Laboratorium Architektury Komputerów

Ćwiczenie 5

Operacje na liczbach zmiennoprzecinkowych

Wprowadzenie

Liczby zmiennoprzecinkowe (zmiennopozycyjne) zostały wprowadzone do techniki komputerowej w celu usunięcia wad zapisu stałoprzecinkowego. Wady te są wyraźnie widoczne w przypadku, gdy w trakcie obliczeń wykonywane są działania na liczbach bardzo dużych i bardzo małych. Warto dodać, że format zmiennoprzecinkowy dziesiętny stosowany jest od dawna w praktyce obliczeń (nie tylko komputerowych) i polega na przedstawieniu liczby w postaci iloczynu pewnej wartości (zwykle normalizowanej do przedziału <1, 10) i potęgi o podstawie 10, np. $3.37 \cdot 10^6$. Dane w tym formacie wprowadzane do komputera zapisuje się zazwyczaj za pomocą litery e, np. 3.37e6.

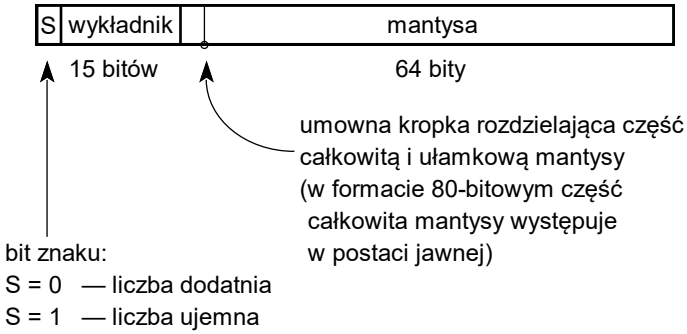
W komputerach używane są binarne formaty liczb zmiennoprzecinkowych, które od około trzydziestu lat są znormalizowane i opisane w amerykańskim standardzie IEEE 754. Wszystkie współczesne procesory, w tym koprocesor arytmetyczny w architekturze *x86,* spełniają wymagania tego standardu.

Ponieważ działania na liczbach zmiennoprzecinkowych są dość złożone, zwykle realizowane są przez odrębny procesor zwany *koprocesorem arytmetycznym*. Koprocesor arytmetyczny jest umieszczony w jednej obudowie z głównym procesorem, chociaż funkcjonalnie stanowi on oddzielną jednostkę, która może wykonywać obliczenia niezależnie od głównego procesora. Koprocesor arytmetyczny oferuje bogatą listę rozkazów wykonujących działania na liczbach zmiennoprzecinkowych, w tym działania arytmetyczne, obliczanie wartości funkcji (trygonometrycznych, logarytmicznych, itp.) i wiele innych.

Ze względu na stopniowo wzrastający udział przetwarzania danych multimedialnych (dźwięki, obrazy), około roku 2000 w procesorach wprowadzono nową grupę rozkazów określaną jako *Streaming SIMD Extension*, w skrócie SSE. Występujący tu symbol SIMD oznacza rodzaj przetwarzania wg klasyfikacji Flynn'a: *Single Instruction, Multiple Data,* co należy rozumieć jako możliwość wykonywania działań za pomocą jednego rozkazu jednocześnie (równolegle) na kilku danych, np. za pomocą jednego rozkazu można wykonać dodawanie czterech par liczb zmiennoprzecinkowych. Zagadnienia te omawiane są szerzej w dalszej części opracowania.

Architektura koprocesora arytmetycznego

Koprocesor arytmetyczny stanowi odrębny procesor, współdziałający z procesorem głównym, i znajdujący się w tej samej obudowie. Koprocesor wykonuje działania na 80-bitowych liczbach zmiennoprzecinkowych, których struktura pokazana jest na rysunku. W tym formacie liczb zmiennoprzecinkowych część całkowita mantysy występuje w postaci jawnej, a wartość umieszczona w polu wykładnika jest przesunięta w górę o 16383 w stosunku do wykładnika oryginalnego.



Liczby, na których wykonywane są obliczenia, składowane są w 8 rejestrach 80-bitowych tworzących stos. Rozkazy koprocesora adresują rejestry stosu nie bezpośrednio, ale względem wierzchołka stosu. W kodzie asemblerowym rejestr znajdujący się na wierzchołku stosu oznaczany jest ST(0) lub ST, a dalsze ST(1), ST(2),..., ST(7).

Z każdym rejestrem stosu koprocesora związany jest 2-bitowy rejestr pomocniczy (nazywany czasami *polem stanu rejestru*), w którym podane są informacje o zawartości odpowiedniego rejestru stosu. Ponadto aktualny stan koprocesora jest reprezentowany przez bity tworzące 16-bitowy *rejestr stanu koprocesora*. W rejestrze tym m.in. zawarte są informacje o zdarzeniach w trakcie obliczeń (tzw. wyjątki), które mogą, opcjonalnie, powodować zakończenie wykonywania programu lub nie.

Z kolei również 16-bitowy *rejestr sterujący* pozwala wpływać na pracę koprocesora, m.in. możliwe jest wybranie jednego z czterech dostępnych sposobów zaokrąglania.

Koprocesor oferuje bogatą listę rozkazów. Na poziomie asemblera mnemoniki koprocesora zaczynają się od litery F. Stosowane są te same tryby adresowania co w procesorze, a w polu operandu mogą występować obiekty o długości 32, 64 lub 80 bitów. Przykładowo, rozkaz

```
fadd      ST(0), ST(3)
```

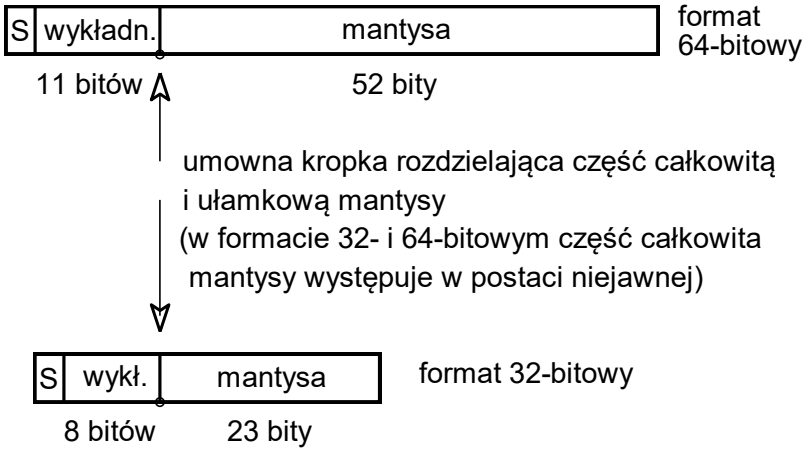
powoduje dodanie do zawartości rejestru ST(0) zawartości rejestru ST(3). Rejestr ST(0) jest wierzchołkiem stosu, natomiast rejestr ST(3) jest rejestrem oddalonym od wierzchołka o trzy pozycje. Warto dodać, że niektóre rozkazy nie mają jawnego operandu, np. `fabs` zastępuje liczbę na wierzchołku stosu przez jej wartość bezwzględną.

Do przesyłania danych używane są przede wszystkim instrukcje (rozkazy) FLD i FST. Instrukcja FLD ładuje na wierzchołek stosu koprocesora liczbę zmiennoprzecinkową pobraną z lokacji pamięci lub ze stosu koprocesora. Instrukcja FST powoduje przesłanie zawartości wierzchołka stosu do lokacji pamięci lub do innego rejestru stosu koprocesora. Obie te instrukcje mają kilka odmian, co pozwala m.in. na odczytywanie z pamięci liczb całkowitych w kodzie U2 z jednoczesną konwersją na format zmiennoprzecinkowy (instrukcja FILD, natomiast analogiczna instrukcja FIST zapisuje liczbę w pamięci w postaci całkowitej w kodzie U2). Dostępne są też instrukcje wpisujące na wierzchołek stosu niektóre stałe matematyczne, np. FLDPI.

Warto zwrócić uwagę, że załadowanie wartości na wierzchołek stosu powoduje, że wartości wcześniej zapisane na stosie dostępne będą poprzez indeksy większe o 1, np. wartość ST(3) będzie dostępna jako ST(4). Z tych powodów poniższa sekwencja instrukcji jest błędna:

```
FST       ST(7)
FLD       xvar    ; błąd! — ST(7) staje się ST(8), a takiego rejestru nie ma
```

Wartości zmiennoprzecinkowe obliczone przez koprocesor zapisywane są w pamięci zazwyczaj nie w postaci liczb 80-bitowych (chociaż jest to możliwe), ale najczęściej w formatach krótszych: 64-bitowym formacie *double* lub 32-bitowym formacie *float*. Struktura tych formatów pokazana jest na rysunku.



Wartości umieszczone w polu wykładnika są przesunięte względem wykładnika oryginalnego: w formacie 64-bitowym (*double*) o 1023 w górę, a w formacie 32-bitowym (*float*) o 127 w górę.

Liczba zmiennoprzecinkowa zapisana na wierzchołku stosu koprocesora może być zapisana w pamięci za pomocą rozkazu `FST`. Ponieważ ten sam rozkaz `FST` używany jest do zapisywania liczb 32- i 64-bitowych, konieczne jest podanie rozmiaru w postaci:

`dword PTR` dla liczb 32-bitowych

`qword PTR` dla liczb 64-bitowych.

Przykładowo, zapisanie zawartości wierzchołka stosu koprocesora w zmiennej `wynik` w postaci liczby 32-bitowej wymaga użycia rozkazu

```
fst  dword PTR wynik
```

W szczególności, użycie operatora PTR jest konieczne w przypadku tzw. odwołań anonimowych, tj. takich, w których nie występuje nazwa zmiennej, np.

```
fst  qword PTR [ebx]
```

Podobnie, w przypadku ładowania na wierzchołek stosu koprocesora wartości pobranej z pamięci używa się rozkazu `fld` także z operatorem `PTR`, np.:

```
fld  dword PTR [ebp+12]
```

Jeśli liczba pobierana z pamięci jest zwykłą liczbą całkowitą ze znakiem (w kodzie U2), to w takim przypadku używa się rozkazu `fild`, np.

```
fild      dword PTR [ebp+12]
```

Rozkaz ten automatycznie zamienia liczbę całkowitą na postać zmiennoprzecinkową i zapisuje na wierzchołku stosu koprocesora `st(0)`. Analogiczne działanie ma rozkaz `fist`.

Na liście rozkazów koprocesora arytmetycznego znajdują się między innymi rozkazy wykonujące działania arytmetyczne: dodawanie `FADD`, odejmowanie `FSUB`, mnożenie `FMUL` i dzielenie `FDIV`. Rozkazy te wykonują działania na dwóch argumentach, przy czym jednym z nich musi rejestr `st(0)` stanowiący wierzchołek stosu rejestrów koprocesora, np.

```
fmul st(3), st(0)
```

W powyższym przykładzie wynik mnożenia wpisywany jest do `st(3)`. W przypadku, gdy drugi argument operacji znajduje się w pamięci, to argument `st(0)` pomija się, np.

```
fsub dword PTR [esi]
```

Podany rozkaz odejmuje od liczby znajdującej się na wierzchołku stosu koprocesora (tj. `st(0)`) liczbę w formacie *float* znajdującą się w komórce pamięci o adresie podanym w rejestrze ESI. Zauważmy, że wyrażenie `dword PTR` opisuje argument 32-bitowy.

Dość użyteczny jest także rozkaz, który usuwa liczbę z wierzchołka stosu koprocesora nigdzie jej nie wpisując:

```
fstp  st(0)
```

Często rozkaz ten jest poprzedza operacja arytmetyczna, np.:

```
fmul  st(1), st(0)
fstp  st(0)
```

Oba te rozkazy można zapisać w jednym wierszu:

```
fmulp  st(1), st(0)
```

lub jeszcze krócej w postaci bezargumentowej: `fmulp` lub `fmul`.

Uwagi o operandach instrukcji koprocesora

Sposób zapisu operandów instrukcji koprocesora może niekiedy budzić wątpliwości, np. ta sama instrukcja dodawania `FADD` może występować w postaci z dwoma operandami, z jednym, lub w ogóle bez operandów. Poniżej podajemy najważniejsze zalecenia dotyczące tego problemu.

1. Zawartości rejestrów procesora (np. ECX) nie mogą być operandami rozkazów koprocesora – z tego względu poniższy rozkaz jest błędny:

   ```
   fld     ecx ; błąd !
   ```

   Ewentualne przesłanie zawartości rejestru procesora do koprocesora lub odwrotnie wykonuje się za pośrednictwem zmiennej w obszarze danych programu. W przypadku liczb całkowitych wygodnie jest posługiwać się zmienną dynamiczną ulokowaną na zwykłym stosie. Przykładowo, przesłanie liczby całkowitej (w kodzie U2) znajdującej się w rejestrze ECX na wierzchołek stosu koprocesora połączone z zamianą na postać zmiennoprzecinkową można zrealizować następująco:

   ```
   mov     ecx, 7          ; liczba przykładowa
   push    ecx
   fild    dword PTR [esp]
   add     esp, 4
   ```

2. Liczby niecałkowite biorące udział w obliczeniach wykonywanych przez koprocesor umieszcza się zazwyczaj sekcji danych programu (`.data`) w asemblerze. Asembler automatycznie przekształca liczby dziesiętne zawierające kropkę dziesiętną na postać zmiennoprzecinkową 32- lub 64-bitową w zależności od zastosowanych dyrektyw.

Przykładowo, jeśli w sekcji danych programu można zadeklarować zmienną (64-bitową) `przebieg` i nadać jej wartość początkową:

```
przebieg  dq   -2504.35
```

Z kolei, w części rozkazowej programu zmienną tę można załadować na wierzchołek stosu koprocesora za pomocą rozkazu:

```
fld  qword PTR przebieg
```

3. Instrukcje wykonujące działania arytmetyczne: `FADD`, `FSUB`, `FMUL`, `FDIV` mają zawsze dwa operandy, z których jeden lub dwa mogą być niejawne.
   a. jeśli oba operandy odnoszą się do rejestrów koprocesora `st(0)`, `st(1)`, `st(2)`, . . . , to oba operandy muszą być podane w postaci jawnej, przy czym jednym z nich musi być `st(0)`.
   b. jeśli w mnemoniku instrukcji występuje dodatkowa litera `P` (skrót od `POP`), to po wykonaniu właściwych czynności przez rozkaz nastąpi usunięcie liczby na wierzchołku stosu koprocesora, np. `FMULP st(3), st(0)`
   c. jeśli jednym z operandów jest zawartość lokacji pamięci, to pierwszym operandem (niejawnym) jest `st(0)`, natomiast drugi operand (jawny) opisuje położenie danej w pamięci, np. rozkaz

      ```
      fdiv qword PTR [ebx]
      ```

      spowoduje podzielenie liczby znajdującej się na wierzchołku stosu koprocesora (tj. w `st(0)`) przez liczbę (tu: typu *double*) znajdującej się komórce pamięci o adresie wskazanym przez zawartość rejestru EBX.
   d. rozkazy bez operandów (omawiane wcześniej) wykonują działania na elementach stosu koprocesora `st(1)` i `st(0)` – przykładowo, instrukcja `fmul`, jest równoważna poniższej:

      ```
      fmulp  st(1), st(0)
      ```

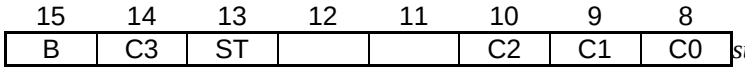
      tak samo dla instrukcji `FADD`, `FSUB`, `FDIV`.
4. Rozkaz `FCHS` przeprowadza zmianę znaku liczby na wierzchołku stosu koprocesora (`st(0)`) i nie ma operandów.
5. Rozkaz `FXCH` zamienia zawartość wierzchołka stosu koprocesora z zawartością podanego operandu. Rozkaz ma tylko jeden operand, np. `fxch  st(5)`. W postaci bez operandów rozkaz zamienia zawartości rejestrów koprocesora `st(0)` i `st(1)`. W szczególności rozkazy `fxch` i `fxch st(1)` działają identycznie.
6. Rozkaz `FRNDINT` zaokrągla liczbę znajdującą się w `st(0)` do najbliższej liczby całkowitej (typ zaokrąglenia zależy od zawartości bitów RC w rejestrze sterującym koprocesora).

Porównywanie liczb zmiennoprzecinkowych

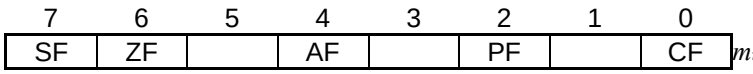
W obliczeniach zmiennoprzecinkowych porównania występuje znacznie rzadziej w zwykłym procesorze. Dostępnych jest kilka rozkazów porównujących wartości zmiennoprzecinkowe,

przy czym wynik porównania wpisywany jest do ustalonych bitów rejestru stanu koprocesora. M.in, rozkaz `FCOM x` porównuje ST(0) z operandem `x` i ustawia bity C3 i C0 w rejestrze stanu koprocesora: C3=C0=0, gdy ST(0) > `x` albo C3=0, C0=1 w gdy ST(0) < `x`. Jeśli porównywane wartości są równe, to C3=1, C0=0. Stan C3=C0=1 oznacza, że porównanie nie mogło być przeprowadzone.

Bity w rejestrze stanu koprocesora określające wynik porównania zostały umieszczone na pozycjach odpowiadających znaczników w rejestrze procesora – pozwala to na wykorzystanie zwykłych instrukcji skoków warunkowych (dla liczb bez znaku). Przedtem trzeba jednak przepisać starszą część rejestru stanu koprocesora do młodszej części rejestru znaczników procesora. Ilustruje to podana niżej sekwencja rozkazów.



starsze bity rejestru stanu koprocesora



młodsze bity rejestru znaczników procesora

```
FCOM      ST(1)     ; porównanie ST(0)  i  ST(1)
FSTSW     AX        ; zapamiętanie rej.stanu koproc. w AX
SAHF           ; przepisanie AH do rejestru znaczników
JZ        ROWNE
JA        WIEKSZE
```

Począwszy od procesora Pentium Pro dostępny jest także rozkaz `FCOMI`, który wpisuje wynik porównania od razu do rejestru znaczników procesora. Stan znaczników procesora (ZF, PF, CF) po wykonaniu rozkazu `FCOMI` podano w poniższej tabeli. Warto porównać zawartość poniższej tabeli z opisem działania rozkazu `CMP`, który używany jest porównywania liczb stałoprzecinkowych (ćw. 2, str. 12).

	ZF	PF	CF
ST(0) > x	0	0	0
ST(0) < x	0	0	1
ST(0) = x	1	0	0
niezdefiniowane	1	1	1

Rozkaz `FCOMI` ma dwa operandy, z których pierwszy jest zawsze wierzchołkiem stosu koprocesora, a drugi innym rejestrem stosu koprocesora, np. `FCOMI ST(0), ST(7)`. Nie jest dostępny format `FCOMI` bez operandów.

Wartości specjalne w koprocesorze arytmetycznym

W formatach liczb zmiennoprzecinkowych obsługiwanych przez koprocesor arytmetyczny wyodrębniono pewne szczególne wartości, które interpretowane są w niestandardowy sposób. Wartości te, określane jako *wartości specjalne*, kodowane są w postaci liczb zmiennoprzecinkowych, w których pole wykładnika zawiera same bity zerowe albo same bity jedynkowe. Między innymi do wartości specjalnych należy liczba 0 (a także −0), również wartości bliskie zera, dla których część całkowita mantysy (niejawna) jest równa

0, i wiele innych. Dalsze informacje na ten temat podane są w materiałach pomocniczych do wykładu.

Przykład: fragment programu wyznaczający pierwiastki równania kwadratowego

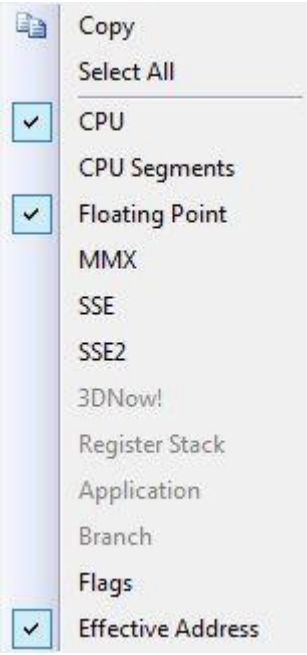
Poniżej podano fragment programu, w którym rozwiązywane jest równanie kwadratowe $2x^2 - x - 15 = 0$, przy czym wiadomo, że równanie ma dwa pierwiastki rzeczywiste różne. Współczynniki równania a = 2, b = –1, c = –15 podane są w sekcji danych w postaci 32-bitowych liczb zmiennoprzecinkowych (format *float*). Fragment programu nie zawiera rozkazów wyświetlających pierwiastki równania (x1 = –2.5, x2 = 3) na ekranie — działanie programu można sprawdzić posługując się debuggerem (zob. dalszy opis).

```
.686
.model flat

.data
; 2x^2 - x - 15 = 0
wsp_a          dd        +2.0
wsp_b          dd        -1.0
wsp_c          dd        -15.0

dwa       dd        2.0
cztery    dd        4.0
x1        dd        ?
x2        dd        ?
_   _   _   _   _   _   _   _   _   _
.code
_   _   _   _   _   _   _   _   _   _

          finit
          fld     wsp_a         ; załadowanie współczynnika a
          fld     wsp_b         ; załadowanie współczynnika b
          fst     st(2)         ; kopiowanie b

; sytuacja na stosie: ST(0) = b, ST(1) = a, ST(2) = b

          fmul    st(0),st(0)   ; obliczenie b^2
          fld     cztery

; sytuacja na stosie: ST(0) = 4.0, ST(1) = b^2, ST(2) = a,
; ST(3) = b

           fmul    st(0), st(2) ; obliczenie 4 * a
           fmul    wsp_c        ; obliczenie 4 * a * c
           fsubp   st(1), st(0) ; obliczenie b^2 - 4 * a * c
```

```
; sytuacja na stosie: ST(0) = b^2 - 4 * a * c, ST(1) = a,
; ST(2) = b

          fldz                ; zaladowanie 0

; sytuacja na stosie:   ST(0) = 0, ST(1) = b^2 - 4 * a * c,
;                  ST(2) = a, ST(3) = b

; rozkaz FCOMI - oba porównywane operandy musza być podane na
; stosie koprocesora
          fcomi   st(0), st(1)


; usuniecie zera z wierzchołka stosu
          fstp    st(0)

          ja      delta_ujemna ; skok, gdy delta ujemna

; w przykładzie nie wyodrębnia się przypadku delta = 0

; sytuacja na stosie: ST(0) = b^2 - 4 * a * c, ST(1) = a,
; ST(2) = b

          fxch    st(1)      ; zamiana st(0) i st(1)

; sytuacja na stosie: ST(0) = a, ST(1) = b^2 - 4 * a * c,
; ST(2) = b

          fadd    st(0), st(0) ; ; obliczenie 2 * a
          fstp    st(3)

; sytuacja na stosie: ST(0) = b^2 - 4 * a * c,  ST(1) = b,
; ST(2) = 2 * a

          fsqrt               ; pierwiastek z delty
; przechowanie obliczonej wartości
         fst     st(3)

; sytuacja na stosie: ST(0) = sqrt(b^2 - 4 * a * c),
; ST(1) = b, ST(2) = 2 * a, ST(3) = sqrt(b^2 - 4 * a * c)

          fchs                ; zmiana znaku
          fsub    st(0), st(1); obliczenie -b - sqrt(delta)
          fdiv    st(0), st(2); obliczenie x1
          fstp    x1          ; zapisanie x1 w pamięci

; sytuacja na stosie: ST(0) = b, ST(1) = 2 * a,
; ST(2) = sqrt(b^2 - 4 * a * c)
```

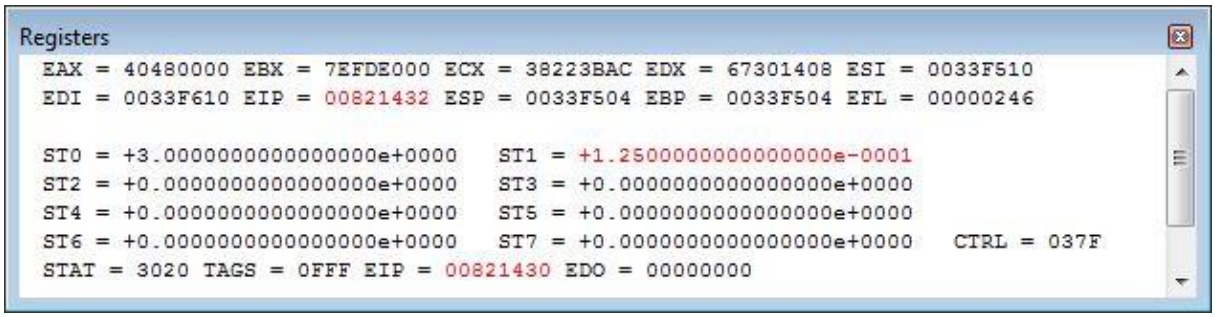
```
fchs                ; zmiana znaku
fadd    st(0), st(2)
fdiv    st(0), st(1)
fstp    x2

fstp    st(0)       ; oczyszczenie stosu
fstp    st(0)
```

Wykorzystanie debuggera do śledzenia operacji zmiennoprzecinkowych



Wykorzystanie *debuggera* zintegrowanego z systemem Visual Studio w trakcie uruchamiania programów opisane jest szczegółowo w instrukcji do ćwiczenia 1. *Debugger* wspomaga także uruchamianie programów wykorzystujących rozkazy koprocesora arytmetycznego.

Przypomnijmy, że w systemie Microsoft Visual Studio *debuggowanie* programu jest wykonywane po naciśnięciu klawisza F5. Przedtem należy ustawić punkt zatrzymania (ang. breakpoint) poprzez kliknięcie na obrzeżu ramki obok rozkazu, przed którym ma nastąpić zatrzymanie. Po uruchomieniu *debuggowania*, można otworzyć potrzebne okna, wśród których najbardziej przydatne jest okno prezentujące zawartości rejestrów procesora. W tym celu wybieramy opcje Debug / Windows / Registers. Następnie, w oknie rejestrów klikamy prawym klawiszem myszki i rozwijanym menu zaznaczamy opcję Floating Point (zob. rysunek) — w rezultacie w oknie rejestrów wyświetlane będą także zawartości rejestrów roboczych koprocesora `st(0)`, `st(1)`, `...`, `st(7)`. Ponadto, w oknie rejestrów wyświetlana jest także zawartość rejestru sterującego koprocesora (symbol `CTRL`) i rejestru stanu koprocesora (symbol `STAT`).



Po naciśnięciu klawisza F5 program jest wykonywany aż do napotkania (zaznaczonego wcześniej) punktu zatrzymania. Można wówczas wykonywać pojedyncze rozkazy programu poprzez wielokrotne naciskanie klawisza F10. Podobne znaczenie ma klawisz F11, ale w tym przypadku śledzenie obejmuje także zawartość podprogramów.

Wybierając opcję Debug / Stop debugging można zatrzymać *debuggowanie* programu. Prócz podanych, dostępnych jest jeszcze wiele innych opcji, które można wywołać w analogiczny sposób.

Zadanie 5.1. Napisać podprogram w asemblerze przystosowany do wywoływania z poziomu języka C. Prototyp funkcji implementowanej przez ten podprogram ma postać:

```
float srednia_harm (float * tablica, unsigned int n);
```

Podprogram ten powinien obliczyć średnią harmoniczną

$$\frac{n}{\frac{1}{a_1}+\frac{1}{a_2}+\frac{1}{a_3}+\ldots+\frac{1}{a_n}}$$

dla `n` liczb zmiennoprzecinkowych a_1, a_2, a_3,..., a_n, zawartych w tablicy `tablica`.
Napisać także krótki program przykładowy w języku C ilustrujący sposób wywoływania tego podprogramu.
Wskazówki:

1. Podprogram (jeśli zwraca wartość `float` lub `double`) powinien pozostawić obliczoną wartość na wierzchołku stosu rejestrów koprocesora.
2. W języku C do wprowadzania liczb zmiennoprzecinkowych z klawiatury używa się zazwyczaj funkcji `scanf` lub `scanf_s`. W przypadku liczb typu `float` stosuje się format `%f`, a w przypadku liczb typu `double` — format `%lf`. W przypadku wyświetlania wyników za pomocą funkcji `printf`, format `%lf` stosuje się do wartości typu `long double`, natomiast `%f` dla liczb typu `float` i `double`.

Przykład: fragment programu wyznaczający wartość e^x

Obliczenia realizowane za pomocą koprocesora arytmetycznego wymagają dość często dostosowania formuł obliczeniowych do specyfiki koprocesora. Mogą tu być przydatne niektóre wzory znane ze szkoły średniej, między innymi jeśli konieczna jest zmiana podstawy algorytmu, to można wykorzystać poniższy wzór:

$$\log_a b = \frac{\log_c b}{\log_c a}$$

Charakterystycznym przykładem może być, niżej pokazane, obliczenie wartości funkcji e^x za pomocą koprocesora arytmetycznego, co w porównaniu ze zwykłym kalkulatorem może się wydawać dość skomplikowane. Obliczenie to wymaga użycia poniższych rozkazów:

F2XM1 obliczenie ST(0) ← $(2^{ST(0)} - 1)$, przy czym ST(0) ∈ <–1, +1>
FSCALE obliczenie ST(0) ← ST(0) ∗ $2^{ST(1)}$, przy czym ST(1) jest wartością całkowitą
FLDL2E wpisanie na wierzchołek stosu koprocesora wartości $\log_2 e$
FRNDINT zaokrąglenie zawartości wierzchołka stosu do liczby całkowitej

Podane dalej symbole $[\ ]_c$ i $[\ ]_u$ oznaczają, odpowiednio, część całkowitą i ułamkową wartości podanej w nawiasach.

$$e^x = 2^{x\log_2 e} = 2^{[x\log_2 e]_c} \cdot 2^{[x\log_2 e]_u} = 2^{[x\log_2 e]_c} \cdot ((2^{[x\log_2 e]_u} - 1) + 1) =$$
$$= \text{FSCALE}\ (\text{F2XM1}([x\log_2 e]_u) + 1, [x\log_2 e]_c)$$

W obliczeniach wykorzystuje się zależność $a^b = 2^{b * \log_2 a}$, skąd wynikają podane niżej instrukcje

```
     fldl2e                    ; log 2 e
     fmulp     st(1), st(0)    ; obliczenie x * log 2 e

; kopiowanie obliczonej wartości do ST(1)
     fst       st(1)

; zaokrąglenie do wartości całkowitej
     frndint

     fsub      st(1), st(0)    ; obliczenie części ułamkowej

     fxch                      ; zamiana ST(0) i ST(1)
; po zamianie: ST(0) - część ułamkowa, ST(1) - część całkowita

; obliczenie wartości funkcji wykładniczej dla części
; ułamkowej wykładnika
     f2xm1

     fld1                      ; liczba 1
     faddp     st(1), st(0)    ; dodanie 1 do wyniku

; mnożenie przez 2^(część całkowita)
     fscale

; przesłanie wyniku do ST(1) i usunięcie wartości
; z wierzchołka stosu
     fstp      st(1)

; w rezultacie wynik znajduje się w ST(0)
```

Zadanie 5.2. Napisać podprogram w asemblerze przystosowany do wywoływania z poziomu języka C. Prototyp funkcji implementowanej przez ten podprogram ma postać:

```
float  nowy_exp  (float  x);
```

Podprogram ten powinien obliczyć sumę 20 początkowych wyrazów szeregu

$$1 + \frac{x}{1} + \frac{x^2}{1 \cdot 2} + \frac{x^3}{1 \cdot 2 \cdot 3} + \frac{x^4}{1 \cdot 2 \cdot 3 \cdot 4} + \ldots$$

Napisać także krótki program przykładowy w języku C ilustrujący sposób wywoływania tego podprogramu.

Wskazówka: podprogram (jeśli zwraca wartość `float` lub `double`) powinien pozostawić obliczoną wartość na wierzchołku stosu rejestrów koprocesora.

Rozkazy dla zastosowań multimedialnych

Zauważono pewną specyfikę programów wykonujących operacje na obrazach i dźwiękach: występują tam fragmenty kodu, które wykonują wielokrotnie powtarzające się działania arytmetyczne na liczbach całkowitych i zmiennoprzecinkowych, przy dość łagodnych wymaganiach dotyczących dokładności.

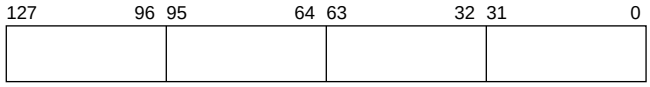
W architekturze x86 wprowadzono specjalne grupy rozkazów MMX i SSE przeznaczone do wykonywania ww. operacji. Rozkazy te wykonują równoległe operacje na kilku danych. Wprowadzone rozkazy przeznaczone są głównie do zastosowań w zakresie grafiki komputerowej i przetwarzania dźwięków, gdzie występują operacje na dużych zbiorach liczb stało- i zmiennoprzecinkowych.

Rozkazy grupy MMX wykorzystują rejestry 64-bitowe, które stanowią fragmenty 80-bitowych rejestrów koprocesora arytmetycznego, co w konsekwencji uniemożliwia korzystanie z rozkazów koprocesora, jeśli wykonywane są rozkazy MMX. Z tego względu, w miarę poszerzania opisanej dalej grupy SSE, rozkazy MMX stopniowo wychodzą z użycia.

Typowe rozkazy grupy SSE wykonują równoległe operacje na czterech 32-bitowych liczbach zmiennoprzecinkowych — można powiedzieć, że działania wykonywane są na czteroelementowych *wektorach liczb zmiennoprzecinkowych*. Wykonywane obliczenia są zgodne ze standardem IEEE 754. Dostępne są też rozkazy wykonujące działania na liczbach stałoprzecinkowych (wprowadzone w wersji SSE2).

Dla SSE w trybie 32-bitowym dostępnych jest 8 rejestrów oznaczonych symbolami XMM0 ÷ XMM7. Każdy rejestr ma 128 bitów i może zawierać:

- 4 liczby zmiennoprzecinkowe 32-bitowe (zob. rysunek), lub



- 2 liczby zmiennoprzecinkowe 64-bitowe, lub
- 16 liczb stałoprzecinkowych 8-bitowych, lub
- 8 liczb stałoprzecinkowych 16-bitowych, lub
- 4 liczby stałoprzecinkowe 32-bitowe.

W trybie 64-bitowym dostępnych jest 16 rejestrów oznaczonych symbolami XMM0 ÷ XMM15. Dodatkowo, za pomocą rejestru sterującego MXCSR można wpływać na sposób wykonywania obliczeń (np. rodzaj zaokrąglenia wyników).

Zazwyczaj ta sama operacja wykonywana jest na każdej parze odpowiadających sobie elementów obu operandów. Zawartości podanych operandów można traktować jako wektory złożone z 2, 4, 8 lub 16 elementów, które mogą być liczbami stałoprzecinkowymi lub zmiennoprzecinkowymi (w tym przypadku wektor zawiera 2 lub 4 elementy). W tym sensie rozkazy SSE mogą traktowane jako rozkazy wykonujące działania na wektorach.

Zestaw rozkazów SSE jest ciągle rozszerzany (SSE2, SSE3, SSE4, SSE5). Kilka rozkazów wykonuje działania identyczne jak ich konwencjonalne odpowiedniki — do grupy tej należą rozkazy wykonujące bitowe operacje logiczne: `PAND`, `POR`, `PXOR`. Podobnie działają też rozkazy przesunięć, np. `PSLLW`. W SSE4 wprowadzono m.in. rozkaz obliczający sumę kontrolną CRC–32 i rozkazy ułatwiające kompresję wideo.

Ze względu na umiarkowane wymagania dotyczące dokładności obliczeń, niektóre rozkazy (np. RCPPS) nie wykonują obliczeń, ale wartości wynikowe odczytują z tablicy — indeks potrzebnego elementu tablicy stanowi przetwarzana liczba.

Dostępne są operacje "poziome", które wykonują działania na elementach zawartych w tym samym wektorze. W przypadku rozkazów dwuargumentowych, podobnie jak przypadku zwykłych rozkazów dodawania lub odejmowania, wyniki wpisywane są do obiektu (np. rejestru XMM) wskazywanego przez pierwszy argument.

Wśród rozkazów grupy SSE nie występują rozkazy ładowania stałych. Potrzebne stałe trzeba umieścić w pamięci i miarę potrzeby ładować do rejestrów XMM. Prosty sposób zerowania rejestru polega na użyciu rozkazu PXOR, który wyznacza *sumę modulo dwa* dla odpowiadających sobie bitów obu operandów, np. `pxor xmm5, xmm5`. Wypełnienie całego rejestru bitami o wartości 1 można wykonać za pomocą rozkazu porównania PCMPEQB, np. `pcmpeqb xmm7, xmm7`.

Dla wygody programowania zdefiniowano 128-bitowy typ danych oznaczony symbolem XMMWORD. Typ ten może być stosowany do definiowania zmiennych statycznych, jak również do określania rozmiaru operandu, np.

```
odcinki   XMMWORD   ?
_   _   _   _   _   _   _   _   _   _   _   _
; przesłanie słowa 128-bitowego do rejestru XMM0
          movdqa   xmm0, xmmword PTR [ebx]
```

Analogiczny typ 64-bitowy MMWORD zdefiniowano dla operacji MMX (które jednak wychodzą z użycia).

Niektóre rozkazy wykonują działania zgodnie z regułami tzw. arytmetyki nasycenia (ang. saturation): nawet jeśli wynik operacji przekracza dopuszczalny zakres, to wynikiem jest największa albo najmniejsza liczba, która może być przedstawiona w danym formacie. Także inne rozkazy wykonują dość specyficzne operacje, które znajdują zastosowanie w przetwarzaniu dźwięków i obrazów.

Operacje porównania wykonywane są oddzielnie dla każdej pary elementów obu wektorów. Wyniki porównania wpisywane są do odpowiednich elementów wektora wynikowego, przy czym jeśli testowany warunek był spełniony, to do elementu wynikowego wpisywane są bity o wartości 1, a w przeciwnym razie bity o wartości 0. Poniższy przykład ilustruje porównywanie dwóch wektorów 16-elementowych zawartych w rejestrach xmm3 i xmm7 za pomocą rozkazu PCMPEQB. Rozkaz ten zeruje odpowiedni bajt wynikowy, jeśli porównywane bajty są niejednakowe, albo wpisuje same jedynki jeśli bajty są identyczne.

xmm3

11111110	00100011	11111011	00000111					01101101	10001111	01111111

xmm7

11111110	00000011	11111011	00000111					01101101	10001111	10111111

Po wykonaniu rozkazu **`pcmpeqb xmm3, xmm7`**

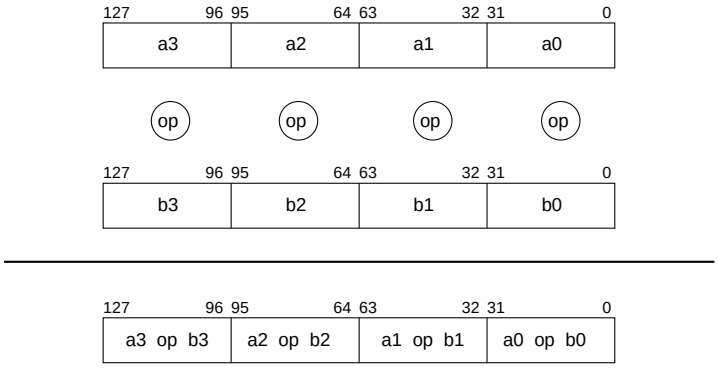
xmm3

11111111	00000000	1111111	11111111					11111111	11111111	00000000

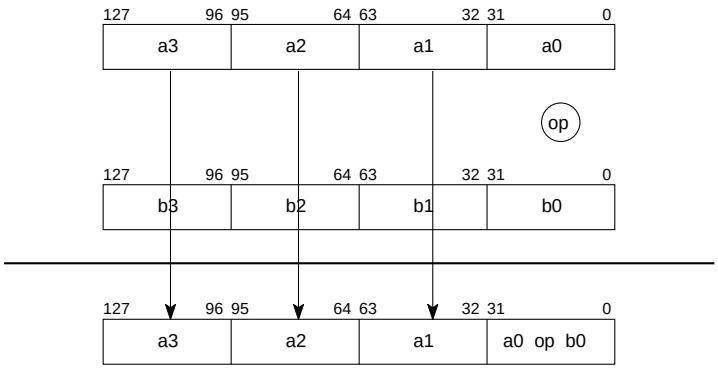
Przy omawianej organizacji obliczeń konstruowanie rozgałęzień w programach za pomocą zwykłych rozkazów skoków warunkowych byłoby kłopotliwe i czasochłonne. Z tego powodu instrukcje wektorowe typu if ... then ... else konstruuje się w specyficzny sposób, nie używając rozkazów skoku, ale stosując w zamian bitowe operacje logiczne. Zagadnienia te wykraczają poza zakres niniejszego opracowania.

Rozkazy grupy SSE mogą wykonywać działania na danych:

- *upakowanych* (ang. packed instructions) — zestaw danych obejmuje cztery liczby; instrukcje działające na danych spakowanych mają przyrostek ps;



- *skalarnych* (ang. scalar instructions) — zestaw danych zawiera jedną liczbę, umieszczoną na najmniej znaczących bitach; pozostałe trzy pola nie ulegają zmianie; instrukcje działające na danych skalarnych mają przyrostek ss;



Debugger zintegrowany z systemem Visual Studio może być także wykorzystany do śledzenia rozkazów z grupy SSE. W tym przypadku (zob. rys. str. 5) w oknie rejestrów, po naciśnięciu prawego klawisza myszki trzeba wybrać opcję SSE — w oknie rejestrów zostaną wyświetlone zawartości rejestrów XMM.

```
; Program przykładowy ilustrujący operacje SSE procesora

; Poniższy podprogram jest przystosowany do wywoływania
; z poziomu języka C (program arytmc_SSE.c)

.686
.XMM  ; zezwolenie na asemblację rozkazów grupy SSE
.model flat

public _dodaj_SSE, _pierwiastek_SSE, _odwrotnosc_SSE

.code

_dodaj_SSE PROC
            push  ebp
            mov   ebp, esp
            push  ebx
            push  esi
            push  edi

            mov   esi, [ebp+8]    ; adres pierwszej tablicy
            mov   edi, [ebp+12]   ; adres drugiej tablicy
            mov   ebx, [ebp+16]   ; adres tablicy wynikowej

; ładowanie do rejestru xmm5 czterech liczb zmiennoprzecin-
; kowych 32-bitowych - liczby zostają pobrane z tablicy,
; której adres poczatkowy podany jest w rejestrze ESI

; interpretacja mnemonika "movups" :
; mov - operacja przesłania,
; u - unaligned (adres obszaru nie jest podzielny przez 16),
; p - packed (do rejestru ładowane są od razu cztery liczby),
; s - short (inaczej float, liczby zmiennoprzecinkowe
; 32-bitowe)

            movups   xmm5, [esi]
            movups   xmm6, [edi]

; sumowanie czterech liczb zmiennoprzecinkowych zawartych
; w rejestrach xmm5 i xmm6
            addps    xmm5, xmm6
```

```
; zapisanie wyniku sumowania w tablicy w pamięci
            movups   [ebx], xmm5

            pop   edi
            pop   esi
            pop   ebx
            pop   ebp
            ret
_dodaj_SSE ENDP


;=========================================================

_pierwiastek_SSE   PROC
            push  ebp
            mov   ebp, esp
            push  ebx
            push  esi
            mov   esi, [ebp+8]    ; adres pierwszej tablicy
            mov   ebx, [ebp+12]   ; adres tablicy wynikowej

; ładowanie do rejestru xmm5 czterech liczb zmiennoprzecin-
; kowych 32-bitowych - liczby zostają pobrane z tablicy,
; której adres początkowy podany jest w rejestrze ESI

; mnemonik "movups": zob. komentarz podany w funkcji dodaj_SSE
            movups   xmm6, [esi]

; obliczanie pierwiastka z czterech liczb zmiennoprzecinkowych
; znajdujących sie w rejestrze xmm6
; - wynik wpisywany jest do xmm5
            sqrtps    xmm5, xmm6

; zapisanie wyniku sumowania w tablicy w pamięci
            movups   [ebx], xmm5

            pop   esi
            pop   ebx
            pop   ebp
            ret

_pierwiastek_SSE   ENDP

;=========================================================

; rozkaz RCPPS wykonuje obliczenia na 12-bitowej mantysie
; (a nie na typowej 24-bitowej) - obliczenia wykonywane są
; szybciej, ale są mniej dokładne
```

```
_odwrotnosc_SSE    PROC
             push  ebp
             mov   ebp, esp
             push  ebx
             push  esi

             mov   esi, [ebp+8]    ; adres pierwszej tablicy
             mov   ebx, [ebp+12]   ; adres tablicy wynikowej

; ladowanie do rejestru xmm5 czterech liczb zmiennoprzecin-
; kowych 32-bitowych - liczby zostają pobrane z tablicy,
; której adres poczatkowy podany jest w rejestrze ESI

; mnemonik "movups": zob. komentarz podany w funkcji dodaj_SSE
             movups   xmm5, [esi]

; obliczanie odwrotności czterech liczb zmiennoprzecinkowych
; znajdujących się w rejestrze xmm6
; - wynik wpisywany jest do xmm5
             rcpps    xmm5, xmm6


; zapisanie wyniku sumowania w tablicy w pamieci
             movups   [ebx], xmm5

             pop   esi
             pop   ebx
             pop   ebp
             ret

_odwrotnosc_SSE    ENDP

END

=====================

/* Program przykładowy ilustrujący operacje SSE procesora
   Program jest przystosowany do współpracy z podprogramem
   zakodowanym w asemblerze (plik arytm_SSE.asm)
*/


#include <stdio.h>

void dodaj_SSE (float *, float *, float *);
void pierwiastek_SSE (float *, float *);
void odwrotnosc_SSE (float *, float *);

int main()
```

```
{
  float p[4] = {1.0,   1.5,  2.0,  2.5};
  float q[4] = {0.25, -0.5,  1.0, -1.75};
  float r[4];

  dodaj_SSE (p, q, r);
  printf ("\n%f  %f  %f  %f", p[0], p[1], p[2], p[3]);
  printf ("\n%f  %f  %f  %f", q[0], q[1], q[2], q[3]);
  printf ("\n%f  %f  %f  %f", r[0], r[1], r[2], r[3]);

  printf("\n\nObliczanie pierwiastka");

  pierwiastek_SSE (p, r);
  printf ("\n%f  %f  %f  %f", p[0], p[1], p[2], p[3]);
  printf ("\n%f  %f  %f  %f", r[0], r[1], r[2], r[3]);

  printf("\n\nObliczanie odwrotności - ze względu na \
stosowanie");
  printf("\n12-bitowej mantysy obliczenia są mało dokładne");

  odwrotnosc_SSE (p, r);
  printf ("\n%f  %f  %f  %f", p[0], p[1], p[2], p[3]);
  printf ("\n%f  %f  %f  %f", r[0], r[1], r[2], r[3]);

  return 0;
}
```

Zadanie 5.3. Wzorując się na podanych przykładach napisać program w języku C i w asemblerze, który wyznaczy sumy odpowiadających sobie elementów dwóch tablic `liczby_A` i `liczby_B`, z których każda zawiera 16 liczb 8-bitowych ze znakiem (typ `char`):

```
char liczby_A[16] = {-128, -127, -126, -125, -124, -123, -122,
     -121, 120,  121,  122,  123,  124,  125,  126, 127};

char liczby_B[16] = {-3, -3, -3, -3, -3, -3, -3, -3,
                      3,  3,  3,  3,  3,  3,  3,  3};
```

Do sumowania wykorzystać rozkaz `PADDSB` (wersja SSE), który sumuje, z uwzględnieniem nasycenia, dwa wektory 16-elementowe złożone z liczb całkowitych 8-bitowych. Wyjaśnić (pozorne) błędy w obliczeniach.

Zadanie 5.4. Napisać podprogram w asemblerze przystosowany do wywoływania z poziomu języka C. Podprogram powinien zamienić dwie liczby całkowite typu `int` umieszczone w tablicy calkowite na dwie liczby zmiennoprzecinkowe typu `float` i umieścić je w tablicy

`zmienno_przec`. Napisać także krótki program w języku C ilustrujący sposób wywoływania obu wersji podprogramu.

Prototyp funkcji implementowanej przez podprogram ma postać:

```
void int2float (int * calkowite, float * zmienno_przec);
```

Zamianę na format `float` należy zrealizować za pomocą rozkazu `cvtpi2ps` (z grupy SSE), który zamienia dwie liczby całkowite typu `int` na dwie liczby typu `float`. Wartości wynikowe zostają zapisane w rejestrze SSE, a operandem źródłowym może być 64-bitowa lokacja pamięci, np.

```
cvtpi2ps xmm5, qword PTR [esi]
```

Przykładowy fragment programu w języku C może mieć postać:

```
int a[2] = {-17, 24} ;
float r[4];
// podany rozkaz zapisuje w pamięci od razu 128 bitów,
// więc muszą być 4 elementy w tablicy

int2float(a, r);
printf ("\nKonwersja = %f  %f\n", r[0], r[1]);
```

Zadanie 5.5. Napisać podprogram w asemblerze przystosowany do wywoływania z poziomu języka C. Prototyp funkcji implementowanej przez ten podprogram ma postać:

```
void pm_jeden (float * tabl);
```

gdzie `tabl` jest tablicą zawierającą cztery liczby zmiennoprzecinkowe typu `float`. Podprogram ten, korzystając z rozkazu ADDSUBPS (grupa SSE3) powinien dodać 1 do elementów tablicy o indeksach nieparzystych i odjąć 1 od pozostałych elementów tablicy. Do testowania opracowanego podprogramu można wykorzystać poniższy program w języku C.

```
#include <stdio.h>
void pm_jeden (float * tabl);
int main()
{
  float tablica[4]={27.5,143.57,2100.0, -3.51};
  printf("\n%f   %f   %f   %f\n", tablica[0],
          tablica[1], tablica[2], tablica[3]);
  pm_jeden (tablica);
  printf("\n%f   %f   %f   %f\n", tablica[0],
          tablica[1], tablica[2], tablica[3]);
  return 0;
}
```

Wskazówki:

1. W sekcji danych modułu w asemblerze należy zdefiniować tablicę zawierającą cztery liczby 1.0 w formacie `float`.

2. Rozkaz ADDSUBPS wykonuje działania na czterech odpowiadających sobie 32-bitowych liczbach zmiennoprzecinkowych, które znajdują się w dwóch rejestrach XMM. Działanie rozkazu wyjaśnia poniższy przykład (rozkaz `ADDSUBPS xmm3, xmm5`).

Pierwszy operand: xmm3

a	b	c	d

Drugi operand: xmm5

e	f	g	h

Wynik: xmm3

a + e	b – f	c + g	d – h

Zadanie 5.6. Poniżej podano kod prostego programu (w języku C i w asemblerze), który wyznacza sumy odpowiadających sobie elementów trzech tablic zawierających liczby zmiennoprzecinkowe: `tabl_A`, `tabl_B`, `tabl_C`. Na początku sekcji danych występuje dyrektywa `ALIGN 16`, która powoduje że znajdująca się za nią dana zostanie umieszczona w lokacji pamięci o adresie podzielnym przez 16. Uruchomić program w środowisku VS2013. Następnie usunąć znak komentarza przed daną `liczba db 1` i ponownie uruchomić program. Wyjaśnić dlaczego wprowadzenie dodatkowej danej spowodowało błąd wykonania programu. Zaproponować sposób korekcji programu, tak by wyeliminować błąd wykonania. Wskazówka: porównać opisy instrukcji `movaps` i `movups`.

```
.686
.XMM
.model flat
public _dodawanie_SSE

.data
ALIGN 16
tabl_A    dd   1.0, 2.0, 3.0, 4.0
tabl_B    dd   2.0, 3.0, 4.0, 5.0
;liczba   db   1
tabl_C    dd   3.0, 4.0, 5.0, 6.0

.code
_dodawanie_SSE  PROC
                push          ebp
                mov           ebp, esp
                mov           eax, [ebp+8]

                movaps        xmm2, tabl_A
```

```
                movaps          xmm3, tabl_B
                movaps          xmm4, tabl_C

                addps           xmm2, xmm3
                addps           xmm2, xmm4
                movups          [eax], xmm2

                pop       ebp
                ret
_dodawanie_SSE  ENDP
END
```

```
#include <stdio.h>
void dodawanie_SSE(float * a);
int main()
{
     float wyniki[4];
     dodawanie_SSE(wyniki);
     printf("\nWyniki = %f  %f  %f  %f\n",
          wyniki[0], wyniki[1], wyniki[2], wyniki[3]);
     return 0;
}
```

Rozkazy do obliczeń wektorowych
(ta część instrukcji jest nadobowiązkowa – opisuje działania na instrukcjach grupy AVX 2.0, które są niedostępne w starszych typach procesorów Intel)

Jednym z paradygmatów przetwarzania w komputerowych systemach równoległych jest model SPMD (Single Program, Multiple Data). Stanowi on podkategorię w modelu MIMD (Multiple Instruction, Multiple Data) w klasyfikacji Flynna.

W paradygmacie SPMD, wiele autonomicznych procesorów jednocześnie (równolegle) wykonuje ten sam program w niezależnych punktach sterowania, podczas gdy w SIMD równoległe wykonanie sprowadza się do wykonania pojedynczej instrukcji na pewnym zbiorze danych (np. 8 liczbach zmiennoprzecinkowych).

Do tej pory, w SPMD procesy były wykonywane na procesorach ogólnego przeznaczenia. SIMD wymagały użycia procesora wektorowego do przetwarzania strumieni danych. Aktualnie, ze względu na dołączenie jednostek wektorowych (VPU, Vector Processor Unit) do autonomicznych rdzeni procesorów wielordzeniowych, obydwa paradygmaty zostały podłączone.

W architekturze x86 wprowadzono specjalne grupy rozkazów AVX, AVX 2.0 przeznaczone do wykonywania operacji na wektorach liczb. Podobnie jak w SSE i MMX, rozkazy te wykonują równoległe operacje na kilku danych. AVX stanowi rozszerzenie zestawu SSE. Planowane jest w najbliższym czasie pojawienie się procesorów w rozkazami AVX-512 (rejestry 512 bitowe).

Rozkazy grupy AVX wykorzystują rejestry 256-bitowe, które są dwa razy większe niż rejestry SSE. Rejestrów tych jest 16 i mają przyporządkowane nazwy YMM0 ÷ YMM15. Istniejące w SSE 128-bitowe rejestry XMM stanowią młodsze części rejestrów YMM.

Typowe rozkazy grupy AVX wykonują równoległe operacje na ośmiu 32-bitowych liczbach zmiennoprzecinkowych (*wektorach)*. Każdy rejestr ma 256 bitów i może zawierać:

- 8 liczby zmiennoprzecinkowe 32-bitowe, lub
- 4 liczby zmiennoprzecinkowe 64-bitowe, lub
- 32 liczb stałoprzecinkowych 8-bitowych, lub
- 16 liczb stałoprzecinkowych 16-bitowych, lub
- 8 liczby stałoprzecinkowe 32-bitowe.

Dodatkowo, za pomocą rejestru sterującego MXCSR można wpływać na sposób wykonywania obliczeń (np. rodzaj zaokrąglenia wyników).

Dla wygody programowania zdefiniowano 256-bitowy typ danych oznaczony symbolem YMMWORD. Typ ten może być stosowany do definiowania zmiennych statycznych, jak również do określania rozmiaru operandu, np.

```
Dane      YMMWORD   ?
_   _   _   _   _   _   _   _   _   _   _   _
; przesłanie słowa 256-bitowego do rejestru YMM0
          vmovaps ymm0,Dane
```

Debugger zintegrowany z systemem Visual Studio może być także wykorzystany do śledzenia rozkazów z grupy AVX. W tym przypadku w oknie rejestrów, po naciśnięciu prawego klawisza myszki trzeba wybrać opcję AVX/AVX2/AVX512 — w oknie rejestrów zostaną wyświetlone zawartości rejestrów YMM.

```
; Program przykładowy ilustrujący operacje AVX 2.0 procesora

; Poniższy podprogram jest przystosowany do wywoływania
; z poziomu języka C++ (program arytmc_AVX.cpp) w trybie 64
; bitowym

public FMA
;   podprogram oblicza matX=scalarA*matX + matY, tzw. AXPY
;   ostatni paramter count informuje o dlugosci wektora
; (float * matX, float * matY, float scalarA, int count);
;    rcx         rdx             xmm2          r9
; matA
; call -> rbp+8
; rbp  -> rbp

.code

FMA:
     ;prolog i zapamiętanie rejestrów
     push rbp
     mov rbp,rsp
     push rbx
```

```
     push rsi
     push rdi

     mov rsi,rcx    ; utwórz kopię adresu macierzy A
     mov rdi,rdx    ; utwórz kopię adresu macierzy B

; wyznaczenie liczby powtórzeń ecx<- count/32
; długość wektora musi być wielokrotnością liczby 32
     mov rdx,0
     mov rbx,32
     mov rax,r9
     div rbx
     xchg rdx,rax
     cmp rax,0
     jne koniec

     mov rcx,rdx  ; w rcx ilosc wykonan
```

```
; właściwa pętla obliczeniowa
again:

; xmm2 do pamięci (czyli mnożnik scalarA)
vmovups XMMWORD PTR dana32, xmm2

; przeniesienie wartosci scalarA do wszystkich 8 części ymm2
vbroadcastss  ymm2,dana32
; w rejestrze ymm2 jest 8 razy scalarA


; załadowanie 8 kolejnych elementów  macierzy matA do ymm0
vmovaps ymm0,YMMWORD PTR [rsi]

; załadowanie 8 kolejnych elementów macierz matB do ymm1
vmovaps ymm1, YMMWORD PTR [rdi]


; rozkaz mnożenia typu FMA ymm0 <- ymm0 * ymm2 + ymm1
VFMADD132PS ymm0,ymm1,ymm2   ; ymmA <- ymmA * ymmC + ymmB
; czyli wykonano fa[k] = a * fa[k] + fb[k];

; zapis wyniku do macierzy matA
vmovaps YMMWORD PTR [rsi],ymm0
```

```
;aktualizacja wskazników
add rsi,8*4
add rdi,8*4
loop again

koniec:

pop rdi
pop rsi
pop rbx
pop rbp
ret

.data
dana32  dd  4 dup (?)  ; miejsce na parametr scalarA

END
```

```
=====================

/* Program przykładowy ilustrujący operacje FMA z
wykorzystaniem instrukcji AVX procesora
Program jest przystosowany do współpracy z podprogramem
zakodowanym w asemblerze (plik funkcjeAVX.asm)
*/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <Windows.h>
#include <time.h>


extern "C" int FMA(float * matA, float * matB,
     float scalar, int count);

const __int64 DELTA_EPOCH_IN_MICROSECS = 11644473600000000;

#define FLOPS_ARRAY_SIZE (1024*1024)
#define MAXFLOPS_ITERS 100000000
#define LOOP_COUNT 128
#define FLOPSPERCALC 2

__declspec(align(64)) float fa[FLOPS_ARRAY_SIZE];
__declspec(align(64)) float fb[FLOPS_ARRAY_SIZE];
```

```
struct timezone2
{
     __int32  tz_minuteswest;
     bool  tz_dsttime;
};

struct timeval2 {
     __int32    tv_sec;         /* seconds */
     __int32    tv_usec;        /* microseconds */
};
int gettimeofday(struct timeval2 *tv, struct timezone2 *tz)
{
     FILETIME ft;
     __int64 tmpres = 0;
     TIME_ZONE_INFORMATION tz_winapi;
     int rez = 0;

     ZeroMemory(&ft, sizeof(ft));
     ZeroMemory(&tz_winapi, sizeof(tz_winapi));

     GetSystemTimeAsFileTime(&ft);

     tmpres = ft.dwHighDateTime;
     tmpres <<= 32;
     tmpres |= ft.dwLowDateTime;

     tmpres /= 10;
     tmpres -= DELTA_EPOCH_IN_MICROSECS;
     tv->tv_sec = (__int32)(tmpres*0.000001);
     tv->tv_usec = (tmpres % 1000000);

     rez = GetTimeZoneInformation(&tz_winapi);
     tz->tz_dsttime = (rez == 2) ? true : false;
     tz->tz_minuteswest = tz_winapi.Bias +
     ((rez == 2) ? tz_winapi.DaylightBias : 0);

     return 0;
}

double dtime()
{
     double tseconds = 0.0;
     struct timeval2 mytime;
     struct timezone2 myzone;
     gettimeofday(&mytime, &myzone);
     tseconds =
          (double)(mytime.tv_sec + mytime.tv_usec*1.0e-6);
     return tseconds;
```

```
}


int main(int argc, char *argv[])
{
     /*  celem programu jest obliczenie czasu wykonania
     operacji FMA dla dwóch macierzy fa i fb
     o wymiarach 1024x1024 ( FLOPS_ARRAY_SIZE)

     liczba powtórzeń obliczeń wynosi
                         MAXFLOPS_ITERS  100000000
     */

     int i, j, k;
     double tstart, tstop, ttime;
     double gflops = 0.0;
     float a = 2.0;

     printf("Inicjalizacja \r\n");

     // wypełnienie tablicy fa i fb pewnymi wartościami
     for (i = 0; i < FLOPS_ARRAY_SIZE; i++)
     {
          fa[i] = (float)i + 0.1f;
          fb[i] = (float)i + 0.2f;
     }

     printf("Początek obliczeńMAXFLOPS_ITERS \n");


     tstart = dtime();

     // MAXFLOPS_ITERS

          for (j = 0; j < MAXFLOPS_ITERS; j++)
          {
          if (FMA(fa, fb, a, LOOP_COUNT) != 0) exit(0);
          // obliczenie wartości z wykorzystaniem instrukcji
          // AVX2   fa = a*fa + fb


          /*
          Ten komentarz zawiera odpowiednik funkcji FMA w
          języku C
          */
          for ( k = 0; k < LOOP_COUNT; k++)
          {
          fa[k] = a*fa[k] + fb[k];
```

```
        }

    }

    tstop = dtime();


    gflops = (double)(1.0e-9 * LOOP_COUNT * MAXFLOPS_ITERS *
                                            FLOPSPERCALC);
    ttime = tstop - tstart;
    if (ttime > 0.0)
    {
        printf("GFlops = %10.3lf, secs =%10.2lf\n", gflops,
                            ttime);
    }

    return 0;

}
```

Zadanie 5.7. Wzorując się na podanych przykładach obliczyć różnicę w wydajności obliczeń przy wykorzystaniu wektoryzacji z AVX oraz bez jej wykorzystania.