Naučte se assembler

Základní pojmy

Na úvod bude dobré ujasnit si některé základní pojmy.

Mikroprocesor se skládá z jednoho (nebo více) integrovaných obvodů. Tohle označení se používalo hlavně v 70. a 80. letech 20. století, aby se odlišily nové *mikroprocesory* (většinou obvody vysoké a velmi vysoké integrace – LSI, VLSI) od starých *procesorů*, poskládaných z mnoha jednodušších obvodů. S masivním nástupem PC třídy 386 a vyšší se zase začalo používat slovo „procesor". Slovo „mikroprocesor" tak zůstává spojeno hlavně s osmibitovou érou.

Mikroprocesor většinou obsahuje nějakou řídicí jednotku, která vykonává program, sadu registrů, v nichž jsou uložena data, se kterými se pracuje, a nějakou jednotku, ve které se provádějí matematické operace.

Ve světě mikroprocesorů nejsou žádné příkazy známé z vyšších programovacích jazyků. PRINT tu nenajdete. Jediné, co procesor umí, je vzít z paměti číslo – **operační (strojový) kód instrukce** – a podle něj provést nějakou činnost, třeba přesunout hodnotu odněkud někam nebo sečíst dvě čísla. Jediné, čemu mikroprocesor rozumí, je tedy *číslo*. A protože hovoříme o osmibitových mikroprocesorech, je přirozené, že ta nejčastěji používaná čísla budou právě osmibitová, tedy v rozsahu 0 až 255, hexadecimálně 00h až FFh.

Platí tedy. že každý osmibitový procesor zná přesně 256 instrukcí? Ani náhodou! Některé jich znají mnohem míň (třeba procesor 8080), takže některé operační kódy nemají žádnou přiřazenou instrukci. Jiné naopak mají mnohem víc instrukcí (Z80) a řeší to tím, že některé instrukce mají operační kód vícebajtový.

Program se skládá z instrukcí, které procesor vykonává po řadě tak, jak jdou po sobě v paměti, s výjimkou skoků. Program je zapsaný třeba takto:

21h 55h 3Ah

Tyto tři bajty jsou procesorem 8080 interpretovány tak, že do registrů H a L uloží hodnoty 3Ah a 55h. U procesoru Z80 mají tato tři čísla stejný efekt (protože Z80 je s 8080 na úrovni strojového kódu kompatibilní). U procesoru 6502 mají zcela jiný význam – vezme se obsah registru X, k němu se přičte číslo 55h, výsledek se ořízne na osm bitů a dostaneme adresu. Z paměti na téhle adrese se vezme jeden bajt, z paměti na adrese o jedničku vyšší druhý bajt. Tyto dva bajty dají dohromady 16bitovou adresu v paměti, kam procesor sáhne pro číslo, a nakonec provede operaci AND mezi hodnotou v registru A a tímto číslem (výsledek uloží do registru A). No a pak následuje prázdná instrukce (3Ah).

Různé procesory mají tedy odlišné vnímání stejných kódů, což je důvod, proč nelze vzít program pro jeden z nich a spustit bez úprav na druhém.

Jazyk symbolických adres (JSA) slouží k tomu, aby si programátor nemusel pamatovat tyhle číselné kódy (což o to, většinou si je stejně pamatují), a aby u složitějších programů nemusel počítat adresy, tj. na jaké adrese v paměti je která instrukce. Proto zavádí jednak symbolická návěští (takže se zapisuje nikoli JUMP 1234h, ale třeba JUMP START), a jednak zavádí symbolická jména pro instrukce. Takže třeba u procesoru 8080 nemusí programátor zapisovat zmíněný skok jako kód C3h, ale symbolicky: „JMP" (jako že *jump*, rozumíme si…)

Assembler je nástroj, který překládá program, zapsaný v těchto symbolických jménech, do konkrétních kódů jednotlivých instrukcí.

Co to znamená, když se řekne „Program je napsaný v assembleru"? Ve skutečnosti to, že program je napsaný v jazyku symbolických adres. Zkrátka se stalo, že se tomuhle jazyku začalo říkat „assembler", i když správně je assembler vlastně ten překladač.

A když někdo řekne, že napsal program „ve strojáku"? Pravděpodobně ho napsal v tom symbolickém jazyce a přeložil assemblerem do strojového kódu. „Stroják" je takové označení, které se taky přeneslo na jazyk symbolických adres. *Ale mohlo se stát, že je dotyčný hardcore programátor a opravdu z hlavy nadiktoval instrukční kódy…*

Teď jsme si vymezili základní pojmy, co dál? V zásadě se můžeme pustit do popisu procesoru a jeho instrukcí. Můžete se podívat na procesor, který vás zajímá, ale pokud nevíte, jaký vás zajímá, začněte od toho nejstaršího, kterému se budu věnovat, a tím je Intel 8080.

8080 – architektura

Než se podíváme na jednotlivé instrukce, ukážeme si, co v takovém procesoru 8080 vlastně je a jak to můžeme použít.

Procesor 8080 byl uveden na trh v roce 1974. Bohužel, trpěl několika nedostatky, z nichž snad největší bylo to, že výstupy procesoru snesly jen malé zatížení. Brzy přišla varianta 8080A, která byla naprosto totožná co do vývodů a instrukcí, ale měla posílené vývody a byla rychlejší. Když se dneska hovoří o mikroprocesoru 8080, má mluvčí na mysli nejčastěji právě „áčkovou" variantu.

Mikroprocesor 8080 byl v jednom pouzdře DIL se 40 vývody.

Z těchto 40 vývodů bylo 16 pro adresovou sběrnici (A0-A15), 8 pro datovou (D0-D7), 4 piny sloužily k napájení (zem, +5V, +12V, -5V) a zbytek byly řídicí signály(RESET, INT, WR apod.)

Mikroprocesor pro svou činnost vyžadoval dva podpůrné obvody – 8224, který se staral o správné generování a fázování hodin, a 8228, který posiloval datovou sběrnici a zároveň se staral o generování řídicích signálů MEMR, MEMW, IOR, IOW a dalších. (MEMx obsluhovaly paměť, IOx vstupně-výstupní zařízení, R znamenalo čtení, W zápis).

Díky šestnácti adresním linkám může mikroprocesor adresovat paměť o kapacitě 64kB (2^16). Z paměti může přenést osmibitové slovo (tedy hodnotu 0-255).

ARCHITEKTURA 8080

V procesoru se nachází několik základních bloků:

- **Časovací a řídicí obvody** se starají o správnou synchronizaci ostatních částí, o správné vyhodnocení řídicích signálů, co přicházejí zvenčí, a o posílání stavových informací ven.
- **Registr instrukcí** drží instrukční kód, načtený z paměti, dokud jej nezpracuje dekodér.
- **Dekodér instrukcí** rozloží instrukční kód na posloupnost základních operací, o jejichž provedení se postará řadič.
- **ALU**, neboli aritmeticko-logická jednotka se stará o základní matematické operace (sčítání, odčítání, AND, OR, XOR) s osmibitovými čísly.
- **Akumulátor** (též registr A) je osmibitový pracovní registr procesoru. S ním se provádí naprostá většina operací, zejména aritmetické, logické či porovnávací.
- **Pracovní registry** B, C, D, E, H a L jsou registry, které může programátor využít pro dočasné uložení hodnot. Jsou opět osmibitové. Některé instrukce ale dokážou pracovat s tzv. registrovým párem – v takovém případě se berou registry B a C, resp. D a E / H a L jako jeden šestnáctibitový registr. Pár registrů H a L je často používán pro uložení adresy při

nepřímém adresování – např. „do akumulátoru se přesouvá hodnota, která je uložena v paměti na adrese, uložené v registrech HL".

- **Ukazatel zásobníku**, zvaný též registr SP, je šestnáctibitový registr, v němž je uložena adresa tzv. zásobníku. Více si o jeho funkci řekneme v kapitole o podprogramech.
- **Programový čítač** (registr PC) je šestnáctibitový registr, v němž je uložena adresa, z níž procesor čte instrukci. Po přečtení instrukce se zvýší o 1, 2 nebo 3, takže ukazuje na další instrukci. (Záleží na délce instrukce)
- **Příznakový registr F** není pro většinu instrukcí samostatně přístupný. Obsahuje několik tzv. příznakových bitů, které udržují informaci o tom, jestli poslední výsledek byl 0, jestli byl záporný, jestli při výpočtu přetekla hodnota přes hranici 255 (nebo pod 0) atd. Podle těchto bitů pak může procesor vykonat tzv. podmíněné instrukce.

Krom těchto částí obsahuje procesor i další skryté registry (W, Z, ACT), ke kterým ale nemá programátor přístup. Mikroprocesor je používá pro ukládání mezivýsledků dle své potřeby.

OPERACE, CYKLY T, M

Po zapnutí napájení nebo po signálu RESET je procesor uveden do základního stavu. Je zakázáno přerušení (k pojmu přerušení se dostaneme později) a do registru PC (šestnáctibitový ukazatel na instrukci) je uložena nula. Ostatní registry mají obsah takový, jaký měly před RESETem (po zapnutí napájení tedy náhodný).

Hodiny běží… Co se v procesoru děje?

Probíhá první strojový cyklus (zvaný M1). V tomto cyklu procesor pošle na adresovou sběrnici obsah registru PC (teď to je 0), zvýší obsah tohoto registru o 1 a pošle požadavek „čtení z paměti" (MEMR). Systém okolo by se měl postarat, aby na tento požadavek odpověděla paměť a poslala po datové sběrnici obsah na dané adrese. Mikroprocesor si tento obsah přečte, uloží do registru instrukcí, dekóduje a provede požadovanou činnost. Někdy není potřeba žádný dodatečný čas, jindy se například zapisuje do paměti nebo z ní čte, takže procesor zařadí některé z cyklů M2, M3, M4, M5. Po dokončení instrukce se celý proces opakuje.

V případě, že instrukce vyžaduje nějaký parametr, následují cykly čtení z paměti. Instrukce může jako parametr požadovat 1 nebo 2 bajty. Ty jsou načteny z adresy PC+1 a PC+2. PC je zvýšen o 2 nebo 3. Instrukce je pak provedena (což si opět může vyžádat nějaké strojové cykly) a po jejím dokončení se celý proces zase opakuje.

Strojový cyklus M1 se skládá ze čtyř nebo pěti taktů systémových hodin. Protože každá instrukce obsahuje vždy cyklus M1, v němž je načtena a dekódována, tak je jasné, že nejrychlejší instrukce zabere čtyři hodinové cykly (zapisuje se jako 4T). Pokud běží hodiny na frekvenci 2MHz, tak budou ty nejrychlejší instrukce prováděny s frekvencí 500kHz (tj. každá bude trvat 2 mikrosekundy). Nejpomalejší instrukce zaberou 18 taktů (18T). Informace o tom, jak je která instrukce „dlouhá" (tj. kolik T trvá procesoru, než ji zpracuje) je důležitá ve chvíli, kdy je třeba optimalizovat kód a zrychlit jej.

Základy assembleru

Ano, já vím, terminologicky to není správné označení, ale *mí čtenáři mu rozumí*. Půjde o **Jazyk symbolických adres**, všeobecně (a nesprávně) nazývaný *assembler*.

Procesor, jak jsme si už řekli, zpracovává instrukce tak, že čte z paměti jejich strojový kód po bajtech, ty dekóduje a podle nich provádí požadované činnosti.Zápis programů v číselných hodnotách by byl velmi nepohodlný, proto se používá zápis pomocí mnemotechnických názvů instrukcí. Takovému zápisu se říká „jazyk symbolických adres", nebo (nesprávně, ale všeobecně srozumitelně) „zápis v assembleru" (anglicky *assembly language*).

Definice JSA na Wikipedii.

Program se v JSA zapisuje jako v jiných jazycích do textového souboru po řádcích. Řádky mají přesně definovaný formát a skládají se z následujících prvků:

1. **Návěští:** řetězec znaků [A-Z, 0-9, podtržítko], který nezačíná číslicí. Některé překladače rozlišují jen prvních 6 znaků, jiné prvních 8, další pak všechny. Návěští je zakončeno dvojtečkou.
2. **Instrukce**: mnemotechnická zkratka instrukce, popřípadě pseudoinstrukce (direktivy)
3. **Parametry**: Některé instrukce a direktivy vyžadují další údaje, se kterými musí instrukce pracovat. Zapisují se za mnemotechnické označení instrukce, od které se oddělí mezerou (nebo mezerami). Pokud je parametrů víc, oddělují se od sebe čárkami.
4. **Poznámka:** Cokoli, co si potřebujete poznamenat. Na začátku je znak středník (;), vše, co je za tímto znakem, překladač ignoruje.

Ukažme si různé typy řádků.

- **Prázdný řádek** překladač prostě ignoruje
- **Řádek, kde je jen poznámka**. Pokud je prvním znakem na řádku (mezery nepočítáme) znak středník, bere se zbytek řádku jako poznámka a překladač jej ignoruje.
- **Řádek s instrukcí:** Na řádku je uvedena existující (pseudo)instrukce, popřípadě její parametry, má-li nějaké. Překladač ji přeloží na odpovídající operační kód. Za parametry může být ještě poznámka, kterou překladač ignoruje
- **Řádek s návěštím**: Libovolný typ výše uvedených řádků může ještě začínat návěštím, tj. řetězcem ukončeným dvojtečkou. V takovém případě překladač přiřadí tomuto řetězci adresu, která odpovídá internímu počítadlu adres. (K tomu se ještě vrátím.)

Podívejme se na ukázku kódu:

```
1       ; Program zacina
2       .org 0100H    ; nastavime pocatek kodu
3
4       ZACATEK:
5               nop
6               nop
7               nop
8       NAVEST: nop
9               nop
```

V tomto kódu najdete řádek s komentářem (1), prázdný řádek (3), řádky s instrukcemi i řádky s návěštím.

Použil jsem jen jednu instrukci, totiž instrukci NOP, kterou znají všechny tři procesory a její význam je prostý: Nedělej nic. Možná se vám zdá zbytečné mít instrukci, která nedělá nic, ale ona se docela dobře hodí – když potřebujete někde „chvilku počkat", nebo když si chcete někde nechat místo pro budoucí změnu. U procesorů 8080 / Z80 je její kód 0, u procesoru 6502 je to 0EAh.

Instrukce „.org" není instrukcí v pravém slova smyslu. Nemá totiž žádný operační kód. Pouze říká překladači: Teď se chovej tak a tak. Patří tedy mezi pseudoinstrukce (nebo taky „direktivy"). Org je zkratka pro „origin", tedy počátek. Tady říká překladači: *Ber to tak, že tady, jak jsem já, bude nějaká konkrétní adresa, a následující instrukce tedy ukládej od té adresy dál.*

Pseudoinstrukce .org má jeden parametr – totiž tu adresu. Tady je zapsaná v hexadecimální podobě.

JAK LZE ZAPSAT KONSTANTY V ASSEMBLERU?

- Hexadecimální zápis
  - Pomocí číslic 0-9 a znaků A-F. Číslo začíná vždycky číslicí a je ukončeno znakem H (h). Takže: 1234h, 00AAH, 23beh, 0cfh, 18H, …
  - Pomocí číslic 0-9 a znaků A-F, před kterým je znak $. Takže: $1234, $00AA, $23be, $cf, $18, …
  - Některé překladače umožňují i zápis v „céčkové“ syntaxi „0x…“
- Binární zápis
  - Řetězec znaků 0,1 ukončený písmenem „b“
  - Řetězec znaků 0,1, před kterým je znak „%“
- Desítkové číslo
  - Zapsáno tak, jak jsme zvyklí.
- Znaková konstanta
  - Znak zapsaný v apostrofech nebo uvozovkách. Hodnotou je ASCII kód daného znaku. Např. „@“ = 64 = 40h

Tady je na místě připomenout, že konkrétní syntaxe jednotlivých překladačů se může lišit. V dalším popisu budu vycházet z tvaru, jaký jsem implementoval v překladači ASM80. Totéž platí i pro pseudoinstrukce – někde je správný zápis „org“, u jiného překladače je to důsledně „.org“ s tečkou. ASM80 povoluje oba tvary. Původní assemblery braly zápis bez tečky.

LISTING

Když si vezmete výše zapsaný kód a zkusíte ho přeložit překladačem – např. výše zmíněným ASM80 – získáte na výstupu jednak binární podobu (která nebude moc zajímavá), jednak takzvaný *listing*, což je výpis programu spolu s adresami a instrukčními kódy.

```
1    0000                ; Program zacina
2    0100           .ORG   0100H   ; nastavime pocatek kodu
3    0100           ZACATEK:
4    0100   00               NOP
5    0101   00               NOP
6    0102   00               NOP
7    0103   00      NAVEST:  NOP
8    0104   00               NOP
9
10   _PC        0104
11   ZACATEK    0100
12   NAVEST     0103
```

Každý řádek začíná adresou, na jaké je daná instrukce uložena. Vidíte, že na prvním řádku ještě překladač neví, kde chceme program mít, a tak počítá s tím, že bude uložený od adresy 0. Na druhém řádku pseudoinstrukcí .org říkáme, že program bude v paměti uložen od adresy 0100h. Překladač si proto nastaví interní počítadlo adres (_PC) na hodnotu 0100h. Třetí řádek je „prázdný řádek s návěštím“. To znamená, že do tabulky návěští je zařazeno návěští se jménem ZACATEK a je mu přiřazena hodnota interního počítadla adres, tedy 0100h.

Čtvrtý řádek obsahuje konečně nějakou reálnou instrukci. Je to instrukce NOP, je přeložena na operační kód (00) a ten bude uložen na adrese 0100h. Počítadlo se posune na následující adresu, tedy 0101h.

Pátý řádek: NOP, kód je zase 00, je uložen na adrese 0101h a počítadlo se posouvá…

Šestý řádek: NOP, kód je zase 00, je uložen na adrese 0102h a počítadlo se posouvá…

Na sedmém řádku je návěští NAVEST. Je mu tedy přiřazena hodnota interního počítadla adres (0103h). Na tomtéž řádku je i instrukce. Její operační kód je uložen na tuto adresu a pokračujeme dál…

Na samotném konci listingu vidíte výpis všech návěští. _PC je interní počítadlo adres (které skončilo na hodnotě 0104h), no a návěští ZACATEK a NAVEST mají ty hodnoty, jaké jsme si popsali výše.

Základy jsou tedy jasné, pojďme na skutečné programování v assembleru!

8080 – první program

Napíšeme si první program. Nekecám, opravdu první program v assembleru pro procesor 8080!

Nebude to Hello world, bude to něco sofistikovanějšího. Sečteme dvě čísla. Třeba 18H a 23H. (Výsledek je 03BH, kdo nevěří, ať tam běží.) Nebojte se toho, že nezačínám od základů, že ještě žádnou instrukci neznáte (ne, znáte jednu, z minulé lekce víte o NOP) ani že neřeknu hned všechno (třeba zamlčím existenci „registru“ M). Jen si odskočíme, abyste viděli, jak se v assembleru pracuje, a už v příští lekci to budeme brát systematicky instrukci po instrukci…

Ale teď: **Jak sečíst dvě čísla v assembleru?** Asi už tušíte, že to nebude prosté jako napsat „18+23“. Procesor 8080 totiž nic takového jako „sečti dvě čísla, která ti sem napíšu“ neumí. Co umí?

Umí sečíst dvě čísla, to je v pořádku. K sečtení slouží instrukce ADD, se kterou se seznámíme v dalších lekcích. ADD má jeden parametr, a tím je jméno registru. Vzpomeňte si na kapitolu o architektuře 8080: máme k dispozici registry B, C, D, E, H, L a akumulátor A. Instrukce ADD má symbolický tvar „ADD r“ (kde „r“ je právě to jméno registru) a její funkce je následující: Vezme obsah registru A, přičte k němu obsah zadaného registru, a výsledek uloží zase do A. Nějak takhle:

INSTRUKCE	FUNKCE
ADD B	A = A + B
ADD C	A = A + C
ADD D	A = A + D
ADD E	A = A + E
ADD H	A = A + H
ADD L	A = A + L

INSTRUKCE	FUNKCE
ADD A	A = A + A

Postup bude tedy následující: Do jednoho registru si uložíme první číslo, tedy 18H, do druhého registru druhé číslo (23H), a pak je pomocí instrukce ADD sečteme. Protože ADD vyžaduje, aby jeden ze sčítanců byl v registru A, ušetříme si práci tím, že jedno z čísel umístíme rovnou do A, druhé třeba do B.

Jak do registrů nahrajeme nějaké číslo? Použijeme k tomu instrukci MVI (Move Immediate). Tato instrukce má dva parametry – první je jméno registru, druhý je hodnota 0-255, tedy 1 bajt. Symbolicky zapsáno je to „MVI r, n8" (n8 se označuje osmibitové číslo). Všimněte si, že pořadí parametrů je podobné jako ve vyšších jazycích, tedy vlevo je to, KAM se ukládá, vpravo CO se tam má uložit. V našem případě použijeme dvojici instrukcí „MVI A, 23H" a „MVI B, 18H".

Celý program bude vypadat takto:

```
1    ORG 0
2    MVI A, 023h
3    MVI B, 018h
4    ADD B
```

Co se tam děje? Nejprve říkáme překladači, ať si nastaví adresu 0, od té adresy se budou ukládat instrukce do paměti. Pak je instrukce, která do registru A uloží hodnotu 23h, pak do registru B uložíme 18h, a pak vykonáme ADD B, což je, jak už víme, ekvivalent „A = A + B".

Ukažme si ještě listing:

```
1    0000            .ORG   0
2    0000   3E 23    MVI   A,023h
3    0002   06 18    MVI   B,018h
4    0004   80       ADD   B
```

Instrukce MVI zabírá vždy dva bajty. První bajt je kód instrukce (3Eh = MVI A; 06h = MVI B), druhý bajt je hodnota, která se má do registru nahrát.

Tak, to by bylo. A teď otázka nejdůležitější: Bude to fungovat? Bude. V tuhle chvíli vám nezbývá nic jiného, než mi věřit. Anebo si to vyzkoušet. Buď použijte svůj oblíbený assembler, spusťte si kód na svém počítači (třeba PMI-80) – no a nebo využijte ASM80, kde si můžete kód napsat, přeložit a odladit v emulátoru, a to všechno přímo v prohlížeči (tedy, pokud máte aspoň trošku modernější prohlížeč)! V okénku, co je pod tímto textem, máte k dispozici editor, překladač i debugger, takže si můžete výše zmíněný kód vyzkoušet. Napište ho do levého editoru (on už tam je předvyplněný…) a kliknutím na COMPILE ho přeložíte. V pravém editoru se ukáže listing. Funkčnost můžete otestovat kliknutím na Emulator – levý editor zmizí a místo něj se objeví okno s výpisem paměti, s výpisem jednotlivých registrů a s tlačítky Single step, 50 steps, Animate a Stop. Po vstupu do emulátoru je PC nastaven na 0000, procesor je připraven na první instrukci. Kliknutím na Single step ji provedete. Všimněte si, že se v registru A objevilo 23 a kurzor se posunul na další instrukci. Opět klikněte na Single Step, v registru B se objeví 18, no a po dalším kliknutí na Single step se provede i ta sčítací instrukce. Takto si můžete krokovat svoje programy a dívat se, jestli dělají to, co dělat mají.

Náš program nic víc neudělá. Když si ho odkrokujete, dostanete výsledek v registru A a pak se nestane nic.

Ale co, Řím taky nepostavili za den!

No related posts.

Instrukce 8080 – adresní módy a registry

Jak instrukce ví, kde má vzít data, se kterými pracuje?

Některé instrukce (jako už zmíněný NOP) nevyžadují žádné parametry a nepracují s žádnými daty. Tam je to jasné. Ovšem většina instrukcí *něco* dělá, to znamená, že potřebuje odněkud vzít nějaké údaje, někam je uložit atd. Procesor 8080 má čtyři různé způsoby, jak říct, kde se data nacházejí.

PŘÍMÝ OPERAND

Tímto způsobem zapíšeme přímo hodnotu do kódu a procesor ji použije. S přímým operandem pracuje např. instrukce MVI, kterou jsme si ukázali v minulé kapitole, Součástí instrukce je přímo hodnota, která je použita.

PŘÍMÉ ADRESOVÁNÍ

Je podobné předchozímu. I v tomto případě je součástí instrukce hodnota, která ale není použita tak, jak je, ale je brána jako adresa (v paměti nebo vstupně-výstupní brány). Například instrukce „LDA 554Ah" používá právě tento typ adresování. Procesor udělá to, že hodnotu 554Ah vezme jako adresu místa v paměti, přečte z něj jeden bajt a ten uloží do registru A.

ADRESOVÁNÍ REGISTREM

Součástí instrukce je jméno registru, registrového páru nebo dvojice registrů, se kterými se operace provádí.

NEPŘÍMÉ ADRESOVÁNÍ REGISTREM

Instrukce bere hodnotu v dvojici registrů (BC, DE, HL) nebo v registru SP jako adresu do paměti. Pracuje pak s obsahem na této adrese.

Některé instrukce pracují s tzv. „implicitním operandem", což znamená, že operand (jeden z operandů) je daný už samotnou instrukcí. Například zmíněná instrukce ADD používá dva operandy – jeden je vždy A a druhý je určen parametrem.

REGISTRY

U osmibitových instrukcí můžeme použít libovolný osmibitový registr – tedy A (akumulátor), B, C, D, E, H, L a M.

Moment! Registr M? Nikde jsem se o něm nezmínil… Není to nějaký renonc?

Není. „Registr M" (memory) je ve skutečnosti buňka v paměti na adrese, která je uložena v registrech H a L. Pokud je v registru H hodnota 12h a v registru L hodnota 34h, bude instrukce, která pracuje s „registrem M", pracovat ve skutečnosti s obsahem paměti na adrese 1234h. Instrukce ADD M přičte k obsahu registru A hodnotu z paměti na adrese 1234h.

Všimněte si, že nelze přímo přistupovat k registru F. Ono to většinou není k ničemu dobré, ale kdybyste opravdu potřebovali znát jeho hodnotu, dá se to obejít takovým trikem, který si popíšeme později.

Pokud instrukce pracuje s šestnáctibitovou hodnotou, využívá šestnáctibitový registr SP nebo dvojice registrů B, D, H. „Dvojregistr" B jsou vlastně registry B a C, v B je vyšší bajt, v C nižší. Obdobně pro D a H. Instrukce pro práci se zásobníkem používají „šestnáctibitový registr" PSW, což je ve skutečnosti dvojice registrů A a F (A vyšší byte, F nižší).

MALÝ INDIÁN

Procesory Intel, Zilog Z80 i 6502 používají zápis vícebytových čísel ve formátu „Little Endian". Znamená to, že na nižší adrese je méně významná část čísla. Příklad: Na adresu 0100h uložíme naši obligátní hodnotu 1234h. Paměť je organizovaná po bajtech,

hodnota je dvoubajtová (16 bitů), jak to tedy uložit? Použijeme dvě buňky paměti, 0100h a 0101h. Na tu nižší přijde nižší část čísla (34h), na tu vyšší zase vyšší část (12h). V paměti to tedy bude vypadat nějak takto:
0100h: 34h
0101h: 12h
Některé procesory (z těch známějších především procesory od Motoroly) používají opačný způsob („Big Endian"), kdy se části čísla zapisují do paměti od nejvýznamnějšího bajtu.

Instrukce 8080 – přesuny dat

První skupina instrukcí, kterou si probereme, přesouvá data. Z registrů, do registrů, i z paměti a do paměti…
Až budete psát programy v assembleru, zjistíte, že nejvíc kódu nezabírají nějaké hyper super složité výpočty, ale přesouvání dat odněkud někam. V podstatě většina činností, které mikroprocesorový systém navenek dělá, je z valné části *přesouvání dat*. Samotného počítání je minimum. Takže je jen logické, že začneme právě u těchto instrukcí.

MOV

Univerzální instrukce pro kopírování obsahu z jednoho registru do druhého. Instrukce má dva operandy – názvy osmibitových registrů (z minulého dílu víme, že to jsou A, B, C, D, E, H, L a M). Jako první se uvádí cílový, jako druhý zdrojový. MOV A,B tedy vezme obsah registru B a zkopíruje ho do registru A. MOV A,C zkopíruje obsah z registru C do registru A. MOV C,A přesně naopak – z registru A do registru C.
Ptáte se, kolik je kombinací? No všechny dostupné – tedy 8 x 8 = 64. Čímž neříkám, že jsou všechny smysluplné či funkční. Instrukce MOV A,A je samosebou platná, zkopíruje obsah z registru A do registru A, ale upřímně: k čemu to je?
Znovu připomínám, že M je „pseudoregistr", který ve skutečnosti odkazuje na místo v paměti s adresou, která je uložená v dvojici registrů H, L. Takže MOV A,M znamená „vezmi obsah z paměti na adrese, která je v registrech H,L, a zkopíruj ho do registru A". MOV M,E znamená „vezmi obsah registru E a zkopíruj ho do paměti na adresu, tkerá je v registrech H,L". No a MOV M,M – to je výjimka. Taková instrukce neexistuje. Její operační kód zaujímá instrukce HLT, která zastaví procesor.

MVI

Tato instrukce vloží do osmibitového registru přímo zadanou hodnotu.

LXI

Tato instrukce vloží do registrového páru B, D, H nebo registru SP přímo zadanou šestnáctibitovou hodnotu.

LDA

Instrukce má jeden parametr – šestnáctibitovou adresu. Obsah paměti na téhle adrese zkopíruje do registru A

STA

Protipól předchozí instrukce – ukládá hodnotu z registru A do paměti na adresu zadanou jako operand.
A nyní si prosvištime nektera sloviška a cele fěty – pardon, zkusíme si, jak tyhle instrukce pracují. Opět k tomu použijeme interaktivní techniku – assembler ASM80.
Kód přeložte a spusťte emulaci. První instrukce je MVI A, 10. Po provedení prvního kroku bude tedy v registru A hodnota… aha! Je tam 0Ah. Emulátor totiž ukazuje všechny hodnoty hexadecimálně, zatímco v kódu je zapsaná konstanta 10, bez H na konci, takže je to tedy desítková hodnota, a 10 desítkově je v šestnáctkové soustavě 0A. (A nestěžujte si, různé způsoby zápisu konstant jsme už brali!)
Takže tedy znovu: První instrukce uloží do registru A hodnotu 0Ah (=10), druhá instrukce uloží do registru B hodnotu 14h (=20). Třetí instrukce je LXI H, *číslo* – mrkněte se o kousek výš… jasně! Tahle instrukce uloží zadané číslo do dvojice registrů HL. Tady je to 0040h, takže do H půjde vyšší (00) a do L nižší (40h) část čísla.
Na dalším řádku je instrukce MOV M,A. Z toho, co o ní víme, by se mělo stát následující: Obsah registru A (což je teď těch 0Ah) se zkopíruje do paměti na adresu, která je uložená v registrech HL. Tak schválně – v registrech HL je uložena hodnota 0040h, takže by se v paměti na adrese 0040h měla objevit hodnota 0Ah. Zkuste si sami – vidíte, že vlevo nahoře, v oblasti nadepsané MEMORY, se na adrese 0040h objeví právě ta hodnota 0Ah.
Bystřejší si všimli, že od adresy 0 jsou v paměti nějaká čísla. Ti, co jsou ještě bystřejší, už vědí: To je přeci ten náš program! A právě do té oblasti sáhne další instrukce – LDA 0003h vezme obsah paměti na adrese 0003h, což je zrovna shodou okolností číslo 14h (které je součástí instrukce MVI B, 20) a uloží ho do registru A.
Poslední instrukce pak uloží obsah registru A na adresu 0041h, takže od adresy 0040h budete mít vedle sebe hezky 0Ah a 14h. Máte? Skvělé, pojďme na další instrukce!

LDAX

Tak jako LDA vezme hodnotu z paměti na adrese, kterou mu zadáme, a uloží ji do A, tak i LDAX vezme hodnotu z paměti a uloží ji do A. Adresu, se kterou se bude pracovat, najde v registrovém páru B nebo D (tedy v registrech B, C, resp. D, E).

STAX

Jako má LDA svoje STA, tak má LDAX svůj STAX. Funguje stejně jako LDAX, jen přenos dat probíhá v opačném směru.

LHLD

Tahle instrukce opět pracuje s dvojicí bajtů. Jako parametr dostane adresu a vykoná následující: Do registru L zkopíruje hodnotu z paměti na dané adrese a do registru H zkopíruje hodnotu z paměti na adrese o 1 vyšší.

SHLD

Už to tak je. Instrukce L*něco* mají svou obdobu v podobě S*něco*. Tam, kde L z paměti čte (load), tam S do paměti zapisuje (store). Takže SHLD uloží obsah registru L na zadanou adresu, obsah registru H na adresu o 1 vyšší.
Všimněte si, že LHLD i SHLD dodržují pravidlo „malého indiána" – na nižší adresu jde méně významný bajt.

XCHG

Poslední instrukce pro přesun dat je tak trošku výjimka. Nemá žádné operandy – to, s čím pracuje, je dáno implicitně (jsou to dvojregistry D a H). A na rozdíl od předchozích instrukcí, které kopírovaly hodnoty, tato instrukce je vyměňuje. XCHG vymění obsah registrů D a E za obsah registrů H a L. Tedy to, co bylo v D, bude teď v H – a naopak. Kdybychom udělali třeba MOV H,D a MOV L,E, dosáhli bychom něčeho jiného obsah registrů D a E by se nevyměnil, ale ZKOPÍROVAL, takže by bylo v D totéž co v H a v E totéž co v L.
Pokud chceme vyměnit údaje ve dvou registrech, musíme použít pomocný registr. Například výměnu obsahu registrů B a C zařídíme takto:

```
1    MOV A,B
2    MOV B,C
3    MOV C,A
```

Tedy přes registr A. Jeho původní hodnotu samozřejmě ztratíme…
A opět drobná ukázka. Až si pohrajete, můžete si zkusit jedno cvičení. První samostatný program. Poslyšte zadání, je jednoduché: **Program prohodí obsah registrů BC a HL.**
No related posts.

8080 – příznaky a zásobník

Procesor potřebuje vědět, jak dopadla ta která instrukce, a občas si potřebuje nějaká data odložit na později.

PŘÍZNAKY

O registru F jsem se už zmiňoval. Je to takový lehce mystický registr, do kterého se nedá přímo zapisovat a ze kterého se nedá přímo číst. Psal jsem, že v něm jsou uložené takzvané *příznaky*. Co to je?

Některé instrukce, převážně ty, co něco počítají, mohou dát najevo, že výsledek je v určitém formátu. Například že je nulový, záporný, jakou má paritu, nebo že při výpočtu přetekl výsledek z rozsahu 0-255. A přesně tyto informace ukládá jako *příznakové bity* do registru F.

U procesoru 8080 je příznakových bitů 5 a jsou v registru uloženy takto:

BIT	7	6	5	4	3	2	1	0
PŘÍZNAK	S	Z	0	AC	0	P	1	CY

- S (sign) informuje o znaménku výsledku. Je-li výsledek kladný, je to 0, je-li výsledek záporný, je to 1
- Z (zero) je roven 1 v případě, že výsledek je nula. Pokud je nenulový, je Z = 0
- AC (auxiliary carry) je roven 1, pokud při operaci došlo k přenosu přes polovinu bajtu (mezi nižší a vyšší čtveřicí)
- P (parity) je nastaven vždy tak, aby doplnil počet jedniček ve výsledku na lichý (tedy 0, je-li lichý, 1, je-li sudý) – viz též paritní bit.
- CY (carry) je 1, pokud dojde k přenosu z nejvyššího bitu.

Termín „přenos" je možná nejasný, pojďme si ho upřesnit. Co se stane, když budu sčítat čísla 0FAh a 0Ah (250 a 10) v osmibitovém registru? Výsledek je 260, hexadecimálně 104h – a to je číslo, které má devět bitů. Devátý (nejvyšší) bit se do registru nevejde, tam zůstanou jen bity 0-7 (tedy osm bitů). Pokud taková situace nastane, je příznak CY nastaven na jedničku. Tento příznak je velmi užitečný, pokud sčítáte vícebajtová čísla. Sečtete nejprve čísla na nižších řádech, a pokud je náhodou větší než možný rozsah, *přenesete si*jedničku do vyššího řádu.

Tentýž příznak funguje i pro odčítání – zde zase funguje jako označení „výpůjčky" do nejvyššího řádu v případě, že výsledek je menší než 0. Když od čísla 04h odečteme číslo 05h, bude výsledek 0FFh (nezapomeňte, že hodnoty jdou za sebou a po nule je 0FFh). Příznak CY bude nastaven („půjčili" jsme si jedničku), příznak S bude nastaven (výsledek je záporný).

Ve skutečnosti je -1 a 255 jedno a totéž číslo a je jen na nás, jestli se na něj díváme jako na číslo se znaménkem, nebo na číslo bez znaménka. -2 je 254, -3 je 253 atd. Pokud chceme v binární soustavě převést číslo na jeho zápornou hodnotu, uděláme bitovou negaci (zaměníme 0 za 1 a opačně) a přičteme jedničku (procesor na to samosebou má vhodnou instrukci).

A k čemu jsou příznaky dobré? Jednak pomáhají s výpočty, a jednak se můžete podle výsledků rozhodovat a provádět podmíněné skoky a volání podprogramů.

ZÁSOBNÍK

Občas procesor potřebuje, jak jsem už psal, nějaká data „odložit" na později. Ať už to jsou data, co si potřebuje uložit programátor, nebo třeba adresa, kam se má program vrátit z podprogramu. Používá k tomu strukturu zvanou zásobník (stack).

Stack je struktura typu LIFO (Last In – First Out, tedy co se poslední uloží, to se první vrátí). Implementace zásobníku je jednoduchá: V paměti si vyhradíte oblast, a adresu jejího konce +1 uložíte do registru SP (Stack Pointer). Zásobník je připravený.

Do zásobníku se **vždy ukládají dva bajty**, tedy buď adresa, nebo celý registrový pár. Pokud na zásobník uložíme pár HL, stane se toto: Procesor sníží SP o 1, na tuto adresu uloží obsah registru H, pak zase sníží SP o 1 a na tuto adresu uloží obsah registru L. Teď uložíme třeba pár BC: Procesor sníží SP o 1, na tuto adresu uloží obsah registru B, pak zase sníží SP o 1 a na tuto adresu uloží obsah registru C.

Když teď vyzvedneme hodnotu ze zásobníku – třeba do registrů DE – procesor vezme obsah z adresy SP, uloží ho do registru E, SP zvýší o 1, vezme obsah z adresy SP, uloží ho do registru D a opět zvýší SP o 1. SP teď ukazuje na předchozí položku a v DE máme tu poslední uloženou. Což byla, shodou okolností, zrovna hodnota z registrů B a C – nikde není psáno, že se musí údaje vybrat do téhož registru, z jakého byly uloženy.

Zásobník se tedy posouvá od vyšších adres k nižším. Proto se umisťuje na konec RAM, aby měl dost prostoru kam růst. Což s sebou nese riziko, že pokud uložíte do zásobníku příliš mnoho hodnot, klesne SP až tak, že další uložená data můžou přepsat váš program (tomuto stavu se říká *přetečení zásobníku*, stack overflow). Pokud umístíte zásobník POD program, může se zase stát, že poklesne až třeba do oblasti, kde je ROM. Oba stavy jsou chybné, znamenají téměř stoprocentně pád programu a většinou nastávají, když se program dostane do nekonečné smyčky, ve které se odskakuje nebo ukládá.

PUSH A POP

K uložení obsahu registrů do zásobníku slouží instrukce PUSH s jedním parametrem, a tím je jméno dvojice registrů (B, D, H), nebo „PSW" – o kterém jsem se už zmiňoval. PSW (Program Status Word) je dvojice registrů A a F. Pro vybrání hodnot ze zásobníku slouží instrukce POP (a stejné parametry). Pojďme si zaexperimentovat: nastavíme si zásobník do paměti a budeme do něj ukládat a zase vybírat data.

Všimněte si nejprve toho, jak se do paměti ukládají data, když se vykonává instrukce PUSH. Později, při načítání zpět, je načítáme v jiném pořadí, takže registry nemají původní obsah.

Na konci minulé lekce jsem zadal úkol: vymyslet způsob, jak prohodit obsah registrových párů BC a HL. Předpokládám, že jste napsali něco takového:

```
1    MOV A,B
2    MOV B,H
3    MOV H,A
4    MOV A,C
5    MOV C,L
6    MOV L,A
```

Což je správně, ale přijdeme tím o obsah, který je v registru A. Pokud nám to nevadí, není co řešit. Ale pokud by nám to vadilo, dáme na začátek PUSH PSW a na konec POP PSW. Samozřejmě, spoléháme na to, že zásobník je správně nastavený, ale vzhledem k tomu, jak často je používaný, tak je to jedna z prvních věcí, co slušný programátor udělá.

No a s informacemi z této lekce můžete stejné zadání vyřešit třeba takto:

```
1    PUSH B
2    PUSH H
3    POP B
```

4	POP H

SPHL

Tato instrukce naplní registr SP obsahem dvojice registrů HL. Tedy SP = HL. Ptáte se, jak se zařídí opak, tedy jak zjistíte hodnotu registru SP a uložíte ji do HL? Tipujete instrukci HLSP? Ne, žádná taková není. Musíte použít konstrukci LXI H,0 a DAD SP (o instrukci DAD si řekneme víc v další lekci).

XTHL

Další instrukce, která vyměňuje hodnoty dvojic registrů. XTHL vymění hodnotu dvojice registrů HL s „poslední uloženou hodnotou na zásobníku". Tedy: poslední hodnota, uložená na zásobníku, se ocitne v HL, a hodnota HL bude poslední hodnota na zásobníku.

Instrukce 8080 – aritmetika

Je to počítač, ne? Tak to má taky počítat!

Procesor samosebou není živ jen přesouváním dat tam a zpátky. Občas taky musí něco spočítat. K tomu mu slouží aritmetické instrukce pro sčítání, odčítání a porovnání. (Kde je násobení, dělení a hyperbolický sinus? Dozvíte se později!)

ADD

Instrukce ADD r vezme obsah v osmibitovém registru r (tedy A, B, C, D, E, H, L nebo M, což je ve skutečnosti obsah paměti, ale to už jsem psal asi pětkrát…) a přičte ho k registru A. Výsledek sčítání uloží do registru A. Pokud vám připadá, že se registr A používá nějak často, tak se vám to nezdá, je to opravdu tak, a protože se k němu vztahují skoro všechny aritmetické i logické instrukce, tak se mu říká „akumulátor".

Po sčítání nastaví patřičně příznaky S, Z, AC, P a CY – viz popis funkce příznaků v minulé lekci.

Pojďme se podívat na pár příkladů sčítání a toho, jak ovlivní příznaky. Všechna čísla budou hexadecimálně.

SČÍTANEC 1	SČÍTANEC 2	VÝSLEDEK	S	Z	AC	CY	POZNÁMKA
03	05	08	0	0	0	0	Nic zvláštního, vešli jsme se dokonce do jedné pozice
09	08	11	0	0	1	0	Při sčítání bylo třeba přenést jedničku z nejnižší pozice do vyšší
FF	01	00	0	1	1	1	Výsledek je 0 (Z), přetekl rozsah registru (CY) a přenášela se jednička mezi čtveřicemi bitů (AC)
F0	05	F5	1	0	0	0	Pokud používáme aritmetiku se znaménkem, tak je výsledek záporný (S)

SE ZNAMÉNKEM…?

Jak to vlastně je, když chceme použít místo čísel bez znaménka (0-255) čísla se znaménkem (-128 až 127)? Kde to dáme procesoru najevo? Odpověď zní: Nikde! Procesoru to je úplně jedno. Vezměme si poslední řádek z předchozí tabulky. Pokud se bude jednat o čísla bez znaménka, tak sčítáme hodnotu 240 (F0h) a 5. To je dohromady 245, tedy hexadecimálně F5h. Když je budeme vnímat jako čísla se znaménkem, tak sčítáme -16 (F0h) a +5. Výsledek je -11, což je hexadecimálně F5h.

Předposlední řádek je podobný: při sčítání bez znaménka je FFh+01h=100h, což se do osmi bitů nevejde, máme tedy výsledek 00 a nastavený přenos (1). Pokud sčítáme se znaménkem, sčítáme -1 (FFh) a +1 a výsledek je – opět nula!

To, jestli pracujeme s čísly se znaménkem nebo bez znaménka, si musíme určit coby programátoři sami. Procesor je zpracovává stále stejně a je jen na nás, jak je interpretujeme. Jen je důležité je interpretovat vždy stejně.

ADI

Často potřebujeme přičíst nějakou předem známou konstantu. Pomocí ADD bychom si ji museli nejprve uložit do některého registru, a ten pak přičíst. Naštěstí to jde rychleji, pomocí jedné instrukce ADI. Ta má jediný parametr, a tím je osmibitová konstanta. Tu přičte k registru A. Zbytek, tedy nastavování příznaků apod., platí stejně jako u instrukce ADD.

ADC

Funguje stejně jako instrukce ADD, ale k výsledku přičte ještě hodnotu příznaku CY. Hodí se to například při sčítání větších čísel. Představme si, že máme dvě šestnáctibitová čísla, jedno v registrech BC, druhé v registrech DE, a chceme je sečíst a výsledek uložit do registrů BC. Takhle jednoduše to nejde, žádnou instrukci na to nemáme, takže to musíme udělat krok po kroku. Nejprve si sečteme čísla na nižších pozicích (C + E) a výsledek uložíme do C. Ovšem pokud součet těchto číslic bude větší než 255, tak musíme při sčítání vyšší pozice přičíst (přenesenou) jedničku! Což je bez problémů, protože, jak víme: ADD nastaví správně příznak CY, a ADC ho dokáže započítat. Součet tedy naprogramujeme třeba takto:

1	mov a,c
2	add e
3	mov c,a
4	mov a,b

5	adc d
6	mov b,a

Naštěstí instrukce MOV nijak neovlivňuje příznaky, takže stav příznaku CY tak, jak ho nastaví instrukce ADD, se až do provedení instrukce ADC nezmění.

Funguje to? Ověříme v emulátoru! Sečteme si čísla 1000 a 2000 (03E8h a 07D0h), výsledek by měl být 3000 (0BB8h)…

Všimněte si, že při součtu nižších řádů (E8h+D0h) je nastaven bit přenosu, protože výsledek je větší než FFh.

ACI

Jako má ADD své ADI, tak má ADC své ACI. K reistru A přičte zadanou konstantu a hodnotu příznaku CY.

SUB, SUI

Instrukce SUB od hodnoty v registru A odečte hodnotu zadaného registru a výsledek uloží do A. SUI dělá totéž, ale s osmibitovou konstantou. Pokud je výsledek menší než 0, tak se pokračuje od FFh dolů (-1 = FFh, -2 = FEh, …) a je nastaven příznak CY, který má tentokrát význam „výpůjčky“ (borrow).

SBB, SBI

Totéž jako SUB, SUI, ale tentokrát od výsledku odečte 1, pokud je nastaven příznak CY.

CMP, CPI

Tyto instrukce porovnají obsah registru A s jiným registrem (CMP) nebo konstantou (CPI) a nastaví příznakové bity takto:

POROVNÁNÍ	S	Z	CY
A=n	0	1	0
A>n	0	0	0
A<n	1	0	1

Podmíněné instrukce skoku se pak mohou podle těchto hodnot rozhodnout.

Zajímavost: Instrukce CMP a CPI vnitřně dělají totéž jako instrukce SUB, SUI, tedy odečtou od A hodnotu parametru, nastaví příznaky, ale výsledek neuloží do A, ale zahodí ho.

INR, DCR

Velmi častá operace je zvýšení hodnoty o 1 nebo naopak odečtení 1. Vyšší jazyky, které vyšly z jazyka C, na to mají i operátory „++“ a „–„. INR přičte k registru 1, DCR odečte 1. Instrukce ovlivní příznaky S, Z, AC a P, CY ponechá tak jak je (v tom se liší od ADI 1 / SUI 1)

INX, DCX

INR a DCR pracují s osmibitovým registrem, INX a DCX pracují s dvojicí registrů, jako by byl jeden šestnáctibitový. Parametrem může být B, D, H nebo SP.

DAD

Sčítání v šestnácti bitech. K obsahu registrů HL přičte BC, DE, HL nebo SP.

DAA

Tahle instrukce souvisí s čísly ve formátu BCD. Jen ji tu takhle zmíním, že existuje, a její popis si nechám na jindy, adekvátně jejímu významu…

A jak se tedy násobí a dělí? Dostaneme se k tomu, nebojte, ale teď si odskočíme. Jump!

Instrukce 8080 – skoky

Kdyby nebyly skoky, tak by program prostě jen jel, jel, jel… a na konci by jel zase od začátku. Něco jako flašinet.

Předpokládám, že znáte nějaký vyšší jazyk. Pravděpodobně ten jazyk obsahuje příkaz, kterému se vyhýbáte jako čert kříži, i když tam z nějakého důvodu je. Pokud jste mladší ročníky a začali jste s nějakým moderním jazykem, tak před vámi možná dokonce existenci toho příkazu úplně zatajili. Každopádně vám řekli, že to je zlo, nesystémovost, vede to ke špatným věcem a vůbec – pokaždé, když použijete GOTO, tak Bůh zabije koťátko.

Vítejte ve světě, kde je GOTO tím hlavním nástrojem.

Totiž, aby bylo jasno: V assembleru nejsou objekty, třídy, metody… Dokonce tam nejsou ani řídicí struktury. Není tam cyklus FOR, WHILE, není ani IF / ELSE. Ba co hůř – nejsou ani datové typy. Zapomeňte na řetězce, zapomeňte na datové struktury, jediné, co máte, je číslo. Zapomeňte na lokální proměnné. Nic z toho v assembleru nedostanete. Je to tvrdý svět – člověk proti křemíku. Nic vám neodpustí, nic za vás nedomyslí. Ale zase má jiné výhody: můžete program optimalizovat do nejmenších podrobností a detailů a za týden práce máte vymazlený program, co byste ve vyšším jazyce psali jedno odpoledne. Možná je rychlejší, možná je kratší, každopádně v něm můžou číhat velmi zákeřné chyby, které nemáte ve vyšším jazyce šanci udělat.

„Ale jak tedy udělám smyčku? Jak cyklus? Jak vnořím podmínky? Vždyť bez těchto základních konstrukcí nelze programovat!“ To si povíme za chvíli, teď si ukážeme, co máme k dispozici.

JMP

To je ono, to zlé GOTO. Tady se jmenuje JMP (jako že „jump“) a má jeden parametr. Tím je šestnáctibitové číslo – adresa, na kterou se skočí, tj. ze které bude procesor číst další instrukci. Vnitřně funguje tak, že to šestnáctibitové číslo, které je součástí instrukce, uloží do registru PC. Ten obsahuje (už jsme si to říkali) adresu, ze které se čte aktuální instrukce. Když tedy do PC uložíme číslo 0123h, bude se další instrukce číst z adresy 0123h.

Když takový program píšete, musíte znát adresu, kde bude ta cílová instrukce uložena. Naštěstí si to nemusíte počítat, to za vás dělá překladač. Vy jen překladači řeknete, kde bude ležet první instrukce (.org, vzpomínáte?), a zbytek adres si dopočítá za vás. Vy si můžete jednotlivé body pojmenovat symbolickými názvy a překladač je za vás správně vyhodnotí. Nějak takhle:

Což je samosebou zcela smyšlený příklad, ve skutečnosti by ho takhle nikdo nenapsal, ale je hezky vidět, jak instrukce JMP skáčou dopředu i dozadu. Zkuste si ho chvíli krokovat a uvidíte…

CALL, RET

Víte, nebyl jsem tak úplně upřímný. GOTO není jediné, co máte k dispozici. Ještě můžete použít GOSUB / RETURN. Pokud znáte BASIC, víte, co tyto instrukce dělají – volají podprogram. Ve vyšších jazycích je to *něco jako procedura / funkce*, ale opravdu jen velmi vzdáleně: nejsou žádné lokální proměnné, nejsou parametry, nejsou návratové hodnoty. Je tedy konvence, která říká, že podprogram, pokud něco vrací, tak by to měl vracet v registru A nebo HL, ale není to vůbec nezbytné – když na to přijde, tak váš podprogram může vrátit údaje v registru E, B a návdavkem ještě v příznaku CY.

Pokud v podprogramu změníte hodnotu registru, zůstane změněná i po návratu do hlavního programu. Musíte s tím počítat, a pokud si nechcete přepisovat hodnoty v registrech, tak si nezapomeňte na začátku podprogramu potřebné hodnoty uložit

na zásobník (PUSH) a před návratem zase ze zásobníku vybrat (POP). Ale pozor! Vždycky musíte vybrat přesně tolik údajů, kolik jste jich uložili, protože jinak se stanou strašlivé věci. Jaké? Hned si ukážeme.
Jak taková instrukce CALL funguje? CALL má jeden parametr, kterým je šestnáctibitová adresa, jako u instrukce JMP. CALL nejprve vezme adresu následující instrukce (říká se jí „návratová adresa"), tu uloží na zásobník, a pak provede skok na zadanou adresu, stejně jako JMP. Instrukce RET vezme hodnotu ze zásobníku (měla by tam být ta návratová adresa, kterou tam uložila instrukce CALL) a skočí na ni. provádění programu tak pokračuje za tou instrukcí CALL, která volala podprogram.
Pokud si v podprogramu uložíte obsah registru (třeba HL pomocí PUSH H), bude na zásobníku HL a pod ním návratová adresa. Před návratem tedy musíte HL zase odebrat (nejčastěji pomocí POP něco), aby se RET vrátila tam, kam má. Pokud byste hodnotu neodebrali, RET by obsah zásobníku, tj. uloženou hodnotu registrů HL, považovala za návratovou adresu a skočila by tam. Naopak pokud byste si ze zásobníku vyzvedli víc údajů, než jste do něj vložili, byla by mezi nima i návratová adresa, takže RET by se vrátila úplně někam jinam.
Ano, OBČAS se takové triky dělají, ale upozorňuju dopředu: dělají je programátoři, kteří vědí co dělají a proč. Mají své místo ve chvíli, kdy hledáte, jak ušetřit nějaký ten bajt / nějaký ten takt procesoru, ale v naprosté většině případů je přebývající POP nebo PUSH chyba, která povede k pádu celého systému.

PCHL

Prosím, nečíst P-Ch-L (podle vzoru *Pche*!), ale P-C-H-L. Když to přečtete správně, napoví vám, jak funguje. Do registru PC uloží obsah HL (PC=HL). Je to tedy něco jako JMP na adresu, uloženou v HL.

RST

RST je taková zkratka pro CALL na některé oblíbené adresy. Instrukcí RST je 8, číslovaných 0-7, a skáčou na adresu 8*N (kde N je to číslo). RST 0 je tedy totéž co CALL 0, RST 1 je CALL 8, RST 2 je CALL 16 (CALL 0010h) a tak dál… až RST 7 je totéž jako CALL 0038h. Oproti instrukci CALL, která zabírá 3 bajty (1 bajt instrukční kód 2 bajty adresa) má instrukce RST jen 1 bajt, navíc se provede rychleji než CALL. V mnoha systémech jsou proto na těchto adresách připravené často používané podprogramy. Namátkou ZX Spectrum používá RST 2 (v mnemonice procesoru Z80 označená jako RST 10h) pro vypsání znaku s ASCII kódem, který je v registru A, na výstup (obrazovku).

PODMÍNĚNÉ SKOKY

Je hezké, že program skáče jako srnka, ale mnohem lepší a užitečnější by bylo, kdyby mohl nějak reagovat na to, co se počítá. K tomu slouží podmíněné instrukce Jcond, Ccond a Rcond. „Cond" je označení podmínky, která se má testovat. Vždy se testuje hodnota některého příznaku S, Z, CY nebo P.

COND	PROVEDE SE, POKUD…
C	CY = 1
NC	CY = 0
Z	Z = 1
NZ	Z = 0
M	S = 1
P	S = 0
PE	P = 1
PO	P = 0

Podmínky mají i své mnemotechnické názvy: Carry, Not Carry, Zero, Not Zero, Minus, Plus, Parity Even, Parity Odd.
Podmíněné skoky, odvozené od JMP, jsou tedy JC, JNC, JZ, JNZ, JM, JP, JPE a JPO. Pracují tak, že otestují příslušný podmínkový bit. Pokud platí podmínka, provede se skok, pokud není splněna, pokračuje se dál. Například instrukce JZ 0123h zkontroluje nejprve, jestli je příznak Z=1. Pokud ano, skočí na adresu 0123h, pokud ne, pokračuje dál.
Ano, takhle nějak funguje IF cond THEN GOTO x. „Pokud je splněno, skoč tam a tam." V praxi budete chtít často zapsat něco jako „Pokud je výsledek 0, udělej ještě to a to…" (tedy chování obdobné známé konstrukci IF) V assembleru to vyřešíte tak, že zapíšete „Pokud výsledek není 0, tak přeskoč až za blok příkazů".

```
1   ; IF (CY) {INC A, DEC D}
2
3   JNC navesti ; Pokud podmínka neplatí, přeskoč
4   INC A       ; blok příkazů
5   DEC D
6   navesti:    ; a tady se pokračuje dál.
7
8   ; IF (NZ) {INC A, DEC D} else {DEC A, DEC E}
9
10  JZ else     ; Pokud podmínka neplatí, přeskoč
11  INC A       ; blok příkazů
12  DEC D
13  JMP endif   ; bez toho by se provedl i blok ELSE
14  else:
15  DEC A
16  DEC E
17  endif:      ; a tady se pokračuje dál.
```

Takto tedy funguje konstrukce IF – THEN a IF – THEN – ELSE.
Podobně jako jsou podmíněné instrukce skoku JMP, tak můžeme vytvořit podmíněné skoky do podprogramu (Ccond – CZ, CNZ, CC, CNC, CM, CP, CPE, CPO) a podmíněné návraty (RZ, RNZ, RC, RNC, RM, RP, RPE, RPO). Fungují stejně jako CALL a RET, ale provedou se pouze pokud je splněná podmínka.

WHILE!

Dobře, ukážeme si jednu konstrukci z vyššího jazyka, přepsanou do assembleru. Představte si takový počítaný cyklus WHILE:

```
1 while (--B) {
2 A = ctiKlav();
3 pisObr(A+1);
4 }
```

Tedy dokud je hodnota B vyšší než 0, tak od něj odečti 1, zavolej funkci ctiKlav, výsledek ulož do A, a zavolej funkci pisObr s parametrem v A.

Řekněme si na rovinu – nic takového jako „výsledek ulož do registru A" nemáme. Je na autorovi podprogramu ctiKlav, aby vrátil požadované v registru A. Stejně tak nemůžeme podprogramu pisObr předávat nějaké parametry – pokud podprogram očekává nějakou hodnotu v registru A, je na nás, abychom ji do toho registru připravili. Nemluvě o tom, že budeme používat registr B jako počítadlo průchodů, takže pokud nám ho některý z podprogramů přepíše, tak algoritmus nebude fungovat.

Proto je, a to vám kladu na srdce, dobrým zvykem u každého podprogramu napsat do dokumentace, ve kterém registru očekává parametr, ve kterém vrací hodnotu, a zmínit, pokud nějaký registr přepíše, ať s tím může ten, kdo ten podprogram použije, správně naložit. (Kdysi jsem četl historku o tom, jak se kdosi v 80. letech bál použít podprogramy monitoru, protože v dokumentaci u nich psali poznámky jako: *Ničí registry D, E*. Čímž autor samosebou myslel, že přepíše hodnoty v nich uložené, ovšem nebohý juniorní programátor se bál toho výrazu *ničí* – počítač je tak drahý stroj, tak si přeci nevezme na svědomí, aby se vevnitř něco zničilo.)

Tak jak tedy to výše zmíněné přepsat do assembleru? Řešení:

```
1 while:
2 DCR B         ; --B
3 JZ endwhile   ; je už 0? Skok na konec...
4 CALL ctiKlav  ; volání
5 INR A         ; A = A + 1
6 CALL pisObr   ; volání
7 JMP while     ; další průchod smyčkou
8 endwhile:
```

Budete používat nejrůznější varianty této konstrukce – s podmínkou na konci (do {…} while()) nebo s podmínkou uprostřed, ale vždy to bude plus mínus něco podobného.

Ony totiž všechny ty konstrukce, které znáte a máte rádi (for, while apod.), jsou nakonec často přeloženy do strojového jazyka, a to nějakým způsobem, který je velmi podobný tomuto. Tedy samé IF…THEN GOTO a GOTO.

GOTO!

Instrukce 8080 – rotace

AUTOR: MARTIN MALY PUBLIKOVÁNO 23. 12, 2013

„Cože? Rotace? Proč ne něco Opravdu Důležitého?" – ale ony jsou docela důležité, věřte mi, a když je správně použijete, tak ušetří spoustu času.

Procesor 8080 oplývá sadou instrukcí, která dokáže provádět takzvané bitové rotace registru A. Co to znamená? Představme si registr A jako osm bitů, očíslovaných (od nejnižšího) 0-7. Představme si je napsané vedle sebe.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

To je on. S ním budeme teď pracovat a jako první si ukážeme operaci „rotace vlevo":

RLC

Instrukce RLC posune bity o 1 doleva. To znamená, že bit 0 se přesune na pozici 1, bit 1 na pozici 2, bit 2 na pozici 3 a tak dál, a bit 7, který nám vypadne zleva ven, se zase vrátí zprava na pozici 0. (A právě proto, že takhle „krouží", se té operaci říká „rotace". Kdyby vypadl a byl zahozen, byl by to „posuv" – ale takové instrukce 8080 nemá). Bit 7 je zároveň zkopírován do příznaku CY.

OPERACE	POZICE V REGISTRU A								PŘÍZNAK
	7	6	5	4	3	2	1	0	CY
VÝCHOZÍ STAV	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	CY
RLC	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Bit 7	Bit 7

RRC

Instrukce RRC posune bity o 1 doprava. To znamená, že bit 7 se přesune na pozici 6, bit 6 na pozici 5, bit 5 na pozici 4 a tak dál, a bit 0, který tentokrát vypadne vpravo, se vrátí zleva na pozici 7, a návdavkem je zkopírován do příznaku CY.

OPERACE	POZICE V REGISTRU A								PŘÍZNAK
	7	6	5	4	3	2	1	0	CY
VÝCHOZÍ STAV	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	CY
RRC	Bit 0	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

RAL

Další dvě instrukce berou v úvahu i obsah příznaku CY. RAL funguje stejně jako RLC, ale s jedním rozdílem – bit 7, který zleva vypadne, je zapsán do příznaku CY, a hodnota z CY je přesunuta do pozice 0 v registru A. Graficky to vypadá nějak takto:

OPERACE	POZICE V REGISTRU A								PŘÍZNAK
	7	6	5	4	3	2	1	0	CY
VÝCHOZÍ STAV	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	CY
RAL	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	CY	Bit 7

RAR

RAR je obdoba předchozí funkce RAL, jen směr je opačný – bity se posouvají doprava, bit 0 jde do CY a „staré CY" jde na pozici 7.

OPERACE	POZICE V REGISTRU A								PŘÍZNAK
	7	6	5	4	3	2	1	0	CY
VÝCHOZÍ STAV	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	CY
RAR	CY	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Víc rotací procesor 8080 nemá.

Jaké mají tyto operace smysl? Když se nad tím zamyslíte, je to analogické jako u desítkové soustavy: když máte číslo (123) a posunete ho o jednu pozici doleva (a zprava doplníte nulou), dostanete jeho desetinásobek (1230). Když ho posunete doprava, dostanete celočíselný výsledek dělení deseti. U dvojkové soustavy platí totéž – posun doleva je totéž jako vynásobit dvěma, posun doprava je dělení dvěma.

Trochu tam hapruje to rotování kolem dokola, ale to lze obejít pomocí vhodného domaskování nebo tím, že si předem nastavíte příznak CY na požadovanou hodnotu.

Dají se použít ještě na spoustě dalších míst – ale to si ještě ukážeme.

STC, CMC A JAK NASTAVIT PŘÍZNAK CY?

Přiznám se, že jsem váhal, kam zařadit instrukce STC a CMC, a nakonec je zařadím sem. Instrukce STC nastaví CY na 1, instrukce CMC neguje jeho hodnotu. Pokud chcete CY nastavit na nulu, musíte buď udělat STC a CMC, nebo (pokud vám nevadí, že přijdete o hodnotu v příznacích S, Z, P) zkusit třeba CMP A – tato instrukce porovná registr A se sebou samotným. Výsledek je, nepřekvapivě, „hodnota se sobě rovná", takže instrukce nastaví Z na 1 a S a CY nuluje.

ROTACE 8080 – SOUHRN

OPERACE	POZICE V REGISTRU A								PŘÍZNAK
	7	6	5	4	3	2	1	0	CY
VÝCHOZÍ STAV	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	CY
RLC	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Bit 7	Bit 7
RRC	Bit 0	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
RAL	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	CY	Bit 7
RAR	CY	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

No related posts.

8080 – násobení

Když se sejdou tři sobi, tak se radost násobí…

Už jsme si řekli, že násobení nepatří mezi základní operace, kterými procesor 8080 oplývá. A rovnou můžu prozradit, že to ani u Z80, ani u 6502 není v tomto směru o moc lepší. Po pravdě si nevzpomínám na žádný osmibitový procesor, který by měl instrukci pro násobení. Neříkám že neexistoval, jen si na něj nevzpomínám. (Paměť mi osvěžil až Roman Bórik: osmibitové jednočipy – mikrokontroléry řady 8051/8052 mají instrukce násobení i dělení.)

Pokud chceme násobit, a taková potřeba na každého někdy přijde, musíme si vystačit s tím co máme a známe. A taky trošku zavzpomínat na základní školu.

NÁSOBENÍ POPRVÉ – NAIVNÍ

Víme, že 3 krát 5 je vlastně 5 + 5 + 5. Třikrát se sečte pětka. Mohlo by to vypadat nějak takhle – násobíme obsah registru B obsahem registru C a výsledek ukládáme v A:

Rovnou upozorňuju, že to je jeden z nejzoufalejších kódů, co tu budou zveřejněné. Je špatný z mnoha důvodů. Zaprvé – když násobím dvě osmibitová čísla, může být výsledek až šestnáctibitový (prostý test: 255 * 255 = 65025 (FE01h)). Já tu sčítám hodnoty v registru A, který je osmibitový a přeteče už u čísla 255. Což, to by se dalo napravit snadno – místo sčítání v registru A můžu použít registry HL, přičítat k nim číslo z registrů DE a bude to… Nějak takto (opět násobíme B * C, výsledek jde do HL):

Teď už můžeme násobit v plném rozsahu, tenhle problém jsme vyřešili, ale je tu jiný problém. Totiž ten, že násobení trvá dlouho. Představte si, že B bude třeba 250 a C dvě. Budeme 250krát přičítat dvojku, stále dokola – ostatně zkuste si to v emulátoru nasimulovat, uvidíte sami. Co s tím?

Můžeme čísla na začátku prohodit (místo „250x přičti dvojku“ udělat „2x přičti 250“), ale to je řešení na půli cesty. Vlastně ne na půli cesty, ve skutečnosti na pytel! Správný postup je zahodit tenhle algoritmus a jít na to jinak.

NÁSOBENÍ PODRUHÉ – RAFINOVANÉ

Spousta lidí tváří v tvář dvojkové soustavě strne a očekává, že přestanou fungovat postupy, které znají ze soustavy desítkové. Ale ony fungují docela dobře. Třeba to násobení se dá obstojně zařídit stejně, jako jsme se to učili ve třetí třídě ZŠ, když jsme násobili na papíře dvojciferná čísla pod sebou. Pamatujete?

```
1       26
2     *  34
3     ------
4    104   (4 * 26)
5  +  78    (3 * 26)
6     ======
7     884
```

Postup popíšu slovy. Začneme od nejnižšího řádu druhého činitele (4). Tím vynásobíme prvního činitele (4*26) a dostaneme první mezivýsledek (104). Ten platí pro nejnižší řád (je zapsaný úplně vpravo, zarovnaný s tím řádem, který právě zpracovávám). Pokračuju k vyššímu řádu druhého činitele. Vida, je to 3. Postup opakuju: vynásobím tím číslem první činitel (3*26) a druhý mezivýsledek (78) si zapíšu na pozici desítek, tedy o řád výš než předchozí. A takhle budu pokračovat, dokud nepronásobím všechny řády druhého činitele. Nakonec mezivýsledky sečtu.

Tak. A jde to tak i ve dvojkové soustavě? Zkusíme to. Kolik je dvojkově 13 * 9?

```
1       1101  13
2     *   1001   9
3       ---------
4      1101  1 * 1101
5   +   0000   0 * 1101
6   +  0000    0 * 1101
7   + 1101     1 * 1101
8      =========
9      1110101  117
```

Postup je naprosto stejný – taky násobíme a posouváme o řád, ovšem situaci máme o mnoho jednodušší: Protože pracujeme ve dvojkové soustavě, tak násobíme vždy jen buď jedničkou (tedy mezisoučet je roven prvnímu činiteli), nebo nulou (a mezisoučet je 0). Mezisoučty postupně posouváme doleva a přičítáme.

Tak, s touhle vědomostí by určitě šlo napsat násobící algoritmus, který nemá tu nectnost, že by závisel na hodnotě činitele. Jeho hlavní smyčka proběhne tolikrát, kolik má druhý činitel bitů. Ale předtím, než se do něj pustíme, si řekneme ještě pár tipů.

K posunu registru o jeden bit doleva můžeme samozřejmě použít instrukci rotace. Ovšem tady budeme rotovat rovnou dva registry najednou. Nejjednodušší způsob bude použít instrukci DAD H. Ta, jestli si vzpomínáte, přičte k registrovému páru HL hodnotu svého parametru – tedy HL. Sečíst HL a HL znamená vlastně udělat 2*HL, a z hlediska bitového je to totéž jako posunout obsah registru doleva o 1 bit a zprava doplnit nulou. Tedy přesně to, co se děje v tom algoritmu výše.

Troufnete si napsat algoritmus, který vynásobí obsah registrů D a E a výsledek uloží do HL? Zkuste to…

Řešení je následující:

Může vám připadat trošku zmatené, ale nebojte, hned si to projdeme.

Na začátku nastavím zásobník na rozumnou pozici. Připravím si do registrů D a E hodnoty, které budeme násobit (13 a 9). Samosebou bych mohl použít LXI D, 0x0D09, ale pro názornost je to takhle rozepsané. Pak volám podprogram Mul8 – jako že „Multiplication, 8 bits“.

V podprogramu si nejdřív vynuluju registry HL. Tam se budou průběžně sčítat mezivýsledky. Do B si připravím hodnotu 8 – to je počítadlo, kolikrát provedu hlavní smyčku. Budu ji provádět osmkrát, pro každý řád jednou, tak proto ta osmička. Teď bych potřeboval, aby v DE byl jen druhý činitel (tedy v D 0 a v E to, co tam je). Přesunu si tedy obsah D do registru C (mám teď činitele v registrech C a E) a registr D si vynuluju. A teď může začít vlastní kouzlo…

Podívám se, jakou hodnotu má nejnižší bit registru C, a posunu si ho o jednu pozici doprava (v příštím průchodu tak budu kontrolovat pozici 1, pak pozici 2 atd.) To ale nemůžu udělat přímo, takže si jej musím nejprve přesunout do registru A, tam provést RRC a pak uložit zpátky do C. Instrukce RRC dělá obě věci, co potřebuju, najednou – uloží nejnižší bit (do příznaku CY) a rotuje o jednu pozici doprava.

Podle hodnoty nejnižšího bitu (mám ho teď v CY) se rozhodnu, jestli mezisoučet (DE) přičtu k celkovému výsledku (pokud je 1), nebo nepřičtu (CY=0). Použiju podmíněný skok: Pokud není CY (JNC), tak skoč dál. Pokud je CY, pokračuj a přičti DE k HL (HL=HL*DE)

Tím mám jeden krok skoro za sebou. Teď už jen musím posunout mezisoučet v DE o jednu pozici doleva. Vyřeším to tak, že ho přičtu k sobě samotnému (tedy DE = DE + DE). A protože na to instrukci nemáme, tak si pomocí XCHG na chvíli prohodíme DE a HL a provedeme HL = HL + HL.

Pak už jen snížím počet průchodů smyčkou (DCR B), a pokud ještě nejsme na nule, tak to celé provedeme znovu.

A věřte nebo ne, na tomhle principu je založena naprostá většina softwarových násobiček.

NÁSOBENÍ DVAAPŮLTÉ – OPTIMALIZOVANÉ

Můžeme u předchozího algoritmu ušetřit pár bajtů, dva registry a nějaký ten čas, a to tím, že druhý činitel umístím do registru H, zatímco mezisoučet bude neustále v HL. Zní to divně? Divné to teprve začne být!

Ve skutečnosti při tomto postupu nepůjdeme od nejmenšího bitu, ale od nejvyššího. Náš postup nebude ten, že při každém průchodu přičteme první činitel, patřičně posunutý doleva (*2), ale při každém průchodu naopak vynásobíme mezisoučet dvěmi a případně přičteme první činitel. Matematicky místo $(A * 2^0 * b0) + (A * 2^1 * b1) + (A * 2^2 * b2) + \ldots$ (A je první činitel, b0-b7 bity druhého činitele) budeme provádět (((A*b7)*2+(A*b6))*2+(A*b5))*2… Pro zájemce: Hornerovo schéma. *(Náš učitel programování tomu říkal tvrdošíjně „Hornerovo šéma" – ale on říkal taky místo while cosi, co znělo jako [hwajl]…)*

Využijeme toho, jak se chová instrukce DAD H. Už jsme si řekli, že udělá HL = HL + HL, tedy HL * 2. Tedy vlastně posune obsah HL o jednu pozici doleva. Zároveň nastaví příznak CY, pokud hodnota přeteče – v tomto případě přeteče, pokud je nejvyšší bit registru H = 1… takže jako by zkopíroval ten nejvyšší bit do příznaku CY.

Takže jednou instrukcí tu máme:

- posun hodnoty v registru H o jeden bit doleva (tedy můžeme jít bit po bitu od b7 po b0)
- při každém posunu udělá vlastně jeden bit prostoru pro mezisoučet v HL (to, že H v průběhu výpočtu ve vyšších bitech obsahuje nějaký zmatek, to nevadí, protože „zmatek" se postupem času odsune pryč)
- vynásobí mezisoučet dvěma.

Tolik dobra za jednu cenu, že?

Vyšší dívčí? Nebojte se, vpravíte se do toho. V očích vám teď vidím otázku: Zešílels? Tohle přeci není normální!

Takže: *Ano, zešílel, ale to s tím nijak nesouvisí, a ne, v assembleru je tohle naprosto normální.* Když chcete, hrajete o každý bit v registru, takt procesoru, bajt v paměti… A někdy ani nechcete, ale musíte!

NÁSOBENÍ POTŘETÍ

V assembleru má optimalizace vždycky dvě fáze. V té první vyhazujete instrukce, které jste napsali zbytečně – program bude kratší i rychlejší. V té druhé si musíte vybrat: Chcete kratší kód? Musíte dělat triky, co vás budou stát nějaký ten čas. Chcete kód rychlý? Musíte obětovat prostor. I u algoritmu násobení stojíte před stejným rozhodováním: rychleji? Bude to něco stát…

Úplně nejrychlejší algoritmus by byl takový, který by měl v paměti výsledky všech možných kombinací pro násobení (v případě osmibitových činitelů by jich bylo 65536) a prostě by si jen sáhnul tam, kam potřebuje. Taková tabulka by ale zabrala spoustu paměti a pro vlastní program by v adresním prostoru běžného osmibitového procesoru zůstalo… ehm… přesně 0 bajtů. Naštěstí existují kompromisní řešení, kdy si do tabulky uložíme jen pár konstant (třeba 1024) a zbytek dopočítáme. Je to opět kompromis mezi rychlostí a velikostí. A jak tedy bude násobení vzor 3 vypadat? To si nechme na jindy…

Instrukce 8080 – logické operace

„Milý autore, tvůj seriál je zmatený a nemá žádnou logiku! Logika v něm chybí! Váš čtenář." – „Milý čtenáři, logika tu nechybí. Logika je tu právě teď!"

Protože předpokládám, že všichni, co to čtete, jste už nějaký programovací jazyk zvládli, tak si nemusíme, doufám, vysvětlovat, co je to AND, OR a XOR. Některým tužším borcům naznačím: &, | a ^. Už jo, už to berou, tak pojďme na to.

ANA, ANI

Provede operaci AND mezi registry A a vybraným registrem (ANA), nebo mezi registrem A a zadanou osmibitovou konstantou (ANI). Hezky bit po bitu, a výsledek jde zpátky do A. Pokud se na parametr budeme dívat jako na masku, kterou je třeba aplikovat na registr A, tak vězte, že kde má maska 1, tam zůstane A nedotčeno, kde má 0, tam bude vynulováno. Instrukce ovlivňují všechny příznaky.

ORA, ORI

Provede operaci OR mezi registry A a vybraným registrem (ORA), nebo mezi registrem A a zadanou osmibitovou konstantou (ORI). Hezky bit po bitu, a výsledek jde zpátky do A. Pokud se na parametr budeme dívat jako na masku, kterou je třeba aplikovat na registr A, tak vězte, že kde má maska 0, tam zůstane A nedotčeno, kde má 1, tam bude nastaveno na 1. Instrukce ovlivňují všechny příznaky.

XRA, XRI

Provede operaci XOR mezi registry A a vybraným registrem (XRA), nebo mezi registrem A a zadanou osmibitovou konstantou (XRI). Hezky bit po bitu, a výsledek jde zpátky do A. Pokud se na parametr budeme dívat jako na masku, která se bude aplikovat na registr A, tak vězte, že kde má maska 0, tam zůstane A nedotčeno, kde má 1, tam bude změněna hodnota příslušného bitu v A. Instrukce ovlivňují všechny příznaky.

CMA

Vezme obsah registru A a zneguje ho, tj. všechny 1 prohodí za 0 (a obráceně). Instrukce nijak neovlivní příznaky.

PRAKTICKÉ POUŽITÍ

Takhle odsud to možná vypadá jako nějaká zajímavost pro nerdy, ale věřte, že v assembleru budete tyhle instrukce používat často. Například pro nastavení konkrétního bitu použijete ORI 01h (02h, 04h, 08h, 10h, 20h, 40h, 80h). Pro vynulování bitu použijete AND, ale v negované podobě – tam, kde chcete bit nastavit na 0, nechte 0, všude jinde 1. Chcete-li vynulovat bit 3 (To je on: „….3…"), použijete konstantu, která má všude 1, s výjimkou pozice 3: „11110111". Tedy F7h.

Operace XOR slouží krom změny hodnot bitů i k jednomu hezkému fíglu: pokud potřebujete vynulovat registr A a nezáleží vám na tom, že se změní hodnota příznaků, použijte XRA A. Taková instrukce zabere jen jeden bajt a je i proti MVI A,0 rychlejší.

Instrukce CMA má efekt stejný jako XRI FFh – ale nemění příznaky a zabere jen jeden bajt.

Teď jsme se dostali do bodu, kdy zbývá probrat už jen šest instrukcí procesoru 8080, takže je opět vhodná chvíle na drobné odbočení, ať to máme pestřejší…

No related post

Instrukce a operační kód

… a když na vás o půlnoci zakřičím: „C3h!", tak vy vyskočíte!

Jistě jste se v předchozích kapitolách dívali na to, jaké kódy patří k jakým instrukcím a zaujalo vás, že „MVI A, něco" má vždycky první bajt 3Eh. (*Cože? Nevšimli? Nezaujalo? A zvážili jste, že byste se místo assembleru učili třeba Javu?*)

Nebo taková instrukce MOV – ať už je jakákoli, je její kód vždycky mezi 40h a 7Fh. Má to svoji logiku. Schválně, podívejte se sami (ten program je naprosto nesmyslný, proto se nedívejte, co dělá, ale jak je přeložený!)

Dá se v tom vypozorovat určitá pravidelnost, že? Instrukce MOV obecně vypadá totiž po jednotlivých bitech takto:

0	1	d	d	d	s	s	s

Trojice bitů D a S kódují jednotlivé registry (D = cílový, S = zdrojový). Registry jsou kódované následujícím způsobem:

R2	R1	R0	REGISTR
0	0	0	B
0	0	1	C

R2	R1	R0	REGISTR
0	1	0	D
0	1	1	E
1	0	0	H
1	0	1	L
1	1	0	M
1	1	1	A

Dva nejvyšší bity jsou 0 a 1. Pokud má instrukce nejvyšší dva bity takto nastavené, ví dekodér instrukcí, že jde o MOV, že se bude přesouvat z registru do registru, a informaci o tom, ze kterého do kterého najde v bitech 5-3, resp. 2-0. (Ano, je tu výjimka – přesun z registru M do registru M není možný a operační kód, který by taková instrukce měla (76h) je vyhražen pro jinou instrukci.)

Co třeba instrukce s takovýmhle bitovým obrazem?

0	0	d	d	d	1	1	0

Možné kódy jsou 06h, 0Eh, 16h, …, 3Eh. Tato bitová kombinace říká dekodéru, že se jedná o instrukci MVI, tedy přesun následujícího bajtu do některého z registrů. Který registr to bude, to je opět zakódované v operačním kódu – jsou to ty tři bity „d".

A co takováhle instrukce?

0	0	d	d	d	0	1	0

Možné kódy jsou 04h, 0Ch, 14h, …, 3Ch. Tentokrát se jedná o instrukci INR a v operačním kódu je opět určeno, o jaký registr půjde.

Když se podíváte do tabulky instrukcí, zjistíte, že instrukce nejsou rozmístěny nahodile. V první čtvrtině (00-3Fh) jsou nejrůznější přesuny, rotace, inkrementace a dekrementace. Druhá čtvrtina (40h-7Fh) patří přesunům (MOV), třetí třetina (80h-BFh) obsahuje aritmetické instrukce, které pracují s registry (ADD, ADC, …) a poslední čtvrtina (C0h-FFh) obsahuje převážně skoky, operace s přímým operandem a operace se zásobníkem.

Takováto výstavba instrukcí je dána potřebou mít co nejjednodušší dekodér. Později, až se podíváme na operační kódy procesorů 6502 nebo 680x, zjistíme, že jsou v dodržování tohoto principu mnohem důslednější a že i přes své zdánlivé limity nabízejí programátorovi mnohem větší svobodu.

s.

8080 – Ahoj světe!

Konečně! Konečně jsme se dostali do bodu, kdy si vypíšeme ono známé a populární AHOJ SVĚTE, a pak už nás nic nezastaví! Samozřejmě, s čistým assemblerem toho moc nedokážeme. Potřebujeme taky stroj, který náš program vykoná. Teď nastala ta chvíle, kdy si můžete vytáhnout své pečlivě schraňované PMI-80 a… Cože, vy nemáte PMI-80? No dobrá tedy. Použijeme PMD-85. Ten snad všichni máte. Támhle vzadu někdo přikyvuje… jo aha, to je RM-Team, to je výjimka, vy ostatní, hm… nemáte… To nevadí. Půjčte si alespoň emulátor PMD-85.

Tak. Náš první opravdový program vypíše na obrazovku „AHOJ SVETE" a skončí. Jak to udělá?

No, zaprvé bude potřeba někde v paměti mít uložená písmenka A, H, O, J… atd. jak se mají vypsat. Jedno po druhém budeme brát a zobrazovat je na displeji. Bude tedy rozumné, když si adresu toho řetězce uložíme do nějakých registrů. Z toho, co jsme si řekli, víme, že pro podobné účely je nejvhodnější dvojice registrů HL. Znak načteme do registru A pomocí MOV A,M, na další pozici se přesuneme pomocí INX H… Základní kostra tedy bude vypadat nějak takhle:

```
1    .ORG    0
2    LXI     h,text
3    SMYCKA:
4    MOV     A,M
5    tady nějak vypsat znak z registru A
6    INX     H
7    JMP     smycka
8
9    TEXT:   Tady nějak bude ten text
```

Zbývá vyřešit pár detailů. Třeba: Jak vypíšeme ten znak? Tady nám pomůže monitor PMD-85. Monitor není, prosím pěkně, ta bedna, na kterou civíte – ve světě osmibitových počítačů se „monitor" říkalo základnímu obslužnému programu, který umožňoval nějaké elementární operace se systémem – prohlédnout si paměť, změnit její obsah, spustit program, uložit ho na záznamové zařízení atd. Něco jako pozdější „operační systém". Takový monitor býval uložený v pamětech typu ROM (spíš EPROM, ale to je nepodstatný detail) a krom toho, že komunikoval s obsluhou, tak většinou nabízel i nějaké podprogramy, které může využít programátor ke své práci.

Když se podíváme do výpisu monitoru PMD-85 (k dispozici je třeba na stránkách RM-teamu), co nám nabízí, najdeme následující kousek:

```
1    PRTOUT 8500H - výstup znaku na obrazovku
2    Program se snaží zobrazit všechny znaky kromě 0AH, který ignoruje, na 1CH
3    smaže kompletně celou obrazovku (ale zápisník nechá) a znak 0DH odřádkuje a
4    nastaví kurzor na začátek řádku (FB00H). Ignoruje mód kreslení bodu!!!
5    vstup:   A - kód znaku
6    (C03EH) - kurzor - adresa ve V-RAM
7    (C03CH) - ukazatel tabulky znaků
8    (C03AH) - barevné atributy
```

```
9    výstup:  obrazovka
10   používá: PSW
```

To je asi to, co hledáme. Podprogram vypíše znak z registru A na obrazovku. Pokud to je 1Ch, tak smaže obrazovku, 0Dh odřádkuje, 0Ah ignoruje, zbytek vypíše. Vstup očekává v registru A. Ty tři řádky pod tím – *(C03EH) – kurzor – adresa ve V-RAM* – teď budeme ignorovat, budeme doufat, že je to všechno nastavené, jak má být. (Pro zvídavé – C03EH v závorkách naznačuje, že se jedná o obsah paměti na adrese C03Eh… Tady půjde pravděpodobně o dva bajty na adresách C03Eh a C03Fh – ale to bych sem teď opravdu netahal.)

V popise je dále napsáno, že výstupem je obrazovka – jo, to tak nějak čekáme, a že program používá PSW. Vzpomeňte si, jak jsme si říkali, že je dobré do dokumentace napsat, které registry náš podprogram přepíše. Tento podprogram je slušný, změní jen A a F (neboli PSW), zbytek zůstane zachovaný, tudíž nemusíme hodnotu registrů HL (kde máme adresu toho řetězce) uschovávat.

Takže je jasno: Výpis si vyřídíme pomocí CALL 8500h, neboli CALL PRTOUT.

PRTOUT je symbolické jméno, abychom si nemuseli pamatovat adresu 8500h – napovídá, že půjde o nějaký výpis (PRINT OUT). Podobných programů je v monitoru velké množství – třeba KEYBD, jak už název napovídá, bude mít něco společného s klávesnicí (KEYBoarD), pravděpodobně čtení znaku z klávesnice. Každopádně se KEYBD pamatuje snáze než 84A1h, navíc když uděláte překlep a napíšete KEYDB, překladač vás upozorní, že takovou adresu nezná, zato když se uklepnete na 841Ah, nikdo vás neupozorní a program nebude fungovat. Jak používat tyto symbolické názvy se dozvíte v další lekci.

Další problém, který je třeba vyřešit: Program musí nějak poznat, že už vypsal všechny znaky a že má přestat. Kdyby to nepoznal, tak vypíše všechno až doalelujá a nikdy neskončí. Můžeme si zase zvolit nějaký registr jako počítadlo znaků, dát do něj předem délku, při každém průchodu zmenšit o 1, a když dojdeme k nule, tak skončit. To je řešení logické, ale má svoje nevýhody: Zaprvé, musíme znát délku řetězce. To by za nás dokázal pohlídat překladač. Zadruhé: nejdelší řetězec bude mít 256 znaků. I to by se dalo vyřešit, kdybychom pro počítadlo vyhradili dvojici registrů. Jenže takové řešení je delší než to, které nakonec použijeme my.

My totiž uložíme do paměti řetězec znaků AHOJ SVETE, a hned za něj uložíme nulu. Ti z vás, kteří znají céčko, teď pravděpodobně pokyvují hlavou, že jim je tohle řešení povědomé. Ano, je to přesně to. Ve smyčce pak čteme jednotlivé znaky a kontrolujeme, jestli to není náhodou 0. Pokud to je 0, tak jsme skončili.

Jak zkontrolovat, jestli je bajt 0? MOV A,M nijak nepomůže, tato instrukce neovlivňuje příznaky. Můžeme použít CPI 0, tedy instrukci pro porovnání. Nebo můžeme opět použít jednoduchý trik: Použijeme instrukci, která nezmění hodnotu, ale nastaví příznaky… Nabízí se operace OR nebo AND, tedy instrukce ORA A nebo ANA A. OR hodnoty s ní samotnou je stále ta hodnota – ale nastaví příznaky S, Z, AC, P a CY. Nás bude zajímat příznak Z, který bude 1, je-li hodnota nulová.

Nová verze vypadá tedy takto:

```
1        .ORG    0
2        LXI     H,text
3        SMYCKA:
4        MOV     a,m
5        ORA     a
6        JZ      stop
7        CALL    8500h
8        INX     h
9        JMP     smycka
10
11       STOP:   RET
12
13       TEXT:   Tady nějak bude ten text
```

Krok po kroku: Připravíme si adresu textu do registrů HL (LXI H, text). Smyčka začíná tím, že přečteme znak z adresy HL do registru A (MOV A,M), porovnáme hodnotu samu se sebou (ORA A) a pokud je to nula, skáčeme na adresu STOP (JZ stop). Pokud ne, tak voláme PRTOUT, zvýšíme adresu v HL o 1 a pokračujeme v provádění smyčky.

Stop se v našem programu implementuje jako RET – tedy návrat zpátky do toho místa, odkud byl program spuštěn. V našem případě tedy zpátky do monitoru PMD-85. Je vyřešen trochu nešikovně – podívejte se sami: JZ na adresu STOP, a na té adrese je jen RET. Pokud bychom po vypsání chtěli dělat ještě něco, můžeme to napsat od návěští STOP dál. Ale jestliže nic dělat nechceme, můžeme se přímo ze smyčky vrátit – místo instrukce JZ na adresu, kde je instrukce RET, můžeme přímo použít podmíněnou instrukci RZ.

Zbývá vyřešit poslední problém, totiž jak napsat to „AHOJ SVETE" do paměti od adresy TEXT.

K tomu slouží pseudoinstrukce DB. Podobně jako ORG říká překladači „program bude uložen od této adresy", tak DB říká: „To, co je parametrem téhle pseudoinstrukce ulož do kódu jako jeden bajt." Takže kdekoli se ve zdrojovém kódu objeví třeba DB 41h, tam překladač vloží bajt 41h. Což je zároveň ASCII kód znaku A. Pokud chceme vložit ASCII kód znaku A, nemusíme hledat v tabulkách, že to je 41h – stačí zapsat DB „A" a překladač to vyřeší za nás.

Takže můžeme napsat DB „A", „H", „O", „J" … anebo jednodušeji: DB „AHOJ SVETE" Když překladač za DB narazí na řetězec, uloží kódy jednotlivých znaků do paměti po sobě tak jak jdou v řetězci.

Samozřejmě je dobré dát si pozor na to, aby se program nikdy na tuhle adresu nedostal, protože kdyby začal provádět „instrukce" (ve skutečnosti nějaká data), tak by se mohlo stát cokoli.

Náš program tedy vypadá nakonec takto:

Tentokrát nám krokování v debuggeru nepomůže – u instrukce CALL 8500h debugger vykolejí, protože na takové adrese nic nemá. Musíme náš program přenést do počítače. Přeložte si ho a soustřeďte se na listing:

```
1    0000              .ORG 0
2    0000 21 0D 00     LXI h,text
3    0003              SMYCKA:
4    0003 7E           MOV a,m
5    0004 B7           ORA a
6    0005 C8           RZ
7    0006 CD 00 85     CALL 8500h
8    0009 23           INX h
9    000A C3 03 00     JMP smycka
```

```
10   000D           TEXT:
11   000D 41 48 4F 4A 20   DB "AHOJ "
12   0012 53 56 45 54 45 21 DB "SVETE!"
13   0018 00              DB 0
```

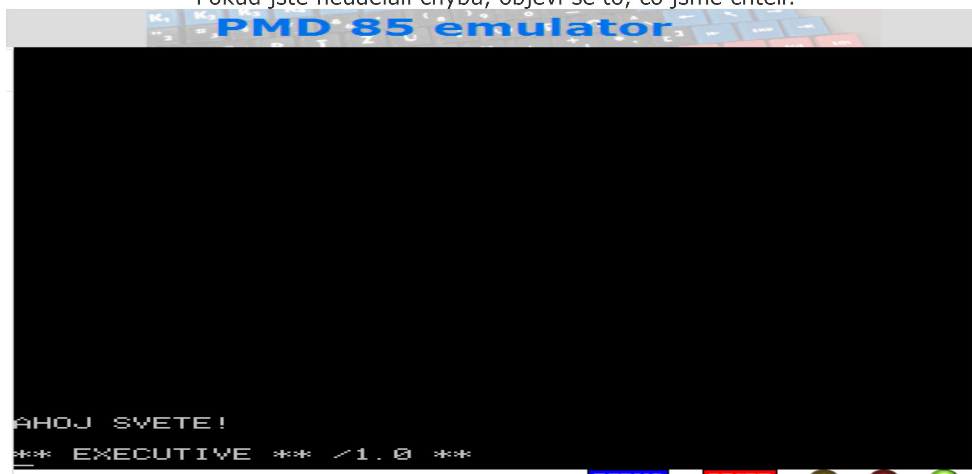
Na adresu 0000 je třeba zapsat hodnotu 21 (vše hexadecimálně), na 0001 hodnotu 0D, na 0002 hodnotu 00, na 0003 hodnotu 7E… na další adresy pak hodnoty B7, C8, CD, 00, 85, 23, … a konečně na adresu 0018 zapíšeme nulu.
Zapněte si svá Péemdéčka (nebo si spusťte v druhém okně emulátor PMD-85) a napište: SUB 000021 (a klávesu EOL / ENTER). Tím se do paměti na adresu 0000 uloží hodnota 21. V příkazovém řádku se teď objevilo SUB 0001 – stačí tedy dopsat 0D a EOL (ENTER)… Nebo můžete využít toho, že příkaz SUB dokáže uložit víc hodnot najednou, a napsat po sobě toto:

```
1   SUB 0000210D007EB7C8CD00
2   SUB 00088523C3030041484F
3   SUB 00104A2053564554452100
```

Monitor vám po odeslání každého řádku připraví nový příkaz SUB… Když odešlete poslední řádek, stiskněte EOL (na PC emulátoru ENTER) bez zadání dat. PMD vypíše chybu a očekává nový příkaz. Ke spuštění programu slouží v monitoru PMD-85 instrukce JUMP xxxx, kde xxxx je hexadecimálně zadaná adresa. Zkuste to:

```
1   JUMP 0000
```

(a EOL / Enter)
Pokud jste neudělali chybu, objeví se to, co jsme chtěli:



Zvládli jsme to! Pokořili jsme kus křemíku a donutili ho udělat to, co jsme chtěli. Malý krok pro člověka… atakdál!
Tip: Můžete si ten program vyzkoušet samozřejmě i v online vývojovém prostředí ASM80. Uložte si tento soubor pod nějakým názvem s příponou .a80. Do kódu napište na prázdný řádek, třeba na samotný začátek, pseudoinstrukci „.ENGINE pmd" (bez uvozovek, samosebou). Když pak kliknete na EMULATE, program se přeloží, spustí se emulátor PMD-85 a v něm už bude váš program nahraný od té adresy, kterou jste zadali v ORG. Ušetří vám to práci s přepisováním všech těch kódů. A od toho přeci počítače máme – aby nám ušetřili práci. Ne?

Pseudoinstrukce

A jestli vám vadí to „pseudo-„, nemáte rádi věci, co jsou jen „jako" a raději chcete mít vše jasně nařízené, tak to nazývejte třeba „direktivy".
Překladač (správně nazývaný assembler, ale už jsme si vysvětlili, že tohle slovo se taknějak přemigrovalo i na Jazyk Symbolických Adres, a já tentokrát potřebuju zdůraznit, že mám na mysli ne ten jazyk, ale opravdu překladač, *přičemž jaksi mimochodem píšu asi nejdelší vsuvku na celém tomhle webu a doufám, že si vzpomenu, jak jsem chtěl tu větu dokončit…*) plní hlavně dva úkoly:

1. Překládá instrukce, zapsané v mnemotechnickém tvaru, tj. v podobě nějakých snadno zapamatovatelných názvů, na jejich instrukční kódy (MOV B, C => 41h).
2. Počítá pozici v paměti a dovoluje je nazvat nějakým lidským jménem, abychom mohli napsat „CALL podprogram" a nemuseli přemýšlet, na jaké že adrese ten podprogram je.

Kromě těchto instrukcí („pravých" instrukcí) pracuje překladač ještě s takzvanými pseudoinstrukcemi. „Pseudo-" proto, že se zapisují jako instrukce, ale ve skutečnosti je procesor nezná. Zná je překladač a nějak se podle nich zachová. Některé už jsme si ukázali.

ORG ADRESA

Oznámí prohlížeči, že toto místo programu bude na dané adrese a další instrukce se uloží za něj. Díky tomu si překladač správně spočítá adresy.

DB ČÍSLO [,ČÍSLO, …]

Uloží do výsledného kódu bajt (nebo bajty). Místo čísla můžeme zapsat i řetězec do uvozovek – překladač s ním naloží jako se seznamem ASCII kódů.

DW ČÍSLO [,ČÍSLO, …]

Uloží do výsledného kódu šestnáctibitové číslo jako dva bajty. Pokud je víc parametrů, uloží víc dvojbajtových hodnot.

DS POČET

Řekne překladači, že má vynechat určitý počet bajtů. Používá se v případech, že chceme nechat nějaký prostor volný (např. pro buffer) a je nám jedno, jaký bude jeho obsah na začátku, stačí, když bude N bajtů dlouhý.

NÁVĚŠTÍ EQU HODNOTA

Přiřadí jménu návěští nějakou hodnotu. Třeba z minulé lekce:

```
1   PRTOUT EQU 8500h
```

Od této chvíle už nemusíme řešit, jestli jsme nespletli adresu – prostě napíšeme CALL PRTOUT a překladač zjistí, že jsme PRTOUT nadefinovali takovouhle hodnotu, tak ji použije.

POZNÁMKA K NÁVĚŠTÍM

Každé jméno návěští je unikátní. To znamená, že pokud použijete stejné jméno znovu k označení nějaké pozice v paměti, nebo pokud už nadefinovanému jménu budete přiřazovat hodnotu pomocí EQU, překladač vám vyhodí chybové hlášení. Je to logické – kdybyste si definovali dvakrát stejný název, překladač by nevěděl, jaká adresa platí…

Následující pseudoinstrukce jsou popsané tak, jak je implementuje překladač ASM80. V jiných překladačích se jejich syntax může lišit.

.INCLUDE JMÉNO-SOUBORU

Na místo této instrukce vloží obsah zadaného souboru, jako byste ho tam vložili pomocí copy-paste. Ale asi není potřeba vysvětlovat víc. Všimněte si tečky na začátku pseudoinstrukce… Správně by se měla psát u všech pseudoinstrukcí (je to taková docela užitečná konvence), ale z historických důvodů se u těch výše zmíněných nepoužívá.

.IF VÝRAZ – .ENDIF

Překladač vyhodnotí výraz, a pokud je nenulový, pokračuje v překladu. Pokud je nula, vynechá všechny řádky až do nejbližšího .ENDIF – obdoba podmíněných bloků třeba u preprocesoru jazyka C.

.IFN VÝRAZ – .ENDIF

Stejná konstrukce, jen obrácená podmínka: následující instrukce až do .ENDIF se překládají, pokud je výraz nulový!

BCD a přerušení

Už se blížíme ke konci, už vysvětluju to, co jsem průběžně odsouval…

Kód BCD (Binary Coded Decimal) je způsob zápisu desítkových čísel v šestnáctkové soustavě tak, že se místo znaků 0-F používají jen znaky 0-9 a když k 09h přičtete jedničku, nedostanete 0Ah, ale 10h. Odborné vysvětlení si můžete přečíst na Wikipedii (BCD), já se přiznám, že mi to moc jasné nebylo. Vezmu to tedy „vlastními slovy".

Někdy se hodí vypsat číslice takovým způsobem, jakému rozumějí normální smrtelníci. Ti, jestli si ještě vzpomínáte, považují za číslice jen ty znaky od nuly do devítky, a kdybyste na ně vyrukovali s tvrzením, že A až F jsou taky číslice, tak vám nebudou rozumět. A ačkoli se někteří programátoři (pohříchu často vyrostlí na assembleru) domnívají, že člověk se má buď naučit rozumět počítačům tak, jak je přirozené strojům, nebo na ně nešahat a jít pracovat do marketingu, je převažující postoj opačný. Takže i v tom assembleru nezaškodí čas od času nasimulovat starou dobrou desítkovou soustavu.

Jestli jste si zkusili převést strojově osmibitové číslo na desítkovou hodnotu, dáte mi zapravdu, že to není úplně procházka růžovou zahradou. Dokonce ani počet bitů na číslici není vždy stejný… Proto vznikl kód BCD, kde se do jednoho osmibitového čísla vejde desítkové číslo z rozsahu 0 – 99. Pokud je číslo v kódu BCD, tak 99 (1001 1001) neznamená 153 desítkově, ale 99 desítkově.

Což, jako vyjádření čísla dobré, ale teď – jak se s tím pracuje? Procesor na to má nástroj.

Představte si, že sečtete dvě čísla, řekněme 13h a 28h. To máme 3Bh (je to totiž desítkově 19 + 40 = 59). A přesně to nám spočítá následující kód:

1	.ORG 0
2	MVI A, 13h
3	ADI 28h

Pokud budeme obě čísla považovat za čísla v kódu BCD, tedy vlastně třináct a dvacet osm, budeme očekávat, že výsledek bude 41 – v BCD kódu 41h. Jak tedy sečíst dvě čísla v kódu BCD? Zcela normálně: sečteme je běžnou instrukcí ADD/ADI, a po ní použijeme instrukci DAA.

DAA

znamená „Decadic Adjust after Addition", neboli Desítková úprava po sčítání. Tahle instrukce vezme příznakové bity (zejména bit AC, který se nikde jinde nepoužívá) a pomocí nich upraví výsledek (3Bh) tak, aby byl v kódu BCD (41h).

Při odčítání platí

Požadavek na přerušení!

V procesorových systémech se občas stane něco významného a důležitého, na co je potřeba okamžitě zareagovat. Třeba někdo zmáčkne klávesu na klávesnici. *(Aspoň vidíte, co je ve světě procesorů Opravdu Důležité – kdyby se náhodou vznal a hořel, bylo by mu to jedno, ale zmáčknutý čudlík zasluhuje okamžitou reakci!)* Nebo přijdou nějaké údaje ze sériového rozhraní a je třeba je přečíst, než tam někdo pošle další a tenhle se ztratí. Nebo se stane to, že uplyne nějaký předem nadefinovaný čas – tedy něco jako minutka na vaření vajíček. V takovém případě požádá to zařízení, které má *výjimečnou událost*, procesor o přerušení. Ten všeho nechá, odskočí si do podprogramu, který obslouží, co je potřeba, a pak se zase hezky vrátí zpátky k tomu, co měl rozdělané.

KONEC!

… to samé jako při sčítání. Stačí použít instrukci DAA, která upraví výsledek tak, aby zase obsahoval číslo v kódu BCD.

Po pravdě řečeno – s instrukcí DAA se člověk moc nesetkává. Kupodivu i spousta algoritmů na převod čísel z desítkové do šestnáctkové soustavy a zpátky se bez téhle instrukce obejde. Proto je to takový docela otloukánek, většinou ho vysvětlují všichni až na konci… kromě špatných knih o programování v assembleru x86, které probírají instrukce ne podle jejich složitosti nebo funkce, ale podle abecedy. Tam se totiž ta samá instrukce jmenuje AAA! Představte si, co byste říkali, kdybych na vás jako první instrukci vytáhnul právě DAA a BCD kód!

Požadavek na přerušení!

Přerušení je něco jako telefonát – taky vás zaskočí v nejnevhodnější chvíli a vy se mu musíte hned teď věnovat. Proto je dobré, aby jeho obsluha byla co nejkratší („Ano, pane řediteli! – Jistě pane řediteli! – Tak to jste se asi pos*al, pane řediteli! – Ano, na personální si dojdu zítra ráno! Nashledanou!") Vyřídit to nezbytné, ať se můžeme vrátit k poctivé práci.

KONEC!

BCD kód bych, s dovolením, považoval tímto za probraný – ještě se k němu vrátíme v nějakých ukázkách algoritmů, ale víc už asi moc ne. Namísto toho se vypořádáme s přerušením…

DI, EI

Jak přerušení vypadá, to jste názorně mohli vidět v předchozím textu. Naštěstí existují instrukce DI (ta přerušení zakáže) a EI (ta ho opět povolí). DI použijeme ve chvíli, kdy procesor potřebuje dodělat svou práci a je to důležitější než obsloužit periferie – třeba něco časově kritického. V zásadě by mohl mít zakázané přerušení neustále (a některé systémy to tak mají), ale zase na druhou stranu, když zůstanu u alegorie s telefonem: Nikdo se vám nedovolá!

Teď tedy používám jedno virtuální DI, aby už další výklad žádné přerušení nerušilo, a řekneme si, jak to vlastně funguje.

Procesor má vstup INT. Za normálních okolností by na něm měla být nula, a taky tam je. Pokud některá část počítače (klávesnice, sériový port, časovač, …) požaduje přerušení, nastaví příslušné obvody tento vstup na hodnotu 1. Procesor dokončí

aktuální instrukci a zkontroluje stav tohoto vstupu. Když je 0, pokračuje dál, pokud je ale 1, a zároveň je přerušení povolené, udělá následující kroky:

1. Zakáže přerušení
2. Potvrdí, že požadavek převzal (tj. ve stavovém slovu pošle informaci INTA – Interrupt Acknowledge)
3. Přečte z datové sběrnice kód instrukce, kterou má vykonat. Je to jedna z instrukcí RST0-RST7, nebo instrukce CALL.
4. Načtenou instrukci pak provede.

Všimněte si, že jsem nepsal, že ji čte „z paměti", ale „ze sběrnice". Je na systému, aby v takovém stavu připravil procesoru tu správnou instrukci.

Asi nejjednodušší způsob, který lze dosáhnout zapojením jednoho odporu, je ten, že vždy, když procesor vyšle INTA a čeká na instrukci, tak mu pomocný obvod 8228 vrátí RST 7. Vzpomeňte si, co jsme si říkali o instrukci RST n: Je to volání podprogramu na adrese 8*n. RST 7 tedy zavolá podprogram na adrese 38h a tam by měla proběhnout obsluha přerušení.

Složitější způsoby, které využívají třeba obvod 8214, dokážou obsloužit osm různých požadavků podle jejich priorit a adekvátně k nim zavolat různé instrukce RST. Specializované řadiče přerušení (8259) dokážou například cyklicky měnit priority nebo volat ne osm pevně daných instrukcí, ale pomocí instrukce CALL zavolat libovolnou adresu.

Ale u těch jednodušších systémů je většinou vše zapojeno tak, že INT znamená vykonání instrukce RST 7.

Obslužná rutina by měla být co nejkratší a nejrychlejší. Nezapomeňte na konci zase povolit přerušení instrukcí EI, jinak obsluha proběhne jen jednou. A pokud náhodou budete psát obsluhu přerušení pro různé RST, uvědomte si, že na to máte jen osm bajtů – nejčastěji ty první tři použijete pro JMP a pět zůstane volných.

HLT

Víte, co dělá procesor v počítači většinu času? Čeká. Čeká, až mu něco řeknete, pak to chvíli zpracovává, a pak zase čeká. Mezi zmáčknutím kláves čeká. Když zmáčknete dvě klávesy za sekundu, tak za tu dobu provede 8080A rovné dva miliony taktů. Obsluha každého zmáčknutí zabere, no, ať nežeru, i třeba 500 taktů! Procesor tedy 1000 taktů pracoval a 1 999 000 taktů jen čekal, jestli se nestane něco zajímavého. Váš den, kdybyste byli procesor, by vypadal tak, že byste 43 sekund dělali něco zajímavého, a zbývajících 23 hodin 59 minut a 17 sekund by se nedělo nic. Vůbec nic. (A tohle jim děláme neustále, mimochodem!)

Instrukce HLT zastaví procesor. Stojí na místě, neděje se nic, žádné instrukce neprovádí, zkrátka stojí. Stojí a stojí. Nic nedělá. Šmitec, šlus, finito. Z téhle letargie ho probudí buď RESET, nebo právě požadavek na přerušení. Představte si, jak ten procesor zpracovává instrukci HLT, stojí stále na místě, jakoby v nekonečné smyčce, když v tom přijde požadavek na přerušení. Procesor udělá ty kroky, které jsem popsal a provede (třeba) instrukci RST 7. Ta vykoná nějaké operace a vrátí se za instrukci HLT. Program tak může pokračovat. Často se tenhle fígl používá např. k tomu, že si nastavíme časovač (onu „minutku") a procesor pomocí HLT zastavíme. Až doběhne požadovaný časový interval, přijde přerušení, a to nás vyvede ze stavu HALT.

Sledujte: **HLT!**

Instrukce 8080 – práce s periferiemi

Pracovat s pamětí je fajn, ale pro uživatele je paměť neviditelná. Uživatel ocení třeba nějaké to písmeno na displeji, nějaký ten zvuk, chce zmáčknout tlačítko, zakvrdlat joystickem či tak něco…

Počítačové periferie jsou „všechno co není procesor nebo paměť". Takže třeba ty klávesnice, výstupní zařízení, magnetofony, reproduktory, … Ty se nějakým způsobem připojují do systému (jakým přesně, to záleží na tvůrci systému) a procesor s nimi komunikuje.

Některé procesory (6502 například) nerozlišují periferie od paměti. Zkrátka a dobře určité adresy z adresního rozsahu neobsadí paměť, ale periferie. „Pošli znak 12 sériovou linkou" je pro ně stejná operace jako „ulož znak 12 do paměti na konkrétní adresu". Má to svoje výhody, ale i nevýhody.

Procesor 8080 (nebo Z80) naproti tomu mají speciální vývody, které říkají: Teď se nepřistupuje do paměti, ale ke vstupně-výstupnímu zařízení, k periferii. Paměť může zůstat neaktivní a adresa čtení nebo zápisu se vztahuje na periferní zařízení (je pro to takové slovo „port" – „přistupujeme na porty" třeba). U procesoru 8080 může být až 256 periferních zařízení, protože pro tyto operace používá osmibitovou adresu.

IN, OUT

Procesor 8080 má dvě instrukce pro práci s periferiemi. IN slouží ke čtení, OUT k zápisu. Obě instrukce mají jeden přímý parametr, a tím je osmibitová adresa periferie. Instrukce IN F8h přečte bajt z periferie na adrese F8h a uloží ho do registru A. OUT 23h vezme obsah registru A a pošle ho na periferii na adresu 23h.

Víc k těmto instrukcím asi nelze říct. Jsou opravdu tak jednoduché, jak vypadají. Konkrétní adresy už jsou na tvůrci toho kterého systému.

Například v počítači PMI-80 je na adresách F8h-FBh připojen obvod 8255, který slouží jako programovatelný obvod pro řízení vstupů a výstupů. Tentýž obvod v PMD-85 sídlí na adresách F4h-F7h (a tam obsluhuje klávesnici, LED a reproduktor), další stejného typu je na adresách F8h-FBh (a tam se stará o načítání dat z ROM modulu), no a na adresách 1Eh a 1Fh je připojený obvod 8251, který pro změnu slouží pro komunikaci s kazetovým magnetofonem.

Ostatně, když už jsem zmínil klávesnici…

KLÁVESNICE

Když jsem v minulém dílu psal o tom, že stisk klávesy vyvolá přerušení, tak, po pravdě řečeno, spíš ne… Totiž, ne že by to nešlo, ne že by to nebyl dobrý nápad, ale v dobách největšího rozmachu osmibitů snad žádný z nich nepoužíval přerušení při stisku klávesy (vzpomínám jen na jednu zvláštnost, a tou byla konstrukce klávesnice z nějakého Amatérského Rádia, která vyvolala přerušení při stisku klávesy). Naprostá většina systémů v té době používala zcela jiný princip.

Klávesy na klávesnici se zapojovaly do matice N x M, kdy každá klávesa sídlila v nějakém průsečíku. Jedno z těchto čísel bylo často 8, aby se líp připojovala k periferním obvodům. Například klávesnice u ZX Spectra byla organizovaná do 8 řádků po 5 sloupcích. Řádky byly připojeny na vstupní port (s klidovým stavem 1), sloupce na výstupní. Když chtěl programátor vědět, jaká klávesa je stisknuta, poslal postupně 0 na jednotlivé vodiče sloupců a pokaždé si přečetl hodnotu řádků. Pokud byly všude 1, znamenalo to, že v tom sloupci není žádná klávesa stisknuta, když našel 0, věděl, že našel nějakou stisknutou klávesu.

U jednodeskových počítačů platilo totéž. Stejně byla zapojena klávesnice PMI-80, PMD-85, IQ-151 a dalších. U Atari 800 třeba taky, ale tam se o tohle testování stisknuté klávesy nestaral procesor, ale specializovaný obvod POKEY. U Commodore C64 byla rovněž matice a specializovaný obvod CIA.

DISPLEJ

Některé osmibitové počítače, zejména jednodeskové, měly jako displej nejrůznější sedmisegmentovky. Ty se většinou připojovaly přes nějaký port. U počítače BOB-85 měly dokonce i latche, tj. „paměti" aktuálního stavu; naopak u PMI-80 bylo potřeba postupně vysílat informace o tom, co se má na které sedmisegmentovce zobrazovat. Displej byl připojen podobně jako klávesnice a během čtení jednotlivých sloupců se také rozsvěcely jednotlivé sedmisegmentovky…

Displeje u „větších“ počítačů byly řešeny buď jako znakové (IQ-151, SAPI-1), nebo grafické (PMD, Spectrum, …) – některé systémy měly inteligentní řadiče, které se dokázaly přepnout do různých módů. Ať tak či onak, většinou se k displeji přistupovalo jako k paměti (přesněji řečeno: řadič displeje si sahal do vyhrazené oblasti v paměti RAM a zobrazoval odtamtud data).

Konec? Ani náhodou!

V tomto místě jsme skončili s přehledem instrukcí procesoru 8080. Jenže v assembleru víc než v jiném jazyce platí: mýlí se ten, kdo zaměňuje znalost jazyka za znalost jeho příkazů. Takže nebojte, zanedlouho se zase setkáme a budeme pokračovat. Ukážeme si, jak v assembleru řešit nějaké úlohy, a hlavně: máme před sebou ještě minimálně dva procesory!

Instrukce 8080 – přehledně v jedné tabulce

Tak si to shrňme, ano?

ZOBRAZ ZÁZNAMŮ

HLEDAT:

OPERAČNÍ KÓD	INSTRUKCE	POČET BYTE	OVLIVŇUJE PŘÍZNAKY	FUNKCE
0x00	NOP	1		
0x01	LXI B,D16	3		B <- byte 3, C <- byte 2
0x02	STAX B	1		(BC) <- A
0x03	INX B	1		BC <- BC+1
0x04	INR B	1	Z, S, P, AC	B <- B+1
0x05	DCR B	1	Z, S, P, AC	B <- B-1
0x06	MVI B, D8	2		B <- byte 2
0x07	RLC	1	CY	A = A << 1; bit 0 = prev bit 7; CY = prev bit 7
0x08	-			
0x09	DAD B	1	CY	HL = HL + BC
0x0a	LDAX B	1		A <- (BC)
0x0b	DCX B	1		BC = BC-1
0x0c	INR C	1	Z, S, P, AC	C <- C+1
0x0d	DCR C	1	Z, S, P, AC	C <-C-1
0x0e	MVI C,D8	2		C <- byte 2
0x0f	RRC	1	CY	A = A >> 1; bit 7 = prev bit 0; CY = prev bit 0

Zobrazuji 1 až 16 z celkem 256 záznamů

PředchozíDalší

No related posts.

Trocha assemblerové teorie

Tuhle jsem do rozhovoru pro programátorský magazín (ve skutečnosti Zdroják) říkal: *ASM80 je dvouprůchodový assembler*. Prý to mám vysvětlit, co to znamená…

V dobách osmibitů byl „dvouprůchodový assembler“ de facto standard. Ale co to znamená? Musíme si říct, jak vlastně překladač funguje.

Bere řádek po řádku a kouká se, jestli tam je instrukce nebo pseudoinstrukce. Pokud ano, začlení si její kód do připravovaného výsledného útvaru, popř. provede to, co pseudoinstrukce nařizuje. Tím, že už v téhle fázi připravuje kód, tak vlastně ví, jak dlouhá která instrukce bude. Takže si může průběžně vytvářet i tabulku adres, kde má ke každému symbolickému jménu uloženou jeho adresu (nebo hodnotu). To je první průchod.

Ve druhém průchodu tohle všechno už má připravené, takže jede znovu, a jen na místech, kde je potřeba vyčíslit nějakou konkrétní hodnotu s odkazem, spočítá jeho hodnotu a doplní do kódu.
Takhle tedy funguje ASM80. Fungoval stejně i GENS (např.) Něco podobného (podobného!) dělal i Prométheus, ale ten, nakolik jsem pochopil, dělal část prvního průchodu už při editaci textu, kdy si jednotlivé instrukce „předpřekládal".
Co jiné počty průchodů? Tříprůchodový? Jasně, existují. Dokážou třeba vyřešit problém několikanásobné dopředné reference, navíc by mohly trošku optimalizovat (plné skoky nahradit relativním, dlouhé varianty instrukcí těmi se zero page u 6502 a podobně).
Šel by napsat jednoprůchodový assembler? No jasně, šel – představte si první průchod, při něm to vyhodnocování výrazů… Když narazí na adresu, kterou už zná, tak ji použije, když ji ale ještě nezná, zapíše si do tabulky symbolických názvů tenhle název s poznámkou: „Až na tohle návěští narazím, doplním ho tam a tam".
Počet průchodů tedy dokáže ovlivnit některé aspekty chování překladače (dopředné reference by jednoprůchodový zvládnul).
A kolikaprůchodové se používají dneska?
Turbo assembler, poslední, který jsem na PC platformě používal, je „multiprůchodový". Ale vzhledem k jeho komplexnosti to ani jinak nejde. Ten hlavní rozdíl proti „starým pákám" je v tom, že assembler dneska (ani včera) nepřipraví hotový binární kód pro spuštění. Sestaví něco, čemu se říká „objektkód", což je v podstatě výsledek posledního průchodu, ale nevadí mu, když nějaké adresy nerozpoznal. Pravděpodobně patří nějaké knihovní rutině. Objektkód pak dostane do spárů program, kterému se říká „linker", a ten spojí vše potřebné – všechny části kódu, knihovní rutiny, data, vše. Teprve když tady nějaký symbol chybí, tak je zle. Překladač oddělený od linkeru má tu výhodu, že program může být napsaný v různých jazycích, v jednom třeba výpočty, v druhém UI, knihovny ve třetím – a linker to poslepuje dohromady. Toto rozdělení samozřejmě není nijaká novinka. Není to dokonce ani výmysl šestnáctibitového světa. I osmibity měly překladače a linkery zvlášť, např. u CP/M. Tohle rozdělení funguje tam, kde není problém pracovat se soubory. U osmibitového domácího počítače s kazeťákem by to moc smysl nedávalo.

8080 – algoritmy 1

Algoritmy! Neseme čerstvé algoritmy! Berte, paní, jsou zadarmo a přitom tak užitečné!

PŘESUN BLOKU DAT

Máme 32 bajtů na nějaké adrese a chceme je zkopírovat na adresu úplně jinou. Jak na to? Jednoduše – použijeme instrukci LDIR a… cože? Ajo vlastně, instrukci LDIR má až procesor Z80, u 8080 nic takového není, tam si to musíme pořešit jináč. Třeba takhle:
Nejprve si naplním registry patřičnými údaji – do registrů HL dám adresu, kde se blok dat nachází, do registrů DE adresu, kam chci blok přesunout, do registrů BC počet bajtů, které chci přesunout. Ve smyčce LDIR (která se víceméně chová shodně s instrukcí LDIR procesoru Z80, až na to, že tady je pomalejší a přepisuje obsah registru A) se děje následující: Do registru A vezmu bajt z adresy HL, uložím ho na adresu DE, obě adresy zvýším o 1, od počítadla BC odečtu jedničku, a když je BC nenulové, opakuju smyčku.
Bohužel (tedy někdy spíš bohudík) instrukce INX, DCX nemění stav příznaků. Což je dobře, jak uvidíme v dalších algoritmech, ale v případě, jako je tento, by se nám hodilo, kdyby DCX dokázala nastavit příznak Z, jako že je výsledek nula. Ale nedělá to, takže si to musíme vyřešit jinak (zvykejte si, to je assembler!)
Každý podobný případ lze řešit několika způsoby, ale časem se ustálí vždy jeden *návrhový vzor*, který má nejméně nevýhod. V případě „zkontroluj, jestli je ve dvojici registrů 0" se používá ten, který jsem ukázal v kódu: do A si zkopíruju jeden registr z dvojice, udělám OR s druhým (vzpomeňte – výsledkem OR je nula, pokud všechny bity obou operandů jsou nulové) – a protože OR už příznakový bit Z změní podle výsledku, tak jsme získali, co jsme chtěli.
Jak bychom to ale řešili, kdyby v A bylo něco, co chceme zachovat? Řešit by to samosebou šlo, jen – jinak a náročněji.

PŘEPSÁNÍ BLOKU DAT

V procesoru Z80 se instrukce LDIR používá i k jiné operaci, než je přesun bloků dat – totiž k mazání (přesněji k nastavení celého bloku na konkrétní hodnotu). Využívá se toho, že nastavíme registry HL a DE určitým způsobem – HL na začátek bloku, který se má smazat, DE na adresu o 1 vyšší. Ručně pak nastavíme první bajt bloku (ten, kam ukazuje HL) na požadovanou hodnotu a LDIR postupně první bajt bloku nakopíruje na další a další adresy. BC je v takovém případě (délka bloku – 1). Pokud nastavíme DE na adresu větší než HL+1, třeba HL+4, vyplní LDIR blok paměti vzorkem dat (HL, HL+1, HL+2, HL+3).
Fungovalo by to i s tím naším LDIRem? Ale určitě fungovalo, jen je to v případě nastavení nějaké oblasti v paměti na konkrétní hodnotu trošku kanón na vrabce (a to v assembleru znamená vždy: stojí to paměť a čas). Jednodušší postup je zde:

Všimněte si, že hodnota, která se do bloku paměti zapisuje, není v registru A, ale v registru E. Je to právě kvůli výše popsanému postupu na testování BC na nulu – využívá registr A. Kdybychom v něm měli tu hodnotu, přepsali bychom si ji. Abychom si ji nepřepsali, museli bychom si ji někam uložit (pravděpodobně třeba do toho registru E), pak zase načíst zpátky… Výsledek by byl sice funkční, ale pomalejší a delší. A to jen kvůli tomu, že chceme číslo v jiném registru?! Kdepak.

DĚLENÍ

Tuhle jsme si ukazovali, jak násobit dvě celá čísla bez znaménka. Dělení můžeme napsat analogicky – buď jako postupné odčítání, nebo pomocí rotací a odčítání. Ten druhý postup opět můžeme napsat bez triků, nebo s trikem…
Tady je postup pro dělení dvojregistru HL registrem D. Matematicky: HL = HL / D, zbytek po dělení je v registru A. Je to postup s trikem (používá se „trojitý registr" AHL, do A se postupně nasouvají bity z HL a kontroluje se, jestli už je mezistav větší než dělitel).

Ukázali jsme si pár triků a předvedli pár algoritmů. Ovšem v programování obecně a pro assembler zvlášť platí, že:
1. Není důležité si tyhle algoritmy pamatovat, důležitější je umět je napsat, když je potřeba
2. Je dobré si tyhle algoritmy pamatovat, aby je člověk nepsal zbytečně znovu
3. Nejdůležitější je dokázat se rozhodnout, jestli pro daný konkrétní případ platí 1 nebo 2!

Instrukce 6502 v jedné tabulce

ZOBRAZ ZÁZNAMŮ
HLEDAT:

OP. KÓD	OP. KÓD (HEX)	INSTRUKCE	DÉLKA	ADRESNÍ MÓD
0	00	BRK	1	imp

OP. KÓD	OP. KÓD (HEX)	INSTRUKCE	DÉLKA	ADRESNÍ MÓD
1	01	ORA	2	izx
2	02	-	0	
3	03	SLO	2	izx
4	04	-	0	
5	05	ORA	2	zpg
6	06	ASL	2	zpg
7	07	SLO	2	zpg
8	08	PHP	1	imp
9	09	ORA	2	imm
10	0A	ASL	1	ima
11	0B	ANC	2	imm
12	0C	-	0	
13	0D	ORA	3	abs
14	0E	ASL	3	abs
15	0F	SLO	3	abs
16	10	BPL	2	rel
17	11	ORA	2	izy
18	12	-	0	
19	13	SLO	2	izy
20	14	-	0	
21	15	ORA	2	zpx
22	16	ASL	2	zpx
23	17	SLO	2	zpx
24	18	CLC	1	imp
25	19	ORA	3	aby
26	1A	-	0	
27	1B	SLO	3	aby
28	1C	-	0	
29	1D	ORA	3	abx
30	1E	ASL	3	abx
31	1F	SLO	3	abx

Zobrazuji 1 až 32 z celkem 256 záznamů
PředchozíDalší
No related posts.

Procesory a procesory

Už jsem zmiňoval, že mezi mikroprocesory jsou poměrně zásadní rozdíly – čímž samozřejmě nemyslím to, že je každý jiný, ale mám na mysli rozdíly v přístupu, takříkajíc *filosofické*.

Ukázali jsme si procesor Intel 8080. Tento procesor vychází ideově z předchozího procesoru téže firmy, totiž 8008, a ten svou architekturou zase vyšel z procesoru terminálu Datapoint 2200. Původně měl být v tomto terminálu použit, ale Intel ho nestihl

vyvinout a vyrobit včas, takže nakonec v Datapointu byl procesor sestavený z SSI a MSI obvodů a Intel navrhl univerzální procesor 8008.
Jeho nejznámější rival byl vyvinut firmou Motorola a byl označen 6800. Motorola při návrhu vyšla z procesoru počítačů PDP-11. Některé rysy měl procesor 6800 shodné s procesorem 8080 (třeba 16bitovou adresovou sběrnici, osmibitovou datovou sběrnici), ale ve spoustě věcí se lišil.
6800 používá zápis vícebajtových čísel stylem Big Endian (8080 stylem Little Endian). Pokud máte na adresy 0000 a 0001 zapsat hexadecimální číslo 1234h, pro procesor 8080 ho zapíšete tak, že na nižší adresu zapíšete nižší bajt (34h), na vyšší adresu vyšší (12h). U Motoroly 6800 to bude přesně naopak. Ve výpisu paměti na jednom řádku to pak bude vypadat takto:

```
1    Intel:    0000: 34 12 xx xx
2    Motorola: 0000: 12 34 xx xx
```

Procesor od Motoroly nerozlišuje paměť a porty a pracuje s obojím stejně. Nemá tedy pro čtení a zápis z/do portů speciální instrukce, ale používá standardní instrukce pro práci s pamětí, které jsou mnohem flexibilnější. Nevýhodou je, že se tak návrháři připravují o souvislý prostor 64kB pro paměť, protože část musí vyhradit pro porty.
Motorola použila v procesoru 6800 pouhé dva osmibitové registry (A a B) a jeden šestnáctibitový indexový regist X. (Ukazatel zásobníku SP a programový čítač PC jsou i zde, oba šestnáctibitové).
Na první pohled to může vypadat jako nedostatečné – vždy´t některé algoritmy, co jsme si ukazovali u procesoru 8080, využily klidně šest, sedm registrů. Jak to tedy řešit u 6800?
Procesor 6800 přišel s jednoduchým trikem, který po něm převzali i jeho následníci. Paměť si rozdělil na 256 stránek po 256 bajtech (tj. horní byte adresy je „číslo stránky" a dolní „adresa ve stránce"). Stránka 0 pak má speciální roli, protože je snadno adresovatelná. Co to znamená?
Procesor 8080 měl u spousty instrukcí pevně dáno, s jakými daty pracuje. Buď s registrem, s přímo zadanou konstantou, nebo s obsahem paměti, adresovaným registry HL nebo 16bitovou adresou. 6800 naproti tomu měl instrukční sadu mnohem víc *ortogonální*.
Ortogonální instrukční sada je taková, kdy instrukce může pracovat s různými daty naprosto stejně, bez ohledu na to, jestli jsou v registru, nebo v paměti.
Procesor 6800 proto u instrukcí nabízel několik adresních módů. Operand mohl být určen:

1. Implicitně. Například instrukce INCA pracuje s registrem A
2. Bezprostředně (Immediate). LDAA #$02 nahraje do registru A (LoaD A) konstantu „bezprostředně zapsanou" za operačním kódem – tedy 02h
3. Přímou adresou (Direct). LDAA $02 nahraje do registru A hodnotu, která je uložená na adrese, co je zapsaná za operačním kódem. Adresa je pouze osmibitová, vyšších osm bitů se doplní nulami, adresa je tedy 0002h. Instrukce zabírá dva bajty, první je operační kód, druhý je zkrácená adresa – tedy vlastně „adresa ve stránce 0"
4. Rozšířenou adresou (Extended). Obdoba předchozího zápisu, ale adresa je dvojbajtová, lze tedy adresovat paměť v celém rozsahu. LDAA $1234 nahraje do registru A hodnotu z adresy 1234h
5. Indexovanou adresou, tedy adresou, která vznikne součtem hodnoty z indexového registru X a osmibitové konstanty.
6. Relativně. V takovém případě se k adrese PC přičetla hodnota konstanty v rozsahu -128..+127

Pokud programátor chce k obsahu akumulátoru (tedy registru A nebo B) přičíst nějaké číslo, použije instrukci sčítání a podle adresního módu určí, jestli se bude zadávat přímo nebo z paměti, a pokud z paměti, tak jestli plnou adresou (výhoda: může být kdekoli, nevýhoda: instrukce je o bajt delší a časově náročnější), nebo ze stránky 0, nebo pomocí indexového registru… Hodnota může být přitom klidně načtena z portu, protože, jak jsme si už řekli, procesor 6800 nerozlišuje mezi porty a pamětí a ke všemu přistupuje stejně.
Z popisu adresních módů vyplývá, že stránka 0 je snadno dostupná a je to takový kompromis mezi plnou adresou a registrem v procesoru. Přístup k těmto 256 buňkám paměti je rychlý a instrukce jsou i kratší, lze je tedy považovat za určitý ekvivalent vnitřních registrů procesoru.
Assemblerem procesoru 6800 se zabývat nebudu, přesto jsem ho tu uvedl. Z architektury 6800 totiž vyšel nejen legendární procesor 6502 (kterému se věnovat budu), ale i další procesory – třeba „poslední osmibitový procesor" 6809 (samozřejmě v uvozovkách – tento procesor měl na tehdejší dobu ojedinělé možnosti, ale přišel bohužel pozdě) a celá řada procesorů počínaje procesorem MC68000 (což nebyl „osmibit roztáhnutý na 16 bitů", ale regulerní 32bitový procesor, ořezaný na 16 bitů, s vysoce ortogonální instrukční sadou, dvěma režimy procesoru apod.) přes další procesory téže řady až k MC68060 a dál k PowerPC. Tato řada se v běžných domácích a kancelářských počítačích už nevyskytuje (od přechodu Apple na Intel), ale dodneška se udržuje v embedded zařízeních apod.
Dovolte mi nostalgickou odbočku. Po 8080 jsem přešel na Z80 (byl jsem Spectrista) a tenhle procesor mi učaroval. Stovky instrukcí, mnoho registrů, šestnáctibitová aritmetika… Na 6502 jsem se díval s totálním nepochopením: vždyť to má jen tři registry a je to pomalé (=má to malou frekvenci). ve skutečnosti byla „datová propustnost" srovnatelná. Ostatně operace s pamětí v zero page zabraly 3 takty procesoru, nejrychlejší operace s pamětí u Z80 trvala 7 taktů. Krásu téhle architektury jsem docenil až později, když jsem se naučil assembler 68000 (Atari ST, Amiga). V čerstvé paměti jsem měl tehdy naučený asembler 8086, se všemi jeho „segmenty" a „instrukcemi, co používaly vždy přesně určené registry" – a najednou máte k dispozici ortogonální sadu a osm adresních + osm datových registrů. Třeba instrukce PUSH (která tam vlastně nebyla) fungovala jako obyčejný přesun hodnoty z registru do paměti s adresním módem „přenes do paměti na adresu danou některým adresním registrem, ale předtím hodnotu sniž" (predekrement).
A po obecném úvodu pojďme už na assembler 6502!

Architektura 6502

AUTOR: MARTIN MALY PUBLIKOVÁNO 09. 02, 2014

Kurz assembleru 6502 začnu popisem jeho architektury a rozdílů proti (alespoň v ČR a SR) známější architektuře x80.
Doufám, že jste nepřeskočili předchozí díl, kde jsem popisoval základní rozdíly mezi architekturou procesorů 8080 a 6800. Téměř všechno, co jsem psal o architektuře 6800, lze vztáhnout na procesor 6502, s jedinou výjimkou, a tou je zápis vícebajtových hodnot. Na rozdíl od 6800 používá 6502 zápis „Little Endian", tedy stejný jako 8080 (nejprve je nižší byte, pak vyšší).
Procesor 6502, stejně jako jeho vzor 6800, nesází na velký počet interních registrů. Kromě programového čítače (PC), ukazatele zásobníku (SP) a registru příznaků má pouhé tři registry. Jeden je akumulátor, vůči němu se provádějí matematické a logické operace, a dva mají funkci indexových registrů pro přístup do paměti (registry X a Y). Kromě PC jsou všechny registry osmibitové (PC má 16 bitů).

REGISTRY 6502											
15	14	13	12	11	10	09	08	07	06	05	04
									Akumulátor		
			—								
									Indexové registry		
			—								
			—								
									Ukazatel zásobníku		
0	0	0	0	0	0	0	1				
									Programový čítač		
							PC				
									Stavový registr (registr příznaků)		
			—					N	V	1	B

Jak je to tedy zařízeno se zásobníkem, jestliže ukazatel má jen 8 bitů? Vyšší byte je vždy roven 01h, takže zásobník sídlí na adresách 0100h-01FFh. Ano, má maximálně 256 bajtů. Při zápisu se, stejně jako u 8080, postupuje směrem k nižším adresám. Když dojde na konec, tj. na adresu 0100h, pokračuje se zase od začátku (od 01FFh). Zásobník tedy funguje ve *stránce 1*.

Princip paměťových stránek jsme si popisovali. Zmiňoval jsem, že u 6800 má paměťová stránka 0 speciální význam – existuje adresovací mód, který místo kompletní 16bitové adresy používá pouze osmibitovou (a horní byte je vždy 00h). Tento mód je, nepřekvapivě, nazván „zero page".

Takže u 6502 máte k dispozici tři vnitřní registry (A, X, Y) a 256 bajtů nulté stránky (v praxi si ale velkou část této oblasti sebere pro sebe operační systém, takže aplikačním programátorům už moc nezbývá). Tento koncept vychází z možností tehdejší techniky, kdy paměti byly rychlejší než procesory a dobře navržený čip (jako je třeba právě 6502) zvládne na tři takty procesoru přečíst operační kód, přečíst parametr (adresu v zero page) a přečíst nebo zapsat hodnotu z/do paměti. Srovnejme si časování instrukce, která přenese do akumulátoru obsah paměti na adrese 1234h:

- Intel 8080: LDA 1234h – 3 byte, 13 taktů procesoru
- Z80: LD A, (1234h) – 3 byte, 13 taktů
- 6502: LDA 1234h – 3 byte, 4 takty

U nejobvyklejšího časování zvládne 8080 tímto způsobem přenést 153.846 bajtů za sekundu (2MHz), Z80 za tu dobu přenese 269.230 bajtů (3.5MHz), 6502 přenese 447.500 bajtů (taktováno na 1.79MHz). Omlouvám se, že tu explicitně počítám takovou samozřejmost, ale je na místě ukázat a připomenout, že frekvence procesoru není všechno a nelze říct: *V Atari má procesor 1,79MHz, ve Spectru 3,5MHz, je tedy jasné, co je rychlejší* (a to ani nezmiňuju, že ve starých počítačích nejedou procesory neustále na *plnej kotel*, občas je okolní systém zpomaluje). Nahradit registry přímo na čipu (které jsou sice nejrychlejší, ale poměrně drahé) rychlým přístupem do paměti je (nebo alespoň v 70. letech bylo) docela důvtipný nápad.

6502, stejně jako 6800, nemá speciální systém pro periferie. Návrhář systému musí tedy tyto obvody (klávesnice apod.) namapovat do prostoru paměti. Nevýhodou je, že se tím připravíme o prostor pro paměť, výhodou je, že s periferiemi může programátor pracovat stejně jako s pamětí a využít k tomu libovolnou instrukci, která dokáže číst nebo zapisovat z/do paměti.

Adresní módy 6502

Procesor 6502 je, jak jsme si už řekli, velmi intenzivně závislý na práci s pamětí. Jeho instrukční sada není, jako u procesoru 8080, tvořena spoustou instrukcí, které se od sebe liší podle toho, jestli (například) čtou z paměti přímo adresované (LDA), nepřímo adresované dvojicí HL (MOV A,M) nebo nepřímo adresované dvojicemi BC / DE (LDAX). U procesoru 6502 je jediná instrukce, která se jmenuje LDA, a její funkce je: „Do registru A (akumulátoru) zkopíruj hodnotu operandu".

Co je ale hodnotou operandu? To právě určuje onen adresní mód. Podle toho, jaký mód použijeme, tak se vyhodnotí operand.

IMMEDIATE (IMM)

Tento mód říká, že operandem je ta hodnota, která je na následující pozici za operačním kódem. Instrukce, které používají tento mód, zabírají dva byty (první je operační kód, druhý hodnota)

V assembleru se použití tohoto módu označuje zapsáním znaku # před operand: LDA #23, LDA #0, LDA #23h, LDA #$5A

IMPLIED (IMP)

Též „implicitní mód". Instrukce samotná pracuje vždy se stejným operandem – registrem, hodnotou na zásobníku apod. Proto není potřeba žádný operand dodávat.

ABSOLUTE (ABS)

Operandem je hodnota v paměti, která leží na adrese MMNNh, kde NN je byte, uvedený za operačním kódem na druhé pozici, MM na třetí pozici. Instrukce s absolutním adresováním tedy zabírají tři bajty. V assembleru se zapisuje tento mód plnou adresou: LDA 1234h, LDA $1234

ZERO PAGE (ZP)

Operandem je hodnota v paměti, která leží na adrese 00NNh, kde NN je byte, uvedený za operačním kódem. Instrukce, které adresují zero page, mají dva bajty. Zapisují se stejně jako předchozí, tedy adresou: LDA 12h, LDA $12

Nojo, ale! Jak překladač pozná, jestli chci použít mód abs nebo zp, když napíšu „LDA 12h"? Co když mám na mysli 0012h a chci plnou adresu? A jak to pozná, když je adresa zadaná symbolicky, tedy „LDA promenna"? Adresa by měla být absolutní, ale když je „promenna" v nulté stránce, může použít zp… a jak to zjistí, když třeba v tu chvíli hodnota ještě není známá? Řešení jsou různá. Assembler se může řídit podle toho, jak je hodnota zapsaná (0012h je abs, 12h je zp). Dál může použít speciální instrukci pro definici proměnných v zero page (EQU pro normální, EZP pro zp). Může použít speciální zápis, kterým natvrdo řekne, že

vyžaduje adresaci ZP (např. pomocí hvězdičky: LDA *promenna). Může využít další průběh, ve kterém optimalizuje instrukce, které používají absolutní mód a jejichž operand je menší než 0100h…

Já jsem v ASM80 použil následující postup: Pokud je hodnota zapsaná přímo, nebo pokud je už dřív v kódu jednoznačně definovaná, tak se pro adresy mezi 0000h a 00FFh použije zero page. Pokud je adresa v tu chvíli ještě neznámá, používá se absolutní mód, ale lze vynutit zero page pomocí zápisu s hvězdičkou.

ABSOLUTE INDEXED (ABX, ABY)

Operandem je hodnota v paměti. Její adresa je získána tak, že se k adrese, zapsané v dvou následujících bajtech (jako u absolutního adresování) přičte hodnota registru X nebo Y. Výsledek je použit jako adresa do paměti. V assembleru se zapíše jako číslo, následované čárkou a znakem X (nebo Y): LDA 12h,X

ZERO PAGE INDEXED (ZPX, ZPY)

Operandem je hodnota v nulté stránce paměti. Její adresa je získána tak, že se k bajtu, následujícímu za operačním kódem, přičte hodnota registru X nebo Y. Výsledek se ořízne na osm bitů (pokud tedy vyjde např. 0106h, bude to 06h) a je použit jako adresa v zero page. V assembleru se zapíše jako číslo, následované čárkou a znakem X (nebo Y): LDA 12h,X

RELATIVE (REL)

Operandem je adresa, která je spočítána tak, že se k aktuální hodnotě registru PC přičte hodnota bajtu, který je zapsaný za operačním kódem. Jeho hodnota je brána jako číslo se znaménkem (00 = 0, 07Fh = +127, 080h = -128, 0FFh = -1). Toto adresování se používá u podmíněných skoků.

INDIRECT (IND)

Nepřímé adresování (indirect) spočívá v tom, že adresa toho místa, o které jde, je uložená v paměti na nějaké úplně jiné adrese, a ta je zadána. Instrukce, které využívají nepřímé adresování, zabírají tři bajty – první je operační kód, druhý je nižší a třetí vyšší bajt nepřímé adresy. Procesor vezme tuto nepřímou adresu (NA) a přečte si obsah dvou buněk (NA a NA+1) z paměti. Tento obsah dá dohromady cílovou (efektivní) adresu.

Nepřímé adresování může použít pouze instrukce JMP. Zapíše se pak např. jako JMP (1234h). V takovém případě vezme procesor obsah na adrese 1234h (řekněme, že to je 0DAh) a na adrese o 1 vyšší, tedy 1235h (řekněme, že tam je 0DEh). Tyto dvě hodnoty dají dohromady adresu 0DEDAh, a na tu se skočí.

Aby nebyly věci tak jednoduché, tak procesor 6502 obsahuje chybu, která způsobuje, že nepřímá adresa, končící na FFh (např. 12FFh), nevezme vyšší část adresy z 1300h, ale z 1200h. Ve skutečnosti totiž pouze přičte ke spodnímu bajtu 1, ale případný přenos do vyšší části adresy ignoruje.

INDEXED INDIRECT (IZX)

Tento mód se v assembleru zapisuje jako (nn,X) – tedy podobně jako indexovaný mód, ale v závorkách. LDA (12h,X) vezme parametr (12h), k němu přičte hodnotu registru X a získá nepřímou adresu v zero page (podobně jako u zero page indexed). Na rozdíl od „zpx" tím ale nic nekončí – z paměti na nepřímé adrese (nezapomeňte – v zero page!) jsou načteny dva bajty a z nich je složena cílová adresa.

Zůstaňme u instrukce LDA (12h,X) a řekněme, že v registru X je hodnota 3. Co se stane? Procesor vezme parametr 12h a k němu přičte obsah registru X (12h+03h = 15h), aby získal nepřímou adresu do nulté stránky (0015h). Z adresy 0015h načte nižší bajt výsledné adresy, z adresy 0016h vyšší bajt. Nakonec do registru A uloží obsah paměti na této výsledné adrese.

INDIRECT INDEXED (IZY)

Tento mód je podobný předchozímu, ale liší se v pořadí operací „sečtu" a „přečtu nepřímou adresu". Zapisuje se (nn),Y. Vše osvětlí příklad, řekněme instrukce LDA (12h),Y. Instrukce vezme parametr (12h) a považuje ho za adresu v zero page. Z té načte dva bajty (resp. z adres 0012h a 0013h) a získá tak nepřímou adresu. K ní přičte hodnotu registru Y a výsledek je cílová adresa, ze které se přečte požadovaný bajt do registru A.

UFF.

Já vím. První setkání s adresními módy je pro nezvyklého člověka ukrutná nálož. První půlka ještě jde, ale nepřímé adresování s indexováním je už docela *makačka na představivost*. Nakonec se ale do toho vpravíte, nebojte – i když to není přímočaré jako u 8080.

Bohužel problém je, že tyto adresní módy nejsou úplně ortogonální. Instrukci LDA („ulož do registru A hodnotu") můžete použít s módy IMM (bezprostředně zadaná hodnota), ABS (dvoubajtová adresa), ABX a ABY (dvoubajtová adresa s indexem v X, Y), ZP (jednobajtová adresa do zero page), ZPX (jednobajtová indexovaná adresa do ZP), ZPY… moment, ZPY použít nelze! Proč? No, prostě nelze. Zato můžete využít indexovaně-nepřímé IZX a IZY.

Navíc si všimněte, že je (nn,X), ale není (nn,Y) – na druhou stranu je (nn),Y, ale není (nn),X. Indexové registry nejsou tedy plně ortogonální.

Proto si budeme u každé instrukce ukazovat, jaké adresní módy s nimi lze využít.

NENÍ 65 JAKO 02!

Ještě jednu věc je potřeba předeslat. 6502 má hnedle několik verzí, které se od sebe docela podstatně liší.

- MOS6502 – „původní", originální 6502, použitá např. v Commodore PET nebo KIM-1.
- 6502C – 6502 s vývodem HALT (použitá v počítačích Atari)
- 6510 – 6502 s přidaným portem, využitá v Commodore C64
- 8500 – CMOS verze 6510 (Commodore C64C a C64G)
- 8502 – rychlejší 8500 (až 2 MHz – C128)
- 7501 – HMOS-1 verze 6502 (C16/C116/Plus4)
- 8501 – HMOS-2 verze 6502

Tyto procesory jsou softwarově kompatibilní. Zajímavost je, že procesor, původně vyvinutý jako univerzální společností MOS Technology, se po koupi této společnosti firmou Commodore vyvíjel především tak, aby vyhovoval autorům domácích počítačů od Commodoru.

Na trhu se objevily i další procesory, odvozené z 6502:

- 65C02 – neplést s 6502C! Tato verze přidala některé instrukce
- 65SC02 – zmenšená verze 65C02, která má opět některé instrukce odebrané.
- 65CE02 – rozšířená verze 65C02 (použita v počítači Commodore C65 – tento počítač jste pravděpodobně nikdy neviděli, jejich počet se odhaduje na 50 až 2000 kusů a na eBay se prodával jeden v dubnu 2013 za cenu přesahující 17.000 EUR, tedy cca půlmilion Kč.)
- 65816 – hybridní procesor, který rozšiřuje 65C02 o šestnáctibitové instrukce.

Tyto procesory jsou „rozšířenou 6502" a chovají se jinak, zejména u „nedokumentovaných" instrukcí (některé z nich mají jiný význam, jiné se chovají jako NOP a nezaseknou celý procesor jako u 6502).

V následujícím popisu se budu věnovat té první skupině, tedy „originál 6502".

Instrukce 6502 – přesuny

Instrukční soubor 6502 začneme probírat od instrukcí, které přesouvají data.

Už jsem zmiňoval, že na rozdíl od 8080, kde se liší mnemotechnický zápis instrukcí pro přesuny mezi registry od instrukcí pro přesun z/do paměti (a tam se navíc rozlišuje, zda je adresa uložená v registrech, nebo zapsaná přímo), vystačí si 6502 s několika málo instrukcemi, které přesunou data z/do registru, a to, odkud (nebo kam) se přesouvá, je určeno adresním módem. Základních instrukcí je šest: LDA, LDX, LDY, STA, STX, STY. Trojice LD* (LOAD) přesouvá do registrů (A, X, resp. Y), instrukce ST* (STORE) naopak data z registrů ukládá.

LDA

přesune operand do registru A. Na této instrukci jsme si minule ukazovali adresní módy, její funkce je tedy jasná. Můžeme ji použít s následujícími módy:

- imm: LDA #nn – přesune do registru A přímo hodnotu nn (jednobajtové číslo)
- zp: LDA nn (nebo LDA *nn) – přesune do registru A hodnotu na adrese 00nn
- zpx: LDA nn,X (nebo LDA *nn,X) – dtto jako předchozí, ale k nn je přičten obsah registru X
- abs: LDA nnnn – přesune do registru A hodnotu na adrese nnnn (adresa je 16bitové číslo)
- abx: LDA nnnn,X – dtto jako předchozí, ale k adrese se přičítá obsah registru X
- aby: LDA nnnn,Y – dtto jako předchozí, ale k adrese se přičítá obsah registru Y
- izx: LDA (nn,X) – nepřímé adresování operandu, viz adresní módy
- izy: LDA (nn),Y – nepřímé adresování operandu, viz adresní módy

Všimněte si, že nelze použít mód zpy (tedy zero page s indexací přes Y). Zde je malá ukázka chování těchto instrukcí (pokud jste se ještě s embedovanou verzí assembleru ASM80 nesetkali, tak vězte, že kód kliknutím na COMPILE přeložíte, kliknutím na EMULATOR se otevře emulační okno, kde je možné kód procházet a sledovat, jak se mění obsah paměti či hodnoty v registrech) (V kódu jsem použil instrukci LDX #2, která – analogicky – naplní registr X hodnotou 2)

Jeden zajímavý detail, který může člověka, co přechází ze světa procesorů 8080, zarazit, splést a škaredě překvapit: instrukce LDA (i LDX, LDY a další) mění příznaky Z a N, tedy pokud je přenášená hodnota nulová, nastaví Z, pokud je záporná (=nejvyšší bit je 1), nastaví příznak N. O příznacích budeme teprve mluvit, ale tato zvláštnost, tj. že příznaky ovlivní i některé instrukce pro přenos dat, je natolik důležitá, že ji zmiňuju už teď.

LDX

Analogicky k LDA pracuje tato instrukce s registrem X. Oproti LDA můžete použít jen následující adresní módy:

- imm: LDX #nn – přesune do registru X přímo hodnotu nn (jednobajtové číslo)
- zp: LDX nn (nebo LDX *nn) – přesune do registru X hodnotu na adrese 00nn
- zpy: LDX nn,Y (nebo LDX *nn,Y) – dtto jako předchozí, ale k nn je přičten obsah registru Y. *POZOR – nelze použít mód zpx!*
- abs: LDX nnnn – přesune do registru X hodnotu na adrese nnnn (adresa je 16bitové číslo)
- aby: LDX nnnn,Y – dtto jako předchozí, ale k adrese se přičítá obsah registru Y. *POZOR – nelze použít mód abx!*

LDY

Instrukce je obdobná předchozí, ale u indexovaného přístupu zase nedokáže použít obsah registru Y, tj. naopak umožňuje pouze zpx a abx.

- imm: LDY #nn – přesune do registru X přímo hodnotu nn (jednobajtové číslo)
- zp: LDY nn (nebo LDY *nn) – přesune do registru X hodnotu na adrese 00nn
- zpx: LDY nn,X (nebo LDY *nn,X) – dtto jako předchozí, ale k nn je přičten obsah registru X. *POZOR – nelze použít mód zpy!*
- abs: LDY nnnn – přesune do registru Y hodnotu na adrese nnnn (adresa je 16bitové číslo)
- abx: LDY nnnn,X – dtto jako předchozí, ale k adrese se přičítá obsah registru X. *POZOR – nelze použít mód aby!*

STA, STX, STY

Ukládací instrukce, které přenášejí data v opačném směru než jejich LD* obdoby. Dovolují stejné adresní módy jako odpovídající LD instrukce, s jednou výjimkou: nelze použít mód imm. Dává to smysl, protože nelze uložit hodnotu z registru do konstanty. Instrukce STX a STY navíc neumožňují použít mód „absolutní indexovaný" (abx,aby). Instrukce ST* navíc nemění stav příznaků. Pro úplnost jen doplním seznam možných adresovacích módů:

- STA: zp, zpx, abs, abx, aby, izx, izy
- STX: zp, zpy, abs
- STY: zp, zpx, abs

PŘESUNY

Tím jsme si prošli šest základních instrukcí pro přesun mezi registry a pamětí. Ve světě 8080 by jim zhruba odpovídaly instrukce MVI, MOV r,M, MOV M,r. Pro přesun mezi jednotlivými registry je k dispozici série instrukcí T** (Transfer)

TAX, TXA, TAY, TYA

Čtveřice instrukcí, která kopíruje hodnotu mezi akumulátorem a registry X,Y. Adresní mód je implicitní, tj. instrukce sama ví, odkud se má kam co přesouvat a nepotřebuje žádné další informace. TAX přesouvá hodnotu z registru A do X, TAY analogicky do registru Y, TXA přesouvá z registru X do registru A, TYA z registru Y do registru A. Na přímé přesuny mezi registry X a Y instrukce nejsou. Všechny čtyři navíc, podobně jako LD*, ovlivňují příznaky N a Z.

TSX, TXS

TSX vezme hodnotu registru SP (ukazatel zásobníku) a zkopíruje ji do registru X. Přitom nastaví příznaky N a Z podle přenášené hodnoty. TXS naopak přesune hodnotu z registru X do registru S a příznaky nemění.

ZÁSOBNÍK

Už jsem zmiňoval, že procesor 6502 má ukazatel zásobníku (SP) pouze osmibitový. Adresa v paměti je napevno v první stránce, tedy na adresách 0100h – 01FFh. Zásobník stejně jako u 8080 roste směrem k nižším adresám. Pokud je ukazatel roven 0 a vy uložíte další hodnotu, zapíše se na adresu 0100h a SP se sníží o 1, tedy na FFh. Což znamená, že další ukládání přepíše hodnotu na adrese 01FFh!

PHA, PLA

PUSH A, resp. POP A – PHA uloží hodnotu z registru A do zásobníku, tj. na adresu (0100h+SP) a sníží hodnotu SP o 1. PLA funguje analogicky v opačném směru, tj. zvýší hodnotu SP o 1 a do registru A uloží obsah z adresy (0100h+SP). Navíc nastaví příznaky N a Z.

PHP, PLP

Obdoba předchozích dvou instrukcí, ale nepracuje se s hodnotou registru A, ale s registrem P (příznakový registr). PHP uloží na zásobník obsah příznakového registru, PLP naopak ze zásobníku takovou hodnotu přečte (a, logicky, změní hodnoty všech příznaků).

SHRNUTÍ

Probrali jsme instrukce, které u procesoru 6502 přenášejí data. Na jednu stranu mají poměrně bohaté možnosti, na druhou stranu „ne všechno lze použít se vším" (LDX například dokáže použít absolutní adresu s indexem Y, STX ne), adresní mód zpy

funguje jen u instrukcí LDX, STX (ano, jen u těchto dvou, u žádných jiných se s tímto módem už nesetkáme)… Navíc je potřeba mít na paměti, že instrukce, které přenášejí hodnotu do registru A, X, Y, taky nastavují příznaky N a Z. No a v neposlední řadě dostává ortogonalita na zadek u instrukcí přesunů – hodnotu z registru X do registru Y nepřesunete napřímo, do zásobníku můžete uložit jen registr A (a příznak), pokud chcete uložit X, Y, musíte přes registr A, a pokud chcete nastavit hodnotu ukazatele zásobníku, musíte k tomu zase využít registr X, nemůžete použít A (tady bych si tipnul historický vliv předchůdce 6800, kde SP a X byby oba šestnáctibitové).

6502 – příznaky a instrukce pro práci s nimi

Příznaky a stavový registr procesoru 6502.

Podobnou roli, jakou má v procesoru 8080 registr F, zastává u 6502 registr P.

BIT	7	6	5	4	3	2	1	0
PŘÍZNAK	N	V	1	B	D	I	Z	C

- N (negative) informuje o znaménku výsledku (nebo přenesených dat, vizpopis instrukce LDA). Je-li kladný, je to 0, je-li záporný, je to 1
- V (overflow) značí přetečení čísel se znaménkem (viz dál).
- B (break) je nastaven na 1, pokud bylo přerušení vyvoláno instrukcí BRK
- D (decimal) lze nastavit na 1, pak procesor zpracovává hodnoty v kódu BCD.
- I (interrupt) můžeme nastavit na 1, pokud chceme zakázat přerušení
- Z (zero) je 1, pokud byl výsledek nebo načtený bajt nulový.
- C (carry) se nastavuje na 1, jestliže došlo k přetečení ze 7. bitu

Pojem „přenosu" jsme si vysvětlovali v kapitole o příznacích 8080 (doporučuju přečíst, i když jde o jiný procesor). U 6502 je potřeba věnovat pozornost příznaku V, který nemusí být zcela jasný.

Některé popisy se omezují na málo říkající a nepřesný „přenos ze 6. bitu". Jiné popisy vysvětlují, že se jedná o XOR mezi přenosem ze 6. bitu a ze 7. bitu, což je technicky možná OK, ale neříká, co to vlastně znamená. Pojďme si to vysvětlit názorněji. Příznak V říká, jestli došlo k přetečení čísla se znaménkem. Představme si, že sečteme dvě čísla – 127 a 1 (hexadecimálně 7Fh a 01h). Výsledek je 128 (tedy 80h). Pokud bychom ale používali aritmetiku se znaménkem, tak zjistíme, že 127 + 1 = -128, a to je špatně! Příznak V nás upozorňuje, že došlo k něčemu takovému, tj. že výsledek je mimo rozsah <-128;127>.

Při sčítání FFh a 01h sice dojde k normálnímu přetečení (C), ale z hlediska čísel se znaménkem se vlastně sčítalo „-1 + 1" a výsledek je 0, bez přetečení. V tedy bude 0.

Při sčítání 80h a FFh bude výsledek 7Fh. U čísel bez znaménka došlo k přetečení (128 + 255), u čísel se znaménkem (-128 + -1) taky. Budou tedy nastaveny příznaky C i V.

Demonstrační kód si můžete vyzkoušet opět v emulátoru. Instrukce CLC slouží k nulování příznaku C a ADC sčítá dvě čísla (6502 má pouze instrukci sčítání s příznakem C, proto je ho potřeba nejprve nulovat, ale k tomu se ještě dostaneme). Můžete si vyzkoušet chování; sledujte hlavně stav bitů V a C.

(Další podrobnosti o výpočtu příznaku V: The 6502 overflow flag explained mathematically)

Další bit v příznakovém registru, který zaslouží vysvětlení, je bit I. Pokud je tento bit roven 1, je zakázáno („zamaskováno") přerušení a procesor nereaguje na signál, přivedený na přerušovací vstup IRQ (6502 má dva druhy přerušení, maskovatelné IRQ a nemaskovatelné NMI, ale k nim se ještě dostaneme). Příznak může na hodnotu 1 nastavit programátor instrukcí SEI, popřípadě procesor poté, co přišel požadavek na přerušení – tím se zabrání, aby bylo vyvoláno přerušení dřív, než skončila obsluha předchozího.

Příznak B označuje, že obsluha přerušení byla vyvolána instrukcí BRK, nikoli vnějším signálem IRQ. Instrukce návratu z obsluhy přerušení jej opět nuluje.

Příznak D může programátor nastavit na 1 a tím vynutit, aby procesor pracoval v režimu BCD – tedy jako by po každé operaci sčítání a odčítání prováděl dekadickou korekci.

INSTRUKCE PRO PRÁCI S PŘÍZNAKOVÝM REGISTREM

Příznakové bity nastavují různé instrukce v rámci své normální činnosti (většinou aritmetické, logické nebo instrukce přenosu dat), ale existuje i sada instrukcí pro nastavení či nulování konkrétních bitů.

CLC, SEC

Instrukce nuluje (CLC – CLear Carry) nebo nastavuje (SEC – SEt Carry) příznak C

CLD, SED

Instrukce nuluje (CLD) nebo nastavuje (SED) příznak D

CLI, SEI

Instrukce nuluje (CLI) nebo nastavuje (SEI) příznak I

CLV

Instrukce nuluje příznak V. (Vidíte správně, žádná instrukce SEV není.)

Tyto instrukce nemají žádný parametr (je tedy použit „implicitní mód").

6502 – přerušovací systém

Nejen o přerušení, ale také o tom, co se děje, když zapnete napájení.

Napětí je připojeno, hodinový takt běží, procesor 6502 začíná pracovat. Co udělá ze všeho nejdřív? *Správnou odpověď do vzkazů, pokud neuhodnete, musíte si dát tuto hádanku na svůj Facebook!*

Promiňte, samozřejmě to není hádanka a nic si na Facebook dávat nemusíte. Řekneme si to hned teď. Ale začneme přerušením. Procesor 6502 má, stejně jako jiné procesory, k dispozici přerušovací systém.Princip přerušení jsme si už popisovali u procesoru 8080: v situaci, kdy je potřeba zareagovat (např. přišel kompletní načtený znak z terminálu) si vyžádají okolní obvody pozornost procesoru přerušovacím signálem. Procesor k takovému účelu má extra přerušovací vstup (někdy víc), a zareaguje tak, že uloží svůj stav na zásobník a provede určitou instrukci, která většinou způsobí skok do podprogramu.

Přerušení u 8080 je maskovatelné, to znamená, že pomocí instrukce může programátor zakázat, aby procesor na přerušení reagoval (obvykle v časově kritických místech).

U 6502 jsou dva přerušovací vstupy. Jeden z nich je maskovatelný (IRQ) – signál na tomto vstupu vyvolá přerušení pouze v případě, že není nastaven příznak I. Druhý přerušovací vstup je nemaskovatelný (NMI, Non-Maskable Interrupt). Signál na tomto vstupu vyvolá přerušení vždy.

Co se stane, když systém vyvolá přerušení? Procesor v tu chvíli uloží na zásobník hodnotu registru PC (nejprve vyšší, potom nižší bajt), pak uloží rovněž na zásobník hodnotu příznakového registru P a pak skočí na adresu obsluhy přerušení.

A tu zjistí kde přesně? Správná otázka. Pokud šlo o maskovatelné přerušení IRQ, tak si ji přečte na adresách FFFEh a FFFFh, tedy na posledních dvou adresách adresního prostoru. Pokud vás trápí otázka „a jak se tam ta adresa dostane?", odpověď zní:

To záleží na návrháři systému. Pokud je tam pevná paměť (ROM/PROM/EPROM/atd.), je v ní adresa obsluhy přerušení („přerušovací vektor") uložená napevno. Když je tam RAM, zapsal si ji tam programátor.
Pokud šlo o nemaskovatelné přerušení NMI, přečte si adresu obslužné rutiny na adresách FFFAh a FFFBh.
Zajímá vás, co je mezi tím? Na FFFAh a FFFBh je adresa obsluhy NMI, adresa obsluhy IRQ je na FFFEh a FFFFh, zbývá volný prostor FFFCh a FFFDh… Tam je adresa RESETu.
Ano, čtete dobře. Po zapnutí napájení nebo po přivedení signálu RESET se procesor nenastavuje do nějakého definovaného stavu, on prostě jen skočí na adresu, která je zapsaná v buňkách FFFCh a FFFDh. Jediný rozdíl proti přerušení (viz výše) je v tom, že RESET neukládá PC a P na zásobník.
Vzhledem k tomu je potřeba, aby na těchto adresách byla při startu systému smysluplná adresa. Nemůžeme spoléhat na to, že ji tam zapíše programátor, takže je to potřeba buď vyřešit tím, že na konci paměťového rozsahu je paměť ROM, nebo nějakým obvodovým hackem, který po startu „podvrhne" procesoru tu správnou adresu.

INSTRUKCE RTI

Instrukce RTI – Return from Interrupt doplňuje přerušení, jak jsme si popsali výše. Provádí přesně opačné kroky, tj. ze zásobníku načte obsah registru P, pak nižší a vyšší bajt registru PC. Postará se tedy o správný návrat z rutiny přerušení.
Jednoduchý příklad v assembleru: program uloží do registru A hodnotu 3 a tu pak zapíše do nulté stránky na adresu 0. Zápis pak probíhá stále dokola – můžete si ověřit pomocí krokování. Pokud kliknete na IRQ, procesor si odskočí do obsluhy přerušení, která změní obsah registru A. Po návratu do nekonečné zapisovací smyčky se už tedy bude zapisovat jiná hodnota.

Všimněte si, že tentokrát program začíná na adrese 0200h a od adresy FFFCh jsou zadány hodnoty startovací adresy a přerušovací rutiny.
Při krokování si všimněte, že při provádění obsluhy přerušení je nastaven příznak I.

INSTRUKCE BRK

Co se stane, když procesor provede instrukci BRK? Uloží na zásobník hodnotu registru PC (nejprve vyšší, potom nižší bajt), pak uloží na zásobník hodnotu příznakového registru P a pak skočí na adresu obsluhy přerušení IRQ, kterou si přečte na adresách FFFEh a FFFFh.
Možná vám to připadá povědomé… Ano, přesně totéž se děje při maskovatelném přerušení IRQ! Není to náhoda.
Ve skutečnosti je procesor 6502 zapojen tak, že přerušovací požadavek po kontrole příznaku I uloží do registru, kam si načítá kód další instrukce, operační kód instrukce BRK (který je, čistě pro zajímavost, roven 00h). Takže se opravdu provádí to samé – s jedinou výjimkou: instrukce BRK před skokem na obsluhu ještě nastaví příznakový bit B. Podle něj lze poznat, jestli obsluhu přerušení vyvolal vnější systém (B=0) nebo instrukce BRK (B=1).
Pro zájemce jen dodám, že stejně funguje i obsluha NMI, která rovněž podvrhne BRK, ale změní i adresy vektoru, a v zásadě i signál RESET, který ale místo „ukládání do paměti" při práci se zásobníkem aktivuje signál „čtení z paměti" – více o těchto vnitřních zajímavostech naleznete v článku Internals of BRK/IRQ/NMI/RESET on a MOS 6502. Z tohoto článku ocituju i souhrnnou tabulku, co se děje v procesoru 6502 při přerušení a instrukci BRK:

PŘÍČINA	VEKTOR	UKLÁDÁ PC A P NA ZÁSOBNÍK?	NASTAVUJE PŘÍZNAK B?
signál NMI	$FFFA/$FFFB	ano	ne
signál RESET	$FFFC/$FFFD	ne	ne
signál IRQ	$FFFE/$FFFF	ano	ne
instrukce BRK	$FFFE/$FFFF	ano	ano

Instrukce 6502 – skoky a podprogramy

V minulé lekci jsem použil v kódu instrukci skoku a popisoval jsem návrat z přerušení. Pojďme si tedy doplnit sérii a probrat zbývající instrukce skoků.

NEPODMÍNĚNÝ SKOK – JMP

Instrukce nepodmíněného skoku dělá přesně to, co u jiných procesorů – tedy to, co se označuje známým „GOTO". Skočí se na jinou adresu a pokračuje se odtamtud. Pokud pracujete s assemblerem, nemáte žádné programátorské struktury, žádné smyčky ani bloky IF-ELSE-ENDIF, všechno musíte řešit pomocí skoků a podmíněných skoků.
Instrukce JMP používá dva adresní módy – buď absolutní adresování, nebo nepřímé. Absolutní znamená, že se skáče přímo na zadanou adresu:
Nepřímé adresování (viz díl o adresních módech) pracuje tak, že ze zadané adresy (a z adresy o 1 vyšší) se načtou dva bajty, které dohromady dají dvoubajtovou *efektivní adresu* a skáče se na ni.

Všimněte si, že v tomto druhém případě neskáče JMP přímo na adresu LOOP, ale na (ind). ind je návěští, na kterém jsou uložené dva bajty (viz výpis přeloženého programu).

PODMÍNĚNÉ SKOKY – BXX

Podmíněných skoků je osm pro osm různých podmínek – podle čtyř příznakových bitů, vždy 0 nebo 1. Zde jsou v přehledné tabulce:

PŘÍZNAK	STAV	
	0	1
N	BPL	BMI
V	BVC	BVS

PŘÍZNAK	STAV	
	0	1
C	BCC	BCS
Z	BNE	BEQ

Mnemotechnika těchto názvů je prostá. Instrukce jsou skoky (Branch), z toho je písmeno B. Další dvě písmena jsou název podmínky – u příznaku N (Negative), který říká, jestli je číslo kladné nebo záporné, je to BPL (Branch if PLus) a BMI (Branch if MInus). U příznaku Z, který udává, jestli je číslo nula, to není Zero-Nonzero, ale využívá se toho, že tento skok bývá často prováděn po testu na rovnost (který interně probíhá jako odčítání). Při nerovnosti, nenulovém výsledku (Z=0) se skáče instrukcí BNE (Branch if Not Equal), analogicky při rovnosti, a tedy Z=1, se skáče instrukcí BEQ (Branch if EQual).

U příznaků C a V, které nemají takhle jednoznačné „vysvětlení", se používá mnemotechnika „Branch if V is Clear" (BVC), „Branch if C is Set" (BCS) apod.

Adresní mód těchto instrukcí je vždy relativní. To znamená, že se skáče v rozmezí -128..+127. Využívá se toho, že podobné podmíněné skoky jsou většinou součástí krátkých smyček. Pokud tomu tak není, musíte použít „náhražkovou" konstrukci:

```
1    BCC NekamDaleko ; nelze, pokud je adresa vzdálená víc než 128 pozic
2                    ; Místo toho je třeba použít:
3
4    BCS Skip        ; Obrácená podmínka, která přeskočí následující instrukci
5    JMP NekamDaleko ; tady se provede samotný skok
6    Skip: ....      ; a tady se pokračuje
```

Překladač naštěstí za vás spočítá správnou hodnotu odskoku z aktuální adresy návěští a cílové adresy. Hodnota 0 znamená následující adresu (tj. žádný efekt), hodnota 0FEh skáče o dvě místa zpátky, tj. na tu samou adresu, kde je instrukce (je to hodnota -2, a Bxx jsou dvoubajtové).

BRK, RTI

Tyto instrukce jsme probrali v minulém díle – slouží pro vyvolání přerušení a pro návrat z obslužné rutiny přerušení.

PODPROGRAMY – JSR, RTS

To, k čemu u 8080 sloužily instrukce CALL a RET, zajišťují u 6502 instrukce JSR a RTS (Jump to Subroutine / Return from Subroutine). JSR používá pouze absolutní adresní mód, tj. za instrukcí jsou nižší a vyšší bajty cílové adresy. JSR uloží na zásobník vyšší a nižší bajt návratové adresy, a pak do PC nahraje přečtenou adresu. Čímž se vlastně provede skok na nějakou adresu (jako u JMP), s tím rozdílem, že na zásobníku je adresa, kam se má program vrátit (ve skutečnosti je o 1 menší, s čímž počítá instrukce RTS).

K návratu slouží instrukce RTS. Ta přečte ze zásobníku dva bajty, z nich složí adresu, přičte 1 (viz výše) a na ni skočí. Pokud podprogram zanechal zásobník v takovém stavu, v jakém ho našel, tak se skočí na instrukci, následující za příslušnou instrukcí JSR.

Rozdíl mezi RTS a RTI je v tom, že RTI načítá ze zásobníku i uloženou hodnotu příznakového registru P a k návratové adrese nepřičítá 1 (instrukce BRK i přerušení ukládají pravou návratovou adresu).

Úhrnem lze o instrukční sadě procesoru 6502 v souvislosti se skoky říct, že je hodně omezená. Podmíněné skoky pouze relativní, skoky do podprogramu a návraty pouze nepodmíněné a s absolutní adresou, jen nepodmíněný skok lze adresovat i nepřímou adresou (ale nelze relativně).

Instrukce 6502 – aritmetika

V tomto díle konečně donutíme procesor 6502 něco spočítat.

Když jsem v minulých dílech naznačoval, že to s ortogonalitou instrukční sady procesoru 6502 není, ani přes velké množství adresních módů, moc slavné, tak věřte, že jsem si to nejhorší šetřil až na závěr.

INC, DEC

Nejjednodušší aritmetické instrukce jsou inkrement a dekrement, tedy přičtení jedničky a odečtení jedničky. 6502 má k tomu účelu instrukce INC a DEC. Tyto instrukce zvýší (INC) nebo sníží (DEC) obsah paměťové buňky o 1. Můžete je použít s následujícími adresními módy:

- abs: INC 1234h – zvýší obsah buňky na adrese 1234h o 1
- zp: INC 12h – zvýší obsah buňky na adrese 0012h o 1
- abx: INC 1234h,X – zvýší obsah buňky na adrese (1234h + X) o 1
- zpx: INC 12h,X – zvýší obsah buňky v nulté stránce paměti na adrese (12h+X) o 1 (nezapomeňte, že nultá stránka má vždycky horní byte adresy rovný 0, pokud tedy bude v X hodnota FFh, nebude se pracovat s adresou 0111h, ale 0011h!)

Totéž pro instrukci DEC.

INX, INY, DEX, DEY

Obdoba instrukcí INC, DEC, ale místo obsahu paměti se pracuje s registry X (INX, DEX) a Y (INY, DEY).

Instrukce inkrementu a dekrementu nastavují podle výsledku operace příznaky N a Z.

Možná jste si všimli, že jsem v seznamu neuvedl instrukce, které inkrementují/dekrementují obsah akumulátoru A. Neuvedl jsem je, protože je procesor 6502 nemá.

ADC, SBC

Sčítání a odčítání 6502 samozřejmě obsahuje. ADC (Addition with Carry) přičte parametr a hodnotu příznaku C k registru A a výsledek ponechá v registru A. SBC (Subtraction with Carry) odečte od obsahu registru A parametr a negaci příznaku C a výsledek uloží do registru A.

Co z toho vyplývá? Zaprvé: 6502 vždycky uvažuje stav příznaku C (přenos). Neexistuje instrukce pro sčítání nebo odčítání, která by jeho stav ignorovala. Pokud chceme „jen" sčítat dvě čísla, je potřeba předtím nastavit C na nulu instrukcí CLC, jinak bude výsledek o 1 vyšší. Pokud výsledek sčítání přeteče 255, bude C=1, jinak zůstane nulový.

U instrukce pro odčítání platí přesný opak – pokud chceme zanedbat přenos, musíme příznak nastavit na 1 (instrukcí SEC).

Pokud výsledek při odčítání podteče nulu (výsledkem je záporné číslo), bude příznak C roven 0, jinak 1.

A tak se může stát, pokud neošetříte příznaky správně, že dvojice instrukcí ADC #1, SBC #1 ve skutečnosti dělají věci nečekané.

Viz následující kód – sledujte instrukce a výsledek v registru A:

Adresní módy těchto instrukcí jsou stejné jako např. u instrukce LDA, tedy:

- imm – přímý operand: ADC #1 přičte 1
- abs, zp – přímo zadaná adresa, buď plná, nebo v zero page
- abx,aby – absolutní adresa, zvýšená o obsah registru X či Y
- zpx – adresa v zero page, indexovaná přes registr X
- izx, izy – nepřímo adresovaný operand (viz adresní módy)

POROVNÁNÍ – CMP

Instrukce CMP porovná hodnotu v registru A s operandem. Vnitřně funguje tak, že od hodnoty v registru A odečte hodnotu operandu, podle výsledku nastaví příznaky N, Z a C a výsledek zahodí.
Mohou nastat tři situace, které si ukážeme v následující tabulce:

SITUACE	N	Z	C
A = operand	0	1	1
A > operand	0	0	1
A < operand	1	0	0

(Hodnoty uvažujeme jako čísla bez znaménka)
Instrukce CMP nabízí stejné adresační možnosti jako instrukce ADC či SBC.

CPX, CPY

Podobně jako existují obdoby instrukcí INC a DEC pro práci s registry X a Y, tak i CMP má obdoby CPX a CPY. Liší se od CMP tím, že neporovnávají operand s hodnotou registru A, ale s registrem X, resp. Y. CPX a CPY mají jen tři adresní módy: přímý operand (imm), absolutní adresa (abs) nebo adresa v nulté stránce (zp).

Instrukce 6502 – logické a bitové operace

Blížíme se ke konci, zbývá doprobrat už jen pár instrukcí, konkrétně logické operace a manipulace s bity.

AND, ORA, EOR

Trojice instrukcí pro základní bitové operace – and, or, xor (exclusive or) se u procesoru 6502 jmenují AND, ORA a EOR. Provedou danou logickou operaci s obsahem registru A, výsledek uloží do A a nastaví příznaky N a Z.
Všechny tři instrukce mají poměrně bohaté možnosti adresování:

- imm – přímý operand: AND #1 provede operaci A = A & 01
- abs, zp – přímo zadaná adresa, buď plná, nebo v zero page
- abx,aby – absolutní adresa, zvýšená o obsah registru X či Y
- zpx – adresa v zero page, indexovaná přes registr X
- izx, izy – nepřímo adresovaný operand (viz adresní módy)

ROTACE – ROL, ROR

Instrukce ROL a ROR rotují bitově obsah registru A nebo paměti (ROL doleva, ROR doprava). Při rotaci se rotuje přes příznak C. ROL posune bity o 1 doleva. To znamená, že bit 0 se přesune na pozici 1, bit 1 na pozici 2, bit 2 na pozici 3 a tak dál, a bit 7, který nám vypadne zleva ven, je zapsán do příznaku C, a původní hodnota z C je přesunuta do pozice 0 v registru A. ROR funguje stejně, jen obráceným směrem. Graficky to vypadá nějak takto:

OPERACE	POZICE V OPERANDU								PŘÍZNAK
	7	6	5	4	3	2	1	0	C
VÝCHOZÍ STAV	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	C
ROL	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	C	Bit 7
ROR	C	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Instrukce ROL a ROR nabízejí opět několik adresních módů. Bez operandu pracuje s obsahem registru A. Pokud chcete pracovat s obsahem paměti, můžete buňku adresovat buď absolutně (abs, 2 bajty adresa), v nulové stránce (zp, 1 bajt), nebo indexovaně (abx nebo zpx).

POSUNY – ASL A LSR

Posuny se od rotací liší v tom, že „vypadnuvší" bit je přesunut do příznaku C, ale původní hodnota tohoto příznaku je zahozena a místo ní vstoupí zpět hodnota 0. ASL posouvá doleva, zprava doplní 0, LSR posouvá doprava, zleva doplní 0. Adresní módy jsou stejné jako u rotací – bez operandů se pracuje s registrem A, pokud chcete pracovat s pamětí, můžete použít abs, zp, abx nebo zpx.
Matematicky odpovídá posun doleva vynásobení hodnoty dvojkou, posun doprava pak celočíselnému dělení 2 (zbytek je v příznaku C). Grafické znázornění zde:

OPERACE	POZICE V OPERANDU								PŘÍZNAK
	7	6	5	4	3	2	1	0	C
VÝCHOZÍ STAV	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	C

OPERACE	POZICE V OPERANDU								PŘÍZNAK
	7	6	5	4	3	2	1	0	C
ASL	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	0	Bit 7
LSR	0	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

INSTRUKCE BIT

Instrukce BIT provede logický součin (AND) obsahu registru A a operandu, který je adresován buď absolutní adresou (abs), nebo nulovou stránkou (zp). Výsledek je zahozen, ale předtím je podle něj nastaven stav příznaku Z. Je-li tedy výsledek 0, je Z=1.
Instrukce BIT ještě nastaví příznaky N a V – zkopíruje do nich šestý a sedmý bit operandu (bit 7 do příznaku N, bit 6 do příznaku V).

6502 – sčítání a násobení

Několik užitečných algoritmů pro vaše procesory 6502.

16BITOVÝ SOUČET

U procesoru 8080 není sčítání 16bitových čísel problém – procesor má dostatek registrů a má k tomu i speciální instrukci DAD. U 6502 nemáme ani instrukci, ani registry. Proto se musí sčítání dvoubajtových čísel řešit algoritmem. Naštěstí není příliš složitý, využívá jen instrukci ADC.
Všimněte si, že obě čísla (P1 a P2) jsou uloženy v zero page (na adresách 60h a 62h) a do zero page se ukládá i součet (R, 64h). Nejprve se nuluje příznak přenosu, pak se sečtou dva nižší bajty a poté dva vyšší. Případný přenos z nižšího do vyššího bajtu zařídí příznak C.
Odčítání je pak naprosto analogické, jen na začátku je potřeba příznak C nastavit na 1, nikoli nulovat.

NÁSOBENÍ

Vzpomínáte, jak jsme u procesoru 8080 násobili dvě osmibitová čísla? Můžeme podobný postup použít i u 6502? Tak v zásadě ano, ale se stejnými výhradami jako u sčítání: Nejsou registry.
Algoritmus rovněž využívá rotace do vyššího bajtu, jako u 8080, ale protože 6502 jaksi nemá nic jako „vyšší bajt", jedná se zas o místo v paměti. A protože k paměti se nepřistupuje žádnou 16bitovou operací, není nezbytně nutné, aby „vyšší bajt" byl hned za „nižším bajtem". A protože není 16bitové sčítání, viz výše, je nahrazeno rovněž algoritmem.
Výsledek násobení je v registrech X (vyšší bajt) a A (nižší bajt).
Všimněte si, že násobení dvojkou je zařízeno jako „šestnáctibitový shift" – pomocí ASL a ROL.

6502 – Ahoj, světe…

Nastala chvíle, kdy křemík ožije a provede, co po něm chceme. Tedy aspoň ten virtuální…
U 8080 to bylo prosté – nabídnul jsem PMD-85, k němu mám i emulátor, takže nebyl problém. U 6502 by šlo využít třeba Atari nebo Commodore C64, problém je, že k nim emulátor nemám. Zato mám emulátor jednodeskového počítačeSBC6502 od Granta Searla. Jeho počítač je extrémně jednoduchý, tvoří ho, včetně procesoru, sedm integrovaných obvodů a můžete si ho poskládat na nepájivém kontaktním poli, když na to přijde.
Grantův počítač obsahuje procesor 6502, u něho je generátor hodin, logika pro dekódování adres a sériový komunikační obvod ACIA 6850. Od adresy 0000h do adresy 7FFFh je 32 kB RAM. ROM má 16 kB a je na adresách C000h až FFFFh. V prostoru A000h – BFFFh je namapován sériový komunikační obvod, k němuž se ještě dostanu.
Grant do svého počítače použil Microsoft BASIC 6502 (od adresy C000h) a k němu přidal rutiny pro komunikaci přes sériový port. Počítač je tak možné připojit k sériovému portu u PC (třeba i pomocí převodníku RS232-USB) a komunikovat s ním pomocí terminálu (Hyperterminal, PuTTY atd.) Já jsem do emulátoru napsal i emulaci terminálu, takže nemusíte nic s ničím spojovat a všechno funguje v prohlížeči. Můžete si vyzkoušet emulátor SBC6502 se zabudovaným BASICem.
Ale co když nebude BASIC? Jak oživit takový holý počítač? Začneme výpisem obligátního HELLO WORLD. K tomu ale budeme muset nějak ovládat ten komunikační obvod… Naštěstí to není složité, a tak si aspoň ukážeme, jak v takových chvílích postupovat.
Co máme? Máme typ obvodu – 6850. ACIA je zkratka (Asynchronous Communications Interface Adapter), která říká, že se jedná o komunikační rozhraní. Výrobce je Motorola (a další), obvod je z rodiny podpůrných obvodů procesoru 6800. Chvilku poGooglíme a najdeme takzvaný datasheet. Datasheet je materiál, dodávaný výrobcem čipu, kde jsou popsány veškeré důležité skutečnosti – od napájecího napětí přes popis vývodů a pouzdra až k programátorskému rozhraní. naštěstí ho nemusíte číst, přečetl jsem ho za vás, a tady jsou shrnuty základní informace – vynechám to, co je pro nás nepodstatné (řízení spojení pomocí signálů RTS, CTS apod. nebo přerušení)

ACIA 6850

Obvod 6850 je sériový komunikační obvod. (Datasheet, programátorský popis) Je připojen k procesoru a jeho hlavním úkolem je převést zaslaný bajt na sériový signál standardu RS-232 (tj. správně odvysílat start bit, datové bity, případně paritní bit, a nakonec stop bit) a opačně, tj. načíst správně časovaný sériový signál a připravit ho k předání procesoru.
Asynchronní v popisu znamená, že spolu s daty není přenášen hodinový signál ani není činnost přesně časována – když přijde bajt, je vyslán, když přijdou vstupní sériová data, jsou načtena. Synchronizace, tj. to, že bude načteno opravdu to, co načteno být má, zajišťují právě start a stop bity – podle jejich správného průběhu pozná obvod, že jsou data v pořádku.
Vysílání dat po výstupu TXDATA (Tx = transmit) a příjem na vstupu RXDATA (Rx = receive) je časován pomocí systémových hodin. U Grantova počítače je systémový kmitočet roven 1,8432 MHz. Tento kmitočet je v ACIA vnitřně dělen, dělitel je programově nastavitelný. Grant používá dělení šestnácti, což znamená, že komunikační rychlost je 1843200 / 16 = 115200 bitů za sekundu (baud). Pokud jste někdy používali sériové rozhraní, víte, že 115200 je jedna z používaných komunikačních rychlostí.
Obvod 6850 s procesorem komunikuje pomocí osmibitové datové sběrnice (D0-D7), několika CS vstupů (CS = Chip Select), které určují, kdy se s obvodem komunikuje (CS0 = 1, CS1 = 1, CS2 = 0 – ve všech ostatních případech je datová sběrnice odpojena), vstupu E (Enable, obvod komunikuje jen pokud je E=1), vstupu R/W, který udává, zda se z obvodu čte (1) nebo se do něj zapisuje (0) a vstupu RS, který udává, jestli se čtou/zapisují data (1), nebo řídicí hodnoty (0).
Přístup k obvodu má pak určitá pravidla toho, jak mají a mohou po sobě jednotlivé signály následovat, za jak dlouho po přivedení signálu jsou připravená data apod. Z hlediska programátora jsou tyto informace většinou (ne vždy!) irelevantní, protože o časování signálů se stará vnější logika. *Někdy se ale nepostará, a pak je potřeba, aby např. programátor mezi posláním dvou hodnot nějakou chvíli počkal, ale to návrhář systému většinou zmíní.*

Z hlediska programátora se tedy obvod 6502 jeví jako dva registry (vybrané pomocí vstupu RS), z nichž jeden je datový, druhý „systémový". Funkci osvětlí tabulka:

VSTUPY		REGISTRY	
RS	R/W	TYP REGISTRU	FUNKCE
0	0	Zápis	Řídicí registr (Control Register, CR)
0	1	Čtení	Stavový registr (Status Register, SR)
1	0	Zápis	Data k vyslání (Transmit Data Register, TDR)
1	1	Čtení	Přijatá data (Receive Data Register, RDR)

Grant Searle svůj počítač zapojil tak, že na adrese A000h je řídicí / stavový registr (CR/SR, podle toho, jestli se čte nebo zapisuje), na adrese A001h jsou datové registry. Vzhledem k tomu, že použil jen jednoduchý dekodér, tak se CR/SR objevuje i na adresách A002h, A004h, … až do BFFEh, datový pak na adresách o 1 vyšší (A001h, A003h, … BFFFh). Podobné „nedokonalé" dekódování je poměrně časté – šetří to totiž „drahé" součástky a nezbytnou logiku.

Všimněte si, že do řídicího registru je možné pouze zapisovat, nelze z něj číst, naopak stavový registr lze pouze číst, nelze do něj zapisovat. Není to totiž potřeba. Totéž s datovými registry – při čtení se čte to, co obvod přijal (a nezajímá nás to, co jsme odeslali). Při zápisu je jasné, že chceme data vyslat, nedávalo by smysl zapisovat do přijatých dat.

ŘÍDICÍ REGISTR CR

Osmibitový registr CR řídí čtyři funkce obvodu:

- Bity 0 a 1 nastavují dělicí poměr hodin, viz výše.

CR1	CR0	DĚLITEL
0	0	1
0	1	16
1	0	64
1	1	RESET

- Poslední kombinace nenastavuje dělitele, ale celý obvod resetuje do výchozího nastavení, tj. vyprázdní registry a nuluje příznaky. Grant používá kombinaci 01, tj. dělení 16. Kdyby použil kombinaci 10, tj. dělení 64, komunikoval by obvod rychlostí 28800 Bd.
- Bity 2, 3 a 4 nastavují délku vysílaných a přijímaných dat (sedmibitové nebo osmibitové), zda se pracuje s paritou a kolik je stop bitů. Tyto informace najdete např. i v nastavení Hyperterminálu, když půjdete hledat detaily připojení. Pro naše účely použijeme kombinaci 101, tj. osmibitový přenos, bez parity, 1 stop bit.
- Bity 5 a 6 určují, jestli vysílač po odvysílaném bajtu bude žádat o přerušení a v jakém stavu bude výstup RTS
- Bit 7 určuje, jestli přijímač bude vyvolávat přerušení v případě chyby.

Pro bližší popis odkazuju zájemce opět k datasheetu, my použijeme hodnotu 15h, tj. osmibitový přenos, bez parity, přerušení zakázané a přenosová rychlost 115200 Baud (Baud je jednotka „bitů za sekundu" – včetně start a stop bitů).

STAVOVÝ REGISTR SR

Svět není dokonalý, život není fér a asynchronní sériové přenosy nejsou nijak sladěné s biorytmy našeho procesoru. Pokud pošlu bajt do 6850, začne ho obvod vysílat. Ovšem předtím je dobré podívat se, jestli už dokončil vysílání toho předchozího. Při příjmu je zase dobré se podívat, jestli nějaký bajt už načetl, a pokud ho načetl, tak ho zpracovat, aby se uvolnilo místo pro další bajt. Popřípadě získat informaci o tom, jestli nedošlo k chybě. Tyhle informace jako když najdete ve stavovém registru SR.

Pokud načtete bajt ze SR, tak vás jednotlivé bity informují o následujícím:

- Bit 0 – Receiver Data Register Full (RDRF). Pokud je nastaven na 1, znamená to, že obvod načetl bajt po sériové lince do přijímače a bylo by záhodno ho zpracovat. Jakmile procesor přečte stav datového registru, je bit RDRF nastaven na 0. Nula znamená, že žádný nový bajt nepřišel.
- Bit 1 – Transmitter Data Register Empty (TDRE). Jakmile zapíšete do datového registru bajt, nastaví se TDRE na 0 a obvod začne bajt vysílat po sériové lince. Jakmile ho vyšle, nastaví tento bit na 1. Programátor by se měl před tím, než nějaký bajt pošle, zkontrolovat, že může – tedy že bit TDRE = 1.
- Bity 2, 3 pracují s řídicími signály CTS, DCD, a já je tady s klidným svědomím opomenu.
- Bit 4 – Framing Error (FE) znamená, že přijatá data byla špatně časována, např. že nepřišel požadovaný STOP bit.
- Bit 5 – Receiver Overrun (OVRN). Overrun, neboli hezky česky *přeběh*, je stav, kdy přijímač přijal bajt, ale procesor ještě nezpracoval předchozí přijatý. Ten nově přijatý je tedy zahozený (protože jej není kam dát, žádný vnitřní buffer není) a nastaví se OVRN na 1, aby bylo jasné, že došlo k chybě.
- Bit 6 – Parity Error (PE). Pokud využíváme přenos s paritou, zkontroluje 6850 paritní bit. Pokud byl chybný, nastaví příznak PE.
- Bit 7 – Interrupt Request (IRQ) říká, že obvod požádal o přerušení z nějakého závažného důvodu. Buď byl přijat bajt a je povolené přerušení při přijetí dat, nebo byl odeslán bajt a je povoleno přerušení při odeslání, nebo vypadla nosná (DCD).

JAK PRACOVAT S 6850?

Na začátku musíme nastavit řídicí registr tak, jak potřebujeme. Už jsme si řekli, že to bude hodnota 15h. Tím je obvod nastaven a připraven k přijímání a vysílání dat.

```
1   ACIA        = $A000
2   ACIACONTROL = ACIA+0
3   ACIASTATUS  = ACIA+0
```

```
4        ACIADATA     =  ACIA+1
5
6        LDA     #$15    ; 115200 Bd, 8 bit, no parity, 1 stop bit, no IRQ
7        STA     ACIAControl
```

Na začátku jsou pojmenované adresy ACIA (bázová adresa obvodu 6850 v Grantově počítači), ACIAControl, ACIAStatus a ACIAData. Vyhneme se tak programátorskému moru, „magickým konstantám". (Syntaxe „návěští = hodnota" je ekvivalentní zápisu „návěští EQU hodnota", který jsme používali u 8080. Funkčně je to totéž, jen u asesemblerů 8080 je zvykem zápis s EQU, u 6502 zápis s rovnítkem.)
Do registru A uložím požadované řídicí slovo (15h) a instrukcí STA ho zapíšu na adresu ACIAControl (tj. A000h). Vzpomeňte si, že procesor 6502 nerozlišuje mezi pamětí a porty, obojí má v jednom adresním prostoru, takže používáme normální ibnstrukci pro zápis do paměti. Vnější logika Grantova počítače se postará o to, že data neskončí v paměti, ale tam, kde mají, tj. v obvodu 6850.
Co dál? Obvod je nastaven, teď je zapotřebí vypsat ono obligátní HELLO WORLD. Někde v paměti tedy bude tenhle řetězec a my ho budeme bajt po bajtu procházet a vysílat na sériový výstup.
Když se řekne „vysílat", tak si na to uděláme podprogram. Bude se jmenovat třeba SEROUT (jako že SERial OUTput) a jeho funkce bude, že vyšle hodnotu v registru A. Předtím si ale zkontroluje, jestli je vysílač volný. Musí si tedy načíst hodnotu stavového registru SR a zkontrolovat bit 1. Pokud je nulový, musí počkat, až bude 1.

```
1  SEROUT:   PHA
2  SO_WAIT:  LDA     ACIAStatus
3            AND     #2
4            BEQ     SO_WAIT
5            PLA
6            STA     ACIAData
7            RTS
```

Na začátku si uložím obsah registru A. Mám v něm ten bajt, co chci vyslat, ale budu ten registr potřebovat, protože si do něj načtu hodnotu stavového registru. Takže si jeho hodnotu uložím na zásobník.
Na dalším řádku načtu hodnotu stavového registru do registru A (LDA). Pak provedu logický součin (AND) s hodnotou 2. Hodnota 2 totiž binárně vypadá takto: 00000010 – jsou to tedy samé nuly, jen na pozici bitu 1, který potřebuju testovat, je jednička. Výsledkem logického součinu bude buď hodnota 2, pokud je bit 1 nastaven, nebo 0, pokud je nulový.
Připomeňme si: pokud je bit 1 stavového registru nulový, znamená to, že obvod 6850 ještě vysílá předchozí data a my musíme počkat, dokud to nedokončí. Tedy pokud je (hodnota stavového registru AND 02) rovna nule, čekáme. A přesně to zajišťuje další instrukce BEQ. Vzpomeňte si – pokud je příznak Z=1 (tedy předchozí operace skončila s výsledkem 0), tak BEQ skáče. Tady se skáče opět na načtení stavového bajtu a vše se opakuje, dokud není výsledek nenulový. V tu chvíli už víme, že má 6850 volno a můžeme vysílat.
Pokud je tedy volno, přečteme si ze zásobníku zpět hodnotu, co byla původně v registru A a pomocí STA ji zapíšeme do datového registru 6850 – ACIAData.
Správná otázka je: Co se stane, když náhodou bude obvod 6850 vadný, nebo nebude zapojený správně a bude vracet pořád hodnotu 0? V takovém případě, ano, tušíte správně, jste právě vygenerovali nekonečnou smyčku, ve které se bude procesor točit do skonání věků – pardon, do vypnutí napájení, do RESETu nebo do přerušení.

JEŠTĚ TO DÁT DOHROMADY…

Už zbývá vlastně jen drobnost – vzít jednotlivé znaky onoho slavného nápisu a jeden po druhém vyslat po sériové lince ven. Znaky zapíšeme do paměti pomocí pseudoinstrukce DB, která uloží hodnotu jednotlivých znaků jako jejich ASCII kód. Řetězec ukončíme hodnotou 0. Tím smyčka pozná, že je konec a že už je všechno vysláno. K adresaci použijeme indexový registr Y. Na počátku bude jeho hodnota 0 a po každém znaku se hodnota zvýší o 1. Pomocí adresního módu ABY (absolutní adresa, indexovaná s registrem Y) budeme načítat do registru A postupně jednotlivé znaky Slavného Nápisu a pak budeme volat SEROUT, aby je odeslal po sériové lince – tedy do terminálu, který je zobrazí.

```
1          LDY     #0
2  LOOP:
3          LDA     Message,Y
4          BEQ     DONE
5          JSR     SEROUT
6          INY
7          BNE     LOOP
8  DONE:   JMP DONE
9
10 Message:
11         DB      $0C,"My hovercraft is full of eels!",$0D,$0A,$00
```

Po načtení znaku je jednoduchý test: je-li načtený znak roven 0, tak instrukce LDA nastavila příznak Z na jedničku. Instrukce BEQ v takovém případě skočí na návěští DONE. Následuje volání podprogramu (JSR), zvýšení ukazatele (indexového registru Y) a pokud ještě není 0 (což by znamenalo, že se vyslalo 256 bajtů a jedeme znovu od začátku), tak se skáče na návěští LOOP, tedy na načtení bajtu z adresy (Message+Y).
Na návěští DONE je pak nekonečná smyčka, která de facto zastaví procesor. V reálném nasazení to asi nebude žádoucí, ale pro nás je dobré, aby procesor udělal, co udělat má, a pak nikde netrajdal a nedělal něco, co dělat nemá.

A TO JE VŠECHNO?

Ano, to je všechno. Celá ta nádhera vypadá takhle:

```
1  ; Nastavení adres pro komunikační obvod ACIA 6850
2  ACIA         =  $A000
3  ACIACONTROL  =  ACIA+0
4  ACIASTATUS   =  ACIA+0
5  ACIADATA     =  ACIA+1
6
7  ; program začíná na adrese C000h, tedy tam, kde začíná ROM
```

```
8   .ORG    $C000
9   ; Emulátor má začít odsud
10  .ENT    $
11  ; K testu použij emulátor počítače SBC6502 - pouze pro IDE ASM80.com
12  .ENGINE sbc6502
13  ; Vstupní adresa
14  RESET:
15  ; Nastavíme si ukazatel zásobníku
16  LDX     #$FF
17  TXS
18  ; Nastavení řídicího registru ACIA
19  LDA     #$15
20  STA     ACIAControl
21
22  ; Začínáme vypisovat znaky, Y je ukazatel
23  LDY     #0
24  LOOP:
25  LDA     Message,Y ; Načti znak ze zprávy na pozici Y
26  BEQ     DONE      ; Jestli je to 0, tak hop!
27  JSR     SEROUT    ; Jinak zavolej podprogram pro vyslání znaku
28  INY               ; Y++ - abychom adresovali další bajt
29  BNE     LOOP      ; a jestli toho ještě nebylo dost, tak hop na začátek
30  DONE:     JMP     DONE      ; TO JE KONEC!!! :(
31
32  Message:
33  DB      $0C,"My hovercraft is full of eels!",$0D,$0A,$00
34
35  ; podprogram pro vyslání hodnoty z registru A
36  ; přes sériový obvod 6850 na terminál
37  SEROUT:   PHA     ; Uschováme hodnotu, protože registr A potřebujeme
38  SO_WAIT:  LDA     ACIAStatus ; Je volno?
39  AND     #2        ; Bit 1 nám to řekne
40  BEQ     SO_WAIT   ; Není? Tak to zkusíme znovu, dokud nebude
41  PLA               ; Už je, takže si vrátíme zpět hodnotu z registru A
42  STA     ACIAData  ; a pošleme ji do 6850
43  RTS               ; už není co na práci, tak se můžeme vrátit
44
45  ; Nastavíme vektory, které 6502 potřebuje, aby věděl, kam
46  ; má po resetu systému skočit.
47  .ORG    $FFFC
48  DW      reset
49  DW      reset
```

Krásné, že?

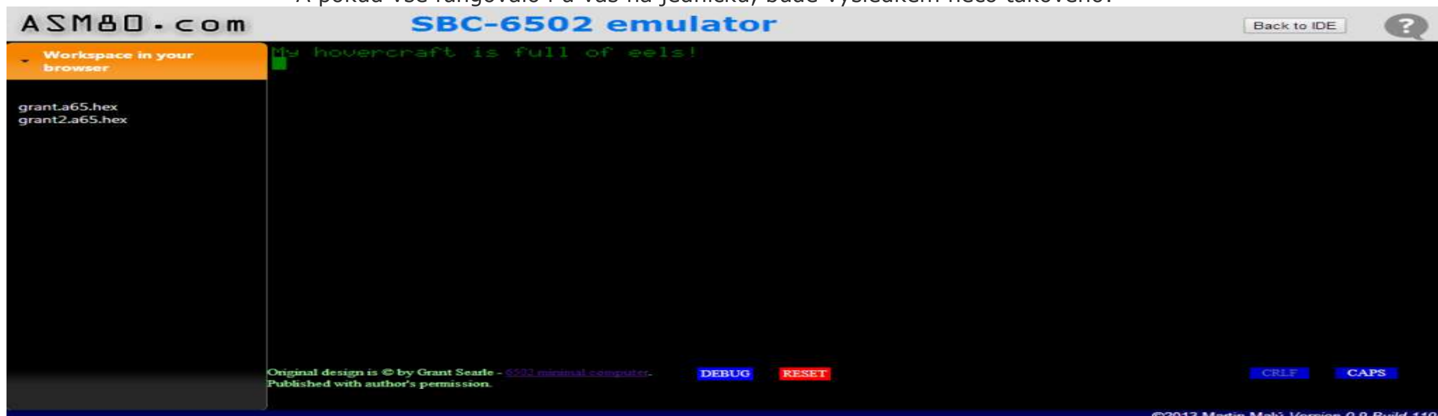
Možná vás zarazily dvě věci. Jednak že hexadecimální čísla zapisuju ne jako 0C000h, ale jako $C000. Je to ekvivalentní zápis, ale byl jsem, po právu, upozorněn, že zápis s $ na začátku je „klasičtější" a snáze rozpoznatelný.

Druhá věc, co vás mohla zarazit, jsou kódy $0c, $0d a $0a u řetězce k výpisu. Jedná se o řídicí znaky pro terminál. 0C smaže celou obrazovku, 0D přesune kurzor na začátek řádku, 0A přesune kurzor na nový řádek.

Jak to otestovat? Nejjednodušší bude použít moje IDE ASM80.com, kde je k dispozici překladač i emulátor. Zkopírujte si výše uvedený zdrojový kód do okna editoru, uložte jej (vpravo tlačítko Save file as…) pod názvem „strojak1.a65" (přípona .a65 říká překladači, že je použitý procesor 6502) a zkuste kliknout na Compile (nebo stisknout F9). Pokud je vše OK, vypíše se zpráva o úspěšném překladu a vlevo, v seznamu souborů, přibudou dva soubory strojak1.a65.hex a strojak1.a65.lst. Těch si teď všímat nemusíte a vesele můžete stisknout F10 (nebo kliknout na Emulator). Díky direktivě „.engine", kterou jsme použili v kódu, se nespustí debugger, na jaký jsme zvyklí z ukázek kódu, ale rovnou emulátor počítače SBC6502, do paměti se uloží náš přeložený kód a spustí se.

(Místo kopírování do editoru můžete kliknout na tento odkaz: Přidat strojak1.a65 do workspace ASM80.com – tím se objeví v seznamu souborů. Pak ho můžete jednoduše otevřít kliknutím na jeho název v levém sloupci.)

A pokud vše fungovalo i u vás na jedničku, bude výsledkem něco takového:



Tak. Děkuju za pozornost u zatím snad nejdelšího článku, případné dotazy a připomínky prosím jako vždy do komentářů.

6502 – !etěvs ,johA

Všechno jde udělat i jinak, hlavně v osmibitovém assembleru. Takže ani pro Hello, world není jen jediný předepsaný postup. V první ukázce jsem použil postup, kterému se říká ASCIIZ – tedy ASCII řetězec, ukončený bajtem s hodnotou 0 (ASCII + Zero). Tento způsob zápisu řetězců používají třeba překladače jazyka C – kdo z vás zná Céčko, tak ví.

Jazyk Pascal používal jiný přístup – první bajt udával délku řetězce ve znacích, a pak následovaly kýžené znaky. Někdy může být tento přístup výhodnější – hlavně tehdy, když nám délku spočítá překladač a natvrdo uloží do kódu. Jak by se to přepsalo do assembleru 6502?

Tak, základ zůstane stejný – definice konstant, inicializace i rutina SEROUT, jen ten vnitřek se změní. Registr X použijeme jako počítadlo a index znaku v řetězci. Když dosáhne hodnoty rovné délce řetězce, přestaneme. Ta zásadní pasáž kódu bude vypadat nějak takhle:

```
1  ; Začínáme vypisovat znaky
2  TOUT:   LDX   #0 ; pozice
3  TLOOP:  LDA   text,x
4          JSR   SEROUT
5          INX   ; X++
6          CPX   #tsize ; už jsme na konci?
7          BNE   tloop ; pokud ne (tj. X<délka), tak pokračujeme
8
9  ENDLOOP:
10         JMP   endloop
11
12 TEXT:   DB    $0C,"My hovercraft is full of eels!",$0D,$0A
13 TSIZE   EQU   $-TEXT
```

A CO WOZ?

Logout mě upozornil, že Steve Wozniak v monitoru pro Apple I použil trik, kterým ušetřil několik bajtů (a tedy taktů procesoru). Ačkoli mi jeho trik připadá v tomto případě jako klasický příklad „overengineeringu" (tedy optimalizace až přehnaná), tak si ji ukážeme, protože ilustruje schopnost podívat se na problém z naprosto neobvyklého úhlu. Což se zase při programování v assembleru často hodí.

Úvodní úvaha je jednoduchá: na začátku se nuluje registr X, v průběhu se pak zvyšuje o 1 a kontroluje se, jestli už má hodnotu N. Kdybychom na začátku do registru X uložili hodnotu N a šli v opačném pořadí, tedy směrem k nule, tak by odpadla instrukce porovnání a konec by nastal ve chvíli, kdy instrukce DEX (X=X-1) dojde k nule. Pak nastaví příznak Z (viz popis instrukce). Nese to s sebou jeden drobný problém: čtení znaků by fungovalo od konce. Hm, a co? Tak je tam zapíšeme pozpátku!

```
1  ; Začínáme vypisovat znaky
2  TOUT:   LDX   #tsize ; X je počet znaků, co zbývý vypsat
3  TLOOP:  LDA   text-1,x
4  ; A protože X jde od hodnoty TSIZE k nule,
5  ; tak se znaky berou od konce
6          JSR   SEROUT
7          DEX   ; X--
8          BNE   tloop ; a dokud není 0, tak pokračujeme
9
10 ENDLOOP:
11         JMP   endloop
12
13 TEXT:   DB    $0a,$0d,"slee fo lluf si tfarcrevoh yM",$0c
14 TSIZE   EQU   $-TEXT
```

Ušetřili jsme dva bajty a nějaký ten takt, přišli jsme o snadnou čitelnost. Ano, při programování osmibitů jsou situace, kdy dva bajty či pár taktů znamená hodně. (Spousta programátorů, co na assemblerech vyrostla, pak logicky považuje cokoli jiného za plýtvání ad absurdum.)

JEŠTĚ NĚJAKÝ TRIK, PROSÍM!

„Ale no jistě," odpovědělo sluchátko. Pojďte se podívat na následující kód:
Přeložte si ho, spusťte emulátor a pojďme krokovat:

```
1  0000              .ORG 0
2  0000 A2 FF        LDX #$FF
3  0002 9A           TXS
4  0003 20 09 00     JSR label
5  0006 4C 00 00     JMP 0
6  0009 60    LABEL: RTS
```

První instrukce (adresa 0000, 2 bajty) nastaví X na hodnotu FFh, druhá (adresa 0002, 1 bajt) tuto hodnotu zkopíruje do ukazatele zásobníku S. Třetí instrukce (adresa 0003, 3 bajty) je instrukce volání podprogramu. Volá se podprogram na adrese 0009…

V tuto chvíli se zastavíme a podíváme se na vrchol zásobníku, co se tam uložilo. Víme, že instrukce JSR ukládá dva bajty návratové adresy. Protože byl ukazatel nastaven na FFh, budou tyto dva bajty na adresách 01FEh a 01FFh. Co tam najdeme? Na zásobníku je uložena hodnota 0005. Vidíme, že to není adresa instrukce za voláním JSR (ta je 0006), ale o 1 nižší. Instrukce RTS vezme hodnotu ze zásobníku, k ní přičte 1 a na tu adresu skočí.

K čemu nám bylo tohle mentální cvičení? Ukážeme si totiž jeden princip, který se u osmibitových assemblerů používá docela často, a nejčastěji právě u výpisu různých textů. Pokud se text vyskytuje v programu jen jednou a je konstantní (tj. typicky nějaké hlášení), tak je pro programátora pohodlné napsat ho přímo do kódu, tam, kde potřebuje. Ušetří tím sekvenci „zadej někam adresu hlášení, co chceš vypsat – zavolej rutinu, která vypíše řetězec ze zadané adresy". Místo toho jen zavolá podprogram, jehož funkce se dá popsat slovy „vypiš znaky, co se nacházejí za instrukcí JSR, a až narazíš na ukončovací znak 00, tak se vrať za tu nulu, tam pokračuje program."

Nějak takhle (stále upravujeme kód z předchozího příkladu):

```
1    JSR     PRIMM
2    DB      $0C,"My hovercraft is full of eels!",$0D,$0A,$00
3
4    DONE:    JMP     DONE ; TO JE KONEC!!! :(
```

Vidíte, že je tam instrukce volání podprogramu PRIMM (PRint IMMediately), za ní jsou přímo znaky požadované hlášky, ukončené nulou, a za tím zase pokračuje program.
Co musí udělat podprogram PRIMM? Představte si, že je vyvolán. V tu chvíli je na zásobníku „adresa instrukce za JSR – 1“. Ukazatel SP je jeden bajt POD touto hodnotou, návratová adresa je tedy na adresách SP+1 a SP+2.
Nejdřív si uložíme pracovní registry A, X a Y – tím se SP sníží o 3 a situace na zásobníku bude vypadat takto:

SP+5	Vyšší bajt návratové adresy
SP+4	Nižší bajt návratové adresy
SP+3	Obsah registru A
SP+2	Obsah registru X
SP+1	Obsah registru Y
SP	První volná pozice na zásobníku

Takže na adrese SP + 0100h + 4 je nižší bajt návratové adresy, na adrese SP + 0100h + 5 je vyšší. Tuto hodnotu si můžeme někam zkopírovat – ideálně do zero page do dvou buněk vedle sebe. Výhodně pak využijeme adresní mód IZY. Připomeňme si: tento mód vezme adresu ze dvou vedle sebe ležících paměťových míst, k té adrese přičte obsah registru Y a výsledek udává adresu, kam se má sahat pro data.
Jakmile narazíme na konec řetězce (anebo nám přeteče registr Y), tak končíme s vypisováním. Teď je potřeba vzít tu původní adresu, k ní přičíst počet vypsaných znaků (tedy registr Y), tu pak zase zapsat na zásobník – a pak už jen standardně vrátit obsah registrů a provést RTS.
A protože vlastní studium zdrojového kódu řekne víc, než sáhodlouhé popisy, tak bez dalšího vysvětlování – podprogram PRIMM:

```
1    PRIMM:
2    PHA     ; Uložím A
3    TXA
4    PHA     ; Uložím X
5    TYA
6    PHA     ; Uložím Y
7    TSX     ; Ukazatel na zásobník si načtu do X
8    LDA     $0104,X ; Nižší byte návratové adresy
9    ; ($0100 je základní adresa zásobníku, X je tu aktuální
10   ; ukazatel zásobníku, +4 proto, že ukazatel SP ukazuje na
11   ; první volné místo, SP+1 je uložený registr X,
12   ; SP+2 je uložený registr Y, SP+3 je uložený registr A
13   ; (na začátku podprogramu jsme si je ukládali)
14   ; SP+4 a SP+5 jsou nižší a vyšší bajt návratové adresy,
15   ; tedy poslední bajt instrukce JSR
16
17   STA     $00 ; Uložíme do ZP (třeba na adresu 00)
18   LDA     $0105,X ; Analogicky vyšší byte návratové adresy...
19   STA     $01 ; ... ukládáme do ZP na adresu 01
20   LDY     #$01 ; Nastavíme Y na počáteční hodnotu. Měla by to být
21   ; nula, ale protože víme, že návratová adresa je ve
22   ; skutečnosti o 1 nižší, než adresa prvního bajtu za JSR,
23   ; tak začneme od jedničky.
24   PRIM2:
25   LDA     ($00),Y ; Načteme bajt. Adresa je "obsah buněk 00 a 01" + Y
26   BEQ     PRIM3 ; Načetli jsme nulu? Tak končíme!
27
28   JSR     SEROUT ; Nenulový znak ale vypíšeme
29   INY     ; posuneme se na další adresu
30   BNE     PRIM2 ; a pokud jsme ještě nepřetočili počítadlo, tak
31   ; pokračujeme v tisknutí znaků.
32   ; Když už je Y nulové, tak je načase skončit.
33
34   PRIM3:
35   TYA     ; V Y je "počet znaků + 1" - přesuneme do A
36   CLC     ; budeme sčítat, je potřeba vynulovat C
37   ADC     $00 ; K A si přičteme nižší bajt původní návratové adresy
38   STA     $0104,X ; a "podvrhneme" ji do zásobníku
39   LDA     #$00 ; Vynulujeme A
40   ADC     $01 ; a přičteme hodnotu vyššího byte návratové adresy.
41   ; Pokud při předchozím sčítání došlo k přenosu, tak se
42   ; vyšší bajt zvedne o 1, jinak zůstane stejný
43   STA     $0105,X ; A opět vyšší bajt návratové hodnoty analogicky
44   ; uložíme na zásobník a budeme se tvářit, že to tak
45   ; už bylo
```

```
46 PLA ; Přečteme uloženou hodnotu
47 TAY ; co patří do registru Y
48 PLA ; a úplně stejně tu, co
49 TAX ; patří do registru X
50 PLA ; ještě původní hodnotu A
51 RTS ; a návrat!
```

Můžete si zkusit složit celý zdrojový kód a vyzkoušet, jak hezky funguje. (Nebo si můžete kliknout sem a on se vám automaticky přidá do pracovního prostoru v ASM80 IDE).

(Rutina PRIMM pochází z operačního systému Commodore C64 a je mírně upravena, viz zdroj.)

Pro pozorné: v kódu ~~je jedna chyba, která~~ jsou dvě chyby, které se projeví při určité konstelaci – zkuste na ně přijít.

Mimochodem – existuje ještě jeden používaný způsob označování konce řetězců, hlavně u anglických textů. Kromě zadaného počtu znaků + řetězce nebo řetězce ukončeného bajtem 00h (u CP/M služba pro vypisování řetězců používá ukončování znakem $) můžeme použít i trik, který počítá s tím, že anglická abeceda si v ASCII vystačí se znaky z rozsahu 00h-7Fh. Pak stačí poslednímu znaku nastavit nejvyšší bit na 1 (tj. posunout jej do rozsahu 80h-FFh). Ze znaku „!" (kód 21h) se tak stane znak s kódem A1h. No a postup je prostý – před vypsáním znaku použijeme AND s hodnotou 7Fh (abychom nastavili nejvyšší bit na 0), znak vypíšeme a pak zkontrolujeme, jestli není nejvyšší bit roven 1 (u 6502 třeba tak, že ho načteme do registru A – tím se nejvyšší bit zkopíruje do příznaku N). Pokud ano, byl to poslední znak a my se můžeme vrátit. Napsání rutiny, která bude takto pracovat, nechám už na vás, máte to za domácí úkol…

6502 – čtení kláves a pořádek v kódu

Dnes nás čeká druhý krok při oživování počítačů. Už nás pozdravil, tak ještě aby nás poslouchal. A až bude poslouchat, tak si řekneme něco o štábní kultuře.

Když jsme začali experimentovat se SBC6502, tak jsem popisoval obvod ACIA. V počítači SBC6502 je k tomuto obvodu připojen terminál. Terminál ale není jen obrazovka, která vypisuje znaky. Je tam i klávesnice, kterou se s počítačem komunikuje. Klávesnice je připojená úplně stejně jako obrazovka, jen komunikace probíhá opačným směrem, tj. do počítače. Když se podíváte do popisu obvodu ACIA, zjistíte dvě užitečné informace:

- Přijatý znak si můžete přečíst z datového registru, pokud tedy byl nějaký znak přijatý.
- O tom, jestli byl nějaký znak už přijatý, informuje bit 0 stavového registru.

Takže teoreticky by mělo stačit kontrolovat bit 0 stavového registru, a pokud bude nastavený, tak to znamená, že přišel nějaký znak z klávesnice. V takovém případě ho můžeme přečíst z datového registru.

Co s ním uděláme? Hmmm… co ho třeba zase vypsat? To by šlo:

```
1 ; Nastavení adres pro komunikační obvod ACIA 6850
2 ACIA equ $A000
3 ACIACONTROL equ ACIA+0
4 ACIASTATUS equ ACIA+0
5 ACIADATA equ ACIA+1
6
7 ; program začíná na adrese C000h, tedy tam, kde začíná ROM
8 .ORG $C000
9 ; Emulátor má začít odsud
10 .ENT $
11 ; K testu použij emulátor počítače SBC6502 - pouze pro IDE ASM80.com
12 .ENGINE sbc6502
13 ; Vstupní adresa
14 RESET:
15 ; Nastavíme si ukazatel zásobníku
16 LDX #$FF
17 TXS
18 ; Nastavení řídicího registru ACIA
19 LDA #$15
20 STA ACIAControl
21
22 ; Začínáme vypisovat znaky, Y je ukazatel
23 LDY #0
24 LOOP:
25 LDA Message,Y ; Načti znak ze zprávy na pozici Y
26 BEQ key ; Jestli je to 0, tak hop!
27 JSR SEROUT ; Jinak zavolej podprogram pro vyslání znaku
28 INY ; Y++ - abychom adresovali další bajt
29 BNE LOOP ; a jestli toho ještě nebylo dost, tak hop na začátek
30 KEY: LDA ACIAStatus ; Přišel nějaký znak?
31 AND #1 ; Bit 0 nám to řekne
32 BEQ KEY ; Nepřišel? Tak to zkusíme znovu, dokud nějaký nepřijde
33 LDA ACIAData
34 JSR serout
35
36 JMP KEY ; TO JE KONEC!!! :(
37
38 MESSAGE:
39 DB $0C,"My hovercraft is full of eels!",$0D,$0A,$00
40
41 ; podprogram pro vyslání hodnoty z registru A
42 ; přes sériový obvod 6850 na terminál
43 SEROUT: PHA ; Uschováme hodnotu, protože registr A potřebujeme
44 SO_WAIT: LDA ACIAStatus ; Je volno?
```

```
45      AND     #2 ; Bit 1 nám to řekne
46      BEQ     SO_WAIT ; Není? Tak to zkusíme znovu, dokud nebude
47      PLA     ; Už je, takže si vrátíme zpět hodnotu z registru A
48      STA     ACIAData ; a pošleme ji do 6850
49      RTS     ; už není co na práci, tak se můžeme vrátit
50
51      ; Nastavíme vektory, které 6502 potřebuje, aby věděl, kam
52      ; má po resetu systému skočit.
53      .ORG    $FFFC
54      DW      reset
55      DW      reset
```

Použil jsem první příklad s vypisováním znaků a dopsal právě kontrolu klávesnice a výpis znaků. Zkuste si tenhle příklad přeložit v ASM80.com a spustit v emulátoru SBC6502. Program vypíše hlášku a po ní čeká na stisknutí klávesy. Jakmile je nějaká stisknuta, pošle terminál její ASCII kód po sériovém rozhraní do počítače, tam jej zpracuje procesor a vypíše zpátky.
SBC6502 nepoužívá přerušení, které by systém upozornilo na to, že přišel znak a je možno ho zpracovat. Proto procesor musí pravidelně kontrolovat, jestli se už něco neděje, a během té kontroly nedělá nic jiného. V nejjednodušších single task systémech je to přijatelné řešení.
Složitější systémy mohou použít přerušení, buď od obvodů klávesnice, které signalizují, že přišel znak, nebo třeba přerušení od časovače (ZX Spectrum takhle testovalo klávesnici každou padesátinu sekundy při systémovém přerušení). V obsluze přerušení pak načtou znak a uloží ho do nějakého bufferu k dalšímu zpracování.

ORGANIZACE KÓDU

Protože vy, čtenáři, snad všichni znáte nějaké vyšší jazyky, tak nemusím moc složitě představovat koncepty modulů a lokálních proměnných. Ano, i tyhle věci v assembleru máme, ale není to tak úplně prosté…

MODULY

No, říkejme tomu tak. Ve skutečnosti se jedná jen o jednoduchý INCLUDE „jméno“, který na to místo načte obsah externího souboru.

Některé staré assemblery vůbec žádný include neměly. Ono by to třeba u ZX Spectra 48 s páskou nebylo moc pohodlné. Čímž neříkám, že takové kompilery nebyly, třeba HiSoft C měl #include, jak se na céčko sluší a patří, a při překladu jste spustili magnetofon, kde byly soubory ke slinkování, překladač si je prošel, načetl, přeložil… Ano, bylo to tak děsivé, jak to zní.

Většina modernějších assemblerů include samozřejmě má, jen se liší jeho syntax. Některé assemblery používají .INC, některé INCLUDE, některé .INCLUDE, takže nezbývá než si přečíst manuál k tomu kterému kousku. Já ve svém překladači používám tvar *.INCLUDE název souboru*.

Řekněme, že mi připadá jako dobrý nápad (a on to dobrý nápad je) přesunout tyhle rutiny pro výpis znaku a načtení znaku někam stranou, do nějaké společné (common) knihovny (library), kterou si důvtipně nazvu „comlib.a65“. V hlavním programu tak budu moci vesele tyhle rutiny používat, aniž by mi překážely ve zdrojáku, stačí jen, když je vhodně includuju.
Tím se nám zdroják rozštípnul na dva soubory: comlib.a65 (knihovna) a vlastní zdrojový kód.

```
1       ; COMLIB.A65 - základní komunikační knihovny
2
3       ; Nastavení adres pro komunikační obvod ACIA 6850
4       ACIA equ        $A000
5       ACIACONTROL equ     ACIA+0
6       ACIASTATUS equ    ACIA+0
7       ACIADATA  equ     ACIA+1
8
9       ACIAINIT:      ; Nastavení řídicího registru ACIA
10      LDA     #$15
11      STA     ACIAControl
12      RTS
13
14      ; podprogram pro vyslání hodnoty z registru A
15      ; přes sériový obvod 6850 na terminál
16      SEROUT:   PHA     ; Uschováme hodnotu, protože registr A potřebujeme
17      SO_WAIT:  LDA     ACIAStatus ; Je volno?
18      AND     #2 ; Bit 1 nám to řekne
19      BEQ     SO_WAIT ; Není? Tak to zkusíme znovu, dokud nebude
20      PLA     ; Už je, takže si vrátíme zpět hodnotu z registru A
21      STA     ACIAData ; a pošleme ji do 6850
22      RTS     ; už není co na práci, tak se můžeme vrátit
23
24      ; podprogram pro načtení stisknuté klávesy do registru A
25      ; přes sériový obvod 6850
26      ; Podprogram čeká na stisk klávesy!
27      SERIN:    LDA     ACIAStatus ; Je klávesa?
28      AND     #1 ; Bit 0 nám to řekne
29      BEQ     SERIN ; Není? Tak to zkusíme znovu, dokud nebude
30      LDA     ACIAData ; a přečteme z 6850
31      RTS     ; máme hotovo, tak se můžeme vrátit
```

Tenhle soubor pak elegantně načteme v hlavním programu:

```
1       ; program začíná na adrese C000h, tedy tam, kde začíná ROM
2       .ORG    $C000
3       ; Emulátor má začít odsud
4       .ENT    $
5       ; K testu použij emulátor počítače SBC6502 - pouze pro IDE ASM80.com
```

```
6	.ENGINE sbc6502
7	; Vstupní adresa
8	RESET:
9	; Nastavíme si ukazatel zásobníku
10	LDX     #$ff
11	TXS
12	
13	JSR ACIAINIT
14	
15	LOOP:    JSR SERIN
16	JSR SEROUT
17	
18	JMP     LOOP ; Stále dokola...
19	
20	; ještě někam musíme vložit tu knihovnu...
21	; třeba sem, sem se hlavní program nedostane
22	.include comlib.a65
23	
24	.ORG    $FFFC
25	DW      reset
26	DW      reset
```

Do comlib.a65 si klidně můžeme přihodit i rutinu PRIMM z minulé lekce. K tomu ale až na konci. Teď si musíme ukázat ještě jednu důležitou vlastnost assemblerů…

Assembler totiž sám o sobě nemá lokální jména. Jakmile jednou nadefinujete konstantu, návěští, něco, tak to je vidět v celém kódu. Což je docela problém, protože u složitějšího programu vám brzy dojde fantazie při pojmenovávání např. smyček. „LOOP1", „LOOP2", … to není moc elegantní.

Nemluvě o tom, že třeba použijete návěští „LOOP" v nějaké knihovní funkci. V hlavním programu na to zapomenete (nebo o tom ani nevíte, protože tu knihovnu dělal někdo jiný), a překladač – logicky – zařve, že návěští bylo už použité. Co s tím?

LOKÁLNÍ NÁVĚŠTÍ

V téhle situaci se hodí lokální návěští. Špatná zpráva je, že ne každý assembler je podporuje, a pokud ano, tak má svou konvenci, které bude pravděpodobně odlišná od všech ostatních konvencí všech ostatních assemblerů.

Některé mocnější assemblery zavádějí pseudokonstrukce „procedure" a deklaraci „local" apod., jiné se staví k problému z druhé strany a dovolují pro drobné smyčky a návěští používat jakási „pseudonávěští" a odkazovat se na ně zápisem „skoč na předchozí pseudonávěští", „skoč na následující pseudonávěští", „skoč o dvě pseudonávěští zpátky"…

Já jsem v ASM80 zvolil cestu bloků. Blok začíná direktivou „.block" a končí direktivou „.endblock". Všechna návěští, co jsou v něm definována, jsou lokální. To znamená že v bloku na ně můžete odkazovat, mimo něj nejsou vidět. Pokud chcete, aby bylo návěští vidět i mimo blok, dejte před jeho název znak @ – ten se nestane součástí jména, jen říká, že toto návěští bude globální. „@SEROUT:" říká „Definuj globální návěští se jménem SEROUT".

Díky tomu můžu jako první řádek knihovny comlib napsat .block, na poslední .endblock, a vím, že pokud takovou knihovnu includuju, tak mi z ní nic „nevyteče" ven, pokud explicitně neřeknu, co má být vidět zvenčí. Takže nová, *šetrná* verze comlib vypadá takto:

```
1	; COMLIB.A65 - základní komunikační knihovny
2	
3	.block
4	; díky deklaraci BLOCK nebudou následující návěští vidět
5	; ve zbytku kódu, kromě těch, před kterými je @
6	
7	; Nastavení adres pro komunikační obvod ACIA 6850
8	ACIA equ       $A000
9	ACIACONTROL equ     ACIA+0
10	ACIASTATUS equ    ACIA+0
11	ACIADATA  equ     ACIA+1
12	
13	@ACIAINIT:      ; Nastavení řídicího registru ACIA
14	LDA     #$15
15	STA     ACIAControl
16	RTS
17	
18	; podprogram pro vyslání hodnoty z registru A
19	; přes sériový obvod 6850 na terminál
20	@SEROUT:   PHA     ; Uschováme hodnotu, protože registr A potřebujeme
21	SO_WAIT:  LDA     ACIAStatus ; Je volno?
22	AND     #2 ; Bit 1 nám to řekne
23	BEQ     SO_WAIT ; Není? Tak to zkusíme znovu, dokud nebude
24	PLA     ; Už je, takže si vrátíme zpět hodnotu z registru A
25	STA     ACIAData ; a pošleme ji do 6850
26	RTS     ; už není co na práci, tak se můžeme vrátit
27	
28	; podprogram pro načtení stisknuté klávesy do registru A
29	; přes sériový obvod 6850
30	; Podprogram čeká na stisk klávesy!
31	@SERIN:   LDA     ACIAStatus ; Je klávesa?
32	AND     #1 ; Bit 0 nám to řekne
33	BEQ     SERIN ; Není? Tak to zkusíme znovu, dokud nebude
```

```
34    LDA     ACIAData ; a přečteme z 6850
35    RTS     ; máme hotovo, tak se můžeme vrátit
36    .endblock
```

Díky uzavření, „zapouzdření" zdrojáku můžu v hlavním programu použít klidně návěští SO_WAIT, a nedojde k chybě, protože to, které jsem použil v comlib.a65, bude vidět pouze v comlib.a65, nikde jinde. Stejně tak názvy jako ACIAData, ACIAStatus apod. Jediné, co bude vidět zvenčí, je SERIN, SEROUT a ACIAINIT.

VYLEPŠENÝ PRIMM

Minule jsem tu ukazoval funkci PRIMM, která pomocí jednoduchého triku dokáže vytisknout konstantní řetězec znaků, zapsaný přímo v kódu, přímo za voláním JSR PRIMM. Kód funguje většinou dobře, ale jsou dva stavy, ve kterých fatálně selže.

První případ je, že je počet znaků větší než 255. Registr Y u posledního znaku „přeteče" do nuly, rutina tím končí, ale bohužel se tím návratová adresa ocitne někde uprostřed textu a výsledek bude katastrofální. Řešení existuje – nechat přetočit Y, ale přitom si uloženou adresu zvýšit o 0100h, jak navrhnul v komentářích Roman Bórik.

Druhý problém nastane v případě, že ukazatel zásobníku SP je nízko. V takovém případě přeteče přes nulu, dostane se opět do vysokých hodnot FDh-FFh, a v takovém případě jednoduchý přepočet pomocí vzorce „SP + 104h" selže, protože se ocitneme mimo zásobník, na adresách 0200h a vyšších, a nejen že načteme nesmysly, ale taky nesmysly uložíme. I tuto situaci by bylo možné ošetřit, ale v tomto případě to není asi úplně potřeba, pokud inicializujeme zásobník standardně, tj. na hodnotu FFh. Pokud se totiž za takové situace stane, že se zásobník protočí přes nulu, tak máme zásadnější problém někde jinde…