V/V operácia priama adr. 8b
nepriama adr. 16b [DX]

(7) PREDNÁŠKA 22.3.2006

- test okolo 9. týždňa

(1) ARITMETICKÉ INŠTRUKCIE:

- základné sú +, −, /, *

- 8 alebo 16 bit čísla, so znamienkom (v doplnkovom kóde) alebo bez znamienka
- pri BCD funguje korekcia (okrem delenia) po vykonaní aritm. úkonu operácie, ale podmienkou je, že operandy musia byť korektne kódované v BCD tvare
- ak nastavujú vsetky príznakové bity

SČÍTANIE:

destination source

ADD dst, src dst = dst + src

ADC dst, src dst = dst + src + CF
↳ add with carry, pripočíta aj carry flag

R1, M ; R, M veľkosť 2÷4B

R1, M ; data → priama hodnota

AC ; data → pripočítavam k akumulátoru

INC R, M
INC R
AAA - desiatková korekcia pre voľný formát

rády → 7654 3210

```
      0000 1001        09
    + 0000 0001        01
DA    0000 1010   ?    10
    + 0000 0110        +06
      0001 0000  → 10
```

↳ kontroluje či nastal prenos medzi 3. a 4. rádom a či je menší polbajt menší ako 9
↳ ak áno pripočítaj 6

DAA → používa sa pri korekcii zhusteného formátu

ODČÍTANIE:

SUB dst, src dst = dst − src

SBB dst, src dst = dst − src − CF → s prenosom

- typy operandov ako pri sčítaní

DEC -> zníženie obsahu o 1

NEG -> zmena znamienka
R, M (register alebo pamäťová bunka)
-> vykoná sa doplnkovo, od samých 1111 sa odčíta operand a pripočíta sa 1

CMP -> vykoná sa odčítanie, ale do cieľa sa nezapíše výsledok, ale iba sa nastaví príznakový register

AAS - korekcia
DAS - —"— pre zhustený formát

NÁSOBENIE:

MUL R, M
8 bit AX ← AL * src
16 bit DX:AX ← AX * src

IMUL -> integer multiplication
AAM -> korekcia

DELENIE:

DIV R, M
IDIV - celé čísla

8 bit AL ← AX : src
AH ← zvyšok po delení
16 bit AX ← (DX:AX) ÷ SRC
DX ← zvyšok
AAD -> dekadická korekcia

CBW -> convert byte to word
↳ ak AL < 80H => AH ← 0000 0000 (00H)
ak AL ≥ 80H => AH ← 0FFH
to je tam len z formy kvôli zladiteľnosti

CWD -> convert word to double word
ak AX < 80H => DX ← 00H
ak AX ≥ 80H => DX ← 0FFH

2. BITOVÉ OPERÁCIE (INŠTRUKCIE)

- príznakový CF je pri týchto operáciách vynulovaný

NOT R, M -> mení 1 na 0 a naopak (unárna operácia)

LOGICKÉ:

AND, OR, XOR, TEST — dst, src
- dst ← dst AND src
- dst ← dst OR src
- dst ← dst ⊕ src
- „F" ← dst AN src - len sa nastaví flag, výsledok neukl

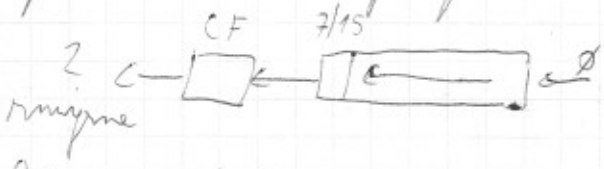
ciele:	R_1, M	R, M	AC
zdroje:	R_2, M	dáta	dáta

POSUNY A ROTÁCIE:

CF sa nenuluje, preto nie je vhodné 8 a 16 bit. operandy

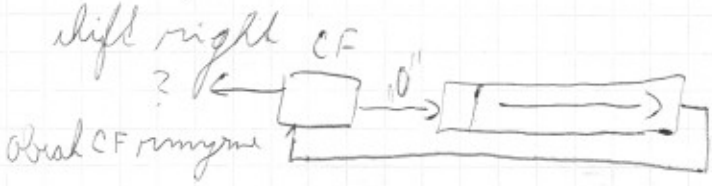
SHL shift left dst, count -> o koľko posunúť
R/M -> ak neuvedie, posunie sa o 1 bit

SAL shift aritmetically left

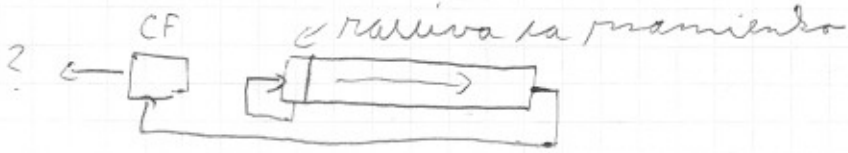


sú identické operácie

SHR shift right

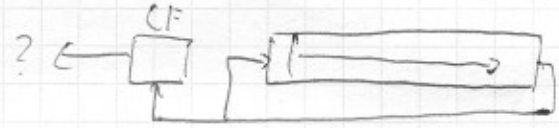


SAR

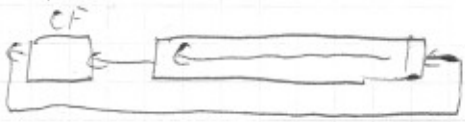


ROL rotate left

ROR



RCL rotate s CF



RCR rotate carry right

REŤAZCOVÉ INŠTRUKCIE:

SIC: (zdrojový operand) DS : SI

DIC: (cieľový operand) ES : DI

byte MOVSB (presuň string) dst, src [ES : DI] ← [DS : SI]

word MOVSW

CMPSB } „F" ← dst - src ← nastavia sa príznaky
CMPSW }

SCASB } skenuj
SCASW } relácie „F" ← AC - dst
↳ hľadám jeden údaj

LODSB load AC ← SIC
LODSW

STOSB store dst ← AC
STOSB

Prefix → dáva sa pred inštrukciu a dáva pokyn na opakovanie
REP → nepodmienené, aplikovateľný na MOV, LOD, STO
→ počet opakovaní je v C registri, treba nastaviť

REPE → repeat if equal
(REPZ) pokiaľ sa ZF nenastaví na Ø ale maximálne CX-krát

REPNE → if not equal
(REPNZ) → čaká ka na ZF: 1 maximálne ale iba CX-kra

premiestnenie pole bajtov

```
MOVEB:   MOL   AL, [SI]
         MOL   [DI], AL
         INC   SI
         INC   DI
         DEC   CX
         JNZ   MOVEB
```

[DS : SI]
[ES : DI] [DF] destination flag
[CX]
REP MOVSB

```
COMPR:   MOV   AL, [SI]
         CMP   [DI], AL
         JE    EQUAL
         INC   SI
         INC   DI
         DEC   CX
         JNZ   COMPR
EQUAL:
```

REPNZ CMPSB
JZ EQUAL

RIADIACE INŠTRUKCIE

- 1B, vykonávajú sa max. počas 2 taktov
- ovládanie bitov príznakového registra

CLC - clear carry
CMC - complement carry
STC - set carry 1

CLD clear direction
STD set direction
CLI
STI

- synchronizácia

HLT . halt -> zastaví vykonávanie inštrukcií a čaká na prerušenie

WAIT -> vkladá do strojového cyklu čakacie takty
-> testuje TEST a ak je Ø pokračuje v cykle ďalej

LOCK -> aby po sbernici vysielal len jeden procesor

NOP -> nič sa nevykoná

[CS:IP] -> blízke NEAR -> keď voláme podprogram alebo návestie, ktoré je v tom istom segmente ako sa práve nachádzam
zmena miesta výberu inštrukcie
-> mení sa iba IP
FAR -> ak je to v inom segmente
↳ musíme zmeniť CS a aj IP register
- ak voláme podprogramy ovplyvňuje sa tým stack pointer
- ovplyvňuje sa aj F

SKOKY:

JMP cieľová adresa

1. blízky priamy -> v rámci akt segmentu
-> priamy -> práve určená adresa

JMP displacement
IP ← IP + disp

2. blízky priamy krátky -> displacement je ~~dlhý~~ veľký iba 1B

JMP $disp_L$
-> všetky podmienené skoky už takéto

3. blízky nepriamy

JMP R/M IP ← R/M vloží sa to do IP registra

4. ďaleký priamy

op. code
JMP adr.

OC	OFFSET	SEG/SEG
	IP	CS

5B spolu má ta inštrukcia

5. ďaleký nepriamy

JMP M IP ← [M]
CS ← [M+2]

- to isté aj pri volaní podprogramu pomocou CALL okrem 3. prípadu + sa ešte do zásobníka odpamätáva adresa návratu

CALL
[SP] ← CS
SP ← SP - 2
[SP] ← IP
IP ← offset
CS ← seg

návrat z podprogramu RET IP ← [SP]
SP ← SP + 2
CS ← [SP]
SP ← SP + 2

- preto nie je možné na podprogram skákať cez JMP, lb
sa neofaniatá adresa odkiaľ sme skočili a potom po
volaní RET skočíme na iné miesto ako sme chceli

→ nova ku skokom → blízke priame krátke

vyšší → ABOVE } bez znamienka
nižší → BELOW
väčší → GREATER } so znamienkom
menší → LESS

JA → skoč ak vyšší
(JNBE)

JAE (JNB)
JB (JNAE)
JBE (JNA)
JE (JZ)
JNE (JNZ)

JC jumy if carry
JNC
JE (JZ)
JG (JNLE)
JGE (JNL)
JL (JNGE)
JLE (JNG)
JP jumy if parity
JNP
JS jumy if sign
JNS
↳ podobne na vlastny primat
tiky

LOOP adr - nepodmienená slučka
CX - 1, pokiaľ je CX ≠ 0 vykonáva sa skok

LOOPNE (LOOPNZ) - sleduje sa CX
- iba ak testuje primalový ZF

LOOPE (LOOPZ) → pokiaľ Z = 1 opakuje sa max. CX krát

② PREDNÁŠKA 5.4.2006

- priebežný test 24.4.2006

inštrukcia → príkaz pre procesor na vykonanie operácie

pseudoinštrukcia → príkaz pre prekladač

premenná → segment, offset, typ
↳ bajt B nabera

- zložitejší údajový typ

<u>ŠTRUKTÚRA</u>: premenná, ktorej položkami sú premenné s minimálnou dĺžkou 1B. Každá položka štruktúry je samostatne prístupná.

- meno štruktúry STRUC
  definovanie položiek → DB
  DD
  DW

  meno štruktúry ENDS

```
DATUM     STRUCT
  DEN     DB   ?      (definujeme deň)
  MES     DB   'JUL'  dátumy má 6 B
  ROK     DW   ?
DATUM     ~~STRUCT~~ ENDS

OSOBA     STRUCT
PRIEZV    DB   15 DUP (Ø)
MENO      DB   10 DUP (Ø)
DNAR      DATUM <>
OSOBA     ENDS
```

MENO_STRUK → definuje vnútornú štruktúru premennej

[meno_prem] MENO_STRUK [DUP(]<[poc.dní]>[

DNES DATUM <5, 'APR', 2006>
POLE_DAT DATUM DUP (<>)

vymedzí sa 100 x 6 B s preddefinovaným obsahom štruktúry

OS1 OSOBA <'MRKVA', 'PETER', , , >
KARTO OSOBA 100 DUP (<>)

100 x 31 B - lebo je tam 100 osôb

Sprístupnenie

meno_prem.meno_položky

MOV AX, DNES.ROK

MOV AX, OS1.DNAR.ROK

- chcem dát narodenia B osoby z kartotéky

MOV AX, KART[ ].DNAR.ROK

ku sa musím nastaviť na začiatok 13. osoby
(13-1) * dĺžka, položka osoba
(13-1) * LENGTH OSOBA

- či použijem SIZE alebo LENGTH závisí od toho, či pri definovaní použijem DUP

TABUĽKA MOCNÍN

2	2	4	8	16	32	
4	4					
8	8					
10	10	100	1000	10000	100000	← najväčšia hodnota v tabuľke

5

4

```
RIADOK   STRUCT                    - definuje sa riadok tabuľky
R        DW  10 DUP(?)

RIADOK   ENDS
                  BX -> tu bude základ
ZAKL     DW  2,4,8,10              - definujú sa základy
TAB      RIADOK  4 DUP(<>)
```

1. riadok = 20B

P1 P2 P3 P4 P5 . . .

SI ukazuje na poč. riadok 4B = 2W

DI ukazuje na aktuálne miesto v pamäti

```
NAPLNTAB   PROC
           XOR   SI,SI            - vynulujem registre
           XOR   BX,BX
           XOR   DX,DX
NR         MOV   CX, 5            -> v CX je počítadlo cyklov
           MOV   AX, 1
           XOR   DI, DI           -> ukazovateľ na začiatok
           ADD   DI, SI           -> pripočítam k DI register SI
                                     aby sme boli na začiatku riadku
RIAD       MUL   ZAKL[BX]         -> vynásobí sa s AX, je to 16 bit
                                     násobenie a výsledok je v dvojici
                                     premenných AX a DX
           MOV   TAB.R[DI], DX
           MOV   TAB.R[DI+2], AX  -> presunieme výsledok
                                     násobenia do 1. položky
           ADD   DI, 4            -> presunieme sa o 4 bajty na ďalšie
           LOOP  RIAD               prebehne po 5 x, lebo
                                    automaticky dekrementuje CX
           ADD   SI, 20           -> prejdeme na ďalší riadok
           ADD   DX, 2            -> presunieme sa na ďalší základ
           CMP   BX, 8            -> kontrolujem, či som už nie na
           JB    NR                 konci základov
           RET
NAPLNTAB   ENDP
```

ZÁZNAM -> premenná, kt. položky sú bitovo orientované
zápis → počet bitov

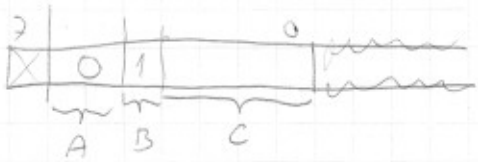
```
MENO_ZAZN   RECORD   položka : dĺžka [počiatočná hodnota] [ ... ]
```

- súčet dĺžok nemôže presiahnuť 2B = 16b

[meno_prem] meno_záznamu [DUP(] <počiatočné hodnoty> [)]

príklady

BITREC RECORD A:2=0, B:1=1, C:4

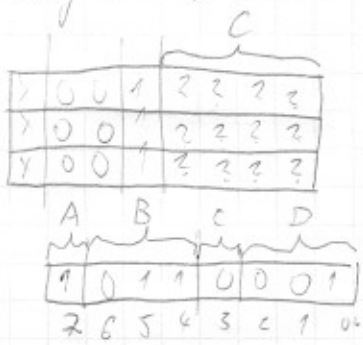


BITT BITREC <>

POLEBIT BITREC 3 DUP (<>) → vytvorí sa 3 bajtové pole bitov

FARBA RECORD A:1, B:3, C:1, D:3

FRB FARBA <1, 3, 0, 1>



užitočné operátory

1. MOV AL, C (AL←3) - do AL sa umiestni hodnota umiestnená položky C
2. MASK AL, MASK C (AL←00001000) → do AL sa presunie maska položky C
3. WIDTH - počet bitov, ktoré položka/premenná obsahuje

MOV AL, WIDTH B (AL←3)

- chcem B zmeniť z 3 na 5

MOV AL, 5 AL←00000101

MOV CL, B v CL bude hodnota 4 (00000100)

- meno položky v rekorde musí byť jednoznačné

SHL AL, CL → posuň register AL doľava o počet miest, ktorý je v registri CL

AL ← 0101 0000 → toto sa posunulo

MOV AH, FRB → do hornej polovice prenesie nieco

AH ← 1010001

AND AH, NOT MASK B

```
  1010001
  1000111
  -------
  1000001
```

OR AH, AL AH ← 1101 0001 → toto je tá 5-ka

MOV FRB, AH

- zaujímavosti:

```
MOV  AX, P1
ADD  AX, P2
MOV  P3, AX
```

- často sa nám v programe vyskytuje takáto sekvencia inštrukcií a chceme ju robiť nejako efektívnejšie

- keď by to bola procedúra, jej volanie a tiež návrat môže trvať viac ako samotný výpočet
- až ju, naberá na takéto meno v zápisoch, lebo vždy vo prekladoc sa to musí dokodovať

- používam MAKRO INŠTRUKCIU

```
meno MACRO parametre
        {
        }
        ENDM
```

```
SUM MACRO  A,B,V
    MOV  AX, A
    ADD  AX, B
    MOV  V, AX
    ENDM
```

} zadefinované makro

```
SUM  ALFA, BETA, GAMA  <  MOV AX, ALFA
                          ADD AX, BETA
                          MOV GAMA, AX

SUM  P1, P2, P3        <  MOV AX, P1
                          ADD AX, P2
                          MOV P3, AX
```

- keď na ňu prekladač narazí tak ju nahradí tými inštrukciami, ktoré sú uvedené v makre

- podmienená pseudoinštrukcia

```
IF výraz
   príkazový riadok
[  ELSE
   príkazový riadok ]
ENDIF
```

- predchádzajúce makro upravíme tak, že ak nie je definovaná nejaká premenná tak sa nekopíruje

```
SUM  MACRO  A,B,V
     IFNDEF  A        - ak nie je def. A
     EXITM            - koniec makra
     ENDIF            - koniec podmienky
     IFNDEF  B
     EXITM
     ENDIF
     IFDEF   V        - ak je V definovaná
     MOV AX, A
     ADD AX, B
     MOV C, AX
```

ENDIF - koniec IF
ENDM - koniec makra

- DOSovská funkcia

```
DOSFN MACRO FUNKCIA
      MOV AH, FUNKCIA
      INT 21H
      ENDM
```

- nebolaby potom písať vždy INT 21H, ale namiesto toho sa použije makro DOSFN

```
ZOBRAZ MACRO ZNAK
       MOV DL, ZNAK
       DOSFN 02
       ENDM
```

← aj vo vnútri makra môžem použiť už definované makro

ZOBRAZ 'A' → MOV DL, A
→ MOV AH, 02
→ INT 21H

toto sa to rozvinie

OPERATORY

PTR - pointer
typ PTR výraz

```
BT   DB 10H, 20H
WRD  DW 30H
```

v pamäti 10H 20H
v pamäti FEM 30H ← nižší bajt

```
MOV CL, BYTE PTR WRD

MOV AX, WORD PTR BT
```

→ daj do AX slovo, na ktoré ukazuje premenná BT
- ja si určím toto sa bude presúvať

JMP SHORT N → nevr. pri dopredných skokoch priamo povieme toto sa má vytvoriť miesta pre skok

TYPE - vracia veľkosť premennej v B
DB - 1
DW - 2
⋮
T - 10

POLE B DB 10 DUP (?)
→ LENGTH vráti 10 (vždy počet prvkov poľa
- ak tam nie je DUP, vrátilo by 1

SIZE = TYPE * LENGTH → počet bytov rezervovaných v pamäti

- priorita operátorov

1, LENGTH, SIZE, WIDTH, MASK, (), [], <>
2, štruktúrované premenné
3, explicitné definovanie segmentu
4, PTR, OFFSET, SEGMENT, TYPE
5, unárne +, −
6, *, /, MOD, SHL, SHR
7, binárne +, −
8, relačné EQ, NE, LT, LE, GT, GE
9, NOT
10, AND
11, OR, XOR

8. CVIČENIE

- ako vložiť do registra hodnotu -> niečo ako assembler
- keď dávam konštantu do EAX

```
ASM ("movl $99, %EAX");
```

zdroj cieľ

- iné ako sa intelovská syntax
- keď dávam premennú

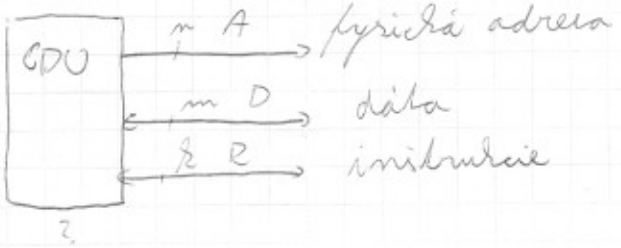
int i=5; -> premenná

- ktorú je vložím

```
ASM ("" :: "a" (i));
```

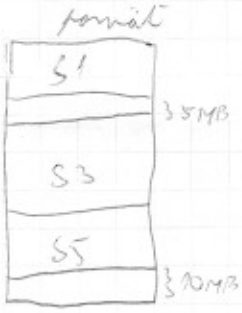
register EAX medzera

(8) PREDNÁŠKA 12.4.2006



Logická adresa BAS : RA
segmentová → SEG : EA → efektívna adresa
FA

pri 8086 sa FA = 16 x SEG + EA



- pri segmentovaní môže nastať tento prípad
- nastáva externá fragmentácia
- pri ukladaní na segment iste rozloží na drobné, a tie sa rozloží do voľný pamäte

- o segmente musíme uvádzať ešte dodatočné informácie, o veľkosti umiestnení, prístupových právach a podobne

údaje → prístup, prítomnosť v pamäti, BAS (bázová adresa), veľkosť } deskriptor

Logická adresa → SEL : RA
SEL ← selektor; RA ← relatívna adresa (už v rámci segmentu, nemení sa)

SEL → vyberie z tabuľky jednu časť, získa bázovú adresu a spolu s RA z toho dostane fyzickú adresu

→ ukazuje do tabuľky deskriptorov, kde nájde BAS a s RA vypočíta FA

STRUKTÚRA DESKRIPTORA - INTEL — má 8 B
8086

8086 ich nevyužíva

7	Rezervované		6
5	Práva	BAS 16-23	4
3	BAS 8-15	BAS 0-7	2
1	limit 8-15	limit 0-7	0

BIU - bus interface unit

EU - execution unit

SP286

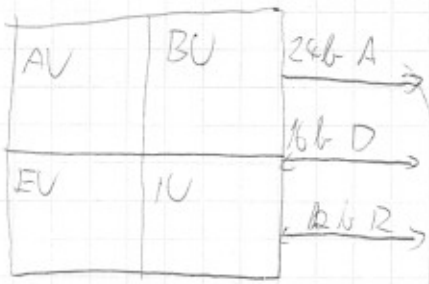
BU - bus unit

IU - instruction unit (predbežiaca inštrukcia)

EU - execution unit

AU - vypočítava fyzickú adresu

riadiaca



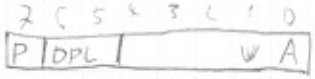
→ môže sa adresovať 2^{24} B → 16 MB

LA = SEL : OFFSET (synonymá RA - rel. adresa, EA - efektívna adresa)

SEL → R bitov
OFFSET → S bitov
} virtuálny pamäťový priestor je 2^{R+S}

80286 konkrétne S = 16 b, R = 14 b } 2^{30} = 1 GB → virtuálny priestor procesora ~~SP286~~ 286

PRÍSTUPOVÝ BAJT:



P → či je segment vnútorný, alebo ne v pamäti (prezent)

DPL → descriptor privileg level (má 4 úrovne privilegovanosti) 0 je najprivilegovanejší a 3 najmenej → privilegovať môže len ten, kto má úroveň vyššiu alebo rovnú

A → access → prístup

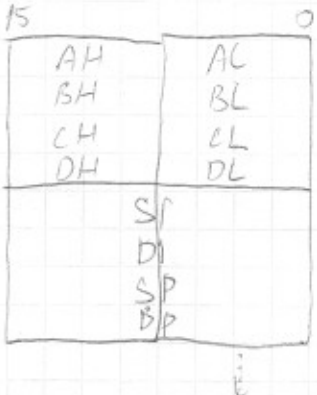
W → writable

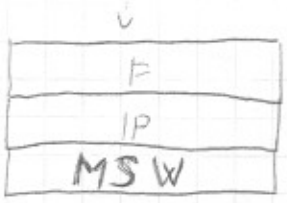
SEL: 16 bitový

RPL - request privileg level → ~~[illegible]~~

T - table → globálne 0
→ lokálne 0

8086





F - príznakový (16 b)

F

15					8	7						0
NT	IOPL	O	D	I	T	S	Z	X	A	Y	P	X	C

- I/O privileg. level → úroveň privilegovanosti I/O operácií
- NT – nested task → pri prepínaní

MSW → machine status word – 16 B

15							8	7							0
											TS	EM	MP	PE

na ovládanie mat. koprocesora
- nepodstatný

TS – task

PE – protection enable
↳ 2 režimy → 1. REAL → chová sa ako 8086 (aj keď je to 80386)

DS
ES
SS
CS

$\boxed{PE = 0}$

→ 2. PVAM – protected virtual advanced mode
- virt. adr. priestor s ochranou

$\boxed{PE = 1}$

- nastaví premenné
- pre segmentových registrov sa stanú selektory, ktoré budú ukazovať do tabuľky deskriptorov

47	39	15	0
priestup	BAS	LIMIT	

→ pomocná pamäť, ktorú používateľ nepristupuje

GDTR – global descriptor table register 40 b

BAS	LIMIT

16 b limit, 23 bázová adresa segmentu

↳ po prepnutí do režimu PVAM sa vytvorí 1 globálna tabuľka a register GDTR bude ukazovať na jej začiatok

IDTR – interrupt descriptor table register → ~~ba~~ ukazuje na tabuľku ~~podobnosť~~ deskriptorov prerušení

40 b

LDTR - 16 b

v podstate je to isto ako selektor

-> index do tabuľky globálnych deskriptorov z ktorej sa dostaneme do ~~l~~ tabuľky lokálnych deskriptorov

- vieme, že úloha U má lokálne tabuľky tam a tam a selektory (ktoré vznikli zo segmentových registrov) už idú do tejto lokálnej tabuľky

priznaky	BASE	LIMIT

TR - task register (chová sa ako selektor)

TSS - task segment 42 B (stavový)

15	0
SEL RODICA	0
SP pre	CPL 0
SS pre	CPL 0
SP pre	CPL 1
SS pre	CPL 1
SP pre	CPL 2
SS pre	CPL 2
IP	ES
F	CS
AX	SS
BX	DS
CX	SEL úlohy
DX	
SP	
BP	
SI	
DI	

42

-> obsah TR predošlej úlohy (z ktorej sme prišli)

-> current privilege level

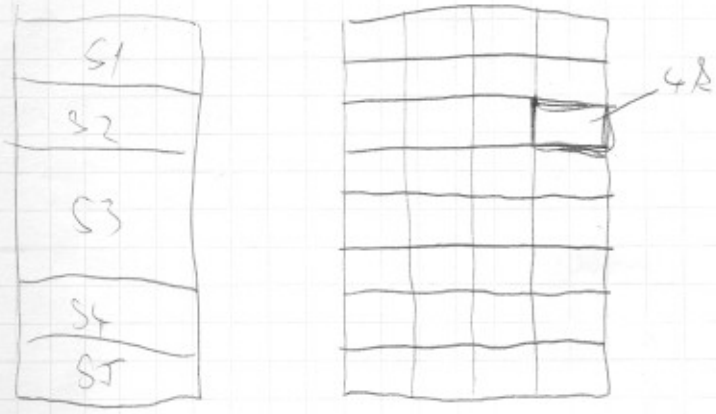
TR -> vbehne do GDT, nájde TSS, zoberie odtiaľ SEL úlohy a ten prenesie do LDTR*, ktorý sa znova pozrie do GDT a odtiaľ zistí, kde sa nachádza LDT (dostane sa jej bázu) a z tých deskriptorov sa už dozvie, kde je segment

- register CS sa z LDT ~~pozr~~ pozrie ~~do~~ ~~d~~ na deskriptor

GDTR -> ukazuje na segment glob. deskr.

* ďalšie informácie

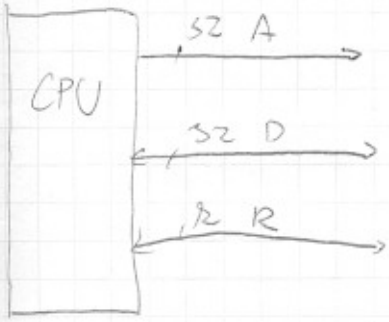
↳ vnútorný mechanizmus virtuálneho priestoru procesora 80286

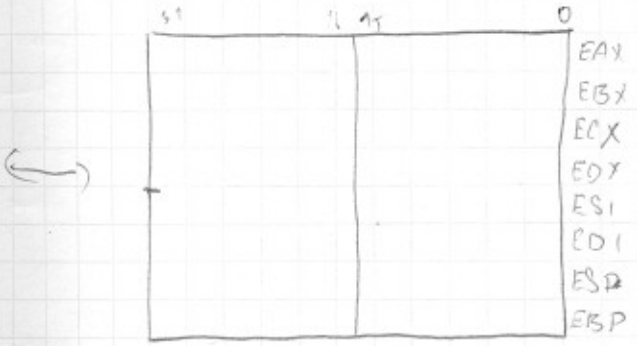


- ~~kb~~ segmenty rozdelíme na stránky

interná fragmentácia -> stránka nebude zaplnená úplne

PROCESOR 386



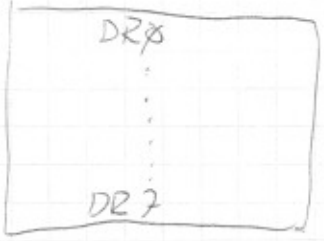


-> používateľsky nepristupné miesto pre dekriptory selektorov, ktoré vnútri premenné segmentových registrov

F-register 32 bitový

RF-resume flag - súvisí s ochranou S-mód 32 b. ladiacich registrov

VM-virtual machine



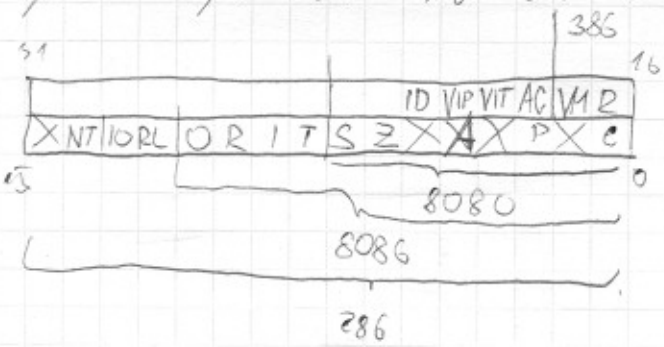
- debug register

9. PREDNÁŠKA

TEST - vnútorné zobrazenia
- 5 možností -> 1 správna
- tlačme kódy čísel a zobrazení
- 15 otázok

386
- za predok brany

- oproti 286 tu chýba register MSW (machine status word) v ktorom mal 1 bit na stavoch zapnutie chráneného režimu
- prebransformoval sa do CR0 + niečo navyše



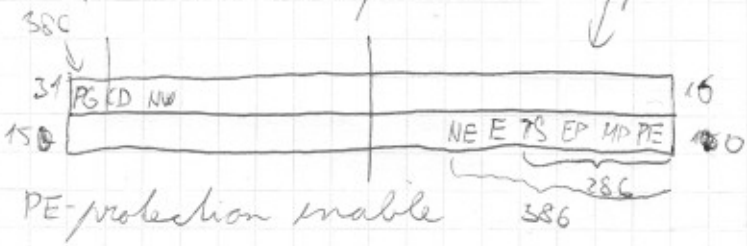
R - resume flag -> súvisí s ladiacimi registrami

VM - virtual mode - indikuje spôsob pracovania úlohy

VIT - virtual interrupt

VIP - virtual IP

ID -> na identifikáciu



CD - cache disable
NW - not write through

PE - protection enable

MP - monitor processor EP - extendet processor

TS - task switch

NE - neth enable (znášanie úloh)

E - extendet

PG - paging

CR2 -> lineárna adresa chybnej stránky

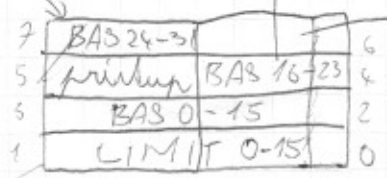
CR3 -> obsahuje adresu adresára tabuliek

CR4 -> má len Pentium

- je tu ešte GDTR - glob. desk. table reg. } pevné
IDTR - interrupt desk. table reg.
LDTR - lok. desk. table reg.
TR - task reg.

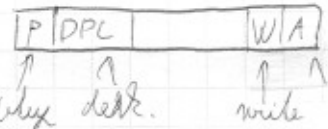
- vyrovnávacia pamäť deskriptora -> 8B

len pri 386 a vyššie



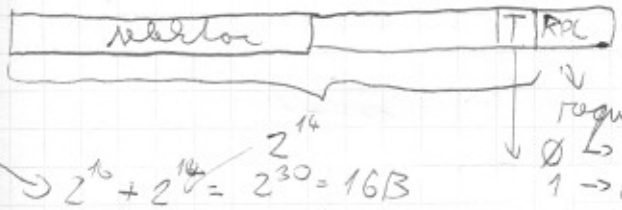
$LIMIT_{16-19}$ využité 4 bity

príznak



príznak desk. prav level (najvyššia 0, najnižšia 3) write access

~~sel~~ selektor

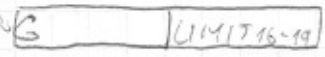


request privileg level

0 -> GDT
1 -> LDT

$2^{16} + 2^{14} = 2^{30} = 16B$

-> virtuálneho pam. priestoru pri 286



G - granularity -> ak 0 segment je do 1MB (daný vektorom limitu)
-> ak 1 hodnota limitu sa násobí 4kB

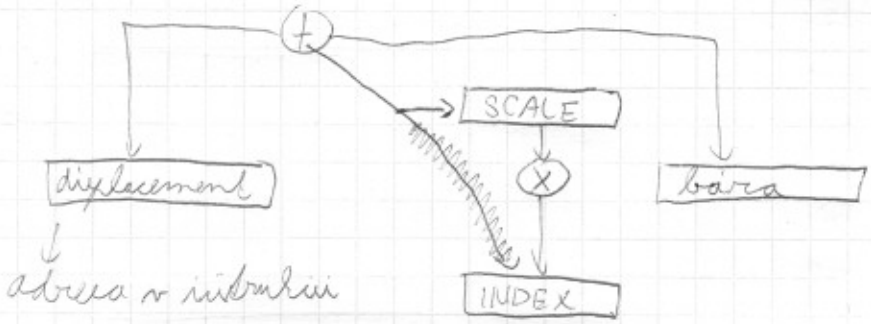
- virtuálny priestor = $\underbrace{\text{fyz } 2^{32} \cdot \text{virt } 2^{14} \text{ selektor veľkých } 2^{32}}_{2^{46} = 64\,TB}$

Logická adresa

z ~~BAS~~ deskriptora ho stočíme na dostanem pomocou SEL

lineárna adresa

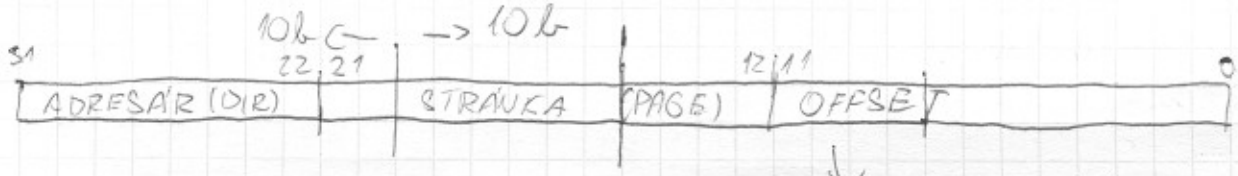
OFFSET = Efektívna adresa



SCALE -> násobíme ním index podľa toho koľko B prvok chcem adresovať

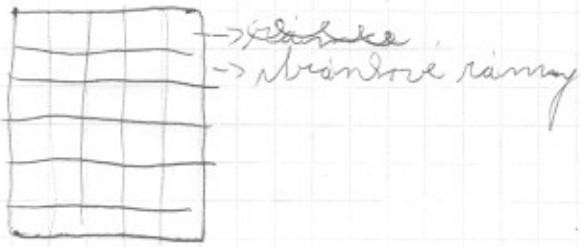
STRÁNKOVANIE

lineárna adresa

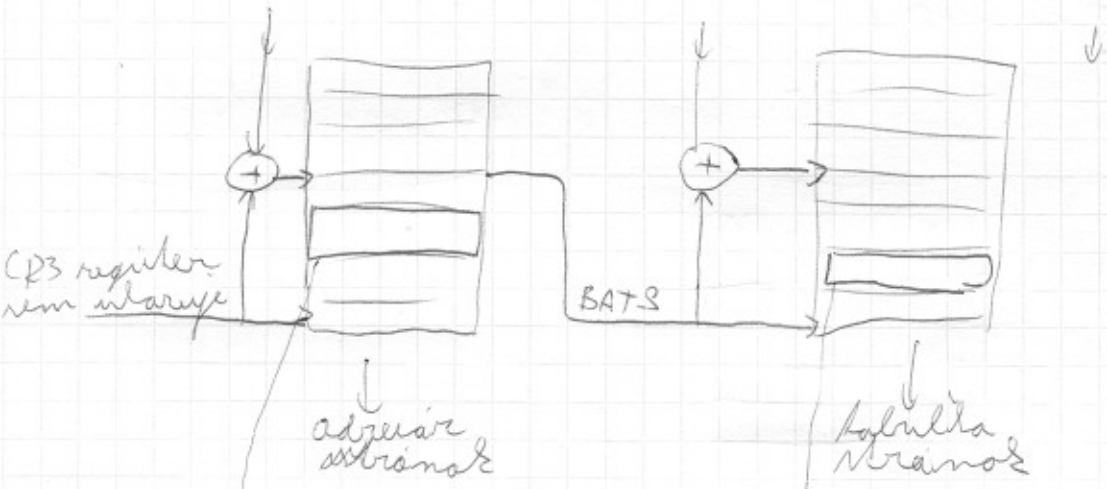


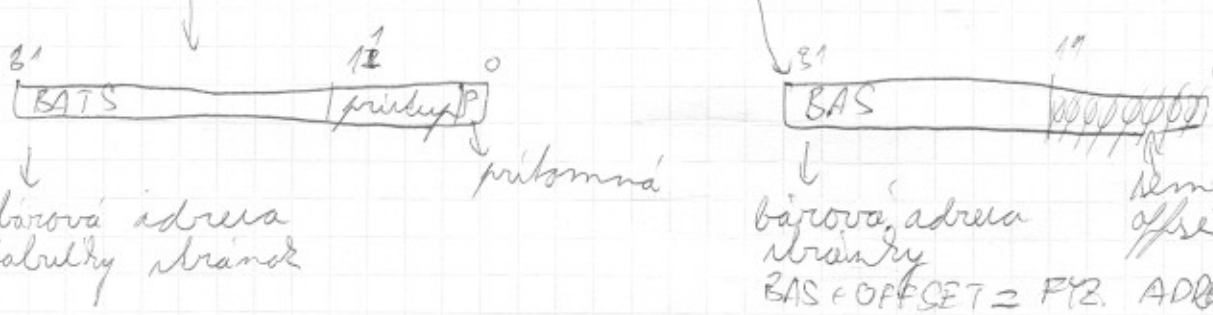
v rámci aktuálnej stránky s ktorou chcem pracovať (adresujem len 4kB)

fyz. adr. priestor



- 2 úrovňové stránkovanie





P: prítomná

báza tabuľky stránok

báza stránky

BAS + OFFSET = FYZ. ADRESA

8086 ⟷ 8087 → aritmetický koprocesor (pohyblivá rodina)

↓

80286 ⟷ 80287

80386 ⟷ 80387 ⟷ CASHE

↓

80486 → dalo sa to dokopy 8K CASHE

↓

80586 → PENTIUM → aby sa to dalo patentovať

A ⟶ 32 b
D ⟷ 64 b
R ⟷ 3 b

CASHE → DATA 8 (16) K
→ INSTR 8 (16) K } ⟶ PENTIUM-PRO
L_1
L_2 - 128 KB (256 kB)

PENTIUM MMX
↳ rozšírenie o podporu multimediálne aplikácie

MMX+PRO = PENTIUM II

⇓

P-XEON - L_2 rádovo aj niekoľko MB

PENTIUM III → rozšírenie inštrukčnej rady

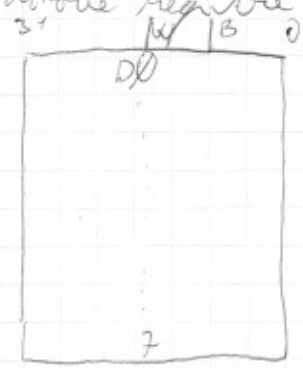
PENTIUM 4 → pridaná ďalšia pamäť L3

MOTOROLA

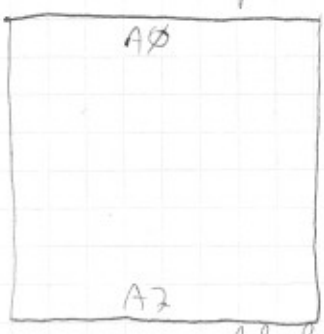
8 bit 6800

16 bit 68000 → vnútorný princíp ostával zachovaný
10 → iba to bol 32 bitový systém
20 ≈ 286
30 ≈ 386
40 ≈ 486

dátové registre



adresové reg.

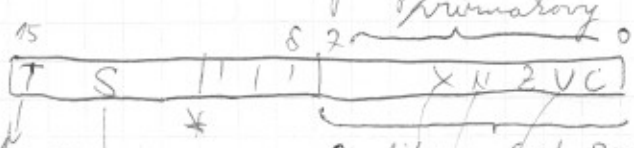


register A7 = SP

IP = PC má 32 b

SR - stavový register 16 b

primárny



* nastavuje sa 7 úrovní prerušenia
dosiahne sa rovný alebo vyšší ukazovateľ

trace

1-krokový režim

systémový režim

Condition Code Register

overflow

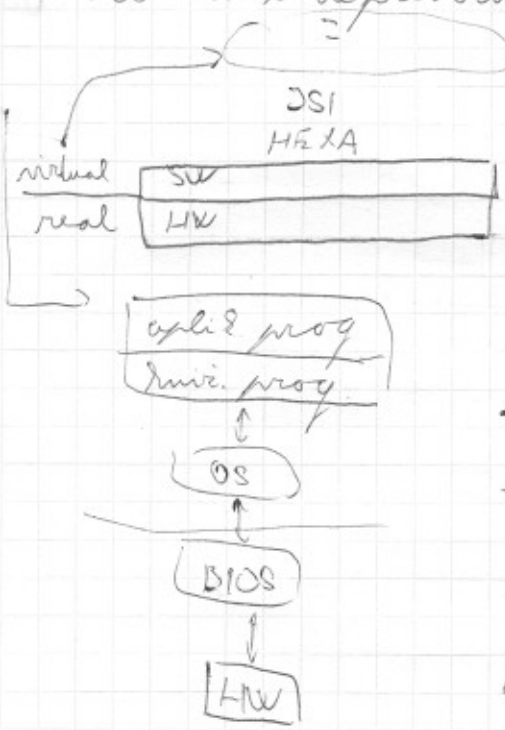
prenos

extended

OPERÁCIA zdroj, cieľ -> naopak ako INTEL

(11) PREDNÁŠKA 10.5.2006

1 -> čo sa bude preberať dnes nebude na skúške :-)



- preklad ↕ viacnásobný
- linkovanie
- zavádzanie
- implementácia
  - makro
  - procedúry
  - virtuálna pamäť

- používateľia
- programátory – špeciálne
  - obyčajné adresáre

Procesy -> realizácia úloh

POŽIADAVKY NA SYSTÉMOVÉ PROGRAMOVANIE:

1, spoľahlivosť <-> správnosť
2, Rýchlosť <-> reálny čas
3, optimálne nároky na tech. prostriedky
4, náklady a rýchlosť tvorby
5, jednoduchosť používania (dokumentácia)
6, tolerancia voči chybám
7, modulárnosť
8, prenositeľnosť

SKÚŠKA

24.5.2006

- dá to ité ako už bolo, širší rozsah otázok

v CPU

$11^{10} \rightarrow 8^{00}$ $13^{00} \rightarrow$ last chance

$13^{05} \rightarrow 9^{30}$ → nie je

$15^{00} \rightarrow 11^{00} \rightarrow$ ÚA

$16^{55} \rightarrow 12^{30}$

55 otázok → 60 min

(10) PREDNÁŠKA

BASIC, PASCAL, C

JSI – MOV, AX, 20H

SW HEXA 3 E 2 0] asembler (vulladač)

0011 1110 0010 0000

BASIC → zdrojový jazyk • analýza

↳ interpretačný j. • interpretácia

PASCAL → zdroj. jazyk • analýza

• kompilácia

PREKLADAČ → zabezpečí prechod zo zdrojového do cieľového jazyka

na 2 fázy: 1) fáza analýzy – niekoľko úrovní:

- lexikálna analýza (rozlišovanie povolených znakov)
- syntaktická
- sémantická → zaoberá sa významom znakov

2) fáza syntézy

- interpretácia
- kompilácia [spustiteľný súbor
  [premiestniteľný ~~súbor~~ modul

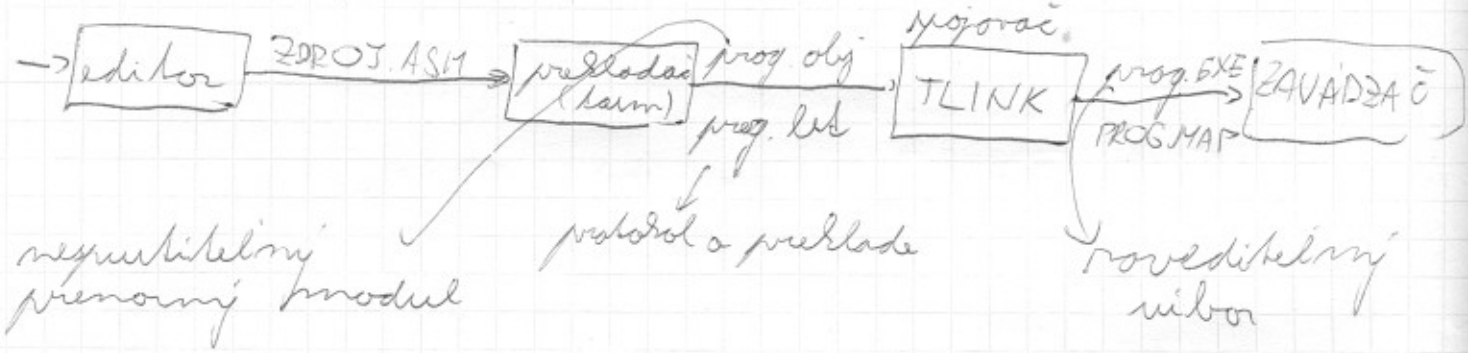
Zvolný cyklus programu:

1, def. problému
2, editor
3, preklad
4, spojovač
5, zavádzač
6, spustenie
7, ladenie
8, použitie

Prekladač - viac prechodov

1. prechod -> vygenerovanie - tabuľky symbolov
- tabuľky inštrukcií

2. prechod -> generovanie cieľového programu
-> generovanie protokolu o preklade



rezidentný prekladač (tasm) -> funguje na stroji, na ktorom bude výsledný program aj pracovať (spustený)

krížový prekladač -> prekladá, program bude púšťaný na inom syst.

prog.map -> protokol o linkovaní (info o segmentoch a p.)

linker spája viacero segmentov? PROG 1 ? 3 .OBJ } prog. exe

každý *.OBJ má vlastné segmenty

ZAVÁDZAČ:

- pozerá sa na hlavičku .EXE súboru, ktorá obsahuje:

- rozsah položek, kt. treba upravit
- umiestnenie zásobníkového a kódového segmentu
- rel. hodnota IP registra
- 40 5A h (začiatok hlavičky)

---

jedna má 512 B

zavádzač vygeneruje PROGRAM STATUS PREFIX -> 256 B -> info o zavedenom súbore
- kto program používa
- kto pravome konbrolu po skončení

spustiteľný súbor .COM (command)
↳ nie dlhší ako 1 segment (≤ 64 kB) -> všetky 3 segmenty sú v jednom
nemá