Pokročilé asemblery

Cvičení 3 – Úvod do optimalizace kódu, 2017

Pár slov úvodem

- Cílem optimalizace kódu je zlepšení některého z aspektů (nejčastěji paměťové náročnosti nebo výpočetní náročnosti).

- Optimalizace kódu pro CPU může probíhat na určité úrovni abstrakce:
  - **Nejvyšší**: použití knihoven, specializovaných překladačů
  - **Nejnižší**: psaní kód v asembleru, použití SIMD instrukcí

- Další možnosti optimalizace:
  - Grafické karty: CUDA, OpenGL,….

Výkonnost počítačů

Intel Core i5 2410m	teoreticky	10 GFlops (GPU 200 GFlops)
Iphone 6S (Soc)	teoreticky	115.2 Gflops
Adreno 530 MSM8996	teoreticky	500 GFlops
Anselm (Ostrava)	teoreticky	94 000 GFlops
Salomon (Ostrava)	teoreticky	2 000 000 GFlops
National Super Computer Center in Guangzhou	teoreticky	54 000 000 GFlops

Základy optimalizace kódu pomocí ASM

- Primárně pracovat s registry a omezit přístupy do paměti.

- Snažte se pracovat s celými registry (např. 32 bitový režim – 32bitové registry).

```
mov eax, [mem1]
imul eax, 6
mov [mem2], eax
mov ax, [mem3]
add ax, 2
mov [mem4], ax
```

- Vyhýbejte se závislostem mezi sekvencemi kódu (i když to moderní procesory už umí vyřešit).

```
mov eax, [mem1]
sub eax, 6
mov [mem2], eax
mov eax, [mem3]
add ax, 2
…..
```

Základy optimalizace kódu pomocí ASM

- Bitové operace místo podmíněných skoků (v C ternární operátor) (příklady pro nalezení minima)
- Bez znaménka

```
sub eax, ebx ; = a-b
sbb edx, edx ; = (b > a) ? 0xFFFFFFFF : 0
and edx, eax ; = (b > a) ? a-b : 0
add ebx, edx ; Result is in ebx
```

- Se znaménkem

```
sub eax, ebx ; Will not work if overflow here
cdq ; = (b > a) ? 0xFFFFFFFF : 0
and edx, eax ; = (b > a) ? a-b : 0
add ebx, edx ; Result is in ebx
```

Základy optimalizace kódu pomocí ASM

- Pokud je to možné, je dobré se vyvarovat řetězci závislosti, příkladem může být sečtení pole o 1000 prvcích.

- Použití podmíněného přesunu může urychlit kód (instrukce **CMOVcc).**

Tabulka přibližné latence jednotlivých instrukcí (hodně závisí na mikro architektuře):

Instruction	Typical latency	Typical reciprocal throughput
Integer move	1	0.33-0.5
Integer addition	1	0.33-0.5
Integer Boolean	1	0.33-1
Integer shift	1	0.33-1
Integer multiplication	3-10	1-2
Integer division	20-80	20-40
Floating point addition	3-6	1
Floating point multiplication	4-8	1-2
Floating point division	20-45	20-45
Integer vector addition (XMM)	1-2	0.5-2
Integer vector multiplication (XMM)	3-7	1-2
Floating point vector addition (XMM)	3-5	1-2
Floating point vector multiplication (XMM)	4-7	1-4
Floating point vector division (XMM)	20-60	20-60
Memory read (cached)	3-4	0.5-1
Memory write (cached)	3-4	1
Jump or call	0	1-2

Table 9.1. Typical instruction latencies and throughputs

Source: Optimizing subroutines in assembly language An optimization guide for x86 platforms

Optimalizace pomocí MMX

- MMX – přidává k instrukční sadě 57 instrukcí (by Intel 1996)
- Syntax instrukce:

  paddw – instrukce pro součet sbalených dat se specifickým operandem

  add – operace
  w – velikost jednotlivých dat v registru (word 16bit -> 4x do 64bitového registru MMX)

- K nahrání dat z registru nebo paměti slouží instrukce **movd, movq**
- **MMX registry jsou splonečné s registry st0-st7**
- MMX instrukce zřejmě zaniknou ☺

MMX instrukce

paddb	PADDB	add packed byte integers
paddd	PADDD	add packed doubleword integers
paddsb	PADDSB	add packed signed byte integers with signed saturation
paddsw	PADDSW	add packed signed word integers with signed saturation
paddusb	PADDUSB	add packed unsigned byte integers with unsigned saturation
paddusw	PADDUSW	add packed unsigned word integers with unsigned saturation
paddw	PADDW	add packed word integers
pmaddwd	PMADDWD	multiply and add packed word integers
pmulhw	PMULHW	multiply packed signed word integers and store high result
pmullw	PMULLW	multiply packed signed word integers and store low result
psubb	PSUBB	subtract packed byte integers
psubd	PSUBD	subtract packed doubleword integers
psubsb	PSUBSB	subtract packed signed byte integers with signed saturation
psubsw	PSUBSW	subtract packed signed word integers with signed saturation
psubusb	PSUBUSB	subtract packed unsigned byte integers with unsigned saturation
psubusw	PSUBUSW	subtract packed unsigned word integers with unsigned saturation
psubw	PSUBW	subtract packed word integers

MMX instrukce

pcmpeqb	PCMPEQB	compare packed bytes for equal
pcmpeqd	PCMPEQD	compare packed doublewords for equal
pcmpeqw	PCMPEQW	compare packed words for equal
pcmpgtb	PCMPGTB	compare packed signed byte integers for greater than
pcmpgtd	PCMPGTD	compare packed signed doubleword integers for greater than
pcmpgtw	PCMPGTW	compare packed signed word integers for greater than

Source: https://docs.oracle.com/cd/E18752_01/html/817-5477/eojdc.html

Ušetření práce s koprocesorem

- Načtení floating point do st0:

```
movq mm0,[esi]
movq [edi],mm0

//In 64-bit mode, use:
mov rax,[rsi]
mov [rdi],rax
```

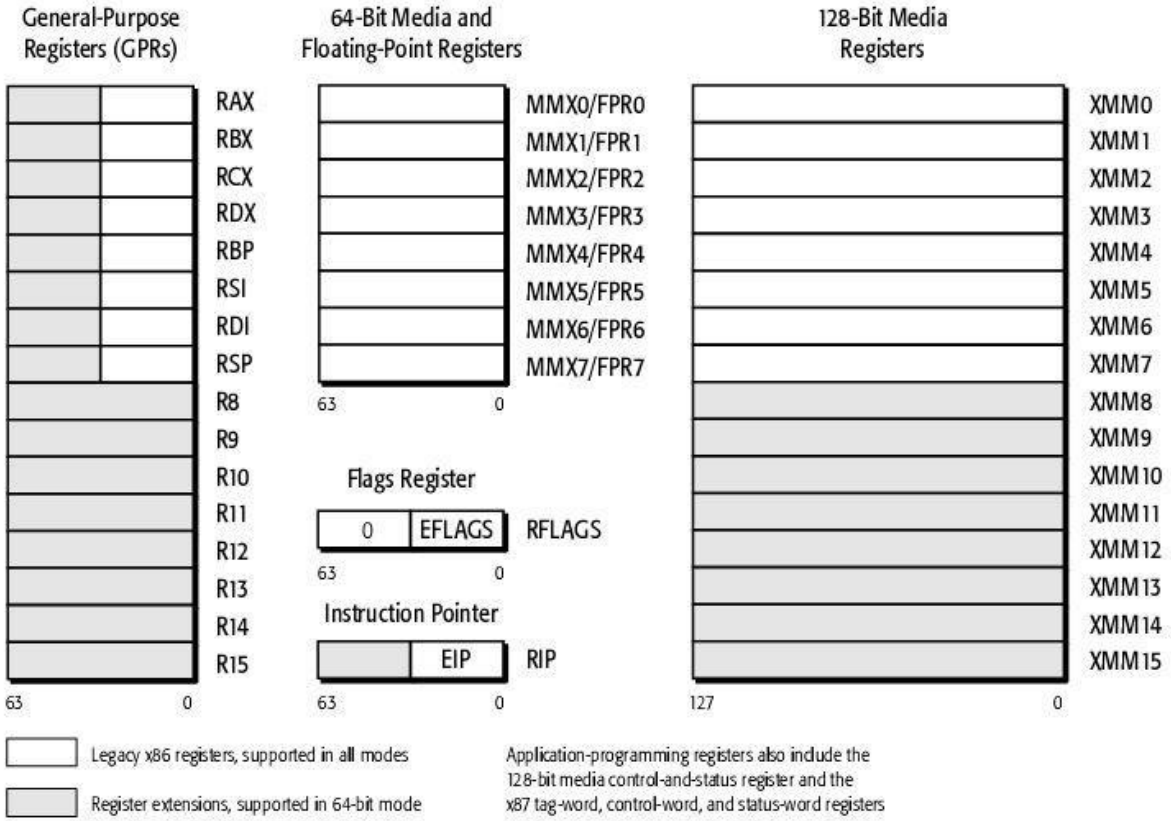
- Porovnání 2 float hodnot

```
mov eax, [a]
mov ebx, [b]
cmp eax, ebx
jb ASmallerThanB
```

Source: Optimizing subroutines in assembly language An optimization guide for x86 platforms

64 bitový režim

- Konvence volání ABI – první 4 parametry jsou v RCX, RDX, R8, R9, další pak na zásobníku
- Argumenty typu *float a double* jsou v registrech XMM



The registers RAX, RCX, RDX, R8, R9, R10, R11 are considered volatile (caller-saved).

The registers RBX, RBP, RDI, RSI, RSP, R12, R13, R14, and R15 are considered nonvolatile (callee-saved).

Úkoly

- Implementujte algoritmus pro sečtení 5 000 číslic v poli pomocí instrukční sady x86.
- Implementujte algoritmus pro sečtení 5 000 číslic v poli pomocí instrukční sady x86 a MMX. Vymyslete takové řešení, aby bylo nejrychlejší.

  1 bod

- Snižte/Zvyšte jas obrázku o libovolnou hodnotu pomocí MMX.

  1 bod

Source: Optimizing subroutines in assembly language An optimization guide for x86 platforms

Projekty

- Dotazy