Marta Głowacka, 234999 Wrocław, 18.05.2018 r.

WT-N-10

LABOLATORIUM ARCHITEKTURY KOMPUTERÓW

5.Łączenie kodu języka C z asemblerem

1. Treść ćwiczenia

Zakres i program ćwiczenia:

- Jednostka zmiennoprzecinkowa

Zrealizowane zadania:

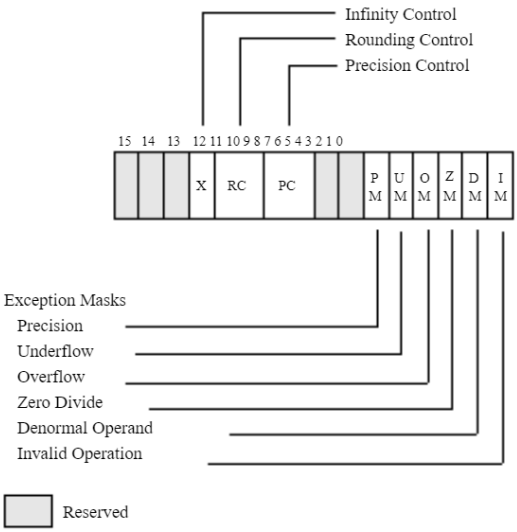
- Program w języku C: funkcja w języku asemblera pozwalająca na sprawdzenie aktualnie ustawionego trybu zaokrąglenia oraz funkcja pozwalająca na ustawienie trybu zaokrąglenia (bez zmiany innych ustawień).
- Aproksymacja funkcji e^x wg szeregu Taylora (funkcja w języku asemblera) x-zmiennoprzecinkowa, n – liczba kroków

2. Przebieg ćwiczenia

Ustawienie i zmiana trybu zaokrąglania - kod w C plus funkcja napisana w asemblerze

Wszystkie ustawienia jednostki zmiennoprzecinkowej zapisywane są w specjalnie przeznaczonym do tego celu rejestrze kontrolnym (control word). Zmieniając niektóre bity tego rejestru, możemy wpływać na ustawienia fpu.

Bity 10 oraz 9 to bity RS (roundness control) i odpowiadają za tryby zaokrąglania liczb zmiennoprzecinkowych.



Źródło: https://xem.github.io/minix86/manual/intel-x86-and-64-manual-vol1/o_7281d5ea06a5b67a-197.html

Odpowiednia sekwencja tych dwóch bitów ustawia wybrany tryb zaokrąglania.

The **RC** field (bits 11 and 10) or **R**ounding **C**ontrol determines how the FPU will round results in one of four ways:

00 = Round to nearest, or to even if equidistant (this is the initialized state)
01 = Round down (toward -infinity)
10 = Round up (toward +infinity)
11 = Truncate (toward 0)

W poniższym przykładzie zaprezentuję sprawdzenie oraz zmianę trybu zaokrąglania jednostki zmiennoprzecinkowej. Wykorzystam do tego interface napisany w języku C oraz funkcję w asemblerze.

Kod w języku C jest stosunkowo prosty i wybiera akcję w zależności od int na wejściu podanego przez użytkownika. Program ustawia lub wyświetla ustawiony tryb, korzystając z funkcji napisanej w asemblerze.

```
#include <stdio.h>

extern int sprawdzTryb();
extern int ustawTryb(int ktoryTryb);


int main(void)
{
    int wybor = 1;
    int wynik = 1;
    int wybranyTryb = 0;

    do
    {
```

```
        printf("1 - sprawdz tryb, 2 - ustaw tryb, 3 - koniec\n");
        scanf("%d", &wybor);

        switch(wybor)
        {
        case 1:
            switch(sprawdzTryb())
            {
            case 0:
            printf("Round to the nearest\n");
            break;
            case 1:
            printf("Round up - to +inf\n");
            break;
            case 2:
            printf("Round down - to -inf\n");
            break;
            case 3:
            printf("Truncate - to zero\n");
            break;

            }
        break;

        case 2:
            printf("Ustaw wybrany tryb zaokraglenia, wybierz nearest - 0, down - 1, up
- 2 lub to zero - 3\n");
            scanf("%d", &wybranyTryb);
            if(wybranyTryb < 0 || wybranyTryb > 3 ) printf("Zla wartosc\n");
            else ustawTryb(wybranyTryb);
        break;
        }


    }while(wybor != 3);
    return 0;


}
```

Poniżej kod asemlerowy.

```
.data
slowo_kontrolne: .short 0    #Tutaj zapisywane są ustawienia jednostki FPU

.text
```

Deklaracja funkcji.

```
.global ustawTryb, sprawdzTryb
.type ustawTryb, @function
.type sprawdzTryb, @function
```

Funkcja ustawiająca tryb zaokrąglenia.

```
ustawTryb:

    #Argument znajduje sie w rdi
    #0 - 00 - round to nearest
```

```
    #1 - 01 - round down
    #2 - 10 - round up
    #3 - 11 - truncate

    push %rbp        #Początkowe działania na stosie
    mov %rsp, %rbp        #Przesunięcie wskaźnika stosu
```

Pobranie zawartości rejestru kontrolnego do rejestru ax. Przy pomocy rozkazu fstcw można zapisać zawartość rejestru kontrolnego do pamięci, a następnie instrukcją mov przenieść do rejestru ax.

```
    mov $0, %rax
    fstcw slowo_kontrolne    #W rejestrze kontrolnym zapisane ustawienia dla jednostki
FPU
    fwait
    mov slowo_kontrolne, %ax
```

Nie chcemy zmieniać innych ustawień, więc zerujemy jedynie bity zaokrąglania, nakładając maskę 1111 0011 1111 1111, gdyż zera występują w niej na 9 i 10 bicie.

```
    #===Wyzerowanie bitów zaokrąglania===
    and $0xf3ff, %ax    #1111 0011 1111 1111
```

W zależności od wybranego trybu zaokrąglania, przechodzimy do odpowiedniej etykiety (czynimy to 2 razy - najpierw porównujemy wybór z jedynką (tryb 0,1 rozstrzygnięte), a później ewentualnie z 2 (rozstrzyganie trybu 2 czy 3).

```
    #===Zmiana trybu zaokraglania w zaleznosci od argumentu===
    porownanie:
    cmp $1, %rdi
    jl round_to_nearest
    jg compare_once_again

    #===Dla wartosci 01===
    round_down:
    xor $0x400, %ax        #0000 0100 0000 0000
    jmp koniec
```

W zależności od wybranego trybu, ustawiamy na bitach RS sekwencję 00,, 01, 10 lub 11. Czynimy to instrukcją xor, a nie mov, gdyż nie chcemy, aby inne ustawienia uległy zmianie.

```
    #===Dla wartosci 00===
    round_to_nearest:
    xor $0x000, %ax        #0000 0000 0000 0000
    jmp koniec

    #===Żadanie może byc 10 lub 11===
    compare_once_again:
    cmp $2, %rdi
    jg truncate

    #===Dla wartosci 10===
    round_up:        #0000 1000 0000 0000
```

```
    xor $0x800, %ax
    jmp koniec

    #===Dla wartosci 11===
    truncate:        #0000 1100 0000 0000
    xor $0xc00, %ax

    koniec:
    mov %ax, slowo_kontrolne
    fldcw slowo_kontrolne

    mov %rbp, %rsp
    pop %rbp
    ret
```

Sprawdzenie trybu zaokrąglania. Początek wygląda analogicznie do ustawiania.

```
sprawdzTryb:
    push %rbp        #Początkowe działania na stosie
    mov %rsp, %rbp        #Przesunięcie wskaźnika stosu

    #==Pobranie slowa kontrolnego==
    movq $0, %rax
    fstcw slowo_kontrolne
    fwait
    mov slowo_kontrolne, %ax

    #Bity 11 i 10 to RC Field - Rounding Control
```

Wyzerowanie bitów poza bitami kontroli precyzji i przesunięcie tych bitów na koniec.

```
    and $0xC00, %ax        #0000 1100 0000 0000
    shr $10, %ax
```

Zwracana wartość jest w rejestrze eax i przyjmuje wartość 0, 1, 2 lub 3.

Aproksymacja funkcji ex szeregiem Taylora

Wartość funkcji ex przybliżana jest ze wzoru:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \qquad \text{dla } x \in R,$$

W tym programie będziemy korzystać z rejestrów zmiennoprzecinkowych ST(0), ST(1) itd. Rejestry działają na zasadzie stosu - element umieszczany jest zawsze w rejestrze ST(0), a kolejne wartości są przepisywane: ST(0) = nowa wartość, ST(1) = ST(0) itd.

Program dodaje do siebie kolejne wyrazy ciągu, wykorzystując do tego wyraz poprzedni.

Program w C realizuje interface, wczytuje potęgę dla e oraz liczbę kroków (liczba wyrazów szeregu)

```
#include <stdio.h>

//x potęga dla e, n liczba krokow
extern double szereg_taylora(double x, int n);

int main(void)
{
    double potega;
    int kroki;

    printf("Podaj x - potęgę dla e: ");
    scanf("%lf", &potega);

    printf("Podaj liczbe wyrazow ciagu: ");
    scanf("%d", &kroki);

    printf("Wynik: %lf \n", szereg_taylora(potega, kroki));

    return 0;
}
```

Na końcu wypisuje wynik, korzystając z funkcji szereg_taylora napisanej w asemblerze.

```
.data
```

Zaczynamy od silni równej zero (wg wzoru)

```
silnia: .double 0.0

#Deklaracja funkcji
.text
.global szereg_taylora
.type szereg_taylora, @function
```

W rejestrze xmm0 znajduje się potęga dla e, a w rdi liczba kroków - n.

```
#xmm0 - x - potęga
#rdi - n -  czyli liczba krokow
```

Funkcja licząca wartość szeregu.

```
szereg_taylora:
    push %rbp        #Korzystamy z jednostki zmiennoprzecinkowej
    mov %rsp, %rbp
```

Instrukcja fld1 ładuje bit 1 do rejestru ST(0), kolejne rejestry ST są “przesuwane”.

```
#Zaladowanie wartosci z rejestru xmm0 - zmiennoprzecinkowego do ST(0), ST(1)...
    sub $8, %rsp
    movsd %xmm0, (%rsp)
    fldl (%rsp)        #Zapisujemy do ST(0) potege (ładowany jest double)
    fld1             #Przesunięcie wartości do innych ST: do ST(1) idzie ST(0), a do
ST(0) = 1
    fld1
```

```
#Po przesunięciu sytuacja wygladać będzie tak - zawsze na początku pętli:
#ST(0) - aktualny wyraz ciagu (poczatkowo 1)
#ST(1) - dotychczasowa suma (poczatkowo 1)
#ST(2) - to, przez co mnozymy licznik, czyli x
```

Inicjujemy licznik pętli - by wykonała się tyle razy, ile podał użytkownik w liczbie kroków (zaczynamy indeksować od zera, gdyż będzie to wygodniejsze z uwagi na wzór szeregu).

```
movq $0, %rsi    #Licznik pętli
fwait        #Oczekiwanie na zakonczenie wykonywanych przez fpu obliczen

dec %rdi    #Konieczne, aby liczyło od wcześniejszej wartości (zaczynamy indeksować od 0)
```

Rozpoczynamy pętlę sumującą, która najpierw porównujemy licznik z rejestrem liczby kroków, aby wiedzieć, kiedy operacje należy zakończyć.

```
petla_sumujaca:
    cmp %rdi, %rsi    #Petla sie zakonczy po obliczeniu wszystkich wyrazow rdi - liczba krokow do wykonania, a rsi - licznik petli
    je koniec
    inc %rsi    #Zwiększenie licznika pętli
```

Obliczamy licznik, mnożąc aktualny wyraz ciągu zawsze przez x.

```
    #===Obliczanie licznika: kolejno: 1, x, x^2, x^3 itd...
    fmul %st(2), %st    #Aktualny wyraz ciągu (ST(0)) przemnożony przez x (st(2)) -> wynik zapisany w st(2)
```

Aby obliczyć mianownik, najpierw musimy załadować do rejestrów ST stałą 1 oraz poprzednią silnię - na jej podstawie wyliczmy kolejną silnię (jeśli wcześniej było 2!, teraz będzie 3, które posłuży do przemnożenia 2! * 3 = 3!).

```
    #===Obliczenie mianownika===
    fld1        #...ST(2) = ST(1), ST(1) = ST(0), ST(0) = 1; ładowana stała 1, a pozostale rejestry przesuwane
    fldl silnia

    #Obliczenie kolejnego wyrazu silni, jeśli poprzedni wyraz silni był 2, to teraz bedzie 3
    #Obliczenia (poprzedni wyraz * x)/(Y+1) -> da nam to w wyniku kolejny wyraz ciągu
```

Obliczamy kolejny wyraz silni, dodając 1 do poprzedniego wyrazu silni.

```
    fadd %st(1), %st


    #Usuniecie niepotrzebnych wartosci
    fxch %st(1)    #Zamiana miejscami st 0 z st1
    fstp %st    #Ściągnięcie ze "stosu" fpu ostatniej wartosci - i wrzucenie do ST(0)

    #Aktualnie:
```

```
    #ST(0) - silnia (obecny dzielnik)
    #ST(1) - wartosc do podzielenia - aktulany wyraz ciagu
    #ST(2) - aktualna suma ciagu
    #ST(3) - przeslany kat x
```

Chcemy przeprowadzić obliczenia: (poprzedni wyraz ciągu * x)/ (wyliczony nastepny wyraz silni). Dla przykładu:

$$\frac{\frac{x^2}{2!} \times x}{3} = \frac{x^3}{3!}$$

Na koniec dodajemy wyliczony wyraz do ogólnego wyniku.

```
    fdivr %st, %st(1)    #Dzielenie obecnego wyrazu przez dzielnik (silnie) -> wynik
do st(1)
    fstpl silnia    #Zapisanie numeru ost wyrazu silni i sciagniecie go ze "stosu" fpu
            #fstpl - wartosc z rejestru ST(0) kopiowana jest do zmiennej silnia,
numeracja rejestrow zmienia się 1->0, 2->1        #Usuniecie obecengo dzielnika ze
"stosu" fpu, zawartość rejestrów jak na początku pętli
    fadd %st, %st(1)    #Dodanie wartosci obecnego wyrazu do ogólnego wyniku

jmp petla_sumujaca
```

Podsumowanie.

```
    koniec:
    fstp %st
    fstpl (%rsp)
    movsd (%rsp), %xmm0

    mov %rbp, %rsp
    pop %rbp
    ret
```

Wnioski

Aby mieć pewność, że obliczenia zmiennoprzecinkowe zdążą się wykonać, należy użyć instrukcji fwait.

Jednostka zmiennoprzecinkowa oferuje wiele możliwości dzięki zmianie ustawień w rejestrze kontrolnym. Również otwiera nam dostęp do korzystania z rejestrów ST.

Źródła:

Professional Assembly Language, Richard Blum

https://docs.oracle.com/cd/E19120-01/open.solaris/817-5477/epmnw/index.html

https://xem.github.io/minix86/manual/intel-x86-and-64-manual-vol1/o_7281d5ea06a5b67a-197.html

http://www.website.masmforum.com/tutorials/fptute/fpuchap1.htm

http://prac.im.pwr.wroc.pl/~pfrej/wyklad3_szeregi_potegowe.pdf