архитектура x86-64 под скальпелем ассемблерщика

крис касперски ака мыщъх no-email

32-битная эпоха уходит в прошлое, сдаваясь под натиском новых идей и платформ. оба флагмана рынка представили 64-битные архитектуры, открывающие дверь в мир больших скоростей и производительных ЦП. это настоящий прорыв — новые регистры, новые режимы работы… попробуем с ним разобраться? в этой статье мы рассмотрим архитектуру AMD64 (она же x86-64) и покажем как с ней бороться.

введение

64-битный лейбл звучит возбуждающе, но в практическом плане это всего лишь хитрый маркетинговый трюк, скрывающий не только достоинства, но и недостатки. Нам дарованы 64-битные операнды и 64-битная адресация. Казалось бы, лишние разряды карман не тянут и если не пригодятся, то по крайней мере не помешают. Так ведь нет! С ростом разрядности увеличивается и длина машинных команд, а, значит, время их загрузки/декодирования и размеры программы, поэтому для достижения ***не худшей*** производительности 64-битный процессор должен иметь более быструю память и более емкий кэш. Это раз.

Рисунок 1 64-разрядный лейбл, для разнообразия на китайском

64-битные целочисленные операнды становятся юзабельны только при обработке чисел порядка 2^{33+} (8.589.934.592) и выше. Там, где 32-битному процессору требуется несколько тактов, 64-битный справляется за один. Но где вы видели такие числа в домашних и офисных приложениях? Не зря же инженеры из Intel пошли на сокращение разрядности АЛУ (арифметичного-логичесокго устройства), "ширина" которого в Pentium-4 составляет всего 16 бит, против 32-бит в Pentium-III. Это не значит, что Pentium-4 не может обрабатывать 32-разрядные числа. Может. Только он тратит на них больше времени, чем Pentium-III. Но, поскольку, процент подлинно 32-разрядных чисел (т. е. таких, что используют свыше 16 бит) в домашних приложениях относительно невысок, производительность падает незначительно. Зато ядро содержит меньше транзисторов, выделяет меньше тепла и лучше работает на повышенной тактовой частоте, т. е. в целом эффект положительный.

64-битная разрядность… Помилуйте! Адресовать 18.446.744.073.709.551.616 байт памяти не нужно даже Microsoft'у со вмести графическими заворотами! Из 4 Гбайт адресного пространства Windows Processional и Windows Server только 2 Гбайта выделяют приложениям. 3 Гбайта выделяет лишь Windows Advanced Server, и не потому, что больше выделить невозможно! x86 процессоры с легкостью адресуют вплоть до 16 Гбайт (по 4 Гбайта на код, данные, стек и кучу), опять-таки обходясь минимальной перестройкой операционной системы! Почему же до сих пор это не было сделано? Почему мы как лохи сидим на "жалких" 4 Гбайтах из которых реально доступны только два?! Да потому, что больше никому не нужно! Систему, адресующую 16 Гбайт, просто так не продашь, кого эти гигабайты интересуют? Вот "64-бита" совсем другое дело! Это освежает! Вот все вокруг них и танцуют.

Сравнивать 32- и 64-битные процессоры бессмысленно! Если 64-битный процессор на "домашнем" приложении оказывается быстрее, то отнюдь не за счет своей 64-битности, а благодаря совершенно независимым от нее конструктивным ухищрениям, на которых инженеры едва не разорвали себе задницы!

Впрочем, не будем о грустном. 64-бита все равно войдут в нашу жизнь. Для некоторых задач они очень даже ничего. Вот, например, криптография. 64-бита это же 8 байт! 8-символьные пароли можно полностью уместить в один регистр, не обращаясь к памяти, что дает невероятный результат! Скорость перебора увеличивается чуть ли не на порядок! Ну так чего же мы ждем?! Вперед! На штурм 64-битных вершин!

Рисунок 2 AMD Athlon 64 во всей своей красе

что нам понадобиться?

Для программирования в 64-режтме желательно иметь компьютер с процессором AMD Athlon FX или Opertorn, но на худой конец можно обойтись и эмулятором. Существует не так уж много эмуляторов под x86-64 платформу и все они недоделанные и жутко багистные, но для знакомства с AMD 64 их будет вполне достаточно, а дальше пусть каждый решает сам — нужна ли ему 64-битность или ну ее на фиг!

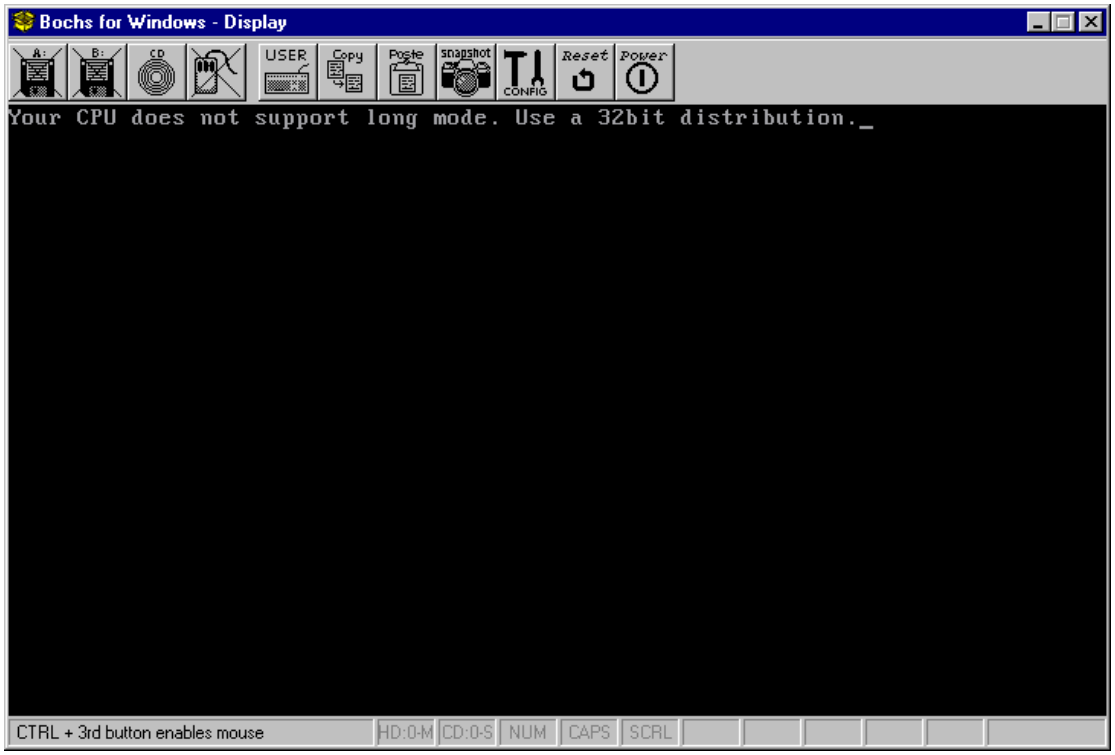

Рисунок 3 реакция 64-битного Линуха, запущенного под стандартной сборкой BOCHS'a ("your CPU does not support long mode. use a 32bit distribution")

Большой популярностью пользуется бесплатный эмулятор BOCHS (в просторечии называемый "борщом"), распространяемый в исходных текстах. Поддержка архитектуры x86-64 впервые появилась в версии 2.2-pre3 и затем была включена в релиз 2.2.1 на правах экспериментальной фичи. На официальном сайте (http://bochs.sourceforge.net/) можно найти несколько готовых бинарных сборок под разные платформы, но… только для архитектуры x86. Эмуляция x86-64 требует обязательной перекомпиляции под UNIX'ом. Скачиваем исходные тексты (http://prdownloads.sourceforge.net/bochs/bochs-2.2.1.tar.gz?download), распаковываем архив, запускам конфигуратор с ключом **`--enable-x86-64`** и затем даем make.

```
$./configure --enable-x86-64
$make
```

Листинг 1 сборка BOCHS'a с поддержкой эмуляции x86-64

Образуется исполняемый файл bochs, требующий для своей работы bios'a и bxrc-сценария, которые можно позаимствовать из готовой бинарной сборки. Для компиляции под Windows-платформу следует запустить скрипт "conf.win32-vcpp", а затем выполнить "`make win32_snap`". Для этого, естественно, необходимо иметь Линух, поскольку Windows shell-скриптов с упор не понимает (правда, можно воспользоваться Cygwin'ом, но сборка с ним имеет проблемы):

```
sh .conf.win32-vcpp
make win32_snap
```

Листинг 2 сборка BOCHS'a для компиляции Microsoft Visual C++

Сборка компилятором Microsoft Visual C++ 6.0 проходит не очень гладко (точнее не проходит совсем) и приходится устранять многочисленные ошибки, допущенные разработчиками эмулятора, что требует времени и квалификации, но… как говориться, зачем рвать задницу, когда она уже давно разорвана до нас? В сети можно найти множество сборок борща, например: http://www.psyon.org/bochs-win32/bochs-x86-64-20050508.exe.

Тем не менее, со своей работой борщ справляется из рук вон плохо и к тому же сильно тормозит (мой Pentium-III 733 затормаживается до < 1 Мгц AMD 64, отставая даже от Машины Бэббиджа, собранной на шестеренках и приводимой в движение паровым двигателем). Многие 64-битные Линухи "вылетают" еще на стадии загрузки ядра. Побаловаться x86-64 режимом под

борщом еще можно, но на рабочий инструмент он не тянет. Впрочем, в последующих версиях ошибки эмуляции скорее всего будут исправлены, и тогда единственным недостатком останется низкая скорость, а вот это уже фундаментально. Обладателям low-end процессоров по любому приходится искать что-то еще.

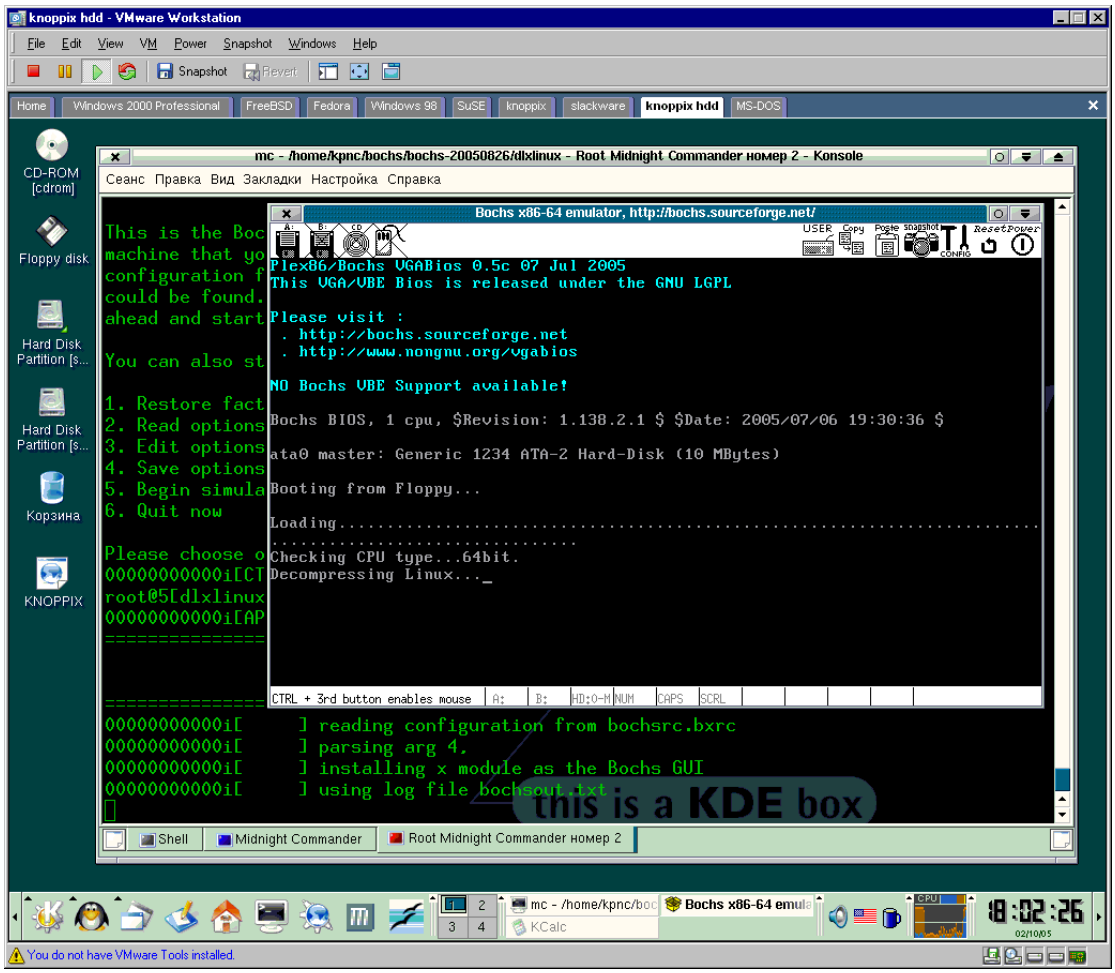

Рисунок 4 специальная сборка BOCHS'а успешно переходит в x86-64 режим, уверенно чувствуя себя под виртуальной машиной VM Ware, так что это уже двойная эмуляция получается!

QEMU – бесплатный динамический эмулятор, основанный на BOCHS. Архитектура x86-64 эмулируются на Pentium-III с ничуть не худшей скоростью, чем x86 под коммерческим VM Ware. Стабильность работы так же выше всяких похвал. На официальном сайте (http://fabrice.bellard.free.fr/qemu/) выложены исходные тексты и готовые сборки под Линух. Обладателям Windows приходится заниматься компиляцией самостоятельно или рыскать в поисках добычи по сети. Добросовестный билд лежит на хорошем японском сервере http://www.h7.dion.ne.jp/~qemu-win/. Там же можно найти драйвер-акселератор, ускоряющий эмуляцию в несколько раз. Кстати говоря, помимо x86-64, QEMU эмулирует x86, SPARC, PowrPC и некоторые другие архитектуры. И еще. QEMU это единственный эмулятор, в котором виртуальная сеть встает сама без плясок с бубном и не загаживает основную операционную систему левыми адаптерами.

Рисунок 5 загрузка 64-разрядного Дебиана под эмулятором QEMU

Так же нам потребуется 64-разрядная операционная система. Дотянутся до 64-битных регистров и прочих "вкусностей" x86-64 архитекторы можно только из специального 64-разрядного режима (он же "long mode"). Ни под реальном, ни под 32-разрядным защищенным x86-режимом они не доступны. И хотя мы покажем как перевести процессор из реального в 64-разрядный режим, создание полнофункциональной операционной системы не входит в наши планы, а без нее никуда!

Проще всего, конечно, взять Windows XP 64-Bit Edition, но… не все хакеры разделяют этот путь (правильную вещь буквой X не назовут). Если выпрямить земную ось, то поднимется такой цунами, что всю Америку вместе с Редмондом смоет на хрен в океан. А Линух делают и в континентальной Европе, до которой никакие цунами не достанут! (Правда, ей угрожает ледник и первой пострадают небезразличные для Линуха скандинавские страны). Большинство производителей либо уже выпустили x86-64 порты, либо собираются это сделать в ближайшем будущем. Приверженцам традиционного немецкого качества можно порекомендовать **SuSE LiveCD 9.2**, не требующий установки (http://suse.osuosl.org/suse/x86_64/live-cd-9.2/SUSE-Linux-9.2-LiveCD-64bit.iso), но лично я больше предпочитаю Дебиан, неофициальный порт которого в формате businesscard-CD лежит на http://cdimage.debian.org/cdimage/unofficial/sarge-amd64/iso-cd/debian-31r0a-amd64-businesscard.iso. Там же можно найти и другие порты.

Рисунок 6 64-битная версия Windows в стадии начальной загрузки

Теперь перейдем к подготовке инструментария. Как минимум нам понадобится ассемблер и отладчик. Мы будем использовать **FASM** (http://flatassembler.net/). Он бесплатен, работает под LINUX/Windows/MS-DOS, поддерживает x86-64, обладает удобным синтаксисом и т. д. Любители классической миссионерской могут качнуть бесплатный Windows Server 2003 SP1 Platform SDK (http://www.microsoft.com/downloads/details.aspx?FamilyId=A55B6B43-E24F-4EA3-A93E-40C0EC4F68E5), в состав которого входит 64-разрядный **MASM**. Синтаксически оба этих ассемблера несовместимы, так что попеременно пользоваться ими не удастся, и нужно сразу выбирать какой-то один.

Практически во все x86-64 порты Линуха входит **GNU Debugger**, которого для наших задач вполне достаточно. Обладатели Windows могут воспользоваться **Microsoft Debugger**'ом, входящим в состав бесплатного Microsoft Debugging Tools (http://www.microsoft.com/whdc/devtools/debugging/installamdbeta.mspx).

Для погружения в режим 24-часового хачинья потребуется энергичная музыка с плоским спектром (это когда все инструменты сразу). Рекомендую последние альбомы Penumbra, особенно "The Last Bewitchment". Это что-то невероятное! Нежный женский вокал в компании колоритного мужского рыка с резкими переходами от плавных партий к ястребиному крику сносит башню окончательно и бесповоротно. А тексты! Возникает странное ощущение сопричастности, словно кто-то иной, на другом краю земли чувствует и мыслит так же как ты…

обзор x86-64

За подробным описанием x86-64 архитектуры лучше всего обратится к фирменной документации "AMD64 Technology — AMD64 Architecture Programmer's Manual Volume 1:Application Programming" (http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf). Мы же ограничимся только беглым обзором основных нововведений.

Наконец-то AMD сжалилась над нами и подарила программистам то, что все так долго ждали. К семи регистрам общего назначения (восьми — с учетом ESP) добавилось еще восемь, в результате чего их общее количество достигло 15 (16)!

Старые регистры, расширенные до 64-бит, получили имена RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, RIP и RFLAGS. Новые регистры остались безымянными и просто пронумерованы от R8 до R15. Для обращения к младшим 8-, 16- и 32-битам новых регистров можно использовать суффиксы b, w и d соответственно. Например, R9 – это 64-разряный регистр, R9b – его младший байт (по аналогии с AL), а R9w – младшее слово (тоже самое, что AX в EAX). Прямых наследников AH к сожалению не наблюдается и для манипуляции со средней частью регистров приходится извращаться со сдвигами и математическими операциями. Абыдно, конечно, но ничего не поделаешь!

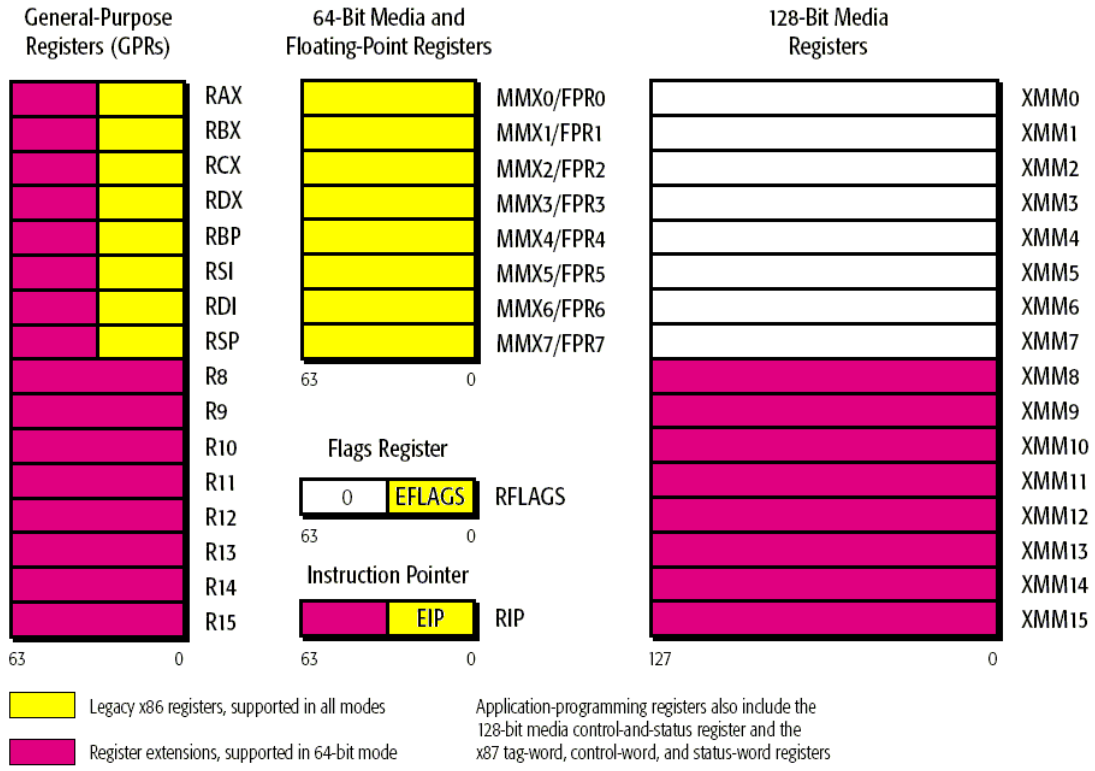

Рисунок 7 регистры, доступные в x86-64 режиме

Регистр указатель команд RIP теперь адресуется точно так же, как и все остальные регистры общего назначения. Программисты, заставшие живую PDP-11 (или ее отечественный клон "Электроники БК" или "УКНЦ"), только презрительно хмыкнут — наконец-то до разработчиков "писюка" стали доходить очевидные истины, которые на всех нормальных платформах были реализованы еще черт знает когда (в эпоху меча и топора).

Возьмем простейший пример: загрузить в регистр AL опкод следующей машинной команды. На x86 приходится поступать так.

```
call $ + 5      ; запихнуть в стек адрес след. команды и передать на нее управление
pop ebx         ; вытолкнуть из стека адрес возврата
add ebx, 6      ; скорректировать адрес на размер команд pop/add/mov
mov al, [ebx]   ; теперь AL содержит опкод команды NOP
NOP             ; команда, чем опкод мы хотим загрузить в AL
```

Листинг 3 загрузка опкода следующей машинной команды в классическом x86

Это же умом поехать можно пока все это писать! И еще здесь очень легко ошибиться в размере команд, которых приходится либо вычислять вручную, либо загромождать листинг никому не нужными метками, к тому же неизбежно затрагивается стек, что в ряде случаев нежелательно или недопустимо (особенно в защитных механизмах, нашпигованных антиотладочными приемами).

А теперь перепершем тот же самый пример на x86-64:

```
mov al,[rip]   ; загружаем опкод следующей машинной команды
NOP            ; команда, чем опкод мы хотим загрузить в AL
```

Листинг 4 загрузка опкода следующей машинной команды на x86-64

Крррасота! Только следует помнить, что RIP всегда указывает на следующую, а отнюдь не текущую инструкцию! К сожалению, ни Jx RIP, ни CALL RIP не работают. Таких команд в лексиконе x86-64 просто нет. Но это еще что! Исчезла абсолютная адресация, а это гораздо хужее. Если нам надо изменить содержимое ячейки памяти по конкретному адресу, на x86 мы поступаем приблизительно так:

```
dec byte ptr [666h]    ; уменьшить содержимое байта по адресу 666h на единицу
```

Листинг 5 абсолютная адресация в классическом x86

Под x86-64 транслятор выдает ошибку ассемблирования, вынуждая нас прибегать к фиктивному базированию:

```
xor r9, r9             ; обнулить регистр r9
dec byte ptr [r9+666h] ; уменьшить содержимое байта по адресу 0+666h на единицу
```

Листинг 6 использование фиктивного базирования на x86-64 для абсолютной адресации

Есть и другие отличия от x86, но они не столь принципиальны. Важно то, что в режиме совместимости с x86 (Legacy Mode) ни 64-битные регистры, ни новые методы адресации недоступны! Никакими средствами (включая черную и белую магию) дотянуться до них нельзя и прежде чем что-то сделать, необходимо перевести процессор в "длинный" режим (long mode), который делиться на два под-режима: режим совместимости с x86 (compatibility mode) и 64-битный режим (64-bit mode). Режим совместимости предусмотрен только для того, чтобы 64-разрядная операционная система могла выполнять старые 32-битные приложения. Никакие 64-битные регистры здесь и не ночевали, поэтому нам этот режим фиолетов как заяц.

Реальная 64-битность обитает только в 64-bit long mode, о котором мы и будем говорить!

<table>
<tr><th colspan="2" rowspan="2">Operating Mode</th><th rowspan="2">Operating System Required</th><th rowspan="2">Application Recompile Required</th><th colspan="2">Defaults</th><th rowspan="2">Register Extensions</th><th rowspan="2">Typical GPR Width (bits)</th></tr>
<tr><th>Address Size (bits)</th><th>Operand Size (bits)</th></tr>
<tr><td rowspan="3">Long Mode</td><td>64-Bit Mode</td><td rowspan="3">New 64-bit OS</td><td>yes</td><td>64</td><td rowspan="2">32</td><td>yes</td><td>64</td></tr>
<tr><td rowspan="2">Compatibility Mode</td><td rowspan="2">no</td><td>32</td><td rowspan="2">no</td><td>32</td></tr>
<tr><td>16</td><td>16</td><td>16</td></tr>
<tr><td rowspan="4">Legacy Mode</td><td rowspan="2">Protected Mode</td><td rowspan="3">Legacy 32-bit OS</td><td rowspan="4">no</td><td>32</td><td>32</td><td rowspan="4">no</td><td rowspan="2">32</td></tr>
<tr><td>16</td><td>16</td></tr>
<tr><td>Virtual-8086 Mode</td><td rowspan="2">16</td><td rowspan="2">16</td><td rowspan="2">16</td></tr>
<tr><td>Real Mode</td><td>Legacy 16-bit OS</td></tr>
</table>

Таблица 1 режимы работы процессора AMD-64 и их особенности

>>> врезка переход в 64-разрдяный режим

В исходниках FreeBSD можно найти файл amd64_tramp.S, быстро и грязно переводящий процессор в 64-режим. Откомпилировав, его можно записать в boot-сектор, загружающий нашу собственную операционную систему (вы ведь пишите ее, правда?) или слинковать com-файл, запускаемый из реального x86-режима (для этого потребуется чистая MS-DOS безо всяких экстендеров). В общем, вариантов много…

```
//$FreeBSD: /repoman/r/ncvs/src/sys/boot/i386/libi386/amd64_tramp.S,v 1.4 2004/05/14

/*
 * Quick and dirty trampoline to get into 64 bit (long) mode and running
 * with paging enabled so that we enter the kernel at its linked address.
 */
#define MSR_EFER      0xc0000080
#define EFER_LME      0x00000100
#define CR4_PAE       0x00000020
#define CR4_PSE       0x00000010
#define CR0_PG        0x80000000

/* GRRR. Deal with BTX that links us for a non-zero location */
#define VPBASE 0xa000
#define VTOP(x)       ((x) + VPBASE)

        .data

        .p2align 12,0x40

        .globl  PT4
PT4:
        .space  0x1000
        .globl  PT3
PT3:
        .space  0x1000
        .globl  PT2
PT2:
        .space  0x1000

gdtdesc:
        .word   gdtend - gdt
        .long   VTOP(gdt)             # low
        .long   0                     # high

gdt:
        .long   0                     # null descriptor
        .long   0
        .long   0x00000000            # %cs
        .long   0x00209800
        .long   0x00000000            # %ds
        .long   0x00008000
gdtend:

        .text
        .code32

        .globl  amd64_tramp
amd64_tramp:
        /* Be sure that interrupts are disabled */
        cli

        /* Turn on EFER.LME */
        movl    $MSR_EFER, %ecx
        rdmsr
        orl     $EFER_LME, %eax
        wrmsr

        /* Turn on PAE */
        movl    %cr4, %eax
        orl     $(CR4_PAE | CR4_PSE), %eax
        movl    %eax, %cr4

        /* Set %cr3 for PT4 */
        movl    $VTOP(PT4), %eax
        movl    %eax, %cr3

        /* Turn on paging (implicitly sets EFER.LMA) */
        movl    %cr0, %eax
        orl     $CR0_PG, %eax
        movl    %eax, %cr0

        /* Now we're in compatability mode. set %cs for long mode */
        movl    $VTOP(gdtdesc), %eax
        movl    VTOP(entry_hi), %esi
        movl    VTOP(entry_lo), %edi
        lgdt    (%eax)
```

```
        ljmp    $0x8, $VTOP(longmode)

        .code64
longmode:
        /* We're still running V=P, jump to entry point */
        movl    %esi, %eax
        salq    $32, %rax
        orq     %rdi, %rax
        pushq   %rax
        ret
```

Листинг 7 перевод процессора в 64-разрядный режим

hello world на x86-64

Программирование под 64-битную версию Windows мало чем отличается от традиционного, только все операнды и адреса по умолчанию 64-разярные, а параметры API-функций передаются через регистры, а не через стек. Первые четыре аргумента всех API-функций передаются в регистрах RCX, RDX, R8 и R9 (регистры перечислены в порядке следования аргументов, крайний левый аргумент помещается в RCX). Остальные параметры кладутся на стек. Все это называется **x86-64 fast calling conversion** (соглашение о быстрой передаче параметров для x86-64), подробное описание которой можно найти в статье "The history of calling conventions, part 5 amd64" (http://blogs.msdn.com/oldnewthing/archive/2004/01/14/58579.aspx). Так же нелишне заглянуть на страничку бесплатного компилятора Free PASCAL и поднять документацию по способам вызова API: http://www.freepascal.org/wiki/index.php/Win64/AMD64_API.

В частности, вызов функции с пятью аргументами API_func(1,2,3,4,5) выглядит так:

```
        mov     dword ptr [rsp+20h], 5 ; кладем на стек пятый слева аргумент
        mov     r9d, 4                 ; передаем четвертый слева аргумент
        mov     r8d, 3                 ; передаем третий слева аргумент
        mov     edx, 2                 ; передаем второй слева аргумент
        mov     ecx, 1                 ; передаем первый слева аргумент
        call    API_func
```

Листинг 8 пример вызова API-функции с пятью параметрами по соглашению x86-64

Смещение пятого аргумента относительно верхушки стека требует пояснений. Почему оно равно 20h? Ведь адрес возврата занимает только 8 байт. Какая су… сущность съела все остальные? Оказывается, они "резервируются" для первых четырех аргументов, переданных через регистры. "Зарезервированные" ячейки содержат неинициализированный мусор и по-буржуйски называются "spill", что переводится как "затычка" или "потеря".

Вот минимум знаний, необходимых для выживания в мире 64-битной Windows при программировании на ассемблере. Остается разобрать самую малость. Как эти самые 64-бита заполучить?! Для перевода FASM'a в x86-64 режим достаточно указать директиву "use64" и дальше шпрехать как обычно.

Ниже идет пример простейшей x86-64 программы, которая не делает ничего, только возвращает в регистре RAX значение "ноль".

```
; сообщаем FASM'у, что мы хотим программировать на x86-64
use64

xor r9,r9               ; обнуляем регистр r9
mov rax,r9              ; пересылаем в rax,r9 (можно сразу mov rax,0, но неинтересно)
ret                     ; выходим туда откуда пришли
```

Листинг 9 простейшая 64-битная программа

Никаких дополнительных аргументов командной строки указывать не надо, просто сказать "`fasm file-name.asm`" и все! Через несколько секунд образуется файл file-name.bin, который в hex-представлении выглядит следующим образом:

```
4D 31 C9        xor     r9, r9
4C 89 C8        mov     rax, r9
C3              retn
```

Листинг 10 дизассемблерный листинг простейшей 64-битной программы

Формально, это типичный com-файл, вот только запустить его не удастся (во всяком случае, ни одна популярная ось его не "съест") и необходимо замутить законченный ELF или PE, в заголовке которого будет явно прописана нужна разрядность.

Начиная с версии 1.64 ассемблер FASM поддерживает специальную директиву "`format PE64`", автоматически формирующую 64-разрядный PE-файл (директиву "use64" в этом случае указывать уже не нужно), а в каталоге EXAMPLES можно найти готовый пример PE64DEMO в котором показано как ее использовать на практике.

Ниже приведен пример x86-64 программы "hello, world" с комментариями:

```
; пример 64-битного PE файла
; для его выполнения необходимо иметь Windows XP 64-bit edition

; указываем формат
format PE64 GUI

; указываем точку входа
entry start

; создать кодовую секцию с атрибутами на чтение и исполнение
section '.code' code readable executable
start:
        mov     r9d,0                   ; uType == MB_OK (кнопка по умолчанию)
                                        ; аргументы по соглашению x86-64
                                        ; передаются через регистры, не через стек!
                                        ; префикс d задает регистр размером в слово,
                                        ; можно использовать и mov r9,0, но тогда
                                        ; машинный код будет на байт длиннее

        lea     r8,[_caption]           ; lpCaption передаем смещение
                                        ; команда lea занимает всего 7 байт,
                                        ; а mov reg, offset - целых 11, так что
                                        ; lea намного более предпочтительна

        lea     rdx,[_message]          ; lpText передаем смещение выводимой строки

        mov     rcx,0                   ; hWnd передам дескриптор окна-владельца
                                        ; (можно так же использовать xor rcx,rcx
                                        ;  что на три байта короче)

        call    [MessageBox]            ; вызываем функцию MessageBox

        mov     ecx,eax                 ; заносим в ecx результат возврата
                                        ; (Функция ExitProcess ожидает 32-битный параметр
                                        ; можно использовать и mov rcx,rax, но это будет
                                        ; на байт длиннее)

        call    [ExitProcess]           ; вызываем функцию ExitProcess

; создать секцию данных с атрибутами на чтение и запись
; (вообще-то в данном случае атрибут на запись необязателен,
; поскольку мы ничего не пишем, а только читаем)
section '.data' data readable writeable

  _caption db 'PENUMBRA is awesome!',0       ; ASCIIZ-строка заголовка окна
  _message db 'Hello World!',0               ; ASCIIZ-строка выводимая на экран

; создать секцию импорта с атрибутами на чтение и запись
; (здесь атрибут на запись обязателен, поскольку при загрузке PE-Файла
; в секцию импорта ; будут записываться фактические адреса API-функций)
section '.idata' import data readable writeable

        dd 0,0,0,RVA kernel_name,RVA kernel_table
        dd 0,0,0,RVA user_name,RVA user_table
        dd 0,0,0,0,0            ; завершаем список двумя 64-разряными нулеми!!!

kernel_table:
        ExitProcess dq RVA _ExitProcess
        dq 0                    ; завершаем список 64-разряным нулем!!!

user_table:
        MessageBox dq RVA _MessageBoxA
        dq 0

kernel_name db 'KERNEL32.DLL',0
```

```
user_name db 'USER32.DLL',0

_ExitProcess dw 0
        db 'ExitProcess',0
_MessageBoxA dw 0
        db 'MessageBoxA',0
```

Листинг 11 64-битное приложение "hello, world" под Windows на FASM'e

Рисунок 8 64-битный файл — первый полет

Ассемблируем файл (fasm PE64DEMO.ASM) и запустим образовавшийся EXE на выполнение. Под 32-разрядной Windows он, естественно, не запустится и она скажет "мяу":

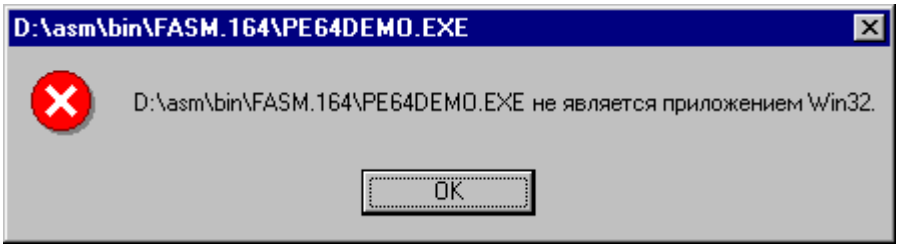

Рисунок 9 реакция 32-битной Windows на попытку запуска 64-битного PE-файла

Вдоволь наигравшись нашем первым x86-64 файлом, загрузим его в дизассемблер (например, в IDA Pro 4.7. Она хоть и материться, предлагая использовать специальную 64-битную версию, но при нажатии на "yes" все конкретно дизассемблирует, во всяком случае до тех пор пока не столкнется с подлинным 64-битным адресом или операндом, с которым произойдет обрезание, в частности mov r9,1234567890h дизассемблируется как mov r9, 34567890h, так что переход на 64-битную версию IDA все же очень желателен, тем более, что начиная с IDA 4.9 она входит в базовую поставку). Посмотрим, что у него внутри?

А внутри у него вот что:

```
.code:0000000000401000 41 B9 00 00 00 00     mov     r9d, 0
.code:0000000000401006 4C 8D 05 F3 0F 00 00  lea     r8, aPENUMBRA
.code:000000000040100D 48 8D 15 03 10 00 00  lea     rdx, aHelloWorld ; "Hello World!"
.code:0000000000401014 48 C7 C1 00 00 00 00  mov     rcx, 0
.code:000000000040101B FF 15 2B 20 00 00     call    cs:MessageBoxA
.code:0000000000401021 89 C1                 mov     ecx, eax
.code:0000000000401023 FF 15 13 20 00 00     call    cs:ExitProcess
```

Листинг 12 дизассемблерный листинг 64-битного приложения "hello, world!"

Что ж… довольно громоздко, объемно и концептуально. Для сравнения, дизассемблированный листинг аналогичного 32-разрядного файла приведен ниже. Старый x86 код в 1,6 раз короче! А ведь это только демонстрационная программа из нескольких строк! На полновесных приложениях разрыв будет только нарастать! Так что не стоит злоупотреблять 64-разрядным кодом без необходимости. Его следует использовать только там, где 64-битная арифметика и 8 дополнительных регистров действительно дают ощутимый выигрыш. Например, в математических задачах или программах для вскрытия паролей.

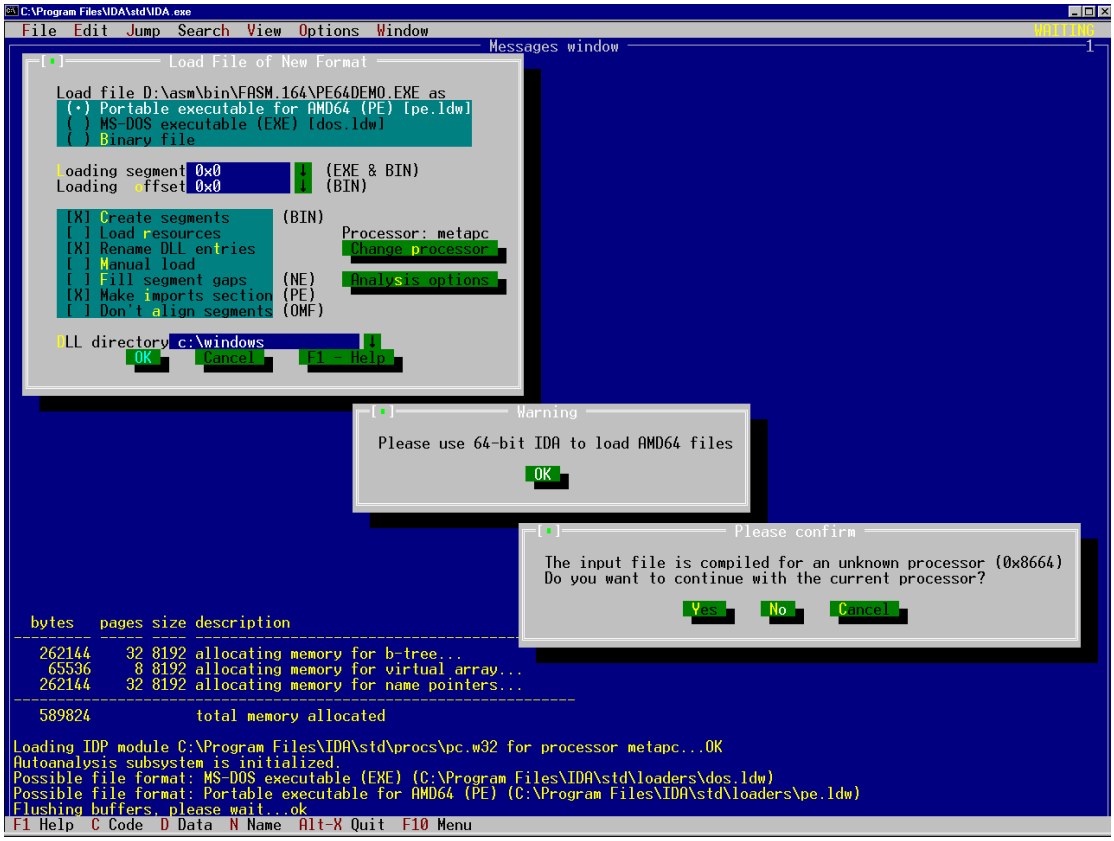

Рисунок 10 дизассемблирование 64-битного PE-файла 32-битной версий IDA Pro

```
code:00401000 6A 00                          push    0
code:00401002 68 00 20 40 00                 push    offset aPENUMBRA
code:00401007 68 17 20 40 00                 push    offset aHelloWorld
code:0040100C 6A 00                          push    0
code:0040100E FF 15 44 30 40 00              call    ds:MessageBoxA
code:00401014 6A 00                          push    0
code:00401016 FF 15 3C 30 40 00              call    ds:ExitProcess
```

Листинг 13 дизассемблерный листинг 32-битного приложения "hello, world!"

В качестве заключительно упражнения перепишем наше приложение в стиле MASM, поклонников которого нужно не бить, а уважать (как ни крути, а все-таки патриарх). Никаких радикальных отличий не наблюдается:

```
; объявляем внешние API-функции, которые мы будем вызывать
extrn MessageBoxA: PROC
extrn ExitProcess: PROC

; секция данных с атрибутами по умолчанию (чтение и запись)
.data
mytit db 'PENUMBRA is awesome!', 0
mymsg db 'Hello World!', 0

; секция кода с атрибутами по умолчанию (чтение и исполнение)
.code
Main:
mov r9d, 0              ; uType = MB_OK
lea r8,  mytit          ; LPCSTR lpCaption
lea rdx, mymsg          ; LPCSTR lpText
mov rcx, 0              ; hWnd = HWND_DESKTOP
call MessageBoxA
mov ecx, eax            ; uExitCode = MessageBox(...)
call ExitProcess
End Main
```

Листинг 14 64-битное приложение "hello, world" под Windows на MASM'e

Ассемблирование и линковка проходит так: "`ml64 XXX.asm /link /subsystem:windows /defaultlib:kernel32.lib /defaultlib:user32.lib /entry:main`" в результате чего образуется

готовый к употреблению exe-файл с румяной поджаренной корочкой нашего ЦП (FASM ассемблирует намного быстрее).

Примеры более сложных программ легко найти в сети. Как показывает практика, запросы типа "x86-64 [AMD64] assembler example" катастрофически неэффективны и гораздо лучше использовать "mov rax" (без кавычек) или вроде того.

заключение

Вот мы и познакомились с архитектурой x86-64! Здесь действительно есть место где развернутся и чему поучиться! Насколько эти знания окажутся востребованы на практике — так сразу и не скажешь. У AMD есть хорошие шансы пошатнуть рынок, но ведь и Intel не дремлет, активно продвигая собственные 64-разрядные платформы, известные под общем именем IA64, но о них как ни будь в другой раз…