Лекция 3. Зри в корень… Behold at the root

Эпиграф.

«Отыщи всему начало  
И ты многое поймешь!»  
                Козьма Прутков

Find the beginning of everything

And you will understand a lot!

Упражнение 501. Сразу после прочтения ответить на вопрос:

Есть фрагмент текста на языке записи алгоритмов assembler

MOV CX, 0

Povt: LOOP Povt

ВОПРОС: ВНИМАТЕЛЬНО ЧИТАЙТЕ ВОПРОС!!! НЕ ОТВЕЧАЙТЕ С БУХТЫ-БАРАХТЫ!!!!!

ВОПРОС: СКОЛЬКО РАЗ БУДЕТ ИСПОЛНЕНА ЭВМ-ой КОМАНДА LOOP? Ответ, как всегда в начале лекции в течение 5 минут прислать мне на электронную почту!!!!

**НИКАКОЙ РЕГИСТР НЕ МОЖЕТ БЫТЬ ЗНАКОВЫМ ИЛИ БЕЗНАКОВЫМ ДО ТЕХ ПОР, ПОКА КАКАЯ-ЛИБО КОМАДА НЕ НАЧНЕТ ИСПОЛЬЗОВАТЬ ЭТОТ РЕГИСТР.**

***Напоминание.***

***Толкование 6. Программа*** - это последовательность команд или операторов, которая после расшифровки ее ВЫЧИСЛИТЕЛЬНОЙ МАШИНОЙ может заставить эту же МАШИНУ выполнить определенную работу. **Программное Обеспечение** - это группа взаимодействующих друг с другом программ.

**Толкование 7. *Вычислительная машина (ВМ)*** - это устройство, которое автоматически и непрерывно производит вычисления, выбирая свои действия из множества заранее установленных команд. Универсальная вычислительная машина - это ВМ, команды которой записаны в память и могут быть достаточно быстро заменены там другими командами без инженерного вмешательства человека.

***Толкование 8.*** ***Операционная система*** - совокупность программных и аппаратных средств, предназначенных для управления устройствами ЭВМ и предоставлению пользователям определенного сервиса по управлению этими устройствами. Любая ОС состоит как минимум из трех элементов: файловая подсистема, базовая система ввода-вывода, командный язык.

***Замечание 1. В связи с толкованиями 6 и 7 в рамках данного курса лекций запрещается произносить словосочетания типа : «Язык программирования «такой-то», так как можно «программировать» только кого-либо или что-либо, а ассемблер, С++, Java, ПЛ1, Фортран, Паскаль, Ада, Модула, Пайсон и т.д. и т.п. СЛУЖАТ НЕ ДЛЯ ПРОГРАММИРОВАНИЯ ЭВМ, а представляют собой средства записи алгоритмов, которые (записи) после обработки специальными программными средствами (трансляторами, компиляторами, интерпретаторами) становятся программами – последовательностями команд, которые после их расшифровки ЭВМ могут заставить ЭВМ выполнить определенные действия. То есть все существующие языки представляют собой НЕ ЯЗЫКИ ПРОГРАММИРОВАНИЯ ЭВМ, а языки записи алгоритмов. Причем то, что получается в результате записи алгоритма на языке, будем называть исходным текстом на языке записи алгоритмов.***

***Замечание 2. В связи с толкованием 8 в рамках данного курса лекций запрещается произносить словосочетания типа: «Операционная система Java», «Операционная система Windows» и т.д и т.п., так как классическое определение «Операционной системы» предполагает включение в качестве элементов системы «аппаратных средств». Следует говорить «Операционная оболочка Windows», «Программное средство Windows».***

Продолжение повторения.

На одной из прошлых лекций шла речь о том, что бывают Badware. Принципы, которые положены в основу сих «произведений» можно изучить на примере построения считающегося первым вирусом в истории компьютерной техники – Dhog68, который представлен на стр. 256 -258 книги О.В. Бурдаев, М.А. Иванов, И.И. Тетерин. Ассемблер в задачах защиты информации. Однако, прежде чем просто посмотреть на текст этого вируса, написанного на языке записи алгоритмов assembler, конечно, же необходимо исследовать ОБЪЕКТ заражения – электронно-вычислительную машину и так называемую «операционную систему» - совокупность АППАРАТНЫХ и программных средств, ПРЕДНАЗНАЧЕННЫХ ДЛЯ УПРАВЛЕНИЯ устройствами ЭВМ и предоставления сервиса пользователям, КОТОРЫМИ ВЫСТУПАЮТ не КОНЕЧНЫЕ ПОЛЬЗОВАТЕЛИ (люди, сидящие за компьютером), а ПРИКЛАДНЫЕ ПРОГРАММЫ (совокупности команд или инструкций, которые после декодирования их ЭВМашиной могут заставить эту ЭВМ выполнить некоторую последовательность действий). А также необходимо, основываясь на опыте, накопленном человечеством в борьбе с различной заразой, хотя бы кратко ознакомиться с имеющимися подходами к классификации badware. Например, той, которая предложена на стр. 244 указанного выше первоисточника. А на ст.р. 87 – 125 книги Емельяновой и др. подробно рассмотрены программно-технические угрозы компьютерной безопасности. В настоящее время наиболее известной системой в области защиты данных является то, что делается в лаборатории Касперского. Поэтому есть смысл ориентироваться именно на неё, что будет сделано далее.

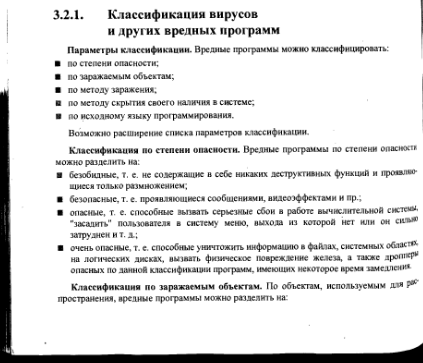
***Толкование 101. Вирус.*** Хотя многие по-прежнему используют слово «вирус» по отношению к любым вредоносным программам, на самом деле оно относится лишь к самовоспроизводящемуся коду, который копирует себя из файла в файл на одном компьютере. Сейчас вирусы, которые некогда были основным типом угроз, стали редким явлением. Однако есть и любопытные исключения, и недавно нам попалось одно из них: это первый настоящий вирус в дикой среде за долгое время.

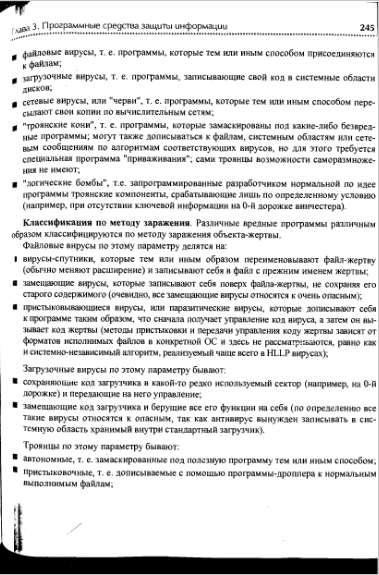
Вирус под названием [KBOT](https://securelist.ru/kbot-sometimes-they-come-back/95486/) проникает на компьютер жертвы через интернет, локальную сеть или зараженный внешний носитель. После запуска зараженного файла зловред прочно закрепляется в системе, прописывая себя в автозагрузку и планировщик заданий, а затем с помощью веб-инжектов пытается похитить банковские и персональные данные пользователя. Также KBOT может загружать дополнительные модули, которые похищают важные сведения о жертве, такие как логины и пароли, данные криптокошельков, перечни файлов и установленных приложений и т. п., и пересылают их на командный сервер. Все свои файлы и похищенные данные KBOT хранит в виртуальной файловой системе, зашифрованной с помощью алгоритма RC6, что затрудняет обнаружение вируса

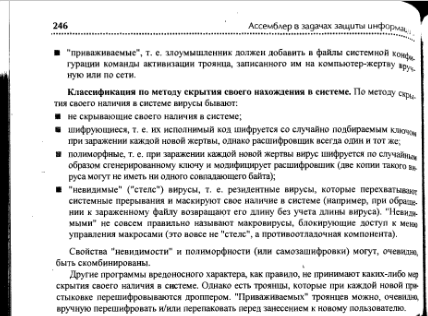
[<https://securelist.ru/it-threat-evolution-q1-2020/95979/?utm_source=threats.kaspersky.com&utm_medium=blog&utm_campaign=news_block>]

***Но если сравнить то, что говорится о «зловредах» в настоящее время, с тем, что говорилось ранее (см. ниже), легко видеть, что за последние 15 лет мало что изменилось с точки зрения классификации этих зловредов и с точки зрения принципов их функционирования и принципов противодействия им. Далее представлена возможная классификация badware, представленная в книге про ассемблер в задачах защиты данных. А в работах лаборатории Касперского можно найти ещё одну классификацию:*** <https://www.kaspersky.ru/resource-center/threats/viruses-malware>***,***

<https://www.kaspersky.ru/resource-center/threats/malware-classifications>







***Замечание.*** Книга про ассемблер издана в 2004 году, однако, как мы увидим далее, в деле обеспечения безопасности информационных систем НИЧЕГО НЕ ИЗМЕНИЛОСЬ – по-прежнему основные угрозы реализуются посредством использования badware – нехорошего программного обеспечения, которое в конечном счете представляет собой совокупность программ, после декодирования которых ЭВМ сама по себе выполняет нехорошую для себя последовательность действий.

Чтобы «не фонтанировать» той информацией, которую можно легко найти на «помойке», вернёмся к «нашим баранам» - принципам построения badware (malware – ещё так говорят), но сначала следует поговорить о «предмете заражения» - электронно-вычислительной машине, под которой понимается, как мы уже знаем:

**Толкование 7. *Вычислительная машина (ВМ)*** - это устройство, которое автоматически и непрерывно производит вычисления, выбирая свои действия из множества заранее установленных команд. Универсальная вычислительная машина - это ВМ, команды которой записаны в память и могут быть достаточно быстро заменены там другими командами без инженерного вмешательства человека.

***Очевидно, что основными понятиями, которые есть в этом толковании, являются понятия «устройство» (процессор), производящее вычисления автоматически и непрерывно, и «память», в которую записаны команды вычислительной машины и, конечно, же данные записаны в память тоже.***

Смысл работы ЭВМ состоит в следующем:

1. Команды и данные вводятся с внешних устройств в оперативную память посредством использования так называемых драйверов устройств.
2. Процессор по определенным правилам выбирает из оперативной памяти КОМАНДУ, расшифровывает ее, после чего достает из оперативной памяти данные в соответствии с заложенными в расшифрованной команде правилами и производит определенные в выбранной команде ДЕЙСТВИЯ.
3. Результаты выполнения команды (полученные данные) процессор пересылает в определенное в команде место Оперативной Памяти, которые (данные) посредством использования драйверов устройств могут быть размещены на внешних устройствах ЭВМ.

Из сказанного со всей очевидностью следует, что любая «зараза» для реализации своих чёрных замыслов обязательно должна попасть в оперативную память, должна представлять собой в конечном итоге совокупность команд, после расшифровки которых самой ЭВМ, они (команды) заставят эту ЭВМ выполнить действия, приводящие к различным несанкционированным законным пользователем последствиям.

***Толкование 502.*** ПАМЯТЬ - устройство ЭВМ, предназначенное для хранения обрабатываемой информации (КОМАНД и ДАННЫХ) и представляющее собой совокупность большого количества физических элементов, имеющих два устойчивых состояния. Одно из них называется состоянием 0 , другое - состоянием 1. В каждый отдельный момент времени элемент находится в каком-либо из этих состояний. Указанные физические элементы в программировании называют РАЗРЯДАМИ или БИТАМИ.

Слово бит - по-английски пишется bit и является сокращением двух английских слов: Binary Digit.

Графически сказанное можно представить так

bbbbbbbbbbb

П А М Я Т Ь = bbbbbbbbbbb

bbbbbbbbbbb

, где b - bit

Данные и команды, которые должны находиться в памяти (см. определение универсальной вычислительной машины), представлены в этой памяти в виде этих ноликов и единиц. Но как? По-видимому существуют какие-то строго определенные правила, придуманные, наверное, дядьками с IBM (скоро, наверное, некоторые будут говорить: «Знал бы, кто это придумал – убил бы!»). Эти строго определенные правила позволяют сказать, например, что такой набор ноликов и единиц в оперативной памяти:

0100 0001

в одном случае - представляет собой символ-букву 'А '

в другом случае - число, равное 65

в третьем случае - команду "сложить".

При этом сразу запоминаем один из краеугольных камней разработки программ для ЭВМ, который говорит и больно говорит, что все зависит от того, ГДЕ находится эта совокупность 0 и 1 и КАКОЙ к этой совокупности осуществляется ДОСТУП!

***Толкование 503.*** БАЙТОМ называется набор из стандартного числа (из 8) битов, обрабатываемых как единое целое. ТЕТРАДА - совокупность из 4 битов.

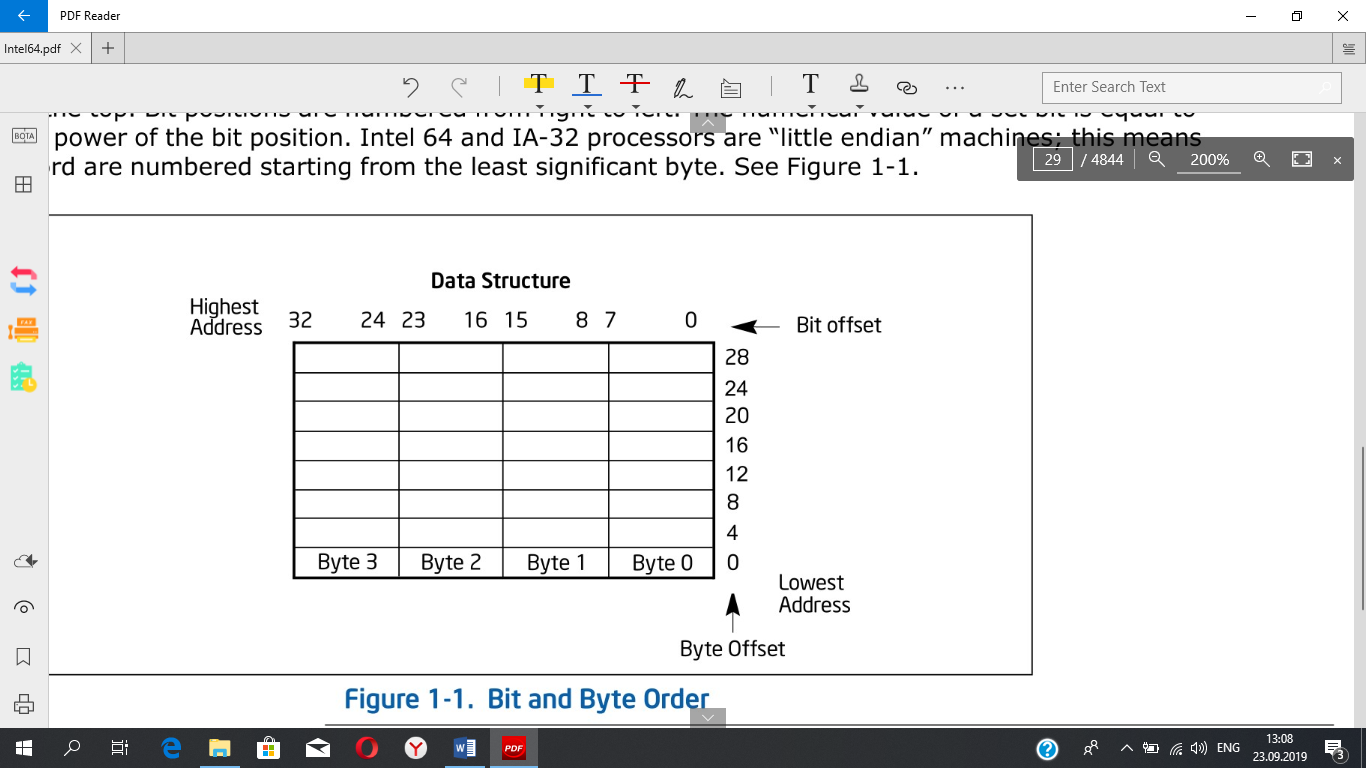
СЛОВО - совокупность из 2 байтов.

ДВОЙНОЕ СЛОВО - совокупность из 4 байтов.

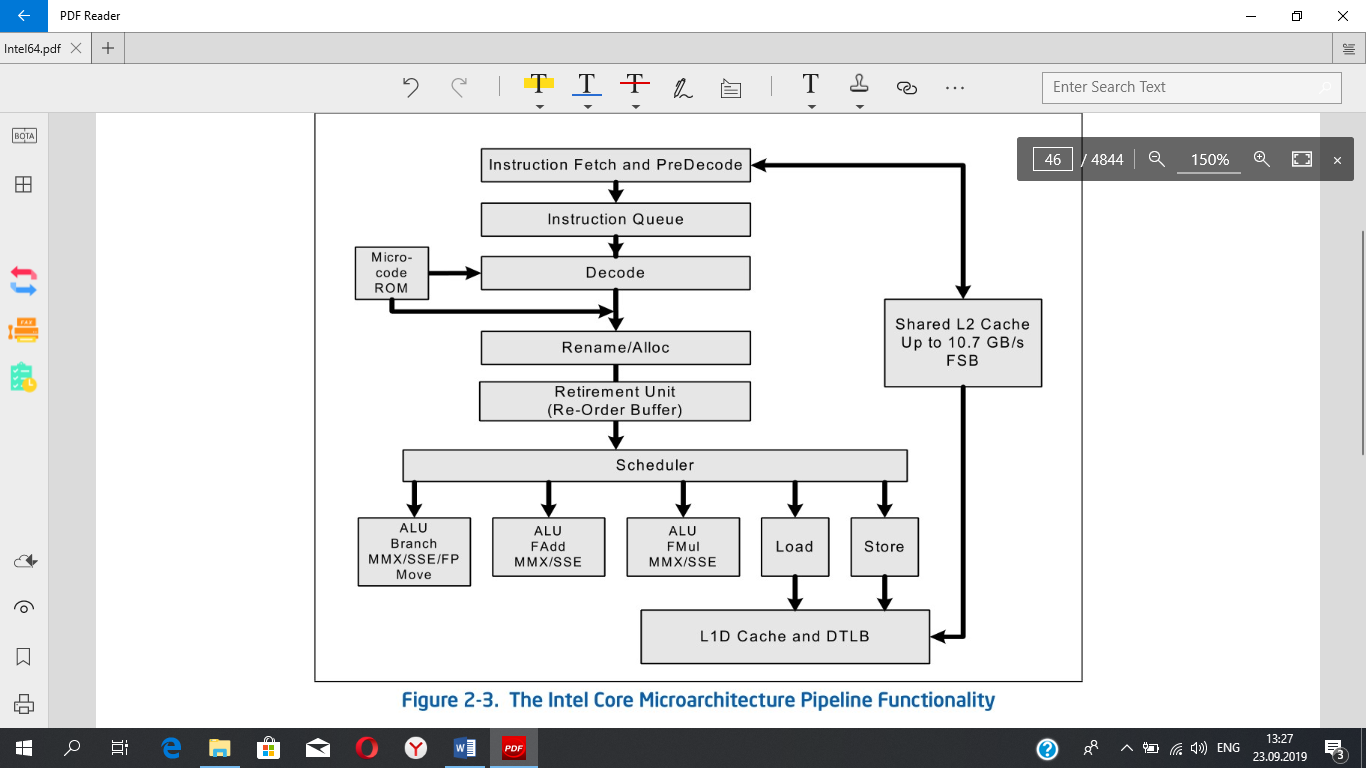
***Толкование 503”. АДРЕС*** представляет собой  ***число*** , определяющее ***номер***  некоторого **байта в Оперативной Памяти.**

***Толкование 504.*** Intel technologies features and benefits depend on system configuration and may require enabled hardware, software, or service activation**. Learn more at intel.com**. ***No computer system can be absolutely secure.***

In illustrations of data structures in memory, smaller addresses appear toward the bottom of the figure; addresses increase toward the top. Bit positions are numbered from right to left. The numerical value of a set bit is equal to two raised to the power of the bit position. Intel 64 and IA-32 processors are “little endian” machines; this means the bytes of a word are numbered starting from the least significant byte. See Figure 1-1.

+

***Толкование 505.*** Figure shows a conceptual view of the processor microarchitecture.



Смысл работы Микропроцессора состоит в следующем:

1. По содержимому регистров, предназначенных для формирования адреса команд, определяется адрес команды, которая должна быть выполнена.

2. Вычисленный адрес команды помещается в Оперативную Память.

3. Из оперативной памяти выбирается вычисленный на предыдущем шаге адрес команды, расшифровывается. После чего осуществляется доступ к команде, записанной по этому адресу. Выбранная из памяти команда пересылается в Процессор.

4. Выбранная на предыдущем шаге Команда помещаются в очередь команд. Процессор подвергает каждую команду расшифровке и поочередному выполнению в следующем режиме:

4.1. Если для выполнения команды требуются ДАННЫЕ, то АДРЕС их месторасположения вычисляется в блоке преобразования адресов по информации, содержащейся в АДРЕСНОЙ ЧАСТИ КОМАНДЫ, по БАЗОВЫМ адресам СЕГМЕНТОВ, а также по адресам, содержащимся в называемых адресных регистрах.

4.2. Данные, считанные из основной памяти, пересылаются в регистр данных или в АЛУ и обрабатываются в соответствии с тем, что сказано в команде.

4.3. Результаты обработки помещаются в какую-либо область ОП или в какой-либо из регистров.

***Толкование 505”. BASIC EXECUTION ENVIRONMENT***

This chapter describes the basic execution environment of an Intel 64 or IA-32 processor as seen by assembly language programmers. ***It describes how the processor executes instructions and how it stores and manipulates data. The execution environment described here includes memory (the address space), general-purpose data registers, segment registers, the flag register, and the instruction pointer register.***

***The IA-32 architecture*** supports three basic operating modes: protected mode, real-address mode, and system management mode. The operating mode determines which instructions and architectural features are accessible:

***•Protected mode*** — this mode is the native state of the processor. Among the capabilities of protected mode is the ability to directly execute “real-address mode” 8086 software in a protected, multi-tasking environment. This feature is called virtual-8086 mode, although it is not actually a processor mode. Virtual-8086 mode is actually a protected mode attribute that can be enabled for any task.

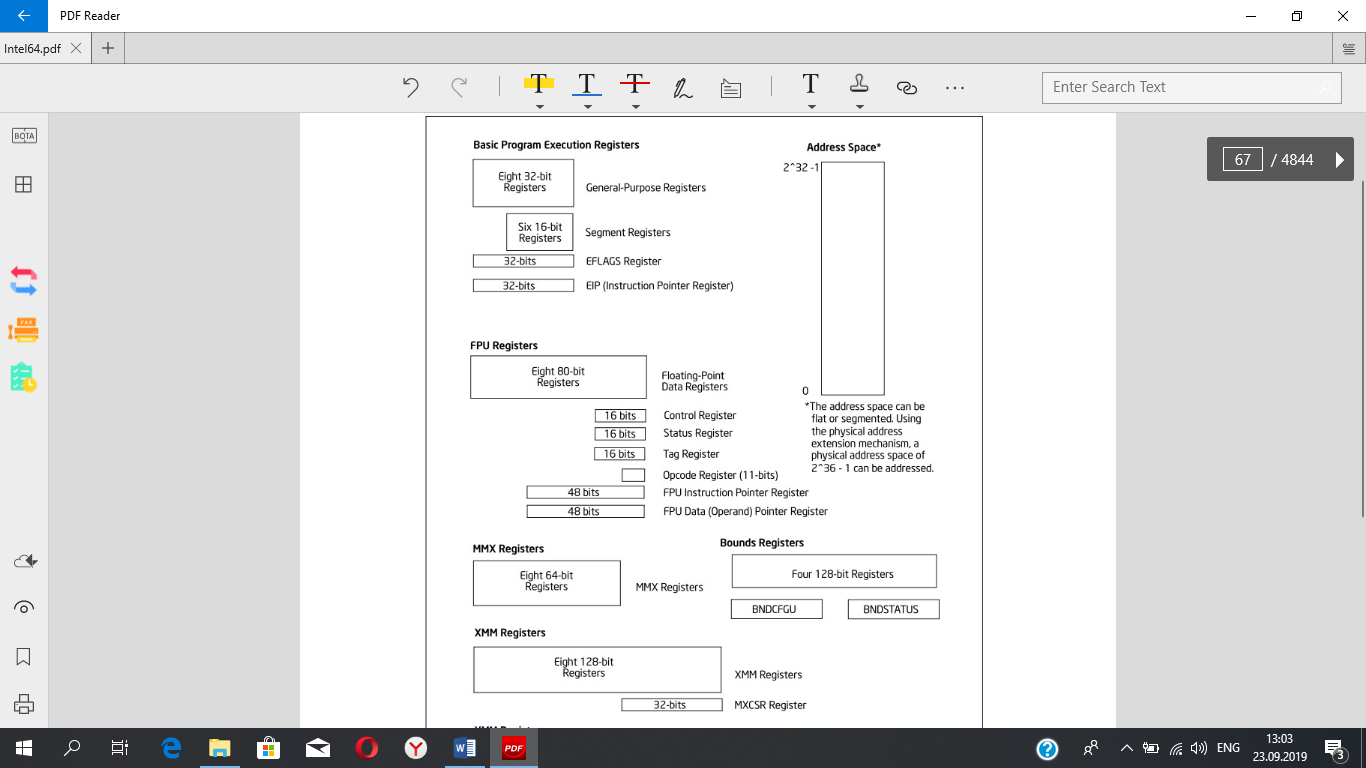
***•Real-address mode*** — this mode implements the programming environment of the Intel 8086 processor with extensions (such as the ability to switch to protected or system management mode). The processor is placed in real-address mode following power-up or a reset.

***•System management mode*** (SMM) — this mode provides an operating system or executive with a transparent mechanism for implementing platform-specific functions such as power management and system security. The processor enters SMM when the external SMM interrupt pin (SMI#) is activated or an SMI is received from the advanced programmable interrupt controller (APIC).

In SMM, the processor switches to a separate address space while saving the basic context of the currently running program or task. SMM-specific code may then be executed transparently. Upon returning from SMM, the processor is placed back into its state prior to the system management interrupt. SMM was introduced with the Intel386™ SL and Intel486™ SL processors and became a standard IA-32 feature with the Pentium processor family.

***OVERVIEW OF THE BASIC EXECUTION ENVIRONMENT***

Any program or task running on an IA-32 processor is given a set of resources for executing instructions and for storing code, data, and state information. These resources (described briefly in the following paragraphs and shown in Figure 3-1) make up the basic execution environment for an IA-32 processor.



An Intel 64 processor supports the basic execution environment of an IA-32 processor, and a similar environment under IA-32e mode that can execute 64-bit programs (64-bit sub-mode) and 32-bit programs (compatibility sub-mode).

The basic execution environment is used jointly by the application programs and the operating system or executive running on the processor.

**•Address space** — Any task or program running on an IA-32 processor can address a linear address space of up to 4 GBytes (232 bytes) and a physical address space of up to 64 GBytes (236 bytes).

***•Basic program execution registers*** — the eight general-purpose registers, the six segment registers, the EFLAGS register, and the EIP (instruction pointer) register comprise a basic execution environment in which to execute a set of general-purpose instructions. These instructions perform basic integer arithmetic on byte, word, and doubleword integers, handle program flow control, operate on bit and byte strings, and address memory.

***x87 FPU instruction pointer register***, the x87 FPU operand (data) pointer register, the x87 FPU tag register, and the x87 FPU opcode register provide an execution environment for operating on single-precision, double-precision, and double extended-precision floating-point values, word integers, doubleword integers, quadword integers, and binary coded decimal (BCD) values. See Section 8.1, “x87 FPU Execution Environment,” for more information about these registers.

***•MMX registers*** — the eight MMX registers support execution of single-instruction, multiple-data (SIMD) operations on 64-bit packed byte, word, and doubleword integers.

***•XMM registers*** — the eight XMM data registers and the MXCSR register support execution of SIMD operations on 128-bit packed single-precision and double-precision floating-point values and on 128-bit packed byte, word, doubleword, and quadword integers.

***•YMM registers*** — The YMM data registers support execution of 256-bit SIMD operations on 256-bit packed single-precision and double-precision floating-point values and on 256-bit packed byte, word, doubleword, and quadword integers.

***•Bounds registers*** — each of the BND0-BND3 register stores the lower and upper bounds (64 bits each) associated with the pointer to a memory buffer. They support execution of the Intel MPX instructions.

•***BNDCFGU and BNDSTATUS***— BNDCFGU configures user mode MPX operations on bounds checking. BNDSTATUS provides additional information on the #BR caused by an MPX operation

***•Stack*** — to support procedure or subroutine calls and the passing of parameters between procedures or subroutines, a stack and stack management resources are included in the execution environment. The stack (not shown in Figure 3-1) is located in memory.

In addition to the resources provided in the basic execution environment, the IA-32 architecture provides the following resources as part of its system-level architecture. They provide extensive support for operating-system and system-development software. Except for the I/O ports, the system resources are described in detail in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 3A & 3B.

***•I/O ports*** — The IA-32 architecture supports transfers of data to and from input/output (I/O) ports. See Chapter 18, “Input/Output,” in this volume.

***•Control registers*** — the five control registers (CR0 through CR4) determine the operating mode of the processor and the characteristics of the currently executing task.

***•Memory management registers*** — The GDTR, IDTR, task register, and LDTR specify the locations of data structures used in protected mode memory management.

***•Debug registers*** — the debug registers (DR0 through DR7) control and allow monitoring of the processor’s debugging operations.

***•Memory type range registers (MTRRs)*** — The MTRRs are used to assign memory types to regions of memory.

***•Machine specific registers (MSRs)*** — the processor provides a variety of machine specific registers that are used to control and report on processor performance. Virtually all MSRs handle system related functions and are not accessible to an application program. One exception to this rule is the time-stamp counter.

***•Machine check registers*** — the machine check registers consist of a set of control, status, and error-reporting MSRs that are used to detect and report on hardware (machine) errors.

**Performance monitoring counters** — the performance monitoring counters allow processor performance events to be monitored. ***The processor uses byte addressing.*** This means memory is organized and accessed as a sequence of bytes. Whether one or more bytes are being accessed, a byte address is used to locate the byte or bytes memory. The range of memory that can be addressed is called an address space. The processor also supports segmented addressing. This is a form of addressing where a program may have many independent address spaces, called segments. For example, a program can keep its code (instructions) and stack in separate segments. Code addresses would always refer to the code space, and stack addresses would always refer to the stack space. The following notation is used to specify a byte address within a segment:

For example, the following segment address identifies the byte at address FF79H in the segment pointed by the DS register: DS:FF79H

The following segment address identifies an instruction address in the code segment. The CS register points to the code segment and the EIP register contains the address of the instruction. CS:EIP

***An exception*** is an event that typically occurs when an instruction causes an error. For example, an attempt to divide by zero generates an exception. However, some exceptions, such as breakpints, occur under other conditions. Some types of exceptions may provide error codes. An error code reports additional information about the error. An example of the notation used to show an exception and error code is shown below:

#PF(fault code)

This example refers to a page-fault exception under conditions where an error code naming a type of fault is reported. Under some conditions, exceptions that produce error codes may not be able to report an accurate code. In this case, the error code is zero, as shown below for a general-protection exception: #GP(0)

***The memory that the processor addresses*** on its bus is called physical memory. Physical memory is organized as a sequence of 8-bit bytes. Each byte is assigned a unique address, called a physical address. The physical address space ranges from zero to a maximum of 236 − 1 (64 GBytes) if the processor does not support Intel 64 architecture.

Virtually any operating system or executive designed to work with an IA-32 or Intel 64 processor will use the processor’s memory management facilities to access memory. These facilities provide features such as segmentation and paging, which allow memory to be managed efficiently and reliably. The following paragraphs describe the basic methods of addressing memory when memory management is used.

***IA-32 Memory Models***

When employing the processor’s memory management facilities, programs do not directly address physical memory. Instead, they access memory using one of three memory models: flat, segmented, or real address mode:

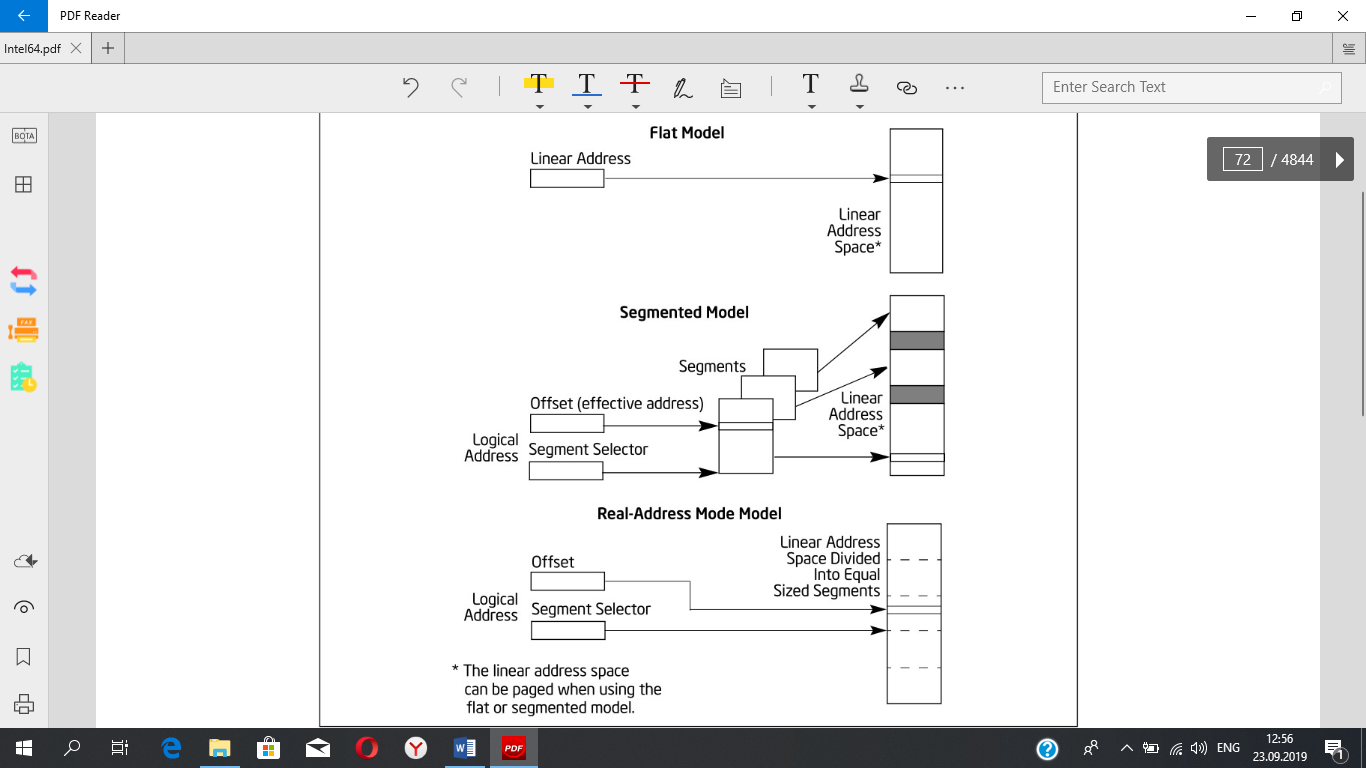
***•Flat memory model*** — Memory appears to a program as a single, continuous address space (Figure 3-3). This space is called a linear address space. Code, data, and stacks are all contained in this address space. Linear address space is byte addressable, with addresses running contiguously from 0 to 232 - 1 (if not in 64-bit mode). An address for any byte in linear address space is called a linear address.

***•Segmented memory model*** — Memory appears to a program as a group of independent address spaces called segments. Code, data, and stacks are typically contained in separate segments. To address a byte in a segment, a program issues a logical address. This consists of a segment selector and an offset (logical addresses are often referred to as far pointers). The segment selector identifies the segment to be accessed and the offset identifies a byte in the address space of the segment. Programs running on an IA-32 processor can address up to 16,383 segments of different sizes and types, and each segment can be as large as 232 bytes.

Internally, all the segments that are defined for a system are mapped into the processor’s linear address space. To access a memory location, the processor thus translates each logical address into a linear address. This translation is transparent to the application program.

The primary reason for using segmented memory is to increase the reliability of programs and systems. For example, placing a program’s stack in a separate segment prevents the stack from growing into the code or data space and overwriting instructions or data, respectively.

***•Real-address mode memory model*** — this is the memory model for the Intel 8086 processor. It is supported to provide compatibility with existing programs written to run on the Intel 8086 processor. The real-address mode uses a specific implementation of segmented memory in which the linear address space for the program and the operating system/executive consists of an array of segments of up to 64 KBytes in size each. The maximum size of the linear address space in real-address mode is 220 bytes.



When writing code for an IA-32 or Intel 64 processor, a programmer needs to know the operating mode the processor is going to be in when executing the code and the memory model being used. The relationship between operating modes and memory models is as follows:

•Protected mode — When in protected mode, the processor can use any of the memory models described in

this section. (The real-addressing mode memory model is ordinarily used only when the processor is in the

virtual-8086 mode.) The memory model used depends on the design of the operating system or executive.

When multitasking is implemented, individual tasks can use different memory models.

•Real-address mode — When in real-address mode, the processor only supports the real-address mode

memory model.

•System management mode — When in SMM, the processor switches to a separate address space, called the

system management RAM (SMRAM). The memory model used to address bytes in this address space is similar

to the real-address mode model.

•Compatibility mode — Software that needs to run in compatibility mode should observe the same memory

model as those targeted to run in 32-bit protected mode. The effect of segmentation is the same as it is in 32-

bit protected mode semantics.

•64-bit mode — Segmentation is generally (but not completely) disabled, creating a flat 64-bit linear-address

space. Specifically, the processor treats the segment base of CS, DS, ES, and SS as zero in 64-bit mode (this

makes a linear address equal an effective address). Segmented and real address modes are not available in 64-

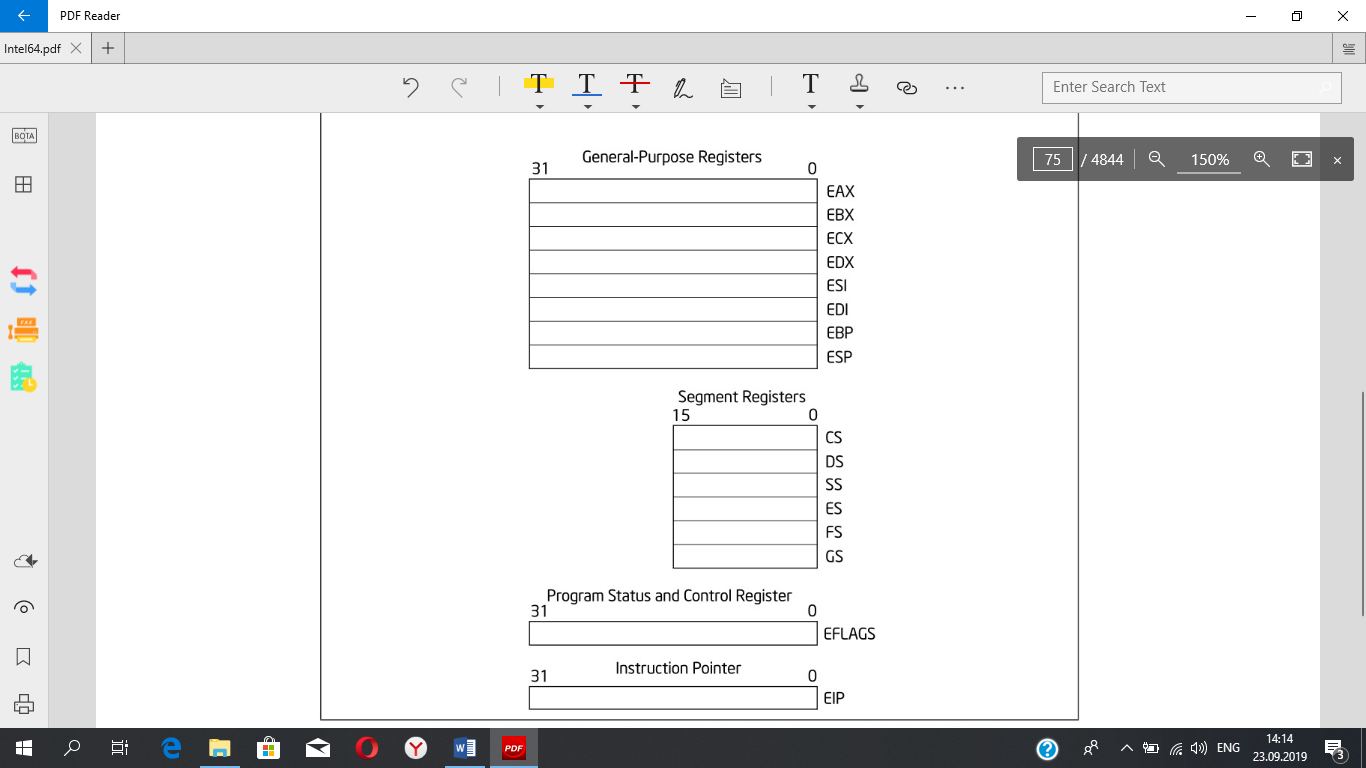
bit mode.

***Execution 501. In English, please. Would you like to write everything you know about BUS and PORT!!!!***

***BASIC PROGRAM EXECUTION REGISTERS***

IA-32 architecture provides 16 basic program execution registers for use in general system and application

programming (see Figure 3-4). These registers can be grouped as follows:



The following is a summary of special uses:

•EAX — Accumulator for operands and results data

•EBX — Pointer to data in the DS segment

•ECX — Counter for string and loop operations

•EDX — I/O pointer

•ESI — Pointer to data in the segment pointed to by the DS register; source pointer for string operations

•EDI — Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for

string operations

•ESP — Stack pointer (in the SS segment)

•EBP — Pointer to data on the stack (in the SS segment)

As shown in Figure, the lower 16 bits of the general-purpose registers map directly to the register set found in

the 8086 and Intel 286 processors and can be referenced with the names AX, BX, CX, DX, BP, SI, DI, and SP. Each

of the lower two bytes of the EAX, EBX, ECX, and EDX registers can be referenced by the names AH, BH, CH, and

DH (high bytes) and AL, BL, CL, and DL (low bytes).

***In 64-bit mode,*** there are 16 general purpose registers and the default operand size is 32 bits. However, general-purpose registers are able to work with either 32-bit or 64-bit operands. If a 32-bit operand size is specified: EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D - R15D are available. If a 64-bit operand size is specified: RAX, RBX, RCX,

RDX, RDI, RSI, RBP, RSP, R8-R15 are available. R8D-R15D/R8-R15 represent eight new general-purpose registers.

All of these registers can be accessed at the byte, word, dword, and qword level. REX prefixes are used to generate

64-bit operand sizes or to reference registers R8-R15. Registers only available in 64-bit mode (R8-R15 and XMM8-XMM15) are preserved across transitions from 64-bit mode into compatibility mode then back into 64-bit mode. However, values of R8-R15 and XMM8-XMM15 are undefined after transitions from 64-bit mode through compatibility mode to legacy or real mode and then back through compatibility mode to 64-bit mode.

***Упражнение 502***. Сразу после прочтения всего вышеизложенного, ответить на вопрос, прислав ответ на него мне на почту. Вопрос: Нарисовать (используя Excel и прислав мне на почту файл)взаимосвязь (гробиками изобразить) между регистрами: AH, AL, AX, EAX, RAX!!!!!

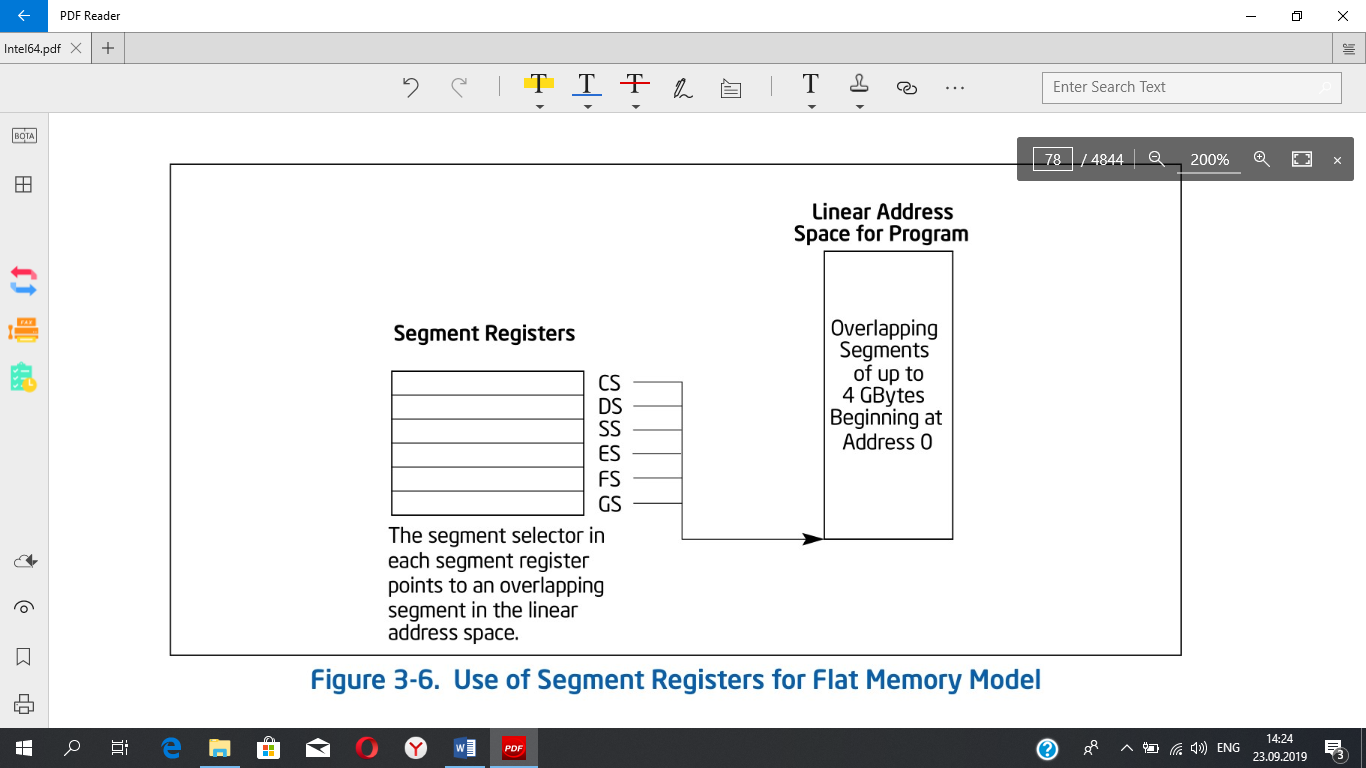
The ***segment registers*** (CS, DS, SS, ES, FS, and GS) hold 16-bit segment selectors. A segment selector is a special pointer that identifies a segment in memory. To access a particular segment in memory, the segment selector for that segment must be present in the appropriate segment register. When writing application code, programmers generally create segment selectors with assembler directives and symbols. The assembler and other tools then create the actual segment selector values associated with these directives and symbols. If writing system code, programmers may need to create segment selectors directly.

How segment registers are used depends on the type of memory management model that the operating system or executive is using. When using the flat (unsegmented) memory model, segment registers are loaded with segment

selectors that point to overlapping segments, each of which begins at address 0 of the linear address space (see

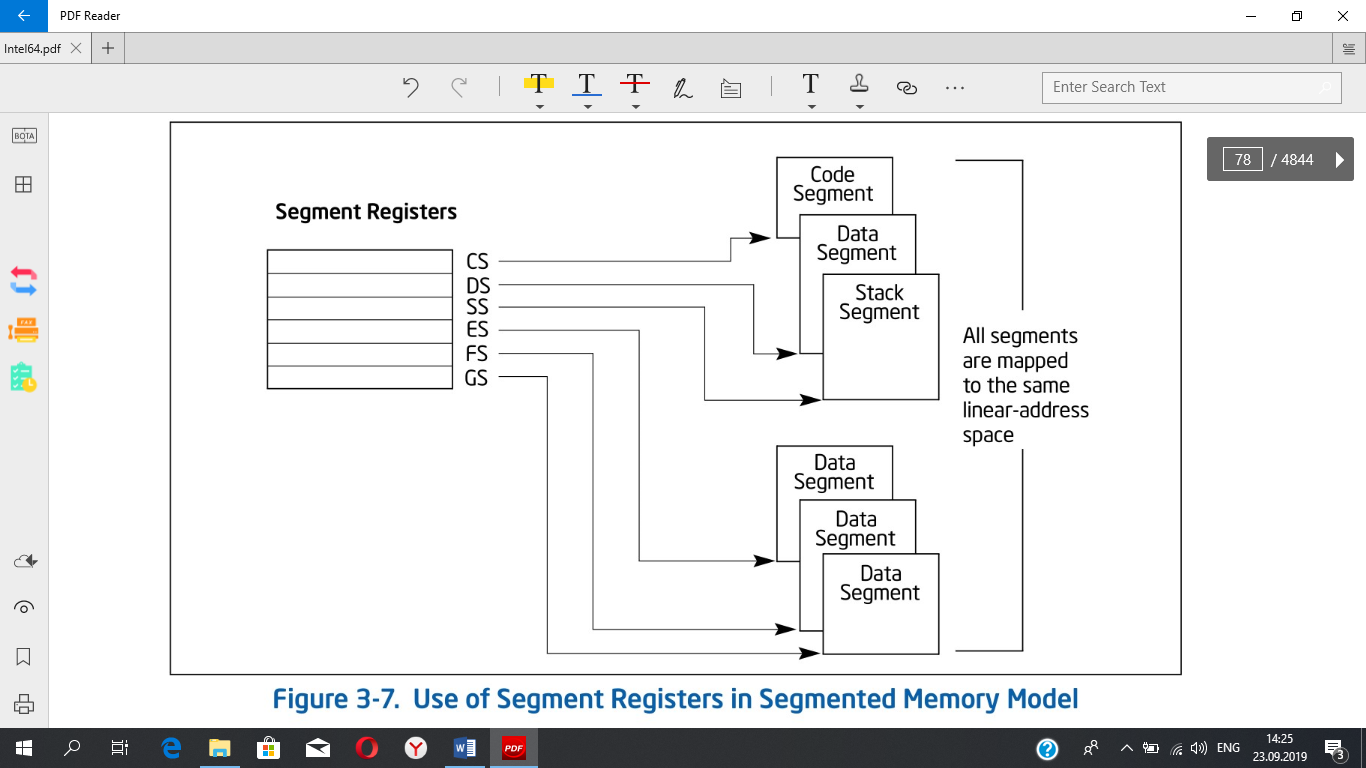
Figure 3-6).

***Упражнение 503 Перевести с английского на русский два последних абзаца и нарисовать, что такое selectors? Registers? Pointers? Перевод и рисунок прислать мне на почту сразу же после прочтения!!!!!***



These overlapping segments then comprise the linear address space for the program. Typically, two

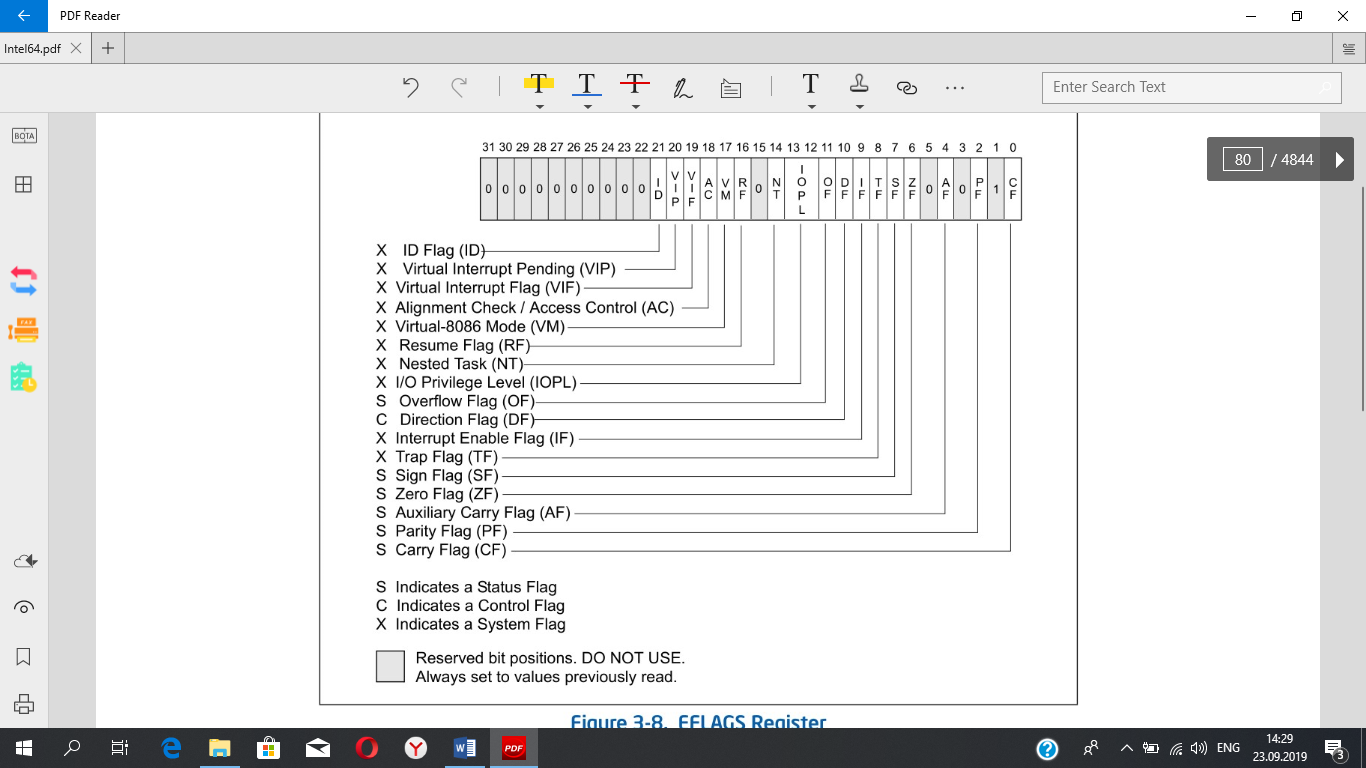
overlapping segments are defined: one for code and another for data and stacks. The CS segment register points

to the code segment and all the other segment registers point to the data and stack segment. When using the segmented memory model, each segment register is ordinarily loaded with a different segment selector so that each segment register points to a different segment within the linear address space. 

At any time, a program can thus access up to six segments in the linear address space. To access a

segment not pointed to by one of the segment registers, a program must first load the segment selector for the

segment to be accessed into a segment register.



The status flags (bits 0, 2, 4, 6, 7, and 11) of the EFLAGS register indicate the results of arithmetic instructions,

such as the ADD, SUB, MUL, and DIV instructions. The status flag functions are:

***CF (bit 0) Carry flag*** —

Set if an arithmetic operation generates a carry or a borrow out of the most-

significant bit of the result; cleared otherwise. This flag indicates an overflow condition for

unsigned-integer arithmetic. It is also used in multiple-precision arithmetic.

***PF (bit 2 ) Parity flag —***

Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise.

***AF (bit 4) Auxiliary Carry flag*** —

Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic.

***ZF (bit 6) Zero flag*** —

Set if the result is zero; cleared otherwise.

***SF (bit 7) Sign flag —***

Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)

***OF (bit 11) Overflow flag —***

Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two’s complement) arithmetic. Of these status flags, only the CF flag can be modified directly, using the STC, CLC, and CMC instructions. Also the bit instructions (BT, BTS, BTR, and BTC) copy a specified bit into the CF flag.

The status flags allow a single arithmetic operation to produce results for three different data types: unsigned integers, signed integers, and BCD integers. If the result of an arithmetic operation is treated as an unsigned integer,the CF flag indicates an out-of-range condition (carry or a borrow); if treated as a signed integer (two’s complement number), the OF flag indicates a carry or borrow; and if treated as a BCD digit, the AF flag indicates a carry or borrow. The SF flag indicates the sign of a signed integer. The ZF flag indicates either a signed- or an unsignedinteger zero.

When performing multiple-precision arithmetic on integers, the CF flag is used in conjunction with the add with carry (ADC) and subtract with borrow (SBB) instructions to propagate a carry or borrow from one computation to the next.

The condition instructions Jcc (jump on condition code cc), SETcc (byte set on condition code cc), LOOPcc, and CMOVcc (conditional move) use one or more of the status flags as condition codes and test them for branch, setbyte, or end-loop conditions.

DF Flag

The direction flag (DF, located in bit 10 of the EFLAGS register) controls string instructions (MOVS, CMPS, SCAS, LODS, and STOS). Setting the DF flag causes the string instructions to auto-decrement (to process strings from high addresses to low addresses). Clearing the DF flag causes the string instructions to auto-increment (process strings from low addresses to high addresses). The STD and CLD instructions set and clear the DF flag, respectively.

System Flags and IOPL Field

The system flags and IOPL field in the EFLAGS register control operating-system or executive operations. They should not be modified by application programs. The functions of the system flags are as follows:

TF (bit 8)Trap flag — Set to enable single-step mode for debugging; clear to disable single-step mode.

IF (bit 9)Interrupt enable flag — Controls the response of the processor to maskable interrupt requests. Set to respond to maskable interrupts; cleared to inhibit maskable interrupts.

IOPL (bits 12 and 13) I/O privilege level field — Indicates the I/O privilege level of the currently running program or task. The current privilege level (CPL) of the currently running program or task must be less than or equal to the I/O privilege level to access the I/O address space. The POPF and IRET instructions can modify this field only when operating at a CPL of 0.

NT (bit 14) Nested task flag — Controls the chaining of interrupted and called tasks. Set when the current task is linked to the previously executed task; cleared when the current task is not linked to another task.

RF (bit 16) Resume flag — Controls the processor’s response to debug exceptions.

VM (bit 17)Virtual-8086 mode flag — Set to enable virtual-8086 mode; clear to return to protected mode without virtual-8086 mode semantics.

AC (bit 18)Alignment check (or access control) flag — If the AM bit is set in the CR0 register, alignment checking of user-mode data accesses is enabled if and only if this flag is 1. If the SMAP bit is set in the CR4 register, explicit supervisor-mode data accesses to user-mode pages are allowed if and only if this bit is 1

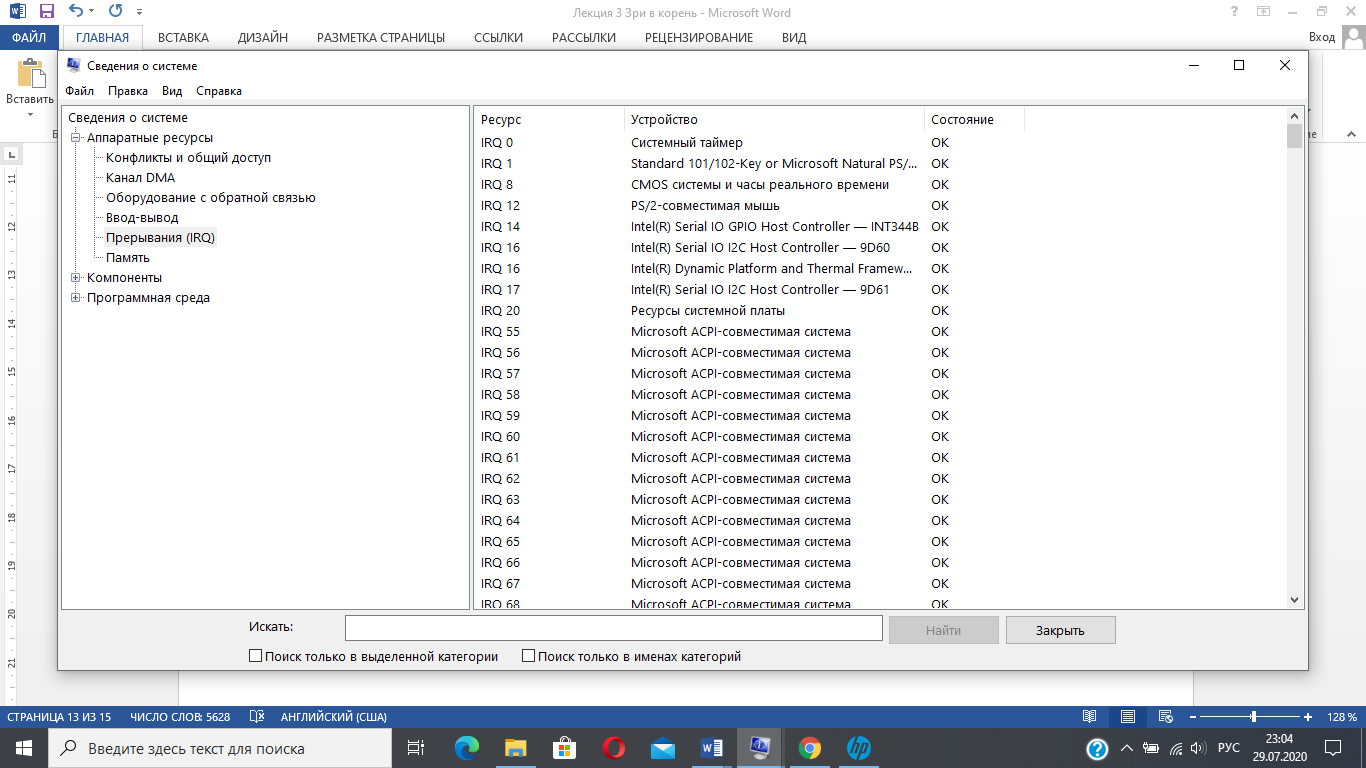
VIF (bit 19) Virtual interrupt flag — Virtual image of the IF flag. Used in conjunction with the VIP flag. (To use this flag and the VIP flag the virtual mode extensions are enabled by setting the VME flag in control register CR4.)

VIP (bit 20) Virtual interrupt pending flag — Set to indicate that an interrupt is pending; clear when no interrupt is pending. (Software sets and clears this flag; the processor only reads it.) Used in conjunction with the VIF flag.

ID (bit 21)Identification flag — The ability of a program to set or clear this flag indicates support for the CPUID instruction.

The instruction pointer (EIP) register contains the offset in the current code segment for the next instruction to be executed. It is advanced from one instruction boundary to the next in straight-line code or it is moved ahead or backwards by a number of instructions when executing JMP, Jcc, CALL, RET, and IRET instructions. The EIP register cannot be accessed directly by software; it is controlled implicitly by control-transfer instructions (such as JMP, Jcc, CALL, and RET), interrupts, and exceptions. The only way to read the EIP register is to execute a CALL instruction and then read the value of the return instruction pointer from the procedure stack. The EIP register can be loaded indirectly by modifying the value of a return instruction pointer on the procedure stack and executing a return instruction (RET or IRET). All IA-32 processors prefetch instructions. Because of instruction prefetching, an instruction address read from the bus during an instruction load does not match the value in the EIP register. Even though different processor generations use different prefetching mechanisms, the function of the EIP register to direct program flow remains fully compatible with all software written to run on IA-32 processors.

Из всего написанного выше и нарисованного на сайте разработчиков Intel, в связи с принятым в области развития вычислительной техники постулатом о том, что всё программное обеспечение, созданное для компьютеров с меньшей разрядностью, может быть использовано и в компьютерах с большей разрядностью ВНЕ ЗАВИСИМОСТИ ОТ ИСПОЛЬЗУЕМОЙ ОПЕРАЦИОННОЙ ОБОЛОЧКИ, и сравнивая с тем, что может быть сказано об архитектуре (и сказано в Лекциях Зубовича для студентов 1 курса факультета прикладной математики ещё в 2000 году) ЭВМ предыдущего 32-разрядного поколения ЭВМ, следует, что с точки зрения разработки программ с использованием языка записи алгоритмов assembler, мало что изменилось. Пожалуй, если не согласны – свистите. Отличие заключается лишь в том, что: а) вместе с двухбайтовыми регистрами (AX, BX, …, SI, DI….) могут быть использованы ещё четырехбайтовые (EAX, EBX, … , ESI, EDI…); б) аппарат прерываний (interrupt) «заменен» на некоторую «придумку» под названием «exception», что привело к появлению непонятных APIC в оболочках типа Windows 10.



И в связи с вышеизложенным, а также в связи с тем, что перед нами далее будет стоять задача разобраться со Спаем (а значит с понятием «клавиатура» и позже будет стоять задача рассмотреть принципы построения вирусов и антивирусных программ на примере первого вируса Dhog68, будем делать это так, как имеет место быть в оригинале (ассемблер в задачах защиты информации) с использованием команд, используемых в программах для 32-разрядных ЭВМ. А так как в последнее время почти повсеместно используются 64-разрядные ЭВМ с соответствующими им операционным оболочкам типа Windows 10, при эксплуатации которых в свою очередь возникают проблемы при попытках исполнения программ, использующих аппарат прерываний (насильственно навязывается аппарат exception и аппарат WinAPI (при разговоре об операционной оболочке Windows) - якобы с целью обеспечения большей безопасности для информационных систем ), разумным будет (в рамках изучения принципов построения нехороших программ) использование программных средств типа DosBox, моделирующих функционирование программ под управлением непосредственно операционной системы. Что и начнём делать в последующих лекциях. При этом во главу угла будем пытаться ставить тезис, основанный на том, что необходимо акцентировать внимание не на результате – получение вируса и антивируса, а на процессе – желании знать, по какому пути следует идти, чтобы разобраться в том, что значит «реализация систем защиты данных» на уровне, близком к аппаратному уровню. Можно даже сказать: «Результат – это ничто, процесс – это всё».

***Замечание.*** Вспоминая толкование «операционной системы»:

***Толкование 8.*** ***Операционная система*** - совокупность программных и аппаратных средств, предназначенных для управления устройствами ЭВМ и предоставлению пользователям определенного сервиса по управлению этими устройствами. Любая ОС состоит как минимум из трех элементов: файловая подсистема, базовая система ввода-вывода, командный язык –

необходимо отметить следующее. Почти во всех, написанных и изданных первоисточниках, имеющих в своем названии слова «операционная система», речь идёт ТОЛЬКО О ПРОГРАММНЫХ СРЕДСТВАХ, предназначенных для управления устройствами ЭВМ и предоставлению пользователям сервиса по управлению этими устройствами. Но ничего не говорится об аппаратной части подобных систем и о том, что пользователями сервиса являются не прикладные программы, а конечные пользователи вычислительной техники. При таком определении (без аппаратной части и ориентацией на конечного пользователя) любое программное средство (прикладная программа), в той или иной степени, управляющее устройствами ЭВМ и предоставляющее услуги пользователям, может быть названо «операционной системой» (например, чем не «операционная система» - DosBox? Или LINUX?). В связи с вышеизложенным в данном замечании, как мне кажется, есть смысл говорить не об «операционной системе» Windows, а об «операционной оболочке» Windows. Автор данного замечания, к сожалению, или, наоборот, к счастью, не знает названия «реальной операционной системы», которая используется в современных компьютерах, но это точно не Windows. Но он знает, что использование словосочетания «операционная система Windows», как и использование других словосочетаний со словами «операционная система», являются следствием маркетинговой политики различных разработчиков программных средств для вычислительной техники и специалистов, документирующих возможности подобных средств, с целью извлечения дополнительной прибыли.

***Упражнение 504.*** На лабораторном занятии про MessageBox была решена задача по выводу его на экран. В исходном тексте есть команда:

**mov message+6,'h'**

Используя любые доступные Вам средства, указать:

1. КАКОЙ АДРЕС имеет эта команда,
2. нарисовать шестнадцатеричное представление этой команды в памяти ЭВМ,
3. указать содержимое регистра EIP, когда исходный текст будет оттранслирован с получением объектного модуля, который в свою очередь будет далее пролинчеван с получением на внешнем устройстве исполнимого модуля, который в свою очередь будучи загружен операционной системой в оперативную память превратится в программу, которой будет передано управление, и когда в конце концов дойдёт дело до исполнения этой команды.
4. А также указать в шестнадцатеричном виде АДРЕС ОБЛАСТИ ПАМЯТИ, поименованной как message.

CDBYMZ: Под доступными средствами понимаются разного рода отладчики, трансляторы, получающие разного рода листинги.

***Ответ НА ЭТО УПРАЖНЕНИЕ МОЖНО ПРИСЫЛАТЬ ДО КОНЦА СЛЕДУЮЩЕГО ВТОРНИКА!***