ROZDZIAŁ CZTERNASTY: **ARYTMETYKA ZMIENNO PRZECINKOWA**

Chociaż liczby całkowite dostarczają dokładnej reprezentacji dla wartości liczbowych, cierpią one z powodu dwóch głównych wad: niemożność przedstawiania liczb ułamkowych i ograniczenie zakresu dynamiki. Arytmetyka zmienno przecinkowa rozwiązuje te dwa problemy kosztem dokładności i, na niektórych procesorach . szybkości. Większość programistów jest świadomych utraty szybkości związanej z arytmetyką zmienno przecinkową; jednakże są beztrosko nieświadomi problemów z dokładnością.

Dla wielu aplikacji korzyści ze zmienno przecinkowości przeważają nad wadami. Jednakże dla właściwego zastosowania arytmetyki zmienno przecinkowej w każdym programie musimy nauczyć się jak działa arytmetyka zmienno przecinkowa. Intel , rozumiejąc znaczenie arytmetyki zmienno przecinkowej w nowoczesnych programach dostarczył wsparcia dla arytmetyki zmienno przecinkowej w najwcześniejszym swoim projekcie 8086 – FPU 8087 (jednostka zmienno przecinkowa lub koprocesor matematyczny). Jednak na procesorach wcześniejszych niż 80486 (lub na 80486sx) procesor zmienno przecinkowy jest urządzeniem opcjonalnym; to znaczy, że nieobecne urządzenie musi być zasymulowane programowo.

Rozdział ten zawiera cztery główne sekcje. Pierwsza sekcja omawia arytmetykę zmienno przecinkową z punktu widzenia matematycznego. Druga sekcja omawia binarną postać zmienno przecinkowości powszechnie używanej w procesorach Intela. Trzecia omawia zmienno przecinkowość programową i podprogramy matematyczne ze Standardowej Biblioteki UCR. Czwarta omawia chip 80x87 FPU.

14.0 WSTĘP

Ten rozdział zawiera cztery główne sekcje: opis formatu zmienno przecinkowego i operacji (dwie sekcje), omówienie wsparcia zmienno przecinkowości w Bibliotece Standardowej UCR i omówienie 80x87 FPU (jednostki zmienno przecinkowej). Poniższe sekcje które maja znak „•” są niezbędne. Sekcje z „⊗” omawiają tematy zaawansowane, które możemy chcieć opuścić.

- Matematyczna arytmetyka zmienno przecinkowa
- Formaty zmienno przecinkowe IEEE
- Podprogramy zmienno przecinkowe Biblioteki Standardowej UCR
- Koprocesor zmienno przecinkowy 80x87
- Instrukcje przesuwania danych FPU

⊗ Konwersje

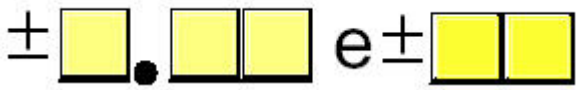
- Instrukcje arytmetyczne
- Instrukcje porównań

⊗ Instrukcje stałe
⊗ Instrukcje transcendentalne
⊗ Instrukcje różnorodne
⊗ Operacje całkowite
⊗ Dodatkowe operacje trygonometryczne

14.1 MATEMATYCZNA ARYTMETYKA ZMIENNO PRZECINKOWA

Dużym problem z arytmetyką zmienno przecinkową jest taki, że nie stosuje standardowych zasad. Niemniej jednak wielu programistów stosuje normalne zasady algebraiczne kiedy stosuje arytmetykę zmienno przecinkową. Jest to źródło błędów w wielu programach. Jednym z podstawowych celów tej sekcji jest opisanie ograniczeń arytmetyki zmienno przecinkowej, więc zrozumiemy jak stosować ją właściwie.

Zwykłe zasady algebraiczne stosują tylko arytmetykę o nieskończonej precyzji. Rozważmy prostą instrukcję x: = x +1, x jest liczbą całkowitą. Na nowoczesnych komputerach ta instrukcja będzie korzystała ze zwykłych zasad algebry , tak długo dopóki nie wystąpi przepełnienie. To znaczy ta instrukcja jest poprawna tylko



Rysunek 14.1: Prosty format zmienno przecinkowy

dla pewnych wartości x (minint <= x < maxint). Większość programistów nie ma z tym problemu, ponieważ są świadomi faktu, że liczby całkowite w programie nie stosują się do standardowych zasad algebraicznych (np. 5 / 2 ≠ 2.5).

Liczby całkowite nie stosują się do standardowych zasad algebry ponieważ ich komputerowa reprezentacja ma skończoną liczbę bitów. Nie możemy przedstawić żadnej (całkowitej) wartości powyżej maksymalnej liczby całkowitej lub poniżej minimalnej liczby całkowitej. Wartości zmienno przecinkowe cierpią na ten sam problem, tylko gorzej. Jednak liczby całkowite są podzbiorem liczb rzeczywistych. Dlatego też wartości zmienno przecinkowe muszą przedstawiać taki sam nieskończony zbiór liczb całkowitych. Jednak jest nieskończona liczba wartości pomiędzy dwoma wartościami rzeczywistymi, więc ten problem jest nieskończenie gorszy. Dlatego też mając ograniczone wartości pomiędzy zakresem maksymalnym a minimalnym, nie możemy przedstawić również wszystkich wartości pomiędzy tymi dwoma zakresami.

Do przedstawienia liczb rzeczywistych większość formatów zmienno przecinkowych stosuje notację naukową i używa jakiejś liczby bitów do przedstawiania mantysy i mniejszej liczby bitów do przedstawienia wykładnika. Końcowy rezultat jest taki, że liczba zmienno przecinkowa może tylko przedstawiać liczby z określoną liczbą znaczących cyfr. Ma to duży wpływ na to jak działa arytmetyka zmienno przecinkowa. Łatwo zobaczymy wpływ arytmetyki o ograniczonej precyzji, kiedy przyjmiemy uproszczony format dziesiętny zmienno przecinkowy dla naszych przykładów. Nasz format zmienno przecinkowy dostarczy mantysy z trzema znaczącymi cyframi i dziesiętny wykładnik z dwoma cyframi. Mantysy i wykładniki są wartościami ze znakiem (zobacz rysunek 14.1)

Kiedy dodajemy i odejmujemy dwie liczby w notacji naukowej, musimy zmodyfikować dwie wartości, żeby ich wykładniki były takie same. Na przykład, kiedy dodajemy 1.23e1 i 4,56e0, musimy zmodyfikować wartości tak, żeby miały takie same wykładniki. Jednym sposobem zrobienia jest tego jest skonwertowanie 4.56e0 do 0.456e1 a potem je dodać. Todaje1.686e1. Niestety wynik nie mieści się w trzech znaczących cyfrach, wiec musimy zaokrąglić lub skrócić wynik do trzech znaczących cyfr. Zaokrąglanie, generalni, tworzy bardziej precyzyjny wynik, więc zaokrąglamy wynik do 1.69e1. Jak widzimy, brak precyzji (liczba cyfr lub bitów w obliczeniu) wpływa na dokładność (poprawność obliczenia)

W poprzednim przykładzie mogliśmy zaokrąglić wynik ponieważ mieliśmy cztery znaczące cyfry podczas obliczania. Jeśli nasze obliczenia zmienno są ograniczone do trzech znaczących liczb podczas obliczania, musimy obciąć ostatnią cyfrę mniejszej liczby uzyskując 1.86e1, które jest mniej poprawne. Dodatkowa cyfra dostępna podczas obliczania jest znana jako pozycja chroniona wyniku (lub bit zabezpieczenia w przypadku formatu bitowego). One wielce podnoszą dokładność podczas długiego szeregu obliczeń.

Strata dokładności podczas pojedynczego obliczenia zazwyczaj nie jest taka aby się martwić , chyba ,że martwisz się rzeczywiście precyzją swoich obliczeń. Jednakże jeśli obliczamy wartość , która jest wynikiem sekwencji operacji zmienno przecinkowych, błąd może się gromadzić wielce wpływając na sam wynik. Przypuśćmy na przykład, że dodamy 1.23e3 z 1.00e0. Modyfikując liczby tak, żeby ich wykładniki były takie same przed dodawaniem uzyskujemy 1.23e3 + 0.001e3 . Suma tych dwóch wartości , nawet po zaokrągleniu to 1.23e3. To może wydać się nam zupełnie racjonalne; w końcu możemy tylko zajmować się trzema znaczącymi cyframi, dodanie małej wartości nie powinno wcale wpłynąć na wynik. Jednak przypuśćmy, że będziemy dodawać 1.00e0 do 1.23e3 dziesięć razy. Pierwszy raz dodając 1.00e0 do 1.23e3 dostajemy 1.23e0. Podobnie uzyskamy taki sam wynik za drugim, trzecim, czwartym...dziesiątym razem. Z drugiej strony dodając 1.00e0 do samej siebie dziesięć razy , a potem dodając wynik (1.00e1) do 1.23e3 otrzymamy inny wynik , 1.24e3. Jest ważna rzecz do zapamiętania w arytmetyce o ograniczonej precyzji :

Porządek wyliczenia może wpływać na precyzję wyniku

Uzyskamy bardziej dokładny wynik jeśli odpowiednie wielkości (to znaczy wykładniki) są bliżej jeden drugiego Jeśli wykonujemy szereg operacji wymagających dodawania i odejmowania, powinniśmy zgrupować właściwe wartości.

Inny problem z dodawaniem i odejmowaniem jest taki, że możemy skończyć z fałszywą precyzją. Rozpatrzmy obliczenie 1.23e0 – 1.22e0. Daje to 0.01e0. Chociaż jest to równoważne matematycznie 1.00e-2, ta druga postać wskazuje na to, że ostatnie dwie cyfry są dokładnie zerami. Niestety mamy tylko pojedynczą znaczącą cyfrę tym razem. Rzeczywiście, niektóre FPU lub pakiet oprogramowania zmienno przecinkowego może rzeczywiście mogą wprowadzać losowe cyfry lub bity na najmniej znaczące pozycje. Jest to druga ważna zasada dotycząca arytmetyki o ograniczonej precyzji:

> Kiedykolwiek odejmujemy dwie liczby z takimi samymi znakami lub dodajemy dwie liczby z różnymi znakami, precyzja wyniku może być mniejsza niż precyzja dostępna w formacie zmienno przecinkowym.

Mnożenie i dzielenie nie cierpi z tego samego powodu co dodawanie i odejmowanie ponieważ nie musimy modyfikować wykładników przed tymi działaniami; wszystko co musimy zrobić to dodać wykładniki i pomnożyć mantysy (lub odjąć wykładniki i podzielić mantysy). Mnożenie i dzielenie nie tworzą szczególnie słabych wyników. Jednakże mają one skłonności do zwielokrotniania błędów , które już istnieją w wartości. Na przykład, jeśli pomnożymy 1.23e0 przez dwa, kiedy powinniśmy pomnożyć 1.24e0 przez dwa, wynik jest tym bardziej niedokładny. To daje nam trzecią ważną zasadę kiedy pracujemy z arytmetyką o ograniczonej precyzji:

> Kiedy wykonujemy łańcuch obliczeń wymagających dodawania, odejmowania mnożenia i dzielenia, próbujemy najpierw wykonać mnożenie i dzielenie.

Często, przez zastosowanie normalnych transformacji algebraicznych, możemy ułożyć obliczenia tak, że mnożenie i dzielenie wystąpią najpierw. Na przykład ,przypuśćmy, że chcemy obliczyć x* (y+z). Zwykle dodajemy razem y i z a potem ich sumę mnożymy przez z. Jednakże uzyskamy trochę bardziej dokładny wynik jeśli przetransformujemy x*(y+z) do x*y + x*z i obliczymy wynik, najpierw obliczając mnożenie.

Mnożenie i dzielenie też nie są pozbawione problemów. Kiedy mnożymy dwie bardzo duże lub bardzo małe liczby, jest całkiem możliwe wystąpienie przepełnienia lub niedomiaru. Taka sama sytuacja wystąpi kiedy dzielimy małą liczbę przez dużą lub dużą przez małą. Daje to nam czwartą zasadę , którą powinniśmy próbować stosować kiedy mnożymy lub dzielimy wartości:

> Kiedy mnożymy lub dzielimy zbiór liczb, próbujmy ułożyć mnożenia tak, żeby mnożyć duże i małe liczby razem; podobnie próbujmy dzielić liczby, które maja takie same względne wartości.

Porównywanie liczb zmienno przecinkowych jest bardzo niebezpieczne. Ze względu na nieścisłości obecne w obliczeniach (wliczając konwersję ciągu wejściowego na wartość zmienno przecinkową) nie powinniśmy nigdy porównywać dwóch wartości zmienno przecinkowych aby zobaczyć czy są równe. W binarnym formacie zmienno przecinkowym , różne obleczenia , które tworzą taki sam wynik (matematyczny) mogą różnić się w swoich najmniej znaczących bitach. Na przykład dodając 1.31e0 + 1.69e0 powinniśmy otrzymać 3.00e0. podobnie dodając 2.50e0 + 0.50e0 powinniśmy otrzymać 3.00e0. Jednakże porównując (1.31e0+1.69e0) i (2.50e0 + 0.50e0) możemy odkryć, że sumy te nie są równe jedna drugiej. Test dla równości jest pozytywny wtedy i tylko wtedy kiedy wszystkie bity (lub cyfry) w dwóch argumentach są takie same. Ponieważ nie jest to koniecznie prawda ,po dwóch różnych obliczeniach zmienno przecinkowych , które powinny tworzyć taki sam wynik, prosty test na równość może nie działać.

Standardowym sposobem dla sprawdzenia równości miedzy liczbami zmienno przecinkowymi jest określenie na jaki błąd (lub jaką tolerancję) pozwolimy w porównaniu i sprawdzamy czy jedna wartość znajduje się wewnątrz zakresu błędu innej. Prostym sposobem zrobienia tego jest użycie testu takiego jak poniższy:

```
if Value1  >=  (value2 –  błąd)  i Value1 <= (Value2 + błąd)  then  ....
```

innym popularnym sposobem wykonania tego samego porównania jest zastosowanie instrukcji w postaci:

```
if   abs (Value1 – Value2) <= błąd then....
```

Większość tekstów które omawiają porównania zmienno przecinkowe zatrzymuje się bezpośrednio po omówieniu problemu równości , zakładając, że inne formy porównań są zupełnie OK. dla liczb zmienno przecinkowych. To nie jest prawda! Jeśli założymy, że x = y i jeśli x jest wewnątrz y ± błąd, wtedy proste porównanie na poziomie bitowym x i y określi, że x < y jeśli y jest większe niż x ale mniejsze niż y ± błąd. Jednakże w takim przypadku x powinno być potraktowane rzeczywiście jako równe y , nie mniejsze niż y. Dlatego też musimy zawsze porównywać dwie liczby zmienno przecinkowe przy użyciu zakresów, bez względu na rzeczywiste porównanie jakie chcemy wykonać. Próbując porównywać dwie liczby zmienno przecinkowe bezpośrednio może prowadzić do błędu. Dla porównania dwóch liczb zmienno przecinkowych xi y, powinniśmy użyć jedną z poniższych form:

```
=     if  abs(x-y) <= błąd then....
≠     if  abs(x-y) > błąd then.....
<     if (x-y)  < błąd then.....
≤     if (x-y)  <= błąd then...
```

> if (x-y) > błąd then....

≥ if (x-y) >= błąd then...

Musimy postępować ostrożnie kiedy wybieramy wartość błędu. Powinna to być wartość odrobinę większa niż największa ilość błędów, jakie wkradną się do naszych obliczeń. Dokładna wartość będzie zależała od określonego formatu zmienno przecinkowego jakiego używamy, ale więcej o tym, trochę później. Końcową zasadą tej sekcji jest:

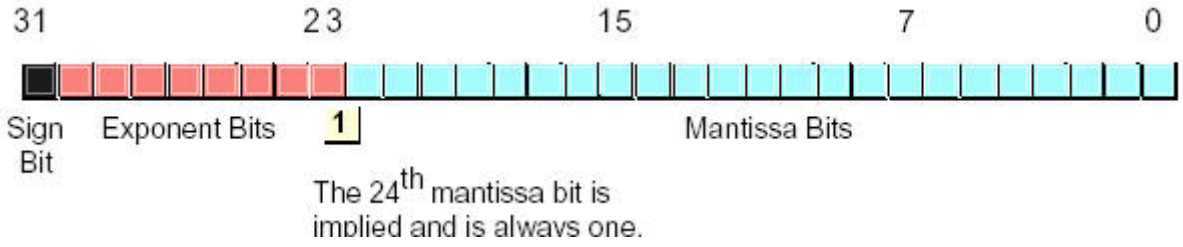
> Kiedy porównujemy dwie liczby zmienno przecinkowe, zawsze porównujemy jedną wartość aby zobaczyć czy jest ona w zakresie danym przez drugą wartość, plus minus jakaś mała wartość błędu.

Jest wiele innych problemów, które mogą wystąpić kiedy stosujemy wartości zmienno przecinkowe. Ten tekst może tylko wskazać główne problemy i uświadomić na fakt, że nie możemy traktować arytmetyki zmienno przecinkowej tak jak rzeczywistej arytmetyki – niedokładności obecne w arytmetyce o ograniczonej precyzji mogą spowodować wiele kłopotów jeśli nie będziemy ostrożni. Dobry tekst o analizie numerycznej lub obliczeniach naukowych może pomóc wypełnić szczegóły, które są poza zakresem tego tekstu. Jeśli będziemy pracowali z arytmetyką zmienno przecinkowa, w jakimś języku, powinniśmy znaleźć czas na przestudiowanie wpływu arytmetyki o ograniczonej na nasze obliczenia.

14.2 FORMAT ZMIENNO PRZECINKOWY IEEE

Kiedy Intel planował wprowadzenie koprocesora zmienno przecinkowego dla swoich nowych mikroprocesorów 8086, dość elegancko zrealizowali to inżynierowie elektrycy i fizycy, którzy zaprojektowali chipy, być może nie najlepsi ludzie do wykonania koniecznej analizy numerycznej i do wybrania najlepszej możliwie binarnej reprezentacji dla formatu zmienno przecinkowego. Więc Intel wynajął najlepszego analityka numerycznego, który mógł znaleźć pomysł na format zmienno przecinkowy dla ich FPY 8087.Osoba ta wynajęła innych ekspertów w terenie a trzech z nich (Kahn, Coonan i Stone) zaprojektowali Intelowski format zmienno przecinkowy. Wykonali tak dobrą robotę projektując Standard Zmienno Przecinkowy KCS, że organizacja IEE zaadoptowała go dla formatu zmienno przecinkowego IEEE.

Do wymaganego działania w szerokim zakresie wydajności i precyzji Intel w rzeczywistości wprowadził trzy formaty zmienno przecinkowe: o pojedynczej precyzji, podwójnej precyzji i precyzji podwyższonej. Pojedyncza i podwójna precyzja odpowiadają typom float i double z C lub typom real i double z FORTRAN'a. Intel zmierzał do użycia precyzji podwyższonej dla długiego łańcucha obliczeń. Podwyższona precyzja zawiera 16 dodatkowych bitów które



Rysunek 14.2: Bity formatu pojedynczej precyzji zmienno przecinkowej

dla obliczenia mogą być używane jako bity zabezpieczenia przed zaokrąglaniem w dół do wartości podwójnej precyzji, kiedy przechowują wynik.

Format pojedynczej precyzji używa uzupełnionej jedynkami 24 bitowej mantysy i ośmiu bitów nadwyżki – 128 wykładnika. Mantysa zazwyczaj przedstawia wartość pomiędzy 1.0 a 2.0. Najbardziej znaczący bit mantysy jest zazwyczaj jedynką i przedstawia wartość na lewo od przecinka dwójkowego. Pozostałe 23 bity mantysy pojawiają się na prawo od przecinak dwójkowego. Dlatego mantysa reprezentuje wartość :

1.mmmmmmm mmmmmmmm mmmmmmmm

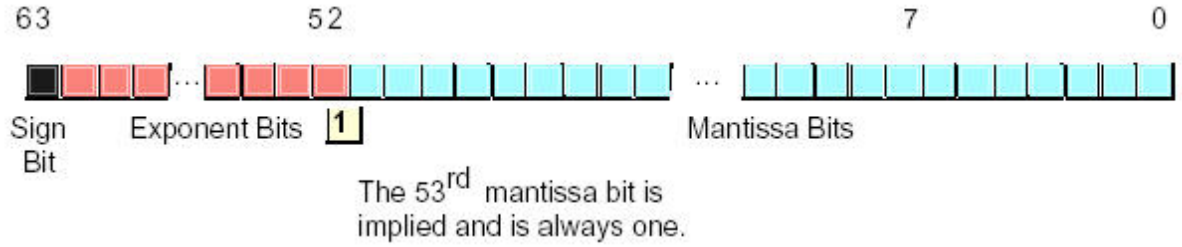
Znaki „mmmmmm....” przedstawiają 23 bity mantysy. Zapamiętajmy, że pracujemy tu z liczbami binarnymi. Dlatego każda pozycja na prawo od przecinka binarnego przedstawia wartość (zero lub jeden) razy następujące po sobie ujemne potęgi dwójki. Jeden założony bit jest zawsze mnożony przez 2^0, czyli jeden. Jest tak dlatego, że mantysa jest zawsze większa niż lub równa jeden. Nawet jeśli wszystkie bity mantysy są zerami, założony jeden bit zawsze daje nam zero. Oczywiście, nawet jeśli mieliśmy prawie nieskończoną liczbę bitów jeden po przecinku binarnym nie mogą one zsumować się do dwóch. Jest tak ponieważ mantysa może reprezentować wartości w zakresie od jeden do dwóch.

Chociaż jest nieskończona liczba wartości pomiędzy jeden i dwa ,możemy tylko przedstawić osiem milionów z nich ponieważ mamy 23 bity mantysy (24 bity to zawsze jeden). To jest powód niedokładności w arytmetyce zmienno przecinkowej – jesteśmy ograniczeni do 23 bitowej precyzji w obliczeniach wymagających wartości o pojedynczej precyzji zmienno przecinkowej.

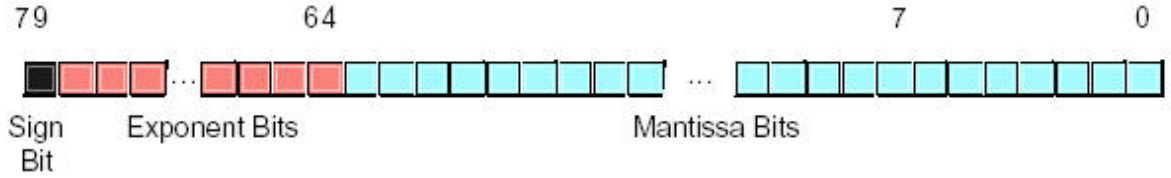
Mantysa używa formatu uzupełnienia jedynkami zamiast uzupełnienia do dwóch. To znaczy, że 24 bitowa mantysy jest po prostu bez znakową liczbą binarną a bit znaku określa czy wartość ta jest dodatnia czy ujemna. Liczby uzupełnione jedynkami mają niezwykła właściwość, którą są dwie reprezentacje zera 9 z ustawionym bitem znaku i wyzerowanym) . generalnie jest to ważne tylko dla osób projektujących oprogramowanie zmienno przecinkowe lub system sprzętowy. My założymy, że wartość zera ma zawsze wyzerowany znak bitu.

Przy przedstawianiu wartości poza zakresem 1.0 do 2.0, wchodzi do gry część wykładnika formatu zmienno przecinkowego. Format zmienno przecinkowy podnosi dwa do potęgi określonej przez wykładnik a potem mnoży mantysę przez tą wartość. Wykładnik jest ośmio bitowy i jest przechowywany w formacie nadmiarowym-127. W formacie tym wykładnik 2^0 jest reprezentowany przez wartość 127 (7fh). Dlatego też konwersja wykładnika do formatu nadmairu-127 to po prostu dodanie 127 do wartości wykładnika. Stosowanie formatu nadmiaru-127 czyni łatwiejszym porównywanie wartości zmienno przecinkowych. Format pojedynczej precyzji zmienno przecinkowej przybiera postać pokazaną na rysunku 14.2

Z 24 bitową mantysą możemy uzyskać w przybliżeniu uzyskać precyzję 6- ½ cyfr (połówka precyzji cyfry oznacza, że pierwsze sześć cyfr może być w zakresie 0..9 ale siódma cyfra może być tylko w zakresie 0...x gdzie x < 9 i generalnie jest blisko pięć). Z ośmio bitowym wyk³adnikiem nadmiaru-128



Rysunek 14.3: Format 64 bitowej podwójnej precyzji zmienno przecinkowej



Rysunek 14.4: Format 80 bitej podwyższonej precyzji zmienno przecinkowej

zakres dynamiki liczb zmienno przecinkowych o pojedynczej precyzji to w przybliżeniu $2^{\pm 128}$ do $10^{\pm 38}$.

Chociaż liczby zmienno przecinkowe o pojedynczej precyzji są odpowiednie dla wielu aplikacji, zakres dynamiki jest czasami za mały dal wielu aplikacji naukowych a duże ograniczenie dokładności jest nie odpowiednie dla wielu finansowych, naukowych i innych aplikacji. Co więcej w długim łańcuchu obliczeniowym ograniczenie dokładności formatu pojedynczej precyzji może wprowadzać poważne błędy.

Format podwójnej precyzji pomaga przezwyciężyć problemy pojedynczej precyzji zmienno przecinkowej. Używając dwóch obszarów, format podwójnej precyzji ma 11 bitów nadmiar-1023 wykładnika i 53 bity mantysy (z niejawnym bardziej znaczącym bite jako jedynką) plus znak bitu. Dostarcza to dynamiki zakresu od około $10^{\pm 308}$ i 14-1/2 precyzji cyfr, wystarczającą dal większości aplikacji. Wartości zmienno przecinkowe o podwójnej precyzji przybierają postać jak pokazano na rysunku 14.3

Żeby móc zapewnić dokładność podczas długiego łańcucha obliczeń liczb zmienno przecinkowych o podwójnej precyzji. Intel zaprojektował format o podwyższonej precyzji. Format podwyższonej dokładności używa 80 bitów. Dwanaście z szesnastu dodatkowych bitów jest połączonych z mantysą, cztery z dodatkowych bitów jest dołączonych na koniec wykładnika. W odróżnieniu od wartości od pojedynczej i podwójnej precyzji, format podwyższonej precyzji nie ma niejawnego bardziej znaczącego bitu, który jest zawsze jedynką. Dlatego format podwyższonej precyzji dostarcza 64 bitowej mantysy, 15 bitów wykładnika nadmiar – 16383 i jednego bitu znaku. Format dla wartości zmienno przecinkowej o podwyższonej precyzji jest pokazany na rysunku 14.4

W FPU 80x87 i CPU 80486 wszystkie obliczenia są robione przy użyciu postaci o podwyższonej precyzji. Kiedykolwiek ładujemy wartość o pojedynczej lub podwójnej precyzji, FPU automatycznie konwertuje je do wartości o rozszerzonej precyzji. Podobnie, kiedy przechowujemy wartość o pojedynczej lub podwójnej

precyzji w pamięci, FPU automatycznie zaokrągla wartość w dół do właściwego rozmiaru przed jej zachowaniem. Przez pracę z formatem o podwyższonej precyzji Intel gwarantuje dużą liczbę bitów zabezpieczenia obecnych dla zapewnienia dokładności naszego obliczenia. Niektóre teksty mylnie stwierdzają, ze nie powinniśmy nigdy używać formatu o rozszerzonej precyzji ponieważ Intel gwarantuje dokładność obliczeń tylko dla formatów o pojedynczej i podwójnej precyzji. To jest głupie. Przez wykonywanie wszystkich obliczeń używając 80 bitów , Intel zapewnia (ale nie gwarantuje) ,że uzyskamy pełną 32 lub 64 bitową precyzję w naszych obliczeniach. Ponieważ FPU 80x87 i CPU 80486 nie dostarcza dużej liczby bitów zabezpieczenia w 80 bitowych obliczeniach , pewne błędy nieuchronnie pojawią się w mniej znaczących bitach obliczenie o podwyższonej precyzji. Jednakże, jeśli nasze obliczenie jest poprawne dla 64 bitów , obliczenie 80 bitowe będzie zawsze dostarczało przynajmniej 64 poprawnych bitów. Większość czasu będzie to nawet więcej. Ponieważ nie możemy założyć, że uzyskamy prawidłowe 80 bitowe obliczenie, możemy zazwyczaj zrobić lepiej niż 64 kiedy stosujemy format o podwyższonej precyzji .

Zachowując maksimum dokładności podczas obliczenia, większość obliczeń używa wartości znormalizowanych. Znormalizowana wartość zmienno przecinkowa jest wartością, którą ma najbardziej znaczący bit mantysy równy jeden. Prawie każda nie znormalizowana wartość może być znormalizowana przez przesunięcie bitów mantysy w lewo i zmniejszanie wykładnika o jeden , dopóki nie pojawi się jeden w najbardziej znaczącym bicie mantysy. Pamiętajmy, że wykładnik jest wykładnikiem binarnym. Za każdym razem kiedy zwiększamy wykładnik mnożymy wartość zmienno przecinkową przez dwa. Podobnie kiedy zmniejszamy wykładnik, dzielimy wartość zmienno przecinkową przez dwa. Z tych samych powodów, przesuniecie mantysy w lewo o jeden bit mnoży wartość zmienno przecinkową przez dwa; podobnie przesunięcie mantysy w prawo dzieli wartość zmienno przecinkową przez dwa. Dlatego też przesunięcie mantysy w lewo i zmniejszenie wykładnika nie zmienia wcale wartości liczby zmienno przecinkowej.

Utrzymywanie znormalizowanych liczb zmienno przecinkowych jest korzystne ponieważ zachowuje maksymalną liczbę bitów precyzji dla obliczenia jeśli bardziej znaczące bity mantysy, wszystkie są zerami, mantysa ma o wiele mniej bitów precyzji dostępnych dla obliczenia. Dlatego też obliczenia zmienno przecinkowe będą o wiele bardziej dokładne jeśli wykorzystamy tylko wartości znormalizowane.

Są dwa ważne przypadki gdzie wartości znormalizowane nie mogą być znormalizowane. Wartość 0.0 jest specjalnym przypadkiem. Oczywiście nie może być znormalizowana ponieważ przedstawianie zera w postaci zmienno przecinkowej nie ma bitów jeden w mantysie. To jednak nie jest problemem ponieważ możemy dokładnie przedstawić wartość zero tylko pojedynczym bitem.

Drugi przypadek to wtedy kiedy mamy jakieś bardziej znaczące bity w mantysie są zerami ale przesunięty wykładnik również jest zerem (i nie możemy zmniejszyć go do znormalizowania mantysy). Zamiast odrzucać pewne małe wartości , kiedy bardziej znaczące bity mantysy i przesunięty wykładnik są zerami (większość możliwych ujemnych wykładników), standard IEE pozwala na specjalne nieznormalizowane wartości dla przedstawienia tych mniejszych wartości. Chociaż użycie nieznormalizowanych wartości pozwala obliczeniom zmienno przecinkowym IEEE na uzyskiwanie lepszych wyników niż jeśli wystąpił by niedomiar, zapamiętajmy, że wartości nieznormalizowane proponują mniejszą precyzję bitów i są z natury mniej dokładne.

Ponieważ FPU 80x87 i CPU 80486 zawsze konwertują wartości o pojedynczej i podwójnej precyzji na podwyższoną precyzję, arytmetyka podwyższonej precyzji jest rzeczywiście szybsza niż precyzji pojedynczej lub podwójnej.. Dlatego oczekiwane osiągnięcie korzyści z zastosowania mniejszych formatów nie jest obecne na tych chipach. Jednak kiedy projektowani Pentium /586, Intel przeprojektował wbudowaną jednostkę zmienno przecinkową dla lepszego współzawodnictwa z chipami RISC. Większość chipów RISC wspiera własny 64 bitowy format podwójnej precyzji, który jest szybszy niż Intelowski format podwyższonej precyzji. Dlatego też Intel wprowadził własne 64 bitowe operacje w Pentium dla lepszego konkurowania z chipami RISC. Dlatego też, format podwójnej precyzji jest najszybszy na Pentium i późniejszych chipach.

14.3 PODPROGRAMY ZMIENNO PRZECINKOWE STANDARDWOEJ BIBLIOTEKI UCR

W większości tekstów o języku asemblera , które zajmują się arytmetyką zmienno przecinkową, sekcja ta jest zazwyczaj opisywana jako sposób zaprojektowania własnych podprogramów dodawania, odejmowania, mnożenia i dzielenia. Ten tekst nie robi tego z kilku powodów .Po pierwsze, stworzenie dobrej biblioteki zmienno przecinkowej wymaga solidnych podstaw analizy numerycznej; warunki wstępne tego tekstu nie zakładają, że zna je czytelnik.. Po drugie Standardowa Bibliotek UCR dostarcza już sensownego zbioru podprogramów zmienno przecinkowych w formie kodu źródłowego: dlaczego marnować miejsce w tekście kiedy są dostępne łatwo źródła? Po trzecie jednostki zmienno przecinkowe szybko stają się standardowym wyposażeniem we wszystkich nowoczesnych CPU lub płytach głównych.; nie ma większego sensu opisywać jak ręcznie wykonać obliczenie zmienno przecinkowe niż opisywanie jak ręcznie wykonać obliczenia całkowite. Dlatego też ta sekcja będzie opisywała jak zastosować podprogramy Biblioteki Standardowej UCR jeśli nie mamy dostępnego FPU; późniejsze sekcje opiszą zastosowanie jednostki zmienno przecinkowej.

Standardowa Biblioteka UCR dostarcza dużej liczby podprogramów wspierających obliczenia zmienno przecinkowe i I/.O. Biblioteka ta używa takiego samego formatu pamięci dla liczb zmienno przecinkowych FPU 80x87 32, 64 i 80 bitowych. Podprogramy zmienno przecinkowe biblioteki Standardowej UCR niezbyt dokładnie spełniają wymagania IEEE pod względem warunków błędu i innych przypadków i mogą tworzyć odrobinę inne wyniki niż FPU 80x87 ale wyniki będą bardzo bliskie. Ponieważ Biblioteka Standardowa używa takiego samego formatu pamięci dla liczb 32, 64 i 80 bitowych jak FPU 80x87, możemy swobodnie połączyć obliczenia wymagające wartości zmienno przecinkowych, pomiędzy FPU a podprogramami Biblioteki Standardowej.

Biblioteka Standardowa UCR dostarcza licznych podprogramów do manipulowania liczbami zmienno przecinkowymi. Poniższe sekcje omówią każdy z tych podprogramów kategoriami.

14.3.1 ŁADOWNIE I PRZECHOWYWANIE PODPROGRAMÓW

Ponieważ COU 80c86 bez FPU nie dostarczają 80 bitowych rejestrów, Bibliotek Standardowa UCR musi użyć zmiennych bazujących na pamięci dla przechowania wartości zmienno przecinkowych podczas obliczenia. Podprogramy Standardowej Biblioteki UCR używają dwóch pseudo rejestrów, rejestru akumulatora i rejestru argumentów, kiedy wykonujemy operacje zmienno przecinkowe. Na przykład podprogram dodawania zmienno przecinkowego dodaje wartości w rejestrze argumentów zmienno przecinkowych do rejestru akumulatora zmienno przecinkowego, pozostawiając wynik w akumulatorze. Ładowanie i przechowywanie podprogramów pozwala nam ładować wartości zmienno przecinkowe do akumulatora zmienno przecinkowego i rejestrów argumentów jak również przechowywać wartości z akumulatora zmienno przecinkowego w pamięci. Do podprogramów z tej kategorii zaliczają się: accop, xacoop, lsfpa , ssfpa, ldfpa, sdfpa, lefpa, sefpa, lefpal, lsfpo, lefpo i lefpol

Podprogram accop kopiuje wartość z akumulatora zmienno przecinkowego do rejestru argumentów zmienno przecinkowych. Podprogram ten jest użyteczny kiedy chcemy użyć wyniku jednego obliczenia jako argumentu dla drugiego obliczenia.

Podprogram xacoop zamienia wartości w akumulatorze zmienno przecinkowym i rejestrze argumentów. Zauważmy, że wiele obliczeń zmienno przecinkowych niszczy wartości w rejestrze argumentów zmienno przecinkowych, więc nie możemy ślepo zakładać, że podprogramy zachowają rejestr argumentów. Dlatego też wywołanie tego podprogramu tylko ma sens po wykonaniu jakichś obliczeń, o których wiemy, że nie wpłyną na rejestr argumentów zmienno przecinkowych.

Lsfpa, ldfpa, i lefpa ładują do akumulatora, odpowiednio, wartość z pojedynczą, podwójną lub podwyższoną precyzją. Biblioteka Standardowa UCR używa swojego własnego wewnętrznego formatu dla obliczeń. Podprogramy te konwertują określone wartości do tego wewnętrznego formatu podczas ładowania. Na wejściu do każdego z tych podprogramów, es:di musi zawierać adres zmiennej, którą chcemy załadować do akumulatora zmienno przecinkowego. Poniższy kod demonstruje jak wywołać te podprogramy:

```
rVar        real4   1.0
drVar       real8   2.0
xrVar       real10  3.0
            -
            -
            -
            lesi    rVar
            lsfpa
            -
            -
            -
            lesi    drVar
            ldfpa
            -
            -
            -
            lesi    xrVar
            lefpa
```

Podprogramy lsfpo, ldfpo i lefpo są podobne do podprogramów lsfpa, ldfpa i lefpa z wyjątkiem oczywiście tego, że ładują one rejestr argumentów zmienno przecinkowych zamiast zmienno przecinkowego akumulatora wartościami spod adresu es:di

Lefpal i lefpol ładują akumulator zmienno przecinkowy lub rejestr argumentów dosłowną 80 bitową stałą zmienno przecinkową pojawiającą się w strumieniu kodu. Używając tych dwóch podprogramów następuje wywołanie z dyrektywą real10 i właściwą stałą, np.

```
        lefpal
        real10  1.0
        lefpol
        real10  2.0e5
```

podprogramy ssfpa, sdfpa i sefpa przechowują wartość w akumulatorze zmienno przecinkowym w zmiennej zmienno przecinkowej bazującej na pamięci której adres pojawia się w es:di. Nie odpowiadają one podprogramom ssfpo, sdfpo lub sefpo ponieważ wynik jaki chcemy przechować nie powinien pojawić się nigdy w rejestrze argumentów zmienno przecinkowych. Jeśli zdarzy się nam pobrać wartość argumentu zmienno przecinkowego , który chcemy przechować w pamięci po prostu użyjemy podprogramu xaccop do zamiany rejestrów akumulatora i argumentów, potem użyjemy podprogramów przechowania akumulatora do przechowania wyniku. Poniższy kod demonstruje użycie tych podprogramów:

```
rVar    real4   1.0
drvar   real8   2.0
xrVar   real10  3.0
        -
        -
        -
        lesi    rVar
        ssfpa
        -
        -
        -
        lesi    drVar
        sdfpa
        -
        -
        -
        lesi    xrVar
        sefpa
```

14.3.2 KONWERSJA WARTOŚCI CAŁKOWITE / WARTOŚCI ZMIENNO PRZECINKOWE

Biblioteka Standardowa UCR zawiera kilka podprogramów do konwersji pomiędzy binarnymi liczbami całkowitymi a wartościami zmienno przecinkowymi. Te podprogramy to itof, utof, ltof, ftoi, ftou, ftol i ftoul. Pierwsze cztery podprogramy konwertują liczby całkowite ze znakiem i bez znaku na format zmienno przecinkowy, ostatnie cztery obcinają wartości zmienno przecinkowe i konwertują je na wartości całkowite.

Itof konwertuje 16 bitową wartość ze znakiem w ax na wartość zmienno przecinkową i pozostawia wynik w akumulatorze zmienno przecinkowym. Ten podprogram nie wpływa na rejestr argumentów zmienno przecinkowych. Utof konwertuje wartości całkowite bez znaku w ax w podobny sposób. Ltof i ultof konwertują 32 bitowe ze znakiem (ltof) lub bez znaku (ultof) wartości całkowite w dx:ax do wartości zmienno przecinkowych pozostawiając wartość w akumulatorze zmienno przecinkowym . Te podprogramy zawsze kończą się sukcesem.

Ftoi konwertuje wartość z akumulatora zmienno przecinkowego do wartości całkowitej ze znakiem, pozostawiając wynik w ax. Konwersja odbywa się przez obcięcie; podprogram zatrzymuje część całkowitą i odrzuca część ułamkową. Jeśli wystąpi przepełnienie ponieważ część całkowita nie mieści się w 16 bitach, ftoi ustawia flagę przeniesienia. Jeśli konwersja wystąpi bez błędu , ftoi wyzeruje flagę przeniesienia. Ftou działa w podobny sposób, z wyjątkiem konwersji wartości zmienno przecinkowej do wartości całkowitej bez znaku w ax, i ustawia flagę przeniesienia jeśli wartość zmienno przecinkowa byłą ujemna.

Ftol i ftoul konwertują wartości z akumulatora zmienno przecinkowego do 32 bitowej wartości całkowitej pozostawiając wynik a dx:ax. Ftol działa na wartościach ze znakiem, ftoul na wartościach bez znaku. Podobnie jak ftoi i ftou podprogramy te zwracają ustawioną flagę przeniesienia jeśli wystąpi błąd konwersji.

14.3.3 ARYTMETYKA ZMIENNO PRZECINKOWA

Arytmetyka zmienno przecinkowa jest wykonywana przez podprogramy fpadd, fpsub, fpcmp, fpmul i fpdiv. Fpadd dodaje wartości w akumulatora zmienno przecinkowego do akumulatora zmienno przecinkowego. Fpsub odejmuje wartość argumentu zmienno przecinkowego od akumulatora zmienno przecinkowego. Fpmul mnoży wartość z akumulatora przez rejestr argumentów zmienno przecinkowych. Fpdiv dzieli wartość z akumulatora zmienno przecinkowego przez wartość w rejestrze argumentów zmienno przecinkowych. Fpcmp porównuje wartość w akumulatorze zmienno przecinkowym z rejestrem argumentów zmienno przecinkowych.

Podprogramy arytmetyczne Biblioteki Standardowej UCR wykonują trochę kontroli błędów. Na przykład, jeśli przepełnienie arytmetyczne wystąpi podczas dodawania, odejmowania ,mnożenia lub dzielenia, Biblioteka Sztandarowa po prostu ustawia wynik na największą poprawną wartość i ją zwraca.. Jest to jedno z głównych odchyleń od standardu zmienno przecinkowego IEEE. Podobnie, kiedy wystąpi niedomiar, podprogramy po prostu ustawiają wynik na zero i zwracają. Jeśli podzielimy jakąś wartość przez zero, podprogramy biblioteki Standardowej ustawią wynik na największą możliwą wartość i zwracają. Możemy musieć zmodyfikować podprogramy biblioteki standardowej jeśli musimy skontrolować przepełnienie, niedomiar lub dzielenie przez zero w naszych programach.

Podprogram porównania zmienno przecinkowego (fpcmp) porównuje akumulator zmienno przecinkowy z rejestrem argumentów zmienno przecinkowych i zwraca –1, 0 lub 1 w rejestrze ax jeśli akumulator jest mniejszy niż, równy lub większy niż rejestr argumentów. Porównuje również ax z zerem bezpośrednio przed zwróceniem, więc ustawia flagi aby można było użyć instrukcji jg, jge, jl, jle, je i jne bezpośrednio po wywołaniu fpcmp. W odróżnieniu od fpadd, fpsub, fpmul i fpdiv, fpcmp nie niszczy wartości w akumulatorze zmienno przecinkowym lub rejestrze argumentów zmienno przecinkowym. Pamiętajmy o problemach związanych z porównywaniem liczb zmienno przecinkowych!

14.3.4 KONWERSJA WARTOŚCI ZMIENNO PRZECINLKOWYCH NA TEKST I PRINTFF

Standardowa Biblioteka UCR dostarcza trzech podprogramów, ftoa, etoa i atof, które pozwalają nam konwertować liczby zmienno przecinkowe na ciągi ASCII i vice versa; dostarcza również specjalnej wersji printf, printff, która zawiera zdolność do drukowania wartości zmienno przecinkowych tak jak innych typów danych.

Ftoa konwertuje liczbę zmienno przecinkową na ciąg ASCII, który jest dziesiętnym odpowiednikiem tej liczby zmienno przecinkowej. Na wejściu, akumulator zmienno przecinkowy zawiera liczbę, którą chcemy skonwertować do ciągu. Para rejestrów es:di wskazuje bufor w pamięci gdzie ftoa będzie przechowywać ciąg. Rejestr al. zawiera pole szerokości (liczbę pozycji do druku) Rejestr ah zawiera liczbę pozycji do wyświetlenia na prawo od punktu dziesiętnego. Jeśli ftoa nie może wyświetlić liczby używając formatu druku określonego przez al. i ah, stworzy ciąg znaków „#”, długi na ah znaków. Es:di musi wskazywać tablicę bajtów zawierającą przynajmniej al.+1 znaków i al. powinien zawierać przynajmniej pięć. Pole szerokości i dziesiętna wartość długości w rejestrach al. i ah są podobne do wartości pojawiających się po liczbach zmienno przecinkowych w Pascalowskiej instrukcji write, np.:

```
write (floatVar :al.: ah);
```

Etoa wyprowadza liczby zmienno przecinkowe w postaci wykładniczej. Podobnie jak przy ftoa, es:di wskazuje bufor gdzie etoa będzie przechowywać wynik. Rejestr al. musi zawierać przynajmniej osiem i jest polem szerokości dla liczby. Jeśli al. zawiera mniej niż osiem , etoa wyprowadzi ciąg znaków „#”. Ciąg , który wskazuje es:di musi zawierać przynajmniej al.+1 znaków. Ten podprogram konwersji jest podobny do pascalowskiej procedury write , kiedy zapisujemy wartości rzeczywiste z wyspecyfikowanym pojedynczym polem szerokości :

```
write (realVar:al.)
```

Podprogram Biblioteki Standardowej printff dostarcza wszystkich możliwości standardowego podprogramu printf plus umiejętność wyprowadzania wartości zmienno przecinkowych. Podprogram printff wprowadza kilka nowych specyfikacji formatu dla druku liczb zmienno przecinkowych w postaci dziesiętnej lub przy użyciu notacji naukowej. Te specyfikacje to:

* %x.y F Wydruk 32 bitowej liczby zmienno przecinkowej w postaci dziesiętnej
* %x.yGF Wydruk 64 bitowej liczby zmienno przecinkowej w postaci dziesiętnej
* %x.yLF Wydruk 80 bitowej liczby zmienno przecinkowej w postaci dziesiętnej
* %zE Wydruk 32 bitowej liczby zmienno przecinkowej używając notacji naukowej
* %zGE Wydruk 64 bitowej liczby zmienno przecinkowej używając notacji naukowej
* %zLE Wydruk 80 bitowej liczby zmiennoprzecinkowej używając notacji naukowej

W powyższych formatach ciągów, x i z są stałymi całkowitymi, które oznaczają pole szerokości liczby do wydruku. Pozycja y jest również stałą całkowitą , która określa liczbę pozycji do druku po przecinku dziesiętnym. Wartości x y są porównywalne z wartościami przekazywanymi do ftoa w al. i ah. Wartość z jest porównywalna z wartością etoa oczekiwaną w rejestrze al.

Z wyjątkiem tych sześciu nowych formatów, podprogram printff jest identyczny z podprogramem printf. Jeśli użyjemy podprogramu printff w programie asemblerowym, nie powinniśmy już używać podprogramu printf. Printff duplikuje wszystkie udogodnienia printf i używanie obu byłoby marnotrawieniem pamięci

14.4 KOPROCESOR ZMIENNO PRZECINKOWY 80x87

Kiedy latach siedemdziesiątych pojawiły się pierwsze CPU 8086, technologia półprzewodnikowa nie była w takim punkcie, gdzie Intel mógł włożyć instrukcje zmienno przecinkowe bezpośrednio do CPU 8086. Dlatego też, podzielili oni projekt, dzięki czemu mogli użyć drugiego chipu dla wykonania obliczeń zmienno przecinkowych – jednostka zmienno przecinkowa (lub FPU). Swój oryginalny chip zmienno przecinkowy 80x87 udostępnili w 1980 roku. Ten szczególny FPU działał z CPU 8086, 8088,80186 i 80188. Kiedy Intel wprowadził CPU 80286, udostępnili przeprojektowany chip FPU 80287. Chociaż 80287 był kompatybilny z CPU 80386 , Intel zaprojektował lepszy, 80387, dla stosowania z 80386. CPU 80486 był pierwszym CPU Intela zawierający FPU zintegrowany z układem. Krótko po udostępnieniu 80486, Intel wprowadził CPU 80486sx, który był 80486 ale bez wbudowanego FPU Dla uzyskania możliwości zmienno przecinkowych na tym chipie, musiano dodać chip 80487, chociaż w rzeczywistości 80487 był niczym więcej niż pełnym 80486, który przejął chip „sx" w systemie. Chipy Intela Pentium/ 586 dostarczyły jednostkę zmienno przecinkową o dużej wydajności bezpośrednio w CPU. Nie ma koprocesora zmienno przecinkowego dostępnego dla chipu Pentium.

Zbiorczo będziemy się odnosić do wszystkich tych chipów jak FPU 80x87. Ze względu na zestarzenie się chipów 8086, 80286,8087 i 80287, ten tekst skoncentruje się na chipie 80386 i późniejszych. Jest kilka różnic pomiędzy jednostkami zmienno przecinkowymi 80387 / 80487/ Pentium a wcześniejszymi FPU. Jeśli musisz napisać kod, który będzie wykonywany na tych wcześniejszych maszynach, powinieneś sprawdzić właściwą Intelowską dokumentację dla tego urządzenia.

14.4.1 REJESTRY FPU

FPU 80x87 dodał 13 rejestrów do 80386 i późniejszych procesorów : osiem rejestrów danych zmienno przecinkowych, rejestr sterujący, rejestr stanu, rejestr znaczników, wskaźnik instrukcji i wskaźnik danych. Rejestry danych są podobne do zbioru rejestrów ogólnego przeznaczenia 80x86 na tyle na ile obliczenia zmienno przecinkowe mają miejsce w tych rejestrach. Rejestr sterujący zawiera bity , które pozwalają nam decydować jak radzić będzie sobie 80x87 w pewnych przypadkach takich jak zaokrąglanie niedokładnych obliczeń, kontroli precyzji itd. Rejestr stanu jest podobny do rejestru flag 80x86 : zawiera bity kodu warunkowego kilka innych flag zmienno przecinkowych , które opisują stan chipu 80x87 . Rejestr znaczników zawiera kilka grup bitów , które określają stan wartości w każdym z ośmiu rejestrów ogólnego przeznaczenia. Rejestry wskaźników instrukcji i danych zawierają pewne informacje o stanie

Rysunek 14.5 Rejestr stosu zmienno przecinkowego 80x87

O ostatnio wykonanej instrukcji zmienno przecinkowej. Nie będziemy rozpatrywać ostatnich trzech rejestrów w tym tekście.

14.4.1.1. REJESTRY DANYCH FPU

FPU 80x87 dostarcza ośmiu 80 bitowych rejestrów danych zorganizowanych w formie stosu. Jest to znaczące odejście od organizacji rejestrów ogólnego przeznaczenia w CPU 80x86, które składają się na standard

zbioru rejestrów ogólnego przeznaczenia. Intel odnosi się do tych rejestrów jako ST(0), ST(1), ..., ST(7). Większość asemblerów zaakceptuje ST jako skrót dla ST(0)

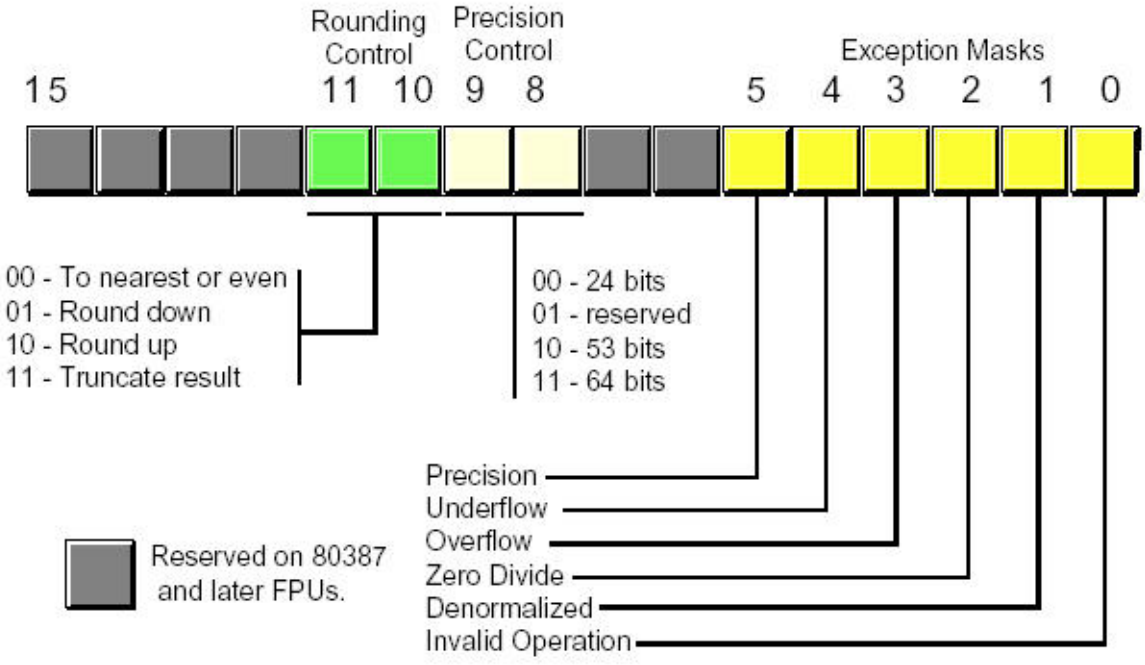
Największą różnicą pomiędzy zbiorem rejestrów FPU a zbiorem rejestrów 80x86 jest organizacja stosu. W CPU 80x86, rejestr ax jest zawsze rejestrem ax , bez względu na to co się dzieje. W 80x87 jednak, zbiór rejestrów jest ośmio elementowym stosem 80 bitowych wartości zmienno przecinkowych (zobacz rysunek 14.5). ST(0) odnosi się do pozycji na szczycie stosu, ST(1) odnosi się do następnej pozycji na stosie i tak dalej. Wiele instrukcji zmienno przecinkowych odkłada i zdejmuje coś na stos; dlatego też, ST(1) będzie się odnosić do poprzedniej zawartości ST(0) po odłożeniu przez nas czegoś na stos. Trzeba będzie przywyknąć myślowo i praktycznie do faktu, że rejestry są zmieniane przez nas, ale ten problem jest łatwy do przezwyciężenia.

14.4.1.2 REJESTR STERUJĄCY FPU

Kiedy Intel zaprojektował 80x87 (i, zasadniczo, standard zmienno przecinkowy IEEE), nie było standardu zmienno przecinkowego sprzętu. Różne (duże i mini) fabryki komputerów , wszystkie miały różne i niekompatybilne formaty zmienno przecinkowe. Niestety wiele aplikacji zostało napisanych biorąc pod uwagę cechy tych różnych formatów zmienno przecinkowych. Intel chciał zaprojektować FPU, który mógłby pracować z większością tego oprogramowania (pamiętajmy, że IBM był trzy, cztery lata w tyle, kiedy Intel zaczynał projektowanie 8087, nie mogli polegać na tej „górze" oprogramowania dostępnego na PC czyniąc ich chip popularnym). Niestety wiele cech znajdowanych w tych starszych formatach zmienno przecinkowych było wzajemnie wykluczających. Na przykład w jakimś systemie zmienno przecinkowym występowało zaokrąglanie kiedy była niewystarczająca precyzja, w innych występowało obcięcie. Niektóre aplikacje będą pracowały z jednym systemem zmienno przecinkowym a z innym nie. Intel chciał, żeby tak wiele aplikacji jak to możliwe pracowało z niewielkimi zmianami na FPU 80x87, więc dodał specjalny rejestr, rejestr sterujący FPU, który pozwala użytkownikowi wybrać jeden z kilku możliwych trybów działania 80x87

Rejestr sterujący 80x87 jest zorganizowany w 16 bitów jak pokazano na rysunku 14.6

Bit 12 tego rejestru jest obecny tylko w chipach 8087 i 80287. Steruje on tym jak 80x87 reaguje na nieskończoność. 80387 i późniejsze chipy zawsze używa postaci znanej jako zamknięcie afiniczne ponieważ jest to jedyna postać wspierana przez standardy IEEE 754/ 854.



Rysunek 14.6: Rejestr sterujący 80x87

Dalej będziemy ignorować ten bit i założymy ,że zawsze jest zaprogramowany na jeden.

Bity 10 i 11 dostarczają kontroli zaokrąglania według poniższej tabeli:

Bity 10 i 11	Funkcja
00	Najbliżej lub parzyście
01	Zaokrąglanie w dół
10	Zaokrąglanie w górę

11	Obcięcie

Tablica 58: Sterowanie zaokrąglaniem

Ustawienie „00” jest domyślne. 80x87 zaokrągla wartości powyżej połówkowego najmniej znaczącego bitu w górę. Zaokrągla w dół poniżej połówkowego najmniej znaczącego bitu. Jeśli wartość poniżej najmniej znaczącego bitu jest dokładnie połówkowym najmniej znaczącym bitem, 80x87 zaokrągla wartość w kierunku wartości, której najmniej znaczący bit jest zerem. Dla długiego łańcucha obliczeń, dostarcza sensownego, automatycznego sposobu uzyskania maksimum precyzji.

Opcje zaokrąglania w dół i w górę są obecne dla tych obliczeń, gdzie ważne jest śledzenie dokładności podczas obliczeń. Przez ustawienie kontroli sterowania na zaokrąglanie w dół i wykonanie działania, powtarzanie działania z kontrolą zaokrąglania ustawioną w górę, możemy określić minimalny i maksymalny zakres pomiędzy którym padnie prawdziwy wynik.

Opcja obcinania wymuszana wszystkich obliczeniach, obcinania nadmiarowych bitów podczas obliczania. Będziemy rzadko używali tej opcji, jeśli ważna dla nas jest dokładność. Jednakże, jeśli przenosimy starsze oprogramowanie pod 80x87, możemy użyć tej opcji pomagającej przy przenoszeniu oprogramowania.

Bity osiem i dziewięć rejestru sterującego sterują precyzją podczas obliczenia. Ta zdolność jest dostarczona głównie ze względu na kompatybilność ze starszym oprogramowaniem wedle standardu IEEE 754. Bity sterowania precyzją używają następujących wartości:

Bity 8 i 9	Sterowanie precyzją
00	24 bity
01	Zarezerwowane
10	53 bit
11	64 bity

Tabela 59: Mantysa bitów sterujących precyzją

Dla nowoczesnych aplikacji, bity sterowania precyzją powinny zawsze być ustawione na „11” dl uzyskania 64 bitowej precyzji. Stworzy to najbardziej dokładny wynik podczas obliczeń numerycznych.

Bity od zero do pięć są maskami wyjątków. Są one podobne do bitu zezwalającego na przerwania w rejestrze flag 80x86. Jeśli te bity zawierają jedynki, odpowiedni warunek jest ignorowany przez FPU 80x87. Jednakże jeśli jakiś bit zawiera zero i wystąpi odpowiedni warunek, wtedy FPU natychmiast generuje przerwanie, aby program mógł obsłużyć ten warunek.

Bit zero odpowiada błędowi niewłaściwej operacji. Generalnie występuje jako wynik błędu programistycznego. Problem, który zgłasza wyjątek niepoprawnej operacji zawiera odłożenie więcej niż ośmiu pozycji na stos lub próba zdjęcia czegoś z pustego stosu, wyliczenia pierwiastka kwadratowego z liczby ujemnej, lub załadownia nie pustego rejestru.

Bit jeden maskuje nieznormalizowane przerwanie, które wystąpi jeśli tylko spróbujemy manipulować wartościami nieznormalizowanymi. Wartości nieznormalizowane generalnie pojawiają się wtedy kiedy lądujemy dowolne wartości o podwyższonej precyzji do FPU lub pracujemy z bardzo małymi liczbami z poza zakresu zdolności FPU. Zwykle , prawdopodobnie nie będziemy aktywować tego wyjątku.

Bit dwa maskuje wyjątek dzielenia przez zero. Jeśli ten bit zawiera zero, FPU wygeneruje przerwani jeśli spróbujemy podzielić wartość niezerową przez zero. Jeśli nie włączymy wyjątku dzielenia przez zero , FPU będzie tworzył NaN (not a number – nie liczbę), kiedy będziemy wykonywali dzielnie przez zero.

Bit trzy maskuje wyjątek przepełnienia. FPU zgłasza wyjątek przepełnienia jeśli nastąpi przepełnienie przy obliczeniu lub jeśli spróbujemy przechować wartość, która jest zbyt duża do przechowania w argumencie przeznaczenia (tj. przechowanie dużej wartości o podwyższonej precyzji w zmiennej o pojedynczej precyzji).

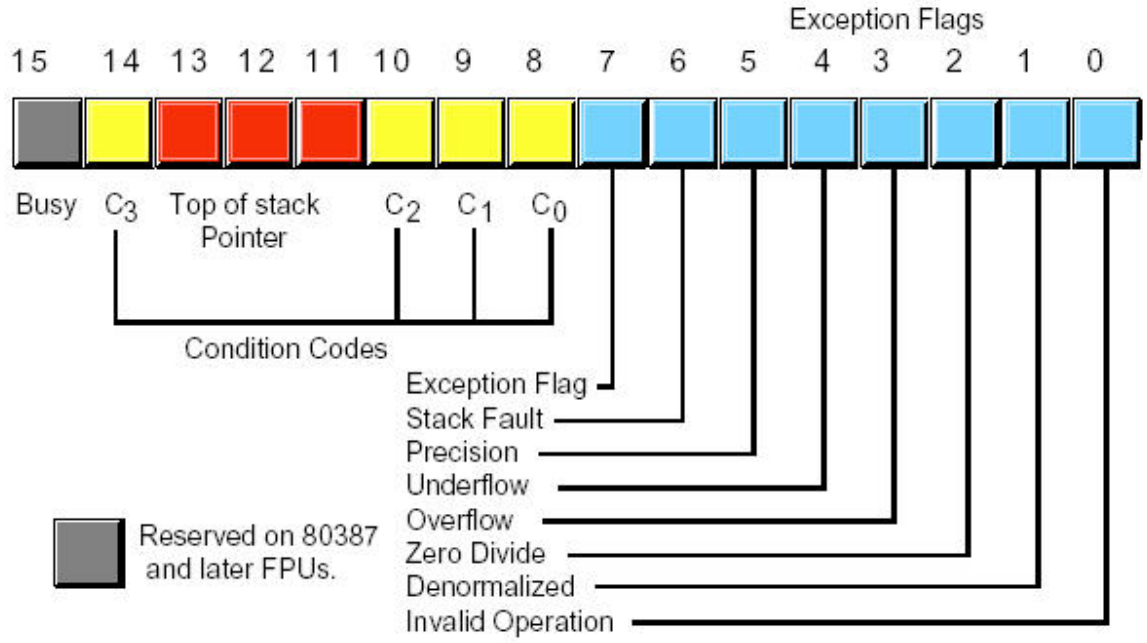
Bit cztery, jeśli jest ustawiony, maskuje wyjątek niedomiaru. Niedomiar wystąpi kiedy wynik jest zbyt mały do przechowania w argumencie przeznaczenia. Podobnie jak przepełnienie, wyjątek ten może wystąpić, kiedy przechowujemy małą wartość o rozszerzonej precyzji w mniejszej zmiennej (pojedynczej lub podwójnej precyzji)lub kiedy wynik obliczenia jest zbyt mały dla rozszerzonej precyzji.

Bit pięć kontroluje czy może wystąpić wyjątek precyzji. Wyjątek ten wystąpi kiedy FPU tworzy nieprecyzyjny wynik, generalni wynik wewnętrznej operacji zaokrąglania. Chociaż wiele operacji będzie tworzyło dokładny wynik , o wiele więcej nie. Na przykład , dzieląc jeden przez dziesięć uzyskujemy wynik nie dokładny. Dlatego też ten bit jest zazwyczaj jedynką , ponieważ wyniki niedokładne są bardzo powszechne.

Bity sześć i od trzynaście do piętnaście w rejestrze sterującym są aktualnie niezdefiniowane i zarezerwowane dla późniejszego zastosowania. Bit siedem jest maską zezwolenia n przerwanie, ale jest tylko czynna na FPU 80x87; zero w tym bicie zezwala na przerwania 80x87 a jeden nie.

80x87 dostarcza dwóch instrukcji, FLDCW (ładuj słowo sterujące) i FSTCW (przechowaj słowo sterujące), które pozwalają nam załadować i przechować zawartość rejestru sterującego. Pojedynczy argument

tych instrukcji musi być 16 bitową komórką pamięci. Instrukcja FLDCW ładuje rejestr sterujący z określonej komórki pamięci, FSTCW przechowuje zawartość rejestru sterującego w określonej komórce pamięci.



Rysunek 14.7: Rejestr stanu FPU

14.4.1.3 REJESTR STANU FPU

Rejestr stanu FPU dostarcza stanu koprocesora w chwili jego odczytu. Instrukcja FSTSW przechowuje 16 bitowy zmienno przecinkowy rejestr stanu w argumencie mod/ reg / rm. Rejestr stanu jest 16 bitowym rejestrem, a jego układ jest pokazany na rysunku 14.7.

Bity od zero do pięć są znacznikami wyjątków. Bity te pojawiają się w takim samym porządku jak maski wyjątków w rejestrze sterującym. Jeśli odpowiedni warunek istnieje, wtedy bit jest ustawiony. Bity te są niezależne od masek wyjątków w rejestrze sterującym. 80x87 ustawia i zeruje te bity bez względu na odpowiadające ustawienia masek.

Bit sześć (czynny tylko w procesorach 80386 i późniejszych) wskazuje zakłócenie stosu. Zakłócenie stosu wystąpi kiedykolwiek stos jest przepełniony lub niedopełniony. Kiedy ten bit jest ustawiony, bit kodu błędu C_1 określa czy było przepełnienie stosu ($C_1 = 1$) lub niedomiar ($C_1=0$).

Bit siedem rejestru stanu jest ustawiony jeśli jest ustawiony jakikolwiek bit warunku wystąpienia błędu. Jest to logiczne OR bitów od zero do pięć. Program może przetestować ten bit szybko określając czy istnieje warunek wystąpienia błędu.

Bity osiem, dziewięć, dziesięć i czternaście są bitami kodów warunkowych koprocesora. Różne instrukcje ustawiają bity kodów zakończenia tak jak pokazano w poniższej tablicy:

Instrukcje	Bity kodu zakończenia C3	C2	C1	C0	Warunek
fcom, fcomp,	0	0	X	0	ST > źródła
fcompp,	0	0	X	1	ST < źródła
ficom,	1	0	X	0	ST = źródło
ficomp	1	1	X	1	ST lub niezdefiniowane źródło
	X = nie ważne				
ftst	0	0	X	0	ST jest dodatnie
	0	0	X	1	ST jest ujemne
	1	0	X	0	ST to zero (+ lub -)
	1	1	X	1	ST jest nieporównywalne
fxam	0	0	0	0	+ Nieznormalizowane
	0	0	1	0	- Nieznormalizowane

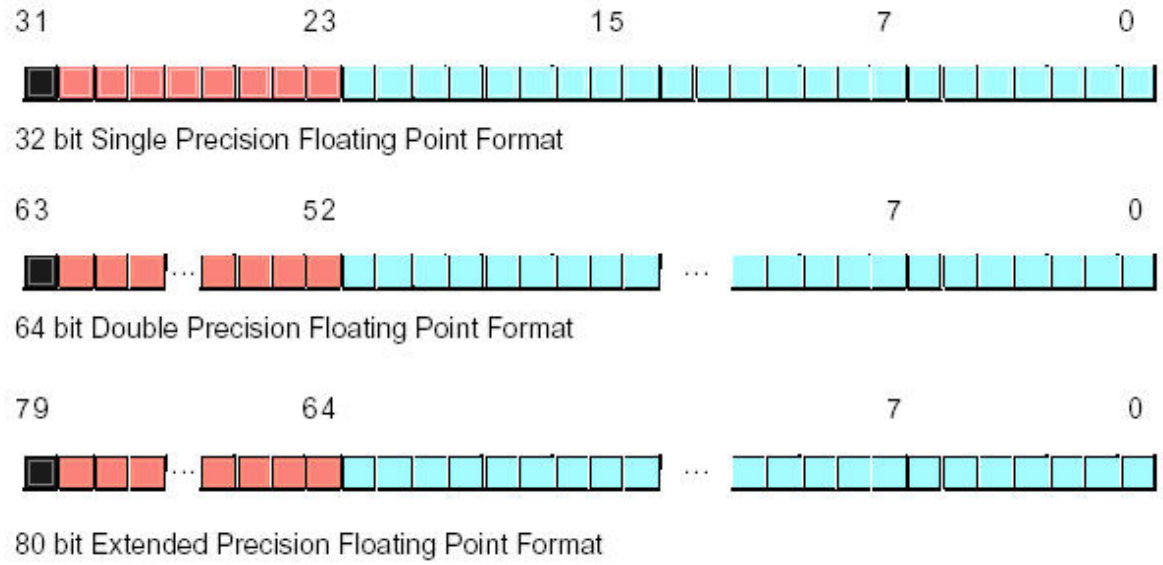
	0	1	0	0	+ Znormalizowane
	0	1	1	1	- Znormalizowane
	1	0	0	0	+0
	1	0	1	0	-0
	1	1	0	0	+Deznormalizowane
	1	1	1	0	-Deznormalizowane
	0	0	0	1	+NaN
	0	0	1	1	-Nan
	0	1	0	1	+Nieskończoność
	0	1	1	1	-Nieskończoność
	1	X	X	1	Rejestr pusty
fucom,	0	0	X	0	ST > źródła
fucomp;	0	0	X	1	ST < źródła
fucompp	1	0	X	0	ST = źródło
	1	1	X	1	Nieuporządkowane
	X = nie ważne				

Tablica 60: Bity kodów warunkowych FPU

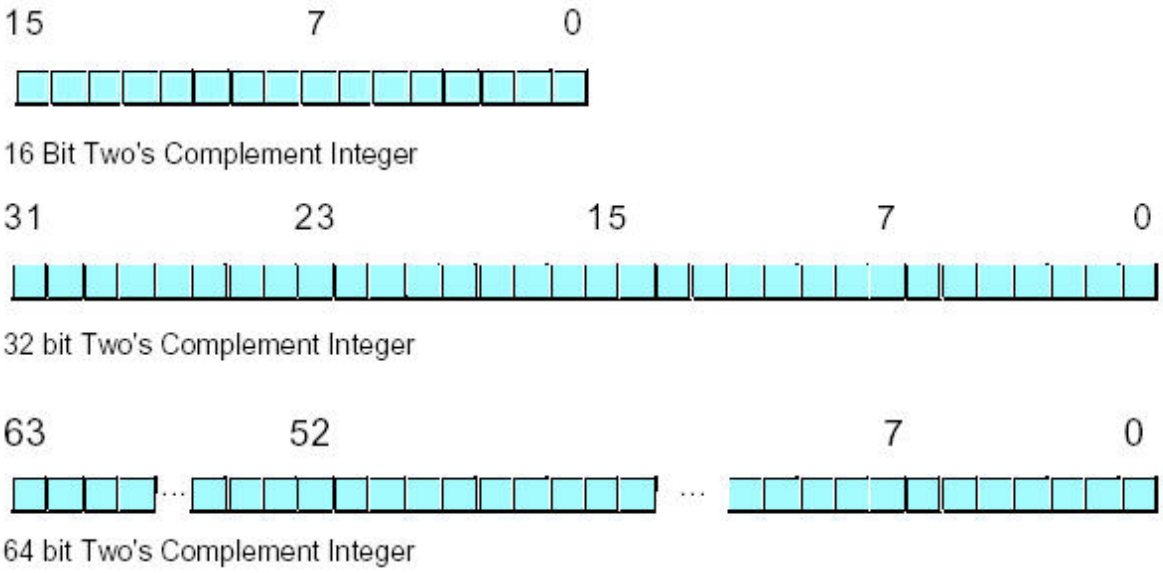
Instrukcja(-e)	C_0	C_3	C_2	C_1
fcom, fcomp, fcmpp, ftst, fucom, fucomp, fucompp, ficom, ficomp	Wynik porównania. Zobacz powyższą tabelę	Wynik porównania. Zobacz powyższą tabelę	Argument nie jest porównywalny.	Wynik porównania (zobacz powyższą tabelę) lub przepełnienie / niedomiar stosu (jeśli bit wyjątku stosu jest ustawiony
fxam	Zobacz poprzednią tabelę	Zobacz poprzednią tabelę	Zobacz poprzednią tabelę	Znak wyniku , lub przepełnienie /niedomiar stosu (jeśli bit wyjątku stosu jest ustawiony)
fprem, fprem1	Bit 2 reszty	Bit 0 reszty	0 – skrócenie zrobione 1- skrócenie niekompletne	Bit 1 reszty lub przepełnienie / niedomiar stosu (jeśli bit wyjątku stosu jest ustawiony)
fist, fbstp, frndint, fst, fstp, fadd, fmul, fdiv, fdivr, fsub, fsubr, fscale, fsqrt, fpatan, f2xm1, fyl2x,fyl2xp1	Niezdefiniowany	Niezdefiniowany	Niezdefiniowany	Wystąpienie zaokrąglenia górę lub przepełnienie / niedomiar (jeśli bit wyjątku stosu jest ustawiony)
fptan, fsin, fcos, fsincos	Niezdefiniowany	Niezdefiniowany	0 – skrócenie zrobione 1- skrócenie niekompletne	Wystąpienie zaokrąglenia w górę lub przepełnienie / niedomiar stosu (jeśli bit wyjątku stosu jest ustawiony)
fchs, fabs, fxch, fincstp,fdecstp, Stałe ładowane ,fxtract, fld, fild, fbld, fstp (80 bit)	Niezdefiniowany	Niezdefiniowany	Niezdefiniowany	Wynik zero lub przepełnienie / niedomiar stosu (jeśli bit wyjątku stosu jest ustawiony)
fldenv, fstor	Przywrócony z operandu pamięci	Przywrócony z operandu pamięci	Przywrócony z operandu pamięci	Przywrócony z operandu pamięci
fldcw, fstenv ,fstcw,	Niezdefiniowany	Niezdefiniowany	Niezdefiniowany	Niezdefiniowany

fstsw, fclex				
finit, fsave	Wyczyszczony do zera	Wyczyszczony do zera	Wyczyszczony do zera	Wyczyszczony do zera

Tablica 61: Interpretacja kodów warunkowych



Rysunek 14.8: Format zmienno przecinkowy 80x87



Rysunek 14.9: Format całkowity 80x87

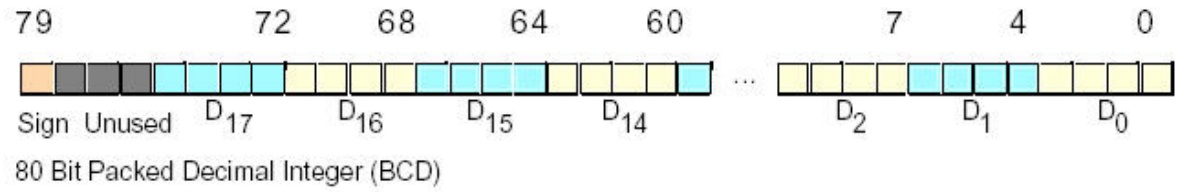
Bity 11-13 rejestru stanu FPU dostarczają liczy rejestrów na szczycie stosu. Podczas obliczenia, 80x87 dodaje (modulo osiem) logiczną liczę rejestrów dostarczonych przez programistę do tych trzech bitów, określając fizyczną liczbę rejestrów w czasie wykonania.

Bit 15 rejestru stanu jest bitem zajętości. Jest ustawiony wtedy kiedy FPU jest zajęty Wiele programów będzie miało powód do uzyskania dostępu do tego bitu.

14.4.2 TYPY DANYCH FPU

FP 80x87 wspiera siedem różnych typów danych: trzy typy całkowite, upakowany typ dziesiętny i trzy typy zmienno przecinkowe. Ponieważ CPU 80x87 już wspiera całkowite typy danych, jest niewiele powodów dla których używać będziemy typów całkowitych 80x87. Typ upakowanej liczby dziesiętnej dostarcza 17 cyfrową liczę całkowitą ze znakiem (BCD) Jednakże pominiemy arytmetykę BCD w tym tekście, więc

zignorujemy ten typ danych w FPU 80x87.Pozostałe trzy typy danych są 32 bitowymi, 64 bitowymi i 80 bitowymi typami zmienno przecinkowymi. Typy danych 80x87 pojawiają się na rysunku 14.8, rysunku 14.9 i rysunku 14.10



Rysunek 14.10: Format upakowanej liczby dziesiętnej 80x87

FPU 80x87 generalnie przechowuje wartości w formacie znormalizowanym. Kiedy liczba zmienno przecinkowa jest znormalizowana, najbardziej znaczący bit jest zawsze jeden. W formacie zmienno przecinkowym 32 i 64 bitowym, 80x87 właściwie nie przechowuje tych bitów,80x87 zawsze zakłada, że jest to jeden. Dlatego też, liczby zmienno przecinkowe 32 i 64 bitowe są zawsze znormalizowane. W 80 bitowym formacie o podwyższonej precyzji, 80x87 nie zakłada, że najbardziej znaczący bit mantysy to jeden, najbardziej znaczący bit pojawia się jako część ciągu bitów.

Wartości znormalizowane dostarczają największej precyzji dla danej liczby bitów. Jednakże jest duża liczba wartości nie znormalizowanych, które mogą być przedstawiane w formacie 80 bitowym. Wartości te są bardzo bliskie zeru i przedstawiają zbiór wartości, których mantysa bardziej znaczącego bitu nie jest zerem. FPU 80x87 wspiera specjalną formę 80 bitową znaną jako wartości denormalizowane. Wartości denormalizowane pozwalają 80x87 zakodować bardzo małe wartości, których nie może zakodować używając wartości znormalizowanych, ale za wysoką cenę. Wartości denormalizowane oferują mniejszą precyzję bitową niż wartości znormalizowane. Dlatego też używając wartości denormalizowanych w obliczeniach możemy wprowadzić drobną niedokładność do obliczenia. Oczywiście, jest to zawsze lepsze niż zbliżenie wartości denormalizowanej do zera (co może uczynić obliczenie jeszcze mniej dokładnym), ale musimy zapamiętać, że jeśli pracujemy z bardzo małymi wartościami, możemy zgubić pewną dokładność w naszych obliczeniach. Zauważmy, że rejestr stanu 80x87 zawiera bit, który możemy użyć do wykrycia kiedy FPU używa wartości denormalizowanych w obliczeniu.

14.4.3 ZBIÓR INSTRUKCJI FPU

FPU 80387 (i późniejsze) dodał ponad 80 nowych instrukcji do zbioru instrukcji 80x86. Możemy zaklasyfikować te instrukcje jako instrukcje przesuwania danych, konwersji, instrukcje arytmetyczne, porównania, instrukcje stałe, instrukcje przestępne i instrukcje różne. Poniższe sekcje omawiają każą z tych instrukcji w tych kategoriach.

14.4.4 INSRUKCJE PRZSUNIĘCIA DANYCH FPU

Instrukcje przesunięcia danych przenoszą dane pomiędzy wewnętrznymi rejestrami FPU a pamięcią. Instrukcje w tej kategorii to fld, fst, fstp, i fxch. Instrukcja fld zawsze odkłada swoje operandy na stos zmienno przecinkowy. Instrukcja fstp zawsze zdejmuje szczyt stosu po zachowaniu szczytu stosu (tos) w jej operacji. Pozostałe instrukcje nie wpływają na liczbę pozycji na stosie.

14.4.4.1 INSTRUKCJA FLD

Instrukcja fld ładuje 32, 64 lub 80 bitową wartość zmienno przecinkową na stos. Instrukcja ta konwertuje 32 i 64 bitowy operand na80 bitową wartość o podwyższonej precyzji przed odłożeniem wartości na stos zmienno przecinkowy

Instrukcja fld zmniejsza wskaźnik tos bity 11 – 13 rejestru stanu) a potem zachowuje 80 bitową wartość w rejestrze fizycznym określonym przez nowy wskaźnik tos. Jeśli argument źródłowy instrukcji fld jest rejestrem danych zmienno przecinkowych ST(i), wtedy rzeczywisty rejestr 80x87 używany do operacji ładowania jest numerem rejestru przed zmniejszeniem wskaźnika tos. Dlatego też, fld st lub fld st(0) duplikują wartość na szczyt stosu.

Instrukcja fld ustawia bit zakłócenia stosu jeśli wystąpi przepełnienie stosu. Ustawia bit wyjątku denormalizacji jeśli załadujemy 80 bitową wartość denormalizowaną. Ustawia bit nieprawidłowej operacji jeśli próbujemy załadować pusty rejestr zmienno przecinkowy na szczyt stosu (lub wykonujemy jakąś inną nieprawidłową operację)

Przykład:

```
fld     st(1)
fld     mem_32
fld     MyRealVar
fl      mem_64[bx]
```

14.4.4.2 INSTRUKCJE FST I FSTP

Instrukcje fst i fstp kopiują wartość ze szczytu rejestru stosu zmienno przecinkowego do innego rejestru lub 32, 64 lub 80 bitowej zmiennej pamięciowej. Kiedy kopiujemy dane do 32 lub 64 bitowej zmiennej pamięciowej, wartość 80bitowa o podwyższonej precyzji na szczycie stosu jest zaokrąglana do mniejszego formatu, określonego przez bit kontroli zaokrąglania w rejestrze sterującym FPU.

Instrukcja fstp zdejmuje wartość ze szczytu stosu kiedy przenosimy ją do lokacji przeznaczenia. Robi to przez zwiększenie wskaźnika szczytu stosu w rejestrze stanu po uzyskaniu dostępu do danych w st(0). Jeśli operand przeznaczenia jest rejestrem zmienno przecinkowym, FPU przechowa wartość pod określonym numerem rejestru przed zdjęciem danych ze szczytu stosu.

Wykonując instrukcję fstp st(0) faktycznie zdejmuje dane ze szczytu stosu bez transferu danych. Przykład:

```
fst     mm_32
fstp    mem_64
fstp    mem_64 [ebx*8]
fst     mem_80
fst     st (2)
fstp    st (1)
```

Ostatni przykład faktycznie zdejmuje st (1) podczas gdy st(0) pozostaje na stosie.

Instrukcje fst i fstp ustawiają bit wyjątku stosu jeśli wystąpi niedomiar stosu (próbując przechować wartość pustego rejestru stosu). Ustawiają bit precyzji jeśli jest stracona precyzja podczas operacji przechowania (to nastąp, na przykład, kiedy przechowujemy wartość 80 bitową o podwyższonej precyzji w zmiennej pamięciowej 32 lub 64 bitowej i są gubione jakieś bity podczas konwersji). Ustawiają bit wyjątku niedomiaru kiedy przechowujemy wartość 80 bitową w zmiennej pamięciowej 32 lub 64 bitowej, ale wartość jest zbyt mała do przechowania w operandzie przeznaczenia. Podobnie, instrukcje te ustawia bit wyjątku przepełnienia jeśli wartość na szczycie stosu jest zbyt duża aby przechować ją w 32 lub 64 bitowej zmiennej pamięciowej. Instrukcje fst i fstp ustawiają flagę denormalizacji, kiedy próbujemy przechować wartość denormalizacyjną w 80 bitowym rejestrze lub zmiennej. Ustawiają flagę niepoprawnej operacji jeśli wystąpi niepoprawna operacja (taka jak przechowanie pustego rejestru). W końcu, instrukcje te ustawiają bit warunkowy C_1 jeśli wystąpi zaokrąglanie podczas operacji przechowania (wystąpi tylko kiedy przechowujmy w zmiennej pamięciowej 32 lub 64 bitowej i musimy zaokrąglić mantysę aby zmieściła się w miejscu przeznaczenia).

14.4.4.3 INSTRUKCJA FXCH

Instrukcja fxch wymienia wartość ze szczytu stosu z jednym z rejestrów FPU. Instrukcja ta przybiera dwie postacie: jedną z pojedynczym rejestrem FPU jako argumentem, drugą bez żadnych argumentów. Pierwsza postać wymienia szczyt stosu z określonym rejestrem. Druga postać fxch wymienia szczyt stosu z st(1).

Wiele instrukcji FPU, np. fsqrt działa tylko ze szczytem rejestru stosu. Jeśli chcemy wykonać taką operację na wartości , która nie jest na szczycie stosu, możemy użyć instrukcji fxch do wymiany tos z oryginalnym rejestrem. Poniższy przykład pobiera pierwiastek kwadratowy z st(2):

```
fxch    st (2)
fsqrt
fxch    st (2)
```

Instrukcja fxch ustawia bit wyjątku stosu jeśli stos jest pusty. Ustawia bit niepoprawnej operacji jeśli określimy pusty rejestr jako argument. Ta instrukcja zawsze zeruje bit kodu warunku C_1.

14.4.5 KONWERSJE

Chip 80x87 wykonuje wszystkie arytmetyczne operacje na rzeczywistych 80 bitowych wartościach. W takim sensie instrukcje fld i fst/ fstp są instrukcjami konwersji w równym stopniu jak instrukcjami przesuwania danych ponieważ one automatycznie konwertują pomiędzy wewnętrznym 80 bitowym rzeczywistym formatem a 32 lub 64 bitowym formatem pamięci. Pomimo to będziemy je klasyfikować jako operacje przesunięcia danych , zamiast konwersji, ponieważ przesuwają on wartości rzeczywiste do i z pamięci. FPU 80x87 dostarcza pięciu

podprogramów, które konwertują do lub z formatu liczb całkowitych lub BCD kiedy przesuwamy dane. Instrukcje te to fild, fist ,fistp, fbld i fbstp.

14.4.5.1 INSRUKJA FILD

Instrukcja fild (ładuj liczbę całkowitą) konwertuje 16 , 32 lub 64 bitową liczbę całkowitą uzupełnioną dwójkowo do 80 bitowego formatu o podwyższonej precyzji i odkłada wynik na stos. Instrukcja ta zawsze oczekuje pojedynczego operandu. Operand ten musi być adresem słowa, podwójnego słowa lub poczwórnego słowa zmiennej całkowitej. Chociaż format instrukcji dla fild używa dobrze znanych pól mod / rm, operand musi być zmienną pamięciową, nawet dla liczb całkowitych 16 i 32 bitowych. Nie możemy wyszczególnić jednego z 16 lub 32 bitowych rejestrów ogólnego przeznaczenia. Jeśli chcemy odłożyć rejestr ogólnego przeznaczenia 80x86 na stos FPU musimy najpierw przechować go w zmiennej pamięciowej a potem użyć fild do odłożenia tej wartości z tej zmiennej pamięciowej

Instrukcja fild ustawia bit wyjątku stosu. i C_1 (odpowiednio) jeśli wystąpi przepełnieni stosu podczas odkładania skonwertowanej wartości. Przykład:

```
fild    mem_16
fild    mem_32 [ecx*4]
fild    mm_64 [ebx + ecx*8]
```

14.4.5.2 INSTUKCE FIST I FISTP

Instrukcje fist i fistp konwertują zmienną 80 bitową o podwyższonej precyzji ze szczytu stosu do 16, 32 lub 64 bitowej liczby całkowitej i przechowują wynik w zmiennej pamięciowej określonej przez pojedynczy operand. Instrukcje te konwertują wartość z tos do liczby całkowitej według ustawień zaokrąglania w rejestrze sterującym FPU (bity 10 i 11). Instrukcje fist i fistp nie pozwalają nam określić jednego z rejestrów 16 lub 32 bitowych ogólnego przeznaczenia 80x86 jako operandów przeznaczenia

Instrukcja fist konwertuje wartość ze szczytu stosu do liczby całkowitej a potem przechowuje wynik; inaczej ni wpływa na rejestr stosu zmienno przecinkowego. Instrukcja fistp zdejmuje wartość z rejestru stosu zmienno przecinkowego po przechowaniu wartości skonwertowanej.

Instrukcje te ustawiają bit wyjątku stosu jeśli rejestr stosu zmienno przecinkowego jest pusty (również zeruje C_1). Ustawiają bity precyzji (operacje nieprecyzyjne) i C_1 jeśli wystąpi zaokrąglenie (to znaczy jeśli jest jakaś część ułamkowa wartości w st(0)). Instrukcje te ustawiają bit wyjątku niedomiaru jeśli wynik jest zbyt mały (tj. mniejszy niż jeden ale większy niż zero lub mniejszy niż zero ale większy niż –1). Przykład:

```
fist    mem_16 [bx]
fist    mem_64
fistp   mem_32
```

Nie zapomnijmy, że instrukcje te używają ustawień sterujących zaokrąglaniem określających jak będą konwertowane dane zmienno przecinkowe do liczb całkowitych podczas operacji zapamiętywania. Domyślnie, kontrola zaokrąglania jest zazwyczaj ustawiona na tryb „round”; mimo to większość programistów oczekuje od fist/ fistp obcinania części dziesiętnych podczas konwersji. Jeśli chcemy aby fist/ fistp obcinały wartości zmienno przecinkowe kiedy konwertują j do wartości całkowitych , musimy ustawić właściwie bity sterownia zaokrąglaniem w rejestrze sterującym zmienno przecinkowym.

14.4.5.3 INSTRUKCJE FBLD I FBSTP

Instrukcje fbld i fbstp ładują i przechowują 80 bitowe wartości BCD. Instrukcja fbld konwertuje wartość BCD do jej 80 bitego odpowiednika o podwyższonej precyzji i odkłada wynik na stos. Instrukcja fbstp zdejmuje rzeczywiste wartości o podwyższonej precyzji z tos, konwertuje na 80 bitową wartość BCD (zaokrąglając według bitów w rejestrze sterującym zmienno przecinkowym) i przechowuje skonwertowany wynik pod adresem określonym przez operand przeznaczenia pamięci. Zauważmy, że nie ma instrukcji fbst, która przechowuje na tos bez zdejmowania jej.

Instrukcja fbld ustawia bit wyjątku stosu i C_1 jeśli wystąpi przepełnienie stosu. stawia bit niepoprawnej operacji jeśli próbujmy załadować niepoprawną wartość BCD. Instrukcja fbstp ustawia bit wyjątku stosu i zeruje C_1 jeśli wystąpi niedomiar stosu (stos jest pusty). Ustawia flagę niedomiaru na takich samych warunkach jak fist i fistp. Przykłady:

```
;Zakładając mniej niż osiem pozycji na stosie, poniższa sekwencja kodu jest odpowiednikiem instrukcji fbst:
        fld     st (0)                  ;duplikowanie wartości na TOS
        fbstp   mem_80
;poniższy przykład łatwo konwertuje 80 bitową wartość BCD do 64 bitowej liczby całkowitej:
```

```
fbld    bcd_80          ;pobranie wartości BCD do konwersji
fist    mem_64          ;przechowanie jako liczby  całkowitej
```

14.4.6 INSTRUKCJE ARYTMETYCZE

Instrukcje arytmetyczne stanowią mały ale ważny podzbiór zbioru instrukcji 80x87. Instrukcje te dzielą się na dwie kategorie: te które działają na wartościach rzeczywistych i te które działają na wartościach rzeczywistych i całkowitych

14.4.6.1 INSTRUKCJE FADD I FADDP

Te dwie instrukcje przybierają następujące postacie:

```
fadd
faddp
fadd    st  (1),  st (0)
fadd    st (0), st (1)
faddp   st(1), st(0)
fadd    mem
```

Pierwsze dwie formy są równoważne. Zdejmują dwie wartości ze stosu, dodają je i odkładają ich sumę z powrotem na stos.

Następne dwie formy instrukcji fadd, te z dwoma argumentami rejestrów FPU, zachowują się jak instrukcja add 80x86. Dodają wartość z argumentu drugiego rejestru do wartości argumentu pierwszego rejestru. Zauważmy, że jednym z tych rejestrów musi być st (0).

Instrukcja faddp z dwoma operandami dodaje st (0) (które mus być zawsze drugim operandem) do operandu przeznaczenia (pierwszego) a potem zdejmuje st(0). Operand przeznaczenia musi być jednym z innych rejestrów FPU.

Ostatnia, powyższa postać fadd z operandem pamięci dodaje 32 lub 64 bitową zmienną zmienno przecinkową do wartości w st(0). Instrukcja ta skonwertuje 32 lub 64 bitowy operand do 80 bitowej wartości o podwyższonej precyzji przed wykonaniem dodawania. Zauważmy, że instrukcja ta nie wpływa na 80 bitowy operand pamięci..

Instrukcje te mogą zgłosić wyjątki stosu, precyzji, niedomiaru, przepełnienia, denormalizacji i niepoprawnej operacji. Jeśli wystąpi wyjątek zakłócenia stosu, C1 oznacza przepełnienie lub niedomiar.

14.4.6.2 INSTRUKCJE FSUB, FSUBP, FSUBR I FSUBRP

Te cztery instrukcje przybierają następujące formy:

```
fsub
fsubp
fsubr
fsubrp

fsub    st(1), st(0)
fsub    st(0), st(1)
fsubp   st(1), st(0)
fsub    mem

fsubr   st(1), st(0)
fsubr   st(0), st(1)
fsubrp  st(1), st(0)
fsubr   mem
```

Instrukcje fsub i fsubp bez żadnego operandu są identyczne. Zdejmują one st0) i st(1) z rejestru stosu, obliczają st0) – st(1) i odkładają różnicę z powrotem na stos. Instrukcje fsubr i fsubrp (odwrócone odejmowanie) działają w prawie identyczny sposób z wyjątkiem tego, że obliczają st(1) – st(0) i odkładają tą różnicę na stos.

Z dwoma argumentami rejestrowymi (przeznaczenie i źródło), instrukcja fsub oblicza przeznaczenie := przeznaczenie – źródło. Jednym z tych dwóch rejestrów musi być st(0). Z dwoma rejestrami jako operandami, fsubp również oblicza przeznaczenie := przeznaczenie – źródło a potem zdejmuje st(0) ze stosu po obliczeniu różnicy. Dla instrukcji fsubp, operandem źródłowym musi być st(0).

Z dwoma operandami rejestrów instrukcje fsubr i fsubrp działają w podobny sposób do fsub i fsubp, z wyjątkiem tego, że obliczają przeznaczenie := źródło – przeznaczenie.

Instrukcje fsub mem i fsubr mem akceptują 32 i 64 bitowe operandy pamięci. Konwertują one operand pamięci do 80 bitowej wartości o podwyższonej precyzji i odejmują tą postać st(0) (fsub) lub odejmują st(0) z tą wartością (fsubr) i przechowują wynik w st(0).

Instrukcje te mogą zgłaszać wyjątek stosu, precyzji, niedomiaru, przepełnienia, denormalizacji i niepoprawnej operacji. Jeśli wystąpi wyjątek zakłócenia stosu, C_1 oznacza przepełnienie lub niedomiar stosu.

14.4.6.3 INSTRUKCJE FMUL I FMULP

Instrukcje fmul i fmul mnożą dwie wartości zmienno przecinkowe. Instrukcje te mają następujące formy:

```
fmul
fmulp

fmul    st(0), st(1)
fmul    st(1), st(0)
fmul    mem

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

Bez żadnego operandu, fmul i fmulp robią to samo – zdejmują st(0) i st(1), mnożą te wartości i odkładają iloczyn z powrotem na stos. Instrukcja fmul z dwoma operandami rejestrowymi oblicza przeznaczenie := przeznaczenie * źródło. Jeden z rejestrów (źródło lub przeznaczenie)musi to być st(0).

Instrukcja fmulp st(i), st(0) oblicza st(i) := st(i) * st(0) a potem zdejmuje st(0). Instrukcja ta używa wartości dla i przed zdjęciem st(0). Instrukcja fmul mem wymaga 32 lub 64 bitowego operandu pamięci. Konwertuje określoną zmienną pamięciową do 80 bitowej wartości o podwyższonej precyzji i mnoży st(0) przez tą wartość.

Instrukcje te mogą zgłaszać wyjątki stosu, precyzji, niedomiaru, przepełnienia , denormalizacji i niepoprawnej operacji. Jeśli wystąpi zaokrąglenie podczas obliczania, instrukcje te ustawią bit kodu warunkowego C_1. Jeśli wystąpi wyjątek zakłócenia stosu, C1 oznacza przepełnienie lub niedomiar.

14.4.6.4 INSTRUKCJE FDIV, FDIVP, FDIVR I FDIVRP

Te cztery instrukcje przybierają następujące formy:

```
fdiv
fdivp
fdivr
fdivrp

fdiv    st(0), st(1)
fdiv    st(1), st(0)
fdivp   st(1), st(0)

fdivr   st0), st(1)
fdivr   st(1), st(0)
fdivrp  st(1), st(0)

fdiv    mem
fdivr   mem
```

Instrukcje fdiv i fdivp bez operandów zdejmują st(0) i st(1), obliczają st(0)/st(1) i odkładają wynik z powrotem na stos. Instrukcje fdivr i fdivrp również zdejmują st(0) i st(1) ale obliczają st(1)/st(0) przed odłożeniem ilorazu na stos.

Z dwoma operandami rejestrowymi instrukcje te obliczają poniższe ilorazy:

```
fdiv    st(0), st(1)        ;st(0) := st(0) / st(1)
fdiv    st(1), st(0)        ;st(1) := st(1) / st(0)
fdivp   st(1), st(0)        ;st(1) := st(1) / st(0)
fdivr   st(1), st(1)        ;st(0) := st(0) / st(1)
fdivrp  st(1), st(0)        ;st(0) := st(0) / st(1)
```

Instrukcje fdivp i fdivrp również zdejmują st(0) po wykonaniu operacji dzielenia. Wartość dla i w tych dwóch instrukcjach jest obliczana przed zdjęciem st0)

Instrukcje te mogą zgłosić wyjątki stosu, precyzji, niedomiaru, przepełnienia, denormalizacji, dzielenia przez zero i niepoprawnej operacji. Jeśli wystąpi zaokrąglani podczas obliczeń, instrukcje te ustawią bit kodu warunkowego C_1. Jeśli wystąpi wyjątek zakłócenia stosu, C_1 oznacza przepełnienie lub niedomiar stosu.

14.4.6.5 INSTRUKCJA FSQRT

Podprogram fsqrt nie pozwala na żadne operandy. Oblicza pierwiastek kwadratowy z wartości na tos i zastępuje st(0) tym wynikiem. Wartość na tos musi być zerem lub dodatnia, w innym przypadku fsqrt wygeneruje wyjątek niepoprawnej operacji.

Instrukcja ta zgłasza wyjątki stosu, precyzji, denormalizacji i niepoprawnej operacji. Jeśli wystąpi zaokrąglenie podczas obliczenia, fsqrt ustawi bit C_1. Jeśli wystąpi wyjątek zakłócenia stosu, C_1 oznacza przepełnienie lub niedomiar stosu.

Przykład:

```
; Obliczenie Z:= sqrt (x** + y**2);
                fld     x       ;ładuje X
                fld     st(0)   ;duplikuje X na TOS
                fmul            ;obliczenie x**2

                fld     y       ;załadowanie Y
                fld     st(0)   ;duplikowanie Y na TOS
                fmul            ;obliczenie Y**2

                fadd            ;obliczenie x**2 + y **2
                fsqrt           ;obliczenie sqrt (x**2+y**2)
                fst     Z       ;przechowanie wyniku w Z
```

14.4.6.6 INSTRUKCJA FSCALE

Instrukcja fscale zdejmuje dwie wartości ze stosu. Mnoży st(0) przez $2^{st(1)}$ i odkłada wynik z powrotem na stos. Jeśli wartość w st(1) nie jest całkowita, fscale obcina ją w stronę zera przed wykonaniem tej operacji.

Instrukcja ta zgłosi wyjątek stosu jeśli nie ma obecnie dwóch pozycji na stosie (również wyzeruje C_1 ponieważ wystąpi przepełnienie stosu). Zgłosi wyjątek precyzji jeśli zagubimy precyzję należąca do tj operacji to wystąpi jeśli st(1) zawiera dużą, ujemną wartość).Podobnie ta instrukcja ustawi bit wyjątku przepełnienia lub niedomiaru jeśli pomnożymy st(0) przez dużą dodatnią lub ujemną potęgę dwójki. Jeśli wynik mnożenia jest bardzo mały, fscale może ustawić bit denormalizacji. Instrukcja to może również ustawić bit niepoprawnej operacji jeśli próbujemy użyć niewłaściwej wartości. Fscale ustawi C_1 jeśli wystąpi zaokrąglenie w skądinąd poprawnym obliczeniu. Przykład:

```
                fild    Sixteen         ;odłożenie sixteen na stos
                fld     X               ;obliczenie x*(2**16)
                fscale
                -
                -
                -
Sixteen         word    16
```

14.4.6.7 INSTRUKCJE FPEM I FPEM1

Instrukcje fprem i fpem1 obliczają resztę częściową. Intel zaprojektował instrukcję fprem przed tym zanim IEEE sfinalizował swój standard zmienno przecinkowy. W końcowym projekcie standardu zmienno przecinkowego IEE, definicja fprem trochę różniła się od Intelowskiego projektu. Niestety, Intel musiał utrzymać zgodność z istniejącym oprogramowaniem, które używało instrukcji fprem, więc zaprojektowali nową wersję obsługującą operację częściowej reszty IEEE, fpem1. Powinniśmy zawsze używać fprem1 w programach jakie piszemy, dlatego też tylko będziemy omawiać tutaj fpem1, chociaż fprem używamy w podobny sposób.

Fprem1 oblicza częściową resztę z st(0) / st(1). Jeśli różnica pomiędzy wykładnikami st(0) i st(1) jest mniejsza niż 64, fprem1 może obliczyć dokładną resztę w jednej operacji. W innym przypadku będziemy musieli wykonać fprem1 dwa lub więcej razy dla uzyskania poprawnej wartości reszty. Bit kodu warunkowego C2 określa kiedy obliczenie jest ukończone. Zauważmy, że fprem1 nie zdejmuje dwóch operandów ze stosu;

pozostawia częściową resztę w st(0) a oryginalny dzielnik w st(1) w przypadku kiedy musimy obliczyć inny cząstkowy iloraz dla kompletnego wyniku.

Instrukcja fpem1 ustawia flagę wyjątku stosu jeśli nie ma dwóch wartości na szczycie stosu. Ustawia bity wyjątku przepełnienia i denormalizacji jeśli wynik jest zbyt mały .Ustawia bit nieoprawnej operacji jeśli wartości na tos są niewłaściwe. Ustawia bit C_2 jeśli operacja reszty cząstkowej nie jest ukończona. W końcu, ładuje C_3,C_1 i C_0 bitami zero, jeden i dwa ilorazu, odpowiednio .
Przykład:

```
;Obliczamy Z := X mod Y
                fld     y
                fld     x
PartialLp:      fprem1
                fstsw   ax              ;pobranie bitu warunku  w ax
                test    ah, 100b        ;zobacz czy C2 jest ustawione
                jnz     PartialLp       ;powtórz jeśli jeszcze nie zrobione
                fstp    Z               ;przechowanie reszty
                fstp    st(0)           ;zdjęcie starej wartości y
```

14.4.6.8 INSTRUKCJA FRNDINT

Instrukcja frndint zaokrągla wartość na tos do najbliższej wartości całkowitej używając algorytmu określonego w rejestrze sterującym.

Instrukcja ta ustawia flagę wyjątku stosu jeśli nie ma wartości na tos (również zeruje C_1 w tym przypadku). Ustawia bity wyjątków precyzji i denormalizacji jeśli była utrata precyzji. Ustawia flagę niepoprawnej operacji jeśli wartość na tos nie jest poprawną liczbą.

14.4.6.9 INSTRUKCJA FXTRACT

Instrukcja fxtract jest uzupełnieniem instrukcji fscale. Zdejmuje wartość ze stosu i odkłada wartość która jest całkowitym odpowiednikiem wykładnika (w 80 bitowej rzeczywistej postaci), a potem odkłada mantysę z zerowym wykładnikiem (3fffh w formie obarczonej błędem).

Instrukcja zgłasza wyjątek stosu jeśli jest niedomiar stosu kiedy zdejmujemy wartość oryginalną lub przepełnienie stosu kiedy odkładamy dwa wyniki (C_1 określa czy wystąpiło przepełnienie czy niedomiar). Jeśli pierwotnie szczyt stosu był zerem, fxtract ustawia flagę wyjątku dzielenia przez zero. Flaga denormalizacji jest ustawiana jeśli wynik to gwarantuje; a flaga niepoprawnej operacji jest ustawiana jeśli mamy niepoprawne wartości wejściowe, kiedy wykonujemy fxtract.
Przykład:

```
;Poniższy przykład wyciąga wykładnik binarny z X i  przechowuje go w 16 bitowej zmiennej całkowitej
;Xponent

                fld     x
                fxtract
                fstp    st(0)
                fistp   Xponent
```

14.4.6.10 INSTRUKCJA FABS

Fabs oblicza wartość bezwzględną st(0) przez wyzerowanie bitu znaku st(0). Ustawia bit wyjątku stosu i niepoprawnej operacji jeśli stos jest pusty.

Przykład:

```
;Obliczamy X := sqrt(abs(x));
                fld     x
                fabs
                fsqrt
                fstp    x
```

14.4.6.11 INSTRUKCA FCHS

Fchs zmienia znak wartości st(0) przez odwrócenie jego bitu znaku. Ustawia bit wyjątku stosu i niepoprawnej operacji jeśli stos jest pusty. Przykład:

```
;Obliczamy  X := -X jeśli X jest dodatnie, X := X jeśli x  jest ujemne
                fld     x
                fabs
                fchs
                fstp    x
```

14.7 INSTRUKCJE PORÓWNAŃ

80x87 dostarcza kilku instrukcji dla porównywania wartości rzeczywistych. Instrukcje fcom, fcomp, fcompp, fucom, fucomp i fucompp porównują dwie wartości na szczycie stosu i ustawiają właściwe kody warunkowe. Instrukcja ftst porównuje wartość ze szczytu stosu z zerem. Instrukcja fxam sprawdza wartość na tos i przekazuje informacje o znaku, normalizacji i znaczniku.

Generalnie, większość programów testuje bity kodów warunków bezpośrednio po porównaniu. Niestety, nie ma żadnych instrukcji skoków warunkowych , które wykonywały by rozgałęzienia w oparciu o kody warunków FPU. Zamiast tego możemy użyć instrukcji fstsw do skopiowani rejestru stanu do rejestru ax; potem możemy użyć instrukcji sahf do skopiowania rejestru ah do bitów kodów warunków 80x86. Po wykonaniu tego, możemy użyć instrukcji skoków warunkowych do testowania jakiegoś warunku. Ta technika kopiuje C_0 do flagi przeniesienia, C_2 do flagi parzystości a C_3 do flagi zera. Instrukcja sahf nie kopiuje C_1 do żadnego bitu flagi 80x86.

Ponieważ instrukcja sahf nie kopiuje żadnego bitu stanu procesora 80x87 do flagi znaku lub przepełnienia, nie możemy użyć instrukcji jg, jl, jge lub jle . Zamiast tego, użyjemy instrukcji ja, jae, jb, jbe, je i jz kiedy testujemy wyniki porównań zmienno przecinkowych. Tak , te skoki warunkowe zwykle testują wartości bez znaku a liczby zmienno przecinkowe są wartościami ze znakiem. Jednakże, używamy bez znakowych rozgałęzień warunkowych; instrukcje fstsw i sahf ustawiają rejestr flag 80x86 do stosowania skoków bez znaku

14.4.7.1 INSTRUKCJE FCOM, FCOMP I FCOMPP

Instrukcje fcom, fcomp i fcompp porównują st(0) do określonego operandu i ustawiają odpowiedni bit warunkowy 80x87 w wyniku porównania. Poprawne formy tych instrukcji to:

```
fcom
fcomp
fcompp

fcom    st(1)
fcomp   st(1)

fcom    mem
fcomp   mem
```

Bez żadnych operandów, fcom, fcomp i fcompp porównują st(0) z st(1) i ustawiają stosownie flagi procesora. Dodatkowo fcomp zdejmuje st(0) ze stosu a fcompp zdejmuje ze stosu i st(0) i st(1).

Z pojedynczym operandem rejestrowym, fcom i fcomp porównują st(0) z określonym rejestrem. Fcomp również zdejmuje st(0) po porównaniu.

Z 32 lub 64 bitowymi operandami pamięci, instrukcje fcom i fcomp konwertują zmienną pamięciową do wartości 80 bitowej o podwyższonej precyzji a potem porównuje st(0) z tą wartością, ustawiając odpowiednio bity kodów warunkowych. Fcomp również zdejmuje st90) po porównaniu.

Instrukcje te ustawiają C_2 (który kończy się we fladze parzystości) jeśli dwa operandy są nieporównywalne (np. NaN). Jeśli jest możliwe dla niepoprawnej wartości zmienno przecinkowej zakończenie porównania, powinniśmy sprawdzić flagę parzystości na obecność błędu przed sprawdzeniem żądanego warunku.

Instrukcje te ustawiają bit zakłócenia stosu jeśli nie ma dwóch pozycji na szczycie rejestru stosu. Ustawiają bit wyjątku denormalizacji jeśli któryś lub oba operandy są całkowicie NaN. Instrukcje te zawsze zerują kod warunkowy C_1.

14.4.7.2 INSTRUKCJE FUCOM, FUCOMP I FUCOMPP

Instrukcje te są podobne do instrukcji fcom, fcomp i fcompp, chociaż przybierają tylko takie formy:

```
fucom
```

```
fucomp
fucompp
fucom  st(1)
fucomp  st(1)
```

Różnica pomiędzy fcom/fcomp/fcompp a fucom/fucomp/fucompp jest stosunkowo mała. Instrukcje fcom/fcomp/fcompp ustawiają bit wyjątku niepoprawnej operacji, jeśli porównuje dwa NaN'y. Instrukcje fucom/fucomp/fucompp nie. W pozostałych przypadkach te dwa zbiory instrukcji działają identycznie.

14.4.7.3 INSTRUKCJA FTST

Instrukcja ftst porównuje wartość w st(0) z 0.0. Działa podobnie jak instrukcja fcom, jeśli st(1) zawierałoby 0.0.Zauważmu, że instrukcja ta nie rozróżnia -0.0 od +0.0. Jeśli wartość w st(0) jest jedną z tych wartości, ftst ustawi C3 oznaczającą równość. Jeśli potrzebujemy rozróżnić miedzy –0.0 a +0.0 użyjemy instrukcji fxam. Zauważmy ,że instrukcja ta nie zdejmuje st(0) ze stosu.

14.4.7.4 INSTRUKCJA FXAM

Instrukcja fxam bada wartość w st(0) i odnotowuje wynik w bitach kodu warunkowego (zobacz „Rejestr stanu FPU aby zobaczyć jak fxam ustawia te bity), Instrukcja ta nie zdejmuje st(0) ze stosu.

14.4.8 INSTRUKCJE STAŁE

FPU 80x87 dostarcza kilku instrukcji, które pozwalają nam załadować powszechnie używane stałe do rejestru stosu FPU. Instrukcje te ustawiają flagi zakłócenia stosu, niepoprawnej operację i C_1 jeśli wystąpi przepełnienie stosu; w innym przypadku nie wpływają na flagi FPU.

Do instrukcji z tej kategorii zaliczamy:

```
fldz            ;odłożenie +0.0
fldl            ;odłożenie +1.0
fldpi           ;odłożenie π
fldl2t          ;odłożenie log2 (10)
fldl2e          ;odłożenie log2 (e)
fldlg2          ;odłożenie log10 (2)
fldln2          ;odłożenie ln (2)
```

14.4.9 INSTUKCJE PRZESTĘPNE

80387 i późniejsze FPU dostarcza osiem instrukcji przestępnych (logarytmicznych i trygonometrycznych) do obliczenia częściowego tangensa, arctangensa, $2^x - 1$, $y^* \log_2(x)$, i $y^* \log_2(x+1)$.Używając różnych algebraicznych identyfikatorów, łatwo jest obliczyć większość z pozostałych powszechnych funkcji przestępnych używając tych instrukcji

---

14.4.9.1 INSTRUKCJA F2XM1

F2xm1 oblicza $2^{st(0)} - 1$. Wartość w st(0) musi być z zakresu $-1.0 \le st(0) \le +1.0$. Jeśli st(0) jest poza zakresem f2xm1 wygeneruje wynik nieokreślony ale nie zgłosi wyjątku. Wyliczona wartość zamieni wartość w st(0). Przykład:

;Obliczamy 10^x używając identyfikatora : $10x = 2^{x*\lg(10)}$ ($\lg = \log_2$)

```
fld     x
fldl2t
fmul
f2xm1
fldl
fadd
```

Zauważmy, że f2xm1 oblicza $2^x - 1$, a powyższy kod dodaje 1.0 do wyniku na końcu obliczenia.

14.4.9.2 INSTRUKCJE FSIN, FCOS I FSINCOS

Instrukcje te zdejmują wartość ze szczytu rejestru stosu i obliczają sinus, cosinus lub oba i odkłada wynik(i) z powrotem na stos. Fsincos odkłada sinus po którym następuje cosinus z oryginalnego argumentu, i przechowuje cos(st(0)) w st(0) a sin(st(0)) w st(1).

Instrukcje te zakładają, że st(0) określa kąt w radianach, a ten kąt musi być w zakresie $-2^{63} < st(0) < +2^{63}$. Jeśli oryginalny operand jest poza zakresem, instrukcje te ustawiają flagę C_2 i pozostawiają st(0) nie zmienione. Możemy użyć instrukcji fprem1 z dzielnikiem 2π , do zredukowania argumentu do stosownego zakresu .

Instrukcje te ustawiają flagi zakłócenia stosu/ C_1, precyzji, niedomiaru, denormalizacji i niepoprawnej operacji według wyniku obliczenia.

14.4.9.3 INSTRUKCA FPTAN

Fptan oblicza tangens z st(0) i odkłada tą wartość a potem odkłada 1.0 na stos. Podobnie jak instrukcje fsin i fcos, wartość st(0) jest przyjmowana w radianach i musi być w zakresie $-2^{63} < st(0) < +2^{63}$. Jeśli wartość jest poza zakresem, fptan ustawia C_2 aby wskazywał ,że konwersja nie miał miejsca. Podobnie jak przy instrukcjach fsin, fcos i fsincos, możemy zastosować instrukcję fprem1 do zredukowania tego argumentu do właściwego zakresu używając dzielnika 2π.

Jeśli argument jest niewłaściwy (tj. zero lub π radianów, które powodują dzielnie przez zero) wynik jest niezdefiniowany a instrukcja ta nie zgłasza żadnych wyjątków. Fptan ustawi zakłócenie stosu, precyzję, niedomiar, denormalizację, niepoprawną operację , bity C_2 i C_1 jakie są wymagane przy operacji.

14.4.9.4 INSTRUKCJA FPATAN

Instrukcja ta oczekuje dwóch wartości na szczycie stosu. Zdejmuje je i oblicza co następuje:

$$st(0) = \tan^{-1}(st(1) / st(0))$$

Wartość wynikowa jest arctangensem współczynnika na stosie wyrażonym w radianach. Jeśli mamy wartość z jakiej życzymy sobie obliczyć tangens, użyjemy fld1 do stworzenia właściwego współczynnika a potem wykonamy instrukcję fpatan.

Instrukcja ta wpływa na bity zakłócenia stosu / C_1, precyzji , niedomiaru, denormalizacji i niepoprawnej operacji jeśli wystąpi problem podczas obliczenia. Ustawia bit kodu warunkowego C_1 jeśli musi zaokrąglić wynik.

14.4.9.4 INSTRUKCJE FYL2X I FYL2XP1

Instrukcje fyl2x i fyl2xp1 obliczają odpowiednio st(1)* $\log_2(st(0))$ i st(1)* $\log_2(st(0)+1)$. Fyl2x wymaga, żeby st(0) był większy od zera, fyl2xp1wymaga aby st(0) było z zakresu:

$$\left(-1-\left(\frac{\sqrt{2}}{2}\right)\right) < st(0) < \left(1-\left(\frac{\sqrt{2}}{2}\right)\right)$$

Fyl2x jest użyteczne przy obliczaniu logarytmów opartych na podstawie innej niż dwa; fyl2xp1 jest użyteczne dla obliczenia procentów składanych przy zachowaniu maksimum precyzji podczas obliczenia.

Fyl2x może wpływać na wszystkie flagi wyjątków. C1 oznacza zaokrąglanie jeśli nie ma innych błędów, przepełnieni / niedomiar stosu jeśli jest ustawiony bit zakłócenia stosu.

Instrukcja fyl2xp1 nie wpływa na flagi wyjątków przepełnienia i dzielenia przez zero. Wyjątki te występują jeśli st(0) jest bardzo małe lub wynosi zero. Ponieważ fyl2xp1 dodaje jeden do st(0) przed obliczeniem funkcji, ten warunek nigdy się nie spełni. Fyl2xp1 wpływa na inne flagi w sposób identyczny jak fyl2x.

14.4.10 INSTRUKCJE RÓŻNE

FPU 80x87 zawiera kilka dodatkowych instrukcji, które sterują FPU, synchronizują działania i pozwalają nam testować i ustawiać różne bity stanu. Instrukcje te zawierają finit / fninit, fdisi / fndisi, feni / fneni , fldcw, fstcw / fnstcw, fclex / fnclex, fsave / fnsave , frstor, frstpm, fstsw / fnstsw, fstenv/ fnstenv, fldenv, fincstp, fdecstp, fwait, fnop i ffree. Fdisi / fndisi , feni / fneni i frstpm są aktywne tylko na FPU wcześniejszych niż 80387, więc nie będziemy ich rozpatrywać tutaj.

Wiele z tych instrukcji ma dwie formy. Pierwszą formą jest Fxxxx a drugą Fnxxxx. Wersja bez „N" wysyła wcześniej opcodu instrukcji fwait (która jest standardem dla większości instrukcji koprocesora). Wersja z „N" nie wysyła opcodu fwait („N" oznacza „no wait" –żadnego czekania)

14.4.10.1 INSTRUKCJE FINIT I FNINIT

Instrukcja finit inicjalizuje FPU dla właściwej operacji. Nasza aplikacja powinna wykonać tą instrukcję przed wykonaniem każdej innej instrukcji FPU. Instrukcja ta inicjalizuje rejestr sterujący 37Fh, rejestr stanu zerem a znacznik słowa 0FFFFh. Inne rejestry są niezmienione.

14.4.10.2 INSTRUKCJA FWAIT

Instrukcja fwait pauzuje system dopóki nie zakończy się wykonywanie bieżącej instrukcji FPU. Jest to wymagane ponieważ FPU na 80486 i wcześniejszych związkach CPU/FPU może wykonywać instrukcje równolegle z CPU. Dlatego też instrukcje FPU które odczytują lub zapisują do pamięci mogą cierpieć n zagrożenie danych jeśli główne CPU uzysk dostęp do tej samej komórki pamięci przed odczytem lub zapisem tej komórki przez FPU. Instrukcja fwait pozwala nam zsynchronizować działania z FPU poprzez oczekiwani dopóki nie zakończy się bieżąca instrukcja FPU. O rozwiązuje zagrożenie danych przez, skuteczne wprowadzenie „opóźnienia' do wykonywanego strumienia.

14.4.10.3 INSTRUKCJE FLDCW I FSTCW

Instrukcje fldcw i fstcw wymagają pojedynczego 16 bitowego operandu pamięci:

```
fldcw    mem_16
fstcw    mem_16
```

Te dwie instrukcje ładują rejestr sterujący z komórki pamięci (fldcw) lub przechowują słowo sterujące w 16 bitowej komórce pamięci (fstcw)

Kiedy używamy instrukcji fldcw do włączenia jednego z wyjątków, jeśli odpowiadająca flaga wyjątku jest ustawiona kiedy dopuszczamy ten wyjątek, FPU będzie generował natychmiastowe przerwanie przed wykonaniem przez CPU następnej instrukcji. Dlatego też powinniśmy używać instrukcji fclex do wyzerowania każdego przerwania w toku przed zmienieniem bitów dopuszczających wyjątki FPU.

14.4.10.5 INSTRUKCJE FLDENV, FSTENV I FNSTENV

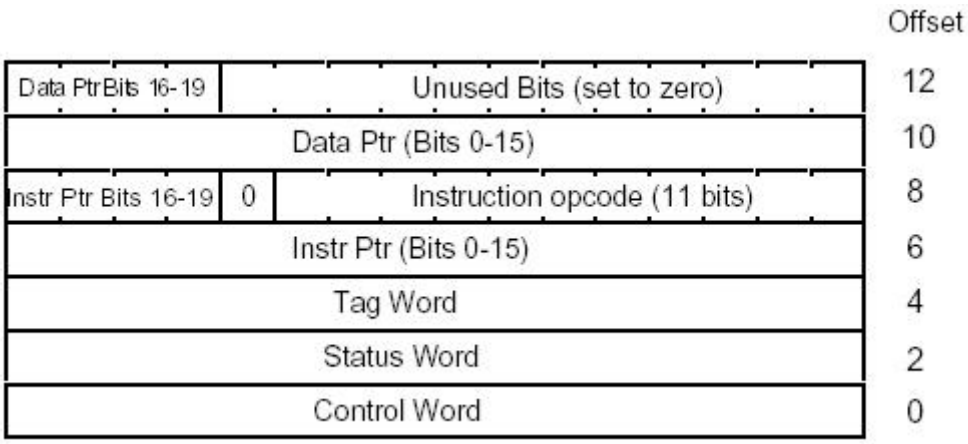
```
fstenv    mem_14b
fnstenv   mem_14b
fldenv    mem_14b
```

Instrukcje fstenv / fnstenv przechowują 14 bitowy rekord środowiskowy FPU w określonym operandzie pamięci. Kiedy działamy w trybie rzeczywistym (jedyny tryb rozważany w tym tekście), rekord środowiskowy przybiera postać jak na rysunku 14.11

Musimy wykonywać instrukcje fstenv i fnstenv przy wyłączonych przerwaniach CPU. Co więcej powinniśmy zawsze być pewni ,że FPU nie jest zajęta przed wykonaniem tej instrukcji. Jest to łatwe do wykonania przy zastosowaniu następującego kodu:

```
pushf                   ;przechowanie flagi I
cli                     ;wyłączenie przerwań
fstenv    mem_14b       ;niejawne oczekiwanie czy nie zajęty
fwait                   ;oczekiwanie na koniec operacji
popf                    ;przywrócenie flagi I
```

Instrukcja fldenv ładuje środowisko FPU z określonej komórki pamięci. Zauważmy, że instrukcja ta pozwala nam załadować słowo stanu. Nie ma żadnej jasnej instrukcji, takiej jak fldcw do wykonania tego.



Rysunek 14.11 Rekord środowiskowy FPU (16 bitowy tryb rzeczywisty)

14.4.10.6 INSTRUKCJE FSAVE, FNSAVE I FRSTOR

```
fsave   mem_94b
fnsave  mem_94b
frstor  mem_94b
```

Instrukcje te zachowują i przywracają stan FPU. Wymaga to zachowania wszystkich wewnętrznych rejestrów sterujących, stanu i danych. Lokacja przeznaczenia dla fsave / fnsave (lokacja źródłowa dla fstor) musi by długa na 94 bajty. Pierwsze 14 bajtów odpowiada użyciu rekordów środowiska instrukcji fldenv i fstenv; pozostałe 80 bajtów przechowuje dane z rejestru stosu FPU wypisanych od st(0) do st(7). Frstor przeładowuje rekord środowiska i rejestry zmienno przecinkowe z określonego operandu pamięci.

Instrukcje fsave / fnsave i frstor są głównie przeznaczone dla przełączania zadań. Możemy również użyć fsave / fnsave i frstor jako sekwencji „zdejmij wszystko" i „odłóż wszystko" dl zachowania stanu FPU.

Podobnie jak przy instrukcjach fstenv i fldenv przerwania powinny być wyłączone podczas zachowywania lub przywracania stanu FPU. W przeciwnym razie inny podprogram obsługi przerwań może manipulować rejestrami FPU i unieważnić działanie operacji fsave / fnsave i frstor. Poniższy kod właściwie ochrania dane środowiskowe podczas zachowywania i odtwarzania statusu FPU:

```
;Zachowanie stanu FPU, zakładamy, że di wskazuje na rekord środowiska w pamięci.
        pushf
        cli
        fsave   [si]
        fwait
        popf
        -
        -
        -
        pushf
        cli
        frstor  [si]
        fwait
        popf
```

14.4.10.7 INSTRUKCE FSTSW I FNSTSW

```
fstsw   ax
fnstsw  ax
fstsw   mem_16
fnstsw  mem_16
```

Instrukcje te przechowują rejestr stanu FPU w 16 bitowej komórce pamięci lub rejestrze ax. Instrukcje te są niezwykłe w tym sensie, że mogą one kopiować wartość FPU do jednego z rejestrów ogólnego przeznaczenia 80x86. Oczywiście, celem poza zezwoleniem na przesłanie rejestru stanu do ax jest zezwolenie CPU na łatwe testowani rejestru kodów warunku instrukcją sahf.

14.4.10.9 INSTRUKCJE FINCSTP I FDECSTP

Instrukcje fincstp i fdecstp nie pobierają żadnych argumentów. Po prostu zwiększają i zmniejszają bit wskaźnika stosu (mod 8) w rejestrze stanu FPU. Te dwie instrukcje zerują flagę C_1 ale nie wpływają na inne bity kodu warunkowego w rejestrze stanu FPU.

14.4.10.9 INSTRUKCJA FNOP

Instrukcja fnop jest po prostu synonimem dla fst st, st(0). Nie wykonuje żadnych działań na FPU

14.4.10.10 INSTRUKCJA FFREE

```
ffree    st (1)
```

Instrukcja ta modyfikuje bity znacznika dla rejestru i w rejestrze znaczników oznaczając określony rejestr jako pusty. Wartość jest nienaruszona przez tą instrukcję, ale FPU już nie może uzysk dostępu do tej danej (bez przestawienia właściwych bitów znacznika)

14.4.11 OPERACJE CAŁKOWITE

FPU 80x7 dostarcza specjalnych instrukcji, które łączą liczby całkowite przekształcone do podwyższonej precyzji wraz z różnymi operacjami arytmetycznymi i porównań. Instrukcje te to:

```
fiadd   int
fisub   int
fisubr  int
fimul   int
fidiv   int
fidivr  int

ficom   int
ficomp  int
```

Instrukcje te konwertują swoje 16 lub 32 bitowe argumenty całkowite na 80 bitową wartość zmienno przecinkową o podwyższonej precyzji a potem używają tej wartości jako operandu źródłowego dla określonych działań. Instrukcje te używają st(0) jako argumentu przeznaczenia.

14.8 POSUMOWANIE

Dla wielu aplikacji arytmetyka liczb całkowitych ma dwie wady nie do pokonania – nie jest łato przedstawiać wartości ułamkowe z liczbami całkowitymi i liczby całkowite mają ograniczony zakres dynamiki. Arytmetyka zmienno przecinkowa dostarcza przybliżenia do rzeczywistej arytmetyki, która pokonuje te dwa ograniczenia.

Arytmetyka zmienno przecinkowa jednakże też ni jest pozbawiona własnych problemów. Arytmetyka zmienno przecinkowa cierpi z powodu ograniczonej precyzji. W wyniku tego może wkraść się niedokładność do obliczeń. Dlatego też, arytmetyka zmienno przecinkowa nie do końca stosuje zasady zwykłej algebry. Jest pięć ważnych zasad do zapamiętania, kiedy używamy arytmetyki zmienno przecinkowej: (1) Kolejność wyliczenia może wpływać na wynik (2) Kiedykolwiek dodajemy lub odejmujemy liczby, dokładność wyniku może być mniejsza niż precyzja dostarczona przez format zmienno przecinkowy (3) Kiedy wykonujemy łańcuch obliczeń dodawania, odejmowania, mnożenia i dzielenia, próbujmy wykonywać najpierw mnożenie i dzielenie (4) Kiedy mnożymy i dzielimy wartości, próbujemy mnożyć duże i małe liczby razem najpierw, a potem próbujmy dzielić liczby według tych samych względnych rozmiarów (5) Kiedy porównujemy dwie liczy zmienno przecinkowe, zawsze pamiętajmy że błąd może wkraść się do obliczenia, dlatego też powinniśmy sprawdzać czy jedna wartość mieści się wewnątrz pewnego zakresu drugiej.

* Matematyczna arytmetyka zmienno przecinkowa

Intel wcześnie rozpoznał potrzebę sprzętowej jednostki zmienno przecinkowej. Wynajęli trzech matematyków do zaprojektowania bardzo dokładnego formatu zmienno przecinkowego i algorytmów dla ich

rodziny FPU 80x87. Formaty te, z drobnymi modyfikacjami, stały się standardami zmienno przecinkowymi IEEE 754 i IEEE 854. Standard IEEE w rzeczywistości dostarcza trzech różnych formatów: 32 bitowego formatu o standardowej precyzji, 64 bitowego formatu o podwójnej precyzji i formatu o podwyższonej precyzji. Intel zaimplementował format o podwyższonej precyzji używając 80 bitów. 32 bitowy format używa 24 bitowej mantysy (najbardziej znaczący bit jest niejawną jedynką i nie jest przechowywany w 32 bitach), ośmio bitowego stałego 127 wykładnika i jednego bitu znaku. 64 bitowy format dostarcza 53 bitowej mantysy (ponownie najbardziej znaczący bit jest zawsze jedynką i nie jest przechowywany w wartości 64 bitowej), 11 bitów dodatkowego 1023 wykładnika i jednego bitu znaku. 80 bitowy format o podwyższonej precyzji używa 64 bitowego wykładnika, 15 bitów dodatkowego 16363 wykładnika i pojedynczego bitu znaku.

* Format zmienno przecinkowy IEEE

Chociaż FPU 80x87 i CPU z wbudowanym FPU (80486 i Pentium) stają się bardzo popularne, jest całkiem możliwe, że będziemy musieli wykonać kod, który używa arytmetyki zmienno przecinkowej na maszynie bez FPU. W takim przypadku będziemy potrzebować wsparcia podprogramów dla wykonania arytmetyki zmienno przecinkowej. A szczęście Standardowa Biblioteka UCR dostarcza zbioru podprogramów zmienno przecinkowych, które możemy wywołać. Biblioteka Standardowa wprowadza podprogramy do ładowania i przechowywania wartości zmienno przecinkowych, konwersji pomiędzy liczbami całkowitymi a formatem zmienno przecinkowym, dodawania, odejmowania, mnożenia i dzielenia wartość zmienno przecinkowych, konwersji pomiędzy wartościami ASCII a zmienno przecinkowymi i wyprowadzania wartości zmienno przecinkowych. Nawet jeśli mamy zainstalowany FPU, podprogramy konwertujące i wyjściowe Biblioteki Standardowej są całkiem użyteczne.

* Podprogramy zmienno przecinkowe Biblioteki Standardowej UCR

Dla szybkiej arytmetyki zmienno przecinkowej, oprogramowanie nie ma szans przeciwko sprzętowi. FPU 80x87 dostarczają szybkich operacji zmienno przecinkowych poprzez rozszerzenie zbioru instrukcji 80x86 do działania z arytmetyką zmienno przecinkową. Dodatkowo do tych nowych instrukcji, FPU 80x87 dodaje osiem nowych rejestrów danych, rejestr sterujący, rejestr stanu i kilka innych rejestrów wewnętrznych. Rejestry danych FPU, w odróżnieniu od rejestrów ogólnego przeznaczenia 80x86 są zorganizowane w stos. Chociaż jest możliwe działanie na tych rejestrach tak jak by były one rejestrami standardowymi, większość aplikacji FPU używa mechanizmu stosu kiedy oblicza wynik zmienno przecinkowy. Rejestr sterujący FPU pozwala nam inicjalizować FPU 80x87 jednym z kilku różnych trybów. Rejestr sterujący pozwala nam ustawić nam sterowani zaokrąglaniem, precyzję dostępną podczas obliczania, i wybór tego jaki wyjątek przyczyni się do przerwania. Rejestr stanu 80x87 raportuje bieżący stan FPU. Rejestr ten dostarcza bitów które określają czy FPU jest aktualnie zajęty, określa czy poprzednia instrukcja wygenerowała wyjątek, określa liczę fizycznych rejestrów na szczycie rejestru stosu i dostarcza kodów warunkowych FPU

* Koprocesor zmienno przecinkowy 80x87
* Rejestry FPU
* Rejestry danych FPU
* Rejestr sterujący FPU
* Rejestr stanu FPU

W dodatku do typów danych IEEE o pojedynczej, podwójnej i podwyższonej precyzji, FPU 80x87 również wspiera różne całkowite i BCD typy danych. FPU będzie automatycznie konwertować do i z tych typów danych kiedy ładuje i przechowuje takie wartości.

* Typy danych FPU

FPU 80x87 dostarcza szerokiego zakresu operacji zmienno przecinkowych poprzez zwiększenie zbioru instrukcji 80x86. Możemy zaklasyfikować instrukcje FPU w osiem kategorii: przesunięcia danych, konwersji, instrukcji arytmetycznych, instrukcji porównań, instrukcji stałych, instrukcji przestępnych, instrukcji różnych i instrukcji całkowitych.

* Zbiór instrukcji FPU
* Instrukcje przesunięcia danych FPU
* Konwersje
* Instrukcje arytmetyczne
* Instrukcje porównań
* Instrukcje stałe

* Instrukcje przestępne
* Instrukcje różne
* Operacje całkowite

Chociaż 80387 i późniejsze FPU dostarczają bogaty zbiór funkcji przestępnych, jest wiele funkcji trygonometrycznych , inwersji trygonometrycznych , wykładniczych i logarytmicznych nieobecnych w tym zbiorze instrukcji. Jednakże, brakujące funkcje lato jest zsyntetyzować używając identyfikatorów algebraicznych. Ten rozdział dostarczył kodu źródłowego dla wielu z tych podprogramów jako przykład programowania FPU.

14.9 PYTANIA

1) Dlaczego nie można zastosować zwykłych zasad algebry do arytmetyki zmienno przecinkowej?
2) Oda przykład sekwencji operacji w których kolejność obliczeń tworzy różne wyniki arytmetyki o skończonej precyzji
3) Wyjaśnij dlaczego operacje dodawania i odejmowania o ograniczonej precyzji mogą powodować zagubienie precyzji podczas obliczeń.
4) Dlaczego powinniśmy ,jeśli to możliwe, wykonywać najpierw mnożenie i dzielenie w obliczeniach zawierających mnożenie i dzielenie jak również dodawanie i odejmowanie?
5) Wyjaśnij różnice pomiędzy wartością zmienno przecinkową znormalizowana, nieznormalizowaną i denormalizowaną
6) Używając Biblioteki Standardowej UCR skonwertuj następujące wyrażenia do kodu asemblerowego 80x86 zakładając ,że wszystkie zmienne są wartościami 64 bitowej podwójnej precyzji). Aby być pewnym wykonania koniecznych manipulacji załóż maksymalną dokładność. Możesz założyć, że wszystkie zmienne są z zakresu ±1e-110...± e+10:
   a) Z := X*X + Y*Y b) Z:= (X-Y)* Z
   c) Z := X*Y - X/Y d) Z:= (X+Y)/(X-Y)
   e) Z:= (X*X)/(Y*Y) f) Z:=X*X + Y +1.0
7) Skonwertuj powyższe instrukcje do kodu FPU 80x87
8) Następujących problemów dostarczają definicje hiperbolicznych funkcji trygonometrycznych. Zakoduj każdą z nich używając instrukcji FPU 80x87 i podprogramów exp(x) i ln(x) pokazanych w tym rozdziale

a) $\sinh x = \frac{e^x - e^{-x}}{2}$

b) $\cosh x = \frac{e^x + e^{-x}}{2}$

c) $\tanh x = \frac{\sinh x}{\cosh x}$

d) $\operatorname{csch} x = \frac{1}{\sinh x}$

e) $\operatorname{sech} x = \frac{1}{\cosh x}$

f) $\coth x = \frac{\cosh x}{\sinh x}$

g) $\operatorname{asinh} x = ln(x + \sqrt{x^2 + 1})$

h) $\operatorname{acosh} x = ln(x + \sqrt{x^2 - 1})$

i) $\operatorname{atanh} x = \frac{ln(\frac{1+x}{1-x})}{2}$

j) $\operatorname{acsch} x = ln(\frac{x \pm \sqrt{1+x^2}}{x})$

k) $\operatorname{asech} x = ln(\frac{x \pm \sqrt{1-x^2}}{x})$

l) $\operatorname{atanh} x = \frac{ln(\frac{x+1}{x-1})}{2}$

9) Stwórz funkcję log (x,y), która obliczy $\log_y x$. Algebraiczna tożsamość dla tego to:

$$\log_y x = \frac{\log_2 x}{\log_2 y}$$

10) Przedział arytmetyczny wymaga wykonania obliczeń, z każdym wynikiem zaokrąglonym w dół a potem powtarzaniem obliczeń z każdym wynikiem zaokrąglonym w górę. Na końcu tych dwóch obliczeń, wiesz, że prawdziwy wynik musi leżeć pomiędzy tymi dwoma obliczonymi wynikami. Bit sterowania zaokrąglaniem w rejestrze sterującym FPU pozwala Ci wybrać tryb zaokrąglania w górę i w dół. Powtórz pytanie sześć stosując przedział arytmetyczny i oblicz dwie granice dla każdego z tych problemów (a-f)
11) Mantysa bitu sterującego precyzją w rejestrze sterującym FPU steruje po prostu kiedy FPU zaokrągla wynik. Wybierając mniejszą precyzję nie poprawiamy wydajności FPU. Dlatego też, każdy nowy , napisany program powinien ustawić te dwa bity na jedynki uzyskując 64 bitową precyzję, kiedy wykonujemy obliczenia. Czy możesz podać jeden powód dlaczego możemy chcieć ustawić precyzję inną niż 64 bity?
12) Przypuśćmy, że masz dwie 64 bitowe wartości X i Y, które chcesz porównać aby sprawdzić czy są równe. Jak wiesz nie powinieneś porównywać ich bezpośrednio aby zobaczyć czy są równe, ale raczej zobaczyć czy są mniejsze niż jakąś mała , oddzielna wartość. Przyjmijmy, że ε, stały błąd, to 1e-300. Dostarcz kod, który załaduj ax zerem jeśli X=Y i załaduje ax jedynką jeśli $X \neq Y$.
13) Powtórz problem 12 zakładając test dla:
    a) $X \leq Y$ b) $X < Y$
    c) $X \geq Y$ d) $X > Y$
    e) $X \neq Y$

14) Jaką instrukcję możemy zastosować aby zobaczyć czy wartość w st(0) jest zdenormalizowana?
15) Zakładając brak przepełnienia lub niedomiaru stosu, jakie jest zazwyczaj zastosowanie dla bitu kodu warunku C_1?
16) Wiele tekstów, kiedy opisuje chip FPU, sugeruje, że możesz użyć FPU dla wykonania arytmetyki całkowitej. Podawanym argumentem jest to, że FPU może wspierać 64 bitowe wielkości całkowite podczas gdy CP spiera tylko 16 lub 32 bitowe wielkości całkowite. Co jest nie tak w tej argumentacji? Dlaczego nie będziemy chcieli zastosować FPU do wykonania arytmetyki całkowitej? Dlaczego FPU nie dostarcza nawet instrukcji całkowitych?
17) Przypuśćmy, że ,masz w pamięci 64 bitową wartość zmienno przecinkową o podwójnej precyzji. Opisz jak możesz pobrać wartość bezwzględną tej zmiennej bez stosowania FPU (tj. przez użycie tylko instrukcji 80x86)
18) Wyjaśnij jak zmienić znak zmiennej z pytania 17
19) Wyjaśnij możliwy problem z następującą sekwencją kodu:

```
stp     mem_64
xor     byte ptr mem_64+7, 80h          ;zmień bitu znaku
```