Damian Koper, 241292
SR-TN-17
Wrocław, 10.04.2019r.

Lab 2: Stos, arytmetyka wielkich liczb, funkcje bibioteczne

Cel ćwiczenia

- Zapoznanie się z zawartością stosu w momencie uruchomienia programu.
- Wypisanie na ekran argumentów programu i zmiennych środowiskowych
- Zapoznanie się z podstawowymi instrukcjami arytmetycznymi
- Arytmetyka wielkich liczb - podstawowe działania
- Konwersja liczb z różnych reprezentacji do docelowej reprezentacji binarnej

Śledzenie stosu po początkowym wywołaniu

Ilość argumentów `argc` znajduje się na szczycie stosu:

```
(gdb) x /d $esp
```

Wskaźniki na kolejne argumenty programu znajdują się w dalszej kolejności na stosie w ilości `argc`:

```
(gdb) x /s *(char **)($esp+4) # <- pierwszy argument to ścieżka wywołania
(gdb) x /s *(char **)($esp+8)
...
```

Następnie na stosie znajduje się wartość *0*, po której znajdują się wskaźniki na kolejne stringi ze zmiennymi środowiskowymi. Nie podano z góry ilości tych zmiennych, dlatego na końcu tego ciągu występuje również wartość *0*

```
(gdb) x /s *(char **)($esp+12) # <- +12 to najmniejszy możliwy offset dla
zmiennych środowiskowych przy wywołaniu bez argumentów. [argc, arg1, 0,
1stENV, ...]
```

Arytmetyka wielkich liczb

Operacje na liczbach większych niż największa możliwa długość słowa maszynowego wymagają zastosowania specjalnych algorytmów, które działają tylko na część tej liczby. Sprowadza się to do tego, że liczby te traktowane są jako liczby o rozmiarze będącym wielokrotnością największego słowa.

Abstrakcja

Wszystkie przypadki, poza dzieleniem, można rozpatrywać jako operacje na liczbach (tak jak na kartce) interpretowanych jako liczby w systemie o podstawie 2^n gdzie *n* jest dozwoloną długością słowa.

Dodawanie i odejmowanie

Dodawanie i odejmowanie są najprostrzymi operacjami do zaimplementowania dla wielkich liczb. Są również podstawą, na której opierają się bardziej skomplikowane operacje np. mnożenie i dzielenie. Dodając (odejmując) dwie liczby trzeba wykonać na kolejnych ich częściach podstawowe operacje dodawania (odejmowania), które uwzględniają przeniesienie lub pożyczkę wejściową. Jeśli podczas tej operacji wystąpi *overflow*, zostanie ustawiona flaga *CF* - carry flag, która określa wartość przeniesienia (pożyczki) dla następnej operacji. Instrukcje, które wykorzystują flagę *CF* w tym przypadku to `adc` i `sbb`.

Mnożenie

Mnożąc 2 liczby wykonujemy mnożenie każdego członu z każdym, następnie dodając iloczyn do wyniku z odpowiednim przesunięciem. Przy mnożeniu 2 liczb o rozmiarze 4 słów wykonamy 16 mnożeń i w optymalnej wersji 7 dodawań. Dla implementacji wykorzystającej dwie zagnieżdżone pętle wykonamy 16 mnożeń i tyle samo dodawań.

Dzielenie

Wykonując dzielenie trzeba przeskalować dzielną i dzielnik wykonując rotacje poszczególnych słów z przeniesieniem w lewo. Wykorzystać do tego trzeba rozkaz `rclx` - gdzie `x` jest rozmiarem słowa. Wynik uzyskamy stosując algorytm dzielenia nieodtwarzającego.

Użycie funkcji bibliotecznych

W celu użycia funkcji języka C używając linkera `ld` trzeba przekazać argumenty mówiące o dołączeniu do pliku wykonywalnego odpowiednich bibliotek:

```
ld -melf_i386 -dynamic-linker /lib/i386-linux-gnu/ld-linux.so.2 $^  -o $@ -
lc
```

Argument `-lc` (*--library=c*) dołącza do programu bibliotekę C. Argument `-dynamic-linker /lib/i386-linux-gnu/ld-linux.so.2` ustawia odpowiedni linker dla wersji 32bit, która kompilowana jest na maszynie 64bit.

`printf`, `scanf`

Użycie funkcji `printf` i `scanf` sprowadza sie do położenia na stos kolejnych argumentów zaczynając od najstarszego, a następnie wykonanie rozkazu `call` do wybranej funkcji. Po zakończeniu wykonywania funkcji, w celu zachowania integralności stosu, należy zdjąć ze stosu dane wcześniej argumenty.

Programy

Kody źródłowe wszystkich programów: https://github.com/damiankoper/OiakLab/tree/master/lab_2

Kody źródłowe właściwych funkcji: https://github.com/damiankoper/OiakLab/tree/master/lab_2/utils

W napisanych przeze mnie programach, w celu uproszczenia, wszystkie powtarzalne fragmenty kodu obudowane są w funkcje, które znajdują się w osobnych plikach i udostępniają globalnie swoją etykietę. Funkcje są odpowiedzialne za zdejmowanie dostarczonych argumentów ze stosu - rozkaz `ret $x` zdejmujący `x` bajtów ze stosu przy powrocie.

`showEnv`

Program `showEnv` wyświetla wedle założeń argumenty wywołania programu i zmienne środowiskowe. Wykorzystałem tutaj swoją funkcję `printStr`, która liczy długość stringa i wywołuje przerwanie systemowe w celu wypisania wartości.

Operacje arytmetyczne

- Za format liczby przyjąłem konfigurowalną wielokrotność 32 bitów.
- Liczby są przekazywane do funkcji za pomocą wskaźnika na pierwsze słowo.
- Funkcje modyfikują dostarczoną zawartość
- We wszystkich opisanych niżej programach, jeśli dane nie zostaną dostarczone poprzez argumenty, program czeka na ich wpisanie

`addDemo`

- Format danych - dwie liczby dziesiętne o rozmiarze max 128 bitów

Kluczowym fragmentem jest pętla, która iteruje po kolejnych członach liczby:

```
loopAddc:
        movl -4(%ebx, %ecx, 4), %eax
        adc -4(%edx, %ecx, 4), %eax
        movl %eax, -4(%edx, %ecx, 4)
        loop loopAddc
```

W tym wypadku nie jest możliwa operacja z dwoma odniesieniami do pamięci - używam pomocniczo rejestru `%eax`. W tej pętli rejestr `%ecx` ma zawsze wartość o jeden większą niż wymagany index, stąd wartość `-4` przy adresowaniu. Wynik dodawania wyświetlany jest szesnastkowo.

`subDemo`

- Format danych - dwie liczby szesnastkowe w notacji `0x123...`, oraz liczba określająca ich długość w słowach np. `0x0000000000000002 0x0000000000000008 0x00000002` -> 2 - 8

Program ten wykorzystuje funkcje biblioteki C - `printf` i `scanf` do interakcji z użytkownikiem. Dzięki podobieństwu w konwencji wywołania dla wszystkich funkcji kod programu jest podobny do kodu w `addDemo.s`. Zmianie uległy tylko nazwy ww. funkcji oraz format danych wejściowych - szesnastkowy. Aby

zobaczyć wynik wypisany przez funkcję `printf` trzeba również zakończyć program z użyciem funkcji `exit`, a nie poprzez przerwanie systemowe jak dotychczas - daje to szansę na wypisanie bufora na ekran.

W pętli iterującej po liczbach zmienił się tylko właściwy rozkaz:

```
loopSubb:
        movl -4(%ebx, %ecx, 4), %eax
        sbbl -4(%edx, %ecx, 4), %eax
        movl %eax, -4(%ebx, %ecx, 4)
        loop loopSubb
```

`mulDemo`

- Format danych - dwie liczby szesnastkowe w notacji, oraz liczba określająca ich długość w słowach np. `0x00000000000000000000000000000002 0x00000000000000000000000000000008 0x00000004`

Ilość słów, z których składa się liczba w tym programie została ustalona z góry na 4. Wiąże się to z brakiem możliwości alokacji pamięci na bufor, którego rozmiar jest z góry ustalony. Dynamiczna alokacja pamięci nie jest przedmiotem tego laboratorium. Program zwraca wyższą i niższą część wyniku na miejscu mnożnika i mnożnej.

`divDemo`

- Format danych - dwie liczby szesnastkowe w notacji, oraz liczba określająca ich długość w słowach np. `0x00000000000000000000000000000002 0x00000000000000000000000000000008 0x00000004`

Dzielenie wykorzystuje przesunięcie bitowe - rozkaz `rcl` w celu skalowania dzielnika i dzielnej, oraz do wstawiania kolejnych bitów wyniku, powstałego poprzez działanie algorytmu dzielenia nieodtwarzającego. Program pozwala na dzielenie tylko liczb dodatnich i nie został przetestowany do końca. Nie generuje również reszty. Zaletą okazało się wydzielenie operacji dodawania i odejmowania jako osobne funkcje. Znacznie uprościło to kod i jego późniejszą analizę. Ograniczeniem, tak jak w przypadku mnożenia, jest również rozmiar buffora, który z góry został na 4.

Fragment kodu - dzielenie nieodtwarzające, dodanie albo odjęcie dzielnej:

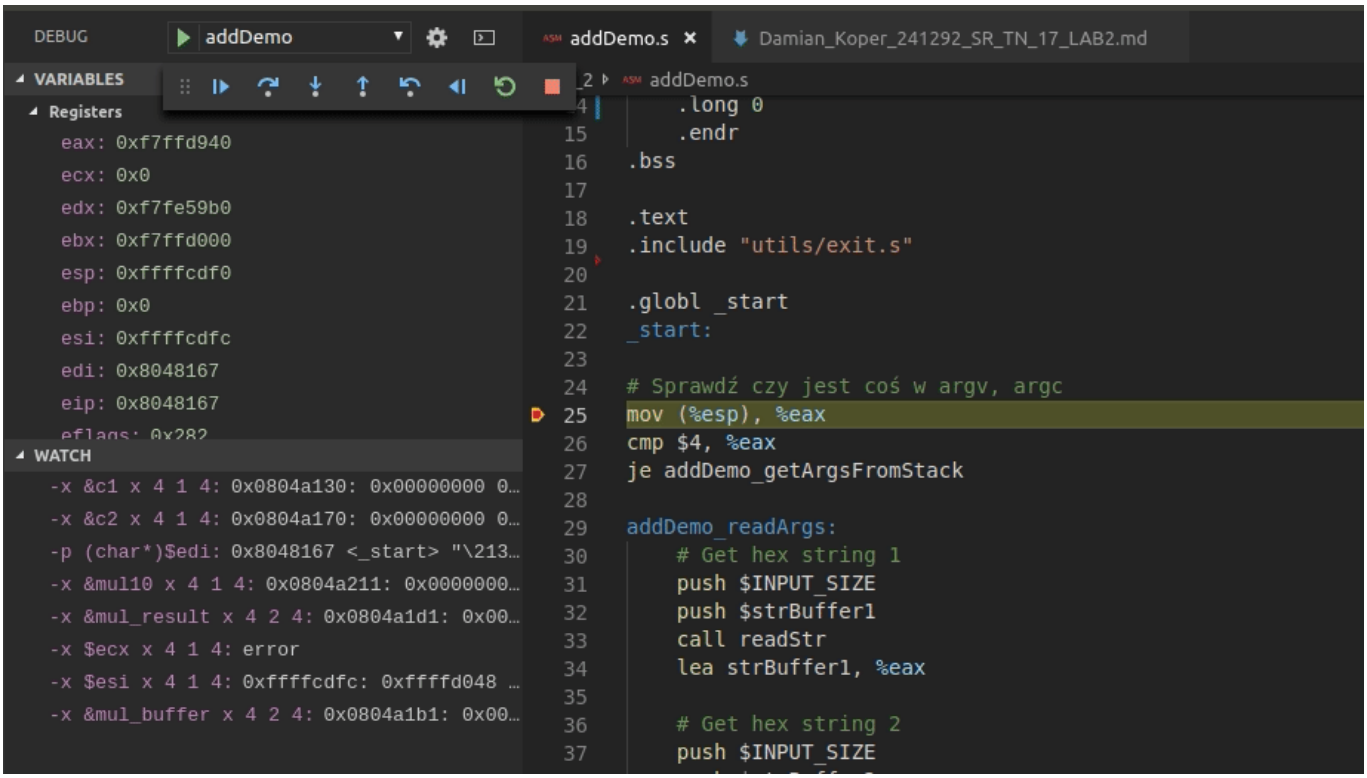
```
addR:
        pushl $4
        pushl %ebx
        pushl %edx
        call addFn
        jmp addsubEnd

subR:
        pushl $4
        pushl %edx
        pushl %ebx
        call subFn
```

```
        pushl $4
        pushl $result
        pushl $add1
        call addFn
addsubEnd:
```

Widać tutaj, że z powodzeniem używane są funkcje dodawania i odejmowania z poprzednich programów.

Debugger - GDB / Machine Interface

Niezbyt praktyczne i wizualnie nieatrakcyjne GBD, nawet uruchomione w trybie *Text User Interface*, skłoniło mnie to napisania rozszerzenia do Visual Studio Code, które po rozpoczęciu sesji debugowania uruchamia w tle GDB w trybie interpretera *mi* - Machine Interface. Komunikacja VsCode z GDB odbywa się poprzez odpowiednie parsowanie strumienia wyjściowego GDB i wysyłanie na wejście poleceń, które specyfikuje dokumentacja GDB/MI.



VsCode Debug Protocol pozwala również na ustawienie *watchy*, wyświetlanie stosu wywołań, oraz podgląd zmiennych, zamiast których wyświetlam wszystkie rejestry. Interakcja z GDB za pomocą interfejsu konsolowego jest możliwa poprzez *Debug Console*. Z możliwości napisanego rozszerzenia korzystałem w procesie debugowania powyżej przedstawionych programów, sam debugger jednak posiada niedoskonałości i ma duże pole do rozwoju, który, w ramach rosnących potrzeb, będę uskuteczniał.

Wnioski

Nawet w kodzie tak niskopoziomowym podział kodu na zwięzłe bloki wykonujące określone funkcje (zasada pojednczej odpowiedzialności) jest możliwy i przynosi korzyści - poprawia czytelność kodu, ułatwia modyfikacje i zmniejsza ryzyko błędów, ponieważ zamknięte bloki mogą być testowane oddzielnie.

Literatura

1. Wikibooks x86 Assembly - https://en.wikibooks.org/wiki/X86_Assembly
2. Laboratorium AK –ATT asembler (LINUX) - http://zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Linux-AK2-lab-2018%20May.pdf
3. University of Virginia Computer Science - x86 Assembly Guide http://www.cs.virginia.edu/~evans/cs216/guides/x86.html
4. Prezentacja do wykładu
5. Intel Manual - https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf
6. Dokumentacja GDB - https://sourceware.org/gdb/current/onlinedocs/gdb/
7. Dokumentacja GDB/MI - https://sourceware.org/gdb/current/onlinedocs/gdb/GDB_002fMI.html
8. `gdb help`
9. `ld --help`
10. Dokumentacja ld - ftp://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_mono/ld.html
11. `man command`