Organizacja i architektura komputerów

laboratorium wt TP 07:30

Sprawozdanie - laboratorium II

Autor:
Mateusz Śliwka 241375

Prowadzący:
Mgr inż. Tomasz Serafin

Wrocław 10.04.2020

Spis treści

1 Wstęp

Powierzone zadanie polegało na przygotowaniu programu realizującego działania arytmetyczne takie jak dodawanie, odejmowanie i mnożenia przeprowadzone na liczbach o bardzo dużym rozmiarze (kilkaset bitów). Funkcje te miały zostać zaimplementowane z wykorzystaniem wbudowanych w procesor mechanizmów umożliwiających wykonywanie takich operacji w prostszy sposób, np. obsługę propagacji przeniesienia.

2 Przebieg prac

Z uwagi na to, że pisanie wyżej wymienionego programu było jedną z pierwszych styczności z programowaniem w assemblerze na początku należało zapoznać się ze składnią języka, metodami kompilacji programów oraz ich debuggowania. W kolejnym kroku zająłem się próbą implementacji samego algorytmu dodawania oraz odejmowania na z góry określonych liczbach o jednakowej wielkości tak aby przy użyciu debuggera dobrze zrozumieć sposób jego działania. Finalnie program został rozbudowany o obsługę liczb o różnej długości oraz dodana została implementacja algorytmu mnożenia do którego dobre zrozumienie działania dodawania okazało się niezbędne.

3 Napotkane problemy

Podczas implementacji tego programu głównym problemem okazało się brak oswojenia z językiem. Najwięcej czasu zajęło poznanie składni języka i opisu działania jego funkcji. Oprócz tego, najtrudniejszym etapem było wdrożenie obsługi liczb o różnej długości ze względu na konieczność prawidłowej interpretacji wag pozycji podczas dodawania, co również było konieczne podczas mnożenia.

4 Opis implementacji

Każdy z algorytmów został zaimplementowany jako osobny program. Domyślnie, wszystkie programy przyjmują jako bazowe dwie liczby te zaproponowane przez Prowadzącego, jednak ich działanie zostało przetestowane dla różnych zestawów danych wejściowych.

```
.data

liczba1:
        .long 0x10304008, 0x701100FF, 0x45100020, 0x08570030
liczba1_len = (. - liczba1)/4 #zmienna przechowujaca dlugosc liczby1

liczba2:
        .long 0xF040500C, 0x00220026, 0x321000CB, 0x04520031
liczba2_len = (. - liczba2)/4 #zmienna przechowujaca dlugosc liczby2
```

Rysunek 1: Domyślne dane wejściowe w każdym z programów

4.1 Dodawanie

Sam algorytm dodawanie polega na iterowanie po obu liczbach, pobieraniu kolejnych fragmentów 32bitowych i dodawanie ich do siebie. Każda z liczb ma swój licznik (jest to dekrementowana przy każdym obiegu długość liczby pomniejszona o 1) i przy każdej iteracji pętli należy sprawdzić czy któryś z tych liczników już nie jest równy 0. Taka sytuacja oznacza, że przeiterowaliśmy już całą liczbę i teraz należy albo do drugiej liczby dodawać 0 albo jeżeli obie liczby już przeiterowaliśmy przejść do sprawdzenia nadmiaru. Podczas dodawania ważne jest to, że należy używać funkcji która uwzględnia przeniesienie. Przeniesienie obsługiwane jest przez flagi o których poprawność również należy zadbać i dla bezpieczeństwa w newralgicznych momentach przechowywać je na stosie.

```
dodawaj:
popf #sciagamy flagi ze stosu
movl liczba1(,%edi, 4), %eax #wpisanie do rejestru eax liczby bedacej na pozycji o indeksie %edi w liczba1
movl liczba2(,%esi, 4), %ebx #wpisanie do rejestru eax liczby bedacej na pozycji o indeksie %edi w liczba2
adcl %ebx, %eax #dodanie obu rejestrow z uwzglednieniem przeniesienie
push %eax #odlozenie wyniku na stos
pushf #odlozenie flag na stos
dec %edi #dekrementacja licznika liczby1
dec %esi #dekrementacja licznika liczby2
cmp $0,%esi #porownanie licznika liczba2 do zera po to zeby nie wyjsc za jej zakres
jl dodawaj_do_liczba1 #jezeli jest 0 to znaczy ze liczba2 sie skonczyla i do liczba1 bedziemy dodawac zera
cmp $0,%edi #porownanie licznika liczby1 do zera po to zeby nie wyjsc za jej zakres
jl dodawaj_do_liczba2  #jezeli jest 0 to znaczy ze liczba 1 sie skonczyla i do liczba2 bedziemy dodawac zera
jmp dodawaj #jak nie to znaczy ze nalezy dalej iterowac po fragmentach liczb i je dodawac
```

Rysunek 2: Algorytm dodawania kolejnych fragmentów liczb

Gdy któraś z liczb się skończy przechodzimy do funkcji dodawaj_do_liczba1 lub dodawaj_do_liczba2. Obie funkcje działają analogicznie. Omówiona więc zostanie jedna z nich. Funkcja ta na początku sprawdza czy jednocześnie licznik drugiej liczby się nie skończył. Jeżeli tak, to znaczy, że przeiterowane zostały obie liczby w całości i należy przejść do nadmiaru, który zostanie omówiony w kolejnym akapicie. Jeżeli nie to funkcja ta realizuje dodawanie na wcześniejszych zasadach jednak zamiast dodawać do jednej liczby drugą to dodaje zera, które są rozszerzeniem lewostronnym krótszej liczby.

```
dodawaj_do_liczba1:
cmp $0,%edi
jl sprawdz_nadmiar
movl liczba1(,%edi, 4), %eax #wpisanie do rejestru eax wycietej liczby z liczba1 o dlugosci 4 o okrelsonym indeksie
popf #sciagam ze stosu flagi
adcl $0,%eax #dodaje zero do eax z uwzglednieniem przeniesienia
push %eax #wkladam wynik dodawania na stos
dec %edi #zmniejszam licznik petli
pushf #odkladam na stos flagi
jmp dodawaj_do_liczba1 #jezeli nie to kontynuujemy dodawanie zera do liczby dluzszej
```

Rysunek 3: Dodawanie zer do drugiej liczby

Finalnym etapem działania algorytmu jest sprawdzenie na podstawie flagi CF (flaga przeniesienia) wystąpienia przeniesienia i jego dodania lub bezpośredniego zakończenia programu.

```
sprawdz_nadmiar: #sprawdzanie nadmiaru
popf #sciagamy flagi ze stosu
jc dodaj_nadmiar #skok do dodaj_nadmiar jezeli CF=1
jnc exit #skok do exit jezeli CF=0

dodaj_nadmiar: #zapisyoswanie nadmiaru na najwyzsza pozycje wyniku
push $0x1 #wrzucamy go na stos
```

Rysunek 4: Obsługa nadmiaru

Wynik programu znajduje się na stosie i można odczytać jego zawartość w debuggerze komendą *x/5 $esp* (5 można zamienić na ilość pozycji ze stosu jakie chcemy wyświetlić)

4.2 Odejmowanie

Odejmowanie w ogólnym sposobie implementacji jest dość podobne do dodawania. Metoda pobierania kolejnych fragmentów liczb, działania pętli i warunków ich zakończenia jest taka sama. Algorytmy różnią się przez używaną funkcję odejmowania z uwzględnieniem pożyczki zamiast dodawania oraz przez inny sposób dodawania nadmiaru. Omówione

więc zostaną te elementy algorytmu.

W poniższym wycinku widać zastosowaną funkcję sbbl obsługująca dodawanie z uwzględnieniem pożyczki. Reszta algorytmu działa na zasadzie wcześniej omówionej części dodawania. Oczywiście po osiągnięciu zera przez którykolwiek z liczników również wywoływane są funkcje zajmujące się odejmowanie zer lub odejmowaniem od zera działające analogicznie do tych w dodawaniu (Rysunek 3)

```
odejmuj:
popf #sciagamy flagi ze stosu
movl liczba1(,%edi, 4), %eax #wpisanie do rejestru eax liczby bedacej na pozycji o indeksie %edi w liczba1
movl liczba2(,%esi, 4), %ebx #wpisanie do rejestru eax liczby bedacej na pozycji o indeksie %edi w liczba2
sbbl %ebx, %eax #odjecie obu rejestrow z uwzglednieniem pozyczki (roznica wzgledem dodawaia!)
push %eax #odlozenie wyniku na stos
pushf #odlozenie flag na stos
dec %edi #dekrementacja licznika liczby1
dec %esi #dekrementacja licznika liczby2
cmp $0,%esi #porownanie pierwszego licznika czy moze pobralismy juz cala liczbe
jl odejmuj_od_liczba1 #jezeli przeslimy cala liczba2 to teraz od liczby1 odejmujemy zera
cmp $0,%edi #porownujemy drugi licznij
jl odejmuj_od_liczba2  #jezeli tak to teraz od zero odejmujemy liczba2
jmp odejmuj #jak nie to znaczy ze zadna z liczb sie jeszcze nie skonczyla i odejmujemy je obie od siebie dalej
```

Rysunek 5: Algorytm odejmowania

Rożnica w obsłudze nadmiaru polega właściwie jedynie na tym w jakiej formie został on zapisany.

```
sprawdz_nadmiar: #sprawdzanie nadmiaru
popf #sciagamy flagi ze stosu
check_flag:
jc dodaj_nadmiar #skok do dodaj_nadmiar jezeli CF=1
jnc exit #skok do exit jezeli CF=0

dodaj_nadmiar: #zapisywanie nadmiaru na najwyzsza pozycje wyniku
push $0xFFFFFFFF #wrzucamy go na stos
```

Rysunek 6: Obsługa nadmiaru w odejmowaniu

Wynik programu znajduje się na stosie i można odczytać jego zawartość w debuggerze komendą *x/5 $esp* (5 można zamienić na ilość pozycji ze stosu jakie chcemy wyświetlić)

4.3 Mnożenie

Algorytm mnożenia jest najbardziej rozbudowanym algorytmem spośród tych trzech do zaimplementowania. Dużym ułatwienie przed przystąpieniem do jego implementacji było rozrysowanie schematu jego działania, zaplanowanie kroków algorytmów w postaci kroków a następnie ich zakodowanie. Dodatkowo, w algorytmie mnożenie dodana została

zmienna w której przechowywany będzie finalny wynik. Można go odczytać w debuggerze za pomocą komendy *p/x (long[10]) wynik*

Działanie programu można podzielić na trzy etapy:

- Działanie dużej, zewnętrznej pętli, która pobiera kolejne fragmenty mnożnika
- Działanie małej, wewnętrznej pętli, która pobiera kolejne fragmenty możnej
- Obsługę nadmiaru

Duża pętla ma za zadania przypisywania kolejnych fragmentów mnożnej do rejestru eax, tak aby można było przez nią mnożyć podczas działania małej pętli. Dodatkowo podczas tej pętli inkrementuje się licznik, który wskazuje na obecną pozycje zapisu fragmentu wyniku i tym samym oznacza przesunięcie jakie należy wykonać w małej pętli podczas sumowania kolejnych iloczynów cząstkowych. W pętli zewnętrznej sprawdzany jest także warunek zakończenia działania algorytmu.

```
_start:
clc #czyszczenie flag
pushf #odlozenie czystych flag na stos
push $liczba1_len #odlozenie na stos dlugosci liczby1
push $0 #odlozenie na stos 0, za chwile zdjemiemy je do edi

duza_petla:
pop %edi #zdjecie ze stosu wartosci licznika (bedzie to index pozycji w wyniku) do rejestru edi
inc %edi #zwiekszenie licznika
pop %esi #sciagam esi ze stosu (bedzie licznikiem petli duza_petla)
cmp $0,%esi #sprawdzam czy esi jest juz zerem
jz exit #jak jest to koncze
dec %esi #jak nie to zmniejszam licznik o 1
movl liczba1(,%esi, 4), %eax #wpisuje do eaxa aktualny fragment liczby1 wedlug licznika esi
push %esi #odkladam licznika na stos
push %edi #odkladam edi na stos
mov $liczba2_len, %esi #wpisuje do rejestru esi dlugosc liczby drugiej (bedzie to licznik malej petli)
```

Rysunek 7: Działanie pętli zewnętrznej

Pętla wewnętrzna pobiera kolejne fragmenty mnożnej i wymnaża je z mnożnikiem. Wynik tego działania dodaje do wyniku z odpowiednim przesunięciem jednoznacznym z wagą tego wyniku. Podczas działania małej pętli sprawdzane jest wystąpienie nadmiaru podczas sumowania iloczynów.

```
_start:
clc #czyszczenie flag
pushf #odlozenie czystych flag na stos
push $liczba1_len #odlozenie na stos dlugosci liczby1
push $0 #odlozenie na stos 0, za chwile zdjemiemy je do edi

duza_petla:
pop %edi #zdjecie ze stosu wartosci licznika (bedzie to index pozycji w wyniku) do rejestru edi
inc %edi #zwiekszenie licznika
pop %esi #sciagam esi ze stosu (bedzie licznikiem petli duza_petla)
cmp $0,%esi #sprawdzam czy esi jest juz zerem
jz exit #jak jest to koncze
dec %esi #jak nie to zmniejszam licznik o 1
movl liczba1(,%esi, 4), %eax #wpisuje do eaxa aktualny fragment liczby1 wedlug licznika esi
push %esi #odkladam licznika na stos
push %edi #odkladam edi na stos
mov $liczba2_len, %esi #wpisuje do rejestru esi dlugosc liczby drugiej (bedzie to licznik malej petli)
```

Rysunek 8: Działanie pętli wewnętrznej

Nadmiar ten również dodawany jest na odpowiednią pozycje wyniku a następnie program wraca do funkcji realizującej dalszą część mnożenia.

```
dodaj_nadmiar1:
adcl $0,wynik(,%edi,4) #dodaje nadmiar tzn 0 z uwzglednieniem przeniesienia na kolejna pozycje wyniku
clc #czyszcze flagi
jmp wroc1 #wracam do malej petli
```

Rysunek 9: Obsługa nadmiaru w sumowaniu iloczynów cząstkowych

Wynik programu znajduje się w zmiennej wynik i można odczytać ją odczytać w debuggerze komendą *p/x long[5] wynik* (5 można zamienić na ilość pozycji wyniku jakie chcemy wyświetlić)

5 Opis uruchomienia programu

Program uruchamiany jest przy pomocy makefile, który asembluje i linkuje wszystkie trzy programy na raz. Przy asemblacji użyte zostały takie opcje jak *–32* (informacje o kodzie dla systemu 32bit) oraz *-g* (generowanie flag dla debuggera). Do komendy linkowania dołożona została informacja o wybranym trybie emulacji elf_i386.

```
all: odejmowanie dodawanie mnozenie

odejmowanie: odejmowanie.o
    ld -m elf_i386 -o odejmowanie odejmowanie.o

odejmowanie.o: odejmowanie.s
    as -g --32 -o odejmowanie.o odejmowanie.s

dodawanie: dodawanie.o
    ld -m elf_i386 -o dodawanie dodawanie.o

dodawanie.o: dodawanie.s
    as -g --32 -o dodawanie.o dodawanie.s

mnozenie: mnozenie.o
    ld -m elf_i386 -o mnozenie mnozenie.o

mnozenie.o: mnozenie.s
    as -g --32 -o mnozenie.o mnozenie.s
```

Rysunek 10: Zawartość pliku makefile