Рэй Дункан. Оптимизация программ на ассемблере.

Часть 1

Несмотря на все более широкое распространение языков программирования и интегрированных сред программирования, оптимизация программ на ассемблере остается актуальной темой дискуссий для программистов. Можно упомянуть, например, форум програамистов, проведенный сетью PC MagNet, который стал ареной многочисленых "дуэлей": то один, то другой участник предлагал всем желающим решить небольшую, но интересную задачу программирования - и рассматривал присылаемые решения, ожидая, кто жее и как решит задачу наименьшей кровью, то есть затратив минимум байтов на программу. Подобно этому проведенная сетью BIX конференция по языку ассемблера для процессора 8088 стала трибуной немалого числа основательных рассуждений по поводу неочевидных аспектов оптимизации ассемблерных программ.

Несмотря на самый общий и широкий интерес к проблеме, литература по оптимизации ассемблерных программ для процессора Intel 80x86 на удивление скудна. Пару лет назад, готовясь к докладу по развитию программного обеспечения, я просмотрел оглавления всех основных журналов по программированию и обнаружил лишь горстку статей на эту тему. С другой стороны, литература по оптимизации программ для компиляторов высокого уровня весьма обширна, и многие концепции, развитые в ней, будут полезны и при программировании на языке ассемблера. Так что говорить, что литературы совсем нет, было бы несправедливо. Ниже мы сначала рассмотрим общие методики оптимизации, а затем обсудим более серьезный вопрос - когда и что стоит оптимизировать.

Оптимизация по быстродействию

Если вы пришли к выводу, что ваша программа работает недостаточно быстро, первое, что надо сделать, - это убедиться, что вы решаете задачу, пользуясь наилучшими алгоритмами и представлениями данных. Замена примитивного или неадекватного алгоритма более подходящим может ускорить выполнение вашей программы на порядок и более. Так что если вы проведете несколько дней, штудируя Кнута или Седжуика (в надежде подобрать алгоритм, до которого вряд ли додумались бы сами), - будьте уверены: вы сделали выгодное вложение своего драгоценного времени. Аналогично переход от "очевидной", но простой структуры данных (такой, как связный список) к более сложной (например, бинарному дереву) может дать такие результаты, которые с лихвой окупят ваши усилия по усовершенствованию программы.

Если вы уверены, что выбрали наилучшие алгоритмы и структуры данных, следующее, на что следует обратить внимание, - это использование программой данных, хранимых в памяти. Даже самые быстрые устройства работы с дисками, используемые в персональных компьютерах, работают несравнимо медленнее, чем такие мощные вычислители, как процессоры 80386 или 80486, так что постарайтесь свести обращения к диску до возможного минимума. Ознакомьтесь со всеми имеющимися типами памяти, доступными программе DOS, - расширяемой и отображаемой памятью, старшими блоками памяти и так далее - и полностью используйте возможности оперативной памяти для хранения в ней данных, с которыми работает ваша программа, чтобы время обращения к ним было минимальным. Полезно также ознакомиться с техникой уплотнения данных, поскольку почти во всех случаях быстрее оказывается сначала уплотнить, а затем распаковать данные, когда в них возникает необходимость, чем по нескольку раз считывать неуплотненную информацию с диска.

Еще одной темой, заслуживающей обсуждения, является уменьшение объема вычислений в программе. Составители компиляторов пользуются забавными терминами - устранение общих подвыражений, сворачивание констант, распространение констант - для того, чтобы, по сути, сказать одно и то же: во время работы программы одно и то же значение ни в коем случае не должно вычисляться дважды. Вместо этого программа должна рассчитать каждое значение лишь один раз и сохранить его для повторного использования. Еще лучше преносить вычисления со стадии исполнения на стадию ассемблирования всякий раз, когда это позволяют ограниченные математические "способности" MASM (или TASM, или OPTASM). Совершенно аналогично вам, возможно, удастся существенно повысить быстродействие программы, преобразовав вычисления в обращения к таблицам, котрые заранее генерируются и сохраняются отдельной программой.

Если вы сделали все возможное на абстрактном уровне по всем названным направлениям, пора спуститься на грешную землю. Осталось немногое - "отжать из программных кодов всю воду" и "прочистить все циклы", особенно, если программа существенно использует работу с дисплеем и выводит на экран графику. Вот некотрые из самых общих процедур этой категории.

- Замещенин универсальных инструкций на учитывающие конкретную ситуацию, например, замена умножения на степень двойки на команды сдвига (отказ от универсальности).
- Уменьшение количества передач управления в программе: за счет преобразования подпрограмм в макрокоманды для непосредственного включения в машинный код; за счет преобразования условных переходов так, чтобы условие перехода оказывалось истинным значительно реже, чем условие для его отсутствия; перемещение условий общего характера к началу разветвленной последовательности переходов; преобразование вызовов, сразу за которыми следует возврат в программу, в переходы ("сращивание хвостов" и "устранение рекурсивных хвостов") и т.д.
- Оптимизация циклов, в том числе перемещение вычислений неизменяющихся величин за пределы циклов, разворачивание циклов и "соединение" отдельных циклов, выполняемых одно и то же количество раз, в единый цикл ("сжатие цикла").
- максимальное использование всех доступных регистров за счет храниения в них рабочих значений всякий раз, когда это возможно, чтобы минимизировать число обращений к памяти, упаковка множественных значений или флагов в регистры и устранение излишних продвижений стека (особенно на входах и выходах подпрограмм).
- Использование специфических для данного процессора инструкций, например, инструкции засылки в стек непосредственного значения, которая имеется в процессоре 80286 и более поздних. Другие примеры - двухсловные строковые инструкции, команды перемножения 32-разрядных чисел, деление 64-разрядного на 32-разрядное число и умножение на непосредственное значение, которые реализованы в процессорах 80386 и 80486. Ваша программа должна, разумеется, вначале определить, с каким типом процессора она работает! Для этого может быть полезна программа, приведенная на рис. 1 и выдающая код, характеризующий тип ЦП. [Содержится в файле ASM_OPT1.ASM - Прим. набивальщика]

В процессорах 80286 и более поздних вам, возможно, также удастся увеличить скорость исполнения программы на несколько процентов за счет выравнивания расположения данных и меток, на которые происходит передача управления, относительно некоторых границ. Процессоры 8088 и 80286 - 16-разрядные и им "удобнее" работать с данными, выровненными относительно 16-битовых границ (размер одного слова); процессор 80386 предпочитает выравнивание в 32 бита (два слова); из-за устройства его внутренней кэш-памяти процессору 80486 легче работать, если произведено выравнивание на параграф (16-байтовое).

Если же все остальное успеха не принесло, а вы пишете некую штучную программу, а не программный пакет, предназначенный для продажи на массовом рынке, проблему быстродействия, может быть, проще "решить" аппаратным путем. При существующих сегодня ценах [О, да!!! - Прим. набивальщика] часто оказывается намного выгоднее заказать новый, более мощный компьютер для работы с вашей специализированной программой, чем тратить время на мучения, связанные с переделкой программы, ее оптимизацией и последующей отладкой заново. К сожалению, безоглядное и некритическое принятие этого подхода такими производителями программных изделий, как Microsoft и Lotus, привело к рождению громоздких монстров, подобных Windows 3.0, Programmer's Workbench и Lotus 1-2-3/G, - но это уже тема другого разговора.

Оптимизация по объему

Если работоспособность вашей программы ограничена ее размером, а не скоростью исполнения, то вам надо вновь подумать над стратегией оптимизации - на этот раз ухищрениями, в точности противоположными тем, что вы использовали для увеличения быстродействия. Тщательно изучите свою программу и определите, что создает основную проблему - размер кода или объем данных.

Если вам приходится работать с болшими блоками данных, то нужный эффект может дать их организация в нетривиальные структуры. Однако замена быстрообрабатываемых, но неплотных массивов и таблиц более компактными структурами типа связных списков или упаковка данных с использованием битовых полей, вероятно, даст не слишком большие преимущества. Примитивное уплотнение таблиц и их последующее, по необходимости, разуплотнение обычно не очень полезно, так как часто требуется разуплотнять все данные лишь для того, чтобы добраться до какого-то одного пункта, а программы уплотнения/разуплотнения сами по себе обычно занимают заметный объем памяти.

Большие просмотровые таблицы и массивы можно поместить в дисковый файл и при необходимости считывать в память малыми частями. Но это может нанести сокрушительный удар по производительности, если данные запрашиваются в случайном порядке. Часто можно вообще отказаться от просмотровых таблиц и производить вычисления значений переменных всякий раз, когда в последних возникает необходимость. Вы также должны искать и устранять константы и переменные, которые реально никогда не используются программой, поскольку вместо них она использует другие величины, вычисленные ранее в процессе разработки и отладки. Во всяком случае, я еще раз хочу подчеркнуть, что чрезвычайно важно усвоить, как пользоваться всеми видами памяти, доступными компьютеру, и сделать программу достаточно гибкой, чтобы она могла использовать все и каждую из них.

Оптимизация программы с целью уменьшения размера - это совсем не то же самое, что оптимизацмя для повышения быстродействия. Во-первых, вы должны просмотреть весь текст программы и устранить все предложения и процедуры, которые никогда не выполняются или недоступны ни из каой точки программы (мертвые коды). Если вы работаете с большой прикладной программой, над которой до вас уже потрудилось несколько программистов, вас, возможно, даже удивит, как много строк можно безболезненно удалить. Во-вторых, проанализируйте программу заново и соберите все идентичные или функционально сходные последовательности кода в подпрограммы, которые могут быть вызваны из любой точки программы. Чем более универсальными вам удастся сделать подпрограммы, тем более вероятно, что их код может быть использован повторно. Если вы будете последовательно придерживаться этого подхода, где только возможно, то получите очень компактную программу модульного типа, состоящую главным образом из вызовов подпрограмм.

Если вы сделали все, что я рекомендовал выше, а вам по-прежнему не хватает памяти, то попробуйте поискать удачи еще на нескольких путях. Вы можете перекомпоновать свою программу в относительно независимые модули, которые могут считываться в память, как оверлеи. Вы можете закодировать функционирование вашей программы в таблицах, "управляющих" ее исполнением. Вы можете прибегнуть к методике "прошитого" кода или псевдокода и представить логику вашей программы с использованием гораздо меньшего объема памяти за счет некторого снижения быстродействия. Можно, наконец, прибегнуть к тяжелой артиллерии современной компьютерной технологии и использовать стандартный интерпретатор или компилятор "малого языка", который, в свою очередь, будет виртуальной машиной для прикладной программы [Подобно Multi-Edit, в котором это набирается - Прим. набивальщика]. Quick Basic, Excel и BRIEF - самые привычные примеры применения соответственно стратегии прошитого кода, псевдокода и малого языка.

В конечном счете, если вы пришете специализированную прикладную программу, преодолеть проблемы памяти, возможно, - снова - удастся объединенным программно-аппаратным путем. Среди многих возможностей есть такие: использование более высоких версий операциооной системы, таких как DOS 5.0 или OS/2, для расширения объема рабочей памяти (в пределах первых 640 Кбайт); установка еще одной платы расширения памяти; пеерход к компьютерам на процессорах 80386 или 80486, котрые поддереживают большую память по сравнению с использовавшейся вами машиной на процессоре 8088 или 80286; и/или запуск вашей программы под управлением расширителя DOS.

А стоит ли оптимизировать?

Оптимизация кодов на любом языке всегда требует идти на компромиссы, среди которых такие, как:

- сокращение потребного объема памяти за счет снижения быстродействия;
- повышение быстродействия за счет ухудшения возможностей сопровождения и доступности текста программы для чтения;
- сокращение времени исполнения программы за счет увеличения времении ее разработки.

Все это кажется очевидным, но в большинстве реальных ситуаций принять решение не так просто, как кажется на первый взгляд. Классическим примером является выбор между двумя приведенными ниже инструкциями, каждая из которых может быть использована для перемещения некоторого значения из регистра DX в регистр AX (конечно, с различными побочными эффектам):

```
        XCHG    AX, DX
или
        MOV     AX, DX
```

В процессоре 8088 команда MOV занимает 2 байта и требует двух тактов ЦП, тогда как команда XCHG занимает 1 байт, но требует трех тактов. Пока все кажется ясным: надо выбирать между скоростью и памятью. Но реальное время исполнения команды существенно зависит от контекста, размера очереди команд ЦП, размера и характеристик кэш-памяти системы и так далее, тогда как даже число циклов, требуемых для выполнения инструкций, меняется от одной модели ЦП к другой. Как оказывается, практически невозможно предсказать скорость исполнения нетривиальной последовательности инструкций в процеессоре фирмы Intel, особенно в последних моделях 80386 и 80486, в которых интенсивно используется конвейерная обработка, - вам придется составлять различные возможные комбинации инструкций, запускать их и экспериментально определять время их исполнения.

Аналогично, баланс между временем исполнения программы и временем ее разработки и баланс между возможностями сопровождения и ее быстродействием редко бывают столь однозначны, как нам бы хотелось, а долговременные последствия возможных ошибочных решений могут быть весьма неприятными. Вообще же, занимаясь оптимизацией, важнее всего понимать, когда ее делать, а когда лучше оставить все как было. Процитируем Дональда Кнута: "Многие беды программирования проистекают от преждевременной оптимизации". Прежде чем думать о настройке своей программы, убедитесь, что она правильная и полная, что вы используете верный алгоритм для решения поставленной задачи и что вы составили самый ясный, самый прямой, самый структурированный код, который только был возможен.

Если программа удовлетворяет всем этим критериям, то, на самом деле, ее объем и скорость исполнения в болшинстве случаев будут вполне приемлемыми без каких-либо дальнейших усовершенствований. Одно только использование ассемблера само по себе приводит к увеличению скорости исполнения программы в два-три раза и примерно к такому же уменьшению размера по сравнению с аналогичной программой на языке высокого уровня. И еще: если что-то упрощает чтение программы и ее сопровождение, то обычно это же приводит к увеличению скорости исполнения - здесь можно назвать отказ от макаронных кодов со многими ненужными переходами и вызовами подпрограмм (которые являются тяжелой работой для процессора с архитектурой Intel 80x86, поскольку они сбрасывают очередь команд), а также предпочтение простых прямолинейных участков машинных команд аналогичным сложным.

Тем не менее, вашей наиглавнейшей заботой должны быть ощущения потенциального пользователя при работе с вашей программой: насколько производительной покажется программа ему? Прислушаемся к словам Майкла Эбраша: "Всякая оптимизация, ощущаемая пользователем, заслуживает того, чтобы ее сделать". И наоборот, если в массах пользователей о вашей программе складывается мнение, как о тупой и неуклюжей, то очень вероятно, что она не будет должным образом оценена. Примером может служить судьба пакета ToolBook. Следовательно, должно казаться, что ваша программа мгновенно откликается на действия пользователя даже тогда, когда она занята длительными вычислениями иил операциями с диском. Она должна поддерживать экран дисплея в "живом" состоянии, заполняя его чем-то вроде циферблатов, термометров, и позволять пользователю безопасно прерывть длительные операции в любое аремя, если его намерения изменились и он решил заняться чем-нибудь еще.

Если вы действительно вынуждены прибегнуть к шлифовке кода и циклов с помощью методов, о которых я говорил выше, то постарайтесь затратить свои время и силы на действия в правильном направлении. Помните о естественной иерархии временных масштабов: среди операций, перечисленных ниже, каждая следующая требует на порядок больше времени, чем предыдущая. Ита: это операции регистр/регистр, операции с памятью, операции с диском и операции взаимодействия с пользователем. Так что не тратьте силы на то, чтобы сократить несколько машинных циклов в программе, если скорость ее исполнения ограничена операциями с дисковыми файлами: вместо этого попытайтесь найти способы сократить число таких операций. А после того, как вы сделали что-то, что в принципе могло бы быть оптимизацией, проведите дотошную проверку полученных результатов и вообще - проверяйте, проверяйте, проверяйте.

В своей превосходной книге "Пишем эффективные программы" (Writing Efficient Programs - Prentice Hall, 1982) Джон Бентли рассказывает кошмарную историю из жизни фирмы Bell Labs - Историю, которую мы все и всегда должны помнить:

"В начале 60-х годов Виктор Высоцкий [Victor Vysotsky] работал над усовершенствованием компилятора Фортрана, причем в число исходных требований входило отсутствие сколько-нибудь заметного снижения времени компиляции. Одна из подпрограмм исполнялась редко (во время разработки Высоцкий оценил, что она должна вызываться примерно в одном проценте компиляций, причем в каждой лишь однажды), но работала крайне медленно. Поэтому Высоцкий затратил неделю на удаление из нее лишних циклов. Модифицированный компилятор работал достаточно быстро. После двух лет интенсивной эксплуатации компилятор выдал сообение о внутренней ошибке при компиляции одной программы. Когда Высоцкий

исследовал компилятор, он обнаружил, что ошибка была в прологе "критической" подпрограммы и что эта ошибка содержалась в данной подпрограмме всегда, с самого начала производства".

Высоцкий совершил три принципиальных ошибки: он не провел полной анализ программы перед тем, как приступить к оптимизации, так что он напрасно тратил время на оптимизацию подпрограммы, не влияющей на быстродействие; он не смог сделать оптимизацию правильно; и он не провел испытаний оптимизированной подпрограммы, чтобы убедиться, что она работает согласно спецификации.

Я совсем не хочу тыкать пальцем в мистера Высоцкого: в своей жизни я совершил множество промахов куда как серьезнее, но (поскольку я не работаю в фирме Bell Labs) эти промахи, к счастью, не увековечены в книге Джона Бентли. Однако этот случай из жизни Высоцкого - хороший пример того, как время и энергия могут быть растрачены впустую на святое дело оптимизации и как рано или поздно проявляется отказ от методичного исполнения всех основных процедур профилирования и контроля в процессе оптимизации.

Часть 2

В предыдущей части мы обсуждали некоторые общие вопросы оптимизации, а затем поговорили о тех компромиссах, на которые приходится идти, оптимизируя быстродействие и размер программы. В этой и в следующей частях мы подробнее рассмотрим некоторые классические образцы "локальной" оптимизации. Но важно помнить, что эти частные методики следует использовать только при определенных обстоятельствах - а именно: после того, как вы убедитесь, что применили правилные алгоритмы и структуры данных, что полностью отладили программу и что средства профилирования показали вам те самые фрагменты программы, которые ограничивают производительность.

Отказ от универсальности

Операции умножения и деления требуют немалых усилий от почти любого ЦП, поскольку должны быть реализованы (аппаратно или программно) через сдвиги и сложения или сдвиги и вычитания соответственно. Старинные 4- и 8-разрядные процесоры не содержали машинных команд для умножения или деления, так что эти операции приходилось реализовывать при помощи длинных подпрограмм, где явным образом выполнялись сдвиги и сложения или вычитания. Первые 16-разрядные микропроцессоры, такие, как 8086 и 68000, действительно позволяли производить операции умножения и деления аппаратными средствами, но соответствующие процедуры были невероятно медленными: в процессоре 8086, к примеру, для деления 32-разрядного числа на 16-разрядное требовалось примерно 150 тактов.

Поэтому маленькие хитрости для ускорения или устранения операций умножения и деления были и пока остаются среди первых приемов, которые изучает любой программист, стремящийся к совершенству. Большинство из этих хитростей относятся к категории, которую называют "отказ от универсальности". Под этим понимается замена рассчитанных на общий случай команд (или вызовов соответствующих подпрограмм) последовательностями сдвигов и сложений или вычитаний для случая конкретных операндов.

Давайте сначала рассмотрим простейшую процедуру оптимизации умножения. Чтобы умножить число на степернь двойки, его достаточно просто сдвинуть влево на необходимое число двоичных (битовых) позиций. Вот так, например, выглядит некотрая общая, но медленная последовательность команд при умножении значения переменной myvar на 8:

```
mov    ax, myvar
mov    bx, 8
mul    bx
mov    myvar, ax
```

В процессорах 8086/88 эта программа может быть преобразована в более "быструю" последовательность сдвигов:

```
mov    ax, myvar
shl    ax, 1        ; * 2
shl    ax, 1        ; * 4
shl    ax, 1        ; * 8
mov    myvar, ax
```

или даже в такую: shl myvar, 1 shl myvar, 1 shl myvar, 1

Если требуется сдвиг на одну или две позиции, то обычно проще всего выполнить эти операции над операндами в памяти. Если нужен сдвиг на многоо позиций, то повышенная скорость работы над регистровыми операндами вполне компенсирует дополнительную команду для загрузки числа в какой-либо регистр и извлечения его оттуда после сдвига.

Но не торопитесь - даже эта простая оптимизация не так тривиальна, как кажется! Очередь команд в процессорах семейства 80x86, конкретная модель процессора, которая используется в вашем компьютере, и наличие или отсутствик кэш-памяти могут в совокупности сыграть самые причудливые шутки. В некотрых случаях и на некоторых моделях ЦП иногда соит использовать эту команду в варианте "сдвиг на указанное в CX число позиций":

```
mov    ax, myvar
mov    cx, 3
shl    ax, cx
mov    myvar, ax
```

А в процессоре 80186 и более поздних имееется вариант "сдвиг на число позиций, заданное непосредственным операндом", что еще удобнее:

```
shl    myvar, 3
```

Если вам требуется умножать на степень двойки числа длиной более 16 разрядов, для организации операций над двумя и более регистрами используется флажок переноса. Например, для умножения 32-разрядного числа в DX:AX на 4 можно записать:

```
shl    ax, 1        ; * 2
rcr    dx, 1
shl    ax, 1        ; * 4
rcr    dx, 1
```

Творчески сочетая сдвиги и умножения, можног организовать быстрое умножение на почти любое конкретное значение. Следующий фрагмент производит умножение значения в регистре AX на 10:

```
mov    bx, ax       ; скопировать исходное число
shl    ax, 1        ; * 2
shl    ax, 1        ; * 4
add    ax, bx
shl    ax, 1        ; * 10
```

Использование отказа от универсальности для деления несколько болле ограничено. Деление на степень двойки, конечно, очень просто. Вы просто сдвигаете число вправо, следя лишь за выбором родходящей команды сдвига для желаемого типа деления (со знаком или без знака). Например, для выполнения деления без знака на 4 содержимого регистра AX можно написать:

```
shr    ax, 1
shr    ax, 1
```

а для процессора 80186 и более поздних можно вместо этого использовать команду

```
shr    ax, 2
```

Деление со знаком на 4 обеспечит, например, последовательность

```
sar    ax, 1
sar    ax, 1
```

или для процессора 80186 и более поздних

```
sar    ax, 2
```

Единственная разница между командой логического (без знака) сдвига SHR и командой арифметического (со знаком) сдвига SAR состоит в том, что SHR копирует старший (знаковый) разряд в следующий, а затем заменяет знаковый разряд нулем, тогда как SAR копирует знаковый разряд в следующий младший разряд, оставляя его исходное значение неизменным.

Выбор правильной команды сдвига для быстрого деления очень важен, особенно если вы имеете дело с адресами. Если вы случайно использовали арифметическое деление (со знаком) вместо деления без знака, которое предполагали сделать, последствия этого иногда проявляются сразу же, а иногда и нет - образовавшаяся "блоха" может до поры притаиться и укусить вас позже, когда какое-нибудь изменение размера или порядка компоновки прикладных программ выпустит ее на волю. (Между просим, не забывайте, что мнемообозначения SHL и SAL транслируются в одну и ту же машинную команду, и не без причины, не так ли?)

Деление на степени двойки со сдвигами может быть ваыполнено с помощью флага переноса, и оно ничуть не более сложно, чем умножение. Например, для выполнения деления со знаком на 8 значения, в регистрах DX:AX можно использовать последовательность

```
sar    dx, 1        ; / 2
rcr    ax, 1
sar    dx, 1        ; / 4
rcr    ax, 1
sar    dx, 1        ; / 8
rcr    ax, 1
```

Но, в отличие от операции умножения, использование сдвигов для быстрого деления на произволные числа, такие как 3 или 10, а не на степени двойки, на удивление хлопотно. Если вы решите покорпеть над написанием программы быстрого деления на 10, в которой используется метод, аналогичный приведенному выше методу умножения на 10, то вскоре обнаружите, что полученная программа - длинная, работает медленно, и, более того, - вы уже сделали 90 % работы, необходимой для составления обобщенной программы деления, использующей сдвиги и вычитания. Обычно целесообразнее вместо этого обдумать всю ситуацию заново и преобразовать алгоритм или структуру данных так, чтобы избежать деления на "неудобные" числа.

Прежде чем оставить эту тему и двигаться дальше, я должен упомянуть одну изящную оптимизацию, авторство которой приписывают Марку Збиковскому [Mark Zbikovski], одному из авторов версий 2.x и 3.x системы MS-DOS. Приведенный ннже фрагмент делит значение в регистре AX на 512:

```
shr    ax, 1
xchg   ah, al
cbw
```

Теперь, когда вы увидели этот нетривиальный прием, у вас наверняка возникло множество идей о том, ка организовать умножение или деление на относительно большие степени двух: 256, 512 и т.д., при помощи последовательностей команд XCHG или MOV.

Оптимизация переходов и вызовов подпрограмм

Макаронные программы, изобилующие ветвлениями и переходами во всех направлениях, нежелательны во всех смыслах, а при работе с процессорами серии 80x86 - особенно. Можете считать это утверждение напутствием, цель которого - побудить программистов на ассемблере и тех, кто занимается оптимизацией компиляторов, должным образом структурировать программы. Тут есть свои проблемы, но прежде чем обсуждать оптимизацию переходовв и вызовов, давайте обсудим некоторые особенномти процессоров фирмы Intel.

Быстродействие этих процессоров в значительной мере определяется их архитектурой, основанной на простой конвейерной схеме, содержащей три компоненты: шинный интерфейс (BIU - bus interface unit), очередь упереждающей выборки и исполнительный модуль (EU - execution unit). Когда шина памяти находится в нерабочем состоянии (например, прри выполнении команды из многих циклов, операнды которой находятся в регистрах), шинный интерфейс извлекает байты команд из памяти и помещает их в

очередь упреждающей выборки, последовательно продвигаясь от текущего положения командного счетчика ЦП. Когда исполнительный модуль завершает исполнение очередной команды, он в первую очередь ищет следующую команду в очереди упреждающей выборки: если она там действительно имеется, то к ее расшифровке можно приступить сразу же, не обращаясь лишний раз к памяти.

Как же при такой реализации конвейерной обработки происходят переходы и вызовы подпрограмм? Всякий раз, когда исполнительный модуль расшифровывает команду перехода, он аннулирует текущее содержимое очереди упреждающей выборки и устанавливает новое содержимое счетчика команд. После этого шинный интерфейс должен снова выбирать байты команд, теперь уже начиная с нового адреса, и заносить их в очередь. исполнительный модуль вынужден в это время "простаивать" до тех пор, пока не будет восстановлена полная команда. Кроме того, все обращения к памяти, которые требуются для исполнения команды перехода по новому адресу, также оказывают влияние на выборку следующих команд из памяти. Может пройти немалое время, прежде чем шина снова заполнит очередь упреждающей выборки так, чтобы исполнительный модуль мог работать с полной скоростью. ситуацию усложняет то, что размер очереди командных байтов разный для разных моделей ЦП. Он составляет всего 6 байтов в ранних моделях и 32 байта в компьютерах последних моделей. Это один из факторов, делающих крайне сложным предсказание времен исполнения для конкретных последовательностей команд исходя из количества тактов и длин в байтах. Кроме того, состояние очереди команд для разных типов ЦП зависит и от "выравнивания" команд. Шинный интерфейс должен выбирать команды в соответствии с разрядностью адресной и информационной частей шины. Поэтому производительность очереди команд может существенно ухудшиться из-за неудачных адресов вызовов или переходов. Например, в процессоре 8086 с 16-разрядной шиной памяти выборка из памяти всегда происходит по 16 бит за один раз, так что если команда, на которую передается управление при вызове подпрограммы, начинается с нечетного адреса, половина первого обращения к памяти пропадает впустую.

Все это подталкивает нас к осознанию первого правила оптимизации переходов и вызовов: проверьте, что их точки назначения попадают в подходящие границы адресов для того типа процессора, на котором ваша программа будет исполняться чаще всего. Сделайте это, добавив подходящий атрибут выравнивания (WORD или DWORD) в объявлении сегментов и вставив директиву ALIGN перед каждой меткой. Процессор 8088 имеет 8-разрядную внешнюю шину, так что он абсолютно нечувствителен к выравниванию. Если потенциальными потребителями вашей программы являются пользователи компьютеров на процессоре 8088, к выравниванию прибегать не стоит, поскольку оно лишь потребует дополнительной памяти, но не увеличит производительность.

В то же время, если программе предстоит главным образом работать на компьютерах с процессорами 8086 или 80286, следует произвести выравнивание в границах WORD, а если она рассчитана на процессоры 80386DX, 80486DX - используйте выравнивание DWORD. (Для процессора 80486, в котором есть внутренняя кэш-память, лучше, когда позиции лежат на 16-байтовых границах, но тратить в среднем 8 байт на каждую метку мне кажется непозволительной роскошью.)

Следующее эмпирическое правило, относящееся к переходам и вызовам, очень простое: избавляться от них везде, где только можно. По крайней мере в тех частях программы, где быстродействие определяется в основном процессором. Для этого организуйте программу так, чтобы она исполнялась прямым, последовательным образом, с минимальным числом точек принятия решения. В результате очередь команд будет почти всегда заполнена, а вашу программу будет легче читать, сопровождать и отлаживать.После того, как вы улучшили программу насколько возможно за счет "хорошего" структурирования, следует решить, надо ли двигаться дальше, и постараться увеличить производительность введением в программу "плохой структуры", например, преобразовав переходы к общим точкам выхода из подпрограмм в множественные выходы. В крайних случаях короткие подпрограммы можно преобразовать в макрокоманды, тогда команды процессора будут исполняться последовательно и дополнительных затрат времени на передачу и возврат управления не будет.

Если в программе требуется условный переход, проанализируйте точку принятия решения и организуйте программу так, чтобы вероятность перехода была ниже, чем его отсутствия. Таким образом вы уменьшите число сбросов очереди команд. Например, если программа производит проверку знака переменной, которая может быть отрицательной только в редких случаях, при особых обстоятельствах, то сделайте так, чтобы ваша прогамма "проскакивала" через точку разветвления при положительном значении переменной:

```
cmp    balance, 0
jl     in_the_read  ; бывает редко
```

Аналогично, если разные значени некой переменной инициируют различные действия и вам требуется произвести множественные сравнения, за которыми следуют условные переходы, то попытайтесь переместить сравнения с наиболее вероятным значением ближе к началу цепочки:

```
        cmp     ax, m           ; наиболее вероятное значение
        jne     L1
        .                       ; действия для m
        .
        .
        jmp     L3
L1:     cmp     ax, l           ; менее вероятное значение
        jne     L2
        .                       ; действия для l
        .
        .
        jmp     L3
L2:     cmp     ax, ll          ; наименее вероятное значение
        jne     L3
        .                       ; действия для ll
        .
        .
L3:
```

Если требуется произвести сравнения со многими значениями, разбросаннами по диапазону с больщими разрывами, реализовать множественные сравнения можно по принципу бинарного дерева, сначала сделав рассечение диапазона каким-нибудь одним значением из середины всего диапазона, а затем проверяя, в каком отношении к этому диапазону находится контролируемая переменная (больше, меньше, равно), затем (если переменная больше или меньше параметра сравнения) деля оставшийся интервал другим значением, и так далее. Такая стратегия чрезвычайно эффективна, если значения распределены более или менее однородно и редко. Если же они распределены плотно, то часто наилучшим решением является использование "таблицы переходов". Например, представьте, что в регистре AL находится значение переменной, являющейся ASCII-символом, и есть набор подпрограмм, запускаемых при некоторых определенных символах. Сначала мы составим таблицу адресов подпрограмм в позициях, соответствующих численным значениям ASCII-кодов, затем реализуем разветвления через таблицу следующим образом:

```
table   dw   routine_00
        dw   routine_01
        .
        .
        dw   routine_FE
        dw   routine_FF
        .
        .
        mov  bl, al     ; скопировать символ в BL
        xor  bh, bh     ; сформировать указатель к таблице
        shl  bx, 1      ; перейти по таблице
        jmp  [table+bx]
```

Есть еще две методики оптимизации, связанные с переходами и вызовами, которые требуют внесения определенной степени "деструктурированности" в во всем остальном верную программу и называются "сращиванием хвостов" и "устранением рекурсивных вызовов". Каждая из них подразумевает преобразование вызовов в переходы: вызовы по самой своей природе требуют большего времени, чем переходы, поскольку помещают в стек адрес возврата и в результате требуют больше обращений к памяти. Сращивание хвостов - это просто преобразование команды CALL, непосредственно за которой следует команда RETURN, в команду JMP. Например, последовательность

```
proc1   proc   near
        .
        .
        .
        call   proc2
        ret
```

```
proc1   endp

proc2   proc   near
        .
        .
        .
        ret
proc2   endp
```

преобразуется в более быструю

```
proc1   proc   near
        .
        .
        .
        jmp    proc2
proc1   endp

proc2   proc   near
        .
        .
        .
        ret
proc2   endp
```

Такая оптимизация приводит к следующему: поскольку адрес команды, вызывающей PROC1, находится в стеке, на входе в PROC2, процедура PROC2 возвращается прямо к исходной вызывающей программе, тепм самым устраняя лишние команды CALL и RETURN. Если процедура PROC2 физически (в памяти) следует за программой PROC1, то можно обойтись даже без команды JMP PROC2, и за выполнением PROC1 может сразу же следовать PROC2.

Устранение рекурсивных вызовов очень похоже на сращивание хвостов. Когда программа последовательно вызывает сама себя и этот вызов расположен непосредственно перед командой RETURN в программе, вызов может быть преобразован в переход, что и увеличит скорость, и уменьщшит необходимый объем памяти в стеке. Например, программа

```
proc1   proc   near
        .
        .
        .
        cmp    ax, some_value
        je     exit
        call   proc1
exit:   ret

proc1   endp
```

может быть преобразована в

```
proc1   proc   near
        .
        .
        .
        cmp    ax, some_value
        jne    proc1
        ret
proc1   endp
```

Такая рекурсивная программа часто может быть еще оптимизирована за счет преобразования рекурсии в цикл.

Часть 3

В предыдущих двух статьях данной серии мы обсуждали некоторые общие концепции оптимизации, а затем рассматривали конкретные методы, относящиеся к переходам и вызовам подпрограмм, а также метод отказа от универсальности. В этой статье мы поговорим еще о нескольких методиках "локальной" оптимизации: об оптимизации циклов, о применении таблиц управляющих параметров, а также об ориентированных на конкретные модели процесоров командах. Но сначала я хотел бы еще раз подчеркнуть: обращаться к оптимизации следует только после тщательного выбора алгоритма и структур данных, после того, как вы полностью реализовали, проверили и отладили свою программу и локализовали все "узкие места" при помощи соответствующих тестовых примеров и инструментальных средств профилирования. Стоит еще раз повторить мудрое изречение д-ра Кнута: "Многие беды программирования проистекают от преждевременной оптимизации".

Оптимизация циклов

Литература о компиляторах переполнена обсуждением методов оптимизации циклов, которым присваивают самые экзотические названия: "разгрузка циклов", "вывод инвариантов за циклы", "устранение индуктивных переменных", "сращивание циклов", "разматывание циклов" и т.п. На самом деле все перечисленные методы можно свести к двум простым эмпирическим правилам:

- никогда не делайте в цикле ничего такого, что можно сделать за его пределами;
- везде, где это возможно, избавляйтесь от передач управления внутри циклов.

Первое из правил следует из старинной истины, гласящей, что 90 % времени исполнения программы приходится на 10 % ее кода. Если вы попытаетесь найти роковые 10 %, скорее всего окажется, что это циклы того или иного рода. Поэтому первое, что следует сделать, когда вы пытаетесь ускорить исполнение программы, - это найти в ней "горячие точки" и проверить все циклы в них в качестве потенциальных объектов оптимизации. Цикл отнюдь не всегда представляет собой изящную конструкцию, завершающуюся командами LOOP, LOOPZ или LOOPNZ (в сосбенности, разумеется, если программу писали не вы, а кто-то другой!); часто это просто серия команд, исполнение которых повторяется в зависимости от значения некоторой управляющей переменной или флажка.

Циклы можно подразделить на две категории: к первой относятся циклы, время исполнения которых определяется какими-то внешними механизмами синхронизации, ко второй - те, в которых работает только процессор. В качестве примера цикла первой разновидности можно привести, скажем, такой, в котором набор символов передается на параллельный порт [обычно принтер - Прим. набивальщика]. Скорость исполнения программы ни при каких обстоятельствах не будет превышать темпа приема байтов параллельным портом, а быстродействие этого порта по крайней мере на два порядка ниже, чем время исполнения любой приемлемой кодовой последовательности управления портом. Вы, конечно, можете ради собственного удовольствия заняться оптимизацией подобных циклов, но для дела лучше поискать точку приложения сил в другом месте. Такой точкой вполне могут стать циклы второй категории - свободные от взаимодействия с "внешним миром".

Для полной оптимизации циклов необходим методический подход к проблеме. Прежде всего тщательно проверьте каждый из циклов с целью отыскания операций, которые никак не связаны с переменной цикла, и разгрузите цикл от этих вычислений (в большинстве случаев соответствующие команды удается разместить перед циклом). Проанализируйте оставшиеся коды и по возможности упростите их, применяя просмотр управляющих таблиц, ориентированные на конкретную модель процессора команды, откажитесь от универсальности и примените любые другие мзвестные вам приемы, которые позволят сократить кодовые последовательности и избавиться от "дорогостоящих" команд. Если в каких-то вычислениях используется текущее значение переменной цикла, попытайтесь вывернуть ситуацию наизнанку, рассчитывая нужные величины из начального и конечного значений переменной цикла, т.е. без перебора.

В качестве примера рассмотрим не слишком удачную программу, которая суммирует все кратные 5 элементы массива с однократной точностью и оставляет результат в регистре AX:

```
items     equ   100
array     dw    items dup (?)
          .
          .
          xor    cx, cx       ; инициализация счетчика
```

```
            xor    ax, ax       ; инициализация суммы

L1:         mov    bx, bx       ; формирование указателя
            add    bx, bx       ; * 2
            add    bx, bx       ; * 4
            add    bx, bx       ; * 5
            add    bx, bx       ; * 10
            add    ax, [bx+array]
            inc    cx
            cmp    cx, (items / 5)
            jne    L1
```

Упрстив этот цикл, мы получим:

```
items       equ   100
array       dw    items dup (?)
            .
            .
            xor    ax, ax       ; инициализация суммы
            mov    bx, offset array

L1:         add    ax, [bx]
            add    bx, 10
            cmp    bx, offset array + (items * 2)
            jb     L1
```

После того, как вы оптимизировали содержимое цикла насколько это было возможно, стоит посмотреть, не удастся ли где-то избавиться от управляющих циклом операций перехода и вызова подпрограмм. Идея здесь та же, что и при оптимизации переходов и вызовов, которые мы обсуждали в предыдущей статье. Суть дела в том, что в процессорах серии 80x86 фирмы Intel интенсивно используется простая конвейерная система, состоящая из шинного интерфейса, очереди упреждающей выборки, в которую поступают ждущие исполнения и извлекаемые из памяти команды, и исполнительного устройства. получающего информацию из очереди для декодирования и исполнения. При обнаружении перехода или вызова подпрограммы очередь очищается и все циклы памяти, котрые потребовались для заполнения позиций в ней после командв передачи управления, пропадают зря. При этом исполнительное устройство должно ожидать, пока шинный интерфейс извлечет и передаст в очередь новые команды из новых адресов. Так что переходы и вызовы подпрограмм обходятся гораздо "дороже", чем может показаться, если просто подсчитать нужное для них число тактов, руководствуясь документацией фирмы Intel.

Один из способов избавиться от сравнений и условных переходов называется объединением или сращиванием циклов. При этом коды реорганизуются так, чтобы сделать один цикл из нескольких повторяющихся одинаковое число раз. Например, из двух циклов вида

```
     mov  cx, 100
L1:                 ; что-то делаем
     loop L1

     mov  cx, 100
L2:                 ; делаем что-то lheujt
     loop L2
```

часто можно сделать один:

```
     mov  cx, 100
L1:                 ; что-то делаем
                    ; делаем что-то lheujt
     loop L1
```

Другой способ избавится от циклов - "размотать" их, т.е. устранить управляющие циклом кодовые последовательности, просто повторив содержимое цикла нужное число раз. Это дает особенно хорошие результаты в тех случаях, когда время, необходимое для исполнения содержимого цикла, оказывается меньше, чем время выполнения операций, управляющих циклом. Например, цикл

```
        mov  cx, 3
L1:     add  ax, [bx]
        add  bx, 2
        loop L1
```

можно переписать так:

```
        add  ax, [bx]
        add  bx, 2
        add  ax, [bx]
        add  bx, 2
        add  ax, [bx]
        add  bx, 2
```

или даже так:

```
        add  ax, [bx]
        add  ax, [bx+2]
        add  ax, [bx+4]
```

"Разматывание" циклов - классический пример повышения быстродействия за счет объема необходимой памяти. Каждый раз, когда вы решаете, стоит ли прибегнуть к данному приему, следует посмотреть, насколько это оправдано с точки зрения длины цикла и числа его повторений с учетом доступных для вашей программы вычислительных ресурсов. Иногда "разматывание" удается применить в самых неожиданных точках программы. К примеру, те из нас, кому доводилось составлять программы на языке ассемблера для процессоров Intel 80x86, научились распознавать ситуации, в которых можно использовать специальные команды обработки символьных последовательностей. Достаточно часто мы включали в свои программы примено такие коды:

```
        mov  cx, 3
        mov  si, offset string1
        mov  di, offset string2
rep     movsw
```

В приведенной последовательности иммется цикл, хотя и неявный. При обработке префикса REP требуется определенное время на установку начальных параметров, а результат исполнения REP в точности такой же, как если бы мы написали:

```
L1:     movsw
        loop L1
```

При работе на некоторых процессорах фирмы Intel и при небольшом числе итераций часто оказывается лучше не пользоваться префиксом REP, а выписать команды обработки строк подряд: movsw movsw movsw

Но это один из случаев, когда требуется выяснить, насколько хорош будет тот или иной вариант в условиях вашей конкретной программы. Время исполнения любого из вышеприведенных фрагментов невозможно точно определить, подсчитывая такты и байты команд, так как это время зависит еще и от программного контекста, а также от аппаратных характеристик системы: разрядности шины памяти, глубины упреждающей очереди команд, наличия или отсутствия и объема кэш-памяти процессора и т.д.

Управляющие таблицы

Мы уже отмечали, что почти всегда бывает целесообразно перенести вычисления из цикла за его пределы (или из "глубоко" вложенного цикла во внешний цикл), а также отсрочить вычисления до тех пор, пока их результаты реально не потребуются. Еще более эффективный вариант оптимизации состоит в том, чтобы приурочить вычисления не ко времени исполнения программы, а к моменту ассемблирования, либо выполнять вычисления с помощью специализированной программы, сохранять результаты в промежуточном временном файле и извлекать их оттуда по мере необходимости. Особенно удобная категория оптимизации этого типа называется просмотром управляющих таблиц.

Рассмотрим прикладную систему, в которой будет особенно целесообразно использовать управляющие таблицы. Представьте себе программу, в которой требуется поворачивать и перемещать отрезки линий, чтобы создать у пользователя иллюзию объемного изображения. Такая программа, естественно, должна определять синусы и косинусы углов. Для вычисления этих функций обычно применяют числа с плавающей точкой и разложение в ряды, расчет которых влечет за собой многократные умножения и деления, а эти операции с точки зрения времени счета "стоят" недешево. Кроме того, получаемые таким способом величины имеют значительно более высокую точность, чем это реально требуется для обычных графических адаптеров персональных компьютеров: даже числа с плавающей точкой одинарной точности (32 разряда) вычисляются с точностью до восьми десятичных знаков, из которых нужны только четыре или пять (разрешение графики в лучшем случае 1024 x 768 элементов изображения [требует адаптера SuperVGA с объемом памяти 1 Мбайт - Прим. набивальщикаъ).

Именно здесь и можно воспользоваться преимуществами таблицы, в которую можно занести синусы углов с шагом в 1 градус и с точностью до четырех десятичных знаков.

Прежде всего образуем структуру данных, в которой номер каждого элемента будет соответствовать углу в градусах, а значение - синусу этого угла, умноженного на 10000:

```
table     dw   0000 ; sin(0)
          dw   0175 ; sin(1)
          dw   0349 ; sin(2)
          .
          .
```

Подготовив такую таблицу, мы без труда можем составить неболшую подпрограмму, которая будет принимать значение угла в диапазоне 0 - 359 градусов в регистре AX, а в ответ выдавать значение синуса этого угла в том же регистре:

```
; Функция SINE
; При вызове AX = угол в градусах
; При возврате управления AX = 10000 * sin
sine      proc near
          push bx
          mov  bx, ax
          add  bx, bx
          mov  ax, [bx+table]
          pop  bx
          ret
sine      endp
```

Для дополнительного ускорения работы своей программы мы можем преобразовать подпрограмму в макроопределение, чтобы коды вставлялись в программу везде, где это потребуется. На рис. 2 показана более общая форма данной процедуры и ссответствующая процедура расчета косинуса.

Иногда управляющие таблицы можно весьма эффективно использовать в ситуациях, где мысль о них просто не приходит в голову. Пусть, например, вам поручено составить подпрограмму, которая будет подсчитывать число ненулевых разрядов в байте. Первое побуждение обычно состоит в том, чтобы составить цикл со сдвигами и действительно подсчитывать ненулевые разряды. Но ведь значительно быстрее будет воспользоваться таблицей, позиции в которой будут соответствовать значениям байта - от 0 до 255, а значения в этих позициях - числу ненулевых разрядов для каждого из таких значений байта:

```
table     db   0    ; ненулевые разряды в 00h
          db   1    ; ненулевые разряды в 01h
          db   1    ; ненулевые разряды в 02h
          db   2    ; ненулевые разряды в 03h
          .
          .
          .
          db   8    ; ненулевые разряды в FFh

; Функция BITS
; При вызове AL = значение байта
```

```
; При возврате AL = число ненулевых разрядов
bits      proc near
          push bx
          mov  bl, al
          xor  bh, bh
          mov  ax, [table+bx]
          pop  bx
          ret
bits      endp
```

И снова, чтобы еще более повысить быстродействие, можно оформить эту подпрограмму как макроконструкцию и встраивать в программу везде, где требуется. Для байтовых таблиц можно еще повысить производительность, замещая команды MOV на специальные команды обработки строк XLAT. Вместе с тем здесь необходимо подчеркнуть, что таким образом можно будет обрабатывать отнюдь не только байтовые таблицы. Мне довелось познакомиться с великолепной программой подсчета слов, (ее автор Терье Матисен), в которой использовалась таблица из 64 Кбайт для просмотра всех двухсимвольных комбинаций в поисках разделителей слов. Утверждается также, что Гордон Летуин применил аналогичную методику для сканирования битовой карты свободного пространства на диске в подсистеме обработки файлов HPFS [High-Performance File System - Прим. набивальщика] операционной системы OS/2.

Оптимизация для конкретных моделей ЦП

Если ваша программа будет работать только на компьютерах с конкретными моделями процессоров или вы ситаете, что есть смысл подготовить несколько версий программы для работы на разных машинах, можно попытаться воспользоваться ориентированными на конкретные модели процессоров командами.

По сравнению с процессорами 8086 и 8088, набор команд процессора 80286 ощутимо дополнен. Многие из новых команд позволяют повысить производительность программы:

- линейные и циклические сдвиги с непосредственным аргументом, отличным от единицы;
- команда PUSH с непосредственным операндом;
- команды обмена со стеком содержимым всех регистров - PUSHA и POPA;
- команды ENTER и LEAVE для выделения и освобождения кадра стека;
- команда проверки соблюдения границ массива BOUND.

Если ваша программа будет исполняться только на компьютерах с процессорами 80286 и более мощных, можно также выравнять по границе слов все элементы данных и целевые адреса для команд передачи управления. Таким образом можно повысить производительность на несколько процентов за счет относительно небольшого объема памяти. (При работе на процессорах 8086 выравнивание по границам слов также дает определенный выигрыш, но для процессоров 8088 с 8-разрядной шиной - это бессмысленно.)

Составляя программу для процессорв 80386DX и их разновидностей, можно повысить производительность 16-разрядной программы, пользуясь всеми вышеперечисленными командами для процессоров 80286 и, кроме того, выравнивая данные и адреса по границам 32-разрядных слов [т.е. DWORD], а также с помощью следующих дополнительных особенностей:

- 32-разрядных регистров (но пользоваться ими следует с осторожностью, так как их содержимое не сохраняется при работе с некоторыми эмуляторами системы DOS, например, модулем совместимости с DOS системы OS/2 версий до 1.3);
- команд пересылки с распространением нуля или знакового бита (MOVZX или MOVSX);
- установки в байте значений "истина" или "ложь" в зависимости от содержимого флажкоы центрального процессора, что позволяет избавиться от команд условного перехода (SETZ, SETC и т.д.);
- команд проверки, установки сброса и просмотра битов (BT, BTC, BTR, BTS, BSP, BSR);
- обобщенной индексной адресации и режимов адресации с масштабированием индексов;
- быстрого умножения при поможи команды LEA с использованием масштабированной индексной адресации;
- "дальних" условных переходов;
- перемножения 32-разрядных чисел и деления 64-разрядного числа на 32-разрядное;
- дополнительных сегментных регистров (FS и GS).

Естественно, выжать максимум из архитектуры 80386 можно при помощи расширителей DOS или операционной системы OS/2 2.0, переводя вашу программу в защищенный режим. В резульате вы получите доступ ко всем 32-разрядным регистрам, расширенной адресации и к адресному пространству до 4 Гбайт.

В процессоре 80486 предусмотрено всего две дополнительных команды, которых нет у процессоров 80386 и к которым можно обратиться из прикладной программы:

- BSWAP (изменение порядка расположения 4 байтов в 32-разрядном слове на противоположное);
- CMPXCHG (сравнение и обмен - для работы с семафорами).

Вместе с тем длительность команд, выраженная в числе тактов, у процессоров 80486 совсем не такая, как у процессоров 80386 и 80286. В общем, можно сказать, что что время исполнения простейших команд было сокращено за счет более сложных [RISC-архитектура (Reduced Instruction Set Computing - вычисления с сокращенным набором команд) - Прим. набивальщика]. Это означает, что программа, первоначально составленная для процессора 8088, возможно, будет работать быстрее на процессоре 80486, чем ее аналог, в исходном варианте предназначавшийся для процессоров 80286 или 80386. Такова жизнь. И еще кое-что, о чем стоит подумать, работая с процессором 80486: встроенная кэш-память объемом 8 Кбайт, которая работает лучше всего, если данные выравнены по ганицам 16-байтных параграфов, а также встроенный математический сопроцессор. К сожалению, недавний выпуск фирмой Intel процессоров 80486SX без (функционирующего) математического сопроцессора означает, что гарантированной возможности работать с числами с плавающей точкой на самых высокопроизводительных моделях компьютеров у нас по-прежнему нет.