Wrocław, dn. 4 maja 2017

Hanna Grodzicka, 226154

Jednostka zmiennoprzecinkowa

sprawozdanie z laboratorium przedmiotu „Architektura Komputerów 2”

Rok akad. 2016/2017, kierunek: INF

Prowadzący:

mgr inż. Aleksandra Postawka

1. Cel ćwiczenia

Laboratoria wymagały korzystania z FPU (ang. *floating-point unit*), czyli jednostki zmiennoprzecinkowej zwanej też koprocesorem. Na tę jednostkę składa się 8 rejestrów 80-bitowych ułożonych w stos (*x87 FPU data registers*) oraz z rejestrów specjalnych: *status register, control register, tag word register, last instruction pointer register, last data (operand) pointer register, opcode register*.

Pierwszy program miał być napisany w języku C i powinien wywoływać dwie funkcje w języku Asemblera:

- sprawdzenie bieżących ustawień precyzji;
- zmiana bieżących ustawień precyzji.

Ponadto należało wykazać zmiany w działaniu, po ustawieniu precyzji.

Kolejne zadanie to aproksymacja funkcji cosinus z wykorzystaniem szeregu Taylora, napisane w języku Asemblera.

Rozwinięcie funkcji w szereg:

$$\cos x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} \quad = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots$$

2. Przebieg ćwiczenia

A. Program w języku C

Jednostka zmiennoprzecinkowa dysponuje trzema ustawieniami precyzji:

- *Single Precision* - 24 bity na mantysę;
- *Double Precision* - 53 bity na mantysę;
- *Double Extended Precision* - 64 bity na mantysę.

O ustawionej precyzji obliczeń (*Precision Control*) FPU decydują bity ósmy i dziewiąty w rejestrze *Control Word*.

Precision	PC Field
Single Precision (24-Bits)	00B
Reserved	01B
Double Precision (53-Bits)	10B
Double Extended Precision (64-Bits)	11B

Program w języku C:

Program w pętli wykonuje jedną z trzech czynności wybraną przez użytkownika, czyli ustawia precyzję obliczeń, wyświetla bieżące ustawienie lub wywołuje dwie funkcje dla podanego argumentu. Obie funkcje wykonują obliczenia w FPU z wykorzystniem podanej zmiennej, a następnie drukują ją C.

```
#include <stdio.h>

extern int sprawdz();
extern void ustaw(int);
extern double f_fun(double);  // funkcje do weryfikacji
extern double g_fun(double);  // 'psucia sie' precyzji

int main () {
  int precyzja_obliczen, wybor_akcji;
  double x = 0;             // arg do f_fun i g_fun

  do {
```

```
    printf("\n\tWybierz akcje.\n");
    printf("\t1. Sprawdz precyzje obliczen.\n");
    printf("\t2. Ustaw wybrana precyzje obliczen.\n");
    printf("\t3. Wywolaj f(x) i g(x).\n");
    printf("\t0. Zakoncz program\n");
    printf("\t>");
    scanf("%d", &wybor_akcji);

    switch(wybor_akcji) {
      case 1:     //sprawdz precyzje obliczen
        printf("\n\tPrecyzja obliczen: ");
        switch(sprawdz()) {
          case 0:
            printf("Single Precision.\n");
            break;
          case 2:
            printf("Double Precision.\n");
            break;
          case 3:
            printf("Double Extended Precison.\n");
            break;
        }
      break;
    case 2:     //ustaw wybrana precyzje
      printf("\tPodaj precyzje obliczen (0, 2 lub 3): ");
      scanf("%d", &precyzja_obliczen);
      if (precyzja_obliczen != 0 && precyzja_obliczen != 2 &&
precyzja_obliczen != 3)
        printf("\n\tPodano zla precyzje!\n");
      else ustaw(precyzja_obliczen);
      break;
    case 3:
      printf("\n\tPodaj arguemnt do funkcji: ");
      scanf("%lf", &x);
      printf("\tf(x): %.18lf\n", f_fun(x));
```

```
      printf("\tg(x): %.18lf\n", g_fun(x));
      break;
    } while (wybor_akcji != 0);
    printf("\n\nKoniec programu.\n\n");
    return 0;
}
```

Funkcje w języku Asemblera:

W sekcji `.data` deklarowana jest dwubajtowa zmienna, która będzie używana do modyfikowania rejestru *Control Word* z FPU o tej samej wielości.

```
.data
control_word:     .short 0      #2 bajty (16 bitów)
double_precision: .short 0x0200
double_extended:  .short 0x0300
```

Deklaracje funkcji w sekcji `.text`:

```
.text
.global ustaw,   sprawdz, , f_fun, g_fun
.type   ustaw,   @function
.type   sprawdz, @function
.type   f_fun, @function
.type   g_fun, @function
```

Funkcja sprawdzająca bieżące ustawienie precyzji poprzez pobranie ósmego i dziewiątego bitu (bity *Precision Control*) z *Control Word* i zwrócenie ich przez funkcję - przekazanie zwracanej wartości w rejestrze `rax`:

```
#   sprawdz ();
#   %rax
sprawdz:
    pushq   %rbp
    movq    %rsp, %rbp
    movq    $0, %rax    #wyzerowanie rejestru do obliczeń
```

Pobranie zawartości rejestru *Control Word* do zmiennej o podanym adresie:

```
    fstcw   control_word
```

Instrukcja służąca do „poczekania” na wykonanie obliczeń w jednostce FPU, w której odbywa się sprawdzenie i obsłużenie wyjątków, które mogły się pojawić:

```
    fwait
```

Przeniesienie zmiennej do rejestru w celu wykonania obliczeń. Operacją logiczną `and` wszystkie bity poza 8. i 9. są zerowane. Następnie operacją przesunięcia bitowego bity kontroli precyzji trafiają na dwie najmniej znaczące pozycje w `rax`.

```
    movw    control_word, %ax
    andw    $0x300, %ax
    shrw    $8, %ax
```

Są trzy możliwe rezultaty powyższych operacji. Rejestr `rax` może przechowywać w sobie wartości liczbowe 0, 2 lub 3, co odpowiada odpowiednio *Single Precision*, *Double Precision* i *Double Extended Precision*. Wartości te są przekazywane dalej przy wywołaniu funkcji w programie w języku C oraz odpowiednio interpretowane przez instrukcje warunkowe.

```
    movq    %rbp, %rsp      #zakończenie funkcji
    popq    %rbp
    ret
```

Druga funkcja ustawia bity precyzji z *Control Word*. Precyzja, która ma być ustawiona jest przekazywana jako argument przyjmujący analogiczne wartości (0, 2 lub 3) do tych zwracanych przez funkcję sprawdzającą precyzję.

```
#   ustaw (prezycja_obliczen);
#   %rax    %rdi
ustaw:
    pushq   %rbp
    movq    %rsp, %rbp      #standardowe operacje na stosie

    # if(precyzja_obliczen == 0) single
    # if(precyzja_obliczen == 2) double
```

```
    # if(precyzja_obliczen == 3) extended double
```

Pobranie obecnego *Control Word* do rejestru `ax` pośrednio przez zmienną `control_word`:

```
    movq    $0, %rax
    fstcw   control_word
    fwait
    movw    control_word, %ax
```

Tym razem operacja logiczna `and` ma na celu wyzerowanie bitów precyzji jednocześnie nienaruszając innych bitów, ponieważ później Control Word zostanie załadowany zawartością rejestru `ax`.

```
    andw    $0xFCFF, %ax
```

Po tej operacji *Precision Control* to 00B, więc *Single Precision*. Jeśli przesłany argument wymagał tej precyzji, to funkcja przeskoczy do etykiety `end`.

```
    cmp     $2, %rdi
    jl      end
    je      set_double

    #jg
    set_extended:    #ustawienie dwóch bitów z PC (11B)
    xorw    double_extended, %ax
    jmp     end

    set_double:      #ustawienie 10B w PC
    xorw    double_precision, %a
```

Na koniec zawartość rejestru `ax` jest zapisywana do zmiennej `control_word`, a przez adres ten zmiennej załadowana do rejestru *Control Word* w FPU.

```
    end:
    movw    %ax, control_word
    fldcw   control_word
```

```
    movq    %rbp, %rsp
    popq    %rbp
    ret
```

Dwie wspomniane wcześniej funkcje wykonujące operacje na FPU:

```
#   f_fun (x);
#   %rax   %xmm0
    f_fun:
     pushq   %rbp
     movq    %rsp, %rbp

     subq    $8, %rsp
     movsd   %xmm0, (%rsp)
     fldl    (%rsp)
     fmul    %st(0)  # x^2
     fld1
     fadd    %st(1)  # x^2+1
     fsqrt           # sqrt(x^2+1)
     fld1
     fxch    %st(1)
     fsub    %st(1)  # sqrt(x^2+1)-1
     fstpl   (%rsp)
     movsd   (%rsp), %xmm0
     addq    $8, %rsp

     movq    %rbp, %rsp
     popq    %rbp
     ret
```

Wynik działania funkcji `f_fun`:

$f(x) = \sqrt{x^2 + 1} - 1$

oraz `g_fun`:

$g(x) = \frac{x^2}{\sqrt{x^2+1}+1}$

```
#   g_fun (x);
#   %rax   %xmm0
     g_fun:
     pushq   %rbp
     movq    %rsp, %rbp

     subq    $8, %rsp
     movsd   %xmm0, (%rsp)
     fldl    (%rsp)
     fmul    %st(0)      # x^2
     fld1
     fadd    %st(1)      # x^2+1
     fsqrt               # sqrt(x^2+1)
     fld1
     fadd    %st(1)      # sqrt(x^2+1)+1
     fldl    (%rsp)
     fmul    %st         # x^2
     fdiv    %st(1)      # x^2/(sqrt(x^2+1)+1)
     fstpl   (%rsp)
     movsd   (%rsp), %xmm0
     addq    $8, %rsp

     movq    %rbp, %rsp
     popq    %rbp
     ret
```

Zgodnie z przewidywaniami dla małych wyników działań funkcji f_fun i g_fun np. dla argumentów 0.00001, 0.0000001, ustawiona precyzja obliczeń w Control Word miała wpływ na wypisywaną wartość. Precyzja była gubiona dla mniejszych warości.

B. Aproksymacja funkcji cosinus

Program w języku C:

W programie wywoływana jest funkcja „cosinus” w języku Asemblera.

```
#include <stdio.h>
#include <math.h>          // skorzystanie ze stałej PI
```

Zewnętrzna funkcja pobiera dwa argumenty: kąt w radianach i liczbę kroków tj. liczbę wyrazów z szeregu do wyliczenia. Druga funkcja („taylor”) to prototyp funkcji asemblerowej i była wykorzystywana do porównania poprawności działania pierwszej funkcji.

```
extern double cosinus(double, int);
double taylor(double, int);
```

W programie głównym użytkownik proszony jest o podanie dwóch argumentów przekazywanych do funkcji (tj. kąt i liczba kroków). Podanie kąta może być zarówno w radianach jak i stopniach; przy drugiej opcji kąt jest przeliczany na radiany i podawany do wywołania.

```
int main()
{
    int liczba_wyrazow;
    double kat_rad;
    int wybor = 0;
    printf("\nKat bedzie podany w:\n1. Radianach\n2.
Stopniach\n>");
    scanf("%d", &wybor);
    switch(wybor) {
        case 1:
            printf("\nKat w radianach: ");
            scanf("%lf", &kat_rad);
            break;
        case 2:
```

```
            printf("\nKat w stopniach: ");
            scanf("%lf", &kat_rad);
            kat_rad = kat_rad * M_PI / 180;
            break;
    }

    printf("Liczba krokow: ");
    scanf("%d", &liczba_wyrazow);

    printf("\nWynik cosinus(asm): %lf\n", cosinus(kat_rad,
liczba_wyrazow));
    printf("Wynik taylor(C): %lf\n\n", taylor(kat_rad,
liczba_wyrazow));
    return 0;
}
```

Funkcja, na której była wzorowana funkcja „cosinus”:

```
double taylor(double x, int liczba_wyrazow) {
    double wyraz = 1, kwadrat = x*x, suma = 1;
    int n = 1;
    while(n < liczba_wyrazow) {
        wyraz *= -kwadrat/((2*n - 1) * 2*n);
        n++;
        suma += wyraz;
    }
    return suma;
}
```

Funkcja w języku Asemblera:

Każdy kolejny wyraz z aproksymacji cosinsa szeregiem Taylora jest obliczany przez mnożenie poprzedniego wyrazu przez $\frac{-x^2}{(2n-1)*2n}$, gdzie *x* - kąt w radianach, a *n* to numer aktualnego wyrazu (wyrazy numerowane od 0).

```
.data
minus:  .double -1.0

.text
.global cosinus
.type cosinus, @function

# cosinus (kat_rad, liczba_wyrazow)
# %rax    %xmm0     %rdi
cosinus:
pushq   %rbp
movq    %rsp, %rbp
```

Argument zmiennoprzecinkowy (kąt w radianach) przekazywany jest do funkcji przez rejestr `xmm0`. W przeniesieniu go na stos FPU pośredniczy zwykły stos.

```
subq    $8, %rsp        #64b rezerwy na stosie
movsd   %xmm0, (%rsp)   #usadzenie 128b na stosie
```

Instrukcja `movsd` oznacza „move scalar double-precision floating-point value”.

W następnych instrukcjach wartość kąta zostanie pobrana ze stosu do odpowiedniego rejestru FPU.

Instrukcje `fld` służą do załadowania określonych wartości na stos FPU. Ładowanie wartości jest zawsze na szczyt stosu, dlatego wszystkie wartości wcześniej załadowane zmieniają swoją numerację. Wszystkie dyrektywy operacji dokonywanych na tym stosie są poprzedzone literką „f”, a nazwy samych operacji są podobne do tych używanych poza FPU.

W poniższym fragmencie `fld1` ładuje do rejestru na szczycie stosu FPU liczbę „1”. `fldz` ładuje „0” („z” od „zero”). `fldl` ładuje liczbę ze zwykłego stosu lub z podanego adresu zmiennej.

```
fld1                    # 1.0 na stosie FPU
fld1                    # 1.0
fldz                    # 0.0
fldl    (%rsp)          # ST(0) = kat_rad;
fld1                    # ST(1) = ST(0) = 1.0
fld1                    # ST(2) = ST(1) = ST(0) = 1.0
```

```
# Stos FPU:
# ST(5) - $1.0
# ST(4) - wynik silni, (2n)!
# ST(3) - silnia jako nr wyrazu, 2*n (mianownik)
# ST(2) - kat_rad (kąt podany przez użytkownika)
# ST(1) - suma(suma wszystkich wyrazów i wynik do przesłania)
# ST(0) - wyraz (aktualny wyraz ciagu)

movq    $0, %rsi        #licznik pętli
fwait                   #instrukcja synchronizacji
```

Instrukcja `fwait` zapewnie synchronizację między jednostkami FPU a jednostką liczb całkowitych procesora.

```
loop:
cmpq    %rdi, %rsi
jge     end  #skok, gdy licznik (rsi) >= liczba_wyrazow (rdi)
incq    %rsi

# Obliczanie następnego wyrazu ciągu
# wyraz *= -x^2/(2n*(2n-1))

# Licznik
# wyraz *= -x^2
```

Operacja `fmul`, jak sugeruje nazwa, służy do mnożenia. W przypadku przemnażania przez zmienną w pamięci, konieczne jest dopisanie literki „l” w dyrektywie i podanie adresu zmiennej.

```
fmull   minus           # wyraz *= (-1)
fmul    %st(2), %st     # wyraz *= x
fmul    %st(2), %st     # wyraz *= x

# Mianownik
# (2n-2)! *= (2n-1) * 2n
```

`fxch` podmienia zawartości rejestru ze szczytu stosu FPU, czyli ST(0) z rejestrem

FPU podanym po dyrektywie. `fadd` dodaje dwie liczby zmiennoprzecinkowe. W każdej z tych operacji (`fxch`, `fadd`, `fmul`) musi brać udział rejestr ze szczytu stosu - ST(0); nie można dodawać czy mnożyć między sobą rejestrów FPU w dowolny sposób.

```
fxch    %st(3)   #podmiana zawartości rejestrów ST(0) i ST(3)
fadd    %st(5), %st     # 1 + (2n-2) = 2n-1
fmul    %st, %st(4)     # (2n-1) * (2n-2)! = (2n-1)!
fadd    %st(5), %st     # 1 + (2n-2) = 2n
fmul    %st, %st(4)     # 2n * (2n-1) = 2n!
fxch    %st(3)          #zmiana na ST(0) i ST(3); powrót do
                        #poprzedniego stanu na stosie FPU
```

W tym momencie na stosie FPU wyliczone są wyniki licznika i mianownika następnego wyrazu z szeregu Taylora (odpowiednio w rejestrach ST(0) i ST(4)). Następnym krokiem jest podzielenie licznika przez mianownik.

```
fldz
fadd    %st(1), %st    # 0 += licznik
fdiv    %st(5), %st     # wyraz /= (2n)!
fadd    %st, %st(2)     # suma += wyraz
fstp    %st             # zdjęcie wyrazu ze stosu
jmp     loop
```

Instrukcja `fstp` utylizuje wartość na szczycie stosu (przesuwa wskaźnik stosu).

```
end:
fstp    %st         #zdjęcie ze stosu FPU ST(0) i przesunięcie
                    #wszystkich wartości na stosie
fstpl   (%rsp)          #zdjęcie wartości ze szczytu
                        #stosu FPU i załadowanie na stos
movsd   (%rsp), %xmm0   #podanie sumy do xmm0 przez stos

movq    %rbp, %rsp      #operacje kończące funkcję
popq    %rbp
ret
```

3. Podsumowanie i wnioski

Jednostka zmiennoprzecinkowa to układ scalony wspomagający procesor w obliczeniach głównie zmiennoprzecinkowych, ale też na liczbach całkowitych. W architekturze x86-64 jej funkcjonalność jest połączona z jednostkami, aby realizować zadania zgodnie z metodologią SIMD (ang. *Single Instruction, Multiple Data*).

4. Bibliografia

- https://en.wikipedia.org/wiki/Floating-point_unit
- IA-32 Intel Architekture Software Developer's Manual, Volume 1: Basic Architecture, 2003 (rozdział ósmy)