12-13. Překladač asembleru a linker

Filip Orság

Příkazový řádek

`CMD.EXE`

- Konzola = program OS Windows (ale i Linuxu = terminál)
- CLI (*Command Line Interface*) = UI pro komunikaci příkazy v příkazovém řádku (zapomeňte na myš a různé „vychytávky“)
- Naleznete nejčastěji zde:

Start → Všechny programy → Příslušenství → Příkazový řádek

- Příkazy většinou ve formě:

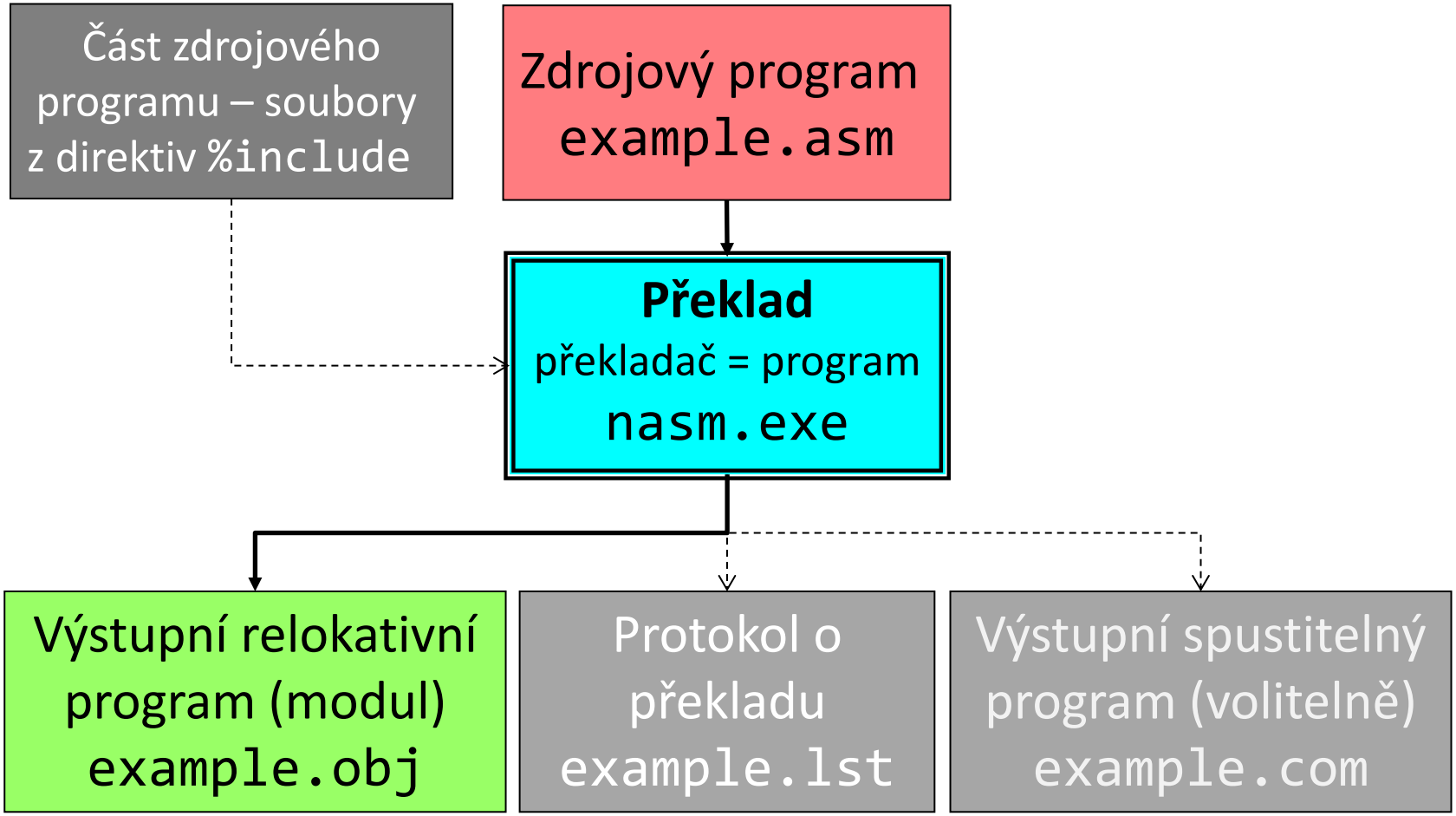
```
PŘÍKAZ [volby] [soubory]
```

- Přesměrování výstupu obrazovky do souboru:

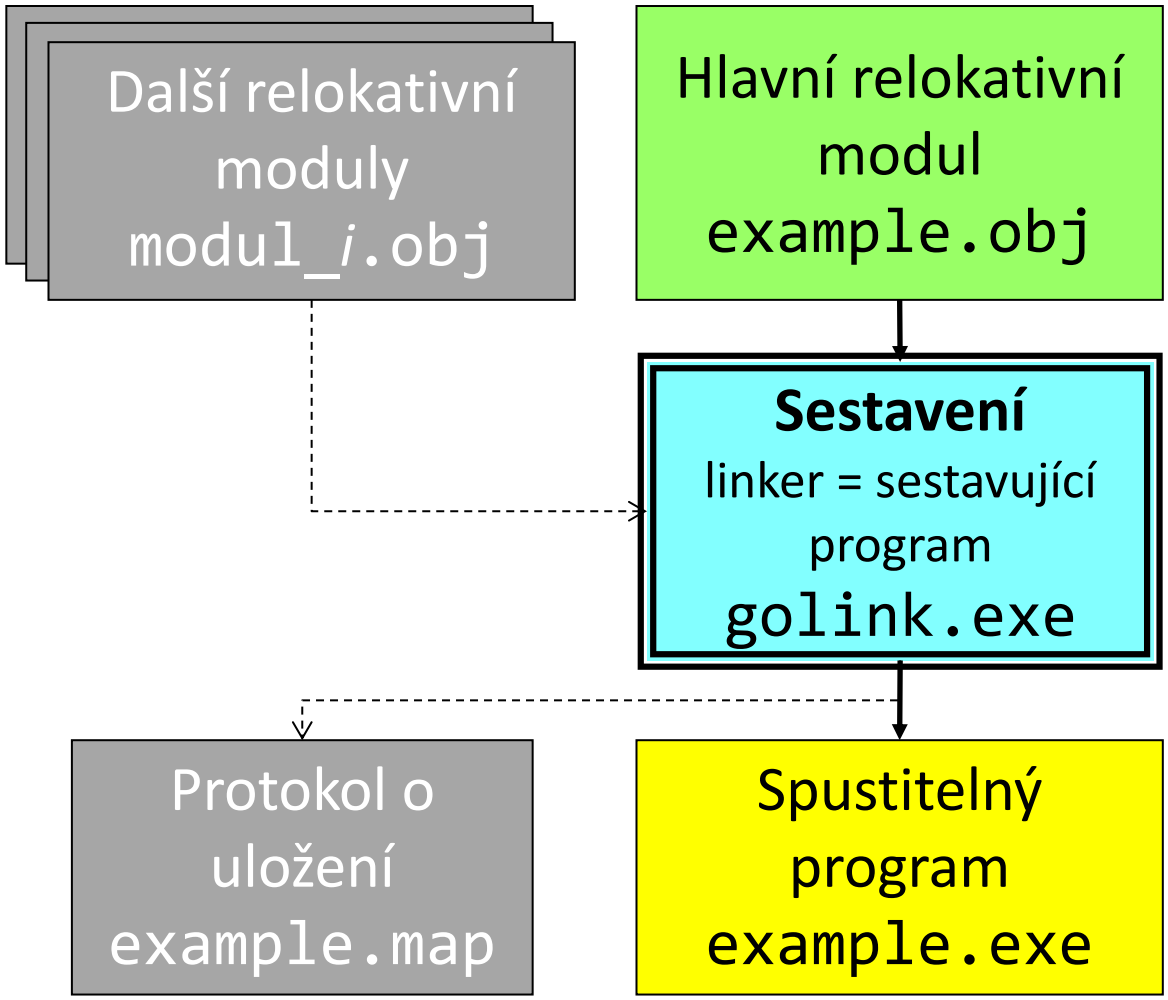
```
PŘÍKAZ [volby] [soubory] > muj_soubor.txt
```

PŘEKLAD
SESTAVENÍ
SPUŠTĚNÍ

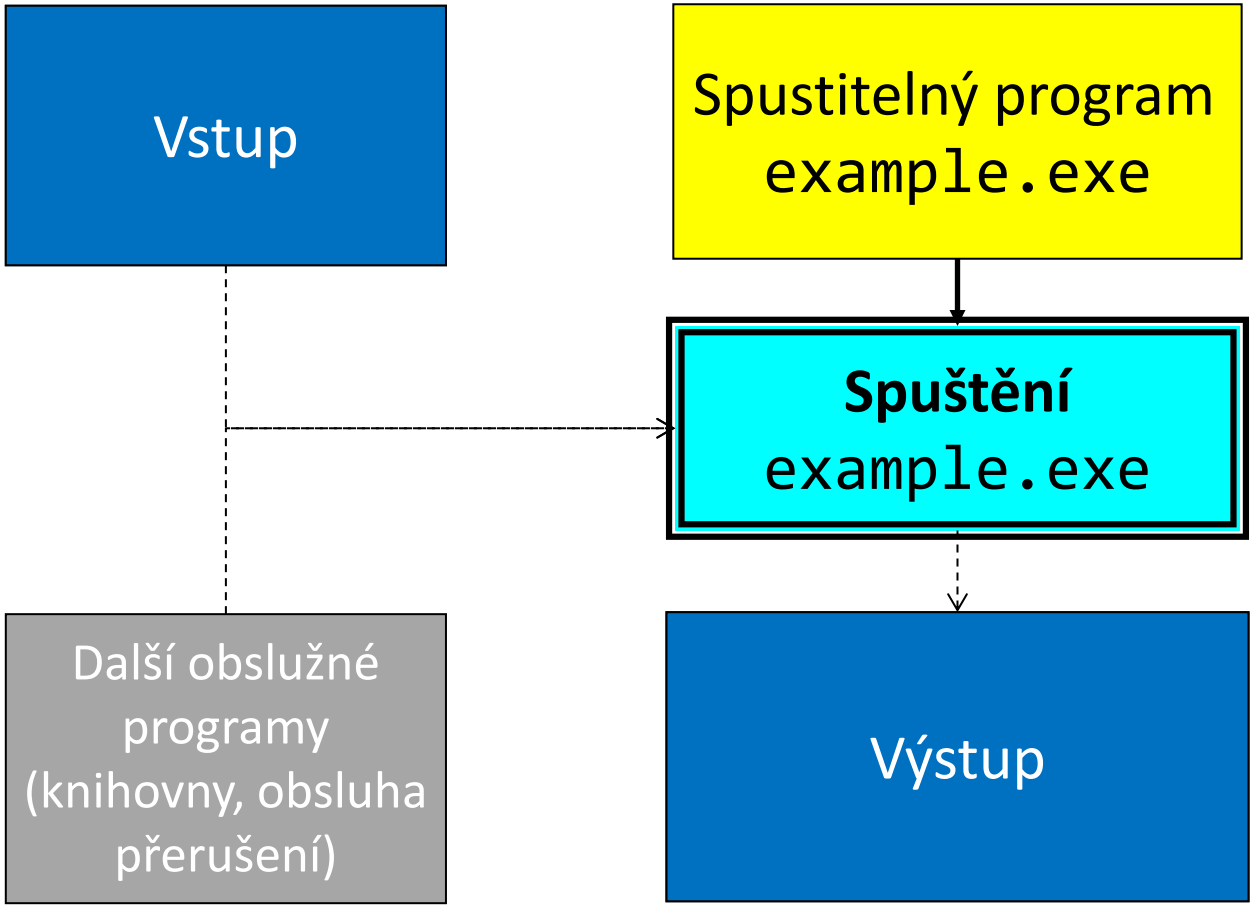
Překlad


Část zdrojového programu – soubory z direktiv %include
Zdrojový program example.asm
Překlad
překladač = program nasm.exe
Výstupní relokativní program (modul) example.obj
Protokol o překladu example.lst
Výstupní spustitelný program (volitelně) example.com

Sestavení


Další relokativní moduly
modul_i.obj
Hlavní relokativní modul
example.obj
Sestavení
linker = sestavující program
golink.exe
Protokol o uložení
example.map
Spustitelný program
example.exe

Spuštění:


Vstup
Spustitelný program
example.exe
Spuštění
example.exe
Další obslužné programy (knihovny, obsluha přerušení)
Výstup

PŘEKLADAČE
SESTAVUJÍCÍ PROGRAMY

Překladače (asemblery):

Pro architektury x86:

- TASM Turbo Assembler (Borland)
- MASM Micro Assembler (Microsoft)
- GoASM Go Assembler (Jeremy Gordon)
- FlatASM Flat Assembler (Tomasz Grysztar)
- gas GNU Assembler (projekt GNU)

- **NASM Netwide Assembler** (skupina vývojářů)

Licence:	BSD
Původní autoři:	Simon Tatham, Julian Hall
Dostupné na:	http://www.nasm.us

NASM - příkazový řádek:

NASM [-*parametr hodnota*] *zdrojový_soubor*

-o *jmeno_souboru*

jmeno_souboru = jméno **výstupního** souboru, není-li tento parametr uveden, použije NASM implicitní jméno (např. vstupní soubor example.asm a formát obj bude mít výstupní soubor implicitní jméno example.obj)

-f *format*

format specifikuje formát přeloženého souboru různé typy formátů, seznam získáme příkazem NASM –hf, implicitní je formát bin, my budeme využívat formát obj pro 16/32 bitový OMF formát

-e

provede pouze předzpracování souboru =odstraní komentáře, rozvine makra, odstraní direktivy

-l *jmeno_souboru*

aktivace výpisu protokolu o překladu do souboru *jmeno_souboru* uvedeného bezprostředně za tímto parametrem

Typická příkazová řádka:

```
NASM –fobj –o example.obj -l example.lst example.asm
```

Sestavující programy (linkery):

- TLINK Turbo Linker (Borland)
- LINK Linker (Microsoft)
- ALINK a Linker (A. A. J. Williams)

- **GoLink Go Linker** (Jeremy Gordon)

Licence: free
„The author has made every effort to ensure that the output of GoLink is accurate, but you do use it entirely at your own risk.“
Autor: Jeremy Gotdon
Dostupné na: http://www.godevtool.com

GoLink – příkazový řádek

GoLink [*volby*] *soubor* [*další soubory*]

/fo *jmeno_souboru*

jmeno_souboru je název cílového souboru

/entry *symbol*

symbol je návěští, na které se skočí při spuštění programu

/dll

výstup bude ve formátu DLL

/console

nastaví, jaký cílový subsystém konzolu (okno programu CMD.EXE)

Typická příkazová řádka:

```
GoLink /console /fo app.exe /entry start m1.obj m2.obj msvcrt.dll
```

NASM

Symbolické instrukce
Pseudoinstrukce
Direktivy
Výrazy a operátory
Paměťové operandy

Symbolické instrukce v NASM (I)

Formát zápisu instrukce:

```
[[.]návěští:] [instrukce [operandy]] [; komentář]
```

Globální návěští `[návěští:]`

- globální, nepovinné, unikátní identifikátor, který reprezentuje adresu první slabiky instrukce nebo dat následujících bezprostředně za ním,
- rozlišuje se velikost písmen, kromě písmen může identifikátor obsahovat i číslice a znaky: **`_ $ # @ ~ . ?`**
- identifikátor může začínat pouze písmenem nebo znaky: **`. _ ? @`**

Lokální návěští `[.návěští:]`

- začíná tečkou (např.: `.cyklus`), je platné od předcházejícího globálního návěští do následujícího globálního návěští, mimo tento rámec se lze na toto návěští odkázat úplným názvem (např.: `globalni1.cyklus`)

```
globalni1:          JMP .lokalni
.lokalni:           JMP globalni1
globalni2:          JMP .lokalni
.lokalni:           JMP globalni1.lokalni
```

Symbolické instrukce v NASM (II)

`[; komentář]`

- nepovinný,
- vše za znakem ; je ignorováno.

`[instrukce` `[operandy]`**`]`**

- nepovinná,
- operandy jsou povinné pro instrukce s operandy,
- instrukce je
    - symbolické jméno instrukce (např. `MOV`, `PUSH` apod.),
    - pseudoinstrukce (např. `DB`, `DW`, `EQU`, `TIMES` apod.),
    - makro
- pokud má operandy, uvedou se za instrukci (závisí na konkrétní instrukci, např. `MOV  EAX,10`).

Symbolické instrukce v NASM (III)

- NASM podporuje všechny instrukce procesorů x86 a x86-64, včetně instrukcí FPU, MMX, SSE, 3DNow, AES, AVX a některé další.
- NASM si neukládá/nepamatuje velikosti proměnných (datové typy) => nepřeloží instrukce XLAT a řetězové instrukce MOVS, LODS, CMPS atd. => je nutné používat výhradně instrukce s explicitním vyjádřením typu (např. XLATB, MOVSB/MOVSW/MOVSD, atd.)
- Případnou změnu segmentového registru zdrojového operandu lze „vnutit“ zápisem tohoto registru před instrukci, například:

```
ES LODSB
```
(v 32bitovém režime nepoužívejte)

- u instrukcí FPU označuje registry FPU jako `st0, st1`, atd.

Pseudoinstrukce v NASM

- Nejsou přímo instrukce procesoru, ale i přesto se vkládají do instrukčních polí.

Dostupné pseudoinstrukce

- Definice inicializovaných dat:
  - `DB, DW, DD, DQ, DT, DO, DY`
- Definice neinicializovaných dat:
  - `RESB, RESW, RESD, RESQ, REST, RESO, RSY`
- Vložení externího binárního souboru (například obrázek apod.):
  - `INCBIN`
- Definice konstant:
  - `EQU`
- Opakování instrukce nebo definice dat:
  - `TIMES`

Definice konstant (EQU)

- Jméno konstanty JE povinné.
- Za EQU následuje konkrétní hodnota konstanty.
- Definovaná konstanta nemůže být v jednom programu předefinována.
- Konstanta nezabírá místo v paměti, je to pouze hodnota použitá při překladu.

```
dve EQU 2

message     DB  ‘Hello, students!’
message_len EQU  $ - message
```

Symbol `$` zastupuje aktuální adresu v paměti, kde je daný symbol použit, t.j. adresu symbolu `message_len`.

Definice dat

Pseudoinstrukce pro definici inicializovaných dat:

`DB, DW, DD, DQ, DT, DO, DY`

Pseudoinstrukce pro definici neinicializovaných dat:

`RESB, RESW, RESD, RESQ, REST, RESO, RESY`

	Význam	bytů	bitů	Odpovídající datový typ v jazyce C
DB, RESB	Define/REServe Byte	1	8	char
DW, RESW	Define/REServe Word	2	16	short
DD, RESD	Define/REServe Double-word	4	32	long, int, float
DQ, RESQ	Define/REServe Quad-word	8	64	double
DT, REST	Define/REServe Ten-byte	10	80	long double
DO, RESO	Define/REServe Octal-word	16	128	někdy long double
DY, RESY	Define/REServe YMM-word	32	256	

Definice dat – příklady (Dx)

Návěští NENÍ povinné u definice dat. Za Dx musí následovat konkrétní hodnota nebo seznam hodnot oddělených čárkou.

```
        DB   -43              byte – dekadicky se znaménkem
        DB   01001011b        byte – binárně
        DB   0xA2, 0A2h       byte – hexadecimálně
alpha   DB   'hello',13,10,'$'      ; řetězec znaků = posloupnost bytů
        DW   37543            word – dekadicky
        DW   0xFFFF           word – hexadecimálně (totožné s DW -1)
        DD   12876            double-word – dekadicky
        DD   12345678h        double-word – hexadecimálně
        DD   alpha            double-word = adresa symbolu alpha
        DD   1.789            double-word – reálné číslo, jednoduchá přesnost,
                                            datový typ float
        DQ   2.71e15          quad-word – reálné číslo, dvojitá přesnost,
                                            datový typ double
        DT   1.56e-15         ten-byte – reálné číslo, rozšířená přesnost,
                                            datový typ long double
```

Definice dat – příklady (RESx)

Návěští NENÍ povinné u definice dat. Za RESx nenásleduje konkrétní hodnota, ale počet položek datového typu x rezervovaných v paměti. Obsah rezervované paměti není definován = může tam být cokoliv.

počet rezervovaných bytů = velikost datového typu x * počet položek

`RESB 10`	rezervuje místo pro 10 bytů = 10*1 = 10 bytů
`RESW 5`	rezervuje místo pro 5 wordů = 5*2 = 10 bytů
`RESD 5*dve`	rezervuje místo pro 5*dve double-wordů = 5*dve*4 = 40 bytů (pokud dve = 2), výraz dve musí být při překladu vyčíslitelný, tj. definovaný například pomocí makra nebo pseudoinstrukce EQU, před jeho použitím
`RESQ 12`	rezervuje místo pro 12 quad-wordů = 12*8 = 96 bytů
`REST 2`	rezervuje místo pro 2 ten-byty = 2*10 = 20 bytů

Struktury

```
struc jméno_struktury
    .prvek1: RESx počet
    .prvek2: RESx počet
    …
    .prvekN: RESx počet
endstruc
```

Deklarace instancí struktury:

```
návěští:
istruc jméno_struktury
    at .prvek1, Dx hodnota
    at .prvek2, Dx hodnota
    …
    at .prvekN, Dx hodnota
iend
```

Použití:

```
MOV EAX,[EDI + jméno_struktury.prvek1]
```

Příklady

```
    struc auto
        .typ:   RESB 2
        .rok:   RESW 1
        .km:    RESD 1
        .nazev: RESB 64
    endstruc
```

```
Konstanty:

    auto.typ   … 0
    auto.rok   … 2
    auto.km    … 4
    auto.nazev … 8
```

```
moje_bmw:
    istruc auto
      at auto.typ,   DB 'OS'
      at auto.rok,   DW 2001
      at auto.km,    DD 128000
      at auto.nazev, DB "BMW",0
    iend
```

```
moje_bmw:

    DB 'OS'
    DW 2001
    DD 128000
    DB "BMW",0
```

```
MOV EAX,[moje_bmw + auto.rok]

MOV [moje_bmw + auto.km], dword 500
```

```
MOV EAX,[moje_bmw + 2]

MOV [moje_bmw + 4], dword 500
```

Opakování instrukce nebo definic dat (TIMES)

```
[návěští:]    TIMES N {instrukce|definice dat}
```

Tato pseudoinstrukce vloží *N*-krát *instrukci* nebo *definici dat* v době překladu (NEVYKONÁVÁ instrukci, pouze ji vloží na určené místo).

```
zerobuf:    TIMES 3 DB 0          vloží 3× pseudoinstrukci DB 0
                                  téhož dosáhneme použitím RESB 3,
zerobuf:    DB 0                  ale hodnota v paměti nebude
            DB 0                  definovaná, pomocí TIMES N
            DB 0                  zajistíme naplnění paměti požadovanou
                                  hodnotou (v tomto případě 0)
    …
            TIMES 2 INC EAX       vloží 2× instrukci INC EAX

            INC EAX
            INC EAX
```

Vložení binárního souboru (INCBIN)

INCBIN "*jméno souboru*"[, SKIP, LIMIT]

Tato pseudoinstrukce vloží na místo, kde se vyskytuje, obsah binárního souboru, jehož jméno je za ní uvedeno v uvozovkách.

Hodí se pro vložení obrázku, zvukových souborů, apod.

`INCBIN "file.dat"`	vloží celý soubor
`INCBIN "file.dat",1024`	vynechá prvních SKIP=1024 bytů souboru
`INCBIN "file.dat",1024,512`	vynechá prvních SKIP=1024 bytů souboru a vloží nejvýše LIMIT=512 bytů

Direktivy

- Řídí chod překladu programu.

Dostupné direktivy jsou

- Volba módu procesoru pro překlad
  - `BITS`
- Definice segmentu
  - `SECTION, SEGMENT`
- Definice externích a globálních návěští
  - `EXTERN, GLOBAL, COMMON`
- Další direktivy (nebudeme probírat)
  - Pevně definovaná adresa návěští v paměti
    - `ABSOLUTE`
  - Definice konkrétního typu procesoru
    - `CPU`
  - Nastavení chování koprocesoru
    - `FLOAT`

Direktiva BITS

`[BITS xx]`

- hodnota *xx* udává mód procesoru: 16-, 32- nebo 64-bitový (my budeme používat 32bitový, tedy `[BITS 32]`, je nastaveno v souboru rw32-xxxx.inc)
- nepřepíná procesor do vybraného režimu, pouze přeloží soubor pro daný režim
- direktiva se vkládá nejčastěji na začátek zdrojového souboru
- existuje makro `BITS xx`, které lze použít místo `[BITS xx]`
- `[BITS 16]` má alias `USE16`
- `[BITS 32]` má alias `USE32`

`BITS 16`	použijeme makro, které vloží direktivu
`[BITS 16]`	použijeme direktivu
`BITS 32`	použijeme makro, které vloží direktivu
`USE32`	použijeme alias, který vloží direktivu

Direktiva SECTION/SEGMENT

[SEGMENT *jméno*] nebo [SECTION *jméno*]

- SECTION i SEGMENT je totéž.
- Pro jména segmentů platí stejná omezení, jako pro ostatní jména.
- Některé formáty vyžadují určitý omezený počet segmentů s přesně danými jmény (například .TEXT, .DATA, .BSS).
- Existují makra SECTION *jméno* (bez hranatých závorek) a SEGMENT *jméno*, která se chovají podobně, ale navíc definují makro __SECT__, které lze využít v dalších makrech jako referenci na aktuální sekci/segment.

Rozšíření direktivy SECTION pro výstupní formát OBJ (I)

```
[SECTION jméno [combine] [ALIGN=A] [CLASS=C] [use]]
[SEGMENT jméno [combine] [ALIGN=A] [CLASS=C] [use]]
 SECTION jméno [combine] [ALIGN=A] [CLASS=C] [use]
 SEGMENT jméno [combine] [ALIGN=A] [CLASS=C] [use]
```

Parametry **combine**, **ALIGN**, **CLASS** a **use** jsou volitelné, když je nepoužijeme, vybere si NASM sám implicitní hodnoty.

- **combine** informuje NASM, jakým způsobem se má chovat k sekcím:
  - PRIVATE privátní segment není spojen s žádným jiným
  - PUBLIC segmenty stejného jména se spojí v jediný
  - COMMON segmenty stejného jména se překryjí
  - STACK segmenty stejného jména se spojí v jediný
- **ALIGN=*A*** určuje umístění segmentu (tzv. zarovnání) :
  - počáteční adresa segmentu musí být beze zbytku dělitelná zadaným číslem *A*,
  - *A* = 1, 2, 4, 16, 256, 4096,
  - jiné hodnoty budou zaokrouhleny na nejbližší vyšší povolenou hodnotu,
  - zarovnání na hodnotu 4096 není podporováno všemi sestavujícími programy

Rozšíření direktivy SECTION pro výstupní formát OBJ (II)

- **CLASS=*C*** informuje NASM, jaká je třída dané sekce, přičemž sekce stejné třídy jsou při sestavení uloženy blízko sebe, jméno sekce *C* může být jakékoliv povolený název.
- **use** určuje implicitní chování při překladu dané sekce
  - na výběr je buď USE16 nebo USE32

Příklady:

```
SECTION .data PRIVATE ALIGN=16
[SECTION STACK CLASS=stack]
```

Přednastavené (*default*) atributy sekcí jsou:

PUBLIC, ALIGN=1, USE16, třída není definována (*no class*)

Direktivy GLOBAL, EXTERN, COMMON

EXTERN *jméno*

- Definuje symbol *jméno* jako externí (tj. vyskytuje se někde jinde – jiný modul, případně DLL knihovna – sestavující program poté musí symbol najít).
- Pro dané jméno lze použít i vícekrát (NASM bude další výskyty stejného jméno deklarovaného direktivou EXTERN ignorovat)

GLOBAL *jméno*

- Definuje symbol *jméno* jako globální (tj. viditelný pro všechny moduly, které se budou podílet na sestavení).

COMMON *jméno velikost*

- Definuje symbol *jméno* jako globální v segmentu neinicializovaných dat (v sekci, která se jmenuje `.bss`), která v paměti zabírá *velikost* bytů.
- Symbol může být takto definován ve více modulech, při překladu bude nakonec uložen pouze jedenkrát na jedno místo v paměti.

Počátek a konec programu

Pro formát OBJ je definován vstupní bod programu (návěští) jako:

```
..start:
```

Pro formát Win32 je nutné definovat vstupní bod programu prostřednictvím linkeru.

Konec programu je ve Windows zajištěn budˇ instrukcí RET (funkční varianta i v jiných operačních systémech) nebo voláním funkce `ExitProcess`:

```
      RET
```

nebo

```
      CALL ExitProcess
```

Výrazy a operátory (I)

- Podobné syntaxi v jazyce C
- Vyhodnocovány jsou jako 64bitové a poté jsou přizpůsobeny
- Použitelné pouze pro ordinální typy operandů

Binární operátory podle priority (od nejnižší k nejvyšší):

|	bitový součet (OR)	`MOV AL,0x32h | 5`	`(= 0x37)`
^	exkluzivní bitový součet (XOR)	`MOV AL,0x15 ^ 9`	`(= 0x1C)`
&	bitový součin (AND)	`MOV AL,0x9F & 15`	`(= 0x0F)`
<<	bitový posun vlevo	`MOV AL,5 << 3`	`(= 0x28)`
>>	bitový posun vpravo	`MOV AL,0x16 >> 3`	`(= 0x02)`
+	součet	`MOV AL,0x16 + 1`	`(= 0x17)`
–	rozdíl	`MOV AL,0x0A - 10`	`(= 0x00)`
*	součin	`MOV AL,1 * -1`	`(= 0xFF)`
/	bezznaménkové dělení	`MOV AL,1 / -1`	`(= 0x00)`
//	znaménkové dělení	`MOV AL,1 // -1`	`(= 0xFF)`
%	bezznaménkový zbytek po dělení	`MOV AL,1 % -1`	`(= 0x01)`
%%	znaménkový zbytek po dělení	`MOV AL,1 %% -1`	`(= 0x00)`

Výrazy a operátory (II)

Unární operátory

+	nemá vliv na operand (je zde kvůli symetrii s operátorem –)
–	mění znaménko operandu (doplňkový kód)
~	dvojkový doplněk (inverze bitů)
!	logická negace (výsledek je buď 0 nebo 1, dle hodnoty operandu)
SEG	vrací segment operandu například: `MOV AX,SEG alpha`

Pozn.: Symbol $ zastupuje adresu první slabiky symbolické instrukce uvedené na stejném řádku,
dvojitý symbol $$ zastupuje začátek příslušné sekce/segmentu.

Paměťové operandy - efektivní adresa

- hodnoty segmentových registrů za nás naplní operační systém,
- v 32bitovém režimu je ofset roven efektivní adrese,
- hodnota efektivní adresy se vypočítá z konstantního **posunutí**, **základu**, **indexu** a **měřítka** indexu takto:

`EA = ofset = základ + měřítko*index + posunutí`

  - **`posunutí`** = konstantní hodnota – 8bitová, 16bitová nebo 32bitová
  - **`základ`** = některý z registrů `EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI`
  - **`index`** = některý z registrů `EAX, EBX, ECX, EDX,` ~~**`ESP`**~~`, EBP, ESI, EDI`
  - **`měřítko`** = jedna z hodnot 1, 2, 4 nebo 8, kterou bude vynásoben index

- implicitní segmentový registr je dán registrem použitým jako **`základ`**:

  `EAX, EBX, ECX, EDX` = **DS**,
  `ESP, EBP` = **SS**,
  `ESI, EDI` = **DS** (nebo **ES**, závisí na použité instrukci)

Paměťové operandy – povolené kombinace

Základ		Měřítko		Index		Posunutí
EAX	+	1	*	EAX	+	žádné
EBX		2		EBX		8bitové
ECX		4		ECX		16bitové
EDX		8		EDX		32bitové
EDI				EDI		
ESI				ESI		
EBP				EBP		
ESP				---		

Příklady platných adres:

```
[EAX]
[2*EBX]
[586]
[0x00408000]
[ESP]
[EBX + 1*EAX]
[EAX + EAX]
[ESP + 4*ECX]
[EAX + 2*EBX + 10]
```

Paměťové operandy – příklady

`MOV EAX,alpha`	do registru EAX uloží ofset = efektivní adresu symbolu/jména/návěští `alpha`
`MOV EAX,alpha+4`	do registru EAX uloží ofset = efektivní adresu symbolu/jména/návěští `alpha + 4`
`MOV EAX,[alpha]`	do registru EAX uloží hodnotu z paměti uloženou na adrese `alpha`
`MOV EAX,[alpha+8]`	do registru EAX uloží hodnotu z paměti uloženou na adrese `alpha+8`
`MOV [beta],EAX`	do paměti na adresu `beta` uloží obsah registru EAX
`MOV EAX,[pole+EBX]`	do registru EAX uloží hodnotu z paměti uložené na adrese `pole+EBX`
`MOV [alpha],byte 5`	do paměti na adresu `alpha` uloží 0x05 (1 byte)
`MOV [beta+4],word 5`	do paměti na adresu `beta+4` uloží 0x0005 (2byty)
`MOV [EBP],dword 5`	do paměti na adresu `alpha` uloží 0x00000005 (4 byty)
`LEA EDI,[EAX+4*EBX+16]`	do registru EDI uloží efektivní adresu danou výrzem EAX+4*EBX+16 (například pokud v registru EBX bude hodnota 10 a v registru EAX hodnota 100, pak v registru EDI bude hodnota 100 + 4*10 + 16 = 156

Makra (`%define, %idefine`)

- Překlad probíhá v NASM běžně ve dvou průchodech:
  - Předzpracování (*preprocessing*):
    - Velmi dobré prostředky pro zpracování kódu před samotným překladem.
    - Předzpracování nahradí symbolické konstanty hodnotami a rozvine makra.
  - Předklad kódu:
    - Vytvoří výsledný soubor se strojovým kódem v požadovaném formátu.
- NASM obsahuje mnoho vestavěných maker – *direktiv preprocesoru*.
- Vlastní makra definujeme direktivou pro předzpracování:

  `%define` název makra rozlišuje velikost písmen,
  `%idefine` název makra nerozlišuje velikost písmen.
- Vlastní (nová) makra jsou buď jednořádková nebo víceřádková.
- Funkčnost maker lze ověřit použitím volby `–e` při volání NASM:

```
NASM –fwin32 -e example.asm > example.pre
```

  - `>` je přesměrování výstupu z obrazovky do souboru `example.pre`
  - Výsledek předzpracdování bude v souboru `example.pre`

Jednořádková makra

```
%define  jméno[(parametry)] tělo_makra
%idefine jméno[(parametry)] tělo_makra
```

- Jméno je povinné a může se shodovat čímkoliv co se již v NASM vyskytuje = dojde k předefinování významu.
- Varianta `%idefine` nerozlišuje velikost písmen.
- Parametry makra nejsou povinné.

Příklad:

```
%define CR_LF 13,10
```

Použití:	**Překlad:**
`message DB 'hello',CR_LF,0`	`message DB 'hello',13,10,0`

Příklad:

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

Použití:	**Překlad:**
`MOV EAX,pamet(EBX,ESI,10)`	`MOV EAX,[EBX+ESI+10]`

Víceřádková (pravá) makra

```
%macro jméno_makra N
    tělo makra
%endmacro
```

```
%imacro jméno_makra N
    tělo makra
%endmacro
```

- Pro název platí totéž, co u jednořádkových maker.
- *N* = počet parametrů makra (kladné číslo, hodnota 0 = žádné parametry).

Použití:

```
jméno_makra [parametr1, parametr2, … , parametrN]
```

Příklad:

```
%macro fn_begin 0
    PUSH EBP
    MOV EBP,ESP
%endmacro
```

Použití:	**Překlad:**
fn_begin	PUSH EBP MOV EBP,ESP

Parametry makra (I)

Počet parametrů je pevně dán hodnotou *N*. Odkaz na parametry makra:

`%0` … počet zadaných parametrů = *N*
`%1` … 1. parametr
…
%N … *N*. parametr

Greedy (hltavé) parametry

`%macro` *jméno_makra* *N+*

- Počet parametrů není omezen, *N* udává počet povinných parametrů, cokoliv navíc je pro použití makra „dobrovolné“.

Příklad:

```
%macro string 2+
    %1: DB %2
%endmacro
```

Použití:	**Překlad:**
`string muj_retezec,65,'A',"ahoj"` `└──%1──────┘ └────%2──────┘`	`muj_retezec: DB 65,'A',"ahoj"`

Parametry makra (II)

Proměnný počet parametrů

```
%macro jméno_makra M-N
```

- Počet parametrů je stanoven na minimální počet *M* povinných parametrů a makro umí přijmout maximálně *N* parametrů (*M* < *N*).
- Je **nutné** určit implicitní (*default*) hodnotu parametrů na pozici >*M*.

Příklad:

```
%macro fn_begin 1-2 0
    %1:
        PUSH EBP
        MOV EBP,ESP
        SUB ESP,%2*4
%endmacro
```

Použití:	**Překlad:**

`fn_begin ma_funkce,5`	`ma_funkce:`
	`  PUSH EBP`
	`  MOV EBP,ESP`
	`  SUB ESP,20`

Parametry makra (III)

Proměnný, předem nespecifikovaný počet parametrů

```
%macro jméno_makra N-*
```

- Minimální počet parametrů je *N*, makro umí přijmout i větší počet parametrů (>*N*) s tím, že jejich maximální počet ani implicitní hodnota není definována.
- Využíváme makro `%0` ke zjištění počtu skutečných parametrů makra.
- Lze se odkázat i na interval parametrů, ne pouze na jeden z nich, výrazem:

```
%{X:Y}
```

- Přičemž *X* a *Y* jsou hraniční indexy a pokud $X \leq Y$, pak je zachováno pořadí a pokud je $X > Y$, pak je pořadí obráceno (toho lze docílit i záporným indexem).
- Trik, jak se „dopracovat“ k poslednímu parametru při proměnném počtu parametrů je použití: `%{-1:-1}`.

Spojování parametrů makra s okolním textem

```
xxx%1          nebo   xxx%{1}
%1yyyy         nebo   %{1}yyyy
xxx%1yyyy      nebo   xxx%{1}yyyy
```

Parametry makra (IV)

Příklady:

Příklad:	Použití a překlad:

`%macro mpar 1-*`
`  DB %{3:5}`
`%endmacro`
	`mpar 1,2,3,4,5,6`
	
	`DB 3,4,5`

`%macro mpar 1-*`
`  DB %{5:3}  ≡ %{-2:-4}`
`%endmacro`
	`mpar 1,2,3,4,5,6`
	
	`DB 5,4,3`

`%macro mpar 1-*`
`  DB %{-1:-3}  ≡ %{6:4}`
`%endmacro`
	`mpar 1,2,3,4,5,6`
	
	`DB 6,5,4`

`%macro mpar 1-*`
`  DB %{-1:-1}  ≡ %{6:6}`
`%endmacro`
	`mpar 1,2,3,4,5,6`
	
	`DB 6`

Parametry makra (V)

Podmínkové kódy jako parametr

- Někdy se hodí předání podmínkového kódu (**C**, **Z**, **E**, **NE**, apod.) jako parametru.
- Lze využít speciální odkaz na parametr, který má obsahovat podmínkový kód – NASM zkontroluje, zda daný parametr je platný podmínkový kód a případně umí tento kód i negovat:

%+X = přímý kód nebo *%-X = negovaný kód*

- Přičemž *X* je číslo parametru.

Lokální návěští v makrech

- V rámci makra je občas potřeba definovat skoky => nutnost definovat návěští.
- Nelze definovat globálním návěštím (nejednoznačnost) => používá se lokální návěští ve spojení s globálním, které generuje NASM sám = nemusíme se starat o jeho definici.
- V makrech se lokální návěští definuje jako:

```
%%návěští:
```

Parametry makra (VI)

Příklady:

```
%macro ret_if_cond 1
    J%-1 %%skip
    RET
%%skip:
%endmacro
```

Použití:	Překlad:

`ret_if_cond Z`	`JNZ @0001.skip`
	`RET`
	`@0001.skip:`

`ret_if_cond NE`	`JE @0054.skip`
	`RET`
	`@0054.skip:`

Direktiva preprocesoru: %rotate

%rotate *X*

- Umožní posun pořadí parametrů makra o daný počet ve směru určeném znaménkem čísla *X*.
- Pro *X* > *0* = rotace vlevo, pro *X* < *0* = rotace vpravo.

Příklad:

```
%macro makro param1,param2, …, paramN-1, paramN

    %rotate 1 =>  param2,param3, …, paramN, param1
                  └─%1─┘ └─%2─┘  … └─%N-1─┘ └─%N─┘

    %rotate -1 => paramN,param1, …, paramN-2, paramN-1
                  └─%1─┘ └─%2─┘  …  └─%N-1─┘  └─%N───┘

    %rotate 3  => param4,param5, …,  param2, param3
                  └─%1─┘ └─%2─┘  …  └─%N-1─┘ └─%N─┘
 %macro makro A,B,C
     %rotate 0  … nezpůsobí nic
     %rotate 3  … nezpůsobí nic
     %rotate -3 … nezpůsobí nic
```

Direktiva preprocesoru: %rep … %endrep

- Makro pro opakování úseku kódu.

```
%rep N
        kód, který má být opakován N-krát
    [%exitrep] = předčasné ukončení
%endrep
```

Příklady:

Použití:	Překlad:

`%rep 4`
`  ADD EAX,EAX`
`%endrep`
	`ADD EAX,EAX`
	`ADD EAX,EAX`
	`ADD EAX,EAX`
	`ADD EAX,EAX`

`%rep 5`
`  DD 0`
`%endrep`
	`DD 0`
	`DD 0`
	`DD 0`
	`DD 0`
	`DD 0`

Příklady

Multi-push a multi-pop

```
%macro mpush 1-*
    %rep %0
        PUSH %1
        %rotate 1
    %ednrep
%endmacro
```

```
%macro mpop 1-*
    %rep %0
        %rotate -1
        POP %1
    %ednrep
%endmacro
```

Použití:

```
mpush EAX, EBX, ECX


…

mpop EAX, EBX, ECX
```

Překlad:

```
PUSH EAX
PUSH EBX
PUSH ECX

…

POP ECX
POP EBX
POP EAX
```

Příklady:

```
%macro call_cdecl 1-*
    %if %0 > 1
        %rep %0-1
            %rotate -1
            PUSH dword %1
        %endrep
        %rotate -1
    %endif
    CALL %1
    %if %0 > 1
        ADD ESP,(%0-1)*4
    %endif
%endmacro
```

```
%macro call_pascal 1-*
    %if %0 > 1
        %rep %0-1
            %rotate 1
            PUSH dword %1
        %endrep
        %rotate 1
    %endif
    CALL %1

%endmacro
```

Použití:

```
call_cdecl ma_funkce1, 5, x, [y]
```

Překlad:

```
PUSH dword [y]
PUSH dword x
PUSH dword 5
CALL ma_funkce1
ADD ESP,12
```

```
call_pascal ma_funkce2, 5, [y]
```

```
PUSH dword 5
PUSH dword [y]
CALL ma_funkce2
```

Příklady:

```
%macro fn_begin 1-2 0
    %1:
        PUSH EBP
        MOV EBP,ESP
    %if %2 > 0
        SUB ESP,%2*4
    %endif
%endmacro
```

```
%macro fn_end
    MOV ESP,EBP
    POP EBP
    RET
%endmacro
```

Použití:

```
fn_begin ma_funkce1,5
```

Překlad:

```
ma_funkce1:
    PUSH EBP
    MOV EBP,ESP
    SUB ESP,20
```

```
fn_begin ma_funkce2
```

```
ma_funkce2:
    PUSH EBP
    MOV EBP,ESP
```

```
fn_end
```

```
    MOV ESP,EBP
    POP EBP
    RET
```

Direktivy preprocesoru: %include a %assign

Vložení souboru – %include

%include “*jméno souboru*“

Přiřazení hodnoty symbolu – %assign

%assign *jméno_symbolu výraz(číselný)*

Příklady:	
Použití:	**Překlad:**
%include “rw32.inc”	Vloží obsah souboru rw32.inc do souboru na místo, kde se nachází makro **%include**
%assign n 5 **%assign** i i + 1	Na výstupu se přímo neprojeví. Preprocessor interně definuje proměnné, které lze použít (viz složitější příklady) n = 5 i = i + 1

Podmíněný překlad (I)

%if / **%ifn** *podmínka1* … **%elif** / **%elifn** *podmínka2* … **%else** … **%endif**

- Podmíněný překlad – úsek kódu bude přeložen jen pokud bude splněna podmínka. `%if` = test numerické podmínky.

%if *podminka1*
zdrojový kód, který se přeloží při splnění *podminka1*
[**%elif** *podminka2*
zdrojový kód, který se přeloží při nesplnění *podminka1* a při současném splnění *podminka2*]
....
[**%elif** *podminkaN*
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

Podmíněný překlad (II)

Operátory numerických podmínek podmíněného překladu

- Porovnání: = (nebo ==), <, >, <=, >=, <> (nebo !=)
- Logické operátory: && (and), ^^ (xor), || (or)
- Logické operátory považují nenulové hodnoty operandů za logické hodnoty TRUE.

- Test existence *jednořádkového* makra:

%ifdef / **%ifndef** *jméno_makra …* **%elifdef** / **%elifndef** **… %endif**

- Test existence *pravého (víceřádkového)* makra s parametry:

%ifmacro / **%ifnmacro** *jméno_makra …* **%elifmacro** / **%elifnmacro** **… %endif**

- Ve všech variantách lze použít **%else**.

Podmíněný překlad (III)

- Test shody *textových řetězců* `txt1` a `txt2`:

%ifidn **%ifidni** **%ifnidn** **%ifnidni**	*txt1, txt2* …	**%elifidn** **%elifidni** **%elifnidn** **%elifnidni**	… **%endif**

- Test *typu* parametru (identifikátor, číslo, řetězec):

%ifid **%ifnid** **%ifnum** **%ifnnum** **%ifstr** **%ifnstr**	*parametr* …	**%elifid** **%elifnid** **%elifnum** **%elifnnum** **%elifstr** **%elifnstr**	… **%endif**

Příklady – call_cdecl – ještě lépe

```
%macro call_cdecl 1-*
    %if %0 > 1
        %assign i 0
        %rep %0-1
            %rotate -1
            %assign i i+1
            %ifstr %1
                [section .data]
                    %%newString %+ i: DB %1, 0
                [section .text]
                    PUSH %%newString %+ i
            %else
                PUSH dword %1
            %endif
        %endrep
        %rotate -1
    %endif
    CALL %1
    %if %0 > 1
        ADD ESP,(%0-1)*4
    %endif
%endmacro
```

%+ … spojení řetězců „%%newString“ a hodnoty proměnné „i“ převedené na řetězec

například pro i = 5:

%%newString %+ i … %%newString5

Příklady – call_cdecl – ještě lépe (překlad)

Použití:

```
call_cdecl printf,"retezec %s, %s, %d","ret1","ret2",5
```

Překlad:

```
PUSH dword 5
    [section .data]
    ..@57.newString2: DB "ret2",0
[section .text]
PUSH ..@57.newString2
    [section .data]
    ..@57.newString3: DB "ret1",0
[section .text]
PUSH ..@57.newString3
    [section .data]
    ..@57.newString4: DB "retezec %s, %s, %d",0
[section .text]
PUSH ..@57.newString4
CALL printf
ADD ESP,(5-1)*4
```

Příklad využití knihovny jazyka C v asembleru (1)

- vložíme „rw32“ a informujeme NASM o externích funkcích
- ~~definujeme řetězce, které se v programu vyskytnou~~
- definujeme proměnné

```
%include 'rw61-2015.inc'
extern malloc
extern fopen
extern fgets
extern strlen
extern printf
extern strcmp
extern fclose
extern free
```

```
segment .data
    sFileName       DB "stdlibtest.asm",0
    sFileOpenMode   DB "r",0
    sEmptyLine      DB "; prázdný řádek",EOL,0
    sNOP            DB "NOP",EOL,0
    sNIC            DB "; NIC",EOL,0
    sFormatString   DB "%s",0
    sError          DB "Chyba při otvírání"
                    DB " souboru!",EOL,0

    pFile           DD 0
    pBuffer         DD 0
```

Příklad využití knihovny jazyka C v asembleru (2)

- naprogramujeme kód

```
segment .text
; int main(void) {
main:
;    char* pBuffer = (char *)malloc(100);
    call_cdecl malloc, 100
    MOV [pBuffer],EAX
;    FILE* pFile = fopen("stdlibtest.asm", "r");
    call_cdecl fopen,"stdlibtest.asm","r"
    MOV [pFile],EAX
;    if (pFile) {
    CMP EAX,0
    JE else
```

```
PUSH dword 100
CALL malloc
ADD ESP,4
```

```
PUSH sFileOpenMode
PUSH sFileName
CALL fopen
ADD ESP,8
```

Příklad využití knihovny jazyka C v asembleru (3)

```
;       while (fgets(pBuffer, 100, pFile) != NULL) {
while:
    call_cdecl fgets,[pBuffer],100,[pFile]
    CMP EAX,0
    JZ endwhile
;           if (strlen(pBuffer) <= 2) printf("; prázdný řádek\n");
    call_cdecl strlen,[pBuffer]
    CMP EAX,2
    JNLE elseif_skipcomment

    call_cdecl printf,{"; prazdny radek",EOL}
    JMP while
```

```
PUSH dword [pFile]
PUSH dword 100
PUSH dword [pBuffer]
CALL fgets
ADD ESP,12
```

```
PUSH dword [pBuffer]
CALL strlen
ADD ESP,4
```

{ … } … ohraničení parametru, který obsahuje čárky

```
PUSH sEmptyLine
CALL printf
ADD ESP,4
```

Příklad využití knihovny jazyka C v asembleru (4)

```
;           else if (pBuffer[0] != ';') printf("%s", pBuffer);
elseif_skipcomment:
    MOV EAX,[pBuffer]
    CMP byte [EAX],';'
    JE while

    call_cdecl printf,"%s",[pBuffer]
    JMP while
;       }
endwhile:
;       fclose(pFile);
    call_cdecl fclose,[pFile]
    JMP endif
```

```
PUSH dword [pBuffer]
PUSH sFormatString
CALL printf
ADD ESP,8
```

```
PUSH dword [pFile]
CALL fclose
ADD ESP,4
```

Příklad využití knihovny jazyka C v asembleru (5)

```
;    } else {
else:
;        printf("Chyba při otvírání souboru!\n");
    call_cdecl printf,{"Chyba pri otvirani souboru!",EOL}
;    }
endif:
;    free(pBuffer);
    call_cdecl free,[pBuffer]
;}
    RET
```

```
PUSH sError
CALL printf
ADD ESP,4
```

```
PUSH dword [pBuffer]
CALL free
ADD ESP,4
```