piątek/tydzień nieparzysty/10^{15} Wrocław, 20. kwietnia 2012

Artur Zochniak, nr albumu 184725

Sprawozdanie
z laboratorium kursu „Architektura komputerów 2”
rok akademicki 2011/2012, kierunek INF

(2. marca 2012) funkcje arytmetyczne,
(16. marca 2012) funkcje rekurencyjne,
(30. marca 2012) łączenie C z ASM

Prowadzący:

prof. dr hab. inż. Janusz Biernat

Spis treści

2. marca 2012: Zapoznanie z operacjami arytmetycznymi mnożenia i dzielenia na platformie Linux/x86

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 źródłowy

```
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 czynniki naturalne

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
```

Uruchamianie

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 przed mnozeniem,
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 mnożący 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`.

Program mnożący duże liczby całkowite

Kod źródłowy

programu zadimul.a

```
EXIT=1
.data
```

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

```
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.

Bibliografia

1) http://www.zak.ict.pwr.wroc.pl/materialy/architektura/laboratorium/Programowanie/Linux-asm-lab-07-03-2012.pdf - używanie gdb strona 9.

16. marca 2012: Implementacja funkcji rekurencyjnych w Assemblerze dla platformy Linux/x86

Cele laboratorium

Celem labolatorium było:

- napisanie prostej funkcji rekurencyjnej w celu zapoznania się z tematem,
- napisanie funkcji rekurencyjnej wykorzystującej.

Podstawy wywoływania funkcji

Miejsce argumentów wywołania funkcji

Każde wywołanie funkcji poprzedzone jest przekazaniem jej parametrów. Parametry można przekazywać rejestrami (tak jak np. realizowane przy obsłudze przerwań - `int`) lub umieszczać na stosie (`push $4` jeśli argumentem jest liczba 4).
Funkcja rekurencyjna może działać poprawnie tylko jeśli ma indywidualny dla każdego wywołania kontekst. Kontekst ten można rozumieć jak zbiór zmiennych lokalnych, które parametryzują dane wywołanie funkcji, z tego powodu, aby wywołać, funkcję rekurencyjną należy przekazywać jej argumenty przez stos, w przeciwnym wypadku wszystkie wywołania funkcji będą pracować w tym samym kontekście.

Opis realizacji

Wywołując funkcję poleceniem `call fnc` procesor otrzymuje polecenie, aby z aktualnie przetwarzanego miejsca przeskoczył do miejsca w pamięci fnc. Jednak nie jest ważne tylko gdzie skierujemy procesor, ale, równie bardzo ważną rzeczą jest to, czy procesor będzie wiedział gdzie ma wrócić.

Umożliwia to zachowanie tzw. *śladu skoku*, czyli miejsca, do którego funkcja, po zakończeniu wykonywania się powinna wrócić. Za każdym razem, przed wywoływaniem polecenia `call`, procesor umieszcza na stosie wartość rejestru `%EIP` (natomiast polecenie `ret` zdejmuje te wartość ze stosu) oraz otrzymuje informację o następnnej instrukcji programu, która zostałaby wykonana, gdyby nie wywołano funkcji, i zapisuje ją w rejestrze bazowym `%ebp`, `%bp` (ang. *(extended) base pointer*).

Przez czas, kiedy zwykład (nie rekurencyjna) funkcja jest wykonywana, rejestr `%ebp` nie jest modyfikowany, ale w przypadku funkcji rekurencyjnej należy liczyć się z tym, że jeszcze raz zostanie wywołana funkcja, a to wiązałoby się z utratą informacji zapisanej w `%ebp` (powędrowałby tam adres fragmentu ciała wykonywanej funkcji), aby uniknąć utraty informacji o miejscu wywołującym funkcję należy zapisać ślad skoku (rejestr `%ebp`) w miejscu, które umożliwiłoby przechowywanie indywidualnej informacji o każdorazowym wykonaniu funkcji rekurencyjnej. Takim miejscem jest stos.

Konwencją jest to, że pierwszym krokiem po wywołaniu funkcji jest zapisanie kopii rejestru wskaźnika bazy (`%ebp`) na stos i zaraz przed wyjściem z funkcji wczytanie go z tego samego miejsca, do którego został skopiowany (realizowane jest to poleceniami `push %ebp` oraz `pop %ebp`).

Odczytywanie argumentów funkcji w jej ciele

Pozostaje jeszcze jeden ważny aspekt. Łatwość odczytywania argumentów funkcji.

Stos budowany jest „w tył”, tj. im później informacja zostałą do niego dodana, tym będzie się znajdowała w pamięci z adresem o mniejszej wartości.

Mechanizm działania funkcji

Zakładając, że program skompilowany jest dla architektury 32-bitowej, można wywnioskować, że długość pojedyńczego słowa to 4 bajty.

1. Na stos wędruje 4-bajtowa wartość wskaźnika %EIP,
2. następnie (zgodnie z konwencją) umieszczamy na stosie wartość rejestru bazowego `%ebp`,
3. aktualna wartość stosu znajduje się w rejestrze `%esp` (ang. *extended stack pointer*), jest to również informacja o kontekście wywołania funkcji (umożliwia odczytywanie argumentów wywołania). Konwencja mówi, że wartość tego rejestru możemy skopiować do rejestru `%ebp` (którego wartość przed chwilą skopiowano na stos) i używać rejestru `%ebp` jako bazy dla kontekstu wywołania funkcji. np. aby dostać się do pierwszego argumentu w ciele funkcji nie trzeba dbać o to, czy modyfikowano stos, można po prostu skorzystać z notacji `8(%ebp)` dla pierwszego 4-bajtowego argumentu funkcji, `12(%ebp)` dla drugiego 4-bajtowego argumentu funkcji, itd.
4. należy pamiętać, aby przywrócić stan stosu do momentu wywoływania funkcji, realizuje się to wywołaniem `movl %ebp, %esp`. Po wywołaniu tej instrukcji

możemy być pewni, że pierwszą wartością na stosie jest wartość rejestru bazowego `%ebp`, następnie wartość rejestru `%EIP` umieszczona na stosie przez polecenie `call`.

5. Umieszczamy ponownie informację o śladzie skoku w rejestrze `%ebp` wywołując `pop %ebp`.
6. Poleceniem `ret` wędrujemy do następnej instrukcji po wywołaniu funkcji i kontynuujemy pracę.
7. Wynik działania funkcji najczęściej wędruje do rejestru `%eax`.

Funkcja rekurencyjna: suma liczb od 1 do n

```
1.  .type sum @function
2.  sum:
3.    push %ebp
4.    mov %esp, %ebp
5.  function_body:
6.    mov 8(%ebp), %ebx
7.    mov $1, %eax
8.    cmp $1, %ebx
9.    jbe finish_sum
10.   dec %ebx
11.   push %ebx
12.   call sum
13.   inc %ebx
14.   add %ebx, %eax
15. finish_sum:
16.   mov %ebp, %esp
17.   pop %ebp
18.   ret
```

Dane wejściowe:

n sumę ile pierwszych liczb naturalnych obliczyć

Algorytm:

Funkcja zwraca sumę [`linia 14.`] *aktualne przekazanego argumentu* oraz *wartość funkcji wywołanej ze swoim argumentem wywołania pomniejszonym o jeden.*

Jeśli argumentem funkcji jest liczba mniejsza lub równa 1 [`linia 8.`], to funkcja zwraca 1 (`linia 7.`).

Przykład: program uruchomiony z argumentem działa następująco:

sum(10) = 10+sum(10-1)=10+9+sum(8) = … = 10+9+…+3+sum(1) = 55,

co jest zgodne z prawdą, gdyż: *10*(10+1)/2=55.*

Dane wyjściowe:

suma liczb od 1 do n.

Funkcja rekurencyjna generująca n-tą liczbę Fibonacciego

```
1. generateNthFibbNr=6
2. EXIT=1
3. SYSC=0x80
4. STDIN = 0
5. STDOUT = 1
6. READ=3
7. WRITE=4
8. .section .data
9. .globl _start
10..globl exiting
11..type fibb @function
12.fibb:
13.  push %ebp
14.  mov %esp, %ebp
15.function_body:
16.  push %ebx # (A) function is using ebx as temporary register
17.  mov 8(%ebp), %ebx
18.  mov $1, %eax
19.  cmp $2, %ebx #first two fibb numbers are 1's
20.  jbe finish
21.started:
22.  dec %ebx
23.  push %ebx
24.  call fibb # now f(n-1) is in eax
25.  push %eax # (B) f(n-1) is on stack
26.  dec %ebx
27.  push %ebx
28.  call fibb #f(n-2) is in %eax
29.  pop %ebx # recovering f(n-1) from (B)
30.  add %ebx, %eax
31.  pop %ebx #neutralizing (A)
32.finish:
33.  mov %ebp, %esp
34.  pop %ebp
35.  ret
36._start:
37.  nop
38.  pushl $generateNthFibbNr
39.  call fibb
40.  result:
41.  mov %eax, %ebx
42.exiting:
43.  movl $EXIT, %eax
44.  int $SYSC
```

Dane wejściowe:

n – którą liczbę fibonacciego obliczyć.

Algorytm:

parametr wywołania funkcji jest zmniejszany o 1 [`linia 22.`] i otrzymana wartość staje się argumentem rekurencyjnego wywołania [`linia 23.`] – w wyniku otrzymujemy w `linii 25. n-1`-ty wyraz ciągu fibonacciego. Wynik znajduje się w rejestrze `%eax`, ale następne wywołanie funkcji zmieni ten rejestr, zatem wynik jest kopiowany na stos [`linia`

`25.`]. Kolejna dekrementacja [`linia 27.`] i wywołanie funkcji [`linia 28.`] – w wyniku czego obliczony zostaje `n-2`-gi wyraz ciągu fibonacciego. wynik znajduje się w eax. Wczytanie poprzedniego wyrazu ciagu ze stosu [`linia 29.`] oraz zsumowanie dwóch poprzednich wyrazów ciągu [`linia 30.`].

$$fibb(n)=fibb(n-1)+fibb(n-2)$$

Dane wyjściowe:

n-ta liczba fibonacciego

Podsumowanie

Przy implementowaniu funkcji rekurencyjnych należy pamiętać o tym, aby gromadzić kolejne informacje o *śladach skoków*.

Korzystanie z rejestru bazowego `%EBP` zawierającego informacje o kontekście wywołania funkcji pomaga w odnalezieniu (indeksowaniu) argumentów funkcji.

Wiedza zdobyta dzięki laboratorium pozwala zrozumieć jak to możliwe, że pomimo tego, że w jęzku C++ nie ma możliwości odczytania argumentu dowolnego numeru, to dzięki skorzystaniu z Assemblera można zaimplementować np. funkcję z biblioteki standardowej wejścia/wyjścia `<stdio.h> printf()`.

Bibliografia

2) „Programming from the Ground Up” http://download.savannah.gnu.org/releases/pgubook/ - rozdział 4. „all about functions” [od str. 33]
3) http://www.zak.ict.pwr.wroc.pl/materialy/architektura/laboratorium/Debugging%20with%20GDB.pdf - instrukcja gdb
4) http://www.zak.ict.pwr.wroc.pl/materialy/architektura/laboratorium/Programowanie/Linux-asm-lab-07-03-2012.pdf - używanie gdb strona 9.

30. marca 2012: Łączenie kodu programu napisanego w języku c/c++ z kodem assemblera na platformie Linux/x86

Cele laboratorium

Celem laboratorium było:

- napisanie programu w assemblerze i odwołanie się w nim do funkcji napisanej w C/C++,
- napisanie programu w C/C++ i użycie weń funkcji napisanej w assemblerze,
- napisanie programu w assemblerze i odwołanie się w nim do funkcji zdefiniowanej w standardzie C/C++.

Wstęp

Kompilacja to proces zamiany kodu czytelnego dla człowieka na kod czytelny dla komputera. Zarówno programy napisane w języku C lub C++, jak i programy napisane w assemblerze podlegają kompilacji, która umożliwia późniejsze linkowanie. Wszystko to po to, aby program mógł zostać uruchomiony na tej samej maszynie.
Musi zatem istnieć sposób, aby połączyć kody programów napisanych, fragmentami, w dwóch językach. Na tworzenie programu składają się: kompilacja i linkowanie.

Łączenie plików wynikowych

W wyniku kompilacji, np. poleceniem `gcc` lub `c++` czy `as`, powstają pliki wynikowe. Wynikiem kompilacji jest plik zawierający kod programu, lecz w formie przystępnej dla komputera. Rezultatem kompilacji jest plik kodu maszynowego oraz zbioru powiązań, które są wymagane, aby program działał prawidłowo.
Połączenie programu napisanego w języku C/C++ z programem napisanym w assemblerze jest możliwe po otrzymaniu plików wynikowych. Następny etap – linkowanie – łączy oba programy w spójną całość, dołączając również wymagane statyczne importy.

Wywołanie w asm funkcji napisanej w C++

Kompilator C/C++ zajmuje się zarządzaniem stosem (tj. jeśli zostaje zdeklarowana zmienna lokalna, to programista nie musi się martwić o to, aby zmniejszać wartość wskaźnika stosu

`%esp`, lecz pracuje na wyższym poziomie abstrakcji, mając dzięki temu więcej czasu na tworzenie algorytmu.
Jeśli jednak te operacje są automatyzowane, to musi istnieć usystematyzowany sposób przekazywania argumentów do funkcji.
Funkcja skompilowana w C++ oczekuje, że argumenty będą umieszczone na stosie począwszy od argumentu o największym indeksie kończąc na pierwszym.

Plik defineAdd.cpp

```
#include <stdio.h>
#include <stdlib.h>

int add(int a, int b)
{
     printf("wynik=%d+%d=%d\n",a,b,a+b);
     return a+b;
}
```

Plik callAdd.asm

W wywołaniu funkcji w assemblerze argumenty wędrują na stos i wywoływana jest funkcja. Funkcja dodawania sama w sobie wyświetla wynik.

```
.section .data

.globl main
.extern add
main:
     push $2
     push $3
     call add
     mov $1, %eax
     mov $0, %ebx
     int $0x80
```

Program można skompilować poleceniem `Make prog` po uprzednim stworzeniu pliku `Makefile` zawierającego:

```
asm.o:
     as -o asm.o callAdd.asm --32
cpp.o:
     gcc -m32 -o cpp.o defineAdd.c -c
prog: asm.o cpp.o
     gcc -m32 -o prog cpp.o asm.o
run:
     ./prog
clean:
     rm *.o && rm ./prog
```

Po uruchomieniu programu można zaobserwować przebieg obsługiwania argumentów przez funkcji - pierwszy argument to ten, który został umieszczony na stosie jako ostatni.

```
s184725@lak:~/03lab/call_my_cpp_function_from_asm_code$ ./prog
wynik=3+2=5
```

Wywołanie w C++ funkcji napisanej w asm

Każda funkcja w języku C/C++ musi zostać (co najmniej) zadeklarowana. Korzystając z funkcji zdefiniowanej w assemblerze z pomocą przychodzi słowo kluczowe extern. Oznacza ono dla kompilatora tyle, że funkcja (lub zmienna), przy której zostało ono użyte, na pewno będzie już znana w procesie linkowania, jednakże teraz (w procesie kompilacji) jeszcze jej nie znamy. Dodatkowo, każda funkcja w C/C++ musi mieć swój jasno sprecyzowany typ, dlatego niebywale istotne jest, aby funkcja napisana w assemblerze była kompatybilna z deklaracją funkcji w C/C++ (i vice versa).
Funkcja napisana w asm to funkcja dodająca dwie liczby. Przyjmuje ona jako argumenty 2 liczby 32-bitowe w U2 (`int`), a zwraca ich sumę w postaci liczby dodatniej. Deklaracja funkcji w C/C++ wygląda następująco:

```
extern int dodaj_liczby(int a, int b);
```

W momencie implementacji funkcji w asm należy pamiętać, że na stosie będą znajdować się dwie liczby przeznaczone do wykorzystania w tej funkcji, a język C/C++ będzie oczekiwał otrzymania wyniku w rejestrze akumulatora `%eax`.

Plik `cpp.cpp`:

```
#include <stdio.h>
#include <stdlib.h>

extern int dodaj_liczby(int a, int b);
int main(int argc, char *argv[])
{
  int a=2;
  int b=18;
  printf("wynik funkcji zdefiniowaniej w asm wywolanej z
poziomu c=%d\noczekiwany wynik to %d\n"
  ,dodaj_liczby(a,b), a+b);
     return 0;
}
```

Plik z kodem źródłowym assemblera jedyne co robi to zdejmuje argumenty ze stosu [linie oznaczone przez `(A)`] oraz je dodaje, a wynik przechowuje w rejestrze `%eax` [linia oznaczona w kodzie jako `(B)`].

Plik asm.asm**:**

```
.section .data

.globl dodaj_liczby

```

```
.type dodaj_liczby @function
dodaj_liczby:
     push %ebp
     mov %esp, %ebp
     mov 8(%esp),%eax (A)
     mov 12(%esp),%ecx (A)
     add %ecx,%eax #(B)
     mov %ebp,%esp
     pop %ebp
     ret
```

Program można skompilować poleceniem

`Make prog`

Po uprzednim stworzeniu pliku Makefile zawierającym:

```
prog: asm.o cpp.o
        gcc -m32 -g -o prog cpp.o asm.o
cpp.o:
        gcc -m32 -g -x c -o cpp.o call.c -c
asm.o:
        gcc -m32 -g -x assembler -o asm.o asm.asm -c
clean:
        rm *.o && rm ./prog
```

Wynik działania programu wygląda następująco:

```
s184725@lak:~/03lab/call_asm_from_c_32bit$ ./prog
wynik funkcji zdefiniowaniej w asm wywolanej z poziomu c=20
oczekiwany wynik to 20
```

Wywołanie w asm funkcji zdefiniowanej jako standardowa w C/C++

Linker `g++` wykorzystywany również do kompilowania i linkowania doskonale daje sobie radę z odnalezieniem definicji funkcji standardowych.

Celem programu będzie uruchomienie funkcji `printf` wyświetlającej liczbę podaną przez użytkownika (dzięki funkcji `scanf`).

```
.section .data
read:     .ascii "%u\0"
write:    .ascii "%u\n\0"

.global main
.extern printf
.extern scanf
main:
     pushl %ebp
     movl %esp, %ebp
     pushl $n #ostatni argument funkcji (w postaci adresu)
     pushl $read #podanie pierwszego argumentu funkcji
before:
     call scanf
done:
     pushl n #ostatni argument printf (w postaci wartości)
     pushl $write #ciąg znaków formatujący wywołanie printf
     call printf
     movl %ebp, %esp
     popl %ebp
     mov $0, %eax
ret
```

Między etykietami before i done wywoływana jest funkcja scanf, której wywołanie jest równoznaczne z wywołaniem zapisanym w języku C/C++ jako: `scanf((char*)read, &n);`. Argumenty w assemblerze, zgodnie z konwencją. Następnie wywołujemy funkcję printf w sposób, który w C/C++ zostałby zapisany jako: `printf((const char*)write, n);`.

`([const ]char*)` informuje w tym przypadku tylko o tym, że oczekiwany argument funkcji to wskaźnik na ciąg znaków.

Program można skompilować poleceniem `make prog` po uprzednim utworzeniu pliku `Makefile` zawierającego

```
asm.o:
     gcc -g -m32 -x assembler -o asm.o call.asm -c
#    as -o asm.o call.asm --32 #alternatywne rozwiazanie
prog: asm.o
     gcc -g -m32 -o prog asm.o
run: prog
     ./prog
clean:
     rm *.o && rm ./prog
debug: prog
     gdb ./prog
```

Podsumowanie

Łączenie kodu napisanego w C/C++ oraz assemblerze jest możliwe dzięki dwuetapowym tworzeniu programu: kompilacja i linkowanie. Kompilacja pozwala na sprowadzenie obu fragmentów kodu do „wspólnego mianownika”, a linkowanie umożliwia połączenie tych części w jedną całość, dodając przy tym wszystkie potrzebne powiązania (statycznie).

Bibliografia

1) http://pl.wikibooks.org/wiki/Asembler_x86
2) http://www.osdev.org/howtos/1/#C użycie parametru `-c` kompilatora `g++`.
3) „Programming from the Ground Up” http://download.savannah.gnu.org/releases/pgubook/