Kapitola 1

Systémové programovanie

Software počítača môžeme rozdeliť na dva druhy programov: *systémové programy*, ktoré riadia operácie samotného počítača a *aplikačné programy*, ktoré riešia užívateľské úlohy.

Jednou z charakteristík, ktorou sa väčšina systémových programov odlišuje od aplikačných programov je *závislosť na počítači (procesore)*.

Aplikačný program sa hlavne sústreďuje na riešenie nejakého problému, pričom používa počítač ako prostriedok. Systémové programy majú podporovať operácie a použitie počítača samotného, nie jednotlivých aplikácií. Preto sa zvyčajne vzťahujú k štruktúre počítača, na ktorom bežia. Napr. asemblery prekladajú mnemonické inštrukcie do strojového kódu, takže formát inštrukcií, adresné módy atď. priamo ovplyvňujú design asemblera. Podobne kompilátory generujú strojový kód berúc do úvahy také hardwarové charakteristiky ako počet a použitie registrov a dostupné strojové inštrukcie. Operačné systémy riadia všetky prostriedky počítačového systému.

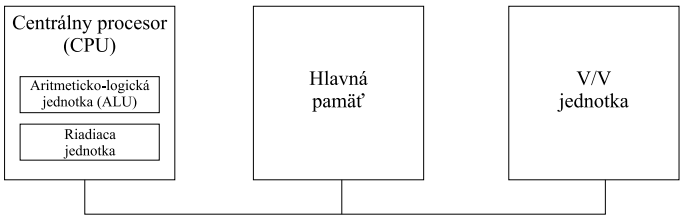
Na druhej strane sú isté aspekty systémového softwaru, ktoré priamo nesúvisia s typom systému, na ktorom pracujú. Napr. všeobecný design a logika asemblera je v základe rovnaká na všetkých procesoroch. Niektoré techniky optimalizácie kódu používané kompilátormi sú nezávislé od počítača. Podobne linkovanie nezávisle asemblerom prekladaných podprogramov zvyčajne nezávisí od použitého počítača.

Okrem operačného systému, ktorý je najzákladnejší systémový program, medzi systémové programy ďalej patria asemblery, kompilátory, makroprocesory, linkre, loadre, editory, debbugovacie systémy.

1.1 Štruktúra počítača

Zjednodušený model typického počítača - ako ho zaviedol v polovici 40-tych rokov 20. stor. matematik John von Neumann - sa skladá z nasledujúcich častí:

- *centrálny procesor (central processing unit)* – pozostáva z riadiacej jednotky, aritmeticko-logickej jednotky a internej pamäte (pracovných registrov – na uchovanie informácie, ktorá má byť rýchlo dostupná)
- *hlavná pamäť* – slúži na uchovávanie informácií a inštrukcií
- *vstupno-výstupná jednotka* – spája počítač s perifémymi zariadeniami



Niektoré registre slúžia na špeciálne účely, napr. *instruction register (IR)* na uloženie práve vykonávanej inštrukcie, *program counter (PC)* na uloženie adresy nasledujúcej inštrukcie, *stack pointer (SP)* na prístup k zásobníku, *stavové slovo procesora – processor status word (PSW)*, ktorý obsahuje informácie o stave súčasného procesu.

1.2 Reprezentácia dát

Počítače slúžia na spracovanie dát. Preto je dôležité vedieť, s akými typmi dát pracujú, aké operácie s nimi môžu vykonávať a ako sú dáta reprezentované v počítači.

Dátový typ je definovaný svojou:

- množinou hodnôt alebo prvkov
- množinou operácií na prvkoch

Na dátový typ sa možno pozerať 3 spôsobmi:

- ako na množinu abstraktných entít a príslušných operácií, ktoré nemajú vzťah k počítaču – *abstraktný dátový typ*
- ako na entity, ktoré definuje a používa nejaký programovací jazyk – *virtuálny dátový typ*
- ako na entity, ktoré sú fyzicky uložené a s ktorými narába hardware počítača – *fyzický dátový typ*

My sa teraz zaujímame o fyzické dátové typy. Všetky dáta sú reprezentované ako skupiny bitov. Vzťah medzi množinou bitov a prvkami typu sa nazýva *kód (kódovanie)*. Použitý kód určuje fyzickú reprezentáciu prvkov dátového typu.

1.2.1 Numerické dátové typy

Počítač pracuje s dvomi hlavnými typmi numerických dát: s celými číslami (integer data types) a číslami v pohyblivej rádovej čiarke (floating point data types).

Najprirodzenejší spôsob reprezentácie nezáporných celých čísel je reprezentácia v dvojkovej sústave.

Pre reprezentáciu záporných celých čísel sú možné tri prístupy:

- *sign and magnitude*: najľavejší bit určuje znamienko čísla (0=kladné, 1=záporné), ostatné bity dávajú absolútnu hodnotu čísla. Nevýhody: 1. dve reprezentácie čisla 0, 2. obvody pre sčítanie čísel sa nedajú použiť pre odčítanie.
- *1's complement (doplnok do 1)*: záporné číslo získame z kladného čísla (ktoré má v najľavejšom bite 0) negáciou po bitoch. Nevýhoda: dve reprezentácie čísla 0. V tomto prípade sa sčítací obvod dá použíť pre odčítanie (pripočíta sa číslo opačné a k výsledku sa pripočíta bit prenosu - "end around carry").
- *2's complement (doplnok do 2)*: záporné číslo vznikne ako negácia kladného čísla po bitoch zväčšená o 1. U tejto reprezentácie už nie sú dve rôzne reprezentácie nuly a sčítací obvod sa dá použiť na odčítanie (bit prenosu - carry bit - sa ignoruje).

Čísla v pohyblivej rádovej čiarke treba previesť do dvojkovej sústavy a zapísať v *normalizovanom tvare*: $(-1)^{znamienko} * mantisa * 2^{exponent}$, kde mantisa je jednoznačne určená v závislosti od použitého formátu (napr. pre VAX: pred desatinnou čiarkou je 0 a bezprostredne za ňou je číslica 1; IEEE štandard požaduje, aby to bolo číslo v tvare "1,zlomok").

V závislosti od požadovaného rozsahu a presnosti čísel potom jednotlivé formáty ukladajú mantisu a exponent do istého počtu bitov. Exponent sa zvyčajne zvýši o nejakú hodnotu N, aby mal kladnú

hodnotu a ukladá sa ako bezznamienkové číslo. Napr. v štandarde IEEE vo formáte "single precision" sa používa na uloženie reálneho čísla 32 bitov, z toho 1 bit je na znamienko, 8 bitov na zvýšený exponent (pôvodný exponent sa zvýši o 127, čiže pôvodný exponent mohol byť v rozsahu -127 až 128) a 23 bitov na zlomkovú časť mantisy. Čísla vo formáte "double precision" sa ukladajú do 64 bitov, z nich je na zvýšený exponent vyhradených 11 bitov (pôvodný exponent sa zvýši o 2047) a na zlomkovú časť mantisy sa používa 52 bitov.

1.3 Jazyk asemblera

Jazyk asemblera (asembler) je mnemonický jazyk, ktorý nahrádza inštrukcie strojového jazyka mnemonikami (symbolmi).

Na rozdiel od jazykov vyššej úrovne nie je asembler prenositeľný, lebo je úzko spätý so strojovým jazykom daného počítača, s jeho architektúrou.

Tým však programátor môže plne využiť všetky výhody architektonických čŕt počítača. Programy v jazyku asemblera majú minimálny čas vykonávania a efektívne využívajú systémové prostriedky.

1.3.1 Typy a formát inštrukcií

Základné informácie o programovaní v jazyku asemblera si uvedieme pre asembler počítača VAX.

VAX asembler používa 3 typy inštrukcií:

- *strojové inštrukcie (výkonné)* - tie, ktoré sú prekladané do strojového kódu a vykonávajú nejaké operácie
- *direktívy (nevýkonné)* - riadiace informácie pre prekladač (napr. na rezervovanie miesta pre premenné), začínajú bodkou
- *makroinštrukcie* - pseudoinštrukcie zavedené používateľom

Strojové inštrukcie môžeme ďalej rozdeliť na 4 základné skupiny:

- *prenos dát*
- *aritmetické a logické operácie*
- *riadenie programu* - rozhodovania a skoky
- *vstupno-výstupné inštrukcie*

Formát inštrukcie:

[Návestie :] KódOperácie [Operand(y)] [;Komentár]

Zvyčajne posledný operand je *cieľový* – teda ten, do ktorého sa uloží výsledok operácie.

VAX asembler používa 16 registrov veľkosti 32 bitov (= 4 bajty = dlhé slovo–*longword*):

R0 - R11 sú všeobecné registre (používané na ukladanie medzivýsledkov)
R12 = AP – Argument Pointer
R13 = FP – Frame Pointer
R14 = SP – Stack Pointer
R15 = PC – Program Counter

1.3.2 Adresné spôsoby

Adresný spôsob (adresný mód) je spôsob špecifikácie umiestnenia operandov. Až na niekoľko výnimiek môže byť ľubovoľný adresný mód použitý s ľubovoľnou inštrukciou. Skoro všetky adresné spôsoby môžu špecifikovať aj dáta aj cieľový operand.

Operand môže byť v registri, v pamäti alebo v samotnej inštrukcii.

Popíšeme si niekoľko základných adresných spôsobov a súčasne uvedieme, ako sa tieto adresné spôsoby prekladajú do strojového kódu.

1. **Registrový mód: Rn**

   Určuje, že operandom je všeobecný register.

   Napr. inštrukcia presunu dlhého slova (MOVL): MOVL R3, R7

   hovorí, že sa má obsah registra R3 presunúť (skopírovať) do registra R7.

   Preklad do strojového kódu: inštrukcia MOVL má kód D0 (v šestnástkovej sústave) - čiže zaberá 1 bajt. Operand v registrovom móde sa tiež prekladá do 1 bajtu, pričom v pravom polbajte je číslo registra (0-F) a v ľavom polbajte je 5 (určuje, že ide o registrový mód).

   Takže preklad uvedenej inštrukcie je: 57 53 D0 (adresy rastú smerom sprava doľava).

2. **Nepriamy registrový mód: (Rn)**

   V registri Rn je pamäťová adresa operandu (obsah registra Rn je smerník do pamäte na operand).

   Napr. MOVL (R3), R7

   hovorí, že sa má obsah pamäťového miesta veľkosti 4 bajty, ktorého adresa je v registri R3, presunúť do registra R7.

   Preklad do strojového kódu: inštrukcia MOVL má kód D0, nepriama registrová adresácia má v ľavom polbajte operandu číslo 6, pravý polbajt udáva číslo registra: 57 63 D0.

   Ak by sme použili operáciu presunu bajtu MOVB (R3), R7 – tak sa obsah pamäťového miesta veľkosti 1 bajt, ktorého adresa je v registri R3, presunie do najpravejšieho bajtu (najnižšie rády) registra R7.

3. **Autoinkrementový mód: (Rn)+**

   V registri Rn je adresa operandu (obsah registra Rn je smerník do pamäte na operand), po určení adresy sa obsah registra automaticky zvýši.

   Napr. MOVL (R3)+, R7

   hovorí, že sa má obsah pamäťového miesta veľkosti 4 bajty, ktorého adresa je v registri R3, presunúť do registra R7. Po určení adresy prvého operandu sa obsah registra R3 automaticky zvýši o 4 (pretože sme použili inštrukciu narábajúcu s dlhými slovami = 4 bajty) - čiže bude obsahovať adresu nasledujúceho dlhého slova.

   Tento adresný spôsob je významný pre prácu s poľami.

   Preklad do strojového kódu: v ľavom polbajte operandu je číslo 8, pravý polbajt udáva číslo registra: 57 83 D0.

4. **Autodekrementový mód: -(Rn)**

   Obsah registra Rn sa najprv automaticky zníži (o 1, 2 alebo 4 – podľa použitej inštrukcie) a až potom sa použije ako adresa operandu.

   Napr. MOVL -(R3), R7

   hovorí, že sa má obsah registra R3 znížiť o 4 a potom sa má obsah pamäťového miesta veľkosti 4 bajty (longword), ktorého adresa je v registri R3, presunúť do registra R7.

   Tento adresný spôsob možno použiť pre prácu s poľami v opačnom poradí.

   Preklad do strojového kódu: v ľavom polbajte operandu je číslo 7, pravý polbajt udáva číslo registra: 57 73 D0.

5. **Relatívny mód: adresa**

   Používa sa pre operandy uložené v pamäti, ktoré sú určené adresou (návestím).

   Napr. MOVL A, R10

   hovorí, že sa má obsah pamäťového miesta veľkosti 4 bajty s adresou A presunúť do registra R10.

   Preklad do strojového kódu: pri preklade do strojového kódu sa neuloží priamo adresa A, ale rozdiel medzi adresou A a obsahom PC registra (teda sa prekladá relatívne k PC registru). Na uloženie vypočítaného rozdielu sa vezme najmenší možný priestor (1, 2 alebo 4 bajty), do ktorého sa zmestí. Preklad operandu v relatívnom móde sa potom skladá z 2, 3 alebo 5 bajtov. Prvý bajt (informačný) obsahuje v pravom polbajte F (PC register) a v ľavom polbajte A, C alebo E podľa toho, či rozdiel vojde do 1, 2 alebo 4 bajtov. Nasledujúce 1, 2 alebo 4 bajty obsahujú rozdiel.

   Výhodou takéhoto prekladu je to, že je nezávislý od umiestnenia programu v pamäti. Spomínaný rozdiel je vlastne vzdialenosť pamäťového miesta, s ktorým inštrukcia narába, od tejto inštrukcie a táto vzdialenosť je rovnaká bez ohľadu na to, kde je program umiestnený. Ďalšou výhodou je, že rozdiel je možné vypočítať v čase prekladu z "logických" (relatívnych) adries – program adresujeme od 0 – a netreba poznať adresu, na ktorú bude program do pamäte zavedený.

   Adresa operandu sa vypočíta pri vykonávaní inštrukcie ako súčet obsahu PC registra (to už bude "fyzická" adresa) a rozdielu.

   Nech napr. (relatívna) adresa A je 0002 (hexadecimálne) a nech vyššieuvedená inštrukcia začína na adrese 0142. Na uloženie rozdielu budú potrebné 2 bajty. PC register bude v čase určovania rozdielu (a tiež v čase určovania adresy operandu) ukazovať na bajt nasledujúci za miestom na uloženie rozdielu, takže v našom príklade bude jeho hodnota 0146 (adresa 0142 = kód inštrukcie, 0143 = informačný bajt CF – rozdiel je v 2 bajtoch, 0144 a 0145 = rozdiel). Takže rozdiel je: 0002 - 0146 = FEBC.

   Preklad inštrukcie do strojového kódu: 5A FE BC CF D0

6. **Literál a priamy mód: #číslo alebo #výraz**

   Operandom je priamo hodnota uvedená v inštrukcii. Môže to byť celočíselná konštanta alebo konštanta v pohyblivej rádovej čiarke. Táto konštanta môže byť opísaná číslom alebo výrazom (zvyčajne sa používa len symbol).

   Literál a priamy mód vyzerajú rovnako, líšia sa však prekladom do strojového kódu (veľkosťou miesta na ich uloženie). Pod pojmom literál myslíme celočíselnú konštantu od 0 po 63 (max. 6 bitov) – pri preklade do strojového kódu sa používa len 1 bajt a doň sa priamo zapíše hodnota.

   Príklad: MOVL #25, R11 (do registra R11 vlož číslo 25)

   Preklad do strojového kódu: 5B 19 D0

   Rovnako sme mohli definovať konštantu a potom ju použiť v inštrukcii — preklad do strojového kódu je rovnaký:

   MAX=25 MOVL #MAX, R11

   Priamy mód zaberá 2, 3 alebo 5 bajtov – podľa veľkosti dát, s ktorými narába inštrukcia. Prvý bajt obsahuje vždy 8F a v nasledujúcich 1, 2 alebo 4 bajtoch je uložená konštanta.

   Príklad: MOVL #-2, R11 (do registra R11 vlož číslo -2)

   Preklad do strojového kódu: 5B FF FF FF FE 8F D0 (na konštantu sme použili 4 bajty, lebo inštrukcia MOVL narába s longwordami)

   Ak by sme mali inštrukciu MOVB #-2, R11, preklad by bol 5B FE 8F 90 (kód inštrukcie MOVB je 90, konštanta je uložená do 1 bajtu, pretože inštrukcia MOVB narába s bajtami).

7. **Nepriama adresácia s doplnkom: d(Rn)**

   Adresa operandu sa vypočíta tak, že sa k obsahu registra Rn pripočíta číslo (doplnok) uvedené pred zátvorkou (POZOR! Obsah registra Rn sa nezmení.).

   Doplnok môže byť výraz, ale zvyčajne sa používa len číslo (môže byť kladné aj záporné).

Príklad: MOVL 28(R5), R9

Obsah pamäťového miesta veľkosti 4 bajty s adresou, ktorú vypočítame ako súčet obsahu registra R5 a čísla 28, sa presunie do registra R9.

Preklad do strojového kódu: preklad operandu s doplnkom zaberá 2, 3 alebo 5 bajtov, v závislosti od veľkosti miesta potrebného na uloženie doplnku (prekladač sa snaží uložiť doplnok do najmenšieho miesta, do ktorého sa zmestí). Prvý bajt je informačný – v pravom polbajte obsahuje číslo registra, vzhľadom na ktorý sa adresuje, v ľavom polbajte je A, C alebo E, podľa toho, či doplnok vojde do 1, 2 alebo 4 bajtov. Nasledujúce 1, 2 alebo 4 bajty slúžia na uloženie doplnku v reprezentácii doplnok do 2.

Preklad uvedenej inštrukcie bude: 59 1C A5 D0 (informačný bajt je A5 - adresuje sa vzhľadom k registru R5 a doplnok sa uloží do 1 bajtu, doplnok $28_{10} = 1C_{16}$)

1.3.3 Štruktúra programu

Program v jazyku asemblera má nasledovnú štruktúru:

- deklarácia premenných a konštánt
- definície procedúr a makier
- hlavný program

Deklarácia premenných a konštánt

- *premenné*: pomocou direktívy .BLKx n sa vyhradí miesto pre 'n' bajtov, slov, dlhých slov – podľa toho, či sme namiesto 'x' použili B, W alebo L.

  Pre inicializáciu premenných (vyhradenie miesta spolu s priradením počiatočnej hodnoty) sa používajú direktívy .BYTE zoznam, .WORD zoznam alebo .LONG zoznam, kde 'zoznam' obsahuje hodnoty priradené do vyhradených pamäťových miest oddelené čiarkami.

  Napr. A: .BLKL 10 – vyhradí 10 dlhých slov (40 bajtov) a označí ich adresou A.

  B: .LONG 10,2 – na adrese B sa vyhradia dve dlhé slová, do prvého sa vloží hodnota 10, do druhého hodnota 2.
- *konštanty*: meno = výraz

1.3.4 Niektoré príkazy jazyka asemblera

V názve inštrukcie budeme používať písmená x, y na označenie rozmeru dát, s ktorými narábame (môže to byť B = bajt, W = word, L = longword).

Aritmetické operácie

CLRx	čo	čo:=0
INCx	čo	čo:=čo+1
DECx	čo	čo:=čo-1
MNEGx	čo, kam	aritmetická negácia (kam:=-čo)
ADDx2	čo , kam	kam:= kam + čo
ADDx3	čo1, čo2, kam	kam:= čo2 + čo1
SUBx2	čo, kam	kam:= kam - čo
SUBx3	čo1, čo2, kam	kam:= čo2 - čo1
MULx2	čo, kam	kam:= kam * čo
MULx3	čo1, čo2, kam	kam:= čo2 * čo1
DIVx2	čo, kam	kam:= kam div čo
DIVx3	čo1, čo2, kam	kam:= čo2 div čo1

Presuny a konverzie

MOVx	čo,kam	presun: kam:=čo
CVTxy	čo,kam	rozšírenie/skrátenie reprezentácie dát s doplnením znamienkového bitu
MOVZxy	čo,kam	rozšírenie/skrátenie reprezentácie dát s doplnením 0
MOVAx	náv,kam	presun adresy dát rozmeru x (kam:=adresa náv)

Skoky

Príkaz skoku môže spôsobiť, že do PC registra sa načíta nová adresa, teda sa nebude vykonávať nasledujúca inštrukcia. Väčšina príkazov skoku sú podmienené skoky, ktoré menia PC v závislosti od podmienky na dátach. VAX (a mnohé iné počítače) používa jednobitové príznaky nazývané *podmienkové bity (condition codes)* na zaznamenanie vlastností operandov inštrukcií – tieto príznaky sú súčasťou stavového slova procesora (PSW). Podmienené skoky testujú tieto príznaky, aby zistili, či treba meniť PC.

Podmienkové bity:

- N – Negative: N=1, ak výsledok operácie bol záporný
- Z – Zero: Z=1, ak výsledok operácie bol nula
- V – Overflow: V=1, ak nastalo pretečenie (výsledok presiahol vyhradený priestor)
- C – Carry: ak operácia mala prenos alebo záporný prenos v najľavejšom bite

Podmienkové bity sú automaticky nastavované vzhľadom na výsledok väčšiny operácií (napr. pri operácii sčítania sa nastavia podľa výsledku operácie, pri operácii prenosu sa nastavia podľa prenášaného čísla, pri operácii nulovania sa vždy nastaví N na 0, Z na 1, V na 0).

Niekedy je treba urobiť takéto nastavenie pre nejakú premennú alebo register v inom čase ako po vykonaní operácie alebo treba vyjadriť vzťah medzi dvoma porovnávanými hodnotami.
Na to slúžia dva príkazy:

TSTx	čo	test na nulu
CMPx	čo1, čo2	porovnanie operandov

Operácia TSTx nastaví Z a N bity podľa obsahu operandu (bity V a C vynuluje).

Operácia CMPx porovná operandy ako celé čísla v doplnku do 2 aj ako bezznamienkové čísla a podľa výsledku porovnania nastaví Z, N a C bity (vlastne robí porovnanie rozdielu čo1-čo2 s nulou – obsah operandov čo1 a čo2 sa pritom nezmení!):

Z=1, ak $čo1 = čo2$

N=1, ak $čo1 < čo2$ v doplnku do 2

C=1, ak $čo1 < čo2$ ako bezznamienkové čísla

Napr. ak $A = 6A_{16}$, $B = 94_{16}$, tak operácia CMPB A,B nastaví Z na 0 ($A \neq B$), N na 0 ($A - B \not< 0$, a teda $A \not< B$, lebo A je kladné a B je záporné - ako znamienkové čísla v doplnku do 2), C na 1 ($A < B$ bezznamienkovo) a V na 0.

Podmienené skoky

Na základe nastavenia podmienkových bitov podmienené skoky buď naplnia PC novou adresou (operand náv) alebo bude program pokračovať nasledujúcou inštrukciou.

BEQL	náv	ak rovné	– ak Z=1
BNEQ	náv	ak nerovné	– ak Z=0
BGTR	náv	ak väčšie	– ak N=0 a zároveň Z=0
BGEQ	náv	ak väčšie alebo rovné	– ak N=0
BLSS	náv	ak menšie	– ak N=1
BLEQ	náv	ak menšie alebo rovné	– ak N=1 alebo Z=1

Pri preklade do strojového kódu sa ukladá (podobne ako u relatívneho adresného módu) rozdiel medzi návestím náv a PC registrom – tu sa však tento rozdiel vždy ukladá do 1 bajtu (preklad celej inštrukcie podmieneného skoku tak zaberá 2 bajty) – takže je možné skákať len na návestia vzdialené 128 bajtov pred alebo 127 bajtov za aktuálnou pozíciou.

Nepodmienené skoky

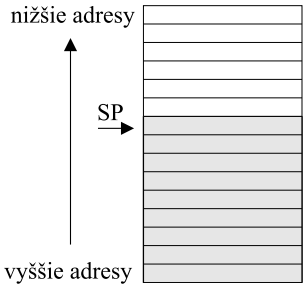
Nepodmienené skoky vždy zmenia obsah PC registra.

V preklade do strojového kódu sa u inštrukcií BRB a BRW ukladá opäť rozdiel medzi návestím a PC registrom, pri BRB sa uloží do 1 bajtu (celá inštukcia zaberá 2 bajty), pri BRW sa uloží do 2 bajtov (celá inštukcia zaberá 3 bajty). Pri inštrukcii JMP sa môže použiť na určenie cieľa ľubovoľný adresný mód (okrem priameho a literálu) – preklad potom závisí od použitého adresného módu.

Práca so zásobníkom

Zásobník je súvislé pole dátových miest používané na uloženie dočasných dát a informácie súvisiacej s volaním procedúr. Dátové položky sú do zásobníka vkladané a zo zásobníka vyberané metódou LIFO (last in first out). Na posledne vloženú položku zásobníka ukazuje premenná nazývaná *stack pointer - SP* (na VAXe je to register R14). Po zavedení programu do pamäte operačný systém automaticky vyhradí blok pamäte v adresnom priestore používateľa a nastaví SP.

Na VAXe zásobník rastie smerom k nižším adresám.



Inštrukcie pre prácu so zásobníkom:

PUSHL	čo	vlož do zásobníka dlhé slovo	≡ MOVL čo, -(SP)
POPL	kam	vyber zo zásobníka dlhé slovo	≡ MOVL (SP)+, kam
PUSHR	#^M<zoznam_registrov>	ulož do zásobníka registre z masky od registra s najvyšším číslom po najnižšie	
POPR	#^M<zoznam_registrov>	vyber zo zásobníka dlhé slová a daj do registrov z masky od registra s najnižším číslom po najvyššie	
PUSHAx	adr	ulož do zásobníka adresu adr	(x=B,W,L)

Poznámka: pre vloženie a vybratie dát iného rozmeru ako longword treba použiť inštrukcie MOVx čo, -(SP) a MOVx (SP)+, kam, kde x je rozmer dát, s ktorými narábame.

1.3.5 Procedúry

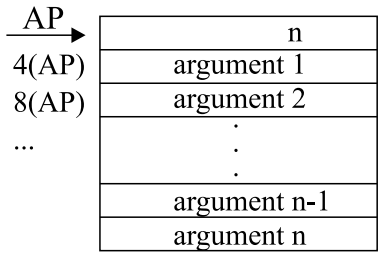
Procedúry umožňujú rozdeliť riešenie úlohy na časti, ktoré sú ľahšie modifikovateľné a odladiteľné.

VAX asembler poskytuje 2 volania procedúr:

- CALLG adresa_zoznamu_argumentov, meno
- CALLS počet_argumentov, meno

Oba spôsoby používajú *zoznam argumentv*, líšia sa však v tom, kde je tento zoznam uložený: v prípade CALLG (Call General) je to hocikde v pamäti (napr. naň vyhradíme miesto na začiatku programu - v časti deklarácí), u CALLS (Call Stack) sa uloží zoznam argumentov do zásobníka. V oboch prípadoch na zoznam argumentov ukazuje register R12 = AP (*Argument Pointer*).

Formát zoznamu argumentov:



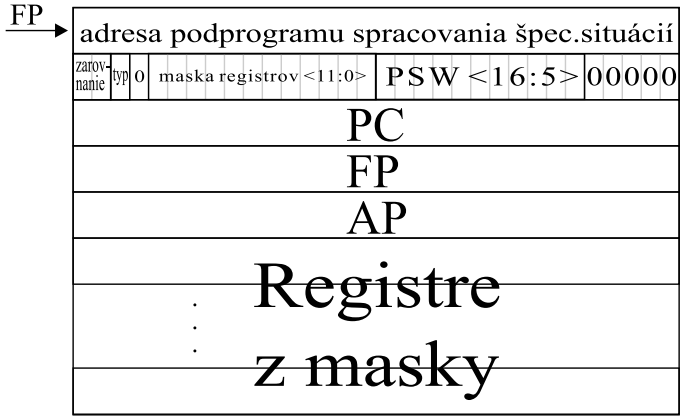
Formát procedúry:

(dátové definície, ak sú)
.ENTRY meno, maska_registrov
príkazy
RET

V maske registrov sú vymenované registre (R2 – R11), ktoré majú byť odložené do zásobníka pri vstupe do procedúry a po jej dokončení obnovené. Maska registrov má tvar: ^M<zoznam_registrov>. Registre AP, FP a PC budú uložené automaticky.

Ďalšou štandardizovanou dátovou štruktúrou pre volanie procedúry je *blok volania (call frame)* – slúži na uchovanie registrov a ďalšej informácie o stave procesu pri volaní procedúry. Je automaticky ukadaný do zásobníka pri oboch spôsoboch volania procedúry.

Formát bloku volania:



Najvrchnejšie dlhé slovo obsahuje adresu podprogramu spracovania špeciálnych situácií (condition handler address). Ak sa v procedúre objaví chyba, sem sa uloží adresa podprogramu, ktorý ju spracuje. Inak je tam uložená 0.

Ďalšie dlhé slovo obsahuje viaceré informácie:

- zarovnanie: 2 bity nadobúdajúce hodnotu 0 – 3, určujúce potrebné zarovnanie v momente volania procedúry (pretože blok volania musí byť vždy uložený od adresy, ktorá je násobkom 4).
- typ volania: 1 bit obsahujúci 0, ak sa vykonalo volanie CALLG, 1, ak sa vykonalo CALLS.
- maska registrov: 12 bitov pre registre z masky (R0 – R11)
- stavové slovo procesora: 16 bitov. Bity 0 – 4 stavového slova procesora sú vždy pred uložením vymazané. Procedúra môže tieto bity nejako nastavovať a indikovať pomocou nich nastatie nejakej podmienky. Po návrate do hlavného programu sa PSW obnoví a uvedené bity slúžia ako príznaky nejakých udalostí.

Na vrch bloku volania ukazuje register R14 = FP (*Frame Pointer*).

Volanie CALLG:

Ako príklad uvedieme procedúru SORT, ktorá má 2 vstupné argumenty: adresu triedeného poľa a dĺžku poľa.

Pri volaní CALLG musíme vyhradiť mieto pre argumenty v časti deklarácí.

```
POLE:       .BLKL 100
ARG_LIST:   .LONG 2             ;počet argumentov
            .ADDRESS POLE       ;direktíva na vyhradenie 4 bajtov a vloženie adresy
DLZ:        .BLKL 1             ;miesto pre dĺžku poľa
```

Volanie procedúry:

```
MOVL DLZKA, DLZ
CALLG ARG_LIST, SORT
```

(do AP registra sa dá adresa uvedená vo volaní ako 1. argument)

Nevýhodou tohto typu volania je to, že argumenty procedúry sú uložené v programe na inom mieste, než je volanie, čo môže spôsobovať neprehľadnosť pri čítaní programu a tiež to, že tento typ nie je vhodný pre rekurzívne procedúry.

Volanie CALLS:

Argumenty sa pred volaním ukladajú do zásobníka a ich počet sa odovzdá procedúre ako argument (hneď po zavolaní procedúry sa toto číslo automaticky zapíše do zásobníka – na vrch zoznamu argumentov – a naň sa nastaví AP register).

Volanie procedúry:

```
PUSHL DLZKA
PUSHAL POLE
CALLS #2, SORT
```

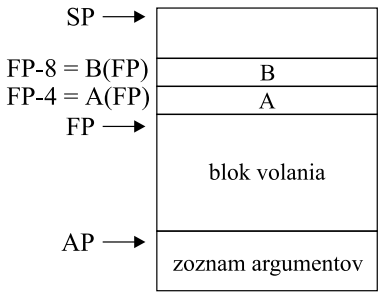
Lokálne premenné:

V zásobníku je možné uchovávať počas behu procedúry lokálne premenné a adresovať ich cez FP register.

Napr. chceme v procedúre PROC používať 2 lokálne premenné – A, B.

```
.ENTRY PROC, ^M<...>
A=-4
B=-8
SUBL2 #8, SP   ;urobiť miesto pre 2 dlhé slová na zásobníku
...
MOVL R0, A(FP)
MOVL R1, B(FP)
...
RET
```

Lokálne premenné adresujeme vzhľadom na FP register, a nie vzhľadom k SP, lebo SP sa môže meniť – zásobník sa môže používať aj na lokálne výpočty.



Návrat z procedúry:

Návrat z procedúry zabezpečuje inštrukcia RET, ktorá zo zásobníka vyberie blok volania (naplní registre PC, AP, FP a registre z masky pôvodnými hodnotami, naplní PSW uloženými údajmi), ak išlo o volanie CALLS vyberie aj zoznam argumentov a príslušne zmení SP (tým automaticky zruší alokáciu

miesta pre lokálne premenné).

Vrátenie hodnôt a príznakov:

Na VAXe je konvencia, že ak ide o funkciu, hodnota funkcie sa vráti v registri R0 (v prípade dát vyššej presnosti v R0 a R1).

Na uloženie príznakov (napr. či sa úloha úspešne vykonala, či nastali nejaké špeciálne situácie) sú dohodnuté dve miesta: register R0 alebo podmienkové bity – tie boli pred uložením do zásobníka, do bloku volania, vynulované. Procedúra ich môže nastaviť a po návrate do hlavného programu (po naplnení PSW) je možné ich otestovať.

Rekurzia:

Rekurzívne procedúry nemôžu mať dáta uložené staticky (.LONG, .BLKx, ...), ale všetky lokálne premenné musia byť uložené v zásobníku tak, že premenné z jedného volania nie sú modifikované ďalším rekurzívnym volaním.

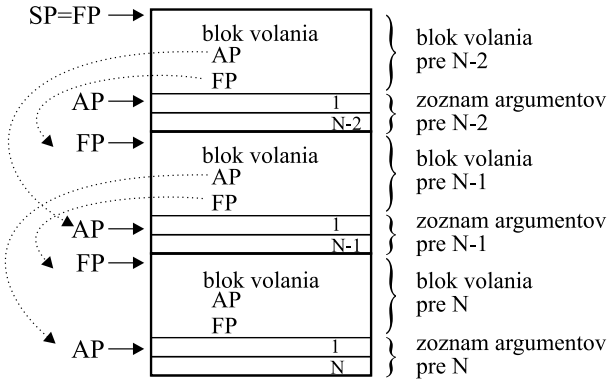
Ako príklad uvedieme výpočet faktoriálu: $N! = N.(N-1)!$, ak $N > 0$, $N! = 1$, ak $N = 0$.

```
        .ENTRY FAKT, ^M<R2>
        MOVL #1, R0             ;výsledok bude v R0 - je to funkcia
        MOVL 4(AP), R2          ;N daj do R2
        BEQL VON                ;končíme, keď N = 0
        SUBL3 #1, R2, -(SP)     ;do zásobníka daj N-1
        CALLS #1, FAKT          ;rekurzívne volanie procedúry
        MULL2 R2, R0            ;N.(N-1)!
VON:    RET
```

Hlavný program:

```
.BEGIN FAKTORIAL
    ⋮
PUSHL N
CALLS #1, FAKT
    ⋮
RET
.END FAKTORIAL
```

Po niekoľkonásobnom volaní rekurzívnej funkcie bude zásobník vyzerať takto:



1.4 Asembler - prekladač

Asembler je program, ktorý prekladá zdrojový program v jazyku asemblera do strojového kódu. Okrem strojového kódu vytvára ďalšie informácie, ktoré potom využije linker a loader (viď. kap. Linker a loader). Výsledkom prekladu je *objektový modul*.

Počas prekladania asembler priraďuje symbolickým výrazom ich numerické hodnoty a adresy. Na

určenie týchto hodnôt používa premennú LC = *Location counter*, ktorá funguje počas prekladu tak, ako PC za behu programu. Asembler vždy zvyšuje hodnotu LC o dĺžku inštrukcie, takže LC vždy obsahuje adresu nasledujúcej inštrukcie.

Podľa počtu prechodov cez zdrojový text rozlišujeme asemblery:

- dvojprechodové
- jednoprechodové

Dvojprechodový asembler

1. prechod: jeho úlohou je prejsť vstupný text, priradiť miesto každej inštrukcii a tým definovať hodnoty návestí. Vytvára *tabuľku symbolov*, do ktorej zapíše všetky nájdené symbolické mená spolu s ich hodnotami alebo adresami a prípadne ďalšou informáciou (premenná lokálna, globálna, externá).

Postup pri vytváraní tabuľky symbolov je nasledovný: na začiatku prvého prechodu sa nastaví LC na 0. Postupne asembler číta riadky zdrojového textu, ak riadok obsahuje návestie, zapíše ho do tabuľky symbolov spolu s aktuálnou hodnotou LC. Ak v tabuľke symbolov už symbol s rovnakým názvom existuje, vypíše chybu „Viacnásobne definovaný symbol“. LC zvýši o dĺžku inštrukcie a opakuje uvedený postup, až kým nepríde na koniec programu.

Na zistenie dĺžky inštrukcie a tiež overenie platnosti inštrukcie je potrebné prehľadať *tabuľku kódov inštrukcií* – obsahuje meno inštrukcie, jej ekvivalent v strojovom kóde, prípadne informáciu o formáte a dĺžke inštrukcie.

Prvý aj druhý prechod asemblera môžu ako vstup používať zdrojový program, ale je výhodnejšie, ak prvý prechod vytvorí upravený zdrojový program, ktorý sa potom stane vstupom pre druhý prechod. Upravený program obsahuje zdrojové riadky spolu s ich adresou, indikátormi chyby, môžu tu byť uložené aj smerníky do tabuľky kódov inštrukcií (pre kód inštrukcie) a tabuľky symbolov (pre každý použitý symbol), aby nebolo nutné opätovné prehľadávanie týchto tabuliek v druhom prechode.

2. prechod: druhýkrát sa prechádza vstupný (príp. upravený) text a robí sa preklad do strojového kódu. Ak sa v inštrukcii vyskytne symbol, dosadí sa jeho numerická hodnota alebo adresa z tabuľky symbolov.

Jednoprechodový asembler

Pri jednoprechodovom asembleri sa číta zdrojový text iba raz a v tomto jednom prechode sa vyrába tabuľka symbolov aj prekladá do strojového kódu. K problémom dochádza pri priradení numerických hodnôt symbolom (návestiam), ktoré sa v programe definujú neskôr, ako sa použijú. Tento problém možno riešiť tak, že sa vytvorí linkovaný zoznam nedefinovaných návestí. Po ukončení čítania vstupného textu sa len doplnia hodnoty návestí na miesta označené uvedeným zoznamom.

1.5 Makrá, makroprocesory

Makro je pomenovaná skupina inštrukcií, ktoré sa vložia do kódu na mieste, kde sa makro použije (volá).

Definícia makra môže byť daná programátorom v programe, v ktorom sa používa alebo môže byť v knižnici makier, ktorá je prístupná jednému alebo viacerým používateľom.

Proces nahradenia výskytu mena makra – *volania makra* – príslušnými príkazmi, sa nazýva *rozvoj makra* (macro expansion). Rozvoj makra nemusí byť pri každom volaní rovnaký, lebo v makre je možné použiť aj parametre.

V porovnaní s procedúrami je použitie makier nevýhodnejšie z hľadiska dĺžky výsledného kódu (lebo každé volanie makra vedie k vloženiu jeho tela na miesto volania, kým procedúry potrebujú v pamäti len jednu kópiu svojho kódu), ale je výhodnejšie z časového hľadiska (pri volaní procedúry vznikajú časové straty na vytvorenie prepojenia medzi programovými modulmi – napr. uloženie bloku volania, ktoré pri makrách nie sú).

Definícia makra:

```
.MACRO meno [zoznam_parametrov]
```

telo makra (inštrukcie, direktívy, volania alebo definície makier)

.ENDM [meno]

Parametre makra sú oddelené čiarkami, medzerami alebo tabulátormi. Môžu mať zadanú *implicitnú hodnotu*, ktorá sa dosadí za parameter, ak pri volaní makra nebude daná hodnota tohto parametra. Implicitná hodnota je zadaná tak, že v definícii makra za menom parametra nasleduje rovnítko a hodnota parametra.

Volanie makra:

meno [hodnoty_parametrov]

Príklad: makro na výmenu obsahu dvoch premenných

```
.MACRO VYMEN P1,P2,P3=POM
MOVL V1,V3
MOVL V2,V1
MOVL V3,V2
.ENDM VYMEN
```

Volanie: VYMEN R2,R7 má rozvoj:

```
MOVL R2,POM
MOVL R7,R2
MOVL POM,R7
```

Volanie: VYMEN R2,R7,R11 má rozvoj:

```
MOVL R2,R11
MOVL R7,R2
MOVL R11,R7
```

čiže, ak bola zadaná hodnota parametra, má prednosť pred implicitnou hodnotou danou v definícii makra.

Hodnoty parametrov makra môžu byť zadané dvoma spôsobmi:

- *pozične*: hodnoty pre parametre sú uvedené v takom poradí, ako sú parametre v definícii makra. Ak niektorý parameter (nie posledný) má implicitnú hodnotu, ktorú vo volaní chceme ponechať, musí vo volaní makra byť zadaná „prázdna hodnota“ – tj. idú za sebou 2 čiarky.
- *nepozične*: hodnoty nemusia byť zadané v presnom poradí podľa definície, ale sú zadávané v tvare parameter = hodnota_parametra

Príklad:

.MACRO XX MENO,DLZ=#20,DOL=#0,HOR=#19,TYP=L

má 5 parametrov, z ktorých 4 majú implicitnú hodnotu. Ak chceme volať toto makro a zadať parameter MENO s hodnotou POLE a HOR s hodnotou #100, tak v prípade pozične syntaxe použijeme volanie:

XX POLE,,,#100

a pri nepozičnej syntaxi:

XX MENO=POLE,HOR=#100 alebo aj XX HOR=#100,MENO=POLE (nemusíme dodržať poradie parametrov, ako bolo v definícii)

Možná je aj kombinácia pozičného a nepozičného volania, ale vždy musí začať pozičné a potom nepozičné (za ním už pozičnú syntax nemožno použiť):

XX POLE,HOR=#100

Spájanie parametrov:

Niekedy je užitočné spojiť parameter s textom – používa sa na to operátor spojenia: apostrof.

Napr.

```
.MACRO SUM A,B,C,TYPE
```

```
ADD'TYPE'3 A,B,C
.ENDM
```

má pri volaní SUM R3,R4,R7,W rozvoj ADDW3 R3,R4,R7.

Ak treba spojiť 2 parametre, medzi ne dáme dva apostrofy:

```
.MACRO XXX A,B,C,OP,TYPE
OP"TYPE'3 A,B,C
.ENDM
```

má pri volaní XXX R3,R4,R7,MUL,B rozvoj MULB3 R3,R4,R7.

Návestia v makrách:

Majme makro na výpočet absolútnej hodnoty premennej:

```
        .MACRO ABS CO,KAM
        MOVL CO,KAM
        BGEQ KON
        MNEGL KAM,KAM
KON:    .ENDM ABS
```

Ak sa toto makro volá len raz, nevznikne problém, ale ak bude volané viackrát, v programe sa vyskytne viacero návestí KON.

Jedno možné riešenie je pridať parameter makra NAV:

```
        .MACRO ABS CO,KAM,NAV
        MOVL CO,KAM
        BGEQ NAV
        MNEGL KAM,KAM
NAV:    .ENDM ABS
```

takže ak pri rôznych volaniach budeme zadávať rôzne hodnoty parametra NAV, konflikt nevznikne – je to ale pre používateľa veľmi „nepohodlné“ riešenie.

Druhou možnosťou je špecifikovať v zozname parametrov makra *lokálne návestia* (majú tvar n$ a platia v úseku medzi dvoma užívateľsky definovanými návestiami), ktoré budú automaticky pri rozvoji makra nahradzované hodnotami, ktoré sa nebudú opakovať – vkladajú sa návestia od 30000$.

```
        .MACRO ABS CO,KAM,?NAV
        MOVL CO,KAM
        BGEQ NAV
        MNEGL KAM,KAM
NAV:    .ENDM ABS
```

Pri volaní ABS A,B vznikne rozvoj:

```
        MOVL A,B
        BGEQ 30000$
        MNEGL B,B
30000$: .ENDM ABS
```

Pri ďalšom volaní sa na miesto parametra NAV vloží 30001$, potom 30002$ atď.

Makrá definujúce makrá:

Ak sa v tele makra nachádza definícia ďalšieho makra, tak „vnútorné“ makro nemožno použiť, pokiaľ sa nezrealizovalo volanie „vonkajšieho“ makra.

Príklad:

```
.MACRO DEF MENO
...
.MACRO MENO A
CLRL A
.ENDM MENO
...
```

.ENDM DEF
Rozvoj volania DEF ZMAZ je:
...
.MACRO ZMAZ A
CLRL A
.ENDM ZMAZ
...
takže po tomto už môžeme použiť ZMAZ R5 a rozvoj bude CLRL R5.

Ak voláme DEF CISTI, zadefinuje sa makro CISTI a môžeme použiť volanie CISTI R5, ktoré má takisto rozvoj CLRL R5.

Makroprocesor:
Makroprocesor je program, ktorý má tieto funkcie:

1. nájsť a uložiť definície makier
2. nájsť volania makier a rozvinúť ich s dosadením parametrov

Makroprocesor môže byť program funkčne nezávislý od asemblera, výstup z makroprocesora (program v jazyku asemblera, v ktorom sa nevyskytujú makrá) je potom vstupom do asemblera.

Podľa počtu prechodov zdrojovým textom rozlišujeme dva typy makroprocesorov:

- dvojprechodové
- jednoprechodové

Dvojprechodový makroprocesor

1. prechod: jeho úlohou je prejsť vstupný text a uložiť nájdené definície makier. Názvy makier ukladá do *tabuľky mien makier* spolu so smerníkom na telo makra, uložené v *tabuľke definícií makier*. V tabuľke definícií makier je uložený najprv tzv. prototyp makra, čiže zoznam parametrov aj s implicitnými hodnotami, aby bolo možné použiť aj nepozičné volanie makra. V tomto prechode sa tiež robia rozvoje systémových makier.
2. prechod: číta zdrojový text a vytvára výstupný text nasledovne: ak ide o inštrukciu alebo direktívu, riadok zdrojového textu sa skopíruje do výsledného textu. Ak sa nájde volanie makra, do výsledného textu sa budú kopírovať riadky z tabuľky definícií makier (čiže telo makra). Podľa smerníka v tabuľke mien makier sa nájde definícia makra v tabuľke definícií, pripraví sa *pole zoznamu parametrov makra*, ktoré sa naplní hodnotami parametrov z volania makra a môžu sa do výsledného textu kopírovať riadky z tela makra, do ktorých sa dosádzajú parametre z uvedeného poľa.

Ak je v tele makra volanie ďalšieho makra, pole zoznamu parametrov a aktuálna pozícia v tabuľke definícií makier sa uložia do zásobníka, pripraví sa pole zoznamu parametrov pre vnorené makro, nájde sa jeho definícia a vkladá sa telo tohto makra. Keď je rozvoj vnoreného makra dokončený, zo zásobníka sa obnoví stav pred vnoreným rozvojom a pokračuje sa v rozvoji vonkajšieho makra.

Dvojprechodový makroprocesor nevie spracovať vnorené definície. Problém je v tom, že definícia vnútorného makra sa objaví až v druhom prechode makroprocesora – pri rozvoji definujúceho makra. Teda táto nová definícia nie je zapísaná v tabuľke mien a definícií makier a preto keď sa vyskytne volanie nového makra, nebude možné urobiť jeho rozvoj. Bolo by v takomto prípade nutné zopakovať oba prechody makroprocesora.

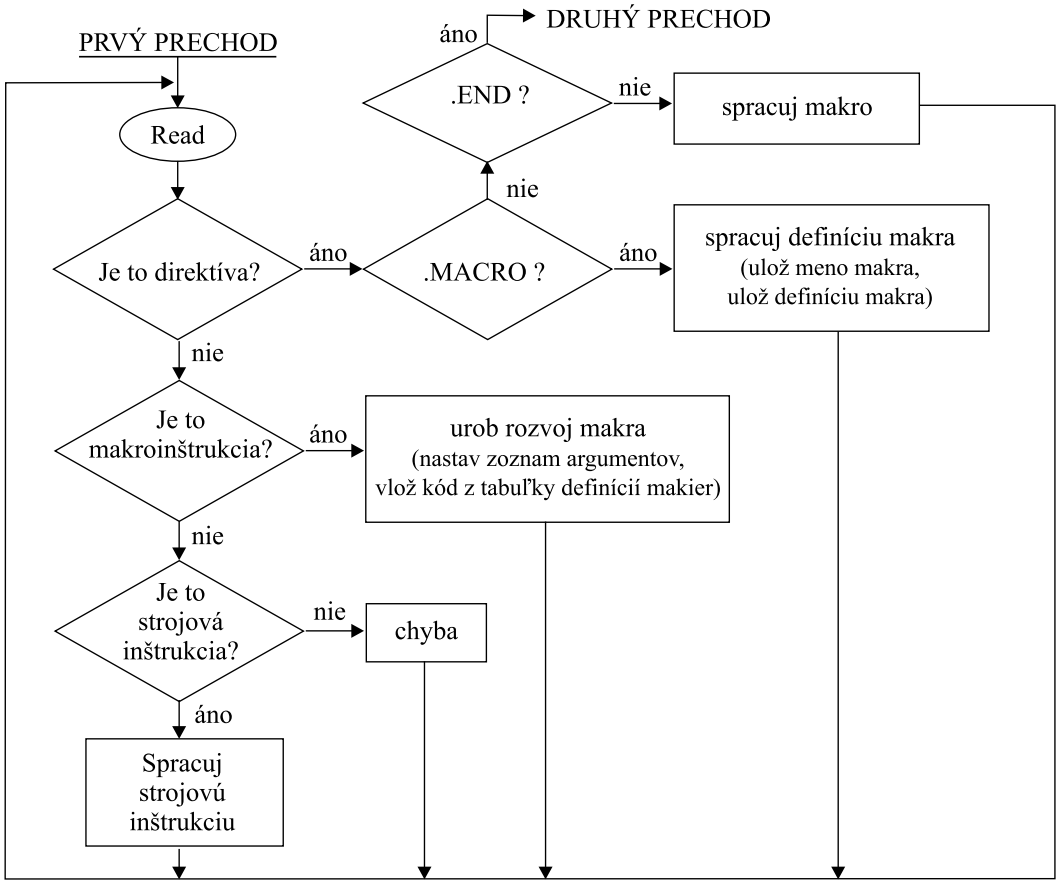
Jednoprechodový makroprocesor

Jednoprechodový makroprocesor v rámci jedného prechodu zdrojovým textom ukladá definície makier a robí aj rozvoje makier. Jedinou požiadavkou je, aby vždy definícia makra predchádzala jeho volaniu. Dokáže (podobne ako dvojprechodový makroprocesor) spracovať vnorené volania makier a tiež makrá definujúce iné makrá.

Makroasembler

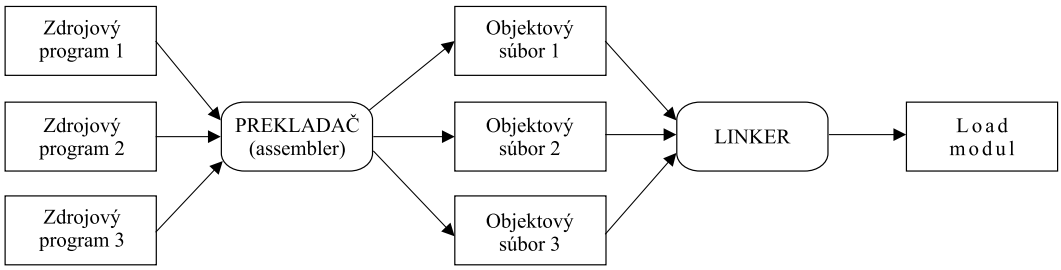
Makroprocesor sa môže pridať ako predprocesor pred asembler, ale je tiež možné implementovať jednoprechodový makroprocesor do prvého prechodu asemblera – výsledok sa nazýva *makroasembler*.

Toto spojenie vylučuje náklady na vytváranie prechodných súborov a tiež mnohé činnosti nie je potrebné implementovať dvakrát (čítanie zdrojového riadku, testovanie typu príkazu, ...).



1.6 Linker a loader

Väčšina programov pozostáva z viacerých procedúr. Kompilátory a asemblery zvyčajne prekladajú vždy len jednu procedúru a preložený výstup uložia na disk. Pred tým, ako je možné spustiť program, musia byť nájdené všetky potrebné preložené procedúry a musia byť správne spojené. Výsledný modul je potom zavedený do pamäte.



Úlohou *linkera* je spojiť separátne preložené procedúry do jedného modulu, zvyčajne nazývaného *load module*. *Loader* potom nahrá load modul do pamäte. Tieto funkcie sú často kombinované.

Preloženie každej procedúry ako separátnej entity má výhodu v tom, že pri zmene v niektorej procedúre stači prekompilovať len zmenenú procedúru (aj keď treba vykonať nanovo linkovanie), a nie všetky, ako by to bolo nutné, ak by kompilátor čítal sériu procedúr a priamo vyrábal spúšťateľný program.

Linker

Pri štarte prvého prechodu asemblera sa nastaví location counter (LC) na 0. Tento krok je ekvivalentný predpokladu, že objektový modul bude umiestnený na (virtuálnej) adrese 0.

Linker, ktorý spája určené moduly do jedného celku, tiež zvyčajne predpokladá, že program začína na adrese 0 (v takomto prípade vytvára „relative load modul"). Keďže na túto adresu možno umiestniť len jeden modul, ostatné musí linker zaradiť zaň. V týchto moduloch musí linker upraviť adresy podľa toho, kde začínajú. K adresám v týchto moduloch sa pripočítava tzv. *relokačný faktor*. Toto je však potrebné len u adries, ktoré nie sú prekladané relatívne, čiže vzhľadom k PC registru.

Pri spájaní modulov musí linker vedieť, ktoré adresy sú v poriadku a ktoré treba relokovať. Túto informáciu mu zapíše asembler do objektového modulu. Ak všetky pamäťové odkazy v module sú vzhľadom k PC registru, nemusí linker robiť žiadne úpravy adries. Takéto moduly nazývame *nezávislé od umiestnenia (position independent code)*.

Ďalej musí linker vyriešiť odkazy medzi modulmi (napr. volanie procedúry definovanej v inom module). Počas prekladu asembler nemôže na miesta týchto odkazov vložiť adresy odkazovaných procedúr (ani relatívne). Návestia (symboly) definované v iných moduloch, než je práve prekladaný modul, sú pre tento modul *externé* (na rozdiel od tých, čo sú definované v súčasnom module, ktoré nazývame *interné* alebo *lokálne*). Asembler uloží informáciu o externých návestiach v objektovom súbore.

Ak k nejakému externému návestiu nenájde linker v ostatných moduloch jeho definíciu, čiže nebude v niektorom module toto návestie definované ako *globálne*, tak vyhlási chybu.

Linker spája separátne adresové priestory objektových modulov do jedného lineárneho adresného priestoru v nasledovných krokoch:

1. Vytvorí tabuľku objektových modulov a ich dĺžok.
2. Na základe tejto tabuľky priradí začiatočné adresy jednotlivým objektovým modulom.
3. Nájde všetky inštrukcie obsahujúce pamäťové adresy a pripočíta k týmto adresám relokačný faktor, rovný začiatočnej adrese modulu, v ktorom sa vyskytuje.
4. Nájde všetky inštrukcie obsahujúce odkazy do iných modulov a naplní tieto odkazy adresami referencovaných objektov.

Štruktúra objektového modulu

Objektový modul (súbor) pozostáva zo šiestich častí:

- **Identifikácia**: meno modulu, čas prekladu, niektoré informácie potrebné pre linker, ako napr. dĺžky jednotlivých častí objektového modulu.
- **Tabuľka globálnych symbolov (Entry point table)**: zoznam symbolov definovaných v module, na ktoré sa môžu odkazovať iné moduly, spolu s ich hodnotami (adresami).
- **Tabuľka externých symbolov (External reference table)**: zoznam symbolov použitých v module, ktoré v ňom nie sú definované, spolu so zoznamom inštrukcií, ktoré ich používajú.
- **Preložený kód (Machine instructions and constants)**: to je jediná časť objektového modulu, ktorá bude nahratá do pamäte na vykonávanie.
- **Tabuľka relokácií (Relocation dictionary)**: zoznam adries, ktoré musia byť relokované pripočítaním relokačného faktora.
- **End-of-module**: adresa začiatku programu – štartovacia adresa (ak ide o hlavný program), prípadne „checksum" na kontorlu chýb pri čítaní modulu.

Väčšina linkerov pracuje v dvoch prechodoch. V prvom prechode linker číta všetky objektové moduly a vyrobí tabuľku názvov objektov a ich dĺžok a tiež *globálnu tabuľku symbolov (global symbol table)* pozostávajúcu zo všetkých globálnych a externých symbolov. V druhom prechode sú objektové moduly čítané, relokované a spojené do jedného modulu.

Loader

Loader umiestňuje load modul do operačnej pamäte a pripraví ho na spustenie. Keďže začiatočná adresa modulu, ktorú predpokladal linker, je zvyčajne rôzna od adresy, na ktorú je program zavedený, loader musí tiež upraviť adresy. Preto musí byť súčasťou load modulu zoznam adries, ktoré treba takto modifikovať. Loader použije uvedenú informáciu na úpravu adries, ale z výslednej podoby strojového kódu ju vymaže. Vykonávanie začina, keď sa urobí skok na štartovaciu adresu programu.

„Čas viazania" (Binding time) a dynamická relokácia

V systémoch so zdieľaním času môžu byť programy umiestnené do pamäte, potom na nejaký čas presunuté na disk a potom opäť nahraté do pamäte. Zvyčajne sa nedá zabezpečiť, aby sa program nahral späť do pamäte na tú istú adresu, ako bol predtým. Ak bol program relokovaný, po opätovnom nahratí do pamäte sú všetky pamäťové odkazy nesprávne. Ak by aj bola ešte dostupná informácia o relokáciách, zaberalo by to mnoho času každý raz po presune programu relokovať všetky adresy.

Problém presúvania zlinkovaných a relokovaných programov súvisí s časom, kedy sa robí „viazanie" (mapovanie) symbolických mien na fyzické adresy. Existuje aspoň 6 možností na „čas viazania" (binding time):

- Keď sa program píše.
- Keď sa program prekladá.
- Keď sa program linkuje, ale pred loadovaním (v tomto a predošlom prípade vzniká „absolute load modul".
- Keď sa program loaduje (nahráva do pamäte).
- Keď sa loaduje bázový register používaný na adresovanie.
- Keď sa vykonáva inštrukcia obsahujúca adresu.

Ak napr. prekladač vytvára priamo „absolute load modul", „viazanie" prebehlo v čase prekladu a program musí byť spustený na adrese, ktorú predpokladal prekladač.

Tu sa vlastne stretávame s dvoma súvisiacimi problémami: prvý – kedy sa symbolické mená mapujú na virtuálne adresy, druhý – kedy sa virtuálne adresy mapujú na fyzické adresy. Až keď prebehnú obe tieto operácie, ukončí sa „viazanie". Keď linker spája separátne adresové priestory do jedného, v skutočnosti vlastne vytvára virtuálny adresný priestor. Relokácia a linkovanie slúžia na namapovanie symbolických mien na určité virtuálne adresy. Toto platí bez ohľadu na to, či systém používa virtuálnu pamäť (kap. 10).

Ak napr. systém používa mechanizmus „run-time" relokačného registra, tak tento register vždy ukazuje na začiatok súčasného programu. K všetkým pamäťovým adresám sa hardwarovo pripočíta obsah relokačného registra, skôr než sa pošlú do pamäte. Keď sa program presunie v pamäti, operačný systém musí zmeniť obsah relokačného registra.

Dynamické linkovanie

Metóda linkovania, ako sme si ju vysvetlili, má tú vlastnosť, že všetky procedúry, ktoré by mohol program volať, sú zlinkované pred spustením programu. Mnoho programov však má procedúry, ktoré sú volané len pri „nezvyčajných" okolnostiach.

Flexibilnejšia je metóda, pri ktorej budú procedúry linkované až pri ich prvom použití. Tento proces je známy ako *dynamické linkovanie*. Umiestnenie preložených modulov na disku je niekde zapamätané (napr. v adresári), takže linker ich môže ľahko nájsť, keď ich bude potrebovať. Keď sa v programe volá procedúra z iného modulu, linker nájde príslušný modul, pridelí mu virtuálnu adresu a vyrieši odkaz na procedúru. Inštrukcia volania procedúry sa opätovne spustí a umožní pokračovanie programu od miesta, kde bol prerušený.

Obsah