M A C H I N E   C O D E - A S S E M B L E R

Z 8 0

Učebnice programování mikroprocesoru Z80

Ladislav Zajíček, 1986

****

O B S A H

****

Předmluva

KAPITOLA 1 — 1
Několik slov k assemblerovým panicům (i pannám?)

KAPITOLA 2 — 13
Programovací jazyky a způsoby zápisu

KAPITOLA 3 — 17
Registry, paměť a datový přenos

KAPITOLA 4 — 28
Adresovací módy Z80

KAPITOLA 5 — 40
Skoky, volání a návraty

KAPITOLA 6 — 52
Logické instrukce

KAPITOLA 7 — 59
Bitové manipulace, rotace a posuvy

KAPITOLA 8 — 66
Aritmetické instrukce a blokové prohledávání

KAPITOLA 9 — 75
O interfacingu

KAPITOLA 10 — 84
Závěr

Příloha

****

Předmluva

****

Máte v rukou studijní materiál pro samostatnou výuku programování ve strojovém kódu, resp. assembleru.

Tato učebnice není čtením do tramvaje, ale ani vysokoškolskými skripty. Látka je podána netradičním způsobem, který se snaží vás uvést do jejích tajů co nejpřátelštějším způsobem. Na některých místech jsou alespoň stručně začleněny i obtížnější detaily (princip adresování, Morganův teorém apod.), bez nichž se v amatérské praxi klidně obejdete, proto není nutné, abyste se jimi zabývali dopodrobna. Jsou spíše nástinem toho, že assemblerové instrukce netvoří výpočetní techniku. Jsou jen jedním z nástrojů jimiž to všechno, co leží "za nimi", můžeme ovládat.

V závěru jednotlivých kapitol najdete programové příklady užití instrukcí Z80, které tvoří nejdůležitější, praktickou část celé výuky. Je proto nezbytné, abyste si příklady ozkoušeli přímo na počítači a zároveň si sami dávali obdobné úlohy k řešení. Jedině tak se vám podaří plně porozumět tomu, co se v programech odehrává. Pokud byste se např. snažili pochopit stavové indikátory Z80 jen čtením textu, byl by to poměrně ztracený čas. Umístěním jakéhokoli programu ve strojovém kódu do počítače a jeho následným krokováním pomocí monitoru strojového kódu vám bude funkce indikátorů rázem srozumitelná, stane se poznanou zkušeností (monitory paměti názorně ukazují všechny změny indikátorů, k nimž v průběhu programu dochází). Proto práci s příklady věnujte zvýšenou pozornost.

Naučit se programovat ve strojovém kódu není nikterak prosté. Jedna věc je znát instrukce Z80, druhá umět je používat. Proto mějte na paměti, že i když se vám podaří obsah učebnice zvládnout do 3-6 měsíců, teprve další letitá praxe, naplněná studiem dalších souvisejících materiálů z vás postupně udělá programátora, před nímž neobstojí žádný sebesložitější problém. Pochopení práce s mikroporocesorem Z80 vám však umožní v mnohem větší míře pochopit jakýkoli vyšší programovací jazyk, protože všechny jsou odvozeninami téhož zdroje. Podobně vám pak někdy v budoucnu nebude činit větších potíží přechod na programování 16ti či 32tibitových mikroprocesorů.

Obsah učebnice vychází ze studia množství zahraniční literatury i z vlastní praxe autora. Vedle autorských ukázek je většina programových příkladů převzata ze seriózních pramenů, v nichž se chyby vyskytují zřídka. Pokud autor některé objevil, samozřejmě je pro vás opravil. S vědomím toho, že každý se při studiu programování může dostat do "slepé uličky", nabízí autor možnost krátkých telefonických konzultací (nikoli tvorbu programů za vás) na pražském telefonním čísle 5358183 (jen mezi 14. a 18.hod.).

V učebnici je jen stručná zmínka o interfacingu, který je samostatnou disciplínou výpočetní techniky. Pokud byste si chtěli rozšířit své vědomosti o něm, opatřete si některou z knih uvedených v doporučené literatuře.

I když jste teprve na začátku studia, přečtěte si napřed kapitolu 10, v níž jsou shrnuty některé základní principy přístupu k programování ve strojovém kódu. Napoprvé vám z ní nebude leccos jasné, ale nastíněný přístup je platný i pro vlastní studium obsahu této učebnice.

...a čas od času se k závěrečné kapitole vracejte.

****

KAPITOLA 1

Několik slov k začátečníkům

****

Tuto kapitolu mohou zkušenější inženýři computerových programů přeskočit. Budeme se v ní jen stručně, "letem světem" zabývat vnitřním organismem počítače jak z hlediska jeho stavby, tak i manipulace a zpracování dat. Adepti výpočetní techniky, kteří se chtějí naučit programovat ve strojovém kódu či assembleru, by však tuto kapitolu vynechat rozhodně neměli.

Ping pong jedniček a nul

Myslete si třeba číslo 15. Máte? Jak je máte uloženo ve své vlastní paměti? Jako číslice 1 a 5 vedle sebe? Nebo 10 a 5? Nebo 3 krát 5? Nebo jinak? A jak vnímá patnáctku počítač? Takhle:

$$
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0
\end{array}
$$

Je to divné? Přinejmenším nezvyklé. Ale železně logické. Tato logika se jmenuje binární (dvojková), zatímco my jsme navyklí počítat v soustavě dekadické. Počítač skutečně umí myslet jen ve dvou výrazech - všechno je buď 0 nebo 1. Rozložení jedniček a nul v horní řádce je zároveň rozložením velkých nebo malých nábojů (potenciálů) na vstupech nebo výstupech hradel (tranzistorů), které se buď tohle naše číslo 15 chystají zpracovat (vstup) podle našich instrukcí, nebo je oněch 15 výsledkem nějaké operace (výstup). Tyto jedničky a nuly "běhají" v počítači mezi jednotlivými jeho částmi po osmi vodičích (datová sběrnici). To platí pro osmibitové mikropočítače; šestnácti nebo dvaatřicetibitové jich mohou mít 16 nebo 32. Utajili jsme vám ještě adresovou sběrnici (má dvakrát více vodičů), pomocí níž určujeme, na jakou adresu se mají tyto jedničky a nuly v paměti poslat, nebo z které adresy paměti je vzít pro další zpracování. Je to opravdu nesmírně rychlý ping pong jedniček a nul. Jedno jejich přemístění, přesněji provedení jedné instrukce (příkazu), trvá pár nanosekund.

Každá z oněch osmi jedniček nebo nul představuje jeden bit. Všech osm tvoří jeden bajt. To jsou základní jednotky výpočetní techniky. Proč je jich právě osm? Inu, mohl by jich být i jiný počet, ale pro binární logiku se ukázal být nejvhodnější počet osmi bitů v jednom bajtu i proto, že tak můžeme snadno užít i hexadecimální číselnou soustavu (2 krát 8 je 16). Je zvykem označovat bit vpravo za pořadově nejnižší (nultý), ten vlevo za nejvyšší (sedmý). Čím vyšší bit, tím vyšší mocnina čísla 2 (ale to už si "domýšlíme" my - počítač to neví, pro něj platí jen ta logická 0 nebo log.1). Abychom se dobrali nějakého výsledku, ten bit, který má hodnotu log.1, pro nás vždy znamená, že číslo 2 umocníme číslem, označujícím pořadí onoho bitu v bajtu. Kde bude log.0, nebudeme činit nic. Výsledek všech mocnění pak sečteme. Zkuste to u našeho čísla 15. Postupujeme od nejnižšího bitu:

1 + 2 + 4 + 8 + 0 + 0 + 0 + 0 je opravdu 15

Otázka - kolik čísel můžeme vyjádřit pomocí nižších čtyř bitů jednoho bajtu? Pokud odpovíte, že 15, chyba lávky. Je jich 16 - nezapomeňte, že když jsou všechny čtyři bity ve stavu log.0, pak je výsledek roven nule. A nula je také číslo! A právě ve výpočetní technice velmi důležité! Zkuste tedy znova a správně odpovědět - kolik čísel můžeme vyjádřit pomocí všech osmi bitů jednoho bajtu? A v jakém intervalu?

Nalinkované myšlení

Napřed odpověď na otázku. Největší dekadické číslo, které můžeme vyjádřit jedním bajtem, je 255. Včetně nuly může tedy jeden bajt nabýt 256 různých hodnot. Díky registrům a instrukčnímu souboru (příkazům) mikroprocesorů může bajt reprezentovat i čísla záporná, ale o tom až později.

Pro pochopení dějů odehrávajících se v počítači je nutné poznat způsob jeho myšlení. Musíme se trochu zbavit ješitných představ o tom, že jediný možný způsob myšlení je myšlení lidské. Připusťme, že počítač má jistou umělou inteligenci, tedy i svůj způsob myšlení. Pleteme se teď poněkud do oblastí filozofických, ale už dnes lze s jistotou říci, že současný způsob zpracování informací dnešními počítači je základem vývoje budoucích umělých inteligencí vyššího stupně.

Počítač s polovodičovým mikroprocesorem zpracovává všechny příkazy a údaje programu, který do něj vložíme, postupně. V každém okamžiku je schopen provádět jen jednu jedinou operaci, aniž by jakkoli chápal souvislosti nebo význam a obsah zpracovávaných dat. Jeho jedinou předností je vlastně jen rychlost, s jakou vložené informace dokáže zpracovat. A tak, chudák, dostal i nehezkou přezdívku "nesmírně rychlý blbec". Myšlení počítače jde skutečně po jediné lince, kterou mu určujeme my naším programem. Ale když tvrdíme, že je počítač tak rychlý, tak by přece měl projít od začátku do konce programu za chvilku, ne? Jistě, to by mohl. Jenže my mu do cesty stavíme něco, bez čeho by celá výpočetní technika nebyla myslitelná – podmínky, vlivem kterých počítač provádí cykly (opakování operací) a odskoky v programu. Jak třeba počítač vynásobí 3 krát 5? Schématicky znázorněno mu musíme připravit takovýto program:

1. vynuluj určené místo (adresu A) v paměti
2. uchop do programu námi vložené číslo 5 a přičti je k obsahu adresy A (5 plus 0, tedy 1 krát 5)
3. už jsi číslo 5 přičetl třikrát? POKUD NE (podmínka), vrať se a opakuj
4. přičti číslo 5 k obsahu adresy A (už tam bylo 5, teď to bude 10, tedy 2 krát 5)
5. už jsi číslo 5 přičetl třikrát? POKUD NE, vrať se a opakuj
6. přičti číslo 5 k obsahu adresy A (už tam bylo 10, teď to bude 15, tedy 3 krát 5)
7. už jsi číslo 5 přičetl třikrát? POKUD NE, opět se vrať (na tuto podmínku teď počítač samozřejmě nereaguje); POKUD ANO, pípni, zablikej, zobraz na monitoru obsah adresy A a zastav se (čekej na případný další příkaz)

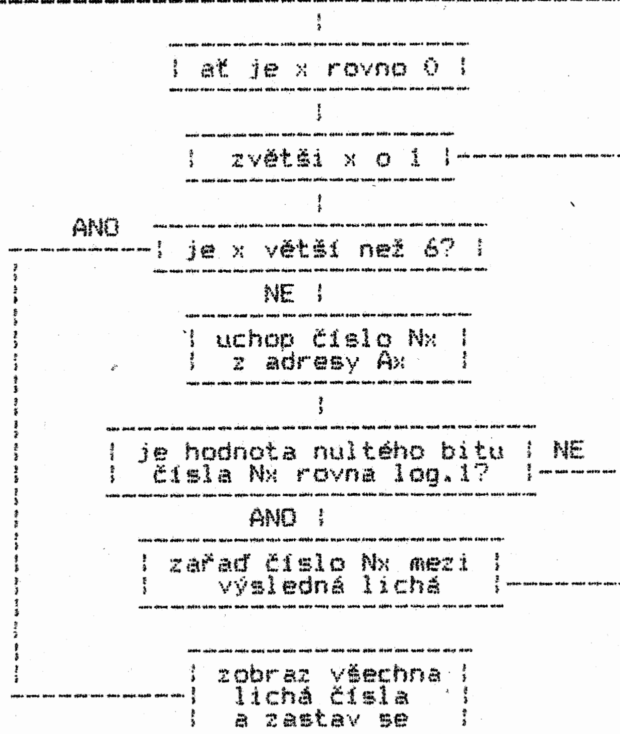
Počítač pípne, obrazovka zabliká a na ní se objeví číslo 15 (pokud je necháme zobrazit v soustavě dekadické). Kdybychom se podívali na bitovou prezentaci bajtu na adrese A, objevila by se nám už dobře známá sestava 0 0 0 0 1 1 1 1. Jak z "nalinkovaného myšlení", které musíme pro sestavení jakéhokoli programu plně respektovat, vyplývá, počítač neumí ani malou násobilku. Umí totiž jen "sečítat" elektrické potenciály (log.0 a log.1) na vstupech několika typů hradel (různě zapojených spínacích tranzistorů), resp. počítat v hranicích jednoduché Booleovy algebry (o ní si povíme později).

Schéma vývoje programu se samozřejmě nerozepisuje tak zeširoka. Vývojový diagram (angl.flow chart), který je mnohem přehlednější, si nakreslíme níže. Zkuste si ale už například sestavit vývojové schéma programu, který bude postupně zjišťovat, která z řady čísel 22, 14, 53, 2, 64, 123 jsou lichá. S tím souvisí základní otázka pro správnou konstrukci vývoje programu zadaného úkolu – čím se v bitové prezentaci bajtu od sebe liší sudá a kladná čísla?

BAJT – základní slovo jazykového Babylonu

Podstatou minulé úlohy byla odpověď na otázku: Kdy je obsah bajtu sudý a kdy lichý? Řešení je nasnadě – **vše závisí na hodnotě nultého (nejnižšího) bitu. Má-li hodnotu log.0, je bajt vždy sudý, při log.1 je vždy lichý.** Ostatní bity reprezentují vždy buď "pořadovou" mocninu dvojky (vyšší než nultou), nebo mohou být samy nulou – proto je výsledkem součtu všech těchto sedmi bitů vždy číslo sudé, resp.0. Teprve přičtení hodnoty nultého bitu rozhodne, zda obsah celého bajtu reprezentuje číslo sudé či liché.

Jednoduchý **vývojový diagram** pro vyhledání lichých čísel v intervalu od 0 do 255 z řady např. šesti čísel Nx (kde x postupně nabývá hodnoty 1 až 6), uložených na sousedících adresách Ax, bude vypadat následovně:



Vložili-li jsme ke zkoumání řadu šesti čísel z naší minulé úlohy (22,14,53,2,64,123), na obrazovce se objeví všechna lichá čísla této řady, tedy 53 a 123 - jejich nultý bit má hodnotu log.1.

Povšimněte si mluvy našeho diagramu. Není vám nápadné, že zkoumá jen, zda je "něco" větší nebo menší či rovno "čemusi". Nebo k něčemu něco přičítá či "tomu" přiřazuje hodnotu, něco někam ukládá a zase to odebírá...rozhodně žádná velká květomluva. Takovýchto operací je opravdu poměrně omezený počet. A tak jsme se dostali i k mluvě, resp. **syntaxi programovacích jazyků**, pro jejichž výčet by vám dnes už nestačily prsty obou rukou. Všem těmto jazykům se říká **vyšší programovací jazyky** - jsou vlastně systémovým programátorem vytvořenou důmyslnou soustavou řady menších podprogramů ve strojovém kódu. Tento kód je zcela nejzákladnější řečí nám už známých primárních nosičů informace - bajtu a jeho bitů.

Každý z nás, kdo si opatřil mikropočítač, si dříve či později položil otázku - programovat ano či ne? Pokud se ještě rozhodujete, přečtěte si další kapitolku.

Kolik jazyků umím, tolikrát jsem programátorem?

Odpověď na danou otázku je jednoznačná - NE. Obdobně bychom se mohli zeptat - Kolik (lidských) jazyků znám, tolikrát jsem básníkem? Co tedy vlastně dělá programátora programátorem? Tuto otázku stavíme jako zásadní z toho důvodu, aby si každý, kdo se chce pustit do zápolení s bajty, uvědomil, co ho čeká, resp. zda jím vynaložená námaha bude adekvátní dosahovaným výsledkům.

Dobrý programátor musí nejen znát "gramatiku" jednotlivých jazyků, ale musí také chápat vlastní systémovou strukturu každého z nich. Musí do značné míry znát a umět využívat hardware počítače, s nímž pracuje, i hardware a software k němu připojených periférií. Musí své znalosti stále prohlubovat, trvale sledovat světový vývoj celé oblasti. Tolik jen velmi stručně ke znalostem. V jejich pozadí, jak je tomu vždy u lidského umu, stojí talent a pracovitost. Zdatný programátor musí mít velkou invenci a fantazii, jež jsou nezbytností každé tvůrčí práce. Tomu všemu jsou pak úměrné výsledky, kterých dosahuje, a nakonec i uspokojení z míry vlastní seberealizace, která přináší nemalý užitek i ostatním uživatelům jeho programů.

Na nejvyšší příčce hierarchie programátorů stojí programátoři systémoví. Ti vymýšlejí programovací jazyky a obecně jakýkoli software, jenž je sám o sobě subsystémem pracujícím pro nějakou aplikaci, která teprve tvoří "nadstavbu" základního systému. Tak např. databanka je systémem, který "ožívá", "jest aplikován" teprve vkládáním a zpracováváním našich dat. V tomto případě hovoříme o **programech uživatelských**, při jejichž užití není

znalost programování nutná. Systémy, které jsou subsystémy, nebo i "jen" pomocnými systémy (především jazyky a programy pro práci se systémem počítače včetně jeho periferií) pro vytváření různých jiných programovaných aplikací, nazýváme **programy systémovými**. Jejich využití není možné bez znalosti programování.

Všichni samozřejmě nemůžeme být systémovými programátory či konstruktéry computerového hardwaru. Většina z nás bude svůj mikropočítač využívat tak, aby jim sloužil jako, řekněme, o dost chytřejší lednička nebo video. To by však nemělo znamenat, že se zřekneme základních informací o celé výpočetní technice. Už teď víte, co je bajt, bit, co se v počítači zhruba odehrává, i něco o tom, jak se sestavuje program. Pro ty z vás, kteří chtějí, aby se jim vědomosti o počítačích staly pracovním nástrojem, přidáme v následujících kapitolách plyn.

Kolik kterých jazyků?

Basic, Forth, Pascal, Ada, PL-1, Fortran, Micro Prolog, Logo, Karel - to je jen několik názvů z řady těch nejznámějších vyšších programovacích jazyků. Který nebo které z nich si tedy vybrat? Možná vás trochu překvapí tvrzení, že nemusíte pracovat ani s jedním z nich, a přesto se můžete stát vynikajícím programátorem. Jak to tedy vlastně je?

Mikroprocesor, jako srdce mikropočítače, "pumpuje" svými komorami (říkáme jim registry) proud nábojů (log.1 a log.0). Krevní oběh představují datové a adresové sběrnice, kterými protékají jednotlivé náboje (bity bajtů) z a do buněk (adres) počítačového těla (hardwaru) podle instrukcí (programu, tedy softwaru), jež do počítače vložíme. Programátor, který zná hardware svého počítače, se s ním dokáže dorozumět přímo, pomocí primárních nosičů informace - bitů a bajtů. Program do počítače ukládá tak, že na jednotlivých adresách vytváří takové bitové sestavy jednotlivých bajtů (co adresa, to jeden bajt), které mu po spuštění programu dokáží prezentovat jeho představy jako perfektně fungující soukolí, kde všechno do sebe zapadá, jak má, a "nezadrhne" se (program nezkolabuje). Takovéto softwarové manipulaci s dispozicemi hardwaru se říká programování ve strojovém kódu (angl.machine code). Věc má ovšem svůj háček. Programové instrukce (příkazy), pomocí nichž velíme mikroprocesoru, sestávají z jednoho až čtyř po sobě jdoucích bajtů. Každý bajt má 256 kombinací rozmístění jedniček a nul. A i když se v instrukčním souboru mikroprocesoru nevyužívá všech těchto kombinací, zapamatovat si, jaká znamená který příkaz, je poněkud nad síly normálního člověka. Konstrukce programu ve strojovém kódu dále zabere mnoho času, sama o sobě je dost komplikovaná. Odměnou za ni je perfektnost, rychlost, s níž program pracuje a možnost využití všech dispozic počítačového hardwaru.

Právě důvody vysoké obtížnosti konstrukce programu ve strojovém kódu vedly k vymyšlení, vývoji **vyšších programovacích jazyků**. Ty nejsou ničím jiným, než zahuštěním více příkazů strojového kódu do jednotlivých příkazů toho kterého jazyku. Snahou je vytvořit takovou jazykovou syntaxi, která programování co nejvíc zpřehlední. Protože se jedná opravdu o "vymýšlení si", počítač by na nás při jeho oslovení takovým jazykem koukal naprosto nechápavě - pokud by neměl při sobě tlumočníka, který ovládá jak strojový kód, tak i náš jazyk. Takového tlumočníka musí tvůrce jazyku k němu vždy dodat. Jsou jich dva typy:

1. Interpretor - překládá příkazy jazyka v průběhu programu, "stále dokola"; typickým představitelem je Basic

2. Kompilátor - převádí (kompiluje) program do strojového kódu "jednou provždy"; příkladem budiž Pascal

Funkce tlumočníků je v tom, že ono příkazové (syntaktické) zahuštění zpětně rozpustí do jednotlivých instrukcí strojového kódu. Asi uhádnete základní nedostatek takové komunikace. Je stejný jako při tlumočení rozhovoru dvou lidí, z nichž každý nezná jazyk toho druhého - časová prodleva, o níž se vzájemná komunikace prodlužuje. Pochopitelně je delší u interpretace, i když kompilace má zase některé jiné "zádrhele". Dalším a nikoli posledním nedostatkem je nemožnost využití všech dispozičních nuancí počítače. Proto - chtě nechtě - musí každý dobrý "vyšejazyčný" programátor občas skočit přímo mezi bity bez pomoci přítele tlumočníka.

Tak se nám aspoň trochu osvětlila odpověď na titulní otázku. Znalost každého jazyku je dobrá, každý má něco specifického. Bez znalosti strojového kódu však nikdy nebudeme "panem Programátorem". A umět sestavit efektivní program ve strojovém

kódu předpokládá i znalost hardwaru techniky, s níž pracujeme. Kruh se uzavírá. Abychom v něm nezabloudili, skočíme rovnýma nohama do základní soft-hardwarové logiky výpočetní techniky, které kraluje Booleova algebra.

Hradla v řečišti bitů

Tok bitů v počítači podléhá předem stanoveným zákonitostem, na nichž je celá orientace jeho systému postavena. Zákonem nejvyšším je **Booleova algebra**. Ve své transformaci do výpočetní techniky se jedná o velmi jednoduchý logický systém, v principu pracující jen se dvěma hodnotami - jedničkou a nulou. Tvůrce této logiky, anglický matematik G.Boole, ji objevil v 19. století, kdy vize o počítacích strojích vzrušovala jen několik málo nepokojných a vynalézavých duchů. Všichni ostatní považovali Booleovu algebru za zajímavý matematický výstřelek bez jakéhokoli užitku - nikoli první, avšak ani poslední ironie evoluce lidského poznání.

I když Booleova algebra byla matematiky postupně rozpracována do značné šířky, my si povšimneme hlavních tří logických funkcí, které jsou základem operací s bity a bajty (a doplníme je čtvrtou - XOR - ze souboru Z80). Tyto operace vykonávají logická hradla, která prošla bouřlivým konstrukčním vývojem od relé přes elektronky a tranzistory až po dnešní integrované obvody, v jejichž mikroarchitektuře jsou propojeny statisíce hradel - spínacích tranzistorů a pasivních prvků (odporů a kondenzátorů) na jednom malém silikonovém čipu.

U každé z logických funkcí je uvedena pravdivostní tabulka, která udává, jaká hodnota bude na výstupu Q hradla při uvedené kombinaci hodnot na obou jeho vstupech A a B (u funkce NOT jen jednom - A):

AND (logický součin)

vstupy A	vstupy B	výstup Q
0	0	0
0	1	0
1	0	0
1	1	1

OR (logický součet)

vstupy A	vstupy B	výstup Q
0	0	0
0	1	1
1	0	1
1	1	1

NOT (negace,inverze)

vstup A	výstup Q
0	1
1	0

XOR (exkluz.součet)

vstupy		výstup
0	0	0
0	1	1
1	0	1
1	1	0

Kromě základních funkcí existuje řada dalších, které jsou jejich kombinací, např. NOR (NOT-OR), NAND (NOT-AND) a kombinací těchto kombinací. Jaký význam má tato logika ve vlastním hardwarovém zapojení polovodičových hradel? Naprosto zásadní. Vezměme si např.ukládání hodnot bajtů na jednotlivé adresy paměti RAM. Podle toho, jaká kombinace je na vstupech hradel, je na každém z osmi konečných výstupů (pro každý bit jeden) hodnota log.1 nebo log.0 - tím je dána i číselná reprezentace takového bajtu na dané adrese, kterých může osmibitový mikroprocesor obsáhnout celkem 65536 (včetně adresy 0) neboli 64K neboli 65,536 kB (**1K je 1024 bajtů, ale 1 kB je 1000 bajtů**). Potenciál, který odpovídá hodnotám log.0 nebo log.1, je pro každý bit udržován na kondenzátoru, připojeném k danému hradlu, které napětí na kondenzátoru buď sníží (log.0) nebo zvýší (log.1). Paměť RAM je tedy vlastně **pamětí kapacitní**. Má ale tu nepříjemnou vlastnost, že její kondenzátory neudrží přidělený potenciál déle, než zlomek sekundy. Proto je počítač tak trochu zároveň občerstvovací stanicí svých neustále "unavených" kondenzátorků, na nichž v rychlých **občerstvovacích (angl.refresh) cyklech** udržuje patřičnou energii; podobně asi jako artista s točícími se talíři na pružných tyčích musí stále "osvěžovat" zastavující se talíře, aby nespadly na zem. To se týká dynamických pamětí RAM. Statické RAMky potenciál udrží, ale na úkor zvýšení spotřeby proudu a tepelného sálání.

Význam uvedené logiky je samozřejmě stejně významný i pro softwarové ovládání počítače a jeho periférií. Objasňovat si jej budeme v dalších kapitolách, proto mějte pravdivostní tabulky po ruce.

Logice bajty neutečou

Připomeňme si obsah části o nalinkovaném myšlení počítače. Do programově vytvořené linky jsou programátorem vřazeny některé testovací funkce (podmínky), pomocí nichž je řízen běh programu. V momentech splnění takové podmínky je proveden odskok do jiné části programu - podobně jako ve složitém kolejišti projíždí vlak tak, jak mu dovolují nastavované výhybky. Vytvoření takového programového kolejiště, v němž se výhybky přehazují tak, aby lokomotiva nejen nevykolejila, ale projížděla elegantně a co nejefektivněji jen tudy, kudy má, je jedním ze znaků programátorské dovednosti.

Představme si, že máme v programu děj, který se stále opakuje, tzv. **programovou smyčku (angl.loop)**. Takových smyček je v každém programu celá řada. My si pro další výklad zvolíme jednoduchou časovací smyčku, jejímž umístěním kdekoli v programu můžeme pozdržet průběh jakékoli jeho části, resp. ji přesně časovat.

Ale před tím se ještě musíme seznámit s dvoubajtovým číslem. Zatím máme tu čest jen s jednobajtovým. Víme např., že je-li všech osm bitů bajtu ve stavu log.1, je bajt roven číslu 255. Při číselné reprezentaci bajtů se však nemusíme omezovat jen na jeden bajt. A tak si hned přibereme druhý a oba posadíme vedle sebe. Pro jednoduchost všechny bity uvedeme do stavu log.1:

Vyšší bajt								Nižší bajt							
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}

Mocnina dvojky nám současně ukazuje pořadové číslo každého bitu v tomto páru bajtů. Protože jsou všechny bity ve stavu log. 1, jsou všechny mocniny platné. Sečteme-li je dohromady, získáme velmi významné číslo 65535, které nám zároveň říká, kolik adres může adresovat (a manipulovat s jejich obsahy - bajty) každý osmibitový mikroprocesor. Ten má totiž dvoubajtovou - tedy 16tibitovou - adresovou sběrnici. Pokud jste text bedlivě sledovali, určitě už jste vzkřikli - Chyba! Ano, adres je 65536, protože i adresa 0 existuje - ovšem nejvyšší adresovatelný bajt paměti leží na adrese 65535. Počítat se dvoubajtovými čísly můžeme díky tomu, že osmibitový mikroprocesor má i párové registry. Do každého registru můžeme uložit jen jeden bajt; **párové registry mohou s oběma bajty operovat buď jako s jedním číslem v rozsahu 0-65535 nebo je vnímat jako dvě samostatná čísla, každé v rozsahu 0-255.** Záleží jen na programátorovi, pro kterou eventualitu se kdy rozhodne.

Naše plánovaná časovací smyčka bude mít za úkol vykonat 65535 opakovaných cyklů a po jejich vykonání dát příkaz k jejímu ukončení odskočením na jinou programovou adresu, kterou si nazveme KONEC. Celý krátký prográmek napíšeme v jazyku assembler, který má ze všech jazyků nejblíž ke strojovému kódu. Použijeme přitom zápisu v "nářečí" jednoho z nejužívanějších osmibitových mikroprocesorů Z80 firmy Zilog. Strategie tvorby smyčky je tato: musíme napřed někam uložit číslo 65535 (samozřejmě do párových registrů, zde BC). Budeme od něj postupně odečítat vždy 1, čímž obdržíme 65535 průchodů, tedy cyklů (každý začíná na adrese ZNOVA). A nakonec musíme zjistit, kdy budou registry BC obsahovat nulu, abychom celý cyklus ukončili. Pokud bychom test nezařadili, dostali bychom se do tzv. věčné smyčky, z níž není lehko uniknout. Moment vynulování těchto registrů nesmíme propást. K číhání na okamžik, kdy budou všechny bity obou bajtů ve stavu log.0, použijeme Booleovy logiky. Program pak bude vypadat následovně (vpravo je komentář k jednotlivým instrukcím):

```
        LD BC,65535   ;Ulož do reg.BC číslo 65535
ZNOVA:  DEC BC        ;Sniž obsah reg.BC o 1
        LD A,B        ;Ulož do reg.A obsah reg.B
        OR C          ;Proveď log.operaci OR mezi reg.A a C
        JP NZ,ZNOVA   ;Není-li výsledkem 0, vrať se na adr.ZNOVA
        JP KONEC      ;Je-li výsledkem 0, skoč na adresu KONEC
```

Vemte si k ruce pravdivostní tabulky. U log. funkce OR vidíte, že pouze při porovnání dvou nulových bitů je výsledkem log.0 (jinak vždy log.1). Co to znamená pro náš program? Ze po přesunu obsahu reg.B do reg.A a následné operaci OR reg.A s reg.C bude výsledkem nula tehdy a jen tehdy, budou-li všechny bity obou registrů ve stavu log.0 (logické instrukce se účastní vždy ty dva bity, které v obou bajtech mají stejné pořadové číslo). A proč přesouváme napřed obsah reg.B do reg.A? Jednoduše proto, že mikroprocesor umí provádět log. operace jen s registrem A (operaci B OR C tedy nezná).

Uvedená forma výuky programování se rovná hození neplavce do vody. Naši časovací smyčku i jednoduchou řeč assembleru použijeme při dalším výkladu funkce mikroprocesoru. Během následujících tří pokračování se vše rozjasní natolik, že budete moci sami sestavit krátký fungující assemblerový program (což dokonce někteří programátoři vůbec neumějí!).

Mikropočítačová kardiologie

Srdce to lidské, ach bože přebože, za lásku může snad, za zlobu nemůže... Umělci všeho druhu se odjakživa zabývali srdečními problémy; bohužel k nim v tomto století přibyly i početné týmy lékařů. Zda bude někdy planout láskou srdce počítače - mikroprocesor - přenechme leckdy veselým úvahám computerových metafyziků. My se na tohle srdce podíváme spíše pohledem lékaře, abychom se seznámili s jeho základními funkcemi.

Mikroprocesor sestává z několika registrů. Můžete si je představit jako srdeční komory nebo i prozaičtěji jako šuplíky kredence. Vemte si k ruce schématické znázornění registrů Z80 z přílohy. Jednotlivé registry mají tyto názvy:

reg.A - Accumulator (akumulátor)
reg.F - Flags (praporky - stavové indikátory)
reg.BC - Byte Counter (bajtový čítač)
reg.DE - DEstination (určení - cílová adresa přenosu bajtů)
reg.HL - High Low (vyšší a nižší část dvoubajtové adresy)
reg.IX - pro indexované adresování
reg.IY - " "
reg.SP - Stack Pointer (ukazatel zásobníku adres návratů atd.)
reg.PC - Program Counter (programový čítač - adresa kroku)
reg.R - Refresh (občerstvovací - viz výše)
reg.I - Interrupt vector (vektor přerušení)

Všechny registry značené dvěma písmeny jsou 16tibitové. Registry BC, DE a HL zvané **párové** tvoří důležitou výjimku v tom, že kteroukoli jejich "půlku" lze používat i jako samostatný (osmibitový) registr. Registry značené jen jedním písmenem jsou jednobajtové. Názvy registrů jsou zvoleny velmi vtipně a adekvátně jejich poslání. V tom opět určitou výjimku hrají registry BC, DE, HL, které jsou mnohem univerzálnější, než by napovídaly jejich názvy.

Registry A,F,B,C,D,E,H,L jsou v mikroprocesoru obsaženy dvakrát, ve dvou bankách, číslovaných 0 (hlavní) a 1 (vedlejší). Registry vedlejší banky se značí apostrofem u jejich názvu. K jejich aktivování je musíme programově "prohodit" s registry banky hlavní.

Výsadní postavení mezi všemi mají registry A a F. K reg.A se vztahují některé zvláštní operace, které nemůžeme provádět se žádným jiným registrem (především to jsou logické operace, jak jsme se s tím již setkali v našem minulém programu, a operace aritmetické).

Zcela specifické funkce plní reg.F a jeho **stavové indikátory (zkráceně SI)**. Každý z nich je reprezentován jedním bitem, který může být programově testován, zda se nachází ve stavu log.1 nebo log.0. Výsledek takového testu je pak směrodatný pro další běh programu. Jedná se o test nám již známých podmínek, jejichž splněním jsou prováděny programové odskoky. Jako každý jiný registr, má i reg.F osm bitů svého bajtu, avšak v praxi se jich využívá jen šest - v každém programovém kroku (realizaci jednoho příkazu) může být testován vždy jen jeden ze čtyř (CY,Z,S,P!V).

Průběh jakéhokoli programu je plně řízen mikroprocesorem, ev.i s ním spolupracujícími koprocesory. Jsme opět u ping pongu jedniček a nul jednotlivých bajtů mezi adresami paměti a mikroprocesorem. Jím jsou bajty zpracovány podle námi zadaného příkazu a opět "odpinknuty" na adresu svého určení, ev. na tzv. port, ze kterého jsou odebrány připojenou periferií, která ovšem může stejně tak posílat přes **port** "své" bajty do počítače pro jejich další zpracování programem.

Je li počítač pod napětím, ale zrovna "nežije" žádným programem, neznamená to, že nežije vůbec. V počítači (a samozřejmě i mikroprocesoru) probíhá současně **řada precizně časovaných operací**, které jsou vnitřně synchronizovány kmitočtem zabudovaného oscilátoru. Mikroprocesor má řadu vstupních a výstupních linek (celkem 38), jimiž probíhají jednotlivé digitální impulsy a které slouží nejen pro přenos dat po datové a adres po adresové sběrnici, ale upozorňují jej, zda má v ten který moment "číst" (READ) nebo "zapisovat" (WRITE) bity a bajty či "čekat" (WAIT) na programem definovaný vnitřní nebo vnější podnět atd., atd.

Tak jako lidské, organické srdce, je i to anorganické počítačové napohled poměrně jednoduché (pár komůrek). Při podrobnějším zkoumání bychom však zjistili, že se jedná o jednoduchost, která je složitostí samou. V naší další práci si proto budeme všímat jen toho "jednoduššího", co nám k obecnému užitku přináší vnitřní složitost mikroprocesoru a všech jeho životních funkcí. Sejdeme se v registru F.

Svědomitý výhybkář

Uvolněme svou fantazii a představme si program uložený do paměti jako kolej. Už víme, že program tvoří programátor tak, že ukládá jednotlivé bajty na adresy paměti počítače. A tak si adresy v naší koleji představme jako pražce. Mikroprocesor, který "luští" význam jednotlivých bajtů na adresách (pražcích), si představme jako lokomotivu. Tato lokomotiva je napojena speciálními pantografy na řadu trakcí (vodičů), které spojují mikroprocesor především s adresovou a datovou sběrnicí, pomocí níž sbírá bitová data z každého právě přejížděného pražce adresově určeného léti bity reg.PC (ten určuje každý programový krok - posun od jednoho pražce ke druhému).

Teď se vraťme k minulému programu (časovací smyčce). Jeho první instrukce ukládá číslo 65535 do párových registrů BC. Má assemblerový tvar LD BC,65535 a ve strojovém kódu je jako tříbajtová uložena do tří adres v tomto pořadí: 01 FF FF. Binárně tak každý z pražců reprezentuje toto bitové vyjádření: 00000001 11111111 11111111.

Když lokomotiva najede na první pražec, reaguje tak, že bajt 01 "pochopí" jako počátek instrukce LD BC, "něco". Ví hned, že to "něco" bude obsahem dvou bajtů ležících na příštích dvou pražcích. Zároveň si připraví registry BC a po postupném přejetí zbývajících dvou pražců do nich převede námi na ně uložené hodnoty FF FF. Tím je pro ni instrukce skončena. Ví, že na dalším pražci bude zase první bajt instrukce nové. Jak už bylo řečeno, instrukce mohou být jedno- až čtyřbajtové. Proč je tomu tak, zde nebudeme rozebírat; berte to jako fakt. Důležité je, že mikroprocesor vždy postupně pozná, kolik bajtů která instrukce obsahuje a nesplete se.

Obě hexadecimální čísla FF jsou decimálně 255 v rozsahu 0-255, což je 256 celočíselných kombinací. Číslo FFFF uložené v obou párových registrech je decimálně 65535 (tedy 255*256+255) v rozsahu od 0 do 65535 (tedy 65536 kombinací). Mějte na paměti, že vyšší bajt vždy čítá "přetečení" nižšího při jeho přeplnění přes mezní hodnotu 255 (a naopak). Tak výsledek součtu 255+1 bude ve dvoubajtovém hexadecimálním vyjádření 0100.

Vraťme se k pohybu naší lokomotivy. Pokud by program plynul hladce, bez podmínek (výhybek), lokomotiva by prostě projela "přímo za nosem" ze startovní do cílové stanice. Tak tomu ovšem nebývá. Konstruktéři mikroprocesorů je vybavili skvělým náčiním, které podstatně rozšiřuje možnosti tvorby programu, resp. jeho struktury. Program tak může sestávat z řady menších kolejových úseků (podprogramů), kterými lze projet vždy znova, kdykoli potřebujeme, aby program vykonal určitou funkci či operaci, která je v některém z těchto úseků naprogramována.

Výše zmíněným náčiním jsou stavové indikátory registru F (angl.flags); viz minulá část. Označme si je zkráceně SI. Testováním těchto SI - zda jsou ve stavu log.0 nebo log.1 - zároveň rozhodujeme, zda lokomotiva pojede dál, nebo přejede (vychýlíme ji - vpřed či vzad) do námi předem daného místa koleje, odkud bude v "luštění" pražců pokračovat (vždy ve směru od nižší adresy k vyšším). Tyto odbočky (přejezdy) do jiných míst ovšem žádné pražce nemají - jedná se spíše o přeskok z jednoho místa koleje na druhé. Mikroprocesor jej vykoná vmžiku.

Každý ze šesti užívaných SI, tedy bitů registru F, testuje (přesněji řečeno - indikuje) něco jiného - o tom dále. V každém případě si však zapamatujme základní pravidlo. Indikátory jsou ovlivňovány neustále během chodu programu a stále indikují výsledný stav každé právě provedené operace. Tak např. SI zvaný Z (zero), tedy nulový indikátor, se dostane vždy do stavu log.1, když výsledkem provedené operace je nula. Jinak je ve stavu log.0. Zařazením testu tohoto SI hned po provedené (a testované) operaci můžeme lokomotivě zavelet, aby přejela na jiné místo koleje, či pokračovala dál. A můžeme si ještě vybrat, zda tak má učinit jen tehdy, je-li nulový SI ve stavu log.1 nebo log.0.

Z každého programového testu tedy vedou jen dvě námi programově předem dané cesty, lokomotiva se však vydá jen po jedné z nich. Adresa pražce (lze říci i jeho pořadové číslo) se objeví před každým programovým krokem v reg.PC. Tak se

mikroprocesorová lokomotiva orientuje, kudy má jet. Jede-li přímo, bez odbočení, obsah reg.PC se stále zvyšuje o 1 (o další pražec). Při odbočení se vydá přímo na adresu, kterou jsme v programovém testu určili a která se při splnění zadané podmínky (testu) objeví předem v reg.PC. Svědomitý výhybkář bydlící v reg.F a obsluhující reg.PC nás nikdy nezklame.

Volání programové divočiny

Nejeden panic i muž již tomuto počítačovému vábení podlehl (bohužel v míře mnohem menší se tak děje u slabšího pohlaví). Nebudeme se však zabývat vábeními citovými, ale programovými instrukcemi volání.

Jak se dozví naše mikroprocesorová lokomotiva, na jakou adresu (pražec) odbočit, je-li splněna podmínka pro takové odbočení? Velmi jednoduše. V instrukcích volání (ale i skokových - o nich později) je obsažena naráz jak podmínka pro případné přehození výhybky, tak i adresa, na kterou má lokomotiva odbočit a najet. Assemblerový tvar instrukce je: CALL podmínka,adresa. Angl.slovo call znamená volej. Tato instrukce může stát i sama o sobě, bez uvedení podmínky: CALL adresa. V tom případě lokomotiva vždy odbočí na udanou adresu.

Abychom si udělali představu o tom, jaké podmínky můžeme zadávat (a jakými podmíněnými instrukcemi), musíme znát stavové indikátory registru F. (Podrobněji si všimneme jen těch, které budeme teď potřebovat. Informace o ostatních najdete v dalších částech učebnice.)

C (Carry) - indikátor bitového přenosu - lidově "přetečení" (nebo "podtečení"). Je ve stavu log.1 vždy, když je výsledkem testované operace přeplnění bajtu - buď "přetečení", tedy iluzorně by jeho binární obsah měl být vyšší než 255, nebo "podtečení", kdy by obsah měl být nižší než 0. Indikátor C značíme v textu jako CY, aby se nám nepletl s registrem C.

H (Half Carry) - slouží interní indikaci přenosu ve vztahu k bitům 3 a 4 (v pořadí ovšem čtvrtému a pátému!) uvnitř samotného bajtu. Jedná se tedy o totéž jako u indikátoru CY - s tím rozdílem, že se testuje jen přenos mezi bity 3 a 4.

Z (Zero) - nulový indikátor - je ve stavu log.1, když výsledek provedené operace je roven nule (s výjimkami, které si probereme dále).

S (sign) - indikátor znaménka (plus, mínus) - indikuje, zda nejvyšší bit bajtu je ve stavu log.1 (pak je i S roven 1) a naopak. Vztahuje se ke dvojkově komplementárním číslům (o nich později).

P|V (Parity|Overflow) - indikátor se dvěma funkcemi. Parita se testuje po provedení logické operace (AND,OR,XOR) a instrukcích rotace - indikuje se, zda počet bitů log.1 (tedy i log.0) v bajtu je sudý nebo lichý. U jiných operací Overflow (přetečení) indikuje změnu polarity dvojkově komplementárních čísel.

N (Add|Subtract) - slouží pro interní indikaci odečítání.

U volání s podmínkou můžeme zjišťovat oba stavy indikátorů:

```
CALL Z,adr. (je Z 1?)          CALL NZ,adr. (je Z 0?)
CALL C,adr. (je CY 1?)         CALL NC,adr. (je CY 0?) apod.
```

V případě, že je podmínka splněna, reg.PC se naplní číslem adresy (adr.) a naše lokomotiva "přefrčí" na tuto adresu, od níž bude pokračovat v "luštění" pražců (bajtů na adresách) dál. Zároveň s tím se do zásobníku (adresovaného registrem SP) "na čas odloží" číslo adresy, které je o 1 vyšší, než je adresa posledního bajtu tříbajtové adresy volání. Této adrese se říká adresa návratu - to je zvláštnost instrukcí volání. Jsou totiž spjaty s instrukcemi návratů - RET (zkratka pro return). Jakmile naše lokomotiva narazí na tuto instrukci, náš svědomitý výhybkář převede do reg.PC onu adresu návratu ze zásobníku a lokomotiva na ni ihned přejede - vrátí se těsně za instrukci volání, aby odtud pokračovala zase dál.

Stojí-li instrukce RET jen sama o sobě, provede se návrat vždy. Ale stejně tak jako instrukce volání, mohou i instrukce návratu mít připojeny tytéž podmínky (RET Z, RET NZ atd.). T.zn., že návrat z takové instrukce se provede jen tehdy, je-li podmínka splněna.

Představte si, že mikroprocesor "najede" v programu na tyto instrukce:

```
LD B,1        ;ulož do reg.BC číslo 1
DEC B         ;sniž obsah reg.BC o 1
CALL Z,aLfa
```

Co myslíte, odbočí naše lokomotiva na adresu zde symbolicky zvanou alfa?

Skotačivé programy

Odpověď na otázku zní - ano, lokomotiva odbočí na adresu alfa, protože instrukcí DEC B se obsah reg.B sníží na nulu, což signalizuje indikátor nuly reg.F tak, že jeho bit se převrátí do stavu log.1. Podmínka instrukce volání CALL Z,alfa je tak splněna a odskok se provede.

Z několika předchozích částí kapitoly vyplývá jeden z hlavních požadavků na tvorbu programů - strukturalizace. Dosahujeme jí tak, že celý program rozkládáme do jednotlivých podprogramů, které pak do činnosti zapojujeme vždy, když potřebujeme, aby byla vykonána operace, kterou ten který podprogram "umí".

Např. chceme, aby se po stisku tlačítka počítačové klávesnice pohnul nějaký bod na obrazovce monitoru o stanovený "kousek" doprava. Jak už asi tušíte, musíme do programu zařadit test, který zjišťuje, zda je tlačítko stisknuto, či nikoli. Jakmile je stisknuto, aktivuje se instrukce volání (CALL), která převede řízení programu na 1.adresu podprogramu, který umí posunout bod na obrazovce o "kousek" doprava. Poté, co mikroprocesorová lokomotiva projede celým podprogramem, narazí na instrukci návratu (RET), vrátí se tam, odkud předtím odbočila a pokračuje v jízdě dál, než opět narazí na nějakou (podmíněnou či nepodmíněnou) instrukci, která ji opět převede na nějaký podprogram...a tak stále dokola. Až dokud celý program nevykoná, co od něj požadujeme, ev.dokud jej nezastavíme nebo nevymažeme.

Vedle instrukcí CALL a jejich "bratříčka" RET, je v programu možno provádět odskoky na jiné adresy skokovými instrukcemi. Jsou jich dva základní druhy - skoky přímé (JP - jump), s tzv. absolutní adresou, a relativní (JR - jump relative). Podobně jako předchozí, mohou i tyto instrukce obsahovat podmínku, při jejímž splnění je skok proveden na danou adresu (JP NZ,adr. apod.). Ale na rozdíl od nich neukládají žádnou adresu návratu do zásobníku. Jinými slovy - je to prostý skok bez možnosti návratu pomocí instrukce RET. U relativních skoků je věc trochu komplikovanější. V jejich instrukcích není uvedena přímá adresa, ale jen jednobajtové číslo, které uvádí, o kolik adres vpřed či vzad (v intervalu plus 127, mínus 128; tedy v rozsahu oněch počítačových 256 - včetně nuly) má lokomotiva odbočit. Poslední speciální relativní skokovou a trošku komplikovanější instrukcí mikroprocesoru Z80 je DJNZ.

Tak jsme dospěli ke zjištění, že program je záležitostí velmi skotačivou. V profesionálních programech jsou desítky a stovky odskoků, z nichž prakticky všechny jsou v průběhu programu aktivovány zdaleka ne jednou, ale stokrát, tisíckrát a vícekrát. Tak naše mikroprocesorová lokomotiva - podle toho, jak jí velí řada skokových instrukcí a připojených podmínek - projíždí ohromnou rychlostí jednotlivými podprogramy, vrací se, odbočuje a najíždí tak, jak ji kolej programu a z ní sbíraná data (i data vnější) vedou.

Vše, co jsme si dosud řekli o programových instrukcích jazyka assembler (a lehce i o strojovém kódu), platí v nezmenšené míře i v ostatních počítačových jazycích. Stukturování programu v jakémkoli z nich je jedním z nejdůležitějších požadavků při jeho tvorbě. Podle struktury, jakou má program, poznáme i to, zda jeho autor je dobrým či špatným programátorem. Přehledný, chytrými nápady "vyšperkovaný" program nejen že funguje jak náleží, ale poskytuje i možnost jeho snadných úprav a obměn pro jeho další, v něčem odlišné aplikace. S programem, který je nerozluštitelnou motanicí, nic moc nenaděláme.

Nejsoučasnější trend, jakým se tvorba programů ubírá, reprezentují programy tak inteligentně vystavěné a strukturované, že (pomocí velmi jednoduchého ovládání) svému uživateli dokonce poskytují možnost vytvářet určitou vlastní "nadřazenou" strukturu sestav údajů a dat, které do programu potřebuje zařadit, resp. je dále zpracovávat. Výhledem do nedaleké budoucnosti se ukazují být programy, které se dokáží samy učit ze zkušeností, které během dialogu se svým uživatelem i svým okolím postupně získávají. První náznak vzniku něčeho, čemu můžeme už dost oprávněně říkat "umělá" inteligence.

Jako každý um, i ten softwarový má své základy, jejichž část jsme již poznali. Abychom však mohli sestavit alespoň krátký a smysluplný program, musíme se obeznámit ještě s několika instrukcemi, které se na programové koleji objevují mezi jednotlivými skoky.

Skok do tmy

Jak už víme, každý počítačový program je protkán skokovými instrukcemi, které převádějí řízení programu do řady jednotlivých podprogramů, jež tvoří vlastní strukturu programu. Název této kapitoly byl zvolen se žertovnou ironií, která při vlastním programování už tak žertovné výsledky nepřináší. Narazíte-li někdy na podmračeného, nevrlého člověka, obklopeného haldou papírů a knih, mezi nimiž se nevinně tyčí monitor počítače, můžete si být jisti, že se jedná o programátora, který právě takový skok do tmy absolvoval. Jeho podprogram - proboha, proč jen?! - se nechová, jak by měl; dokonce ničí všechno, co dosud po dlouhé týdny vytvářel. Nezbývá, než program trpělivě, krok za krokem, **odladit** - jedna z nejdůležitějších činností programátora, která mnohdy zabere mnohem více času, než vytvoření první podoby programu. I to je neodmyslitelnou součástí programování.

Pomyslný skok do tmy absolvujete i vy. V příští části na vás čeká úkol - doplnit chybějící část programu v jazyce assembler. Abyste k němu měli vše, co budete potřebovat, rozšíříme si ještě trochu naše dosavadní poznatky o assembleru.

Prosté instrukce pro sečítání a odečítání mají tuto mnemoniku: ADD A,r a SUB r, kde r je registr, jehož obsah přičítáme (ADD) k nebo odečítáme (SUB) od akumulátoru, tedy registru A. Místo r lze rovněž použít přímo určené číslo v intervalu 0-255. Je-li obsahem reg.A třeba číslo 8 a reg.C 5, pak po provedení instrukce ADD A,C bude obsah akumulátoru 13, po SUB C bude v reg.A číslo 3. Pokud by ovšem součet nebo odečet překročil interval 0-255 z kterékoli strany, dojde k přeplnění registru, které je indikováno stavovým indikátorem CY (carry) registru F. Je-li v reg.A 255 a v reg.L 2, bude po operaci ADD A,L v reg.A číslo 1 (reg.A "přetekl horem" a začíná opět počítat od nuly nahoru). Při operaci SUB D, kde v reg.A je 1 a v reg.D jsou 3, bude výsledek v reg.A 254 (reg.A "přetekl dolem" a počítá opět od 255 dolů). CY je v tomto případě vždy ve stavu log.1. Toho můžeme využít v testech výsledků operací vztahujících se (jen) k reg.A v instrukcích skoků a volání. Názorný příklad pro pochopení "jednobajtové aritmetiky přetečení":

```
         LD B,20          ;Uložení č.20 do reg.B
         LD A,10          ;Uložení č.10 do reg.A
         SUB B            ;odečet 10-20; v reg.A je 246
         JR C,součet      ;Je CY=1? ANO; skok na adr.součet
         RET              ;NE; návrat zpět k adrese volání
součet:  ADD A,B          ;246+20; v reg.A je opět 10, CY=1
         CALL podprogram  ;Volání adr.podprogram
```

Je-li číslo v reg.B větší než v reg.A, bude CY vždy ve stavu log.1. Instrukce RET se provede jen tehdy, bude-li CY=0, tedy číslo v reg.B bude rovno nebo menší než číslo v reg.A. Instrukce SUB má své dvojče CP (compare - porovnej), které se liší je tím, že nemění obsah akumulátoru.

Divide et impera

Latinské Rozděl a panuj se přímo vztahuje k obsahu našeho úkolu. Je jím doplnění chybějící části programu, který bude bezchybně dělit čísla v rozsahu 0-255 tak, že se stanete jeho skutečnými pány. Při dokonalé znalosti jazyka assembler a strojového kódu by bylo možno program napsat jinak. Pro splnění úkolu však stačí, když použijete vědomostí, které jste dosud načerpali.

Dále uvedený assemblerový program dělí čísla v intervalu 0-255 čísly v intervalu 1-255 (dělení nulou je nepřípustné). V případě, že výsledkem dělení není celé číslo, musí program určit zbytek dělení. Další vstupní podmínkou, kterou si určíme, je, že číslo dělené (dělenec) nesmí být menší než číslo, jímž dělíme (dělitel) - program by pak byl mnohem komplikovanější. Dělence si označíme symbolem "ec", dělitele "el". Nakonec si musíme uvědomit, že dělení je vlastně odečítání. Tak 31:5 je totéž, co šestinásobný odečet čísla 5 od 30 se zůstatkem 1.

```
Start:     LD C,00          ;Vynulování reg.C pro ukládání výsledku
           LD A,el          ;Uložení dělitele do reg.A
           CP 00            ;Je dělitel roven 0?
           RET Z            ;ANO, pak návrat zpět (0 je nepřípustná)
           LD B,A           ;Převod obsahu reg.A (dělitele) do reg.B
           LD A,ec          ;Uložení dělence do reg.A
           CP B             ;Porovnání el s ec - je el větší než ec?
           RET C            ;ANO...návrat zpět - ec nesmí být menší!
           CP 00            ;Je ec=0? Když ANO, skok na adresu
           JR Z,Výsledek    ;Výsledek (reg.C zůstane nulový)
Odečet:    INC C            ;Zvýšení obsahu reg.C o 1 (počet odečtů)
           SUB B            ;Provedení odečtu dělitele od dělence
           JR Z,Výsledek    ;Je-li výsledek 0, skok na adr.Výsledek,
           JR C,Zůstatek    ;je-li výsledek "záporný", na adr.Zůst.,
           JR Odečet        ;jinak pokračuj v odečítání (přímý skok)
Zůstatek:  ------------------------------------------------------------
           ------------- je vaším úkolem; viz text --------------
           ------------------------------------------------------------
Výsledek:  CALL Tisk        ;Volání podprogramu pro tisk výsledku
           RET              ;Návrat zpět
```

Celý tento podprogram voláme (imaginárně) odněkud z programu instrukcí CALL Start. Proto jsou v něm i instrukce návratů RET Z a RET C (podmíněné stavy indikátorů Z a CY) a RET (přímá). V instrukci INC C postupně narůstá počet provedených odečtů (INC r zvyšuje obsah registru r o 1; opakem je instrukce DEC r, kde je obsah reg.r snižován o 1). Stane-li se, že odečet (samozřejmě poslední v řadě) bude mít výsledek 0 (nulový indikátor Z=1 při testu JR Z,výsledek), znamená to, že dělení proběhlo beze zbytku s výsledkem (celým číslem) v reg.C. Není-li však při posledním odečtu Z=1, pak dochází k přetečení reg.A, které se projeví změnou stavu indikátoru na CY=1 s odskokem na adresu Zůstatek, kde instrukce zjistí hodnotu zůstatku. **A právě tuto část programu jsme před vámi "skryli". Vaším úkolem je doplnit jej na tomto místě tak, aby určil velikost zůstatku dělení a samozřejmě aby výsledek dělení sám byl bezchybný.** V případě váhavosti doporučujeme zahledět se především do minulé části.

Správné řešení najdete na konci 2.kapitoly. Doporučujeme vám, abyste se však napřed snažili úlohu vyřešit sami. Každá snaha, každé zamyšlení někam vedou a obohacují o drobné poznatky, aniž si to třeba sami uvědomíte. Jedině tak se vám podaří nejen naučit, ale i aplikovat obsah celé učebnice.

Závěrem

Pokud jste úkol vyřešili správně, máte velké dispozice pro zvládnutí jakéhokoli oboru, v němž se nelze obejít bez logického myšlení. Můžete se stát i dobrým programátorem. Pokud jste úlohu nezvládli, buď se nedokážete soustředit (vlivem okolí či nemáte schopnost koncentrace na tento typ studijní látky), nebo vám to prostě jen pomaleji myslí. V prvním případě se budete s učebnicí dost trápit a bylo by možná lepší se zamyslet nad tím, zda byste své schopnosti neměli rozvíjet jiným směrem. Ve druhém případě se pusťte do látky se vší vervou a zůstaňte věrný své pomalosti, která neznamená nějaké minus - o to lépe si naučené budete pamatovat. Počítejte však s tím, že vlastní tvorba programu vám většinou zabere víc času.

Učebnice je sestavena tak, že každá instrukce je probrána velice zevrubně, s mnoha příklady jejího užití. Držte se té zásady, že dokud funkci jedné instrukce nepochopíte, nepouštějte se do dalších. Pokud máte počítač, sežeňte si generátor a monitor strojového kódu (pro ZX Spectrum např.GENS3 a MONS3, pro Amstrad GENA a MONA apod.) a jednotlivé instrukce si hned vyzkoušejte "v terénu". Ohromně vám to usnadní a urychlí celé studium assembleru a strojového kódu.

Na počítači sestavujte i krátké vlastní programy, vytvořené z instrukcí, které jste se zatím naučili. Přitom vydatně využívejte jedné z funkcí monitoru strojového kódu, která se jmenuje krokování (ovládání různých generátorů a monitorů se v lecčems liší, musíte se je naučit podle manuálu). Krokováním budete programem postupovat od jedné instrukce ke druhé, přičemž na obrazovce uvidíte všechny změny, které provedení každé instrukce přinese (ve stavových indikátorech, registrech i na jednotlivých adresách). Když vám váš prográmek bude fungovat a budete jeho funkci rozumět, budete mít vedle přirozené radosti o to větší chuť do další práce.

****

KAPITOLA 2

Programovací jazyky a způsoby zápisu

****

V úvodní kapitole jste měli možnost stručně se obeznámit s binárním, hexadecimálním i assemblerovým zápisem. Jejich porovnáním se budeme zabývat i v této kapitole.

Binární kód

Tento kód je skutečným jazykem strojového kódu. Vše ostatní jsou více či méně komplikované odvozeniny, které vyžadují kompilaci nebo interpretaci, jednoduše řečeno převod (překlad) na binární kód. Pokud byste tedy zvládli programování v tomto kódu, nemuseli byste se v podstatě zabývat žádnou jeho odvozeninou. Jenže...to byste museli mít v hlavě aspoň jeden mikroprocesor, nebo se vyznačovat schopností zapamatovat si obsah telefonního seznamu. Protože nic z toho zdrcující většina obyvatel Země nemá, bylo nutno přikročit k takovým formám zápisu, které jsou bližší lidské, leč vzdálenější computerové logice. Čím blíže je zápis lidské logice, tím rychleji může programátor sestavit fungující program. Ale za všechno se platí. V tomto případě je základní investicí čas. Čím jasnější řeč, hutnější forma zápisu, tím déle bude běh programu a zpracování dat trvat. Proto je tak zdrcující rozdíl mezi rychlostí, s jakou pracuje program ve strojovém kódu vůči programu zapsaném např. v Basicu.

Ukažme si na příkladu, jak vypadá zápis programu v binárním kódu. Jedná se o velmi prostý "kousek" programu, který sečítá dvě čísla z adres 0160H a 0161H a výsledek ukládá na adresu 0162H (program obsahuje 11 bajtů po osmi bitech):

```
00111010
01100000
00000001
01000111
00111010
01100001
00000001
10000000
00110010
01100010
00000001
```

Počítač by se nad takovým "počteníčkem" jen rozplýval. Vy už asi o mnoho méně. (Vysvětlení číselného vyjádření binárního kódu je v 1.kapitole).

Hexadecimální kód

Cesky řečeno se jedná o kód v šestnáctkové číselné soustavě (binární je dvojková). Hexadecimální kód není o moc srozumitelnější než binární. Binární tvar bajtu vyjádřený osmi číslicemi 0 a 1 nabývá formu dvoučíselnou, přičemž každá z obou hexadecimálních číslic je v intervalu 16ti po sobě jdoucích čísel 0-F (0123456789ABCDEF) - decimálně 0-15. Počítání v tomto kódu je pro nás trochu nezvyklé, protože jsme naučeni počítat v soustavě desítkové. Na věci však není nic složitého.

Chceme-li např. zjistit, co je 23 v zápise hexadecimálním, budeme postupovat podobně jako v případě čísla decimálního. 23 decimálně je 2*10+3. 23 hexadecimálně je 2*16+3=35 decimálně. Tedy místo deseti násobíme (resp. dělíme) šestnácti. U vícemístných čísel dodržíme postup podobný, na jaký jsme zvyklí z desítkové soustavy. Podle "pořadového místa" číslice v čísle tuto číslici násobíme (vzato "odzadu") číslem 1, 16, 16*16, 16*16*16, 16*16*16*16 atd. Podobně v decimální soustavě je to: 1, 10, 10*10, 10*10*10, 10*10*10*10 atd. Tak 2BE hexadecimálně je 2*256+11*16+14=702 decimálně.

Trochu problémy možná bude někomu zpočátku přinášet nutnost uvědomit si, že nula je též jednotkou číselné soustavy. Proto FFH (připojený symbol H označuje hexadecimální číselný zápis) je 15*16+15=255, ale pro počítač (i pro vás!) je to 256 číselných jednotek, protože jako číselnou jednotku zahrnuje i nulu - tedy 256 čísel v intervalu 0-255 (to je zároveň číselný rozsah jednoho bajtu). Jde o uvědomění si často zakořeněného zlozvyku chápat i jednotky desítkové soustavy v rozsahu 1-10, a nikoli 0-9. Na to vždy pamatujte! Vyhnete se tak naprosto zbytečným a triviálním chybám, které vás při jejich mylném hledání v

programu mohou stát příliš mnoho času a nervů (budete totiž hledat chybu v programu, zatímco chyba bude ve vás - do té doby, než si ji sami uvědomíte). Proto také **adresovací rozsah osmibitového mikroprocesoru zahrnuje 65536 adres, ale nejvyšší adresa je 65535 (FFFFH), protože nejnižší je nula (0000H)!**

Výše uvedený binární zápis 11ti bajtů bude ve formě hexadecimální vypadat takto:

3A
60
01
47
3A
61
01
80
32
62
01

Žádné počteníčko ani pro počítač, ani pro člověka. Počítač pro vyluštění těchto čísel potřebuje pomocný program pro jejich převod do binárního kódu. Pokud jde o člověka, jen největší fanatik programování ve strojovém kódu je schopen si zapamatovat hexadecimální tvar všech instrukcí Z80. A to jsme teprve u osmibitového mikroprocesoru - ten má jen kolem 700 základních instrukcí (ve skutečnosti je jich mnohem více; běžné monitory a generátory strojového kódu je však nezahrnují). Ale představte si instrukční soubor takového 16tibitového, nebo dokonce 32tibitového mikroprocesoru! Záplava hexadecimálních kódů (spolu s jejich kombinacemi) naroste a se schopnostmi své paměti jsme pak už zcela vedle. Proto byl pro programování ve strojovém kódu sestaven srozumitelný soubor ASSEMBLERových instrukcí (mnemonický kód), které mají ke strojovému kódu nejblíž.

Mnemonický kód

Jistě si ze svých školních let vzpomenete na nějakou mnemotechnickou pomůcku, kterou se vám učitelé snažili pomoci při zapamatování si nějaké "složitosti". Třeba EKOKRPKA - pomůcka pro zapamatování si schématické struktury vývoje dramatu ("flow chart of drama"): Expozice, KOlize, KRize, Peripetie, KAtastrofa (aneb jak Vávra ke své poslední kávě od Maryšky přišel). Ale přejděme od jednoúčelových nápojů ke smysluplnější činnosti.

Mnemonické písmo předcházelo písmu současnému a bylo vlastně kódovaným sdělením pomocí zářezů do různých materiálů, uzlů na provázku a pod. Mnemonika i mnemotechnika mají mnoho společného. Jak vidět, kódování holdovali už naši dávní předkové.

Dovedně sestavená mnemonika strojového kódu nám ve výsledku podává vyjádření všech instrukcí mikroprocesoru ve velmi účelné, triviální a snadno zapamatovatelné (srozumitelné) formě. Její naučení se "nazpaměť" zabere méně času než šprtání syntaxe jakéhokoli vyššího programovacího jazyku, Basicem počínaje a třeba Adou konče.

Musíme si však opět uvědomit, že jestli jsme se vlastní řeči počítače vzdálili už hexadecimálním kódem, pak mnemonika strojového kódu nás zavádí ještě dál. Nic strašného se však neděje, protože mnemonické kódy, tvořící programovací jazyk assembler, budeme používat jen při vlastním programování za použití pomocného systémového programu zvaného generátor strojového kódu (někdy též assembler). Do něj ukládáme assemblerové instrukce jako zdrojový kód (source code). Jak název sám napovídá, umí generátor převést náš assemblerový program přímo do binárního kódu, obrazně zvaného též cílový kód (object code), v němž už program zůstane k dalšímu použití, včetně možnosti jeho dodatečných úprav. Podívejme se nyní, jak bude vypadat zápis výše uvedených 11ti bajtů v assembleru:

```
LD      A,(0160H)
LD      B,A
LD      A,(0161H)
ADD     A,B
LD      (0162H),A
```

Počítač si nepočte, my však ano (i když možná až o něco později). Každopádně i absolutní laik musí poznat, že v tomto zápisu se ze zadané úlohy čitelně objevují "aspoň" čísla adres. Zatímco z předchozích dvou zápisů nikdo příliš nezmoudří. To, že assemblerových instrukcí není zrovna 11, je dáno tím, že jednotlivé instrukce obsahují 1 až 4 bajty (podle jejich typu). Samozřejmě si je v textu učebnice všechny pečlivě probereme. Ukázky programů budou zapisovány v této formě:

Adresa paměti	hexadec. kód	assembler	komentář
0150	3A6001	LD A,(0160H)	;Převod obsahu adr.0160 do reg.A
0153	47	LD B,A	; " " reg.A do reg.B
0154	3A6101	LD A,(0161H)	; " " adr.0161 do reg.A
0157	80	ADD A,B	;Součet " reg.A a reg.B
0158	326201	LD (0162H),A	;Uložení výsledku v reg.A na ;adresu 0162

Uvedená forma zápisu vám poskytne veškerou potřebnou orientaci stran umístění instrukcí v paměti, umožní vám je do ní ukládat jako hexadecimální kód (s pomocí monitoru strojového kódu) nebo jako assemblerové instrukce (s pomocí generátoru strojového kódu) a nakonec vám připojenými komentáři vysvětlí, jak program pracuje. Komentář je u každého assemblerového programu velmi cenný, protože bez něj se funkce, struktura a průběh programu zjišťuje jen velmi obtížně. **V učebnici uvedené adresy uložení jednotlivých instrukcí a dat jsou užitelné u počítačů, které nemají pevně instalovanou paměť ROM na spodních adresách. Čísla adres však nejsou nikterak směrodatná. Program můžete v paměti RAM umístit kamkoli libo. Musíte však dát pozor na to, abyste v případě výskytu instrukcí volání, přímých skoků atd. nezapomněli provést odpovídající adresovou, resp. datovou transpozici.**

V dalším textu si zavedeme ještě jednu zkratku. Slovo hexadecimální zkrátíme na HD. HD kód bude tedy znamenat hexadecimální kód.

Nejen v této učebnici, ale v computerové literatuře celého světa je zápis mnemoniky mikroprocesoru Z80 naprosto shodný. Pochází přímo od jeho autora-výrobce - firmy Zilog.

Vyšší programovací jazyky

Ty jsou dalším krokem, který nás vzdaluje od binárního kódu. Jejich význam tkví především v tom, že programátorovi umožňují plně se soustředit na řešení daného problému, aniž by se musel starat o to, jak se "daří" mikroprocesoru, resp.jeho registrům, co zrovna máme v zásobníku, co nám říkají jednotlivé linky sběrnic, co na ně kdy poslat, odkud co dekódovat atd. Programy jako FORTRAN, COBOL, PL/1, PASCAL, ALGOL atd. obsahují kompilátor, který nakonec celý program převede z vyššího jazyku do binárního (strojového) kódu.

Jazyky jako třeba Basic kompilátor nemají. Jsou vybaveny interpretorem, který je neustále "v akci". Nepřeloží tedy najednou celý program do binárního kódu, ale vykonává postupně a stále dokola jednotlivé příkazy programu za pomoci jejich interpretace rutinami strojového kódu, obsaženými v interpretoru. Jeho kvality v návaznosti na hardwarové provedení počítače přímo ovlivňují "svižnost" tohoto typu programů.

Všechny jazyky mají svá plus i mínus. Kompilátory nikdy nepřevedou program vyššího jazyku na ideální strukturu binárních instrukcí. Ale sestavení programu je rychlejší. Rovněž jsou vybaveny řadou testovacích funkcí, které nás informují o tom, jakou (případně i kde) jsme udělali syntaktickou nebo jinou chybu. Po stránce rychlosti zpracování je na tom nejhůř Basic a spol. Jejich interpretory jsou velkými časovými brzdami při provádění programových příkazů. Další společnou nevýhodou je spousta nářečí vyšších programovacích jazyků, včetně rozdílností v jejich kompilaci a interpretaci. Určité snahy o kompatibilitu byly přehlušeny prudkým vývojem programovacích jazyků, takže zmíněná džungle ještě zhoustla. Přesto však určité standardy u některých jazyků existují, i když se pochopitelně stále vyvíjejí, tedy i mění.

Výhodou assembleru je možnost přímého využití všech softwarem ovladatelných hardwarových funkcí počítače, tedy ohromná efektivita běhu programu. K tomu se řadí i nejefektivnější využití rozsahu paměti, s čímž souvisí bezkonkurenční rychlost jeho funkcí. Nevýhodou je obtížnost, tedy i zdlouhavost vytváření assemblerových programů. Jedním z nejvyšších požadavků kladených na programy obecně je jejich strukturálnost, tedy i přehlednost, nezbytná nejen v případě potřebné úpravy programů při různých aplikacích. Vytvoření struktury assemblerového programu je jeden z největších programátorských oříšků - stále musíme mít na zřeteli řadu různých průběhů a změn. Při špatně vytvořené struktuře a jejím ošetření jsou následky katastrofální. Jednou z nejčastějších chyb je ztráta vlády nad adresami návratů ukládaných do zásobníku. Podobně negativní působení má proběhnutí rutinami, během nějž se poruší sled a smysl ukládání a odebírání obsahů registrů, resp. dojde k jejich promíchání se zmíněnými adresami návratů. Z toho důvodu je ladění assemblerových programů jednou z časově i duševně

nejnáročnějších činností programátora. Dobře vytvořený assemblerový program je však v porovnání s programy sestavenými v jakémkoli vyšším programovacím jazyku bez konkurence.

Ke všemu, tedy i k programování, je třeba přistupovat s nadhledem a rozvahou. Prvotní otázka, na niž si před zasednutím k počítači musíme odpovědět, je, zda z hlediska účelu a funkce programu má smysl sestavit jej celý v assembleru. V mnoha případech je odpověď naprosto jednoznačně záporná. Je tomu tak především v případech tvorby matematických programů. Vyššími programovacími jazyky můžeme simulovat složité matematické funkce neporovnatelně snadněji než assemblerem. Na druhou stranu u aplikací typu slovního procesoru, datových bází, zpracování obrazu apod. bychom měli jednoznačně dát přednost assembleru právě pro jeho rychlost, která je u tohoto typu programů požadavkem nejvyšším.

A tak nám jako vždy zbývá zlatá střední cesta - jakýkoli vyšší programovací jazyk doplňovat rutinami strojového kódu, kdykoli se to ukáže být účelným, resp. nezbytným. A naopak.

Chceme-li se však stát dobrými programátory i umět jakýkoli program upravit tak, aby sloužil našim požadavkům, pak se bez znalosti binárního kódu a assembleru vůbec neobejdeme. A tak vzhůru do tajemné říše instrukčního souboru dnes nejrozšířenějšího osmibitového mikroprocesoru Z80!

Odhoďte svůj obav širák v dál
strhejte roušky tajemství

jednu
za
druhou

...za tou poslední vás čeká
binární slast -
assembleru nekonečný ráj!

Správné řešení úlohy z 1.kapitoly:

```
Zůstatek: DEC C       ;Napřed vyrovnej "přetečený" výsledek
          ADD A,B     ;Zjisti zůstatek
```

****

KAPITOLA 3

Registry, paměť a datový přenos

****

V patetickém paverší závěru minulé kapitoly by se slušelo slovo slast nahradit slovem strast v množném čísle. Bylo by to pravdě mnohem blíž. Na druhou stranu nelze zakrývat, že radost nad vytvořením perfektně fungujícího programu ve strojovém kódu je neporovnatelná s nadšením plynoucím z obstojně prorostlé basicové kýty. Rozdíl je dán tím, že vyšší programovací jazyk za vás v rámci svých schopností udělá vše, co se týče jeho styku s vnitřnostmi počítače. Tak sice můžete leccos)ale nikoli vše(naprogramovat, avšak počítač sám zůstane pro vás španělskou vesnicí. Při programování v assembleru jste nuceni jít věci)především architektuře hardwaru(až na kloub. Proto neleňte a shánějte všechny možné informace o svém počítači, obklopte se literaturou, časopisy, zprávami. Když se někdy při programování dostanete do úzkých, ať máte odpověď vždy někde na dosah ruky. Když chcete nakreslit tečku na obrazovce, vyšší programovací jazyk má pro to ve své výbavě syntakticky jednoduchý příkaz, který je při provedení ovšem složitost sama. Strojovým kódem však musíte onu tečku binárně naprogramovat a vědět, kam, jak a kudy ji do hardwaru poslat, aby se nakonec na obrazovce objevila tam, kde ji chcete mít.

Instrukční soubor Z80

Mnemonický kód obsahuje operační kód, kterým je vyjádřena specifická operace, již mikroprocesor provádí. Osmi bity jednoho bajtu můžeme vyjádřit až 256 různých jednobajtových operačních kódů. Jsou uvedeny spíše pro zajímavost, protože pro mikropočítače prakticky neexistují generátory strojového kódu, pomocí nichž by bylo možno programovat v binárním kódu; i když data v binární formě do mnoha z nich zapisovat lze. Znalost sestavy binárního kódu je však možno využít pro modifikaci programových instrukcí přímo v programu.

Pokud někdo z vás pracoval s instrukčním souborem mikroprocesoru 8080, zjistí, že Z80 je obohacen o instrukce EXX, JR (relativní skoky) a řadu čtyřbajtových instrukcí, které začínají HD kódy: CB, DD, ED nebo FD.

První dva bity binárního kódu určují typ instrukce. U jednobajtových:

01 - LD instrukce
10 - aritmetické a logické instrukce (DAA,OR,AND,XOR)
00 - relativní skoky)JR(
11 - přímé skoky, volání a návraty (JP,CALL,RET)

Aby mikroprocesor poznal, jakou instrukci má vykonat, nejdříve dekóduje první dva bity binárního kódu operačního kódu. Následujících 6 bitů jednobajtových instrukcí je kódem dvou registrů - co registr, to 3 bity:

Registr	Binární kód
B	000
C	001
D	010
E	011
H	100
L	101
(HL)	110
A	111

Zvláštní postavení zde má (HL), což je obsah adresy v paměti, určené obsahem párového registru HL. Značení (HL) má i jiný význam, k němuž se dostaneme později. Obdrží-li mikroprocesor binární kód ve tvaru 01111010, znamená to:

```
0 1 ........... LD instrukce
    1 1 1 ..... A je registr určení
          0 1 0 D je zdrojový registr
```

Uvedený binární kód má mnemoniku LD A,D. U aritmetických a logických funkcí 3.,4. a 5. bit určuje, k jakému registru se instrukce vztahuje. Totéž platí pro instrukce začínající 00.

CB instrukce

Obsahují 2 bajty. Jim - i instrukcím s prvním bajtem DD, FD a ED - se říká multibajtové. Např. instrukce BIT 2,C má dva bajty operačního kódu: CB 51. Struktura druhého bajtu (řazená podle svých prvních dvou bitů) je tato:

00 - rotační nebo shiftová instrukce; další tři bity jsou kódem registru
01 - instrukce BIT; další tři bity určují registr, poslední tři adresovaný bit
10 - instrukce RES; další bity s významem jako u BIT
11 - instrukce SET; " "

DD a FD instrukce

Mohou mít dvou nebo tříbajtový operační kód. Instrukce DD se vztahují k registru IX, FD k IY. Všechny tříbajtové operační kódy Z80 mají následující strukturu:

bajt 1 - DD nebo FD v závislosti na registru (IX nebo IY)
bajt 2 - CB
bajt 3 - odchylka (angl.displacement) - není součástí oper.kódu
bajt 4 - první dva bity určují rotaci nebo shift, BIT, RES nebo SET jako u CB instrukcí výše. Další tři bity jsou vždy 110. Zbývající bity určují typ rotace nebo shiftu nebo čísla bitu jako v CB instrukcích.

ED instrukce

Všechny mají dvoubajtový operační kód.

Adresovací módy Z80

Takřka všechny instrukce pracují s daty, umisťovanými buď do registrů Z80 nebo na jednotlivé adresy paměti, či je mohou vysílat na periférie, nebo data z nich přijímat (I/O porty - Input/Output ports). Termín adresovací mód se vztahuje k metodě, jejíž pomocí instrukce data zpracovává. K tomu si víc povíme o něco dál. Adresovací módy jsou velmi důležitou částí vedoucí k pochopení funkce mikroprocesoru ve vztahu k jeho okolí. Proto spíš než samotné instrukce se snažte pochopit princip jejich funkce.

Jednobajtový přenos obsahu registru (adresy paměti)
Registrový adresovací mód LD u,z

V instrukčním souboru Z80 je 63 různých instrukcí tohoto typu. Každá z nich má mnemoniku LD u,z.

u - registr určení
z - zdrojový registr

Instrukční kódy v HD tvaru jsou v intervalu 40-7F (s jedinou výjimkou - 76, což je instrukce HALT). Osmibitová forma instrukce je:

0 1 u u u z z z

Binární kód registrů je týž, jak uvedeno o něco výše. Nyní si můžeme definovat registrový adresovací mód - je to technika užívající skupin bitů v instrukčním kódu Z80 tak, že určují, které registry budou instrukcí ovlivněny či do ní zapojeny.

LD C,B	přenos obsahu registru B do registru C
LD A,C	" " " C " A
LD D,E	" " " E " D
LD A,H	" " " H " A
LD L,A	" " " A " L
LD (HL),A	" " " A do adresy paměti určené obsahem registru HL
LD A,(HL)	" " adresy v paměti určené obsahem registru HL do registru A

Přenosy do a z míst v paměti budou podrobněji vysvětleny dále. Je nutno si uvědomit, že po tomto přenosu obsah registru zdrojového zůstává nezměněn, zatímco obsah registru určení se změní (jeho obsah se stane kopií obsahu zdrojového registru).

Bezprostřední přenos dat do registru
LD r,NN

Anglicky se tento typ přenosu jmenuje Load immediate to register, takže gramaticky přesnější by bylo říci uložení. Jedná se o "otisk" obsahu (ať už registrů nebo dat) probíhající po 8 linkách, takže slovo přenos je mnohem výstižnější.

Mnemonika tohoto přenosu je LD r,NN, přičemž r je registr nebo adresa paměti (HL), NN znamená číslo v rozsahu 00H-FFH. Binární kód instrukce lze schematicky vyjádřit takto:

bajt 1 - 0 0 u u u 1 1 0
bajt 2 - datový bajt (přenášené číslo)

Písmena uuu znamenají registr určení - jejich binární kódy už známe.

Na tomto místě bude vhodné seznámit se s tím, co to znamená doba trvání provedení jedné instrukce a proč se v tabulkách tato doba uvádí. Čas potřebný pro provedení instrukce je přímo závislý na frekvenci, s níž pracuje mikroprocesor počítače. Z80 se vyskytuje v několika provedeních; podle typu (i schopností výrobce) může pracovat v kmitočtu 2-4 MHz. Jak si v tabulkách můžete povšimnout, mnohé instrukce mají shodný počet kmitů pro své provedení. Je to dáno časovými průběhy operací, které v mikroprocesoru neustále probíhají (o nich později). Instrukce LD r,NN vyžaduje 7 kmitů, což odpovídá času 2,8 mikrosekundy (při 4 MHz).

Všechny registrové i datové přenosy Z80 se dějí paralelně - všech 8 bitů je převedeno současně po 8 linkách. Speciální podmínky vyžaduje pouze přenos dat z místa nebo na místo paměti (HL) - o nich dále.

Nepřímý registrový přenos s akumulátorem
LD A,(rr); LD (rr),A

Jedná se o adresovací mód, při němž párový registr BC, DE nebo HL obsahuje adresu paměti (číslo v rozsahu 0000H-FFFFH), z níž nebo na níž je uskutečněn přenos (mezi ní a akumulátorem). Tak např. LD A,(DE) znamená, že obsah adresy (jeden bajt) určené číslem v párovém registru DE je převeden (uložen) do registru A. Naopak - LD (BC),A způsobí přenos obsahu registru A na adresu paměti určenou obsahem párového registru BC. Operační kódy těchto instrukcí jsou:

LD A,(rr): 0 0 r r 1 0 1 0
LD (rr),A: 0 0 r r 0 0 1 0

Pro rr platí - 00 je BC, 01 je DE, 11 je HL.

Bezprostřední dvoubajtový přenos dat do párových registrů
LD rr,NNNN

Tato instrukce patří k tzv. 16tibitovým, protože přenáší dva bajty, tedy 16 bitů do jednoho z těchto párových registrů: BC, DE, HL a reg. SP. V angličtině se tento přenos jmenuje Load immediate extended to register pair. Immediate proto, že se jedná o okamžitý, nikoli zprostředkovaný přenos (data jsou přímou součástí instrukce). Extended znamená rozšířený adresovací mód, nejedná se však o nic jiného, než že se přenáší ne jeden, ale dva bajty. Operační kód instrukce vyhlíží takto:

0 0 r r 0 0 0 1

Pro rr platí - BC je 00, DE je 01, HL je 10 a SP je 11.

Dvoubajtový přenos mezi párovými registry a pamětí
LD rr,(adr); LD (adr),rr

Instrukce tohoto typu náleží opět k rozšířenému adresovacímu módu. Zde se poprvé setkáváme se základním pravidlem syntaxe instrukčního souboru Z80, v němž zásadně platí, že pokud za operačním kódem následují dva datové bajty (vždy se jedná o číselnou reprezentaci adresy), do paměti se za operační kód ukládá **napřed nižší bajt adresy a teprve za něj vyšší bajt adresy.** Co tyto bajty znamenají, si vysvětlíme na příkladu. Dejme tomu, že do instrukce potřebujeme zapsat adresu 65534 (dekadicky), což je FFFEH. Vyšším bajtem adresy se zde rozumí její horní "půlka", tedy FFH, nižším dolní "půlka", tedy FEH. Budeme-li chtít provést dvoubajtový přenos mezi pamětí s adresou určení FFFEH a párovým registrem BC, jehož mnemonika je LD(FFFEH),BC, pak HD tvar zápisu instrukce bude ED 43 FE FF.

ED 43 - operační kód instrukce
FE - nižší bajt adresy
FF - vyšší bajt adresy

Z toho vidíte, že assembler dodržuje běžnou logiku sledu zápisu čísel, zatímco HD prezentace instrukce se podřizuje pořadí, v jakém bajty musejí být uloženy přímo do paměti. Na to nikdy nezapomeňte! Omyly končívají tragický skonem programu.

Uvedená instrukce je ve své podstatě poněkud složitější a její plnou funkci si vysvětlíme o něco dále.

Zvýšení obsahu registru o 1
INC r

Anglicky zkratka INC znamená increment (zvyš). Po provedení této instrukce se obsah registru r zvýší o 1. Tak např. je-li obsahem registru B číslo 16, po instrukci INC B bude jeho obsah 17. Tato jednoduchá jednobajtová instrukce má základní binární tvar

0 0 r r r 1 0 0

Jaké registry můžete touto i jinými instrukcemi ovlivňovat, se lehce dozvíte z přiložených tabulek instrukčního souboru Z80. Z toho důvodu je zbytečné je v textu uvádět všechny. Instrukce INC r (podobně jako některé další instrukce) má nejkratší možnou dobu provedení - 4 kmity (necelé 2 mikrosekundy).

Snížení obsahu registru o 1
DEC r

Opak instrukce INC r. Po jejím provedení se obsah registru sníží o 1. K oběma těmto instrukcím jedna poznámka na připamatování si důležité věci. Pokud obsah registru před provedením instrukce INC r je FFH, po jejím provedení bude roven 0. Provedeme-li nato instrukci DEC r, bude obsah registru FFH. Jedná se o jev zvaný přeplnění registru, o němž jsme se zmínili ke konci 1. kapitoly a jímž se budeme ještě vydatně zabývat. Operační kód instrukce DEC r je:

0 0 r r r 1 0 1

Přímý skok s podmínkou NOT ZERO
JP NZ,XXXX

Zde zatím jen na ukázku typ podmíněné skokové instrukce (s ostatními se seznámíme později). Tato instrukce způsobí, že když je stavový indikátor (flag) nuly ve stavu log.0, provede se skok na adresu XXXX, odkud bude program pokračovat dál. Pokud výsledk testované operace nebude nulový, instrukce je ignorována a program pokračuje dál instrukcí následující. Sama instrukce je tříbajtová, s jednobajtovým operačním kódem:

1 1 0 0 0 0 1 0

Opět nesmíme zapomenout, že v HD zápisu do paměti musí nižší bajt adresy předcházet vyšší. Doba provedení instrukce je 10 kmitů. Tato instrukce má široké využití, a to např. v převedení programového řízení na adresy, které jsou od ní dále, než kam "dosáhne" instrukce relativního skoku JR, nebo třeba při konstrukcích časovacích smyček.

Blokový přenos dat
LDD, LDI, LDDR, LDIR

Dosud jsme probrali několik způsobů přenosů dat mezi registry a adresami paměti v rozsahu 1-2 bajtů. Z80 má čtyři velmi užitečné funkce, které dokáží přenášet z jedné části paměti na druhou celý blok dat najednou (přičemž se tyto oblasti paměti mohou prolínat). Před provedením této instrukce je nutno inicializovat (nastavit) obsah párových registrů BC, DE a HL takto: **HL - adresa prvního zdrojového bajtu**

DE - adresa prvního bajtu určení
BC - počet přenášených bajtů

Provedení instrukce LDI (Load-Increment) vyvolá následující pochody:

1. Bajt v paměťové adrese určené obsahem HL je přenesen a uložen na adresu určenou párovým registrem DE.
2. Obsahy registrů HL i DE se automaticky zvýší o 1.
3. Obsah BC se sníží o 1.

Provedení instrukce LDIR (load-increment repeat) má týž účinek jako instrukce LDI, ovšem s tím velmi důležitým rozdílem, že se najednou provede přesun takového počtu bajtů, který je roven číslu obsaženém v párovém registru BC:

1. Bajt z paměťové adresy určené registrem HL je přenesen na adresu určenou registrem DE.
2. Obsahy registrů HL i DE jsou oba zvýšeny o 1.
3. Obsah registru BC se sníží o 1.
4. Obsah registru BC je testován. Když není roven 0, pak se kroky 1,2,3 a 4 zopakují. Když je nulový, provádění instrukce se zastaví a program pokračuje následující programovou instrukcí.

Anglické slovo repeat znamená opakuj.

Instrukce LDD (load-decrement) a LDDR (load-decrement repeat) jsou shodné se svými výše uvedenými dvojčaty, ale s tím rozdílem, že během provádění kroku 2 se v obou případech obsahy registrů HL i DE snižují o 1.

U instrukcí s opakováním můžeme tedy přenést nejvíce FFFFH bajtů (65536 decimálně).

Pro snadnější porozumění funkce těchto instrukcí si text uvedený u instrukce LDIR porovnejte se schématem z přílohy.

Následujících šest experimentů demonstruje, co jste se naučili v této kapitole. Jedná se o přenosy dat. Více než vřele doporučujeme, abyste si zápisy jednotlivých experimentů na svém počítači provedli. Funkci každého programu tak pochopíte mnohem rychleji a snadněji, i když nad věcí strávíte trochu více času. Je to však investice, která se vám bohatě vyplatí. I z toho důvodu jsou následující programové ukázky nazvány experimenty, nikoli jen prostými příklady k přelétnutí očima.

EXPERIMENT č.1

Přenos jednobajtových dat a obsahů registrů.

```
0100  0680   LD B,80H     ;Datový bajt 80 je uložen do reg.B
0102  04     INC B        ;Zvýšení obsahu reg.B o 1
0103  48     LD C,B       ;Obsah reg.B je přenesen do reg.C
0104  0C     INC C        ;Zvýšení obsahu reg.C o 1
0105  51     LD D,C       ;Obsah reg.C do reg.D
0106  14     INC D        ;Zvýšení obsahu reg.D o 1
0107  5A     LD E,D       ;Obsah reg.D do reg.E
0108  1D     DEC E        ;Snížení obsahu reg.E o 1
0109  63     LD H,E       ;Obsah reg.E do reg.H
010A  25     DEC H        ;Snížení obsahu reg.H o 1
010B  6C     LD L,H       ;Obsah reg.H do reg.L
010C  2D     DEC L        ;Snížení obsahu reg.L o 1
010D  7D     LD A,L       ;Obsah reg.L do reg.A
010E  C9     RET          ;Návrat k adrese volání
```

Instrukci RET (Return - návrat) si probereme později. Teď si zapamatujte, že nás vrátí tam, odkud jsme rutinu volali (např. z Basicu příkazem typu RANDOMIZE USR 100H, LET 1=USR 100H apod. - to záleží na počítači, který máte). Jinými slovy - RET je obdobou basicového příkazu RETURN. Pokud změníte adresy uložení programu, musíte pochopitelně změnit i adresu volání rutiny.

KROK 1

Zapište uvedený program do paměti od libovolné adresy. Zkontrolujte si, jestli je program zapsán dobře. Protože jste se dali na cestu programátora, musíte si zvyknout na preciznost - odpovězte na tyto kontrolní otázky:

Kolik jednobajtových instrukcí je v programu? A dvoubajtových? Tříbajtových? Čtyřbajtových?

Pokud jste ještě neodpověděli, nedívejte se na správnou odpověď, nechcete-li podvádět sami sebe. Odpověď zní 13, 1, 0, 0. První instrukce programu je jedinou jeho dvoubajtovou instrukcí.

Kolik instrukcí používá přenos jednobajtových dat? A kolik registrový přenos?

Správná odpověď - 1 a 12. Instrukce LD B,80H je jedinou, která přenáší data. Ostatní instrukce přenášejí obsah jednoho registru do druhého.

KROK 2

Analyzujte program (nejdříve s tužkou v ruce a potom třeba krokováním) a určete, jaký bude HD obsah registrů B,C,D,E,H,L,A po skončení programu.

KROK 3

Pokud jste programem krokovali, určitě jste si v průběhu krokování všimli všech změn obsahů registrů programem ovlivněných. Správná odpověď na předchozí otázku je:

B=81, C=82, D=83, E=82, H=81, L=80, A=80

Umět předvídat změny obsahu registrů i jiné změny programu ve strojovém kódu je velmi důležité pro úspěšné odladění jakéhokoli programu. Vřele vám doporučujeme, abyste si tuto schopnost vypěstovali. Bez tréninku ji však nezískáte.

KROK 4

Změňte datový bajt na adrese 0101 na 01. A zkuste opět předpovědět konečné obsahy výše uvedených registrů. Správná odpověď:

B=02, C=03, D=04, E=02, H=02, L=01, A=01

KROK 5

Změňte datový bajt na adrese 0101 na FFH. Opět se pokuste určit konečný obsah registrů. Správná odpověď:

B=00, C=01, D=02, E=01, H=00, L=FF, A=FF

Čímž se potvrzuje, na co jsme naposledy upozorňovali v odstavci věnovaném instrukci DEC r. Když zvýšíme FFH o 1, dostaneme 00H. Když snížíme 00H o 1, dostaneme FFH. Na vašem monitoru strojového kódu jste si mohli povšimnout, že při tomto přeplnění registru v obou směrech se bit přenosu registru F překlopil z nuly do stavu log.1 (angl. Carry flag; obvykle používaná zkratka CY), tedy CY=1. K tomuto jevu se podrobněji dostaneme později.

EXPERIMENT č.2

Demonstrace dvoubajtového, rozšířeného a nepřímého registrového adresovacího módu.

```
0110  211C01   LD HL,011CH    ;011CH do reg.HL; 01 do H; 1C do L
0113  36FF     LD (HL),FFH    ;FFH na adresu 011CH
0115  2C       INC L          ;Zvýšení obsahu reg.L o 1. Tak je
                              ;v reg.HL 011DH
0116  36EE     LD (HL),EEH    ;EEH na adr.011DH
0118  2A1C01   LD HL,(011CH)  ;Rozšířený adres.mód. Bajt z adresy
                              ;011CH do reg.L, z 011DH do reg.H
011B  C9       RET            ;Návrat k adrese volání
```

KROK 1

Zapište program a zkontrolujte správnost jeho zápisu v počítači.

KROK 2

Celý program pracuje pouze s registry HL a dvěma adresami paměti 011CH a 011DH. Program na tyto adresy ukládá bajty s obsahem FFH a EEH. Poté převede obsahy těchto dvou adres do párového registru HL. Nejedná se zrovna o program, který by nadchl labužníka, ale dostatečně ilustruje několik závažných faktů všech tří zmíněných adresovacích módů.

Porovnejte tyto dva mnemonické kódy:

```
LD HL,011CH
LD HL,(011CH)
```

Na pohled se liší pouze závorkami. Avšak pro assembler má tato diference zásadní význam. V případě první instrukce jde o přenos dvou bajtů 01 a 1C do párového registru HL - jedná se tedy o přímý dvoubajtový přenos. Ve druhém případě však číslo 011CH v závorce znamená první adresu, z níž bude proveden přenos dat (čili přenos obsahu bajtu z uvedené adresy) do párového registru HL. Přestože tuto instrukci si probereme ještě později, tak abychom předešli možným omylům, sdělíme vám předem, že po provedení této instrukce bude bajt z adresy 011CH převeden do registru L a bajt z adresy o 1 vyšší (tedy z 011DH) do reg.H (napřed z nižší do nižšího, pak z vyšší do vyššího registru).

U instrukcí jednobajtového přenosu dat LD (HL),EEH a LD (HL),FFH je třeba si uvědomit, že zde nedochází ke změně obsahu reg.HL vlivem přenosu. V reg.HL zůstává číslo adresy, na níž se datový bajt přenáší, nezměněno. Změní se však vlastní obsah adresy určené obsahem reg.HL.

KROK 3

Určete, jaký HD obsah budou nakonec mít registry H a L a co bude na adresách 011CH a 011DH.

KROK 4

Zjistěte si z monitoru správnost svých odpovědí: H=EE, L=FF, (011C)=FF, (011D)=EE.

EXPERIMENT č.3

Sestavení programové smyčky - v uvedeném případě jde o časovací smyčku za použití instrukce JP NZ,XXXX.

Rutina A

```
0120  0E00        LD C,00H      ;Vynulování reg.C
0122  0D    LOOP:DEC C          ;Snížení obsahu reg.C o 1
0123  C22201      JP NZ,LOOP    ;Když je reg.C různý od nuly, skok
                                ;na adresu LOOP
0126  C9          RET           ;Když je reg.C=0, návrat
```

Rutina B

```
0130  0600         LD B,00H      ;Vynulování reg.B
0132  0E00  LOOP1:LD C,00H       ;  "        "  C
0134  0D    LOOP2:DEC C          ;Obsah reg.C-1
0135  C23401       JP NZ,LOOP2   ;Když reg.C není 0, skok na LOOP2
0138  05           DEC B         ;Obsah reg.B-1
0139  C23201       JP NZ,LOOP1   ;Když reg.B není 0, skok na LOOP1
013C  C9           RET           ;Návrat
```

KROK 1

Zapište do počítače obě rutiny a zkontrolujte správnost zápisu.

KROK 2

Obě rutiny si probereme trochu podrobněji, abychom přesně pochopili jejich funkci. Rutina A je jednoduchou časovací smyčkou. Novinkou pro vás je použití návěští (label) v zápisu programu - LOOP, LOOP1, LOOP2. Tato návěští jsou nesmírnou pomůckou při programování v assembleru. Jak jste se z úvodní kapitoly dozvěděli, je každý program protkán instrukcemi skoků a volání. Pokud budeme skok na adresu nebo její volání chápat jen jako přeskok na nějaké číslo adresy, budeme "v tom" mít za chvíli pořádný zmatek. Pojmenujeme-li si však tyto adresy nějakým symbolickým názvem (třeba adresu rutiny, která pohybuje bodem na obrazovce vpravo slovem PRAHYB, vlevo VLEHYB, rutinu tisku TISK apod.), struktura programu bude rázem přehlednější.

To však ještě není vše, čím návěští ulehčují život programátorův. Generátory strojového kódu jsou vybaveny jednou skvělou předností - dokáží s nimi pružně pracovat tak, že můžeme mezi návěští a jejich skokové instrukce libovolně vkládat nebo naopak ubírat bajty dle libosti, aniž se musíme starat o to, zda skok nebo volání budou provedeny správně. Jak sami vidíte, nemusíme na adrese 0123 psát JP NZ,0122, stačí pouhé JP NZ,LOOP. Mezi adresy 0122 a 0123 můžeme cokoli vkládat, přesto však kdykoli dojde řada na provedení instrukce JP NZ,LOOP, pak v případě splněné podmínky NZ program skočí vždy na adresu LOOP. Pokud by generátor tuto schopnost neměl, museli bychom během všech oprav a přesunů instrukcí v rutinách stále měnit i adresy skoků a volání, což, jak sami uznáte, by bylo více než nepříjemné a únavné. Generátorem si obvykle ještě můžeme nechat vypsat všechna dosud použitá návěští i s jejich adresami, což dále zvyšuje orientaci zvláště v delším programu.

Nyní k samotné rutině A. Reg.C po snížení svého obsahu z nuly bude mít obsah 255 (FFH). Protože podmínka v instrukci JP NZ,LOOP stanoví, že pokud obsah reg.C nebude roven nule, má program skočit vždy na adresu LOOP, učiní tak vždy, dokud instrukcí DEC C nebude snížen obsah reg.C na nulu (celkem tedy 256krát). Poté bude program pokračovat v chodu na adresu s instrukcí RET a vrátí řízení zpět k adrese volání rutiny. Tak vlastně program vykonal 256 skoků v časovací smyčce. Její trvání můžeme řídit stanovením vstupní hodnoty reg.C. Čím nižší tato hodnota bude, tím kratší bude i doba trvání smyčky.

Pro programy tohoto typu je dobré si nakreslit jejich vývojový diagram - mohli jste se seznámit v 1.kapitole. Pokud se vám stane, že máte program zaplaven podmínkami a odskoky, v nichž se už sami přestáváte vyznávat, diagram si nakreslete. Pomůže vám nejen najít chyby, ale někdy dodatečně zjistíte, že program můžete i výhodně zkrátit.

Rozličné časovací smyčky se v programech objevují velmi často. Jsou nezbytné všude tam, kde probíhá nějaký proces, jehož rychlost potřebujeme zbrzdit, resp. potřebujeme nastavit přesný čas provedení nějaké části programu, nebo čekáme po určitý časový úsek na nějakou datovou informaci (u periférií) apod. Podobné smyčky se používají i tehdy, když chceme, aby proběhly vícekrát po sobě nějaké operace, když inicializujeme obsahy registrů nebo míst paměti apod.

KROK 3

Rutina B je příkladem možnosti začlenění dvou smyček do sebe. Smyčka LOOP2 je vnitřní, LOOP1 vnější. LOOP2 se provede vždy 256krát. Délku časové prodlevy způsobené touto časovací rutinou nastavujeme vstupní hodnotou registru B. Takových smyček bychom mohli prokombinovat mnoho. V dalších částech učebnice se dozvíme, jak sestavovat časovací smyčky i jiným, úspornějším a efektivnějším způsobem.

KROK 4

Pro úplnost si ještě ukážeme, jak do rutiny B začlenit ještě jednu vnější smyčku:

```
012E   1630      LD D,30H     ;Inicializace proměnné vnější smyč.
;
;Na adresách 0130-013B je umístěna rutina B (bez instr.RET)
;
013C   15        DEC  D       ;Snížení obsahu reg.D o 1
013D   C23001    JP NZ,0130H  ;Pokud není D=0, skok na obě vnitř-
                              ;ní smyčky
0140   C9        RET          ;Je-li D=0, návrat
```

KROK 5

Pokud jste programátorským začátečníkem, měl byste se pokusit zakreslit vývojový diagram celé rutiny se všemi třemi smyčkami. Hodně vám poví.

KROK 6

Spusťte program od adresy 012EH. Jak sami zjistíte, čas jeho provedení se značně natáhl. Čekáte-li však na jeho provedení déle než minutu, máte v uložení programu do počítače nějakou chybu. Musíte provést RESET počítače a zkusit to znova, teď už s pečlivější kontrolou svého zápisu.

KROK 7

Zkuste měnit bajt na adrese 012FH, ať se přesvědčíte, jak jeho hodnota ovlivní dobu trvání průběhu rutiny.

EXPERIMENT č.4

Blokový přenos s instrukcí LDDR.

```
0150   217501    LD HL,0175H   ;Nejvyšší zdrojová adresa bloku, z
                               ;nějž budou data odebírána
0153   116F01    LD  DE,016FH  ;Nejvyšší adresa určení nového
                               ;uložení přenášeného bloku dat
0156   010500    LD BC,0005H   ;Do reg.BC počet bajtů přenosu
0159   EDB8      LDDR          ;Blokový přenos
015B   C9        RET           ;Návrat
```

KROK 1

Proveďte obvyklé vložení programu a kontrolu jeho zápisu.

KROK 2

Podle obsahu párového registru BC jste jistě poznali, že se jedná o přenos pěti bajtů, který v mžiku proběhne v tomto sledu:

bajt z adresy:	je přenesen na adresu:
0175	016F
0174	016E
0173	016D
0172	016C
0171	016B

KROK 3

Uložte na adresy 0171H-0175H postupně tyto bajty: AAH, BBH, CCH, DDH, EEH. Spusťte program a poté si prohlédněte obsah adres, na něž mají být uvedené bajty přeneseny. Měly by se vám na nich objevit ve stejném sledu. Můžete si vyzkoušet i osazení

jiného počtu bajtů zdrojového bloku se změnami obsahu reg.BC, případně i změnami reg.HL a DE, abyste se s instrukcí seznámili co nejdůvěrněji.

Instrukce blokového přenosu dat patří mezi instrukce s tzv. mezními stavy. Např. když zahájíte přenos s obsahem reg. BC=0000, pak se BC převrátí do stavu FFFFH a instrukce provede přenos 65536 bajtů (přičemž zničíte všechny programy, které v paměti máte)! Na to je třeba při stavbě programu pamatovat. Při tak velkém rozsahu přenosu se ovšem neprovede přenos všech 65536 bajtů, ale tolika, kolik jich bude přeneseno do doby, než začne být přepisována sama instrukce přenosu (program zničí sám sebe).

EXPERIMENT č.5

Obsahuje analýzu instrukce LDI. Nově zařadíme další podmíněné instrukce přímého skoku JP Z,XXXX a JP PE,XXXX a logickou instrukci OR r.

```
0180  21A001      LD HL,01A0H    ;Zdrojová adresa
0183  11C001      LD DE,01C0H    ;Adresa určení přenosu
0186  011000      LD BC,0010H    ;Počet bajtů přenosu
0189  7E    LOOP:LD A,(HL)       ;Uložení každého zdroj.bajtu do r.A
018A  B7          OR A           ;Nastavení indikátoru nuly Z v
                                 ;reg.F na log.1, když obsah reg.A=0
018B  CA9301      JP Z,KONEC     ;Je-li A=0, skok na adr.KONEC
018E  EDA0        LDI            ;Přenos bajtu
0190  EA8901      JP PE,LOOP     ;Dokud BC není 0, skok na adr.LOOP
0193  C9    KONEC:RET            ;Návrat
```

KROK 1

Proveďte zápis do počítače a jeho kontrolu.

KROK 2

Nyní se podíváme na nám zatím neznámé instrukce:

OR A - logická operace OR akumulátoru samého se sebou. Např. OR C by byla operace OR reg.C s akumulátorem, k němuž se tato instrukce vztahuje vždy. Indikátor Z=1 tehdy, když výsledkem operace je nula - a to nastane jen tehdy, když obsahy obou na operaci zúčastněných bajtů jsou nulové. Této vlastnosti funkce OR se velmi často využívá v testech zjišťujících, zda obsahy nějakých dvou bajtů (nebo obsah akumulátoru) jsou nulové. Logickými instrukcemi se budeme zabývat později. Tento test, který patří k programátorským finesám, si zapamatujte pro svou budoucí praxi.

JP Z,XXXX - Podmíněná instrukce přímého skoku na adresu XXXX. Provede se tehdy, je-li indikátor Z=1. Jinými slovy tehdy, je-li výsledkem poslední předchozí operace, která ovlivňuje stav indikátoru Z, nula.

JP PE,XXXX - Podmíněná instrukce přímého skoku na adresu XXXX. Dokud obsah reg.BC není nulový, indikátor P/V je ve stavu log.1 - podmínka PE je splněna a provede se skok na adresu XXXX. Při BC=0 je platná podmínka PO. Tento test má u instrukcí blokového přenosu a prohledávání své zvláštní postavení (nesouvisí se zjišťováním parity - viz i podmínka PO v programu 2 na str.73). O podmínkách PE a PO vztahujících se ke zjištění parity si povíme později.

Vzpomeňte si, že vedle těchto podmíněných skokových instrukcí známe ještě jednu - JP NZ,XXXX. Jak jste si už určitě uvědomili, mnemonika Z80 řadí instrukce, které jsou si něčím podobné, do skupin tak, že nám umožňuje snadnou orientaci v celém instrukčním souboru Z80.

Podmíněné instrukce mají jedno společné - všechny testují stav některého z indikátorů v reg.F a provedou se jedině tehdy, je-li jimi vytčená podmínka splněna. Jinak jsou ignorovány. Ovšem ne tak zcela - jak vysvítá z tabulky, v níž jsou uvedeny časy potřebné pro provedení jednotlivých instrukcí, i v případě nesplnění podmínky určitý čas zabere dekódování instrukce a zjišťování, zda je podmínka splněna - čas je ovšem mnohem kratší, než v případě podmínky splněné. Proto jsou u podmíněných instrukcí (a těch, u nichž je nějaká "skrytá" podmínka jejich vlastní přirozenosti) uvedeny vždy dvě časové hodnoty.

KROK 3

Proveďme si analýzu našeho programového experimentu. První tři instrukce nastavují obsahy párových registrů pro blokový přenos dat instrukcí LDI. Další dvě instrukce zjišťují, zda obsahem

přenášeného bajtu (z adresy určené číslem v reg.HL) není nula. Bajt z této adresy je přenesen do reg.A, aby s ním hned nato byla provedena logická operace OR A. Je-li obsah reg.A=0, podmínka instrukce JP Z,KONEC je splněna a program přeskočí na adresu KONEC, na níž je instrukce návratu RET - program se ukončí. Není-li podmínka splněna, provede se vlastní přenos bajtu instrukcí LDI. Jak už výše uvedeno, indikátor P!V se provedením instrukce dostane do stavu log.1, přičemž je nutno testovat paritu (podmínky PE a PO), nikoli přeplnění (C a NC). Dokud obsah reg.BC není roven nule, provede se vždy skok na adr. LOOP. Jakmile je reg.BC=0, indikátor P!V bude rovněž nulový - instrukce JP PE,LOOP se neprovede, program bude pokračovat na instrukci RET a ukončí se.

Test na adrese LOOP je ukázkou toho, jak je možno využít nějaké hodnoty bajtu (zde je to nula) k tomu, aby se řetěz provádění instrukcí přerušil a program přešel na jinou svou část tak, jak mu navelíme v podmínce. Kdykoli se tedy mezi našimi 16ti bajty přenosu objeví bajt s obsahem nula, program se ukončí. Toho se dá využít např. při sestavování nápisů na obrazovce - dejme tomu, že oddělíme jednotlivá slova (jejich písmena budou bajty s obsahem kódů ASCII) oddělíme nulami. Po testu nuly se programové řízení převede na rutinu zadání nové pozice začátku tisku dalšího slova na obrazovce. Z této rutiny se program vrátí na přenos dalších písmen na obrazovku. Jakmile v přenosu znova narazí na nulu, přejde se na zadání nové pozice tisku, pak zase zpět atd., dokud nebude patřičný počet slov vypsán. Tento počet si zase určíme nějakým testem, který zjistí, jestli je vypsáno vše, co bylo třeba a po splnění podmínky se převede řízení programu na jinou programovou rutinu, jak sami - jakožto programátoři - stanovíme.

Z toho přibližně vidíte, jak takový program ve strojovém kódu pracuje - jedna rutina za druhou jsou aktivovány tak, aby program vykonával, co od něj zařazením podmínek se skoky, voláními, prohledáváním paměti atd., atd. vyžadujeme. Jde vlastně nejen o detailní vytvoření samotného programu, ale především jeho struktury, která bude funkční, optimální, efektivní a bezchybná. Bez předchozího promyšlení "generálního strategického plánu" stavby našeho programu se v něm při jeho předem nepromyšleném vytváření snadno a brzy utopíme. Strukturou programu je v tomto případě myšleno blokové schéma programu, sestavené z jednotlivých rutin s uvedením jejich funkce, vstupních podmínek a výstupních parametrů.

KROK 4

Uložte do paměti od adresy 01A0H libovolné nenulové bajty v počtu 16 a spusťte program. Po jeho skončení si prohlédněte obsahy registrů HL, DE, BC. Měly by obsahovat HD čísla 01B0, 01D0 a 0000. Tak je potvrzena správná funkce programu.

KROK 5

Nyní na uvedené adresy uložte tyto bajty:

```
01A0 - 10
01A1 - 0F
01A2 - 0E
01A3 - 00
01A4 až 01AF - FF
```

Spusťte program. Co se stane, až dojde k testu nulového bajtu z adresy 01A3H? Program se samozřejmě zastaví, přičemž obsahy registrů budou tyto: BC=000DH, HL=01A3H a DE=01C3H. Přenos bajtů s obsahem FFH už proveden nebude. Tím je potvrzena naše předchozí analýza programu.

Pokud se tážete, k čemu je nám opakování instrukce LDI nebo LDD když máme možnost použít LDIR nebo LDDR, které se opakují samy, pak je odpověď velmi jednoduchá. Do přenosu prováděného instrukcemi s automatickým opakováním nemáme možnost zasáhnout, tedy ani testovat jednotlivé přenášené bajty, kdežto instrukce LDI a LDD nám zařazení testu umožňují.

EXPERIMENT č.6

V něm si ukážeme, kolik jakých instrukcí navíc bychom museli použít pro náhradu instrukce LDIR, kdyby ji mikroprocesor Z80 neobsahoval ve svém instrukčním souboru (např. mikroprocesor 8080 tuto instrukci nezná). Zároveň si porovnáme i dobu, po kterou je prováděna instrukce LDIR s dobou, kterou zabere její náhrada.

Rutina A

```
01D0  210002   LD HL,0200H   ;Inicializace registrů k provedení
01D3  110102   LD DE,0201H   ;instrukce LDIR
01D6  016400   LD BC,0064H
01D9  EDB0     LDIR
01DB  C9       RET
```

Rutina B

```
01D0  210002      LD HL,0200H   ;Obvyklá inicializace registrů
01D3  110102      LD DE,0201H
01D6  016400      LD BC,0064H
01D9  7E     LOOP:LD A,(HL)     ;Obsah adresy HL do reg.A
01DA  12          LD (DE),A     ;Přenos obsahu reg.A na adresu DE
01DB  23          INC HL        ;Zvýšení HL o 1
01DC  13          INC DE        ;Zvýšení DE o 1
01DD  0B          DEC BC        ;Snížení BC o 1
01DE  78          LD A,B        ;Test obsahu BC; je B i C rovno 0?
01DF  B1          OR C          ;a je tedy obsah reg.BC=0?
01E0  C2D901      JP NZ,LOOP    ;Pokud ne, skok na adr.LOOP
01E3  C9          RET           ;Návrat
```

KROK 1

Obě rutiny jsou ekvivalentní. Obě přenášejí blok 100 bajtů. Rutina A však zabírá jen 14 bajtů paměti a její provedení trvá 2095 kmitů, tj. 2095*0,000004=0,00838 vteřiny.

Oproti tomu Rutina B zabírá 22 bajtů paměti a její provedení při přenosu pouhého 100 bajtů trvá 5000 kmitů, tj. 0,02 vteřiny, což je více než dvakrát tolik! S rostoucím počtem přenášených bajtů poroste úměrně i doba provedení přenosu bloku dat Rutinou B.

KROK 2

Ozkoušejte si funkci obou rutin, abyste zjistili, že jsou skutečně ekvivalentní. Zvyšováním počtu bajtů přenosu v Rutině B si můžete ověřit, že doba jejího provedení začne být brzy patrná.

V Rutině B je několik nových instrukcí zvyšování a snižování obsahů párových registrů. Probereme si je později. Nerozlučný pár instrukcí testů obsahu dvou registrů logickou funkcí OR už znáte z předchozích experimentů.

****

KAPITOLA 4

Adresovací módy Z80

****

Doufáme, že obsah minulé kapitoly byl pro vás dostatečně stravitelný. Stále stojíme teprve na začátku. I když jste byli - zatím jen lehce - ovanuti i jinými instrukcemi, než typu LD s jednobajtovým operačním kódem, to hlavní na vás teprve čeká. Od této kapitoly přidáme plyn a začneme pomalu přerůstat v odborníky strojového kódu. Ale nemějte obavy, nic se nejí tak horké, jak to zpočátku vypadá. Nepůjde-li vám to hned do "trávícího traktu", umístěného za klenbou lebeční, nechte to trochu vychladnout a zakousněte si ještě jednou z kapitoly minulé - zřejmě vám z ní všechno ještě pořádně neslehlo. A hlavně - v ničem nešiďte sami sebe! Jakékoli ošizení výukových dávek se dříve či později negativně projeví. A mohlo by vás dokonce přivést i k zanevření na strojový kód, který by v tom ovšem byl zcela nevinně.

Adresovacími módy jsme se začali zabývat již v minulé kapitole. Z80 jich má celkem 10. Pět z nich máme šťastně za sebou - registrové, registrové nepřímé, datové jedno- i dvoubajtové a rozšířené. V této kapitole si probereme především dvojkovou aritmetickou komplementaci, která je základem pro pochopení indexovaného a relativního adresování v několika typech instrukcí. Nakonec si uděláme přehled všech adresovacích módů.

Dvojkově komplementární binární reprezentace

Následující řádky budou poněkud hutnější pro ty z vás, kteří ve škole více holdovali přestávkám než matematice. Snad vám trochu usokojení přinese ujištění, že mnohem více než o matematiku samu (nakonec se budeme pohybovat jen v rozmezí čísel -128 až +127), půjde o logiku uvažování.

Zatím víme, že číslo, které se "vejde" do jednoho bajtu, není menší než nula a větší než 255, což dává dohromady 256 čísel. Rovněž víme, že těchto 256 čísel je vlastně 256 možných kombinací jedniček a nul na pozicích osmi bitů jednoho bajtu. Řekněme si hned na začátku, že dvojková komplementace (budeme ji dále značit zkratkou DK) nám v určitých případech umožňuje pracovat s jedním bajtem jako číslem v rozsahu -128 až +127 včetně nuly - v podstatě se jedná jen o jiný přístup k chápání stále stejného počtu 256 kombinací binárního kódu osmi bitů jednoho bajtu. K čemu je tento doplněk dobrý, si ukážeme hned poté, co si vysvětlíme, jak to s tou DK vlastně je.

Abychom si to na začátku trochu zjednodušili, budeme počítat jen se čtyřmi bity. Tak si podstatu DK vysvětlíme na příkladu čtyřbitové DK reprezentace. A hned do ní skočíme po hlavě:

Decimální číslo	HD číslo	4-bitová DK reprezentace
7	7	0 1 1 1
6	6	0 1 1 0
5	5	0 1 0 1
4	4	0 1 0 0
3	3	0 0 1 1
2	2	0 0 1 0
1	1	0 0 0 1
0	0	0 0 0 0
-1	F	1 1 1 1
-2	E	1 1 1 0
-3	D	1 1 0 1
-4	C	1 1 0 0
-5	B	1 0 1 1
-6	A	1 0 1 0
-7	9	1 0 0 1
-8	8	1 0 0 0

Na první pohled je patrné, že čísla kladná a záporná se liší obsahem nejvyššího bitu. Kladná (a nula) mají tento bit ve stavu 0, zatímco záporná ve stavu 1. Zapamatujte si jednu mnemotechnickou pomůcku - zařaďte si v duchu nulu mezi kladná čísla. Pak vám nebude dělat potíže uvědomění si, že kladných čísel (větších než 0) je o 1 méně než záporných (neboť s nulou je jejich počet stejný).

Normální binární reprezentace 4 bitů by zahrnovala čísla v intervalu 0-15. V případě DK budou čísla v rozsahu -8 až +7:

-2**(n-1) až +(2**(n-1))-1 (kde n je počet bitů)

Kladné DK číslo je identické s jeho normálním binárním kódem; HD tvar DK čísla je vždy identický s jeho binárním kódem!!!

Dvojkovým komplementem čísla X je takové číslo, které přičteno k číslu X způsobí, že výsledek binárního součtu bude o jeden řád vyšší, přičemž všechny zbývající číslice součtu budou rovny nule:

```
  0 0 0 1        1 0 1 0        0 1 0 1
  1 1 1 1        0 1 1 0        1 0 1 1
 ---------      ---------      ---------
 1 0 0 0 0      1 0 0 0 0      1 0 0 0 0
```

To znamená, že dvojkovým komplementem čísla 0001 je číslo 1111, čísla 1010 číslo 0110 a čísla 0101 číslo 1011. Přitom je třeba si uvědomit, že ve vlastním provádění operací s bajty je při DK součtu výsledkem 0000, tedy nikoli 10000. Jedná se o nám již známé přeplnění bajtu. Podíváme-li se tedy na celou věc z hlediska bajtových operací, můžeme říci, že vzájemně dvojkově komplementární jsou taková čísla, jejichž součet vynuluje obsah bajtu, do nějž se výsledek součtu ukládá (s následným CY=1).

V dalším se dozvíme, že sečítání DK čísel je v přímém vztahu k odečítání běžných čísel, protože odečítání je ekvivalentní DK součtu obou čísel odečítání se účastnících.

Jak najdeme DK číslo k číslu danému? Napřed převrátíme stavy všech bitů čísla do stavů opačných a nakonec přičteme číslo 1:

```
DK čísla 1010 je: 0101 + 0001 = 0110
DK čísla 0000 je: 1111 + 0001 = 0000
DK čísla 1000 je: 0111 + 0001 = 1000  (NE!)
```

V posledním případě se jedná o DK čísla -8, které nemá svou DK. V jakémkoli binárním rozsahu platí vždy, že nejvyšší záporné číslo nemá svůj DK!

Jaké bude nejvyšší kladné číslo osmibitového binárního DK kódu? Protože kladné číslo musí začínat nulou, budou na zbývajících sedmi pozicích jedničky: 01111111 (to je decimálně +127).

Jaké bude nejnižší DK číslo téhož kódu? Napřed převrátíme stavy všech bitů (10000000) a přičteme číslo 1. Výsledek je 10000001 (decimálně -127). Z toho je patrno, že může existovat ještě jedno menší číslo (10000000), což je DK reprezentace čísla -128. Tím se opět potvrzuje, že nejvyšší záporné číslo (zde -128) nemá svůj vlastní DK.

Osmibitová binární DK reprezentace v sobě tedy zahrnuje čísla v intervalu -128 až +127, čímž jsme si potvrdili naše tvrzení v úvodu této kapitoly.

Vše výše uvedené vám pomůže kdykoli určit DK reprezentaci jakéhokoli binárního čísla. V případě, že nejvyšší bit bajtu je nulový (pak i SI znaménka S=0), jedná se o kladné DK číslo, které je ekvivalentní jeho běžné hodnotě. Je-li nejvyšší bit bajtu ve stavu log.1 (S=1), jedná se vždy o záporné číslo DK reprezentace. Chceme-li zjistit jeho decimální vyjádření, musíme k němu dle výše uvedených pravidel najít dvojkový komplement a pak před něj jednoduše zařadíme znaménko mínus. Doporučujeme vám, abyste si s uvedenými pravidly pro DK čísla trošku pohráli, aby vám přešla do krve. Na závěr se pokusíme vyřešit úlohu, v níž máme najít DK číslo k dekadickému číslu -13:

1. Binární kód čísla 13 je 00001101
2. Binární DK reprezentace 00001101=11110011 (tj.11110010+1)
- DK reprezentace čísla -13 je tedy 11110011

DK reprezentace decimálního čísla 100 je rovna jeho binárnímu kódu 01100100. Pokud chceme najít DK k decimálnímu číslu 100, můžeme použít výrazu (2**n)-x, kde x je kladné číslo, k němuž hledáme DK a n je počet bitů v jeho binární reprezentaci. Tak bude DK k dekadickému číslu +100:

(2**n)-x=(2**8)-100=156 (t.j.10011100), což je v DK -100

Uvedená DK čísla nejsou nikterak samoúčelná a ve výpočetní technice mají svůj velký význam. Pro plné obeznámení se s nimi si probereme ještě operace zvané

Dvojkově komplementární sečítání (i odečítání)

Součet DK čísel je velmi triviální:

```
0 0 0 0 0 1 1 1   (+7)
0 0 0 0 0 0 1 0   (+2)
---------------------
0 0 0 0 1 0 0 1   (+9)

1 1 1 1 1 1 0 0   (-4)
0 0 0 0 0 0 1 1   (+3)
---------------------
1 1 1 1 1 1 1 1   (-1)

1 1 1 1 1 0 0 1   (-7)
1 1 1 1 0 0 1 1   (-13)
---------------------
1 1 1 0 1 1 0 0   (-20)

0 1 1 0 0 0 0 0   (+96)
0 1 0 1 0 0 0 0   (+82)
---------------------
1 0 1 1 0 0 0 0   NELZE!

1 0 1 1 1 0 0 1   (-71)
1 0 1 1 1 0 0 0   (-72)
---------------------
0 1 1 1 0 0 0 1   NELZE!
```

K prvním třem příkladům není třeba nic dodávat. V posledních dvou došlo k přeplnění ("přetečení") rozsahu osmibitové binární DK reprezentace. Kladné číslo nesmí být vyšší než 127, záporné menší než -128. K detekování takového přeplnění používáme stavový indikátor DK "přetečení" P/V. Dojde-li přitom k přeplnění binární reprezentace bajtu, i indikátor přenosu (carry flag) v registru F. O tom si povíme až dále.

Nyní se vrátíme k vlastním adresovacím módům Z80 a probereme si hezky všech deset popořádku.

Registrové adresování

To je takové adresování, kdy operační kód instrukce určuje, který (které) registr (-y) se účastní provedení instrukce. Registry jsou v operačním kódu reprezentovány tříbitovým kódem. Tak např. instrukce LD A,B má operační kód:

```
0 1 1 1 1 0 0 0
    -----  -----
      A      B
```

HD reprezentace této instrukce je 78H - v tomto operačním kódu jsou obsaženy dva registry, asice A a B.

Bezprostřední adresování

Je užíváno ve vícebajtových instrukcích, které operují s jednobajtovými daty, např. LD C,03 (HD kód je 0E 03). Tato instrukce uloží do reg.C číslo 03. Protože se de facto jedná opět o přenos čísla do registru, operační kód v sobě obsahuje tříbitový registrový kód, takže je zde vlastně použit i mód registrového adresování.

Bezprostřední rozšířené adresování

Pro instrukční soubor Z80 by bylo mnohem jednodušší nazývat předchozí mód jednobajtovým adresováním a tento dvoubajtovým - v něm se nejedná o nic jiného, než že do párového registru ukládáme dvoubajtové (šestnáctibitové) číslo. Operační kód je jednobajtový; zbytek tříbajtové instrukce tohoto módu doplňují dva datové bajty. Např. LD B,0421H (HD kód 01 21 04). Vzpomeňte si, že pro ukládání HD kódu do paměti počítače platí striktní pravidlo pořadí ukládaných datových bajtů - napřed ukládáme bajt nižší (21) a potom bajt vyšší (04).

Registrové nepřímé adresování

Ukládaná (přenášená) data byla v dosud uvedených módech vždy přímo obsažena v instrukci. V tomto módu operujeme s datovými bajty, které nejsou součástí instrukce, ale registry v instrukci uvedené adresují (určují, na kterou či které dvě uložit nebo ze které či kterých dvou adres, resp. jednoho registru či jejich páru odebrat) jeden nebo dva bajty. Proto se tomuto módu říká nepřímé adresování. Např. LD A,(HL) (HD kód 7E) způsobí přenos bajtu z adresy, jejíž číslo tvoří obsah párového registru HL do akumulátoru. Zde tedy funguje registr HL jakožto ukazatel adresy a samy o sobě ani reg.L, ani reg.H nejsou přenosem jakkoli změněny. Změněn je pouze obsah registru A (samozřejmě ve všech případech, kdy obsah adresy adresované párovým reg.HL se liší od

původního obsahu akumulátoru). Sluší se dodat, že v tomto případě nezměněn zůstane i obsah adresy (HL). Některé instrukce používají nepřímé adresování pro přenos dvou bajtů. Např. POP BC (HD kód C1) přenese obsah adresy (SP) do reg.C a obsah adresy (SP+1) do reg.B - k tomu však až později.

Rozšířené adresování

Instrukce tohoto módu obsahují přímo číslo adresy ve svých posledních dvou bajtech. Jedná se tedy o adresování s přímo udaným číslem adresy. Např. LD (1234H),A - HD kód 32 34 12. Tato instrukce přenese obsah akumulátoru (1 bajt) na adresu 1234H. Syntax mnemoniky Z80 dodržuje pravidlo, že adresa (ať už adresovaná nepřímo nebo určená přímo jejím číslem) se píše do závorek. Jsou ovšem výjimky z pravidla: JP 4321H znamená přímý skok na adresu 4321H. Adresa zde není v závorce proto, že se nejedná o přenos dat, ani o nepřímé adresování.

Adresování RST

Z80 obsahuje celkem 8 zvláštních instrukcí, které způsobují převedení řízení programu na instrukcemi pevně stanovené adresy: RST 00H, RST 08H, RST 10H, RST 18H, RST 20H, RST 28H, RST 30H a RST 38H. Tyto instrukce jsou jednobajtové, zatímco srovnatelná instrukce CALL XXXX je tříbajtová. V tom je jejich jediná zvláštnost i přednost. Číslo v instrukci RST je číslem adresy, na níž je program po jejím provedení převeden. RST XX se používá pro aktivování zásadních, tedy i velice často užívaných rutin operačního systému počítače. Anglicky se tomuto módu říká poněkud složitě Modified page zero addressing, což česky znamená adresování s modifikovanou stránkou nula, čímž se chce vyjádřit, že vyšší bajt adresy je vždy 00.

Implikované adresování

znamená, že jednotlivé skupiny instrukcí se vždy vztahují k jednomu registru (mají jej v sobě vždy zahrnut). Příkladem takové skupiny instrukcí jsou aritmetické a logické instrukce, v nichž je implikován reg.A. Např. SUB B znamená, že od obsahu reg.A bude odečten obsah reg.B. Přestože reg.A není přímo obsažen v zápisu instrukce, je v ní implikován. Podobně instrukce XOR H provede logickou operaci mezi registry A a H. Jedinou výjimku v zápisu implikovaných instrukcí vztahujících se k reg.A, tvoří instrukce součtu (např.ADD A,C), kde je reg.A přímo uveden (i když by vlastně být nemusel).

Bitové adresování

Jedná se o instrukce, které se váží k jednotlivým bitům obsahu adres paměti nebo registrů. Bity jsou číslovány od 0 do 7, přičemž nejnižší bit je bit 0. Např. SET 3,B nastaví bit 3 reg.B do stavu log.1. Kromě SET patří do této skupiny RES a BIT.

Indexované adresování

Z80 má dva speciální registry - IX a IY - které nelze chápat jako registry párové (nelze tedy operovat s jejich polovinami), ale jen jako registry 16tibitové. Adresování probíhá pomocí uložení čísla odchylky (angl.displacement) od adresy v reg.IX, resp.IY do patřičného bajtu instrukce. Tato odchylka může být jen v rozmezí -128 až +127 (viz první příklad užití DK čísel). Např. LD A,(IX+02) - HD kód DD 7E 02 - znamená, že obsah adresy o 2 vyšší než určuje obsah reg.IX bude přenesen do reg.A. Instrukce LD (IY+FDH),A (HD kód 7D 77 FD) přenese obsah reg.A na adresu o 3 nižší, než jakou udává obsah reg.IY (FD je DK reprezentace decimálního čísla -3). Z uvedeného vyplývá, že před používáním takovéto instrukce musíme napřed stanovit adresu IX nebo IY. Je vhodné ji stanovit tak, aby byla přibližně uprostřed bloku dat, s nimiž budeme často pracovat - v operačních systémech to bývá např. oblast dat systémových proměnných apod.

Pro lepší porozumění odchylkám v indexovaném adresování si uvedeme pár příkladů (d je odchylka):

(IX+d)	odchylka od adresy (IX) decimálně
(IX+7FH)	+127
(IX+09H)	+9
(IX+00H)	=(IX), tedy nulová odchylka
(IX+FFH)	-1
(IX+F0H)	-16
(IX+C0H)	-48
(IX+80H)	-128

V zápisech assemblerových programů se můžete někdy setkat s tvarem (IX-2), kde číslo -2 přímo ukazuje hodnotu záporné odchylky. Pro vlastní zápis do paměti v HD tvaru si je však musíte převést na DK číslo FEH. Některé generátory strojového kódu nám usnadňují zápis DK čísel právě tímto způsobem. M.j. umožňují vkládat čísla i v jejich binárním tvaru, i v kódu HD či tvaru decimálním. Proto se neděste toho, že byste při vlastním programování museli vše neustále přepočítávat a pro samý přepočet by vám na vlastní tvorbu programu už nezbyl čas. Je však nezbytně nutné pochopit, co, proč a jak se v počítači při provádění jednotlivých instrukcí děje - bez toho by vám nakonec jakýkoli sebelepší generátor nebyl nic platný. Proto - nepolevujte!

Relativní adresování

Tento mód se vztahuje pouze na jeden typ instrukcí, kterým se říká relativní skoky (angl.relative jumps). Jejich mnemonika je JR XX (HD kód 18 XX), kde XX je odchylka zadaná DK číslem (další užití tohoto typu čísel!). Instrukce je dvoubajtová - 1.bajt je operační kód, 2.obsahuje odchylku (displacement), opět v rozsahu -128 až +127. Odečet odchyky má poněkud jiné "startovní" podmínky než v instrukcích indexovaného adresování, kde odečet začíná od adresy určené obsahem reg.IX nebo IY, přičemž adresa (IX) či (IY) sama reprezentuje nulovou odchylku.

Právě tato nulová odchylka relativních skoků leží - pozor, pamatovat! - na adrese, která je ihned za posledním (tedy druhým) bajtem instrukce relativního skoku - což je adresa o 2 vyšší, než adresa, na níž instrukce začíná. Věc si osvětlíme příklady. Stanovme si, že naše instrukce leží třeba na adrese 1000 decimálně:

Instrukce	Skok na adresu
JR 09H	1000+2+9=1011
JR 7FH	1000+2+127=1129
JR 00H	1000+2+0=1002
JR FEH	1000+2-2=1000
JR F0H	1000+2-15=987
JR 80H	1000+2-128=874

Leží-li tedy instrukce relativního skoku na adrese 1000, můžeme z ní "odskočit" na jakoukoli z okolních adres v intervalu 874-1129. Povšimněme si jedné zvláštnosti, které se ve svých porgramech zásadně vyvarujte - instrukce JR FEH se po provedení vrací sama na sebe a provede se znova a opět - program se tak dostane do nekonečné smyčky, z níž není úniku!

Poněkud beze smyslu je zařazení instrukce JR 00 - program pokračuje hned na další adrese za instrukcí. Kdyby tedy instrukce JR 00 v programu nebyla, nic by se nestalo. Je tu však možné uplatnění jedné z programových fines, kdy programově (odjinud) měníme obsah bajtu odchylky, což má za následek skoky do různých částí programu z téhož místa. V tom případě by uplatnění instrukce JR 00 mohlo mít svůj smysl. Uplatnit by se mohla i při ladění časovacích smyček a jiných časově kritických rutin.

Užití relativních skoků je významné ze dvou hledisek. Oproti přímému skoku JP XXXX, který je tříbajtový, je relativní skok dvoubajtový - jeho užitím šetříme paměť (ale nikoli čas!). A nakonec - potřebujeme-li relokovat nějaký program v paměti (celý jej přemístit na jiné adresy), nemusíme odchylky relativních skoků měnit! Zatímco u přímých skoků a řady dalších instrukcí je po relokaci nutno provést adresovou transpozici.

Tak jsme si představili všech 10 adresovacích módů Z80. Příklady jejich programového uplatnění najdete na konci této kapitoly i ve všech experimentech této učebnice.

Instrukce přenosu 16tibitových dat

Již jsme se seznámili se všemi skupinami instrukcí 8mibitového přenosu dat. U přenosu 16tibitového jsme poznali instrukce typu LD. K nim dále patří skupina instrukcí, které pracují s přenosem obsahu mezi registry a zásobníkem - PUSH a POP.

Zásobník (Stack) je oblast paměti, jejíž spodní adresa je adresována obsahem 16tibitového registru SP (Stack Pointer). Na adresy tohoto zásobníku můžeme instrukcemi PUSH ukládat a z něj instrukcemi POP odebírat 16tibitová data (2 bajty). Příklad:

Registr SP jsme (instrukcí LD SP,65100) nastavili na adresu 65100. Dále si ukážeme, co se bude dít v zásobníku při použití instrukcí PUSH a POP:

SP před instrukcí	Instrukce	SP po po instrukci	Obsah zásobníku
65100	PUSH AF	65098	FA0000000...
65098	PUSH BC	65096	CBFA00000...
65096	PUSH DE	65094	EDCBFA000...
65094	PUSH HL	65092	LHEDCBFA0...
65092	POP HL	65094	EDCBFA000...
65094	POP DE	65096	CBFA00000...
65096	POP BC	65098	FA0000000...
65098	POP AF	65100	000000000...

Z příkladu vyplývá několik zásadních pravidel:

1 - Do zásobníku ukládané 2 bajty jsou uloženy tak, že napřed je uložen vyšší a potom nižší bajt párového registru nebo registrů A a F.
2 - Po uložení 2 bajtů se obsah registru SP i spodní adresa zásobníku vždy sníží o 2.
3 - Po každém odběru se obsah registru SP i spodní adresa zásobníku vždy zvýší o 2.
4 - Ze zásobníku odebírané bajty jsou odebírány podle pravidla: co šlo první dovnitř, jde poslední ven. Pokud jde o registry, napřed se naplní nižší, pak vyšší.

Kromě instrukcí v příkladu uvedených obsahuje soubor Z80 ještě instrukce PUSH a POP pro registry IX a IY. Příklad je sestaven tak, aby byl co nejnázornější - proto jsou v zásobníku jednotlivá písmena registrů - avšak jen ve funkci symbolů. Ve skutečnosti je to tak, že jakmile jednou uložíme obsah registru do zásobníku, přestává mezi uloženými bajty a registry AF, BC, DE, HL, IX či IY existovat jakákoli spřízněnost. Díky tomu můžeme pomocí těchto instrukcí vzájemně měnit obsahy uvedených registrů. Tak chceme-li vyměnit obsahy registrů třeba mezi DE a BC (pro takovou výměnu Z80 žádnou instrukci nemá), zařadíme za sebe instrukce takto:

```
       PUSH BC------!
!------PUSH DE      !
!------POP BC       !
       POP DE-------!
```

Původní obsah reg.BC bude v DE a původní obsah reg.DE přejde do BC. Profesionální programátoři provádějí se zásobníkem mnohá kouzla - např. jej v některých průbězích programu používají jako oblasti proměnných, změnami obsahu reg.SP vytvářejí několik různých zásobníků v paměti pro různá využití jejich obsahu nebo pro různé funkce částí programu atd.

Zásobník je podřízen jedné automatické funkci Z80, která je na jednu stranu nezbytná, na druhou stranu je zdrojem mnoha programátorských strastí, které plynou z nepozornosti. Nad tím, co všechno se během chodu programu odehrává v zásobníku, je někdy těžko udržet přehled. Kdykoli je totiž volána nějaká rutina instrukcí CALL XXXX, je do zásobníku uložena tzv. adresa návratu, na niž se program vrátí poté, co v programu narazí na odpovídající instrukci RET. Obě instrukce si probereme později; v Basicu mají svůj ekvivalent v příkazech GO SUB n a RETURN. Pokud použijete v Basicovém programu RETURN bez toho, že by mu předcházel GO SUB n, program se zastaví a chybové hlášení vás na tento nedostatek upozorní. Ale když v assembleru porušíte toto pravidlo, instrukce RET si prostě "odebere", co v zásobníku zrovna na spodním konci je a program skočí na adresu určenou obsahem dvou odebraných bajtů. Obvykle následuje krach programu.

Se zásobníkem je nutno pracovat velice obezřetně, protože je jednou z nejčastějších příčin programových kolapsů, pramenících z našich chyb.

Instrukce výměn obsahu registrů

Vyjmenujme si je všechny hned na začátku: EX AF,AF´; EX DE,HL; EX (SP),HL; EX (SP),IX; EX (SP),IY a EXX.

EX je zkratkou anglického slova exchange (výměna). Funkci instrukcí si vysvětlíme na instrukci EX DE,HL:

Obsahy registrů před provedením EX DE,HL		Obsahy registrů po provedení EX DE,HL
00	D	02
01	E	03
02	H	00
03	L	01

Instrukce s adresou (SP) danou obsahem registru SP provádějí výměnu mezi obsahem dvou spodních adres zásobníku a párovým registrem v instrukci užitým. Např. při EX (SP),HL se vymění obsah adresy (SP) s obsahem reg.L a adresy (SP+1) s reg.H! Ale obsah samotného reg.SP se nezmění! Mění se jen bajty jím adresované!

Instrukce EX AF,AF´ provádí výměnu mezi uvedenými registry obou registrových bank Z80.

Instrukce EXX provádí výměnu mezi registry BC,DE,HL z banky 0 a stejnojmennými registry BC´,DE´,HL´ banky 1 mikroprocesoru.

Oba poslední typy instrukcí výměn jsou zároveň jedinými, které provádějí operace s registry druhé registrové banky Z80. Zde jedno velmi důležité upozornění - každý počítač okupuje pro operace probíhající v jeho systému některé z registrů! U každého typu to bývá jiné. Použití těchto registrů se musíte ve svých programech vyhnout, protože by počítač začal "zlobit", nebo by zcela zkolaboval. Někdy některé z těchto "zakázaných" registrů lze přechodně použít za určitých podmínek - to však vyžaduje perfektní znalost systému počítače. I monitory a generátory strojového kódu mívají drobná omezení, která se dozvíte z jejich manuálu. V každém případě se snažte všechna tato omezení zjistit - jinak byste mohli zcela zbytečně bádat nad tím, proč vám váš program nefunguje, přestože po stránce teoretické je naprosto v pořádku.

Cvičení

Proveďte si je, protože vás včas upozorní na to, čemu jste v textu neporozuměli nebo tomu nevěnovali patřičnou pozornost. Správné odpovědi najdete hned za posledním cvičením.

1. Určete 8mibitový dvojkový komplement následujících 8mibitových binárních čísel:

a. 00000001	e. 00001110
b. 11011010	f. 10000000
c. 01010101	g. 11111111
d. 11101110	

2.a. Jaké je nejvyšší kladné a nejvyšší záporné (v absolutní hodnotě) decimální číslo v reprezentaci 8mibitového DK?
b. Jako v bodě a., ale pro 16tibitovou reprezentaci.

3. Určete decimální čísla reprezentovaná následujícími 8mibitovými DK čísly:

a. 01111000	e. 11110011
b. 10100011	f. 01010100
c. 00000011	g. 11011001
d. 11111111	

4. Určete 8mibitovou DK reprezentaci těchto decimálních čísel:

a. 1	e. 128
b. 16	f. 121
c. -16	g. -90
d. -128	

5. Následující skokové instrukce JP XXXX nahraďte instrukcemi relativního skoku JR XX (čísla jsou decimální), přičemž odchylku zapište vždy decimálně i hexadecimálně.

	adresa uložení	instrukce
a.	10000	JP 10020
b.	11111	JP 11240
c.	30	JP 0
d.	30	JP 39
e.	65500	JP 65374

6. Pro uvedené assemblerové instrukce vyhledejte v tabulkách (kde potřeba dopsat čísla, dopište) jejich HD kód:

a. LD A,B	g. LD SP,HL
b. JR 127	h. LD BC,0109H
c. LD A,(IX+06)	i. LD (1030H),BC
d. LD (IX+06),A	j. LD IX,(1000H)
e. LD (1234H),A	k. PUSH BC
f. LD (IX+09),33	l. POP IX

7. Zjistěte z tabulek, zda jsou následující instrukce obsaženy v instrukčním souboru Z80:

a. LD AF,BC
b. LD B,(BC)
c. LD (BC),B
d. LD IX,IY
e. LD HL,BC
f. LD (1234H),56H
g. LD (1234H),B
h. LD (DE),45H
i. PUSH 1234H
j. POP SP

Odpovědi

1.
a. 11111111
b. 00100110
c. 10101011
d. 00010010
e. 11110010
f. neexistuje
g. 00000001

2.
a. 01111111=+127; 10000000=-128
b. 0111111111111111=(2**15-1)=32767
1000000000000000=(-2**15)=-32768

3.
a. 120
b. -93
c. 3
d. -1
e. -13
f. 84
g. -39

4.
a. 00000001
b. 00010000
c. 11110000
d. 10000000
e. neexistuje
f. 01111001
g. 10100110

5.
a. JR 18 JR 12H
b. JR 127 JR 7FH
c. JR -32 JR E0H
d. JR 7 JR 07H
e. JR -128 JR 7FH

6.
a. 78
b. 18FB
c. DD7E06
d. DD7706
e. 323412
f. DD360933
g. F9
h. 010901
i. ED433010
j. DD2A0010
k. C5
l. DDE1

7. Odpověď je jednoduchá - žádná z těchto instrukcí není se Z80 proveditelná. Pamatujte si, že soubor instrukcí má své meze a nelze tedy užívat ekvivalenty jednotlivých instrukcí ve všech jejich možných kombinacích. Uvedené instrukce lze provést jen jejich rozložením na více existujících instrukcí.

EXPERIMENT č.1

Procvičení indexovaného adresování. V komentářích budeme dále používat schématické vysvětlení tvorby programu bez podrobného popisování vlastní funkce jednotlivých instrukcí.

```
0100  010300          LD BC,0003      ;3 bajty pro každou řádku
0103  FD212001        LD IY,0120H     ;1.adresa tabulky
0107  FD 7E00   LOOP:LD A,(IY)        ;Sloupec 1 do reg.A
010A  B7              OR A            ;Je reg.A=0?
010B  280A            JR Z,END        ;ANO, pak konec
010D  FD8601          ADD A,(IY+01)   ;NE, přičtení sloupce 2
0110  FD7702          LD (IY+02),A    ;Součet do sloupce 3
0113  FD09            ADD IY,BC       ;IY je další řádka tabulky
0115  18F0            JR LOOP         ;Skok na adr.LOOP; opakování
0117  C9         END:RET              ;Návrat
```

KROK 1

Zkontrolujte správnost svého zápisu do počítače.

KROK 2

Tento program sečítá vždy na jedné řádce ve dvou sloupcích umístěná čísla a výsledek každého součtu uloží na téže řádce do sloupce 3. Tabulka je v paměti uložena od adresy 0120H:

	adresa	sloupec 1	sloupec 2	sloupec 3
řádka 1	0120H	01	02	?
řádka 2	0123H	10	04	?
řádka 3	0126H	23	13	?
řádka 4	012CH	00 (bajt 00 zastaví program)		

Jedná se tedy v podstatě o programovou simulaci stavby takovéto tabulky. Její rozsah záleží na umístění bajtu 00 v paměti. Čísla sloupců 1 a 2 musíme samozřejmě uložit do paměti předem na adresy uvedené ve sloupci adres a adresy vždy o 1 vyšší, než je adresa ve sloupci adres uvedená. Výsledek prvního součtu bude na adrese 0122H, druhého na adrese o 3 vyšší atd.

KROK 3

Uložte čísla tabulky do paměti a vyzkoušejte funkci programu prohlédnutím obsahu adres s výsledky součtů. Z uvedené aplikace registru IY je patrno, jak pohodlná je manipulace s daty dvourozměrné tabulky při jeho použití. Řádka tabulky je specifikována obsahem reg.IY, zatímco data jsou manipulována pomocí jeho odchylky.

EXPERIMENT č.2

V tomto experimentu si ukážeme, že při sestavování programu, který má plnit zadaný úkol, můžeme postupovat různými způsoby. Jinými slovy - k jednomu cíli vede větší počet cest. Jednou z velmi efektních technik tvorby programu je taková, během níž dovedně řadíme instrukce tak, že po spuštění programu jsou některé z nich modifikovány (přepisovány) jinými instrukcemi. Upozornění: Tato technika je velmi náročná a nikdo vám nemůže zaručit, že ji ovládnete. Na druhou stranu se však lze obejít i bez ní. Self-modification, jak se tato technika jmenuje anglicky, není v profesionálních programech používána nijak hustě. Její použití je limitováno schopností člověka udržet si orientaci v programu. Nevýhodou této techniky je, že rozbor takového programu je nesmírně obtížný, ne-li nemožný, což stoupenci této techniky považují za výhodu, protože tak docílí jednoho z nejvyšších stupňů utajení programu. Problém je ovšem v tom, že tak leckdy utají program i sami před sebou. Techniku samu lze využít pochopitelně jen v pamětech typu RAM. V pamětech typu ROM se s ní tedy setkat nemůžete. Nespornou výhodou techniky bývá rychlost, s jakou program pracuje, i ušetření značné části paměti.

Rutina A1

```
02DF  010700          LD BC,0007     ;Počet sloupců na řádce
0300  1E06            LD E,06        ;Čítač počtu řádek
0302  FD218003        LD IY,0380H    ;Adresa 1.bajtu řádky
0306  211103          LD HL,0311H    ;Adresa bajtu odchylky
                                     ;v instrukci ADD (IY+d)
0309  3600     řádka:LD (HL),00      ;Inicial.odchylky na adr.(HL)
030B  3E00            LD A,00        ;     "     reg.A
030D  1606            LD D,06        ;Počet sčítanců na řádce
030F  FD86d    sloup:ADD A,(IY+d)    ;Přičtení; d se mění programem!
0312  00              NOP            ;Žádná operace ("volný bajt")
0313  34              INC (HL)       ;Zvýšení odchylky d na adr.(HL)
0314  15              DEC D          ;Snížení čítače řádky o 1
0315  20F8            JR NZ,sloup    ;Když D není 0, další součet
0317  FD7706          LD (IY+06),A   ;Když D=0, součet do sloupce 7
031A  FD09            ADD IY,BC      ;IY na 1.bajt další řádky
031C  1D              DEC E          ;Snížení čítače řádek o 1
031D  20EA            JR NZ,řádka    ;Když E není 0, na další řádku
031F  C9              RET            ;Když E=0, návrat
```

Rutina A2

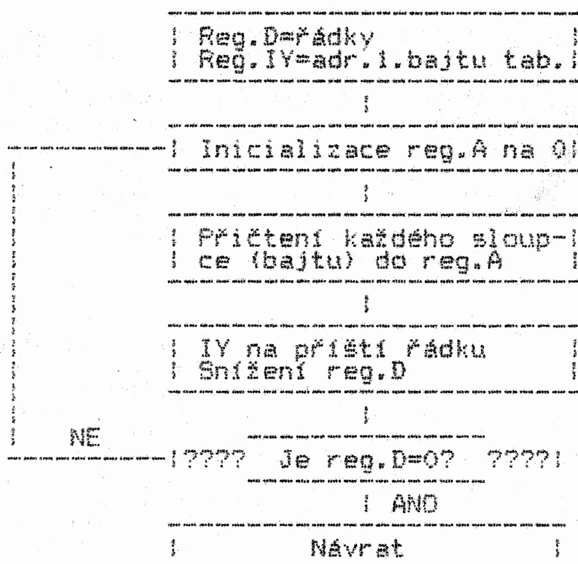
```
0320  010700          LD BC,0007
0323  1E06            LD E,06
0325  FD218003        LD IY,0380H
0329  3E00     řádka:LD A,00
032B  FD8600          ADD A,(IY+00)
032E  FD8601          ADD A,(IY+01)
0331  FD8602          ADD A,(IY+02)
0334  FD8603          ADD A,(IY+03)
0337  FD8604          ADD A,(IY+04)
033A  FD8605          ADD A,(IY+05)
033D  FD7706          ADD (IY+06),A
0340  FD09            ADD IY,BC
0342  1D              DEC E
0343  20E4            JR NZ,řádka
0345  C9              RET
```

Rutina A3

```
0360  1E06            LD E,06
0362  FD218003        LD IY,0380H
0366  3E00     řádka:LD A,00
0368  1606            LD D,06
036A  FD8600   sloup:ADD (IY)
036D  FD23            INC IY
036F  15              DEC D
0370  20F8            JR NZ,sloup
0372  1D              DEC E
0373  FD7700          LD (IY+00),A
0376  FD23            INC IY
0378  10EC            JR NZ,řádka
037A  C9              RET
```

KROK 1

Při prvním pohledu na tyto tři programy zjistíme, že co do výsledku své činnosti jsou naprosto shodné. Do paměti je umístěna tabulka šesti řádek v šesti sloupcích od adresy 0380H. Podobně jako v experimentu předešlém, i tento sečítá čísla na jednotlivých řádkách a výsledek umisťuje vždy do jejich posledního sloupce. Programy se liší rozsahem paměti, kterou zabírají, i časem provedení. Pro všechny tři programy platí tento vývojový diagram:



RUTINA A1

Hlavním algoritmem rutiny je modifikace bajtu odchylky d přímo v instrukci ADD A,(IY+d). Reg.HL je naplňován obsahem adresy, na níž leží třetí bajt instrukce ADD, tedy odchylkou. Nejdříve je odchylka inicializována na nulu, vynulován je i akumulátor. V průběhu čítání sloupců je obsah adresy (IY+d) přičítán k akumulátoru a odchylka vždy zvýšena o 1 instrukcí INC (HL). Poté, co byla všechna čísla ve sloupcích na řádce sečtena, čítač sloupců se vynuluje. To je detekováno instrukcí JR NZ, která způsobí převedení řízení programu na výpočet další řádky.

Jak jsme se zmínili, program modifikuje sám sebe. Týká se to těchto čtyř instrukcí: LD HL,0311H; LD (HL),00; ADD A,(IY+d); INC (HL).

Dvě z těchto instrukcí mění program, který vlastně zvažuje sám sebe jako svá data. Znovu opakujeme, že užití této techniky si mohou dovolit jen absolventi "vyšší dívčí" - ale i ti mají co dělat, aby tvorbu programu s použitím této techniky přežili ve zdraví.

RUTINA A2

Zde jsou odchylky řazeny postupně za sebou v jednotlivých instrukcích ADD A,(IY+d). Jak vidno, tento postup zabere podstatně víc paměti než předchozí, ale na druhou stranu je velmi přehledný. Pokud bychom pracovali s rozlehlejší tabulkou, stal by se však tento způsob řazení instrukcí neudržitelný.

RUTINA A3

Zde je zvolen zcela zvláštní způsob pojímání dvourozměrné tabulky. Celá je převedena do jednorozměrného řetězce, jehož indexem je registr IY. Data jsou tedy pojímána sekvenčně. Podmínkou užití tohoto způsobu načítání dat je, že data musejí být v paměti řazena za sebou. Pokud by data byla v paměti všelijak "rozházená", tento způsob by nebyl k ničemu.

Zařadíme si nový termín z oblasti výpočetní techniky - flexibilita. Označuje se jí stupeň obtížnosti modifikace programu. Čím je tento stupeň nižší, tím je flexibilnější, modifikovatelnější (a také "čitatelnější"). Sebe modifikující programy mají flexibilitu nejnižší, často dokonce nulovou.

Závěrem lze obecně říci, že čas a prostor jsou u programů nepřímo úměrné. Čím více prostoru paměti spotřebujeme, tím je program pomalejší a naopak (ale nemusí to platit vždy!).

KROK 2

Uložte do počítače rutinu A1 a krokujte ji pomocí vašeho monitoru strojového kódu. Největší pozornost věnujte adrese 0311H, na níž je uložena odchylka v instrukci ADD A,(IY+d). Pro zpracování dat tabulky vložte do paměti tato data:

Adresa	Sl.1	Sl.2	Sl.3	Sl.4	Sl.5	Sl.6	Sl.7	
0380	01	02	03	04	05	06	X	řádka 1
0387	02	02	02	02	02	02	X	řádka 2
038E	01	03	01	03	01	03	X	řádka 3
0395	03	03	03	03	03	03	X	řádka 4
039C	08	08	01	01	08	08	X	řádka 5
03A3	04	04	04	04	08	08	X	řádka 6

Písmeno X znamená místo uložení výsledku součtu.

KROK 3

V tomto experimentu jste se poprvé setkali s tím, že programování je stejně věda jako umění. Způsobů, jak napsat nějaký program pro daný účel, je vždy víc. Záleží na mnoha okolnostech, který z možných způsobů zvolíme. Abyste při tvorbě programu nepropadli pocitům méněcennosti, můžeme vám říci, že i ti nejlepší programátoři přepisují své assemblerové programy třeba i dvakrát nebo třikrát. Je to proto, že se opravdu jedná o tvorbu, tedy tvůrčí proces, jehož jedním z hlavních znaků je, že v něm neplatí nic absolutního.

EXPERIMENT č.3

V něm si ukážeme užití instrukcí PUSH, POP a instrukcí výměn.

```
0130          PUSH AF      ;Obsah reg.AF do zásobníku
              PUSH BC      ;Obsah reg.BC do zásobníku
              PUSH DE      ;Obsah reg.DE do zásobníku
              PUSH HL      ;Obsah reg.HL do zásobníku
              EX AF,AF'    ;Výměna obsahu reg.AF a AF'
              EXX          ;Výměna obsahu párových reg.
              POP HL       ;Obsah zásobníku do reg.HL
              POP DE       ;Obsah zásobníku do reg.DE
              POP BC       ;Obsah zásobníku do reg.BC
              POP AF       ;Obsah zásobníku do reg.AF
              RET          ;Návrat
```

KROK 1

Pro zápis tohoto jednoduchého programu užijte monitor strojového kódu, do nějž budete zapisovat HD kódy jednotlivých instrukcí. Tyto kódy jsme do programu záměrně nezařadili, abyste se naučili používat převodních tabulek. Po zapsání HD kódů si monitorem zjistěte, jestli listing programu odpovídá uvedenému zápisu.

KROK 2

Zapište program a zkontrolujte správnost jeho zápisu.

KROK 3

Nyní je na vás, abyste zvolili adresu zásobníku, čili obsah registru SP. Pro tento účel jsou v instrukčním souboru Z80 zahrnuty tyto instrukce: LD SP,HL; LD SP,IX; LD SP,IY; LD SP,NNNN; LD SP,(XXXX). Kteroukoli z těchto instrukcí umístěte na začátek programu. Pokud si přesně nevzpomínáte na funkci zásobníku, vraťte se k předchozímu výukovému textu.

KROK 4

Nyní musíte nastavit obsahy všech zúčastněných registrů. Zkuste třeba tyto hodnoty: A=01; F=02; B=03; C=04; D=05; E=06; H=07; L=08. Obsahy těchto registrů a reg.SP můžete nastavit přímo monitotem.

KROK 5

Krokujte programem a pečlivě sledujte, co se bude dít v registru SP a s obsahem zásobníku. Uvidíte, že vše probíhá tak, jak uvedeno v předchozím výukovém textu.

KROK 6

Zkuste změnit obsahy registrů, abyste se ujistili o tom, že funkce přenosu mezi registry a zásobníkem funguje, jak náleží.

KROK 7

Během krokování jste měli možnost si povšimnout, co se stane po provedení dvou instrukcí výměn. Ze všeho vyplývá, že když registry banky 1 převedeme do banky 0, můžeme je nastavit na požadované hodnoty - např. použitými instrukcemi typu POP a tyto registry opět uschovat do banky 1. Na tomto místě opět upozorňujeme, že je třeba se mít na pozoru před tím, abychom v těchto výměnách nepoužili, resp. nezměnili registry používané vlastním systémem počítače. Registry banky 1 můžeme někdy pojmout jako určitý "zásobník", v němž si uschováme číselné parametry pro jejich případné pozdější užití.

KROK 8

Protože Z80 má omezený počet registrů a my v programu pracujeme neustále s hromadou čísel, slouží nám instrukce PUSH a POP pro dočasné uschování číselných parametrů, jejichž hodnoty budeme v programu ještě potřebovat.

****

KAPITOLA 5

Skoky, volání a návraty

****

Všechny tyto instrukce převádějí řízení programu na jiné adresy. Lidově řečeno - po provedení těchto instrukcí program "odskočí jinam".

Programové řízení

Program je v podstatě soustavou smysluplně řazených bajtů umístěných na po sobě jdoucích adresách paměti. Těmito bajty mohou být buď instrukce nebo data. Instrukcemi řídíme běh programu, zatímco data nám slouží jako číselné parametry instrukcí.

Zde je třeba si uvědomit, že počítač neví, které bajty jsou instrukcemi a které daty. Dáme-li počítači příkaz, aby začal provádět instrukce od určité adresy, počítač automaticky dekóduje jednotlivé po sobě jdoucí bajty a snaží se je číst jako instrukce. Pokud by se nám programové řízení "zaběhlo" do oblasti, v níž jsou uložena data, počítač by je bez nejmenšího uzardění "luštil" jako instrukce - pochopitelně by na to brzy doplatil. Z uvedeného vyplývá jeden problém, s nímž se setkáme vždy, kdykoli budeme chtít "rozluštit" nějaký cizí program. Stejně jako počítači, i nám by se mohlo stát, že bychom oblast dat četli bludně jako instrukce. Zmíněný problém je jedním z největších kamenů úrazu vždy, kdykoli jsme postaveni před nutnost relokace nějakého programu. Pak nezbývá než program analyzovat opravdu detailně, abychom mohli přesně určit, které bajty jsou datové a které instrukcemi.

K pochopení programového řízení je dále nutno si plně uvědomit známou skutečnost, že instrukce se provádějí postupně směrem od nižších k vyšším adresám. Tato posloupnost je přerušena pouze instrukcemi skoků, volání a návratů, které převedou řízení programu na jinou danou adresu. Od této adresy je program prováděn opět posloupně, dokud znova nenarazí na jeden ze tří typů uvedených instrukcí (nebo instrukce přerušení, které jsou však ještě daleko před námi).

Programové řízení je zajišťováno registrem PC - Program Counter (programový čítač). Ten v průběhu programu obsahuje vždy tu adresu, na niž bude programové řízení převedeno. To znamená, že po každém dekódování každé instrukce se obsah registru PC zvýší o tolik, kolik bajtů dekódovaná instrukce obsahuje.

V případě, že program narazí na instrukci skoku (JP XXXX, JR XX apod.), obsah registru PC se těsně před provedením skoku naplní adresou, na niž má být skok proveden. Je-li skok relativní, provede se odpovídající výpočet skokové adresy a výsledek se převede do registru PC, s následným skokem.

V případě, že program narazí na instrukci volání (CALL XXXX), provede se stejná operace jako v případě skoku, ovšem s tím rozdílem, že do zásobníku se uloží adresa návratu, která je o 1 vyšší než je adresa posledního bajtu instrukce volání. Poté, co program narazí na instrukci návratu (RET), převede se automaticky adresa návratu ze zásobníku do registru PC. To bude mít za následek, že programové řízení přejde na instrukci, která je umístěna hned za instrukcí volání. Zde ovšem platí pravidlo, které musíte nutně dodržet - na spodních dvou adresách zásobníku musí být patřičná adresa návratu vždy přesně v tom momentu, kdy program narazí na instrukci RET. Pokud by tomu tak nebylo, program by byl převeden někam jinam (na adresu, která by byla stanovena obsahem dvou bajtů zrovna se vyskytujících na spodních adresách zásobníku).

Sám registr PC je uživateli nepřístupný, resp. instrukční soubor Z80 neobsahuje žádnou instrukci, jejíž součástí by registr PC byl. Obsah registru PC však můžeme ovlivnit např. instrukcí RET s předem nastavenými obsahy dvou spodních bajtů zásobníku, které můžeme stanovit přímo instrukcemi, jež v sobě obsahují registr SP.

Nepodmíněné skokové instrukce

Mnemoniku těchto instrukcí už známe. První z nich je JP XXXX, kde XXXX je adresa, na niž bude převedeno programové řízení (adresou XXXX se naplní registr PC). Nic jiného, než že svým 2.a 3.bajtem naplní reg.PC, tato tříbajtová instrukce neprovádí.

Druhá - JR XX má tentýž efekt. Liší se jen tím, že je dvoubajtová a XX neobsahuje absolutní adresu skoku, ale odchylku (displacement) ve dvojkově komplementární reprezentaci - viz podrobné vysvětlení v předchozím textu.

K typu instrukcí JP patří ještě instrukce s registrovým nepřímým adresováním: JP (HL), JP (IX), JP (IY). Adresa skoku je zde dána obsahem zúčastněných registrů.

Malá poznámka ke skokům a jejich začleňování do programů: Snažte se programy psát s co nejmenším počtem skokových instrukcí. Program protkaný skokovými instrukcemi je značně nepřehledný, zamotává se. Když už je musíte použít, sestavujte strukturu programu tak, abyste nemuseli používat skoky JP, ale JR. Nejen že tím šetříte paměť, ale při relokaci programu si nemusíte dělat starosti s přepisováním absolutních adres instrukcí JP. Výjimku tvoří užití instrukcí JP při testech parity a polarity (stavové indikátory P/V a S) - instrukce JR tyto testy provádět nemohou.

Strukturalizovat program se snažte vytvářením jednotlivých podprogramů (subrutin) s předem stanovenými funkcemi tak, že jejich hlavní bloky budou volány instrukcemi CALL XXXX (s následnými instrukcemi návratů). Nepromyšlená struktura programu vás přinutí používat dlouhé přeskoky programem a nakonec vám nezbyde, než do něj zařazovat stále rostoucí (a strukturu zatemňující) počet instrukcí JP.

Nejlepší systémové a užitkové programy jsou relokovatelné. S tím, aby program byl relokovatelný, musí jeho tvůrce počítat předem a tvorbu programu tomuto požadavku podřídit. Zkuste to taky - a uvidíte, že vás tento požadavek nakonec přinutí sestavit program ve strojovém kódu tak, že se bude blížit určitému ideálu. Teoreticky lze říci, že každý program by mohl být sestaven tak, že by byl plně relokovatelný. Že tomu tak není, je proto, že sestavení dobré struktury programu je velmi obtížné, ale i proto, že programátoři se nechávají tvorbou programu příliš živelně unášet a pak už se jim nechce jej předělávat. Uvedený požadavek se samozřejmě nevztahuje na tvorbu herních programů pro mikropočítače - tyto programy většinou zabírají celou paměť počítače, takže je ani není kam (ale i proč) relokovat. Zato nerelokovatelnost systémového programu je prakticky i teoreticky neopodstatnitelná.

Podmíněné skoky a stavové indikátory

"Letem světem" jsme se s těmito skoky i indikátory seznámili již v první kapitole. Teď si je probereme podrobně.

Existují tři typy instrukcí, které testují stav stavových indikátorů (budeme si je dále označovat zkratkou SI) - podmíněné skoky, instrukce volání a návratů. Ty poslední dvě si probereme hned po skocích.

Z80 má v reg.F celkem 6 SI. Instrukcemi můžeme testovat 4 z nich. Dva zbylé slouží interním testům mikroprocesoru. Ve skokové instrukci uvedená podmínka testuje, zda je ten který testovaný SI ve stavu log.0 nebo log.1.

Mikroprocesor Z80 obsahuje tyto stavové indikátory:

CARRY FLAG (indikátor přenosu) - CY

je ovlivňován především instrukcemi provádějícími součet, odečet, bitovou rotaci a posuv. Tyto operace provádí celá řada instrukcí.

Při součtu je tento SI (budeme jej dále značit CY - aby se nám nepletl s reg.C) ve stavu log.1 vždy, když dojde k přeplnění akumulátoru, do nějž je ukládán výsledek součtu. Přeplnění znamená, že akumulátor by po něm měl mít obsah minimálně 100000000, což nejde, protože to je celkem 9 bitů, a registr jich má jen 8. Onen 9. bit tedy "přepadne horem" a "zmizí". V součtech vícebajtových však můžeme tento bit přenést do vyššího bajtu a přičíst jej k jeho nejnižšímu bitu. Odtud název indikátor přenosu.

Při odečtu se vše odehrává "v opačném směru". CY=1 i tehdy, když výsledek odečtu je (by měl být) menší než 0. Toho se opět využívá ve vícebajtových odečtech, kdy "dolem přepadlý" bit je přenesen a odečten od nejvyššího bitu nižšího bajtu.

Jak víme, přičteme-li k registru obsahujícímu FFH číslo 1, jeho obsah se nastaví na nulu. Odečteme-li od registru s obsahem 0 číslo 1, jeho obsah bude FFH. V obou těchto případech bude CY ve stavu log.1.

Nedojde-li při operaci k přenosu "9.bitu", je CY ve stavu log.0. Instrukce rotace a posuvu pojednávají CY jako 9. bit akumulátoru. Probereme si je zevrubně v jedné z dalších kapitol.

CY je jediný SI přímo ovlivnitelný dvěma instrukcemi:

SCF (Set Carry Flag) nastavuje CY do stavu log.1
CCF (Complement Carry Flag) jej převrací do opačného stavu, než v jakém byl před provedením instrukce. Např. je-li CY=1, po CCF bude CY=0, po opětném provedení CCF bude CY=1

ZERO FLAG (indikátor nuly) - Z

Jeho logický stav je ovlivňován mnoha instrukcemi. U tohoto SI si musíte zapamatovat, že Z=1, když výsledkem testované operace je nula (mimo dále uvedené výjimky). A naopak - pokud je výsledek operace různý od nuly, je Z=0! Začátečníci si to zpočátku často pletou. Uvědomte si, že Z nám indikuje nulový výsledek. Stav log.0 indikátoru Z berte jako "tak o tohle nemám zájem, nic se neděje". Zato log.1 (výsledkem operace je nula) říká: "pozor, je to tady, zvyšuju napětí!".

Nejen u tohoto indikátoru, ale i u všech ostatních si musíte v tabulkách pečlivě probrat (a při programování je mít stále po ruce), jaké instrukce ovlivňují které SI, jinými slovy - kdy a čím lze co a jak testovat a co a jak mění stavy kterých SI. I poměrně zkušení programátoři opomíjejí mnohé z poměrně širokého spektra testovacích možností při provádění různých programových operací a zůstávají jen u několika málo základních testů, které si pamatují z doby, kdy se před lety učili programovat. Díky tomuto přístupu se však mohou dopustit i nepříjemných chyb. Jednou z těch častějších je opomenutí toho, že zatímco instrukce snížení obsahu jednoho registru (např.DEC C) ovlivňují Z, pak ty, jež snižují obsah párového registru (např.DEC BC) indikátor Z neovlivňují. Chyba nastává tehdy, kdy programátor za instrukci DEC BC zařadí test nuly, který se samozřejmě nemůže provést, čímž programátor uvádí sám sebe do nemilého omylu.

Kromě zmíněné instrukce DEC r ovlivňují indikátor Z např. instrukce INC r a typu BIT (testuje stav jednoho z bitů registru nebo místa v paměti), CP (porovnává obsahy dvou bajtů) atd. Indikace Z si budeme ve výukových textech všímat u všech dalších instrukcí, ke kterým se vztahuje.

Zda je indikátor Z ve stavu log.1 nebo log.0, zjišťujeme zařazením podmínky Z nebo NZ v podmíněných instrukcích skoku, volání a návratu. Přičemž si musíme uvědomit, že neprovádíme test stavu indikátoru Z, ale test výsledku předchozí (nebo některé z předchozích) operace, který je indikátorem indikován.

SIGN FLAG (indikátor znaménka - plus, mínus) - S

Jedná se o test stavu nejvyššího bitu bajtu (S jej automaticky přebírá) v rámci dvojkově komplementárních čísel. De facto se jedná o test polarity, tedy zjištění, zda testované DK číslo je záporné (jeho nejvyšší bit je ve stavu log.1) nebo kladné (tento bit je ve stavu log.0).

PARITY!OVERFLOW FLAG (parita nebo aritmetické přeplnění) - P!V

Při testu parity zjišťujeme, zda je počet bitů log.1 (tedy i bitů log.0 - bitů je přece osm!) sudý nebo lichý. V případě sudé parity je stav indikátoru log.1, v případě liché log.0. Vztahuje se především k logickým operacím.

Aritmetické přeplnění se indikuje tehdy, je-li výsledkem součtu dvou záporných DK čísel číslo kladné nebo je součet dvou kladných DK čísel záporný. V tom případě bude stav indikátoru log.1. Neporozuměli-li jste, vraťte se k výukovému textu o DK číslech. Je důležité si nesplést přeplnění indikované tímto a CY indikátory. Proveďte si jejich porovnání a rozdíl si zapamatujte. Pomůžeme vám v tom příkladem:

```
1 1 1 1 1 0 1 1   DK reprezentace čísla  -5
1 1 1 1 0 0 0 0   DK reprezentace čísla -16
---------------
1 1 1 0 1 0 1 1   DK reprezentace čísla -21
```

CY=1, protože se jedná o přenos z nejvyššího bitu (do "9.")
V=0, protože nedošlo k aritmetickému přetečení

V=1 např. při součtu DK čísel -80 a -70 (výsledek přesahuje interval -128 až +127).

HALF-CARRY a SUBTRACT FLAGS - H a N

To jsou indikátory polovičního přenosu a odečítání. Jak už bylo uvedeno, tyto dva SI slouží vnitřnímu systému Z80 a programátorovi jsou nepřístupné (nemůžeme je testovat přímo). Indikátor H nás informuje o tom, že v bajtu došlo k přenosu bitu mezi jeho 4. a 5. bitem (v obou směrech). Indikátor N nám oznamuje, že byl proveden odečet. Oba SI jsou důležité zvláště při aritmetických operacích s čísly BCD (viz dále).

Bity SI jsou v registru F umístěny takto:

S Z - H - P|V N C

Znaménko "-" znamená, že bit na jeho místě nemá žádné použití.

Skokové testy

U skoků typu JP můžeme testovat všechny čtyři testovatelné SI:

Instrukce	Skok se provede, když:
JP NZ	Z=0 (výsledek operace není nulový)
JP Z	Z=1 (výsledek operace je nula)
JP NC	C=0 (nedošlo k přeplnění bajtu)
JP C	C=1 (došlo k přeplnění bajtu, resp.přenosu bitu)
JP PO	P|V=0 (lichá parita nebo žádné aritm.přeplnění)
JP PE	P|V=1 (sudá parita nebo arit.přeplnění DK čísel)
JP P	S=0 (DK výsledek je kladné číslo)
JP M	S=1 (DK výsledek je záporné číslo)

Už jsme si řekli, že oproti skokům typu JP nemohou skoky JR testovat všechny SI. Relativní skoky testují pouze dva SI - CY a Z. Jejich podmíněné instrukce jsou: JR NZ, JR Z, JR NC, JR C. Jsou ekvivalentní obdobným JP instrukcím.

Instrukce DJNZ

Zcela zvláštní skokovou instrukcí je DJNZ. Jedná se v podstatě o instrukci podmíněného relativního skoku s opakováním. Počet opakování skoků se inicializuje registrem B. Příklad:

```
0100            LD B,08
0102  odečet:   DEC C
0103            DJNZ odečet
0105            RET
```

Inicializací registru B na adrese 0100H jsme nastavili počet skoků instrukce DJNZ na 8. Kdykoli dojde program na instrukci DJNZ odečet, bude proveden skok na udanou adresu (zde je stanovena návěštím odečet). Adresa se uvádí stejně jako u skoků relativních, tedy odchylkou udanou jednobajtovým DK číslem (zde by DK odchylka byla FDH). Těsně před provedením každého skoku instrukce DJNZ sníží obsah registru B o 1. Skoky se budou provádět, dokud obsah registru B nebude nulový. Z našeho příkladu lze tedy usoudit, že instrukce DEC C bude provedena celkem osmkrát (obsah registru C bude snížen o 8). DJNZ můžeme tedy použít vždy, kdykoli potřebujeme provést nějakou instrukci nebo část rutiny vícenásobně.

Volání a návraty

V přeneseném slova smyslu se jedná o volání jednotlivých částí programové divočiny. Tou divočinou jsou samozřejmě míněny podprogramy. Instrukce volání CALL XXXX má vždy absolutní adresu XXXX a může být buď nepodmíněná nebo podmíněná (tedy obdoba instrukce typu JP). Touto instrukcí můžeme testovat všechny stavy stejně jako u instrukce JP: CALL NZ, CALL Z, CALL NC, CALL C, CALL PO, CALL PE, CALL M, CALL P. Výsledky i důsledky testů jsou tytéž jako u podmíněných instrukcí JP. Každá z instrukcí CALL musí mít (přesněji - měla by mít) někde v programu přiřazenu jednu instrukci návratu RET. Jejich vztah je popsán v textu, zabývajícím se programovým řízením (registrem PC).

I instrukce RET může být buď nepodmíněná (tehdy se návrat k posledně provedené instrukci CALL provede vždy), nebo podmíněná - stejně jako u instrukcí JP a CALL se testy vztahují ke všem čtyřem SI: RET NZ, RET Z, RET NC, RET C, RET PO, RET PE, RET M, RET P. Uvedené instrukce se provedou jen tehdy, jsou-li podmínky v nich uvedené splněny.

Podobná nepodmíněné instrukci CALL je instrukce RST. Víme, že těchto speciálních jednobajtových instrukcí je celkem osm. Tuto informaci si doplníme údajem o tom, že i tyto instrukce musejí

mít někde v programu své dvojče RET. Návrat se provede stejným způsobem jako je tomu v případě instrukcí CALL. Instrukce RET může být pochopitelně i podmíněná.

Naše znalosti si ještě rozšíříme o upozornění, že musíme dát pozor, abychom do zásobníku neuložili takové kvantum adres návratů, že by zásobník začal přepisovat náš program. Stejně tak musíme opět připomenout, že při vlastním programování musíte dát velký pozor na to, aby při provedení instrukce RET tato instrukce odebrala ze zásobníku opravdu tu adresu návratu, která k ní patří. Většina počítačů po své inicializaci sama nastavuje adresu zásobníku na některou z nejvyšších adres paměti. Může se ovšem stát, že tomu tak nebude. Pak by se zásobník mohl ocitnout někde uprostřed vašeho programu, který by pochopitelně ve velmi krátké době zkolaboval. Proto je někdy lepší, když do svého programu zařadíte inicializaci registru SP podle svých potřeb.

A zákon poslední - strukturalizace, strukturalizace a zase strukturalizace!!!

EXPERIMENT č.1

V něm si probereme funkci instrukce DJNZ.

```
0100  0609        LD B,09     ;V reg.B počet cyklů smyčky LOOP1
0102  0EFF LOOP1:LD C,FFH    ;V reg.C    "      "     " LOOP2
0104  16FF LOOP2:LD D,FFH    ;V reg.D    "      "     " LOOP3
0106  15   LOOP3:DEC D       ;Odečítání cyklů LOOP3
0107  20FD        JR NZ,LOOP3 ;Je-li Z=0, na LOOP3, jinak dál
0109  0D          DEC C       ;Odečítání cyklů LOOP2
010A  20F8        JR NZ,LOOP2 ;Je-li Z=0, na LOOP2, jinak dál
010C  10F4        DJNZ LOOP1  ;Odečítání cyklů LOOP1
010E  C9          RET         ;Návrat, když obsah reg.B=0
```

KROK 1

Zapište program do počítače a vyzkoušejte jeho funkce s různými hodnotami obsahu registru B. Jistě si vzpomenete na časovací smyčky z našich předchozích experimentů. Instrukce DJNZ je pro jejich tvorbu jako stvořená. Proto se velmi často používá v rutinách, kde je časování kritické - např. v částech operačních systémů, které vykonávají funkce SAVE a LOAD.

V našem experimentu dosáhneme změnou obsahu registru B těchto časových výsledků doby trvání provedení celé rutiny:

Registr B	Exekuční čas rutiny
01	0,4420844 sec
09	3,66123772 sec
FFH	105,47 sec

EXPERIMENT č.2

Ukážeme si první příklad programového použití části programu jako subrutiny, kterou bude rutina z předchozího experimentu.

```
011B  31FF01      LD SP,01FFH  ;Incializace reg.SP
011E  0603        LD B,03      ;Počet cyklů smyčky DELAY
0120  CD0201      CALL DELAY   ;Volání subrutiny DELAY
0123  C9          RET          ;Konečný návrat

0102  0EFF DELAY:LD C,FFH
0104  16FF LOOP2:LD D,FFH
0106  15   LOOP3:DEC D
0107  20FD        JR NZ,LOOP3
0109  0D          DEC C
010A  20F8        JR NZ,LOOP2
010C  10F4        DJNZ DELAY
010E  C9          RET          ;Návrat ze subrutiny
```

KROK 1

Poté, co program provede instrukci volání, přejde programové řízení na subrutinu DELAY (registr PC se naplní adresou 0102H). Zároveň je do zásobníku přenesena adresa, na níž leží první instrukce za instrukcí volání CALL DELAY. Subrutina se provede. Jakmile registr B dosáhne hodnoty nula, instrukce RET převede obsah spodních dvou adres zásobníku do registru PC a provede se zpětný skok na první instrukci za instrukcí CALL DELAY.

V subrutině jsme oproti jejímu zápisu v experimentu č.1 provedli malou změnu. Místo názvu návěští LOOP1 jsme použili název DELAY. Důvod pro tento krok je spíše estetický - zároveň

však zvyšuje srozumitelnost a přehlednost programu. Je lepší, když ve struktuře programu máme subrutinu s plně sdělným názvem DELAY (časová prodleva) než LOOP1 (smyčka 1).

Subrutiny se sestavují tak, aby jejich funkce podléhala vstupním parametrům nastaveným v předchozí rutině (jsou to tzv. přenášené parametry). Jinými slovy - přenášené parametry jsou zpracovávány subrutinou jako její vstupní parametry. Přitom nesmíme zapomenout uschovat na bezpečné místo obsahy všech registrů, které budeme po návratu ze subrutiny ještě potřebovat. Proto je u každé subrutiny důležitý údaj o tom, se kterými registry pracuje, abychom věděli, obsahy kterých jsou subrutinou "ohroženy" a (je-li třeba) učinili kroky pro jejich přechodné uložení jinam. To se provádí většinou příkazy PUSH (uložení bajtů do zásobníku); způsobů uložení je však celá řada. Po návratu ze subrutiny můžeme přenést uložené hodnoty bajtů (i výstupních parametrů) zpět do registrů instrukcí POP či způsobem jiným, jak si ukážeme v příštích experimentech přenosu parametrů do subrutin. Přenos uložených čísel provedeme ihned, jakmile s nimi chceme pracovat, ale i v případě, že by nám v zásobníku či jinde překážely, nebo by jim hrozilo přepsání, tedy zničení.

Stejně jako vstupní parametry, musíme zvážit i výstupní - tedy jaké, jak a proč je subrutina má zpracovat, a co s nimi dál.

Obecně lze říci, že subrutiny musejí splňovat dvě kritéria - funkční a ekonomické. Funkční jsme si probrali v předchozích dvou odstavcích. Ekonomickým se myslí např. to, zda zařazení nějaké subrutiny není zbytečným luxusem - tedy zda by jí zpracovaná funkce nešla zpracovat buď nějak jinak ("kratčeji"), nebo ji zahrnout do jiné části programu, resp.zda by nebylo výhodné uvažovat o jejím sloučením s podobnou funkcí jiné subrutiny. Ve výsledku jde tedy o šetření pamětí nebo zkrácení prováděcího času celého programu či jeho částí. Ekonomičnost musíme posuzovat i z hlediska toho, zda by některé funkce programu, které se v různých jeho částech opakují nebo jsou si nějak příbuzné, nebylo vhodnější převést do nějaké nové subrutiny, která by je provedla.

Obě tato kritéria se však někdy mohou dostat do výrazně nepřímé závislosti. V součtu patří prováděcí doba instrukcí CALL a RET v instrukčním souboru Z80 k jedněm z nejdelších. Je to pochopitelné - při provádění CALL XXXX se instrukce dekóduje, reg.PC je naplněn číslem absolutní adresy "skoku", je snížen obsah reg.SP o 2, a vypočtena adresa návratu, která se nakonec ještě přenese do zásobníku. Pak teprve je převedeno řízení programu na adresu XXXX. U RET se přenesou dva spodní bajty zásobníku do PC, zvýší se obsah reg.SP o 2 a provede se návrat těsně za instrukci volání. To vše samozřejmě zabere dost času. Proto je v některých časově kritických operacích lepší subrutiny nesestavovat, i když je to na úkor rozsahu volné paměti. Zde by se jevilo jako určité východisko použít krátkých instrukcí RST. Tyto instrukce jsou však takřka vždy využity výrobcem počítače k operacím prováděným systémem computeru.

A opět - pozor na řazení ukládaných a pořadí odebíraných dat jednotlivých bajtů a adres návratů v zásobníku!

KROK 2

Krokujte programem a zaměřte svou pozornost na registry PC, SP a zásobník před a po provedení instrukcí CALL a RET. Uvědomte si příčiny změn, které v nich probíhají. Pokud vám některá z nich bude ještě záhadou, vraťte se k učebnímu textu.

Nyní se pokuste analyzovat celý program z hlediska obou výše uvedených kritérií. Pokud jde o kritérium ekonomické, jak časově, tak i prostorově je na tom užití subrutiny v našem programu velmi bledě. Proč? Protože subrutina je užita jen jednou. Čas provedení celého programu se tak zbytečně prodlužuje o instrukce CALL a RET, které navíc zabírají 4 bajty paměti. Pro tento případ by bylo mnohem výhodnější funkci subrutiny zařadit do přímé linky programu:

```
        LD B,03
DELAY:  LD C,FFH
LOOP2   LD D,FFH
LOOP3   DEC D
        JR NZ,LOOP3
        DEC C
        JR NZ,LOOP2
        DJNZ DELAY
        LD A,00
        RET
```

Součet bajtů tohoto programu je 19, zatímco předchozí jich měl 28. Ušetřili jsme tedy skoro celou třetinu bajtů! Tím se nám potvrdilo, že zařazení jakékoli funkce části programu do přímé linky jeho vývoje je vždy rychlejší, než je tomu při užití subrutin. V případě, že bychom subrutinu programu však použili vícenásobně, je rozhodně lepší přimhouřit oko nad nějakým tím zlomkem vteřiny navíc a subrutinu do programu zařadit. Tímto strukturalizačním krokem dosáhneme mnohem větších výhod, které s sebou dobře sestavená programová struktura přináší. Z hlediska funkčnosti jsou obě varianty v pořádku.

EXPERIMENT č.3

Kromě jiného si ukážeme použití instrukce RST XX.

```
; Hlavní rutina:
;
0130  31FF01     LD SP,01FFH     ;Inicializace SP
0133  010001     LD BC,0100H     ;Čítač počtu bajtů pro vynulo-
                                 ;vání
0136  210004     LD HL,0400H     ;1.adr.nulovaného bloku bajtů
0139  D7         RST 16          ;Volání adresy 10H
013A  C9         RET             ;Návrat
;
; Subrutina:
;
0010  F5         PUSH AF         ;Uschova obsahů "ohrožených"
0011  C5         PUSH BC         ;registrů do zásobníku
0012  D5         PUSH DE
0013  E5         PUSH HL
0014  3600       LD (HL),00      ;Naplnění 1.adresy bajtem nula
0016  54         LD D,H          ;Přenos obsahu reg.HL do reg.DE
0017  5D         LD E,L
0018  13         INC DE          ;Přičtení 1 k DE
0019  EDB0       LDIR            ;Vynulování celého bloku bajtů
001B  0603       LD B,03         ;Časová konstanta prodlení
001D  CD0201     CALL DELAY      ;Volání subrutiny DELAY
0020  E1         POP HL          ;"Vyskladnění" uschovaných ob-
0021  C1         POP BC          ;sahů registrů ze zásobníku
0022  D1         POP DE          ;zpět do registrů
0023  F1         POP AF
0024  C9         RET             ;Návrat do hlavní rutiny (na
                                 ;adresu 013AH)
```

KROK 1

Proveďte zápis programu do počítače i s kontrolou jeho správnosti. Subrutinu DELAY použijte z minulého experimentu.

KROK 2

Uvedený program nuluje určený počet bajtů (zde 100H) v určeném bloku sousedících adres paměti (zde od adresy 0400H). Podobnou funkci obsahuje řada systémových i užitkových programů.

Hlavní program používá registry BC a HL pro přenos parametrů do subrutiny umístěné od adresy 0010H. BC obsahuje počet adres, jejichž obsah je určen k vynulování; v HL je první adresa tohoto bloku adres. V subrutině je uložena jedna nula na první adresu bloku instrukcí LD (HL),00. Funkce LDIR pak "okopíruje" nulový obsah této adresy do všech zbývajících adres bloku tak, že její obsah napřed přenese na adresu DE, která je o 1 vyšší. Obsahy HL a DE se pak BC+1 krát (na to pozor!) zvýší o 1, a BC-krát se provede přenos mezi adresou s nulovým bajtem (HL) a adresou o 1 vyšší (DE). Výsledkem bude, že všechny adresy bloku budou mít nulový obsah. Subrutina je volána jednobajtovou instrukcí RST 16 - stejně tak bychom ji mohli volat instrukcí CALL 0010H, ovšem ta je tříbajtová.

Na začátku subrutiny s blokovým přenosem dat LDIR jsou instrukce PUSH, které dočasně uschovávají obsahy všech párových registrů a reg.A a F do zásobníku. Tak jejich původní hodnoty zůstanou nedotčeny. Pokud by některá ze subrutin hlavního programu nepoužívala některý z párových nebo AF registrů, nemuseli bychom obsah těchto "neohrožených" registrů uschovávat.

Volání subrutiny DELAY je zde na ukázku toho, že i ze subrutin můžeme volat další subrutiny...a z nich třeba zase dalších "iks" subrutin. Toto řetězení subrutin je velmi náročné z hlediska uhlídání adekvátních adres návratů v zásobníku i orientace ve struktuře programu. Je však velmi silnou zbraní programátora. Subrutiny dokonce mohou volat samy sebe - tato technika jejich použití se jmenuje rekurzivní. Jde o užití podmíněné instrukce CALL na způsob instrukce DJNZ. Pokud použijeme nepodmíněnou instrukci CALL, musí být v subrutině umístěna nějaká jiná (splnitelná) podmínka, která nás z ní vyvede.

Vedle vývojového diagramu je velmi užitečnou orientační pomůckou paměťová mapa programu. Pro náš experiment vypadá takto (můžete si ji vždy doplnit řadou vlastních poznámek):

0000	
0010 0020 0025	Subrutina
0030 0100	
0102 010D	Subrutina DELAY
0110 0120	
0130 013D	Hlavní program
0140	
01E0 01E8 01F0 01F8 01FF 0200	Zásobník
0400 : 04FF	Data

EXPERIMENT č.4

Na čtyřech příkladech si ukážeme čtyři základní způsoby přenosu parametrů do subrutin. Všechny programy jsou funkčně shodné s programem předchozím (subrutina DELAY je vynechaná).

```
;==============================================================
;
; Technika 1. - přenos parametrů prostřednictvím registrů
;
0200  31FF08          LD SP,08FFH     ;Inicializace SP
0203  010001          LD BC,0100H     ;Počet adres k vynulování
0206  210004          LD HL,0400H     ;1.adresa bloku
0209  CD1002          CALL ZERO1      ;Volání subrutiny ZERO1
010C  C9              RET
;
; Subrutina ZERO1
;
0210  3600     ZERO1:LD (HL),00       ;Nula na 1.adresu bloku
0212  54              LD D,H          ;Přenos HL do DE
0213  5D              LD E,L
0214  13              INC DE
0215  ED80            LDIR            ;Přenos nul na adresy bloku
0217  C9              RET
;==============================================================
;
; Technika 2. - přenos parametrů prostřednictvím zásobníku
;
0220  31FF08          LD SP,08FFH     ;Inicializace SP
0223  010001          LD BC,0100H     ;Počet adres k vynulování
0226  210004          LD HL,0400H     ;1.adresa bloku
0229  C5              PUSH BC         ;Uložení parametrů (obsahu
022A  E5              PUSH HL         ;registrů) do zásobníku
022B  CD1002          CALL ZERO2      ;Volání subrutiny ZERO2
010C  C9              RET
;
; Subrutina ZERO2
;
0231  D1       ZERO2:POP DE           ;Adresa návratu do DE
0232  E1              POP HL          ;Přenos parametrů ze zásobníku
0233  C1              POP BC
0234  D5              PUSH DE         ;Adesa návratu zpět do zásobn.
0235  3600            LD (HL),00
0237  54              LD D,H
0238  5D              LD E,L
0239  13              INC DE
023A  EDB0            LDIR
023C  C9              RET
```

```
;========================================================================
;
; Technika 3. - přenos parametrů prostřednictvím bloku paměti
;
0240  31FF08          LD SP,08FFH
0243  010001          LD BC,0100H
0246  210004          LD HL,0400H
0249  ED430008        LD (0800H),BC  ;Parametry na adresy 0800H až
024D  220208          LD (0802H),HL  ;0803H
0250  CD5602          CALL ZERO3
0253  C9              RET
;
; Subrutina ZERO3
;
0256  ED4B0008 ZERO3:LD BC,(0800H);Parametry zpět do registrů
025A  2A0208          LD HL,(0802H)  ;z adres jejich uložení
025D  3600            LD (HL),00
025F  54              LD D,H
0260  5D              LD E,L
0261  13              INC DE
0262  EDB0            LDIR
0264  C9              RET
;
;========================================================================
;
; Technika 4. - přenos parametrů prostřednictvím adres následu-
;               jících těsně za instrukcí volání subrutiny
;
0265  31FF08          LD SP,08FFH
0268  CD7202          CALL ZERO4
026B  0001            DEFW 0100H      ;Uložení parametrů jako defino-
026D  0004            DEFW 0400H      ;vaných slov (DEFined Words)
026F  C9              RET
;
; Subrutina ZERO4
;
0272  DDE1     ZERO4:POP IX           ;Adresa "návratu" do IX
0274  DD4E00          LD C,(IX+00)    ;Přenos parametrů z adres
0277  DD4601          LD B,(IX+01)    ;026B až 026E do reg.BC a HL
027A  DD6E02          LD L,(IX+02)
027D  DD6603          LD H,(IX+03)
0280  110400          LD DE,0004      ;"Odchylka" adresy návratu
0283  DD19            ADD IX,DE       ;Nastavení skutečné adr.návratu
0285  DDE5            PUSH IX         ;Uschování adr.návratu do zás.
0287  3600            LD (HL),00
0289  54              LD D,H
028A  5D              LD E,L
028B  13              INC DE
028C  EDB0            LDIR
028E  C9              RET             ;Návrat na adresu 026FH
```

KROK 1

Zapište všechny programy do počítače a zkontrolujte zápis.

KROK 2

Přestože výsledkem činnosti všech čtyř programů je totéž, v lecčems se liší. Uvedené základní techniky přenosu parametrů do subrutin však neznamenají, že nelze vymyslet ještě nějaké další. Hned si uvedeme příklad:

```
;========================================================================
;
; Ukázka techniky speciální (netradiční, tvůrčí)
;
0100  310F01          LD SP,0110H     ;Inicializace SP na 1.DEFW
0103  1801            JR ZERO5        ;Relat.skok ("místo CALL")
0105  C9              RET             ;Konečný návrat
;
; Subrutina ZERO 5
;
0106  C1       ZERO5:POP BC           ;Přenos bajtů z oblasti DEFW
0107  E1              POP HL
0108  E5              PUSH HL
0109  D1              POP DE          ;Přenos HL do DE
010A  13              INC DE
010B  3600            LD (HL),00      ;Nula na 1.adresu bloku adres
010D  EDB0            LDIR
010F  C9              RET
;
0110  0001            DEFW 0100H      ;Počet nulovaných adres; reg.BC
0112  0004            DEFW 0400H      ;1.adresa bloku adres; reg.HL
0114  0501            DEFW 0105H      ;Adresa návratu ze subrutiny
```

KROK 3

Jak z ukázky speciální techniky vidíte, je programování - zvláště ve strojovém kódu - činností skutečně tvůrčí. Překvapivých a netradičních programových konstrukcí lze při fungující představivosti a zásobě znalostí vytvořit v každém programu celou řadu. A to jsme zatím jen u jedné subrutiny, která obsahuje "pár bajtů". Zkuste si představit, kolik možností řešení pak má takový program, který obsahuje 1000 nebo 5000, či dokonce nějakých 30000 bajtů! Množství možných programových kombinací řešení je přímo nepředstavitelné!!!

Nyní si porovnáme všech pět technik z hlediska ekonomického - kolik adres paměti zabírají jednotlivé programy:

Technika	Hlavní rutina	Subrutina	DEFW	Součet bajtů
1	15	8	0	23
2	17	12	0	29
3	22	15	0	37
4	9	29	4	42
speciální	6	10	6	22

Napřed porovnáme jen první 4 programy. I když technika 4 vypadá na první pohled nejhůř, je nutno si uvědomit, že pro volání a nastavení parametrů před jejich přenosem do subrutiny spotřebuje vždy jen 7 bajtů. Techniky 1., 2 i 3. spotřebují bajtů vždy 9. Z toho vyplývá jeden zajímavý poznatek - čím vícekrát budeme subrutinu sestavenou technikou 4. volat, tím více se bude snižovat rozdíl v časové ztrátě jejího provedení. A po překročení určité hranice počtu volání může být součet časů provedení nakonec nižší než u kratších subrutin vázaných na techniky, jež pro volání a nastavení parametrů vyžadují bajtů více. V našem případě je však subrutina se 7mibajtovým voláním a přenosem příliš dlouhá, proto nemůže porazit ani jednu z předchozích tří.

Absolutním vítězem této soutěže je program s užitím speciální techniky. Nejenže je ze všech nejkratší, ale je i podstatně rychlejší díky jedné inteligentní finese - místo instrukce CALL je v programu zařazena instrukce JR. Zatímco provedení CALL trvá 17 kmitů, u JR je to 12 kmitů. Pokud bychom chtěli být ještě rychlejší, můžeme použít místo JR instrukci JP, která se provádí jen 10 kmitů - jenže ta zabírá 3 bajty, zatímco JR jen 2 (jedná se tedy o obvyklý kompromis mezi časem a prostorem).

Instrukce JR samozřejmě neukládá žádnou adresu návratu do zásobníku. Proto vedle přenášených parametrů, které jsou součástí zásobníku, inicializujeme v něm jako další DEFW adresu návratu pro instrukci RET. Tato adresa musí být na konci zásobníku v momentu provádění instrukce RET!

V podstatě všechny uvedené programy můžeme použít i pro přenos mnohem většího počtu parametrů, než jen zde přenášených čtyř. Zcela nevhodná pro tento účel je technika 1. Ani 2. by nám moc nepomohla. O dost lépe už je na tom 3.technika. S malými úpravami subrutiny by byla nejvhodnější 4.technika, v níž jsou bajty uloženy přímo na adresy, z nichž si je subrutina odebírá. Z hlediska funkčnosti je speciální technika v podstatě shodná s technikou 4. Ovšem její subrutina by pro přenos většího počtu parametrů musela být rovněž trochu upravena. Nejlépe by však bylo pro tento účel sestavit zcela nový program. V učebnici se s takovými samozřejmě ještě setkáte. Uvedené programy se tedy vyznačují spíše jistým stupněm jednoúčelovosti.

Jeden důležitý dodatek ke všem programům v experimentech uváděným. Na jejich konci je vždy umístěna instrukce RET pro konečný návrat k instrukci volání těchto programů samotných. Protože je v programech často měněn obsah SP, bude nutno, aby našim programům předcházelo buď jen uložení původní adresy návratu pro její vyvolání konečnou instrukcí RET, nebo bude nutno před vstupem do programu uschovat původní hodnotu SP a před konečným návratem ji opět nastavit. U některých počítačů je možno využít i přímého skoku do jejich operačního systému (interpreteru Basicu), který převezme řízení programu, aniž by cokoli zkolabovalo. U většiny našich programů však nastavovat SP nemusíme, protože bývá automaticky nastavován systémem počítače. Proto můžete instrukci LD SP,XXXX všude tam, kde není přímo vázaná na nějaký přesně definovaný proces rutiny, vynechat. Nelze ji vynechat např. v naší poslední ukázce užití speciální techniky. Takže pozor na změny SP!!!

Pěti ukázkami provedení téhož různými způsoby jste se konečně nechali ovanout tím, co už lze nazvat opravdovým programováním ve strojovém kódu. Doporučujeme vám pohrát si s uvedenými

programy především pomocí krokování, abyste jejich chod plně pochopili.

EXPERIMENT č.5

Následujícím programem si budeme demonstrovat jednu z mnoha možností užití datové oblasti definovaných bajtů. V tomto případě se jedná o sestavení a využití dat jako tabulky adres skokových instrukcí.

```
0900  61                DEFB 61H      ;1.bajt porovnání pro případný
0901  4109              DEFW 0941H    ;skok na tuto 1.adresu - JP (HL)
0903  62                DEFB 62H      ;2.bajt...atd.
0904  4509              DEFW 0945H
0906  63                DEFB 63H
0907  5009              DEFW 0950H
0909  64                DEFB 64H
090A  5509              DEFW 0955H
090C  00                DEFB 00       ;Indikace konce tabulky
0915  21FD08 START:LD HL,08FDH ;Začátek programu
0918  23        NEXT:INC HL
0919  23             INC HL
091A  23             INC HL
091B  7E             LD A,(HL)    ;Obsah adr.(HL) do reg.A
091C  B7             OR A         ;Log.součet obsahu reg.A se sebou
091D  C8             RET Z        ;Je obsah A=0? ANO, pak návrat
0920  B8             CP B         ;NE, pak porovnání reg.B s reg.A
0921  20F5           JR NZ,NEXT   ;Není-li B=A, pro další DEFB
0923  23             INC HL       ;Je-li B=A, z další adr.(HL) pře-
0924  5E             LD E,(HL)    ;nes nižší bajt adresy skoku do E
0925  23             INC HL       ;a z další adr.(HL) přenes
0926  56             LD D,(HL)    ;vyšší bajt adr.skohu do reg.D
0927  62             LD H,D       ;Převeď obsah reg.D do H
0928  6B             LD L,E       ;a reg.E do L
0929  E9             JP (HL)      ;Proveď skok na nalezenou adresu
```

KROK 1

V oblasti dat se nám vedle terminu DEFW objevil nový - DEFB. V obou případech se nejedná o nic zvláštního, celá záležitost je spíše triviální. DEFW (DEFined Word) je definované slovo, které obsahuje dva bajty. DEFB (DEFined Byte) je definovaný bajt, a jak už sám název říká, obsahuje jeden bajt. Jedná se čistě o symboliku doplňkových funkcí generátoru strojového kódu a s vlastní mnemonikou Z80 to nemá nic společného. V assemblerových programech budeme tyto tzv. pseudoinstrukce (též zvané pseudooperační kódy) často používat, abychom se vyznali v tom, co jsou instrukce a co je oblast dat. Rozdíl mezi instrukcemi a definovanými bajty jsme si již vysvětlili. Při programování v assembleru nám pomocné generátory umožňují používat i jiné pseudoinstrukce. Pro definování řetězce definovaných bajtů (např. textu) bývají některé generátory vybaveny pseudoinstrukcí DEF$ a dalšími. V podstatě jde vždy jen o uložení bajtů určité hodnoty na místa paměti, která v souhrnu tvoří oblast dat. Ať už se označují jakkoli, neuděláme chybu, když jim všem budeme říkat prostě definované bajty.

Do programu zařazená instrukce typu CP je pro nás zatím ještě neznámá, řada na ni dojde později. Je to instrukce, která porovnává obsah registru (nebo adresy) jí určeného s obsahem akumulátoru, přičemž obsahy všech zúčastněných prvků zůstávají beze změny. V našem případě se jedná o odečet obsahu registru B od obsahu akumulátoru instrukcí CP B. Výsledek odečtu se nikam neukládá, pouze jsou jím ovlivňovány indikátory registru F. V programu za touto instrukcí testujeme stav indikátoru Z.

Zapište program do počítače a zkontrolujte správnost zápisu.

KROK 2

Datovou oblast uvedeného programu si nazveme skoková tabulka. V případě, kdy nějaký program provádí operaci, ze které existuje řada výstupů (resp. vstupů do řady různých subrutin) podle výsledku prováděné operace, je možné pro skoky do různých subrutin volit právě sestavení takové tabulky. V operaci ovšem musíme po provedení testu jejího výsledku přiřadit (zde registru B) DEFB, který pak určí, jaký skok bude proveden.

Pro náš program to znamená, že bude-li potřeba provést skok na adresu 0950H, bude nutné, aby se obsah registru B v předchozí části programu naplnil číslem 63H. Instrukcí CP B se zjistí, kdy bude obsah akumulátoru tentýž. V tom případě bude ignorována instrukce JR NZ,NEXT a registr HL bude v závěru obsahovat adresu skoku.

Pár instrukcí LD A,(HL) a OR A testuje, kdy je obsahem registru A nula. Nulu v uvedené tabulce používáme pro indikaci jejího konce. Pokud by některý z DEFB byl rovněž roven nule, museli bychom pro indikaci konce tabulky zvolit jiný bajt, tedy takový, který neobsahuje ani jeden z předchozích DEFB.

Pro demonstraci sestrojování a využití datových tabulek je uvedený případ dostatečně ilustrativní. Z hlediska funkčního by však program bylo možno sestavit o něco lépe, ostatně téměř jako vždy. V tomto případě by stálo za úvahu, zda by nebylo lepší zařadit za sebou přímo instrukce JP XXXX s definovanými absolutními adresami a pomocí DEFB a jejich přenosu do jedné hlavní skokové instrukce (ev.volání) provést skok na skok definovaný. Variant je samozřejmě celá řada. Nevýhodou prováděného testu (CP B) je nutnost prostupovat celou tabulku vždy od jejího začátku až do nálezu DEFB shodného s obsahem registru B. Tak je doba provedení závislá na tom, jak daleko od začátku tabulky hledaný DEFB je.

Podobnou tabulku lze využít např. pro přepočty různých hodnot, které vycházejí z jednoho základního tvaru modifikovaného právě rutinami, do nichž se provádí hledané skoky apod. Aplikace oblasti dat si ukážeme i v dalších experimentech.

****

KAPITOLA 6

Logické instrukce

****

Tyto instrukce mají svůj ohromný význam v mnoha operacích, které provádíme s bity bajtů. Jejich použitím můžeme leckdy podstatně zkrátit program a zefektivnit všechna programová kritéria. Proto je zvládnutí těchto instrukcí a hlavně možností jejich uplatnění v programu naprosto nutným předpokladem pro zvládnutí programování ve strojovém kódu. Bohužel je nutno konstatovat, že mnozí programátoři si při tvorbě programu neuvědomují možnosti zařazení logických instrukcí do jednotlivých programových operací, které pak provádějí zbytečným nánosem "substitučních" instrukcí. Čili podle známého hesla "proč se drbat takhle, když to jde takhle". Využití logických instrukcí závisí případ od případu. Jedná se tedy o proces tvůrčí, který je plně ve vašich rukou. Proto nemohou v závěru kapitoly uvedené experimenty ani zdaleka ukázat všechny možnosti jejich uplatnění.

Co je logická instrukce?

Tento typ instrukce se řídí zákony Booleovy algebry, jinak též Booleovy logiky. Všechny logické instrukce provádějí operace mezi akumulátorem a zvoleným registrem nebo obsahem místa paměti nebo určeným číslem. Úvod do této logiky je obsažen v kapitole 1, včetně pravdivostních tabulek logických funkcí AND, OR, NOT a XOR, uvedených v 1.kapitole. Tyto tabulky byste měli zvládnout nazpaměť - do té doby je mějte stále po ruce.

Booleova algebra

V instrukčním souboru Z80 jsou zařazeny tři logické instrukce: AND, OR a XOR.

AND - logický součin
OR - logický součet
XOR - exkluzívní součet

Pokud jste se ještě s touto logikou nesetkali, pak si pamatujte, že slova součin a součet mají v Booleově algebře poněkud jiný význam než v běžné školní aritmetice. Z hlediska hardwaru jsou v podstatě všechny operace, které mikroprocesor provádí, podřízeny této logice. Tzn., že na vstupech jednotlivých hradel, sestavených ze spínacích tranzistorů, integrovaných na polovodičový čip (integrovaný obvod), se provádějí logické operace mezi jednotlivými úrovněmi vstupních napětí. Podle funkce hradla je v přímé závislosti na vstupních napětích provedena logická operace, jejíž výsledek se objeví na jeho výstupu. Integrované obvody provádějí mnohem více logických operací, než kolik jich můžeme použít prostými instrukcemi Z80. V samotné obvodové technice log.1 znamená vyšší úroveň napětí (cca 5 V), zatímco log.0 je napětí nižší (kolem 2 V). Je-li tedy binárně-kódová reprezentace bajtu 00001111, pak je skutečně na jeho nižších bitech vyšší napětí než na zbývajících čtyřech vyšších. Z čehož vyplývá, že programování není nic jiného než dovedné kombinování dvou napěťových úrovní na vstupech (ale i výstupech) hradel v integrovaných obvodech na principu Booleovy algebry.

Multibitové operace

Veškeré logické operace mezi bajty je třeba chápat jako logické operace mezi jejich jednotlivými bity se stejným pořadovým číslem (tedy třeba bit 3 akumulátoru provádí logickou operaci vždy s bitem 3 druhého operandu). Logická operace AND B bude provádět logický součin mezi jednotlivými bity registrů A a B (s výsledky Q) takto:

A0.B0=Q0
A1.B1=Q1
A2.B2=Q2
A3.B3=Q3
A4.B4=Q4
A5.B5=Q5
A6.B6=Q6
A7.B7=Q7

Čili - multibitové logické operace jsou sérií logických operací mezi jednotlivými bity. Jednodušší, než rozepisovat operace výše uvedeným způsobem, je tento zápis:

```
11110000
00111100
--------
00110000
```

To je zápis logické operace AND B, kde obsah A=11110000 a obsah B=00111100. Výsledek operace O je uveden pod čarou. Obsah bajtů A a B ponecháme a provedeme zápis operace OR B:

```
11110000
00111100
--------
11111100
```

A s týmiž obsahy bajtů A a B zapíšeme operaci XOR B:

```
11110000
00111100
--------
11001100
```

I když operaci NOT v instrukčním souboru nemáme, provádíme ji několika jinými (komplementačními) instrukcemi, např. CCF je komplementace indikátoru přenosu CY, tedy v podstatě provedení operace NOT s tímto bitem registru F.

De Morganův teorém

Je důležitým teorémem Booleovy algebry. Může být zapsán těmito dvěma různými způsoby:

$$\overline{A.B}=\overline{A}+\overline{B} \qquad \overline{A+B}=\overline{A}.\overline{B}$$

Tento teorém je důležitý při sestavování obvodů vlastní elektroniky počítače a při jeho komunikaci s periferiemi. Znamená, že můžete vytvořit funkci NOR negací všech vstupů hradla AND. Alternativně můžete vytvořit funkci NAND negací všech vstupů hradla OR. Podobně lze sestavit funkci AND negací všech vstupů hradla NOR a funkci OR negací všech vstupů hradla NAND. V podstatě De Morganův teorém demonstruje, jak můžete jednoduše vytvořit OR a NOR funkce z hradel NAND.

Poznatky plynoucí z De Morganova teorému jsou uplatnitelné při sestavování a modifikaci programů pro styk počítače s periferiemi. Jedná se o oblast programování, která má svou výraznou specifiku. Anglicky se jmenuje INTERFACING. Tato oblast vyžaduje speciální, mnohem rozsáhlejší a zevrubnější studium než programování bez styku počítače s periferiemi, resp.bez řízeného přenosu dat mezi periferiemi a počítačem. Interfacing přesahuje rámec této učebnice, i když si v ní probereme i instrukce, které se interfacingu týkají.

Skupina logických instrukcí Z80

Při seznamování se s nimi věnujte zvýšenou pozornost způsobu, jakým ovlivňují jednotlivé indikátory registru F. Na tom se do značné míry zakládá i efektivita jejich využití.

KOMPLEMENTACE akumulátoru - CPL; NEG

Jak už jsme se zmínili, komplementace představuje provedení operace NOT. CPL ovlivňuje obsah bajtu v akumulátoru (bylo by tedy možno říci, že CPL je operací NOT A). Neovlivňuje ani jeden indikátor reg.F.

```
Reg.A před CPL     10111010
Reg.A po CPL       01000101
```

Užitečnou aplikací instrukce CPL je programové převedení obsahu akumulátoru na osmibitové DK číslo:

```
CPL
INC A
```

Pro uvedený postup dvojkové komplementace akumulátoru má Z80 instrukci NEG, která provede DK naráz. Rozdíl mezi oběma způsoby provedení DK obsahu reg.A je i v chování SI.

INSTRUKCE AND

Jako všechny logické instrukce, i tato operuje s akumulátorem. Instrukční soubor Z80 obsahuje 11 různých instrukcí typu AND. Po provedení instrukce AND se indikátor přenosu CY vždy nastaví na nulu. Indikátory Z a S jsou přímo ovlivněny výsledkem logické operace. Indikátor parity je log.1 při sudé paritě, log.0 při liché. Podstatu funkce AND si ukážeme na příkladu instrukce AND B (na stavu SI před operací nezáleží):

	Akumulátor	S	Z	P/V
Reg.A před AND B	11001100			
Reg.B	10001011			
Reg.A po AND B	10001000	1	0	1

Protože při funkci AND je logický součin nul roven nule a logický součin jedniček roven jedničce, mohlo by se zdát užití instrukce AND A naprosto beze smyslu - výsledkem této operace je původní obsah akumulátoru (čili se nic nestane). AND A má však velmi významnou funkci při zjištění, zda je obsah akumulátoru nulový (jen tehdy bude Z=1). S=1 jen tehdy, je-li obsah akumulátoru před provedením AND A záporné DK číslo (nejvyšší bit akumulátoru je ve stavu log.1). Pomocí AND A můžeme vynulovat indikátor CY, aniž bychom se dotkli obsahu reg.A.

Další užitečnou funkcí operace AND obecně je její využití při tzv. maskování - to je logická technika, při níž jsou určité bity multibitového slova vynulovány, nebo naopak nastaveny na log.1, přičemž ty důležité mohou zůstat beze změny. Maskováním např. vynulujeme ty bity bajtu, které nemají být v nějaké operaci aktivní nebo naopak - to záleží přímo na situaci v programovém místě maskování. Například chceme testovat, zda je nejnižší bit (tedy bit 0) nějakého bajtu (umístěného třeba na adrese ABCDH) ve stavu log.1 nebo 0:

```
LD A,(ABCDH)
AND 01
```

Ať je obsah adresy jakýkoli, maska 01 operace AND vynuluje všechny bity akumulátoru v rozsahu bit 1 až bit 7. V případě, že bit 0 testovaného obsahu adresy bude 1, bude obsah akumulátoru roven rovněž jedné. Když bude bit 0 nulový, i obsah akumulátoru bude roven nule (a indikátor Z bude 1). Takovouto maskou a zjištěním stavu Z můžeme zjistit stav jakéhokoli bitu jakéhokoli bajtu.

Velmi často se maskování používá při interfacingu, resp. při programovém ovládání jakéhokoli hardwaru. Často se při něm setkáme s tím, že na paralelní datové sběrnici ovlivňuje každý bit (nebo jejich skupiny) některou z více funkcí hardwaru. Pokud potřebujeme vymezit jen jednu z jejich počtu, použijeme masku. Tak např. potřebujeme, aby se vykonala jen určitá funkce, ovlivňovaná čtveřicí nižších bitů (bity 0-3) a ostatní funkce ovlivňované zbylými čtyřmi bity zůstaly dezaktivované. Pak musíme zajistit, aby bity 4-7 námi konstruovaného výstupního bajtu byly nulové. Dosáhneme toho maskou:

```
AND 0FH
```

Binárně kódová reprezentace bajtu 0FH je 00001111. Všechny 4 nižší výstupní linky pak zůstanou v předem definovaném stavu, ale vyšší 4 linky budou ve stavu log.0. Zde je třeba doplnit, že někdy může být některá z výstupních linek aktivována naopak tehdy, je-li na ní log.0. Pak pro její dezaktivaci (nastavení na úroveň log.1) musíme použít jinou logickou operaci.

Během výkladu funkcí logických operací se pomocí pravdivostních tabulek snažte ve věci orientovat, abyste jim plně porozuměli.

INSTRUKCE XOR

Ovlivňování SI reg.F je shodné s operací AND. I XOR má v instrukčním souboru 11 variant. Funkci XOR si ukážeme na instrukci XOR (HL), která provádí operaci XOR obsahu adresy (HL) s obsahem akumulátoru:

	Akumulátor	S	Z	P/V
Reg.A před XOR (HL)	10010001			
Obsah adresy (HL)	00110000			
Reg.A po XOR (HL)	10100001	1	0	0

Podobně jako operace AND, i XOR má své "zvláštní stavy". Např. provedením operace XOR 00 zůstane stav reg.A nezměněn, ale indikátory CY, Z, S i P/V budou patřičně ovlivněny. Takže pokud chceme zjistit, zda je obsah jakéhokoli bajtu nulový či nenulový, zda je jeho DK reprezentace negativní či pozitivní, ale i zda je jeho parita sudá či lichá, použijeme XOR 00.

Instrukcí XOR A vynulujeme akumulátor a tedy i nastavíme indikátor Z na log.1. Tato jednobajtová instrukce se v programech používá velmi často, protože pro účel často požadovaného vynulování reg.A je kratší i rychlejší než dvoubajtová LD A,00.

XOR FFH má tentýž efekt jako CPL, ovšem s tou výjimkou, že ovlivňuje všechny čtyři testovatelné SI, zatímco CPL ani jeden.

INSTRUKCE OR

11 instrukcí OR ovlivňuje SI stejně jako AND a XOR. Funkci OR si ozřejmíme na příkladu OR (IX+20H). Provedeme tak operaci OR mezi obsahem adresy o 32 vyšší než je obsah IX a akumulátorem.

	Akumulátor	S	Z	P:V
Reg.A před OR (IX+20H)	00000011			
Obsah adresy (IX+20H)	00110010			
Reg.A po OR (IX+20H)	00110011	0	0	1

Funkci OR A (resp.OR 00) lze použít pro indikaci toho, zda je obsah akumulátoru nulový. S tímto použitím instrukce OR jsme se již setkali - při testu, zda je obsah párového registru roven nule. Jedná se opět o velmi časté použití funkce OR, proto si je zapamatujte:

```
LD A,B
OR C
```

Tuto testovací kombinaci nejen že můžeme, ale přímo musíme použít, kdykoli potřebujeme zjistit, zda je obsah kteréhokoli párového registru nulový po jeho dekrementaci (resp. inkrementaci) instrukcemi DEC BC (resp.INC BC). Je to proto, že 16tibitové instrukce DEC rr a INC rr neovlivňují žádný SI - na rozdíl od snižování a zvyšování 8mibitového, tedy DEC r a INC r. Pokud tedy oba párové registry budou nulové, bude i výsledek uvedené operace nulový a indikátor Z=1 nám tento výsledek jasně oznámí. Samozřejmě, že takto testovat můžeme obsah jakýchkoli dvou 8mibitových registrů kdykoli, tedy nejen pro uvedený případ.

Protože instrukce AND, XOR i OR automaticky nulují indikátor CY, je možno kteroukoli z nich použít pro tento účel. Použijeme samozřejmě takovou, která nebude mít žádný negativní vliv na průběh našeho programu.

Užití logických instrukcí v operacích s perifériemi

Jak už jsme se zmínili, interfacing je sám o sobě značnou vědou, která zahrnuje nejen znalost programování, ale i "hromady" hardwaru. Při paralelní komunikaci meži počítačem a periferním zařízením (jímž může být cokoli - od žárovky přes tiskárnu až po motor vozu formule 1) počítač provádí obousměrnou komunikaci prostřednictvím svých výstupních a vstupních portů. Z nich se data posílají ven (OUT) z počítače nebo jimi do (IN) počítače přicházejí. Řídící funkce počítače spočívá v tom, že po zjištění, v jakém stavu se periférie nachází, na základě vloženého programu rozhodne, zda její stav nějak změní, resp. na ni pošle nějaká data. Tento cyklus se s různými obměnami stále opakuje.

Při práci s perifériemi je význam logických funkcí značný. Díky nim můžeme poměrně jednoduše zjišťovat stavy na vstupních portech i konstruovat bajty před jejich odesláním na výstup.

Typický osmibitový datový výstup umožňuje řídit až 8 externích zařízení - ovšem jen ve smyslu vypnout-zapnout. Pro komplexnější a efektivnější řízení jednoho zařízení je nutno využít více linek pro vstup i výstup. Pro náš jednoduchý případ určeme, že každá z osmi paralelních linek bude rozsvěcet a zhasínat jednu diodu LED. Rozsvícení nastane tehdy, bude-li adekvátní bit ve stavu log.1, při log.0 dioda zhasne. Výstup bude rozdělen takto:

```
Bit 0 - LED 0
Bit 1 - LED 1
Bit 2 - LED 2
Bit 3 - LED 3
Bit 4 - LED 4
Bit 5 - LED 5
Bit 6 - LED 6
Bit 7 - LED 7
```

Nyní kontrolní otázky pro vás. U všech máme na mysli jen samotný princip maskování logickými instrukcemi (bez realizace transportu bajtu z reg.A na výstup).

1 - Reg.A má obsah nula. Jak zpracujete jeho obsah, aby bajt z registru A odeslaný na výstup rozsvítil LED 0 a LED 7?

2 - Reg.A má obsah 01010101. Jak dosáhnete rozsvícení čtyř diod - LED 1 až LED 4?

3 – Reg.A má obsah FFH. Jak provedete cyklus, který bude střídavě rozsvěcet a zhasínat dolní a horní polovinu diod? Sestavte rutinu s logickými funkcemi. Moment odeslání bajtu na výstup v ní označte jen slovem Výstup.

Odpovědi:

```
1 - OR 10000001

2 - XOR A                  nebo:   2 - XOR 01001011
    OR 00011110

3 - BLIK:XOR 11110000
         Výstup dolní
         XOR 11111111
         Výstup horní
         OR 00001111
         JR BLIK
```

Jak ze schématu rutiny BLIK vysvítá, instrukce OR 00001111 a JR BLIK prodlužují dobu svícení horní poloviny diod o dobu jejich trvání. Pokud bychom chtěli, aby doby svícení (i zhasnutí) obou polovin bylo shodné, museli bychom mezi Výstup dolní a XOR 11111111 zařadit časovací rutinu, která by zajistila prodlevu doby svitu dolní poloviny diod. Uvedený příklad je poměrně jednoduchý. O něco komplikovanější program si probereme v Experimentu 2.

EXPERIMENT č.1

Demonstrace užití instrukce typu AND pro operaci AND mezi obsahy registrů BC a DE. Výsledek se ukládá do reg.HL.

```
01FA  01MMMM        LD BC,MMMM ;Inicializace obsahu BC číslem MMMM
01FD  11NNNN        LD DE,NNNN ;     "          "    DE    "   NNNN
0200  78     AND16:LD A,B      ;Přenos B do A
0201  A2            AND D      ;Operace B AND D
0202  67            LD H,A     ;Výsledek do H
0203  79            LD A,C     ;Přenos C do A
0204  A3            AND E      ;Operace C AND E
0205  6F            LD L,A     ;Výsledek do L
0206  B4            OR H       ;Je-li výsledek=0, pak ať Z=0
0207  C9            RET
```

KROK 1

Zkontrolujte správnost svého zápis programu do počítače. Proveďte buď přímou inicializaci reg.BC a DE (pomocí monitoru strojového kódu) – v tom případě vynechte instrukce na adresách 01FAH a 01FDH; nebo obsahy registů zapište přímo na adresy čísel MMMM a NNNN. Nyní již umíte přenášet data i mezi paměťovými adresami – co kdybyste si zkusili inicializovat reg.BC a DE jinak? Zkoušejte, tvořte, uvažujte nad možností řešit programy nebo jejich části i jinak, než je přímo uvedeno.

Krokujte programem s různými vstupními hodnotami reg.BC a DE:

```
1.        BC  00110011 11001001 (=33C9H)
          DE  11110000 01110011 (=F073H)
              -----------------
          HL                        (zkuste vždy určit předem)

2.        BC  10100101 10100101 (=A5A5H)
          DE  11001100 01011111 (=CC5FH)
              -----------------
          HL

3.        BC  11111111 00000000 (=FF00H)
          DE  00000000 11111111 (=00FFH)
              -----------------
          HL
```

Rovněž předem určete závěrečné stavy všech čtyř SI registru F. Spuštěním programu se přesvědčete o správnosti svých výpočtů. Výsledky operací AND v reg.HL a konečné stavy SI v pořadí Z,S,P/V,CY by měly být tyto:

1.3041H 0010 2.8405H 0100 3.0000H 1010

EXPERIMENT č.2

Následující program stanoví, která z připojených periferních zařízení změnila svůj předchozí stav. Jinými slovy – která z nich přešla ze stavu zapnuto do vypnuto (on-off) ale i naopak – ze stavu vypnuto do stavu zapnuto (off-on). Jedná se tedy o příjem dat z periférie (z jejího hlediska výstupních) do počítače (z jeho hlediska vstupních) – čili opak výše uvedeného

příkladu s diodami LED, kde komunikace šla opačným směrem - z počítače na periférii. Vstup je paralelní (8mibitový). Jako jednotlivá zařízení budeme opět zvažovat diody LED, které budou připojeny k počítači stejně jako v závěru učebního textu této kapitoly. Je-li úroveň bitu vstupního bajtu log.1, zařízení je zapnuto, a naopak. K počítači však nebudeme a ani nemusíme nic připojovat. Simulovat připojení diod (resp.hodnotu vstupního bajtu) budeme pomocí inicializace reg.B a reg.A.

```
0100  0688    LD B,88H   ;Simulační bajt předchozího stavu
0102  3E09    LD A,09    ;     "       "  momentálního stavu
0104  4F      LD C,A     ;Momentální stav zařízení do reg.C
0105  A8      XOR B      ;Bit log.1 po XOR znamená změnu
0106  57      LD D,A     ;Výsledek operace XOR B do D
0107  A0      AND B      ;Bit log.1 znamená změnu on-off
0108  67      LD H,A     ;Výsledek operace AND B do H
0109  2F      CPL        ;"NOT A" - převrácení stavů bitů
010A  A2      AND D      ;Bit log.1 znamená změnu off-on
010B  6F      LD L,A     ;Výsledek operace AND D do L
001C  C9      RET
```

KROK 1

Proveďte kontrolu správnosti zápisu programu do počítače.

KROK 2

Zda jednotlivé diody prošly změnou on-off nebo off-on (či zůstaly beze změny), zjišťujeme postupně:

1. Když bit N (N je v intervalu 0-7) registru D je ve stavu log.1, pak v diodě N došlo ke změně stavu.
2. Když bit N registru L je ve stavu log.1, pak dioda N prošla změnou on-off (ze zapnuto na vypnuto, tedy zhasla)
3. Když bit N registru H je ve stavu log.1, pak dioda N prošla změnou off-on (rozsvítila se)
4. Když bit N registru L je ve stavu log.0, pak stav diody N zůstal nezměněn (svítí i nadále)
5. Když bit N registru H je ve stavu log.0, pak stav diody N zůstal nezměněn (nesvítí i nadále)

KROK 3

Pro absolutní porozumění bitových změn si projděte následující znázornění všech čtyř logických operací programu:

Předchozí stav vstupního bajtu, inicializovaný v reg.B, je 10001000. Jeho momentální ("nový") stav, inicializovaný v reg.A, je 00001001.

```
Instrukce XOR B:     10001000   předchozí stav
                     00001001   momentální stav
                     --------
                     10000001   log.1-změněné stavy

Instrukce AND B:     10001000   předchozí stav
                     10000001   log.1-změněné stavy
                     --------
                     10000000   log.1 on-off

Instrukce CPL:       10000000   výsledek AND B
                     --------
                     01111111   log.1-bez změny off-on

Instrukce AND D      01111111   výsledek minulé CPL
                     10000001   výsledek XOR B (viz výše)
                     --------
                     00000001   log.1 off-on, 0-bez změny
```

Z toho vyplývá, že zhasla dioda LED 7, rozsvítila se LED 0, zatímco ostatní zůstaly v předchozím stavu (tzn., že LED 3 svítí nadále a LED 1, 2, 4, 5, 6, zůstaly zhasnuté). Nyní tedy svítí diody LED 0 a LED 7, ostatní ne - což je potvrzeno bajtem momentálního stavu na vstupu (inicializovaný reg.A).

KROK 4

Proč je na adrese 0104H přenesen obsah reg.B do reg.C? Pokud bychom chtěli aktivovat tento program jen jednou, byla by instrukce LD C,A zbytečná. Pokud však budeme chtít pro indikaci stavu připojené periférie program využít vícekrát, musíme uschovat bajt momentálního stavu, protože propříště bude bajtem stavu předchozího. Tzn., že před dalším vstupem do programu musíme bajt z reg.C přenést do reg.B, čímž jej inicializujeme hodnotou předchozího stavu (a do reg.A opět přijde vstupní bajt stavu momentálního). Zamyslete se nad tím, jak byste z uvedeného programu udělali subrutinu testu změn periférie.

Cvičení

Operace Booleovy algebry si přímo vyžadují jejich procvičení. Proto je v žádném případě nevynechte! Správné odpovědi jsou uvedeny opět za poslední úlohou.

1. Určete výsledky uvedených operací (u HD čísel si proveďte jejich převod na binární):

   a. 11001011 AND 01011010
   b. 00100000 OR 11011111
   c. 00100000 AND 11011111
   d. 10101010 XOR 10100100
   e. CCH AND 0BH
   f. A6H XOR 80H
   g. 37H XOR 04H
   h. 49H AND 1BH

2. Opět si představte zapojení osmi diod LED 0 až LED 7 (jakožto periférie počítače) připojených na paralelní osmibitový vstup – budeme tedy zjišťovat, které diody svítí (adekvátní bit bude ve stavu log.1) pro různé hodnoty vstupních bajtů:

   a.53H b.40H c.64H d.20H e.02H f.30H g.06 h.C0H i.01H j.28H

Převody HD čísel na binární si proveďte, i když je to činnost poněkud zdlouhavá. Je však nutné, abyste se s ní dobře obeznámili, protože v případě, že se s nutností provést převod setkáte při programování, nebudete si pak vědět rady. A buďte si jisti, že to nebude zdaleka jen jednou, ani jen padesátkrát!

3. Podle Experimentu č.2 jako vzoru zjistěte, k jakým změnám došlo v periferně připojených diodách při těchto údajích:

	Předchozí stav	Momentální stav
a.	84H	46H
b.	27H	63H
c.	02H	07H
d.	A7H	D8H

Odpovědi

1. a. 01001010
   b. 11111111
   c. 00000000
   d. 00001110
   e. 00001000 (=08H)
   f. 00100110 (=26H)
   g. 00110011 (=33H)
   h. 00001001 (=09H)

2. Svítit budou tyto diody LED:

   a. 6,4,2,0
   b. 6
   c. 6,5,2
   d. 5
   e. 1
   f. 5,4
   g. 2,1
   h. 7,6
   i. 0
   j. 5,3

3.a. Napřed převedeme HD čísla na binární:
   84H = 10000100 předchozí stav
   46H = 01000110 momentální stav
   Pak provedeme následující operace:
   10000100 XOR 01000110 = 11000010
   10000100 AND 11000010 = 10000000
   NOT 10000000 = 01111111
   01111111 AND 11000010 = 01000010
   Svítily diody 7 a 3, teď svítí diody 6, 3 a 2. Zhasla dioda 7, rozsvítila se dioda 6.

Podobně postupujte ve zbylých třech částech 3.bodu s těmito výsledky (čísla zhasnuvších a rozsvítivších se diod):
   b. on-off 3,2; off-on 6
   c. on-off žádné; off-on 2,0
   d. on-off 5,2; off-on 6,4,3

****

KAPITOLA 7

Bitové manipulace, rotace a posuvy

****

Instrukce BIT, SET a RES

Tyto instrukce zabírají značný díl instrukčního souboru Z80 - 80 pro každou z nich, tedy celkem 240. Pomocí nich je dostupný každý bit registrů A,B,C,D,E,H,L a jakéhokoli místa v paměti. Funkce instrukcí jsou tyto:

BIT - test stavu bitu (je ve stavu log.1 nebo log.0?)
SET - uvedení bitu do stavu log.1 (resp.ponechání jej v log.1)
RES - vynulování bitu (resp.ponechání jej ve stavu log.0)

Např. po provedení instrukce SET 1,B bude bit 1 reg.B ve stavu log.1. Po instrukci RES (z angl.RESet) ve tvaru RES 3,(HL) bude bit 3 obsahu adresy (HL) ve stavu log.0. Stav ostatních bitů na operaci zúčastněného registru nebo místa paměti zůstává nedotčen.

Instrukcí BIT 0,A testujeme log.stav bitu 0 akumulátoru. Podle toho, v jakém log. stavu testovaný bit je, bude ovlivněn indikátor Z reg.F. Je-li bit nulový, bude Z=1, je-li ve stavu log.1, bude Z=0. Tato instrukce nemění obsah testovaného bitu, tedy ani bajtu.

Instrukce BIT je užitečná tehdy, chceme-li testovat stav jednoho bitu bajtu. Můžeme se tak "vyhnout" binárnímu výpočtu bajtu maskování logickou operací, která navíc mění obsah maskovaného bajtu. Např.:

```
LD A,(1234H)
AND 10H
```

je totéž, co:

```
LD A,(1234H)
BIT 4,A
```

V prvním případě je pro test bitu 4 obsahu adresy 1234H použit maskovací bajt 10H. Jak víme, touto operací se všechny ostatní bity reg.A vynulují. Instrukce BIT 4,A ponechává akumulátor v původním stavu. Oba způsoby testování zabírají 5 bajtů paměti. Který z obou postupů použijeme, závisí na konkrétní programové situaci.

Skupina instrukcí rotace a posuvů

Těchto instrukcí je 74. Kromě čtyř z nich (RLCA, RRCA, RLA a RRA), které mají vliv jen na indikátor CY, všech ostatních 70 ovlivňuje všechny čtyři SI. Zvláštním typem této skupiny instrukcí jsou dvě, které mohou pracovat s decimálními čísly (RLD a RRD). Pro výuku (a později i programování) těchto instrukcí si vemte k ruce přílohu, v níž jsou jejich funkce znázorněny graficky.

Rotační instrukce

Rotace vlevo cirkulační - RLC r

Cirkulační se některé rotace jmenují proto, že v bajtu dochází k rotaci samotných osmi bitů jeho obsahu, bez toho, aby do něj vstupoval indikátor CY (jak je tomu u jiných rotací - viz dále).

Funkce RLC je patrná z obrázku přílohy. V dalším budeme značit bity mezinárodní zkratkou D (Digit - česky číslo) s připojeným pořadovým číslem bitu. D3 je tedy bit 3. U každé instrukce si uvedeme tabulkový příklad jejího provedení.

Instrukce RLC A provádí cirkulační rotaci vlevo s bity reg.A:

	Bit CY	Akumulátor							
Před RLC A	CY	D7	D6	D5	D4	D3	D2	D1	D0
Po RLC A	D7	D6	D5	D4	D3	D2	D1	D0	D7

	Bit CY	Akumulátor							
Před RLC A	-	1	1	0	1	1	1	0	0
Po RLC A	1	1	0	1	1	1	0	0	1

Vidíme, že po každém provedení instrukce RLC A se všechny bity reg.A posunou o jeden bit doleva, přičemž CY bude nastaven na počáteční log. stav bitu 7. Pomlčka "-" na místě CY před provedením instrukce znamená, že pro instrukci samu nemá jeho předchozí stav žádný význam.

Rotace vpravo cirkulační - RRC r

Jako příkladu použijeme instrukci RRC C, která provede uvedenou bitovou rotaci reg.C:

```
              Bit CY              Registr C
Před RRC C    CY         D7 D6 D5 D4 D3 D2 D1 D0
Po RRC C      D0         D0 D7 D6 D5 D4 D3 D2 D1

              Bit CY              Registr C
Před RRC C    -          1  1  0  1  1  1  0  0
Po RRC C      0          0  1  1  0  1  1  1  0
```

Instrukce RRC r je analogická instrukci RLC r. Jen je třeba pamatovat, že vlivem posunu bitů doprava se v CY objeví log.stav bitu 0.

Rotace vlevo - RL r

Provedení instrukce RL A:

```
              Bit CY              Akumulátor
Před RL A     CY         D7 D6 D5 D4 D3 D2 D1 D0
Po RL A       D7         D6 D5 D4 D3 D2 D1 D0 CY

              Bit CY              Akumulátor
Před RL A     0          1  1  0  1  1  1  0  0
Po RL A       1          1  0  1  1  1  0  0  0
```

Zde můžeme indikátor CY zvažovat jako 9.bit (bit 8) akumulátoru. Lidověji řečeno - bitová rotace probíhá "skrze" CY. To nám umožňuje, abychom bit CY začlenili do bajtu. Význam a užití této možnosti si ukážeme později.

Rotace vpravo - RR r

Provedení instrukce RR D:

```
              Bit CY              Registr D
Před RR D     CY         D7 D6 D5 D4 D3 D2 D1 D0
Po RR D       D0         CY D7 D6 D5 D4 D3 D2 D1

              Bit CY              Registr D
Před RR D     -          1  1  0  1  1  1  0  0
Po RR D       0          0  1  1  0  1  1  1  0
```

Opět jde o analogii instrukce RL r. Do CY bude přenesen bit 0.

Tyto čtyři rotační instrukce mají široké využití. Obsah rotovaných registrů může plnit funkci čítače osmi průběhů, které se tak často v programech objevují. Při rotacích se využívá testu stavy indikátoru CY. Podle sestavy log.1 a log.0 rotovaného bajtu pak lze řídit chod nějakých operací v daném sledu sestavy. Naplňováním bajtu log. stavem CY, který indikuje nějaký dvoustavový průběh, jej pak můžeme zpětně zjistit, atd.

Uvedeme si příklady použití rotace. Rutina zjišťuje, zda a kolik nul je na vyšších bitových pozicích bajtu:

```
        LD C,N      ;Do reg.C testovaný bajt N
        LD A,00     ;Čítač nulových bitů testovan.bajtu
KOLIK:  RL C        ;Rotace (nejvyšší bit do CY)
        JR C,KONEC  ;Když CY=1, skok na adr.KONEC
        INC A       ;Zvýšení čítače o 1
        CP 08       ;Už rotovalo 8 bitů?
        JR NZ,KOLIK;Když NE, pak na adr.KOLIK
KONEC:  RET         ;Návrat
```

V případě, že na nejvyšších bitových pozicích reg.C jsou tři nuly, provedou se tři rotace se třemi zvýšeními obsahu reg.A, kde tak bude číslo 3. Při čtvrtém vstupu do smyčky KOLIK se do CY přesune první bit ve stavu log.1 a podmíněný skok JR C,KONEC se provede, program se ukončí. Hledaný počet nul bude v reg.A.

V následujícím příkladu si ukážeme, jak můžeme inicializací bajtu stanovit, kolik (a které) z osmi subrutin se má provést. Počet kombinací provedení osmi subrutin je 256:

```
RRA
CALL C,SBR1
RRA
CALL C,SBR2
RRA
CALL C,SBR3
RRA
CALL C,SBR4
RRA
CALL C,SBR5
RRA
CALL C,SBR6
RRA
CALL C,SBR7
RRA
CALL C,SBR8
```

Kombinací jedniček a nul v akumulátoru dosáhneme požadované kombinace provedení subrutin volaných instrukcemi CALL vždy, když po rotaci bude CY=1. Rutiny budou samozřejmě provedeny tak, že ať už bude počet jedniček v akumulátoru jakýkoli, nikdy nebude subrutina s vyšším pořadovým číslem provedena před subrutinou s číslem nižším (což by nám někdy mohlo vadit, ale to bychom museli celou věc řešit jinak).

Jistě jste si všimli, že v programu je užita instrukce RRA, zatímco vy jste se zatím seznámili s instrukcí RR A. To je určitá zvláštnost Z80. Několik instrukcí jeho souboru je funkčně takřka ekvivalentních. Liší se jen počtem bajtů, časem provedení a počtem ovlivňovaných SI (a samozřejmě svým zápisem). Tak např. zatímco provedení dvoubajtové RR A (ta má vliv na všechny SI) trvá 8 kmitů, stejnou operaci provede jednobajtová RRA za pouhé 4 kmity. A to už je nějaký rozdíl! RRA však ovlivňuje jen CY. Ale o ten nám v našem programu jde, takže pochopitelně dáme instrukci RRA přednost. Podobně je tomu u instrukcí RL A a RLA.

V této souvislosti vás ještě upozorníme na dvě naprosto ekvivalentní instrukce, z nichž obě nemají vliv ani na jeden SI. Liší se jen časem a prostorem. Jedná se o instrukci LD HL,(ADDR) ve dvou HD provedeních - ED6BXXX a 2AXXXX. Samozřejmě budeme volit její kratší tvar, který má i o pětinu rychlejší prováděcí dobu. Pokud budete programovat s pomocí generátoru strojového kódu, ani vám nic jiného nezbyde, protože výrobci generátorů do nich delší tvar instrukce ani nezařazují. Bohužel do nich nezařazují i některé nestandardní instrukce, které se tak musejí zapisovat přímo do paměti (ale i bez této "extrakávy" lze programovat, co hrdlo ráčí).

Instrukce posuvu

Rozdíl mezi rotací a posuvem je ten, že rotace probíhá v pomyslném kruhu, kdežto posuv po lince (byť někdy "zakroucené"). Pokud se vám z rotací nezatočila hlava, pojďme posouvat:

Posuv vlevo aritmetický - SLA r

Funkce SLA je patrna z obrázku přílohy. Doplníme si ji opět tabulkou:

	Bit CY	Akumulátor
Před SLA r	-	D7 D6 D5 D4 D3 D2 D1 D0
Po SLA r	D7	D6 D5 D4 D3 D2 D1 D0 0

Příklady průběhů instrukce SLA A s různými obsahy reg.A:

	Bit CY	Akumulátor	
Před SLA A	-	0 0 1 1 0 0 1 1	= 51 decim.
Po SLA A	0	0 1 1 0 0 1 1 0	=102 "

	Bit CY	Akumulátor	
Před SLA A	-	0 1 1 0 0 1 1 0	=102 decim.
Po SLA A	0	1 1 0 0 1 1 0 0	=204 "

	Bit CY	Akumulátor	
Před SLA A	-	1 0 0 1 1 0 0 0	=152 decim.
Po SLA A	1	0 0 1 1 0 0 0 0	= 48 "

Podívejte se pozorně, co je vlastně výsledkem funkce SLA. Je to přece násobení dvěma! Pochopitelně jen tehdy, pokud nedojde k přeplnění registru - to se stalo v posledním případě, proto i výsledek není očekávaných 304, protože to je číslo větší než 256 (převyšuje možnou reprezentaci bajtu). Výsledek je tedy 2*152-256, což je 48. Posouvat můžeme i obsahy více bajtů, pomyslně umístěných "v řadě vedle sebe":

```
SLA E
RL D
```

což je posuv obsahu párového registru DE. Podobné operace můžeme provádět i s obsahy adres:

```
LD IX,1234H
SLA (IX)
RL (IX+01)
```

Multibajtové posuvy bitů jsou velmi užitečné při násobení a dělení větších čísel. Velký význam mají i při operacích s grafikou obrazové paměti počítačů.

Posuv vpravo aritmetický - SRA r

```
              Bit CY            Registr
Před SRA r      -       D7 D6 D5 D4 D3 D2 D1 D0
Po SRA r        D0      D7 D7 D6 D5 D4 D3 D2 D1

              Bit CY           Akumulátor
Před SRA A      -       0  0  0  0  1  1  1  1  = 15 decim.
Po SRA A        1       0  0  0  0  0  1  1  1  =  7   "

              Bit CY           Akumulátor
Před SRA A      -       1  0  0  0  1  1  1  0 =-114 decim.
Po SRA A        0       1  1  0  0  0  1  1  1 = -57   "
```

Je zde určitá analogie s funkcí SLA - až na to, že místo předpokládaného naplnění místa bitu 7 nulou zůstává na jeho místě původní log.stav. Na první pohled vidíme, že pomocí SRA můžeme provádět dělení dvěma. Při opakování funce pak samozřejmě čtyřmi, osmi, šestnácti, atd. Zůstatek dělení (je-li nějaký), se objeví v indikátoru CY. Ovšem pozor - jak je patrno z tabulky, jedná se o dělení DK čísel (tedy vlastně 7mibitových, kde bit 7 hraje roli znaménka + nebo -)!!! Instrukci typu SRA můžeme použít ve spojení s RR pro vícebajtový posuv vpravo; např.:

```
SRA H
RR L
```

Posuv vpravo logický - SRL r

```
              Bit CY            Registr
Před SRL r      -       D7 D6 D5 D4 D3 D2 D1 D0
Po SRL r        D0      0  D7 D6 D5 D4 D3 D2 D1

              Bit CY           Akumulátor
Před SRL A      -       1  0  0  0  0  0  1  1  =131 decim.
Po SRL A        1       0  1  0  0  0  0  0  1  = 65   "
```

Funkce SRL simuluje dělení osmibitových čísel, zůstatek dělení se zaznamená v CY. Jak posuneme vpravo obsahy např. osmi bajtů v řadě vedle sebe, ukazuje následující krátká rutina (bajty leží na adresách 0100H-0107H):

```
      LD B,08
      LD HL,0107H
      SRL (HL)
ROT:  DEC HL
      DEC B
      JP Z,END
      RR (HL)
      JP ROT
END:  RET
```

Dále se budeme zabývat dvěma velmi speciálními funkcemi rotací celých polovin bajtu (skupin čtyř bitů). Poslouží nám výhodně pro rotaci celých decimálních číslic. Proto byly nazvány Rotate Digit (Rotuj číslo):

Rotace vlevo číselná - RLD

Protože se jedná o rotace mezi dvěma bajty, a to akumulátoru a a místem paměti určeným obsahem reg.HL, ponecháme označení bitů pro akumulátor Dx a bity paměti si označíme Bx.

```
                Akumulátor                 Místo paměti (HL)
Před RLD   D7 D6 D5 D4 D3 D2 D1 D0    B7 B6 B5 B4 B3 B2 B1 B0
Po RLD     D7 D6 D5 D4 B7 B6 B5 B4    B3 B2 B1 B0 D3 D2 D1 D0
```

A tak např.:

```
                Akumulátor                 Místo paměti (HL)
Před RLD   1  1  0  1  1  0  0  0     0  0  1  0  1  1  0  0
Po RLD     1  1  0  1  0  0  1  0     1  1  0  0  1  0  0  0
```

Analogicky rotuje skupinami čtyř sousedících bitů dvou bajtů instrukce:

Rotace vpravo číselná - RRD

	Akumulátor	Místo paměti (HL)
Před RRD	D7 D6 D5 D4 D3 D2 D1 D0	B7 B6 B5 B4 B3 B2 B1 B0
Po RRD	D7 D6 D5 D4 B3 B2 B1 B0	D3 D2 D1 D0 B7 B6 B5 B4

A tak např.:

	Akumulátor	Místo paměti (HL)
Před RRD	1 1 0 1 0 0 0 1	1 0 1 1 1 1 1 0
Po RRD	1 1 0 1 1 1 1 0	0 0 0 1 1 0 1 1

Všimněte si, že operací se neúčastní vyšší "půlka" reg.A. RLD a RRD se používají zvláště při tzv. BCD reprezentaci čísel. BCD je Binárně Kódované Decimální číslo. Jeden bajt může obsahovat dvě decimální číslice, každou z nich binárně kódovanou čtyřmi bity bajtu. Např. BCD reprezentace čísel 83 a 70 je:

1 0 0 0	0 0 1 1	0 1 1 1	0 0 0 0
8	3	7	0

Dále si ukážeme vícebajtovou rotaci BCD čísel mezi adresami 0100H-0103H s vynulováním nižší poloviny 1.bajtu celého řetězce:

```
        LD B,04      ;Čítač čtyř rotací
        LD HL,0100H
        XOR A        ;Vynulování reg.A
ZNOVA:  RLD
        INC HL       ;Na další adresu
        DEC B        ;Už proběhlo 8 rotací?
        JP NZ,ZNOVA;Když ne, skok na adr.ZNOVA
        RET
```

	reg.A		0103		0102		0101		0100		
Počáteční stav	0	0	8	7	6	5	4	3	2	1	(poř.č.
Konečný stav	0	8	7	6	5	4	3	2	1	0	"půlek")

V následujících experimentech si ukážeme příklady použití instrukcí probraných v této kapitole.

EXPERIMENT č.1

Předpokládáme, že jste se již dávno seznámili s tím, co je to ASCII kód znaku. Pokud ne, podívejte se do soupisu znaků v manuálu vašeho počítače. Pro decimální čísla 0-9 je hodnota ASCII kódů 30H-39H (tak např. číslo 5 má ASCII reprezentaci 35H). Kódovaných reprezentací jsme v našem učebním textu měli už celou řadu - ASCII kód je jen dalším v řadě; nejde tedy o nic záhadného. Důležité je, že byl uznán jako mezinárodní standard pro kódy nejužívanějších znaků. Proto není třeba pro každou tiskárnu kódy předefinovávat. Jinými slovy - kdo by vyrobil tiskárnu, která by se neopírala o standard ASCII, musel by co nejrychleji prodat i své poslední kalhoty.

V tomto experimentu si pro procvičení převodu binárního kódu na ASCII provedeme jen jednoduchý převod binárních číslic 1 a 0 na jejich ASCII ekvivalenty 31H a 30H. Obsah reg.B, třeba - 01001101, bude převeden na 8 bajtů paměti v ASCII ekvivalentu - 30 31 30 30 31 31 30 31 (HD čísla).

```
0100  0E09          LD C,08      ;Čítač osmi bitů
0102  210002        LD HL,0200H;1.adresa pro ukládání ASCII kódů
0105  3630   DALSI:LD (HL),30H;Uložení ASCII 30H (nula decim.)
0107  CB40          BIT 0,B      ;Test bitu 0 reg.B
0109  2801          JR Z,NULA    ;Když bit 0=0, skok na adr.NULA
010B  34            INC (HL)     ;Bit 0=1, zvýš 30 na 31 na adr.HL
010C  23      NULA:INC HL        ;Další adr.uložení ASCII
010D  CB18          RR B         ;Rotace pro test dalšího bitu
010F  0D            DEC C        ;Čítač sniž o 1
0110  20F3          JR NZ,DALSI;Už jich bylo 8? Když ne, na DALSI
0112  C9            RET          ;Návrat
```

KROK 1

Před spuštěním programu inicializujte obsah reg.B dle libosti. Pak se podívejte na adresy 0200H-0207H, kde budou uloženy ASCII ekvivalenty binárního obsahu reg.B. Hezkou programovou finesou je předběžné uložení 30H (ASCII reprezentace nuly) na adresu HL. Teprve potom se testuje, zda bit 0 reg.B je ve stavu log.0 nebo 1. Je-li 0, obsah (HL) se nezmění, je-li 1, instrukce INC (HL) zvýší 30H o 1, tedy na správných 31H.

KROK 2

Pokuste se sami upravit program tak, aby prováděl opačný převod (z ASCII na binární) a porovnejte si svůj nový program s následujícím experimentem.

EXPERIMENT č.2

Převod ASCII kódů 30H a 31H na binární tvar (0 a 1). Tedy opačný postup, než jaký byl užit v Experimentu č.1. ASCII kódy jsou na adresách 0200H-0207H, binární ekvivalenty jsou převáděny do reg.B.

```
0120  0E08          LD C,08      ;Cítač osmi
0122  210002        LD HL,0200H;1.adr.ASCII kódů
0125  7E      DALSI:LD A,(HL)    ;Přenos ASCII kódu do reg.A
0126  1F            RRA          ;Bit 0 do CY
0127  CB18          RR B         ;CY do reg.B
0129  23            INC HL       ;Další adresa s ASCII kódem
012A  0D            DEC C        ;Cítač o 1 dolů
012B  20F8          JR NZ,DALSI;Už bylo 8 přenosů? Ne-li, na DALSI
012D  C9            RET          ;Návrat
```

KROK 1

Před spuštěním programu umístěte na adresy 0200H-0207H ASCII kódy 30H a 31H v libovolné kombinaci. Zde je pomocí rotací zjišťován log.stav bitu 0 obsahu adres (HL). Je-li obsahem (HL) 30H, je bit 0=0, je-li to 31H, je bit 0=1. Přenos této informace do reg.B probíhá rotací přes indikátor CY.

KROK 2

Zkuste přepsat program tak, aby odebíral kódy nikoli od adresy 0200H nahoru, ale od 0207H dolů. Jak to ovlivní instrukce rotace?

EXPERIMENT č.3

Ukážeme si převod BCD na ASCII reprezentaci. Každá skupina čtyř bitů BCD čísla bude převedena na jeden bajt ASCII kódu. Např. tyto 4 bajty osahující 8 BCD číslic:

```
0 0 1 1  1 0 0 1  = 39 (BCD)
0 1 0 1  1 0 0 0  = 58 (BCD)
0 0 0 0  0 0 0 1  = 01 (BCD)
0 1 1 1  0 0 0 0  = 70 (BCD)
```

budou převedeny na 8 bajtů ASCII kódu: 33 39 35 38 30 31 37 30 - 33H je ekvivalentem BCD čísla 3, 39H je ekv.BCD čísla 9 atd. Nebudeme tedy převádět celá čísla BCD každého bajtu, ale každou číslici (po dvou z každého bajtu) samostatně. Tomuto pojímání obsahu bajtů s BCD čísly se anglicky říká - packed BCD (česky přibližně soubor BCD číslic).

Program převádí řetězec "packed" BCD bajtů z adres (HL) na ASCII kódy a ukládá je na adresy (DE). V reg.C je počet převáděných BCD bajtů (zde jsou to 4 bajty):

```
0130  3E30          LD A,30H     ;Vyšší 4 BCD bity v reg.A budou 03H
0132  211002        LD HL,0210H;1.adr.bajtu "packed" BCD čísla
0135  110103        LD DE,0301H;1.adr.pro uložení ASCII kódu
0138  0E04          LD C,04      ;Cítač bajtů "packed" BCD bajtů
013A  ED67      BCD:RRD          ;Nižší 4 bity do akum., vyšší 4
                                 ;bity (číslo 03H) už v reg.A jsou
013C  12            LD (DE),A    ;Uložení ASCII bajtu z nižší půlky
013D  1B            DEC DE       ;(DE) pro druhou půlku "packed" BCD
013E  ED67          RRD          ;Vyšší 4 bity do nižší půlky akum.
0140  12            LD (DE),A    ;Uložení ASCII bajtu z vyšší půlky
0141  ED67          RRD          ;Uvedení bajtu BCD do původ.tvaru
0143  13            INC DE       ;Nastavení DE na další převod
0144  13            INC DE
0145  13            INC DE
0146  23            INC HL       ;   "     HL    "     "
0147  0D            DEC C        ;Všechny BCD bajty převedeny?
0148  20F0          JR NZ,BCD    ;Když NE, skok na adr.BCD
014A  C9            RET          ;Návrat
```

KROK 1

Program si projděte krokováním, abyste mu dobře porozuměli.

KROK 2

ASCII kódy budou uloženy na adresách (DE) 0300H-0307H. Reg.DE inicializujeme vždy na adresu o 1 vyšší, protože napřed zpracováváme vyšší 4 bity bajtu BCD. Jde tedy jen o zachování dohodnutého pořadí zpracování BCD číslic. Bajty BCD jsou uloženy na adresách (HL) 0210H-0213H. Znázorníme si to tabulkami:

Adresa	Vyšší 4 bity	Nižší 4 bity	ASCII kód sestaven po:
0210	BCD1	BCD2	
0211	BCD3	BCD4	
0212	BCD5	BCD6	
0213	BCD7	BCD8	
0300	ASCII1		2. RRD
0301	ASCII2		1. RRD
0302	ASCII3		5. RRD
0303	ASCII4		4. RRD
0304	ASCII5		8. RRD
0305	ASCII6		7. RRD
0306	ASCII7		11. RRD
0307	ASCII8		10. RRD

	Akumulátor		Adr. 0210	
Po inicializaci reg.A	3	0	BCD1	BCD2
Po 1.RRD (ASCII2 na 0301)	3	BCD2	0	BCD1
Po 2.RRD (ASCII1 na 0300)	3	BCD1	BCD2	0
Po 3.RRD	3	0	BCD1	BCD2

Po 3.RRD je původní BCD bajt uveden do původního stavu a reg.A opět inicializován do vstupního stavu. Protože po uložení ASCII1 je v reg. DE 0300, a my budeme ukládat ASCII4 na adr.0303, musíme obsah reg.DE zvýšit o 3 (třikrát INC DE).

Ptáte-li se, proč program tak složitě nakládá se skupinami 4 bitů, když by bylo možno převést BCD2 na adresu 0300H, BCD1 na 0301H atd., je to proto, že kdybychom nedodrželi uvedené pořadí, čísla by se vytiskla např.na tiskárně "pozadu", zpřeházeně. Pokud bychom se však přece jen dali tou kratší cestou, museli bychom nějakou přídavnou rutinou pořadí jejich uložení (nebo odeslání na výstup pro tisk) přehodit dodatečně.

Nyní jste již notně nakoukli do kuchyně využití strojně kódového programu v praxi výpočetní techniky. Uvedeným způsobem převedená čísla do ASCII kódu mohou být přímo odeslána na rutiny jejich tisku na obrazovku nebo přímo do tiskové rutiny interfacu připojeného k tiskárně. Způsobů programového zpracování čísel a znaků obecně je nepřeberné množství. Jedná se vždy o jejich kódování a dekódování. Jednou ze zvláštních forem práce s daty (i znaky) je jejich komprimace. Výrazně šetří paměť počítače - hlavně v případě užití databank. Slučuje skupiny písmen, která se tak do paměti nezapisují jednotlivě, ale jako jeden kód. Komprimace je vědou, nabízející řadu variant pro práci s větším počtem dat. Nakonec však vždy před jejich odesláním na tiskárnu nebo obrazovku musejí být převedena na ASCII kód - jinak bychom si vzájemně nic moc nesdělili.

****

KAPITOLA 8

Aritmetické instrukce a blokové prohledávání

****

Po těchto instrukcích si pak odskočíme už "jen pro několik" posledních z říše interfacingu. A to bude - věřte-nevěřte - vše. Jenže to hlavní vás bude teprve čekat - užití nabytých znalostí při vlastním programování. Ale to už bude jen věcí vašeho vlastního umu, opírajícího se o vlastní poznání a nezbytné i nekončící rozšiřování svých vědomostí.

Osmibitové aritmetické instrukce

Provádí se jimi sčítání (ADD, ADC) a odečítání (SUB, SBC) bajtů. K této skupině instrukcí patří i INC (zvýšení obsahu bajtu o 1) a DEC (jeho snížení o 1). Všechny tyto instrukce ovlivňují všechny indikátory. Až na jednu výjimku - INC a DEC neovlivňují stav indikátoru CY. Na to pozor! Často se v tom chybuje a zapomětliví programátoři testují přeplnění obsahu bajtu, na něž u těchto dvou instrukcí nemohou nikdy být upozorněni. Chybu pak většinou hledají tam, kde není. Skupinu uvedených instrukcí doplňují ještě instrukce CP a speciální DAA.

Instrukcemi ADD (SUB) přičítáme k (odečítáme od) akumulátoru obsah druhého bajtu operace se účastnícího. Např. ADD B znamená, že k obsahu reg.A přičteme obsah reg.B. Při SUB (HL) od obsahu reg.A odečítáme obsah adresy (HL). Apod.

Indikátory S,Z,P!V a CY se chovají jak náleží. Opět nezapomeňte, že v případě P!V se jedná o test přeplnění DK čísel. I na to se zpočátku "rádo" zapomíná.

I když indikátory H a N přímo testovat nemůžeme, je jejich stav v tabulkách uveden, protože nám někdy při krokování programem (monitory strojového kódu jejich stav zobrazují) může log.stav H říci, zda došlo či nedošlo k přenosu bitu uvnitř bajtu (mezi bity 3 a 4) - pak je H ve stavu log.1. To je užitečné především při práci s BCD čísly. Indikátor N je ve stavu log.1 vždy, kdykoli je prováděna operace odečítání. Provádí-li se sčítání, je nulový.

Instrukce	HD obsah akumulátoru před	po	Indikátory po provedení S	Z	H	P!V	N	C
INC A	04	05	0	0	0	0	0	NE!
INC A	FF	00	0	1	1	0	0	NE!
DEC A	00	FF	1	0	1	0	1	NE!
ADD 80H	00	80	0	0	0	0	0	0
ADD 80H	80	F0	1	0	0	1	0	0
ADD F0H	F0	E0	1	0	0	0	0	1
ADD 11H	22	33	0	0	0	0	0	0
ADD 18H	29	41	0	0	1	0	0	0
ADD 94H	93	27	0	0	0	1	0	1
ADD 99H	99	32	0	0	1	1	0	1
SUB 33H	33	00	0	1	0	0	1	0
SUB 02H	10	0E	0	0	1	0	1	0
SUB 22H	10	EE	1	0	1	0	1	1

Pokud jste novopečenými adepty programování, pro pochopení průběhu uvedených instrukcí musíte mít buď vysokou představivost, nebo (lépe) tužku a papír a zapsat si vstupní i výstupní binární kódy operací jednotlivých instrukcí. Probereme si třeba instrukci ADD 18H, kde obsah reg.A před jejím provedením byl 29H:

```
  0 0 1 0 1 0 0 1
+ 0 0 0 1 1 0 0 0
-----------------
  0 1 0 0 0 0 0 1
```

Stavy indikátorů:
S=0 (bit 7=0)
Z=0 (výsledek je nenulový)
H=1 (došlo k přenosu z bitu 3 do bitu 4)
P!V=0 (součet dvou DK kladných čísel je opět kladné DK číslo)
N=0 (operace není odečítáním)
C=0 (nedošlo k přeplnění bajtu, resp.k přenosu z bitu 7)

Obdobou instrukcí ADD a SUB jsou: ADC a SBC. Změna je pouze v jediném - k výsledku součtu (resp.od výsledku odečtu) se přičte (resp.odečte) log. stav indikátoru CY. Při CY=0 bude výsledek stejný jako při užití instrukcí ADD a SUB. Bude-li log. stav CY=1, pak při ADC se k výsledku součtu přičte ještě číslo 1, při

SBC se 1 odečte. Záleží na log.stavu CY těsně před provedením instrukce. Příklady:

Instrukce	reg.A před	CY před	reg.A po	S	Z	H	P/V	N	C
ADC 00H	01	1	02	0	0	0	0	0	0
ADC 00H	01	0	01	0	0	0	0	0	0
ADC 90H	97	1	28	0	0	0	1	0	1
ADC 19H	39	1	53	0	0	1	0	0	0
SBC 00H	00	1	FF	1	0	1	0	1	1
SBC 01H	02	1	00	0	1	0	0	1	0
SBC 80H	00	1	7F	1	0	1	1	1	1

(Sloupce S až C: Indikátory po provedení)

Instrukce ADC a SBC jsou velmi užitečné při sečítání a odečítání vícebajtových čísel - právě díky účasti indikátoru CY, který nám pomáhá při přenosu bitu ze "sousedícího" bajtu v jejich řadě. Vzdáleně to připomíná účast CY při vícebajtovém násobení a dělení instrukcemi rotací a posuvů. Vzpomínáte? Jestli ne, neberte učení tak "hákem"! Při programování pak budete učebnicí listovat víc, než by bylo zdrávo; a nakonec se může stát, že to vzdáte. Strojový kód chce klid a koncentraci. Jeho logika je precizní, i když zpočátku těžko stravitelná. Ale jakmile ji jednou pochopíte, všechno půjde takřka samo. O to víc se budete moci věnovat tvůrčímu myšlení, než hledání ve stozích papíru...jakže se to tady, né to není ono, kde jen to...atd. Jediné, co byste měli mít při programování po ruce, je soupis tvarů zápisu jednotlivých instrukcí a jejich vlivu na SI (při precizním časování rutin i prováděcí časy instrukcí). A samozřejmě poznámkový papír pro tvorbu struktury programu.

Nyní si ukážeme sečítání dvou sčítanců, z nichž každý obsahuje 8 bajtů:

```
        LD C,08       ;Počet bajtů v každém ze sčítanců
        LD HL,0200H;Adresa 1.bajtu prvního sčítance
        LD IX,0210H;Adresa 1.bajtu druhého      "
        LD IY,0220H;1.adresa uložení výsledku
        LD A,FFH
        SUB A         ;CY=0 (pro 1.součet)
ADDB:   LD A,(HL)     ;1.bajt 1.sčítance
        ADC (IX)      ;Přičtení 1.bajtu 2.sčítance
        LD (IY),A     ;Uložení výsledku na adr.(IY)
        INC HL        ;Zvýšení všech adres pro další
        INC IX        ;součet
        INC IY
        DEC C         ;Už bylo provedeno 8 součtů?
        JR NZ,ADDB    ;Když NE, skok na adr.ADDB
        JR C,CARRY    ;Pokud CY=1 po 8.součtu, (viz text)
        RET           ;Návrat
```

Funkce programu je naprosto průzračná. Po vstupních inicializacích reg.C,HL,IX,IY a A instrukce SUB A zařídí, aby indikátor CY byl vynulován a nepřešel jako 1 do prvního součtu. V reg.A může být jakékoli číslo mimo 0 (pak by CY byl 1!). Pochopitelně bychom mohli místo instrukcí LD A,FFH a SUB A použít i jiné, které nastaví CY na log.0 (např. SCF a CCF), nebo jen jednu (některou z log.funkcí AND, OR, XOR, které automaticky nulují CY - což by v tomto případě byla volba mnohem vhodnější).

Pokud dojde k přeplnění registru A po některém ze součtů, nastaví se CY na log.1 a ta se pak přičte k příštímu součtu vyššího řádu. Podobně přece počítáme i my, když sečítáme prostá decimální čísla - když nám třeba stovky "přetečou", přesuneme "CY" jako jedničku do tisícovek. Analogie s výše uvedeným programem je zřejmá.

Když se nám objeví CY=1 i po posledním součtu (a už onu 1 nemáme kam přičíst), skočí program na subrutinu CARRY, která tu není uvedena - musí však provést patřičnou "kosmetickou úpravu" výsledku, aby odpovídal pravdě; nebo si s onou 1 může pohrát podle potřeb programu.

Kdybychom tento program chtěli použít pro součet DK čísel, museli bychom jej upravit pro test a přenos indikátoru P/V. Zkuste upravit sami. Ale na počítači! Na papíru se váš program může tvářit jako "milius", ale pak třeba...ouha!

Instrukce DAA

Jde o speciální aritmetickou instrukci pro práci s BCD čísly. Převádí binární číslo obsažené v akumulátoru na formát BCD. Z80 zná jen jedno sečítání a odečítání - binární. V samotné akci Z80 je jedno, zda sečítáme čísla binární, či DK (pro nás se operace liší jen testem patřičného SI - CY nebo P/V). U decimálních

čísel to však už tak jednoduché není. Posuďte sami na příkladu, kde je uveden postup součtů čísel binárních i BCD:

```
0 0 0 0 1 0 0 0   = 8 (decim.) či 08 (packed BCD)
0 0 0 0 1 0 0 1   = 9 (decim.) či 09 (packed BCD)
---------------
0 0 0 1 0 0 0 1   =17 (decim.) či špatně 11 (packed BCD)
```

Binární výsledek je správný. Ale reprezentace BCD je chybná. Důvod leží v číselném základu. Decimální číslo má základ 10 (BCD jej musí mít taky 10). BCD je tu však mikroprocesorem zvažováno jako číslo se 16ti možnými kombinacemi, tedy hexadecimální. U BCD každá skupina 4 bitů tak může chybně reprezentovat jedno HD číslo. Při uvedeném součtu došlo k přenosu z bitu 3 do bitu 4, což signalizuje, že výsledek je větší než 16! Výsledek 11 tedy v tomto případě není BCD, ale HD. Pro správnou reprezentaci BCD při aritmetických operacích by se tedy měl přenos objevit nikoli při dosažení čísla 16, ale 10! Proč věc takhle rozebíráme, se dozvíte za chvíli.

A tak binární součet není BCD číslem,:

1. když součet skupiny 4 bitů je v intervalu 10-15:

```
1 0 0 1    9
0 0 1 0    2
-------
1 0 1 1    B  jako HD číslo (ale mělo by být 11 jako BCD)
```

V tomto případě nedochází k přenosu mezi bity 3 a 4 (H=0)

2. když součet skupiny 4 bitů je větší než 15:

```
  1 0 0 1    9
  1 0 0 1    9
  -------
1 0 0 1 0   12  jako HD číslo (ale mělo by být 18 jako BCD)
```

Zde dochází k přenosu mezi bity 3 a 4 (indikátor H=1), ale pro účely BCD má "zpoždění šesti jednotek". Právě proto byla instrukce DAA vybavena schopností zjistit, kdy dojde k jedné z obou uvedených eventualit a patřičnou šestku přičte k obsahu skupiny 4 bitů. Po přičtení čísla 6 (tedy "dohnání zpoždění") v prvním případě bude binární obsah obou skupin 4 bitů v bajtu 00010001, což je správné BCD číslo 11. Ve druhém případě to bude 00011000, což je BCD reprezentace 18.

V 1.případě, kdy se ve skupině 4 bitů objeví HD číslo A,B,C,D,E nebo F (tedy víc než 9, ale míň než 16), Z80 přičte šestku. Určitou zvláštností je, že zde se 6 přičte, i když se nikde neindikuje žádná změna indikace v reg.F. Indikace je tedy vnitřní, mimo reflexi reg.F.

Ve 2.případě, kdy dojde k přenosu, je tento přenos indikován indikátorem H (při součtu nižších 4 bitů) nebo CY (přeplnění bajtu - při součtu vyšších 4 bitů). Z80 přičte 6 k výslednému obsahu té které "půlky" bajtu. V případě vyšší "půlky" bajtu to z hlediska decimálního můžeme pojímat jako přičtení čísla 60 k celému výsledku. V případě současného výskytu obou případů se vlastně přičte 66.

Funkci instrukce DAA si opět vysvětlíme tabulkově. Použijeme pro to sled dvou instrukcí - ve výsledku DAA provede BCD reprezentaci binárního součtu reg.B a reg.A (binární výsledek je před provedením DAA samozřejmě v akumulátoru):

```
ADD B
DAA
```

Reg.B i A budeme inicializovat na pět různých hodnot:

	1.	2.	3.	4.	5.
Reg.B	11	19	91	99	09
Reg.A před ADD B	22	18	81	88	05
Po ADD B: reg.A	33	31	12	21	0E
H (+6)	0	1	0	1	0
CY (+60)	0	0	1	1	0
Reg.A po DAA	33	37	72	87	14

Ve sloupci 3. a 4. je součet vyšší než 99, což je nejvyšší BCD číslo, jaké může bajt po operaci DAA obsahovat. Jednička stovky tedy "přetekla" (CY=1) a v bajtu zůstal výsledek menší o 100. Toto přeplnění můžeme přenést zase buď do jiného bajtu nebo s ním v programu pracovat nějak jinak - jak zrovna potřebujeme. O použití této řádové jednotky můžete přemýšlet, když si poslední uvedený program multibajtového součtu binárních čísel přeměníte na multibajtový součet BCD čísel - stačí jen když mezi instrukce ADC (IX) a LD (IY),A vložíte DAA.

Pokud provádíme operaci DAA, je třeba mít na paměti, že tak musíme učinit ihned po součtu, protože DAA potřebuje znát log. stavy SI poslední součtové operace (resp. dříve, než je nějaká instrukce změní).

16tibitové aritmetické instrukce

Všechny předchozí instrukce - kromě SUB a DAA - mají své 16tibitové analogie. Liší se v zápisu, registrech, s nimiž operují a ovlivňování SI reg.F. Zásadně si pamatujte, že všechny 16tibitové instrukce INC a DEC neovlivňují vůbec žádné indikátory!!! Proto po jejich provedení není proč zjišťovat stavy SI! Vyvarujete se tak toho, že byste sami sobě kopali jámu, aniž byste o tom věděli.

Instrukce porovnávání - CP r

Funkce CP je shodná s funkcí instrukce SUB - až na to, že po jejím provedení se nezmění obsah akumulátoru (zatímco po SUB se obsah reg.A sníží o obsah bajtu od něj odečítaného). Proto se o této instrukci říká jen, že porovnává. Její užití je velmi výhodné vždy, když testujeme obsah akumulátoru v rozhodovacích uzlech programu, aniž chceme změnit obsah reg.A. Instrukce ovlivňuje všechny SI v závislosti na výsledku porovnání (odečtu). Trošku složitě si můžeme funkci instrukce CP B namodelovat jako tento sled instrukcí:

```
LD C,A     ;Uschování obsahu reg.A do reg.C
SUB B      ;Odečet B od obsahu reg.A
LD A,C     ;Původní stav reg.A zpět do reg.A
```

Akumulátor se nezměnil, ale všechny SI indikují výsledek instrukce SUB B. V jednom z předchozích programů, v němž jsme si demonstrovali konstrukci skokové tabulky, jsme se už s funkcí CP setkali.

Instrukce prohledávání bloku dat

Jsou čtyři - CPI, CPD, CPIR a CPDR. Mají vzdálenou analogii v instrukcích přenosu bloku dat (LDI, LDD, LDIR a LDDR). Ovšem až na některé výjimky a - což je podstatné - jejich funkce je zcela odlišná. Ta obsahuje princip funkce CP, tedy porovnávání. Tím už jsme nakousli, o co půjde:

Reg.A - obsahuje bajt, jehož ekvivalent hledáme v bloku dat
Reg.BC - počet adres, jejichž obsah chceme porovnávat s obsahem reg.A
Reg.HL - obsahuje adresu (u instrukcí s opakováním 1.adresu prohledávaného bloku), jejíž obsah (obsahy) budeme porovnávat s obsahem reg.A

Na rozdíl od funkce CP prohledávání neovlivňuje log. stav CY! Prohledávání ovlivňuje indikátory Z, S a H.

Průběh provedení instrukce CPI (ComPare and Increment):

1. Bajt na adrese (HL) je porovnán s obsahem akumulátoru. Indikátory Z,S,H jsou ovlivněny výsledkem porovnání.
2. Obsah reg.HL je zvýšen o 1 (ve smyslu INC HL).
3. Obsah reg.BC je snížen o 1 (ve smyslu DEC BC) - Indikátor Z=1, když je obsah akumulátoru shodný s obsahem adr.(HL), tedy když výsledkem porovnání je nula. P|V=0, když je obsah BC nulový (jinak je P|V=1).

Průběh provedení instrukce CPIR (ComPare-Increment-Repeat), čili porovnávání s opakováním:

V bodech 1.-3. jako u CPI.
4. Pokud obsah akumulátoru není shodný s obsahem adresy (HL), nebo obsah reg.BC není nulový, instrukce se opakuje opět od bodu 1., dokud nenastane jeden ze zde uvedených testovaných momentů.

Instrukce CPD i CPDR jsou analogické, až na to, že obsah reg.HL je snižován o 1 (ve smyslu DEC HL). U CPI a CPIR jde tedy o prohledávání bloku dat směrem k vyšším adresám, u CPD a CPDR k nižším. Případně nalezený bajt leží na adrese HL-1 (CPIR) nebo HL+1 (CPDR).

EXPERIMENT č.1

Program násobí dvě 16tibitová binární čísla mezi sebou (jsou uložena na adresách 0130H-0131H a 0132H-0133H). Výsledek je uložen na adresy 0134H-0135H. Později si ukážeme i násobení 64-bitových čísel.

```
0100 210000        LD HL,0000     ;Inicializace vstupních hodnot
0103 ED5B3001      LD DE,(0130H)  ;registrů
0107 ED4B3201      LD BC,(0132H)
010B 7A            LD A,D         ;Test - je DE=0?
010C B3            OR E
010D C8            RET Z          ;Když ANO, návrat (výsledek=0)
0110 CB38      NAS:SRL B          ;Když NE, posuv a rotace - CY ob-
0112 CB19          RR C           ;sahuje hodnotu násobícího bitu
0114 3004          JR NC,NCF      ;Je CY=1?
0116 19            ADD HL,DE      ;Ano, pak přičti DE k HL
0117 D8            RET C          ;Při přeplnění po součtu návrat
011A 78        NCF:LD A,B         ;CY=0. Test - je BC=0?
011B B1            OR C
011C CA2901        JP Z,VYSL      ;Když ANO, konec, návrat
011F CB23          SLA E          ;Když NE, posuv a rotace
0121 CB12          RL D
0123 D8            RET C          ;Při CY=1 návrat
0126 C31001        JP NAS         ;Když CY=0, pokračování násobení
0129 223401   VYSL:LD (0134H),HL  ;Uložení výsledku násobení
012C C9            RET            ;a návrat
```

KROK 1

Program si probereme trochu podrobněji, protože se v něm odehrává řada procesů, které na první pohled nemusejí být zřejmé. Princip si ukážeme zjednodušeně na násobení 4-bitovém (lze jej však aplikovat na násobení s libovolným počtem bitů). Předpokládejme, že chceme číslo 0101 násobit 0011 krát (čili třikrát). Jedna z možných procedur provedení operace je shodná s běžnou decimální aritmetikou:

```
          0 0 1 1
        * 0 1 0 1
        ---------
          0 0 1 1    1*0011
        0 0 0 0      0*0011
      0 0 1 1        1*0011
    0 0 0 0          0*0011
    -------------
    0 0 0 1 1 1 1
```

Z toho je zřejmá jedna zákonitost - po každém dílčím násobku se jeho (dílčí) výsledek posune o 1 místo doleva. Násobíme-li "dílčí" nulou, je pochopitelně dílčí výsledek nulový. Ale při násobení jedničkou se na posouvanou pozici promítne obsah násobeného čísla. Tyto dílčí násobky se nakonec sečtou (nebo se mohou sečítat průběžně). Při operacích posuvu vlevo (SRL) naplňujeme registr nulami zprava - což nám plně vyhovuje. V našem programu reprezentuje DE číslo násobené, BC počet dílčích násobení, do HL je ukládán výsledek násobení.

KROK 2

Zkuste projít program s různými HD obsahy reg.DE a BC, např.:

0400*0020=8000
00FF*00FF=FE01
0100*00FF=FF00

Ale co v případě: 0100*0100=????

Výsledek v HL bude 0000, což je špatně. Správně je to 010000, výsledek je tedy tříbajtový (01 00 00), ale my do HL ukládáme dva bajty. Pro tento případ přeplnění by bylo nutno program rozšířit. Testy přeplnění jsou v programu na dvou místech (pomocí instrukcí RET C) - po posuvu násobeného čísla v reg.DE a po přičtení dílčího násobku k reg.HL. Úprava není složitá. Zkuste si ji vytvořit sami.

KROK 3

Předchozí techniku násobení dvoubajtových binárních čísel můžeme lehce převést na násobení vícebajtové. Zkuste pro ně upravit předchozí program a vyzkoušejte si správnost vaší úpravy.

EXPERIMENT č.2

Odečet N-bajtových BCD čísel. Obě čísla jsou uložena od adres XN a YN, kde leží jejich nejnižší bajty. Rozdíl je postupně ukládán od adresy ZN.

```
0200  06NN           LD B,N        ;V reg.B počet odečtů (bajtů)
0202  DD21X2X1       LD IX,XN      ;Inicializace adres
0206  FD21Y2Y1       LD IY,YN
020A  21Z2Z1         LH HL,ZN
020D  37             SCF           ;CY=1;stovkový doplněk pro 1.odč.
020E  3E99    DALSI:LD A,99        ;Hledání doplňku 99 nebo 100
0210  CE00           ADC A,00      ;menšitele (CY=0 nebo 1)
0212  FD9600         SUB (IY)      ;Odečet s CY
0215  DD8600         ADD A,(IX)    ;Přičtení menšence
0218  27             DAA           ;Převod na dec.číslo
0219  77             LD (HL),A     ;Uložení výsledku
021A  DD23           INC IX        ;Přechod na další bajty
021C  FD23           INC IY
021E  23             INC HL
021F  10E8           DJNZ DALSI    ;Na DALSI bajt, dokud není B=0
0221  C9             RET           ;Návrat
```

KROK 1

Vzpomeňte si (pokud se vám to nepodaří, nalistujte!) na odečítání DK čísel - díky jejich "bajtové konstrukci" je DK odečítání shodné se součtem dvojkových komplementů (doplňků) každého z nich. Totéž platí pro čísla BCD - ovšem místo formování DK musíme pro součet určit komplement stovkový (zkráceně 100K) Pokud je vám věc komplementů jasná, mějte náš kompliment. Ale pro jistotu si provedeme názorné příklady:

BCD bajt	03	94	30	01	50
100K	97	06	70	99	50

Až příliš triviální. 100K k decimálnímu dvoumístnému číslu najdeme tak, že toto číslo odečteme od čísla 100. Podobně bychom mohli najít 10K pro "půlky" bajtů, ale to nepotřebujeme. Podívejme se teď na uplatnění techniky sečítání 100K v případě odečítání tříbajtových čísel - třeba 256925-133639:

1. Určíme 100K nejnižšího bajtu čísla odečítaného (menšitele):

100-39=61

2. Výsledek přičteme k nejnižšímu bajtu čísla menšeného (menšence):

25+61=86 (nedošlo k přenosu)

3. Protože nedošlo k přenosu (výsledek není větší než 99), určíme 99K dalšího bajtu menšitele:

99-36=63

4. Výsledek opět přičteme k dalšímu bajtu menšence:

69+63=32 (došlo k přenosu!)

5. Protože došlo k přenosu (tedy přenosu 1 do vyššího řádu), určíme 100K k dalšímu (zde již nejvyššímu) bajtu menšitele:

100-13=87

6. Výsledek přičteme k nejvyššímu bajtu menšence:

25+87=12 (došlo k přenosu!)

7. Výsledek celého 100K součtu (tedy odečtu BCD čísel) je 123286, což je správně. Správnost výsledku je potvrzena log. stavem CY=1. Pokud by byl CY=0, výsledek by byl chybný. Protože však BCD čísla jsou vždy větší než nula, nebo jí rovna, přenos se objeví vždy, kdykoli je menšenec větší než menšitel.

Co do přenosu je sečítání komplementů opačným procesem odečítání nekomplementárních čísel:

Přenos (CY=1) při odečítání čísel je ekvivalentní nulovému přenosu (CY=0) při součtu jejich komplementů.

V našem experimentu není závěrečný test přeplnění indikován. Ale můžete si jej doplnit sami. Otestujte si jeho správnou funkci na počítači.

EXPERIMENT č.3

Dělení 16tibitového čísla (v reg.HL) 8mibitovým (v reg.D). Výsledek dělení se objeví v reg.L, přičemž v reg.H bude zůstatek dělení. Pro to, aby byl výsledek (jeho celočíselná část) 8mibitový ("vešel" se do jednoho registru), musejí být splněny tyto podmínky:

1. 16tibitový dělenec musí mít nejvyšší bit nulový,
2. vyšší bajt dělence musí být menší než dělitel.

```
02FB  21NNNN        LD HL,NNNN   ;Uložení dělence
02FE  16NN          LD D,NN      ;Uložení dělitele
0300  0608          LD B,08      ;Čítač bitů dělitele
0302  1E00          LD E,00      ;Vynulování reg.E pro SBC HL,DE
0304  29      DALSI:ADD HL,HL    ;Posuv obsahu HL vlevo, vynulová-
                                 ;ní nejnižšího bitu
0305  AF            XOR A        ;Vynulování CY
0306  ED52          SBC HL,DE    ;Test pro dělení - je další bit
0308  23            INC HL       ;0 nebo 1?
0309  3002          JR NC,HUP    ;Je-li nula, přeskok na HUP
030B  19            ADD HL,DE    ;Přičtení DE zpět k HL
030C  2B            DEC HL       ;Odečtení 1 od nejnižšího bitu
030D  10F5      HUP:DJNZ DALSI   ;Na DALSI, dokud B není nula
030F  C9            RET          ;Návrat
```

KROK 1

Užitý algoritmus je podobný výtému dělení decimálnímu. Je však jednodušší o to, že počítáme jen s číslicemi 1 a 0. Věc si probereme na dělení 16tibitového čísla 6EH (01101110) 8mibitovým 08H (1000). Je zde analogie s výše provedeným vysvětlením násobení. Stejně tak i při dělení "vynecháváme dělení" při objevení se nulového bitu v dělenci:

```
01101110:1000=1101
-1000
 ----
  1011
 -1000
  ----
   01110
   -1000
    ----
     110
```

Výsledek je 1101 (0DH) se zůstatkem 0110 (06H).

Osvětlíme si, jak uvedený postup probíhá v experimentu. Protože nejvyšší bit reg.HL je nulový (vstupní podmínka), můžeme provést jeho posuv vlevo (to je totéž, co násobení dvěma, tedy totéž, co ADD HL,HL), přičemž se vynuluje nejnižší bit reg. HL. Tímto posouváním (s následnými změnami vlivem dalších operací) se nám nakonec celočíselná část výsledku objeví v reg.L. Porovnáme obsahy HL a DE - když je DE menší nebo rovno HL, výsledek porovnání (v CY) je 1 - provede se instrukce INC HL, kterou se k nejnižšímu bitu reg.L přičte (byl předtím posuvem vynulován) hodnota 1, bude tedy ve stavu log.1. Když je DE větší, bude výsledkem 0 (CY=0) - pak musíme do reg.HL vrátit jeho původní obsah (instrukcí ADD HL,DE) se zrušením předtím zvýšené hodnoty nejnižšího bitu reg.HL - čili se INC HL vyneguje instrukcí DEC HL. Při tom všem nezapomeňte, že obsah reg.E je stále nulový (je tedy poněkud "mimo" hru, nikoli však doslova). Dělení tedy probíhá mezi reg.H a D. Jde o naprostou analogii s výše uvedeným "papírovým" dělením. Posuvem dosahujeme toho, že uvolňujeme postupně nepotřebné bity reg.L, do nichž ukládáme dílčí výsledky dělení. Dílčí zůstatky jsou v reg.H. Po konečném odečtu zůstane v reg.H poslední zůstatek, v reg.L "naposouvaný" celočíselný výsledek. Instrukci SBC HL,DE používáme proto, že v instrukčním souboru Z80 neexistuje 16tibitová instrukce typu SUB. Protože SBC odečítá i hodnotu log.stavu CY, musíme jej před provedením instrukce vynulovat - zde pomocí XOR A.

Opět připomínáme - proveďte si krokování programem - vše se vám rychle vyjasní.

EXPERIMENT č.4

Užití instrukcí porovnávání a blokového prohledávání.

```
; Program 1 - hledání určeného znaku v řetězci různých znaků
;
0400  21000A       LD HL,0A00H   ;1.adresa řetězce
0403  012000       LD BC,0020H   ;Počet znaků v řetězci
0406  3E24         LD A,2AH      ;Hledaný znak (ASCII kód "*")
0408  EDB1         CPIR          ;Blokové prohledání
040A  C20F04       RET NZ        ;Když Z=0, "*" nenalezen
040D  2B           DEC HL        ;HL-1=adresa se znakem "*"
040E  03           INC BC        ;BC+1=pořadí "*" v řetězci
040F  C9           RET           ;Návrat

;Program 2 - hledání tříbajtového řetězce v tabulce dat (šest
;              záznamů, z nichž každý má délku 8 bajtů)
;
0412  31000f       LD SP,0F00H   ;Inicializace reg.SP na adr.0F00H
0415  21000C       LD HL,0C00H   ;Adresa posledního záznamu
0418  010600       LD BC,0006    ;Počet záznamů v tabulce
041B  3A000B       LD A,(0B00H)  ;1.znak hledaného řetěz.do reg.A
041E  11F9FF       LD DE,FFF9H   ;Dvoj.kompl.délky záznamu plus 1
0421  EDA9   DALSI:CPD           ;Porovnání prvních bajtů
0423  2809         JR Z,POR      ;Když O.K.,porovnej další
0425  E2001A       JP PO,KONEC   ;Všechny zázn.otest.? ANO-návrat
0428  19     OBNOV:ADD HL,DE     ;NE-do HL adresa dalšího záznamu
0429  3A000B       LD A,(0B00H)  ;Do reg.A opět 1.znak řetězce
042C  18F3         JR DALSI      ;Skok-porov.1.bajtu dalšího zázn.
042E  3A010B   POR:LD A,(0B01H)  ;Do reg.A 2.bajt řetězce
0431  E5           PUSH HL       ;V HL je poslední adr.předchozího
0432  DDE1         POP IX        ;záznamu-její přenos do reg.IX
0434  DDBE02       CP (IX+02)    ;Porovnání 2.bajtů
0437  20EF         JR NZ,OBNOV   ;Nález? Když NE,obnov.parametrů
0439  3A020B       LD A,(0B02H)  ;Když ANO, porovnání posledních
043C  DDBE03       CP (IX+03)    ;bajtů
043F  20E7         JR NZ,OBNOV   ;Nález? Když NE,obnov.parametrů
0441  23           INC HL        ;V HL 1.bajt obsahu záznamu
0442  C9     KONEC:RET           ;Návrat
```

Oba programy jsou představiteli velmi často užívané metody prohledávání bloku dat. Jejich užití je velmi široké. Např. když potřebujeme vědět, jaké tlačítko bylo stisknuto a na základě toho třeba ještě vyvolat nějakou akci, která je uložena v tabulce (jde tedy o sloučení obou programů dohromady). Prohledávání bloku dat je základním principem všech databázových programů (databank). Prakticky nenajdete program, v němž by se něco neporovnávalo s něčím některou z instrukcí porovnávání - ať bez či s opakováním.

Program 1 demonstruje způsob zjištění, zda bajt dané hodnoty je obsažen v bloku dat. Vše je naprosto jasné, pokud si pamatujete průběh provedení instrukce CPIR (pokud ne, vraťte se k ní). Když nedojde k nálezu, Z=0. V opačném případě byl nález proveden - v tabulce se našel bajt, jehož obsah je shodný s obsahem bajtu v reg.A. Potřebujeme-li znát jeho adresu, musíme snížit HL o 1, pro zjištění jeho pořadí v tabulce zvýšit BC o 1.

Program 2 ukazuje, jaký je princip práce databanky při zjišťování, zda databanka obsahuje hledaný záznam, a v případě, že ano, na jaké adrese je uložen (pro jeho projekci na obrazovku i případné další zpracování). Předpokládejme, že máme v paměti počítače takovouto tabulku záznamů:

			Identifikace			Další element záznamu				
Adresa	Záznam č.	Bajt č.:	1	2	3	4	5	6	7	8
0BD8	1		41	51	46	31	32	33	30	36
0BE0	2		42	46	47	33	36	30	30	34
0BE8	3		41	42	43	36	35	34	32	31
0BF0	4		43	42	41	36	36	36	36	36
0BF8	5		43	41	42	34	33	34	33	34
0C00	6		42	42	42	31	32	33	32	31

Tabulka obsahuje 6 záznamů. HD čísla na jednotlivých adresách tabulky jsou ASCII kódy znaků - v tomto případě čísel a písmen. Nám jde při hledání o obsah dalšího elementu záznamu (bajty 4-8), kterým může být např. telefonní číslo osoby, jejíž šifra je obsažena v identifikačním elementu téhož záznamu (bajty 1-3). Identifikační šifru zde reprezentuje náš řetězec tří bajtů, které porovnáváme s řetězcem tří bajtů (1-3) každého záznamu postupně. Když je hledání úspěšné (nález identifikován), program

nám obsahem reg.HL hlásí, od jaké adresy záznam (jeho 1.bajt) začíná. Tak např. hledáme-li telefonní číslo osoby s identifikační šifrou BBB (ASCII 424242), která je v záznamu 6, v HL bude adresa 0C00H a z adres 0C00H+3 až 0C00H+8 si můžeme vyvolat její telefonní číslo 12321 (ASCII 3132333231). Samozřejmě, že šifru BBB musíme uložit (v ASCII kódu) na adresy 0B00H-0B02H, odkud jsou jednotlivé bajty postupně přenášeny do reg.A pro porovnávání pomocí instrukce CP.

Analýza Programu 2 - HL obsahuje adresu posledního záznamu (0C00H). BC je čítačem počtu záznamů (je jich 6). DK obsah DE je nastaven tak, aby přičítal -7 k obsahu HL. Nezapomeňte, že instrukcí CPD se obsah HL sníží o 1. Proto je nutno od něj odečíst 7, abychom se dostali na 1.adresu dalšího záznamu, který je uložen vždy od adresy o 8 adres nižší než předchozí. Po nalezení shodnosti prvních bajtů se porovnávají další. Pokud bychom chtěli porovnávat delší řetězec, bylo by vhodnější zařadit do programu smyčku. Není-li nalezena žádná shoda bajtů, program se vrátí bez nálezu po provedení instrukce JP PO,KONEC.

Ovšem pozor! Není zde ošetřen jeden mezní stav. Pokud je BC=0 a A=(HL), program se provedením instrukce JR Z,POR vyhne zjištění stavu P/V v instrukci JP PO,KONEC. Při dalším provedení instrukce CPD se obsah BC sníží o 1 (na FFFFH). Tak se program neukončí a bude následovat dalších 65535 exekucí instrukce CPD. A poté znova, "navěky věkův".

Tento případ neošetření mezního stavu v programu jsme uvedli záměrně, abychom vás upozornili na to, že může vést k velmi nepříjemným důsledkům. Mezní stavy se vyskytují prakticky v každém programu. Je na samotném programátorovi, aby všechny mezní stavy svého programu dokázal rozpoznat (předpovědět je). Již nejednou se stalo, že opomenutý mezní stav v profesionálním softwaru se projevil třeba až po mnoha měsících provozu v nějakém podniku. A pokud nedával na první pohled zřejmé chybné výsledky, mohl za čas své nerušené aktivity nadělat v řízení nebo evidenci podniku slušnou paseku. Proto ve svých programech nikdy neopomeňte vychytat všechny potencionální zdroje vzniku mezních stavů a náležitě je ošetřete.

****

KAPITOLA 9

O interfacingu

****

Věřte-nevěřte, právě jste absolvovali všechny instrukce Z80 kromě několika posledních, které se vztahují pouze k příjmu dat z periferních zařízení a odesílání dat na periférie. Jedná se tedy o softwarové ovládání hardwaru pomocí vstupních a výstupních portů. To, jak budou tato data "vypadat", programujete instrukcemi, které jsme si dosud probrali.

Slovo port ve více jazycích znamená přístav. Můžete si port představit jako přístav, do nějž buď z dálky připlouvají data, kde si je vyzvedáváme, nebo jsou do něj expedována před jejich odplutím do hardwarových dálav (periférií).

Jak už bylo v textu uvedeno, interfacing v sobě zdaleka, předaleka nezahrnuje jen pár instrukcí pro komunikaci mezi počítačem a perifériemi. Interfacing je především vědou o hardwaru. A hardware sám je převeliká věda, vyžadující speciální studium. Protože naše učebnice je zaměřená na programování ve strojovém kódu, nemůžeme se tu hardwarem zabývat v míře vyčerpávající téma. Pokud bychom tak chtěli učinit, byl by text nikoli dvakrát, ale nejméně třikrát rozsáhlejší.

Povíme si jen několik základních informací, které jsou naprosto nezbytné pro pochopení funkce instrukcí ve styku s perifériemi.

Mikropočítače komunikují s připojenými zařízeními dvěma typy stykových jednotek - interfaců - paralelními a sériovými. Sám interface je "černou krabičkou", která vykonává funkce nezbytné pro úspěšný převod dat z počítače do periférie, a většinou i ve směru opačném. Vlastní "čistý" vstup a výstup počítačů je konstruován jako tzv. porty, k nimž se interfacy připojují. Stejně jako interfacy, jsou i porty paralelní a sériové.

Zde je nutno poznamenat, že někdy (zvláště v četných computerových časopisech) se nedělá rozdíl mezi slovy port a interface, což není správné. Port může být součástí interfacu, ale taky nemusí. Interfacem se rozumí "černá krabička", která má řadu aktivních i pasivních součástek a její funkce leží především v tom, že data TRANSFORMUJE do podoby, která je pro tu kterou komunikaci nezbytná, a zajišťuje synchronizaci jejich přenosu. Interfacem může být rozuměn dokonce i jen samotný software, který "černou krabičku" může někdy nahradit.

Paralelní port - může být vstupní nebo výstupní. U osmibitových počítačů má pro odesílání nebo příjem dat osmilinkovou datovou sběrnici. To znamená, že na této sběrnici se v každém okamžiku může pracovat současně (paralelně) až s osmi informacemi (log.1 nebo log.0 na každé lince), tedy jedním bajtem.

Sériový port - vstupní nebo výstupní. Datová informace jde jen po jedné lince. Konstrukce jednoho bajtu je tedy možná jen postupně - po lince jde bit po bitu. Tyto porty vyžadují kódovací a dekódovací software, který je schopen podle daného formátu dat (způsobu jejich konstrukce) probíhající informaci kódovat resp.dekódovat tak, aby ani ve vysílajícím, ani v přijímacím zařízení nedošlo k sebemenší destrukci dat.

Porty však potřebují další doplňující linky, kterými se obě komunikující strany informují o průběhu komunikace a zároveň ji synchronizují. Např. při tisku na tiskárně jsou data (řídící kódy pro ovládání tisku a ASCII kódy znaků, které mají být vytisknuty) vysílána z výstupního portu počítače. Tiskárna je však přijme jen tehdy, kdy není zaměstnána jinými pracemi. O tom předává informace počítači po doplňujících linkách. Má-li tiskárna plnou hlavu vlastního tisku, sdělí, že je BUSY - zaměstnána (má napilno) a data nepřijímá. Počítač zastaví jejich odesílání na výstup. Jakmile se však objeví signál, že tiskárna už všechno "snědla", počítač okamžitě po přijetí synchronizačního pulsu začne chrlit data do jejího bufferu (vyrovnávací paměti). Tento přenos je obrovsky rychlý. Po naplnění bufferu daty tiskárna opět oznámí, že je BUSY a celý cyklus se opakuje, dokud má počítač co dodávat. Těmto komunikačním linkám se anglicky vtipně říká handshake - přímý překlad je třesení rukou. Volně si to můžeme přeložit jako tahání za rukáv - tiskárna a počítač se vždy zatahají za rukáv, když po tom druhém něco chtějí: Už! Dělej! nebo Chvilinku počkej!

Výhody paralelního přenosu dat jsou zřejmé - jednodušší software a minimálně osminásobná rychlost přenosu. Proč se tedy vůbec zabývat sériovým? Protože počítače vstoupily do "jednodrátové" civilizace. Jistě by bylo mnohem méně hospodárné měnit všechny existující linky (především telefonní pro připojení modemů) na vícežilové pro každou zúčastněnou stanici. Nezbývá tedy, než se spokojit s tím, co je a počítačový výstup i vstup tomu podřídit. Dále - jak jinak zaznamenat data na jednu či dvě stopy magnetofonu, než sériově? Zdůvodnění existence sériového přenosu dat by bylo možno uvést víc.

Kdy to bude jinak? Až bude počítač pracovat na principu interferometru s koherentním laserovým světlem. Ale to je skok do budoucnosti, o němž se dnes ještě zdaleka neví, jaké všechny převratné změny s sebou přinese. Určitě však přinese změny nejen v přenosu dat, ale především v computerové logice s přímým vlivem na hard- i software. Stručně - zánik binární logiky díky možnosti zpracování dat paralelně (svazek paprsků) jednotlivými stavebními prvky hardwaru (optickými tranzistory - transfázory) v několika logických stavech pro každý z paprsků! Raději nemyslet na hmotnost učebnic programování světelného mikroprocesoru.

Nedá se koherentně svítit - jsme v době polovodičově kamenné. A tak nám, jakožto computerovým neandrtálcům, nezbývá, než se trápit primitivním programováním primitivních krabiček, nad nimiž se naši vnuci budou pobaveně uchechtávat (dokud nebudou ve starožitnostech stát 100krát víc než jejich superstroje).

Porty nejsou ničím jiným, než soustavou hradel (spínacích tranzistorů), které po odeslání bitů na jejich vstupy dávají potřebnou log.úroveň na výstupu, odkud je jejich úroveň předána datové sběrnici, na níž se už přímo připojuje patřičný interface (a naopak). Nesmíte se však domnívat, že komunikace počítače s externími zařízeními probíhá jen z něj vyvedenou sběrnicí (s dalšími linkami). Např. už sám výstup pro připojení obrazovky (TV nebo monitoru) obsahuje přetransformovaná, obrazovým interfacem zpracovaná obrazová data. Tento interface je samozřejmě v počítači. Podobně je tomu s výstupem a vstupem pro záznam a čtení dat z pásku. Interface není nutně jen hardware, ale pod tímto pojmem se skrývá i jeho software (operační systém), bez nějž by hardware nemohl plnit dané funkce.

Některé počítače mají už zabudován paralelní i sériový interface, nebo interface MIDI pro přímé připojení elektronických hudebních nástrojů apod. Proto se také konektor, obsahující přímý výstup adresovaného portu, jmenuje expansion bus (sběrnice s rozšiřitelným využitím, která však může obsahovat i linky mikroprocesoru). Je často vyveden jako výstupek oboustranného plošného spoje - desky počítače - buď z jeho boku nebo zadní části.

Instrukční soubor Z80 umožňuje adresovat (vysílat data nebo je přijímat) maximálně 256 vstupních i 256 výstupních portů. Obvykle se z jejich počtu využívá jen část. Pro ostatní prostě není žádné využití - umíte si snad představit svůj mikropočítač napojený na, řekněme, 230 periferních zařízení? Porty jsou osmibitové. Každý z těchto bitů každého portu může ovládat nějakou z jeho funkcí (nebo o nich podávat informace). To záleží jen na konstrukci počítače a interfacu. Proto pro jejich ovládání potřebujete znát tzv.firmware počítače, v němž výrobce uvádí, jak jednotlivé porty pracují.

Z boje o standardizaci spojení mikropočítačů s interfacy a jejich styku s periferiemi nakonec vyšly vítězně dva interfacy - paralelní Centronics a sériový RS232 (a jejich varianty). Interfacy obou typů mají svůj vlastní operační systém (v paměti ROM nebo EPROM), který se u jednotlivých jejich výrobců více či méně liší - je však vždy jen obdobou téhož. Proto je nutno vědět alespoň to, jak je softwarově ovládat. Každý výrobce ke svému interfacu přikládá manuál, kde je ovládání uvedeno (bohužel často jen v Basicu). Jakékoli hardwarové úpravy však po připojení interfacu Centronics nebo RS232 není třeba dělat a vše by mělo (takřka vždy) fungovat napoprvé. Samozřejmě, že periférie musí být vybavena pro komunikaci s daným typem interfacu (bez ohledu na jeho operační systém).

Veškerá činnost hardwaru musí být perfektně synchronizována. Synchronizační pulsy produkuje krystalový generátor, který je součástí každého počítače. Bez synchronizace by počítač ani jeho periférie nebyly k ničemu. V počítači probíhá neustále řada striktně časovaných dějů s přesně vymezenými průběhy. Zobrazují se jako časové diagramy (pro laika dost komplikovaně vyhlížející), z nichž vyplývá vzájemná závislost jednotlivých logických hardwarových funkcí, odehrávajících se spojitě s časem. Tyto průběhy můžeme měnit různými zásahy - i ty však mají svá pevně stanovená pravidla. Na tomto místě nezbývá, než

odkázat na odbornou literaturu o hardwaru počítačů, interfaců, periférií a jejich vzájemné komunikaci.

Pokud jde o programové ovládání hardwaru, jde opět o nám již známé posílání binárních jedniček a nul na pravé místo v pravý čas a ve správné posloupnosti.

Instrukce vstupní - IN a výstupní - OUT

Vstupními instrukcemi zjišťujeme log. stav osmi bitů zvoleného vstupního portu. Instrukcí IN A,port převedeme po datové sběrnici obsah portu (jeho číslo je v intervalu 0-255) do bitů reg.A. Instrukcí IN r,(C) (r je jeden z registrů A,B,C,D,E,H nebo L) bude převeden obsah portu určeného obsahem registru C do reg.r. Chceme-li např. zjistit stav nějaké periférie, která odesílá do vstupu počítače informace o sobě na vstupní port 255, pak tyto informace můžeme převést do reg.A nebo r těmito způsoby:

```
IN A,FFH      nebo:     LD C,FFH
                        IN H,(C)
```

Jednoduché, že? Nebýt ovšem toho, že vlastní interfacing většinou požaduje, aby v některých případech čtení stavů portů bylo blokováno přerušení, ev. se čeká na puls přerušení, kterých může být řada, a jimiž se mohou vrstveně spouštět a zastavovat subrutiny zpracovávající data, atd... O něčem z toho až dále.

Vstupní instrukce IN pracuje v tomto sledu:

1. Na nižších 8 bitů adresové sběrnice uloží číslo portu a na vyšších 8 bitů " " " v případě instrukce IN A,port předchozí obsah reg.A; při IN r,(C) předchozí obsah reg.B.
2. Aktivují se signály IORQ a RD (linky Z80)
3. Po datové sběrnici převede do reg.A, ev. reg.C, obsah adresovaného portu.

Zvláštnosti popsané v bodu 1 (reg.B jako vyšší bajt adr.sběrnice) se při komunikaci využívá různým způsobem. Záleží na zapojení hardwaru. Vyšším bajtem adresové sběrnice můžeme třeba zjišťovat stav nějakého jiného portu současně s portem adresovaným nižším bajtem této sběrnice. Nebo těmito vyššími linkami poslat nějakou informaci interfacu pro její další transformaci směrem na periférii apod.

Další typy blokových vstupních instrukcí mají svou analogii v instrukcích LDI, CPI, LDIR, CPIR, LDD, CPD, LDDR a CPDR. Samozřejmě, že s některými odchylkami. Všechny se inicializují takto:

reg.B - počet bajtů odebíraných z portu
reg.C - číslo portu
reg.HL - 1.adresa uložení odebraného bajtu

Průběh instrukce INI (Input-Increment):

1. Z portu adresovaného obsahem reg.C se odebere bajt a uloží se na adresu (HL).
2. Obsah reg.B se sníží o 1 (ve smyslu DEC B).
3. Obsah reg.HL se zvýší o 1 (ve smyslu INC HL).
4. Když je obsah reg.B nulový po kroku 2, pak Z=1. N=1 vždy.

Průběh instrukce INIR (Input-Increment-Repeat):
1. Jako INI
2. Jako INI
3. Jako INI
4. Pokud obsah reg.B není nulový, vše se opakuje od bodu 1. Průběh končí, když B=0. Pak program pokračuje jeho další instrukcí. Stav SI tentýž jako u INI.

Analogicky probíhají instrukce IND a INDR (D - Decrement) Rozdíl je jen v tom, že obsah reg. HL se snižuje o 1 (ve smyslu DEC HL).

Pro instrukce výstupní platí analogie s instrukcemi vstupními. Rozdíl je v tom, že místo odebírání dat z adresovaných portů data na adresované porty odesíláme; u Z80 se neaktivuje signál RD (READ - čti), ale WR (WRITE - zapiš).

Výstupní instrukce	analogická ke vstupní instrukci
OUT port,A	IN A,port
OUT (C),r	IN r,(C)
OUTI	INI
OTIR	INIR
OUTD	IND
OTDR	INDR

Tak např. instrukce OUT port,A odešle po datové sběrnici obsah reg.A na adresovaný port - v případě OUT FDH,A na port 253. Apod. Ostatní rozdíly mezi instrukcemi IN a OUT jsou čistě hardwarové.

Instrukce přerušení - DI a EI

V každém počítači probíhá určitý (a ne malý) počet synchronizovaných (převážnou cyklických) dějů, které jsou "vměstnávány" do sebe (resp. časově řazeny). Probíhají skokově, přerušovaně, prolínají se dle stanovených algoritmů svých průběhů. Tak třeba každý počítač po připojení ke zdroji napřed provede kontrolu svého hardwaru. Když tento test najde někde chybu, počítač se "zakousne" a nedá se s ním dělat už nic jiného, než chybu najít a opravit. Pokud test proběhne v pořádku, celý hardware počítače se začne "točit" podle povelů, obsažených v jeho operačním systému. I vstupní test je součástí operačního systému. Ty se mezi různými typy počítačů velmi liší. Snaha o standardizaci systémem MSX zkrachovala. Přišli s ním Japonci a mysleli to dobře. Jenže MSX se stal brzdou vývoje, závažím, které měsíc po měsíci těžklo. A tak máme v operačních systémech slušnou džungli. I když každý na to jde trochu jinak, mají všechny něco společného.

Každý systém musí stále testovat klávesnici (zda je některé - a jaké - z tlačítek stisknuto); stále je nutno tvořit televizní obraz pro zobrazení výsledků programových instrukcí, jež jsou ukládány do obrazové paměti počítače; probíhá řada testů a změn.

U mikropočítačů bývá zvykem, že klávesnice je testována jednou za padesátinu vteřiny (to je čas pro tvorbu jednoho TV půlsnímku). Všechny průběžné procesy ovládané jedním mikroprocesorem nemohou probíhat souběžně. Jejich cyklický sled je dán operačním systémem. To je umožněno cyklicky se opakujícími instrukcemi přerušení a jejich módy - probereme si je níže. Pokud se vám zachce mikroprocesor "vyhodit" ze systému podléhajících operačních cyklů, provedete instrukci DI (Disable Interrupt) - blokování přerušení. Jí vyřadíte z činnosti operační systém řízený mikroprocesorem. Většinou tím však nemusíte (záleží na počítači) vyřadit např. tvorbu obrazu. Z toho vyplývá, že kdykoli se vám nehodí, aby operační systém prováděl své operace, jednoduše jej vyřadíte z účasti instrukcí DI. Otevření (uvolnění) přerušení obstará funkce EI (Enable Interrupt). To se však nevztahuje k vnějšímu, nemaskovatelnému přerušení (viz dále).

Funkci DI zařazujeme do programu zásadně vždy, kdykoli by maskovatelné přerušení běhu programu přineslo destrukci výsledku prováděných operací - především v případech provádění časově kritických operací jako je záznam a čtení dat při komunikaci se záznamovým nebo přenosovým zařízením. Pokud by do ní vstoupilo nevítané přerušení, tvorba formátu přenášených dat i jejich dekódování by přinášely naprosto chybné výsledky.

Kdykoli potřebujeme znovu otevřít přerušení, zařadíme do programu instrukci EI. K ní se váže speciální instrukce návratu z obslužné rutiny přerušení - RETI - pro návrat z maskovatelného přerušení. V některých případech však místo ní lze zařadit nám už dobře známou RET. K návratu z nemaskovatelného přerušení slouží instrukce RETN.

Módy přerušení a jejich instrukce

Napřed si probereme dva základní typy přerušení (nikoli módy!) - maskovatelné a nemaskovatelné.

Představte si sled níže uvedených dějů:

1. Čtete si knížku.
2. Začne zvonit telefon.
3. Odložíte knížku a zvednete telefon. Začnete s někým mluvit.
4. Vtom zazvoní zvonek u dveří.
5. Odložíte sluchátko a jdete otevřít.
6. Vyřídíte záležitost u dveří a vrátíte se.
7. Zvednete sluchátko a dokončíte hovor.
8. Vezmete si knížku a budete pokračovat ve čtení.

Vaše čtení bylo přerušeno telefonem. Telefonní hovor byl přerušen zvonkem na dveřích. Vždy jste reagovali tak, že jste předchozí činnosti postupně přerušovali a šli vstříc novým podnětům. Po posledně vstoupivším přerušení jste pak ve zpětném sledu vyřizovali, co s sebou jednotlivá přerušení přinesla. Znázorníme-li si uvedené hlavní činnosti písmeny A,B,C, pak (už

bez dalších přerušení) jste akce dokončili v pořadí C,B,A, přičemž v A (čtení) setrváváte, s nadějí, že už nebudete rušeni. Ale když se rušení objeví, budete je akceptovat.

Máte však jednu šanci - když nechcete zvednout telefon ani jít otevřít dveře, prostě nebudete na podněty reagovat. Budete se "maskovat". Nepřipustíte přerušení své momentální činnosti. Tento druh přerušení se nazývá maskovatelný. Je umožněn instrukcí DI - pak mikroprocesor jakoby ignoruje přicházející přerušení a nereaguje na ně. K přijetí přerušení a reakci na ně jej zase přimějete instrukcí EI.

Co však, když si čtete a začne hořet dům? Zřejmě budete reagovat okamžitě. Tomuto druhu přerušení se říká nemaskovatelné. Mikroprocesor je nemůže ignorovat, a kdykoli se objeví, okamžitě na ně reaguje. Toto vnější přerušení se užívá především pro případy havarijních situací (výpadek proudu, porucha nějakého zařízení apod.).

Nyní si sesumarizujeme obecné způsoby řízení mikroprocesorů v interfacingu. Jsou dva:

1. **Softwarové řízení** prostřednictvím programu, obsahujícím subrutiny, skoky, volání a návraty. Program nepravidelně nebo cyklicky zjišťuje, zda periférie vyžaduje nebo povoluje styk.

2. **Soft-hardwarové řízení** programem, který:
a - obsahuje módy přerušení:
Během každého provedení instrukce mikroprocesor zároveň zjišťuje, zda se neaktivuje vstup přerušení. V případě, že ano, právě prováděnou instrukci dokončí a poté vstoupí do daného módu přerušení.
b - umožňuje provedení DMA procesu:
DMA (Direct Memory Access) - přímý přístup do paměti. Používá se pro rychlý přenos bloků dat z periférie (i vedlejší paměti) do počítače a naopak. Rychlost přenosu, prováděná speciálními obvody, je cca 1,1 MB za vteřinu. Během tohoto přenosu je mikroprocesor vyřazen z činnosti; po jeho skončení ve své činnosti pokračuje.

Uvedená řízení si převeďte na analogii s obsluhou telefonu. Pod bodem 1. budete stále zvedat a pokládat sluchátko, abyste zjistili, zda někdo nevolá (zvonek je vypnut). V bodu 2.a reagujete na vyrušení zvonkem telefonu. U 2.b máte k telefonu připojenou tzv. automatickou sekretářku. Přenos probíhá "mimo vaše vědomí". Obsah vzkazu si přehrajete, až budete chtít nebo potřebovat.

Z hlediska hardwarového se externí zařízení dožadují komunikace přerušením ve třech provedeních:

- Přerušení po jedné lince
Jakmile CPU zjistí přítomnost signálu přerušení na svém vstupu přerušení, ihned přeruší momentální činnost a věnuje se periférii. Tato metoda je nejrychlejší, když je k počítači připojeno jen jedno externí zařízení. Při jejich větším počtu je většinou nutno provést zjištění, které z nich si přerušení vyžádalo. K jedné lince může být připojeno libovolné množství periférií.

- Přerušení po více linkách
Každá periférie má svou linku přerušení. Mikroprocesor tak při jeho objevení se nemusí zjišťovat, které z nich se dožaduje komunikace. Počet připojitelných periférií je závislý na počtu linek přerušení mikroprocesoru (běžné mají max.4 takové linky) - netýká se Z80.

- Vektorovaná přerušení
Po vyslaném signálu přerušení externí zařízení vyšle po sběrnici buď jednobajtový operační kód instrukce k jejímu okamžitému provedení (IM 0), nebo část vektorové adresy, na níž je ihned převedeno programové řízení (IM 2). Rutině, která pak bude provedena, se říká obslužná rutina přerušení.

Nemaskovatelné přerušení je identifikováno vstupním hradlem Z80 NMI. Aktivuje se úrovní log.0. Tehdy Z80 přeruší jakoukoli svou dosavadní činnost a okamžitě převede programové řízení na adresu 0066H. Od ní je uložena rutina tohoto přerušení. De facto je signál NMI analogií instrukce CALL 0066H. De iure je však "pachatelem" periferní zařízení. Návratu z rutiny slouží instrukce RETN.

Mikroprocesor je vybaven dvěma flip-flopy pro blokování či otevření přijetí signálu maskovatelného přerušení - IFF1 a IFF2. Jejich log. stav je přímo nastavitelný instrukcemi DI a EI. DI je nuluje (blokuje), EI nastavuje na 1 (uvolňuje přerušení).

Okamžikem přijetí maskovatelného přerušení se stav obou IFF automaticky překlopí do log.0. IFF2 slouží k uchování log.stavu IFF1 v případě, kdy se aktivuje signál NMI. Instrukce RETN pak převede stav IFF2 do IFF1. Log.stav IFF2 lze převést do indikátoru P!V provedením instrukce LD A,I nebo LD A,R a testovat podmínkami PE (0) a PO (1). Stav IFF1 testovat nelze.

Módy maskovatelného přerušení jsou tři. Každý z nich má svou instrukci: IM 0, IM 1, IM 2. Jejím provedením se aktivuje jí určený mód přerušení (nastavením kombinace na dalším flip-flopu IMFa-IMFb).

Mód 0 - periférie pošle na datovou sběrnici 1 bajt, který Z80 čte jako instrukci. To však neznamená, že instrukce předaná počítači může být jen jednobajtová. Kombinací bajtu s jinými lze vytvořit instrukci vícebajtovou.

Mód 1 - je operačním ekvivalentem instrukce RST 38H.

Mód 2 - je o něco málo komplikovanější než ostatní. Periférie vyšle na datovou sběrnici 1 bajt, který tvoří nižší část adresy vektorové tabulky adres obslužných rutin. Vyšší část adresy je v reg.I (zvaném Interrupt Vector - vektor přerušení). Na takto formované adrese leží nižší bajt a na adrese o 1 vyšší pak vyšší bajt adresy obslužné rutiny tohoto přerušení. Jinými slovy - na vektorové adrese a adrese o 1 vyšší leží dva bajty pro doplnění adresy instrukce CALL XXXX, která je poté ihned (automaticky!) provedena. Pokud neměníme inicializovaný obsah reg.I (řekněme, že jeho obsah je třeba AAH), pak při různých obsazích bajtů posílaných z periférií na datovou sběrnici v intervalu 0-255 můžeme mít v počítači až 256 obslužných rutin - bajty jejich 1.adresy jsou na adresách AA00H a AA01H, poslední na adresách AAFFH a AB00H. Samozřejmě, že i reg.I můžeme v průběhu programu měnit. Ale takové množství (65536) obslužných rutin by se nám do počítače ani nevešlo!

Pro lepší pochopení funkce IM 2 si provedeme příklad. Dejme tomu, že potřebujeme, aby poté, co se po přerušení v tomto módu na datové sběrnici objeví bajt 01 (odeslaný z periférie do počítače), byla spuštěna rutina od adresy 1234H. Napřed musíme mít v reg.I vyšší bajt adresy vektorové tabulky, na níž leží nižší bajt adresy 1234H, tedy 34H. Opět si můžeme zvolit...tak třeba FFH. Tzn., že na adrese FF01H bude bajt 34H, na adrese FF02H bajt 12H. Přenos FFH do reg.I provedeme takto:

```
LD A,FFH
LD I,A
```

Do místa programu, od nějž budeme "souhlasit" s možností přerušení periférií v módu 2, zařadíme instrukci IM 2. Nyní, kdykoli si bude chtít (nebo potřebovat) periférie s námi popovídat, vyšle na flip-flop IFF1 signál přerušení. Protože se jedná o přerušení maskovatelné, pro jeho provedení musíme mít uvolněno přerušení - nesmí tedy být blokováno působením instrukce DI. Pokud je tomu tak, musíme provést instrukci EI. Je-li vše v pořádku, na datové sběrnici se objeví bajt s obsahem 01. K němu se připojí obsah reg.I a spolu vytvoří adresu FF01H. Mikroprocesor z této vektorové adresy odebere do reg.PC (jeho nižší poloviny) bajt 34H; a z vektorové adresy o 1 vyšší, tedy z FF02H bajt 12H a uloží jej do vyšší části reg.PC. V něm se tak vytvoří adresa 1234H, na níž (protože je v reg.PC) bude okamžitě převedeno programové řízení. Adresa návratu již byla před realizací přerušení uložena do zásobníku. Po provedení obslužné rutiny přerušení od adresy 1234H bude proveden návrat (instrukcí RETI) tam, kde byl program přerušen, a odkud bude opět pokračovat (až do objevení se dalšího přerušení). Schématicky:

```
           Registr I              Datová
                                 sběrnice
        ---------------       ---------------
        !    FFH      !       !     01      !
        ---------------       ---------------

                     Obsahy  adres
                    ---------------
                    !             !
                    ---------------
                    !             !
                    ---------------
  Adresa FF01H      !     34H     !    Nižší bajt a
                    ---------------
         FF02H      !     12H     !    Vyšší bajt adresy
                    ---------------    obslužné rutiny,
                    !             !    na niž bude pře-
                    ---------------    vedeno programové
                    !             !    řízení
                    ---------------
```

Nechcete-li, aby sama obslužná rutina byla přerušena dalším možným maskovatelným přerušením, na dobu jejího provádění zablokujte přerušení instrukcí DI. Na konci rutiny musíte pak za sebe zařadit instrukce EI a RETI. Přitom je dobré vědět, že uvolnění přerušení nastane až po vykonání instrukce naásledující - zde RETI - tedy nikoli hned po EI. Zatímco instrukce DI blokuje přerušení ihned po svém provedení. Nezapomínejte rovněž na uschování obsahů všech registrů (instrukce PUSH) na začátku každé obslužné rutiny! A před jejím skončením na jejich zpětný přenos do registrů (instrukce POP). Nikdy totiž nemůžete vědět, v jakém místě programu k přerušení dojde.

Instrukce zastavení - HALT

Tato instrukce zastaví běh programu až do objevení se jakéhokoli přerušení - ale jen tehdy, je-li přerušení uvolněno (EI). Po provedené operaci s přerušením související pak program pokračuje (není-li stanoveno jinak) provedením instrukce umístěné hned za HALT. Při zablokovaném přerušení (DI) se běh programu obnoví jen při aktivaci linky NMI (nemaskovatelném přerušení). Pokud tedy použijete tuto zvláštní instrukci po DI a nemáte možnost aktivovat NMI, program se vám zastaví navždy. Proto pozor při jejím užití. Instrukce HALT během svého působení provádí stále dokola "NOPy", takže ve výsledku se "nic neděje".

EXPERIMENT poslední

Ukázka užití IM 0 a tří obslužných rutin v IM 2. Bohužel si ji na počítači nemůžete demonstrovat, pokud k němu nemáte připojena tři experimentální periferní zařízení.

```
; Uložení adres obslužných rutin přerušení do vektorové tabulky
; těchto adres (tabulka je na adresách FF00H-FF06H)
;
  DEFW: 1234H            ; 2 bajty na adresách FF00H a FF01H
        2345H            ; 2    "          "    FF03H a FF04H
        3456H            ; 2    "          "    FF05H a FF06H
;
;==================================================================
;
 START: DI               ;Blokování maskovatelného přerušení
        LD A,FFH         ;Vyšší bajt vektorové adresy do reg.A
        LD I,A           ;Jeho přenos do reg.I
        EI               ;Uvolnění přerušení
        IM 0             ;Nastavení na mód přerušení 0
        HALT             ;Čekání na signál přerušení
        IM 2             ;Nastavení na mód přerušení 2
        HALT             ;Čekání na signál přerušení
        JP PROGRAM       ;Skok na adresu PROGRAM
;
;==================================================================
;
; OR1 začíná na adrese 1234H
;
  OR1: DI                ;Blokování maskovatelného přerušení
       PUSH AF           ;Uložení obsahů všech reg.do zásobníku
       PUSH BC
       PUSH DE
       PUSH HL
       PUSH IX
       PUSH IY
;
; Zařazení jakékoli subrutiny, která bude provádět požadovanou
; funkci při aktivaci přerušení z periférie 1
;
       POP IY            ;Reinicializace obsahů registrů
       POP IX
       POP HL
       POP DE
       POP BC
       POP AF
       EI                ;Uvolnění přerušení
       RETI              ;Návrat z obslužné rutiny přerušení
;
;==================================================================
;
; OR2 začíná na adrese 2345H
;
  OR2: EI
;
; Následuje uložení obsahů všech reg.do zás.(PUSH) jako u OR1
; Zařazení obslužné subrutiny pro přerušení z periférie 2
; Přenos uložených bajtů ze zásobníku zpět do registů (POP)
;
       RETI              ;Návrat
```

```
;======================================================================
;
; OR3 začíná na adrese 3456H
;
  OR3: EI
;
; Opět následuje uložení obsahu registrů (PUSH) jako u OR1
; Zařazení obslužné subrutiny pro přerušení z periférie 3
; Přenos uložených bajtů ze zásobníku zpět do registů (POP)
;
        RETI              ;Návrat
```

KROK poslední

Hardwarové jsou k počítači připojena tři periferní zařízení, číslovaná 1, 2 a 3. Na začátku rutiny START je blokováno přerušení (DI), aby jeho případné objevení se nevstoupilo do programu dříve, než je provedena inicializace registru I. Dále je přerušení uvolněno (EI) a nastaven IM 0. Poté, kdy se na IFF1 objeví signál přerušení z některé periférie, provede se instrukce, jejímž operačním kódem bude bajt, periférií odeslaný na datovou sběrnici - mohou to být např. jednobajtové instrukce RST, IM, LD, logické, aritmetické atd. Po vykonání instrukce (pokud nás nezavede jinam, bez návratu na další instrukci rutiny START) je mód přerušení změněn na IM 2. Jednotlivá periferní zařízení (PZ) při dožádání se spojení v tomto módu přerušení posílají na datovou sběrnici (např.) tyto bajty:

PZ1	00
PZ2	02
PZ3	04

Tak např. když si komunikaci vyžádá PZ2, na datové sběrnici bude bajt 02. S obsahem reg.I bude sestavena vektorová adresa FF02. Z této adresy a adresy o 1 vyšší budou přeneseny 2 bajty do reg.PC, vlivem čehož přejde programové řízení na adresu 2345H. Na ní je obslužná rutina OR2, která po vykonání svých funkcí provede instrukci návratu RETI a vrátí se zpět tam, kde bylo přerušení aktivováno.

Zde si povšimněte toho, že zatímco v rutině OR1 je zpočátku zablokováno přerušení (rutina se tedy provede "v jednom kuse" celá - pokud se neobjeví nemaskovatelné přerušení), OR2 a OR3 mohou být během svého provádění kdykoli přerušeny maskovatelným (a samozřejmě i nemaskovatelným) přerušením. Průběh jednotlivých rutin (jejich prolnutí) je pak provedeno ve sledu analogickém s naším výše uvedeným příkladem se čtením, telefonem a zvonkem u dveří. Tomuto řazení exekucí obslužných rutin se říká "nested" - má svou vzdálenou analogii v ukládání a odebírání dat ze zásobníku - první dovnitř, poslední ven. Protože před vstupem do obslužné rutiny je z reg.PC do zásobníku uložena adresa návratu k instrukci, po jejímž provedení bylo přerušení přijato, je nutno brát ohled na to, aby nám při užití tohoto typu prolínání rutin zásobník nakonec neprolnul do programu.

Prolínání obslužných operací umožňuje např. zapojení typu "daisy chain". Jedná se o řazení periférií do řetězce podle stupně jejich priority, přičemž periférie s vyšší prioritou může na čas potřebný pro svou obsluhu přerušit obsluhu periférie s prioritou nižší. V obou uvedených případech se jedná o specifické soft-hardwarové řízení komunikace, které je nutno řešit případ od případu. Zde nezbývá, než odkázat na odbornou literaturu.

Zpět k programu - po prvním maskovatelném přerušení v IM 2 (čeká se na něj opět po instrukci HALT), následném provedení patřičné obslužné rutiny a návratu z ní zpět do rutiny START, je provedena instrukce skoku do libovolného programu (JP PROGRAM), který někde v počítači máme. Jeho průběh bude přerušen vždy, kdykoli si to některé z PZ bude přát (samozřejmě jen tehdy, bude-li přerušení uvolněno).

Připojíme-li nyní k počítači (na linku NMI) další periférii, bude aktivací této linky přerušen chod programu vždy, bez ohledu na cokoli, a programové řízení převedeno na adresu 0066H. Pro doplnění - nejvyšší prioritu ve struktuře přerušení má aktivace linky BUSREQ, kterou se zajišťuje spolupráce Z80 např. s koprocesory.

Může se vám zdát, že v případě aktivace NMI a IM 1 je výběr jeho následků značně chudý - "jen" skok na adresy 0066H nebo 0038H. Nezapomeňte však, že v těchto rutinách můžeme programově cokoli stále měnit v závislostech, které si sami stanovíme. Proto se i výsledky, které provedení těchto dvou rutin s sebou přinese, mohou vždy něčím lišit. Pokud máte počítač, který má na spodních adresách pevně zabudovanou ROMku, budete v jejich

využití omezeni - ale jen do té doby, než využijete možnosti stránkování paměti pro připojení vedlejší paměti s vaším vlastním operačním systémem.

V programu uvedená komunikace je jednosměrná, navíc velmi chudá (je tu jen jedno prosté "zatahání za rukáv", dokonce bez přenosu dat mezi počítačem a PZ). V praxi vše probíhá členitěji. Především komunikace sama většinou bývá obousměrná. Firma Zilog (autor Z80) vyrábí pro interfacing se Z80 speciální integrované obvody - PIO a SIO (s porty pro paralelní a sériový přenos dat).

Prakticky jakýkoli integrovaný obvod se může stát součástí interfacu nebo periférie. Podobně jako programování, je stavba hardwaru činností tvůrčí, tedy bez limitních hranic - vše jde vždy řešit různě - standardně, složitě, jednoduše...ale i - a o to by mělo jít především - chytře, neotřele. Právě v tom tkví výjimečný úspěch některých počítačů. Jsou totiž po stránce hardwarové (spolu s jejich chytře komponovaným operačním systémem) řešeny lépe, zajímavěji než jejich předchůdci, rozšiřují možnosti využití techniky, kterou reprezentují. Jsou-li i po stránce marketingu dobře zabezpečeny, začneme se brzy dozvídat, že obsazují první příčky computerových žebříčků obliby i prodejnosti.

Pokud budete někdy chtít vymýšlet a konstruovat přenosová zapojení, doporučujeme věnovat se přenosu paralelnímu; sériový je neporovnatelně komplikovanější. V každém případě se do ničeho nepouštějte, dokud nezvládnete teorii hardwaru. Vyhnete se tak pohledu na dýmající počítač.

Nakonec jsme si nechali instrukci zcela nejjednodušší:

NOP

Znamená No Operation, tedy žádná operace. Po připojení ke zdroji má počítač těchto "nopů" plnou paměť RAM. HD kód instrukce NOP je 00. I když na adrese, obsahující samé nuly, vlastně nic není, nelze to brát tak doslova. Provedení této instrukce, spočívající v jejím dekódování, zabere 4 cykly. Z toho vyplývá i její užití v časovacích rutinách a cyklech, kde pomocí ní ladíme časově kritické operace. Její provedení nemá žádný vedlejší účinek.

****

KAPITOLA 10

Závěr

****

Závěrem několik praktických rad pro programování ve strojovém kódu, resp.assembleru.

*** Informujte se o tom, který generátor i monitor strojového kódu je pro váš typ počítače nejlepší. Udělejte vše pro to, abyste je získali (i s manuálem!). Naučte se s nimi pracovat tak, abyste manuál už vůbec nepotřebovali - obvykle to dá trošku zabrat, ale budete-li se studiu manuálu i praxi plně věnovat, nemělo by vám to trvat déle než 2-3 týdny. Začínající programátoři se často naučí zacházet jen s monitorem, který je "čtivější a koukavější". Tím si však stelou Záhořovo lože. Programování samotným monitorem je velmi zdlouhavé a zvyšuje možnost nasekání chyb, které se pak špatně hledají. Možnost užití návěští v generátoru (i jazyku assembler místo HD kódů) je prostě k nezaplacení. Jakmile si na váš generátor a monitor zvyknete, pracujte jen s nimi. Všechny kvalitní generátory i monitory umějí přibližně totéž, ale mívají odlišné ovládání. Proto je zbytečné po nich pokukovat. Jen v případě, že by se pro váš počítač objevila nějaká softwarová hvězda, přeorientujte se.

*** Každý kousek vytvořeného programu si nahrajte před tím, než vyzkoušíte jeho funkci. Počítejte s tím, že většinou vám nic nebude fungovat napoprvé. Tady se riskovat nevyplatí ani trochu. Chyba v programu v 99 procentech případů znamená jeho odchod do věčných lovišť!

*** Stále si veďte dokumentaci vytvářených programů. Jednak vám usnadní orientaci při práci, a pak, když se budete chtít k některému z nich po nějakém čase vrátit, bez dokumentace budete muset celý program zdlouhavě "luštit", protože se v něm už nevyznáte! Nejlepší dokumentací je komentovaný výpis a vývojový diagram programu. Být pečlivý v archivaci znamená šetřit svůj čas i zdraví. Ztracený "papírek s důležitými poznámkami" vás snadno přivede k zuřivosti.

*** Dávejte pozor na záludnosti, které mohou vyplynout z assemblerových "dvojsmyslů". Píšete-li si nějaký kousek programu tužkou na papír, buďte pečliví. Třeba LD B,0A (i když by správněji mělo být LD B,0AH; používá se i forma LD B,#0A) budeme vždy chápat jako přenos čísla 0AH do reg.B. Napíšete-li v roztržitosti jen LD B,A, může se vám stát, že později budete zápis chápat jako přenos obsahu reg.A do reg.B. Když takovou chybu přenesete do programu, těžko ji pak budete hledat. Proto hexadecimální obsah jednoho bajtu zapisujte vždy jako číslo dvoumístné.

*** Pokud ještě nemáte tiskárnu, pořiďte si ji co nejdříve. Programování bez ní je o poznání obtížnější a zdlouhavější. Krátké výseky výpisu programu na blikající obrazovce vám nepoví tolik, co listing programu z tiskárny. O přehlednosti oproti propisovačkou popsaných a proškrtaných dokumentů nemluvě.

*** Jedním z největších kamenů úrazu při vlastním programování je zásobník, do nějž se ukládají a z něj odebírají adresy návratů. Do něj samozřejmě můžeme ukládat i leccos jiného a vůbec s ním všelijak operovat. Má-li váš program už slušnou délku (pár set bajtů) a vzdorovitě krachuje, přestože se vše zdá být v pořádku, proberte si pečlivě (ale opravdu pečlivě!), co se děje v zásobníku během programu. Takřka stoprocentně najdete chybu právě tam.

*** S tím souvisí i ztráta orientace v přenosech parametrů mezi rutinami, subrutinami i jinde. Jako u programu ve strojovém kódu obecně, stačí jeden chybný přenos a dílo zkázy je hotovo.

*** Vysoký stupeň obezřetnosti vyžaduje ošetření mezních stavů, které obsahuje prakticky každý program. Jedná se o programové situace s nízkým až velmi nízkým stupněm pravděpodobnosti jejich výskytu. Chyby, které přinášejí, se nemusejí nutně projevit ihned, ani nemusejí být příliš patrné. Mohou, ale nemusejí se řetězit. Neošetřená místa programu, která jsou potencionálním zdrojem takovýchto chyb, můžeme nazvat "studenými spoji" softwaru.

*** Další časté chyby bývají "zaklety" v ležérním přístupu ke stavovým indikátorům. Až příliš často se testuje netestovatelné - např. indikátor Z nebo CY při DEC a INC párových registrů nebo CY při DEC a INC jednobajtových registrů atd. Pokud se nenaučíte vlivy instrukcí na indikátory zpaměti (což by bylo nejlepší),

mějte při programování po ruce tabulku, v níž je uvedeno, jak které instrukce indikátory ovlivňují. Druhou, i když ne přímo programovou chybou, je nevyužívání ovlivňování indikátorů různými instrukcemi. Program se tak zcela zbytečně "nafukuje", stává se neefektivním a "přihlouplým".

*** S předchozím odstavcem přímo souvisí další typ programátorské ležérnosti. Jak běží čas, programátor si oblíbí některé instrukce "na úkor" jiných - na jejich existenci postupně pozapomene, až se mu "vykouří" z hlavy úplně. Jeho programy pak řeší řadu programových problémů oklikou, místo aby šly na věc přímo. A už vůbec takový programátor nemůže vytvořit program, který by byl zajímavý, tvůrčí, protože nepracuje se všemi možnostmi (ani si je nemůže uvědomit), které mikroprocesor nabízí. Proto si čas od času prohlédněte soupis instrukcí, jestli se tam někde "nekrčí" nějaká pozapomenutá.

*** Všechny rutiny se snažte vymyslet tak, aby byly relokovatelné. Velmi vám to pomůže při jejich rychlém, nekomplikovaném zařazování do jiných programů, protože nebudete muset přepisovat adresy skoků, volání apod. Speciálně rutinky, které můžete použít ve spojení s řadou jiných programů, by měly být relokovatelné "bez diskuse" - např. rutina přenosu obrazových bitů na tiskárnu, překódování souboru ASCII znaků z různých slovních procesorů atd. (viz dále i stavebnicové programování).

*** Rozfázujte si vytvářený program na jednotlivé "pohyby" jako políčka filmu. Tyto části převeďte do subrutin programu. Při jeho vytváření pak zjistíte, že některé subrutiny můžete sloučit - "udělat z nich jednu". Kdybyste program psali jako milostný román, nejen že by se vám tato řešení ani neobjevila, ale už po několika desítkách bajtů by vám tvorba začala výrazně "drhnout". Pamatujte - napřed struktura, pak teprve instrukce!!!

*** Pro programy (delší) platí jedno pravidlo - takřka neexistuje program, který by nešel zkrátit. Nebo napsat nějak jinak, třeba lépe. Proto se vždy zamyslete, jestli to, co jste vymysleli, je opravdu to jediné možné nejlepší. Máte-li filipa, dříve či později přijdete na jiná, zajímavější řešení. Programátor zraje spojitě s časem (naplněným prací! - neplést si s hruškou na větvi).

*** Program by měl být nejen co nejkratší, ale hlavně co nejrychlejší (kromě časovaných rutin, kde je čas tvrdě určen). Ale ne vždy je kratší program rychlejší než jiné řešení, které má třeba o nějaký ten bajt navíc. Sledujte, kudy všudy musí pro vyplnění nějaké funkce program proběhnout, a zamyslete se, zda by tato cesta nemohla být řešena nějak jinak. Typický případ je zasazení spousty smyček do sebe, které vytvářejí "spirálovou mlhovinu", jež může průběh operace pěkně natáhnout. Někdy je lepší takovou motanici nahradit pár inteligentními testy, které převedou řízení programu do jednoduchých kaskád, jejichž cesta je pak v součtu podstatně kratší.

*** Co nejdříve se naučte používat logické funkce. V nich leží ohromné možnosti krácení programu i doby jeho průběhu. Maskování není jedinou možností jejich užití. Hodně programátorů zapomíná právě na tento typ instrukcí, protože pro ně nevidí žádné užití, tak co se jimi zabývat. Totéž platí o dvojkově komplementárních číslech. Je zajímavé, že špatný programátor se pozná i podle toho, že mu v programech chybějí logické operace, DK čísla a povětšinou i instrukce rotací a posuvů. Užití těchto instrukcí nemusí být zrovna přesně jen v tom, o čem se zmiňuje učebnice - ta hovoří jen o typickém užití. Tvůrčí proces má však licenci na netušené!

*** Pokud budete mít dojem, že vám to jde, "jak psovi pastva", nic si z toho nedělejte. Buď máte běžný útlum, nebo se dáváte zbytečně složitou cestou (struktura!). Výzkumy bylo zjištěno, že dobří programátoři laborující se strojovým kódem (assemblerem) mají produktivitu kolem 20 instrukcí denně (včetně ladění programu). Každý program zpočátku přibývá poměrně rychle. S přibývajícími bajty rychlost kynutí klesá. Někdy je dokonce nutno si říci - Tak tudy ne! - a začít jinak. Nebojte se přepisovat, škrtat, prostě experimentovat. Je to rozhodně lepší, než si pod sebou budovat močál a do něj se stále hlouběji propadat.

*** Jednou z dobrých forem práce je stavebnicové programování. Je to analogie stavebnicové korespondence, kde se za sebe řadí v tom momentu potřebné a vhodné standardizované odstavce s případnými malými úpravami nebo doplňky - tak se dopisy nemusejí vymýšlet stále znova. Určité rutiny, které se často používají, mějte někde archivované na záznamovém médiu. Jde především o rutiny testů klávesnice, práce s obrazovou pamětí (pohyby bodů

na obrazovce, barevné změny), rutiny zvukového výstupu, hledání určité proměnné (i řetězcové) atd., apod. V případě potřeby patřičnou rutinu jen zařaďte do programu - někdy ji ani není třeba upravovat. Chce to však si takový archív pořídit a vědět, co v něm je. Tento postup je jedním ze základů úspěchu softwarových firem. Měly-li by všechno vymýšlet stále dokola, jednoduše by zkrachovaly pod tlakem časových ztrát. Proto vřele doporučujeme. Rutiny najdete v literatuře, ve speciálních sestavách, které jsou v prodeji na páskách a discích, některé si samozřejmě vytvoříte sami.

*** I když vám bude hrdost bránit ve zjednodušení si práce použitím Basicu, nebraňte se tomu zcela. Příkazy Basicu klidně použijte tam, kde by bylo programování v assembleru zbytečnou a otravnou machou. Někdy vám Basic pomůže v "poukování" důležitých parametrů, jejichž ukládání by jinak bylo zbytečně náročné na čas i orientaci. Rovněž nemá cenu nahrazovat strojovým kódem příkazy pro ovládání záznamu a jeho čtení. Na druhou stranu vám ale nikdo nebrání se Basicu ani netknout.

*** Novicům strojového kódu lze však jen doporučit, aby Basic postupně doplňovali rutinami strojového kódu tak, že jimi budou nahrazovat části Basicového programu, až na Basic nakonec budou moci zapomenout zcela.

*** Jedním z "vyšších principů" programování je takový program, který modifikuje sám sebe. Dokud nezvládnete "běžné" programování, do těchto temných vod se příliš nepouštějte. Rychle byste se začali topit. Pokud jde "jen" o modifikaci parametrů coby dat, ještě se to dá zvládnout bez ztráty vědomí. Ale modifikace instrukcí, či jejich sestav, které od adresy A dělají jedno, ale od adresy A+1 něco zcela jiného (i když na adrese A je třeba tříbajtová instrukce) a ještě se v průběhu programu modifikují, to už je značně silný tabák. Takovýto program je prakticky nerozluštitelný (i pro autora, pokud nemá perfektní dokumentaci). Jeho výhoda je hlavně v šetření paměti. A je-li opravdu geniální, i ve zvýšení rychlosti. Protože takovéto programy ve své velké většině nejsou upravitelné pro změněné aplikační podmínky, nerady se vidí v případě systémových a užitkových programů. Rozhodně však nic proti nim.

*** Každý počítač má nějaký operační systém. Mikropočítače jej mají většinou v pamětech ROM. Sežeňte si jeho údaje a komentovaný výpis jeho softwaru. Jednak v něm najdete řadu podnětů, zjistíte, jak co jde dělat a budete moci přímo využívat jeho rutin ve svých programech (nebudete je tak muset vymýšlet a ušetříte čas i paměť svou i počítače).

*** Nakonec jen jedno - své vědomosti stále doplňujte čtením odborné literatury, sledováním vývoje a převáděním toho všeho do své praxe. Mějte o svém počítači všechnu možnou dostupnou dokumentaci, shánějte všechny možné programové finesy a zajímavá programová řešení. Pokud nevíte nic moc o hardwaru, nezbyde vám, než se jím začít zabývat. Jinak nebudete moci využít nic ze široké palety interfacingu - a to by byla ohromná škoda. Interfacing je možný už s tím, co má počítač v sobě - ovládání portů tvorby zvuku, formátu dat pro sériový přenos, hrátky s obrazem, tiskárnou atd. - podle typu počítače. Tvorba a poznání tvoří nekonečnou smyčku nejen v oblasti výpočetní techniky.

S shakespearovským

Býti bity bit či bity býti nebit?

Vám mnoho zdaru ve vaší programátorské dřině

přeje

AUTOR

****

P ř í l o h a

Výpočet doby provedení programu
a
diagram registrů Z80

1 list

Schématická znázornění průběhu vybraných instrukcí
(s uvedením parametrů
před a po provedení instrukce)

4 listy

Blokový diagram Z80
a
vývody pouzdra integrovaného obvodu Z80

1 list

Soubor instrukcí Z80
(assemblerový tvar, hexadecimální ekvivalent a počet cyklů T)

3 listy

Vliv instrukcí Z80 na stavové indikátory

1 list

Literatura a software

1 list

****

****

Výpočet doby provedení programu

****

Každá instrukce se provádí určitý počet cyklů (kmitů) vnějších hodin. Tyto cykly si označíme T. Jsou odvozeny od kmitočtu začleněného generátoru kmitů (prakticky vždy krystalového). Další časovou veličinou, k níž se váže prováděcí doba, je strojový cyklus M. Během každého cyklu M mikroprocesor provede operaci s 1 bajtem. Je pravidlem, že **1.bajt instrukce se provede vždy za dobu 4 T, další bajty pak za dobu 3 T.** Je-li tedy instrukce jednobajtová, provede se za dobu 4 T, neboli 1 M. Je-li tříbajtová, provede se za 4+3+3=10 T, resp. 3 M. Odlišnosti jsou u instrukcí, jejichž exekuce je vázána na provedení dalších interních operací (např. u instrukcí s podmínkami). Dobu provedení ovlivňují i stavy WAIT, během nichž Z80 čeká potřebný nebo stanovený počet cyklů T. V praxi pak další odlišnosti mohou vyvstat i vlivem systému počítače (např. u Amstradu se provede každý bajt za dobu 4T).

Budeme zvažovat počítač s hodinovým kmitočtem 2,5 MHz. Tzn., že 1 cyklus T má hodnotu 1:2,5*1000000=0,0000004 sec = 400 nsec. V příloze uvedené instrukce mají přiřazen počet cyklů T. Znáte-li kmitočet hodin vašeho počítače, snadno si provedete výpočet doby provedení jakékoli části programu. Např.:

	počet cyklů T		počet provedení	doba provedení	
LD A,35	7		1	2,8	(mikrosec)
LD B,49	7		1	2,8	
OR B	4		1	1,6	
AND 99	7		1	2,8	
RL A	4		1	1,6	
			celkem	11,6	
CALL XXXX	17		1	6,8	
RET	10		1	4,0	
			celkem	10,8	
LD A,06	7		1	2,8	
LD B,08	7		1	2,8	
LOOP:INC A	4		9	14,4	
DEC B	4		9	14,4	
JR NZ,LOOP	12	(Z=0)	8	22,4	
	7	(Z=1)	1	4,8	
			celkem	61,6	
LD HL,0100H	10		1	4,0	
LD DE,0200H	10		1	4,0	
LD BC,0010H	10		1	4,0	
LDIR	21		15	126,0	
	16	(BC=0)	1	6,4	
			celkem	144,4	

REGISTRY MIKROPROCESORU Z80

HLAVNÍ BANKA 0

A	F
B	C
D	E
H	L
I X	
I Y	
S P	
P C	
I	R

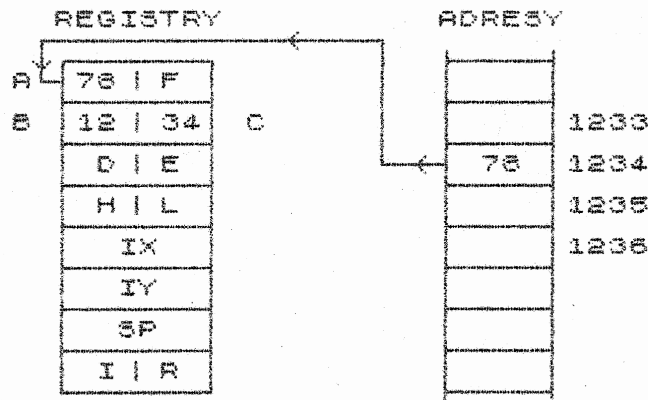
VEDLEJŠÍ BANKA 1

A´	F´
B´	C´
D´	E´
H´	L´

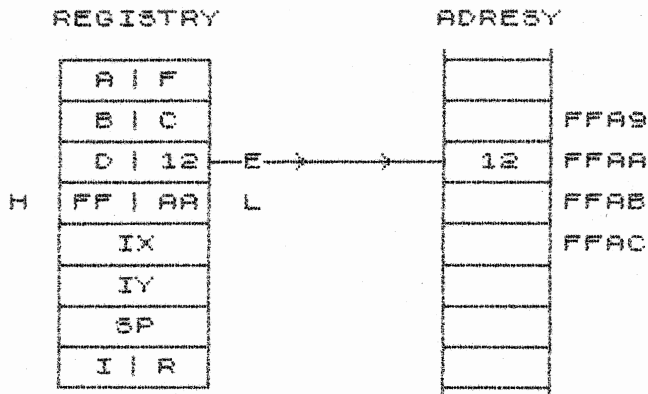
KLOPNÉ OBVODY PŘERUŠENÍ

IFF1	IFF2

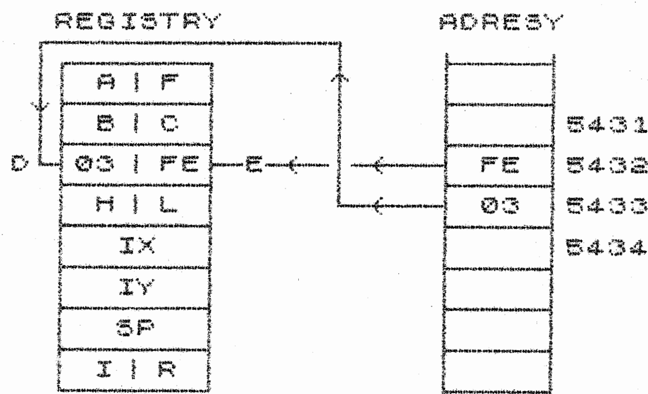
INSTRUKCE LD A,(BC) BC=1234
(BC)=78



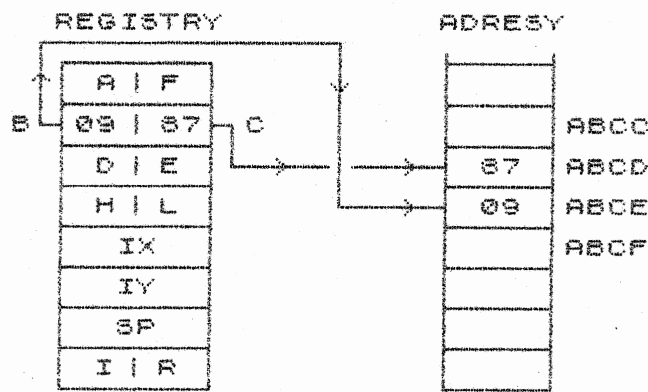
INSTRUKCE LD (HL),E HL=FFAA
E=12



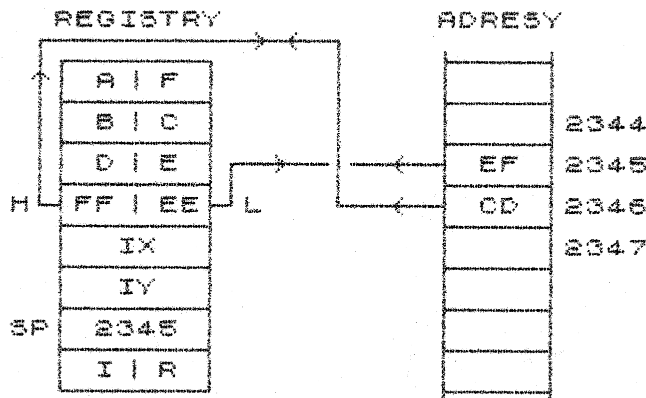
INSTRUKCE LD DE,(ADR) ADR=5432
(ADR)=FE (ADR+1)=03



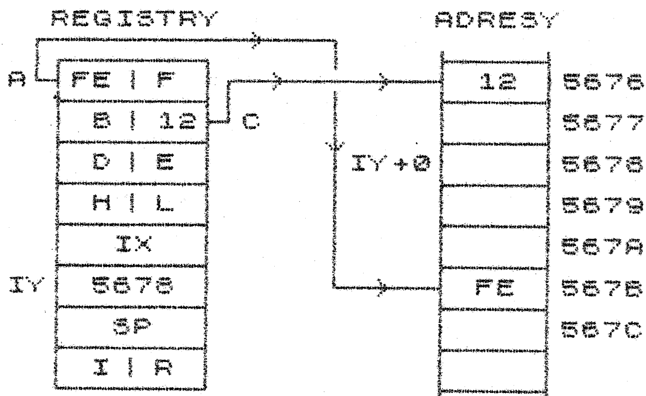
INSTRUKCE LD (ADR),BC ADR=ABCD
BC=0987



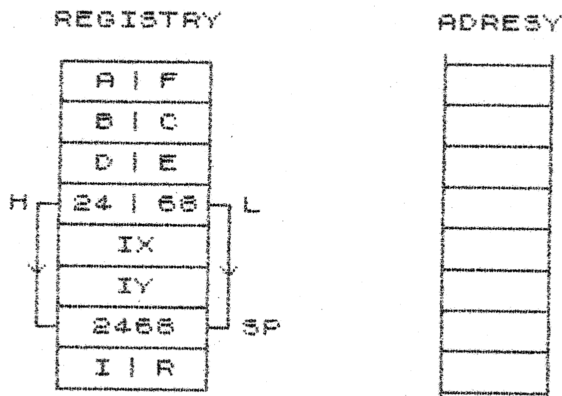
INSTRUKCE EX (SP),HL SP=2345
(SP)=FF (SP+1)=EE HL=CDEF



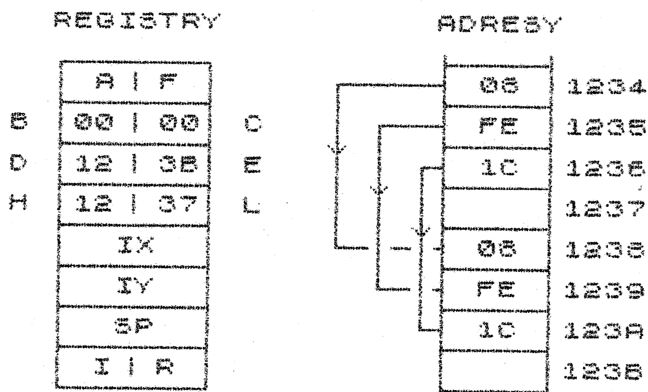
INSTRUKCE LD (IY+03),A IY=5678
LD (IY+FE),C A=FE C=12



INSTRUKCE LD SP,HL HL=2468



INSTRUKCE LDIR HL=1234 DE=1238
BC=0003



INSTRUKCE CPIR HL=0100 BC=0008 A=FA

REGISTRY

A	FA	F	
B	00	04	C
	D	E	
H	01	04	L
	IX		
	IY		
	SP		
	I	R	

ADRESY

	0100
	0101
	0102
FA	0103
	0104
	0105
	0106
	0107

INSTRUKCE CPDR HL=0107 BC=0008 A=FA

REGISTRY

A	FA	F	
B	00	03	C
	D	E	
H	01	02	L
	IX		
	IY		
	SP		
	I	R	

ADRESY

	0100
	0101
	0102
FA	0103
	0104
	0105
	0106
	0107

INSTRUKCE JP(HL) JE NA ADR 0105 HL=0102

REGISTRY

H	01	02	L

PŘED

PC	0105

PO

PC	0102

ADRESY

		0100
		0101
SKOK →		0102
		0103
		0104
	E9	0105
		0106
		0107

INSTRUKCE JR FA JE NA ADR 0105

REGISTRY

PŘED

PC	0105

PO

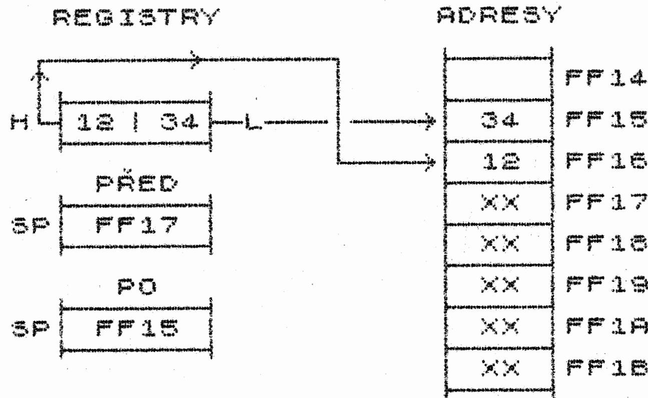
PC	0101

ADRESY

		0100
SKOK →		0101
FB		0102
FC		0103
FD		0104
FE	18	0105
FF	FA	0106
00		0107

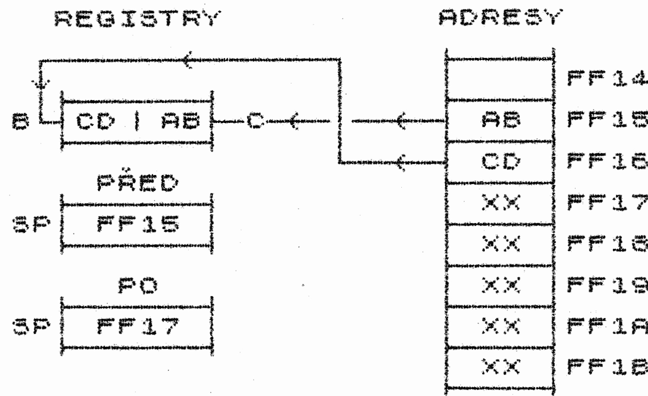
INSTRUKCE PUSH HL

HL=1234
SP=FF17



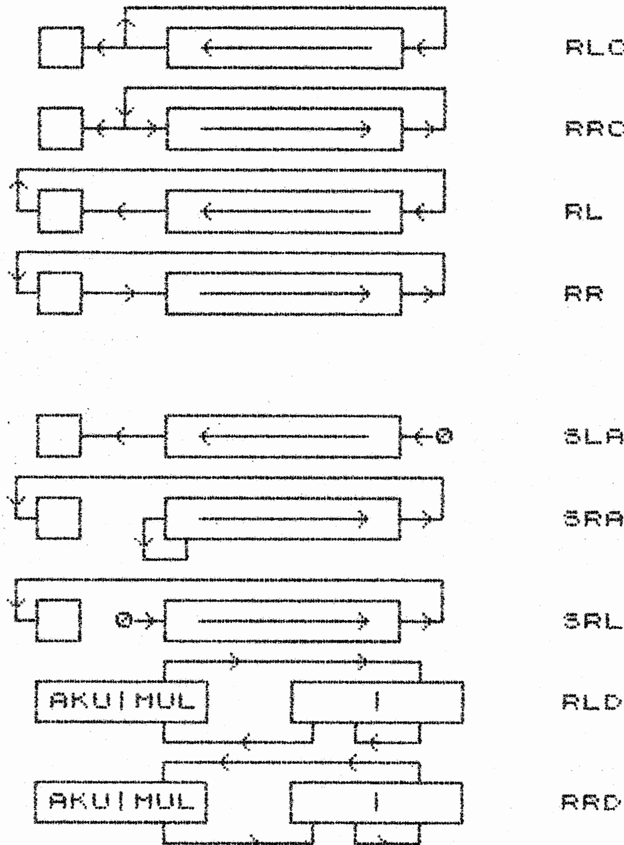
INSTRUKCE POP BC

SP=FF15



INSTRUKCE ROTACE A POSUVU

CARRY BAJT



```
                        Z80
                   ┌──────────────────┐
           ←───────┤ M1         A0    ├───→
ŘÍZENÍ     ←───────┤ MREQ       A1    ├───→
           ←───────┤ IORQ       .     │   .
SYSTÉMU    ←───────┤ RD         .     │   .ADRESOVÁ
           ←───────┤ WR         .     │   .
           ←───────┤ RFSH       .     │   .SBĚRNICE
                   │            .     │   .
           ←───────┤ HALT       .     │   .
ŘÍZENÍ     ───────→┤ WAIT       .     │   .
           ───────→┤ INT        .     │   .
CPU Z80    ───────→┤ NMI        A15   ├───→
           ───────→┤ RESET            │
ŘÍZENÍ             │            D0    ├←──→
           ───────→┤ BUSREQ     D1    ├←──→
SBĚRNIC    ←───────┤ BUSACK     .     │   .
                   │            .     │   .DATOVÁ
HODINY     ←───────┤ CLK        .     │   .
           ←───────┤ +5V        .     │   .SBĚRNICE
A ZDROJ    ←───────┤ GND        D7    ├←──→
                   └──────────────────┘
```

```
            ┌──────────────────┐
A11   ─┤ 1               40 ├─A10
A12   ─┤ 2               39 ├─A9
A13   ─┤ 3  POUZ-        38 ├─A8
A14   ─┤ 4   DRO         37 ├─A7
A15   ─┤ 5               36 ├─A6
CLK   ─┤ 6   CPU         35 ├─A5
D4    ─┤ 7   Z80         34 ├─A4
D3    ─┤ 8               33 ├─A3
D5    ─┤ 9               32 ├─A2
D6    ─┤ 10              31 ├─A1
+5V   ─┤ 11              30 ├─A0
D2    ─┤ 12              29 ├─GND
D7    ─┤ 13              28 ├─RFSH
D0    ─┤ 14              27 ├─M1
D1    ─┤ 15              26 ├─RESET
INT   ─┤ 16              25 ├─BUSREQ
NMI   ─┤ 17              24 ├─WAIT
HALT──┤ 18               23 ├─BUSACK
MREQ──┤ 19               22 ├─WR
IORQ──┤ 20               21 ├─RD
            └──────────────────┘
```

```
BLOK.DIAGRAM Z80                  ↑ 8—BUS
                                  ↓
+5V →                         ┌─────┐
GND →                         │ DBI │
CLK →                         └─────┘
                                  ↕
              ┌─────┐    ┌───┐         ┌─────┐
              │ I D │←───│ R │←── I ──→│ ALU │
              └─────┘    └───┘         └─────┘
                 ↓                ↕
          T ← ┌─────┐         ┌─────┐
              │ T C │         │ R A │
              └─────┘         └─────┘
              ↓     ↑             ↕
              8O    5I        ┌─────┐
                              │ ALB │
                              └─────┘
                                  ↓
                               16—BUS
```

SYMBOLS

8—BIT:8—BIT DATA BUS
16—BIT:16—BIT ADDRESS BUS
T:CPU TIMING
8O:8 SYSTEMS AND CPU CONTROL
OUTPUTS
5I:5 CPU CONTROL INPUTS
DBI:DATA BUS INTERFACE
IDB:INTERNAL DATA BUS
ALU:ARITHM.AND LOG.UNIT
R:INSTRUCTION REGISTER
ID:INSTRUCTION DECODER
TC:CPU TIMING CONTROL
RA:REGISTER ARRAY
ALB:ADDRESS LOGIC AND BUFFERS
I:INTERNAL DATA BUS

L I T E R A T U R A

Barrow David: Assembler Routines for the Z80
Century Communications, London, GB, 1985

120 univerzálních assemblerových rutin pro programování mikroprocesoru Z80, podrobný komentář, 192 stran.

Wadsworth Nat: Z80 Instruction Handbook
Hayden Book Company, New Jersey, USA, 1978

Ucelený a systematický přehled funkcí všech instrukcí Z80 se stručnými komentáři. 118 stran.

Nichols J.C.a kol.: Nanocomputer
SGS-ATES, USA, 1978

Dvě doprovodné učebnice k mikropočítačové stavebnici. První se zabývá programováním ve strojovém kódu a assembleru Z80 /290 stran/, druhá výukou interfacingu /500 stran/. Velmi podrobné.

Zilog Components Data Book 1983/84, USA

Firemní pulbikace s popisy mikroprocesorů Z80, Z800, Z8000 a příslušejících PIO, SIO, CTC, DMA aj. obvodů z výrobního programu firmy Zilog. Obsahuje soubory instrukcí, diagramy časových průběhů a řízení všech integrovaných obvodů. 850 stran.

Bishop Graham: Spectrum Interfacing and Projects
McGraw-Hill Book Company, GB, 1983

Návody ke stavbě hardwarových doplňků k ZX Spectru, 136 stran.

Logan Ian: Understanding Your Spectrum
Melbourne House Publishers, GB, 1982

Stručný popis mikropočítače ZX Spectrum s ukázkami využití assembleru pro ovládání jeho hardwaru. 190 stran.

Logan I.,O'Hara F.: The Complete Spectrum ROM Disassembly
Melbourne House Publishers, GB, 1983

Komentovaný výpis obsahu paměti ROM mikropočítače ZX Spectrum. Velmi vhodný doplněk ke studiu assembleru Z80. 230 stran.

Godden Bruce: CPC 464 Firmware
Amsoft, GB, 1984

Zevrubný, komentovaný soubor softwarového ovládání hardwaru mikropočítače Amstrad CPC 464, cca 450 stran.

Kolektiv autorů: Výpočetní a řídící technika
Oborová encyklopedie SNTL, 1982

Encyklopedie s abecedně řazenými hesly, 372 stran.

Dědina B, Valášek P.: Mikroprocesory a mikropočítače
Knižnice výpočetní techniky, SNTL, 1983

Stručný přehled základních pojmů, instrukcí a mikroprocesorů 8080, MC6800, Z80, 8085, 8086, jejich odvozenin a hardwarových doplňků. 304 stran.

Kolektiv autorů: Anglicko-český a česko-anglický elektrotechnický a elektronický slovník, SNTL, 1982

Zahrnuje řadu termínů z výpočetní techniky, a to i v jejich nejčastěji užívaných anglických zkratkách, 924 stran.

S O F T W A R E (pro ZX Spectrum)

Machine Code Tutor

Velmi názorný program pro výuku programování mikroprocesoru Z80. Možno vytvářet i vlastní krátké rutiny - chyby jsou ošetřeny, proto nemají kolapsové následky.

Supercode 3

122 krátkých pomocných rutin ve strojovém kódu, které je možno využít ve vlastních programech.

GENS3, MONS3, GENS3E, MON2

Generátor a monitor strojového kódu. GENS3E je vylepšenou verzí GENS3 /má full screen editor a 42 znaků na řádku/. MON2 je prvním spectrovským monitorem s funkcí tracing.

****

I N S T R U K C E

CPU ----------------- Z80

****

Instrukce	Kód	Takty
ADC A,(HL)	8E	7
ADC A,(IX+dis)	DD 8E XX	19
ADC A,(IY+dis)	FD 8E XX	19
ADC A,A	8F	4
ADC A,B	88	4
ADC A,C	89	4
ADC A,D	8A	4
ADC A,E	8B	4
ADC A,H	8C	4
ADC A,L	8D	4
ADC A,NN	CE NN	7
ADC HL,BC	ED 4A	15
ADC HL,DE	ED 5A	15
ADC HL,HL	ED 6A	15
ADC HL,SP	ED 7A	15
ADD A,(HL)	86	7
ADD A,(IX+dis)	DD 86 XX	19
ADD A,(IY+dis)	FD 86 XX	19
ADD A,A	87	4
ADD A,B	80	4
ADD A,C	81	4
ADD A,D	82	4
ADD A,E	83	4
ADD A,H	84	4
ADD A,L	85	4
ADD A,NN	C6 NN	7
ADD HL,BC	09	11
ADD HL,DE	19	11
ADD HL,HL	29	11
ADD HL,SP	39	11
ADD IX,BC	DD 09	15
ADD IX,DE	DD 19	15
ADD IX,IX	DD 29	15
ADD IX,SP	DD 39	15
ADD IY,BC	FD 09	15
ADD IY,DE	FD 19	15
ADD IY,IY	FD 29	15
ADD IY,SP	FD 39	15
AND (HL)	A6	7
AND (IX+dis)	DD A6 XX	19
AND (IY+dis)	FD A6 XX	19
AND A	A7	4
AND B	A0	4
AND C	A1	4
AND D	A2	4
AND E	A3	4
AND H	A4	4
AND L	A5	4
AND NN	E6 NN	7
BIT 0,(HL)	CB 46	12
BIT 0,(IX+dis)	DD CB XX 46	20
BIT 0,(IY+dis)	FD CB XX 46	20
BIT 0,A	CB 47	8
BIT 0,B	CB 40	8
BIT 0,C	CB 41	8
BIT 0,D	CB 42	8
BIT 0,E	CB 43	8
BIT 0,H	CB 44	8
BIT 0,L	CB 45	8
BIT 1,(HL)	CB 4E	12
BIT 1,(IX+dis)	DD CB XX 4E	20
BIT 1,(IY+dis)	FD CB XX 4E	20
BIT 1,A	CB 4F	8
BIT 1,B	CB 48	8
BIT 1,C	CB 49	8
BIT 1,D	CB 4A	8
BIT 1,E	CB 4B	8
BIT 1,H	CB 4C	8
BIT 1,L	CB 4D	8
BIT 2,(HL)	CB 56	12
BIT 2,(IX+dis)	DD CB XX 56	20
BIT 2,(IY+dis)	FD CB XX 56	20
BIT 2,A	CB 57	8
BIT 2,B	CB 50	8
BIT 2,C	CB 51	8
BIT 2,D	CB 52	8
BIT 2,E	CB 53	8
BIT 2,H	CB 54	8
BIT 2,L	CB 55	8
BIT 3,(HL)	CB 5E	12
BIT 3,(IX+dis)	DD CB XX 5E	20
BIT 3,(IY+dis)	FD CB XX 5E	20
BIT 3,A	CB 5F	8
BIT 3,B	CB 58	8
BIT 3,C	CB 59	8
BIT 3,D	CB 5A	8
BIT 3,E	CB 5B	8
BIT 3,H	CB 5C	8
BIT 3,L	CB 5D	8
BIT 4,(HL)	CB 66	12
BIT 4,(IX+dis)	DD CB XX 66	20
BIT 4,(IY+dis)	FD CB XX 66	20
BIT 4,A	CB 67	8
BIT 4,B	CB 60	8
BIT 4,C	CB 61	8
BIT 4,D	CB 62	8
BIT 4,E	CB 63	8
BIT 4,H	CB 64	8
BIT 4,L	CB 65	8
BIT 5,(HL)	CB 6E	12
BIT 5,(IX+dis)	DD CB XX 6E	20
BIT 5,(IY+dis)	FD CB XX 6E	20
BIT 5,A	CB 6F	8
BIT 5,B	CB 68	8
BIT 5,C	CB 69	8
BIT 5,D	CB 6A	8
BIT 5,E	CB 6B	8
BIT 5,H	CB 6C	8
BIT 5,L	CB 6D	8
BIT 6,(HL)	CB 76	12
BIT 6,(IX+dis)	DD CB XX 76	20
BIT 6,(IY+dis)	FD CB XX 76	20
BIT 6,A	CB 77	8
BIT 6,B	CB 70	8
BIT 6,C	CB 71	8
BIT 6,D	CB 72	8
BIT 6,E	CB 73	8
BIT 6,H	CB 74	8
BIT 6,L	CB 75	8
BIT 7,(HL)	CB 7E	12
BIT 7,(IX+dis)	DD CB XX 7E	20
BIT 7,(IY+dis)	FD CB XX 7E	20
BIT 7,A	CB 7F	8
BIT 7,B	CB 78	8
BIT 7,C	CB 79	8
BIT 7,D	CB 7A	8
BIT 7,E	CB 7B	8
BIT 7,H	CB 7C	8
BIT 7,L	CB 7D	8
CALL ADDR	CD XX XX	17
CALL Z,ADDR	CC XX XX	17/10
CALL NZ,ADDR	C4 XX XX	17/10
CALL C,ADDR	DC XX XX	17/10
CALL NC,ADDR	D4 XX XX	17/10
CALL PE,ADDR	EC XX XX	17/10
CALL PO,ADDR	E4 XX XX	17/10
CALL M,ADDR	FC XX XX	17/10
CALL P,ADDR	F4 XX XX	17/10
CCF	3F	4
CP (HL)	BE	7
CP (IX+dis)	DD BE XX	19
CP (IY+dis)	FD BE XX	19
CP A	BF	4
CP B	B8	4
CP C	B9	4
CP D	BA	4
CP E	BB	4
CP H	BC	4
CP L	BD	4
CP NN	FE NN	7
CPD	ED A9	16
CPDR	ED B9	21/16
CPI	ED A1	16
CPIR	ED B1	21/16
CPL	2F	4
DAA	27	4
DEC (HL)	35	11
DEC (IX+dis)	DD 35 XX	23
DEC (IY+dis)	FD 35 XX	23
DEC A	3D	4
DEC B	05	4
DEC C	0D	4
DEC D	15	4
DEC E	1D	4
DEC H	25	4
DEC L	2D	4
DEC BC	0B	6
DEC DE	1B	6
DEC HL	2B	6
DEC IX	DD 2B	10
DEC IY	FD 2B	10
DEC SP	3B	6
DI	F3	4
DJNZ,dis	10 XX	13/8
EI	FB	4
EX (SP),HL	E3	19
EX (SP),IX	DD E3	23
EX (SP),IY	FD E3	23
EX AF,AF'	08	4
EX DE,HL	EB	4
EXX	D9	4
HALT	76	4
IM 0	ED 46	8
IM 1	ED 56	8
IM 2	ED 5E	8
IN A,port	DB NN	11
IN A,(C)	ED 78	12
IN B,(C)	ED 40	12
IN C,(C)	ED 48	12
IN D,(C)	ED 50	12
IN E,(C)	ED 58	12
IN H,(C)	ED 60	12
IN L,(C)	ED 68	12
INC (HL)	34	11
INC (IX+dis)	DD 34 XX	23
INC (IY+dis)	FD 34 XX	23
INC A	3C	4
INC B	04	4
INC C	0C	4
INC D	14	4
INC E	1C	4
INC H	24	4
INC L	2C	4
INC BC	03	6
INC DE	13	6
INC HL	23	6
INC IX	DD 23	10
INC IY	FD 23	10
INC SP	33	6
IND	ED AA	16
INI	ED A2	16
INDR	ED BA	21/16
INIR	ED B2	21/16
JP (HL)	E9	4
JP (IX)	DD E9	8
JP (IY)	FD E9	8
JP ADDR	C3 XX XX	10
JP Z,ADDR	CA XX XX	10
JP NZ,ADDR	C2 XX XX	10
JP C,ADDR	DA XX XX	10
JP NC,ADDR	D2 XX XX	10
JP PE,ADDR	EA XX XX	10
JP PO,ADDR	E2 XX XX	10
JP M,ADDR	FA XX XX	10
JP P,ADDR	F2 XX XX	10
JR dis	18 XX	12
JR Z,dis	28 XX	12/7

JR NZ,dis	20 XX	12/7
JR C,dis	38 XX	12/7
JR NC,dis	30 XX	12/7
LD (ADDR),A	32 XX XX	13
LD (ADDR),BC	ED 43 XX XX	20
LD (ADDR),DE	ED 53 XX XX	20
LD (ADDR),HL	ED 63 XX XX	20
LD (ADDR),HL	22 XX XX	16
LD (ADDR),IX	DD 22 XX XX	20
LD (ADDR),IY	FD 22 XX XX	20
LD (ADDR),SP	ED 73 XX XX	20
LD (BC),A	02	7
LD (DE),A	12	7
LD (HL),A	77	7
LD (HL),B	70	7
LD (HL),C	71	7
LD (HL),D	72	7
LD (HL),E	73	7
LD (HL),H	74	7
LD (HL),L	75	7
LD (HL),NN	36 NN	10
LD (IX+dis),A	DD 77 XX	19
LD (IX+dis),B	DD 70 XX	19
LD (IX+dis),C	DD 71 XX	19
LD (IX+dis),D	DD 72 XX	19
LD (IX+dis),E	DD 73 XX	19
LD (IX+dis),H	DD 74 XX	19
LD (IX+dis),L	DD 75 XX	19
LD (IX+dis),NN	DD 36 XX NN	19
LD (IY+dis),A	FD 77 XX	19
LD (IY+dis),B	FD 70 XX	19
LD (IY+dis),C	FD 71 XX	19
LD (IY+dis),D	FD 72 XX	19
LD (IY+dis),E	FD 73 XX	19
LD (IY+dis),H	FD 74 XX	19
LD (IY+dis),L	FD 75 XX	19
LD (IY+dis),NN	FD 36 XX NN	19
LD A,(ADDR)	3A XX XX	13
LD A,(BC)	0A	7
LD A,(DE)	1A	7
LD A,(HL)	7E	7
LD A,(IX+dis)	DD 7E XX	19
LD A,(IY+dis)	FD 7E XX	19
LD A,A	7F	4
LD A,B	78	4
LD A,C	79	4
LD A,D	7A	4
LD A,E	7B	4
LD A,H	7C	4
LD A,L	7D	4
LD A,NN	3E NN	7
LD A,I	ED 57	9
LD A,R	ED 5F	9
LD B,(HL)	46	7
LD B,(IX+dis)	DD 46 XX	19
LD B,(IY+dis)	FD 46 XX	19
LD B,A	47	4
LD B,B	40	4
LD B,C	41	4
LD B,D	42	4
LD B,E	43	4
LD B,H	44	4
LD B,L	45	4
LD B,NN	06 NN	7
LD BC,(ADDR)	ED 4B XX XX	20
LD BC,NNNN	01 NN NN	10
LD C,(HL)	4E	7
LD C,(IX+dis)	DD 4E XX	19
LD C,(IY+dis)	FD 4E XX	19
LD C,A	4F	4
LD C,B	48	4
LD C,C	49	4
LD C,D	4A	4
LD C,E	4B	4
LD C,H	4C	4
LD C,L	4D	4
LD C,NN	0E NN	7
LD D,(HL)	56	7
LD D,(IX+dis)	DD 56 XX	19
LD D,(IY+dis)	FD 56 XX	19
LD D,A	57	4
LD D,B	50	4
LD D,C	51	4
LD D,D	52	4
LD D,E	53	4
LD D,H	54	4
LD D,L	55	4
LD D,NN	16 NN	7
LD DE,(ADDR)	ED 5B XX XX	20
LD DE,NNNN	11 NN NN	10
LD E,(HL)	5E	7
LD E,(IX+dis)	DD 5E XX	19
LD E,(IY+dis)	FD 5E XX	19
LD E,A	5F	4
LD E,B	58	4
LD E,C	59	4
LD E,D	5A	4
LD E,E	5B	4
LD E,H	5C	4
LD E,L	5D	4
LD E,NN	1E NN	7
LD H,(HL)	66	7
LD H,(IX+dis)	DD 66 XX	19
LD H,(IY+dis)	FD 66 XX	19
LD H,A	67	4
LD H,B	60	4
LD H,C	61	4
LD H,D	62	4
LD H,E	63	4
LD H,H	64	4
LD H,L	65	4
LD H,NN	26 NN	7
LD HL,(ADDR)	ED 6B XX XX	20
LD HL,(ADDR)	2A XX XX	16
LD HL,NNNN	21 NN NN	10
LD L,(HL)	6E	7
LD L,(IX+dis)	DD 6E XX	19
LD L,(IY+dis)	FD 6E XX	19
LD L,A	6F	4
LD L,B	68	4
LD L,C	69	4
LD L,D	6A	4
LD L,E	6B	4
LD L,H	6C	4
LD L,L	6D	4
LD L,NN	2E NN	7
LD IX,(ADDR)	DD 2A XX XX	20
LD IX,NNNN	DD 21 NN NN	14
LD IY,(ADDR)	FD 2A XX XX	20
LD IY,NNNN	FD 21 NN NN	14
LD I,A	ED 47	9
LD R,A	ED 4F	9
LD SP,(ADDR)	ED 7B XX XX	20
LD SP,HL	F9	6
LD SP,IX	DD F9	10
LD SP,IY	FD F9	10
LD SP,NNNN	31 NN NN	10
LDD	ED A8	16
LDI	ED A0	16
LDDR	ED B8	21/16
LDIR	ED B0	21/16
NEG	ED 44	8
NOP	00	4
OR (HL)	B6	7
OR (IX+dis)	DD B6 XX	19
OR (IY+dis)	FD B6 XX	19
OR A	B7	4
OR B	B0	4
OR C	B1	4
OR D	B2	4
OR E	B3	4
OR H	B4	4
OR L	B5	4
OR NN	F6 NN	7
OTDR	ED BB	21/16
OTIR	ED B3	21/16
OUT port,A	D3 XX	11
OUT (C),A	ED 79	12
OUT (C),B	ED 41	12
OUT (C),C	ED 49	12
OUT (C),D	ED 51	12
OUT (C),E	ED 59	12
OUT (C),H	ED 61	12
OUT (C),L	ED 69	12
OUTD	ED AB	16
OUTI	ED A3	16
POP AF	F1	10
POP BC	C1	10
POP DE	D1	10
POP HL	E1	10
POP IX	DD E1	14
POP IY	FD E1	14
PUSH AF	F5	11
PUSH BC	C5	11
PUSH DE	D5	11
PUSH HL	E5	11
PUSH IX	DD E5	15
PUSH IY	FD E5	15
RES 0,(HL)	CB 86	15
RES 0,(IX+dis)	DD CB XX 86	23
RES 0,(IY+dis)	FD CB XX 86	23
RES 0,A	CB 87	8
RES 0,B	CB 80	8
RES 0,C	CB 81	8
RES 0,D	CB 82	8
RES 0,E	CB 83	8
RES 0,H	CB 84	8
RES 0,L	CB 85	8
RES 1,(HL)	CB 8E	15
RES 1,(IX+dis)	DD CB XX 8E	23
RES 1,(IY+dis)	FD CB XX 8E	23
RES 1,A	CB 8F	8
RES 1,B	CB 88	8
RES 1,C	CB 89	8
RES 1,D	CB 8A	8
RES 1,E	CB 8B	8
RES 1,H	CB 8C	8
RES 1,L	CB 8D	8
RES 2,(HL)	CB 96	15
RES 2,(IX+dis)	DD CB XX 96	23
RES 2,(IY+dis)	FD CB XX 96	23
RES 2,A	CB 97	8
RES 2,B	CB 90	8
RES 2,C	CB 91	8
RES 2,D	CB 92	8
RES 2,E	CB 93	8
RES 2,H	CB 94	8
RES 2,L	CB 95	8
RES 3,(HL)	CB 9E	15
RES 3,(IX+dis)	DD CB XX 9E	23
RES 3,(IY+dis)	FD CB XX 9E	23
RES 3,A	CB 9F	8
RES 3,B	CB 98	8
RES 3,C	CB 99	8
RES 3,D	CB 9A	8
RES 3,E	CB 9B	8
RES 3,H	CB 9C	8
RES 3,L	CB 9D	8
RES 4,(HL)	CB A6	15
RES 4,(IX+dis)	DD CB XX A6	23
RES 4,(IY+dis)	FD CB XX A6	23
RES 4,A	CB A7	8
RES 4,B	CB A0	8
RES 4,C	CB A1	8
RES 4,D	CB A2	8
RES 4,E	CB A3	8
RES 4,H	CB A4	8
RES 4,L	CB A5	8
RES 5,(HL)	CB AE	15
RES 5,(IX+dis)	DD CB XX AE	23
RES 5,(IY+dis)	FD CB XX AE	23
RES 5,A	CB AF	8
RES 5,B	CB A8	8
RES 5,C	CB A9	8

****

Vliv instrukcí Z80 na stavové indikátory

****

Instrukce	CY	Z	P/V	S	N	H	Komentář
ADC HL,XX	*	*	V	*	0	?	
ADC X;ADD X	*	*	V	*	0	*	
ADD XX,XX	*	.	.	.	0	?	
AND X	0	*	P	*	0	*	
BIT b,X	.	*	?	?	0	1	Z=log.stav bitu b
CALL instrukce			bez vlivu				
CCF	*	.	.	.	0	?	
CP X	*	*	V	*	1	*	
CPD;CPI;CPDR;CPIR	.	*	V	*	1	*	P/V=0, když BC=0 Z=1, když A=(HL)
CPL	.	.	.	.	1	1	
DAA	*	*	P	*	.	*	
DEC X	.	*	V	*	1	*	8mibitová operace!
DEC XX			bez vlivu				
DI;EI			bez vlivu				
DJNZ			bez vlivu				
EX;EXX instrukce			bez vlivu				
HALT			bez vlivu				
IM 0;IM 1;IM 2			bez vlivu				
INC X	.	*	V	*	0	*	8mibitová operace!
INC XX			bez vlivu				
IN A,port			bez vlivu				
IN r,(C)	.	*	P	*	0	*	
IND;INI	.	*	?	?	1	?	Z=1, když B=0
INDR;INIR	.	1	?	?	1	?	Z=1, když B=0
JP;JR instrukce			bez vlivu				
LD A,I;LD A,R	.	*	IFF	*	0	0	P/V zobrazí stav IFF
LDD;LDI	.	.	*	.	0	0	
LDDR;LDIR	.	.	0	.	0	0	P/V=0, když BC=0
ostatní LD instrukce			bez vlivu				
NEG	*	*	V	*	1	*	
NOP			bez vlivu				
OR X	0	*	P	*	0	*	
OTDR;OTIR	.	1	?	?	1	?	Z=1, když B=0
OUTD;OUTI	.	*	?	?	1	?	Z=1, když B=0
OUT (C),r;OUT port,(A)			bez vlivu				
RET instrukce			bez vlivu				
POP XX; PUSH XX			bez vlivu				
RES b,X			bez vlivu				
RLA;RLCA;RRA;RRCA	*	.	.	.	0	0	
RLD;RRD	.	*	P	*	0	0	
RL X;RLC X;RR X;RRC X SLA X;SRA X;SRL X	*	*	P	*	0	0	
SBC HL,XX	*	*	V	*	1	?	
SCF	1	.	.	.	0	0	
SET b,X			bez vlivu				
SBC X;SUB X	*	*	V	*	1	*	
XOR X	0	*	P	*	0	*	

Vysvětlivky

0 a 1 - indikátor je ve stavu log.0 nebo log.1
* - stav indikátoru je závislý na výsledku operace
? - stav indikátoru je nepředpokládatelný
P - ovlivnění indikátoru parity
V - ovlivnění indikátoru přetečení
. - stav indikátoru se nemění
X - 8 bitů (ať registru nebo adresy paměti)
XX - 16 bitů (16tibitového či párového registru nebo dvou adres)
r - 8mibitový registr
b - pořadové číslo bitu v bajtu X
IFF - Interrupt enable flip-flop (klopný obvod přerušení)

Pozn.: Pro instrukce CPI, CPD, LDI a LDD (včetně opakování) dále platí:

- podmínka PE je splněna, když obsah BC je různý od 0
- podmínka PO je splněna tehdy, když BC=0

RES 5,D	CB AA	8
RES 5,E	CB AB	8
RES 5,H	CB AC	8
RES 5,L	CB AD	8
RES 6,(HL)	CB B6	15
RES 6,(IX+dis)	DD CB XX B6	23
RES 6,(IY+dis)	FD CB XX B6	23
RES 6,A	CB B7	8
RES 6,B	CB B0	8
RES 6,C	CB B1	8
RES 6,D	CB B2	8
RES 6,E	CB B3	8
RES 6,H	CB B4	8
RES 6,L	CB B5	8
RES 7,(HL)	CB BE	15
RES 7,(IX+dis)	DD CB XX BE	23
RES 7,(IY+dis)	FD CB XX BE	23
RES 7,A	CB BF	8
RES 7,B	CB B8	8
RES 7,C	CB B9	8
RES 7,D	CB BA	8
RES 7,E	CB BB	8
RES 7,H	CB BC	8
RES 7,L	CB BD	8
RET	C9	10
RET Z	C8	11/5
RET NZ	C0	11/5
RET C	D8	11/5
RET NC	D0	11/5
RET PE	E8	11/5
RET PO	E0	11/5
RET M	F8	11/5
RET P	F0	11/5
RETI	ED 4D	14
RETN	ED 45	14
RL (HL)	CB 16	15
RL (IX+dis)	DD CB XX 16	23
RL (IY+dis)	FD CB XX 16	23
RL A	CB 17	8
RL B	CB 10	8
RL C	CB 11	8
RL D	CB 12	8
RL E	CB 13	8
RL H	CB 14	8
RL L	CB 15	8
RLA	17	4
RLC (HL)	CB 06	15
RLC (IX+dis)	DD CB XX 06	23
RLC (IY+dis)	FD CB XX 06	23
RLC A	CB 07	8
RLC B	CB 00	8
RLC C	CB 01	8
RLC D	CB 02	8
RLC E	CB 03	8
RLC H	CB 04	8
RLC L	CB 05	8
RLCA	07	4
RLD	ED 6F	18
RR (HL)	CB 1E	15
RR (IX+dis)	DD CB XX 1E	23
RR (IY+dis)	FD CB XX 1E	23
RR A	CB 1F	8
RR B	CB 18	8
RR C	CB 19	8
RR D	CB 1A	8
RR E	CB 1B	8
RR H	CB 1C	8
RR L	CB 1D	8
RRA	1F	4
RRC (HL)	CB 0E	15
RRC (IX+dis)	DD CB XX 0E	23
RRC (IY+dis)	FD CB XX 0E	23
RRC A	CB 0F	8
RRC B	CB 08	8
RRC C	CB 09	8
RRC D	CB 0A	8
RRC E	CB 0B	8
RRC H	CB 0C	8
RRC L	CB 0D	8
RRCA	0F	4
RRD	ED 67	18
RST 00	C7	11
RST 08	CF	11
RST 10	D7	11
RST 18	DF	11
RST 20	E7	11
RST 28	EF	11
RST 30	F7	11
RST 38	FF	11
SBC A,(HL)	9E	7
SBC A,(IX+dis)	DD 9E	19
SBC A,(IY+dis)	FD 9E	19
SBC A,A	9F	4
SBC A,B	98	4
SBC A,C	99	4
SBC A,D	9A	4
SBC A,E	9B	4
SBC A,H	9C	4
SBC A,L	9D	4
SBC A,NN	DE NN	7
SBC HL,BC	ED 42	15
SBC HL,DE	ED 52	15
SBC HL,HL	ED 62	15
SBC HL,SP	ED 72	15
SCF	37	4
SET 0,(HL)	CB C6	15
SET 0,(IX+dis)	DD CB XX C6	23
SET 0,(IY+dis)	FD CB XX C6	23
SET 0,A	CB C7	8
SET 0,B	CB C0	8
SET 0,C	CB C1	8
SET 0,D	CB C2	8
SET 0,E	CB C3	8
SET 0,H	CB C4	8
SET 0,L	CB C5	8
SET 1,(HL)	CB CE	15
SET 1,(IX+dis)	DD CB XX CE	23
SET 1,(IY+dis)	FD CB XX CE	23
SET 1,A	CB CF	8
SET 1,B	CB C8	8
SET 1,C	CB C9	8
SET 1,D	CB CA	8
SET 1,E	CB CB	8
SET 1,H	CB CC	8
SET 1,L	CB CD	8
SET 2,(HL)	CB D6	15
SET 2,(IX+dis)	DD CB XX D6	23
SET 2,(IY+dis)	FD CB XX D6	23
SET 2,A	CB D7	8
SET 2,B	CB D0	8
SET 2,C	CB D1	8
SET 2,D	CB D2	8
SET 2,E	CB D3	8
SET 2,H	CB D4	8
SET 2,L	CB D5	8
SET 3,(HL)	CB DE	15
SET 3,(IX+dis)	DD CB XX DE	23
SET 3,(IY+dis)	FD CB XX DE	23
SET 3,A	CB DF	8
SET 3,B	CB D8	8
SET 3,C	CB D9	8
SET 3,D	CB DA	8
SET 3,E	CB DB	8
SET 3,H	CB DC	8
SET 3,L	CB DD	8
SET 4,(HL)	CB E6	15
SET 4,(IX+dis)	DD CB XX E6	23
SET 4,(IY+dis)	FD CB XX E6	23
SET 4,A	CB E7	8
SET 4,B	CB E0	8
SET 4,C	CB E1	8
SET 4,D	CB E2	8
SET 4,E	CB E3	8
SET 4,H	CB E4	8
SET 4,L	CB E5	8
SET 5,(HL)	CB EE	15
SET 5,(IX+dis)	DD CB XX EE	23
SET 5,(IY+dis)	FD CB XX EE	23
SET 5,A	CB EF	8
SET 5,B	CB E8	8
SET 5,C	CB E9	8
SET 5,D	CB EA	8
SET 5,E	CB EB	8
SET 5,H	CB EC	8
SET 5,L	CB ED	8
SET 6,(HL)	CB F6	15
SET 6,(IX+dis)	DD CB XX F6	23
SET 6,(IY+dis)	FD CB XX F6	23
SET 6,A	CB F7	8
SET 6,B	CB F0	8
SET 6,C	CB F1	8
SET 6,D	CB F2	8
SET 6,E	CB F3	8
SET 6,H	CB F4	8
SET 6,L	CB F5	8
SET 7,(HL)	CB FE	15
SET 7,(IX+dis)	DD CB XX FE	23
SET 7,(IY+dis)	FD CB XX FE	23
SET 7,A	CB FF	8
SET 7,B	CB F8	8
SET 7,C	CB F9	8
SET 7,D	CB FA	8
SET 7,E	CB FB	8
SET 7,H	CB FC	8
SET 7,L	CB FD	8
SLA (HL)	CB 26	15
SLA (IX+dis)	DD CB XX 26	23
SLA (IY+dis)	FD CB XX 26	23
SLA A	CB 27	8
SLA B	CB 20	8
SLA C	CB 21	8
SLA D	CB 22	8
SLA E	CB 23	8
SLA H	CB 24	8
SLA L	CB 25	8
SRA (HL)	CB 2E	15
SRA (IX+dis)	DD CB XX 2E	23
SRA (IY+dis)	FD CB XX 2E	23
SRA A	CB 2F	8
SRA B	CB 28	8
SRA C	CB 29	8
SRA D	CB 2A	8
SRA E	CB 2B	8
SRA H	CB 2C	8
SRA L	CB 2D	8
SRL (HL)	CB 3E	15
SRL (IX+dis)	DD CB XX 3E	23
SRL (IY+dis)	FD CB XX 3E	23
SRL A	CB 3F	8
SRL B	CB 38	8
SRL C	CB 39	8
SRL D	CB 3A	8
SRL E	CB 3B	8
SRL H	CB 3C	8
SRL L	CB 3D	8
SUB (HL)	96	7
SUB (IX+dis)	DD 96 XX	19
SUB (IY+dis)	FD 96 XX	19
SUB A	97	4
SUB B	90	4
SUB C	91	4
SUB D	92	4
SUB E	93	4
SUB H	94	4
SUB L	95	4
SUB NN	D6 NN	7
XOR (HL)	AE	7
XOR (IX+dis)	DD AE XX	19
XOR (IY+dis)	FD AE XX	19
XOR A	AF	4
XOR B	A8	4
XOR C	A9	4
XOR D	AA	4
XOR E	AB	4
XOR H	AC	4
XOR L	AD	4
XOR NN	EE NN	7

****

Několik základních assemblerových operací

****

```
--------------------------- Nulování ---------------------------
- akumulátoru .... SUB A ... nebo: XOR A ... nebo: LD A,00
- registru ..... LD r,00 ;resp.LD r,nul.reg. či nul.místo paměti
- místa paměti ... LD HL,ADR ... nebo: SUB A ;resp.XOR A,LD A,00
                   LD (HL),00          LD (ADR),A
- reg.páru ..... LD rr,0000 ;resp.LD rr,(ADR),kde (ADR)=0
- dvou sousedících míst paměti ADR a ADR+1 ..... LD HL,0000
                                                 LD (ADR),HL
- indikátoru CY ..... AND A ... nebo: OR A ... nebo: SCF
  (AND A a OR A nemění obsah akumulátoru)              CCF

------- Převedení programového řízení (větvení, skoky) ---------
        na adresu obsaženou:
- ve dvou spodních bajtech zásobníku ................. RET
- v reg.HL ........................................... JP (HL)
- v reg.DE ........................................... EX DE,HL
                                                       JP (HL)
- v index.reg. .......................... JP (IX); resp. JP (IY)
- ve dvou místech paměti ADR a ADR+1 ................. LD HL,ADR
                                                       JP (HL)
- v tabulce skokových adres  (na vstupu reg.A obsahuje  pořadové
číslo - počítáno od nuly - adresy v tabulce):
         LD HL,SKOTAB  ;Adresa tabulky do reg.HL
         ADD A,A       ;Pořadové číslo hledané adresy krát 2
         LD E,A        ;Přenos obsahu reg.A do reg.E
         LD D,00       ;Vynulování reg.D
         ADD HL,DE     ;Nalezení místa uložení hledané adresy
         LD E,(HL)     ;Přenos obsahů nižšího (do reg.E) a
         INC HL        ;vyššího (do reg.D) bajtu hledané adresy
         LD D,(HL)
         EX DE,HL      ;Výměna obsahů reg.DE s reg.HL
         JP (HL)       ;Skok na hledanou adresu
SKOTAB   DEFW NULTA    ;Tabulka skokových adres
         DEFW PRVNI
         DEFW DRUHA
         ...
-  (alternativa  předchozího  příkladu  s  použitím  programové
modifikace):
         LD HL,SKOTAB  ;Totéž jako výše
         ADD A,A
         LD C,A        ;Přenos obsahu reg.A do reg.c
         LD B,00       ;Vynulování reg.B
         ADD HL,BC     ;Nalezení místa uložení hledané adresy
         LD C,2        ;Inic.reg.C pro přenos 2 bajtů (B=0)
         LD DE,SKOK+1  ;Do reg.DE adresa o 1 vyšší než SKOK
         LDIR          ;Přenos na adresy SKOK+1 a SKOK+2
SKOK     JP 0000       ;Místo instrukce JP možno použít i CALL
  Pro vlastní přenos dat lze v tomto případě použít i jiný  sled
instrukcí, např.:
         LD C,(HL)
         INC HL
         LD B,(HL)
         LD (SKOK+1),BC

---------------- Podmíněné programové větvení ------------------
Když obsah testovaného prvku -  registru,  místa  paměti,  bitu,
IFF2 - je nulový ("test na nulu"):
----------------------------------------------------------------
- akumulátoru ...................................... AND A
                                                     JR Z,SKOK
- 8-bitového registru .............................. INC r
                                                     DEC r
                                                     JR Z,SKOK
- místa paměti ..................... LD HL,ADR  nebo: LD A,(HL)
                                     INC (HL)         AND A
                                     DEC (HL)         JR Z,SKOK
                                                      JR Z,SKOK
- párového registru ................................ LD A,r1
                                                     OR r2
                                                     JR Z,SKOK
- indexového registru .............................. PUSH xy
                                                     POP rr
                                                     LD A,r1
                                                     OR r2
                                                     JR Z,SKOK
- obsahů dvou sousedících míst paměti .............. LD HL,(ADR)
                                                     LD A,H
                                                     OR L
                                                     JR Z,SKOK
- bitu registru .................................... BIT n,r
                                                     JR Z,SKOK
- bitu místa paměti ................................ LD HL,ADR
                                                     BIT n,(HL)
                                                     JR Z,SKOK
- bitu 7 akumulátoru ............. AND A       nebo: RLA
                                   JP P,SKOK         JR NC,SKOK
- bitu 6 akumulátoru ............................... ADD A,A
                                                     JP P,SKOK
- bitu 0 akumulátoru ............................... RRA
                                                     JR NC,SKOK
- IFF2 (přerušení je zablokováno) .................. LD A,I
                                                     JP PO,SKOK

----------------------------------------------------------------
Když obsah testovaného prvku (registru, místa paměti, bitu, IFF)
je různý od nuly, pak ve výše uvedených testech:
----------------------------------------------------------------
- místo JR Z,SKOK použijeme JR NZ,SKOK
-   "   JP P,SKOK     "     JP M,SKOK
-   "   JR NC,SKOK    "     JR C,SKOK
-   "   JP PO,SKOK    "     JP PE,SKOK

----------------------------------------------------------------
Větvení na základě testu zjištění obsahu prvků:
----------------------------------------------------------------
- je akumulátor nulový? ............................ CP NN
                                                     JR Z,SKOK
- je obsah registru=1? ............................. DEC r
                                                     JR Z,SKOK
- je obsah registru=FFH? ........................... INC r
                                                     JR Z,SKOK
- jsou obsahy akumulátoru a registru shodné? ....... CP r
                                                     JR Z,SKOK
- jsou obsahy akumulátoru a místa paměti shodné? ... LD HL,ADR
                                                     CP (HL)
                                                     JR Z,SKOK
- je obsah párového registru rr shodný s určitou 16-bitovou
  hodnotou NNNN? ................................... LD HL,NNNN
                                                     AND A
                                                     SBC HL,rr
                                                     JR Z,SKOK
- je obsah dvou sousedících bajtů spodku zásobníku shodný
  s určitou 16bitovou hodnotou? .................... LD HL,NNNN
                                                     AND A
                                                     SBC HL,SP
                                                     JR Z,SKOK
- totéž v porovnání s obsahem reg.HL ............... AND A
                                                     SBC HL,SP
                                                     JR Z,SKOK
- je obsah indexového registru xy shodný s určitou 16bitovou
  hodnotou NNNN? ................................... PUSH xy
                                                     POP rr
                                                     LD HL,NNNN
                                                     AND A
                                                     SBC HL,SP
                                                     JR Z,SKOK
  Pozn.-chceme-li provést větvení:
  a/ na  základě  neshodnosti  prvků,  ve  všech  případech  výše
uvedených místo instrukce JR Z,SKOK použijeme JR NZ,SKOK;
  b/ je-li obsah  prvku  pozitivním  (resp.negativním)  dvojkově
komplementárním  číslem,  místo  JR  Z,SKOK  užijeme  JP  P,SKOK
(resp.JP M,SKOK); u testů  s  index.reg.  můžeme  použít  kratší
sekvence: PUSH xy
          POP AF
          AND A        ;stačí test vyššího bajtu
          JP P,SKOK    ;(resp.JP M,SKOK pro negat.č.)
...k tomuto bodu dále:

- je 16-bitové číslo na adresách ADR a ADR+1 pozitivní (resp.
  negativní)?
     1/ LD A,(ADR+1)   ;stačí test vyššího bajtu
        AND A
        JP P,SKOK      ;(resp.JP M,SKOK pro negat.č.)
     2/ LD HL,ADR+1
        BIT 7,(HL)     ;test bitu 7 vyššího bajtu
        JR Z,SKOK      ;(resp.JR NZ,SKOK pro negat.č.)

----------------------------------------------------------------
Větvení  při  použití   dvojkově   komplementární   reprezentace
(nejvyšší bit indikuje znaménko +, je-li ve stavu log.0, nebo  -
při log.1):
----------------------------------------------------------------
  Pozn.: návěští NIC symbolizuje skok  při  nesplnění  podmínky,
SKOK při jejím splnění; hodn je symbolická zkratka pro  číselnou
hodnotu, s níž je porovnáván obsah akumulátoru (může  jí  být  i
obsah registru nebo místa paměti).
```

- je DK obsah akumulátoru větší než určitá DK číselná hodnota?

```
        CP hodn
        JP PE,TEST    ;Když V=1, skok na TEST
        JP M,NIC      ;Když V=0 a S=1, je akum.menší
        JR NZ,SKOK    ;Když V=0, S=0 a Z=0, podmínka splněna
        JR NIC        ;Když V=0, S=0 a Z=1, je výsledkem 0
TEST    JP M,SKOK     ;Když V=1 a S=1, podmínka splněna
        JR NIC        ;Když V=1 a S=0, je akum.menší
```

Tato sekvence převede programové řízení na adresu SKOK, když je výsledek porovnání větší než 0 a přitom nedojde k aritmetickému přetečení, nebo když je výsledek menší než nula a přitom dojde k aritmetickému přetečení.

- je DK obsah akumulátoru větší než určitá DK číselná hodnota či je s ní shodný?

```
        CP hodn
        JP PE,TEST    ;Když V=1, skok na TEST
        JP P,SKOK     ;Když V=0 a S=0, podmínka splněna
        JR NIC        ;Když V=0 a S=1, je akum.menší
TEST    JP M,SKOK     ;Když V=1 a S=1, podmínka splněna
        JR NIC        ;Když V=1 a S=0, je akum.menší
```

Tato sekvence převede programové řízení na adresu SKOK, když je výsledek větší než 0, či jí roven, a přitom nedojde k aritmetickému přetečení, nebo když je výsledek menší než 0 a přitom dojde k aritmetickému přetečení.

- je DK obsah akumulátoru menší než určitá DK číselná hodnota?

```
        CP hodn
        JP PE,TEST    ;Když V=1,skok na TEST
        JP M,SKOK     ;Když V=0 a S=1, podmínka splněna
        JR NIC        ;Když V=0 a S=0, je akum.větší nebo shodný
TEST    JP P,SKOK     ;Když V=1 a S=0, podmínka splněna
        JR NIC        ;Když V=1 a S=1, je akum.větší nebo shodný
```

Tato sekvence převede programové řízení na adresu SKOK, když je výsledek menší než 0 a přitom nedojde k aritmetickému přetečení, nebo když je výsledek větší než 0, či jí roven, a přitom dojde k aritmetickému přetečení.

- je DK obsah akumulátoru menší než určitá DK číselná hodnota či je s ní shodný?

```
        CP hodn
        JP PE,TEST    ;Když V=1, skok na TEST
        JP M,SKOK     ;Když V=0 a S=1, podmínka splněna
        JR Z,SKOK     ;Když V=0, S=0 a Z=1, podmínka splněna
        JR NIC        ;Když V=0, S=0 a Z=0, je akum.větší
TEST    JP M,NIC      ;Když V=1 a S=1, je akum.větší
        JR NZ,SKOK    ;Když V=1, S=0 a Z=0, podmínka splněna
        JR NIC        ;Když V=1, S=0 a Z=1, je akum.větší
```

Tato sekvence převede programové řízení na adresu SKOK, když je výsledek menší než nula, či jí roven, a přitom nedojde k aritmetickému přetečení, nebo když je výsledek větší než 0 a přitom dojde k aritmetickému přetečení.

Další příklady programového větvení (bez dvojk. kompl.), když:

- je obsah akumulátoru větší než určitá hodnota N:

```
        CP N                  nebo:  CP N+1       ;CP shodných čí-
        JR C,NIC                     JR NC,SKOK   ;sel nuluje CY
        JR NZ,SKOK
```

- je obsah akumulátoru větší než obsah registru:

```
        CP r                  nebo:  INC r        ;Resp.DEC A
        JR C,NIC                     CP r
        JR NZ,SKOK                   JR NC,SKOK
```

- je obsah akumulátoru větší než obsah adresy:

```
        LD HL,ADR             nebo:  LD r,A
        CP (HL)                      LD A,(ADR)
        JR C,NIC                     CP r
        JR NZ,SKOK                   JR C,SKOK
```

- je obsah reg.HL větší než jiný párový registr rr:

```
        SCF           ;CY=log.1
        SBC HL,rr     ;HL-(rr+1)
        JR NC,SKOK
```

- je obsah reg.HL větší než určitá hodnota NN:

```
        LD rr,-NN-1
        ADD HL,rr     ;přičtení DK
        JR C,SKOK
```

- je obsah reg.SP větší než obsah reg.HL:

```
        AND A         ;Vynulování CY
        SBC HL,SP
        JR C,SKOK
```

- je obsah index.reg. větší než obsah reg.HL:

```
        PUSH IX       ;Resp.IY
        POP rr        ;Kromě HL
        AND A         ;Vynulování CY
        SBC HL,rr     ;"De facto" HL-IX
        JR C,SKOK
```

- je obsah akumulátoru menší nebo roven určité hodnotě N:

```
        CP N                  nebo:  CP N+1
        JR C,SKOK                    JR C,SKOK
        JR Z,SKOK
```

- je obsah akumulátoru menší nebo roven obsahu jiného registru:

```
        CP r                  nebo:  INC r
        JR C,SKOK                    CP r
        JR Z,SKOK                    JR C,SKOK
```

- je obsah akumulátoru větší nebo roven obsahu adresy:

```
        LD HL,ADR             nebo:  LD r,A
        CP (HL)                      LD (ADR),A
        JR C,SKOK                    CP r
        JR Z,SKOK                    JR NC,SKOK
```

- je obsah reg.HL menší nebo roven párovému registru:

```
        SCF
        SBC HL,rr
        JR C,SKOK
```

- je obsah reg.HL menší nebo roven dvoubajtové hodnotě NN:

```
        LD rr,-NN-1
        ADD HL,rr     ;HL-(NN+1)
        JR NC,SKOK
```

- je obsah reg.SP menší nebo roven obsahu reg.HL:

```
        AND A         ;VynulováNí CY
        SBC HL,SP
        JR NC,SKOK
```

- je obsah index.reg. menší nebo roven obsahu reg.HL:

```
        PUSH IX       ;Resp.IY
        POP rr        ;Kromě HL
        AND A         ;Vynulování CY
        SBC HL,rr     ;"De facto" HL-IX
        JR NC,SKOK
```

- je obsah akum.menší než určitá hodnota N (obsah jiného reg.):

```
        CP N          ;Nebo CP r
        JR C,SKOK
```

- je obsah akumulátoru menší než obsah adresy:

```
        LD HL,ADR
        CP (HL)
        JR C,SKOK
```

- je obsah reg.HL menší než obsah jiného párového registru:

```
        AND A         ;Vynulování CY
        SBC HL,rr
        JR C,SKOK
```

- je obsah reg.HL menší než dvoubajtová hodnota NN:

```
        LD rr,-NN
        ADD HL,rr     ;HL+(-NN)
        JR NC,SKOK
```

- je obsah reg.SP menší než obsah reg.HL:

```
        SCF           ;CY=log.1
        SBC HL,SP
        JR NC,SKOK
```

- je obsah index.reg. menší než obsah reg.HL:

```
        PUSH IX       ;Resp.IY
        POP rr        ;Kromě HL
        SCF
        SBC HL,rr
        JR NC,SKOK
```

- je obsah akumulátoru větší nebo roven určité hodnotě N (nebo obsahu jiného registru):

```
        CP N          ;Nebo CP r
        JR NC,SKOK
```

- je obsah akumulátoru větší nebo roven obsahu adresy:

```
        LD HL,ADR
        CP (HL)
        JR NC,SKOK
```

- je obsah reg.HL větší nebo roven obsahu jiného párového reg.:

```
        AND A
        SBC HL,rr
        JR NC,SKOK
```

- je obsah reg.HL větší nebo roven dvoubajtové hodnotě NN:

```
        LD rr,-NN
        ADD HL,rr     ;HL+(-NN)
        JR C,SKOK
```

- je obsah reg.SP větší nebo roven obsahu reg.HL:

```
        SCF           ;CY=log.1
        SBC HL,SP     ;HL-(SP+1)
        JR C,SKOK
```

- je obsah index.reg. větší nebo roven obsahu reg.HL:

```
        PUSH IX       ;Resp.IY
        POP rr        ;Kromě HL
        SCF
        SBC HL,rr
        JR C,SKOK
```

- akumulátor obsahuje platné BCD číslo:

```
        LD r,A        (Pozn.:        zvýšení BCD o 1: ADD A,1   )
        ADD A,0       (                               DAA        )
        DAA           (                                          )
        CP r          (snížení BCD o 1: SUB 1   nebo: ADD A,99   )
        JR NZ,SKOK    (                 DAA           DAA        )
```