Zbiór zadań na egzamin z Architektury Komputerów

Emilian Zawrotny

Random thoughts

Szybkie enkdowanie/dekodowanie wykładnika floata

Często dużo szybciej liczy się efektywny wykładnik poprzez obliczenie wartości bitów z wyjątkiem najstarszego, a potem dodanie do tego 1. Analogicznie z liczeniem binarnej reprezentacji wykładnika, wystarczy odjąć od efektywnego 1 i zapalić najstarszy bit.

Przykłady:

- **0x43800000** = 0 | 10000111 | 00... Szybciej policzyć **1+2+4+1**, niż liczyć całkowitą wartość wykładnika i odejmować od tego bias
- Wykładnik = 15: 15 - 1 = 14 = 0b1110, stąd binarna reprezentaja: 0b10001110

Wrzucanie wyrażeń do FPU

Nie wiem dlaczego nigdy dr Dziubich o tym nie wspomniał, ale wszelakie obliczenia wykonywane w koprocesorze nie różnią się absolutnie niczym, od Odwróconej Notacji Polskiej, którą to poznaliśmy podczas drugiego semestru na przedmiocie **Algorytmy i Struktury Danych** (tu kiedys pojawi sie obszerne wytłumaczenie jak konwertować wyrażenia do ONP)

Zadanka

Zad. 1

Zakładając, że zawartość rejestru xmm0 wynosi:

```
xmm0 = FF800000-EF800000-DF800000-CF800000
```

gdzie '-' wstawione są dla rozdzielenia zawartości (wartość w formacie szesnastkowym), podaj zawartość bitów 28-31 wyrażoną jako liczba szesnastkowa po wykonaniu instrukcji:

```
addss xmm0, xmm0
```

Zad. 2

Zakładając, że zawartość rejestru xmm0 wynosi:

```
xmm0 = FF800000-EF800000-DF800000-CF800000
```

gdzie '-' wstawione są dla rozdzielenia zawartości (wartość w formacie szesnastkowym), podaj zawartość bitów 60-63 wyrażoną jako liczba szesnastkowa po wykonaniu instrukcji:

```
addss xmm0, xmm0
```

Zad. 3

Zakładając, że - w programie asemblerowym 64-bitowym (kompilator MASM, system Windows) napisano prawidłową funkcję AKO - w funkcji AKO chcemy wywołać funkcję z języka C o prototypie `float testPointScale(wchar_t *, long long, void *, float, float);` - wartość rejestru RSP przed wykonaniem pierwszego rozkazu funkcji AKO była równa 000000804FAFF8C8H - na samym początku funkcji AKO zostały odłożone rejestry RBP,RSI,RDI

Podaj wartość bitów 4-7 rejestru RSP (wyrażoną jako liczbę szesnastkową) przed wykonaniem pierwszego rozkazu z funkcji testPointScale.

Rozwiązania

Zad. 1

Instrukcja `addss` dodaje do siebie **floaty** znajdujące się na najmłodszych 32 bitach podanych rejestrów. W tej sytuacji

```
addss xmm0, xmm0
```

W pierwszej kolejności musimy zdekodować naszego floata:

```
0xCF800000 = 0b 1|10011111|0000000...
```

Z binarnej reprezentacji łatwo możemy odczytać: - bit znaku = 1 (liczba jest ujemna) - wykładnik wynosi 32 - Mantysa wynosi 1

Zatem wartość floata to: $-1.0 \cdot 2^{32} = -2^{32}$. Teraz pozostaje obliczyć nam wynik działania instrukcji `addss`: $2 \cdot -(2^{32}) = -(2 \cdot 2^{32}) = -(2^{33})$ Pozostaje nam zakodować wartość $-(2^{33})$ w formacie float: - liczba ujemna $\implies$ bit znaku = 1 - mianowwykładniknik wynosi 160 (+127 bias) - Mantysa wciąż wynosi 1 (lecz jawną jedynkę się pomija)

Nasza szukana wartość: 1 | **1010**0000.... 0b1010 = **0xD**

Zad. 2

Instrukcja `addss` dodaje do siebie floaty znajdujące się na najmłodszych 32 bitach podanych rejestrów. Nie robi natomiast nic z pozostałymi bitami. Stąd wartość bitów 63-60 nie ulegnie zmianie (pozostanie równa **0xD**).

Zad. 3

```
AKO PROC
    ; W tym miejscu RSP=0x000000804FAFF8C8
    ; Bezpośrednio po wywołaniu procedury, adres NIGDY nie jest podzielny przez 16
    ; Natomiast parzysta liczba pushy powoduje spełnienie wymogu podzielności
    ; przez 16 przed wywołaniem kolejnej procedury
    push rbp    ; RSP -= 8
    push rsi    ; RSP -= 8
    push rdi    ; RSP -= 8
    ; Po wykonaniu 3 pushy, RSP jest podzielny przez 16
    mov rcx, ...
    mov rdx, ...
    mov r8, ...
    mov r9, ...
```

```
push [jakistam float]    ; RSP -= 8
sub rsp, 32 ; shadowspace RSP -= 32 (odpowiednik 4 pushy)
; W tym miejscu RSP jest niepodzielny przez 16 (na skutek parzystej liczby pushy)
sub rsp, 8  ; dopelnienie   RSP -=8
call _testPointScale     ; RSP -= 8 na skutek odłożenia adresu powrotu
```

Po zsumowaniu wszystkich pushy mamy **RSP -= 80 = 0x50**, wystarczy wziąć pod uwagę bity które nas interesują i mamy:

0xC - 0x5 = **0x8**