



«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)

18.02.24

(Подпись, дата)

А. П. Плютто

(И. О. Фамилия)

Преподаватель

18.02.24

(Подпись, дата)

(И. О. Фамилия)

Москва 2024 г.

Содержание

Цель работы	3
Создание папок и файлов	4
Заготовка программы	5
Отладка арифметических операций	12
Работа с памятью	15
Определение данных	17
Работа с данными	19
Переполнение	20
Вывод	23

Цель работы

Изучение процессов создания, запуска и отладки программ на ассемблере Nasm под управлением операционной системы Linux, а также особенностей описания и внутреннего представления данных.

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
```

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 компановщику, 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.

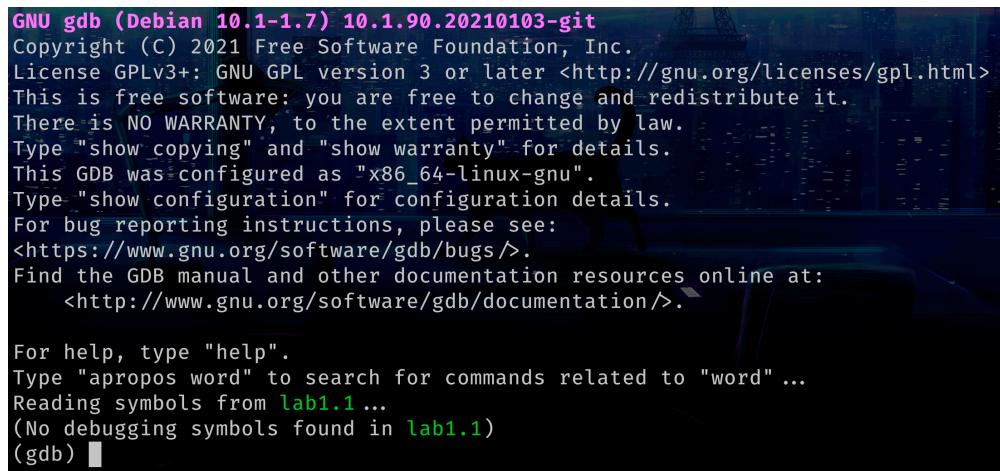
Тут следует немного подробнее остановиться на том как мы получаем этот размер строки, объявленный в `section .data`. Знак `$` воспринимается в диалекте `nasm` как адрес текущей ячейки памяти, т.е. если использовать знак `$` после объявления массива то мы получим адрес, который на единицу превосходит 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`.

The screenshot shows a terminal window with the following content:

```
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.

The screenshot shows a terminal window with the following content:

```
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 says: "A debugging session is active. Breakpoint 1, 0x08049000 in _start ()". 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>

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.

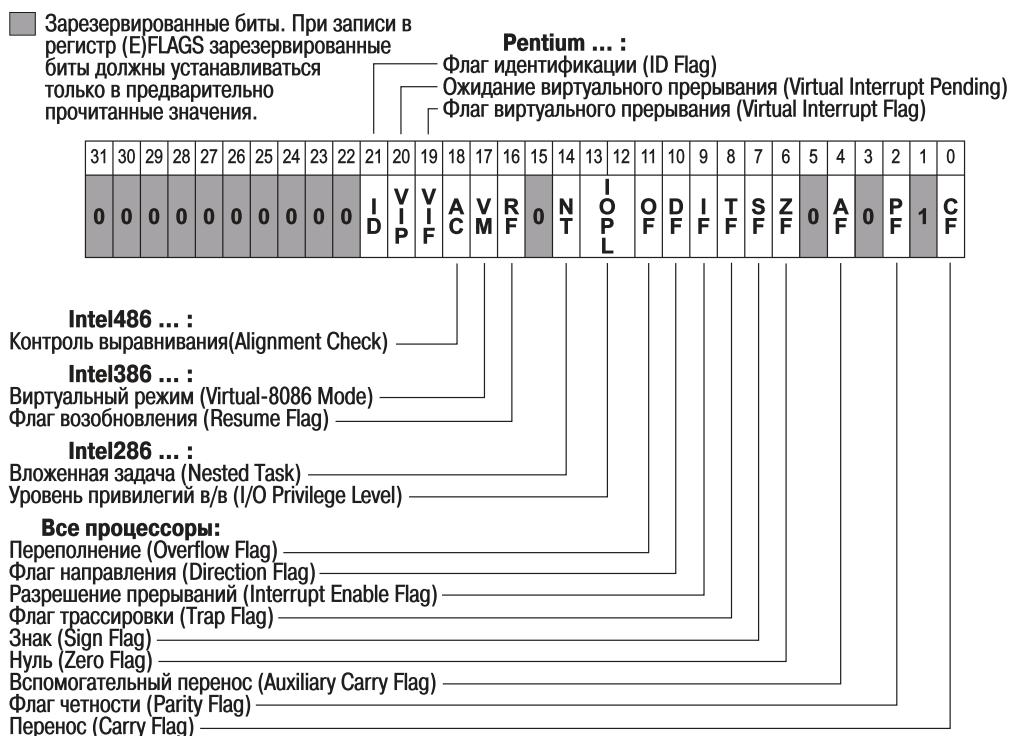


Рисунок 10 — Таблица флагов

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 → 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.

```

eflags      0x202      [ IF ]
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+ 0x08049000 <_start>    mov    eax,ds:0x804a000
0x08049005 <_start+5>    add    eax,0x5
0x08049008 <_start+8>    sub    eax,DWORD PTR ds:0x804a002
0x08049009 <_start+14>   mov    ds:0x804a004,eax
0x08049011 <_start+19>   mov    eax,0x1
0x08049014 <_start+24>   int    0x80
0x08049015 <_start+25>   add    BYTE PTR [eax],al
0x08049016 <_start+28>   add    BYTE PTR [eax],al
0x08049018 <_start+30>   add    BYTE PTR [eax],al
0x08049020 <_start+32>   add    BYTE PTR [eax],al
0x08049022 <_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

```

Register group: general
eax         0x15ffe2  1441762
ecx         0x0          0
edx         0x0          0
ebx         0x0          0
esp         0xfffffd390  0xfffffd390
ebp         0x0          0x0
esi         0x0          0
edi         0x0          0
eip         0x8049005  0x8049005 <_start+5>

B+ 0x08049000 <_start>    mov    eax,ds:0x804a000
0x08049005 <_start+5>    add    eax,0x5
0x08049008 <_start+8>    sub    eax,DWORD PTR ds:0x804a002
0x08049009 <_start+14>   mov    ds:0x804a004,eax
0x08049011 <_start+19>   mov    eax,0x1
0x08049014 <_start+24>   int    0x80
0x08049015 <_start+25>   add    BYTE PTR [eax],al
0x08049016 <_start+28>   add    BYTE PTR [eax],al
0x08049018 <_start+30>   add    BYTE PTR [eax],al
0x08049020 <_start+32>   add    BYTE PTR [eax],al
0x08049022 <_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) r

```

Рисунок 13 — *mov eax, [a]*

```

eflags      0x206      [ PF IF ]
eax         0x15ffe7  1441767
ecx         0x0          0
edx         0x0          0
ebx         0x0          0
esp         0xfffffd390  0xfffffd390
ebp         0x0          0x0
esi         0x0          0
edi         0x0          0
eip         0x8049008  0x8049008 <_start+8>

B+ 0x08049000 <_start>    mov    eax,ds:0x804a000
0x08049005 <_start+5>    add    eax,0x5
0x08049008 <_start+8>    sub    eax,DWORD PTR ds:0x804a002
0x08049009 <_start+14>   mov    ds:0x804a004,eax
0x08049011 <_start+19>   mov    eax,0x1
0x08049014 <_start+24>   int    0x80
0x08049015 <_start+25>   add    BYTE PTR [eax],al
0x08049016 <_start+28>   add    BYTE PTR [eax],al
0x08049018 <_start+30>   add    BYTE PTR [eax],al
0x08049020 <_start+32>   add    BYTE PTR [eax],al
0x08049022 <_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) r

```

Рисунок 14 — *add eax, 5*

```

eflags      0x206      [ PF IF ]
eax         0x15ffd2  1441746
ecx         0x0          0
edx         0x0          0
ebx         0x0          0
esp         0xfffffd390  0xfffffd390
ebp         0x0          0x0
esi         0x0          0
edi         0x0          0
eip         0x804900e  0x804900e <_start+14>

B+ 0x08049000 <_start>    mov    eax,ds:0x804a000
0x08049005 <_start+5>    add    eax,0x5
0x08049008 <_start+8>    sub    eax,DWORD PTR ds:0x804a002
0x08049009 <_start+14>   mov    ds:0x804a004,eax
0x08049011 <_start+19>   mov    eax,0x1
0x08049014 <_start+24>   int    0x80
0x08049015 <_start+25>   add    BYTE PTR [eax],al
0x08049016 <_start+28>   add    BYTE PTR [eax],al
0x08049018 <_start+30>   add    BYTE PTR [eax],al
0x08049020 <_start+32>   add    BYTE PTR [eax],al
0x08049022 <_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) r

```

Рисунок 15 — *sub eax, [b]*

```

eflags      0x206      [ PF IF ]
eax         0x15ffd2  1441746
ecx         0x0          0
edx         0x0          0
ebx         0x0          0
esp         0xfffffd390  0xfffffd390
ebp         0x0          0x0
esi         0x0          0
edi         0x0          0
eip         0x8049013  0x8049013 <_start+19>

B+ 0x08049000 <_start>    mov    eax,ds:0x804a000
0x08049005 <_start+5>    add    eax,0x5
0x08049008 <_start+8>    sub    eax,DWORD PTR ds:0x804a002
0x08049009 <_start+14>   mov    ds:0x804a004,eax
0x08049013 <_start+19>   mov    eax,0x1
0x08049014 <_start+24>   int    0x80
0x08049015 <_start+25>   add    BYTE PTR [eax],al
0x08049016 <_start+28>   add    BYTE PTR [eax],al
0x08049018 <_start+30>   add    BYTE PTR [eax],al
0x08049020 <_start+32>   add    BYTE PTR [eax],al
0x08049022 <_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) r

```

Рисунок 16 — *mov [x], eax*

```

eflags      0x206      [ PF IF ]
eax         0x1          1
ecx         0x0          0
edx         0x0          0
ebx         0x0          0
esp         0xfffffd390  0xfffffd390
ebp         0x0          0x0
esi         0x0          0
edi         0x0          0
eip         0x8049018  0x8049018 <_start+24>

B+ 0x08049000 <_start>    mov    eax,ds:0x804a000
0x08049005 <_start+5>    add    eax,0x5
0x08049008 <_start+8>    sub    eax,DWORD PTR ds:0x804a002
0x08049009 <_start+14>   mov    ds:0x804a004,eax
0x08049013 <_start+19>   mov    eax,0x1
0x08049014 <_start+24>   int    0x80
0x08049015 <_start+25>   add    BYTE PTR [eax],al
0x08049016 <_start+28>   add    BYTE PTR [eax],al
0x08049018 <_start+30>   add    BYTE PTR [eax],al
0x08049020 <_start+32>   add    BYTE PTR [eax],al
0x08049022 <_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) r

```

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 18 — 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`.

Так же стоит отметить, что изначально мы выделяем памяти на 1 байт меньше для ответа, так что использовать ключевое слово `byte` все равно придется.

Работа с памятью

В отличии от высокоуровневых языков со строгой типизацией в ассемблере, при выделении памяти в нее можно положить любое значение, оно будет транслировано компилятором в численное значение.

```
section .data
    val1  db 255
    chart dw 256
    lue3  dw -128
    v5    db 10h
          db 100101B
    beta   db 23,23h,0ch
    sdk    db "Hello",10
    min    dw -32767
    ar     dd 12345678h
    valar  times 5 db 8

section .bss
    alu   resw 10
    f1    resb 5

section .text
global _start

_start:
    mov eax, 1
    int 80h
```

Листинг 5 — Код для работы с памятью

Скомпилируем, соберем и откроем в отладчике код. В коде нас интересуют только данные в памяти:

```
(gdb) x/10x 0x804a000
0x804a000: 0x800100ff 0x172510ff 0x65480c23 0xa6f6c6c
0x804a010: 0x56788001 0x08081234 0x00080808 0x00000000
```

Рисунок 19 — Память программы

Разбирать данные будем снизу вверх, справа налево, каждые 32 бита, слева направо каждый. Для удобства напишу таблицу, в которой показано какой байт к чему относится, адресация идет на каждый байт, в строке 4 байта поэтому адрес меняется на 4, уменьшается адрес потому что мы читаем снизу вверх (адрес полностью не вместился так что к написанному смещению необходимо прибавить `0x804a000`):

Смещение	Значение 32 бит	1 байт	2 байт	3 байт	4 байт
30	00 00 00 00	f1	f1	f1	f1
2c	00 00 00 00	f1	alu	alu	alu
28	00 00 00 00	alu	alu	alu	alu
24	00 00 00 00	alu	alu	alu	alu
20	00 00 00 00	alu	alu	alu	alu
1c	00 00 00 00	alu	alu	alu	alu
18	00 08 08 08	alu	valar	valar	valar
14	08 08 12 34	valar	valar	ar	ar
10	56 78 80 01	ar	ar	min	min
0c	0a 6f 6f 6c	sdk \n	sdk o	sdk l	sdk l
08	65 48 0c 23	sdk e	sdk H	beta	beta
04	17 25 10 ff	beta		v5	lue3
00	80 01 00 ff	lue3	chart	chart	val1

Определение данных

Задание: Определите в памяти следующие данные:

1. целое число 25 размером 2 байта со знаком;
2. двойное слово, содержащее число -35;
3. символьную строку, содержащую ваше имя (русскими буквами и латинскими буквами).

Вот получившийся по данному заданию код:

```
section .data
    a    dw 25
    b    dd -35
    name dw "Andrey Андрей"

section .text
global _start

_start:
    mov eax, 1
    int 80h
```

Листинг 6 — *Определение данных*

Скомпилируем, соберем и откроем в отладчике код. В коде нас снова интересуют только данные в памяти:

```
(gdb) x/10x 0x804a000
0x804a000: 0xffffd0019      0x6e41ffff      0x79657264      0xd090d020
0x804a010: 0xd1b4d0bd     0xd0b5d080     0x000000b9     0x00000000
```

Рисунок 20 — *Память программы*

Составим такую же таблицу, что и в предыдущем задании.

Смещение	Значение 32 бит	1 байт	2 байт	3 байт	4 байт
18	00 00 00 b9				н ў
14	d0 b5 d0 80	п ў	п е	п е	п р
10	d1 b4 d0 bd	п р	п д	п д	п н
0c	d0 90 d0 20	п н	п А	п А	п
08	79 65 72 64	п у	п е	п r	п d
04	6e 41 ff ff	п п	п А	б	б
00	ff dd 00 19	б	б	а	а

Так как буквы кириллицы не входят в ASCII, то для их записи необходимо 2 байта, для латиницы используется только 1 байт. Тут пример из начала с получением длины не сможет сработать, ведь у нас каждая буква кодируется разным количеством байт.

Работа с данными

Задание: Определите несколькими способами в программе числа, которые во внутреннем представлении (в отладчике) будут выглядеть как 25 00 и 00 25. Проверьте правильность ваших предположений, введя соответствующие строки в программу.

Вот получившийся по данному заданию код:

```
section .data
    a dd 0x25
    b dd 0x2500

section .text
global _start

_start:
    mov bl, [a]
    mov ch, [a]

    mov eax, 1
    mov ebx, 0
    int 80h
```

Листинг 7 — Работа с данными

В данном задании требуется несколькими способами вывести числа. В отладчике все числа выводятся в 16-ричном формате, так что буду работать с ним. Первым способом я выбрал простой ввод в память двух чисел. Чтобы они не «слиплись» использую двойное слово – 32 бита.

```
(gdb) x/10x 0x804a000
0x804a000:      0x00000025          0x00002500
```

Рисунок 21 — Первый способ

Вторым способом я выбрал использование регистров, ведь 16-тибитные `ax`, `bx`, , `dx` разделяются на верхние `ah`, `bh`, `ch`, `dh` и нижние `al`, `bl`, `cl`, `dl`. Таким образом если `0x25` внести в нижний регистр то в отладчике он так и будет, а если в верхний, то он автоматически умножится на 256, т.е. станет `0x2500`.

```
(gdb) info registers ebx
ebx          0x25              37
(gdb) info registers ecx
ecx          0x2500            9472
```

Рисунок 22 — Второй способ

Переполнение

Добавьте в программу переменную `F1 = 65535` размером слово и переменную `F2 = 65535` размером двойное слово. Вставьте в программу команды сложения этих чисел с 1:

```
add [F1],1  
add [F2],1
```

Проанализируйте и прокомментируйте в отчете полученный результат (обратите внимание на флаги).

Вот получившийся код:

```
section .data  
f1 dw 65535 ; 0xffff  
f2 dd 65535 ; 0x0000ffff  
  
section .text  
global _start  
  
_start:  
    mov ax, [f1]  
    mov ebx, [f2]  
    add ax, 1      ; 0x0000, OF  
    add ebx, 1      ; 0x00010000  
  
    mov eax, 1  
    int 80h
```

Листинг 8 — Переполнение

Заходим в отладчик и запускаем программу:

```

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   ax,ds:0x804a000
0x8049006 <_start+6>    mov   ebx,DWORD PTR ds:0x804a002
0x804900c <_start+12>   add   ax,0x1
0x8049010 <_start+16>   add   ebx,0x1
0x8049013 <_start+19>   mov   eax,0x1
0x8049018 <_start+24>   int   0x80
0x804901a                   add   BYTE PTR [eax],al
0x804901c                   add   BYTE PTR [eax],al
0x804901e                   add   BYTE PTR [eax],al
0x8049020                   add   BYTE PTR [eax],al
0x8049022                   add   BYTE PTR [eax],al

(gdb) br _start
Breakpoint 1 at 0x8049000
(gdb) r
Starting program: /home/debianuser/labs/lab1/lab1.6

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

```

Рисунок 23 — Запуск

Добавляем значения в регистры:

```

Register group: general
eax      0xffff      65535
ecx      0x0          0
edx      0x0          0
ebx      0xffff      65535
esp      0xfffffd390  0xfffffd390
ebp      0x0          0x0
esi      0x0          0
edi      0x0          0
eip      0x804900c    0x804900c <_start+12>

B+ 0x8049000 <_start>    mov   ax,ds:0x804a000
0x8049006 <_start+6>    mov   ebx,DWORD PTR ds:0x804a002
>0x804900c <_start+12>   add   ax,0x1
0x8049010 <_start+16>   add   ebx,0x1
0x8049013 <_start+19>   mov   eax,0x1
0x8049018 <_start+24>   int   0x80
0x804901a                   add   BYTE PTR [eax],al
0x804901c                   add   BYTE PTR [eax],al
0x804901e                   add   BYTE PTR [eax],al
0x8049020                   add   BYTE PTR [eax],al
0x8049022                   add   BYTE PTR [eax],al

(gdb) br _start
Breakpoint 1 at 0x8049000
(gdb) r
Starting program: /home/debianuser/labs/lab1/lab1.6

Breakpoint 1, 0x08049000 in _start ()
(gdb) si
0x08049006 in _start ()
0x0804900c in _start ()
(gdb) 

```

Рисунок 24 — Значения в регистрах

Устраиваем переполнение для регистра `ax` :

```

eflags      0x257      [ CF PF AF ZF IF ]
eax        0x0        0
ecx        0x0        0
edx        0x0        0
ebx      0xffff      65535
esp      0xffffd390  0xffffd390
ebp        0x0        0x0
esi        0x0        0
edi        0x0        0
eip      0x8049010  0x8049010 <_start+16>

B+ 0x8049000 <_start>    mov    ax,ds:0x804a000
0x8049006 <_start+6>    mov    ebx,DWORD PTR ds:0x804a002
0x804900c <_start+12>   add    ax,0x1
>0x8049010 <_start+16>  add    ebx,0x1
0x8049013 <_start+19>   mov    eax,0x1
0x8049018 <_start+24>   int    0x80
0x804901a           add    BYTE PTR [eax],al
0x804901c           add    BYTE PTR [eax],al
0x804901e           add    BYTE PTR [eax],al
0x8049020           add    BYTE PTR [eax],al
0x8049022           add    BYTE PTR [eax],al

(gdb) br _start
Breakpoint 1 at 0x8049000
(gdb) r
Starting program: /home/debianuser/labs/lab1/lab1.6

Breakpoint 1, 0x08049000 in _start ()
(gdb) si
0x08049006 in _start ()
0x0804900c in _start ()
0x08049010 in _start ()
(gdb) 

```

Рисунок 25 — *Переполнение*

Так как мы прибавили 1, то при переполнении ах перешагнул на 1 и стал 0, поэтому поднялся нулевой флаг **ZF** и флаг четности **PF**. **AF** и **CF** как флаги переноса свидетельствуют о переполнении регистра.

ebx имеет размер в 2 раза больше, поэтому при прибавлении единицы значение в этом регистре станет просто **0x10000** :

```

eflags      0x216      [ PF AF IF ]
eax        0x0        0
ecx        0x0        0
edx        0x0        0
ebx      0x10000     65536
esp      0xffffd390  0xffffd390
ebp        0x0        0x0
esi        0x0        0
edi        0x0        0
eip      0x8049013  0x8049013 <_start+19>

B+ 0x8049000 <_start>    mov    ax,ds:0x804a000
0x8049006 <_start+6>    mov    ebx,DWORD PTR ds:0x804a002
0x804900c <_start+12>   add    ax,0x1
0x8049010 <_start+16>  add    ebx,0x1
>0x8049013 <_start+19>  mov    eax,0x1
0x8049018 <_start+24>   int    0x80
0x804901a           add    BYTE PTR [eax],al
0x804901c           add    BYTE PTR [eax],al
0x804901e           add    BYTE PTR [eax],al
0x8049020           add    BYTE PTR [eax],al
0x8049022           add    BYTE PTR [eax],al

(gdb) br _start
Breakpoint 1 at 0x8049000
(gdb) r
Starting program: /home/debianuser/labs/lab1/lab1.6

Breakpoint 1, 0x08049000 in _start ()
(gdb) si
0x08049006 in _start ()
0x0804900c in _start ()
0x08049010 in _start ()
0x08049013 in _start ()
(gdb) 

```

Рисунок 26 — **ebx** не переполняется

Вывод

В процессе работы были изучены процессы создания, запуска и отладки программ на ассемблере NASM под управлением Linux. Были освоены специфики описания и внутреннего представления данных, а также изучены операции ввода-вывода через прерывания, арифметические операции, работа с памятью и отслеживание переполнения. Этот опыт позволил более глубоко понять внутреннее устройство компьютера, архитектуру и специфику работы операционной системы на низком уровне.