Wojciech Adles, 209853 Wrocław, 07.06.17r

ŚR-N-15

Laboratorium Organizacji i Architektury Komputerów

Laboratorium 5:

Jednostka zmiennoprzecinkowa (FPU)

1. Treść ćwiczenia

- Napisać program w języku C, który będzie sprawdzał i zmieniał precyzję obliczeń, korzystając z funkcji napisanych w języku Asemblera **(1,5 pkt)**
- Wykazać działanie poprzedniego programu (różnice w precyzji obliczeń) **(1 pkt)**
- Implementacja funkcji e^x w języku Asemblera, wywołanie z poziomu programu napisanego w języku C **(2,5 pkt)**

2. Przebieg ćwiczenia

2.1 Program sprawdzający i zmieniający precyzję obliczeń FPU

Pierwszym zadaniem było napisanie programu korzystającego z dwóch funkcji: jednej sprawdzającej aktualną precyzję obliczeń jednostki zmiennoprzecinkowej i drugiej zmieniającą ją. Pisanie programu zacząłem od implementacji prostego menu konsolowego w języku C, którego kod nie jest istotny dla głównego celu tego ćwiczenia. Przejdźmy zatem do funkcji napisanych w języku Asemblera, ich kod znajduję się poniżej.

```
.section .data

.section .bss

.comm control, 2

.section .text
.type checkprec @function
.type changeprec @function
.globl checkprec, changeprec

checkprec:

        push %rbp
        movq %rsp, %rbp

        fstcw control
        movq $0, %rax
        movw control, %ax
        and $0x0003, %ax


        movq %rbp, %rsp
        pop %rbp
        ret
changeprec:

        push %rbp
        movq %rsp, %rbp

        fstcw control
        movw control, %ax
        or $0x0003, %ax
        cmp $0, %rdi
        jne dbl
        xor $0x0003, %ax                #single precision
        jmp end
dbl:
        cmp $1, %rdi
        jne dblx
        xor $0x0001, %ax                #double precision
        jmp end
dblx:
        cmp $2, %rdi
        xor $0x0, %ax                   #double-ex precision
end:
        movw %ax, control
        fldcw control

        movq %rbp, %rsp
        pop %rbp
        ret
```

Listing 1: Kod programu lab5.s

W powyższym pliku zostały napisane dwie funkcje realizujące sprawdzenie jak i zmianę precyzji obliczeń FPU. Pierwsza z nich tj. *checkeprec* za pomocą instrukcji *fstcw* kopiuje słowo kontrolne do 2-bajtowego bufora *control,* a stamtąd do rejestru *%ax*. Właśnie w słowie kontrolnym znajdują się dwa bity (dokładnie 9 i 10 bit) odpowiedziane za kontrolę precyzji obliczeń. Jako, że przy ich zapisie jest

zachowany format zapisu little endan, wykonując operację logiczną AND z wartością *$0x0003* zerujemy pozostałe bity w rejestrze *%ax*, dzięki czemu pozostaje w nim liczba definiująca aktualną precyzję. Po powrocie z funkcji odczytujemy w programie zwróconą wartość i informujemy użytkownika o wyniku. Zwrócona wartość wynosi: 0 dla pojedynczej precyzji, 2 dla podwójnej precyzji i 3 dla rozszerzonej podwójnej precyzji. Jak można było zauważyć funkcja ta nie jest skomplikowana, opiera się ona de facto na jednej instrukcji *fstcw*. Przejdźmy zatem do funkcji *changeprec* manipulującej precyzją obliczeń.

Funkcja *changeprec* przyjmuję jedną wartość całkowitą, która zależy od zadanej przez użytkownika precyzji. Najpierw jednak przenosimy słowo kontrolne do rejestru *%ax*, gdzie zostaje ono poddane operacji logicznego OR, która ma za zadanie ustawić bity odpowiedzialne za zmianę precyzji na '1', nie naruszając pozostałych wartości. Następnie sprawdzamy porównujemy zawartość rejestru *%rdi* z *$0*, jeśli wartości te są różne, wykonujemy skok warunkowy do następnej sekcji programu, gdzie rejestr ten jest porównywany z kolejnymi wartościami zależnymi od oczekiwanej precyzji obliczeń. Jeśli jednak są takie same, oznacza to, że użytkownik zadeklarował chęć zmiany precyzji na pojedynczą. Przeprowadzamy zatem operację logicznego XOR naszego rejestru *%ax* z wartością *$0x00001* która zeruje odpowiednie bity w słowie kontrolnym, tak aby wskazywały na pojedynczą precyzje obliczeń. Po wykonaniu tej instrukcji, funkcja wykonuje skok bezwarunkowy do etykiety *end* gdzie zmodyfikowany rejestr *%ax* jest kopiowany do bufora *control*, który zostanie następnie podmieniony ze słowem kontrolnym, dzięki czemu zmieniona zostanie precyzja obliczeń. Funkcja kończy swoje działanie, a precyzja obliczeń została zmieniona.

2.2 Program implementujący funkcję e^x w języku Asemblera

Kolejny program do wykonania na laboratorium dotyczył napisania programu w języku C, który wywoływałby funkcję napisaną w języku Asemblera, obliczającą wartość matematycznej funkcji e^x dla zadanego przez użytkownika argumentu. Jako, że tak jak w przypadku poprzedniego programu, część napisana w języku C nie jest istotna dla celu laboratorium, przejdę od razu do omówienia kodu napisanego w Asemblerze. Znajduję się on w poniższym listingu.

```
.section .data

x:      .long 0
y:      .double 2.718281828459

.section .text

.type exfunc @function
.globl exfunc

exfunc:
        push %rbp
        movq %rsp, %rbp

        movq %rdi, x

        finit
        fildl x
        fldl y
        fyl2x
        fld1
        fld %st(1)
        fprem
        f2xm1
        faddp
        fscale
        fxch %st(1)
        fstp %st
        fstp y
        movsd y, %xmm0

        movq %rbp, %rsp
        pop %rbp
        ret
```

Listing 2: Kod programu lab5_2.s

Jak można zauważyć funkcja ta nie jest wyjątkowo obszerna, jednak „dzieje” się w niej całkiem dużo. Zacznę od faktu, że dla x87 FPU istnieje jedna instrukcja implementująca operację podniesienie liczby do zadanej potęgi, niestety podnoszona liczba musi być dwójką. Zatem aby móc obliczyć wartość funkcji e^x skorzystałem z przekształcenia matematycznego, z którego otrzymałem $e^x = 2^{xlog_2 e}$. Jako że jak napisałem wcześniej mamy instrukcję realizującą operację 2^x oraz w liście rozkazów znajduje się również instrukcja obliczająca $xlog_2 y$ taka implementacja jest możliwa. Zaczynając jednak od początku funkcji, kopiujemy zadaną wartość x otrzymaną jako argument i umieszczamy ją w zmiennej o tej samej nazwie. Następnie instrukcją *fildl* kopiujemy ją na szczyt stosu FPU, a instrukcją *fldl* kopiujemy przybliżenie stałej e na stos. W tej chwili stan naszego stosu wygląda następująco:

st(0) = e, st(1) = x

W kolejnym kroku wykonujemy instrukcję *fyl2x* realizującą operację $xlog_2 e$. Jej wynik znajdzie się na stosie w miejscu st(1), a wartość z góry stosu zostanie zdjęta, przez co po końcowy rezultat będzie wyglądał następująco:

st(0) = $xlog_2 e$

Chcąc wykonać operację potęgowania za pomocą funkcji *f2mx1*, która przyjmuje jako argumenty tylko liczy z zakresu [-1,1], musimy zmodyfikować zmienną znajdującą się na stosie, tak aby mogła być ona przetworzona przez tę funkcję. Musimy zatem wyciągnąć cześć ułamkową z $xlog_2e$ (którą będę zapisywał jako $xlog_2e \bmod 1$) i poddać ją operacji potęgowania. Następnie wynik trzeba będzie skorygować tak, aby uwzględniał pominiętą część całkowitą, co wykonamy za pomocą instrukcji *fscale* która wykonując operację $2^{\lfloor xlog_2e \rfloor}$ (warto zwrócić uwagę na znak „podłogi", oznaczający zaokrąglenie do części całkowitej poprzez „obcięcie") dla wartości $xlog_2e$ znajdującej się w st(1), jednocześnie mnożąc ją z wartością w st(0), gdzie będzie przechowywany wynik dla części ułamkowej. W praktyce będzie wyglądało to następująco. Instrukcją *fld1* „wrzucamy" na szczyt stosu wartość 1, dzięki czemu:

$$\text{st(0)} = 1, \text{st(1)} = xlog_2e$$

Kopiujemy instrukcją *fld %st(1)* wartość $xlog_2e$ na szczyt stosu, aby kolejność operandów dla operacji *fprem* była zachowana. Aktualny stan stosu:

$$\text{st(0)} = xlog_2e\ , \text{st(1)} = 1, \text{st(2)} = xlog_2e$$

Następnie wspomnianą instrukcją *fprem* dzielimy wartość w st(0) przez wartość w rejestrze st(1), zachowując resztę z tego dzielenia w st(0). Po tej instrukcji stos prezentuje się następująco:

$$\text{st(0)} = xlog_2e \bmod 1\ , \text{st(1)} = 1, \text{st(2)} = xlog_2e$$

Kolejnym krokiem jest wykonanie z pomocą otrzymanej wartości operacji potęgowania. Wykorzystujemy do tego celu instrukcję *f2xm1* która w naszym przypadku realizuje funkcję $2^{xlog_2e \bmod 1} - 1$. Jak widać wynik jest zmniejszony o 1, co skompensujemy dodając do niego jedynkę pozostałą na stosie. Stos po wykonaniu instrukcji *f2xm1*:

$$\text{st(0)} = 2^{xlog_2e \bmod 1} - 1\ , \text{st(1)} = 1, \text{st(2)} = xlog_2e$$

Następnie instrukcją *faddp* dodajemy rejestry st(0) i st(1), wynik przechowując w st(1) i następnie „zrzucając" ze stosu wartość w st(0), przez co po jej wykonaniu stos wygląda następująco:

$$\text{st(0)} = 2^{xlog_2e \bmod 1}\ , \text{st(1)} = xlog_2e$$

Teraz musimy skorygować nasz wynik o pominiętą wartość całkowitą $xlog_2e$. Jak wcześniej wspomniałem do tego celu używamy instrukcji *fscale* która idealnie odpowiada naszym potrzebą. Po jej wykonaniu stos będzie wyglądał tak:

$$\text{st(0)} = 2^{xlog_2e \bmod 1} \cdot 2^{\lfloor xlog_2e \rfloor} = 2^{xlog_2e}, \text{st(1)} = xlog_2e$$

Dzięki temu w rejestrze st(0) otrzymaliśmy pożądany przez nas wynik, zamieniamy teraz dwie pozostałe na stosie wartości miejscami i „zrzucamy" ze szczytu stosu st(0) = $xlog_2e$, dzięki czemu jedyną pozostałą wartością jest obliczona przez nas wartość $2^{xlog_2e} = e^x$. Przenosimy ją do rejestru *%xmm0* i kończymy działanie funkcji, zwracając wynik w postaci zmiennej typu float.

3. Wnioski

Na laboratorium zapoznałem się z budową i działaniem jednostki zmiennoprzecinkowej x87, wykonując dwa z trzech zadań. W szczególności implementacja funkcji e^x pokazuje jak złożona jest realizacja obliczeń zmiennoprzecinkowych w architekturze komputerów.