System vertical blank interrupt processing
XITVBV	$E462	Exit from vertical blank processing
SIOINV	$E465	Serial Input/Output initialization
SENDEV	$E468	Serial bus send enable routine
INTINV	$E46B	Interrupt handler routine
CIOINV	$E46E	Central Input/Output initialization
BLKBDV*	$E471	Blackboard mode vector to memopad mode
WARMSV	$E474	Warm start entry (follows SYSTEM RESET)
COLDSV	$E477	Cold start entry point (follows power-up)
RBLOKV	$E47A	Cassette read block routine vector
CSOPIV	$E47D	Cassette open for input vector

Značka “*” říká, že vektory se v současnosti nepoužívají. Všimněte si, že řada vektorů ukazuje na totéž místo v OS, $E783, což je adresa centrální rutiny přerušení. Ta určuje

charakter přerušení a posílá program na příslušná místa v OS, kde se nachází rutiny, které ošetří dané přerušení.

Tyto vektory nejsou na rozdíl od ROM vektorů uspořádány do tabulky, takže se na ně nelze odvolat pomocí JSR. Avšak zase ukazují na rutiny v OS, které jsou zakončeny instrukcí RTS, takže bychom je rádi vyvolali užitím instrukce JSR. Nejvhodnější je metoda, kdy nastavíme JSR na paměťovou lokaci, ze které nepřímo skočíme na vektor. Příklad:

```
40   JSR LOKACE
45 .
50 .
55 .
60 .
70 LOKACE JMP (DOSINI)
```

Vzhledem k řádku 45, instrukce RTS na konci rutiny vrátí řízení na řádek 45, odkud bude program pokračovat.

Nyní se budeme trochu zabývat prací CIO a naučíme se psát programy, které přicházejí do styku s reálným světem.

V/V řídicí blok (IOCB)

CIO systém má dvě části: V/V řídicí blok (IOCB) a OBSLUŽNOU TABULKU. Postupně se jimi budeme zabývat.

IOCB je část paměti na stránce 3, která obsahuje informace dané programátorem. Tyto se týkají požadovaného zařízení a informace, jež mu (zařízení) má být předána. Každý IOCB vyžaduje 16 bytů. Celkem je k dispozici osm IOCBů. Názvy a lokace jsou uvedeny v následující tabulce:

Název	Lokace
IOCB 0	$340 - $34F
IOCB 1	$350 - $35F
IOCB 2	$360 - $36F
IOCB 3	$370 - $37F
IOCB 4	$380 - $38F
IOCB 5	$390 - $39F
IOCB 6	$3A0 - $3AF
IOCB 7	$3B0 - $3BF

Tři z těchto bloků jsou systémem předdefinovány, ačkoliv programátorovi tím není bráněno, aby si je upravil pro vlastní účely. Ovšem obvykle se k tomu využívá zbývajících pěti. OS používá těchto tří:

1. IOCB 0 – obrazovkový editor. Řízením výstupu na IOCB 0 předáváme informaci obrazovkovému editoru. Tento řídicí blok 0 také ovládá textové okénko v libovolném grafickém režimu s textovým okénkem.

2. IOCB 6 – pro zobrazování na obrazovce v grafických modech větších než 0. Tento řídicí blok se používá pro všechny grafické příkazy jako PLOT, DRAWTO, FILL aj.

3. IOCB 7 – pro zpracování příkazu LPRINT v BASICu, jenž řídí výstup na tiskárnu. V praxi se ovšem pro výstup tiskárnu daleko častěji používá jiných bloků, protože tak máme lepší možnosti, např. formátování.

Jak jste si zajisté již povšimli, BASIC používá čísel IOCB (0, 6 a 7) pro řízení výstupu na daná zařízení, např. při tištěni v GR.1 nebo 2… PRINT 6; „HELLO“.

16 bytů každého IOCB popisuje nasledující tabulka:

Návěstí	Číslo bytu	Délka	Popis
ICHID	0	1	Index into device name table for this IOCB
ICDNO	1	1	Device number
ICCOM	2	1	Command byte: determines action to be taken
ICSTA	3	1	Status returned by device
ICBAL/H	4,5	2	Two-byte buffer address of stored information
OCPTL/H	6,7	2	Address-1 of devices put character routine
ICBLL/H	8,9	2	Buffer length
ICAX1	10	1	First auxiliary byte
ICAX2	11	1	Second auxiliary byte
ICAX3/4	12,13	2	Auxil. bytes 3-4 for BASIC NOTE and POINT
ICAX5	14	1	Fifth auxil. byte - for NOTE and POINT also
ICAX6	15	1	Spare auxiliary byte - unused at present

Příklad jednoduchého V/V s použitím IOCB

Dřív, než se budeme zabývat detaily různých bytů, jež vyžaduje každá možná funkce bloku, uveďme si vzorový program, který poslouží k lepšímu porozumění. Vezměme si jednoduchý příkaz v BASICu a převeďme ho do assembleru.

```
CLOSE #4:OPEN #4,6,0,“D:*.*”
```

Pro tuto chvíli potřebujeme vědět, že příkaz CLOSE vyžaduje $C v ICCOM a příkaz OPEN 3. Pro otevření disku musí v ICAX 1 být 6. Nyní program v assembleru.

```
                0100 ; ..........
                0110 ;
                0120 ; ..........
0000            0130       * = $0600
0341            0140 ICDNO   = $0341
0342            0150 ICCOM   = $0342
0344            0160 ICBAL   = $0344
0345            0170 ICBAH   = $0345
034A            0180 ICAX1   = $034A
E456            0190 CIOV    = $E456
                0200 ; ..........
                0210 ;
                0220 ; ..........
0600 A240       0230         LDX #$40
0602 A90C       0240         LDA #$C
0604 9D4203     0250         STA ICCOM,X
0607 2056E4     0260         JSR CIOV
                0270 ; ..........
                0280 ;
                0290 ; ..........
060A A240       0300         LDX #$40
060C A901       0310         LDA #1
060E 9D4103     0320         STA ICDNO,X
0611 A903       0330         LDA #3
0613 9D4203     0340         STA ICCOM,X
0616 A906       0350         LDA #6
0618 9D4A03     0360         STA ICAX1,X
061B A929       0370         LDA #FILE & 255
061D 9D4403     0380         STA ICBAL,X
0620 A906       0390         LDA #FILE/256
0622 9D4503     0400         STA ICBAH,X
0625 2056E4     0410         JSR CIOV
0628 60         0420         RTS
                0430 ; ..........
                0440 ;
                0450 ; ..........
0629 44         0460 FILE .BYTE $44,$3A,$2A,$2E,$2A,$9B
062A 3A
062B 2A
062C 2E
062D 2A
062E 9B
```

V obou částech programu, jak pro CLOSE tak po OPEN, ukládáme do registru X hodnotu $40, která označuje IOCB 4 (pro IOCB 6 by to bylo $60). Pak uložíme příkazový byte $C do ICCOM pro IOCB a provedeme skok do podprogramu na adrese CIOV, kde se uskuteční požadované CLOSE. Je vždy dobré před otevřením nějakého kanálu tento kanál nejprve zavřít, abychom tak předešli případnému otevření již otevřeného kanálu. Při návratu z CIO by totiž došlo k hlášení chyby. Zjišťovat, zda došlo k chybě při JSR CIOV, lze jednoduchým testováním znaménkového bitu ve stavovém registru procesoru BMI ERROR. Pro naše účely předpokládejme, že vše je O.K. Tento předpoklad byste si nikdy neměli dovolit při vlastních programech. Otevření souboru (OPEN…) provedeme vložením 1 do ICDNO čtvrtého bloku (chápej, každý blok má svůj ICDNO), vložením příkazového bytu 3 do ICCOM a 6 do ICAX 1. Vše, co nám zbývá provést dřív, než zavoláme CIO, je nastavit adresu bafru na název souboru, který chceme otevřít. Tento název je na řádku 460 a má návěstí LABEL. Hexadecimální kód $9B na konci názvu je kódem návratu vozíku. Měl by být na konci každého názvu souboru či zařízení (S: nebo P: aj.).

Abychom mohli bafr nastavit, potřebujeme převést jeho adresu do struktury vyšší byte a nižší byte. Tyto pak uložíme do ICBAL a ICBAH a pak zavoláme CIO.

Tento jednoduchý příklad ukazuje nejen jak otevřít disk, ale také jak se provádí volání CIO na ATARI. Nejprve nastavíme příslušné byty v IOCB a pak jednoduše provedeme JSR CIOV, ať už se jedná o OPEN či CLOSE, ale i čtení informace z disku, pásku či obrazovky. Všechny tyto operace se na ATARI uskutečňují provedením posloupnosti týchž úkonů, což činí V/V tak jednoduchým (pokud jste jednou porozuměli). Všimněte si, že pokud chcete volat CIO, není nutné nastavit všech 16 bytů. Uvidíme, že některé operace vyžadují nastavení pouze jednoho či dvou bytů.

Detailní rozbor bytů řídicího bloku IOCB

Nyní si probereme celé spektrum informací, jež musíme mít, pokud chceme implementovat libovolnou V/V funkci. Probereme si jednotlivé byty v IOCB tak, jak jdou za sebou.

První byte je ICHID. Nese informaci, která označuje index zařízení, s nímž má IOCB co do činění. Je nastaven OS-em a vy ho nepotřebujete nastavovat pro žádnou operaci.

Druhý byte je ICDNO. Označuje číslo zařízení a nejčastěji se používá při práci s disketovou jednotkou, abychom uvedli, se kterým drajvem si přejeme pracovat. Práce s diskodrajvem 1 si vyžaduje odlišný IOCB než práce s diskem 2.

Třetí byte je ICCOM - příkazový byte. Přehledně byl zpracován do následující tabulky:

Příkaz	Byte	Popis
Open	3	Otevírá soubor
Get record	5	Čte záznam
Get character	7	Čte znak
Put record	9	Ukládá záznam
Put character	11	Ukládá znak
Close	12	Zavírá soubor
Status	13	
Draw line	17	
Fill command	18	
Format disk	254	

Čtvrtý byte je ICSTA. Ten je nastaven OS-em po návratu z CIO. Informace se vždy přepisuje z registru Y, takže váš program může číst buď tento registr nebo ICSTA pro určení, zda V/V operace byla úspěšná či nikoliv. Libovolný záporný status (číslo větší než 127 nebo $7F) říká, že došlo k chybě.

Následující dva byty slouží jako ukazovátko bafru používaného pro vstup a výstup a jsou v obvyklém uspořádání nižší/vyšší. Nazývají se ICBAL a ICBAH. Bafrem rozumíme část paměti, která obsahuje informace, které si přejete poslat na výstup. Do bafru lze také umístit informace, která vstupují. Kupříkladu si přejete poslat text na tiskárnu. ICBAL a ICBAH jsou nastaveny tak, aby ukazovaly na tu část paměti, kde já uložen váš text. Podobně je tomu, pokud chcete uložit informace z disku do paměti.

ICPTL a ICPTH fungují jako další dvoubytové ukazovátko. V tomto případě však ukazují na adresu rutiny „ulož byte“ zařízení, - 1. Každé zařízení, které lze otevřít pro výstup, musí mít za tímto účelem rutinu „ulož byte“, která počítači říká, jak zařízení poslat informaci. Podrobnější informace uvedeme při vysvětlování OBSLUŽNÉ TABULKY.

Další dva byty řídicího bloku, jsou ICBLL a ICBLH, které obsahují délku V/V bafru v bytech. Jak uvidíme, existuje speciální případ V/V, ve kterém tím, že oba registry nastavíme

na nulu, je i celková délka bafru nulová. V tomto případě se jedná o přesun informace z/do akumulátoru a ne z/do paměti.

Protože na mnoha k ATARI připojitelných zařízeních lze nadefinovat více funkcí, uživatel musí být schopný říct řídicímu bloku, kterou z nich požaduje. K tomu slouží další byte IOCB, ICAX 1. V následující tabulce jsou použity zkratky:

TW – označuje oddělené textové okénko, např. takové, jaké vznikne po zadání příkazu GR.3;
RE – umožněna operace čtení (Read Enable) z obrazovky
RD – operace čtení není povolena (Read Disable)

Zařízení	ICAX1 Byte	Funkce
Obrazovkový editor	9	Výstup na obrazovku
	12	
	13	
Obrazovkový display	8	
	12	
	24	
	29	
Obrazovkový display	40	
	44	
	56	
	60	
Keyboard	4	
Printer	8	
Tape recorder	4	
	8	
RS-232 port	5	
	8	
	9	
	13	
Disk drive	4	
	6	
	8	
	9	
	12	

Poslední byte řídicího bloku, kterým se budeme zaobírat, je ICAX 2, druhý pomocný byte. Používá se pouze ve speciálních případech, jinak je nastaven na nulu. Pokud je v tomto bytu při práci s kazetovým magnetofonem hodnota 128, jsou mezi bloky informací na pásku použity krátké mezinahrávací mezery, které umožňují rychlejší nahrávání. Hodnota 0 způsobí normální, delší mezinahrávací mezery. Pokud do ICAX 2 uložíme

při práci s tiskárnou ATARI 820 hodnotu 83, bude tiskárna tisknout na šířku papíru místo normálně. Hodnota 70 umožňuje normální znaky, 8? dvojnásobné šířky.

A konečně grafická režimy 0 až 11 jsou v příkaze OPEN specifikovány tím, že se do ICAX 2 uloží požadovaná hodnota režimu. V kombinaci s ICAX 1 dává ICAX 2 programátorovi v assembleru plnou kontrolu nad grafickými režimy, textovými okénky, mazáním obrazovky a funkcemi čtení a zápisu z/do obrazovky.

Obslužná tabulka

Nyní, když jsme si objasnili různé části IOCB-ů, ve stručnosti popíšeme tzv. obslužnou tabulku (handler table) a způsob, jakým spolu s IOCB-y vytváří V/V systém s CIO. Potom si uvedeme několik názorných příkladů, které vám ukáží, jak těchto informací použít k uskutečnění mnoha různých typů V/V operací v assembleru. Nejednodušším způsobem, jak porozumět obslužné tabulce, je podívat se na krátký program v assembleru:

```
0100 PRINTV = $E430
0110 CASETV = $E440
0120 EDITRV = $E400
0130 SCRENV = $E410
0140 KEYBDV = $E420
0150 ; ..........
0160 ; začátek HATABS = $031A
0170 ; ..........
0180 *= $031A
0190 .BYTE „P“
0200 .WORD PRINTV
0210 .BYTE „C“
0220 .WORD CASETV
0230 .BYTE „E“
0240 .WORD EDITRV
0250 .BYTE „S“
0260 .WORD SCRENV
0270 .BYTE „K“
0280 .WORD KEYBDV
0290 .BYTE 0
0300 .WORD 0,0 .
O310 .BYTE 0
0320 .WORD 0,0
0330 .BYTE 0
```

```
0340 .WORD 0,0
0350 .BYTE 0
0360 .BYTE 0,0
0370 .BYTE 0
0380 .WORD 0,0
0390 .BYTE 0
0400 .WORD 0,0
0410 .BYTE 0
0420 .WORD 0,0
```

Každá položka v obslužné tabulce sestává z prvního písmene daného zařízení a vektoru, který ukazuje na místo v paměti, kde jé uložena informace potřebná k práci s daným zařízením. Jak lze odpozorovat, je sedm položek v tabulce nepoužitých. Programátor má možnost nadefinovat si zařízení potřebná k libovolným účelům, a ta pak budou ošetřena jako zařízení již nadefinovaná. Je třeba si uvědomit jeden důležitý fakt. Kdykoliv se OS obrací na obslužnou tabulku, aby zjistil, kam se má do paměti podívat, pročítá ji zdola nahoru! Je to tak schválně proto, aby když si uživatel nadefinuje vlastní rutiny, např. pro práci s tiskárnou, bude vektor na ně ukazující nalezen OS-em dřív, než původní předdefinovaný. Jednoduše tedy vložíme „P“ do jednoho z nižších bytů položky následované dvoubytovým vektorem adresujícím vaši novou obslužnou rutinu.

Nyní se krátce podívejme na tabulku, na kterou ukazují položky v obslužné tabulce. Např. vektor PRINTV ukazuje na druhou tabulku, tzv. tabulku obsluhy tiskárny (printer handler entry point table). Tabulky všech zařízení mají tutéž organizaci. Obsahují adresu rutiny zmenšenou o 1 v následujícím uspořádání:

```
rutina OPEN zařízení
       CLOSE
       READ
       WRITE
       STATUS
       SPECIÁLNÍ funkce
```

Tabulka obsluhy daného zařízení je vždy ukončena tříbytovou instrukcí JMP, která.ukazuje na inicializační rutinu tohoto zařízení. POZOR tedy: adresy v obslužné tabulce neukazují přesně na rutiny OPEN či CLOSE, ale na byte o jeden menší než je její začátek. To je samozřejmě zapotřebí mít na paměti při sestavováni vlastní obslužná tabulky!

Jednoduchá V/V rutina

Ukažme si, jak lze použít CIO pro jednoduchou funkci – zápis na obrazovku. Víme, že pokud chceme něco napsat na obrazovku pomocí BASICu, stačí jednoduchý příkaz jako

PRINT „A SUCCESSFUL WRITE!“

Nyní, když víme, jak použít IOCB a CIO, to nebude problém ani v assembleru. Jen pro zopakování: nemusíme otevírat obrazovku jako zařízení, protože IOCB 0 je pro tento účel operačním systémem předurčen. Proto můžeme do registru X uložit 0 a použít ho jako označení IOCB. Jiná možnost je prostě použit absolutní adresu, protože budeme používat první IOCB. V níže uvedené rutině jsme použili z pedagogických důvodů první variantu, abyste si ozřejmili normální proceduru vložení informací do IOCB.

```
                0100 ; ..........
                0110 ;
                0120 ; ..........
0340            0130 ICHID = $0340
0341            0140 ICDNO = $0341
0342            0150 ICCOM = $0342
0343            0160 ICSTA = $0343
0344            0170 ICBAL = $0344
0345            0180 ICBAH = $0345
0346            0190 ICPTL = $0346
0347            0200 ICPTH = $0347
0348            0210 ICBLL = $0348
0349            0220 ICBLH = $0349
034A            0230 ICAX1 = $034A
034B            0240 ICAX2 = $034B
E456            0250 CIOV  = $E456
0000            0260     * = $0600
                0270 ; ..........
                0280 ;
                0290 ; ..........
0600 A200       0300       LDX #0
0602 A909       0310       LDA #9
0604 9D4203     0320       STA ICCOM,X
0607 A91F       0330       LDA #MSG & 255
0609 9D4403     0340       STA ICBAL,X
060C A906       0350       LDA #MSG/256
060E 9D4503     0360       STA ICBAH,X
0611 A900       0370       LDA #0
0613 9D4903     0380       STA ICBLH,X
```

```
0616 A9FF   0390       LDA #$FF
0619 9B4803 0400       STA ICBLL,X
            0410 ; ..........
            0420 ; nyní vlož do obrazovky
            0430 ; ..........
061B 2056E4 0440       JSR CIOV
061E 60     0450       RTS
            0460 ; ...........
            0470 ;
            0480 ; ...........
061F 41     0490 MSG   .BYTE „A SUCCESSFUL WRITE!‘“,$9B
0620 20
0621 53
0622 55
0623 43
0624 43
0625 45
0626 53
0627 53
0628 46
0629 55
062A 4C
062B 20
062C 57
062D 52
062E 49
062F 54
0630 45
0631 21
0632 9B
```

Samozřejmě, že zápis na obrazovku je v BASICu tak jednoduchý, že není důvod psát tuto rutinu jako podprogram pro program v BASICu. Proto byla také vynechána obvyklá instrukce PLA. Protože však budete potřebovat psát na obrazovku při odlaďování programů v assembleru, může se tato rutina stát jednou z nejužívanějších.

Jestliže chcete tuto rutinu odzkoušet poté, co jste ji naťukali, jednoduše zadejte ASM a po přeložení pak BUG, abyste se dostali do režimu DEBUG kartridže ASS/EDITOR. Potom zadejte G600; tím ji spustíte od adresy $600. Jestliže jste program naťukali správně, objeví se nápis „A SUCCESSFUL WRITE!“ následovaný výpisem obsahů registrů procesoru 6502. Ty se totiž objevují po každá rutině, jež pracuje s kartridží. Tento postup byste měli používat při zkoušení každé

rutiny uvedené v této knize. Jestliže vyvstane nějaký problém, překontrolujte program, který jste naťukali.

PROSÍM POZOR! NAHREJTE SI SVÉ PROGRAMY JEŠTĚ PŘEDTÍM, NEŽ JE SPUSTÍTE! JESTLIŽE SE POTOM ZHROUTÍ BUĎ PROGRAM NEBO CELÝ SYSTÉM, NEBUDETE MUSET VÁŠ PROGRAM PSÁT CELÝ ZNOVU!

V uvedené rutině zapisujeme vzkaz na obrazovku použitím příkazu PUT RECORD, do ICCOM tedy uložíme 9. Adresa vzkazu je pak nám již známým způsobem uložena do ICBAL a ICBAH. Do vyššího bytu délky vzkazu uložíme nulu, do nižšího pak $FF. Proč ale $FF, když délka vzkazu je 20 bytů? Totiž pokud použijeme CIO v režimu PUT RECORD, záznam je vypisován byte po bytu, dokud není buď dosaženo délky záznamu nastavené v ICBLL a ICBLH nebo dokud není v záznamu zaznamenáno RETURN. Všimněte si, že náš vzkaz začínající na řádku 490 je ukončen bytem $9B, což je právě RETURN. Proto bude na obrazovku poslán nejprve vzkaz, pak návrát vozíku a tím bude rutina ukončena. Nastavíme tedy délku vzkazu záměrně delší, než ve skutečnosti je, protože chceme, aby byl výstup ukončen RETURNem. Tímo způsobem nemůžeme udělat chybu v tom, že bychom nechtěně usekli vzkaz nevhodným nastavením ICBLL a ICBLH.

Stoji za povšimnuti, že jsme nenastavili všechny byty v IOCB. Nastavili jsme pouze ty, které naše rutina potřebovala. Jak uvidíte dále, tak je tomu vždy v případě centrálních ATARI rutin. Skutečného výstupu na obrazovku je dosaženo zavoláním centrální V/V rutiny na řádku 440 a RTS na dalším řádku vrací řízení kartridži. Kdyby tato rutina byla částí delšího programu v assembleru, pokračoval by pak od řádku 450 bez RTS.

Jiné formy V/V rutiny

Řekneme si, jak lze s využitím CIO psát na obrazovku jiným způsobem. Místo uložení do ICCOM čísla 9 (put record), uložíme 11, což znamená PUT BYTES. Ostatní byty jsou až na ICBLL, kde je skutečná délka vzkazu, nastaveny stejně jako prve. Opět nezapomeňte na konec vzkazu napsat $9B, RETURN, a započítat ho do počtu bytů. Program pak bude vypadat takto:

```
                0100 ; ..........
                0110 ;
                0120 ; ..........
0340            0130 ICHID = $0340
```

```
0341          0140 ICDNO = $0341
0342          0150 ICCOM = $0342
0343          0160 ICSTA = $0343
0344          0170 ICBAL = $0344
0345          0180 ICBAH = $0345
0346          0190 ICPTL = $0346
0347          0200 ICPTH = $0347
0348          0210 ICBLL = $0348
0349          0220 ICBLH = $0349
034A          0230 ICAX1 = $034A
034B          0240 ICAX2 = $034B
E456          0250 CIOV  = $E456
0000          0260     * = $0600
              0270 ; ..........
              0280 ;
              0290 ; ..........
0600 A200     0300       LDX #0
0602 A90B     0310       LDA #11
0604 9D4203   0320       STA ICCOM,X
0607 A91F     0330       LDA #MSG & 255
0609 9D4403   0340       STA ICBAL,X
060C A906     0350       LDA #MSG/256
060E 9D4503   0360       STA ICBAH,X
0611 A900     0370       LDA #0
0613 9D4903   0380       STA ICBLH,X
0616 A914     0390       LDA #20
0619 9B4803   0400       STA ICBLL,X
              0410 ; ..........
              0420 ; nyní vlož do obrazovky
              0430 ; ..........
061B 2056E4   0440       JSR CIOV
061E 60       0450       RTS
              0460 ; ...........
              0470 ;
              0480 ; ...........
061F 41       0490 MSG   .BYTE „A SUCCESSFUL WRITE!‘“,$9B
0620 20
0621 53
0622 55
0623 43
0624 43
0625 45
0626 53
0627 53
0628 46
0629 55
062A 4C
```

```
062B 20
062C 57
062D 52
062E 49
062F 54
0630 45
0631 21
0632 9B
```

Tento program vede k témuž cíli jako předchozí, ale činí tak jiným způsobem. Může nastat speciální případ, kdy nechceme, aby byl text ukončen návratem vozíku, třeba při tištění otazníku u příkazu INPUT, nebo v případě, že chceme nějakým způsobem formátovat obrazovku. V BASICu prostě stačí za příkazem PRINT „text“ uvést středník(y), který zabrání návratu vozíku. V assembleru by to vypadalo takto:

```
                0100 ; ..........
                0110 ;
                0120 ; ..........
0340            0130 ICHID = $0340
0341            0140 ICDNO = $0341
0342            0150 ICCOM = $0342
0343            0160 ICSTA = $0343
0344            0170 ICBAL = $0344
0345            0180 ICBAH = $0345
0346            0190 ICPTL = $0346
0347            0200 ICPTH = $0347
0348            0210 ICBLL = $0348
0349            0220 ICBLH = $0349
034A            0230 ICAX1 = $034A
034B            0240 ICAX2 = $034B
E456            0250 CIOV  = $E456
0000            0260     * = $0600
                0270 ; ..........
                0280 ;
                0290 ;
                0300 ; ..........
0600 A200       0310       LDX #0
0602 A90B       0320       LDA #11
0604 9D4203     0330       STA ICCOM,X
0607 A900       0340       LDA #0
0609 9D4903     0350       STA ICBLH,X
060C A900       0360       LDA #0
060E 9B4803     0370       STA ICBLL,X
                0380 ; ..........
                0390 ;
```

```
            0400 ; ..........
0611 A93E   0410       LDA #62
0613 2056E4 0440       JSR CIOV
0616 60     0450       RTS
```

Jestliže nastavíme délku bafru na 0 (čehož dosáhneme tím, že jak do ICBLL, tak do ICBLH uložíme 0), potom zavoláním CIOV bude na dané výstupní zařízení poslán znak obsažený v akumulátoru bez toho, že by následoval návrat vozíku. To platí pro všechna zařízení od disků, přes magnetofony, tiskárny až po obrazovku. Je právě velkou výhodou ATARI počítačů, že vstup a výstup jsou na zařízení nezávislé. Znamená to, že OS ošetřuje všechna zařízení obdobně, takže se nemusíme učit, jak psát text na obrazovku, potom jiný způsob, jak provést tutéž operaci s tiskárnou apod. Pro ilustraci si uveďme rutinu, jež posílá týž vzkaz na tiskárnu.

Výstup na tiskárnu

Nejprve zavřeme (CLOSE) IOCB 2, potom otevřeme (OPEN) tiskárnu jako zařízení užívající IOCB 2 a pak pošleme náš vzkaz.

```
            0100 ; ..........
            0110 ;
            0120 ; ..........
0340        0130 ICHID = $0340
0341        0140 ICDNO = $0341
0342        0150 ICCOM = $0342
0343        0160 ICSTA = $0343
0344        0170 ICBAL = $0344
0345        0180 ICBAH = $0345
0346        0190 ICPTL = $0346
0347        0200 ICPTH = $0347
0348        0210 ICBLL = $0348
0349        0220 ICBLH = $0349
034A        0230 ICAX1 = $034A
034B        0240 ICAX2 = $034B
E456        0250 CIOV  = $E456
0000        0260     * = $0600
            0270 ; ..........
            0280 ; zavri a otevri IOCB2
            0290 ; ..........
0600 A220   0300       LDX #$20
0602 A90C   0310       LDA #12
0604 9D4203 0320       STA ICCOM,X
0607 2056E4 0330       JSR CIOV
```

```
060A A220   0340       LDX #$20
060C A903   0350       LDA #3
060E 9D4203 0360       STA ICCOM,X
0611 A908   0370       LDA #8
0613 9D4A03 0380       STA ICAX1,X
0616 A94C   0390       LDA #NAM & 255
0618 9D4403 0400       STA ICBAL,X
061B A906   0410       LDA #NAM/256
061D 9D4503 0420       STA ICBAH,X
0620 A900   0430       LDA #0
0622 9D4903 0440       STA ICBLH,X
0625 A9FF   0450       LDA #$F
0627 9B4803 0460       STA ICBLL,X
062A 2056E4 0470       JSR CIOV
            0480 ; ..........
            0490 ;
            0500 ; ..........
062D A220   0510       LDX #$20
062F A909   0520       LDA #9
0631 9D4203 0530       STA ICCOM,X
0634 A91F   0540       LDA #MSG & 255
0636 9D4403 0550       STA ICBAL,X
0639 A906   0560       LDA #MSG/256
063B 9D4503 0570       STA ICBAH,X
063E A900   0580       LDA #0
0640 9D4903 0590       STA ICBLH,X
0643 A9FF   0600       LDA #$FF
0645 9B4803 0610       STA ICBLL,X
0648 2056E4 0620       JSR CIOV
064B 60     0630       RTS
064C 50     0640 NAM   .BYTE „P“,$9B
064D 3A
064E 9B
064F 41     0490 MSG   .BYTE „A SUCCESSFUL WRITE!‘“,$9B
0650 20
0651 53
0652 55
0653 43
0654 43
0655 45
0656 53
0657 53
0658 46
0659 55
065A 4C
065B 20
065C 57
```

```
065D 52
065E 49
065F 54
0660 45
0661 21
0662 9B
```

Samozřejmě, že pokud používáte ATARI tiskárnu a chcete tisknout rozšířeným tiskem, musíte dřív, než zavoláte CIOV, nastavit ICAX 1 a ICAX 2. Avšak to je triviální. Všimněte si, že po vyslání vzkazu jsme IOCB nezavřeli, takže kdybychom chtěli tisknout něco dalšího, jednoduše tak učiníme pomocí IOCB 2, aniž bychom jej znovu otvírali. Samozřejmě to ovšem znamená, že v dané chvíli nelze IOCB 2 použít pro cokoli jiného, jako disk apod. Jestliže chceme mít přístup k disku, buď použijeme jeden z dalších IOCBů, nebo IOCB 2 zavřeme a poté otevřeme pro práci s diskem.

Výstup na disk

Jediná změna oproti programu pro tiskárnu uvedenému výše je ta, že názvem zařízení bude diskový soubor, který si přejete otevřít. Proto je program až na řádek 640, kde bude něco jako

```
640 NAM.BYTE „D1:MYFILE1“,$9B
```

stejný. To by bylo k tomuto vše. Vidíte tu krásu používání identických CIO rutin pro všechna zařízení? Nyní byste měli být schopni vyslat v assembleru informace na libovolná zařízení dle libovůle.

Vstup s použitím CIOV

Metoda pro vstup z vnějšího zařízení do vašeho ATARI je přesně táž jako pro výstup, avšak zařízení musí být otevřeno pro vstup. Zpětně bychom opět mohli získat vzkaz z našeho diskového souboru „D1:MYFILE1“ tím, že otevřeme tento soubor pro vstup. To znamená 3 do ICCOM a 4 do ICAX 1, ICBAL a ICBAH nastavit na lokace v paměti, do kterých chceme vzkaz přemístit. Např. jestliže chceme, aby vzkaz začínal na adrese $680, nastavíme ICBAL na $80 a ICBAH na 6. Po zavoláni CIOV budou paměťové lokace $680 až $694 obsahovat byty vzkazu, který pak lze dále programově zpracovat.

Jednoduchost a snadnost V/V na ATARI by neměla zůstat nedoceněna. U mnoha mikropočítačů je pochopení práce s tím či oním zařízením vždy samostatnou kapitolou; vstup i výstup mohou používat různá rutiny, z nichž každá může mít svá specifika. Centrální V/V filosofie použitá v ATARI tento proces značně usnadňuje.

Ještě jednu závěrečnou poznámku k V/V rutinám. Kdybychom otevřeli jeden IOCB pro vstup ze souboru ne disku a druhý pro výstup na tiskárnu, úkol přemístit informace z jednoho zařízení na druhé by pak byl triviální – oba IOCBy bychom nastavili tak, aby ukazovaly na stejný bafr. (printing hard copy) z disku a kopírování paměti na něj je jednoduchá. Umíme dokonce přesunout informace z diskové jednotky na obrazovku nebo na pásek.

KAPITOLA DESÁTÁ

GRAFIKA

Jedním z nejvíc vzrušujících a nejjedinečnějších rysů počítačů ATARI je jejich vynikající grafika. Pokud ji budeme co do kvality srovnávat s jinými mikropočítači, je ATARI jasný vítěz. Vlastně většina pohybových her, jež jsou k dostání na více počítačů, vypadá nejlépe na ATARI a reklamní rubriky obvykle používají fotografií obrazovek generovaných ATARI počítačem.

Všechny grafické rutiny jsou umístěny v OS, a proto jsou dostupná pomocí kteréhokoliv jazyka. Nyní si ukážeme, jak těchto rutin použít v assembleru.

Každý program, který vyžaduje příkazy jako GR.n, PLOT nebo jiné grafické příkazy, ve všeobecnosti využívá těchto rutin několikrát v průběhu programu. Je proto nejjednodušší uvést tyto rutiny jako množinu podprogramů v assembleru, kterou lze zavolat z libovolnáho programu. Tyto rutiny lze uchovat na disk jako celek a pak pomocí ENTER připojit k programu, který to vyžaduje. Při práci s těmito rutinami budete většinou muset uložit do akumulátoru a registrů X a Y příslušná hodnoty a pak skočit (JSR) do příslušné rutiny. Parametry implementované v uvedených registrech jsou pro jednoduchost vysvětleny v komentářích ke každé rutině. Podrobný rozbor podprogramů bude následovat v části za výpisy programů.

Assemblerovské grafické podprogramy

```
                0100 ; ..........
                0110 ;
                0120 ; ..........
0340            0130 ICHID  = $0340
0341            0140 ICDNO  = $0341
0342            0150 ICCOM  = $0342
0343            0160 ICSTA  = $0343
0344            0170 ICBAL  = $0344
0345            0180 ICBAH  = $0345
0346            0190 ICPTL  = $0346
0347            0200 ICPTH  = $0347
0348            0210 ICBLL  = $0348
0349            0220 ICBLH  = $0349
```

```
034A        0230 ICAX1  = $034A
034B        0240 ICAX2  = $034B
E456        0250 CIOV   = $E456
            0260 ; ..........
            0270 ;
            0280 ; ..........
02C4        0290 COLOR0 = $02C4
0055        0300 COLCRS = $55
0054        0310 ROWCRS = $54
02FB        0320 ATRCHR = $02FB
00CC        0330 STORE1 = $CC
00CD        0340 STOCOL = $CD
0000        0350      * = $0600
            0360 ; ..........
            0370 ; rutina SETCOLOR
            0380 ; ...........
            0390
            0400
            0410
            0420
            0430
            0440
            0450 SETCOL
0600 0A     0460        ASL A
0601 0A     0470        ASL A
0602 0A     0460        ASL A
0603 0A     0490        ASL A
0604 85CC   0500        STA STORE1
0606 90     0510        TYA
0607 18     0520        CLC
0608 65CC   0530        ADC STORE1
060A 9DC402 0540        STA COLOR0,X
060D 60     0550        RTS
            0560 ; ..........
            0570 ;
            0580 ; ..........
            0590
            0600
            0610
            0620
            0630
            0640
            0650 COLOR
060E 85CD   0660        STA STOCOL
0610 60     0670        RTS
            0680 ; ...........
            0690 ;
```

```
            0700 ; ...........
            0710
            0720
            0730
            0740
            0750 GRAPHIC
0611 48     0760        PHA
0612 A260   0770        LDX #$60
0614 A90C   0780        LDA #$0C
0616 9D4203 0790        STA ICCOM,X
0619 2056B4 0800        JSR CIOV
061C A260   0810        LDX #$60
061B A903   0820        LDA #3
0620 9D4203 0830        STA ICCOM,X
0623 A9AD   0840        LDA #NAME & 255 ; jméno je „S“
0625 9D4403 0850        STA ICBAL,X     ; nižší byte
O628 A906   0860        LDA #NAME/256   ; vyšší byte
062A 934503 0870        STA ICBAH,X
062D 68     0880        PLA
062E 9D4B03 0890        STA ICAX2,X
0631 28F0   0900        AND #$F0
0633 4910   0910        EOR #$10
0635 090C   0920        ORA #$C
0637 9D4A03 0930        STA ICAX1,X
063A 2056E4 0940        JSR CIOV
063D 60     0950        RTS
            0960 ; ...........
            0970 ;
            0960 ; ...........
            0990
            1000
            1010
            1020
            1030
            1040
            1050 POSITN
063E 8655   1060        STX COLCRS
0640 8556   1070        STA COLCRS+1
0642 8454   1080 STY ROWCRS
0644 60     1090 RTS
            1100 ; ...........
            1110 ;
            1120 ; ...........
            1130
            1140
            1150 PLOT
0642 203E06 1160        JSR POSITN
```

```
0648 A260   1170        LDI #$60
064A A90B   1180        LDA #$B
064C 9D4203 1190        STA ICCOM,X
064F A900   1200        LDA #0
0651 9D4803 1210        STA ICBLL,X
0654 9D4903 1220        STA ICBLH,X
0657 A5CD   1230        LDA STOCOL
0659 2056E4 1240        JSR CIOV
065C 60     1250        RTS
            1260 ; ..........
            1270 ;
            1280 ; ..........
            1290
            1300
            1310 DRAWTO
065D 203E06 1320        JSR POSITN
0660 A5CD   1330        LDA STOCOL
0662 8DFB02 1340        STA ATACHR
0665 A260   1350        LDX #$60
0667 A911   1360        LDA #$11
0669 9D4203 1370        STA ICCOM,X
066C A90C   1380        LDA #$C
066B 9DHA03 1390        STA ICAX1,X
0671 A9C0   1400        LDA #0
0673 9D4B03 1410        STA ICAX2,X
0676 2056E4 1420        JSR CIOV
0679 60     1430        RTS
            1440 ; ..........
            1450 ;
            1460 ; ..........
            1470
            1480
            1490
            1500
067A 203B06 1510        JSR POSITN
067D A5CD   1520        LDA STOCOL
067F 8DFB02 1530        STA ATACHR
0682 A260   1540        LDX #$60
0684 A912   1550        LDA #$12
0686 9D4203 1560        STA ICCOM,X
0689 A90C   1570        LDA #$C
068B 9D4A03 1580        STA ICAX1,X
068E A900   1590        LDA #O
0690 9D4B03 1600        STA ICAX2,X
0693 2056E4 1610        JSR CIOV
0696 60     1620        RTS
            1630 ; ...........
```

```
             1640 ;
             1650 ; ...........
             1660
             1670
             1680
             1690
             1700 LOCATE
0697 203E06  1710        JSR POSITN
069A A260    1720        LDX #$60
069C A907    1730        LDA #7
069E 9D4203  1740        STA ICCOM,X
06A1 A900    1750        LDA #0
06A3 9D4803  1760        STA ICBLL,X
06A6 9D4903  1770        STA ICBLH,X
06A9 2056B4  1780        JSR CIOV
06AC 60      1790        RTS
             1800 ; ...........
             1810 ;
             1820 ; ...........
06AD 53      1830 NAME   .BYTE „S:“,$9B
06AE 3A
06AF 9B
```

Rozbor grafických podprogramů

První věc, o které je třeba se zmínit, je, že v těchto rutinách jsou použita standardní CIO označení, která jsme už viděli předtím, plus 6 nových. Nepotřebujeme celou množinu nových označení, protože používáme standardní ATARI CIO rutiny. Z těch šesti nových jsou dvě pouze uchovávací lokace: STOCOL k uchování informace COLOR a STORE1 k uchování dočasně potřebné informace. Jsou umístěny na lokacích $CD a $CC, ovšem vy si je můžete umístit na libovolné bezpečné místo. Např. by to mohly být poslední dvě buňky zásobníku $100 a $101. Další z nových označení je COLOR0, což je první z pěti lokací nesoucích informaci o barvě, nacházejících se na adresách 706–712 dec. Další je COLCRS, dvoubytová lokace nesoucí informaci o sloupci, ve kterém se právě nachází kurzor. Protože v GR.8 existuje 320 horizontálních pozic a v jednom bytu lze uchovat nanejvýš 256 různých hodnot, potřebujeme dva byty na adresách $55 a $56. Avšak ve všech ostatních grafických režimech je zřejmé, že v $56 bude vždy 0. Třetím novým označením je ROWCRS, lokace která nese informaci o vertikální pozici kurzoru. V žádném grafickém režimu není více než 192 možných pozic, takže jeden byte stačí. Posledním novým pojmem je ATACHR, lokace $2FB, která nese

informaci o barvě čáry kreslené jak pomocí rutiny FILL, tak DRAWTO.

První rutina je assemblerovský ekvivalent BASICového příkazu SETCOLOR. Víme, že standardní tvar příkazu je:

SE. barvicí registr,odstín,jas

V assembleru musíme nejprve do registrů 6502-ky uložit ekvivalentní informace. Rutina, která bude simulovat konkrétní příkaz SE.2,4,10, bude vypadat následovně:

```
LDX #2
LDY #10
LDA #4
JSR SETCOL
```

Abychom vykonali příkaz SETCOLOR, musíme sečíst jas s 16násobkem odstínu a výsledek uložit do příslušného barvicího registru. K získání šestnáctinásobku odstínu jednoduše použijeme čtyřikrát instrukci ASL A. Jelikož každé posunutí bitů zdvojnásobí hodnotu bytu, bude výsledná hodnota šestnáctkrát původní. Po násobení si výsledek odložíme do jedné z lokací dočasné informace a pomocí TYA přesuneme do akumulátoru hodnotu jasu. Vynulujeme carry, jak je to obvyklé před sčítáním, a obsah akumulátoru sečteme s výsledkem předchozího násobení. Nakonec použijeme hodnotu v registru X jako index pro jeden z barvících registrů.

Další rutina, příkaz COLOR, je daleko nejjednodušší ze všech rutin. BASICovský příkaz COLOR 3 má v aaembleru ekvivalent

```
LDA #3
JSR COLOR
```

Rutina jednoduše ukládá na lokaci STOCOL námi vybranou barvu, takže je pak k dispozici pro jiné rutiny.

Podobně implementován příkaz GRAPHICS:

```
LDA #23
JSR GRAPHIC
```

V první řadě si musíme odložit požadovaný grafický mód. Mohli bychom použít STORE1, ale uložení do zásobníku je rychlejší (v tomto případě); nemusíme provádět operace sčí-

tání a násobení jako u SETCOLOR. V dalších čtyřech řádcích jednoduše zavřeme obrazovku jako periferii. To jen pro jistotu. I kdyby už byla zavřena, neudělali jsme nic špatného. Všimněte si, že prostým použitím IOCB 6 (uložením do registru X $60) specifikujete obrazovku. Využíváme předdefinovaný kanál ATARI firmou. Zbytek rutiny už pouze otevírá obrazovku v příslušném grafickém režimu. Název obrazovky je S: (screen); ukládáme adresu názvu do ICBAL a ICBAH. Grafický režim je potom vybrán ze zásobníku a jeho hodnota uložena do druhého pomocného bytu. Jediné důležité bity v ICAX2 jsou spodní 4 bity, které určují samotný grafický režim, v tomto případě GR.7. Horní 4 bity řídí mazání obrazovky, přítomnost textového okénka atd., jak je to popsáno v osmé kapitole. Tyto bity odmaskujeme instrukcí AND $F0. OS vyžaduje, aby byl nejvyšší bit invertován – to zabezpečíme instrukcí EOR $10. [předešlá věta je imho blábol] Nakonec nastavíme spodní 4 bity na $C, což dovoluje jak zápis do obrazovky, tak čtení z ní. Celý byte uložíme do ICAX1. Zavoláme CIO a grafický režim je nastaven.

Jak již bylo probráno, příkaz POSITION pro grafický režim 8 vyžaduje 320 pozic na ose X. Abychom tedy simulovali příkaz POSITION 285,73, uložíme do registru X nižší byte X-ové souřadnice, vyšší byte do akumulátoru a Y-ovou souřadnici do registru Y:

```
25 LDX #30
30 LDA #1
35 LDY #73
40 JSR POSITN
```

Vlastní rutina pak jednoduše uloží informaci do příslušných lokací. X-ovou souřadnici do COLCRS a COLCRS+1, Y-ovou do ROWCRS.

Příkaz PLOT, řekněme PLOT 258,13, zapíšeme pomocí assembleru následovně:

```
25 LDX #3
30 LDA #1
35 LDY #13
40 JSR PLOT
```

Platí zde stejná dohoda jako u příkazu POSITION. Rutina PLOT vlastně začíná odskokem do POSITN. Protože chceme výstup na obrazovku, použijeme IOCB 6 a příkazový byte $B pro PUT RECORD (ulož záznam). V tomto případě požadujeme výstup

jednoho bytu, takže nastavíme délku bafru na 0. Potom do akumulátoru vložíme informaci COLOR, kterou chceme PLOTnout, a zavoláme CIOV.

Rutiny DRAWTO a FILL jsou tak podobné, že budou probrány spolu. Pro simulaci řekněme DR.42,80 použijeme posloupnost instrukcí:

```
25 LDX #42
30 LDA #0
35 LDY #80
40 JSR DRAWTO
```

Pokud chceme FILL, změníme řádek 40 na JSR FILL.

Rutina opět začíná odskokem do POSITN. Poté je do ATACHR uložena informace o barvě a opět použit IOCB 6 s tím, že pro DRAWTO je v ICCOM $11 a pro FILL $12. V ICAX1 musí být $C a před zavoláním CIOV vynulujeme ICAX2. Rutiny jsou přesnou analogií BASICovských příkazů XIO, jež vedou ke stejnému cíli. Např. pro nakreslení čáry použijeme příkaz XIO 17,#6,12,0,"S:". Příkaz si rozebereme: 17 je $11 v příkazovém bytu; 6 je číslo IOCB; 12 je uložena v ICAX1; 0 v ICAX2; „S:" je název zařízení. Úplně stejný příkaz lze použít pro FILL (zabarvení plochy) jednoduchou změnou 17 na 18 ($12).

Poslední rutina je vlastně identická s příkazem PLOT. Jedná se o příkaz LOCATE a rozdíl je v tom, že místo PUT RECORD používá GET RECORD. Konkrétní příkaz LOC.10,12,A přepíšeme do assembleru následovně:

```
25 LDX #10
30 LDA #0
35 LDY #12
40 JSR LOCATE
```

V tomto případě bude v akumulátoru hodnota barvy (popřípadě ATASCII znaku) nalezena na souřadnicích 10,12, takže instrukcí STA by šlo tuto informaci uchovat nebo ji lze okamžitě porovnat s nějakou požadovanou hodnotou.

Tímto uzavíráme rozbor assemblerovských protějšků BASICovských příkazů. Jejich použití je vlastně stejně snadné jako v BASICu. Protože však jak BASIC, tak assembler používají týchž rutin OS, neočekávejte, že assemblerovské rutiny budou kdovíjak rychlejší. Abyste dosáhli až takového zrychleni jako u rutin uvedených v dřívějších kapitolách, museli

byste si vymyslet vlastni rutiny, založené na zcela odlišných principech. Taková rutiny (daleko rychlejší než rutiny OS) již byly napsány, ale nejsou veřejně přístupné.

Pamatujte si, že vše, co je možné v BASICu, je proveditelné i v assembleru. Často uváděným příkladem je animace pomoci rotace barvových registrů:

```
15 LDA $708
20 STA STOCOL
25 LDA $709
30 STA $706
35 LDA $710
40 STA $709
45 LDA $711
50 STA $710
55 LDA $712
60 STA $711
65 LDA STOCOL
70 STA $712
```

Nyní, kdy umíte pomocí assembleru nakreslit velice podrobné obrázky, lze tento trik použít k animaci obrázku prakticky bez zpomaleni programu. Pomocí rotace barvových registrů lze vyvolat dojem pohybu.

PMG v assembleru

Další vymožeností počítačů ATARI je tzv. PMG grafika, PMG (Player–Missile Graphics). Uvedli jsme si již příklad, ve kterém jsme assembler využili k pohybu hráče. Ovšem celá příprava k tomu byla v BASICu, pouze rutina pohybu byla v assembleru. Abychom ukázali, jak uskutečnit tytéž operace pomocí čistě assembleru, byl dříve uvedený BASICovský program celý převeden do assembleru, a ten je uveden dále. V programu byly užity adresy v decimálním tvaru, protože tímto způsobem byl napsán i program v BASICu.

```
              0100 ; ..........
              0110 ;
              0120 ; ..........
0340          0130 ICHID  = $0340
0341          0140 ICDNO  = $0341
0342          0150 ICCOM  = $0342
0343          0160 ICSTA  = $0343
0344          0170 ICBAL  = $0344
```

```
0345        0180 ICBAH  = $0345
0346        0190 ICPTL  = $0346
0347        0200 ICPTH  = $0347
0348        0210 ICBLL  = $0348
0349        0220 ICBLH  = $0349
034A        0230 ICAX1  = $034A
034B        0240 ICAX2  = $034B
E456        0250 CIOV   = $E456
            0260 ; ..........
            0270 ;
            0280 ; ..........
00CC        0290 YLOC   = $CC
00CE        0300 XLOC   = $CE
00D0        0310 INITX  = $D0
00D1        0320 INITY  = $D1
0100        0330 STOTOP = $0100
D300        0340 STICK  = $D300
D000        0350 HPOSPO = $D000
0000        0360      * = $0600
            0370 ; ..........,
            0380 ;
            0390 ; ..........
0600 A56A   0400       LDA 106
0602 8D0001 0410       STA STOTOP
0605 38     0420       SEC
0606 E908   0430       SBC #8
0608 856A   0440       STA 106
060A 8D07D4 0450       STA 54279
060D 85CF   0460       STA XLOC+1
060F A900   0470       LDA #0
0611 85CE   0480       STA XLOC
            0490 ; ..........
            0500 ;
            0510 ; ..........
0613 A900   0520       LDA #0
0615 48     0530       PHA
0616 A260   0540       LDX #$60
0618 A90C   0550       LDA #$C
061A 9D4203 0560       STA ICCOM,X
061D 2056E4 0570       JSR CIOV
0620 A260   0580       LDX #$60
0622 A903   0590       LDA #3
0624 9D4203 0600       STA ICCOM,X
0627 A9ED   0610       LDA #NAME & 255
0629 9D4403 0620       STA ICBAL,X
062C A906   0630       LDA NAME/256
062E 9D4503 0640       STA ICBAH,X
```

```
0631 68     0650       PLA
0632 9D4B03 0660       STA ICAX2,X
0635 29F0   0670       AND #$F0
0637 4910   0680       EOR #$10
0639 090C   0690       ORA #$C
063B 9D4A03 0700       STA ICAX1,X
063E 2056E4 0710       JSR CIOV
            0720 ; ..........
            0730 ;
            0740 ; ..........
0641 A978   0750       LDA #120
0643 85D0   0760       STA INITX
0645 A932   0770       LDA #50
0647 85D1   0780       STA INITY
0649 A92B   0790       LDA #46
064B 8D2F02 0800       STA 559
            0810 ; ..........
            0820 ;
            0830 ; ..........
064E A000   0840       LDY #0
0650 A900   0850       LDA #0
            0860 CLEAR
0652 91CE   0870       STA (XLOC),Y
0654 88     0880       DEY
0655 D0FB   0890       BNE CLEAR
0657 E6CF   0900       INC XLOC+1
0659 A5CF   0910       LDA XLOC+1
065B CD0001 0920       CKP STOTOP
065E F0F2   0930       BEQ CLEAR
0660 90F0   0940       BCC CLEAR
            0950 ; ..........
            0960 ;
            0970 ;
            0980 ;
            0990 ; ..........
0662 A56A   1000       LDA 106
0664 18     1010       CLC
0665 6902   1020       ADC #2
0667 85CD   1030       STA YLOC+1
0669 A5D1   1040       LDA INITY
066B 85CC   1050       STA YLOC
066D A000   1060       LDY #0
            1070 INSERT
066F B9F006 1080       LDA PLAYER,Y
0672 91CC   1090       STA (YLOC),Y
0674 C8     1100       INY
0675 C008   1110       CPY #8
```

```
0677 D0F6   1120       BNE INSERT
0679 A5D0   1130       LDA INITX
067B 8D00D0 1140       STA 53248
067E 85CE   1150       STA XLOC
0680 A944   1160       LDA #68
0682 8DC002 1170       STA 704
0685 A903   1180       LDA #3
0687 8D1DD0 1190       STA 53277
            1200 ; ..........
            1210 ;
            1220 ; ..........
            1230 MAIN
068A 209A06 1240       JSR RDSTK
068D A205   1250       LDX #5
068F A000   1260       LDY #O
            1270 DELAY
0691 88     1280       DEY
0692 D0FD   1290       BNE DELAY
0694 CA     1300       DEX
0695 D0FA   1310       BNE DELAY
0697 4C8A06 1320       JMP MAIN
            1330 ; ..........
            1340 ; nyní čti joystick 1
            1350 ; ..........
            1360 RDSTK
069A AD00D3 1370       LDA STICK
069D 2901   1380       AND #1
069F F016   1390       BEQ UP
06A1 AD00D3 1400       LDA STICt
06A4 2902   1410       AND #2
06A6 F020   1420       BEQ DOWN
06A8 A000D3 1430 SIDE  LDA STICK
06AB 2904   1440       AND #4
06AD F02E   1450       BEQ LEFT
06AF AD00D3 1460       LDA STICK
06B2 2908   1470       AND #8
06B4 F02F   1480       BEQ RIGHT
06B6 60     1490       RTS
            1500 ; ..........
            1510 ;
            1520 ;
            1530 ; ..........
06B7 A001   1540 UP    LDY #1
06B9 C6CC   1550       DEC YLOC
06BB B1CC   1560 UP1   LDA (YLOC),Y
06BD 88     1570       DEY
06BE 91CC   1580       STA (YLOC),Y
```

```
06C0 C8     1590       INY
06C1 C8     1600       INY
06C2 C00A   1610       CPY #10
06C4 90F5   1620       BCC UP1
06C6 B0B0   1630       BCS SIDE
            1640 ; ..........
            1650 ;
            1660 ; ..........
06C8 A007   1670 DOWN  LDY #7
06CA B1CC   1680 DOWN1 LDA (YLOC),Y
06CC C8     1690       INY
05CD 91CC   1700       STA (YLOC),Y
06CF 88     1710       DEY
06D0 88     1720       DEY
06D1 10F7   1730       BPL DOWN1
06D3 C8     1740       INY
06D4 A900   1750       LDA #0
C6D6 91CC   1760       STA (YLOC),Y
06D8 E6CC   1770       INC YLOC
06DA 18     1780       CLC
06DB 90CB   1790       BCC SIDE
            1800 ; ..........
            1810 ;
            1820 ; ..........
06DD C6CE   1830 LEFT  DEC XLOC
06DF A5CE   1840       LDA XLOC
06E1 8D00D0 1850       STA HPOSP0
06E4 60     1860       RTS
            1870 ; ..........
            1880 ;
            1890 ; ..........
06E5 E6CE   1900 RIGHT IMC XLOC
06E7 A5CE   1910       LDA XLOC
06E9 8D00D0 1920       STA HPOSP0
06EC 60     1930       RTS
            1940 ; ..........
            1950 ;
            1960 ; ..........
06ED 53     1970 NAME .BYTE „S:“,$9B
06BE 3A
06EF 9B
06F0 FF     1980 PLAYER .BYTE 255,129,129,129,129,129,129,255
06F1 81
06F2 81
06F3 81
06F4 81
06F5 81
```

```
06F6 81
06F7 FF
```

Tento program využívá mnohé z rutin, které jsme již vysvětlili. Zde jsou prostě spojeny dohromady, aby jako celek splnily všechny úlohy PM grafiky. Protože se jedná o program analogický k BASICovaké verzi, začíná zúžením RAMTOPu o 6 stránek a tak vyhrazením místa pro PMG. Řádky 400 až 440 vykonávají tuto funkci; řádek 410 uchovává hodnotu RAMTOPu pro vymazávací rutinu později. Řádek 450 říká ATARI, kde je PMBASE, nová hodnota RAMTOPu. Lokace XLOC a XLOC+1 využijeme jako dočasné nepřímo adresované lokace v nulté stránce k vymazávací rutině. Řádky 520 až 710 opět nastaví GR.0 pod novým RAMTOPem. Řádky 750 až 780 uchovávají inicializační hodnoty X a Y, obrazovkové souřadnice hráče. Tyto hodnoty budou použity až později a zde jsou uvedeny právě kvůli analogii a BASICovským programem. Není totiž zapotřebí tyto souřadnice uchovávat, ale lze je přímo použit později v programu, na řádcích 790 až 800 nastavíme dvojitou velikost, a pak vymažeme celou PM plochu.

V BASICovském programu dosáhneme správné Y-ová souřadnice zápisem

```
PMBASE + 512 + INITY
```

Víme, že 512 bytů nad PMBASE představuje dvě stránky. Proto tedy vyšší byte této adresy musí být v našem assemblerovském programu dvakrát větší než PMBASE. V řádcích 1000 a 1030 si vytáhneme PMBASE (přidáme 2), přičteme dvojku a výsledek uložíme do YLOC+1, vyššího bytu Y-ové pozice v paměti. Nižším bytem je INITY. Pamatujte si, že níže chcete, aby se hráč objevil na obrazovce, tím výše musí být umístěn v paměti.

Pro uložení hráče na správné místo v paměti čteme postupně vždy jeden byte z tabulky dat zvané PLAYER a ukládáme ho nepřímým adresováním do paměťové lokace, kterou jsme právě nastavili. Když se hodnota v registru Y rovná osmi, jsme hotovi, protože jsme začali nulou a máme přemístit pouze 8 bytů. Kdyby byl náš hráč býval větší, jednoduše bychom změnili jeden byte na řádku 1110 na hodnotu o 1 větší než je počet bytů v hráči. Dále přečteme inicializační X-ovou pozici z INITX a uložíme do registru horizontální pozice hráče 0,53248 a do XLOC pro další použití v rutině pohybu.

Hráče pak zbarvíme do červena tím, že do barvicího registru hráče 0, 704, vložíme 68 a spustíme PM grafiku uložením čísla do GRACTL, 53277.

Hlavni smyčka programu je jednoduchost sama. Pomocí JSR odskočíme do programu, kde se čte joystick a pohybuje hráč, a pak do krátké zpožďovací smyčky. Kdybychom ji vynechali, hráč by se pohyboval tak rychle, že jeho pohyb by byl nekontrolovatelný! Pak opět skáčeme do smyčky, kde je čten joystick a uskutečněn pohyb hráče.

Pokud byste se chystali program prodloužit předáním vašich vlastních efektů (určeni kolize hráč – pozadí, vytvoření překážek apod.), nezapomeňte změnit počáteční adresu programu, protože zde uvedený program je už natolik dlouhý, že dalším prodloužením byste zasahovali do DOSu. Prostě změňte počáteční adresu na $6000 nebo nějakou jinou bezpečnou lokaci.

Přejeme vám mnoho úspěchů a hodně zábavy při implementaci vlastních PMG programů!

ČÁST ČTVRTÁ

PŘÍLOHY

PŘÍLOHA JEDNA

Instrukční soubor mikroprocesoru 6502

S ohledem na použití instrukčního souboru mikroprocesoru 6502 je třeba se především zmínit o jedné věci: každý počítačový jazyk má svou vlastní syntax, tj. způsob, jakým musí být jednotlivé příkazy napsány, aby jim počítač rozuměl. Různé assemblery počítače ATARI mají různou syntax, jež byly podrobně popsány v šestá kapitole.

V této příloze se naučíme příkazům, která mikroprocesor 6502 umí vykonat. Kompletní sada příkazů mikroprocesoru se všeobecně nazývá instrukční soubor. Pro snazší orientaci uvádíme jeho přehled v abecedním pořadí. Vysvětlení každé instrukce bude sestávat z:

1. příkladů jejího použití;
2. popisu adresovacích módů, jež dané instrukci přísluší (viz pátá kapitola);
3. vlivu vysvětlované instrukce na indikační registry stavového registru procesoru (viz třetí kapitolu).

ADC – ADD WITH CARRY (sečti spolu s indikačním registrem přenosu)

Jak jsme se již zmínili ve čtvrté kapitole, je každá instrukce 6502-ky představovaná třípísmennou zkratkou. Instrukce ADC je jedinou informací zabezpečující sčítáni dvou čísel. Zde je příklad jejího použití:

```
. ; tyto řádky jsou
. ; předcházejícími
. ; informacemi programu
ADC #2 ; přičti k akumulátoru číslo 2
.
.
```

Vyzkoušejme si, co se stane, když provedeme tuto instrukci. Mikroprocesor 6502 přičte k obsahu akumulátoru, ať je tam již cokoli, decimální číslo 2. Výsledný součet je opět v akumulátoru. Avšak je tu jeden problém. Pamatujete si? Název instrukce je „sečti s indikačním registrem přenosu“. Jistě také víte, že indikační registr přenosu je jedním z in-

dikačních bitů stavového registru procesoru. Kdykoliv tedy dojde na vykonání instrukce ADC, k součtu je přičtena také hodnota indikačního registru přenosu. Schématicky si to můžeme znázornit takto:

Akumulátor	Přenos	Instrukce	= Součet	Přenos
0	0	ADC #2	2	0
0	1	ADC #2	3	0

Všimněte si, že jestliže byl indikační registr přenosu původně nastaven na 1, je po vykonání instrukce ADC vynulován. Má to svůj smysl – jestliže by ho mikroprocesor nevynuloval, mohlo by dojít k jeho dvojnásobnému použití, aniž bychom si toho všimli.

Vliv na stavový registr procesoru

Mikroprocesor 6502 nastaví indikační registr přenosu podle příslušné instrukce:

Akumulátor	Přenos	Instrukce	= Součet	Přenos
253	0	ADC #6	3	1
253	1	ADC #6	4	1

V prvním příkladě je výsledný součet 253+6=259, ale víme, že největší možné číslo, které lze do akumulátoru uložit, je 255. Číslo 256 představuje nulu s přenosem 1, takže 259 je 3 s přenosem 1. Číslo 3 je uloženo do akumulátoru a indikační registr přenosu je nastaven na 1.

Ve druhém příkladě je situace obdobná, ovšem s tím rozdílem, že indikační registr přenosu je nastaven na 1 již před vykonáním instrukce. Projeví se to tím, že výsledný součet je o 1 větší.

Již dříve jsme se zmínili o schopnosti 6502-ky pracovat v decimální aritmetice. V tom případě je největší možné číslo, které lze do akumulátoru uložit, 99. Pracujeme-li v tomto režimu, je indikační registr přenosu nastaven poté, kdy výsledný součet po vykonání instrukce ADC je větší než 99.

Stav některých jiných indikačních registrů je také podmíněn instrukcí ADC. Indikační registr přetečení, V, je nastaven na 1, jestliže v důsledku sčítáni došlo ke změně

sedmého, nejvýznamnějšího, bitu. Indikační registr zápornosti, N, je nastaven, jestliže výsledkem sčítáni je číslo větší než 127, tj. v případě, kdy je sedmý bit roven 1. Vzpomeňte si, že pro mikroprocesor 6502 je každé číslo mezi 128 a 255 záporné, protože tato čísla mají sedmý bit rovný jedné. Konečně indikační registr nulovosti, Z, se nastaví v případě, že výsledek sčítáni je nulový. Prostudujte si podrobně následující tabulku:

```
  A N Z C V Instrukce =   A N Z C V Poznámka
-------------------------------------------------
  2 0 0 0 0 ADC #3        5 0 0 0 0 Přímé sčítání
  2 0 0 1 0 ADC #3        6 0 0 0 0 Přičte i C!
  2 0 0 0 0 ADC #254      0 0 1 1 0
  2 0 0 0 0 ADC #253    255 1 0 0 0
253 1 0 0 0 ADC #6        3 0 0 0 1
125 0 0 1 0 ADC #2      128 1 0 0 1
```

Nyní by již mělo být zřejmé, že testováním různých indikačních registrů celkem lehce získáme informaci o výsledku sčítání. Samozřejmě, že jsou k dispozici i informace pro testováni těchto indikačních registrů. Je nasnadě, že patři mezi nejpoužívanější.

Způsoby adresování

Instrukce ADC nám dovoluje použít kterýkoliv z osmi adresovacích způsobů, jež jsme probírali v páté kapitole u instrukce LDA. K jejich pochopení nám opět poslouží přehledná tabulka:

Způsob	Instrukce	Byty	Význam
Přímý	ADC #2	2	A+ #2
Absolutní	ADC $3420	3	A+obsah adresy $3420
Nultá stránka	ADC $F6	2	A+obsah adresy $F6
Nultá stránka,X	ADC $F6,X	2	A+obsah adresy $F6+X
Absolutní,X	ADC $3420,X	3	A+obsah adresy $3420+X
Absolutní,Y	ADC $3420,Y	3	A+obsah adresy $3420+Y
Indexovaný nepřímý	ADC($F6,X)	2	A+obsah adresy daný obsahem ($F6+Y)
Nepřímý indexovaný	ADC($F6),X	2	A+obsah adresy daný obsahem ($F6)+Y

Potřebujete-li podrobnější vysvětlení způsobů adresace, prosím, pročtěte si pátou kapitolu.

AND – THE LOGICAL AND INSTRUCTION (logický součin)

Instrukce AND uskutečňuje logický součin mezi obsahem akumulátoru a operandem, tzn. srovnává dvě čísla bit po bitu. Zopakujme si: jestliže oba porovnávané bity jsou rovny 1, je i výsledek roven 1. Jestliže je aspoň jeden z porovnávaných bitů roven 0, bude i výsledek roven 0. Podívejme se na příklad. Akumulátor obsahuje číslo 5:

```
AND #$0E
```

Nejlepší způsob, jak si názorně vysvětlit operaci AND, je převést obě čísla do binárního tvaru:

```
Hexa    Binární
------------------
#5    = #%00000101
#$0E  = #%00001110
Výsledek %00000100 = 4
```

Projděte si obě čísla bit za bitem a aplikujte na ně výše uvedené pravidlo. Zkusme si ještě jiný příklad. V akumulátoru je číslo 147:

```
AND #$1D
```

Opět převedeme na binární tvar:

```
#147    = #%10010011
#$1D    = #%00011101
Výsledek   %00010001 = 17
```

Instrukci AND používáme k tzv. maskování bytů. Předpokládejme, že chceme znát pouze hodnotu spodního niblu (poslední 4 bity) nějakého čísla. Abychom ji zjistili, jednoduše provedeme logický součin (AND) daného čísla s $0F. Protože vyšší 4 bity čísla $0F jsou všechny nulové a spodní nibl jsou samé jedničky, bude výsledek roven spodnímu niblu původního testovaného čísla. Přejeme-li si zjistit hodnotu horního niblu, provedeme AND $F0.

Vliv na stavový registr procesoru

Instrukce AND ovlivňuje indikační registry Z a N.

Akumulátor	Z	N	Instrukce	= A	Z	N	Význam
#5	0	0	AND #8	0	1	0	výsledek=0, Z=1
#$FE	0	1	AND #$5F	#$5E	0	0	výsledek 127,N=0

Způsoby adresování

Při práci s instrukcí AND lze tak jako u instrukce ADC použít všech osm způsobů adresování, které jsme vysvětlovali v páté kapitole na instrukci LDA.

ASL – ARITHMETIC SHIFT LEFT (aritmetický posuv vlevo)

Instrukce ASL využívá indikačního registru přenosu ve stavovém registru procesoru jako devátého bitu daného čísla a posouvá každý bit v čísle o jednu pozici doleva; odtud tedy název aritmetický posuv vlevo. Do nultého, nejméně významného bitu se uloží 0 a sedmý bit přejde do indikačního registru přenosu, tj. C. Následující příklady vydají za stránku vysvětlování:

```
C    76543210
0 <- 10110101 <- 0 (operace naznačena)
1    01101010      (operace vykonána)

C    76543210
0 <- 01101101 <- 0
0    11011010
```

Můžete se však zeptat: „Jaký je účel, význam této instrukce?“ Nuže, podívejme se pozorněji na druhý příklad. Původní číslo %01101101 je decimálně vyjádřeno 109. Vyjádříme-li si v decimálním tvaru číslo, jež dostaneme po vykonání instrukce ASL, dostaneme 218. Jedinou instrukcí jsme číslo v akumulátoru zdvojnásobili! Protože každá pozice v binárním čísle představuje právě dvojnásobek hodnoty na pozici o jedno místo vpravo, posuv vlevo zdvojuje hodnotu každého bitu. Skýtá se nám tedy úžasně jednoduchý způsob násobení čísel mocninami dvou – pro každý stupeň mocniny 2 jedna instrukce ASL.

POZOR: Ačkoliv se to zdá velmi jednoduché, musíte být velmi opatrní a mít stále na paměti přetečení do C. Nyní se

zaměřme na první příklad. Začínali jsme s číslem %10110101, decimálně 181. Po vykonání instrukce ASL dostáváme %01101010, což je ovšem decimálně 106. Je zřejmé, že 2 x 181 není 106, ale 362. Všimněte si však, že to je právě 106, výsledek, jenž jsme obdrželi, plus 256. Z toho tedy plyne, že kdykoliv násobíme instrukcí ASL, musíme brát do úvahy i hodnotu indikačního registru přenosu C a příslušným způsobem konečný výsledek opravit.

Vliv na stavový registr procesoru

Jak jsme již byli poučeni, indikační registr přenosu, C, bude vždy obsahovat nejvýznamnější bit původního čísla. Indikační registr N bude nastaven podle nejvýznamnějšího bitu výsledku (šestý bit původního čísla). Instrukce ASL dále ovlivňuje indikační registr Z. V následujících příkladech jsme použili způsobu adresování akumulátor.

A	N	C	Z	Instrukce	=	A	N	C	Z
128	1	0	0	ASL A	=	0	0	1	1
64	0	0	0	ASL A	=	128	1	0	0
192	1	0	0	ASL A	=	128	1	1	0
8	0	0	0	ASL A	=	16	0	0	0

Způsoby adresování

V následující tabulce jsou uvedeny způsoby adresace přípustné pro instrukci ASL.

Způsob	Instrukce	Byty	Význam
Akumulátor	ASL A	1	ASL hodnoty v akumulátoru
Absolutní	ASL $3420	3	ASL obsahu adresy $3420
Nultá stránka	ASL $F6	2	ASL obsahu adresy $F6
Nultá stránka,X	ASL $F6,X	2	ASL obsahu adrtsy $F6+X
Absolutní,X	ASL $3420,X	3	ASL obsahu adresy $3420+X

BCC – BRANCH ON CARRY CLEAR (větvení, jestliže C=0)

V BASICu rozlišujeme dva typy příkazů, které přesunují řízení programu na nový řádek. První je přímočarý příkaz GOTO. Jeho použití by mělo být všem zřejmé:

```
45 GOTO 60
50 .
55 .
60 ? „Tento řádek následuje po řádku 45.“
```

Tento způsob přesunu řízení nazýváme nepodmíněný.

Druhý typ přesunu v BASICu využívá porovnávacích a větvících možností počítače:

```
45 IF X=32 THEN 90
30 .
55 .
60 .
90 ? „Tento řádek může následovat po řádku 45.“
```

Tento způsob přesunu řízení se nazývá podmíněný. Podmínkou je zde hodnota proměnné X.

Instrukce BCC znamená, že jestliže je v indikačním registru přenosu nula (C=0), musí dojít k větvení programu na specifickou lokaci. Právě tak jako v BASICu i zde platí, že jestliže podmínka není splněna – v tomto případě jestliže je v C jednička – není jako další vykonán specifikovaný řádek, ale další řádek programu.

V assembleru vlastně ve všech případech specifikujeme větveni návěstími. Návěstím může být téměř jakýkoliv název, který chcete dát určitému řádku v assemblerovském programu. Podívejme se na krátký příklad:

```
        BCC SKIP
        LDA $0345
        .
        .
        .
        .
        .
SKIP LDA $4582
```

Proberme si tuto ukázku řádek po řádku. Jako první je instrukce BCC SKIP. V okamžiku, kdy je instrukce vykonávána, může být indikační registr přenosu, C, bud jedna nebo nula. Nejprve předpokládejme, že je roven jedné. V tom případě není splněna podmínka instrukce BCC. Znamená to, že protože není v C nula, neuskuteční se větvení programu na řádek opatřený návěstím SKIP, ale do akumulátoru se uloží obsah adresy $0343, jak říká instrukce na dalším řádku. Program dále pokračuje dalšími následujícími řádky.

Nyní předpokládejme, že v okamžiku vykonávání instrukce BCC je v C nula. V tomto případě je splněna podmínka větvení a jako bezprostředně další se vykoná instrukce uložení obsahu adresy $4582 (viz stále příklad) do akumulátoru. Program pokračuje vykonáním instrukce na řádku následujícím za SKIP LDA $4582.

Nyní by již mělo být zřejmé, jak důležité jsou indikační registry stavového registru procesoru. Instrukční soubor mikroprocesoru nám pak dovoluje přímo tyto bity ovládat.

Další podobností mezi instrukcí BCC a podmíněným větvením v BASICu spočívá v tom, že větvení lze provádět jak dopředu, tak dozadu. Předchozí příklad byl příkladem větvení dopředu. Podívejme se, jak by vypadalo větveni dozadu:

```
SKIP LDA $0245
     .
     .
     BCC SKIP
```

Jestliže je tedy C=0, instrukce BCC SKIP nás vrátí na řádek označený návěstím SKIP. Jestliže je v C jednička, není splněna podmínka větvení a program bude pokračovat vykonáním instrukce na řádku následujícím bezprostředně za řádkem BCC SKIP.

Na tomto místě je důležité zmínit se o důležitém omezení týkajícím se instrukcí větvení v assembleru. Instrukce BCC může přemístit řízení programu pouze 127 bytů dopředu či dozadu. Hodnota, o kolik bytů se bude větvit, je v jednom bytu následujícím za instrukcí, a každé jednobytové číslo větší než 127 je považováno za číslo záporné. Kdybychom se tedy pokusili o skok o 130 bytů dopředu, mikroprocesor 6502 bude tento pokyn chápat jako větvení záporné, o 255–130=125 bytů dozadu. Většina assemblerů nám na chybu, jež nastane pokusem o skok větší než je dovolený, přijde a označí ji v době

asemblování ztrojového textu. Může se však jednat o značnou časovou ztrátu. Jednodušší samozřejmě je vyvarovat se těmto chybám již v době psaní programu.

Ještě jedno upozornění týkající se relativního větvení. Většina assemblerů dovoluje tento zápis:

```
BCC +7
```

který programovému čítači říká, aby skočil o 7 bytů dopředu v případě splnění podmínky větvení. POZOR! – TOTO JE VELMI ŠPATNÁ PROGRAMÁTORSKÁ PRAXE! Chcete-li po nějakém čase program pochopit nebo ho dokonce změnit, narazíte na velký problém. Vložení řádku za tuto instrukci by nutně vedlo k zhroucení programu, protože by došlo k větvení na nesprávná místo. Použijeme-li pro označení řádku kam skočit případným větvením návěstí, program nabude na přehlednosti a snížíme tak i pravděpodobnost výskytu chyby.

Vliv na stavový registr procesoru

Žádný.

Způsoby adresování

Jediný přípustný způsob adresace všech instrukcí větvení (jak uvidíme dále, BCS, BEQ, BPL a další) je způsob relativní. Větvení je relativní na stávající (okamžitou, aktuální) pozici programového čítače, který normálně ukazuje na řádek bezprostředně následující za instrukcí větvení. Instrukce větvení vyžadují 2 byty a jejich vykonání trvá 2 strojové cykly.

BCS – BRANCH ON CARRY SET (větvení, jestliže C=1)

Jedná se o pravý opak instrukce BCC. Větvení nastane v případě, že indikační registr přenosu, C, je roven jedné. Ve všech ostatních aspektech je tato instrukce identická s BCC.

Vliv na stavový registr procesoru

Žádný.

Způsoby adresování

Opět jediným přípustným je způsob relativní. Pro další podrobnosti viz popis a způsob adresace instrukce BCC.

BEQ – BRANCH ON EQUAL TO ZERO (větvení, jestliže Z=1)

I tato instrukce je podobná instrukcím BCC A BCS. Liší se však v jedné podstatné věci. Jako určující faktor vyhodnocení, zda nastane větvení nebo ne, tato instrukce používá indikační registr nulovosti, Z, a ne indikační registr přenosu, C. Vzpomeňte si, že Z=1 pouze v případě, že výsledkem nějaké operace byla nula. Jestliže platí Z=0, potom výsledkem právě ukončené operace nebyla nula, a tudíž ani není splněna podmínka větvení. Zdá se to být trochu popletené, ale uvědomte si, že Z je indikační registr, a ten je nastaven tehdy, jestliže je splněna podmínka, kterou on vyjadřuje svým názvem: indikační registr nulovosti – tj. podmínka splněna, jestliže výsledkem nějaké operace je nula, a právě v tom případě se Z=1 (logická jednička – podmínka splněna).

```
        LDA #0
        BEQ SKIP
        .
        .
        .
SKIP    LDA $2F
```

Protože jsme před testováním indikačního registru nulovosti uložili do akumulátoru nulu, je Z=1 a dojde k přesunu řízení programu na řádek označený návěstím SKIP.

Vliv na stavový registr procesoru

Žádný.

Způsoby adresování

Platí totéž, co pro instrukce BCC a BCS.

BIT – TEST BITS IN MEMORY WITH ACCUMULATOR (nastavení bitů stavového registru procesoru)

Instrukce BIT vykonává logický součin (AND) mezi číslem uloženým v akumulátoru a číslem uloženým v jiné paměťové lokaci adresované instrukcí, ale odlišuje se v jedné velmi důležité věci od instrukce AND. Instrukce AND vykonává logický součin mezi číslem v A a jiným číslem v paměti a výsledek uchovává v akumulátoru. BIT taká vykonává logický součin, ale neukládá výsledek do A. Můžete se tedy ptát, k čemu je vlastně instrukce BIT dobrá.

Vzpomeňte si, že instrukce AND kromě logického součinu i nastaví indikační bity stavového registru procesoru. Instrukce BIT tedy provede logický součin daných dvou čísel s tím, že výsledek neuloží nikam, ale nastaví indikační registry procesoru.

Vliv na stavový registr procesoru

BIT ovlivňuje registry N, V a Z podle nám již známých pravidel. V následující tabulce předpokládáme, že v paměťové lokaci $0345 je hodnota $F3.

A	N	V	Z	Instrukce	=	A	N	V	Z
128	1	0	0	BIT $0345	=	128	1	1	0
5	0	0	0	BIT $0345	=	1	1	1	0
4	0	0	0	BIT $0345	=	0	1	1	1
3	0	0	1	BIT $0345	=	3	1	1	0

Instrukce BIT se používá především tehdy, kdy se chceme dovědět něco o hodnotě uložené někde v paměti počítače, aniž bychom porušili hodnotu v A. A to je velmi významné!

Způsoby adresování

Instrukce BIT dovoluje pouze dva způsoby adresace – absolutní a nultá stránka.

Způsob	Instrukce	Byty	Význam
Absolutní	BIT $3420	3	A AND obsah adresy $3420
Nultá stránka	BIT $F6	2	A AND obsah adresy $F6

BMI – BRANCH ON MINUS (větvení, jestliže N=1)

Jedná se o další instrukci podmíněného větvení. Určujícím faktorem je indikační registr zápornosti, N. Jinak platí všechno to, co u předchozích instrukcí podmíněného větvení včetně vlivu na stavový registr procesoru a způsobů adresování.

BNE – BRANCH ON NOT EQUAL TO ZERO (větvení, jestliže Z=0)

Jedná se o přesný opak instrukce BEQ, jako instrukce BCS je opakem BCC. Vše ostatní platí jako u předchozích instrukcí.

BPL – BRANCH ON PLUS (větvení, jestliže N=0)

BPL je opakem instrukce BMI. Při práci s těmito instrukcemi mějte na paměti, že ne všechny instrukce ovlivňují indikační registr zápornosti, N. Pro další podrobnosti viz instrukce BCC.

BRK – BREAK (zastavení)

BRK je analogická k BASICovské instrukci STOP. Víme, že příkaz STOP způsobuje, že se vykonávaný program zastaví a v daném místě je řízení předáno BASICu, který se ohlásí oznámením o zastavení a nápisem READY.

Příkaz STOP používáme nejčastěji při odlaďování programu, a to tak, že vložením táto instrukce na různá místa programu zjišťujeme, ke kterému STOP se program dostane. Obdobně lze využit assemblerovské instrukce BRK. Můžete ji vložit na určité místo v programu a spustit. Jestliže se program k instrukci BRK vůbec nedostal, víte, že zůstal „viset“ někde předtím. Většina dostupných ladicích programů (tzv. debuggerů) po vykonáni instrukce BRK vypíše obsah stavového registru procesoru, který tato instrukce neovlivňuje.

Způsoby adresování

Přípustný je pouze jeden – implicitní. Instrukce je jednobytová.

BVC – BRANCH ON OVERFLOW CLEAR (větvení, jestliže V=0)

Určujícím faktorem větvení je indikační registr přetečení V. Podmínkou větvení je, aby V=0. Opět platí to, co pro všechny ostatní instrukce podmíněného větvení.

BVS – BRANCH ON OVERFLOW SET (větvení, jestliže V=1)

BVS je protějškem instrukce BVC. I tato instrukce využívá (jako všechny ostatní) pouze relativního způsobu adresování a je dvojbytová.

CLC – CLEAR THE CARRY BIT (vynuluj indikační registr přenosu)

Tato instrukce má přímý a stálý vliv na indikační registr stavového registru procesoru – nuluje indikační registr přenosu, tj. C. Znamená to, že ať bylo před vykonáním této instrukce v C cokoli, tj. 0 nebo 1, po vykonání tam bude vždy 0. Nejčastěji se používá těsně před instrukci ADC. Víme již, že instrukce ADC přičítá k výslednému součtu i hodnotu v C, a proto si mnohdy potřebujeme být jisti, že je v C nula. Je to jediný způsob, jak zajistit, aby 2 plus 1 byly 3 a ne 4, jak se může stát! Názorně to lze vidět na krátké sekvenci instrukcí:

```
LDA #2
CLC
ADC #1
```

Vliv na stavový registr procesoru

CLC ovlivňuje jediný registr – již zmíněným způsobem ovlivňuje indikační registr přenosu, C.

Způsoby adresování

S instrukcí CLC je spojen pouze implicitní způsob adresování. Instrukce je jednobytová.

CLD – CLEAR THE DECIMAL FLAG (vynuluj indikační registr decimální aritmetiky)

Jak jsme se již zmínili dříve, mikroprocesor 6502 může pracovat buď v decimální nebo v binární aritmetice. Instrukce CLD nuluje indikační registr decimální aritmetiky, D, a uvádí tak 6502-ku do režimu práce v binární aritmetice. Pouze zřídka budou v této knize používány příklady využívající decimálního režimu. V tomto režimu totiž každá čtveřice bitů (tzv. nibl) reprezentuje jedno decimální číslo. Systém kódování se nazývá binárně kódovaná desítková čísla (BCD – z anglického Binary-Coded Decimal):

Bity	Reprezentace
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9

Rozdíl mezi decimální a binární aritmetikou je nejvíce patrný při sčítání a odčítání. Vyzkoušejme si například následující ukázku:

```
LDA $0345 ; tato lokace obsahuje #$59
CLC
ADC $0302 ; tato lokace obsahuje #$13
```

Jaké bude v akumulátoru číslo po vykonání těchto řádků? Řekli bychom, že $6C, a kdyby se sčítání odbývalo v binární aritmetice, měli bychom pravdu. Za předpokladu však, že jsme mikroprocesor již dříve uvedli do decimálního režimu, bude v akumulátoru 72! 6502-ka totiž byty na adresách $0345 a $0302 interpretovala jako BCD čísla a sečetla je v decimálním režimu: 59+13=72. Decimálního režimu použijeme tam,

kde chceme, aby byl výsledek rychle zobrazen na obrazovce. Jelikož každý nibl čísla reprezentuje decimální číslici, lze jednoduchým AND příslušnou číslici odmaskovat a rychle poslat na display.

Vliv na stavový registr procesoru

Instrukce CLD pouze nastavuje indikační registr decimální aritmetiky na nulu, žádné jiné účinky nemá.

Způsoby adresování

Tak jako u všech instrukcí pracujících přímo a indikačními registry stavového registru procesoru, je u CLD přípustný pouze implicitní způsob adresování. Instrukce je jednobytová.

CLI – CLEAR THE INTERRUPT FLAG (vynuluj indikační registr přerušení)

Tato instrukce pracuje přímo s indikačním registrem přerušení. Zřejmě bude následovat po nějaká instrukci, která nastavila tento registr I na 1. Pamatujte si, je-li I nastaven na 1, není povoleno maskované přerušení. Do této kategorie přerušení patři tzv. snímkové zatemnění (VBI – Vertical Blank Interrupt) a přerušení display listu (DLI – Display List Interrupt). Protože jsou tato přerušení používána velice často, nabývá instrukce CLI na důležitosti, protože je svým účinkem povoluje.

Vliv na stavový registr procesoru

Nuluje indikační registr přerušení, I.

Způsoby adresování

Přípustný je pouze implicitní, instrukce je jednobytová.

CLV – CLEAR THE OVERFLOW FLAG (vynuluj indikační registr přetečení)

Tak jako u předchozích instrukcí CLD a CLI je účinek CLV přímý – nuluje indikační registr přetečení. Do úvahy přichází pouze implicitní způsob adresace, instrukce je také jednobytová.

CMP – COMPARE MEMORY AND THE ACCUMULATOR (porovnej paměť s A)

V BASICu porovnáváme dvě čísla a určujeme, zda jsou si rovna, zda A je větší než B (A>B) nebo A je menší než B (A<B) apod. Např. takto:

```
35 IF A < B THEN…
```

Také instrukční soubor mikroprocesoru 6502 je vybaven instrukcemi, která umožňují porovnávat dvě čísla, třebaže trošku odlišným způsobem. Jedno z porovnávaných čísel musí být v akumulátoru, A, druhé pak kdekoliv v paměti počítače. Instrukce CMP odečítá od hodnoty v A obsah specifikované paměťové lokace a dle výsledku nastavuje (ovlivňuje) některé z indikačních registrů stavového registru procesoru. Obsah akumulátoru se vykonáním instrukce CMP nemění! Dojde pouze k nastavení některých indikačních registrů! Zjistíme-li jejich hodnoty, můžeme usuzovat na výsledek porovnávání. Nejprve si tedy proberme změny indikačních registrů a pak se zamyslíme nad několika příklady použiti instrukce CMP.

Vliv na stavový registr procesoru

Instrukce CMP nastaví Z na 1, jestliže čísla v paměti a v akumulátoru jsou si rovna. Jestliže se nerovnají, uloží se do Z nula. Jestliže výsledek odčítáni prováděného instrukcí CMP je větší než 127, nastaví se N na 1. Jinak tam bude nula. K nastavení C na 1 dojde, jestliže číslo v paměti je větší nebo rovno číslu v A. Pakliže je číslo v paměti menší než hodnota uložená v A, je C=0. V následující tabulce se předpokládá, že v paměťové lokaci $0345 je číslo #26:

A	Z	N	C	Instrukce	=	A	Z	N	C	Poznámka
26	0	0	0	CMP $0345	=	26	1	0	1	A=$0345
48	0	0	0	CMP $0345	=	43	0	0	1	A>=$0345
130	0	1	0	CMP $0345	=	192	0	0	1	A-$0345<128
8	0	0	0	CMP $0345	=	8	0	1	0	A-$0345>127

Třetí příklad demonstruje jeden zajímavý fakt týkající se instrukce CMP. Záporné číslo je samozřejmě menší než kladné číslo, podle výsledku odčítání by se však zdálo, že tomu tak není. Instrukce CMP nepoužívá znaménkovou aritmetiku – pouze porovnává dvě čísla mezi 0 a 255 s tím, že obě chápe jako kladná celá čísla. Chcete-li pracovat se znaménkovou aritmetikou, musíte zajistit příslušné ošetření úpravy.

Ještě jednou si celou tabulku projděte a na uvedené příklady aplikujte výše uvedená pravidla – jedná se o klíčovou instrukci.

Následující tabulka říká, po kterých instrukcích větvení by byly splněny podmínky pro jeho uskutečnění.

A	Paměť	Instrukce
8	8	BEQ, BCS, BPL
9	6	BCS, BPL, BNE
8	9	BMI, BCC, BNE

Na závěr ještě kratičký příklad, jak bychom přepsali do assembleru příkaz z ATARI BASICu:

```
IF R < B THEN GOTO Q
```

který znamená odskok na řádek Q, jestliže hodnota proměnné R je menši než hodnota proměnné B.

```
LDA R; do A ulož obsah paměťové lokace R
CMP B; srovnej s paměťovou lokací B
BCC Q; větvení, jestliže C=1 (tj. R<B)
```

Způsoby adresování

Instrukce CMP používá týchž osm způsobů adresace jako instrukce LDA v kapitole páté.

Způsob	Instrukce	Byty
Přímý	CMP #2	2
Absolutní	CMP $3420	3
Nultá stránka	CMP $F6	2
Nultá stránka,X	CMP $F6,X	2
Absolutní,X	CMP $3420,X	3
Absolutní,Y	CMP $3420,?	3
Index nepřímý	CMP ($F6,X)	2
Nepřímý index	CMP ($F6),Y	2

CPX – COMPARE INDEX REGISTER X WITH MEMORY (porovnej X s pamětí)

Tato instrukce porovnává obsah registru X s obsahem specifikované paměťové lokace na rozdíl od CMP, která porovnává obsah akumulátoru a paměťovou lokaci. Ve všech ostatních aspektech jsou tyto instrukce shodné. CPX se používá především k testování hodnoty registru X, používáme-li tento jako index neboli čítač např. nějakého cyklu.

```
      LDX #0 ; nastav čítač
LOOP LDA $D253,X
      STA $0600,X
      INX
      CPX #7 ; cyklus se vykoná 7x
      BNE LOOP
```

Vliv na stavový registr procesoru

Instrukce CPX ovlivňuje tytéž indikační registry a týmž způsobem jako instrukce CMP.

Způsoby adresování

Instrukce CPX připouští tři způsoby adresování uvedené níže:

Způsob	Instrukce	Byty
Přímý	CPX #2	2
Absolutní	CPX $3420	3
Nultá stránka	CPX $F6	2

CPY – COMPARE INDEX REGISTER Y WITH MEMORY (porovnej Y s pamětí)

Tato instrukce, jak již sám název napovídá, je s instrukcemi CMP, ale hlavně CPX identická až na fakt, že s určenou paměťovou lokací porovnává obsah registru Y. Naprosto shodným způsobem jako CPX i CMP ovlivňuje indikační registry; používá týchž způsobů adresování jako CPX.

DEC – DECREMENT MEMORY (zmenši o 1 paměť)

Za účelem dekrementace libovolné paměťové lokace (zmenšení jejího obsahu o 1) lze použít instrukci DEC. Ve skutečnosti existují dva způsoby, jak snížit, zmenšit obsah paměťová lokace. První, a daleko nejjednodušší, je přímo použít instrukci DEC. Druhý, a daleko krkolomnější, je napřed uložit obsah dané paměťové lokace do A, poté odečíst 1 a opět uložit výsledné číslo do původní paměťová lokace.

Vliv na stavový registr procesoru

Instrukce DEC ovlivňuje dva indikační registry – N a Z. Je-li výsledkem dekrementace nula, bude Z=1, jinak bude Z=0. Bude-li výsledkem dekrementace číslo větší než 127, nastaví se N na 1, jinak N=0.

Způsoby adresování

S instrukcí DEC se pojí čtyři způsoby adresování:

Způsob	Instrukce	Byty
Absolutní	DEC $3420	3
Nultá stránka	DEC $F6	2
Nultá str., X	DEC $F6,X	2
Absolutní,X	DEC $3420,X	3

DEX – DECREMENT THE X REGISTER (zmenši o 1 registr X)

Vykonáním táto instrukce se zmenši hodnota registru X o 1. Používá se především v cyklech, kde jako čítač cyklu využíváme registr X.

Vliv na stavový registr procesoru

Instrukce DEX má naprosto shodné účinky na indikační registry stavového registru procesoru jako instrukce DEC. Ovlivňuje tedy N a Z.

Způsoby adresování

Jak jste již jistě tušili, jediný způsob adresace, který se u táto instrukce uplatní, je implicitní způsob adresování.

DEY – DECREMENT THE Y REGISTER (zmenši o 1 registr Y)

Instrukce DEY dekrementuje obsah registru Y. Jinak je naprosto shodná s DEX.

EOR – EXCLUSIVE OR (exkluzivní logický součet)

Instrukce EOR má něco podobného s instrukcí AND. Vzpomeňte si, že instrukce AND provádí logický součin bit po bitu. Instrukce EOR provádí také bit po bitu, ale tzv. exkluzivní logický součet. Jestliže jeden, pouze jeden bit je roven jedné (myslí se z dvojice zpracovávaných čísel), bude i výsledný bit roven jedné. Pokud jsou oba bity rovné jedná nebo nule současně, bude na dané pozici ve výsledku nula. Výsledné číslo je uloženo v akumulátoru. Uveďme si příklad:

```
LDA 133
EOR 186

     Dec.   Bin.
     133 = %10000101
     186 = %10111010
Výsledek = %00111111 = 63
```

Jak vidíte, pouze šestý a sedmý bit u obou čísel se shodovaly, a proto je výsledný bit roven nule. Nejčastějším využitím instrukce EOR je vytváření doplňku čísla. Chceme-li např. najít doplněk čísla 143, provedeme EOR #$FF:

```
     143 = %10001111
     $FF = %11111111
Výsledek = %01110000 = 112
```

Vliv na stavový registr procesoru

Instrukce EOR dle výše uvedeného pravidla (u některých předchozích instrukcí) ovlivňuje N a Z.

Způsoby adresování

Instrukce EOR využívá oněch osm adresovacích módů jako instrukce LDA v kapitole páté. Pro jejich podrobnější popis se tedy obraťte tam.

INC – INCREMENT MEMORY (inkrementuj paměťovou lokaci)

Instrukce INC je přesným opakem instrukce DEC – způsobuje, že hodnota uložená v dané paměťová lokaci je zvětšena o 1. Instrukce ovlivňuje indikační registry N a Z. Používá týchž způsobů adresování jako DEC.

INX – INCREMENT THE X REGISTER (inkrementuj registr X)

Instrukce zvětší hodnotu uloženou v registru X. Nastavuje indikační registry N a Z. Pro její použití viz příklad u instrukce CPX. Lze ji adresovat jedině implicitně. Je jednobytová.

INY – INCREMENT THE Y REGISTER (inkrementuj registr Y)

Až na skutečnost, že tato instrukce inkrementuje registr Y, platí totéž, co pro instrukci INX.

JMP – JUMP TO ADDRESS (skok na adresu)

Již jsme probrali několik příkladů podmíněného přesunu řízení programu použitím instrukcí větvení (BCS, BEQ, BPL a další). Jsou jakousi analogií BASICovského příkazu IF. Víme však, že BASIC má také příkaz nepodmíněného přesunu GOTO.

```
30 GOTO 150
40 .
50 .
```

Víme, že po řádku 30 bude následovat řádek 150 a ne 40. Nezáleží to na žádné podmínce, skok je tedy zcela nepodmíněný. Také assembler je vybaven instrukcí podobného typu jako GOTO – je to instrukce JMP. Zde je její tvar:

```
JMP Q
```

Bez ohledu na předcházející tok programu, kdykoliv přijde na řadu vykonání této instrukce, bude program následovat od adresy Q. Opět se zde neuplatňuje žádná podmínka, přesun je nepodmíněný.

Vliv na stavový registr procesoru

Žádný.

Způsoby adresování

Instrukce JMP má pouze dva způsoby adresování. Nejprve se zmíníme o absolutním, protože ten již známe. V tomto modu je instrukce tříbytová a skok se uskuteční na adresu uvedenou ve druhém a třetím bytu.

Druhým způsobem adresace instrukce je tzv. nepřímý způsob, který je přípustný pouze u instrukce JMP. Abychom mohli nepodmíněný skok adresovat tímto způsobem, musíme někde v paměti vyhradit dva byty, v nichž bude cílová adresa skoku. Např. chceme provést skok na adresu $0620 nepřímo. Využijeme k tomu buňky $0423 a $0424. Nejprve do buňky $0423 vložíme nižší byte cílové adresy skoku, tj. $20, pak do buňky $0424 vyšší byte, tj. $06. Nepřímý skok má potom tvar:

```
JMP ($0423)
```

Závorky indikuji nepřímý skok; v závorkách je nižší byte nepřímé adresy. Pro adresu $0423 lze také použít návěstí. Zápis by potom vypadal takto:

```
JMP (SKOK)
```

Následující obrázek znázorňuje celou situaci graficky:

```
Lokace  Obsah
————————————
0421     A9
0422     B3
0423     20
```

```
0424     06
0425     95
0426     DE
```

JSR – JUMP TO SUBROUTINE (skok do podprogramu)

Instrukce JSR je assemblerovským protějškem BASICovského příkazu GOSUB. Instrukce JSR vloží hodnotu programového čítače na vrchol zásobníku, kde zůstane tak dlouho, dokud není podprogram ukončen. Hodnota je pak vytažena zpět, takže program může pokračovat od příslušného místa.

```
LDA #124
JSR SKOK
STA $4567
```

Na tomto místě je vhodné se zmínit o jedné věci týkající se odskoků do podprogramů v assembleru. V BASICu platí, že program běží rychleji, jestliže často používané podprogramy umístíme na začátek. Je to z toho důvodu, že BASIC interpreter, jestliže má najít místo, řádek, odkud podprogram začíná, prohledává celý program od začátku. Nevyplatí se proto umisťovat podprogramy na konec. Jiná je situace v assembleru. Aktuální adresa podprogramu je vlastně uvedena hned za kódem instrukce, nedojde tedy k žádnému prohledávání a je tudíž jedno, zda je podprogram na začátku či na konci. Ba právě naopak se vyplatí shromáždit všechny podprogramy na konci kvůli zvýšení čitelnosti celého programu. Ale to už je vaše věc.

Vliv na stavový registr procesoru

Žádný.

Způsoby adresování

Instrukce JSR dovoluje pouze absolutní adresování. Je tříbytová.

LDA – LOAD THE ACCUMULATOR (vlož do akumulátoru)

Výsledkem instrukce LDA je, že se do akumulátoru vloží číslo bud přímo nebo zkopírováním nějaké hodnoty uchovávané

v nějaké z paměťových lokaci počítače. Spolu s instrukcí STA se pravděpodobně jedná o nejčastěji používanou instrukci celého instrukčního souboru 6502-ky. Její hlavní využití je v uložení dané hodnoty do paměti, např.

```
LDA #1
STA 752; zhasne kurzor
```

a v přemístění obsahu paměťově lokace do jiné, např.

```
LDA $0520
STA $0344
```

Instrukce byla důkladně rozebrána ve čtvrté kapitole.

Vliv na stavový registr procesoru

Instrukce LDA nastavuje indikační registry N a Z.

Způsoby adresování

Dostáváme se k oněm osmi způsobům adresace, tak často vzpomínaným u jiných instrukcí.

Způsob	Instrukce	Byty
Přímý	LDA #2	2
Absolutní	LDA $3420	3
Nultá stránka	LDA $F6	2
Nultá str.,X	LDA $F6,X	2
Absolutní,X	LDA $3420,X	3
Absolutní,Y	LDA $3420,Y	3
Index nepř.	LDA ($F6,X)	2
Nepř. index	LDA ($F6),Y	2

LDX – LOAD THE X REGISTER (vlož do registru X)

Instrukce LDX se od LDA liší tím, že naplňuje registr X. Také nastavuje indikační registry N a Z, ale používá jiných způsobů adresování:

Způsob	Instrukce	Byty
Přímý	LDX #2	2

```
Absolutní      LDX $3420    3
Nultá stránka  LDX $F6      2
Nultá str.,Y   LDX $F6,Y    2
Absolutní,Y    LDX $3420,Y  3
```

LDY – LOAD THE Y REGISTER (vlož do registru Y)

Instrukci LDY už asi ani není nutné blíže objasňovat. Taktéž nastavuje indikační registry N a Z. Používá týchž způsobů adresování jako LDX, avšak pozor. Instrukce LDY nemůže použít indexování registrem Y, proto je nutné je změnit na X. Jedná se o poslední dva způsoby adresování ve výše uvedené tabulce (bude tam Nultá stránka,X; Absolutní,X).

LSR – LOGICAL SHIFT RIGHT (logický posun doprava)

Tato instrukce je opozitem instrukce ASL. Výsledkem LSR je, že do sedmého bitu se vloží nula a všechny bity se odrotují o jednu pozici doprava s tím, že poslední nultý bit přejde do C, indikačního registru přenosu.

```
      76543210    C
před: 10110101 -> 0
po:   01011010 -> 1
```

Vzpomeňte si; že ASL „násobí“ hodnotu dvěma. Protože na druhou stranu platí, že každý bit binárního čísla je přesně polovina hodnoty nejbližšího levého souseda, instrukce LSR danou hodnotu dělí dvěma:

```
%10110101 = 181
%01011010 = 90 a C=1
(čísla jsou z předchozího příkladu)
```

POZOR! Používáme-li znaménkovou aritmetiku, potom první byte v našem příkladu není 181, ale -(255-181)=-74, a víme, že polovina z -74 není 90. Abychom tedy vydělili záporné číslo dvěma, musíme si zapamatovat, že je záporné, převést ho na jeho kladný protějšek, vydělit a pak ho znovu převést zpět na záporná. Brr, je toho dost!

```
LDA #$FE ; -2
EOR #$FF ; doplněk
CLC
ADC #1   ; teď je to +2
LSR A    ; dělení
EOR #$FF ; převod zpátky
CLC
ADC #1
STA ...  ; výsledek, -1
```

Vliv na stavový registr procesoru

Je zřejmá, že je ovlivňován indikační registr zápornosti N, protože to je vlastně sedmý bit, a ten instrukce vždy zaplní nulou. Nastavován je rovněž C, jak vyplývá z popisu instrukce. Dále se nastavuje indikační registr nulovosti, Z.

Způsoby adresování

Způsob	Instrukce	Byty
Absolutní	LSR $3420	3
Nultá stránka	LSR $F6	2
Nultá str.,X	LSR $F6,X	2
Absolutní,X	LSR $3420,X	3
Akumulátor	LSR A	1

NOP – NO OPERATION (prázdná instrukce)

Instrukce NOP dělá to, co se dá očekávat z jejího názvu: nic! K čemu tedy je? Instrukci NOP lze použít pro rezervování místa pro modifikování instrukce nebo při odlaďování k eliminování některé instrukce, aniž bychom museli měnit lokace všech následujících instrukcí. Neovlivňuje žádný indikační registr, je jednobytová a oadresujeme ji pouze implicitně.

ORA – OR MEMORY WITH THE ACCUMULATOR (logický součet paměti a A)

Instrukce ORA je poslední ze tří logických instrukcí 6502-ky (předchozí dvě byly AND a EOR). Instrukce ORA porovnává dvě čísla bit po bitu a nastaví výsledný bit na 1, jestliže

alespoň jeden z dvojice porovnávaných bitů je roven jedné. Podívejte se na příklad:

```
1. číslo : %10100101
2. číslo : %01101100
ORA      : %11101101
```

Instrukci ORA použijeme tam, kde chceme nastavit některý bit daného čísla. Kupříkladu máme-li v paměťové lokaci $4235 nějaká číslo a chceme je použít s posledním bitem rovným jedné (nultý bit = 1), jednoduše provedeme:

```
LDA #1
ORA $4235
```

Akumulátor bude obsahovat číslo z paměťové lokace $4235 s jedničkou v nultém bitu.

Vliv na stavový registr procesoru

V závislosti na výsledném čísle v A po provedení instrukce ORA se nastaví N a Z.

Způsoby adresování

Instrukce ORA využívá oněch již slavných osmi způsobů adresování, jako LDA.

PHA – PUSH THE ACCUMULATOR ONTO STACK (vlož A do zásobníku)

Programujeme-li v assembleru, máma ve všeobecnosti k dispozici několik míst, do kterých můžeme dočasně uchovat nějakou hodnotu. Můžeme ji „schovat“ do předem vyhrazené paměti, do registrů X nebo Y, nebo do zásobníku. Z těchto možností pouze poslední nenaruší žádnou jinou informaci, jak by se to mohlo stát u instrukcí TAY nebo TAX. Při práci se zásobníkem musíme mít na zřeteli fakt, že zásobník je také používán na zapamatování návratových adres instrukcí JSR. Vložili-li bychom do zásobníku velké množství čísel, mohlo by to vést ke zhroucení počítače.

Instrukce neovlivňuje žádný z indikačních registrů, je jednobytová, adresuje se implicitně.

PHP – PUSH THE PROCESSOR STATUS REGISTER ONTO THE STACK (vlož stavový registr do zásobníku)

Instrukce PHP uloží do zásobníku byte obsahující indikační registry, tj. stavový registr procesoru. Účelem tohoto je uchovat jeho stav pro budoucí použití, zatímco se vykonávají jiné mezikroky. I zde platí stejné upozornění jako u instrukce PHA. Rovněž platí, že instrukce je jednobytová a připouští pouze implicitní způsob adresování.

PLA – PULL THE ACCUMULATOR FROM THE STACK (vyber A ze zásobníku)

Jedná se o protějšek instrukce PHA. Instrukce PLA vybere hodnotu z vrcholu zásobníku a uloží ji do akumulátoru.

Zmíníme se o jednom velice významném využití této instrukce, ale nejprve si musíme říci něco o strojových subrutinách volaných z BASICu. Předpokládejme, že ATARI BASIC nemá žádnou instrukci AND, a chceme tedy napsat rutinu ve strojovém kódu, který provede logický součin dvou čísel a výsledek vrátí do BASICu. Stojíme tedy před problémem, jak předat naše dvě čísla strojové rutině.

První metoda je univerzální a lze ji použít na téměř všech mikropočítačích. Vypočítáme si nejprve vyšší a nižší byty našich dvou čísel a příkazem POKE je vložíme do paměti. Strojová rutina tak má přístup k oběma číslům. Po provedení AND je opět uloží do vyhrazených paměťových lokací, odkud je lze vybrat již za pomoci BASICu. Celé by to potom vypadalo takto:

```
10 HIGHP=INT(P/256)
20 LOWP=P-256*HIGHP
30 HIGHQ=INT(Q/256)
40 LOWQ=Q-256*HIGHQ
50 POKE ADDR1,LOWP
60 POKE ADDR2,HIGHP
70 POKE ADDR3,LOWQ
60 PCKE ADDR4,HIGHQ
90 X=USR(1536)
100 LOWANS=PEEK(ADDR4)
110 HIGHANS=PEEK(ADDR5)
120 ANSWER=LOWANS+256*HIGHANS
```

Třebaže tento postup funguje, je trošku „přitažený za vlasy“. Ovšem pro mnoho druhů mikropočítačů je tento postup jediný možný. ATARI však nabízí mnohem elegantnější a jednodušší řešení.

Finta spočívá v tom, že z BASICu lze rutině ve strojovém kódu předat jistý počet parametrů. Těmito parametry mohou být adresy řetězců nebo různé konstanty vašeho BASICovského programu. Předávání se děje jejich prostým uvedením za adresou začátku strojového programu. Takto:

```
10 ANSWER=USR(1536,P,Q)
```

Ano! Pouze jeden řádek!

Váš ATARI si vezme čísla P a Q, upraví je do tvaru vyšší – nižší byte a uloží je na vrchol zásobníku, odkud je lze zpracovat strojovou rutinou pomocí instrukce PLA.

Pozor však na skutečnost, že tím, že ATARI dovoluje předávání parametrů příkazem USR, musí strojové rutině dát i informaci, kolik parametrů je předáváno! Tato informace je automaticky uložena na vrchol zásobníku, jakmile je vykonán BASICovský řádek. V případě řádku 10 bude na vrcholu zásobníku číslo 2. A to ještě dřív, než tam budou uloženy hodnoty čísel P, Q. V případě, že nepředáváme žádný parametr

```
120 X=USR(40960)
```

uloží se na vrchol zásobníku 0. Nezačali bychom tedy naši rutinu prostým PLA, mohla by i tato nula vést ke zhroucení celého systému, protože by došlo k porušení návratových adres.

Vliv na stavový registr procesoru

Instrukce PLA nastavuje indikační registry N a Z v závislosti na čísle vybíraném ze zásobníku.

Způsoby adresování

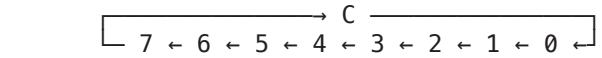
Jediný přípustný je implicitní. Instrukce je jednobytová.

PLP – PULL THE PROCESSOR STATUS REGISTER FROM THE STACK (vyber stavový registr ze zásobníku)

Instrukce PLP je do páru s instrukcí PHP. Slouží k nastavení podmínek, které byly v době vykonání PHP. Ovlivňuje tedy všechny indikační registry. Je jednobytová, adresuje se implicitně.

ROL– ROTATE LEFT (rotuj doleva)

Instrukce ROL je podobná instrukcím ASL a RSL tím, že také posouvá bity daného čísla, na rozdíl od těchto dvou instrukcí však neukládá do nejvyššího či nejnižšího bitu nulu, ale řídí se následujícím pravidlem: bit indikačního registru přenosu C je vložen do nultého bitu, všechny bity zarotují o jednu pozici doleva a nejvyšší bit přejde do C.



Provedeme-li instrukci ROL osmkrát, dostaneme se opět do výchozího stavu. Ne tak u instrukci ASL nebo LSR. Použitím osmi po sobě jdoucích ASL bychom celý byte vynulovali.

Máme-li před provedením ROL v C nulu, se stejným omezením jako u ASL a LSR platí, že instrukce ROL násobí dvěma.

Vliv na stavový registr procesoru

Instrukce ROL nastavuje N, C a Z.

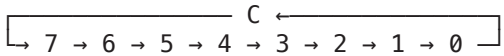
Způsoby adresování

Používá týchž způsobů adresování jako ASL nebo LSR.

Způsob	Instrukce	Byty
Absolutní	ROL $3420	3
Nultá stránka	ROL $F6	2
Nultá str.,X	ROL $F6,X	2
Absolutní,X	ROL $3420,X	3
Akumulátor	ROL A	1

ROR – ROTATE RIGHT (rotuj doprava)

K pochopení instrukce ROR by měl po pochopení instrukci ROL stačit schématický nákres:



Instrukce nastavuje tytéž indikační registry a používá týchž způsobů adresování.

RTI – RETURN FROM INTERRUPT (návrat z přerušení)

ATARI celkem často využívá přerušení. Jedná se vesměs o krátké rutiny, které přeruší tok hlavního programu, vykonají nějakou specifickou činnost a pak vrací řízení do místa, kde došlo k přerušení. K návratu slouží instrukce RTI.

V okamžiku přerušení 6502-ka přemístí obsah stavového registru procesoru a programového čítače do zásobníku. Vykonáním RTI se tyto hodnoty restaurují zpět. Instrukce RTI tak nastavuje všechny indikační registry. Je jednobytová, adresuje se implicitně.

RTS – RETURN FROM SUBROUTINE (návrat z podprogramu)

Assemblerovská instrukce RTS je analogií BASICovského příkazu RETURN. Způsobuje, že do programového čítače se vloží návratová adresa ze zásobníku, která tam byla uschována instrukcí JSR. Na tomto místě bychom rádi znovu upozornili na případné zhroucení systému, pracujeme-li nevhodně s instrukcemi PLA, PHA apod. Instrukce je jednobytová, neovlivňuje žádné indikační registry a používá implicitního způsobu adresování.

SBC – SUBTRACT WITH CARRY (odečti spolu s C)

Instrukce SBC je jediná instrukce instrukčního souboru 6502-ky, která dovoluje určit rozdíl dvou čísel tak, že první vložíme do akumulátoru a druhé odečteme.

```
LDA #8
SBC #6
```

Výsledná 2 zůstává v A.

V případě, že potřebujeme odečítat „přes desítku“ (24-9; 32-3 atd.), musíme před provedením SBC nastavit indikační registr přenosu. Protože jen stěží můžete dopředu vědět, jaká čísla budete odčítat, nezapomeňte před SBC vždy provést SEC (tak jako před ADC vždy CLC).

```
SEC ; C = 1
LDA #24
SBC #26
?   ; výsledek
```

Jaký bude výsledek? Nastavením C prakticky nezačínáme odečítáni s 24, ale s 256+24=280. Potom tedy 280-26=254, a tato hodnota bude jako výsledek v akumulátoru.

Vliv na stavový registr procesoru

Potřebovali-li jsme jedničku v C jako vypůjčení si vyššího řádu, bude po vykonáni operace C=0. Dále se nastaví N a V, indikační registr přetečení. V=1, jestliže je výsledek větší než plus nebo minus 127.

Způsoby adresování

Instrukce SBC využívá osm adresovacích módů jako instrukce LDA.

Způsob	Instrukce	Byty
Přímý	SBC #2	2
Absolutní	SBC $3420	3
Nultá stránka	SBC $F6	2
Nultá str., X	SBC $F6,X	2
Absolutní,X	SBC $3420,X	3

```
Absolutní,Y    SBC $3420,Y    3
Index nepř.    SBC($F6,X)     2
Nepř. index    SBC($F6),Y     2
```

SEC – SET THE CARRY BIT (nastav C=1)

Instrukci používáme všude tam, kde potřebujeme, aby C=1. Jiné indikační registry neovlivňuje.

SED – SET THE DECIMAL MODE (nastav D=1)

Instrukce SED nastaví indikační registr decimální aritmetiky a tak ji povolí.

SEI – SET THE INTERRUPT FLAG (nastav I=1)

Instrukce SEI nastaví indikační registr přerušení a tím ho zakáže. Žádná z instrukci si nevšímá jiných indikačních registrů. Všechny (SEC, SED i SEI) jsou jednobytové a adresují se implicitně.

STA – STORE THE ACCUMULATOR IN MEMORY (uchovej A v paměti)

Tato instrukce může uchovat libovolnou hodnotu nacházející se v akumulátoru do paměti. Všimněte si, že nemá nejmenší vliv na hodnotu v akumulátoru. Prostě tuto hodnotu zkopíruje do nějaké paměťové lokace. Lze tedy psát:

```
LDA #1
STA 752; uloží sem 1
STA 755; i zde bude 1
```

Jak již bylo naznačeno u LDA, lze STA využít pro přesun bloku paměti:

```
     LDY #10
LOOP LDA $5699,Y
```

```
        STA $0831,Y
        DEY
        BNE LOOP
```

Tato, celkem bez smyslu, rutina přemístí 10 bytů nacházejících se od adresy $5699 na adresu $831 a níže. V BASICu by obecně vypadala takto:

```
10 FOR I=1 TO 10
20 POKE ADR1+I,PEEK(ADR2+I)
30 NEXT I
```

Vliv na stavový registr procesoru

Žádný.

Způsoby adresování

Instrukce STA používá 7 z oněch osmi způsobů adresování jako u LDA, protože samozřejmě nelze použít přímý způsob.

Způsob	Instrukce	Byty
Absolutní	STA $3420	3
Nultá stránka	STA $F6	2
Nultá str.,X	STA $F6,X	2
Absolutní,X	STA $3420,X	3
Absolutní,Y	STA $3420,Y	3
Index nepřímý	STA($F6,X)	2
Nepřímý index	STA($F6),Y	2

STX – STORE THE X REGISTER (uchovej registr X)

Instrukci lze použít týmž způsobem jako STA s tím ovšem, že se do paměti zkopíruje obsah registru X. Nenastavuje žádné indikační registry a připouští tyto způsoby adresace:

Způsob	Instrukce	Byty
Absolutní	STX $3420	3
Nultá stránka	STX $F6	2
Nultá str.,Y	STX $F6,Y	2

STY – STORE THE Y REGISTER (uchovej registr Y)

Instrukce kopíruje obsah registru Y do paměti. Nenastavuje žádné indikační registry. Lze ji oadresovat absolutně, nultou stránkou a nultou stránkou,X (počet bytů viz tabulka u STX).

Poslední instrukce shrneme do jednoho celku. Jedná se o instrukce přesunu, pro které platí následující pravidla. Všechny dále uvedené instrukce nastavují indikační registr nulovosti Z a zápornosti N. Všem je společný implicitní způsob adresování a fakt, že jsou jednobytové. Pro přesun platí následující: vždy se kopíruje obsah registru uvedeného v názvu instrukce na druhém místě do registru, jehož označení je na třetím místě v názvu instrukce! Konkrétně, máme instrukci TXA. Tato instrukce kopíruje obsah registru X do akumulátoru, A. Podobně TAY zkopíruje obsah akumulátoru do registru Y. Písmeno S označuje ukazatel zásobníku (Stack pointer).

TAX – TRANSFER ACCUMULATOR TO THE X REGISTER
TAY – TRANSFER ACCUMULATOR TO THE Y REGISTER
TSX – TRANSFER THE STACK POINTER TO THE X REGISTER
TXA – TRANSFER THE X REGISTER TO THE ACCUMULATOR
TXS – TRANSFER THE X REGISTER TO THE STACK POINTER
TYA – TRANSFER THE Y REGISTER TO THE ACCUMULATOR

Assembler pre začiatočníkov

Zostavil: Marián Korčák
Vydal: Klub elektroniky Zväzarmu Tlmače
Náklad: 1000 ks
Tlač povolil: Odbor kultúry ONV Levice č.j.16/87
Určené pre vnútorné potreby Zväzarmu.
Neprešlo jazykovou úpravou.
Prvé vydanie 1987