Programowanie Mikroprocesorów

Materiały do nauki GNU Asemblera na x86

v0.9

Janusz 'Ivellios' Kamieński

Dokument zawierający opracowanie kodów w języku GNU Asembler, z informacjami wprowadzającymi kolejne instrukcje oraz konstrukcje języka.

WSTĘP DO OPRACOWANIA

Opracowanie to powstało jako notatki to przedmiotu "Programowanie mikroprocesorów" na drugim roku studiów z Informatyki Stosowanej na wydziale EAIiE AGH Kraków.

Kontakt z autorem: ivellios [kropa] mirimafea [na] gmail [kropa] com

Jeżeli uważasz, że mój trud włożony w tę pracę przydał Ci się do nauki, zaliczenia jakiegoś kolokwium, zdania egazminu, mile widziane będzie odwdzięczenie się piwkiem (namiary znajdziesz powyżej).

OGRANICZONA ODPOWIEDZIALNOŚĆ

Autor tej pracy nie odpowiada za poprawność treści w niej zawartych. Nie odpowiada również za wszelkie szkody spowodowane stosowaniem porad zamieszczonych w niniejszej pracy. Praca ta ma na celu pomoc w nauce, jednak z całą pewnością nie może być wolna od wad, toteż stosowanie jej z pewną dozą ostrożności nie zaszkodzi.

Kody programów zawarte w niniejszym opracowaniu oparte są na kodach wykorzystywanych na zajęciach z Programowania Mikroprocesorów, na drugim roku Informatyki Stosowanej Wydziału EAIiE Akademii Górniczo-Hutniczej w Krakowie, prowadzonych przez dr inż. Zbigniewa Bublińskiego.

LABORATORIA

LABORATORIUM 0 - WSTĘP

O czym my w ogóle mówimy?

Mówiąc o asemblerze nie sposób nie zrobić tekstu wprowadzającego do tematyki, który przy okazji zaprezentuje najprostszy program. Laboratorium 0. ma na celu przedstawienie bardzo podstawowych informacji na temat programowania w asemblerze procesorów x86.

Stałe

Na potrzeby naszego kursu, na początek pokażemy, że dyrektywa niżej zaprezentowana tworzy stałe. Przyjrzyjmy się następującemu kodowi:

```
  .equ  write,0x04
```

Jak widać już zapis dyrektywy odbywa się poprzez zapis ".equ" następnie podajemy nazwę stałej oraz jej wartość po przecinku. Wartości te ustawiamy już na początku programu. Jak zaraz się okaże dyrektywy (czyli różnego rodzaju polecenia zaczynające się od kropki) będą bardzo ważne.

Dane

Tak jak .equ używamy do ustawiania zmiennych tak teraz wykorzystamy dyrektywę .data do określenia zmiennych. Używa się jej jednak inaczej niż .equ. Dyrektywę .data wpisujemy na początku bloku zmiennych, po czym następuje cała seria zmiennych, które będziemy oznaczać za pomocą etykiet (dobrze znanych nam także z innych języków programowania). Zobaczmy jak deklarować zmienne:

```
  .data # dyrektywa ustalajaca ze od tego miejsca mamy dane

starttxt:
  .ascii  "Start\n"
endtxt:
  .ascii  "Koniec\n"

.equ  startlen, endtxt - starttxt
.equ  endlen, . - endtxt
```

Jak widzimy, w Asemblerze, tak samo jak w C, etykiety tworzymy poprzez wpisanie nazwy i dwukropka, po których podajemy instrukcje. W naszym konkretnym przykładzie tworzymy zmienną starttxt, która przechowuje ciąg znaków ASCII z napisem "Start" i znakiem nowej linii. Dyrektywa .ascii każe zarezerwować pamięć na cały ciąg znaków podany po spacji (tabulatorze). Jak widać poniżej w bloku .data możemy umieszczać także stałe. Jest to oczywiste, ponieważ niektóre stałe wymagają znajomości pewnych wartości, których na początku pliku nie mamy. W tym wypadku mam na myśli adresy przypisane etykietom, by obliczyć rozmiary tekstów. Na uwagę zasługuje też tajemniczy zapis:

```
[...] . - endtxt [...]
```

Oznacza on nie mniej nie więcej jak: weź adres pamięci, w którym się znajdujesz i odejmij od niego adres przypisany do etykiety starttxt. W ten sposób możemy policzyć rozmiar tekstu "Koniec".

Etykiety przypisane do pamięci?

Tak właśnie jest. Gdy mówimy o zmiennych, to ich nazwy reprezentują pewne adresy w pamięci, gdzie znajdują się dane. Jeżeli popatrzymy na nasz program, to składa się on z szeregu instrukcji. Każda instrukcja jest przechowywana w pamięci pod jakimś adresem. Kolejne instrukcje to kolejne adresy w pamięci (tak na prawdę mnemonik rozkazu zajmuje jakiś obszar, zapis np. rejestru zajmuje inny obszar). Etykiety pozwalają nam oznaczyć pewne adresy w tym stosie programu, dzięki czemu możemy do niego np. skoczyć lub odwołać się, by pobrać wartości. Zatem pisząc, że:

```
.equ startlen, endtxt - starttxt
```

mamy na myśli dokładnie: weź adres przypisany etykiecie endtxt i odejmij od niego adres przypisany do etykiety starttxt, następnie przypisz go stałej startlen. W ten sposób możemy otrzymać rozmiar tekstu, ponieważ każda litera takiego tekstu zajmuje jeden bajt na stosie programu, a ciąg znaków traktujemy tutaj jako nieprzerwaną tablicę bajtów.

Dyrektywa text

Dyrektywa ta informuje kompilator, że w tym miejscu zaczynają się rozkazy programu, które mają zostać przetworzone po kolei (choć nie koniecznie jak się okaże). Odtąd ruszamy z regularnym programem. A więc do dzieła!

Dyrektywa global

Dyrektywa .global służy do ujawnienia pewnej etykiety kompilatorowi oraz linkerowi. Jest ona używana (a przynajmniej będzie przez nas w tym kursie) do wskazywania, która etykieta (czyli gdzie jest) wskazuje na miejsce gdzie powinien się zacząć proces pracy programu. Jako argument tej dyrektywy, jak nietrudno zgadnąć jest nazwa etykiety, wskazującej na początek instrukcji.

Transport danych w rejestrach

Korzystając z mnemonika MOV (w uogólnieniu, ponieważ MOVL przenosi liczby 4-bajtowe, MOVW 2B, MOVB 1B) możemy kazać przenieść pewną liczbę z jednego miejsca na drugie. Możemy przenosić liczby z rejestru do rejetru, z pamięci do rejestru i odwrotnie, ale nie możemy przenosić wartości pomiędzy dwoma zmiennymi (pamięć na pamięć). W każdym procesie operacji w procesorze musi pośredniczyć w jakiś sposób rejestr.

Wypisywanie na ekranie

Pora w końcu na wypisanie jakiegoś tekstu na ekranie, jak to bywa w każdym pierwszym programie. Do tego celu wykorzystujemy funkcję write systemu, która ma adres 0x04 w

pamięci funkcji systemu. Trzeba pamiętać, że używając funkcji systemowych trzeba podać im argumenty korzystając z rejestrów. W rejestrze %eax umieszczamy adres funkcji. W naszym przypadku do %ebx przekażemy uchwyt do miejsca, gdzie chcemy pisać, w naszym wypadku będzie do liczba 1, która oznacza standardowe wyjście (czyli ekran). Rejest %ecx przyjmie adres pamięci, pod którym znajduje się tekst do wypisania (dokładnie jego początek, bo tekst jest tablicą). Rejestr %edx będzie przechowywał długość tekstu. Aby wywołać funkcję systemową musimy wywołać przerwanie procesora na adres systemu (u nas będzie to przerwanie o numerze 0x80). Przerwania wywołujemy poleceniem INT (od Interrupt - przerwanie). Oto zatem jak będzie wyglądała nasza lista instrukcji wypisania tekstu "starttxt":

```
MOVL  $0x04,%eax # funkcja wypisania
MOVL  $1,%ebx # standardowe wyjście
MOVL  $starttxt,%ecx # adres tekstu (tablicy znaków)
MOVL  $startlen,%edx # ilość znaków
INT  $0x80 # przerwanie
NOP # odczekanie jednobajtowej operacji nic nie robiącej
```

A co to te dolary?

Znaku dolar ($) używamy na trzy sposoby. Pierwszy z nich pozwala nam przekazywać wprost liczby w postaci literalnej. Instrukcja:

```
MOVL $1, %ebx
```

po prostu wprowadza liczbę 1 do rejestru. Nie można napisać samego "1", gdyż oznaczałoby to adres w pamięci o numerze 1. Jeżeli użyjemy znaku dolar razem z etykietą, będzie to oznaczało tyle co: weź adres, który przypisany jest do etykiety. Natomiast użycie etykiety bez dolara oznacza pobranie wartości znajdującej się pod zmienną oznaczoną przez etykietę. W takim razie pisząc:

```
MOVL $starttxt,%ecx
MOVL starttxt,%edx
```

W pierwszym przypadku przekażemy do rejestru %ecx adres, pod którym zaczyna się nasz tekst. W drugim przypadku wpiszemy do %edx numer kodu ASCII pierwszej litery tekstu starttxt. Jak widzimy w przykładzie funkcji wypisującej tekst na ekranie, musimy podać adres do początku tablicy znaków, aby funkcja mogła ją wypisać przechodząc po każdym elemencie tyle razy ile przekażemy w rejestrze %edx. (Nie ma pewności..)

Zróbmy w końcu program!

Nie stoi nam już nic na przeszkodzie by wykonać program, który wypisze na ekranie napisy "Start" oraz "Koniec". Warto też zauważyć, że aby zakończyć program trzeba użyć przerwania do systemu, przesyłając funkcję z adresu 0x01, która odpowiada funkcji exit. Tak program po przerwaniu zostanie zamknięty przez system. Jeżeli czujesz się na siłach, spróbuj sam napisać ten program. Poniżej rozwiązanie z komentarzami pomocniczymi.

```
#-----------------------------------------------------------------
# Program LAB_0.S - Asemblery Laboratorium IS II rok
# Program prezentujący wypisywanie na ekranie napis Start oraz Koniec
```

```
#------------------------------------------------------------------

# dyrektywa .equ nadaje etykiety stale (odpowiednik #define z C)

  .equ  kernel,0x80  #Adres przerwania dla Kernela w Linux
  .equ  write,0x04  #Numer funkcji pisania na ekranie
  .equ  exit,0x01  #Kod wyjścia z programu

  .data # dyrektywa ustalajaca ze od tego miejsca mamy dane

starttxt:      #first message
  .ascii  "Start\n" # umieszcza w pamieci tekst podany w cudzysłowiach
endtxt:        #second message
  .ascii  "Koniec, a w zasadzie dopiero poczatek, bo pierwszy lab\n"

.equ  startlen, endtxt - starttxt
    # ustalamy długość tekstu "start" do tego celu używamy etykiety
.equ  endlen, . - endtxt
    # długość tekstu "Koniec", wykorzystujemy kropkę, która zwraca
    # nam adres pamięci w tym miejscu programu

  .text # kod programu w postaci tekstu

  # .global okresla ze symbol ma byc globalny (ważne dla kompilatora i
linkera!! )
  .global _start

_start:
  # ustalamy jaka funkcja bedzie uzywana, w tym wypadku pisanie
  MOVL  $write,%eax
  # ustalamy gdzie chcemy pisać, w naszym wypadku na standardowe wyjście
  MOVL  $1,%ebx
  # Możliwe tryby to:
  # 0 stdin
  # 1 stdout
  # 2 stderr

  # przekazanie tekstu, który chcemy wypisać
  MOVL  $starttxt,%ecx

  # przekazujemy dlugość wypisywanego ciągu znaków
  MOVL  $startlen,%edx

  # przerwanie do systemu przerwania
  INT  $kernel
  NOP


  # to samo dla drugiego napisu
  MOVL  $write,%eax
  MOVL  $1,%ebx
  MOVL  $endtxt,%ecx
  MOVL  $endlen,%edx
  INT  $kernel
  NOP
theend:
  MOVL  $exit,%eax # przekazanie funkcji końca programu do systemu
  INT  $kernel # przerwanie i koniec programu
```

LABORATORIUM 1 - DODAJEMY LICZBY

Wstęp

Program, który dzisiaj wykonamy nie będzie się różnił zasadniczo od poprzedniego. Postaramy się utworzyć zmienne liczbowe oraz wykonać na nich podstawowe operacje. Ponieważ jednak wypisywanie liczb na ekranie nie jest takie proste (na ekranie wypisujemy znaki ASCII, które nie odpowiadają wprost liczbom), na razie będą prowadzone tylko operacje. Kolejne laboratorium pokaże jak wypisywać liczby w postaci szesnastkowej na ekranie.

Dyrektywy dla liczb

Gdy w poprzednim laboratorium opisywana była dyrektywa .ascii, powiedziane było, że tworzy tablicę znaków. Tym razem poznamy trzy kolejne dyrektywy pozwalające tworzyć zmienne liczbowe. Będą one także umieszczane w bloku .data i wykorzystamy je do późniejszych operacji na nich
Pierwszą dyrektywą zmiennej liczbowej jest .byte . Dyrektywa ta tworzy w pamięci zmienną o rozmiarze jednego bajta. Jako argument dyrektywy, po spacji wprowadzamy liczbę, która ma być w zmiennej wpisana na początku. Należy pamiętać, że liczba taka mieści się w rejestrze przynajmniej al, ah, ale także w większych (wtedy reszta nie jest zmieniana rejestru).
Druga dyrektywa to .word. Tworzy ona zmienną 2-bajtową. Używamy jej tak samo. Mieści się ona w rejestrach ax i większych.
Dyrektywa .long tworzy zmienną 4-bajtową, która zmieści się jedynie w rejestrach eax , ebx. Wartości początkowe ustawiamy tak samo.

Dodawanie liczb i odejmowanie

Aby dodać do siebie liczby korzystamy z instrukcji o mnemoniku ADD, jako argumenty podając rejestr pierwszej liczby, oraz rejestr liczby drugiej, do którego to także zapiszemy wynikową sumę. Z odejmowaniem sytuacja ma się tak samo: używamy mnemonika SUB i podajemy liczbę, którą chcemy odjąć, a następnie od której odejmujemy. Wynik zostanie zapisany w tej drugiej (czyli tej, od której odejmowaliśmy).

Program do wykonania

Jesteśmy już gotowi by wykonać zadany program. Możemy do tego celu wykorzystać program poprzedni, odpowiednio go modyfikując lub napisać zupełnie od nowa. W programie mamy wypisać napis "Start" oraz "Koniec", a pomiędzy tymi napisami mamy przeprowadzić operację: dodania liczby drugiej do pierwszej oraz odjęcia liczby trzeciej od pierwszej, a następnie zapisania tego wyniku do zmiennej result. Jako ciekawostkę można tutaj podać, że jako drugi argument (%ebx) systemowej funkcji exit (0x01) można podać kod zakończenia programu. W przykładzie poniżej liczba obliczona w programie zostanie zwrócona jako kod wykonania programu do systemu.

```
#-----------------------------------------------------------------
# Program LAB_1.S - Asemblery Laboratorium IS II rok
# Zadaniem programu jest przeprowadzenie podstawowych operacji
# na liczbach całkowitych.
#-----------------------------------------------------------------
```

```
  .equ  kernel,0x80  #Numer przerwania do systemu
  .equ  write,0x04  #write data to file function
  .equ  exit,0x01  #exit program function
  .equ  stdout,0x01  #handle to stdout

  .data

starttxt:      #first message
  .ascii  "Start\n"
endtxt:        #second message
  .ascii  "Koniec\n"
arg1:        #liczba 8-bitowa
  .byte    1
arg2:        #liczba 16-bitowa
  .word    2
arg3:        #liczba typu long
  .long    3
result:        #miejsce gdzie przechowamy wyniki
  .long    0

.equ  startlen, endtxt - starttxt
.equ  endlen, arg1 - endtxt

  .text
  .global _start

_start:
  nop # nic nie robienie w tym momencie
  MOVL  $write,%eax
  MOVL  $stdout,%ebx
  MOVL  $starttxt,%ecx
  MOVL  $startlen,%edx
  INT  $kernel
  NOP
  MOVL  arg1,%eax # wprowadzenie liczb do rejestrow
  MOVL  arg2,%ebx
  MOVL  arg3,%ecx
  ADD  %ebx,%eax # dodanie do siebie liczb
  SUB  %ecx,%eax # odjęcie od siebie liczb
  MOVL  %eax,result # zapisanie wyniku do zmiennej result
  NOP # odczekamy

  MOVL  $write,%eax # wypisujemy jak w lab0 pliczek
  MOVL  $stdout,%ebx
  MOVL  $endtxt,%ecx
  MOVL  $endlen,%edx
  INT  $kernel
  NOP

theend:
  MOVL  $exit,%eax
  MOVL  result, %ebx # system przekaze wynik jako wartosc zakonczenia
programu
  INT  $kernel
```

LABORATORIUM 2 - LICZYMY I TO WIDAĆ

Nadal rozwijamy

Tym razem czeka nas trochę więcej pracy. Rozszerzymy nasz poprzedni program tak, by wykonywał kilka operacji na dwóch liczbach i wypisywał wyniki. Liczby będą wypisywane w postaci szesnastkowej, gdyż jak się okaże w programie, tak jest najłatwiej reprezentować liczby całkowite.

Funkcje w Asemblerze

W asemblerze można tworzyć własne funkcje, które będą wywoływane podczas jego pracy. Ogólny wzorzec tworzenia funkcji jest następujący:

```
  .type mojaFunkcja,@function
mojaFunkcja:
  [...] instrukcje [...]
  RET
```

Jak widać wykorzystujemy tutaj dyrektywę .type . W każdej funkcji bardzo ważna jest instrukcja RET, która pozwala programowi wrócić do poprzedniego miejsca pracy, z którego "wyskoczył" by wykonać funkcję.
Funkcje wywołujemy w programie poprzez podanie instrukcji CALL oraz jako argumentu etykiety początku funkcji (w tym wypadku mojaFunkcja).

Kolejne operacje logiczne

Podczas tego laboratorium będziemy potrzebowali skorzystać z funkcji logicznych. Aby przeprowadzić logiczne operacje na liczbach bajtowych korzystamy z instrukcji ANDB (i,oraz/koniunkcja), ORB (lub/alternatywa), XORB (albo/dysjunkcja). Tak jak miało to miejsce w innych instrukcjach tego typu, tak i tutaj jako argumenty podajemy dwie liczby bajtowe, a wynik otrzymamy w drugim z podanych rejestrów.

Instrukcje porównania

Jest to niestety temat bardzo grząski i niebezpieczny, bo instrukcji porównania i skoków jest od groma i tak na prawdę ciężko byłoby to wszystko spamiętać. Warto jednak poznać choć kilka instrukcji pozwalających na porównywanie rejestrów i liczb.
Pierwszą taką instrukcją będzie CMPB, która jako argumenty przyjmuje dwie liczby bajtowe (pierwsza z nich może być stała) i je porównuje. Wynik tej instrukcji nie zapisuje się w drugim argumencie, lecz jest zapamiętywany w specjalnym rejestrze. Nie jest ważnym w jakim rejestrze będzie zapamiętany wynik, tylko jak go wykorzystać. Do tego celu wykorzystamy instrukcje skoków.

Skoki

Kolejny grząski temat. Ale i w tym wypadku nie należy załamywać rąk, tylko zapamiętać pewne rzeczy. Wykorzystując instrukcję CMPB porównaliśmy dwie liczby. Teraz używamy instrukcji JB aby doknać skoku. Co to jest skok? Generalnie rzecz biorąc skok oznacza to samo co w C, czyli przeniesienie się do pamięci programu w miejsce oznaczone przez

etykietę podaną jako argument skoku. Skok, który zaraz zaprezentuję jest skokiem warunkowym i zależy od wcześniejszego porównania. JB, o którym już wspomniałem skacze do wskazanej etykiety, jeżeli wartość drugiego argumentu była mniejsza od wartości argumentu pierwszego. Spójrzmy na przykład:

```
  CMPB  $10,%al
  JB  mniejsze
```

W powyższym przykładzie porównujemy zawartość rejestru %al z liczbą 10 i jeżeli w rejestrze jest liczba mniejsza to skaczemy do miejsca etykietowanego jako "mniejsze", by wykonać dalej instrukcje.
Skoki mogą być bardzo problematyczne, jednak wprawa w pisaniu programów w asemblerze pozwala z czasem nabrać nawyku w dobieraniu ich oraz poprawnym stosowaniu.
Drugim typem skoku jaki dzisiaj poznamy jest skok bezwarunkowy JMP. Jako jego argument podajemy etykietę, do której skaczemy. Ważne jest to, że skaczemy zawsze! Skok ten często jest używany aby ominąć pewne instrukcje, które były użyte już przy skoku warunkowym. Przykład:

```
  CMPB $10,%al
  JB mniejsze
  SUB $1,%al
  JMP dalej
mniejsze:
  ADD $1,%al
dalej:
  [...]
```

W powyższym przykładzie porównujemy liczbę z rejestrem i jeżeli w rejestrze jest liczba mniejsza od 10 to dodajemy do niej 1, a jeżeli jest większa (nie nastąpi skok JB i pójdziemy dalej) odejmiemy 1 z rejestru i przeskoczymy dodanie liczby 1 (gdyż np. byłoby dla nas to bezcelowe, nie to chcemy osiągnąć) przechodząc do etykiety "dalej". Chyba skok ten jest już jasny.

Przesunięcie bitowe

Czasami chcemy skorzystać z przesunięcia bitowego by pozbyć się młodszych bitów w bajcie. Nazywamy to przesunięciem bitowym w prawo. Aby wykonać taką instrukcję korzystamy z mnemonika SHR, który przesuwa nam bity w prawo. Jako argument pierwszy podajemy ilość bitów do przesunięcia, a jako drugi rejestr, na którym będziemy przeprowadzać operację:

```
MOVB $4,%al
SHR $2,%al
```

Po wykonaniu takich instrukcji w %al będziemy posiadali liczbę 1 (usunęliśmy dwa młodsze bity równe 0.

Jeszcze o ASCII

Gdy przychodzi pora na wypisywanie jakiś liczb na ekranie, trzeba zacisnąć zęby i pomyśleć, jak zaprezentować pewne cyfry za pomocą znaków ASCII. Nie jest to jednak takie trudne. W większości przypadków wystarczy instrukcja:

```
  ADD $'0',%al
```

Podajemy znak cyfry zero jako argument (program przeliczy ten znak na postać liczby w kodzie ASCII. Jeżeli w rejestrze %al mieliśmy wybraną cyfrę od 0-9 to dodanie do niej numeru (kodu) znaku zero pozwoli nam otrzymać kod znaku wybranej cyfry. Gorzej gdy mamy liczby większe od 9. Wtedy najlepsze jest stosowanie liczb szesnastkowych, lecz trzeba bajt podzielić na dwie liczby po 4 bity. Trzeba też sprawdzać czy liczba jest większa od 9 (jeżeli tak, trzeba podobną metodą policzyć wtedy kod znaku odpowiedniego A-F dla 10-15). To własnie trzeba będzie osiągnąć w programie końcowym.

Pora na zadanie

Tym razem mamy przygotować aplikację, która wypisze na ekranie po kolei pierwszą liczbę, drugą liczbę, ich sumę, różnicę, alternatywę, koniunkcję i dysjunkcję. Liczby mają być przedstawione w postaci szesnastkowej (podpowiedź: można użyć przesunięcia bitowego by przeliczyć na szesnastkową formę po 4 bity bajta, warto też użyć koniunkcji 0x0F by zdobyć 4 młodsze bity). Poniżej program, który realizuje zadanie:

```
#-----------------------------------------------------------------
# Program LAB_2.S - Asemblery Laboratorium IS II rok
# Zadaniem progrmu jest przetworzenie liczb zap omocą pewnych operacji
# a następnie wypisanie wyników. Operacje wykonujemy na liczbach bajtowych
#-----------------------------------------------------------------

  .equ  kernel,0x80  #Linux system functions entry
  .equ  write,0x04  #write data to file function
  .equ  exit,0x01  #exit program function
  .equ  stdout,0x01   #handle to stdout

  .data

arg1txt:
  .ascii  "Arg1 = " # przygotowujemy teksty potrzebne do wyswietlenia liczb
arg2txt:
  .ascii  "Arg2 = "
sumtxt:
  .ascii  "Sum  = "
difftxt:
  .ascii  "Diff = "
ortxt:
  .ascii  "OR   = "
andtxt:
  .ascii  "AND  = "
xortxt:
  .ascii  "XOR  = "
arg1:        #przygotowujemy liczbę pierwszą do operacji zapisem
szesnastkowym (160)
  .byte  0xA0    # zapis taki jest prostszy
arg2:        #przygotowujemy drugą liczbę (5)
```

```
  .byte  0x05
result:        #tutaj będziemy przechowywać wyniki (nie moga przekroczyć
255)
  .byte  0
tmp: # zmienna pomocnicza do obliczeń
  .byte  0
restxt: # tekst, który będzie trzymał nasze wyniki
  .ascii  "  \n"
txtlen: # tutaj ustalamy dlugość dla tekstow do wyswietlania liczb (wyzsze
linijki)
  .long  7
reslen: # ustalamy dlugosc tekstu z wynikiem
  .long  3

  .text
  .global _start

_start:
  # przygotowujemy argumenty do wyswietlenia
  MOVB  arg1,%al  # przekazujemy wartosc liczbową arg1
  MOVL  $arg1txt,%ecx # przekazujemy adres w pamieci gdzie mamy tekst dla
zmiennej 1
  CALL  displine # wywolujemy funkcje wyswietlania linii
  NOP

  # to samo dla drugiej liczby
  MOVB  arg2,%al
  MOVL  $arg2txt,%ecx
  CALL  displine
  NOP

  # dodajemy liczby
  MOVB  arg1,%al
  ADDB  arg2,%al
  MOVB  %al,result # przekazujemy wynik do zmiennej
  MOVL  $sumtxt,%ecx
  CALL  displine
  NOP

  # odejmujemy liczby od siebie
  MOVB  arg1,%al
  SUBB  arg2,%al
  MOVB  %al,result
  MOVL  $difftxt,%ecx
  CALL  displine
  NOP

  # tym razem stosujemy operator logiczny OR
  MOVB  arg1,%al
  ORB  arg2,%al
  MOVB  %al,result
  MOVL  $ortxt,%ecx
  CALL  displine
  NOP

  # operator AND
  MOVB  arg1,%al
  ANDB  arg2,%al
  MOVB  %al,result
  MOVL  $andtxt,%ecx
  CALL  displine
```

```
  NOP

  # operator XOR
  MOVB  arg1,%al
  XORB  arg2,%al
  MOVB  %al,result
  MOVL  $xortxt,%ecx
  CALL  displine
  NOP
theend:
  MOVL  $exit,%eax
  INT  $kernel

#---------------------------------------------------------------------
# displine - displays line (prompt + hexadecimal number)
#---------------------------------------------------------------------
  .type displine,@function
  # tworzymy tutaj funkcję, która będzie wypisywać nasz wynik
displine:
  MOVB  %al,tmp # pobieramy podany wynik do zmiennej tmp
  NOP
  MOVL  $write,%eax # wypisujemy tekst
  MOVL  $stdout,%ebx
    # nie podajemy %ecx, gdyz bylo ustawione juz przed wywolaniem funkcji
  MOVL  txtlen,%edx
  INT  $kernel
  NOP

  MOVB  tmp,%al # wprowadzamy liczbe z tmp
  AND  $0x0F,%al # operujemy na niej by pobrać tylko młodsze 4 bity
  CMPB  $10,%al # sprawdzamy czy liczba jesst mniejsza niz 10 (jest cyfra)
  JB  digit1 # gdy jest cyfra skaczemy do obslugi cyfry
  ADDB  $('A'-0x0A),%al # w przeciwnym razie bierzemy litere 'A' i
odejmujemy 10,
      # po czym przesuwamy o liczbe w %al, ktora jest wieksza od 10
  JMP  insert1 # przeskakujemy do dalszej obslugi
digit1:
  ADDB  $'0',%al # jest cyfra wiec dodajemy do znaku 0, zeby otrzymac cyfre
0-9
insert1:
  MOV  %al,%ah # przenosimy otrzymana cyfre szesnastkowa do %ah (czyli
pierwsza liczba)
  NOP
  # znowu przenosimy tmp -> %al
  MOVB  tmp,%al
  SHR  $4,%al
      # przesunięcie bitowe w prawo o 4, w ten sposob mamy teraz tylko
starsze bity
  CMPB  $10,%al # czy mniejsze od 10
  JB  digit2
  ADDB  $('A'-0x0A),%al #tak samo jak poprzednio
  JMP  insert2
digit2:
  ADDB  $'0',%al
insert2:
  MOVW  %ax,restxt # otrzymany wynik szesnastkowy przenosimy do tekstu
wyniku
  NOP
  MOVL  $write,%eax
  MOVL  $stdout,%ebx
  MOVL  $restxt,%ecx #tekst do wyniku
```

```
MOVL  reslen,%edx # wartosc reslen - dlugosc wyniku 3 znaki
INT  $kernel
RET
```

LABORATORIUM 3 - SORTOWANIE

Wstęp

Kolejne laboratorium wprowadza studenta w następny stopień wtajemniczenia używania asemblera. Tym razem napisany zostanie program, który realizował będzie sortowanie tablicy na dwa sposoby: poprzez sortowanie bąbelkowe oraz przez wstawianie. Najpierw jednak omówimy nowe elementy języka, które poznamy i wykorzystamy w tym laboratorium.

Tworzenie tablic

Tworzenie tablic w asemblerze najprościej zrealizować poprzez utworzenie w sekcji .data listy kolejnych elementów. Przykładowo:

```
.data
tabela: .long 2,4,6,2,6,8,2,3,4,6
```

W ten sposób utworzymy tablicę, na której będziemy mogli operować. Dobrym pomysłem (wręcz niezbędnym) będzie też ustalenie ilości elementów tak, by mogła ona być zawsze szybko modyfikowana w kodzie. Do tego celu użyjemy kodu:

```
lElementow: .long ( . - tabela ) >> 2
```

Trzeba jednak pamiętać, by instrukcję tę zamieścić bezpośrednio pod deklaracją tablicy. Ponadto musimy patrzeć na typ danych, które chcemy trzymać w tablicy. Przesunięcie bitowe o dwa miejsca (realizowane poleceniem >> 2) powoduje podzielenie ilości bajtów zajmowanych przez tablicę na 4, gdyż każda liczba typu long zajmuje 4B w pamięci. W ten sposób mamy już rozmiar tablicy.
Aby się odwołać do tablicy stosujemy najczęściej zapis w formie:

```
MOVL tabela(,%ebp,4),%eax
```

Pierwszym argumentem jest etykieta do miejsca w pamięci, gdzie tablica się zaczyna (w naszym przypadku tabela). Następnie podaje się offset, jednak nie jest on zwykle stosowany, gdyż wystarczy nam etykieta użyta przed chwilą. Później podaje się rejestr przechowujący indeks oraz rozmiar danych, które służą do obliczenia przesunięcia się indeksem po tablicy i informują procesor ile bajtów danych ma pobrać. Za pomocą polecenia MOV dane te są kopiowane do rejestru %eax.

Skok JZ

W poprzednim laboratorium zaprezentowane zostały instrukcje porównania i skoku. Tym razem wykorzystana zostanie instrukcja JZ, która wykonuje skok, gdy wynikiem porównania jest 0 (odwrotnością tej instrukcji jest JNZ, który dokonuje skoku, gdy wynik porównania jest różny od 0). Oznacza to nie mniej, nie więcej, że ma być wykonany skok, gdy porównywane dwie wartości są sobie równe. Można też stosować skok JE, który wykona się, gdy wartości będą sobie równe. W ten sposób poznajemy całą gamę skoków:

```
JB # skok jeżeli jest mniejsza (below)
JBE # skok gdy mniejsza lub równa
JE # skok gdy równa (equal)
JAE # skok gdy większa lub równa
JA # skok gdy większa (above)
```

Pętla LOOP

Stosując skoki warunkowe możemy przemieszczać się w obrębie programu sprawdzając warunki przez nas wybrane. Jednak istnieje dużo prostsza pętla, która niejednokrotnie sprawia duże problemy. Jej prostota polega na sposobie użycia. W miejscu gdzie chcemy skoczyć piszemy polecenie LOOP oraz etykietę do miejsca skoku, a skok ten bezwarunkowo jest wykonywany. Proste! Jednak warunek tak na prawdę istnieje, lecz jest ukryty. Przed skokiem sprawdzana jest zawartość rejestru %ecx jest zmniejszana o 1 i jeżeli jest nadal większa od 0 następuje skok. W przeciwnym razie skok jest omijany i program idzie dalej. Powoduje to niejednokrotnie problemy, ponieważ nie można (teoretycznie o czym w przyszłości!) modyfikować rejestru %ecx oraz musimy pamiętać, by nadać mu przed pętlą żądaną wartość. Pętla taka jest odpowiednikiem pętli do{}while(); z C.

Zamiana wartości miejscami

Jeżeli mając jakąś wartość wpisaną do rejestru i inną do pamięci, chcielibyśmy zamienić je miejscami, nie musimy stosować trzech instrukcji korzystających ze zmiennej tymczasowej (lub kombinować na obliczeniach by jej nie użyć). Asembler pozwala na wykonanie takiej operacji w jednej instrukcji za pomocą polecenia XCHG (XCHGL dla liczb 4B). Jako argumenty podajemy mu rejestr i miejsce w pamięci, w których znajdują się dane, które chcemy zamienić miejscami.

Używanie stosu

Stos jest w asemblerze ważnym elementem, jeżeli chcemy pracować na danych, których nie mamy gdzie przechowywać. Warto tutaj zauważyć, że podczas gdy w pamięci przydzielonej programowi, jej część dla programu i zdefiniowanych zmiennych znajduje się na początku, to stos znajduje się na końcu tego segmentu, co pozwala na efektywne wykorzystanie przestrzeni pamięci programu. Dlatego też gdy dodajemy coś do stosu, jego wskaźnik się zmniejsza, a nie zwiększa (przesuwamy się w pamięci w lewo, lub też w dół, zależy jak patrzeć), natomiast gdy pobieramy ze stosu wskaźnik się zwiększa. Operacje na stosie są wykonywane za pomocą poleceń PUSH i POP. Pierwsze z nich umieszcza na stosie jakąś zmienną, którą podajemy jako argument. Polecenie POP pobiera nam zmienną ze stosu i umieszcza w rejestrze podanym jako argument tego polecenia. Ważne jest to, że argumentem tych poleceń może być tylko rejestr (nie przeniesiemy z pamięci do pamięci). Warto także wiedzieć, że używanie stosu wpływa na rejestr %ebp, dlatego trzeba uważać w pracy z nim. Więcej na ten temat w następnych laboratoriach. Na razie wystarczy nam wiedzieć jak umieszczać i zdejmować ze stosu zmienne.

Funkcje - pierwsze informacje

Używanie i tworzenie funkcji jest ważnym elementem każdego języka programowania. W przypadku asemblera jest to dosyć proste i stwarza wiele możliwości, trzeba jednak pamiętać o pewnych zasadach. Aby utworzyć funkcję w asemblerze korzystamy z polecenia:

```
  .type mojaFunkcja,@function
mojaFunkcja:
   ...
   RET
```

Jak widzimy korzystamy z dyrektywy .type i podajemy po niej nazwę funkcji, a po przecinku specjalny znacznik @function. Ważne jest to, że etykieta znajdujące się na początku funkcji musi być taka sama jak jej nazwa. Po napisaniu funkcji, na jej końcu umieszczamy polecenie RET, które jest znakiem powrotu z funkcji. Funkcje mogą być pisane w dowolnym miejscu, np. na końcu programu, by nie psuć czytelności kodu. Aby wywołać funkcję w programie korzystamy z polecenia CALL, gdzie jako argument podajemy etykietę (nazwę) funkcji. Funkcje mogą same siebie tak wywoływać (rekurencja), o czym będzie w kolejnych laboratoriach. Problemem stosowania funkcji jest przekazywanie parametrów (argumentów). Jak widać, nigdzie nie deklarujemy typów, liczby i zmiennych, które służyłyby za parametry. Argumenty można przekazywać do funkcji na trzy sposoby. Pierwszym z nich jest stosowanie zmiennych programu, które dla funkcji są dostępne, gdy jest pisana w tym samym programie. Jednak narzuca to tworzenie funkcji i programu przez jedną osobę (nazwy zmiennych, etykiet), a w dodatku zmienne te muszą cały czas być zadeklarowane w programie. Uniemożliwiałoby to stosowanie zatem bibliotek. Drugą metodą jest stosowanie rejestrów (co wykorzystamy w programie na końcu, z racji prostoty i niewielkiej skali działania programu). W ten sposób argumenty są już przesyłane do funkcji systemowych (choćby wypisywanie na ekranie - patrz laboratorium 1). Jednak ta metoda jest ograniczona przez wzgląd na ilość rejestrów. Pozwala to na przekazanie do około sześciu wartości. Ostatnim, wydaje się, że najlepszym, a zarazem najbardziej skomplikowanym sposobem jest użycie stosu. Pozwala to na wrzucenie (niemal) nieograniczonej liczby zmiennych i wykorzystanie przez funkcję. Trzeba jednak wtedy bardzo ściśle pilnować wskaźników na stos. O tym jednak w kolejnych laboratoriach. Stosując też funkcje należy pamiętać, żeby na stos wrzucić rejestry, których chcemy użyć, a mogą przechowywać dane ważne z punktu widzenia programu wywołującego funkcję. Gdy będziemy już mówili o stosowaniu stosu do przekazywania argumentów funkcji, zobaczymy, że po stronie użytkownika leży odpowiedzialność za umieszczenie na stosie swoich rejestrów, a po stronie tworzącego funkcję sprzątanie po wszystkim co umieścił na stosie w swojej funkcji.

Program realizujący sortowanie

Poniżej znajduje się kod programu realizującego dwa rodzaje sortowania: bąbelkowego i przez wstawianie. Metoda jest ustalana na poziomie pisania programu przez zmienną switch. W zmiennej events zachowywane są ilości zmian jakie należało wykonać (XCHG) w każdym z algorytmów, co można wykorzystać np. w prezentacji jakości poszczególnych metod. Reszta kodu jest dobrze opisana, a najlepiej można zrozumieć te algorytmy sprawdzając w debuggerze lub notując zawartość rejestru na kartce.

```
#---------------------------------------------------------------
# Program LAB_3.S - Asemblery Laboratorium IS II rok
#---------------------------------------------------------------

.equ  kernel,  0x80
.equ  exit,  0x01
.equ  stdout,  0x01
.equ  write,  0x04
.equ  dig0,  '0'
```

```
#-----------------------------------------------------------------

  .data

table:          # table of items
  .long  10,70,50,90,60,80,40,20,0,30,55
  # mamy tablice liczb typu long, ktora chcemy cos jakos uzyc
count:          # count of items
  .long  ( . - table ) >> 2
  # obliczamy "to miejsce" - poczatek tabeli dzielone przez 4 bajty każdego
elementu
  # (przesuniecie bitowe powoduje podzielenie, to logiczne jest) i mamy
liczbe elementow
events:
  .long  0
switch:          # switch :-)
  .byte  1
  # czy wybieramy babelkowe(0) czy wstawianie(1)
item:
  .string  "Element "
line_no:
  .string  "   "
itemval:
  .string  " = "
number:
  .string  "     \n"
before:
  .string  "\nBefore:\n"
after:
  .string  "\nAfter:\n"
dataend:

.equ  item_len, before - item
.equ  bef_len, after - before
.equ  aft_len, dataend - after

#-----------------------------------------------------------------

  .text
  .global _start

_start:
  # wypisanie napisu "Before: "
  MOVL  $write,%eax
  MOVL  $stdout,%ebx
  MOVL  $before,%ecx
  MOVL  $bef_len,%edx
  INT  $kernel

  # wypisanie tablicy
  CALL  disptable

  # wybranie formy sortowania
  CMPB  $0,switch
  JZ  bubble # było 0 więc skaczemy do bąbelkowego


insert:
# tutaj sortowanie przez wstawianie
# najpierw przygotowanie elementow przed sortowaniem
  MOVL  count,%edx # podajemy liczbę elementów tablicy
```

```
      # iterator zewnetrznej petli
  MOVL  $0,events # wprowadzamy 0
  XOR  %ebp,%ebp # zerujemy rejestr pomocniczy
    # będzie służył do indeksowania tablicy
    # już posortowanej
outer1:
  DEC  %edx # zmniejszamy rozmiar tablicy nieposortowanej
  MOV  %ebp,%esi # przenosimy indeks końca tablicy posortowanej
  INC  %esi
  # zwiększamy indeks (wskazuje na pierwszy niesortowany)
  MOV  %edx,%ecx # przenosimy wartość rozmiaru nieposortowanej
    # dla pętli będzie LOOP
  MOVL  table(,%ebp,4),%eax # pobieramy ostatni posortowany element
  MOV  %ebp,%edi # jeszcze przenosimy indeks konca posortowanej
inner1:
  CMPL  table(,%esi,4),%eax # pobieramy pierwsza nieposortowana liczbe
      # i porownujemy z ostatnim elementem posortowanej tablicy
  JBE  noexch # jezeli posortowany jest mniejszy lub rowny nowemu to
      # nie zamieniamy niczego i...
      # zeby sortowac malejaco wystarczy zmienic na JAE
  MOV  %esi,%edi # znalazlem liczbę mniejszą od dotychczas najmniejszej
  MOVL  table(,%esi,4),%eax
    # pobieram te liczbe i ona bedzie teraz najmniejsza
  INCL  events
noexch:
  INC  %esi # ... tylko zwiekszamy indeks wskazujacy na pierwszy
      # nieposortowany element
  LOOP  inner1
    # inner1 wykona sie dopoki %ecx nie bedzie ==1
    # po wykonaniu w %eax mamy minimum
  XCHGL  %eax,table(,%ebp,4) # zamieniamy dwie liczby ze soba miejscami
  MOVL  %eax,table(,%edi,4) # wpisujemy te liczbe na miejsce minimum
  INC  %ebp # zwiekszamy indeks konca tablicy posortowanej (rozmiar)
  CMPL  $1,%edx
    # porównanie ilości pozostałych elementów w tablicy posortowanej
  JNZ  outer1 # jezeli nie jest równa 1 to od nowa robimy
  JMP  theend # inaczej wyskakujemy, bo to koniec



bubble:
# tutaj sortowanie przez bomblowanie
  MOVL  count,%edx
    # znowu rozmiar tablicy do edx to counter petli outer
  MOVL  $0,events # ilosc zmian
outer:
  DEC  %edx # i-- (sortujemy od konca tablicy)
  XOR  %esi,%esi # zerowanie esi
  MOV  %edx,%ecx # j=i
inner:
  MOVL  table(,%esi,4),%eax # zerowy element do tmpa
  CMPL  table+4(,%esi,4),%eax # porownanie tmp?=z elementem nastepnym
  JBE  noswap # tmp<nastepny wiec przeskok
  XCHGL  table+4(,%esi,4),%eax #zamiana miejscami tmp z drugim
  MOVL  %eax,table(,%esi,4)
  INCL  events # byla zamiana
noswap:
  INC  %esi # nastepny do porownania
  LOOP  inner
  CMPL  $1,%edx # skonczyla sie tablica? ojej...
  JNZ  outer # jezeli nie, to od nowa
```

```
theend:
# tutaj juz jest koniec...
  MOVL  $write,%eax
  MOVL  $stdout,%ebx
  MOVL  $after,%ecx
  MOVL  $aft_len,%edx
  INT  $kernel
  movw  $0x2020,line_no
  CALL  disptable
  MOVL  events,%ebx
  MOVL  $exit,%eax
  INT  $kernel

#-----------------------------------------------------------------

  .type disptable,@function

disptable:
# wyswietlanie tablicy
  XOR  %esi,%esi # zerowanie esi
  MOVL  count,%ecx # liczba elementow tablicy do ecx
dispitem:
  push  %ecx  # potrzebne ecx, wiec zapamietamy na stosie
  MOVL  table(,%esi,4),%ebx # pobieramy liczbe z tablicy do ebx
  CALL  makestr # i funkcja osobna nam z tej liczby robi string

  # i to wypisujemy sobie
  MOVL  $write,%eax
  MOVL  $stdout,%ebx
  MOVL  $item,%ecx
  MOVL  $item_len,%edx
  INT  $kernel

  pop  %ecx # oddajemy ecx
  INC  %esi # nastepny element tablicy bedzie
  LOOP  dispitem
    # polecenie LOOP skoczy do etykiety wybranej
    # wykorzystuje rejestr ecx do kontroli licby powtórzeń
    # jeżeli ecx!=0 to skok, else nieskok
    # decrementuje zawsze ecx przed sprawdzeniem
    # przy wielu petlach najlepiej uzywac tylko w najbardziej
    # wewnetrznej
  RET

#-----------------------------------------------------------------
  .type makestr,@function
makestr:
  MOV  %esi,%eax
  MOVL  $line_no + 2,%edi # ustawiamy sie na koncu ciagu znakow
  CALL  n2str
  MOV  %ebx,%eax
  MOVL  $number + 4,%edi # j.w.
  CALL  n2str
  RET


#-----------------------------------------------------------------
  .type n2str,@function
n2str:
```

```
  PUSH  %ebx
    # wrzuca dane na stosik
  PUSH  %edx
  MOVL  $10,%ebx
  movl  $0x20202020,number
  # wyczyszczenie napisu w pierwszych 4 znakach

nextdig:
  XOR  %edx,%edx
    # czyszczenie rejestru poprzez wyzerowanie
  DIV  %ebx
    # dzieli edx:eax przez podany argument
    # wynik w eax
    # mod w edx
  ADDB  $dig0,%dl
    # obliczamy numer znaku cyfry teraz wyliczonej
  MOVB  %dl,(%edi)
  # na miejsce wskazywane przez %edi daje wynik mod
  CMPL  $0,%eax # sprawdzenie czy jeszcze jest co dzielic
  JZ  empty # nie ma co dzielic, koniec!
  DEC  %edi # jak jest co dzielic, to cofamy sie w tablicy znakow
  JMP  nextdig
empty:
  POP  %edx
    # z powrotem pobieramy dane ze stosu
  POP  %ebx
  RET

#-----------------------------------------------------------------
```

LABORATORIUM 4 - OPERACJE NA PLIKACH

Wstęp

Kontynuując naukę języka Asembler nie sposób w końcu nie zająć się w sposób poważniejszy funkcjami systemowymi. Dotąd korzystaliśmy z funkcji pisania na ekran. Jednak z czasem przychodzi pora by nauczyć się także wczytywać znaki z klawiatury, zapisywać dane do plików (także je wcześniej tworzyć) i odczytywać dane z plików, wcześniej zapisanych. Tym własnie zajmiemy się na laboratorium 4.

Funkcje kernela

Korzystając z funkcji systemowych powinniśmy pamiętać (lub mieć gdzieś zapisane) numery, którym odpowiadają w pamięci przydzielonej systemowi. Tworząc odpowiednie stałe w programie nie będziemy musieli korzystać bezpośrednio z kodów liczbowych, tylko z symbolicznych nazw. Poniżej wylistowałem potrzebne na tym laboratorium funkcje jądra systemu razem z ich numerami:

```
.equ  create,  0x08  #tworzenie pliku
.equ  open,  0x05  #otwieranie pliku
.equ  close,  0x06  #zamykanie pliku
.equ  read,  0x03  #czytanie danych z pliku
.equ  write,  0x04  #zapis do pliku
.equ  exit,  0x01  #wyjście z programu
```

Część z tych funkcji już znamy. Chocby pisanie, którego używaliśmy, żeby wyświetlić coś na monitorze. Zapis na monitor niczym nie różni się od zapisu do pliku z wyjątkiem argumentu wyjścia, na które piszemy. O pisaniu do pliku, będzie w stosownym czasie. Drugą znaną nam funkcją jest exit, któa kończy działanie programu. Funkcje systemowe stosujemy poprzez wywołanie przerwania na adresie 0x80 (zapisywanym u nas zwykle jako kernel). Jest to przerwanie na którym system oczekuje na polecenia i wykonuje instrukcje. Przypomnijmy sobie teraz używanie funkcji z poprzedniego laboratorium. Korzystaliśmy tam z polecenia CALL. Dla kernela nie używamy tego polecenia. Do tego służy nam polecenie INT (interrupt) tworzące przerwanie na wybranym adresie. Pod adresem kernela czeka na nas system, który wie, że argumenty posiada zapisane w rejestrach. Warto tutaj zaznaczyć różnicę: kernel używa rejestrów jako argumentów swoich funkcji. Dla funkcji z bibliotek lub pisanych przez nas samych zwykle powinniśmy wykorzystywać stos. Pora zatem omówić poszczególne funkcje.

Tworzenie pliku

Aby utworzyć plik stosujemy funkcję o adresie 0x08. Do rejestru %eax przekazujemy ten numer, by system wiedział jaką funkcji ma użyć. Drugim argumentem, w rejestrze %ebx, jest zmienna w pamięci programu, w której przechowywany później będzie uchwyt do pliku (zwrócony przez system). Jako trzeci argument podajemy liczbę, która będzie kodem uprawnień do pliku stoswanym w systemach *nix (gdyż kompilujemy nasze programy w takim środowisku). Oto przykład:

```
.equ  kernel,  0x80
.equ  create,  0x08
file_n:      #nazwa pliku
```

```
  .string  "testfile.txt"
file_h:      #uchwyt pliku
  .long    0
[...]
  MOVL  $create,%eax  #adres funkcji tworzącej
  MOVL  $file_n,%ebx  #EBX przechowuje uchwyt do pliku
  MOVL  $mode,%ecx  #ECX przechowuje atrybuty dla pliku
  INT  $kernel
  NOP
[...]
```

Od tego momentu w zmiennej uchwytu pliku będziemy mieli zapisany adres, który będzie nam identyfikował jednoznacznie plik na dysku. Dzięki temu wykonamy operacje zapisu i odczytu. Warto także pamiętać, że jeżeli coś poszło nie tak z utworzeniem pliku, wartość w rejestrze %eax po użyciu funkcji będzie mniejsza od 0. W przeciwnym razie wszystko jest w porządku. Można to wykorzystać sprawdzając %eax warunkiem i obsługiwać w ten sposób błędy. Liczba większa od 0 w tym rejestrze będzie właśnie uchwytem do pliku i musimy ją niezwłocznie przenieść do jakiejś zmiennej w pamięci, by móc potem z niej korzystać.

Zapis do pliku

Tak na prawdę funkcję, którą zaraz przedstawię znamy już z poprzednich laboratoriów. Jednak wtedy stosowaliśmy zapis na ekran, podając jako argument uchwytu pliku wartość 0x01. Jest to zarezerwowany uchyt, który wskazuje na ekran. W ten własnie sposób, możemy pisać zarówno na ekran, jak i do pliku, stosując tę samą funkcję systemu. Składnia zapisu jest już nam znana, jednak przypomnijmy ją:

```
  MOVL  %eax,file_h  #zapisujemy do file_h uchwyt do pliku
  MOVL  $write,%eax  #funkcja pisania
  MOVL  file_h,%ebx  #EBX przechowuje uchwyt do pliku
  MOVL  $txtline,%ecx  #ECX wskazuje na bufor tekstu do zapisania
  MOVL  txtlen,%edx  #ilosc bajtow do zapisania
  INT  $kernel
```

Powyższy przykład jest już nam dobrze znany. Różnicą jedyną tutaj jest podanie uchwytu do pliku jako argumentu w rejestrze %ebx, co już było tłumaczone. W efekcie zastosowania tych instrukcji, w pliku pojawią się napisy. I tym razem jeżeli coś poszło nie tak możemy sprawdzić to poprzez ustalenie wartości %eax (w ogólności funkcje systemowe zawsze zwracają wynik poprzez rejestr %eax). Jeżeli zawartość %eax będzie różna od %edx (ilośc tekstu, który chcieliśmy zapisać), to znaczy, że nie zapisano całego ciągu znaków.

Odczyt pliku

Do odczytu pliku stosujemy funkcję read (0x03). Poniżej przedstawiam kod razem z objaśnieniami:

```
  MOVL  $read,%eax  #funkcja czytania
  MOVL  file_h,%ebx  #uchwyt do pliku
  MOVL  $buffer,%ecx  #ECX wskazuje na bufor w pamięci
  MOVL  bufsize,%edx  #ile bajtów mamy zczytać
  INT  $kernel
```

Trzeba jednak pamiętać, że jeżeli w pliku jest więcej danych niż w buforze się mieści, nie będziemy mogli ich zczytać. Jako rozmiar bufora trzeba zawsze podawać jego rozmiar, a nie ilość znaków w pliku (inaczej spowoduje to poważne uszkodzenie programu). Inną ważną rzeczą jest pamiętanie o tym, by czytać tylko z pliku który jest otwarty. Nowo utworzony plik z poprzednich punktów od razu zostaje otwarty i można do niego pisać. Po zapisie musi być zamknięty. Nowe otwarcie musi nastąpić przed czytaniem. Jeżeli w wyniku czytania coś się nie uda, w rejestrze %eax będziemy mieli zapisaną wartości ujemne.

Otwieranie pliku

Nie można czytać z pliku dopiero co utworzonego, gdyż będzie on od razu otwarty w trybie do zapisu. Aby móc czytac z pliku, trzeba go najpierw zamknąć (opisane dalej), a następnie otworzyć następującym sposobem:

```
  MOVL  $open,%eax  #otwieranie pliku
  MOVL  $file_n,%ebx  #EBX wskazuje na nazwę pliku
  MOVL  $flags,%ecx  #flagi otwieranego pliku (np. zapis, odczyt,
nadpisywanie)
  MOVL  $mode,%edx  #tryb otwieranego pliku (atrybuty, tak jak przy
tworzeniu)
  INT  $kernel
```

W rejestrze %eax po wykonaniu tych instrukcji będziemy posiadali uchwyt do otwartego pliku. Skorzystamy z tego uchwytu np. przy odczytywaniu zawartości pliku.

Zamykanie pliku

Aby zamknąć plik korzystamy z funkcji systemowej o numerze 0x06. Funkcja ta jest najprostsza, gdyż do jej użycia nie potrzeba wielu parametrów:

```
  MOVL  $close,%eax  #funkcja 0x06
  MOVL  file_h,%ebx  #uchwyt do pliku
  INT  $kernel
```

I tym razem jeżeli coś się nie uda w %eax będziemy mieli ujemną liczbę

Pora na program

W końcu piszemy program. Ma on na celu utworzenie pliku, zapisanie w nim jakiegoś ciągu znaków i zamknięcie. Następnie ten sam plik otwieramy ponownie, wypisujemy jego zawartość na ekranie i zamykamy. Jeżeli coś nie pójdzie po naszej myśli przeskakujemy do sekcji obsługi błędów i po wypisaniu komunikatu wychodzimy z programu. Program nie jest prosty w zrozumieniu. Wystarczy znać funkcje, o których pisałem wcześniej.

```
#-----------------------------------------------------------------
# Program LAB_4.S - Asemblery Laboratorium IS II rok
#-----------------------------------------------------------------

.equ  kernel,  0x80  #Linux system functions entry
.equ  create,  0x08  #create file function
.equ  open,  0x05  #open file function
.equ  close,  0x06  #close file function
```

```
.equ  read,  0x03  #read data from file function
.equ  write,  0x04  #write data to file function
.equ  exit,  0x01  #exit program function

.equ  mode,  0x180  #attributes for file creating
.equ  flags,  0  #attributes for file opening

.equ  stdout, 1 # standardowe wyjscia
.equ  stderr, 2

  .data

file_n:      #file name (0 terminated)
  .string  "testfile.txt"
file_h:      #file handle
  .long    0
txtline:    #text to be written to file
  # ascii umieszcza bez "\0", a string z "\0". Ważne dla OSa
  .ascii  "Tekst zapisany w pliku\n"
txtlen:      #size of written data
  .long    ( . - txtline )
buffer:      #buffer for file data
  # pozwala utworzyc bufor space(wielkość bufora, wartość początkowa
bajtów)
  # wypełnia dodatkowo przestrzeń zerami
  .space    128, 0
bufsize:    #size of buffer
  .long    ( . - buffer )
b_read:      #size of read data
  .long    0
errmsg:      #file error message
  .ascii  "Błąd pliku!\n"
errlen:
  .long    ( . - errmsg )
toomsg:      #file too big error message
  .ascii  "Plik zbyt duży!\n"
toolen:
  .long    ( . - toomsg )
cntmsg:      #another message
  .ascii  "W pliku zapisano następujący tekst:\n"
cntlen:
  .long    ( . - cntmsg )

  .text
  .global _start

_start:
  # najpierw utworzenie pliku
  MOVL  $create,%eax  #adres funkcji tworzącej
  MOVL  $file_n,%ebx  #EBX przechowuje uchwyt do pliku
  MOVL  $mode,%ecx  #ECX przechowuje atrybuty dla pliku
  INT  $kernel
  NOP

  # sprawdzanie poprawnosci utworzonego pliku
  CMP  $0,%eax
  JL  error    #jeżeli EAX<0 to znaczy, że coś jest źle

  # wpisujemy do pliku
  MOVL  %eax,file_h  #zapisujemy do file_h uchwyt do pliku
  MOVL  $write,%eax  #funkcja pisania
```

```
MOVL  file_h,%ebx  #EBX przechowuje uchwyt do pliku
MOVL  $txtline,%ecx  #ECX wskazuje na bufor tekstu do zapisania
MOVL  txtlen,%edx  #ilosc bajtow do zapisania
INT  $kernel

# spradzenie czy wpisano jak trza
CMP  %edx,%eax
JNZ  error    #jeżeli EAX<>EDX to cos poszlo nie tak

# zamykanie pliku
MOVL  $close,%eax  #close function
MOVL  file_h,%ebx  #file handle in EBX
INT  $kernel

# czy wszystko w porzadku?
CMP  $0,%eax
JL  error    #if EAX<0 then something went wrong

# otwieranie na nowo pliku
MOVL  $open,%eax  #open function
MOVL  $file_n,%ebx  #EBX points to file name
MOVL  $flags,%ecx  #flags of opened file in ECX
MOVL  $mode,%edx  #mode of opened file in EDX
INT  $kernel

# bledy
CMP  $0,%eax
JL  error    #if EAX<0 then something went wrong

# czytamy z pliku
MOVL  %eax,file_h  #store file handle returned in EAX
MOVL  $read,%eax  #read function
MOVL  file_h,%ebx  #file handle in EBX
MOVL  $buffer,%ecx  #ECX points to data buffer
MOVL  bufsize,%edx  #bytes to be read
INT  $kernel

# czy bledy?
CMP  $0,%eax
JL  error    #if EAX<0 then something went wrong

# to co przeczytalismy zapisujemy sobie
MOVL  %eax,b_read  #store count of read bytes

# zamkniecie pliku
MOVL  $close,%eax  #close function
MOVL  file_h,%ebx  #file handle in EBX
INT  $kernel

# ew. bledy
CMP  $0,%eax
JL  error    #if EAX<0 then something went wrong

# znowu bierzemy tekst
MOVL  b_read,%eax
CMPL  bufsize,%eax  #whole file was read ?
JAE  toobig    #probably not

# wypisujemy tekst ze mamy tekst...
MOVL  $write,%eax  #write function
MOVL  $stdout,%ebx  #file handle in EBX
```

```
  MOVL  $cntmsg,%ecx  #ECX points to message
  MOVL  cntlen,%edx  #bytes to be written
  INT  $kernel

  # wypisanie tekstu
  MOVL  $write,%eax  #write function
  MOVL  $stdout,%ebx  #file handle in EBX
  MOVL  $buffer,%ecx  #offset to first character
  MOVL  b_read,%edx  #count of characters
  INT  $kernel
  XOR  %ebx,%ebx
  JMP  theend     # i idziemy do konca
toobig:
  MOVL  $write,%eax  #write function
  MOVL  $stderr,%ebx  #file handle in EBX
  MOVL  $toomsg,%ecx  #ECX points to toobig message
  MOVL  toolen,%edx  #bytes to be written
  INT  $kernel
  MOVL  $stdout,%ebx
  JMP  theend
error:
  MOVL  $write,%eax  #write function
  MOVL  $stderr,%ebx  #file handle in EBX
  MOVL  $errmsg,%ecx  #ECX points to file error message
  MOVL  errlen,%edx  #bytes to be written
  INT  $kernel
  MOVL  $stderr,%ebx
theend:
  MOVL  $exit,%eax  #exit program function
  INT  $kernel
```

LABORATORIUM 5 - OPERACJE NA CIĄGACH ZNAKÓW

Wstęp

Podczas pracy nad poprzednimi programami tworzyliśmy już własne funkcje, korzystaliśmy z funkcji systemowych i zapisywaliśmy pliki, odczytywaliśmy ich zawartość i pisaliśmy je na ekranie. Nadeszła pora by nauczyć się wczytywać znaki z klawiatury i obrabiać wprowadzone w ten sposób ciągi.

Czytanie z klawiatury

W poprzednim artykule napisałem jak używać funkcji wczytywania z pliku znaków. Aby wczytać znaki z klawiatury korzystamy z tej samej funkcji, jednak jako uchwyt do pliku przekazujemy wartość 0. Najlepiej jest wtedy korzystać ze wspólnej zmiennej w pamięci, która może być użyta zarówno do plików jak i ekranu oraz klawiatury. Wartość tej zmiennej dla zapisu musi wynosić 0, gdyż jest to kod odczytywania ze standardowego wejścia systemowego (klawiatury). Zatem fragment kodu wczytujący znaki z klawiatury wygląda tak:

```
        MOVL    $read,%eax      #funkcja wczytywania
        MOVL    file_h,%ebx     #uchwyt do pliku (tutaj klawiatura = 0 )
        MOVL    $buffer,%ecx    #wskazanie na bufor, do którego wczytujemy
dane
        MOVL    bufsize,%edx   #ilość danych do wczytania (zwykle rozmiar
bufora)
        INT     $kernel
```

Widzimy zatem, że funkcja jest zupełnie taka sama jak w przypadku poprzedniego tekstu. Nic się nie zmieniło, po za możliwościami, jakie nam teraz oferuje znajomość czytania z klawiatury.

Operacje na znakach

Aby przeprowadzać operacje na znakach zapisanych w buforze będziemy korzystali ze specjalnych poleceń Asemblera. Żeby przejść po całym buforze skorzystamy z rejestru %ecx oraz pętli LOOP (patrz poprzednie teksty). Jednak, żeby operować na elementach bufora skorzystamy z instrukcji LODSB oraz STOSB. Aby jednak móc z nich korzystać, musimy do rejestru %esi przekazać adres w pamięci, gdzie znajduje się nasz bufor. Dodatkowo informację tę przekazujemy także do rejestru %edi. Oba będą wykorzystane przez wyżej wymienione instrukcje do pobierania i umieszczania danych na miejscu.

Teraz pora powiedzieć do czego służyć będą nam instrukcje LODSB oraz STOSB. Pierwsza pobiera z bufora bajt danych i umieszcza go w rejestrze %al. Możemy dzięki temu rozpocząć obróbkę pobranych znaków. Instrukcja ta operuje także na jednym z rejestrów %esi lub %edi, by się przesuwać po tekście.

Instrukcja STOSB umieszcza nam bajt danych z powrotem w buforze. O ile wcześniej pobraliśmy dane za pomocą LODSB, będzie ona umieszczona w tym samym miejscu. Przeskakiwanie tej instrukcji pozwala nam na nie umieszczanie niepożądanych znaków w buforze. Wykorzystuje ona odrębny rejestr (prawdopodobnie %edi), aby wskazywać na miejsce w którym umieszcza dane i powiększa go przy każdym użyciu. Jeżeli zatem

pominiemy tę instrukcję w którymś obiegu pętli, będziemy odtąd umieszczać kolejne pobrane znaki, na pozycji o jeden mniejszej. Pozwoli to w naszym programie na usuwanie np. spacji.

Program

To wszystko, co nowego można poznać na dzisiaj. Poniżej program z laboratorium, który wykorzystuje podane wyżej informacje. Wczytuje on ciąg znaków z klawiatury i poddaje obróbce, by wypisać ostateczną formę tekstu.

```
#---------------------------------------------------------------------
# Program LAB_5.S - Asemblery Laboratorium IS II rok
#---------------------------------------------------------------------

.equ    kernel, 0x80    #Linux system functions entry
.equ    create, 0x08    #create file function
.equ    read,   0x03    #read data from file function
.equ    write,  0x04    #write data to file function
.equ    exit,   0x01    #exit program function

.equ  stdout, 1
.equ  stderr, 2

       .data

buffer:                 #buffer for file data
       .space          1024, 0
bufsize:                #size of buffer
       .long           ( . - buffer )
b_read:                 #size of read data
       .long           0
errmsg:                 #file error message
       .ascii  "File error!\n"
errlen:
       .long           ( . - errmsg )
toomsg:                 #file too big error message
       .ascii  "File too big!\n"
toolen:
       .long           ( . - toomsg )
promptmsg:
       .ascii  "String: "
promptlen:
       .long           ( . - promptmsg )
befmsg:
       .ascii  "Before:\n"
beflen:
       .long           ( . - befmsg )
aftmsg:
       .ascii  "After:\n"
aftlen:
       .long           ( . - aftmsg )
file_h: # tak na prawde to to jest stdin=0
       .long  0

       .text
       .global _start

_start:
       NOP
       # wypisanie komunikatu o wpisaniu ciagu znaków
```

```
        MOVL    $write,%eax     #write function
        MOVL    $stdout,%ebx    #file handle in EBX
        MOVL    $promptmsg,%ecx #ECX points to message
        MOVL    promptlen,%edx #bytes to be written
        INT     $kernel
        NOP

        # wczytywanie ciągu znaków
        MOVL    $read,%eax      #read function
        MOVL    file_h,%ebx     #file handle in EBX
        MOVL    $buffer,%ecx    #ECX points to data buffer
        MOVL    bufsize,%edx   #bytes to be read
        INT     $kernel
        CMP     $0,%eax
        JL      error           #if EAX<0 then something went wrong
        MOVL    %eax,b_read     #store count of read bytes
        CMPL    bufsize,%eax   #whole file was read ?
        JAE     toobig          #probably not
        NOP

        # wypisanie napisu przed i tego napisu
        MOVL    $write,%eax     #write function
        MOVL    $stdout,%ebx    #file handle in EBX
        MOVL    $befmsg,%ecx    #ECX points to message
        MOVL    beflen,%edx    #bytes to be written
        INT     $kernel
        NOP
        MOVL    $write,%eax     #write function
        MOVL    $stdout,%ebx    #file handle in EBX
        MOVL    $buffer,%ecx    #offset to first character
        MOVL    b_read,%edx    #count of characters
        INT     $kernel
        NOP

        # teraz obrobka bedzie
        MOVL    $buffer,%esi
        MOVL    %esi,%edi
        MOVL    b_read,%ecx

# zmiana tekstu na podstawie reguł
next:
        LODSB
        cmpb  $' ',%al
        je spacja
        cmpb  $'0',%al
        jb skip
        cmpb  $'9',%al
        jbe liczby2podkr
        CMPB  $'A',%al
        jb skip
        cmpb  $'Z',%al
        jbe zmiana
        CMPB    $'a', %al
        JB skip
        CMPB    $'z', %al
        JA skip
zmiana:
        xorb     $0x20, %al
        jmp skip
liczby2podkr:
  movb $'_',%al
```

```
  jmp skip
spacja:
    push %eax
    movl b_read,%eax
    sub  $1,%eax
    movl %eax,b_read
    pop %eax
    jmp skacz
skip:
       STOSB
skacz:
       LOOP next
    # wypisanie napisu After i tekstu zmienionego
       NOP
       MOVL    $write,%eax     #write function
       MOVL    $stdout,%ebx    #file handle in EBX
       MOVL    $aftmsg,%ecx    #ECX points to message
       MOVL    aftlen,%edx    #bytes to be written
       INT     $kernel
       NOP
       MOVL    $write,%eax     #write function
       MOVL    $stdout,%ebx    #file handle in EBX
       MOVL    $buffer,%ecx    #offset to first character
       MOVL    b_read,%edx    #count of characters
       INT     $kernel
       NOP
       MOV     b_read,%ebx
       JMP     theend

toobig:
       MOVL    $write,%eax     #write function
       MOVL    $stderr,%ebx    #file handle in EBX
       MOVL    $toomsg,%ecx    #ECX points to toobig message
       MOVL    toolen,%edx    #bytes to be written
       INT     $kernel
       MOVL    $stdout,%ebx
       JMP     theend
error:
       MOVL    $write,%eax     #write function
       MOVL    $stderr,%ebx    #file handle in EBX
       MOVL    $errmsg,%ecx    #ECX points to file error message
       MOVL    errlen,%edx    #bytes to be written
       INT     $kernel
       MOVL    $stderr,%ebx
theend:
       MOVL    $exit,%eax      #exit program function
       INT     $kernel
```

LABORATORIUM 6 - FUNKCJE JĘZYKA C

Wstęp

Prędzej czy później przychodzi pora na połączenie możliwości języka Asembler z innymi dostępnymi od wielu lat. Podczas naszych laboratorium zapoznajemy się z możliwością pisania programów, które wykorzystują język C jako główny, użyty do pisania programów, jak i w stronę odwrotną. Dowiemy się zatem także, jak napisać program w Asemblerze, który używałby funkcji C. I właśnie ta druga możliwość zostanie zaprezentowana w tym artykule.

Kompilacja

Zaczynając dyskusję na temat kompilacji plików Asemblera i C, trzeba najpierw wyjaśnić kwestię kompilatora. Kompilator systemu linux, który zajmuje się językiem asembler nosi nazwę *as*. Jednak pozwala on jedynie na kompilację programów pisanych wjęzyku asemblera. Jeżeli chcemy móc korzystać z funkcji i możliwości języka C, musimy skorzystać z jego kompilatora. Każdy kompilator języka C potrafi także ptrzetwarzać kody asemblera, ponieważ do takiego właśnie kodu sprowadza kod C, zanim go skompiluje. Pytanie brzmi: dlaczego zatem w ogóle pisać cokolwiek w asemblerze, skoro C też zamienia się w asm? Odpowiedź na to pytanie pojawi się w późniejszym czasie (dodatku do laboratorium 6). Kompilatorem, którego my będziemy używali będzie GCC. O jego użyciu powiemy jednak dalej.

Programy w asemblerze w stylu C

Aby pisać programy, które będą traktowane jako elementy kodu C, trzeba trzymać się pewnych konwencji. Tak na prawdę korzystając z kompilatora C, powiemy mu jedynie, żeby uwzględnił nasze pliki. Mogą one być jedynie odrębnymi funkcjami, które wykorzystamy w programie C (lub innym programie asemblerowym) lub też (jak za chwilę pokażę) całymi programami. Możemy zatem powiedzieć GCC aby jako naszej głównej funkcji (main, znanej z C) używał kodu ze źródła asemblera. Jak? Odpowiedź jest bardzo prosta. Wystarczy nadć naszemu kodowi globalną etykietę o nazwie: *main*. To dosyć logiczne. Dlaczego jednak nie *_start* jak robiliśmy to poprzednio? Tak na prawdę gcc stworzy program, który będzie posiadał etykietę *_start*, a wewnątrz jego wywoła funkcje *main*. Tak się dzieje, gdy piszemy w C. Tak samo będzie też się działo gdy napiszemy kod w asemblerze z etykietą globalną main.

```
  .text
  .global main

main:
...
```

Używanie funkcji z C

Głównym jednak celem użycia kompilatora dla C, będzie korzystanie z dostępnych dla niego funkcji. Kompilator gcc podczas kompilowania pliku asemblera automatycznie dodaje standardowe biblioteki, których możemy używać w programie. Zatem pisząc program, gdy chcemy go zakończyć nie trzeba wywoływać przerwań zakończenia programu. Wystarczy wywołać funkcję *exit*. Poniżej pokazuję jak to zrobić:

```
call exit
```

Argumenty funkcji

Pokazany wyżej kod był trywialny. Jednak program zwykle potrzebuje do użycia funkcji, by podano mu argumenty. W poprzednim laboratorium pisałem na temat możliwości przekazywania argumentów funkcji. Dzisiaj czas wyjaśnić, że używając funkcji C zawsze umieszczamy argumenty na stosie. Sprawą funkcji i jej autora jest zadbanie o nasze rejestry i ich przywrócenie po działaniu oraz wyczyszczenie stosu, po zakończeniu działania funkcji. Jak umieścić dane na stosie? O tym była już mowa. Korzystamy z instrukcji *push*, która umieszcza wybrane dane na stosie.

Funkcja printf

Skoro już przedstawiliśmy niezbędny wstęp do funkcji, pora by zrobić wreszcie coś sensownego. Funkcja printf jako argumenty przyjmuje ciąg znaków nazywany maską oraz argumenty odpowiadające wartościom podstawianym do maski. Następnie cały ciąg jest wypisywany. Trzeba jednak pamiętać, że na stosie dane są odkładne w kolejności odwrotnej. Oznacza to, że jeżeli funkcja printf jako pierwszy argument bierze maskę, to musi być ona na stosia umieszczona jako ostatnia. Zatem aby wypisać na ekranie tekst wystarczy przekazać na stos wartości a na końcu adres do miejsca w pamięci, gdzie znajduje się maska. Poniższy program jest zwieńczeniem tego artykułu i prezentuje użycie printfa. Jeszcze jedną ważną rzeczą jest użycie gcc. Używamy go następująco:

```
$> gcc -o program program.s
```

Program końcowy

```
  .data
fmt:
  .asciz   "Wartosc = %d\n"
value:
  .long    15

  .text
  .global main

main:
  movl value, %eax # umieszczamy wartości
  pushl %eax
  movl $fmt, %ebx # podajemy maskę
  pushl %ebx # i także umieszczamy na stosie
  call printf
  addl $8, %esp
  nop
  pushl $0 # argument returna
  call exit
```

Mała uwaga! O stosie i jego wskaźnikach (które są tutaj użyte) przeczytać możesz w laboratorium 8.

LABORATORIUM 6A - DODATEK O KODZIE ASM W C

Wstęp

Dodatek ten ma pokazać jak używać funkcji pisanych w asemblerze w programach napisanych w C. Poniżej zaprezentuję program, który porównuje wyniki z obliczeń liczb Fibbonaciego w wersji C oraz Asemblera. Ma to służyć przybliżeniu zagadnienia pisania funkcji dla C za pomocą asemblera. Gdyby ktoś zadał sobie trud modyfikacji poniższego programu i zwiększenia ilości liczb liczonych, zobaczy, że w pewnym momencie wersja asemblera podaje złe wyniki. Jest to spowodowane ograniczoną ilością bajtów w rejestrze. Widać zatem, że dla dużych liczb C pozwala na użycie bardzo dużych rejestrów (a raczej ich fragmentów), tak by móc liczyć nieco dalej.

Test Fibbonaciego

Do celu testów przygotowany został następujący kod C:

```
#include <stdio.h>

int fib( unsigned int k )
{
  if( k == 0 )
    return 0;
  else if( k == 1 )
    return 1;
  else
    return fib( k - 2 ) + fib( k - 1 );
}


int main( void )
{
 int i;

 for( i = 0; i <= 40; i++ )
   printf( "Fib( %2d ) = %d FibA= %d\n", i, fib( i ), fiba( i ) );
 return 0;
}
```

Jak widać nie posiada on wywoływanej funkcji fiba(i), jednak będzie to funkcja napisana w asemblerze, a jej kod podaję poniżej:

```
  .type fiba, @function
  .globl fiba
fiba:  pushl %ebp
  movl %esp, %ebp
  subl $4, %esp
  movl 8(%ebp), %eax
  cmpl $0, %eax
  jz  f_0
  cmpl $1, %eax
  jz  f_1
  pushl %eax
  subl $2, %eax
  pushl %eax
```

```
  call fiba
  addl $4, %esp
  movl %eax, -4(%ebp)
  popl %eax
  dec %eax
  pushl %eax
  call fiba
  addl $4, %esp
  addl -4(%ebp), %eax
  jmp f_e
f_0:
  movl $0, %eax
  jmp f_e
f_1:
  movl $1, %eax
f_e:
  movl %ebp, %esp
  popl %ebp
  ret
```

Kod powienien być jasny, biorąc pod uwagę poprzednie laboratoria.

Uruchamianie

Aby uruchomić program trzeba najpierw dokonać kompilacji. Wykonamy ją poleceniem:

```
gcc -o fib fib_a.s fib.c
```

Oczywiście pliki fib_a.s oraz fib.c to są te, które przed chwilą napisalilśmy. Pora uruchomić program.

LABORATORIUM 7 - OPERACJE ZMIENNOPRZECINKOWE

Wstęp

Poznaliśmy możliwości korzystania z funkcji C w naszych programach. Pora jednak nieco oderwać się na chwilę od tego tematu i przyjrzeć jeszcze jednemu ważnemu działowi programowania w Asemblerze. Tym razem poznamy możliwości prowadzenia operacji na liczbach zmiennoprzecinkowych. Aby móc prowadzić obliczenia na liczbach tego typu wprowadzono specjalny koprocesor obliczeń zmiennoprzecinkowych, który wykonuje taki operacje zamiast podstawowego procesora.

Inicjalizacja koprocesora

Aby pracować na koprocesorze będziemy najpierw musieli go zainicjalizować. Wykonuje się to poleceniem:

```
finit # inicjalizacja koprocesora
```

Stos danych koprocesora

Ponieważ koprocesor operuje na zupełnie innym typie danych, niż dotąd było nam dane poznać, należy zapoznać się także z metodą przechowywania danych przez niego. Całość operacji przeprowadzanych przez koprocesor odbywa się na specjalnym stosie danych, który przechowuje liczby. Koprocesor posiad szereg instrukcji, które pozwalają na nim operować. Dla celów naszego laboratorium pokażemy jak wykonać kilka operacji. Oto lista tych, które będą użyte:

```
  FLDPI      # ładuje stałą PI na wierzch stosu [ST(0)]
  FADDP      # dodaje do siebie dwie liczby z wierzchu stosu [ST(0)+ST(1)]
       po czym zapisuje na pozycji ST(1), a usuwa pozycję ST(0)
       czyli usuwa argumenty i pozostawia na wierzchu tylko wynik
  FIMUL  ilosc    # przemnaża wiercholek przez liczbę całkowitą zachowaną
pod
           zmienną ilosc i nadpisuje
  FIDIV  liczba    # działa tak samo jak FIMUL, ale dzieli zamiast mnożyć
  FDIV  %ST(1), %ST(0) # podzieli st(1)/st(0)
  FADD  %ST(1), %ST(0) # doda do siebie st(1)+st(0)
  FSIN       # oblicza wartość sinusa od liczby zapisanej na wierzchołku
stosu
  FIADD  liczba    # dodaje do wierzchołka stosu liczbę całkowitą
  FISTP  y    # usuwa wartość z wierzchołka stosu i zapisuje w zmiennej y
  FILD  i    # załaduje liczbę całkowitą na stos [ST(0)]
  FLDL  x    # tak samo jak powyżej
  FSQRT      # liczy wartość pierwiastka od ST(0)
  FSIN      # liczy wartość sinusa od ST(0)
```

Warto także pamiętać, że korzystając ze stosu możemy używać jedynie 8 danych, gdyż tylko tyle on może pomieścić.

Program

Korzystając z takich aplikacji możemy stworzyć prosty program wyświetlający nam na ekranie sinusoidę pionowo (tak jest łatwiej rysować):

```
.equ  KERNEL,  0x80
.equ  WRITE,  0x04
.equ  EXIT,  0x01

.equ  STDOUT,  1

.equ  LAST_X,  80

  .data

line:
  .space    80, ' '
x:
  .long    0
y:
  .long    0
periods:
  .long    4
x_scale:
  .long    LAST_X - 5

y_shift:
  .long    25
y_scale:
  .long    20
startmsg:
  .ascii  "Plot of y=sin(x) function\n"
startlen:
  .long    ( . - startmsg )
finalmsg:
  .ascii  "End of plot\n"
finallen:
  .long    ( . - finalmsg )
newline:
  .string  "*\n"

  .text
  .global _start

_start:
  MOVL  $WRITE,%eax
  MOVL  $STDOUT,%ebx
  MOVL  $startmsg,%ecx
  MOVL  startlen,%edx
  INT  $KERNEL
  FINIT
  MOVL  $0,x
next:
    # F poczatkowe to jest zmiennoprzecinkowa
    # P na koncu oznacza zdjecie ze stosu
    # I oznacza liczbe calkowita
    # maksymalne zagniezdzenie w stosie to 8 (0-7) ST(0)-ST(7)
  FLDPI
  FLDPI
  FADDP
```

```
FIMUL  periods
FIMUL  x
FIDIV  x_scale
FSIN
FIMUL  y_scale
FIADD  y_shift
FISTP  y
NOP
MOVL  $WRITE,%eax
MOVL  $STDOUT,%ebx
MOVL  $line,%ecx
MOVL  y,%edx
INT  $KERNEL

MOVL  $WRITE,%eax
MOVL  $STDOUT,%ebx
MOVL  $newline,%ecx
MOVL  $2,%edx
INT   $KERNEL

INCL  x

CMPL  $LAST_X,x
JNZ  next

MOVL  $WRITE,%eax
MOVL  $STDOUT,%ebx
MOVL  $finalmsg,%ecx
MOVL  finallen,%edx
INT  $KERNEL

MOVL  $EXIT,%eax
MOVL  $0,%ebx
INT  $KERNEL
```

LABORATORIUM 8 - STOS I ZMIENNE ŚRODOWISKOWE

Wstęp

Język asemblera posiada wiele niuansów, które choć z początku mogą wydawać się nieistotne, to jednak po pewnym czasie okazują się ważne. Jednym z takich aspektów jest odwoływanie się do zmiennych środowiskowych systemu. Gdy program zostaje wywołany, na stosie umieszczane są zmienne środowiska (możemy je wypisać w systemie poleceniem env). Zmienne te przydają się niejednokrotnie, gdy chcemy skorzystać z pewnych spersonalizowanych ustawień użytkownika. Jak się jednak do nich odwołać?

Poruszanie się po stosie

Najpierw należy wyjasnić działanie stosu w programach, które piszemy. Nie raz już korzystaliśmy ze stosów, jednak mechanika ich działania nie była dotąd jasno wytłumaczona. Bez wnikania w zbędne szczegóły wyjaśniam od razu, że stos (którego struktura jest znana wszystkim) tworzony jest w pamięci przydzielonej programowi na końcu. Co z tego wynika? Gdy myślimy o stosie widzimy pamięć, której indeks zwiększa się wraz z każdą dodaną porcją informacji. Jednak tak na prawdę indeks stosu (przechowywany w rejestrze %esp) zmniejsza się za każdym razem, gdy dodamy dane, oraz zwiększa gdy je zdejmiemy. Osoby zainteresowane powodami, dla którch się tak dzieje, odsyłam do ostatniej części artykułu. Poruszanie się po stosie nie jest trudne. Wystarczy tylko odpowiednio modyfikować rejestr %esp lub też korzystać z pomocniczego rejestru %esi (z niego korzystamy najczęściej). Ponieważ %esp jest najważniejszy (modyfikują go instrukcje pop i push), powinniśmy raczej przepisywać jego zawartość do %esi i nim manipulować, lub też zapamiętywać gdzieś osobno %esp, nim go zmienimy.

Pobieranie danych z adresu na stosie

Załóżmy, że %esp wskazuje na punkt X w naszym stosie i jest to wierzch. Jeżeli będziemy chcieli pobrać jakieś dane znajdujące się bezpośrednio pod tymi danymi (i zakładamy że są wszystkie typu long) należy podać taką instrukcję:

```
movl 4(%esp),%eax # przeniesiemy dane niżej w stosie do %eax
```

Jak już wspomniałem, to co jest wyżej na stosie ma mniejszy indeks. Dlatego też odwołując się do danych niżej w stosie musimy użyć dodatniego offsetu, zamiast ujemnego. Użycie drugiego z wymienionych zwróciłoby nam wartość znajdującą się ponad punktem X (oczywiście gdyby było tam coś sensownego).

Warto zatem zauważyć, że gdy chcemy przesunąć wskaźnik %esi niżej w stosie, to modyfikujemy go następująco:

```
addl $4,%esi
subl $4,%esi # przesunie nas w górę stosu
```

Różnice między gcc a własnym linkowaniem

Trzeba teraz wyjaśnić, że istnieje zasadnicza różnica, między korzystaniem z gcc, a użyciem linkera systemu jakim jest program ld. Gdy użyjemy gcc, wymaga on by etykieta programu asemblera nazywała się *main*. Jak było już wyjasnione, dzieje się tak dlatego, że gcc tworzy program, który przetwarza wstępne dane (np. funkcje zegara systemowego), zanim uruchomi funkcję main (tak się dzieje właśnie w C). Powoduje to jednak problemy z dobraniem się do zmiennych środowiskowych. W C mamy do nich dostęp przez wskaźniki przekazywane jako argumenty funkcji main. W asemblerze chcemy dobrać się do nich bezpośrednio. Jednak będzie to bardzo trudne, gdyż używając gcc wiemy, że zanim uruchomione zostanie main, wywołane będą inne funkcje, które mogą przesunąć nam wskaźnik stosu %esp dalej i w ten sposób, nie będziemy w stanie odnaleźć zmiennych.

Jak zatem temu zapobiec? Zamiast korzystać z gcc skorzystamy ze standardowego systemowego linkera *ld*. Pozwala on nam na kompilowanie źródeł z dołączeniem standardowych bibliotek. Uzycie dla niego opcji *-lc* dołączy do programu bibliotekę *libc*, która zawiera podstawowe funkcje. Ponadto dokonamy dynamicznego połączenia biblioteki ld-linux.so.2 z naszym programem, by móc korzystać z funkcji libc.

```
as -o program.o program.s
ld -dynamic-linker /lib/ld-linux.so.2 -lc -o program program.o
```

Co nam to daje? Dzięki takiej metodzie tworzenia programu mamy dostęp do funkcji standardowej biblioteki C, równocześnie korzystając ze zwykłego programu asemblera. Program używa etykiety _start, a poruszając się po stosie w dół (na większe wartości wskaźnika) uzyskujemy dostęp do zmiennych środowiskowych. W ten sposób uzyskujemy porządany efekt. Poniżej podaję kod programu, który wypisze zmienne środowiskowe na ekranie. Należy go skompilować podaną wyżej metodą.

```
  .data
argc:
  .asciz "argc = %d\n"
args:
  .asciz "%s\n"
sep:
  .asciz "----------------------------\n"

  .text
  .global _start

_start:
  nop
  movl (%esp),%ecx
  pushl %ecx
  pushl $argc
  call printf
  addl $4,%esp
  popl %ecx
  movl %esp,%ebp
  addl $4, %ebp
next:
  pushl %ecx
  pushl (%ebp)
  pushl $args
  call printf
  addl $8,%esp
  popl %ecx
```

```
  addl $4,%ebp
  loop next
  nop
  pushl $sep
  call printf
  addl $4,%esp
  nop
  addl $4,%ebp
next1:
  cmpl $0,(%ebp)
  je finish
  pushl (%ebp)
  pushl $args
  call printf
  addl $8,%esp
  addl $4,%ebp
  jmp next1
finish:
  pushl $0
  call exit
```

Dlaczego stos działa jak działa?

Na koniec drobne wyjaśnienie dotyczące stosu. Było wcześniej już powiedziane, że stos rośnie w stronę numerów pamięci malejących. Powód jest bardzo prosty. Jeżeli system przydziela programowi pamięć, w ramach której może działać, musimy tak zagospodarować tę przestrzeń, by zmieścić kod programuy ze zmiennymi oraz obszar stosu. Jak zrobić to jednak na tyle optymalnie, by kod programu się mieścił zawsze, a równocześnie stos nie był ograniczony końcem obszaru pamięci? Jeżeli przydzielimy programowi obszar początkowy pamięci, a stosowi końcowy (ale będzie on rozrastał się w dół) wykorzystanie pamięci będzie optymalne. Możliwe będzie zabranie pamięci na rzecz programu (od lewej strony), ale także stos będzie mógł spokojnie rosnąć od prawej strony, jednak i wtedy ograniczeni jesteśmy momentem gdy stos spotka się z kodem programu. Mogłoby to szczególnie uszkodzić program. To jednak na razie nas nie martwi.