**Практическая работа № 11-14.**

# *Низкоуровневое программирование*

## *Текст задания:*

**Задание 1.** Выполнить и проанализировать, используя

**https://www.jdoodle.com/compile-assembler-nasm-online**

```
;Листинг 01 - минимальная программа для Linux
;Приемы оптимизации не применяются для упрощения кода

global _start

_start:

Mov eax, 4
Mov ebx, 1
Mov ecx, msg
Mov edx, msglen
int 0x80

mov eax, 1
mov ebx, 0
int 0x80

section .data

msg: db "Linux rulez 4ever",0x0A,0
msglen equ $-msg
```

Рассмотрим программу поподробнее:
**Знак ';'** (точка с запятой) означает *комментарий* - все что находится правее этого символа ассемблер игнорирует

**global _start** - директива **global** указывает ассемблеру сделать *глобальной (экспортируемой)* метку "_start". Подробнее об экспортируемых метках см. ниже

**_start:** - объявление *метки* с именем "_start". Фактически это означает, что в программе будет определена константа **_start**, которая будет иметь значение равное *адресу, по которому объявлена данная метка*

Предыдущие три строчки были *директивами* ассемблера, т.е. не являлись командами процессора, и не преобразовывались при компиляции в машинный код. Следующие строчки являются именно *командами* процессора:

**mov eax, 4** - машинная команда MOV копирует данные из второго операнда в первый. В данном случае первый операнд - это *регистр* EAX (подробнее о регистрах - в следующем уроке). Второй операнд - это *константа*(определенное в момент компилирования и неизменяемое значение). Результатом выполнения этой команды будет то, что в регистре EAX окажется число 4. Операнды команды разделяются запятой

**Mov ebx, 1** - то же самое, но помещается единица в регистр EBX

**mov ecx, msg** - на первый взгляд эта команда отличается от двух предыдущих, но она тоже выполняет перемещение данных, только в данном случае используется константа **msg**, которая определена ниже и регистр ECX

**movedx, msglen** - содержимое определенной ниже константы **msglen** помещается в регистр EDX

**int 0x80** — команда **int** процессора вызывает т.н. *программное прерывание*. Грубо говоря - программное прерывание - это команда перехода выполнения программы в определенной операционной системе - *обработчика прерывания*. Всего процессор поддерживает 256 обработчиков для 256 прерываний и операнд этой команды указывает на обработчик какого прерывания нужно передать выполнение программы. **0x80** - 80 в шестнадцатеричной системе счисления (на шестнадцатеричную систему указывают первые два символа: **0x**). В случае ОС Linux, прерывание с номером 0x80 является *системным вызовом* - передачей управления ядру системы с целью выполнения каких-либо действий. В регистре EAX должен находиться *номер системного вызова*, в зависимости от которого ядро системы будет выполнять какие-либо действия. В данном случае мы помещаем в EAX число 4, т.е. указываем ядру выполнить системный вызов номер 4 (write). Этот системный вызов используется для записи данных в файл или на консоль (которая тоже в принципе представлена файлом). В EBX мы поместили *дескриптор*(идентификатор) консоли - stdout. В ECX и EDX содержатся *адрес начала сообщения (адрес первого байта)* и длина сообщения в байтах. Т.е. этот системный вызов должен выполнить вывод строчки, находящейся по адресу **msg**, на консоль.

**moveax, 1** - в EAX помещается 1 - номер системного вызова "exit"

**movebx, 0** - в EBX помещается 0 - параметр вызова "exit" означает *код,* с которым завершится выполнение программы

**int 0x80** - системный вызов. После системного вызова "exit" выполнение программы завершается

**section .data** Директива ассемблера **section** определяет следующие данные, как находящиеся в указанном в качестве параметра сегменте. Сегмент **.text** - *сегмент кода*, в котором должен находиться исполняемый код программы и чтение из которого запрещено. Сегмент **.data** - *сегмент данных*, в котором должны находиться данные программы. Выполнение (передача управления) на сегмент данных запрещена. Поскольку следующие строчки нашей программы - данные, то мы определяем сегмент данных.

**msg: db "Linuxrulez 4ever",0x0A,0** - вначале мы определяем метку **msg** (напоминаю, что метка - текущий адрес), и сразу после нее - строчку, т.е. метка **msg** будет указывать на первый байт строки. Директива **db**указывает ассемблеру поместить в данном месте байт данных. Несколько байт могут быть разделены запятой. Если нужно поместить символ, то запись **'X'** означает код символа 'X', а форма записи **"abcde"** эквивалентна **'a', 'b', 'c', 'd', 'e'**. Код символа 0x0A означает переход строки, а нулевой байт является концом строки. Поскольку вызов write знает точно, сколько байт нужно выводить, то нулевой байт в конце строки необязателен, но мы его все равно поставим :). Он необходим для программ, взаимодействующих с GLIBC, т.к. функции стандартной библиотеки Си вычисляют длину строки, как расстояние между первым байтом и ближайшим нулевым байтом.

**msglenequ $-msg** - директива **equ** определяет константу, расположенную слева от директивы и присваивает ей значение, находящееся справа. **Символ $** является специальной константой ассемблера, значение которой всегда равно адресу по которому она находится, т. е. в данном случае выражение **$ - msg** как раз будет равно длине строки, т.к. в данном месте программы **$** равно адресу следующего за строкой байта. Результат этой директивы - мы определили константу **msglen**, значение которой равно длине определенной выше строки.

## Задание 2.

Выполнить и объяснить операторы программы, использовать электронный учебник§2-3

```
section .text

global _start

_start:

moveax, [x]
subeax, '0'
movebx, [y]
subebx, '0'
addeax, ebx
addeax, '0'

mov     [sum], eax

movecx, msg
movedx, len
movebx, 1
moveax, 4
int     0x80

movecx, sum
movedx, 1
movebx, 1
moveax, 4
```

```
int    0x80

moveax, 1
int    0x80

section .data
xdb '5'
ydb '3'
msgdb  "sum of x and y is "
lenequ $ - msg

segment .bss

sumresb 1
```

## Задание 3.

Выполнить и объяснить операторы программы, использовать электронный учебник§2-3

```
; ------------------------------
section.data
; Храним здесь инициализированные данные (переменные)

; этострочка
str1db  'Here is string 1', 0xA

; это константа (длина объявленной выше строчки)
str1_len    equ $ - str1

; это двухбайтовая инициализированная переменная
```

```
pidw  0x123d


; Это четырехбайтовая переменная
ksidd  0x12345678


; Можно и так
ksi2   dd  'acde'


; -------------------------------
section.bss
; Область неинициализированных данных (резерв)
memresb    12800


; -------------------------------
section.text


; elf entry point
global _start


_start:
; systemwrite ->stdout
; аргументы для системного вызова write помещаются в регистры
;  ebx(куда) , ecx (что), edx (какой длины)
moveax, 4; номер системного вызова write
movebx, 1; номер потока вывода (stdout)
movecx, str1; адрес, по которому лежит строчка
movedx, str1_len; количество байт (символов), которые нужно вывести
int 80h; системный вызов



; а теперь просто для тренировки
```

```
; скопируем строчку в раздел неинициализированых данных
movecx, str1_len

.loop:; посимвольно копируем строчку
movesi, ecx;ecx — "переменная" цикла
moval, byte [str1+esi]
movbyte [mem+esi], al
loop .loop; эта команда уменьшает ecx на 1 и стравнивает с 0
; если 0, то переходит к следующей команде
moveax, 4; Напечатаем строчку из того места,
movebx, 1; куда мы её скопировали
movecx, mem             ;
movedx, str1_len
int 80h

push 1234abc1h          ; вызовемфункциюprint_hex,
callprint_hex; которая описана ниже

; system exit 0
moveax, 1
movebx, 0
int 80h

print_hex:
pushebp
movebp, esp
subesp, 8h
; берем первый (и последний аргумент)
movecx, [ebp+8]
movesi, 8
```

```
.loop:
moveax, ecx
andeax, 0xf                    ; эквивалентноеax = eax % 16
; (остаток от деления на 16)


cmpal, 9; результат сравнения сохраняется в специальном
; регистрефлагов
jle .print_decimal; jump if less or equal
; — смотрит результат сравнения в регистре флагов
; и переходит на метку .print_decimal, если  al<= 9
.print_hex:
subal, 10
addal, 'a'
jmp .print1
.print_decimal:
addal, '0'
.print1:
decesi
movbyte [esp+esi], al
shrecx, 4                ; эквивалентноесx = ecx / 16
jz  .ret
jmp .loop
.ret
moveax, 4
movebx, 1
movecx, esp
movedx, 8
int 80h
leave
ret
```

**Перечень объектов контроля и оценки**

| Наименование объектов контроля и оценки | Основные показатели оценки результата | Оценка |
|---|---|---|
| **36** принципы работы кэш-памяти; | знание архитектуры ЦП и кэш-памяти и принципов обмена данными | |

За правильный ответ на вопросы или верное решение задачи выставляется положительная оценка – 5 баллов.

За не правильный ответ на вопросы или неверное решение задачи выставляется отрицательная оценка – 2 балла (неуд).

## **Практическая работа № 14.** Работа с прерываниями и подпрограммами

### Текст задания:

**Задание 1.**

Выполнить программу в онлайн версии ассемблера

https://www.jdoodle.com/compile-assembler-nasm-online

В отчет по практической работе включить объяснение структуры программы и назначение операторов в ходе решения. Учебник по операторам языка смотри в той же папке.

**Задание 2**

Данная программа имеет простую логику: считывает целое число и выводит его. Но в коде вместо описания функции`scan_int`стоит "заглушка", которая всегда возвращает 133.

```
section.bss
bufferresb 20

section.text
global _start
_start:
callscan_int; читатьintизstdinврегистреax
pusheax; поместить eax в стек — это будет
; аргументом для следующей функции

callprint_int; напечатать int, который находится в вершине стека
; в стандартный поток вывода
; "callfunc" эквивалентно
;    push<адрес_следующей_инструкции>;
;    jmpfunc

moveax, 1; Эти три строчки эквиваленты exit(0)
```

```
movebx, 0;
int 0x80;

scan_int:; эта функция должна читать int из stdin,
moveax, 133; но пока она не реализована и просто
ret; возвращает число 133

print_int:; функция печати целого числа в stdout
; аргумент (4-байтовое целое число)
; находится в вершине стека
; ebp содержит адрес начала stackframe
; esp содержит адрес вершины стека
; esp<ebp, то есть вершина имеет меньший адрес
; в начале по адресам (ebp-4, ebp-3, ebp -2, ebp -1) лежат
; четыре байта целого числа, которое нам передали
; в качестве аргумента

pushebp; поместим в стек адрес начала стека
; этот push автоматически делает esp -= 4
movebp, esp; теперь ebp равно esp

; аргументы находятся по адресу ebp + 8
movecx, [ebp+8]; значение переданного нам целого числа поместим в ecx

xoredx, edx; обнулим edx
movesi, 10; на 10 мы будем делить.

movedi, 18; символы-цифры нашего числа мы будем помещать
; поадресам buffer + 17, buffer+16, buffer+15, ...

movbyte [buffer + 18], 0xA; 19-й и 20-й символы — это перенос строчки
movbyte [buffer + 19], 0; и символ конца строки

.loop:
moveax, ecx;
xoredx, edx; данные четыре строки дают
divesi;   ecx = ecx / 10
movecx, eax;


addedx, '0'; '0' ассемблером интерпретируется как ASCII код символа '0'
decedi
movbyte [buffer+edi], dl
cmpecx, 0
jne .loop

moveax, 4         ; эквивалентно write( 1, buffer + edi, 19 - edi )
movebx, 1
movecx, buffer    ; можнокороче — lea ecx, [buffer+edi]
addecx, edi
movedx, 19
subedx, edi
int 0x80

leave; эквивалентномovesp, ebp
;                 popebp
ret; эквивалентно  pop IP
;
```

1. Напишите правильную реализацию функции `scan_int`.

2. Напишите программу на NASM, которая считывает два целых числа и выводит результат их сложения и умножения. Используйте результаты предыдущей задачи.
3. В отчет включить свой вариант функции и объяснение алгоритма

За правильный ответ на вопросы или верное решение задачи выставляется положительная оценка.

***Контрольные вопросы?***

1. Где хранятся аргументы команды sub?
2. Где хранится результат команды sub?
3. Где хранятся аргументы команды mul?
4. Где хранится результат команды mul?
5. Что делает команда push?
6. Какая команда завершает подпрограмму?
7. Каким образом передается результат работы подпрограммы?

За неправильный ответ на вопросы или неверное решение задачи выставляется отрицательная оценка – 0 баллов.

*Шкала оценки образовательных достижений*

| Процент результативности (правильных ответов) | Оценка уровня подготовки | |
|---|---|---|
| | балл (отметка) | вербальный аналог |
| 90 ÷ 100 | 5 | отлично |
| 80 ÷ 89 | 4 | хорошо |
| 70 ÷ 79 | 3 | удовлетворительно |
| менее 70 | 2 | неудовлетворительно |