08.04.2020

Michał Dunat, 248862

Oddano: 09.04.2020

Ocena:

Operacje dodawania, odejmowania i mnożenia na wielkich liczbach

Sprawozdanie z laboratorium „Organizacja i Architektura Komputerów”

Rok akad. 2019/2020, kierunek INF

Prowadzący: mgr inż. Tomasz Serafin

Spis treści

Przebieg pracy nad programem

Na początku pracy zapoznano się z instrukcjami assemblera dotyczącymi operacji arytmetycznych uwzględniających flagi przeniesienia. W przypadku dodawania i odejmowania pierwsze napisano prostsze wersje programów realizujące te operacje na liczbach o równej długości. Następnie dokonano rozszerzenia programów o algorytmy umożliwiające dodawanie i odejmowanie na liczbach o różnych długościach. Programy te nie różnią się od siebie za bardzo, algorytmy są takie same, zmieniają się wyłącznie poszczególne instrukcje. W przypadku mnożenia (które zaczęto robić później) program pisano od razu z myślą o liczbach o różnej długości. Z debuggera „gdb" korzystano cały czas podczas pracy, dzięki temu narzędziu w prosty sposób można było dowiedzieć się czy program zachowuje się tak jak powinien. Jeżeli kod generował błędne wyniki to „gdb" w znaczącym stopniu ułatwiał jego naprawę (np. poprzez wyświetlanie zawartości rejestrów w poszczególnych momentach pracy programu). Na samym końcu kod uzupełniono o szczegółowe komentarze (które są widoczne tylko w plikach z kodem).

Napotkane problemy

Najpoważniejszym problemem był „samo" czyszczący się stos w przypadku dodawania i odejmowania. Do momentu dojścia do subrutyny odpowiedzialnej za ostatnie przeniesienie/pożyczkę działanie programu było zgodne z przewidywaniami. Po krokowym przejściu przez wspomniane subrutyny zauważono, że jeżeli na stosie próbuje się umieścić zawartość rejestru „rbx" gdy ten jest pusty, następuje całkowite wyczyszczenie stosu. Problem rozwiązano skokiem warunkowym – jeżeli rejestr jest pusty następuje przeskok do końca programu. Następnym problemem, którego nie udało się rozwiązać było mnożenie. Tutaj problem polega na braku poprawnego pomysłu na algorytm realizujący generowanie iloczynów częściowych i dodawanie ich do siebie zgodnie z zasadami mnożenia. Zrealizowano tylko generowanie iloczynów częściowych i umieszczanie ich na stosie.

Kluczowe fragmenty kodu

We wszystkich programach liczby zapisane w pamięci są takie same:

```
.data
num1:
    .long 0x10304008, 0x701100FF, 0x45100020, 0x08570030
num1_len = ( . - num1)/4
num2:
    .long 0xF040500C, 0x00220026, 0x321000CB, 0x04520031
num2_len = ( . - num2)/4
```

Dzieląc długości liczb przez 4 otrzymano rozmiar odpowiadający rozmiarowi pojedynczego słowa, co umożliwi wykonywanie operacji częściowych.

```
mov $num1_len, %esi
mov $num2_len, %edi
dec %esi
dec %edi
```

Ustawienie zawartości rejestrów, żeby mogły służyć jako liczniki iteracji operacji po poszczególnych liczbach. Należy zmniejszyć ich wartość o 1, ponieważ gdy liczba ma rozmiar 4 słów, są potrzebne 4 iteracje, a iteracje liczy się do 0 włącznie. Główna pętla programu:

```
addLoop:
clc
popf
movl num1(,%esi, 4), %eax
movl num2(,%edi, 4), %ebx
adcl %eax, %ebx
push %rbx
pushf
clc

cmp $0, %esi
jz isSecondDone
cmp $0, %edi
jz isFirstDone

dec %esi
dec %edi
jmp addLoop
```

Na początku następuje ustawienie flagi przeniesienia na 0, pobranie flag ze stosu, umieszczenie w rejestrach odpowiednich części liczb dzięki wykorzystaniu trybu adresowania indeksowego. Dodanie zawartości rejestrów z uwzględnieniem przeniesienia, umieszczenie wyniku częściowego i flag na stosie. Następnie sprawdzane jest czy któraś z liczb została już wyczerpana, jeżeli tak to następuje przeskok do subrutyny odpowiedzialnej za sprawdzenie czy pozostała liczba również jest skończona.

Jeżeli do żadnego skoku nie dojdzie następuje dekrementacja ilości iteracji i skok do góry, żeby wykonać następne dodawanie. Subrutyny sprawdzające czy pozostała liczba również się skończyła:

```
isFirstDone:
cmp $0, %esi
jz lastCarry
jmp secondDone

isSecondDone:
cmp $0, %edi
jz lastCarry
jmp firstDone
```

Jeżeli obie liczby się skończyły to sterowanie zostaje przekazane do miejsca gdzie realizowane jest operacja ostatniego przeniesienia. W drugim przypadku następuje skok do subrutyny odpowiedzialnej za dodawanie liczby, która nie jest jeszcze wyczerpana:

```
firstDone:
clc
popf
dec %edi
movl $0, %eax
movl num2(,%edi, 4), %ebx
adcl %eax, %ebx
push %rbx
pushf
jmp isSecondDone

secondDone:
clc
popf
dec %esi
movl num1(,%esi, 4), %eax
movl $0, %ebx
adcl %eax, %ebx
push %rbx
pushf
```

```
jmp isFirstDone
```

To dodawanie wygląda analogicznie jak w głównej pętli z trzema różnicami. Dekrementacja ilości iteracji następuje przed dodawaniem, ponieważ gdy przekazaliśmy sterowanie do „isFirstDone” albo „isSecondDone” dekrementacja nie nastąpiła. Do drugiego rejestru zostaje załadowane zero (nie możemy przepisywać liczby, ponieważ przy pierwszym skoku może wystąpić przeniesienie, które należy uwzględnić). Na końcu od razu następuje skok do odpowiedniej subrutyny sprawdzającej czy należy dalej dodawać. Ostatnie przeniesienie:

```
lastCarry:
clc
popf
mov $0, %eax
mov $0, %ebx
adc %eax, %ebx
cmp $0, %ebx
jz exit
push %rbx
jmp exit
```

Po pobraniu flag ze stosu następuje dodanie 0 do siebie z uwzględnieniem przeniesienia. Jeżeli wynik jest różny od 0 (czyli jest równy 1) to zostaje umieszczony na stosie, jak nie to od razu następuje skok do wyjścia z programu.

Program dla odejmowania działa na identycznych algorytmach z tą różnicą, że tam gdzie jest instrukcja dodawania pojawia się instrukcja odejmowania. Odejmowana jest liczba pierwsza od liczby drugiej. Jeżeli wystąpi pożyczka na ostatniej pozycji to na stosie znajdzie się lewostronne rozszerzenie reprezentujące liczbę ujemną. Poniżej pokazane są tylko te subrutyny, którę się różnią:

```
subLoop:
clc
popf
movl num1(,%esi, 4), %eax
movl num2(,%edi, 4), %ebx
sbb %eax, %ebx
push %rbx
pushf
```

```
clc

cmp $0, %esi
jz isSecondDone
cmp $0, %edi
jz isFirstDone

dec %esi
dec %edi
jmp subLoop


firstDone:
clc
popf
dec %edi
movl $0, %eax
movl num2(,%edi, 4), %ebx
sbb %eax, %ebx
push %rbx
pushf
jmp isSecondDone

secondDone:
clc
popf
dec %esi
movl num1(,%esi, 4), %eax
movl $0, %ebx
sbb %eax, %ebx
push %rbx
pushf
jmp isFirstDone


lastCarry:
clc
popf
```

```
mov $0, %eax
mov $0, %ebx
sbb %eax, %ebx
cmp $0, %ebx
jz exit
push %rbx
jmp exit
```

W przypadku mnożenia początkowe umieszczenie ilości iteracji następuje analogicznie jak w dodawaniu i odejmowaniu. Program dzieli się na dwie pętle: pętle wewnętrzną i zewnętrzną.

```
innerMultiLoop:
movl num1(, %esi, 4), %eax
movl num2(, %edi, 4), %ebx
mull %ebx
push %rax
push %rdx

cmp $0, %esi
jz outerMultiLoop

dec %esi
jmp innerMultiLoop
```

W tej pętli następuje przemnożenie całej liczby pierwszej przez odpowiednią część liczby drugiej. Na początku umieszczane są odpowiednie wartości w rejestrach. Następuje mnożenie i umieszczenie iloczynów częściowych na stosie (po operacji mnożenia młodsza część wyniku umieszczana jest w jednym rejestrze, druga w drugim). Później sprawdzone zostaje czy przemnożono przez całą liczbę pierwszą, jeżeli tak program udaje się do zewnętrznej pętli mnożenia, w przeciwnym razie następuje dekrementacja licznika i następne mnożenie częściowe.

```
outerMultiLoop:
cmp $0, %edi
jz exit
dec %edi
mov $num1_len, %esi
dec %esi
```

```
jmp innerMultiLoop
```

Na początku program sprawdza czy to była ostatnia iteracja po drugiej liczbie, jeżeli tak to występuje skok do końca programu. W przeciwnym razie dekrementuje się ilość iteracji po drugiej liczbie, do rejestru zostaje przepisana ilość iteracji po pierwszej liczbie (ponieważ w pętli wewnętrznej na koniec ustawiono tą wartość na 0) i następuje skok do pętli wewnętrznej.

Opis uruchomienia programu

Program został złożony (assembly) i zlinkowany za pomocą pliku makefile i komendy „make". Do debuggowania, kontrolowania zawartości rejestrów i stosu (wyników) posłużono się debuggerem „gdb". Wyniki działaniu programu można otrzymać następująco: włączyć debugger „gdb", ustawić breakpoint na etykietę „exit", rozpocząć przejście przez program i wyświetlić stos za pomocą komendy „info stack". Zawartość pliku „makefile":

```
all: dodawanieWielkieLiczby odejmowanieWielkieLiczby mnozenieWielkieLiczby

dodawanieWielkieLiczby.o: dodawanieWielkieLiczby.s
	as -g -o dodawanieWielkieLiczby.o dodawanieWielkieLiczby.s

dodawanieWielkieLiczby: dodawanieWielkieLiczby.o
	ld -g -o dodawanieWielkieLiczby dodawanieWielkieLiczby.o

odejmowanieWielkieLiczby.o: odejmowanieWielkieLiczby.s
	as -g -o odejmowanieWielkieLiczby.o odejmowanieWielkieLiczby.s

odejmowanieWielkieLiczby: odejmowanieWielkieLiczby.o
	ld -g -o odejmowanieWielkieLiczby odejmowanieWielkieLiczby.o

mnozenieWielkieLiczby.o: mnozenieWielkieLiczby.s
	as -g -o mnozenieWielkieLiczby.o mnozenieWielkieLiczby.s

mnozenieWielkieLiczby: mnozenieWielkieLiczby.o
	ld -g -o mnozenieWielkieLiczby mnozenieWielkieLiczby.o
```