Damian Koper, 241292
SR-TN-17

Wrocław, 10.04.2019r.

Lab 3:

Cel ćwiczenia

- Zapoznanie się z jednostką zmiennoprzecinkową, rejestry, instrukcje **FPU**
- Analiza standardu *IEEE-754*
  - Zapoznanie się z podstawowymi operacjami arytmetycznymi - dodawanie, odejmowanie, mnożenie, dzielenie i algorytmami, które za nimi stoją
  - Wykorzystanie instrukcji stałoprzecinkowych do działań na formacie pojedynczej precyzji
- Wypisanie liczby w formacie pojedynczej precyzji na ekran decymalnie

IEEE-754 - Single

Liczba w formacie pojedynczej precyzji przechowywana jest w pamięci w następującym formacie:

```
znak           ułamek
x | xxxxxxxx | xxxxxxxxxxxxxxxxxxxxxxx
    wykładnik
```

Ułamek zapisny jest z dodanym obciążeniem, co pozwala zachować ciągłość reprezentacji i ułatwia porównania. Dla formatu single obiążenie to wynosi *+127*. Jeśli wykładnik nie reprezentuje liczb zdenormalizowanych (wartość *0x00*), ułamek zawiera ukrytą jedynkę z przodu.

Dodawanie, odejmowanie

Dodawanie i odejmowanie mogą być zaimplementowane jako jedna operacja, co wynika z zależności *A-B = A+(-B)*. Jako, że implementowane dodawanie jest działaniem przemiennym, warto rozpatrywać zawsze jeden przypadek, gdzie *A* <= *B*. W przypadku kiedy *A* > *B* należy zamienić oba składniki miejscami.

Następnie trzeba wyrównać wykładnik mniejszej liczby. Wiemy, że *A* <= *B*, więc musimy przesunąć bity mantysy B o `exp[A]-exp[B]` w lewo. Na tym etapie, znając utracone bity, możemy zaokrąglić otrzymaną liczbę.

Jeśli znaki obu składników są takie same, należy na mantysach wykonać operację dodawania z zachowaniem znaku wyniku, a jeśli różne, odejmowania razem z ustawieniem znaku wyniku na minus.

Po uzyskaniu wyniku trzeba go znormalizować przesuwając go w lewo lub w prawo jednocześnie zmniejszając lub zwiększając wykładnik wyniku, który początkowo ma wartośc `exp[A]`. Przy wynikach, dla których wartość nie mieści sie w przedziale wartości liczb pojedynczej precyzji należy pamiętać o ustawieniu w odpowiednich przypadkach wartości *0* i *inf*. Wartość *NaN* nie występuje jako wynik w przypadku tych operacji przy poprawnych składnikach.

Mnożenie

Mnożąc liczby ustawiamy znak wyniku wg zależności `sign[C] = sign[A] XOR sign[B]`. Następnie sprawdzamy, czy którykolwiek ze składników ma wartośc *0*. Jeśli tak to zwracamy *0* z odpowiednim znakiem (jeśli chcemy mieć znakowane *0*).

Wynikowy wykładnik otrzymamy poprzez dodanie wartości wykładników składników, pamiętając o odjęciu obiążenia, a następnie korygując go w procesie normalizacji. W celu normalizacji iloczynu mantys, wiedząc, że iloczyn liczb 24 bitowych zawsze da wynik maksymalnie 48 bitowy, możemy sprawdzić bit 48 wyniku tego działania. Jeśli ma on wartośc 1 oznacza to, że wartość jest za duża. Trzeba zwiększyć wykładnik i przesująć mantysę w prawo.

W przypadku, gdy wartość jest zbyt mała, trzeba sprawdzać bit 47 iloczynu, zmniejszać wykładnik i przesuwać mantysę w lewo, aż owy bit nie będzie miał wartości 1 lub wykładnik nie będzie równy `0x01` - w takim wypadku otrzymamy wynik zdenormalizowany. W przypadku wyniku zdenormalizowanego wykładnik reprezentowany jest jako `0x00`.

Zaraz po wykonaniu mnożenia mantys, znając pozostałe bity wyniku, które zostaną utracone, możemy wykonać zaokrąglanie. Tak jak w dodawaniu/odejmowaniu, w przypadku przekroczenia zakresu musimy pamiętać o ustawieniu wartości *+/-inf*.

Dzielenie

Dzielenie odbywa się na podobnej zasadzie co mnożenie.

W tym przypadku ważna jest początkowa walidacja liczb. Rozróżniamy przypadki, którym trzeba ustawić specjalne wartości:

- `A/0 = inf`
- `0/0 = NaN`

Aby uzyskać wykładnik trzeba odjąć od siebie wykładniki dzielnej i dzielnika, a następnie dodać obciążenie. Interpretując mantysę jako liczbę w formacie *Q23*, stosując arytmetykę stałoprzecinkową, przesuwając bity dzielnej o 26 w lewo, jako wynik dzielenia `frac[A]/frac[B]` otrzymamy liczbę w formacie Q26, co daje nam wymagane 23 bity mantysy i trzy dodatkowe bity GRS. Przy czym bit S:

```
S = S | mod(frac[A]/frac[B]) != 0
```

uwzględnia bity reszty.

Normalizacja wyniku przebiega podobnie jak w przypadku mnożenia. Różni się tylko rozmiarem wyniku, a co za tym idzie pozcjami bitów które świadczą o potrzebie normalizacji.

W przypadku przekroczenia zakresu musimy pamiętać o ustawieniu wartości *+/-0*.

FPU

FPU jest jednostką obliczeniową, której zadaniem jest wykonywanie szybkich działań na liczbach w formacie zmiennoprzecinkowym. Posiada swoje rejestry, które tworzą stos `st0-st7`. Rejestry te są współdzielone z

rejestrami MMX, co nie pozwala używać tych dwóch rozszerzeń na raz. Kiedyś FPU był oddzielnym układem scalonym, teraz jednak stanowi jeden układ razem z procesorem.

GDB i liczby zmiennoprzecinkowe

Do wyświetlania liczb zmiennoprzecinkowych w postaci decymalnej służy opcja formatowania `f`.

```
(gdb) x /f &f1
0x804a018:      2.5
```

W zależności od rozmiaru formatu należy dodać modyfikator rozmiaru:

- `w` - 32b (float)
- `g` - 64b (double)

Interesującą obserwacją jest to, że po jednorazowym wpisaniu modyfikatora rozmiaru przy wyświetlaniu, GDB pamięta poprzedni rozmiar. Więc w niektórych sytuacjach może zwracać błędny wynik:

```
(gdb) x /f &f1    <--float
0x804a018:      2.5

(gdb) x /fg &d2   <--double
0x804a024:      2.5

(gdb) x /f &f1
0x804a018:      5.3153507849862534e-315
```

Inną opcją wyświetlania jest wywołanie polecenia `print` z jawnie zrzutowanym wskaźnikiem na odpowiedni typ zmiennej.

```
print *(float*) &f1
$1 = 2.5

print *(double*) &d2
$2 = 2.5
```

Programy

- Kody programów `fAdd`, `fMul`, `fDiv`, `fPrint`, znajdują się pod adresem: https://github.com/damiankoper/OiakLab/tree/master/lab_3

Makra

W programach zastosowane zostały makra, które znacząco zwiększają czytelność kodu i jednocześnie nie tworzą dużego narzutu jak w przypadku funkcji.

Do rejestru `reg` trafia wykładnik liczby `f` z obsługą liczb zdenormalizowanych:

```
.macro expOfTo f reg
  movl \f, \reg
  andl $0x7f800000, \reg
  shr $23, \reg
  cmp $0, \reg # Przypadek liczby zdenormalizowanej
  jne 5f
    inc \reg
  5:
.endm
```

Do rejestru `reg` trafia ułamek liczby `f` z obsługą ukrytej jedynki:

```
.macro fracOfTo f reg
  # Analizuj wykładnik czy dodać ukrytą jedynkę
  movl \f, \reg
  andl $0x7f800000, \reg
  shr $23, \reg

  # Dodaj albo nie
  cmp $0, \reg
  movl \f, \reg
  je 4f
    orl $0x00800000, \reg
  4: # local label
  andl $0x00ffffff, \reg
.endm
```

Macro przy procesie kompilacji wstawiane jest w miejsce wywołania. W przypadku etykiet rodziłoby to konflikty nazw. Dlatego istnieje potrzeba stworzenia lokalnej etykiety. Etykiety lokalne tworzy się nadając im nazwę jako liczbę całkowitą. Podczas skoku litera *f* i *b* po numerze etykiety informuje kompilator czy etykieta znajduje się za, czy przed miejscem skoku.

Operacje arytmetyczne

Wszystkie programy jako zmienne przechowują dwie hardkodowane liczby zmiennoprzecinkowe i na nich wykonują operacje z użyciem instrukcji stałoprzecinkowych według opisanych wyżej procedur. Mimo wielokrotnych owocnych testów poprawności, może zdarzyć się, że otrzymany wynik będzie niepoprawny, szczególnie przy analizie liczb zdenormalizowanych.

Wypisywanie liczby decymalnie

Printf

Jako, że w treści zadania nie ma wymogu o stosowaniu operacji stałoprzecinkowych explicite, do wypisywania liczby na ekran użyta została funkcja `printf` z biblioteki `C`. Przyjmuje ona jednak liczbę w formacie `double`. Zaszła więc potrzeba rozszerzenia liczby do 64b. Wywołanie funkcji wygląda następująco:

```
pushl d1+4
pushl d1
push $formatStr # .string "%E"
call printf
```

Fukcja ta zdejmuje ze stosu 32b + 64b. Jako, że program kompilowany jest w architekturze 32b, to zachodzi potrzeba położenia liczby na stos w dwóch częściach. Przy ręcznym rozszerzaniu liczby z reprezentacji `single` do `double`, trzeba mieć też na uwadze sposób ułożenia bajtów w pamięci - *little endian*.

Wnioski

Przy pisaniu programów warto często rozważyć opcje użycia zamiast funkcji, makra, a wysokopoziomowo funkcji `inline`. Są one szybsze i w przypadku nieskomplikowanych zadań takich jak właśnie pobranie wykładnika czy ułamka liczby, przy wielokrotnym wywołaniu, zwiększają czytelność kodu i stanowią minimalny narzut czasowy.

Do testowania stworzonych programów idealnym byłoby podzielenie ich na funkcje realizujące poszczególne działania i napisanie wielu testów jednostkowych, jednak w tym przypadku stosunek wartości efektów do czasu, który trzeba przeznaczyć na stworzenie tych testów jest mały. Celem ćwiczenia jest bowiem trenowanie podstaw asemblera i wiedzy teoretycznej z zakresu arytmetyki, a nie tworzenie idealnie działającej biblioteki.

Wykonywania obliczeń na liczbach zmiennoprzecinkowych za pomocą instrukcji stałoprzecinkowych można włączyć dodając do kompilatora flagę `-fsoft-float`.

Literatura

1. http://www.rfwireless-world.com/Tutorials/floating-point-tutorial.html
2. http://x86asm.net/articles/fixed-point-arithmetic-and-tricks/
3. Programowa realizacja FPU - https://github.com/lattera/glibc/tree/master/soft-fp
4. Wikibooks x86 Assembly - https://en.wikibooks.org/wiki/X86_Assembly
5. Laboratorium AK –ATT asembler (LINUX) - http://zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Linux-AK2-lab-2018%20May.pdf
6. University of Virginia Computer Science - x86 Assembly Guide http://www.cs.virginia.edu/~evans/cs216/guides/x86.html
7. Prezentacja do wykładu
8. Dokumentacja GDB - https://sourceware.org/gdb/current/onlinedocs/gdb/
9. `gdb help`
10. `man command`
11. Lokalne etykiety - https://stackoverflow.com/questions/39602313/why-cannot-define-same-local-label-in-multiple-functions