Занятие 09: Метки. Генератор случайных чисел.

Меткой называется **точка в коде**, на которую **можно осуществить переход**. Фактически метка это способ языка ассемблера запоминать адреса внутри приложения для последующего обращения к ним. Всего выделяют три вида меток

1. Глобальные
2. Локальные
3. Анонимные

Глобальные метки самые часто используемые. Областью их видимости является всё приложение. Объявить метку можно тремя способами.

Первый способ это указание **имени метки с последующим двоеточием**. Это самый простой способ. Обычно так объявляются метки в коде. Пример:

Start:
…
Jmp start

Второй способ это **объявление переменной**. Ведь фактически переменная это всего лишь метка с адресом в сегменте данных.

Третий способ это использование директивы label. У нее может быть три параметра. Обязательным является только первый параметр — имя метки. Второй параметр — оператор размера (byte, word и т.д.). Он связывает с меткой размер переменной, аналогично тому, как это делают директивы объявления

данных. Далее может быть указан оператор at и адрес метки. Адрес может представлять собой константу, числовое выражение или имя другой метки. Если адрес не указан, то для создания метки используется адрес того места, где она объявлена. Например

```
Label x1            ; аналогично x1:
Label x2 byte
Label x3 word
```

В примере нет параметра at, и память фактически под метку не резервируется, поэтому все три метки указывают на один и тот же адрес памяти. Пример по сложнее

```
.data
X dw 12345
Label xh byte at x+1
```

Метка xh инициализируется адресом переменной x с добавлением единицы. Фактически она теперь указывает на старший байт слова x.
Локальная метка — это метка, имя которой начинается с точки. Во время генерации кода FASM автоматически добавляет к имени локальной метки имя последней объявленной «глобальной» метки. Таким образом, имена локальных меток могут повторяться, если между ними есть хотя бы одна «глобальная» метка.

Локальные метки удобно использовать, например, внутри процедуры. Можно дать им простые, понятные имена и не

беспокоиться, что где-то в коде уже объявлена метка с таким именем. Например:

```
…
Beginproc:
….
.loop1:
…
.exit1:
….
```

При обращении к метке .loop1 внутри процедуры достаточно указать её имя. Можно обратится к метке и из любой другой части приложения

```
Jmp  beginproc.loop1
```

Придется только указать предшествующую ей глобальную метку и – через точку – локальную метку. Отдельно стоят метки, имя которых начинается с двух точек. Они ведут себя как глобальные, но не становятся префиксом для локальных меток, то есть не разделяют области видимости локальных меток и не могут быть им префиксом.

Анонимная метка — это метка с именем @@. В программе можно объявлять сколько угодно анонимных меток, но обратиться получится только к ближайшей. Для этого существуют специальные имена:

- вместо @b (или @r) FASM подставляет адрес предыдущей анонимной метки
- вместо @f — адрес следующей анонимной метки

Этого достаточно, чтобы реализовать простой цикл, переход или проверку условия. Таким образом можно избавиться от большого количества «неанонимных» меток. Например:

```
@@:
…
Dec ecx
Cmp ecx,0
Jz @f
Jmp @b
@@:
```

Генератор случайных чисел.

При написании программ часто требуется генератор случайных чисел. В ASM нет встроенного генератора и его приходится писать самостоятельно. В простейшем случае (однократный вызов) можно воспользоваться аппаратной командой

```
Rdtsc
```

которая считывает счётчик tsc (time stamp counter) и возвращает в edx:eax значение 64-битного счётчика тактов с момента последней инициализации процессора.
Воспользоваться этой командой можно начиная с процессора Pentium, в более ранних процессорах эта инструкция отсутствует. В качестве постоянного источника случайных чисел явно не подходит.
Наиболее распространённый алгоритм генератора псевдослучайных чисел это последовательность вида

X(i+1) = A*X(i) mod M

где A=16807, а M=2147483647. Константы взяты из работ Льюиса, Гудмана, Парка и Миллера. Обычно алгоритм называют **«Генератор Парка-Миллера»**.
На языке ассемблера можно провести умножение «влоб», но в остальных языках это вызовет переполнение. Давайте рассмотрим «приближенный» алгоритм, основанный на предположении, что

M = A * Q + R

где R<Q, а 0<X(i)<M-1 . При этом A*(X(i) mod Q) и R*((X(i) / Q) лежат всегда в интервале от 0 до M-1. Для вычисления X(i+1) воспользуемся формулой

X(i+1) = A*((X(i) mod Q) – R*[X(i) / Q]

Если же вдруг X(i+1) < 0 то X(i+1) = X(i+1) + M . Обычно берут Q=12773 , а R = 2836. Опытным путем замечено, что алгоритм «зацикливается» если X(i+1) вдруг окажется равным 0. Для этого добавим в начало проверку этого события и в случае его возникновения инициируем X(i) равным EAX после вызова rdtsc.
По итогу, код процедуры генератора случайных чисел от 0 до 100000 выглядит так:

```
proc    WRandom
        push    edx ecx
        mov     eax,[randombuffer]
```

```
    or      eax,eax
    jnz     @f          ; если eax 0
    rdtsc
    xor     eax,edx   ; применим xor для большей
«случайности» числа
    mov     [randombuffer],eax
@@:
    xor     edx,edx
    mov     ecx,127773
    div     ecx     ;  edx =  остаток от X(i) / Q, то есть X(i)
mod Q
    mov     ecx,eax  ;  ecx = целое от X(i) / Q, то есть
[X(i)/Q]
    mov     eax,16807
    mul     edx  ;  eax = a * (X(i) mod Q)
    mov     edx,ecx
    mov     ecx,eax
    mov     eax,2836
    mul     edx ; eax = R*[X(i)/Q]
    sub     ecx,eax ; ecx = a * (X(i) mod Q) - R*[X(i)/Q]
   ; по идее надо проверить на отрицательность и
   ; выполнить сложение с M , но
   ; переполнение в ассемблере сделает это за нас
    xor     edx,edx
    mov     eax,ecx
    mov     [randombuffer],ecx
    mov     ecx,100000
    div     ecx ; edx = остаток от eax / 100000
    mov     eax,edx
    pop     ecx edx
    ret
```

endp

Используется заранее определенная внешняя переменная **randombuffer** размера dd.
Наиболее внимательные заметят, что в нашем факторизованном примере M существенно меньше оригинального значения, но нам его все равно достаточно. Так же понятно, что через определенное количество повторений генератор «зациклится». В нашем примере это порядок 10^8 повторений. В качестве решения предлагается алгоритм Л`Экюера, который предлагает комбинировать вывод двух последовательностей с близкими значениями M, A, Q и R. В этом случае число вызовов превысит 10^18, но для наших целей такая точность избыточна.

Задание:
Вывести на экран 2000 символов #, постоянно меняя цвет.
Решение:

```
        mov ecx,2000
loop1:  cmp ecx,1
        jz endloop1
        push ecx

        stdcall WRandom
        and eax,0000000000001111b
        push eax
        push [stdoutputhandle]
        call [SetConsoleTextAttribute]
```

```
        push 0
        push inputnum
        push 1
        push strtext
        push [stdoutputhandle]
        call [WriteConsole]
        pop ecx
        dec ecx
        jmp loop1
endloop1:
```

где stdoutputhandle хранит дескриптор окна вывода, а strtext содержит символ #. Назначение строки and eax,0000000000001111b понятно – оставляет только значения от 0 до 15.