Версия документа от 12 декабря 2019 г.

УСТЮГОВ В.А.

Введение в низкоуровневое программирование

– Практикум по NASM –

ИЗДАТЕЛЬСТВО СГУ ИМ. ПИТИРИМА СОРОКИНА

О Г Л А В Л Е Н И Е

1 Программная модель процессоров x86 3
1.1 Регистры 3
1.2 Статическая разметка памяти 3
1.3 Адресация памяти 3
1.4 Инструкции 4

2 Работа с функциями DOS 11
2.1 Простейший ввод-вывод 11
2.2 Чтение символа с клавиатуры 12
2.3 Простейший калькулятор 13
2.4 Двухразрядный калькулятор 16
2.5 Вывод текущей даты 19

3 Работа с функциями BIOS 23
3.1 Вывод символов на экран 23
3.2 Атрибуты символов 24
3.3 Вывод строк средствами BIOS 25
3.4 Работа в графическом режиме 26
3.5 Рисование квадратов 27
3.6 Работа с мышью 30
3.7 Рисования с помощью мыши 31
3.8 Работа с часами реального времени 34

Литература 37

Приложение. Схема системы команд x86 38

Введение

Казалось бы все уже давно решено, однако в Сети до сих пор регулярно появляются горячие обсуждения необходимости изучения языка ассемблера в условиях современности. Среди аргументов противников обычно фигурирует тезис о неприменимости ассемблера для решения подавляющего большинства задач, которые стоят перед современными программистами, будь то веб-программирование, работа с базами данных или научное моделирование. Это действительно так, большое количество библиотек, фреймворков и других способов поддержки программистов дают настолько высокий уровень абстрагирования от особенностей аппаратуры и архитектуры операционных систем, что большинство начинающих разработчиков могут даже не задумываться о процессах, происходящих под капотом, например, языка PHP. Однако следует заметить, что по мере развития разработчика требования к коду, к его быстродействию и безопасности начинают возрастать. Именно тогда и возникает необходимость изучения таких вопросов как строение и принципы функционирования компьютерной памяти, кэша процессора, жизненного цикла процессов в операционных системах и т.д.

Источник: https://xkcd.com/676/

Изучение языка ассемблера неразрывно связано с изучением функционирования процессора, что, в свою очередь, совершенно необходимо для понимания принципов работы операционных систем. Кроме того, за ассемблером неизменно остаются некоторые ниши мира программирования, например, реализация критических секций в программах (согласно принципу Парето 80% времени исполняется всего 20% кода, так что есть смысл именно эти 20% и оптимизировать, к примеру, переписав их на ассемблере), программирование встраиваемых систем, а также реализация модулей ядер операционных систем.

Кроме того, программирование на ассемблере можно рассматривать как особый вид головоломки, решение которой приносит не только эстетическое и интеллектуальное удовольствие, но и конкретную пользу.

Для работы мы выбираем программирование на ассемблере `nasm` под операционную систему DOS. Это может показаться анахронизмом, однако выбор обусловлен тем, что именно работая в реальном режиме можно наиболее быстро ознакомиться с принципами взаимодействия между программой и операционной системой, а также с периферийными устройствами. Описание API системы DOS, в отличие от API других систем, в силу очевидных причин имеет существенно меньший объем, переведено на русский язык и исключает необходимость использования строгих правил взаимодействия с операционной системой. В любом случае, первичное знакомство с любой операционной системой и любой системой команд процессора снизит порог вхождения в понимание других аппаратных и программных архитектур.

Продуктивного обучения!

P.S. Перевод текста на иллюстрации. 64-битный процессор безустанно работает, выполняя несколько миллиардов операций в секунду, чтобы запустить ядро XNU, которое через уровень POSIX-совместимой абстракции поднимает систему DARWIN, лежащую в основе OS X, которая, в свою очередь, напрягается, чтобы запустить Firefox и его движок Gecko, который создаёт flash-обьект, который отрисовывает несколько десятков кадров видео в секунду.

И всё из-за того, что я хочу посмотреть на то, как кошка запрыгивает в коробку и спотыкается.

Я — бог.

Глава 1

Программная модель процессоров x86

1.1. Регистры

Современные процессоры x86 имеют 8 регистров общего назначения (General Purpose Register, GPR)[1]. Названия регистров отражают их назначение в прошлом. Например, регистр EAX называется аккумулятором (Accumulator), поскольку именно в нем часто оказывался результат арифметических операций, регистр ECX назывался счетчиком (Counter) и используется для организации циклических конструкций. При этом большинство этих регистров потеряли свое изначальное предназначение, однако регистры для работы со стеком (указатель вершины стека ESP и база стека EBP) по-прежнему зарезервированы.

Регистры EAX, EBX, ECX, EDX могут быть разделены на части, которые используются как отдельные регистры. Например, младшая половина регистра EAX называется AX и может работать как независимый 16-разрядный регистр. В свою очередь младшая и старшая половины (байты) регистра AX называются AL и AH (от слов Low и High). Эти половины также могут функционировать как независимые регистры.

Аналогичная иерархия организуется и для регистров EBX, ECX, EDX.

1.2. Статическая разметка памяти

С помощью специальных директив ассемблера можно размечать статически области памяти. Близкий аналог в языках программирования — глобальные переменные. Директивы DB, DW, DD используются для размещения в памяти групп байтов, двух- или четырехбайтных слов. Размеченная область памяти может быть снабжена меткой для удобства использования (аналог — имя переменной).

```
var    db 64      ; Размещение байта с меткой var и значением 64
var2   db ?       ; Размещение неинициализированного байта с меткой var2.
       db 10      ; Размещение байта 10 без метки
X      dw ?       ; Размещение 2-байтового неинициализированного значения
Y      dd 30000   ; Размещение 4-байтного значения 30000 с меткой Y
```

1.3. Адресация памяти

Современные процессоры архитектуры x86 способны адресовать до 2^{32} байт памяти, поскольку шина адреса имеет 32 линии. В примерах выше мы использовали метки, которые на стадии трансляции программы ассемблером автоматически заменяются на 32-битными значениями, равными адресам соответствующих областей памяти. Помимо меток ассемблер предоставляет множество возможностей для вычисления адресов операндов инструкций и множество способов обратиться к ним.

[1] Глава подготовлена на основе документа http://www.cs.virginia.edu/~evans/cs216/guides/x86.html.

Ниже приведены примеры использования различных способов адресации с инструкцией пересылки данных mov.

```
mov eax, [ebx]       ; Скопировать в регистр eax значение, лежащее по адресу,
                     ; записанному в ebx
mov [var], ebx       ; Скопировать значение регистра ebx в ячейку памяти
                     ; с адресом var
mov eax, [esi-4]     ; Скопировать в регистр eax значение, лежащее по адресу,
                     ; равному значению esi, уменьшенному на 4
mov [esi+eax], cl    ; Скопировать значение cl в ячейку памяти, адрес которой
                     ; равен сумме значений регистров eax и esi
mov edx, [esi+4*ebx] ; Скопировать в регистр edx значение из ячейки памяти,
                     ; адрес которой вычисляется по формуле esi+4*ebx
```

1.4. Инструкции

Инструкциями называют команды, которые может исполнять процессор. Любая программа состоит из последовательности инструкций, каждая из которых включает в себя код операции (opcode, числовой идентификатор, который дает понять процессору, что тот должен сделать), а также адреса операндов, либо сами операнды (если это обычная числовая константа). Полный набор инструкций включает более тысячи наименований, официальная документация фирмы Intel по системе команд содержит порядка 2900 страниц, поэтому мы ограничимся лишь небольшой частью наиболее часто используемых команд.

Вообще, инструкции можно разделить на три основные группы: инструкции пересылки данных, группа команд для выполнения арифметических или логических действий, а также команды управления потоком исполнения. Для описания команд будем использовать следующие обозначения:

```
<reg32>  Любой 32-битный регистр (EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP)
<reg16>  Любой 16-битный регистр (AX, BX, CX, или DX)
<reg8>   Любой 8-битный регистр (AH, BH, CH, DH, AL, BL, CL, или DL)
<reg>    Любой регистр

<mem>    Адрес памяти (например, [eax], [var + 4], или dword ptr [eax+ebx])
<con32>  Любая 32-битная константа
<con16>  Любая 16-битная константа
<con8>   Любая 8-битная константа
<con>    Любая 8-, 16-, или 32-битная константа
```

1.4.1. Команды пересылки данных

mov — инструкция копирования значения, которое содержит или на которое ссылается второй операнд, в ячейку-приемник, адресуемую первым операндом (это может быть регистр или ячейка памяти). Команда позволяет совершать пересылки данных между регистрами, между регистрами и памятью, однако операции пересылки из памяти в память напрямую не поддерживаются. Чтобы это сделать, требуется сначала поместить значение из ячейки-источника в регистр, после чего из регистра переслать значение в приемную ячейку памяти.

Синтаксис:

```
mov <reg>, <reg>
mov <reg>, <mem>
mov <mem>, <reg>
mov <reg>, <const>
mov <mem>, <const>
```

Примеры использования:

```
mov eax, ebx          ; копируем содержимое ebx в eax
mov byte ptr [var], 5 ; записать число 5 в байт с адресом var
```

— —

push — инструкция помещения в стек, т.е. в специальную область памяти работающую по принципу, согласно которому первый попавший в стек элемент будет извлечен из стека последним. Работа со стеком имеет аппаратную поддержку в лице двух регистров. Регистр ebp указывает на начало стека, а регистр esp, который называют указателем на вершину стека, указывает на первую свободную позицию в стеке.

Синтаксис:

```
push <reg32>
push <mem>
push <con32>
```

Примеры использования:

```
push eax              ; Сохраняем в стеке значение eax
push [var]            ; Сохранить в стеке 4 байта, лежащие по адресу var
```

— —

pop — инструкция извлечения элемента из стека. При ее выполнении происходит чтение элемента памяти [sp] и помещение его в регистр или другой адрес памяти, указанный в виде операнда.

Синтаксис:

```
pop <reg32>
pop <mem>
```

Примеры использования:

```
pop edi               ; Извлечь крайний элемент из стека в регистр edi
pop [ebx]             ; Извлечь крайний элемент из стека и поместить его
                      ; по адресу, сохраненному в регистре ebx
```

— —

lea (Load Effective Address) — процессор помещает адрес, заданный вторым операндом, в регистр, указанный первым операндом. При этом никаких обращений к памяти не происходит, несмотря на похожий синтаксис, совершается только вычисление эффективного адреса.

Синтаксис:

```
lea <reg32>,<mem>
```

Примеры использования:

```
lea edi, [ebx+4*esi]  ; В регистр edi помещается значение ebx+4*esi
lea eax, [var]        ; В регистр eax помещается значение var
```

1.4.2. Арифметические и логические операции

add — инструкция для целочисленного сложения. Производит сложение двух операндов, помещая результат в первый. Адресом памяти может являться только один операнд из двух, тогда как регистрами могут быть оба операнда.

Синтаксис:

```
add <reg>,<reg>
add <reg>,<mem>
add <mem>,<reg>
add <reg>,<con>
add <mem>,<con>
```

Примеры использования:

```
add eax, 10            ; Сложить значение eax и 10, результат поместить в eax
add byte ptr [var], 10 ; Прибавить 10 к значению в памяти по адресу var
```

— —

sub — целочисленное вычитание. В первый операнд помещается результат вычитания из первого операнда значения второго операнда.

Синтаксис:

```
sub <reg>,<reg>
sub <reg>,<mem>
sub <mem>,<reg>
sub <reg>,<con>
sub <mem>,<con>
```

Примеры использования:

```
sub al, ah             ; Вычесть из значения al значение ah, результат поместить в al
sub eax, 216           ; Вычесть из значения eax число 216, результат поместить в eax
```

— —

inc, dec — инкремент и декремент, увеличение и уменьшение операнда на единицу.

Синтаксис:

```
inc <reg>
inc <mem>
dec <reg>
dec <mem>
```

Примеры использования:

```
dec eax                ; Уменьшить на 1 значение в eax
inc dword ptr [var]    ; Увеличить на 1 32-битное значение по адресу var
```

— —

imul — целочисленное умножение. В отличие от предыдущих инструкций, `imul` может иметь различное количество операндов. Если операндов два, то процессор вычисляет их произведение и помещает результат в первый операнд, которым обязательно должен быть регистр. Если операндов

три, то вычисляется произведение второго и третьего операндов (последний должен быть константой), а результат помещается в первый операнд (также регистр, как и в первом случае).

Синтаксис:

```
imul <reg32>,<reg32>
imul <reg32>,<mem>
imul <reg32>,<reg32>,<con>
imul <reg32>,<mem>,<con>
```

Примеры использования:

```
imul eax, [var]         ; Поместить в eax произведение значения eax
                        ; и значения по адресу var
imul esi, edi, 25       ; Поместить в esi произведение значения edi на 25
```

— —

idiv — целочисленное деление. В результате выполнения инструкции выполняется деление 64-битного значения, сохраненного в паре регистров edx:eax на значение операнда. Целая часть результата деления помещается в eax, остаток от деления — в edx.

Синтаксис:

```
idiv <reg32>
idiv <mem>
```

Примеры использования:

```
idiv ebx                ; Поделить содержимое пары eax:edx на значение ebx, целую часть
                        ; результата поместить в eax, остаток в edx
```

— —

and, or, xor — побитовые операции И, ИЛИ и ИСКЛЮЧАЮЩЕЕ ИЛИ. Операции совершаются над операндами, результат помещается в первый операнд.

Синтаксис:

```
and <reg>,<reg>
and <reg>,<mem>
and <mem>,<reg>
and <reg>,<con>
and <mem>,<con>

or <reg>,<reg>
or <reg>,<mem>
or <mem>,<reg>
or <reg>,<con>
or <mem>,<con>

xor <reg>,<reg>
xor <reg>,<mem>
xor <mem>,<reg>
xor <reg>,<con>
xor <mem>,<con>
```

Примеры использования:

```
and eax, 0x0f           ; Очистить биты регистра eax, исключая 4 младших бита
xor edx, edx            ; Сбросить в ноль значение регистра edx
```

— —

not — побитовая операция НЕ. Изменяет на противоположное значение все значения битов операнда.

Синтаксис:

```
not <reg>
not <mem>
```

Примеры использования:

```
not byte ptr [var]      ; Инвертировать все биты значения по адресу var
```

— —

neg — изменяет знак операнда[1].

Синтаксис:

```
neg <reg>
neg <mem>
```

Примеры использования:

```
neg eax                 ; Изменить знак значения eax
```

— —

shl, shr — логический сдвиг влево и вправо. Процессор производит сдвиг значения первого операнда на количество бит, указанных с помощью второго операнда, коим может быть константа, либо значение регистра `cl`. Освободившиеся битовые позиции при этом заполняются нулями.

Синтаксис:

```
shl <reg>,<con8>
shl <mem>,<con8>
shl <reg>,<cl>
shl <mem>,<cl>

shr <reg>,<con8>
shr <mem>,<con8>
shr <reg>,<cl>
shr <mem>,<cl>
```

Примеры использования:

```
shl eax, 1              ; Произвести сдвиг значения eax на 1 позицию влево,
                        ; что эквивалентно умножению числа на 2 (если старший
                        ; бит eax равен нулю)
```

[1] Здесь надо помнить, что отрицательные числа хранятся в дополнительном коде.

1.4.3. Инструкции управления потоком исполнения

Инструкции управления потоком (flow control instructions или по-русски команды перехода) предназначены для неявного изменения регистра процессора `eip`, который содержит адрес инструкции, которая будет выполнена следующей. Эти манипуляции позволяют организовывать условные переходы, циклические конструкции и т.п.

Для обозначения адресов инструкций в ассемблере часто используют метки, которые ассемблер при трансляции автоматически заменяет на адреса. В последующих примерах мы будем использовать обозначение `<label>` для абстрактной метки.

`jmp` — инструкция безусловного перехода. Процессор начинает исполнять инструкции, расположенные с адреса `<label`.

Синтаксис:

```
jmp <label>
```

Примеры использования:

```
jmp begin               ; Произвести переход по метке begin
```

— —

`jcondition` — условный переход по флагу. Регистр состояния процессора содержит набор битовых флагов, характеризующих результат выполнения инструкции. Если в результате инструкции был получен ноль, флаг нуля `Z, Zero` становится единицей. Также существуют флаги, сигнализирующие о переносе (`C, Carry`), о получении отрицательного значения (`S, Sign`), о переполнении, когда результат не поместился в регистр-приемник (`C, Carry`) и другие.

Синтаксис:

```
je <label>              ; (jump when equal)
                        ; Переход, если равно
jne <label>             ; (jump when not equal)
                        ; Переход, если не равно
jz <label>              ; (jump when last result was zero)
                        ; Переход, если результат нулевой
jg <label>              ; (jump when greater than)
                        ; Переход, если больше
jge <label>             ; (jump when greater than or equal to)
                        ; Переход, если больше или равно
jl <label>              ; (jump when less than)
                        ; Переход, если меньше
jle <label>             ;(jump when less than or equal to)
                        ; Переход, если меньше или равно
```

Примеры использования:

```
cmp eax, ebx            ; При выполнении инструкции cmp происходит неразрушающее
                        ; вычитание, при равенстве операндов результат ноль
jle done                ; Произвести переход по метке, если eax<=ebx
```

cmp — инструкция сравнения. Производит сравнение операндов и устанавливает соответствующие состояния арифметических флагов. Эквивалентна по действию инструкции sub, однако в результат вычитания никуда не сохраняется.

Синтаксис:

```
cmp <reg>,<reg>
cmp <reg>,<mem>
cmp <mem>,<reg>
cmp <reg>,<con>
```

Примеры использования:

```
cmp dword ptr [var], 10 ; Сравнить содержимое ячейки памяти с адресом var и 10
jeq loop                ; Произвести переход, если значение равно 10
```

— —

call, ret — инструкции вызова процедуры и возврата из нее. В ходе выполнения инструкции call происходит автоматическое сохранение текущего значения регистра eip в стеке. Это необходимо для того, чтобы продолжить исполнение текущего кода по окончанию выполнения процедуры. Соответственно, при исполнении инструкции ret адрес возврата извлекается из стека и помещается в регистр eip.

Синтаксис:

```
call <label>
ret
```

Глава 2

Работа с функциями DOS

2.1. Простейший ввод-вывод

Рассмотрим простейшую программу, выводящую на экран заглавную латинскую букву A.

Согласно общему правилу мы должны поместить в регистр ah номер функции DOS, которую хотим вызвать. В нашем случае это функция `Console I/O` (т.е. ввод-вывод, связанный с экраном в текстовом режиме), имеющая номер `0x06`. Поскольку функция должна что-либо выводить на экран, ей необходимо в качестве аргумента передать код символа для отображения. Согласно документации мы должны передать символ через регистр `dl`. После этого можно вызывать программное прерывание `int 0x21`. Поскольку системный вызов начинает работу только с этого момента, мы можем формировать значения регистров или значения в памяти в любом удобном порядке.

Обратим отдельное внимание на то, что в ходе работы функция `Console I/O` не модифицирует значения регистров ax и dx, однако всегда следует обращаться к документации, чтобы точно знать, будут ли изменены регистры с входными аргументами!

Поскольку процессор извлекает инструкции из памяти последовательно (за исключением ситуаций с переходами), необходимо в нужный момент остановить этот процесс. Для этого используется инструкция `ret`, возвращающая управление командной строке. Существует также возможность передавать статус выполнения программ порождающей их оболочке, так что последняя может контролировать, выполнилась ли программа корректно, либо в ходе работы была обнаружена некоторая ошибка.

Если инструкцию `ret` опустить, то процессор продолжит извлекать из памяти числа (вообще говоря, непредсказуемые, фактически — «мусор»), пытаться интерпретировать их как команды с аргументами, и выполнять, что, конечно, приведет к непредсказуемому результату и краху системы (поскольку средства самозащиты DOS очень скромны).

ЛИСТИНГ 2.1

```
; Программа выводит на экран заглавную латинскую A
;

org 0x100

mov ah, 0x06                ; Функция Console IO
mov dl, 0x41                ; Код буквы 'A'
int 0x21                    ; Вызов ядра DOS

ret
```

Задачи для самостоятельного решения

1. Выведите на экран свое имя.
2. Выведите на экран числа от 0 до 9.

2.2. Чтение символа с клавиатуры

Для чтения символов с клавиатуры можно использовать функцию DOS `Keyboard Input`, номер которой, согласно документации, 1. Именно это число мы должны поместить в регистр `ah`, других входных аргументов функция не требует.

После вызова операционной системы программа «зависнет» в ожидании нажатия клавиши на клавиатуре, после чего в регистре `al` окажется код символа, соответствующего нажатой клавише. Наша цель вывести на экран числа от 0 до одноразрядного числа, введенного пользователем, поэтому мы должны преобразовать код цифрового символа в то число, которое данная цифра обозначает[1]. Согласно таблице символов ASCII код цифры отличается от числа, которое цифра обозначает, на 0x30 в большую сторону. Поэтому для получения числа мы должны из значения регистра `al`, в котором оказывается результат ввода, вычесть 0x30.

Далее организуем цикл с помощью регистра-счетчика `cx`. Поместим в его младший байт `cl` число итераций, т.е. значение из `al`. После этого, поскольку вывод чисел мы начинаем с нуля, инкрементируем значение `cl`.

Для вывода на экран используем функцию `Console I/O (0x06)` и поместим в `dl` код первого символа, т.е. нуля. После этого поставим метку, обозначающую начало цикла.

Наша задача последовательно выводить на экран числа, после чего требуется сделать перевод курсора на новую строку. Также в каждом проходе значение в `dl` требуется инкрементировать, поэтому для организации перевода на новую строку в начале итерации мы сохраняем в стеке значение `dx`. После этого мы можем использовать `dl` для вывода двух специальных символов перевода на новую строку и возврата каретки, не беспокоясь о том, что текущее число для вывода будет потеряно. После этого извлекаем из стека сохраненное значение `dx`, выводим на экран символ, и инкрементируем `dl`, чтобы в нем оказался код следующего символа.

В конце тела цикла стоит инструкция `loop label`, за которой скрывается проверка на равенство нулю регистра `cx`. Если его значение больше нуля, то происходит переход по метке к началу тела цикла, а также автоматический декремент значения. В противном случае исполняется инструкция, следующая за циклом.

ЛИСТИНГ 2.2

```
; Программа запрашивает одноразрядное число с клавиатуры,
; а затем выводит на экран числа от 0 до введенного в столбик

org 0x100

mov ah, 0x01                ; Функция Keyboard Input
int 0x21                    ; Вызываем операционную систему

sub al, 0x30                ; Вычитаем из результата ввода 30h, чтобы
                            ; получить из ASCII-кода число

xor cx, cx                  ; Очищаем на всякий случай cx

mov cl, al                  ; Регистр cl используется для организации цикла,
                            ; в нашем случае число итераций лежит в al
```

[1] По-хорошему мы должны, конечно, проверить, является ли введенное число одноразрядным. Оставим это на самостоятельное решение.

```
16                                  ; после вызова DOS
17
18  inc cl                          ; Увеличиваем на 1, чтобы вывести числа,
19                                  ; включая(!) введенное
20
21  mov ah, 0x06                    ; Функция Console IO
22  mov dl, 0x30                    ; Код нуля
23
24  label:                          ; Метка начала цикла
25
26      push dx                     ; Сохраняем временно в стеке значение dx
27
28      mov dl, 0x0d                ; Выводим на экран "возврат каретки"
29      int 0x21
30      mov dl, 0x0a                ; Выводим на экран "перевод строки"
31      int 0x21
32
33      pop dx                      ; Возвращаем сохраненное значение dx
34
35      int 0x21                    ; Выводим текущий символ
36      inc dl                      ; Инкрементируем dl, переходя к следующему символу
37
38  loop label                      ; Переходим к метке, если значение cl положительное,
39                                  ; при этом происходит автодекремент регистра
40
41  ret
```

Задачи для самостоятельного решения

1. Выведите на экран числа в обратном порядке.
2. Добавьте в программу проверку того, является ли введенное пользователем число одноразрядным.
3. Выведите на экран пирамиду из символов `*`, число этажей в которой вводит пользователь.
4. Попробуйте реализовать вариант программы с двухразрядными числами.

2.3. Простейший калькулятор

Разработаем более сложную программу с более приятным пользовательским интерфейсом. Программа будет запрашивать у пользователя два числа и выводить результат их сложения. Для простоты ограничимся числами, сумма которых содержит только один разряд[1].

Для вывода сообщений пользователю применим функцию `Display String (0x09)`. В качестве аргумента ей необходимо передать полный адрес строки для вывода, включающий адрес сегмента и смещение в его пределах. Поскольку наша программа умещается в одном сегменте, мы размещаем строку в нем же, и передавать функции можно только смещение (т.е. адрес первого символа строки в текущем сегменте). Обратим внимание на то, что функция `Display String` последовательно извлекает из памяти байты и выводит их на экран. По этой причине необходим некий признак конца строки. Согласно документации DOS, предназначенные для вывода функцией `0x09` строки должны оканчиваться символом доллара `$`. Если это условие не выполнить, на экран будет выводиться содержимое памяти («мусор») до тех пор, пока по очередному адресу не будет совершенно случайно найден указанный символ.

[1] Ограничение будет частично снято в следующей программе!

Для экономии строк кода мы создаем строку `newline`[1], содержащую символы перевода строки и возврата каретки, и, разумеется, терминирующий доллар. Эта строка будет выводиться тогда, когда потребуется переход на новую строку при выводе текста.

Итак, программа начинается с вывода на экран приглашения ввести первое число и перевода строки. После этого с помощью функции `Keyboard Input (0x01)` запрашивается символ с клавиатуры. Код этого символа, согласно документации, сохраняется в регистре `al`. Предотвращая потерю текущего символа в ходе ввода следующего, сохраним значение регистра ax в стеке. Аналогичным образом, выводя вторую строку-приглашение, запросим второе число.

После этого начинаем процедуру вывода результатов вычислений на экран в формате $x + y = z$. Из стека в регистры dx и bx пересылаем первое и второе число (точнее, коды цифр, поскольку никаких коррекций мы пока не совершали), и с помощью функции `Console I/O (0x09)` выводим первое число на экран. Далее возвращаем его в стек и символ + и второе число.

После этого требуется выполнить сложение чисел. Забираем из стека число в регистр dx, выполняем сложение и сразу корректируем результат, вычитая из значения регистра с суммой `0x30` (т.к. мы фактически сложили два кода чисел, а для вывода результата нам также требуется код цифры). Наконец, знакомыми способами выводим символ равенства и искомое число.

ЛИСТИНГ 2.3

```
; Программа запрашивает у пользователя последовательно
; два числа и выводит на экран их сумму
;

org 0x100

mov ah, 0x09                ; Функция Display String
mov dx, string1             ; Адрес строки в текущем сегменте
int 0x21                    ; Вызов операционной системы

mov dx, newline             ; Переход на новую строку
int 0x21

mov ah, 0x01                ; Запрашиваем число с клавиатуры
int 0x21

push ax                     ; Сохраняем в стеке введенное значение

mov ah, 0x09                ;
mov dx, newline             ; Переход на новую строку
int 0x21                    ;

mov dx, string2             ; Адрес второй строки
int 0x21                    ; Вызов операционной системы

mov ah, 0x09                ;
mov dx, newline             ; Переход на новую строку
int 0x21                    ;

mov ah, 0x01                ; Запрашиваем число с клавиатуры
int 0x21
```

[1] Фактически, конечно, мы не создаем строку, а выделяем в памяти несколько байт, записываем в их нужные значения, а слово `newline` есть просто псевдоним для адреса первого байта последовательности. Это же верно для всех остальных «строк» в программе.

```
32
33  push ax                     ; Сохраняем в стеке введенное значение
34
35  mov ah, 0x09                ;
36  mov dx, newline             ; Переход на новую строку
37  int 0x21                    ;
38
39  mov dx, string3             ; Адрес третьей строки
40  int 0x21                    ; Вызов операционной системы
41
42  mov ah, 0x09                ;
43  mov dx, newline             ; Переход на новую строку
44  int 0x21                    ;
45
46  pop bx                      ; Забираем из стека второе число
47  pop dx                      ; Забираем из стека первое число
48
49  mov ah, 0x06                ; Вывод первого числа на экран
50  int 0x21
51
52  push dx                     ; Запоминаем в стеке первое число
53
54  mov dl, 0x2b                ; Символ '+'
55  int 0x21
56
57  mov dx, bx                  ; Выводим второе число на экран
58  int 0x21
59
60  pop dx                      ; Вспоминаем первое число
61  add bx, dx                  ; Выполняем сложение
62  sub bx, 0x30                ; Корректируем результат сложения
63                              ; для вывода на экран
64
65  mov dl, 0x3d                ; Символ '='
66  int 0x21
67
68  mov dl, bl                  ; Выводим результат
69  int 0x21
70
71
72  ret
73
74  ; В строке числа 0x0a и 0x0d - символы перевода строки и возврата каретки
75  ; Для работы с функцией 0x09 Display String строка должна заканчиваться
76  ; символом $
77  string1:
78      db 'Input first number:$'
79
80  string2:
81      db 'Input second number:$'
82
83  string3:
84      db 'The result is:$'
```

```
85
86 newline:
87     db 0x0a, 0x0d, '$'
```

Задачи для самостоятельного решения

1. Дополните программу проверками корректности вводимых чисел. Для определенности будем считать, что числа не должны превышать 4.

2.4. Двухразрядный калькулятор

Дополним предыдущую программу возможностью получения и вывода двухразрядного результата. Складываются по-прежнему одноразрядные числа.

Ввод значений происходит способом, аналогичным тому, который использовался в предыдущей программе простого калькулятора. Небольшая разница заключается в том, что планируя использовать операцию коррекции сложения, мы должны обнулить значение в ah перед сохранением результата пользовательского ввода в стеке (фактически, это «лишние» биты введенного числа, сохраненного в ax). Для этого мы могли бы использовать простую команду пересылки данных `mov ah, 0x00`, но вместо нее применим более рациональный способ, который дает в выигрыш в размере исполняемого файла [1] и в скорости выполнения. Как известно, операция побитового логического ИЛИ числа с самим собой дает в результате нуль. Это свойство мы и используем: `xor ah, ah`.

Итак, для десятичной коррекции результатов после сложения двух неупакованных двоично-десятичных чисел (фактически речь идет о сложении числе в виде ASCII кодов) мы используем команду `aaa` (Adjust After Adding):

```
pop ax
pop bx
add ax, bx
aaa
```

В результате в `al` будет записан младший разряд (не код!) результата сложения, в ah — старший. При этом, если старший разряд не равен нулю, то отследить это можно с помощью флага переноса Carry, что мы и делаем. Переход по метке `twodigits` происходит, если флаг переноса поднят, т. е. в результате сложения получилось двухразрядное число.

Процедура вывода результата на экран должна быть абсолютна понятна при условии изучения предыдущего материала. Для вывода используется функция `Console I/O (0x06)`, в случае вывода двух разрядов предварительно требуется сохранить значение ax в стеке, поскольку номер функции передается операционной системе через ah, и его значение, следовательно, мы потеряем, если не предпримем специальных мер.

ЛИСТИНГ 2.4

```
1 ; Программа запрашивает у пользователя последовательно
2 ; два числа и выводит на экран их сумму.
3 ; Вариант, работающий с двухразрядным результатом
4
5 org 0x100
6
7 mov ah, 0x09                  ; Функция Display String
8 mov dx, string1               ; Адрес строки в текущем сегменте
9 int 0x21                      ; Вызов операционной системы
```

[1] Выигрыш составляет целый байт! В этом случае не требуется хранить нулевую константу в составе ассемблерной инструкции.

```
10
11 mov dx, newline              ; Переход на новую строку
12 int 0x21
13
14 mov ah, 0x01                 ; Запрашиваем число с клавиатуры
15 int 0x21
16
17 xor ah, ah                   ; Очищаем ah, т.к. в нем лишняя 1
18 push ax                      ; Сохраняем в стеке введенное значение
19
20 mov ah, 0x09                 ;
21 mov dx, newline              ; Переход на новую строку
22 int 0x21                     ;
23
24 mov dx, string2              ; Адрес второй строки
25 int 0x21                     ; Вызов операционной системы
26
27 mov ah, 0x09                 ;
28 mov dx, newline              ; Переход на новую строку
29 int 0x21                     ;
30
31 mov ah, 0x01                 ; Запрашиваем число с клавиатуры
32 int 0x21
33
34 xor ah, ah                   ; Очищаем ah, т.к. в нем лишняя 1
35 push ax                      ; Сохраняем в стеке введенное значение
36
37 mov ah, 0x09                 ;
38 mov dx, newline              ; Переход на новую строку
39 int 0x21                     ;
40
41 mov dx, string3              ; Адрес третьей строки
42 int 0x21                     ; Вызов операционной системы
43
44 mov ah, 0x09                 ;
45 mov dx, newline              ; Переход на новую строку
46 int 0x21                     ;
47
48 pop bx                       ; Забираем из стека второе число
49 pop dx                       ; Забираем из стека первое число
50
51 mov ah, 0x06                 ; Вывод первого числа на экран
52 int 0x21
53
54 push dx                      ; Запоминаем в стеке первое число
55
56 mov dl, 0x2b                 ; Символ +
57 int 0x21
58
59 mov dx, bx                   ; Выводим второе число на экран
60 int 0x21
61
62 push bx                      ; Возвращаем в стек второе число
```

```
63
64 mov dl, 0x3d                ; Символ =
65 int 0x21
66
67 pop ax                      ; Вспоминаем второе число
68 pop bx                      ; Вспоминаем первое число
69 add ax, bx                  ; Выполняем сложение в ASCII-кодах
70 aaa                         ; Коррекция после сложения
71
72 jc twodigits                ; Переход, если получилось два разряда
73
74 mov ah, 0x06                ; Функция Console IO
75 mov dl, al                  ; Результат коррекции лежит в al
76 add dl, 0x30                ; Для вывода символа
77 int 0x21
78
79 ret                         ; Конец. Дальнейший код для двух разрядов
80
81 twodigits:
82
83     push ax                 ; Запоминаем результат коррекции
84     mov dl, ah              ; Старшиий байт результата коррекции
85     add dl, 0x30            ; Дл вывода на экран
86     mov ah, 0x06            ; Функция Console IO
87     int 0x21
88
89     pop dx                  ; Вспоминаем результат коррекции
90     add dl, 0x30            ; Младший байт оказался сразу в dl
91     int 0x21
92
93 ret
94
95 ; В строке числа 0x0a и 0x0d - символы перевода строки и возврата каретки
96 ; Для работы с функцией 0x09 Display String строка должна заканчиваться
97 ; символом $
98 string1:
99     db 'Input first number:$'
100
101 string2:
102     db 'Input second number:$'
103
104 string3:
105     db 'The result is:$'
106
107 newline:
108     db 0x0a, 0x0d, '$'
```

Задачи для самостоятельного решения

1. Дополните программу проверками корректности вводимых чисел.

2.5. Вывод текущей даты

Для запроса у операционной системы текущей даты используется функция `Get Date (0x2a)`. Для работы ей не требуется никаких аргументов, по выполнении мы получаем дату в следующем формате (согласно документации): номер дня месяца в регистре `dl`, номер месяца в `dh`, текущий год — в `cx`. Основная сложность, которая здесь возникает, заключена в том, что полученные числа требуется перевести из шестнадцатеричной системы счисления в десятичную.

Решение этой задачи мы разделим на две части. Для перевода года мы используем полноценную процедуру для представления шестнадцатеричного числа в виде последовательности разрядов соответствующего десятичного, сохраненных в специально выделенном буфере. А для преобразования одно- или двухразрядных номеров месяца и дня месяца можно использовать несколько не очевидный, но быстрый способ.

Итак, после получения даты значение регистра `dx`, в котором оказались номер месяца и номер дня, сохраняется в стеке, поскольку в ходе дальнейших действий мы можем потерять число, до вывода которого мы еще не дошли[1]. После этого мы очищаем старший байт регистра `ax` и выполняем команду `aam`, предназначенную для коррекции результата после умножения[2] неупакованных двоично-десятичных чисел. Фактически, команда производит деление значения `al` на `0x0a`, т. е. на 10, при этом частное помещается в `ah`, а остаток от деления — в `al`. Нетрудно заметить, что эти действия эквивалентны переводу числа в десятичную систему счисления.

После проведения коррекции вызывается функция `PrintTwoDigits`. Она предполагает, что два одноразрядных числа, подлежащих выводу, хранятся в старшем и младшем байтах регистра `ax`. Если взглянуть на код функции, можно увидеть, при условии внимательного изучения предшествующего материала и выполнения заданий для самостоятельной работы, что производятся стандартные и привычные действия для работы с системным вызовом `Console I/O (0x06)`. В первых строках функции значения затрагиваемых регистров сохраняются в стеке, в последних строках они восстанавливаются, причем в обратном порядке, согласно работе стека как структуры хранения данных.

Возвратимся к основной ветке кода. После вывода номера дня в качестве элемента оформления выводится знак минус, после чего из стека выталкивается сохраненный номер месяца и повторяется процедура перевода его в десятичный вид и подготовки к вызову `PrintTwoDigits`.

Наконец, приступим к выводу номера текущего года. Для перевода четырехразрядного числа в в десятичную систему используем функцию `ConvertNumber`. На входе она требует поместить в `ax` шестнадцатеричное число для преобразования, в `bx` — основание системы счисления, в которую требуется осуществить перевод. Для хранения разрядов искомого числа в виде ASCII-кодов функция использует специальный буфер, который в программе называется `buffer` и указатель на текущую позицию в буфере, который хранится в регистре общего назначения `si`.

Основной цикл преобразования ограничен метками `.convert` и `.end`. Обратите внимание на точку перед именами меток. Такой синтаксис позволяет объявлять блоки кода, доступные локально из более крупного блока, определенного именем без точки, в нашем случае это локальные метки внутри процедуры `ConvertNumber`.

Внутри цикла мы очищаем `dx`, после чего производим целочисленное деление `ax` на основание целевой системы счисления, записанное в `bx`. При этом частное оказывается в `ax`, остаток от деления — в `dl`. Собственно, этот остаток есть младший разряд искомого результата. Выполняем коррекцию, добавляя к результату код нуля, и выполняем проверку полученного кода с помощью инструкции сравнения. Если получилась десятичная цифра (т.е. код меньше кода девятки), то с помощью инструкции `jbe .store` (Jump if Below or Equal) выполняем условный переход в блок сохранения результатов в буфере.

В противном случае, если полученная цифра оказалась шестнадцатеричной (наша процедура универсальная и работает с любым основанием системы счисления от 2 до 16), то мы должны преобразовать полученный код к коду этой цифры. Для этого мы вычитаем код нуля, приводя значение `dl` к исходному, вычитаем 10, чтобы получить «номер» шестнадцатеричной цифры, и прибавляем код

[1] Как говорят программисты на Perl (правда, немного в другом контексте, но тем не менее), «There's More Than One Way To Do It». В некоторых случаях можно не пользоваться стеком, а при потере данных просто сделать запрос снова. Но временное хранение в стеке более изящно, дешевле в отношении времени получения потерянных данных и, в конце концов, вполне естественно.

[2] Может возникнуть вопрос: «А где, собственно, было сделано умножение?» — То, что некоторое число не изменялось, эквивалентно умножению его на единицу.

латинской заглавной буквы A. Чтобы полностью это осознать, представьте, что после деления в `dl` оказалось число, к примеру, `0x0C`, и проделайте на бумаге последовательно все имеющиеся в коде программы преобразования и проверки.

Сохранение производится просто: мы смещаемся по буферу на одну позицию к его началу, декрементируя `si`, сохраняем разряд инструкцией пересылки данных. После этого проверяем равенство нулю частного от деления, которое, как мы помним, осталось в ax. Проверка производится контролем флага нуля после операции `and ax, ax`. Очевидно, что результатом будет само значение ax, а флаг нуля будет поднят только если в ax хранится ноль. Это будет означать, что цикл преобразования должен быть окончен, из стека должны быть вытолкнуты сохраненные значения регистров, а подпрограмма должна быть завершена. В противном случае происходит переход к началу цикла преобразования.

После возврата к основной ветке кода мы располагаем указателем на старший разряд числа `si`. Поскольку мы заранее знаем количество разрядов для вывода, мы организуем цикл на 4 итерации, записав это число в cx[1]. В цикле `output` происходит последовательный вывод символов из буфера на экран с помощью обычного вызова `Console I/O` и инкремент указателя.

ЛИСТИНГ 2.5

```
; Программа запрашивает у ОС текущую дату
; и выводит ее на экран
;

org 0x100

mov ah, 0x2a                ; Функция Get Date
int 0x21                    ; Вызов операционной системы

push dx                     ; Запоминаем в стеке текущий месяц и день

mov ax, dx                  ; Переносим dx в аккумулятор для коррекции
xor ah, ah                  ; Очищаем ah, т.к. нам нужен только al
aam                         ; Коррекция

call PrintTwoDigits         ; Выводим номер дня в двухразрядном формате

mov ah, 0x06                ; Функция Console IO
mov dl, '-'
int 0x21

pop ax                      ; Забираем из стека текущий месяц (окажется в ah)
mov al, ah
xor ah, ah                  ; Переносим ah в al и очищаем первый
aam                         ; Коррекция

call PrintTwoDigits         ; Выводим номер месяца в двухразрядном формате

mov ah, 0x06                ; Функция Console IO
mov dl, '-'
int 0x21

mov ax, cx                  ; Число для перевода, см. интерфейс ConvertNumber
```

[1] Потомки из 11 тысячелетия должны записать в cx число 5.

```
34 mov bx, 0x0a                ; Основание системы счисления
35 call ConvertNumber          ; Преобразуем число в десятичное, ASCII-вид
36
37 mov cx, 0x04                ; Количество разрядов в номере года знаем заранее
38 mov ah, 0x06                ; Функция Console IO
39
40 output:
41
42     mov dl, [si]            ; si ссылается на старший разряд в буфере
43     int 0x21                ; Вызов ОС
44     inc si                  ; Двигаемся по буферу
45
46 loop output                 ; Зацикливаемся, cx уменьшается автоматически
47
48 ret
49
50 ;   PrintTwoDigits
51 ;   Вход:
52 ;   AH хранит старший разряд
53 ;   AL хранит младший разряд
54
55 ;   Выход:
56 ;
57
58 PrintTwoDigits:
59
60     push ax                 ; Сохраняем затрагиваемые регистры
61     push dx
62
63     push ax                 ; Требуется для алгоритма
64
65     mov dl, ah              ; Старший разряд
66     add dl, 0x30            ; Коррекция для вывода
67
68     mov ah, 0x06            ; Функция Console IO
69     int 0x21
70
71     pop ax                  ; Вспоминаем значение ax
72     mov dl, al              ; Теперь младший разряд
73     add dl, 0x30
74
75     mov ah, 0x06
76     int 0x21
77
78     pop dx                  ; Восстанавливаем значение регистров
79     pop ax
80
81     ret                     ; Возврат в основную программу
82
83
84
85
86
```

```
87  ; ConvertNumber
88  ;   Вход:
89  ;     ax хранит число для преобразования
90  ;     bx хранит основание системы счисления
91  ;          для преобразования
92  ;
93  ;   Выход:
94  ;     si содержт начало буфера с разрядами
95  ;          преобразованного числа в ASCII-кодах
96
97  ConvertNumber:
98      push ax                 ; Сохраняем затрагиваемые регистры
99      push bx
100     push dx
101     mov si, bufferend       ; Встаем на конец буфера
102
103 .convert:
104     xor dx, dx              ; Очищаем dx
105     div bx                  ; Делим ax на основание системы счисления,
106                             ;   результат деления в ax, остаток - dl
107     add dl, '0'             ; Преобразуем в ASCII-вид
108     cmp dl, '9'             ; Проверяем, получилась ли десятичная цифра
109     jbe .store              ; Да, переходим к сохранению
110     add dl, 'A'-'0'-10      ; Нет, преобразуем к нужному виду
111
112 .store:
113     dec si                  ; Смещаемся по буферу назад
114     mov [si], dl            ; Сохраняем полученное значение
115     and ax, ax              ; Проверяем равенство нулю основного результата
116     jnz .convert            ; Если не ноль, значит есть еще разряды для
117                             ;   преобразования
118 .end:
119     pop dx                  ; Восстанавливаем значение регистров
120     pop bx
121     pop ax
122     ret
123
124 buffer: times 16 db 0       ; 16 нулевых позиций в памяти
125 bufferend: db 0             ; Конец буфера
```

Глава 3

Работа с функциями BIOS

3.1. Вывод символов на экран

API системы BIOS предоставляет для программиста ряд низкоуровневых функций, дающих возможность работать с устройствами ввода-вывода, не используя вызовы операционной системы. При этом, например, функции для вывода символов и строк на экран дают больше свободы в настройке параметров (координаты на экране, цвет), чем вызовы DOS. Доступ к функциям BIOS можно получить, в частности, через обработчик прерывания с номером `0x10`. При этом номер функции, как и при системных вызовах DOS, требуется размещать в регистре ax.

Рассмотрим простую программу для вывода символа в центре экрана. Функция с номером `0x00` позволяет задать рабочий видеорежим. От видеорежима зависит, как будет отображаться информация на экране — в виде символов, либо отдельных точек, а также их количество вдоль горизонтальной и вертикальной сторон экрана, и цвет. Список поддерживаемых видеорежимов можно найти в документации, а непосредственно в программе номер выбранного режима передается функции `0x00` через регистр `al`.

При работе с вводом-выводом BIOS требуется вручную управлять положением курсора на экране. Для этого используется функция `0x02`. В качестве входных данных она принимает координаты курсора в паре регистров `dh` (номер строки) и `dl` (номер столбца). Обратите внимание, что для лучшей читаемости в программе эти числа переданы в регистры в десятичном виде (ассемблер nasm поддерживает разные формы записи чисел). Предельные значения номеров строк и столбцов определяются текущим видеорежимом (для выбранного в программе — 25 строк и 40 столбцов). Разумеется выполнение кода функции происходит после вызова программного прерывания `int 0x10`.

За непосредственный вывод символов ответственна функция `0x09`. Код символа передается ей через регистр `al`. Далее мы должны через регистр cx указать число повторений символа (каретка при этом сдвигается автоматически). Наконец, регистр `bl` должен содержать атрибут символа, описывающий его цветовые настройки: биты 2-0 отвечают за цвет самого символа, бит 3 определяет яркость цвета символа, биты 6-4 задают цвет фона, бит 7 позволяет включить режим мигания символа. В нашем случае цвет символа задается кодом `100` (красный), а также поднимается упомянутый выше 7-й бит.

ЛИСТИНГ 3.1

```
; Программа выводит на экран заглавную латинскую A
; красного цвета в центре экрана средствами BIOS

org 0x100

mov ah, 0x00                ; Функция BIOS выбора видеорежима
mov al, 0x03                ; Текстовый видеорежим 80x25
int 0x10                    ; Вызов BIOS

mov ah, 0x02                ; Функция BIOS управления положением курсора
```

```
11 mov dh, 12                  ; Номер строки, где будет выведен символ
12 mov dl, 40                  ; Номер столбца
13                             ; Внимание! Значения DH и DL задаем для простоты
14                             ; в десятичной системе счисления.
15 int 0x10                    ; Вызов BIOS
16
17 mov ah, 0x09                ; Вывести символ с заданным атрибутом
18 mov al, 'A'                 ; Символ
19 mov bl, 0b10000100          ; Атрибут символа
20 mov cx, 0x0001              ; Число повторений символа
21 int 0x10                    ; Вызов BIOS
22
23 ret
```

Задачи для самостоятельного решения

1. Выведите на экран свое имя.
2. Выведите на экран числа от 0 до 9.

3.2. Атрибуты символов

Поскольку для хранения атрибута символа используется один восьмибитный регистр, можно задать всего 255 различных атрибутов. Напишем программу, в которой некоторый символ выводится 255 раз, причем каждый раз с новым атрибутом.

Как и в предыдущем примере, необходимо перейти в соответствующий текстовый видеорежим, для чего используем функцию BIOS `0x00`.

Изменять номера строк и столбцов выводимых символов мы будем изменять в циклической конструкции, причем счетчиками будут регистры, хранящие атрибут и номер столбца. Задаем начальные условия: обнуляем регистры, хранящие координаты курсора и атрибут, помещаем в регистр `al` код символа, который будем выводить, а также задаем режим вывода символа в единственном экземпляре.

Внешний цикл, изменяющий атрибут, начинается с метки `list_attributes`. В первых строках этого цикла вызывается функция `0x02`, помещающая курсор в необходимую позицию, после чего на экран выводится символ с текущим атрибутом. Далее инкрементируем значение регистра, хранящего номер столбца. После этого необходимо выполнить проверку на достижение конца строки. Если это не произошло, делаем переход по метке `.next_column`. В коде, следующем после метки, увеличивается на 1 атрибут, и производится проверка на переполнение регистра с атрибутом (`test bl, 0xff` вернет ноль, и при этом в регистре состояния процессора поднимется флаг нуля, если `bl` хранит нулевое значение.) Если переполнения не было, возвращаемся к началу цикла.

В случае, если при проверке на достижение конца строки конец строки был обнаружен, выполняются две дополнительные строчки инкремента номера строки и обнуления номера текущего столбца.

ЛИСТИНГ 3.2

```
1  ; Программа выводит на экран заглавную латинскую A
2  ; с перебором всех возможных атрибутов средствами BIOS
3
4  org 0x100
5
6  mov ah, 0x00                ; Функция BIOS выбора видеорежима
7  mov al, 0x03                ; Текстовый видеорежим 80x25
8  int 0x10                    ; Вызов BIOS
9
```

```
10 mov dh, 0                    ; Начальная строка (десятичная система!)
11 mov dl, 0                    ; Начальный столбец (десятичная система!)
12
13 mov al, 'A'                  ; Символ
14 xor bl, bl                   ; Начальный атрибут символа нулевой
15 mov cx, 0x0001               ; Число повторений символа
16
17 list_attributes:
18
19     mov ah, 0x02             ; Функция BIOS управления положением курсора
20     int 0x10                 ; Вызов BIOS
21
22     mov ah, 0x09             ; Вывести символ с заданным атрибутом
23     int 0x10                 ; Вызов BIOS
24
25     inc dl
26     cmp dl, 80               ; Добрались до конца строки?
27
28     jnz .next_column         ; Нет
29
30     inc dh                   ; Идем в начало следующей строки
31     mov dl, 0
32
33     .next_column:
34
35     inc bl                       ; Увеличиваем атрибут на 1
36     test bl, 0xff                ; Проверяем BL на равенство нулю
37     jnz list_attributes          ; Зацикливаемся
38
39 ret
```

Задачи для самостоятельного решения

1. Преобразуйте программу, чтобы на экран выводились символы с различными атрибутами по строкам, причем в каждой строке должен быть один определенный цвет символа. Строк должно получиться 16 по числу возможных цветов.

3.3. Вывод строк средствами BIOS

Для вывода строк BIOS предлагает набор готовых инструментов.

Сначала, разумеется, необходимо перейти в нужный текстовый видеорежим. Строка с заданным атрибутом выводится с помощью функции `0x13`. В регистр `al` помещаем 1, что позволяет после вывода строки поместить курсор в следующую позицию на экране. Указать адрес строки (фактически — первого байта) требуется через регистр `bp`. Также функции требуется передать длину строки через `cx`. Координаты, как обычно для функций BIOS, передаются через пару регистров `dh` и `dl`. Аналогично процедуре вывода символа необходимо указать атрибут.

ЛИСТИНГ 3.3

```
; Программа выводит на экран строку символов
; красного цвета в центре экрана средствами BIOS

org 0x100

mov ah, 0x00                ; Функция BIOS выбора видеорежима
mov al, 0x03                ; Текстовый видеорежим 80x25
int 0x10                    ; Вызов BIOS

mov ah, 0x13                ; Вывести строку с заданным атрибутом
mov al, 0x01                ; Режим вывода: поместить курсор после строки
mov cx, 15                  ; Число символов в строке
mov bl, 0b00000100          ; Атрибут
mov dh, 12                  ; Номер строки, где будет выведена строка
mov dl, 32                  ; Номер столбца
mov bp, string              ; Адрес строки
int 0x10

ret

string:
    db 'Pitirim Sorokin'
```

Задачи для самостоятельного решения

1. Запросите у пользователя число от 1 до 10. Выведите на экран строки вида «X негритят отправились обедать», в каждой следующей строке уменьшая счетчик. Последней выведите строку: «И никого не стало».

3.4. Работа в графическом режиме

Рассмотрим принцип работы в графическом режиме средствами BIOS. Наша первая программа выведет в центре экрана красную точку.

С помощью знакомой функции с номером `0x00` мы должны попасть в графический видеорежим, в котором на экране размечается 320 на 200 точек, каждая точка имеет один из 256 цветов.

Для того, чтобы отрисовать точку, необходимо использовать функцию `0x0c`. Ее номер, как всегда, передается BIOS через регистр `ah`. Цвет точки указывается через регистр `al`. Очевидно, что также требуется передать функции координаты. Поскольку значение координаты может не поместиться в восьмибитную половину регистров, для хранения координаты используются 16-рзрядные регистры `cx` (горизонтальная координата) и `dx` (вертикальная).

После вызова `int 0x10` BIOS отрисует точку. Перед завершением программы необходимо перевести экран в текстовый режим. Однако, если сделать это сразу после отображения точки, экран будет очищен, и точку мы не увидим. Чтобы предотвратить такую ситуацию, организуем ожидание нажатия клавиши после отрисовки точки. В нашей программе это выполнено с помощью функции BIOS `0x01` для чтения с клавиатуры. Если пользователь не нажимал на клавиши, функция `0x01` вернет нулевой код, и будет поднят флаг нуля в регистре состояния процессора. Если это так, то мы делаем условный переход в начало цикла и снова опрашиваем клавиатуру.

ЛИСТИНГ 3.4

```
; Программа переводит консоль в графический режим
; и отрисовывает красную точку в центре

org 0x100

mov ah, 0x00                ; Функция BIOS выбора графического режима
mov al, 0x13                ; Графический режим 320x200, 256 цветов
int 0x10                    ; Вызов BIOS

mov ah, 0x0c                ; Функция BIOS отображения точки
mov al, 0x04                ; Задание цвета
mov cx, 160                 ; Горизонтальная координата (десятичная система!)
mov dx, 100                 ; Вертикальная координата (десятичная система!)
int 0x10                    ; Вызов BIOS

nokey:
    mov ah, 0x01            ; Функция чтения клавиатуры
    int 0x16                ; Вызов BIOS
    jz nokey                ; Переходим, если ничего не прочитано, т.е.
                            ; находимся здесь, пока не будет нажата клавиша

mov ah, 0x00                ; Функция BIOS выбора графического режима
mov al, 0x03                ; Текстовый режим 80x25, 16 цветов
int 0x10                    ; Вызов BIOS

ret
```

Задачи для самостоятельного решения

1. Вы научились рисовать точку. Проявите фантазию и придумайте себе задание, изобразите с помощью точек какой-нибудь несложный образ, возможно, абстрактный.
2. Изобразите две линии, крест-накрест перечеркивающие экран.
3. Изобразите квадрат в центре экрана.
4. Изобразите на экране российский флаг.
5. Запросите у пользователя ширину линии в пикселях, и затем изобразите линию с заданной шириной.

3.5. Рисование квадратов

Напишем программу, в результате работы которой на экране будут отрисованы 15 квадратов доступных стандартных цветов[1]. Для удобства процедуру рисования квадрата оформим в виде функции.

Основная часть программы начинается обычно: переходим в графический режим, задаем начальные условия для рисования квадратов. В паре регистров `cx` и `dx` будем хранить координаты левого верхнего угла квадрата, регистр `si` будет содержать размер квадратов в пикселях (здесь нужно проследить, чтобы квадраты поместились на экран в текущем видеорежиме), а в `al` — цвет.

С помощью метки `plotting` организуем цикл, в котором будем перебирать цвета, вызывать функцию отрисовки квадрата `PlotSquare` и увеличивать горизонтальную координату на 20 пикселей, из которых 15 это длина стороны квадрата, а 5 — небольшой зазор между квадратами.

[1] Конечно, всего цветов доступно 16, но один из них черный.

Возврат в начало цикла происходит при проверке условия равенства нулю значения в `al`, т.е. мы перебираем все цвета, а на черном цвете, код которого 0, цикл будет остановлен.

По окончанию выполнения цикла мы, как и в предыдущей программе, должны задержать завершение программы до нажатия клавиши, после чего вернуть экран в текстовый режим.

Теперь рассмотрим работу функции `PlotSquare`. Поскольку нам понадобятся регистры для вычислений, сохраняем в стеке текущие значения всех регистров, задействованных в работе.

После этого в «служебных» целях сохраним в `di` значение ширины стороны квадрата. Далее в цикле, ограниченном меткой `.plot` и командой условного перехода на эту метку, с помощью функции рисования точки на экран выводятся сразу все 4 стороны квадрата. Для понимания того, как происходит рисование, рекомендуем взять листок бумаги и по шагам поотмечать точки, которые будут отображаться[1]. Помните, что горизонтальную координату функция рисования точки берет из `cx`, вертикальную — из `dx`. Понимание облегчит и знание того, что при отрисовке каждой из сторон позиция рисующего «карандаша» хранится в `si`.

ЛИСТИНГ 3.5

```
; Программа отрисовывает в середине экрана
; квадраты, перебирая при этом 15 стандартных цветов

org 0x100

mov ah, 0x00                ; Функция BIOS выбора графического режима
mov al, 0x13                ; Графический режим 320x200, 256 цветов
int 0x10                    ; Вызов BIOS

mov cx, 10                  ; Координата левого верхнего угла первого квадрата
mov dx, 80                  ; Координата, определяющая позицию строки с квадратами
mov si, 15                  ; Размер квадратов
mov al, 15                  ; Цвет первого квадрата

plotting:

    call PlotSquare          ; Отрисовываем квадрат

    add cx, 20               ; Передвигаемся по горизонтали
    dec al                   ; Изменяем цвет, а также декрементируем счетчик

    jnz plotting             ; Переход, если не достигли нуля

nokey:
    mov ah, 0x01             ; Функция чтения клавиатуры
    int 0x16                 ; Вызов BIOS
    jz nokey                 ; Переходим, если ничего не прочитано, т.е.
                             ; находимся здесь, пока не будет нажата клавиша

mov ah, 0x00                ; Функция BIOS выбора графического режима
mov al, 0x03                ; Текстовый режим 80x25, 16 цветов
int 0x10                    ; Вызов BIOS

ret


```

[1] Описать это нормальным текстом, так, чтобы это не выглядело глупо, на мой взгляд, невозможно.

```
37 PlotSquare:
38
39 ; Вход:
40 ;   cx хранит горизонтальную координату левого верхнего угла
41 ;   dx хранит вертикальную координату левого верхнего угла
42 ;   si хранит размер стороны квадрата
43 ;   al хранит цвет
44
45     push si                  ; Запоминаем значения регистров
46     push cx
47     push dx
48     push ax
49     push di
50
51     mov di, si               ; В di будет храниться ширина
52     mov ah, 0x0c             ; Функция BIOS отображения точки
53
54 .plot:
55 ; Здесь в одном цикле отрисовываются все точки квадрата.
56 ; Начальное значение счетчика si равно длине стороны,
57 ; координаты точек записываются в пару регистров cx, dx.
58 ; Для понимания порядка отрисовки точек сделайте рисунок на бумаге.
59
60     add cx, si
61     int 0x10                 ; Вызов BIOS
62
63     add dx, di
64     int 0x10                 ; Вызов BIOS
65
66     sub cx, si
67     sub dx, di
68     add dx, si
69     int 0x10                 ; Вызов BIOS
70
71     add cx, di
72     int 0x10                 ; Вызов BIOS
73
74     sub cx, di
75     sub dx, si
76
77     dec si                   ; Декрементируем счетчик
78     jnz .plot                ; Переход, если не все точки отрисованы
79
80 int 0x10                     ; Отрисовываем последнюю точку
81
82 pop di                       ; Восстанавливаем значения регистров
83 pop ax                       ; Внимание! Восстановление идет в порядке,
84 pop dx                       ;    обратному тому, с котором регистры
85 pop cx                       ;    были помещены в стек.
86 pop si
87
88 ret
```

Задачи для самостоятельного решения

1. Изобразите на экране 15 квадратов разного размера и цветов так, чтобы каждый меньший квадрат располагался в середине большего квадрата.

3.6. Работа с мышью

Рассмотрим работу с периферийными устройствами на примере мыши.

Особенность работы с периферией заключается в том, что для различных событий (например, нажатие клавиши) необходимо назначить обработчики, которые будут вызываться асинхронно. Это позволяет избежать неизящного и плохо масштабируемого кода, в котором в бесконечном цикле происходит опрос состояний периферийных устройств и реагирование при обнаружении истинности некоторых условий.

Итак, мышь работает в графическом режиме, так что первые строки программы осуществляют переход в один из доступных графических режимов с помощью функции `0x00`.

Для работы с мышью предназначено программное прерывание `int 0x33`. Первоначально необходимо инициализировать мышь. Это можно сделать, используя функцию прерывания `int 0x33`, и, как обычно в архитектуре x86, номер функции передается через регистр `ah`.

Затем, для того, чтобы отобразить на экран указатель мыши (в виде белой стрелки), надо вызвать функцию `0x10`.

Как отмечено выше, реакция на события мыши происходит асинхронно путем прерывания выполнения текущего кода и вызова специальной процедуры обработчика. Для назначения обработчика мы вызываем функцию `0x0c`. Затем требуется выбрать, к какому событию будет привязан обработчик. Это могут быть нажатия клавиш мыши, либо ее движение. Выбираем нажатие левой клавиши, этому соответствует число `0x02`, которое надо поместить в регистр `cx`. Наконец, в регистр `dx` запишем адрес обработчика, в нашем случае это адрес кода по метке `MouseHandler`. Не забываем вызвать прерывание `int 0x33`.

После этой процедуры мы должны перейти в бесконечный цикл ожидания событий (между меткой `nopress` и командой условного перехода на эту метку), как и всегда при подобной организации программ. Однако надо предусмотреть и выход из этого цикла. Для этого мы размещаем в памяти байт `was_pressed`, который будет содержать признак того, что кнопка мыши была нажата. Внутри цикла производится загрузка значения из памяти по этому адресу в регистр `al`, затем выполняется проверка равенства этого значения единице. Если в `al` оказалось не единичное значение, в результате выполнения команды `cmp al, 0x01` *не будет* поднят флаг нуля[1], а следовательно, произойдет переход к началу цикла.

Когда признак нажатия окажется единицей, мы покинем бесконечный цикл. Перед завершением работы программы необходимо вернуть систему в первоначальное состояние, т.е. отменить пользовательский обработчик событий мыши с помощью функции `0x14` и вернуть экран в исходный текстовый режим.

Наконец, рассмотрим код обработчика. Поместить в память числовую константу непосредственно нельзя, поэтому мы временно сохраняем регистр `ax` в стеке, затем записываем в `al` единицу, а затем содержимое `al` копируем в память. Перед выходом из обработчика мы возвращаем из стека значение `ax`[2].

Для возврата из обработчика используется специальная команда `retf`.

ЛИСТИНГ 3.6

```
1 ; В программе инициализируется мышь. При нажатии левой
2 ; кнопки мыши программа завершается
3
```

[1] Как мы помним, команда сравнения выполняет не модифицирующее значения регистров вычитание!

[2] Это кажется неэкономным, т.к. можно было бы использовать любой из незадействованных в программе регистров, например, `si`, и сэкономить такты, не сохраняя его в стеке, однако в реальных больших программах программист может забыть о том, что некий регистр все-таки в какой-то части кода используется. Будьте внимательны. Не забывайте также извлекать помещенные в стек значения, чтобы не допустить его переполнения.

```
4  org 0x100
5
6  mov ah, 0x00                    ; Функция BIOS выбора графического режима
7  mov al, 0x12                    ; Режим 640x480
8  int 0x10                        ; Вызов BIOS
9
10 mov ax, 0x0000                  ; Функция инициализации мыши
11 int 0x33                        ; Вызов прерывания
12
13 mov ax, 0x0001                  ; Показать курсор мыши
14 int 0x33                        ; Вызов прерывания
15
16 mov ax, 0x000c                  ; Установка обработчика событий мыши
17 mov cx, 0x0002                  ; Событие - нажатие левой кнопки
18 mov dx, MouseHandler            ; Адрес обработчика
19 int 0x33                        ; Вызов прерывания
20
21 nopress:
22
23     mov al, [was_pressed]
24     cmp al, 0x01                ; Проверяем состояние флага в памяти
25     jnz nopress
26
27 mov ax, 0x0014                  ; Удаление обработчика событий мыши
28 mov cx, 0x0000                  ;
29 int 0x33                        ; Вызов прерывания
30
31 mov ah, 0x00                    ; Функция BIOS выбора графического режима
32 mov al, 0x03                    ; Текстовый режим 80x25, 16 цветов
33 int 0x10                        ; Вызов BIOS
34
35 ret
36
37 MouseHandler:
38
39     push ax
40     mov al, 0x01
41     mov [was_pressed], al           ; Устанавливаем флаг в памяти
42     pop ax
43     retf                            ; Выходим из обработчика события
44
45 was_pressed db 0x00
```

Задачи для самостоятельного решения

1. Разработайте программу, которая будет завершаться не после первого, а после второго нажатия на клавишу мыши.

3.7. Рисования с помощью мыши

Разработаем более сложную программу, в которой курсор мыши при движении будет оставлять за собой след из пикселей некоторого цвета, а при нажатии на левую клавишу мыши цвет пикселей будет изменяться.

Как и предыдущей программе, нам необходимо сначала перейти в графический режим и инициализировать мышь. Для этого используются знакомые нам функции знакомых программных прерываний.

В качестве обрабатываемого события на этот раз выберем любое перемещение мыши. Для этого в регистр cx при вызове функции назначения обработчика необходимо записать значение `0x01`.

Бесконечный цикл организуем с помощью функции ввода BIOS, программа будет завершаться при нажатии на любую клавишу на клавиатуре, как в первых программах с демонстрацией работы в графическом режиме. По нажатию клавиши мы перейдем в конечную часть программы, в которой удаляется пользовательский обработчик события мыши и осуществляется переход в изначальный текстовый режим.

В этой программе интерес представляет обработчик мыши `MouseHandler`. Он вызывается при любом движении мыши. Для того, чтобы изменить (или не изменить) цвет остающихся пикселей, мы должны проверить, нажата ли клавиша мыши. Для этого используется функция `0x03`, которая возвращает в регистр bx набор бит, описывающих состояние мыши. Проверка бит регистра производится с помощью инструкции `test`. Если в регистре поднят нулевой бит, то это означает, что нажата левая клавиша мыши. В этом случае мы переходим в локальную функцию `.change_color`. В этой функции производится изменение значения в памяти, хранящегося по адресу `pointer_color`. Очевидно, что это значение используется в качестве кода цвета при отрисовке точек. Заметим, что поскольку в процессе записи значения в память мы используем регистр ax, а он может использоваться где-то в основной части программы, мы значение этого регистра перед использованием последнего сохраняем в стеке, а затем по окончании работы восстанавливаем.

В конце функции `.change_color` после инкремента значения кода цвета происходит безусловный возврат в обработчик события мыши. Теперь для того, чтобы нарисовать пиксель, нам необходимо на время скрыть курсор мыши. Это делается с помощью функции `0x02`. После этого мы готовы нарисовать точку (точнее, почти готовы, подробности через пару строк). Мы можем сделать это с помощью функции BIOS с номером `0x0c`, как было это было реализовано в ранее написанных программах. Однако тут есть один подводный камень. Функция `0x0c` использует регистр bx как указатель на текущую страницу видеопамяти, а при этом в регистре лежит слово состояния мыши, так что если до вызова `int 0x10` регистр bx не очистить, можно получить непредсказуемый результат[1]. Очистка bx происходит с помощью инструкции вычисления исключающего ИЛИ регистра с его собственным значением.

После того, как новая точка нарисована, можно снова отобразить курсор мыши.

Все части программы разобраны.

ЛИСТИНГ 3.7

```
; В программе движение мыши приводит появлению точек
; на экране. При нажатии на клавишу цвет изображаемых
; точек изменяется.

org 0x100

mov ah, 0x00                    ; Функция BIOS выбора графического режима

mov ax, 0x0012
int 10h

mov ax, 0x0000                  ; Функция инициализации мыши
int 0x33                        ; Вызов прерывания

mov ax, 0x0001                  ; Показать курсор мыши
int 0x33                        ; Вызов прерывания
```

[1] Он по-настоящему непредсказуем, т.к., вообще говоря, мышь может находиться в различных состояниях.

```
17
18 mov ax, 0x000c                  ; Установка обработчика событий мыши
19 mov cx, 0x0001                  ; Событие - любое перемещение мыши
20 mov dx, MouseHandler            ; Адрес обработчика
21 int 0x33                        ; Вызов прерывания
22
23 nokey:
24     mov ah, 0x01                ; Функция чтения клавиатуры
25     int 0x16                    ; Вызов BIOS
26     jz nokey                    ; Переходим, если ничего не прочитано, т.е.
27                                 ; находимся здесь, пока не будет нажата клавиша
28
29 mov ax, 0x0014                  ; Удаление обработчика событий мыши
30 mov cx, 0x0000                  ;
31 int 0x33                        ; Вызов прерывания
32
33 mov ah, 0x00                    ; Функция BIOS выбора графического режима
34 mov al, 0x03                    ; Текстовый режим 80x25, 16 цветов
35 int 0x10                        ; Вызов BIOS
36
37 ret
38
39 MouseHandler:
40
41     mov ax, 0x0003              ; Проверка состояния мыши
42     int 0x33                    ; Вызов прерывания
43
44     test bx, 0x0001             ; Поднят ли бит, отвечающий за сигнализацию
45                                 ; о нажатии кнопки?
46
47     jnz .change_color           ; Если поднят - изменяем цвет
48
49 .further:
50
51     mov ax, 0x0002              ; Функция скрытия курсора
52     int 0x33                    ; Вызов прерывания
53
54     xor bx, bx                  ; Очищаем bx, т.к. этот регистр используется
55                                 ; для указания текущей страницы видеопамяти
56     mov al, [pointer_color]     ; Забираем из памяти значение цвета
57     mov ah, 0x0c                ; Функция отображения точки
58     int 0x10                    ; Вызов BIOS
59
60     mov ax, 0x0001              ; Показываем курсор
61     int 0x33
62
63     retf                        ; Возврат из обработчика события мыши
64
65 .change_color:
66
67     push ax                     ; Запоминаем ax
68     mov al, [pointer_color]     ; Забираем из памяти значение цвета
69     inc al                      ; Увеличиваем номер цвета на 1
```

```
70      mov [pointer_color], al      ; Сохраняем номер цвета в памяти
71      pop ax                       ; Восстанавливаем ax
72
73      jmp .further                 ; Возврат из фрагмента кода
74
75  pointer_color db 1
```

3.8. Работа с часами реального времени

На современных компьютерах в южный мост встраиваются часы реального времени (Real Time Clock), необходимые для непрерывного отсчета времени компьютером. Часы функционируют даже при выключенном основном питании, источником энергии для них служит элемент питания типоразмера CR2032 (диаметром 20 мм, высотой 3.2 мм).

Часы представляют собой автономное периферийное устройство, обмен командами и данными с которым осуществляется через набор портов ввода-вывода.

На начальном этапе мы можем настроить часы так, чтобы они возвращали время в подходящем для нас формате. В нашем случае это будет двоично-десятичный формат (англ. BCD). Для этого мы отправляем часам адрес их управляющего регистра следующим образом: адрес `0x0b` записывается в `al`, а затем содержимое последнего отправляется в порт путем выполнения инструкции `out 0x70, al`. Число `0x70` есть номер порта RTC.

После этого необходимо прочесть ответ от часов[1]. Для этого мы читаем данные из порта `0x71`. Побитовым умножением мы обнуляем в полученном значении третий бит (это и соответствует режиму BCD), и перезаписываем настроечный байт путем отправки его в тот же порт `0x71`.

Теперь мы готовы прочесть и вывести на экран текущие время и дату. Дальнейшая структура программы будет строиться следующим образом: сначала мы записываем в `al` адрес того числа, которое нам требуется для вывода. Это будут последовательно: день, месяц, старшие и младшие цифры года, часы, минуты, секунды. После записи адреса всегда будет вызываться функция `print_rtc` в коде которой непосредственно осуществляется запрос данных от часов и вывод на экран.

Между этими операциями при помощи функции DOS быстрого вывода на экран `int 0x29` будут выводиться символы-разделители (пробелы, дефисы, двоеточия).

Таким образом, остается только разобрать работу вышеупомянутой функции. Первые две ее строчки представляют собой непосредственный запрос нужного в данный момент числа от RTC и прием ответа в регистр `al`. Затем перед преобразованиями в стек сохраняем значение ax.

Преобразования сводятся к тому, что нам надо выделить 4 старших бита и 4 младших бита, и вывести их значения на экран. Для этого мы используем в первом случае операцию логического сдвига влево на 4 бита (4 младших бита при этом пропадают), во втором случае обнуляем логическим умножением на маску 4 старших бита. Перед выводом (который производится также функцией быстрого вывода) требуется скорректировать значение в `al`, т.е. добавить `0x30`, как в самых первых наших примерах.

ЛИСТИНГ 3.8

```
1  ; Программа запрашивает у часов реального времени текущую дату
2  ; и время и вывводит их на экран
3
4  org 0x100
5
6  mov al, 0x0b                 ; 0x0B - управляющий регистр RTC
7  out 0x70, al                 ; Выбираем этот регистр для чтения, порт 0x70
8
9  in al, 0x71                  ; Читаем значение регистра
```

[1] На некоторых компьютерах может потребоваться небольшая задержка после отправки запроса. Будьте внимательны.

```
10 and al, 0b11111011              ; Обнуляем второй бит полученного значения, т.е.
11                                 ; задаем BCD-формат для вывода даты и времени
12
13 out 0x71, al                    ; Отправляем RTC обновленные настройки
14
15                                 ; Далее следует серия запросов к RTC с выводом
16                                 ; полученных данных на экран.
17
18 mov al, 0x07                    ; Номер текущего дня
19 call print_rtc
20
21 mov al, '-'                     ; Символ-разделитель
22 int 0x29                        ; Используем быстрый вывод на экран
23
24 mov al, 0x08                    ; Номер текущего месяца
25 call print_rtc
26
27 mov al, '-'                     ; Символ-разделитель
28 int 0x29                        ; Используем быстрый вывод на экран
29
30 mov al, 0x32                    ; Две старшие цифры года
31 call print_rtc
32
33 mov al, 0x09                    ; Две младшие цифры года
34 call print_rtc
35
36 mov al, ' '                     ; Символ-разделитель
37 int 0x29                        ; Используем быстрый вывод на экран
38
39 mov al, 0x04                    ; Текущий час
40 call print_rtc
41
42 mov al, ':'                     ; Символ-разделитель
43 int 0x29                        ; Используем быстрый вывод на экран
44
45 mov al, 0x02                    ; Текущая минута
46 call print_rtc
47
48 mov al, ':'                     ; Символ-разделитель
49 int 0x29                        ; Используем быстрый вывод на экран
50
51 mov al, 0x00                    ; Текущая секунда
52 call print_rtc
53
54 ret
55
56 print_rtc:
57
58     out 0x70, al                ; Делаем запрос к RTC
59     in al, 0x71                 ; Получаем ответ
60
61     push ax                     ; Запоминаем значение ax перед модификацией
62
```

```
63      shr al, 4                       ; Выделяем старшие 4 бита ответа
64      add al, '0'                     ; Коррекция результата перед выводом
65      int 0x29                        ; Используем быстрый вывод на экран
66
67      pop ax
68
69      and al, 0x0f                    ; Выделяем младшие 4 бита
70      add al, '0'                     ; Коррекция результата перед выводом
71      int 0x29                        ; Используем быстрый вывод на экран
72
73      ret
```

Задачи для самостоятельного решения

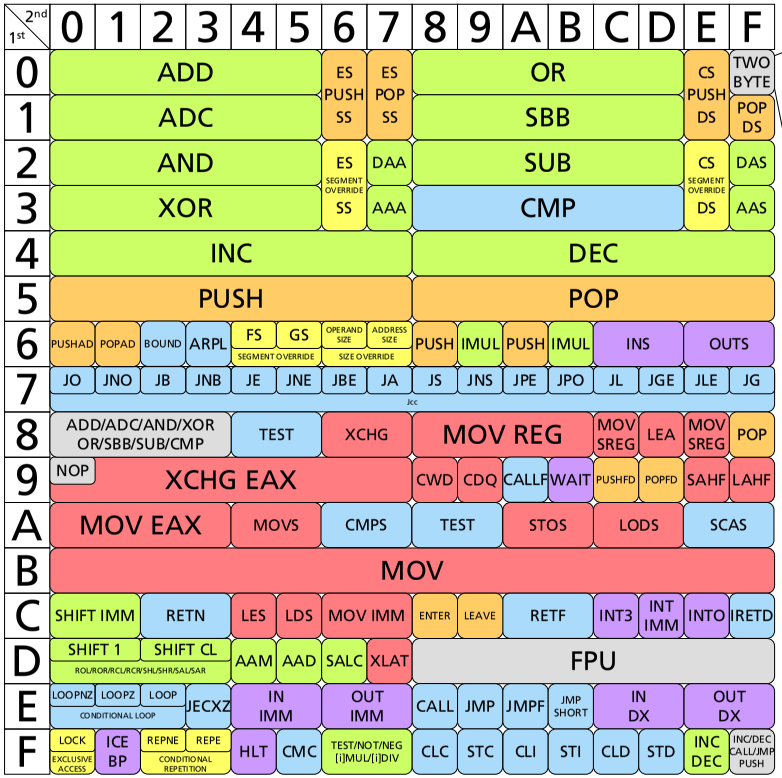
1. Разработайте программу, которая будет непрерывно выводить на экран текущее время.
2. Разработайте программу-таймер. Пользователь с клавиатуры вводит число секунд, программа должна завершиться через соответствующее время.

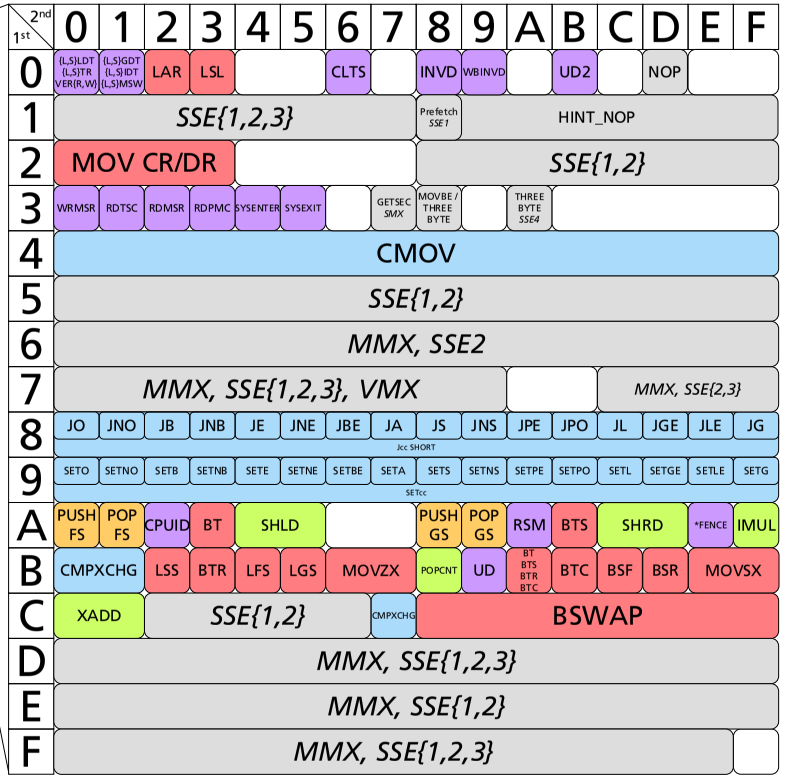
Литература

1. *Duncan R.* Advanced MS DOS programming: the Microsoft guide for Assembly language and C programmers. — Microsoft Press, 1988.
2. *Kerrisk M.* The Linux programming interface: a Linux and UNIX system programming handbook. — No Starch Press, 2010.
3. *Гук М.* Аппаратные средства IBM PC. — Питер, 2006.
4. *Зубков С.* Assembler для DOS, Windows и UNIX. — ДМК Пресс, Питер, 2004.
5. *Иртегов Д.* Введение в операционные системы. — 2-е изд., [перераб. и доп.] — БХВ-Петербург, 2008.
6. *Калашников О.* Ассемблер? Это просто. Учимся программировать. — БХВ-Петербург, 2011.
7. *Максимов Н., Партыка Т., И. П.* Архитектура ЭВМ и вычислительных систем. — Форум, Инфра-М, 2013.
8. *Таненбаум Э., Бос Х.* Современные операционные системы. — 4-е издание. — Питер, 2015.
9. *Харрис Д., Харрис С.* Цифровая схемотехника и архитектура компьютера. — 2015.
10. *Юров В.* Assembler. — Питер, 2010.

Fraunhofer
FKIE
FRAUNHOFER-INSTITUT FÜR KOMMUNIKATION, INFORMATIONSVERARBEITUNG UND ERGONOMIE FKIE

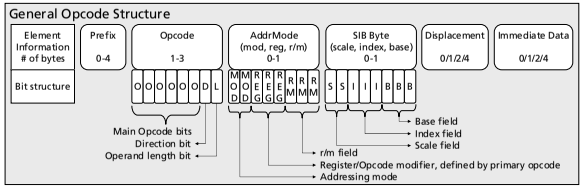
x86 Opcode Structure and Instruction Overview





Arithmetic & Logic
Memory
Stack
Control Flow & Conditional

Prefix
System & I/O
No Operation (NOP) / Multiple Instructions / Extended Instruction Set



Addressing Modes

mod	00		01		10		11
r/m	16bit	32bit	16bit	32bit	16bit	32bit	r/m // REG
000	[BX+SI]	[EAX]	[BX+SI+disp8]	[EAX]+disp8	[BX+SI]+disp16	[EAX]+disp32	AL / AX / EAX
001	[BX+DI]	[ECX]	[BX+DI+disp8]	[ECX]+disp8	[BX+DI]+disp16	[ECX]+disp32	CL / CX / ECX
010	[BP+SI]	[EDX]	[BP+SI+disp8]	[EDX]+disp8	[BP+SI]+disp16	[EDX]+disp32	DL / DX / EDX
011	[BP+DI]	[EBX]	[BP+DI+disp8]	[EBX]+disp8	[BP+DI]+disp16	[EBX]+disp32	BL / BX / EBX
100	[SI]	SIB	[SI]+disp8	SIB+disp8	[SI]+disp16	SIB+disp32	AH / SP / ESP
101	[DI]	disp32	[DI]+disp8	[EBP]+disp8	[DI]+disp16	[EBP]+disp32	CH / BP / EBP
110	disp16	[ESI]	[BP]+disp8	[ESI]+disp8	[BP]+disp16	[ESI]+disp32	DH / SI / ESI
111	[BX]	[EDI]	[BX]+disp8	[EDI]+disp8	[BX]+disp16	[EDI]+disp32	BH / DI / EDI

SIB Byte Structure

encoding	scale (2bit)	Index (3bit)	Base (3bit)
000	$2^{[0]}$=1	[EAX]	EAX
001	$2^{[1]}$=2	[ECX]	ECX
010	$2^{[2]}$=4	[EDX]	EDX
011	$2^{[3]}$=8	[EBX]	EBX
100	--	none	ESP
101	--	[EBP]	disp32 / disp8+ [EBP] / disp32 + [EBP]
110	--	[ESI]	ESI
111	--	[EDI]	EDI

SIB value = index * scale + base

v1.0 – 30.08.2011
Contact: Daniel Plohmann – +49 228 73 54 228 – daniel.plohmann@fkie.fraunhofer.de

Source: Intel x86 Instruction Set Reference
Opcode table presentation inspired by work of Ange Albertini