Практика 03: работа с памятью.

В предыдущих практиках мы использовали строковые и числовые переменные. Разберем этот вопрос по подробнее.

Резервирование памяти.

В любом языке программирования есть переменные. ASM не исключение, но в нем есть свои особенности, определенные близостью языка к аппаратному обеспечению. Для использования переменной нужно **зарезервировать** под неё память, то есть, нужно **объявить** определенный объем **данных** под именем переменной. Для объявления данных существует несколько директив

- db – 1 байт
- dw или du – 2 байта
- dd – 4 байта
- dp или df – 6 байт
- dq – 8 байт
- dt – 10 байт
- file – определяется размером включаемого файла, или явно указывается при объявлении.

Для каждого из типов - кроме file - есть аналогичный, который резервирует память, но не объявляет значение, хранящееся в этой памяти. Отличается первой буквой — вместо d используется r.

Объявляется память достаточно просто

x DB 5

Так например объявляется переменная x, размером в 1 байт и содержащая 5. Имя требуется только для удобства адресации к ячейке, но для резервирования памяти оно не требуется

db 5

так же зарезервировало ячейку памяти в 1 байт со значением 5.

Под массивом понимается последовательность переменных одинакового размера. Для объявления достаточно через запятую перечислить исходные значения ячеек

array1 db 1,1,1,1,1,1

Так будет зарезервирован массив из 6 однобайтовых переменных. При обращении fasm будет вместо названия массива подставлять адрес его первого элемента. Аналогично можно зарезервировать и через служебное слово dup

array1 db 6 dup (1)

То же самое можно объявить и так

array1 db 3 dup(1,1)

Строки представляют собой массив байт-символов и записываются при объявлении в одинарных кавычках

namestr DB 'Name'

Для корректного использования строк в командах нужно в конце строки оставлять специальный терминальный символ. Обычно это символ с кодом 0

namestr Db 'name',0

Для MS-DOS это был символ $.

Для включения в состав программы содержимого файла есть директива file. Используется для добавления в тело программы данных (текстовых или бинарных, например картинок) или включение кода из другого файла. Например

- data1 file 'data.bmp' ; полное включение файла data.bmp
- data2 file 'data.bin':20 ; добавить данные из data.bin начиная с 20 байта
- data3 file 'data.txt':20,25 ; добавить из файла data.txt данные с 20 байта по 45

Данные включаются в состав исполняемого файла на этапе компиляции и для дальнейшего использования включаемый файл не требуется.

Кроме непосредственного объявления значения ячейки, можно зарезервировать ячейку не указывая содержимое.

- ***X1 db ?***
- ***X2 dw ?,?,?***
- ***x3 dd 10 dup(?)***

или используя аналоги инструкций объявления памяти

- ***x1 rb 1***
- ***x2 rw 3***
- ***x3 rd 10***

Использование подобных неинициализированных ячеек в операторах сравнения чревато абсолютно случайным результатом. Стоит помнить, что в этих ячейках лежит мусор, оставшийся от предыдущего использования памяти, а не пустое значение.

Команды работы с памятью.

Основной командой для работы с памятью в ASM является машинная команда MOV. Формат команды прост

MOV [приемник],[источник]

Происходит копирование информации из источника в приёмник. Флаги в слове состояния процессора не изменяются.
Источником может быть

- Ячейка или область памяти
- Регистр общего назначения
- Константа
- Сегментный регистр

Приемником может служить

- Ячейка или область памяти
- Регистр общего назначения
- Сегментный регистр

Из-за особенностей архитектуры intel x86 не все пары приемник-источник являются допустимыми. Точнее нет следующих пар

Приемник	***Источник***
Ячейка памяти	Ячейка памяти
Сегментный регистр	Константа
Сегментный регистр	Сегментный регистр

В этих случаях необходимо скопировать значение сначала в регистр общего назначения и только потом в желаемый приёмник.

Так же нужно всегда помнить, что источник и приёмник должны быть одного размера. Команда типа

MOV EAX,BH

Не допустима. В этих случаях нужно выбрать операнды одного размера, например

MOV AL,BH

В случае, когда копирование происходит в переменную можно воспользоваться указанием размера источника в принудительном формате. Например, если требуется скопировать байт в переменную типа dword можно указать это явно

```
.data
X dw 0
....
.code
....
MOV BYTE[X],BH
```

Дополнительные команды работы с памятью

MOVZX
MOVSX
XCHG

Будут рассмотрены позднее.

Указатели.

Выше – и далее - мы использовали знакомый термин ***переменная***, но как таковых переменных в языке Ассемблера нет. Есть лишь указатели на область памяти в сегменте данных. Именно

так их и надо рассматривать. Для «разыменования» указателя его надо обрамить символами [и] . Например код

```
.data
X DD 12
.code
MOV EDX,X
```

Запишет в регистр EDX значение 0, то есть адрес указателя X в сегменте данных. Чтобы в EDX оказалось значение 12 нужно писать

```
MOV EDX,[X]
```

Имея это ввиду, можно оперировать указателями в памяти довольно свободно. Например

```
.data
X DD 12
Y DD 24
.code
MOV EAX,[Y]
MOV EBX,[X+4]
```

В этом примере EAX=EBX=14. Действительно, в команде MOV EBX,[X+4] мы берем значение, располагающееся по адресу указателя X плюс 4 байта, то есть значение на которое указывает Y.

Теперь становится понятна, к примеру, команда

```
Call [ExitProccess]
```

Ведь фактически мы вызываем процедуру, адрес которой записан в ячейке памяти, на которую указывает указатель

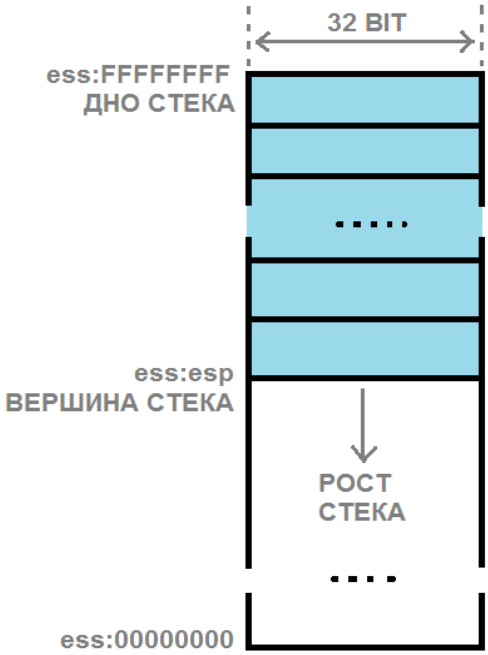
ExitProccess . Легко можно найти этот указатель в одном из включаемых через файл win32ax.inc файлов.

Стек.

Отдельно нужно говорить о структуре данных под названием **стек**. Эта структура основана по принципу LIFO – последним пришёл, первым вышел (в отечественной литературе её часто называют «магазин» - по аналогии с магазином автоматического стрелкового оружия). Стек является неотъемлемой частью архитектуры процессора и поддерживается на аппаратном уровне: в процессоре есть специальные регистры (ESS, EBP, ESP) и команды (PUSH, PUSHF, PUSHA, POP, POPF, POPA) для работы со стеком. Обычно стек используется для

- хранения адресов возврата
- локальных переменных
- сохранение значений регистров процессора

Схема организации стека в процессоре 80386 показана на рисунке



Стек располагается в оперативной памяти в сегменте стека, и поэтому адресуется относительно сегментного регистра ESS. Шириной стека называется размер элементов, которые можно помещать в него или извлекать. В нашем случае ширина стека равна 4 байтам или 32 битам. Регистр ESP (указатель стека) содержит адрес последнего добавленного элемента. Этот адрес также называется вершиной стека. Противоположный конец стека называется дном.

Дно стека находится в верхних адресах памяти. При добавлении новых элементов в стек значение регистра ESP уменьшается, то есть стек растёт в сторону младших адресов. Как вы помните, для COM-программ данные, код и стек находятся в одном и том же сегменте, поэтому если постараться, стек может разрастись и затереть часть данных и кода.

В EXE приложениях существует отельный сегмент памяти под стек, но возможность переполнения стека всё равно остаётся.

Для стека существуют две основные операции:

- PUSH - добавление элемента на вершину стека
- POP - извлечение элемента с вершины стека

Команда **PUSH** добавляет значение на вершину стека. Команда имеет один операнд, который может быть непосредственным значением, или любым регистром. При этом на величину значения уменьшается значение регистра ESP. Например, при выполнении

push eax

содержимое регистра EAX записывается в стек по адресу, указываемому регистром ESP и сразу же значение ESP уменьшается на 4 — размерность регистра EAX.

Существуют ещё 2 команды для добавления в стек. Команда **PUSHF** помещает в стек содержимое регистра флагов. Команда **PUSHA** помещает в стек содержимое всех регистров общего назначения в следующем порядке: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI (значение EDI будет на вершине стека). Значение ESP помещается то, которое было до выполнения команды. Обе эти команды не имеют операндов.

Извлечение информации из стека выполняется командой **POP**. Команда так же имеет один операнд, который может быть адресом в памяти (т. е. 32Битной переменной), или регистром, кроме регистра ECS. Например

pop ebx

извлекает значение из стека (т. е. Из адреса, на который указывает ESP) в регистр EBX и увеличивает значение ESP на 4. Заметим, что ячейки памяти в стеке не обнуляются.

Соответственно, есть ещё 2 команды. **POPF** помещает значение с вершины стека в регистр флагов. **POPA** восстанавливает из стека все регистры общего назначения (но при этом значение для ESP игнорируется).

Переменные в сегменте стека.

Часто можно встретить программы, где в качестве переменных используются конструкции вида

DWORD [esp+36]

предваряемые инструкциями

```
push    ebp
mov     ebp, esp
and     esp, -16
sub     esp, 48
```

Таким образом, пытаются использовать сегмент стека для хранения переменных и не выделяют им отдельного сегмента. Делать так осмысленно можно только в том случае, если действительно памяти настолько мало и программа так мала, что использование одного сегмента под переменные и стек является необходимым (например, в 16битной архитектуре есть только 32 кб памяти под программу и без использования стека не обойтись). В остальных случаях это просто упражнение на знание памяти компьютера и не более.

Более того, указанная часто встречающаяся, команда является не правильной. Рассмотрим следующий пример

mov dword[esp+16], 0

```
mov dword[esp+12], inputnum
mov dword[esp+8], 8
mov dword[esp+4], hellotext
mov eax,[stdoutputhandle]
mov dword[esp],eax
call [WriteConsole]
```

Фактически мы разместили в правильном порядке в стеке параметры вызова процедуры WriteConsole напрямую, минуя команду push. К чему это приводит? Фактически код работает, но только до первого вызова команды push или pop. Стоит выполнить хоть один вызов push или pop при использовании адресации с помощью регистра esp, то адресация разрушится, так как изменится значение регистра esp. Плюс нам пришлось использовать конструкцию

```
mov eax,[stdoutputhandle]
mov dword[esp],eax
```

так как варианта команды mov из ячейки памяти в ячейку памяти нет.

Если всё таки придется использовать хранение переменных в стеке, то делать это надо с использованием регистра ebp. Инициализация выглядит так

```
push epb
sub esp,32000
mov epd, esp
```

а в конце программы надо восстановить ebp

```
add esp,32000
pop ebp
```

и обращаться к переменным при помощи регистра ebp

mov [ebp+2], 12

и так далее. В этом случае команды push и pop не будут влиять на адресацию, так как изменяют только значение esp.

Пример использования переменных.

Дальнейшее интерпретирование переменных лежит на использующих их командах. Например, ячейку памяти в 1 байт можно считать и символом, и числом. Причем проводить действия с этой ячейкой можно не производя преобразования типа переменной, так как вообще такого понятия в ассемблере нет.

Например, символ таблицы ascii, соответствующий цифре 0, имеет код 48. При операциях с числами, например сравнение или инкремент это значение интерпретируется как число. А при операциях с символами — хоть при том же сравнении — значение уже будет интерпретировано как 0.

Для начала напечатаем строку, состоящую из 0.

```
.data
numberstr DB 48,0
.code
 push STD_OUTPUT_HANDLE
 call [GetStdHandle]
 push 0
 push inputnumber
 push 1
 push numberstr
 push eax
```

```
call [WriteConsole]
```

Выведется 0 на экран. Теперь добавим цикл, который выведет на экран все цифры от 0 до 9, благо в ascii таблице они располагаются последовательно.

```
  mov ecx,48
startloop:
  cmp ecx,58
  jz endloop
  push ecx
  push STD_OUTPUT_HANDLE
  call [GetStdHandle]
  push 0
  push inputnumber
  push 1
  mov byte[numberstr],cl
  push numberstr
  push eax
  call [WriteConsole]
  pop ecx
  inc ecx
  jmp startloop
endloop:
```

Для обращения к элементу строки как к байту использовали конструкцию byte, то есть принудительное указание размера хранящейся в переменной информации.

Задание 1:
вывести на экран всю ASCII таблицу.
Решение:
начальное значение цикла поставить 0, конечное 256

Задание 2:

символы таблицы при выводе заключать в кавычки и разделять пробелами.

Решение:

```
.data
numberstr DB '" _" ',0
.code
  mov cx,0
startloop:
  cmp cx,256
  jz endloop
  push ecx
  mov byte[numberstr+1],cl
  push STD_OUTPUT_HANDLE
  call [GetStdHandle]
  push 0
  push inputnumber
  push 4
  push numberstr
  push eax
  call [WriteConsole]
  pop ecx
  inc cx
  jmp startloop
endloop:
```

Структуры.

В ASM есть возможность определять свои собственные типы данных — структуры. Фактически структура это список переменных заданного формата. Структуры могут использоваться для удобного

представления данных, но основное их предназначение — это обращение к системным вызовам, где часто требуются структуры в качестве входных параметров.

Объявление структуры размещается за пределами сегментов файла, чаще всего — перед сегментом data. Формат объявление структуры следующий

```
struc ИМЯ СТРУКТУРЫ АРГУМЕНТ1,АРГУМЕНТ 2, и т. д.
{
  .АРГУМЕНТ1 ТИП АРГУМЕНТА ?
  .АРГУМЕНТ2 ТИП АРГУМЕНТА ?
}
```

Например

```
struc coord x,y
{
.x DW ?
.y DW ?
}
```

Пример объявления переменной типа структуры

```
coordinati coord 10,20
```

Для получения отдельного доступа до полей структуры достаточно указать название параметра после названия переменной. Пример

```
mov [coordinati.x],25
mov [coordinati.y],6
```

Фактически, структура представляет собой резервирование памяти размера равного размеру аргументов структуры, поэтому обращаться к ним можно и при помощи смещения. Предыдущий пример может выглядеть и так

```
mov byte[coordinati],25
mov byte[coordinati+2],6
```

хотя читаемость кода сильно снижается.

Примером использования структуры является системный вызов WriteConsoleOutputCharecter.

```
.data
simpletext DB ‘Test output.’,0
inputnum dw 0
position coord <?>
.text
mov [coord.x],20
mov [coord.y],5
push STD_OUTPUT_HANDLE
call [GetStdHandle]
push inputnum
move edx,[coord]
push edx
push 13
push simpletext
push eax
call [WriteConsoleOutputCharacter]
```

указанный код выведет строку, начиная с 20 символа 5 строки.

Задание 4:

Вывести символы ASCII таблицы с 32 по 127 строками по 32 символа, начиная с 3 строки консоли.

В качестве дополнительного примера заменим внешний вид курсора консоли. Для этого понадобится структура

```
struc CONSOLE_CURSOR_INFO  dwSize,bVisible
{
.dwSize DD ?
.bVisible DB ?
}
```

Для правильного заполнения структуры лучше сначала её прочитать, потом поправить до нужного значения и применить.

```
.data
    mycur CONSOLE_CURSOR_INFO <?,?>
.text
        push mycur
        push [stdoutputhandle]
        call [GetConsoleCursorInfo]
        mov [mycur.dwSize],100
        push mycur
        push [stdoutputhandle]
        call [SetConsoleCursorInfo]
```

Структура содержит два поля

- dwSize – размер полоски курсора в процентах от высоты символа
- bVisible – видимость курсора на экране. True значит курсор видим.

Специальные символы.

FASM поддерживает два специальных токена в выражениях, позволяющих вычислять текущую позицию assembly:

- $
- $$

$ вычисляется равным адресу в исполняемом файле начала строки, содержащей выражение. Например, таким образом можно закодировать бесконечный цикл, используя JMP $. Это разумеется не основное применение этого символа. Чаще всего он применяется вместе с инструкцией EQU.

EQU определяет символ для заданного постоянного значения: при использовании EQU исходная строка должна содержать метку. Действие EQU состоит в том, чтобы определить заданное имя метки по значению ее (только) операнда. Это определение является абсолютным и не может быть изменено позже. Так, например,

```
message     db     'hello, world'
msglen      equ    $-message
```

определяет msglen как константу 12.

$$ используется для ссылки на адрес начала текущего раздела . Например:

```
section .text
  Mov A,0x0000
  Mov B,0x0000
  Mov C,0x0000
```

для 3-й строки $$ относится к адресу 1-й строки (где начинается раздел).

Домашнее задание:

вывести ascii таблицу, используя системный вызов WriteConsole только один раз.

Решение:

создать строку нужной длины, заполнить её в цикле и вывести на экран. В качестве смещения внутри строки используем регистр ESI