**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 Программное обеспечение ЭВМ и информационные технологии

# ОТЧЕТ ПО УЧЕБНОЙ ПРАКТИКЕ

Студент Сергеев Михаил Алексеевич

Группа ИУ7-48Б(В)

Тип практики _____

Название предприятия_____

Студент _____ **Сергеев М.А.**
*подпись, дата* *фамилия, и.о.*

Руководитель практики _____ **Кузнецов Д.А.**
*подпись, дата* *фамилия, и.о.*

Оценка _____

*2020 г.*

# ЗАДАНИЕ НА ЭКСПЛУАТАЦИОННУЮ ПРАКТИКУ

Изучить формат представления различных структур данных в памяти компьютера и реализацию отдельных распространённых алгоритмов на уровне машинного кода.

Структуры данных: числа, символы, строки, одномерные и многомерные массивы.

Алгоритмы: посимвольная обработка строк, ввод/вывод числа, поиск минимума/максимума в массиве, построчная обработка матрицы на примере поиска чётных элементов.

Обработка процессором вызова подпрограммы и возврата из подпрограммы.

Отчёт должен содержать исходный код реализации указанных алгоритмов.

**ОГЛАВЛЕНИЕ**

# ВВЕДЕНИЕ

В настоящее время профессия программиста становится всё более популярной и востребованной. Огромное количество как очных, так и онлайн-курсов предлагают освоить в совершенстве за кратчайший период множество самых разных языков программирования. Видится необходимым, однако, отметить различие между ролями программиста как такового и инженера программного обеспечения. Особенно ярко такое различие отмечается в английском языке, где существуют разные, в общем-то, специализации, имеющие названия «programmer» и «software engineer» соответственно. Предполагается, что инженеру программного обеспечения недостаточно лишь писать программы на одном из распространённых языков высокого уровня, а необходимо детально представлять, какие информационные процессы происходят при работе ЭВМ. Высокоуровневые языки не дают представления об этом, компиляторы лишь транслируют программы, написанные на таких языках, в машинный код, который и используется вычислительным устройством. Очевидно, что для детального понимания того, как работают программы, было бы исключительно полезно познакомиться с архитектурой вычислительной машины, определить, какие её ресурсы доступны для выполнения тех или иных операций, в частности, узнать, какие встроенные команды содержит тот или иной микропроцессор, как осуществляется ввод, обработка, хранение и вывод информации. Поскольку в рамках курса дисциплины «Машинно-зависимые языки программирования» мы работаем исключительно с персональными компьютерами, построенными в соответствии с принципами архитектуры фон Неймана, вычислительные машины, использующие другую архитектуру, нами рассматриваться не будут. Вместе с тем, необходимо отметить, что даже персональные ЭВМ, имеющие, в общем-то, одинаковую структурную схему, могут быть построены на основе микропроцессоров, использующих различную архитектуру (наиболее характерными примерами являются с одной стороны персональные

компьютеры, использующие архитектуру x86, а с другой – современные мобильные телефоны на архитектуре ARM; более того, ряд производителей анонсировал появление на рынке персональных компьютеров с процессорами на архитектуре ARM). Следствием этого является то, что, в зависимости от платформы, программа, выполняющая для пользователя одни и те же функции, по факту будет по разному представлена в машинном коде, то есть он будет зависеть от конкретной машины. Поэтому такой язык называется машинно-зависимым. Для более комфортной работы с таким кодом предусмотрено мнемоническое сокращение команд. Это обеспечивает удобство при взаимодействии с ним, облегчая как внесение изменений в существующие программы, так и написание собственных программ. Совокупность таких команд можно определить как язык ассемблера.

Настоящий отчёт посвящён изучению формата представления различных структур данных в памяти компьютера и реализации отдельных распространённых алгоритмов на уровне машинного кода (на примере использования языка ассемблера).

# 1. СТРУКТУРЫ ДАННЫХ.

## 1.1. ЧИСЛА.

Очевидно, что числа являются очень удобным средством представления информации. Так, с помощью совокупности чисел мы можем закодировать, к примеру, изображение на плоскости (используя три параметра – координаты и условный номер цвета), текст (поставив в соответствие каждому символу определённое число) или порядок изменения цвета новогодней гирлянды. Дискретность значительно повышает удобство передачи информации без искажений (в особенности в сочетании с использованием помехоустойчиых кодов). Именно цифровые вычислительные машины получили наибольшее распространение, в особенности с семидесятых годов XX века, когда микропроцессоры стали относительно доступны. Тогда же был выпущен и первый 16-разрядный процессор Intel 8086. Реализованная в нем архитектура набора команд стала основой архитектуры, известной сейчас как x86. Именно на примере этой архитектуры мы и будем рассматривать средства, доступные нам в языке ассемблера.

Процессоры на архитектуре x86 хранят и обрабатывают числа в двоичном коде. Такие числа удобно представлять для нас в шестнадцатеричной системе счисления. Так, число $14_{10}$ в двоичной системе счисления выглядит как $1110_2$, а в шестнадцатеричной – как $E_{16}$. Инструкции ассемблера позволяют работать напрямую с числами в двоичной, восьмеричной, десятичной и шестнадцатеричной системе счисления. Необходимо, при этом, однако, указать выбранную систему счисления. Делается это при помощи букв «b», «o», «d» и «h» соответственно, записанных непосредственно за числом. Числа без этого символа по умолчанию воспринимаются как десятичные. Перед буквами (для шестнадцатеричной системы счисления) необходимо написать символ «нуль»:

```
1110b
7o
7d
0Eh
```

Числа могут храниться как в оперативной памяти компьютера, так и в регистрах процессора. Фактически регистры тоже являются внутрипроцессорной сверхбыстрой оперативной памятью, однако в принято разделять эти понятия. К ним относятся такие регистры, как AX, BX, CX, DX, BP, SI, DI, SP и другие. Каждый из них имеет своё название и назначение. Так, они подразделяются на несколько категорий. Для обработки и хранения чисел используются в первую очередь регистры общего назначения: AX, BX, CX и DX. Их названия, надо отметить, не случайны, а являются мнемоническим отражением тех функций, для которых они зачастую используются (accumulator, base, count, data).

Каждый из них имеет размер 2 байта, что позволяет работать с числами в диапазоне от 0 до 65535 (в беззнаковом представлении). Однако каждый из них, в свою очередь делится на старший и младший регистр, что позволяет работать уже с двумя числами, пусть и в меньшем диапазоне – от 0 до 255 (в беззнаковом представлении). Однако, как уже было сказано, регистры являются не единственным средством хранения числовых данных. Гораздо удобнее хранить их в оперативной памяти компьютера, а сами регистры использовать для вычислений. Оперативная память представлена ячейками памяти, каждая из которых имеет размер, равный одному байту. Каждая такая ячейка имеет свой адрес, что позволяет обратиться к ней и получить соответствующее числовое значение.

## 1.2. СИМВОЛЫ

Как уже было сказано, числа чрезвычайно удобны для кодирования. Приведённый выше пример кодирования текста с помощью чисел, собственно говоря, реализован на примере настоящего отчёта по практике – каждый символ, который в нём можно видеть, для компьютера является каким-то числом. Очевидна необходимость ввода какого-то стандарта кодирования символов – чтобы, закодировав такой текст на одной машине, мы могли бы декодировать его на другой, не потеряв и не исказив при этом его значение. Таким стандартом стала разработанная в 1963 году таблица ASCII. В ней наиболее распространённым печатным и непечатным символам были сопоставлены числовые коды. Исходная кодировка включала 128 символов, позднее таблица была расширена до 256 символов, что позволило использовать национальные символы (в том числе кириллицу). Следует отметить также, что в таблице были представлены не только печатные, но и управляющие символы.

ASCII легла в основу созданного позднее стандарта Unicode, который включал несметно большее количество символов, тем не менее, коды символов, содержащихся, в ASCII не изменились, что обеспечило совместимость. ASCII представлена ниже, по вертикали представлены первые цифры кодов соответствующих символов, по горизонтали – вторые (в шестнадцатеричном формате).

**ASCII Code Chart**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL | BS | HT | LF | VT | FF | CR | SO | SI |
| 1 | DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | CAN | EM | SUB | ESC | FS | GS | RS | US |
| 2 | | ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - | . | / |
| 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 4 | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 5 | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | _ |
| 6 | ` | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| 7 | p | q | r | s | t | u | v | w | x | y | z | { | \| | } | ~ | DEL |

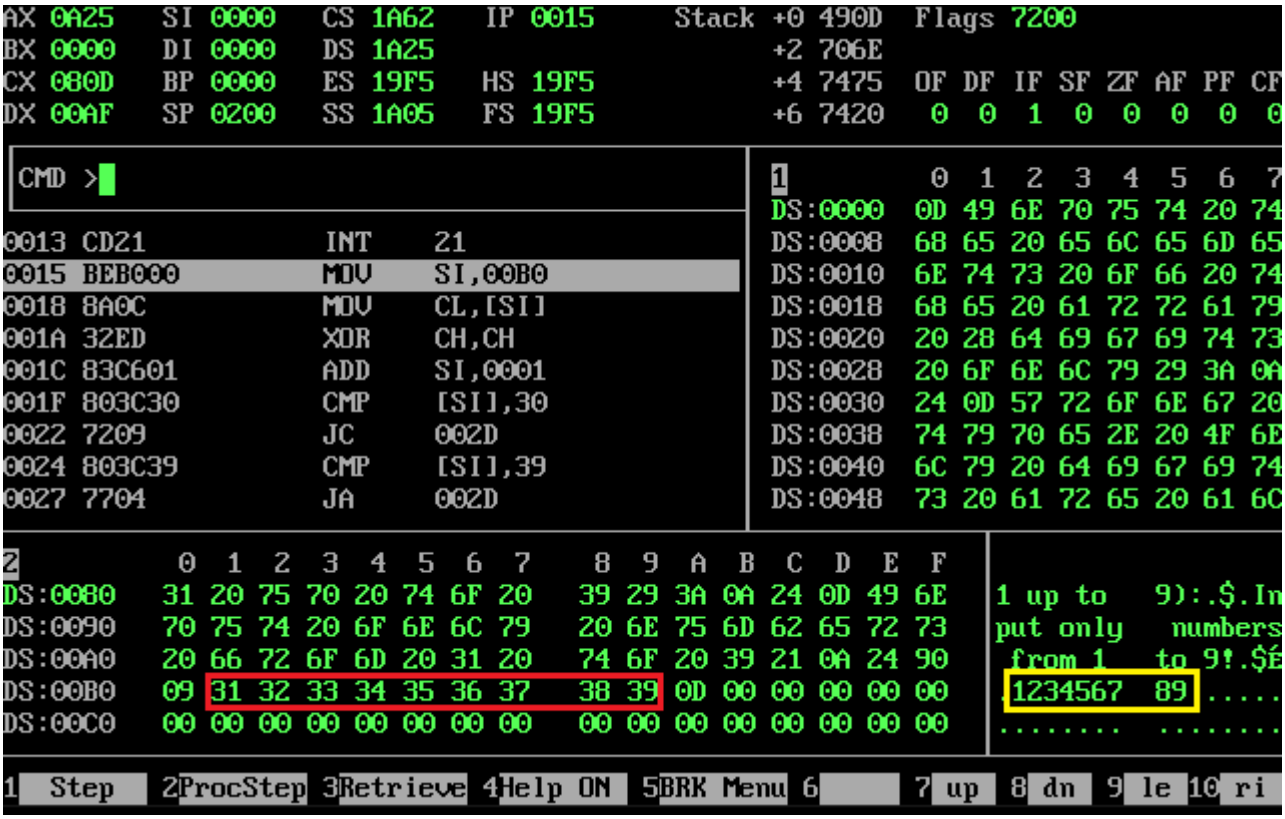
Такой стандарт позволяет нам, к примеру, закодировать адрес сайта «bmstu.ru» как совокупность символов

```
62h 6Dh 73h 74h 75h 2Eh 72h 75h
```

## 1.3. СТРОКИ, ОДНОМЕРНЫЕ И МНОГОМЕРНЫЕ МАССИВЫ

В большинстве современных высокоуровневых языках программирования строки и массивы являются различными типами данных (например, «char» и «string» в C++). Однако, так было не всегда. Так, ранее в высокоуровневых языках программирования строки представляли собой одномерный массив символов. Учитывая, что для определения как строки, так и массива имеет значение состав и порядок элементов, текстовую строку теоретически можно рассматривать как частный случай массива.

В языке ассемблера, однако, такой тип данных отсутствует. В нём строку или массив можно определить как совокупность байтов, хранящихся в оперативной памяти. Важно отметить, что данные при этом располагаются последовательно. Такое их расположение позволяет использовать инструкции процессора для ввода и вывода всей строки (массива) целиком, без обращения по отдельности к каждому байту. Ниже содержится пример введённой строки: в оперативной памяти располагаются коды символов «123456789» (отмечены красным цветом). Эти же символы в привычном нам представлении выделены жёлтым.



Благодаря последовательному расположению этих символов в памяти, их очень удобно можно вывести с помощью цикла, используя в качестве счётчика длину строки (её значение содержится по адресу DS:00B0) с помощью функции 02 прерывания 21h, однако ещё удобнее сделать это, разместив в ячейке памяти по адресу DS:00BAh символ «$» (а, если быть точнее, его ASCII-код) и воспользоваться функцией 09 того же прерывания. Использование же управляющих символов позволяет выводить, к примеру, двумерные массивы (скажем, числовые матрицы). Обладая определёнными навыками, представляется возможным даже вывод трёхмерных массивов данных (при небольшом числе элементов, входящих в массив), однако массивы таких (и тем более выше) размерностей, видимо, всё же целесообразно выводить, используя соответствующие символы (скобки, запятые).

# 2. АЛГОРИТМЫ

## 2.1. ПОСИМВОЛЬНАЯ ОБРАБОТКА СТРОК

Рассмотрим более детально предложенные способы вывода элементов строки, но прежде обратим внимание на то, как может осуществляться ввод строк. Как и в примере выше, мы можем либо вводить строку посимвольно, используя цикл. Например, если выделенный сегмент данных позволяет нам разместить строку по адресу DS:00B0h и она состоит из двадцати символов, то мы можем использовать следующий код:

```
MOV BX, 00B0h
MOV CX, 020d
MOV AH, 01h

Input:
INT 21h
INC BX
MOV byte ptr [BX], AL
loop Input
```

Однако, он содержит определённые недостатки. Так, мы вынуждены вводить строго определённое количество символов, процессор выполняет избыточные операции, а для подсчёта количества введённых символов нам пришлось бы дописать данный код. Всех этих недостатков можно избежать, используя другую функцию прерывания 21h – функцию 0Ah. Удобство её использования наглядно показывает код ниже:

```
MOV DX, 00B0h
MOV AH, 0Ah
INT 21h
```

Такой код делает всё то же самое и даже больше: теперь без ввода дополнительных счётчиков из ячейки DS:00B0h можно получить количество введённых символов, а мы можем ввести нужное количество символов (разумеется, если ранее мы предусмотрели размер сегмента данных).

Ситуация с выводом строки выглядит аналогично. Допустим, что в ячейке DS:00B0h у нас уже есть количество введённых символов. Тогда эту строку можно выводить посимвольно:

```
XOR CX, CX
MOV BX, 00B0h
MOV CL, byte ptr [BX]

Output:
INC BX
MOV DL, byte ptr [BX]
MOV AH, 02h
INT 21h
loop Output
```

Однако предпочтительнее всё же использовать другую функцию прерывания 21h. Это позволит тоже сократить размер кода и повысить удобство. Выглядеть такой вывод будет следующим образом:

```
MOV DX, 00B0h
MOV AH, 0Ah
INT 21h
```

В этом случае, однако важно не забыть использовать символ «$», разместив его в ячейке памяти по адресу, следующим за адресом ячейки, в которой находится последний символ строки.

## 2.2. ВВОД/ВЫВОД ЧИСЛА

Итак, для ввода и вывода строки мы используем прерывание 21h. Это же прерывание используется и для ввода и вывода числа. В то же время, операции ввода и вывода числа используют функции, отличные от функций обработки строк. Обратимся к описанному нами ранее коду, внеся в него, однако, некоторые изменения, поскольку теперь он предназначается исключительно для ввода одного символа

```
MOV BX, 00B0h
MOV AH, 01h
INT 21h
MOV byte ptr [BX], AL
```

ASCII-код введённого нами символа появится по адресу DS:00B0h. Следует отличать ASCII-код числа от самого числа. Таблица в разделе 1.2. настоящего отчёта подчёркивает эту разницу – в первых двух строках располагаются коды управляющих символы, а цифра 0 имеет код 30h. С другой стороны, вывод цифры не представляет трудности. Например, если в регистре AL у нас есть число «3», его можно вывести, прибавив к нему 30h и воспользовавшись функцией 02h прерывания 21h.

```
MOV AL, 03h
MOV DL, AL
ADD DL, 30h
MOV AH, 02h
INT 21h
```

Нужно, однако, отметить, что работа с двух- и более разрядными числами обстоит несколько сложнее. В зависимости от количества разрядов предстоит умножать цифру соответствующего разряда на нужную степень числа «10», а если такое число больше 255, то нужно ещё и правильно записать его в память.

## 2.3. ПОИСК МИНИМУМА/МАКСИМУМА В МАССИВЕ

Язык ассемблера не содержит удобных функций поиска минимума и максимума в массиве, свойственных языкам высокого уровня. Поиск таких элементов выполняется посредством использования цикла и посимвольного сравнения. Ниже рассмотрим пример кода, выполняющего поиск этих значений в текстовой строке, количество элементов которой находится в ячейке с адресом DS:00B0h (и равно девяти), а сами элементы – в ячейках DS:00B0h – DS:00B9h. Поиск максимального значения элемента строки можно выполнить следующим образом:

```
    XOR AL, AL
    MOV BX, 00B0h
    MOV CL, byte ptr [BX]

    SearchMax:
    INC BX
    CMP AL, byte ptr [BX]
    JA Skip
    MOV AL, byte ptr [BX]
    Skip:
    loop SearchMax
```

Рассмотрим работу данного кода. В цикле увеличивается значение смещения адреса элемента строки на единицу, что позволяет сравнивать все элементы, каждый элемент с помощью команды CMP сравнивается со значением числа, хранящегося в регистре AL (значение которого мы предусмотрительно сделали равным нулю). В случае, если число в регистре AL оказывается больше того, с которым мы сравнивали элемент строки, следует переход к следующей итерации. Если же это условие не выполняется, машина вынуждена записать такое значение в регистр AL, после чего также переходит к

следующей итерации. Таким образом, по окончании цикла в регистре AL окажется максимальное число.

Поиск минимального значения выполняется аналогично, однако, с некоторыми различиями. Прежде всего видится необходимым записать в регистр AL максимально большое число. Другое отличие – в команде условного перехода. Вместо «JA», как в примере выше, здесь мы будем использовать «JB». В остальном алгоритм поиска остаётся без изменений.

```
MOV AL, 0FFh
MOV BX, 00B0h
MOV CL, byte ptr [BX]

SearchMin:
INC BX
CMP AL, byte ptr [BX]
JB Skip
MOV AL, byte ptr [BX]
Skip:
loop SearchMin
```

## 2.4. ПОСТРОЧНАЯ ОБРАБОТКА МАТРИЦЫ НА ПРИМЕРЕ ПОИСКА ЧЁТНЫХ ЭЛЕМЕНТОВ

Итак, ранее было отмечено, что массивы в памяти представляют собой последовательность элементов строки, чья размерность равна, разумеется, единице. Тем не менее, как мы выяснили, это не мешает представить данные в виде, к примеру, двумерной матрицы, если использовать управляющие символы (в частности, символ перевода строки), а для более комфортного восприятия информации в качестве разделителей элементов каждой строки матрицы использовать пробелы. Рассмотрим задачу поиска чётных элементов в строке матрицы и замену их на пробелы. Будем считать, что наша матрица представляет собой двумерный массив из 9 чисел (3x3), нулевой элемент (количество элементов) которой находится по адресу DS:00B0h, а остальные элементы – по адресам 00B1h – 00B9h.

```
XOR CX, CX
XOR DX, DX
MOV AH, 01h
MOV BX, 00B0h
MOV DH, 3
MOV CL, DH

OutputArray:
MOV DL, 0Ah
INT 21h

PUSH CX
MOV CL, DH
OutputString:
INC BX
TEST [BX], 1
```

```
JNZ Odd
Even:
MOV DL, “ “
INT 21h
JMP Skip

Odd:
MOV DL, byte ptr [BX]
INT 21h

Skip:
loop OutputString

POP CX
loop OutputArray
```

Используемая в данном коде команда TEST выставляет флаг чётности (Parity Flag), его анализирует команда условного перехода JNZ и далее, в зависимости от того, оказалось число чётным или нет, следует вывод соответственно пробела или исходного числа. Надо отметить, что ASCII построена таким образом, что позволяет сравнивать на чётность сразу коды чисел, получая корректный результат. Именно поэтому нам не пришлось прибавлять/вычитать число 30h – очевидно, что, поскольку оно чётное, на чётность анализируемого числа это не влияет.

# 3. ОБРАБОТКА ПРОЦЕССОРОМ ВЫЗОВА ПОДПРОГРАММЫ И ВОЗВРАТА ИЗ ПОДПРОГРАММЫ

Ранее мы рассмотрели процедуру вывода матрицы с заменой чётных элементов на пробелы. Данная схема является рабочей, однако, имеет свои недостатки. Исходная матрица в памяти компьютера не изменяется и если мы заходим вывести изменённую матрицу повторно, нам снова придётся применять весь изложенный алгоритм. Гораздо удобнее было бы выделить в сегменте данных место для хранения изменённой матрицы и выводить уже, собственно, изменённую матрицу. В случае, если мы изменяем исходную матрицу несколько раз, очевидно, что процедура вывода изменённой матрицы остаётся прежней. В подобных случаях является предпочтительным использование подпрограмм или, как их ещё называют, процедур. Описание процедур позволяет сократить объём кода, что положительно сказывается на его читаемости. В качестве иллюстрации использования подпрограмм будем использовать следующий код (условимся, что ранее мы записали обработанную матрицу, включая символы пробела и перевода строки, начиная с ячейки по адресу DS:0116h, а после всех символов такого массива поставили «$»):

```
CALL OutPutProg
JMP Continue

OutputProg proc
MOV AH, 09h
MOV DX, 0116h
INT 21h
RET
OutputProg endp

Continue:
```

Рассмотрим её работу подробнее. Команда `CALL OutPutProg` вызывает процедуру под названием OutPutProg, код которой содержится ниже, но ещё до этого в стеке она сохраняет адрес следующей команды, к которому должен вернуться компьютер. Чтобы процедура не выполнялась дважды, после команды её вызова следует команда безусловного перехода, которая минует описание данной подпрограммы. В самой же процедуре используется функция 09h прерывания 21h, используемая для вывода строки. После выполнения вывода, из стека загружается адрес возврата, компьютер выполняет операцию с кодом `JMP Continue`, а далее следует, например, выполнение другого алгоритма обработки исходной матрицы и запись изменённой матрицы по тому же адресу, что и в первый раз, что позволяет снова вывести изменённую матрицу, используя ту же команду `CALL OutPutProg`. Во второй раз описание подпрограммы не требуется (и даже является лишним). В частности, в этом и заключается удобство их использования.

# ЗАКЛЮЧЕНИЕ

Рассмотренные в настоящем отчёте примеры являются подтверждением того, что код на языке ассемблера существенно отличается от кода высокоуровневых языков программирования. Писать на нём программы несколько сложнее, чем на популярных языках высокого уровня, а написанные программы занимают больше строк, чем высокоуровневые языки. Однако, вместе с тем очевидно, что именно использование языка ассемблера позволяет получить наиболее полный доступ к ресурсам компьютера и, в теории, писать максимально эффективный код – как по объёму программ, так и по быстродействию. В то же время компиляторы высокоуровневых языков продолжают совершенствоваться и разница в быстродействии между ними и языком ассемблера становится всё менее заметной. Несмотря на это, его изучение является крайне важным в образовательных целях, позволяя получить представление об архитектуре компьютера и о наборе инструкций, которые использует процессор. Кроме этого, ассемблерные вставки всё же ещё достаточно широко используются в определённых целях – там, где критически важно быстродействие или же где существует серьёзное ограничение по объёму используемой памяти. Всё это лишний раз подтверждает необходимость изучения как языка ассемблера в частности, так и данной дисциплины – «машинно-зависимые языки программирования» - в целом.