



«Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ

ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ

КАФЕДРА

КОМПЬЮТЕРНЫЕ СИСТЕМЫ И СЕТИ

О Т Ч Е Т

по лабораторной работе № 1

Дисциплина: Машинно-зависимые языки и основы компиляции

Название: Изучение среды и отладчика ассемблера

Студент

ИУ6-42Б

(Группа)

17.02.24

(Подпись, дата)

А. П. Плютто

(И. О. Фамилия)

Преподаватель

17.02.24

(Подпись, дата)

(И. О. Фамилия)

Москва 2024 г.

Цель работы

1

Выполнение

1

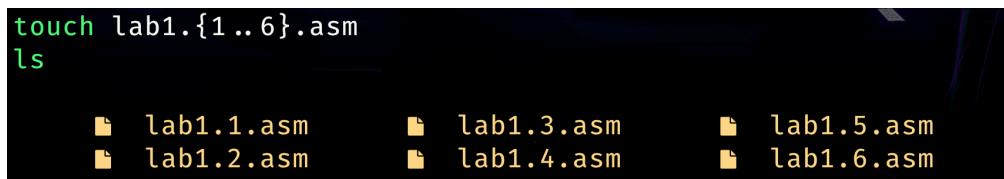
Создание папок и файлов

При работе с операционной системой `linux` удобно использовать для большинства действий терминал, поэтому я буду использовать только его. Редактор кода `neovim`, дебагер `gdb`.

Создадим папку и файлы для 1 лабораторной работы. Для этого воспользуемся утилитами `mkdir`, `cd`, `touch` и выведем все файлы в папке на экран с помощью `ls`.

```
mkdir -r ~/labs/lab1
cd ~/labs/lab1
touch lab1.{1..6}.asm
ls
```

Листинг 1 — Создаем папки и файлы



```
touch lab1.{1..6}.asm
ls

lab1.1.asm  lab1.3.asm  lab1.5.asm
lab1.2.asm  lab1.4.asm  lab1.6.asm
```

A screenshot of a terminal window. The user has run the command 'ls' after creating files. The output shows six files named lab1.1.asm through lab1.6.asm, each represented by a small yellow folder icon.

Рисунок 1 — Вывод команды `ls`

Заготовка программы

Возьмем заготовку программы для 32 битной архитектуры.

```
section .data          ; сегмент инициализированных переменных
; выводимое сообщение
ExitMsg db "Press Enter to Exit",10
lenExit equ $-ExitMsg

section .bss          ; сегмент неинициализированных переменных
InBuf resb 10         ; буфер для вводимой строки
lenIn equ $-InBuf

section .text          ; сегмент кода
global _start

_start:
; write
    mov eax, 4      ; системная функция 4 (write)
    mov ebx, 1      ; дескриптор файла stdout=1
    mov ecx, ExitMsg ; адрес выводимой строки
    mov edx, lenExit ; длина выводимой строки
    int 80h          ; вызов системной функции

; read
    mov eax, 3      ; системная функция 3 (read)
    mov ebx, 0      ; дескриптор файла stdin=0
    mov ecx, InBuf   ; адрес буфера ввода
    mov edx, lenIn   ; размер буфера
    int 80h          ; вызов системной функции

; exit
    mov eax, 1      ; системная функция 1 (exit)
    xor ebx, ebx     ; код возврата 0
    int 80h          ; вызов системной функции
```

Листинг 2 — Заготовка программы

Программу скомпилируем и скомпаниуем при помощи компилятора `nasm` и копановщика `ld`. Так как процессор у меня 64-ех разрядный, а программа написана для 32-ух разрядного процессора укажем компоновщику, что нужно использовать режим эмуляции: `ld -m elf_i386`.

И так вся команда для сборки бинарного файла будет выглядеть так:

```
mod=lab1.1 # Название ассемблерного файла без расширения  
nasm -f elf -o $mod.o $mod.asm  
ld -m elf_i386 -o $mod $mod.o
```

Листинг 3 — Команда для создания бинарного файла

После этого программу можно запустить при помощи команды `./$mod`.



```
lab1 » ./lab1.1  
Press Enter to Exit
```

Рисунок 2 — Вывод программы I

Разберем основные идеи в программе.

Она разделена на 3 подпрограммы: вывод, ввод, выход. Между подпрограммами используется механизм прерываний, который выполняет системные функции указанные в регистре `eax`. Следует отметить, что сейчас все прерывания обрабатываются программно и доступ к действительным аппаратным прерываниям современные операционные системы не предоставляют.

Для вывода используем системную функцию 4. Для этого помещаем значение 4 в `eax`. В регистр `ebx` помещаем декриптор файла `stdout`. Дескрипторы файлов `stdin`, `stdout` и `stderr` – 0, 1 и 2, соответственно. В `ecx` указываем ссылку на первый элемент массива с нашей строкой (которая в памяти представлена как набор чисел). В `edx` помещаем размер строки.

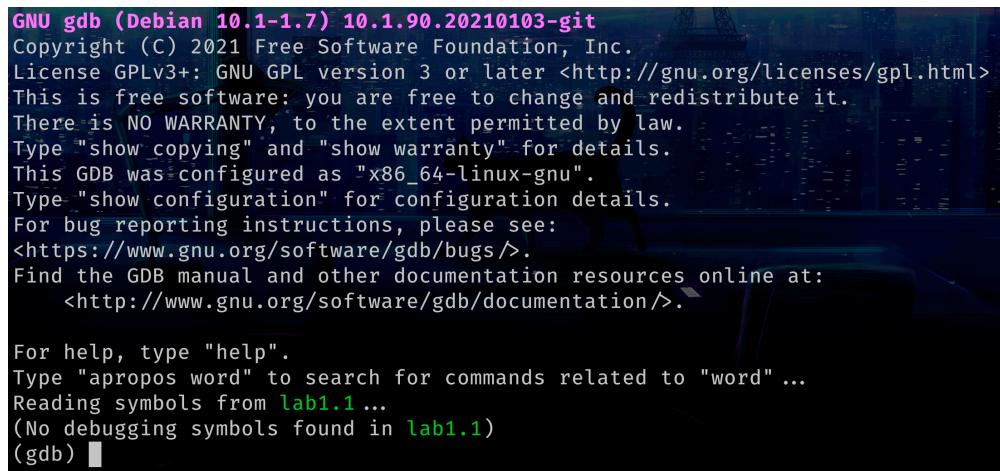
Тут следует немного подробнее остановиться на том как мы получаем этот размер строки, объявленный в `section .data`. Знак `$` воспринимается в диалекте `nasm` как адрес текущей ячейки памяти, т.е. если использовать знак `$` после объявления массива то мы получим адрес, который на единицу превосходит адрес последнего элемента массива. Вычитая из полученного адреса адрес начала массива мы получаем длину массива, а так как эти операции были проделаны не с литералами, а с адресами мы используем `equ`. По такому же принципу мы получаем и `lenIn`.

Для чтения выделим отдельно неинициализированную память для 10 букв(после ввода 11 ввод перенаправится в командную строку по завершении

программы). Используем системную функцию 3, файл `stdin`, выделенную память и ее длину для регистров `eax`, `ebx`, `ecx` и `edx` соответственно.

Для выхода используем системную функцию 0, возвращаем код 0, чтобы показать, что программа успешно завершилась.

Посмотрим как выглядит машинное представление программы, ее дизассемблированный код, содержимое регистров в отладчике. Для этого запустим отладку с помощью команды `gdb $mod`.



```
GNU gdb (Debian 10.1-1.7) 10.1.90.20210103-git
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word" ...
Reading symbols from lab1.1 ...
(No debugging symbols found in lab1.1)
(gdb) █
```

Рисунок 3 — Интерфейс `gdb`

Создадим новый «слой» в котором сверху будут регистры, а снизу ассемблерный код, созданный на основе полученного бинарного кода. Сам машинный код это набор чисел, так что он совершенно не нагляден для отладки.

```
tui new-layout ra regs 1 asm 1 cmd 1
```

После создания слоя прейдем в него с помощью команды `lay ra`. Тут создадим точку останова на метке начала нашей программы `br start`.

Перед запуском убедимся что в память загрузилось все, что мы объявили в `section .data`.



```
(gdb) x/10x 0x804a000
0x804a000: 0x73657250      0x6e452073      0x20726574      0x45206f74
0x804a010: 0x00746978      0x00000000      0x00000000      0x00000000
0x804a020: 0x00000000      0x00000000
(gdb) █
```

Рисунок 4 — Занятая память

Как мы видим в начале памяти идет `"Press Enter to Exit"`. Попробую представить это более наглядно:

50	72	65	73	73	20	45	6e	74	65	72	20	74	6f	20	45	78	69	74
P	r	e	s	s	E	n	t	e	r	t	o	E	x	i	t			

10 байт после этой записи зарезервированы под ввод, а константы просто заменяются самим компилятором nasm, поэтому памяти они не используют (аналогично `#define`).

Итак теперь все готово к запуску, вводим комманду `run`.

```

Register group: general
eax          0x0          0
ecx          0x0          0
edx          0x0          0
ebx          0x0          0
esp 0xfffffd390 0xfffffd390
ebp          0x0          0x0
esi          0x0          0
edi          0x0          0
eip 0x8049000 0x8049000 <_start>

B+>0x8049000 <_start>    mov    eax,0x4
0x8049005 <_start+5>    mov    ebx,0x1
0x804900a <_start+10>   mov    ecx,0x804a000
0x804900f <_start+15>   mov    edx,0x14
0x8049014 <_start+20>   int    0x80
0x8049019 <_start+22>   mov    eax,0x3
0x804901b <_start+27>   mov    ebx,0x0
0x8049020 <_start+32>   mov    ecx,0x804a014
0x8049025 <_start+37>   mov    edx,0xa
0x804902a <_start+42>   int    0x80
0x804902c <_start+44>   mov    eax,0x1

(gdb) br _start
Breakpoint 1 at 0x8049000
(gdb) run
Starting program: /home/debianuser/labs/lab1/lab1.1

Breakpoint 1, 0x08049000 in _start ()
(gdb) 
```

Рисунок 5 — Запуск программы

Как мы видим регистры `eax`, `ebx`, `ecx` и `edx` пусты, а курсор на первой ассемблерной инструкции. Выполним первую часть кода и посмотрим как изменится содержимое регистров.

```

eflags 0x202 [ IF ]
eax 0x4 4
ecx 0x804a000 134520832
edx 0x13 19
ebx 0x1 1
ebp 0x0 0x0
esi 0x0 0
edi 0x0 0
eip 0x804902a 0x804902a <_start+42>
14 14 +20>

0x8049014 <_start+20> int 0x80
B+ 0x8049000 <_start> mov eax,0x4
0x8049005 <_start+5> mov ebx,0x1
0x804900a <_start+10> mov ecx,0x804a000
0x804900f <_start+15> mov edx,0x13
>0x8049014 <_start+20> int 0x80
0x8049019 <_start+22> mov eax,0x3
0x804901b <_start+27> mov ebx,0x0
0x8049020 <_start+32> mov ecx,0x804a014
0x8049025 <_start+37> mov edx,0xA PTR [eax],al
0x804902a <_start+42> int 0x80
0x804902c <_start+44> mov eax,0x1

A debugging session is active.
The program is not being run.
The program is not being run.
(gdb) r
Starting program: /home/debianuser/labs/lab1/lab1.1

Breakpoint 1, 0x08049000 in _start ()
(gdb) stepi
0x08049005 in _start ()
0x0804900a in _start ()
0x0804900f in _start ()
0x08049014 in _start ()
(gdb) 
```

Рисунок 6 — Первая часть кода

Переместили все значения в регистры, и результатом стало, что все значения просто лежат в регистрах, но после системного прерывания значение в регистре `eax` поменяется на коды, возвращенные системой после печати.

The screenshot shows a debugger interface with several windows. At the top is a register window showing:

Register	Value	Description
eflags	0x202	[IF]
eax	0x13	19
ecx	0x804a000	134520832
edx	0x13	19
ebx	0x1	1
ebp	0x0	0x0
esi	0x0	0
edi	0x0	0
eip	0x804902a	0x804902a <_start+42>

Below the registers is an assembly code window showing the following instructions:

```
0x8049014 <_start+20> int    0x80
B+ 0x8049000 <_start>    mov    eax,0x4
0x8049005 <_start+5>     mov    ebx,0x1
0x804900a <_start+10>    mov    ecx,0x804a000
0x804900f <_start+15>    mov    edx,0x13
0x8049014 <_start+20>    int    0x80
>0x8049016 <_start+22>   mov    eax,0x3
0x804901b <_start+27>   mov    ebx,0x0
0x8049020 <_start+32>   mov    ecx,0x804a014
0x8049025 <_start+37>   mov    edx,0xAE PTR [eax],al
0x804902a <_start+42>   int    0x80
*0x804902c <_start+44>  mov    eax,0x1
```

Further down, there is a message about a debugging session being active, and the assembly code continues with:

```
A debugging session is active.
The program is not being run.
(gdb) r
Starting program: /home/debianuser/labs/lab1/lab1.1

Breakpoint 1, 0x08049000 in _start ()
```

The assembly code ends with:

```
(gdb) stepi
0x08049005 in _start ()
0x0804900a in _start ()
0x0804900f in _start ()
0x08049014 in _start ()
Press Enter to Exit0x08049016 in _start ()
(gdb) █
```

Рисунок 7 — Вызываем прерывание и видим как меняется значение регистра

После выполнения первой части кода уже понятно, как ведут себя регистры при операциях перемещения и вызове системы, поэтому просто для наглядности приведу еще один скриншот программы с перемещением в регистры второй части программы.

The screenshot shows a debugger interface with assembly code and register values. The assembly code at address 0x0804902a is:

```
0x08049014 <_start+20> int    0x80
0x08049016 <_start+22> mov    eax,0x3
0x08049018 <_start+27> mov    ebx,0x0
0x08049020 <_start+32> mov    ecx,0x804a014
0x08049025 <_start+37> mov    edx,0xa
0x0804902a <_start+42> int    0x80
0x0804902c <_start+44> mov    eax,0x1
0x08049031 <_start+49> xor    ebx,ebx
0x08049033 <_start+51> int    0x80    E PTR [eax],al
0x08049035 <_start+53> add    BYTE PTR [eax],al
0x08049037 <_start+55> add    BYTE PTR [eax],al
```

A message at the bottom left says "A debugging session is active." The registers show:

eflags	0x202	[IF]
eax	0x3	3
ecx	0x804a014	134520852
edx	0xa	10
ebx	0x0	0
ebp	0x0	0x0
esi	0x0	0
edi	0x0	0
eip	0x0804902a	0x0804902a <_start+42>

The stack dump shows:

```
0x08049005 in _start ()
0x0804900a in _start ()
0x0804900f in _start ()
0x08049014 in _start ()
Press Enter to Exit0x08049016 in _start ()
0x0804901b in _start ()
0x08049020 in _start ()
0x08049025 in _start ()
0x0804902a in _start ()
```

(gdb) █

Рисунок 8 — Готовим регистры для вызова ввода

Итак, мы нажали на enter, программа завершилась с кодом 0. Осталось только проверить флаги. Они хранятся в регистре **eflags**. Вот, что находится в этом регистре после завершения программы:

eflags	0x246	[PF ZF IF]
--------	-------	--------------

Рисунок 9 — Флаги после завершения программы

Приведу таблицу флагов.

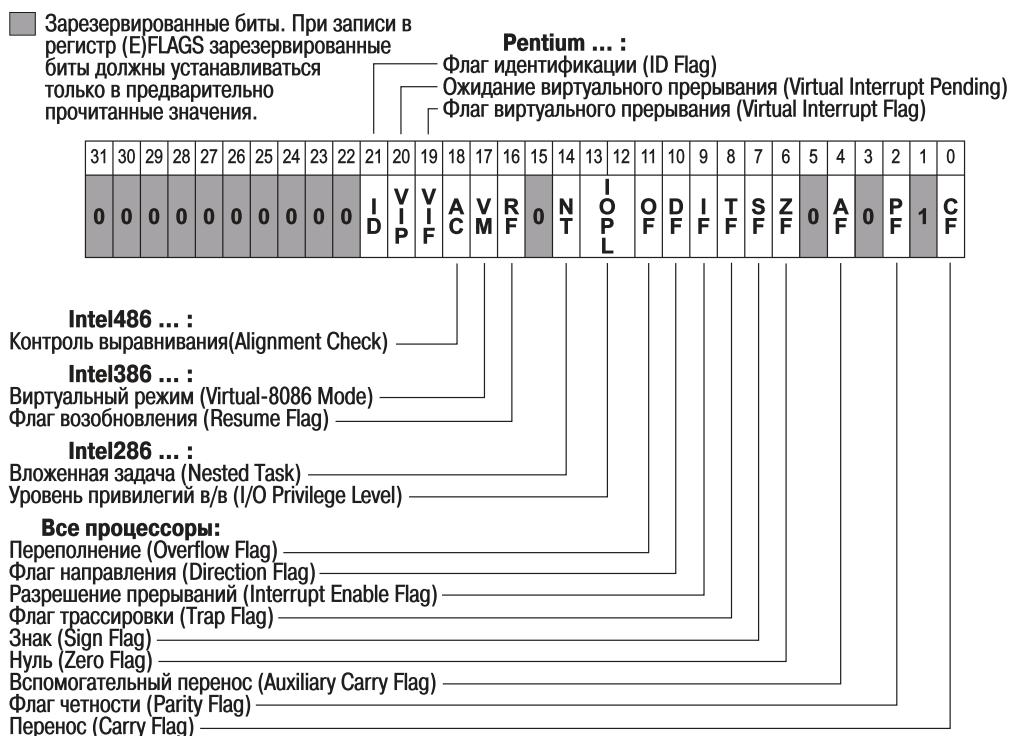


Рисунок 10 — Таблица флагов

Из таблицы видно, что так как программа завершилась нулем, она подняла флаг четности **PF** и флаг нуля **CF**. **IF** по умолчанию всегда поднят, если у программы есть возможность вызвать прерывание.

Отладка арифметических операций

Приведем другой ассемблерный код с арифметическими операциями:

```
section .data
    a dw -30
    b dw 21

section .bss
    x resd 1

section .text
global _start

_start:
    mov eax, [a] ; eax = -30
    add eax, 5   ; eax += 5 (eax = -25)
    sub eax, [b] ; eax -= 21 (eax = -46)
    mov [x], eax ; [x] = -46

    mov eax, 1
    int 80h
```

Листинг 4 — Код для отладки

Скомпилируем, соберем и запустим файл. Он просто завершит работу. Все операции мы можем посмотреть опять обратившись к отладке. Тут я построчно приведу всю отладку начиная с `_start`.

Но для начала снова проверим память:

```
(gdb) x/10x 0x804a000
0x804a000: 0x0015ffe2
```

Рисунок 11 — Память

Тут -30 это `0xffe2` (правило получения числа $-a$: переводим a в двоичную форму (прямой код), заменяем все 0 на 1 , а 1 на 0 (обратный код), прибавляем 1 к записи (дополнительный код)). Убедимся в правильности записи $30 = 0x1e$, преобразуем в обратный $0x1e = 00011110b \rightarrow 11100001b = 0xe1$ тут стоит заметить что на самом деле в памяти мы выделяем слово, получается 30 это не `0x1e`, а `0x001e`, тогда обратный код `0xffe1`, добавим 1 , чтобы получить дополнительный и получится как раз `0xffe2`.

А `b` положительно, поэтому представляется в памяти в виде простой формы $21 = 16+5 = 0x15$.

Теперь запускаем программу и смотрим на результат.

```

Registers:
eflags 0x202 [ IF ]
eax 0x0 0
ecx 0x0 0
edx 0x0 0
ebx 0x0 0
esp 0xffffd390 0xffffd390
ebp 0x0 0x0
esi 0x0 0
edi 0x0 0
eip 0x8049000 0x8049000 <_start>

B+ 0x08049000 <_start> mov eax,ds:0x804a000
0x8049005 <_start+5> add eax,0x5
0x8049008 <_start+8> sub eax,DWORD PTR ds:0x804a002
0x8049009 <_start+14> mov ds:0x804a004,eax
0x8049011 <_start+19> mov eax,0x1
0x8049014 <_start+24> int 0x80
0x8049015 <_start+25> add BYTE PTR [eax],al
0x8049016 <_start+28> add BYTE PTR [eax],al
0x8049018 <_start+30> add BYTE PTR [eax],al
0x8049020 <_start+32> add BYTE PTR [eax],al
0x8049022 <_start+34> add BYTE PTR [eax],al

(gdb) stepi
0x08049005 in _start()
0x08049008 in _start()
0x08049009 in _start()
0x08049011 in _start()
0x08049018 in _start()
[Inferior 1 (process 245293) exited normally]
(gdb) r
Starting program: /home/debianuser/labs/lab1/lab1.2

Breakpoint 1, 0x08049000 in _start()
(gdb)

```

Рисунок 12 — Запуск

```

Registers:
eflags 0x15ffe2 [ IF ]
eax 0x15ffe2 1441762
ecx 0x0 0
edx 0x0 0
ebx 0x0 0
esp 0xffffd390 0xffffd390
ebp 0x0 0x0
esi 0x0 0
edi 0x0 0
eip 0x8049005 0x8049005 <_start+5>

B+ 0x08049000 <_start> mov eax,ds:0x804a000
0x8049005 <_start+5> add eax,0x5
0x8049008 <_start+8> sub eax,DWORD PTR ds:0x804a002
0x8049009 <_start+14> mov ds:0x804a004,eax
0x8049011 <_start+19> mov eax,0x1
0x8049014 <_start+24> int 0x80
0x8049015 <_start+25> add BYTE PTR [eax],al
0x8049016 <_start+28> add BYTE PTR [eax],al
0x8049018 <_start+30> add BYTE PTR [eax],al
0x8049020 <_start+32> add BYTE PTR [eax],al
0x8049022 <_start+34> add BYTE PTR [eax],al

(gdb) br _start
Breakpoint 1 at 0x8049000
(gdb) r
Starting program: /home/debianuser/labs/lab1/lab1.2

Breakpoint 1, 0x08049000 in _start()
(gdb) stepi
0x08049005 in _start()
(gdb)

```

Рисунок 13 — *mov eax, [a]*

```

Registers:
eflags 0x206 [ PF IF ]
eax 0x15ffe7 1441767
ecx 0x0 0
edx 0x0 0
ebx 0x0 0
esp 0xffffd390 0xffffd390
ebp 0x0 0x0
esi 0x0 0
edi 0x0 0
eip 0x8049008 0x8049008 <_start+8>

B+ 0x08049000 <_start> mov eax,ds:0x804a000
0x8049005 <_start+5> add eax,0x5
0x8049008 <_start+8> sub eax,DWORD PTR ds:0x804a002
0x8049009 <_start+14> mov ds:0x804a004,eax
0x8049011 <_start+19> mov eax,0x1
0x8049014 <_start+24> int 0x80
0x8049015 <_start+25> add BYTE PTR [eax],al
0x8049016 <_start+28> add BYTE PTR [eax],al
0x8049018 <_start+30> add BYTE PTR [eax],al
0x8049020 <_start+32> add BYTE PTR [eax],al
0x8049022 <_start+34> add BYTE PTR [eax],al

(gdb) br _start
Breakpoint 1 at 0x8049000
(gdb) r
Starting program: /home/debianuser/labs/lab1/lab1.2

Breakpoint 1, 0x08049000 in _start()
(gdb) stepi
0x08049005 in _start()
0x08049008 in _start()
0x08049009 in _start()
(gdb)

```

Рисунок 14 — *add eax, 5*

```

Registers:
eflags 0x15ffd2 [ IF ]
eax 0x15ffd2 1441746
ecx 0x0 0
edx 0x0 0
ebx 0x0 0
esp 0xffffd390 0xffffd390
ebp 0x0 0x0
esi 0x0 0
edi 0x0 0
eip 0x804900e 0x804900e <_start+14>

B+ 0x08049000 <_start> mov eax,ds:0x804a000
0x8049005 <_start+5> add eax,0x5
0x8049008 <_start+8> sub eax,DWORD PTR ds:0x804a002
0x8049009 <_start+14> mov ds:0x804a004,eax
0x8049011 <_start+19> mov eax,0x1
0x8049014 <_start+24> int 0x80
0x8049015 <_start+25> add BYTE PTR [eax],al
0x8049016 <_start+28> add BYTE PTR [eax],al
0x8049018 <_start+30> add BYTE PTR [eax],al
0x8049020 <_start+32> add BYTE PTR [eax],al
0x8049022 <_start+34> add BYTE PTR [eax],al

(gdb) br _start
Breakpoint 1 at 0x8049000
(gdb) r
Starting program: /home/debianuser/labs/lab1/lab1.2

Breakpoint 1, 0x08049000 in _start()
(gdb) stepi
0x08049005 in _start()
0x08049008 in _start()
0x08049009 in _start()
(gdb)

```

Рисунок 15 — *sub eax, [b]*

```

Registers:
eflags 0x206 [ PF IF ]
eax 0x15ffd2 1441746
ecx 0x0 0
edx 0x0 0
ebx 0x0 0
esp 0xffffd390 0xffffd390
ebp 0x0 0x0
esi 0x0 0
edi 0x0 0
eip 0x8049013 0x8049013 <_start+19>

B+ 0x08049000 <_start> mov eax,ds:0x804a000
0x8049005 <_start+5> add eax,0x5
0x8049008 <_start+8> sub eax,DWORD PTR ds:0x804a002
0x8049009 <_start+14> mov ds:0x804a004,eax
0x8049013 <_start+19> mov eax,0x1
0x8049014 <_start+24> int 0x80
0x8049015 <_start+25> add BYTE PTR [eax],al
0x8049016 <_start+28> add BYTE PTR [eax],al
0x8049018 <_start+30> add BYTE PTR [eax],al
0x8049020 <_start+32> add BYTE PTR [eax],al
0x8049022 <_start+34> add BYTE PTR [eax],al

(gdb) br _start
Breakpoint 1 at 0x8049000
(gdb) r
Starting program: /home/debianuser/labs/lab1/lab1.2

Breakpoint 1, 0x08049000 in _start()
(gdb) stepi
0x08049005 in _start()
0x08049008 in _start()
0x08049009 in _start()
0x08049013 in _start()
(gdb)

```

Рисунок 16 — *mov [x], eax*

```

Registers:
eflags 0x1 1
eax 0x1 1
ecx 0x0 0
edx 0x0 0
ebx 0x0 0
esp 0xffffd390 0xffffd390
ebp 0x0 0x0
esi 0x0 0
edi 0x0 0
eip 0x8049018 0x8049018 <_start+24>

B+ 0x08049000 <_start> mov eax,ds:0x804a000
0x8049005 <_start+5> add eax,0x5
0x8049008 <_start+8> sub eax,DWORD PTR ds:0x804a002
0x8049009 <_start+14> mov ds:0x804a004,eax
0x8049013 <_start+19> mov eax,0x1
0x8049014 <_start+24> int 0x80
0x8049015 <_start+25> add BYTE PTR [eax],al
0x8049016 <_start+28> add BYTE PTR [eax],al
0x8049018 <_start+30> add BYTE PTR [eax],al
0x8049020 <_start+32> add BYTE PTR [eax],al
0x8049022 <_start+34> add BYTE PTR [eax],al

Breakpoint 1 at 0x8049000
(gdb) r
Starting program: /home/debianuser/labs/lab1/lab1.2

Breakpoint 1, 0x08049000 in _start()
(gdb) stepi
0x08049005 in _start()
0x08049008 in _start()
0x08049009 in _start()
0x08049013 in _start()
0x08049018 in _start()
(gdb)

```

Рисунок 17 — *mov eax, 1*

Так как числа записываются в память друг за другом, и по шине данных проходит ровно 32 бита, то в регистр `eax` попала не только `a`, но и `b`. Но операции не изменили знак числа, поэтому в конечном итоге мы получили правильный ответ, который записали в память:

```
(gdb) x/10x 0x804a000
0x804a000:      0x0015ffe2      0x0015ffd2
```

Рисунок 12 — Память программы после завершения

Как мы видим в память записывается весь регистр `eax`, но так как память выделена только для 1 байта, то неинициализированные переменные могут перезаписать код, например если в `section .bss` добавить `nw resb 2`, а в `_start` добавить перед выходом `mov [nw], 0`, то `15ff` затрется.

Но есть и другие способы решения этой проблемы. Сначала разберем, что нам необходимо с потерей данных, но все же записать в оперативную память некоторые данные размер которых превышает размер выделенной памяти. Тогда необходимо объявить сколько именно байт мы хотим записать в память, т.е. заменить `mov [x], eax` → `mov byte[x], eax`.

Но в идеале изначально отсечь все, что не относится к нашему числу, для этого можно заменить `mov eax, [a]` → `mov ax, [a]`, а дальше продолжить работать с `eax`. Еще есть более сложный способ – применение маски: дело в том, что если нам необходимо из числа `0xff271369` сделать `0x00000369`, то можно просто воспользоваться логической операцией `and` с литералом `0x00000fff`, тогда все «ненужные» числа логически умножаются на ноль и останутся только те, что логически умножились на 1. Для данного случая маску можно применить после `mov eax, [a]` так: `and eax, 0x0000ffff`.