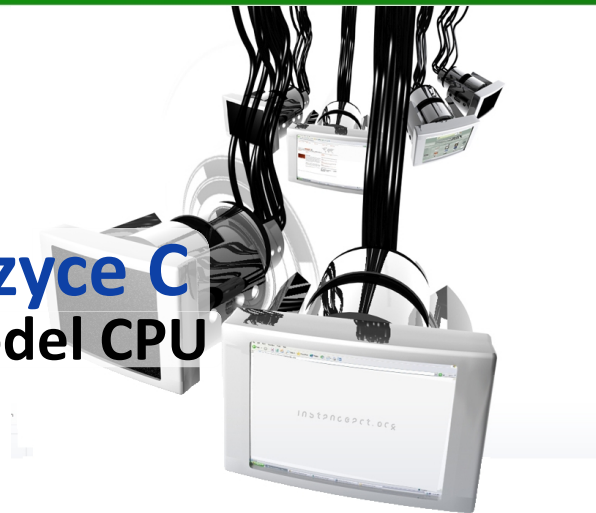




Programování v jazyce C

A Programátorský model CPU



- Vlastnosti CPU
- Instrukční soubor
- Datové typy
- Techniky adresování paměti
- Základy programování v assembleru
- Souvislost s vyššími programovacími jazyky



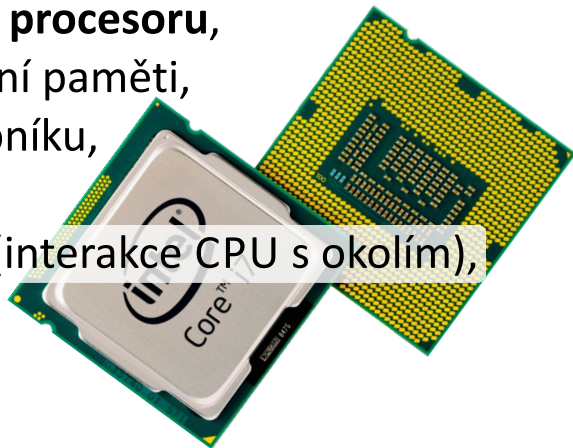
Programátorský model CPU

Co musí programátor vědět o procesoru?

Programátorský model = soubor vlastností procesoru, které ovlivňují jeho programování v nízkoúrovňových vyšších jazycích (a které má programátor znát, aby psal „rozumný“ kód).

Do programátorského modelu se zahrnuje zejména:

- (i) **instrukční soubor procesoru,**
 - (ii) **počet, uspořádání a pojmenování registrů,**
 - (iii) **nativní datové typy daného procesoru,**
 - (iv) techniky a rozsahy adresování paměti,
 - (v) konstrukce a používání zásobníku,
 - (vi) přerušovací systém,
 - (vii) ovládání externích zařízení (interakce CPU s okolím),
- a další vlastnosti...





Architektury moderních CPU

(na kterých má smysl si ukázat, co a jak...)

V současnosti (2024) se globálně nejvíce používají zejména **32-bitové** a **64-bitové** architektury CPU:

CISC

1. Intel IA-32/x86 (80386, 80486, 80586, P6/Pentium Pro)
2. x86-64/AMD64* (Opteron, Athlon, Turion, Phenom, Ryzen, ...)
3. x86-64/Intel 64* (Pentium 4, Celeron, Xeon, Core 2, Core iX)

RISC

4. ARM32/ARM64 (... , ARMv7, ARMv8, ARMv9, ...)

└─ Apple M1 (= ARMv8.4-A)
└─ Apple M2 (= ARMv8.6-A)

→ Výklad programátorského modelu CPU se zaměřením na architektury x86(-64) a ARM pokrývá většinu používaných CPU...

*Neplést s IA-64 (Itanium) — to je něco úplně jiného!



Registry procesoru

„Místo“ přímo v CPU pro dočasné uložení dat

- malé úložiště dat umístěné přímo v procesoru (tj. odpadá zdoluhavá komunikace s vnější operační pamětí)
- extrémně rychlý přístup, dostupné pomocí strojových instrukcí – instrukce si v registrech „přebírají“ argumenty a ukládají do nich své výsledky
- velikost registru je obvykle stejná jako velikost **slova procesoru** (a to je dáno **šířkou datové sběrnice**), tj. 32-bitový CPU má obvykle 32-bitové registry, 64-bitový má 64-bitové
- jednoduché procesory (mikrokontroléry, monolitické μ -počítače) mohou mít i jen jeden registr, tzv. **akumulátor ALU**; běžně jednotky až desítky registrů (CISC méně, RISC více);
- speciální procesory (DSP, GPU) mají stovky až tisíce registrů
- data se do registru ukládají pomocí instrukce **MOV** (Move), např.: **MOV R0, 100** = naplní registr R0 hodnotou 100



Registry procesoru

Registry pro všeobecné použití

Intel x86(-64)

- 64-bitové verze “původních” registrů procesorů 80x86:

63	...	31	23	15	7	0	
RAX	...	EAX		AH AX AL			<u>A</u> ccumulator
RBX	...	EBX		BH BX BL			<u>B</u> ase
RCX	...	ECX		CH CX CL			<u>C</u> ounter
RDX	...	EDX		DH DX DL			<u>D</u> ata
RSI	...	ESI		SI			<u>S</u> ource <u>I</u> ndex
RDI	...	EDI		DI			<u>D</u> estination <u>I</u> ndex
RBP	...	EBP		BP			<u>B</u> ase <u>P</u> ointer
RSP	...	ESP		SP			<u>S</u> tack <u>P</u> ointer

- “Nové” 64-bitové registry (při přechodu x86 → x86-64):
R8, R9, R10, R11, R12, R13, R14, R15
- Mnoho registrů pro operace s floating-point čísly a SIMD



Registry procesoru

Registry pro všeobecné použití

ARM32/64

- Registry pro argumenty/návratové hodnoty (32-/64-bitové):
R0, R1, R2, R3
- Registry bez zvláštního významu (32-/64-bitové):
R4, R5, R6, R7, R8, R9, R10, R11
 - aktuální hodnoty musí být vždy před případným použitím ukládány (do zásobníku)
- Stack pointer: **R13/SP**
- Link register: **R14/LR**
- Program Counter: **R15/PC**
- Mnoho speciálních řídicích registrů (**CPSR, SCTLR, CPACR, SCR, NSACR, ...**), k jejichž jednotlivým bitům „umí“ přistupovat konkrétní instrukce.



Registry procesoru

Práce s registry v assembleru

Intel x86(-64)

...

```
XOR    RAX, RAX
MOV    RBX, 10
MOV    RDX, 15
MUL    RBX, RDX
MOV    RCX, 20
ADD    RBX, RCX
MOV    RAX, RBX
INC    RAX
RET
```

...

ARM32/64

...

```
PUSH   {R4, R5, R6}
EOR    R0, R0, R0
MOV    R5, #10
MOV    R6, #15
MUL    R5, R5, R6
MOV    R4, #20
ADD    R0, R4, R5

ADD    R0, R0, #1
RET
```

...

→ Assembler se liší podle architektury CPU: Jiná jsou jména registrů a některé instrukce. Obecná koncepce je vždy stejná, tj. **v dalším výkladu se lze omezit na Intel x86.**



Intel 80386 (i386)

Etalon 32-bitového procesoru

- první 32-bitový procesor společnosti Intel, uveden na trh v roce 1985 — současné nejmodernější CPU Intel stále využívají zavedené koncepce (např. instrukční sadu IA-32, techniky adresování paměti, pojmenování registrů, atd.)
⇒ **proto má smysl ho studovat**
- dokáže obhospodařovat 4 GB **fyzické** a 64 TB **virtuální** paměti (jednotka správy paměti integrovaná na čipu)
- 3 režimy činnosti: **reálný** (*Real*), **chráněný** (*Protected*) a **virtuální 8086**
- CPU tvoří 6 paralelně pracujících jednotek (Bus Interface Unit, Code Prefetch Unit, Instruction Decoding Unit, Execution Unit, Segmentation Unit, Paging Unit)
- do fyzické paměti se data ukládají po 4KB stránkách



Typy dat a deklarace proměnných těchto typů

- 8-bitový **Byte** (slabika)
 - 16-bitový **Word** (slovo)
 - 32-bitový **Double Word** (dvojslovo)
 - 64-bitový **Quad Word** (čtyřslovo) — od 80586 dále
- všechny typy mohou být znaménkové/neznaménkové (*Signed/Unsigned*) podle instrukce použité pro výpočet (např. **SHL/SHR** vs **SAL/SAR**)

Toto jsou tzv. *nativní* datové typy. Jiné typy dat CPU nezná, protože pouze tyto lze přímo ukládat do registrů (aritmetiku reálných čísel se řeší FPU, má vlastní datové typy).

_DATA **SEGMENT**

char **DB** 'A'

var1 **DW** ?

handle **DD** 0

bignum **DQ** 0

capt **DB** 'Retezec', 0

_DATA **ENDS**

Define Byte

Define Word

... DoubleWord

... QuadWord



Typy dat a deklarace proměnných těchto typů

- 32-bitový procesor akceptuje jako operandy instrukcí maximálně 32-bitová slova (registry jsou 32-bitové).
- Je-li potřeba pracovat s 64-bitovými slovy (QW), používají se k tomu tzv. **registrové páry**:
EAX+EDX, EBX+ECX (obvykle jsou spárovány takto)
- v assembleru lze deklarovat i pole pomocí direktivy **DUP** (**duplicate**)

```
unsigned long int pointers[20] = {0};
```

```
_DATA SEGMENT
  pointers DD 20 DUP(0)
  colour   DB 100 DUP(?)
  prompt   DB 13, 10, 'Zadej něco:', 0
_DATA ENDS
```



Registry procesorů Intel 80x86

Segmentové registry (od 80386 dále)

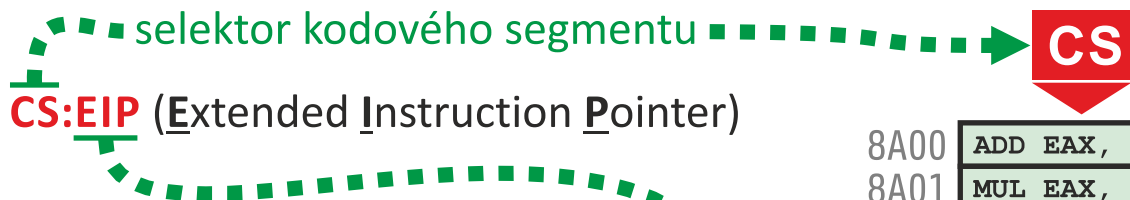
15	7	0	
			<u>C</u> ode <u>S</u> egment
			<u>D</u> ata <u>S</u> egment
			<u>S</u> tack <u>S</u> egment
			<u>E</u> xtra <u>S</u> egment
			Extra Segment
			Extra Segment

- viditelná část segmentových registrů je 16-bitová, plnit je lze instrukcí **MOV** nebo instrukcemi **LDS**, **LES**, **LFS**, **LGS**, **LSS**
- **změna obsahu CS má fatální následky** (obsahuje tzv. *selektor kódového segmentu*)



Registry procesorů Intel 80x86

Registry se zvláštním významem



EIP ukazuje na pozici v kódovém segmentu, kde leží právě prováděná instrukce, tj. **pár CS:EIP udává pozici právě vykonávané instrukce**, hodnotu EIP zvyšuje prováděcí jednotka CPU (není to přímo adresa v paměti, jedná se o **index do tabulky GDT/LDT** (Global/Local Descriptor Table), pozice GDT je v registru GDTR, LDT v registru LDTR – viz dále **Segmentace paměti**)

8A00	ADD EAX, 5
8A01	MUL EAX, EBX
8A02	JC 8A0A
8A03	SHR EAX, 1
8A04	CMP EAX, 0
8A05	JE 8A0D
8A06	NEG EAX
8A07	JMP 8A0F
8A08	...
8A09	
8A0A	
8A0B	
8A0C	
8A0D	
8A0E	
8A0F	
8A10	



Registry procesorů Intel 80x86

Registry se zvláštním významem

- CS:EIP** - pozice vykonávané instrukce (mění ji sám procesor)
- SS:ESP** - pozice vrcholu zásobníku (mění ji instrukce **PUSH** a **POP**)
- DS:ESI** - zdrojová adresa pro instrukce blokového přesunu dat (**MOVS/MOVSb/MOVSsw**)
- ES:EDI** - cílová adresa pro instrukce blokového přesunu dat
- pro programátora (zvláště nezkušeného) **read-only!**

Kopírování řetězce (pole bytů) v assembleru:

```
LDS SI, <adresa zdrojového řetězce>
LES DI, <adresa cílového řetězce>
MOV CX, <počet prvků řetězce>
REP MOVSB
```

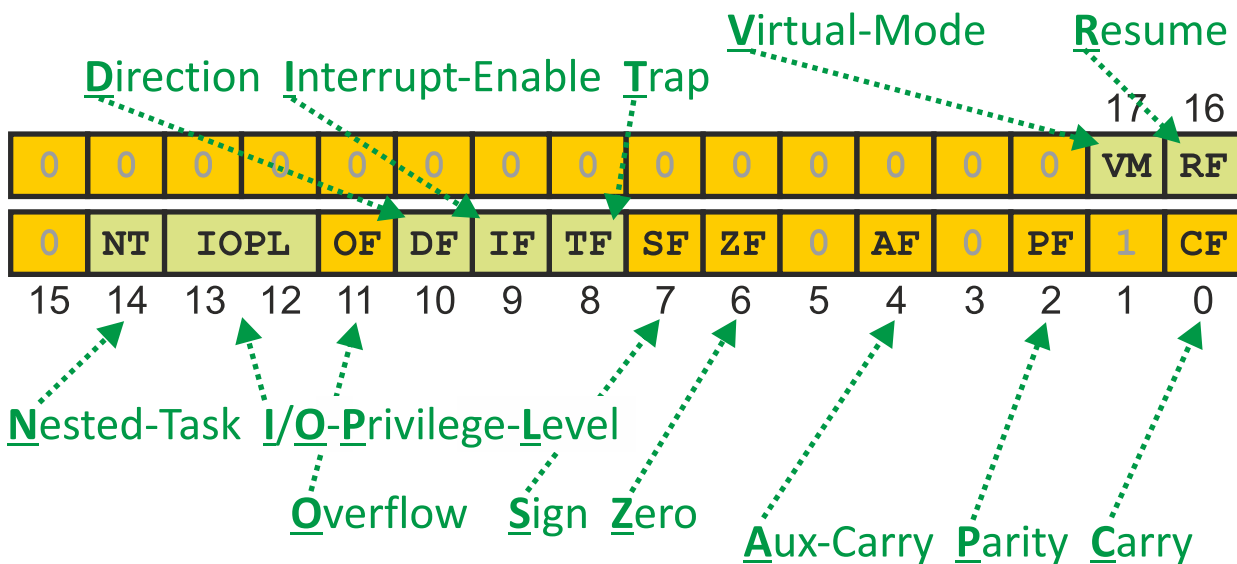




Registry procesorů Intel 80x86

Příznakový registr EFLAGS

- Obsahuje dva druhy **vlajek** (1-bitových příznaků):
 - nastavované procesorem** po provedení instrukce indikují vlastnosti výsledku (CF, PF, AF, ZF, SF, OF),
 - nastavované programátorem** řídí činnost procesoru (TF, IF, DF, VM, RF, NT, IOPL).

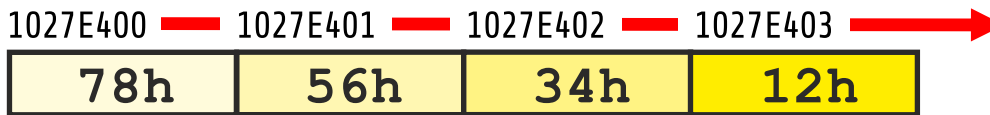




Endian processoru

Uspořádání dat ve slovech v paměti

Procesory Intel (a jejich klony) používají tzv. **Little Endian**, což znamená, že nižší řády čísla jsou na nižších adresách:



num DD 12345678h

PŘÍČINA ČASTÝCH
PROBLÉMŮ

- Procesory Motorola, AIM PowerPC (po G5), Sun SPARC (do verze V9), IBM System/370 používají **Big Endian**, tzn. nižší řády na vyšších adresách (tj. tak, jak se číslo píše na papír).
- Některé procesory umí endian podle potřeby přepínat, např. ARM, SPARC V9, MIPS, PA-RISC, IA64, DEC Alpha, některé PowerPC ⇒ tzv. **Bi-Endian**.



Techniky adresování paměti

Typy adres

- 3 druhy adres:
 - (i) **logická** (virtuální) – adresuje 64 TB virtuální paměti, skládá se z 16-bitového **selektoru** a 32-bitového **offsetu**. Segmentační jednotka CPU jí převádí na lineární adresu.
 - (ii) **lineární** – 32-bitová adresa (adresuje 4 GB paměti). Je-li stránkovací jednotka vyřazena, adresuje přímo fyzickou paměť počítače.
 - (iii) **fyzická** – 32-bitová adresa (tj. adresuje až 4 GB), která představuje skutečnou pozici v paměti počítače. Je shodná s lineární adresou, pokud není v činnosti stránkovací jednotka.

Programátor aplikací nejčastěji pracuje pouze s offsetem logické adresy (méně často s úplnou logickou adresou). Adresy (ii) a (iii) se využívají pouze při programování jádra OS či driverů.





Adresování paměti v praxi

Čtení/zápis do paměti v assembleru

```
_DATA SEGMENT
    prompt DB 'Hello, world!', 0
_DATA ENDS

_CODE SEGMENT
    MOV     EAX, OFFSET prompt
    ADD     EAX, 12
    MOV     BL, '?'
    MOV     [EAX], BL

    MOV     EAX, OFFSET prompt
    PUSH    EAX
    CALL    WriteLine@4
_CODE ENDS
```

získání adresy objektu
(proměnné, funkce, atp.)

zapsání hodnoty z registru
BL na adresu uvedenou
v registru EAX





Segmentace a stránkování paměti

Smysl a účel transformace adres

- U komplexního výpočetního systému **nemůže mít adresa význam** přímo **pozice v paměťových obvodech**:
 - (i) všechny konfigurace takového systému by musely mít **stejnou velikost paměti**;
 - (ii) nemožné implementovat **ochranu adresního prostoru** jednotlivých běžících procesů, tj. velmi problematická implementace multitaskingu;
 - (iii) nemožné používat **virtuální paměť**, tj. více paměti pro procesy, než je fyzicky instalováno v počítači;
 - (iv) další, avšak mnohem méně důležité důvody.
- Proto mají procesory Intel (a klony) řady 80x86 implementované sofistikované mechanismy transformace adres.



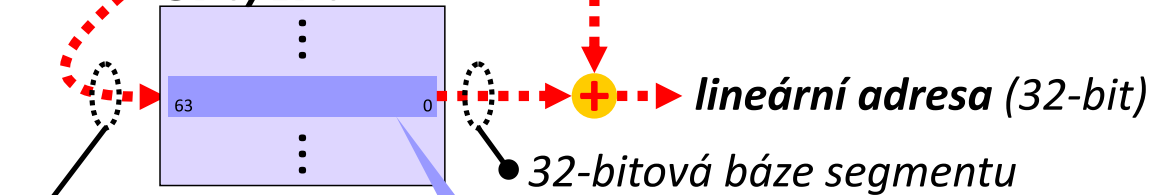
Dvě úrovně transformace adres (úroveň 1)

- převod **logické adresy** na **lineární** via GDT/LDT

logická adresa



GDT/LDT



- index jedné z 8192 položek tabulek popisovačů segmentů

**Global Descriptor
Table (GDT) nebo LDT**

popisovač segmentu

báze segmentu (31 – 24) G s 0 A V L limit (19 – 16)	přístup. práva báze segmentu (23 – 16)
báze segmentu (15 – 0)	limit segmentu (15 – 0)

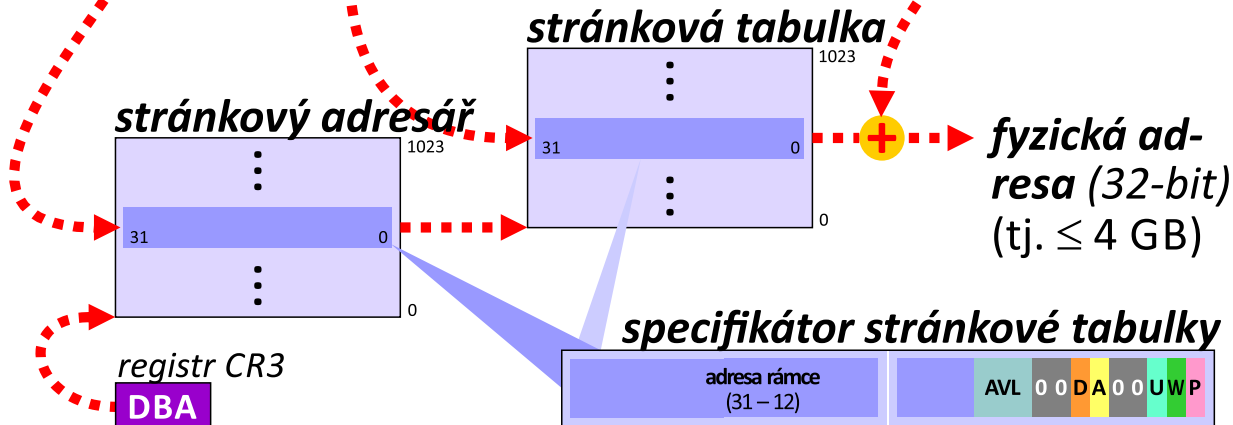
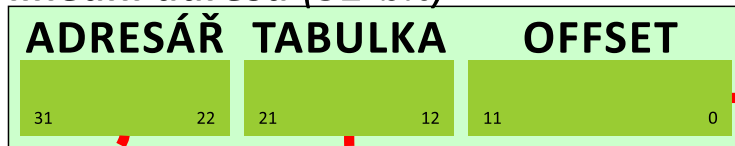


Stránkování paměti

Dvě úrovně transformace adres (úroveň 2)

- převod **lineární adresy** na **fyzickou** prostřednictvím stránkovací jednotky CPU (*Paging Unit*)

lineární adresa (32-bit)

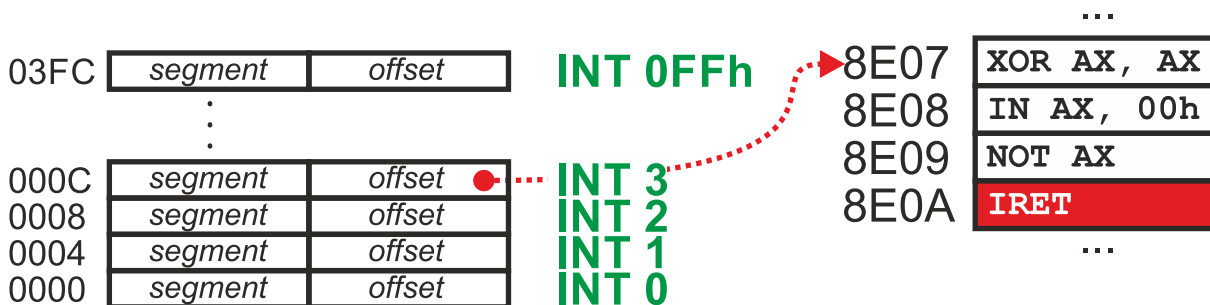




Přerušení (Interrupt)

Jednoduché μ -procesory, Intel 8086

- **tabulka přerušovacích vektorů** je umístěná na fyzickém počátku paměti od adresy 0 (i8086 0000:0000)
- adresa ukazuje na začátek obslužné rutiny přerušení (musí končit instrukcí **IRET**)
- přerušení může být vyvolané buď HW (z vnějšku přivedením log. úrovně na daný pin procesoru) či SW instrukcí **INT n**
- HW přerušení lze **maskovat** vynulováním příznaku **IF** instrukcí **CLI** (kromě vnitřních a nemaskovatelných, tzv. **Non-Maskable Interrupt** / NMI).





Činnost CPU při přerušení

Jednoduché μ -procesory, Intel 8086

Nastalo přerušení **n** nebo CPU dekódoval instrukci **INT n**:

- (i) do zásobníku se uloží registr příznaků (**FLAGS**),
- (ii) vynulují se příznaky **IF** a **TF**,
- (iii) do zásobníku se uloží registr **CS**,
- (iv) **CS** se naplní obsahem adresy $n * 4 + 2$,
- (v) do zásobníku se uloží **IP** ukazující na další **neprovedenou instrukci** přerušeného procesu,
- (vi) **IP** se naplní obsahem adresy $n * 4$.





Přerušení (Interrupt)

Komplexní CPU se segmentací paměti

- **tabulka popisovačů obsluhy přerušení** (IDT = *Interrupt Descriptor Table*) může být umístěna kdekoliv v paměti, adresa IDT je umístěna v registru **IDTR** (čte/plní se instrukcí **SIDT/LIDT**)
- položka IDT se nazývá **popisovač brány přerušení**, brány jsou pro (i) maskovatelná přerušení (Interrupt Gate) a (ii) nemaskovatelná přerušení (Trap Gate)

IDTR (*Interrupt Descriptor Table Register*)

<i>báze</i>	<i>limit</i>
-------------	--------------

offset (31 – 16)	oprávnění
selektor	offset (15 – 0)

⋮

+000C	offset (31 – 16)	oprávnění
+0008	selektor	offset (15 – 0)
+0004	offset (31 – 16)	oprávnění
+0000	selektor	offset (15 – 0)

INT 1

INT 0

10FA8E07	XOR AX, AX
10FA8E09	IN AX, 00h
10FA8E0B	NOT AX
10FA8E0D	IRET

...

...



Hardwarová přerušení

Upozornění CPU na výjimečné stavy z vnějšku

- **INT 0** až **INT 15** jsou rezervována pro ošetřování výjimečných stavů generovaných hardwarem počítače

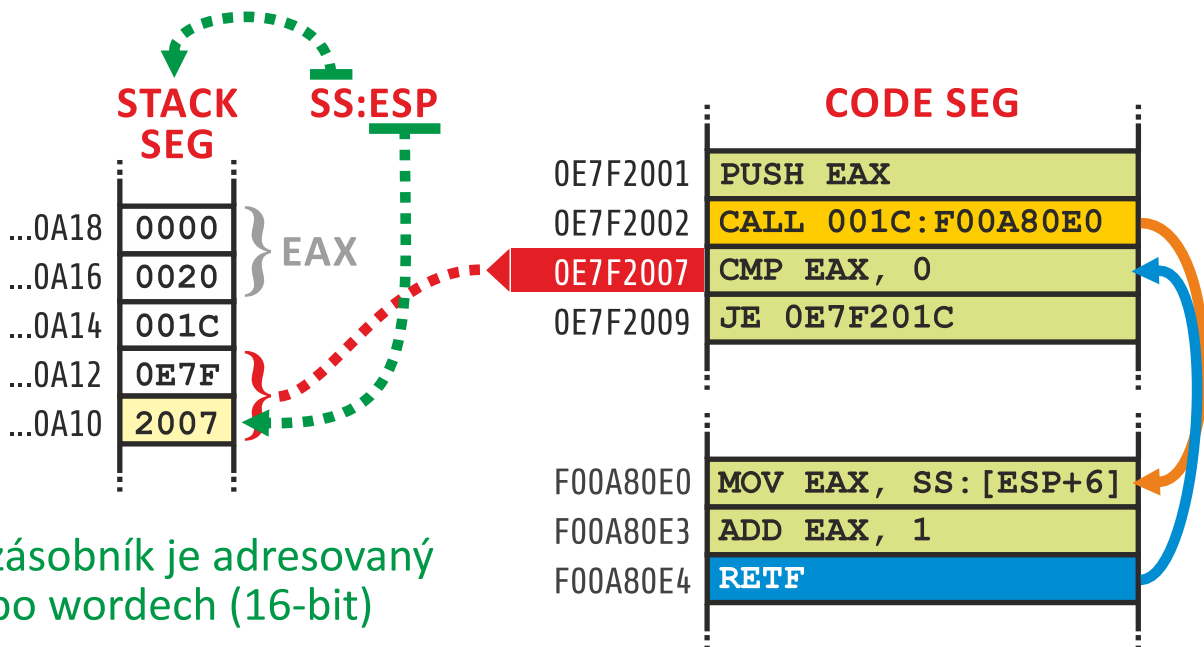
INT_NUM	Short Description PM		
0x00	Division by zero	0x0D	General protection fault
0x01	Debugger	0x0E	Page fault
0x02	NMI	0x0F	reserved
0x03	Breakpoint	0x10	Math Fault
0x04	Overflow	0x11	Alignment Check
0x05	Bounds	0x12	Machine Check
0x06	Invalid Opcode	0x13	SIMD Floating-Point Exception
0x07	Coprocessor not available		
0x08	Double fault		
0x09	Coprocessor Segment Overrun (386 or earlier only)		
0x0A	Invalid Task State Segment		
0x0B	Segment not present		
0x0C	Stack Fault		



Volání podprogramů

Mechanismus předávání parametrů

- před odskokem do podprogramu se do zásobníku uloží ***návratová adresa*** (její tvar záleží na typu volání: **near** / **far**)
- parametry se podprogramu předávají buď přes zásobník nebo v registrech (záleží na překladači: **stdcall** / **fastcall**)





Volání podprogramů

Realizace v assembleru

- pokud se podprogram v ASM linkuje k programu v C, musí se shodovat **paměťový model** (použití segmentových registrů) a **volací konvence** (způsob předávání parametrů)

```
_TEXT SEGMENT WORD PUBLIC 'CODE'
```

```
    public _power2
```

```
_power2 proc near
```

```
    push ebp
```

```
    mov ebp, esp
```

```
    mov eax, [ebp+4] ; první argument
```

```
    mov ecx, [ebp+6] ; druhý argument
```

```
    shl eax, cl      ; EAX = EAX * ( 2 ^ CL )
```

```
    pop ebp
```

```
    ret
```

```
_power2 endp
```

```
_TEXT ends
```

```
END
```

pod tímto názvem funkci uvidí linker, tj. v jazyce C jí lze volat jako `int power2(int, int)`

paměťový model **FLAT**, tj. **CS = DS = SS** (= ES, FS, GS) – vše je v jediném segmentu (jinak: `mov eax, ss:[ebp + 4]`)



Paměťové modely

Alokace segmentů pro potřeby procesu

- **Paměťový model** (na procesorech Intel 6 různých) určuje, jak jsou procesu alokovány segmenty paměti, tj. co je obsahem segmentových registrů a jaký tvar a velikost má ukazatel.
- 16-bitové programy: TINY, SMALL, MEDIUM, COMPACT, LARGE, HUGE
- 32-bitové programy: **FLAT** (= TINY), tzn. všechny segmentové registry jsou nastavené na stejnou hodnotu (selektor)
- 64-bitové programy: 7 různých paměťových modelů
- Problematika paměťových modelů (a v návaznosti na to segmentace paměti a stránkování) je velmi rozsáhlá a komplikovaná, mimo rámec předmětu KIV/PC. Vážní zájemci nechtě si prostudují **Intel® 64 and IA-32 Architectures Developer's Manual**, PDF ke stažení z <http://www.intel.com>



Porty procesoru

Komunikace s vnějšími HW zařízeními

- pomocí tzv. I/O portů, sběrnice může přenášet data buď mezi CPU a pamětí nebo CPU a ostatními zařízeními na MB
- signál $M/\overline{IO}$ určuje, zda adresa nastavená CPU na adresních vodičích $A_0 - A_{31}$ je adresou v paměti nebo I/O portu

@L1:

```
mov al, 0Ah ; 0Ah - offset 'valid'
out 70h, al ; 70h - CMOS index port
in al, 71h ; 71h - CMOS data port
test al, 10000000b
jnz @L1 ; bit7 = 1, znovu
```

```
xor al, al
out 70h, al
```

```
in al, 71h ; čteme bajt 0 - sekundy
```

```
in eax, 61h
in eax, dx
```

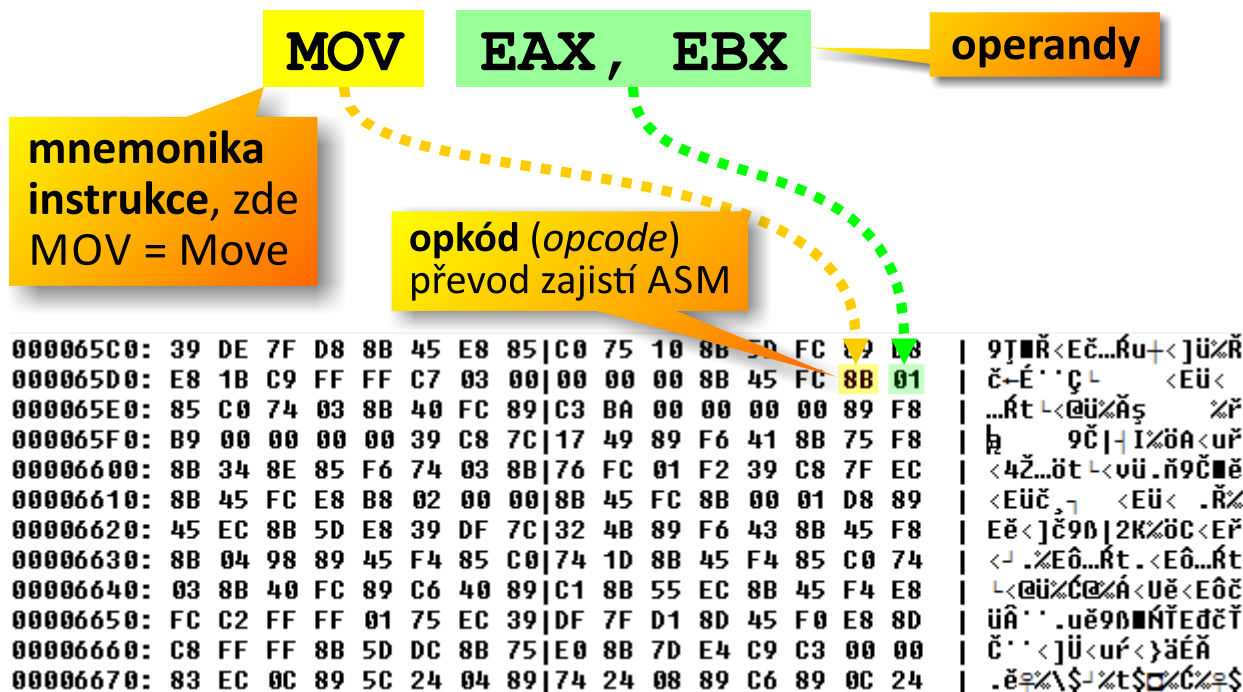
```
out 20h, eax
out dx, eax
```



Instrukční sada

Tvar instrukcí CPU s úplnou instrukční sadou

- CPU Intel – tzv. *úplná instrukční sada* (*Complete Instruction Set Computer*, CISC), řádově stovky instrukcí





Instrukční sada

Instrukce pro přesun dat

- Instrukce **MOV** přenáší obsah zdrojového operandu (tj. toho, který stojí více vpravo) do cílového operandu. Obsah zdrojového operandu se nemění.

```
MOV    ESI, OFFSET retez1
MOV    EDI, OFFSET retez2
MOV    ECX, 10
REP    MOVSB
```

- **MOVSB/MOVSW/MOVSDB** – byte/word/dword z adresy DS:ESI přepíše na adresu ES:EDI. Lze použít prefix **REP**, který zajistí opakování podle obsahu registru CX (při každém cyklu se sníží hodnota CX, při dosažení 0 cyklus končí).



Instrukční sada

Aritmetické instrukce

MOV EAX, 0

$EAX \leftarrow EAX + 1$

INC EAX

SUB EAX, EBX

$EAX \leftarrow EAX - EBX$

DEC EAX, CL

$EAX \leftarrow EAX - CL$

MUL EBX

$EAX \leftarrow EAX \times EBX$

NEG EAX

$EAX \leftarrow -EAX$ (dvojkový doplněk)

DIV EDX

$EAX \leftarrow EAX / EDX$

- **MUL**: Je-li výsledek větší než 32-bitů, je umístěn do reg. páru EDX+EAX
- **DIV**: Je-li operand 32-bitový, dělí se obsah páru EDX+EAX. Do EAX je uložen celočíselný podíl, do EDX zbytek po dělení.



Instrukční sada

Logické instrukce

- **CMP** (= Compare): Porovnává zdrojový a cílový operand.
- **AND**: logický součin
- **OR**: logický součet
- **XOR**: nonekvivalence
- **NOT**: negace (jedničkový doplněk)
- **TEST**: logické porovnání pomocí konjunkce

Instrukcí jsou stovky – nemá smysl je všechny dopodrobna rozebírat (není na to v rámci KIV/PC čas).

Zájemci o programování v assembleru:

- Vlad Pirogor: *Mistrovství v jazyce Assembler*. Computer Press, Brno, 2005. ISBN 80-251-0888-0.
- Intel® 64 and IA-32 Architectures Software Developer Manuals. <http://www.intel.com>