PS1: к тексту **"Нарушение защиты памяти между задачами Meltdown".**
http://grizlyk1.narod.ru/sys_new/msg_01/x86_meltdown_200719.pdf

Про **"DS/ES это рабочие сегментные регистры x86"**.

Полянин М.А.
09 июля 2020.

Вот на это замечание в исходном тексте:
6. Процессор x86 обладает множеством иных особенностей:
г) процессор x86 имеет только встроенные в опкоды префиксы сегментов
для модульной архитектуры надо уметь индексировать сегментные регистры, а не только статически задавать их префиксы в опкоде

Мы получили пояснение, что: "*DS/ES это рабочие сегментные регистры x86, поэтому индексация сегментов на x86 есть и разработчики x86 хотя и не пишут программы, но программисты все же рассказали им о потребностях модульных программ в индексации сегментов для шаблонов кода времени выполнения*".

То что DS/ES это рабочие сегментные регистры, а не регистры предзагруженного контекста, т.е. идея что каждое обращение к памяти на x86 должно пройти одновременно с установкой DS или ES, это само по себе интересное замечание по архитектуре x86, такое поведение очень хорошо вписывается в систему опкодов x86.

Сегментные регистры DS/ES называются рабочими потому, что не хранят никакого предзагруженного контекста и заново загружаются каждый раз когда нужен доступ к памяти:
- как в обычной функции при доступе через указатель переданный в качестве параметра (сегмент известен только во время выполнения);
- так и при статическом доступе (сегмент известен во время компиляции), если нет другого выделенного сегментного регистра, который хранит нужный сегмент уже как предзагруженный контекст модуля.

Тогда давайте посмотрим, правда ли что став рабочими сегментные регистры DS/ES обеспечат механизм эквивалентный индексации сегментов в плоском указателе?

Часть 1

В качестве примера использования сегментных регистров DS/ES как рабочих, мы:
- напишем пару любых простых функций;

- напишем любой простой пример программы эти функции использующей;

и в итоге посмотрим на код который получается, чтобы оценить хорошо ли использовать сегментные регистры DS/ES как рабочие, посмотрим слишком ли тяжелы получаются вычисления от перезагрузки DS/ES.

1.

Язык высокого уровня (типа C) тут не годится, нам надо формировать пролог и эпилог, передавать параметры в соответствии с нужной нам ролью регистров, поэтому для нашего примера подходит:

- какой нибудь OMF асм для x86 (например tasm версии 3.1);
- в паре с каким либо OMF линкером (например Microsoft link версии 4).

Парный к tasm линкер tlink работает с ошибками, например неправильно выравнивает сегменты нулевого размера:

если новый сегмент в группе имеет нулевой размер, то он не влияет на align следующего сегмента в группе (а должен влиять, он выравнивает все следующие сегменты в группе);

```
например не будет работать вот это "00148H -> align para -> 00150H"
      00000H 00147H 00148H _TEXT
      00150H 00150H 00000H _BEGTD
```

Все асмы без исходников и все сложные асмы с исходниками (вряд ли вы проверили все исходники такого большого размера) приходится каждый раз лично (вручную) проверять, что они генерят то что нужно, а не делают ошибки в самых простых вещах. Это все довольно неприятно и необычно для транслятора или языка программирования, в которых явные ошибки вообще довольно редко попадаются (даже в стандартных библиотеках, сам программист допускает ошибки намного чаще). Если рабочее подмножество опций у этих асмов и есть, то об этом знают только сами авторы этих асмов.

1.1

Список файлов примера в архиве (http://grizlyk1.narod.ru/sys_new/msg_01/x86_meltdown_090720-ps1-v01.zip)

заголовки	
def.asi	базовые настройки (первый файл для включения)
except.asi	применяемый формат исключений
fproto.asi	прототипы применяемого формата функций

*ehead.asi	первый код в модуле main (включение только в модуль main)
раздельно компилируемые модули с определениями функций	
ex01.asm	_memset
ex02.asm	_memcpy
ex03.asm	_memchr
*ex01r.asm	entry (модуль main)

Выглядят эти функции примерно так (ex01.asm)

```
%noincl
include def.asi
include except.asi
include fproto.asi

;*****************************************************************
comment #
     функция memset по правилам рабочих сегментных регистров DS/ES
#
public _memset

;void *far memset( void *far const dst, const char c, const unsigned n ) throw();
@st_TEXT

_memset         proc near

     arg        @h1: word, @h2: far ptr byte, @v1: far ptr byte, @v2: byte: 2, @v3: word = @vsz
     ;returns

     ;@h1 reserved for _throw_ip
     _ret_ptr              equ @h2
     ;_this                equ @h3
```

```
     v_dst                equ @v1
     v_c                  equ @v2
     v_n                  equ @v3

     ;_stack_size         equ @h5
     @arg_size            equ ( @vsz )

     ;local     @a1: word = _lsz
     @local_size = ( 0 )

     @assume
     enter @local_size, 0

     ;
     les   di, [v_dst]

     ;if( !dst )throw;
     or    di, di
     jz    @@throw00
     mov   ax, es
     ;в защищенном режиме код проверки селектора: test  ax, 0FFFCh
     or    ax, ax
     jnz   @@else00

@@throw00:
     @throw_on_return
     jmp   @@ex00

@@else00:
     lds   bx, [_ret_ptr]

     ;if( !ret )skip;
     or    bx, bx
```

```
     jz    @@skip01
     mov   ax, ds
     or    ax, ax
     jz    @@skip01

@@else01:
     mov   word ptr [bx], di
     mov   word ptr [bx+2], es

@@skip01:
     ;
     mov   cx, [v_n]
     jcxz  @@ex00

     ;assume we use opcode that tells for cpu to make a kind of "memory coalescing"
     mov   al, byte ptr [v_c]
     cld
     rep   stosb

@@ex00:
     leave
     ret   @arg_size

_memset         endp
_TEXT      ends

     end
```

а вот этот же текст (ex01.asm) в редакторе far с подсветкой

```
    @assume
    enter @local_size, 0

    ;
    les   di, [v_dst]

    ;if( !dst )throw;
    or    di, di
    jz    @@throw00
    mov   ax, es
    ;в защищенном режиме код проверки селектора: test  ax, 0FFFCh
    or    ax, ax
    jnz   @@else00

@@throw00:
    @throw_on_return
    jmp   @@ex00

@@else00:
    lds   bx, [_ret_ptr]

    ;if( !ret )skip;
    or    bx, bx
    jz    @@skip01
    mov   ax, ds
    or    ax, ax
    jz    @@skip01
```

Вот что получается после компиляции (ex01.lst)

```
      1                         ;
      2                         include def.asi
2     3    0000                 _TEXT   segment byte public 'CODE'    use16
2     4    0000                 _TDATA  segment dword public 'TDATA' use16
2     5    0000                 _TBSS   segment dword public 'TBSS' use16 uninit
2     6    0000                 _STACK  segment dword stack 'STACK' use16 uninit
      7                         include except.asi
      8                         include fproto.asi
      9
     10                         ;*************************************************************
     11                         comment #
     12                              функция memset по правилам рабочих    сегментных регистров DS/ES
     13                         #
     14                         public _memset
     15
     16                         ;void *far      memset( void *far const dst, const char c
     17                         @st_TEXT
1    18    0000                 _TEXT   segment byte public 'CODE'    use16
     19
     20    0000                 _memset         proc near
     21
     22          =000E                     arg       @h1: word, @h2: far ptr byte,
     23
     24                               ;returns
     25
     26                               ;@h1 reserved for _throw_ip
     27                               _ret_ptr                  equ @h2
     28                               ;_this               equ @h3
     29
     30                               v_dst                equ @v1
     31                               v_c                       equ @v2
     32                               v_n                       equ @v3
```

```
     33
     34                                 ;_stack_size          equ @h5
     35          =000E                  @arg_size                  equ ( @vsz )
     36
     37                                 ;local  @a1: word = _lsz
     38          =0000                  @local_size = ( 0 )
     39
     40                                 @assume
1    41                                       assume cs:FGROUP, ss:AGROUP, ds:nothing, es:nothing
     42    0000  C8 0000 00             enter @local_size, 0
     43
     44                                 ;
     45    0004  C4 7E 0A               les   di, [v_dst]
     46
     47                                 ;if( !dst )throw;
     48    0007  0B FF                  or       di, di
     49    0009  74 06                  jz       @@throw00
     50    000B  8C C0                  mov   ax, es
     51                                 ;в    защищенном режиме код проверки селектора: test   ax, 0FFFCh
     52    000D  0B C0                  or       ax, ax
     53    000F  75 08                  jnz   @@else00
     54
     55    0011                   @@throw00:
     56                                 @throw_on_return
1    57    0011  8B 46 04                 mov   ax, ss: word    ptr [_throw_ip]
1    58    0014  89 46 02                 mov   ss: word ptr    [_ret_ip], ax
     59    0017  EB 1D                  jmp   @@ex00
     60
     61    0019                   @@else00:
     62    0019  C5 5E 06               lds   bx, [_ret_ptr]
     63
     64                                 ;if( !ret )skip;
     65    001C  0B DB                  or       bx, bx
```

```
    66    001E  74 0B              jz      @@skip01
    67    0020  8C D8              mov   ax, ds
    68    0022  0B C0              or      ax, ax
    69    0024  74 05              jz      @@skip01
    70
    71    0026                  @@else01:
    72    0026  89 3F                   mov   word ptr [bx], di
    73    0028  8C 47 02                mov   word ptr [bx+2], es
    74
    75    002B                  @@skip01:
    76                                  ;
    77    002B  8B 4E 10                mov   cx, [v_n]
    78    002E  E3 06                   jcxz  @@ex00
    79
    80                                  ;assume we use opcode that tells for cpu to make a
    81    0030  8A 46 0E                mov   al, byte ptr    [v_c]
    82    0033  FC                      cld
    83    0034  F3> AA                  rep   stosb
    84
    85    0036                  @@ex00:
    86    0036  C9                      leave
    87    0037  C2 000E                 ret   @arg_size
    88
    89    003A                  _memset         endp
    90    003A                  _TEXT      ends
    91
    92                                  end
```

вот так выглядит вызов этой функции (ex01r.lst)

```
     75                                 ;set _memset params
     76    001B  6A 04                  push  4
     77    001D  68 FF02                push  0FF02h
     78
     79    0020  8D 86 FC04             lea   ax, byte ptr [bp - @local_size + 4]
     80    0024  16                     push  ss
     81    0025  50                     push  ax
     82
     83                                 ;here may push 0:0 if no need return
     84    0026  8D 86 FC00             lea   ax, byte ptr [bp - @local_size]
     85    002A  16                     push  ss
     86    002B  50                     push  ax
     88
     89    002C  68 006Br               push  offset @@catch00
     90    002F  E8 0000e               call  _memset
```

Единственные сегментные регистры, которые можно использовать как предзагруженный контекст модуля при статическом обращении это:

CS для сегмента кода (f для схемы сегментов fdatu)

SS для сегмента стека нити (a для схемы сегментов fdatu)

В добавок к нашему желанию иметь предзагруженный контекст модуля, путаницу вносят и соглашения ПО об использовании регистров, в том числе сегментных DS/ES, такие соглашения настолько для нас стали привычными, что мы уже забыли что они не имеют отношения к самому процессору x86.

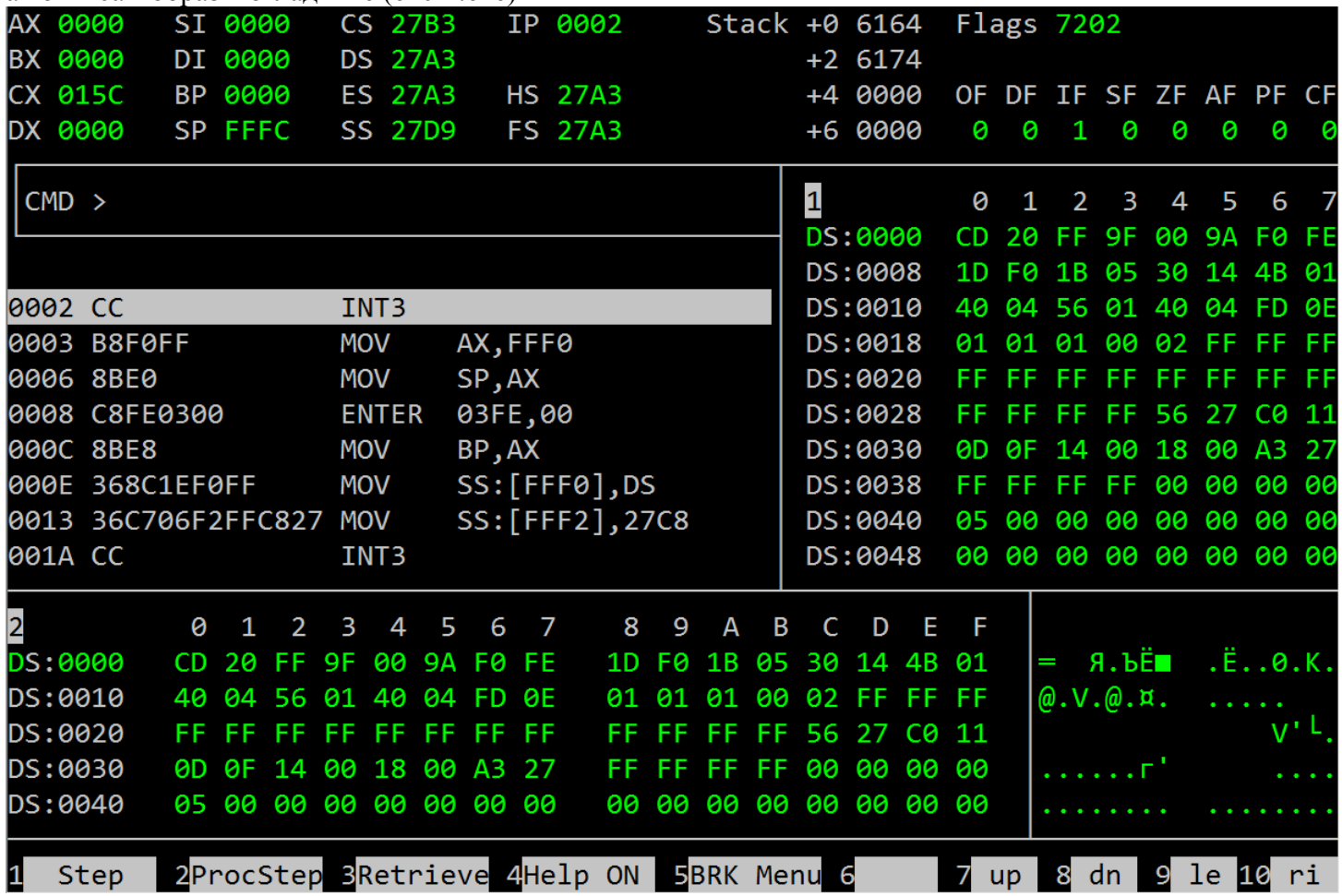
Как видите превращение DS/ES в рабочие регистры визуально вообще не повлияло на внешний вид кода для x86, в программе не появилось никаких чудовищных конструкций связанных с проблемами перезагрузки DS и с утратой предзагруженного в DS состояния DGROUP.

DS просто такой же рабочий регистр как например DI, и загрузка данных в любой рабочий регистр x86 это просто загрузка параметра для опкода, который иначе был бы указан прямо в самом опкоде, это не боле сложно чем выполнение самого опкода, обычных регистров на x86 нет, а оптимизация загрузки регистров тогда это в основном устранение redundant load.

вот карта образа (ex01r.map)

```
 Start  Stop   Length Name                   Class
 00000H 00000H 00000H _BEGF                 BEGF
 00000H 00147H 00148H _TEXT                 CODE
 00150H 00150H 00000H _BEGTD                BEGTD
 00150H 0015BH 0000CH _TDATA                TDATA
 0015CH 0025FH 00104H _TBSS                 TBSS
 00260H 00260H 00000H _BEGST                BEGST
 00260H 1025BH 0FFFCH _STACK                STACK

 Origin   Group
 0026:0   AGROUP
 0000:0   FGROUP
 0015:0   TGROUP

 Address         Publics by Value

 0000:0000       _null_jmp
 0000:0002       entry
 0000:0086       _memset
 0000:00C0       _memcpy
 0000:0106       _memchr
 0015:0000       _null_data_ptr
 0015:0004       _exc_limit
 0015:0006       _exc_base
 0015:0008       _exc_sp
 0015:0010       _exc_stack

Program entry point at 0000:0002
```

а вот и сам образ в отладчике (ex01r.exe)



Операнды и результаты в регистрах x86 ведут себя как write through cache данных, регистры не содержат не сохраненного в памяти состояния, поэтому в наших примерах сразу исчезли такие вещи, как бесконечные свопы регистров в стек и обратно с предварительной загрузкой регистров x86 из постоянного места хранения данных в памяти.

Пропал код такого вида:

```
    push ds
```

```
    push es
    push di
    push si

    mov  di, 1
    mov  si, 2

    call _f1
         push ds
         push es
         push di
         push si

         mov  di, 3
         mov  si, 4

         pop  si
         pop  di
         pop  es
         pop  ds
         ret

    add  si, 0
    add  di, 1

    call _f2
         push ds
         push es
         push di
         push si

         mov  di, 3
         mov  si, 4
```

```
        pop  si
        pop  di
        pop  es
        pop  ds

    add  si, 1
    add  di, 0

    pop  si
    pop  di
    pop  es
    pop  ds
    ret
```

Часть 2

Как только программисту удается сообразить, что в x86 нет регистров и операции с памятью допустимы, он перестает предпринимать попытки бессмысленной оптимизации, настроение программиста улучшается, он успокаивается, а код становится удобным, оптимальным и рациональным.

Однако как бы мы не описывали примеры такого оптимального для архитектуры x86 кода, на практике есть много случаев когда программисты пытаются задействовать на x86 регистры (которых на x86 нет) как регистровый файл. Почему такое происходит?

Зачем именно это нужно, когда часть регистров x86 хранят исходные операнды и накапливают промежуточные результаты вычислений, кроме общих слов что "регистры ускоряют работу программы"?

2.
Для учебного примера рассмотрим компьютер XT, скорость работы оптимально настроенной шины памяти для него равна 1 Мбайт/с.

Вычислим на таком компьютере функцию y=k*x+b из диапазона x от -10.0 до 10.0 в виде выходного массива значений в памяти из диапазона y от -2000.0 до 2000.0 (k < (2000-b)/10).

Функция y=k*x+b интересна тем что процессоры обработки сигналов выполняют такие операции как элементарные, а также тем что для x86

достаточно регистров операндов опкодов для хранения в них всех параметров функции. Также будем считать что операция умножения выполняется на идеализированном x86 достаточно быстро и не является тем компонентом, который определяет характер (скорость) работы всей функции.

При хранении всех параметров в памяти функция y=k*x+b требует четыре операции с памятью на каждый шаг вычислений:
    три чтения (k,x,b)
    одну запись (y)
скорость вычислений результата (1 Мбайт/с div 4) = 250 Кбайт/с

При хранении всех параметров в регистрах функция y=k*x+b требует одну операцию с памятью на каждый шаг вычислений:
    одну запись (y)
скорость вычислений результата = 1 Мбайт/с

2.1
При хранении всех параметров в регистрах эта программа практически работает в 4 раза быстрее.

"Зачем нужны регистры" это опять из серии "сложно о простом":
    - если крысу в опыте вообще не поить и не кормить, то она заболеет и даже может быть умрет;
    - ну или лучше жить в плановой интернациональной экономике, чем иметь в РФ в рыночной нациоанальной экономике 15 миллионов нищих, бездомных и безработных "с высокой производительностью труда".

Про экономику. Для атеиста (т.е. для отрицающего десять христианских заповедей как базовые ценности) это совершенно обычное дело, верить что "благодаря высокой производительности труда при рыночной нациоанальной экономике" на работе много людей уже не нужно, потому что меньше людей выполняют ту же самую работу, но одновременно оставшимся на улице миллионам людей не хватает даже на нищенскую жизнь, и так происходит тысячелетиями во все странах мира (мы улыбаемся тем, кто улыбается когда политики тысячелетиями говорят то, что они уже когда то слышали).

Никакого противоречия (это тоже самое, но при этом часть того же самого физически отсутствует) атеист тут не видит, потому что не подозревает ни о существовании:
    системы базовых ценностей, которая используется при всех рассуждениях как "базовая точка опоры";
    физических и математических моделей применяемых для описания всех рассуждений и всех явлений;
атеист просто ест, спит, ну и т.п., живет как поросенок в стойле, другие люди рыночный нациоанальный режим лживых воров, в котором каждому надо:
    преступным путем создавать;

преступным путем удерживать;
преступное неравновесие для извлечения личной выгоды;
не станут поддерживать.

Если вам не нравиться лгать и грабить, то рыночная нациоанальная экономика с "чурками рабами" вам просто ни к чему, непонятно зачем бы она была нужна, она больше ничего не позволяет делать, кроме как лгать и грабить, это рай для тех кому именно лгать и грабить и нужно.

2.2
В 4 раза быстрее, для программы это много или мало?

Разница "в 2 раза быстрее" уже настолько большая что видна невооруженным глазом, например при работе обычных приложений в системе, при тестировании (при преднамеренном внимании на производительность) одноканальную и двухканальную память можно отличить при оценке визуально, без точных замеров на тестах.

Разница "в 4 раза быстрее" это просто совсем много. Если в вашей программе очень много функций, которые поддаются такой оптимизации, то вся ваша программа при оптимизации работает заметно лучше. Такая оптимизация позволяет например делать на 286 компьютере с 640К памяти такие игры как вольф 3Д.

Глядя на то, как у современного софта печально медленно обрабатываются окна содержащие всего лишь текст и картинки (сделано по принципу: работа программиста очень дорогая, компьютер же работает за электричество (от АЭС), усилили компьютер в 10 раз и это компенсирует недостатки ПО и все будет дешево и много), старинный вольф 3Д на 640К кажется невероятным чудом, в этой игре разрешение экрана и цветовая глубина ограничена прежде всего объемом доступной памяти реального режима (разрешение (в пикселях, символах) окна на экране сильно влияет на потребление памяти).

2.3
В нашем примере с вычислением "y=k*x+b" мы видим и некоторые ограничения на пригодность задач к оптимизации путем использования регистров.

Программы которые:
- имеют большой: входной и выходной файл;
- не имеют малочисленные наборы: константных параметров и накопительных результатов для больших циклов работы с такими наборами;
- имеют параллельную обработку одних и тех же данных с помощью нескольких процессоров;

не поддаются оптимизации с помощью регистров.

Обычные функции не поддаются оптимизации с помощью регистров, для них важнее большие объемы быстрой памяти, для них и сделана архитектура x86 которая в 1980 году на 16 бит компьютере имеет 20 бит память со скоростью в 1 Мбайт/с (в то время 100 нс память это быстро).

2.4
Однако архитектура x86 однозначно имеет такой минус, как отсутствие регистров, это ухудшает работу всех особых алгоритмов, которые оптимизируются именно с помощью регистрового файла.

Естественный для архитектуры x86 путь добавления регистрового файла это установка сопроцессора с набором целочисленных регистров и интеграция его обслуживания в систему (флаг TSx в MSW, постоянное место атрибута типа TSS в контексте TSS, постоянное место самих регистров в контексте TSS, ну и т.п.).

Регистровый файл может обслуживаться как ALU самого процессора, так и отдельным LIW сопроцессором. Такой регистровый файл не может подменять контекст сопроцессора x87, это независимые вещи.

MMX сделан совместимым с x87 чтобы позволить запускать все новые приложения (которые не будут изготовлены в двух версиях отображения MMX: на контекст x87 или отдельно) для новых процессоров на всех старых ОС которые не обновляются, это возможность применить улучшения нового процессора на старых ОС (это реакция производителя x86 на предыдущий случай невостребованного "ортогонального" защищенного режима в 286).

Наличие TSS ограничивает число одновременно выполняющихся на x86 раздельных аппаратных конфигураций только возможностями конкретной модели x86 процессора по аппаратной реализации заданной в каждом TSS архитектуры компьютера (системы команд, набора регистров и т.п.), потому что число элементов на кристалле процессора ограничено и не может быть сколь угодно много встроенных процессоров разного типа.

2.5
Итак, напишем оптимизированную для регистров x86 функцию нашего примера y=k*x+b

Мы не будем оптимизировать линейное умножение (k*x+b), потому что в сложной функции множитель случайный и поставляется во время выполнения.

Используем 16 бит арифметику с 4 битной фиксированной точкой (shl 4), точность 10^-1 (0.1), т.е.

значения от -2048 до 2048 (12 бит)
шаг 1/16 (4 бит)

32 битные регистры намного лучше для арифметики с фиксированной точкой, например при 10 битах по точности дают 10^-3 (.001), что достаточно для обычных инженерных расчетов, а также значения целой части вырастают до 4 миллионов (22 бита)

2.5.1
Вот эта функция в оптимизированном для регистров виде (ex11.asm)

```
     ;inline
     arg   obuf: r(es:di), k: r(cx), b: r(bx)
     local x: r(si)

     ;set ptr
     ;;if(!obuf)throw;
     @throw_if_null es, di, ERR_ZPTR

     ;set x
     mov   ax, (-10 shl 4)
     mov   x, ax

     ;set loop
     cld

     ;;optim out for es:di const in loop
     ;;optim out for ax const in loop
@@next_y:
     mul   ax, k
     ;@throw_if_notz dx, 0FFF0h, ERR_EVAL

     ;;после умножения (0.1 * 0.1) получили в dx:ax число с 8 битной точностью (0.01)
     ;;вернем точность 4 бит сдвинув результат арифметически вправо на 4
     ;;shrd  ax, dx, 4
     shr   ax, 4
     shl   dl, 4
```

```
    or    ah, dl

    add   ax, b
    stosw

    inc   x
    mov   ax, x

    cmp   ax, (10 shl 4)
    jle   @@next_y
```

она же после компилирования (ex11.lst)

```
    21                              ;arg   obuf: r(es:di), k: r(cx), b: r(bx)
    22                              ;local x: r(si)
    23          =    (cx)           k     equ (cx)
    24          =    (bx)           b     equ (bx)
    25          =    (si)           x     equ (si)
    26
    27                              @assume
1   28                              assume cs:FGROUP, ss:AGROUP, ds:nothing, es:nothing
    29
    30                              ;set ptr
    31                              ;;if(!obuf)throw;
    32                              ;@throw_if_null es, di, ERR_ZPTR
    33
    34                              ;set x
    35    0000  B8 FF60             mov   ax, (-10 shl 4)
    36    0003  8B F0               mov   x, ax
    37
    38                              ;set loop
    39    0005  FC                  cld
    40
    41                              ;;optim out for es:di const in loop
```

```
42                                  ;;optim out for ax const in loop
43    0006                  @@next_y:
44    0006  F7 E1                   mul   k
45                                  ;@throw_if_notz dx, 0FFF0h, ERR_EVAL
46
47          ;;после умножения (0.1 * 0.1) получили в dx:ax число с 8 битной точностью (0.01)
48          ;;вернем точность 4 бит сдвинув результат арифметически вправо на 4
49                                  ;;shrd  ax, dx, 4
50    0008  C1 E8 04                shr   ax, 4
51    000B  C0 E2 04                shl   dl, 4
52    000E  0A E2                   or    ah, dl
53
54    0010  03 C3                   add   ax, b
55    0012  AB                      stosw
56
57    0013  46                      inc   x
58    0014  8B C6                   mov   ax, x
59
60    0016  3D 00A0                 cmp   ax, (10 shl 4)
61    0019  7E EB                   jle   @@next_y
```

2.5.2

А вот эта же функция в естественном для x86 коде,т.е. без применения регистрового файла (ex12.asm)

```
;near
arg   obuf: far ptr word, k: word, b: word = @vsz
local x:word, p: far ptr word = @lsz

enter @lsz, 0

;set ptr
;;check sel
les   di, dword ptr [obuf]
;;if(!obuf)throw;
```

```
     @throw_if_null es, di, ERR_ZPTR
     ;;
     mov   word ptr [p], di
     mov   word ptr [p+2], es

     ;set x
     mov   ax, (-10 shl 4)
     mov   word ptr [x], ax

     ;set loop
     les   di, dword ptr [p]
     cld
     mov   ax, word ptr [x]

     ;;optim out for es:di const in loop
     ;;optim out for ax const in loop
@@next_y:
     mul   ax, word ptr [k]
     ;@throw_if_notz dx, 0FFF0h, ERR_EVAL

     ;;после умножения (0.1 * 0.1) получили в dx:ax число с 8 битной точностью (0.01)
     ;;вернем точность 4 бит сдвинув результат арифметически вправо на 4
     ;;shrd  ax, dx, 4
     shr   ax, 4
     shl   dl, 4
     or    ah, dl

     add   ax, word ptr [b]
     stosw

     ;;can not optim out (es:di is write through)
     mov   word ptr [p], di
```

```
    mov   ax, word ptr [x]
    inc   ax
    ;;can not optim out (ax is write through)
    mov   word ptr [x], ax

    cmp   ax, (10 shl 4)
    jle   @@next_y

    leave
    ret   @lsz
```

она же после компиляции (ex12.lst)

```
22                              ;near
23          =0008               arg   obuf: far ptr word, k: word,    b: word    = @vsz
24          =0006               local x: word, p: far ptr word = @lsz
25
26    0000  C8 0006 00          enter @lsz, 0
27
28                              ;set ptr
29                              ;;check sel
30    0004  C4 7E 04            les   di, dword ptr [obuf]
31                              ;;if(!obuf2)throw;
32                              ;@throw_if_null es, di, ERR_ZPTR
33                              ;;
34    0007  89 7E FA            mov   word ptr [p], di
35    000A  8C 46 FC            mov   word ptr [p+2], es
36
37                              ;set x
38    000D  B8 FF60             mov   ax, (-10 shl 4)
39    0010  89 46 FE            mov   word ptr [x], ax
40
41                              ;set loop
42    0013  C4 7E FA            les   di, dword ptr [p]
```

```
    43    0016  FC                    cld
    44    0017  8B 46 FE              mov   ax, word ptr [x]
    45
    46                                ;;optim out for es:di const in loop
    47                                ;;optim out for ax const in loop
    48    001A                  @@next_y:
    49    001A  F7 66 08              mul   word ptr [k]
    50                                ;@throw_if_notz dx, 0FFF0h, ERR_EVAL
    51
    52          ;;после умножения (0.1 * 0.1) получили в dx:ax число с 8 битной точностью (0.01)
    53          ;;вернем точность 4 бит сдвинув результат арифметически вправо на 4
    54                                ;;shrd  ax, dx, 4
    55    001D  C1 E8 04              shr   ax, 4
    56    0020  C0 E2 04              shl   dl, 4
    57    0023  0A E2                 or    ah, dl
    58
    59    0025  03 46 0A              add   ax, word ptr [b]
    60    0028  AB                    stosw
    61
    62                                ;;can not optim out (es:di is write through)
    63    0029  89 7E FA              mov   word ptr [p], di
    64
    65    002C  8B 46 FE              mov   ax, word ptr [x]
    66    002F  40                    inc   ax
    67                                ;;can not optim out (ax is write through)
    68    0030  89 46 FE              mov   word ptr [x], ax
    69
    70    0033  3D 00A0               cmp   ax, (10 shl 4)
    71    0036  7E E2                 jle   @@next_y
    72
    73    0038  C9                    leave
    74    0039  C2 0006               ret   @lsz
```

2.6
Код обоих функций (регистровой и не регистровой) внешне выглядит довольно схоже, однако регистровый вариант в 4 раза быстрее (напомним, что не все функции подаются такой оптимизации).

Регистров в архитектуре x86 нет, но имеется L1 кэш с хранящимися в нем блоками памяти размером с кэш линию. Размер кэш линии это характеристика модели x86 процессора (аналогично тому как меняется размер регистра ALU или ширина системной шины).

Качество (размер, скорость и число входов) L1 кэша становится очень важным для тех алгоритмов, которые оптимизируются путем регистрового файла, на x86 они начинают работать лучше, если качество L1 кэша растет.

И сам по себе (безотносительно к отсутствию регистров на x86), несмотря на то что L1 кэш намного медленнее чем регистры, он намного больше типового регистрового файла и это дает возможность применять на x86 алгоритмы, которые не оптимизируются с помощью регистров из за больших размеров промежуточных данных, но оптимизируются именно с помощью L1 кэша (благодаря максимизации произведения требуемый_размер*скорость), это что то среднее между регистровой и нерегистровой схемой.

Часть 3

Итак, все же вернемся к нашей теме об индексах сегментов. Написали мы тестовую программу, посмотрели на использование DS/ES в качестве рабочих сегментных регистров, и теперь подумаем, а что же у нас с индексацией то сегментных регистров?

Если понять что DS/ES это рабочие сегментные регистры, то становится ясно что для реального режима загрузка селектора в DS/ES пожалуй это и получается требуемая нами индексация сегментных регистров.

В реальном режиме нет дескрипторной таблицы и индекс сразу преобразуется в физический адрес, индекс сегмента и старшая часть физического адреса базы сегмента в реальном режиме совпадают и нет нужды предзагружать контекст в сегментный регистр (контекст модуля как бы всегда предзагружен, надо просто выбрать индексом нужный сегмент).

Это неожиданно. Оказалось что DS/ES это не просто рабочие сегментные регистры, а любой сегментный регистр x86 (который мы хотели бы индексировать в плоском указателе) для реального режима сам играет роль индексного регистра сегмента (хранит индекс сегмента), а программно доступных сегментных регистров в реальном режиме нет совсем (они вычисляются автоматически, в контекст модуля нечего предзагружать).

Неожиданно потому что традиции программного обеспечение для реального режима x86 трактуют сегментные регистры как предзагруженный контекст модуля и не сильно стремятся использовать сегментный регистр x86 как индекс сегмента.

Как видим разработчики x86 последовательны в своих намерениях и преследуя свою концепцию "безрегистрового" процессора, ориентированного на хранение всех данных в памяти, а не в явно выделенных программистом кэширующих регистрах, не ограничились такими полумерами как отказ от регистров общего назначения, в x86 нет даже и сегментных регистров, есть только индексы сегментов.

Вернее сегментные регистры в реальном режиме на x86 есть, но они программно недоступны.

3.1
Отметим что в реальном режиме селектор это именно индекс сегмента, а не особенный длинный указатель плоской памяти размером 32 бита.

Индекс размером 16 бит это не слишком ли много? Поскольку x86 ориентирован на выравнивание 16 бит и имеет 16 битную шину (если намеренно не принять мер по ее сужению до 8 бит), то все индексы шириной от 1 до 16 бит будут с одинаковой эффективностью читаться из памяти, нет большого резона экономить (разве что в отведенной под индекс памяти хранить какой нибудь байт данных по нечетному адресу +3).

В реальном режиме упаковать (индекс + смещение) в 16 бит почти невозможно, если это сделать поместив индекс в младшие биты, то например размер char в абстрактной машине C для x86 станет равным 16 или даже 32 бит, что плохо.

3.2
Но вот в 32 битном смещении уже есть разные варианты помещения индекса в пределах 32 бит сверху.

Размещение индекса в плоском указателе ограничивает максимальный размер сегмента, но чем больше битность плоского указателя, тем выгоднее размещать в таком указателе индексы сегментных регистров (тем меньше проблем для практических задач дает снижение максимального размера сегмента):

> в добавок для x86 линейный адрес ограничен размером 4Г и в эти 4Г надо уместить !четыре базовых сегмента размером по 4Г из схемы сегментов "fdatu" (пусть даже на x86 не поддержаны модули и сегментов только четыре), что дает еще меньше проблем при снижении максимального размера сегмента (они все равно все не помещаются в 4Г);
>
> интерес для x86 представляет вариант совпадения максимального размера сегмента с PAE страницей размером 1Г (30 бит максимальный размер сегмента), как раз четыре сегмента помещается и при этом вся физическая память доступна;
>
> потребности эффективного IPC обмена данными для x86 могут потребовать чтобы размер каждого из 32 битных сегментов стал еще

меньше чем 1Г.

Для 32 битных приложений упакованный в плоский указатель индекс сегмента это важное требование, такое размещение индекса позволяет модульным программам обычного типового размера (размера далекого от предельных значений размера сегментов):

- не терять в производительности по сравнению с плоскими моделями памяти;
- при этом предоставлять приложению все выгоды от аппаратной поддержки типизации памяти с помощью сегментов.

Упакованный в плоский указатель индекс сегмента требует особой поддержки процессора, естественный для архитектуры x86 путь включения такой поддержки это атрибуты в сегменте TSS (регистр TR), но мы пока не будем это рассматривать, а вернемся к тому что было для x86 в 1980 году.

3.3
В реальном режиме сегментные регистры вычисляются из индексов автоматически, но в защищенном режиме сегменты описываются уже в дескрипторах. Как предзагружаются сегменты контекста модуля x86 в защищенном режиме?

В защищенном режиме, если сегментные регистры это опять индексы, уже нужны такие блоки и правила процессора, которые смогут обеспечивать индексацию с помощью селекторов, индексацию эквивалентную нужной нам схеме работы:
- загрузить контекст модуля;
- использовать короткий адрес с индексом сегмента.

Это например такие блоки и правила процессора как:
- кэш дескрипторов DC (отдельный специальный кэш, не TLB 386);
- DC ассоциативный либо все сегментные регистры x86 становятся ссылками на записи в массиве DC.

Первые обращения к селекторам как к индексам сегментов загружают кэш дескрипторов DC из дескрипторной таблицы в памяти, после того как контекст модуля загружен, все остальные обращения по тем же самым селекторам будут адресовать данные в этом кэше, индексация сегментов при наличии DC будет работать также как в реальном режиме.

Эти блоки и правила процессора изменяют характер программ защищенного режима по отношению к реальному режиму, программы защищенного режима будут использовать немного иной порядок загрузки селекторов и работы с ними, порядок который учитывает предзагрузку контекста модуля.

286 делался довольно быстро и не все блоки процессора могли быть готовы в первых же моделях процессора, но саму схему работы с

селекторами в защищенном режиме надо было анонсировать сразу, при первом появлении защищенного режима, потому что эта схема работы с селекторами влияет на код программ.

Подготовить сами аппаратные блоки к появлению в более поздних моделях (например к 386) тоже было надо.

3.4
Вопрос "понимали или не понимали во времена реального режима сами разработчики архитектуры x86" (разработчики архитектуры x86 не пишут программы), что для работы модуля нужно сохранять схему:

- загрузить контекст модуля;
- использовать короткие указатели;

остается тайной известной только самим разработчикам архитектуры x86.

В реальном режиме x86 правильный вариант срабатывает автоматически, но возможно это просто совпадение и непреднамеренное выполнение нужной нам задачи (сегменты реального режима были сделаны такими по иным соображениям).

Но к моменту создания защищенного режима 286 об этой последовательности инициализации контекста модуля разработчики архитектуры x86 как минимум забыли (даже если и помнили во времена реального режима).

Индексов сегментов в защищенном режиме x86 нет.

Можно подумать что разработчики x86 были в то время заняты виртуализацией x86, это важной практической задачей, далекой от защищенного режима, но это сказалось бы только на неготовности самих блоков для защищенного режима, а не на схеме работы защищенного режима.

Защищенный режим x86 в то время на самом деле также развивался и даже уже в 32 битном виде, но **отсутствие кэша дескрипторов DC (одновременно с появлением TLB кэша в 386) подсказывает нам то, что разработчик x86 фокусировал свое внимание в то время на плоской модели (база DS == база SS)** и увлекался плоской моделью настолько, что умудрился ограничить до 4Г максимальный размер линейного адреса 32 битного режима в 8 байтном дескрипторе (потребности плоской памяти не указывали на проблемы при таком ограничении линейного адреса в 4Г)

3.5
Сам по себе 8 байтный дескриптор не создает ограничений на размер линейного адреса.

Давайте мы сами, заместо разработчиков x86 образца 1980-х годов, опишем поля базы и предела дескриптора x86 только имеющегося

формата, но опишем исходя из потребностей многосегментных 32 битных программ (а не из потребностей плоских 32 битных программ):

```
сегменты IA16 (биты G=0, D=0)
гранулярность байт для размера и смещения для дескрипторов
     предел:    16 бит IA16 /16! бит IA32 (IA32 есть проблемы если больше 16 бит)
     база:      24 бит IA16 /32! бит IA32 (IA32 нет проблем если больше 24 бит)
```

Потребности байтовых размеров сегментов для IA32 достаточно реализованы в IA16 (G=0, D=0), если мало 16 бит предела, то и 20 бит при фиксированном выравнивании базы также может не хватать.

Для сегментов больших размеров гранулярность 4К подходит также хорошо, как гранулярность 4К подходит для страниц.

> Сегмент это не такой тонкий типизирующий инструмент, чтобы в добавок к описанию логической области памяти обязательно иметь точность до байта (дескрипторы служат не для того чтобы адресовать отдельные байты). Когда приходится выбирать, то большой размер сегмента полезнее чем байтовая гранулярность.
>
> Допустимо что большой сегмент будет иметь выравнивание базы, а на границах в сегменте будут резервироваться небольшие системные области памяти, безопасные для чтения и записи через этот сегмент, это все не повлияет на статические смещения указателей на данные пользователя в этом сегменте.

```
сегменты IA32 (биты G=1, D=1)
гранулярность 4K для размера и смещения для 32 битных дескрипторов
     предел:    32 бит IA32 (есть проблемы если больше 32 бит)
     база:      44 бит IA32 (нет проблем если больше 32 бит)
                могут быть варианты выделения битов базы под атрибуты,
                но не желательно иметь базу менее 40 бит
```

Бит G просто не нужен, если в 32 битном режиме (D=1) есть гранулярность базы равная 4К.

Теоретически может быть вариант IA32 (G=0, D=1), когда байтовый размер сегмента ограничен 20 битами и байтовый размер базы ограничен 32 битами, но неизвестны практические задачи когда важна байтовая гранулярность сегмента 20 битного размера в пределах 4Г (даже для реального режима гранулярность 16 байт), зато полно задач когда 32 битному приложению нужна большая виртуальная память в 44 бит, и есть примеры когда в дескрипторе не хватает бит для атрибутов.

Можно понять, что есть сложности при создании блоков процессора работающих с многобитным линейным адресом (особенно когда физической памяти всего 1 мегабайт, а системная шина 16 бит), но даже самый стартовый вариант 32 битного процессора требует реализовать хотя бы 34 бита линейного адреса, чтобы в виртуальной памяти можно было бы разместить и адресовать хотя бы 4 полноразмерных (по 4Г) 32 битных сегмента из одного TSS.

===
Конец текста