# оптимизация наоборот
# или как затормозить компьютер с помощью SSE

крис касперски ака мыщъх, no-email

**многие программы поддерживают режимы оптимизации под специальные ЦП (MMX, MMXext, SSE, SSE2, SSE3, SSE4, 3DNow! 3DNowExt, и т. д.), однако, попытка форсировать компиляцию под SSE4, круче которого ничего нет, часто заканчивается чуть ли не катастрофой — от полного нежелания запускаться до падения производительности в десятки раз. почему же так происходит?!**

## *введение*

Программы под UNIX как правило распространяются в исходных текстах и предусматривают возможность компиляции под различные ЦП, дающие в лапы пользователя кучу опций, в которых не так-то просто разобраться, но от которых зависит быстродействие программы, причем зависит, прямо так скажем, весьма радикально, так что разбираться по любому все равно придется.

## *архитектура кремниевых сооружений*

Учет архитектурных особенностей конкретных ЦП теоретически способен дать огромный выигрыш, но практически языки высокого уровня абстрагируют программиста от деталей конкретной реализации, перекладывая все заботы на плечи компилятора, но что может сделать компилятор?! Переупорядочить инструкции, выровнять структуры данных по кратным адресам, избавиться от ветвлений, заменить медленные команды (например, инструкцию целочисленного деления DIV, так же отвечающую за взятие остатка) их более быстрыми аналогами и т. д. Во времена господства Intel 80486, Intel Pentium-I/II, AMD K5/K6, когда архитектура и правила оптимизации под каждую модель процессора существенно отличались, оптимизирующие компиляторы давали колоссальный выигрыш, временами увеличивающий производительность в несколько раз.

Но, начиная с Pentium Pro, процессоры научились оптимизировать код самостоятельно, разбивая поток машинных команд на микроинструкции, распределяемые по функциональным устройствами (типа АЛУ или блока вещественной арифметики) и выполняя их с максимальной эффективностью. Сейчас, в начале XXI века, производительность в основном определяется "крутостью" оптимизатора и, естественно, опциями компилятора, отвечающими за глубину разворота циклов, агрессивность встраивания функций, удаление "хвостовой" рекурсии и т. д.

Многообразие ключей оптимизации затрудняет работу с компилятором и потому разработчикам последних пришлось заложить в них специальные шаблоны — программист просто указывает тип целевого процессора и компилятор автоматически выставляет оптимальные (с его точки зрения!) параметры оптимизации по умолчанию. В частности, выбор Pentium-4 обычно ведет к крайне агрессивному развороту циклов и встраиванию функций, что приводит к неоправданному "разбуханию" кода и как следствие — падению производительности (особенно если интенсивно выполняемые циклы "вылетают" за пределы кэш-памяти первого уровня).

Экспериментируя с различными ключами оптимизации на своем Prescott'e, работающим под Linux-ядром версии 2.4.27, мыщъх пришел к выводу, что большинство программ, компилируемых GCC 3.4.3 показывают значительно лучший результат при выборе "обобщенной унипроцессорной" архитектуры i686 ('`-march=i686 -mtune=prescott`'), чем при "`march=prescott`", уступающим в производительности…. даже "`march=i386`". Ключи "`march=pentium3`" и "`march=pentium4`" не обнаружили (на Prescott'e!) никакой заметной невооруженному глазу разницы (правда, под "pentium4" компиляция иногда проваливается). Под другими процессорами наблюдалась весьма схожая картина.

Сначала мыщъх списывал этот эффект на глюк данной версии GCC и кривизну своих лап умноженную на градиент упругости хвоста, но поиск по форумам показал, что глюк носит характер призрака, блуждающего по всем континентам и оставляющим следы не только на форумах, но и в солидных исследовательских статьях наподобие "Intel Hyper-Threading on Linux: Fact or Myth" (Intel Hyper-Threading на Linyx: факт или миф), отрывок из которой приводится ниже:

**Рисунок 1 черная магия оптимизации белого ПК**

"One compiler option can kill any performance gain that you would expect with Hyper-Threading, and in some cases, actually cause a performance degradation. For example, the 2.4 kernel performs faster with the "-march=i686" than with the "-march=pentium4" option by 33%. Worst case performance gain due to incorrect compiler options would be 16%, cutting in half your expected gain. The 2.6 kernel seems to be the inverse of the 2.4 kernel. Using the "-march=i686" compiler option with the 2.6 kernel causes a performance hit. So, the rule of thumb would be, for Fedora at least, to use the "-march=i686" option on the 2.4 series of kernels and the "-march=pentium4" option with the 2.6 series of kernels. At first I thought it may be compiler related. So I tested three versions of gcc on each kernel series. The table does not show any correlation between compiler versions and machine architecture options. The only correlation is between kernels. Switching to the 2.6 kernel series would be a gain of 10% over the 2.4 kernel with Hyper-Threading enabled"

("*Одна-единственная опция компилятора способа погубить весь выигрыш в производительности, которого вы ожидаете от технологии Hyper-Threading и в некоторых случаях проигрыш становится поистине драматическим. Например, Linux ядро версии 2.4 с опцией "-march=i686" выполняется на 33% быстрее, нежели с "-march=pentium4". В худшем случае (при выборе неправильных ключей компиляции), прирост производительности составит 16%, что составляет лишь половину ожидаемого ускорения. Ядро версии 2.6, ведет себя прямо противоположным образом. Использование опции "-march=i686" вызывает снижение производительности. Таким образом, мы можем вывести следующее эмпирическое правило (по крайней мере для дистрибутива Fedora) –* ***использовать опцию "-march=i686" на ядрах семейства 2.4 и опцию "-march=pentium4" на ядрах семейства 2.6.*** *Сперва я думал, что это связано с компилятором и протестировал три версии GCC на каждом из ядер, но… не обнаружил никакой корреляции между версией компилятора и -march опцией, зато обнаружилась корреляция между ядрами. Переход на ядра семейства 2.6 в среднем давал 10% прирост производительности по сравнению с ядрами семейства 2.4, при условии, что Hyper-Threading был активирован*" — перевод мой, КК).

Тем не менее, приведенное выше эмпирическое правило срабатывает далеко не всегда и определить оптимальную комбинацию ключей можно _только_ экспериментально, однако, если оптимизация под новые типы процессоров способна вызвать обвальное падение

производительности, то со старыми типами в этом смысле дела обстоят вполне нормально. Как говориться: тише едешь — дальше будешь.

Вот только два соображения: разработчики склоны тестировать и профилировать свои приложения под наиболее массовые архитектуры (то есть те, за которые отвечает опция `-march=i686`). Новейшие модели процессоров большинству членов сообщества Open-Source недоступны и оптимизацию приходится выполнять на "ощупь" или не выполнять вообще.

Соображение номер два: программа, откомпилированная под новейшую модель процессора, становится немобильной и нетранспортабельной. Перенос на соседнюю ("морально устаревшую") машину потребует повторной перекомпиляции, а это время… К чему создавать себе лишние проблемы, соблазнившись незначительным выигрышем в производительности?!

## *векторные команды*

Помимо наращивания тактовой частоты, расширения посевной площади кэш-памяти всех уровней и других архитектурных излишеств, разработчики процессоров предлагают нам наборы векторных команд, ориентированные на обработку с графикой, цифровым звуком и видео. Естественно, сами по себе они производительности никак не добавляют и воздействуют только на те приложения, которые их явным образом используют, причем, компиляторы до векторных команд еще не доросли. В общем случае, они вообще не подозревают об их существовании, в худшем же — пытаются векторизовать циклы (особенно этим славится Intel C++), но делают это наугад и абы как. Поэтому, мы будем рассматривать лишь примеры ручной оптимизации приложений под заданный набор векторных команд.

Исторически первым таким набором оказался MMX, реализованный корпорацией Intel в "перво-пне" и получивший дальнейшее развитие в своем расширении MMXext (где "ext" сокращение от "extension").

Компания AMD, в тяжелых условиях конкурентной борьбы, нанесла ответный удар в виде своего собственного векторного набора команд, зарегистрированного под торговой маркой 3DNow!, что, по замыслу маркетологов, символизировало трехмерную графику и предполагало привлечь игроманов всех мастей. На самом же деле, одной лишь трехмерной графикой 3DNow! ничуть не ограничивался и распространял свое влияние так же и на обработку цифрового видео со звуком, то есть, фактически, представлял собой тот же самых MMX, только реализованный в другой манере.

С этого момента между Intel и AMD произошел раскол, положивший конец совместимости, к которой так стремилась AMD, а вместе с ней и все программисты. Впрочем, несмотря на все протесты со стороны Intel, AMD скопировала набор MMX, сделав его на долгие годы стандартом де-факто.

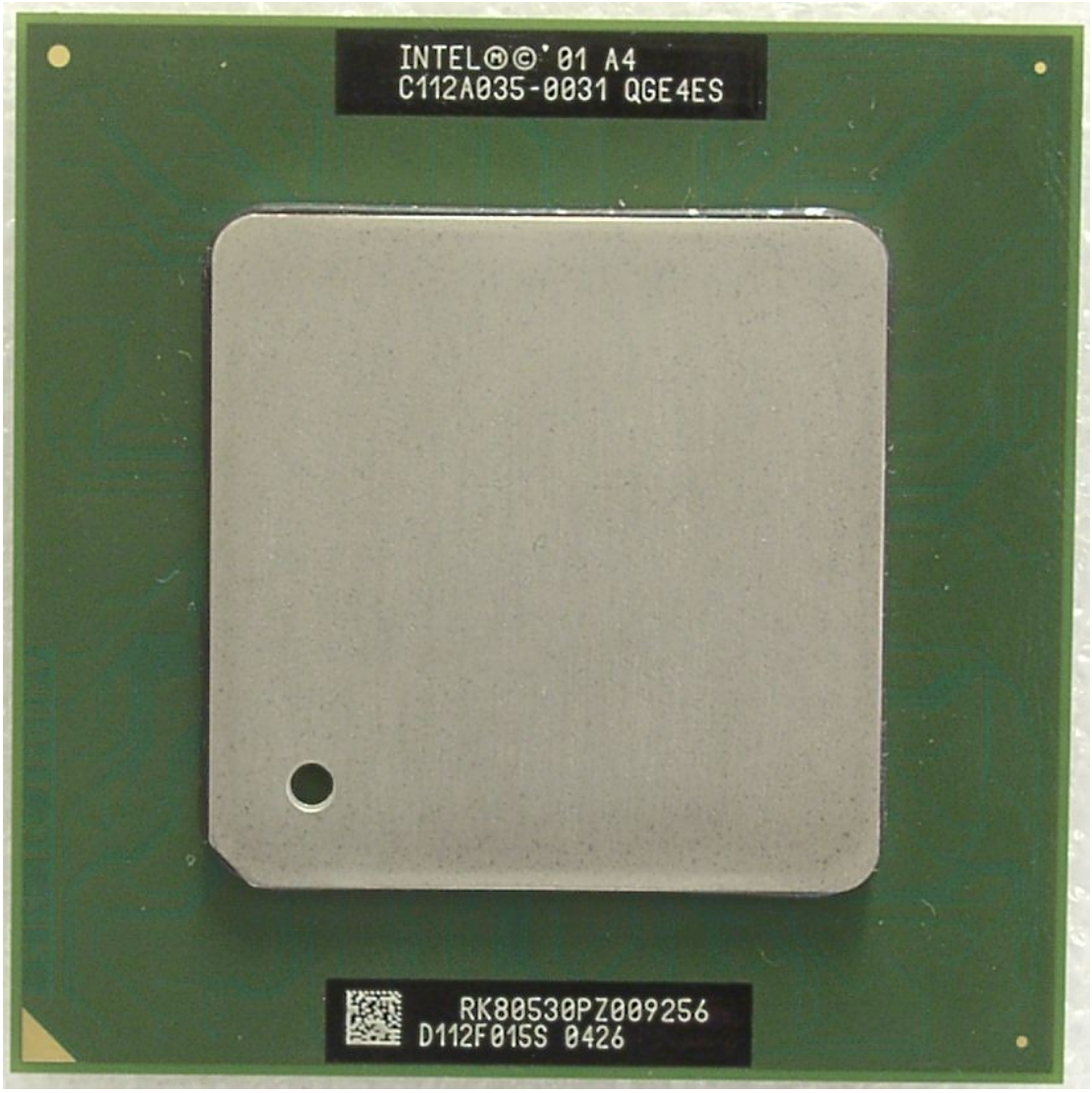


**Рисунок 2 Pentium-III с поддержкой набора векторных инструкций SSE3**

В процессе разработки Pentium-III корпорация Intel добавила в его лексикон 70 новых векторных инструкций и восемь 128-битных регистров, упакованных в аббревиатуру торговой марки SSE, "передранную" компанией AMD и перенесенную в поздние модели процессоров Athlon XP, поскольку без сохранения совместимости с лидером рынка, AMD была бы обречена на вымирание.

С появлением Pentium-4 появился и новый набор векторных инструкций, получивший название SSE2 (а SSE во избежании путаницы был переименован в SSE1). Помимо команд, оперирующих с плавающими числами двойной точности (64-бит) и 8-, 16-, 32-битных целочисленных инструкций, Intel наконец-то устранила досадное ограничение, связанное с побочным влиянием SSE-команд на MMX-регистры. Среди программистов пронесся вздох облегчения и многие из них окрестили SSE2 "должным образом реализованным SSE1".

Очередная реконструкция состоялась в Prescott'ах, добавивших инструкции, ориентированные на сигнальную обработку, прежде доступную только в специальных DSP-процессорах (Digital Signal Processor — Процессор, обрабатывающий Цифровые Сигналы), плюс команды управления виртуальными процессорами (а, точнее, их ядрами). Обновленный набор, не мудрствуя лукаво, обозвали SSE3.

Процессорная архитектура, разрекламированная под торговой маркой Core, принесла с собой 16 новых векторных инструкций, зарегистрированных под грифом SSSE3, часто воспринимаемым редактором популярных журналов как случайная опечатка.

Писком моды стал набор SSE4, представляющий собой довольно кардинально доработанный SSSE3, с кучей целочисленных инструкций (так полезных аудио и видео кодекам) и прочими соблазнительными новшествами. Первым процессором, поддерживающим

SSE4 в железе, а не на бумаге оказался 'Penryn', построенный по архитектуре Core 2. Более подробную информацию обо всех вышеперечисленных типах инструкций можно получить у самой Intel: http://www.intel.com/technology/architecture/new_instructions.htm.

Хвосту понятно, что SSE4 круче, чем SSE3, а SS3 круче, чем MMX. Подчеркиваю еще раз: это обстоятельство понятно только _хвосту_ то есть, оболваненному рекламой пользователю, уже научившемуся компилировать чужие программы, но никогда не программировавшего самостоятельно. Весь вопрос в том, какие именно векторные команды выбирает программист для решения поставленной перед ним задачи. Если набора SSE2 оказывается вполне достаточно, то заручаться поддержкой SSE3/SSE4 совершенно необязательно, тем более, что это ограничивает круг потенциальных пользователей программы.

Сами по себе векторные команды — это просто лексический балласт, а, как известно, искусство владения языком (не важно каким — машинным или человеческим!) определяется в первую очередь не количеством известных слов, а умением выразить свою мысль теми немногочисленными словами, которые крутятся в голове. В практическом плане это означает, что большинство разработчиков крайне скептически относятся к новым наборам инструкций и неохотно включают их в свои программы. Тем не менее, мы не в девяностых годах живем и MMX активно вытесняется SSE1/SSE2.

Программы, критичные к быстродействию, либо определяют тип процессора автоматом, либо предоставляют пользователю возможность выбрать используемый набор (наборы) векторных инструкций самостоятельно (такой режим получил название "форсированного"). Тут-то большинство пользователей совершает роковую ошибку, выбирая один единственный набор — самый "крутой" из всех имеющихся. Вот только производительность от этого никак не увеличивается, а даже наоборот, уменьшается. Почему?!

Да потому, что если в программе заявлена поддержка "оптимизации под SSE4", то это ровным счетом ничего не значит! Откуда мы знаем _какая_ часть кода реально написана под SSE4?! Это может быть всего пара особо критичных функций. Очень часто именно так и случается. Программа на 90% написана на Си, 9% приходится на ассемблерные MMX-модули и 1% — на SSEx. Несмотря на то, что SSEx включает в себя подмножество MMX, утилита configure этого не "знает" и при выборе одного лишь SSEx отключает оптимизированные MMX-модули, заменяя их не оптимизированными Си-аналогами.

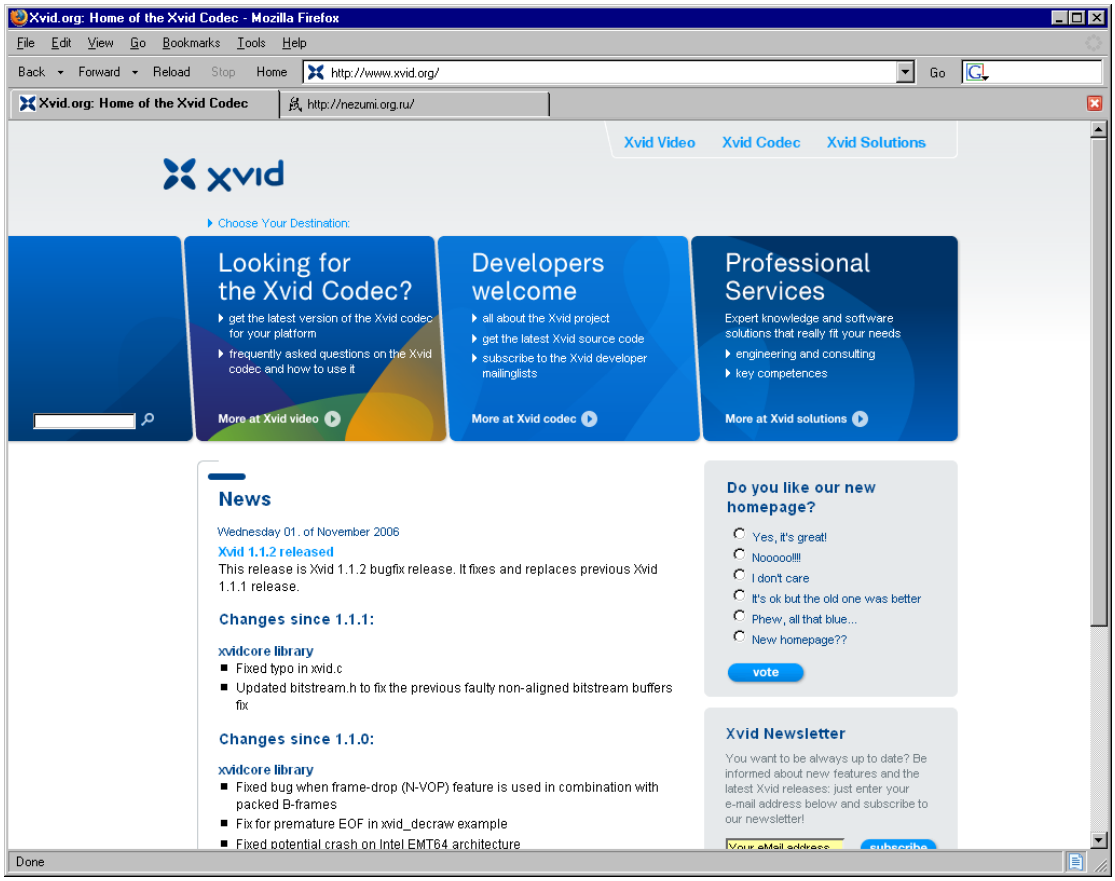

**Рисунок 3 демонстрация неоднозначности выбора наборов векторных инструкций на примере кодека XviD**

Ладно, все это демагогия. Возьмем какую-нибудь мультимедийную программу (например, кодек XviD, последнюю версию исходных текстов которого можно скачать с http://www.xvid.org/Downloads.43.0.html) и посмотрим как сочетается теория с практикой.

Распаковав архив, ищем контекстным поиском что-нибудь типа "SSE2" или "3DNow" и находим в файле config.c следующий фрагмент кода, позволяющий пользователю форсировать выбор конкретного набора векторных инструкций, включающих в себя MMX, MMXext, SSE, SSE2, 3DNow! и 3DNow!Ext:

```
case IDD_COMMON :
        cpu_force                       = IsDlgChecked(hDlg, IDC_CPU_FORCE);
        EnableDlgWindow(hDlg, IDC_CPU_MMX,          cpu_force);
        EnableDlgWindow(hDlg, IDC_CPU_MMXEXT,       cpu_force);
        EnableDlgWindow(hDlg, IDC_CPU_SSE,          cpu_force);
        EnableDlgWindow(hDlg, IDC_CPU_SSE2,         cpu_force);
        EnableDlgWindow(hDlg, IDC_CPU_3DNOW,        cpu_force);
        EnableDlgWindow(hDlg, IDC_CPU_3DNOWEXT,     cpu_force);
        break;
```

**Листинг 1 xvidcore-1.1.2\vfw\src\config.c**

А теперь откроем каталог ./SRC и посмотрим что у нас там: 26 файлов, оптимизированных под MMX, 6 — под MMXext, 5 — под SSE2, 2 — под 3DNow!, 6 — под 3DNow!Ext, плюс еще наблюдается некоторое количество модулей, написанных для архитектур Intel Itanium, AMD x86-64 и Power PC, но о них сейчас разговор не идет (поскольку, они не являются ни подмножеством, ни надмножеством рассматриваемых нами наборов векторных инструкций).

Количество функций, написанных на том или ином наборе инструкций, подсчитать несложно, но муторно да и без этого видно, что SSE – отдыхает и если вырубить MMX, то XviD будет работать ну очень медленно…

С другой стороны, наличие функций, оптимизированных под SSE еще не доказывает их превосходства над MMX. Ведь это _разные_ функции, зачастую написанные _разными_ программистами с непредсказуемой квалификацией (или отсутствуем таковой). Допустим, в некотором проекте содержится большое количество годами вылизываемого MMX-кода, написанного талантливыми людьми, владеющими техникой профилировки не хуже, чем Троица владеет карате. И представим, что в ряды разработчиков вливается пионер, прочитавший руководство по SSE-командам по диагонали и написавший чудовищно тормозной код, включенный в финальный проект по недосмотру координатора (а координатор не бог и знать всех аспектов оптимизации он не может). В любом случае, у программистов накоплен огромный опыт работы с MMX, а SSEx последних версий им еще предстоит разгрызть и освоить. Печально, когда в стремлении не отстать от прогресса, в программы включается сырой код.

Короче говоря, возникает тупиковая ситуация. Доверять программе автоматический выбор векторных команд мы не можем, поскольку никто не знает насколько они хорошо реализованы, а применение форсированной оптимизации ограничено неполнотой наших представлений о структуре программы. До тех пор, пока мы не распотрошим исходные тексты и не прикинем какая часть кода на каких наборах реализована, форсированный режим будет давать непредсказуемый результат.

Что же остается?! А остается Его Величество Эксперимент!!! Поскольку, наборов векторных инструкций существует не так уж и много, выбор оптимального сочетания не займет много времени. В одних случаях более быстрым окажется SSEx, в других — MMX. Про 3DNow! мы помним, но в силу малой рыночной доли процессоров AMD, скромно промолчим.

## *заключение*

UNIX-системы предоставляют пользователю практически неограниченную свободу для творчества, оставляя его наедине со множеством рычагов управления, многие из которых вообще никак не подписаны, а подписанные содержат магические аббревиатуры, объясняемые совершенно в других местах. Документация (даже если она и присутствует) покрывает лишь малую часть вопросов…

Это и есть расплата за свободу. Если Windows/Mac OS – это Кадиллак, то UNIX скорее похож на трактор, водитель которого способен разобрать мотор с закрытыми глазами и собрать его обратно. Многих это коробит. Трудно представить, чтобы человек с именем Анастасия

читал Intel Manual и курил спецификации на MPEG2 перед запуском DVD, но… другие просто не представляют себе как можно ездить на машине, не внеся в нее пару десятков конструктивных изменений.

Это два мира и умение компилировать программы еще не означает умение компилировать их _хорошо_

**Рисунок 4 UNIX – это бульдозер, а бульдозер — это сила, особенно если за штурвалом сидит человек по имени Анастасия**

## >>> врезка что в имени твоем

- MMX – аббревиатура MMX, расшифровываемая то как MultiMedia eXtension (Мультимедийное Расширение), то как Multiple Math (Мультиплексная Математика), то как Matrix Math eXtension (Матричное Математическое Расширение) по официальной версии представляет собой бессмысленный набор символов, типа слогана, однако, документы судебных баталий между Intel и AMD показывают, что MMX все-таки расшифровывается как "Matrix Math Extensions", подробнее об этом можно узнать на: http://en.wikipedia.org/wiki/MMX;
- SIMD — Single Instruction, Multiple Data (Одна Инструкция, Много Данных) торговая марка, объединяющая под своим крылом различные наборы ***векторных*** инструкций, обрабатывающих более одной порции данных одновременно, например, складывающих два массива чисел;
- SSE — в девичестве ISSE: Internet Streaming SIMD Extensions (Расширение Потоковых SIMD инструкций Интернета) — позднее было переименовано в Streaming SIMD Extensions (Потоковое SIMD расширение), кодовое название KNI – Katmai New Instructions (Новые Инструкции процессора Katmai, официально выпущенного под торговой маркой Pentium III);

## >>> врезка ссылки по теме

- **Intel Hyper-Threading on Linux: Fact or Myth**:
  - статья, рассказывающая о превратностях оптимизации (на английском языке): http://www.linuxelectrons.com/News/HowTO/20040226231747944;

- **XviD**:
  - главная страница проекта XviD, откуда можно скачать исходные тексты, готовые бинарные сборки и короткий faq (на английском языке): http://www.xvid.org;
- **MMX**:
  - статья по технологии MMX на свободной энциклопедии (на английском языке): http://en.wikipedia.org/wiki/MMX;
- **Streaming SIMD Extensions**:
  - статья по SSE-командам на свободной энциклопедии (на английском языке): http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions;
- **X86 assembly instructions you always wanted but intel didnt give them to you**:
  - интересная статья по векторной SSE-оптимизации (на английском языке): http://guru.multimedia.cx/category/optimization;

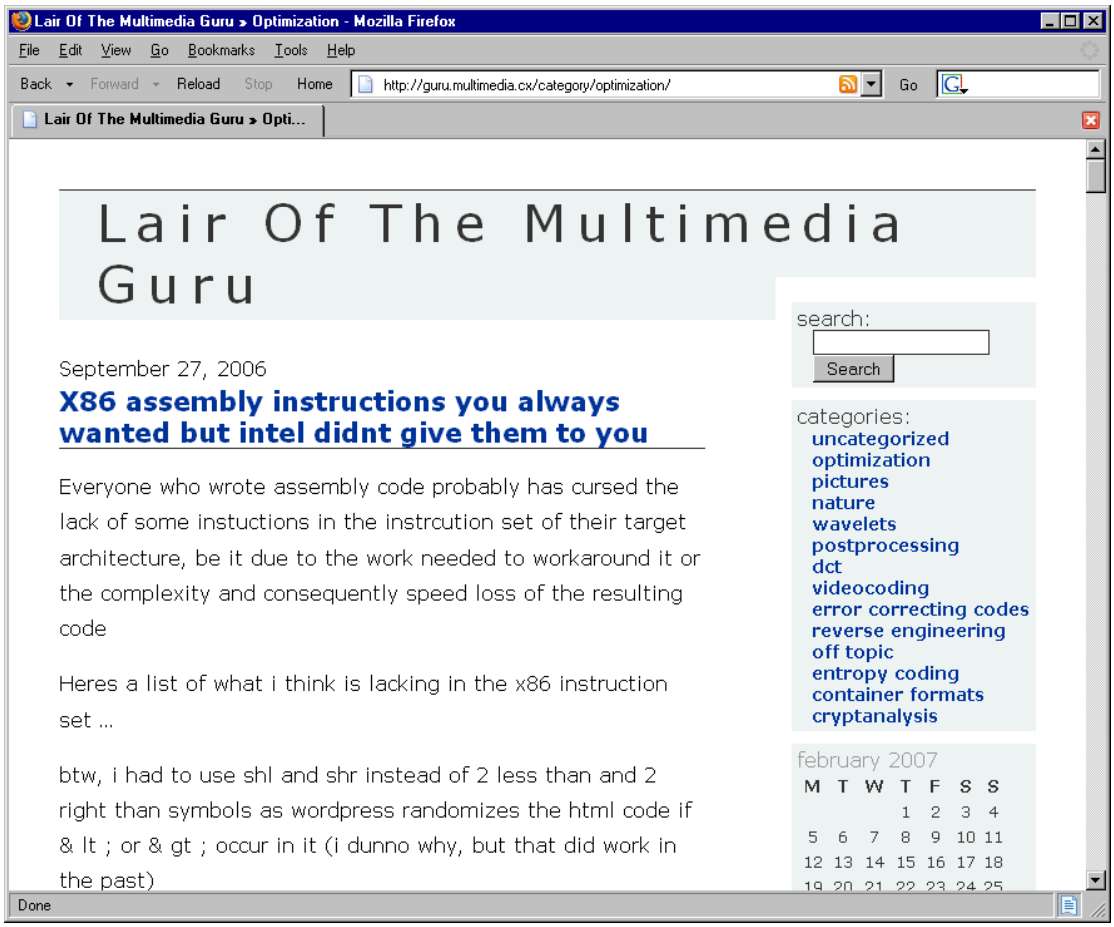


**Рисунок 5 вся правда о SSEx**