Programování v assembleru (vpa)

(pomocný studijní text)

Ing,. Jan Roupec, Ph.D., Ústav automatizace a informatiky FSI VUT v Brně

Brno, listopad 2004

Obsah

0 ÚVOD

Zvyšování výkonu počítačů a snižování jejich cen má za následek neustálé rozšiřování aplikačních oblastí počítačů a také nutnost vytváření stále komplikovanějších aplikací. Tato komplikovanost se týká nejen vlastního algoritmu, ale hlavně luxusu a pohodlí obsluhy. Tvorba sw proto vyžaduje vývoj nových výkonnějších programátorských technologií. Ve světle uvedených skutečností používání assembleru jakožto jazyka nejbližšího stroji (strojový kód již nelze za jazyk považovat) ztrácí na významu. Tvorba rozsáhlých aplikací splňujících dnešní požadavky na profesionální sw v assembleru je v zásadě nemyslitelná, neboť by vyžadovala nereálně velké týmy programátorů a spotřebovala nepřijatelně mnoho času.

Přesto není znalost programování v assembleru pro profesionály v oblasti informatiky nadbytečná. Jednak umožňuje mnohem lépe než cokoliv jiného pochopit děje v počítači pobíhající, jednak se tvorba programů v assembleru v některých oblastech stále ještě udržuje. Jedná se zejména o programování jednoúčelových řídících aplikací nejčastěji s využitím jednočipových mikropočítačů, programování driverů pro obsluhu různých zařízení na rozličných platformách, mikrokontroléry apod. I při používání počítačů dnešní výkonnosti stále ještě občas nastávají situace, kdy naprogramování některých speciálních akcí ve vyšším programovacím jazyce nezajišťuje požadovanou rychlost. Zde se často používá technika zvaná *mixed language programming*, kdy se každá část projektu naprogramuje v jiném, pro tuto část vhodném programovacím jazyce.

Problémem assembleru je jeho závislost na konkrétní platformě. Tento pomocný studijní text si neklade za cíl popsat do všech podrobností konkrétní assembler pro konkrétní počítač a tak suplovat technickou dokumentaci. Snaha popsat principy programování v assembleru zcela obecně nutně musí selhat pro rozmanitost assemblerů a instrukčních souborů konkrétních procesorů. Text se snaží o kompromis mezi těmito krajními přístupy. Za základ byl vzat assembler pro procesor I8086, jednotlivé konstrukce ale nepopisují všechny detailní rysy tohoto assembleru, ale snaží se o obecnější pohled. Volba procesoru a assembleru nebyla náhodná – procesory nejrozšířenějších počítačů PC dodržují zpětnou kompatibilitu s procesorem I8086, a tak je možné si získané poznatky snadno prakticky ověřovat na běžných PC. V textu je obsažen popis těch konkrétních vlastností asembleru I8086, bez kterých nelze v tomto assembleru psát programy nebo které lze zobecňovat i pro jiné assemblery, naopak specifické vlastnosti, které jsou zbytné a které nemají obdobu u jiných assemblerů, popisovány zcela záměrně nejsou.

1 OPAKOVÁNÍ ZÁKLADNÍCH POJMŮ A OPERACÍ

Při programování v assembleru se programátor pohybuje velmi blízko stroji – počítači. Počítač vnitřně funguje ve dvojkové soustavě, která je pro člověka nezvyklá. K názornému popisu dějů v počítači, obsahu jednotlivých paměťových míst apod. není „oblíbená“ desítková soustava vhodná, neboť neexistuje žádný jednoduchý výpočetně nenáročný postup, kterým by bylo možné převádět hodnoty zapsané ve dvojkové soustavě na zápis desítkový a naopak (příčinou je, že hodnota deset není celistvou mocninou hodnoty dvě). Proto se v oblasti výpočetní techniky často používají kromě dvojkové a desítkové ještě soustavy, které mají ke dvojkové soustavě „přátelštější“ vztah než soustava desítková. I když znalost číselných soustav patří k základním matematickým (či spíše početním) dovednostem, skutečné znalosti a schopnost praktického používání nedesítkových soustav bývají nevalné. Pro úspěšné zvládnutí programování na úrovni blízké stroji je práce s nedesítkovými soustavami nezbytně nutná, proto je vhodné tuto problematiku v lepším případě zopakovat, v horším vyložit.

1.1 Číselné soustavy

Na první pohled by se mohlo zdát, že v praxi se používá výhradně desítková soustava a že tomu tak bylo vždy. Tento dojem je ale mylný. V minulosti byly nedesítkové soustavy naopak velmi rozšířené, připomeňme alespoň počítání na tucty (12), veletucty (12 x 12 = 144) a kopy (60 – i dnes obsahuje jedno plato 30 vajec, kupujeme tedy půl kopy), časové nebo úhlové jednotky (den má 24 hodin, hodina má 60 minut, minuta 60 sekund, podobně kruh má 360 stupňů, stupeň má 60 úhlových minut a každá tato minuta 60 úhlových vteřin), v neposlední řadě lze zmínit i starou anglickou měnu v soustavě libra, šilink, pence a podobných příkladů by se našlo velmi mnoho. Přesto při aritmetických operacích automaticky předpokládáme použití desítkové soustavy.

Z hlediska prezentace informace lze používat zobrazení *analogové* (např. délka rtuťového sloupce rtuťového teploměru) nebo *diskrétní*. Diskrétní zobrazení může být tvořeno alfanumerickými znaky (abeceda) nebo čísly, případně jinak (např. řadou čárek, počtem prstů apod.). Zápis hodnoty pomocí čísel může být *poziční* (polyadický) nebo *nepoziční* (nepolyadický). K nepozičním systémům patří např. římské číslice nebo kód BCD. Běžně se číselné hodnoty zapisují v pozičních systémech.

Poziční systém zápisu číselné hodnoty předpokládá použití číselné soustavy o nějakém základu. Volbu právě hodnoty deset za základ nejběžněji používané číselné soustavy lze považovat za náhodu, jako pravděpodobná příčina se obvykle uvádí skutečnost, že člověk má deset prstů a počítání na prstech se u málo zdatných počtářů praktikuje dosti často. Teoreticky je tento základ nepříliš výhodný – deset nemá celočíselnou odmocninu a má pouze dva možné dělitele – hodnoty dvě a pět. Teoreticky nejvýhodnější by byl základ číselné soustavy rovný hodnotě Eulerova čísla *e,* ovšem prakticky je nutné, aby základem číselné soustavy bylo celé číslo, a tuto podmínku *e* nesplňuje. Jako vhodný volba by se mohla jevit k *e* nejbližší celá hodnota – tři, soustava o tomto základu se ale v praxi nepoužívá vůbec. Technicky lze realizovat nejlépe výpočty v soustavě se základem dvě, protože taková soustava obsahuje pouze dvě cifry a dvoustavová zařízení lze snadno realizovat elektronicky i jinak. Zajímavá je i souvislost s logikou, ve které také potřebujeme právě dvě hodnoty – nepravda a pravda. Ve dvojkové soustavě skutečně lze vysledovat souvislosti mezi aritmetickými a logickými operacemi. Jednou z nejdůležitějších zásad pro pochopení práce s různými číselnými soustavami (a s různými způsoby zápisu číselných hodnot vůbec) je fakt, že **hodnota je atribut nezávislý na způsobu**

zápisu. To znamená, že pokud známe např. např. počet rohlíků v bedně, tato hodnota se nemůže změnit pouze tím, že ji zapíšeme pomocí římských číslic, v desítkové soustavě nebo ve dvojkové soustavě, nebo ji vyjádříme třeba odpovídajícím počtem čárek, zápalek apod.

Samotný princip pozičních systémů znamená, že hodnota, kterou symbol vyjadřuje, nezávisí pouze na druhu symbolu, ale také na pozici, na které se vyskytuje. Např. v zápisu hodnoty **212** (dvě stě dvanáct) se vyskytují pouze dva symboly (jednička a dvojka). Dvojka se vyskytuje dvakrát, každý její výskyt ale představuje jinou hodnotu (jiný počet „rohlíků“) – první výskyt symbolu **2** znamená hodnotu 200 (dvě stě), druhý hodnotu 2 (dva). Hodnota, kterou symbol skutečně vyjadřuje, je součinem hodnoty samotného symbolu (v tomto případě dvě) a jeho váhy (první výskyt symbolu **2** zleva má v tomto případě váhu 100, druhý 1). Zápis n+1 ciferného čísla v desítkové soustavě lze symbolicky zapsat:

$$a_n a_{n-1} \ldots a_1 a_0, \quad a_i \in \{0,1,2,\ldots,9\}$$

zapisovaná hodnota A se vypočítá

$$A = a_0 \cdot 10^0 + a_1 \cdot 10^1 + \ldots + a_n \cdot 10^n = \sum_{i=0}^{n} a_i \cdot 10^i \text{ (1)} \tag{1.1}$$

Při používání desítkové soustavy se uvedený výpočet vědomě neprovádí, je nahrazen „přečtením“ zapsaného čísla, které dává představu o zapisované hodnotě (tomu se člověk odmala učí). Přitom každou hodnotu lze zapsat mnoha způsoby, omezíme-li se na poziční systémy, lze pro základ namísto hodnoty 10 použít jakoukoliv jinou celočíselnou hodnotu větší než jedna. Obecně si lze zápis hodnoty v číselné soustavě o základu Z představit jako:

$$A = a_0 \cdot Z^0 + a_1 \cdot Z^1 + \ldots + a_n \cdot Z^n = \sum_{i=0}^{n} a_i \cdot Z^i \tag{1.2}$$

Pro odlišení se hodnoty zapsané v soustavě o jiném základu než deset označují základem jako dolním pravostranným indexem (někdy v závorce), např. **1234**$_7$ nebo **1234**$_{(7)}$ nebo **1234(7)** představuje hodnotu zapsanou v sedmičkové soustavě. V některých případech se používá prefixová nebo postfixová notace (např. zápis **$1234** v jazyce Pascal znamená hodnotu zapsanou v šestnáctkové soustavě, tedy **1234**$_{(16)}$).

K zápisu hodnoty v soustavě o základu Z je nutné mít k dispozici symboly pro hodnoty nula až Z-1 (hodnota Z již znamená přechod o jeden řád nahoru). Pro soustavy o základech dva až deset (hodnota menší než dvě nemůže být základem číselné soustavy) jsou těmito symboly běžné (tzv. arabské) číslice. Pro soustavy o základu větším než deset již se symboly číslic nelze vystačit a je nutné zavést symboly pro další hodnoty (o hodnotách deset a více), obvykle se používají písmena anglické abecedy (symbol **A** znamená hodnotu 10, **B** znamená 11 atd.). Není tedy nic neobvyklého spatřit zápis např. ve tvaru **1ABC29F**$_{(16)}$. Při použití soustav o základu větším než 10 (tedy při používání písmen jako symbolů cifer) se silně doporučuje, aby zapisovaná hodnota začínala „skutečnou“ číslicí; toho se dosahuje užíváním levostranné nuly (např. zápisu **0BABA**$_{(16)}$ se dává přednost před zápisem **BABA**$_{(16)}$).

Jak bylo uvedeno, hodnota nezávisí na způsobu, jakým je zapsaná, přitom lze k zápisu hodnot používat různé způsoby. Pokud hodnotu známe (nebo známe některý její zápis), můžeme jednoznačně vytvořit všechny její ostatní zápisy (nepřesně se hovoří o „převodech číselných soustav“). K převodům zápisu v jedné soustavě do soustavy jiné je nezbytně nutné provádět aritmetické výpočty. Protože člověk obvykle umí počítat v soustavě desítkové, liší se způsob převodu zápisu podle toho, zda desítková soustava je zdrojovou nebo cílovou soustavou (nebo není ani zdrojovou, ani cílovou).

1.1.1 Převod ze zápisu v nedesítkové soustavě do soustavy desítkové

Rovnici (1.2) lze přímo chápat jako návod k převodu zápisu v soustavě s nedesítkovým základem do soustavy desítkové, postačuje dosadit konkrétní hodnoty. Tak např. pro hodnotu zapsanou v sedmičkové soustavě jako $\mathbf{1234_{(7)}}$ lze k zápisu v desítkové soustavě dojít jednoduchým výpočtem:

$$A = 4 \cdot 7^0 + 3 \cdot 7^1 + 2 \cdot 7^2 + 1 \cdot 7^3 = 4 \cdot 1 + 3 \cdot 7 + 2 \cdot 49 + 1 \cdot 343 = 466_{(10)}$$

1.1.2 Převod ze zápisu v desítkové soustavě do soustavy nedesítkové

V tomto případě je cílovou soustavou soustava nedesítková. Protože v nedesítkové soustavě obvykle člověk počítat neumí, nelze použít vztah (1.2) a všechny výpočty je třeba provádět ve výchozí desítkové soustavě. Nejpoužívanějším postupem v tomto případě je postupné dělení novým základem a zapisování zbytků – původní hodnotu dělíme novým základem, zbytek bude poslední (nejvíce vpravo) cifrou výsledku, podíl znovu dělíme novým základem (zbytek bude předposlední – druhou zprava – cifrou výsledku, takto se pokračuje dále, dokud podíl nebude nula – poslední zbytek je první (nejvíce vlevo) cifrou výsledku. Např. hodnotu $\mathbf{1234_{(10)}}$ lze v osmičkové soustavě zapsat po provedení následujících výpočtů:

1234 : 8 = 154, zb. 2
154 : 8 = 19, zb. 2
19 : 8 = 2, zb. 3
2 : 8 = 0, zb. 2

výsledný zápis je $\mathbf{2322_{(8)}}$. Obvykle se uvedený výpočet zapisuje do tabulky:

1234	8
154	**2**
19	**2**
2	**3**
0	**2**

kde v záhlaví je původní desítkový zápis a nový základ, v dalších řádcích je v prvním sloupci podíl a ve druhém zbytek, novým zápisem jsou potom zbytky (hodnoty ve druhém sloupci) postupně zapisované odzadu. Uvedený postup lze snadno mechanicky zapamatovat (a zapomenout), se zdůvodněním ale někdy bývají problémy. Logicky lze k celé záležitosti přistupovat např. takto: při dělení čísla základem číselné soustavy dojde k posuvu řádové čárky o jedno místo doleva; při počítání se zbytkem pak poslední cifra vypadne ve formě zbytku. Např.

$1234_{(10)} : 10_{(10)} = 123$, zb. 4
$2232_{(8)} : 8_{(10)} = 223_{(8)}$, zb. 2

apod. Protože hodnota je nezávislá na způsobu, kterým je zapsaná, pak pokud platí, že

$$1234_{(10)} = 2232_{(8)},$$

nutně musí také platit

$$1234_{(10)} : 8_{(10)} = 2232_{(8)} : 10_{(8)} = 223_{(8)}, \text{ zb. } 2_{(8)} = 154_{(10)}, \text{ zb. } 2_{(10)}$$

čili postupným dělením novým základem postupně „vytlačujeme“ nejnižší cifru v nové soustavě.

Převod desítkového na nedesítkový zápis lze provádět i jinými způsoby (pro některé soustavy i „nesportovně“ s využitím kalkulačky), např. lze vycházet ze znalosti mocnin nového základu. Pokud

chceme převádět do zápisu v osmičkové soustavě, je třeba znát mocniny nového základu (váhy pro poziční zápis) – $8^0 = 1$, $8^1 = 8$, $8^2 = 64$, $8^3 = 512$, $8^4 = 4096$, … Při požadavku na zápis konkrétní hodnoty v nové nedesítkové soustavě pak postupně porovnáváme a odečítáme mocniny nového základu. Např. při požadavku zapsat $1234_{(10)}$ v osmičkové soustavě je zřejmé, že nejvyšším řádem zápisu v nové soustavě bude 8^3, v osmičkové soustavě se tedy bude jednat o čtyřciferné číslo a nejvyšší cifra bude znamenat, kolikrát je v hodnotě $1234_{(10)}$ obsažena hodnota $512_{(10)}$ – dvakrát, nejvyšší cifrou tedy bude dvojka. Zápisem číslice 2 na místo čtvrté cifry nového zápisu jsme ze zapisované hodnoty „umazali“ $2*8^3 = 1024$, na zbývajících třech cifrách je tedy nutné zapsat hodnotu $1234_{(10)} - 1024_{(10)} = 210_{(10)}$. Dále se zjišťuje, kolikrát je ve zbývající hodnotě ($210_{(10)}$) obsažena další mocnina nového základu ($8^2 = 64$) a postup se opakuje, dokud není zapsána celá hodnota. Jak je vidět, opět se vychází z principu pozičního zápisu, pouze se zjišťují číslice v opačném pořadí (zleva doprava).

1.1.3 Převod ze zápisu v nedesítkové soustavě do soustavy nedesítkové

Jedná se o zcela obecný případ. Obvykle člověk neumí počítat ve výchozí ani cílové soustavě. Možným řešením je rozklad problému do dvou kroků – převod z výchozího nedesítkového do desítkového zápisu a následně převod z desítkového do cílového nedesítkového zápisu. Tento postup je vhodný pro algoritmické vyjádření. Pro některé speciální případy (viz 1.1.4) lze použít jiný, jednodušší postup, který lze použít pro výpočty „zpaměti“ pouze s poznámkami na papíře a při jisté zkušenosti i bez nich. Ačkoliv se zdá, že současné technické prostředky potřebu výpočtů „zpaměti“ zcela eliminují, není to tak úplně pravda – při běžné rutinní práci je nutné znát výsledek okamžitě, nikoliv až po použití techniky (kalkulačka, program).

1.1.4 Soustavy o základech 2, 8 a 16

Počítače jsou konstruované tak, že vnitřně používají operace v číselné soustavě o základu 2. Hodnota dvě je přitom nejmenším myslitelným základem číselných soustav. Používání dvojkové soustavy člověkem má jisté problémy a nectnosti. Jednak dvojková soustava používá nejmenší myslitelný počet symbolů (dva, tedy znaky **0** a **1**), jednak vzhledem k nízkému základu obsahují zápisy běžných hodnot mnoho cifer. V důsledku toho jsou dvojkové zápisy dlouhé a obtížně zapamatovatelné. Navíc hodnota dvě je dosti vzdálená obvyklému základu číselných soustav (10), a proto člověku obvykle chybí i schopnost základní klasifikace dvojkově zapsaných hodnot ve smyslu „mnoho“, „málo“, „přiměřeně“. (Je třeba připomenout, že při zápisu hodnot ve dvojkové soustavě se dvojkové cifry nazývají **bity**, slovo bit vzniklo ze zkratky *binary digit* – dvojková číslice, současně se jedná o slovní hříčku, neboť *bit* znamená anglicky *kousek* – v počítačích se jedná o nejmenší jednotku, se kterou je stroj schopen operovat, současně jde o nejmenší množství informace, které lze přenášet nebo zapamatovat).

Při popisu dějů v počítači (přirozeně dvojkových) se desítková soustava uplatní jen velmi omezeně, neboť z desítkového zápisu nelze jednoduše poznat hodnotu jednotlivých dvojkových cifer (s jedinou výjimkou – je-li hodnota lichá, pak nejnižší cifra dvojkového zápisu je 1, jinak 0). Poměrně jednoduchý postup lze použít pro převod zápisů hodnot mezi soustavami, jejichž základ je mocninou společného čísla. Má-li být jednou z těchto soustav soustava o základu 2, pak další takovou soustavou může být soustava o základu 4, 8, 16, 64 atd.

V minulosti hrála dominantní roli soustava o základu 8, protože hodnota 8 nejen že je celistvou mocninou hodnoty 2, ale je ze všech mocnin čísla 2 nejméně vzdálená od hodnoty 10, která je „implicitním“ základem číselných soustav. Blízkost hodnotě 10 znamená, že hodnota zapsaná v osmičkové soustavě bude „opticky“ blízká zápisu téže hodnoty v desítkové soustavě, tzn. že bude možné

i pro hodnoty zapsané v osmičkové soustavě použít základní soudy typu „mnoho“, „málo“. Vzhledem k blízkosti hodnot 8 a 10 je poměrně snadné naučit se počítat v osmičkové soustavě, to platí zejména pro operaci sčítání a odčítání.

Při převodech mezi zápisy v osmičkové a dvojkové soustavě se využívá fakt, že osm je dvě na třetí, a proto jedna osmičková cifra bude zápisem tří cifer dvojkových a naopak. Při požadavku převodu zápisu dvojkové hodnoty do osmičkové soustavy lze postupovat tak. že se dvojkový zápis rozdělí zprava doleva (od řádové čárky) po třech cifrách a hodnota každé trojice dvojkových cifer se zapíše jednou cifrou osmičkovou. Např. $\mathbf{101011100110001_{(2)}}$ lze v osmičkové soustavě zapsat:

1	0	1	0	1	1	1	0	0	1	1	0	0	0	1
5			**3**			**4**			**6**			**1**		

tzn. že výsledný zápis je $\mathbf{53461_{(8)}}$. Opačný převod (z osmičkové do dvojkové soustavy) se dělá obdobně – každá osmičková cifra se zapíše pomocí **právě tří** cifer dvojkových (tzn. včetně levostranných nul!).

Desítkově	Dvojkově	Osmičkově	Šestnáctkově
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11

Tab. 1.1 – *Vztah mezi soustavami o základech 2, 8, 16*

Význam osmičkové soustavy upadl se zavedením pojmu byte (bajt), který bezprostředně souvisí s rozšířením mikroprocesorů. Původně byla délka slova procesoru různá („jak se výrobci podařilo“) a tvořila např. 10 nebo 12 bitů (dvojkových cifer). První mikroprocesor byl původně vyvinut jako integrovaný obvod pro elektronické kalkulátory, které pracovaly s čísly v kódu BCD (viz 1.2.3). Tím byla délka slova stanovena na čtyři bity (kratší slovo by funkčně nevyhovovalo, delší bylo nad technologické možnosti výroby integrovaných obvodů). Později se podařilo délku slova zdvojnásobit a osmibitové slovo se stalo na několik let standardem, kterému se přizpůsobily ostatní části počítače (paměti, sběrnice, …). Shluk osmi bitů byl pojmenován **byte** (bajt, někdy se užívá český termín slabika); k reprezentaci dat, která vyžadovala více bitů, se začaly používat skupiny bytů. Používání do té doby dominantní osmičkové soustavy ve spojení s byty ale představuje problém, neboť k popisu

hodnoty bytu jsou potřeba tři osmičkové cifry, nejvyšší z nich ale nevyužije všechny hodnoty – tři osmičkové cifry popisují devět, nikoliv osm bitů. Řešením je použití šestnáctkové soustavy – dvě šestnáctkové cifry zahrnují právě osm cifer dvojkových.

Šestnáct je dvě na čtvrtou, proto při převodech mezi dvojkovým a šestnáctkovým zápisem platí, že každá šestnáctková cifra odpovídá čtyřem cifrám dvojkovým a naopak, postup je obdobný jako u osmičkové soustavy. Pro zápis hodnot v šestnáctkové soustavě je potřebná existence cifer pro hodnoty nula až patnáct, pro hodnoty vyšší než devět se používají písmena (hodnotě deset odpovídá symbol A, hodnotě patnáct potom F). Např. zápisu hodnoty $\mathbf{101011100110001_{(2)}}$ v šestnáctkové soustavě je postup následující:

0	1	0	1	0	1	1	1	0	0	1	1	0	0	0	1
	5				**7**				**3**				**1**		

Používání šestnáctkové (hexadecimální) soustavy je natolik rozšířené, že byla zavedena zvyklost označovat hodnotu zapsanou v šestnáctkové soustavě postfixem **H** (např. **5731H**).

Při převodech mezi osmičkovou a šestnáctkovou soustavou se s výhodou nepoužívá jako mezikrok soustava desítková, ale dvojková – v tomto případě lze převod zvládnout s minimem počítání nebo dokonce zpaměti. Vztah mezi dvojkovým, osmičkovým a šestnáctkovým zápisem je uveden v tabulce 1.1.

1.1.5 Necelá čísla

Stejně jako v desítkové soustavě, i v ostatních soustavách jsou vpravo od desetinné (v tomto případě spíše řádové) čárky postupně uváděny počty záporných mocnin základu v zapisované hodnotě. Tedy např. dvojkově zapsaná hodnota lze vyjádřit desítkově:

$$\mathbf{1{,}1011_{(2)}} = 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} = 1 \cdot 1 + 1 \cdot 0{,}5 + 0 \cdot 0{,}25 + 1 \cdot 0{,}125 + \\ + 1 * 0{,}0625 = \mathbf{1{,}6875_{(10)}}$$

Napravo od řádové čárky jsou ve dvojkové soustavě namísto desetin, setin atd. postupně poloviny, čtvrtiny, osminy atd. Převod z nedesítkového zápisu do desítkového je i u necelých čísel zřejmý. Ve směru opačném se využívá podobného postupu jako u celých čísel – při postupném násobení necelého čísla menšího než 1 základem číselné soustavy se cifry nové soustavy za řádovou čárkou postupně objevují jako celá část výsledku. Např. převedeme $\mathbf{0{,}906925_{(10)}}$ do dvojkové soustavy:

0,90625 x 2 = **1**,81250
0,81250 x 2 = **1**,625
0,625 x 2 = **1**,25
0,25 x 2 = **0**,5
0,5 x 2 = **1**,0

výsledkem je dvojkový zápis **0,11101** (zapisujeme odpředu celé části součinu). Postup končí, pokud zlomová část vyjde nula. Obvykle se celý výpočet zapisuje do tabulky:

x 2	0,90625
1	81250
1	6250
1	250
0	5
1	0

kde v záhlaví je naznačeno násobení novým základem a původní hodnota ve výchozí soustavě (desítkové). V dalších řádcích je v prvním sloupci celá část součinu, ve druhém jeho zlomková část. Při požadavku na převod **0,906925$_{(10)}$** do šestnáctkové soustavy se postupuje obdobně:

x 16	0,90625
14	5
8	0

výsledkem je **0,E8H**.

Při převodu necelé hodnoty jejíž celá část je větší než jedna je nutné zvlášť převádět celou a zvlášť zlomkovou část, např. při převodu **12,906925$_{(10)}$** do dvojkové soustavy se napřed převede celá část např. postupným dělením novám základem:

12	2
6	0
3	0
1	1
0	1

(výsledek celé části je **1100$_{(2)}$**) a potom se převede zlomková část

x 2	0,90625
1	81250
1	6250
1	250
0	5
1	0

výsledný zápis je **1100,11101$_{(2)}$**.

Při převodech mezi dvojkovými, osmičkovými a šestnáctkovými zápisy necelých hodnot lze opět namísto zdlouhavých výpočtů s výhodou využít příbuznosti těchto soustav. Např. při převodu **1100,11101$_{(2)}$** do šestnáctkové soustavy:

1	1	0	0	,	1	1	1	0	1	0	0	0
C				**,**	**E**					**8**		

výsledkem je tedy 0C,E8H. Jak je vidět, opět platí, že každé čtyři dvojkové cifry se zapíší pomocí jedné cifry šestnáctkové. Výchozí dvojkový zápis se dělí po čtyřech cifrách na obě strany od řádové čárky. U zlomkové části je nezbytně nutné doplnit pravostranné nuly tak, aby i poslední skupina obsahovala čtyři číslice (pro šestnáctkovou soustavu, pro osmičkovou samozřejmě tři).

Je vhodné se zmínit o jedné skutečnosti, kterou si programátoři ve vyšších jazycích často neuvědomují. Zejména v numerických výpočtech iterativního charakteru jsou zhusta používány hodnoty $0,1_{(10)}$, $0,01_{(10)}$, $0,001_{(10)}$ atd. Přitom tyto hodnoty nelze ve dvojkové soustavě zapsat pomocí konečné-

ho počtu cifer. Např. hodnota $^1/_{10}$ se ve dvojkové soustavě zapíše jako $0{,}0001\overline{0011}$, skupina **0011** se bude periodicky opakovat. Protože počítače vnitřně pracují s hodnotami ve dvojkové soustavě, znamená to, že tyto oblíbené hodnoty budou v paměti vždy uloženy nepřesně, protože počítač samozřejmě čísla ukládá na konečný počet platných (dvojkových) cifer. Tato nepřesnost může vést k numerickým chybám ve výpočtech, zejména při velmi mnoha opakováních cyklů nebo při práci s paměťově úspornými datovými typy, které používají k uložení hodnot malý počet bitů. Vhodným řešením je používání hodnot, které ve dvojkové soustavě na konečném počtu zapsat lze, tedy např. $^1/_8$, (0,125), $^1/_{64}$ (0,015625), $^1/_{512}$ apod.

1.1.6 Aritmetické operace v nedesítkových soustavách

Pravidla aritmetiky jsou samozřejmě ve všech číselných soustavách stejná. Přesto je poměrně obtížné zvládnout v nedesítkových soustavách rutinně jiné operace než sčítání a odčítání. Právě tyto dvě operace jsou pro programátora v assembleru nejpotřebnější např. při výpočtech adres jistých zajímavých míst v paměti (často bývá k dispozici informace, že hledaný byte je o jistý počet bytů vzdálen od jisté výchozí adresy, obě hodnoty bývají nejčastěji známy v šestnáctkové soustavě). I když lze použít kalkulačku nebo převodu do desítkové soustavy a zpět, při častější práci si programátor zvykne provádět tyto výpočty zpaměti nebo s tužkou a papírem přímo v dané soustavě, protože všechny ostatní způsoby ho budou zdržovat.

Základní početní operace, tak jak se učí v prvních stupních základních škol, jsou založeny na pozičním zápisu hodnot a lze je tedy analogicky použít i v jiných než desítkových soustavách. Pouze je třeba mít na paměti, že základ není hodnota 10, ale hodnota jiná.

Nejjednodušší operací je sčítání. Při sečítání jednociferných čísel je možné pomoci si provedením operace v desítkové soustavě a výsledek zapsat v soustavě nové. Např.

$5_8 + 7_8 = 12_{10} = 14_8$
$2H + 0AH = 2_{10} + 10_{10} = 12_{10} = 0CH$

U sčítání víceciferných hodnot se použitím klasického způsobu tzv. „písemného sčítání“ (pod sebou) problém převede opět na operace s jednocifernými hodnotami, neboť se zpracovává každý řád samostatně s možným přenosem z nižšího do vyššího sousedního řádu. Podobně se provádí odčítání.

U násobení lze opět použít klasické „písemné násobení“, praktická potřeba této činnosti je ale velmi nízká. Při násobení jednociferných hodnot je obvyklé si pomáhat v soustavě desítkové, jinak by bylo nutné naučit se malou násobilku i v jiné než desítkové soustavě. Např.

$5_8 \times 7_8 = 5_{10} \times 7_{10} = 35_{10} = 43_8$

Samostatnou kapitolou je provádění aritmetických operací ve dvojkové soustavě. Používá se poziční zápis, takže operace ve dvojkové soustavě jsou vcelku jednoduché. Pro sčítání platí:

$0 + 0 = 0$
$0 + 1 = 1 + 0 = 1$
$1 + 1 = 10$

pro odčítání potom:

$0 - 0 = 0$
$0 - 1 = -1$
$1 - 0 = 1$
$1 - 1 = 0$

a pro násobení

$0 \cdot 0 = 0$
$0 \cdot 1 = 0$
$1 \cdot 0 = 0$
$1 \cdot 1 = 1$

Při výpočtech na papíře není žádnou komplikací, pokud výsledkem operace se dvěma *n*-ciferným čísly je číslo o větším počtu cifer *m* (*m* > *n*). Počítač ale pracuje s jistým pevným počtem cifer. Pokud tímto počtem bude např. 8, pak operace 1 + 1 bude v počítači součtem dvou osmiciferných čísel:

```
 00000001
+00000001
 00000010
```

V tomto případě je součtem dvou jednociferných čísel číslo dvojciferné, ovšem v počítači jsou jak operandy tak i výsledek osmiciferné. Jinak tomu bude např. v případě:

```
 01000001
+11000001
 10000010
```

kdy součtem dvou osmiciferných čísel je devíticiferná hodnota. V počítači je nutné při provádění každé operace předepsat, kam bude uložen výsledek, jedním z atributů místa v paměti (nebo registru procesoru) je jeho délka (počet bitů). Při práci s osmibitovými hodnotami a osmibitovými paměťovými místy se výsledek předchozí operace do požadovaného paměťového místa „nevleze“. Tato situace se označuje jako **přetečení** a je jedním z důležitých chybových stavů. S ohledem na přetečení lze základní tabulku pro sčítání zapsat ve tvaru:

$0 + 0 = 0$
$0 + 1 = 1 + 0 = 1$
$1 + 1 = 0 +$ *přenos*

Přenos (*carry*) znamená, že do vyššího řádu bude přenášena jednička, k přetečení (*overflow*) pak dochází v případě, že vyšší řád, do kterého by se měl uskutečnit přenos, neexistuje.

Při odčítání může docházet k podobnému jevu. Při výpočtu 0 – 1 je výsledek matematicky jasný (mínus jedna), počítač ale pracuje pouze se dvěma hodnotami (symboly), a to **0** a **1**. Proto je výsledkem této operace **0** plus tzv. **výpůjčka** (*borrow*), základní tabulka pro odčítání se potom zapisuje ve tvaru:

$0 – 0 = 0$
$0 – 1 = 0 +$ výpůjčka
$1 – 0 = 1$
$1 – 1 = 0$

Výpůjčka znamená, že pro zdárné dokončení operace bylo nutné si vypůjčit jedničku z vyššího řádu. Pokud tento vyšší řád existuje, je vše v pořádku, pokud neexistuje, opět dochází k přetečení. Tato situace úzce souvisí se způsobem, jak jsou v počítači reprezentována záporná čísla.

1.2 Reprezentace hodnot v paměti počítače

1.2.1 Záporná čísla

Početně nepředstavují záporná čísla žádný problém. Jistou komplikací dvojkové aritmetiky tak, jak je realizovaná v počítači, je fakt, že počítače jsou vnitřně schopné pracovat pouze se dvěma hodnotami (symboly) – **0** a **1**. K práci se zápornými čísly ve dvojkové soustavě jsou ovšem nutné symboly tři: **nula**, **jednička** a **znaménko mínus**. Proto je třeba i záporné hodnoty vhodně zakódovat pouze pomocí nul a jedniček. Jako příznak zápornosti čísla se používá hodnota 1 v nejvyšším bitu – při práci s hodnotami se znaménkem se prohlašuje nejvyšší bit za *znaménkový* a ostatní bity za *hodnotové*. Tak např. pro osmibitové hodnoty by nejvyšší sedmý bit (bity se označují zprava postupně nultý, první, … podle mocniny základu, kterou reprezentují) byl znaménkový a zbylých sedm hodnotových. Zavedení znaménkového bitu však samo o sobě není dostatečným řešením. Pokud by se totiž pro záporná čísla používal pouze znaménkový bit při nezměnění významu bitů hodnotových (např. hodnota 00000011 by znamenala $+3_{10}$ a hodnota 10000011 znamenala -3_{10}), vznikl by systém s řadou nedostatků. Jednak by v tomto systému existovaly dva způsoby vyjádření hodnoty nula (00000000 a 10000000, tedy kladná a záporná nula), jednak by se lišil způsob práce s kladnými a zápornými čísly. Je rozumné požadovat, aby ve zvoleném kódování platilo:

$$a - b = a + (-b) \tag{3}$$

Jinou, opět nevyhovující, možnost představuje tzv. **inverzní kód** (**jedničkový doplněk**), kde by se změna znaménka prováděla změnou hodnot všech bitů – např. 00000011 znamená +3 a 11111100 znamená –3. Opět se zde projevuje problém „dvojí nuly“ a ani požadavek (3) není splněn. Řešením je **dvojkový doplňkový kód**. Představu o tomto kódu pro osmibitové hodnoty lze nalézt v tabulce 1.2.

01111111	127
01111110	126
…	…
00000010	2
00000001	1
00000000	0
11111111	-1
11111110	-2
…	…
10000001	-127
10000000	-128

Tab. 1.2 – *Dvojkový doplňkový kód*

Změna znaménka (vynásobení hodnoty mínus jedničkou) se ve dvojkovém doplňkovém kódu provádí následovně:

1. zjistí se jedničkový doplněk (změní se hodnoty všech bitů)
2. k jedničkovému doplňku se přičte hodnota 1

Uvedený postup je univerzální, platí tedy pro změnu znaménka z kladného na záporné i opačně.

Pro práci se zápornými čísly se v počítačích používá právě dvojkového doplňku. Výhodou je, že aritmetické operace se provádějí u celých čísel vždy stejně a je jen na programátorovi, zda operandy chápe jako bezznaménkové (unsigned) nebo znaménkové (signed) hodnoty. Např.

```
 01000001
+10000011
---------
 11000100
```

lze chápat jako $65_{10} + 131_{10} = 196_{10}$ nebo jako $65_{10} + (-125_{10}) = -60_{10}$.

1.2.2 Hodnoty v pohyblivé řádové čárce

V paměti počítače jsou necelé hodnoty často ukládány v semilogaritmickém tvaru. Přesný formát a počty cifer pro mantisu i exponent mohou být různé. Určující může být formát podporovaný instrukcemi pro práci s čísly v pohyblivé řádové čárce (tzv. *hardwarový real* nebo *hardwarový float*) nebo způsob zvolený tvůrci překladače vyššího programovacího jazyka.

1.2.3 Kód BCD

Někdy je výhodné zobrazovat číselné hodnoty po jednotlivých desítkových cifrách. V tomto případě je ve dvojkové soustavě třeba zakódovat celkem deset symbolů (symboly 0 až 9). Pro zakódování deseti stavů jsou nutné čtyři bity ($\log_2 10 \cong 3.3219$, tedy tři bity by nedostačovaly). Kód BCD (*binary coded decimal*) spočívá v postupném zápisu desítkových cifer v dvojkové soustavě, kde hodnota každé desítkové cifry se zapíše pomocí právě čtyř cifer dvojkových. Např. hodnota **1259** bude v kódu BCD představována posloupností **0001 0010 0101 1001**. Je třeba důrazně připomenout, že se nejedná o zápis hodnoty ve dvojkové soustavě, i když postup zápisu připomíná převod mezi šestnáctkovou a dvojkovou soustavou. (Zmíněná desítková hodnota **1259** by se ve dvojkové soustavě zapsala jako **10011101011**.)

Nejmenší samostatně zpracovávanou a ukládanou skupinou bitů je byte, v jednom byte lze uložit dvě desítkové cifry v kódu BCD, to odpovídá hodnotám 0 až 99 desítkově. Pokud byte obsahuje hodnoty v kódu BCD, nelze s těmito hodnotami provádět aritmetické operace běžným způsobem. Je nutno kontrolovat, aby hodnota každé poloviny bytu (čtveřice bitů) nepřesáhla hodnotu 9_{10} (1001_2) a aby nedošlo k přenosu mezi těmito čtveřicemi (samozřejmě přenos „mimo byte“ je zajímavý rovněž). Např. sčítání hodnot $37_{10} + 15_{10} = 52_{10}$ vypadá v kódu BCD následovně:

0011 0111 + 0001 0101 = 0101 0010,

což je v rozporu s pravidly dvojkové aritmetiky. Procesory počítačů mají pro práci s hodnotami v kódu BCD buď speciální instrukce, nebo se používají instrukce běžné a následně se provede „korekční“ instrukce, která výsledek upraví. Kód BCD neobsahuje žádný obecný způsob, jak zobrazovat necelá a/nebo záporná čísla.

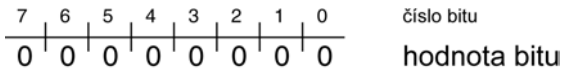
Kód BCD se často používá při přenosu desítkových hodnot (např. hodnoty naměřené měřícími přístroji, přenos po přístrojové sběrnici HP-IB), při jejich zobrazování (typicky sedmisegmentové LED nebo LCD zobrazovače) a také při některých numerických výpočtech, má-li být zaručena přesnost na určitý daný počet desítkových cifer a snadná vazba na vstupní a výstupní zařízení, která pracují v desítkové soustavě (typicky kapesní kalkulačky).

1.3 Logické a bitové operace

Každý bit může nabývat jedné ze dvou hodnot, lze jej tedy považovat nejen za dvojkovou cifru, ale také za logickou proměnnou. Při operacích s bity se logické operace využívají dosti často. Současné procesory poskytují mnoho instrukcí pro logické operace i pro práci s jednotlivými bity. Při práci s jed-

notlivými bity bývá zvykem bity číslovat podle jejich dvojkového logaritmu (exponentu mocniny dvou, kterou zastupují), takže např. první bit zprava se označuje jako bit nula, apod., viz obr. 1.1.

K základním logickým operacím patří negace (unární operace) a binární operace logického součinu, součtu a výhradního logického součtu. Pravdivostní tabulky uvedených operací jsou notoricky známé, pro připomenutí lze použít tab. 1.3.



Obr. 1.1 – *Číslování bitů v byte*

Operace **negace** je unární operací. Obvykle jsou procesory vybaveny instrukcí pro negaci celého operandu (např. byte), tzn. všech jeho bitů. Např. výsledkem negace bytu s hodnotou 01111001 je hodnota 10000110. Kromě negace všech bitů operandu (jedničkový doplněk) bývá k dispozici i instrukce pro získání dvojkového doplňku (změna znaménka, tzn. vynásobení operandu hodnotou -1).

A	B	$\overline{A}$	AB	$A+B$	$A \oplus B$
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Tab. 1.3 – Logické operace

Operace **logického součinu** má dva operandy. Aplikována bývá na odpovídající bity operandů; výsledek se skládá z bitů, z nichž každý je logickým součinem odpovídajících bitů obou operandů. Např. logický součin bytů 01111001 a 10100110 je hodnota 00100000. Pro oblast využití logického součinu (podobně jako u každé logické bitové operace) je zajímavé si uvědomit, že platí:

$$A \cdot A = A \tag{4}$$

$$A \cdot 1 = A \tag{5}$$

$$A \cdot 0 = 0 \tag{6}$$

Při potřebě **vynulovat** některé bity operandu a jiné ponechat beze změny se vzhledem k (5) a (6) uplatní právě logický součin. Pomocné hodnotě, která určuje, které bity mají být vynulovány a které nezměněny, se říká **maska**. V masce budou na hodnotu 0 nastaveny bity na pozicích, které mají být vynulovány, ostatní bity budou mít hodnotu 1. Např. při požadavku vynulovat bity č. 0 a 3 operandu o velikosti jeden byte bude vhodná hodnota masky 11110110. Uplatnění této masky na hodnotu 01111001 povede k výsledku:

01111001
11110110
01110000

Vztah (4) má také praktické využití. Pokud je potřeba otestovat jistou hodnotu (tzn. např. zjistit, zda je kladná nebo záporná, zda je nulová nebo nenulová apod.), lze použít operaci logického součinu hodnoty samy se sebou k nastavení registru příznaků (příznaky procesor nastavuje vždy po provedení aritmetické nebo logické operace, některé procesory i v jiných situacích, podrobnosti budou uvedeny dále). Tato operace je nedestruktivní, tzn. nedochází k žádné modifikaci výchozí hodnoty.

Operace **logického součtu** má rovněž dva operandy. Výsledek se skládá z bitů, z nichž každý je logickým součtem odpovídajících bitů obou operandů. Např. výsledkem logického součtu bytů 01010101 a 00110011 je hodnota 01110111. Podobně jako u logického součinu jsou pro použití operace logického součtu zajímavé následující operace:

$$A + A = A \tag{7}$$

$$A + 1 = 1 \tag{8}$$

$$A + 0 = A \tag{9}$$

Z (8) a (9) plyne, že logický součet se používá v případě, kdy požadujeme nastavení některých bitů operandu na hodnotu 1 a ponechání zbývajících bitů beze změny. Tak např. při požadavku nastavení bitu č. 4 v byte na hodnotu 1 při zachování původní hodnoty zbývajících bitů je třeba provést logický součet s maskou o hodnotě 00010000. Využití vztahu (7) je obdobné jako využití (4) u logického součinu.

Výhradní logický součet (*exclusive or*, **xor**) má rovněž dva operandy a výsledek se opět skládá z bitů, z nichž každý je výhradním logickým součtem odpovídajících bitů obou operandů. Např. výsledkem výhradního logického součtu bytů 01010011 a 00110101 je hodnota 01100110. Analogicky jako u předchozích binárních operací je využitelnost dána výsledky operací:

$$A \oplus A = 0 \tag{10}$$

$$A \oplus 1 = \overline{A} \tag{11}$$

$$A \oplus 0 = A \tag{12}$$

Z (11) a (12) plyne, že operace **xor** najde uplatnění v případech, kdy některé bity operandu mají být negovány (odpovídající bit v masce bude mít hodnotu 1) a ostatní nezměněny (odpovídající bit masky nastaven na nulu). S využitím (10) lze zase realizovat efektivní vynulování operandu u procesorů, které nemají instrukci pro nulování.

2 PRINCIP PRÁCE POČÍTAČE

Počítač lze považovat za programovatelný sekvenční automat. Z hlediska porozumění činnosti počítače jsou důležité dvě hlavní části počítače, a to procesor a paměť. procesor jako výkonná část umí provádět elementární operace zvané instrukce, každé instrukci odpovídá číselný kód. Podle principu Johna von Neumanna jsou program i jím zpracovávaná data uloženy ve stejné paměti. Program je v paměti uložen jako posloupnost kódů instrukcí (tedy posloupnost čísel), hovoří se o programu ve strojovém kódu. Procesor má uvnitř zvláštní paměťové buňky zvané registry. Jeden z reistrů má privilegované postavení a funkci – je v něm uložena adresa paměťového místa, ve kterém je uložen kód instrukce, která má být právě provedena; tento registr se obvykle nazývá čítač programu (PC, program counter) neb ukazatel instrukcí (IP, instruction pointer). Procesor jako konečný automat vykonává základní cyklus:

1. procesor provede čtení z paměti z adresy určené obsahem čítače programu
2. procesor modifikuje čítač programu, aby ukazoval na další instrukci, přečtená data chápe jako instrukci a odpovídající instrukci provede

Podle uvedeného popisu by procesor byl schopen vykonávat instrukce pouze sekvenčně, pro realizaci algoritmů je nutné provádět i ostatní řídící struktury. Proto existují instrukce, které na základě splnění nějaké podmínky nebo bezpodmínečně změní hodnotu čítače programu – snížení hodnoty povede k zopakování některé již dříve provedené činnosti, zvýšení znamená přeskočení některé činnosti.

Počítač by bylo možné programovat přímo vytvořením programu ve strojovém kódu. Je to ale pracné a nepraktické. Pokud každou instrukci pojmenujeme a symbolicky (textově) označíme její operandy, lze vcelku snadno realizovat program, který z textu obsahujícího po sobě zapsané názvy instrukcí a jejich operand strojový kód vygeneruje. Tomuto „programovacímu jazyku“, který není ničím jiným, než symbolickým zápisem strojového kódu, se říká assembler.

2.1 Registry

Uvnitř procesoru se nacházejí paměťová místa nazývaná registry. Registrů je omezený počet, ve srovnání s kapacitou operační paměti je objem dat, který mohou pojmout, zanedbatelný. Protože jsou registry fyzicky částí procesoru, je práce s nimi mnohem rychlejší a pro procesor z hlediska technické realizace snadnější – některé operace procesor nedokáže provádět s jinými operandy, než s registry.

Jednou z hlavních charakteristik procesoru, kterou je nutné znát při programování v assembleru, je počet, jména a vlastnosti registrů. Procesor I8086 obsahuje následující registry:

1. univerzální registry:
   - ax (accumulator)
   - bx (base)
   - cx (counter)
   - dx (data)

   Tyto registry jsou šestnáctibitové, pro případ potřeby práce s byty lze pracovat i s jejich osmibitovými polovinami, které mají v názvu jako druhé písmeno namísto **x** znak **l** (dolní polovina) a h (horní polovina). Lze tedy hovořit také o osmi osmibitových univerzálních registrech – al, ah, bl, bh, cl, ch, dl, dh.
2. Indexové registry
   - si (source index)

- di (destination index)

Tyto registry jsou šestnáctibitové a nelze je půlit.

3. Bázové registry
   - bp (base pointer)
   - sp (stack pointer)

   Také tyto registry jsou šestnáctibitové a nelze je půlit.
4. Segmentové registry
   - cs (code segment)
   - ds (data segment)
   - es (extra segment)
   - ss (stack segment)

   Registry jsou šestnáctibitové a mají zvláštní určení.
5. Instruction pointer (IP, čítač instrukcí)
   Registr je šestnáctibitový.
6. Registr příznaků (flags)
   Registr je šestnáctibitový.

2.1.1 Registr příznaků

Registr příznaků je šestnáctibitový, je ovšem organizován po bitech a všechny bity nejsou využity. Jednotlivé bity registru slouží jako příznaky a jsou nastavovány instrukcemi k signalizování různých stavů. Hodnota bitů registru příznaků slouží pro řízení instrukcí podmíněného skoku.

Struktura registru příznaků je následující:

a význam jednotlivých bitů je:

CF carry flag
: Nastaven na 1, když pole pro výsledek nestačí délkou k uložení výsledku, jinak 0.

PF parity flag
: Nastaven na 1, když ve výsledku je sudý počet bitů s hodnotou 1.

AF auxiliary carry flag
: Obdobně jako CF, ale signalizuje přenos mezi dolní a horní polovinou výsledku.

ZF zero flag
: Nastaven na 1, když je výsledek nulový.

SF sign flag
: Nastaven na 1, pokud je výsledek záporný.

TF trap flag
: Nastavuje se instrukcí, pokud je nastaven na 1, je procesor v režimu krokování.

IF interrupt enable flag
: Nastavuje se instrukcí, pokud je nastaven na 1, procesor přijímá žádosti o maskovatelná přerušení, jinak tyto žádosti ignoruje.

DF direction flag
: Nastavuje se instrukcí, řídí směr adresace řetězových instrukcí (0 znamená vzestupně).

OF overflow flag
: Nastaven na 1, když výsledek přetekl do znaménkového bitu (pouze pro signed operandy).

2.1.2 Segmentace paměti

Procesor 8086 generuje 20-tibitovou adresu. Procesor má pouze šestnáctibitové registry, fyzická adresa (20 bitů) se generuje jako součet hodnot dvou složek: **offset** + 16***segment**, každá složka má 16 bitů. Jako segment musí být použit jeden ze segmentových registrů.

Adresy se často udávají ve tvaru segment:offset, např. 123:45, předpokládá se, že hodnoty jsou zapsány v šestnáctkové soustavě. Uvedená hodnota 123:45 odpovídá fyzické adrese

123H*10H + 45H = 1230H + 45H = 1275H

Kombinace segment:offset je jednoznačná, ovšem pro danou fyzickou adresu nikoliv jediná možná. Při zvyšování offsetu negeneruje procesor automaticky přenos do segmentu. Vzhledem k existenci segmentových registrů a způsobu generování adres procesorem může být program v paměti rozdělen do čtyř příp. i vzájemně nesousedících segmentů, každý o velikosti 64 kB.

3 PROGRAM V ASSEMBLERU

3.1 Adresní módy (typy operandů)

Instrukce procesoru mohou mít operandy. V minulosti byla snaha vyrábět procesory s vyšším počtem operandů; důvodem bylo přiblížení se požadavkům vyšších programovacích jazyků. Např. chceme-li realizovat operace přiřazení zapsanou v jazyce C:

a = b + c

jedinou instrukcí, bylo by nutné, aby instrukce mohla mít tři operandy. Současné procesory obvykle umožňují používání maximálně dvou operandů. Výše uvedené přiřazení by tedy muselo být realizováno dvěma instrukcemi – operace by se musela rozepsat na dva kroky, každý o dvou operandech, např.:

a = b

b += c

Opět je použit zápis inspirovaný jazykem C.

Pokud instrukce používají operand (nebo dva operandy), je důležité vědět, jaký datový objekt může být operandem instrukce. U vyšších programovacích jazyků rozumíme pod pojmem druh datového objektu (datový typ) druh hodnoty (integer, real, boolean , …). V assembleru je důležité, kde je operand uložen (paměť, registr procesoru), a pokud se tento nachází v operační paměti (u proměnných je to obvyklé), je důležitý způsob určení jeho adresy. Určení samotného druhu hodnoty se potom omezuje na velikost operandu – byte, slovo, dvojslovo, …

Formát zápisu instrukce je následující:

```
instrukce  [operand1 [, operand2]]
```

Jak je vidět, oba operandy jsou obecně nepovinné. Konkrétní instrukce ale má vždy určitý počet operandů (žádný, jeden nebo dva). Má-li instrukce dav operandy, je operand1 operandem cílovým (určuje, kam se uloží výsledek) a operand2 operandem zdrojovým (tzn. provedením instrukce se nemění). Některé procesory (PDP, Motorola mohou mít opačné pořadí operandů a tedy cílovým může být druhý operand.

Následuje přehled možných typů operandů pro mikroprocesor I8086. K ilustraci je použita instrukce mov (move), která slouží k přesunu hodnoty (kopírování, přiřazení), tedy **`mov a,b`** by bylo možné v Pascalu zapsat jako **`a := b`**.

3.1.1 Registr

Registr může být zdrojovým i cílovým operandem. V zápisu instrukce se na místě operandu uvádí jméno registru. Je-li registr operandem cílovým, potom je odpovídající hodnota uložena do tohoto registru. Je-li registr operandem zdrojovým, vstupuje do instrukce hodnota, která je uložena v tomto registru, a hodnota registru se nemění.

Např. instrukce

```
mov   ax, bx
```

zkopíruje hodnotu, která se nachází v registru **bx**, do registru **ax**. Hodnota v registru **bx** se nezmění.

3.1.2 Konstanta (přímý operand)

Konstantu lze použít pouze jako zdrojový operand. V zápisu instrukce se uvádí hodnota, kterou v okamžiku překladu dokáže dosadit překladač (konstanta – desítková, dvojková, … – nebo konstantní výraz – výraz, který dokáže vyhodnotit překladač v čase překladu). Z hlediska strojového kódu jsou konstanty uloženy přímo v instrukci (v kódu programu, nikoliv v oblasti dat) – proto název přímý operand. Např.

```
mov ax, 0ABC1H
```

má význam

```
ax := $ABC1;          // pomocný pseudozápis  v Pascalu
```

3.1.3 Přímá adresa

Přímá adresa může být používána jako zdrojový i jako cílový operand. Pracuje se s paměťovým místem, jehož adresa je přímo uvedena jako operand instrukce. Hodnota adresy může být uvedena číselnou konstantou nebo (častěji) symbolickým vyjádřením hodnoty (obvykle pomocí návěští). I8086 používá adresu ve tvaru *segment:offset*. Segment může být vyjádřen segmentovým registrem nebo jménem segmentu. Není-li segment specifikován, implicitně se použije registr **ds**.

```
mov   dx, ss:40h
```

Do registru **dx** se uloží hodnota, která se nachází v paměti na adrese 40h relativně od místa, kam ukazuje **ss** registr. Hodnota tohoto paměťového místa se nezmění.

```
mov   ax, word ptr es:pos
```

Do registru **ax** se uloží hodnota, která se nachází v paměti na adrese, která je součtem hodnoty v registru **es** (bere se jako segment) a hodnoty, kterou má symbol **pos**.

Pozn.: Jak u operandu typu konstanta, tak u přímé adresy, se v textu programu objevuje na místě operandu konstantní výraz. Aby bylo jasné, jak má být hodnota tohoto výrazu použita, je nutné rozlišení obou případů na úrovni textu programu. Toto je řešeno v různých assemblerech různě např. prefixovými znaky. U I8086se používá klíčových slov určujících typ operandu a/nebo hranatých závorek. Blíže viz 3.2.7.

3.1.4 Nepřímá adresa

Může být zdrojovým i cílovým operandem. Pracuje se s obsahem paměťového místa, jehož adresa je obsahem určeného registru. Offset může být uložen v jednom z registrů **bx**, **bp**, **si** nebo **di**. Segment je udán pomocí segmentového registru nebo jména segmentu. Není-li segment uveden, použije se **ss**, je-li offset uložen v **bp** a **ds** ve všech ostatních případech.

```
mov   ax,[di]
```

Do registru ax se uloží obsah paměťového místa o adrese: segment určený obsahem registru **ds**, offset určený obsahem registru **di**.

```
mov   al, ss:[bx]
```

Pozn.:Podobně jako v předchozím případě, i zde je nutno určit, zda má instrukce pracovat s registrem nebo s paměťovým místem, jehož adresa je obsažena ve specifikovaném registru. V různým assemblerech je to řešeno různě, např. pomocí závorek nebo prefixu. U I8086 se používají hranaté závorku příp. doplněné klíčovými slovy pro určení typu operandu, blíže viz 3.2.7.

3.1.5 Bázovaná (bázová) adresa

Tento adresní mód je podobný předchozímu, opět lze operand použít jako zdrojový i cílový. Pracuje se s obsahem paměťového místa, jehož adresa se získá jako součet tří složek. Jednou ze složek je segment určený segmentovým registrem nebo jménem segmentu. Offset se určí jako součet zbývajících dvou složek: obsahu specifikovaného registru (jeden z registrů **bx** nebo **bp**) a konstanty nazývané posunutí (displacement). Pokud není segment uveden, bere se implicitně registr **ds** pro offset určený registrem **bx** a **ss** pro offset určený registrem **bp**. Operand lze uvádět v jednom ze tří významově ekvivalentních tvarů:

```
segment:posuv[registr]
segment:[registr+posuv]
segment:[registr].posuv
```

V konkrétním případě může instrukce využívající tento typ operandu vypadat takto:

```
mov   ax,es:2[bx]
mov   ax,es:[bx+2]
mov   ax,es:[bx].2
```

3.1.6 Indexovaná (indexová) adresa

Podobné jako 3.1.5, ovšem v určení offsetu jsou namísto registrů **bp** nebo **bx** použity registry **si** nebo **di**. Není-li uveden segment, implicitně se bere registr **ds**.

```
mov   ax,d s:2[si]
mov   ax,ds:[si+2]
mov   ax,ds:[si].2
```

3.1.7 Bázově indexovaná adresa

Jedná se o kombinaci dvou předcházejících adresních módů. Pracuje se s paměťovým místem, jehož offset je určen součtem obsahů dvou registrů a posunutí. Jeden z registrů musí být bázový (**bx** nebo **bp**) a druhý indexový (**si** nebo **di**). Pokud je posuv roven nule, nemusí se uvádět. Není-li segment uveden, pak se implicitně bere registr **ss** v případě, že je pro specifikaci offsetu použit registr **bp**, ve všech ostatních případech se implicitně použije **ds**. Jsou možné čtyři významově ekvivalentní tvary zápisu tohoto typu operandu:

```
segment:posuv[reg1][reg2]
segment:[reg1+reg2+posuv]
segment:[reg1+reg2].posuv
segment:[reg1]+[reg2]+posuv
```

V konkrétním případě může instrukce využívající tento typ operandu vypadat takto:

```
mov ax, es:4[bx][di]
mov ax, es:[bx+di+4]
mov ax, es:[bx+di].4
mov ax, es:[bx]+[di]+4
```

3.2 Struktura programu

Program v assembleru je složený z řádků, zápis jedné akce nesmí překračovat hranice řádků. Řádek může mít jednu z následujících podob (hranaté závorky signalizují nepovinný výskyt příslušné položky):

[*návěští***:**]	[*instrukce*]	[*operand1* [, *operand2*]]	[**;***komentář*]
[*návěští***:**]	[*direktiva*]	[*parametry*]	[**;***komentář*]
[*návěští***:**]	[*makro*]	[*parametry*]	[**;***komentář*]

Jak je vidět, všechna pole jsou nepovinná a proto i prázdný řádek je syntakticky správný. **Direktivy** jsou povely pro překladač, negenerují žádný strojový kód, řídí využívání paměti, segmentaci programu, umožňují definici dat, řízení podmíněného překladu apod. **Makro** je symbolicky označená část zdrojového textu, která se na příslušné místo zdrojového textu zařadí pouze zápisem svého jména; makra lze parametrizovat (blíže viz 3.6).

Formální úprava programu v zásadě není stanovena, je však nutné, aby jméno instrukce, direktivy nebo makra bylo od operandů/parametrů odděleno alespoň jednou mezerou nebo tabulátorem, operandy/parametry se navzájem oddělují čárkou. Je vhodné (není to však podmínkou) psát program tak, aby jednotlivá odpovídající pole byla pod sebou. Rovněž se doporučuje, aby každý řádek měl komentář. Toto doporučení se sice programátorům uvyklým na vyšší programovací jazyky může jevit jako podivné, má však svoje opodstatnění – je třeba si uvědomit, že text programu v assembleru má jen minimální samodokumentační schopnost.

Struktura zdrojového textu (pořadí a kombinace instrukcí, direktiv a maker) je ve většině aspektů z formálního hlediska libovolná. Makra a konstanty musejí být definovány před prvním použitím. Zdrojový soubor musí končit direktivou **END**.

Každý program musí mít **startovací bod** (**startovací adresu**). Tento označuje instrukci, která bude po spuštění programu provedena jako první (na adresu dané instrukce bude nastavena dvojice registrů **CS:IP** po zavedení programu do paměti). Startovací bod programu musí mít návěští a toto návěští musí být uvedeno jako parametr direktivy **END**. Pokud je program sestavován z více zdrojových souborů, potom se startovací adresa vyskytuje právě v jednom z nich, v ostatních souborech direktiva **END** parametr mít nesmí.

Při popisu programovacího jazyka bývá zvykem začínat **množinou přípustných znaků** (znaky, které se smějí objevit ve zdrojovém textu programu – s výjimkou znakových konstant, ve kterých se pochopitelně mohou vyskytovat všechny existující znaky). Assembler nemá z tohoto pohledu žádné speciální požadavky. Assemblery *nebývají case-sensitivní*, tzn. že nezáleží na používání velkých a malých písmen. Proto názvy instrukcí, registrů, direktiv, návěští a ostatní elementy programy mohou být psány libovolně malými i velkými písmeny (symboly **abc**, **ABC**, **Abc**, **ABc**, **AbC**, **aBC** jsou pro assembler identické). Assembler I8086 pracuje tak, že zdrojový text s výjimkou znakových konstant převádí na velká písmena a tento teprve zpracovává, tzn., že veškeré symboly (konstanty, návěští, …) obsahují pouze písmena velké abecedy. Pokud by tato vlastnost byla nežádoucí (např. při spojování modulů v assembleru s moduly v jazyce C), lze ji zakázat parametrem příkazového řádku (parametr **/ml** způsobí, že u symbolů se bude překladač chovat jako case-sensitivní, u klíčových slov – jména instrukcí, registrů – nikoliv). Množina přípustných znaků je tvořena:

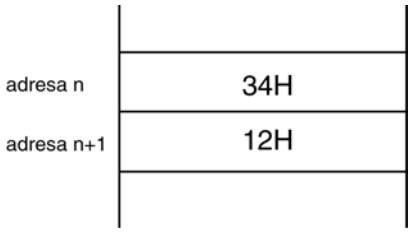
1. písmeny malé i velké anglické abecedy
2. číslicemi
3. speciálními znaky – v podstatě všechny ostatní zobrazitelné znaky. Některé znaky však mají speciální význam a některé lze použít pouze ve znakových konstantách.

3.2.1 Konstanty

Konstantou se rozumí pevně určená hodnota známá již v okamžiku překladu programu. Její hodnota se pochopitelně za běhu programu nemůže měnit. Konstanty jsou překladačem převáděny do vnitřního tvaru (do podoby, ve které jsou hodnoty uloženy v paměti počítače). Konstanty se používají obvykle jako přímé operandy nebo jako prvky výrazů.

a) celá čísla

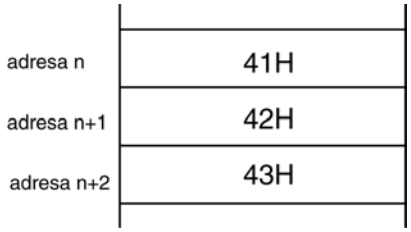
- Povoleny jsou číselné soustavy o základech 2, 8, 10, 16.
- Soustavu lze nastavit direktivou **.RADIX**
- Před prvním použitím direktivy **.RADIX** je implicitně nastavena desítková soustava.
- V jiné než nastavené soustavě lze hodnoty zapisovat pomocí postfixu:
  - dvojková konstanta – postfix **B** nebo **b**, např. **11101b**
  - osmičková konstanta – postfix **o**, **O**, **q** nebo **Q**, např. **123q**
  - desítková konstanta – postfix **D** nebo **d**, např. **20D**
  - šestnáctková konstanta – postfix **h** nebo **H,** např. **123H** (šestnáctková konstanta musí začínat číslicí, např. **0BABAH**)
- Uložení do paměti – u vícebytových konstant je důležité pořadí ukládaných bytů, u I8086 se ukládá postupně od nejnižších řádů na vzrůstající adresy. Např. hodnota `1234H` se uloží do paměti:



b) znakové konstanty

- Znaky se uzavřou mezi znaky " " (uvozovky) nebo ' ' (apostrof), omezovač zprava i zleva musí být pro jednu konstantu samozřejmě stejný).
- Konstanta může obsahovat všechny zobrazitelné znaky.
- Pokud má konstanta obsahovat i znak, který je použitý jako omezovač, uvádí se tento znak dvakrát.

  `"řekl: ""Ahoj! ""."` konstanta obsahující znaky **ř e k l dvojtečka mezera uvozovky A h o j vykřičník uvozovky tečka**
- Délka konstanty není omezená (použitelnost záleží na kontextu).
- Uložení do paměti – ukládá se ASCII kód znaku, každý znak na jeden byte, první znak na nejnižší adresu, ostatní vzestupně. Např. `"ABC"` se uloží do paměti:



3.2.2 Klíčová slova

V assembleru se za klíčová slova považují jména instrukcí, registrů a direktiv. Konkrétní výčet záleží na použitém procesoru a překladači.

3.2.3 Jména

Jména slouží k symbolickému označení různých objektů (proměnné, návěští, konstanty, makra, segmenty, skupiny, …). Pro vytváření jmen v assembleru I8086 platí následující pravidla:

- Jméno může obsahovat malá i velká písmena anglické abecedy, číslice a znaky **. _ ? $ @ % (**tečka, podtržítko, otazník, dolar, zavináč, procento). O vlivu velký a malých písmen byla zmínka v 3.3.
- Tečka může být použita pouze jako první znak.
- Prvním znakem jména nesmí být číslice.
- Jméno nesmí tvořit samotný znak **$** nebo **?**
- Délka jména je omezena na 255 znaků, významných je pouze prvních 31 znaků (počet významných znaků lze nastavit parametrem **/mv** příkazového řádku).

Je zakázáno používat jako jména pro vlastní objekty tzv. rezervovaná slova (klíčová slova) – jména registrů, instrukcí a direktiv.

3.2.4 Definice symbolických konstant

Symbolickou konstantou se rozumí symbolické označení konstantní hodnoty. Nejedná se tedy o proměnnou, ale o symbolicky zapsanou konstantu o hodnotě, která odpovídá popisu v 3.2.1. Assembler I8086 nabízí dva způsoby, jak symbolické konstanty definovat:

a) direktiva EQU

<jméno>* EQU *<hodnota>

Symbolu **jméno** takto přiřazenou hodnotu nelze dále v textu programu změnit. Hodnota nemusí být číselná, ale může se jednat o libovolný řetězec. Překladač během překladu provádí textovou náhradu – kdykoliv se ve zdrojovém textu programu vyskytne symbol **jméno**, nahradí jej definovanou hodnotou.

```
SEDM       EQU   7
INSTRUKCE  EQU   mov
OPERAND    EQU   [bp+20]
...
mov        ax, SEDM        ; totéž jako mov ax, 7
INSTRUKCE  ax, bx          ; totéž jako mov ax, bx
mov        OPERAND, ax     ; totéž jako mov [bp+20], ax
```

b) operátor =

Operátor je obdobný použití **EQU** s následujícími změnami:

- Přiřazovaná hodnota musí být číslo v rozsahu 0 až 65535.
- Hodnota symbolu může být v programu předefinovaná.

```
HODNOTA = 10
mov  al, HODNOTA           ; totéž jako mov al, 10
HODNOTA = 35
```

```
mov   dh, HODNOTA          ; totéž jako mov dh, 35
```

3.2.5 Výrazy

Výraz je kombinace operandů a operátorů. Operandem může být konstanta nebo symbolicky definovaná konstanta.

```
      K1 = 6
PR:   DB   2,3,4,5
      ...
      mov   ax, K1+2              ; totéž jako mov ax, 8
      mov   al, byte ptr PR+2     ; odkaz na paměť
      K1 = K1 + 1                 ; K1 má nyní hodnotu 7
      mov   si, K1                ; totéž jako mov si, 7
```

Kromě běžných aritmetických operátorů lze v různých assemblerech obvykle používat celou řadu dalších operátorů (např. relační, logické, bitové apod.). Přehled aritmetických operátorů pro assembler I8086 je v tab. 3.1, relační operátory jsou v tab. 3.2. Dále lze používat logické operátory NOT, AND, OR a XOR. Operace se provádějí po bitech, význam operátorů je zřejmý.

Operátor	Význam
+a	unární plus, nemění hodnotu operandu
-a	unární minus, vytváří dvojkový doplněk operandu
a + *b*	sčítání
a - *b*	odčítání
a * *b*	násobení
a/*b*	dělení
a MOD *b*	modulo, zbytek po celočíselném dělení *a*/*b*
a SHL *b*	posuv vlevo, posune *a* o *b* bitů vlevo (tzn. výsledkem je hodnota $a \cdot 2^b$)
a SHR *b*	posuv vpravo, posune *a* o *b* bitů vpravo (tzn. výsledkem je hodnota $a / 2^b$)

Tab. 3.1 – *Aritmetické operátory*

Relace	Význam
a EQ *b*	rovnost, $a = b$
a NE *b*	nerovnost, $a \neq b$
a LT *b*	menší než, $a < b$
a LE *b*	menší nebo rovno, $a \leq b$
a GT *b*	větší než, $a > b$
a GE *b*	větší nebo rovno, $a \geq b$

Tab. 3.2 – *Relační operátory*

Jako operandy pro sčítání a odčítání lze použít i jména proměnných, návěští a globální symboly. Jména proměnných a návěští označují místo v paměti a jejich absolutní hodnota v čase překladu není známa (známá je pouze hodnota relativní vztahující se k začátku segmentu, do kterého odkazují). Jejich použití je z toho důvodu omezeno: k jménům proměnných a návěštím lze pouze

přičíst nebo od nich odečíst konstantu (vznikne odkaz na paměťové místo více či méně vzdálené od počátku segmentu). Dále lze odečíst hodnoty dvou návěští a/nebo jmen proměnných. V tomto případě je výsledkem konstanta, která odpovídá vzdálenosti obou odkazovaných paměťových míst (v bytech), a platí samozřejmá podmínka, že oba operandy musí být definovány v tomtéž segmentu.

Výrazy jsou vyhodnocovány v čase překladu, jejich používání pouze usnadňuje tvorbu zdrojového textu programu. V žádném případě je nelze využít k naprogramování výpočtů prováděných za běhu programu.

3.2.6 Proměnné

Proměnná je místo v paměti pro ukládání dat. Definice proměnné znamená rezervaci paměťového místa požadované velikosti a pojmenování tohoto místa pro snadnější práci s touto proměnnou. Symbol použitý k pojmenování má význam adresy rezervovaného paměťového místa (v případě, že proměnná obsazuje několik bytů, má jméno hodnotu adresy prvního z nich, tedy adresu nejnižší). Definice proměnné se provádí zápisem:

jméno direktiva inicializace

Jméno je libovolné přípustné jméno, které bude používáno k odkazování příslušného paměťového místa. Přehled direktiv je v tab. 3.3.

Direktiva	Význam
DB	*define byte*, rezervuje jeden byte
DW	*define word*, rezervuje slovo (2 B)
DD	*define doubleword*, rezervuje dvojslovo (4 B)
DQ	*define quarteword*, rezervuje čtyřslovo (8 B)
DT	*define ten bytes*, rezervuje 10 B

Tab. 3.3 – *Direktivy pro definování proměnných*

Inicializace znamená nastavení počáteční hodnoty paměťového místa. Inicializace provádí překladač, děje se tedy v čase překladu a inicializátory mohou být pouze konstanty nebo výrazy, jejichž hodnota je známá v době překladu. K inicializaci lze použít:

- **konstantní číselnou hodnotu**

```
PROM    DB    20H
```

- **adresní výraz**

```
PROM    DB    20H
...
A       DW    PROM+20
```

- **řetězec znaků**

```
B       DB    "ABC "
```

- **žádná inicializace**

```
B       DB    ?
```

Použití tohoto způsobu má význam zejména u proměnných, které budou odkazovat do paměti ROM, některé vyšší programovací jazyky také rozlišují inicializovaná a neinicializovaná data. V běžných situacích překladače assembleru I8086 nastavují neinicializovaná data na nulu.

Na jednom řádku lze definovat a inicializovat více proměnných. V tomto případě všechny musejí být stejné velikosti a jméno proměnné se vztahuje k adrese první z nich.

```
C       DB      "ABC ",10,13
D       DW      0,0,10000, 0BABAH
```

Při potřebě rezervovat větší paměťové úseky (různé buffery apod.) se využívá opakovací parametr **DUP**. Zápis:

```
E       DB      256 DUP(0)
```

vyhradí 256 bytů a všechny inicializuje na nulu. **DUP** lze používat i pro komplikovanější inicializace:

```
F       DW      10 DUP(1,2,3)
```

vyhradí 30 slov (60 bytů) inicializovaných opakující se posloupností 1, 2, 3, 1, 2, 3, … S parametrem DUP lze vytvářet složitější konstrukce jeho vnořováním. Zápis:

```
G       DW      2 DUP(1, 3 DUP (7))
```

je ekvivalentní zápisu:

```
G       DW      1,7,7,7,1,7,7,7
```

Kromě definice proměnných se symbolické jméno určující adresu používá také pro označení adresy instrukce, která je cílem skoku, příp. pro označení vstupního bodu podprogramu. Takovému symbolickému jménu se říká **návěští**. Proměnné je možné definovat i s využitím návěští, v některých assemblerech je to jediná možnost (formálně se návěští od jména proměnné odlišuje znakem dvojtečka, kterým se návěští ukončuje). Překladače assembleru pro I8086 ovšem symbol definovaný jako jméno proměnné a jako návěští rozlišují a zacházení s nimi se v detailech mírně liší, odlišnosti jsou závislé na použitém způsobu segmentace (např. v segmentu datovém se návěští vůbec vyskytovat nemohou). Detailní popis těchto rozdílů není cílem tohoto textu.

3.2.7 Datové typy

V assembleru datové typy ve smyslu chápání vyšších programovacích jazyků neexistují. Pro assembler je důležitá znalost velikost operandu (délka v bytech) a jeho význam (operand, adresa operandu, adresa adresy operandu apod.). V některých situacích nedokáže překladač velikost a/nebo způsob chápání operandu jednoznačně automaticky určit. Typickým případem, kdy překladač nedokáže určit velikost operandu, je např. instrukce:

```
mov     ds:[si], ss:[bx]
```

Instrukce může požadovat přesun bytu nebo slova, z operandů to však nelze vyčíst bude hlášena chyba při překladu. Je tedy nutné překladači poskytnout doplňující informaci. K tomu slouží speciální operátory (někdy se označují jako operátory změny).

- **PTR**

  Operátor **PTR** (zkratka z pointer) explicitně určuje typ operandu, na který adresa použitá v instrukci ukazuje. Odkazovaným typem může být **BYTE**, **WORD**, **DWORD**, **QWORD**, **TBYTE**, význam je zřejmý. Výše uvedený nejednoznačný zápis by bylo možné upřesnit např. takto:

  ```
  mov     word ptr ds:[si], word ptr ss:[bx]
  ```

  Další možnost použití uvedeného operátoru je dokumentována zde:

  ```
  A:      DB 8,9
  ...
          mov ax, word ptr A    ; 8 do al, 9 do ah
  ```

Operátor **PTR** se používá také u instrukcí skoku a volání podprogramu, v těchto případech ve spojení s typy **NEAR** a **FAR**.

- **SEG**
  Operátor **SEG** slouží ke zjištění bázové adresy segmentu, ve které se nachází odkazovaný objekt (vrací segmentovou část adresy).

```
PROM:   DB    12
...
AAA:    DW    SEG PROM
...
        MOV AX, SEG PROM
```

- **OFFSET**
  Operátor OFFSET vrací relativní adresu (offset) určeného objektu.

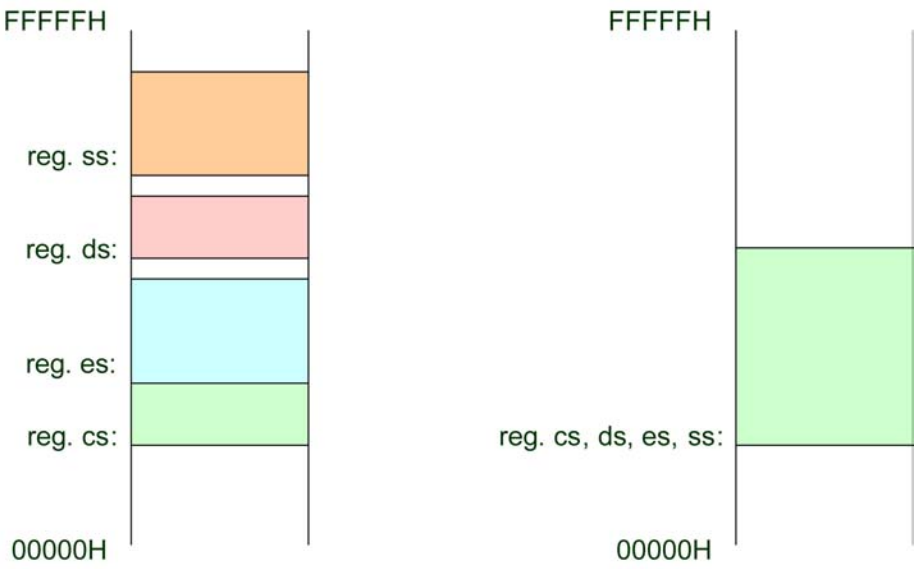
```
PROM:   DB    12
...
        MOV AX, OFFSET PROM
```

3.3 Segmentace programu

Obecně se jedná o problematiku **řízení ukládání** generovaného kódu a **řízení sestavení.** Ve vyšších jazycích je toto obvykle ovladatelné pouze nepřímo (např. tzv. paměťový model u 16-tibitových aplikací MS-DOS, popisovač COMMON ve FORTRANU apod.). Řeší se specifika využití paměti operačním systémem (sdílené knihovny a datové oblasti), rozdělení adresního prostoru paměti dané hw konfigurací (přístup do spec. oblastí paměti – videopaměť, ROM, buffery příd. karet apod.), řízení MMU (memory management unit). V jisté podobě jsou tyto prostředky přítomné v každém assembleru, jejich realizace je ale různá, neboť konkrétní situace závisí na procesoru a operačním systému (v některých assemblerech se jedná např. o členění programu do programových sekcí). Segmentace (a její obdoby v jiných assemblerech) se ovládá pomocí direktiv.

U assembleru I8086 se hovoří o tzv. segmentaci, pojem souvisí s MMU těchto procesorů, která používá segmentové registry k určení segmentu adresy. Segment je úsek paměti, který obsahuje kód nebo data a který je přístupný beze změny hodnoty segmentového registru, max. délka segmentu je 64 KB. Beze změny hodnoty segmentového registru lze pomocí offsetu (16 bitů) přistupovat k paměti o velikosti 64 KB (65.536 B). Mají-li všechny segmentové registry stejnou hodnotu, pak v jednom okamžiku velikost přístupného kódu i dat je 64 KB, jsou-li všechny registry různé (a rozdíly mezi jejich hodnotami jsou alespoň 1000H), pak v jednom okamžiku je pomocí offsetu adresovatelná paměť o velikosti 4*64 KB (256 KB = 262.144 B).

Segment má **segmentovou adresu** (adresa prvního byte segmentu, obvykle je offset 0, ale nemusí nutně být). Program a jeho data mohou být umístěny v jednom nebo mnoha segmentech. Zdrojem „nepohodlí“ je omezený počet segmentových registrů – bez ohledu na celkový počet segmentů, kterými je program tvořen, nelze v jednom okamžiku přistupovat k paměti ve více než čtyřech segmentech. situace je znázorněna na obr. 3.1.



Obr. 3.1 – *Přístupnost dat a kódu v paměti pro program uložený ve čtyřech segmentech (vlevo) a v jediném segmentu (vpravo)*

Základními prostředky pro řízení segmentace jsou direktivy **SEGMENT** a **ENDS**. Pomocí těchto direktiv lze řešit individuální požadavky na ukládání dat a kódu do segmentů (krátký vs. dlouhý program, data ve stejném nebo jiném segmentu než kód apod.) a problém přidělování paměti kódu a datům vzniklých z různých zdrojových modulů. Zjednodušený příklad programu zabírajícího jediný segment je následující:

```
jméno       SEGMENT parametry
            ; kód nebo data
START:      ; ...
            ; ...
jméno       ENDS
            END START
```

Platí zásady, že žádný kód ani data nesmějí být ukládány mimo segment, a že segmenty se nesmějí vnořovat.

Obecný tvar direktivy **SEGMENT** je:

```
jméno SEGMENT [uložení] [sestavení] [třída]
```

hranaté závorky opět signalizují nepovinnost parametru. **Jméno** je jméno segmentu (platí zásady pro jména). Parametr **uložení** určuje zarovnání začátku segmentu. Jsou možné následující hodnoty:

- **BYTE** – segment může začínat na libovolné adrese
- **WORD** – segment musí začínat na hranici slova (sudá adresa)
- **PARA** – segment musí začínat na hranici paragrafu (16 B), tedy na adrese dělitelné šestnácti (na hranici fyzického paměťového segmentu), default
- **PAGE** – segment musí začínat na hranici stránky, tj. na adrese dělitelné 100H

Parametr sestavení určuje, jak jsou do paměti ukládány segmenty shodného jména případně vzniklé z různých modulů. Možné hodnoty jsou:

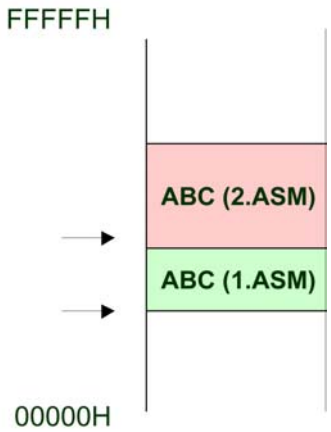
- **PRIVATE** (nebo vynechaná hodnota)
  V případě, že v různých modulech jsou definovány segmenty shodného jména, považuje se toto shodné jméno za náhodné, jedná se o dva různé segmenty, každý má svoji segmentovou adresu, každý má maximální délku 64 KB. V tomto případě je-li v souboru 1.asm obsažen kód:

```
ABC   SEGMENT
...
ABC   ENDS
```

  a v souboru 2.asm:

```
ABC   SEGMENT
...
ABC   ENDS
```

  bude sestavení programu následující:



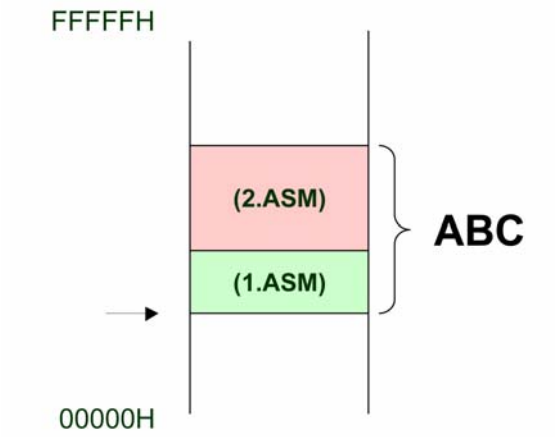
- **PUBLIC** nebo **MEMORY**
  V případě, že v různých modulech jsou definovány segmenty shodného jména, spojí se tyto segmenty do jediného segmentu s jedinou segmentovou adresou, offsety se určují od začátku tohoto výsledného spojeného segmentu, celková délka max 64 KB. V tomto případě je-li v souboru 1.asm obsažen kód:

```
ABC   SEGMENT PUBLIC
...
ABC   ENDS
```

  a v souboru 2.asm:

```
ABC   SEGMENT PUBLIC
...
ABC   ENDS
```

  bude sestavení programu následující:



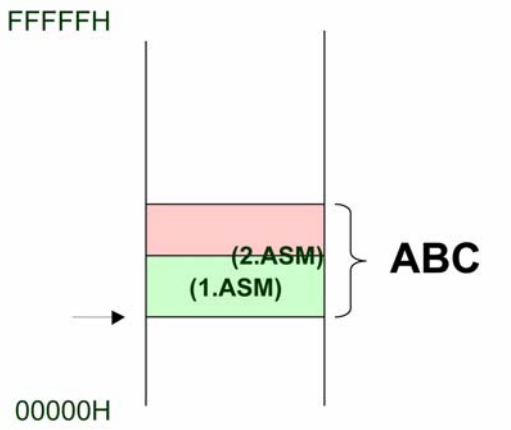
- **COMMON**
  V případě, že v různých modulech jsou definovány segmenty shodného jména, budou mít tyto segmenty shodnou počáteční adresu segmentu (budou „přes sebe“), délka je délkou nejdelšího z nich (tzn. nejdelší z nich může mít délku max. 64 KB). Použití je např. pro společná data (obdoba COMMON ve FORTRANU), sdílené knihovny apod. Způsob fungování je závislý na operačním systému, sdílení dat nebo knihoven může nastávat i mezi různými aplikacemi. . V tomto případě je-li v souboru 1.asm obsažen kód:

```
ABC   SEGMENT COMMON
...
ABC   ENDS
```

  a v souboru 2.asm:

```
ABC   SEGMENT COMMON
...
ABC   ENDS
```

  bude sestavení programu následující:



- **STACK**
  Rezervovaný segment pro zásobník, SS a SP by se měly automaticky inicializovat do tohoto segmentu.
- **AT výraz**
  Segment se umístí na adresu segmentu určenou výrazem. Tento segment nesmí definovat kód ani inicializovaná data (tento segment se při spuštění programu nezavádí do paměti). Obvyklé použití je definice návěští pro přístup do ROM

Na základě dosud uvedených informací lze uvést kostru pro použitelný program v assembleru:

```
DATA    SEGMENT
A:      DW    1,2,3
B:      db    'ahoj'
DATA    ENDS
   ;
CODE    SEGMENT
        ASSUME cs:CODE, ds:DATA, ss:ZASOB
START:  mov   ax, DATA
        mov   ds, ax      ; ds ukazuje do segmentu DATA
        ; ...
CODE    ENDS
   ;
ZASOB   SEGMENT STACK
   dw   100 dup(0)
ZASOB   ENDS
   ;
   END  START
```

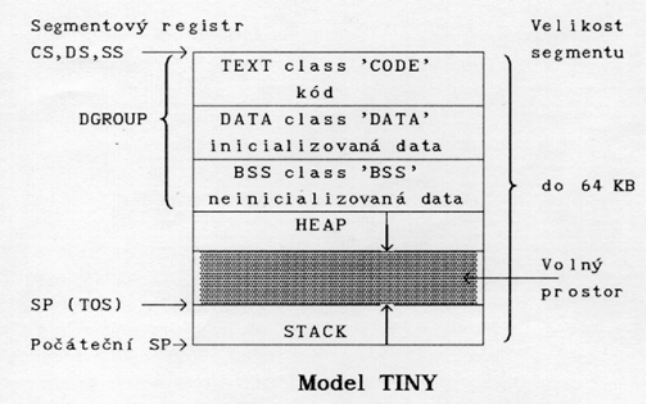
Za povšimnutí slouží direktiva **ASSUME**. V této direktivě se uvádí, jak budou v jednotlivých segmentech využity segmentové registry. Tuto informaci překladač využije ke kontrole, zda k přístupu k datům a při skocích a voláních podprogramů jsou vhodně používány segmentové registry a také k doplnění segmentu v případě jeho neuvedení. Další zajímavostí je dvojice instrukcí od návěští **START** – tato slouží k naplnění vhodné hodnoty do registru ds (vhodnou hodnotou je v tomto případě segmentová adresa segmentu **DATA**; protože **DATA** je jméno segmentu, představuje segmentovou adresu a není nutné používat operátor **SEG**).

3.3.1 Paměťový model

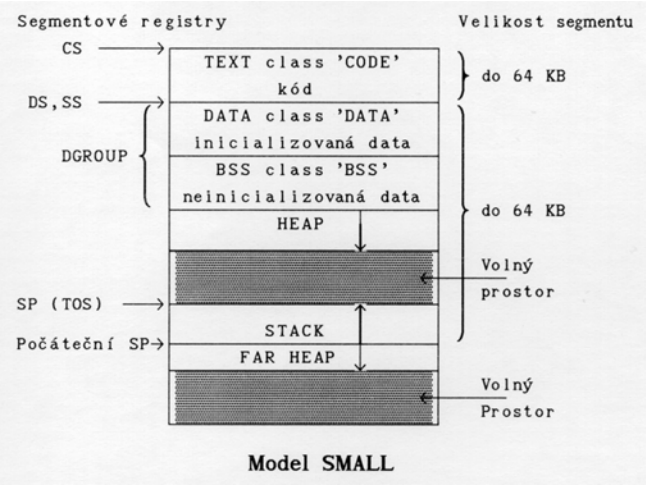
Pojem paměťový model se vztahuje k uspořádání segmentů programu. Tento pojem je spojen s šestnáctibitovými aplikacemi systému MS-DOS realizovanými v různých programovacích jazycích, tedy v assembleru i vyšších programovacích jazycích. Zavedení paměťových modelů vychází z představy, že programy podle své velikosti (kódu i dat) používají určité typické uspořádání segmentů. Obvykle se používají paměťové modely:

- **TINY:** Vše je v jednom segmentu, velikost celého programu včetně dat a zásobníku je omezena na 64 kB, všechny odkazy na data i kód jsou implicitně near (pouze offset). Uspořádání je na obr. 3.2.
- **SMALL**: Jeden segment je použit pro kód a druhý pro data a zásobník. Velikost kódu je tak omezena na 64 kB a velikost dat (včetně zásobníku) také na 64 kB. Implicitně se používají odkazy near. Model je znázorněn na obr. 3.3.
- **MEDIUM**: Pro kód se používají far odkazy, pro data near. Rozsah dat je omezen na 64 kB, kód je teoreticky omezen na 1 MB.Situace je na obr. 3.4.
- **COMPACT**: Opak modelu MEDIUM – far odkazy na data a near odkazy na kód. Obr. 3.5.
- **LARGE**: Používají se far odkazy na kód i data. velikost kódu i dat teoreticky omezena na 1 MB. Obr. 3.6.

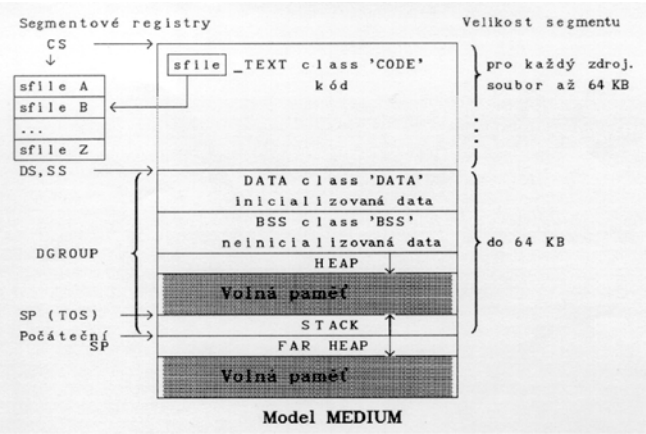
- **HUGE**: Do značné míry stejný jako LARGE. Většina překladačů vyšších programovacích jazyků nepřipouští, aby statická data přesáhla jeden segment (64 kB), v modelu HUGE toto omezení neplatí.



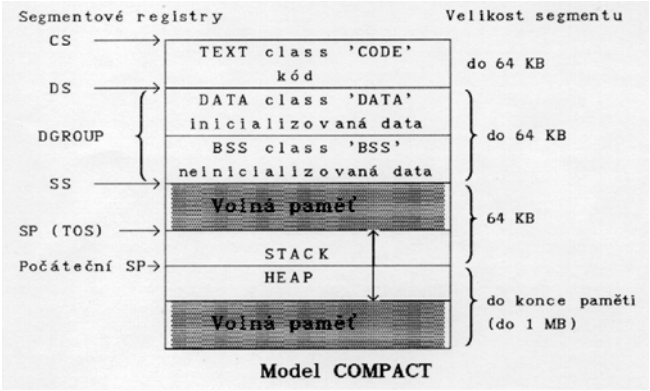
Obr. 3.2 – *Paměťový model TINY*



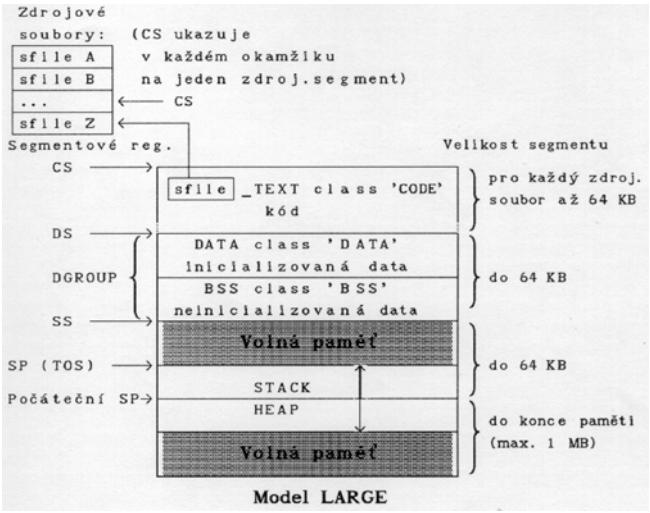
Obr. 3.3 – *Paměťový model SMALL*



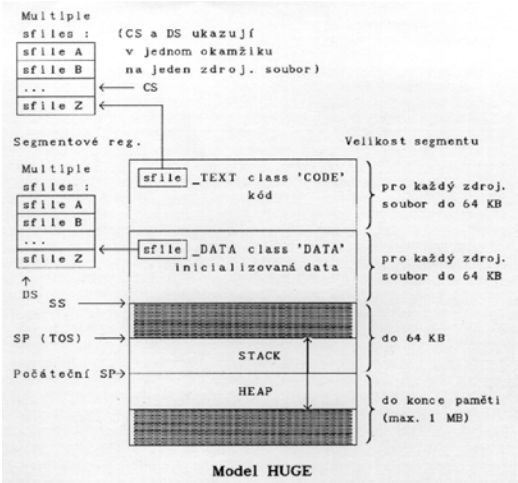
Obr. 3.4 – *Paměťový model MEDIUM*



Obr. 3.5 – *Paměťový model COMPACT*



Obr. 3.6 – *Paměťový model LARGE*



Obr. 3.7 – *Paměťový model HUGE*

3.3.2 Direktivy pro zjednodušené řízení segmentace

Pokud se programátor rozhodne používat některý z existujících datových modelů popsaných v předchozí kapitole, má k dispozici direktivy umožňující velmi pohodlné řízení segmentace programu.

První direktiva, která musí být použita, je direktiva **.MODEL**, parametrem direktivy je jméno paměťového modelu (TINY, SMALL, MEDIUM, COMPACT, LARGE, HUGE). Dále lze používat direktivy **.CODE**, **.DATA** a **.STACK**.

.CODE označuje začátek kódového segmentu programu

.DATA Zahajuje segment pro data. Je obvyklé, aby do tohoto segmentu ukazoval registr **ds**, tato inicializace ale není provedena automaticky a je nutné registr inicializovat ve vlastní režii např. instrukcemi:

```
mov   ax, @data  ;@data je aut. generované jméno segmentu
mov   ds, ax
```

.STACK *velikost*

Definice segmentu pro zásobník. Argument udává velikost zásobníku v bytech. Registry **ss** a **sp** jsou automaticky inicializovány do tohoto segmentu.

Následuje příklad kostry programu s využitím uvedených direktiv:

```
.MODEL   SMALL
.DATA
A        DW    1,2,3
B        db    'ahoj'
.CODE
START:   mov   ax, @data
         mov   ds, ax
         ; ...
.STACK   100H
END      START
```

3.4 Realizace základních řídících struktur

Assembler nemá žádnou podporu pro realizaci řídicích struktur. S ohledem na princip činnosti procesoru je program sekvencí,, neboť procesor vykonává instrukce postupně se vzrůstající adresou. K narušení sekvenčnosti provádění instrukcí je nutno používat **skoky**. Existují skoky:

- nepodmíněné (instrukce **JMP**)
- podmíněné (instrukce **Jxx** – mnoho různých instrukcí, viz dále)

Operandem instrukce skoku je vždy **adresa** (adresa instrukce, na kterou se má skočit), tato adresa může být vyjádřena buď přímo (to je málo obvyklé a nepohodlné) nebo častěji pomocí **návěští**.

Vyhodnocení jakékoliv podmínky, ať je již použita k řízení cyklu nebo k alternativě, se provádí instrukcí (např. porovnání), která nastaví příznaky v registru příznaků (flags). Dále se použije instrukce podmíněného skoku, která v případě, že je nastavena příslušná kombinace příznaků, provede skok na adresu, která je operandem instrukce podmíněného skoku. Instrukce podmíněného skoku u většiny assemblerů neumí přenést řízení příliš „daleko“. Toto omezení je způsobeno tím, že cíl skoku bývá v těchto instrukcích ve strojovém kódu uváděn relativně vůči místu použití instrukce a na relativní vzdálenost bývá vyhrazeno obvykle jen málo bitů (typicky je možná vzdálenost –128 až +127 bytů). Rejstřík instrukcí podmíněného skoku pro procesor I8086 je uvedený v tabulce 3.4.

Instrukce	význam	příznaky	operand
JA JNBE	> not(≤)	CF=0 ∧ ZF=0	*unsigned*
JAE JNB JNC	≥ not(<)	CF=0	*unsigned*
JB JNAE JC	< not(≥)	CF=1	*unsigned*
JBE JNA	≤ not(>)	CF=1 ∨ ZF=1	*unsigned*
JE JZ	=	ZF=1	-
JNE JNZ	≠	ZF=0	-
JG JNLE	> not(≤)	OF=SF ∧ ZF=0	*signed*
JGE JNL	≥ not(<)	OF=SF	*signed*
JL JNGE	< not(≥)	OF≠SF	*signed*
JLE JNG	≤ not(>)	OF≠SF ∨ ZF=1	*signed*
JS	záporné	SF=1	*signed*
JNS	nezáporné	SF=0	*signed*
JPO JNP	lichá parita	PF=0	-
JPE JP	sudá parita	PF=1	-
JNO	nebyl. pom. přenos	OF=0	-
JO	byl pom. přenos	OF=1	-
JCXZ	CX=0	-	-

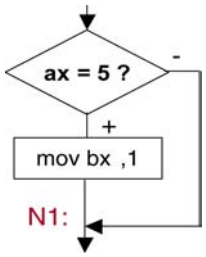
Tab. 3.4 – *Instrukce podmíněného skoku procesoru I8086*

3.4.1 Alternativa (neúplná)

Mějme za úkol naprogramovat obrat, který je dále uveden pomocí zápisu v jazyce C:

```
if(ax == 5) bx = 1;
```

což odpovídá vývojovému diagramu:

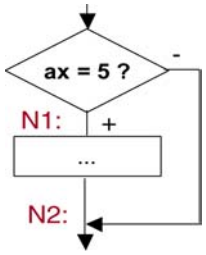


Odpovídající zápis v asembleru bude:

```
        cmp   ax,5  ; porovnání
        jne   N1    ; skok když ne
        mov   bx,1  ; kladná větev
N1:     ...         ; dále společně
```

Jak je vidět z textu programu i z vývojového diagramu, přeskakuje se kladná větev v případě, že podmínka splněna není.

Pro případ, kdy podmíněně prováděný úsek je delší než 127 B, je nutné použít i nepodmíněný skok tak, aby nebyla překročena maximální vzdálenost skoku pro instrukci skoku podmíněného.




```
cmp   ax,5        ; porovnání
      je    N1    ; skok když ano
      jmp   N2    ; když ne, tak přeskočit
N1:   ...         ; podmíněná část
      ...         ; (dlouhá)
N2:               ; společné pokračování
```

Pozn.: Pokud jsou k dokumentaci programů v assembleru použity vývojové diagramy, bývá zvykem pro větší přehlednost zapisovat návěští do příslušných míst diagramu.

3.4.2 Alternativa (úplná)

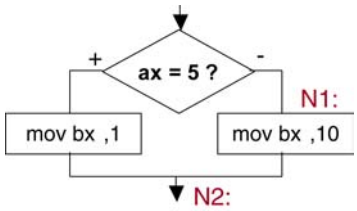
Jako v předchozím případě, máme za úkol naprogramovat následující obrat:

```
if(ax == 5) bx = 1; else bx = 10;
```

Odpovídající zápis v asembleru bude:

```
        cmp   ax,5  ; porovnání
        jne   N1    ; skok když různo
        mov   bx,1  ; kladná větev
        jmp   N2    ; přeskočení záporné větve
N1:     mov   bx,10 ; záporná větev
N2:               ; a dále společně
```

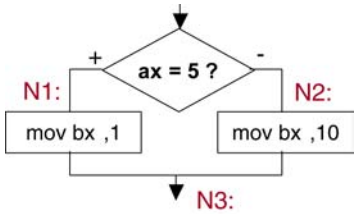
Vývojový diagram této alternativy je:



Pro případ, že by obě větve byly delší než připouští podmíněný skok, bylo by nutné provést úpravu:

```
        cmp   ax,5  ; porovnání
              je    N1    ; skok když ano
              jmp   N2    ; skok na zápornou větev
        N1:   ...         ; dlouhá kladná větev
              jmp   N3    ; přeskočení záporné větve
        N2:   ...         ; dlouhá záporná větev
        N3:   ...         ; společné pokračování
```

příslušný vývojový diagram:

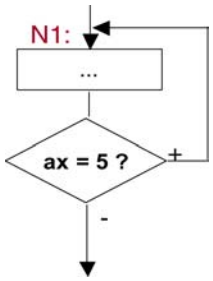


3.4.3 Cyklus s podmínkou na konci

Na rozdíl od vyšších programovacích jazyků je cyklus s podmínkou na konci v assembleru častěji používaný než cyklus s podmínkou na začátku. V jazyce C symbolicky zapsanou konstrukci:

```
do {...} while(ax == 5);
```

si lze představit ve formě vývojového diagramu:



a odpovídající zápis v asembleru bude:

```
N1:   ...        ; tělo cyklu
      ...
      cmp   ax,5 ; porovnání
      je    N1   ; když platí, tak znovu
      ...        ; pokračování
```

Praktickým příkladem pro demonstraci cyklu by mohl být program na realizaci celočíselného dělení pomocí odčítání. Některé procesory (zejména starší) nemají instrukci pro dělení a dělení je potom nutné realizovat pomocí opakovaného odčítání. (Procesor I8086 instrukci pro dělení má, jedná se pouze o příklad.) Použitý algoritmus lze zapsat v jazyce C:

```
delenec = 100;          // ax
delitel = 3;            // bx
podil = 0;              // cx
do    {
      podil++;
       delenec -= delitel;
} while (delenec >= 0);
podil--;                // korekce – bylo předěleno
delenec += delitel;     // zbytek do delenec
```

zápis v assembleru bude překvapivě jednoduchý:

```
mov   ax,100          ; dělenec
      mov   bx,3      ; dělitel
      xor   cx,cx     ; podíl
N1:   inc   cx        ; podíl + 1
      sub   ax,bx
      jns   N1        ; >= 0, znovu
      dec   cx        ; výsledek
      add   ax,bx     ; v ax je zbytek
```

U cyklů je často znám počet opakování, v tomto případě by řešený vzorový problém mohl být např.:

```
ax = 0; cx = 20;
do
  ax += cx--;
while(cx);
```

V těchto případech se u procesoru I8086 s výhodou používá jako čítač registr **cx**. Text fragmentu programu v assembleru bude:

```
        xor   ax,ax      ; nulování ax
        mov   cx,20
N1:     add   ax,cx
        loop  N1         ; dec cx, jne N1
```

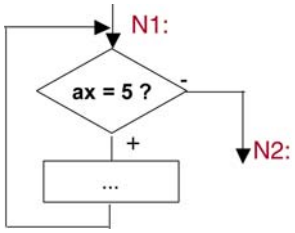
S výhodou se v těchto situacích používá instrukce **LOOP**. Tato instrukce sníží hodnotu registru **cx** o 1 a v případě, že obsah **cx** osud není nulový, provede skok na návěští, které je operandem instrukce. Za zmínku rovněž stojí použití instrukce **xor** pro nulování registru **ax**, Tento obrat je kratší a rychlejší než instrukce **mov ax,0**, neboť nula je zde přímý operand, musí být uložena v paměti a při provádění instrukce musí být tato (zde šestnáctibitová) nula čtena z paměti a přenesena do procesoru.

3.4.3 Cyklus s podmínkou na začátku

Opět zvolíme modelový problém, který bude v jazyce C zapsán:

```
while(ax == 5) {};
```

odpovídající vývojový diagram je:



a zápis programu v assembleru:

```
...
N1:   cmp   ax,5  ; porovnání
      jne   N2    ; když ne, tak konec
      ...         ; tělo cyklu
      jmp   N1    ; znovu na podmínku
N2:   ...         ; pokračování programu
```

Pokud je znám počet opakování, je realizace cyklu s podmínkou na začátku nepatrně obtížnější než u cyklu s podmínkou na konci. Problém v jazyce C bude např.:

```
for(ax=si=0;si<20;si++)
          ax += si;
```

jedná se tedy o úkol v registru **ax** vytvořit součet $\sum_{i=0}^{19} i$. V assembleru lze tento fragment zapsat:

```
      xor   ax,ax
      xor   si,si
N1:   cmp   si,20
      jge   N2
      add   ax,si
      inc   si
      jmp   N1
N2:   ...
```

3.5 Zásobník

Zásobník je datová struktura (LIFO), která slouží k dočasnému odkládání různých hodnot. Bez existence zásobníku by nebylo možné realizovat obsluhu přerušení jakožto reakci na asynchronní děje ani volání podprogramů. Každý procesor má speciální registr určený pro přístup k zásobníku (tzv. *ukazatel zásobníku*, *stack pointer*), u I8086 je to dvojice registrů **ss:sp**. Tato dvojice registrů ukazuje do paměti na tzv. *vrchol zásobníku* (*top of stack, TOS*). V případě požadavku na zápis do zásobníku (vložení hodnoty do zásobníku) se provedou následující operace:

1. hodnota registru **sp** se sníží o 2
2. hodnota, která má být zapsána, se uloží do paměti na adresu ss:sp

Podobně v případě čtení ze zásobníku (vyjmutí hodnoty ze zásobníku) se provede:

1. přečte se hodnota z adresy **ss:sp**
2. hodnota registru **sp** se zvýší o 2

Uvedené operace obvykle není nutné programovat ve dvou krocích, ale pro práci se zásobníkem existují speciální instrukce. Běžný název je **push** pro vložení do zásobníku a **pop** pro vyjmutí ze zásobníku. Např.

```
push  ax    ; vložení ax do zásobníku
push  bx    ; vložení bx do zásobníku
...
pop   bx    ; vyjmutí hodnoty ze zásobníku a její
            ; uložení do bx
pop   ax
```

Jak je vidět, ukazatel zásobníku ukazuje na TOS, kde je naposledy uložená hodnota. Většina procesorů nepřipouští lichou hodnotu ukazatele zásobníku, stejně je tomu u I8086, který umožňuje do zásobníku vkládat a z něj vyjímat pouze šestnáctibitové hodnoty.

Kromě již uvedené souvislosti s přerušením a podprogramy se zásobník používá v případě, kdy je nutné dočasně uložit nějakou hodnotu (zejména není-li předem znám počet takto ukládaných hodnot). V případě uložení většího počtu hodnot je třeba dát pozor na správné pořadí vyjímání hodnot ze zásobníku.

3.6 Podprogramy

Stejně jako ve vyšších programovacích jazycích lze i v assembleru používat podprogramy. Význam podprogramů je v assembleru stejný jako v ostatních programovacích jazycích – zkrácení textu programu, zpřehlednění textu programu, usnadnění modifikovatelnosti programu, usnadnění analýzy a realizace (montáže) programu, umožnění týmové práce apod.

Podprogram má vstupní bod – instrukci, kterou podprogram začíná, na tento vstupní bod je třeba přenést řízení. Po ukončení provádění podprogramu je třeba zase řízení vrátit na místo, odkud byl podprogram zavolán (resp. na instrukci bezprostředně následující za voláním podprogramu). K volání podprogramu slouží instrukce call, jejím operandem je adresa vstupního bodu podprogramu. Instrukce call uloží čítač instrukcí **ip** do zásobníku (v případě, že volaný podprogram je v jiném segmentu, uloží i registr **cs**) a provede skok na vstupní bod podprogramu. K návratu z podprogramu slouží instrukce **ret** (return), která obnoví ze zásobníku hodnotu registru **ip** a příp. i **cs**.

Podprogramy obvykle mají parametry. K předávání parametrů lze teoreticky použít:

1. smluvené paměťové místo (globální proměnnou), pak ale nelze používat rekurzivní volání podprogramu (přímo ani nepřímo),
2. registry – vhodné pouze při malém počtu a velikosti parametrů,
3. zásobník – univerzální metoda, je využívána většinou vyšších programovacích jazyků. tomto případě je ještě nutné učinit rozhodnutí, zda parametry ze zásobníku vyprázdní volaný podprogram nebo volající.

Podobné možnosti jsou pro předávání výsledku z podprogramu, pokud podprogram výsledek produkuje. Nejčastěji bývají používány registry (obvykle **ax**), ovšem není to pravidlem.

Nejjednodušší případ nastává, když je podprogram ve stejném segmentu, odkud je volán:

```
DATA  SEGMENT
prom1:     dw    2,3
DATA  ENDS

CODE  SEGMENT
      ASSUME     cs:CODE, ds:DATA, ss:ZASOB
CODE  ENDS

GLOBAL     pprg : FAR
      ;
START:     mov   ax,DATA
      mov   ds,ax
      mov   ax,ZASOB
      mov   ss,ax
      ;
      mov   ax,word ptr prom1
      mov   bx,offset prom1+2
      call  CD1:pprg        ; nebo jen call pprg
;
      mov   ax,4c00h
      int   21h
      ;
pprg: mov   si,bx
      add   ax,word ptr [si]
      ret
      ;
CODE  ENDS
      ;
ZASOB SEGMENT STACK
      dw    100 dup(0)
ZASOB ENDS
      ;
       END START
```

V příkladu je použit jednoduchý podprogram, který k registru ax přičítá obsah paměťového místa na adrese **ds:bx**, výsledek je vracen v registru **ax**. V tomto případě zcela postačuje vstupní bod podprogramu realizovat jako návěští. U některých assemblerů ani jiné možnosti realizace vstupního bodu podprogramu neexistují.

Pokud je podprogram umístěný v jiném segmentu, situace se mírně zkomplikuje:

```
DATA  SEGMENT
prom1:     dw    2,3
DATA  ENDS

CODE  SEGMENT
      ASSUME     cs:CODE, ds:DATA, ss:ZASOB
CODE  ENDS

GLOBAL     pprg : FAR
      ;
START:     mov   ax,DATA
      mov   ds,ax
      mov   ax,ZASOB
      mov   ss,ax
      ;
      mov   ax,word ptr prom1
      mov   bx,offset prom1+2
      call  CD1:pprg        ; nebo jen call pprg
      ;
      mov   ax,4c00h
      int   21h
      ;
CODE  ENDS
      ;
CD1   SEGMENT
      ASSUME cs:CD1, ss:ZASOB, ds:DATA
      ;
pprg  PROC FAR
      mov   si,bx
      add   ax,word ptr [si]
      ret
pprg  ENDP
      ;
CD1   ENDS
      ;
ZASOB SEGMENT STACK
      dw    100 dup(0)
ZASOB ENDS
      ;
      END START
```

Činnost podprogramu je stejná jako v předchozím případě. Teoreticky by bylo možné i v tomto případě vstupní bod realizovat pomocí návěští. Instrukce **call** zvládne přenést řízení na vzdálené návěští, problém nastane u návratu z podprogramu – je nutné použít instrukci **far return**, ta existuje, ale překladač nezná její název a neumí ji přeložit. Proto je nutné podprogram realizovat s využitím direktivy **PROC**, u které je parametrem specifikováno, že se jedná o podprogram volaný „far“, tedy

mezisegmentově. V takto deklarovaném podprogramu překladač na místě instrukce ret vygeneruje správný strojový kód pro návrat ze vzdáleného podprogramu.

Pokud se podprogram nachází nejen v jiném segmentu, ale i v jiném modulu, je možné řešení následující:

zdrojový soubor obsahující volání:

```
DATA  SEGMENT
prom1:     dw    2,3
DATA  ENDS

CODE  SEGMENT
      ASSUME     cs:CODE, ds:DATA, ss:ZASOB
CODE  ENDS
      ;
EXTRN pprg : FAR
      ;
START: mov ax,DATA
      mov   ds,ax
      mov   ax,ZASOB
      mov   ss,ax
      ;
      mov   ax,word ptr prom1
      mov   bx,offset prom1+2
      call  CD1:pprg        ; nebo jen call pprg
      ;
      mov   ax,4c00h
      int   21h
      ;
CODE  ENDS
      ;
ZASOB SEGMENT STACK
      dw    100 dup(0)
ZASOB ENDS
      ;
      END START
```

zdrojový soubor obsahující podprogram:

```
PUBLIC     pprg
      ;
CD1   SEGMENT
      ASSUME cs:CD1, ss:ZASOB, ds:DATA
      ;
pprg  PROC FAR
      mov   si,bx
      add   ax, word ptr [si]
      ret
pprg  ENDP
```

```
        ;
CD1     ENDS
        ;
        END
```

Direktiva **PUBLIC** způsobuje, že specifikovaný symbol **pprg** bude přístupný i mimo zdrojový soubor, ve kterém je definován (doplnění adresy k instrukci call v tomto případě provádí až sestavovací program – **linker**).

3.7 Počítadlo adres

Počítadlo adres (někdy též lokální počítadlo) reprezentuje v průběhu překladu aktuální adresu uvnitř překládaného segmentu. Má hodnotu offsetu, tzn. je 16-bitové. Význam je počet dosud uložených slabik ve výstupu překladače do daného segmentu (je-li použito jako operand instrukce, je v počítadle zahrnuta i délka této instrukce). V textu programu je k dispozici jako symbol označovaný znakem **$**. Hodnotu počítadlo lze využívat jako operand výrazů. Lze také modifikovat jeho hodnotu direktivami **ORG** (nastavení počítadla na konkrétní hodnotu) a **EVEN** (nastavení na nejbližší sudou hodnotu - je-li liché, provede se inkrementace, jinak nic).

```
MYSEG       SEGMENT PUBLIC
TXT1        db 'AHOJ',10,13,'$'
LTXT1 = $ - TXT1
...
MYSEG ENDS
```

Zde je vidět jedno z typických použití počítadla adres. Symbol LTXT1 je nastaven na aktuální počet znaků uložených v proměnné TXT1.

```
ORG   120H
...
ORG   $+4
...
ORG   $-10 ;asi špatně
```

Příklad ukazuje použití direktivy ORG. Zvýšení počítadla adres způsobí vynechání prostoru v generovaném kódu, snížení počítadla bude pravděpodobně chybou, neboť dojde k přepsání (tedy ztrátě) části již vygenerovaného kódu.

3.8 Podmíněný překlad

Tento termín se používá pro ovládání způsobu překladu zdrojového kódu v závislosti na výsledku nějakého testu (např. vygenerování úlohy pouze s některými vybranými funkcemi, úlohy s kódem používaným jen pro ladění apod.). Podmínky se testují při překladu, nikoliv za běhu úlohy! K testování a řízení překladu slouží tav. podmínkové direktivy. Jsou často používány v deklaracích maker. Existují dva druhy podmínkových direktiv:

- **direktivy pro podmíněný překlad** - podmíněné zařazování bloku textu do překladu
- **podmíněné chybové direktivy** - v závislosti na výsledku textu generují chybu při překladu

Direktivy pro podmíněný překlad se používají při potřebě generovat z jednoho zdrojového textu více verzí úlohy které se v detailech liší (např. verze pro ladění, verze se simulovaným okolím, verze se zablokovanými některými funkcemi, demoverze, verze pro různý počet uživatelů, zařízení, …). Obecný tvar podmíněně překládaného textu je následující:

```
IFx
  kód
ELSE
  kód
ENDIF
```

Následuje několik příkladů použití různých direktiv pro podmíněný překlad.

```
A = 3
IF A+1              splněno (nenulový výraz)
…
IF A+1 EQ 7         nesplněno (4 != 7)
…
IFE A               nesplněno (nenulový výraz)
…
IF1                 splněno v prvním průchodu překladu
…
IF2                 splněno v druhém průchodu překladu
…
IFDEF A             splněno (A je definovaný symbol)
…
IFB, IFIDN          pro použití v makrech
```

Podmíněné chybové direktivy se používají při ladění programu a při kontrole překladu (např. nepovolená kombinace nebo nevhodné hodnoty symbolů řídících podmíněný překlad). Pokud je splněna testovaná podmínka, je hlášena chyba při překladu. Následuje přehled chybových direktiv:

```
.ERR1               vždy v prvním průchodu
.ERR2               vždy ve druhém průchodu
.ERR                vždy
.ERRE A             když A=0
.ERRNZ A            když A!=0
.ERRNDEF A          když není definováno A
.ERRDEF A           když je definováno A
.ERRB, .ERRNB, .ERRIDN, .ERRDIF        pro použití v makrech
```

3.9 Makra

Velmi často se v programech opakovaně provádějí krátké posloupnosti instrukcí, které se od sebe buď vůbec neliší nebo se liší pouze v drobnostech. Tyto posloupnosti je možné definovat jako tzv. makra a v příslušných místech programu, kde má být některá posloupnost provedena, uvést pouze jméno definovaného makra s případnými parametry. Zápis jména makra do programu se nazývá *volání makra*. Překladač dosadí do místa volání makra instrukce, které jsou uvedeny v jeho definici.

Parametry makra slouží k modifikaci dosazovaných instrukcí podle konkrétních požadavků. Použití maker redukuje velikost zdrojového textu a často ho také zpřehledňuje. Makra je možno vkládat do sebe (vnořovat). Makro se někdy rovněž označuje termínem nepravý podprogram.Při volání makra provádí překladač textovou náhradu, tzv. rozvoj makra, potom překládá vzniklý text. Překladače assembleru, které umožňují práci s makry, se někdy nazývají makroassemblery.

Makra mají rozmanité použití:

- jako podprogram, pokud z nějakého důvodu není vhodný pravý podprogram volaný instrukcí **call**
- jako podprogram, který má být při překladu modifikován (ve spojení s podmíněným překladem nebo s využitím parametrizace)
- volání služeb operačního systému
- definování a inicializace datových struktur

Makra se často umísťují do zvláštních souborů (knihovny maker), které se v případě potřeby začleňují do zdrojového textu direktivou **INCLUDE**. V definicích maker se používají direktivy **MACRO**, **ENDM**, **LOCAL**, **EXITM**, **PURGE**.

Příkladem jednoduchých bezparametrických maker může být např.:

```
EXIT  MACRO                ; definice makra
      mov ax, 4C00H
      int 21H
ENDM
```

Je definováno makro EXIT, které zavolá službu operačního systému pro ukončení úlohy. Makro se zavolá uvedením svého jména. Makra většinou parametry mají, makro definované v předchozím příkladě by bylo možné modifikovat tak, aby jeho parametrem byl exit status úlohy (hodnota předávaná procesu, který končící proces spustil):

```
EXIT  MACRO  kod
      mov ah, 4CH
      mov al, kod
      int 21H
ENDM
```

Parametr makra je označen kod, jeho hodnotu je třeba specifikovat při volání makra. Jména parametrů makra jsou lokální uvnitř makra. Makra lze předefinovávat, platí vždy poslední definice makra téhož jména. Parametry maker jsou nepovinné, pomocí podmíněného překladu lze tato záležitost ošetřit:

```
EXIT  MACRO  kod
      mov ah, 4CH
IFB <kod>
      xor al, al
ELSE
      mov al, kod
ENDIF
      int 21H
ENDM
```

IFB je podmínková direktiva, která je splněna v případě, že její argument, který musí být formálním parametrem makra, je prázdný. (Tento argument se v direktivě **IFB** musí uzavřít do úhlových závorek.) Tato poslední verze makra **EXIT** se bude pro bezparametrické volání rozvíjet jako

```
mov   ah, 4CH
xor   al, al
int   21H
```

zatímco při zavolání s parametrem o hodnotě 7 (volání ve tvaru **EXIT 7**) se rozvine:

```
mov   ah, 4CH
mov   al, 7
int   21H
```

S makry úzce souvisejí tzv. *opakovací bloky*. To jsou bloky, které mají být do překládaného textu loženy několikrát za sebou, a to buď v identické nebo pozměněné podobě. Jednoduchým příkladem je úkol definovat tisíciprvkové pole šestnáctibitových hodnot a jednotlivá slova naplnit postupně hodnotami 1, 2, 3, …, 1000. Jednoduchým řešením je makro:

```
A = 1

REPT 1000
    DB A
    A = A+1
ENDM
```

Direktiva **REPT** opakuje text až po ukončovací direktivu **ENDM**, počet opakování je parametrem direktivy **REPT**. Jiné možnosti přináší opakovací blok využívající direktivy **IRP**/**ENDM**. V podprogramech i obsluze přerušení často potřebujeme uschovat do zásobníku vybrané registry, následující příklad definuje makro, které při volání SAVEREGS <ax, cx, si, bp> vygeneruje rozvoj:

```
push  ax
push  cx
push  si
push  bp
```

Definice makra je následující:

```
SAVEREGS MACRO A
         IRP B,<A>
         PUSH B
         ENDM
ENDM
```

Možnosti maker jsou velmi široké, umožňují např. generovat unikátní lokální návěští, mají řadu operátorů pro zvláštní zacházení s parametry apod. Filozoficky jsou možnosti maker u různých assemblerů podobné, konkrétní provedení se však liší. Detailní popis vlastností překladače pro II8086 není cílem tohoto textu.

4 PŘERUŠENÍ

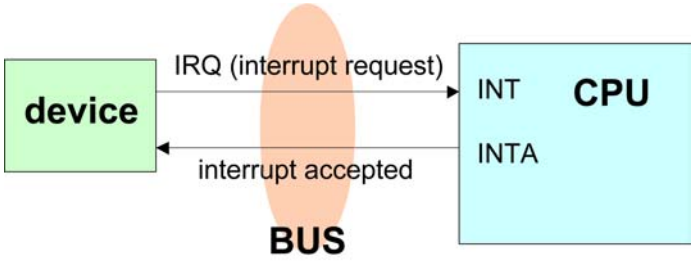
Přerušení slouží k zajištění reakce na asynchronní děje. Takovým dějem se rozumí událost, která nebyla v daném okamžiku běžícím programem očekávána. Takovou událostí může být např. pohyb myši, příjem dat po sériové lince, z počítačové sítě apod. Je jasné, že psát programy tak, aby periodicky testovaly (např. každou desátou instrukcí), zda nedošlo k pohybu myší, aby bylo možné překreslit její kurzor, by bylo absurdní. Každý aplikační program by takto musel neustále periodicky testovat všechny periferie, zda nevyžadují obsluhu.

Přerušení představuje efektivní způsob, jak tuto situaci vyřešit. Přerušení znamená (dočasné) přerušení běhu aktivního programu, provedení nutné obsluhy a návrat do přerušeného procesu. Přerušený proces se obvykle nedozví, že byl přerušen. Přerušení se používá také v operačních systémech pro zajištění multitaskingu (zdrojem přerušení může být časovač, periodicky přepíná mezi úlohami obv. se zohledněním priority). Přerušení je nutná podmínka pro real time applikace, multitaskové OS, správnou činnost driverů apod.

Všechny používané procesory mají subsystém přerušení, způsob realizace se může lišit:

- signál žádosti o přerušení je pin (nožička) procesoru
- obecně se přítomnost žádosti o přerušení testuje před čtením každé instrukce
- maskovatelné a nemaskovatelné přerušení (maskovatelné lze zakázat – instrukcí – nastavením bitu v registru příznaků)

Principiální zapojení pro jeden zdroj přerušení je následující:



Pro úspěšnou obsluhu je nutné identifikovat zařízení, které přerušení vyvolalo. K tomu je více možností, např.:

- jediný vstup přerušení na procesoru, rozhodnutí je na sw obsluze
- více vstupů přerušení na procesoru (pro každé zařízení jeden)
- zařízení se identifikuje v reakci na signál INTA (např. nastaví něco na datovou sběrnici)

U I8086 je tzv. vektorové přerušení. Vektor přerušení je datová struktura, která obsahuje adresy programového kódu obsluhy přerušení. Po přijetí přerušení procesorem vysílá zařízení (resp. obvod zvaný řadič přerušení) na sběrnici číselnou identifikaci přerušení (tzv. typ přerušení). Toto číslo udává na kolikáté položce se nachází adresa obsluhy tohoto přerušení. Vektor přerušení musí být předem naplněn, toto naplnění se děje při startu operačního systému a/nebo při zavedení odpovídajícího driveru zařízení. Obslužná rutina přerušení (ISR, Interrupt Service Routine) provede potřebné akce a provede návrat z přerušení.

Při přijetí přerušení procesor ukládá do zásobníku adresu instrukce, která by se vykonávala, kdyby bylo bývalo nedošlo k přerušení, a registr příznaků. Instrukce návratu z přerušení **iret** obnoví ze zásobníku uložené údaje, takže přerušená úloha pokračuje v místě a ve stavu, v jakém byla přerušena.

Přerušení lze vyvolat i softwarově – instrukcí, hovoří se o hw a sw přerušení. Parametrem instrukce vyvolávající přerušení je typ přerušení. Obsluha přerušení probíhá v obou případech stejně.

Softwarové přerušení se používá pro přístup k službám operačního systému a službám poskytovaných drivery. pomocí volání přerušení s vhodně naplněnými parametry v registrech má úloha v assembleru přístup ke službám OS a může tak požadovat V/V operace, manipulace s pamětí (alokace, dealokace, diskové operace, apod.).