

5 VYBRANÉ STROJOVÉ INSTRUKCE PROCESORŮ INTEL PENTIUM




50 hod



5.1 Skupiny instrukcí procesorů Intel Pentium

Tato kapitola představuje významnou referenční část opory předmětu IAS. Student se v ní seznámí s nejdůležitějšími instrukcemi procesoru Pentium, doplněnými příklady jejich použití. Naučí se syntaxi těchto instrukcí, jejich funkci a na jednoduchých příkladech uvidí, jak tyto instrukce typicky používat.



Instrukce označené symbolem  (prakticky všechny instrukce v této kapitole) patří mezi základní sadu instrukcí, které mohou být vyžadovány u zkoušky.

Instrukce, které je možné používat v základním režimu procesorů Pentium lze rozdělit na:

- přenosové instrukce
- instrukce binární aritmetiky
- instrukce pro předávání řízení
- instrukce pro ovládání příznaků
- instrukce pro práci se segmentovými registry
- logické instrukce
- instrukce posuvů a rotací
- jiné instrukce
- instrukce pro práci s bity a slabikami
- řetězové instrukce
- instrukce dekadické aritmetiky
- systémové instrukce
- instrukce FPU

instrukce MMU

5.2 Přenosové instrukce

MOV



Instrukce MOV má dva operandy a slouží k přesunutí dat z operandu src do operandu dst. Cílový operand dst může být registr nebo místo v paměti. Zdrojový operand src může být registr, místo v paměti nebo přímá hodnota. Prováděná operace může být osmibitová, šestnáctibitová nebo 32bitová (ve 32 bitovém režimu). Typ operace je dán velikostí obou operandů – oba operandy musí mít stejnou velikost.

Syntaxe instrukce:

```
MOV dst, src
```

Jsou tedy povoleny následující kombinace operandů:

Typ instrukce	Příklad
mov register, register	mov DX, CX
mov register, immediate	mov BL, 100
mov register, memory	mov BX, count
mov memory, register	mov count, SI
mov memory, immediate	mov count, 23

V případě, že ani z jednoho operandu není zřejmá jeho velikost, např.:

```
mov [BX], 100
```

je třeba velikost určit explicitně, např.:

```
mov WORD [BX], 100
```

Instrukce MOV má i zvláštní variantu, kdy jedním z operandů může být segmentový registr, např.:

```
mov AX, DS
mov ES, AX
```

Instrukce nenastavuje žádné příznaky.

x+y

Příklady použití:

```
MOV AX, BX      ; Move to AX the 16-bit value in BX
MOV AH, AL      ; Move to AL the 8-bit value in AX
MOV AH, 12h     ; Move to AH the 8-bit value 12H
MOV AX, 1234h   ; Move to AX the value 1234h
MOV AX, [1234h] ; Move to AX the memory value at 1234h
MOV [X], AX     ; Move to the memory location
                  ; pointed to by DS:X the value in AX
MOV AX, [DI]    ; Move to AX the 16-bit value pointed
                  ; to by DS:DI
```

```
MOV AX, [BX] ; Move to AX the 16-bit value pointed
              ; to by DS:BX
MOV [BP], AX ; Move to memory address SS:BP
              ; the 16-bit value in AX
MOV AX, [BX+TAB] ; Move to AX the value in memory
                  ; at DS:BX + TAB
MOV [BX+TAB], AX ; Move value in AX to memory
                  ; address DS:BX + TAB
MOV AX, [BX + DI] ; Move to AX the value in memory
                  ; at DS:BX + DI
```

XCHG



Instrukce XCHG vymění hodnoty dvou operandů dst a src. Alespoň jeden operand musí být registr. Přímé operandy nejsou povoleny.

Syntaxe instrukce

```
xchg    dst, src
```

Instrukce nenastavuje žádné příznaky.



Příklady

```
xchg    EAX, EDX
xchg    response, CL
xchg    total, DX
```

CBW, CWD, CDQ



Instrukce CBW, CWD a CDQ provádějí znaménkové rozšíření registru a to následovně:

- CBW (convert byte to word) rozšíří AL do AH
- CWD (convert word to doubleword) rozšíří AX do DX
- CDQ (convert doubleword to quadword) rozšíří EAX do EDX:EAX

Syntaxe instrukcí:

```
CBW
CWD
CDQ
```



Příklad:

```
mov ax, 0FF9Bh
cwb                      ; DX:AX = FFFFFFFF9Bh
```

XLATB

Instrukce XLATB (Table Look-up Translation) se obvykle používá pro transformaci znaků pomocí tabulky. Instrukce uloží do registru AL hodnotu z paměti na adrese DS:[BX+AL].

Syntaxe instrukce:

```
XLATB
XLAT mem
```

Explicitní vyjádření operandu v instrukci XLAT umožňuje přepis segmentového registru, základním registrem zdrojového operandu je vždy registr BX.

Instrukce nemění příznaky.

Příklad:



Jednoduché šifrování číslic pomocí tabulky

Vstupní číslice: 0 1 2 3 4 5 6 7 8 9

Výstupní číslice: 4 6 9 5 0 3 1 8 7 2.

```
.DATA
xlat_table DB '4695031872'
...
.CODE
mov     BX, xlat_table
mov     AL, input
sub     AL, '0' ; converts input character to index
xlatb           ; AL = encrypted digit character
mov     output, AL
```

LDS, LES

Instrukce LDS a LES naplní segmentový registr určený druhým a třetím písmenem instrukce a cílový registr ukazatelem (segment:offset) z paměti adresované zdrojovým operandem.

Syntaxe instrukcí

```
LDS dst, src
LES dst, src
```

Instrukce nenastavuje žádné příznaky.

LEA

Instrukce LEA vypočítá efektivní adresu (offset) operandu src a uloží jej do registru dst.

Syntaxe instrukce

LEA dst, src

Instrukce nenastavuje žádné příznaky.



Kontrolní otázky:

- Vysvětlete rozdíl mezi instrukcemi LEA a LES.
- Jakou instrukcí by bylo možné nahradit instrukci XLAT?
- Instrukce CBW je určena pro čísla se znaménkem nebo bez znaménka?
- Lze použít instrukci MOV pro práci se segmentovými registry?

5.3 Instrukce binární aritmetiky

INC, DEC



Instrukce INC přičte jedničku k obsahu operandu, instrukce DEC jedničku odečte. Operand může být registr nebo místo v paměti. Prováděná operace může být osmibitová, šestnáctibitová nebo 32bitová (ve 32 bitovém režimu).

Syntaxe instrukce:

```
INC dst
DEC dst
```

Instrukce nastavuje příznaky S, A, Z, P, O. Nemění příznak C!

Příklady



```
INC AX      ; AX = AX + 1
INC [BX]    ; memory location increased by 1
INC [MYVAR] ; memory location increased by 1
```

ADD, SUB



Instrukce ADD provádí operaci aritmetického sčítání operandu src a dst a uloží výsledek do prvního cílového operandu dst. Instrukce SUB provádí operaci aritmetického odečítání operandu dst-src a uloží výsledek do prvního cílového operandu dst. Cílový operand dst může být registr nebo místo v paměti. Zdrojový operand src může být registr, místo v paměti nebo přímá hodnota. Prováděná operace může být osmibitová, šestnáctibitová nebo 32bitová (ve 32 bitovém režimu).

Syntaxe instrukce:

```
ADD dst, src
SUB dst, src
```

Instrukce nastavuje příznaky OF, SF, ZF, AF, PF a CF (podle výsledku).

Příklady:



```
ADD AX, BX    ; register addition
ADD AX, 5h    ; immediate addition
ADD [BX], AX  ; addition to memory location
ADD AX, [BX]  ; memory location added to register
ADD DI, MYVAR ; memory offset added to register
```

ADC, SBB



Instrukce ADC provádí operaci aritmetického sčítání operandu src a dst a **hodnoty příznaku přenosu C** a uloží výsledek do prvního cílového operandu dst. Instrukce SUB provádí operaci aritmetického odečítání operand dst-src-C a uloží výsledek do prvního cílového operandu dst. Cílový operand dst může být registr nebo místo v paměti. Zdrojový operand src může být registr, místo v paměti nebo přímá hodnota. Prováděná operace může být osmibitová, šestnáctibitová nebo 32bitová (ve 32 bitovém režimu).

Syntaxe instrukce:

```
ADC dst, src
SBB dst, src
```

Instrukce se používá pro sčítání delších čísel než 16 bitů (v 16-bitovém režimu) nebo větších než 32 bitů (v 32-bitovém režimu).

Instrukce nastavuje příznaky OF, SF, ZF, AF, PF a CF (podle výsledku).



Příklad:

Provedeme sečtení obsahu 32 bitových čísel, uložených ve dvojicích registrů BX_AX a DX_CX.

```
add ax, cx ; this produces the 32 bit sum of
adc bx, dx ; bx_ax + dx_cx
```

NEG



Instrukce NEG obrací znaménko operandu. Operand může být registr nebo místo v paměti. Prováděná operace může být osmibitová, šestnáctibitová nebo 32bitová (ve 32 bitovém režimu).

Syntaxe instrukce:

```
NEG dst
```

Instrukce nastavuje příznaky CF, OF, SF, ZF, AF a PF (podle výsledku).



Příklad:

```
valB DB -1
valW DW +32767
mov al,[valB] ; AL = -1
neg al ; AL = +1
neg [valW] ; valW = -32767
```



Problém k zamyšlení:

Předpokládejme, že AX obsahuje hodnotu -32768 a provedeme instrukci NEG AX. Jak to dopadne?

MUL, IMUL



Násobení se provádí instrukcí MUL (násobení čísel bez znaménka) nebo instrukcí IMUL (násobení čísel se znaménkem). Prováděná operace může být osmibitová (součin dvou 8 bitových čísel, 16 bitový výsledek), šestnáctibitová nebo 32bitová (ve 32 bitovém režimu).

Násobení na procesorech Pentium vychází z následujících předpokladů:

- Součin po násobení je vždy dvojnásobně velký než činitelé
- Pokud násobíme dvě 8-bitová čísla, výsledek je 16-bitový

- Pokud násobíme dvě 16-bitová čísla, výsledek je 32-bitový
- Pokud násobíme dvě 32-bitová čísla, výsledek je 64-bitový
- V tomto případě nemůže dojít k přetečení

Syntaxe základní verze instrukcí:

```
mul src
imul src
```

Pro různé velikosti operandu src se tedy provede:

- Pro 8 bitový operand: $AX = AL * src$
- Pro 16 bitový operand: $AX:DX = AX * src$
- Pro 32 bitový operand: $EAX:EDX = EAX * src$

Později byly instrukce pro násobení se znaménkem rozšířeny o varianty, které mají více operandů a kde cílový registr a zdrojový registr jsou zadávány odděleně. V tomto případě ale už může dojít k přetečení. Tyto instrukce mají následující syntaxi:

```
imul dst, src          ; dst = dst * src
imul dst, imm          ; dst = dst * imm
imul dst, src, imm     ; dst = src * imm
```

Příznaky jsou nastaveny tak, že většina příznaků je nedefinována. Příznaky O a C jsou vynulovány, pokud se výsledek vejde do polovičního registru. Tj. pro násobení dvou 16-bitových čísel, pokud 16 nejvýznamnějších bitů je nulových, příznaky C a O jsou nulové.

x+y

Příklad násobení bez znaménka:

```
mul cx    ; DX:AX = AX * CX
```

Příklady násobení se znaménkem:

```
imul bl    ; AX = BL * AL
imul bx    ; DX:AX = BX * AX
```

Příklady násobení se znaménkem s více operandy:

```
imul cl, [bl]    ; CL = CL * [DS:BL]
                  ; Může dojít k přetečení (overflow)!
imul cx, dx, 12h ; CX = 12h * DX
```

DIV, IDIV



Dělení se provádí instrukcí DIV (dělení čísel bez znaménka) nebo instrukcí IDIV (dělení čísel se znaménkem). Prováděná operace může být osmibitová, šestnáctibitová nebo 32bitová (ve 32 bitovém režimu). Velikost dělitele určuje, která z variant se použije. Dělenec má dvojnásobnou délku a je v registru AX, DX:AX nebo EDX:EAX. Dělitel je registr nebo místo v paměti.

Dělení na procesorech Pentium vychází z následujících předpokladů:

- Výsledkem operace dělení je podíl a zbytek po dělení
- Dělenec je vždy dvojnásobně velký než dělitel, podíl a zbytek
- Může dojít k chybě dělení nulou

Syntaxe instrukcí:

```
div src
idiv src
```

Pro různé velikosti operandu src se tedy provede:

- Pro 8 bitový operand: $AL = AX / src$ $AL = AX \% src$
- Pro 16 bitový operand: $AX = AX:DX / src$ $DX = AX:DX / src$
- Pro 32 bitový operand: $EAX = EAX:EDX / src$ $EDX = EAX:EDX / src$

Pokud podíl nelze uložit do paměťového prostoru daného velikostí cílového operandu generuje se výjimka 0 – chyba dělení. Hodnoty příznaků CF, OF, SF, ZF, AF a PF nejsou definovány.

Při dělení dvou stejně velkých hodnot je třeba dělence rozšířit na dvojnásobek. Pro čísla bez znaménka se vloží 0 do horního slova. Pro čísla se znaménkem je třeba dělence znaménkově roztáhnout. K tomu lze použít instrukce:

- CBW (convert byte to word)
 $AX = xxxx\ xxxx\ snnn\ nnnn$
 $AX = ssss\ ssss\ snnn\ nnnn$
- CWD (convert word to double)
 $DX:AX = xxxx\ xxxx\ xxxx\ xxxx\ snnn\ nnnn\ nnnn\ nnnn$
 $DX:AX = ssss\ ssss\ ssss\ ssss\ snnn\ nnnn\ nnnn\ nnnn$
- CWDE (convert double to double-word extended)

x+y

Příklad:

Dělení čísel bez znaménka NUMB číslem NUMB1 (obě jsou v paměti).

```
MOV AL, [NUMB]      ;get NUMB
MOV AH, 0           ;zero extend
DIV byte [NUMB1]
MOV [ANSQ], AL      ;save quotient
MOV [ANSR], AH      ;save remainder
```

Příklad:

Dělení čísel se znaménkem NUMB číslem NUMB1 (obě jsou v paměti).

```

MOV AL, [NUMB]      ;get NUMB
CBW                  ;signed-extend
IDIV byte [NUMB1]
MOV [ANSQ], AL       ;save quotient
MOV [ANSR], AH       ;save remainder

```

Co můžeme udělat se zbytkem? Zapomenout, a výsledek je ořezán nebo použít jej pro zaokrouhlení. Pro čísla bez znaménka zaokrouhlení znamená, že dvojnásobek zbytku je porovnán s dělitelem.

Příklad:

Dělení AX obsahem BL a zaokrouhlení:

```

DIV BL
ADD AH, AH      ;double remainder
CMP AH, BL      ;test for rounding
JB .NEXT
INC AL
.NEXT

```



Kontrolní otázky:

- Které aritmetické instrukce mají jeden operand a které dva?
- Jaké příznaky a jak nastavují příznaky instrukce ADD a SUB?
- Jak je nastaven příznak C po použití instrukce ADD a SUB?
- Jak je nastaven příznak C po použití instrukce INC?
- Vysvětlete, k čemu jsou vhodné instrukce ADC a SBC.
- Vysvětlete rozdíl mezi instrukcemi MUL a IMUL.

5.4 Logické instrukce

AND, OR, XOR, NOT



Logické instrukce jsou instrukce AND, OR, XOR a NOT. Tyto instrukce pracují po jednotlivých bitech. Instrukce AND (resp. OR a XOR) provádí bitovou operaci AND (respektive OR a nonekvivalenci) mezi svými dvěma operandy a uloží výsledek do prvního cílového operandu dst. Cílový operand může být registr nebo místo v paměti. Zdrojový operand src může být registr, místo v paměti nebo přímá hodnota. Instrukce NOT provádí bitovou negaci svého operandu. Prováděná operace může být osmibitová, šestnáctibitová nebo 32bitová (ve 32 bitovém režimu).

Instrukce assembleru	Zápis v jazyce C
NOT A	$A = \sim A$
AND A, B	$A \&= B$
OR A, B	$A = B$
XOR A, B	$A ^= B$

Syntaxe instrukcí:

```
NOT dst
AND dst, src
OR dst, src
XOR dst, src
```

Vyjma instrukce NOT nastavují tyto instrukce příznaky takto:

- Nulují carry (C)
- Nulují overflow (O)
- Nastavují zero flag (Z) podle výsledku
- Nastavují sign flag (S) podle výsledku
- Nastavují parity bit (P) podle výsledku
- Ničí auxiliary carry flag (A)

Instrukce NOT neovlivňuje žádné příznaky.



Instrukce AND a OR se často používají pro vymaskování dat. Hodnota použité masky slouží k násilnému nastavení některých bitů na 1 nebo 0. Instrukce AND nastaví vybrané bity na nulu, např.

```
AND CL, 0Fh
```

Instrukce OR nastaví vybrané bity na jedničku, např.:

```
OR CL, 0Fh
```

Následující tabulka ukazuje příklady aplikace některých logických instrukcí:

AL	1100 1010
NOT AL	
AL	0011 0101

AL	0011 0101
BL	0110 1101
AND AL, BL	
AL	0010 0101

AL	0011 0101
BL	0000 1111
AND AL, BL	
AL	0000 0101

AL	0011 0101
BL	0110 1101
OR AL, BL	
AL	0111 1101

AL	0011 0101
BL	0000 1111
OR AL, BL	
AL	0011 1111

AL	0011 0101
BL	0110 1101
XOR AL, BL	
AL	0101 1000



Kontrolní otázky:

- Které logické instrukce mají jeden operand a které dva?
- Jaké příznaky a jak nastavují příznaky logické instrukce?
- Jak je nastaven příznak C po použití libovolné logické instrukce?
- Vysvětlete rozdíl mezi instrukcemi NEG a NOT.

5.5 Instrukce posuvů a rotací

**SHL, SHR, SAR,
RCL, RCR,
ROL, ROR**



Instrukce rotací a posuvů pracují s posloupností bitů v registru nebo v paměti. Tyto instrukce bity posouvají doleva (instrukce SHL), posouvají doprava (instrukce SHR), posouvají doprava s ponecháním znaménkového bitu (instrukce SAR), rotují doprava (instrukce ROR), rotují doleva (instrukce ROL), rotují doprava přes příznak C (instrukce RCR) a rotují doleva přes příznak C (instrukce RCL).

Instrukce rotací a posuvů pracují nad cílovým operandem reg. Cílový operand může být registr nebo místo v paměti. Počet bitů count, o které je hodnota posouvána/rotována je dán zdrojovým operandem: buď je 1, nebo je dán obsahem registru CL, nebo přímým operandem. Skutečný počet je vyčíslen pomocí vztahu (count and 1FH), t.j. je omezen hodnotou 31.

Prováděná operace může být osmibitová, šestnáctibitová nebo 32bitová (ve 32 bitovém režimu).

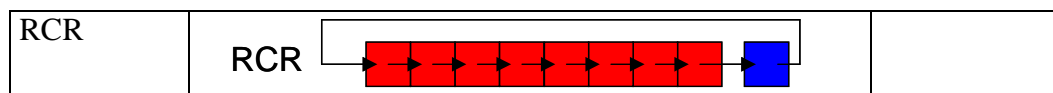
Syntaxe instrukcí:

```
SHL reg, count      ; Přímo zadaná hodnota count
SHL reg              ; Hodnota count v registru CL
SHR reg, count
SHL reg
...
```

Pro count = 0 se příznaky nemění. Jinak se příznaky SF, ZF, a PF nastavují podle výsledku a hodnota příznaku AF není definována.

Činnost jednotlivých instrukcí je přehledně naznačena v následující tabulce:

Instrukce	Obrázek	Poznámka
SHL		
SHR		
SAR		
ROL		
ROR		
RCL		



x+y

Příklady hodnot registrů po použití instrukcí rotací a posuvů

```

mov ax,3      ; Initial register values      AX = 0000 0000 0000 0011
mov bx,5      ;                               BX = 0000 0000 0000 0101
or ax,9       ; ax <- ax | 0000 1001         AX = 0000 0000 0000 1011
and ax,10101010b ; ax <- ax & 1010 1010     AX = 0000 0000 0000 1010
xor ax,0FFh   ; ax <- ax ^ 1111 1111         AX = 0000 0000 1111 0101
neg ax ; ax <- (-ax)                          AX = 1111 1111 0000 1011
not ax ; ax <- (~ax)                          AX = 0000 0000 1111 0100
Or ax,1       ; ax <- ax | 0000 0001         AX = 0000 0000 1111 0101
shl ax,1      ; logical shift left by 1 bit  AX = 0000 0001 1110 1010
shr ax,1      ; logical shift right by 1 bit  AX = 0000 0000 1111 0101
ror ax,1      ; rotate right (LSB=MSB)       AX = 1000 0000 0111 1010
rol ax,1      ; rotate left (MSB=LSB)        AX = 0000 0000 1111 0101
mov cl,3      ; Use CL to shift 3 bits        CL = 0000 0011
shr ax,cl     ; Divide AX by 8                AX = 0000 0000 0001 1110
mov cl,3      ; Use CL to shift 3 bits        CL = 0000 0011
shl bx,cl     ; Multiply BX by 8              BX = 0000 0000 0010 1000
    
```



Kontrolní otázky:

- Vysvětlete, proč neexistuje instrukce aritmetického posuvu vlevo.
- Po kolika použitích instrukce „RCR AX,1“ bude obsah registru AX nezměněn?
- Po kolika použitích instrukce „ROL AX,2“ bude obsah registru AX nezměněn?

5.6 Instrukce pro ovládání příznaků

STC, CLC, CMC, STD, CMD, STI, CLI



Tyto instrukce slouží pro změnu hodnot některých jednotlivých příznaků. Každá z těchto instrukcí mění pouze jeden příznak a ostatní příznaky nemění.

Funkce těchto instrukcí je uvedena v následující tabulce:

Instrukce	Název instrukce	Funkce
STC	Set Carry Flag	CF = 1
CLC	Clear Carry Flag	CF = 0
CMC	Complement Carry Flag	CF = not CF
STD	Set Direction Flag	DF = 1
CLD	Clear Direction Flag	DF = 0
STI	Set Interrupt Flag	IF = 1
CLI	Clear Interrupt Flag	IF = 0

LAHF, SAHF



Instrukce LAHF (Load Status Flags into AH Register) uloží obsah některých příznakových bitů (konkrétně SF:ZF:0:AF:0:PF:1:CF) do registru AH. Instrukce neovlivňuje žádné příznaky.

Instrukce SAHF (Store AH into FLAGS Register) uloží obsah registru AH do některých příznakových bitů (konkrétně SF:ZF:0:AF:0:PF:1:CF).

Syntaxe instrukcí:

LAHF

SAHF

5.7 Instrukce pro předávání řízení

Nepodmíněné skoky



Nepodmíněné skoky jsou instrukce, které způsobí to, že se příští instrukce přečte z jiného místa, než je instrukce následující za prováděnou instrukcí. Odpovídají příkazu “goto”. Podle dosahu se nepodmíněné skoky dělí na tři typy:

- Short – 2-bytová instrukce, která dovoluje skočit na místo v rozsahu +127 and -128 bytů od místa, které následuje za příkazem JMP.
- Near – 3-bytová instrukce, která dovoluje skočit v rozsahu +/- 32KB od následující instrukce v rámci běžného kódového segmentu.
- Far– 5-bytová instrukce, která dovoluje skočit na jakékoli místo v celém adresovém prostoru.

Syntaxe instrukce:

JMP dst

Operandem instrukce dst může být návěští, šestnáctibitový registr nebo šestnáctibitové místo v paměti.

Instrukce nemění žádné příznaky.

Podmíněné skoky



V závislosti na stavu příznaků můžeme provádět podmíněné skoky, tj. skoky, které se provedou, pokud je jistá podmínka splněna.

Syntaxe instrukce:

JXX dst

Operandem instrukce dst může být pouze návěští.

Podmíněné skoky testují příznaky sign (S), zero (Z), carry (C), parity (P), a overflow (O). Je však také možno provádět podmíněné skoky na základě předchozího porovnání dvou čísel instrukcí CMP. Je však třeba rozlišovat, zda jde o čísla se znaménkem (používáme termíny Greater/Less/Equal) nebo bez znaménka (termíny Above/Below/Equal).

K dispozici máme následující instrukce podmíněného skoku:

Příkaz	Popis	Podmínka
JA=JNBE	Jump if above	C=0 & Z=0
	Jump if not below or equal	
JBE=JNA	Jump if below or equal	C=1 Z=1
JAE=JNB=JNC	Jump if above or equal	C=0
	Jump if not below	
	Jump if no carry	
JB=JNA=JC	Jump if below	C=1
	Jump if carry	

JE=JZ	Jump if equal	Z=1
	Jump if Zero	
JNE=JNZ	Jump if not equal	Z=0
	Jump if not zero	
JS	Jump Sign (MSB=1)	S=1
JNS	Jump Not Sign (MSB=0)	S=0
JO	Jump if overflow set	O=1
JNO	Jump if no overflow	O=0
JG=JNLE	Jump if greater	
	Jump if not less or equal	S=0 & Z=0
JGE=JNL	Jump if greater or equal	S=0
	Jump if not less	
JL=JNGE	Jump if less	S^O
	Jump if not greater or equal	
JLE=JNG	Jump if less or equal	S^O Z=1
	Jump if not greater	
JCXZ	Jump if register CX=zero	CX=0

CMP



Instrukce CMP provádí operaci porovnání hodnot dvou operandů src a dst. Instrukce ve skutečnosti provede operaci aritmetického odečítání operandu dst-src avšak výsledek nikam neuloží, pouze nastaví příznaky pro pozdější podmíněný skok. Cílový operand dst může být registr nebo místo v paměti. Zdrojový operand src může být registr, místo v paměti nebo přímá hodnota. Prováděná operace může být osmibitová, šestnáctibitová nebo 32bitová (ve 32 bitovém režimu).

Syntaxe instrukce:

```
CMP dst, src
```

Instrukce nastavuje stejné příznaky jako instrukce SUB.



Příklad: Testování vstupu na znak konce řádku.

```
read_char:
    . . .
    cmp    AL, 0DH        ; 0DH = ASCII carriage return
    je     CR_received
    inc    CL
    jmp    read_char
    . . .
CR_received:
```

TEST

Instrukce TEST provádí nedestruktivní operaci AND mezi dvěma operandy. Instrukce provede operaci AND avšak výsledek nikam neuloží, pouze nastaví



příznaky pro pozdější podmíněný skok. Cílový operand `dst` může být registr nebo místo v paměti. Zdrojový operand `src` může být registr, místo v paměti nebo přímá hodnota. Prováděná operace může být osmibitová, šestnáctibitová nebo 32bitová (ve 32 bitovém režimu).

Syntaxe instrukce:

```
TEST dst, src
```

Instrukce nastavuje stejné příznaky jako instrukce `AND`.



Příklad: skoč pokud buď bit 0 nebo 1 v `AL` je nastaven.

```
test al, 00000011b
jnz ValueFound
```

LOOP



Instrukce `LOOP` je kombinací instrukcí `DEC` a `JNZ`. Instrukce dekrementuje registr `CX` a pokud `CX` není 0, skočí na návěští, které je operandem instrukce. Pokud je `CX` nula, je provedena instrukce následující za `LOOP`.

Syntaxe instrukce:

```
LOOP label
```



Příklad: Kopírování bloku 100 bajtů.

```
mov cx, 100                ;load count
mov si, BLOCK1
mov di, BLOCK2
.Again
mov al, [SI]
inc SI                    ; precti znak
mov [DI], al
inc DI
loop .Again               ;repeat 100 times
```

LOOPE, LOOPNE, LOOPZ, LOOPNZ



Tyto instrukce jsou vhodné pro smyčky, které se zastaví buď po proběhnutí několika iterací nebo na základě jiné podmínky

Syntaxe instrukce:

```
LOOPE label
LOOPZ label
```

Význam:

$$(E)CX = (E)CX - 1$$

if (E)CX > 0 and ZF=1, jump to label

Vhodná pro hledání prvního prvku v poli, který se nerovná dané hodnotě.

Syntaxe instrukce:

LOOPNE label

LOOPNZ label

Význam:

(E)CX = (E)CX - 1

if (E)CX > 0 and ZF=0, jump to label

Vhodná pro hledání prvního prvku v poli, který se rovná dané hodnotě.

x+y

Příklad:

Mějme posloupnost bajtů String1, která obsahuje řetězec znaků, ukončený bajtem s nulovou hodnotou. Chceme jej zkopírovat do řetězce v maximální délce 127 bajtů..

```

mov si, String1 ; si points to beginning of String1
mov di, String2 ; di points to beginning of String2
mov cx, 127      ; Max 127 chars to String2
.StrLoop:
mov al, [SI]     ; get char from String1
inc si          ; AL=[DS:SI]; SI = SI + 1
mov [DI], al
inc DI          ; [DS:DI] = AL; DI = DI + 1
cmp al, 0        ; see if zero terminator
loopne .StrLoop ; quit if AL or CX = 0

```

?

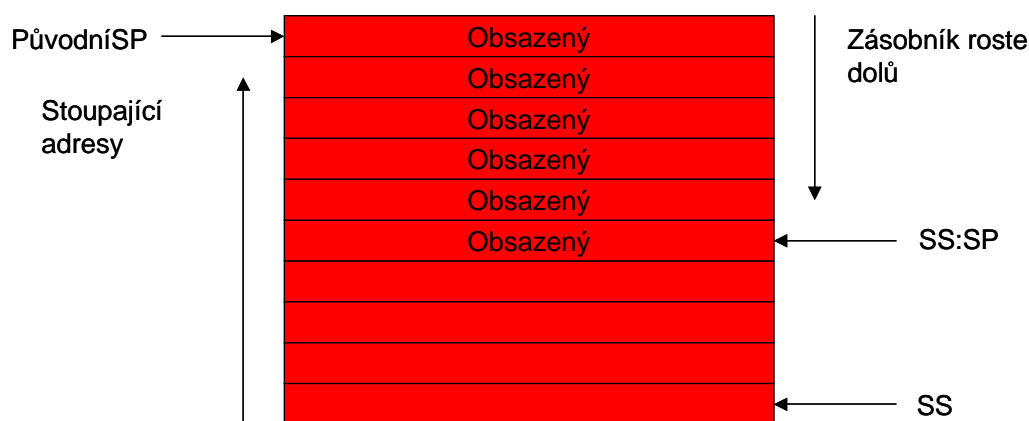
Kontrolní otázky:

- Vysvětlete pojmy short, near a far.
- Vysvětlete rozdíl mezi instrukcemi JA a JG.
- Vysvětlete rozdíl mezi instrukcemi JA a JNBE.
- Vysvětlete rozdíl mezi instrukcemi CMP a SUB.
- Vysvětlete rozdíl mezi instrukcemi TEST a AND.
- Vysvětlete rozdíl mezi instrukcemi LOOPE a LOOPZ.

5.8 Instrukce pro práci se zásobníkem

Zásobník slouží k ukládání dočasných dat. Data jsou ze zásobníku vybírána v opačném pořadí než jsou vkládána – jde o Last-in-first-out (LIFO) přístup. Zásobník má jediný bod přístupu, kterým je vrchol zásobníku. Se zásobníkem se pracuje pomocí instrukcí PUSH a POP.

Zásobník je uložen vždy v zásobníkovém segmentu a vrchol zásobníku je určen obsahem registru ukazatele zásobníku SP (Stack Pointer), nebo ESP ve 32bitovém režimu.



PUSH



Instrukce PUSH vloží hodnotu na vrchol zásobníku. Operandem instrukce může být šestnáctibitový registr, 32bitový registr nebo přímá hodnota. Operandem také může být segmentový registr.

Syntaxe instrukce:

```
PUSH src
```

Instrukce nemění žádné příznaky.

POP



Instrukce POP vybere hodnotu ze zásobníku. Operandem instrukce může být šestnáctibitový registr nebo 32bitový registr. Operandem také může být segmentový registr vyjma CS.

Syntaxe instrukce:

```
POP dst
```

Instrukce nemění žádné příznaky.



Příklad: Použití instrukcí PUSH a POP pro dočasné uschování obsahu registrů AX a BX.

```
PUSH AX ; Place AX on the stack
PUSH BX ; Place BX on the stack
...
```

```
< modify contents of AX and BX >
...
POP BX ; Restore original value of BX
POP AX ; Restore original value of AX
```

PUSHF, POPF, PUSHFD, POPFD



Protože se instrukce PUSH a POP nedají použít pro registr příznaků, existují dvě speciální instrukce PUSHF (PUSH 16-bit flags) a POPF (POP 16-bit flags). Pro 32 bitové příznaky se použijí instrukce PUSHFD a POPFD.

Syntaxe instrukce:

```
PUSHF
POPF
PUSHFD
POPFD
```

Instrukce PUSHF a PUSHFD nemění žádné příznaky.

PUSHA, POPA, PUSHAD, POPAD



Instrukce PUSHA (Push All General-Purpose Registers) a POPA (Pop All General-Purpose Registers) slouží k uschování (obnovení) skupiny registrů do zásobníku. Uschovávají se registry AX, CX, DX, BX, původní SP, BP, SI a DI. Obnovují se registry DI, SI, BP, BX, DX, CX a AX. Pro uschování nebo obnovení 32bitových registrů se používají instrukce PUSHAD a POPAD. Tyto instrukce pracují s registry EAX, ECX, EDX, EBX, ESP, EBP, ESI a EDI.

Syntaxe instrukce:

```
PUSHA
POPA
PUSHAD
POPAD
```

Instrukce nemění žádné příznaky.

CALL



Instrukce CALL se používá pro volání podprogramu (procedure). Instrukce může být typu NEAR (provádí volání podprogramu uvnitř segmentu) nebo FAR, kdy instrukce provádí volání podprogramu mezi segmenty.

Instrukce NEAR CALL provádí následující kroky:

- $SP = SP - 2$
- $(SS:SP) = IP$
- $IP = \text{adresa podprogramu}$

Instrukce FAR CALL provádí následující kroky:

- $SP = SP - 4$
- $(SS:SP) = CS:IP$
- $CS:IP = \text{adresa podprogramu}$

Syntaxe instrukce:

`CALL dst`

Operandem `dst` může být návěští, šestnáctibitový registr, 32bitový registr, šestnáctibitové místo v paměti, 32bitové místo v paměti.

Instrukce nenastavuje žádné příznaky.

RET



Instrukce `RET` přenese řízení z podprogramu zpátky do volajícího programu. Použije se pro to návratová adresa, uložená do zásobníku instrukcí `CALL`. Adresa musí být v okamžiku provádění `RET` na vrcholu zásobníku. Instrukce může být typu `NEAR` (provádí návrat z podprogramu uvnitř segmentu) nebo `FAR`, kdy instrukce provádí návrat z podprogramu mezi segmenty. Instrukce může mít volitelný celočíselný operand, který říká, kolik bajtů se má navíc odstranit ze zásobníku.

Instrukce `NEAR RET` provádí následující kroky:

- $IP = (SS:SP)$
- $SP = SP + 2 + [\text{volitelný celočíselný operand}]$

Instrukce `FAR RET` provádí následující kroky:

- $CS:IP = (SS:SP)$
- $SP = SP + 4 + [\text{volitelný celočíselný operand}]$

Syntaxe instrukce:

`RET`

`RET imm`

Operandem `imm` může být pouze přímý operand.

Instrukce nenastavuje žádné příznaky.

IRET



Instrukce `IRET` se používá pro návrat z podprogramu pro ošetření přerušení (`ISR`, `Interrupt Service Routine`). Instrukce pracuje podobně jako instrukce `FAR RET`, avšak navíc vyjme ze zásobníku ještě jedno slovo, obsahující příznaky.

Instrukce `IRET` provádí následující kroky:

- $CS:IP = (SS:SP)$

- $SP = SP + 4$
- POPF

Syntaxe instrukce:

IRET



Kontrolní otázky:

- Vysvětlete pojem LIFO.
- Které příznaky mění instrukce PUSH a POP?
- Které příznaky mění instrukce PUSHF?
- Které příznaky mění instrukce POPF?
- Popište činnost dvojice instrukcí CALL a RET?
- Vysvětlete význam parametru „imm“ u instrukce „RET imm“.

5.9 Řetězcové instrukce

Řetězcové operace jsou vhodným způsobem jak provést nějakou operaci s velkým polem dat. Typickou takovouto operací je kopírování bloku dat, porovnání bloku dat nebo vyplnění bloku dat hodnotou.

Řetězcové instrukce slouží k tomu, abychom mohli provést nějakou operaci s větším blokem dat, umístěných v paměti. Instrukce pracují s indexregistry SI (adresa zdrojového operandu) a DI (adresa cílového operandu). Obsahy těchto registrů jsou po provedení každé řetězcové instrukce inkrementovány nebo dekrementovány. Příznak DF (Direction flag) ovládá inkrementaci nebo dekrementaci SI a DI takto:

- DF = clear (0): increment SI a DI
- DF = set (1): decrement SI a DI

MOVSB, MOVSW, MOVSD



Instrukce MOVSB, MOVSW a MOVSD kopíruje data z adresy DS:SI na adresu ES:DI. Registry SI a DI jsou automaticky inkrementovány/dekrementovány o 1 (MOVSB), 2 (MOVSW) nebo 4 (MOVSD).

Syntaxe instrukce:

```
MOVSB
MOVSW
MOVSD
MOVS dst, src
```

Explicitní vyjádření operandů v instrukci MOVS umožňuje přepis segmentového registru zdrojového operandu - adresa cílového operandu je vždy dána předpisem ES:(E)DI, indexregistrem zdrojového operandu je vždy registr (E)SI.

Tyto instrukce příznaky nemění.



Příklad:

```
segment data
source DW 0FFFFh
target DW ?
segment code
mov si, source
mov di, target
movsw
```

REP



Před instrukcí MOVSB může být vložen prefix REP (repeat prefix). V tom případě registr CX obsahuje čítač opakování, kolikrát je tato instrukce provedena.



Příklad: Kopíruj 20 slov ze source do target


```

segment data
source resw 20
target resw 20
segment code
cld                ; direction = forward
mov cx, 20         ; set REP counter
mov si, source
mov di, target
rep movsw

```

CMPSB, CMPSW, CMPSD



Instrukce CMPSB, CMPSW, a CMPSD porovnávají hodnotu operandu DS:SI a ES:DI a v závislosti na hodnotě příznaku DF (0/1) zvýší/sníží obsahy obou indexregistru o hodnotu 1/2/4 (pro slabiku/slovo/dvouslovo).. Instrukce CMPSB porovnává dva bajty, CMPSW dvě slova, CMPSD dvě dvouslova. Obvykle se používají s prefixem REP.

Syntaxe instrukcí:

```

CMPSB
CMPSW
CMPSD
CMPS dst, src

```

Explicitní vyjádření operandů v instrukci CMPS umožňuje přepis segmentového registru zdrojového operandu - adresa cílového operandu je vždy dána předpisem ES:(E)DI, indexregistrem zdrojového operandu je vždy registr (E)SI.

Tyto instrukce mění příznaky stejně jako instrukce CMP.



Příklad: Porovnání dvou hodnot

```

segment data
source DW 1234h
target DW 5678h
segment code
mov si, source
mov di, target
cmpsw    ; compare words
ja L1    ; jump if source > target
jmp L2   ; jump if source <= target

```

REP, REPE, REPZ, REPNE, REPNZ

Prefix REP (který má jiné názvy REPE, REPZ) a prefix REPNE (který má jiný název REPNZ) může být vložen před libovolnou řetězovou instrukci. V tom případě je činnost této instrukce popsána následovně:



Pro instrukce MOV_S/LOD_S/STO_S/IN_S/OUT_S:

```
while (E)CX <> 0 do {MOVS/LODS/STOS/INS/OUTS};
```

Pro instrukce CMPS/SCAS:

```
if (E)CX <> 0 then
  repeat
    {CMPS/SCAS};
  until ((E)CX = 0) or
        (REPE/REPZ and (ZF=0)) or
        (REPNE/REPNZ and (ZF=1));
```

Tyto instrukce příznaky nemění.

Příklad: Porovnání dvou polí.



```
segment data
source resw COUNT
target resw COUNT
segment code
mov cx, COUNT; repetition count
mov si, source
mov di, target
cld      ; direction = forward
repe cmpsw ; repeat while equal
```

SCASB, SCASW, SCASD Instrukce SCASB, SCASW a SCASD porovnávají obsah AL/AX/EAX s hodnotou na adrese ES:DI. Jsou užitečné pro:



- Hledání konkrétního prvku v poli
- Hledání prvního prvku v poli, jehož hodnota se liší

Syntaxe instrukcí:

```
SCASB
SCASW
SCASD
SCAS dst
```

Porovná slabiku/slovo/dvouslovo na adrese ES:(E)DI s obsahem střídače AL/AX/EAX (nastaví příznaky na základě výsledku AL/AX/EAX – dest !) a v závislosti na hodnotě příznaku DF (0/1) zvýší/sníží obsah indexregistru (E)DI o hodnotu 1/2/4 (pro slabiku/slovo/dvouslovo).

Explicitní vyjádření operandu v instrukci SCAS nemá prakticky žádný význam - adresa cílového operandu je vždy dána předpisem ES:(E)DI.

x+y

Příklad: Vyhledání znaku 'F' v řetězci.

```

segment data
alpha DB "ABCDEFGH",0
segment code
mov di, alpha
mov al,'F'    ; search for 'F'
mov cx, 8
cld
repne scasb   ; repeat while not equal
jnz quit
dec di    ; DI points to 'F'

```

STOSB, STOSW, STOSD Instrukce STOSB, STOSW a STOSD uloží obsah AL/AX/EAX do paměti na adresu ES:DI.



Syntaxe instrukcí:

```

STOSB
STOSW
STOSD
STOS dst

```

Uloží obsah střádače AL/AX/EAX na adresu ES:(E)DI a v závislosti na hodnotě příznaku DF (0/1) zvýší/sníží obsah indexregistru (E)DI o hodnotu 1/2/4 (pro slabiku/slovo/dvouslovo).

Explicitní vyjádření operandu v instrukci STOS nemá prakticky žádný význam - adresa cílového operandu je vždy dána předpisem ES:(E)DI.

Příznaky nemění.

LODSB, LODSW, LODD



Instrukce LODSB, LODSW, LODD přečtou hodnotu z adresy DS:SI do AL/AX/EAX.

Syntaxe instrukcí:

```

LODSB
LODSW
LODSD
LODS src

```

Načte slabiku/slovo/dvouslovo z adresy DS:(E)SI do střádače AL/AX/EAX a v závislosti na hodnotě příznaku DF (0/1) zvýší/sníží obsah indexregistru (E)SI o hodnotu 1/2/4 (pro slabiku/slovo/dvouslovo).

Explicitní vyjádření operandu v instrukci LODS umožňuje přepis segmentového registru, indexregistrem zdrojového operandu je vždy registr (E)SI.

Příznaky nemění.

x+y

Pro použití řetězcových operací je třeba provést následující kroky:

- Nastavit zdrojový segment a offset
- Nastavit cílový segment a offset
- Nastavit směr (direction), který je obvykle dopředný
- Specifikovat počet jednotek (bajtů, slov, dvouslov) pro provedení operace
- Provést operaci

Příklad:

```

; Nastavení DS jako cílového segmentu:
mov     ax, ScratchSeg    ; from a defined segment
mov     ds, ax
; Nastavení ES jako cílového segmentu:
mov     ax, 0A000h        ; graphics segment
mov     es, ax
; Nastavení směru:
cld                      ; set direction flag forward
; Zdrojový offset:
mov     esi, ScratchPad   ; set the source offset
; Cílový offset:
xor     edi, edi          ; set to 0
; Specifikovat počet opakování:
mov     ecx, 16000
; Provést operaci:
rep movsd

```



Kontrolní otázky:

- Vysvětlíte účel příznaku DF.
- Vysvětlíte, k čemu slouží parametry dst a src u instrukce „MOVS dst, src“. Je užitečné tyto parametry používat?
- Popište použití prefixu REP.
- U kterých instrukcí má použití prefixu REP smysl?
- Jaký je rozdíl mezi prefixem REP a REPE?
- Jaký je rozdíl mezi prefixem REPE a REPZ?
- Jaký je rozdíl mezi prefixem REPNE a REPNZ?
- Které registry používá instrukce SCASB pro určení adresu svého operandu? Liší se to nějak od jiných řetězcových instrukcí?

- K jakým typům činností může být užitečná instrukce SCASB? Uveďte aspoň dva různé příklady.
- Jakou posloupností jiných instrukcí je možné nahradit instrukci LODSB?
- Jakou posloupností jiných instrukcí je možné nahradit instrukci STOSW?
- Popište obecně posloupnost činností, které je třeba provést pro bezpečné použití řetězcových instrukcí.

5.10 Shrnutí



V této kapitole jsou popsány důležité instrukce vybrané z úplného instrukčního souboru procesorů Pentium a na jednoduchých příkladech je zde ukázáno jejich typické použití. Nezbytně nutná je znalost instrukcí MOV, CBW, LDS, LES, INC, DEC, ADD, SUB, MUL, IMUL, DIV, IDIV, AND, OR, XOR, NOT, SHL, SHR, SAR, RCL, RCR, ROL, ROR, STC, CLC, CMC, STD, CLD, JMP, JC, JO, TEST, JZ, CMP, JE, JB, JA, JL, JG, JCXZ, LOOP, PUSH, POP, PUSHF, POPF, CALL, RET, INT, MOVS, LODS, STOS, SCAC, CMPS, a předpon pro přepis segmentů, předpon pro změnu velikosti operandů a předpon REP, REPE, REPNE.

Prospěšná je i znalost instrukcí XCHG, CWD, XLATB, LEA, ADC, SBB, NEG, STI, CLI, LAHF, SAHF, PUSHA, POPA, IRET i dalších.



10 hod

6 DIREKTIVY

Cílem této kapitoly je seznámit studenty s nejužitečnějšími a nejpoužívanějšími direktivami, které jsou akceptovány překladačem NASM. Jak již bylo vysvětleno v kap. 3 při popisu hypotetického počítače, je direktiva příkazem pro překladač, nikoliv instrukcí pro procesor. Direktivy tedy nejsou překládány do strojového kódu a používají se především používají pro:

- Definování konstant
- Definování místa v paměti pro uložení dat
- Vytváření segmentů paměti
- Vkládání jiných souborů

Direktivy pro definici inicializovaných dat

Pro definici inicializovaných dat se používají direktivy DB (Define Byte), DW (Define Word), DD (Define Doubleword), DQ (Define Quadword) a DT (Define Tenword).

**DB, DW, DD,
DQ, DT**



Příklad použití:

```
db      0x55                      ; bajt 0x55
db      0x55,0x56,0x57            ; tři bajty
db      'a',0x55                  ; znak
db      'hello',13,10,'$'        ; řetězec
dw      0x1234                    ; 0x34 0x12
dw      'a'                       ; 0x41 0x00 (číslo)
dw      'ab'                      ; 0x41 0x42 (dva znaky)
dw      'abc'                    ; 0x41 0x42 0x43 0x00 (řetězec)
dd      0x12345678                ; 0x78 0x56 0x34 0x12
dd      1.234567e20              ; single-precision
dq      1.234567e20              ; double-precision
dt      1.234567e20              ; extended-precision
```

Direktivy pro definici neinicializovaných dat

Pro definici neinicializovaných dat se používají direktivy RESB (Reserve Bytes), RESW (Reserve Words), RESD (Reserve Doublewords), RESQ (Reserve Quadwords) a REST (Reserve Tenwords). Operand těchto direktiv udává počet položek, rezervovaných v paměti.

**RESB, RESW,
RESD, RESQ,
REST**



Příklad:

```
buffer:      resb      64          ; rezervuj 64 bajtů
wordvar:     resw      1           ; rezervuj slovo
realarray    resq      10         ; pole reálných čísel
```

INCLUDE

Direktiva `%INCLUDE` slouží k tomu, abychom mohli vložit do zdrojového souboru jiný textový soubor.

Syntaxe direktivy:

```
%include "macros.mac"
```

Direktiva ve výše uvedeném příkladě vloží obsah souboru `macros.mac` do zdrojového souboru, který obsahuje direktivu `%include`.

EQU

Direktiva `EQU` přiřazuje symbolu (který je návěštím na řádku s direktivou `EQU`) konstantní hodnotu, která je dána operandem direktivy `EQU`. Tato definice je konečná a nelze ji později změnit.

V následujícím příkladu:

```
message          db      'hello, world'
msglen           equ     $-message
```

definuje symbol `msglen` s hodnotou 12.

TIMES

Prefix `TIMES` způsobí, že následující instrukce nebo direktiva je přeložena /provedena několikrát. V následujícím příkladu

```
zerobuf:         times 64 db 0
```

je direktiva `TIMES` použita s konstantou 64. Argument direktivy `TIMES` však nemusí být konstanta, ale libovolný číselný výraz. Je tedy možné zapsat obrat

```
buffer: db        'hello, world'
          times 64-$+buffer db ' '
```

pomocí kterého bude vygenerováno tolik mezer, aby buffer měl délku přesně 64 bajtů. Direktivu `TIMES` lze také aplikovat na běžné instrukce, lze tedy zapsat

```
times 100 movsb
```

Mezi obraty „`times 100 resb 1`“ a „`resb 100`“ není žádný rozdíl vyjma toho, že druhý obrat je přeložen rychleji.



Podrobnou dokumentaci direktiv překladače NASM naleznete na webové stránce předmětu: <https://www.fit.vutbr.cz/study/courses/IAS/private/>

**Kontrolní otázky:**

- Vysvětlete rozdíl mezi instrukcí a direktivou.
- Vyjmenujte datové typy podporované překladačem NASM, jejich velikosti a odpovídající instrukce pro definici inicializovaných dat těchto typů.
- Vyjmenujte datové typy podporované překladačem NASM, jejich velikosti a odpovídající instrukce pro definici neinicializovaných dat těchto typů.
- Vysvětlete rozdíl mezi direktivou DB a EQU.
- Uveďte příklad použití direktivy INCLUDE.
- Vysvětlete rozdíl mezi použitím direktivy TIMES a instrukce LOOPZ.

6.1 Shrnutí

V této krátké kapitole jsme se seznámili s direktivami pro definici konstant (DB, DW, DD, DQ, DT), vyhrazení místa v paměti (RESB, RESW, RESD, RESQ, REST) a direktivami %INCLUDE, EQU a TIMES.



10 hod

7 MAKRA

Cílem této kapitoly je seznámit studenty s nejužitečnějšími a nejpoužívanějšími makry, která jsou akceptována překladačem NASM. Makro (někdy nazývané též makropříkaz nebo makrodefinice) je mechanismus, kterým můžeme nadefinovat posloupnost příkazů, vykonávající určitou činnost a tuto posloupnost příkazů později vložit na různá místa programu.

Překladač NASM zná dvojí typ maker – jednořádková makra a víceřádková (pravá) makra.

7.1 Jednořádková Jednořádková makra makra

Jednořádková makra jsou analogická příkazu `#define`, tak jak ho známe z jazyka C. Struktura jednořádkového makra je následující:

```
%define jméno[(parametry)] tělo
```

Klíčové slovo „`%define`“ určuje definici jednořádkového makra. Symbol „jméno“ definuje jméno makra, kterým budeme později volat makro v programu. Pak mohou následovat v závorce nepovinné parametry. Řetězec „tělo“ obsahuje text, který bude vložen na místo, kde bude makro zavoláno. Pokud má makro i parametry, budou v těle jména parametrů nahrazena skutečnými hodnotami.



Příklad: Nadefinujme a použijme makro „`crlf`“, které bude sloužit pro vložení znaků pro odřádkování

Příklad definice makra:

```
%define crlf 13,10
```

Volání nadefinovaného makra:

```
mess db 'hello', crlf, '$'
```

Překlad makra:

```
mess db 'hello', 13, 10, '$'
```



Příklad definice makra s parametry:

```
%define param(b,i,d) [b+i+d]
```

Volání nadefinovaného makra:

```
mov ax,param(bx,si,10)
```

Překlad makra:

```
mov ax,[bx+si+10]
```

7.2 Víceřádková Víceřádková makra makra

Víceřádková (pravá) makra se skládá z příkazu definice makra a z příkazu použití makra. Syntaxe definice makra je následující:

```
%macro jméno počet_parametrů
    tělo definice
```

```
%endmacro
```

Pokud má makro parametry, jsou tyto parametry uvnitř těla definice označovány symboly %1, %2, %3 atd., kde číslo za znakem % určuje pořadí parametru. Tyto parametry budou při použití makra nahrazeny skutečnými hodnotami parametrů.

x+y

Příklad definice makra:

```
%macro prologue 1
    push bp
    mov bp,sp
    sub sp,%1
%endmacro
```

Volání nadefinovaného makra:

```
myproc: prologue 10
```

Překlad makra:

```
myproc: push bp
        mov bp,sp
        sub sp,10
```

Uvnitř makra je možné použít tzv. lokální návěští. Symbol lokálního návěští začíná dvojicí znaků %%. Při rozgenerování makra je takovéto lokální návěští nahrazeno jedinečným nově vytvořeným návěštím tak, aby se při vícenásobném použití makra zabránilo vícenásobným definicím stejného návěští.

x+y

Příklad definice makra s lokálním návěštím:

```
%macro retz 0
    jnz %%skip
    ret

%%skip:                ; lokální návěští pro každé
%endmacro               ; volání makra
```

Volání nadefinovaného makra:

```
retz
...
retz
```

Překlad makra:

```
    jnz  ..@0001:
    ret
..@0001:
...
    jnz  ..@0002:
    ret
..@0002:
```

Mechanismus hltavých (greedy) parametrů je jeden ze způsobů, jak vytvořit makro s proměnným počtem parametrů. Chceme-li použít hltavé parametry,

x+y

vložíme při definici makra za počet parametrů znak +. Pokud je například počet parametrů 2+, uvnitř těla makra můžeme použít symbol %1, který znamená první parametr a symbol %2, který označuje konkatenci druhého a všech dalších parametrů. Následuje ukázka.

Příklad definice makra:

```
%macro write_to_file 2+
    jmp %%endstr
%%str:    db %2
%%endstr:
    mov dx, %%str
    mov cx, %%endstr - %%str
    mov bx, %1
    mov ah, 40h
    int 21h
%endmacro
```

Volání nadefinovaného makra:

```
write_to_file [filehandle], "hello students",13,10
               -----
               %1                %2
```

Pokud bychom chtěli, abychom některé parametry makra nemuseli při některých voláních makra zadávat a parametry se doplnily samy, můžeme použít tzv. implicitní (default) parametry. Definice makra pak má následující syntaxi

```
%macro jméno min-max [(max-min) předdefinovaných parametrů]
```

kde za jméno makra je třeba zadat minimální a maximální počet zadávaných parametrů a za tento počet (max-min) implicitních parametrů. Použití si ukážeme na příkladu:

x+y

Příklad definice makra:

```
%macro message 0-1+ "All OK",10,13, '$'
    write_to_file 2, %1 ; jiné makro
    mov ah, 4ch
    int 21h
%endmacro
```

Volání nadefinovaného makra:

```
message "Error No1",10,13, '$'
```

Jiný způsob volání nadefinovaného makra:

```
message
```

Speciálním případem makra je makro pro opakování příkazů rep. Jeho syntaxe je následující:

```
%rep počet_opakování
    tělo makra
```

```
%endrep
```

Toto makro slouží k tomu, abychom mohli několikrát po sobě vložit tělo makra do textu programu. Pokud chceme vložené texty například očíslovat, může být užitečné použít příkaz %assign, který přiřadí symbolu hodnotu.

```
%assign jméno numerická_hodnota
```

Přiřazení se provádí během překladač a symbolu můžeme hodnotu přiřadit opakovaně.

Následující příklad ukazuje kombinované použití obou výše uvedených příkazů.

x+y

Příklad použití makra rep:

```
%assign i 0
%rep 5
    dw i
    %assign i i+1
%endrep
```

Překlad makra:

```
dw 0
dw 1
dw 2
dw 3
dw 4
```

Posledním způsobem, jak vytvořit makro s proměnným počtem parametrů, jsou tzv. rotující parametry. Definice makra má pak syntaxi

```
%macro jméno 1-*
```

kde znak * zastupuje libovolný (předem nedefinovaný) počet parametrů. Protože počet parametrů neznáme, je k dispozici symbol %0, který vrací skutečný počet parametrů zadaných při volání makra. Pomocí příkazu

```
%rotate počet
```

můžeme zadané parametry rotovat o zadaný počet pozic. Kladná hodnota „počet“ způsobí rotaci parametrů doleva, záporná způsobí rotaci doprava o odpovídající počet pozic. Příklad použití si ukážeme na makrech multipush a multipop pro uschování několika registrů do zásobníku a jejich obnovení.

x+y

Příklad definice makra:

```
%macro multipush 1-*
    %rep %0
        push %1
        %rotate 1
    %endrep
%endmacro
```

Volání nadefinovaného makra:

```
multipush si,di,ds,es
```

x+y

Příklad definice makra:

```
%macro  multipop  1-*
    %rep  %0
    %rotate  -1
        pop  %1
    %endrep
%endmacro
```

Volání nadefinovaného makra:

```
multipop  si,di,ds,es
```

7.3 Podmíněný překlad

Podmíněný překlad

Direktivy pro podmíněný překlad umožňují, abychom na základě splnění některých podmínek mohli přeložit pouze některé části programu a některé jiné ignorovat. Struktura těchto direktiv připomíná konstrukce if, if-else a if-elseif-else, známé z vyšších programovacích jazyků.

```
%if  <condition_1>
    ; zdrojový kód, který se přeloží při splnění podmínky 1
[%elif <condition_2>
    ; zdrojový kód, který se přeloží při nesplnění podmínky 1
    ; a při současném splnění podmínky 2]
[ .... ]
[%elif <condition_n>
    ; zdrojový kód, který se přeloží při nesplnění
    ; předchozích podmínek a současném splnění podmínky n]
[%else
    ; zdrojový kód, který se přeloží při současném nesplnění
    ; všech předcházejících podmínek]
%endif
```

Uvnitř podmínky je možno používat následující logické operátory:

=	nebo	= =	
<			
>			
<=			
>=			
<>	nebo	!=	
&&			and
^^			xor
			or

Podmínka může obsahovat symboly, u kterých je jejich nenulová hodnota považována za logickou hodnotu TRUE.

**Kontrolní otázky:**

- Jaký je rozdíl mezi makrem a procedurou (podprogramem)?
- K čemu jsou vhodné lokální symboly v makru?
- Jaký jsou vlastnosti makra a procedury (podprogramu) z pohledu rychlosti?
- Jaký jsou vlastnosti makra a procedury (podprogramu) z pohledu velikosti výsledného programu?
- Jaký je význam default parametrů makra?
- Jaký je význam rotujících parametrů makra?
- Jaký je význam greedy (hltačích) parametrů makra?
- Jak se používá makro pro opakování?
- K čemu slouží podmíněný překlad?

7.4 Shrnutí

V této kapitole byla popsána nejpoužívanější makra akceptovaná překladačem NASM. Z jednořádkových maker bylo popsáno důležité makro %DEFINE, z víceřádkových maker, kterých bylo popsáno více, bude požadována znalost definice klasických maker %MACRO, ..., %ENDMACRO s předem daným počtem parametrů a maker REP a ASSIGN.



5 hod

8 Asembler NASM

Cílem této kapitoly je stručný popis překladače NASM, sestavujícího programu LINK a příkazového souboru RUN.BAT.

NASM

Pro překlad programu v assembleru budeme používat assembler, který se jmenuje NASM (Netwide Assembler), který je volně dostupný. NASM je program, který je volán z příkazového řádku a parametry jsou programu předávány také pomocí příkazového řádku. Syntaxe příkazového řádku je

```
nasm [-parametr hodnota] <zdrojový soubor> [-parametr hodnota]
```

Hodnoty parametrů je možno nalézt v dokumentaci k programu NASM. Nejdůležitější parametry jsou tyto:

- f specifikace formátu výstupního souboru (obj, win32, elf, atd.)
- l aktivace výpisu protokolu o překladu do souboru, jehož jméno je uvedeno bezprostředně za tímto parametrem
- o specifikace výstupního souboru (není-li tento parametr uveden, použije NASM implicitní jméno – např. pro vstupní soubor example.asm a formát obj bude mít výstupní soubor implicitní jméno example.obj)

Příklad zavolání programu NASM:

```
nasm.exe -f obj -l example.lst example.asm
```

V tomto příkladu bude překladač zpracovávat soubor „example.asm“, protokol o překladu bude v souboru „example.lst“ a výstupní formát bude „obj“ (který bude dále zpracován programem LINK).

LINK

Po překladu programu provedeme sestavení sestavovacím programem LINK. Program LINK je opět volně dostupný. Tento program je volán z příkazového řádku a má několik parametrů, avšak v našich příkladech bude aspoň z počátku postačující zjednodušené volání, kde jediným parametrem programu LINK bude jméno přeloženého souboru s příponou .obj:

```
link.exe example.obj
```

nebo ve složitější variantě

```
link.exe example.obj, example.exe, example.map
```

kde je specifikováno také jméno výstupního souboru (example.exe) a jméno

souboru se sestavovací mapou (example.map).

RUN.BAT

Pro snazší překládání, sestavování a spouštění programů je připraven příkazový soubor RUN.BAT, který činnost podstatně automatizuje. Pro přeložení, sestavení a spuštění programu pak stačí napsat

```
run.bat example
```

Je třeba dát pozor na to, že jméno vstupního souboru se zadává bez přípony!

x+y

Nejdůležitější část příkazového souboru RUN.BAT:

... Přeskočme úvodní kontroly

```
echo Překlad souboru "%1.asm" pomocí NASM.EXE.
nasm -f obj %1.asm -l %1.lst
```

```
if not exist %1.obj goto compile_err
if not exist LINK.EXE goto no_linker
```

```
echo Sestavování pomocí LINK.EXE:
link %1.obj,%1.exe,%1.map,,nul.def
```

```
if not exist %1.exe goto link_err
```

```
echo Spuštění %1.EXE:
%1.EXE
```

```
echo Program správně ukončen, řízení vráceno operačnímu
systému.
goto exit
```

... Přeskočme výpisy chyb

Všimněme si, že v příkazovém souboru RUN.BAT jsou k parametru %1, který obsahuje jméno překládaného souboru doplňovány různé přípony, což vysvětluje, proč je třeba zadávat jméno vstupního souboru bez přípony.

?

Kontrolní otázka:

Jak přeložíte, sestavíte a spustíte program pokus.asm:

- bez použití příkazového souboru run.bat
- s použitím příkazového souboru run.bat

8.1 Shrnutí

Σ

V této kapitole byly popsány základní principy práce s překladačem NASM, sestavujícím programem LINK a příkazovým souborem RUN.BAT – znalost všech uvedených informací je nezbytně nutná pro úspěšný překlad, sestavení a spuštění každého uživatelského programu.

Literatura

Literatura

- [1] Bradley D., J.: Assembly Language Programming for the IBM Personal Computer, ruský překlad, Radio i svjaz, Moskva, 1988
- [2] Irvine K., R.: Assembly Language for the IBM-PC, Macmillan Publishing Company, New York, 1989
- [3] Strauss E.: 80386 Technical Reference, A Brady Book, New York, 1987
- [4] Zbořil F.: Mikroprocesorová technika, skriptum VUT, ES VUT Brno, 1990
- [5] Zbořil F.: Strojově orientované jazyky - Jazyk symbolických instrukcí osobních počítačů IBM, skriptum VUT, ES VUT Brno, 2003
- [6] i486 Microprocessor, Intel Literature 1991