B35APO: Architektury počítačů

Lekce 11. Architektura x86

Petr Štěpán
stepan@fel.cvut.cz

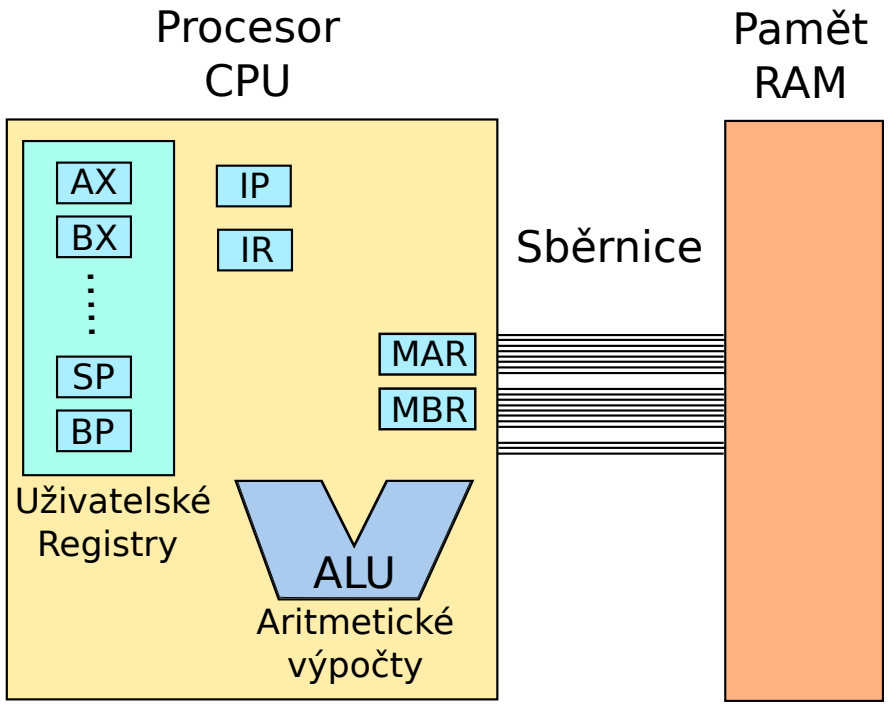
16. května, 2020

Obsah

Procesor – CPU

Základní vlastnosti:

- šířka datové a adresové sběrnice
- počet a velikost vnitřních registrů
- rychlost řídicího signálu – frekvence
- instrukční sada



Historie – x86/AMD64

- x86 - rodina procesorů, x je zkratka pro hodnoty x - 0,1,2,3,4,5,6

8086 – A16 F20 (1978) první IBM PC (8088 - 1979)
80286 – A16 F24 (1982) protected mode
80386 – A32 F32 (1985) stránkování
80486 – A32 F32 (1989) pipelining, FPU, cache
80586 – A32 F32 (1993) Pentium superscalar
80686 – A32 F36 (1995) Pentium Pro PAE, L2 cache, out-of-order & speculative exec
IA-64 – A52 F52 (2001) Itanium 64-bitová verze
AMD64 – A40 F40 (2003) Athlonn 64-bitová verze od AMD
Core2 – A36 F36 (2006) Intel 64 EM64T, SSSE3, μop, virtualization

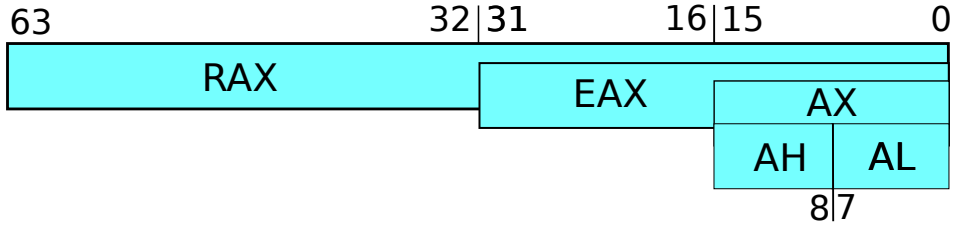
- Přehledný popis – `https://en.wikibooks.org/wiki/X86_Assembly`

Obsah

Procesor – x86/AMD64

Uživatelské registry

- Všechny registry vzhledem ke zpětné kompatibilitě jsou 64/32/16/8 bitové
- obecné registry pro ukládání hodnot programu `eax`, `ebx`, `ecx`, `edx`
- registry specializované jako ukazatel do paměti `esi`, `edi`, `ebp`
- `esp` – stack pointer – ukazatel zásobníku - detailněji dále
- AMD64/EM64T přidává 8 dalších registrů `r8`-`r15`,ve formě `r8b` nejnižší bajt, `r8w` nejnižší slovo (16 bitů), `r8d` – nižších 32 bitů, `r8` – 64 bitový registr

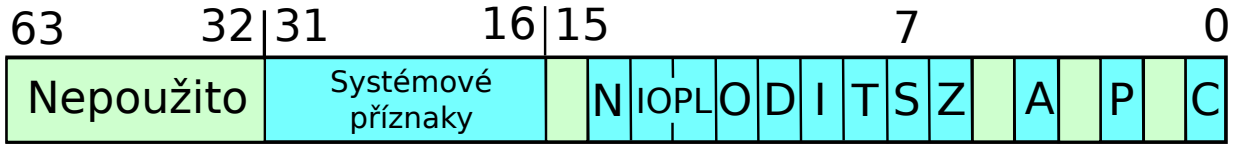


Řídicí a stavové registry

- `IP/EIP/RIP` – instruction pointer – adresa zpracovávané instrukce
- `FLAGS/EFLAGS/RFLAGS` – stav procesoru

Registr FLAGS

RFLAGS registr



C – Carry flag
P – Parity flag
Z – Zero flag
S – Sign flag
O – Overflow flag
A – Auxiliary flag (BCD)

I – Intterupt enable
T – Trap flag
IOPL – I/O privilege level
Systémové příznaky:

- VM – Virtual 8086 Mode
- VIF – Virtual Interrupt Flag
- VIP – Virtual Interrupt Pending

Režimy práce procesoru

FLAGS registr

- Dva režimy práce procesoru IOPL – základ hardwarových ochran
  - CPL0[1] = privilegovaný (systémový) režim
    - procesor může vše, čeho je schopen
  - CPL3 = uživatelský (aplikační) režim
    - privilegované operace jsou zakázány
- Privilegované operace
  - ovlivnění stavu celého systému (halt, reset, Interrupt Enable/Disable, modifikace Flags, modifikace registrů MMU)
  - instrukce pro vstup/výstup (in, out)
- Přechody mezi režimy
  - Po zapnutí stroje systémový režim
  - Přechod do uživatelského – modifikace Flags (popf nebo reti)
  - Přechod do systémového – pouze přerušení vč. programového

[1] Current privilege level

Obsah

Instrukce – x86/AMD64

Instrukce “ulož hodnotu”
(běžně se používají dvě různé syntaxe pro zápis assembleru)

AT&T	**Intel**
`movq` zdroj 64b, cíl	`mov` cíl, zdroj
`movl` zdroj 32b, cíl	
`movw` zdroj 16b, cíl	
`movb` zdroj 8b, cíl	
registry se značí	
`%ax`	pouze `ax`
hodnoty $, hex 0x	číslo, hex postfix h
`movl $0xff, %ebx`	`mov ebx, 0ffh`

Instrukce – x86/AMD64

Načti hodnotu z adresy (odkaz do paměti)

AT&T	Intel
`movl (%ecx),%eax`	`mov eax, [ecx]`
`movl 3(%ebx), %eax`	`mov eax, [ebx+3]`
`movl (%ebx, %ecx, 0x2), %eax`	`mov eax, [ebx+ecx*2h]`
`movl -0x20(%ebx, %ecx, 0x4), %eax`	`mov eax, [ebx+ecx*4h-20h]`

- odkaz má 4 složky: *základ+index*měřítko+posun*
- *měřítko* může nabývat hodnot 1,2,4,8
- lze implementovat přístup do pole struktur: *základ* je ukazatel na první prvek, *index*měřítko* říká, který prvek chceme a *posun*, kterou položku uvnitř struktury potřebujeme.
- není potřeba použít všechny 4 složky

Instrukce opakování – x86/AMD64

Instrukce pro řetězce - `REP` opakování pro pole hodnot

- opakuj dokud `ecx>0`:
  - `operace (%esi), (%edi)`
  - `esi += d*operand_size`
  - `edi += d*operand_size`
  - `ecx - -`
- operace může být `movs, cmps, lods, stos, scas, ins, outs`
- `d` - určuje směr a je buď $+1$, nebo -1
- `REP` opakování podle hodnoty `ecx`
- `REPE/REPNE` opakování podle hodnoty `ecx` a podle porovnání
  - operace `cmps` se navíc zastaví, pokud je/není rozdíl mezi `(edi)` a `(esi)`
  - operace `scas` se navíc zastaví, pokud je/není rozdíl mezi `(edi)` a hodnotou v registru `eax`

Instrukce opakování – x86/AMD64

Příklad nastav pole na hodnotu -1:

```
int array[128];
for (int i=0; i<128; i++) {array[i]=-1;}
```

přeloženo:

```
mov array, %edi ; Nastav do edi ukazatel na zacatek pole
mov $128, %ecx ; Nastav pocet opakovani
mov $-1, %eax ; Nastav hodnotu pro ulozeni
rep stosd ; Vypln cele pole
```

Instrukce opakování – x86/AMD64

Najdi konec řetězce:

```
char str[128];
int i;
for (i=0; i<128; i++) {if (str[i]==0) break;}
```

přeloženo:

```
mov  str,  %edi  ; Nastav do EDI zacattek pole
mov  $128, %ecx  ; Nastav pocet opakovani
mov  $0,   %eax  ; Nastav hodnotu konce retezce
rep scasb        ; Projdi str a najdi 0 hodnotu
```

Instrukce – x86/AMD64

Aritmetika – AT&T syntax

operace co, k čemu

`addq $0x05,%rax`	rax = rax + 5
`subl -4(%ebp), %eax`	eax = eax – mem(ebp-4)
`subl %eax, -4(%ebp)`	mem(ebp-4) = mem(ebp-4)-eax

následující instrukce mají argumenty typu X – b, w, l, q

`andX`	bitový and
`orX`	bitový or
`xorX`	bitový xor (nejrychlejší vynulování registru)
`mulX`	násobení čísel bez znamének
`divX`	dělení čísel bez znamének
`imulX`	násobení čísel se znaménky
`idivX`	dělení čísel se znaménky

Instrukce – x86/AMD64

Aritmetika s jedním operandem – AT&T syntax

operace s cim

`incl %eax`	eax = eax + 1
`decw (%ebx)`	mem(ebx) = mem(ebx)-1
`shlb $3, %al`	al = al<<3
`shrb $1, %bl`	bl=11000000, po bl=01100000
`sarb $1, %bl`	bl=11000000, po bl=11100000
`rorx, rolx`	bitová rotace doprava a doleva
`rcrx, rcl`	bitova rotace – pres C – carry flag

Instrukce – x86/AMD64

Podmíněné skoky

`test a1, a2`	tmp = a1 AND a2, Z tmp=0, C tmp<0
`cmp a1, a2`	tmp = a1-a2, Z tmp=0, C tmp<0

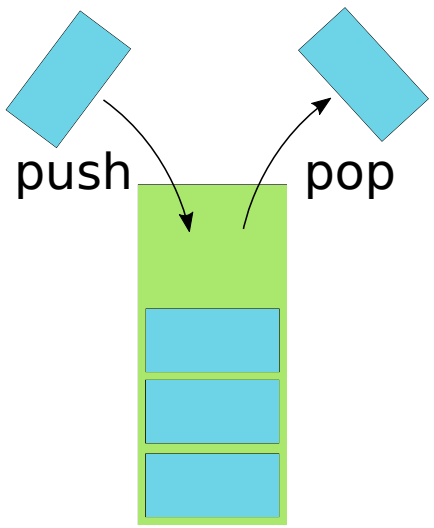
pak lze použít následující skoky

`jmp kam`	nepodmíněný skok, vlastně `%eip=kam`
`je kam`	jmp equal – skoč při rovnosti
`jne kam`	jmp not equal – skoč při nerovnosti
`jg/ja kam`	jmp greater – skoč pokud je a1 > a2 (sign/unsig)
`jge/jae kam`	skoč pokud je a1 >= a2 (sign/unsig)
`jl/jb kam`	jmp less – skoč pokud je a1 < a2 (sign/unsig)
`jle/jbe kam`	skoč pokud je a1 <= a2 (sign/unsig)
`jz/jnz kam`	skoč pokud je Z=1/0
`jo/jno kam`	skoč pokud je O (overflow) = 1/0

Zásobník

Zásobník:

- obecná struktura LIFO
- operace push vloží data do zásobníku
- operace pop vybere data ze zásobníku



Implemetace:

- implementace registrem *SP* - ukazuje na vrchol zásobníku
- konvence - při každém pop se zvětšuje registr *SP* o velikost operandu, při push se *SP* zmenšuje.

`pushl %eax`	ulož eax na zásobník
`popw %bx`	vyber ze zásobníku 2 bajty do *bx*
`pushf/popf`	ulož/vyber register EFLAGS
`pusha/popa`	ulož/vyber všechny uživatelské registry

Funkce zásobníku

Zásobník:

- parametry pro funkci
- kam se vrátit po ukončení funkce, místo odkud program volal funkci
- lokální proměnné funkce
  - zásobník je vetšinou malý
  - omezená velikost lokálních proměnných
  - pozor při rekurzi - lépe se rekurzi vyhnout

Instrukce – x86/AMD64

Volání funkce

`call adr` vlastně `push %eip`, `jmp adr`

`ret` vlastně `pop %eip`

`leave` vlastně `mov %ebp, %esp` a `pop %ebp`

Lokální proměnné ve funkci – příklad implementace

```
push %ebp       ; Ulozime hodnotu EBP do zasobniku
mov %esp, %ebp  ; Zkopirujeme hodnotu registru ESP to EBP
sub $12, %esp   ; Snizime ukazatel zasobniku o 3x4 bajty
```

První proměnná bude na adrese `-4(%ebp)`, druha `-8(%ebp)`

První parametr bude na adrese `8(%ebp)`, další `12(%ebp)`

```
mov %ebp, %esp  ; Vratime ukazatel zpet na puvodni pozici.
pop %ebp        ; Obnovime puvodni hodnotu registru EBP
ret             ; Navrat z funkce
```

Instrukce – x86/AMD64

Složitost assembleru

- Algoritmus se dá přeložit různými způsoby do assembleru
- Strojový překlad je někdy hodně kostrbatý
  - např. `mov 0x12345, %esi; mov %esi, %ebx` místo `mov 0x12345, %ebx`
- Různé způsoby pracují různě rychle a jsou rozdílně dlouhé a rozdílně přehledné
  - `xor %ebx, %ebx` je to samé jako `mov $0, %ebx`
  - `lea` adresa, registr – load effective address – nastaví hodnotu ukazatele do zadaného registru
  - `lea -12(%esp), %esp` je to samé jako `sub $12, %esp`
  - lea je výhodnější vzhledem k předzpracování instrukcí, nezatěžuje ALU jednotku (ovšem třeba Atom má zpracování adr. pomalejší než ALU).

Obsah

FPU coprocesor – x87

Speciální součást procesoru pro práci s reálnými čísly

- Podporuje single-32, double-64, extended-80 i exotické formáty BCD
- Obsahuje vlastních 8 registrů o 80 bitech
- Registry jsou organizovány v zásobníku (push, pop), ale umožňují i přímý přístup (0-7)
- Každá operace pracuje s vrcholem zásobníku a jedním dalším registrem, nebo hodnotou
- Původně oddělený procesor, od 486 on-die – na jednom čipu
- Podporuje všechny IEEE-754 operace:
  - fadd, fsub, fmul, fdiv, fsqrt, fcmp, fsin, ...

Způsob práce FPU

Základní operace slouží pro uložení reálného čísla z/do registrů:

- fld - uloží hodnotu z paměti na zásobník registrů – push
- fst - uloží hodnotu z registru do paměti bez pop
- fstp - uloží hodnotu z registru do paměti a udělá pop

Základní operace pro uložení celého čísla (integer) z/do registrů:

- fild - uloží celé číslo z paměti na zásobník registrů – push
- fist - uloží celé číslo z registru do paměti bez pop
- fistp - uloží celé číclo z registru do paměti a udělá pop
- fisttp - uloží zaokrouhlené celé číclo z registru do paměti a udělá pop

Operace FPU

Práci základních operací si ukážeme na sčítání (ostatní oprace mají shodný tvar, ST(0) je vrchol zásobníku, ST(1) hodnota pod ním, atd.):

- fadd float/double - přičti obsah paměti k ST(0) a výsledek ulož do ST(0)
- fiadd short/int - přičti celé číslo z paměti k ST(0) a výsledek ulož do ST(0)
- fadd ST(0), ST(i) - sečti obsah ST(0) a ST(i) a výsledek ulož do ST(0)
- fadd ST(i), ST(0) - sečti obsah ST(i) a ST(0) a výsledek ulož do ST(i)
- faddp ST(i), ST(0) - sečti obsah ST(i) a ST(0) a výsledek ulož do ST(i) a udělej pop operaci (zruš hodnotu ST(0))
- faddp - sečti obsah ST(1) a ST(0) a výsledek ulož do ST(1) a udělej pop operaci (zruš hodnotu ST(0))

Operace FPU

Operace SUB a DIV mají navíc i reverzní formu, tedy otočení pořadí operandů (ve všech verzích, jak s pamětí, tak s registry):

- fsub ST(0), ST(i) - výsledek ST(0) - ST(i) ulož do ST(0)
- fsubr ST(0), ST(i) - výsledek ST(i) - ST(0) ulož do ST(0)

Unární funkce sin, cos:

- fsin/fcos - ST(0) nahraď hodnotou sin/cos(ST(0))

Logaritmus - výpočet $y \cdot \log_2 x$:

- fyl2x - ST(1) nahraď hodnotou ST(1)*(log_2 ST(0)) a udělej pop

Načtení konstant:

- fldz/fld1 - uloží 0.0/1.0 na zásobník
- fldpi/fldl2e - uloží $\pi/\log_2 e$ na zásobník

Příklad programu s FPU

Příklad výpočtu 1.1*2.2+sin(3.3):

```
fldl    adr_1.1   ; Nacteme prvni operand
fmull   adr_2.2   ; Vynasobime prvni op druhym op
fldl    adr_3.3   ; Nacteme treti operand
fsinl             ; Spocti sin tretiho operandu
faddp             ; Secti dve cisla a ponech jen soucet
fstp    adr_vysl  ; Uloz vysledek do pameti
```

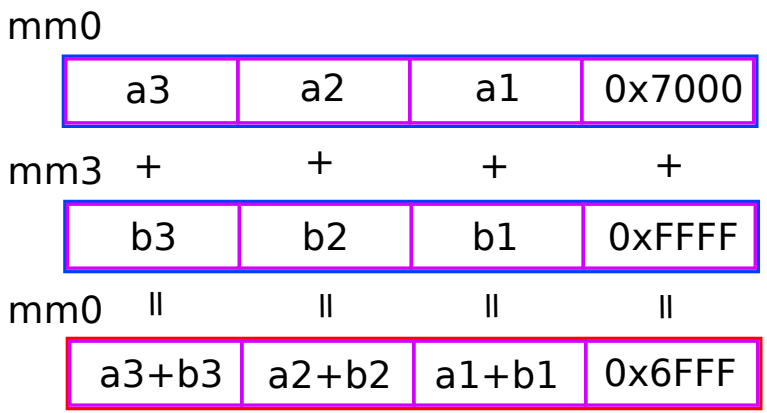
Obsah

SIMD - MMX

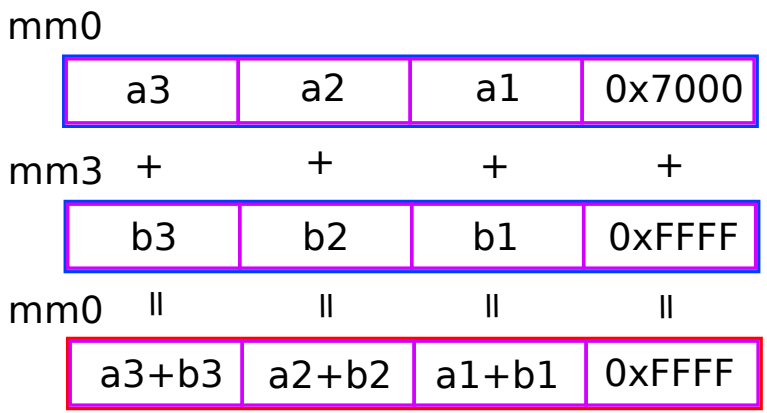
- SIMD - Single Instruction Multiple Data - provedení jednoho typu instrukce na více datech najednou
- MMX - MultiMedia eXtension (někdy se vysvětluje jako Multiple Math eXtension)
- Využívají stejné registry jako FPU x87, nelze je tedy používat současně
- 64-bitový registr může fungovat v následujících módech:
  - B - 8× bajt
  - W - 4× short int
  - D - 2× int
- Operace:
  - Aritmetické - sčítání, odčítání, násobení
  - Logické - and, or, rotace, porovnání
  - Konverze - pack, přesuny mezi registry

MMX operace

`PADDW` - součet po částech (packed)

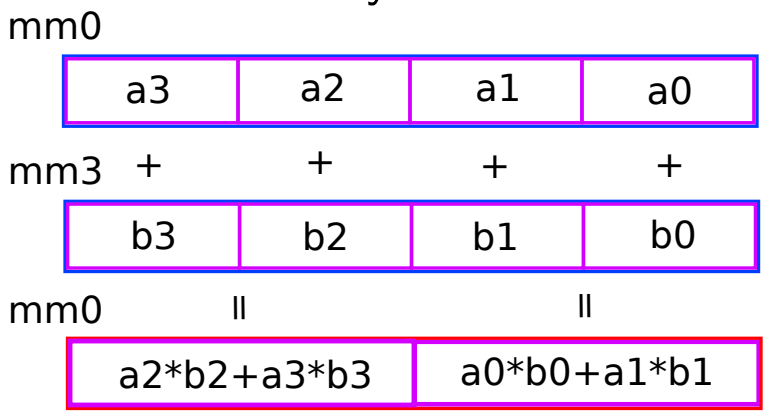


`PADDUSW` - součet saturovaný (bez přetečení)

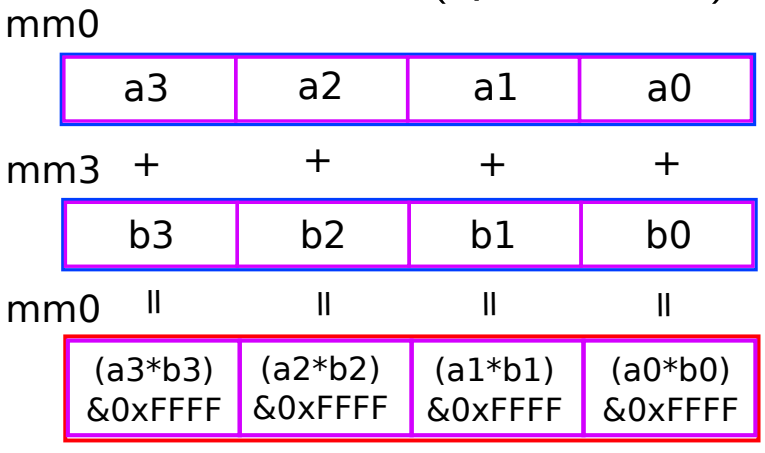


MMX operace

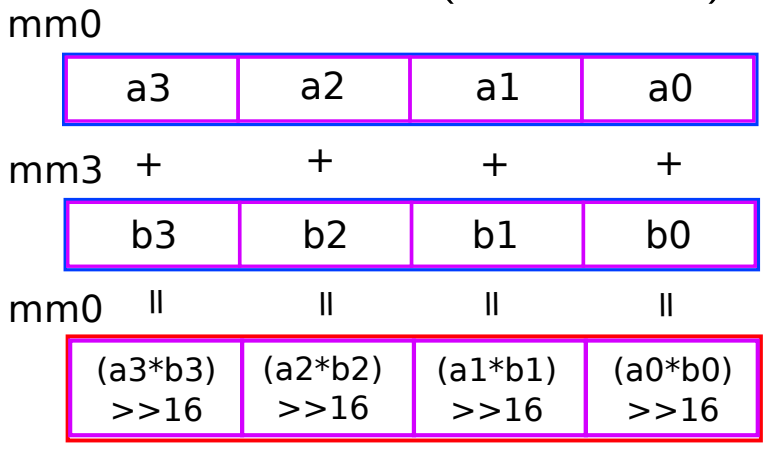
PMADDWD - vynásob a sečti



PMULLW - součin (spodní část)



PMULHW - součin (vrchní část)



MMX příklad

Maskování obrazu v obraze:

```
unsigned char mask[size],
 obr1[size ],  obr2[size ];
if  (mask[i]==0) {
   new_img[i] = obr1[i];
} else  {
   new_img[i] = obr2[i];
}
```

MMX implementace 8 pixelů najednou

```
movq    mask_ptr, %mm0
pcmpeqb %mm0, 0
movq    %mm0, %mm1
pand    %mm1, obr1_ptr
pandn   %mm0, obr2_ptr
por     %mm0, %mm1
movq    %mm0, new_img_ptr
```

3Dnow! rozšíření MMX

- Rozšíření 3Dnow! přidalo ve stávajících registrech `mm0-mm7` práci s reálnými čísly.
- Umožňuje pouze dělení registru na dvě reálná čísla po 32bitech
- Přidává konverzi celých čísel na reálná čísla a zpět, také pomocí průměrování 8-bitových a 16-bitových celých čísel
- Sčítání, odčítání, násobení, dělení reálných čísel po složkách
- Porovnávání reálných čísel a nalezení minim a maxim

Obsah

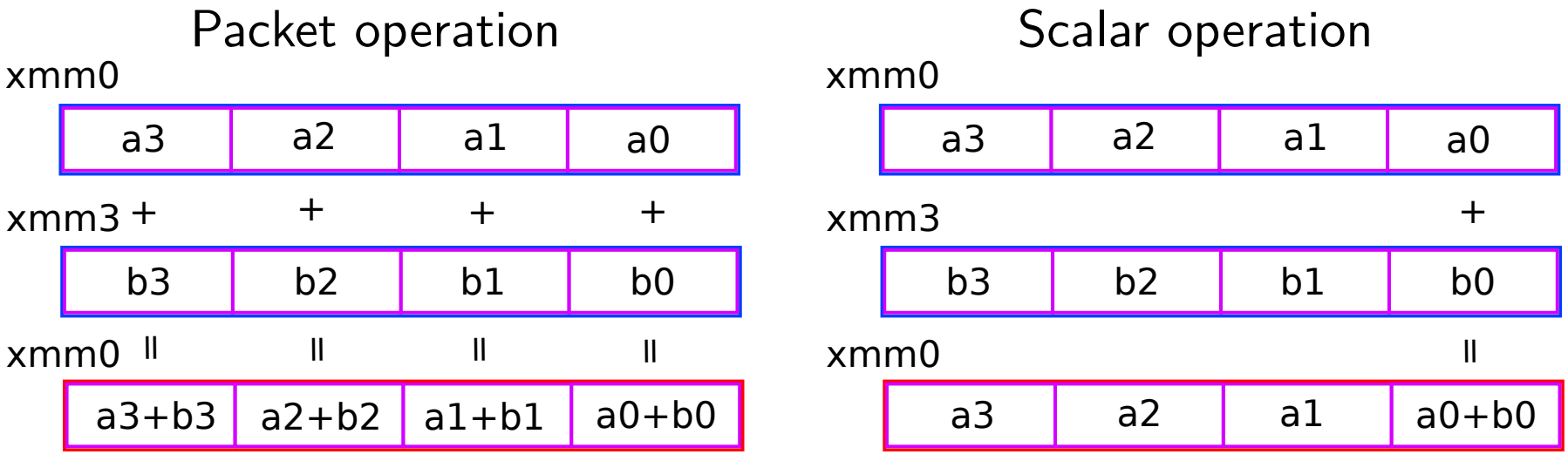
SSE další SIMD

- SSE - Streaming SIMD Extension
- nové registry xmm0-xmm7
- každý registr 128-bitů, možné dělení
- 4× float - 32-bitové reálné číslo
- 2× double - 64-bitové reálné číslo
- rozšíření celočíselných operací MMX na 128-bitové registry

Instrukce SSE

- Operace: packet suffix -ps, scalar suffix -ss
- Uložit z/do paměti: mov
- Aritmetické operace float: add, sub, mul, div, rcp, sqrt, max, min, rsqrt
- Logické operace: and, or, xor, andn
- Porovnání: cmp, comi, ucomi
- Scalar operation: addss, subss, mulss, divss

Packet SSE


Packet operation
xmm0
a3
a2
a1
a0
xmm3
+
+
+
+
b3
b2
b1
b0
xmm0
=
=
=
=
a3+b3
a2+b2
a1+b1
a0+b0
Scalar operation
xmm0
a3
a2
a1
a0
xmm3
+
b3
b2
b1
b0
xmm0
=
a3
a2
a1
a0+b0

Rozšíření SSE

Další rozšíření:

- SSE2 - přidala dalších 144 nových instrukcí
- SSE3 - dalších 13 instrukcí
- SSSE3 - dalších 16 instrukcí
- SSE4 - dalších 47 nových instrukcí
- SSE4.2 - dalších 170 nových instrukcí