piątek/tydzień nieparzysty/10:30 Wrocław, dn. 15. marca 2012

Student:

Artur Zochniak, nr albumu 184725

Zapoznanie z operacjami arytmetycznymi mnożenia i dzielenia na platformie Linux/x86

sprawozdanie z laboratorium kursu „Architektura komputerów 2”

rok akademicki 2011/2012, kierunek INF

Prowadzący:

prof. dr hab. inż. Janusz Biernat

Contents

Cele laboratorium

Podczas zajęć należało zrealizować następujące czynności:

- napisanie programu zawierającego niezbędne minimum i skompilowanie, linkowanie i uruchomienie go,
- napisanie programu wykorzystującego następujące funkcje i zapoznanie się z ich działaniem obserwując zmiany w rejestrach przy pomocy debuggera:
    - `mul %reg` – funkcja mnożąca rejestr `%eax` przez `%reg` i zapisująca wynik, który należy interpretować jako `2^32*%edx+%eax`, dla dostatecznie małych liczb `%edx=0`,
    - `imul` – działanie analogiczne do `mul`, ale dla liczb zapisanych w U2,
    - `div %reg` – dzielenie liczby z rejestrów `%edx`,`%eax` (którą polecenie `div` rozumie jako liczbę o wartości `2^32*%edx+%eax`) przez liczbę w `%reg`, wyniki dzielenia: (iloraz, reszta) wędrują kolejno do (`%eax`, `%edx`),
    - idiv %reg – dzielenie liczb analogiczne do `div`, ale dla liczb zapisanych w U2,
- napisanie programu wczytującego ciąg znaków ze standardowego wejścia i wypisujące je na standardowe wyjście,
- zmodyfikowanie programu wczytującego ciąg znaków tak, aby na wyjściu zwracał wejście zakodowane szyfrem przesuwającym (szyfrem cezara).

Program zawierający niezbędne minimum, kompilacja i linkowanie

Każdy program powinien zawierać:

- informację o początku programu,
- powinien bezpiecznie się zakończyć wywołując przerwanie systemowe z parametrem `EXIT: %eax=1`
- ewentualnie sekcję `.data`, która zawiera informacje o zmiennych globalnych.

Kod źródłowy

Oto kod źródłowy programu, który po uruchomieniu zostanie bezpiecznie zakończony.

```
.section .data

.globl _start
_start:
movl $1, %eax
int $0x80
```

Kompilacja

kompilacja to proces zamiany kodu programu czytelnego dla człowieka na kod czytelny dla komputera, lecz nazwy użytych zmiennych, bibliotek i funkcji zostają weń zapisane, aby w procesie linkowania było możliwe połączenie odpowiednich części programu w całość.

Kompilacja może zostać wykonana poleceniem:

```
as plik_zrodlowy –o plik_wynikowy.o –g ––32
```

gdzie:

`–o nazwa` to przełącznik nazwy pliku wynikowego,

`-g` lub `––gstabs` to przełącznik umożliwiający wygodne debugowanie (dołączenie wszystkich symboli do programu)

`--32` to przełącznik wymuszający kompilator, aby architektura docelowa była 32-bitowa

Linkowanie

linkowanie to proces połączenia plików wynikowych kompilacji z bibliotekami statycznymi i innymi modułami programu.

Może zostać zrealizowana poleceniem:

```
ld plik_wynikowy_kompilacji[.o] –o plik_wykonywalny -m
elf_i386
```

gdzie:

`-o` nazwa to przełącznik pliku wynikowego (wykonywalnego)

`-m` to przełącznik umożliwiający zdefiniowanie trybu emulacji (wspierane to: `elf_x86_64 elf_i386 i386linux elf_l1om`)[1]

[1] źródło: `ld --help | grep "supported emulations"`

Mnożenie liczb naturalnych - funkcja mul

Program mnożący liczby naturalne

kod źrodłowy

```
EXIT=1
.data
czynnik1: .long 9
czynnik2: .long 111

.globl _start
_start:

mov $czynnik1, %eax
mov $czynnik2, %ebx
przedmnozeniem:
mul %ebx
pomnozeniu:
mov $EXIT, %eax
int $0x80
```

Kompilacja i linkowanie

```
as -o zadmul.o zadmul.a -g --32
ld zadmul.o -o zadmul -m elf_i386
```

Przedstawiam fragment zrzutu z działania debuggera `gdb>zadmul_gdb.txt`

```
Breakpoint 1, 0x08049067 in przedmnozeniem ()
eax            0x9  9
ecx            0x6f 111
edx            0x0  0
ebx            0x0  0
esp            0xffffdcd0     0xffffdcd0
ebp            0x0  0x0
esi            0x0  0
edi            0x0  0
eip            0x8049067 0x8049067 <przedmnozeniem>
eflags         0x202    [ IF ]
cs             0x23 35
ss             0x2b 43
ds             0x2b 43
es             0x2b 43
fs             0x0  0
gs             0x0  0
Single stepping until exit from function przedmnozeniem,
which has no line number information.

Breakpoint 2, 0x08049069 in pomnozeniu ()
eax            0x3e7    999
ecx            0x6f 111
edx            0x0  0
ebx            0x0  0
```

```
esp            0xffffdcd0     0xffffdcd0
ebp            0x0  0x0
esi            0x0  0
edi            0x0  0
eip            0x8049069 0x8049069 <pomnozeniu>
eflags         0x202    [ IF ]
cs             0x23 35
ss             0x2b 43
ds             0x2b 43
es             0x2b 43
fs             0x0  0
gs             0x0  0
```

Najistotniejsze elementy zostału **pogrubione**, można zauważyć, że do rejestru eax procesor wyprowadził wynik mnożenia 9*111.

Program mnożący duże czynnikinaturalne

kod źródłowy

```
EXIT=1
.data
czynnik1: .long 0x00000010
czynnik2: .long 0xffffffff

.globl _start
_start:

mov czynnik1, %eax
mov czynnik2, %ecx
przedmnozeniem:
mul %ecx
pomnozeniu:
mov $EXIT, %eax
int $0x80
```

Oczekiwanym wynikiem działania mnożenia powinna być liczba `czynnik2` przesunięta w lewo o jeden znak szesnastkowy (przesunięcie bitowe w lewo o 4 bity).

Kompilacja wykonując polecenie:

```
as -o zadmul.o zadmul.a -g –32 && ld zadmul.o -o zadmul -m
elf_i386
```

Uruchamanie

Program zwraca:

```
Breakpoint 1, 0x08049067 in przedmnozeniem ()
eax            0x10 16
ecx            0xffffffff     -1
edx            0x0  0
ebx            0x0  0
esp            0xffffdcd0     0xffffdcd0
ebp            0x0  0x0
esi            0x0  0
```

```
edi            0x0  0
eip            0x8049067 0x8049067 <przedmnozeniem>
eflags         0x202    [ IF ]
cs             0x23 35
ss             0x2b 43
ds             0x2b 43
es             0x2b 43
fs             0x0  0
gs             0x0  0
Single stepping until exit from function przedmnozeniem,
which has no line number information.

Breakpoint 2, 0x08049069 in pomnozeniu ()
eax            0xfffffff0     -16
ecx            0xffffffff     -1
edx            0xf  15
ebx            0x0  0
esp            0xffffdcd0     0xffffdcd0
ebp            0x0  0x0
esi            0x0  0
edi            0x0  0
eip            0x8049069 0x8049069 <pomnozeniu>
eflags         0xa03    [ CF IF OF ]
cs             0x23 35
ss             0x2b 43
ds             0x2b 43
es             0x2b 43
fs             0x0  0
gs             0x0  0
```

Jest to dowodem na to, że starsza część wyniku mnożenia dwóch liczb 32-bitowych wędruje do rejestru `edx`.

Mnożenie liczb całkowitych przy pomocy funkcji imul

Programu mnożacy liczby całkowite przeciętnej wielkości

kod źródłowy

```
EXIT=1
.data
czynnik1: .long 0x00000010 # 16 w u2
czynnik2: .long 0xffffffff #-1 w u2
.globl _start
_start:
mov czynnik1, %eax
mov czynnik2, %ecx
przedmnozeniem:
imul %ecx #oczekiwany wynik -16
pomnozeniu:
mov $EXIT, %eax
int $0x80
```

Kompilacja i linkowanie

```
NM=zadimul
as -o $NM.o $NM.a -g –32 && ld $NM.o -o $NM -m elf_i386
```

Zmienna `NM` to nazwa pliku źródłowego bez rozszerzenia.

Uruchamianie

W celu automatyzacji pracy w debuggerze zapiszę szereg komend do pliku autoinicjalizacji `.gdbinit`[2] debuggera gdb (skrypt ten jest wykonywany za każdym uruchomieniem debuggera z danej ścieżki)

```
echo -e "file ./zadimul\nbreak przedmnozeniem\nb
pomnozeniu\nrun\np \$eax\np \$edx\np \$ecx\np \$eflags\ns\np
\$eax\np \$edx\np \$ecx\np \$eflags\nquit">.gdbinit
cat .gdbinit
```

Mając już wypełniony plik `.gdbinit` można uruchomić debugger, który zatrzyma się na etykietach `przedmnozeniem` oraz `pomnozeniu` oraz wyświetli zawartość rejestrów po tych etykietach.

Po wykonaniu polecenia gdb>gdbimul.txt, w pliku gdbimul.txt pojawi się wynik debuggowania:

```
Breakpoint 1, 0x08049067 in przedmnozeniem ()
$1 = 16
$2 = 0
$3 = -1
$4 = [ IF ]
Single stepping until exit from function przedmnozeniem,
which has no line number information.

Breakpoint 2, 0x08049069 in pomnozeniu ()
$5 = -16
$6 = -1
$7 = -1
$8 = [ IF ]
```

wartości `($1, $2, $3, $4)` oraz `($4, $5, $6, $7)`, to kolejno (rejestr `%eax, %edx, %ecx, %eflags` – rejestr flag).

Można zauważyć, że `$1*$3=$5`, czyli wynik mnożenia `%eax*%ecx` powędrował do rejestru `%eax`, a rozszerzenie U2 (rozszerzenie ujemne – same jedynki) znajduje się również w rejestrze `%edx`.

[2] http://www.ibm.com/developerworks/aix/library/au-gdb.html

Program mnożący duże liczby całkowite

Kod źródłowy

programu zadimul.a

```
EXIT=1
.data
czynnik1: .long 0x7fffffff #duza liczba dodatnia
czynnik2: .long 0xf0000000 #duza liczba ujemna
.globl _start
_start:
mov czynnik1, %eax
mov czynnik2, %ecx
przedmnozeniem:
imul %ecx #oczekiwany wynik: duża liczba ujemna
pomnozeniu:
mov $EXIT, %eax
int $0x80
```

Kompilowanie i linkowanie

```
NM=zadimul
as -o $NM.o $NM.a -g --32
ld $NM.o -o $NM -m elf_i386
```

Uruchamianie programu

po wykonaniu polecenia `gdb>biginteger.txt`, w pliku tekstowym zostało zapisane:

```
Breakpoint 1, 0x08049067 in przedmnozeniem ()
$1 = 2147483647
$2 = 0
$3 = -268435456
$4 = [ IF ]
Single stepping until exit from function przedmnozeniem,
which has no line number information.

Breakpoint 2, 0x08049069 in pomnozeniu ()
$5 = 268435456
$6 = -134217728
$7 = -268435456
$8 = [ CF IF OF ]
```

Można zauważyć, że po wymnożeniu dwóch dużych liczb `($1=$eax)*($3=$eax)`, z czego jedna była ujemna, wynik to duża liczba ujemna, której wartość można wyrazić jako `(%edx=$1)2^32+%eax`.

Dodatkowo działanie funkcji imul skutkowało „zapaleniem” flag `CF` (ang. carriage flag) powodu wygenerowanego przeniesienia oraz `OF` (ang. overflow) – przekroczenie zakresu.

Dzielenie liczb naturalnych – funkcja div

Program dzielący liczby naturalne z resztą

Kod źródłowy

```
EXIT=1
.data
dzielna: .long 21
dzielnik: .long 7
.globl _start
_start:
mov dzielna, %eax
mov $0, %edx # unikniecie nieumyslnego zwiekszenia dzielnika
mov dzielnik, %ecx
przedmnozeniem:
idiv %ecx #zachodzi rownosc %%ecx*%eax+%edx*=%eax/%ecx
#wynik dzielenie w %eax
#reszta z dzielenia w %edx
pomnozeniu:
mov $EXIT, %eax
int $0x80
```

Kompilacja i linkowanie

```
NM=zaddiv && as -o $NM.o $NM.a -g --32 && ld $NM.o -o $NM -m
elf_i386
```

Uruchomienie

```
Breakpoint 1, 0x0804906c in przed ()
$1 = 22
$2 = 0
$3 = 7
$4 = [ IF ]
Single stepping until exit from function przed,
which has no line number information.

Breakpoint 2, 0x0804906e in po ()
$5 = 3
$6 = 1
$7 = 7
$8 = [ IF ]
```

Wynik dzielenia został zapisany w rejestrze `$5=%eax`,a reszta z dzielenia w rejestrze `$6=%edx`.

Dzielenie liczb całkowitych – funkcja idiv

Program dzielący liczby całkowite z resztą

Kod źródłowy

```
EXIT=1
.data
dzielna: .long 22
dzielnik: .long 7
.globl _start
_start:
mov dzielna, %eax
mov $0, %edx
mov dzielnik, %ecx
#zamiana ecx na u2 - start
not %ecx
inc %ecx
#zamiana ecx na u2 - koniec
#teraz w ecx jest -7
przed:
idiv %ecx #zachodzi rownosc %%ecx*%eax+%edx*=%eax/%ecx
#wynik dzielenie w %eax
#reszta z dzielenia w %edx
#oczekiwany wynik=22/-7=-3+R
po:
mov $EXIT, %eax
int $0x80
```

Kompilacja i linkowanie

```
NM=zadidiv && as -o $NM.o $NM.a -g --32 && ld $NM.o -o $NM -m
elf_i386
```

Uruchomienie

```
gdb ./zadidiv
b przed
b po
r
info registers
s
info registers
quit
```

Wynik działania:

```
(gdb) b przed
Breakpoint 1 at 0x804906f
(gdb) b po
Breakpoint 2 at 0x8049071
```

```
(gdb) r
Starting program: /home/susers/s184725/zaddiv

Breakpoint 1, 0x0804906f in przed ()
(gdb) info registers
eax            0x16 22
ecx            0xfffffff9     -7
edx            0x0  0
ebx            0x0  0
esp            0xffffdcd0     0xffffdcd0
ebp            0x0  0x0
esi            0x0  0
edi            0x0  0
eip            0x804906f 0x804906f <przed>
eflags         0x286    [ PF SF IF ]
cs             0x23 35
ss             0x2b 43
ds             0x2b 43
es             0x2b 43
fs             0x0  0
gs             0x0  0
(gdb) s
Single stepping until exit from function przed,
which has no line number information.

Breakpoint 2, 0x08049071 in po ()
(gdb) info registers
eax            0xfffffffd     -3
ecx            0xfffffff9     -7
edx            0x1  1
ebx            0x0  0
esp            0xffffdcd0     0xffffdcd0
ebp            0x0  0x0
esi            0x0  0
edi            0x0  0
eip            0x8049071 0x8049071 <po>
eflags         0x286    [ PF SF IF ]
cs             0x23 35
ss             0x2b 43
ds             0x2b 43
es             0x2b 43
fs             0x0  0
gs             0x0  0
```

Wyjście z debuggera dowodzi tego, że wynik z dzielenia `(%eax=22)/(%ecx=-7)` powędrował do: część całkowita do `%eax=-3`, reszta z dzielenia do `%edx=1`, równanie `22/-7=7*(-3)+1` jest prawdziwe.

Program wczytujący wejście ze standardowego wejścia i wypisujące je na standardowe wyjście

Program będzie korzystał z systemu przerwań procesora, aby wczytywać i wyświetlać dane.

```
Kod programu EXIT=1
SYSC=0x80
STDIN = 0
STDOUT = 1
READ=3
WRITE=4

.section .data

BUF: .ascii "            \n"
BUF_SIZE = .-BUF

.globl _start
_start:
wczytaj:
movl $READ, %eax
movl $STDIN, %ebx
movl $BUF, %ecx
movl $BUF_SIZE, %edx
int $SYSC

wypisz:
movl $WRITE, %eax
movl $STDOUT, %ebx
movl $BUF, %ecx
movl $BUF_SIZE, %edx
int $SYSC

.globl exiting
exiting:
movl $EXIT, %eax
int $SYSC
```

Program jedynie wczytuje i wypisuje tekst na ekran

Kompilacja i linkowanie

```
NM=readwrite && as -o $NM.o $NM.a -g --32 && ld $NM.o -o $NM -
m elf_i386
```

Uruchomienie

```
./rewrite
Test programu
Test programu
```

Program zamieniający litery na przeciwną wielkość (małe na duże i odwrotnie)

Kod programu

```
EXIT=1
SYSC=0x80
STDIN = 0
STDOUT = 1
READ=3
WRITE=4

.section .data

BUF: .ascii "            \n"
BUF_SIZE = .-BUF

.globl _start
_start:

movl $READ, %eax
movl $STDIN, %ebx
movl $BUF, %ecx
movl $BUF_SIZE, %edx
int $SYSC

movl $BUF_SIZE, %ecx

mov $0, %ebx
swapLetterCase:
movb BUF(,%ebx,1),%dl
cmpb $10, %dl
je wypisuj
movb %dl,%al
zmiana:
or $0x20,%al
cmpb $'a', %al
jl skip
cmpb $'z', %al
jg skip

letter:
xor $0x20,%dl
movb %dl, BUF(,%ebx,1)
skip:
inc %ebx
loop swapLetterCase

movl $1, %ecx
```

```
wypisuj:
movl $WRITE, %eax
movl $STDOUT, %ebx
push %ecx
movl $BUF, %ecx
movl $BUF_SIZE, %edx
int $SYSC
pop %ecx
.globl exiting
exiting:
movl $EXIT, %eax
int $SYSC
```

Program najpierw wczytuje ciąg znaków ze standardowego wejścia, następnie zamienia wielkość liter (wykrywa czy znak jest literą – jeśli tak, to zmienia bit o wadze 2^4 na 1 – małe litery). Jeśli znak to litera, to zamienia jej wielkość na przeciwną. Następnie wypisuje ciąg zamienionych znaków na wyjście.

Kompilacja i linkowanie

```
NM=changecase && as -o $NM.o $NM.a -g --32 && ld $NM.o -o $NM
-m elf_i386
```

Uruchomienie

```
./zad4
TeSt pRoGrAmU
tEsT PrOgRaMu
```

Podsumowanie

- Należy pamiętać o tym, że w Assemblerze funkcje mnożące i dzielące przyjmują (lub zwracają) wynik podwójnego słowa maszynowego (na procesorze 32-bitowym wynikiem może być liczba 64-bitowa). Zapomnienie o tym fakcie (np. niewyzerowanie rejestru `%edx` przed dzieleniem malej liczby) może doprowadzić do sztucznego zwiększenia wyniku o bardzo dużą liczbę.
- Podczas pisania programów bardzo użytecznym narzędziem jest debugger gdb, szczególnie ze względu na pułapki `break nazwa_etkiety`, praca krokowa `s[tep]`, możliwość wypisywania aktualnej wartości rejestru `p/ht $reg` [tylko jedno: `t` – binarnie lub `h` – szesnastkowo], `display/c $reg` – wypisywanie aktualnego znaku znajdującego się w `%reg` po każdym zatrzymaniu programu.