Лаб. работа № 1: «Команды языка Assembler.»

Структура программы

```
;===[ Начало сегмента кода ]=============================================
MYCODE: segment .code
org 100h      ; Обязательная директива ТОЛЬКО для COM-файлов
START:        ;---[ Точка старта ]----------------------------------------------------------------

              ; ... Здесь должно быть тело программы
              ; -  комментарий до конца строки

              ;---[ Стандартное завершение программы ]---------------------------------------
              mov AX, 4C00h
              int 21h

;===[ Начало сегмента данных ]===========================================
;         ...
```

Работа с ПО ”NASM 2.07” (Netwide ASseMbler):

!MYFILE.ASM	- исходный файл с текстом вашей программы
!MYFILE.COM	- исполняемый COM-файл на машинном языке (размер не более 64 Кб)
!MYFILE.OBJ	- объектный файл (необходим для создания исполняемого EXE -файла)
!MYFILE.EXE	- откомпилированный исполняемый EXE-файл на машинном языке
NASM.EXE	- компилятор из *.ASM в *.OBJ и *.COM
ALINK.EXE	- компоновщик объектного кода (создает файлы *.EXE из *.OBJ)
INSIGHT.COM	- дебаггер: позволяет пошагово выполнить (трассировать) *.COM и *.EXE

1. Создайте на диске D: или E: директорию с номером группы, а в ней – со своей фамилией
2. В силу того, что для компиляции и трассировки файлов требуется задавать длинные командные строки с множеством параметров, ваша работа будет сводиться лишь к запуску специально подготовленных BAT-файлов в нужном порядке:

1.Редактировать !myfile	- открывает текст программы в блокноте
2.Компиляция COM-файла	- компилирует COM-файл
3.Компиляция EXE-файла	- компилирует OBJ- и EXE-файл
4.Трассировка COM-файла	- пошаговая трассировка программы
5.Трассировка EXE-файла	- пошаговая трассировка программы

Горячие клавиши:

F2	– поставить/снять точку останова (прекращает автоматическое выполнение)
F4	– запустить автоматическое выполнение до строки на позиции курсора
F7	– пошаговая трассировка (каждый раз выполняется только одно действие)
F8	– покомандная трассировка (не заходим внутрь циклов и подпрограмм)
F10	– верхнее меню
Ctrl-F9	– запустить автоматическое выполнение до конца программы
Ctrl-F2	– сброс программы после завершения (снова можно запускать по **F7/F8**)
ALT-F5	– просмотр окна вывода результатов (черный экран)
ALT-X	– выход из дебаггера

Назначение окон редактора:

Центр слева - основное окно (сегмент кода, в который загружена программа)
Вверху справа - окно регистров (14 штук)
Чуть ниже - окно флагов
Еще ниже - часть стека
Внизу экрана - "дамп памяти" (сегмент данных)

Те значения, которые изменились при выполнении очередной инструкции, выделяются ярко бирюзовым. Значения регистров и флагов, проверяются в соответствующих окнах.

3. После того, как программа отлажена и одобрена преподавателем, скопируйте ее в резервную директорию и переименуйте. Файл **!MYFILE.ASM** предстоит использовать каждый раз для решения новой задачи (к нему идут все обращения из BAT-файлов).

На 1-й и 2-й Л/Р мы будем работать с COM-файлами. Их главное отличие от EXE: все манипуляции происходят только внутри одного сегмента памяти!!! Отсюда ограничение размера COM-файла = 64 Кб (MAX), минус служебный раздел 256 б (ORG 100h).

Прорешать всем (правильные результаты – на доску) и сохранить для последующих задач:

1) BX = 4*AX - научиться умножать на степени двойки путем сдвига (до этого что-то занести в AX)
2) BX = 5*AX - научиться умножать на другие числа путем разложение по степеням
3) AX = 12A8h → AX = 00A0h - научиться применять маски (оставить только нужную цифру)
4) AX = 12A8h → BX = A000h - научиться исп. сдвиги для перемещения цифры в другую позицию

Лаб. работа № 2: «Подпрограммы, Макросы и Библиотеки»

2.1. Подпрограмма – логически выделенная часть программы, которая имеет самостоятельный смысл.
Обычно подпрограммы используют в 3 случаях:

1. Программа содержит многократно повторяющиеся фрагменты кода;
2. Программа слишком громоздка для понимания, и ее необходимо разбить на подзадачи;
3. Программу разрабатывают несколько программистов, каждый реализует свою часть подзадач.

Пример использования подпрограммы в NASM (это типичная процедура – не возвращает никакого значения):

```
;===[ Начало сегмента кода ]=============================================
MYCODE: segment .code
org 100h      ; Обязательная директива ТОЛЬКО для COM-файлов
START:        ;---[ Точка старта ]----------------------------------------------------------

              MOV AX,     1111H
              MOV BX,     2222H
              MOV CX,     3333H
              MOV DX,     4444H
              CALL        MySubProg

              ;---[ Стандартное завершение программы ]---------------------------------------
              mov AX, 4C00h
              int 21h

MySubProg:
              ;---[ Начало подпрограммы ]----------------------------------------------------
              MOV AX,     5555H
              MOV BX,     6666H
              MOV CX,     7777H
              MOV DX,     8888H
              RET
              ;---[ Конец подпрограммы ]-----------------------------------------------------

;===[ Начало сегмента данных ]===========================================
;         ...
```

Комментарии:
Появились 2 новые команды:

CALL – переход по указанному имени подпрограммы (адресу в памяти)
RET – возврат на следующую строку после вызова (по адресу в «верхнем» элементе стека)

Если выполнять трассировку в дебаггере с помощью F8, то мы наблюдаем мгновенную смену значений в 4 регистрах. Если с помощью F7 – мы попадем “внутрь” нашей процедуры, выполним ее покомандно и вернемся обратно.

Обратите внимание, что подпрограмма располагается в сегменте кода после стандартного завершения программы, т.е. выполнить ее команды можно только путем вызова. Другим способом управление ей никогда не передастся!

Обратите внимание еще на одну особенность работы с подпрограммами. Как программа узнает, куда ей надо вернуться? Ведь таких вызовов может быть несколько! Команда **CALL** определяет адрес возврата (следующей за ней команды) и кладет его в стек. А команда **RET** – наоборот, извлекает адрес из стека и передает управление той команде, которая там находится.

В случае, если размер подпрограммы слишком большой, может потребоваться «дальний» возврат с помощью команды **RETF**

2.2. Создание макросов.

В случае, когда какие-то часто повторяющиеся фрагменты кода являются шаблонными, но не могут быть описаны с помощью одной подпрограммы (например, разные действия выполняются над одними и теми же данными или наоборот – одинаковые действия над разными данными), возможно гибко конструировать код с помощью макросов:

Пример: В AX дано число. Разбить его на шестнадцатеричные цифры, каждую из которых поместить в свой регистр.
Пример: AX = 1234H. Получить: BH = 1H, BL = 2H, CH = 3H, CL = 4H.

Решение без макросов		Решение с макросом
;===[Начало сегмента кода]========== **MYCODE: segment .code** **org 100h** ; Директива для COM-файлов **START:** ;---[Точка старта]---------- **MOV AX, 1234H**; Присвоим **MOV BH, AH** ; 12H **AND BH, 0F0H** ; 10H **SHR BH, 4** ; 01H - 1-я **MOV BL, AH** ; 12H **AND BL, 0FH** ; 02H - 2-я **MOV CH, AL** ; 34H **AND CH, 0F0H** ; 30H **SHR CH, 4** ; 03H - 3-я **MOV CL, AL** ; 34H **AND CL, 0FH** ; 04H - 4-я ;---[Стандартное завершение программы]-- **mov AX, 4C00h** **int 21h**	➔	**%macro MyGetNumber 4** ; 4 параметра **MOV %1, %2** ; Копир. 2 цифры **AND %1, %3** ; Получ. 1 цифру **SHR %1, %4** ; Сдвиг (на 0 или 4) **%endmacro** ;===[Начало сегмента кода]========== **MYCODE: segment .code** **org 100h** ; Директива для COM-файлов **START:** ;---[Точка старта]---------- **MOV AX, 1234H**; Присвоим **MyGetNumber BH,AH,0F0H,4** **MyGetNumber BL,AH, 0FH,0** **MyGetNumber CH,AL,0F0H,4** **MyGetNumber CL,AL, 0FH,0** ;---[Стандартное завершение программы]-- **mov AX, 4C00h** **int 21h**

Комментарии:
Макросы позволяют основательно сократить и разгрузить основной код – они “конструируют на лету“ требуемый фрагмент программы, собирая его из общей статической части и динамических вставок передаваемых параметров.
Описание макроса должно располагаться до его вызова (перед сегментом кода), при этом оно не занимает память!!!
По сути, мы создаем новую команду языка (**MyGetNumber**), которую можем использовать далее согласно описанию.
В нашем примере описан макрос, принимающий 4 параметра: регистр-приемник данных, регистр-источник данных, маску для отсечения нужной цифры и величину окончательного сдвига цифры.
Из них, в итоге, и формируются нужные блоки команд, которые вы увидите в дебаггере на местах вызовов макроса!
Важно: Если в макросе присутствуют метки, то его можно использовать лишь однократно, при повторной подстановке в программе получатся 2 и более меток с одинаковым именем и возникнет ошибка повторного описания.

Примечание:
Если в качестве параметра нужно передать символ запятая, то весь содержащий ее аргумент необходимо заключить в фигурные скобки.
Пример:
MyMacro NEG, CX
MyMacro MOV, {AX, BX}

Особенности использования подпрограмм и макросов:
На вызов подпрограммы затрачивается некоторое время: необходимо определить адрес возврата, положить его в стек, передать управление по адресу вызываемой подпрограммы.
При использовании макроса задержка на формирование кода происходит лишь на этапе компиляции, а исполняемый файл будет выполняться без дополнительных временных затрат и действий (т.е. чуть быстрее).

Для облегчения понимания исходного кода пригодны оба средства, но при трассировке в дебаггере в случае подпрограмм будут видны все их вызовы и возвраты (т.е. картина почти идентична тексту программы), в то время как макросы будут заменены на сформированные блоки команд (вся программа предстанет громоздким «монолитом»).

2.3. Создание библиотек.

Предположим, в вашей программе используется большое количество вспомогательных макросов и библиотек. Объем программного кода измеряется десятками страниц печатного текста. Разобраться в подобной «свалке» становится все труднее. Тут-то и придет на помощь механизм подключения библиотечных файлов! Он позволяет любой фрагмент кода вынести во внешний файл, который можно подключать в исходной программе одной командой:

```
        %include "!Macro.txt" ; Сюда спрятаны все макросы

;===[ Начало сегмента кода ]================================================
MYCODE: segment .code
org 100h    ; Обязательная директива ТОЛЬКО для COM-файлов
START:      ;---[ Точка старта ]-------------------------------------------------------------------

            MOV AX,     1111H
            MOV BX,     2222H
            MOV CX,     3333H
            MOV DX,     4444H
            CALL        MySubProg ; Вызов подпрограммы (уже из библиотеки)

            ;---[ Стандартное завершение программы ]-----------------------------------
            mov AX, 4C00h
            int 21h

            %include "!Library.txt" ; Сюда спрятаны все подпрограммы

;===[ Начало сегмента данных ]==============================================
;       ...
```

Директива %**include** подставляет вместо себя содержимое указанного файла. В нашем примере мы «спрятали» описание макросов в отдельный файл **!Macro.txt**, а подпрограммы – в файл **!Library.txt**

Таким образом, становится возможным разгружать основной файл кода, делать его более понятным, создавать многофайловые проекты и библиотеки часто используемых подпрограмм.

Так, имеет смысл делать два подключаемых файла:
1 – для описания всех макросов перед сегментом кода
2 – для описания всех подпрограмм после стандартного завершения

Лаб. работа № 3: «Работа с памятью ЭВМ»

Как уже говорилось ранее, память ЭВМ представляет собой очень большой одномерный массив из машинных слов (2 байта). Для удобства пользователя она поделена на сегменты - порции размером до 64 Кбайт. В COM-программах может использоваться только один сегмент, при этом все сегментные регистры ссылаются на него (в него загружается программа, там же хранятся данные, а в конце сегмента расположен стек). Собственно, поэтому и размер таких программ не может превышать 64 Кбайт.

Работать с памятью нам предстоит в нижнем окне дебаггера, перейти в которое можно с помощью комбинации клавиш **Shift-D** и далее стрелочками (возврат - **Esc**).

Итак, у нас есть 256 байт для работы в служебной области данных (отступ задан с помощью директивы **org 100h**). Мы можем обращаться к отдельным ее ячейкам, считывать и записывать в них значения с помощью знакомой команды:

```
mov byte  [0]     , 'H'   ; Записываем в 0ю ячейку один байт - символ 'H'
mov word [1]      , 'el'  ; Записываем в две ячейки (1ю и 2ю) одно слово - символы 'el'
mov word [3]      , 'lo'  ; Еще два символа
```

Если мы не напутали с адресами ячеек, то в нижнем окне сможем увидеть слово 'Hello'. После него идет некий «мусор» - набор каких-то непечатных символов. Их можно затирать, если в соответствующие ячейки записывать новые значения, например пробелы (код символа - 20h). Чтобы обратиться к произвольной ячейки памяти, можно использовать регистр-указатель **SI**:

```
mov SI            , 6     ; Выбираем ячейку сами (когда-то ее номер придется даже вычислять)
mov byte [SI]     , 20h   ; Записываем в нее символ пробел
mov byte [SI-1]   , 20h   ; Причем от ее номера можно задавать смещение…
mov byte [SI+2]   , 20h   ; … как в плюс, так и в минус на константу

mov BX            , 1     ; Только регистр BX допускается использовать в паре с SI
mov byte [SI+BX]  , 20h   ; За счет него возможно делать произвольное смещение
```

byte – тип данных, равный 1 байту (1 символ)
word – тип данных, равный 2 байтам, 1 слову (2 символа)
…

Задание 1 для всех (через подпрограмму)

Создать в памяти надпись «Hello, World!», начиная с позиции в памяти, равной номеру варианта.

Задание 2 для всех (через подпрограмму):

Создать надпись «Hello, World!» через **SI**, т.е обеспечить ей легкую перемещаемость на любую позицию в памяти.

Лаб. работа № 4: «Константы, переменные»

У нас есть 256 байт в служебной области для произвольных данных, но если требуется хранить больший объем или структурировать данные по типам, необходимо создать собственные элементы данных. Это делается после кода программы и всех подпрограмм.

```
;===[ Начало сегмента данных ]=========================================
        align 16, db 90h          ; Выравнивание по границе параграфа (90h = NOP)
        db '=[MYDATA BEGIN]='     ; 16-байтовая строка, покажет начало сегмента
str1    db    'Some Text'         ; Это текстовая строка (переменная с именем str1)
a       db    12h                 ; Это однобайтовая целочисленная переменная (с именем a)
b       dw    1234h               ; Это двухбайтовая целочисленная переменная (с именем b)
c       dd    12345678h           ; Это четырехбайтовая целочисленная переменная (с именем c)
MY      equ   123                 ; Это константа (с именем MY), размер автоопределяется
        align 16, db 32h          ; Выравнивание по границе нового параграфа (32h = пробел)
        times 16 db '='           ; Это тоже строка текста 16 раз повторяется символ равно)
```

Данные мы можем помещать в требуемом порядке. Переменные лучше называть маленькими буквами, а константы – большими. Еще одна особенность: переменные располагаются в ячейках памяти друг за другом, а константы, как и макросы, вообще не хранятся в памяти, их значения подставляются в код программы при компиляции (имена переменных заменяются на адреса)!

Примеры использования переменных и констант:

Константы можно использовать в любом месте в любое время:

```
        mov ax,      MY     ; Присвоить в AX значение константы MY (123 = 7Bh)

        mov bx,      word [b]     ; Теперь мы можем использовать эти имена,
        mov byte [a],      33h    ; они обращаются к нужным ячейкам памяти по именам,
                                  ; без необходимости запоминать их числовой адрес, как раньше!
        mov word [b],      4455h        ; Главное – не забывать про разрядность операндов!
        mov dword [c],     12345678h    ; А также – про порядок байт в памяти: ”55 44” и ”78 56 34 12”
```

Длинная арифметика

Данная концепция предполагает работу с длинными и очень длинными числами, отдельные части (цифры, байты) которых хранятся в смежных ячейках памяти. При этом корректное выполнение арифметических операций над ними должны обеспечивать специально написанные подпрограммы, учитывающие переносы между ячейками – при переполнениях, заимствованиях или сдвигах.

Задача 1: Получить в памяти такую комбинацию переменных и обрамляющих их текстов:

```
=[MYDATA BEGIN]=
A1A2 = [xx]……..
B1   = [x]……..
C1C2 = [xx]……..
```

(**x** – какие-то изначально присвоенные значения, все размерностью 1 байт)

Проскролить нижнее окно - область памяти за предел первого экрана можно комбинацией клавиш **Shift-D** и стрелочками. Выход обратно **Esc**.

Задача 2: Используя задачу 1, написать подпрограмму, которая складывает значения A1A2 + B и помещает результат в C1C2 (помните про порядок старшинства байт). При этом возможный перенос разряда из C2 в C1 нужно отслеживать и реализовывать самостоятельно!

Задача 3: Используя задачу 1, написать подпрограмму, которая вычитает из значения A1A2 значение B и помещает результат в C1C2 (помните про порядок старшинства байт). При этом возможное заимствование из C1 в C2 нужно отслеживать и реализовывать самостоятельно!

Задача 4: Используя задачу 1, написать подпрограмму, которая делает сдвиг значения A1A2 на 1 разряд влево (умножение на 2) и помещает результат в C1C2 (помните про порядок старшинства байт). При этом перенос старшего разряда из C2 в C1 нужно реализовывать самостоятельно!

Задачи 5-8 (на один плюсик)

5) Сложение C1C2C3 = A1A2A3 + B – трехбайтное и однобайтное.
6) Вычитание C1C2C3 = A1A2A3 - B – трехбайтное и однобайтное.
7) Сложение C1C2 = A1A2 + B1B2 – двухбайтное и двухбайтное.
8) Вычитание C1C2 = A1A2 - B1B2 – двухбайтное и двухбайтное.

Задачи 9-10 (на два плюсика):

9) Сложение C1..Cn = A1..An + B – многобайтное и однобайтное.
10) Вычитание C1..Cn = A1..An - B – многобайтное и однобайтное.
11) Умножение C1C2C3 = A1A2 * B – двухбайтное и однобайтное.

Лаб. работа № 5: «Работа с прерываниями»

Прерывание – это сигнал, поступающий от устройства или программы, сообщающий системе, что произошло определенное событие, которое необходимо обработать. Та задача, которая выполнялась в данный момент, будет отложена, и загрузится стандартная процедура для обработки данного события (из библиотеки прерывания). После этого будет продолжена работа прерванной задачи. Именно за счет системы прерываний в ЭВМ согласованно работает такое множество разнообразных устройств.

С эти примером мы уже знакомы. Для завершения любой программы надо передать управление родительской задаче, т.е. операционной системе. Это делает функция с номером **4Ch** (заносим в **AH**), параметр **00h** (заносим в **AL**) – код выхода без ошибок, **21h** – номер вызываемого прерывания (**DOS**).	mov ah, 4Ch ; Стандартное
	mov al, 00h ; завершение
	int 21h ; программы

Считывание кода нажатой клавиши с клавиатуры. Функция **01h** остановит программу и будет ждать нажатия клавиши. Если нажата обычная клавиша, то ее код будет занесен в **AL**, если нажата расширенная клавиша или комбинация (F1…12, стрелочки, ALT+x), то будет получен ноль. У таких клавиш код состоит из 2 байт – первый ноль, затем расширенный код. Запомните: если мы получили ноль, надо вызвать прерывание еще раз, чтобы считать вторую часть кода! Программу при этом нужно проходить по F8.	mov ah, 01h ; В AL заносится ASCII-код
	int 21h ; нажатой клавиши
	
	mov ah, 01h ; И еще раз повторить
	int 21h ; в случае получения нуля

Вывод на экран длинных сообщений. Прежде чем выводить на экран, сообщение надо получить в памяти. Мы создали байтовую последовательность символов, начинающуюся по адресу **MYSTR**! Обратите внимание: **$** - признак конца строки (обязателен).	MYSTR db 'Hello World!$' Переменная со строкой сообщения.

Теперь надо указать параметры вызова прерывания. За вывод строки на экран отвечает функция **09h**. MYSTR задает смещение от начала сегмента. Ее значение обязательно должно быть занесено в **DX** (используется этой функцией по умолчанию).	mov ah, 09h ; № функции
	mov dx, MYSTR ; Адрес строки
	int 21h ; Вызов прерывания

Примечание 1: Для просмотра результатов нажмите ALT-F5 (содержимое черного экрана).
Примечание 2: Если не поставить **$** в конце, после вывода строки у вас будет много мусора!
Примечание 3: MOV DX, MYSTR+1 - строка напечатается со 2-й буквы (смещение)
Примечание 4: При вводе строки с клавиатуры мы должны заранее зарезервировать достаточно места в памяти и поставить знак '**$**' после последнего введенного символа.
Примечание 5: Если нужно вывести многострочный текст, то прямо в строке ставим символы конец строки и перевод курсора на следующую: MYSTR DB '…',13,10, '…'
Примечание 6: Если нужно напечатать на экран сам символ '$', то для этого нужно использовать функцию с номером AH = 02h, выводящую на экран 1 символ, код которого находится в DL!
Примечание 7: Полный список функций библиотеки 21h: http://www.codenet.ru/progr/dos/int_0026.php

Прорешать всем (и сохранить себе для последующих задач):

1) Считать с клавиатуры код одной клавиши (1 или 2 байта). Определить и выписать в тетрадь, с каких номеров начинаются / заканчиваются последовательности цифр и английского алфавита: (0...9, a...z, A...Z, пробел, Enter, Backspace, $, двоеточие).
2) В **DL** присвоить число (0...9). Вывести на экран его 16-ричный код (цифра как символ 0...9). Программа должна правильно работать для любой цифры.
3) В **DL** присвоить число (0...15). Вывести на экран его 16-ричный код (цифра как символ 0...F). Программа должна правильно работать для любой цифры.