Michał Surmacki 241130
TN ŚR. $17^{05} - 20^{05}$

Wrocław, dn. 10 kwietnia 2019

Laboratorium 3
OiAK, tydzień 12/13

sprawozdanie z laboratorium przedmiotu „Organizacja i Architektura Komputerów”

Rok akad. 2018/2019, kierunek: INF

PROWADZĄCY:
dr inż. Piotr Patronik

Spis treści

1 Cel ćwiczenia **2**

2 Przebieg ćwiczenia **2**
2.1 Parametry wywołania programu 2
2.2 Konstrukcja programu wyświetlającego parametry wywołania i zawartość środowiskową 3
2.3 Wczytywanie liczb zrealizowane z wykorzystaniem funkcji bibliotecznych . 4
2.4 Konwersja liczb ze standardowego wejścia w bazie 16 5
2.5 Dodawanie i odejmowanie liczb większych niż 64-bitowe 6
2.6 Mnożenie liczb większych niż 64-bitowe 7

3 Podsumowanie i wnioski **8**

Bibliografia **8**

4 Uwagi **8**

1 Cel ćwiczenia

Celem ćwiczenia było wykonanie zadanych programów realizujących konkretne zadania tj:

- drukowanie parametrów wywołania programu„
- proste operacje arytmentyczne na liczbach ”dowolnej precyzji”, w bazie dziesiętnej oraz szestnastkowej,
- zaznajomienie się z wywoływaniem funkcji bibliotecznych.

2 Przebieg ćwiczenia

2.1 Parametry wywołania programu

Aby prawidłowo wykonać pierwsze zadanie, zgodnie ze wskazówką zapoznano się z zawartością stosu w momencie uruchomienia ”funkcji”_`start`. Dokonano tego poprzez użycie narzędzia `gdb`. W celu dodania argumentów wywołania programu, narzędzie `gdb` uruchomiono z opcją `--args` w następujący sposób: `gdb --args ./program h e l l o`. Ustawiono **breakpoint** linijkę pod etykietą `_start`, uruchomiono program pod kontrolą **gdb** poleceniem `run`, następnie wypisano 10 słów znajdujących się na stosie - dokonano tego poleceniem: `x/10xw $esp`. Odpowiednio za `x/` znajdują się: `10` - jest liczbą elementów do wypisania, `x` - jest opcją formatowania wyświetlania, w tym wypadku `x` = szestnastkowo (hexadecymalnie), `w` jest jednostką wielkości `w` = słowo (word), a `$esp` to odwołanie do rejestru przechowującego wskaźnik na górę stosu. Po wykonaniu polecenia uzyskano następujący wynik:

```
(gdb) x/10xw $esp
0xbffff090:     0x00000006      0xbffff293      0xbffff2b7      0xbffff2b9
0xbffff0a0:     0xbffff2bb      0xbffff2bd      0xbffff2bf      0x00000000
0xbffff0b0:     0xbffff2c1      0xbffff2d6
```

Pierwsza na stosie, znajduje się liczba argumentów wywołania (argc), w tym przykładzie 6 - następny na stosie znajduje się adres pod, którym znajdziemy ścieżkę wywołania oraz następne 5 adresów zawierających parametry wywołania (litery - h e l l o). Przystąpiono do sprawdzania co znajduje się pod adresami znajdującymi się dalej na stosie. Pierwszy ardes znajdujący się po liczbie argumentów zawiera ścieżkę wywołania:

```
(gdb) x/s 0xbffff293
0xbffff293:     "/home/michal/Desktop/Lab2/1/program"
```

Ścieżka wywołania jest formatowana do wyświetlenia jako ”string”, powoduje to wywołanie polecenia `x` z opcją `s`. Następnie sprawdzone zostały kolejne adresy znajdujące się na stosie, zgodnie z przypuszczniami pod tymi adresami znajdowały sie argumenty wywołania - litery ’h’ ’e’ ’l’ ’l’ ’o’.

```
(gdb) x/s 0xbffff2b7
0xbffff2b7:     "h"
```

```
(gdb) x/s 0xbffff2b9
0xbffff2b9:     "e"
```

```
(gdb) x/s 0xbffff2bb
0xbffff2bb:     "l"
```

```
(gdb) x/s 0xbffff2bd
0xbffff2bd:     "l"
```

```
(gdb) x/s 0xbffff2bf
0xbffff2bf:     "o"
```

Następnie na stosie znajduje się oddzielające **0**, które oddziela parametry wywołania od zawartości środowiskowej. Wnioskiem z przeprowadzonych testów, jest to, że w momenci uruchomienia funkcji, na stosie znajdują się elementy w następującej kolejności: **argc** - liczba parametrów wywołania, **argv** - parametry wywołania, **null (0)** - oddzielające zero, **zawartość środowiskowa**, **null (0)** - informujące o końcu zawartości środowiskowej.

2.2 Konstrukcja programu wyświetlającego parametry wywołania i zawartość środowiskową

Przystąpiono do pisania programu, wyświetlającego parametry wywołania i zawartość środowiskową. Użyto wcześniej poznanych rozkazów takich jak: `movl` (move long - litera l oznacza 32-bitowe słowo), `pop`, `cmpl`. Podobnie do przykładowego programu w instrukcji laboratoryjnej, użyto nazw symbolicznych, które poprawiają czytelność kodu. Na początku programu, ściągana jest ze stosu wartość argc - znajdująca się na szczycie stosu zaraz po uruchomieniu programu, i umieszczana w rejestrze `edi`, jest to dokonywane poprzez użycie rozkazu `pop %edi`. Rejestr, w którym umieszczono liczbę parametrów wywołania jest licznikiem, który mówi ile razy przeprowadzić operację wyświetlania dla parametrów. Następnie w każdej iteracji pętli, której licznikiem jest `edi`, następuje, dekrementacja wartości znajdującej się w rejestrze `edi`, porównanie jej z zerem, jeśli jest zerem, skok poza główną pętle wyświetlającą parametry wywołania programu, następuje również pobranie ze stosu następnej wartośći, jest to parametr wywołania, umieszczany jest w rejestrze `ecx` - instrukcja `pop %ecx`. Operacja wyświetlania, składa się z kolejnej pętli, w której liczona jest długość łańcucha znaków do wyświetlenia. W każdej iteracji pętli, sprawdzany jest więc warunek zakończenia liczenia długości - napotkanie znaku kończącego łańcuch znaków, `NULL`, bajt zerowy. Do sprawdzenia tego warunku użyto dwóch wcześniej nie używanych wariantów instrukcji:

- `movb` (move byte - litera b oznacza 8-bitowe słowo),
- `cmpb` (b oznacza, że porównywane są 8-bitowe słowa),

oraz rejestru `al`, który jest ośmio-bitowym rejestrem, jest podczęścią rejestru `ax` - rejestr `ax` jest podczęścią rejestru `eax`. Aby wydzielić pojedyncze bajty do porównywania czy nie są znakami końca lini - bajtami zerowymi, używane jest polecenie:

- `movb (%ecx, %edx, 1), %al`

Użyty został tutaj tryb ardesowania zwany indeksowym, jest to rozwiązanie mające podwójną korzyść, ponieważ rejestr określający indeks - `edx` jest zarazem rejestrem przechowującym długość łańcucha znaków do wyświetlenia. W tej metodzie dostępu do danych, pierwszy argument za nawiasem otwierającym - rejestr `ecx`, jest adresem od, którego ma być rozpoczęte zczywytanie pamięci, drugi argument - rejestr `edx`, jest indeksem, ostatni argument - 1, jest mnożnikiem - adres pamięci do zczytania jest wiec obliczany ze wzoru: `Adres ecx + Wartość w edx * Mnożnik 1`. Następnie porównywany jest zczytany bajt, czy jest bajtem zerowym, - instrukcja `cmpb $0, %al`, jeśli warunek okaże się prawdziwy następuje skok poza pętlę liczącą długość łańcucha znaków i wyświetlenie tego łańcucha. Następuje wypisanie na ekran znaku nowej lini, dla poprawienia czytelności. Wyświetlania na ekran są dokonywane wcześniej używanym poleceniem systemowym - **write**.

Zamiast stosować indeks, argc, można zastosować metodę pobierania argumentów ze stosu aż do pobrania pustego elementu (zerowego), tzn. parametry wywołania są oddzielone od zawartości środowiskowej zerem, oraz zawartość środowiskowa jest oddzielona zerem od reszty pamięci - zatem można zdejmować ze stosu argumenty i sprawdzać czy ich zawartość jest zerowa, jeśli jest to przestać drukować argumenty na ekran. W przykładzie, do wyświetlania parametrów wywołania zastosowano metodę wykorzystującą pierwszy zdjęty argument - argc, natomiast w celu wyświetlenia zawartości środowiskowej wykorzystywany jest spobób ze sprawdzaniem czy zdjęty ze stosu element jest zerowy. Zatem po wykonaniu wyświetlania parametrów wywołania - argv, na stosie zostało jeszcze oddzielające zero, więc przed wyświetlaniem zawartości środowiskowej należy zdjąć to zero ze stosu. Po tej operacji, kolejno ze stosu są zdejmowane elementy, sprawdzany jest warunek końca wyświetlania - argument zerowy, i tak jak przy wyświetlaniu łańcuchu znaków parametrów wywołania, liczona jest długość zczytanego elementu, następnie wyświetlenie łańcucha znaków.

Program realizyjący to zadagnienie to program nazwany - `program`.

2.3 Wczytywanie liczb zrealizowane z wykorzystaniem funkcji bibliotecznych

Program wczytujący liczby i odpowiadającym nimi, za pomocą funkcji bibliotecznych, wywołania systemowego i wczytaniu za pomocą parametru wywołania napisano jako drugi z kolei. Kiedy używamy bibliotek współdzielonych, program jest linkowany-dynamicznie. Oznacza to tyle, że nie wszystkie instrukcje potrzebne do uruchomienia programu są zawarte w środku tego programu, ale w osobnych bibliotekach. Kiedy uruchamiamy program, plik biblioteki jest ładowany pierwszy, jest linkowany do programu, wyszukiwane jest miejsce w, którym znajduje się biblioteka, następnie w kodzie źródłowym programu wyszukiwane są miejsca, w których występują odwołania do biblioteki, kiedy znajdą się takie, program zostaje uzupełniony w kod który odpowiada wywoływanym funkcjom. Aby poprawnie zlinkować program, należy użyć polecenia:

```
ld -dynamic-linker /lib/ld-linux.so.2 -o program5 program5.o -lc
```

Opcja `-dynamic-linker /lib/ld-linux.so.2` pozwala na linkowanie dynamiczne bibliotek do programu, przed utworzeniem pliku wykonywalnego system załaduje `/lib/ld-linux.so.2` w celu poprawnego załadowania zewnętrznych bibliotek, programem który to wykonuje jest **dynamic linker**. Opcja `-lc`, mówi o tym żeby zlinkować program z biblioteką `c - libc.so` znajdującej się w systemie GNU/Linux. Linker dodaje na początku `c` przedrostek `lib`, oraz przytostek/rozszerzenie `.so`.

Napisany program wyświetla na dwa i sczytuje na trzy różne sposoby. Dokonuje tego wywołaniem systemowym, wywołaniem funkcji bibliotecznej, oraz wczytaniem jako parametry wywołania. Pierwszy sposób polega na wywołaniu poleceń systemowych **write** oraz **read**, drugi, który jest istotą przeprowadzanego ćwiczenia - wywoływania funkcji bibliotecznych, polega na wywołaniu funkcji **printf** oraz **scanf**. Zgodnie z dokumentacją aby poprawnie wywołać funkcje biblioteczne należy wcześniej przygotować stos, tzn. włożyć na stos odpowiednie elementy - w odwrotnej kolejności do argumentów w deklaracji funkcji. Funkcje printf i scanf mogą przyjmować wiele argumentów, ich wygląd i formatowanie muszą jednak odpowiadać odpowiednim standardom. Aby poprawnie wczytać dane od użytkownika, na stos zostały "wrzucone" następujące elementy `$value2` - adres

miejsca w, które mają zostać wczytane dane, `$format2 - "%4s"` - łańcuch formatujący, wczytywanie łańcucha znaków długości cztery. Następnie wywołano funkcję **scanf** - `call scanf`. Ostatnim sposobem wczytywania danych jest wczytywanie za pomocą parametrów wywołania. Jak opisano w akapicie 2.1 oraz 2.2 paramenty wywołania znajdują się w odpowiednim porządku na stosie, do operacji wyświetlania zostały więc ściągane, następnie porównywane czy nie są elementem zerowym, następnie wyświetlane. W terminalu w którym został uruchomiony program, wyswietlona zostaje informacja o wczytanych danych jako parametry wywołania programu, gdy nie ma parametrów wywołania, program wyświetla tylko linijkę informacyjną. Przy wczytywaniu za pomocą wywołania systemowego można wprowadzić trzy znakowy input, jest to zapisane "na twardo" w kodzie programu, uzywając funkcji **scanf** można wpisać liczbę cztero cyfrową, program odpowie "Input: x! Input: x!" gdzie w miejscu x znajdą się wpisane numery. W tym programie również użyto nazw symbolicznych dla czytelności kodu, zadeklarowano również "zmienne - etykiety, które są zamieniane na adresy w procesie assemblowania", których użyto do poinformowania użytkownika jakim sposobem będą zczytywane liczby.

2.4 Konwersja liczb ze standardowego wejścia w bazie 16

Przed przystąpieniem do pisania programu wykonujacego operacje matematyczne na liczbach "większej precyzji", napisano program/funkcję, którą wykorzystano w późniejszym pisaniu programów realizujących działania matematyczne. Program ten, pobiera ze stosu są dwa elementy: **adres** gdzie przechowywany jest łańcuch znaków oraz następny element znajdujący się na stosie - **długość tego łańcucha**. Następnie odbywa się konwersja z łańcucha znaków na liczbę zapisaną w bazie 16. Jeśli rozmiar wczytanego łańcucha jest większy niż 8 bajtów, funkcja "rozrywa" łańcuch na pomniejsze elementy, które umieszcza pokolei na stosie. Po wykonaniu wszystkich czynności związanych z parsowaniem łańcucha na liczbę hex. funkcja umieszcza jeszcze na stosie 0. Aby "powrócić" z funkcji w odpowiednie miejsce, elementy które zostały wrzucone na stos w toku działania funkcji zostają zdjęte, tak aby adres powrotu znajdował się pierwszy na stosie. Tak więc w programach wykonujących operacje arytmetyczne, sposób wywoływania funkcji parsującej jest następujący: należy "wrzucić" na stos rozmiar łańcucha znaków, następnie należy "wrzucić" na stos ten łańcuch, i wywołać funkcję. Aby móc wywołać funkcję należy dodać w kodzie źródłowym linijkę `.extern stox` - stox to nazwa napisanej wcześniej funkcji (string to hex), funkcję wywołuje się poleceniem `call stox`. Należy pamiętać również o odpowiednim zlinkowaniu programu, kod źródłowy funkcji musi zostać zassemblowany i następnie poprawnie zlinkowany. Aby ułatwić pracę przy pisaniu funkcji i programu ją używającego, napisano odpowiedni plik **Makefile**, który pozwolił na zautomatyzowanie pracy.

2.5 Dodawanie i odejmowanie liczb większych niż 64-bitowe

Aby poprawnie dodać liczby, których wielkość przekracza rozmiar jednego rejestru należy robić to partiami. Najniższe 32 bity można dodać poleceniem **addl**, jednak uwzględniając że wynikiem tego działania może być liczba, która zajmuje więcej niż 32 bity, następne dodawanie musi się odbyć z uwzględnieniem flagi nadmiaru, (carry flag). Rozkaz **addc** uwzględnia w obliczeniach tę flagę. Rozkazy addc powtarzamy dopóty dopóki dysponujemy częściami liczb które chcemy dodać, na koniec wyświetlane zostają na ekranie części które są wynikami poszczególnych działań. Napisany program rozpoczyna się od pobrania inputu od użytkownika, dokonano tego wywołaniem funkcji **scanf**. Po wczytaniu od użytkownika liczb, przekonwertowano je w taki sposób aby były przechowywane w pamięci jako liczby, a nie jako łańcuch znaków. Dokonano tego napisaną funkcją **stox**. Funkcję zadeklarowano dyrektywą `.extern stox`, cały program zlinkowano w odpowiedni sposób - dodając - stox.o. Funkcja **stox** konwertuje znaki ascii na cyfry oraz odpowiednie litery na liczby - dotyczy to liter a, b, c, d, e, f (również wielkich). Wprowadzone przez użytkownika liczby są "dowolnej precyzji"to znaczy, ażeby dobrze przekonwertować wprowadzone liczby trzeba je podzielić na części. Funkcja **stox**, konwertuje, dzieli i wrzuca na stos odpowiednie elementy. Wrzuca je w następującym porządku:

- najmłodsza część wprowadzonej liczby,
- (...)
- najstarsza część wprowadzonej liczby
- liczba części składających się na pierwszą liczbę
- 0

Po dwóch wykonaniach funkcji **stox**, które konwertują dwie wpisane przez użytkownika liczby, w programie głównym - `program5.s`, zostaje wywołana następna funkcja **add**. Funkcję **add** również zadeklarowano dyrektywą **`.extern add`** i odpowiednio zlinkowano program - add.o. Funkcja **add** odpowiada za dodanie części, które znajdują się na stosie po wywołaniach funkcji stox. W ciele funkcji dochodzi do dodania w odpowiedni sposób, kolejnych części liczb. Najmłodsze części są dodawane poleceniem **addl**, starsze natomiast muszą być potraktowane uważniej, ponieważ istnieje możliwość że dodawanie najmłodszych części wygenerowało przeniesienie. Na stos zostały wszucone flagi - polecenie **pushf**. Zostało ono wywołane zaraz po operacji dodawania najmłodszych elementów, ponieważ po dodawaniu tych elementów występują polecenie takie jak - **cmpl**, które te flagi modyfikują. Następnie tuż przed dodawaniem starszych częśći, wywołano polecenie **popf**, które przywróciło statusy flag z momentu zaraz po dodawaniu najmłodszych pozycji. Samo dodawnie odbyło się za pośrednictwem polecenia - **adcl** - dodawanie z uwzględnieniem flagi przeniesienia, bit 0 rejestru **eflags**. Następne starsze pozycje są dodawane analogicznie, to jest z taką samą uwagą na flagi, oraz z wykorzystaniem rozkazu **adcl**. Na koniec została przeprowadzona operacja sprawdzenie czy nie wystąpił nadmiar (przeniesienie ponad 128 bit), w tym celu rejestr flag został wrzucony na stos i następnie przeniesiony do rejestru **eax** - poleceniem `pop %eax`. Flaga przeniesienia znajduje się na pozycji 0 w rejestrze flag. Dokonano operacji logicznej AND - rozkazem `and %eax, %ebx`, w rejestrze `ebx` umieszczono liczbę 1, która odpowiada binarnie pozycji flagi nadmiaru. Wyniki operacji logicznej **AND** znajdujący się w rejestrze **eax**, jest sprawdzany i w razie wystąpienia nadmiaru (przeniesienia na nieistniejącą pozycję), zostaje wyświetlona informacja o tym.

Dostęp do zasobów znajdujących się na stosie był dokonywany przez użycie wcześniej wrzuconego na stos i ustawionego rejestru **%ebp**, przez użycie indeksowanego trybu adresowania.

```
michal@michal-VirtualBox:~/Desktop/Lab2/2$ ./program2
11111111ffffffffFFFFFFFFffffffff
1
111111120000000000000000000000000
```

Napisany program można w łatwy sposób zmodyfikować, osiągając substraktor. Aby przeprowadzić działanie odejmowania liczb o "dowolnej precycji", należy zamienić rozkazy **addl** na **subl** oraz rozkazy **adcl** na rozkaz **sbbl** - (substraction with borrow) odejmowanie z pożyczką. Również w przypadku odejmowania rozkaz **sbbl** korzysta z flagi przeniesienia (pożyczki).

2.6 Mnożenie liczb większych niż 64-bitowe

Aby przeprowadzić mnożenie liczb rozszerzonej precyzji należy mieć opanowane dodawanie liczb rozszerzonej precyzji, ponieważ ostatnim etapem mnożenia liczb, jest dodawanie otrzymanych składników. Rozpoczynamy od mnożenia części najmłodszych liczb, następnie najmłodsza część mnożnika jest mnożona ze starszymi częściami mnożnej. Następnie następuje przejście do starszej części mnożnika, ta część jest mnożona, tak jak wcześniejsza część, przez wszystkie części mnożnej poczynając od najmłodszej. W identyczny sposób przeprowadza się operacje, aż do wymnożenia wszystkich części przez siebie.

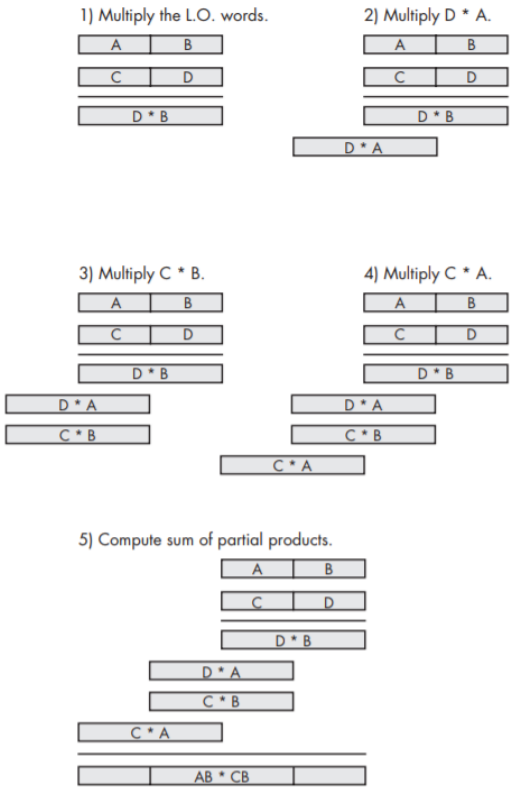

Figure 8-2: Extended-precision multiplication

Rysunek 1: Strona wycięta z [4], przedstawiająca proces mnożenia liczb rozszerzonej precyzji

Nie udało się napisać programu realizującego mnożenie, dlatego przybliżam tylko koncept na jakim, by to zostało wykonane.

3 Podsumowanie i wnioski

Wykonując ćwiczenia zadane na laboratoriach, zostały opanowane koncepty takie jak: pisanie funkcji i wywoływanie ich w programie, wywoływanie funkcji systemowych, konwertowanie liczb wpisanych ze standardowego wejścia. Nie udało się napisać programów wykonujących wszystkie operacje arytmetyczne, jednak proces wykonywanie tych operacji został opisany. Wiele pracy zostało włożone w napisanie sumatora liczb "dowolnej precyzji"i wyświetlanie parametrów wywołania i zawartości środowiskowej. Napisane programy były testowane za pomocą gdb, w ostatecznym rachunku tego nie widać ile czasu zostało poświęcone na debugowanie programów. Aby zautomatyzować pracę - proces assemblowania i linkowania, stworzono dla każdego programu plik Makefile.

Literatura

[1] Jonathan Bartlett, *Programming from the Ground Up*, 2004.

[2] IA-32 Intel® Architecture Software Developer's Manual Volume 1: Basic Architecture,

[3] IA-32 Intel® Architecture Software Developer's Manual Volume 2: Instruction Set Reference,

[4] Randall Hyde, *THE ART OF ASSEMBLY LANGUAGE, 2ND EDITION* str. 477–529, 2010

4 Uwagi

Listingi kodów nie zostały zamieszczone, natomiast podaję link do repozytorium na githubie, gdzie znajdować się będą kody wykonanych programów. Programy zostały przetestowane i działają poprawnie, zgodnie z początkowymi założeniami.

https://github.com/MichalSurmacki/OiAK-Lab/tree/master/Lab2.

- Program wypisujący parametry wywołania i zawartość środowiskową znajduje się w folderze "1".
- Sumator liczb rozszerzonej precyzji znajduje się w folderze "2".
- W folderzee "4" znajduje się, program, który konwertuje liczbę w bazie 16 na liczbę w bazie 10 (tylko małe litery).
- W folderze "5" znajduje się program wczytujący input od użytkownika, wywołaniem systemowym, wywołaniem funkcji **scanf** oraz za pomocą parametrów wywołania.