

B35APO: Architektury počítačů

Lekce 11. Architektura x86

Petr Štěpán

stepan@fel.cvut.cz



30. srpna, 2024

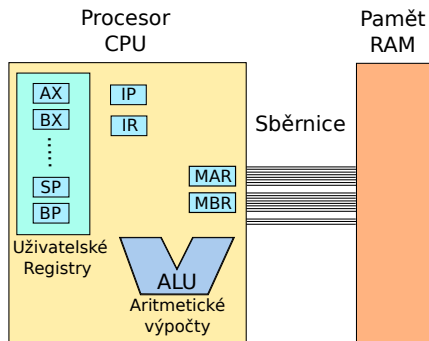
Obsah

- 1 Historie x86
- 2 Instrukce x86
- 3 FPU - x87
- 4 Rozšíření MMX
- 5 Rozšíření SSE

Processor – 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 – R16 A20 (1978) první IBM PC (8088 - 1979)

80286 – R16 A24 (1982) protected mode

80386 – R32 A32 (1985)
stránkování

80486 – R32 A32 (1989)
pipelining, FPU, cache

80586 – R32 A32 (1993) Pentium superscalar

- Přehledný popis –

https://en.wikibooks.org/wiki/X86_Assembly

80686 – R32 A36 (1995) Pentium Pro PAE, L2 cache, out-of-order & speculative exec

IA-64 – R64 A52 (2001) Itanium 64-bitová verze

AMD64 – R64 A40 (2003)

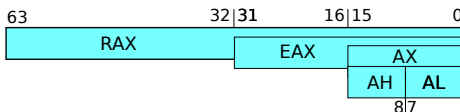
Athlonn 64-bitová verze od AMD

Core2 – R64 A36 (2006) Intel 64 EM64T, SSSE3, μ op, virtualization

Registry – 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

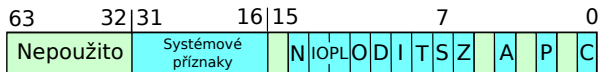


Řídící a stavové registry

- `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 – Interrupt 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
 - CPL0¹ = 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

¹Current privilege level

Obsah

- 1 Historie x86
- 2 Instrukce x86**
- 3 FPU - x87
- 4 Rozšíření MMX
- 5 Rozšíření SSE

Instrukce – x86/AMD64

Instrukce “ulož hodnotu”

(běžně se používají dvě různé syntaxe pro zápis assembleru)

AT&T

movq zdroj 64b, cíl

movl zdroj 32b, cíl

movw zdroj 16b, cíl

movb zdroj 8b, cíl

registry se značí

%ax

hodnoty \$, hex 0x

Intel

mov cíl, zdroj

pouze ax

číslo, hex postfix h

movl \$0xff, %ebx mov ebx, 0ffh

Pokud je zdroj menší než cíl, existují dvě verze:

- movsX – sign extension, na nejvyšší bity zopakuj znaménko
- movzX – zero extension, nejvyšší bity vynuluj

Instrukce – x86/AMD64

Načti hodnotu z adresy (odkaz do paměti)

AT&T

```
movl (%ecx),%eax
```

```
movl 3(%ebx), %eax
```

```
movl (%ebx, %ecx, 0x2), %eax
```

```
movl -0x20(%ebx, %ecx, 0x4), %eax
```

Intel

```
mov eax, [ecx]
```

```
mov eax, [ebx+3]
```

```
mov eax, [ebx+ecx*2h]
```

```
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 začátek pole  
mov    $128, %ecx    ; Nastav počet opakování  
mov    $-1, %eax     ; Nastav hodnotu pro uložení  
rep    stosd         ; Vyplň celé 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 začátek pole  
mov    $128, %ecx   ; Nastav počet opakování  
mov    $0, %eax     ; Nastav hodnotu konce řetězce  
repne scasb        ; Projdi str a najdi hodnotu 0
```

Instrukce – x86/AMD64

Aritmetika – AT&T syntax

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

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

andX co , k čemu bitový and

orX co , k čemu bitový or

xorX co , k čemu bitový xor (nejrychlejší vynulování registru)

mulX čím násobení eax číslem bez znamének

divX čím dělení edx:eax číslem bez znamének

imulX čím násobení eax číslem se znaménky

idivX čím dělení edx:eax číslem se znaménky

Instrukce – x86/AMD64

Aritmetika s jedním operandem – AT&T syntax

operace s cílem

incl %eax $eax = eax + 1$ decw (%ebx) $mem(ebx) = mem(ebx) - 1$ shlb \$3, %al $al = al \ll 3$ shrb \$1, %bl $bl = 11000000$, po $bl = 01100000$ sarb \$1, %bl $bl = 11000000$, po $bl = 11100000$

rorX, rclX bitová rotace doprava a doleva

rcrX, rclX bitová rotace přes 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

Quiz

Liší se velikost programů pro CISC (x86) a RISC (RISC V)?

- A neliší, je přibližně stejná
- B programy pro CISC jsou delší
- C programy pro RISC jsou delší

Porovnání CISC vs. RISC

CISC programy jsou většinou kratší:

```
incl 10(%ecx)  -      lw  t2, 10(t1)
                  addi t2, t2, 1
                  sw  t2, 10(t1)

rep movs      - l1:  lw  t3, 0(t1)
                  sw  t3, 0(t2)
                  addi t1, t1, 4
                  addi t2, t2, 4
                  addi t4, t4, -1
                  jne  t4, zero, l1
```

Quiz

Jsou programy pro CISC (x86) rychlejší než pro RISC (RISC V)?

- A Ano
- B Ne
- C Jak kdy, záleží na procesoru

Zásobník

Zásobník:

- datová struktura LIFO

Implementace:

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

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

- operace push vloží data do zásobníku
 - `sub $size, sp`
 - `mov %eax, 0(sp)`
- operace pop vybere data ze zásobníku
 - `mov 0(sp), %eax`
 - `add $size, sp`
- lze k zásobníku přistupovat podobně, jako v RISC-V:
 - `sub $16, sp`
 - `mov %eax, 0(sp)`
 - `mov %ebx, 4(sp)`
 - `mov %ecx, 8(sp)`
 - `mov %edx, 12(sp)`

Kvíz

Který program provede superskalární architektura rychleji:

A

```
push %eax  
push %ebx  
push %ecx  
push %edx
```

B

```
sub $16, sp  
mov %eax, 0(sp)  
mov %ebx, 4(sp)  
mov %ecx, 8(sp)  
mov %edx, 12(sp)
```

- A Oba přibližně stejně
- B A rychleji než B
- C B rychleji než A
- D Nelze určit

Volání funkce – x86 cdecl

- Volající uloží na zásobník všechny parametry
- Pořadí ukládání na zásobník je od posledního parametru k prvnímu

Volání funkce

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

Návrat z funkce

```
ret    vlastně pop %eip
```

- Návratová hodnota funkce bude uložena v registru eax.
- Registry ebp, ebx, esi, edi nesmí funkce změnit. Pokud je chce funkce využít, musí být uložena původní hodnota na zásobník.

Volání funkce – x86 – rámec funkce

Registr ebp ukazuje na zásobník, kam byla uložena stará hodnota rámce předchozí funkce

Úvod funkce – příklad implementace

```
push %ebp          ; Uložíme hodnotu EBP do zásobníku
mov  %esp, %ebp   ; Nastav EBP na současnou hodnotu zásobníku
sub  $12, %esp    ; Připrav místo na zásobníku pro 12 bajtů
                    ; lokálních proměnných
```

První proměnná bude na adrese $-4(\%ebp)$, druhá $-8(\%ebp)$

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

Ukončení funkce:

```
mov  %ebp, %esp   ; Vrať stav zásobníku na původní pozici.
pop  %ebp         ; Obnov původní hodnotu registru EBP
ret              ; Návrat z funkce
```

Speciální instrukce pro návrat z funkce: leave vlastně:

```
mov  %ebp, %esp
pop  %ebp
```

Volání funkce – x86 – příklad

Volání funkce:

```
t = secti(1, 2, 3, 4);
```

Překlad do x86

```
push $4
push $3
push $2
push $1
call 10e2 <secti>
add $0x10, %esp
```

Začátek funkce

```
push %ebp
mov %esp, %ebp
push %ebx
sub $0x10, %esp
```

Lokální proměnné

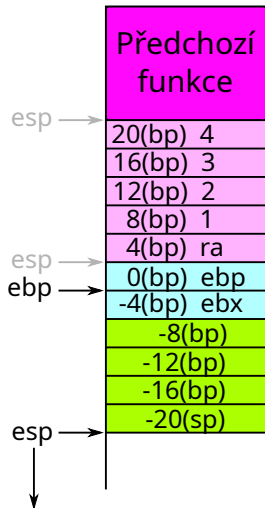
```
mov %eax, -4(%ebp)
```

První a druhý parametr:

```
mov 0x8(%ebp), %edx
mov 0xc(%ebp), %eax
```

Zakončení funkce:

```
leave
ret
```



Volání funkce – AMD64 Linux

- Volající uloží do registrů rdi, rsi, rdx, rcx, r8, r9, zmm0-7 prvních 6 parametrů celočíselných, nebo 8 parametrů reálných čísel
- Ostatní parametry na zásobník

- Návratová hodnota funkce bude uložena v registru rax a rdx.
- Registry rbp, rbx, r12-r15 nesmí funkce změnit. Pokud je chce funkce využít musí být uložena původní hodnota na zásobník.
- Red Zone – zóna 128 bajtů od pointeru rsp, kterou nesmí měnit obsluha přerušení. Tato zóna umožňuje využívat tuto paměť pro dočasné proměnné bez posunování rsp ukazatele. Samozřejmě volání funkce tuto zónu mění.

Funkce zásobníku

Zásobník:

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

Quiz

Mějme program:

```

int factorial (int x) {
    int y;
    if (x<=1) {
        return 1;
    } else {
        y = factorial (x-1)
        return y*x;
    }
}

int i;
int main() {
    i=10;
    i=factorial (i);
    return i;
}

```

Kde se bude nacházet proměnná i?

- A Pro RISC V i pro x86 na zásobníku.
- B Pro RISC V dynamicky alokovaná, pro x86 na zásobníku.
- C Pro RISC V i pro x86 na místě v paměti určeném při překladu.
- D Pro RISC V i pro x86 pouze v registru.

Quiz

Mějme program:

```

int factorial (int x) {
    int y;
    if (x<=1) {
        return 1;
    } else {
        y = factorial (x-1)
        return y*x;
    }
}

int i;
int main() {
    i=10;
    i=factorial (i);
    return i;
}

```

Kde se bude nacházet proměnná y při překladu bez optimalizace?

- A Pro RISC V i pro x86 na zásobníku nebo v registru
- B Pro RISC V i pro x86 dynamicky alokovaná
- C Pro RISC V i pro x86 na místě v paměti určeném při překladu
- D Pro RISC V i pro x86 nelze určit

Quiz

Mějme program:

```

int factorial (int x) {
    int y;
    if (x<=1) {
        return 1;
    } else {
        y = factorial (x-1)
        return y*x;
    }
}

int i;
int main() {
    i=10;
    i=factorial (i);
    return i;
}

```

Kde se bude nacházet proměnná x při překladu bez optimalizace?

- A Pro RISC V i pro x86 na zásobníku
- B Pro RISC V v registru, pro x86 na zásobníku
- C Pro RISC V v registru a následně bude uložena na zásobník, pro x86 na zásobníku
- D Pro RISC V na zásobníku, pro x86 v registru
- E Pro RISC V na zásobníku, pro x86 v registru a následně na zásobníku

Quiz

Mějme program:

```

int factorial (int x) {
    int y;
    if (x<=1) {
        return 1;
    } else {
        y = factorial (x-1)
        return y*x;
    }
}

int i;
int main() {
    i=10;
    i=factorial (i);
    return i;
}

```

Kde se bude nacházet návratová adresa z funkce factorial?

- A Pro RISC V i pro x86 na zásobníku
- B Pro RISC V v registru, pro x86 na zásobníku
- C Pro RISC V v registru a následně bude uložena na zásobník, pro x86 na zásobníku
- D Pro RISC V na zásobníku, pro x86 v registru
- E Pro RISC V na zásobníku, pro x86 v registru a následně na zásobníku

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

- 1 Historie x86
- 2 Instrukce x86
- 3 FPU - x87**
- 4 Rozšíření MMX
- 5 Rozšíření SSE

FPU coprocessor – 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 číslo z registru do paměti a udělá pop
- fisttp - uloží zaokrouhlené celé číslo z registru do paměti a udělá pop

Operace FPU

Práci základních operací si ukážeme na sčítání (ostatní operace 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) nahrad' hodnotou $\sin/\cos(ST(0))$

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

- fyl2x - ST(1) nahrad' 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
```

FPU pro RISC V

- Dvě rozšíření RV64F – float, RV64D – double
- 32 interních registrů, buď o velikosti 32 bitů pro float, nebo o velikosti 64 bitů pro double
- Nové instrukce pro načtení load a store – flw, fsw (fld, fsd)
- Nové instrukce pro operace:
 - fadd.s, fsub.s, fmul.s, fdiv.s (*.d pro double)
 - fadd.s $F[rd]=F[rs1]+F[rs2]$
 - fsqrt.s – square root – odmocnina $F[rd] = \text{sqrt}(F[rs1])$
 - fmadd.s – násob a sečti, $F[rd]=F[rs1]*F[rs2]+F[rs3]$
 - fmsub.s – násob a odečti, $F[rd]=F[rs1]*F[rs2]-F[rs3]$
 - fmin.s – $F[rd] = (F[rs1]<F[rs2]) ? F[rs1] : F[rs2]$
 - převody mezi float, double a celými čísly

Obsah

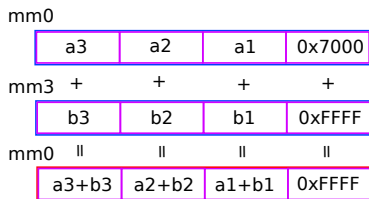
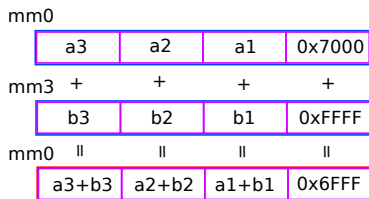
- 1 Historie x86
- 2 Instrukce x86
- 3 FPU - x87
- 4 Rozšíření MMX**
- 5 Rozšíření SSE

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 - $8 \times$ bajt
 - W - $4 \times$ short int
 - D - $2 \times$ 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

mm0

a3	a2	a1	a0
----	----	----	----

mm3

+ + + +

b3	b2	b1	b0
----	----	----	----

mm0

||

||

$a2*b2+a3*b3$	$a0*b0+a1*b1$
---------------	---------------

PMULLW - součin (spodní část)

mm0

a3	a2	a1	a0
----	----	----	----

mm3

+ + + +

b3	b2	b1	b0
----	----	----	----

mm0

||

||

||

||

$(a3*b3)$ &0xFFFF	$(a2*b2)$ &0xFFFF	$(a1*b1)$ &0xFFFF	$(a0*b0)$ &0xFFFF
----------------------	----------------------	----------------------	----------------------

PMULHW - součin (vrchní část)

mm0

a3	a2	a1	a0
----	----	----	----

mm3

+ + + +

b3	b2	b1	b0
----	----	----	----

mm0

||

||

||

||

$(a3*b3)$ >>16	$(a2*b2)$ >>16	$(a1*b1)$ >>16	$(a0*b0)$ >>16
-------------------	-------------------	-------------------	-------------------

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

- 1 Historie x86
- 2 Instrukce x86
- 3 FPU - x87
- 4 Rozšíření MMX
- 5 Rozšíření SSE**

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