Programowanie jednostki zmiennoprzecinkowej x87.

Jednostka zmiennoprzecinkowa (Floating-Point Unit – FPU) była dawniej odrębnym układem (koprocesorem), rozszerzającym systemy oparte na procesorach 8088 – 80386 o możliwość sprzętowego wykonywania arytmetyki zmiennoprzecinkowej – od prostych operacji typu +, –, *, /, po obliczanie pierwiastka kwadratowego, funkcje trygonometryczne i logarytmiczne.

Ze względu na ówczesną dużą złożoność układu (większą od prostych procesorów np. 8088), rzutującą na jego końcową cenę, FPU była na płytach głównych montowana opcjonalnie, w zależności od zastosowań komputera PC. Układy te nosiły oznaczenia, odpowiednio, 8087 – 80387, stąd ogólnie przyjęta nazwa x87.

Nie wdając się w szczegóły techniczne (sposób dekodowania instrukcji i współpracy – CPU i FPU współdzieliły magistrale danych i adresową), wraz z rozwojem technologii wytwarzania układów scalonych FPU zaczęto integrować w jednej strukturze razem z procesorem. Zaczynając od 80486DX, wszystkie CPU Pentium I i jego pochodne już były w nią standardowo wyposażone.

To, co ważne z punktu widzenia programisty, to zgodność FPU x87 z normą IEEE 754 oraz specyficzny model programowy (stos rejestrów, instrukcje, typy i transfer danych), nadal traktujący jednostkę zmiennoprzecinkową jako oddzielny układ (utrzymanie zgodności programowej z poprzednimi generacjami).

Uwaga: arytmetyka zmiennoprzecinkowa, typy danych, zaokrąglanie itp. została przedstawiona na jednym poprzednich wykładów (prezentacja).

Bardziej szczegółowy opis programowania FPU zawarty jest w:

Intel 64 and IA-32 Architectures Software Developer's Manual
Volume 1: Basic Architecture, Chapter 8 – Programming with the x87 FPU (pdf na stronie przedmiotu).

(Tylko) najważniejsze cechy:

- FPU x87 może wykonywać obliczenia na typach danych:

  – rozszerzonej precyzji (*double extended precision*, 80 bitów, *long double* w języku C) – domyślny,
  – podwójnej precyzji (*double precision*, 64 bity, *double*),
  – pojedynczej precyzji (*single precision*, 32 bity, *float*).

- FPU posiada osiem 80–bitowych rejestrów danych **zorganizowanych jako stos** (o tym dalej) i kilka rejestrów specjalnego przeznaczenia (*Control Register, Status Register, Tag…*).

- Dane w rejestrach przechowywane są **w formacie 80-bitowym**.

- FPU może pobierać i zapisywać dane z i do pamięci w jednym z trzech ww. typów zmiennoprzecinkowych, jak również liczby całkowite ze znakiem i spakowane BCD – konwersja dokonywana jest „w locie".

- Argumentami rozkazów FPU mogą być **tylko jego własne rejestry lub adresy komórek pamięci** (tryby adresowania pamięci i składnia są takie jak w x86 - można wykorzystywać rejestry głównego procesora).

- **Nie jest możliwe bezpośrednie ładowanie do rejestrów FPU zawartości rejestrów ogólnego przeznaczenia procesora x86** (*%rax, %rdi…*) oraz stałych – liczb podanych jako argument rozkazu (*immediate*).

- FPU posiada natomiast dedykowane instrukcje pozwalające szybko załadować często używane w obliczeniach stałe (zero, jeden, π, e – z odpowiednio dużą dokładnością).

- Jednostka zmiennoprzecinkowa dysponuje swoimi instrukcjami porównań, flagami warunkowymi, a w przypadku błędów danych, niewłaściwych operacji, przepełnienia stosu może generować przerwania-wyjątki.

1. Stos i odwrotna notacja polska.

Osiem 80–bitowych rejestrów danych (*R0* - *R7*) tworzy stos jednostki zmiennoprzecinkowej.

W asemblerze **wierzchołek stosu FPU** – rejestr przechowujący **ostatnio odłożony** na stos element – **oznacza się *ST(0)***. Pozostałe rejestry nazwane są od *ST(1)* do *ST(7)*.

Nowe dane ładowane są (*push*) zawsze na wierzchołek stosu – do *ST(0)* a dotychczasowy *ST(0)* staje się *ST(1)*, analogicznie dalej: *ST(1)* przechodzi do *ST(2)* itd.

Dostęp do wartości w innych rejestrach odbywa się **względem wierzchołka stosu**. Czyli przed–przedostatnia wrzucona na stos wartość będzie w rejestrze *ST(2)*.

Zapis danych z wierzchołka stosu *ST(0)* do pamięci może odbywać się z równoczesnym zdjęciem zapisywanej wartości ze stosu (*pop*) – po tej operacji numery rejestrów przesuwają się „w dół”: *ST(1)* staje się *ST(0)* itd.

W FPU wierzchołek stosu *TOP* adresowany jest trzema bitami (przechowywanymi w *Status Register*).

Jeżeli *TOP* wskazuje na *R0* (000B), przy kolejnym ładowaniu – przepełnieniu stosu nastąpi „zawinięcie” (modulo 8) i *TOP* ustawi się na *R7* (111B). Analogiczna sytuacja może wystąpić przy ściąganiu elementów ze stosu. W zależności od ustawień wyjątków FPU, w przypadku zawinięcia może zostać wygenerowany wyjątek: stack over/underflow).

$$
\begin{array}{lll}
 & R7 & ST(4) \\
 & R6 & ST(3) \\
\uparrow & R5 & ST(2) \\
pop & R4 & ST(1) \\
push & R3 & ST(0) \quad \leftarrow TOP(011_{BIN}) \\
\downarrow & R2 & ST(7) \\
 & R1 & ST(6) \\
 & R0 & ST(5)
\end{array}
$$

Przekształcenia wyrażeń arytmetycznych na kod programu wykorzystującego stos rejestrów można wykonać stosując **odwrotną notację polską** (*Reverse Polish Notation* - RPN, *postfix format*).

Przykład:

wyrażenie: $b / (b + c)$

w RPN ma postać: $b\ c + b /$

Czyli:

1. Wrzuć na stos *b* i *c*.

2. Pobierz ze stosu *b* i *c*, wykonaj dodawanie.

3. Wynik odłóż na stos.

4. Wrzuć na stos *b*.

5. Ściągnij ze stosu dwie ostatnie liczby (*sumę* i *b*) i wykonaj dzielenie.

6. Wynik odłóż na stos.

W asemblerze x87 **może*** to wyglądać następująco (*b*, *c* – adresy-etykiety liczb typu *double*, rozkazy zostały omówione w następnej części, składnia AT&T):

FLDL *b* # załaduj double *b* z pamięci do *ST(0) = b*

	$R7$		$ST(4)$	
	$R6$		$ST(3)$	
↑	$R5$		$ST(2)$	
pop	$R4$		$ST(1)$	
push	$R3$	b	$ST(0)$	$\leftarrow TOP(011_{BIN})$
↓	$R2$		$ST(7)$	
	$R1$		$ST(6)$	
	$R0$		$ST(5)$	

FLDL *c* # załaduj double *c* z pamięci do *ST(0)*: *ST(1) = b*, *ST(0) = c*

	$R7$		$ST(5)$	
	$R6$		$ST(4)$	
↑	$R5$		$ST(3)$	
pop	$R4$		$ST(2)$	
push	$R3$	b	$ST(1)$	
↓	$R2$	c	$ST(0)$	$\leftarrow TOP(010_{BIN})$
	$R1$		$ST(7)$	
	$R0$		$ST(6)$	

FADDP # dodaj: *ST(1) = ST(0) + ST(1)* i ściągnij ze stosu *ST(0)*. *ST(1)* staje się *ST(0)*: *ST(0) = b + c*

	$R7$		$ST(4)$	
	$R6$		$ST(3)$	
↑	$R5$		$ST(2)$	
pop	$R4$		$ST(1)$	
push	$R3$	$b+c$	$ST(0)$	$\leftarrow TOP(011_{BIN})$
↓	$R2$		$ST(7)$	
	$R1$		$ST(6)$	
	$R0$		$ST(5)$	

FLDL *b* # *ST(0) = b , ST(1) = b + c*

	$R7$		$ST(5)$	
	$R6$		$ST(4)$	
↑	$R5$		$ST(3)$	
pop	$R4$		$ST(2)$	
push	$R3$	$b+c$	$ST(1)$	
↓	$R2$	b	$ST(0)$	$\leftarrow TOP(010_{BIN})$
	$R1$		$ST(7)$	
	$R0$		$ST(6)$	

FDIVP # podziel *ST(1) = ST(0)* / *ST(1)* i ściągnij ze stosu *ST(0)*. *ST(1)* staje się *ST(0)*: *ST(0) = b* / (*b + c*)

	$R7$		$ST(4)$	
	$R6$		$ST(3)$	
↑	$R5$		$ST(2)$	
pop	$R4$		$ST(1)$	
push	$R3$	$b/(b+c)$	$ST(0)$	$\leftarrow TOP(011_{BIN})$
↓	$R2$		$ST(7)$	
	$R1$		$ST(6)$	
	$R0$		$ST(5)$	

Przekształcenie z notacji konwencjonalnej (*infix*) na RPN (*postfix*) jest proste i wykorzystywane np. w automatycznym tłumaczeniu kodu źródłowego na ciągu instrukcji (ko)procesora – bez wykorzystywania nawiasów zachowana jest kolejność wykonywania działań.

Więcej przykładów konwersji *infix-postfix* można znaleźć w internecie np. w książce *The Art Of Assembly Language*:

https://www.plantation-productions.com/Webster/www.artofasm.com/Linux/HTML/RealArithmetica3.html#1013285

* Niestety, stosowanie RPN nie zawsze skutkuje otrzymaniem optymalnego kodu (pięć instrukcji, trzy dostępy do pamięci). W powyższym przykładzie zmienna *b* jest ładowana z pamięci do rejestru dwa razy. Można temu zaradzić trzymając wielokrotnie wykorzystywane zmienne na stosie (o ile jest miejsce) – cztery instrukcje, dwa dostępy do pamięci:

```
FLDL    b               # ST(0) = b
FLDL    c               # ST(1) = b, ST(0) = c
FADD    %ST(1) , %ST(0) # ST(0) = ST(1) + ST(0) i niczego nie zdejmuj
FDIVRP                  # dzielenie "odwrotne" ST(1)=ST(1) / ST(0) i dopiero teraz zdejmij; iloraz w ST(0)
```

Można również wartość zmiennej *c* pobrać z pamięci podczas dodawania (też cztery instrukcje, ale niestety ponownie trzy dostępy do pamięci):

```
FLDL    b               # ST(0) = b
FADDL   c               # ST(0) =  ST(0) + c
FLDL    b               # ST(0) = b
FDIVP                   # ST(1)=ST(0) / ST(1) i zdejmij; iloraz w ST(0)
```

Analogicznie: wyrażenie np. (*a***b*)/(*b*+*c*) można wykonać na różne sposoby:

```
FLDL    a
FLDL    b
FMULP
FLDL    b
FLDL    c
FADDP
FDIVRP
```

```
FLDL    a
FLDL    b
FLDL    c
FADD    %ST(1) , %ST(0)
FDIVRP
FMULP
```

2. ŚCIĄGA – do wykorzystania podczas pisania własnych programów.

(Tylko) wybrane instrukcje FPU x87 – składnia AT&T

2.1. Operacje przemienne: FADD, FMUL.

Dodawanie do *ST(0)* liczb typu *float* i *double* z pamięci (adresowanie komórek tak samo jak w x86)

FADD**S** *m32* *ST(0) = ST(0) + float*
FADD**L** *m64* *ST(0) = ST(0) + double*

Dodawanie do *ST(0)* liczb całkowitych ze znakiem (16– lub 32–bitowych) ładowanych z pamięci,
(konwersja do postaci zmiennoprzecinkowej odbywa się „w locie”):

F**I**ADD**S** *m16* *ST(0) = ST(0) + short*
F**I**ADD**L** *m32* *ST(0) = ST(0) + int*

Operacje na rejestrach FPU:

FADD *%ST(0), %ST(i)* *ST(i) = ST(0) + ST(i)*

FADD**P** *%ST(0), %ST(i)* *ST(i) = ST(0) + ST(i)*, potem ściągnij *ST(0)* ze stosu (P–pop): wynik w *ST(i–1)*

FADD *%ST(i)* *ST(0) = ST(0) + ST(i)*; *ST(i)* – dowolny rejestr FPU

FADD**P** *%ST(i)* jak wyżej, potem ściągnij *ST(0)* ze stosu

FADD**P bez argumentów** *ST(1) = ST(0) + ST(1)*, potem ściągnij *ST(0)* ze stosu: wynik będzie w *ST(0)*

Analogicznie działają rozkazy mnożenia F(I)MUL(R)(P).

2.2. Operacje nieprzemienne: odejmowanie i dzielenie zostały zaimplementowane w wersjach zwykłej i „odwrotnej” (R).

FSUB**S** *m32* *ST(0) = ST(0) – float*
FSUB**L** *m64* *ST(0) = ST(0) – double*

FSUB**RS** *m32* *ST(0) = float – ST(0)*
FSUB**RL** *m64* *ST(0) = double – ST(0)*

W ten sam sposób działa rodzina instrukcji F**I**SUB(R)(S/L) – przed odejmowaniem konwertując odpowiednio typy 16– i 32–bitowe (*short* i *int*).

Operacje odejmowania na rejestrach FPU:

FSUB *%ST(i)* *ST(0) = ST(0) – ST(i)*
FSUB**R** *%ST(i)* *ST(0) = ST(i) – ST(0)*

FSUB**P** *%ST(i)* *ST(i) = ST(0) – ST(i)*, potem ściągnij *ST(0)* ze stosu (P–pop): wynik w *ST(i–1)*
FSUB**RP** *%ST(i)* *ST(i) = ST(i) – ST(0)*, potem ściągnij *ST(0)* ze stosu (P–pop): wynik w *ST(i–1)*

FSUB *%ST(i), %ST(0)* *ST(0) = ST(0) – ST(i)*
FSUB**R** *%ST(i), %ST(0)* *ST(0) = ST(i) – ST(0)*

Rozkazy odejmowania bez argumentów:

FSUBP *ST(1) = ST(0) – ST(1)*, potem ściągnij *ST(0)* ze stosu (P-pop): wynik w *ST(0)*
FSUBRP *ST(1) = ST(1) – ST(0)*, potem ściągnij *ST(0)* ze stosu (P-pop): wynik w *ST(0)*

Analogicznie wygląda składnia rozkazów dzielenia F(I)DIV(R)(P).

Niestety, w asemblerach używających składni AT&T jest błąd – niekonsekwencja:

https://sourceware.org/binutils/docs/as/i386_002dBugs.html

dlatego też, instrukcje mające jako pierwszy operand (źródłowy) *ST(0)* i drugi *ST(i)* działają tak:

FSUB	*%ST(0), %ST(i)*	*ST(i) = ST(0) – ST(i)*
FSUBR	*%ST(0), %ST(i)*	*ST(i) = ST(i) – ST(0)*
FSUBP	*%ST(0), %ST(i)*	*ST(i) = ST(0) – ST(i)*, potem ściągnij *ST(0)* ze stosu (P–pop): wynik w *ST(i–1)*
FSUBRP	*%ST(0), %ST(i)*	*ST(i) = ST(i) – ST(0)*, potem ściągnij *ST(0)* ze stosu (P–pop): wynik w *ST(i–1)*

2.3 Instrukcje jednoargumentowe (np. FABS – wartość bezwzględna, FSQRT – pierwiastek kwadratowy, FCHS – zmiana znaku) jak również instrukcje **ładujące często używane w obliczeniach stałe** (np. 0 – FLDZ, 1 – FLD1, π – FLDPI) operują **tylko na wierzchołku** stosu (ładowanie stałych do *ST(0)*, odczyt – modyfikacja – zapis *ST(0)* itp.) i nie wymagają podawania argumentu po mnemoniku rozkazu.

Pełny wykaz tego typu instrukcji zawarty jest w dokumentacji firmy Intel.

2.4 Transfer danych.

Ładowanie liczb całkowitych ze znakiem – na wierzchołek stosu (do *ST(0)*, reszta przesuwa się o jeden dalej)

Adresowanie pamięci wygląda tak samo, jak w przypadku typowych instrukcji x86, odczytane z pamięci liczby całkowite są automatycznie konwertowane do formatu *double extended precision* (80–bitowy).

FILD**S**	*m16*	(*short*)
FILD**L**	*m32*	(*int*)
FILD**Q**	*m64*	(*long*)

Ładowanie typów zmiennoprzecinkowych z pamięci:

FLD**S**	*m32*	(*float*)
FLD**L**	*m64*	(*double*)
FLD**T**	*m80*	(*long double*)

Załadowanie rejestru *ST(i)* na wierzchołek stosu *ST(0)*:

FLD	*ST(i)*

Zapis danych będących w *ST(0)* – jako liczby całkowite ze znakiem (konwersja automatyczna):

F**I**ST**S**	*m16*	(*short*)
F**I**ST**L**	*m32*	(*int*)

Zapis danych z *ST(0)* jako liczby całkowitej ze znakiem i następnie ściągnięcie jej ze stosu (dotychczasowe *ST(1)* staje się *ST(0)* itd.):

F**I**ST**PS**	*m16*	(*short*)
F**I**ST**PL**	*m32*	(*int*)
F**I**ST**PQ**	*m64*	(*long*)

Zapis danych zmiennoprzecinkowych z *ST(0)* do pamięci:

FST**S**	*m32*	(*float*)
FST**L**	*m64*	(*double*)

i wersje ze ściąganiem *ST(0)* ze stosu:

FST**PS**	*m32*	(*float*)
FST**PL**	*m64*	(*double*)
FST**PT**	*m80*	(*long double*)

Kopiowanie zawartości *ST(0)* do innego rejestru *ST(i):*

FST(P) *ST(i)*

3. Porównania liczb zmiennoprzecinkowych.

W przeciwieństwie do porównywania liczb całkowitych, w przypadku reprezentacji zmiennoprzecinkowej może zaistnieć sytuacja, gdy jeden (lub oba) z argumentów to *NaN* (*Not a Number*, patrz wykład). W takiej sytuacji, *NaN*, który (z definicji!) nie oznacza liczby – nie może być porównany i określony jako „mniejszy", „większy" czy równy innej liczbie. Podczas programowania FPU x87 sytuacja, gdy co najmniej jeden z porównywanych argumentów to *NaN* nosi nazwę *unordered*.

W wyniku wykonania jednej z instrukcji porównujących (np. FCOM, FTST) FPU, podobnie jak „zwykły" procesor, ustawia cztery flagi warunkowe (*C0* - *C3*).

Ponieważ pierwotnie FPU x87 była odrębnym układem scalonym, flagi te znajdują się nie w rejestrze flag (R/E)FLAGS procesora x86, a w dedykowanym rejestrze FPU – *Status Register* (16–bitowym).
Układ „bitowy" flag warunkowych jednak odpowiada układowi flag arytmetycznych procesora głównego:

C0 –> Carry
C2 –> Parity
C3 –> Zero

Aby móc korzystać np. ze skoków warunkowych, trzeba przekopiować flagi z rejestru statusu FPU do rejestru flag procesora x86 (EFLAGS), np. sekwencją rozkazów:

```
FCOM          # bez argumentów: porównuje ST(0) z ST(1)
FSTSW %ax     # zapisz Status Word do rej. %ax (16–bit)
SAHF          # zapisz (wybrane) bity z %ah do rejestru flag procesora x86
```

Teraz już można „normalnie" używać zwykłych skoków warunkowych, np.:

ST(0) > ST(1) JA (JNBE)
ST(0) < ST(1) JB (JNAE)
ST(0) = ST(1) JE (JZ)

Instrukcja z rodziny FCOM może również przyjmować za argument:

– liczbę zmiennoprzecinkową odczytaną z pamięci (np. porównanie z zmienną *x* typu *double*: FCOML *x*),
– liczbę całkowitą 16– lub 32–bitową umieszczoną w pamięci,
– dowolny rejestr FPU *ST(i)*,

może również ściągnąć ze stosu jedną (FCOMP) lub obie porównywane liczby (FCOMPP).

Kopiowanie flag z FPU stanowi dodatkowy narzut czasowy (kłopotliwe może być również nadpisanie stanu „oryginalnych" flag procesora). Od architektury P6 (Pentium Pro), czyli już po zintegrowaniu FPU z CPU, dla programisty dostępne są instrukcje*:

FCOMI / FCOMIP - porównujące *ST(0)* z *ST(1)* i od razu ustawiające flagi głównego procesora (druga ściąga ze stosu *ST(0)*).

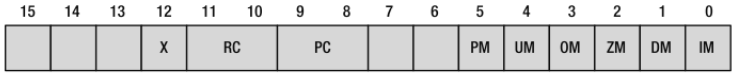
Inne porównania, jak FTST (porównanie *ST(0)* z zerem), jak i warunkowy transfer danych między rejestrami ST(i) a ST(0) (FMOVcc) opisane zostało w dokumentacji.

* z innych „ciekawostek": rejestry rozszerzenia wektorowego MMX (*%MM0 – %MM7*) wykorzystują fizycznie stos rejestrów x87, co może sprawiać problemy przy „mieszaniu" instrukcji FPU z wektorowymi - MMX…

4. Control and Status Registers: precyzja obliczeń, sposób zaokrągleń i zgłaszanie wyjątków

Spośród kilku rejestrów specjalnego przeznaczenia (patrz: dokumentacja FPU) dwa wymagają uwagi:

Control Register zawierający **Control Word** – konfigurację FPU:



Najmłodsze 6 bitów zawiera maski odpowiadające za włączenie (0) lub wyłączenie (1 – maskowanie) zgłaszania wyjątków, np. (tylko niektóre):

IM – *Invalid operation Mask* (przepełnienie „zawinięcie" stosu, zły format danych itp.),
ZM – dzielenie przez – *Zero Mask*,
UM/OM – *Underflow / Overflow Mask* – wystąpienie niedomiaru / nadmiaru.

Wyjątek, w uproszczeniu, jest rodzajem przerwania – sygnału powiadamiającego procesor o zajściu zdarzenia wymagającego (z reguły szybkiej) obsługi. Ponieważ przerwania generalnie mają wyższy priorytet niż program użytkownika, jest on (jak sama nazwa wskazuje) przerywany, a procesor przechodzi do wykonywania tzw. procedury obsługi przerwania (wyjątku). Po powrocie – o ile jest taka możliwość – program użytkownika zostaje kontynuowany.

Uwaga: wystąpienie wyjątku (włączonego, „odmaskowanego") powoduje ustawienie flagi ES (Exception Summary) w **Status Register** FPU (opis poniżej). Procesor sprawdza stan flagi ES podczas wykonywania **następnej** instrukcji FPU (lub synchronizującej np. FWAIT) – i dopiero wtedy zostanie wywołana procedura obsługi wyjątku.

PC – *Precision Control* – dwubitowe pole ustala precyzję, z jaką wykonywane są obliczenia:

00 – single (float, 32 bity),
10 – double (64 bity),
11 – double extended (80 bitów).

RC – *Rounding Control* – dwubitowe pole ustalające sposób zaokrąglania.

Przykład: zaokrąglanie z 4 do 3 bitów części ułamkowej:

1.0111B = 1.4375 (DEC)

Bity RC	Wynik dokładny	Zaokrąglenie
00 round to nearest even	1.0111B = 1.4375	1.100B = 1.5
01 round down toward +	1.0111B = 1.4375	1.011B = 1.375
10 round up toward +	1.0111B = 1.4375	1.100B = 1.5
11 truncate	1.0111B = 1.4375	1.011B = 1.375

–1.0111B = –1.4375 (DEC)

Bity RC	Wynik dokładny	Zaokrąglenie
00 round to nearest even	-1.0111B = -1.4375	-1.100B = -1.5
01 round down toward +	-1.0111B = -1.4375	-1.100B = -1.5
10 round up toward +	-1.0111B = -1.4375	-1.011B = -1.375
11 truncate	-1.0111B = -1.4375	-1.011B = -1.375

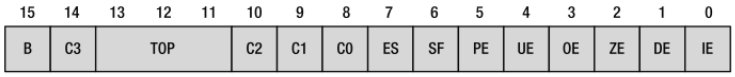
Uwaga: przed pierwszym użyciem w programie FPU należy włączyć instrukcją FINIT (FNINIT).

Wartość *Control Word* ustawiona jest domyślnie na: 0x037F: 0000 00 11 01 111111B.

czyli: zgłaszanie wyjątków jest wyłączone, ustawiona największa precyzja i zaokrąglanie „*to nearest even*".

Wartość *Control Word* można załadować z *Control Register* do pamięci RAM, a po modyfikacji zapisać ponownie do FPU instrukcjami: odpowiednio FSTCW *m16* i FLDCW *m16*. Zmodyfikować należy tyko niezbędne ustawienia. Bity niewykorzystane (6, 7, 13–15) należy pozostawić w stanie niezmienionym.

Status Register - rejestr stanu FPU:



Najmłodsze 6 bitów – flagi wyjątków (mogą być maskowane przez odpowiadające im bity *Conrol Word*),
C0 – C3: flagi warunkowe – ustawiane na podstawie wyników porównań,
TOP – adres wierzchołka stosu (jak na schematach pokazujących operacje na stosie).

Zawartość *Status Register* można zapisać (do pamięci lub rejestru *%ax*) instrukcją FSTSW, natomiast stan wszystkich rejestrów FPU (108 bajtów) – instrukcjami FSAVE / FRSTOR (np. przy przełączaniu zadań przez system operacyjny).

Pełny opis ww. rejestrów oraz mechanizmu synchronizacji zgłaszania wyjątków zawarty jest w dokumentacji firmy Intel (przytoczonej na pierwszej stronie).

5. Alokacja pamięci na zmienne o różnej precyzji w GNU asemblerze:

```
.data
x:     .float       3.14   # 4 bajty
y:     .double      2.72   # 8 bajtów
z:     .tfloat      4.86   # 10 bajtów

str_f: .string      “float_x = %f \n”
str_d: .string      “double_x = %lf \n”
str_t: .string      “long_double_x = %Lf \n” # (albo w postaci wykładniczej: %Le)
```

6. Przekazywanie argumentów typów zmiennoprzecinkowych w funkcjach języka C:

Zgodnie z ABI, argumenty typu *float* i *double* przekazuje się w rejestrach XMM (128-bitowe rejestry rozszerzenia wektorowego SSE).

Wyświetlenie liczby typu *float*:

```
movss       x , %xmm0           # Move Scalar Single precision,  można użyć rozkazu movd
cvtss2sd    %xmm0 , %xmm0       # Convert Scalar Single precision to Scalar Double precision

mov         $str_f , %rdi
mov         $1 , %eax           # używamy jednego XMMa – więc %eax = 1
call        printf
```

–> Zamiast `mov $1,%eax` można wykonać: `xor %eax,%eax  inc %eax`

Wyświetlenie liczby typu *double*:

```
movsd       y , %xmm0           # Move Scalar Double precision,  można użyć movq

mov         $str_d , %rdi
mov         $1 , %eax
call        printf
```

Dla odmiany, zgodnie z ABI x86–64, typy *long double* przekazuje się na stosie:

Liczby te są 10–bajtowe, należy więc na stosie „zarezerwować” 16 bajtów:

```
sub         $16 , %rsp          # przesuń wskaźnik stosu o 16 bajtów w dół
fstpt       (%rsp)              # zapisz wynik obliczeń: ściągnij ze stosu FPU, wrzuć na stos CPU

mov         $str_t , %rdi
xor         %eax , %eax         # teraz nie używamy XMMów, %eax = 0
call        printf
add         $16 , %rsp          # ”usuń” argument ze stosu
```

Uwaga: przed wywołaniem funkcji z bibliotek języka C operujących na typach zmiennoprzecinkowych, koniecznie pilnować by stos był wyrównany do granicy 16 bajtów (zależy od wersji GCC).