Wrocław, 28 maja 2018

Marcin Kotas, 235098 WT-P-7.15

prowadząca: mgr Aleksandra Postawka

Laboratorium Architektury Komputerów
(5) Jednostka zmiennoprzecinkowa (FPU)

1 Treść ćwiczenia

Zakres i program ćwiczenia:

Pierwszy program napisany w języku C wywołuje funkcje napisane w języku asemblera:

1. pozwalająca na sprawdzenie wystąpienia wyjątków,
2. pozwalająca na ustawienie wybranego bitu maski wyjątków

Drugi program napisany w języku C wywołuje funkcję napisaną w języku asemblera: aproksymacja funkcji ln(1+x) za pomocą szeregu Taylora
x - argument zmiennoprzecinkowy
n - liczba iteracji

2 Program pierwszy

2.1 Program główny

W programie dostępne jest menu zaimplementowane jako pętla. Do wyboru są następujące operacje:

1. Sprawdzanie, czy wystąpiły wyjątki,
2. Ustawienie wybranych masek wyjątków,
3. Wyzerowanie wszystkich masek,
4. Przeprowadzenie testu dzielenia przez zero

Pierwsza opcja wywołuje funkcję zewnętrzną `checkExc()`, która zwraca liczbę definiującą znalezione wyjątki. Następnie po kolei sprawdza flagi poprzez wykonanie operacji modulo 2 (sprawdzenie czy bit zerowy jest ustawiony) i przesunięcie bitowe w prawo aby sprawdzić kolejną flagę.

Druga opcja pozwala na wybranie maski do ustawienia — od 0 do 5. Następnie wywoływana jest zewnętrzna funkcja `maskExc(int exc)`. Jako argument przekazywana jest liczba 2 podniesiona do potęgi równej wybranej masce. W ten sposób uzyskiwany jest ustawiony bit na pozycji odpowiadającej wybranej masce.

Opcje trzecia oraz czwarta wywołują odpowiednio `clrMasks()` oraz `testDivByZero()`.

```
printf("1. Check exceptions\n"
       "2. Mask exceptions\n"
       "3. Clear all masks\n"
       "4. Divide by zero test\n0. Exit\n>");
scanf("%d", &choice);

if (choice == 1){
    exceptions = checkExc();
    if (exceptions == 0){
        printf("No exceptions are set\n");
        continue;
    }
    if (exceptions % 2 == 1)    // bit 0
        printf("Invalid-Operation Exception\n");

    exceptions = exceptions >> 1;
    if (exceptions % 2 == 1)    // bit 1
        printf("Denormalized-Operand Exception\n");

    exceptions = exceptions >> 1;
    if (exceptions % 2 == 1)    // bit 2
        printf("Zero-Divide Exception\n");

    exceptions = exceptions >> 1;
    if (exceptions % 2 == 1)    // bit 3
        printf("Overflow Exception Exception\n");

    exceptions = exceptions >> 1;
    if (exceptions % 2 == 1)    // bit 4
        printf("Underflow Exception Exception\n");
```

```
    exceptions = exceptions >> 1;
    if (exceptions % 2 == 1)    // bit 5
        printf("Precision Exception Exception\n");
}

else if (choice == 2){
    printf("Choose exception to mask:\n"
    "0. Invalid-Operation Exception\n"
    "1. Denormalized-Operand Exception\n"
    "2. Zero-Divide Exception\n"
    "3. Overflow Exception\n"
    "4. Underflow Exception\n"
    "5. Precision Exception\n>");
    scanf("%d", &exceptions);
    if (exceptions > 5 || exceptions < 0){
        printf("Wrong argument!\n");
        continue;
    }

    maskExc((int)pow(2,exceptions)); // to get correct bits set
}

else if (choice == 3){
    clrMasks();
}

else if (choice == 4){
    testDivByZero();
}
```

2.2 Funkcje asemblerowe

Wszystkie funkcje są zadeklarowane w obrębie jednego pliku.

```
.text
.global checkExc, clrMasks,
        maskExc, testDivByZero
.type checkExc, @function
.type maskExc, @function
.type testDivByZero, @function
```

2.2.1 checkExc

Funkcja wczytuje zawartość rejestru statusu do rejestru `ax` za pomocą funkcji `fstsw`. Funkcja `fwait` wykonywana jest, aby upewnić się, że następna instrukcja wykona się po wczytaniu rejestru statusu, a nie w trakcie. Następnie czyści starszą część rejestru, aby zostawić tylko flagi wyjątków. Wynik zwraca przez rejestr `rax`.

```
checkExc:
    mov $0, %rax
    fstsw %ax
# store status word
    fwait
    and $0x0ff, %ax
# clear all but exception bits
```

2.2.2 clrMasks

Funkcja wczytuje zawartość rejestru kontrolnego do pamięci za pomocą funkcji `fnstcw`, a następnie kopiuje do rejestru `ax`. Następnie czyści młodsze 6 bitów rejestru, które odpowiadają maskom wyjątków. Na koniec ładuje zawartość rejestru `rax` do pamięci i ładuje do rejestru kontrolnego za pomocą funkcji `fldcw`.

```
clrMasks:
    fnstcw controlWord
    fwait
    mov controlWord, %ax
    and $0xffc0, %ax
    mov %ax, controlWord
    fldcw controlWord
# load control word
ret
```

2.2.3 maskExc

Funkcja wczytuje zawartość rejestru kontrolnego jak poprzednio. Następnie operacją `OR` ustawia odpowiednie maski przekazane do funkcji w `rdi`. Na koniec ładuje zawartość rejestru `rax` do rejestru kontrolnego.

```
maskExc:
# exception type in %rdi
    fnstcw controlWord
    fwait
    mov controlWord, %ax
    or %di, %ax
    mov %ax, controlWord
    fldcw controlWord
ret
```

2.2.4 testDivByZero

Na początku ładowane są do stosu FPU zero oraz jeden. Następnie wykonywane jest dzielenie `ST(0) / ST(1)` funkcją `fdivp`, która również usunie wartość na szczycie stosu. Na koniec usuwana jest ostatnia wartość pozostająca w stosie.

```
testDivByZero:
    fldz
    fld1
    fdivp
    fstp %st    # cleanup
ret
```

3 Program drugi

3.1 Program główny

Program wczytuje argumenty zewnętrznej funkcji `lnApprox` za pomocą funkcji `printf`. `x` musi zawierać się w przedziale $(-1, 1]$, ponieważ rozszerzenie Taylora bazuje na sumie szeregu geometrycznego. `steps` oznacza ilość kroków do wykonania. Następnie drukuje wynik wywołania tej funkcji.

```
double x;
int steps;
printf("x from (-1,1]:");
scanf("%lf", &x);
printf("Steps:");
scanf("%d", &steps);

double res = lnApprox(x,steps);
printf("ln(1+%lf) approximation: %lf\n", x, res);
```

3.2 Funkcja lnApprox

Kolejność działań bazuje na następującym wzorze:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots$$

3.2.1 Ładowanie zmiennych do FPU

Na początku funkcja przekazuje argument z `xmm0` na stos, aby możliwe było załadowanie go do rejestru FPU. Następnie potrzebne zmienne i stałe ładowane są do `FPU`:

- `finit` czyści i inicjalizuje jednostkę `FPU`
- `fld1` ładuje do rejestru `FPU` wartość 1
- `fldl (%rsp)` wczytuje liczbę ze stosu
- `fldz` ładuje wartość 0

```
    movsd %xmm0, (%rsp)
    mov %rdi, %r10  # number of steps

    finit
    fld1            # ST(3|4) = 1.0 (divisor)
    fldl (%rsp)     # ST(2|3) = x
    fldz            # ST(1|2) = result
    fld1            # ST(0|1) = next pows of x
                    # ST(  0) = buf
```

3.2.2 Pętla ln_loop

W rejestrze `ST(0)` znajdują się kolejne potęgi `x`. Przed uruchomieniem pętli jego wartość wynosi 1. Na początku każdej iteracji pętli wartość tego rejestru wymnażana jest przez `x` (`ST(2)`). Następnie wartość tego rejestru jest ładowana ponownie (`fld %st`), aby można było wykonać na niej działania jednocześnie zachowując potęgę `x` do dalszych obliczeń.

Funkcja `fdiv` dokonuje dzielenia `ST(0)/ST(4)`. `faddp` dodaje wartość `ST(0)` do `ST(2)` oraz usuwa wartość z `ST(0)`.

Następnie na stos FPU ładowana jest wartość 1, która dodawana jest do `ST(4)`, w celu inkrementacji dzielnika. Funkcja `fchs` zmienia znak `ST(0)` (kolejne potęgi `x`) na przeciwny.

```
ln_loop:
    fmul %st(2), %st    # ST(0) * x
    fld %st             # store pow of x
    fdiv %st(4), %st    # ST(0) = ST(0)/ ST(4)
    faddp %st, %st(2)   # ST(2) = ST(0) + ST(2) & pop
    fld1                # inc divisor
    faddp %st, %st(4)   #
    fchs                # next pow x * (-1)

    dec %r10
    cmp $0, %r10
    jg ln_loop
```

3.2.3 Zwracanie wartości

Funkcja najpierw usuwa wartość z góry (potęgi dwójki) za pomocą `fstp %st` — kopiowanie na siebie i 'pop'. Następnie przenosi wynik ze stosu FPU na wskaźnik stosu — `fstpl (%rsp)`, po czym kopiuje tą wartość do rejestru `xmm0`.

Na końcu funkcja usuwa dwie pozostałe wartości ze stosu FPU.

```
    fstp %st              # pop pows of 2
    fstpl (%rsp)          # pop result
    movsd (%rsp), %xmm0 # return result in xmm0

    fstp %st    # cleanup
    fstp %st
```

4 Wnioski

Podczas korzystania z jednostki FPU należy pamiętać o 'przesuwaniu' się rejestrów podczas ładowania wartości. Przed rozpoczęciem działań warto wywołać funkcję `finit`, aby upewnić się, że stos FPU jest pusty oraz ustawienia są domyślne. Podczas ładowania oraz odkładania wartości z pamięci lub stosu należy pamiętać o odpowiednim przyrostku:

- `s` jeśli wartość jest 32 bitowa (single)
- `l` jeśli jest 64 bitowa (double)
- `t` jeśli jest 80 bitowa (extended double)

Przyrostki te nie są zgodne z tymi stosowanymi w każdym innym miejscu w języku asemblera (`l` dla wartości 32 bitowych, `q` dla 64 bitowych). Odnoszą się one do nazw formatów zmiennoprzecinkowych — `s`hort real, `l`ong real, `t`emporary real.