Архив статей wasm.ru. Том 4: Оптимизация, компиляторы и защита от отладки

Собрание русскоязычных статей сайта wasm.ru о программировании, ассемблере, реверс-инжиниринге и компьютерной безопасности. Том 4 из 11. Разделы: 6. Оптимизация; 7. Компиляторы; 8. Защита от отладки. Все приложенные файлы находятся в wasm_files.zip.

6. Оптимизация

Оптимизация для процессоров семейства Pentium: 3. Вызов ассемблерных функций из языка высокого уровня

Автор: Агнер Фог, пер. Aquila

Дата: 2002-01-01

Вы можете использовать встроенный ассемблер или написать процедуру полностью на ассемблере и вставить ее в свой проект. Если вы выберете последний вариант, рекомендуется, что компилятор умел переводить высокоуровневый код напрямую в ассемблер. Это даст вам гарантию, что вы получите верный метод вызова функции. Большинство компилторов C++ умеют это делать.

Методы вызова функций и манглинг имен могут быть достаточно сложными. Существует много различных соглашений о вызовах функций, и разные марки компиляторов не совместимы друг с другом в этом отношении. Если вы вызываете ассемблерную процедуру из C++, самым логичным и совместимым будет объявляление вашей функции как extern "C" и _cdecl. К имени ассемблерной функции надо прибавить символ подчеркивания и компилировать ее нужно указав чувствительность к регистрам в именах внешних функций и переменных (опция -mx).

Если вам нужно использовать перегружаемые функции, перегружаемые операторы, функции-члены классов и другие расширения C++, тогда вам нужно сначала написать это на C++ и заставить ваш компилятор перевести это в ассемблер, чтобы получить верную информацию о линковке и методе вызова, так как они могут отличаться у разных компиляторов. Если вам нужна ассемблерная функция, соглашение о вызове которой отличается от extern "C" и _cdecl, вам нужно получить ее имя, создаваемое компилятором. Вот пример перегруженной функции вычисления квадратного корня:

```
; int square (int x);
SQUARE_I PROC NEAR              ; целочисленная функция вычисления квадратного корня
@square$qi LABEL NEAR           ; имя, создаваемое компилятором Borland
?square@@YAHH@Z LABEL NEAR      ; имя, создаваемое компилятором Microsoft
_square__Fi LABEL NEAR          ; имя, создаваемое компилятором Gnu
PUBLIC @square$qi, ?square@@YAHH@Z, _square__Fi
        MOV     EAX, [ESP+4]
        IMUL    EAX
        RET
SQUARE_I ENDP

; double square (double x);
SQUARE_D PROC NEAR              ; функция вычисления квадратного корня с двойной точностью

@square$qd LABEL NEAR           ; имя, создаваемое компилятором Borland
?square@@YANN@Z LABEL NEAR      ; имя, создаваемое компилятором Microsoft
_square__Fd LABEL NEAR          ; имя, создаваемое компилятором Gnu
PUBLIC @square$qd, ?square@@YANN@Z, _square__Fd
        FLD     QWORD PTR [ESP+4]
        FMUL    ST(0), ST(0)
        RET
SQUARE_D ENDP
```

Способ передачи параметров зависит от соглашения о вызове функции:

```
соглашение          порядок помещения параметров в стек  кто очищает стек

_cdecl              обратный                             вызывающий
_stdcall            обратный                             процедура
```

```
_fastcall          зависит от компилятора              процедура
_pascal            прямой                              процедура
```

Использование регистров в 16-ти битном режиме или Windows, C или C++: 16-ти битное значение возвращается в AX, 32-х битное значение в DX:AX, значение с плавающей запятой в ST(0). Регистры AX, BX, CX, DX, ES и арифметические флаги могут быть изменены процедурой; другие регистры должны быть сохранены и восстановлены. Процедура может полагаться на то, что при вызове другой процедуры значение SI, DI, BP, DS и SS не изменится.

Использование регистров в 32-х битных Windows, C++ и других языках программирования: целочисленное значение возвращается в EAX, значение с плавающей точкой возвращается в ST(0). Регистры EAX, ECX, EDX (но не EBX) могут быть изменены процедурой; все другие регистры должны быть сохранены и восстановлены. Сегментные регистры нельзя изменять даже временно. CS, DS, ES и SS указываеют на плоский сегмент. FS используется операционной системой. GS не используется, но зарезервирован. Флаги могут меняться процедурой, но со следующими ограничениями: флаг направления равен 0 по умолчанию. Его можно изменить временно, но при выходе из процедуры необходимо очистить. Флаг прерывания не может быть очищен. Стековый регистр плавающей запятой пуст при входе в процедуру и должен быть пустым при выходе, если только ST(0) не используется для возвращения значения. Регистры MMX можно изменять, и если это было сделано, их нужно очистить с помощью инструкции EMMS перед возвращением или вызовом любой другой процедуры, которая может использовать регистры плавающей запятой. Все XMM регистры можно свободно изменять. Правила для передачи параметров и возвращения значений через регистры XMM объяснены в интеловском документе AP 589. Процедура может полагаться на неизменность EBX, ESI, EDI, EBP и всех сегментных регистров при вызове другой процедуры.

Оптимизация для процессоров семейства Pentium: 1. Введение

Автор: Агнер Фог, пер. Aquila

Дата: 2002-01-01

Это руководство подробно рассказывает о том, как писать оптимизированный код на ассемблере, с ориентированием на семейство микропроцессоров Pentium.

Большая часть приведенной здесь информации основывается на моих собственных исследованиях. Много людей прислали мне полезную информацию и исправления для этого руководства, и я обновляю его каждый раз, когда у меня появляется важная информация. Поэтому данное руководство наиболее точно, подробно и исчерпывающе, нежели любой другой источник информации, и оно содержит множество подробностей, которые трудно найти гед-бы то ни было еще. Эта информация во многих случаях позволит вам, сколько тактов займет выполнение определенного кода. Тем не менее, я не утверждаю, что вся информация в данном руководстве верна: некоторые измерения очень трудно или невозможно сделать точно, и у мнея нет доступа к информации о внутренней реализации, которая есть у тех, кто пишет интеловские руководства.

В данном руководстве обсуждаются следующие версии процессоров Pentium:

аббревиатура	имя
PPlain	обычный старый Pentium
PMMX	Pentium с MMX
PPro	Pentium Pro
PII	Pentium II (включая Селероны и Ксеоны)
PIII	Pentium III (включая его разновидности)

В данном руководстве используется синтаксис MASM 5.10. Для языка ассемблера x86 не существует официального стандарта, но это фактически является стандандартом, так как многие ассемблеры имеют режим совместимости с MASM 5.10. (Я не рекомендую использовать сам MASM версии 5.10, так как в нем есть серьезный баг, проявляющийся в 32-х битном режиме. Используйте TASM или более позднюю версию MASM).

Некоторые замечания в данном руководстве могут показаться критикой Intel. Это не нужно понимать, что другие марки лучше. Семейство микропроцессоров Pentium вполне сравнимы со своими конкурентами, они лучше документированы и у них есть другие отличительные особенности.

Программированить на ассемблере гораздо сложнее, чем на языке высокого уровня. Сделать ошибку очень легко, а найти ее очень трудно. Вы были предупреждены! Предполагается, что у читателя уже есть опыт в программировании на ассемблере. Если нет, пожалуйста, прочитайте соответствующие книги по этому теме и наберитесь определенного опыта в программировании, прежде чем вы начнете выполнять сложную оптимизацию.

Арихтектура чипов PPlain и PMMX имеет много особенностей, которые оптимизированны специально под некоторые наиболее часто используемые инструкции или комбинации инструкций. В этих случаях нельзя использовать общие методики оптимизации. Следовательно, правила для оптимизирования программного обеспечения весьма сложны и имеют много исключений, но возможный выигрыш в производительности значителен. У процесоров PPro, PII и PIII совершенно разный дизайн в том, как процессор берет на себя часть оптимизации, запуская инструкции в совершенно разном порядке, но усложненный дизайн этих процессоров становится причининой множества потенциальных узких мест, поэтому можно много выиграть, оптимизируя код вручную для этих процессоров. У Pentium 4 также другой дизайн, и пути оптимизации для Pentium 4

отличаются от тех, которые используются для других версий. Это руководство не рассказывает о Pentium 4 - читатель может обратиться к руководствам от Intel.

Прежде, чем вы начнете конвертировать ваш код в ассемблер, убедитесь, что используемый вами алгоритм оптмален. Зачастую вы можете гораздо сильнее улучшить ваш код путем оптимизации алгоритма, чем переписыванием кода на ассемблере.

Затем вам нужно выделить критические части вашей программы. Зачастую более, чем 99% времени процессора тратится на внутренний цикл программы. В этом случае вам нужно только оптимизировать этот цикл и оставить все остальное на языке высокого уровня. Некоторые асм-программисты тратят огромное количество энергии на оптимизирование не тех частей своей программы, единственным значительным результатом чего становится то, что программу становится труднее отлаживать и распространять!

Если не совсем понятно, какие части вашей программы являются критическим, вы можете использовать профайлер, чтобы найти их. Если выясняется, что узким местом является доступ к диску, вы можете модифицировать вашу программу таким образом, чтобы сделать доступ к диску последовательным (для улучшения работы кэша диска), а не переключаться на ассемблерное программирование. Если узким местом является вывод графики, вы можете поискать путь к уменьшению вызовов графических процедур.

Некоторые высокоуровневые компиляторы предлагают относительно хорошую оптимизацию для некоторых процессоров, но дальнейшая оптимизация вручную обычно дает лучшие результаты.

Пожалуйста, не посылайте мне ваши вопросы о программировании. Я не собираюсь делать за вас вашу домашнюю работу.

Удачи в охоте за наносекундами!

Оптимизация для процессоров семейства Pentium: 2. Литература

Автор: Агнер Фог, пер. Aquila

Дата: 2012-11-24

Множество полезную литературы и туториалов можно скачать бесплатно с www-сайта Intel или заказать на CD-ROM. Рекомендуется, чтобы вы изучили эту литературу для ознакомления с архитектурой процессора. Тем не менее, документы от Intel не всегда точны - особенно много ошибок содержат туториалы (похоже, что они не тестят собственные примеры).

Я не будут давать здесь прямые ссылки, потому что местоположение файлов часто меняется. Вы можете найти нужные вам документы используя функцию поиска на developer.intel.com или перейдя по ссылке, которые находятся на www.agner.org/assem.

Некоторые документы в формате .PDF. Если у вас нет никакого программного обеспечения для просмотра или распечатки файлов .PDF, вы можете скачать просмотровщик файлов Acrobat с www.adobe.com.

VTUNE - это утилита от Intel для оптимизирования кода. Я не тестировал его и поэтому не могу ничего сказать по этому поводу.

Кроме Intel существует множество другой полезной информации. Эти источники перечислены в FAQ ньюсгруппы comp.lang.asm.x86. Также обратите внимание на ссылки, приведенные на www.agner.org/assem.

Оптимизация для процессоров семейства Pentium: 4. Отладка

Автор: Агнер Фог, пер. Aquila

Дата: 2012-11-24

Отладка ассемблерного кода может оказаться довольно трудоемкой и неприятной, как вы, возможно, уже заметили. Я рекомендую вам сначала написать то, что вы хотите оптимизировать как подпрограмму на языке высокого уровня. Затем напишите тестовую программу, в которой вы будете отлаживать подпрограмму. Убедитесь, что тестовая программа удовлетворяет всем условиям ветвления и выравнивания.

Затем переведите код на язык ассемблера.

Теперь вы можете начать оптимизировать. Каждый раз, когда вам нужно сделать изменения, вы будете переходить к тестовой программе, чтобы убедиться, что она работает. пронумеруйте все ваши версии и сохраните их, чтобы вы могли в случае ошибки вернуться к ним.

Протестируйте скорость наиболее критичных частей вашей программы с помощью метода, изложенного в главе 30 или с помощью тестовой программы. Если код значительно медленнее, чем ожидалось, тогда возможно, что: неправильно используется кэш (глава 7), невыравненны операнды (глава 6), цена первого запуска (глава 8), неправильное предсказание переходов (глава 22), проблемы загрузки кода (глава 15), потери скорости при чтении регистра (глава 16) или долгая цепь зависимости (глава 20).

Оптимизация для процессоров семейства Pentium: 5. Модель памяти

Автор: Агнер Фог, пер. Aquila

Дата: 2002-01-01

Пентиумы спроектированны в основном для 32-х битного кода, и качество ужасно при использовании 16-битного кода. Сегментирование вашего кода и данных также значитель отражается на производительности, поэтому вам следует использовать 32-х битный плоский режим и операционную систему, которая поддерживает этот режим. Примеры, приводимые в данном руководстве, если иное не оговорено особо.

Оптимизация для процессоров семейства Pentium: 6. Выравнивание

Автор: Агнер Фог, пер. Aquila

Дата: 2002-01-01

Все данные в RAM должны быть выравненены так, чтобы их адреса были кратны 2, 4, 8 или 16 согласно данной схеме:

	выравнивание	
размер операнда	PPlain и PMMX	PPro, PII и PIII
1 (byte)	1	1
2 (word)	2	2
4 (dword)	4	4
6 (fword)	4	8
8 (qword)	8	8
10 (tbyte)	8	16
16 (oword)	n.a.	16

На PPlain и PMMX при обращении к невыравненным данным будет теряться по меньшей мере 3 такта, если пересечена граница в 4 байта. Потери будут выше при пересечении границы кэша.

На PPro, PII и PIII невыравненные данные удут стоить вам 6-12 дополнительных тактов, если пересечена граница кэша. Невыравненные операнды, меньшие чем 16 байтов и не перешедшие границу в 32 байта, не приводят к потерям.

Выравнивание данных на 8 или 16 в стеке двойных слов может стать проблемой. Общий метод решения - установить выравненный указатель на кадр стека. Функция с выравненными локальными данными может выглядеть примерно так:

```
_FuncWithAlign PROC NEAR
        PUSH    EBP                         ; пролог
        MOV     EBP, ESP
        AND     EBP, -8                     ; выравнивание указателя на кадр
                                              стека на 8
        FLD     DWORD PTR [ESP+8]           ; параметр функции
        SUB     ESP, LocalSpace + 4         ; резервируем локальные данные
        FSTP    QWORD PTR [EBP-LocalSpace] ; теперь сохраняем что-нибудь в
                                            ; локальной переменной
        ...
        ADD     ESP, LocalSpace + 4         ; эпилог, восстанавливаем ESP

        POP     EBP                         ; (потеря скорости AGI на PPlain/PMMX)
        RET
_FuncWithAlign ENDP
```

В то время как выравнивание данных важно всегда, выравнивание кода не является необходимым на PPlain и PMMX. Принципы выравнивания кода на PPro, PII и PIII изложены в главе 15.

Оптимизация для процессоров семейства Pentium: 7. Кэш

Автор: Агнер Фог, пер. Aquila

Дата: 2012-11-24

У PPlain и PPro 8 килобайт кэша первого уровня для кода и 8 килобайт для данных. У PMMX, PII и PIII по 16 килобайт для кода и данных. Данные в кэше первого уровня можно читать или перезаписывать всего лишь за один такт, в то время как выход за границы кэша может стоить множества тактов. Поэтому важно, понимать, как работает кэш, чтобы использовать его более эффективно.

Кэш данных состоит из 256 или 512 рядов по 32 байта в каждом. Каждый раз, когда вы считываете элемент данных, который еще не находится в кэше, процессор считает весь кэш-ряд из памяти. Кэш-ряды всегда выравниваются по физическому адресу, кратному 32. Когда вы считываете байт по адресу, который кратен 32, чтение или запись в следующие 31 байт вам не будет стоить практически ничего. Вы можете использовать это к своей выгоде, организовав данные, которые используются вместе, в блоки по 32 байта. Если, например, у вас есть цикл, в котором обрабатываются два массива, вы можете перегруппировать эти два массива в один массив структур, чтобы данные, которые используются вместе, находились в памяти рядом друг с другом.

Если размер массива кратен 32 байтам, вам следует и выравнивать его по границе 32 байта.

Кэш set-ассоциативен. Это означает, что кэш-ряд нельзя ассоциировать с произвольным адресом памяти. У каждого кэш-ряда есть 7-ми битное значение, которое задает биты 5-11 адреса в физической памяти (биты 0-4 определяются положением элемента данных в кэш-ряде). У PPlain и PPro два кэш-ряда для каждого из возможных 128 set-значений, поэтому с определенным адресом в памяти можно ассоциировать только два жестко заданных кэш-ряда. В PMMX, PII и PIII - четыре.

Следствием этого является то, что кэш может содержать не более двух или четырех различных блока данных, у которых совпадают биты 5-11 их адресов. Вы можете определить, совпадают ли эти биты у двух адресов следующим образом: удалите нижние пять битов каждого адреса, чтобы получить значение, кратное 32. Если разница между этими обрезанными значениями кратна 4096 (=1000h), значит у этих адресов одно и то же set-значение.

Давайте я проиллюстрирую вышеизложенное с помощью следующего кода, где ESI содержит адрес, кратный 32:

```
AGAIN:  MOV  EAX, [ESI]
        MOV  EBX, [ESI + 13*4096 +  4]
        MOV  ECX, [ESI + 20*4096 + 28]
        DEC  EDX
        JNZ  AGAIN
```

У всех трех адресов, использованных здесь, совпадают set-значения, потому что разница между обрезанными адресами будет кратна 4096. Этот цикл будет очень плохо выполняться на PPlain и PPro. Во время записи в ECX нет ни одного свободного кэш-ряда, поэтому тот, который был использован для значения, помещенного в EAX, заполняется данными с [ESI+20*4096] до [ESI+20*4096+31] и записывает значение в ECX. Затем во время чтения EAX оказывается, что кэш-ряд, содержавший значение для EAX, уже изменен, поэтому то же самое происходит с кэш-рядом, содержащим значение для EBX, и так далее. Это пример нерационального использования кэша: цикл занимает около 60 тактов. Если третью линию изменить на:

```
        MOV  ECX, [ESI + 20*4096 + 32]
```

мы пересечем границу в 32 байта, поэтому у нас будет другое set-значение, и не возникнет никаких проблем с ассоциирование кэш-рядов к этим трем адресам. Цикл теперь занимает 3 такта (не считая первого раза) - весьма значительное улучшение! Стоит упомянуть, что у PMMX, PII и PIII для каждого set-значения есть четыре кэш-ряда. (Некоторые интеловские документы ошибочно утверждают, что у PII по два кэш-ряда на каждое set-значение).

Определить, одинаковые или у переменных set-значения, может оказаться достаточно сложным, особенно, если они разбросаны по разным сегментам. Лучшим, что вы можете придумать для избежания проблем подобного рода - это держать все данные, используемые в критической части вашей программы внутри одного блока, который будет не больше, чем кэш, либо в двух блоках, не больших, чем половина от его размера (например, один блок для статических данных, а другой для данных в стеке). Это будет гарантией того, что ваши кэш-ряды используются оптимальным образом.

Если критическая часть вашего кода имеет дело с большими структурами данных или данными, располагающимися в памяти случайным образом, тогда вам стоит подумать о том, чтобы держать наиболее часто используемые переменные (счетчики, указатели, флаговые переменные и т.д.) в одном блоке с максимальным размеров в 4 килобайта, чтобы использовались все кэш-ряды. Так как вам еще требуется стек для параметров подпрограммы и возвращаемых адресов, лучшее, что вы можете сделать - это скопировать наиболее часто используемые статические данные в динамические переменные в стеке, а после выхода из критической части кода скопировать обратно те из них, которые подверглись изменению.

Чтение элемента данных, которого нет в кэше первого уровня приводит к тому, что весь кэш-ряд будет заполнен из кэша второго уровня. Если ее нет и в кэше второго уровня, то вы получите еще большую задержку, которая может оказаться совсем гигантской, если вы пересечете границу между страницами памяти.

При считывании больших блоков данных из памяти, скорость ограничена временем, уходящим на заполнение кэш-рядов. Иногда вы можете увеличить скорость, считывая данные в непоследовательном порядке: еще не считав данные из одного кэш-ряда, начать читать первый элемент из следующего кэш-ряда. Этот метода можете повысить скорость на 20-40% при считывании данных из основной памяти или кэша второго уровня на PPlain и PMMX, или из кэша второго уровня на PPro, PII и PIII. Недостаток этого метода заключается в том, что код программы становится очень запутанным и сложным. Другую информацию об этом методе вы можете получить на www.intelligentfirm.com.

Когда вы пишите в адрес, которого нет в кэше первого уровня, тогда значение пойдет прямо в кэш второго уровня или в память (в зависимости от того, как настроен кэш второго уровня) на PPlain и PMMX. Это займет примерно 100 ns. Если вы пишите восемь или более раз в тот же 32-байтный блок памяти, ничего не читая из него, и блок не находится в кэше первого уровня, возможно стоит сделать фальшивое чтение из него, чтобы он был загружен в кэш-ряд. Все последующие чтения будут производиться в кэш, что занимает всего лишь один такт. На PPlain и PMMX иногда происходит небольшая потеря в производительности при повторяющемся чтении по одному и тому же адресу без чтения из него между этим.

На PPro, PII и PIII запись обычно ведет к загрузке памяти в кэш-ряд, но возможно указать область памяти, с которой следует поступать иначе (смотри Pentium Pro Family Developer's Manual, vol. 3: Operating System Writer's Guide").

Другие пути увеличения скорости чтения и записи в память обсуждаются в главе 27.8.

У PPlain и PPro есть два буфера записи, у PMMX, PII и PIII - четыре. На PMMX, PII и PIII у вас может быть до четырех незаконченных записей в некэшированную память без задержки последующих инструкций. Каждый буфер записи может обрабатывать операнды до 64 бит длиной.

Временные данные удобно хранить в стеке, потому что стековая область как правило находится в кэше. Тем не менее, вам следует избегать проблем с выравниваем, если элементы ваших данных больше, чем размер слова стека.

Если время жизни двух структур данных не пересекается, они могут разделять одну и ту же область памяти, чтобы повысить эффективность кэша. Это согласуется с общей практикой держать временные переменные в стеке.

Хранение временных данных в регистрах, разумеется, более эффективно. Так как регистров мало, вы, возможно, захотите использовать [ESP] вместо [EBP] для адресации данных к стеку, чтобы освободить EBP для других целей. Только не забудьте, что значение ESP меняется каждый раз, когда вы делаете PUSH или POP. (Вы не можете использовать ESP под 16-ти битным Windows, потому что прерывание таймера будет менять верхнее слово ESP тогда, когда это невозможно предугадать.)

Есть отдельных кэш для кода, которых схож с кэшем данных. Размер кэша кода равен 8 килобайтам на PPlain и PPro и 16 килобайтам на PMMX, PII и PIII. Важно, чтобы критические части вашего кода (внутренние циклы) умещались в кэш кода. Часто используемые процедуры следует помещать рядом друг с другом. Редко используемые ветви или процедуры нужно дерaжть в самом низу вашего кода или где-нибудь еще.

Оптимизация для процессоров семейства Pentium: 8. Исполнение кода в первый раз

Автор: Агнер Фог, пер. Aquila

Дата: 2012-11-24

Обычно исполнение кода в первый раз занимает намного больше, чем при последующих повторениях в силу следующих причин:

• Загрузка кода из RAM в кэш занимает намного больше времени, чем его выполнение.
• Любые данные, к которым обращается код должны быть загружены в кэш, что также занимает много времени. Когда код повторяется, данные, как правило, уже находятся в кэше.
• Инструкция перехода не будет находится в буфере предсказывания переходов в первый раз, поэтому маловероятно, что она будет предсказана правильно. Смотри главу 22.
• На PPlain декодирование инструкций является узким месом. Если определение длины инструкции занимает один такт, то процессор не может раскодировать за такт две инструкции, так как он не знает, где начинает вторая инструкция. PPlain решает эту проблему, запомина длину любой инструкции, которая осталась в кэше. Как следствие, ряд инструкции не спариваются на PPlain во время первого исполнения, если только первые две инструкции не занимают по одному байту. У PMMX, PPro, PII и PIII таких потерь нет.

В силу этих четырех причин, код внутри цикла будет занимать больше времени, исполняясь в первый раз, нежели в последующие.

Если у вас большой цикл, который не влезает в кэш кода, у вас будут постоянные потери в производительности, потому что код будет исполняться не из кэша. Вам следует реорганизовать цикл, чтобы он умещался в кэш.

Если в вашем цикле очень много переходов и вызовов, то у вас будут потери из-за регулярных ошибок предсказания переходов.

Также у вас будет потеря производительности, если ваш цикл периодически обращается к структурам данных, которые слишком велики для кэша данных.

Оптимизация для процессоров семейства Pentium: 9. Задержка генерации адреса

Автор: Агнер Фог, пер. Aquila

Дата: 2012-11-24

Чтобы высчитать адрес в памяти, который нужен инструкции, требуется один такт. Обычно эти вычисления делаются одновременно с выполнение предыдущей инструкции или спаренных инструкций. Но если адрес зависит от результат инструкции, которая выполнялась в предыдущем такте, тогда вам придется подождать дополнительный такт, чтобы получить требуемый адрес. Это называется задержкой AGI.

Пример:

```
ADD EBX,4 / MOV EAX,[EBX] ; задержка AGI

Задержку в данном примере можно убрать, если поместить какие-нибудь другие
инструкции между 'ADD EBX, 4' и 'MOV EAX, [EBX]' или переписав код следующим
образом:

MOV EAX,[EBX+4] / ADD EBX,4

Также вы можете получить задержку AGI при использовании инструкции, которые
используют ESP для адресации, например, PUSH, POP, CALL и RET, если ESP был
именен в предыдущем такте такими инструкциями как MOV, ADD и SUB. У PPlain
и PMMX есть специальная схема, чтобы предсказывать значение ESP после
стековой операции, поэтому у вас не будет задержки AGI при изменении ESP с
помощью PUSH, POP или CALL. При использовании RET она может случиться только
если у него есть числовой операнд, который прибавляется к ESP.

Примеры:

ADD ESP,4 / POP ESI            ; задержка AGI
POP EAX   / POP ESI            ; нет задержки, спариваются
MOV ESP,EBP / RET              ; задержка AGI
CALL L1 / L1: MOV EAX,[ESP+8]  ; нет задержки
RET / POP EAX                  ; нет задержки
RET 8 / POP EAX                ; задержка AGI

Инструкция LEA также может стать причиной задержки AGI, если она использует
базовый или индексный регистр, который был изменен в предыдущем такте.

Пример:

INC ESI / LEA EAX,[EBX+4*ESI]  ; задержка AGI
```

У PPro, PII и PIII нет задержки AGI для чтения из памяти и LEA, но есть задержка при записи в память. Это не очень важно, если только последующий код не должен ждать, пока операция записи не будет выполнена.

Оптимизация для процессоров семейства Pentium: 11. Разбивка сложных инструкций на более простые (PPlain и PMMX)

Автор: Агнер Фог, пер. Aquila

Дата: 2012-11-24

Вы можете разбить инструкции чтения/модифицирования и инструкции чтения/модифицирования/записи, чтобы улучшить спаривание.

Пример:

```
ADD [mem1],EAX / ADD [mem2],EBX ; 5 тактов
```

Этот код можно разбить на следующую последовательность, которая будет занимать только 3 такта:

```
MOV ECX,[mem1] / MOV EDX,[mem2] / ADD ECX,EAX / ADD EDX,EBX
MOV [mem1],ECX / MOV [mem2],EDX
```

Таким же образом вы можете разбивать не спариваемые инструкции на спариваемые:

```
PUSH [mem1]
PUSH [mem2]  ; не спаривается
```

Разбивается на:

```
MOV EAX,[mem1]
MOV EBX,[mem2]
PUSH EAX
PUSH EBX     ; все спаривается
```

Другие примеры неспариваемых инструкций, которые можно разбить на простые спариваемые:

```
CDQ разбивается на MOV EDX,EAX / SAR EDX,31
NOT EAX меняется на XOR EAX,-1
NEG EAX разбивается на XOR EAX,-1 / INC EAX
MOVZX EAX,BYTE PTR [mem] разбивается на XOR EAX,EAX / MOV AL,BYTE PTR [mem]
JECXZ разбивается на TEST ECX,ECX / JZ
LOOP разбивается на DEC ECX / JNZ
XLAT меняется на MOV AL,[EBX+EAX]
```

Если разбиваение иснтрукций не повышает скорось, вы можете оставить сложные или неспариваемые конструкции, чтобы уменьшить размер кода.

Разбивание инструкций не требуется, кроме тех случаев, когда это будет генерировать меньше уопов (uops).

Оптимизация для процессоров семейства Pentium: 10. Спаривание целочисленных инструкций (PPlain и PMMX)

Автор: Агнер Фог, пер. Aquila

Дата: 2002-01-01

10.1 Совершенное спаривание

У PPlain и PMMX есть два конвеера, выполняющих инструкции, которые называются U-конвеер и V-конвеер. В определенных условий можно выполнить две инструкции одновременно, одну в U-конвеере, а другую в V-конвеере. Это практически удваивает скорость. Поэтому имеет смысл перегруппировать ваши инструкции и сделать их спариваемыми.

Следующие инструкции могут находится в любом конвеере.

• MOV регистр, память или уже заданного числа в регистр или память
• PUSH регистр или число, POP регистр
• LEA, NOP
• INC, DEC, ADD, SUB, CMP, AND, OR, XOR,
• и некоторые разновидности TEST (смотри главу 26.14)

Следующие инструкции спариваемы только в U-конвеере:

• ADC, SBB
• SHR, SAR, SHL, SAL с число в качестве аргумента
• ROR, ROL, RCR, RCL с единицей в качестве аргумента

Следующие инструкции спариваемы только в V-конвеере:

• ближний вызовой
• короткий и ближний переход
• короткий и ближний условный переход

Все другие целочисленные инструкции могут выполняться только в U-конвеере и не могут спариваться.

Две следующие друг за другом инструкции будут спариваться, если соблюденые следующие условия:

• Первая инструкция спариваема в U-конвеере, а вторая - в V-конвеере.
• Вторая инструкция не читает и не пишет из регистра, если пишет первая инструкция.

Примеры:

```
MOV EAX, EBX / MOV ECX, EAX     ; чтение после записи, не спаривается
MOV EAX, 1   / MOV EAX, 2       ; запись после записи, не спаривается
MOV EBX, EAX / MOV EAX, 2       ; запись после чтения, спаривается
MOV EBX, EAX / MOV ECX, EAX     ; чтение после чтения, спаривается
MOV EBX, EAX / INC EAX          ; чтение и запись после чтения, спаривается
```

3. Неполные регистры считаются полными регистрами. **Пример:**

```
MOV AL, BL  /  MOV AH, 0
```

пишут в разные части одного и того же регистра, не спаривается.

4. Две инструкции, пишущие в разные части регистра флагов могут спариваться не смотря на правила 2 и 3. **Пример:**

```
SHR EAX, 4 / INC EBX            ; спаривается
```

5. Инструкция, которая пишет в регистр флагов может спариваться с условным переходом несмотря на правило 2. **Пример:**

```
CMP EAX, 2 / JA LabelBigger     ; спаривается
```

6. Следующая комбинация инструкций может спариваться несмотря на тот факт, что обе изменяют указатель на стек:

```
PUSH + PUSH,  PUSH + CALL,  POP + POP
```

7. Есть ограничения на спаривание инструкций с префиксами. Есть несколько типов префиксов:

• у инструкций, обращающихся к сегменту, не являющемся сегментом по умолчанию, есть сегментный префикс.
• у инструкций, использующих 16-ти битные данные в 32-ом режиме или 32-х битные данные в 16-битном режиме, есть префикс размера операнда.
• у инструкций, использующих 32-х битную адресацию через регистры в 16-ти битном режиме, есть префикс размера адреса.
• у повторяющихся есть префикс повторения.
• у закрытых инструкций есть префикс LOCK.
• у многих инструкций, которых не было на процессоре 8086 есть двухбайтный опкод, чей первый байт равен 0FH. Байт 0FH считается префиксом только на PPlain. Наиболее часто используемые инструкции с этим префиксом следующие: MOVZX, MOVSX, PUSH FS, POP FS, PUSH GS, POP GS, LFS, LGS, LSS, SETcc, BT, BTC, BTR, BTS, BSF, BSR, SHLD, SHRD, и IMUL с двумя операндами и без числового операнда.

На PPlain инструкция с префиксом может выполняться только в U-конвеере, кроме ближних условных переходов.

На PMMX инструкции с префиксами размера операнда, размера адреса или 0FH могут выполняться в любом конвеере, в то время как инструкции с префиксами сегмента, повторения или LOCK могут выполняться только в U-конвеере.

8. Инструкция, в которой используется одновременно адресация со смещение и числовые данные на PPlain не спариваются, на PMMX могут спариваться только в U-конвеере:

```
MOV DWORD PTR DS:[1000], 0    ; не спаривается или только в U-конвеере
CMP BYTE PTR [EBX+8], 1       ; не спаривается или только в U-конвеере
CMP BYTE PTR [EBX], 1         ; спаривается
CMP BYTE PTR [EBX+8], AL      ; спаривается
```

(Другая проблема, связанная с подобными инструкциями, выполняющимися на PMMX, заключается в том, что такие инструкции могут быть длинее семи байтов, а это приводит к невозможности раскодировки больше одной инструкции за такт, подробнее это объясняется в главе 12.)

9. Обе инструкции должны быть заранее загружены и раскодированы. Это объяснение в главе 8.
10. Есть специальные правила спаривания для инструкций MMX на PMMX:

• MMX-сдвиг, инструкции упаковки или распаковки могут выполнять в любом конвеере, но не могут спариваться с другим MMX-сдвигом, инструкциями упаковки или распаковки.
• MMX-инструкции умножения могут выполняться в любом конвеер, но не могут спариваться с другими MMX-инструкциями умножения. Они занимают 3 такта, и последние 2 такта могут перекрыть последующие инструкции, также как это делают инструкции плавающей запятой (смотри главу 24).

• MMX-инструкция, обращающаяся к памяти или целочисленным регистрам может выполняться только в U-конвеер и не могут спариваться с не-MMX инструкцией.

10.2 Несовершенное спаривание

Бывают ситуации, когда две спаривающиеся инструкции не будут выполняться одновременно или будут частично рассинхронизированы во времени. Пока обе инструкции не выполняться (каждая в своем конвеере) ни одна другая инструкция не начнет выполняться.

Несовершенное спаривание возникает в следующих случаях:

• Если вторая инструкция приводит к задержке AGU (глава 9).
• Две инструкции не могут обращаться к одному и тому двойному слову в памяти одновременно:

```
MOV AL, [ESI] / MOV BL, [ESI+1]
```

Два операнда внутри одного и того же двойного слова, поэтому они не могут выполняться одновременно. Пара займет два такта для выполнения.

```
MOV AL, [ESI+3] / MOV BL, [ESI+4]
```

Эти два операнда находятся в разных двойных словах, поэтому они прекрасно спариваются и занимают всего один такт.

3. Правило 2 расширяет зону своего действия, если биты 2-4 обоих адресов одинаковы (конфликт банков кэша). Для адресов размеров в двойное слово это означает, что они не разность между обоими адресами не должна делиться на 32. **Примеры:**

```
MOV [ESI], EAX / MOV [ESI+32000], EBX ;  несовершенное спаривание
MOV [ESI], EAX / MOV [ESI+32004], EBX ;  совершенное спаривание
```

Спариваемые целочисленные инструкции, которые не обращаются к памяти, требуют один такт для выполнения, кроме неправильно предсказанных переходов. Инструкции MOV, читающие или пишущие в память также занимают один такт, если данные находятся в кэше и правильно выравненны. Нет потери в скорости при использовании сложных способов адресации, такие как смещение и масштабирование.

Спариваемая целочисленная инструкция, которая читает из памяти, делает какие-то вычисления, а затем сохраняет результат в регистрах или флагах, занимает 2 такта (инструкции чтения/модифицирования).

Спариваемая целочисленная инструкция, которая читает из памяти, делает какие-то вычисления, а затем записывает результат обратно в память, занимает 3 такта (инструкции чтения/модифицирования/записи).

4. Если инструкция чтения/модифицирования/записи спаривается с инструкцией чтения/модифицирования или чтения/модифицирования/записи, тогда они спарятся несовершенно.

Количество тактов, которые потребуются для выполнения такой пары, даны в следующей таблице:

Первая инструкция	Вторая инструкция		
	MOV или регистр	чтение/изменение	чтение/изменение/запись
MOV или регистр	1	2	3
чтение/изменение	2	2	3
чтение/изменение/запись	3	4	5

Пример:

```
ADD [mem1], EAX / ADD EBX, [mem2] ; 4 такта
```

```
ADD EBX, [mem2] / ADD [mem1], EAX ; 3 такта
```

В Когда две спаривающиеся инструкции требуют дополнительное время для выполнения из-за неоптимального использования кэша, невыровненности или неправильно предсказанного условного перехода, они будут выполнятся дольше, чем каждая инструкция, но меньше, чем сумма времени, требующаяся на выполнение каждой из них по отдельности.

6. Спариваемая инструкция плавающей запятой и следующая за ней FXCH повлекут несовершенное спаривание, если следующая инструкция не является инструкцией плавающей запятой.

Чтобы избежать несовершенного спаривания, вы должны знать, какие инструкции пойдут в U-конвеер, а какие - в V-конвеер. Вы можете выяснить это, просмотрев свой код и поискав инструкции, которые неспариваются, или спариваются только в определенном конвеере, или не могут спариваться в силу одного из вышеизложенных правил.

Несовершенное спаривание можно зачастую избежать, реорганизовав свои инструкции.

Пример:

```
L1:     MOV     EAX,[ESI]
        MOV     EBX,[ESI]
        INC     ECX
```

Здесь две инструкции MOV формируют несовершенную пару, потому что обе обращаются к одной и той же области в памяти, поэтому последовательность займет 3 такта. Вы можете улучшить этот код, реорганизовав инструкции так, чтобы 'INC ECX' спаривалась с одной из инструкции MOV.

```
L2:     MOV     EAX,OFFSET A
        XOR     EBX,EBX

        INC     EBX
        MOV     ECX,[EAX]
        JMP     L1
```

Пара 'INC EBX / MOV ECX,[EAX]' несовершенная, потому что следующая инструкция приводит к задержке AGI. Последовательность занимает 4 такта. Если вы вставите NOP или какую-нибудь другую инструкцию, чтобы 'MOV ECX,[EAX]' спаривался с 'JMP L1', последовательность займет только три такта.

Следующий пример выполняется в 16-ти битном режима, предполагается,что SP делится на 4:

```
L3:     PUSH    AX
        PUSH    BX
        PUSH    CX

        PUSH    DX
        CALL    FUNC
```

Инструкции PUSH формируют две несовершенные пары, потому что оба операнда в каждой паре обращаются к одному и тому же слову в памяти. 'PUSH BX' могла бы совершенно спариться с PUSH CX (потому что они находятся по разные стороны от границы, отделяющей двойные слова друг от друга), но этого не происходит, потому что она уже спарена с PUSH AX. Поэтому последовательность занимает 5 тактов. Если вы вставите NOP или другую инструкцию, чтобы 'PUSH BX' спаривалась с 'PUSH CX', а 'PUSH DX' с 'CALL FUNC', последовательность займет только 3 такта. Другой путь разрешения данной проблемы - это убедиться, что SP не кратен четырем. Правда, узнать это в 16-ти битному режиме довольно сложно, поэтому лучший выход - использовать 32-х битный режим.

Оптимизация для процессоров семейства Pentium: 12. Префиксы (PPlain и PMMX)

Автор: Агнер Фог, пер. Aquila

Дата: 2002-01-01

Инструкция с одним или более префиксами не может исполняться в V-конвеере (смотри главу 10, секцию 7).

На PPlain возникает задержка в один такт для каждого префикса, кроме 0FH в инструкциях условного ближнего перехода.

У PMMX не задержки раскодирования из-за префикса 0FH. Префиксы сегментов и повторения занимают один такт для раскодирования. PMMX может раскодировать две иснтрукции, если у первой инструкции нет префиксов или есть префикс сегмента или повторения, у второй инструкции нет префиксов. Инструкции с префиксами размера адреса или операнда на PMMX раскодировываются только отдельно. Инструкции с несколькими префиксами занимают по одному такту на каждый префикс.

Префиксы размера адреса нужно избегать, используя 32-х битный режим. Префиксы сегментов можно избежать в 32-х битном режиме, используя плоскую модель памяти. Префикс размера операнда можно избежать в 32-х битном режиме, используя только 8-ми битные и 32-х битные числа.

Когда префисы нельзя избежать, задержку раскодирования можно скрыть, если предыдущая инструкция занимает больше одного такта для исполнения. Правило для PPlain такого: каждая инструкция, которая занимает N тактов для выполнения (не для раскодирования) может 'затенять' задержку раскодирования N-1 префиксов в следующих двух (иногда трех) инструкциях или пар инструкций. Другими словами, кажды дополнительных так, который требует инструкция для выполнения, может быть использован для раскодирования одного префикса в следующей иснтрукции. Этот эффект иногда распространяется даже на правильно предсказанный переход. Любая инструкция, которая занимает больше одного такта для выполнения, и любая инструкция, чье выполнение задерживается из-за AGI, неправильного использования кэша, неправильного выравнивания или неправильного предсказания перехода, создает эффект 'затемнения'.

У PMMX есть похожий эффект 'затемнения', но он имеет другой механизм. Раскодировываемые инструкции хранятся в прозрачном "первым-вошел-первым-вышел" (FIFO) буфере, в котором может храниться до четырех инструкций. Когда буфер пуст, инструкции выполняются сразу после раскодировки. Бфер заполняется, когда инструкции раскодировываются быстрее, чем выполняются, то есть они неспарены или являются мультитактовыми. Буфер FIFO опустошается, когда иснтрукции выполняются быстрее, чем раскодировываются, то есть когда возникают задержки из-за префиксов. Буфер FIFO становится пустым после неправильно предсказанного перехода. Буфер FIFO может получить две инструкции за такт, если у второй инструкции нет префиксов и ни одна из инструкций не длинее 7 байт. Два конвеера (U и V) могуть получать по одной инструкции за такт из буфера FIFO.

Примеры:

```
CLD / REP MOVSD
```

Инструкция CLD занимает два такта, и поэтому затемняет задержку раскодировки префикса REP. Код занял бы на один такт больше, если бы инструкция CLD находилась достаточно далеко от REP MOVSD.

```
CMP DWORD PTR [EBX],0 / MOV EAX,0 / SETNZ AL
```

Инструкция CMP занимает два такта, потому что это инструкция чтения/модифицирования. Префикс 0FH инструкции SETNZ раскодировывается во время второго такта выполнения CMP, поэтому задержка раскодирования скрыта на PPlain (у PMMX нет задержки для 0FH).

Потери производительности, связанные с префиксами, на PPro, PII и PIII объясняются в главе 14.

Оптимизация для процессоров семейства Pentium: 13. Обзор конвеера PPro, PII и PIII

Автор: Агнер Фог, пер. Aquila

Дата: 2012-11-24

Архитектура микропроцессоров PPro, PII и PIII хорошо объяснена и проиллюстрирована в различных руководствах от Интела. Рекомендуется, сначала изучить этот материал, чтобы понимать, как работают эти процессоры. Я коротко объясню его структуру с упором на те элементы, которые необходимы для оптимизирования кода.

Коды доставляются из кэша кода выравненными 16-ти байтными чанками в двойной буфер, в который умещается два таких чанка. Затем код поступает из двойного буфера в декодеры в виде блоков, названными мной блоками доставки инструкций (БДИ). Обычно длина БДИ 16 байт, но он может быть не выравнен. Цель двойного буфера - сделать возможным раскодировку инструкций, которые пересекают границу в 16 байт (например адрес, кратный 16).

БДИ поступает в декодер длины инструкции, который определяет, где начинается каждая из инструкций и где заканчивается, а затем попадает в декодеры инструкций. Есть три декодера, поэтому вы можете декодировать до трех инструкций за такт. Группа из трех инструкций, декодируемых за один и тот же такт, называется декодируемой группой.

Декодеры переводят инструкции в микрооперации (сокращенно "мопы"). Простые инструкции генерируют только один моп, в то время, как более сложные инструкции могут генерировать несколько мопов. Например, инструкция 'ADD EAX,[MEM]' декодируется в два мопа, один, считывающий исходный операнд из памяти, а другой, который выполняет сложение. Целью разбития инструкций на мопы является сделать обработку инструкций более эффективной.

Три декодера называются D0, D1 и D2. D0 может обрабатывать все инструкции, в то время как D1 и D2 может обрабатывать только простые инструкции, генерирующие только один моп.

Мопы из декодеров через короткую очередь поступают в таблицу распределения регистров (RAT). Выполнение мопов осуществляется на временных регистрах, после чего результат записывается в постоянные регистры EAX, EBX и так далее. Целью RAT является указание мопам, какие временные регистры использовать и позволить переименование регистров (смотри далее).

После RAT мопы поступают в буфер перегруппировки (ROB). Целью ROB является неупорядоченное выполнение. Мопы остаются в области рекреации (reservation station), пока не станут доступны операнды, которые им необходимы. Если операнд для уопа задерживается из-за того, что генерирующий его предыдущий моп еще не закончил свою работу, тогда ROB может найти дургой моп в очереди, чтобы сэкономить время.

Готовые к выполнению мопы посылаются в модули выполнения (execution units), которые сгруппированы вокруг пяти портов: порт 0 и 1 могут обрабатывать все арифметические операции, переходы и так далее. Порт 2 берет на себя операции считывания из памяти, порт 3 высчитывает адреса для записи в память, а порт 4 выполняет эту запись.

После того, как инструкция была выполнена, она помечается в ROB как готовая к удалению, после чего поступает в область удаления (retirement station). Здесь содержимое временных регистров, использованных мопами, записывается в постоянные регистры. Хотя мопы могут запускаться не по порядку, последний должен быть восстановлен при удалении.

Оптимизация для процессоров семейства Pentium: 14. Раскодировка инструкций (PPro, PII и PIII)

Автор: Агнер Фог, пер. Aquila

Дата: 2002-01-01

Я рассказываю о раскодировке инструкций до доставки инструкций, потому что вам необходимо знать, как работают раскодировщики, чтобы понимать возможные способы доставки.

Декодеры могут обрабатывать три инструкции за такт, но только, если соблюдены определенные условия. Декодер D0 может обработать за один такт любую инструкцию, которая генерирует до 4 мопов. Декодеры D1 и D2 могут обрабатывать только те инструкции, которые генерируют 1 моп и эти инструкции не могут занимать больше 8 байт.

Резюмируем правила для декодирования двух или трех инструкций за один такт:

• Первая инструкци (D0) не должна генерировать больше 4-х мопов.
• Вторая и третья инструкции не должны генерировать больше, чем по одному мопу.
• Вторая и третья инструкции не могут быть занимать больше 8-ми байтов каждая.
• Инструкции должны содержаться внутри одного 16-байтного БДИ (смотри следующую главу).

Нет ограничение на длину инструкции в D0 (несмотря на то, то руководства от Интел говорят об обратном) пока все три инструкции влезают в один 16-ти байтный блок.

Инструкция, которая генерирует больше 4-х мопов требует два или больше такта для раскодировки, и ни одна другая инструкция не может раскодировываться параллельно.

Из вышеприведенных правил следует, что декодеры могут генерировать максимум шесть мопов за такт, если первая инструкция в каждой раскодировываемой группе разбивается на 4 мопа, а другие две - на один каждая. Минимальное количество - это два мопа за такт, еси все инструкции генерируют по два мопа, так что D1 и D2 никогда не используются.

Для максимальной производительности рекомендуется перегруппировать ваши инструкции в блоки 4-1-1: инструкции, которые генерирует 2 или 4 мопа можно разбить на две простые одномопные инструкции, что не будет вам стоить ни такта.

Пример:

```
MOV     EBX, [MEM1]     ; 1 uop  (D0)
INC     EBX             ; 1 uop  (D1)
ADD     EAX, [MEM2]     ; 2 uops (D0)
ADD     [MEM3], EAX     ; 4 uops (D0)
```

Это занимает три такта для раскодировки. Вы можете сохранить один такт, перегруппировав инструкции в две декодируемые группы:

```
ADD     EAX, [MEM2]     ; 2 uops (D0)
MOV     EBX, [MEM1]     ; 1 uop  (D1)
INC     EBX             ; 1 uop  (D2)
ADD     [MEM3], EAX     ; 4 uops (D0)
```

Теперь декодеры генерируют 8 мопов за два такта, что удовлетворительно. На

более поздних стадиях конвеер может обрабатывать только 3 мопа за такт, поэтому при скорости декодирования высшей, чем эта, вы можете считать, что декодирование не является узким местом. Тем не менее, сложности в механизме доставки могут задержать раскодирование так, как это рассказывается в следующей главе, поэтому чтобы быть спокойным, лучше стремиться к скорости раскодировки большей 3-х мопов за такт.

Вы можете узнать количество генерируемых инструкцией мопов в таблице в главе 29.

Префиксы инструкций также приводят к потере скорости раскодировки. У инструкций могут быть префиксы следующих видов:

• Префикс размера операнда требуется, когда вы используете 16-ти битный операнд в 32-х битном окружении или наоборот. (Не считая инструкций, у которых операнды могут быть только одного размера, например FNSTSW AX). Префикс размера операнда вызывает потерю нескольких тактов, если у инструкции есть числовой 16-ти или 32-х битный операнд, потому что размер операнда меняется префиксом.

Примеры:

```
ADD BX, 9      ; no penalty because immediate operand is 8 bits
MOV WORD PTR [MEM16], 9  ; penalty because operand is 16 bits

ADD BX, 9      ; нет потерь, так как числовой операнд занимает 8 бит
MOV WORD PTR [MEM16], 9  ; есть потери, так как операнд занимает 16 бит
```

Последную инструкцию следует заменить на

```
MOV EAX, 9
MOV WORD PTR [MEM16], AX  ; no penalty because no immediate

MOV EAX, 9
MOV WORD PTR [MEM16], AX  ; нет потерь, так как нет числовых операндов
```

• Префикс размера адреса используется при 32-х битной адресации в 16-ти битном режиме или наоборот. Это редко требуется и этого следует избегать. Префикс размера адреса вызывает потери каждый раз, когда операнды используются явно, потому что интерпретация изменяется с помощью префикса. Инструкции, использующие оеранды памяти неявно (например строковые операции), не приводят к потерям, связанных с префиксом размера операнда.

• Префиксы сегментов используются, когда вы обращаетесь к данным, находящимся не в сегменте данных по умолчанию. На PPro, PII и PIII префиксы сегментов не приводят к потерям.

• Префиксы повторения и префиксы закрытия (lock prefixes) не приводят к потерям при декодировании.

• Всегда есть потери, когда у вас больше одного префикса. Обычно уходит по одному такту на префикс.

Оптимизация для процессоров семейства Pentium: 15. Доставка инструкций (PPro, PII и PIII)

Автор: Агнер Фог, пер. Aquila

Дата: 2002-01-01

Код доставляется в двойной буфер из кэша кода чанками по 16 байт. Двойной буфер называется так, потому что он содержит два таких чанка. Затем код берется из двойного буфера и скармливается декодерам поблочно (каждый блок обычно 16 байтов длиной, но необязательно, он может быть и не выравнен по этой границе). Я называю эти блоки БДИ (блоки доставки инструкций). Если БДИ пересекает границу 16 байт в коде, его нужно изъять из обоих чанко и поместить в буфер. Таким образом, двойной буфер нужен для того, чтобы позволить доставку инструкций, пересекающих барьер в 16 байт.

Двойной буфер может доставить один 16-ти байтный чанк за такт и может сгенерировать один БДИ за это же время. БДИ обычно 16-ти байтов длиной, но могут быть короче, если есть предсказанный переход в блоке. (Смотри главу 22 о предсказании переходов).

К сожалению, двойной буфер недостаточно велик, чтобы обрабатывать инструкции связанные с переходами без задержек. Если БДИ, который содержит инструкцию перехода, пересекает 16-ти байтную границу, двойному буферу требуется держать два последовательных 16-ти байтных чанка, чтобы сгенерировать его. Если первая инструкция после перехода пересекает 16-ти байтную границу, тогда двойной буфер должен загрузить два новых 16-ти битных чанка кода, прежде чем будет сгенерировать БДИ. Это ознаает, что в худшем случае раскодировка первого инструкции после перехода может быть задержана на два цикла. У вас происходят потери из-за 16-ти байтных границ в БДИ, содержащем переход, и из-за пересечения 16-ти байтной границы после перехода. Вы можете получить выигрыш в производительности, если у вас больше, чем одна раскодировываемая группа в БДИ, которая содержит переход, потому что это дает двойному буферу дополнительное время для доставки одного или двух чанков кода, следующих за переходом. Подобные выигрыши могут компенсировать потери согласно таблице ниже. Если двойной буфер доставляет только один 16-ти байтный чан после перехода, тогда первый БДИ после перехода будет идентичен этом чанку, то есть выравнен по 16-ти байтной границе. Другими словами, первый БДИ после перехода не будет начинаться с первой инструкции, но с ближайшего предшествующего адреса, кратного 16. Если у двойного буфера есть время, чтобы загрузит два чанка, тогда новый БДИ может пересечь 16-ти байтную границу и начаться с первой инструкции после перехода. Эти правила кратко прорезюмированы в следующей таблице:

Количество декодируемых груп в БДИ, содержащем переход	16-байтная граница в этом БДИ	16-байтная граница в первой инструкции после перехода	задержка декодера	выравнивание первого БДИ после перехода
1	0	0	0	на 16
1	0	1	1	к инструкции
1	1	0	1	на 16
1	1	1	2	к инструкции
2	0	0	0	к инструкции
2	0	1	0	к инструкции

2	1	0	0	на 16
2	1	1	1	к инструкции
3 или больше	0	0	0	к инструкции
3 или больше	0	1	0	к инструкции
3 или больше	1	0	0	к инструкции
3 или больше	1	1	0	к инструкции

Переходы задерживают доставку, поэтому цикл всегда занимает на два такта больше за выполнение, чем количество 16-ти байтных границ в цикле.

Следующая проблема с механизмом доставки инструкций заключается в том, что новый БДИ не сгененирируется, пока предыдущий не будет полностью отработан. Каждый БДИ может содержать несколько раскодировываемых групп. Если БДИ длиной 16 байт заканчивается незавершенной инструкцией, тогда следующий БДИ начнется в начале этой инструкции. Первая инструкция в БДИ всегда идет в D0, а следующие две инструкции направляются в D1 и D2, если это возможно. Как следствие, D1 и D2 используются не совсем оптимально. Если код структурирован согласно правилу 4-1-1, а инструкция, которая, как предполагалось, должна направиться в D1 или D2, оказывается первой инструкцией в БДИ, тогда она попадает в D0, что ведет к потере одного такта. Вероятно, это недостаток архитектуры процессора. Из-за этого время, которое займет раскодировка определенного кода может зависеть от того, где начнется первый БДИ.

Если скорость раскодирования инструкций критична, и вы хотите избежать этой проблемы, вам нужно знать, где начинается каждый БДИ. Это довольно нудная работа. Вначале вам разделить ваш сегмент на параграфы, чтобы знать, где находятся 16-ти байтные границы. Затем вам нужно взглянуть на ассемблерный листинг, чтобы увидеть, какова длина каждой инструкции. (Рекомендуется, изучить, как кодируются инструкции, чтобы уметь предсказывать их длину.) Если вы знаете, где начинается один БДИ, тогда вы можете найти, где начинается другой следующим образом: сделать блок 16 байтов длиной. Если он кончается на границе между инструкциями, значить следующий БДИ начнется здесь. Если он включает в себя часть инструкции, тогда следующий БДИ начнется с этой инструкции. (Здесь нужно подсчитывать только длины инструкций, не имеет значения, сколько мопов они генерируют.) Таким образом вы можете обработать весь код и отметить, где начинается каждый из БДИ-блоков. Единственная проблема - это узнать, где находится первый БДИ. Вот несколько подсказок.

• Первый БДИ после перехода, вызова или возвращения может начинаться либо с первой, либо с ближайшей предшествующей 16-ти байтной границе, согласно вышеприведенной таблице. Если вы выравняете первую инструкцию так, чтобы она начиналась с 16-байтной границы, вы можете быть уверены, что первый БДИ начнется здесь. Вы можете выравнять подобным образом важные процедур и циклы, чтобы убыстрить работу программы.

• Если комбинированная длина двух последовательных инструкций больше 16 байтов, вы можете быть уверены, что вторая не влезет в тот же БДИ, что и первая, следовательно вы можете быть увенны, что вторая инструкция будет первой в БДИ. Вы можете использовать ее в качестве стартовой точки для того, чтобы найти где начинаются следующие БДИ.

• Первый БДИ после неправильного предсказания перехода начинается по 16-ти байтной границе. Как объясняется в главе 22.2, цикл, который повторяется больше 5 раз, всегда будет приводить к неправильному предсказанию перехода при выходе. Первый БДИ после такого цикла будет начинаться по ближайшей предшествующей 16-ти байтной границе.

• Есть также другие события, приводящие к подобному эффекту, например прерывания, исключения, самомодифирующийся код и такие инструкции как CPUID, IN и OUT.

Я уверен, что теперь вы хотите получить пример:

```
адрес        инструкция              длина     мопы   ожидаемый декодер

----------------------------------------------------------------------
1000h        MOV ECX, 1000             5         1       D0
1005h   LL:  MOV [ESI], EAX            2         2       D0
1007h        MOV [MEM], 0             10         2       D0
1011h        LEA EBX, [EAX+200]        6         1       D1
1017h        MOV BYTE PTR [ESI], 0     3         2       D0
101Ah        BSR EDX, EAX              3         2       D0
101Dh        MOV BYTE PTR [ESI+1],0    4         2       D0
1021h        DEC ECX                   1         1       D1
1022h        JNZ LL                    2         1       D2
```

Давайте предположим, что первый БДИ начинается по адресу 1000h и заканчивается в 1010h. Это до завершения инструкции 'MOV [MEM], 0', поэтому следующий БДИ начнется в 1007h и закончится в 1017h. Это граница между инструкциями, поэтому третий БДИ начнется в 1017h и захватит весь остаток цикла. Количество тактов, которое уйдет на раскодировку - это количество инструкций, попадающих в D0. Всего их в цикле 5. Последний БДИ содержит три раскодировываемых блоков, включая последние пять инструкций, и одну 16-ти байтную границу (1020h). Если мы еще раз посмотрим на таблицу, то увидим, что первый БДИ после перехода будет начинаться с первой инструкции после перехода (та, где стоит метка LL) и заканчиваться у 1015h. Это перед концом инструкции LEA, поэтому следующий БДИ будет начинаться с 1011h до 1021h, а последний будет идти от 1021h до самого конца. Теперь LEA и DEC попадают в начало БДИ, поэтому они обе направляются в D0. У нас есть 7 инструкции в D0 и на раскодироваку цикла во втором повторении уходит 7 тактов. Последний БДИ содержит только одну раскодировываемую группу (DEC ECX / JNZ LL) и у него нет 16-ти байтной границы. Согласно таблице, следующий БДИ после перехода начнется с 16-ти байтной границы, то есть 1000h. Мы оказываемся в той же ситуации, что и в первом повторении, и вы увидите, что раскодировка цикла занимает соответственно 5 и 7 тактов. Так как других узких мест нет, на выполнение цикла 1000 раз уйдет 6000 тактов. Если бы стартовый адрес был другим, и в первой или последней инструкции была бы 16-ти байтная граница, то это бы заняло 8000 тактов. Если вы перегруппируете цикл, так чтобы инструкции для D1 или D2 не попадали в начало БДИ, тогда выполнение цикла заняло бы еще меньше - 5000 тактов.

Вышеприведенный пример был преднамеренно построен так, что единственным узким местом является доставка инструкций и их раскодировка. Самый легкий путь избежать этой проблемы - это структурировать ваш код, чтобы он генерировал больше 3 мопов за такт, чтобы раскодировка не была узким местом несмотря на приведенные потери скорости. В маленьких циклах это может быть невозможно, поэтому вам нужно найти путь, чтобы оптимизировать доставку инструкций и раскодировку.

Например вы можете изменить стартовый адрес вашей процедуры, чтобы избежать 16-ти байтных границ, когда они вам не нужны. Помните, ваш сегмент кода должен быть выравнен по параграфу, чтобы вы знали, где находятся границы.

Если вы вставите директиву 'ALIGN 16' до начала цикла, тогда ассемблер поместит NOP и другие подобные инструкции, чтобы выравнять код. Большинство

ассемблеров используют инструкцию 'XCNG EBX, EBX' в качестве двухбайтного заполнителя (иногда ее называют "двухбайтным NOP'ом"). Кому бы ни пришла в голову эта идея, лучше данную инструкцию не использовать, потому что она занимает больше времени, чем два NOP'а на большинстве процессоров. Если цикл выполняется много раз, тогда то, что находится за его пределами неважно, если говорить о скорости и вы не должны заботиться о том, какую инструкцию использовать в качестве заполнителя. Но если время, занимаемое этими инструкциями играет роль, тогда вы можете выбрать инструкцию-заполнитель самостоятельно. Вы также можете использовать инструкции, которые делают что-нибудь полезное, например обновляют регистр, дабы избежать задержек при чтении регистра. Например, если вы используете регистр EBP для адресации, но редко пишете в его, вы можете использовать 'MOV EBP,EBP' или 'ADD EBP, 0' в качестве заполнителя, чтобы снизить возможность вышеупомянутых задержек. Если вам не делать ничего подобного, вы можете использовать FXCH ST(0), потому что она не создает никакой нагрузки на порты выполнения, при условии, что ST(0) содержит верное значение с плавающей запятой.

Еще одним лекарством может стать перегруппировка ваших инструкций таким образом, что границы между БДИ не приносили вреда. Это может стать сложной головоломкой и найти приемлимое решение не всегда возможно.

Другая возможность - манипуляции с длинами инструкций. Иногда вы можете заменить одну инструкцию другой с иной длиной. Многие инструкции можно закодировать различными вариантами с разной длиной. Ассемблер всегда выбирает наиболее короткую версию инструкции, но все можно закодировать более длинную версию. Например 'DEC ECX' занимает один байт длиной, 'SUB ECX, 1' - тир байта, и вы можете закодировать 6-ти байтовую версию, используя чиловой параметр размером в двойное слово, используя этот трюк:

```
SUB ECX, 9999
ORG $-4
DD 1
```

Инструкции с операндами в памяти можно сделать на оди байт длинее с помощью SIB-байта, но самым легким путем сделать инструкцию на один байт длинее является добавление сегментного префикса DS (DB 3Eh). Микропроцессоры обычно принимают избытычные и ничего не значащие префиксы (не считая LOCK), если длина инструкции не превышает 15-ти байт. Даже инструкции без операндов памяти могут иметь сегментный префикс. Поэтому если вы хотите, чтобы инструкция 'DEC ECX' была два байта длиной, напишите:

```
DB  3Eh
DEC ECX
```

Помните, что при этом у вас будут потери при раскодировке, если у инструкции больше одного префикса. Вомзожно, что инструкции с ничего не значащими префиксами, особенно префиксами повторения и закрытия, будут использоваться в будущих процессорах для новых инструкций, когда не останется свободных кодов, но полагаю, что использование сегментных префиксов с уже существующими инструкциями вполне безопасно.

Оптимизация для процессоров семейства Pentium: 16. Переименование регистров (PPro, PII и PIII)

Автор: Агнер Фог, пер. Aquila

Дата: 2002-01-01

16.1 Уничтожение зависимостей

Переименование регистров - это продвинутая техника, используемая этими микропроцессорами, чтобы убрать зависимости между различными частями кода. Пример:

```
MOV EAX, [MEM1]
IMUL EAX, 6
MOV [MEM2], EAX
MOV EAX, [MEM3]
INC EAX
MOV [MEM4], EAX
```

Здесь последние три инструкции независимы от трех первых в том смысле, что им не требуется результат, полученный после их выполнения. Чтобы оптимизировать этот код на ранних процессорах вы были должны использовать другой регистр, отличный от EAX, в последних трех инструкциях и перегруппировать инструкции так, чтобы последние три выполнялись параллельно с первыми тремя. PPro, PII и PIII делают это автоматически. Они назначают новый временный регистр для EAX каждый раз, когда вы пишете в него. Таким образом инструкции 'MOV EAX, [MEM3]' становятся независимыми от предшествующих инструкций. С помощью выполнения не по порядку 'MOV [MEM4], EAX' может выполниться до того окончания обработки медленной инструкции IMUL.

Переименование регистров происходит полностью автоматически. Новый временный регистр назначается как псевдоним постоянному регистру каждый раз, когда инструкция пишет в этот регистр. Например инструкция 'INC EAX' использует один временный регистр для ввода и другой временный регистр для вывода. Это, разумеется, не убирает зависимости, но имеет некоторое значение для последующих чтений из регистра, о чем я расскажу позже.

Все регистры общего назначения, указатель на стек, флаги, регистры плавающей запятой, регистры MMX, регистры XMM и сегментные регистры могут быть переименованы. Контрольные слова и слово статуса плавающей запятой не могут быть переименованы, поэтому навряд ли у вас кончаться временные регистры.

Общей практикой установки значения регистра в ноль является 'XOR EAX,EAX' или 'SUB EAX, EAX'. Эти инструкции не распознаются как независимые от предыдущего значения регистра. Если вы хотите убрать зависимость от медленных предшествующих инструкций, используйте 'MOV EAX, 0'.

Переименование регистров контролируется таблицей псевдонимов регистров (RAT) и буфером перегруппировки (ROB). Мопы из декодеров поступают в RAT через очередь, затем в ROB, а после чего в резервационную станцию (reservation station). RAT может обрабатывать только 3 мопа за такт. Это означает, что суммарная производительность процессора не может превышать 3 мопа за такт.

Практически нет никаких ограничений на количество переименований. RAT может переименовывать три регистра за такт, и он может даже переименовать один и тот же регистр три раза за один такт.

16.2 Задержки чтения регистров.

Но существует другое ограничение, которое может быть весьма серьеным, и это то, что за один такт вы можете читать только из двух постоянных регистров. Это ограничение относится ко всем регистрам, используемых инструкциям, не считая тех регистров, в которые инструкции только пишут. Пример:

```
MOV [EDI + ESI], EAX
MOV EBX, [ESP + EBP]
```

Первая инструкция генерирует два мопа: один считывает EAX, а другой - EDI и ESI. Вторая инструкция генерирует один моп, который читает ESP и EBP. EBX не учитывается, поскольку инструкция только пишет в него. Давайте предположим, что эти три мопа идут вместе через RAT. Я буду использовать слово триплет для группы из трех последовательных мопов, которые идут вместе через RAT. Так как ROB может обрабатывать только два чтения из постоянных регистров за такт, а нам нужно пять чтений, то наш триплет будет задержен на два дополнительных такта, прежде чем он попадет в резервационную станцию. При трех или четырех чтений из постоянных регистров он был бы задержен на один такт.

Из одного регистра в одном триплете можно читать больше одного раза без ущерба для качества. Если вышеприведенные инструкции поменять на:

```
MOV [EDI + ESI], EDI
MOV EBX, [EDI + EDI]
```

тогда произойдет только два чтения из регистров (EDI и ESI) и триплет не будет задержен.

Регистр, в который будет произведена запись текущим мопом, сохраняется в ROB, поэтому из него можно свободно читать, пока он не будет выгружен оттуда, что занимает по меньшей мере три такта, а обычно еще больше. Выгрузка из ROB является финальной стадией выполнения, когда значение становится доступным. Другими словами, вы можете читать любое количество раз из регистра в RAT без задержек, если их значение еще не стало доступным у модулей выполнения, поэтому вы можете быть уверены, что из регистра, в который была произведена в одном триплете, можно свободно читать как минимум в трех последующих. Если выгрузка (writeback) была задержена перегруппировкой, медленными инструкциями, цепочками зависимости, задержкой кэша или по какой-то другой причине, тогда из регистра можно свободно читать еще некоторое время.

Пример:

```
MOV EAX, EBX
SUB ECX, EAX
INC EBX
MOV EDX, [EAX]
ADD ESI, EBX
ADD EDI, ECX
```

Эти шесть инструкций генерирую 1 моп каждая. Давайте предположим, что первые три мопа идут через RAT вместе. Эти три мопа читают регистры EBX, ECX и EAX. Но так как мы пишем в EAX до того, как начали из него читать, чтение прозводится "бесплатно" и у нас нет никаких задержек. Следующие три мопа читают EAX, ESI, EBX, EDI и ECX. Так как оба EAX, EBX и ECX были изменены предыдущим триплетом и не были еще выгружены, из них можно свободно читать, поэтому учитываются только ESI и EDI, и у нас нет задержек и во втором триплете. Если инструкцию 'SUB ECX, EAX' в первом триплете заменить на

'CMP ECX, EAX', тогда в ECX не производится запись, и у нас происходит задержка во втором триплете при чтении ESI, EDI и ECX. Подобным же образом, если инструкция 'INC EBX' в первом триплете поменять на NOP или что-нибудь вроде этого, тогда у нас будет задержка во втором триплете при чтении ESI, EBX и EDI.

Ни один моп не может читать больше, чем из двух регистров. Поэтому все инструкции, читающие больше, чем из двух регистров, разбиваются на два или больше мопа.

Чтобы подсчитать количество прочитываемых регистров, вам нужно включить все регистры, которые считываются инструкцией. В это число входят все целочисленные регистры, флаговые регистры, указатель на стек, регистры плавающей запятой и регистры MMX. Регистр XMM идет за два, кроме тех случаев, когда используется только его чать, например в ADDSS и MOVHLPS. Сегментные регистр и указатель на инструкцию не учитываются. Например в 'SETZ AL' вы считаете флаговый регистр, но не AL. В 'ADD EBX, ECX' считаются и EBX и ECX, но не регистр флагов, потому что в него только производится запись. 'PUSH EAX' читает EAX и указатель на стек, а потом пишет в последний.

Инструкция FXCH является особым случаем. Она работает с помощью переименования, но не считывает никаких значений, поэтому она не попадает под действие какого-либо правила о задержке чтения из регстра. Инструкция FXCH ведет себя как один моп, который ни читает, ни пишет в регистр, когда дело касается правил задержек чтения из регистра.

Не путайте триплеты мопов с раскодировываемыми группами. Последняя может генерировать от одного до шести мопа, и даеж если раскодировываемая группа имеет три инструкции и генерирует три мопа, нет никакой гарантии, что эти три мопа попадут в RAT вместе.

Очередь между декодерами и RAT так которотка (10 мопов), что вы не можете предположить, что задержки чтения регистров не оказывают влияния на декодеры, или какие-то изменения в выводе декодеров не задерживают RAT.

Очень трудно предсказать, какие мопы пойдут через RAT вместе, если очередь не пуста, и для оптимизированного кода очередь должна быть пуста только после неправильно предсказанного прехода. Несколько мопов, генерируемых одной инструкцией не обязательно пойдут через RAT вместе; мопы просто берутся последовательно из очереди по три за раз. Последовательность не нарушается предсказанным переходом: мопы до и после перехода могут пойти через RAT вместе. Только неправильно предсказанный переход сбросит очередь и начнет все сначала, поэтому три следующих мопа точно попадут в RAT вместе.

Если три последовательных мопа читают больше, чем из двух регистров, то вы бы, конечно, предпочли, чтобы они не шли вместе друг с другом в RAT. Вероятность того, что они пойдут вместе - одна третья. Задержка чтения трех или четырех выгруженных регистра в одной тройке мопов - один такт. Вы можете считать задержку в один такт эквивалентом загрузки еще трех мопов в RAT. С вероятностью в 1/3 что три мопа пойдут в RAT вместе, средняя задержка будет эквивалентна 3/3 = 1 моп. Чтобы посчитать следнее время, которое займет для некоторого кода проход через RAT, добавьте количество потенциальных задержек чтения регистров к количеству мопов и поделите на три. Вы можете видеть, что нет смысла убирать задержки с помощью добавления дополнительных регистров, пока вы точно не уверены, какие мопы пойдут в RAT вместе, или же вы можете

предотвратить больше, чем одну задержку чтения регистра с помощью одной дополнительной инструкции.

В ситуациях, когда вашей целью является производительность 3 мопа за такт, ограничение в чтении только двух постоянных регистров за такт может стать узким местом, которое будет трудно обойти. Возможные пути обойти возможные задержки чтения регистра следующие:

• Держать мопы, которые читают один и тот же регистр близко друг к другу, чтобы они попали в один триплет.
• Держать мопы, которые читают разные регистры подальше друг от друга, чтобы они не могли попасть в один триплет.
• Располагать мопы, которые читают регистр не дальше трех-четырех триплетов от инструкции, которая писала или изменяла этот регистр, чтобы он не был выгружен прежде, чем будут произведены все необходимые операции чтения (не важно, если у вас есть переход, если он будет правильно предсказан). Если у вас есть основания полагать, что запись в регистр будет задержена, вы можете спокойно читать из него еще некоторое количество инструкций.
• Использовать абсолютные адреса вместо указателей, чтобы снизить количество читаемых регистров.
• Вы можете переименовать регистр в триплете, если это не вызывает задержку, чтобы предотвратить задержку чтения для этого регистра в одном или больше следующих далее триплетов. Пример: 'MOV ESP,ESP / ... / MOV EAX,[ESP+8]'. Этот метод стоит дополнительный моп, поэтому его стоит применять только, если среднее предполагаемое количество предотвращенных задержек чтения больше 1/3.

Для инструкций, которые генерируют больше одного мопа, вы можете захотить узнать порядок мопов, генерируемых инструкцией, чтобы сделать предсказание возможных задержек чтения регистра более точным. Ниже я перечислил наиболее общие случаи.

Запись в память

Запись в память генерирует два мопа. Первый (в порт 4) - это операция сохранения, считывающая регистр, который нужно сохранить, второй моп вычисляет адрес памяти, читая любые регистры-указатели.

Примеры:

```
MOV [EDI], EAX
```

Первый моп читает EAX, второй читает EDI.

```
FSTP QWORD PTR [EBX+8*ECX]
```

Первый моп читает ST(0), второй моп читает EBX и ECX.

Чтение и модифицирование

Инструкция, которая читает операнд в памяти и модифицирует регистр с помощью какой-либо арифметической или логической операции генерирует два мопа. Первый (порт 2) - это инструкция загрузки из памяти, читающая любой регистр-указатель, второй моп - это арифметическая инструкция (порт 1 или 2), читающая и пишущая в регистр назначения и, возможно, пишущая в регистр флагов.

Пример:

```
ADD EAX, [ESI+20]
```

Первый моп читает ESI, второй моп читает EAX и пишет EAX и регистр флагов.

Чтение/модифицирование/запись

Инструкция этого рода генерирует четыре мопа. Первый моп (порт 2) читает регистр-указатель, второй (порт 0 или 1) читает и производит запись в исходный регистр и, возможно, пишет в регистр флагов, третий моп (порт 4) считывает только временный результат, которые не учитывается здесь, четвертый (порт 3) читает все регистры-указатели снова. Так как первый и четвертый регистр не могут идти в RAT вместе, вы не получить преимущество от того, что они читают один и тот же регистр-указатель.

Пример:

```
OR [ESI+EDI], EAX
```

Первый моп читает ESI и EDI, второй читает EAX и пишет в EAX и в регистр флагов, третий читает только временный результат, четвертый читает ESI и EDI снова. Вне зависимости о того, в каком порядке эти мопы пойдут в RAT, вы можете быть уверены, что моп, которы читает EAX пойдет вместе с тем, который читает ESI и EDI. Поэтому задержка чтения регистра неизбежна в этой инструкции, если только один из регистров не был изменен ранее.

Сохранение регистра в стек.

Сохранение регистра в стек генерирует 3 мопа. Первый (порт 4) - это инструкция сохранения, чтение регистра. Второй моп (порт 4) генерирует адрес, считывая указатель на стек. Третий (порт 0 или 1) вычитает размер слова из указателя на стек, читая и модифицируя его.

Обратная операция генерирует два мопа. Первый (порт 2) загружает значение, считывая указатель на стек и записывая значение в регистр. Второй моп (порт 0 или 1) корректирует указатель на стек, читая и модифицируя его.

Вызов

Ближний вызов генерирует 4 мопа (порты 1, 4, 3, 01). Первый два мопа читают указатель на инструкцию (EIP), который не учитывается, потому что он не может быть переименован. Третий моп читает указатель на стек. Последний моп читает и модифицирует его.

Возврат

Ближний возврат генерирует 4 мопа (порт 2, 01, 01, 1). Первый моп считывает ESP. Третий читает и модифицирует его.

Пример того, как избежать задержку чтения регистра можно найти в примере 2.6.

Оптимизация для процессоров семейства Pentium: 17. Выполнение кода не по порядку (PPro, PII и PIII)

Автор: Агнер Фог, пер. Aquila

Дата: 2012-11-24

Буфер перегруппировки вмещает 40 мопов. Каждый моп ждет в ROB, пока все операнды не будут готовы и не появится свободный модуль для их выполнения. Это делает возможным выполнение кода не по порядку. Если одна часть кода задерживается из-за загрузки в кэш, это не повлияет на выполнение других частей кода, если они не зависят от первой части.

Запись в память не может быть выполнена по порядку, если она зависит от других записей. Есть четыре буфера записи, поэтому вы можете ожидать большие потери производительности во время такой записи в память или записи в некэшированную память, поэтому рекомендуется, чтобы вы делали четыре записи за раз и были уверены в том, что у процессора есть чем еще заняться, прежде чем вы нагрузите его еще четырьмя записями. Чтение из памяти и другие инструкции могут выполняться не по порядку, кроме IN, OUT и синхронизирующих операций.

Если ваш код пишет в память по какому-то адресу, а вскоре считывает оттуда же, тогда чтение по ошибке может быть произведено до записи, потому ROB не имеет понятия об адресах во время перегруппировки. Эта ошибка отлавливается, когда подсчитывается адрес памяти, поэтому операция чтения в таком случае выполняется снова. Потери при этом составляют 3 такта. Единственный путь избежать этого - это убедиться, что у модуля выполнения будет занят делом между последовательными записью в память и чтением из нее по тому же адресу.

Есть несколько модулей выполнения, сгруппированных вокруг пяти портов. Порт 0 и 1 - для арифметических операций. Простые пересылки, арифметические и логические операции попадают в порт 0 или 1, в зависимости от того, какой из них свободен. Порт 0 также обрабатывает умножение, деление, целочисленные побитовые сдвиги и операции с плавающей запятой. Порт 1 в дополнение к основным функциям занимается переходами и некоторыми операциями MMX и XMM. Порт 2 обрабатывает все чтения из памяти и несколько строковых операций. В главе 29 вы найдете полный список мопов, генерируемых инструкциями кода с указанием того, в какой порт они попадают. Заметьте, что все операции записи в память требуют два мопа, один для порта 3, а другой для порта 4, в то время как операции чтения из памяти требуют только один моп (порт 2).

В большинстве случаев каждый порт может получать один моп за такт. Это означает, что вы можете выполнять до 5 операций мопов за такт, если они попадают в пять разных портов, но так как есть ограничение на 3 мопа за такт, установленное ранее в конвеере, вам никогда не удастся выполнить больше 3 мопов за такт в среднем.

Вы должны убедиться, что никакой порт не получает больше одной трети мопов, если вы хотите достичь производительности в 3 мопа за такт. Используйте таблицу мопов в главе 29 и подсчитайте, сколько мопов попадет в каждый порт. Если порты 0 и 1 перегружены, в то время как порт 2 свободен, вы можете улучшить свой код, заменив некоторые пересылки вида "регистр, регистр" или "регистр, числовое значение" на пересылку вида "регистр, память", чтобы перенести часть нагрузки с портов 0 и 1 на порт 2.

Большинство мопов занимают один такт для выполнения, но умножения, деления и многие операции с плавающей запятой занимают больше:

Сложения и вычитания плавающей запятой занимают 3 такта, но модуль выполнения полностью конвееризован, поэтому он может принимать новые FADD и FSUB каждый такт, пока предыдущие инструкции выполняются (конечно, если они независимы друг от друга).

Целочисленное умножение занимает 4 такта, умножения плавающей запятой - 5, а умножения MMX - 3 такта. Умножения целых чисел и MMX конвееризованы, поэтому они могут получать новые инструкции каждый такт. Умножения плавающей запятой частично конвееризованы: модуль выполнения может получать новую инструкцию FMUL через два такта после получения предыдущей, поэтому максимальная производительность будет равная одному FMUL за каждые два такта. "Дыры" между FMUL'ами нельзя заполнить целочисленными умножениями, так как они используют ту же схему. Сложения и умножения XMM занимают 3 и 4 такта соотвественно и полностью конвееризованы. Но так как каждый регистр XMM представлен в виде двух 64-х битных регистров, упакованная инструкция XMM будет состоять из двух мопов, а производительность будет равна одной арифметической инструкции XMM каждые два такта. Инструкции сложения и умножения XMM могут выполняться параллельно, потому что оне не используют одни и те же порты.

Деление целых чисел и чисел с плавающей запятой занимает до 39 тактов и не конвееризовано. Это означает, что модуль выполнения не может начать новое деление, пока предыдущие не будет выполнено. Вышесказанное относится к квадратному корню и трансцендентальным функциям.

Также инструкции переходов, вызовов и возвратов не полностью конвееризованы. Вы не можете выполнить новый переход в первом же такте после предыдущего перехода. Поэтому максимальная производительность для переходов, вызовов и возвратов равна одному за каждые два такта.

Конечно вы должны избегать инструкций, которые генерируют много мопов. Например инструкцию LOOP XX нужно заменить на DEC ECX / JNZ XX.

Если у вас есть две последовательные POP-инструкции, вы можете заменить их на другие, чтобы снизить количество мопов.

```
POP ECX / POP EBX / POP EAX     ; можно заменить на:
MOV ECX,[ESP] / MOV EBX,[ESP+4] / MOV EAX,[ESP] / ADD ESP,12
```

Первый код генерирует 6 мопов, последующий - 4 и декодируется быстрее. Делать то же самое с инструкциями PUSH менее выгодно, потому разбитый код будет генерировать задержки чтения регистров, если только у вас нет других инструкций, которые можно вставить между ними или какие-то регистры не были переименованы раньше. Делать подобное с инструкциями CALL и RET означает мешать правильному предсказанию перехода. Также обратите внимание, что 'ADD ESP' может вызвать задержку AGI в более ранних моделях процессоров.

Оптимизация для процессоров семейства Pentium: 18. Вывод из обращения (PPro, PII и PIII)

Автор: Агнер Фог, пер. Aquila

Дата: 2002-01-01

Вывод из обращения (retirement) - это процесс, когда временные регистры, используемые мопами, копируют в постоянные регистры EAX, EBX и так далее. Когда моп выполнен, он помечается в ROB как готовый к выводу из обращения.

Станция вывода из обращения может обрабатывать три мопа за такт. Может показаться, что здесь нет никакой проблемы, потому что вывод уже ограничен в RAT тремя мопами за такт. Тем не менее, вывод из обращения может стать узким местом по двум причинам. Во-первых, инструкции должны выводиться из обращения по порядку. Если моп был выполнен не по порядку, то он не может быть выведен из обращения, пока все мопы, предшествующие ему по порядку, не будут выведены из обращения до него. И второе ограничение - это то, что полученные переходы должны быть выведены из обращения в трех первых слотах станции. Как декодеры D1 и D2 могут быть неактивны, если следующая инструкция помещается только в D0, последние два слота станции вывода из обращения могут быть неактивны, если следующий моп, который должен быть веведен из обращения - это полученный (taken) переход. Это важно, когда у вас есть маленький цикл, количество мопов в котором не кратно трем.

Все мопы находятся в буфере перегруппировки (ROB), пока они не будут изъяты из обращения. ROB вмещает 40 мопов. Это устанавливает ограничение на количество инструкций, которые могут быть выполнены во время большой задержки, вызванной, например, делением или другой медленной операцией. Прежде, чем будет закончено деление, ROB будет заполнен выполняющимися мопами, ожидающими своего изъятия из обращения. Только когда деление будет закончено и изъято, последующие могут сами начать изыматься из обращения, потому что этот процесс должен выполняться по порядку.

В случае предварительного выполнения предсказанных переходов (смотри главу 22), предварительно выполненные мопы не могут быть изъяты из обращения, пока процессор не убедится, что предсказание верно. Если окажется обратное, тогда предварительно выполненные мопы сбрасываются без изъятия из обращения.

Следующие инструкции не могут быть предварительно выполнены: записи в память, IN, OUT и синхронизирующие операции.

Оптимизация для процессоров семейства Pentium: 19. Частичные задержки (PPro, PII и PIII)

Автор: Агнер Фог, пер. Aquila

Дата: 2012-11-24

19.1 Частичные задержки регистра

Частичная задержка регистра - это проблема, которая возникает, когда вы пишите в часть 32-х битного регистра, а затем читаете из всего регистра или его большей части. Пример:

```
MOV AL, BYTE PTR [M8]
MOV EBX, EAX            ; частичная задержка регистра
```

Происходит задержка в 5-6 тактов. Причина состоит в том, что в соответствие AL был поставлен временный регистр (чтобы сделать его независимым от AH). Модулю выполнения пришлось ждать, пока запись в AL не будет выведена из обращения, прежде чем станет возможным соединить значение AL с тем, что находится в остальной части EAX. Задержку можно избежать, изменив код, поменяв код на:

```
MOVZX EBX, BYTE PTR [MEM8]
AND EAX, 0FFFFFF00h
OR EBX, EAX
```

Конечно вы можете избежать частичную задержку, поместив другие инструкции после записи, чтобы у последней было время на вывод из обращения до того, как вы начнете читать из полного регистра.

Вам нужно остерегаться частичных задержек, когда вы смешиваете различные размеры данных (8, 16 и 32):

```
MOV BH, 0
ADD BX, AX              ; задержка
INC EBX                 ; задержка
```

У вас не будет задержки при чтении части регистра после записи в целый регистр или его большую часть:

```
MOV EAX, [MEM32]
ADD BL, AL              ; нет задержки
ADD BH, AH              ; нет задержки
MOV CX, AX              ; нет задержки
MOV DX, BX              ; задержка
```

Самый легкий путь избегать частиные задержки регистра - это всегда использовать полные регистры и использовать MOVZX или MOVSX при чтении из операндов более мелкого размера. Эти инструкции быстры на PPro, PII и PIII, но медленны на более ранних процессорах. Если вы хотите, чтобы ваш код выполнялся достаточно быстро на всех процессорах, то существует разумный компромисс. 'MOVZX EAX,BYTE PTR [M8]'

```
XOR EAX, EAX
MOV AL, BYTE PTR [M8]
```

Процессоры PPro, PII и PIII делают специальное исключение для этой комбинации, поэтому при последующем чтении из EAX задержки не возникнет. Происходит это потому, что регистр помечается как пустой, когда он XOR'ится сам с собой. Процессор помнит, что верхние 24 бита равны нулю и за счет этого избегается задержка. Этот механизм работает только со следующими комбинациями:

```
XOR EAX, EAX
MOV AL, 3
```

```
        MOV EBX, EAX            ; нет задержки

        XOR AH, AH
        MOV AL, 3
        MOV BX, AX              ; нет задержки

        XOR EAX, EAX
        MOV AH, 3
        MOV EBX, EAX            ; задержка

        SUB EBX, EBX
        MOV BL, DL
        MOV ECX, EBX            ; нет задержки

        MOV EBX, 0
        MOV BL, DL
        MOV ECX, EBX            ; задержка

        MOV BL, DL
        XOR EBX, EBX            ; нет задержки
```

Установка регистра в ноль вычитанием его из самого себя работает так же как XOR, но обнуление регистра с помощью инструкции MOV не предотвращает задержку.

Вы можете установите XOR снаружи цикла:

```
        XOR EAX, EAX
        MOV ECX, 100
LL:     MOV AL, [ESI]
        MOV [EDI], EAX          ; задержка
        INC ESI
        ADD EDI, 4
        DEC ECX
        JNZ LL
```

Процессор помнит, что верхние 24 бита EAX равны нулю пока не происходит вызов прерывания, неправильного предсказания перехода или другого синхронизирующего события.

Вы должны помнить, что необходимо нейтрализовывать возможные частичные задержки регистра вышеописанным способом при вызове процедуры, которая будет PUSH'ить полный регистр:

```
        ADD BL, AL
        MOV [MEM8], BL
        XOR EBX, EBX            ; нейтрализируем BL
        CALL _HighLevelFunction
```

Большинство языков высокого уровня PUSH'ат EBX в начале процедуры, что в вышеприведенном примере привело к частичной задержке регистра, если бы ее не нейтрализовали.

Обнуление регистра с помощью XOR не устраняет его зависимость от предыдущих инструкций:

```
        DIV EBX
        MOV [MEM], EAX

        MOV EAX, 0              ; прерываем зависимость
        XOR EAX, EAX            ; предотвращаем частичную задержку регистра
        MOV AL, CL
        ADD EBX, EAX
```

Обнуление регистра дважды может показаться излишней, но без 'MOV EAX, 0' последние инструкции будут ждать, пока выполниться медленный DIV, а без 'XOR EAX, EAX' случиться частичная задержка регистра.

Инструкция 'FNSTSW AX' особенная: в 32-х битном режиме она ведет себя так же, как если бы писала в весь EAX. Фактически она делает в 32-х битном режиме следующее:

```
AND EAX,0FFFF0000h / FNSTSW TEMP / OR EAX,TEMP
```

Поэтому при чтении регистра после этой инструкции у вас не возникнет частичной задержки регистра в 32-х битном режиме:

```
FNSTSW AX / MOV EBX,EAX         ; задержка только в 16-ти битном режиме
MOV AX,0  / FNSTSW AX           ; задержка только в 32-х битном режиме
```

19.2 Частичные задержки флагов

Регистр флагов также может вызвать частичную задержку:

```
CMP EAX, EBX
INC ECX
JBE XX          ; задержка
```

Инструкция JBE читает и флаг переноса, и флаг нуля. Так как инструкция INC изменяет флаг нуля, но не флаг переноса, то инструкции JBE приходится подождать, пока две предыдущие инструкции не будут выведены из обращения, прежде чем она сможет скомбинировать флаг переноса от инструкции CMP с флагом нуля от инструкции INC. Подобная ситуация больше похожа на баг, чем на преднамеренную комбинацию флагов. Чтобы скорректировать эту ситуацию, измените INC ECX на ADD ECX, 1. Похожий баг, вызывающий задержку, - это 'SAHF / JL XX'/. Инструкция JL тестирует флаг знака и флаг переполнения, но не меняет последний. Чтобы исправить это, измените 'JL XX' на 'JS XX'.

Неожиданно (и в противоположность тому, что говорят руководства от Intel), частичная задержка регистра может случиться, если были изменены какие-то биты регистра флагов, а затем считаны неизмененные.

```
CMP EAX, EBX
INC ECX
JC  XX          ; задержка
```

но не при чтении только изменных битов:

```
CMP EAX, EBX
INC ECX
JE  XX          ; нет задержки
```

Частичные задержки флагов возникают, как правило, при использовании инструкций, которые считывают много или все биты регистра флагов, например LAHF, PUSHF, PUSHFD. Инструкции, которые вызывают задержку, если за ними идут LAHF или PUSHF(D), следующие: INC, DEC, TEST, битовые тесты, битовые сканирования, CLC, STC, CMC, CLD, STD, CLI, STI, MUL, IMUL, и все виды битовых сдвигов и вращений. Следующие инструкции не вызывают задержки: AND, OR, XOR, ADD, ADC, SUB, SBB, CMP, NEG. Странно, что TEST и AND ведут себя различно, хотя по описанию они делают с флагами одно и то же. Вы можете использовать инструкции SETcc вместо LAHF или PUSHF(D) для сохранения значения флага, чтобы избежать задержки.

Примеры:

```
INC EAX   / PUSHFD      ; задержка
ADD EAX,1 / PUSHFD      ; нет задержки

SHR EAX,1 / PUSHFD      ; задержка
SHR EAX,1 / OR EAX,EAX / PUSHFD   ; нет задержки

TEST EBX,EBX / LAHF     ; задержка
AND  EBX,EBX / LAHF     ; нет задержки
TEST EBX,EBX / SETZ AL  ; нет задержки

CLC / SETZ AL           ; задержка
CLD / SETZ AL           ; нет задержки
```

Потери при частичной задержки флагов примерно равны 4 тактам.

19.3 Задержки флагов после сдвигов и вращений

При чтении любого флагового бита после сдвига или вращения (кроме сдвига и вращения на 1, т.н. короткая форма) может возникнуть задержка, похожая на частичную задержку флагов:

```
SHR EAX,1 / JZ XX                ; нет задержки
SHR EAX,2 / JZ XX                ; задержка
SHR EAX,2 / OR EAX,EAX / JZ XX   ; нет задержки

SHR EAX,5 / JC XX                ; задержка
SHR EAX,4 / SHR EAX,1 / JC XX    ; нет задержки

SHR EAX,CL / JZ XX               ; задержка, даже если CL = 1

SHRD EAX,EBX,1 / JZ XX           ; задержка
ROL EBX,8 / JC XX                ; задержка
```

Потери для этого вида задержек приблизительно равны 4 тактам.

19.4 Частичные задержки памяти

Частичная задержка памяти похожа на частичную задержку регистра. Она случается, когда вы смешиваете размеры данных применительно к одному адресу в памяти:

```
MOV BYTE PTR [ESI], AL
MOV EBX, DWORD PTR [ESI]        ; частичная задержка в памяти
```

Здесь случается задержка, потому что процессор должен скомбинировать байт, записанный из AL с тремя следующими байтами, которые были в памяти раньше, чтобы получить все четыре байта, необходимые для произведения записи в EBX. Потери приблизительно равны 7-8 тактов.

В отличии от частичных задержек регистра, частичная задержка памяти случается, когда вы записывает операнд в память, а затем читаете его часть, если она не начинается по тому же адресу:

```
MOV DWORD PTR [ESI], EAX
MOV BL, BYTE PTR [ESI]          ; нет задержки
MOV BH, BYTE PTR [ESI+1]        ; задержка
```

Вы можете избежать этой задержки, если измените последнюю лини на 'MOV BH, AH', но подобное решение невозможно в ситуации, подобной этой:

```
FISTP QWORD PTR [EDI]
MOV EAX, DWORD PTR [EDI]
MOV EDX, DWORD PTR [EDI+4]      ; задержка
```

Интересно, что частичная задержка памяти может также случиться при записи и чтении совершенно разны адресов, если у них одинаковое set-значение в разных банках кэша:

```
MOV BYTE PTR [ESI], AL
MOV EBX, DWORD PTR [ESI+4092]   ; нет задержки
MOV ECX, DWORD PTR [ESI+4096]   ; задержка
```

Оптимизация для процессоров семейства Pentium: 20. Цепочки зависимости (PPro, PII и PIII)

Автор: Агнер Фог, пер. Aquila

Дата: 2012-11-24

Серии инструкций, где выполнение каждой зависит от результата предыдущей, называется цепочкой зависимости. Большие цепочки нужно по возможности избегать, потому что они делают невозможным выполнение не по порядку и параллельное выполнение.

Пример:

```
MOV EAX, [MEM1]
ADD EAX, [MEM2]
ADD EAX, [MEM3]
ADD EAX, [MEM4]
MOV [MEM5], EAX
```

В этом примере инструкция ADD генерирует 2 мопа, один для чтения из памяти (порт 2) и один для сложения (порт 0 или 1). Моп чтения может выполняться не по порядку, в то время как моп сложения дожна ждать, пока выполниться предыдущий моп. Эта цепочка зависимости занимает не очень много времени, так как каждое сложение требует только один такт. Но если в вашем коде содержатся медленные инструкции, такие как умножения, или еще хуже - деление, тогда вам определенно нужно сделать что-нибудь, чтобы убрать цепочку зависимости. Это можно сделать, используя разные приемники:

```
MOV EAX, [MEM1]         ; начало первой цепочки
MOV EBX, [MEM2]         ; начало второй цепочки  с другим приемником
IMUL EAX, [MEM3]
IMUL EBX, [MEM4]
IMUL EAX, EBX           ; в конце соединяем цепочки
MOV [MEM5], EAX
```

Здесь вторая инструкция IMUL может начаться до того, как будет завершено выполнение первой. Так как у инструкции IMUL вызывает задержку в 4 такта и полностью конвееризована, вы можете использовать до 4-х приемников.

Деление не конвееризовано, поэтому вы не можете делать то же самое со связанными делениями, но, разумеется, вы можете умножить все делители и сделать только одно деление в конце.

У инструкций с плавающей запятой более длинная задержка, чем у целочисленных инструкций, поэтому вам стоит разбивать слишком длинные цепочки связанных инструкций с плавающей запятой.

```
FLD [MEM1]          ; начинаем первую цепочку
FLD [MEM2]          ; начинаем вторую цепочку с другим приемником
FADD [MEM3]
FXCH
FADD [MEM4]
FXCH
FADD [MEM5]
FADD                ; соединяем цепочки в конце
FSTP [MEM6]
```

Вам потребуется для этого много инструкций FXCH, но не беспокойтесь: они стоят дешево. Инструкции FXCH обрабатываются в RAT с помощью переименования регистров, поэтому они не создают никакой назгрузки на порты выполнения. Тем не менее, FXCH генерирует один моп в RAT, ROB и в станции вывода из обращения.

Если цепочка зависимости очень длиная, вам может потребоваться три приемника:

```
FLD [MEM1]              ; начинаем первую цепочку
FLD [MEM2]              ; начинаем вторую цепочку
FLD [MEM3]              ; начинаем третью цепочку
FADD [MEM4]             ; третья цепочка
FXCH ST(1)
FADD [MEM5]             ; вторая цепочка
FXCH ST(2)
FADD [MEM6]             ; первая цепочка
FXCH ST(1)
FADD [MEM7]             ; третья цепочка

FXCH ST(2)
FADD [MEM8]             ; вторая цепочка
FXCH ST(1)
FADD                    ; соединяем первую и третью цепочку
FADD                    ; результат соединяем со второй цепочкой
FSTP [MEM9]
```

Избегайте сохранения промежуточных данных в памяти и немедленного их считывания:

```
MOV [TEMP], EAX
MOV EBX, [TEMP]
```

Возникают потерю из-за попытки чтения из памяти до того, как завершена предыдущая запись туда же. В вышеприведенном примере измените последнюю инструкцию на 'MOV EBX, EAX' или поместите какие-нибудь инструкции между ними.

Есть одна ситуация, когда вы не сможете избежать сохранения промежуточных данных в памяти. Она возникает тогда, когда вы перемещаете данные из целочисленного регистра в регистр FPU или наоборот. Например:

```
MOV EAX, [MEM1]
ADD EAX, [MEM2]
MOV [TEMP], EAX
FILD [TEMP]
```

Если вам нечего поместить между записью в TEMP и считыванием из него, вы можете использовать регистр плавающей запятой вместо EAX:

```
FILD [MEM1]
FIADD [MEM2]
```

Последовательные переходы, вызовы или возвраты также можно считать цепочками зависимости. Производительность этих инструкций равна одному переходу за два такта. Поэтому рекомендуется загружать микропроцессор какой-нибудь полезной работой между переходами.

Оптимизация для процессоров семейства Pentium: 21. Поиск узких мест (PPro, PII и PIII)

Автор: Агнер Фог, пер. Aquila

Дата: 2012-11-24

Оптимизируя код для этих процессоров, важно проанализировать, где находятся узкие места. Оптимизация одного узкого места не будет иметь смысла, если есть другое еще уже.

Если вы ожидаете потери при работе с кэшем, вам нужно перегруппировать код, чтобы наиболее используемые части кода были ближе друг к другу.

Если вы ожидаете много потерь при работе с кэшем данных, забудьте обо всем другом и сконцентрируйтес на том, как реструктуризовать ваши данные, чтобы снизить количество потерь (глава 7) и избежать долгих цепочек зависимости при неоптимальной работе с кэшем.

Если у вас много делений, попробуйте снизить их количество и убедитесь, что процессору есть чем заняться во время обработки операции деления.

Цепочки зависимости мешают выполнению не по порядку (глава 20). Попробуйте разрывать длиную цепочку зависимости, особенно, если они содержат медленные инструкции, такие как умножение, деление и инструкции плавающей запятой.

Если у вас много переходов, вызовов или возвратов, особенно если переходы плохо предсказуемы, посмотрите, нельзя ли избежать некоторых из них.Заменяйте условные переходы условными перемещениями, если это возможно, а маленькие процедуры макросами.

Если вы смешиваете разные типы данных (8, 16 и 32 битные целые числа), тогда следите за появление частичных задержек. Если вы используете инструкции PUSHF или LAHF, тогда следите за появлением частичных задержек флагов. Избегайте тестирования флагов после сдвигов больше, чем на 1 (глава 19).

Если вы нацелились на производительность в 3 мопа за такт, тогда следите, чтобы не происходило перебоев в доставке инструкций и их раскодировке (глава 14 и 15), особенно в маленьких циклах.

Ограничение на чтение максимум из двух постоянных регистров может снизить вашу производительность до уровня, меньшего чем 3 мопа за такт. Это может случиться, если вы часто читаете регистры, когда уже истекли четыре такта с момента их последнего изменения. Это может, например, случиться, если вы часто используете указатели для адресации, но редко модифицируете указатели.

Производительность в три мопа за такт требует, чтобы никакой порт выполнения не получал больше 1/3 всех мопов (глава 17).

Станция вывода из обращения может обрабатывать 3 мопа за такт, но может быть менее эффективной при работе с полученными переходами (глава 18).

Оптимизация для процессоров семейства Pentium: 22. Команды передачи управления и переходов (все процессоры)

Автор: Агнер Фог, пер. Aquila

Дата: 2012-11-24

Семейство процессоров Pentium пытаются предсказывать, когда произойдет безусловный переход и будет ли осуществлен условный. Если предсказание оказывается верным, тогда это может сэкономить существенное количество времени, так как в конвеер будут загружены последующие инструкции, и начнется их раскодировка еще до того, как будет осуществлен сам переход. Если пресказание оказывается неверным, тогда конвеер должен быть очищен, что вызовет потери производительности, количество которых зависит от длины конвеера.

Предсказания основываются на буфере переходов (BTB), который сохраняет историю каждого перехода и делает предсказания, основываясь на прежних выполнения каждой инструкции. BTB организован как множественно-ассоциативный (set-associative) кэш, в котором новые элементы используются согласно псевдослучайному методу замещений.

Оптимизируя код важно снижать количество возможных неправильных предсказаний перехода. Это можно сделать при хорошом понимании того, как работает предсказание переходов.

Механизм предсказания переходов адекватно не объясняется ни в руководствах от Интела, ни где-нибудь еще, поэтому здесь дается очень детальное описание. Эта информация основывается на моих собственных исследованиях (осуществленных с помощью Karki Jitendra Bahadur для PPlain).

Далее я буду использовать термин "команда передачи управления" для любой инструкции, которая меняет eip, включая условные и безусловные, прямые и косвенные, ближние и дальние переходы, вызовы и возвраты. Все эти инструкции используют предсказание.

22.1 Предсказывание переходов в PPlain

Механизм предсказывания переходов в PPlain очень отличается от того, как это реализовано в других трех процессорах. Информация, найденная по этой теме в документах от Интела и где-либо еще очень противоречива, и данные в этих документах советы приводят к неполностью оптимизированному коду.

У PPlain есть буфер переходов (BTB), который может содержать информацию о 256 инструкциях перехода. BTB организована в виде 4-х направленного множественно-ассоциативного кэша по 64 элемента на каждом направлении. Это означает, что BTB может содержать не больше чем 4 элемента для каждого set-значения. Как высчитывается set-значение будет объяснено позже. Каждый элемент BTB хранит адрес, куда должен быть совершен переход и состояние предсказания, которое может иметь три разных значения:

• состояние 0: "не будет осуществлен"
• состояние 1: "возможно не будет осуществлен"
• состояние 2: "возможно будет осуществлен"
• состояние 3: "будет осуществлен"

Предсказывается, что инструкция перехода будет осуществлена, когда состояние равно 2 или 3, и пропускается при состоянии 0 или 1. Состояние перехода работает как двух-битный счетчик, поэтому состояние увеличивается на 1, если переход был осуществлен и понижается, если этого не произошло. Понижение и повышение значения происходит по принципу "насыщения", то есть

значение не может понизиться ниже 0 (и стать 3) и не может повыситься выше 3 (и стать 0). По идее, все это должно привести к довольно хорошему проценту правильных предсказаний, потому что до того, как будет изменено предсказание, инструкция перехода должна быть выполнена два раза не так, как она выполняется обычно.

Тем не менее, этот механизм может был поставлен под угрозу тем, что состояние 0 также означает "неиспользованный элемент BTB". Поэтому элемент BTB с состоянием 0 означает примерно то же, что если бы его вообще не было. Это имеет смысл, потому что если у инструкции перехода нет соответствующего элемента BTB, предсказывается, что она не будет осуществлена. Это улучшает использование BTB, так как инструкция перехода, которая редко осуществляется большую часть времени не будет занимать никакого элемента BTB.

Если у инструкции перехода нет своего элемента в BTB, генерируется такой элемент и значение его состояние будет установлено в 3. Это означает, что перейти от состояния 0 к состоянию 1 невозможно (кроме очень специфических случаев, обсуждаемых позже). От состояния 0 вы можете перейти только к состоянию 3, если иснтрукция перехода будет выполнена. Если этого не произойдет, она не получит свой элемент в BTB.

Это серьезный промах в дизайне архитектуры процессора. Очевидно, что таким образом дизайнеры отдали предпочтение минимизации потери роизводительности при загрузке в первый раз часто выполняющихся инструкций перехода и проигнорировали то, что это серьезно искажает первоначальную идею и снижает производительность в маленьких внутренних циклах. Следствием этого изъяна является то, что инструкция перехода, которая большую часть времени не выполняется, может иметь в три раза больше неправильных предсказаний, чем инструкция перхода, которая большую часть времени выполняется. (Похоже, что пока я не опубликовал свои исследования, инженеры Интела были не в курсе насчет этого).

Вы можете принять эту ассиметричность в расчет, организовав ваши ветви так, чтобы они чаще выполнялись, а не пропускались. Посмотрите на этот пример конструкции if-then-else:

```
        TEST EAX,EAX
        JZ   A
<ветвь 1>
        JMP  E
A:      <ветвь 2>
E:
```

Если ветвь 1 выполняется чаще, чем ветвь 2, а последняя выполняется реже в два раза, тогда вы можете снизить количество неправильных предсказаний, поменяв две ветви так, чтобы инструкция перехода выполнялась чаще, чем пропускалась:

```
        TEST EAX,EAX
        JNZ  A
        <ветвь 2>
        JMP  E
A:      <ветвь 1>
E:
```

(Это противоречит рекомендациям в интеловских руководствах и туториалах).

Тем не менее, есть причины, что помечать чаще выполняющуюся вертвь первой:

• Помещение редко используемых ветвей подальше от основного кода делает использование кода кэша более оптимальным.
• Редко выполняющаяся инструкция перехода не будет напрасно занимать элемент BTB, что, вероятно, сделает использование BTB более оптимальным.
• Инструкция перехода будет предсказан как не выполняющаяся, если она была вытеснена другими инструкциями из BTB.
• Ассиметричность предсказывания переходов существует только в PPlain.

Впрочем, эти предположения не играет никакой роли для маленьких критических циклов, поэтому я все же рекомендую организовывать ветви так, как это было описано выше.

Так же вам следует организовать циклы таким образом, чтобы проверка условия производилась внизу цикла, как в нижеприведенном примере:

```
        MOV ECX, [N]
L:      MOV [EDI],EAX
        ADD EDI,4
        DEC ECX
        JNZ L
```

Если N велико, тогда инструкция JNZ будет чаще выполняться, чем пропускаться.

Представьте ситуацию, где инструкция перехода выполняется каждый второй раз. В первый раз она попадает в BTB с состоянием 3, а затем будет колебаться между состояниями 2 и 3. Каждый раз будет предсказываться, что она выполниться, но только 50% этих предсказаний будет верными. Теперь предположите, что однажды она отклонилась от привычного поведения и была пропущена.

```
01010100101010101010101, where 0 means nojump, and 1 means jump.
        ^

01010100101010101010101, где 0 означает пропуск, а 1 означает переход.
        ^
```

Дополнительный пропуск помечен знаком '^'. После этого происшествия элемент BTB будет колебаться между состояниями 1 и 2, что будет давать 100% неправильных предсказаний. И так будет продолжаться, пока не случиться еще одно отклонение. Это худшее, что можете произойти в работе этого механизма предсказания переходов.

22.1.2 BTB смотрит вперед (PPlain)

Механизм BTB подсчитывает инструкции попарно, а не по отдельности, поэтому вы должны знать, как инструкции спариваются, чтобы суметь проанализировать, где храниться элемент BTB. Элемент BTB для любой управляющей инструкции присоединяется к адресу инструкции в U-конвеере из предыдущей пары инструкций (неспариваемая инструкция идет за одну пару).

Пример:

```
SHR EAX,1
MOV EBX,[ESI]
CMP EAX,EBX
JB  L
```

Здесь SHR спаривается с MOV, а CMP спаривается с JB. BTB-элемент для 'JB L' присоединяется к адресу инструкции 'SHR EAX,1'. Когда встречается этот BTB-элемент, и он находится в состоянии 2 или 3, тогда Pentium считает целевой адрес из элемента BTB и загрузит инструкции, следующие за L в конвеер. Это произойдет до того, как будет декодирована инструкция перехода, поэтому Pentium полагается исключительно на информацию, содержащуюся в BTB, когда делает это.

Вы можете вспомнить, что инструкции редко спариваются в первый раз при выполнении (смотрите главу 8). Если инструкции выше не спарились, тогда элемент BTB будет присоединен к адресу инструкции CMP, и данный элемент будет неверен при следующем выполнении, когда инструкции уже спарились. Тем не менее, в большинстве случаев PPlain достаточно умен, чтобы не делать элемент BTB при неиспользованной возможности спаривания, поэтому вы получите BTB-элемент только при втором выполнении, а следовательно, предсказания начнутся только с третьего. (В редких случаях, когда каждая вторая инструкция размером в один байт, элемент BTB может возникнуть уже при первом выполнении, что окажется неверным при втором выполнении, но так как инструкция, к которой был присоединен элемент, попадет в V-конвеер, это игнорируется и не вызывает особых потерь. Элемент BTB считывается только тогда, когда он присоединен к адресу

инструкции для U-конвеера).

BTB-элемент идентифицируется своим set-значением, которое равно битам 0-5 адреса, к которому он присоединен. Биты 6-31 сохраняются в BTB как тег. Адреса, которые находятся на расстоянии кратном 64 байтам, будут иметь одно и то же set-значение. Вы можете иметь не более четырех BTB-элементов с одинаковым set-значением. Если вы хотите проверить, не будет ли ваша инструкция перехода бороться с другой инструкцией за один и тот же элемент BTB, вы можете сравнить биты 0-5 адресов инструкций, направляющихся в U-конвеер, из предыдущих пар инструкций. Это очень утомительно, и я никогда не слышал о том, что кто-нибудь делал подобное. Нет никаких инструментов, которые могли бы выполнить эту работу за вас.

22.1.3 Последовательные ветви (PPlain)

Когда переход пресдказан неправильно, то конвеер очищается. Если следующая пара инструкций также содержит управляющую инструкцию, то PPlain не будет загружать ее цель, так как он не может этого сделать, пока конвеер не очищен. В результатет вторая инструкция предсказывается как невыполняющаяся вне зависимости от состояния элемента BTB. Поэтому, если второй переход также выполняется на самом деле, то вы получите новые потери. Состояние BTB-элемента для второго перехода будет, тем не менее, правильно изменено. Если у вас есть длиная цепочка инструкций передачи управления, и первый переход в цепочке был предсказан неправильно, тогда конвеер будет постоянно очищаться, и предсказания будут постоянно неправильными, пока не будет встречена пара инструкций, в которой не будет перехода. Самый экстремальный случай подобного рода - это цикл, в котором происходят потери при каждой итерации из-за постоянного неправильного предсказания.

Это не едиснтвенная проблема с последовательными управляющими инструкциями. Другая проблема состоит в том, что у вас может быть другая управляющая инструкция между BTB-элементом и управляющей инструкцией, принадлежащей ему. Если первая инструкция производит переход, то могут произойти странные вещи. Представьте следующий пример:

```
        SHR EAX,1
        MOV EBX,[ESI]
        CMP EAX,EBX
        JB  L1
        JMP L2

L1:     MOV EAX,EBX
        INC EBX
```

Когда 'JB L1' пропускается, вы можете получить элемент BTB, присоединенный к адресу 'CMP EAX,EBX'. Но что случится, если 'JB L1' будет выполнена? В тот момент, когда считывается BTB-элемент для 'JMP L2', процессор не знает, что следующая пара инструкций не содержит команду перехода, поэтому он фактически предскажет, что пара инструкций 'MOV EAX,EBX / INC EBX' перейдет на L2. Потери, связанные с предсказанием инструкций, не являющихся командами перехода, равны 3 тактам. Элемент BTB для 'JMP L2' понизит свое состояние на один, потому что он относится к чему-то, что не совершает перхода. Если мы перейдем к L1, тогда состояние элемента BTB для 'JMP L2' будет понижежено до 1 и 0, поэтому данная проблема исчезнет, пока не снова не будет выполнено 'JMP L2'.

Потери при предсказании инструкций, не являющихся управляющими, случаются только тогда, когда предсказавыется переход на L1. В случае, если выполнение 'JB L1' предсказано ошибочно, тогда конвеер очищается, и мы не получим фальшивую загрузку цели L2, поэтому в этом случае мы не заметим потери от предсказания неуправляющих инструкций как выполняющихся, но состояние элемента BTB для 'JMP L2' будет понижено.

Теперь предположите, что мы заменяем инструкцию 'INC EBX' на другую инструкцию перехода. Тогда третья инструкция перехода будет использовать тот же элемент BTB, что и 'JMP L2', что приводит к возможным потерям из-за предсказания неправильной цели (если только целью не

является L2).

Теперь прорезюмируем возможные проблемы, к которым могут привести последовательные переходы:

• невозможность загрузить цель перехода, когда конвеер очищается из-за предыдущего неправильно предсказанного перехода.
• элемент BTB может быть присоединен не к инструкции перехода и предсказать их как выполняющиеся.
• второе последствие вышеизложенного - неправильно присоединенный элемент BTB будет понижать свое состояние, что может привести к последующему неправильному предсказанию перехода, к которому он принадлежит. Даже безусловные переходы из-за этого могут быть предсказанны как невыполняющиеся.
• две инструкции перехода могут использовать один и тот же элемент BTB, что может привести к предсказанию неправильной цели.

Вся эта путаница может привести к многчисленным потерям, поэтому вы определенно должны избегать пар инструкций, содержащих переход, находящихся непосредственно после плохо предсказуемой инструкции передачи управления.

Теперь еще один показательный пример:

```
        CALL P
        TEST EAX,EAX
        JZ   L2

L1:     MOV  [EDI],EBX
        ADD  EDI,4
        DEC  EAX
        JNZ  L1
L2:     CALL P
```

На первый взгляд как будто все в порядке: вызов функции, цикл, который пропускается, если счетчик равняется нулю, и другой вызов функциии. Какие проблему могут возникнуть?

Сначала мы можем заметить, что функция P вызывается из двух различных мест кода. Это означает, что цель возвращения из P будет все время меняться. Соответственно, возврат из P будет все время предсказываться неправильно.

Теперь предположите, что EAX равен нулю. При переходе на L2 его цель не будет загружена, так как неправильно предсказанное возвращение приведет к очищению конвеера. Затем цель второго вызова P также не будет загружена, потому что 'JZ L2' вызовет новое очищение ковеера. Здесь мы имеем ситуацию, в которой цепь последовательных переходов заставляет постоянно очищаться конвеер из-за того, что первый переход был предсказан неправильно. Элемент BTB для 'JZ L2' сохраняется по адресу инструкции возвращения из функции P. Этот элемент BTB будет теперь неправильно присоединен к тому, что должно быть после второго вызова P, но это не приводит к потерям, так как конвеер очищается из-за неправильно предсказанного второго возвращения.

Теперь давайте посмотрим, что случится, если EAX не равен нулю в следующий раз: 'JZ L2' будет всегда предсказываться как невыполняющийся из-за очищения конвеера. Второй 'CALL P' будет иметь свой элемент по адресу 'TEST EAX,EAX'. Этот элемент будет неправильно ассоциирован с парой 'MOV/ADD', предсказывая их переход на P. Это приведет к очищению конвеера, который не даст загрузить цель 'JNZ L1'. Если мы уже были здесь ранее, тогда второй 'CALL P' будет иметь другой элемент BTB с адресом 'DEC EAX'. При втором и третьем повторении цикла этот элемент также будет неправильно ассоциирован с парой 'MOV/ADD', пока ее состояние не понизится до 1 или 0. Это не вызовет потерь во втором выполнении, так как очищение конвеера от 'JNZ L1' не даст загрузить неверную цель, но в третьем выполнении так и случится. Последовательные итерации цикла не приводят к потерям, но если они есть, то JNZ L1 будет предсказан неправильно.

Очищение буфера сделает невозможной загрузку цели 'CALL P', не считая того, что ее элемент BTB уже был уничтожен несколькими неправильными ассоциированиями.

Мы можем улучшить этот код, поместив несколько NOP'ов, чтобы отделить друг от друга последовательные переходы:

```
        CALL P
        TEST EAX,EAX
        NOP
        JZ   L2
L1:     MOV  [EDI],EBX
        ADD  EDI,4
        DEC  EAX
        JNZ  L1
L2:     NOP
        NOP
        CALL P
```

Дополнительные NOP'ы стоят 2 такта, но они сохранят гораздо больше. Более того, 'JZ L2' теперь перемещяется в U-конвеер, что снижает потери в случае неправильного предсказания с 4 до 3 тактов. Единственная оставшаяся проблема - это возвраты из P, которые всегда предсказываются неправильно. Эту проблему можно решить, заменив вызов P на макрос (если у вас есть достаточное количество свободного кэша кода).

Урок, который можно извлечь из этого примера, заключается в том, что вам всегда стоит внимательно контролировать последовательные переходы и смотреть, не можете ли вы сэкономить немного времени с помощью нескольких NOP'ов. Вам следует избегать таких ситуаций, когда неправильное предсказание неибежно, например, выходы из циклов и возвращения из процедур, вызываемых из разных мест. Если у вас есть что-нибудь полезное, что можно поместить вместо NOP'ов, тогда вам, конечно, следует поместить именно это.

Мультиветвенные переходы (выражения условий) можно сделать либо в виде дерева инструкций переходов, либо в виде списка адресов переходов. Если вы выберите дерево инструкций переходов, то вам стоит включить несколько NOP'ов или других инструкций, чтобы отделить последовательные переходы друг от друга, поэтому список адресов может быть более выгодным вариантом на PPlain. Список адресов переходов следует помещать в сегменте данных. Никогда не помещайет данные в сегменте кода!

22.1.4 Маленькие циклы (PPlain)

В маленьких циклах часто происходит доступ к одному и тому же элементу BTB с небольшими интервалами. Вместо того, чтобы ждать, когда обновится элемент BTB, PPlain каким-то образом минует конвеер и получает состояние после последнего перехода, прежде, чем оно записывает в BTB. Этот механизм почти прозрачен для пользователя, но в некоторых случаях он приводит к забавным эффектам: вы можете видеть предсказание перехода от состояния 0 к состоянию 1, а не к состоянию 3, если ноль еще не быль записан в BTB. Это случается, если в цикле не больше четырех пар инструкций. В цикла с двумя парами инструкций может случиться так, что состояние 0 сохраняется на протяжении двух итераций, причем элемент не стирается из BTB. В таких маленьких циклах в редких случаях предсказание использует состояние, бывшее два цикла назад, а не в предыдущем повторении. Эти забавные эффекты обычно не приводят к снижению качества.

22.2 Предсказание переходов в PMMX, PPro, PII и PIII

22.2.1 Организация BTB (PMMX, PPro, PII и PIII)

Буфер переходов (BTB) PMMX состоит из 256 элементов, организованых как 16 направлений * 16 множеств. Каждый элемент идентифицируется битами 2-31 адреса последнего байта инструкции передачи управления, которой он принадлежит. Биты 2-5 определяют множество, а биты 6-31 сохраняются в BTB как тег. Инструкции передачи управления, находящиеся друг от друга на расстоянии 64 байт, имеют одинаковое set-значение, и поэтому могут случайно выдавливать друг

друга из BTB. Так как есть 16 направление на множество, то это не будет происходить слишком часто.

Буфер переходов PPro, PII и PIII имеет 512 элементов (16 направлений * 32 множества). Каждый элемент идентифицируется битами 4-31 адреса последнего байта инструкции передачи управления, к которой он принадлежит. Биты 4-8 определяют множество, и все биты сохраняются в BTB как тег. Инструкции передачи управления, которые находятся друг от друга на расстоянии 512 байт, могут случайно выдавливать друг друга из BTB. Так как есть 16 направлений на множество, это не бдует происходить слишком часто.

PPro, PII и PIII резервируют элемент BTB для любой инструкции передачи управления в первый раз ее выполнения (независимо от того, совершает ли она переход, или пропускается). PMMX резервирует элемент в первый раз, когда совершается переход. Инструкция перехода, которая никогда не совершает переход, будет оставаться вне BTB на PMMX. Как только она однажды совершит переход, она окажется в BTB и будет оставаться там все время, даже если она никогда не будет совершать переход снова.

Элемент может быть выдавлен из BTB, когда другой инструкции передачи управления с тем же самым set-значение требуется элемент BTB.

22.2.2 Потери при неверном предсказании (PMMX, PPro, PII и PIII)

На PMMX потери из-за неверного предсказания условного перехода равны 4 тактам для инструкции в U-конвеере и 5 тактам, если она выполнялась в V-конвеере. Для всех других инструкций передачи управления потери равны 4 тактам.

На PPro, PII и PIII потери из-за неверного предсказания очень высоки, так как у этих процессоров длинный конвеер. Неправильное предсказание обычно стоит от 10 до 20 тактов. Поэтому очень важно избегать плохо предсказуемых ветвей, если программа будет выполняться на PPro, PII и PIII.

22.2.3 Распознавание последовательностей условных переходов (PMMX, PPro, PII и PIII)

У этих процессоров есть продвинутый механизм распознавания последовательностей, который может правильно предсказать выполнение инструкции переходов, например, если так совершает переход каждый четвертый раз и не выполняется в остальные три раза. Фактически они могут предсказать любую повторяющуюся последовательность переходов и не-переходов, если период не превышает пяти, и многие последовательности с более высокими периодами.

Этот механизм называется "двух-уровневой адаптирующейся схемой предсказания". Он был изобретен T.-Y. Yeh и Y. N. Patt. Механизм основывается на разновидности двухбитного счетчика, использующихся в PPlain (но без изъяна, описанного выше). Счетчик повышается, когда совершается переход и понижается, когда этого не происходит. Нет автоматического обнуления при повышении 3 или приравнивания к 3 при понижении 0. Инструкция перехода предсказывается как выполняющаяся, если соответствующий счетчик равен 3 или 3, и предсказывается как несовершающая переход, если он равен 0 или 1. Значительное улучшение данного алгоритма было получено путем введения 16 таких счетчиков для каждого элемента BTB. Он выбирает один из этих шестнадцати счетчиков, основываясь на истории выполнения переходов за четыре последних раза. Если, например, инструкция перехода однажды выполняется, а потом нет на протяжении четырех последних раз. Если, например, инструкция перехода совершает его однажды и затем не делает его три раза подряд, тогда биты истории равны 1000 (1 = переход, 0 = не-переход). Значит, будет использоваться счетчик под номером 8 (1000b = 8) для предсказания в следующий раз и обновления состояния счетчика.

Если последовательность 1000 всегда следует за 1, тогда счетчик под номером 8 вскоре достигнет своего нивысшего состояния (3), что означает постоянное предсказываения последовательности 1000. Потребуется два отклонения от этой последовательности, чтобы изменить предсказание. Посвторяющаяся последовательность 100010001000 будет иметь счетчик 8 в состоянии 3 и счетчик

1, 2 и 4 в состоянии 0. Другие двенадцать счетчиков не будут использоваться.

22.2.4 Последовательности, предсказанные совершенно (PMMX, PPro, PII, PIII)

Повторяющаяся последовательность предсказывается совершенно этим механизмом, если каждая 4-х битная подпоследовательность полного периода уникальная. Ниже представлен список последовательностей, которые предсказываются совершенно:

```
period  perfectly predicted patterns
1-5     all
6       000011, 000101, 000111, 001011
7       0000101, 0000111, 0001011
8       00001011, 00001111, 00010011, 00010111, 00101101
9       000010011, 000010111, 000100111, 000101101

10      0000100111, 0000101101, 0000101111, 0000110111, 0001010011,

        0001011101
11      00001001111, 00001010011, 00001011101, 00010100111

12      000010100111, 000010111101, 000011010111, 000100110111,
        000100111011
13      0000100110111, 0000100111011, 0000101001111

14      00001001101111, 00001001111011, 00010011010111, 00010011101011,
        00010110011101, 00010110100111
        000010011010111, 000010011101011, 000010100110111,
15      000010100111011, 000010110011101, 000010110100111,
        000010111010011, 000011010010111

16      0000100110101111, 0000100111101011, 0000101100111101,
        0000101101001111
```

Читая эту таблицу, вы должны учитывать, что если последовательность предсказывается верно, тогда та же последовательность, но прочитанная задом-наперед также будет предсказана верно, также как и та же последовательность, но с инвертированным битами. Пример: в таблице мы нашли последовательность: 001011. Изменение порядка битов на обратный дает: 1101000. Инвертирование всех битов дает 1110100. Изменение порядка битов и инвертирование вместе: 0010111. Все эти четыре последовательности будут распознаны. Циклический сдвиг всех битов на одну позицию влево дает: 0010110. Это, конечно, не новая последовательность, а всего лишь чуть сдвинутая версия той, которую мы рассматривали. Все последовательности, которые могут быть выведены от одной из перечисленных в таблице с помощью изменения порядка битов, инвертирования и циклического сдвига также могут быть разпознанны. В целях экономии места они не были перечисленны.

У механизма распознавания последовательностей уходит два периода на то, чтобы выучить регулярную повторяющуюся последовательность после того, как был создан соответствующий элемент BTB. Последовательность неправильных предсказаний в период обучения не воспроизводима, вероятно из-за того, что элемент BTB содержит что-то до того, как резервируется. Так как элементы BTB резервируются по случайной схеме, то предсказание того, что произойдет в течении начального периода, маловероятно.

22.2.5 Обработка отклонений от регулярных последовательностей (PMMX, PPro, PII и PIII)

Механизм предсказания также довольно хорошо справляется с обработкой 'почти регулярных' последовательностей или отклонений от регулярной последовательности. Он не только заучивает, как выглядит регулярная последовательность, он также изучает, какие существуют отклонения от нее. Если отклонения все время одни и те же, тогда он будет помнить, что идет после отклонения, и оно будет стоить толко одно неправильное предсказание.

Пример:

```
00011100011100011100010111000111000111000010111000
```

```
                ^                 ^
```

В этой последовательности 0 означает не-переход, а 1 - переход. Механизм выучивает то, что повторяющаяся последовательность равна 000111. Первое отклонение - это неожиданный 0, который я отметил знаком '^'. После этого 0 следующие три перехода могут быть неправильно предсказаны, потому что механизм еще не выучил, что идет после 0010, 0101 и 1011. После одного или двух отклонений того же вида он выучивает, что после 0010 идет 1, после 0101 идет 1, а после 1011 идет 1. Это означает, что после двух отклонений одного и того же вида, он выучивает как их обрабатывать, допуская только одно неправильное предсказание.

Механизм предсказания очень эффективени, когда происходит смена одной регулярной последовательности на другую. Если, например, у нас была последовательность 000111 (с периодом 6), которая повторилась много раз, потом последовательность 01 (период 2) много раз, а затем произошел возврат на последовательность 000111, тогда механизм не должен снова изучать последовательность 000111, так как счетчики, которые были использованы для этой последовательность остались в неприкосновенности. После нескольких смен последовательностей с одной на другую, механизм также выучивает, как обрабатывать изменения ценой только одного неправильного предсказания во время переключения одной последовательности на другую.

22.2.6 Последовательности, не предсказываемые совершенно (PMMX, PPro, PII и PIII)

Простейшая последовательность, которая не может быть предсказана совершенно - это переход, который совершается каждый 6-ой раз. Вот последовательность:

```
00001000001000001
    ^^    ^^    ^^
    ab    ab    ab
```

За последовательностью 0000 следует 0 в позициях, помеченных 'a', и 1, в позициях, помеченных 'b'. Это влияет на счетчик под номером 0, который постоянно повышает и понижает свое значение. Если счетчик 0 был вначале равен 0 или 1, тогда он будет колебаться между 0 и 1. Это приведет к неправильному предсказанию в позиции b. Если счетчик 0 был в самом начале равен 3, тогда он будет колебаться между состояниями 2 и 3, что приведет к неправильному предсказанию в позиции a. Худший случай - это когда его состояние было в начале равно 2. Он будет колебаться между 1 и 2, что приведет к неправильному предсказанию в обоих позициях. (Это аналогично худшему случаю для PPlain, объясненному выше). Какую из этих ситуаций мы получим, зависит от истории элемента BTB до того, как он был зарезервирован. Мы не можем это проконтролировать, так как элементы BTB выбираются случайно.

В принципе, можно избежать худший случай, если задать в самом начале задать специально спроектированную последовательность, чтобы поместить счетчик в желаемое состояние. Такой подход трудно порекомендовать, так как он значительно усложняет код и любая информация, которую мы поместим в счетчик будет, похоже, потеряна при следующем прерывании или переключении задач.

22.2.7 Полностью случайные последовательности (PMMX, PPro, PII и PIII)

Мощная способность распознавания последовательностей имеют небольшой изъян в случае полностью случайных нерегулярных последовательностей.

Следующая таблица перечисляет эспериметально найденную часть неверных предсказаний для полностью случайных нерегулярных последовательностей.

часть переходов/не-переходов	часть неверных предсказаний
0.001/0.999	0.001001
0.01/0.99	0.0101
0.05/0.95	0.0525
0.10/0.90	0.110
0.15/0.85	0.171

0.20/0.80	0.235
0.25/0.75	0.300
0.30/0.70	0.362
0.35/0.65	0.418
0.40/0.60	0.462
0.45/0.55	0.490
0.50/0.50	0.500

Часть неправильных предсказаний немного вы, чем если бы это было без распознавания последовательностей, потому что процессор пытается найти повторяющиеся блоки там, где нет никакой закономерности.

22.2.8 Маленькие циклы (PMMX)

Предсказание переходов ненадежно в маленьких циклах, где у механизма распознавания последовательностей нет времени, чтобы обновить свои данные перед новой встречей с переходом. Это означает, что простые последовательности, которые в обычном случае были бы предсказаны совершенно, не будут распознанны. Также некоторые последовательности, которые в обычном случае не были бы распознаны, будут предсказаны в маленьком цикле. Например, цикл, который всегда повторяется 6 раз, имел бы последовательность 111110 для инструкции ветвления в начале цикла. В этой последовательности было бы одно или два неправильных предсказаний за итерацию, но в маленьком цикле не будет ни одного. То же самое касается цикла, который повторяется 7 раз. С другим количеством повторений предсказуемость будет хуже в маленьких циклах, чем обычно. Это означает, что количество повторений, равное 6 или 7, более предпочтительно для маленьких циклов. Вы можете развернть цикл, чтобы сделать его больше.

Чтобы узнать, подпадает ли цикл на PMMX под действие вышеизложенных правил, вы можете сделать следующее: подсчитайте количество инструкций в цикле. Если количество равно 6 или меньше, тогда цикл является "маленьким". Если у вас больше 7 инструкций, тогда у вас может быть достаточно твердая уверенность в том, что распознавание последовательностей будет работать нормально. Довольно странно, что количество тактов, которое требуется для выполнения инструкций, не играет роли, как и спариваемость инструкций и вызываемые ими задержки. Сложные целочисленные инструкции при подсчете не учитываются. В цикле может быть много таких инструкций, и он будет себя вести как "маленький". Сложная целочисленная инструкция - это не спариваемая целочисленная инструкция, которая всегда требует больше одного такта. Сложные инструкции плавающей запятой и инструкции MMX идут одна за одну. Обратите внимание, что это правило выведено эвристическим путем и не 100% надежно. В важных случаях вы можете провести свое собственное тестирование. Вы можете использовать датчик качества с номером 35 в PMMX, чтобы подсчитать количество неправильно предсказанных переходов. Результаты тестов также не могут быть полностью надежны, потому что предсказания переходов зависят от истории элемента BTB до резервирования.

Маленькие циклы на PPro, PII и PIII предсказываются нормально, и каждое повторение занимает минимум два цикла.

22.2.9 Косвенные переходы и вызовы (PMMX, PPro, PII и PIII) Распознавание последовательностей не работает в случае косвенных переходов и вызовов, а BTB не может запомнить больше одной цели для косвенного вызова. Предсказывается переход к той же цели, что и в прошлый раз. **22.2.10 JECXZ и LOOP (PMMX)** На PMMX распознавание последовательностей не работает с этими двумя инструкциями, просто предсказывается то, что произошло в последний раз. Эти две инструкции следует избегать в критическом коде для PMMX (на PPro, PII и PIII) предсказания осуществляются с помощью распознавания последовательностей, но специальные инструкции циклов все равно ужасны по сравнению с DEC ECX / JNZ). **22.2.11 Возвраты (PMMX, PPro, PII и PIII)**

У процессоры PMMX, PPro, PII и PIII есть стековый буфер возвратов (RSB), который используется для предсказывания инструкций возвратов. RSB работает как FIFO (первым-вошел-первым-вышел)

буфер. Каждый раз, когда выполняется инструкция вызова, соответствующий адрес возврата помещается в RSB. И каждый раз, когда выполняется инструкция возврата, адрес возвращения вынимается из RSB и используется для предсказывания возврата. Этот механизм обеспечивает корректное предсказание этих инструкций, когда одна и та же подпрограмма вызывается из разных мест.

Чтобы этот механизм работал правильно, вы должны убедиться, что все вызовы и возвраты соответствуют. Никогда не выходите из подпрограммы без возвраты и никогда не используйте возврат в качестве косвенного перехода, если скорость играет решающую роль.

RSB может содержать до 4 элементов в PMMX, шестнадцать в PPro, PII и PIII. В случае, когда RSP пуст, инструкция возврата предсказывается также, как и косвенный переход, то есть ожидается переход туда, куда он был совершен в последний раз.

На PMMX, когда подпрограммы вложены друг в друга более, чем на 4 уровня, все возвраты, кроме 4 самых внутренних уровней, поддерживаемых PMMX, используют простой механизм предсказания, пока не происходит новых вызовов. Инструкция возврата, находящаяся в RSB, также занимает один элемент BTB. Четыре элемента в RSB PMMX'а кажутся не весть чем, но это, вероятно, эффективно. Подрограммы, вложенные более, чем на 4 уровня, не являются чем-то необычным, но только внутренние уровни максимально эффективны в плане скорости, не считая возможности рекурсивных процедур.

На PPro, PII и PIII, когда подпрограммы вложены глубже, чем на 16 уровней, внутренние 16 уровней используют RSB, в то время как все последующие возвраты из внешних уровней предсказываются неверно, поэтому рекурсивные процедуры не следует делать вложенными более, чем на 16 уровней.

22.2.12 Статическое предсказание (PMMX)

Инструкция передачи управления, которая не встречалась ранее, или которая не находится в BTB всегда предсказывается как невыполняющаяся на PMMX. Не играет роли, куда совершается переход: вперед или назад.

Инструкция перехода не получит элемент BTB, если она постоянно невыполняется. Как только она выполнится, она попадет в BTB и будет оставаться там вне зависимости от того, сколько раз она не выполнится в дальнейшем. Инструкция передачи управления может исчезнуть из BTB только, если она была выдавлена другой инструкцией передачи управления.

Любая инструкция передачи управления, которая переходит на адрес, непосредственно следующий за ней же, не получает элемент в BTB.

Пример:

```
        JMP SHORT LL
```

LL:

Эта инструкция никогда не поулчит элемент в BTB и поэтому всегда будет предсказываться неправильно.

22.2.13 Статическое предсказывание (PPro, PII и PIII)

На PPro, PII и PIII инструкция передачи управления, которая не встречалась ранее или которой нет в BTB, предсказывается как невыполняющаяся, если она указывает вперед, и как выполняющаяся, если она указывает назад (например, цикл). Статическое предсказываение занимает больше времени, чем динамическое предсказание на этих процессорах.

Если похоже, что ваш код не прокэширован, то предпочтительнее, чтобы наиболее часто встречающаяся инструкция перехода не выполнялась, чтобы улучшить доставку.

22.2.14 Закрытые переходы (PMMX)

На PMMX существует риск, что две инструкции передачи управления разделят один и тот же элемент BTB, если они находятся слишком близко друг к другу. Очевидным результатом станет то, что они всегда будут предсказываться неправильно.

Элемент BTB для инструкции передачи управления определяется по битам 2-31 адреса последнего байта инструкции. Если две инструкции передачи управления находятся так близко друг от друга, что отличаются только битами 0-1 этих адресов, то у нас есть проблема разделяемого элемента BTB. Пример:

```
        CALL    P
        JNC     SHORT L
```

Если последний байт инструкции CALL и последний байт инструкции JNC находятся внутри того же слова памяти, то мы получаем потери. Вы должны взглянуть на сгенерированный ассемблерный листинг, чтобы увидеть, разделены ли эти два адреса границей двойного слова или нет (граница двойного слова - это адрес, который делится на 4).

Есть несколько путей решить эту проблему:

• Переместить код чуть вверх или вниз в памяти, чтобы между двумя адресами была граница двойного слова.
• Изменить короткий переход на ближний переход, чтобы конец инструкции сместился чуть вниз.
• Поместить какую-либо инструкцию между CALL и JNC. Это самый простой метод и единственный, если вы не знаете, где находятся границы двойного слова, например если ваш сегмент не выровнен по двойному слову или потому что код смещается то вверх, то вниз в результате изменений в предшествущем коде.

```
        CALL    P
        MOV     EAX,EAX         ; добавим два байта, чтобы быть спокойными
        JNC     SHORT L
```

Если вы хотите избежать проблем и на PPlain, тогда поместите вместо MOV два NOP'a, чтобы предотвратить спаривание.

Инструкция RET особенно беззащитна перед этой проблемой, потому она всего один байт длиной.

```
        JNZ     NEXT
        RET
```

Здесь вам может потребоваться вставить три байта:

```
        JNZ     NEXT

        NOP
        MOV     EAX,EAX
        RET
```

22.2.15 Последовательные вызовы или возвраты (PMMX)

Возникают потери, когда первая пара инструкций, следующая за меткой-целью вызова, содержит другой вызов или возврат, следующий сразу после другого возврата.

Пример:

```
FUNC1   PROC    NEAR
        NOP             ; избегаем вызова после вызова
        NOP
        CALL    FUNC2
        CALL    FUNC3
        NOP             ; избегаем возврата после возврата

        RET
FUNC1   ENDP
```

Требуется два NOP перед 'CALL FUNC2', потому что один NOP будет спариваться с CALL. Одного NOP достаточно перед RET, потому что эта инструкция неспариваема. Не требуется NOP между двумя инструкциями CALL, потому что нет потерь при вызове после возврата (на PPlain вам потребуется здесь два NOP).

Подобные потери последовательных вызовах возникают только, когда одни и те же подпрограммы вызываются из более, чем одного места (вероятно, это происходит из-за обновления RSB). Последовательные возвраты всегда приводят к потерям. Иногда возникает небольшая задержка при переходе после вызова, но нет потерь при возврате после вызова, вызова после возврата, перехода, вызова или возврата после перехода или перехода после возврата.

22.2.16 Последовательные переходы (PPro, PII и PIII) Переход, вызов или возврат не может выполниться в первый такт после предыдущего перехода, вызова или возврата. Поэтому последовательные переходы занимают по два такта на переход, и вы можете захотеть убедиться, что у процессора есть чем еще заняться. В силу определенных причин цикл требует не менее двух тактов на итерацию на этих процессорах. **22.2.17 Проектирование предсказуемых переходв (PMMX, PPro, PII и PIII)**

Многоветвенные переходы (реализующие высокоуровневые выражения switch/case) реализуются как список адресов переходов или как дерево инструкций переходов. Так как косвенные переходы предсказываются плохо, то последний метод может быть более предпочтителен, если дерево будет представлено в виде хорошо предсказуемой последовательности и у вас есть достаточное количество элементов BTB. В случае, если решите использовать предыдущий метод, рекомендуется, чтобы адреса переходов были помещены в сегмент данных.

Вы можете захотеть реорганизовать ваш код так, чтобы последовательности переходов, которые не предсказываются совершенно, были заменены другими последовательностями. Предположите, например, цикл, который выполняется 20 раз. Условный переход внизу цикла выполнится 19 раз и нет - каждый 20 раз. Эта последовательность регулярна, но не будет узнана механизмом распознавания последовательностей, поэтому невыполнение данной инструкции будет всегда предсказываться неправильно. Вы можете сделать два вложенных цикла по 4 и 5 повторений или развернуть цикл в четыре раза и позволить ему выполниться 5 раз, чтобы были только распознаваемые последовательности. Этот вид сложных схем оправдывает себя только на PPro, PII и PIII, где неверное предсказание стоит слишком дорого. Для циклов с большим количеством повторения особо смысла бороться с одним неправильным предсказанием нет.

22.3. Избегание переходов (все процессоры)

Есть много причин, почему вы можете захотеть снизить количество переходов, вызовов и возвратов:

• неверные предсказания стоят очень дорого
• в зависимости от процессора могут возникнуть различные потери при последовательных вызовах
• инструкции перехода могут выталкивать друг друга из буфера переходов
• возврат занимает 2 такта на PPlain и PMMX, вызовы и возвраты генерируют 4 мопа на PPro, PII и PIII.
• На PPro, PII и PIII доставка инструкций может быть задержена после перехода (глава 15), а вывод из обращения может быть менее эффективным для совершенных переходов, чем для других мопов (глава 18).

Вызовы и возвраты можно избежать, если заменить маленькие процедуры макросами. Во многих случаях можно снизить количество переходов, перегруппировав ваш код. Например, переход на переход можно заменить переходом к конечной цели. В некоторых случаях это возможно сделать даже с условными переходами, если условие то же самое или известно. Переход на возврат можно заменить возвратом. Если вы хотите устранить возврат на возврат, тогда вы не должны

манипулировать стековым указателем, потому что это будет создавать помехи механизму предсказания, основывающимся на RSP. Вместо этого вы можете заменить предыдущий вызов на переход. Например 'CALL PRO1 / RET' можно заменить на 'JMP PRO1', если PRO1 заканчивается тем же видом RET.

Вы можете также устранить переходом, дублируя код, на который совершается переход. Это может быть полезным, если у вас есть двухнаправленная ветвь внутри цикла или до возврата.

Пример:

```
A:      CMP     [EAX+4*EDX],ECX
        JE      B
        CALL    X
        JMP     C
B:      CALL    Y
C:      INC     EDX
        JNZ     A
        MOV     ESP, EBP
        POP     EBP
        RET
```

Переход на C можно устранить, продублировав эпилог цикла:

```
A:      CMP [EAX+4*EDX],ECX

        JE      B
        CALL    X
        INC     EDX
        JNZ     A
        JMP     D
B:      CALL    Y
C:      INC     EDX
        JNZ     A
D:      MOV     ESP, EBP
        POP     EBP
        RET
```

Наиболее часто выполняющийся переход здесь должен быть первым. Переход на D находится вне цикла и поэтому менее критичен. Если переход выполняется так часто, что его нужно тоже оптимизировать, то замените его тремя инструкциями, которые следуют за D.

22.4. Избегание условных переходов, используя флаги (все процессоры)

Самое главное - избавиться от условных переходов, особенно если они плохо предсказуемы. Иногда возможно получить тот же эффект, искусно манипулируя битами и флагами.

```
CDQ
XOR EAX,EDX
SUB EAX,EDX
```

(На PPlain и PMMX используйте 'MOV EDX,EAX / SAR EDX,31' вместо CDQ).

Флаг переноса особенно полезен для подобных трюков:

• Установка флага переноса, если значение равно нулю: CMP [VALUE],1
• Установка флага переноса, если значение не равно нулю: XOR EAX,EAX / CMP EAX,[VALUE]
• Повышение счетчика на 1, если установлен флаг переноса: ADC EAX,0
• Установка бита каждый раз, когда установлен флаг переноса: RCL EAX,1
• Генерация битовой маски, если установлен флаг переноса: SBB EAX,EAX
• Установка бита при определенном условии: SETcond AL
• Установка всех бит при определенном условии: XOR EAX,EAX / SETNcond AL / DEC EAX (помните о том, что в последнем примере условие должно быть сформулировано наоборот).

Следующий пример находит меньшее из двух беззнаковых чисел: if (b < a) a = b

```
SUB EBX,EAX
```

```
SBB ECX,ECX
AND ECX,EBX
ADD EAX,ECX
```

А вот пример, как выбрать между двумя числами: if (a != 0) a = b; else a = c;

```
CMP EAX,1
SBB EAX,EAX
XOR ECX,EBX
AND EAX,ECX
XOR EAX,EBX
```

Стоит или нет применять подобные трюки зависит от того, насколько предсказумы условные переходы, есть или нет возможности под спариванию инструкций, есть или нет рядом другие переходы, из-за которых могут возникнуть потери.

22.5. Замена условных переходов услоными перемещениями (PPro, PII и PIII)

У процессоров PPro, PII и PIII есть инструкции условного перемещения данных, созданных специально для избежания переходов, потому что неверное предсказание перехода стоит на этих процессорах слишком дорого. Есть инструкции условного перемещения данных и для целочисленных регистров и для регистров плавающей запятой. В коде, который будет выполняться только на этих процессорах, вы можете заменить плохо предсказуемые переходы инструкциями условного перемещения там, где это возможно. Если вы хотите, чтобы ваш код выполнялся на всех процессорах, вы можете сделать две версии самого критичного кода, одну для процессоров, которые поддерживают инструкции условного перемещнния и одну для тех, которые не поддерживаются (смотрите главу 27.10, чтобы узнать, как определить, поддерживает ли процессор условные переходы).

Потери из-за неправильного предсказания перехода могут быть столько высоки, что имеет смысл заменить их условными перемещениями, даже если это будет стоить несколько дополнительных инструкций. Но у инструкций условного перехода есть один недостаток: они делают цепочки зависимости длинее. Условное перемещение ждет готовности обоих операндов-регистров, даже если требуется только один из них. Условное перемещение ждет готовности трех операндов: флаг условия и еще два операнда перемещения. Вы должны решить, может ли один из этих трех операндов привести к задержкам из-за ошибок в работе с кешем или из-за цепочек зависимости. Если флаг условия доступе задолго до операндов перемещения, вы можете так же использовать инструкции перехода, потому что возможное неправильное предсказание может быть преодолено, пока ожидается готовность операндов перемещения. В тех ситуация, где вам нужно долго ждать операнд перемещения, который потом еще может и не понадобиться, инструкция перехода будет быстрее, чем условный переход, несмотря на потери, связанные с возможным неправильным предсказанием. Обратная ситуация возникает, когда флаг условия задерживается, а оба операнда перемещения доступны ранее. В этой ситуации условный переход более предпочтителен, чем инструкция перехода, если вероятно неправильно предсказание.

Оптимизация для процессоров семейства Pentium: 23. Уменьшение размера кода (все процессоры)

Автор: Агнер Фог, пер. Aquila

Дата: 2012-11-24

Как было объяснено в главе 7, размер кода кэша равен 8 или 16 килобайтам. Если у вас есть подозрение, что критические части кода не поместятся в кэш, тогда вы можете подумать о том, чтобы уменьшить их размер.

32-х битный код обычно больше, чем 16 битный код, потому что адреса и константы занимают 4 байта в 32-х битном режиме, а 16-ти битном режиме только два. Тем не менее, 16-ти битный код другие потери, такие как префиксы и проблемы с соседнеми словами памяти (смотри главу 10.2). Некоторые другие методы для уменьшения размера кода обсуждаются ниже.

Оба адреса перехода, адреса данных и константы занимают меньше места, если их значение находится между -128 до +127.

Для адресов переходов это означает, что короткие переходы занимают два байта, в то время как переходы более чем 127 байт занимают 5 байтов, если переход безусловный и 6 байтов, если условный.

Таким же образом адреса данных занимают меньше места, если они могут быть выраженны как указатель в интервале от -128 до +127

Пример:

```
MOV EBX,DS:[100000] / ADD EBX,DS:[100004] ; 12 байт
```

Уменьшаем размер:

```
MOV EAX,100000 / MOV EBX,[EAX] / ADD EBX,[EAX+4] ; 10 байт
```

Преимущество использования указателя становится еще более очевидным, если вы используете его много раз. Хранение данных в стеке и использование EBP или ESP в качестве указателя сделает ваш код меньше, чем если бы вы использовали абсолютные адреса, если только, конечно, ваши данные в пределах +/-127 байтах указателя. Использование PUSH и POP для записи и чтения временных данных еще короче.

Константы могут также занимать меньше места, если они между -128 и +127. Большинство инструкций с числовыми операндами имеют короткую форму, где операнд - это один байт со знаком.

Примеры:

```
PUSH 200      ; 5 байт
PUSH 100      ; 2 байт

ADD EBX,128   ; 6 байт
SUB EBX,-128  ; 3 байт
```

Самая важная инструкция с числовым операндом, у которой нет короткой формы, это MOV.

Примеры:

```
MOV EAX, 0              ; 5 байт
```

Можно заменить на:

```
XOR EAX,EAX             ; 2 байта
```

И

```
MOV EAX, 1              ; 5 байтов
```

Можно заменить на:

```
XOR EAX,EAX / INC EAX   ; 3 байта
```

или:

```
PUSH 1 / POP EAX        ; 3 байта
```

И

```
MOV EAX, -1             ; 5 байта
```

Можно заменить на:

```
OR EAX, -1              ; 3 байта
```

Если один и тот же адрес или константа используется несколько раз, вы можете загрузить ее в регистр. MOV с 4-х байтным числовым операндом иногда можно заменить на арфметическую инструкцию, если известно значение регистра до MOV.

Пример:

```
MOV     [mem1],200              ; 10 байтов
MOV     [mem2],200              ; 10 байтов
MOV     [mem3],201              ; 10 байтов
MOV     EAX,100                 ;  5 байтов
MOV     EBX,150                 ;  5 байтов
```

Предполагая, что mem1 и mem3 находятся в пределах -128/127 байтов от em2, это можно изменить на:

```
MOV     EBX, OFFSET mem2        ;  5 байтов
MOV     EAX,200                 ;  5 байтов

MOV     [EBX+mem1-mem2],EAX     ;  3 байта
MOV     [EBX],EAX               ;  2 байта
INC     EAX                     ;  1 байт
MOV     [EBX+mem3-mem2],EAX     ;  3 байта
SUB     EAX,101                 ;  3 байта
LEA     EBX,[EAX+50]            ;  3 байта
```

Остерегайтесь задержек AGI в инструкции LEA (для PPlain и PMMX).

Также стоит учитывать то, что разные инструкции имеют разную длину. Следующие инструкции занимают только один байт и поэтому очень привлекательны: PUSH reg, POP reg, INC reg32, DEC reg32. INC и DEC с 8-ми битовыми регистрами занимают 2 байта, поэтому 'INC EAX' короче, чем 'INC AL'.

'XCHG EAX,reg' также однобайтовая инструкция и поэтому занимает меньше места, чем 'MOV EAX,reg', но это медленнее.

Некоторые инструкции занимают на один байт меньше, когда они используют аккумулятор, а не другой регистр.

Пример:

```
MOV EAX,DS:[100000]  меньше, чем  MOV EBX,DS:[100000]
ADD EAX,1000         меньше, чем  ADD EBX,1000
```

Инструкции с указателями занимают на один байт меньше, чем когда они используют адресацию по базе (не ESP) со сдвигом, а не косвенную адресацию с масштабированием, или и то, и другое вместе, или ESP в качестве базы.

Примеры:

```
MOV EAX,[array][EBX]  меньше, чем  MOV EAX,[array][EBX*4]
```

```
MOV EAX,[EBP+12]      меньше, чем  MOV EAX,[ESP+12]
```

Инструкции с EBP в качестве базы без смещения занимают на один байто больше, чем при использовании других регистров:

```
MOV EAX,[EBX]    меньше чем  MOV EAX,[EBP], но
MOV EAX,[EBX+4]  такого же размера, как и MOV EAX,[EBP+4]
```

Также адресация со сдвигом бывает выгоднее, чем адресация с масштабированием:

```
LEA EAX,[EBX+EBX]  короче, чем  LEA EAX,[2*EBX]
```

Оптимизация для процессоров семейства Pentium: 24. Работа с плавающей запятой (PPlain и PMMX)

Автор: Агнер Фог, пер. Aquila

Дата: 2012-11-24

Инструкции плавающей запятой не могут спариваться так, как это делают целочисленные инструкции, не считая некоторых случаев, определяемых следующими правилами:

• первая инструкция (выполняющаяся в U-конвеере) должна быть FLD, FADD, FSUB, FMUL, FDIV, FCOM, FCHS или FABS.
• вторая инструкция (в V-конвеере) должна быть FXCH.
• инструкция, следующая за FXCH, должна быть инструкцией плавающей запятой, иначе FXCH спарится несовершенно и займет лишний такт.

Это особенное спаривание играет важную роль, что вкратце будет объяснено.

Хотя, как правило, инструкции плавающей запятой не могут спариваться, многие из них конверизуются, то есть одна инструкция может начать выполнение до того, как будет закончена предыдущая инструкция.

Пример:

```
FADD ST(1),ST(0)   ; такты 1-3
FADD ST(2),ST(0)   ; такты 2-4
FADD ST(3),ST(0)   ; такты 3-5
FADD ST(4),ST(0)   ; такты 4-6
```

Очевидно, что выполнение двух инструкциий не может пересекаться, если второй инструкции нужен результат первой. Так как почти все инструкции плавающей запятой работают с вершиной стека регистров ST(0), возможностей сделать их независимыми друг от друга не очень много. Решение этой проблемы состоит в переименовании регистров. Инструкция FXCH в реальности не обменивает содержимое двух регистров, оно только меняет их имена. Инструкции, которые помещают или извлекают значение из стека регистров также работают с помощью переименования. Переименование регистров на Pentium настолько хорошо оптимизировано, что можно переименовать использующийся регистр. Переименование регистров никогда не вызывает задержек - возможно даже переименовать регистр более чем один раз за такт, например, когда вы спариваете FLD или FCOMPP с FXCH.

Правильно используя инструкции FXCH, вы можете создать условия, чтобы инструкции плавающего кода были более независимыми друг от друга.

Пример:

```
FLD     [a1]    ; такт 1
FADD    [a2]    ; такт 2-4
FLD     [b1]    ; такт 3
FADD    [b2]    ; такт 4-6
FLD     [c1]    ; такт 5
FADD    [c2]    ; такт 6-8
FXCH    ST(2)   ; такт 6
FADD    [a3]    ; такт 7-9
FXCH    ST(1)   ; такт 7
FADD    [b3]    ; такт 8-10

FXCH    ST(2)   ; такт 8
FADD    [c3]    ; такт 9-11
FXCH    ST(1)   ; такт 9
FADD    [a4]    ; такт 10-12
FXCH    ST(2)   ; такт 10
```

```
FADD    [b4]    ; такт 11-13
FXCH    ST(1)   ; такт 11
FADD    [c4]    ; такт 12-14
FXCH    ST(2)   ; такт 12
```

В вышеприведенном примере мы создали три независимые ветви. Каждый FADD занимает 3 такта, поэтому мы можем каждый такт начинать выполнение нового FADD. Когда мы начали выполнение ветви 'a', у нас есть время, чтобы начать выполнение двух новых инструкций FADD в ветвях 'b' и 'c' до возвращения к ветви 'a', поэтому каждый третий FADD принадлежит той же ветви. Мы используем инструкции FXCH каждый раз, когда необходимо, чтобы ST(0) стал равен регистру, который принадлежит к желаемой ветви. Как вы можете видеть из примера, это образует регулярную последовательность, но уясните хорошо, что инструкции FXCH повторяются с периодом, равным двум, в то время как у ветвей период равен трем. Это может немного смущать, поэтому вам следует проработать этот пример, чтобы понять где находится какой из регистров.

Все версии инструкций FADD, FSUB, FMUL и FILD занимают три такта и конвееризуются, поэтому вышеописанный метод можно применять и с этими инструкциями. Использование переменных в памяти не отнимает больше времени, чем использование регистров, если переменная в памяти находится в кэше первого уровня и правильно выравнена.

Вы уже, наверное, что у всех правил есть исключения, и у вышеизложенного они тоже есть: вы не можете начать выполнение инструкции FMUL на следующий такт после другой инструкции FMUL, потому что FMUL не может спариваться совершенно. Рекомендуется, чтобы вы поместили другую инструкцию между двумя FMUL'ами.

Пример:

```
FLD     [a1]    ; такт 1
FLD     [b1]    ; такт 2
FLD     [c1]    ; такт 3
FXCH    ST(2)   ; такт 3
FMUL    [a2]    ; такты 4-6
FXCH            ; такт 4
FMUL    [b2]    ; такты 5-7    (задержка)
FXCH    ST(2)   ; такт 5
FMUL    [c2]    ; такты 7-9    (задержка)
FXCH            ; такт 7
FSTP    [a3]    ; такты 8-9
FXCH            ; такт 10     (неспарены)

FSTP    [b3]    ; такты 11-12
FSTP    [c3]    ; такты 13-14
```

Здесь у вас есть задержка между FMUL [b2] и между FMUL [c2], потому что другая FMUL началась в предыдущий такт. Вы можете улучшить этот код, поместив инструкции FLD между FMUL'ами.

```
FLD     [a1]    ; такт 1
FMUL    [a2]    ; такт 2-4
FLD     [b1]    ; такт 3
FMUL    [b2]    ; такт 4-6
FLD     [c1]    ; такт 5

FMUL    [c2]    ; такт 6-8
FXCH    ST(2)   ; такт 6
FSTP    [a3]    ; такт 7-8
FSTP    [b3]    ; такт 9-10
FSTP    [c3]    ; такт 11-12
```

В других случаях вы можете поместить FADD, FSUB или что-нибудь еще между FMUL'ами, чтобы избежать задержек.

Инструкции плавающей запятой, чье выполнение пересекается, требуют, конечно, чтобы у вас были независимые ветви, выполненине которых вы можете совместить. Если у вас есть только одна большая формула, которую надо выполнить, тогда вы можете посчитать части формулы

параллельно, чтобы достигнуть цели. Если, например, вы хотите добавить шесть чисел, тогда вы можете разделить операции на две ветви с тремя числами в каждой, и добавить две ветви в конце:

```
FLD     [a]     ; такт 1
FADD    [b]     ; такты 2-4
FLD     [c]     ; такт 3
FADD    [d]     ; такты 4-6
FXCH            ; такт 4
FADD    [e]     ; такты 5-7
FXCH            ; такт 5
FADD    [f]     ; такты 7-9    (задержка)
FADD            ; такты 10-12  (задержка)
```

Здесь у нас есть задержка в один такт перед FADD [f], потому что она ожидает результата выполнения FADD [d] и задержка в два такта перед последним FADD, потому что она ожидает результата FADD [f]. Более поздняя задержка может быть спрятана путем заполнения ее несколько целочисленными инструкциями, но с первой задержкой это не получится, так как целочисленная инструкция в этом месте приведет к тому, что FXCH будет спариваться несовершенно.

Первую задержку можно избежать, создав три ветви вместо двух, но это будет стоить дополнительно FLD, поэтому мы ничего не выиграем, прибегнув к этому варианту, если только у нас нет восьми чисел, которые нужно сложить.

Выполнение не все инструкций плавающей запятой может пересекаться. И выполнение некоторых инструкций плавающей запятой лучше сочетается с целочисленными инструкциями. Например, инструкция FDIV занимает 39 тактов. Во все, кроме первого такта, выполнение этой инструкции может пересекаться с целочисленными инструкциями, но только в последние два такта она может пересекаться с инструкциями плавающей запятой.

Пример:

```
FDIV         ; такт 1-39  (U-конвеер)
FXCH         ; такт 1-2   (V-конвеер, несовершенное спаривание)
SHR EAX,1    ; такт 3     (U-конвеер)
INC EBX      ; такт 3     (V-конвеер)
CMC          ; такт 4-5   (не спаривается)
FADD [x]     ; такт 38-40 (U-конвеер, ждет, пока FPU не освободится)
FXCH         ; такт 38    (V-конвеер)
FMUL [y]     ; такт 40-42 (U-конвеер, ждет результат FDIV)
```

Первый FXCH спаривается с FDIV, но занимает дополнительный такт, потому что за ней не следует инструкция плавающей запятой. Пара SHR / INC начинает свое выполнение до того, как будет закончено выполнение FDIV, но была вынуждена подождать, пока свое выполнение закончит FXCH.

Если у вас нет ничего, что поместить после инструкции плавающей запятой, которая может выполняться одновременно с целочисленной инструкцией, (FDIV, FSQRT), вы можете поместить чтение значение из какой-нибудь переменной в памяти, которое может вам понадобиться в дальнейшем, чтобы она на 100% была в кэше.

Пример:

```
FDIV    QWORD PTR [EBX]
CMP     [ESI],ESI
FMUL    QWORD PTR [ESI]
```

Здесь мы загружаем загружаем значение в [ESI] в кэш, в то время как FDIV вычисляется (результат самой операции сравнения нам не важен).

В главе 28 приведен полный список инструкций плавающей запятой, и с чем они могут спариваться.

Никаких потерь при использовании переменных в памяти в инструкциях плавающей запятой, потому что модуль арифметических вычислений на один шаг дальше в конвеере, чем модуль чтения. Однако при сохранении данных может случиться задержка, аналогичная задержке AGI:

выполнение инструкции FST или FSTP с переменной в памяти в качестве операнда занимает два такта, но данные должны быть готовы в предыдущем такте, поэтому будет задержка в один такт, если значение, которое нужно сохранить не будет готово еще в предыдущем такте.

Пример:

```
FLD     [a1]    ; такт 1
FADD    [a2]    ; такт 2-4
FLD     [b1]    ; такт 3
FADD    [b2]    ; такт 4-6
FXCH            ; такт 4
FSTP    [a3]    ; такт 6-7
FSTP    [b3]    ; такт 8-9
```

FSTP задерживается на один такт, потому что результат FADD не был готов в предыдущем такте. Во многих случаях вы не можете скрыть этот тип задержек бзе организования вашего кода с плавающей запятой в четыре ветви или помещения внутрь каких-то целочисленных инструкций. Два такта на стадии выполнения инструкции FST(P) не могут спариваться или пересекаться с любой другой последующей инструкцией.

Инструкции с целочисленными операндами, такими как FIADD, FISUB, FIMUL, FIDIV, FICOM можно разделить на простые операции, чтобы улучшить пересекаемость выполнений инструкций.

Пример:

```
FILD    [a]     ; clock cycle 1-3
FIMUL   [b]     ; clock cycle 4-9
```

Разделить на:

```
FILD    [a]     ; такты 1-3
FILD    [b]     ; такты 2-4
FMUL            ; такты 5-7
```

В этом примере вы экономите два такта, пересекая выполнение двух инструкций FILD.

Оптимизация для процессоров семейства Pentium: 25. Оптимизация циклов (все процессоры)

Автор: Агнер Фог, пер. Aquila

Дата: 2012-11-24

Анализируя свои программы, вы можете увидеть, что больше всего ресурсов пожирают внутренние циклы. Используя язык ассемблера можно существенно оптимизировать их. Остальную часть программы можно оставить написанной на языке высокого уровня.

Во всех следующих примерах предполагается, что все данные находятся в кэше первого уровня. Если скорость ограничивается из-за того, что данные загружаются в кэш, нет никакого смысла оптимизировать инструкции. Лучше сконцентрируйтесь на правильной организации данных (глава 7).

25.1 Циклы в PPlain и PMMX

Как правило цикл содержит счетчик, определяющий сколько раз он должен повториться, и достаточно часто еще массив данных, в один элемент которого записываются данные или считываются из него каждое повторение.

Эта процедура на C может выглядеть так:

```
void ChangeSign (int * A, int * B, int N) {
  int i;
  for (i=0; i<N; i++) B[i] = -A[i];}
```

Переводя ее на ассемблер мы могли бы написать эту процедуру следующим образом:

Пример 1.1:

```
_ChangeSign PROC NEAR
        PUSH    ESI
        PUSH    EDI
A       EQU     DWORD PTR [ESP+12]
B       EQU     DWORD PTR [ESP+16]
N       EQU     DWORD PTR [ESP+20]
        MOV     ECX, [N]
        JECXZ   L2

        MOV     ESI, [A]
        MOV     EDI, [B]
        CLD
L1:     LODSD
        NEG     EAX
        STOSD
        LOOP    L1
L2:     POP     EDI
        POP     ESI
        RET                     ; нет дополнительного pop, если объявлено
                                ; соглашение о вызове _cdecl
_ChangeSign     ENDP
```

Это выглядит как достаточно красивое решение, но оно не самое оптимальное, потому что оно использует медленные неспариваемые инструкции. Каждая итерация занимает 11 тактов при условии, что все данные находятся в кэше первого уровня.

Использование только спариваемых инструкций (PPlain и PMMX)

Пример 1.2:

```
        MOV     ECX, [N]
        MOV     ESI, [A]
        TEST    ECX, ECX
```

```
        JZ      SHORT L2
        MOV     EDI, [B]
L1:     MOV     EAX, [ESI]      ; u
        XOR     EBX, EBX        ; v (спаривается)
        ADD     ESI, 4          ; u
        SUB     EBX, EAX        ; v (спаривается)
        MOV     [EDI], EBX      ; u
        ADD     EDI, 4          ; v (спаривается)
        DEC     ECX             ; u

        JNZ     L1              ; v (спаривается)
L2:
```

Здесь мы используем только спариваемые инструкции, и так их располагаем, что они все спариваются. Теперь они занимают только 4 такта за итерацию. Мы можем получить ту же самую скорость не разделяя инструкцию NEG, но другие неспариваемые инструкции все же стоит разделить.

Использования одного регистра как счетчика и индекса

Пример 1.3:

```
        MOV     ESI, [A]
        MOV     EDI, [B]

        MOV     ECX, [N]
        XOR     EDX, EDX
        TEST    ECX, ECX
        JZ      SHORT L2
L1:     MOV     EAX, [ESI+4*EDX]        ; u
        NEG     EAX                     ; u
        MOV     [EDI+4*EDX], EAX        ; u
        INC     EDX                     ; v (спаривается)
        CMP     EDX, ECX                ; u
        JB      L1                      ; v (спаривается)
L2:
```

Использование одного регистра как счетчика и индеска уменьшает количество инструкций в теле цикла, но он по-прежнему занимает 4 такта, так как у нас две неспариваемые инструкции.

Пусть конечное значение счетчика равняется нулю (PPlain и PMMX)

Мы хотим избавиться от инструкции CMP в примере 1.3, сделав так, чтобы последнее значение счетчика равнялось нулю, и использовать флаг нуля как знак того, что цикл закончен (как мы сделали в примере 1.2). Один вариант заключается в том, чтобы исполнить цикл задом наперед, взяв последний элемент первым. Тем не менее, кэш данных оптимизирован для получения последних в прямом порядке, а не в обратном, поэтому если вероятны промахи кэша, вам следует задать значение счетчика -N и повышать его значение до нуля. Базовые регистры тогда должны указывать на конец массива, а не на его начало:

Пример 1.4:

```
        MOV     ESI, [A]
        MOV     EAX, [N]
        MOV     EDI, [B]
        XOR     ECX, ECX
        LEA     ESI, [ESI+4*EAX]        ; указывает на конец массива A
        SUB     ECX, EAX                ; -N
        LEA     EDI, [EDI+4*EAX]        ; указывает на конец массива B
        JZ      SHORT L2
L1:     MOV     EAX, [ESI+4*ECX]        ; u
        NEG     EAX                     ; u
        MOV     [EDI+4*ECX], EAX        ; u

        INC     ECX                     ; v (спаривается)
        JNZ     L1                      ; u
L2:
```

Однако цикл по-прежнему занимает 4 такта из-за плохой спариваемости. (Если адреса и размеры массива являются константами, мы можем сохранить два регистра). Теперь давайте посмотрим, как мы можем улучшить спариваемость.

Спаривание вычислений в цикле (PPlain и PMMX)

Мы можем улучшить спаривание, перемешав инструкции вычисления с инструкциями управления циклом. Если мы хотим поместить что-нибудь между 'INC ECX' и 'JNZ L1', это должно быть что-нибудь, что не влияет на флаг нуля. Инструкция 'MOV [EDI+4*ECX],EBX после 'INC ECX' сгенерирует задержку AGI, поэтому нам придется нужно быть более искусными:

Пример 1.5:

```
        MOV     EAX, [N]
        XOR     ECX, ECX
        SHL     EAX, 2                  ; 4 * N
        JZ      SHORT L3

        MOV     ESI, [A]
        MOV     EDI, [B]
        SUB     ECX, EAX                ; - 4 * N
        ADD     ESI, EAX                ; указывает на конец массива A
        ADD     EDI, EAX                ; указывает на конец массива B
        JMP     SHORT L2
L1:     MOV     [EDI+ECX-4], EAX        ; u
L2:     MOV     EAX, [ESI+ECX]          ; v (спаривается)
        XOR     EAX, -1                 ; u
        ADD     ECX, 4                  ; v (спаривается)
        INC     EAX                     ; u

        JNC     L1                      ; v (спаривается)
        MOV     [EDI+ECX-4], EAX
L3:
```

Здесь я использовал другой способ, чтобы посчитать отрицательное значение EAX: инвертирование всех битов и добавление еще одного. Причина, по которой я задействовал этот метод состоит в том, что я могу использовать хитрый трюк с инструкцией INC: INC не изменяет флаг переноса, в то время как ADD это делает. Я использую ADD вместо INC, чтобы повышать счетчик цикла и проверяю флаг переноса, а не флаг нуля. Тогда можно поместить 'INC EAX' между ними без вреда для флага переноса. Возможно вы подумали, что можно было бы использовать 'LEA EAX,[EAX+1]' вместо 'INC EAX', по крайней мере он не меняет никаких флагов, но инструкция LEA сгенерирует задержку AGI, поэтому это не лучшее решение. Обратите внимание, что трюк с инструкцией INC, которая не меняет флаг переноса, будет полезен только на PPlain и PMMX, так как на PPro, PII и PIII будут сгенерированы частичные задержки регистра флагов.

Здесь мне удалось добиться совершенного спаривания, и цикл теперь занимает только 3 такта. Хотите ли вы повышать значение счетчика цикла на 1 (как в примере 1.4) или на 4 (как в примере 1.5) - это дело вкуса, никакого влияния на время выполнения цикла это не окажет.

Пересечение времени выполнения одной операции с другой (PPlain и PMMX)

Метод, использованный в примере 1.5 подойдет далеко не во всех случаях, поэтому мы можем поискать другие способы улучшения спариваемости. Один из путей реорганизовать цикл - это сделать так, чтобы конец выполнения одной операции пересекался с началом выполнения другой. Я буду называть это свернутым циклом. У свернутого цикла в конце каждого цикла находится инструкция, выполнение которой будет закончено в следующем повторении. Фактически, в примере 1.5 последний MOV спаривается с первым. Мы исследуем этот метод поподробнее:

Пример 1.6:

```
        MOV     ESI, [A]
        MOV     EAX, [N]
        MOV     EDI, [B]
```

```
        XOR     ECX, ECX
        LEA     ESI, [ESI+4*EAX]          ; указывает на массив A
        SUB     ECX, EAX                  ; -N
        LEA     EDI, [EDI+4*EAX]          ; указывает на массив B
        JZ      SHORT L3
        XOR     EBX, EBX
        MOV     EAX, [ESI+4*ECX]
        INC     ECX
        JZ      SHORT L2
L1:     SUB     EBX, EAX                  ; u

        MOV     EAX, [ESI+4*ECX]          ; v (спаривается)
        MOV     [EDI+4*ECX-4], EBX        ; u
        INC     ECX                       ; v (спаривается)
        MOV     EBX, 0                    ; u
        JNZ     L1                        ; v (спаривается)
L2:     SUB     EBX, EAX
        MOV     [EDI+4*ECX-4], EBX
L3:
```

Здесь мы начинаем считывать второе значение до того, как сохранили первое, и это, конечно, улучшает возможности к спариванию. Инструкция 'MOV EBX,0' была помещена между 'INC ECX' и 'JNZ L1' не для того, чтобы улучшить спаривание, а для того, чтобы избежать задержку AGI.

Разворачивание цикла (PPlain и PMMX)

Один из самых часто использующихся способов улучшить спаривание - это сделать две операции на каждый проход, которых будет в два раза меньше. Это называется разворачиванием цикла:

Пример 1.7:

```
        MOV     ESI, [A]
        MOV     EAX, [N]
        MOV     EDI, [B]
        XOR     ECX, ECX
        LEA     ESI, [ESI+4*EAX]          ; указывает на конец массива A
        SUB     ECX, EAX                  ; -N
        LEA     EDI, [EDI+4*EAX]          ; указывает на конец массива B

        JZ      SHORT L2
        TEST    AL,1                      ; тестируем N на нечетность
        JZ      SHORT L1
        MOV     EAX, [ESI+4*ECX]          ; N нечетно
        NEG     EAX
        MOV     [EDI+4*ECX], EAX
        INC     ECX                       ; делаем дополнительную операцию,
                                          ; если счетчик нечетен
        JZ      SHORT L2                  ; N = 1
L1:     MOV     EAX, [ESI+4*ECX]          ; u
        MOV     EBX, [ESI+4*ECX+4]        ; v (спаривается)
        NEG     EAX                       ; u

        NEG     EBX                       ; u
        MOV     [EDI+4*ECX], EAX          ; u
        MOV     [EDI+4*ECX+4], EBX        ; v (спаривается)
        ADD     ECX, 2                    ; u
        JNZ     L1                        ; v (спаривается)
L2:
```

Теперь мы делаем две операции одновременно, что приводит к лучшему спариванию. Нам приходится тестировать N на нечетность, и если оно нечетно, то делаем дополнительную операцию вне цикла, потому что внутри него мы можем сделать только четное количество операций.

В цикле есть задержка AGI, генерируемая первой инструкцией MOV, потому что ECX повышается на 1 в предыдущем такте, поэтому для двух операций цикл занимает 6 тактов.

Реорганизация цикла для удаления задержки AGI (PPlain и PMMX)

Пример 1.8:

```
        MOV     ESI, [A]
        MOV     EAX, [N]
        MOV     EDI, [B]
        XOR     ECX, ECX
        LEA     ESI, [ESI+4*EAX]          ; указывает на конец массива A
        SUB     ECX, EAX                  ; -N

        LEA     EDI, [EDI+4*EAX]          ; указывает на конец массива B
        JZ      SHORT L3
        TEST    AL,1                      ; тестируем N на нечетность
        JZ      SHORT L2
        MOV     EAX, [ESI+4*ECX]          ; делаем дополнительную операцию,
                                          ; если счетчик нечетен
        NEG     EAX                       ; нет возможности к спариванию
        MOV     [EDI+4*ECX-4], EAX
        INC     ECX                       ; делаем счетчик четным
        JNZ     SHORT L2
        NOP                   ; добавляем NOP'ы, если JNZ L2 не предсказуем

        NOP
        JMP     SHORT L3                  ; N = 1
L1:     NEG     EAX                       ; u
        NEG     EBX                       ; u
        MOV     [EDI+4*ECX-8], EAX        ; u
        MOV     [EDI+4*ECX-4], EBX        ; v (спаривается)
L2:     MOV     EAX, [ESI+4*ECX]          ; u
        MOV     EBX, [ESI+4*ECX+4]        ; v (спаривается)
        ADD     ECX, 2                    ; u
        JNZ     L1                        ; v (спаривается)
        NEG     EAX

        NEG     EBX
        MOV     [EDI+4*ECX-8], EAX
        MOV     [EDI+4*ECX-4], EBX
L3:
```

Трюк заключается в том, что найти пару инструкций, которые не используют счетчик цикла в качестве индекса, и реорганизовать цикл так, чтобы счетчик повышал свое значение в предыдущем такте. Сейчас мы близки к 5 тактам для двух операций, что близко к самому лучшему варианту.

Если кэширование данных критично, тогда вы можете улучшить скорость, соединив массивы A и B в один массив, так чтобы каждый B[i] находился непосредственно за соответствующим A[i]. Если структурированный массив выровнен по крайней мере на 8, тогда B[i] всегда будет находится в той же линии кэша, что и A[i], поэтому будет всегда случаться промах кэша при записи в B[i]. Это, конечно, может сглаживаться приростом производительности в других частях программы, поэтому вы должны взвесить все за и против.

Разворачивание более чем на 2 (PPlain и PMMX)

Вы можете прикинуть возможность более чем двух операций за одно повторение, чтобы снизить количество действий, необходимых для организации цикла. Но так как в большинстве случаев время, требуемое для выполнения таких действий, можно снизить только на один такт, то разворачивание цикла в четыре раза, а не в два, сохранит только 1/4 такта на операцию, что вряд ли стоит усилий, которые будут на это потрачены. Только если можно сохранить еще больше, и если N очень велико, вам стоит думать о разворачивании цикла в четыре раза.

Недостатки слишком большого разворачивания цикла следующие:

• Вам будет необходимо посчитать N MODULO R, где R - это коэффициент разворачивания, и сделать N MODULO R операций до или после основного цикла, чтобы выполнить недостающее количество операций. На это уйдет дополнительный код и плохо предсказуемые операции условного перехоа. И, конечно, тело цикла станет больше.

• Кусок кода обычно выполняется в первый раз гораздо дольше, и чем больше ваш код, тем больше будут связанные с этим потери, особенно, если N невелико.
• Значительное увеличение кода делает работу с кэшем менее эффективной.

Одновременная обработка нескольких 8-ми или 16-ти битных операндов в 32-х битных регистра (PPlain и PMMX)

Если вам нужно обрабатывать массивы, состоящие из 8-ми или 16-ти битных операндов, тогда у вас появятся проблемы с развернутыми циклами, потому что вы не сможете спарить две операции доступа к памяти. Например, 'MOV AL,[ESI] / MOV BL,[ESI+1]' не будут спариваться, так как оба операнда находятся внутри одного и того же двойного слова памяти. Но есть продвинутый способ обрабатывать сразу два байта за раз в одном 32-х битном регистре.

Следующий пример добавляет 2 ко всем элементам массива байтов:

Пример 1.9:

```
        MOV     ESI, [A]         ; адрес массива байтов
        MOV     ECX, [N]         ; количество элементов в массиве байтов
        TEST    ECX, ECX         ; проверяем, равен ли N нулю
        JZ      SHORT L2
        MOV     EAX, [ESI]       ; считываем первые четыре байта
L1:     MOV     EBX, EAX         ; копируем в EBX
        AND     EAX, 7F7F7F7FH   ; получаем нижние 7 битов каждого байта в EAX

        XOR     EBX, EAX         ; получаем самый высший бит каждого из байтов
        ADD     EAX, 02020202H   ; добавляем желаемое значение ко всем четырем
                                 ; байтам
        XOR     EBX, EAX         ; снова комбинируем биты
        MOV     EAX, [ESI+4]     ; считываем следующие четыре байта
        MOV     [ESI], EBX       ; сохраняем результат
        ADD     ESI, 4           ; повышаем значение указателя на 4
        SUB     ECX, 4           ; понижаем значение счетчика цикла
        JA      L1               ; цикл
L2:
```

Этот цикл занимает 5 тактов для каждых 4 байт. Массив, разумеется, должен быть выравнен на 4. Если количество элементов в массиве не кратно четырем, тогда вы можете добавить в конец несколько байтов, чтобы сделать его делимым на четыре. Этот цикл будет читать за концом массива, поэтому вам нужно убедиться, что тот не находится в конце сегмента, дабы избежать ошибки общей защиты.

Обратите внимание, что перед прибавлением я "спрятал" наивысший бит каждого байта, чтобы не испортить их значение (если результат сложения для конкретного байта превысит 256). Я использую XOR, а не ADD, когда помещаю последний бит на место по той же самой причине.

Инструкция 'ADD ESI,4' можно избежать, если использовать счетчик цикла в качестве индекса, как это сделано в примере 1.4. Тем не менее, в результате количество инструкций в цикле будет нечетно, а значит одна инструкция будет не спарена, и цикл по-прежнему займет 5 тактов. Делая инструкцию перехода неспаренной сохранит один такт после последней операции, если инструкция была предсказана неверно, но нам придется потратить дополнительный такт в коде пролога, чтобы установить указатель в конец массива и высчитать -N, поэтому эти два метода будут одинакого быстры. Метод, представленный здесь, является самым простым и самым коротким.

Следующий пример находит длину строки, заканчивающейся нулем, путем поиска первого байта, равного нулю. Он быстрее, чем использовать REP SCASB:

Пример 1.10:

```
STRLEN  PROC    NEAR
        MOV     EAX,[ESP+4]             ; получаем указатель
        MOV     EDX,7
        ADD     EDX,EAX                 ; указатель+7 используется в конце
        PUSH    EBX
```

```
        MOV     EBX,[EAX]               ; читаем первые 4 байта
        ADD     EAX,4                   ; повышаем значение указателя

L1:     LEA     ECX,[EBX-01010101H]     ; вычитаем один из каждого байта
        XOR     EBX,-1                  ; инвертируем все байты
        AND     ECX,EBX                 ; и эти два
        MOV     EBX,[EAX]               ; читаем следующие 4 байта
        ADD     EAX,4                   ; повышаем значение указателя
        AND     ECX,80808080H           ; тестируем все биты знаков
        JZ      L1                      ; нет нулевых байтов, продолжаем
                                        ; цикл
        TEST    ECX,00008080H           ; тестируем первые два байта

        JNZ     SHORT L2
        SHR     ECX,16                  ; не в первых двух байтах
        ADD     EAX,2
L2:     SHL     CL,1                    ; используем флаг переноса, чтобы
                                        ; избежать переход
        POP     EBX
        SBB     EAX,EDX                 ; высчитываем длину
        RET
STRLEN  ENDP
```

Здесь мы снова использовали метод пересечения конца выполнения одной операции с началом выполнения другой, чтобы улучшить спаривание. Я не стал разворачивать цикл, так как количество его повторений относительно невелико. Строка, конечно, должна быть выравнена на 4. Код будет считывать несколько байт за строкой, поэтому та не должна располагаться в конце сегмента.

Количество инструкций в цикле нечетно, поэтому одна из них неспарена. Сделав неспаренной инструкцию перехода, а не какую-либо другую, мы сэкономим один такт, если инструкция перехода будет предсказана неверно.

Инструкция 'TEST ECX,00008080H' неспариваема. Вы можете использовать здесь спариваемую инструкцию 'OR CH,CL' вместо нее, но тогда вам придется поместить NOP или что-нибудь еще, чтобы избежать потерь из-за последовательных инструкций перехода. Другая проблема с 'OR CH,CL' состоит в том, что она вызовет частичную задержку регистра на PPro, PII или PIII. Поэтому я выбрал использование неспариваемой инструкции TEST.

Обработка 4-х байт за раз может быть довольно сложной. Код использует формулу, которая генерирует ненулевое значение для байт только тогда, когда байт равен нулю. Это делает возможным протестировать все четыре байта за одну операцию. Этот алгоритм включает вычитание 1 из всех байтов (в инструкции LEA). Я не стал "прятать" наивысший бит перед вычитанием, как я сделал это в предыдущем примере, поэтому операция вычитания может испортить значение следующего байта в EAX, но только если текущий равен нулю, и нам не важно, что будет со следующим байтом. Если бы мы искали нулевой байт в обратном порядке, нам бы пришлось считать двойное слово повторно после обнаружения нуля, а затем протестировать все четыре байта, чтобы найти последний ноль, или использовать BSWAP для изменения порядка байтов.

Если вы хотите найти байт с другим, ненулевым значением, тогда вы можете XOR'ить все четыре байта со значением, которое вы ищете, а затем используйте метод выше для поиска нуля.

Циклы с операциями MMX (PMMX)

Обработка нескольких операндо в одном регистре проще на MMX-процессорах, потому что у них специальные инструкции и регистры именно для этих целей.

Возвращаясь к задаче добавления двух ко всем байтам в массиве, мы можем воспользоваться возможностями, предоставляемыми MMX-инструкциями:

Пример 1.11:

```
.data
```

```
ALIGN   8
ADDENTS DQ      0202020202020202h       ; указываем байт, который нужно
                                        ; добавить 8 раз
A       DD      ?                       ; адрес массива байтов
N       DD      ?                       ; количество итераций

.code
        MOV     ESI, [A]
        MOV     ECX, [N]
        MOVQ    MM2, [ADDENTS]
        JMP     SHORT L2
        ; top of loop
L1:     MOVQ    [ESI-8], MM0    ; сохраняем результат
L2:     MOVQ    MM0, MM2        ; загружаем слагаемые

        PADDB   MM0, [ESI]      ; обрабатываем 8 байт за одну операцию
        ADD     ESI, 8
        DEC     ECX
        JNZ     L1
        MOVQ    [ESI-8], MM0    ; сохраняем последний результат
        EMMS
```

Инструкция сохранения помещена после инструкции управления циклом, чтобы избежать задержки сохранения.

Этот цикл занимает 4 такта, потому что инструкция PADDB не спаривается с 'ADD ESI, 8'. (Инструкция MMX с доступом к памяти не может спариваться с не-MMX инструкцией или с другой инструкцией MMX с доступом к памяти). Мы можем избавиться от 'ADD ESI, 8', используя в качестве индекса ECX, но это приведет к задержке AGI.

Так как затраты на управления циклом значительны, мы можем захотеть развернуть цикл:

Пример 1.12:

```
.data
ALIGN   8
ADDENTS DQ      0202020202020202h       ; значение, добавляемое к 8 байтам
times
A       DD      ?                       ; адрес массива байтов
N       DD      ?                       ; количество итераций

.code
        MOVQ    MM2, [ADDENTS]
        MOV     ESI, [A]
        MOV     ECX, [N]
        MOVQ    MM0, MM2
        MOVQ    MM1, MM2
L3:     PADDB   MM0, [ESI]

        PADDB   MM1, [ESI+8]
        MOVQ    [ESI], MM0
        MOVQ    MM0, MM2
        MOVQ    [ESI+8], MM1
        MOVQ    MM1, MM2
        ADD     ESI, 16
        DEC     ECX
        JNZ     L3
        EMMS
```

Этот развернутый цикл занимает 6 тактов на выполнение при обработке 16 байтов. Инструкции PADDB не спариваются. Две ветви перемешаны, чтобы избежать задержки сохранения.

Использование инструкций MMX приводит к большим потерям, если вы используете инструкции плавающей запятой неподалеку, поэтому могут быть ситуации, когда 32-х битные регистры будут предпочтительнее.

Циклы с инструкциями плавающей запятой (PPlain и PMMX).

Методы оптимизации циклов с инструкциями плавающей запятой примерно те же самые, что и для циклов с целочисленными инструкциями, хотя инструкции плавающей запятой пересекаются, а не спариваются.

У нас есть следующий код языка C:

```
int i, n;  double * X;  double * Y;  double DA;
for (i=0; i<n; i++)  Y[i] = Y[i] - DA * X[i];
```

Этот кусок кода (под названием DAXPY) является ключем к решению линейных уравнений.

Пример 1.13:

```
DSIZE   = 8                                 ; размер данных
        MOV     EAX, [N]                    ; количество элементов
        MOV     ESI, [X]                    ; указатель на X
        MOV     EDI, [Y]                    ; указатель на Y
        XOR     ECX, ECX
        LEA     ESI, [ESI+DSIZE*EAX]        ; указывает на конец X
        SUB     ECX, EAX                    ; -N
        LEA     EDI, [EDI+DSIZE*EAX]        ; указывает на конец Y

        JZ      SHORT L3                    ; тестируем N == 0 или нет
        FLD     DSIZE PTR [DA]
        FMUL    DSIZE PTR [ESI+DSIZE*ECX]   ; DA * X[0]
        JMP     SHORT L2                    ; переходим к циклу
L1:     FLD     DSIZE PTR [DA]
        FMUL    DSIZE PTR [ESI+DSIZE*ECX]   ; DA * X[i]
        FXCH                                ; получаем старый результат
        FSTP    DSIZE PTR [EDI+DSIZE*ECX-DSIZE] ; сохраняем Y[i]
L2:     FSUBR   DSIZE PTR [EDI+DSIZE*ECX]   ; вычитаем из Y[i]

        INC     ECX                         ; увеличиваем значение
                                            ; индекса на 1
        JNZ     L1                          ; цикл
        FSTP    DSIZE PTR [EDI+DSIZE*ECX-DSIZE] ; сохраняем последний
                                            ; результат
L3:
```

Здесь мы использовали те же методы, что и в примере 1.6: использовали счетчик цикла в качестве индекса и пробег от негативных значений к нулю. Конец выполнения одной операции пересекается с началом выполнени другой.

Смешение различных инструкций плавающей запятой работает прекрасно: задержка в 2 такта между FMUL и FSUBR заполняется FSTP предыдущего результата. Задержка в 3 такта между FSUBR и FSTP заполняется инструкциями управления циклом и первые двумя инструкциями следующего повторения. Задержку AGI удалось избежать благодая чтению в первом такте после изменения индекса только тех параметров, которые от индекса не зависят.

Это решение занимает 6 тактов на операцию, и оно лучше, чем аналогичное развернутое решение от Интела!

Разворачивание циклов с инструкциями плавающей запятой (PPlain и PMMX)

Цикл DAXPY, развернутый в 3 раза, довольно сложен:

Пример 1.14:

```
DSIZE = 8                               ; размер данных
IF DSIZE EQ 4
SHIFTCOUNT = 2
ELSE
SHIFTCOUNT = 3
ENDIF

        MOV     EAX, [N]                ; количество элементов
        MOV     ECX, 3*DSIZE            ; погрешность счетчика
        SHL     EAX, SHIFTCOUNT         ; DSIZE*N
```

```
        JZ      L4                      ; N = 0
        MOV     ESI, [X]                ; указатель на X
        SUB     ECX, EAX                ; (3-N)*DSIZE
        MOV     EDI, [Y]                ; указатель на Y
        SUB     ESI, ECX                ; конец указателя - погрешность
        SUB     EDI, ECX
        TEST    ECX, ECX
        FLD     DSIZE PTR [ESI+ECX]     ; первый X
        JNS     SHORT L2                ; меньше чем 4 операции

L1:     ; main loop
        FMUL    DSIZE PTR [DA]
        FLD     DSIZE PTR [ESI+ECX+DSIZE]
        FMUL    DSIZE PTR [DA]
        FXCH
        FSUBR   DSIZE PTR [EDI+ECX]
        FXCH
        FLD     DSIZE PTR [ESI+ECX+2*DSIZE]
        FMUL    DSIZE PTR [DA]
        FXCH
        FSUBR   DSIZE PTR [EDI+ECX+DSIZE]
        FXCH    ST(2)
        FSTP    DSIZE PTR [EDI+ECX]
        FSUBR   DSIZE PTR [EDI+ECX+2*DSIZE]
        FXCH
        FSTP    DSIZE PTR [EDI+ECX+DSIZE]

        FLD     DSIZE PTR [ESI+ECX+3*DSIZE]
        FXCH
        FSTP    DSIZE PTR [EDI+ECX+2*DSIZE]
        ADD     ECX, 3*DSIZE
        JS      L1                      ; цикл
L2:     FMUL    DSIZE PTR [DA]          ; заканчиваем оставшуюся операцию
        FSUBR   DSIZE PTR [EDI+ECX]
        SUB     ECX, 2*DSIZE            ; изменяем погрешность указателя
        JZ      SHORT L3                ; закончили
        FLD     DSIZE PTR [DA]          ; начинаем новую операцию

        FMUL    DSIZE PTR [ESI+ECX+3*DSIZE]
        FXCH
        FSTP    DSIZE PTR [EDI+ECX+2*DSIZE]
        FSUBR   DSIZE PTR [EDI+ECX+3*DSIZE]
        ADD     ECX, 1*DSIZE
        JZ      SHORT L3                ; закончили
        FLD     DSIZE PTR [DA]
        FMUL    DSIZE PTR [ESI+ECX+3*DSIZE]
        FXCH
        FSTP    DSIZE PTR [EDI+ECX+2*DSIZE]
        FSUBR   DSIZE PTR [EDI+ECX+3*DSIZE]
        ADD     ECX, 1*DSIZE
L3:     FSTP    DSIZE PTR [EDI+ECX+2*DSIZE]
```

L4:

Причина, по которой я показываю вам, как развернуть цикл в три раза, не в том, чтобы порекомендовать подобный подход, а в том, чтобы продемонстрровать, как это сложно! Будьте готовы провести значительное количество времени, отлаживая и проверяя ваш код, когда будете делать что-нибудь подобное. Есть несколько проблем, о которых нужно позаботиться: в большинстве случаев вы не сможете устранить все задержки из цикла с инструкциями плавающей запятой, развернутого менее чем на 4, если только вы не свернете его (то есть в конце цикла будут операции, выполнение которых закончится в конце следующего повторения). Последний FLD в основном цикле выше является началом первой операции в следующем повторении. Было бы соблазнительным сделать решение, которое бы считывало после конца массива, а затем бы избавлялось от лишнего значения (как в примере 1.9 и 1.10), но это не рекомендуется с инструкциями плавающей запятой, потому что чтение дополнительного значения может сгенерировать исключение ненормального значение операнда, если после массива не находится

число с плавающей запятой. Чтобы избежать этого, нам приходиться совершать по крайней мере на одну операцию больше после основного цикла.

Количество операций, которое необходимо выполнить снаружи развернутого цикла, как правило будет равно N MODULE R, где N - количество операций, а R - коэффициент развернутости цикла. Но в случае со свернутым циклом, нам нужно сделать на одну операцию больше, то есть (N-1) MODULO R + 1 в силу вышеуказанных причин.

Обычно мы делаем дополнительные операции до основного цикла, но здесь мы должны сделать их после в силу двух причин: одна причина - это позаботиться об оставшемся операнде (из-за свертывания). Другая причина состоит в том, что вычисление количества дополнительных операций требует деления, если R не является степенью 2-х, а деление занимает много времени. Дополнительные операции после цикла решают эту проблему.

Другая задача - это высчитать, как влиять на счетчик цикла так, чтобы он изменил знак в правильный момент, и привести в порядок базовые указатели. Наконец, вам следует убедиться, что оставшийся от свертывания операнд был обработан верно для всех значений N.

Эпилоговый код, делающий 1-3 операции, можно организовать как отдельный цикл, но это будет стоить неправильного предсказания, поэтому решение выше быстрее.

Теперь, когда я напугал вас трудностями разворачивания на 3, я покажу, насколько проще разворачивать на 4:

Пример 1.15:

```
DSIZE   = 8                             ; размер данных
        MOV     EAX, [N]                ; количество элементов
        MOV     ESI, [X]                ; указатель на X
        MOV     EDI, [Y]                ; указатель на Y
        XOR     ECX, ECX
        LEA     ESI, [ESI+DSIZE*EAX]    ; указывает на конец X

        SUB     ECX, EAX                ; -N
        LEA     EDI, [EDI+DSIZE*EAX]    ; указывает на конец Y
        TEST    AL,1                    ; тестируем N на нечетность
        JZ      SHORT L1
        FLD     DSIZE PTR [DA]          ; делаем нечетную операцию
        FMUL    DSIZE PTR [ESI+DSIZE*ECX]
        FSUBR   DSIZE PTR [EDI+DSIZE*ECX]
        INC     ECX                     ; увеличиваем значение счетчика
        FSTP    DSIZE PTR [EDI+DSIZE*ECX-DSIZE]
L1:     TEST    AL,2            ; проверяем, можно ли сделать еще 2 операции

        JZ      L2
        FLD     DSIZE PTR [DA]        ; N MOD 4 = 2 или 3. Делаем еще две
        FMUL    DSIZE PTR [ESI+DSIZE*ECX]
        FLD     DSIZE PTR [DA]
        FMUL    DSIZE PTR [ESI+DSIZE*ECX+DSIZE]
        FXCH
        FSUBR   DSIZE PTR [EDI+DSIZE*ECX]
        FXCH
        FSUBR   DSIZE PTR [EDI+DSIZE*ECX+DSIZE]
        FXCH
        FSTP    DSIZE PTR [EDI+DSIZE*ECX]
        FSTP    DSIZE PTR [EDI+DSIZE*ECX+DSIZE]
        ADD     ECX, 2                ; счетчик не делится 4

L2:     TEST    ECX, ECX
        JZ      L4                      ; больше операций нет
L3:     ; main loop:
        FLD     DSIZE PTR [DA]
        FLD     DSIZE PTR [ESI+DSIZE*ECX]
        FMUL    ST,ST(1)
        FLD     DSIZE PTR [ESI+DSIZE*ECX+DSIZE]
        FMUL    ST,ST(2)
        FLD     DSIZE PTR [ESI+DSIZE*ECX+2*DSIZE]
```

```
        FMUL    ST,ST(3)
        FXCH    ST(2)
        FSUBR   DSIZE PTR [EDI+DSIZE*ECX]
        FXCH    ST(3)
        FMUL    DSIZE PTR [ESI+DSIZE*ECX+3*DSIZE]

        FXCH
        FSUBR   DSIZE PTR [EDI+DSIZE*ECX+DSIZE]
        FXCH    ST(2)
        FSUBR   DSIZE PTR [EDI+DSIZE*ECX+2*DSIZE]
        FXCH
        FSUBR   DSIZE PTR [EDI+DSIZE*ECX+3*DSIZE]
        FXCH    ST(3)
        FSTP    DSIZE PTR [EDI+DSIZE*ECX]
        FSTP    DSIZE PTR [EDI+DSIZE*ECX+2*DSIZE]
        FSTP    DSIZE PTR [EDI+DSIZE*ECX+DSIZE]
        FSTP    DSIZE PTR [EDI+DSIZE*ECX+3*DSIZE]
        ADD     ECX, 4                  ; увеличиваем значение индекса на 4

        JNZ     L3                      ; цикл
L4:
```

Обычно довольно легко добиться отсутствия задержек в цикле, развернутом на 4, и нет нужды в свертывании последней операции. Количество дополнительных операций, которые нужно сделать за пределами основного цикла равно N MODULO 4, что можно легко посчитать без деления, просто протестировав два нижних бита в N. Дополнительные операции делаются до основного цикла, а не после, чтобы сделать обработку счетчика цикла проще.

Недостаток разворачивания циклов в том, что дополнительные операции, выполняемые за пределами цикла медленнее из-за несовершенного пересечения и возможных неправильных предсказаний переходов, а потери при первой загрузке кода выше из-за возросшего размера кода.

В качестве основной рекомендации, я бы посоветовал, что если N велико или если сворачивание цикла без развертывания не может удалить некоторых задержек, тогда вам следует развернуть критические целочисленные циклы в два раза, а циклы с инструкциями плавающей запятой в 4 раза.

25.2 Циклы в PPro, PII и PIII

В предыдущей главе (25.1) я объяснил, как использовать свертывание и разворачивания циклов, чтобы улучшить спаривание в PPlain и PMMX. На PPro, PII и PIII нет никакой причины делать это благодаря механизму выполнения инструкций не по порядку. Но здесь есть другие проблемы, о которых надо позаботиться, связанные с границами БДИ и задержка чтения регистров.

Я выбрал те же примеры, что и в главе 25.1 для предыдущих микропроцессоров: процедура, которая считывает целые числа из массива, изменяет их знак, и сохраняет результаты в другой массив.

На C эта процедура выглядела бы так:

```
void ChangeSign (int * A, int * B, int N) {
  int i;
  for (i=0; i<N; i++) B[i] = -A[i];}
```

Переводя ее на ассемблер, нам следовало бы написать так:

Пример 2.1:

```
_ChangeSign PROC NEAR
        PUSH    ESI
        PUSH    EDI
A       EQU     DWORD PTR [ESP+12]
B       EQU     DWORD PTR [ESP+16]
N       EQU     DWORD PTR [ESP+20]

        MOV     ECX, [N]
```

```
        JECXZ   L2

        MOV     ESI, [A]
        MOV     EDI, [B]
        CLD
L1:     LODSD
        NEG     EAX
        STOSD
        LOOP    L1
L2:     POP     EDI
        POP     ESI
        RET
_ChangeSign     ENDP
```

Это выглядит как довольно красивое решение, но не самое оптимальное, потому что он использует инструкции LOOP, LODSD и STOSD, которые генерируют много мопов. Одна итерация цикла занимает 6-7 тактов, если все данные находятся в кэше первого уровня. Если мы будем избегать данных инструкций, то получим следующее:

Пример 2.2:

```
        MOV     ECX, [N]
        JECXZ   L2
        MOV     ESI, [A]
        MOV     EDI, [B]
ALIGN   16
L1:     MOV     EAX, [ESI]      ; len=2, p2rESIwEAX
        ADD     ESI, 4          ; len=3, p01rwESIwF
        NEG     EAX             ; len=2, p01rwEAXwF
        MOV     [EDI], EAX      ; len=2, p4rEAX, p3rEDI
        ADD     EDI, 4          ; len=3, p01rwEDIwF
        DEC     ECX             ; len=1, p01rwECXwF
        JNZ     L1              ; len=2, p1rF

L2:
```

Комментарии интерпретируются следующим образом: инструкция 'MOV EAX,[ESI]' два байта длиной, она генерирует один моп для порта 2, который читает ESI и пишет в (переименовывает) EAX. Эта информация требуется для анализирования возможных узких мест.

Давайте сначала проанализируем раскодировку инструкций (глава 14): одна из инструкций генерирует два мопа ('MOV [EDI],EAX'). Эта инструкция должна попасть в декодер D0. Есть три раскодировываемые группы в цикле, поэтому его можно раскодировать за 3 такта.

Теперь давайте посмотрим на доставку инструкций (глава 15): если границы между БДИ не дадут первым трем инструкциям раскодироваться вместе, тогда будет три раскодировываемые группы в последнем БДИ, поэтому в следующем повторении БДИ начнется с первой инструкции, там где нам нужно, и мы получим задержку только в первом повторении. Худшим вариантом будет 16-ти байтная граница и граница БДИ в одно из следующих трех инструкций. Согласно таблице, приведенной в соотвествующей главе, это сгенерирует задержку в один такт, и приведет к тому, что в следующем повторении первый БДИ будет выровнен на 16, и проблема будет повторяться каждую итерацию. В результате время доставки будет равно 4 тактам на итерацию (а не 3). Есть два пути предотвратить подобный вариант развития событий: первый метод заключается в том, чтобы контролировать расположение 16-ти байтных границ. Другой метод проще. Так как во всем цикле только 15 байт кода, вы можете избежать 16-ти байтных границ, выровняв цикл на 16, как показано выше. Тогда весь цикл будет умещаться в один БДИ, поэтому никакого дальнейшего анализа доставки инструкций не потребуется.

Третья проблема - это задержки чтения регистров (глава 16). В этом цикле не читается ни один регистр, если по крайней мере несколько тактов назад в него не была произведена запись.

Четвертый анализ - это выполнение инструкций (глава 17). Подсчитывая мопы, предназначающиеся для разных портом, мы получаем:

порт 0 или 1: 4 мопа порт 1 : 1 моп порт 2: 1 моп порт 3: 1 моп порт 4: 1 моп

Предположив, что мопы, которые могут пойти в порт 0 или 1, распределяются оптимальным образом, время выполнения будет равно 2.5 такта на итерацию.

Последний анализ, который необходимо провести, это вывод из обращения (глава 18). Так как количество мопов в цикле не кратно 3, слоты вывода из обращения не будут использоваться оптимальным образом, когда переход будет выводиться из обращения через первый слот. Время, необходимое для вывода из обращения равно количеству мопов, деленное на 3 и округленное в сторону ближайшего целого числа. Это дает 3 такта для вывода из обращения.

В заключение, цикл, представленный выше, может выполняться за 3 такта на итерацию, если код цикла выровнен на 16. Я предполагаю, что условный переход предсказывается каждый раз, кроме выхода из цикла (глава 22.2).

Использование одного и того же регистра для счетчика и индекса, последнее значение счетчика равно нулю (PPro, PII и PIII)

Пример 2.3:

```
        MOV     ECX, [N]
        MOV     ESI, [A]
        MOV     EDI, [B]
        LEA     ESI, [ESI+4*ECX]          ; указывает на конец массива A
        LEA     EDI, [EDI+4*ECX]          ; указывает на конец массива B
        NEG     ECX                       ; -N
        JZ      SHORT L2
ALIGN   16
L1:     MOV     EAX, [ESI+4*ECX]          ; len=3, p2rESIrECXwEAX

        NEG     EAX                       ; len=2, p01rwEAXwF
        MOV     [EDI+4*ECX], EAX          ; len=3, p4rEAX, p3rEDIrECX
        INC     ECX                       ; len=1, p01rwECXwF
        JNZ     L1                        ; len=2, p1rF
L2:
```

Количество мопов было снижено до 6. Базовые указатели указывают на конец массивов, поэтому индекс можно увеличивать от негативных значений до нуля.

Раскодировка: в этом цикле две раскодировываемые группы, поэтому раскодировка пройдет в два такта.

Доставка инструкций:

Цикл всегда занимает по крайней мере на один такт больше, чем количество 16-ти байтных блоков. Так как в этом цикле только 11 байтов кода, можно уместить их все в один БДИ. Выровняв цикл на 16 мы будем уверены, что будет только один 16-ти байтный блок, поэтому возможно осущесвить доставку за 2 такта.

Задержки чтения регистров:

Регистры ESI и EDI прочитаны, но не модифицируются внутри цикла. Поэтому эти считывания будут осуществляться из постоянных регистров, но не в одном триплете. Регистры EAX, ECX и флаги модифицируются внутри цикла и считываются до того, как они записываются обратно, поэтому чтения постоянных регистров не будет. То есть можно сделать заключение, что задержек чтения регистров нет.

Выполнение: порт 0 or 1: 2 моп порт 1: 1 моп порт 2: 1 моп порт 3: 1 моп порт 4: 1 моп Время выполнения: 1.5 такта.

Вывод из обращения: 6 мопов = 2 такта.

Заключение: этот цикл занимает только два такта на итерацию.

Если вы используете абсолютные адреса вместо ESI и EDI, тогда цикл будет занимать 3 такта, потому что он не сможет уместиться в один 16-ти байтный блок.

Разворачивание цикла (PPro, PII и PIII)

Осуществление более чем одной операции за проход, а соответственно, и меньшего количества проходов называется разворачиванием циклов. В предыдущих процессорах вам следовало развернуть циклы, чтобы разворачивать циклы, чтобы добиться лучшего спаривания (глава 25.1). В PPro, PII и PIII это не требуется, потому что об этом заботится механизм выполнения инструкций не по порядку. Нет никакой надобности использовать два разных регистра, потому что этим занимается переименование регистров. Целью разворачивания является снижение расходов на управление циклом при каждом повторении.

Следующий пример по смыслу тот же, что и пример в 2.2, но развернутый в два разм, что означает выполнение двух операций за раз, и меньшее в два раза количество проходов.

Пример 2.4:

```
        MOV     ECX, [N]
        MOV     ESI, [A]
        MOV     EDI, [B]
        SHR     ECX, 1          ; N/2
        JNC     SHORT L1        ; тестируем N на нечетность
        MOV     EAX, [ESI]      ; делаем нечетный раз
        ADD     ESI, 4
        NEG     EAX
        MOV     [EDI], EAX

        ADD     EDI, 4
L1:     JECXZ   L3

ALIGN   16
L2:     MOV     EAX, [ESI]      ; len=2, p2rESIwEAX
        NEG     EAX             ; len=2, p01rwEAXwF
        MOV     [EDI], EAX      ; len=2, p4rEAX, p3rEDI
        MOV     EAX, [ESI+4]    ; len=3, p2rESIwEAX
        NEG     EAX             ; len=2, p01rwEAXwF
        MOV     [EDI+4], EAX    ; len=3, p4rEAX, p3rEDI
        ADD     ESI, 8          ; len=3, p01rwESIwF
        ADD     EDI, 8          ; len=3, p01rwEDIwF

        DEC     ECX             ; len=1, p01rwECXwF
        JNZ     L2              ; len=2, p1rF
L3:
```

В примере 2.2 инструкции управления циклом (то есть изменение значений указателей и счетчика, а также переход назад) занимал 4 мопа, а 'реальная работа' занимала 4 мопа. При разворачивании цикла в два раза

Анализируя доставку инструкций в этом цикле мы увидим, что новый БДИ начинается с инструкции 'ADD ESI, 8', следовательно она пойдет в декодер D0. Поэтому цикл раскодировывается за 5 тактов, а не за 4, как нам хотелось. Мы можем решить эту проблему, заменив предыдущую инструкцию на более длинную версию.

Изменить 'MOV [EDI+4],EAX' на:

```
MOV [EDI+9999],EAX     ; создаем инструкцию с большим смещением
ORG $-4
DD 4                   ; изменяем смещение на 4
```

Это заставит новый БДИ начаться с длинной инструкции 'MOV [EDI+4]', поэтому время раскодировки сейчас близко к 4 тактам. Оставшаяся свободная часть конвеера может обрабатывать 3 мопа за такт, поэтому ожидаемое время выполнения равно 4 тактам или 2 тактам на операцию.

Тестирование этого решения показало, что в реальности оно занимает немного больше. Мои измерения показывают примерно 4.5 такта за итерацию. Вероятно это связано с неоптимальной перегруппировкой мопов. Возможно, ROB не смог найти оптимального порядка выполнения. Проблемы подобного рода непредсказуемы, и только тестирование может выявить их. Мы можем помочь ROB'у сделать часть перегруппировки сами:

Пример 2.5:

```
ALIGN   16
L2:     MOV     EAX, [ESI]       ; len=2, p2rESIwEAX
        MOV     EBX, [ESI+4]     ; len=3, p2rESIwEBX
        NEG     EAX              ; len=2, p01rwEAXwF
        MOV     [EDI], EAX       ; len=2, p4rEAX, p3rEDI
        ADD     ESI, 8           ; len=3, p01rwESIwF
        NEG     EBX              ; len=2, p01rwEBXwF
        MOV     [EDI+4], EBX     ; len=3, p4rEBX, p3rEDI
        ADD     EDI, 8           ; len=3, p01rwEDIwF
        DEC     ECX              ; len=1, p01rwECXwF

        JNZ     L2               ; len=2, p1rF
L3:
```

Цикл теперь выполняется в 4 тактах на итерацию. Также была решена проблема с БДИ. Это стоило нам дополнительного регистра, потому что мы не могли использовать преимущества переименования регистров.

Разворачивание более чем в 2 раза

Развертка циклов рекомендуется, когда инструкции по управлению циклами занимают значительную часть общего времени выполнения. В примере 2.3 они занимают только 2 мопа, поэтому выгода будет невелика, но тем не менее я покажу, как его развернуть, просто ради упражнения.

'Настоящая работа' равна 4 мопам, на управление циклом уходит 2 мопа. Развернув его, мы получим 2*4+3 = 10 мопов. Время вывода из обращения будет равно 10/3, огругляем в сторону ближайшего целого, получается 4 такта. Эти вычисления показывают, что разворачиванием мы ничего не добьемся:

Пример 2.6:

```
        MOV     ECX, [N]
        SHL     ECX, 2                  ; количество, которое нужно
                                        ; обработать
        MOV     ESI, [A]
        MOV     EDI, [B]
        ADD     ESI, ECX                ; указываем на конец массива A

        ADD     EDI, ECX                ; указываем на конец массива B
        NEG     ECX                     ; -4*N
        TEST    ECX, 4                  ; тестируем N на нечетность
        JZ      SHORT L1
        MOV     EAX, [ESI+ECX]          ; N нечетно. Делаем нечетный раз
        NEG     EAX
        MOV     [EDI+ECX], EAX
        ADD     ECX, 4
L1:     TEST    ECX, 8                  ; Тестируем N/2 на нечетность
        JZ      SHORT L2
        MOV     EAX, [ESI+ECX]          ; N/2 нечетно. Делаем два
                                        ; дополнительных раза

        NEG     EAX
        MOV     [EDI+ECX], EAX
        MOV     EAX, [ESI+ECX+4]
        NEG     EAX
        MOV     [EDI+ECX+4], EAX
        ADD     ECX, 8
L2:     JECXZ   SHORT L4
```

```
ALIGN   16
L3:     MOV     EAX, [ESI+ECX]          ; len=3, p2rESIrECXwEAX
        NEG     EAX                     ; len=2, p01rwEAXwF
        MOV     [EDI+ECX], EAX          ; len=3, p4rEAX, p3rEDIrECX
        MOV     EAX, [ESI+ECX+4]        ; len=4, p2rESIrECXwEAX
        NEG     EAX                     ; len=2, p01rwEAXwF

        MOV     [EDI+ECX+4], EAX        ; len=4, p4rEAX, p3rEDIrECX
        MOV     EAX, [ESI+ECX+8]        ; len=4, p2rESIrECXwEAX
        MOV     EBX, [ESI+ECX+12]       ; len=4, p2rESIrECXwEAX
        NEG     EAX                     ; len=2, p01rwEAXwF
        MOV     [EDI+ECX+8], EAX        ; len=4, p4rEAX, p3rEDIrECX
        NEG     EBX                     ; len=2, p01rwEAXwF
        MOV     [EDI+ECX+12], EBX       ; len=4, p4rEAX, p3rEDIrECX
        ADD     ECX, 16                 ; len=3, p01rwECXwF

        JS      L3                      ; len=2, p1rF
L4:
```

БДИ распределяются так, как нам нужно. Время раскодировки равно 6 тактам.

Задержки чтения регистров является здесь проблемой, так как ECX выводится из обращения в конце цикла, а нам нужно читать ESI, EDI и ECX. Инструкции были перегруппированы так, чтобы избежать чтения ESI рядом с концом цикла во избежания задержки чтения регистра. Другими словами причина перегруппировки инструкций и использования дополнительного регистра не та, что в предыдущем примере.

Здесь 12 мопов и цикл выполняется за 6 тактов на итерацию или 1.5 тактов на операцию.

Вы можете прикинуть возможность более чем двух операций за одно повторение, чтобы снизить количество действий, необходимых для организации цикла. Но так как в большинстве случаев время, требуемое для выполнения таких действий, можно снизить только на один такт, то разворачивание цикла в четыре раза, а не в два, сохранит только 1/4 такта на операцию, что вряд ли стоит усилий, которые будут на это потрачены. Только если можно сохранить еще больше, и если N очень велико, вам стоит думать о разворачивании цикла в четыре раза.

Недостатки слишком большого разворачивания цикла следующие:

• Вам будет необходимо посчитать N MODULO R, где R - это коэффициент разворачивания, и сделать N MODULO R операций до или после основного цикла, чтобы выполнить недостающее количество операций. На это уйдет дополнительный код и плохо предсказуемые операции условного перехоа. И, конечно, тело цикла станет больше.
• Кусок кода обычно выполняется в первый раз гораздо дольше, и чем больше ваш код, тем больше будут связанные с этим потери, особенно, если N невелико.
• Значительное увеличение кода делает работу с кэшем менее эффективной.

Использование коэффициента разворачиванивания, не являющегося степенью 2 делает вычисление N MODULO R довольно трудным, и как правило не рекомендуется, если только N не кратно R. Пример 1.14 показывает, как разворачивать в 3 раза.

Обработка нескольких 8-ми или 16-ти битных операндов одновременно в 32-х битных регистрах (PPro, PII или PIII).

Иногда возможно обрабатывать четыре байта за раз в том же 32-х битном регистре. Следующий пример добавляет 2 ко всем элементам массива байтов.

Пример 2.7:

```
        MOV     ESI, [A]        ; адрес массива байтов
        MOV     ECX, [N]        ; количество элементов в массиве байтов
        JECXZ   L2
ALIGN   16
        DB   7  DUP (90H)       ; 7 NOP'ов выравнивания
 L1:    MOV     EAX, [ESI]      ; читаем четыре байта
```

```
        MOV     EBX, EAX         ; копируем в EBX
        AND     EAX, 7F7F7F7FH   ; получаем 7 нижних бит каждого байта
        XOR     EBX, EAX         ; получаем наивысший бит каждого байта

        ADD     EAX, 02020202H   ; добавляем желаемое значение ко всем байтам
        XOR     EBX, EAX         ; снова комбинируем биты
        MOV     [ESI], EBX       ; сохраняем результат
        ADD     ESI, 4           ; увеличиваем значение указателя
        SUB     ECX, 4           ; понижаем значение счетчика
        JA      L1               ; цикл
L2:
```

Обратите внимание, что перед прибавлением я "спрятал" наивысший бит каждого байта, чтобы не испортить их значение (если результат сложения для конкретного байта превысит 256). Я использую XOR, а не ADD, когда помещаю последний бит на место по той же самой причине.

Этот цикл в идеальном случае занимает 4 такта на итерацию, но в реальном случае он может занять немного больше из-за цепочки зависимости и трудностей с перегруппировкой. На PII и PIII вы можете это делать еще более эффективно, используя регистры MMX.

Следующий пример находит длину строки, заканчивающейся нулем. Он гораздо быстрее, чем REPNE SCASB:

Пример 2.8:

```
_strlen PROC    NEAR
        PUSH    EBX
        MOV     EAX,[ESP+8]            ; получаем указатель на строку

        LEA     EDX,[EAX+3]            ; указатель+3 используется в конце
L1:     MOV     EBX,[EAX]              ; читаем первые 4 байта
        ADD     EAX,4                  ; повышаем значение указателя
        LEA     ECX,[EBX-01010101H]    ; вычитаем 1 из каждого байта
        NOT     EBX                    ; инвертируем все байты
        AND     ECX,EBX                ; и эти два
        AND     ECX,80808080H          ; тестируем все биты
        JZ      L1                     ; нет нулевых байтов, продолжаем цикл

        MOV     EBX,ECX
        SHR     EBX,16
        TEST    ECX,00008080H          ; тестируем первые два байта
        CMOVZ   ECX,EBX                ; сдвигаем вправо, если не впервых двух
                                       ; байтах
        LEA     EBX,[EAX+2]
        CMOVZ   EAX,EBX
        SHL     CL,1                   ; используем флаг переноса, чтобы избежать
                                       ; ветвления
        SBB     EAX,EDX                ; высчитываем длину
        POP     EBX
        RET
_strlen ENDP
```

Этот цикл занимает 3 такта на каждую итерацию, тестируя 4 байта. Строка, разумеется, должна быть выравнена на 4 байта. Код может считать несколько байт из памяти, находящейся за концом строки, поэтому строка не должна находиться у конца сегмента.

Обработка 4-х байт за раз может быть довольно сложной. Код использует формулу, которая генерирует ненулевое значение для байт только тогда, когда байт равен нулю. Это делает возможным протестировать все четыре байта за одну операцию. Этот алгоритм включает вычитание 1 из всех байтов (в инструкции LEA). Я не стал "прятать" наивысший бит перед вычитанием, как я сделал это в предыдущем примере, поэтому операция вычитания может испортить значение следующего байта в EAX, но только если текущий равен нулю, и нам не важно, что будет со следующим байтом. Если бы мы искали нулевой байт в обратном порядке, нам бы пришлось считать двойное слово повторно после обнаружения нуля, а затем протестировать все четыре байта, чтобы найти последний ноль, или использовать BSWAP для изменения порядка

байтов.

Если вы хотите найти байт с другим, ненулевым значением, тогда вы можете XOR'ить все четыре байта со значением, которое вы ищете, а затем используйте метод выше для поиска нуля.

Циклы с инструкциями MMX (PII и PIII)

С помощью инструкций MMX мы можем сравнивать 8 байтов за одну операцию:

Пример 2.9:

```
_strlen PROC    NEAR
        PUSH    EBX
        MOV     EAX,[ESP+8]
        LEA     EDX,[EAX+7]
        PXOR    MM0,MM0
L1:     MOVQ    MM1,[EAX]       ; len=3 p2rEAXwMM1
        ADD     EAX,8           ; len=3 p01rEAX
        PCMPEQB MM1,MM0         ; len=3 p01rMM0rMM1
        MOVD    EBX,MM1         ; len=3 p01rMM1wEBX

        PSRLQ   MM1,32          ; len=4 p1rMM1
        MOVD    ECX,MM1         ; len=3 p01rMM1wECX
        OR      ECX,EBX         ; len=2 p01rECXrEBXwF
        JZ      L1              ; len=2 p1rF
        MOVD    ECX,MM1
        TEST    EBX,EBX
        CMOVZ   EBX,ECX
        LEA     ECX,[EAX+4]
        CMOVZ   EAX,ECX
        MOV     ECX,EBX
        SHR     ECX,16
        TEST    BX,BX
        CMOVZ   EBX,ECX
        LEA     ECX,[EAX+2]
        CMOVZ   EAX,ECX

        SHR     BL,1
        SBB     EAX,EDX
        EMMS
        POP     EBX
        RET
_strlen ENDP
```

В этом цикле 7 мопов для порта 0 и 1, что дает среднее время выполнения 3.5 такта на итерацию. Тесты показали 3.8 тактов, что говорит о том, что ROB справляется с ситуацией достаточно хорошо, несмотря на цепочку зависимости, которая равна 6 мопам. Тестирование 8 байтов меньше, чем за 4 такта гораздо быстрее, чем REPNE SCASB.

Циклы с плавающими инструкциями (PPro, PII и PIII)

Методы оптимизирования циклов с плавающей запятой примерно те же, что и для целочисленных циклов, но вы должны еще больше остерегаться цепочек зависимости из-за долгого времени выполнения инструкций.

Предположите код на языке C:

```
int i, n;  double * X;  double * Y;  double DA;
for (i=0; i<n; i++)  Y[i] = Y[i] - DA * X[i];
```

Этот кусок кода (под названием DAXPY) является ключем к решению линейных уравнений.

Пример 2.10:

```
DSIZE   = 8                     ; размер данных (4 or 8)
        MOV     ECX, [N]        ; количество элементов
        MOV     ESI, [X]        ; указатель на X
        MOV     EDI, [Y]        ; указатель на Y
        JECXZ   L2              ; проверяем, равняется ли N нулю
```

```
        FLD     DSIZE PTR [DA]   ; загружаем DA вне цикла
ALIGN   16
        DB    2 DUP (90H)        ; 2 NOP'a для выравнивания
L1:     FLD     DSIZE PTR [ESI]  ; len=3 p2rESIwST0
        ADD     ESI,DSIZE        ; len=3 p01rESI

        FMUL    ST,ST(1)         ; len=2 p0rST0rST1
        FSUBR   DSIZE PTR [EDI]  ; len=3 p2rEDI, p0rST0
        FSTP    DSIZE PTR [EDI]  ; len=3 p4rST0, p3rEDI
        ADD     EDI,DSIZE        ; len=3 p01rEDI
        DEC     ECX              ; len=1 p01rECXwF
        JNZ     L1               ; len=2 p1rF
        FSTP    ST               ; сбрасываем DA
L2:
```

Цепочка зависимости длиной в 10 тактов, но цикл занимает только 4 такта на итерацию, потому что он может начать новую операцию еще до того, как выполнена предыдущая. Цель выравнивания - предотвратить 16-ти байтную границу в последнем БДИ.

Пример 2.11:

```
DSIZE   = 8                              ; размер данных (4 или 8)
        MOV     ECX, [N]                 ; количество элементов
        MOV     ESI, [X]                 ; указатель на X
        MOV     EDI, [Y]                 ; указатель на Y
        LEA     ESI, [ESI+DSIZE*ECX]     ; указатель на конец массива
        LEA     EDI, [EDI+DSIZE*ECX]     ; указатель на конец массива
        NEG     ECX                      ; -N
        JZ      SHORT L2                 ; проверяем, равняется ли N нулю

        FLD     DSIZE PTR [DA]           ; загружаем DA вне цикла
ALIGN   16
L1:     FLD     DSIZE PTR [ESI+DSIZE*ECX]  ; len=3 p2rESIrECXwST0
        FMUL    ST,ST(1)                 ; len=2 p0rST0rST1
        FSUBR   DSIZE PTR [EDI+DSIZE*ECX]  ; len=3 p2rEDIrECX, p0rST0
        FSTP    DSIZE PTR [EDI+DSIZE*ECX]  ; len=3 p4rST0, p3rEDIrECX
        INC     ECX                      ; len=1 p01rECXwF
        JNZ     L1                       ; len=2 p1rF
        FSTP    ST                       ; сбрасываем DA

L2:
```

Здесь мы используем тот же самый трюк, что и в примере 2.3. В идеальном случае, этот цикл будет занимать 3 такта, но измерения говорят примерно о 3.5 в виду длинной цепочки зависимости. Разворачивание цикла сэкономило немного.

Циклы с инструкциями XMM (PIII)

Инструкции XMM на PIII позволят вам оперировать четырьмя числами с плавающей запятой одинарной точности одновременно. Операнды должны быть выравнены на 16.

Алгоритм DAXPY не очень подходит для инструкций XMM, потому что точность невелика, может не быть возможности выравнять операнды на 16, поэтому вам потребуется дополнительный код, если количество операций не кратно четырем. Тем не менее, я покажу вам код в качестве примера цикла с инструкциями XMM:

Пример 2.12:

```
        MOV     ECX, [N]                 ; количество элементов
        MOV     ESI, [X]                 ; указатель на X
        MOV     EDI, [Y]                 ; указатель на Y
        SHL     ECX, 2
        ADD     ESI, ECX                 ; указывает на конец X
        ADD     EDI, ECX                 ; указывает на конец Y
        NEG     ECX                      ; -4*N
        MOV     EAX, [DA]                ; загружаем DA вне цикла
        XOR     EAX, 80000000H           ; меняем знак DA
        PUSH    EAX
```

```
        MOVSS   XMM1, [ESP]             ; -DA
        ADD     ESP, 4
        SHUFPS  XMM1, XMM1, 0           ; копируем -DA во все четыре
                                        ; позиции
        CMP     ECX, -16
        JG      L2
L1:     MOVAPS  XMM0, [ESI+ECX]         ; len=4 2*p2rESIrECXwXMM0
        ADD     ECX, 16                 ; len=3 p01rwECXwF
        MULPS   XMM0, XMM1              ; len=3 2*p0rXMM0rXMM1
        CMP     ECX, -16                ; len=3 p01rECXwF

        ADDPS   XMM0, [EDI+ECX-16]      ; len=5 2*p2rEDIrECX, 2*p1rXMM0
        MOVAPS  [EDI+ECX-16], XMM0      ; len=5 2*p4rXMM0, 2*p3rEDIrECX
        JNG     L1                      ; len=2 p1rF
L2:     JECXZ   L4                      ; check if finished
        MOVAPS  XMM0, [ESI+ECX]         ; 1-3 операции пропущены, делаем
                                        ; еще четыре
        MULPS   XMM0, XMM1
        ADDPS   XMM0, [EDI+ECX]
        CMP     ECX, -8
        JG      L3
        MOVLPS  [EDI+ECX], XMM0         ; сохраняем еще два результата

        ADD     ECX, 8
        MOVHLPS XMM0, XMM0
L3:     JECXZ   L4
        MOVSS   [EDI+ECX], XMM0         ; сохраняем еще один результат
L4:
```

Цикл L1 занимает 5-6 тактов на 4 операции. Инструкции с ECX были помещены до и после 'MULPS XMM0, XMM1', чтобы избежать задержки чтения регистра, которую бы сгенерировало чтение двух частей регистра XMM1 вместе с ESI и EDI в RAT. Дополнительный код после L2 заботится о ситуации, когда N не делится на 4. Обратите внимание, что этот код может прочитать несколько байтов после конца A и B. Это может задержать последнюю операцию, если в этих байтах не находились нормальные числа с плавающей запятой. Если возможно, поместите в массив какие-нибудь дополнительные данные, чтобы сделать количество операций кратным 4 и избавиться от лишнего кода после L2.

Оптимизация для процессоров семейства Pentium: 26. Проблемные инструкции

Автор: Агнер Фог, пер. Aquila

Дата: 2002-01-01

26.1 XCHG (все процессоры)

Инструкция `XCHG регистр, [память]` опасна. По умолчанию эта инструкция имеет неявный префикс `LOCK`, что не дает ей загружаться в кэш. Поэтому выполнение данной инструкции отнимает очень много времени, и ее следует избегать.

26.2 Вращение через флаг переноса (все процессоры)

`RCR` и `RCL`, сдвигающие более, чем один бит, медленны, и их следует избегать.

26.3 Строковые инструкции (все процессоры)

Строковые инструкции без префикса повторения слишком медленны, и их следует заменить более простыми инструкциями. То же самое относится к `LOOP` на всех процессорах и к `JECXZ` на PPlain и PMMX.

`REP MOVSD` и `REP STOSD` довольно быстры, если число повторений не слишком мало. Всегда используйте версию DWORD, где это возможно, и убедитесь, что источник и приемник выравнены на 8.

Некоторые другие методы перемещения данных быстрее в определенных условиях. Подробнее смотрите главу 27.8.

Обратите внимание, что пока инструкция `REP MOVS` записывает слово в приемник, она считывает следующее слово из источника в том же такте. У вас может конфликт банков кэша, если биты 2-4 у этих двух адресов одни и те же. Другими словами, у вас будут неизбежные потери в один такт на итерацию, если `ESI+(размер слова)-EDI` кратно 32. Самый простой путь избежать конфликтов банков кэша - это использовать версию DWORD и выравнивать источник и приемник на 8. Никогда не используйте `MOVSB` или `MOVSW` в оптимизированном коде, даже в 16-ти битном.

`REP MOVS` и `REP STOS` могут выполняться очень быстро, если перемещать целую линию кэша за раз на PPro, PII и PIII:

• источник и приемник должны быть выравнены на 8
• должно быть задано направление вперед (очищен флаг направления)
• счетчик (`ECX`) должен иметь значение равное или большее 64
• разница между `EDI` и `ESI` должна быть численно больше или равна 32

При этих условиях количество мопов будет примерно равно `215+2*ECX` для

REP MOVSD и 185+1.5*ECX для REP STOSD, что дает примерную скорость в 5 байтов в такт для обоих инструкций, что в три раза больше, если какое-нибудь из вышеприведенных условий не будет соблюдено.

Версии этой инструкции для байтов и слов также выиграют от соблюдений данных условий, но они менее эффективны, чем версии для двойных слов.

REP STOSD оптимальна при тех же условиях, что и REP MOVSD.

REP LOADS, REP SCAS и REP CMPS не оптимальны, и их можно заменить на циклы. Смотри пример 1.10, 2.8 и 2.9 для поиска альтернатив REPNE SCASB. REP CMPS может вызвать конфликт баноков кэша, если биты 2-4 одинаковы в ESI и EDI.

26.4 Тестирование битов (все процессоры)

Инструкции BT, BTC, BTR и BTS следует заменять инструкциями типа TEST, AND, OR, XOR или сдвигами на PPlain и PMMX. На PPro, PII и PIII битовых тестов операнда в памяти следует избегать.

26.5 Целочисленное умножение (все процессоры)

Целочисленное умножение занимает до 9 тактов на PPlain и PMMX и до 4 тактов на PPro, PII и PIII. Поэтому часто выгоднее бывает заменить умножение на константу и комбинацию других инструкций, таких как SHL, ADD, SUB и LEA.

Пример:

```
IMUL EAX,10
```

можно заменить на

```
MOV EBX,EAX / ADD EAX,EAX / SHL EBX,3 / ADD EAX,EBX
```

или

```
LEA EAX,[EAX+4*EAX] / ADD EAX,EAX
```

Умножение чисел с плавающей запятой быстрее, чем целочисленное умножение на PPlain и PMMX, но время, затрачиваемое на преобразование целых чисел в числа с плавающей запятой и обратную конвертацию полученного результата, обычно больше, чем время, сэкономленное в результате использования умножения с плавающей запятой, не считая тех случаев, когда количество конвертаций несравнимо с количеством умножений. Умножение MMX достаточно быстро, но доступно только для 16-ти битных операндов.

26.6 Инструкция WAIT (все процессоры)

Зачастую вы сможете повысить скорость, пренебрегнув инструкцией WAIT. Эта инструкция имеет три функции:

1. Старый процессор 8087 требовали инструкцию WAIT перед каждой инструкцией с плавающей запятой, чтобы убедиться, что сопроцессор готов ее получить.

2. WAIT используется для координирования доступа памяти между модулем вычислений плавающей запятой и модулем целочисленных вычислений.
3. WAIT иногда используется, чтобы следить за исключениями. Он сгенерирует прерывание, если бит исключения в слове статуса FPU был установлен предыдущей операцией плавающей запятой.

Примеры:

```
b.1.  FISTP [mem32]
      WAIT             ; ждем, пока FPU запишет в память, а потом..
      MOV EAX, [mem32] ; считываем результат модулем целочисленных вычислений

b.2.  FILD [mem32]
      WAIT             ; ждем, пока FPU считает значение из памяти..
      MOV [mem32],EAX  ; перед ее перезаписью целым числом

b.3.  FLD QWORD PTR [ESP]
      WAIT             ; предотвращаем случайную ошибку от..
      ADD ESP,8        ; перезаписи значения в стеке
```

Относительно a:

Эта функция WAIT была актуальна только на старом 8087. Если только вы не хотите, чтобы ваш код был совместим с 8087, вам следует указать вашему ассемблеру, чтобы он не помещал эти WAIT'ы, задав более современный процессор. Эмулятор плавающей запятой 8087 также вставляет инструкции WAIT, поэтому вам следует указать вашему ассемблеру не генерировать код эмуляции, если только он вам действительно не нужен.

Относительно b:

Инструкции WAIT для координации доступа к памяти были действительно нужны на 8087 и 80287, но на Pentium'ах она в этом качестве совершенно не обязательна. Что касается 80386 и 80486, тут ситуация не совсем ясна. Я сделал несколько тестов на этих интеловских процессорах и не смог спровоцировать ни одной ошибки, пропустив WAIT, на любом из 32-х битных интеловских процессоров, хотя руководства от Intel говорят, что WAIT необходима для этой цели, не считая инструкций FNSTSW и FNSTCW. Пропус инструкций WAIT для координирования доступа к памяти не 100% надежно даже при написании 32-х битного кода, потому что код может быть выполнен на очень редкой комбинации 80386 процессора с 287 сопроцессором, который требует WAIT. Также у меня нет информации о неинтеловских процессорах, и я не тестировал все возможные комбинации железа и программного обеспечения, поэтому могут быть ситуации, когда WAIT окажется нужен.

Если вы хотите быть уверены, что ваш коду будет работать на любом 32-х битном процессоре (включая неинтеловские процессоры), я рекомендую вам использовать WAIT в этом качестве на всякий случай.

Относительно c:

Ассемблер автоматически вставляет WAIT для этих целей перед следующими инструкциями: FCLEX, FINIT, FSAVE, FSTCW, FSTENV, FSTSW. Вы можете пропустить WAIT, написав FNCLEX и т.п. Мои тесты показывают, что в большинстве случаев WAIT не нужен, потому что эти инструкции без WAIT все равно будут генерировать прерывания или исключения, кроме FNCLEX и FNINIT на 80387. (Есть некоторая неопределенность, касаемая того, указывает ли IRET от прерывания на инструкцию FN.. или на следующую инструкцию).

Почти все инструкции плавающей запятой будут также генерировать прерывание, если

предыдущая инструкция плавающей запятой установила бит исключений, поэтому исключение рано или поздно будет обнаружено. Вы можете вставить WAIT после последней инструкции плавающей запятой в вашей программе, чтобы точно поймать все исключения.

Вам все еще может понадобиться WAIT, если нужно точно знать, где случается исключение, чтобы проконтролировать ситуацию. Возьмем, например, код из b.3: если вы хотите проконтролировать исключение, которое сгенерирует в подобной ситуации FLD, вам нужен WAIT, потому что прерывание после ADD ESP,8 перезапишет значение, которое надо загрузить. FNOP будет быстрее, чем WAIT, и предназначается для той же цели.

26.7 FCOM + FSTSW AX (все процессоры)

Инструкция FNSTSW очень медленна на любых процессорах. У процессоров PPro, PII и PIII есть инструкции FCOMI, чтобы избежать этой инструкции. Использование FCOMI вместо обычной последовательности FCOM / FNSTSW AX / SAHF сэкономит вам 8 тактов. Поэтому вам следует использовать FCOMI, чтобы избегать FNSTSW везде, где это возможно, даже если это будет стоить дополнительного кода.

На процессорах без инструкции FCOMI обычной практикой сравнения значений с плавающей запятой является:

```
FLD [a]
    FCOMP [b]
    FSTSW AX
    SAHF
    JB ASmallerThanB
```

Вы можете улучшить этот код, использовал FNSTSW AX вместо FSTSW AX и протестировав AH напрямую, а не используя неспариваемый SAHF (у TASM 3.0 есть баг, связанный с инструкцией FNSTSW AX):

```
FLD [a]
    FCOMP [b]
    FNSTSW AX
    SHR AH,1
    JC ASmallerThanB
```

Тестирование на ноль или равенство:

```
FTST
    FNSTSW AX
    AND AH,40H
    JNZ IsZero     ; (флаг нуля инвертирован!)
```

Проверка, больше ли одно значение другого:

```
FLD [a]
    FCOMP [b]
    FNSTSW AX
    AND AH,41H
    JZ AGreaterThanB
```

Не используйте TEST AH,41H, так как он не спаривается на PPLain и PMMX.

На PPlain и PMMX инструкция FNSTSW занимает 2 такта, но она вызывает задержку в дополнительные 4 такта после любой инструкции с плавающей запятой, потому что она ждет слово статуса FPU. Этого не происходит после целочисленных инструкций. Вы можете заполнить промежуток между FCOM и FNSTSW целочисленными

инструкциями на 4 такта. Спаренный FXCH сразу после FCOM не задерживает FNSTSW, даже если спаривание несовершенно:

```
FCOM                  ; такт 1
    FXCH                  ; такты 1-2 (несовершенное спаривание)
    INC DWORD PTR [EBX]   ; такты 3-5
    FNSTSW AX             ; такты 6-7
```

Вы можете здесь использовать FCOM вместо FTST, потому что FTST не спаривается. Не забудьте включить N в FNSTSW. У FSTSW (без N) префикс WAIT, который задержит ее в дальнейшем.

Иногда быстрее использовать целочисленные инструкции для сравнения значений с плавающей запятой, как это объяснено в главе 27.6.

26.8 FPREM (все процессоры)

Инструкции FPREM и FPREM1 медленны на всех процессорах. Вы можете заменить их следующим алгоритмом: умножьте на противоположный делитель, получайте дробную часть, получите дробную часть, вычитая усеченное значение, затем умножьте на делитель (смотрите главу 27.5, чтобы узнать, как усекать значения).

Некоторые документы говорят, что эти инструкции могут давать неполную редукцию, и поэтому необходимо повторять инструкции FPREM и FPREM1, пока она не будет сделана. Я протестировал это на нескольких процессорах, начиная со старого 8087, и у меня не было ни одной ситуации, когда потребовалось бы повторение FPREM или FPREM1.

26.9 FRNDINT (все процессоры)

Эта инструкция медленна на всех процессорах. Замените ее следующим:

```
FISTP QWORD PTR [TEMP]
    FILD  QWORD PTR [TEMP]
```

Этот код быстрее, несмотря на возможные потери из-за попытки считать [TEMP], когда запись еще не окончена. Рекомендуется поместить какие-нибудь другие инструкции.

26.10 FSCALE и экпоненциальная функция (все процессоры)

FSCALE медленна на всех процессорах. Посчитать целочисленные степени числа 2 можно гораздо быстрее, вставив желаемую степень в поле экспоненты числа с плавающей запятой. Чтобы посчитать 2N, где N - это целое число со знаком, выберите один из примеров ниже, который подходит под границы возможных значений вашего N.

Для |N| < 27-1 вы можете использовать одинарную точность:

```
MOV     EAX, [N]
    SHL     EAX, 23
    ADD     EAX, 3F800000H
```

```
    MOV     DWORD PTR [TEMP], EAX
    FLD     DWORD PTR [TEMP]
```

Для |N| < 210-1 вы можете использовать двойную точность:

```
MOV     EAX, [N]
    SHL     EAX, 20
    ADD     EAX, 3FF00000H
    MOV     DWORD PTR [TEMP], 0
    MOV     DWORD PTR [TEMP+4], EAX
    FLD     QWORD PTR [TEMP]
```

Для |N| < 214-1 используйте длинную двойную точность:

```
MOV     EAX, [N]
    ADD     EAX, 00003FFFH
    MOV     DWORD PTR [TEMP],   0
    MOV     DWORD PTR [TEMP+4], 80000000H
    MOV     DWORD PTR [TEMP+8], EAX
    FLD     TBYTE PTR [TEMP]
```

FSCALE часто используется в вычислениях экспоненциальных функций. Следующий код показывает экспоненциальную функцию без медленных FRNDINT и FSCALE:

```
; extern "C" long double _cdecl exp (double x);
_exp    PROC    NEAR
PUBLIC  _exp
        FLDL2E
        FLD     QWORD PTR [ESP+4]             ; x
        FMUL                                  ; z = x*log2(e)
        FIST    DWORD PTR [ESP+4]             ; round(z)
        SUB     ESP, 12

        MOV     DWORD PTR [ESP], 0
        MOV     DWORD PTR [ESP+4], 80000000H
        FISUB   DWORD PTR [ESP+16]            ; z - round(z)
        MOV     EAX, [ESP+16]
        ADD     EAX,3FFFH
        MOV     [ESP+8],EAX
        JLE     SHORT UNDERFLOW
        CMP     EAX,8000H
        JGE     SHORT OVERFLOW
        F2XM1
        FLD1
        FADD                                  ; 2^(z-round(z))
        FLD     TBYTE PTR [ESP]               ; 2^(round(z))

        ADD     ESP,12
        FMUL                                  ; 2^z = e^x
        RET

UNDERFLOW:
        FSTP    ST
        FLDZ                                  ; return 0
        ADD     ESP,12
        RET

OVERFLOW:
        PUSH    07F800000H                    ; +infinity
        FSTP    ST
        FLD     DWORD PTR [ESP]               ; return infinity
        ADD     ESP,16
        RET

_exp    ENDP
```

26.11 FPTAN (все процессоры)

Согласно руководствам, FPTAN возвращает два значения X и Y и оставляет на программиста деление Y на X для получения окончательного результата, но фактически она всегда возвращает в X 1, поэтому вы можете сэкономить на делении. Мои тесты показывают, что на всех 32-х битные интеловские процессоры с модулем плавающей запятой или сопроцессором, FPTAN всегда возвращает 1 в X независимо от аргумента. Если вы хотите быть абсолютно уверены, что ваш код будет выполняться корректно на всех процессорах, тогда вы можете протестировать, равен ли X одному, что быстрее, чем деление на X. Значение Y может быть очень высоко, но не бесконечно, поэтому вам не надо тестировать, содержит ли Y правильное число, если вы знаете, что аргумент верен.

26.12 FSQRT (PIII)

Быстрый способ вычислить приблизительное значение квадратного корня на PIII - это умножить обратный корень от x на сам x:

```
SQRT(x) = x * RSQRT(x)
```

Инструкция RSQRTSS или RSQRTPS дает обратный корень с точностью 12 бит. Вы можете улучшить точность до 23 бит, используя формулу Ньютона-Рафсона, использованную в интеловской сопроводительной заметке AP-803:

```
x0 = RSQRTSS(a)
x1 = 0.5 * x0 * (3 - (a * x0)) * x0)
```

где x0 - это первое приближение к обратному корню от a, а x1 - лучшее приближение. Порядок вычисления имеет значение. Вы можете использовать эту формулу до умножения, чтобы получить квадратный корень.

26.13 MOV [MEM], ACCUM (PPlain и PMMX)

Инструкции MOV [mem],AL, MOV [mem],AX, MOV [mem],EAX расцениваются механизмом спаривания как пишущие в аккумулятор. Поэтому следующие инструкции не спариваются:

```
MOV [mydata], EAX
    MOV EBX, EAX
```

Эта проблема возникает только в короткой версии инструкции MOV, у которой нет базы или индексного регистра и которой может быть только аккумулятор в качестве источника. Вы можете избежать проблему использованием другого регистра, перегруппировкой инструкций, использованием указателя или закодировав общую форму инструкции MOV самостоятельно.

В 32-х битном режиме вы можете записать основную форму MOV [mem],EAX следующим образом:

```
DB 89H, 05H

    DD OFFSET DS:mem
```

В 16-ти битном режиме вы можете записать основную форму MOV [mem],AX так:

```
DB 89H, 06H
    DW OFFSET DS:mem
```

Чтобы использовать AL вместо (E)AX, вам нужно заменить 89H на 88H.

Этот изъян не был исправлен в PMMX.

26.14 Инструкция TEST (PPlain и PMMX)

Инструкция TEST с числовым операндом спаривается только, если назначением являются AL, AX или EAX.

TEST регистр,регистр и TEST регистр,память всегда спаривается.

Пример:

```
TEST ECX,ECX               ; спаривается
    TEST [mem],EBX             ; спаривается
    TEST EDX,256               ; не спаривается
    TEST DWORD PTR [EBX],8000H  ; не спаривается
```

Чтобы сделать их спариваемыми, используйте один из следующих методов:

```
MOV EAX,[EBX] / TEST EAX,8000H
    MOV EDX,[EBX] / AND  EDX,8000H
    MOV AL,[EBX+1] / TEST AL,80H
    MOV AL,[EBX+1] / TEST AL,AL  ; (результат в флаге знака)
```

(Причина этой неспариваемости, вероятно, состоит в том, что первый байт двухбайтной инструкции та же самая, что и для неспариваемых инструкций, и процессор не может проверить второй байт во время проверки спариваемости.)

26.15 Битовое сканирование (PPlain и PMMX)

BSF и BSR - хуже всего оптимизированные инструкции на PPlain и PMMX, которые занимают приблизительно 11+2*n тактов, где n равен количеству пропущенных нулей.

Следующий код эмулирует BSR ECX,EAX:

```
        TEST    EAX,EAX
        JZ      SHORT BS1
        MOV     DWORD PTR [TEMP],EAX
        MOV     DWORD PTR [TEMP+4],0
        FILD    QWORD PTR [TEMP]
        FSTP    QWORD PTR [TEMP]
        WAIT    ; WAIT требуется только для совместимости со старым 286
                ; процессором

        MOV     ECX, DWORD PTR [TEMP+4]
        SHR     ECX,20        ; изолируем экспоненту
        SUB     ECX,3FFH      ; снижаем значение
        TEST    EAX,EAX       ; очищаем флаг нуля
BS1:
```

Следующий код эмулирует BSF ECX,EAX:

```
        TEST    EAX,EAX
        JZ      SHORT BS2
        XOR     ECX,ECX
        MOV     DWORD PTR [TEMP+4],ECX
```

```
        SUB     ECX,EAX
        AND     EAX,ECX
        MOV     DWORD PTR [TEMP],EAX
        FILD    QWORD PTR [TEMP]

        FSTP    QWORD PTR [TEMP]
        WAIT    ; WAIT требуется только для совместимости со старым 286
                ; процессором

        MOV     ECX, DWORD PTR [TEMP+4]
        SHR     ECX,20
        SUB     ECX,3FFH
        TEST    EAX,EAX       ; очищаем флаг нуля
BS2:
```

Этот код эмуляции не следует использовать PPro, PII и PIII, на которых инструкции битового сканирования занимают только 1 или 2 такта, и где данный код вызовет около двух задежек чтения памяти.

26.16 FLDCW (PPro, PII и PIII)

На PPro, PII и PIII инструкция `FLDCW` вызывает серьезную задержку, если за ней следует любая инструкция плавающей запятой, считывающая контрольное слово (как делают практически все инструкции плавающей запятой).

Компиляторы C или C++ часто генерируют множество инструкций `FLDCW`, потому что конвертация чисел с плавающей запятой в целые числа делается с помощью усечения, в то время как другие инструкции плавающей запятой используют округления. После перевода на ассемблер, вы можете улучшить код, использовав округление вместо усечения, где это возможно, или убрав `FLDCW` из цикла, если требуется усечение внутри него.

Смотрите главу 27.5, чтобы узнать, как сконвертировать число с плавающей запятой в целое без изменения контрольного слова.

Оптимизация для процессоров семейства Pentium: 27. Специальные темы

Автор: Агнер Фог, пер. Aquila

Дата: 2002-01-01

27.1 Инструкция LEA (все процессоры)

Инструкция LEA полезна для самых разных целей, потому что она умеет делать сдвиг, два сложения и перемещение за один такт.

Пример:

```
LEA EAX,[EBX+8*ECX-1000]
```

Гораздо быстрее, чем

```
MOV EAX,ECX / SHL EAX,3 / ADD EAX,EBX / SUB EAX,1000
```

Инструкцию LEA можно использовать, чтобы делать сложение или сдвиг без изменения флагов. Источник и назначение не обязательно должны быть размером в слово, поэтому 'LEA EAX,[BX]' может стать возможной заменой для 'MOVZX EAX,BX', хотя на многих процессорах это не совсем оптимально.

Как бы то ни было, вам следует знать, что инструкция LEA вызывает задержку AGI на PPlain и PMMX, если она использует базовый или индексный регистр, в которой была произведена запись в предыдущем такте.

Так как инструкция LEA спариваема в V-конвеер на PPlain и PMMX, а инструкции сдвига - нет, вы можете использовать LEA в качестве замены SHL на 1, 2 или 3, если вы хотите, чтобы инструкция выполнялась в V-конвеере.

У 32-х битных конвееров нет документированного режима адресации с только индексным регистром, поэтому инструкция LEA EAX,[EAX*2] на самом деле записывается как 'LEA EAX,[EAX*2+00000000] с 4-х байтовым смещением. Вы можете снизить размер инструкции, написав 'LEA EAX,[EAX+EAX]' или, что еще лучше, 'ADD EAX,EAX'. Последний вариант не приведет к задержке AGI на PPlain и PMMX. Если случилось так, что у вас есть регистр, равный нулю (например, счетчик цикла после последнего прохода), вы можете использовать его как базовый регистр, чтобы снизить размер кода:

```
LEA EAX,[EBX*4]     ; 7 байтов
LEA EAX,[ECX+EBX*4] ; 3 байтов
```

27.2 Деление (все процессоры) Деление отнимает очень много времени. На PPro, PII и PIII целочисленное деление занимает 19, 23 или 39 для байта, слова и двойного слова соответственно. На PPlain и PMMX беззнаковое челочисленное деление занимает приблизительно то же время, хотя деление со знаком занимает немного больше. Поэтому более предпочтительно использовать операнды маленького размера, которые не вызовут переполнения, даже если это будет стоить префикса размера операнда, и использовать по возможности беззнаковое деление.
Целочисленное деление на константу (все процессоры)

Целочисленное деление на степень от двух можно сделать, сдвигая значение вправо. Деление беззнакового целого числа на 2N:

```
        SHR     EAX, N
```

Деление целого числа со знаком на 2N:

```
        CDQ
```

```
AND     EDX, (1 SHL N) -1  ; или SHR EDX, 32-N
ADD     EAX, EDX
SAR     EAX, N
```

Альтернативный SHR короче, чем 'AND if N > 7, но может попасть только в порт 0 (или U-конвеер), в то время как AND может попасть как в порт 0, так и в порт 1 (U- или V-конвеер).

Деление на константу можно сделать на обратное число. Чтобы произвести беззнаковое целочисленное деление q = x / d, вам вначале нужно посчитать число, обратное делителю, f = 2r / d, где r определяет позицию двоично-десятичной точки (точка основания системы счисления). Затем нужно умножить x на f и сдвинуть полученный результат на r позиций вправо. Максимальное значение r равно 32+b, где b равно числу двоичных цифр в d минус 1. (b - это самое большое целое число, для которого 2b <= d). Используйте r = 32+b, чтобы покрыть максильное количество возможных значений делимого x.

Этот метод требует некоторых приемов, чтобы скомпенсировать ошибки округления. Следующий алгоритм даст вам верные результаты для деления беззнакового целого чила с усечением, то есть тот же результат, что дает инструкция DIV (спасибо Terje Mathisen, который изобрел этот метод):

```
b = (количество значимых битов в d) - 1
r = 32 + b
f = 2r / d
Если f - целое число, тогда d - это степень от 2: переходим к случаю A.
Если f - не целое число, тогда проверяем, меньше ли дробная часть f 0.5.

Если дробная часть f < 0.5: переходим к случаю B.
Если дробная часть f > 0.5: переходим к случаю C.

случай A: (d = 2b)
результат = x SHR b

случай B: (дробная часть f < 0.5)
округляем f вниз до ближайшего целого числа
результат = ((x+1) * f) SHR r

случай C: (дробная часть f > 0.5)
округляем f вверх до ближайшего целого числа
результат = (x * f) SHR r
```

Пример:

Предположите, что вы хотите разделить на 5.

```
5 = 00000101b.
b = (количество значимых двоичных чисел) - 1 = 2
r = 32+2 = 34

f = 234 / 5 = 3435973836.8 = 0CCCCCCCC.CCC...(hexadecimal)
```

Дробная часть больше, чем половина: используем случай C. Округляем f вверх до 0CCCCCCCDh.

Следующий код делит EAX на 5 и возвращает результат в EDX:

```
MOV     EDX,0CCCCCCCDh
MUL     EDX
SHR     EDX,2
```

После умножения EDX содержит значение, сдвинутое вправо на 32. Так как r = 34, вам нужно сдвинуть еще на 2, чтобы получить окончательный результат. Чтобы поделить на 10, вам нужно всего лишь заменить последнюю строку на 'SHR EDX,3'.

В случае B у вас будет следующее:

```
INC     EAX
MOV     EDX,f
MUL     EDX
SHR     EDX,b
```

Этот код работает для всех значений x, кроме 0FFFFFFFFH, которое дает ноль из-за переполнения в инструкции INC. Если возможно, что x = 0FFFFFFFFH, тогда замените этот код на:

```
        MOV     EDX,f
        ADD     EAX,1
        JC      DOVERFL
        MUL     EDX
DOVERFL:SHR     EDX,b
```

Если значение x ограничено, тогда вам следует использовать меньшее значение r, то есть меньшее количество цифр. Может быть несколько причин для того, чтобы сделать это:

• вы можете установить r = 32 и избежать 'SHR EDX,b' в конце.
• вы можете установить r = 16+b и использовать инструкции умножения, которые дают 32-х битный результат, вместо 64-х битного. Тогда можно освободить регистр EDX: IMUL EAX,0CCCDh / SHR EAX,18
• вы можете выбрать значение r, которое будет чаще приводить к случаю C, а не B, чтобы избежать инструкции 'INC EAX'.

Максимальное значение x в этих случаях равно по крайней мере 2r-b, иногда выше. Вы должны делать систематические тесты, если хотите узнать точное максимально значение x, при котором ваш код будет работать корректно.

Вы можете заменить медленную инструкцию умножения более быстрыми инструкциями, как это объяснено в главе 26.5.

Следующий пример делит EAX на 10 и возвращает результат в EAX. Я выбрал r=17, а не 19, потому что это дает код, который легче оптимизировать, и он покрывает такое же количество значений x. f = 217 / 10 = 3333h, случай B: q = (x+1)*3333h:

```
        LEA     EBX,[EAX+2*EAX+3]
        LEA     ECX,[EAX+2*EAX+3]
        SHL     EBX,4

        MOV     EAX,ECX
        SHL     ECX,8
        ADD     EAX,EBX
        SHL     EBX,8
        ADD     EAX,ECX
        ADD     EAX,EBX
        SHR     EAX,17
```

Проведенные тесты показываеют, что этот код работает правильно для всех значений x < 10004H.

Повторяемое деление целого числа на одно и то же значение (все процессоры)

Если делитель не известен во время ассемблирования программы, но вы делите на одно и то же число несколько раз, вы тоже можете использовать данный метод. Код должен определить, с каким случаем (A, B и C) он имеет дело, и высчитать f до совершения делений.

Следующий далее код показывает, как делать несколько делений на одно и то же число (беззнаковое деление с усечением). Сначала вызовите SET_DIVISOR, чтобы установить делитель и обратное ему число, затем вызовите DIVIDE_FIXED для каждого значения, которое нужно разделить на один и тот же делитель.

```
.data

RECIPROCAL_DIVISOR DD ?            ; округленное число, обратное делителю
CORRECTION         DD ?            ; случай A: -1, случай B: 1, случай C: 0
BSHIFT             DD ?            ; количество бит в делителе - 1

.code

SET_DIVISOR PROC NEAR              ; делитель в EAX
        PUSH    EBX
```

```
        MOV     EBX,EAX
        BSR     ECX,EAX             ; b = количество бит в делителе - 1
        MOV     EDX,1
        JZ      ERROR               ; ошибка: делитель равен нулю
        SHL     EDX,CL              ; 2^b
        MOV     [BSHIFT],ECX        ; сохраняем b
        CMP     EAX,EDX
        MOV     EAX,0
        JE      SHORT CASE_A        ; делитель - степень от 2
        DIV     EBX                 ; 2^(32+b) / d

        SHR     EBX,1               ; делитель / 2
        XOR     ECX,ECX
        CMP     EDX,EBX             ; сравниваем остаток с делителем/2
        SETBE   CL                  ; 1 если случай B
        MOV     [CORRECTION],ECX    ; коррекция возможных ошибок округления
        XOR     ECX,1
        ADD     EAX,ECX             ; добавляем 1 если случай C
        MOV     [RECIPROCAL_DIVISOR],EAX ; округленное число, обратное
                                         ; делителю
        POP     EBX
        RET
CASE_A: MOV     [CORRECTION],-1     ; запоминаем, что у нас случай A

        POP     EBX
        RET
SET_DIVISOR     ENDP

DIVIDE_FIXED PROC NEAR                  ; делимое в EAX, результат в EAX
        MOV     EDX,[CORRECTION]
        MOV     ECX,[BSHIFT]
        TEST    EDX,EDX
        JS      SHORT DSHIFT            ; делитель - степень от 2
        ADD     EAX,EDX                 ; коррекция возможных ошибок округления
        JC      SHORT DOVERFL           ; коррекция при переполнении
        MUL     [RECIPROCAL_DIVISOR]    ; умножаем на число, обратное делителю

        MOV     EAX,EDX
DSHIFT: SHR     EAX,CL                  ; сдвигаем на количество бит
        RET
DOVERFL:MOV     EAX,[RECIPROCAL_DIVISOR] ; делимое = 0FFFFFFFFH
        SHR     EAX,CL                  ; делаем деление с помощью сдвига
        RET
DIVIDE_FIXED    ENDP
```

Этот код даст тот же результат, что и инструкция DIV для 0 <= x < 232, 0 < d < 232.

Обратите внимание, что линия 'JC DOVERFL' и ее цель не нужны, если вы уверены, что x < 0FFFFFFFFH.

Если степени от 2 случаются так редко, что не стоит делать специальную оптимизацию из-за них, то вы можете убрать переход на DSHIFT и делать вместо него умножения с CORRECTION = 0 для случая A.

Если делитель меняется так часто, что процедура SET_DIVISOR нуждается в оптимизации, то вы можете заменить инструкцию BSR кодом, который приведен в главе 26.15 для процессоров PPlain и PMMX.

Деление чисел с плавающей запятой (все процессоры) Деление чисел с плавающей запятой занимает 38 или 39 тактов при самой высокой точности. Вы можете сэкономить время, указав более низкую точность в контрольном слове (на PPlain и PMMX только FDIV и FIDIV более быстры при низкой точности; на PPro, PII и PIII это также относится к FSQRT. Выполнение других инструкций убыстрить этим способом нельзя). **Параллельное деление (PPlain и PMMX)**

На PPlain и PMMX можно производить деление числа плавающей запятой и целочисленное деление параллельно. На PPro, PII и PIII это не возможно, потому что целочисленное деление и деление чисел с плавающей запятой используют один и тот же механизм.

Пример: A = A1 / A2; B = B1 / B2

```
FILD    [B1]
FILD    [B2]
MOV     EAX, [A1]
MOV     EBX, [A2]
CDQ
FDIV
DIV     EBX

FISTP   [B]
MOV     [A], EAX
```

Убедитесь, что вы установили в контрольном слове FPU желаемый метод округления.

Использование обратных инструкций для быстрого деления (PIII)

На PIII вы можете использовать быстрые обратные инструкции RCPSS или PCPPS с делителем, а затем умножить на делимое. Правда, точность будет всего 12 бит. Вы можете повысить ее до 23-х, использовав метод Ньютона-Рафсона, объясненного в интеловской сопроводительной заметке AP-803:

```
x0 = RCPSS(d)
x1 = x0 * (2 - d * x0) = 2*x0 - d * x0 * x0
```

где x0 - это первое приближение к обратному от делителя d, а x1 - лучшее приближение. Вы должны использовать эту формулу перед умножение на делимое:

```
MOVAPS  XMM1, [DIVISORS]        ; загружаем делители
RCPPS   XMM0, XMM1              ; приближенное обратное число
MULPS   XMM1, XMM0              ; формула Ньютона-Рафсона
MULPS   XMM1, XMM0

ADDPS   XMM0, XMM0
SUBPS   XMM0, XMM1
MULPS   XMM0, [DIVIDENDS]       ; результаты в XMM0
```

Это позволяет сделать 4 деления за 18 тактов с точностью 23 бита. Повышение точность, повторяя формулу Ньютона-Рафсона возможно, но не очень выгодно.

Если вы хотите использовать этот метод для целочисленных делений, тогда вам нужно проверять результаты на ошибки округления. Следующие код делает четыре деления с усечением на упакованных целых числах размером в слово за примерно 42 такта. Это дает точные результаты для 0 <= делимое < 7FFFFH и 0 < делитель lt;= 7FFFFH:

```
MOVQ MM1, [DIVISORS]      ; загружаем четыре делителя
MOVQ MM2, [DIVIDENDS]     ; загружаем четыре делимых
PUNPCKHWD MM4, MM1        ; распаковываем делители в DWORD'ы
PSRAD MM4, 16
PUNPCKLWD MM3, MM1
PSRAD MM3, 16
CVTPI2PS XMM1, MM4        ; конвертируем делители в плавающие числа,
                          ; (два верхних из них)
MOVLHPS XMM1, XMM1
CVTPI2PS XMM1, MM3        ; конвертируем нижние два операнда
PUNPCKHWD MM4, MM2        ; распаковываем делимые в DWORD'ы

PSRAD MM4, 16
PUNPCKLWD MM3, MM2
PSRAD MM3, 16
CVTPI2PS XMM2, MM4        ; конвертируем делимые d плавающие числа
                          ; (верхние два операнда)
MOVLHPS XMM2, XMM2
CVTPI2PS XMM2, MM3        ; конвертируем два нижних операнда
RCPPS XMM0, XMM1          ; приближенное обратное число делителей
MULPS XMM1, XMM0          ; улучшаем точность с помощью метода Ньютона-Рафсона
PCMPEQW MM4, MM4          ; создаем четыре целочисленных единицы за раз
PSRLW MM4, 15
```

```
MULPS XMM1, XMM0
ADDPS XMM0, XMM0
SUBPS XMM0, XMM1          ; обратные делители с точностью в 23 бита
MULPS XMM0, XMM2          ; умножаем на делимые
CVTTPS2PI MM0, XMM0       ; усекаем нижние два результата
MOVHLPS XMM0, XMM0
CVTTPS2PI MM3, XMM0       ; усекаем верхние два результата
PACKSSDW MM0, MM3         ; упаковываем четыре результата в MM0
MOVQ MM3, MM1             ; умножаем результаты на делители...

PMULLW MM3, MM0           ; чтобы выявить ошибки округления
PADDSW MM0, MM4           ; добавляем 1, чтобы скомпенсировать
                          ; последнее вычитание
PADDSW MM3, MM1           ; добавляем делитель. он должен быть больше
                          ; делимого
PCMPGTW MM3, MM2          ; проверяем, не слишком ли мал
PADDSW MM0, MM3           ; вычитаем 1, если это не так
MOVQ [QUOTIENTS], MM0     ; сохраняем четыре результата
```

Этот код проверяет, не слишком ли мал результат и делает соответствующую коррекцию. Не нужно проверять, если результат слишком велик.

Избегание делений (все процессоры)

Очевидно, что вам минимизировать количество делений. Деления плавающей запятой на константу или повторяющиеся деления на одно и то же значения следует делать через умножения на обратное число. Но есть много других ситуаций, когда вы можете снизить количество делений. Например: if (A/B >c) можно переписать как if (A > B*C), если B положительны, и как обратное сравнение, если B отрицательны.

A/B + C/D можно переписать как (A*D* + *C*B) / (B*D)

Если вы используете целочисленное деление, вам стоит остерегаться того, что ошибки округления могут стать другими после переписывания формул.

27.3 Освобождение регистров FPU (все процессоры)

Вы должны освободить все использованные регистры FPU до выходы из подпрограммы, не считая регистра, использованного для возвращения результата.

Самый быстрые способ освободить один регистр - это FSTP ST. Самый быстрый способ освбодить два регистра на PPlain и PMMX - это FCOMPP, на PPro, PII и PIII вы можете использовать как FCOMPP, так и FSTP ST дважды.

Не рекомендуется использовать FFREE.

27.4 Переход от инструкций FPU к MMX и обратно (PMMX, PII и PIII)

Вы должны выполнить инструкцию EMMS после инструкции MMX, за которой может последовать код с инструкциями FPU.

На PMMX переключение между инструкциями FPU и MMX вызывает высокие потери. Выполнение первой инструкции FPU после EMMS занимает примерно на 58 тактов больше, а первой инструкции MMX после инструкции FPU - на 38 тактов больше.

На PII и PIII подобных потерь нет. Задержку после EMMS можно скрыть, поместив целочисленные инструкции между EMMS и первой инструкции FPU.

27.5 Конвертации чисел с плавающей запятой в целые (все процессоры)

Все подобные конверсии и обратно должны осуществляться через память:

```
FISTP DWORD PTR [TEMP]
MOV EAX, [TEMP]
```

На PPro, PII и PIII этот код может вызвать потерит из-за попытки считать из [TEMP] до того, как закончена запись туда же, потому что инструкция FIST медленная (глава 17). WAIT не поможет (глава 26.6). Рекомендуется поместить другие инструкции между записью в [TEMP] и чтением оттуда, что бы избежать этих потерь. Это относится ко всем примерам, которые последуют в дальнейшем.

Спецификация языка C и C++ требует, чтобы конверсия из чисел с плавающей запятой в целые числа осуществлялась с помощью усечения, а не округления. Метод, используемый большинством библиотек C, это изменение контрольного слова FPU, чтобы указать инструкции FISTP на усечение, и изменение контрольного слова в прежнее состояние после ее выполнения. Это метод очень медленнен на всех процессорах. На PPro, PII и PIII контрольное слово FPU не может быть переименовано, поэтому все последующие инструкции плавающей запятой будут ждать, пока инструкция FLDCW не будет выведена из обращения.

Если вам нужно осуществить конверсию числа с плавающей запятой в C или C++, вам следует подумать о том, не лучше ли вам использовать округление вместо усечения. Если ваша стандартная библиотека не поддерживает быструю функцию округления, тогда сделайте свою собственную, используя примеры примеры, приведенные ниже.

Если вам нужно усечение внутри чикла, вы можете изменить контрольное слово за его пределами, если инструкции плавающей запятой внутри цикла могут корректно работать с данным режимом конвериторования.

Вы можете использовать различные способы, чтобы усекать без изменения контрольного слова, как это показано в примерах ниже. В данных примерах предполагается, что контрольное слово установлено по умолчанию, то есть округление к ближайшему.

Округление к ближайшему

```
; extern "C" int round (double x);
_round  PROC    NEAR
PUBLIC  _round
        FLD     QWORD PTR [ESP+4]
        FISTP   DWORD PTR [ESP+4]
        MOV     EAX, DWORD PTR [ESP+4]
        RET
_round  ENDP
```

Усечение к нулю

```
; extern "C" int truncate (double x);
_truncate PROC    NEAR
PUBLIC  _truncate
        FLD     QWORD PTR [ESP+4]   ; x
        SUB     ESP, 12             ; память для локальных переменных
        FIST    DWORD PTR [ESP]     ; округленное значение
        FST     DWORD PTR [ESP+4]   ; значение с плавающей запятой
        FISUB   DWORD PTR [ESP]     ; вычитаем округленное значение
        FSTP    DWORD PTR [ESP+8]   ; разность
        POP     EAX                 ; округленное значение

        POP     ECX                 ; значение с плавающей запятой
        POP     EDX                 ; разность (с плавающей запятой)
        TEST    ECX, ECX            ; тестируем знак x
        JS      SHORT NEGATIVE
        ADD     EDX, 7FFFFFFFH      ; устанавливаем флаг переноса, если
                                    ; разность меньше -0
        SBB     EAX, 0              ; вычитаем 1, если x-round(x) < -0
        RET
NEGATIVE:
        XOR     ECX, ECX
        TEST    EDX, EDX
        SETG    CL                  ; 1, если разность > 0
        ADD     EAX, ECX            ; добавляем 1, если x-round(x) > 0
        RET
```

```
_truncate ENDP
```

Усечение к минус бесконечности

```
; extern "C" int ifloor (double x);
_ifloor PROC    NEAR
PUBLIC  _ifloor
        FLD     QWORD PTR [ESP+4]   ; x
        SUB     ESP, 8              ; память для локальных переменных
        FIST    DWORD PTR [ESP]     ; округленное значение
        FISUB   DWORD PTR [ESP]     ; вычитаем округленное значение
        FSTP    DWORD PTR [ESP+4]   ; разность
        POP     EAX                 ; округленное значение

        POP     EDX                 ; разность (с плавающей запятой)
        ADD     EDX, 7FFFFFFFH      ; устанавливаем флаг переноса, если
                                    ; разность меньше -0
        SBB     EAX, 0              ; вычитаем 1, если x-round(x) < -0
        RET
_ifloor ENDP
```

Эти процедуры работают для -231 < x < 231-1. Они не проверяют на переполнение или NAN'ы.

У PIII есть инструкции для усечения чисел с плавающей запятой одинарной точности: CVTTSS2SI and CVTTPS2PI. Эти инструкции очень полезны, если одинарная точность вас удовлетворяет, но если вы конвертироуете число с более высокой точностью в число с одинарной точностью, чтобы использовать эти инструкции, у вас могут столкнуться с тем, что число может быть округлено вверх во время конверсии.

Альтернатива инструкции FISTP (PPlain и PMMX)

Конвертирование числа с плавающей запятой в целое обычно осуществляется следующим образом:

```
        FISTP   DWORD PTR [TEMP]
        MOV     EAX, [TEMP]
```

Альтернативный метод заключает в:

```
.DATA
ALIGN 8
TEMP    DQ      ?
MAGIC   DD      59C00000H   ; FPU-представление 2^51 + 2^52

.CODE
        FADD    [MAGIC]
        FSTP    QWORD PTR [TEMP]
        MOV     EAX, DWORD PTR [TEMP]
```

Добавление 'волшебного числа' 251+252 есть такой эффект, что любое целое число между -231 и +231 будет выравнено в нижних 32-х битах, когда сохраняется как число с плавающей запятой двойной точности. Результат будет такой же, какой бы вы получили с помощью инструкции FISTP со всеми методами округления, кроме усечения к нулю. Результат будет отличаться от FISTP, если в контрольном слове задано усечение или в случае переполнения. Вам может потребоваться инструкция WAIT для совместимости со старым 80287 процессором (глава 26.6)

Этот метод не быстрее использования FISTP, но он дает большую гибкость на PPlain и PMMX, потому между FADD и FSTP 3 такта, которые можно заполнить другими инструкциями. Вы можете умножить или разделить число на степень от друх в той же операции, сделав обратно по отношению к магическому числу. Вы также можете добавить константу, добавив ее к магическому числу, которое тогда будет иметь двойную точность.

27.6 Использование целочисленных инструкция для осуществления операций плавающей запятой (все процессоры)

Целочисленные операции в большинстве своем быстрее, чем инструкции плавающей запятой, поэтому зачастую выгоднее использовать их для осуществления простых операций плавающей запятой. Наиболее очевидное применение - это перемещение данных.

Пример:

```
FLD QWORD PTR [ESI] / FSTP QWORD PTR [EDI]
```

Заменить на:

```
MOV EAX,[ESI] / MOV EBX,[ESI+4] / MOV [EDI],EAX / MOV [EDI+4],EBX
```

Тестируем, не равно ли значение с плавающей запятой нулю:

Значение с плавающей запятой, равное нулю, обычно представляется как 32 или 64 обнуленных бита, но здесь есть один подводные камень: бит знака может быть равен нулю! Минус ноль считается правильным числом с плавающей запятой, и процессор может сгенерировать ноль с уставноленным битом знака, если, например, отрицательное число было умножено на ноль. Поэтому если вы хотите узнать, не равно ли число с плавающей запятой нулю, вам не следует тестировать бит знака.

Пример:

```
FLD DWORD PTR [EBX] / FTST / FNSTSW AX / AND AH,40H / JNZ IsZero
```

Используйте целочисленные инструкции вместо этого и сдвиньте бит знака:

```
MOV EAX,[EBX] / ADD EAX,EAX / JZ IsZero
```

Если число с плавающей запятой имеет двойную точность (QWORD), тогда вам нужно протестировать только биты 32-62. Если они равны нулю, тогда нижняя половина будет также равна нулю, если это верное число с плавающей запятой.

Тест на то, отрицательно ли значение:

Число с плавающей запятой отрицательно, если бит знака и установлен по крайней мере один бит.

Пример:

```
MOV EAX,[NumberToTest] / CMP EAX,80000000H / JA IsNegative
```

Манипулирование битом знака:

Вы можете изменить знак числа с плавающей запятой просто инвертироовав бит знака:

Пример:

```
XOR BYTE PTR [a] + (TYPE a) - 1, 80H
```

Похожим образом вы можете получить асбсолютное значение числа с плавающей запятой, просто cANDив бит знака:

Сравнивание чисел:

Числа с плавающей запятой сохраняются в особом формате, который позволяет использовать целочисленные инструкции для сравнения чисел с плавающей запятой, не считая бита знака. Если вы уверены, что оба сравниваемые числа с плавающей запятой являются правильными и положительными, вы можете простой сравнить их как целые:

Пример:

```
FLD [a] / FCOMP [b] / FNSTSW AX / AND AH,1 / JNZ ASmallerThanB
```

Изменить на:

```
MOV EAX,[a] / MOV EBX,[b] / CMP EAX,EBX / JB ASmallerThanB
```

Этот метод работает только, если у двух чисел одна и та же точность, и вы уверены, что ни у одного из числа не установлен бит знака.

Если возможны отрицательные числа, вы можете их сконвертировать определенным образом и сделать знаковое сравнение:

```
        MOV     EAX, [a]

        MOV     EBX, [b]
        MOV     ECX, EAX
        MOV     EDX, EBX
        SAR     ECX, 31             ; скопировать бит знака
        AND     EAX, 7FFFFFFFH      ; убрать бит знака
        SAR     EDX, 31
        AND     EBX, 7FFFFFFFH
        XOR     EAX, ECX      ; преобразуем, если установлен бит знака
        XOR     EBX, EDX
        SUB     EAX, ECX
        SUB     EBX, EDX
        CMP     EAX, EBX
        JL      ASmallerThanB       ; знаковое сравнение
```

Этот метод работает для всех правильных чисел с плавающей запятой, включая -0.

27.7 Использование инструкции с плавающей запятой, чтобы осуществлять целочисленные операции (PPlain и PMMX)

Целочисленное умножение (PPlain и PMMX)

Умножение плавающей запятой быстрее, чем целочисленное умножение на PPlain и PMMX, но цена конверсии целых чисел в числа с плавающей запятой и конвертирование результата обратно в целое число очень высоко, поэтому умножение плавающей запятой имеет смысл только тогда, если количество требуемых преобразование мало сравнимо с количеством умножений. (Довольно соблазнительно использование ненормальных чисел с плавающей запятой, чтобы пропустить часть конвертаций, но обработка таких чисел очень медленна, поэтому это плохая идея!)

На PMMX инструкции умножения MMX быстрее, чем целочисленное умножение, и могут конвееризоваться, поэтому одним из лучших решений на PMMX может быть использование этих инструкций на PMMX, если вы можете жить с 16-ти битной точностью.

Целочисленное умножение быстрее, чем умножение с плавающей запятой на PPro, PII и PIII.

Целочисленное деление (PPlain и PMMX)

Деление плавающей запятой не быстрее, чем целочисленное деление, но вы можете делать другие целочисленные операции (включая целочисленное деление, но не целочисленное умножение), в то время как работает FPU на деление плавающей запятой (смотри пример выше).

Конвертирование двоичных чисел в десятичные (все процессоры)

Использование инструкции FBSTP - простой и удобный путь для того, чтобы конвертировать двоичные числа в десятичные, хотя не обязательно самый быстрый метод.

27.8 Перемещение блоков данных (все процессоры)

Есть несколько способов перемещения блоков данных. Наиболее общий метод - это REP MOVSD, но при определенных условиях другие методы быстрее.

На PPlain и PMMX быстрее переместить 8 байтов за раз, если место назначения не находится в кэше:

```
TOP:    FILD    QWORD PTR [ESI]
        FILD    QWORD PTR [ESI+8]
        FXCH
        FISTP   QWORD PTR [EDI]
        FISTP   QWORD PTR [EDI+8]
```

```
        ADD     ESI, 16
        ADD     EDI, 16
        DEC     ECX
        JNZ     TOP
```

Источник и место назначения должны быть выравнены на 8. Дополнительное время, используемое медленными инструкциями FILD и FISTP компенсируется тем, что вам требуется сделать в два раза меньше операций записывания. Обратите внимание, что этот метод имеет преимущество только на PPlain и PMMX и только тогда, когда место назначения не находится в кэше первого уровня. Вы не можете использовать FLD и FSTP (без I) с противоположными последовательностями битов, потому что ненормальные числа обрабатываются медленно и не гарантируется, что они останутся неизмененными.

На PMMX, если назначение не находится в кэше, быстрее использовать инструкции MMX для перемещения восьми байтов за раз, если место назначения не находится в кэше.

```
TOP:    MOVQ    MM0,[ESI]
        MOVQ    [EDI],MM0
        ADD     ESI,8
        ADD     EDI,8
        DEC     ECX
        JNZ     TOP
```

Данный цикл не нужно оптимизировать или разворачивать, если ожидаются промахи кэша, потому что здесь узкое место - это доступ к памяти, а не выполнение инструкций.

На процессорах PPro, PII и PIII инструкция REP MOVSD особенно быстра, если соблюдены следующие условия:

• источник и назначение должны быть выравнены на 8
• направление должно быть вперед (очищен флаг направления)
• счетчик (ECX) должен быть больше или равен 64
• разность между EDI и ESI должна быть больше или равна 32

На PII быстрее использовать регистры MMX, если вышеприведенные условия не соблюдены и место назначения находится в кэше первого уровня. Цикл можно развернуть в два раза, а источник и назначение должны быть выравнены на 8.

На PIII самый быстрый путь перемещения данных - это использовать инструкцию MOVAPS, если вышеприведенные условия не соблюдены или если место назначения не находится в кэше первого или второго уровня:

```
        SUB     EDI, ESI
TOP:    MOVAPS  XMM0, [ESI]
        MOVAPS  [ESI+EDI], XMM0
        ADD     ESI, 16
        DEC     ECX
        JNZ     TOP
```

В отличии от FLD, MOVAPS может обрабатывать любую последовательность битов без всяких проблем. Помните, что источник и назначение должны быть выравнены на 16.

Если количество байтов, которые необходимо переместить, не кратно 16-ти, вы можете округлить его до числа, которое ближе всего к 16, и поместить несколько дополнительных байтов в конце буфера назначения, чтобы получить лишние байты. Если это невозможно, тогда вам необходимо переместить оставшиеся байты с помощью других методов.

На PIII у вас также есть опция прямой записи в RAM-память без вовлечения кэша, используя инструкцию MOVNTQ или MOVNTPS. Это может быть полезным, если вы не хотите, чтобы место назначение попало в кэш. MOVNTPS чуть-чуть быстрее, чем MOVNTQ.

27.9 Самомодифицирующийся код (все процессоры)

Потери при выполнении кода сразу после того, как тот был изменен, занимают примерно 19 тактов на PPlain, 31 на PMMX и 150-300 на PPro, PII и PIII. Процессоры 80486 и более ранние требуют переход между модифицирующим и модифицируемым кодом, чтобы очистить кэш кода.

Чтобы получить разрешение на модифицирование кода в защищенное операционной системе, вам потребуется вызвать специальные системные функции: в 16-битной Windows это ChangeSelector, в 32-х битной Windows - VirtualProtect и FlushInstructionCache (или поместить код в сегмент данных).

Самомодифицирующийся код не считается хорошим тоном программирования, но можно пойти на его применение, если выигрыш в скорости значителен.

27.10 Определение типа процессора (все процессоры)

Я думаю, что теперь достаточно очевидно, что оптимальное для одного процессора может не являться таковым для другого. Вы можете сделать несколько вариантов наиболее критичных участков кода вашей программы, чтобы они выполнялись максимально быстро на любом процессоре. Однако вам потребуется определить, на каком процессоре программа выполняется в настоящий момент. Если вы используете инструкции, которые не поддерживаются всеми процессорами, т.е. условные перемещения, FCOMI, инструкции MMX и XMM), то вы можете сначала проверить, поддерживает ли процессор данные инструкции. Процедура, приведенная ниже, проверяет тип процессора и поддерживаемые им технологии.

```
; задаем инструкцию CPUID, если она не известна ассемблеру:
CPUID   MACRO
        DB      0FH, 0A2H
ENDM

; Прототип C++:
; extern "C" long int DetectProcessor (void);

; возвращаемое значение:
; bits 8-11 = семья (5 для PPlain и PMMX, 6 для PPro, PII и PIII)
; bit  0 = поддерживаются инструкции FPU
; bit 15 = поддерживаются условные переходы и инструкция FCOMI
; bit 23 = поддерживаются инструкции MMX
; bit 25 = поддерживаются инструкции XMM

_DetectProcessor PROC NEAR

PUBLIC  _DetectProcessor
        PUSH    EBX
        PUSH    ESI
        PUSH    EDI
        PUSH    EBP
        ; определяем, поддерживает ли микропроцессор инструкцию CPUID
        PUSHFD
        POP     EAX
        MOV     EBX, EAX
        XOR     EAX, 1 SHL 21    ; проверяем, можно ли изменять бит CPUID
        PUSH    EAX
        POPFD
        PUSHFD
        POP     EAX
        XOR     EAX, EBX
        AND     EAX, 1 SHL 21
        JZ      SHORT DPEND      ; инструкция CPUID не поддерживается

        XOR     EAX, EAX
        CPUID                    ; получаем количество функций CPUID
        TEST    EAX, EAX
        JZ      SHORT DPEND      ; функция 1 CPUID не поддерживается
        MOV     EAX, 1
        CPUID                    ; получаем семью и особенности процессора
        AND     EAX, 000000F00H  ; семья
        AND     EDX, 0FFFFF0FFH  ; флаги особенностей
        OR      EAX, EDX         ; комбинируем биты
DPEND:  POP     EBP
```

```
        POP     EDI
        POP     ESI
        POP     EBX

        RET
_DetectProcessor ENDP
```

Обратите внимание, что некоторые операционные системы не позволяют использовать инструкции XMM. Информация о том, как узнать, поддерживает ли операционная система инструкции XMM, можно найти в интеловской инструкции AP-900: "Identifying support for Streaming SIMD Extensions in the Processor and Operating System". Больше информации о идентификации процессора можно найти в инструкции AP-485: "Intel Processor Identification and the CPUID Instruction".

Если ассемблер не поддерживает инструкции MMX, XMM, условного перемещения данных, можно использовать специальные макросы (www.agner.org/assem/macros.zip)

Оптимизация для процессоров семейства Pentium: 28. Список периодов выполнения инструкций для PPlain и PMMX

Автор: Агнер Фог, пер. Aquila

Дата: 2002-01-01

Пояснения:

Операнды: r = регистр, m = память, i = число, sr - сегментный регистр, m32 = 32-х битный операнд памяти и так далее.

Такты: указанные значения являются минимальными. Промахи кэша, невыравненность и исключения могут значительно увеличить количество требуемых для выполнений тактов.

Спариваемость: u = спаривается в u-конвеере, v = спаривается в v-конвеере, uv = спаривается в любом конвеере, np = не спаривается.

28.1 Целочисленные инструкции (PPlain и PMMX)

Инструкции	Операнды	Такты	Спариваемость
NOP		1	uv
MOV	r/m, r/m/i	1	uv
MOV	r/m, sr	1	np
MOV	sr , r/m	>= 2 b)	np
MOV	m , accum	1	uv h)
XCHG	(E)AX, r	2	np
XCHG	r , r	3	np
XCHG	r , m	>15	np
XLAT		4	np
PUSH	r/i	1	uv
POP	r	1	uv
PUSH	m	2	np
POP	m	3	np
PUSH	sr	1 b)	np
POP	sr	>= 3 b)	np
PUSHF		3-5	np
POPF		4-6	np
PUSHA POPA		5-9 i)	np
PUSHAD POPAD		5	np
LAHF SAHF		2	np
MOVSX MOVZX	r , r/m	3 a)	np
LEA	r , m	1	uv
LDS LES LFS LGS LSS	m	4 c)	np
ADD SUB AND OR XOR	r , r/i	1	uv
ADD SUB AND OR XOR	r , m	2	uv
ADD SUB AND OR XOR	m , r/i	3	uv
ADC SBB	r , r/i	1	u
ADC SBB	r , m	2	u
ADC SBB	m , r/i	3	u
CMP	r , r/i	1	uv
CMP	m , r/i	2	uv
TEST	r , r	1	uv
TEST	m , r	2	uv
TEST	r , i	1	f)
TEST	m , i	2	np
INC DEC	r	1	uv
INC DEC	m	3	uv

NEG NOT	r/m	1/3	np
MUL IMUL	r8/r16/m8/m16	11	np
MUL IMUL	all other versions	9 d)	np
DIV	r8/m8	17	np
DIV	r16/m16	25	np
DIV	r32/m32	41	np
IDIV	r8/m8	22	np
IDIV	r16/m16	30	np
IDIV	r32/m32	46	np
CBW CWDE		3	np
CWD CDQ		2	np
SHR SHL SAR SAL	r , i	1	u
SHR SHL SAR SAL	m , i	3	u
SHR SHL SAR SAL	r/m, CL	4/5	np
ROR ROL RCR RCL	r/m, 1	1/3	u
ROR ROL	r/m, i(><1)	1/3	np
ROR ROL	r/m, CL	4/5	np
RCR RCL	r/m, i(><1)	8/10	np
RCR RCL	r/m, CL	7/9	np
SHLD SHRD	r, i/CL	4 a)	np
SHLD SHRD	m, i/CL	5 a)	np
BT	r, r/i	4 a)	np
BT	m, i	4 a)	np
BT	m, i	9 a)	np
BTR BTS BTC	r, r/i	7 a)	np
BTR BTS BTC	m, i	8 a)	np
BTR BTS BTC	m, r	14 a)	np
BSF BSR	r , r/m	7-73 a)	np
SETcc	r/m	1/2 a)	np
JMP CALL	short/near	1 e)	v
JMP CALL	far	>= 3 e)	np
conditional jump	short/near	1/4/5/6 e)	v
CALL JMP	r/m	2/5 e	np
RETN		2/5 e	np
RETN	i	3/6 e)	np
RETF		4/7 e)	np
RETF	i	5/8 e)	np
J(E)CXZ	short	4-11 e)	np
LOOP	short	5-10 e)	np
BOUND	r , m	8	np
CLC STC CMC CLD STD		2	np
CLI STI		6-9	np
LODS		2	np
REP LODS		7+3*n g)	np
STOS		3	np
REP STOS		10+n g)	np
MOVS		4	np
REP MOVS		12+n g)	np
SCAS		4	np
REP(N)E SCAS		9+4*n g)	np
CMPS		5	np
REP(N)E CMPS		8+4*n g)	np
BSWAP		1 a)	np
CPUID		13-16 a)	np
RDTSC		6-13 a) j)	np

Примечания:

a) у этой инструкции есть префикс OFH, который занимает дополнительный такт на PPlain, если до этого не было мультитактовой инструкции (смотри главу 12).
b) у версий с FS и GS есть префикс OFH, смотри примечание a.
c) у версий с SS, FS и GS есть префикс OFH, смотри примечание a.
d) у версий с двумя операндами (не числами) есть префикс OFH, смотри

примечание a.
e) смотри главу 22
f) спаривается, только если в качестве приемника регистр, смотри главу 26.14.
g) добавляет один такт для раскодировки префикса повторения, если ранее не предшествовала мультитактовая инстукция (такая как CLD, например, смотри главу 12).
h) спаривается, как если бы производилась запись в приемник, смотри главу 26.14.
i) 9, если SP кратно 4. Смотри главу 10.2.
j) на PPlain: 6 в привилигированном или реальном режиме, 11 в непривилигированном, ошибка в виртуальном. На PMMX: 8 и 13 тактов соответственно.

28.2 Инструкции FPU (PPlain и PMMX)

Пояснения:

Операнды: r = регистр, m = память, sr - сегментный регистр,
m32 = 32-х битный операнд памяти и так далее.

Такты: указанные значения являются минимальными. Промахи кэша, невыравненность, ненормальные операнды и исключения могут значительно увеличить количество требуемых для выполнений тактов.

Pairability:
• = pairable with FXCH, np = not pairable with FXCH.

Спариваемость: + = спариваемо с FXCH, np = не спариваемо с FXCH.

i-ov: пересечение времени выполнения с целочисленными инструкциями. i-ov = 4 означает, что последние четыре такта могут пересекаться с последующими целочисленными инструкциями.

fp-ov: пересечение времени выполнения с инструкциями FPU. fp-ov = 2 означает, что последние два такта могут пересекаться с последующими инструкциями FPU (WAIT здесь считается как инструкция FPU).

Инструкция	Операнд	Такты	Спариваемость	i-ov	fp-ov
FLD	r/m32/m64	1	+	0	0
FLD	m80	3	np	0	0
FBLD	m80	48-58	np	0	0
FST(P)	r	1	np	0	0
FST(P)	m32/m64	2 m)	np	0	0
FST(P)	m80	3 m)	np	0	0
FBSTP	m80	148-154	np	0	0
FILD	m	3	np	2	2
FIST(P)	m	6	np	0	0
FLDZ FLD1		2	np	0	0
FLDPI FLDL2E etc.		5 s)	np	2	2
FNSTSW	AX/m16	6 q)	np	0	0
FLDCW	m16	8	np	0	0
FNSTCW	m16	2	np	0	0
FADD(P)	r/m	3	+	2	2
FSUB(R)(P)	r/m	3	+	2	2
FMUL(P)	r/m	3	+	2	2 n)
FDIV(R)(P)	r/m	19/33/39 p)	+	38 o)	2
FCHS FABS		1	+	0	0
FCOM(P)(P) FUCOM	r/m	1	+	0	0
FIADD FISUB(R)	m	6	np	2	2
FIMUL	m	6	np	2	2
FIDIV(R)	m	22/36/42 p)	np	38 o)	2
FICOM	m	4	np	0	0
FTST		1	np	0	0
FXAM		17-21	np	4	0
FPREM		16-64	np	2	2

FPREM1		20-70	np	2	2
FRNDINT		9-20	np	0	0
FSCALE		20-32	np	5	0
FXTRACT		12-66	np	0	0
FSQRT		70	np	69 o)	2
FSIN FCOS		65-100 r)	np	2	2
FSINCOS		89-112 r)	np	2	2
F2XM1		53-59 r)	np	2	2
FYL2X		103 r)	np	2	2
FYL2XP1		105 r)	np	2	2
FPTAN		120-147 r)	np	36 o)	0
FPATAN		112-134 r)	np	2	2
FNOP		1	np	0	0
FXCH	r	1	np	0	0
FINCSTP FDECSTP		2	np	0	0
FFREE	r	2	np	0	0
FNCLEX		6-9	np	0	0
FNINIT		12-22	np	0	0
FNSAVE	m	124-300	np	0	0
FRSTOR	m	70-95	np	0	0
WAIT		1	np	0	0

Примечания:

m) значения, которое нужно сохранить, должно быть готово на один такт раньше.
n) 1, если пересекающаяся инструкция тоже FMUL.
o) не может пересекаться с инструкциями целочисленного умножения.
p) FDIV занимает 19, 33 или 39 тактов для 24, 53 и 64-х битной точности соответственно. FIDIV занимает на 3 такта больше. Точность задается битами 8-9 контрольного слова FPU.
r) такты типичны. Тривиальные случаи могут быть быстрее, крайние - медленнее.
s) может быть на 3 такта больше, когда требуется выходной результат FST, FCHS или FABS.

28.3 Инструкции MMX (PMMX)

Список периодов выполнения инструкций MMX не требуется, потому что они все занимают один такт, кроме инструкций умножения MMX, которые занимают три такта. Время выполнения инструкций умножения MMX может пересекаться и конвееризироваться, поэтому можно добиться производительности в одно умножение за такт.

Инструкция EMMS занимает только один такт, но первая инструкция FPU после EMMS занимает примерно на 58 тактов больше, а первая инструкция MMX после инструкции FPU занимает примерно на 38 тактов больше.

Нет потерь при использовании операндов памяти в инструкции MMX, потому что арифметический модуль MMX на один шаг дальше в конвеере, чем модуль загрузки. Но потери будут, когда вы станете сохранять данные из регистра MMX в память или в 32-х битный регистр: данные должны быть готовы на один такт раньше. Это аналогично инструкциям FPU.

Все инструкции MMX, кроме EMMS, спариваются в любом конвеере. Правила спаривания для инструкций MMX объяснены в главе 10.

Оптимизация для процессоров семейства Pentium: 29. Список периодов выполнения инструкций и задержек микроопераций для PPro, PII и PIII

Автор: Агнер Фог, пер. Aquila

Дата: 2012-11-24

Пояснения:

Операнды: r = регистр, m = память, i = число, sr = сегментный регистр, m32 = 32-х битный операнд в памяти и так далее.

Микрооперации: количество микроопераций, которые генерит инструкция для каждого порта выполнения.

• p0: порт 0: ALU и т.д.
• p1: порт 1: ALU, переходы
• p01: инструкции, которые могут попасть как в порт 0, так и в порт 1 (какой будет свободен первым)
• p2: порт 2: загрузка данных и т.д.
• p3: порт 3: генерация адреса для сохранения
• p4: порт 4: сохранение данных

Задержка:

Это задержка, которую генерирует инструкция в цепочке зависимости. (Это не то же самое, что время, потраченное в модуле выполнения. Значения могут быть неточны в тех ситуациях, когда они не могут быть измерены точно, особенно, что касается операндов в памяти). Значения являются минимальными. Промахи кэша, невыравненность и исключения могут повысить количество тактов весьма значительно. Предполагается, что операнды с плавающей запятой являются нормальными. Ненормальные числа, NAN'ы и бесконечность увеличивают задержку на 50-150 тактов, кроме перемещений XMM, перемешиваний и булевых инструкций. Переполнения FPU, потеря значимости, ненормальные или NAN-результаты дают аналогичную задержку.

Производительность: максимальная производительность для нескольких инструкций одного вида. Например, производительность в 1/2 для FMUL означает, что новая инструкция FMUL может начинать выполнение каждые 2 такта.

29.1 Целочисленные инструкции (PPro, PII и PIII)

Инструкции	Операнды	микрооперации						задержка	производительность
		p0	p1	p01	p2	p3	p4		
NOP				1					
MOV	r,r/i			1					
MOV	r,m				1				
MOV	m,r/i					1	1		
MOV	r,sr			1					
MOV	m,sr			1		1	1		
MOV	sr,r	8						5	
MOV	sr,m	7			1			8	
MOVSX MOVZX	r,r			1					
MOVSX MOVZX	r,m				1				
CMOVcc	r,r	1		1					
CMOVcc	r,m	1		1	1				
XCHG	r,r			3					
XCHG	r,m			4	1	1	1	high b)	
XLAT				1	1				
PUSH	r/i			1		1	1		

POP	r		1	1					
POP	(E)SP		2	1					
PUSH	m		1	1	1	1			
POP	m		5	1	1	1			
PUSH	sr		2		1	1			
POP	sr		8	1					
PUSHF(D)		3	11		1	1			
POPF(D)		10	6	1					
PUSHA(D)			2		8	8			
POPA(D)			2	8					
LAHF SAHF			1						
LEA	r,m	1					1 c)		
LDS LES LFS LGS									
LSS	m		8	3					
ADD SUB AND OR XOR	r,r/i		1						
ADD SUB AND OR XOR	r,m		1	1					
ADD SUB AND OR XOR	m,r/i		1	1	1	1			
ADC SBB	r,r/i		2						
ADC SBB	r,m		2	1					
ADC SBB	m,r/i		3	1	1	1			
CMP TEST	r,r/i		1						
CMP TEST	m,r/i		1	1					
INC DEC NEG NOT	r		1						
INC DEC NEG NOT	m		1	1	1	1			
AAS DAA DAS		1							
AAD		1	2				4		
AAM		1	1	2			15		
MUL IMUL	r,(r),(i)	1					4		1/1
MUL IMUL	(r),m	1		1			4		1/1
DIV IDIV	r8	2	1				19		1/12
DIV IDIV	r16	3	1				23		1/21
DIV IDIV	r32	3	1				39		1/37
DIV IDIV	m8	2	1	1			19		1/12
DIV IDIV	m16	2	1	1			23		1/21
DIV IDIV	m32	2	1	1			39		1/37
CBW CWDE			1						
CWD CDQ		1							
SHR SHL SAR ROR									
ROL	r,i/CL	1							
SHR SHL SAR ROR									
ROL	m,i/CL	1		1	1	1			
RCR RCL	r,1	1	1						
RCR RCL	r8,i/CL	4	4						
RCR RCL	r16/32,i/CL	3	3						
RCR RCL	m,1	1	2	1	1	1			
RCR RCL	m8,i/CL	4	3	1	1	1			
RCR RCL	m16/32,i/CL	4	2	1	1	1			
SHLD SHRD	r,r,i/CL	2							
SHLD SHRD	m,r,i/CL	2	1	1	1	1			
BT	r,r/i		1						
BT	m,r/i	1	6	1					
BTR BTS BTC	r,r/i		1						
BTR BTS BTC	m,r/i	1	6	1	1	1			
BSF BSR	r,r		1	1					
BSF BSR	r,m		1	1	1				
SETcc	r		1						
SETcc	m		1		1	1			
JMP	short/near		1						1/2
JMP	far	21		1					
JMP	r		1						1/2
JMP	m(near)		1		1				1/2
JMP	m(far)	21		2					
conditional jump	short/near		1						1/2
CALL	near		1	1		1	1		1/2

CALL	far	28			1	2	2			
CALL	r		1	2		1	1			1/2
CALL	m(near)		1	4	1	1	1			1/2
CALL	m (far)	28			2	2	2			
RETN			1	2	1					1/2
RETN	i		1	3	1					1/2
RETF		23			3					
RETF	i	23			3					
J(E)CXZ	short		1	1						
LOOP	short	2	1	8						
LOOP(N)E	short	2	1	8						
ENTER	i,0			12		1	1			
ENTER	a,b	ca. 18+4b				b-1	2b			
LEAVE				2	1					
BOUND	r,m	7		6	2					
CLC STC CMC				1						
CLD STD				4						
CLI		9								
STI		17								
INTO				5						
LODS					2					
REP LODS				10+6n						
STOS					1	1	1			
REP STOS				ca. 5n a)						
MOVS				1	3	1	1			
REP MOVS				ca. 6n a)						
SCAS				1	2					
REP(N)E SCAS				12+7n						
CMPS				4	2					
REP(N)E CMPS				12+9n						
BSWAP		1		1						
CPUID		23-48								
RDTSC		31								
IN		18						>300		
OUT		18						>300		
PREFETCHNTA d)	m				1					
PREFETCHT0 d)	m				1					
PREFETCHT1 d)	m				1					
PREFETCHT2 d)	m				1					
SFENCE d)						1	1			1/6

Примечания:

a) быстро при определенных условиях: смотри главу 26.3.
b) смотри главу 26.1.
c) 3, если константа без базового или индексного регистра.
d) только PIII.

29.2 Инструкции FPU (PPro, PII и PIII)

Инструкции	Операнды	микрооперации					задержка	производительность
		p0	p1	p01	p2	p3 p4		
FLD	r	1						
FLD	m32/64				1		1	
FLD	m80	2			2			
FBLD	m80	38			2			
FST(P)	r	1						
FST(P)	m32/m64				1	1	1	
FSTP	m80	2			2	2		
FBSTP	m80	165			2	2		
FXCH	r						0	3/1 f)
FILD	m	3			1		5	
FIST(P)	m	2			1	1	5	
FLDZ		1						
FLD1 FLDPI FLDL2E etc.		2						

FCMOVcc	r	2					2		
FNSTSW	AX	3					7		
FNSTSW	m16	1			1	1			
FLDCW	m16	1	1	1			10		
FNSTCW	m16	1			1	1			
FADD(P) FSUB(R)(P)	r	1					3	1/1	
FADD(P) FSUB(R)(P)	m	1		1			3-4	1/1	
FMUL(P)	r	1					5	1/2	g)
FMUL(P)	m	1		1			5-6	1/2	g)
FDIV(R)(P)	r	1					38 h)	1/37	
FDIV(R)(P)	m	1		1			38 h)	1/37	
FABS		1							
FCHS		3					2		
FCOM(P) FUCOM	r	1					1		
FCOM(P) FUCOM	m	1		1			1		
FCOMPP FUCOMPP		1	1				1		
FCOMI(P) FUCOMI(P)	r	1					1		
FCOMI(P) FUCOMI(P)	m	1		1			1		
FIADD FISUB(R)	m	6		1					
FIMUL	m	6		1					
FIDIV(R)	m	6		1					
FICOM(P)	m	6		1					
FTST		1					1		
FXAM		1					2		
FPREM		23							
FPREM1		33							
FRNDINT		30							
FSCALE		56							
FXTRACT		15							
FSQRT		1					69		e,i)
FSIN FCOS		17-97					27-103		e)
FSINCOS		18-110					29-130		e)
F2XM1		17-48					66		e)
FYL2X		36-54					103		e)
FYL2XP1		31-53					98-107		e)
FPTAN		21-102					13-143		e)
FPATAN		25-86					44-143		e)
FNOP		1							
FINCSTP FDECSTP		1							
FFREE	r	1							
FFREEP	r	2							
FNCLEX			3						
FNINIT		13							
FNSAVE		141							
FRSTOR		72							
WAIT			2						

Примечания:

e) не конвеерируется
f) FXCH генерирует 1 микрооперацию, что делается с помощью переименования регистром, порты при этом не задействуются.
g) FMUL использует ту же схему, что и целочисленное умножение. Поэтому комбинированная производительность целочисленных умножений и умножений FPU равна 1 FMUL + 1 IMUL за 3 такта.
h) задержка FDIV зависит от заданной в контрольном слове точности: 64 бита дают задержку в 38 тактов, 53 - в 32 такта, 24 - в 18. Деление на степень от двух занимает 9 тактов. Производительность равна 1/(задержка-1).
i) быстрее для низкой точности.

29.3 Инструкции MMX (PII и PIII)

Инструкция	Операнды	микрооперации					задержка	производительность
		p0	p1	p01	p2	p3	p4	
MOVD MOVQ	r,r			1				2/1
MOVD MOVQ	r64,m32/64				1			1/1

MOVD MOVQ	m32/64,r64				1	1		1/1	
PADD PSUB PCMP	r64,r64			1				1/1	
PADD PSUB PCMP	r64,m64			1	1			1/1	
PMUL PMADD	r64,r64		1				3	1/1	
PMUL PMADD	r64,m64		1		1		3	1/1	
PAND PANDN POR									
PXOR	r64,r64			1				2/1	
PAND PANDN POR									
PXOR	r64,m64			1	1			1/1	
PSRA PSRL PSLL	r64,r64/i		1					1/1	
PSRA PSRL PSLL	r64,m64		1		1			1/1	
PACK PUNPCK	r64,r64		1					1/1	
PACK PUNPCK	r64,m64		1		1			1/1	
EMMS			11				6 k)		
MASKMOVQ d)	r64,r64			1		1	1	2-8	1/30-1/2
PMOVMSKB d)	r32,r64		1				1	1/1	
MOVNTQ d)	m64,r64				1	1		1/30-1/1	
PSHUFW d)	r64,r64,i		1				1	1/1	
PSHUFW d)	r64,m64,i		1		1		2	1/1	
PEXTRW d)	r32,r64,i		1	1			2	1/1	
PISRW d)	r64,r32,i		1				1	1/1	
PISRW d)	r64,m16,i		1		1		2	1/1	
PAVGB PAVGW d)	r64,r64			1			1	2/1	
PAVGB PAVGW d)	r64,m64			1	1		2	1/1	
PMINUB PMAXUB PMINSW									
PMAXSW d)	r64,r64			1			1	2/1	
PMINUB PMAXUB PMINSW									
PMAXSW d)	r64,m64			1	1		2	1/1	
PMULHUW d)	r64,r64		1				3	1/1	
PMULHUW d)	r64,m64		1		1		4	1/1	
PSADBW d)	r64,r64		2		1		5	1/2	
PSADBW d)	r64,m64		2		1	1	6	1/2	

Примечания:

d) только PIII.

k) Вы можете спрятать задержку, вставив другие инструкции между EMMS и последующими инструкциями FPU.

29.4 Инструкции XMM (PIII)

Инструкция	Операнды	микрооперации						задержка	производительность
		p0	p1	p01	p2	p3	p4		
MOVAPS	r128,r128			2				1	1/1
MOVAPS	r128,m128				2			2	1/2
MOVAPS	m128,r128					2	2	3	1/2
MOVUPS	r128,m128				4			2	1/4
MOVUPS	m128,r128		1			4	4	3	1/4
MOVSS	r128,r128			1				1	1/1
MOVSS	r128,m32			1	1			1	1/1
MOVSS	m32,r128					1	1	1	1/1
MOVHPS MOVLPS	r128,m64			1				1	1/1
MOVHPS MOVLPS	m64,r128					1	1	1	1/1
MOVLHPS MOVHLPS	r128,r128			1				1	1/1
MOVMSKPS	r32,r128	1						1	1/1
MOVNTPS	m128,r128					2	2		1/15-1/2
CVTPI2PS	r128,r64		2					3	1/1
CVTPI2PS	r128,m64		2		1			4	1/2
CVTPS2PI									
CVTTPS2PI	r64,r128		2					3	1/1
CVTPS2PI	r64,m128		1		2			4	1/1
CVTSI2SS	r128,r32		2		1			4	1/2
CVTSI2SS	r128,m32		2		2			5	1/2

CVTSS2SI					
CVTTSS2SI	r32,r128		1	3	1/1
CVTSS2SI	r32,m128		1	4	1/2
ADDPS SUBPS	r128,r128		2	3	1/2
ADDPS SUBPS	r128,m128		2	3	1/2
ADDSS SUBSS	r128,r128		1	3	1/1
ADDSS SUBSS	r128,m32		1	3	1/1
MULPS	r128,r128	2		4	1/2
MULPS	r128,m128	2		4	1/2
MULSS	r128,r128	1		4	1/1
MULSS	r128,m32	1		4	1/1
DIVPS	r128,r128	2		48	1/34
DIVPS	r128,m128	2		48	1/34
DIVSS	r128,r128	1		18	1/17
DIVSS	r128,m32	1		18	1/17
ANDPS ANDNPS ORPS					
XORPS	r128,r128		2	2	1/2
ANDPS ANDNPS ORPS					
XORPS	r128,m128		2	2	1/2
MAXPS MINPS	r128,r128		2	3	1/2
MAXPS MINPS	r128,m128		2	3	1/2
MAXSS MINSS	r128,r128		1	3	1/1
MAXSS MINSS	r128,m32		1	3	1/1
CMPccPS	r128,r128		2	3	1/2
CMPccPS	r128,m128		2	3	1/2
CMPccSS	r128,r128		1	3	1/1
CMPccSS	r128,m32		1	3	1/1
COMISS UCOMISS	r128,r128		1	1	1/1
COMISS UCOMISS	r128,m32		1	1	1/1
SQRTPS	r128,r128	2		56	1/56
SQRTPS	r128,m128	2		57	1/56
SQRTSS	r128,r128	2		30	1/28
SQRTSS	r128,m32	2		31	1/28
RSQRTPS	r128,r128	2		2	1/2
RSQRTPS	r128,m128	2		3	1/2
RSQRTSS	r128,r128	1		1	1/1
RSQRTSS	r128,m32	1		2	1/1
RCPPS	r128,r128	2		2	1/2
RCPPS	r128,m128	2		3	1/2
RCPSS	r128,r128	1		1	1/1
RCPSS	r128,m32	1		2	1/1
SHUFPS	r128,r128,i		2 1	2	1/2
SHUFPS	r128,m128,i		2	2	1/2
UNPCKHPS UNPCKLPS	r128,r128		2 2	3	1/2
UNPCKHPS UNPCKLPS	r128,m128		2	3	1/2
LDMXCSR	m32	11		15	1/15
STMXCSR	m32	6		7	1/9
FXSAVE	m4096	116		62	
FXRSTOR	m4096	89		68	

Оптимизация для процессоров семейства Pentium: 30. Тестирование скорости

Автор: Агнер Фог, пер. Aquila

Дата: 2002-01-01

У микропроцессоров семьи Pentium есть встроенный 64-х битный счетчик, который можно считать в EDX:EAX, используя инструкцию RDTSC (read time stamp counter). Эта инструкция очень полезна для того, чтобы точно узнать, сколько тактов занял кусок кода.

Программа ниже полезна для измерения количества тактов, которое занимает код. Программа выполняет код 10 раз и сохраняет 10 значений счетчика. Программу можно использовать как в 16-ти, так и в 32-х битном режиме на PPlain и PMMX.

```
;************Тестовая программа для PPlain и PMMX:*******************

ITER    EQU     10              ; количество повторений
OVERHEAD EQU    15              ; 15 для PPlain, 17 для PMMX

RDTSC   MACRO                   ; определяем инструкцию RDTSC
        DB      0FH,31H
ENDM
;************Data segment:*******************
.DATA                           ; сегмент данных
ALIGN   4
COUNTER DD      0               ; счетчик цикла
TICS    DD      0               ; временная переменная для значения счетчика

RESULTLIST  DD  ITER DUP (0)    ; список тестовых результатов
;************Code segment:*******************
.CODE                           ; сегмент кода
BEGIN:  MOV     [COUNTER],0     ; сбрасываем счетчик цикла
TESTLOOP:                       ; тестовый цикл
;*********Делаем здесь необходимую инициализацию:*****************
        FINIT
;************Конец инициализиции*******************
        RDTSC                   ; считываем значение счетчика тактов

        MOV     [TICS],EAX      ; сохраняем его
        CLD                     ; не спариваемая инструкция
REPT    8
        NOP                     ; 8 NOP'ов, чтобы избежать эффекта "затенения"
ENDM

;***Поместите здесь инструкции, которые нужно протестировать:***
        FLDPI                   ; это всего лишь пример
        FSQRT
        RCR     EBX,10
        FSTP    ST
;********Конец инструкций, которые нужно тестировать*************

        CLC                     ; инструкция против спаривания и затенения

        RDTSC                   ; снова читаем счетчик
        SUB     EAX,[TICS]      ; вычисляем разность
        SUB     EAX,OVERHEAD    ; вычитаем такты, которые использовали
                                ; подсобные инструкции (против спаривания и
                                ; затенения)
        MOV     EDX,[COUNTER]   ; счетчик цикла
        MOV     [RESULTLIST][EDX],EAX   ; сохраняем результат в таблице
        ADD     EDX,TYPE RESULTLIST     ; увеличиваем значение счетчика
        MOV     [COUNTER],EDX           ; сохраняем счетчик
        CMP     EDX,ITER * (TYPE RESULTLIST)
        JB      TESTLOOP                ; повторяем заданное количество раз
```

```
; вставьте здесь код, чтобы считать значения в RESULTLIST
```

Подсобные инструкции до и после тестируемого кода были включены, чтобы получить адекватные результаты на PPlain. CLD - это неспариваемая инструкция, которая была вставлена, чтобы порядок спаривания инструкций в первый раз будет такой же, как и во все остальные. Восемь инструкций NOP были вставлены, чтобы предотвратить возможные префиксы в тестируемом коде, которые могли бы раскодироваться в тени предыдущих инструкций на PPlain. Однобайтовые инструкции использовались здесь, чтобы получить тот же порядок спаривания, что

На PMMX вы можете захотеть вставить 'XOR EAX,EAX / CPUID' перед тестируемым кодом, если вы хотите, чтобы FIFO-буфер инструкций был очищен или какую-нибудь длительную инструкцию (например, CLI или AAD), если вы хотите, чтобы он был полон (CPUID не вызывает эффекта затенения может начаться раскодировка префиксов последующих инструкций).

На PPro, PII и PIII вы можете вставить 'XOR EAX,EAX / CPUID' до и после каждого RDTSC, чтобы предотвратить ее возможное выполнение параллельно с какой-нибудь другой инструкцией и убрать подсобные инструкции. (CPUID - это синхронизирующая инструкция, что означает то, что она очищает конвеер и ждет, пока все исполняющиеся инструкции не будут выполнены, и только тогда выполнится сама. Это может быть полезно для тестирования.)

Инструкция RDTSC не может выполняться в виртуальном режиме на PPlain и PMMX, поэтому если вы запускете DOS-программы, вам нужно перегрузиться в реальный режим. (Нажмите F8 во время загрузки и выберите "safe mode command promt only" или "bypass startup files").

Полный исходник тестовой программы доступен на www.agner.org/assem/.

У процессоров Pentium есть специальные счетчики наблюдения за качеством работы, которые отслеживают и подсчитывают такие события как промахи кэша, невыравненность, различные задержки. Подробности об использовании счетчиков не затрагиваются в данном руководстве, но их можно найти в "Intel Architecture Software Developer's Manual", vol. 3, Appendix A.

Оптимизация для процессоров семейства Pentium: 31. Сравнение различных микропроцессоров

Автор: Агнер Фог, пер. Aquila

Дата: 2012-11-24

В следующей таблице изложены некоторые важные различия между процессорами семьи Pentium:

	PPlain	PMMX	PPro	PII	PIII
кэш кода, кб	8	16	8	16	16
кэш данных, кб	8	16	8	16	16
встроенный кэш 2 уровня, кб	0	0	256	512 *)	512 *)
инструкции MMX	нет	да	нет	да	да
инструкции XMM	нет	нет	нет	нет	да
инструкции условной пересылки данных	нет	нет	да	да	да
выполнение не по порядку	нет	нет	да	да	да
предсказывание переходов	плохо	хорошо	хорошо	хорошо	хорошо
количестов элементво в буфере предсказывания переходов	256	256	512	512	512
размер стекового буфера возвращений	0	4	16	16	16
потери при неправильном предсказании перехода	3-4	4-5	10-20	10-20	10-20
задержки чтения регистров	0	0	5	5	5
время ожидания FMUL	3	3	5	5	5
производительность FMUL	1/2	1/2	1/2	1/2	1/2
время ожидания IMUL	9	9	4	4	4
производительность IMUL	1/9	1/9	1/1	1/1	1/1

*) Селерон: 0-128, Xeon: 512 или больше, доступны другие варианты. На некоторых версиях кэш второго уровня выполняется на половинной скорости.

Комментарии к таблице:

Размер кэша кода важен, если критические части вашей программы занимают достаточно много места.

Размер кэша данных важен для всех программ, которые обрабатывают большое количество данных в критической части.

Инструкции MMX и XMM полезны для тех программ, которые обрабатывают большие массивы данных, такие как звук и изображения. Не всегда от использования инструкций MMX и XMM можно получить выгоду.

Инструкции условной пересылки данных полезны для того, чтобы избавиться от плохо предсказуемых условных переходов.

Выполнение не по порядку улучшает качество (особенно не оптимизированного кода). Оно включает в себя автоматическую перегруппировку и переименование регистров.

Процессоры с хорошим механизмом предсказания переходов умеют предсказывать простые повторяющиеся последовательности. Чем выше потери при неправильном предсказании переходов, тем важнее их предсказуемость.

Стековый буфер возвращений улучшает предсказание инструкций возвращений, когда подпрограмма вызывается из нескольких мест.

Задержки чтения регистров делают обработку смешанных типов данных (8, 16, 32 бита) более сложной.

Время ожидания инструкции умножения зависит от цепочки зависимости. Производительность 1/2 значит, что инструкция может конвееризироваться, поэтому новое умножение может начинаться каждый второй такт. Это определяет скорость обработки параллельных данных.

Большая часть методов оптимизации, приведенных в этом документе имеет небольшой или вообще не имеют отрицательного эффекта на другие микропроцессоры, включая неинтеловские.

Использование инструкций FPU на PPlain и PMMX часто требует большое количество дополнительных инструкций FXCH. Это затормозит выполнение на старых процессорах, но не на процессорах семейства Pentium и продвинутых неинтеловских процессорах.

Использование инструкций MMX на процессорах PMMX, PII и PIII или инструкций условной пересылки данных на PPro, PII и PIII создаст проблемы, если вы хотите, чтобы ваш код был совместим с более ранними процессорами. Решение может состоять в том, чтобы написать несколько версий вашего кода, каждая из которых будет оптимизированна под определенный процессор. Ваша программа должна определять, на каком процессоре она выполняется и выбирать соответствующую версию кода (глава 27.10).

Оптимизация 32-х битного кода

Автор: Benny, пер. Aquila

Дата: 2002-06-16

Дисклеймер переводчика: Данный текст взят из вирмейкерского журнала 29A#4. Зная негативное отношение многих далеких от программирования людей к подобной литературе, сразу оговорюсь, что тема статьи будет интересна не только создателям вирусов, а вообще любому кодеру, столкнувшемуся с задачей оптимизации своего кода. Кроме того, в статье не содержится ничего противозаконного, поэтому вы можете смело читать ее и использовать приводимые в ней техники. **Дисклеймер:** Этот документ предназанчается только в образовательных целях. Автор не ответственен за возможное применение его неправильным образом. **Предисловие**

Эх, на кой ляд я написал эту статью ? Существует много подобных статей об оптимизации. Да, это правда, и также существует много хороших и кульных туторов [Билли, твой док рулит! *]. Но как вы можете видеть, не каждый автор туториала помнит, что означает термин "оптимизация", многие дают советы только по уменьшению кода. Есть много аспектов оптимизации и я хочу обсудить их здесь и продемонстрировать расширенный взгляд на эту проблему.

Когда я начал писать эту статью, я был очень пьян и находился под воздействием наркотиков (хехе), поэтому если вы чувствуете, что я сделал какую-нибудь ошибку или вы думает, что то, что здесь написано, неправда или просто хотите меня поблагодарить, вы можете найти меня на IRC UnderNet, на каналах #vir и/или #virus или написать по адресу benny@post.cz. Спасибо за все положительные (и также отрицательные) комментарии.

3. Введение

Как я сказал несколько секунд назад, оптимизация имеет несколько аспектов. В общем, мы оптимизируем наш код, чтобы он:

• был меньше
• был быстрее
• был меньше и быстрее

Хорошо, это дает нам новую пищу для размышлений. Если мы оптимизируем наш код:

• код будет меньше, но медленнее
• код будет больше, но быстрее
• код будет меньше и быстрее

Мы должны найти компромисс (если мы не можем сделать так, как в третьем пункте) между первым и вторым пунктом. Я уверен, вы не хотите беспокоить юзера ухудшившимся качеством работы системы из-за:

• большого и неоптимизированного кода
• маленького, но медленного кода

или встревожить пользователя резким уменьшением свободного места на диске.

Вы должны решать, какой путь избарть. У нас есть путеводная нить:

• если наш код (или блок кода, например процедура треда) маленькая, мы должны оптимизировать ее так, чтобы она была более быстрой
• если наш код (или блок кода) большой, мы должны найти компромисс между скоростью и размером

Тем не менее, мы должны оптимизировать наш код уменьшая его размер и повышая его скорость, но вы знаете, как это трудно.

Вы понимаете это? Я уверен, что вы уже знаете об этом. Но все же есть еще много аспектов оптимизации. Возьмем для примера две инструкции, которые делают одно и то же, но:

• одна инструкция больше
• одна инструкция меньше
• одна инструкция меняет другой регистр
• одна инструкция пишет в память
• однак инструкция меняет флаги
• одна инструкция быстрее на одном процесоре, но медленнее на другом

```
Пример:        LODSB             MOV AL, [ESI] + INC ESI
                ----------------------------------------
      размер    меньше             больше
                быстрее на 80386  быстрее на 80486, на Pentium один такт
```

Почему LODSB быстрее на 80386 и почему он занимает только один такт на Pentium? Pentium - это суперскалярный процессор, поддерживающий пайплайнинг, поэтому он может запускать пару двух целочисленных инструкций в пайпе, то есть он может запускать эти инструкции одновременно. Две инструкции, которые могут быть запущены одновременно, называются спаренными инструкциями.

Хе-хе, эта статья не об архитектуре процессора Pentium, поэтому вы можете забыть слова, которые я вам только что сказал. Может быть попозже, если я напишу другую статью об оптимизации Pentium-процессоров, я объясню подробнее, что такое пайпы, V-пайп, U-пайп, спаривание и так далее. Сейчас вы можете забыть об этом. Только помните, что значит слово "спаривание".

Сейчас мы шаг за шагом обсудим каждую из техник оптимизации.

4. Оптимизирование кода Хорошо, давайте оптимизировать. Я начну с самой легкой операции. Начинающие, приготовились... **4.1. Обнуление регистра**

Я больше не хочу видеть этого

```
;1)
mov eax, 00000000h ;5 байт
```

Эта самая худшая инструкция, которую я когда-либо видел. Конечно, кажется логичным, что вы пишеть в регист ноль, но вы можете сделать более оптимизировано так:

```
;2)
sub eax, eax ;2 байта
```

или

```
;3)
xor eax, eax ;2 байта
```

На одной инструкции сэкономлено три байта, прекрасно! X-D Но что лучше использовать, SUB или XOR ? Я предпочитаю XOR, потому что Micro$oft предпочитает SUB, а я знаю, что Windozes - мееедленная система, хе-хе. Неет, это не настоящая причина. Как вы думает, что лучше (для вас) - вычесть два числа или сказать, "где 1 и 1, написать 0) ? Теперь вы знаете, почему я предпочитаю XOR (потому что я ненавижу математику X-D).

4.2. Тест на то, равен ли регистр нулю

Хммм, давайте посмотрим на решение "в лоб":

```
;1)
cmp eax, 00000000h ;5 байтов
je _label_         ;2/6 байтов
                   ;(короткий/ближний)
```

[* ПРИМЕЧАНИЕ: Многие арифметические операции оптимизированы для EAX, поэтому код, использующий этот регистр будет быстрее и меньше. Пример: CMP EAX, 12345678h (5

байт). Если я бы предпочел другой регистр вместо EAX, инструкция CMP была бы равна 6 байтам *]

Аррх! Разве нормальный человек может сделать это ? Это 7 или 15 (!) байт для простого сравнения. Нет, нет, нет, не делайте этого, а попытайтесь так:

```
;2)
or eax, eax ;2 байта
je _label_  ;2/6 (короткий/ближний)
```

или

```
;3)
test eax, eax ;2 байта
je _label_    ;2/6 (короткий/ближний)
```

Хмм, намного лучше, 4/8 байтов гораздо лучше, чем 7/15 байтов. Поэтому снова, что лучше, OR или TEST ? OR предпочитает Micro$oft, поэтому и в этот раз я предпочитаю TEST |-). Теперь серьезно, TEST не пишет в регистр (OR пишет), поэтому он лучше спаривается, а значит, код будет более быстрым. Я надеюсь, вы все еще помните, что значит слово "спаривание"... Если нет, прочтите еще раз секцию "Введение".

Теперь настоящее волшебство. Если вам не важно содержимое регистра ECX или неважно, где будет находится содержимое регистров (EAX и ECX), вы можете сделать так:

```
;4)
xchg eax, ecx ;1 байт
jecxz _label_ ;2 байта
```

[* ПРИМЕЧАНИЕ: XCHG оптимизировано для регистра EAX, поэтому если XCHG будет использовать EAX, он будет на один байт длиннее.]

Прекрасно! Мы оптимизировали наш код и сохранили 4 байта.

4.3. Тест на то, равен ли регистр 0FFFFFFFFh

Многие API возвращаю -1, когда вызов функции проваливается, поэтому важно уметь тестировать это значение. Я всегда бываю поражен, когда вижу, когда кодеры тестируют это значение как я сейчас:

```
;1)
cmp eax, 0ffffffffh ;5 байта
je _label_          ;2/6 байтов
```

Я ненавижу это. А сейчас посмотрим, как это можно оптимизировать:

```
;2)
inc eax    ;1 байт
je _label_ ;2/6 байта
dec eax    ;1 байт
```

Да, да, да, мы сохранили три байта и сделали код быстрее ;).

4.4. Переместить 0FFFFFFFFh в регистр

Некоторые API требуют значение -1 в качестве параметра. Давайте посмотрим, как мы можем сделать это:

Наименее оптимизировано:

```
;1)
mov eax, 0ffffffffh ;5 байт
```

Более оптимизировано:

```
;2)
xor eax, eax ;/ sub eax, eax ;2 байта
```

```
dec eax      ;1 байт
```

Или с таким же результатам (Super/29A):

```
;3)
stc          ;1 байт
sbb eax, eax ;2 байта
```

Этот код очень полезен в некоторых случая, например:

```
jnc _label_
sbb eax, eax ;всего два байта!
_label_: ...
```

4.5. Обнулить регистр и переместить что-нибудь в нижнее слово или байт

Пример неоптимизированного кода: ;1) xor eax, eax ;2 байта mov ax, word ptr [esi+xx] ;4 байта

386+ поддерживает новую инструкцию под названием MOVZX.

```
[* ПРИМЕЧАНИЕ: MOVZX быстрее на 386, на 486+ медленее *]
```

Пример оптимизированного кода, когда мы можем сохранить два байта:

```
;2)
movzx eax, word ptr [esi+xx] ;4 байта
```

Следующий пример "уродливого кода":

```
;3)
xor eax, eax              ;2 байта
mov al, byte ptr [esi+xx] ;3 байта
```

Теперь мы можем сохранить ценный 1 байт X-D:

```
;4)
movzx eax, byte ptr [esi+xx] ;4 байта
```

Это очень эффективно, когда вы хотите читать байты/слова из PE-заголовка. Так как вам нужно работать одновременно с байтами/словами/двойными словами, MOVZX лучше всего подходит в этом случае.

И последний пример:

```
;5) xor eax, eax ;2 байта mov ax, bx ;3 байта
```

Лучше используйте этот вариант, который сэкономит два байта:

```
;6)
movzx eax, bx ;3 байта
```

Я использую MOVZX везде, где только возможно. Он мал и не так медленен как другие инструкции.

4.6. Затолкать дрянь

Скажите мне, как вы сохраните 50h в EAX...

Плохо:

```
;1)
mov eax, 50h ;5 байт
```

Лучше:

```
;2)
push 50h ;2 байта
pop eax  ;1 байт
```

Использование PUSH и POP несколько медленнее, но также и меньше. Когда операнд достаточно мал (1 байт длиной), push занимает 2 байта. В обратном случае - 5 байт.

Давайте попробуем другой случай. Затолкаем семь нулей в стек...

Неоптимизированно:

```
;3)
push 0 ;2 байта
push 0 ;2 байта
push 0 ;2 байта
push 0 ;2 байта
push 0 ;2 байта
push 0 ;2 байта
push 0 ;2 байта
```

Опимизировано, но все равно многовато X-D:

```
;4)
xor eax, eax ;2 байта
push eax     ;1 байт
push eax     ;1 байт
push eax     ;1 байт
push eax     ;1 байт
push eax     ;1 байт
push eax     ;1 байт
push eax     ;1 байт
```

Компактнее, но медленнее:

```
;5)
push 7                ;2 байта
pop ecx               ;1 байт
_label_: push 0       ;2 байта
         loop _label_  ;2 байта
```

Ух ты, без всякого напряга мы сэкономили 7 байт ;)).

А теперь история из жизин... Вы хотите переместить что-нибудь из одной переменной в другую. Все регистры должны быть сохранены. Вы, вероятно, делаете это так:

```
;6)
push eax              ;1 байт
mov eax, [ebp + xxxx] ;6 байтов
mov [ebp + xxxx], eax ;6 байтов
pop eax               ;1 байт
```

А теперь, используя только стек, без регистров:

```
;7)
push dword ptr [ebp + xxxx] ;6 байтов
pop dword ptr [ebp + xxxx]  ;6 байтов
```

Это полезно, когда у вас нет свободных регистров. Я использую это, когда хочу сохранить старую точку входа в другую переменную...

```
;8)
push dword ptr [ebp + header.epoint] ;6 байтов
pop dword ptr [ebp + originalEP]     ;6 байтов
```

Это сохранит два байта. Хотя это немного медленнее, чем нормальные манипуляции с EAX (без его сохранения), все же может случиться, когда вы не хотите (или не можете) использовать какой-либо регистр.

4.7. Забавы с умножением

Скажите мне, как вы вычислите смещение последней секции, когда у вас в EAX number_of_sections-1?

Плохо:

```
;1)
mov ecx, 28h ;5 байт
```

```
mul ecx      ;2 байта
```

Лучше:

```
;2)
push 28h ;2 байта
pop ecx  ;1 байт
mul ecx  ;2 байта
```

Гораздо лучше:

```
;3)
imul eax, eax, 28h ;3 байта
```

Что делает IMUL ? IMUL умножает второй регистр с третьим операндом и сохраняет его в первом регистре (EAX). Поэтому вы можете умножить 28h на EBX и сохранить его в EAX:

```
;4)
imul eax, ebx, 28h
```

Просто и эффективно (как в плане скорости, так и размера). Я не хочу представлять, как вы будете это делать с помощью инструкции MUL... X-D

4.8. Строки в действии

Я хочу перепрыгнуть через стену, когда вижу неоптимизированные операции со строками. Вот несколько подсказок, как вы можете оптимизировать ваш код, используя строковые инструкции. Сделайте это, пожалуйста, или я сделаю это сам ! X-D

Начнем с самого начала, как вы можете загрузить байт ?

Быстрее:

```
;1)
mov al, [esi] ;2 байта
inc esi       ;1 байт
```

Меньше:

```
;2)
lodsb ;1 байт
```

Я рекомендую использовать *меньшую* версию. Это однобайтовая инструкция, которая делает то же самое, что и *быстрая* версия. Это быстрее на 80386, но гораздо медленнее на 80486+. На Pentium, *быстрая* версия требует только один такт из-за спаривания. Тем не менее, я думаю, что лучшим решением будет использовать *меньшую* версию.

И как вы можете загрузить слово ? Грхм, НЕ ЗАГРУЖАЙТЕ слова, это слишком медленно в 32-х битном окружении вроде Win32. Но если вы серьезно настроились сделать это, вот ключ к разгадке...

Быстрее:

```
;3)
mov ax, [esi] ;3 байта
add esi, 2    ;3 байта
```

Меньше:

```
;4)
lodsw ;2 байта
```

Что насчет скорости и размера ? Смотри предыдущее описание (LODSB).

Аааах, загрузка слов тоже веселая штука. Посмотрите на это:

Быстрее:

```
;5)
```

```
mov eax, [esi] ;2 байта
add esi, 4     ;3 байта
```

Меньше:

```
;6)
lodsd ;1 байт
```

Смотри описание LODSW.

А теперь следующая полезность... Перемещаем что-нибудь откуда-нибудь куда-нибудь. Это - LODSB/LODSW/LODSD + STOSB/STOSW/STOSD. Вот пример MOVSD:

Быстрее:

```
;7)
mov eax, [esi] ;2 байта
add esi, 4     ;3 байта
mov [edi], eax ;2 байта
add edi, 4     ;3 байта
```

Меньше:

```
;8)
movsd ;1 байт
```

Быстрее на 486+, *Меньше* всегда ;).

Наконец, я хочу сказать, что вам следует всегда использовать слова вместо байтов или слов, потому что процессор 386+ является 32-х битным. То есть вам процессор работает с 32-мя битами, поэтому если вы хотите работать с одним байтом, он вынужден загрузить двойное слово и обрезать его. Аааа, слишком много работы, поэтому если использование байтов/слов не является необходимым, не используйте их.

Теперь... как вы доберетесь до конца строки ?

Вот метод JQwerty:

```
 ;9)
 lea esi, [ebp + asciiz] ;6 байт
s_check:
 lodsb                   ;1 байт
 test al, al             ;2 байта
 jne s_check             ;2 байта
```

И метод Super'a:

```
;10)
lea edi, [ebp + asciiz] ;6 байтов
xor al, al              ;2 байта
s_check: scasb          ;1 байт
jne s_check             ;2 байта
```

Теперь, какой из них лучший ? Хммм, трудно сказать... X-D На 80386+ будет выполняться быстрее метод Super'a, но на Pentium'e метод Jacky будет быстрее из-за спаривания. Хе-хе, оба способа занимают одинаковое количество места, поэтому выбирайте, какой вы хотите использовать... |-)

4.9. Сложная арифметика

Теперь моя любимая тема. К сожалению, эта прекрасная техника не нашла применения у VX-кодеров. Тем не менее, инструкции, о которых я хочу рассказать хорошо известны (хех, но никто не знает, как их можно использовать), очень малы и очень быстры на любом процессоре.

Вообразите, что у вас есть таблица DWORD'ов. Указатель на таблицу находится в регистре EBX, индекс элемента таблицы находится в ECX. Вы хотите увеличить dword-элемент в таблице, чей индекс содержится в ECX (адрес элемента будет примерно такой: EBX+(4*ECX)). Вы не хотите менять какой-либо регистр.

Вы можете сделать это следующим образом (все так делают):

```
;1)
pushad              ;1 байт
imul ecx, ecx, 4    ;3 байта
add ebx, ecx        ;2 байта
inc dword ptr [ebx] ;2 байта
popad               ;1 байт
```

Или сделайте это лучше (никто так не делает):

```
;2)
inc dword ptr [ebx+4*ecx] ;3 байта
```

Это действительно круто!!! Вы сохранили процессорное время (это очень быстро), место в памяти (очень мало, как вы можете видеть) и сделали более читабельным ваш исходный код!!! Вы сохранили 6 байтов одной простой инструкцией!!!

Это не все (не все об инструкции INC). Вообразите другую ситуацию: EBX - указатель на память, ECX - индекс в таблице, вы хотите повысить следующий элемент в таблице EBX+(4*ECX)+1000h. Да, и вы хотите сохранить все регистры. Вы можете сделать это сделать неоптимизированно:

```
;3)
pushad             ;1 байт
imul ecx, ecx, 4   ;3 байта
add ebx, ecx       ;2 байта
add ebx, 1000h     ;6 байтов
inc dwor ptr [ebx] ;2 байта
popad              ;1 байт
```

Или очень оптимизированно...

```
;4)
inc dword ptr [ebx+4*ecx+1000h] ;7 байтов
```

Яхуууу, мы сохранили 8 байтов одной инструкцией (и это при том, что мы использовали IMUL вместо MUL), великолепно!

Эту магию можно использовать с любой арифметической инструкцией, а не только с INC. Вообразите, как много места вы сможете сохранит, если вы будете использовать эту технику вместе с ADD, SUB, ADC, SBB, INC, DEC, OR, XOR, AND и так далее.

А теперь пришло время для самой великой магии. Эй, парни, скажите мне, что делает инструкция LEA. Вы, вероятно, знаете, что эту инструкцию мы используем для манипуляций с переменными в вирусе. Но только некоторые люди знают, как использовать эту инструкцию действительно эффективно.

Инструкция LEA расшифровывается как Load Effective Address. Это название несколько декларативно. Давайте взглянем, что действительно умеет LEA.

Сделайте следующее:

```
lea eax, [12345678h]
```

Как вы думаете, что будет EAX после выполнения этого опкода ?

Другой пример (EBP = 1):

```
lea eax, [ebp + 12345678h]
```

Что будет в регистре EAX ? Правильный ответ 12345679h. Давайте переведем эту инструкцию на "нормальный" язык:

```
lea eax, [ebp + 12345678h]            ;6 байтов
;=========================
mov eax, 12345678h                    ;5 байтов
add eax, ebp                          ;2 байта
```

Как вы можете видеть, LEA не работает с памятью. Она работает только с ее операндами и делает некоторые операции с ними, затем она сохраняет результат в первый операнд (EAX в нашем примере). Теперь взглянем на размер. Невероятно, она делает то же самое (не совсем так, LEA сохраняет флаги), но это короче. Давайте покажем всю ее магию...

5) Давайте посмотрим на неоптимизированный код:

```
mov eax, 12345678h ;5 байтов
add eax, ebp       ;2 байта
imul ecx, 4        ;3 байта
add eax, ecx       ;2 байта
```

6) Откройте ваш рот и смотрите сюда:

```
lea eax, [ebp+ecx*4+12345678h] ;7 байтов
```

Теперь закройте ваш рот. LEA короче, быстрее (гораздо быстрее) и сохраняет флаги. Давайте взглянем еще раз, мы сохраняем 5 байтов и процессорное время (LEA гораздо быстрее на любом процессоре).

Я не буду объяснять здесь каждую арифметическую инструкцию, я думаю, что это не имеет смысла, потому что у них одинаковый синтаксис. Если вы хотите использовать эту технику, единственная вещь, которую вы должны держать в уме, это синтаксис:

```
OPCODE <SIZE PTR> [BASE + INDEX*SCALE + DISPLACEMENT]
```

4.10. Оптимизация дельта-смещения

Ээээх, вы вероятно думаете, что я сумашедший. Если вы, как читатель этой статьи, не являетесь начинающим, вы должны знать, что такое дельта-смещение. Тем не менее, я видел немало VX-кодеров, использующих дельта-смещения неэффективно. Если вы взглянете на мои первые вирусы, то увидите, что я тоже так делал. И я не одинок. Давайте взглянем более подробно..

Вот как обычно используется дельта-смещение...

```
;1)
call gdelta
gdelta: pop ebp
sub ebp, offset gdelta
```

Это обычный путь (но менее эффективно). Давайте взглянем, как мы можем с этим поработать...

```
lea eax, [ebp + variable]
```

Хммм, если вы взглянете на это под каким-нибудь дебуггером, вы увидите следующую линию:

```
;3)
lea eax, [ebp + 401000h] ;6 байтов
```

В первом поколении вируса, регистр EBP будет обнулен. Ок, но давайте посмотрим, что случится, если вы напишите следующее:

```
;4)
lea eax, [ebp + 10h] ;3 байта
```

Хммм, удивительно. Иногда инструкция занимет 6 байтов, в другой раз 3 байта. Это нормально. Многие инструкции оптимизируются для SHORT-значений, например SUB EBX, 3 будет 3 байта длиной. Если вы напишите SUB EBX, 1234h, инструкция будет длиной в 6 байтов. Не только SUB-инструкция, также многие другие инструкции.

Посмотрим, что произойдет, если мы будем использовать "другой" путь, как получить дельта-смещение...

```
;5)
call gdelta
gdelta: pop ebp
```

Всего-то! Как я и сказал, в первом поколении вируса, EBP будет обнулен (в предыдущей версии gdelta) и переменная будет равна 401000h. Это не очень хорошо. Что вы скажете, если 401000h будет находится в EBP и повышаемое значение и будет той самой переменной. Спасибо нашей новой версии gdelta, мы можем использовать SHORT-версию LEA и тем самым сохраним 3 байта на адресации переменной. Вот пример...

```
;6)
lea eax, [ebp + variable - gdelta] ;3 байта
```

Ок, следующее, что мы должны сделать, это вставить все инициализированные переменные рядом с дельта-смещением. Это действительно важно, иначе переменные будут где-то далеко, поэтому не будет использоваться SHORT-версия LEA. Хех, вы, наверное, думаете, что это какой-то трюк, что есть какие-то ограничеиня или что-нибудь в этом роде, иначе бы все использовали это. Не беспокойтесь, никаких ограничений нет. Но какого черта никто не использует эту технику ? На этот вопрос трудно ответить. Я могу сказать, что не знаю почему. Действительно не знаю.

Мой новый вирус использует подобную обработку дельта-смещения, и я сэкономил огромное количество байтов. Почему бы вам тоже не использовать этот метод ?

4.11. Другие способы оптимизации

Сюда я включил те техники оптимизиации, которые не смог приобщить к одной из вышеперечисленных групп... Просто прочитайте, что-то может оказаться вам полезным...

Обнуление регистра EDX, если EAX меньше, чем 80000000h:

```
;1)
xor edx, edx ;2 байта, но быстрее

;2)
cdq ;1 байт, но медленее
```

Я всегда использую CDQ вместо XOR. Почему ? Почему нет ? X-D

Сэкономим место, используя все регистры, вместо EBP и ESP:

```
;1)
mov eax, [ebp] ;3 байта

;2)
mov eax, [esp] ;3 байта

;3)
mov eax, [ebx] ;2 байта
```

Хотите получить эффект зеркала относительно содержимого регистра?

Попробуйте BSWAP.

Пример:

```
mov eax, 12345678h                    ;5 байтов
bswap eax ;2 байта
; теперь eax = 78563412h
```

Я не нашел какое-либо применение этой инструкции в вирусах. Тем не менее, может быть кому-нибудь она пригодится X-D.

Хотите сэкономить несколько байтов на отказе от CALL ?

```
;1)
call _label_ ;5 байтов
ret          ;1 байт

;2)
jmp _label_ ;2/5 (SHORT/NEAR)
```

Хех, мы сэкономили 4 байта и процессорное время. Всегда замещайте call/ret инструкцией jmp, если при вызове не надо помещать никаких регистров в стек...

Хотите выиграть немного времени, сравнивая содержимое регистра и переменной в памяти?

```
;1)
cmp reg, [mem] ;медленее

;2)
cmp [mem], reg ;на один такт быстрее
```

Хотите сэкономить место и процессорное время во время деления на число, являющееся степенью от двух?

Деление

```
;1)
mov eax, 1000h
mov ecx, 4     ;5 байт
xor edx, edx   ;2 байта
div ecx        ;2 байта

;2)
shr eax, 4 ;3 байта
```

Умножение:

```
;3)
mov ecx, 4 ;5 bytes
mul ecx    ;2 bytes

;4)
shl eax, 4 ;3 bytes
```

Без комментариев...

Циклы, циклы и еще раз циклы:

```
;1)
dec ecx     ;1 байт
jne _label_ ;2/6 байтов (SHORT/NEAR)

;2)
loop _label_ ;2 байта

;3)
je $+5      ;2 байта
dec ecx     ;1 байт
jne _label_ ;2 байта

;4)
loopXX _label_ (XX = E, NE, Z or NZ) ;2 байта
```

LOOP меньше, но медленее на 486+.

И следующая незабываемая вещь. Никто в здравом рассудке не может написать такое:

```
;1)
push eax ;1 байт
push ebx ;1 байт
pop eax  ;1 байт
pop ebx  ;1 байт
```

Делайте так и только так. Ничего, кроме этого:

```
;2)
xchg eax, ebx ;1 байт
```

И снова, если операнд XCHG - EAX, он будет занимать 1 байт, в противном случае - 2 байта. Поэтому когда вы хотите обменять ECX с EDX, XCHG будет 2 байта длиной:

```
;3)
```

```
xchg ecx, edx ;2 bytes
```

Если вы только хотите переместить содержимое одного регистра в другой, используйте простую инструкцию MOV. Она лучше спаривается под Pentium'ом и выполняется меньшее время, если операндом не является EAX:

```
;4)
mov ecx, edx ;2 байта
```

Не используйте повторяющийся код (и код процедур):

```
;1) Unoptimized:

lbl1: mov al, 5                          ;2 байта
      stosb                              ;1 байт
      mov eax, [ebx]                     ;2 байта
      stosb                              ;1 байт
      ret                                ;1 байт
lbl2: mov al, 6                          ;2 байта
      stosb                              ;1 байт
      mov eax, [ebx]                     ;2 байта
      stosb                              ;1 байт
      ret                                ;1 байт
                                         ---------
                                         ;14 байтов

;2) Оптимизированно:
lbl1:   mov al, 5                          ;2 байта
lbl:    stosb                              ;1 байт
        mov eax, [ebx]                     ;2 байта
        stosb                              ;1 байт
        ret                                ;1 байт
lbl2:   mov al, 6                          ;2 байта
        jmp lbl                            ;2 байта
                                           ---------
                                           ;11 байтов
```

Помните, если у вас есть любой излишний код, и это больше, чем инструкция jmp, замещайте ею этот код. Если вы пишете свой собственный полиморфный движок, у вас будет много возможностей сделать это. Не упускайте их !

Манипуляции с переменными:

```
;1) Неоптимизированно:
mov eax, [ebp + variable]  ;6 байтов
...
...
mov [ebp + variable], eax  ;6 байтов
...
...
variable dd      12345678h ;4 байта

;2) Оптимизированно:

mov eax, 12345678h ;5 байтов
variable = dword ptr $ - 4
...
...
mov [ebp + variable], eax ;6 байтов
```

Данная методика очень эффективна в плане экономии места, которое занимает наш код. Как вы можете видеть, мы сохранили 5 байта без всякого напряга или потери стабильности (мы всего лишь делаем недействительным содержимое кэша, поэтому это будет немного, совсем немного медленее).

И, наконец, одна недокументированная инструкция. Мы назвали ее SALC (установить AL при переносе), и она работает на Intel 8086+. Я протестировал ее на моем AMD K5 166MHz, и она тоже работает. SALC делает следующее:

```
;1)
jc _lbl1           ;2 байта
mov al, 0          ;2 байта
jmp _end           ;2 байта
_lbl: mov al, 0ffh ;2 байта
_end: ...

;2)
SALC db 0d6h ;1 байт ;)
```

Это идеально для написания полиморфных движков. Я не думаю, что эвристический эмулятор знает все недокументированные опкоды X-D.

И это все, ребята.

5. И, наконец, несколько типов и трюков

Здесь я дам короткий теоретический обзор наиболее важных оптимизационных техник. Вы должны помнить о них и пытаться использовать, когда используете в вашем собственном вирусе (и не только - прим. перев.).

• Насколько это возможно, избегайте использование стека и переменных. Помните, что регистры гораздо быстрее, чем память (и стек, и переменные в памяти!), поэтому...
• Используйте регистры так часто, как это возможно (используйте MOV вместо PUSH/POP)
• Попытайтесь использовать регистр EAX так часто, как это возможно
• Убирайте все ненужные NOP'ы, повысив число проходов (используйте TASM /m9)
• Не используйте директиву JUMPS
• Для вычисления больших выражений используйте инструкцию LEA
• Используйте инструкции 486/Pentium, чтобы убыстрить код
• Не трахайтесь со своей сестрой !
• Не используйте 16-ти битные регистры и опкоды в вашем 32-х битном коде
• Используйте строковые операции
• Не используйте инструкции, чтобы вычислять значения, которые можно вычислить с помощью препроцессора
• Избегайте CALL'ы, если они не нужны и используйте прямой код
• Используйте 32-х битный DEC/INC вместо 8/16-ти битные DEC/INC/SUB/ADD
• Используйте сопроцессор и недокументированные опкоды
• Держите в уме, что инструкции, у которых нет никаких конфликтов с памятью/регистром могут спариваться, поэтому они будут выполняться минимум в два раз быстрее на процессоре Pentium.
• Если какой-то код используется много раз и занимает больше, чем 6 байт ("call label" и "ret" занимают 6 байт), сделайте ее процедурой и используйте вместо написания повторяющегося кода
• Сокращайте использование условных переходов к минимуму, их предсказание появилось начиная с P6+. Слишком много условных переходов может затормозить ваш код в x-раз. Безусловные переходы - это ОК, но, тем не менее, каждый байт можно соптимизировать |-)
• Для арифметических вычислений и последующих операций используйте арифметический расширения инструкций
• Уффф, я больше не знаю, что вам посоветовать. Мммм, прочитайте это снова X-D

И это все, ребята. Давайт встретимся где-нибудь в следующих жизнях...

6. В заключение

Уффф, хорошо, если вы дочитали эту длинную статью. Что я хочу сказать ? Я надеюсь, что вы поняли все, что я изложил (или хотя бы 50 %), и будете использовать это в своем коде. Я знаю, я не один из тех ребят, которые оптимизируют все 100% своего кода. Тем не менее, я пытаюсь это сделать. В основном, я думаю, эта оптимизация кода можно проводить после того, как сделано все остальное. Эта одна из тех вещей, которая делает вас профессиональным кодером. Кодер, который не оптимизирует его собственный код - это не профессиональный кодер. Запомните это.

Хе-хе, и снова моя любимая тема - если вам нравится этот туториал, я буду очень благодарен вам, если вы напишите мне что-нибудь (benny@post.cz). Большое, большое спасибо.

Благодарности: Darkman/29A, Super/29A, Jacky Qwerty/29A, GriYo/29A, VirusBust/29A, MDriler/29A, Billy_Bel/???, MrSandman и всем, кого я забыл...

7. Компиляторы

Как создать invoke'абельную библиотеку импорта

Автор: Iczelion, пер. Aquila

Дата: 2002-06-27

Это короткий очерк о том, как создать библиотеку импорта, которую можно использовать вместе с MASM'ом. Я предполагаю, что вы уже знаете кое-что о библиотеках импорта, то есть вы знаете, что это так далее и так далее. Я сделаю упор на то, как генерировать библиотеки импорта, совместимые с MASM'ом.

Формат библиотек импорта MASM'а

MASM и Visual C++ могут использовать одинаковые библиотеки импорта, что очень удобно. Микрософтовские библиотеки импорта используют разновидность формата COFF, которое отлично от формата OMF, используемого TASM'ом. По этой причине TASM не может использовать MASM'овские библиотеки импорта и наоборот. Я не буду углубляться в детали строения этих библиотек. Достаточно сказать, что каждая библиотека импорта Microsoft'а содержит информацию о функциях из определенных DLL. Эта информация включает в себя имена функций и общий размер параметров, передаваемых функциям. Если вы пробежитесь по kernel32.lib с помощью hex-редактора, вы найдете найдете в ней следующее:

```
_ExitProcess@4
_CreateProcessA@40
```

Имена функций имеют префикс '-'. Число, следующее за @ - это общий размер параметров этой функции в байтах. ExitProcess принимает только один параметр dword, поэтому это число равно 4. Почему включается информация о размере параметров? Эта информация используется MASM'ом, чтобы проверить правильность переданных функции параметров, когда та вызывается с помощью ключевого слова 'invoke'. Если вы просто затолкаете параметры в стек инструкцией 'push' и запустите функцию инструкцией 'call', MASM не будет проверять правильность параметров. Это преимущество делает невозможным создать библиотеки импорта MASM из DLL, потому что DLL не содержит точной информации о размере паарметров, передаваемых функции.

Создание библиотек импорта MASM из DLL

Если вы готовы использовать функции с помощью 'push' и 'call', вы можете создать библиотеку импорта из любой DLL следующим образом.

• Используйте dumpbin.exe, которая поставляется вместе с Visual C++, чтобы получить имена экспортируемой DLL функций.

```
Dumpbin /EXPORTS  blah.dll  >  output.txt
```

• После того, как вы получили список функций, создайте модуль определения файла с его помощью. Например, если DLL содержит только одну функцию, GetSomeLine, напечатайте следующее:

```
LIBRARY blah
EXPORTS
GetSomeLine
```

• И сохраните как blah.def.
• Запустите lib.exe, чтобы создать библиотеку импорта из модуля определения файла:

```
lib /DEF:blah.def
```

Вот и все. Вы получили blah.lib, который можете использовать вместе с MASM, пока вам не требуется использовать 'invoke'.

Создание invoke'абельных библиотек импорта

Я один из тех, кто очень неохотно использует вышеприведенный подход. Использовать invoke гораздо более удобно. Это одна из причин, по которой я предпочитаю MASM TASM'у. Но как было сказано ранее, практически невозможно создать invoke'абельную библиотеку импорта с помощью той процедуры, что была изложена выше. Например, вы можете решить, что если вы измените имена функций в .def-файле, чтобы туда входило "@xx", библиотека импорта может заработать как надо. Поверьте мне. Это не будет работать.

Более легкий путь создать invoke'абельную бибиотеку импорта - это использовать сам MASM. Если вы создатите DLL, то вы обнаружите, что вместе с ней получили библиотеку импорта, которая будет полностью invoke'абельна! Наша стратегия заключается в следующем:

• Получаем информацию об именах функций и общем размере параметров.
• Создаем исходный код DLL, которая будет включать в себя эти функции с правильным числом и размером аргументов.
• Создаем файл определения модуля, в котором экспортируем соответствующие функции.
• Ассемблируем исходный asm-код как DLL-проект.

Вот и все. Вы получите полностью функциональную MASM'овскую библиотеку импорта. Вышеприведенные шаги заслуживают более подробного объяснения.

Получение имен функций и общего размера параметров

Это наиболее сложная часть процесса. Если у вас есть только DLL, вам предстоит утомительное приключение. Ниже изложены несколько методов, которые вы можете использовать, хотя ни один из них не дает 100% гарантию.

• Используйте Interactive Disassembler (IDA), чтобы дизассемблировать DLL. С помощью этого чудесного инструмента вы можете получить полный размер параметров, принимаемых функцией. Однако это не совершенный способ. IDA - потрясающий дизассемблер, но иногда только человек может решить что есть что. Вам надо будет подумать и проработать весь листинг.
• Следите за значением указателя на стек до и после вызова всех функций в DLL. Метод состоит в следующем:
• Получить адрес функций с помощью GetProcAddress.
• Вызвать каждую функцию не передавая ей никаких параметров через стек. Запомнить значение esp до вызова.
• Когда функция возвратит управление, сравнить значение esp после вызова с тем, что было перед вызовом. Логическое обоснование здесь следующее: при передаче параметров в формате stdcall, функция берет на себя ответственность соблюдения секового баланса. Разность значений esp и будет размером параметров, ожидаемых функцией.
• Увы, этот метод не безупречен. Он может не удастся в следующих обстоятельствах.
• Если функции в DLL используют другое соглашение передачи параметров, отличное от stdcall или pascal.
• Если функции не удается очистить стек, например при возникновении исключения.
• Если интересующие нас функции служат для чего-нибудь опасного, например для форматирвания винта (Упаси Господь!)
• Изучите существующие программы, которые используют нужную DLL. Вы можете отладить/дизассемблировать эти программы, чтобы увидеть количество и размер параметров, передаваемых фунциям в DLL. Тем не менее, если в DLL есть функции, которые не используются ни в одной из доступной вам программ, этот метод не будет работать.

Создание исходника DLL, который будет содержать все эти функции

После того, как вы получите имена функций и размер их параметров, самое трудное будет позади. Вам останется создать каркас DLL и написать функции с такими же именами, как и в DLL. Например, в DLL только одна функция, GetSomeLine, которая получает параметров на 16 байт. В исходнике вы набиваете следующие линии:

```
.386
.model flat,stdcall
.code
GetSomeLine proc param1:DWORD, param2:DWORD, param3:DWORD, param4:DWORD
GetSomeline endp
end
```

Вы можете спросить, что это такое? Процедура, в котрой нет ни одной инструкции? Библиотека импорта не содержит никакой информации о том, что должна делать функция. Единственной ее целью является предоставление информации об именах функций и их параметров. Поэтому нам не нужно помещать никаких инструкций в процедуру-болванку. Все равно мы сотрем бесполезную DLL после сборки. Все, что мы хотим - это помесить в исходный код информацию об именах функций и размере параметров, чтобы MASM сгенерировал рабочую библиотеку импорта. Размер каждого параметра по отдельности не важен. Для вашего сведения, в настоящее время MASM всегда рассматривает каждый параметр как DWORD, какой бы модификатор размера вы не поставили. Например, мы можем сделать так:

```
.386
.model flat,stdcall
.code
GetSomeLine proc param1:BYTE, param2:BYTE, param3:BYTE, param4:BYTE
GetSomeline endp
end
```

И MASM создаст в библиотеке импорта '_GetSomeLine@16'.

Создание файла определения модуля

Это простой процесс. Вам потребуется этот файл, что MASM мог сгенерировать DLL и библиотеку импорта. Шаблон файла определения модуля следующий:

```
LIBRARY  <The name of the DLL>
EXPORTS
<The names of the functions>
```

Вам остается указать имя DLL, которое будет так же и именем библиотеки импорта, а затем вставить имена функций после команды EXPORTS, по одному имени функции на каждой линии. Сохраните файла и вы получите рабочий файл определения модуля.

Ассемблирование исходного кода как DLL-проекта

Последний шаг - самый простой. Вам потребуются ml.exe и link.exe.

```
ml /c /coff /Cp blah.asm
link /DLL /NOENTRY /def:blah.def /subsystem:windows blah.obj
```

И вы получите invoke'абельную библиотеку импорта.

Пример

Сухое изложение выше может быть не до конца понятным. Я верю в обучение через действие. Поэтому я сделал пример (находится в архиве `wasm_files.zip`: `files/recovered/1018001/implibexample.zip`), который демонстрирует вышеописанное. Файлы, входящие в пример, следующие:

• Исходный код на ассемблере, который содержит все функции в kernel32.dll (задокументированные).
• Файл определения модуля.
• Батник, который вы можете использовать для сборки библиотеки импорта.

Скомпилировав пример, вы получите kernel32.lib, который вы можете использовать вместо того, который предоставил Microsoft.

Дополнительные инструменты Если вы хотите добавить/убрать функции из/в определенную библиотеку импорта, вы можете использовать две простые утилиты, написанные мной. Например, если вы хотите добавить в kernel32.lib недокументированные функции, эти программы окажутся вам полезными. **Lib2Def**

Она извлекает имена и соответсвующие размеры из любой библиотеки импорта. Запустите ее и она обработает все библиотеки, находящиеся в той же директории. У выходных файлов будет расширение .icz. Их содержимое будет выглядеть примерно так:

```
_ExitProcess@4
_ExitThread@4
_ExitVDM@8
_ExpandEnvironmentStringsA@12
_ExpandEnvironmentStringsW@12   @12
```

Если вы хотите добавить функцию, вам всего лишь нужно вставить новое имя (прибавив к нему префикс '_') и суммарный размер параметров. Если функция экспортируется по ординалу, то за именем надо поставить @xx. "xx" будет ординалом. Обратите внимание, что эта простая утилита не проверяет имена на повторение, потому что в некоторых библиотека импорта имена могут повторяться.

MLib

Эта утилита принимает файлы, генерируемые Lib2Def и создает из них библиотеку импорта. Она обработает все файлы с расширением .icz. К вашему сведению, она парсит линии .icz-файла и создает из них .asm и .def. Затем она вызывает ml.exe и link.exe, чтобы те сгенерировали библиотеку импорта. Файлы .obj, .asm, .exp и .dll удаляются и остается только .lib. Если этой утилите не удается сгенерировать .lib-файл, пожалуйста проверьте, нет ли повторяющихся линий в .icz-файле: это самый частый случай.

Скачайте (находится в архиве `wasm_files.zip`: `files/recovered/1018001/implibtools.zip`) эти две утилиты.

Макросы для компиляции команд

Автор: pts / HI-TECH

Дата: 2002-10-20

За последние несколько лет сменилось три поколения микропроцессоров семейства Pentium. У процессоров каждого поколения и у разных моделей одного поколения появлялось достаточно много новых инструкций. Например, по сравнению с Pentium III, у Pentium 4 прибавилось сразу 144 инструкции!

Компания Intel заблаговременно информирует разработчиков программного обеспечения о своих планах, поэтому Microsoft достаточно оперативно реагирует на технические нововведения и выпускает новые версии компиляторов. Если по каким-то причинам последняя версия MASM вам недоступна, то возможности устаревших версий можно расширить с помощью макроопределений новых команд.

Такие макроопределения преобразуют исходную мнемоническую запись команды на языке ассемблера в объектный код. При этом они выполняют синтаксический разбор записи команды, сложность которого зависит от используемых способов адресации операндов. Мы начнем с рассмотрения простых случаев, а затем перейдем к сложным.

Команда без операндов. Во многих руководствах встречается макроопределение для компиляции инструкции CPUID, которую выполняют все модели микропроцессоров семейства Pentium. Ее могут компилировать только версии MASM, исполняющие директиву .586, в противном случае используется макрос, приведенный в примере 8.

Пример 8. Компиляция инструкции cpuid

```
CPUID MACRO        ; заголовок макроопределения
 db 0Fh, 0A2h ; объектный код инструкции cpuid
 ENDM  ; конец макроопределения
```

Макровызовом этого определения является запись в исходном тексте программы инструкции cpuid, вместо нее в текст программы будет включена директива db 0Fh, 0A2h.

Инструкция имеет операнды. Подходящих для наших целей одноадресных команд нет, поэтому выберем такую группу двухадресных команд, операнды которых адресуются наиболее просто.

У микропроцессора Pentium Pro появились две группы команд условной пересылки **CMOVcc** и **FCMOVcc**. Первая группа относится к категории команд общего назначения. Инструкции второй группы исполняет процессор FPU, они применяются при программировании вычислений с плавающей точкой. В записи конкретной команды вместо маленьких букв cc подставляется мнемоническое обозначение условия, например, CMOVNE или FCMOVB.

Компилировать инструкции этих групп могут только версии MASM, начиная с 6.12 (исполняющие директиву .686), в противном случае надо применять специальные макроопределения. Мы опишем макроопределения для компиляции инструкций группы FCMOVcc. Операнды инструкций этой группы могут находиться только в числовых регистрах, что упрощает анализ и преобразование их имен.

В состав группы входят инструкции, имеющие идентичную структуру кода и компилирующиеся по одной схеме. Это позволяет вместо индивидуальных макроопределений для каждой инструкции использовать специальное **мета-макро определение**. В отличие от обычного, оно порождает при вызовах макроопределения для компиляции конкретных инструкций, имена которых указываются в специальных списках. Поэтому вместо составления группы идентичных макроопределений составляется список допустимых вариантов вызова мета-макроса, содержащий столько строк,

сколько инструкций входит в состав группы.

В примере 9 приведены текст мета-макро определения **FPIDEF** и список его вызовов, содержащий имена восьми инструкций группы FCMOVcc с соответствующими им кодами операций. При вызове FPIDEF **формируется и исполняется** одно из 8-ми обычных макроопределений, **имена которых** совпадают с именем компилируемой инструкции, а параметрами являются ее (инструкции) операнды.

Пример 9. Мета-макро определения инструкций FCMOVcc

```
FPIDEF MACRO OPNAME, OPCODE1, OPCODE2 ; мета-макроопределение
  OPNAME MACRO OP1, OP2 ; заготовка основного определения
; Проверка имени первого операнда
  LOCAL fvsym, reg   ; описание локальных переменных
  IFDIFI <OP1>, <ST> ; сравнение операндов
   IFDIFI   <OP1>, <ST(0)> ; сравнение операндов
   .ERR1 %OUT 'Error: First operand must be st'
   ENDIF ; конец тела второй директивы IFDIFI
  ENDIF ; конец тела первой директивы IFDIFI
;  Проверка имени второго операнда
  reg SUBSTR <OP2>, 4, 1  ; выделение номера регистра
fvsym SUBSTR <st(0)st(1)st(2)st(3)st(4)st(5)st(6)st(7)>,1+reg*5,5 ; выбор эталона
  IFDIFI fvsym, <op2> ; сравнение OP2 с эталоном имени
   .ERR2
   %OUT 'Error: Second operand must be st(i)'
  ENDIF ; конец тела директивы IFDIFI
; Запись кода команды в объектный модуль
  db opcode1, opcode2+reg
  ENDM ; конец макроопределения opname
ENDM   ; конец макроопределения fpidef
; Список вызовов макроопределения FPIDEF
FPIDEF FCMOVB, 0DAh, 0C0h ; пересылка если меньше
FPIDEF FCMOVE, 0DAh, 0C8h ; пересылка если равно
FPIDEF FCMOVBE, 0DAh, 0D0h ; пересылка если меньше или равно
FPIDEF FCMOVU, 0DAh, 0D8h ; пересылка если неупорядочены
FPIDEF FCMOVNB, 0DBh, 0C0h ; пересылка если больше или равно
FPIDEF FCMOVNE, 0DBh, 0C8h ; пересылка если не равно
FPIDEF FCMOVNBE,0DBh, 0D0h ; пересылка если больше
FPIDEF FCMOVNU, 0DBh, 0D8h ; пересылка если упорядочены
```

Предположим, что в тексте программы встретилась команда fcmove st, st(4). MASM ищет любые имена в своих списках зарезервированных символов и в таблицах имен, описанных в компилируемой программе. Обнаружив имя fcmove в списке, приведенном в конце текста примера 9, он формирует макровызов "fpidef fcmove, 0DAh, 0C8h". Макрос fpidef, в свою очередь, вместо OPNAME формирует и исполняет макроопределение "fcmove st, st(4)", которое компилирует команду и вставляет в текст программы директиву db 0DAh, 0CCh.

Макроопределение OPNAME проверяет имена операндов, формирует код второго операнда в переменной **reg** и включает в текст программы указанную выше директиву db, описывающую код команды.

Имя первого операнда проверяется сравнением с его допустимыми образцами st или st(0). Для того чтобы результат не зависел от регистра, на котором набраны буквы имени (st, ST, sT, St), сравнение выполняет директива **IFDIFI** (если различаются). Буква **I** в конце ее имени указывает на то, что перед сравнением операндов коды всех букв имен приводятся к одному регистру.

Для проверки второго операнда в переменную reg с помощью директивы SUBSTR помещается четвертый символ его имени. Если имя указано правильно, то это будет цифра от 0 до 7. Затем вторая директива SUBSTR копирует в переменную **fvsym** образец правильной записи имени второго операнда из списка допустимых имен. Наконец, директива IFDIFI сравнивает содержимое fvsym с именем второго операнда. При несовпадении выдается сообщение об ошибке и компиляция прекращается. В случае совпадения в текст программы вставляется код команды.

Дополнение к примеру 9. При выполнении инструкций группы FCMOVcc проверяется состояние разрядов регистра флагов EFLSGS. Обычным инструкциям FPU регистр флагов недоступен. Поэтому разработчики Pentium Pro ввели 4 новые операции, которые помещают результат сравнения операндов в разряды EFLAGS. Эти четыре инструкции имеют тот же формат, что и инструкции группы FCMOVcc. Поэтому для их компиляции можно использовать макроопределение примера 9 - просто добавьте в конец его текста следующие пять строчек:

```
; Дополнение списка вызовов макроопределения FPIDEF
FPIDEF FCOMI, 0DBh, 0F0h ; простое сравнение операндов
FPIDEF FCOMIP, 0DFh, 0F0h ; тоже, но с освобождением st
FPIDEF FUCOMI, 0DBh, 0E8h ; сравнение неупорядоченных операндов
FPIDEF FUCOMIP, 0DFh, 0E8h ; тоже, но с освобождением st.
```

Напомним, что операнды неупорядочены, если значения одного или обоих лежат за пределами допустимого диапазона чисел. Обычно при выполнении операций над неупорядоченными операндами возникает аварийная ситуация при сравнении этого не происходит, а вырабатывается соответствующий признак.

Операнд в оперативной памяти. У новых двухадресных инструкций групп MMX, SSE1 и SSE2 один операнд находится в регистре mmx или xmm, а другой либо в регистре, либо в оперативной памяти. В большинстве случаев в оперативной памяти находится второй операнд, исключением являются инструкции пересылки. Они позволяют перемещать данные как из памяти в регистры, так и в обратном направлении.

Составить макроопределения для анализа и преобразования имен новых регистров mmx и xmm несложно. Намного сложнее выполнять анализ и преобразование адресов операндов, находящихся в оперативной памяти. При работе с 32-х разрядными адресами код операнда содержит переменное количество байтов, а в его мнемонической записи допускается использование арифметических и логических выражений. В качестве примера приведем три вполне корректные формы записи адресов операндов:

```
dword ptr [ebx + 8]
[eax + 4*ecx + 32]
16[ebx][eax*4];
```

Очевидно, что для анализа и преобразования подобных выражений нужен разбор всех допустимых случаев, при этом текст макроопределения становится слишком большим и трудно обозримым. Пример макроса, обрабатывающего ограниченный набор способов записи адресов операндов, приведен в Интернет на моем сайте www.macro.aaanet.ru. Но и при разумных ограничениях текст макроса остается достаточно длинным, поэтому некоторые авторы выбирают другой путь. К их числу относится Агнер Фог. Разработанные им макросы для компиляции новых инструкций находятся в Интернет, на сайте www.agner.org.

Подмена компилируемой инструкции. В этом случае в макроопределении выполняется только анализ и преобразование имен новых регистров, а адреса операндов, находящихся в оперативной памяти компилирует MASM. Для того чтобы он мог это сделать, в тексте макроопределения формируется псевдокоманда. Она состоит из имени подходящей инструкции общего назначения и операндов, указанных в реальной команде. После компиляции псевдокоманды в полученном объектном коде изменяется код операции. В качестве примера мы рассмотрим такой трюк (иначе его никак не назовешь) в чистом виде, когда в командах отсутствуют имена новых регистров.

Как уже говорилось, начиная с Pentium Pro, микропроцессоры семейства Pentium поддерживают группу CMOVcc, состоящую из 16-ти инструкций условной пересылки. При работе с версиями MASM 6.0 - 6.11 для компиляции инструкций этой группы нужно специальное макроопределение. Его текст приведен в примере 10.

Пример 10. Макроопределение для компиляции инструкций CMOVcc

```
CMOVDEF  MACRO OPNAME, CCODE ; мета-макро определение
```

```
 OPNAME MACRO DST, SRC:VARARG ; заготовка макроопределения
LOCAL X, Y ; описание локальных переменных (меток)
X:  ; сохранение адреса начала команды
BSF  DST, SRC ; компиляция псевдокоманды
Y:  ; сохранение адреса конца команды
ORG X+1 ; возврат ко второму байту кода операции
DB   CCODE ; изменение кода второго байта команды
ORG Y ; восстановление текущего адреса
ENDM ; конец макроопределения OPNAME
ENDM  ; конец макроопределения CMOVDEF
    ; Начало списка инструкций группы вызовов CMPVDEF
CMOVDEF CMOVO, 40h ; пересылка если переполнение
CMOVDEF CMOVNO, 41h ; пересылка если нет переполнения
CMOVDEF CMOVB, 42h ; пересылка если меньше
CMOVDEF CMOVNB, 43h ; пересылка если больше или равно
    ; и так далее вплоть до инструкции CMOVG с кодом 4Fh
```

Текст примера 6.10 состоит из мета-макро определения CMOVDEF и неполного списка его макровызовов с указанием имен 4-х первых инструкций группы CMOVcc и соответствующих им кодов операций.

Предположим, что в тексте программы встретилась команда "cmovb cx, [bx+2]". MASM находит имя инструкции в списке CMOVDEF и вызывает одноименное мета-макро определение с параметрами CMOVB, 42h. Оно, в свою очередь, формирует и выполняет макроопределение "cmovb cx, [bx+2]", в котором производится подмена имени компилируемой инструкции.

Для подмены взята инструкция BSF, ее исполняют микропроцессоры семейства Intel, начиная с модели 386, и могут компилировать все версии MASM, начиная с 5.1. Таким образом, реально компилируется команда "bsf cx, [bx+2]". В результате MASM получается объектный код "0F BC 4F 02". Остается заменить в этом коде содержимое второго байта (BC) на 42, т. е. сформировать код "0F 42 4F 02", соответствующий исходной команде "cmovb cx, [bx+2]"

Особенности работы с макроопределением. Как уже говорилось, подмена компилируемой инструкции это трюк, а применение любых трюков обычно связано с определенными ограничениями. Обсудим два основных ограничения.

1. При корректировке кода инструкции BSF в примере 10 изменяется содержимое второго байта, но в некоторых случаях, нужный байт может быть не вторым, а третьим или четвертым. Давайте разберемся, в каких случаях это происходит.

Начиная с Intel 386, микропроцессоры могут работать в 16-ти разрядном (реальном) и в 32-х разрядном (защищенном) режимах. Независимо от установленного режима у команд могут быть как 16-ти, так и 32-х разрядные операнды и адреса. Если разрядность операнда или адреса не совпадает с разрядностью установленного режима работы процессора, то у команды появляются один или два префикса. Префикс размера операнда имеет код 66h, а префикс размера адреса имеет код 67h.

При наличии префиксов текущий адрес, сохраненный в примере 10 в переменной X, не соответствует адресу начала кода операции, а содержимое адреса X+1 не является вторым байтом кода операции и его изменять нельзя. Как поступать в таких случаях?

MASM определяет разрядность режима работы микропроцессора исходя из модели памяти, указанной в директиве **.MODEL**. Допустимы следующие имена моделей: TINY, SMALL, COMPACT, MEDIUM, LARGE, HUGE, FLAT. Из них только модель FLAT соответствует 32-х разрядному режиму работы микропроцессора.

Если компилируемые инструкции работают **только** с 16-ти разрядными операндами и адресами, то вы можете выбрать любую модель, кроме FLAT. И наоборот, если инструкции работают **только** с 32-х разрядными операндами и адресами, то надо выбрать модель FLAT. Только при выполнении этих условий у команд отсутствуют префиксы, и макроопределение примера 10 является

корректным.

Если по каким-то причинам указанное требование не выполнимо, то надо произвести предварительную компиляцию и получить листинг программы. По листингу уточняется наличие и количество префиксов перед командой BSF и вносится изменение в текст макроопределения примера 10. В зависимости от количества префиксов адрес в директиве ORG X+1 изменяется на X+2 или X+3. Ничего лучшего предложить нельзя.

2. Запись адреса операнда, находящегося в оперативной памяти, может состоять из нескольких слов или выражений, разделенных пробелами. При компиляции таких записей с применением макроопределения примера 10 будет выдано аварийное сообщение "слишком много операндов". Для того чтобы в подобных случаях не возникало аварийной ситуации, запись адреса в команде надо заключить в угловые скобки, например, <dword ptr [eax]>. Сказанное не распространяется на пробелы, используемые при записи индексных выражений, заключенных в квадратные скобки.

Следует подчеркнуть, что указанные особенности или недостатки относятся не только к примеру 10, но и ко всем макроопределениям, в которых применяется подмена компилируемой инструкции.

Условное ассемблирование. Одним из полезных приемов программирования является включение в исходный текст программы условных блоков. Так называют последовательность команд, директив или макроопределений, которые компилируются или не компилируются в зависимости от выполнения заданного условия. Мы опишем пример условного блока, но предварительно поясним его назначение.

Макроопределения, выполняющие компиляцию любых инструкций можно применять только в тех случаях, когда используемая версия MASM не делает это самостоятельно. В противном случае MASM выдаст аварийное сообщение о повторном определении имени известной ему инструкции.

Программист обычно знает номер версии MASM, с которой он работает, и самостоятельно решает вопрос о необходимости применения соответствующих макроопределений. Но если создается общедоступное макроопределение, то заранее неизвестно кто им воспользуется, и какая версия компилятора окажется в его распоряжении. В таком случае текст макроопределения целесообразно оформить в виде условного блока, перед компиляцией которого проверяется поддержка MASM конкретной группы директив. Текст начала такого блока приведен в примере 11.

```
Пример 11. Управление компиляцией макроопределения
 IFDEF @version ; если переменная @version определена, то
  IF @version GE 612 ; если версия MASM 6.12 или выше, то
  ; MASM поддерживает инструкции группы P6
  .686  ; разрешение компиляции инструкций группы P6
  SUPPORT EQU 1 ; определение переменной SUPPORT
  ENDIF  ; конец действия директивы IF
 ENDIF  ; конец действия директивы IFDEF
 IFNDEF SUPPORT  ; если переменная SUPPORT не определена, то
 ; здесь располагается текст макроопределения для компиляции новых
 ; инструкций из группы P6, например, полный текст примера 9 или 10.
 ENDIF  ; конец действия директивы IFNDEF SUPPORT
```

В примере 6.11 проверяется поддержка MASM инструкций общего назначения, входящих в группу P6. Если вы собираетесь компилировать новые инструкции MMX, то, не изменяя версию, замените в тексте примера 6.11 директиву **.686** на **.mmx**. При компиляции инструкций групп SSE надо изменить номер версии на 614, и вместо директивы **.686** указать **.xmm**.

Замечание. Директивы .686, .mmx и .xmm разрешают компиляцию разных категорий инструкций микропроцессоров, поэтому их можно использовать совместно, но директива .686 должна быть указана первой. По крайней мере, так надо поступать при работе с версией MASM 6.15.

Заключение. В пределах статьи невозможно описать все разнообразие макросов и все возможные случаи их применения. Поэтому я стремился показать некоторые общие приемы, которые могут быть полезны при составлении макросов различного назначения. Насколько удачной оказалась эта

попытка судить вам, уважаемые читатели и программисты.

Сам себе компилятор (или руководство по мазохизму для дзенствующих)

Автор: FatMoon / HI-TECH

Дата: 2002-11-21

Предисловие

> *FatMoon, ты извращенец! Знаешь об этом?*
> **_ Serrgio _**
> *Да, об этом я знаю. Подозревать начал еще в раннем детстве, когда*

• пробовал на вкус акварельные "медовые" краски;
• пытался залить чернила в стержень авторучки;
• приносил домой гусениц, кормил их листьями и вылуплял бабочек.

Когда я сел за компьютер, подозрения перешли в твердую уверенность. Да, похоже, что я извращенец. Хотя все относительно - видал я таких извращенцев, по сравнению с которыми я просто … пуританин! А писать на ассемблере под WINDOWS не извращение? Молчите? И в этом молчании я слышу глас рассудка. ї А. Сапковский.

О чем эта статья? О программировании на ассемблере без компилятора, об отладчике DEBUG из стандартного пакета DOS/WINDOWS, о машинном коде и прочим смежным темам.

_ Alex FatMoon _

Как обычно пишутся программы на ассемблере? В редакторе - не суть важно, каком - набирается исходный текст, содержащий директивы ассемблера, мнемокоды инструкций, определения данных и метки. Затем вызывается компилятор - не суть важно, какой - который транслирует исходный файл в объектный. Затем линкер собирает исполняемый файл из одного или нескольких объектных модулей. Знакомая картина, не правда ли? Однако необходимо ли все это для создания работоспособной программы? Нет, конечно же. Для создания .com - файлов вполне достаточно отладчика DEBUG. Вот об этом и будет речь.

Почти любой отладчик имеет функцию транслирования мнемокодов в машинный код с возможностью записи на диск набранного блока. Поэтому в принципе эти же приемы могут быть использованы и с другими утилитами. Почему все-таки я описываю DEBUG? Он есть практически на любой машине, где установлена любая из версий MS-DOS или WINDOWS. Он прост в использовании, не требователен к ресурсам, знает почти все машинные коды и даже поддерживает инструкции сопроцессора. Он - первый друг взломщика сейвов к играм и просто хакера. И вообще - одно из лучших творений МикроСофт. Лежит он, родимый, в директории *C:\DOS* или *C:\WINDOWS\COMMAND* - даже если ваши папки называются по-другому и расположены на других дисках, найти его нетрудно. Переходим к непосредственному использованию.

Программирование без компилятора

> *Русские сначала придумывают себе препятствия, а потом их преодолевают.*
> **_ (известный факт) _**

> *Входите узкими вратами, ибо широки врата и просторен путь, ведущий в геену огненную.*
> **_ (Евангелие от Матфея) _**

Да, если у вас есть компилятор, использовать что-то еще нет нужды. Но представим себе, что компилятора нет. Или мы только начали изучать ассемблер и в принципе знаем мнемокоды по богатому опыту программирования на БК, Спектруме или Микроше, но о директивах имеем только общее представление. И напряженно думаем, с какими ключами запускать этот **tasm**? И почему **tasm** выдает ошибки при компиляции, хотя все написано правильно (кто бы мог подумать, что *ss: mov ax, [si]* на самом деле должно быть записано как *mov ax, ss:[si]?*). Тогда это для вас. А также если есть желание понять, как же работает компилятор и зачем он все-таки нужен. Подумав (от "DOOM") и перекрестясь (наведя перекрестье на последнего из монстров, нажав *"fire primary"* и выйдя из игры), приступим!

Пример программы.

Главная команда отладчика, которую будем использовать, **"A"** ssembly. Итак, входим в отладчик, набираем

```
a 100
```

и видим адрес xxx:0100. От нас ждут ввода инструкций. Собственно, можно набирать программу. Пример:

```
a 100
   mov ah,09
   mov dx,0100
   int 21
   ret
   db "Hello, world!$"
```

и готово. Однако это еще не конец. Надо скорректировать dx - 0100 туда засылается просто для того, чтобы обозначить команду. Ведь когда мы пишем:

```
mov dx,xxx
```

мы еще не знаем адреса, с которого начнется строка "Hello, world!$", не так ли? Этот адрес мы узнаем только после того, как введем "ret". Итак, запоминаем в cx длину нашей программы - она равна адресу инструкции после строки, где мы нажали "Enter", чтобы выйти из режима компиляции, за вычетом 0100 - адрес, с которого начинается программа типа "COM". У меня для вышеприведенной программы получилось 016h. Значит, набираем:

```
rcx
   16
```

А теперь корректируем адрес - строка начинается с 0108. Набираем:

```
a 102
   mov dx,108
```

Осталось сохранить программу на диске:

```
n hello.com
   w
```

Вуаля! Теперь в директории, где мы находились перед вызовом отладчика, есть файл **hello.com** длиной в 22 байта, исправно печатающий при запуске строку "Hello, world!" и возвращающий управление ОС. Просто, не так ли? И никаких компиляторов. На самом деле в роли такового выступили мы сами - назначили строке фиктивный адрес (0100), затем заменили его на точный (0108). Вся помощь отладчика заключалась в переводе мнемокодов в машинные. Не можем же мы помнить **все** машинные коды!

Это кажется элементарным, не спорю. Попробуем сделать что-нибудь более сложное, и затем попытаемся оптимизировать это. То есть, обычная процедура для любой программы - сначала "рыхлый" код, от которого добиваются работоспособности, затем оптимизация. Лично я придерживаюсь именно такой тактики, и нахожу ее достаточно удобной. Для примера выведем бегущую строку на обычный текстовый экран 80*25 - разрешение, к которому все привыкли (надеюсь) в **Norton Commander**. Не претендуя на оригинальность, скажу, что нам потребуется две строки в качестве данных/переменных. Одна из строк будет оригиналом, а вторая - служить для хранения подстроки. Алгоритм весьма прост:

• Цикл: вырезать подстроку выбранной длины;
• переместить курсор на выбранное место на экране;
• вывести подстроку;
• сделать задержку;
• сдвинуться в строке на символ;
• если не конец строки, то продолжать цикл.

Но перед тем, как набивать в отладчике код, я несколько облегчу вам работу.

Перенаправление ввода-вывода в ДОС.

Каждый раз набивать программу в отладчике было бы очень долго и может навсегда отвратить от программирования. К счастью, в ДОС (и, в частности,("в частности" убрать) в коммандной строке Windows (согласен, переставить слово надо)) можно использовать перенаправление стандартного ввода-вывода. Как это делается? Пишем текстовый файл, например, содержащий предыдущую программу "Hello, world!", в виде:

```
----8<---
   a 100
   mov ah,09
   mov dx,108
   int 21
   ret
   db "Hello, world!$"

rcx
   16
   n hello.com
   w
   q

--->8---
```

Не забудьте пустую строку после **DB**(она выводит отладчик из режима ассемблирования), и не забываем поставить в конце команду выхода! Сохраняем его под именем, например, **hello.dbg** и пишем в командной строке:

```
debug < hello.dbg
```

В результате отладчик берет команды не с клавиатуры, а из файла. Вводит программу на языке ассемблера начиная с адреса 0100, и сохраняет на диск под именем hello.com. И выходит в ДОС. А что, почти обычная программа - только вместо

```
ORG 0100h
```

имеем

```
a 100
```

и еще команды в конце.

Точно также, чтобы получить красивый листинг при дизассемблировании, не обязательно делать **PrintScreen** после команды **"u"**. Поскольку это наша программа и мы знаем, где находится код, а где данные, создаем файл **d_hello.dbg**, содержащий следующие строки:

```
u 100,107
   d 108,115
   q
```

И набираем в командной строке

```
debug hello.com < d_hello.dbg > hello.diz
```

Здесь мы перенаправляем как ввод (из файла **d_hello.dbg**), так и вывод (в файл **hello.diz**).

В результате получаем еще и файл **hello.diz**, содержащий дизассемблированную программу.

Таким образом, имеем *debug* в роли компилятора, и некий файл, содержащий команды отладчика и мнемокоды, в качестве программы. Для "компиляции" вызываем отладчик, используя ввод из файла, а для дизассемблирования - используем ввод из файла и вывод в файл вместо ввода с клавиатуры и вывода на экран. Да, не зацикливайтесь на расширениях - они в данном случае произвольны. Все эти файлы в принципе могут быть с расширением **".txt"** или вообще без расширений - кому как удобнее.

[прим.: Но это ещё не всё! В отличие от "простого" компилятора DEBUG может также сразу и выполнять (почти "интерпретировать") нашу "программу" (по аналогии с бейсиком), поэтому сохранять результат трансляции в отдельный файл нет необходимости - достаточно вместо команд RCX и N поставить G=100, не забыв завершить нашу программу инструкцией INT 3, если мы желаем увидеть содержимое регистров, с которым наша программа завершается. Итак, делаем следующий файл:

```
a
     xor ax,ax
     mov es,ax ; сегмент 0
     es:
     mov ax,[46c] ; адрес таймера в области данных BIOS
     es:
     mov dx,[46e]
     int 3 ; DX:AX содержат значение системного таймера

g=100
     q
```

сохраняем это под именем ticks, даём команду DEBUG<TICKS>RESULT, и наблюдаем в файле RESULT в регистрах DX и AX значение системного таймера. Впрочем, конкретно для данного случая наша программа может быть "короче":

```
d 0:46c l 4
     q
```

Использование внешнего файла и комментариев в нём также облегчает "настройку" адресов. Например, в исходном файле на месте меток можно вставить соответствующий комментарий, а в инструкциях перехода вместо имён меток будет фиктивное значение. Разумеется, в исходном файле не должно быть команды G, ведь программа ещё не завершена. Например:

```
a
     mov ax,100
     mov cx,100
     ;top:
     dec ax
     loop 100 ; top
     int 3
```

```
;g=100
     q
```

После первого "прохода" (выполнения DEBUG с перенаправлением ввода-вывода) следует проанализировать вывод на наличие сообщений об ошибках. Потом следует проверить значения закомментированных меток и скопировать эти значения в соответствующие места (в данном случае значение метки top нужно вставить после инструкции loop). Теперь можно расскоментарить команду G и программа готова. :)]

Вывод "бегущей строки" - создание программы с помощью отладчика.

Итак, набираем сразу в виде файла, чтобы потом "скомпилировать" его отладчиком.

```
---------Start of file------------
   a 100
   mov cx,555
   push cx
   mov si,200
   sub cx,555
   sub si,cx
   mov di,400
   mov cx,14
   rep movsb
   mov byte ptr [di],24
   mov bh,0
   mov dx,0c1f
   mov ah,02
   int 10
   mov dx,400
   mov ah,09
   int 21
   mov ah, 86
   inc cx
   inc cx
   xor dx,dx
   int 15
   mov ah,01
   int 16
   pop cx
   jnz 0140
   loop 0103
   jmp 0100
   ret
   db " ******** This is a long long string."
   db " It can be up to 64K long! You never see such long string"
   db " before! Enjoy it. Here may be some of your text and "
   db "advertizing. Just type what you want together"
   db " with WISPA chocolate! This program is debug handwork."
   db " All rights reserved. Decompilation and dizassembly "
   db " prohibited! The length of this string now 477 bytes"
   db " but length of code that print them is only 59 bytes."
   db " Sorry for my english, I write as I can. Alex FatMoon. $"

rcx
   400
   n r_string.com
   w
   q
   ------------End of file----------
```

Не забудьте поставить пробел перед долларом - он нужен, что бы конец строки затирал символы, остающиеся от предыдущего вывода. Сделаю некоторые пояснения:

• 555 - обозначил длину строки. На самом деле тут должно быть число, равное "адрес конца строки" - "адрес начала строки".

• 0200 - обозначим адрес начала строки. Мы его пока не знаем, но потом в листинге легко найдем.
• 0400 - адрес начала подстроки. Подстрока начнется сразу после конца строки. Этот адрес мы тоже пока не знаем.
• 0140 - адрес инструкции "ret". Он идет непосредственно перед строкой. И его тоже пока не знаем.

Сохраняем на диске 1К, поскольку точной длины программы опять же не знаем. Для тех, кто ввел очень длинную строку - можете сохранять не **0400h**, а **ff00h**. Наверняка где-то в начале этого блока все-таки уместилась наша

"гигантская" программа.

Теперь загружаем в отладчик почти готовый **r_string.com**, и дизассемблируем, отмечая исправления, которые надо внести: строка начинается с **013ch**, длина всей программы **0219h**, длина строки =**0319h-013ch=01ddh**, адрес **"ret"** - 013bh. Вносим изменения - можно прямо в отладчике, можно в текстовом файле. Все должно работать. Оптимизация, очистка экрана в начале и прочее - на ваше усмотрение.

> *[прим.: как упоминалось выше, для этого ничего дизассемблировать не нужно, достаточно выполнить один проход "компиляции" с перенаправлением вывода, после чего значения всех меток станут известны]*

Послесловие

> *Блаженны больные, ибо исцелятся.*
> **_ (почти из Библии) _**
> *И тебя вылечат… И меня вылечат!*
> **_ ("Иван Васильевич меняет профессию") _**

Вот собственно для этого и нужен компилятор - назначить адреса меткам и переменным и правильно вставить их в машинный код. Чем больше переменных и переходов, тем сложнее написать программу в отладчике. И слава богу, что это делать не обязательно, поскольку есть компиляторы. Если же читатель достаточно безумен, чтобы попробовать самостоятельно написать довольно объемистый проект, флаг в руки. Моего терпения хватило однажды на двух-килобайтную видеодемку, и я думаю, что писать программы под ДОС, используя только debug, вполне реально. Хотя и неудобно. А вот для написания небольших скриптов и тестовых фрагментов использование DEBUG в качестве компилятора - самое оно. И, кстати, при этом автоматом решается проблема лицензионности компилятора. J

При этом надо помнить несколько вещей:

• смена сегментного регистра делается в две команды, поскольку это префикс, как и REP, LOCK и другие. То есть `mov si, es:[di]` записывается в виде

```
es:
    mov si, [di]
```

• переходы типа **short** возможны лишь в пределах 128 байт выше или ниже. Из-за этого могут быть проблемы при вводе программ вышеописанным способом.
• отладчик не выдаст ошибки, если какая-то инструкция неправильна. То есть, выдаст, но мы ее можем не увидеть, если используем перенаправленный ввод.
• некоторые машинные коды 186-го, 286-го, 386-го и выше процов к сожалению остаются для debug'а неизвестными, а именно:

```
pusha
    popa
    push <immediate>
```

```
bound
arpl
```

• все команды, появившиеся в 486 и позднее
• все команды, использующие 32-битную адресацию и операнды
• insb / insw
• и некоторые другие.

Проблему можно решить, занося непосредственно код через

```
db <opcode>
```

Да, если вы хотите писать в отладчике, все эти машинные коды придется запомнить. После этого в голове возникает каша из цифр и букв, и человек неадекватно реагирует на окружающих. Вот и я… пойду-ка лечиться! Поскольку подобные упражнения легко могут подорвать психическое здоровье, всем рекомендую лекарство - бутылочки 3 пива минимум.

Записки Dzenствующего

Автор: Edmond / HI-TECH

Дата: 2002-11-21

Edmond :: HI-TECH group

1 От автора

Скачайте пример (находится в архиве `wasm_files.zip`: `files/recovered/210/tpl.zip`).

Специальный пакет включаемых файлов (находится в архиве `wasm_files.zip`: `files/recovered/210/win32inc.zip`), а так же авторский windows.mac (находится в архиве `wasm_files.zip`: `files/recovered/210/winmac.zip`).

Настоящая статья призвана расставить некоторые точки над Ё. Автор предполагает, что вы уже неплохо разобрались в основах Win32, и теперь хочет внести некоторые коррективы в то, что вы уже усвоили.

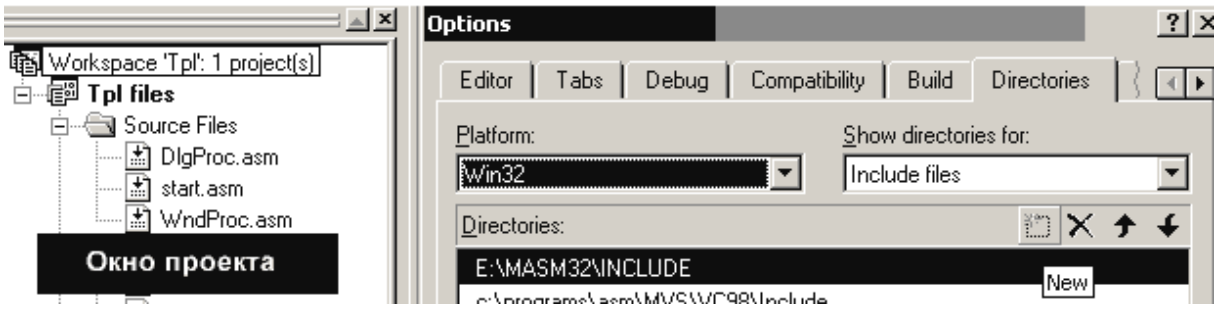
Автор надеется, что Вы прочитали Туториалы Iczelion'a (находится в архиве `wasm_files.zip`: `files/recovered/210/iczelion.zip`).
И ЭТО, и ЭТО.
Увы, но обучение не может идти плавно, оно всегда происходит скачками или, как говорят математики: итерациями.

В этой статье мы проведём очень важную итерацию на пути вашего самосовершенствования. Автор не пытается рассказать в корне что-то новое, но лишь отполировать старое.

2 Visual Studio - среда разработки, или в поисках оазиса.

Тема среды разработки - больная тема для программистов на ассемблере. Она часто, время от времени проходит на форумах, но лучше от этого не становиться. Автор перепробовал множество сред, созданных различными энтузиастами, и не так давно использовал прелестный UltraEdit. (Настройки для подсветки ASM вы можете скачать (находится в архиве `wasm_files.zip`: `files/recovered/210/ued.zip`)). Однако ни что не могло полностью удовлетворить автора. В конечном счете, лучшим мог бы оказаться VS (жаль, нет такого энтузиаста, который смог бы написать расширение под VS, поддерживающее особенности ASM). А это, без сомнения, возможно.

Откройте в VS пример. Слева, на закладке FileView вы увидите структуру проекта Tpl - шаблон. Для успешной компиляции проекта в среде VS должны быть установлены пути к файлам включения и компиляторам. Эта настройка выполняется по пути: Menu > Tools > Options… на вкладке Directories. Добавьте в соответствующие разделы пути. Вот пример того, как это выглядит.



Теперь, когда пути определены, вы можете:

1. Не прописывать их в переменных окружения.
2. Не прописывать их в исходниках.

Чтобы убедиться, что все верно, произведите нужные изменения в файле Win32API.inc, и скомпилируйте пример, нажав Ctrl+F5.

Если всё получилось, мои поздравления! Если Вы ещё не работали с VS 6.0, автор уверен, что никаких проблем не случится. Если вы работали с VS, то вам можно вообще пропустить этот раздел.

Для тех, кто ещё не работал в VS рекомендую прочитать статью SvetlOff. Пример настройки VS вы можете посмотреть в проекте исходника. Единственное, что хочется добавить: Современная VS позволяет делать настройки компиляции в мультирежиме. Поэтому вы можете вставить в проект сразу несколько ASM файлов, а потом на контекстном меню из под Source Files в Settings на закладке Custom Build установить нужные параметры. При этом, параметры будут установлены сразу на всех файлах проекта. Нездоровый пример минимального файла - результат настройки выравнивания линковщика 1000h, то есть 4k. В зависимости от вашего желания вы можете опустить эту планку до 16, советую до 64 при помощи директивы /ALIGN:xx (не обращайте внимание на ругательства линковщика - это он умничает).

3 Вызов функций Win32

Уверен, что большинство из вас, использует метод Invoke для вызова функций. Это действительно удобный метод. Однако, в нём есть два недостатка:

1. Необходимость использования заглушек.
2. Параметры будут помещены в стек.

Что значит заглушек?

В мире программирования существует древняя проблема терминологий.

Заглушкой будем называть такой КОД , цель которого вызвать требуемую функцию. Сам же код заглушки может выполнять подготовительные операции, такие как обеспечение интерфейса, вызывающий код/вызываемый код.

Переменной Переходником (или далее просто переходником) будем называть указатель на функцию, через который осуществляется её вызов.

Если вы уже несколько разобрались в механизме импорта функций, то должны были уяснить, что, по сути, в адресном пространстве программы инициализируется таблица указателей на функции DLL, эта таблица жёстко фиксирована и привязана к базовому адресу загрузки. Так как, в зависимости от многих факторов, адреса функций DLL могут быть различны, то эта таблица указателей заполняется загрузчиком реальными адресами функций. Это значит, что вызов любой

функции Win32 должен выглядеть следующим образом:

```
Call dword ptr Win32Function_pointer
```

На самом деле, можно, конечно, добиться и прямого вызова. Это действительно выполнимо и очень эффективно не только с точки зрения некоторой шустрости call, а и по причинам системного характера.
Интересно будет узнать, что при создании файла lib, наряду с привычными _Fun@x экспортируются указатели (dword) на функции DLL. В своей сущности эти указатели не что иное как переменные, находящиеся в массивах **IMAGE_THUNK_DATA**.

Ещё более интересней то, что, без всякого микроскопа заглянув в LIB их легко увидеть в начале как:
__IMPORT_DESCRIPTOR_USER32 __NULL_IMPORT_DESCRIPTOR USER32_NULL_THUNK_DATA _EditWndProc@16 __imp__EditWndProc@16 _RegisterClassA@4 __imp__RegisterClassA@4 _RegisterClassW@4

** Автор замечает, что эти экспортируемые переменные находятся не в секции data, а в специальной секции IMPORT_DESCRIPTOR. Более подробную информацию читатель сможет найти в документации COFF формата.*

Я уверен, что читатель, если не знал, то догадается без подсказки, что это таблица соответствий указателей и импортируемых имён функций.
При этом посмотрите, какая зависимость в их именовании. Переходник именуется так: __imp__FunName@xx
А функция так: _FunName@xx

Когда вы пишите что-то вроде:

```
call __imp__FunName@xx
```

Вы получаете то же самое, если бы имели

```
EXTERN __imp__FunName@xx:dword
```

то есть косвенный вызов по указателю:

```
call dword ptr __imp__FunName@xx
```

Но, когда вы пользуетесь идентификатором типа _FunName@xx, линкёр преобразует такой вызов в

```
34: call _InitCommonControls@0
 00401090 call InitCommonControls (0040129c)
 35:
```

где InitCommonControls (0040129c) это

```
0040129C jmp dword ptr [__imp__InitCommonControls@0 (00405170)]
```

то есть функция заглушка.
Любопытно будет отметить, что заглушки помещаются в конец кода, это хорошо видно при помощи dumppe.exe (когда Debugger VS искажает информацию)

```
0040122E _ExitProcess@4:
   0040122E FF25A0514000 jmp dword ptr [ExitProcess]
   00401234 _EndDialog@8:
   00401234 FF2538524000 jmp dword ptr [EndDialog]
```

А в версии Debug переходники есть всегда вне зависимости от того, используются они или нет.
Какой ваш вывод? Нет заглушкам? Надеюсь на это. Мало того, что это лишняя команда jmp, так он ещё и увеличивает код. А потом стыдно должно быть перед ЯВУшниками, так как хотя бы тот же C++ уже научили вызывать функции Win32 без заглушек!!!

Так что? Теперь вручную прописать inc файлы? Слава богу, нет! Нашёлся тот человек, который научился при помощи PROTO описывать косвенный вызов в Invoke. Я нашёл этот замечательный пример в пакете MASM32 7.0 в папке L2extia. Там есть подобная утилита, которая генерирует inc файлы, что позволяет использовать Invoke и, при этом, получать код без переходника.

Но перед этим вы обязаны включить в windows.inc макрос.

Я несколько улучшил и переделал эти файлы, разработав полноценный комплекс. Однако об этом попозже.

Следующий шаг, на пути к раскрепощению -- избавиться от противной директивы STDCALL. Почему она такая противная? Да хотя бы, потому, что добавляет слева к имени функциями символ подчёркивания. Конечно, это редко когда мешает, и, тем не менее, программист на ассемблере обязан иметь полную свободу в именовании идентификаторов, иначе, зачем тогда ассемблер, который сковывает программиста.

Решить эту проблему оказалось легко, только немного изменив макрос:

```
ArgCount MACRO number
  LOCAL txt
  txt equ <typedef PROTO STDCALL :DWORD>
  REPEAT number - 1
  txt CATSTR txt,<,:DWORD>
  ENDM
  EXITM <txt>
  ENDM

pr0 typedef PROTO STDCALL
  pr1 ArgCount(1)
  pr2 ArgCount(2)
  ……
```

Теперь вы можете окончательно забыть о прошлых оковах, и использовать Invoke, получая хороший код, и освободившись от STDCALL влияния. Всё в ваших руках.

4 Бессознательная оптимизация в Win32

Бессознательная оптимизация - это такая оптимизация, на которую программист не тратит, или почти не тратит, умственных сил и времени.

Сама по себе, в разной степени, эта способность приходит постепенно, однако, начинать тренироваться стоит уже теперь.

Громадную роль в оптимизации Win32 кода является учёт его особенностей. А так случилось, что код Win32 полон таких особенностей. Итак, в путь! К вершинам!

4.1 Использование регистров или культ STDCALL.

Я напомню мат. часть:
В соглашении вызова процедур **STDCALL** предусматривается следующее:

1. Регистры **EBX, ESI, EDI, EBP** - не могут изменяться в **STDCALL** функции.
2. Параметры помещаются по значению подобно C-конвенции: от последнего к первому.
3. Функция сама отчищает стек от параметров.

Следует похвалить разработчиков **MS** , так как такая конвенция обладает высокой степенью эффективности вызова.

Теперь можно сформировать очень чёткие рекомендации:

1. Используйте регистр **EBX** для хранения часто используемых констант, переменных в вызывающем коде.
2. Используйте изменяемые регистры для хранения констант, переменных между вызовами **STDCALL** функций.
3. Используйте приём предварительного помещения в стек, для вызовов **Win32** функций.
4. Если константа используется более 3-х раз и больше байта, сохраните константу в регистре для последующего использования. То же верно и для переменных.

Не долго думая, автор, как рьяный поклонник Зубкова, сразу вспоминает о хорошей привычке использовать регистр **EBX** для хранения **0** , констант или переменных частого использования. Под Win32 данная рекомендация очень особенна. Именно этот регистр остается свободным, когда **EBP** - хранит данные кадра стека, **ESI** , **EDI** - адрес буфера, буферов, а данные нужны во время вызовов **STDCALL** функций. Единственное спасение это **EBX**.

Разбирая мой непричёсанный пример (и, слава богу, я не хочу навязывать свой стиль) вы заметите, что ebx, он же **$$$__null** , везде, где попало, хранит нулевую константу, которую очень просто получить. Вот так:

```
xor ebx,ebx
```

Как можно увидеть ЯВУ уже то же научилось этому простенькому фокусу, и поэтому человеку не пристало отставать.

Проблема пропадания регистра **ecx** создаёт две дополнительных операции (или больше) в циклах. Это особенно ощутимо в небольших циклах (до 50-40 команд), пока **ЯВУ** продолжает быть верным **ECX** , есть смысл перейти на другой регистр снова же **EBX** , конечно, при условии, если это действительно имеет смысл.
Подобным образом легко обеспечить значение **TRUE** , вот так:

```
inc al
```

Так частый прием, который вы можете увидеть в примере это помещение заранее параметров в стек. Это очень сильный приём и достаточно лёгкий. Обычно, при написании участка кода я «осматриваю» одновременно 10, 20 команд, и бессознательно 50, плюс чувство контекста. Этого вполне достаточно, чтобы, не задумываясь, выполнять приём предварительного размещения параметров в стеке. Со временем приём входит в привычку, и перестаёшь его замечать, концентрируя внимание на алгоритме.
Вот типичный пример использования приёма (здесь **$$$__result** это **eax**):

```
Invoke CreateWindowEx,
 ……
 ;; Приём предварительного помещения в стек
 ;; Мы помещаем дескриптор окна дважды... один раз для
 ;; ShowWindow, другой раз для UpdateWindow

 push $$$__result
 push SW_SHOWNORMAL
 push $$$__result
 call ShowWindow
 call UpdateWindow
```

После вызова **CreateWindow** возвращает хэндл окна, который после используется несколькими функциями подряд. Это и есть тот классический случай, когда следует использовать данный приём. Другой случай, отличается тем, когда API функции получающие результат другой функции находятся достаточно далеко друг от друга, и вам не хватает регистров, чтобы эффективно создать код. В этом случае предварительное помещение параметров в стек, или просто сохранение их там, экономит достаточно ресурсов. Обычно такая проблема решается локальными переменными. То есть после вызова функции, результат сохраняется в переменной, а потом используется. Но это характерно для ЯВУ, и не характерно для АСМ, где вы сами можете управлять методиками.

Кроме того, более эффективен метод смешивания команд push и каких-нибудь других команд процессора, особенно команд состоящих из двух, одной микроопераций. Всё это повышает вероятность написания оптимального кода, так как наиболее легко спариваются команды в вереницах с push, не связанные с операциями промежуточного характера. Например, вы имеете код, который вычисляет что-то, а потом вызывает API функцию. Конечно, вы можете сделать так, как удобно человеку, но можете сделать и так, как удобно машине: смешать команды вычисления, которые очень часто не зависят от многих параметров API функции и только потом осуществить вызов. Пример:

```
;; Исходный код
……
div esi
mul edx,10
add eax,edx
Invoke Win32Fun eax,CONST1,CONST2
……
;; Или
div esi
push CONST2
mul edx,10
push CONST1
add eax,edx
push eax
call Win32Fun
```

Ко всему прочему, вы так же можете заметить подобные приёмы и в основном цикле программы.

4.2 Стековый кадр и локальные переменные

На сегодняшний день вошло в моду использовать только локальные переменные по-возможности, хотя, по-моему, это не столь существенно, в отличие от нежелания хоть сколь нибудь грамотно создавать софт.

Все локальные переменные делятся на несколько групп:

1. Неинициализируемые
2. Инициализируемые
3. До базы стекового кадра ([ebp+xx])
4. По месту применения

Если вы используйте локальные переменные, которые обычно располагаются после базы стекового кадра, (то есть к ним следует ссылаться как [ebp-xx]) придерживайтесь рекомендаций.

Чтобы разместить неинициализируемые переменные достаточно просто изменить esp и пользоваться ebp -- указатель базы стекового кадра. В общем случае это неплохо, если б не AGI (задержка при генерации адреса), которая впрочем, не возникает на современных процессорах.

С инициализируемыми переменными более сложнее, здесь не только следует выделить пространство, а и произвести их инициализацию. Надеюсь мне не нужно разъяснять, насколько это неэффективно, по сравнению с простым объявлением переменных в памяти (статические переменные). А поэтому стоит всё-таки ещё семь раз подумать. Однако, если вам нужно разместить в стеке структуру заполненную нулями или чем-то заполненную, и размер этой структуры не так велик, то более грамотный способ - использовать старые добрые push. Это особенно верно, когда структура небольшая по размерам. Даже метод заполнения нулём при помощи push эффективней, чем все остальные.

```
……
   xor eax,eax
   push eax
   push eax
```

```
   push eax
   push eax
   ;; Заполнение структуры из четырёх dword
```

Как правило, заполнение нулями всей структуры не имеет смысла. Поэтому тренд с push рулит ещё более. Если же структура очень большая, разумно организовать цикл из push. Чтобы цикл был эффективным, следует использовать две или четыре push, если структура кратна 4 dword. Иначе лишние push (которые на самом деле не нужны) так же легко стереть pop, или увеличением esp. Правда здесь ещё можно поспорить с Win32 функцией ZeroMemory, однако, я думаю, не стоит.

Если же вам приходиться изменять локальные данные уже после их создания, где-нибудь в коде, то, к сожалению, воспользоваться push можно только тогда, когда данные, стоящие в стеке после изменяемой структуры и в самой структуре вам не нужны. Делается это просто: Вы изменяете esp так, чтобы он указывал на структуру (а точнее на её последний элемент), а потом push делаете своё дело. В придачу между push, вы можете так же вызывать API функции, результат выполнения которой, значения переменных в этой структуре. Эффективность такого подхода просто восхищает по сравнению со всеми остальными. Пример:

```
 ;; Я хочу проинициализировать структуру POINTS(4*POINT(x,y)), ;; ( (x,y) занимают    вместе
    двойное слово)
 ;; которая расположена в стеке после 16 байт от базы ebp.

          | параметры функции
ebp ==>   | адрес возврата
ebp-16    | Другие переменные
ebp-16-16 | структура POINTS(4*POINT(x,y))
esp ==>   | где-то там далеко (меньшие адреса)

xchg ebp,esp
   sub esp,16
   ;; После этих двух команд параметры процедуры не
   ;; доступны через ebp, если вам кадр стека нужен,
   ;; можно сделать и так, пожертвовав другим регистром
   ;; mov ebx,esp
   ;; lea esp,[ebp-16]

        | параметры функции
ebp ==> | адрес возврата
        | Другие переменные
esp ==> | структура POINTS(4*POINT(x,y)) (конец)
        | структура POINTS(4*POINT(x,y)) (начало)
        | где-то там далеко (меньшие адреса)

;; Всё пошли
   Invoke GetXCoord,параметры
   push eax
   Invoke GetYCoord,параметры
   push eax
   ……….
   ;; Организуем цикл, или без цикла…
   ……
   xchg ebp,esp
   add ebp,16
```

Уверен, изобретательный читатель найдёт и свой вариант сохранения кадра стека, этим и замечателен ассемблер. А он, откровенно говоря, возможен, если увериться, что размер области локальных переменных постоянен.

Посмотрим на этот фокус. При определении стекового кадра мы установим ebp не на область начала параметров процедуры, а на конец области определения локальных данных.

```
        | параметры функции
        | адрес возврата
ebp ==> | Локальные переменные
        | где-то там далеко (меньшие адреса)
```

Если объём локальных данных известен заранее, и при модификации структур стек будет пуст до ebp+0. То мы можем вообще не сохранять esp, так как его значение будет храниться постоянно в ebp.
Естественно, такой приём не сработает, если вы собираетесь содержать локальные данные переменной длины.

Уверен, что у вас возник вопрос: «А если не вызывать функции? Данные стека хранящиеся после, и сама структура не разрушаться, не так ли?» Вообще говоря, верно, если только не одно «Но». Имя этому «Но»: SEH - структурная обработка исключений. Если вы помните, она так же использует стек, и соответственно без всякого на то предупреждения может затереть ваши данные. А ведь SEH используется чаще, чем вы можете догадываться. Оно используется при операциях с памятью, и выполняется в вашей программе даже тогда, когда вы сами не используйте её. Всё это означает, что увеличивать esp - опасно для данных стека. Лучше не создавать трудно обнаруживаемые ошибки.

Но вернёмся к локальным данным переменной длины. Это хороший приём, когда нужно быстро получить участок памяти. Об этом подробно описано здесь.
Как я уже говорил, официальная политика современного программирования есть отказ от глобальных переменных, и вообще от переменных статического характера. Несколько комментариев по этому поводу состоят в том, что программирование движется в сторону «окультуривания к человеку», то есть создание такой программы, которая была бы удобна человеку, а не машине. Только хотя бы эта фраза кажется абсурдом, и, тем не менее, тенденция остается.

Но автор надеться, что читатель имеет свою голову на плечах, и не будет впадать в крайности человеческой жадности, и будет размещать локальные данные не только в стеке, но и в обыкновенном сегменте данных. Или иначе обычное static директива как в C. Конечно, есть много "Но", из-за которых этот метод следует избегать:

1. Рекурсивные процедуры
2. Код, исполняющийся в нескольких потоках

Однако, если:

1. Структура достаточно велика
2. Структуру следует заполнить в основном статическими данными (то есть константами), или заполнить один раз на этапе инициализации.
3. Не все данные в структуре в последствии изменяемы функцией.

То стоит подумать о размещении структуры не в стеке, а в сегменте данных. Это:

1. Сэкономит память, так как размещение структуры в сегменте данных занимает меньше памяти, чем код её размещения в стеке и инициализации
2. Вне конкуренции по скорости инициализации - вообще не тратит время на инициализацию, или тратит, но незначительно
3. Быстрее в доступе, нежели тяжёлые команды с ebp-xxh.

Это всё может дать достаточно ощутимый выигрыш и по размеру и по скорости и по качеству кода. Но, увы, остерегайтесь подводных камней. Тем не менее, данная рекомендация сгодится не только асм-разработчикам, но и разработчикам C++. Так что думайте.

Последнее замечание по доступу к данным стека - так это прекрастная возможность использовать для доступа к стеку не только регистр ebp, но и любой другой. И, при этом, без всяких там префиксов замены сегмента. То есть, такой код будет работать:

```
push 1234h
  mov esi,esp
  mov eax,[esi] ;; eax = 1234h
```

Посмотрите на картину DEBUG окна в VS:

```
CS = 001B DS = 0023 ES = 0023
   SS = 0023 FS = 0038 GS = 0000
```

Насколько можно увидеть у сегментов DS, ES, SS - селекторы одинаковые. Так, то всё очень строго научно, и скажем так, без риска.

4.3 Особенности оптимизация кода под Win32

Если читатель уже ознакомился с премудростями оптимизации кода, это ещё не значит, что эти премудрости так хороши и верны под **Win32**. **Win32** имеет свои подводные камни, о которых нужно знать. Красота и искусство оптимизации на ассемблере под **Win32** - это умение сочетать где надо оптимизацию по скорости, а где надо - по размеру. Как правило, оптимизация по скорости и размеру одновременно случается редко. К счастью **Win32** настолько хорошо подвержено аналитическому разбору, что автор смог выделить очень чёткие рекомендации по особенностям оптимизации кода под **Win32**. Эти рекомендации хорошо представить как таблицу основных мест где стоит и как стоит оптимизировать:

Список мест, которые оптимизируются по скорости (то есть оптимизация по скорости - есть решающей):

1. Процедура окна (анализ сообщений)
2. Длинные вычисления при прорисовке.
3. Циклы автономных функций и конверторов, которые не содержат в нутрии вызовы Win32 API системных функций, особенно из kernel32.dll (функций ядра)
4. Процедуры конвертирования чего угодно, имеющие автономное значение
5. Любой значимый объём кода, не имеющий вызовов Win32
6. Код анализа состояния в мультипотоковых системах.

Список оптимизации по размеру:

1. Код, насыщенный вызовами системных функций ядра
2. Код, выполняющийся рядом с переключением задач
3. Код, перенасыщенный другими тяжёлыми функциями Win32
4. Код, конвертирования, и других преобразований для вывода на экран, и код связанный с **GUI** , кроме пунктов подходящих к предыдущей таблице.
5. Код процедуры диалога
6. Код инициализации чего угодно.
7. Код удаления и ининициализации
8. Код обработки ошибок
9. Код фреймов SEH и проверки ошибок.
10. Код мультипотоковых взаимодействий.

Теперь стоит остановиться, прочитать этот список ещё раз и глубоко вздохнуть. То, что вы теперь прочитаете, не написано ни в одной книге. Автор удивляется, почему об этом молчат профессионалы, или это настолько ужасный секрет, или об этом просто не задумываются. Перед тем, как я объясню суть, я хочу, чтобы вы заметили, что больше пунктов в оптимизации по размеру, однако это не значит, что всегда и объём кода припадающий на вторую группу больше объёма кода первой группы. Хотя это зависит от программы. Это противоречит современному утверждению ЯВУшников: «Памяти много, и оптимизировать стоит по скорости, а не по размеру».
На самом деле, в идеале, оптимизировать действительно следовало бы, отдавая предпочтение времени выполнения, нежели размеру кода. Однако в пунктах первого списка оптимизация по скорости не имеет смысла. А раз так, то смысл имеет оптимизация по размеру. То есть: «Если мы

не можем выиграть в скорости, мы можем выиграть в размере». Причина невозможности оптимизации по времени кроется в длительности исполнения самих функций Win32. Нет смысла оптимизировать код из 10-50 команд (кроме чистого цикла) по скорости, если потом идёт вызов тяжёлой Win32 функции. Тяжёлыми можно считать все функции ядра, так как в любом случае, при переходе в режим ядра тратится более 1000 тактов (как это любит писать Джеффри Рихтер). Кроме того, тяжёлыми следует считать многие функции **GDI** и другие. То есть скоростной оптимизации может подлежать только тот код, который не содержит в себе вызовов Win32. А это значит следующее:

• **Если возможно, избегайте вызовов функций внутри циклов, или делайте так, чтобы все вызовы в цикле находились в одном месте, или в двух, но так, чтобы большая часть цикла была «чистым кодом».**
• **Создавайте код таким образом, чтобы не смешивать Win32 API с кодом от него «независящим». Это не только даст возможность скоростной оптимизации, но и возможность повышения степени переносимости вашего кода.**

А теперь обратите внимание на пункт 6 в первом списке: «Код мультипотоковых взаимодействий». Это некоторое исключение из правил, но только подтверждающее правило. Это значит, что код, располагающийся после функций ожидания, или между функциями ожидания следует оптимизировать по скорости. Причина такой оптимизации - вероятность выполнения кода без прерываний, что так же повышает общую эффективность приложения. Рекомендации 7-10 второго списка характерны не только для Win32, а вообще универсальны для всего программирования. Причина отсутствия оптимизации по скорости в процедурах инициализации, обусловлена следующими факторами:

1. Выполнение этих процедур один или малое количество раз
2. Требования к стабильности кода в этих процедурах
3. Операции выделения памяти/записи в файлы не могут быть быстрыми, и соответственно нет смысла в оптимизации кода по времени (скорости).

Это кстати, ещё один плюс метода предварительного помещения параметров в стек, или метод смешанного помещения. Так как код типа:

```
push xx
push xx
call win32
call win32
Имеет намного высокую вероятность более быстрого выполнения, чем код
push xx
call win32
push xx
call win32
```

Так как команды push стоящие подряд кэшируются, а после вызова функции кэш уже может содержать совсем другой код.

4.4 Рекомендации при построении WndProc

Более подробно оптимизация WndProc была описана в моей статье: «Эффективный анализ сообщений в WndProc». Однако, кроме этого я повторю основные рекомендации.

1. Анализируйте первыми те сообщения, которые приходят в процедуру окна чаще.
2. Не создавайте стековый кадр для анализа сообщений (как это показано в моём примере)
3. Не оптимизируйте код анализа сообщений в процедуре диалога, это имеет малый смысл.
4. Используйте одну процедуру диалога на всю программу, для идентификации используйте **lParam**.

5. Назначайте контролам порядковые идентификаторы
6. Назначайте меню порядковые идентификаторы, чтобы можно было воспользоваться таблицей переходов - это очень эффективно.
7. Группируйте анализ в **WM_COMMAND** WndProc, не по ID сообщения от контрола, а по идентификатору контрола.
8. Группируйте анализ в **WM_COMMAND** DlgProc, не по идентификатору контрола, а по ID сообщения от контрола.

Некоторые комментарии.

Отсутствие оптимизации для Процедур Диалога, связанно с малым влиянием их скорости на общую скорость обработки. Во-первых, диалоговая процедура - это процедура, вызываемая из какой-то WndProc, определённой в **USER32** , и поэтому основная производительность лежит на ней, а не на Вашей процедуре. Во-вторых - диалоговая процедура по определению управляет интерфейсом, а поэтому не может быть быстрее, чем функции прорисовки. Нет смысла скоростной оптимизации.

Приём использования одной диалоговой процедуры - прекрасный приём. Это оказывается естественным и лёгким, так как все контролы имеют разные идентификаторы. Единственную разницу в процедурах можно учитывать с помощью дополнительных элементов в структуре Wnd. Там как раз есть один такой элемент **GWL_USERDATA** , или, в конце концов, **GWL_ID** , **DWLP_USER** , к которым можно иметь доступ при помощи функции SetWindowLong/ **GetWindowLong**. А для учёта типа диалога при инициализации можно воспользоваться значением lParam, для передачи в процедуру диалога дополнительной информации.
Так же следует уделить внимание способу разбора **WM_COMMAND** / **WM_NOTIFY**. Обычно автор видит примеры, где процедуры анализируют **HIWORD(wParam)** , когда более эффективней группировать сообщения не по кодам нотификации, а по контролам, то есть чинить разбор **wID = LOWORD(wParam)**. Это должно быть понятным читателю, так как обычно приложение не отвечает на все коды нотификаций, но за то обрабатывает все контролы.
В диалоговой процедуре, как не странно, имеет смысл поменять такой порядок. Там, например, хорошо определить код нотификации, а потом от кого он пришёл. Это действительно удобно. Например, у вас много кнопок. Вы определяете, что это код нотификации **BN_CLICKED** , а после (если контролы кнопок идут подряд), сделать косвенный вызов по таблице указателей, или нечто другое.

4.5 Оптимальное Размещение кода и данных в приложении

Когда заходит вопрос об оптимальном размещении кода в приложении, а значит и о выравнивании данных, мне становиться не по себе. Нет, ну почему ЯВУ не могут научить грамотно распределять данные? Ужасная проблема выравнивания на самом деле не столько проблема, сколько лень. Итак, данные должны быть выровнены! Но это не значит, что для их выравнивания требуются множество пустых незадействованных байт!!! Нет и ещё раз нет. При помощи грамотного распределения данных вы всегда сможете получить такое выравнивание, какое требуется. Обычно рекомендации таковы:

1. Размещайте данные по закону убывания. Более сложные, большие вперёд, более меньшие назад. (Единственная оговорка состоит в том, что начало сегмента обязано удовлетворять запросам выравнивания данных)
2. Размещайте данные одного размера в одном месте
3. Размещайте данные, используемые в одной зоне кода в одной зоне сегмента данных, и не раскидывайте их впустую.

Это элементарные правила, как и многое из того, что было написано в статье, однако они не соблюдаются, по неизвестным причинам. Размещение данных по закону убывание кажется естественным:

1. STRUCTURS
2. QWORDS
3. DWORDS
4. WORDS
5. BYTES

И зачем вообще думать о выравнивании данных?
Другое вообще никогда не используется: это оптимизация размещения кода.
Хочу напомнить читателю мат. часть, в которой говориться, что Windows использует механизм отображения файлов для файлов EXE во время исполнения. Это означает, что если команда перехода, например, обратилась по адресу незагруженного в память блока кода (данных), то система подгружает его по правилам виртуального адресного пространства: она обнаруживает, что запрошенная виртуальная страница отсутствует в памяти, и загружает данные. То, что этот процесс тратит сравнительно много времени, говорить не приходится.
Говорить приходится о том, что программист, не обязательно асмовец, хотя ему легче, обязан так группировать код и данные, чтобы взаимосвязанный код и данные находились физически в одном месте адресного пространства. Отсюда же образуется правило: Держать код обработки ошибок сбоев, настроек, запуска, инициализации и так далее в отдельном месте от «функционирующего кода». К сожалению, такое положение неудобно для человека, и, тем не менее, вы не представляете, насколько данная оптимизация может поднять производительность в объемных корпоративных продуктах. Или просто в программах больше одного, двух мегабайтов.

5 Особенности Win32

5.1 Особенности выделения в стеке больших объёмов данных

Когда вы имеете код, которому вот-вот на несколько вызовов нужна память менее 2-4 килобайт, вы можете использовать хип. Однако, если ваша функция не рекурсивна, и не выполняется в нескольких потоках, быстрее и качественней выделить буфер или просто участок памяти в стеке. Это более эффективно, чем выделение памяти функциями **Win32**. И при этом, что для доступа к этой памяти можно использовать указатели в esi/edi - стековая память ни чем не отличается от обычной.
Но есть один, два подводных камня подстерегающих вас на пути.
Стек Windows, это обычная страничная память с правами чтения записи. Однако система благоразумно не выделяет все страницы памяти, а только резервирует их. Точнее говоря, на момент старта приложения, вы получаете такую картину:

Сторожевой блок
Выделенный блок
Сторожевой блок

При увеличении объёма стека картина изменяется:

Сторожевой блок
Выделенный блок

Сторожевой блок
Выделенный блок
Сторожевой блок

И так далее, пока не будет исчерпано всё зарезервированное пространство, которое по умолчанию составляет 1M.
Windows использует механизм **SEH** , чтобы выделять дополнительное пространство стека. Сторожевой блок представляет собой виртуальную страницу памяти, при обращении к которой, возбуждается исключение. Результатом обработки такого исключения есть изменение прав доступа к данной странице, и передача памяти следующей страницы, которая становиться очередным сторожевым блоком. И так далее….
В действительности, механизм реализации стека несколько отличен для ядер NT и 9x типа. Механизм для 9x несколько проще, чем для NT. Главное отличие состоит в этом:
Формирование стека под 9x

Сторожевой блок
Выделенный блок
Выделенный блок
Сторожевой блок (Переданная страница с атрибутом NOACCESS)
Reserved Page NOACCESS
Reserved Page NOACCESS

Формирование стека под NT

Сторожевой блок
Выделенный блок
Выделенный блок
Сторожевой блок (Переданная страница с атрибутом PAGE_GUARD)
Reserved Page NOACCESS
Reserved Page NOACCESS

Вообще Windows 9x не поддерживает флаг **PAGE_GUARD** , который не отличается физически от флага **NOACCESS**.
Проблема возникает тогда, когда программа пытается обратиться к области стека за предел сторожевой страницы. В этот момент происходит исключение **0xC0000005** , то есть, и, получив такое исключение, стандартный обработчик Windows, что самое интересное, даже не выдаст никакого окошка. Он просто уничтожит Ваш процесс.

Чтобы этого не произошло, следует выделять память объёмом больше 4k не сразу, а по частям. Сперва 4k, а потом остальную часть, если нужно тоже по 4k - размер виртуальной страницы на Windows.

5.2 Страничная адресация - благо?

Особенность EXE файла, от форматов для ДОС велика. И велика она хотя бы в том, что программа в формате PE Аля Windows - это буквально схема программы, которую уже можно загрузить без каких либо модификаций содержимого.

Новичкам, только осваивающим страничную адресацию, обычно кажется, что вот, наконец - это и есть панацея от бед сегментного мира DOS. Однако так только кажется. Если под DOS программа не привязана к сегменту, в котором выполняется, то код созданный линковщиком, навсегда и всесильно привязан к адресу **400000h**. Именно эту цифру вы и получаете, когда пытаетесь вызвать **GetModuleHandle(0)**

А почему? А потому, что FLAT. Как говорится, за что боролись, в то и влипли. Это значит, что если загрузчику вздумается загрузить всю или часть программы по другому адресу он обязан будет модифицировать все ссылки и метки так, чтобы код корректно работал. И хотя суть идентификации проста - прибавить, отнять смещение от предполагаемого адреса загрузки, представьте себе, сколько нужно сделать операций хотя бы для средненькой программки.
И что самое увлекательное, что этот недостаток присущ не только EXE, но и DLL, которые отличаются от EXE только каким-нибудь флагом в PE заголовке и несколькими другими особенностями.

Самое обидное - это цена, которую приходиться платить, и эта цена не только во времени запуска приложения или DLL. А так же другой фактор.

Дело в том, что, модифицируя код программы, система естественно уже не может пользоваться отображением на файл. Она размещает по сути новый exe (DLL) в страничном файле. Это похоже на кошмар. Более того, DLL, по сути, перестаёт быть DLL, это просто кусок кода связанный с кодом вашего файла. DLL теряет все свойства совместного использования. Так как системой создаётся новая DLL да ещё в страничном файле. Интересно и какое право после этого DLL можно назвать DLL?

Откровенно говоря, наиболее легче было бы при инсталляции просто слепить программу с учётом всех особенностей в один или несколько файлов, но который бы исполнялся. Вам будет интересно узнать, что такая возможность существует. Но думаю это тема уже для следующих статей.

Однако какой после этого должен быть сделать вывод? Вывод должен быть таковым: «**Программа всегда должна загружаться по базовому адресу!!!** » Иначе лучше уже вообще её не загружать.

Разбирая мой пример, вы, наверное, заметили, что я не использую функцию **GetModuleHandle** , для получения hInstance. Вместо этого я пользуюсь константой **PROGRAM_IMAGE_BASE** и в Release линковщику указан ключ **/FIXED**. На всех известных мне платформах Windows приложения без проблем могут загрузиться по адресу **PROGRAM_IMAGE_BASE EQU 400000h**.

6 Что дальше?

В следующей статье «Записки Дzенствующего» мы поговорим о макросах, навсегда истребим проблему недоговорок документации по ним, и окончательно разберёмся со строками и головной болью UNICODE.

Макросы и директивы компилятора FASM

Автор: Dreamer2

Дата: 2003-01-02

Макрокоманды

С помощью макрокоманд Вы можете создавать ваши собственные комплексные инструкции, сокращенно называемые макросами, используя которые можно существенно упростить процесс программирования. В самой простой форме это похоже на определение переменной.

Например, определение следующего макроса позволяет сократить выражение test al,0xFF инструкцией tst:

```
Macro tst {test al, 0xFF}
```

После ключевого слова macro идет имя макроса и его содержимое внутри фигурных скобок {}. Вы можете использовать инструкцию tst в любом месте после её определения и она будет скомпилирована как test al,0xFF. Определение константы tst той величины дало бы тот же эффект, но разница в том, что имя макроса считается мнемоникой инструкции. То есть, макросы заменяются соответствующим кодом раньше, чем символические константы будут заменены их величинами. Так, если Вы определяете макрос и символическую константу с одним именем, и используете это название как мнемонику инструкции, она будет заменена содержанием макроса, но и будет заменена величиной, если символическая константа используется где-нибудь в операндах.

Макросы могут состоять из нескольких строк, так как начало и конец макроса не обязаны быть на той же самой строке что и директива macro. Например:

```
macro stos0
 {
    xor al,al
    stosb
 }
```

При использовании макрос stos0 будет заменен этими двумя инструкциями.

Подобно инструкциям, которые нуждаются в некотором числе параметров, макрос может быть определен, чтобы использовать необходимое число аргументов отделенных запятыми. Имена необходимых аргументов должны следовать за названием макроса на строке с macro и должны быть отделены друг от друга запятыми, если их больше одного. При использовании имени аргумента внутри макроса, оно будет заменено соответствующей величиной, полученной при использовании макроса. Вот пример макроса, который сделает выравнивание данных для вывода двоичном режиме:

```
Macro align value {rb (value-1)-($+value-1) mod value}
```

Когда инструкция align 4 будут найдена после определения этого макроса, она будет заменена его содержанием, и value станет равно 4, так что в результате мы получим rb (4-1) - ($+4-1) mod 4.

Если в макросе использовать инструкцию с тем же самым именем, будет использовано предыдущее значение этого имени. Таким образом, можно сделать полезное определение макроса в макросе, например:

```
macro mov op1,op2
 {
  if op1 in <ds,es,fs,gs,ss> & op2 in <cs,ds,es,fs,gs,ss>
    push  op2
    pop   op1
  else
    mov   op1,op2
  end if
```

```
 }
```

Этот макрос расширил возможности инструкции mov, при его использовании оба аргумента могут быть сегментными регистрами. К примеру mov ds,es будут скомпилированы как push es и pop ds. В остальных случаях будет использована стандартная инструкция mov. Следующий макрос еще более расширяет возможности инструкции mov, используя предыдущий макрос:

```
macro mov op1,op2,op3
 {
  if arg3 eq
    mov   op1,op2
  else
    mov   op1,op2
    mov   op2,op3
  end if
 }
```

Теперь mov может иметь три аргумента, но если опустить третий аргумент, будет использовано только два, потому что когда макросу дают меньше аргументов чем задано, недостающие аргументы будут иметь пустые величины. Когда даны все три аргумента, этот макрос станет двумя макросами предыдущего определения, так что mov es,ds,dx будет скомпилирована как push ds,pop es и mov ds,dx.

Директива purge позволяет удалить последнее определение макроса. Она используется с одним или более имен макросов разделенных запятыми. Если такого макроса не было определено, никаких ошибок не произойдет. Например, после использования mov расширенного макросами, определенными выше, Вы можете удалить синтаксис с тремя аргументами, используя директиву purge mov. Следующий purge mov удалит также синтаксис для сегментных регистров, но далее такие директивы ничего не сделают.

Если после директивы macro Вы задаете некоторую группу имен аргументов в квадратных скобках, это позволит использовать большее количество параметров для этой группы аргументов при использовании этого макроса. Любой аргумент, использованный после последнего аргумента такой группы, начнет новую группу и станет её первым аргументом. Именно поэтому после закрытия квадратной скобки нельзя определять аргументов. Содержание макроса будет обработано для каждой такой группы аргументов отдельно. Самый простой пример с одним аргументом в квадратных скобках:

```
macro stoschar [char]
 {
    mov al,char
    stosb
 }
```

Этот макрос принимает неограниченное количество аргументов, и каждый будет отдельно обработан с этими двумя инструкциями. Для примера stoschar 1,2,3 будет скомпилирован как следующие инструкции:

```
mov al,1
stosb
mov al,2
stosb
mov al,3
stosb
```

Есть некоторые специальные директивы, располагаемые только в определениях макросов. Директива local определяет локальные имена, которые будут заменены новыми значениями каждый раз, когда используется макрос. Она должна сопровождаться именами, отделенными запятыми. Эта директива обычно нужна для констант или меток которые макрос определяет и использует внутри себя. Например:

```
macro movstr
```

```
 {
    local move
  move:
    lodsb
    stosb
    test al,al
    jnz move
 }
```

При использовании этого макроса метка move всегда имеет свое уникальное значение и повторение этого макроса не вызовет ошибок.

Директивы forward, reverse и common делят макрос на блоки, каждый обрабатывается после того, как обработка предыдущих закончена. Они отличаются по поведению, только если макрос позволяет множественные группы аргументов. Блок инструкций, что следует за директивой forward, будет обработан для каждой группы аргументов с первого до последнего - наподобие обычному блоку (не заданному в соответствии с любой из этих директив). Блок, который следует за директивой reverse, будет обработан для каждой группы аргумента в обратном порядке - от последнего до первого. Блок, который следует за директивой common, будет обработан только однажды, сразу для всех групп аргументов. Значение local, определенное в одном из блоков доступно во всех следующих блоках при обработке той же самой группы аргументов где оно было определено, когда оно определено в блоке common, оно доступно во всех следующих блоках независимо какая группа аргументов обрабатывается.

Вот пример макроса, который создаст таблицу адресов к строкам, заданных этими строками:

```
macro strtbl name,[string]
 {
  common
    label name dword
  forward
    local label
    dd label
  forward
    label db string,0
 }
```

Первый аргумент, переданный этому макросу, станет меткой для таблицы адресов, следующие аргументы должны быть строками. Первый блок обрабатывается только один раз и определяет метку, второй блок для каждой строки объявляет ее локальное имя и определяет запись таблицы, содержащую адрес той строки. Третий блок определяет данные каждой строки с соответствующей меткой.

Директива, начинающая блок в макросе может сопровождаться первой инструкцией этого блока на той же самой строке, как показано в следующем примере:

```
macro stdcall proc,[arg]
 {
  reverse push arg
  common call proc
 }
```

Этот макрос может использоваться для вызова процедур, использующих соглашение STDCALL, где аргументы сохраняются на стек в обратном порядке. Для примера stdcall foo, 1,2,3 будет собран как:

```
push 3
push 2
push 1
call foo
```

Если некоторое имя в макросе имеет множественные параметры (одним из аргументов, огороженных в квадратные скобки или локальным именем, определенным в блоке, следующем за директивой forward или reverse) и используется в блоке, следующем за директивой common, это

имя будет заменено всеми ее величинами, отделенными запятыми. Например следующий макрос передаст все дополнительные аргументы предварительно заданному макросу stdcall:

```
macro invoke proc,[arg]
 { common stdcall [proc],arg }
```

Это может использоваться, чтобы вызывать косвенно (по указателю, находящемуся в памяти) процедуру, используя соглашение STDCALL.

Внутри макроса существует также специальный оператор #. Этот оператор связывает два имени в одно. Это может быть полезно, потому что это делается после того, как аргументы и локальные имена меняются на их настоящие значения. Следующий макрос произведет условный переход согласно аргументу cond:

```
macro jif op1,cond,op2,label
 {
    cmp op1,op2
    j#cond label
 }
```

К примеру jif ax,ae,10h,exit будет скомпилировано как cmp ax,10h и jae exit.

Чтобы сделать макрос, ведущий себя в зависимости от типа аргумента, когда аргументы - строки или нет, Вы может использовать факт, что ассемблер отличает напрямую указанные строки от указанных строк в численных выражениях, но не отличает численное выражение, которому предшествуют знак + от того же самого выражения без знака. Так строка, которой предшествуют + со знаком будут обрабатывать как численное выражение и не будет символически равен той же самой строке без любого знака, в то время как любая другая величина будет символически равна тому же самому выражению, которому предшествуют + знак. Вот пример макроса, использующий эту особенность:

```
macro message arg
 {
  if arg eq +arg
    mov   dx,arg
  else
    local str
    jmp   @f
    str   db arg,0Dh,0Ah,24h
    @@:
    mov   dx,str
  end if
    mov   ah,9
    int   21h
 }
```

Этот макрос показывает сообщения в ДОС программах. Когда аргумент этого макроса - некоторая метка, отображается строка с этого адреса, но когда аргумент - указанная строка, создается код правильно обрабатывающий эту строку.

Структуры

Директива struc - специальный вариант макро-директивы, которая используется, чтобы задавать структуры данных. Макрос, заданный директивой struc должен быть предварительно задан меткой (как при определении данных). Эта метка будет также содержаться в начале каждого имени, начинающегося с точки в содержании макроса.

Макрос, определенный с помощью директивы struc может носить тоже имя, что и макрос, определенный с помощью директивы macro. Макрос структуры не предотвращает обработку обычного макроса, когда перед ним нет метки и наоборот.

Все правила относительно стандартных макрокоманд относятся и к макросам структур.

Вот пример структуры:

```
struc point x,y
 {
    .x dw x
    .y dw y
 }
```

Например my point 7,11 определит структуру, с меткой my, содержащую две переменные: my.x с величиной 7 и my.y с величиной 11.

Следующий пример показывает, как расширить директиву db возможностью вычислить размер определенных данных:

```
struc db [data]
 {
  common
    label .data byte
    db data
    .size = $-.data
 }
```

С этим макросом msg db 'Hello!',13,10 определит также константу msg.size, равную размеру определенных данных в байтах и также дополнительную метку msg.data, которая будет указывать на данные типа байт.

Определение структур данных, к которым обращаются, используя регистры или абсолютные значения может быть сделано через директиву virtual в макросе структуры (см.??).

Директивы формата

Директива format, сопровождаемая идентификатором формата, позволяет выбирать формат выходного файла. Эта директива должна находится в начале исходного файла. Формат по умолчанию - плоский двоичный файл, также он может быть выбран командой format binary.

Директивы use16 и use32 вынуждают ассемблер компилировать 16 или 32-разрядный код, опуская установку по умолчанию для выбранного выходного формата.

Директива org устанавливает адрес, в котором как ожидается, будет работать код. Должна сопровождаться адресом.

Ниже описаны различные форматы с директивами, специфичными для этих форматов.

Формат MZ (MZ executable)

Для выбора выходного формата MZ, используйте директиву format MZ. Для этого формата по умолчанию создается 16 битный код.

Директива segment определяет новый сегмент, она должна следовать за меткой, чье значение будет именем сегмента, также можно задать use16, или use32 что определит, будет ли код в сегменте 16 или 32битным. Начало сегмента выравнивается по параграфам (16 байт). Все метки, определенные после этого будут иметь величины относительно начала сегмента.

Директива entry устанавливает точку входа для MZ файла, она должна сопровождаться дальним адресом (название сегмента, двоеточие и смещение в сегменте) точки входа.

Директива stack устанавливает стек для MZ файла. Она может сопровождаться численным выражением, определяющим размер стека, который будет создан автоматически или дальним адресом начальной кадра стека, если Вы хотите установить стек вручную. Когда стек не задан, он создается размером 4096 байт.

Директива heap должна сопровождаться 16 битной величиной, определяющей максимальный размер дополнительной кучи в параграфах (это куча в дополнение к стеку и неопределенным данным). Используйте heap 0 чтобы разместить только действительно необходимую программе память. Размер кучи по умолчанию - 65535.

Формат PE (Portable Executable)

Чтобы выбрать формат PE, используйте директиву format PE. Она может сопровождаться дополнительными типами формата: console, GUI или native оператор выбирает целевую подсистему (значение с плавающей запятой определяет версию подсистемы), DLL создает файл библиотеки. Потом может идти оператор at и численное выражение, определяющее, базу PE образа и затем произвольно on оператор, сопровождаемый строкой в кавычках, содержащей имя файла содержащего MZ заглушку для PE программы (если указанный файл не формата MZ он берется как обычный двоичный файл и конвертируется в MZ). Установка кода по умолчанию для этого формата - 32 бита.

Директива section определяет новый сегмент, за ней должна идти строка в кавычках, определяющая название сегмента за которым могут следовать один или более флагов сегментов. Возможные флаги: code, data, readable, writeable, executable, shareable и discardable. Среди флагов также могут быть определены специальные PE идентификаторы данных, чтобы создать сегмент специальные данных, доступные идентификаторы export, import, resource и fixups. Когда выбрано определение данных fixups, они создаются автоматически и не требуют дополнительной информации. Начало сегмента выравнивается по страницам (4096 байт).

Директива entry устанавливает точку входа для Portable Executable, необходимо значение точки.

Директива stack устанавливает размер стека для PE, за директивой идет значение, резервируемое под стек, произвольная величина стека может следовать через запятую. Когда стек не задан, он создается размером 4096 байт.

Директива heap задает размер кучи для PE, за директивой идет значение, резервируемое под кучу, произвольная величина стека может следовать через запятую. Когда куча не определена, она создается по умолчанию 65536 байт, когда размер кучи совершенно не установлен, она устанавливается в 0.

Директива data начинает определение специальных PE данных, она должна сопровождаться одним из идентификаторов данных (export, imort, resource или fixups) или номером записи данных в PE заголовке. Данные должны быть определены на следующих строках, заканчивающихся директивой end data. Когда выбрано определение данных fixups, они создаются автоматически и не требуют дополнительной информации.

Формат COFF (Common Object File Format)

Чтобы выбрать формат COFF, используйте format COFF или format MS COFF, если Вы хотите создать простой или Microsoft COFF файл. В этом формате код по умолчанию 32 - бит.

Директива section определяет новый сегмент, за ней должна идти строка в кавычках, определяющая название сегмента за которым могут следовать один или более флагов сегментов. Возможные флаги: code и data для обоих COFF вариантов, readable, writeable, executable, shareable и discardable только для Microsoft COFF. Начало сегмента выравнивается по страницам (4096 байт).

Директива extrn определяет внешний символ, она должна сопровождаться названием символа и опционально оператором размера, определяющим размер данных, маркированных этим символом.

Команда public объявляет существующий символ как public, он должен сопровождаться названием символа.

Руководство по проектированию макросов в MASM32

Автор: Edmond / HI-TECH

Дата: 2003-08-25

> Пойми в Хаосе Разное, и стань человеком.
> Осознай Единое в Различном – и будь Богом.
> Автор

I. От автора

В этом руководстве раскрывается тема создания, использования (а главное – проектирования) макросов и макрофункций в проектах на **MASM32**.

Что не так важно в **ЯВУ** , то очень важно в программировании на ассемблере. Если выстроить по приоритетам недостатки программирования на ассемблере, то первым недостатком будет не объём строк написанного кода (как нестранно), а отсутствие средств, обеспечивающих хороший стиль написания кода.

Что значит стиль? А что значит плохой или хороший? Это можно быстро понять на простом примере.

Допустим, у вас есть процедура объёмом на несколько экранов. Вы её написали месяц назад, а теперь вам нужно несколько изменить её поведение. Для того, чтобы сделать это, вам необходимо:

1. Вспомнить её алгоритм (если забыли)
2. Вспомнить особенности реализации (у вас должны быть комментарии)
3. Вспомнить какой участок кода, чем занимается.

Если код процедуры был прооптимизирован, вероятней всего вам захочется, чтобы после модификации он остался настолько же оптимальным, а поэтому вы должны вспомнить все тонкости кода, или только того участка, который подлежит модификации.

А это не так то просто, даже если исходник написан вами, в вашем неповторимом стиле.

Если этот стиль будет хорошим, вы потратите меньшее время, если бы стиль был бы плохим.

Хороший стиль программирования – это сэкономленное время, которое можно потратить на понимание, модификацию или, как это называют, сопровождение кода. Стиль программирования – это архитектура исходного кода – не только его внешнее оформление, но и использование констант, разбиения кода на функции или процедуры, способы вызова функций и процедур, согласованность структур, их потенциал к расширению, гибкость алгоритмов и многое другое. Стиль программирования сложно отделить от архитектуры самой программы, так как хорошо спроектированная программа не может иметь плохого стиля программирования.

Конечно же, на **ЯВУ** легче писать качественно оформленные программы, хотя бы, потому что **ЯВУ** уже имеет готовые средства выражения, и шаблоны мышления.

Что такое шаблоны мышления? Всё чем вы так активно пользуетесь: - типы - функции - классы - массивы - указатели на типы - пространства имён - шаблоны (C++) Всё это направляет ваше понимание программирования как пространства сотканного из таких абстракций.

Недавно я прочёл следующую мысль на форуме WASM.RU:

Да, зачем вы пишите программы на asm под Win32, лучше уже писать под DOS, там хоть нет этого бесконечно однообразного кода создания окон и обработки сообщений.

Такое заявление говорит, что программист не желает писать проекты более чем на 6 000 строк (или 3 000 :)). Вместо того чтобы извлечь великую выгоду из единообразия кода, мы ругаем его. А ведь это первый звонок к автоматизации программирования.

Неужели программирование asm может быть похоже на Delphi (ох как его не любят некоторые)? Снова интегрированная среда? Конечно!!! (Жаль, её всё-таки нет!) Но это не значит, что она играет отрицательную роль. Хотя о средствах автоматизации и их создании мы поговорим в другой работе.

Ассемблер не определяет шаблонов мышления, и практически не имеет средств выражения каких либо шаблонов (из-за чего автор пользуется им).

Очень сложно назвать директиву proc средством выражение процедурной модели программирования.

Однако я могу ручаться, если вы научитесь писать качественно стилизированные программы на ассемблере, то на ЯВУ… :).

Об искусстве стилизации или проектировании архитектуры написано слишком мало, а рассказать хотелось бы слишком много. Только нельзя объять необъятное, и потому цель этого руководства рассказать об использовании макросов в **MASM32** , а также о том, как их можно либо нужно использовать, чтобы более качественно стилизировать код.

I.1 Для тех, кто впервые...

Если вы ещё не работали с макросами, или работали, но очень мало, я спешу признаться, что это руководство не предназначалось для начинающих. Но благодаря рекомендациям и советам **TheSvin/HI-TECH** я решился добавить в него вырезки и упражнения, которые позволят вам быстро войти во вкус макромира **MASM32**. Если же вы уже имеете дело с макросами, тогда это руководство укрепит ваши знания и представления по данной теме.

Для исследования макромира **MASM** мы воспользуемся директивой **echo** , которая позволит вывести нам на экран то, что творится в препроцессоре **MASM**. Очень удобно, а главное наглядно. Я уверен, что вы быстро усвоите этот материал.

I.2. Примечания (обо всём понемногу)

В данной работе я часто пишу: «Препроцессор ML». Кто-то из умников (или просто жаждущих подловить «на горячем») воскликнет: «Да какой же такой **ML.EXE** – препроцессор? Наглая ложь». На всякий случай оговорю, что здесь имеется ввиду не утверждение «ML – препроцессор», а именование его подсистемы – препроцессор.

Всё, что есть в этом руководстве не взято с потолка, и не является вымышленным. Весь код проверен, и работает именно так как описано, если только автор случайно не ошибся, что так же случается.

Многое из того, что написано в этом руководстве недокументированно (или плохо документировано) в официальном. Поэтому вы всегда должны помнить, что если в следующих

версиях **ML** (например, 8.0) что-то не будет работать, никто не виноват.

Если вы думаете, что я дизассемблировал **ML.EXE** – то ошибаетесь. Алгоритмы работы, приведённые здесь, получены логическим путём на основе знаний работы компиляторов, а поэтому их не следует воспринимать как истинные. Важна сама логика работы, понимание которой, поможет вам безболезненно использовать макро, допуская меньшее количество ошибок.

На самом деле **MASM** очень плохо документирован, и видно MS совсем не относится к нему как к продукту (что вполне очевидно). Хотя уже в **MSDN 2002** был внесён раздел *MASM Reference* , и всё равно – вы не найдёте лучше описания чем в **MASM32 by Hutch**.

Когда вы прочтете, то воскликните: « *Да, зачем мне такой ML?* ». Есть **NASM** и **FASM** – главная надежда мира ассемблерщиков. Однако и теперь **ML** всё ещё выигрывает у них по удобству эксплуатации, большей частью видимо благодаря Хатчу, и многим замечательным людям, поддерживающим **MASM32**. Кто знает, может после этой статьи кто-то воскликнет: « *Я знаю, какой должен быть компилятор мечты асмовцев!* ». И напишет новый компилятор. (Автор шутит ?)

Уверен, что программисты из **MS** вряд ли прочтут эту статью (они плохо знакомы с русским), и оно к лучшему. Возможно, такая статья могла бы их огорчить, а я не люблю портить настроение людям, трудами которых пользуюсь. (Снова шутит, только про что?)

И наконец-то мне в свою очередь хочется порадоваться, что многие вопросы по макросам в **MASM** закрыты на долгое время, во всяком случае, для русскоязычной аудитории. (Шутит, или нет? Гм…)

I.3. Особенности терминологии

Терминология этой статьи различается от терминологии принятой в **MASM**.

В частности автором было предложено называть:

```
MacroConstant    EQU  123          ;; Числовая макроконстанта
MacroVar         =  123           ;; Числовая макропеременная
MacroText        EQU  <string>    ;; строковая макропеременная
MacroText        TEXTEQU <string> ;; строковая макропеременная
```

В MASM:

```
MacroConstant    EQU  123          ;; numeric equates
MacroVar         =  123            ;; numeric equates
MacroText         EQU  <string>    ;; text macro
MacroText        TEXTEQU <string>  ;; text macro
```

Можно было бы попросту выбрать терминологию MASM, однако последняя не позволяет объяснять материал систематически. То есть все четыре вида выражений – по сути, являются переменными или константами. Однако в терминологии MASM два последних определения называются текстовыми макро, подчёркивая их связь с макросами.

Если пойти этим путём, то тогда и первые два определения – являются упрощёнными определениями макро. Если разработчики желали подчеркнуть, что сама суть внутренней реализации ML представляет текстовые макросы как макро, то тогда не ясны те все эффекты функциональности, обсуждаемые в этой статье.

Что имеет ввиду автор? Посмотрите что такое макроопределение – это некий текст, который как бы «вставляется» препроцессором в исходный текст программы в месте вызова макро. А что такое в терминологии MASM numeric equates, или text macro – это некоторые переменные, значения которых «подставляются» в исходный текст программы во время компиляции вместо имён этих переменных. Таким образом, можно сказать, что определения представленные выше – макро, но в упрощённом их виде.

Этот спор не решаем, что не так и важно. Поэтому автор отдаёт предпочтение двум терминам для «text macro»: «текстовой макро» и «строковая макропеременная».

Понятие: «numeric equates» является общим для первых двух случаев, и разрывает смысловую связь с двумя последними определениями. Поэтому я пользуюсь своим вариантом терминологии, который подчёркивает, что определения:

```
MacroConstant    EQU  123       ;; Числовая макроконстанта
MacroVar         =  123         ;; Числовая макропеременная
```

являются подобными макро. А, кроме того, первое из низ – константа, а второе – переменная.

I.4. Благодарности

Не могу не написать этот пункт, ибо не только автору обязана эта статья.

Она обязана замечательной версии Win98 с инсталляцией от 2000, которая отформатировала весь мой винчестер, и унесла в небытие первый вариант настоящей статьи. :)

Не малая заслуга в вопросе терминологии **MASM** , и его разрешении принадлежит **Four-F** , который как он сам мне признался, съел на макросах собаку, при чём без соли :).

Когда я думаю, чтобы было бы без самого Маниакального редактора в Inet, **CyberManiaca** , то понимаю: без его правок мои статьи приводили бы в ужас, и лишали разума всех морально неустойчивых читателей. CyberManiac: «Только такой замечательный безумец как ты может выдержать ЭТО!!!» :).

FatMoon , **Rustam** , **The Svin** – вы дали понять мне то, что такая статья действительно нужна, и это, наверное, самое главное. Вряд ли я бы так долго работал над ней, если бы меня никто не подталкивал.

Всех кого я забыл поблагодарить здесь, и кого не забыл, жду в условном месте в условное время для раздачи благодарностей.

С уважением, **Edmond/HI-TECH**

II. Лень – двигатель Макро

Когда говорят, что лень – это двигатель прогресса, видимо лицемерят или преувеличивают. Скорее это нежелание выполнять одну и ту же работу очень часто. Первая парадигма к созданию макро звучит так:

Если есть что-то похожее, что нужно делать очень часто, я могу оформить это как макроопределение.

Ассемблер, дающий программисту полную свободу в использовании методик программирования, совершенно лишает его средств для выражения этих методик. Например, **ООП**. В **MASM32** нет классов, конструкторов и других механизмов, поддерживающих эту абстракцию. Зато вместо **ООП** Вы можете придумать множество других методик и абстракций (как, например модель серверов).

Та или иная методика программирования обязательно состоит из каких-либо компонентов, которые являются подобными друг другу. Например, следующие макро очень любимы в примерах пакета **MASM32** :

```
m2m MACRO M1, M2
```

```
    push M2
    pop  M1
  ENDM

  return MACRO arg
    mov eax, arg
    ret
  ENDM
```

Предположим, что кому-то так надоело писать:

```
push переменная2
pop  переменная1
```

И он решил придумать макро для этого. Эта пара команд осуществляет пересылку данных из одной ячейки памяти в другую. То есть теперь в программе, когда вы захотите написать *push/pop* , вы можете заменить это некой *m2m операнд1, операнд2*. Посмотрите на эти два участка кода:

```
mov wc.cbWndExtra,      NULL
m2m wc.hInstance,       hInst
mov wc.hbrBackground,   COLOR_BTNFACE+1
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
mov wc.cbWndExtra,      NULL
push hInst
pop wc.hInstance,
mov wc.hbrBackground,   COLOR_BTNFACE+1
```

Первый вариант не только занимает меньше строк (что тоже важно), но и намного понятнее, чем push/pop (если вы, знаете что такое m2m). Конечно, если говорить о макро m2m, то он имеет и очень важный недостаток.

Мощь макро была бы сказочной, если бы MASM умел следить за кодом, или ему можно было бы указать, что, например, сейчас регистр ebx == 0, или eax никем не используется. Хотя мы попробуем достичь подобного эффекта самостоятельно.

Этот недостаток потеря контроля над оптимальностью кода. Например, более быстрыми, по сравнению с парой команд *push/pop* , являются **mov eax,…** Употребляя макро **m2m** , вы получаете худший код, если стремитесь оптимизировать по скорости. И здесь есть две стороны проектирования кода:

1. Эффективность кода
2. Совершенство стилистики

Используя макро **m2m** , вы повышаете уровень стилистики, так как сокращаете время на понимание исходного кода (вами же или другим программистом). Однако с другой стороны вы теряете эффективность.

Это одна из вечных задач архитектора – найти баланс между эффективностью в коде и совершенством стилистики.

Другая парадигма использования макро звучит так:

Если, объединяя что-то в одно целое, я улучшаю стиль кода – это можно сделать в виде макроопределения.

Эта парадигма отличается от предыдущей тем, что создание макроопределения обуславливается только улучшением стилизации кода, и не имеет особой практической ценности. Например, я объявил такие макро для определения кода начала и конца в главном модуле программы:

```
$$$WIN32START  macro
PUBLIC l$_ExitProgram
_start:
 xor ebx,ebx
   endm

$$$WIN32END  macro
```

```
l$_ExitProgram:
 push $$$__null
 call ExitProcess
   end _start

   endm
```

В этих макро нет по сути никакой пользы, кроме эстетической. Зато, глядя на код, можно сразу понять, что это не что иное, как начало программы нечто вроде main() в C++.

И последняя парадигма использования макро:

Если ты используешь технологию программирования – попытайся заключить её в комплекс макроопределений. Например, для модульного программирования нужно создать макросы для определения модуля, его частей, кода и данных.

Наиболее важная часть использования макро. Посмотрите, например, файл **Objects.INC** из пакета **MASM32** в папке **oop** (*NaN & Thomas*).
Мы начнём создание первых макро со следующей задачи.

Наверное, вы знаете, что EXE приложения всегда могут загружаться по адресу равному:

```
PROGRAM_IMAGE_BASE EQU 400000h
```

Во-первых, это даёт нам право убрать из приложения всю **Relock** секцию, тем самым, уменьшив объём образа (если эта секция нужна для систем плагинов, её можно держать отдельно).

Во-вторых мы можем более не вызывать функцию **GetModuleHandle** , что так же полезно для нас. Использование константы **PROGRAM_IMAGE_BASE** очень удобно. Однако, что будет значить это удобство, если всё-таки **PROGRAM_IMAGE_BASE** не определено? Это будет означать, что мы обязаны переписать весь код. А если этого кода много?

Определённо об этом нужно позаботится заранее. Давайте же будем решать эту проблему при помощи макро! Для этого нам станут необходимыми некоторые знания о том, как обрабатывается макро, и что это такое.

III. Макромир MASM

Макрос представляет собой именованный участок исходного текста программы, который обрабатывается компилятором каждый раз в том месте, где вызывается макрос.

-= Подробней =-

Пример: Создайте небольшой модуль с именем macro.asm. И напишите в нём несколько строчек Так действует директива echo. С помощью неё можно подсмотреть значения переменных. Если вы не знаете, как это работает, не волнуйтесь, обо всём будет рассказано. А пока несколько экспериментов: Напишите: Взгляните на код программы под отладчиком. Что у вас получилось? Что будет, если вы измените текст внутри макроопределения? Теперь напишите: Каким будет вывод на экран во время компиляции?

```
                              .386
                              .data
                              .code
echo Hello!!!
echo Ты должен увидеть во время компиляции

end

Mycount = 1
%echo @CatStr(%Mycount)
```

```
MyMacro  macro reg

              dec    reg
  endm

  .code

              mov eax,5
MyMacro eax
MyMacro ebx

MyVar = 1

MyMacro  macro

MyVar = MyVar+1
%echo MyVar = @CatStr(%MyVar)

  endm

MyMacro
MyMacro
MyMacro
MyMacro
```

С этого момента вам придётся различать в ассемблере **ML** две подсистемы: препроцессор и компилятор. Если компилятор переводит код мнемоник в машинный код, вычисляет значения меток и смещений, то препроцессор занимается вычислением выражений этапа компиляции, и что самое важное – процессом раскрытия макросов.

Подобно многим объектам мира программирования макро имеет два состояния в исходном тексте: определение, и использование.

Таким образом, мы будем иметь дело с определением макроса (макроопределением), и его вызовом (использованием макроса).
Макроопределением называется любой текст, заключённый между ключевыми словами:

```
MacroName  macro paramlist
макроопределение
   endm
```

При каждом вызове макро, а именно:

```
…
MacroName
или
  mov eax, MacroName()
…
```

Будет анализироваться и исполнятся текст, заключённый в макро. Именно так это и реализовано в **ML**. Поскольку текст в макроопределении не компилируется, то естественно, вы не увидите сообщений об ошибке, даже если с точки зрения ассемблера эта ошибка будет в теле макроопределения. Однако ошибка появится при попытке вызова макроопределения, её могут выдать вам, либо сам препроцессор, либо компилятор, если текст, сгенерированный препроцессором является неверным с точки зрения компилятора.

Каждый раз, когда препроцессор встречает макроопределение, он помещает его имя в специальную таблицу, и копирует его тело к себе в память (это не обязательно именно так, но вам должна быть понятна суть). Встретив макроопределение, препроцессор не проверяет, а есть ли макро с таким же именем. Это значит, что макро можно переопределять.

```
MyMacro         macro
echo Это макро 1
                endm

MyMacro         macro
```

```
echo Это макро 2
               endm

MyMacro
```

Вы можете самостоятельно удалять макроопределения, из памяти препроцессора используя директиву **PURGE** :

```
PURGE macroname
```

Эта директива удаляет тело макроопределения, однако не удаляет имя макро из таблицы имён. Таким образом, в данном случае:

```
MyMacro    macro
 mov     eax,ebx
           endm

PURGE MyMacro


;; После этой директивы, MyMacro эквивалентен:
;; MyMacro macro
;;         endm
;; Определению пустого макро.


MyMacro    ;; Ничего не произойдёт.
```

Разработчики **ML** задумывали эту директиву для разрешения конфликтов между файлами с множеством макросов, однако мне совершенно не ясно как ей можно воспользоваться. Если вы хотите получить эффект «удаления» макро, лучше применять следующий метод:

```
<Имя макро, который нужно «удалить»> macro
.ERR <Попытка вызова макро, который не существует>
                                    endm
```

В этом случае при попытке воспользоваться таким макро, компилятор выдаст ошибку, и вы будете проинформированы о его вызове, что намного лучше неведения. Поэтому просто запомните: «Не нужно использовать директиву PURGE».

Конечно же, использование макро не было бы столь полезным, если бы макро не имел формальных параметров. При вызове макро, препроцессор заменяет все имена формальных параметров их непосредственными значениями в теле макроопределения. Список формальных параметров разделяется запятой, и может иметь вид:

```
MyMacro macro param0, param1:REQ, param2 := <0>,param3:VARARG
```

Здесь:
Param0 – пример определения параметра.
Param1:REQ – ключевое слово **REQ** указывает на то, что этот параметр обязательный. То есть, если он не будет указан, вы получите ошибку этапа компиляции.
Param2:= <0> – пример параметра, который имеет значение по умолчанию. То есть если этот параметр не будет указан при вызове макро, он будет равен этому значению.

```
Заметьте, что при вызове макро параметр может быть не определён:
              MyMacro param1,,param3
              Значение второго параметра неопределенно.
```

Param3:vararg – становится именем параметра, который воспринимает всё остальное как строку. При этом запятые между параметрами так же попадают в строку, а значит число параметров макроса в принципе неограниченно.

Ограничениям являются особенности архитектуры компилятора. Так, например, компилятор имеет ограничение на длину логической строки, которая равна 512 байтам.

Конечно же, после параметра с директивой **vararg** не возможно объявить другие параметры.

Обратите внимание, что если при определении формального параметра в макро нет директивы – он считается необязательным параметром. Более подробно о вызове макро и значении параметров я расскажу далее.

-= Подробней =-

Пример: Так что же происходит с формальными параметрами? Посмотрите, как работает препроцессор ML: 1. Препроцессор берёт текст внутри макро, и заменяет в нём все слова param1, param2, на их значения: « mov eax, var mov ebx, 123 » 2. Полученный текст вставляет на место вызова макро, и передаёт компилятору. Вот интересно, а что будет если:

```
        MyMacro     macro param1,param2

              mov    eax, param1
              mov    ebx, param2

                   endm

        MyMacro var, 123
```

```
MyMacro  macro param1,param2


           MyMacro2   macro  param1
               mov    eax, param1
               mov    ebx, param2
                      endm

          endm

MyMacro var, 123
```

Можно различать два вида макро – макропроцедуры и макрофункции.

В официальном руководстве MASM различается четыре основных вида макро. Text macros – текстовый макрос Macro procedures – макро-процедура Repeat blocks – блок повторения Macro functions – макро-функция Однако автор считает, что разделение макро на два вида – лучше систематизирует материал, и отражает суть темы.

Макрофункции в отличие от *макропроцедур* могут возвращать результат, и получают список формальных параметров в скобках, подобно функциям в **C**. Например:

```
          mov     eax,@GetModuleHandle()
```

Заметьте, что к макрофункции невозможно обратится как к макро, вы всегда должны заключать формальные параметры макрофункции между «**()** », иначе **MASM** не будет распознавать её как макрофункцию:

```
          mov     eax,@GetModuleHandle
  error A2148: invalid symbol type in expression
  : @GetModuleHandle
```

Препроцессор **MASM** анализирует текст макроопределения на наличие директивы **exitm** , и помечает макрос как макрофункцию.

Ключевое слово **exitm <retval>**, аналогично оператору **return** в **C++** , выполнение макро заканчивается, и возвращается необязательный параметр **retval**. Этот параметр – строка, которую должен вернуть макрос.

-= Внимание =-

Если в макро директива EXITM употребляется без параметров: EXITM То препроцессор считает, что это макропроцедура, а не макрофункция. Если в макроопределении есть два вида EXITM с

параметром и без, то ML выдаст ошибку о недопустимом использовании директивы EXITM. EXITM <> EXITM : error A2126: EXITM used inconsistently Это подчёркивает тот факт, что макрофункцией считается только макро, который возвращает значение (хотя бы пустое), а директива EXITM без параметров не возвращает никакого значения, что недопустимо в макрофункции.

Таким образом, окончательно будем считать, что макро, которые не возвращают значение – это макропроцедуры, а макро, которые возвращают значение (хотя бы пустую строку) – это макрофункции.

```
;#########################################################
@GetModuleHandle   macro
Invoke GetModuleHandle,0
     exitm
     endm
   .code
; Это макрофункция так нельзя
@GetModuleHandle ;;- ошибка
; Так можно
@GetModuleHandle()
;#########################################################
@GetModuleHandle   macro
Invoke GetModuleHandle,0
     endm
   .code
; Это макрос. Так правильно
@GetModuleHandle
; Так можно, но всё равно это вызывает ошибку ?
; warning A4006: too many arguments in macro call
@GetModuleHandle()
; Это макро, а не макрофункция так нельзя!!!
  mov eax,@GetModuleHandle
; И так нельзя
  mov eax,@GetModuleHandle()
;#########################################################
```

Что касается директивы endm, которая заканчивает каждое макроопределение, в руководстве написано, что при помощи неё так же можно указать возвращаемый параметр: endm <retvalue> Однако на практике это не так. ? Очень странно, хотя об этом чётко написано в руководстве.

Заметьте, что макропроцедура может быть вызвана только в начале строки:

```
@GetModuleHandle
;; Но не так:
mov eax,@MyMacro
```

Макрофункция может быть вызвана в любых выражениях:

```
;; Так:
mov eax,@GetModuleHandle()
;; И так:
@FunMacro()
;; И так:
@GetModuleHandle() EQU eax
```

III.1. Функционирование макросов

Чтобы строить макросы, важно понимать, как они работают, и как их обрабатывает **MASM**. Давайте рассмотрим типичный макро, и этапы его обработки.

```
MyMacro macro param1,param2,param3:VARARG
echo param1
echo param2
echo param3
      endm
```

```
MyMacro Параметр 1, Параметр 2, Параметр 3, Параметр 4
;; Вывод -=-=-=-=-=-=-=-=
Параметр 1
Параметр 2
Параметр 3,Параметр 4
```

1. Компилятор встречает лексему **MyMacro**

2. Он проверяет, содержится ли эта лексема в словаре ключевых слов

3. Если нет, то он проверяет, содержится ли эта лексема в списке макросов.

4. Если да, он передаёт текст, содержащийся в макро препроцессору. Препроцессор заменяет все вхождения формальных параметров в этом тексте на их значения. В данном случае мы имеем:

```
echo Параметр 1
echo Параметр 2
echo Параметр 3,Параметр 4
```

5. Препроцессор возвращает компилятору обработанный текст, который после компилируется.

Обратите внимание на пункт 4 и 5. Они ключевые. Очень часто при работе с макроопределениями появляются ошибки из-за неверного понимания порядка генерирования макро текста. Например:

```
PROGRAM_IMAGE_BASE EQU 400000h
FunMacro macro
  exitm <Параметр 3,параметр 4>
      endm

MyMacro macro param1,param2,param3:VARARG
echo param1
echo param2
echo param3
      endm

MyMacro PROGRAM_IMAGE_BASE, FunMacro(),Параметр 5
```

А теперь самостоятельно опишите порядок действий компилятора при вызове этого макро. Запишите его себе куда-нибудь, так чтобы сравнить, и смотрите на вывод:

```
PROGRAM_IMAGE_BASE
Параметр 3, Параметр 4
Параметр 5
```

Прежде чем объяснять действительный порядок, я оговорюсь, что директива **echo** никогда не обрабатывает определённые константы, такие как **PROGRAM_IMAGE_BASE**.

Это утверждение справедливо даже тогда, когда перед директивой **echo** стоит оператор **%** , который может раскрывать только текстовые макроопределения. То есть выражение:

```
echo FunMacro()
```

Даст результат:

```
FunMacro()
```

Теперь, когда мы немного порассуждали можно привести тот текст, который генерируется из макро:

```
echo PROGRAM_IMAGE_BASE
echo Параметр 3, Параметр 4
echo Параметр 5
```

Это означает следующее:

1. При вызове макро, значение формальных параметров воспринимается как текст, и передаётся в макро как строка.
2. Исключение составляют лишь макрофункции, результат выполнения которых вычисляется и присваивается значению параметра.

Специальный оператор **%** заставляет ассемблер вычислять текстовую строку, следующую за ним, и только потом подставлять в правое выражение. Например, если мы перепишем макровызов так:

```
MyMacro %PROGRAM_IMAGE_BASE, FunMacro,Параметр 5
```

То получим вывод:

```
4194304   ;; Значение PROGRAM_IMAGE_BASE
Параметр 3, Параметр 4
Параметр 5
```

Давайте рассмотрим ещё один пример, который хорошо показывает, как работает макро. Например, вы определили макропроцедуру (именно его, а не макрофункцию). То когда вы пишите такое:

```
@Macro что-то, что придёт вам в голову [символ возврата каретки]
```

Что делает препроцессор **ML** :

1. Считывает всю строку до символа возврата каретки;

2. Смотрит, как вы определили параметры в макро;

3. Сканирует строку на наличие символа «**,** » или «**<** », «**>** »;

```
Вам может показаться странным, но препроцессору всё равно,
        какие символы идут во время вызова макро. То есть вы можете вызвать макро
        так:
        @MyMacro Привет, это кириллица в файле,\
 и ML не будет на неё ругаться
или
@MyMacro `!@#$%^&*(){}[]
            Посмотрите как СИльно (от буквы СИ)
        будет выглядеть макро в MASM:
        MyMacro{Это что C++?}
MyMacro[Нет, это MASM]
```

4. Назначает формальным параметрам (любого типа, кроме **VARARG**) макро участки строк, которые были определены разделителями запятыми (предварительно очистив от хвостовых и начальных пробелов, если только строка не была определена в угловых кавычках **< >**);

5. Если макро содержит формальный параметр типа **VARARG** , то **ML** сперва инициализирует значениями (согласно пункту 4) обычные формальные параметры, и только потом назначает параметру типа **VARARG** (который может быть только один в конце списка параметров) всю строку до конца.

-= Обратите внимание =-

Если вы пишите макровызов как @Macro Param1 , Param2 То значение параметров будут: param1 = «Param1» param2 = «Param2» Если вы хотите передать сами значения строк, то должны заключит их в угловые кавычки: @Macro < Param1 >,< Param2 >

6. Препроцессор разрешает все вызовы макрофункций, если они есть в лексемах параметра, и присваивает их результат соответствующему параметру. Если лексему в строке параметра предваряет символ **%** , то он вычисляет её значение до того, как передаст строку внутрь макро.

-= Обратите внимание =-

Благодаря именно такому порядку: 1. Разделение строки на макропараметры 2. Поиск и Вызов макрофункций в значениях макропараметров 3. Присвоение результатов соответствующему макропараметру в следующем случае: строка, возращаемая макрофункцией присваивается параметру param1, а не param2, param3

```
MyMacro  macro param1,param2,param3
echo param1
         endm
```

```
------------------------------------
FunMacro     macro param:VARARG
  exitm param
             endm


MyMacro FunMacro(param1, param2, param3)


OUT:
param1, param2, param3
------------------------------------
```

Теперь вы в состоянии объяснить следующую ситуацию:

```
MyMacro    macro
…
           endm


MyMacro()


Предупреждение при компиляции:
: warning A4006: too many arguments in macro call
```

Как нужно было бы изменить этот макро (именно макро, а не макрофункцию), чтобы предупреждение не выдавалось? А почему оно происходит?

Если вы с лёгкостью ответили на этот вопрос, значит, материал усвоен, иначе советую ещё раз прочитать его, и ответить на следующий вопрос.

Как должен понять компилятор следующий код:

```
MyMacro     macro param1
param1
            endm

MyMacro = 2
```

Естественно отвечать на этот вопрос вы должны без помощи компилятора (то есть проверить компиляцией). Если вы не можете ответить на этот вопрос, или неуверенны в верности ответа, я поменяю задание:

```
MyMacro     macro param1
echo param1
            endm

MyMacro = 2
```

Запустите его в **ML**. Если и теперь вы сомневаетесь – перечитайте этот пункт снова и снова, продолжая экспериментировать.

III.2. Определение макро переменных и строк

Я бы назвал следующее:

```
Param = 0
Constant    EQU 123
WASM      EQU <One Wonderful Wonderful ASM>
WASM_RU     TEXTEQU <http://www.wasm.ru>
```

макропеременными (с тем фактом, что переменная может иметь константный тип).

В терминологии **MASM:**

```
WASM       EQU <One Wonderful Wonderful ASM>
WASM_RU    TEXTEQU <http://www.wasm.ru>

;; Такие определения называются текстовыми макро.
;; В этой статье вы встретите два варианта определений
```

Потому что под термином «переменная» понимается:

```
var dd 123
```

Переменные являются частью программы, а макропеременные живут только на этапе компиляции. По сути, они есть более простым видом макроопределений, и поэтому их стоит понимать как специальные макро, которые так же раскрываются препроцессором.

Макропеременная может иметь только три типа – целочисленная макропеременная **INEGER4** (**dword**), целочисленная макроконстанта или текстовой макро (строковая макропеременная).

Автор считает значительным упущением отсутствия возможности определять тип макропеременной. Это очень сильно ограничивает возможности макропрепроцессора. Но что поделать.

При чём, в зависимости от вида определения макропеременной **ML** считает, что:

```
Param = 0          ;; Param – это целочисленная макропеременная
Constant   EQU 123 ;; Макроконстанта
;; Текстовой макро (Макропеременная строкового типа)
;; (Это не так в руководстве MASM)
Var        EQU     qwer
;; Текстовой макро (Макропеременная строкового типа)
WASM       EQU     <One Wonderful Wonderful ASM>
;; Текстовой макро (Макропеременная строкового типа)
WASM_RU    TEXTEQU <http://www.wasm.ru>
```

Как вы уже догадались, каждое макроопределение обладает своими свойствами и возможностями.

1. *Целочисленная макропеременная*. Имеет тип **INT** (**dword**). Может участвовать во всех арифметических выражениях **MASM**. Как переменная она может изменять своё значение.
2. *Макроконстанта* может иметь целочисленное значение. Её значение не может быть повторно изменено.
3. *Текстовой макро* может быть любой строкой не более 255 символов. Поскольку он имеет статус переменной, его значение может быть изменено.

А теперь подробнее. Если с *целочисленными макропеременными* в достаточной степени ясно. То с определениями **EQU** полный бардак.

Как и в случае с вызовами макро, автор попытается построить алгоритм анализа **EQU** выражений:

1. Анализируем правую часть. В анализе правой части препроцессор выделяет лексемы, которые классифицирует как числа, строки. Так, например, в выражении:

```
qqqq EQU 1234567890 string1 23456789012390 macrofun()
```

«1234567890» – это лексема число, а «string1» – это строка, «macrofun()» – это всё равно строка (а не макрофункция!!!).

-= Внимание =-

Именно по этому такое определение будет давать ошибку: qqqq EQU 156n7 : error A2048: nondigit in number

2. Если правая часть является верным определением числа в **MASM** , то есть 123 или 123h или 0101b – выполнить шаг три, иначе шаг четыре.

-= Внимание =-

Обратите внимание, что числа с плавающей запятой в этом случае считаются строкой. Такое поведение связано с внутренней организацией препроцессора ML, который просто «не понимает» чисел с плавающей запятой, и не умеет с ними работать. То есть тип макропеременной Float: будет не числовой, а строковой

```
Float EQU 1.2345
```

3. Если полученное число имеет значение, не превышающее диапазон значений для **dword** – это целочисленная макроконстанта.

-= Интересно =-

Если правая часть для EQU является верным числом более 25 символов, выдаётся ошибка: : error A2071: initializer magnitude too large for specified size При чём такая ошибка появляется даже в том случае, если выражение содержит другие символы через пробел: Это объясняется действиями в пункте 1, когда ML анализирует лексемы. Кроме того, если числовая лексема не соответствует правилам определения чисел в ML, то есть в середине числа появляется символ A-Z, либо другие символы, не входящие в разряд разделителей – то такая лексема порождает ошибку, даже если она содержит число большее dword диапазона.

```
qqqq EQU 12345678901234567890123390 dfdg
```

4. Иначе – это строковая макропеременная.

Теперь попробуйте самостоятельно определить тип макроопределения:

```
qqqq EQU 0x123234
qqqq EQU 123234h
qqqq EQU 012323
qqqq EQU 0.123234
qqqq EQU 123234 342
qqqq EQU 4294967296
```

В данном примере только второй и третий вариант – макроконстанта, остальные – текстовые макро. Последний вариант таким не является, так как превышает диапазон значений для **dword**.

Замете, что поскольку препроцессор в правой части выделяет корректные выражения, правая часть не может состоять из недопустимых символов. Но при этом она может состоять из директивы определения литерала: «**< >**» – угловых кавычек.

Директива <*текст* > – определяет литерал, таким образом, указывая препроцессору **ML** , что он должен воспринимать нечто как строку символов. При этом сами «**< >**» – в строку не попадают. Директива **< >** – является единственной директивой для препроцессора **ML** , которая определяет литералы.

Именно по этой причине, все виды кавычек – двойные, одинарные, - вот такие одинарные, воспринимаются как простые символы, и как следствие проходят к значениям параметров макро. То есть, например: MyMacro “Привет, это строка в двойных кавычках” MyMacro ‘Привет, это строка в одинарных кавычках' MyMacro Привет, это строка в специальных кавычках` MyMacro "Привет, это строка"'И это' И замете, что во всех случаях кавычки так же попадают в значения формального параметра макро. Вы можете использовать этот факт, например, для того, чтобы менять поведение макро, в зависимости от типа кавычек обрамляющих строку.

Кроме директивы, определяющей литерал, препроцессор **ML** имеет свой **ESC** -символ (символ отмены). В отличие от **C** этот символ – «**!** ». Он отменяет действие других символов (<, >, ", ', %, ; , а так же символ запятой), которые могут иметь функциональность в том, или ином выражении. Если вы хотите получить «**!** », вы должны использовать последовательность «**!!** ».

К сожалению, не обходится без проблем и с символом отмены «!». Восстановить точный алгоритм работы мне не удалось. Единственное, что возможно – это привести несколько примеров с

непонятными эффектами при его использовании:

```
literal    EQU <!>      ;; Пустая строка
;; Ошибка –
;;: error A2045: missing angle bracket or brace in literal

literal    EQU <!!>
;; Один символ «!»
literal    EQU <!!!!>
;; Не имеют эффекта
literal    EQU <Привет!" fgd!">
literal    EQU <Привет" fgd">
;; Один символ «>»
literal    EQU <!!!>>          ;; literal = «>»
literal    EQU <Текст!!!>>     ;; literal = «Текст>»
;; Хотя при вызове макро, «!» ведёт себя нормально
;; а так же он ведёт себя нормально в директиве TEXTEQU
Char <Текст!>>
```

Вывод – не пользуйтесь директивой **EQU** для определения литералов, для этого есть другая директива – **TEXTEQU**.

Для директивы **TEXTEQU** алгоритм несколько отличен от алгоритма **EQU** , так как в **TEXTEQU** обрабатывается правое выражение на наличие символа **%**. То есть вы можете определить этот код:

```
literal TEXTEQU %FunMacro()
```

Или

```
literal TEXTEQU %(10-5)*30  ;; literal = “150”
```

На самом деле как вы видите, внутренняя работа **TEXTEQU** значительно отличается от **EQU < >**. Видимо по этому разработчики **ML** решили её ввести.

```
В руководстве MASM32 написано:
        ------------------------------------------------------------------------------------------
        The TEXTEQU directive acts like the EQU directive with text equates but
        performs macro substitution at assembly and does not require angle brackets.
        The TEXTEQU directive will also resolve the value of an expression preceded
        by a percent sign (%). The EQU directive does not perform macro substitution
        or expression evaluation for strings.
        ------------------------------------------------------------------------------------------
              Теперь вы должны понимать, что это не совсем так. Является ли это ошибкой
          разработчиков ML? Видимо да. В частности EQU не должна была переводить
      в статус переменных литералов определения типа:
               NOLITERAL EQU db
                    И конструкция ниже должна была бы вызывать ошибку:
               literal EQU db
       literal EQU dw
                Но ошибка не появляется, более того значение literal меняется на dw
```

В заключении к этому пункту, вы должны осознать, что тип определений невозможно изменить. То есть переменная не может стать целочисленной константой:

```
literal    EQU string
literal    EQU 123       ;; Это текстовой макро
```

Второе переопределение символа **literal** , не изменит его тип на тип целочисленной константы.

```
Думаю, у Вас возник вопрос:
        – Что такое? Недокументированные возможности MASM?
      У меня есть веские основания считать это  ничем иным, как
        ошибкой разработчиков. Давайте предположим, что все макропеременные хранятся
        компилятором в памяти в виде массива структур. Не вдаваясь в подробности,
        пусть эта
        структура будет такая:
        macrodefine struct
   type dd ? ;; тип макроконстанты
   value dd ?
```

```
    ends
             Как видно из структуры, значение макроконстанты может быть только dword'ом.
          Если это строка, то в поле value может быть записан указатель на строку
          (например, ASCIIZ).
                Поле type может принимать только два значения, которое описывает тип
            value: либо value – содержит числовое значение макропеременной (константы).
                Если определяется числовая константа то, вызывается одна функция (назовём
              её setmacrodefine_val()), которая добавляет в таблицу макроконстанту.
                Это конечно предположение. И в действительности всё может быть ещё
                проще или ещё сложнее. Однако вероятность того, что свойства макропеременных
                хранятся именно подобным образом близка к единице. Теперь если
              вы немного подумаете, то поймёте:
string    EQU <string>       ;; Строковая макропеременная
string    TEXTEQU string     ;; Строковая макропеременная
string    EQU string         ;; Должна была быть константой
              Последний случай записывается в таблицу, как строковая макропеременная
          по той простой причине, что string не может быть записано в поле value,
          а поле type не имеет специального значения, чтобы указать, что value        – это
      константный указатель на строку (помните C++?).
                В конце концов, совершенно не важно угадал ли автор причину,
          или нет. Важно другое – что ошибка достаточно явная. А, кроме того, так
          и
              не была исправлена
      до сих пор (версия 7.0). Зато теперь вы сможет с пониманием отнестись к таким
        неожиданным эффектам.
                Видимо разработчики не задумываются о том, что кто-то будет использовать
        определения MASM, иначе, нежели это написано в руководстве. И кому-то взбредёт
        в голову
        проверить, а можно ли переопределить EQU.
                А подумайте, к каким бы серьёзным неуловимым ошибкам произвела бы эта
              халатность, если бы на MASM писали сложные приложения. Но как видно
      их никто не пишет.
```

Свои особенности имеют так же целочисленные выражения с оператором «=». В таких выражениях перед их выполнением осуществляется полная замена всех макроконстант, макропеременных на их значения, и вызов всех макрофункций.

Как вы думаете, что будет в следующем примере:

```
literal EQU Something
literal  = 1234
```

Варианты ответа:

1. Произойдёт ошибка переопределения константы.
2. **literal** = 1234.

Второй вариант ответа мы должны откинуть сразу, потому что в этом пункте чётко определили, что данное переопределение невозможно. Первый вариант ответа больше похож на правду…. Однако не соответствует истине. Что же произошло? А произошло следующее:

1. Препроцессор нашёл лексемы « *literal* » и « *1234* ».
2. Обнаружил, что « *literal* » является текстовым макро, и именно поэтому выполнил замену лексемы « *literal* » на её строковое значение.
3. Проанализировал строку: « *Something = 1234* ».

Этот факт может быть легко доказан, следующим тестом:

```
literal EQU Something
literal  = 1234

%echo @CatStr(%Something)
============================
Вывод:
1234
```

Если вас сбил с толку этот пример, не отчаивайтесь. Всё дело в том, что препроцессор **ML** в разных выражениях по-разному заменяет макропеременные. Вот об этом мы и поговорим в следующем

пункте.

А пока подумайте, что должно случится в этом примере:

```
num  EQU number
num  EQU 123
num  = 1234
```

На этом можно было бы закончить данный пункт, если бы не одна особенность использования строк в вызове макро. А точнее приоритет анализа кавычек и директивы определения литерала **< >**. Не смотря на описанный выше алгоритм поведения макро, оказывается, что препроцессор при вызове макро выполняет определение литерала в кавычках, но что самое интересное, как было отмечено, выше сами кавычки попадают в строку. Если вам нужно передать макро одиночную кавычку вы должны воспользоваться символом отмены «**!** ». Однако самое неприятное таится в том, что символы «**< >**» и кавычки конкурируют между собой в определениях строк. Например, попробуйте сказать, что должно было бы получиться в этом случае:

```
%echo @CatStr(<Раз">,<"Два>)
OUT:
Раз">,<"Два
```

А можно было бы подумать, что **ML** должен принять операторы <> и запятую. Данное место – источник многих сложно обнаруживаемых ошибок. Например:

```
FORC char,<str>
 m$__charcode = \
 @InStr (1,<@ABCDEFGHIJKLMNOPQRSTUVWXYZ>,<char>)
…
```

Если в строке попадается символ кавычки, а макропеременная char заменяется на значение кавычки, имеем:

```
m$__charcode = @InStr (1,<@ABCDEFGHIJKLMNOPQRSTUVWXYZ>,<”>)
```

В этом случае мы получаем ошибку:

```
missing single or double quotation mark in string
```

Так и должно быть, потому что кавычки имеют высший приоритет анализа, чем оператор **< >**. Более того, угловые кавычки <> имеют самый низкий приоритет по отношению ко всем спец. символам, что согласуется с *MASM Reference*. Посмотрите на Дополнение к статье: пункт **3.a.i** , который подозрительно выделен « *жирным* ». В частности, следующее выражение, которое работает без проблем:

```
TEXT   TEXTEQU <"> ;; Это работает?
TEXT   TEXTEQU <;> ;; И это???
```

Появляется закономерный вопрос: для чего символ отмены «**!** »?
Данный пример демонстрирует скрытые глубины анализатора **ML**. А точнее его архитектурное несовершенство. Так как выражения с **TEXTEQU** как видно обрабатываются отдельной функцией, которая проверяет в первую очередь наличие угловых скобок «**< >**». Все другие выражения **ML** обрабатываются другой стандартной функцией, которая была написана задолго до появления **TEXTEQU**.

Замечательная наука всем программистам, которая демонстрирует, во что выливается халатность архитектора при дальнейших попытках расширения продукта.

Зато благодаря **TEXTEQU** пример с поиском символа в строке имеет решение:

```
m$__char TEXTEQU <char>
m$__charcode = \
@InStr (1,<@ABCDEFGHIJKLMNOPQRSTUVWXYZ>,%m$__char)
```

Единственно, отчего не может помочь данный код – это от вылавливания в строке символов «> или <». Для этого можно использовать специальную проверку в условных блоках на наличие символа «>», но при этом придётся отказаться от микроблока **FORC**.

III.3. Обработка выражения в MASM

MASM обрабатывает выражения в правой и левой части в зависимости от контекста. Там, где вам необходима предварительная обработка выражений, используется оператор «**%** ». Он заставляет препроцессор **ML** сначала вычислить выражение после оператора **%**(то есть выражение в правой части относительно **%**), и только потом продолжить анализ всей строки. Например, если вы хотите, чтобы при вызове макро:

```
num  TEXTEQU <123>
FunMacro num
```

макропараметр был бы равен не строке « *num* », а значению текстового макро **num** , вы должны поставить оператор **%** перед **num**. Например:

```
FunMacro %num
;;или
FunMacro %(1+2*num)
```

Но и с оператором **%** не всё гладко.
Оказывается препроцессор **ML** , различает два (фактически три) вида выражений, в которых используется оператор **%**. Первый вид выражений – Арифметические:

Все выражения, содержащие операторы +,-,*,\ а так же сдвиговые и битовые операции

Строковые выражения:

Все выражения результат вычисления которых – строка.

Примеры:

```
;Арифметические выражения
%(num shl 3)
%num = 2134 shl 3 + 2*6
;Всё равно арифметическое выражение
%(num shl 3 @CatStr(num))
;Строковое выражение
%(@CatStr(num shl 3))
;Строковое выражение
%PROGRAM IMAGE
```

Так вот что интересно.

В арифметических выражениях происходит полная замена правой части: вызовы макрофункций, значение макроконстант, макропеременных любых типов, как строковых, так и целочисленных. Так же в левой части выражения: замена строковых макропеременных, и вызов макрофункций.

То есть:

Левая часть = Правая часть (Вызвать все макрофункции, и заменить все строковые макропеременные) = (Вызвать все макрофункции, и заменить все строковые и целочисленные макропеременные и константы)

В строковых выражениях происходит замена только строковых макропеременных (текстовых макро) (замете, что в **ML** нет строковых макроконстант). Это значит что в случае:

```
%echo PROGRAM_IMAGE_BASE
```

Появится: « *PROGRAM_IMAGE_BASE* », а не его числовое значение.

Однако есть и третий частный случай, когда оператор **%** относится только к одному литералу:

```
%literal
```

В этом случае происходит полный комплекс подстановок:

1. Вызываются макрофункции.
2. Заменяются все макропеременные или макроконстанты.

Например:

```
FunMacro %literal
```

Значение **literal** будет подставлено в вызов макро, в независимости от того, какой тип имеет **literal**.

-= Внимание =-

Выдержка их руководства MASM: -= temp TEXTEQU %(SIZEOF array / LENGTHOF array) % ECHO Bytes per element: temp Note that you cannot get the same results simply by putting the % at the beginning of the first echo line, because % expands only text macros, not numeric equates or constant expressions. -=

Следует так же отметить, что в выражениях с **exitm** оператор **%** работает точно так же, как с выражениями в **TEXTEQU**.

III.4. Целочисленные выражения MASM

Целочисленные, побитовые операции так же необходимы разработчику макроопределений. Они дают возможность скрыть обработку битовых полей, или вычисление сложных выражений. Например, как это сделано в макрофункции **$$$MAKELANGID**.

```
$$$MAKELANGID  macro p:REQ,s:REQ
m$__langid = (s SHL 10) or p
   EXITM <m$__langid>
   endm
```

Вы всегда должны помнить, что препроцессор **MASM** не различается знаковые и беззнаковые числа (подобно тому, как это делает **x86**), и значение числа не может выходить за диапазон **dword**. Препроцессор **MASM** не выдаёт предупреждений при переполнении. Следующий пример демонстрирует такое поведение:

```
myint = 0ffffffffh
myint = myint + 1 ;; myint = 0
%echo @CatStr(%myint)

================================

OUT:
0

;; Ещё один пример с умножением:

myint = 0ffffffffh
;;
;; 0ffffffffh * 2 = (dword)1FFFFFFFEh = 4294967294
myint = myint * 2
%echo @CatStr(%myint)
================================
OUT:
4294967294
```

В следующей статье мы поговорим про то, как работать с 64-bits макропеременными, используя данный факт.

Ниже приводится список операций, которые могут участвовать в целочисленных выражениях MASM.

Оператор — Пример — Описание

AND — res = op1 AND op2 — Операция логического «И» над каждым битом операндов op1 и op2.

OR — res = op1 OR op2 — Операция логического «ИЛИ» над каждым битом операндов op1, op2

NOT — res = NOT op1 — Операция логического «НЕ» над каждым битом операнда op1

XOR — res = op1 XOR op2 — Операция XOR между операндами op1, op2

SHL — res = op1 SHL count — Выполняет побитовый сдвиг влево (наподобие команды x86 shl) операнда op1, на количество бит, указанное в операнде count.

SHR — res = op1 SHL count — Выполняет побитовый сдвиг вправо операнда op1, на число бит, указанное в операнде count.

+,-,*,/ — Основные математические операции

MOD — res = op1 MOD op2 — Возвращает остаток от деления операнда op1 на операнд op2

[] — res = op1[op2] — Операция: «Смещение». Выполняет сложение операндов op1 и op2

III.5. Вычисление рекурсивных выражений

Теперь, когда мы рассмотрели правила анализа и вычисления выражений в **MASM** , остаётся раскрыть важный вопрос: « *Как происходит анализ выражений, если они состоят из других выражений?* ».

Обычно это называется короче: *вложенные выражения*.

Вложенное выражение – это такое выражение, элементы которого сами являются выражениями, которые так же могут иметь вложенность.

Замороченное определение, похожее на «Иди туда, не знаю куда, возьми то, не знаю что» – пример старинной народной русской рекурсии, которая так часто встречается в нашей жизни. :))

Например, вызов макрофункции при вызове макро – это вложенное выражение:

```
MyMacro   FunMacro(Мой парамерт)
;;Или это:
%echo FunMacro(Мой параметр)
;;Или это:
MyMacro   FunMacro(Fun2(Привет))
```

Вложенность характеризуется параметром количества уровней вложенности. В недавнем примере уровень вложенности был равен двум. При чём вызов **Fun2()** можно называть выражением низшего уровня вложенности, а вызов макро **MyMacro** – выражением верхнего уровня.

После анализа выражений, и получения их многоуровневой структуры вложенности, препроцессор начинает вычислять результат выражения самого низшего уровня. Потом подставляет его результат в выражение следующего уровня, и так далее.

Например, для случая:

```
Fun2   macro param
 exitm <MyCount = param>
  endm
FunMacro(Fun2(%(12+34)))
```

Порядок вычислений такой:

1. %(12+34) = 46
2. Fun2(46)
3. FunMacro(MyCount = 46)
4. Результат выполнения FunMacro(MyCount = 46)

А иначе препроцессор не смог бы. Если бы он начал вычисления выражений с верхнего уровня, то это то же самое, как если бы он попытался выполнить народную русскую рекурсию: «Пойди туда, не знаю куда…, вычисли то, не знаю что» или FunMacro(???)

То есть: Вложенные выражения вычисляются последовательно от низшего уровня к верхнему, и результаты вычисления каждого уровня становятся материалом для выражений следующего уровня.

Это правило называется рекурсивным вычислением выражений. Оно используется везде, кроме мест вычисления значений макропараметров при вызове макро (как макросов, так и макрофункций). В этом случае действует правило: результат вложенного выражения присваивается макропараметру и не анализируется повторно. Это значит, что в данном примере:

```
myvar EQU <123>

MyMacro macro param1,param2,param3
echo param1
 endm

FunMacro macro param:VARARG
  exitm <param>
  endm

MyMacro FunMacro(var,@CatStr(<%>,myvar),var4)
```

вывод будет таким:

```
var,myvar,var4
```

То есть препроцессор не будет снова вычислять выражение для второго макропараметра функции **FunMacro()**. Если бы он сделал это, то тогда вывод был бы таким, как в этом случае:

```
%echo FunMacro(var,@CatStr(<%>,myvar),var4)
Вывод:
var,123,var4
```

Теперь, когда вы знаете все тонкости вычисления выражений в MASM, настало время рассмотреть Встроенные макрофункции и директивы, которые участвуют в этих выражениях.

III.6. Встроенные макрофункции и директивы

Несмотря на то, что этот пункт не касается самих макросов в **MASM** , он необходим, для того, чтобы строить макросы, и манипулировать выражениями, возникающими внутри макросов.

MASM обладает несколькими встроенными макрофункциями, макропеременными и макроконстантами, которые работают так, как если бы они были макро, определённые вами. Вот список этих предопределений:

Определения Даты и Времени

@Date, текстовое макроопределение (не макрофункция) — Возвращает строку вида ММ/ДД/ГГ Где: ММ – месяц, две цифры ДД – день, две цифры ГГ – год, две цифры

@Time, текстовое макроопределение (не макрофункция) — Возвращает текущее время в 24-х часовом формате вида ЧЧ:ММ:СС ЧЧ – часы, два числа ММ – минуты, два числа СС – секунды, два

числа

Информация об окружении

@Cpu, числовая макроконстанта — Битовая маска, определяющая режим работы процессора. Никакой информации о полях этой маски нет.

@Environ(env), макрофункция — Возвращает строковое значение переменной среды окружения. Например: Вывод: F:\Temp\asm

```
%echo @Environ(TEMP)
```

@Interface, целочисленная макроконстанта — Информация о языковых параметрах вызова.

@Version, строковая макроконстанта — Возвращает версию ML. Например:

```
%echo Version = @Version
Вывод: Version   = 614Или 615 в MASM 6.15
```

Информация о файле

@FileCur, строковая макропеременная — Возвращает имя файла и путь к нему (если есть), так как был подан этот файл в командной строке компилятору ML. Пример:

```
%echo FileCur = @FileCur
Вывод: FileCur = .\start.asm
```

@FileName, строковая макропеременная — Возвращает имя файла, без его расширения. То есть для модуля start.asm:

```
%echo FileName = @FileName
Вывод:FileName = START
```

@Line, целочисленная макроконстанта — Возвращает номер текущей строки в файле. Пример:

```
%echo Line = @CatStr(%@Line)
Вывод:Line = 31
```

Строковые макрофункции

@CatStr(string1 [[, string2...]]), макрофункция — Возвращает строку, созданную объединением строк параметров функции. Пример:

```
%echo @CatStr(<my>,var)
Вывод:Myvar
```

@InStr([[position]], string1, string2), макрофункция — Возвращает позицию вхождения строки string2 в строку string1. Если параметр position определён, тогда поиск начинается именно с этой позиции. Отсчёт позиции начинается с единицы. В случае, если вхождение не найдено макрофункция возвращает значение 0. Параметр position должен быть целым числом больше нуля, но не равным нулю. Пример:

```
%echo @InStr(1,asdfg,s)
Вывод:02
```

@SizeStr(string) макрофункция — Возвращает число, характеризующее длину строки, или, что тоже самое количество символов в строке. Функция возвращает число, однако, поскольку это макрофункция то тип возвращаемого значения – строка.

@SubStr(string, position [[, length]]) макрофункция — Возвращает подстроку строки string, начиная с позиции, указанной в параметре position (отсчёт начинается с 1). Если необязательный параметр length задан, он ограничивает размер возвращаемой строки. Параметр length не может быть меньше нуля, и не может быть строкой. Пример:

```
%echo @SubStr(1234567890,2)
%echo @SubStr(1234567890,1,5)
Вывод:
```

```
234567890
12345
```

Информация о сегментах

@code, строковая макропеременная — Возвращает имя сегмента кода.

@data, строковая макропеременная — Возвращает модель памяти. Пример:

```
%echo data    = @data
Вывод:data    = FLAT
```

@fardata?, строковая макропеременная — Равен имени сегмента FARDATA?

@WordSize, численная константа — Содержит размер слова в байтах. Для 16-bits – 2. Для 32-bits – 4.

@CodeSize, численная константа — Содержит идентификатор типа памяти. 0 – TINY, SMALL, COMPACT, FLAT. 1 – MEDIUM, LARGE, HUGE

@Model, численная константа — 1 – TINY 2 – SMALL 3 – COMPACT 4 – MEDIUM 5 – LARGE 6 – HUGE 7 – FLAT

@CurSeg, строковая макропеременная — Хранит имя текущего сегмента.

@fardata, @stack, строковая макропеременная — Содержат соответствующие имена сегментов

Кроме знания макрофункций, нам так же понадобятся знания о блоках ветвлений или просто **IF** блоках. Эти блоки позволяют исполнять тот или иной участок исходного кода в зависимости от того, выполняется какое-либо условие или нет. Часто это называют « *Условным ассемблированием (компиляцией)* », однако для **MASM** это нечто большее, нежели простое управление компилятором, так как, вы уже поняли, мы имеем дело, как с кодом машины, так и с макрокодом, который вычисляется и живёт только во время компиляции.

Условный блок в **MASM** имеет следующий общий вид:

```
[IFDIRECTIVE]      условие
...
[ELSEDIRECTIVE] условие
...
ELSE
...
ENDIF
```

Если выражение «Условие» равно истине, то выполняется блок кода, идущий после условной директивы, иначе управление передаётся на следующий оператор за блоком. [**IFDIRECTIVE**]/[**ELSEDIRECTIVE**] – могут быть той или иной директивой условия. Стандартные директивы **IF** /**ELSEIF** /**ELSE** требуют, чтобы выражение, стоящее при них, было целочисленным. Если вам необходимо проверять другие условия, то для этого в MASM предусмотрены специальные директивы.

Список [**IFDIRECTIVE**]/[**ELSEDIRECTIVE**]:

Блок — Условие выполнения блока

IF выражение ELSEIF выражение ELSE — если выражение равно истине

IF1 ELSEIF1 — если ассемблер выполняет первый проход

IF2 ELSEIF2 — если ассемблер выполняет второй проход (устарело)

IFE выражение ELSEIFE выражение — если выражение равно нулю

IFDEF выражение ELSEIFDEF выражение — если идентификатор, который является результатом выражения, определен. Идентификатором может быть макро, макропеременная, переменная, макроконстанта, любой другой идентификатор. При помощи этой директивы, можно проверить

была ли определена та или иная переменная, макро, константа.

```
IFDEF     PROGRAM_IMAGE_BASE
;; Выполняем действия если PROGRAM_IMAGE_BASE
;; определена
ELSE
…
```

IFNDEF выражение ELSEIFNDEF выражение — если идентификатор не определён.

IFB строка ELSEIFB строка — если строка пустая. Строка считается пустой, если её длинна равна нулю, либо она содержит одни пробелы. С помощью этой директивы можно определяет присутствие/отсутствие необязательных макропараметров.

```
MyMacro      macro    param1,param2
IFB <param2>
;; Если макропараметр не определён,
;; генерируем ошибку
.ERR <Не определён параметр param2>
```

IFNB строка ELSEIFNB строка — если строка не пуста.

IFDIF str1,str2 ELSEIFDIF str1,str2 — если строки различны.

```
IFDIF <String>,<string>
  echo Этот код выполнится
  echo потому что строки различны
ENDIF
```

IFDIFI str1,str2 ELSEIFDIFI str1,str2 — если строки различны (без учёта различий в регистре букв).

```
IFDIF  <String1>,<string2>
  echo Этот код не выполнится
  echo потому что строки Одинаковы
ENDIF
```

IFIDN str1,str2 ELSEIFIDN str1,str2 — если строки одинаковы.

```
IFDIF  <String1>,<string2>
   echo Этот код не выполнится
   echo потому что строки Различны
ENDIF
```

IFIDN str1,str2 ELSEIFIDN str1,str2 — если строки одинаковы (без учёта различий в регистре букв).

```
IFDIF  <String1>,<string2>
   echo Этот код выполнится
   echo потому что строки Одинаковы
ENDIF
```

На протяжении всей статьи я часто пользовался следующей директивой, которая позволяет выводить текст на консоль во время компиляции. Эта директива **echo**. Как мы узнаем позже, она оказалось просто незаменимой при проектировании макро.

Вы уже, наверное, убедились насколько полезна эта директива, позволяющая заглянуть, а что именно происходит в недрах макроса, или посмотреть значения макропеременных.

Кроме этого, есть ещё одна группа директив, без которой мы не сможем обойтись. Не сможем потому, что макрофункции, или макросы, которые мы собираемся создавать должны быть слегка умными, иначе говоря, иметь «защиту от дурака».

Если кто-то неправильно использует макрос, то код, полученный таким образом может быть неправильным с точки зрения программиста, но не вызовет подозрений у компилятора. Поэтому макро не просто должен завершится, а и каким-то образом остановить компиляцию программы с выдачей сообщения об ошибке.

Именно для этого и существует простой набор директив условной генерации ошибки. Действуют они подобно условным блокам и директиве **echo**. Пример безусловной генерации ошибки:

```
.ERR <Ошибочка вышла, гражданин начальник>
```

Условная генерация ошибки, имеют ту же форму, что и **IFDIRECTIVE** в таблице выше, однако последним дополнительным параметром является строка сообщения. Например:

```
.ERRE  выражение,<ошибка, если выражение равно нулю>
.ERRNZ выражение,<ошибка, если выражение не равно нулю>
.ERRDEF id,<ошибка, если id определен>
.ERRB строка,<ошибка, если строка пуста>
.ERRNB строка,<ошибка, если строка не пуста>
.ERRDIF str1,str2,<ошибка, если строки различны>
.ERRDIFI str1,str2,<ошибка, если строки различны
 (без учёта регистра)>
.ERRIDN str1,str2,<ошибка, если строки одинаковы>
.ERRIDNI str1,str2,<ошибка, если строки одинаковы
 (без учёта регистра)>
```

III.7. Символ макроподстановки

Ещё раз вернёмся к формальным параметрам макро. Как было сказано, при раскрытии макроопределения препроцессор заменяет в теле макро формальные названия на их величины. В **MASM32** предусмотрено ещё одно средство подстановки макропараметров – внутри строкового литерала.

Предположим нам нужно, чтобы макро генерировал строку: « *label_xx* ». Где **xx** – это формальный параметр макро. Это можно сделать двумя способами:

```
@CatStr(label_,xx) ;;Вызов макрофункции конкантенации строк
или
label_&xx&   ;;Использование символа макроподстановки
```

То есть если во время генерации макро, препроцессор встречает в его теле символ «**&** », он анализирует строку после него. Если эта строка однозначно определяет один из макропараметров, препроцессор заменяет выражение **&***макропараметр***&** на значение макропараметра.

Следует отметить, что если макропараметр начинает или заканчивает литерал, то можно использовать только один символ «&»:

```
label_&xx
;;или ещё пример
label_&xx&&xx2  ;; Замена для двух макропараметров xx и xx2
```

III.8. Макроблоки

И, наконец, у читателя должен остаться единственный вопрос: « *А как обрабатывать переменные типа VARARG* »? Например, рассмотрим возможный макро для вызова функций – **STDCALL** :

```
stdcall  macro funname,params:VARARG

       endm
```

Этот макро должен генерировать код вызова функции согласно конвенции STDCALL:

1. Поместить параметры в стек в обратном порядке их определению.
2. Вызвать функцию **funname** , предварительно видоизменив её имя по правилам **STDCALL**.

Получить видоизмененное имя функции по значению параметра **funname** можно было бы при помощи символа макроподстановки.

```
            call _&funname@(количество параметров * 4)
```

Но непонятно, как распознать параметры функции, которые представляют собой строку, где значения разделены символом «,». Более того, не понятно, как вообще можно получить эти параметры, и посчитать их число, ведь макропараметр **params** – это одна строка. То есть при вызове макро:

```
stdcall win32fun,1,2,3
```

Мы должны как-то определить количество параметров, а потом их значения.

Именно для решения этой задачи в **MASM** предусмотрены несколько специальных макроопределений, которые можно назвать *макроблоками*.

Первый из них **FOR** позволяет получить значения элементов, разделённых в строке символом «**,** ».

```
FOR parameter[:REQ | :=default], string
              statements
ENDM
```

Вспоминая **C** конструкцию **FOR** , вы сразу поймёте что это цикл, где значение **parameter** последовательно принимает значения элементов списка **string**.

Вот вам *wonderful* пример:

```
FOR parameter, <It’s, wonderful, wonderful, asm>
  echo parameter
            ENDM
ВЫВОД:
-=-=-=-=-=-
It's
wonderful
wonderful
asm
-=-=-=--=-=-
```

А вот пример макрофункции, который подсчитывает число аргументов **VARARG** :

```
@ArgCount MACRO parmlist:VARARG
           count = 0
           FOR param, <parmlist>
               count = count + 1
           ENDM
           EXITM count
       ENDM
```

Вот в принципе, уже на основе этих знаний можно было бы организовать макрос **stdcall** :

```
stdcall    macro funname,params:VARARG
           count = 0
           FOR param, <parmlist>
               count = count + 1   ;; Считаем число параметров
   push param           ;; Помещаем их в стек
           ENDM
                                   ;;Вызываем функцию
  call ???                ;;А вот как это сделать?
   endm
```

Ещё несколько минут необходимо для того, чтобы понять, что этот макро работает неправильно. Хотя бы потому, что параметры помещаются в стек не так. Нужно было бы помещать их от последнего к первому, а не от первого к последнему. А, кроме того, ведь символ макроподстановки нельзя употреблять к макропеременной **count** , потому что это не макропараметр, это макропеременная.

К сожалению, в MASM нет обратной конструкции **FOR**. Поэтому самый простой выход, который напрашивается сам собой – это изменить порядок параметров в списке, а потом только генерировать команды **push**.

Вторую проблему можно легко решить, воспользовавшись макрофункцией конкатенации строк:

```
call @CatStr(_,funname,@,%(count*4))
```

С параметрами в стек будет посложнее. В принципе я бы решил эту задачу, если бы **MASM** поддерживал бы такой тип макропеременных как массив. Но хотя **MASM** и не поддерживает этот тип, его можно эмулировать.

```
count = 0
           FOR param, <paramlist>
               count = count + 1  ;; Считаем число параметров
   @CatStr(var,%count) TEXTEQU <param>
           ENDM
```

Как вы можете догадаться, в этом примере создаются макропеременные **varXX** , которым присваиваются значения параметров. Теперь с той же лёгкостью можно работать с этими переменными. Можно снова использовать цикл **FOR** , однако в данном случае, было бы грамотней воспользоваться значением **count** , и выполнить цикл столько раз, сколько записано в нашем счётчике параметров. Для этого мы воспользуемся ещё одним макроблоком **rept** , о котором скажем позже:

```
nparams = count


  REPT    nparams             ;; Начало блока
               push @CatStr(var,%count)
   count = count - 1
  ENDM
```

Блок **REPT** выполняется столько раз, сколько указано в **nparams**. Я ввёл эту дополнительную макропеременную, для того, чтобы значение, указанное в **REPT** осталось неизменным. Однако этого не нужно. Можно было бы написать и так:

```
REPT    count             ;; Начало блока
            push  @CatStr(var,%count)
   count = count - 1
  ENDM
```

Значение макропеременной **count** инициализирует цикл только один раз вначале, после чего, она может, как угодно менять значение.

И ещё один макроблок, без которого нам невозможно будет реализовать макрос для определения строк уникода, или макрос, который позволяет писать строки **OEM** в редакторе использующий кодировку win cp-1251 (например, при создании консольных приложений).

Этот макроблок **FORC** :

```
FORC char, string
;;блок
           ENDM
```

Блок **FORC** выполняется столько раз, сколько символов в строке **string** , при этом макропараметр **char** равен текущему символу из строки.
Например, посчитать количество символов в строке можно было бы так:

```
count = 0
FORC char, <Сколько тут символов?>
count = count + 1
           ENDM
%echo @CatStr(%count)
```

А вот так, можно было бы посчитать количество пробельных символов.

```
count = 0
FORC char, <Сколько тут символов?>
 IFB <char>
count = count + 1
```

```
  ENDIF
            ENDM
%echo @CatStr(%count)
```

Упражнение:

TheSvin 'у, как и любому программисту, который часто имеет дело с битами, было бы удобно записывать значения бит по группам, через пробел.

```
;;Вот так неудобно и ненаглядно
        mov eax,011110111011b
;;Вот так удобно и наглядно, но компилятор выдаст ошибку
;;Вариант1
        mov eax, 0111 101 1101 1b
;;А вот так вообще замечательно, только ML неправильно поймёт
;;Вариант2
        mov eax, [0111] [101] [1101] [1]b
```

Хорошо бы было написать некую макрофункцию, которая смогла бы позволить записывать эти выражения:

```
mov eax,nf(0111 101 1101 1b)
```

Напишите такую макрофункцию, которая позволила бы это делать. Напишите её для первого и второго вариантов исполнения.

III.9. Отладка макроопределений и заключение

А напоследок… остаётся маленькая деталь.

И эта деталь не самая приятная. Отладка макроопределений и их испытания невозможны под отладчиком. А, кроме того, если при генерации макро возникает ошибка, то **ML** выдаёт её в жутком виде:

```
.\start.asm(84) : error A2008: syntax error : in directive
 MacroLoop(3): iteration 8: Macro Called From
  .\start.asm(84): Main Line Code
```

То есть он выдаёт относительную строку в макро **MacroLoop(3)** , где эта ошибка появилась. А если ещё макровызовы будут вложенными, то вам лучше не видеть этой замечательной картины.

Единственной возможностью качественно и относительно легко отлаживать макро – это употребление директивы **echo**.

На протяжении статьи вы не раз наблюдали примеры её использования. Но я снова повторюсь:

```
;; Для макропараметров
echo macroparam
;; Для макропеременных типа строка или текстовых макро
%echo macrovar_string
;; Для целочисленных макропеременных, или макроконстант
%echo @CatStr(%macro_num)
```

Заметьте, чтобы вывести значение целочисленной макропеременной необходимо воспользоваться макрофункцией **@CatStr()** , и перед аргументом указать оператор **%**. Почему именно так обсуждалась в пункте III.2. Определение макро переменных и строк.

Теперь вы знакомы с теорией использования макроопределений в **MASM32** , и сможете смело приступать к разработке макро. Именно этим мы и займёмся в следующей практической части нашего руководства, а так же заполним некоторые пробелы, на которые не обратили внимания здесь.

III.10. Абстрактный алгоритм анализа строки MASM (Дополнение)

1. Определены таблицы элементов:

Таблица переменных — Хранит сведения о всех переменных модуля

Таблица меток — Хранит список меток в коде.

Таблица процедур — Хранит таблицу и прототип процедур

Список ключевых слов KEYLIST — Хранит список ключевых слов, на которые реагирует ML

Таблица макрофункций — Хранит тело всех макро, их имена и тип: макрофункция, или макро. Список макропараметров

Таблица макросов — -=-

Таблица макропеременных, или переменных времени компиляции — Хранит тип макропеременной и её значение.

Всё остальное, что не включено

2. Начальное состояние анализа строки.

3. Читать поток символов, пока не встретится символ возврата каретки без предыдущего символа «/». Игнорировать часть строки после «;»

a. Определить наличие лексем первого уровня в строке:
i. Выделить все строковые литералы в кавычках, если только это не выражение с TEXTEQU и символ комментария «;»
ii. Строковые литералы: <текст>
iii. Численные литералы: 1234, 1234h, 01011b
iv. Правильные литералы: строка из символов «A-Z,a-z,_0-9», но не начинающаяся на цифру
v. Литералы разделители: «,.»
vi. Управляющие Литералы: «+-*» Правильные литералы: строка из символов «A-Z,a-z,_0-9», но не начинающаяся на цифру
b. Проверить правильные литералы на совпадение в списке ключевых слов, и определить схему выражения. В зависимости от схемы выражения, выполнить или пропустить:
i. Проверить правильные литералы на совпадение в списке макро (в зависимости от способа вызова в списке макрофункций, или макросов)
ii. Проверить на наличие имени правильного литерала в таблице макропеременных.
iii. Осуществить вызов и замену макро и макропеременных, в соответствии с выражением строки.
iv. Вычислить все выражения допустимые в ML (+-*).
c. Осуществить разбор схемы.
i. Если это определение процедуры, записать в таблице процедур имя и прототип новой процедуры
ii. Если это макроопределение: анализировать его тело. Если найден возвращаемый параметр, записать макроопределение в таблицу макрофункций, иначе в таблицу макросов.
iii. Если это определение EQU вычислить правую часть.
1. Если эта макропеременная уже есть в таблице макропеременных, и её тип – числовой, выдать ошибку. Если эта макропеременная имеет строковый тип, изменить строку, на которую указывает свойство value этой макропеременной.

2. Если правая часть числовой литерал – записать EQU определение в таблицу, и пометить его тип как числовой константы. Записать в свойство макропеременной value значение указателя на строку. Записать свойство value равным числу.
3. иначе EQU – переменная, имеющая указатель на строку. Записать в значения свойства value указатель на строку.
iv. Если это выражение с «=» или подобное, выполнить замену всех литералов на макроконстанты, переменные, вызов всех макрофункций, и только потом выполнять выражение.

4. Перейти к анализу следующей строки.

Программирование на языке Assembler в FASM

Автор: pas

Дата: 2004-01-06

Введение

В начале было слово... Если точнее то было просто предложение от Kinder-а написать статью посвящённую макросам в FASM. Я согласился попробовать, но рассматривать макросы отдельно от синтаксиса как-то не очень правильно, а синтаксис без примеров разобрать сложно и в результате получилось сочинение о том как писать программы в FASM.

В этом сочинении будут рассмотрены основные моменты отличающие FASM от других компиляторов, правила по которым пишутся макросы и форматы выходных файлов создаваемые FASM-ом. Так же в рамках данного сочинения мы создадим несколько контролов в виде макросов.

Основы.

Данный раздел является вольным
пересказом соответствующего раздела
руководства, поставляемого с
компилятором.

Данная статья посвящена использованию макросов в компиляторе FASM. Для начала вопрос: "Что такое макрос?" (поскольку я не могу услышать Ваш ответ прямо сейчас, когда пишу эту статью, отправляйте свои ответы мне по электронной почте, если Ваше мнение отличается от моего мнения).

Макрос - это инструкция препроцессору компилятора развернуть, встретившееся в коде программы, имя макроса (возможно с параметрами) в последовательность строк кода ассемблера с использованием переданных параметров (если таковые заданы).

Далее предлагаю Вам ознакомиться с основными директивами, используемыми внутри определений макросов.

Внутри определений макросов можно использовать инструкции **if** и **else** при этом каждая инструкция if должна закрываться инструкцией **end if**. Пример из руководства, поставляемого вместе с компилятором:

```
macro mov op1,op2
{
if op1 in  & op2 in
   push op2
   pop op1
else
   mov op1,op2
end if
}
```

Этот макрос является расширением инструкции процессора **mov**. В данном случае если оба операнда макроса являются сегментными регистрами используется связка "push - pop", в любом другом случае используется стандартная инструкция "mov op1,op2". Оператор **in** позволяет проверить соответствие операнда нескольким значениям в угловых скобках.

Можно ещё более расширить эту инструкцию для пересылки трёх значений второго в первый, а третьего во второй:

```
macro mov op1,op2,op3
{
   if arg3 eq
```

```
      mov op1,op2
   else
      mov op1,op2
      mov op2,op3
   end if
}
```

В данном случае нас интересует инструкция **eq** , которая позволяет проверить эквивалентен ли операнд какому либо значению, в нашем случае мы проверяем его отсутствие, и если он отсутствует, то подставляется ветвь **if** иначе подставляется ветвь **else**.

Директива **purge** позволяет отменить последнее определение макроса.

Кроме уже изложенного в определение макроса можно передавать заранее неизвестное количество операндов (пар операндов и т.д.), для этого необходимо заключить операнды, количество которых заранее неизвестно, в квадратные скобки. Например для вызова процедур написан макрос **STDCALL** :

```
macro stdcall proc,[arg];
 { reverse
    push arg
   common
    call proc }
```

Операндами этого макроса служат один обязательный (proc) и несколько необязательных параметров (arg). Директива **reverse** сообщает препроцессору, что следующие строки необходимо повторить столько раз, сколько параметров **arg** передано макросу начиная с последнего параметра. Директива **common** сообщает препроцессору, что следующие строки необходимо повторить только один раз. Директива **forward** сообщает препроцессору, что следующие строки необходимо повторить столько раз, сколько параметров **arg** передано макросу начиная с первого параметра. Действие директив **common** , **forward** и **reverse** заканчивается другой директивой **common** , **forward** или **re-verse** соответственно или закрывающейся фигурной скобкой. Если ни одна из этих директив не встречается в определении макроса, то макрос развернётся для всех параметров начиная с первого. Неизвестное количество параметров можно передать и другому макросу:

```
macro invoke proc,[arg]
 { common
    if ~ arg eq
     stdcall [proc],arg
    else
     call [proc]
    end if }
```

В этом примере при выполнении ветви **if** в макрос **stdcall** кроме параметра **[proc]** передаются все полученные параметры **arg**.

Внутри макроопределения может использоваться оператор **#** , который может использоваться для составления операторов ассемблера, например условного перехода:

```
macro jif op1,cond,op2,label
{
  cmp op1,op2
  j#cond label
}
```

При выполнении макроса jif ax,ae,10h,exit макрос будет развёрнут в следующую конструкцию:

```
cmp ax,10h
jae exit
```

Так же этот оператор можно использовать и для составления имён переменных или макросов внутри макроопределений:

```
macro ver [name]
```

```
{
 name#.mas: times 10 db 0
}
```

Теперь при вызове данного макроса в секции данных

```
ver Chif,Rab
```

Мы будем иметь два массива из 10 байт каждый, с именами Chif.mas и Rab.mas.

Аналогично можно создать и переменные и макросы внутри определения макроса. Однако определить макрос внутри макроса обычным путём:

```
Macro mac1
{
 macro mac2
{
  ... ... ...
}
}
```

препроцессор не позволит использовать данную конструкцию, выход в использовании директивы **fix** она эквивалентна директиве **equ** , но препроцессор обрабатывает **fix** позже чем **equ** , что и позволяет использовать подобную конструкцию:

```
 Macro mac1
{
 macro mac2
m_
  ... ... ...
_m
}
m_ fix {
_m fix }
```

Синтаксис компилятора ФАСМ несколько отличается от синтаксиса других компиляторов, но отличия незначительны и в основном касаются макроязыка.

Средства макроязыка мы рассмотрим ниже, здесь же я хотел бы отметить только то, что передача адреса переменной производится указанием имени переменной без каких либо скобок или префиксов, а её содержимого написанием имени переменной в квадратных скобках.

mov eax,var; записываем в регистр eax адрес переменной var mov eax,[var]; записываем в регистр eax значение переменной var

Структура каркасного приложения на ФАСМ-е

В этой части мы рассмотрим, основы
синтаксиса данного компилятора и
рассмотрим каркасные приложения для
основных форматов исполняемых файлов.

Итак, что необходимо для работающего приложения?

Во-первых мы должны указать компилятору какого формата должен быть создаваемый файл. Это делается директивой **format** после, которой следует указание форматов:

• PE формат исполняемых файлов Windows. Далее должно следовать уточнение **GUI** (графический интерфейс), **console** (консольное приложение) и **native** (если Вы знаете, что это такое сообщите мне). Если выбран графический интерфейс, то следует также указать версию графического интерфейса 4.0, а так же зарезервированное слово DLL, если собирается динамическая библиотека.
• **MZ** формат исполняемых файлов **MS DOS**.
• **COFF** или **MS COFF** (для линковки продуктами Microsoft)для создания объектного файла, который в дальнейшем будет линковаться к другому проекту или к ресурсам.

• **ELF** формат для создания исполняемых файлов UNIX подобных систем (Linux).

Как Вы наверно поняли, я не знаком с файлами ELF типа и в данной статье мы не будем его рассматривать. Если директива **format** не указана, компилятор соберёт файл в формате com.

После указания формата выходного файла мы должны указать компилятору точку начала программы. Это делается с помощью директивы **entry** после которой указывается метка, с которой начинается код программы. Например **entry start**.

Так же, в случае необходимости, мы должны указать на начало секций данных, кода, импорта, экспорта или ресурсов. Для этого существует директива **section** , после которой в кавычках следует название секции, далее без скобок тип секции (данные или код) и атрибуты (запись, чтение исполняемый код). Более подробно когда какие атрибуты указывать Вы прочитаете ниже т.к. для разных форматов выходных файлов эти атрибуты отличаются.

_Форматы PE исполняемых файлов Windows.
Консольные приложения_

Консольное приложение может использовать все функции API предоставляемые Windows, но работать с такой программой пользователь может только через командную строку.

Вот простой пример консольного приложения:

```
format PE console
entry start
include 'include\kernel.inc'
include 'include\macro\stdcall.inc'
include 'include\macro\import.inc'
section '.data' data readable writeable
  h4    db Консоль'
  w1 dd 0
  output dd 0
  input dd 0
  _class db 'FCAPTION',0
  podskazka db 'Консоль',0
section '.code' code readable executable
  start:
 invoke    GetStdHandle,STD_OUTPUT_HANDLE
 mov       [output],eax
 invoke    GetStdHandle,STD_INPUT_HANDLE
 mov       [input],eax
 invoke    WriteConsole,[output],h4,7,w1,0
 invoke    ReadConsole,[input],_class,2,w1,0
 invoke ExitProcess,0
section '.idata' import data readable

  _library kernel32,'KERNEL32.DLL',\
      user32,'USER32.DLL',\
      gdi32,'GDI32.DLL',\
      advapi32,'ADVAPI32.DLL',\
      comctl32,'COMCTL32.DLL',\
      comdlg32,'COMDLG32.DLL',\
      shell32,'SHELL32.DLL',\
      wsock32,'WSOCK32.DLL';,\
     include 'C:\FASM135\147\INCLUDE\apia\kernel32.inc'
     include 'C:\FASM135\147\INCLUDE\apia\user32.inc'
     include 'C:\FASM135\147\INCLUDE\apia\gdi32.inc'
     include 'C:\FASM135\147\INCLUDE\apia\advapi32.inc'
     include 'C:\FASM135\147\INCLUDE\apia\comctl32.inc'
     include 'C:\FASM135\147\INCLUDE\apia\comdlg32.inc'
     include 'C:\FASM135\147\INCLUDE\apia\shell32.inc'
     include 'C:\FASM135\147\INCLUDE\apia\wsock32.inc'
```

В данном примере создаётся три секции: данных, кода и секция импортируемых функций. Секцию импорта я создаю при помощи макроса **_library** , в стандартной поставке такого макроса нет, но если написать без подчёркивания перед **library** то всё скомпилируется правильно. Причина проста,

я пользуюсь FASM-ом с версии 1.35, а тогда такого макроса еще не было (а сам я ещё не смог бы его написать) и для совместимости с ранее написанными программами я включил два варианта этого макроса. В ранней версии, необходимо было записывать все используемые в приложении функции вручную, теперь же работу по подключению только используемых функций берёт на себя компилятор, хотя за такое удобство приходится платить некоторым увеличением времени компиляции. Более подробно о принципе работы этого и других макросов будет написано в соответствующем разделе ниже.

Название секции импорта **".idata"** , хотя название можно изменить и это не повлияет на работоспособность программы, т.к. мы указали назначение этой секции **import data** , далее следует указание атрибутов, в данном случае **readable** , что означает для чтения и этого достаточно, если Вы не собираетесь менять содержимое этой секции.

Следующей рассмотрим секцию данных. У нас секция данных имеет название **".data"** , далее следует указание назначения секции **data** , что означает секцию данных, атрибуты указаны **readable writeable** , т.е. эта область памяти будет доступна для чтения и записи. Если указать только первый атрибут (**readable**) при выполнении программы произойдёт ошибка "Программа выполнила недопустимую операцию".

Последней рассмотрим секцию кода, она имеет название **".code"** назначение секции **code** , т.е. исполняемая часть программы, атрибуты **readable executable** , что означает, что из секции кода мы можем читать данные (под данными может подразумеваться и коды инструкций и определённые в этой секции данные) и можем выполнять содержимое этой секции.

Кроме рассмотренных секций, консольное приложение может иметь секции экспортируемых функций и секцию ресурсов.

Это пожалуй всё, что необходимо знать для начала о консольных приложениях. Да ещё я считаю лишний раз напомнить, что хоть консольное приложение и напоминает внешне программы DOS-а, использование прерываний в нём не допускается т.к. это всё же приложение Windows, естественно выполняться такое приложение может только в Windows.

Графическое приложение

Графические приложения Windows это оконные приложения, такие как Word, Excel, Notepad и т.д.

В отличии от консольных приложений, GUI приложения должны содержать как минимум одну процедуру обработки сообщений, приходящих к окну. Вот простой пример оконного приложения:

```
format PE GUI 4.0
entry start
include 'include\kernel.inc'
include 'include\user.inc'
include 'include\macro\stdcall.inc'
include 'include\macro\import.inc'
include 'include\macro\resource.inc'
section '.data' data readable writeable
  Avtor db 'Автор:Пороткин Алексей Сергеевич',0
  Progr db 'Программа построения графиков',0
  MyMessage db 'Отклик',0
  hinstance dd 0
  h4    db 'Сообщение',0
  w1 dd 0
  msg MSG
  wc WNDCLASS
  pt dd 0
  hDC dd 0
  ws equ  WS_VISIBLE+WS_CAPTION+WS_MINIMIZEBOX+WS_SYSMENU
  _class db 'FCAPTION',0
  podskazka db 'Графическое приложение',0
section '.code' code readable executable
  start:
 invoke GetModuleHandle,0
```

```
 mov [hinstance],eax
 invoke LoadIcon,eax,17
 mov [wc.hIcon],eax
 invoke LoadCursor,0,IDC_ARROW
 mov [wc.hCursor],eax
 mov [wc.style],0
 mov [wc.lpfnWndProc],WindowProc
 mov [wc.cbClsExtra],0
 mov [wc.cbWndExtra],0
 mov eax,[hinstance]
 mov [wc.hInstance],eax
 mov [wc.hbrBackground],6;COLOR_BTNFACE+1
 mov [wc.lpszMenuName],0
 mov [wc.lpszClassName],_class
 invoke RegisterClass,wc
 invoke CreateWindowEx,0,_class,podskazka,ws,10,10,680,450,NULL,NULL,[hinstance],NULL

  msg_loop:
 invoke GetMessage,msg,NULL,0,0
 or eax,eax
 jz end_loop
 invoke TranslateMessage,msg
 invoke DispatchMessage,msg

 jmp msg_loop

  end_loop:
 invoke ExitProcess,[msg.wParam]
;include 'include\macro\lib.asm'
proc WindowProc, hwnd,wmsg,wparam,lparam
 enter
 push ebx esi edi
          cmp [wmsg],WM_DESTROY
 je wmdestroy
 cmp [wmsg],WM_CREATE
 je wmcreate
  defwndproc:
 invoke DefWindowProc,[hwnd],[wmsg],[wparam],[lparam]
 jmp finish
wmcreate:

 jmp ext

  wmdestroy:
 invoke PostQuitMessage,0
ext: xor eax,eax
  finish:
 pop edi esi ebx
 return
section '.idata' import data readable writeable
  library kernel,'KERNEL32.DLL',\
   user,'USER32.DLL'
   ;shell,'SHELL32.DLL',\
   ;gdi,'gdi32.dll'
  kernel:
  import GetModuleHandle,'GetModuleHandleA',\
  ExitProcess,'ExitProcess',lstrcpy,'lstrcpyA',\
  FindFirstChangeNotification,'FindFirstChangeNotificationA',\
  FindNextChangeNotification,'FindNextChangeNotification',\
  FindCloseChangeNotification,'FindCloseChangeNotification'
  user:
  import RegisterClass,'RegisterClassA',\
  CreateWindowEx,'CreateWindowExA',\
  DefWindowProc,'DefWindowProcA',\
  SetWindowLong,'SetWindowLongA',\
  RedrawWindow,'RedrawWindow',\
  GetMessage,'GetMessageA',\
  MessageBox,'MessageBoxA',\
  TranslateMessage,'TranslateMessage',\
  DispatchMessage,'DispatchMessageA',\
  SendMessage,'SendMessageA',\
```

```
    LoadCursor,'LoadCursorA',\
    LoadIcon,'LoadIconA',\
    LoadMenu,'LoadMenuA',\
    SetMenu,'SetMenu',\
    InsertMenu,'InsertMenuA',\
    CreateMenu,'CreateMenu',\
    DrawMenuBar,'DrawMenuBar',\
    PostQuitMessage,'PostQuitMessage',\
    FindWindow,'FindWindowA',\
    FindWindowEx,'FindWindowExA',\
    LoadImageA,'LoadImageA',\
    CreatePopupMenu,'CreatePopupMenu',\
    AppendMenu,'AppendMenuA',\
    DestroyMenu,'DestroyMenu',\
    GetCursorPos,'GetCursorPos',\
    SetForegroundWindow,'SetForegroundWindow',\
    TrackPopupMenu,'TrackPopupMenu',\
    PostMessage,'PostMessageA',\
    DestroyWindow,'DestroyWindow',\
    ShowWindow,'ShowWindow',GetDC,'GetDC'
  section '.rsrc' resource data readable
    directory   RT_ICON,icons,\
       RT_GROUP_ICON,group_icons
    icons:
    resource 1,LANG_NEUTRAL,icon_data
    group_icons:
    resource 17,LANG_NEUTRAL,main_icon
    icon main_icon,icon_data,'mp\lv\lv.ico'
```

Так же, как и в примере с консольным приложением, в первой строке указана директива **format PE** , но вместо **console** , как в предыдущем примере, дано указание **GUI 4.0** компилятору собирать приложение с графическим интерфейсом версии 4.0.

Приложение из нашего примера ничего полезного не делает, оно просто создаёт окно, которое реагирует только на нажатие двух системных кнопок в верхнем правом углу окна "Свернуть" и "Закрыть". Приложение содержит секции данных, кода, импорта и ресурсов. После указания точки входа в программу entry start подключаются файлы содержащие макросы и определения констант и структур Windows. Затем следует секция данных, определение которой дано в предыдущем разделе, и секция кода. В секции кода мы получаем хендл нашего модуля (**GetModuleHandle**), загружаем иконку (**LoadIcon**), определённую в секции ресурсов, загружаем стандартный курсор Windows (**LoadCursor**) и заполняем структуру **wc** для дальнейшей регистрации класса нашего окна (**RegisterClass**). После регистрации класса окна мы создаём окно зарегистрированного класса (**CreateWindowEx**). В структуре wc имеется член .lpfnWndProc, в который мы записываем адрес начала процедуры обработки событий окна **WindowProc**. Далее следует цикл обработки сообщений, посылаемых окну системой Windows.

Процедура обработки сообщений получает от Windows четыре параметра:

proc WindowProc, hwnd, wmsg, wparam, lparam

где:

hwnd - хендл нашего окна
wmsg - код сообщения, которое Windows послало нашему окну
wparam и **lparam** - уточняющие параметры посланного сообщения.

proc это не директива компилятора, а макрос содержащий, часть пролога процедуры. Он определён следующим образом:

```
  macro proc name,[arg]
   { common
      name:
      virtual at ebp+8
      if ~ arg eq
     forward
```

```
    local ..arg
    ..arg dd ?
    arg equ ..arg
  common
    end if
    ..ret = $ - (ebp+8)
  end virtual
  local ..dynamic_data,..dynamic_size
  dynamic_data equ ..dynamic_data
  dynamic_size equ ..dynamic_size
  virtual at ebp - dynamic_size
   dynamic_data: }
```

Таким образом следующая строка **proc Win,param** будет развёрнута в следующую конструкцию:

```
  Win:
  virtual at ebp+8
      local ..param
      ..param dd ?
      param equ ..param
      ..ret = $ - (ebp+8)
  end virtual
  local ..dynamic_data,..dynamic_size
  dynamic_data equ ..dynamic_data
  dynamic_size equ ..dynamic_size
  virtual at ebp - dynamic_size
   dynamic_data: }
```

Более подробно это означает, что мы определяем метку **Win** , которая означает начало процедуры. В следующей строке записана директива **virtual at ebp+8** , которая должна заканчиваться директивой **end virtual**. Внутри этого определения помещаются переменные, которые будут размещаться с адреса указанного после **at** , в нашем примере это **ebp+8**. Это означает, что в нашем примере запись "**mov param,eax** " будет заменена на "**mov [ebp+8],eax** ". Если бы в определении процедуры было две или более переменных, то адрес каждой последующей переменной увеличивался бы на размер предыдущей переменной. Таким образом:

```
virtual at ebp+8
   param1 dd ?
   param2 db ?
   param3 dd ?
end virtual
```

определит три переменные с адресами

```
param1 = ebp+8+0,
param2 = ebp+8+4,
param2 = ebp+8+5.
```

У некоторых читателей может возникнуть вопрос: "А зачем все эти пляски?". Ответ прост: регистр **ebp** указывает на вершину стека и все переменные, передаваемые в процедуру, находятся выше (в смысле в памяти с адресами больше чем вершина стека т.к. вершина стека сдвигается вниз когда в стек помещаются данные и вверх, когда данные из стека изымаются) текущей вершины стека.

Первой, в определении переменных внутри **virtual - end virtual** записана директива **local** , которая указывает компилятору, что следующие за ней через запятую метки, локальны в данном макросе.

Последней, внутри описания виртуальных переменных, следует определение макропеременной **..ret = $ - (ebp+8)** , которая фактически содержит количество переданных процедуре байт параметров. Она используется при написании эпилога процедуры для очистки стека при возврате управления инструкции, следующей за вызовом данной процедуры.

После определения виртуальных переменных, передаваемых процедуре в качестве параметров, определяются локальные макропеременные, которые используются во второй части пролога процедуры. Я уже несколько раз использовал термин - **макропеременная**. Макропеременная отличается от переменных, используемых в программе, тем, что она существует только во время

компиляции и используется как константа, которая может иметь разные значения в разных частях программы.

После определения макропеременных начинается блок определения виртуальных переменных, однако переменных в этом макросе нет. После макроса **proc** необходимо ставить макрос **enter** , который содержит заключительную часть пролога, а между ними могут располагаться локальные переменные. Для того, что бы имена переменных были локальными перед ними ставится точка. Например **.param**. Макрос **enter** определён следующим образом:

```
macro enter
{  dynamic_size = $ - dynamic_data
   end virtual
   enter dynamic_size,0
}
```

Таким образом **dynamic_size** - это размер памяти отведённой под локальные переменные. Последняя инструкция, в прологе процедуры, **enter dynamic_size,0**. Это единственная строка, которая попадёт в исполняемый код программы. Завершаться процедура должна макросом return, который определён следующим образом:

```
macro return
{ leave
   ret ..ret }
```

Полагаю здесь не должно быть сложностей с пониманием макроса.

Процедура должна правильно обрабатывать поступившие сообщения. Те сообщения, которые мы хотим обработать сами, выделяются в первых строках процедуры и управление передаётся на те участки кода, которые должны производить необходимые действия, а те сообщения, которые нам нет необходимости обрабатывать, мы доверяем обработать системе Windows (**Def-WindowProc**).

Хочу обратить Ваше внимание на то, что в данной программе используются глобальные метки (**wmdestroy** , **wmcreate** , **defwndproc**), что не очень удобно в программах содержащих несколько процедур (большинство оконных процедур обрабатывают одни и те же сообщения), поэтому рекомендуется внутри процедур использовать локальные метки (**.wmdestroy** , **.wmcreate** , **.defwndproc**).

Теперь пришло время посмотреть на секцию импорта. Она отличается от рассмотренной в предыдущем разделе здесь мы используем макрос **library** , который определён следующим образом:

```
macro library [label,string]
 { forward
    local _label
    dd 0,0,0,rva _label,rva label
   common
    dd 0,0,0,0,0
   forward
    _label db string,0 }
```

Для того, чтобы было более понятно вспомним, как в нашем примере используется этот макрос:

```
library kernel,'KERNEL32.DLL',\
   user,'USER32.DLL'
```

Таким образом, макрос **library** создаёт заголовок таблицы импорта, вначале записывая смещение названия библиотеки DLL, а затем смещение, с которого начинаются имена импортируемых функций.

Далее следуют макросы **import** , определённые следующим образом:

```
macro import [label,string]
 { forward
    local _label
```

```
    label dd rva _label
   common
    dd 0
   forward
    _label dw 0
    db string,0 }
```

Он создаёт тело таблицы импорта для каждой из импортируемых библиотек.

Вначале создавая таблицу смещений имён импортируемых функций, а затем записывая имена импортируемых функций. Более подробно о структуре PE-файла Вы должны прочитать в соответствующей литературе (например в туториалах, которые написал Iczelion).

После таблицы импорта создаётся таблица ресурсов:

```
section '.rsrc' resource data readable
  directory   RT_ICON,icons,\
     RT_GROUP_ICON,group_icons
  icons:
  resource 1,LANG_NEUTRAL,icon_data
  group_icons:
  resource 17,LANG_NEUTRAL,main_icon
  icon main_icon,icon_data,'mp\lv\lv.ico'
```

В нашем примере мы используем только одну иконку.

Динамические библиотеки (DLL)

Последним в данном разделе рассмотрим простую динамическую библиотеку.

Динамическая библиотека - это приложение GUI, который может содержать как исполняемый код, так и ресурсы. Это приложение не предназначено для самостоятельного запуска, оно используется другими приложениями для хранения часто используемых процедур, шрифтов и многого другого.

Мы рассмотрим создание динамической библиотеки на примере процедуры, возвращающей строку, содержащую текущую дату. Мы создадим два модуля один файл динамической библиотеки (dll2.dll) и файл использующий созданную нами библиотеку (usedll.exe).

Вот файл динамической библиотеки:

```
format PE GUI 4.0 DLL
entry DllEntryPoint
include 'include\kernel.inc'
include 'include\user.inc'
include 'include\macro\stdcall.inc'
include 'include\macro\import.inc'
include 'c:\fasm135\include\lib\strings.mac'
include 'include\macro\export.inc'
section '.data' data readable writeable
  _outs: times 60 db 0
  Date SYSTEMTIME
  day db 'День:',0
  mon db ' Месяц:',0
  year db ' Год:',0
  h4: times 20 db 0
  DllState db 'Состояние библиотеки DLL',0
  DllStart db 'Загружена',0
  DllStop  db 'Выгружена',0
  bf: times 20 db 0
section '.code' code readable executable
proc DllEntryPoint, hinstDLL,fdwReason,lpvReserved
 enter
 cmp       [fdwReason],DLL_PROCESS_ATTACH
 je    .start
 cmp       [fdwReason],DLL_PROCESS_DETACH
 je    .stop
 jmp   .stp
 .start:
 invoke    MessageBox,0,DllStart,DllState,MB_OK
```

```
 jmp   .stp
 .stop:
 invoke    MessageBox,0,DllStop,DllState,MB_OK
 jmp   .stp
   .stp:
 mov eax,TRUE
 return
proc Wind
enter
 push ebx esi edi
 invoke  GetLocalTime,Date
 mov  ebx,_outs
 StringCopyHTML ebx,day
 add  ebx,eax
 xor  eax,eax
 mov  ax,[Date.wDay]
 stdcall  IntToString,h4,eax
 StringCopyHTML ebx,h4
 add  ebx,eax
 StringCopyHTML ebx,mon
 add  ebx,eax
 xor  eax,eax
 mov  ax,[Date.wMonth]
 stdcall  IntToString,h4,eax
 StringCopyHTML ebx,h4
 add  ebx,eax
 StringCopyHTML ebx,year
 add  ebx,eax
 xor  eax,eax
 mov  ax,[Date.wYear]
 stdcall  IntToString,h4,eax
 StringCopyHTML ebx,h4
 mov  eax,_outs
 pop edi esi ebx
return
proc IntToString,lpOutBuf,var
   len dd 0
       enter
       push ebx esi edx ecx esi
       ;jmp .ex
       cmp  [var],0
       jne  .norr
       mov   eax,[lpOutBuf]
       mov   dx,030h
       mov   [eax],dx
       jmp  .ex
       .norr:
       mov  eax,10
       mov  [len],eax
       xor eax,eax
       xor edx,edx
       xor ecx,ecx
       mov  esi,bf
       mov  eax,[var]
       cmp eax,0
       jne .ll
       mov eax,48
       mov [esi],al
       inc esi
       jmp .ext
 .ll:
       div  [len]
       add  dl,48
       mov  [esi],dl
       inc  esi
       inc ecx
       xor  edx,edx
       cmp  eax,0
       jne   .ll
 .ext:
      mov [esi],dl
```

```
      mov eax,[lpOutBuf]
      cmp ecx,0
      je .ex
  .l: dec esi
      mov dl,[esi]
      mov [eax],dl
      inc  eax
      dec  ecx
      cmp  ecx,0
      jne .l
      mov  dl,0
      mov  [eax],dl
 .ex:     pop   esi ecx edx esi ebx
return
section '.idata' import data readable writeable
     _library kernel32,'KERNEL32.DLL',\
      user32,'USER32.DLL';,\
     include 'C:\FASM135\147\INCLUDE\apia\kernel32.inc'
     include 'C:\FASM135\147\INCLUDE\apia\user32.inc'
section '.edata' export data readable
  export 'dll2.dll',Wind,'Wind'
section '.reloc' fixups data discardable
```

Первые строки **format PE GUI 4.0 DLL** и **entry DllEntryPoint** указывают компилятору создать PE-файл динамической библиотеки с точкой входа на метке **DllEntryPoint**.

Как Вы могли заметить динамическая библиотека практически ни чем не отличается от обычной программы Windows, кроме того, что входная процедура вызывается системой в четырёх случаях:

• Когда динамическая библиотека загружается в адресное пространство процесса (параметр fdwReason = DLL_PROCESS_ATTACH = 1);
• Когда динамическая библиотека выгружается из адресного пространства процесса (параметр fdwReason = DLL_PROCESS_ DETACH = 0);
• Когда создается новый поток в рамках динамической библиотеки (параметр fdwReason = DLL_ THREAD_ATTACH = 2);
• Когда разрушается (прекращается) поток в рамках динамической библиотеки (параметр fdwReason = DLL_ THREAD_DETACH = 3).

В обработке стартовой процедуры мы определяем значение параметра fdwReason и если динамическая библиотека была загружена или выгружена выдаём соответствующее сообщение.

В конце файла мы создаём секцию экспорта нашей библиотеки:

```
section '.edata' export data readable
  export 'dll2.dll',Wind,'Wind'
section '.reloc' fixups data discardable
```

В таблице мы задаём имя файла динамической библиотеки и имена экспортируемых процедур.

Макрос export определён следующим образом:

```
macro export dllname,[label,string]
{ common
    local module,addresses,names,ordinal,count
    count = 0
   forward
    count = count+1
   common
    dd 0,0,0,rva module,1
    dd count,count,rva addresses,rva names,rva ordinal
    addresses:
   forward
    dd rva label
   common
    names:
   forward
    local name
    dd rva name
```

```
    common
     ordinal: count = 0
    forward
     dw count
     count = count+1
    common
     module db dllname,0
    forward
     name db string,0 }
```

Сперва, объявляются локальные метки и подсчитывается количество экспортируемых процедур:

```
   common
     local module,addresses,names,ordinal,count
     count = 0
    forward
     count = count+1
    common
```

Затем заносится смещение RVA имени библиотеки:

```
     dd 0,0,0,rva module,1
```

Следующая строка заносит общее количество экспортируемых функций, количество функций экспортируемых по имени (при использовании этого макроса все функции экспортируются по имени), смещение начала массива смещений функций, смещение начала массива смещений имен экспортируемых функций и смещение массива смещений ординалов функций.

```
     dd count,count,rva addresses,rva names,rva ordinal
```

Далее объявляется метка начала массива смещений экспортируемых функций и создаётся массив их смещения:

```
addresses:
   forward
    dd rva label
```

Следом объявляется метка начала массива смещений имён экспортируемых функций и создаётся сам массив:

```
common
    names:
   forward
    local name
    dd rva name
```

Затем объявляется метка начала массива смещений ординалов функций и создаётся массив ординалов экспортируемых функций:

```
common
    ordinal: count = 0
   forward
    dw count
    count = count+1
```

Ну и в конце макроса записывается имя динамической библиотеки и создаётся массив имен экспортируемых функций:

```
common
    module db dllname,0
   forward
    name db string,0
```

Для более подробного ознакомления со структурой секции экспорта необходимо читать специальную литературу по структуре PE - файлов.

Теперь давайте посмотрим, на программу вызывающую нашу динамическую библиотеку:

```
format PE GUI 4.0
```

```
entry start
include 'include\kernel.inc'
include 'include\user.inc'
include 'include\macro\stdcall.inc'
include 'include\macro\import.inc'
section '.data' data readable writeable
  Avtor db 'Автор:Пороткин Алексей Сергеевич',0
  podskazka db 'Функция Wind возвратила указатель на строку:',0
section '.code' code readable executable
  start:
   invoke Wind
   invoke MessageBox,0,eax,podskazka,MB_OK
   invoke ExitProcess,0
section '.idata' import data readable writeable
  library kernel,'KERNEL32.DLL',\
   user,'USER32.DLL',\
   MyDll,'C:\FASM135\mp\Dll2\Dll2.DLL'
  kernel:
  import   ExitProcess,'ExitProcess'
  user:
  import   MessageBox,'MessageBoxA'
 MyDll:
  import Wind,'Wind'
```

Обычное графическое приложение не имеющее своего окна и ресурсов размером 2 кБ. Импортирует только три функции две предоставляемые системой Windows и одна написанной нами динамической библиотекой. Как видите подключение созданной нами динамической библиотеки, ничем не отличается от подключения стандартных библиотек Windows. В этом и заключается вся прелесть использования ФАСМ-а, он не берёт на себя слишком много, что в купе с гибким макроязыком позволяет сделать очень многое максимально просто.

Формат исполняемых файлов MS COFF (obj - файлы)

Использование obj-файлов в ФАСМ, лично мне, не кажется жизненно необходимым, однако использование этого формата возможно понадобится при создании проектов на C/C++. Именно на таком примере мы и рассмотрим формат файлов MS COFF.

Для примера напишем программу, выводящую МессаджБокс с текущей датой. Модуль на ассемблере будет составлять строку символов, а модуль на C++ будет выводить эту строку в МессаджБокс.

Вот модуль на ассемблере:

```
format MS COFF
;public _WinMainCRTStartup
public _outs
public ?Wind@@YAPADXZ
extrn '__imp__GetLocalTime@4' as GetLocalTime:dword
include 'include\kernel.inc'
include 'include\macro\stdcall.inc'
include 'c:\fasm135\include\lib\strings.mac'
include 'c:\fasm135\include\lib\strings.asm'
section '.text' code readable executable
 ; _WinMainCRTStartup:
proc ?Wind@@YAPADXZ
 enter
 push ebx esi edi
 invoke   GetLocalTime,Date
 mov      ebx,_outs
 StringCopyHTML ebx,day
 add      ebx,eax
 xor      eax,eax
 mov      ax,[Date.wDay]
 stdcall  IntToString,h4,eax
 StringCopyHTML ebx,h4
 add      ebx,eax
 StringCopyHTML ebx,mon
 add      ebx,eax
```

```
 xor      eax,eax
 mov      ax,[Date.wMonth]
 stdcall  IntToString,h4,eax
 StringCopyHTML ebx,h4
 add      ebx,eax
          StringCopyHTML ebx,year
 add      ebx,eax
 xor      eax,eax
 mov      ax,[Date.wYear]
 stdcall  IntToString,h4,eax
 StringCopyHTML ebx,h4
 add      ebx,eax
 mov      eax,_outs
 pop edi esi ebx
 return
section '.data' data readable writeable
  h4    db 'Сообщение',0
  Date SYSTEMTIME
  day db 'День:',0
  mon db ' Месяц:',0
  year db ' Год:',0
  _outs: times 30 db 0
```

Обратите внимание на строку 2, где закомментировано определение точки входа в программу (_WinMainCRTStartup)если эту строку раскомментировать, то точка входа в программу будет именно в модуле на ассемблере. Результат работы функции записывается в переменную _outs и адрес (указатель на строку) возвращаем в регистре eax. В этом модуле используются макросы копирования строки в строку (StringCopyHTML ebx,day). Этот макрос не входит в поставку компилятора и содержится в файлах strings.mac и strings.asm он просто копирует строку (не включая нулевой символ), указатель на которую передаётся вторым параметром, в строку, указатель которой передаётся вторым параметром, после его выполнения в регистре eax количество скопированных символов. Эта процедура выдаёт строку типа: "День: 28 Месяц: 10 Год:2003". Для использования объектного модуля созданного на ассемблере необходимо в проект на C++ добавить созданный нами объектный файл (Project->Add to project->Files. . . в появившемся окне необходимо выбрать тип файлов obj и указать используемый файл). Теперь рассмотрим текст программы на MS Visual C++ 5.0:

```
#include «windows.h»
extern  char *  __cdecl Wind(void);
LPSTR cmsk,dat;

int PASCAL WinMain(HANDLE   hInstance,HANDLE hPrevInstance,LPSTR lpszCmdParam,int nCmdShow)
{
 dat=Wind();
 MessageBox(0,dat,dat,MB_OK);
}
```

Здесь ничего сложного: подключаем файл «windows.h», затем объявляем внешнюю процедуру, возвращающую указатель на строку символов с нулём в конце, объявляем переменную dat, как указатель на строку символов, и наконец функция WinMain в которой вызываем нашу функцию (Wind) на ассемблере и выводим результат в мессаджбокс. Внимательный читатель скажет: "Стоп, но в модуле на ассемблере функцию мы объявили как ?Wind@@YAPADXZ, а в модуле на C++ вызываем Wind". Всё правильно. Просто MS Visual C++ 5.0, к объявленным внешним (находящимся в скомпилированных модулях) процедурам добавляет дополнительные символы, для контроля соответствия типов передаваемых результатов из процедуры и переменных в процедуру. Вот краткая структура имени функций формируемых MS Visual C++:

Префикс	Объявленное имя переменной	Разделитель	Тип вызова процедуры	Тип результата	Тип параметров	Постфикс
?	Wind	@@Y	A	PAD	X	Z

Причём постфикс изменяется в зависимости от наличия передаваемых в функцию параметров. Если параметры передаются, то постфикс "@Z", а если параметры не передаются то просто "Z".

Типы вызова процедуры следующие:

__cdecl	**A**
__fastcall	I
__stdcall	G

Типы результата и передаваемых переменных имеют одинаковые обозначения:

Тип результата	**Размер**	**Обозначение**
char	1	D
unsigned char	1	E
short	2	F
unsigned short	2	G
enum	2	?AW4__unnamed@@
long	4	J
unsigned long	4	K
int	4	H
unsigned int	4	I
float	4	M
double	8	N
long double	10	O
bool	1	_N

Причём если передаётся (получается) не параметр, а его адрес (например char *) то перед обозначением переменной добавляется PA. Если параметры, получаемые из процедуры или передаваемые в процедуру, отсутствуют, вместо них ставится X.

Читатель может сказать: "Это как же нужно мозг морщить, чтобы всё запомнить и не перепутать ничего при составлении имени процедуры?". Такая работа имеет основное свойство: она не особо интеллектуальна, но муторна и действия при её решении вполне укладываются в рамках алгоритма. Отсюда вывод: компьютер на то и создан, что бы выполнять подобные задачи так пусть сам, и добавляет все эти префиксы, постфиксы и т.п. Напишем макрос, который будет объявлять процедуру составляя её имя из имени которое мы ему передаём и добавляя к нему все необходимые префиксы.

Использовать переменные длиннее 4 байт мы не будем, но думаю Вы сами сможете модернизировать макросы под свои нужды если понадобится.

Для начала мы должны определить символьные константы, соответствующие типам вызова процедуры и типам получаемых и возвращаемых параметров:

```
; константы типов вызовов процедур
__cdecl fix A
__fastcall fix I
__stdcall fix G
; константы типов переменных
void fix X
char$ fix PAD
char  fix D
unsigned_char$ fix PAE
unsigned_char fix E
```

```
short$ fix PAF
short fix F
unsigned_short$ fix PAG
unsigned_short fix G
long$ fix PAJ
long fix J
unsigned_long$ fix PAK
unsigned_long fix K
int$ fix PAH
int fix H
unsigned_int$ fix PAI
unsigned_int fix I
float$ fix PAM
float fix M
```

Далее следует определение макроса:

```
macro exp_proc calltype,returns,name,[arg,type]
{
dr fix
forward
 dr fix dr#type
 common
if type eq void
public ?#name#@@Y#calltype#returns#dr#Z
?#name#@@Y#calltype#returns#dr#Z:
else
public ?#name#@@Y#calltype#returns#dr#@Z
?#name#@@Y#calltype#returns#dr#@Z:
end if
common
 virtual at ebp+8
    if ~ arg eq
   forward
     local ..arg
     ..arg dd ?
     arg equ ..arg
   common
     end if
     ..ret = $ - (ebp+8)
    end virtual
    local ..dynamic_data,..dynamic_size
    dynamic_data equ ..dynamic_data
    dynamic_size equ ..dynamic_size
    virtual at ebp - dynamic_size
     dynamic_data:
}
```

В первой строке мы очищаем временную символьную константу, далее следует указание препроцессору обработать строку "dr fix dr#type" для всех значений параметров type начиная с первого. Эта строчка, просто создаёт константу содержащую все символы соответствующие всем получаемым процедурой параметрам. Например если в процедуру передаётся два параметра типа int, то dr будет эквивалентно HH. Далее следует указание препроцессору следующие строки обработать только один раз (common) в этих строках мы делаем видимой из других модулей программы метку (название) нашей процедуры и собственно следует содержимое макроса proc, рассмотренного ранее.

Полагаю, этот макрос не должен вызвать затруднений. Сложности в понимании может вызвать только строка "?#name#@@Y#calltype#returns#dr#@Z", объясню её более подробно. Сперва стоит знак вопроса (посмотрите табличку со структурой имени процедуры), после которого, значок номера (#), он говорит препроцессору, что далее следует символьная константа, которую следует склеить в одну метку с предыдущим текстом. Если константы с указанным именем не существует, то она включается, как есть. Рассмотрим последовательность действий препроцессора. Метка состоит из вопросительного знака, затем (без разделителей) добавляется, передаваемая макросу переменная name, после неё добавляется текст @@Y,после текста добавляется передаваемая

макросу переменная calltype, затем переменная returns, потом составленная нами константа dr, и наконец постфикс @Z (или просто Z, если в процедуру не передаются параметры).

Всё. Макрос создан. Однако это ещё не всё. С таким определением процедуры не всегда можно использовать стандартные макросы enter и return т.к. тип вызова __cdecl предполагает, что стек после завершения работы процедуры будет очищаться вызывающей процедурой т.е. модулем написанным на C++, а стандартный макрос return сам очищает стек. Вот макрос, содержащий эпилог процедуры для типа вызова __cdecl:

```
macro cdecl_ret
    {
      sub   ebp,dynamic_size
      mov     esp,ebp
      pop     ebp
      ret
    }
```

Но не только эпилог нужно изменить, но и пролог тоже:

```
macro cdecl_enter
    {
      dynamic_size = $ - dynamic_data
        end virtual
        push  ebp
        mov   ebp,esp
        add   ebp,dynamic_size
    }
```

Полагаю эти два коротеньких макроса не вызовут затруднений.

Ну и пример использования данных макросов:

```
exp_proc __cdecl,char$,Wi,first,int,second,int
 cdecl_enter
   push ebx esi edi
   mov    eax,[first]
   stdcall IntToString, _outs,eax
   mov    eax,_outs
   pop edi esi ebx
 cdecl_ret
```

Функция IntToString преобразует 32-х битное без знаковое число (второй параметр) в строку (адрес строки передаётся первым параметром).

Модуль на C++ будет выглядеть следующим образом:

```
#include
extern  char * __cdecl Wind(void);
extern  char * __cdecl Wi(int c,int b);
LPSTR cmsk,dat;
int PASCAL WinMain(HANDLE   hInstance,HANDLE   hPrevInstance,LPSTR    lpszCmdParam,int
    nCmdShow)
{
 dat=Wi(12,30);
 MessageBox(0,dat,dat,MB_OK);
 Wind();
MessageBox(0,dat,dat,MB_OK);
}
```

Думаю больше пояснений не нужно.

_Формат исполняемых файлов MS DOS
Формат исполняемых файлов MZ_

Программирование под ДОС в ФАСМ в принципе почти ничем не отличается от программирования под ДОС в любом другом компиляторе.

Вот простейший пример программы:

```
format MZ
 push cs
 pop ds
 mov ah,9
 mov dx,hello
 int 21h
 mov ax,4C00h
 int 21h
hello db 'Hello world!',24h
```

Как видите ничего сложного.

Формат исполняемых файлов COM

Если директива format не указана создаются файлы формата *.com. Вот пример:

```
org 100h  ;Указание загрузить программу начиная с адреса 100h
use16  ;Создавать 16-ти битный код
        mov ah,9
        mov dx,hello
        int 21h
        int 20h
hello db 13,10,'Привет всем$'
```

Думаю здесь всё понятно.

8. Защита от отладки

Антиотладочные хитрости под Win32

Автор: Billy Belcebu/iKX, пер. Aquila

Дата: 2002-06-27

Здесь я расскажу о нескольких хитростях, которые можно использовать для защиты своих вирусов и/или своих программ против отладчиков всех уровней, уровня приложения и системы. Я надеюсь, что вам понравится эта статья.

Win98/NT: Обнаружение отладчиков уровня приложения с помощью функции IsDebuggerPresent

Этой функции нет в Win95, поэтому вам следует вначале выяснить, существует ли она, и работает только с отладчиками уровня приложения (таким как TD32). Она работает прекрасно. Давайте посмотрим, что написано о ней в справочнике по Win32 API.

Функция IsDebuggerPresent показывает, запущен ли вызывающий ее процесс в контексте отладчика. Эта функция экспортируется из KERNEL32.DLL.

```
BOOL IsDebuggerPresent(VOID)
```

• Аргументы

У этой функции нет аргументов.

• Возвращаемое значение

• Если текущий процесс запущен в контексте отладчика, возвращаемое значение не равно нулю.
• Если текущий процесс не запущен в контексте отладчика, возвращаемое значение равно нулю.

Пример, демонстрирующий эту функцию очень прост. Вот он:

```
;---[ CUT HERE ]-------------------------------------------------------------

       .586p
       .model flat

extrn   GetProcAddress:PROC
extrn   GetModuleHandleA:PROC

extrn   MessageBoxA:PROC
extrn   ExitProcess:PROC

                .data
szTitle         db      "IsDebuggerPresent Demonstration",0
msg1            db      "Application Level Debugger Found",0
msg2            db      "Application Level Debugger NOT Found",0
msg3            db      "Error: Couldn't get IsDebuggerPresent.",10
                db      "We're probably under Win95",0

@IsDebuggerPresent db   "IsDebuggerPresent",0
K32             db      "KERNEL32",0

       .code

antidebug1:
        push    offset K32                      ; Получаем адрес базы KERNEL32
```

```
        call    GetModuleHandleA
        or      eax,eax                         ; Проверяем на ошибки
        jz      error

        push    offset @IsDebuggerPresent       ; Теперь проверяем, существует
        push    eax                             ; ли IsDebuggerPresent. Если
        call    GetProcAddress                  ; GetProcAddress возвращает
        or      eax,eax                         ; ошибку, мы считаем, что мы
        jz      error                           ; в Win95

        call    eax                             ; Вызываем IsDebuggerPresent

        or      eax,eax                         ; Если она возвращает не 0,
        jnz     debugger_found                  ; нас отлаживают

debugger_not_found:
        push    0                               ; Показываем "Debugger not found"
        push    offset szTitle
        push    offset msg2
        push    0
        call    MessageBoxA
        jmp     exit

error:
        push    00001010h                       ; Show "Error! We're in Win95"
        push    offset szTitle
        push    offset msg3
        push    0
        call    MessageBoxA
        jmp     exit

debugger_found:
        push    00001010h                       ; Show "Debugger found!"
        push    offset szTitle
        push    offset msg1
        push    0
        call    MessageBoxA

exit:
        push    00000000h                       ; Exit program
        call    ExitProcess

end     antidebug1

;---[ CUT HERE ]---------------------------------------------------------------
```

Не правда ли, это красиво? Micro$oft сделал эту работу за нас :). Но, конечно, не ждите, что этот метод будет работать с SoftIce'ом, богом отладки.

Win32: Другой путь, как узнать, что мы находимся в контесте отладчика

Если вы смотрели статью "Wub95 Structure and Secrets", которая была написана Murkry/iKX, и опубликована в Xin-3, вы сообразите, что в регистре FS находится очень крутая структура. Давайте взглянем в поле FS:[20h]... Это 'DebugContext'. Всего лишь сделаем следующее:

```
        mov     ecx,fs:[20h]
        jecxz   not_being_debugger
        [...]
```

<--- делайте, что хотите, нас отлаживают :)

Потому, если FS:[20h] равен нулю, нас не отлаживают. Наслаждайтесь этим маленьким и простым методом для обнаружения отладчиков! Конечно, это не сработает против SoftICE...

Win32: Остановка отладчиков уровня приложения с помощью SEH

Я не знаю почему, но отладчики уровня приложения умирают, если программа использует SEH. Эмуляторы код, если мы делаем ошибки, умирают тоже :). SEH, как я писал в своей статье для DDT#1, используется для многих интересных целей. Хорошо, так как я не хочу рассказывать все это заново, я просто скопирую и вставлю мое старое описание :).

--[DDT#1.2_6]---------- Хорошо, это очень простой туториал о Structured Exception Handler. Когда я увидел SEH, реализованный в вирусе, я подумал "Хорошо, это большая работа. Наверное, это было сложно реализовать". Поэтому я просто пропустил это. Но, как только мой Destiny сделал General Protection Fault'ы под NT, я понял, что я должен сделать что-то. И SEH - это единственный путь. Хорошо, мы можем сделать этоу очень сложным или очень простым для понимания. Конечно, я предпочитаю сделать это попроще :). **% Установка фрейма SEH %**

Сначала мы сохраняем его для нашей собственной безопасности простой строкой кода.

```
push    dword ptr fs:[0]
```

А теперь надо установить указатель на наш обработчик (например, представьте, что мы используем call для вызова настройки SEH, а наш обработчик идет непосредственно после этой инструкции call: мы можем использовать смещение ret).

```
push    offset SEH_Handler
mov     fs:[0],esp
```

Ок, просто как только возможно. Что насчет восстановления исходного SEH. Еще проще. Просто вызовите инструкцию, обратную первой.

```
pop     dword ptr fs:[0]
```

Удивительно, что может сделать эта простая для реализации вещь для наших вирусов. Для меня (и мне было важно именно это использование SEH) наиболее главным было то, что я смог избежать всех этих отвратительных голубых экранов, когда мы запускаем наш Win95-вирус в среде NT.

% Пример использования SEH %

Мы можем скомпилировать данный пример следующим образом:

```
 tasm32 /m3 /ml sehtest,,;
 tlink32 /Tpe /aa sehtest,sehtest,,import32.lib

;---[ CUT HERE ]---------------------------------------------------------

        .386p
        .model  flat                            ; 32 бита рулят

extrn   MessageBoxA:PROC                        ; Определенные API
extrn   ExitProcess:PROC

        .data

szTitle         db      "Structured Exception Handler example",0
szMessage       db      "Intercepted General Protection Fault!",0

        .code

start:
        call    setupSEH


exceptionhandler:
```

```
        mov     esp,[esp+8]                     ; Ошибка дает нам старый ESP
                                                ; в [ESP+8]

        push    00000000h                       ; Аргументы для MessageBoxA
        push    offset szTitle
        push    offset szMessage
        push    00000000h
        call    MessageBoxA

        push    00000000h
        call    ExitProcess                     ; Выходим из приложения

setupSEH:
        push    dword ptr fs:[0]                ; Push'им оригинальный
                                                ; обработчик SEH
        mov     fs:[0],esp                      ; И помещаем новый (который
                                                ; находится после первого
                                                ; call)

        mov     ebx,0BFF70000h                  ; Пытаемся писать в ядро (что
        mov     eax,012345678h                  ; вызовет исключение)
        xchg    eax,[ebx]

end     start
;---[ CUT HERE ]---------------------------------------------------------------
```

[...]

--[DDT#1.2_6]----------

Я надеюсь, что вы поняли все это. Если нет... Ладно, забудьте об этом :). Также как и другие методы, представленные выше, он не работает по отношению к SoftICE.

Win9X: Обнаружение SoftICE (I)

Здесь я должен поблагодарить Super/29A, потому что именно он рассказал мне об этом методе. Я разделил его на две части: в этой мы рассмотрим, как использовать его из вирусов Rin-0. Я не буду помещать здесь программу-пример целиком, потому что она займет лишнее место, но вы должны знать, что этот метод должен выполняться в Rin-0, а VxDCall должен быть восстановлен из-за проблемы обратного вызова (вы помните о ней?).

Ок, мы будем использовать сервис менеджера виртуальных машин (VMM) Get_DDB, поэтому сервис будет 00010146h (VMM_Get_DDB). Давайте посмотрим, что говорится об этом сервисе в SDK.

```
        mov     eax, Device_ID
        mov     edi, Device_Name
        int     20h                             ; VMMCall Get_DDB
        dd      00010146h
        mov     [DDB], ecx
```

• Определяет, установлен или нет VxD определенного устройства, и если установлен, возвращает DDB для этого устройства.
• Использует ECX, флаги.
• Возвращает DDB для определенного устройства, если функция была выполнена успешно.
• В противном случае возвращает ноль.
• Device_ID: Идентификатор устройства. Этот параметр может быть равен нулю для именованных устройств.
• Device_Name: Восьмибуквенное имя устройства, которое дополнено (в случае необходимости) пустыми символами. Этот параметр требуется только тогда, когда Device_ID равен нулю. Имя устройства чувствительно к регистру.

Ладно, вы наверняка удивляетесь, о чем идет разговор. Очень просто, поле Device_ID VxD SoftICE постоянно для всех программ, а так как оно зарегистрировано в Micro$oft'e, у нас есть оружие против этого отладчика. Его Device_ID всегда равно 202h. Поэтому мы можем использовать следующий код:

```
mov     eax,00000202h
VxDCall VMM_Get_DDB
xchg    eax,ecx
jecxz   NotSoftICE
jmp     DetectedSoftICE
```

Где NotSoftICE должно быть продолжением кода вируса, а метка DetectedSoftICE должна обрабатывать тот случай, если наш враг жив :). Я не предполагаю здесь никакой деструкции, так как, например, на моем компьютере SoftICE всегда активен :).

Win9X: Обнаружение SoftICE (II)

Ладно, теперь идет другой метод для обнружения моего возлюбленного SoftICE'a, но основанный на той же самой идее, что и выше: 202h ;). Снова я должен поблагодарить Super :). Ладно, в Ralph Brown Interrupt list мы можем найти очень крутой сервис: прерывание 2Fh, функция 1684h.

```
На входе:

       AX = 1684h
       BX = virtual device (VxD) ID (see #1921)
       ES:DI = 0000h:0000h

На выходе:

       ES:DI -> входная точка VxD API, или 0:0, если VxD не поддерживает этот
       API.

N.B.:  некоторые виртуальные устройства в улучшенном режиме Windows
       предоставляют сервисы, которые могут использовать приложения.
       Например, Virtual Display Device предоставляет API, используемый
       WINOLDAP.
```

Поэтому вы поместите 202h в BX, запустите эту функцию. А потом скажете... "Эй, Билли... Через какое место я могу использовать прерывания?". Мой ответ... ИСПОЛЬЗУЙТЕ VxDCALL0!!!

Win32: Обнаружение SoftICE (III)

И наконец, чудесный трюк, которого вы ждали... Универсальный способ найти SoftICE и Win9x и в WinNT! Это очень просто, 100% основанно на API и без всяких "грязных" трюков, которые не идут на пользу совместимости. И ответ находится не так далеко, как вы думаете... ключ заключается в API, который вы уже использовали раньше: CreateFile. Да, именно эта функция... Разве это не прекрасно? Ладно, мы можем попытаться открыть следующее:

• SoftICE для Win9x : "\.\SICE"
• SoftICE для WinNT : "\.\NTICE"

Если функция возвращает нам что-нибудь, отличное от -1 (INVALID_HANDLE_VALUE), SoftICE запущен! Далее следует демонстрационная программа:

```
;---[ CUT HERE ]-------------------------------------------------------------

        .586p
```

```
        .model  flat

extrn   CreateFileA:PROC
extrn   CloseHandle:PROC
extrn   MessageBoxA:PROC
extrn   ExitProcess:PROC

        .data

szTitle         db      "SoftICE detection",0

szMessage       db      "SoftICE for Win9x : "
answ1           db      "not found!",10
                db      "SoftICE for WinNT : "
answ2           db      "not found!",10
                db      "(c) 1999 Billy Belcebu/iKX",0

nfnd            db      "found!    ",10

SICE9X          db      "\\.\SICE",0
SICENT          db      "\\.\NTICE",0

        .code

DetectSoftICE:
        push    00000000h                       ; Проверяем наличии
        push    00000080h                       ; SoftICE для среды Win9x
        push    00000003h
        push    00000000h
        push    00000001h
        push    0C0000000h
        push    offset SICE9X
        call    CreateFileA

        inc     eax
        jz      NoSICE9X
        dec     eax

        push    eax                             ; Закрываем открытый файл
        call    CloseHandle

        lea     edi,answ1                       ; SoftICE найден!
        call    PutFound
NoSICE9X:
        push    00000000h                       ; А теперь пытаемся открыть
        push    00000080h                       ; SoftICE для WinNT...
        push    00000003h
        push    00000000h
        push    00000001h
        push    0C0000000h
        push    offset SICENT
        call    CreateFileA

        inc     eax
        jz      NoSICENT
        dec     eax

        push    eax                             ; Закрываем хэндл файла
        call    CloseHandle

        lea     edi,answ2                       ; SoftICE для WinNT найден!
        call    PutFound
NoSICENT:
        push    00h                             ; Показываем MessageBox с
        push    offset szTitle                  ; результатами
        push    offset szMessage
        push    00h
        call    MessageBoxA

        push    00h                             ; Завершаем программу
        call    ExitProcess
```

```
PutFound:
        mov     ecx,0Bh                         ; Меняем "not found" на
        lea     esi,nfnd                        ; "found"; адрес, где нужно
        rep     movsb                           ; изменить, находится в EDI
        ret

end     DetectSoftICE

;---[ CUT HERE ]---------------------------------------------------------
```

Это действительно работает, поверьте мне :). Тот же метод можно применить к "враждебным" драйверам, просто проведите небольшое исследование.

Заключительные слова

Ладно, вот я и рассказал о нескольких простых антиотладочных трюках. Я надеюсь, что вы сможете использовать их в своих вирусах без проблем. Пока!

Правда о NtLdtSetEntries

Автор: FreeMan

Дата: 2007-11-11

Введение Речь пойдёт об обломе эмуляторов и отладчиков, которые не учитывают то, что в защищенном режиме есть сегментные регистры, и они, к тому же, играют большую роль. Связанно это, в частности, с тем, что сегменты кода и данных в OS Windows имеют базу 0 и при формировании линейного адреса он совпадает со смещением. Облом программ такого рода данным методом сводится к добавлению ещё одного дескриптора в LDT и работу через селектор, который содержит номер данного дескриптора. **NtLdtSetEntries**

Эта замечательная функция из ntdll.dll дает возможность добавить элемент (даже 2 элемента) в локальную таблицу дескрипторов. Вызов данной функции приводит к вызову функции PsSetLdtEntries в ядре. В этой функции производится довольно тщательная проверка дескриптора и селектора. Возможно, добавить дескриптор только если:

Его тип - ReadWrite, ReadOnly, ExecuteRead, ExecuteOnly, или Invalid.

Это не системный дескриптор (что автоматом лишает нас прорулить в ядро с помощью Callgate).

```
DPL=3
Base<MM_HIGHEST_USER_ADDRESS (7FFEFFFF)
Base+Limit<MM_HIGHEST_USER_ADDRESS
```

После этих проверок происходит вызов Ke386SetLdtProcess->Ki386LoadTargetLdtr->KiLoadLdtr-> asm lldt.

Антиэмуляция

Основана на том, что автоматические системы не учитывают, что при формировании линейного адреса к смещению прибавляется база дескриптора, номер которого содержится в одном из сегментных регистров. Рассмотрим небольшой пример. Допустим, есть такой код:

```
xor  edi,edi
stosd
```

Большинство считает, что, будучи выполненным в OS Windows, данный код сгенерирует исключение и произойдет вызов SEH. Но это так не во всех случаях. Ведь stosd эквивалентна паре инструкций:

```
mov   dword[es:edi],eax
add(sub) edi,4
```

В общем случае в пользовательской программе es=23h, указывает на 4ый дескриптор в LDT, который описывает сегмент с базой 0. Тогда обращение произойдет по нулевому линейному адресу и действительно возникнет исключение. Но если, например, добавить свой дескриптор с базой 401000h и в es поместить селектор, который содержит его номер, то в результате выполнения вышеприведенного кода произойдет обращение по адресу 401000h, где может находиться доступный для записи сегмент данных программы.

Приведу в качестве иллюстрации код, который вводит в заблуждение эмулятор NOD'a (на других AB не проверял) и некоторых программистов.

```
format PE GUI 4.0
entry start
include '%fasminc%\win32a.inc'

;
;Данная строка находится по адресу 401000h
;
```

```
mess db '1234AGE',0

;
;Дескриптор, который необходимо добавить в LDT
;
LDT_Entry:
;
;Младшие 16бит лимита
;
dw 0100h

;
;Младшие 24бита базы (401000h)
;
db 0,10h,40h


;
;Тип: 1010 - так как S=1, 0-сегмент данных 010-для чтения/записи
;S(тип дескриптора): 1 (сегмент данных или кода)
;DPL: 11 (ring3)
;P: 1 (сегмент присутствует)
;
db 11110010b

;
;16-19 биты лимита: 1111b
;AVL: 0 - чо угодно
;Reserved: 0 - надо чтоб был 0 иначе конец света
;G: 1 - лимит умножаем на 1000h
;
db 11000000b

;
;Старший байт базы
;
db 0

start:
     ;
     ;Добавим дескриптор в LDT
     ;
     invoke NtSetLdtEntries,1111111b,dword[LDT_Entry],dword[LDT_Entry+4],0,0,0

     ;
     ;Селектор с номером данного дескриптора - в es
     ;
     push   es
     push   1111111b
     pop    es

     ;!!!!!!
     ;Обращение произойдет по адресу 401000h, а не 0
     ;!!!!!!
     mov    eax,'MESS'
     xor    edi,edi
     stosd

     ;
     ;Восстановим es
     ;
     pop    es
     invoke MessageBox,0,mess,mess,MB_OK
     invoke ExitProcess,0

data import
library kernel32,'KERNEL32.DLL',\
          user32,'USER32.DLL',\
          comdlg32,'comdlg32.dll',\
          ntdll,'ntdll.dll'
include '%fasminc%\apia\comdlg32.inc'
```

```
include '%fasminc%\apia\user32.inc'
include '%fasminc%\apia\kernel32.inc'
include '%fasminc%\ntdll.inc'; import ntdll, NtSetLdtEntries,'NtSetLdtEntries'
end data
```

Следует обратить внимание также на то, что перед вызовом АПИ следует обязательно восстановить значения сегментных регистров, так как обращение к какому-либо адресу приведет к тому, что на самом деле произойдет обращение по адресу бОльшему на базу, указанному в дескрипторе. Другими словами, если где-то в коде MessageBoxA встретится команда типа mov eax,es:[77d91234h], то на самом деле будет попытка записи в eax значения ячейки по адресу 78192234h, что, скорее всего, вызовет исключение.

Антиотладка

Основана на том же принципе, только с учетом того, что формируется дескриптор для сегмента кода. Всем известный отладчик OllyDbg при изменении значения cs путем выполнения дальнего вызова, либо перехода в сегмент с другой базой тихонько выпадает в осадок. Приведу код, иллюстрирующий данный подход.

```
format PE GUI 4.0
entry start
include '%fasminc%\win32a.inc'
;
;Опять же адрес данной строки 401000h
;
mess db 'MESSAGE',0
data import
library kernel32,'KERNEL32.DLL',\
          user32,'USER32.DLL',\
          comdlg32,'comdlg32.dll',\
          ntdll,'ntdll.dll'

include '%fasminc%\apia\comdlg32.inc'
include '%fasminc%\apia\user32.inc'
include '%fasminc%\apia\kernel32.inc'
include '%fasminc%\ntdll.inc'
end data

;
;Для удобства вызова функций, рассчитанных на
;работу в сегменте с base=0
;
macro invokes [arg]
{
  common
    if ~ arg eq
  reverse
    pushd arg
  common
    end if
    call invoker
}

;
;На этот раз дескриптор кода
;
LDT_Entry:
;
;Младшие 16бит лимита
;
dw 0ffffh

;
;Младшие 24бита базы 401000h
;
db 0,10h,40h
;
;Тип: 1010 - так как S=1, 1-сегмент кода 010-для чтения/записи
```

```
;S(тип дескриптора): 1 (сегмент данных или кода)
;DPL: 11 (ring3)
;P: 1 (сегмент присутствует)
;
db 11111010b

;
;16-19 биты лимита: 0000
;AVL: 0 - чо угодно
;Reserved: 0 - надо чтоб был 0, иначе конец света
;G: 1 - лимит умножаем на 1000h
;
db 11000000b

;
;Старший байт базы
;
db 0

start:
     ;
      ;Добавляем дескриптор
     ;
     invoke NtSetLdtEntries,1111111b,dword[LDT_Entry],dword[LDT_Entry+4],0,0,0

     mov    ax,cs
     ;
      ;Мега фокус-покус
     ;
      jmp    1111111b:ёпт-401000h

   ёпт:

     ;
     ;База кода теперь 401000, а не 0 :)
     ;Сохраним старое значение cs для успешного вызова АПИ
     ;
     mov    [cseg],ax

     ;
     ;Вызов АПИ следует осуществлять через "переходник"
     ;
     invokes    MessageBox,0,mess,mess,MB_OK
     invokes    ExitProcess,0

     ;
     ;Переходник работает так:
     ; переход к базе кода 401000
     ; вызов АПИ
     ; переход к базе 0
     ;
     invoker:
        ;
        ;дальний переход, чтоб загрузить оригинальный cs
        ;
        db 0eah   ;опкод jmp far
        dd inv    ;смещение
        cseg dw ? ;сегмент

        ;
        ;Возврат из процедуры
        ;
     rets:
        jmp     [reteng]
        reteng  dd ?

        ;
        ;Тут база 0
        ;
     inv:
        ;
```

```
;Запомним адрес вызываемой процы(относительно 0)
;
mov     eax,dword[esp+4]

;
;Запомним адрес возврата из процедуры(относительно 0)
;
mov     edx,dword[esp]
mov     [reteng],edx

;
;Вершина стека должна указывать на параметры, переданные процедуре
;
add     esp,8

;
;Вызов процедуры
;
call    dword[eax]

;
;Прыжок, дабы вернуться к базе 0
;
db 0eah
;
;Смещение относительно 401000h
;
dd rets-401000h
dw 1111111b
```

Удачная комбинация

Также можно рассмотреть комбинацию двух данных подходов - добавить два дескриптора с одинаковой базой: один для кода, один для данных. После этого пройтись по всем модулям процесса, пофиксить релоки с учетом новой базы, да и сам PEB тоже, наткнутся на кучу "подводных камней", после чего спокойно, загрузив в сегментные регистры новые селекторы, вызывать АПИ, не восстанавливая оригинальные значения сегментных регистров.

На этом заканчиваю статью, wasm.ru forever :)