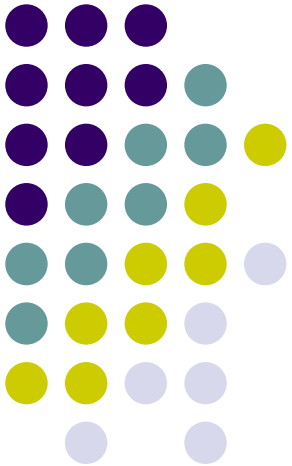
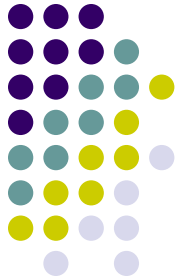
Počítače řady IBM PC

Mikroprocesor Intel 8086



IBM PC/XT



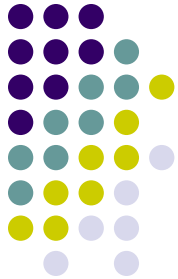
- 8. března 1983 byl uveden na trh počítač **IBM PC/XT**
- Tím začíná éra **4. generace počítačů**
- PC = Personal Computer
- XT = eXtended Technology
- Jde o nástupce počítače IBM PC, který měl ale pouze osmibitový procesor Intel 8088
- Zde je použit 16-bitový procesor Intel 8086 – první z velké rodiny mikroprocesorů, jehož moderní potomky používáme v dnešních moderních osobních počítačích
- Z historického pohledu se právě IBM PC/XT stává důležitým mezníkem
- Počítač má standardně 128kB paměti RAM, 10 MB hard disk a 5 1/4" disketovou mechaniku (kapacita 360 kB)
- Tento počítač se stává praotcem celé generace osobních počítačů v podobě, v jaké je známe dodnes
- Postupně se nyní seznámíme s vývojem počítačů této rodiny, s jejich mikroprocesory a hardwarem

IBM PC/XT

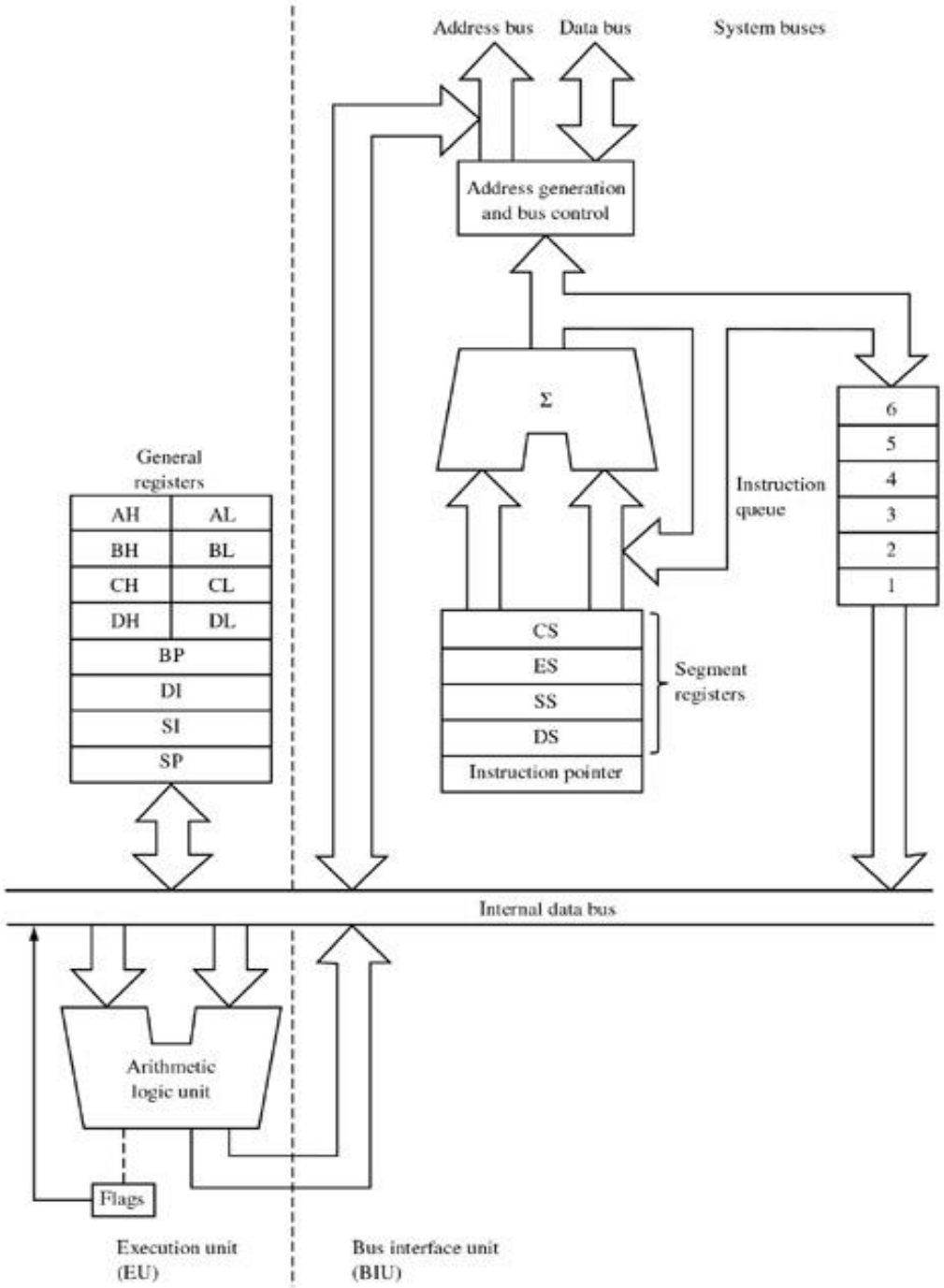
Intel 8086

- navržen v roce 1978
- 16-bitový mikroprocesor
- Umí adresovat paměť max. 1MB (20-bitová adresace)
- První z řady procesorů rodiny x86
- obsahuje 29000 tranzistorů
- Pouzdro DIL40 (40 vývodů)
- 14 šestnáctibitových registrů
- Střadačová architektura
- Výrobní proces 3 µm (3000 nm)
- Varianty: 8086 (5MHz), 8086-1 (10 MHz), 8086-2 (8 MHz)
- Výpočetní výkon zhruba 0,3 až 0,75 MIPS

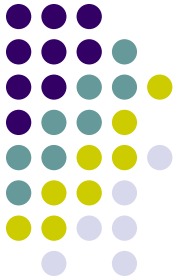
Vnitřní struktura 8086



- Procesor uvnitř se skládá ze dvou jednotek
  - **BIU** – Bus Interface Unit
    - Obstarává komunikaci mikroprocesoru s okolím
    - provádí čtení strojového kódu instrukce z paměti, čtení dat a zápis výsledků
    - Bajty strojového kódu jsou čteny v předstihu a ukládány do 6-bajtové fronty FIFO
    - Po uvolnění alespoň dvou bajtů v instrukční frontě se BIU snaží o načtení dalšího strojového kódu do fronty
    - Při provádění instrukce skoku se obsah fronty vyprázdní (dopředu načtené instrukce ne neprovedou, ale přeskočí)
  - **EU** – Execution Unit
    - obsahuje ALU
    - zpracovává operandy, ovládá registry a příznakové bity
    - nemá přístup k okolnímu světu

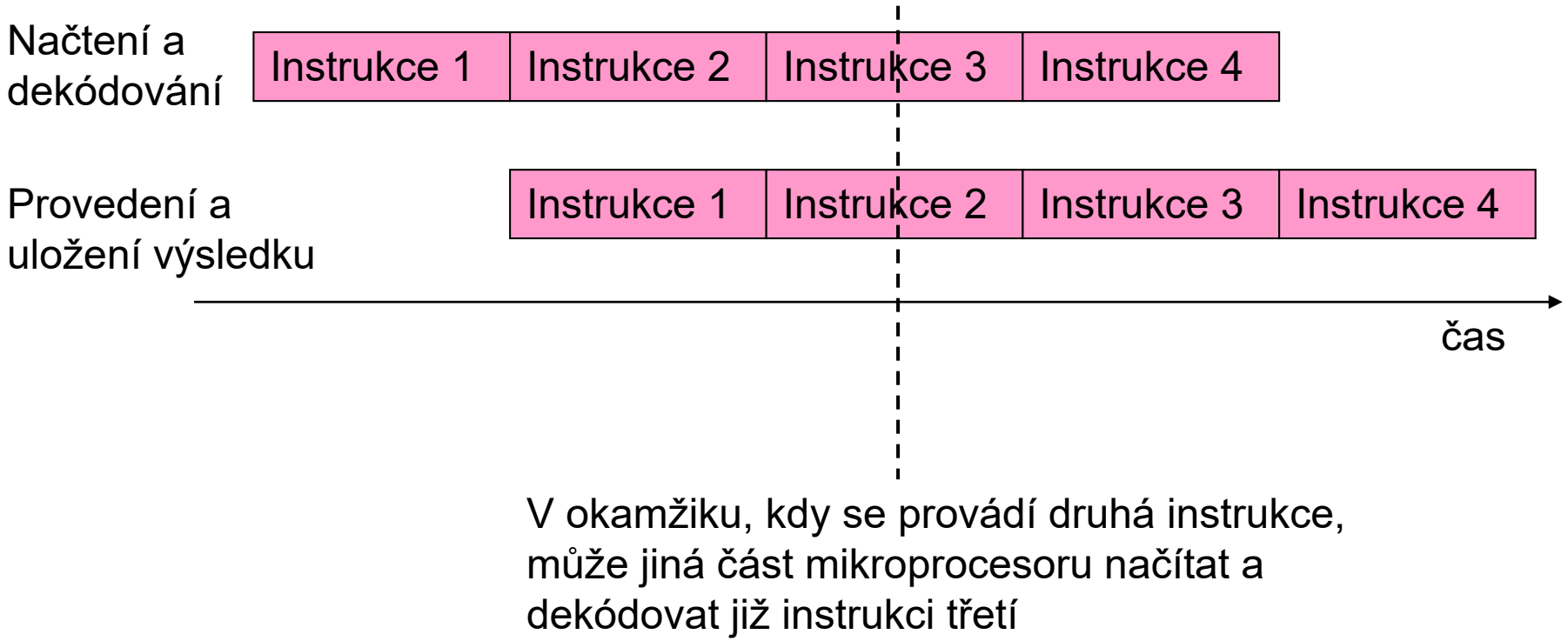

Address bus
Data bus
System buses
Address generation and bus control
Σ
6
5
4
3
2
1
Instruction queue
General registers
AH
AL
BH
BL
CH
CL
DH
DL
BP
DI
SI
SP
CS
ES
SS
DS
Instruction pointer
Segment registers
Internal data bus
Arithmetic logic unit
Flags
Execution unit (EU)
Bus interface unit (BIU)

Pipelining



- **Pipelining** je zřetězené zpracování, či překrývání instrukcí
- Základní myšlenkou je rozdělení zpracování jedné instrukce na více fází mezi různé části procesoru a tím i dosažení možnosti zpracovávat více instrukcí najednou
- Zpracování instrukce je v 8086 rozděleno na **2 části**
  - Načtení a dekódování instrukce (provádí **BIU**)
  - Provedení instrukce a případné uložení výsledku (provádí **EU**)
- To vedlo k vytvoření procesoru složeného ze dvou spolupracujících jednotek **EU** a **BIU**
- každá jednotka realizuje jinou fázi zpracování
- U moderních procesorů se zřetězení se stále vylepšuje a instrukce se dělí na více fází, tudíž současně se v procesoru zpracovává několik instrukcí, přičemž každá z nich je v jiné fázi dokončení

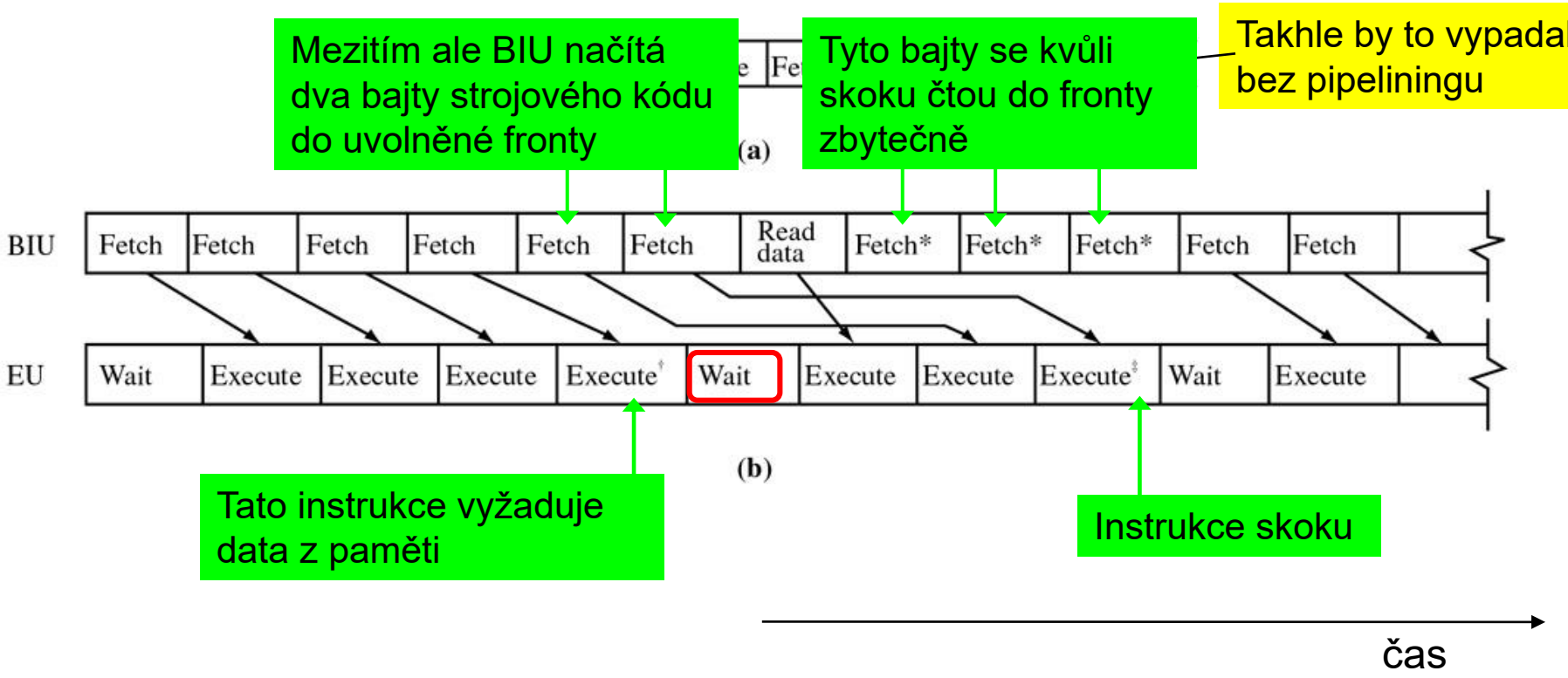
Pipelining



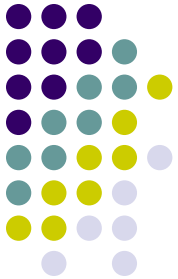
V okamžiku, kdy se provádí druhá instrukce, může jiná část mikroprocesoru načítat a dekódovat již instrukci třetí

Pipelining - i8086

- Jednoduchý pipelining umožňuje EU vykonávat instrukci bez prodlev spojených s načtením strojového kódu instrukce z paměti
- EU se občas ocitá bez práce ve „Wait“ stavu, pokud zpracovává instrukci, která potřebuje načíst operandy z paměti

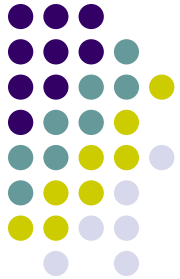


Adresace paměti



- 8086 umí adresovat až **1 MiB** paměti
- Pro adresaci jednoho Megabajtu je třeba **20-bitová** adresa (2^{20}=1048576)
- Všechny registry 8086 jsou ale pouze **16-bitové**
- K paměťové buňce tedy program přistupuje jinak než uvedením její přesné adresy v intervalu <0;FFFFFh>
- Program generuje tzv. **logickou adresu**, která je složena ze **segmentu** a **offsetu** a BIU tuto adresu překládá na skutečnou **fyzickou adresu** paměťové buňky

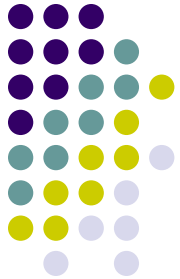
Adresace paměti



- Mikroprocesor tedy komunikuje s pamětí pomocí 20-bitové **fyzické** (skutečné) adresy
- Program ale přistupuje k paměťovému místu prostřednictvím segmentu a ofsetu.
- Takto složenou adresu nazýváme jako **logickou** – jde o dvě šestnáctibitové informace (protože registry jsou 16-bitové) - kombinací vzniká skutečná **fyzická** adresa

- **Fyzická adresa** se musí při každém přístupu do paměti **vypočítat** ze dvou 16-bitových informací
- Tento výpočet provádí 20-bitová sčítačka v **BIU**

Adresace paměti

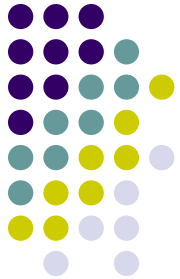


- Do paměti lze vstupovat přes **čtyři segmenty**
  - **Datový segment** (slouží k ukládání zpracovávaných dat)
  - **Kódový segment** (obsahuje strojový kód programu – programátor dopředu neví, kde v paměti bude tento segment umístěn. Veškeré skoky v programu směřují na relativní adresu (cíl skoku je určen pozicí v kódovém segmentu)
  - **Zásobníkový segment** (mikroprocesor v něm buduje zásobník LIFO)
  - **Pomocný datový segment** (další prostor pro zpracovávaná data)

- **Segment** je souvislý blok paměti o velikosti **64 kB**
- Pozici uvnitř segmentu tak lze určit **16-bitovým offsetem**

- Pokud se má program vrátit zpět na začátek, provede programátor skok na pozici s offsetem nula v kódovém segmentu. Programátor neví, kde přesně bude v paměti jeho program uložen. To může být na každém počítači při každém spuštění tohoto programu někde jinde. Programátor tedy nemůže do programu psát „absolutní adresy“, protože umístění programu v paměti je pokaždé jiné

Adresace paměti



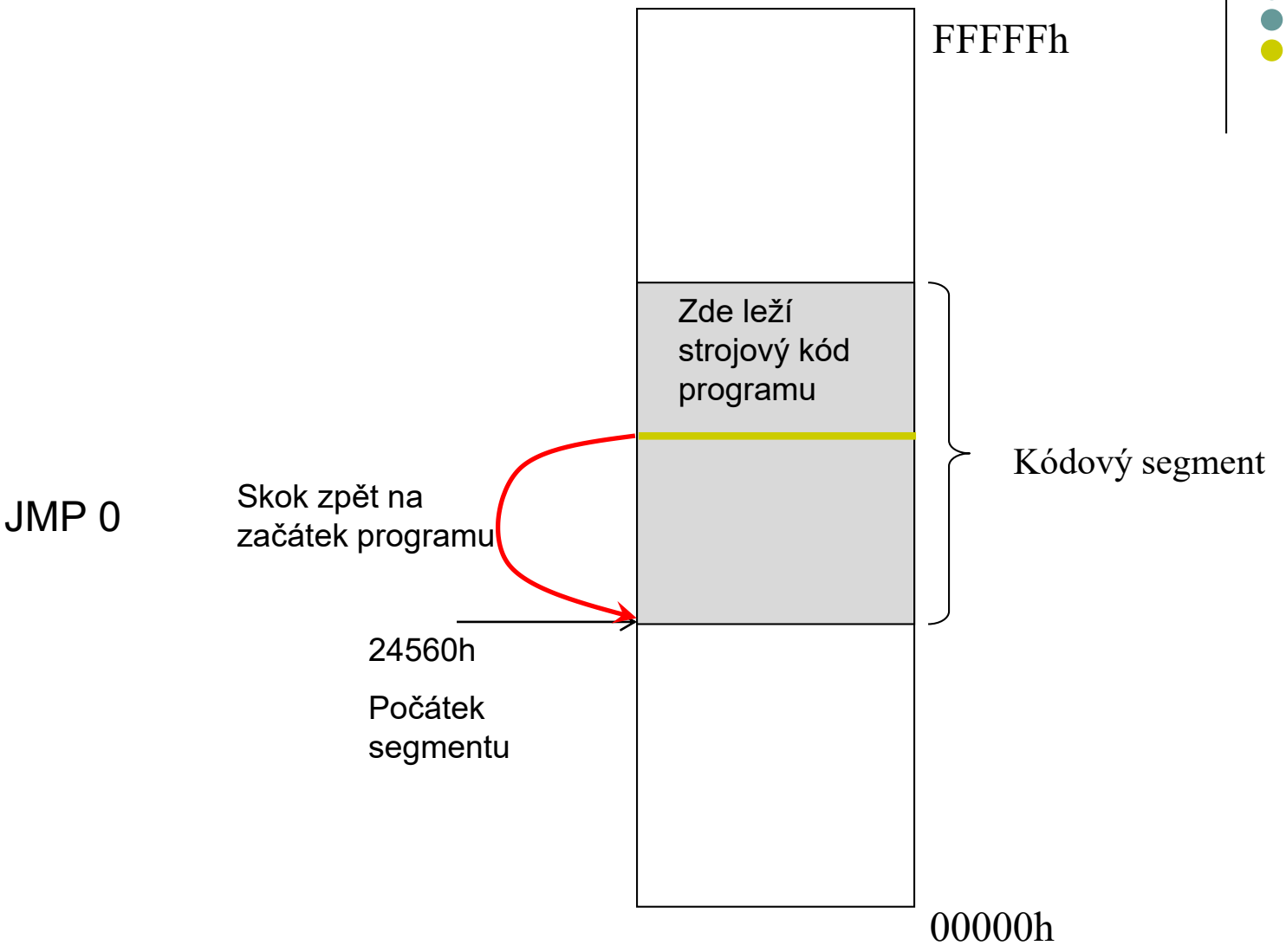
Příklad

- Programátor chce v programu provést skok zpět na jeho začátek (vrátit se na první instrukci programu)
- Toto provede povelem **JMP 0**

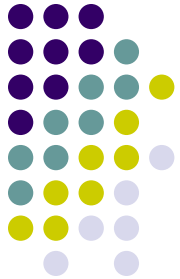
- **JMP** = Jump (zápis v assembleru)
- **0** = offset (na jakou pozici v kódovém segmentu se má skočit)

- BIU vypočítá **skutečnou** adresu, na kterou se skočí (ze které se přečte další bajt strojového kódu)
- Tato skutečná **fyzická adresa** se vypočítá tak, že se **offset 0** přičte k **počáteční adrese kódového segmentu**, ve kterém je v paměti uložen strojový kód programu
- Pokud má kódový segment počáteční adresu například 24560h, pak skok na pozici s offsetem 0, znamená vlastně skok na adresu 24560h
- Programátor, ale nemůže v programu psát JMP 24560h, protože dopředu neví, kde v paměti bude jeho program uložen – to je pokaždé jinde

- **Fyzická adresa = počáteční adresa segmentu + offset**


FFFFFh
Zde leží
strojový kód
programu
Kódový segment
JMP 0
Skok zpět na
začátek programu
24560h
Počátek
segmentu
00000h

Adresace paměti



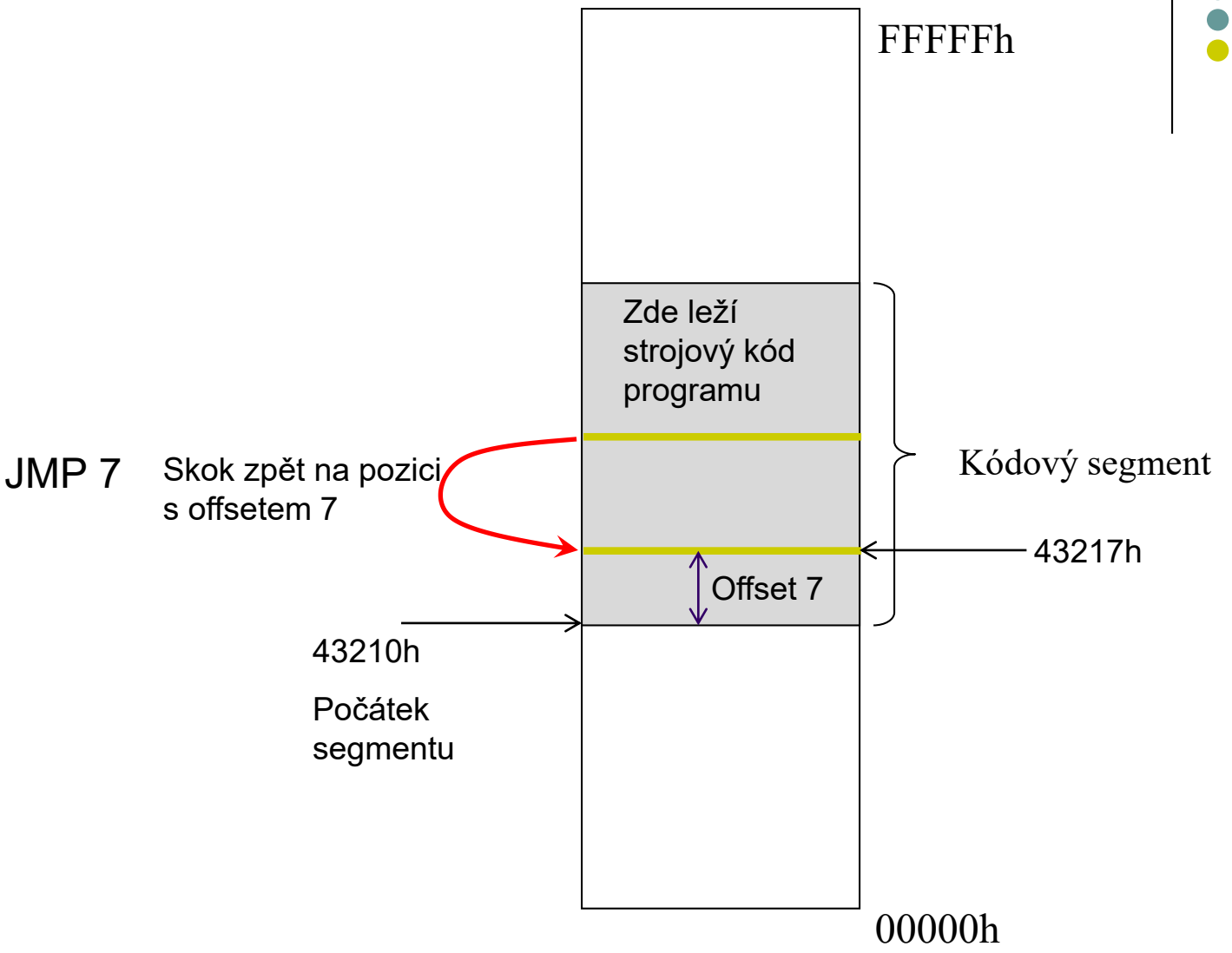
Příklad

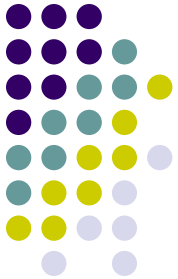
- Počáteční adresa kódového segmentu je 43210h
- Programátor provedl v programu skok povelem **JMP 7**
- **Z jaké adresy se přečte další instrukce strojového kódu?**

- **JMP** = Jump (zápis v assembleru)
- **7** = offset (na jakou pozici v kódovém segmentu se má skočit)

- BIU vypočítá **skutečnou** adresu, na kterou se skočí (ze které se přečte další bajt strojového kódu)
- Tato skutečná **fyzická adresa** se vypočítá tak, že se **offset 7** přičte k **počáteční adrese kódového segmentu**, ve kterém je v paměti uložen strojový kód programu
- Kódový segment má počáteční adresu 43210h (na této adrese leží první bajt strojového kódu programu)
- Fyzická adresa = 43210h + 7 = 43217h
- Další bajt strojového kódu se přečte z adresy 43217h

- **Fyzická adresa = počáteční adresa segmentu + offset**

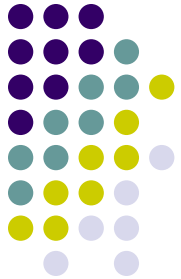

FFFFFh
Zde leží
strojový kód
programu
JMP 7
Skok zpět na pozici
s offsetem 7
Kódový segment
43217h
Offset 7
43210h
Počátek
segmentu
00000h



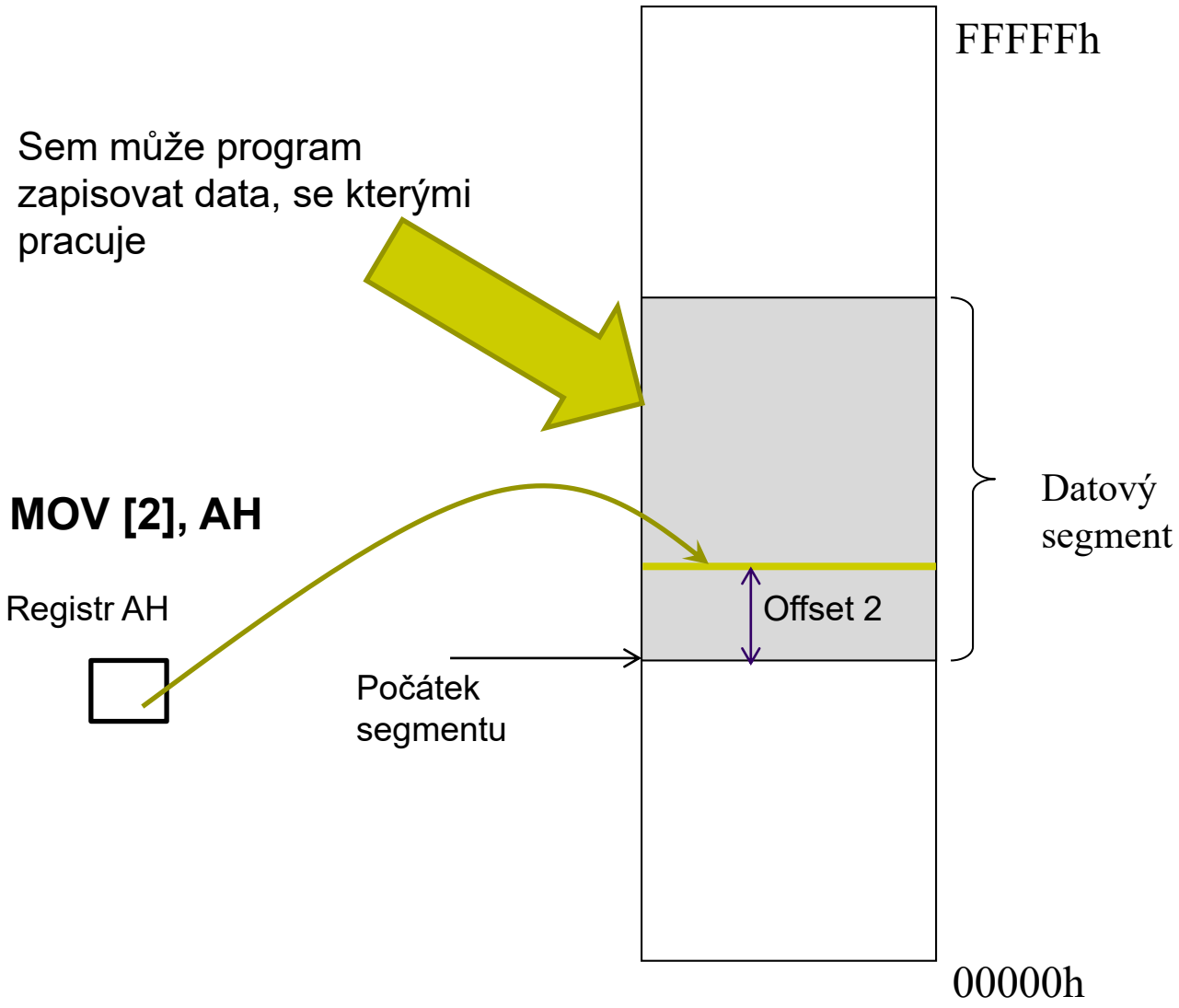
Příklad

MOV [2],AH

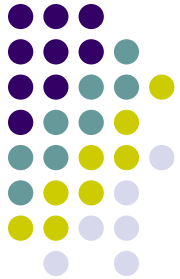
- Touto instrukcí chce programátor zapsat (přesunout) obsah registru AH do paměti
- Číslo 2 v hranatých závorkách představuje **offset**
- Instrukce neříká, že by se data měla do paměti zapsat na fyzickou adresu 2
- Data se zapíší do paměti do **datového segmentu** na místo ležící posunuté o 2 adresy od počátku tohoto segmentu
- Skutečná **fyzická adresa**, na kterou se bude zapisovat musí tedy být mikroprocesorem vypočítána (provede se to v BIU) jako součet bázové adresy datové segmentu + 2



- Programátor nemůže v programu používat instrukce, které zapisují data na konkrétní fyzické adresy, protože dopředu neví, jak velká bude paměť počítače, na kterém jeho programu poběží a co bude v této paměti kde uloženo ještě před tím, než bude spuštěn jeho program
- Programátor ví pouze to, že jeho program po spuštění bude mít pro ukládání dat k dispozici **datový segment**, který je velký 64 KB
- Do tohoto segmentu (do této oblasti paměti) smí program ukládat svá data
- Instrukcemi v programu říkáme, kam do tohoto datového segmentu se má přistupovat – tedy vybíráme offsetem pozici uvnitř tohoto segmentu


FFFFFh
Sem může program zapisovat data, se kterými pracuje
MOV [2], AH
Registr AH
Offset 2
Počátek segmentu
Datový segment
00000h

Adresace paměti



Příklad

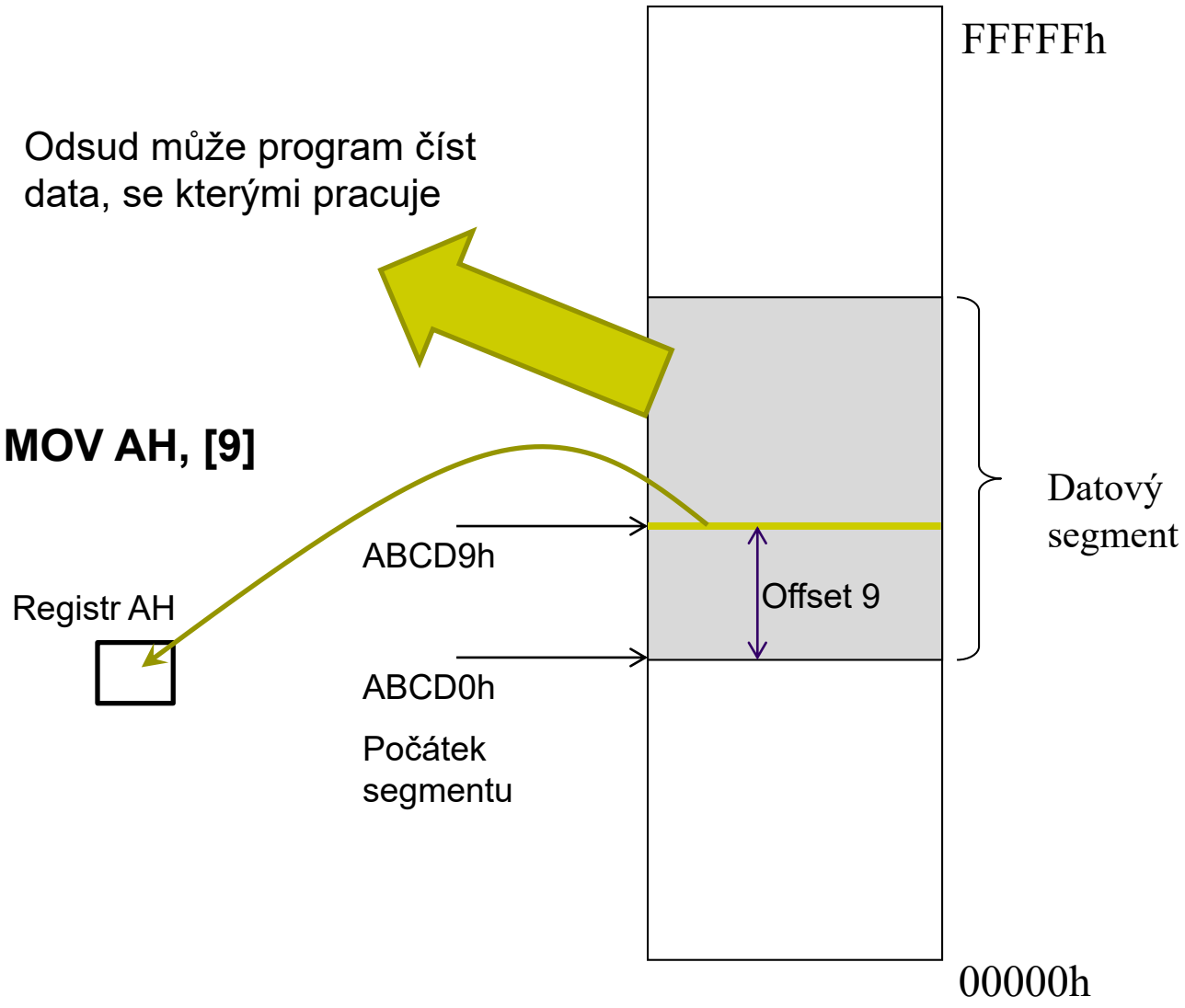
- Programátor chce přečíst bajt z paměti
- Data, se kterými program pracuje, jsou uložena v **datovém segmentu**
- Datový segment začíná na adrese **ABCD0h**
- Čtení dat z paměti se provádí povelem **MOV**

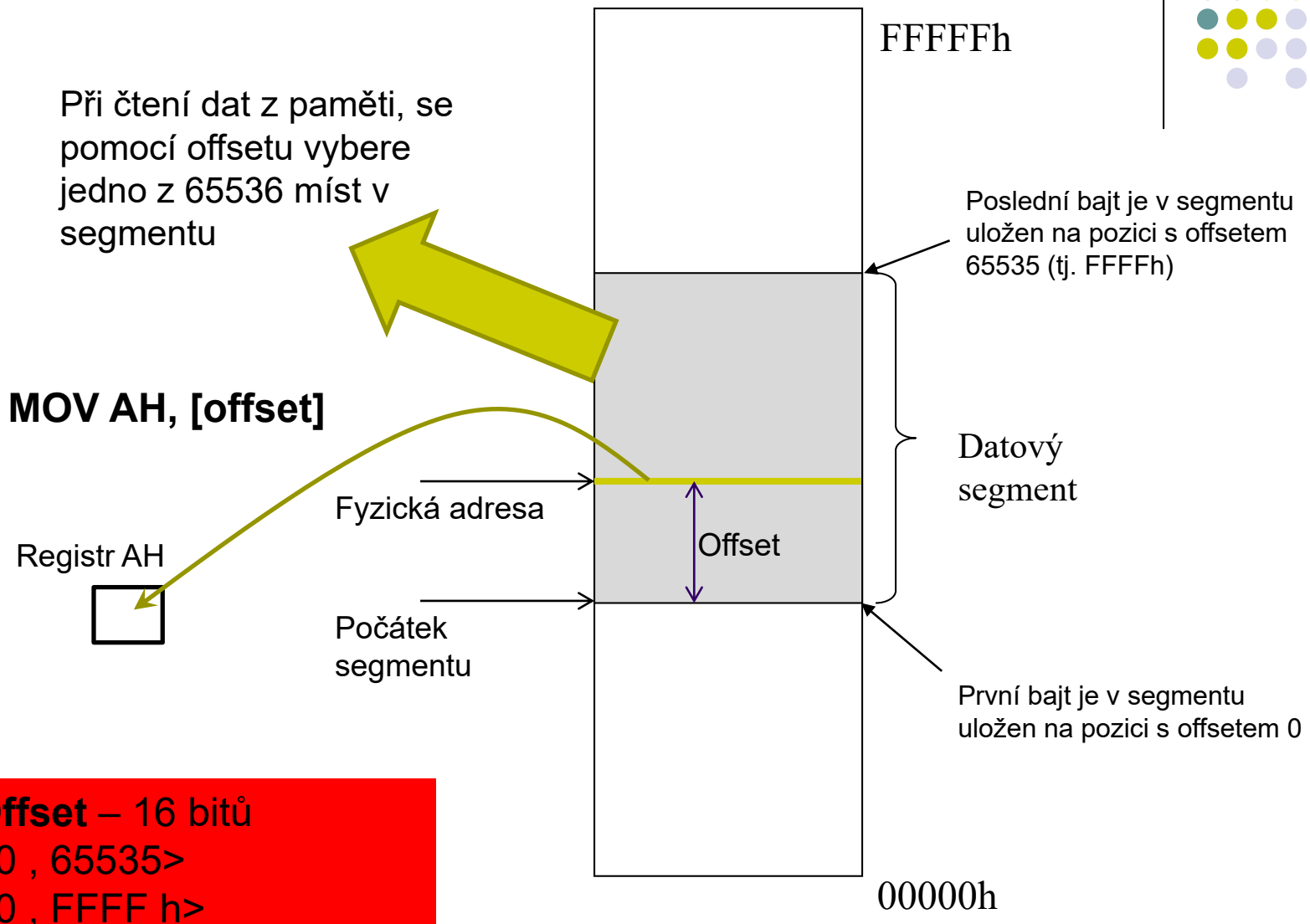
  MOV AH, [9]

- **MOV** = MOVE (zápis v assembleru)
- **AH** = registr, do kterého se má přečtený bajt uložit
- **9** = offset (z kolikáté pozice v datovém segmentu se bude číst)

- BIU vypočítá **skutečnou** adresu, ze které se přečtou data
- Tato skutečná **fyzická adresa** se vypočítá tak, že se **offset 9** přičte k **počáteční adrese datového segmentu**
- **Fyzická adresa = ABCD0h +9 = ABCD9h**

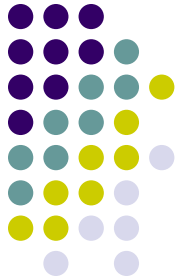
- Datový bajt se přečte z paměti z adresy ABCD9h


FFFFFh
Odsud může program číst
data, se kterými pracuje
MOV AH, [9]
Datový
segment
ABCD9h
Offset 9
Registr AH
ABCD0h
Počátek
segmentu
00000h



Offset – 16 bitů
<0 , 65535>
<0 , FFFF h>

Adresace paměti



Příklad

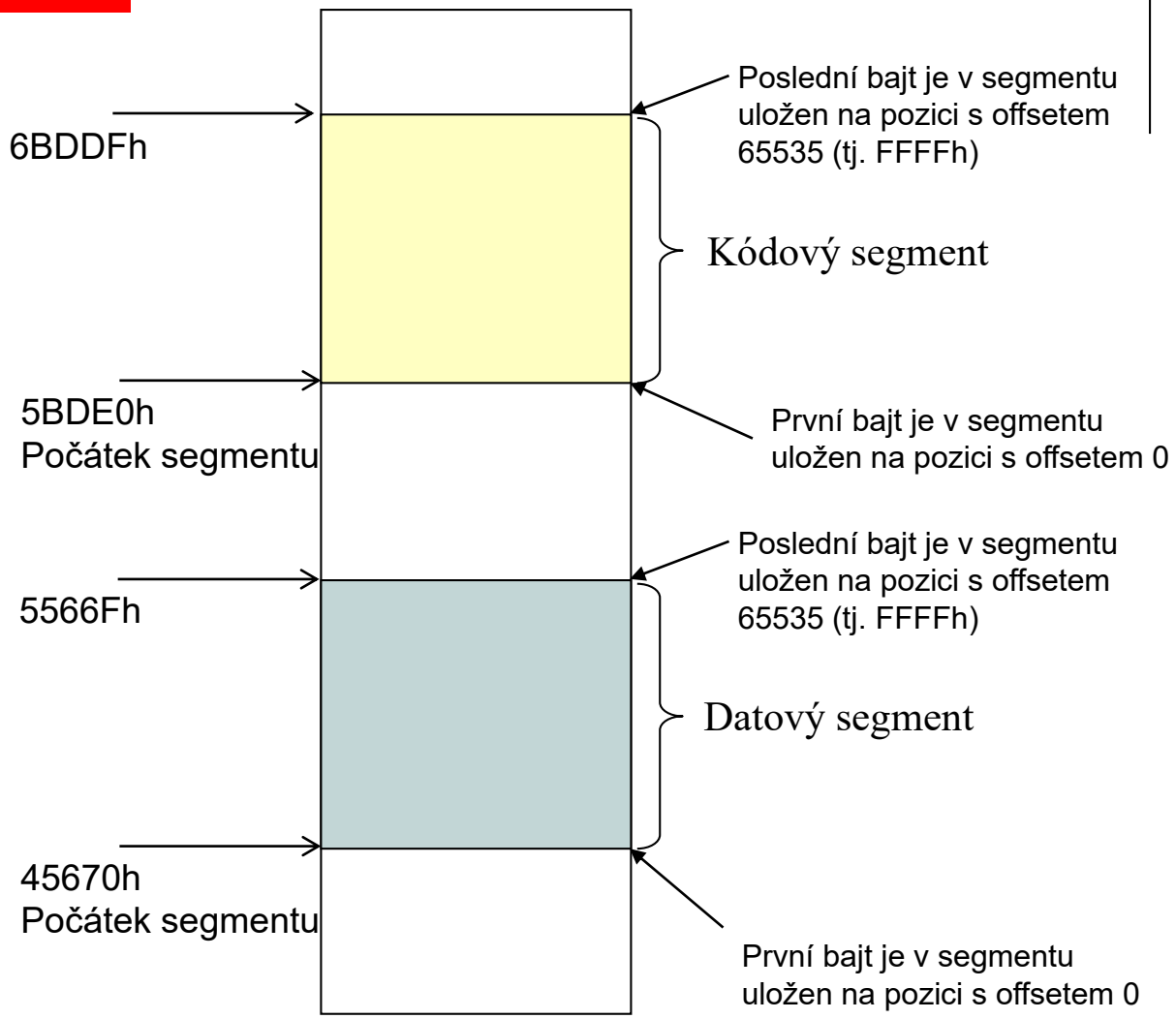
- Datový segment začíná na adrese 45670h
- kódový segment začíná na adrese 5BDE0h
- **Na jakých adresách tyto segmenty končí?**

- Každý segment má velikost 64 KB, tj. 65536 B
- Pozice bajtu uvnitř segmentu je určena 16-bitový offsetem
- Poslední bajt leží na adrese, na které segment končí
- Poslední bajt má offset FFFFh

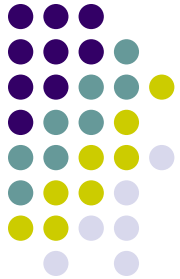
- Datový segment tedy končí adresou 45670h + FFFFh = 5566Fh

- Kódový segment tedy končí adresou 5BDE0h + FFFFh = 6BDDFh

Offset – 16 bitů
<0 , 65535>
<0 , FFFF h>



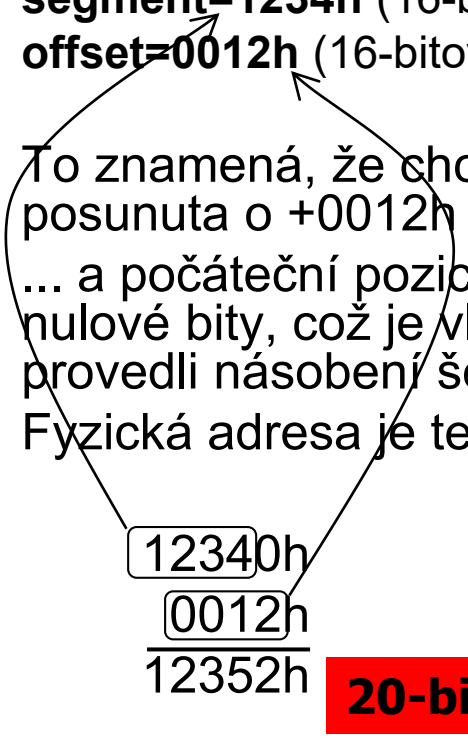
Adresace paměti



- Počáteční adresa každého segmentu může ležet kdekoliv v paměti (tj. mezi 0-FFFFFh) a musí být násobkem **šestnácti**
- čísla, která jsou násobkem **šestnácti,** končí ve dvojkové soustavě **4 nulami**, v **hexadecimální** soustavě **nulou**
- Protože **20-bitová** počáteční adresa segmentu bude mít na konci vždy **4 nuly**, stačí uvádět pouze jejích prvních **šestnáct bitů** (4 nuly vynecháme, víme že tam jsou vždy)
- Informaci o dvacetibitové **počáteční adrese segmentu** tak lze ukládat pouze pomocí **16 bitů**
- **Logická adresa** je tedy tvořena
  - **16-bitovou** informací o počátku segmentu (**počáteční adresou** segmentu oříznutou z 20 bitů na 16 bitů)
  - a **16-bitovým offsetem** (pozicí bajtu uvnitř tohoto 64kB velkého segmentu)

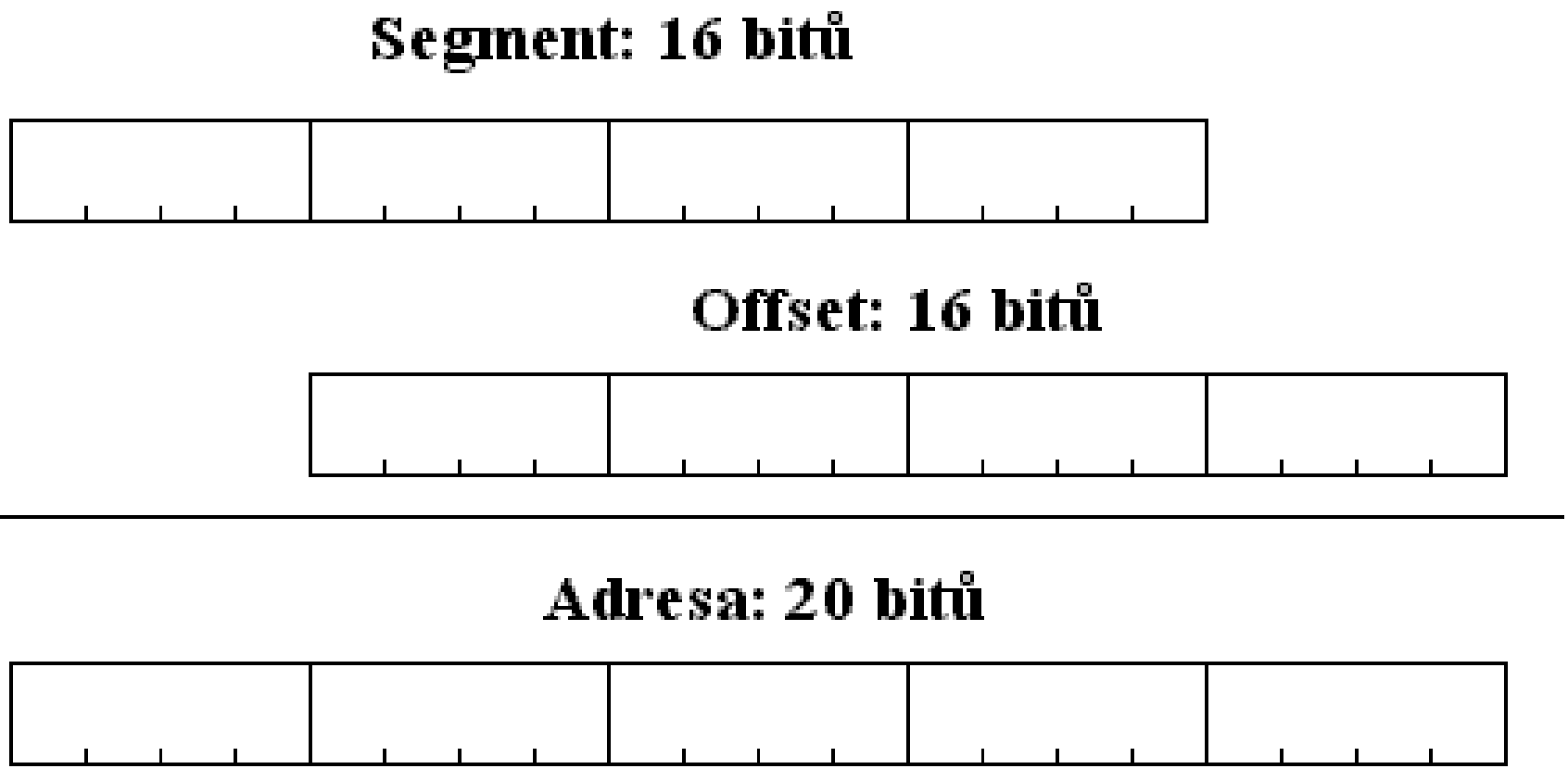
Segment a offset

- Logická adresa
  segment=1234h (16-bitová informace o počáteční adrese segmentu)
  offset=0012h (16-bitová informace o poloze uvnitř segmentu)
- To znamená, že chceme přistupovat k paměťové buňce, která je posunuta o +0012h bajtů vůči počátku segmentu...
- ... a počáteční pozice segmentu v paměti je 12340h (byly přidány čtyři nulové bity, což je vlastně ve dvojkové soustavě totéž jako bychom provedli násobení šestnácti)
- Fyzická adresa je tedy 12340h + 0012h

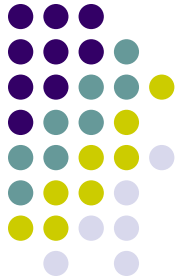


20-bitová fyzická adresa

Výpočet fyzické adresy



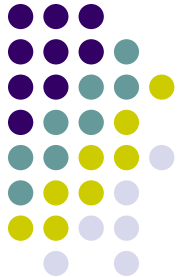
Segment a offset



- K fyzické adresaci tedy potřebujeme **dvě 16-bitové informace**
- Prvním údajem je 16 z 20 bitů **počáteční adresy segmentu**
- Druhým údajem je **offset**

- Do paměti lze vstupovat přes **čtyři segmenty** – kódový, zásobníkový, datový, pomocný datový
- Segment je souvislý úsek paměti o velikosti 64 KB
- Umístění segmentů v paměti je náhodné
- Při každém spuštění programu mohou být segmenty operačním systémem umístěny v paměťovém prostoru někam jinam – tam, kde je zrovna volné místo
- Programátor dopředu neví, kde v paměti, na jakých konkrétních adresách budou uložen program a jeho data a není to ani schopen nijak ovlivnit
- V programu jsou proto uvedeny pouze offsety
- Segmenty mohou být odděleny, ale mohou se i překrývat
- **Počáteční adresa** segmentu bývá také nazývána **bázová adresa**

Segmentové registry



Jsou to registry, v nichž je uložena bázová adresa příslušného segmentu
Tyto registry jsou **16-bitové**

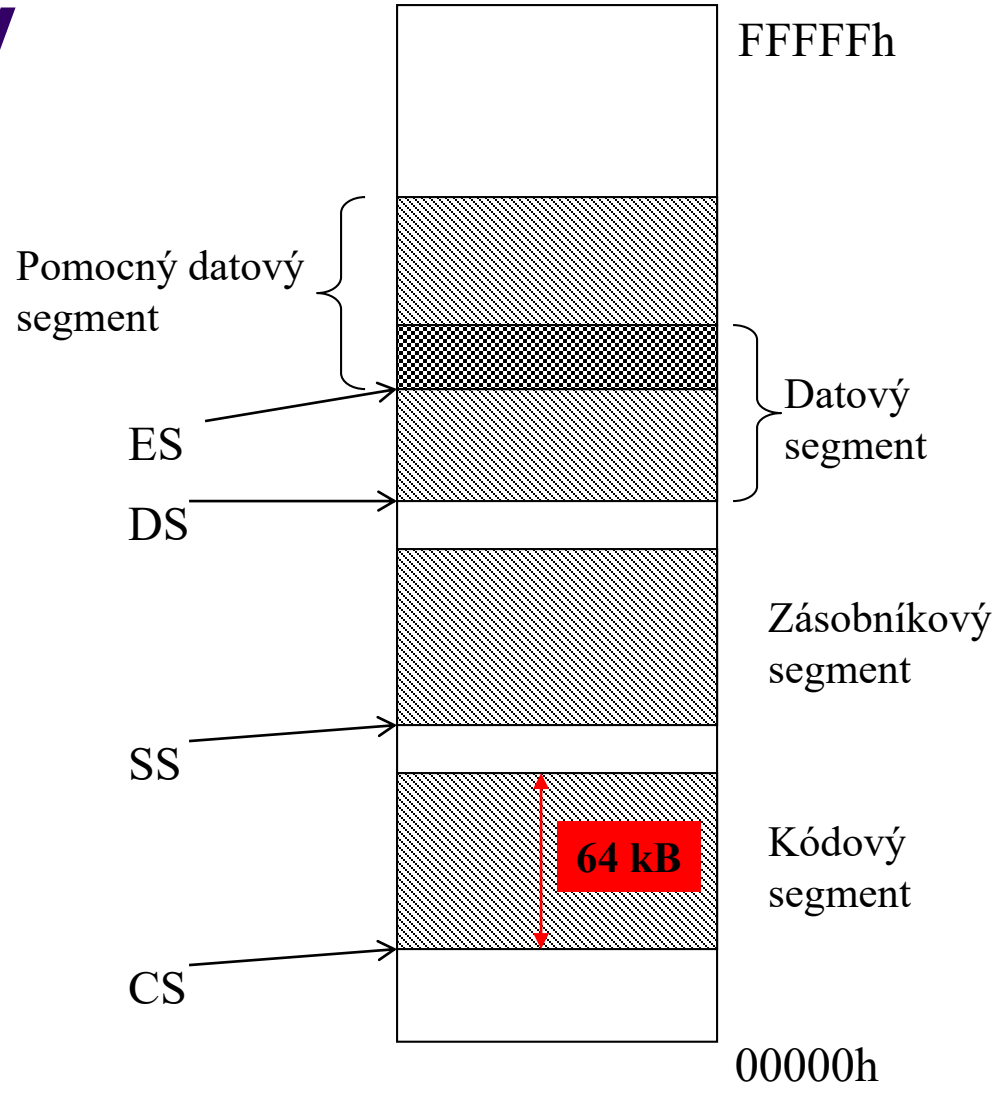
Registr **CS** (Code Segment)
obsahuje informaci o bázové adrese kódového segmentu. V tomto segmentu je uložen strojový kód programu

Registr **SS** (Stack Segment)
ukazuje na počátek zásobníkového segmentu, do kterého jsou ukládány návratové adresy (při volání podprogramu a přerušení)

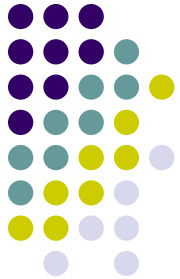
Registr **DS** (Data Segment)
obsahuje informaci o bázové adrese datového segmentu, který obsahuje datové bajty

Registr **ES** (Extra Segment)
obsahuje informaci o bázové adrese pomocného datového segmentu, který má tentýž význam jako běžný datový segment. Některé instrukce implicitně berou data z pomocného segmentu.

Segmenty


FFFFFh
Pomocný datový segment
Datový segment
ES
DS
Zásobníkový segment
SS
64 kB
Kódový segment
CS
00000h

Příklad



- Kódový segment začíná na adrese 87BD0h
- Zásobníkový segment začíná na adrese B4E60h

- **Určete obsah registrů CS a SS**

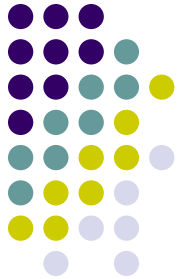
- Počáteční adresa každého segmentu musí být násobkem čísla 16 (tedy dělitelná šestnácti beze zbytku)
- 87BD0h = 1000011110111101**0000** b
- B4E60h = 1011010011100110**0000** b

Užitečných 16-bitů

- Počáteční adresy segmentů jsou 20-bitové, ale informace o ní se dá zakódovat pomocí 16 bitů

- CS = 1000011110111101 b = 87BDh
- SS = 1011010011100110 b = B4E6h

Příklad



- DS = 8A7Eh

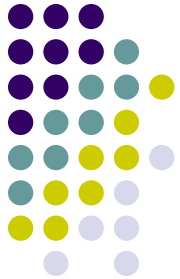
- **Určete na jaké adrese začíná a končí datový segment**

- Počáteční adresa každého segmentu musí být násobkem čísla 16 (tedy dělitelná šestnácti beze zbytku)
- V registru DS je uvedeno pouze užitečných 16 bitů z bázové adresy
- Datový segment začíná na adrese 8A7E0h
- 8A7E0h = 1000101001111110 0000 b

Užitečných 16-bitů

- Poslední bajt v datovém segmentu leží na pozici s offsetem FFFFh a leží na adrese, kterou tento segment končí
- Koncová adresa datového segmentu = 8A7E0h + FFFFh = 9A7DFh

Příklad



- CS = 1B74h
- DS = 2F5Dh
- ES = 48A2h
- SS = 924Ch

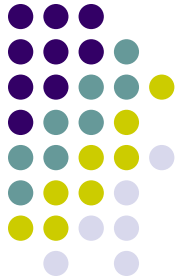
- **Určete na jakou adresu se zapíší do paměti data povelem MOV [1B72h],AL**

- Bajt z registru AL bude zapsán do paměti do datového segmentu
- Datový segment začíná na adrese 2F5D0h (DS=2F5Dh)
- Bajt bude do datového segmentu zapsán na pozici s offsetem 1B72h

- Fyzická adresa = počáteční adresa segmentu + offset
- Fyzická adresa = 2F5D0h + 1B72h = 31142h

- Bajt bude zapsán na adresu 31142h

Příklad



- CS = 1B74h
- DS = 2F5Dh
- ES = 48A2h
- SS = 924Ch

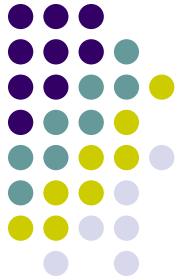
- **Určete na jakou adresu se přeskočí v programu povelem JMP 12h**

- Strojový kód programu leží v kódovém segmentu
- Kódový segment začíná na adrese 1B740h (CS=1B74h)
- Skok směřuje na pozici s offsetem 12h

- Fyzická adresa = počáteční adresa segmentu + offset
- Fyzická adresa = 1B740h + 12h = 1B752h

- Další bajt strojového kódu se přečte z adresy 1B752h

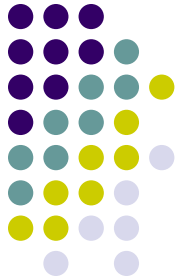
Registry



- 8086 obsahuje **14 registrů** se šířkou **16 bitů**
- 8086 je typický procesor se **střadačovou architekturou** – tzn. že registry nemají univerzální použití a každý lze použít jen pro určité dané operace
- nejprivilegovanější je **střadač - AX**

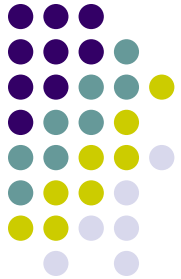
- Registry lze rozdělit na
  - **Segmentové** registry (CS, SS, DS, ES)
  - **Datové** registry (AX, BX, CX, DX)
  - **Ukazatele** a indexové registry (IP, SP, BP, DI, SI)
  - **Příznakový** registr (FLAGS)

Registr IP



- Programový čítač
- **Instruction Pointer** (ukazuje na místo v paměti, ze kterého se právě čte strojový kód instrukce ke zpracování)
- Obsah IP je offsetem ukazujícím do kódového segmentu
- CS:IP je tedy adresa zpracovávané instrukce
- Je to ale ještě trochu složitější kvůli frontě v BIU
- Ve skutečnosti CS:IP ukazuje na následující instrukci, která se uloží do instrukční fronty v BIU

Příklad

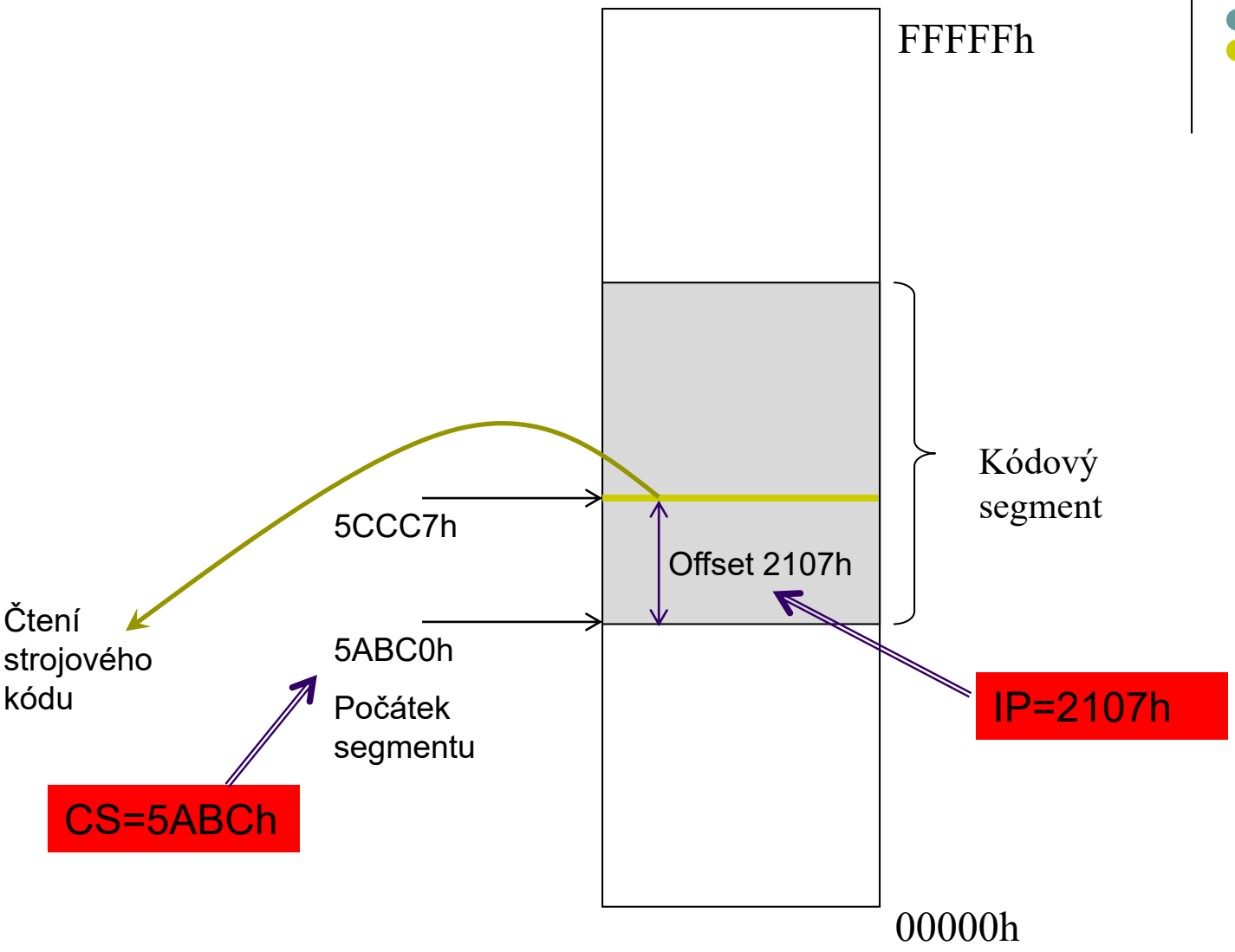


CS=5ABCh

IP=2107h

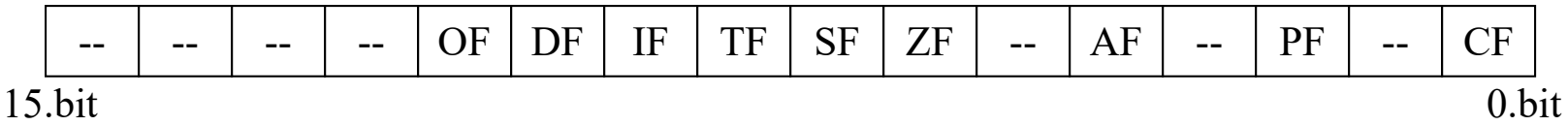
Z jaké adresy se bude číst strojový kód?

- Strojový kód programu je uložen v kódovém segmentu
- Kódový segment začíná na adrese 5ABC0h
- Z tohoto segmentu se bude číst strojový kód z pozice s offsetem 2107h
- Fyzická adresa tedy bude 5ABC0h+2107h = 5CCC7h

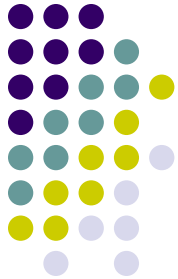

FFFFFh
Kódový segment
5CCC7h
Offset 2107h
Čtení strojového kódu
5ABC0h
Počátek segmentu
IP=2107h
CS=5ABCh
00000h

Příznakový registr FLAGS

- Šestnáctibitový příznakový registr je součástí jednotky EU
- 6 stavových příznaků
- 3 řídící příznaky
- 7 nevyužitých bitů

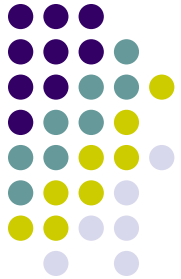


Příznakový registr - FLAGS



- **CF** (Carry Flag) Příznak přenosu. Nastaví se, pokud výsledek není osmi nebo šestnáctibitový (podle typu registru)
- **PF** (Parity Flag) se nastaví na jedničku, pokud dolní osmice bitů výsledku právě provedené operace obsahuje sudý počet "1“ (doplní počet jedniček na lichý)
- **AF** (Auxiliary Carry Flag) je rozšířením příznaku CF pro přenos přes hranici nejnižší půlslabiky operandu (vždy z bitu 3 do 4 bez ohledu na šířku operandu). Má význam v BCD aritmetice.
- **ZF** (Zero Flag) je nastaven při nulovém výsledku operace.
- **SF** (Sign Flag) je nastaven, když výsledek operace je záporný. Je shodný s nejvyšším (znaménkovým) bitem výsledku
- **OF** (Overflow Flag) se nastaví na jedničku, pokud při právě operace způsobí přenos mezi nejvyššími dvěma bity při počítání s čísly se znaménkem (nejvyšší bit je znaménko)
- **TF** (Trap Flag) uvádí procesor do krokovacího režimu, ve kterém je po provedení každé instrukce generováno přerušení (INT 1)
- **IF** (Interrupt Enable Flag) nulou lze zakázat všechna vnější maskovatelná přerušení (generovaná signálem INTR). Nemá vliv na vnitřní přerušení a nemaskovatelné přerušení)
- **DF**(Direction Flag) řídí směr zpracovávání řetězcových operací (s blokem dat lze pracovat od nejnižší adresy k nejvyšší nebo v opačném směru)

Carry flag



- Tento příznakový bit se nastaví, pokud výsledek není osmibitový nebo šestnáctibitový (podle toho, jestli se provádí 8-bitový nebo 16-bitový výpočet)
- Tento bit vlastně signalizuje, že poslední aritmetická operace nám dala neplatný výsledek – výsledek, který nelze zakódovat pomocí 8 nebo 16 bitů
- Mikroprocesor i8086 je 16-bitový
- Aritmetické výpočty mohou být provedeny **8-bitově** nebo **16-bitově**, podle toho, jak určí programátor

Carry Flag (CF)

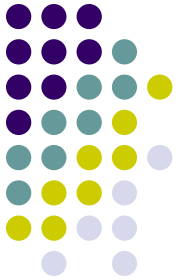
8–bitová aritmetika

- 10+20=30 CF←0
- 255+1 = 0 CF←1
- 255+2 = 1 CF←1
- 200+100 = 44 CF←1
- 10-5 = 5 CF←0
- 0-1 = 255 CF←1
- 5-10 = 251 CF←1
- 100-200 = 156 CF←1

16–bitová aritmetika

- 255+1 = 256 CF←0
- 200+100 = 300 CF←0
- 2000+1000 = 3000 CF←0
- 65535+1 = 0 CF←1
- 30000+40000=4446 CF←1
- 20000-5000=15000 CF←0
- 0-1 = 65535 CF←1
- 5-10 = 65531 CF←1
- 100-200 = 65435 CF←1

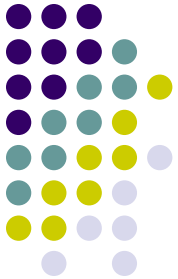
Zero Flag (ZF)



8-bitová aritmetika

- 5-5 = 0 ZF←1 CF←0
- 3+4 = 7 ZF←0 CF←0
- 255+1 = 0 ZF←1 CF←1
- 5-7 = 254 ZF←0 CF←1

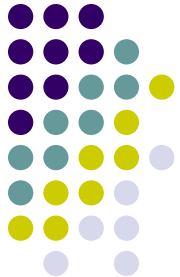
Parity Flag (PF)



- 5+4 = 9 (00001001 b) PF←1
- 250+5 = 255 (11111111 b) PF←1
- 80+20 = 100 (01100100 b) PF←0
- 255+1 = 0 (00000000 b) PF←1
- 10-5 = 5 (00000101 b) PF←1
- 255-3 = 252 (11111100 b) PF←1
- 100+71 = 171 (10101011 b) PF←0

- 128+128 = 256 (0000000100000000 b) PF←1
  (parita jen pro spodních osm bitů výsledku)

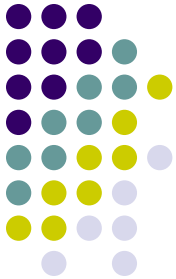
Auxiliary Flag (AF)



- Má význam pouze při počítání v **BCD kódu**
- **BCD** = Binary coded decimal
- V BCD kódu je každá cifra dekadického čísla zakódována pomocí 4 bitů
- Pomocí 8-bitů (bajtu) se tak zakóduje dvouciferné dekadické číslo
- 4 bity se použijí pro zakódování desítek
- 4 bity se použijí pro zakódování jednotek
- Mikroprocesor i8086 umí počítat i s takto zakódovanými čísly (ale není to běžné)

Číslo	BCD kód	binární kód
17	00010111	00010001
65	01100101	01000001
99	10011001	01100011
100	nelze	01100100
3	00000011	00000011

Auxiliary Flag (AF)



- 18 + 23 = 41
- V BCD kódu by výpočet vypadal takto:

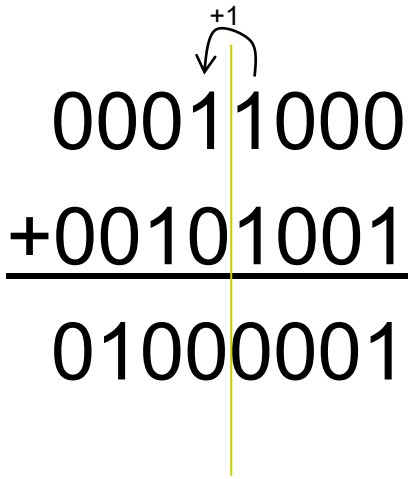
```
  00011000
+ 00100011
  --------
  00111011
```

Výsledkem je neplatný BCD kód
Tato kombinace 8 bitů nemá v BCD kódu žádný smysluplný význam

Bude potřeba provést opravu výsledku
Chybný výsledek v tomto případě rozpoznáme, protože odporuje pravidlům BCD

Auxiliary Flag (AF)

- 18 + 29 = 47
- V BCD kódu by výpočet vypadal takto:



AF←1

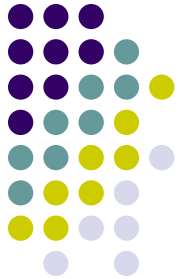
Výsledkem je platný BCD kód
Tato kombinace má v BCD význam 41
To je ale chybný výsledek !!!
Správně by mělo vyjít 47

Bude potřeba provést opravu výsledku
Chybný výsledek v tomto případě nepoznáme, protože vypadá dobře (jde o platné BCD číslo)

Výsledek je chybný, protože došlo k přenosu meze 3. a 4. bitem (ze spodní půlky bajtu do horní)

Přenos mezi 3. a 4. bitem signalizuje příznak AF

Datové registry

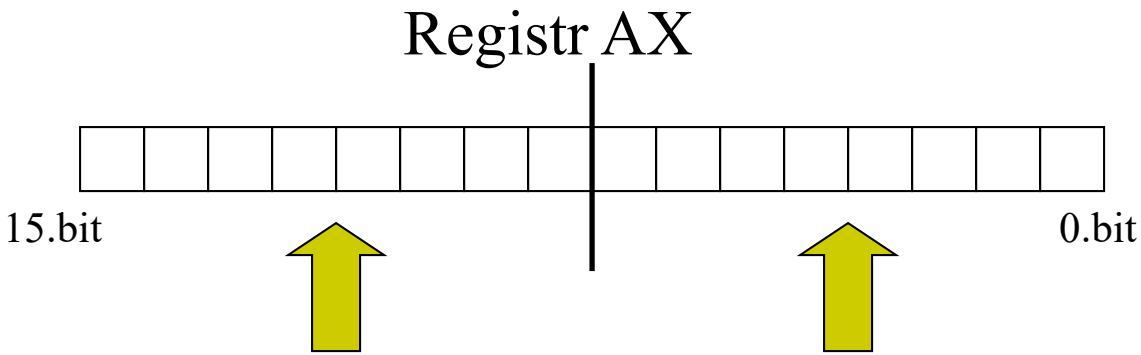


- Datové registry **AX**, **BX**, **CX**, **DX** jsou registry pro běžné použití v aritmetických a logických operacích
- Každý z nich může být použit jako 16-bitový registr nebo jako dva nezávisle 8-bitové registry
- Vyšší bajt v těchto registrech je označen **AH**, **BH**, **CH**, **DH**
- Nižší bajt v těchto registrech je označen **AL**, **BL**, **CL**, **DL**
- Při operacích s osmibitovými registry se mění i obsah šestnáctibitového registru jako celku

```
MOV BH,25h
MOV BL,8Ah          BX = 258Ah

MOV CX,5            CH=0 , CL=5
```

16-bitový registr AX lze rozdělit na dva osmibitové registry AH a AL



Osmibitový registr AH

Totožný s horní osmicí bitů AX

Osmibitový registr AL

Totožný se spodní osmicí bitů AX

Příklad - 8 a 16 bitů

- **Příklad**
  AH =10101111 b
  AL = 11001100 b
  jaký je tedy stav registru AX ?

  - V registru AH je horních 8 bitů (8. až 15. bit) registru AX
  - V registru AL je spodních 8 bitů (0. až 7. bit) registru AX

  - Bity můžeme zapsat v příslušném pořadí dohromady a získáme obsah registru AX

  - AX = 1010111111001100 b

Příklad - 8 a 16 bitů

- **Příklad**
  AH = 5A h
  AL = 78 h
  jaký je tedy stav registru AX ?

  - V registru AH je horních 8 bitů (8. až 15. bit) registru AX
  - V registru AL je spodních 8 bitů (0. až 7. bit) registru AX
  - AH = 01011010 b
  - AL = 01111000b

  - Bity můžeme zapsat v příslušném pořadí dohromady a získáme obsah registru AX

  - AX = 0101101001111000 b
  - To lze snadno převést do hexadecimálního zápisu AX=5A78h

Příklad - 8 a 16 bitů

Obsah registru AX vypadá takto:

V desítkové soustavě

AH = 16
AL = 2
AX = 4098

Pokud situaci v registru AX zapíšeme v desítkové soustavě, vypadá vše záhadně….

V šestnáctkové soustavě

AH = 10 h
AL = 02 h
AX = 1002 h

Pokud situaci v registru AX zapíšeme hexadecimálně, je vše srozumitelné

Příklad - 8 a 16 bitů

- **Příklad**

  DH =12

  DL = 34

  jaký je tedy stav registru DX ?

  - V žádném případě neplatí ~~DX=1234h~~
  - Do registru DH byl vlastně vložen bajt s hodnotou 0Ch
  - Do registru DL byl vložen bajt s hodnotou 22h
  - V registru DX je tedy šestnáctibitové číslo 0C22h

  - Při zápisu hodnot v desítkové soustavě nelze cifry obou hodnot spojit do čtyřciferného čísla. Toto by fungovalo pouze v hexadecimální soustavě, díky tom, že tam jedna cifra odpovídá čtyřem bitům.

Příklad - 8 a 16 bitů

- Pozor
  DH =12**h**
  DL = 34**h**
  jaký je teď stav registru DX ?

  - Teď už je situace poněkud jiná
  - Do registru DH byl vložen bajt s hodnotou 12h
  - Do registru DL byl vložen bajt s hodnotou 34h
  - V registru DX je tedy šestnáctibitové číslo 1234h

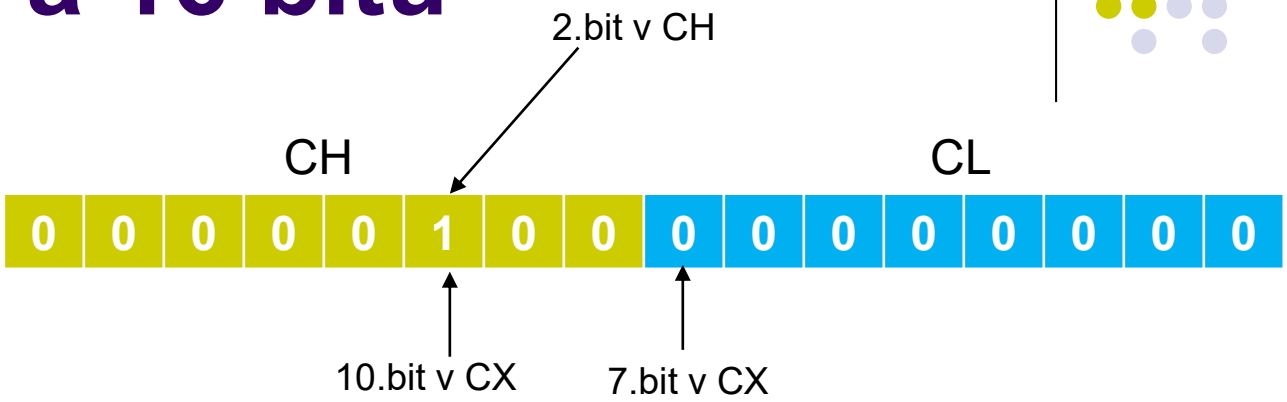
Příklad - 8 a 16 bitů

BX =1000

jaká je teď hodnota registrů BH a BL ?

- V žádném případě neplatí BH=10, BL=0
- Do BX bylo vloženo dekadické číslo 1000
- Toto číslo sem bylo vloženo jako šestnáct bitů 0000001111101000
- Hexadecimálně bychom to zapsali jako 3E8h
- Platí tedy, že BH=3 a BL=E8h

Příklad - 8 a 16 bitů



V registru CH je nastavený 2.bit

Všechny ostatní bity v registru CX jsou nulové

Určete hodnotu CH a CX v desítkové soustavě

- Stav registru CH vypadá takto CH=00000100 b
- V registru CH je nastaven 2.bit, tedy bit s vahou 4
- CH=4

- 2.bit v registru CH odpovídá 10. bitu v registru CX
- V registru CX je nastaven 10.bit, tedy bit s vahou 1024
- CX=1024

Příklad - 8 a 16 bitů

CH=1

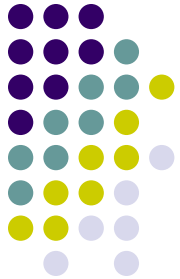
CL=1

Určete hodnotu CX v desítkové soustavě

- V žádném případě neplatí CX=11 !
- V CX je vloženo dekadické číslo 257

- Nastaven je 0. bit a 8.bit registru CX
- Nastaveny jsou tedy bity s vahou 1 + 256

8 bitů a 16 bitů v akci



- INC = **inkrementace**. Tato instrukce zvýší hodnotu registru o +1

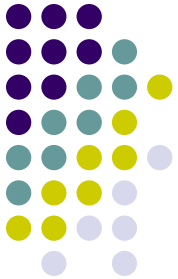
  Příklad

  AL=7

  INC AL

- Po provedení instrukce bude AL=8

8 bitů a 16 bitů v akci



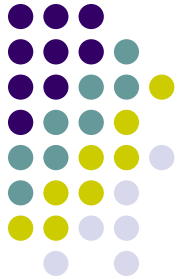
Příklad

AL=255

INC AL

- Výpočet se provádí s 8-bitovým registrem AL a bude tedy proveden 8-bitově
- Po provedení instrukce INC bude AL=0
- 255+1=0
- Došlo k přetečení
- Přetečení bude signalizování bitem CF
- CF=1

8 bitů a 16 bitů v akci



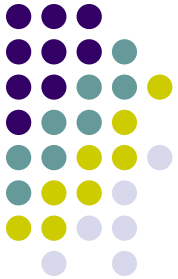
Příklad

AX=255

INC AX

- Výpočet se provádí s 16-bitovým registrem AX a bude tedy proveden 16-bitově
- Po provedení instrukce INC bude AX=256
- 255+1=256
- Nedošlo k přetečení
- CF=0

8 bitů a 16 bitů v akci



- DEC = **dekrementace**. Tato instrukce o jedničku sníží hodnotu registru

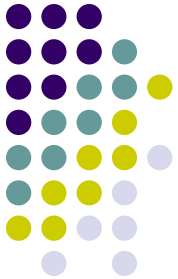
  Příklad

  AL=7

  DEC AL

- Po provedení instrukce bude AL=6

8 bitů a 16 bitů v akci



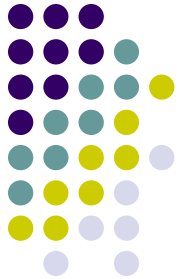
Příklad

AL=0

DEC AL

- Výpočet se provádí s 8-bitovým registrem AL a bude tedy proveden 8-bitově
- Po provedení instrukce DEC bude AL=255
- 0-1 = 255
- Došlo k přetečení
- CF=1

8 bitů a 16 bitů v akci



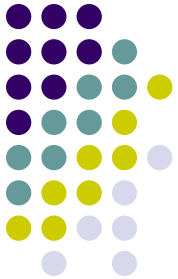
Příklad

AX=0

DEC AX

- Výpočet se provádí s 16-bitovým registrem AX a bude tedy proveden 16-bitově
- Po provedení instrukce DEC bude AX=65535
- 0-1 = 65535
- Došlo k přetečení
- CF=1

8 a 16 bitů v akci



- Přetečení spodního osmibitového registru nemá vliv na horní bajt

  MOV AL,255
  MOV AH,0 v tuto chvíli AX=00FFh
  INC AL AL přeteče, ale hodnota AH se tím nemění

  AX=0000h AL=0, AH=0, je nastaven bit CF

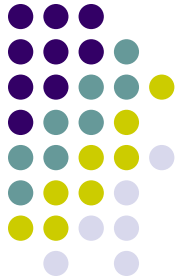
Přetečení registru AL neovlivňuje registr AH. Oba registry se chovají nezávisle při použití osmibitových operací

- Jinak ale bude situace vypadat při použití šestnáctibitové operace

  MOV AL,255
  MOV AH,0 v tuto chvíli AX=00FFh
  INC AX inkrementuje se celý AX

  AX=0100h AL=0, AH=1, není nastaven bit CF

Instrukce MOV



- MOV = MOVE
- Nejpoužívanější instrukce v instrukční sadě
- Slouží k vložení hodnot do registrů, přesunu hodnot mezi registry, čtení z paměti a zápisu do paměti
- Až 40% programu jsou instrukce MOV
- Při analýze programu ve strojovém kódu zjistíte, že se pořád něco někam přesouvá instrukcí MOV
- Abychom lépe pochopili fungování mikroprocesoru, musíme se částečně s touto instrukcí seznámit

Instrukce MOV

Syntaxe: **MOV cíl, zdroj**

Zápis konstanty

MOV AX,0

MOV AX,65535

Při použití 16-bitových registrů lze použít hodnoty z intervalu <0;65535>

MOV BL,300 - nelze

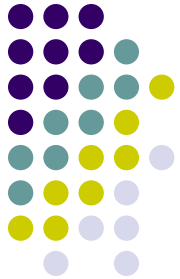
Osmibitové registry lze naplnit pouze hodnotami z intervalu <0;255>

MOV AH,4Fh

MOV DX,8AFBh

*Za hexadecimální hodnotou se uvádí **h***

Instrukce MOV



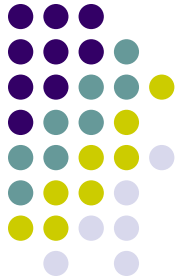
Přesuny mezi registry

```
MOV     CX,AX
MOV     CX,AL - nelze
MOV     CH,AL
MOV     AH,AL
```

Možné jsou všechny kombinace. Šířka zdrojového a cílového operandu musí být shodná - nelze provádět přesuny mezi 8b a 16b registry

Hodnota se „nepřesouvá“ ale „kopíruje se“

Přímá adresace

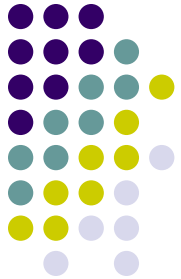


- MOV AH,[100]
- adresa se uvádí v hranatých závorkách a má význam offsetu

- MOV [5],AL
- V tomto případě se uloží obsah registru AL na pátou pozici v datovém segmentu

- Instrukce MOV bere jako implicitní datový segment a jeho bázovou adresu v DS
- Chceme-li použít pomocný datový segment, je třeba to zapsat
  MOV AX,ES:[12]
  MOV ES:[52],CL

Příklad



DS=1234h
ES=2345h

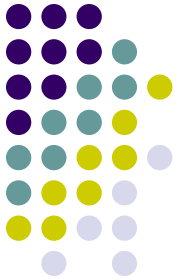
Na jakou adresu v paměti se zapíší data povelem MOV [7],AL ?

- Data se implicitně zapisují do datového segmentu.
- Datový segment začíná na adrese 12340h
- Bajt z registru AL se zapíše do datového segmentu na pozici s offsetem 7
- Fyzická adresa tedy bude 12340h+7 = 12347h

Na jakou adresu v paměti se zapíší data povelem MOV ES:[7],AL ?

- Touto instrukcí se budou data zapisovat do pomocného datového segmentu
- Pomocný datový segment začíná na adrese 23450h
- Bajt z registru se zapíše do pomocného datového segmentu na pozici s offsetem 7
- Fyzická adresa tedy bude 23450h+7 = 23457h

Indexové registry



- Indexové registry SI a DI

- **SI**=Source Index
- **DI**=Destination Index

- Fungují jako offsety - „ukazovátka“ na dvě různá místa v datovém segmentu
- SI bývá použit jako offset zdrojových dat
- DI bývá použit jako offset cíle
- Jsou využívány operacemi, které se týkají bloku dat
- Oba registry se běžně používají pro nepřímou adresaci ve spojení prakticky se všemi instrukcemi

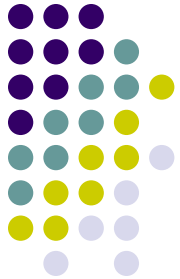
Příklad

- DS=4567h
- DI=5678h

Na jakou adresu v paměti se zapíší data povelem MOV [DI],AL ?

- Data se zapíší do datového segmentu.
- Datový segment začíná na adrese 45670h
- Bajt z registru AL se zapíše do datového segmentu na pozici s offsetem, jehož hodnota není přímo uvedena
- Jedná se o **nepřímou adresaci**
- K nepřímé adresaci je použit registr DI
- Hodnota, kterou obsahuje registr DI se vezme jako offset pro zápis do paměti
- Data se tedy zapíší na pozici s offsetem 5678h
- Fyzická adresa tedy bude 45670h+5678h = 4ACE8h

Příklad



- DS=6543h
- SI=7812h

Z jaké adresy z paměti se přečtou data povelem MOV AL,[SI] ?

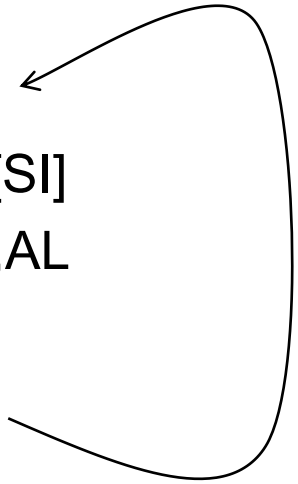
- Data se přečtou z datového segmentu.
- Datový segment začíná na adrese 65430h
- Bajt z registru AL se přečte z datového segmentu z pozice s offsetem, jehož hodnota není přímo uvedena
- Jedná se o **nepřímou adresaci**
- K nepřímé adresaci je použit registr SI
- Hodnota, kterou obsahuje registr SI se vezme jako offset pro čtení z paměti
- Data se tedy přečtou pozice s offsetem 7812h
- Fyzická adresa tedy bude 65430h+7812h = 6CC42h

Příklad

- DS=6543h
- SI=7812h
- DI=89ABh

Opakuj 100x

```
MOV   AL,[SI]
MOV   [DI],AL
INC   SI
INC   DI
```



Tento program bude kopírovat bajty ze zdrojové oblasti do cílové oblasti.

První bajt se přečte z adresy 65430h + 7812h = 6CC42h

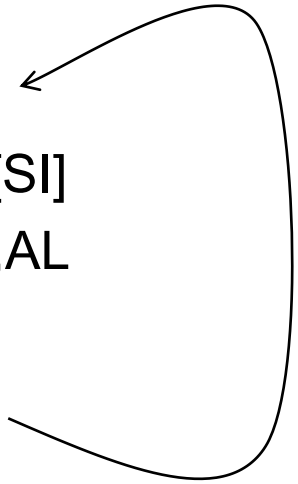
A zapíše se na adresu 65430h + 89ABh = 6DDDBh

Příklad

- DS=6543h
- SI=7812h
- DI=89ABh

Opakuj 100x

```
MOV   AL,[SI]
MOV   [DI],AL
INC   SI
INC   DI
```



Hodnota SI se inkrementací zvýší na 7813h

Hodnota DI se inkrementací zvýší na 89ACh

Druhý bajt se přečte z adresy 65430h + 7813h = 6CC43h

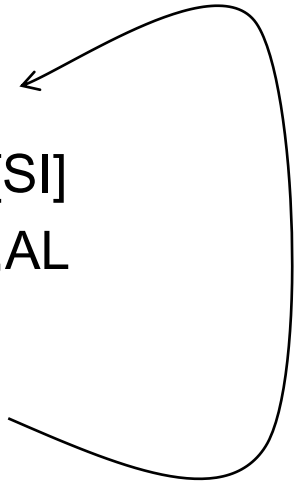
A zapíše se na adresu 65430h + 89ACh = 6DDDCh

Příklad

- DS=6543h
- SI=7812h
- DI=89ABh

Opakuj 100x

```
MOV   AL,[SI]
MOV   [DI],AL
INC   SI
INC   DI
```



Hodnota SI se inkrementací zvýší na 7814h

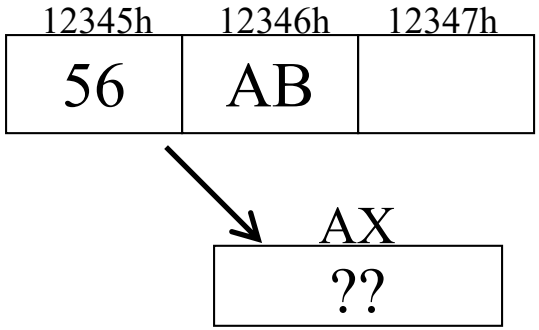
Hodnota DI se inkrementací zvýší na 89ADh

Druhý bajt se přečte z adresy 65430h + 7814h = 6CC44h

A zapíše se na adresu 65430h + 89ADh = 6DDDDh

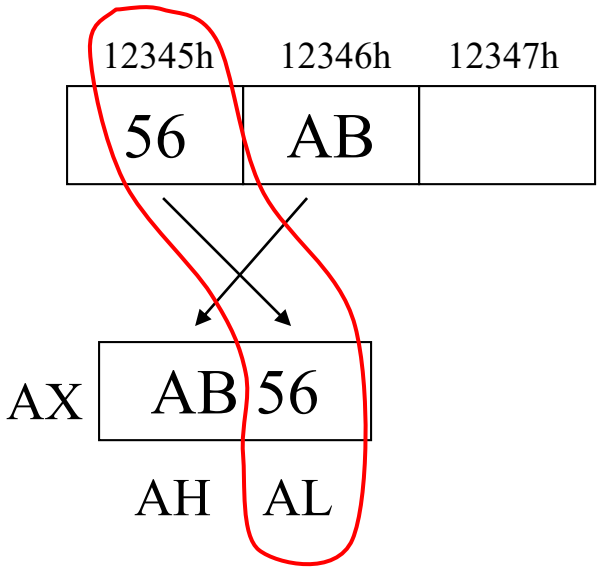
Uložení operandů v paměti

- Představme si, že v paměti na adresách 12345h a 12346h leží tyto bajty



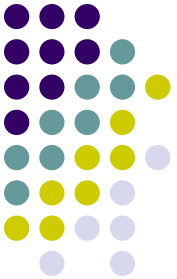
- jaká bude hodnota registru AX po přečtení dat z adresy 12345h ?
- Registr **AX je šestnáctibitový**, takže se naplní šestnáctibitovou (dvoubajtovou) hodnotou
- Registr AX tedy bude naplněn **dvěma bajty** z **dvou adres** 12345h a 12346h
- Bude v registru AX hodnota AB56h nebo 56AB h ?
- To záleží na tom, jakým způsobem procesor pracuje s 16-bitovými operandy v paměti
- Všechny mikroprocesory řady x86 používají uložení nižšího bajtu na nižší adrese a vyšších osm bitů leží na následující vyšší adrese
- Hodnota registru AX tedy bude AB56h (hodnota 56h ležící na adrese 12345h představuje spodních osm bitů 16-bitové hodnoty)

Uložení operandů v paměti



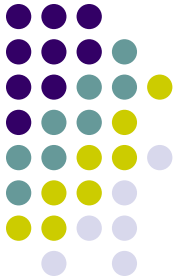
Šestnáctibitová čísla jsou do paměti ukládána vždy tím způsobem, že spodních osm bitů leží na nižší adrese

Uložení operandů v paměti



- **Endianita** (neboli byte order) – jak uložit do paměti (nebo do souboru) dlouhé číslo složené z několika bajtů?
- **Byte order** je jedním ze základních zdrojů nekompatibility při ukládání a výměně dat v digitální podobě
- Byte order (pořadí, v jakém se do paměti uloží jednotlivé bajty dlouhého čísla)
  - **Big-endian** - na paměťové místo s nejnižší adresou uloží nejvíce významný bajt (bajt, který obsahuje nejvyšší bit čísla) a nad něj se ukládají ostatní bajty až po nejméně významný bajt (bajt, ve kterém je nejnižší bit čísla) na konci. Tento způsob je běžný na mikroprocesorech Motorola
  - **Little-endian** - Na paměťové místo s nejnižší adresou uloží nejméně významný bajt (bajt, ve kterém leží nejnižší bit čísla) a nad něj se ukládají ostatní bajty až po nejvíce významný bajt (bajt, ve kterém leží nejvyšší bit čísla)
- Nejpoužívanějším kódováním vícebajtových dat je v současnosti **little-endian**, což je dané rozšířením architektury Intel x86

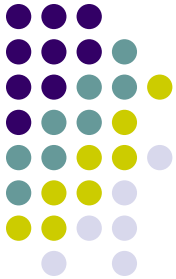
Byte order (endianita)



- **Big-endian**
- 32-bitové číslo 12345678h uložené na adrese 100

Adresa	Bajt
…..	
103	78h
102	56h
101	34h
100	12h
…..	

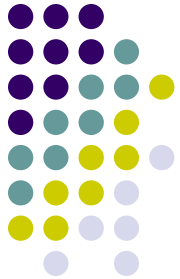
Byte order (endianita)



- **Little-endian**
- 32-bitové číslo 12345678h uložené na adrese 100

Adresa	Bajt
…..	
103	12h
102	34h
101	56h
100	78h
…..	

Uložení operandů v paměti



Příklad

AX=1234h DS=2321h CS=7852h DI=05D2h

MOV [DI],AX

- Na jaké fyzické adresy v paměti byl proveden zápis dat instrukcí MOV a jaké konkrétní bajty a kam přesně byly do paměti uloženy ?

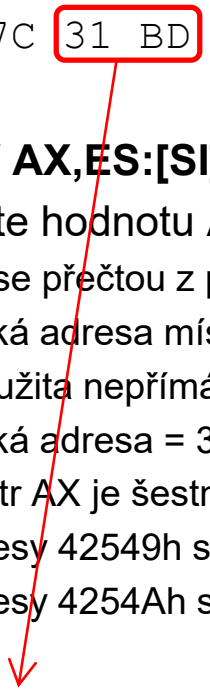
- Data se uloží do datového segmentu, jehož počáteční adresa je 23210h
- Fyzická adresa cílového místa se vypočítá přičtením offsetu k bázové adrese datového segmentu
- Je použita nepřímá adresace a offset je uložen v registru DI – 05D2h
- Fyzická adresa = 23210h+ 5D2h = 237E2h
- Registr AX je šestnáctibitový a jeho obsah se tedy po uložení na adresu 237E2h objeví i na adrese 237E3h
- Víme již, že spodní bajt (tj. vlastně AL) se ukládá na nižší adresu a horních osm bitů (AH) na vyšší adresu
- Do paměti tedy byl zapsán
  - Bajt 12h na adresu 237E3h
  - Bajt 34h na adresu 237E2h

Uložení operandů v paměti

Příklad

DS=5312h ES=3B92h CS=7852h SI=6C29h

```
42540 25 AC 00 F3 B9 7E 66 97
42548 7C 31 BD 42 80 5E 00 12
```



MOV AX,ES:[SI]

- Určete hodnotu AX po provedení tohoto čtení dat z paměti
- Data se přečtou z pomocného datového segmentu, jehož počáteční adresa je 3B920h
- Fyzická adresa místa se vypočítá přičtením offsetu k bázové adrese pom. dat. segmentu
- Je použita nepřímá adresace a offset je uložen v registru SI (6C29h)
- Fyzická adresa = 3B920h+ 6C29h = 42549h
- Registr AX je šestnáctibitový a z paměti je třeba přečíst 16 bitů – 2 bajty
- Z adresy 42549h se přečte bajt 31h, tento bajt se vloží do AL
- Z adresy 4254Ah se přečte bajt BDh, tento bajt se vloží do AH

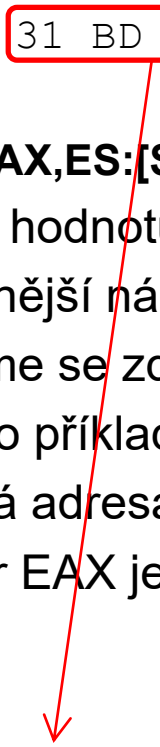
- AX=BD31h

Uložení operandů v paměti

Příklad

DS=5312h ES=3B92h CS=7852h SI=6C29h

```
42540 25 AC 00 F3 B9 7E 66 97
42548 7C 31 BD 42 80 5E 00 12
```

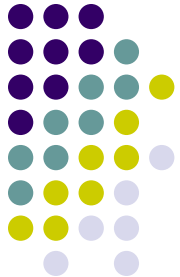


MOV EAX,ES:[SI]

- Určete hodnotu **EAX** po provedení tohoto čtení dat z paměti
- Modernější nástupci i8086 budou 32-bitové a 64-bitové mikroprocesory
- Setkáme se zde s rozšířeným **32-bitovým** registrem EAX
- V tomto příkladu se data čtou ze stejné adresy jako v předchozím
- Fyzická adresa = 3B920h+ 6C29h = 42549h
- Registr EAX je 32-bitový a z paměti je třeba přečíst 32 bitů – 4 bajty

- EAX=8042BD31h

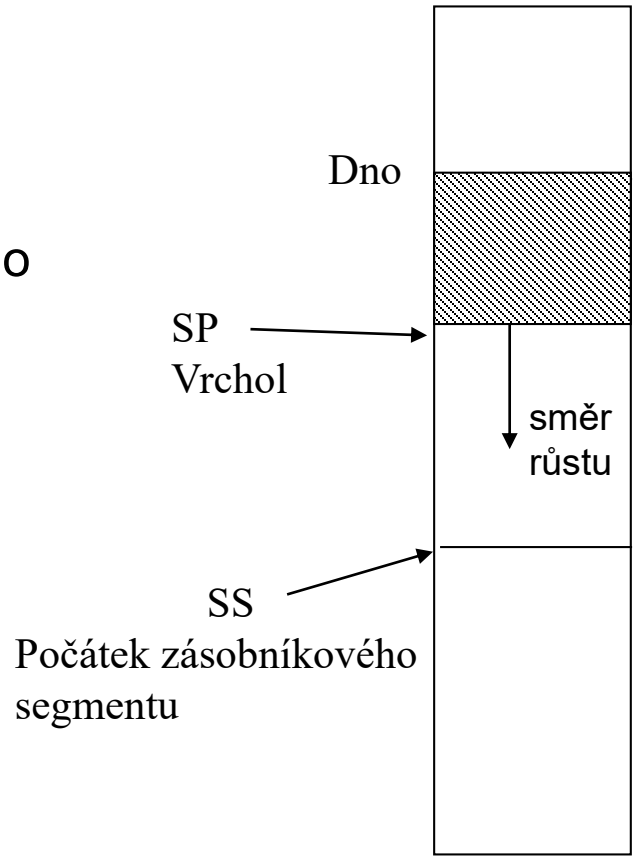
Zásobník



- Používá se k uložení **návratové adresy** při volání podprogramu nebo při obsluze přerušení, ale lze ho použít i k odkládání běžných dat metodou **LIFO**
- **LIFO – Last IN First OUT**
- Běžně ho tedy využívají i programy pro odložení dat k pozdějšímu vyzvednutí v opačném pořadí nebo k předání parametrů volanému podprogramu
- Zásobník si lze představit buď jako kostky naskládané po jedné na sebe do vysokého sloupce nebo jako jámu, do které se na sebe kupí data
- V obou případech platí, že to, co bylo do zásobníku vloženo naposled musí být jako první odebráno. Na data odložená „na dně“ se dostaneme až po odebrání všech dat, která byla vložena po tom
- Tento systém ukládání dat je právě vhodný pro uložení návratových adres při volání podprogramu.
- Z jednoho podprogramu je pak možné volat další podprogram a z něj opět další… návratové adresy se při tom v zásobníku „kupí na sebe“ a v opačném pořadí jsou použity k návratům na místa volání podprogramu.

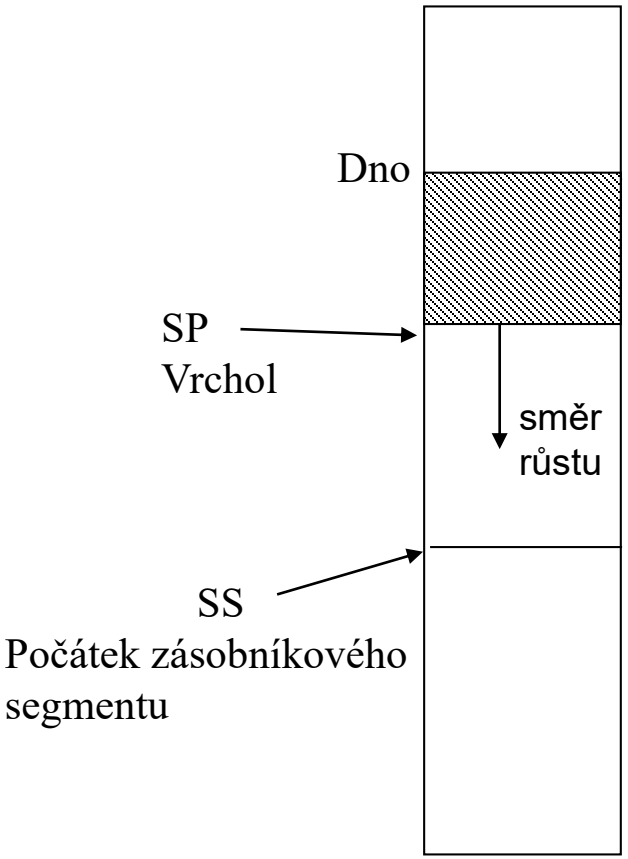
Zásobník

- Zásobník je vytvořen v **zásobníkovém segmentu** (a může tedy mít max, 64kB)
- 8086 buduje zásobník v opačném směru, než jak by se zdálo logické
- Dno zásobníku leží na **vyšší** adrese než jeho **vrchol**
- **Dno** tedy leží na **konci** zásobníkového segmentu
- Začátek zásobníkového segmentu je dán obsahem segmentového registru SS
- Na **vrchol** zásobníku ukazuje offset **SP**
- Pro přístup doprostřed zásobníku mohou programy použít i registr BP (tak mohou přečíst i data uložená uprostřed „hromady“)

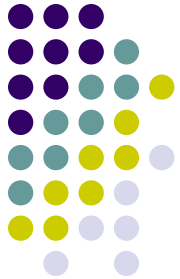


Zásobník

- První bajt se do zásobníku uloží na dno, tedy na konec zásobníkového segmentu
- Pokud je zásobník prázdný, ukazuje SP na konec zásobníkového segmentu (SP=FFFFh)
- Jak se postupně zásobník zaplňuje, hodnota SP klesá
- Pokud se do zásobníku **uloží** jeden bajt, vrchol se posune o jednu adresu směrem dolů a hodnota **SP** se **sníží o jedničku**
- Pokud se ze zásobníku **odebere** jeden bajt, vrchol se posune o jednu adresu směrem nahoru a hodnota **SP** se **zvýší o jedničku**



Zásobník



- Instrukcí **PUSH** lze uložit na vrchol zásobníku obsah registru
- Instrukcí **POP** lze odebrat bajt/bajty z vrcholu zásobníku a uložit je do vybraného registru

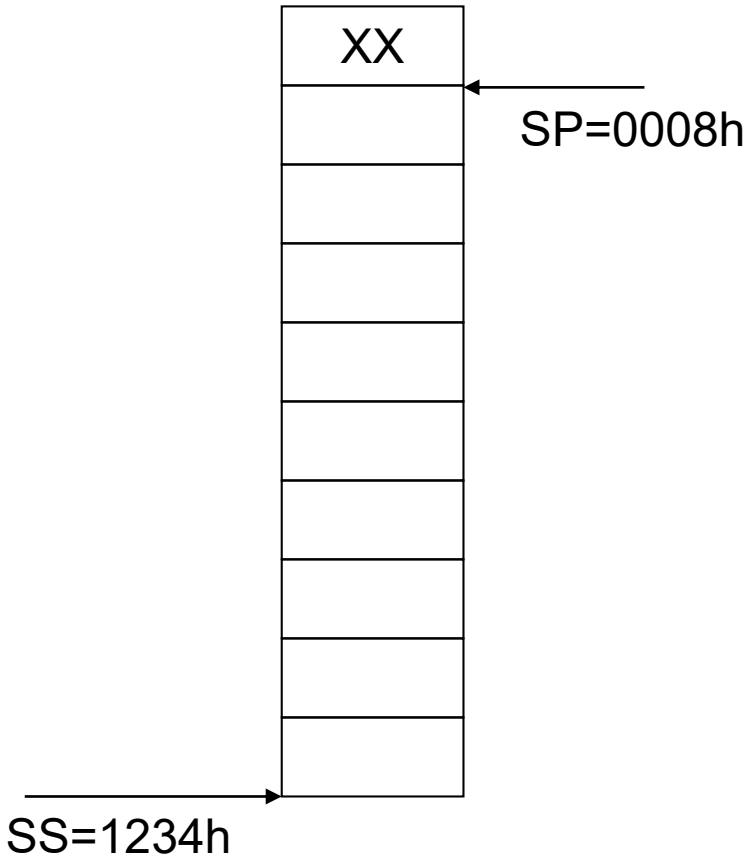
- Příklad:

```
MOV     AX,1234h
MOV     BX,9876h
PUSH    AX
PUSH    BX
POP     AX
POP     BX
```

  - AX=9876h, BX=1234h – hodnota v registrech AX a BX byla prohozena

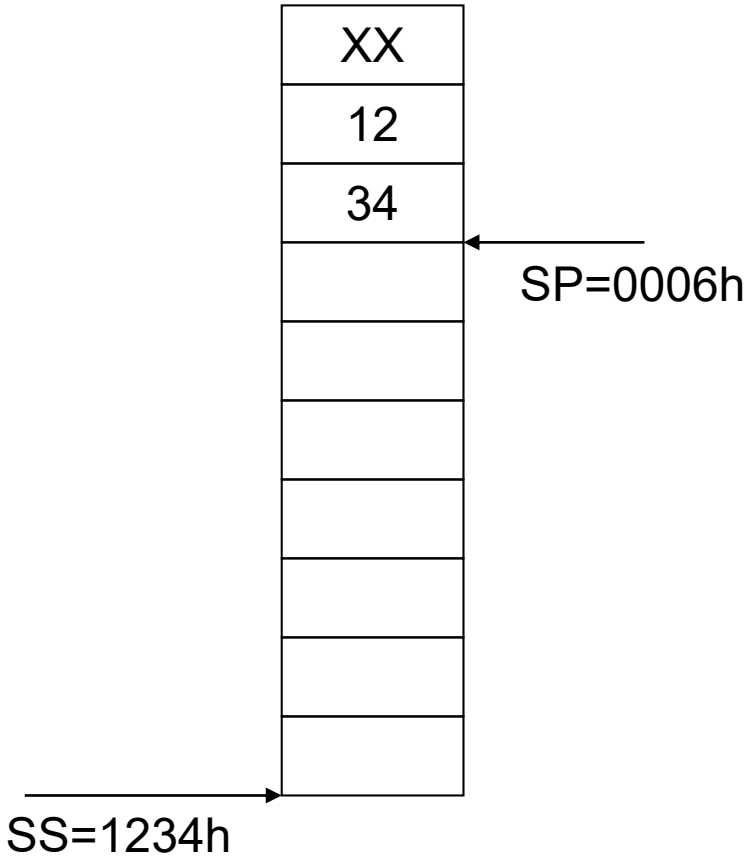
Zásobník

```
MOV  AX,1234h
MOV  BX,9876h
PUSH AX
```



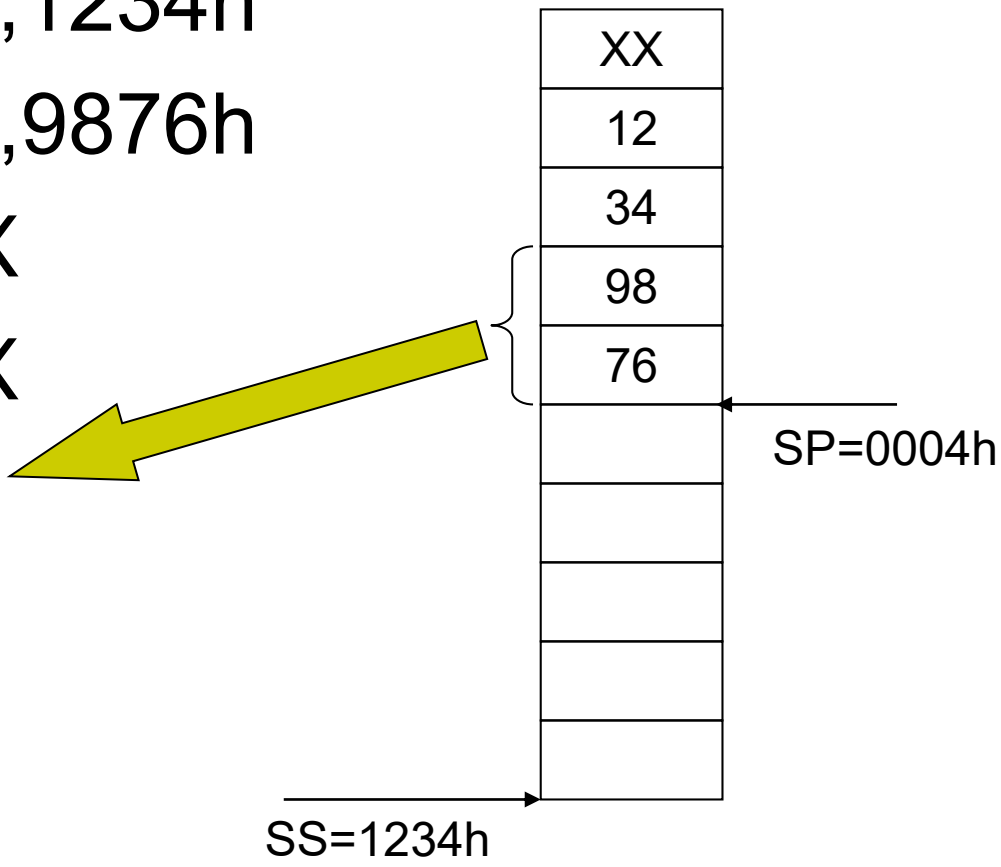
Zásobník

```
MOV  AX,1234h
MOV  BX,9876h
PUSH AX
PUSH BX
```



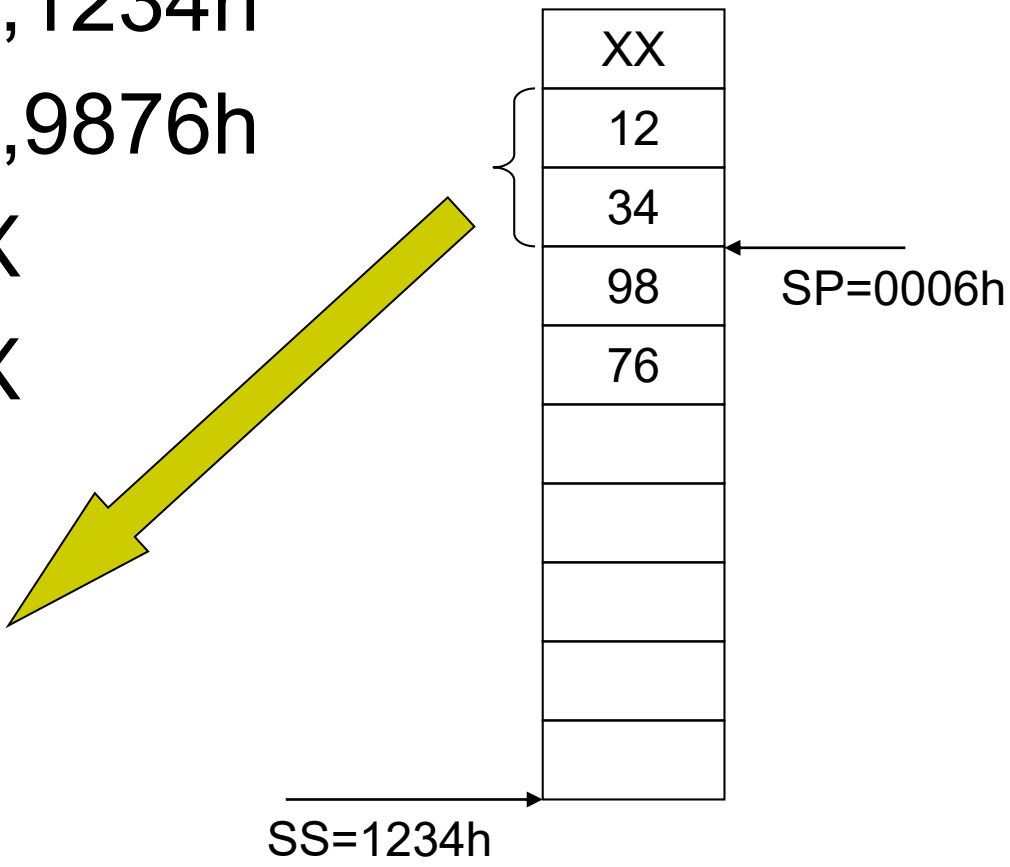
Zásobník

```
MOV  AX,1234h
MOV  BX,9876h
PUSH AX
PUSH BX
POP  AX
```



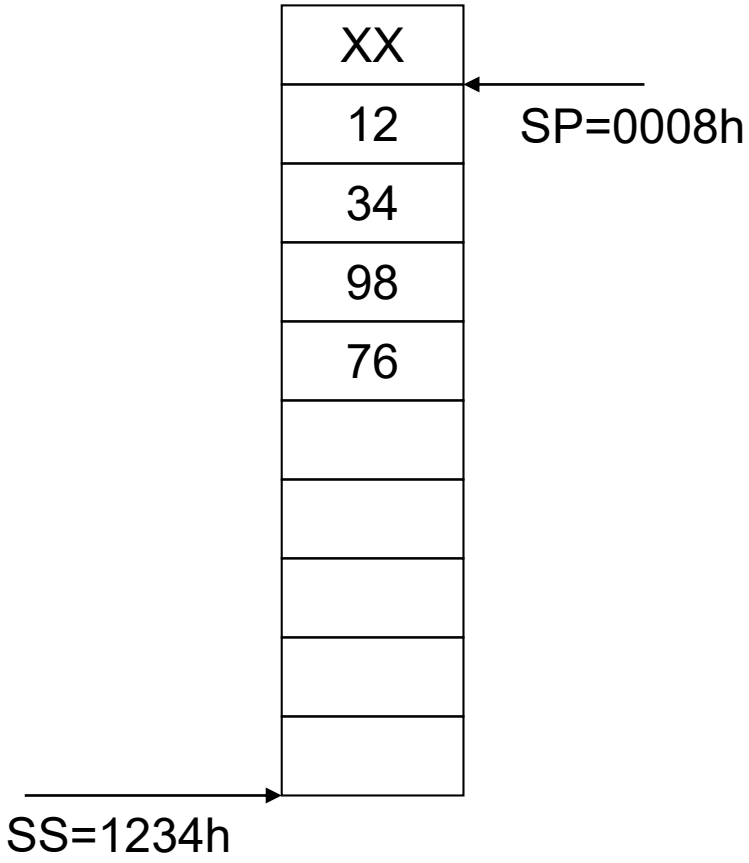
Zásobník

```
MOV  AX,1234h
MOV  BX,9876h
PUSH AX
PUSH BX
POP  AX
POP  BX
```

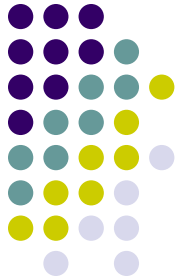


Zásobník

```
MOV  AX,1234h
MOV  BX,9876h
PUSH AX
PUSH BX
POP  AX
POP  BX
```



Zásobník



Příklad

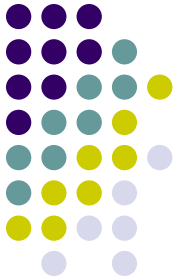
SS=1234h

SP=4345h

AX=5678h

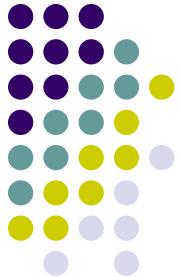
- Na jaké adrese začíná zásobníkový segment?
- Na jaké adrese končí zásobníkový segment?
- Na jaké adrese je vrchol zásobníku?
- Z jaké adresy by se přečetl bajt povelem POP AL?
- Na jaké adresy by se uložila data povelem PUSH AX?
- Jak by se změnil stav registru SS a SP po provedení instrukce PUSH AX?
- Jak by se změnil stav registru SS a SP po provedení instrukce POP AL?

Zásobník



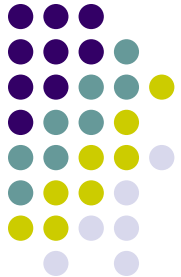
- Instrukcí **PUSHA** se na vrchol zásobníku uloží postupně obsah registrů AX, CX, DX, BX, SP, BP, SI, DI
- Instrukcí **POPA** se obnoví ze zásobníku postupně hodnota v registrech DI, SI, BP, SP, BX, DX, CX, AX
- **PUSHF** - do zásobníku uloží obsah registru F v šestnáctibitovém tvaru (všechny příznaky)
- **POPF** - hodnotou ze zásobníku změní registr F (stav všech příznaků)

Přerušení



- Nástroj pro asynchronní obsluhu událostí, kdy procesor přeruší vykonávání instrukcí a vykoná obsluhu události, která přerušení spustila
- Přerušení slouží především k obsluze **I/O zařízení**
- Díky přerušení nemusí procesor neustále testovat, jestli vstupní a výstupní zařízení něco nepotřebují
- Hardwarové přerušení nastává jako reakce na signál od zařízení, které jím upozorňuje procesor, že potřebuje obsloužit (něco se stalo, byla stisknuta klávesa, přišel síťový paket, uplynul nastavený čas…)
- Procesor při příchodu přerušení přestane provádět probíhající program, uloží stav rozpracované úlohy a zapamatuje si místo, kam se vrátit (k tomu se použije zásobník), a začne vykonávat obsluhu přerušení
- Kdyby procesor neumožňoval přerušení, musely by naše programy neustále periodicky testovat stav klávesnice, myši, hodin, síťového komunikačního rozhraní a dalších zařízení, na která se má reagovat

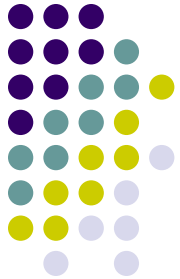
Přerušení



- Podle toho, čím je přerušení generováno jej dělíme na
  - Přerušení **vnější** - technickými hardwarovými prostředky
    - Nemaskovatelné (signál NMI) – vždy se musí obsloužit
    - Maskovatelné (signál INTR) – lze ho ignorovat (obsluhu lze zakázat)
  - Přerušení **programové** (vnitřní)
    - Instrukcí INT – úmyslně vyvolané programátorem
    - Vznikem výjimečné události při provádění programu

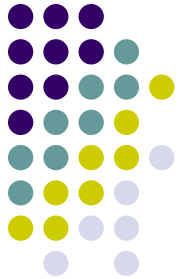
- Mechanismus přerušení u 8086 využívají
  - vnější **I/O zařízení** pro přivolání pozornosti (např. klávesnice oznamuje, že došlo k nějaké události, síťová karta hlásí příjem rámce atd.)
  - procesor pro oznámení **výjimečných událostí** (např. dělení nulou)
  - programy pro **volání služeb** OS a BIOSu (Operační systém nabízí určité služby, které se volají tak, že je vygenerováno přerušení, které je obsluhováno operačním systémem a dle stavu určitých registrů pro předání parametrů, operační systém vykoná požadovanou službu)

Přerušení



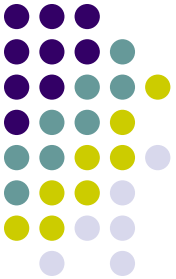
- Procesor rozlišuje **256 různých přerušení** (INT0 až INT255)
- I/O zařízení vyvolají vnější přerušení signálem **INTR** nebo **NMI**
- Přerušení vyvolaná signálem INTR lze **maskovat** (zakázat) nulováním příznakového bitu **IF**, což se provede instrukcí CLI (nastavení naopak instrukcí STI)
- Obsluha přerušení = krátký program, uložený někde v paměti, který se automaticky spustí jako reakce na přerušení, které nastalo – je tedy spuštěn nějakou událostí, na kterou reaguje
- Každé přerušení má svou vlastní obsluhu
- Pro každé přerušení je v paměti uložen **přerušovací vektor** (tj. odkaz na adresu, na které se nalézá jeho **obsluha přerušení**)
- Prvních **1024 bajtů** paměti (adresy 0 až 3FFh) je vyhrazeno pro uložení **256 vektorů přerušení**
- **Vektor přerušení** je **4-bajtový** (16 bitů segment, 16 bitů offset)
- Na adresách 0-3 tedy leží segment a offset pozice, na které začíná obslužná rutina přerušení INT0. Dále na adresách 4-7 leží segment a offset pozice, na které začíná obsluha přerušení INT1 atd...

Obsluha přerušení

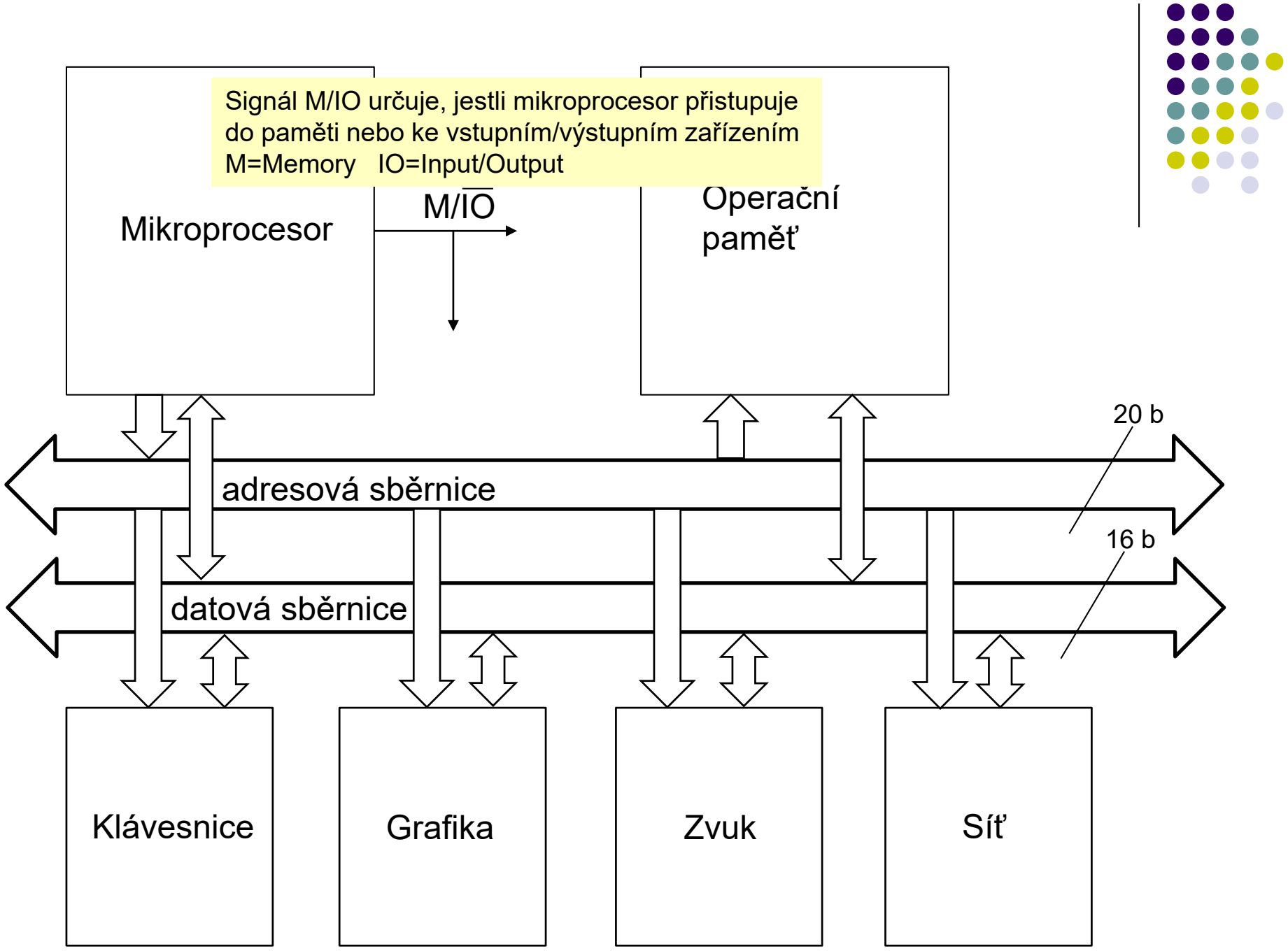


- Instrukce jsou **nedělitelné** - přerušení se obsluhuje až po dokončení rozpracované instrukce
- Obsluha přerušení obvykle na počátku uloží do **zásobníku** obsah všech registrů, jejichž hodnotu mění a před návratem z přerušení je opět obnoví
- **Návrat** z obsluhy přerušení se provádí instrukcí **IRET,** v zásobníku je uložena návratová adresa
- Některé přerušovací vektory jsou rezervovány pro přerušení generovaná procesorem
  - INT0 nastává při **dělení nulou**
  - INT1 nastává po provedení každé instrukce, je-li povoleno **krokování** nastavením příznaku TF
  - INT2 se vygeneruje při příchodu vnějšího **nemaskovatelného** přerušení (přijetí signálu NMI)
  - *Další přerušení jsou generována na modernějších procesorech při porušení pravidel chráněného režimu*

Ovládání I/O zařízení

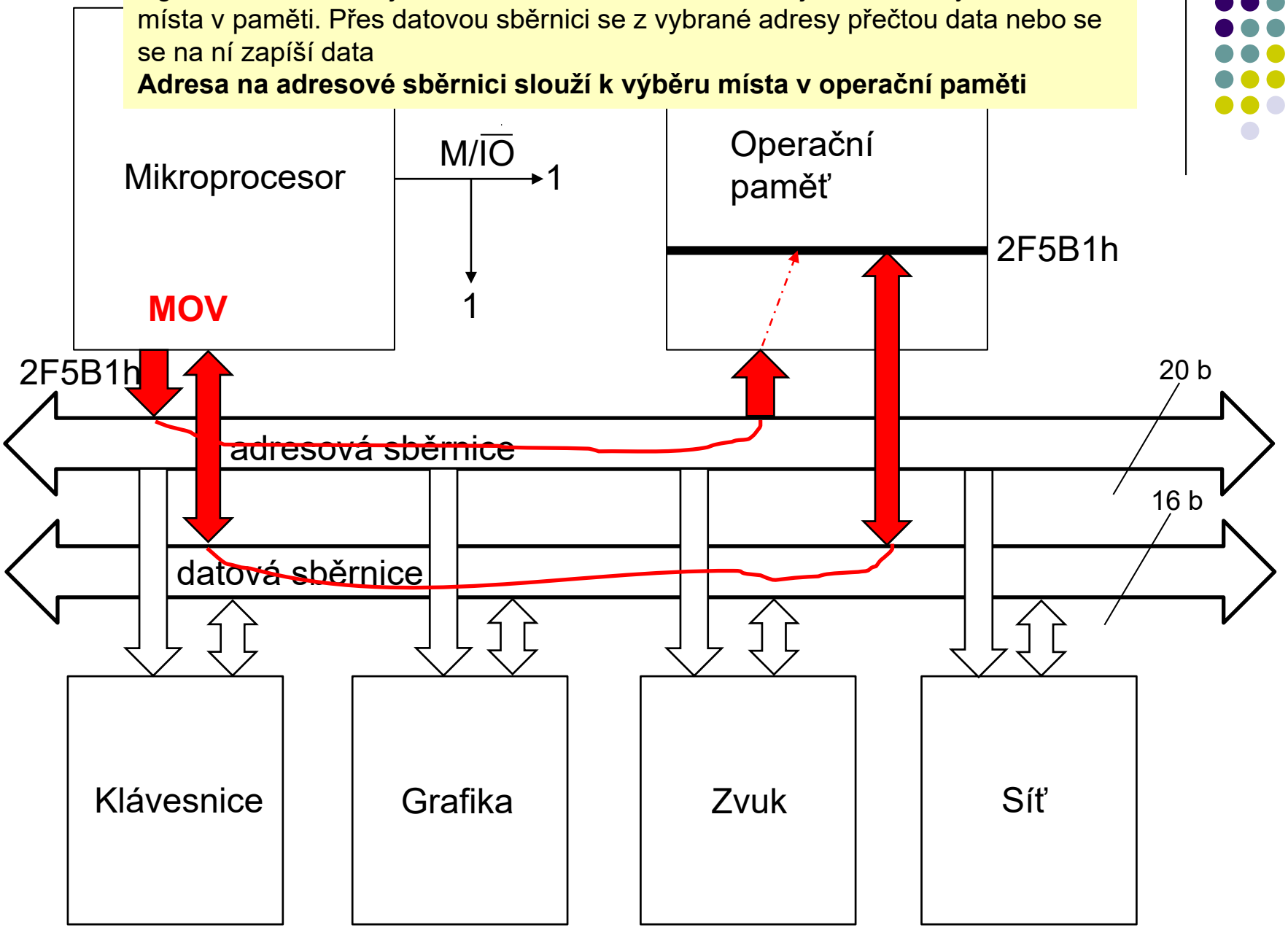


- Komunikace s okolím probíhá výhradně prostřednictvím **adresové a datové sběrnice**
- Komunikace s pamětí probíhá tak, že na adresové sběrnici je nastavena vybraná 20-bitová adresa a po datové sběrnici probíhá zápis nebo čtení 16-bitového slova (2 bajty naráz)
- Zápis nebo čtení z paměti se provádí instrukcí **MOV**
- Komunikace s **I/O zařízením** (tj. např. s pevným diskem, grafickým adaptérem, síťovou kartou…) probíhá velmi podobně – používají se ale jiné instrukce - **IN** a **OUT**
- Na adresové sběrnici je nastavena adresa - „číslo“ zařízení (neboli **brána**)
- Signál **M/IO** (Memory / InputOutput) určuje zda signál na adresové sběrnici má význam **adresy paměti** nebo identifikuje **I/O zařízení**
- Ovládání I/O zařízení probíhá zápisem nebo čtením datové sběrnice
- **Brána (port)** = pomyslná adresa (číslo) zařízení
- Každé I/O zařízení je identifikované jiným číslem portu
- i8086 poskytuje 65536 takových bran (portů)
- Data se na specifikovanou bránu zapisují instrukci **OUT** a čtou se instrukcí **IN**


Signál M/IO určuje, jestli mikroprocesor přistupuje do paměti nebo ke vstupním/výstupním zařízením
M=Memory IO=Input/Output
Mikroprocesor
M/IO
Operační paměť
20 b
adresová sběrnice
16 b
datová sběrnice
Klávesnice
Grafika
Zvuk
Síť

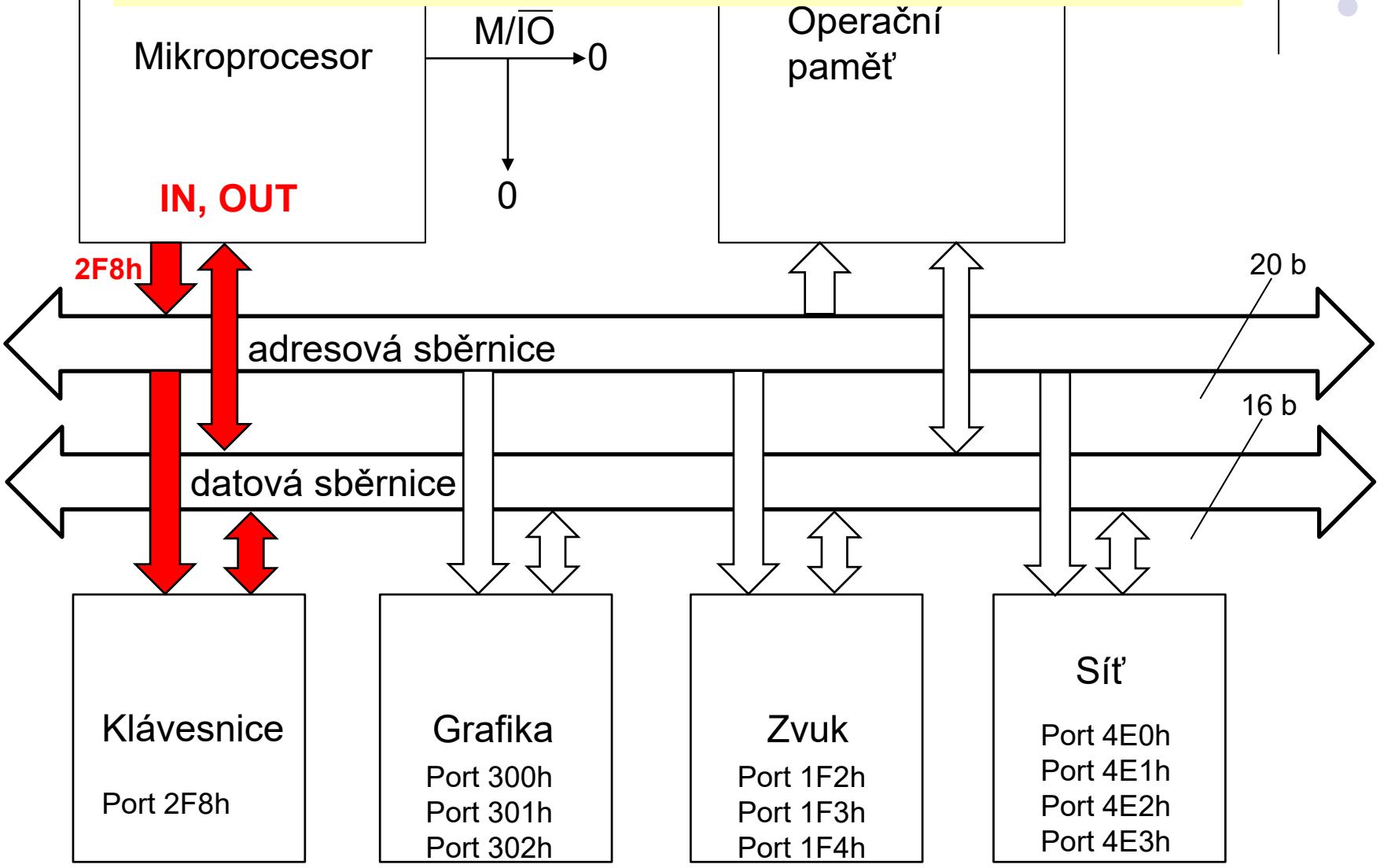
Do paměti mikroprocesor přistupuje pomocí instrukcí MOV. V tomto případě bude signál M/IO ve stavu jedna. Na adresové sběrnici se objeví adresa vybraného místa v paměti. Přes datovou sběrnici se z vybrané adresy přečtou data nebo se se na ní zapíší data

Adresa na adresové sběrnici slouží k výběru místa v operační paměti



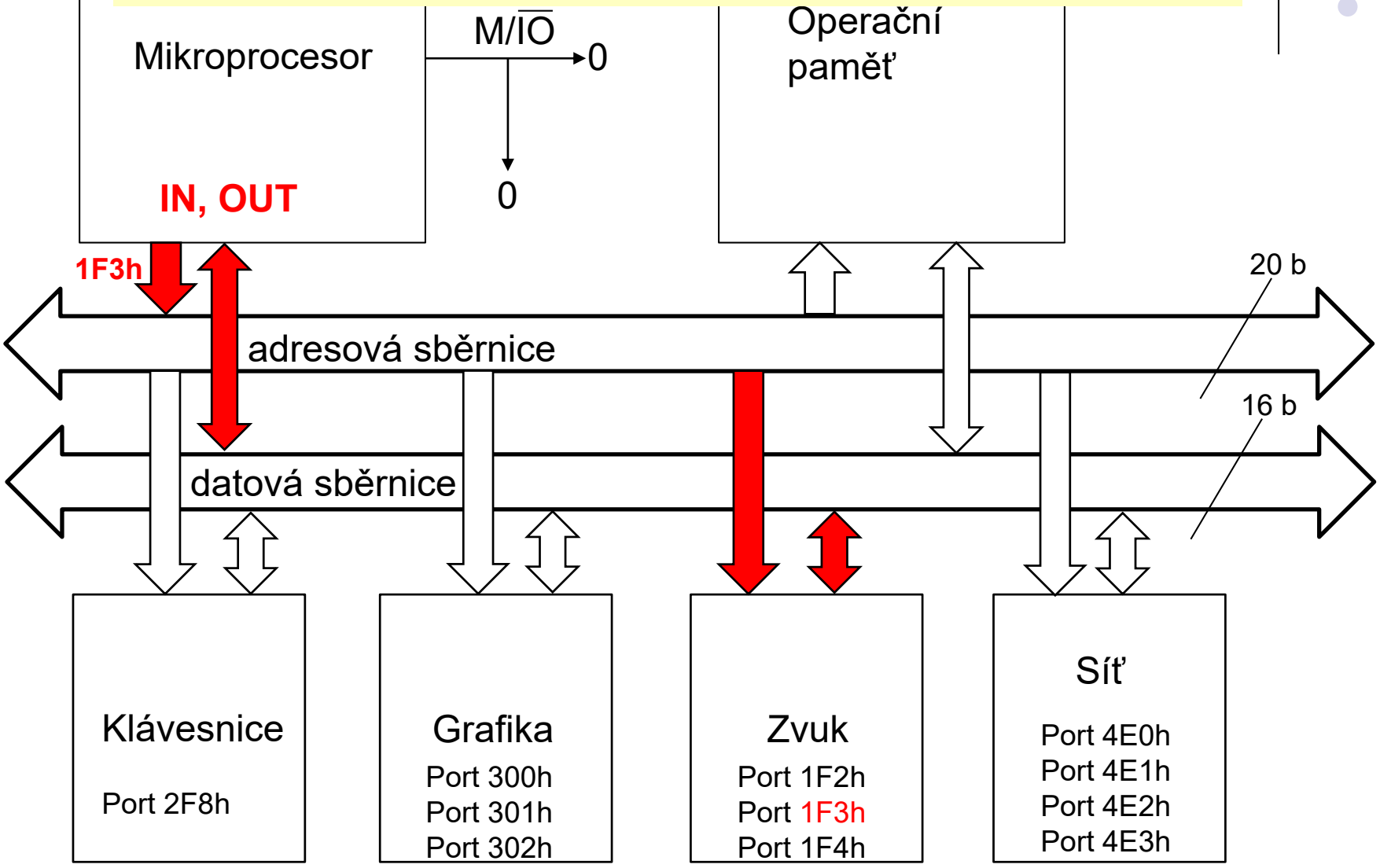
Ke vstupním/výstupním zařízením se přistupuje pomocí instrukcí IN a OUT. V tomto případě bude signál M/IO ve stavu nula. Na adresové sběrnici se objeví číslo brány (portu) a přes datovou sběrnici se z vybraného zařízení přečtou nebo do vybraného zařízení pošlou data.

Adresa na datové sběrnici představuje číslo brány (portu) a slouží k výběru vstupního/výstupního zařízení

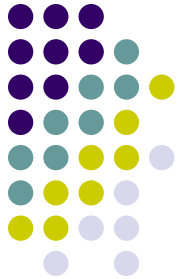


Ke vstupním/výstupním zařízením se přistupuje pomocí instrukcí IN a OUT. V tomto případě bude signál M/IO ve stavu nula. Na adresové sběrnici se objeví číslo brány (portu) a přes datovou sběrnici se z vybraného zařízení přečtou nebo do vybraného zařízení pošlou data.

Adresa na datové sběrnici představuje číslo brány (portu) a slouží k výběru vstupního/výstupního zařízení



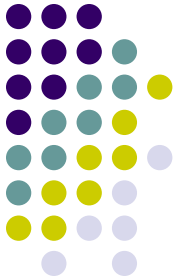
Základní podpůrné obvody



- **8284 - generátor hodinového signálu**
  - používá krystalový oscilátor
- **8259A - řadič přerušení**
  - v kaskádním zapojení umožňuje až 64 vnějších přerušení
  - Vedou do něj přerušovací signály od všech zařízení a podle režimu a priority je generován pro mikroprocesor signál INTR a vybráno přerušení s nejvyšší prioritou
- **8251 – UART**
  - obvod pro sériovou komunikaci
- **8255 – Paralelní port**
- **8237 – Řadič DMA**
- **8253 – Čítač časovač**
  - **3 nezávislé 16- bitové čítače směrem dolů**

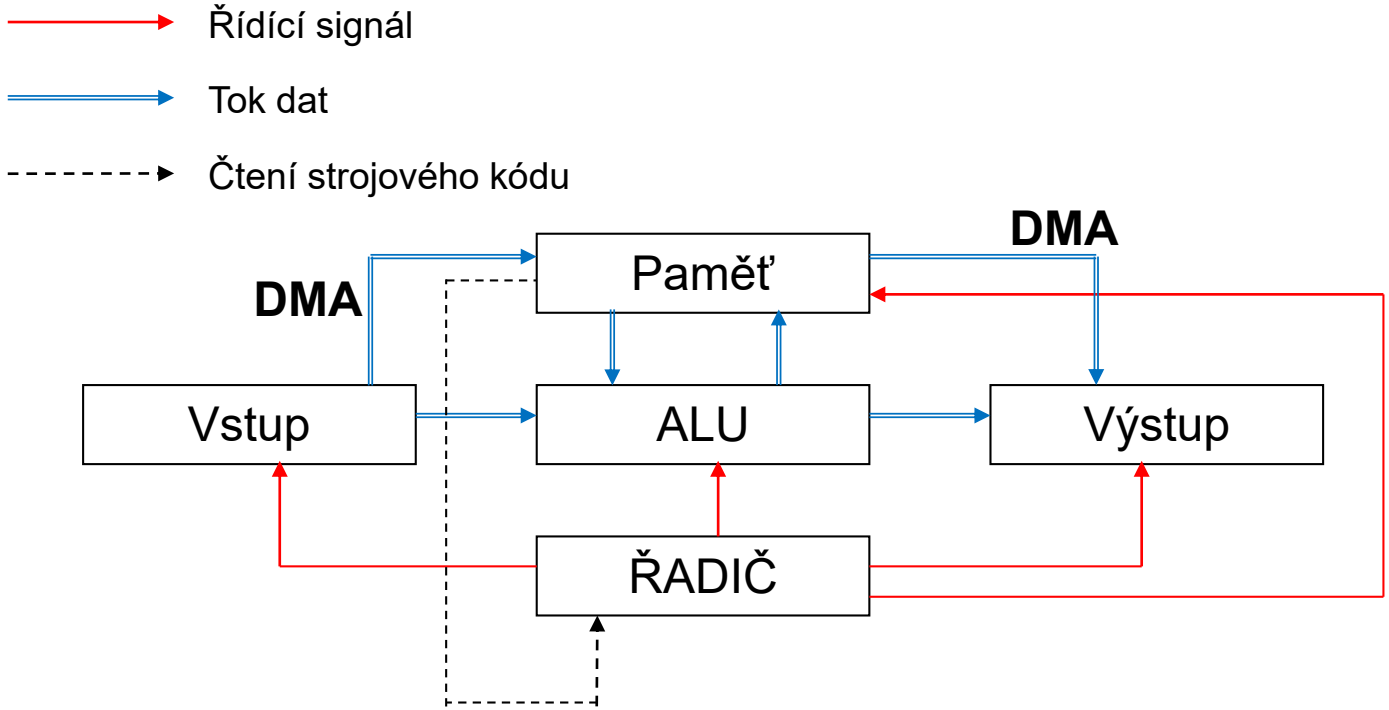
Vidíme tedy, že nejde o jednočipový mikropočítač, kde by bylo vše integrováno na jednom čipu, ale 8086 je klasický mikroprocesor, který potřebuje okolo sebe celou řadu samostatných podpůrných obvodů

DMA



- **Direct Memory Access** - možnost přenosu dat mezi I/O zařízením a pamětí bez účasti procesoru
- I/O zařízení může „samo“ přistupovat (zapisovat nebo číst) k datům v paměti
- Během přenosu dat mezi I/O zařízením a pamětí se mikroprocesor věnuje jiné činnosti
- Příklad využití - zvuková karta sama dostává z paměti zvuková data a generuje na výstupu zvukový signál, tato činnost nezatěžuje procesor
- Bez DMA by mikroprocesor musel každý bajt nejprve přečíst z paměti instrukcí **MOV** a potom ho instrukcí **OUT** poslat na bránu zvukové karty
- DMA přenosy řídí pomocný **řadič 8237**
- Přenos tedy probíhá bez účasti mikroprocesoru, ale I/O zařízení ve skutečnosti nepřistupuje do paměti samo, ale přenos dat řídí zvláštní **DMA řadič**, který je mikroprocesorem naprogramován
- DMA řadič realizuje 4 nezávislé DMA kanály 0,1,2 a 3
- **DMA kanál** = naprogramovaný přenos dat. Vybraná oblast paměti se DMA řadičem přenáší do I/O zařízení, zatímco mikroprocesor se věnuje jiné činnosti
- Od řady IBM PC/AT byly přítomny dva DMA řadiče a tak přibyly další kanály 4,5,6 a 7
- DMA nelze využít k přenosu mezi dvěma I/O - vždy musí jít o přístup zařízení do paměti

DMA


Řídící signál
Tok dat
Čtení strojového kódu
DMA
DMA
Paměť
Vstup
ALU
Výstup
ŘADIČ

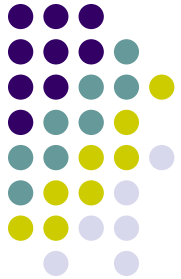
DMA

- Čtení 1 MB souboru z disku

Bez DMA
Čtení bajtu instrukcí IN z disku do registru AL
Zápis bajtu z registru AL do paměti instrukcí MOV
Čtení bajtu instrukcí IN z disku do registru AL
Zápis bajtu z registru AL do paměti instrukcí MOV
Čtení bajtu instrukcí IN z disku do registru AL
Zápis bajtu z registru AL do paměti instrukcí MOV
……a to se milionkrát opakuje, dokud nejsou všechny bajty přečteny z disku a zapsány do paměti, mikroprocesor neustále pracuje. Každý z milionu bajtů putuje z disku do paměti přes mikroprocesor

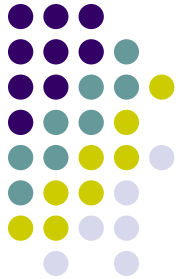
S využitím DMA
Nastavení DMA kanálu – mikroprocesor sdělí DMA řadiči z jakého portu se mají číst data a do jaké oblasti v paměti mají být zapsána
Tím práce pro mikroprocesor skončila a může se věnovat jiné činnosti
Mikroprocesor ani jeden z milionu přenášených bajtů nevidí – jdou mimo něj

BIOS



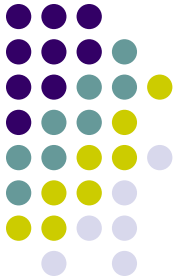
- Basic Input Output System
- Počítač po zapnutí má **prázdnou** operační paměť RAM, do které je potřeba odněkud zavést operační systém, který nám umožní spouštět další programy
- Proto jsou všechny osobní počítače vybaveny pamětí nezávislou na napájení (ROM, EPROM, EEPROM, FLASH), ve které je uložen tzv. BIOS
- Díky BIOSu umí počítač po zapnutí komunikovat s disketovou mechanikou, hard diskem, optickou mechanikou, flash diskem a načíst odtud operační systém nebo bootovat a spustit jiný program (např. Clonezilla, TrueImage…)
- Bez BIOSu by byl počítač po zapnutí pouze mrtvým kusem hardwaru – v prázdné paměti není žádný strojový kód a mikroprocesor nic nedělá

Služby BIOSu



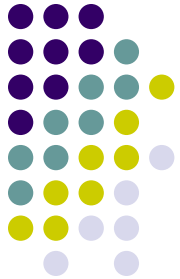
- BIOS umožňuje po zapnutí počítače také například výpis textu na obrazovku, komunikaci s klávesnicí a další základní vstupní a výstupní funkce, díky kterým může počítač po zapnutí komunikovat s okolním světem a „ožije“
- Součástí BIOSu jsou **služby** umožňující základní využití periferií
- Služby BIOSu lze volat z programů
- Tyto služby jsou k dispozici vždy, na každém na počítači a to již před zavedením operačního systému – lze je tedy použít například v programu, který bude spuštěn bez operačního systému (přímo bootován)

Služby BIOSu



- Existuje mnoho různých verzí BIOSu
- V každé verzi BIOSu jsou služby, které nabízí uloženy v paměti na jiném místě
- Pokud chce program zavolat funkci BIOSu pro výpis textu na obrazovku, neví na jaké adrese v paměti jí nalezne, protože na každém počítači to může být jinde
- Ve všech verzích BIOSu se musí jeho služby volat stejně – musí existovat univerzální způsob (něco jako mezinárodní tísňová linka 112)
- Služby BIOSu se z programu volají vygenerováním **softwarového přerušení –** instrukcí INT
- **Služby** se pak chovají jako podprogramy volané různými parametry
- Programátor na každém počítači bez ohledu na verzi BIOSu volá služby stejným způsobem

Služby BIOSu



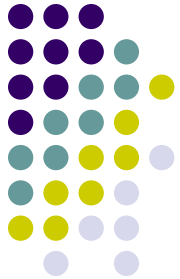
- Příklad
- Volání služby pro výpis znaku na obrazovku

```
MOV AH,9        - číslo služby, která má být zavolána
MOV AL, 65      - ASCII kód znaku, který má být vypsán
INT 10h         - přerušení, kterým voláme služby BIOSu
```

- Služby, které BIOS nabízí jsou očíslovány. Služba pro výpis znaku na obrazovku má pořadové číslo 9. Číslo služby, která je volána, se vždy vloží do registru AH.

- Bez zavolání služby BIOSu by byl výpis znaku na obrazovku pro programátora velmi komplikovanou záležitostí. Musel by složitě komunikovat s grafickým adapterem a uložit do jeho paměti data, která vyústí v rozsvícení požadovaných pixelů

Služby BIOSu



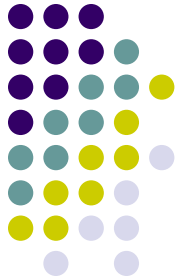
- Příklad
- Volání služby pro výpis znaku na obrazovku

```
MOV AH,9        - číslo služby, která má být zavolána
MOV AL, 65      - ASCII kód znaku, který má být vypsán
INT 10h         - přerušení, kterým voláme služby BIOSu
```

- Služby, které BIOS nabízí jsou očíslovány. Služba pro výpis znaku na obrazovku má pořadové číslo 9. Číslo služby, která je volána, se vždy vloží do registru AH.

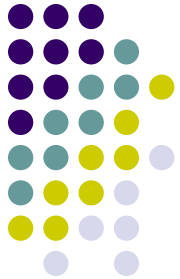
- Bez zavolání služby BIOSu by byl výpis znaku na obrazovku pro programátora velmi komplikovanou záležitostí. Musel by složitě komunikovat s grafickým adapterem a uložit do jeho paměti data, která vyústí v rozsvícení požadovaných pixelů

Služby DOSu



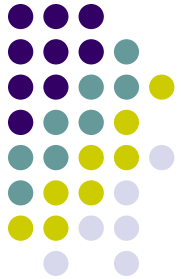
- Tyto služby jsou k dispozici až po zavedení **operačního systému**
- Každá verze DOSu (windows) má tyto služby řešeny trochu jiným způsobem
- Pomocí těchto služeb lze využívat vyšší úrovně abstrakce při styky s periferiemi, kterou nám nabízí operační systém (na disku vidíme soubory, adresáře, lze pracovat se standardním vstupem a výstupem)
- Všechny tyto služby se volají společně jako softwarové přerušení **INT 21h**
- Obsah registru **AH** pak určuje, jaká konkrétní služba se má provést
- Celá věc je tedy realizována tak, že operační systém nastaví vektor přerušení **INT 21h** tak, aby směroval na adresu, kde je kód, který čte obsah registru AH a podle toho se provádí skok na kód příslušné služby DOSu

Kontrolní otázky



- Jakou šířku má fyzická adresa ?
- Jakou šířku má offset ?
- Jakou šířku mají segmentové registry ?
- Jak lze snadno poznat číslo dělitelné šestnácti, když je zapsané v šestnáctkové soustavě ?
- Jak lze snadno poznat číslo dělitelné šestnácti, když je zapsané v dvojkové soustavě ?
- Jak velký je jeden segment ?
- Vysvětlete roli BIU a EU při pipeliningu.
- Proč obsahuje BIU 20-bitovou sčítačku ?
- V jakém roce byl uveden na trh šestnáctibitový mikropočítač IBM PC/XT ?
- Vyjmenujte datové registry.
- Vyjmenujte segmentové registry.
- Vyjmenujte indexové registry.
- Uveďte alespoň tři příznakové bity a vysvětlete jejich význam.

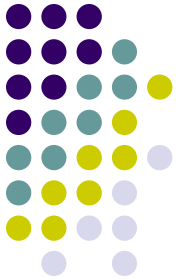
Kontrolní otázky



- Které z uvedených adres mohou být počáteční adresou kódového segmentu ?
  a) 12121h
  b) 0
  c) 10h
  d) 100h
  e) 1000h
  f) 10000h
  g) 100000h
  h) 120h
  i) 1200h
  j) 12000h
  k) 120000h
  l) 12002h

Správná odpověď: e, f, i, j

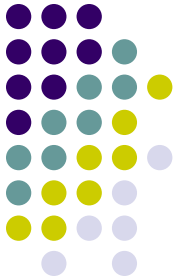
Kontrolní otázky



- SS=1234h, CS=2345h, DS=ABCDh, ES=CDEFh, IP=5231h, SP=5478h, BP=6542h, AX=4573h
  - Určete počáteční adresu datového segmentu
  - Určete adresu, na které končí pomocný datový segment
  - Určete adresu, ze které se právě čte strojový
  - Určete adresu, na které leží vrchol zásobníku
  - Určete adresu, na které leží první bajt, který byl uložen do zásobníku
  - Určete adresu, ne kterou bude proveden zápis bajtu instrukcí MOV [10],AL
  - Určete adresu, ze které bude provedeno čtení bajtu instrukcí MOV AH,ES:[33]
  - Určete adresu místa v paměti, na které se uloží bajt instrukcí PUSH AL
  - Jak se změní hodnota SP po provedení instrukce PUSH AL ?

  - Správné odpovědi: ABCD0h, DDEEFh, 28681h, 177B8h, 2233Fh, ABCDAh, CDF11h, 177B8h, 5477h

Kontrolní otázky

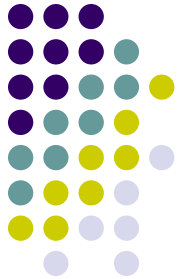


- Nakreslete Von Neumannovu architekturu s DMA

- AH=11, AL=22, BX=1025, CX=2047h, DH=23h, DL=7
  - AX=?
  - BL=?
  - BH=?
  - CH=?
  - CL=?
  - DX=?

  - Správně odpovědi: B16h, 1, 4, 20h, 47h, 2307h

Kontrolní otázky



- DS=1234h, AX=2345h
- Na jaké adresy do paměti bude zapsáno instrukcí MOV [64], AX a jaké konkrétní bajty se sem uloží ?

- Správné odpovědi: 12380h – 45h, 12381h – 23h