Politechnika Wrocławska

Data wykonania ćwiczenia: 06.03.2018
Termin zajęć: $10^{:15}$-13^{15} | TN
Sala: C-3 / 013

SPRAWOZDANIE

Architektura Komputerów 2

Prowadzący: mgr. Aleksandra Postawka

Paweł Szynal 226026

Laboratorium 0
(wprowadzanie)

Wrocław 2018

Spis treści

1. Etapy laboratorium

- Poznanie działania podstawowych komend w systemie Linux/64
- Podstawy składni programów dla gnu asembler
- Sposoby kompilowania i konsolidacji programu
- Utworzenie pliku makefile oraz napisanie programu „Hello_world” w języku assembler AT&T (64-bit)

2. Przebieg laboratorium

2.1. Wprowadzenie do Linux-a

Na początku zajęć dowiedzieliśmy się w jaki sposób korzystać z terminala w środowisku Linux. Wykorzystane i poznane komendy podczas zajęć:

mkdir

MkDir - to skrót od Make Directory.Tworzy katalog. Działa zarówno w środowisku Unix\Linux jak i Windows.

cd ‘katalog’

Jest to polecenie powłoki, służące do przemieszczenia się pomiędzy katalogami w systemie operacyjnym.

cd

Powraca do poprzedniego katalogu.

touch

Tworzy nowy plik, jeśli komenda jest wywoływana z argumentem.

pwd

Z języka angielskiego ‘print working directory’ - wypisz katalog roboczy.

ls

Wypisuje w postaci listy zawartość katalogu.

mv

Przenosi pliki lub katalogi do innego miejsca niż aktualne.

rm

Skrót od ang. remove – usuń. Jest używana do usuwania plików oraz katalogów.

cp Służy głównie do kopiowania plików.

2.2. Podstawy architektury programów w assemblerze AT&T

2.2.1. Rejestry

Rejestr procesora: zespół układów elektronicznych, mogących przechowywać informacje.

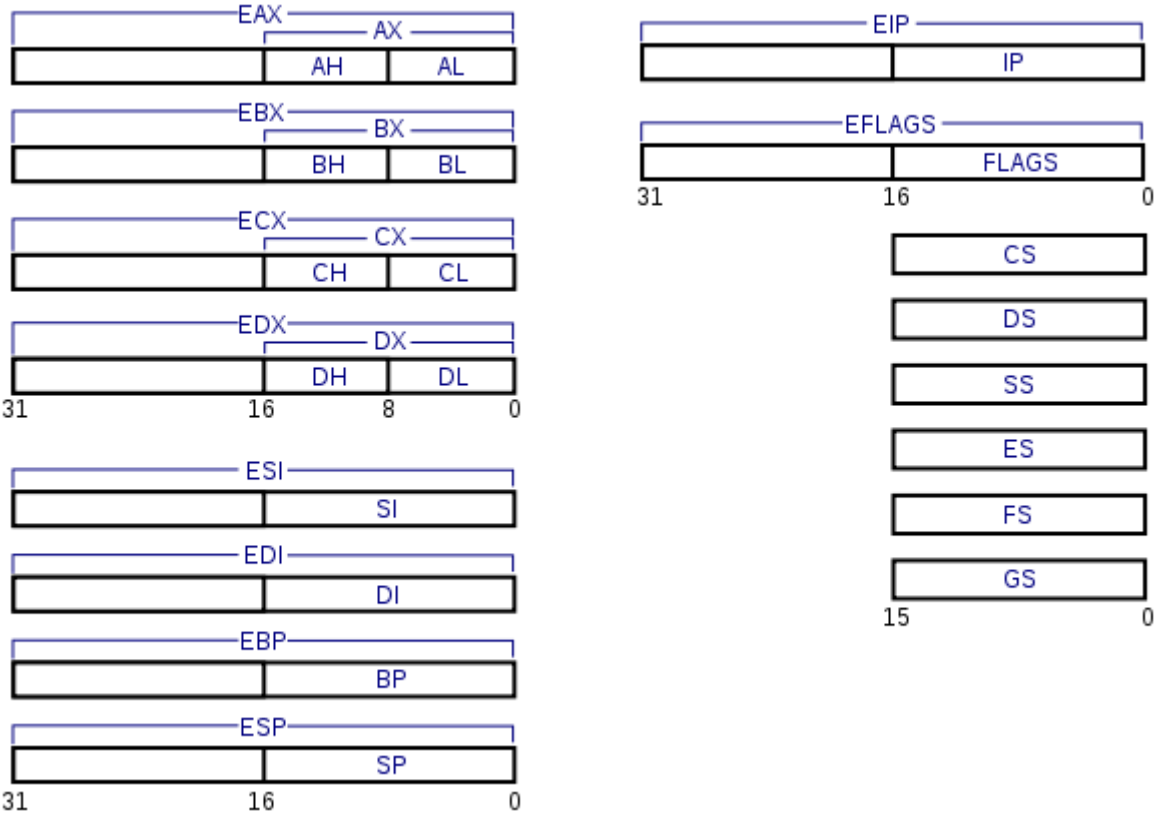

Figure 1 (32 Bity)

2.2.1.1. Rejestry ogólnego użytku

Akumulator: **RAX**

Rejestr ten najczęściej służy do wykonywania działać matematycznych

Rejestr bazowy: **RBX**

Rejestr ten jest używany np. do przechowywania danych.

Licznik: **RCX**

Tego rejestry używamy np. do określenia ilości powtórzeń pętli.

Rejestr Danych: **RDX**

Np. w tym rejestrze przechowujemy adresy różnych zmiennych.

R8B (R8B … R15B)

8 rejestrów 8-bitowych

R8W (R8W … R15W)

8 rejestrów 16-bitowych

R832 (R8B … R15B)

8 rejestrów 32-bitowych

2.2.1.2. Rejestry indeksowe

Indeks źródłowy: **RSI**

Indeks docelowy: **RDI**

2.2.1.3. Rejestry wskaźnikowe

Wskaźnik stosu: **RSP**

Wskaźnik bazowy: **RBP**

Wskaźnik instrukcji: **RIP**

2.3. Składnia asemblera AT&T (GAS)

GAS jest projektem GNU którego celem jest tworzenie darmowego oprogramowania (GNU S.O / Linux). GAS jest wieloplatformowy i jest uruchamiany dla wielu różnych architektur.

Program napisany w asemblerze możemy podzielić na 3 sekcje.

The Data Section: **.data**

Służy do deklarowania stałych.

The Basic Service Set (Podstawowy zestaw usług): **.bss**

Sekcja deklarowania zmiennych i importowania uług.

The Text Section: **.text**

Sekcja do przechowywania kodu. Musi rozpoczynać się o deklaracji:

```
.text
.gklob  _start
_start:
```

Podstawowy format instrukcji w asemblerze (AT&T)

```
[Mnemonic] [źródło], [Cel]
```

2.3.1. Sufiksy w asemblerze AT&T (GAS)

b - liczba 8 bitowa (bajt)
w - liczba 16 bitowa (ward)
l - liczba 32 bitowa (long)
q - liczba 64 bitowa (quad)
t – liczba 80 bitowa (ten)

Warto zaznaczyć, że jeśli przyrostek nie jest określony i nie ma operandów pamięci dla instrukcji, GAS podaje rozmiar argumentu operacji pobierając wielkość argumentu rejestru docelowego.

2.4. Plik makefile

Plik sterujący makefile to wygodniejszy sposób kompilowania pliku. Steruje on procesem kompilacji. Jeżeli mamy do skompilowania więcej niż jeden plik źródłowy pomaga to zaoszczędzić nam czas. Sposób tworzenia pliku „makefile” w środowisku 64-bitowym:

```
hello:  hello.o                 #linkowanie
    ld -o hello hello.o         # Reguła konsolidacji

hello.o:    hello.s             #Asemlacja
    as -o hello.o hello.s       # -o wskazuje plik wynikowy
```

Bardzo ważne jest zachowanie dokładnie takiej budowy, gdyż ominiecie nawet jednej tabulacji spowoduje złe wykonanie się kompilacji.

2.5. VIM

Vim (skrót od ang. vi improved) – edytor tekstu vi, napisany przez Brama Moolenaara z którego korzystaliśmy podczas zajęć.

Isntalacja pakietu vim:

```
sudo apt-get update
sudo apt-get install vim
```

Tryby pracy

Vim posiada kilka trybów pracy. NORMAL, INSERT i COMMAND-LINE/EX są emenetarnymi trybami . W celu wejścia w tryb INSERT należy wcisnąć dowolna literę. Aby z niego wyjść i przejść do trybu NORMAL, należy nacisnąć klawisz ‘Esc’. Przejście w tryb Ex zawsze zaczyna się od dwukropka (poprzedzonego wciśnięciem klawisza Esc w przypadku trybu wyjściowego innego niż NORMAL).

vim plik.o

edycja pliku ‘plik.o’
:q - Zamyka plik pod warunkiem, że nie był zmodyfikowany.
:q! - Zamyka plik bez zapisywania zmian.
:w - Zapisuje zmiany.
:wq - Zapisuje zmiany i zamyka plik.

2.6. gedit

Jest to [1]linuksowy edytor tekstu oparty na bibliotece GTK+. Należy do projektu GNOME (dlatego nazwa zaczyna się od litery „g”). Zapewnia kolorowanie składni. I to właśnie w nim pisałem programy.

Instalacja pakietu i dodatkowych pluginów gedit, które ułatwiły implementację kodu:

```
sudo add-apt-repository ppa:gedit-bc-dev-plugins/releases
sudo apt-get update
sudo apt-get install gedit-plugins
sudo apt-get install gedit-projects-plugin
```

[1] Od grudnia 2008 roku jest również dostępny dla użytkowników systemów Microsoft Windows

2.7. Program „Hello World”

```
# ____________________________________________________________________________
#
#   AUTOR: Paweł Szynal nr albumu 226026
# ____________________________________________________________________________
.data       # inicjalizacja segmanetu danych (Segment danych jest przeznaczony do odczytu i
zapisu)

# Funkcje stadtardowe (syscall -tryb 64 bitowy dla linux-asm)
SYSREAD = 0         # nr funkcji wejscia - odczyt
SYSWRITE = 1        # nr funkcji wyjscia - zapis
SYSEXIT = 60        # nr funkcji zakonczenia i zwrotu sterowania do SO
STDOUT = 1          # nr wyjscia stadardowego (ekran tekstowy)
STDIN = 0           # nr wejscia standardowego (klawiatura)
EXIT_SUCCESS = 0

buf: .ascii "Hello, world!\n"   # kod ascii zapisany do segmendu buffora
buf_len = .-buf             # dlugosc bufora do wyswietlania

.text                       # Sekcja kodu programu
.globl _start               # punkt wejscia programu

_start:                     # start programu

movq $SYSWRITE, %rax        # przeniesienie wartosci z SYSWRITE do rejestru rax
movq $STDOUT, %rdi          # systemowe stdout
movq $buf, %rsi             # wrzucenie Wrzucenie napisu "Hello word" z buf do rejestru rsi
movq $buf_len, %rdx         # dlugosc wyswietlanego łancucha znaków
syscall

movq $SYSEXIT, %rax         # wykonanie funkcji EXIT
movq $EXIT_SUCCESS, %rdi    # Wrzucenie do rejestru kodu wyjscia z programu
syscall                     # zwraca kod bledu w %rdi
```

2.8. Zadanie domowe

```
# ____________________________________________________________________________
#
#   AUTOR: Paweł Szynal nr albumu 226026
# ____________________________________________________________________________
#       Program wczytaj-wypisz i zamien wielkie litery na małe, a małe na wielkie
#               Segment danych
.data

STDIN = 0
STDOUT = 1
SYSWRITE = 1
SYSREAD = 0
SYSEXIT = 60
EXIT_SUCCESS = 0
BUFLEN = 512

#   Dyrektywa as .comm
#   może zostać połączony ze zdefiniowanym lub wspólnym symbolem
#   o tej samej nazwie w innym pliku obiektowym.
# ____________________________________________________________________________
#                Basic Service Set
.bss
.comm textin, 512
.comm textout, 512
# ____________________________________________________________________________
#               Sekcj tekstowa (kod prorgamu)
.text
.globl _start

    # .global (.glob) sprawia, że symbol jest widoczny dla ld. Jeśli zdefiniujemy symbol
    # w programie częściowym, jego wartość jest udostępniana innym programom
      #częściowym,
      # które są z nim powiązane.

_start:

movq $SYSREAD, %rax # kopiujemy wartość SYSREAD do akumulatora.
movq $STDIN, %rdi   # kopiujemy wartość STDIN do indeksu źródłowego.
movq $textin, %rsi  # kopiujemy wartość textin do indeksu docelowego.
movq $BUFLEN, %rdx  # kopiujemy wartość BUFLEN do rejestru ranych.
syscall             # Invoke the operating system.
```

```
dec %rax        # zmniejszamy wartość o 1 (pozbywamy sie '\n')
movq $0, %rdi   # licznik
```

```
zamien_wielkosc_liter:      # rejestr EBX dzieli sie na E, bx, z kolei bx dzieli sie na bh i bl
movb textin(, %rdi, 1), %bh     # bh = rdi * 1 + textin
movb $0x20, %bl                 # bl =  32
xor %bh, %bl                    # bx xor bl
movb %bl, textout(,%rdi, 1)     # rdi * 1 + textout = bl
inc %rdi                        # zwiększa wartość rdi o 1
cmp %rax, %rdi                  # porównuje rejestry ustawiając odpowiednio flagi;
jl zamien_wielkosc_liter

movb $'\n', textout(, %rdi, 1)

movq $SYSWRITE, %rax    # kopiujemy wartość SYSWRITE do akumulatora.
movq $STDOUT, %rdi      # kopiujemy wartość STDOUT do indeksu źródłowego.
movq $textout, %rsi     # kopiujemy wartość textin do indeksu docelowego.
movq $BUFLEN, %rdx      # kopiujemy wartość BUFLEN do rejestru danych.
syscall

movq $SYSEXIT, %rax
movq $EXIT_SUCCESS, %rdi
syscall
```

2.9. Debugger gdb

Instalacja gdb:

```
$ sudo apt-get update
$ sudo apt-get install gdb
```

Gdb to debugger na poziomie źródłowym GNU, który jest standardem W systemach linux. Może być używany zarówno w programach napisanych w językach C i C ++ oraz dla programów napisanych w asemblerze.

Aby uzyskać szczegółowe informacje na temat korzystania z gdb, należy wpisać komendę:

```
info -f /pkgs/gnu/info/gdb
```

```
h[elp] [keyword]
```

Wyświetla informacje pomocy.

```
r[un] [args]
```

Rozpoczna wykonywanie programu.

```
b[reak] [address]
```

Ustawa punkt przerwania pod podanym adresem (lub pod bieżącym adresem, jeśli nie ma go wcale). Adresy można nadać symbolicznie (Etykieta) lub numerycznie
(np. * 0x10a38).

```
c[ontinue]
```

Kontynuowanie po zatrzymaniu w punkcie przerwania.

```
i[nfo] b[reak]
```

Wyświetl numerowaną listę wszystkich aktualnie ustawionych punktów krytycznych.

```
d[elete] b[reakpoints] number
```

Usuń określony numer punktu przerwania.

```
i[nfo] r[egisters] register
```

Sposób drukowania wartości rejestru

```
s[tep]i
```

Wykonaj pojedynczą instrukcję, a następnie wróć do interpretera wiersza poleceń.

```
Q[iot]
```

Wychodzi z gdb.

2.10. Wnioski końcowe

Podczas ćwiczeń nie było większych problemów z poprawnym stworzeniem i uruchomień programu Hello Word.

2.11. Źródła

[1] The GNU Assembler

https://sourceware.org/binutils/docs/as/

[2] Intel 64 and IA-32 ArchitecturesSoftware Developer'sManual

https://www.intel.com/content/reference-manual

[3] gEdit 3

https://help.gnome.org/users/gedit/stable/

[4] How to Install GDB?

http://www.gdbtutorial.com/tutorial/how-install-gdb

[5] Debugging Assembly Code with gdb

http://www.akira.ruc.dk/~keld/teaching/CAN_e14/Readings/gdb.pdf