

Графика Майкла Абраша

Черная книга по программированию, Специальное издание

Майкл Абраш

1. Введение

2. Предисловие

3. Благодарность

4. Часть I

1. Глава 1. Лучший оптимизатор находится у вас между ушами

- Человеческий фактор оптимизации кода
- Понимание высокой производительности
- n Правила построения высокопроизводительного кода
 - Где мы были, что мы видели

2. Глава 2 – Другой мир

The Уникальная природа оптимизации языка ассемблера

- Инструкции: Индивидуум против коллектива
- Собрание принципиально отличается

The Гибкий ум

3. Глава 3 – Ничего не предполагать

- Понимание и использование zen Timer

The Издержки незнания

The Zen Timer

- Время и КОМПЬЮТЕР
- Остановка Zen Timer

Отчет о результатах синхронизации

- Заметки о Zen Timer

n Пример использования дзен-таймера

Долговременный дзен-таймер

- Пример использования долговременного дзен-таймера
- Использование дзен-таймера из C

4. Глава 4 – В логове пожирателей циклов

- n Как аппаратное обеспечение ПК снижает производительность кода
 - пожиратели циклов

! Природа пожирателей циклов

пожиратель циклов на 8-битной шине

- Пожиратель циклов очереди предварительной выборки

■ Динамическое обновление оперативной памяти: невидимая рука

- Состояния ожидания

The Пожиратель циклов адаптера дисплея

5. Глава 5 – Пересечение границы

- Поиск файлов с перезапускаемыми блоками
- Как избежать ловушки строк
- Методы грубой силы
- Использование memchr ()
- Интерпретация того, куда идут циклы

- [Всегда смотрите, куда идет выполнение](#)

6. Глава 6 – Взгляд за пределы номинального значения

- n [Как Машинные инструкции Могут Делать Больше, чем Вы Думаете](#)
- [Математика с помощью адресации в памяти](#)
- [Умножение на LEA с использованием степеней, отличных от двух](#)

7. Глава 7 – Локальная оптимизация

- n [Оптимизация на полпути между алгоритмами и подсчетом циклов](#)
- [Уроки LOOP и JCXZ](#)
- [Локальная оптимизация](#)
- [Разворачивание циклов](#)

8. Глава 8 – Ускорение C с помощью языка ассемблера

- [Переключаешь языки, когда знаешь, что это поможет](#)
- n [Не вызывай у меня свои функции, детка](#)
- [Стек фреймов Слишком медленный](#)
- [Разрываешься между двумя сегментами](#)
- [Доводим дело до конца](#)

9. Глава 9 – Советы, которые дали мне мои читатели

- n [Коэффициенты оптимизация и заканчивается с поля](#)

10. Глава 10 – Кодирования Пациент, Быстрее Код

- [Как быстрая работа может привести к обходу выполнения](#)
- B i [Синдром грубой силы](#)
- [Рекурсия](#)

11. Глава 11 – Нажатие кнопок 286 и 386

- [Новые регистры, новые инструкции, новые тайминги, новые сложности](#)

12. Глава 12 – Запуск 486-го

- [Это не просто увеличенный 386-й](#)
- [Правила для оптимизации с помощью](#)
- [Предостережение программиста](#)
- B [История продолжается](#)

13. Глава 13 – Запуск 486-го

- [Конвейеры и другие опасности высокого уровня](#)
- n [BSWAP: полезнее, чем вы могли подумать](#)
- [Загрузка памяти](#)
- [Оптимальные 1-битные сдвиги и ротации](#)
- n [32-битные режимы адресации](#)

14. Глава 14 – Поиск строк Бойера-Мура

- n [Оптимизация довольно оптимального алгоритма поиска](#)
- [Обновление поиска по строкам](#)
- B [Алгоритм Бойера-Мура](#)
- [Бойер-Мур: хорошее и плохое](#)
- [Дальнейшая оптимизация Boyer-Moore](#)
- [Найдите то, что знаете вы](#)

15. Глава 15 – Связанные списки и простые непреднамеренные проблемы

- [Незнакомые проблемы со знакомыми структурами данных](#)
- [Связанные списки](#)
- n [Макеты и стражи](#)
- [Циклические списки](#)
- [Hi / Lo размером в 24 байта](#)

16. Глава 16 – Не существует такого понятия, как самый быстрый код

- n [Уроков, извлеченных в погоне за совершенным счетчиком слов](#)
- [Подсчет слов в спешке](#)
- [Проблемы и опасности](#)
- [Удивление от оптимизации правым полушарием](#)
- [Уровни оптимизации](#)
- [Уровень 2: новый взгляд](#)

17. Глава 17 – Игра жизни

- n Триумф алгоритмической оптимизации в клеточной игре Игра с автоматами
 - Игра Конвея
 - Куда уходит время?
 - Опасности и преимущества абстракции
 - Мощная оптимизация на C ++
- Задействие правого полушария мозга

18. Глава 18 – Это простая замечательная жизнь

- n Запредельная оптимизация
- Нарушение правил
- . Магия на основе таблиц
- Отслеживание изменений с помощью списка изменений

19. Глава 19 – Pentium: не та старая песня

- Изучение совершенно другого набора правил оптимизации
- The Превращение оптимизации в искусство
- The Pentium: обзор
- Более быстрая адресация и многое другое
- Прогнозирование ветвлений
- Разные темы по Pentium

20. Глава 20 – Правила Pentium

- n Как ваш оптимизатор на основе углерода может поместить “Super” в суперскалярный формат
- Инструкции в каждом канале
- Инструкции с поддержкой V-образных каналов
- Пошаговое выполнение
- Суперскалярные примечания

21. Глава 21 – Высвобождение V-образной трубы Pentium

- Сосредоточение внимания на обеспечении полной загрузки обоих каналов Pentium
- n Блокировки генерации адресов
- Конкуренция в реестре
- Кто первый?
- Оптимизация Pentium в действие

22. Глава 22 – Zenning и гибкий ум

- Прокатившись через то, что вы узнали
- Zenning

5. Часть II

1. Глава 23 – кости и сухожилия

- В самом сердце стандартной компьютерной графики
- В VGA
- Ан Введение в программирование VGA
- В основе
- The Опасности клонов VGA
- Только начало
- The Ассемблер макросов

2. Глава 24 – Параллельная обработка с помощью VGA

- Загрузка графической памяти по четыре байта за раз
 - Программирование VGA: ALU и защелки
- Примечания к демонстрационной программе ALU / Latch

3. Глава 25 – Обработка данных VGA

- Переключатель ствола, битовая маска и механизмы настройки / сброса
 - Вращение данных VGA
- The Битовая маска
- The Схема настройки / сброса VGA
- Примечания по настройке / сбросу
- А Краткое примечание по завершению слов

4. Глава 26 – режиме VGA Запись 3

В Режим записи, который вам нравится все больше

Режим, рожденный в непривычке

n Примечание о сохранении битов регистра

5. Глава 27 – Еще один режим записи VGA

Режим записи 2, сжатые растровые изображения и сосуществование текста и графики

- Режим записи 2 и настройка / сброс
- Когда использовать режим записи 2 и когда использовать настройку / сброс

Режим 13H—320x200 с 256 цветами

- Переключение страниц с текста на графику и обратно

6. Глава 28 – Считывание данных с памяти VGA

Режимы чтения 0 и 1 и регистр "Цвет не важен"

- Режим чтения 0

Режим чтения 1

- Когда всем плоскостям “Все равно”

7. Глава 29 – Сохранение экранов и другие тайны VGA

- Полезные фрагменты из файла VGA Zen
- Сохранение и восстановление экранов EGA и VGA

16 Цвета из 64

Повторное сканирование

- Дополнительный блокиратор

Изменение регистров VGA

8. Chapter 30 – Video Est Omnis Divisa

- Радости и досадные проблемы использования разделенных экранов на EGA и VGA
- Как работает разделенный экран

n Настройка регистров, связанных с разделенным экраном

- Проблема с разделенным экраном EGA
- Разделение экрана и панорамирование

Примечания по настройке и считыванию регистров

- Разделение экранов в других режимах

Насколько безопасно?

9. Глава 31 – Более высокое разрешение 256 цветов на VGA

- Когда 320x200 на самом деле 320x400?
- Почему 320x200? Только IBM знает наверняка

n 256-цветной режим 320x400

Две 256-цветные страницы

Есть о чем подумать

10. Глава 32 – Будет ли это решено: 360x480

- n** Использование 256-цветовых режимов Примерно на том уровне, на который способен стандартный VGA
- Они
- Расширенные 256-цветовые режимы: что здесь не нравится?
- 360x480 256-цветном режиме
- 360x480 как 256-цветный режим работы

11. Глава 33 – медведь йоги и художественной гимнастике противостоять разъем VGA цветов

В Основы цветопередачи VGA

Основы цветопередачи VGA

Если вы не можете вызвать BIOS, кому вы собираетесь позвонить?

В Пример настройки ЦАП

12. Глава 34 – Изменение цветов без записи пикселей

Специальные эффекты за счет манипулирования цветами DAC в реальном времени

Переключение цветов

Суть проблемы

- Тестовая программа для переключения цветов
- n** Работаящие подходы к переключению цветов
- Разные варианты

13. Глава 35 – Bresenham быстро, а быстро это хорошо

Реализация и оптимизация линейного рисования Брезенхэма Алгоритм

В Текущая задача

n Алгоритм линейного рисования Брезенхэма

- Реализация на C

Комментарии к реализации на C

- Алгоритм Брезенхэма в ассемблере

14. Глава 36 – Хорошее, плохое и запущенное - Нарезанный

n Более быстрые линии Брезенхэма с рисованием линий отрезков длиной в длину

Основы построения срезов длиной в длину

n Реализация срезов длиной в длину

- Подробности о срезах длиной в длину

15. Глава 37 – Мертвые кошки и линии молний

n Важная оптимизация рисования отрезков по длине

- Быстрое рисование отрезков по длине

16. Глава 38 – Первичный полигон

n Эффективное и быстрое рисование полигонов

- Заполненные полигоны

Заполнение неперекрывающихся выпуклых многоугольников

- Необычные случаи

17. Глава 39 – Быстрые выпуклые многоугольники

. Заполнение полигонов в спешке

- Быстрое заполнение выпуклых полигонов

The Последний штрих: язык ассемблера

Более быстрая трассировка ребер

18. Глава 40 – О композициях, налогах и простоте сложных полигонов

- Работа с неправильными многоугольными областями

- Заполнение произвольных многоугольников

Сложное заполнение полигонов: реализация

Невыпуклые многоугольники

19. Глава 41 – Эти синие цвета номенклатуры многоугольников снизу вверх

Имена имеют значение при концептуализации структуры данных

Номенклатура в действии

20. Глава 42 – Приготовленный в спешке; Поджаренный, Тушеные на досуге

- Быстрые сглаженные линии с использованием алгоритма Wu

Сглаживание Wu

Трассировка и интенсивность в одном экземпляре

Пример сглаживания Wu

21. Глава 43 – Анимация в битовой плоскости

n Простой и чрезвычайно быстрый метод анимации с ограниченным количеством цветов

- Битовые плоскости: основы

Битовая анимация в действии

- Ограничения битовой анимации

- Сдвиг и перелистывание страниц

n Победа в потрясающем конкурсе

22. Глава 44 – Разделенные экраны Сохраняют день перелистывания страницы

n Анимация с перелистыванием страниц размером 640x480 в 64 КБ...Почти

23. Глава 45 – Собачья шерсть и грязные прямоугольники

- Разные ракурсы анимации

- Plus ça Change

Время доступа к VGA

Анимация грязных прямоугольников

n Грязные прямоугольники в действии

- Перелистывание страниц VGA в высоком разрешении

- Еще один интересный поворот в перелистывании страниц

24. Глава 46 – Кто был тем изображением в маске?

- n Оптимизация анимации грязных прямоугольников

25. Глава 47 – Режим X: магия 256-цветного VGA

Представляем недокументированный “Оптимальный для анимации” режим VGA

- Что делает Mode X особенным?
- n Выбор режима 320x240 с 256 цветами
- n Проектирование с точки зрения Mode X
- Аппаратная поддержка с неожиданной стороны

26. Глава 48 – Режим X Обозначает защелку

Особенности режима наилучшего отображения анимационного видео

- Выделение памяти в режиме X
- n Копирование блоков пикселей в памяти дисплея
- Кто был тем копировщиком изображений в маске?

27. Глава 49 – Режим X 256-цветная анимация

- n Как заставить VGA действительно встать и танцевать
- Скрытое копирование

Анимация

Анимация в режиме X в действии

Работает быстро, выглядит великолепно

28. Глава 50 – Добавление размера

- Трехмерная анимация с использованием режима X

Ссылки на трехмерное рисование

- Конвейер трехмерного рисования

Простой трехмерный пример

В Продолжающееся путешествие

29. Глава 51 – Кроссовки в космосе

Удаление обратной поверхности для устранения скрытых поверхностей

- n Односторонние полигоны: удаление обратной поверхности

Постепенное преобразование

- n Примечание по округлению отрицательных чисел

Представление объектов

30. Глава 52 – Быстрая трехмерная анимация: знакомьтесь, X-Sharp

- n Первая итерация обобщенного пакета трехмерной анимации

Это Демонстрационная программа главы

А Новый фреймворк анимации: X-Sharp

Три ключа к производительности анимации в реальном времени

31. Глава 53 – Скорость Raw и многое другое

The Голая правда о скорости в трехмерной анимации

Скорость Raw, часть 1: язык ассемблера

Скорость Raw, часть II: посмотрите сами

32. Глава 54 – Трехмерное затенение

- Создание реалистичных поверхностей на анимированных трехмерных объектах

Поддержка старых процессоров

33. Глава 55 – Цветовое моделирование в 256-цветном режиме

Размышления о цветовой модели X-Sharp в формате RGB мнение

34. Глава 56 – Пух и космическая станция

- Использование быстрого отображения текстур для размещения Пуха на полигоне

Принципы быстрого отображения текстур

- Быстрое отображение текстур: реализация

35. Глава 57 – 10,000 недавно стригли овец на экране

В Важнейшая роль опыта в создании быстрой и гладкой текстуры

Отображение

36. Глава 58 – Хрустальный шар Хайнлайна, мозг Спока, и 9-цикловый вызов

- n Использование подхода "Всего мозга" для ускорения отображения текстур

Сокращение отображения текстур

Это здорово, но, черт возьми, это точно не 9 циклов

Примечания по отображению текстур

37. Глава 59 – Идея BSP-деревьев

- Что такое BSP-деревья и как ходить по ним
- BSP-деревья

Построение BSP-дерева

Упорядоченные прогулки по BSP-деревьям

- Серфинг среди деревьев

38. Глава 60 – Составление деревьев BSP

Превращение деревьев BSP из концепции в реальность

- Составление деревьев BSP

Оптимизация дерева BSP

- Оптимизация BSP: неизведанная страна

39. Глава 61 – Системы отсчета

B Основы математики, лежащие в основе трехмерной графики

- Скалярное произведение

n Перекрестные произведения и генерация нормалей к полигонам

- Использование знака скалярного произведения
- Использование скалярного произведения для проекции

■ Вращение с помощью проекции

40. Глава 62 – Одна история, два правила и средство визуализации BSP

n Преобразование скомпилированного дерева BSP из логической реальности в визуальную

Перемещение программы просмотра

- Преобразование в Viewspace

Обрезка

Проекция в Screenspace

- Хождение по дереву, отбраковка обратной поверхности и рисование

41. Глава 63 – Вычисление с плавающей запятой для 3D в реальном времени

n Знание того, когда нужно выбросить обычные математические знания в окно

Не отцовский алгоритм с плавающей запятой

n Оптимизация Pentium с плавающей запятой

! Скалярное произведение

- Перекрестное произведение

Преобразование

Проекция

Управление округлением

Прощание с трехмерной системой с фиксированной точкой

42. Глава 64 - Определение видимой поверхности Quake

n Задача отделения всего видимого от всего невидимого

n VSD: самая сложная трехмерная задача из всех

Структура уровней землетрясения

- Отбор проб и определение видимой поверхности

Перерисовка

B Дерево балок

■ Современный движок 3D

Прорыв!

■ Упрощайте и продолжайте пробовать что-то новое

Учитесь сейчас, платите вперед

Ссылки

43. Глава 65 – Трехмерная вырезка и другие мысли

n Определение того, что находится в поле вашего зрения

- Основы трехмерной обрезки

Обрезка полигонов

n Преимущества обрезки видового пространства

Читать далее

44. Глава 66 – Удаление землетрясения со скрытой поверхности	
n Проблемы с решениями для скрытой поверхности в порядке Z	Проблема
n Creative Flux и скрытые поверхности	
Сортированные промежутки	
Ребра в сравнении с промежутками	
Ключи сортировки по краям	
n Отложенные решения	
45. Глава 67 – Сортированные промежутки в действии	
• Реализация независимой сортировки промежутков для рендеринга без перерисовки	
Quake и сортированные промежутки	
Типы сортировки по 1 / z промежуткам	
• 1 / Сортировка по z-диапазону в действии	
46. Глава 68 – Модель освещения Quake	
n Радикально иной подход к освещению полигонов	
• Головоломка освещения	
Затенение Гуро	
The Поиск альтернативного освещения	
Поверхностное кэширование	
47. Глава 69 – Поверхностное кэширование и модели треугольника Quake	
• Зондирование поверхностей с помощью аппаратного обеспечения и быстрая анимация модели без Спрайты	
Кэширование поверхности с помощью аппаратного обеспечения	
Рисование моделей треугольников	
48. Глава 70 – Quake: Вскрытие и взгляд в будущее	
n Предварительная обработка мира	
Потенциально видимый набор (PVS)	
n Отрывки: изменение в последнюю минуту, которого не произошло	
Рисование мира	
Растрирование	
Сущности	
n Как мы провели наши летние каникулы: после отправки Quake	
С нетерпением ждем	
6. Послесловие	
7. Об этой версии	

Введение

Каково было работать с Джоном Кармаком над Quake? Как будто тебя пристегнули к ракете во время взлета — в разгар урагана. Казалось, что весь мир был наблюдал, ожидая, сможет ли id Software превзойти Doom; каждое случайное электронное письмо или разговор с посетителем в течение нескольких часов попадал в Интернет. И тем временем мы вкладывали все, что у нас было, в технологию Quake; я часто приходил утром Джон все еще был там, работая над новой идеей, настолько интригующей, что он не мог уснуть, пока не попробовал это. Ближе к концу, когда я потратил большую часть своего время ускоряло работу, я проводил день в трансе, сочиняя оптимизированную сборку код, выходил, пошатываясь, из Таун-Ист-Тауэр в палящую техасскую жару и каким-то образом езжу домой по автостраде LBJ, не врезавшись ни в один из мчащихся пикапов, проносящихся со свистом мимо меня с обеих сторон. Дома я засыпал прерывистым сном, затем возвращался на следующий день в оцепенении и делал это снова. Все произошло так быстро и под таким давлением, что иногда я удивляюсь, как кто-то из нас выдержал это, не перегорев полностью.

В то же время, конечно, это было невероятно захватывающе. Идеи Джона были бесконечны и блестящи, и Quake в конечном итоге установила новый стандарт для Интернета и 3d-игр от первого лица . К счастью, id просвещенно относится к обмену информацией и была готова позволить мне написать о технологии Quake — как о том, как она работала, так и о том, как она развивалась. За два года, что я проработал в id, я написал несколько колонок о Quake в справочнике доктора Добба, а также подробный обзор для Конференция разработчиков компьютерных игр 1997 года. Вы можете найти их во второй части этой книги; они представляют собой редкий взгляд на разработку и внутреннюю работу передовые разработки программного обеспечения, и я надеюсь, что вам понравится читать их так же, как и мне понравилось разрабатывать технологию и писать о ней.

Остальная часть этой книги - это практически все, что я написал за последнее десятилетие о графике и программировании производительности, которые все еще актуальны для программирования сегодня, и *это охватывает много вопросов. Большая часть Zen of Graphics Programming, 2nd Edition находится* в там (а остальное на компакт-диске); весь Zen of Code Optimization тоже там, и даже на компакт-диске представлена моя книга 1989 года "Дзен языка ассемблера" с устаревшим количеством циклов 8088, но множеством полезных точек зрения. Добавьте к этому последние 20 000 слов из Материалы Quake, и у вас есть большая часть того, что я узнал за последнее десятилетие, в одном аккуратный пакет.

Я рад, что все эти материалы напечатаны в одном месте., потому что за последние десять лет я столкнулся со многими людьми, которые нашли мои работы полезными — и многое другое кто хотел бы прочитать их, но не смог найти. Трудно продолжать программировать материалы (особенно те, которые начинались в виде колонок) в печатном виде очень долго, и я бы хотел бы поблагодарить группу Coriolis и особенно моего хорошего друга Джеффа Дантеманна (без которого не было бы не только этой книги, но и практически всей моей писательской карьеры exist), за помощь в сохранении доступности этого материала.

Я также хотел бы поблагодарить Джона Эриксона, редактора Dr. Dobb's, как за поддержку, так и за общее хорошее настроение и за то, что дали мне возможность написать все, что я хотел о реальном времени 3D. Меня до сих пор поражает, что я мог находить время для написания колонки каждые два месяца во время разработки Quake, и если бы Джон не сделал это таким легким и приятным, этого могло бы никогда не произойти.

Я также хотел бы поблагодарить Криса Хеккера и Дженнифер Палка из Computer Game Конференция разработчиков, без чьей поддержки, подталкивания и случайного благополучия заслуженное ворчание: у меня нет ни малейшего шанса, что я когда—либо написал бы статью для CGDC - статья, которая в итоге стала наиболее полным обзором технологии Quake это, скорее всего, когда-нибудь будет написано и появится на этих страницах.

Мне нечего сказать такого, чего еще не было сказано в других местах этой книги, в одно из введений к предыдущим томам или в одной из удивительно больших количество глав. Как вы увидите по мере чтения, прошло целое десятилетие для программистам микрокомпьютеров и мне чрезвычайно повезло не только быть частью этого, но и иметь возможность вести хронику части этого.

И следующее десятилетие обещает быть не менее захватывающим!

— Майкл Абраш
Бельвью, Вашингтон
Май 1997 г.

Предисловие

Я начал программировать на компьютерах Apple II в школе, и почти все мои ранние годы работал на платформе Apple. После окончания учебы быстро стало очевидно, что у меня были проблемы с оплатой аренды, когда я работал на рынке Apple II в конце восьмидесятых, поэтому я был вынужден очень быстро перейти в среду Intel PC.

То, чему я смог научиться за несколько лет работы в Apple, мне нужно было выучить за несколько месяцев работы на ПК.

Самым большим преимуществом для меня от реального заработка в качестве программиста была возможность покупать все книги и журналы, которые я хотел. Я покупал много. Я был на территории, которую знал почти ни о чем, поэтому я прочитал все, что смог достать. Характеристика статьи, передовицы, даже рекламные объявления содержали информацию, которую я должен был усвоить.

Джон Ромеро заранее ознакомил меня со статьями Майкла Абраша. Хорошие материалы. Графическое оборудование. Оптимизация кода. Знания и мудрость для начинающих Разработчик. Их было даже интересно читать. Долгое время моим личным стремлением было найти копию первой книги Майкла "Дзен языка ассемблера". Я заглянул во все книжные магазины, которые посетил, но так и не нашел ее. Я довольствовался теми статьями, которые смог откопать.

Там я узнал мрачные секреты видеоконтроллера EGA и разработал несколько изящных своих собственных трюков. Некоторые из этих трюков легли в основу Commander Keen серия игр, которая запустила id Software.

Год или два спустя, после Wolfenstein-3D, я впервые столкнулся с Майклом (в виртуальном смысле) . Я просматривал M & T Online, BBS, которым управляет Dr. Dobb's издатель до интернет-взрыва, когда я увидел несколько сообщений от самого человека. Мы обменивались электронной почтой и пару месяцев играли в гурь tag-team на графическом форуме до того, как разработка Doom заняла всю мою жизнь.

Друг Майкла на его новой работе вернул нам связь друг с другом после того, как Doom начал оказывать свое влияние, и у меня наконец-то появился шанс встретиться с ним лично.

В тот день я наговорился до хрипоты, объясняя Майклу и заинтересованной группе его коллег все тонкости Doom. Каждые несколько дней после этого я получал электронное письмо от Майкла с просьбой прокомментировать один из моих тезисов или обсудить аспект будущего графики.

В конце концов, я задал этот вопрос — я предложил ему работу в id. “Только подумай: никакой отчетности перед кем бы то ни было, возможность программировать весь день, начиная с чистого листа бумаги. Шанс поступить правильно как программист ”. Это не сработало. Однако я продолжал в том же духе около года позже я, наконец, убедил его спуститься и взглянуть на id. Я работал над Quake.

Переход от Doom к Quake был огромным шагом. Я знал, где хочу оказаться в конечном итоге, но мне было совсем не ясно, какие шаги нужно предпринять, чтобы достичь этого. Я перепробовал огромное количество подходы и даже неудачи многому меня научили. Должно быть, мой энтузиазм был заразителен, потому что он взялся за эту работу.

За этим последовало много героического программирования. Было написано несколько сотен тысяч строк кода. И переписано. И переписано. И переписано заново.

Оглядываясь назад, я испытываю множество сожалений по поводу различных аспектов Quake, но это редкость человек, который свободно не признает его технический триумф. Мы справились с задачей. Конечно, через год я, вероятно, найду новую перспективу, которая заставит меня съежиться от неуклюжести какой-то части Quake, но на данный момент она все еще выглядит чертовски хорошо по-моему.

Я был очень рад, что Майкл подробно описал технологию Quake в своих продолжающихся журнальных статьях. Мы многому научились, и я надеюсь, что нам удалось чему-то научить.

Когда непрограммист слышит о статьях Майкла или о выпущенном мной исходном коде , я обычно слышу ошеломленное “Ради чего бы ты это сделал”??? посмотри.

Они этого не понимают.

Программирование - это не игра с нулевой суммой. Обучая чему-либо коллегу-программиста у вас это не отнимет. Я рад поделиться тем, что могу, потому что я занимаюсь этим из любви к программированию. Ferrari - это просто соус, честно!

Эта книга содержит множество оригинальных статей, которые помогли мне начать заниматься программированием Карьера. Я надеюсь, что мой вклад в содержание последующих статей сможет послужить аналогичной отправной точкой для других.

—*Джон Кармак*
id Software

Благодарности

Нужно поблагодарить многих людей, потому что эта книга писалась много лет, в множество различных условий, необычайно большое количество людей сыграли свою роль в создании этой книги. Спасибо Дэну Илловски не только за то, что он поделился идеями и ободрение, но также и то, что давным-давно я начал писать статьи, когда мне не хватало уверенности, чтобы делать это самостоятельно, — и за то, что научил меня справляться с деловой частью вещей. Спасибо Уиллу Фасти за то, что дал мне возможность впервые попробовать писать для большой аудитории в давно вышедшем, но все еще скучающем по PC Tech Journal, и за то, что показал мне, как это весело могла бы быть в его еще более давно исчезнувшей, но по-настоящему потрясающей колонке в Creative Computing (самая приятная отдельная колонка, которую я когда-либо читал в компьютерном журнале; Я часто заглядывал в почтовый ящик в начале месяца, просто чтобы посмотреть, что скажет Уилл). Спасибо Роберту Келлеру, Эрин О'Коннор, Лиз Оукли, Стиву Бейкеру и остальным многотысячному составу, который сделал Programmer's Journal уникальным веселым журналом — особенно Эрин, которая сделала больше, чем кто-либо другой, чтобы научить меня правильно пользоваться английским язык. (По сей день Эрин все еще терпеливо объясняет мне, когда следует использовать “это” а когда “который”, несмотря на восемь лет обучения этому и связанные темы не оставили заметного отпечатка в моем мозгу.) Спасибо Тами Земель, Монике Берг и остальной команде журнала доктора Добба за отличное, профессиональное редактирование и просто за то, что вы отличные люди. Спасибо команде Coriolis за их неумолимость тяжелая работа: Джефф Дантеманн, Ким Еофф, Джоди Кент, Роберт Кларфилд и Энтони Сток. Спасибо Джеку Ценгу за то, что он многому научил меня о графическом оборудовании и даже большему о том, как много может изменить тяжелая работа. Спасибо Джону Кокерхэму, Дэвиду Стаффорд, Терье Матисен, БитМен, Крис Хекер, Джим Макраз, Мелвин Лафитт, Джон Навас, Фил Коулман, Антон Трюэнфелс, Джон Кармак, Джон Майлз, Джон Бриджес, Джим Кент, Хэл Харденберг, Дэйв Миллер, Стив Леви, Джек Дэвис, Дуэйн Стронг, Даев Рор, Билл Вебер, Дэн Гочнауэр, Патрик Миллиган, Том Уилсон, Питер Клерингс, Дэйв Метвин, Мик Браун, люди из раздела ibm.pc / fast.code на Vix и все остальные, кто были так щедры со своими идеями и предложениями. Я сделал все возможное, чтобы отметить авторов поименно в этой книге, но если ваше имя опущено, мой приношу свои извинения и считайте, что вы благодарны; эта книга не появилась бы на свет без вас. И, конечно, спасибо Шей и Эмили за их великодушное терпение по отношению к моей страсть к писательству и компьютерам.

Часть I

Глава 1. Лучший оптимизатор – это между вашими ушами

Человеческий фактор в оптимизации кода

Эта книга посвящена теме, близкой и дорогой моему сердцу: написанию программного обеспечения, которое расширяет возможности ПК до предела. Учитывая заурядное программное обеспечение, ПК работают как слабаки весом в 97 фунтов минikomпьютеры, которыми они и являются. Однако ухаживайте за ними должным образом, и эти уродливые коробки способны творить чудеса. Ключ вот в чем: только на микрокомпьютерах вы можете управлять вся машина целиком, без мешающих слоев операционных систем, драйверов и тому подобного . Вы можете делать все, что хотите, и понимать все, что происходит, если вы того пожелаете.

Как мы вскоре увидим, вам действительно следует этого пожелать.

Является ли производительность все еще проблемой в эту эпоху дешевых компьютеров 486 и сверхбыстрых Pentium компьютеры? Еще бы. Сколько программ, которые вы используете, на самом деле работают так быстро, что вы не были бы счастливее, если бы они работали быстрее? Мы настолько привыкли к медленному программному обеспечению, что когда последовательность компиляции и компоновки, занявшая две минуты на ПК, занимает всего десять секунд на мы в восторге от компьютера 486, хотя, по правде говоря, нам следовало бы довольствоваться не меньшим, чем мгновенный ответ.

Вы говорите, невозможно? Не при правильном дизайне, включая инкрементную компиляцию и компоновку, использование расширенной и / или расширенной памяти и хорошо проработанный код. КОМПЬЮТЕРЫ могут делать практически все, что вы только можете себе представить (за несколькими очевидными исключениями, такими как приложения с использованием суперкомпьютера (вычислительный класс), если вы верите, что это возможно быть реализованным, если вы понимаете компьютер изнутри и снаружи, и если вы готовы думать мимо очевидного решения к нетрадиционным, но потенциально более плодотворным подходам.

Моя точка зрения проста: ПК могут творить чудеса. Убедить их в этом нелегко это, но это полезно — и, несомненно, чертовски весело. В этой книге мы собираемся поработать с некоторыми из этих чудес, начиная...

... сейчас.

Понимание высокой производительности

Прежде чем мы сможем создать высокопроизводительный код, мы должны понять, что такое высокая производительность. Цель (не всегда достигаемая) при создании высокопроизводительного программного обеспечения должно обеспечивать возможность программного обеспечения выполнять поставленные перед ним задачи настолько быстро, что оно реагирует мгновенно, насколько это касается пользователя. Другими словами, высокопроизводительный в идеале код производительности должен выполняться так быстро, чтобы любое дальнейшее улучшение кода было бы бессмысленным.

Обратите внимание, что в приведенном выше определении ничего не говорится о создании программного обеспечения как можно быстрее. В нем также ничего не говорится об использовании ассемблера язык, или оптимизирующий компилятор, или, если уж на то пошло, компилятор вообще. Это также не ничего не говорит о том, как был разработан и написан код. Что он говорит, так это то, что высокопроизводительный код не должен мешать пользователю — и это все.

Это важное различие, потому что слишком многие программисты думают, что сборка

язык, или правильный компилятор, или конкретный язык высокого уровня, или определенный дизайн подход - это решение для создания высокопроизводительного кода. Они не более чем выбор определенного набора инструментов является ключом к строительству дома. Вам действительно нужны инструменты чтобы построить дом, но подойдет любой из множества наборов инструментов. Вам также понадобится план, понимание всего, что входит в дом, и умение пользоваться инструментами.

Аналогично, высокопроизводительное программирование требует четкого понимания цели

создаваемого программного обеспечения, общего дизайна программы, алгоритмов реализации

конкретные задачи, понимание того, что может делать компьютер и что делает все соответствующее

программное обеспечение и твердые навыки программирования, предпочтительно с использованием оптимизирующего

компилятора или языка ассемблера. Однако оптимизация в конце - это всего лишь завершающий штрих.



Без хорошего дизайна, хороших алгоритмов и полного понимания работы программы ваш тщательно оптимизированный код будет представлять собой один из наименее плодотворное творение человечества — быстрая медленная программа.

“Что такое быстрая медленная программа?” - спросите вы. Хороший вопрос и краткая (правдивая) история возможно, это лучший ответ.

Когда быстро не значит быстро.

В начале 1970-х, когда на рынке появились первые ручные калькуляторы, я знал парня по имени Ирвин. Он был хорошим учеником и планировал стать инженером. Быть инженером в то время означало знать, как пользоваться логарифмической линейкой, а Ирвин умел обращаться с сразись с лучшими из них. На самом деле, он был настолько хорош, что вызвал на дуэль парня с калькулятором — и победил, став при этом местной легендой.

Однако, если разобраться, Ирвин плевал на ветер. Через несколько коротких годы, когда его с трудом заработанные навыки игры в салки обесценились бы, и вся дисциплина была бы, по сути, стерта с лица земли. Более того, любой, у кого есть хоть половина мозгов, мог предвидеть грядущую смену. Ирвин, по сути, потратил впустую значительные усилия и время, которые он потратил на оптимизацию своих скоро устаревших навыков.

Какое отношение все это имеет к программированию? Много. Когда вы тратите время оптимизация плохо разработанного ассемблерного кода или когда вы рассчитываете на оптимизирующий компилятор чтобы сделать ваш код быстрым, вы тратите оптимизацию впустую, как это сделал Ирвин. Особенно в ассемблере вы обнаружите это без надлежащего предварительного проектирования и всего остального, что к нему прилагается в высокопроизводительном дизайне вы потратите значительные усилия и время на создание изначально медленной программы настолько быстрой, насколько это возможно — которая все еще медленная, — когда вы могли бы легко немного подумав, вы значительно улучшили производительность. Как мы увидим, созданный вручную язык ассемблера и оптимизирующие компиляторы имеют значение, но меньше, чем вы можно подумать, по большому счету — и они вряд ли имеют какое-либо значение, если только они используются в контексте хорошего дизайна и глубокого понимания обоих текущая задача и компьютер.

Правила создания высокопроизводительного кода

У нас есть следующие правила создания высокопроизводительного программного обеспечения:

- Знайте, куда вы направляетесь (понимайте цель программного обеспечения).
- Составьте большую карту (четко представляйте общий дизайн программы, чтобы различные части программы и структуры данных хорошо работают вместе).
- Создайте множество небольших карт (разработайте алгоритм для каждой отдельной части общего дизайна).
- Знайте территорию (точно понимать, как компьютер выполняет каждую задачу).
- Знайте, когда это важно (определите части ваших программ, где важна производительность , и не тратьте время на оптимизацию остальных).

- Всегда рассматривайте альтернативные варианты (не за цикливайтесь на одном подходе; шансы велики есть способ получше, если вы достаточно умны и изобретательны).
- Знайте, как включить сок (оптимизируйте код настолько, насколько вы знаете, когда это имеет значение).

Создавать правила легко; самое сложное - выяснить, как применять их в реальном мире. За мои деньги изучение какого-нибудь реально работающего кода - всегда хороший способ получить разберитесь с концепциями программирования, поэтому давайте рассмотрим некоторые правила производительности в Экшен.

Знай, Куда Ты Идешь

Если мы собираемся создать высокопроизводительный код, сначала мы должны знать, что этот код будет делать . В качестве примера давайте напишем программу, которая генерирует 16-битную контрольную сумму из байтов в файле. Другими словами, программа добавит каждый байт в указанный файл в превратит в 16-битное значение. Это значение контрольной суммы может использоваться для проверки того, что файл не был поврежден, как это могло бы произойти во время передачи через модем или если троянец вирус-конь поднимает свою уродливую голову. Мы не собираемся ничего делать со значением контрольной суммы однако, кроме как распечатать его; прямо сейчас нас интересует только его генерация значение контрольной суммы как можно быстрее.

Создайте большую карту.

Как мы собираемся сгенерировать контрольную сумму для указанного файла? Логический подход заключается в том, чтобы получить имя файла, открыть файл, прочитать байты из файла, сложить их вместе и распечатать результат. Большинство из этих действий просты; единственная сложность часть заключается в чтении байтов и сложении их вместе.

Создайте множество маленьких карт.

На самом деле, мы собираемся создать только одну небольшую карту, потому что у нас есть только одна программа раздел, который требует много размышлений — раздел, который считывает байты и суммирует их. Каков наилучший способ сделать это?

Было бы удобно загрузить весь файл в память, а затем суммировать байты за один цикл. К сожалению, нет гарантии, что какой-либо конкретный файл поместится в доступной памяти; на самом деле, многие файлы точно не поместятся в память, поэтому этот подход out.

Что ж, если весь файл не поместится в память, один байт наверняка поместится. Если мы прочитаем файл один байт за раз, добавляя каждый байт к контрольной сумме перед чтением следующего байта, мы минимизируем требования к памяти и сможем обрабатывать файлы любого размера вообще.

Звучит заманчиво, а? В листинге 1.1 показана реализация этого подхода. Листинг 1.1 использует функцию read() С для чтения одного байта, добавляет этот байт к контрольной сумме, и выполняет обратный цикл для обработки следующего байта до тех пор, пока не будет достигнут конец файла. Код компактный, простой в написании и отлично функционирует — с одной небольшой заминкой:

Он медленный.

ЛИСТИНГ 1.1 L1-1.C

```
>* Program to calculate the 16-bit checksum of
all bytes in the * specified file. Obtains
the bytes one at a time via
read(), * letting DOS perform all data
buffering. */ #включить <stdio.h>
#включить <fcntl.h>
main(int argc, char *argv[]) {
    дескриптор int;
    байт символа без знака;
```

```
<unsigned int Checksum;

int ReadLength;

if ( argc!= 2 ) {

    printf("usage: checksum filename\n");

    exit(1);

}

if ( (Handle = open(argv[1], O_RDONLY | O_BINARY)) == -1 ) {

    printf("Can't open file: %s\n", argv[1]);

    exit(1);

/* Initialize the checksum accumulator */

Checksum = 0;

/* Add each byte in turn into the
checksum accumulator */ while ( (ReadLength = read(Handle, &Byte, sizeof(Byte))) > 0 ) {

    Checksum += (unsigned int) Byte;

    if ( ReadLength == -1 ) {

        printf("Error reading file %s\n", argv[1]);

        exit(1);

    }

/* Report the
result */ printf("The checksum is: %u\n", Checksum);

exit(0);
```

<В таблице 1.1 показано время, затраченное в Листинге 1.1 на формирование контрольной суммы для Файл тезауруса WordPerfect версии 4.2, TH.WP (размером 362 293 байта), на АТ с частотой 10 МГц компьютер без специального происхождения. Время выполнения указано для листинга 1.1, скомпилированного с помощью Компиляторы Borland и Microsoft с включенной и выключенной оптимизацией; все четыре времени являются однако практически то же самое, и все они работают слишком медленно, чтобы быть приемлемыми. Листинг 1.1 требуется более двух с половиной минут для определения контрольной суммы в одном файле!

 Списки 1.2 и 1.3 образуют C/assembly, эквивалентный листингу 1.1, и Списки 1.6 и 1.7 образуют C/assembly, эквивалентный листингу 1.5.

Эти результаты ясно показывают, что глупо полагаться на оптимизацию вашего компилятора для повышения быстродействия ваши программы быстры. Листинг 1.1 просто плохо разработан, и никакого количества компиляторов оптимизация компенсирует этот сбой. Чтобы донести суть, рассмотрим списки 1.2 и 1.3, которые вместе эквивалентны листингу 1.1, за исключением того, что весь цикл контрольных сумм написан в сжатом ассемблерном коде. Реализация на языке ассемблера действительно быстрее чем любая из версий C, как показано в таблице 1.1, но она менее чем на 10 процентов быстрее, и это по-прежнему неприемлемо медленно.

Таблица 1.1 Время выполнения контрольной суммы WordPerfect.

Список коэффицентов оптимизации сборки Borland Microsoft для Borland Microsoft	Оптимизация Соотношение				
	(нет возможности)	(нет возможности)	(возможность)	(возможность)	
1	166.9	166.8	167.0	165.8	155.11.08
4	13.5	13.6	13.5	13.5	1.01
5	4.7	5.5	3.8	3.4	2.7 2.04
Соотношение наилучшего дизайна к наихудшему дизайну	35.51	30.33	43.95	48.76	57.44

Примечание: Время выполнения (в секундах) для списков этой главы было рассчитано по времени, когда скомпилированные списки были запущены в файле тезауруса WordPerfect 4.2 TH.WP (362 293 байта по размеру), как скомпилировано в small model с помощью компиляторов Borland и Microsoft с оптимизация включена (опция) и выключена (опция отсутствует). Все времена измерялись с помощью Paradigm Systems' Программа с ТАЙМЕРОМ на частоте 10 МГц в состоянии ожидания 1 при клонировании с жестким диском продолжительностью 28 мс, с диском кэширование отключено.

ЛИСТИНГ 1.2 L1-2.C

```

>* Program to calculate the 16-bit checksum of the stream of bytes
* from the specified file. Obtains the bytes one at a time in
* assembler, via direct calls to DOS.
*/
#include <stdio.h>
#include <fcntl.h>
main(int argc, char *argv[]) {
    int Handle;
    unsigned char Byte;
    unsigned int Checksum;
    int ReadLength;
    if ( argc!= 2 ) {
        printf("usage: checksum filename\n");
        exit(1);
        if ( (Handle = open(argv[1], O_RDONLY | O_BINARY)) == -1 ) {
            printf("Can't open file: %s\n", argv[1]);
            exit(1);
        }
        if ( !ChecksumFile(Handle, &Checksum) ) {
            printf("Error reading file %s\n", argv[1]);
            exit(1);
        }
        /* Report the result */
        printf("The checksum is: %u\n", Checksum);
        exit(0);
    }
}

```

LISTING 1.3 L1-3.ASM

```

; Assembler subroutine to perform a 16-bit checksum on the file
; opened on the passed-in handle. Stores the result in the
; passed-in checksum variable. Returns 1 for success, 0 for error.
;
; Call as:
;
;     int ChecksumFile(unsigned int Handle, unsigned int *Checksum);
;
; where:
; iHandle = handle # under which file to checksum is open
; Checksum = pointer to unsigned int variable checksum is
; to be stored in
; Parameter structure:
;
;
; Params
;
;     dw _?;pushed BP
;     dw _?;return address
;
; Handledw _?
; Checksumdw _?
; Paramsends
;
;     .model small
;     .data
;
; TempWord labelword
; TempBytedb...?;each byte read by DOS will be stored here
;     db...0;high byte of TempWord is always 0
;           ;for 16-bit adds
;
;     .code
;     public _ChecksumFile
;
; _ChecksumFileproc near
;     pushbp.
;     movbp,sp
;     pushsi _;save C's register variable
;     movbx, [bp+Handle];get file handle
;     subsi,si;zero the checksum ;accumulator
;     movex, 1;request one byte on each ;read
;     movdx,offset TempByte ;point DX to the byte in
;           ;which DOS should store
;           ;each byte read
;
; <>ChecksumLoop:
;     movah,3fh;DOS read file function #
;     int21h;read the byte
;     jc...ErrorEnd;an error occurred
;     andax,ax;any bytes read?
;     jz _Success;no-end of file reached-we're done
;     addsi, [TempWord];add the byte into the
;           ;checksum total
;     jmpChecksumLoop
;
; ErrorEnd:
;     subax, ax;error
;     jmpshort Done
;
; Success:
;     movbx, [bp+Checksum] ;point to the checksum variable

```

<Выполнено :	mov mov	[bx],si; ax,1 ;	сохраните новую контрольную сумму успех
_ файл контрольной суммы	pop pop ret endp end	si bp. —	; восстановить регистровую переменную С

Урок ясен: оптимизация делает код быстрее, но без надлежащего дизайна оптимизация просто создает быстрый медленный код.

Хорошо, тогда как мы собираемся улучшить наш дизайн? Прежде чем мы сможем это сделать, мы должны понять, что не так с текущим дизайном.

Знать территорию.

Почему листинг 1.1 такой медленный? Одним словом: накладные расходы. Библиотека С реализует функция read(), вызывающая DOS для чтения нужного количества байтов. (Я понял это наблюдая за выполнением кода с помощью отладчика, но вы можете купить исходный код библиотеки как от Microsoft, так и от Borland.) Это означает, что листинг 1.1 (а также листинг 1.3) выполняет одну функцию DOS на каждый обработанный байт — и функции DOS, особенно эту во-первых, это сопряжено с большими накладными расходами.

Начнем с того, что функции DOS вызываются с прерываниями, а прерывания являются одними из самых медленных инструкций процессоров семейства x86. Затем DOS должна выполнить внутреннюю настройку и перейти к желаемой функции, затратив при этом больше циклов. Наконец, у DOS есть чтобы выполнить поиск в его собственных буферах и узнать, был ли уже прочитан нужный байт, прочитайте его с диска, если нет, сохраните байт в указанном месте и вернитесь. Все это занимает много времени намного, намного больше, чем остальная часть основного цикла в листинге 1.1. Короче говоря, листинг 1.1 тратит практически все свое время на выполнение функции read(), и большая часть этого времени тратится где-то в DOS.

Вы можете убедиться в этом сами, просмотрев код с помощью отладчика или используя код профилировщик, но поверьте мне на слово: вызовы DOS сопряжены с большими накладными расходами, и именно это лишает жизни листинг 1.1.

Как мы можем ускорить работу с листингом 1.1? Должно быть ясно, что мы должны каким-то образом избегать вызова DOS для каждого байта в файле, а это означает чтение более одного байта за раз время, затем буферизует данные и выделяет их для проверки по одному байту за раз. Автор: боже, это описание функции потокового ввода-вывода С, посредством которой С считывает файлы частями и буферизует байты внутри, передавая их приложению по мере необходимости путем чтения из памяти, а не вызывая DOS. Давайте попробуем использовать потоковый ввод-вывод и посмотрим, что случается.

Листинг 1.4 похож на листинг 1.1, но использует fopen() и getc() (вместо open() и прочитайте()), чтобы получить доступ к файлу с контрольной суммой. Результаты подтверждают наши теории великолепно и подтверждают наш новый дизайн. Как показано в таблице 1.1, листинг 1.4 выполняется быстрее более чем на порядок быстрее, чем даже версия для сборки из листинга 1.1, даже *хотя листинг 1.1 и листинг 1.4 выглядят почти одинаково. Стороннему наблюдателю read() и getc() показались бы немного разными, но в значительной степени взаимозаменяемыми, и все же в в этом приложении разница в производительности между ними примерно такая же, как у между ПК с частотой 4,77 МГц и 16 МГц 386.*



Убедитесь, что вы понимаете, что на самом деле происходит, когда вы вставляете казалось бы, безобидный вызов функции в критически важные по времени части вашего код.

В данном случае это означает знание того, как DOS и библиотеки доступа к файлам С / С ++ выполняют свою работу. Другими словами, знайте территорию!

ЛИСТИНГ 1.4 L1-4.C

```
>* Program to calculate the 16-bit checksum of the stream of bytes
* from the specified file. Получает байты по одному с помощью
* getc(), позволяя С выполнять буферизацию данных.
*/

#включить <stdio.h>
main(int argc, char *argv[]) {
    ФАЙЛ *CheckFile;

    байт int;

    контрольная сумма int без знака;

    if ( argc!= 2 ) {

        printf("использование: имя файла контрольной суммы\n");

        выход(1);

        if ( (CheckFile = fopen(argv[1], "rb")) == NULL ) {

            printf("Не удается открыть файл: %s\n", argv[1]);

            выход(1);

            /* Инициализируйте накопитель контрольной суммы */

            Контрольная сумма = 0;

            /* Добавляйте каждый байт по очереди в накопитель контрольной суммы */

            while ( (Байт = getc(контрольный файл)) != EOF) {

                Контрольная сумма += (беззнаковый int) байт;

            /* Сообщить результат */

            printf("Контрольная сумма равна: %u\n", контрольная
сумма);

            выход(0);
```

<Знайте, когда это важно

В последнем разделе содержится особенно интересная фраза: критичные по времени части вашего кода. Критичные по времени участки вашего кода - это те участки, в которых скорость код оказывает существенное влияние на общую производительность вашей программы— и под “значительным” я не имею в виду, что это делает код на 100 процентов быстрее, или на 200 процентов, или вообще на какую-то конкретную величину, а скорее то, что это делает программу более *отзывчивый и / или полезный с точки зрения пользователя*.

Не тратьте время на оптимизацию кода, не критичного ко времени: кода настройки, кода инициализации и тому подобное. Потратьте свое время на повышение производительности кода внутри часто используемых циклов и в тех частях ваших программ, которые напрямую влияют на время отклика. Обратите внимание, для пример, что я не потрудился реализовать версию программы контрольной суммы полностью в ассемблере; Списки 1.2 и 1.6 вызывают подпрограммы ассемблера, которые обрабатывают операции, критичные по времени, но С по-прежнему используется для проверки параметров командной строки, открытие файлов, печать и тому подобное.



Если бы вы полностью реализовали любой из списков в этой главе сборка, оптимизированная вручную, я полагаю, вы могли бы добиться повышения производительности улучшение на несколько процентов — но я сомневаюсь, что вы добьетесь даже этого многого, и вы наверняка потратите чертовски много времени на любой скудный результат улучшение действительно произойдет. Позвольте С делать то, что у него получается хорошо, и используйте assembly только тогда, когда это дает ощутимую разницу.

Кроме того, мы не хотим проводить оптимизацию до тех пор, пока дизайн не будет доработан к нашему удовлетворению, а этого не произойдет, пока мы не подумаем о других подходах.

Всегда рассматривайте альтернативы

Листинг 1.4 хорош, но давайте посмотрим, есть ли другие — возможно, менее очевидные — способы получения тех же результатов быстрее. Давайте начнем с рассмотрения того, почему листинг 1.4 намного лучше, чем Листинг 1.1. Как и read(), getc() вызывает DOS для чтения из файла; повышение скорости расположение листинга 1.4 поверх листинга 1.1 происходит потому, что getc() считывает много байтов одновременно через

DOS, затем управляет этими байтами для нас. Это быстрее, чем считывать их по одному за раз использование read() — но нет причин думать, что это быстрее, чем использование нашей программы чтение блоков и управление ими само по себе. Проще, да, но не быстрее.

Учтите это: Каждый вызов getc() включает в себя ввод параметра, выполнение вызова функции библиотеки C, получение параметра (в коде библиотеки C), поиск информации о желаемом потоке, разбуферирование следующего байта из потока и возврат к вызывающему коду. Это занимает значительное количество времени, особенно при отличие от простого сохранения указателя на буфер и прокрутки данных в нем буфер внутри одного цикла.

Многие программисты назвали бы четыре причины, по которым они не пытаются улучшить Листинг 1.4:

1. Код и так достаточно быстр.
2. Код работает, и некоторые люди довольны кодом, который работает, даже если он достаточно медленный, чтобы раздражать.
3. Библиотека C написана на оптимизированной сборке и, вероятно, будет быстрее любого кода, который мог бы написать средний программист для выполнения, по сути, той же самой функции.
4. Библиотека C удобно обрабатывает буферизацию файловых данных, и было бы неудобно реализовывать эту возможность.

Я проигнорирую первую причину, как потому, что производительность больше не является проблемой, если код достаточно быстрый, так и потому, что текущее приложение работает недостаточно быстро — 13 секунд это долго. (Остановитесь и подождите 13 секунд, пока вы делаете что-то интенсивное, и вы увидите, насколько это долго.)

Вторая причина является отличительной чертой посредственного программиста. Знайте, когда оптимизация важна — и затем оптимизируйте, когда это важно!

Третья причина часто ошибочна. Библиотечные функции C не всегда написаны на ассемблере и не всегда особенно хорошо оптимизированы. (На самом деле, они часто пишутся для мобильности, которая не имеет ничего общего с оптимизацией.) Более того, они носят общий характер- целевые функции, и часто их может превзойти хорошо, но не блестяще написанный код который хорошо подходит для конкретной задачи. В качестве примера рассмотрим листинг 1.5, в котором используется внутренняя буферизация для одновременной обработки блоков байтов. Из таблицы 1.1 видно, что листинг 1.5 является в 2,5-4 раза быстрее, чем в листинге 1.4 (и в целых 49 раз быстрее, чем в листинге 1.1!), хотя в нем вообще не используется сборка.



Очевидно, что вы можете добиться успеха, используя специальный код C вместо C библиотечная функция — если у вас есть полное представление о том, как C библиотечная функция работает именно так, как нужно вашему приложению. В противном случае вы в конечном итоге перепишите библиотечные функции C на C, что не имеет никакого смысла вообще.

ЛИСТИНГ 1.5 L1-5.C

```
>* Program to calculate the 16-bit checksum
of the stream of bytes * from the specified
file. Buffers the bytes internally, rather
* than letting C or DOS do the work. */
#include <stdio.h> #include <fcntl.h>
#include <alloc.h> /* alloc.h
for Borland,
                                malloc.h for Microsoft */ #define
BUFFER_SIZE 0x8000 /* 32Kb data buffer */
main(int argc, char *argv[]) {
```

```
<>int Handle;

unsigned int Checksum; unsigned

char *WorkingBuffer, *WorkingPtr;

int WorkingLength, LengthCount;

if ( argc!= 2 ) {

    printf("использование: контрольная сумма имени файла\n");

    exit(1);

    if ( (Handle = открыть(argv[1], O_RDONLY | O_BINARY)) == -1) {

        printf("Не удается открыть файл: %s\n", argv[1]);

        выход(1);

        /* Получить память для буферизации данных */

        if ( (WorkingBuffer = malloc(BUFFER_SIZE)) == NULL ) {

            printf("Не удается получить достаточно памяти");

            выход(1);

            /* Инициализировать накопитель контрольной

суммы */ Контрольная сумма = 0;

            /* Обработайте файл порциями

размером с буфер */ выполните {

                if ( (Рабочая длина = чтение(Дескриптор, Рабочий буфер,

                    РАЗМЕР БУФЕРА)) == -1 ) {

                    printf("Ошибка чтения файла %s\n", argv[1]);

                    выход(1);

                    /* Контрольная сумма этого фрагмента */

                    WorkingPtr = WorkingBuffer;

                    LengthCount = WorkingLength;

                    while ( LengthCount-- ) {

                        /* Добавляйте каждый байт по очереди в накопитель контрольной суммы */

                        Контрольная сумма += (unsigned int) *WorkingPtr++;

                    } while ( рабочая длина);

                    /* Сообщить результат */

                    printf("Контрольная сумма равна: %u\n", контрольная

сумма); выход(0);
```

<Это подводит нас к четвертой причине: избегать реализации с внутренней буферизацией, подобной Листинг 1.5 из-за сложности кодирования такого подхода. Правда, проще позволить Библиотечной функции C выполнять всю работу, но выполнить внутреннюю буферизацию не так уж и сложно. ключевым является концепция обработки данных в перезапускаемых блоках; то есть считывание фрагмента данных, работа с данными до тех пор, пока они не закончатся, приостановка операции, пока не поступит больше данных прочитайте, а затем продолжайте, как будто ничего не произошло.

В листинге 1.5 реализация перезапускаемого блока довольно проста, потому что контрольная сумма работает с одним байтом за раз, забывая о каждом байте сразу же после добавления его к сумме. Листинг 1.5 считывает блок байтов из файла, проверяет суммы байты в блоке и получает другой блок, повторяя процесс до тех пор, пока не будет обработан весь файл . В главе 5 мы увидим более сложный перезапускаемый блок реализация, включающая поиск текстовых строк.

В любом случае, листинг 1.5 не намного сложнее листинга 1.4 — и он намного быстрее. Всегда рассматривайте альтернативные варианты; немного умного мышления и редизайн программы может иметь большое значение.

Знайте, как включить сок.

Я снова и снова говорил, что оптимизация бессмысленна, пока дизайн не улажен. Когда однако придет время, и оптимизация действительно может существенно изменить ситуацию. Таблица 1.1 показывает, что оптимизированная версия листинга 1.5, созданная Microsoft C превосходит неоптимизированную версию того же кода более чем на 60 процентов. Что более того, версия листинга 1.5, выполненная в основном на ассемблере, показанная в листингах 1.6 и 1.7, превосходит даже наиболее оптимизированную версию листинга 1.5 на C на 26 процентов. Это значительные улучшения, которые стоит продолжить — после того, как дизайн будет доведен до максимума.

ЛИСТИНГ 1.6 L1-6.C

```

>* Program to calculate the 16-bit checksum of
the stream of bytes * from the specified
file. Buffers the bytes internally, скорее
* чем позволить C или DOS выполнять всю работу
с критичной по времени * частью кода, написанной
на оптимизированном ассемблере.
*/ #включить <stdio.h> #включить <fcntl.h>
#включить <alloc.h> /* alloc.h для Borland,
                        malloc.h для Microsoft */
#define BUFFER_SIZE 0x8000 /* буфер данных
32 КБ */ main(int argc, char *argv[]) {

    <>int Handle;
    unsigned int Checksum;
    unsigned char *WorkingBuffer;
    int WorkingLength;
    if ( argc!= 2 ) {
        printf("usage: checksum filename\n");
        exit(1);
    }
    if ( (Handle = open(argv[1], O_RDONLY | O_BINARY)) == -1 ) {
        printf("Can't open file: %s\n", argv[1]);
        exit(1);
    }
    /* Get memory in which to buffer the data */
    if ( (WorkingBuffer = malloc(BUFFER_SIZE)) == NULL ) {
        printf("Can't get enough memory\n");
        exit(1);
    }
    /* Initialize the checksum accumulator */
    Checksum = 0;
    /* Process the file in 32K chunks */
    do {
        if ( (WorkingLength = read(Handle, WorkingBuffer,
BUFFER_SIZE)) == -1 ) {
            printf("Error reading file %s\n", argv[1]);
            exit(1);
        }
        /* Checksum this chunk if there's anything in it */
        if ( WorkingLength )
            ChecksumChunk(WorkingBuffer, WorkingLength, &Checksum);
    } while ( WorkingLength );
    /* Report the result */
    printf("The checksum is: %u\n", Checksum);
    exit(0);
}

```

LISTING 1.7 L1-7.ASM

```

; Assembler subroutine to perform a 16-bit checksum on a block of
; bytes 1 to 64K in size. Adds checksum for block into passed-in
; checksum.
;
; Call as:
;     void ChecksumChunk(unsigned char *Buffer,
;     unsigned int BufferLength, unsigned int *Checksum);
;
; where:
; Buffer = pointer to start of block of bytes to checksum
; BufferLength = # of bytes to checksum (0 means 64K, not 0)
; Checksum = pointer to unsigned int variable checksum is
; stored in
;
; Parameter structure:
;
Parms struc
    dw _?;pushed BP
    ?;return address

Buffer dw _?
BufferLength dw _?
Checksum dw _?
Parms ends

.model small
.code
public _ChecksumChunk
_ChecksumChunk proc near
    push bp
    mov bp,sp
    push si;save C's register variable
    cld;make LODSB increment SI
    mov si,[bp+Buffer];point to buffer

```

```

<>mov cx, [bp+BufferLength];get buffer length
mov bx, [bp+Checksum];point to checksum variable
mov dx, [bx];get the current checksum
sub ah,ah;so AX will be a 16-bit value after LODSB
ChecksumLoop:
lodsb;get the next byte
add dx,ax;add it into the checksum total
loop ChecksumLoop;continue for all bytes in block
mov [bx],dx;save the new checksum
pop si;restore C's register variable
pop bp
ret
_ChecksumChunk endp
end

```

<Обратите внимание, что в таблице 1.1, оптимизация практически ничего не меняет, за исключением случая с листингом 1.5, где дизайн был значительно доработан. Время выполнения в других случаях преобладает время, проведенное в DOS и / или библиотеке C, поэтому оптимизация кода, который вы пишете, в значительной степени не имеет значения. Более того, в то время как примерно двукратный улучшения у нас есть путем оптимизации не следует пренебрегать, она меркнет на фоне до-50- раз улучшения мы получили путем реорганизации.

Кстати, время выполнения даже списков 1.6 и 1.7 зависит от диска DOS время доступа. Если дисковый кэш включен и файл, подлежащий проверке, уже находится в кэше, версия assembly работает в три раза быстрее версии C. Другими словами, неотъемлемая природа этого приложения ограничивает повышение производительности, которое может быть достигнуто с помощью сборки. В приложениях с большей нагрузкой на процессор и меньшей привязкой к диску, особенно в тех приложениях, в которых могут использоваться строковые инструкции и / или развернутые циклы эффективно используемая сборка, как правило, выполняется значительно быстрее по сравнению с C, чем в этом очень специфический случай.



Не закидывайтесь на оптимизации компиляторов или языка ассемблера — лучший оптимизатор находится у вас между ушей.

По сути, все это способ сказать: знай, куда идешь, знай территорию и знай, когда это важно.

Где мы были, что мы видели

Чему мы научились? Не позволяйте чужому коду — даже DOS — выполнять работу за вас когда важна скорость, по крайней мере, не зная, что делает этот код и насколько хорошо он работает .

Оптимизация имеет значение только после того, как вы выполнили свою часть работы по разработке программы. Рассмотрим соотношения на вертикальной оси таблицы 1.1, которые показывают, что оптимизация является практически полностью потрачено впустую в приложении для подсчета контрольных сумм без эффективного дизайна. Оптимизация - не панацея. В таблице 1.1 показано двукратное улучшение по сравнению с оптимизацией — и улучшение в 50 с лишним раз по сравнению с редизайном. Давние споры о том, какой компилятор C лучше всего оптимизирует код, не имеют такого большого значения в свете таблицы 1.1, правда? Ваш обычный оптимизатор имеет гораздо большее значение, чем оптимизатор вашего компилятора, и всегда есть ассемблер для тех, как правило, небольших участков кода, где производительность действительно имеет значение.

К чему мы идем

В этой главе представлен краткий пошаговый обзор процесса проектирования. Я не утверждаю, что это единственный способ создать высокопроизводительный код; это просто подход, который работает для меня. Создавайте код, как хотите, но никогда не забывайте, что дизайн имеет значение более чем детальная оптимизация. Никогда не прекращайте искать оригинальные способы повышения производительности — и никогда не тратьте время на ускорение кода, который не нуждается в ускорении.

С этого момента я собираюсь сосредоточиться на конкретных способах создания высокопроизводительного кода. В

Главе 5 мы продолжим рассматривать перезапускаемые блоки и внутреннюю буферизацию в форме о программе, которая ищет в файлах текстовые строки.

Глава 2 . "Другой мир"

Уникальная природа оптимизации языка ассемблера

Как я показал в предыдущей главе, оптимизация - это далеко не всегда вопрос “перехода к ассемблеру”. Фактически, при настройке производительности высокоуровневого языкового кода, ассемблер следует использовать редко, и то только после того, как вы убедитесь в неправильном выборе или неуклюже реализованный алгоритм не съедает вас заживо. Конечно, если вы используете ассемблер в вообще, убедитесь, что используете его правильно. Потенциал ассемблерного кода в том, что он выполняется медленно это плохо понято многими людьми, но этот потенциал велик, особенно в руках невежд.

Однако по-настоящему эффективная оптимизация происходит только на уровне сборки, и она происходит в ответ на набор динамических параметров, полностью отличных от тех, которые определяют C / C ++ или Оптимизация на Паскале. Я буду говорить об оптимизации на уровне ассемблера снова и снова в этой книге, но когда я это сделаю, я думаю, будет полезно, если вы разберетесь в этих сборках специфическая динамика.

Как обычно, лучший способ проникнуть в суть дела - это привести реальный пример.

Инструкции: Индивидуальное против коллективного

Некоторое время назад меня попросили поработать над важной подпрограммой сборки, чтобы заставить она выполнялась как можно быстрее. Задача подпрограммы состояла в том, чтобы построить nibble из четырех битов, считанных из разных байтов, вращая и комбинируя биты так, чтобы они в конечном итоге в итоге все было аккуратно выровнено по битам 3-0 одного байта. (На случай, если вам интересно, цель заключалась в создании 16-цветного пикселя из битов, разбросанных по 4 байтам.) Я исследовал подпрограмму построчно, сохраняя цикл здесь и цикл там, пока код не стал действительно оптимизирован. Когда я закончил, ключевая часть кода выглядела примерно вот так:

```
>LoopTop:
    lodsb;get the next byte to extract a bit from
    andal,ah;isolate the bit we want
    rolal,cl;rotate the bit into the desired position
    or bl,al;insert the bit into the final nibble
    dec cx;the next bit goes 1 place to the right
    dec dx;count down the number of bits
    jnzLoopTop ;process the next bit, if any
```

<Сейчас сложно написать код, который был бы намного быстрее семи инструкций, из которых только одна обращается к памяти, и большинство программистов на этом бы закончили. Тем не менее, кое-что меня беспокоило, поэтому я потратил немного времени на повторный просмотр кода. Внезапно, ответ поразил меня — код вставлял каждый бит на место отдельно, так что многобитовая ротация выполнялась каждый раз в цикле, в общей сложности четыре отдельные многобитовые ротации, отнимающие много времени!



В то время как сами инструкции были индивидуально оптимизированы, общий подход не позволил использовать инструкции наилучшим образом.

Я изменил код на следующий:

```
>LoopTop:
    lodsb;get the next byte to extract a bit from
    andal,ah;isolate the bit we want
    or _bl,al;insert the bit into the final nibble
    rolbl,1;освободите место для следующего бита
```

```
<>dec dx ;count down the number of bits
jnz LoopTop;process the next bit, if any
rol bl,cl;rotate all four bits into
their final
; positions at the same time
```

<Это вывело дорогостоящее многобитовое вращение из цикла, так что оно было выполнено только один раз, а не четыре раза. Хотя код может внешне не сильно отличаться от оригинала и фактически по-прежнему содержит точно такое же количество инструкций, производительность только одно это изменение улучшило всю подпрограмму примерно на 10 процентов. (Кстати, на этом оптимизация не закончилась; я устранил инструкции DEC и JNZ с помощью расширяем четыре итерации цикла — но это история для другой главы.)

Суть в следующем: чтобы писать действительно превосходные программы на ассемблере, вам нужно знать, что такое что делают различные инструкции, и какие инструкции выполняются быстрее всего ... и многое другое. Вы также должны научиться смотреть на свои проблемы программирования с разных точек зрения, чтобы вы я могу заставить эти быстрые инструкции работать наиболее эффективными способами.

Сборка принципиально отличается.

Неужели так сложно написать хороший ассемблерный код для ПК? Да! Благодаря явно причудливый характер процессоров семейства x86, язык ассемблера отличается принципиально от других языков, и с ним, несомненно, сложнее работать. На с другой стороны, потенциал ассемблерного кода намного больше, чем у других языков, также.

Чтобы понять, почему это так, рассмотрим, как пишется программа. Программист изучает требования приложения, разрабатывает решение на некотором уровне абстракции, а затем воплощает этот дизайн в кодовой реализации. Если нет при правильном обращении преобразование, происходящее между концепцией и реализацией, может значительно снизить производительность; например, программист, который реализует процедуру поиска по списку из 100 000 отсортированных элементов с помощью линейного метода, а не бинарный поиск в конечном итоге приведет к разочаровывающе медленной программе.

Неэффективность преобразования

Независимо от того, насколько хорошая реализация получена из соответствующего дизайна, однако языки высокого уровня, такие как C / C ++ и Pascal, неизбежно вводят дополнительные неэффективность преобразования, как показано на рисунке 2.1.

Процесс превращения проекта в исполняемый код с помощью языка высокого уровня включает в себя два преобразования: одно выполняется программистом для генерации исходного кода, а другое выполняется компилятором для преобразования исходного кода в машинные языковые инструкции. Следовательно, код машинного языка, генерируемый компиляторы обычно не являются оптимальными, учитывая требования исходного дизайна.

Языки высокого уровня предоставляют искусственные среды, которые относительно хорошо поддаются обработке к человеческим навыкам программирования, чтобы облегчить переход от проектирования к реализации. Ценой такой простоты реализации является значительная потеря эффективности преобразования исходного кода в машинный язык. Это особенно верно учитывая, что семейство x86 в реальном и 16-разрядном защищенном режиме, с его специализированной памятью- инструкции адресации и архитектура сегментированной памяти сами по себе не подходят особенно хорошо подходят для разработки компилятора. Даже 32-разрядный режим 386 и его преемников, благодаря более мощным режимам адресации они предлагают меньше регистров, чем это сделали бы компиляторы Нравится.

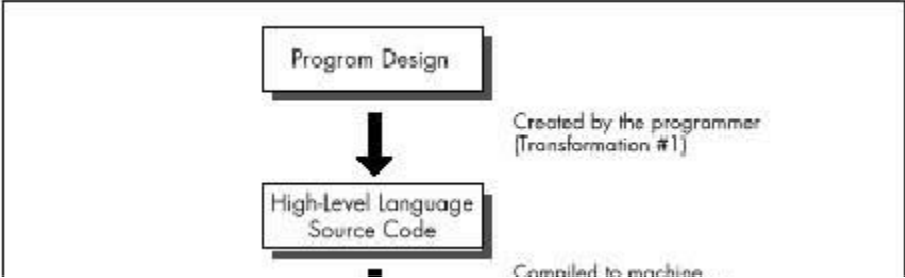




Рисунок 2.1 Неэффективность преобразования языка высокого уровня.

Сборка, с другой стороны, - это просто ориентированное на человека представление машины язык. В результате ассемблер предоставляет сложную среду программирования — голый аппаратное и системное программное обеспечение компьютера —но правильно сконструированная сборка *программы не подвергаются потерям при преобразовании, как показано на рисунке 2.2.*

При создании программы на ассемблере требуется только одно преобразование, и это одиночное преобразование полностью находится под контролем программиста. Ассемблеры не выполняют никакого преобразования исходного кода в машинный язык; вместо этого они просто отображают инструкции ассемблера к инструкциям машинного языка на индивидуальной основе. В результате программист может создавать код машинного языка, который точно соответствует адаптированный к потребностям каждой задачи, требуемой данным приложением.

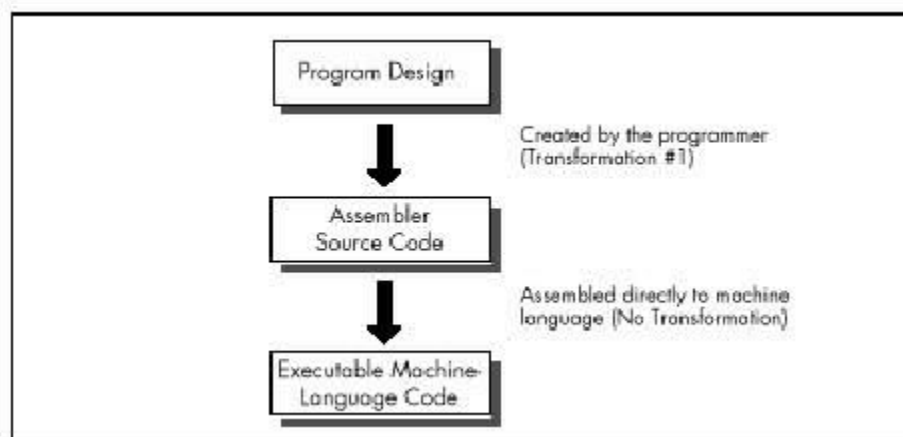


Рисунок 2.2 Правильно построенные программы сборки не страдают от
потерь при преобразовании.

Ключевым фактором, конечно, является программист, поскольку при сборке программист должен по сути, выполнить преобразование из спецификации приложения в машинный язык полностью самостоятельно. (Ассемблер просто выполняет прямой перевод с ассемблера на машинный язык.)

Уверенность в себе

Таким образом, первая часть оптимизации языка ассемблера - это самооптимизация. Ассемблер - это ничто больше, чем инструмент, позволяющий создавать программы на машинном языке без необходимости мыслить в шестнадцатеричных кодах. Таким образом, программисты на языке ассемблера — в отличие от всех других программистов — должны нести полную ответственность за качество своего кода. Поскольку ассемблеры обеспечивают небольшая помощь на любом уровне выше, чем поколение машинного языка, ассемблер программист должен быть способен как непосредственно кодировать любую программную конструкцию, так и управление ПК на самом низком практическом уровне — операционной системой, BIOS и даже оборудованием, где это необходимо. Языки высокого уровня справляются с большей частью этого прозрачно для программиста, но в ассемблере все честно и необходимо, что подводит нас к другому аспекту оптимизации ассемблера: знаниям.

Знания

В мире ПК вам никогда не хватит знаний, и каждый товар, который вы добавляете в ваш магазин сделает ваши программы лучше. Доскональное знакомство с обеими операционными системами системные API и интерфейсы BIOS важны; поскольку эти интерфейсы хорошо документированы и достаточно просты, мой совет - взять одну-две хорошие книги и ввести себя в курс дела. Аналогичным образом, требуется знакомство с аппаратным обеспечением ПК. Хотя в этой теме рассматривается множество наземных адаптеров дисплея, клавиатур, последовательных портов, порты принтера, каналы таймера и DMA, организация памяти и многое другое — большая часть аппаратное обеспечение хорошо документировано, а статьи о программировании основного оборудования компоненты часто упоминаются в литературе, поэтому такого рода знания могут быть приобретены достаточно легко.

Единственный наиболее важный аспект аппаратного обеспечения, о котором труднее всего узнать учиться - это центральный процессор. Процессоры семейства x86 имеют сложный, нерегулярный набор команд, и, в отличие от большинства процессоров, они не являются ни простым, ни хорошо документированным истинным кодом Производительность. Более того, сборка настолько трудно усвоить, что большинство статей и книг что представить ассамблее код довольствоваться код, который просто работает, а не код, который толкает процессор на пределе своих возможностей. Фактически, поскольку большинство статей и книг написано для неопытных программистов на ассемблере, доступно очень мало какой-либо информации о том, как генерировать высококачественный ассемблерный код для процессоров семейства x86. В результате знания об их эффективном программировании, безусловно, собрать сложнее всего. Хороший часть этой книги посвящена поиску таких знаний.



Однако имейте в виду: независимо от того, сколько вы узнаете о программировании ПК на ассемблере, всегда есть что узнать еще.

Гибкий ум

Нескончаемый сбор информации - это все, что нужно для оптимизации сборки, значит? Вряд ли. Знания - это просто необходимая база, на которой можно строить. Давайте воспользуемся моментом, чтобы изучить цели хорошего программирования на ассемблере, и остальная часть факторы, влияющие на оптимизацию ассемблера, встанут на свои места.

В принципе, для высокопроизводительной сборки возможны только две цели программирование: учитывая требования приложения, сведите к минимуму либо количество циклов процессора, необходимых программе для запуска, или количество байтов в программе, или некоторая комбинация того и другого. Мы рассмотрим способы достижения обеих целей, но мы чаще будем заботиться о сохранении циклов, чем байтов, поскольку ПК обычно предлагает относительно больше памяти, чем вычислительных мощностей. Фактически, мы обнаружим, что производительность в два-три раза выше по сравнению с и без того плотной сборкой код часто возможен, если мы готовы потратить дополнительные байты для экономии циклов. Не всегда желательно использовать такие методы для ускорения кода из-за больших требований к памяти, но это почти всегда возможно.

Вы заметите, что мой краткий список целей для высокопроизводительной сборки программирование не включает традиционные цели, такие как простота обслуживания и скорость разработки. Это действительно важные соображения — для отдельных лиц и компаний, которые разрабатывают и распространяют программное обеспечение. Люди, которые действительно покупают программное обеспечение, на с другой стороны, заботятся только о том, насколько хорошо работает это программное обеспечение, а не о том, как оно было разработано или как его поддерживают. В наши дни разработчики тратят так много времени сосредоточение внимания на таких общепризнанно важных вопросах, как ремонтпригодность кода и возможность повторного использования, контроль исходного кода, выбор среды разработки и тому подобное, что они часто делают забывают правило № 1: с точки зрения пользователя производительность имеет фундаментальное значение.



Комментируйте свой код, тщательно разрабатывайте его и пишите части, не требующие критики по времени если хотите, на языке высокого уровня, но когда вы пишете части, которые взаимодействуют с пользователем и / или влияют на время отклика, производительность должна быть вашей первостепенной целью, а сборка - путь к ней цель.

Знания, описанные ранее, абсолютно необходимы для выполнения любой из целей программирования на ассемблере. Чего эти знания сами по себе не делают, так это удовлетворения необходимость написания кода, который одновременно соответствует требованиям текущего приложения и работает максимально эффективно в среде ПК. Знания делают это возможно, но ваши инстинкты программиста заставляют это произойти. И это настолько интуитивно, на- быстрая интеграция спецификации программы и множества фактов о ПК, которая является сердцем оптимизации сборки Zen-класса.

Как и в случае с дзеном любого рода, овладение этим дзеном языка ассемблера - это скорее вопрос изучения, чем того, чтобы тебя учили. Тебе придется найти свой собственный путь обучения, хотя я эта книга поможет вам продвинуться на вашем пути. Приведенные мною неочевидные факты и примеры помогут вам приобрести необходимый опыт, но вы должны продолжить путешествие самостоятельно владейте. Каждая созданная вами программа расширит ваши горизонты программирования и увеличит доступные вам варианты решения следующей задачи. Способность вашего разума находить удивительные новые и более совершенные способы создания превосходного кода на основе концепции — гибкий ум, if you will— это основа хорошего кода на ассемблере, и вы разовьете этот навык, только выполняя.

Никогда не недооценивайте важность гибкого ума. Хороший ассемблерный код лучше, чем хорошо скомпилированный код. Многие люди хотели бы убедить вас в обратном, но они ошибаются. Это не означает, что языки высокого уровня бесполезны; отнюдь. Языки высокого уровня являются лучшим выбором для большинства программистов и для большей части код большинства приложений. Однако, когда требуется наилучший код — максимально быстрый или наименьший возможный код — единственным выходом является ассемблер.

Простая логика подсказывает, что ни один компилятор не может знать столько о том, что фрагмент кода должен выполнять или адаптироваться к этим потребностям, сколько человек, написавший код. Учитывая обладая такой превосходной информацией и адаптивностью, программист на языке ассемблера может генерировать код лучшего качества, чем компилятор, тем более что компиляторы ограничен ограничениями языков высокого уровня и процессом перехода от высокоуровневого к машинному языку. Следовательно, тщательно оптимизирован ассемблер - это не просто язык выбора, это единственный вариант для 1-10% процент кода, обычно состоящий из небольших, четко определенных подпрограмм, который определяет общая производительность программы, и это единственный выбор для кода, который должен быть как можно более компактным также, насколько это возможно. В обычных условиях, части ваших программ, не критичные ко времени, нет смысла тратить время и усилия на написание оптимизированного ассемблерного кода— вместо этого сосредоточьте свои усилия на циклах и тому подобном; но в тех областях, где вам требуется код высочайшего качества, не принимайте заменителей.

Обратите внимание, что я сказал, что программист на ассемблере может сгенерировать лучший код, чем компилятор, not сгенерирует лучший код. Хотя верно, что хороший ассемблерный код лучше, чем хорошо скомпилированный код, также верно, что плохой ассемблерный код часто намного хуже, чем плохо скомпилированный код; поскольку программист на ассемблере имеет такой большой контроль над программируя, он или она имеет практически неограниченные возможности тратить впустую циклы и байты. Меч режет в обе стороны, и хороший ассемблерный код требует большей, а не меньшей предусмотрительности и планирования, чем хороший код, написанный на языке высокого уровня.

Суть всего этого просто в том, что хорошее программирование на ассемблере выполняется в контексте надежная общая структура, уникальная для каждой программы, и гибкий подход - ключ к созданию этой структуры и поддержанию ее целостности.

С чего начать?

Подводя итог, навык оптимизации языка ассемблера - это сочетание знаний, перспективы и образа мышления, которые делают возможным создание абсолютно самого быстрого или самого маленького кода. Имея это в виду, каким должен быть первый шаг ? Развитие гибкого ума - очевидный шаг. Тем не менее, гибкий ум - это не лучше, чем имеющиеся в его распоряжении знания. Первый шаг на пути к овладению Таким образом, оптимизация на этом высоком уровне, по-видимому, заключается в том, чтобы научиться учиться.

Глава 3 – Ничего не предполагать.

Когда вы выходите за рамки написания оптимизированного кода для ПК, вы, вероятно, станете более чем немного навязчив в поиске подходов, которые позволят вам добиться большей скорости от вашего компьютера. В процессе вы неизбежно будете совершать ошибки, что нормально, поскольку при условии, что вы следите за этими ошибками и учитесь на них.

Показательный пример: несколько лет назад я наткнулся на статью о ассемблере 8088"" язык называется "Оптимизация для повышения скорости". "Оптимизировать" - это не то слово, которое следует употреблять легкомысленно; *Девятый новый университетский словарь Вебстера определяет "оптимизировать" как "сделать максимально совершенным*, эффективный или функциональный, насколько это возможно", что, безусловно, оставляет мало места для ошибок. автор, однако, выбрал небольшую, четко определенную процедуру языка ассемблера 8088 для уточнение, состоящее примерно из 30 инструкций, которые не делали ничего, кроме расширения с 8 бит до 16 битов путем дублирования каждого бита.

Автор "Оптимизации" явно тщательно доработал код, изучив альтернативные последовательности команд и суммируя циклы, пока не пришел к реализации, по его расчетам, была почти на 50 процентов быстрее, чем первоначальная процедура. В короче говоря, он использовал всю имеющуюся в его распоряжении информацию для улучшения своего кода, и в результате результат, экономия циклов на бушель. На самом деле было, только одна небольшая проблема с оптимизированная версия программы...

Она работала медленнее, чем исходная версия!

Издержки незнания

Каким бы прилежным ни был автор, он, тем не менее, совершил главный грех x86 программирование на ассемблере: он предположил, что доступная ему информация была правильной и полной. Хотя время выполнения, указанное Intel для своих процессоров, действительно правильное, оно неполное; другое, а часто и большее важно —частью производительности кода является время выборки инструкций, тема, к которой я вернусь в последующих главах.

Если бы автор потратил время на измерение реальной производительности своего кода, он бы не поставил под угрозу свою репутацию из-за относительно низкоэффективного кода. Более того, если бы он действительно измерил производительность своего кода и обнаружил, что он неожиданно медленный , любопытство вполне могло бы побудить его поэкспериментировать дальше и тем самым пополнить свой запас достоверной информации о процессоре.



Вот вам важный принцип оптимизации языка ассемблера: После создания наилучшего из возможных кодов проверьте его в действии, чтобы увидеть, действительно ли это делать то, что ты думаешь. Если он ведет себя не так, как ожидалось, это все к лучшему. хорошо, поскольку разгадывание тайн - это путь к знанию. Так вы узнаете больше, уверяю вас, чем из любого руководства или книги по языку ассемблера.

Ничего не предполагайте. Я не могу подчеркнуть это достаточно убедительно — когда вы заботитесь о производительность, сделайте все возможное, чтобы улучшить код, а затем измерьте улучшение. Если вы не измеряете производительность, вы просто предполагаете, а если вы предполагаете, то это не так весьма вероятно, что я напишу первоклассный код.

Незнание реальной производительности может дорого обойтись. Когда я зарабатывал на жизнь написанием видеоигр, я целыми днями пытался добиться большей производительности от своих графических драйверов. Я переписывал целые разделы кода, чтобы сэкономить несколько циклов, манипулировал регистрами и полагался сильно зависит от размытости -быстрых переходов от регистрации к регистрации. Когда я писал свою последнюю игру, Я обнаружил, что программа работала заметно быстрее, если я использовал справочные таблицы вместо сдвиги и добавления для своих вычислений. Согласно моему циклу, она не должна была работать быстрее подсчет, но это произошло. По правде говоря, выборка инструкций снова подняла голову, как это часто бывает

, и выборка сдвигов и добавлений занимала в четыре раза больше времени, чем номинальное время выполнения этих инструкций.

Незнание также может быть причиной значительной потери усилий. Я вспоминаю дискуссию в колонке писем одного компьютерного журнала о том, насколько быстро можно выводить текст на экране с цветным / графическим адаптером (CGA) без образования снега. Авторы письма подсчитывали каждый цикл в своих циклах синхронизации, точно так же, как автор в истории, с которой все началось глава имела. Как и этот автор, авторы писем не смогли включить очередь предварительной выборки в Учетная запись. Фактически, они также пренебрегли эффектами состояний ожидания видео, поэтому код, который они обсуждали, на самом деле был намного медленнее, чем их оценки. Надлежащий тест позволил бы, конечно, конечно, нужно было запустить код, чтобы увидеть, получился ли снег, поскольку единственным верным показателем производительности кода является наблюдение за ним в действии.

Таймер Zen

Очевидно, что одним из ключей к освоению оптимизации класса Zen является инструмент для измерения производительности кода. Наиболее точный способ измерения производительности - дорогостоящий аппаратные, но приемлемые измерения без каких-либо затрат могут быть выполнены с помощью ПК 8253 микросхема таймера, которая отсчитывает чуть более 1 000 000 раз в секунду. 8253 может быть запущен в начале интересующего блока кода и остановлен в конце этот код с результирующим количеством, показывающим, сколько времени потребовалось для выполнения кода с точностью около 1 микросекунды. (Микросекунда равна одной миллионной доли секунды и составляет сокращенно мкс). Если быть точным, 8253 отсчитывается один раз каждые 838,1 наносекунды. (А наносекунда равна одной миллиардной доли секунды и сокращается как ns.)

В листинге 3.1 показано программное обеспечение таймера на базе 8253, состоящее из трех подпрограмм: ZTimerOn, ZTimerOff и ZTimerReport. В оставшейся части этой книги я буду называть эти процедуры в совокупности “Дзен-таймер”. С-вызываемые версии двух прецизионных дзен-таймеров - это представлен в главе К на прилагаемом компакт-диске.

ЛИСТИНГ 3.1 PZTIMER.ASM

```
>; The precision Zen
timer (PZTIMER.ASM) ; Uses the 8253
timer to time the performance of
code that takes ; less than about 54
milliseconds to execute, with a
resolution ; of better than 10
microseconds. ; ; By Michael Abrash ; ; Externally
callable routines: ; ; ZTimerOn: Starts
the Zen timer, with interrupts
disabled. ; ZTimerOff: Stops the
Zen timer, saves the timer count,
times the overhead code, and restores interrupts to the
state they were in when ZTimerOn was called.
; ZTimerReport: Prints the net time that passed
between starting jand stopping the timer. ; Note: If
longer than about 54 ms passes between ZTimerOn and
ZTimerOff calls, the timer turns over and the count is
inaccurate. When this happens, an error message is displayed
instead of a count. The long-period Zen timer should be used
in such cases. ; Note: Interrupts *MUST* be
left off between calls to ZTimerOn
,and ZTimerOff for accurate timing and for
detection of jtimer overflow. ; Note: These routines
can introduce slight inaccuracies into the
system clock count for each code section timed even if
timer 0 doesn't overflow. If timer 0 does overflow, the
system clock can become slow by virtually any amount of
time, since the system clock can't advance while the
precison timer is timing. Consequently, it's a good idea
to reboot at the end of each timing session. (The
battery-backed clock, if any, is
not affected by the Zen
timer.)

<>; All registers, and all flags except the interrupt flag, are
; preserved by all routines. Interrupts are enabled and then disabled
```

```

>BASE_8253equ 40h
; The address of the timer 0
count registers in the 8253. ;
TIMER_0_8253 equ BASE_8253 + 0
;
; The address of the mode
register in the 8253.
MODE_8253equ BASE_8253 + 3 ;
; The address of Operation
Command Word 3 in the 8259 Programmable
; Interrupt Controller (PIC) (write
only, and writable only when
; bit 4 of the byte written to this
address is 0 and bit 3 is 1).
; OCW3equ 20h ; ; The address of the
Interrupt Request register in
the 8259 PIC ; (read only, and readable
only when bit 1 of OCW3 = 1
and bit 0 ; of OCW3 = 0).
; IRRequ 20h ; ; Macro to
emulate a POPF instruction in order
to fix the bug in some ; 80286 chips
which allows interrupts to occur
during a POPF even when ; interrupts
remain disabled. ; MPOPF macro
    local p1, p2
    jmp short p2 p1:iret; jump
to pushed address & pop flags p2:push
cs; construct far return address to
    call p1; the next instruction
endm
, ; Macro to delay briefly
to ensure that enough time has elapsed
; between successive I/O accesses
so that the device being accessed
; can respond to both accesses even
on a very fast PC. ; DELAY macro
    jmp $+2
    jmp $+2
    jmp $+2
endm
    OriginalFlags db ; storage for upper byte of
                    ; FLAGS register when
                    ; ZTimerOn called TimedCountdw
?; timer 0 count when the timer
                    ; is stopped ReferenceCount
dw?; number of counts required to
                    ; execute timer overhead code OverflowFlag
db?; used to indicate whether the
                    ; timer overflowed during the
                    ; timing interval
;
; String printed to
report results. ;
OutputStr label byte
    db...0dh, 0ah, 'Timed count: ', 5
dup (?) ASCIICountEnd label byte
    db ' microseconds', 0dh, 0ah
    db..."φ"
; String printed to report
timer overflow. ; OverflowStr
label byte
    db...0dh, 0ah
    db...*****
    db...0dh, 0ah
    db...'* The timer overflowed, so the interval timed was *'
    db...0dh, 0ah
    db...'* too long for the precision timer to measure.
    db...0dh, 0ah
    db...'* Please perform the timing test again with the
    db...0dh, 0ah
    db...'* long-period timer.
    db...0dh, 0ah
    db...
    db...0dh, 0ah

```

<db...

; * Вызывается процедура для запуска синхронизации.

ЗАДЕРЖКА **al, al**
sub

выход *TIMER_0_8253, al* ; lsb

ЗАДЕРЖКА *TIMER_0_8253, al* ;msb
выход

>pushcx _
pushf

<; Зафиксировать отсчет.

перемещение al,00000000b ; таймер блокировки 0
режим ВЫХОДА_8253, al

Проверьте, не превысил ли таймер, проверив
8259 на ожидание ; прерывание по таймеру.

```
перемещение al, 00001010b ; OCW3, настроен на считывание
out OCW3, al ; Регистрация запроса на прерывание
ЗАДЕРЖКА
    в _al, IRR ; считывание запроса на прерывание
                ; регистрация
                и al, 1 ; установите AL равным 1, если IRQ0 (the
                        >; timer interrupt) is pending
                        mov cs:[OverflowFlag],al; store the timer overflow
                        ; status
;
; Allow interrupts to happen again.
;
    sti
;
; Read out the count we latched earlier.
    in _al,TIMER_0_8253 ; least significant byte
    DELAY
    movah, al
    in _al,TIMER_0_8253 ; most significant byte
    xchg ah,al
    negax _; convert from countdown
                ; remaining to elapsed
                ; count
    mov cs: [TimedCount], ax
; Time a zero-length code fragment, to get a reference for how
; much overhead this routine has. Time it 16 times and average it,
; for accuracy, rounding the result.
    mov cs: [ReferenceCount],0
    mov cx,16
    cli; interrupts off to allow a
                ; precise reference count
RefLoop:
    call ReferenceZTimerOn
    call ReferenceZTimerOff
    loopRefLoop
    sti
    addcs:[ReferenceCount],8; total + (0.5 * 16)
    movc1,4
    shr cs: [ReferenceCount], cl; (total) / 16 + 0.5
; Restore original interrupt state.
    popax _; retrieve flags when called
    movch,cs:[OriginalFlags] ; get back the original upper
                ; byte of the FLAGS register
    andch,not 0fdh; only care about original
                ; interrupt flag...
    andah,0fdh;...keep all other flags in
                ; their current condition
    or _ah,ch; make flags word with original
                ; interrupt flag
    pushax _; prepare flags to be popped
; Restore the context of the program being timed and return to it.
    MPOPF; restore the flags with the
                ; original interrupt state
    popcx _
    popax _
    ret ZTimerOff
endp
i
; Called by ZTimerOff to start timer for overhead measurements.
;
ReferenceZTimerOn proc near
; Save the context of the program being timed.
    push ax
    pushf; interrupts are already off
; Set timer 0 of the 8253 to mode 2 (divide-by-N), to cause
; linear counting rather than count-by-two counting.
    mov al,00110100b ; set up to load
    outMODE_8253,al; initial
timer count
    DELAY
```

```
<>; Set the timer count to 0.
    sub al,al
    out TIMER_0_8253,al; load count lsb
    DELAY
    out TIMER_0_8253,al; load count msb
    ; Restore the context of the program being timed and return to it.
    MPOPF
    pop ax _
    ret

<>ReferenceZTimerOn endp
; Called by ZTimerOff to stop timer and add result to ReferenceCount
; for overhead measurements.
;
ReferenceZTimerOff proc near
; Save the context of the program being timed.
,
    толкаться _
    толкатьсясх _
    pushf
;
; Зафиксируйте счет и прочтите его.
,
    перемещение al, 00000000b; таймер блокировки 0
    РЕЖИМ_8253 выхода, al
    ЗАДЕРЖКА
    в _al, TIMER_0_8253 ; lsb
    ЗАДЕРЖКА
    перемещениеax, al
    в _al, TIMER_0_8253 ; msb
    xchg ah, al
    negax _; преобразовать из обратного отсчета
                                ; осталось до суммы
                                ; отсчитано
    добавить cs: [ReferenceCount], ax
3
; Восстановите контекст синхронизации программы и вернитесь к нему.
    MPOPF
    popсх _
    popax _
    ret

<>ReferenceZTimerOff endp
. . . . .
; * Routine called to report timing results.
. . . . .
ZTimerReport proc near
    pushf
    push ax
    push bx
    pushсх _
    pushdx _
    pushsi _
    pushds _
    pushcs...; DOS functions require that DS point
    popds _; to text to be displayed on the screen
    assumedс _: Code
;
; Check for timer 0 overflow.
;
    cmp [OverflowFlag],0
    jz PrintGoodCount
    mov dx,offset OverflowStr
    mov ah,9
    int 21h
    jmp short EndZTimerReport
; Convert net count to decimal ASCII in microseconds.
PrintGoodCount:
    movax, [TimedCount]
    sub ax, [ReferenceCount]
    mov si,offset ASCIICountEnd - 1
    ; Convert count to microseconds by multiplying by .8381.
    movdx, 8381
    muldx _
    movbx, 10000
```

```

    <>divbx _,* .8381 = * 8381 / 10000
    ; Convert time in microseconds to 5 decimal ASCII digits.
    mov bx, 10
    movex, 5
CTSLoop:
    subdx, dx
    divbx _
    add d1,'0'
    mov[si,d1
    decsi _
    100p CTSLoop
;
; Print the results.
,
    mov ah, 9
    mov dx, offset OutputStr
    int21h
EndZTimerReport:
    popds _
    popsi _
    popdx _
    popcx _
    popbx _
    popax _
    MPOPF
    ret
ZTimerReport endp
Code ends
    end
```

<Таймер Дзен - это средство, а не цель.

Остаток этой главы мы посвятим тому, чтобы увидеть, на что способен таймер Дзен, изучить, как он работает, и научиться им пользоваться. Я буду использовать дзен-таймер снова и снова на протяжении всей этой книги, поэтому важно, чтобы вы узнали, что может делать дзен-таймер и как им пользоваться. С другой стороны, вам ни в коем случае не обязательно точно понимать, как работает Zen timer. (Интересно, да; существенно, нет.)

Другими словами, таймер Дзен на самом деле не является частью знаний, которые мы ищем; скорее, это один из инструментов, с помощью которых мы приобретем эти знания. Следовательно, вам не стоит беспокоиться, если вы не до конца понимаете внутреннюю работу таймера Дзен. Вместо этого сосредоточьтесь на изучении того, как используйте его, и вы будете на правильном пути.

Запуск Zen Timer

ZTimerOn вызывается в начале сегмента кода, который должен быть синхронизирован. zTimerOn сохраняет контекст вызывающего кода, отключает прерывания, переводит таймер 0 из 8253 в режим 2 (режим деления на N), устанавливает начальный отсчет таймера равным 0, восстанавливает контекст вызывающего кода и возвращает. (Я хотел бы отметить, что, хотя документация Intel для 8253, похоже, указывает, что таймер не будет сброшен на 0, пока не завершит обратный отсчет, на практике, таймеры, похоже, сбрасываются на 0, как только они загружаются.)

Два аспекта ZTimeron заслуживают дальнейшего обсуждения. Один из интересных моментов заключается в том, что ZTimerOn отключает прерывания. (Позже ZTimerOff восстанавливает прерывания до состояния, в котором они были при вызове ZTimeron.) Прерывания не были отключены zTimeron, клавиатурой, мышью, таймер и другие прерывания могут возникать в течение временного интервала, и время, необходимое для обслуживания этих прерываний, будет некорректно и хаотично отображаться как часть время выполнения измеряемого кода. В результате код синхронизируется с помощью Zen таймер не должен ожидать возникновения каких-либо аппаратных прерываний в течение интервала между любой вызов zTimeron и соответствующий вызов zTimeroff и не должен включаться прерывания в течение этого времени.

Время и КОМПЬЮТЕР

Второй интересный момент о ZTimeron заключается в том, что он может вносить небольшую

неточность во время системных часов всякий раз, когда он вызывается. Чтобы понять, почему это итак, нам нужно изучить, как работают как 8253, так и системные часы ПК (которые сохраняют текущее время).

На самом деле 8253 содержит три таймера, как показано на рисунке 3.1. Все три таймера управляется кристаллом системной платы с частотой 14,31818 МГц, делится на 12, чтобы получить частоту 1,19318 Тактовая частота таймеров составляет 1,19318 МГц, поэтому таймеры отсчитывают один раз каждые 838,1 нс. Каждый из трех таймеров ведет программируемый обратный отсчет, генерируя сигнал на своем выходном выводе, когда он ведет обратный отсчет до 0. Каждый таймер может быть остановлен в любое время с помощью уровня 0 на его входном элементе; когда входное значение таймера равно 1, этот таймер ведет постоянный отсчет. В целом, таймеры 8253 по своей сути являются очень гибкими устройствами синхронизации; к сожалению, многое из этого гибкость зависит от способа подключения таймеров к внешним схемам и в ПК таймеры подключаются с учетом конкретных целей.

Таймер 2 приводит в действие динамик, хотя его можно использовать для других целей синхронизации, когда динамик не используется. Как показано на рисунке 3.1, таймер 2 - единственный таймер с программируемый вход вентиля в ПК; то есть таймер 2 является единственным таймером, который можно запускать и останавливать под программным управлением способом, указанным Intel. С другой стороны, выход таймера 2 не подключен ни к чему, кроме динамика. В частности, таймер 2 не может сгенерировать прерывание, чтобы привлечь внимание 8088.

Таймер 1 предназначен для обеспечения динамического обновления оперативной памяти, и его нельзя изменять чтобы не привести к сбоям системы.

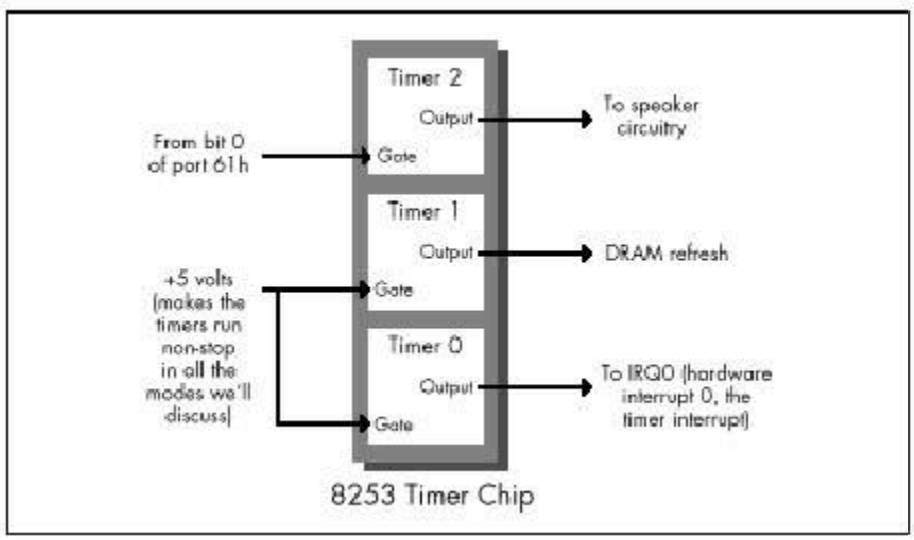


Рисунок 3.1 Конфигурация микросхемы таймера 8253 в ПК.

Наконец, таймер 0 используется для управления системными часами. Как запрограммировано BIOS при включении считается каждые 65 536 (64 КБ), или 54.925 миллисекунд, таймер 0 генерирует нарастающий фронт на выходной линии. (Миллисекунда равна одной тысячной секунды и сокращенно обозначается мс.) Эта линия подключена к линии аппаратного прерывания 0 (IRQ0) на системной плате, поэтому каждые 54,925 мс таймер 0 вызывает аппаратное прерывание 0.

Вектор прерывания для IRQ0 устанавливается BIOS во время включения питания, чтобы указывать на BIOS процедура TIMER_INT, которая ведет отсчет времени суток. TIMER_INT ведет 16-битный отсчет прерываний IRQ0 в области данных BIOS по адресу 0000:046C (все адреса в этом книги приведены в шестнадцатеричных парах сегмент: смещение); этот счетчик повторяется один раз в час (менее нескольких микросекунд), и когда это произойдет, TIMER_INT обновит 16-битный счетчик часов по адресу 0000: 046E в области данных BIOS. Это значение является основой для текущего времени и даты, которые DOS поддерживает с помощью функций от 2AH (2A шестнадцатеричной) до 2DH и by способ команд ДАТЫ и времени.

Каждый канал таймера 8253 может работать в любом из шести режимов. Таймер 0 в обычном режиме работает в режиме 3: прямоугольный режим. В режиме прямоугольной волны начальный отсчет выполняется обратный отсчет ведется по два раза за раз; когда отсчет достигает нуля, состояние вывода изменяется. Начальный отсчет снова отсчитывается по два раза за раз, и состояние вывода переключается обратно, когда счетчик достигает нуля. Результатом является прямоугольная волна, которая изменяет состояние медленнее, чем входной тактовый сигнал, в несколько раз по сравнению с начальным отсчетом. В своем обычном режиме таймер 0 генерирует выходной импульс, который является низким в течение примерно 27,5 мс и высоким в течение около 27,5 мс; этот импульс отправляется на контроллер прерываний 8259, и его нарастающий фронт

генерирует прерывание по таймеру каждые 54,925 мс.

Прямоугольный режим не очень полезен для точного отсчета времени, потому что он ведет обратный отсчет на два дважды за каждое прерывание по таймеру, тем самым делая невозможным точное определение времени. К счастью, 8253 предлагает другой режим таймера, mode 2 (режим деления на N), который является хорошей заменой режиму прямоугольной волны и идеальным режимом для точного отсчета времени.

В режиме деления на N обратный отсчет уменьшается на единицу от начального значения. Когда значение счетчика достигает нуля, таймер переключается и снова начинает обратный отсчет без остановки, а импульс генерируется в течение одного тактового периода. Хотя импульс не удерживается почти так долго, как в в режиме прямоугольной волны это не имеет значения, поскольку контроллер прерываний 8259 настроен в ПК работает по фронту и, следовательно, заботится только о наличии импульса от таймер 0, а не длительность импульса. В результате таймер 0 продолжает генерировать таймер прерывания в режиме деления на N, и системные часы продолжают показывать точное время.

Почему бы не использовать таймер 2 вместо таймера 0 для точного отсчета времени? В конце концов, таймер 2 имеет программируемый вентиляный вход и не используется ни для чего, кроме генерации звука. Проблема таймера 2 в том, что его выходной сигнал не может генерировать прерывание; фактически, таймер 2 ничего не может сделать но управляет динамиком. Нам нужно прерывание, генерируемое выводом таймера 0, чтобы сообщить нам когда счетчик превысил допустимые значения, и вскоре мы увидим, что прерывание по таймеру также позволяет отсчитывать гораздо более длительные периоды времени, чем поддерживает Zen timer, показанный в листинге 3.1 .

Фактически, таймер Zen, показанный в листинге 3.1, поддерживает временные интервалы продолжительностью до примерно 54 мс в длина, поскольку это период времени, который может быть измерен таймером 0 до начала его отсчета переворачивается и повторяется. Пятьдесят четыре мс могут показаться не очень долгим сроком, но даже Такой медленный процессор, как 8088, может выполнить более 1000 делений за 54 мс, а деление единственная инструкция, которую 8088 выполняет наиболее медленно. Если измеренный период оказывается если длительность превышает 54 мс (то есть, если таймер 0 завершил обратный отсчет и переключился), Zen таймер отобразит соответствующее сообщение. Долговременный таймер Zen для использования в таких случаях будет представлен позже в этой главе.

Таймер Zen определяет, был ли активирован таймер 0, проверяя, был ли ожидается прерывание IRQ0. (Помните, прерывания отключены во время работы таймера Zen, поэтому прерывание по таймеру не может быть распознано, пока таймер Zen не остановится и не включится прерывания.) Если ожидание прерывания IRQ0, то таймер 0 переключился и сгенерировал прерывание по таймеру. Напомним, что zT1meron изначально устанавливает таймер 0 на 0, чтобы обеспечить максимально длительный период — около 54 мс — до того, как таймер 0 достигнет 0 и сгенерирует прерывание по таймеру.

Теперь мы готовы рассмотреть способы, которыми таймер Zen может вносить неточности в системные часы. Поскольку таймер 0 изначально установлен в 0 таймером Zen, и поскольку системные часы тикают только тогда, когда таймер 0 отсчитывает 54,925 мс и снова достигает 0, при каждом запуске Zen timer возникает средняя погрешность в половину 54,925 мс, или около 27,5 мс. таймер Zen timer запускается. Кроме того, при включении таймера 0 генерируется прерывание по таймеру переключение из режима 3 в режим 2, опережающее системные часы на 54,925 мс, хотя это происходит только при первом запуске Zen timer после теплой или холодной загрузки. Наконец, при вызове zTimeroff снова может быть потеряно до 54,925 мс, поскольку эта процедура снова устанавливает счетчик таймера равным нулю. Конечный результат: системные часы будут работать до 110 мс (примерно на девятую долю секунды) медленнее при каждом использовании таймера Zen.

Потенциально гораздо большую неточность может вызвать временной код, выполнение которого занимает больше времени, чем около 110 мс. Напомним, что все прерывания, включая прерывание по таймеру, являются отключен во время синхронизации кода с таймером Zen. Контроллер прерываний 8259 способен запоминать не более одного ожидающего прерывания по таймеру, поэтому все прерывания по таймеру выполняются после первого один в течение любого заданного временного интервала Zen игнорируется. Следовательно, если временной интервал превышает 54,9 мс, системные часы фактически останавливаются через 54,9 мс после временного интервала.

Влияние таймера Zen на системное время не вызывает большого беспокойства, поскольку они временные, длятся только до следующей теплой или холодной загрузки. Системы, в которых есть часы с батарейным питанием (машины типа AT; то есть, практически на всех компьютерах общего пользования) автоматический сброс правильного времени при загрузке компьютера и систем часы без аккумулятора запрашивают правильную дату и время при загрузке. Кроме того, повторное использование таймера Zen обычно замедляет системные часы максимум на несколько секунд, если не засекается время выполнения кода, который занимает намного больше 54 мс (в этом случае таймер Zen уведомит вас, что код слишком длинный для засекания времени).

Тем не менее, рекомендуется перезагружать компьютер в конце каждого сеанса с помощью таймера Zen, чтобы убедиться, что системные часы работают правильно.

Остановка таймера Zen

В какой-то момент после вызова `ZTimeron` всегда должен вызываться `ZTimeroff`, чтобы отметить конец временного интервала. `ZTimeroff` сохраняет контекст вызывающей программы, фиксирует и считывает значение таймера 0, преобразует это значение из значения обратного отсчета, заданного таймером, сохраняет количество отсчетов, прошедших с момента вызова `zTimerOn`, и сохраняет Результат. Сразу после фиксации отсчета таймера 0 — и перед включением прерываний — `ZTimerOff` проверяет контроллер прерываний 8259, чтобы увидеть, есть ли ожидающий таймер прерывание, устанавливая флаг, чтобы отметить, что таймер переполнен, если ожидающий действительно есть прерывание по таймеру.

После этого `zTimeroff` выполняет только служебный код `zTimeron` и `zTimeroff` 16 время, а также усредняет и сохраняет результаты, чтобы определить, сколько отсчетов в только что полученном результате синхронизации было произведено из-за накладных расходов Zen timer, а не чем за счет синхронизации кода.

Наконец, `zTimeroff` восстанавливает контекст вызывающей программы, включая состояние флаг прерывания, который действовал, когда вызывался `ZTimeron` для запуска синхронизации, и возвращается.

Одним из интересных аспектов `ZTimeroff` является способ остановки таймера 0 для того, чтобы чтобы прочитать отсчет таймера. На самом деле нам не нужно останавливать таймер 0, чтобы прочитать отсчет; 8253 предоставляет специальную функцию чтения с фиксацией для конкретной цели чтения считайте, пока идет время. (Это тоже хорошо; у нас нет документированного способа остановить таймер 0, если бы мы захотели, поскольку вход его вентиля не подключен. Позже в этой главе, однако, мы увидим, что таймер 0 все-таки можно остановить.) Мы просто говорим 8253 зафиксироваться текущее количество, и 8253 выполняет это без прерывания работы.

Отчет о результатах синхронизации.

`ZTimerReport` может быть вызван для отображения результатов синхронизации в любое время как после `ZTimeron`, так и был вызван `ZTimeroff`. `ZTimerReport` сначала проверяет, работает ли таймер переполнен (отсчитывается до 0 и переворачивается) перед вызовом `ZTimeroff`; если переполнение действительно произошло, `zTimeroff` выводит соответствующее сообщение и возвращает результат. В противном случае, `ZTimerReport` вычитает количество ссылок (представляющее накладные расходы Zen timer) из количества, измеренного между вызовами `ZTimerOn` и `ZTimerOff`, преобразует результат подсчета времени по таймеру в микросекунды и выводит результирующее время в микросекунды до стандартного вывода.

Обратите внимание, что `ZTimerReport` необязательно вызывать сразу после `ZTimeroff`. Фактически, после после вызова `ZTimerOff` `ZTimerReport` может быть вызван в любое время вплоть до следующего вызов `ZTimerOn`.

ZTimerReport мешает нормальному отображению программы. Есть много способов справиться с этим . Одним из подходов является удаление вызовов строковой функции печати DOS (INT 21H с AH, равным 9) из ZTimerReport, вместо этого запустив программу под отладчик, поддерживающий переключение экрана (например, Turbo Debugger или CodeView), устанавливающий точку останова в начале ZTimerReport и непосредственно наблюдающий за подсчетом в микросекунды, как вычисляет ZTimerReport.

Второй подход заключается в модификации отчета ZTimer для размещения результата в некотором безопасном месте местоположение в памяти, например, неиспользуемая часть области данных BIOS.

Третий подход заключается в изменении ZTimerReport для печати результата через последовательный порт на терминал или на другой компьютер, выполняющий роль терминала. Аналогично, многие отладчики могут быть запущены с удаленного терминала по последовательному каналу связи.

Еще одним подходом является модификация отчета ZTimer для отправки результата на принтер через либо функцию DOS 5, либо прерывание BIOS 17H.

Последний подход заключается в изменении отчета ZTimer для вывода результата на вспомогательный вывод через Функцию DOS 4, а затем записать и загрузить специальный драйвер устройства с именем AUX, в который Вывод функции DOS 4 будет направлен автоматически. Этот драйвер устройства может отправлять результат куда угодно, по вашему желанию. Результат может отображаться на дополнительном дисплее адаптер через последовательный порт или к принтеру, или может просто храниться в буфере внутри драйвер, который будет сброшен позже. (Заслуга в этом окончательном подходе принадлежит Майклу Гири, и спасибо Дэвиду Миллеру за то, что поделился со мной идеей.)

Возможно, вы захотите разработать другие подходы, более подходящие для ваших нужд, чем те, что я представил. Действуйте! Я только что перечислил несколько возможностей, чтобы вы могли начать.

Примечания по таймеру Zen

Подпрограммы таймера Zen разработаны таким образом, чтобы их можно было практически вызывать с языка ассемблера код, работающий в общедоступном сегменте Code. Подпрограммы таймера Zen, однако, могут быть вызываться из любого кода на ассемблере или языке высокого уровня, который генерирует OBJ-файлы, которые совместим с компоновщиком Microsoft, просто изменив сегмент, в котором выполняется код таймера , чтобы соответствовать сегменту, используемому синхронизируемым кодом, или изменив Zen процедуры с таймером для процедур far и выполнение вызовов far для кода таймера Zen из код синхронизируется, как описано в конце этой главы. Все три подпрограммы сохраняют все регистры и все флаги, кроме флага прерывания, поэтому вызовы этих подпрограмм прозрачны для вызывающего кода.

Если вы измените процедуры таймера Zen на процедуры far, чтобы вызывать их из кода при запуске в другом сегменте убедитесь, что все процедуры таймера Zen выполнены с использованием far, включая ReferenceZTimerOn и ReferenceZTimerOff. (Вам придется переопределить FAR PTR для вызовы из ZTimeroff к последним двум подпрограммам, если вы все-таки переведете их далеко.) Если эталонные подпрограммы не того же типа — ближние или дальние — что и другие подпрограммы, они не будут отражать истинные накладные расходы, связанные с запуском и остановкой таймера Zen.

Пожалуйста, имейте в виду, что неточность, которую Zen timer может вносить в систему время на часах не влияет на точность измерений производительности, сообщаемых самим Zen timer. 8253 считает один раз каждые 838 нс, что дает нам разрешение счета около 1 мкс, хотя такие факторы, как очередь предварительной выборки (как описано ниже), динамичны Обновление оперативной памяти и внутренние изменения синхронизации в 8253 делают его, возможно, более точным можно описать Zen timer как измеряющий производительность кода с точностью лучше чем 10pps. Фактически, Zen timer наиболее точен в оценке кода производительность при временных интервалах, превышающих примерно 100 единиц. Во всяком случае, мы наиболее заинтересован в использовании таймера Zen для оценки относительной производительности различного кода последовательности — то есть, использовать его для сравнения и настройки кода — и таймер больше, чем достаточно точный для этой цели.

Таймер Zen работает на всех компьютерах, совместимых с ПК, на которых я его тестировал, включая XTs, Компьютеры ATS, PS / 2, а также машины на базе 386, 486 и Pentium. Конечно, у меня нет удалось протестировать это на всех ПК-совместимых устройствах, но я не ожидаю никаких проблем; компьютеры на которых таймер Zen не запускается, на самом деле нельзя назвать “ПК-совместимыми”.

С другой стороны, нет никакой гарантии, что производительность кода, измеряемая Zen timer, будет такой же на совместимых компьютерах, как и на подлинных IBM машин, или что абсолютная или относительная производительность кода будет одинаковой даже на разных моделях IBM; на самом деле верно совершенно обратное. Например, каждая PS / 2 компьютер, даже относительно медленный Model 30, выполняет код намного быстрее, чем ПК или XT. В качестве другого примера я решил установить тайминги для моей более ранней книги "Дзен Язык ассемблера на компьютере, совместимом с XT, только для того, чтобы обнаружить, что компьютер не совсем совместим с IBM в отношении производительности кода. Различия были незначительными, имейте в виду, но мой опыт показывает риск предположения, что компьютер определенной марки будет работать определенным образом без фактической проверки.

Не то чтобы это различие между моделями делало Zen timer хоть на йоту менее полезным — скорее наоборот. Zen timer - отличный инструмент для оценки производительности кода в течение весь спектр компьютеров, совместимых с ПК.

Пример использования Zen Timer

В листинге 3.2 показана тестовая программа для измерения производительности кода с помощью Zen таймер. Эта программа устанавливает DS равным CS (по причинам, которые мы вскоре обсудим), включает код, который будет измерен из файла TESTCODE, и вызывает ZTimer Report для отображения результаты синхронизации. Следовательно, измеряемый код должен быть в файле ТЕСТОВЫЙ КОД, и должен содержать вызовы ZTimerOn и ZTimerOff.

ЛИСТИНГ 3.2 PZTEST.ASM

```
>; Program to measure performance
of code that takes less than ; 54
ms to execute. (PZTEST.ASM) ;
; Link with PZTIMER.ASM (Listing
3.1). PZTEST.BAT (Listing 3.4) ; can
be used to assemble and link both
files. Code to be ; measured must be
in the file TESTCODE; Listing 3.3
shows ; a sample TESTCODE file. ;
; By Michael Abrash ;
mystack segment para stack 'STACK'
    db 512 dup(?)
mystack ends
; Code segment para
public 'CODE'
    assume cs: Code, ds: Code
    extrn ZTimerOn: near, ZTimerOff: near, ZTimerReport: near
    Start proc near
        push cs
        pop ds ; set DS to point to the code segment,
                ; so data as well as code can easily
                ; be included in TESTCODE
        include TESTCODE ;code to be measured, including
                ; calls to ZTimerOn and
ZTimerOff ; Display the
results. ;
        call ZTimerReport
    ;
; Terminate the
program. ;
        mov ah,4ch
        int 21h
    Start endp
Code ends
    end Start
```

<В листинге 3.3 показан некоторый пример кода для синхронизации. Этот список измеряет время требуется для выполнения 1000 загрузок AL из переменной памяти Memvar . Обратите внимание, что Листинг 3.3 вызывает zTimerOn для запуска синхронизации, выполняет 1000 инструкций mov подряд и

вызывает ZTimerOff для завершения синхронизации. Когда листинг 3.2 назван TESTCODE и включен в Листинг 3.3, листинг 3.2 вызывает ZTimerReport для отображения времени выполнения после выполнения кода из Листинг 3.3 выполнен.

ЛИСТИНГ 3.3 LST3-3.ASM

```
> ; Test file;
; Measures the performance of 1,000 loads of AL from
; memory. (Use by renaming to TESTCODE, which is
; included by PZTEST.ASM (Listing 3.2). PZTIME.BAT
; (Listing 3.4) does this, along with all assembly
; and linking.)
;
; jmp Skip ;jump around defined data
;
MemVar db?
;
Skip:
;
; Start timing.
;
        call ZTimerOn
        rept 1000
        mov al,[MemVar]
        endm
; Stop timing.
;
        call ZTimerOff
```

<Стоит отметить, что листинг 3.3 начинается с перехода к переменной памяти MemVar. Такой подход позволяет нам избежать воспроизведения листинга 3.2 полностью для каждого кода фрагмент, который мы хотим измерить; определяя любые необходимые данные прямо в сегменте кода и перемещаясь по этим данным, каждый список становится автономным и может быть подключен непосредственно в листинг 3.2 в качестве ТЕСТОВОГО КОДА. Листинг 3.2 устанавливает DS равным CS перед выполнением что-либо еще точно, чтобы данные могли быть встроены во фрагменты кода, рассчитанные по времени. Обратите внимание, что таймер Zen запускается только после выполнения начального перехода в листинге 3.3, поскольку мы не хотим включать время выполнения кода запуска во временной интервал. Вот почему вызовы ZTimerOn и ZTimeroff выполняются в тестовом коде, а не в PZTEST.ASM; таким образом, у нас есть полный контроль над тем, какая часть тестового кода синхронизирована, и мы можем исключить установочный код и тому подобное из временного интервала.

Листинг 3.3 используется путем присвоения ему названия TESTCODE, объединяющего оба листинга 3.2 (который включает ТЕСТОВЫЙ КОД) и листинг 3.1 с использованием TASM или MASM и связыванием двух результирующих OBJ-файлов вместе с помощью компоновщика Borland или Microsoft. В листинге 3.4 показан пакет файл PZTIME.BAT, который выполняет все это; при запуске этот пакетный файл генерирует и запускает исполняемый файл PZTEST.EXE. PZTIME.BAT (листинг 3.4) предполагает, что файл PZTIMER.ASM содержит листинг 3.1 и файл PZTEST.ASM содержит листинг 3.2. параметр командной строки для PZTIME.BAT - это имя файла, в который нужно скопировать ТЕСТОВЫЙ КОД и включен в PZTEST.ASM. (Обратите внимание, что Turbo Assembler может быть замените MASM, заменив “masm” на “tasm” и “link” на “tlink” в Листинге 3.4. То же самое верно для листинга 3.7.)

ЛИСТИНГ 3.4 PZTIME.BAT

```
>echo off
rem rem *** Listing 3.4 *** rem rem *****
rem * Batch file PZTIME.BAT, which
rem * builds and runs the precision *
rem * Zen timer program PZTEST.EXE to
rem * time the code named as the rem * command-line
rem * parameter. Listing 3.1 must
rem * be named rem * PZTIMER.ASM, and Listing
rem * 3.2 must be named PZTEST.ASM. To
rem * time the code in LST3-3, you'd
rem * type the DOS command: rem *
rem * pztime lst3-3 rem * rem * Note
rem * that MASM and LINK must be in
rem * the current directory or * rem * on
rem * the current path in order for this
rem * batch file to work. rem * rem * This
rem * batch file can be speeded up by assembling
rem * PZTIMER.ASM * rem * once, then
rem * removing the lines: rem
```



```
<>rem * masm pztimer;
rem * if errorlevel 1
goto errend
rem * from this file.
rem * rem * By
Michael Abrash
rem **** rem
rem Make sure a file
to test was
specified. rem if not
x%1==x goto ckexist
echo *****
echo * Please specify
a file to test.
echo ***** goto end
rem rem Make sure the
file exists. rem
: ckexist
if exist %1 goto docopy
echo *****
echo * The specified
file, "%1," doesn't
exist.
echo *****
goto end rem rem copy
the file to measure
to TESTCODE. rem
: docopy copy %1
testcode masm pztest;
if errorlevel 1 goto
errend masm pztimer;
if errorlevel 1 goto
errend link
pztest+pztimer; if
errorlevel 1 goto errend
pztest goto end : errend
echo *****
echo * An error occurred
while building the precision
Zen timer. * : end
```



<Предполагая, что листинг 3.3 называется LST3-3.ASM, а листинг 3.4 называется PZTIME.BAT, код в листинге 3.3 будет синхронизирован с командой:

```
>pztime LST3-3.ASM
```

<которая выполняет всю сборку и компоновку и сообщает время выполнения кода в Листинг 3.3.

Когда приведенная выше команда выполняется на оригинальном IBM PC с частотой 4,77 МГц, время, о котором сообщает таймер Zen, составляет 3619 МКС, или около 3,62 мкс на загрузку AL из памяти. (Хотя точное число составляет 3,619 мкс на загрузку AL, я собираюсь округлить эту последнюю цифру с этого момента. Независимо от того, сколько повторений данной инструкции синхронизировано, в процессе синхронизации возникает слишком много шума — между динамическим обновлением оперативной памяти и предварительной выборкой очередь и внутреннее состояние процессора в начале синхронизации — для того, чтобы эта последняя цифра имела какое-либо значение.) Учитывая тактовую частоту тестируемого ПК 4,77 МГц, это составляет около 17 циклов на тов, что на самом деле намного больше, чем заявленные Intel 10 циклов время выполнения этой инструкции. (Официальное время выполнения смотрите в документации MASM или TASM или в справочниках Intel по процессорам.) Не бойтесь, таймер Zen - это при правильном перемещении [MEMVAR] действительно занимает 17 циклов, как указано в Листинге 3.3. Именно так. почему это так, именно об этом и идет речь в этой книге.

Чтобы выполнить любой из тестов синхронизации, описанных в этой книге, введите листинг 3.1 и назовите его PZTIMER.ASM, введите листинг 3.2 и назовите его PZTEST.ASM, затем введите листинг 3.4 и назовите его PZTIME.BAT. Затем просто введите список, который вы хотите запустить, в поле filename файла и введите команду:

```
>pztime <filename>
```

<Фактически, именно так я рассчитал время выполнения каждого списка в этой книге. Фрагменты кода вам нужны точно таким же образом можно настроить время написания собственного кода. Если вы хотите задать время написания кода непосредственно в рlace в ваших программах, а не в тестовой программе из листинга 3.2, просто вставьте о вызовах ZTimerOn, ZTimerOff и ZTIMER сообщайте в соответствующих местах и связывайте PZTIMER с вашей программой.

Долговременный таймер Zen

За некоторыми исключениями, представленный выше Zen timer будет полезен нам в оставшейся части этой книги, поскольку мы сосредоточимся на относительно коротких кодовых последовательностях, которые как правило, выполнение занимает гораздо меньше 54 мс. Однако иногда нам потребуется использовать более длительные интервалы. Более того, весьма вероятно, что вы захотите установить время для кода последовательности длиннее 54 мс на каком-то этапе вашей карьеры программиста. Соответственно, Я также разработал таймер Zen для периодов длиннее 54 мс. Долговременный Zen таймер (названный так в отличие от только что представленного precision Zen timer), показанный в Листинг 3.5 позволяет измерять периоды продолжительностью до одного часа.

Ключевое различие между долговременным таймером Zen и прецизионным таймером Zen заключается в том, что долговременный таймер оставляет прерывания включенными в течение периода синхронизации. В результате прерывания таймера распознаются ПК, что позволяет BIOS поддерживать точное значение время системных часов в течение периода синхронизации. Теоретически это позволяет измерять сколь угодно длительные периоды. Однако, с практической точки зрения, нет необходимости в таймере, который может отсчитывать больше нескольких минут, поскольку время суток и дата в DOS функционируют (или, действительно, команды ДАТЫ и ВРЕМЕНИ в пакетном файле) отлично работают для более длительных интервалов. Поскольку очень большие временные интервалы не требуются, долговременный Zen таймер использует упрощенный способ расчета прошедшего времени, который ограничен измерением интервалы в час или меньше. Если установлен временной интервал более часа, то долговременный Zen таймер выводит сообщение о том, что он не может установить временной интервал такой длины.

По причинам реализации, долговременный таймер Zen также не способен синхронизировать код который начинается до полуночи и заканчивается после полуночи; если такая возможность возникает, долговременный таймер Zen также не может синхронизировать код. period Zen timer сообщает, что ему не удалось установить время для кода, потому что наступила полночь . Если это случится с вами, просто введите код еще раз, зная, что по крайней мере, вы не столкнетесь с проблемой снова в течение 23 с лишним часов.

Вам не следует использовать долговременный таймер Zen для тайм-кода, который требует выполнения прерываний отключается более чем на 54 мс с перерывом в течение временного интервала, поскольку при отключении прерывания для долговременного таймера Zen действуют те же максимальные 54 мс время измерения в качестве прецизионного таймера Zen.

Хотя разрешение прерывания по таймеру позволяет синхронизировать длительные интервалы, то же самое прерывание делает долговременный таймер Zen менее точным, чем precision Zen timer, поскольку время, затрачиваемое BIOS на обработку прерываний таймера в течение временного интервала, составляет включено во время, измеряемое долговременным таймером. Аналогично, любые другие прерывания , возникающие в течение временного интервала, в первую очередь прерывания работы клавиатуры и мыши, будут увеличить измеряемое время.

Долговременный таймер Zen оказывает примерно такое же влияние на системное время, как и точный таймер Zen, поэтому рекомендуется перезагрузить систему после сеанса работы с долговременный таймер Zen. Однако долговременный таймер Zen не обладает такими же возможностями возможность внесения значительной неточности в время системных часов в течение одного выполнения синхронизации, поскольку оно оставляет включенными прерывания и, следовательно, позволяет системным часам обновляться в обычном режиме.

Остановка часов

Существует потенциальная проблема с долговременным таймером Zen. Проблема заключается в следующем: для того, чтобы измерять время, превышающее 54 мс, мы должны поддерживать не одно, а два синхронизации компоненты, отсчет таймера 0 и отсчет времени суток в BIOS. Отсчет времени суток измеряет прохождение интервалов в 54,9 мс, в то время как счетчик 0 таймера измеряет время в пределах этих интервалов в 54,9 мс. Нам нужно прочитать две составляющие времени одновременно, чтобы получить точные показания. В противном случае мы можем прочитать счетчик таймера

непосредственно перед тем, как он перевернется и сгенерирует прерывание, затем прочитайте время суток в BIOS отсчет сразу после того, как произошло прерывание, которое привело к переключению счета времени суток, в результате погрешность измерения составила 54 мс. (Обратная последовательность — считывание отсчет времени суток, а затем отсчет таймера - может привести к неточности в 54 мс в другом направлении.)

Единственный способ избежать этой проблемы - остановить таймер 0, прочитать показания таймера и время суток отсчет дней при остановленном таймере, а затем перезапустить таймер. Увы, вход затвора в таймер 0 не управляется программой на ПК, таким образом, нет документированного способа остановить таймер. (Функция чтения с фиксацией, которую мы использовали в листинге 3.1, не останавливает таймер; она фиксируется отсчет, но таймер продолжает работать.) Что нам делать?

Как оказалось, недокументированная функция 8253 позволяет остановить таймер застопорился. Перевод таймера в новый режим и ожидание начального отсчета загрузка приводит к остановке таймера до загрузки отсчета. Удивительно, но счетчик таймера остается читаемым и корректным, пока таймер ожидает начальной загрузки.

По моему опыту, этот подход прекрасно работает с чипами, полностью совместимыми с 8253. Однако нет гарантии, что это будет работать всегда, поскольку оно программирует 8253 недокументированным способом. Более того, IBM решила не внедрять совместимость с этим конкретная функция 8253 в пользовательских чипах, используемых в компьютерах PS / 2. В PS / 2 компьютеры, у нас нет другого выбора, кроме как зафиксировать отсчет таймера 0, а затем остановить BIOS подсчет (путем отключения прерываний) как можно быстрее. Нам просто придется принять тот факт, что на компьютерах PS / 2 мы можем иногда получать показания, отклоняющиеся на 54 мс, и оставим все как есть.

Я настроил листинг 3.5 так, чтобы он мог собираться для использования или не использования недокументированного функция остановки по таймеру, как вам будет угодно. PS2 equate позволяет выбирать между двумя режимами работы . Если PS2 равно 1 (как указано в листинге 3.5), то используется метод защелкивания и считывания; если PS2 равно 0, тогда используется недокументированный подход с остановкой по таймеру. Функция "Защелкнуть и прочитать" метод будет работать на всех компьютерах, совместимых с ПК, но иногда может выдавать результаты, которые будут неверными на 54 мс. Подход с остановкой по таймеру позволяет избежать проблем с синхронизацией, но работает не на всех компьютерах.

ЛИСТИНГ 3.5 LZTIMER.ASM

```
<>; The long-period Zen timer. (LZTIMER.ASM)
; Uses the 8253 timer and the BIOS time-of-day
count to time the ; performance of code that
takes less than an hour to execute. ; Because
interrupts are left on (in order to allow the
timer ; interrupt to be recognized), this is less
accurate than the ; precision Zen timer, so it is
best used only to time code that takes ; more
than about 54 milliseconds to execute (code that
the precision ; Zen timer reports overflow
on). Resolution is limited by the ; occurrence of
timer interrupts. ; ; By Michael Abrash
>ZTimerOn: Saves the BIOS time of day count
; Externally-callable routines: ZTimerOff: Stops
and starts the long-period Zen timer.
the long-period Zen timer and saves the timer
count and the BIOS time-of-day count.
;ZTimerReport: Prints the time that passed between starting and
stopping the timer. ; Note: If either more
than an hour passes or midnight falls between
calls to ZTimerOn and ZTimerOff, an error is reported. For
timing code that takes more than a few minutes to execute,
либо командой DOS TIME в пакетном файле до и после
выполнения кода time, либо использованием DOS
функция определения времени суток вместо долговременного таймера Zen
более чем
достаточна. ;

<>; Note: The PS/2 version is assembled by setting the symbol PS2 to 1.
PS2 must be set to 1 on PS/2 computers because the PS/2's
timers are not compatible with an undocumented timer-stopping
```

```

<>feature of the 8253; the alternative timing approach that
must be used on PS/2 computers leaves a short window
during which the timer 0 count and the BIOS timer count may
not be synchronized. You should also set the PS2 symbol to
1 if you're getting erratic or obviously incorrect results.
; Note: When PS2 is 0, the code relies on an undocumented 8253
feature to get more reliable readings. It is possible that
the 8253 (or whatever chip is emulating the 8253) may be put
into an undefined or incorrect state when this feature is
used.

* If your computer displays any hint of erratic behavior
after the long-period Zen timer is used, such as the floppy*
drive failing to operate properly, reboot the system, set
PS2 to 1 and leave
<>Note: Each block of code being timed should ideally be run several
it that way! times, with at least two similar readings required to
establish a true measurement, in order to
; Note: Interrupts must not be disabled for more than 54 ms at a
stretch during the timing interval. Because interrupts
are enabled, keys, mice, and other devices that generate
interrupts should not be used during the timing interval.
; Note: Any extra code running off the timer interrupt (such as
some memory-resident utilities) will increase the time
measured by the Zen timer.
; Note: These routines can introduce inaccuracies of up to a few
tenths of a second into the system clock count for each
code section timed. Consequently, it's a good idea to
reboot at the conclusion of timing sessions. (The
battery-backed clock, if any, is not affected by the Zen
timer.) ; All registers and all
flags are preserved by all routines.

```

```

<>Code segment word public 'CODE'
assume cs:Code, ds:nothing
public ZTimerOn, ZTimerOff, ZTimerReport

```

```

<>; Set PS2 to 0 to assemble for use on a
fully 8253-compatible ; system; when PS2
is 0, the readings are more reliable if
the ; computer supports the undocumented
timer-stopping feature, ; but may be
badly off if that feature is not
supported. In ; fact, timer-stopping may
interfere with your computer's ; overall
operation by putting the 8253 into an
undefined or ; incorrect state. Use with
caution!!! ; ; Set PS2 to 1 to assemble
for use on non-8253-compatible
; systems, including PS/2 computers; when
PS2 is 1, показания ; иногда могут
сбиваться на 54 мс, но код будет
работать ; корректно во всех системах. ;
; Значение 1 безопаснее и будет работать в
большем количестве систем, ; в то время
как значение 0 дает более надежные
результаты в системах ; которые
поддерживают недокументированную функцию
остановки таймера ; 8253. Выбор за вами. ;
PS2равный1 ; Базовый адрес микросхемы
таймера 8253. ; BASE_8253равный40 часов ;
; Адрес счетчика таймера 0 регистрируется
в 8253. TIMER_0_8253equBASE_8253 + 0 ; Адрес
регистра режима в 8253. MODE_8253равный
BASE_8253 + 3 ; Адрес переменной подсчета
таймера BIOS в BIOS ; сегмент данных.
;
TIMER_COUNTравняется 46ch ; Макрос для эмуляции
инструкции POPF, чтобы исправить ошибку
в некоторых ; микросхемы 80286, которые позволяют
выполнять прерывания во время POPF, даже
когда ; прерывания остаются отключенными.
Макрос MPOPF
локальные p1, p2

```

```
<jmp short p2
p1:      ired      ; перейти к отправленному адресу и выставить флажки
p2:      нажать cs      ; сконструировать обратный адрес far для
вызова pl      ; следующая инструкция
endm

;
    >; Macro to delay briefly to ensure that enough time has elapsed
<>; between successive I/O accesses so that the device being accessed
; can respond to both accesses even on a very fast PC.
; DELAY
macro
    jmp $+2
    jmp $+2
    jmp $+2
    endm
    StartBIOSCountLowdw ?;BIOS count low word at the
                                ; start of the timing period
    StartBIOSCountHigh dw?;BIOS count high word at the
                                ; start of the timing period
    EndBIOSCountLowdw _?;BIOS count low word at the
                                ; end of the timing period
    EndBIOSCountHighdw ?;BIOS count high word at the
                                ; end of the timing period
    EndTimedCountdw ?;timer 0 count at the end of
                                ; the timing period
    ReferenceCount?;number of counts required to
                                ; execute timer overhead code
;
; String printed to report results.
;
OutputStr label byte
    db...0dh, 0ah, 'Timed count: '
TimedCountStr db 10 dup (?)
    db...microseconds', 0dh, 0ah
    db...' $'
; Temporary storage for timed count as it's divided down by powers
; of ten when converting from doubleword binary to ASCII.
;
CurrentCountLowdw _?
CurrentCountHigh dw?
; Powers of ten table used to perform division by 10 when doing
; doubleword conversion from binary to ASCII.
;
PowersOfTen label word
    dd _1
    dd _10
    dd 100
    dd _1000
    dd 10000
    dd 100000
    dd 1000000
    dd 10000000
    dd 100000000
    dd 1000000000
PowersOfTenEnd label word
; String printed to report that the high word of the BIOS count
; changed while timing (an hour elapsed or midnight was crossed),
; and so the count is invalid and the test needs to be rerun.
;
Байт метки TurnOverStr
    db 0dh, 0ah
    db
    db 0dh, 0ah
    db '* Либо прошла полночь, либо прошел час или больше * '
    db 0dh, 0ah
    db '* пока шел отсчет времени. Если первое значение было *'
    db 0dh, 0ah
    db ' * в этом случае, пожалуйста, повторите тест; если второе значение *'
    db 0dh, 0ah
    db '* было дело, тестовый код занимает слишком много времени, чтобы *'
    db 0dh, 0ah
    db ' * запуск должен быть рассчитан с помощью долговременного таймера Zen.
    db 0dh, 0ah
    db '* Рекомендации: используйте команду DOS TIME, DOS
    db 0dh, 0ah
    функцию времени db '* или часы.
    db 0dh, 0ah
    db 0dh, 0ah
    db '$'
; * Вызывается процедура
для запуска синхронизации.
```

<>ZTimerOn proc near

```

<>;
; Save the context of the program being timed.
    push ax
    pushf
; Set timer 0 of the 8253 to mode 2 (divide-by-N), to cause
; linear counting rather than count-by-two counting. Also stops
; timer 0 until the timer count is loaded, except on PS/2
; computers.
    mov al,00110100b;mode 2
    out MODE_8253, al
; Set the timer count to 0, so we know we won't get another
; timer interrupt right away.
; Note: this introduces an inaccuracy of up to 54 ms in the system
; clock count each time it is executed.
,
    DELAY
    sub al,al
    out TIMER_0_8253,al;lsb
    DELAY
    out TIMER_0_8253,al;msb
; ; In case interrupts are disabled, enable
interrupts briefly to allow ; the interrupt generated
when switching from mode 3 to mode 2 to be ; recognized. Interrupts
must be enabled for at least 210 ns to allow ; time for
that interrupt to occur. Here, 10 jumps are used for the
; delay to ensure that the delay time will be more than long enough
; even on a very fast PC.
    pushf
    sti
    rept 10
    jmp $+2
    endm
    MPOPF
;
; Store the timing start BIOS count.
; (Since the timer count was just set to 0, the BIOS count will
; stay the same for the next 54 ms, so we don't need to disable
; interrupts in order to avoid getting a half-changed count.)
;
    push ds
    subax, ax
    movds, ax
    movax, ds:[TIMER_COUNT+2]
    movcs: [StartBIOSCountHigh], ax
    movax, ds:[TIMER_COUNT]
    movcs: [StartBIOSCountLow], ax
    popds _
i
; Set the timer count to 0 again to start the timing interval.
    mov al,00110100b;set up to load initial
    out MODE_8253,al; timer count
    DELAY
    subal, al
    outTIMER_0_8253,al; load count lsb
    DELAY
    out TIMER_0_8253,al;Load count msb
; Restore the context of the program being timed and return to it.
    MPOPF
    pop ax
    ret
ZTimerOn endp

```

```

<>;* Routine called to stop timing and get count.

```

```

<>ZTimerOff proc near

```

```

<>; Save the context of the
program being timed. ;
    pushf
    push ax
    push cx ; В случае, если
прерывания отключены, включите прерывания ненадолго, чтобы разрешить
; для обработки любого ожидающего прерывания по таймеру. Прерывания
должны быть ; включен как минимум на 210 нс, чтобы
дать время для возникновения этого прерывания ;. Здесь для задержки
используется 10 переходов, чтобы гарантировать, что ; времени задержки
будет более чем достаточно даже на очень быстром ПК. ;

```

```

<>sti
rept 10
jmp $+2
endm

;
; Latch the timer count.
;

<если PS2

    mov al, 00000000b
    out MODE_8253, al        ; таймер блокировки отсчитывает 0

>; This is where a one-instruction-long window exists on the PS/2.
; The timer count and the BIOS count can lose synchronization;
; since the timer keeps counting after it's latched, it can turn
; over right after it's latched and cause the BIOS count to turn
; over before interrupts are disabled, leaving us with the timer
; count from before the timer turned over coupled with the BIOS
; count from after the timer turned over. The result is a count
; that's 54 ms too long.
;
else
;
; Set timer 0 to mode 2 (divide-by-N), waiting for a 2-byte count
; load, which stops timer 0 until the count is loaded. (Only works
; on fully 8253-compatible chips.)
;
    mov al,00110100b;mode 2
    out MODE_8253,a1
    DELAY
    mov al,00000000b;latch timer 0 count
    out MODE_8253,a1
    endif
    cli;stop the BIOS count
;
; Read the BIOS count. (Since interrupts are disabled, the BIOS
; count won't change.)
;
    push ds
    sub ax,ax
    mov ds,ax
    mov ax,ds:[TIMER_COUNT+2]
    mov cs: [EndBIOSCountHigh],ax
    mov ax,ds: [TIMER_COUNT]
    mov cs: [EndBIOSCountLow],ax
    pop ds
; Read the timer count and save it.
;
    in al,TIMER_0_8253;lsb
    DELAY
    mov ah,al
    in _a1, TIMER_0_8253;msb
    xchg ah,al
    neg ax;convert from countdown
                                ; remaining to elapsed
                                ; count

    mov cs: [EndTimedCount],ax
; Restart timer 0, which is still waiting for an initial count
; to be loaded.
,
ife PS2
    DELAY
    mov al,00110100b;mode 2, waiting to load a
                                ; 2-byte count

    out MODE_8253,a1
    DELAY
    sub al, al
    out TIMER_0_8253,a1; lsb
    DELAY
    mov al,ah
    out TIMER_0_8253,a1;msb
    DELAY
endif
    sti;let the BIOS count continue
; Time a zero-length code fragment, to get a reference for how
; much overhead this routine has. Time it 16 times and
average it,
; for accuracy, rounding the result.

```

```
<>movcs:[ReferenceCount],0
movex, 16
cli;interrupts off to allow a
; precise reference count
RefLoop:

call ReferenceZTimerOn
call ReferenceZTimerOff
loop RefLoop
sti
addcs:[ReferenceCount],8 ;total + (0.5 * 16)
mov cl,4
shr cs:[ReferenceCount],cl ;(total) / 16 + 0.5
; ; Восстановите контекст синхронизации
программы и вернитесь к нему.
pop cx
pop ax
mropf
ret
ZTimerOff endp
; ; Вызывается ZTimerOff для запуска
таймера для измерений накладных расходов.
; Ссылка на
timeron proc рядом
;
; Сохраните контекст программы, для которой выполняется синхронизация.
нажмите ax
pushf ; Установите таймер 0 из 8253
в режим 2 (деление на N), чтобы вызвать
; линейный подсчет, а не счет на два.
перемещение al, 00110100b; режим 2
выходMODE_8253, al
; Установите счетчик таймера на 0.
,
ЗАДЕРЖКА
SUBal,al
выходTIMER_0_8253, al; lsb
ЗАДЕРЖКА
выходTIMER_0_8253,al;msb
j
; Восстановите контекст синхронизируемой программы и вернитесь к нему.
mropf
poraх _
ret

<>ReferenceZTimerOn endp
; Called by ZTimerOff to stop the timer and add the result to
; ReferenceCount for overhead measurements. Doesn't need to look
; at the BIOS count because timing a zero-length code fragment
; isn't going to take anywhere near 54
ms. ReferenceZTimerOff proc near
;
; Save the context of the program being timed.
;
pushf
push ax
push cx
i
; Match the interrupt-window delay in ZTimerOff.
;
sti
rept 10
jmp $+2
endm
mov al,00000000b
outMODE_8253,al;latch timer
; Read the count and save it.
j
DELAY
in _al, TIMER_0_8253 ; lsb
DELAY
movah,al
in _al, TIMER_0_8253;msb
xchg ah,al
negax _;convert from countdown
; remaining to elapsed
```

```
<>; count add cs: [ReferenceCount], ax

; Restore the context

and return.

pop cx

pop ax

MPOPF

ret

<ReferenceZTimerOff endp

>;* Routine called to report timing results.
ZTimerReport proc near
    pushf
    pushax _
    pushbx _
    pushcx _
    pushdx _
    pushsi _
    pushdi _
    pushds _
    pushcs _;DOS functions require that DS point
    popds ; to text to be displayed on the screen
    assume ds: Code
; ; See if midnight or more than
an hour passed during timing. If so,
; notify the user.
    movax, [StartBIOSCountHigh]
    ctrap, [EndBIOSCountHigh]
    jz...CalcBIOSTime;hour count didn't change,
                                ; so everything's fine

    incax _
    ctrap, [EndBIOSCountHigh]
    jnzTestTooLong;midnight or two hour
                                ; boundaries passed, so the
                                ; results are no good

    movax, [EndBIOSCountLow]
    ctrap, [StartBIOSCountLow]
    jb _CalcBIOSTime;a single hour boundary
                                ; passed--that's OK, so long as
                                ; the total time wasn't more
                                ; than an hour
; Over an hour elapsed or midnight passed during timing, which
; renders the results invalid. Notify the user. Это не учитывает
; случай, когда прошло время, кратное 24 часам, но мы будем полагаться на
; проницательность пользователя в обнаружении этого случая.
TestTooLong:
    movah, 9
    mov dx, смещение оборотов
    int21 час
    jmpshort ZTimerReportDone
; Преобразует время BIOS в микросекунды.
CalcBIOSTime:
    movax, [Конечный результат]
    SUBax, [StartBIOSCountLow]
    movdx,54925; количество микросекунд в каждой
                                ; Счетчик BIOS представляет

    muldx _
    movbx, ax; отложите количество в BIOS в
    movex, dx; микросекунды

;
; Преобразуйте количество отсчетов таймера в микросекунды.
;
    movax, [количество на конец времени]
    movsi, 8381
    mulsi _
    движениеси,10000
    число делси _* .8381 = * 8381 / 10000
; Добавьте таймер и подсчеты BIOS вместе, чтобы получить общее время в
; микросекунды.
;
    Добавитьbx, ax
    ацпех, 0
; Вычитите накладные расходы таймера и сохраните результат.
;
    movax, [Количество ссылок]
    movsi, 8381; преобразовать количество ссылок
```

```

        <>mul si, _; to microseconds
        mov si, 10000
        div si, _ * .8381 = * 8381 / 10000
        sub bx, ax
        sbb cx, 0
        mov [CurrentCountLow], bx
        mov [CurrentCountHigh], cx ; Convert the result
to an ASCII string by trial subtractions of
; powers of 10.
        mov di, offset PowersOfTenEnd - offset PowersOfTen - 4
        mov si, offset TimedCountStr
CTSTimeNextDigit:
        mov bl, 00
CTSTimeLoop:
        mov ax, [CurrentCountLow]
        mov dx, [CurrentCountHigh]
        sub ax, PowersOfTen[di]
        sbb dx, PowersOfTen[di+2]
        jc ...CTSTimeNextPowerDown
        inc bl
        mov [CurrentCountLow], ax
        mov [CurrentCountHigh], dx
        jmp CTSTimeLoop
CTSTimeNextPowerDown:
        mov [si], bl
        inc si
        sub di, 4
        jns CTSTimeNextDigit
;
;
; Print the results.
;
        mov ah, 9
        mov dx, offset OutputStr
        int 21h
ZTimerReportDone:
        pop ds
        pop di
        pop si
        pop dx
        pop cx
        pop bx
        pop ax
        MPOPF
        ret
ZTimerReportendp
Codeends
end

```

<Более того, поскольку в нем используется недокументированная функция, подход с остановкой по таймеру может возможно, это приведет к сбоям в работе 8253, что, в свою очередь, может серьезно повлиять на ваш работа компьютера до следующей перезагрузки. В системах, не совместимых с 8253, я наблюдались не только совершенно неправильные результаты синхронизации, но и выход из строя дисководов для работайте должным образом после запуска долговременного таймера Zen с PS2, установленным на 0, поэтому будьте внимательны к признакам неисправности, если вы установите PS2 на 0.

Перезагрузка должна устранить любые проблемы, связанные с таймером, подобного рода, описанные выше. (Это дает нам еще одну причину для перезагрузки в конце каждого сеанса синхронизации кода.) Вам следует немедленно перезагрузиться и установить значение PS2 равным 1, если вы получаете неустойчивые или явно неправильные результаты результаты с длительным таймером Zen, когда для PS2 установлено значение 0. Если вы хотите установить для PS2 значение 0, то было бы неплохо приурочить несколько списков в этой книге к тому, чтобы для PS2 сначала было установлено значение 1 и затем на 0, чтобы убедиться, что результаты совпадают. Если они постоянно отличаются, вы должны установить PS2 равным 1.

Хотя версия, отличная от PS / 2, более опасна, чем версия PS / 2, она также дает более точные результаты, когда она работает. Если у вас компьютер, не совместимый с PS / 2 для ПК , выбор между двумя вариантами синхронизации остается за вами.

Если вы оставите значение PS2 равным 1 в листинге 3.5, вам следует повторить каждую синхронизацию кода запустите несколько раз, прежде чем рассчитывать на точность результатов более 54 мс, поскольку возможны отклонения из-за возможного отсутствия синхронизации между таймером 0 подсчет и подсчет времени суток в BIOS. На самом деле, хорошей идеей будет выделять больше времени на кодирование, чем

однажды, независимо от того, какую версию долговременного таймера Zen вы используете, поскольку прерывания, который должен быть включен для правильной работы долговременного таймера, может произойти в любое время и может существенно изменить время выполнения.

Наконец, обратите внимание, что precision Zen timer отлично работает как на PS / 2, так и на компьютерах, отличных от PS / 2. Рекомендации PS / 2 и 8253, которые мы только что обсудили, применимы только к долговременному таймеру Zen.

Пример использования долговременного таймера Zen

Долговременный таймер Zen имеет точно такой же интерфейс вызова, что и precision Zen таймер, и его можно использовать вместо precision Zen timer, просто привязав его к коду для синхронизации вместо привязки кода точного таймера. Всякий раз, когда precision Zen таймер сообщает вам, что время синхронизации кода занимает слишком много времени для точного таймера, чтобы обработать, все, что вам нужно сделать, это вместо этого ввести ссылку на долговременный таймер.

В листинге 3.6 показана программа для тестирования долговременного таймера Zen. Пока эта программа аналогично листингу 3.2, стоит отметить, что листинг 3.6 ожидает несколько секунд перед вызовом ZTimeron, тем самым позволяя обрабатывать все ожидающие прерывания с клавиатуры. Поскольку прерывания должны быть оставлены включенными, чтобы периоды времени превышали 54 мс, прерывания генерируемые нажатиями клавиш (включая движение вверх при нажатии клавиши Enter, которая запускает программу) — или любые другие прерывания, если на то пошло — могут привести к неправильному увеличению времени записывается долговременным таймером Zen. В свете этого, не поддавайтесь искушению печатать перемещайте мышь вперед или тому подобное, пока долговременный таймер Zen отсчитывает время.

ЛИСТИНГ 3.6 LZTEST.ASM

```
>; Program to measure performance of code that takes longer than
; 54 ms to execute. (LZTEST.ASM)
;
; Link with LZTIMER.ASM (Listing 3.5). LZTIME.BAT (Listing 3.7)
; can be used to assemble and link both files. Code to be
; measured must be in the file TESTCODE; Listing 3.8 shows
; a sample file (LST3-8.ASM) which should be named TESTCODE.
;
; Майкл Абраш
сегмент mystack para stack 'СТЕК'
    db...512 dup (?)
заканчивается mystack
Сегмент кода, являющийся общедоступным "КОДОМ"
    предположим,cs: код, ds: Код
    extrnZTimerOn: рядом, ZTimerOff: рядом, ZTimerReport: рядом
    Запустить процесс рядом
    нажать cs
    нажать ds; укажите DS на сегмент кода,
                ; таким образом, данные, а также код могут быть легко
                ; включены в ТЕСТОВЫЙ код
; Задержитесь на 6-7 секунд, чтобы снова запустить нажатие клавиши Enter, запустившее
; программу.
,
    движениеeah, 2ch
    int21h; узнать текущее время
    movbh, dh; отложить текущее время в сторону
Задержка цикла:
    перемещениеeah, 2ch
    нажмите bx; сохранить время начала
    int21 час; получить время
    popbx _; получить время начала
    стрpdh, bh; отсчет новых секунд меньше, чем
                ; отсчет начальных секунд?
    время проверки jnb; нет
    Добавитьdh, 60; да, должно быть, прошла минута,
                ; поэтому добавьте одну минуту
Проверьте время задержки:
    к югу от dh, bh; получите прошедшее время
    стрpdh,7; прошло уже больше 6 секунд?
    jb _DelayLoop; пока нет
;
    включить ТЕСТОВЫЙ КОД;код, подлежащий измерению, включая вызовы
                ; ZTimerOn и ZTimerOff
; Отобразить
результаты.
;
```

```
<>call ZTimerReport
; Terminate the program.

mov ah,4ch

int 21h

Start endp

Code ends

endStart
```

<Как и в случае с precision Zen timer, программа из листинга 3.6 используется путем присвоения имени файлу , содержащему тестовый код для синхронизации, с последующей сборкой как из листинга 3.6, так и из Листинг 3.5 с использованием MASM или TASM и связывание двух файлов вместе с помощью Компоновщика Microsoft или Borland. В листинге 3.7 показан командный файл с именем LZTIME.BAT, который выполняет все вышеперечисленное, генерируя и запуская исполняемый файл LZTEST.EXE. LZTIME.BAT предполагает, что файл LZTIMER.ASM содержит листинг 3.5, а файл LZTEST.ASM содержит листинг 3.6.

ЛИСТИНГ 3.7 LZTIME.BAT

```
>echo off
rem
rem *** Listing 3.7 ***
rem
rem ...
rem * Batch file
LZTIME.BAT, which builds
and runs the
rem * long-period Zen timer
program LZTEST.EXE to time
the code rem * named as the
command-line
parameter. Listing 3.5 must be
rem * named
LZTIMER.ASM, and Listing 3.6
must be named
rem * LZTEST.ASM. To time the
code in LST3-8, you'd type
the rem * DOS command: rem
rem * lztime lst3-8 rem *
rem * Note that MASM and
LINK must be in the current
directory or * rem * on the current
path in order for this batch
file to work. rem * rem * This
batch file can be speeded up
by assembling LZTIMER.ASM *
rem * once, then removing
the lines: rem * rem * masm
lztimer; rem * if errorlevel
1 goto errend rem *
rem * from this file. rem *
rem * By Michael Abrash rem *****
rem rem Make sure a
file to test was specified.
rem если нет %1==x goto
skexist echo *****
echo * Пожалуйста, укажите
файл для тестирования. echo *****
переход к завершению
rem rem Убедитесь, что
файл существует.
примечание : skexist если
существует %1, перейдите к документации
echo ***** echo * Указанный
файл "%1" не существует.
echo ***** переход
к концу rem rem скопируйте файл
для измерения в TESTCODE.
: doscopy скопируйте %1
testcode masm lztest;
если ошибка уровня 1 приведет
к ошибке masm lztimer; if errorlevel
1 goto исправление ошибки ссыла
lztest + lztimer; if errorlevel
1 goto исправление ошибки
lztest начало завершения : ошибка
echo ***** echo * Произошла
ошибка при сборке долговременного
таймера Zen. : конец
```



<В листинге 3.8 показан пример кода, который можно синхронизировать с помощью программы тестирования, представленной в Листинге 3.6. В листинге 3.8 указано время, необходимое для выполнения 20 000 загрузок AL из памяти, слишком большой промежуток времени для точного таймера Zen на 8088.

ЛИСТИНГ 3.8 LST3-8.ASM

```
>; Measures the performance of 20,000
loads of AL from ; memory. (Use by
renaming to TESTCODE, which is
; included by LZTEST.ASM (Listing
3.6). LZTIME.BAT ; (Listing 3.7) does
this, along with all assembly ; and
linking.) ; ; Note: takes about ten
minutes to assemble on a slow PC
if ;you are using MASM ;
    jmpSkip ;jump around defined data
;
MemVar db?
;
Skip:
;
; Start
timing. ;
    call ZTimerOn
    rept 20000
    mov al, [MemVar]
    endm
; Stop timing.
    call ZTimerOff
```

<Когда LZTIME.BAT запускается на ПК с помощью следующей командной строки (при условии, что код в листинге 3.8 приведен файл LST3-8.ASM)

```
>lztime lst3-8.asm
```

<результат равен 72 544 долларам США, или примерно 3,63 долларам США на загрузку AL из памяти. Это лишь немного дольше, чем время на загрузку AL, измеренное прецизионным таймером Zen, как и следовало бы ожидать, учитывая, что прерывания остаются включенными с помощью долговременного таймера Zen. Дополнительный доли микросекунды, измеряемые на mov, отражают время, необходимое для выполнения Код BIOS, который обрабатывает 18,2 прерывания по таймеру, происходящие каждую секунду.

Обратите внимание, что выполнение команды может занять до 10 минут на медленном компьютере, если вы используете MASM, при этом большая часть этого времени уходит на сборку листинга 3.8. Почему? Потому что MASM, как известно, медленно собирает блоки REPT, и блок в листинге 3.8 повторяется 20 000 раз.

С использованием таймера Zen от C

Таймер Zen можно использовать для измерения производительности кода при программировании на C - но не сразу из коробки. Как было показано ранее, таймер предназначен для вызова из язык ассемблера; требуются некоторые относительно незначительные модификации перед ZTimeron (таймер запуска), ZTimeroff (таймер остановки) и ZTimerReport (отображение результатов синхронизации) подпрограммы могут вызываться из C. Здесь необходимо рассмотреть два отдельных случая: small модель кода и large; Сначала я займусь более простым, моделью small code.

Изменение таймера Zen для привязки к программе small code model C включает следующее шаги: Измените ZTimeron на _ZTimeron, измените ZTimeroff на _ZTimeroff, измените Ztimerрепортируйте в _ZTimerReport и измените код на _TEXT . На рисунке 3.2 показана строка номера и новые состояния всех строк из листинга 3.1, которые необходимо изменить. Эти изменения преобразуют код для использования имен внешних меток в стиле C и маленькой модели C сегмент кода. (В C ++ используйте спецификатор “C”, как в

```
>extern "C" ZTimerOn(void);
```

<при объявлении подпрограмм таймера extern, чтобы не происходило искажения имен, и компоновщик может найти имена подпрограмм в стиле C.)

Это все, что требуется; после выполнения этого вы сможете использовать таймер Zen из C, как, например, в:

```
ZTimerOn():
```

```
>for (i=0, x=0; i<100; i++)
    x += i;
ZTimerOff();
ZTimerReport();
```

<(Здесь я говорю о прецизионном таймере. Долговременный таймер — листинг 3.5 — требует тех же изменений, но в разных строках.)

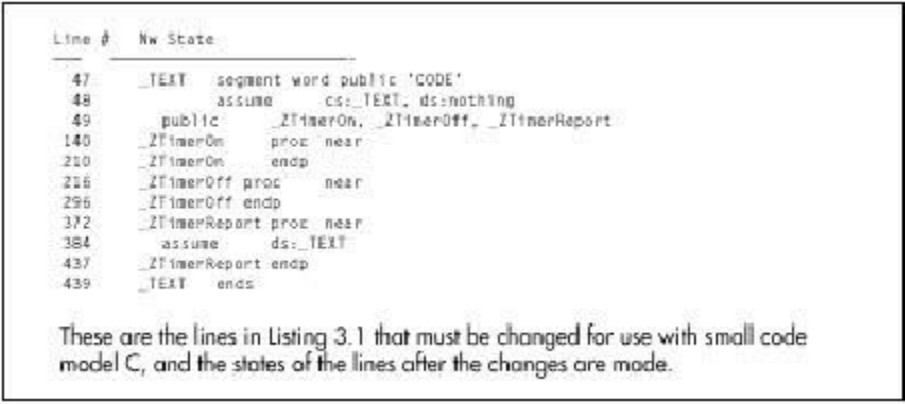


Рисунок 3.2 Изменения для использования с небольшим кодом модели C.

Изменение таймера Zen для использования в модели большого кода C немного сложнее, потому что в дополнение к вышеуказанным изменениям все функции, включая внутреннюю контрольную синхронизацию процедуры, которые используются для расчета накладных расходов, чтобы их можно было вычесть, должны быть преобразованы в far . На рисунке 3.3 показаны номера строк и новые состояния всех строк из Листинг 3.1, который необходимо изменить, чтобы вызвать таймер Zen из модели с большим кодом C. Опять же, номера строк относятся к прецизионному таймеру, но длиннопериодный таймер - это очень похоже.

Полный список таймеров Zen с возможностью C-вызова представлен в главе К на прилагаемом компакт-диске.

Следите за оптимизацией ассемблеров!

Один важный совет по технике безопасности при модификации Zen timer для использования с большим кодом модели C код: остерегайтесь оптимизации ассемблеров! TASM фактически заменяет

```
>callfar ptr ReferenceZTimerOn
```

<на

```
>pushcs _ callnear
ptr ReferenceZTimerOn
```

<(и аналогично для ReferenceZTimerOff), который работает, потому что ReferenceZTimerOn находится в том же сегменте, что и вызывающий код. Обычно это отличная оптимизация, поскольку она и меньше, и быстрее, чем удаленный вызов.

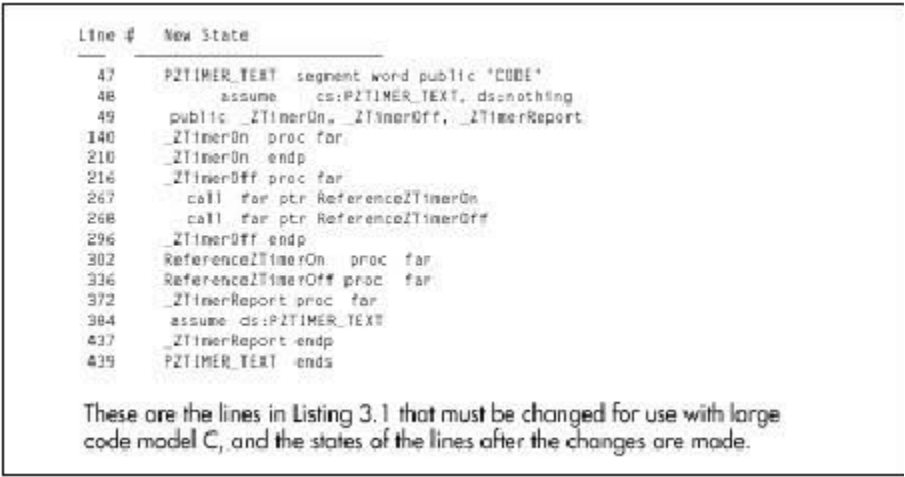


Рисунок 3.3 Изменения для использования с моделью большого кода C.

Однако для таймера Zen это не так здорово, потому что наша цель при вызове контрольный код синхронизации - точно определить, сколько времени занимает служебный код —включая вызовы far для ZTimerOn и ZTimerOf! Путем преобразования вызовов far в пары push / near вызовов в модуле Zen timer, TASM делает невозможной эмуляцию точно такие же накладные расходы, как у Zen timer, и незначительно увеличивает время (около 16 циклов на 386) менее точный.

Каково решение? Поместите директиву NOSMART в начало кода Zen timer. Это

директива предписывает TASM отключить все оптимизации, включая преобразование дальних вызовов в пары push / near вызовов. Кстати, насколько мне известно, такой проблемы нет с MASM до версии 5.10A.

На мой взгляд, весь бизнес по оптимизации ассемблеров - это смешанное благо. В общем, приятно, что ассемблер сокращает переходы и выбирает расширенные по знаку формы инструкции для вас. С другой стороны, преимущества таких приемов, как замена пар push / near пары вызовов для дальних вызовов относительно невелики, и эти приемы могут помешать, когда необходим полный контроль. Конечно, полный контроль требуется очень редко, но когда это происходит, оптимизация ассемблеров может вызвать небольшие проблемы; Я обнаружил изменение far от TASM вызывает только потому, что мне довелось просмотреть код в отладчике, и вы, возможно, захотите сделайте то же самое, если вы используете последнюю версию MASM.

Я протестировал изменения, показанные на рисунках 3.2 и 3.3, с TASM и Borland C++4.0 а также с новейшим компилятором MASM и

Дальнейшее чтение

Для тех из вас, кто желает продолжить изучение механики измерения кода, есть одна хорошая статья об измерении производительности кода с помощью таймера 8253 "Программирование Понимание: высокопроизводительный анализ программного обеспечения на IBM PC" Байрона Шеппарда, который появился в январском выпуске журнала Byte за 1987 год. Для полноты, хотя и несколько загадочно информацию о самом таймере 8253 я отсылаю вас к компонентам микросистемы Intel Справочник, который также является полезным справочником по ряду других компонентов ПК, включая программируемый контроллер прерываний 8259 и контроллер DMA 8237. Для подробные сведения о том, как 8253 используется в ПК, а также множество дополнительных я предлагаю вам ознакомиться с аппаратными средствами ПК и ресурсами BIOS в IBM серия технических справочных руководств для ПК, XT, AT, модели 30 и microchannel компьютеры, такие как модели 50, 60 и 80.

Однако для наших целей не так важно, чтобы вы точно понимали, как работает Zen таймер. Все, что вам действительно нужно знать, - это на что способен Zen timer и как им пользоваться, и мы достигли этого в этой главе.

Вооружившись Zen Timer, вперед и выше

Таймер Zen не идеален. Во-первых, это самое высокое разрешение, которое он может измерять интервал составляет в лучшем случае около 1 секунды, период времени, за который компьютер Pentium с частотой 66 МГц может выполнить до 132 инструкций (хотя ПК на базе 8088 было бы сложно справиться с двумя инструкциями за микросекунду). Другая проблема заключается в том, что синхронизирующий код сам по себе влияет на состояние очереди предварительной выборки и кэша процессора в начало синхронизируемого кода, поскольку синхронизирующий код не обязательно выбирается и не обязательно обращается к памяти точно в той же временной последовательности, что и обычно код, непосредственно предшествующий измеряемому коду. Эта предварительная выборка эффект может приводить к погрешности от 3 до 4 мкс. Аналогично, состояние очередь предварительной выборки в конце кода, который синхронизируется, влияет на то, сколько времени требуется для выполнения кода, который останавливается время, необходимое таймеру для выполнения. Следовательно, таймер Zen имеет тенденцию быть более точным для более длинных кодовых последовательностей, поскольку относительная величина неточности, вносимой таймером Zen, становится меньше с более длительными периодами.

Несмотря на недостатки, Zen timer является хорошим инструментом для изучения кода C и x86 язык ассемблера семейства, и это инструмент, которым мы будем часто пользоваться в оставшейся части этой книги.

Как аппаратное обеспечение ПК поглощает производительность кода

Эта глава, адаптированная из моей предыдущей книги "Дзен языка ассемблера", находится на сопутствующий компакт-диск, раскрывающий суть моей философии оптимизации: Поймите, куда на самом деле уходит время, когда выполняется ваш код. Это может звучать до смешного просто, но, как становится ясно из этой главы, это оказывается сложной задачей действительно, иногда это граничит с черной магией. Эта глава - давняя любимая моя, потому что это была первая — и в значительной степени единственная — работа, которую я знаю об этом обсуждали этот материал, тем самым знакомя поколение программистов ПК с педалями оптимизация по принципу "до-металла".

Эта глава почти полностью посвящена первому популярному процессору семейства x86, 8088. Некоторые из конкретных функций и результатов, которые я привожу в этой главе, больше не являются применимо к современным процессорам семейства x86, таким как 486 и Pentium, как я укажу позже, когда мы будем обсуждать эти процессоры. Тем не менее, общая тема этого глава о том, что понимание малозаметных и плохо документированных гремлинов кода, называемых пожиратели циклов, которые скрываются в вашей системе, необходимы для программирования производительности - это каждый немного актуален и сегодня. Кроме того, в последующих главах часто упоминаются основные пожиратели циклов описанные в этой главе, поэтому эта глава является основой для обсуждения x86- грядет семейная оптимизация. Более того, Zen timer остается отличным инструментом, с помощью которого можно выявлять и проверять циклоедов, как мы увидим в последующих главах, и это глава является лучшей иллюстрацией того, как использовать Zen timer, какую вы, вероятно, найдете.

Итак, не берите ни абсолютное, ни относительное время выполнения, представленное в этой глава как евангелие для новых процессоров, и читайте дальше, чтобы узнать, как пожиратели циклов и правила оптимизации менялись с течением времени, но уделите время на по крайней мере, бегло прочтите эту главу, чтобы получить хорошее представление о материале остальной части этой книги.

Любители циклов

Программирование имеет много уровней, начиная от привычных (языки высокого уровня, DOS вызовы и т.п.) и заканчивая эзотерическими вещами, которые лежат на темном краю страна железа. Я называю их пожирателями циклов, потому что, как монстры в плохих хоррорах 50-х они прячутся в этих тенях, принимая участие в работе вашей программы не обращая внимания на силы добра или армию США. В этой главе мы собираемся переходите сразу к самому низкому уровню, изучая пожирателей циклов, которые находятся под программный интерфейс; то есть, фактически, под вашим приложением, DOS и BIOS, под самим набором инструкций.

Зачем начинать с самого низкого уровня? Просто потому, что пожиратели циклов влияют на производительность всего код на ассемблере, и все же они почти неизвестны большинству программистов. Полное понимание оптимизация кода требует понимания пожирателей циклов и их последствий. Это непростая задача, и на самом деле именно в этой области большинство книг и статей о программировании на ассемблере не дотягивают.

Почти во всей литературе по программированию на ассемблере обсуждается только программирование интерфейс: набор инструкций, регистры, флаги, вызовы BIOS и DOS. Эти разделы наиболее полно охватывают функциональность ассемблерных программ, но это производительность превыше всего, что нам нужно. Никто никогда не рассказывает вам о первичных элементах производительность, которая лежит за интерфейсом программирования, в смутно видимой сфере - заполняется предварительной выборкой команд, динамическим обновлением оперативной памяти и состояниями ожидания — где

программное обеспечение соответствует оборудованию. Эта область является прерогативой инженеров-аппаратчиков и почти никогда не обсуждается, поскольку она касается производительности кода. И все же только понимая механизмы, работающие на этом уровне, мы можем полностью понять их должным образом повысить производительность нашего кода.

Что приводит нас к пожирателям циклов.

Природа велосипедоедов

Велосипедоеды - это гремлины, которые живут в автобусе или на периферии (а иногда и внутри сам процессор), замедляя производительность кода ПК, так что он не выполняется в полном объеме Скорость. Большинство пожирателей циклов (и все те, кто преследует старые процессоры Intel) живут вне исполнительного блока центрального процессора, где они могут влиять на центральный процессор только тогда, когда центральный процессор выполняет доступ к шине (чтение или запись данных памяти или ввода-вывода). Как только ваш код и данные уже внутри центрального процессора, эти пожиратели циклов больше не могут быть проблемой. Только в процессорах 486 и Pentium вы найдете пожиратели циклов внутри чипа, как мы увидим в последующих главах.

Природа и серьезность пожирателей циклов сильно различаются от процессора к процессору процессор и (особенно) архитектура памяти в зависимости от архитектуры памяти. Для того, чтобы чтобы понять их все, нам нужно сначала разобраться с самыми простыми из них, теми, которые преследовали оригинальный IBM PC на базе 8088. Позже в этой книге я смогу лучше объяснить новое поколение пожирателей циклов в терминах тех предков-пожирателей циклов - но сначала мы должны подготовить основу.

Наследственные пожиратели циклов 8088

Внутри 8088 представляет собой 16-разрядный процессор, способный работать на полной скорости в любое время — если только не требуются внешние данные. Внешние данные должны проходить через внешние данные 8088 шина данных и шина данных ПК по одному байту за раз к периферийным устройствам и обратно с использованием пожирателей циклов подстерегающий на каждом этапе пути. Более того, внешние данные включают в себя не только операнды памяти, но и байты инструкций, так что даже инструкции без памяти операнды могут страдать от "пожирателей циклов". Поскольку некоторые из самых быстрых инструкций 8088 - это инструкции только для регистрации, это действительно важно.

Основными потребителями циклов являются:

- 8-разрядная внешняя шина данных 8088.
- Очередь предварительной выборки.
- Динамическое обновление оперативной памяти.
- Состояния ожидания, в частности, отображают состояния ожидания памяти, а в компьютерах AT и 80386 состояния ожидания системной памяти.

Расположение этих пожирателей циклов в ПК на базе primordial 8088 показано на Рисунок 4.1. В этой главе мы рассмотрим каждого из пожирателей циклов по очереди. Материал будет непростым, поскольку циклоеды относятся к числу наиболее тонких аспектов сборки программирование. В то же время, однако, это будет одна из самых важных и полезных глав в этой книге. Не волнуйтесь, если вы не поймете всего в этой главе, но прочтите все это, даже если будет немного трудно. Любители велоспорта играют ключевую роль в последующих главах, поэтому некоторое знакомство с ними крайне желательно.

8-разрядный пожиратель циклов шины

Посмотрите! Вниз, на материнскую плату! Это 16-разрядный процессор! Это 8-разрядный процессор! Это...

... 8088!

Поклонники 8088 называют его 16-разрядным процессором. Поклонники других 16-разрядных процессоров называют 8088 8-разрядным процессором. Правда в том, что 8088 - это 16-разрядный процессор, который

часто работает как 8-разрядный процессор.

8088 внутренне является полностью 16-разрядным процессором, эквивалентным 8086. (Фактически, 8086 идентичен 8088, за исключением того, что у него полноценная 16-разрядная шина. 8088 в основном бедный man's 8086, потому что он позволяет построить более дешевую, хотя и более медленную систему благодаря шине меньшего размера.) С точки зрения набора команд, 8088 явно является 16-разрядным процессором, способным выполнять любую заданную 16-разрядную операцию — сложение, вычитание, даже умножение или деление — с помощью одной инструкции. Однако внешне 8088 является однозначно 8-разрядным процессором, поскольку ширина внешней шины данных составляет всего 8 бит. Другими словами ширина программного интерфейса составляет 16 бит, но аппаратного интерфейса всего 8 шириной в биты, как показано на рисунке 4.2. Результат этого несоответствия прост: данные размером со слово может быть передан между 8088 и памятью или периферийными устройствами только с половиной максимальной скорости 8086, то есть вдвое меньше максимальной скорости, на которую был рассчитан Исполнительный блок 8088.

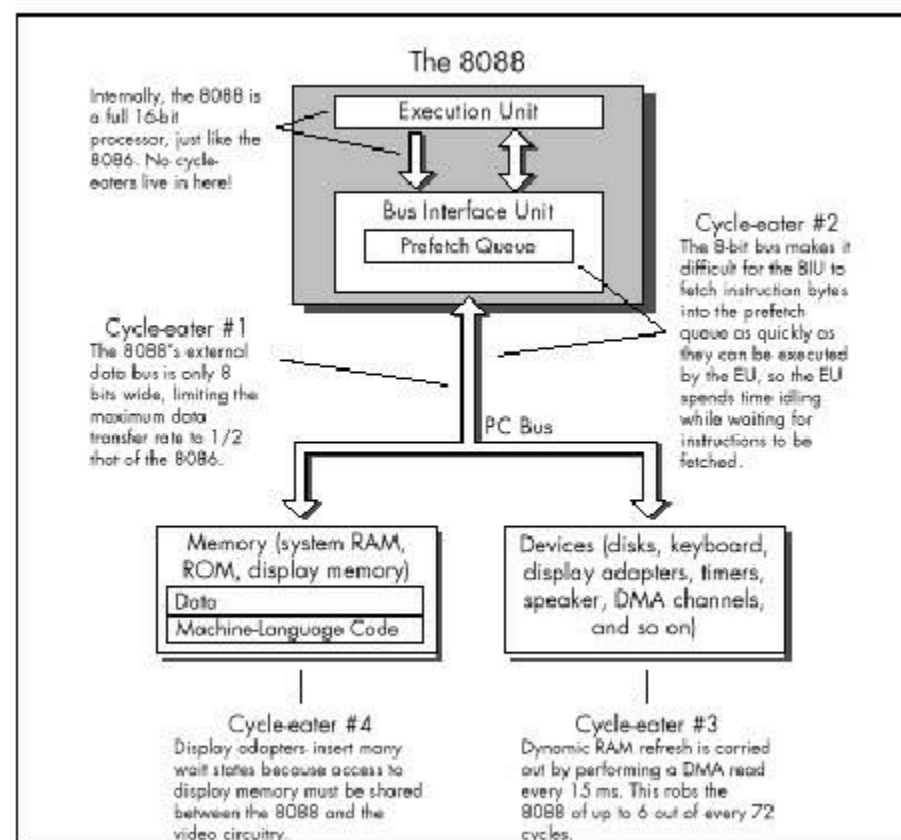


Рисунок 4.1 Расположение основных пожирателей циклов в IBM PC.

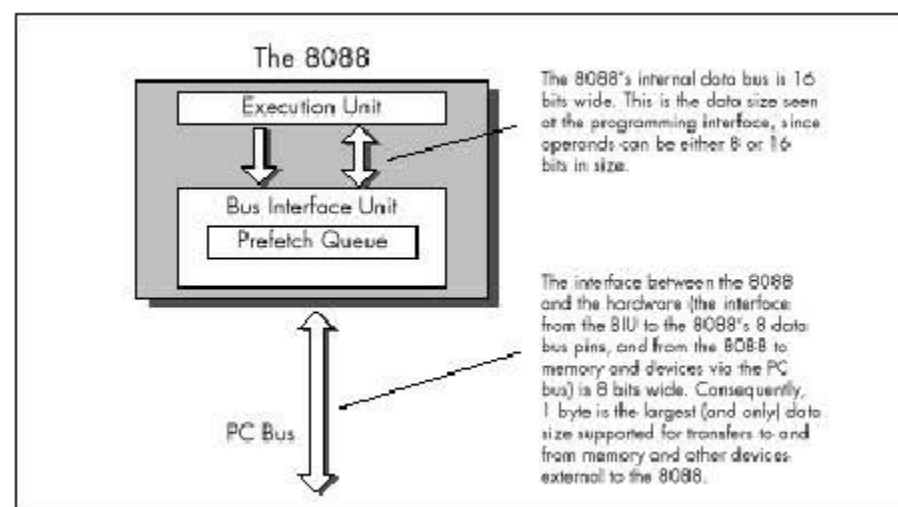


Рисунок 4.2 Ширина внутренней шины данных 8088.

Как показано на рисунке 4.1, 8-разрядная шина управления циклами расположена прямо на внешней шине 8088 шина данных. Технически, возможно, было бы точнее разместить этот пожиратель циклов в шине Интерфейсный блок, который разбивает 16-разрядные обращения к памяти на парные 8-разрядные обращения, но это на самом деле ограниченная ширина внешней шины данных, которая ограничивает поток данных внутрь и из 8088. Правда, ширина шины оригинального ПК также составляет всего 8 бит, но это только для того, чтобы соответствовать 8-разрядная шина 8088; даже если ширина шины ПК составляла 16 бит, данные все равно могли передаваться в сам чип 8088 и из него только 1 байт за раз.

Каждый доступ к шине с помощью 8088 занимает 4 тактовых цикла, или 0,838 мкс в ПК с частотой 4,77 МГц, и передает 1 байт. Это означает, что максимальная скорость, с которой могут передаваться данные в 8088 и из него, составляет 1 байт каждые 0,838 us. В то время как доступ к шине 8086 также занимает 4 такты, каждый доступ к шине 8086 может передавать либо 1 байт, либо 1 слово, что составляет максимум скорость передачи 1 слова каждые 0,838 us. Следовательно, для доступа к памяти размером со слово, эффективная скорость передачи данных 8086 составляет 1 байт на каждые 0,419 us. Напротив, каждый доступ размером со слово на 8088 требует двух обращений к шине продолжительностью в 4 цикла, одного для

старшего байта слова и одного для младшего байта слова. В результате 8088 имеет эффективную скорость передачи данных для обращений к памяти размером со слово - всего 1 слово каждые 1,676 мкс— и это, в двух словах, 8-разрядный пожиратель циклов шины.

Аналогичный пожиратель циклов скрывается под чипом 386SX, который является 32-разрядным процессором внутренне с 16-битным путем к системной памяти. Цифры разные, но способ работы пожирателя циклов точно такой же. АТ-совместимые системы имеют 16-разрядный интерфейс шины данных, по которым можно получить доступ к полному 16-разрядному слову за раз. 386SX может обрабатывать 32 однако биты (двойное слово) за раз и теряется много времени на извлечение этого двойного слова из памяти двумя половинками.

Влияние 8-разрядной шины, поглощающей циклы работы

Одним из очевидных эффектов 8-разрядной шины, поглощающей циклы работы, является то, что доступ к памяти размером со слово операнды на 8088 занимают на 4 цикла больше времени, чем обращения размером в байт. Вот почему официальные тайминги инструкций указывают, что для кода, работающего на 8088, требуется дополнительно 4 циклы требуются для каждого обращения к операнду памяти размером со слово. Например,,

```
mov ax, word ptr [MemVar]
```

чтение слова по адресу MemVar занимает на 4 цикла больше времени, чем

```
перемещение всего, байт ptr [MemVar]
```

требуется для чтения байта по адресу memvar. (На самом деле, разницы между ними нет скорее всего, это будет ровно 4 цикла, по причинам, которые станут понятны после обсуждения пожиратели циклов предварительной выборки и динамического обновления оперативной памяти позже в этой главе.)

Более того, в некоторых случаях одна команда может выполнять несколько обращений размером с слово, что влечет за собой штраф в 4 цикла при каждом обращении. Например, добавление значения к переменной памяти размером со слово требуется два обращения размером со слово — одно для чтения назначения операнд из памяти перед добавлением к нему, и один для записи результата добавления обратно в конечный операнд — и, таким образом, налагается не один, а два штрафа за 4 цикла. Как в результате

```
>add word ptr [MemVar],ax
```

<выполнение занимает примерно на 8 циклов больше времени, чем:

```
добавить байт ptr [MemVar], al
```

Строковые инструкции могут страдать от 8-битного пожирателя циклов шины в большей степени, чем другие Инструкции. Хотите верьте, хотите нет, но одна ПОВТОРНАЯ инструкция MOVSW может потерять столько же, доступ к памяти размером 131 070 слов осуществляется за 4 цикла, или 524 280 циклов к 8-разрядной шине пожиратель циклов! Другими словами, одна инструкция 8088 (по общему признанию, инструкция, которая выполняет очень много) может занять на одну десятую секунды больше времени на 8088, чем на 8086, просто из-за 8-разрядной шины. Одна десятая секунды! Это феноменально долго время в компьютерных терминах; за одну десятую секунды 8088 может выполнить более 50 000 сложений и вычитаний.

Результатом всего этого является то, что 8088 может передавать данные размером с слово в память и из нее скорость памяти лишь вдвое ниже, чем у 8086, что неизбежно снижает производительность проблемы в сочетании с исполнительным модулем, который может обрабатывать данные размером со слово каждый бит так же быстро, как 8086. Эти проблемы возникают с любым кодом, использующим данные размером со слово операнды памяти. Что еще более зловеще, как мы вскоре увидим, 8-битный пожиратель циклов шины может вызывать проблемы с производительностью и при других типах кода.

Что делать с 8-битным потребителем циклов шины?

Очевидным следствием использования 8-битного потребителя циклов шины является то, что переменные памяти размером в байт следует использовать, когда это возможно. В конце концов, 8088 использует память размером в байт.

доступ осуществляется так же быстро, как и 8086. Например, листинг 4.1, в котором используется память размером в байт. переменная памяти в виде счетчика циклов выполняется за 10,03 секунды на цикл. Это на 20 процентов быстрее чем 12.05 МКС на время выполнения петли в листинге 4.2, который использует слово размеров счетчик. Почему разница в времени выполнения? Просто потому, что каждое слово размеров декабрия выполняет обращения к памяти размером в 4 байта (два для чтения операнда размером в слово и два для запишите результат обратно в память), в то время как каждый DEC размером в байт выполняет операции размером только в 2 байта всего выполняется доступ к памяти.

ЛИСТИНГ 4.1 LST4-1.ASM

```
>; Measures the performance of a loop which uses a
; byte-sized memory variable as the loop counter.
;
        jmp Skip
;
Counter db 100
;
Skip:
        call ZTimerOn
LoopTop:
        dec [Counter]
        jnz LoopTop
        call ZTimerOff
```

ЛИСТИНГ 4.2 LST4-2.ASM

```
>; Measures the performance of a loop
which uses a ; word-sized memory
variable as the loop counter. ;
        jmp Skip
;
Counter
dw 100 ;
Skip:
        call ZTimerOn
LoopTop:
        dec [Counter]
        jnz LoopTop
        call ZTimerOff
```

Я хотел бы сделать небольшое замечание относительно оптимизации кода в списках в этой книге. На протяжении всей этой книги я моделировал пример кода после рабочего кода, чтобы результаты синхронизации были применимы к реальному программированию. В листингах 4.1 и 4.2 для например, я мог бы продемонстрировать еще большее преимущество для операндов размером в байт, просто выполнив 1000 инструкций DEC подряд, вообще без ветвления. Однако DEC инструкции не существуют в вакууме, поэтому в списках я использовал код, который одновременно уменьшал счетчик и тестировал результат. Разница в том, что между уменьшением ячейка памяти (просто инструкция) и использование счетчика циклов (функциональная последовательность инструкций). Если вы встретите код в этой книге, который покажется вам менее чем оптимальный, это просто из-за моего желания предоставить код, имеющий отношение к реальному программированию проблемы. С другой стороны, оптимальный код действительно неуловимая вещь; ни в коем случае если вы считаете, что код в этой книге идеален! Изучите его, подвергайте это сомнению и улучшайте это, поскольку пытливый, скептический ум - важная часть дзен оптимизация сборки.

Вернемся к 8-битному потребителю циклов шины. Как я уже говорил, в работе с 8088 вы должны стремиться использовать переменные памяти размером в байт, когда это возможно. Это не означает, что вам следует использовать обращения к памяти размером в 2 байта для управления переменной памяти размером в слово в предпочтении обращению к памяти размером в 1 слово, как, например,

```
перемещение d1, байт ptr [MemVar]
перемещение dh, байт ptr [MemVar+1]
```

по сравнению с:

```
перемещение dx, слово ptr [MemVar]
```

Напомним, что каждый доступ к байту памяти занимает не менее 4 циклов; это ограничение встроено прямо в 8088. 8088 также построен таким образом, что доступ к памяти размером во второй байт для 16-разрядной переменной памяти требуется только эти 4 цикла и не более. Вы никак не

может быстрее манипулировать вторым байтом переменной памяти размером со слово за секунду отдельная инструкция размером с байт менее чем за 4 цикла. На самом деле, вы обязаны доступ к этому второму байту намного медленнее с помощью отдельной инструкции, благодаря накладным расходам на выборку и выполнение команды, вычисление адреса и тому подобное.

Например, рассмотрим листинг 4.3, который выполняет чтение из памяти размером в 1000 слов. Этот код выполняется за 3,77 мкс на каждое слово, считанное на частоте 8088 4,77 МГц. Это на 45 процентов быстрее, чем чтение за 5,49 мкс на слово из листинга 4.4, который считывает те же 1000 слов, что и Чтение в листинге 4.3 выполняется за 2000 байт. Оба списка выполняют точно такое же одинаковое количество обращений к памяти — 2000 обращений, каждый размером в байт, как и все 8088 обращений должно быть количество обращений к памяти. (Помните, что устройство интерфейса шины должно выполнить два обращения к памяти размером в байт, чтобы обработать операнд памяти размером в слово.) Однако листинг 4.3 работает значительно быстрее, поскольку затрачивает всего 4 дополнительных цикла для чтения второго байта каждого слова, в то время как листинг 4.4 выполняет второй LODSB, требуется 13 циклов для чтения второго байта каждого слова.

ЛИСТИНГ 4.3 LST4-3.ASM

```
>; Measures the performance of reading 1,000 words
; from memory with 1,000 word-sized accesses.
    sub si,si
    mov cx,1000
    call ZTimerOn
    rep lodsw
    call ZTimerOff
```

<ЛИСТИНГ 4.4 LST4-4.ASM

```
>; Measures the performance of reading 1000 words
; from memory with 2,000
byte-sized accesses. ;
    sub si,si
    mov cx, 2000
    call ZTimerOn
    rep lodsb
    call ZTimerOff
```

<Короче говоря, если вам необходимо выполнить 16-битный доступ к памяти, позвольте 8088 разделить доступ на для вас доступы размером в два байта. 8088 справляется с этой задачей эффективнее, чем ваш код возможно.

Переменные размером со слово должны храниться в регистрах в максимально возможной степени, поскольку регистры находятся внутри 8088, где 16-битные операции выполняются так же быстро, как и 8-битные потому что 8-битный пожиратель циклов не может до них добраться. На самом деле, это хорошая идея - хранить как можно больше переменных всех видов в регистрах. Инструкции с операндами только для регистра выполняются очень быстро, частично потому, что они позволяют избежать как трудоемких обращений к памяти, так и длительных вычислений адресов, связанных с операндами памяти.

Есть еще одна причина, по которой регистровые операнды предпочтительнее операндов памяти, и это неожиданный эффект 8-битного пожирателя циклов шины. Инструкции только с регистром операнды, как правило, короче (в байтах), чем инструкции с операндами памяти, и когда дело доходит до производительности, чем короче, тем обычно лучше. Чтобы объяснить, почему это правда и как это соотносится с 8-битным потребителем циклов шины, я должен на мгновение отклониться.

На последних нескольких страницах вы, возможно, думали, что 8-разрядная шина потребляет много циклов, хотя это и неприятно, но не кажется особенно тонким или сложным для количественной оценки. В конце концов, любая ссылка на инструкцию точно сообщает нам, сколько циклов каждая команда проигрывает 8-битной пожиратель циклов шины, не так ли?

И да, и нет. Это правда, что в общем случае мы приблизительно знаем, сколько времени потребуется для выполнения данной инструкции с операндом в памяти размером со слово, чем с операндом размером с байт операнд, хотя пожиратели циклов динамического обновления ОЗУ и состояния ожидания (о которых я расскажу немного позже) могут значительно повысить стоимость 8-разрядного пожирателя циклов шины. Однако все обращения к памяти размером со слово теряют 4 цикла из-за 8-разрядного пожирателя циклов шины, и есть один

вид доступа к памяти размером со слово, который мы еще не обсуждали: выборка инструкций. В самом уродливом проявлении 8-битного пожирателя циклов шины на самом деле является пожиратель циклов очереди предварительной выборки .

Пожиратель циклов очереди предварительной выборки

В контексте 8088, вот в двух словах пожиратель циклов очереди предварительной выборки: 8-разрядный 8088 внешняя шина данных не позволяет устройству интерфейса шины извлекать байты инструкций так быстро, как только это возможно 16-разрядное исполнительное устройство может их выполнять, поэтому исполнительное устройство часто простаивает, пока ожидание получения следующего байта инструкции.

Почему именно это происходит? Напомним, что 8088 внутренне является 8086, но обращается к передача данных в память размером с слово с вдвое меньшей скоростью, чем у 8086, благодаря 8-разрядной внешней шине данных 8088. К сожалению, инструкции размером со слово данные, которые извлекает 8086, означают, что 8088 может извлекать инструкции только с половиной скорости 8086. С другой стороны, исполнительный модуль 8088, эквивалентный 8086, может выполнять инструкции так же быстро, как и 8086. Конечным результатом является то, что Исполнительный блок сжигает байты инструкций намного быстрее, чем может модуль интерфейса шины извлекает их и в конечном итоге работает на холостом ходу, ожидая поступления байтов инструкций.

BIU может извлекать байты инструкций с максимальной скоростью в один байт каждые 4 цикла — и *эта скорость в 4 цикла на байт инструкции является предельным ограничением для всей инструкции* время выполнения, независимо от скорости ЕС. Хотя ЕС может выполнить данную инструкцию которая уже находится в очереди предварительной выборки менее чем за 4 цикла на байт, со временем ЕС не может выполнять инструкции быстрее, чем они могут поступить — а они не могут поступать быстрее чем на 1 байт каждые 4 цикла.

Очевидно, тогда, пожиратель циклов очереди предварительной выборки - это не что иное, как один из аспектов 8-битной пожиратель циклов шины. Код 8088 часто выполняется со скоростью, меньшей максимальной для модуля выполнения скорость, потому что 8-разрядная шина данных не справляется со спросом на байты инструкций. Это достаточно просто — так к чему вся эта суeta с циклом очереди предварительной выборки - пожиратель?

Сложность пожирателя циклов очереди предварительной выборки заключается в том, что он недокументирован и непредсказуем. То есть при доступе к памяти размером со слово, таком как

```
>mov [bx],ax
```

<это хорошо задокументировано, что дополнительно 4 циклов всегда будут обязаны оставить верхнюю байт АХ в памяти. Что не так с циклом-людоед очереди предварительной выборки, притаившегося неподалеку. Для например, инструкции

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

<должны выполняться за 10 циклов, поскольку для выполнения каждого SHR требуется 2 цикла, согласно Спецификациям Intel. Эти спецификации содержат официальные инструкции Intel по выполнению время, но в этом случае — и во многих других — спецификации кардинально неверны. Почему? Потому что они описывают время выполнения после того, как команда достигает предварительной выборки очередь. Они ничего не говорят о том, будет ли данная команда находиться в очереди предварительной выборки когда пришло время для выполнения этой инструкции или сколько времени потребуется для достижения этой инструкции очередь предварительной выборки, если ее там еще нет. Благодаря низкой производительности 8088 внешняя шина передачи данных — это вопиющее упущение, но, увы, неизбежное. Давайте посмотрим на почему официальные сроки выполнения неверны и почему с этим ничего не поделаешь.

Официальные сроки выполнения - это только часть истории.

Последовательность из 5 инструкций SHR в последнем примере имеет длину 10 байт. Это означает

что он никогда не сможет выполняться менее чем за 24 цикла, даже если 4-байтовая очередь предварительной выборки заполнена когда это начнется, поскольку для выборки все равно останется 6 байт инструкций, по 4 цикла на выборка. Если очередь предварительной выборки пуста в начале, последовательность может занять 40 циклов. короче говоря, благодаря выборке инструкций код не будет выполняться с задокументированной скоростью и может занять в четыре раза больше времени, чем предполагается.

Почему Intel документирует время выполнения модуля выполнения, а не общую инструкцию время выполнения, которое включает как время выборки команды, так и единицу выполнения (EU) время выполнения? Что ж, выборка команд не выполняется как часть выполнения команд исполнительным модулем, а вместо этого выполняется параллельно модулем интерфейса шины (BIU) всякий раз, когда внешняя шина данных не используется или когда у EU заканчивается инструкция байты для выполнения. Иногда BIU может использовать запасные циклы шины для предварительной выборки байты инструкций до того, как они понадобятся ЕС, поэтому в таких случаях выборка инструкций занимает практически вообще никакого времени. В других случаях ЕС выполняет инструкции быстрее чем БИУ сможет их достать, и тогда выборка инструкций становится значительной частью *общего времени выполнения. В результате эффективное время выборки для данной инструкции варьируется сильно зависит от набора кода, предшествующего этой инструкции. Аналогично, состояние, в котором данная команда покидает очередь предварительной выборки, влияет на общее время выполнения следующих инструкций.*



Другими словами, хотя время выполнения для данной команды является постоянным, время выборки для этой команды сильно зависит от контекст, в котором выполняется команда — объем предварительной выборки разрешенный предыдущими командами — и может варьироваться от полных 4 циклов на байт команды до отсутствия времени вообще.

Как мы увидим позже, другие потребляющие цикл состояния, такие как обновление DRAM и ожидание отображения памяти, могут вызывать различия в предварительной выборке даже при разных исполнениях одного и того же кодовая последовательность. Учитывая это, бессмысленно говорить о времени предварительной выборки для данной инструкции, кроме как в контексте конкретной кодовой последовательности.

Итак, теперь вы знаете, почему официальные сроки выполнения инструкций часто неверны и почему Intel не может предоставить лучшие спецификации. Теперь вы также знаете, почему вам необходимо время ваш код, если вы хотите знать, насколько он быстр на самом деле.

Не существует такого чудовища, как истинное время выполнения инструкции

Влияние кода, предшествующего инструкции, на время выполнения этой инструкции делает использование Zen timer более сложным, чем вы могли ожидать, и усложняет интерпретацию результатов, сообщаемых Zen timer. С одной стороны, Zen timer - это лучше всего использовать для синхронизации кодовых последовательностей длиной более нескольких инструкций; ниже 10 с или около того, эффекты очереди предварительной выборки и ограниченное разрешение часов, управляющие таймер может вызвать проблемы.

Небольшая неточность, вызванная предварительной выборкой из очереди, обычно существует даже при использовании таймера Zen используется для определения времени более длительных последовательностей кода, поскольку вызовы таймера Zen обычно изменяют очередь предварительной выборки кода из ее нормального состояния. (Переходы —переходы, вызовы, возвраты и т. П. нравится —опорожняйте очередь предварительной выборки.) В идеале таймер Zen используется для измерения производительность всей подпрограммы, поэтому эффекты очереди предварительной выборки ветвей в начало и завершение подпрограммы аналогичны эффектам вызовов Zen timer когда вы измеряете производительность подпрограммы.

Другой способ, с помощью которого пожиратель циклов очереди предварительной выборки усложняет использование Zen таймер включает в себя практику многократного синхронизации выполнения нескольких инструкций. Я часто буду повторять одну или две инструкции 100 или 1000 раз подряд в списках в этой книге чтобы получить временные интервалы, достаточные для проведения надежных измерений.

Однако, как мы только что узнали, фактическая производительность любой инструкции 8088 зависит

от набора кода, предшествующего любому данному использованию этой инструкции, что, в свою очередь, влияет на состояние очереди предварительной выборки, когда команда начинает выполняться. Увы, выполнение время выполнения инструкции, которой предшествуют десятки идентичных инструкций, отражает только одну из множество возможных состояний предварительной выборки (и при этом не очень вероятных), а также некоторые другие состояния предварительной выборки вполне могут давать совершенно разные результаты.

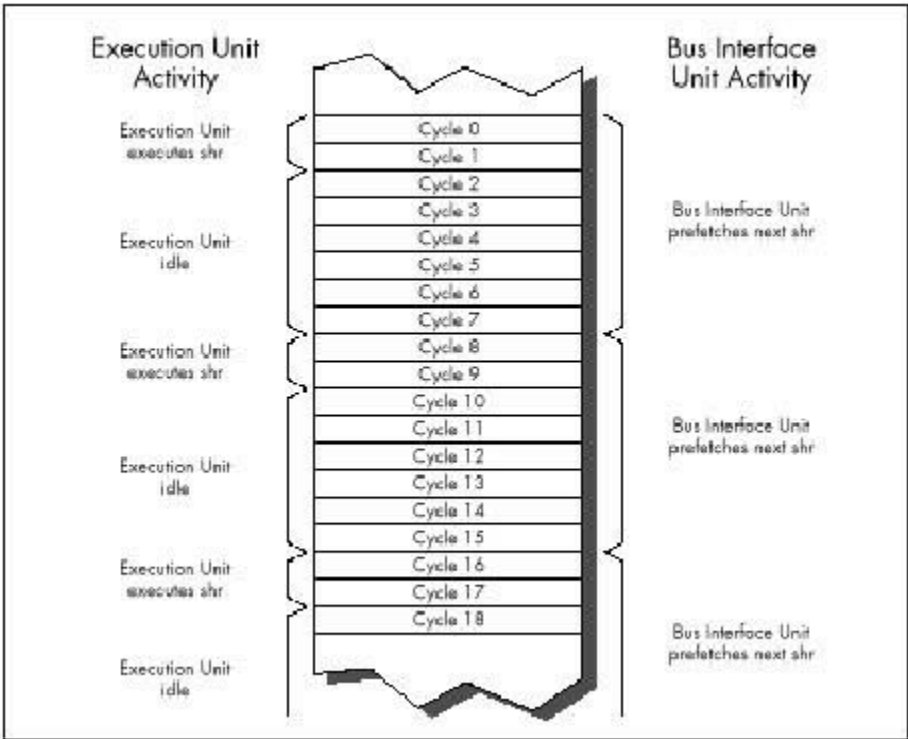
Например, рассмотрим код в листингах 4.5 и 4.6. В листинге 4.5 показан наш знакомый Пример SHR. Здесь, поскольку очередь предварительной выборки всегда пуста, время выполнения должно рассчитайте примерно 4 цикла на байт, или 8 циклов на SHR, как показано на рисунке 4.3. (Рисунок 4.3 иллюстрирует взаимосвязь между выборкой команд и выполнением в упрощенном виде образом, и не предназначен для отображения точного времени выполнения 8088 операций.) Это довольно сильно отличается от официального времени выполнения SHR за 2 цикла. Фактически, Zen timer сообщает, что Листинг 4.5 выполняется за 1,81 мкс на байт, или чуть более 4 циклов на байт. (дополнительное время является результатом динамического пожирателя циклов обновления оперативной памяти, о котором мы вскоре поговорим .) Исходя из листинга 4.5, мы пришли бы к выводу, что “истинное” время выполнения SHR составляет 8,64 цикла.

ЛИСТИНГ 4.5 LST4-5.ASM

```
>; Measures the performance of 1,000 SHR
instructions ; in a row. Since SHR
executes in 2 cycles but is ; 2 bytes
long, the prefetch queue is always
empty, ; and prefetching time determines
the overall ; performance of the code. ;
call ZTimerOn
rept 1000
  shrax,1
endm
call ZTimerOff
```

<ЛИСТИНГ 4.6 LST4-6.ASM

```
>; Measures the performance of 1,000 MUL/SHR instruction
; pairs in a row. The lengthy execution time of MUL
; should keep the prefetch queue from ever emptying.
movex,1000
subax,ax
call ZTimerOn
rept 1000
  mulax _
  shrax,1
endm
call ZTimerOff
```



<Рисунок 4.3 Последовательность выполнения и предварительной выборки команд для листинга 4.5.

Теперь давайте рассмотрим листинг 4.6. Здесь каждый SHR следует инструкции MUL. Поскольку MUL выполнение инструкций занимает так много времени, что очередь предварительной выборки всегда заполнена, когда они завершите, каждый SHR должен быть готов и ожидать в очереди предварительной выборки, когда предыдущий MUL завершится. В результате мы ожидаем, что каждый SHR будет выполняться за 2 цикла; вместе с учетом 118-тактного времени выполнения при умножении 0 на 0, общее время выполнения должно доходить до 120 циклов на пару SHR / MUL, как показано на рис. 4.4. И, клянусь Богом, когда мы запускаем

В листинге 4.6 мы получаем время выполнения 25,14 мкс на пару SHR / MUL, или ровно 120 циклов! Согласно этим результатам, “истинное” время выполнения SHR, по-видимому, составляет 2 цикла, что значительно отличается от вывода, который мы сделали из листинга 4.5.

Ключевой момент заключается в следующем: мы видели одну кодовую последовательность, для выполнения которой SHR потребовалось более 8 циклов, и другую, для которой потребовалось всего 2 цикла. Мы говорим о двух разных формах SHR здесь? Конечно, нет — разница является чисто отражением различий состояния, в которых предыдущий код покинул очередь предварительной выборки. В листинге 4.5, каждый последующий SHR за первыми несколькими следуют множество других инструкций SHR, которые не прошли предварительную выборку очередь иссякла, поэтому общая производительность отражает время выборки команд. Напротив, каждый SHR в листинге 4.6 следует инструкции MUL, которая оставляет очередь предварительной выборки заполненной, поэтому в целом производительность отражает время выполнения единицы выполнения.

Очевидно, что либо время выборки команды, либо время выполнения единицы выполнения — или даже сочетание того и другого, если команда частично предварительно выбрана, — может определять производительность кода. Некоторые люди руководствуются эмпирическим правилом, согласно которому они предполагают, что время выполнения каждой инструкции в 4 цикла умножено на количество байтов в инструкции. Пока это часто это верно для кода только для регистра, но часто не выполняется для кода, который обращается к памяти. Во-первых, правило должно быть в 4 цикла больше объема памяти доступы, а не байты инструкций, поскольку все обращения занимают 4 цикла на ПК на базе 8088. С другой стороны, инструкции доступа к памяти часто выполняются медленнее в исполнительном модуле раз больше, чем требовало бы правило 4 циклов на доступ к памяти, потому что 8088 не очень быстр при вычислении адресов памяти. Кроме того, правило 4 циклов на байт инструкции неверно для инструкций только для регистрации, которые уже находятся в очереди предварительной выборки, когда предыдущая инструкция завершается.

Правда в том, что уменьшение количества циклов или байтов никогда не повредит производительности количество данного бита кода, но нет никакой гарантии, что то или иное улучшится производительности тоже. Например, рассмотрим листинг 4.7, который состоит из серии из 4- цикл, 2-байтовый движок, 0 инструкций, который выполняется со скоростью 1,81 мкс за инструкция. Теперь рассмотрим листинг 4.8, в котором четырехтактное перемещение AL,0 заменяется трехтактным cycle (но все еще 2-байтовый) SUB AL, AL, Несмотря на преимущество в 1 цикл на инструкцию, перечисляет 4.8 выполняется точно с той же скоростью, что и в листинге 4.7. Причина: обе инструкции состоят из 2 байт, и в обоих случаях это время выборки команды за 8 циклов, а не за 3 или 4 цикла Время выполнения модуля выполнения ограничивает производительность.

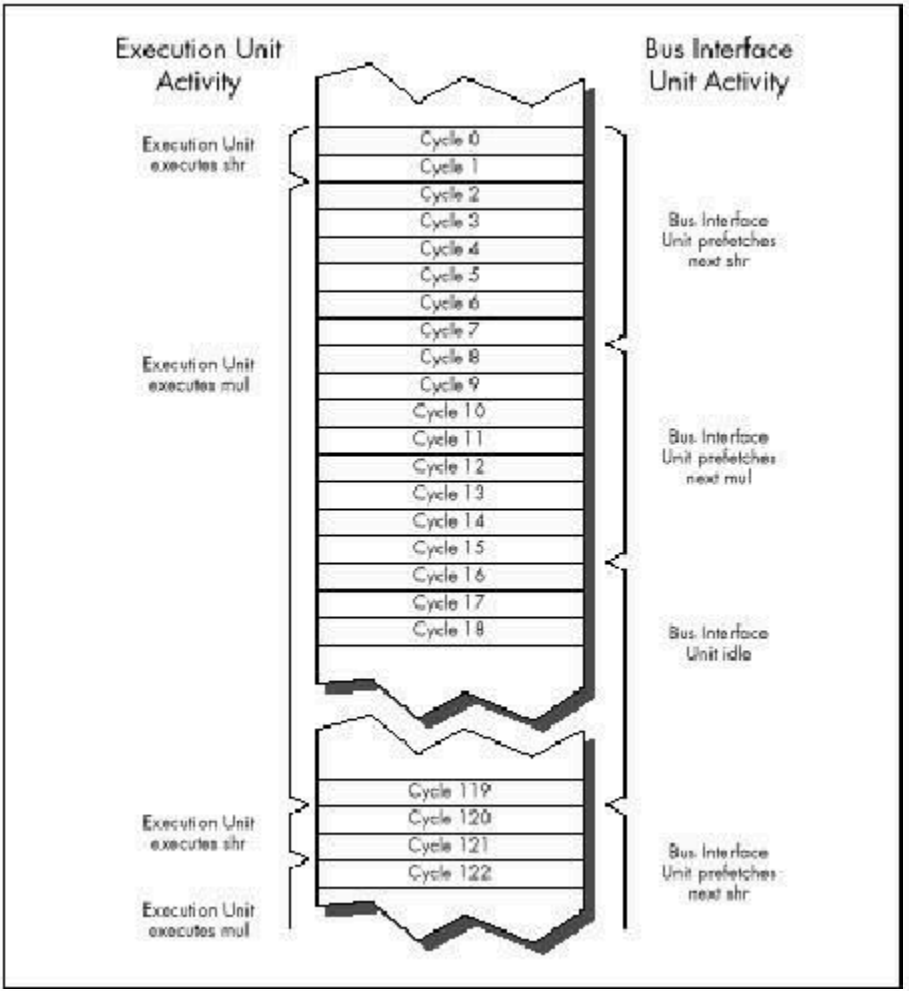


Рисунок 4.4 Последовательность выполнения и предварительной выборки инструкций для листинга 4.6.

ЛИСТИНГ 4.7 LST4-7.ASM

```
>; Measures the performance of repeated MOV AL,0 instructions,  
; which take 4 cycles each according to Intel's official
```

```
<>; specifications.
    sub ax,ax
    call ZTimerOn
    rept 1000
    mov al,0
    endm
    call ZTimerOff
```

<ЛИСТИНГ 4.8 LST4-8.ASM

```
>; Measures the performance of repeated SUB AL,AL instructions,
; which take 3 cycles each according to
Intel's official ; specifications.
;
    sub ax,ax
    call ZTimerOn
    rept 1000
    sub al,al
    endm
    call ZTimerOff
```

<Как вы можете видеть, легко заставить себя думать, что вы экономите циклы, когда это не так. Повысить производительность определенного фрагмента кода можно только за счет уменьшения коэффициента— либо время получения инструкции, либо время выполнения, или иногда сочетание того и другого — это ограничивает производительность этого кода.

На случай, если вы пропустили это в суматохе, вариативность предварительной выборки означает, что наш метод тестирования производительности путем выполнения 1000 инструкций подряд никоим образом не “истинное” время выполнения инструкций не превышает официального времени выполнения в руководства Intel соответствуют “истинному” времени. Дело в том, что данная инструкция требует время выполнения по крайней мере такое же, как указано для этого в руководствах Intel, но может занять на целых 4 цикла на байт больше, в зависимости от состояния очереди предварительной выборки, когда предыдущая команда завершается.



Единственное истинное время выполнения инструкции - это время, измеряемое в определенный контекст, и это время имеет значение только в этом контексте.

Что мы действительно хотим, так это знать, сколько времени требуется для выполнения полезного рабочего кода, а не сколько требуется всего одна инструкция, и Zen timer дает нам инструмент, необходимый для сбора этой Информация. Конечно, было бы проще, если бы мы могли просто сложить аккуратно задокументированные время выполнения инструкций — но этого не произойдет. Без фактического измерения производительности данной кодовой последовательности вы просто не знаете, насколько она быстра. Для поразительно, но даже люди, разработавшие 8088 в Intel, не смогли бы точно сказать вам, как быстро данная последовательность кодов 8088 выполняется на ПК, просто взглянув на нее! Получите привыкнув к идее, что время выполнения имеет значение только в контексте, изучите правила ознакомьтесь с этой книгой и используйте Zen timer для измерения вашего кода.

Приблизительное общее время выполнения.

Не думайте, что, поскольку общее время выполнения команды определяется как временем выборки команды, так и временем выполнения модуля выполнения, эти два времени следует суммировать при оценке производительности. Например, с практической точки зрения, для выполнения каждого SHR в Листинге 4.5 не требуется 8 циклов времени выборки команды плюс 2 цикла выполнения Время выполнения единицы. На рисунке 4.3 показано, что во время выполнения данного SHR начинается выборка следующего SHR, и поскольку две операции перекрываются на 2 циклов, нет смысла указывать время для обеих инструкций. Вы могли бы представить себе, что дополнительное время выборки инструкции для SHR в листинге 4.5 равно 6 циклам, что дает общее время выполнения 8 циклов при добавлении к 2 циклам единицы выполнения время выполнения.

В качестве альтернативы, вы могли бы представить, что для извлечения каждого SHR в листинге 4.5 требуется 8 циклов, и затем выполняется фактически за 0 циклов, пока извлекается следующий SHR. В зависимости от того, что перспектива, которую вы предпочитаете, подойдет. Важным моментом является то, что время, в течение которого выполняется

выполнение одной инструкции и выборка следующей команды должны перекрываться только учитываться при подсчете общего времени выполнения одной из инструкций. В любом случае и для этих целей одна из двух инструкций выполняется без каких-либо затрат производительности, пока существует дублирование.

В качестве рабочего определения, мы рассмотрим время выполнения данной инструкции в определенном контексте, который запускается, когда первый байт инструкции отправляется на выполнение. Блокируется и завершается, когда первый байт следующей инструкции отправляется в ЕС.

Что делать с пожирателем циклов очереди предварительной выборки?

Уменьшение влияния пожирателя циклов очереди предварительной выборки является одним из важнейших принципов высокопроизводительного ассемблерного кода. Как вы можете это сделать? Одним из эффективных методов является минимизация доступа к операндам памяти, поскольку такие обращения конкурируют с инструкциями выборки для доступа к драгоценной памяти. Вы также можете значительно сократить время получения инструкций просто выбрав инструкции: делайте их краткими. Чем меньше времени, тем лучше требуется для выборки инструкций длиной 1 или 2 байта по сравнению с инструкциями длиной 5 или 6 байта. Сокращенная выборка инструкций снижает минимальное время выполнения (минимум время выполнения в 4 цикла превышает количество байтов инструкции) и часто приводит к более быстрому общему выполнению.

Хотя короткие инструкции сводят к минимуму общее время предварительной выборки, по иронии судьбы они на самом деле часто от узкого места в очереди предварительной выборки страдают больше, чем от длинных инструкций. Короткие инструкции обычно выполняются так быстро, что они истощают очередь предварительной выборки несмотря на их небольшой размер. Например, рассмотрим SHR из листинга 4.5, который выполняется всего за 25 процентов времени выполнения его модуля выполнения, хотя его длина составляет всего 2 байта, благодаря узкому месту в очереди предварительной выборки. Тем не менее, короткие инструкции, как правило, быстрее, чем длинные инструкции, благодаря сочетанию меньшего количества байтов инструкций и более быстрого времени выполнения исполнительных модулей, и их следует использовать как можно чаще — просто не ожидайте, что они будут работать на “официальной” задокументированной скорости.

Прежде всего, приведенные выше правила подразумевают максимально активное использование регистров, как потому, что инструкции только для регистров короткие, так и потому, что они не выполняются доступ к памяти осуществляется для чтения или записи операндов. Однако использование регистров - это правило эмпирический опыт, а не заповедь. При некоторых обстоятельствах доступ к ним действительно может быть более быстрым память. (Метод справочной таблицы - один из таких случаев.) Более того, производительность структура очереди предварительной выборки (и, следовательно, производительность каждой инструкции) отличается от одной кодовая последовательность к следующей и может даже отличаться при разных исполнениях одной и той же команды кодовая последовательность.

В целом, написание хорошего кода на ассемблере - это такое же искусство, как и наука. В результате вы следуете следовать эмпирическим правилам, описанным здесь, а затем засечь время работы вашего кода, чтобы увидеть, насколько это действительно быстро. Вы должны свободно экспериментировать, но всегда помните, что фактический, измеренная производительность - вот итог.

Держа в руках 8088.

В этой главе я все дальше и дальше уводил вас в глубины ПК, рассказывая вам снова и снова повторяю, что вы должны понимать компьютер на минимально возможном уровне, чтобы писать хороший код. На этом этапе вы вполне можете задаться вопросом “Неужели мы опустили достаточно?”

Не совсем еще. Пожиратели циклов 8-разрядной шины и очереди предварительной выборки действительно низкоуровневые, но нам еще предстоит пройти один уровень. Динамическое обновление оперативной памяти и состояния ожидания — наши следующие темы — вместе они образуют самый низкий уровень, на котором аппаратное обеспечение ПК влияет на код Производительность. Ниже этого уровня ПК представляет интерес только для инженеров-аппаратчиков.

Прежде чем мы начнем обсуждение динамического обновления оперативной памяти, давайте на мгновение вернемся назад, чтобы

в целом взглянуть на этот самый низкий уровень потребления циклов. По правде говоря, различия между состояниями ожидания и динамическим обновлением оперативной памяти не имеют большого значения для программиста. Что такое *важно, чтобы вы понимали следующее: при определенных обстоятельствах устройства на ПК шина может останавливать процессор на 1 или более циклов, заставляя ваш код выполняться медленнее, чем это казалось бы, должно быть.*

В отличие от всех пожирателей циклов, с которыми мы сталкивались до сих пор, состояния ожидания и динамическая оперативная память обновление являются строго внешними по отношению к центральному процессору, как показано на рисунке 4.1. Адаптеры на Шины ПК, такие как видео и карты памяти, могут устанавливать состояния ожидания при любом доступе к шине, идея в том, что они не смогут завершить доступ должным образом, если доступ ограничен. Аналогично, канал контроллера DMA, выделенный для динамического обновления ОЗУ, может запросить управление шиной в любое время, хотя центральный процессор должен отключить шину до того, как контроллер DMA сможет взять управление на себя. Это означает, что ваш код не может напрямую управлять состояниями ожидания или динамическим обновлением оперативной памяти. Однако иногда код может быть спроектирован таким образом, чтобы минимизировать влияние этих пожирателей циклов, и даже когда они пожиратели циклов замедляют ваш код, и вы ничего не можете с этим поделать, вам все равно лучше понимать, что вы теряете производительность, и знать почему ваш код выполняется не так быстро, как предполагалось, чем вы программировали в незнании.

Давайте начнем с обновления DRAM, которое влияет на производительность каждой программы, которая запускается на ПК.

Динамическое обновление оперативной памяти: невидимая рука

Динамическое обновление оперативной памяти (DRAM) - это своего рода Божий акт. Под этим я подразумеваю, что DRAM обновление незаметно и неумолимо крадет определенную долю всего доступного доступа к памяти время у ваших программ, когда они обращаются к памяти для получения кода и данных. (Когда они получают доступ к кэшу на более современных процессорах, теоретически обновление DRAM пожиратель циклов не вступает в игру, но есть и другие пожиратели циклов, которые ждут своей добычи программы, привязанные к кэшу.) Хотя вы могли бы остановить обновление DRAM, вам бы этого не хотелось поскольку это было бы верным способом привести к сбою вашего компьютера. В конце концов, благодаря При обновлении DRAM почти весь код на ПК выполняется немного медленнее, чем в противном случае, и на этом все.

Немного предыстории: Микросхема статической оперативной памяти (SRAM) - это микросхема памяти, которая сохраняет свое содержимое неограниченно долго, пока поддерживается питание. Напротив, каждый из нескольких блоков битов в микросхеме динамической оперативной памяти (DRAM) сохраняет свое содержимое лишь короткое время после обращения к нему для чтения или записи. Для того, чтобы получить микросхему DRAM для хранения данных в течение длительного периода, к каждому из блоков битов в этом чипе необходимо регулярно обращаться, чтобы обеспечить обновление и достоверность хранящихся в чипе данных . Пока это делается достаточно часто, память DRAM чип будет сохранять свое содержимое неопределенно долго.

Вся системная память ПК состоит из микросхем DRAM. Каждый чип DRAM в ПК должен полностью обновляться примерно раз в четыре миллисекунды, чтобы гарантировать целостность данных, которые он хранит. Очевидно, что крайне желательно, чтобы память в ПК сохраняла правильные данные на неопределенный срок, поэтому каждый чип DRAM в ПК всегда должен быть обновлен в течение 4 мс после последнего обновления. Поскольку нет гарантии, что данная программа будет обращаться к каждому блоку DRAM каждые 4 мс, ПК содержит специальные схемы и программирование для обеспечения обновления DRAM.

Как работает обновление DRAM на ПК

На оригинальном IBM PC на базе 8088 таймер 1 микросхемы таймера 8253 запрограммирован на включение питания для генерации сигнала раз в 72 цикла или раз в 15,08нс. Этот сигнал переходит на канал 0 контроллера 8237 DMA, который запрашивает шину у 8088 при получении сигнала. (DMA означает прямой доступ к памяти, способность устройства

отличный от 8088 для управления шиной и доступа к памяти напрямую, без какой-либо помощи от 8088.) Как только 8088 переключается между обращениями к памяти, он передает управление шина 8237, которая в сочетании со специальной схемой на материнской плате ПК затем выполняет одиночный 4-цикловый доступ на чтение к 1 из 256 возможных адресов, продвигаясь к следующему адресу при каждом последующем доступе. (Доступ на чтение предоставляется только для цель обновления DRAM; считываемые данные не используются.)

256 адресов, к которым обращаются при доступе refresh DMA, расположены таким образом, что взятые вместе они должным образом обновляют всю память ПК. Обращаясь к одному из 256 обращается каждые 15.08 us, вся оперативная память ПК обновляется за 256 x 15.08 us, или 3.86 мс, что примерно соответствует желаемому времени в 4 мс, о котором я упоминал ранее. (На ПК обновлены только первые 640 КБ памяти; видеоадаптеры и другие адаптеры емкостью более 640 КБ содержащие память, требующую обновления, должны обеспечивать собственное обновление DRAM в системах pre- AT.)

Не вдавайтесь здесь в подробности. Важный момент заключается в следующем: по крайней мере, в течение 4 из каждых 72 циклов шина исходного ПК переключается на обновление DRAM и недоступна для 8088, как показано на рисунке 4.5. Это означает, что теряется до 5,56% мощности ПК и без того недостаточной емкости шины. Однако обновление DRAM необязательно остановите 8088 на 4 цикла. Исполнительный блок 8088 может продолжать обработку во время обновления DRAM, если только ЕС не потребуется доступ к памяти. Следовательно, обновление DRAM может снизить производительность кода от 0% до 5,56% (а на самом деле немного больше, как мы вскоре увидим), в зависимости от степени какое обновление DRAM занимает циклы, в течение которых 8088 в противном случае был бы доступ к памяти.

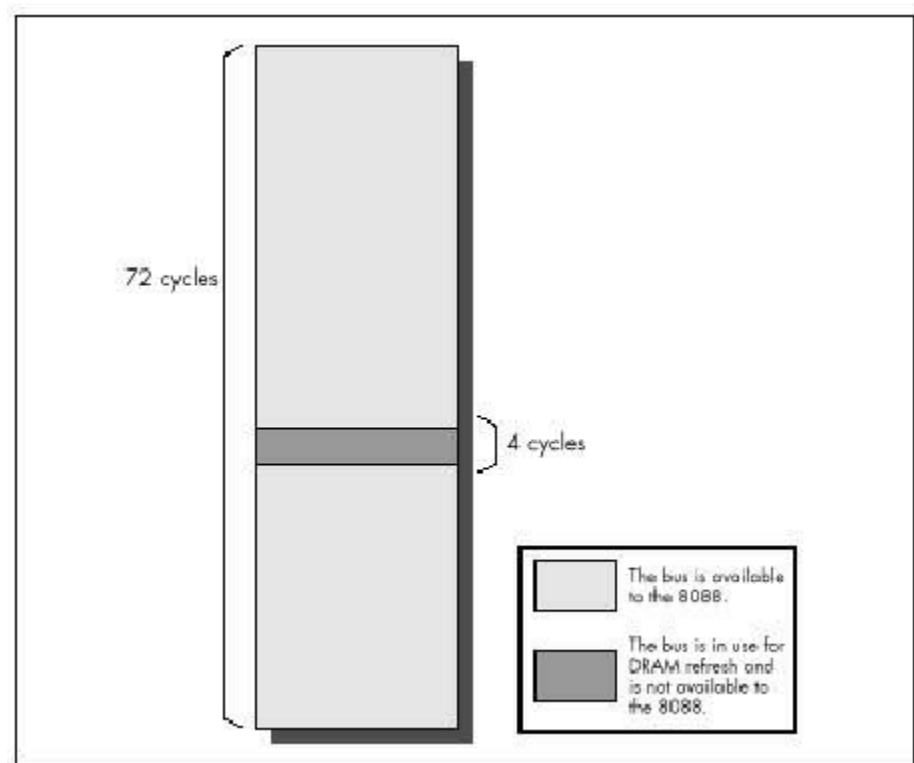


Рисунок 4.5 Обновление динамической оперативной памяти шины ПК (DRAM).

Влияние обновления DRAM

Давайте рассмотрим примеры с противоположных концов спектра с точки зрения влияния Обновление DRAM на производительность кода. Сначала рассмотрим серию инструкций MUL в Листинг 4.9. Поскольку 16-разрядный mul на 8088 выполняется между 118 и 133 циклами и имеет длину всего 2 байта, должно быть достаточно времени для заполнения очереди предварительной выборки после каждая команда, даже после того, как обновление DRAM отняло часть времени доступа к памяти. Следовательно, очередь предварительной выборки должна обеспечивать бесперебойную работу модуля выполнения постоянное снабжение байтами инструкций. Поскольку в листинге 4.9 не используются операнды памяти, исполнительному устройству никогда не должно быть необходимости ждать получения данных из памяти, а обновление DRAM не должно влиять на производительность. (Помните, что исполнительное устройство может работать обычно во время обновления DRAM, если для этого не требуется запрашивать доступ к памяти от блока интерфейса шины.)

ЛИСТИНГ 4.9 LST4-9.ASM

; Измеряет производительность повторяющихся инструкций MUL.,

```
>; which allow the prefetch queue to be full at all times,  
; to demonstrate a case in which DRAM refresh has no impact  
; on code performance.  
sub ax,ax  
call ZTimerOn  
rept 1000  
mul ax  
endm  
call ZTimerOff
```

<Запустив листинг 4.9, мы обнаруживаем, что каждый MUL выполняется за 24,72 секунды, или ровно 118 циклов. Поскольку это наименьшее время, за которое может выполняться MUL, мы видим, что при обновлении DRAM производительность не снижается . Листинг 4.9 наглядно иллюстрирует, что обновление DRAM влияет только на производительность кода при обновлении DRAM вынуждает исполнительное устройство 8088 ждать доступа к памяти.

Теперь давайте посмотрим на серию инструкций SHR, показанных в листинге 4.10. Поскольку SHR выполняется за 2 цикла, но имеет длину 2 байта, очередь предварительной выборки должна быть пустой в листинге 4.10 выполняется без остановки с байтами команды предварительной выборки 8088. В результате время на инструкция из листинга 4.10 должна точно отражать время, необходимое для выборки байты инструкций.

ЛИСТИНГ 4.10 LST4-10.ASM

```
; Измеряет производительность повторяющихся инструкций SHR,  
; которые опустошают очередь предварительной выборки, чтобы продемонстрировать  
; наихудшее влияние обновления DRAM на производительность кода.  
;  
вызовите ZTimerOn  
rept 1000  
shr ax,1  
endm  
вызовите ZTimerOff
```

Поскольку для чтения каждого байта инструкции требуется 4 цикла, мы ожидаем, что каждый SHR будет выполняться за 8 циклов, или 1,676 мкс, если не было обновления DRAM. Фактически, каждый SHR в Листинг 4.10 выполняется за 1,81 доллара США, что указывает на то, что обновление DRAM занимает 7,4 процента от времени выполнения программы. Это почти на 2 процента больше, чем в нашем наихудшем случае оценка потерь из-за накладных расходов на обновление DRAM! Фактически, результат показывает, что DRAM обновление крадет не 4, а 5,33 цикла из каждых 72 циклов. Как это может быть?

Ответ заключается в том, что данное обновление DRAM может фактически задерживать доступ к памяти процессора на целых 6 циклов, в зависимости от времени выполнения запроса DMA на обновление DRAM относительно состояния выполнения внутренней инструкции 8088. При выполнении кода из листинга 4.10 каждое обновление DRAM задерживает работу процессора на 5 или 6 циклов, в зависимости от где 8088 находится при выполнении текущей инструкции SHR при запросе обновления происходит. Теперь мы видим, что все может стать еще хуже, чем мы думали: обновление DRAM *может украсть до 8,33 процента доступного времени доступа к памяти — 6 из каждых 72 циклов — у 8088.*

Какой из двух рассмотренных нами случаев отражает реальность? Хотя возможны оба варианта, последний случай — значительное снижение производительности, достигающее 8,33 процента — гораздо более вероятен. Это особенно верно для высокопроизводительного ассемблерного кода, который использует быстрые инструкции, которые, как правило, вызывают непрерывную выборку инструкций.

Что делать с пожирателем циклов обновления DRAM?

Хммм. Когда мы обнаружили пожиратель циклов очереди предварительной выборки, мы научились использовать short Инструкции. Когда мы открыли 8-битный пожиратель циклов шины, мы научились использовать операнды памяти размером с байт, когда это возможно, и хранить переменные размером со слово в регистрах. Что мы можем сделать, чтобы обойти пожирателя циклов обновления DRAM?

Ничего.

Как я уже говорил ранее, обновление DRAM - это Божий акт. Обновление DRAM - фундаментальный,

неизменяемая часть работы ПК, и ни вы, ни я ничего не можем с этим поделать. Если бы обновления были менее частыми, надежность ПК была бы поставлена под угрозу, поэтому изменение таймера 1 или канала DMA 0 для уменьшения накладных расходов на обновление DRAM исключено. Также нет никакого способа структурировать код, чтобы минимизировать влияние обновления DRAM. Конечно, на некоторые инструкции обновление DRAM влияет меньше, чем на другие, но сколько операций умножения и деления подряд вы действительно можете использовать? Я полагаю, что код мог бы предположительно быть структурирован так, чтобы оставлять свободный доступ к памяти каждые 72 цикла, поэтому обновление DRAM не имело бы никакого эффекта. В старые времена, когда размер кода измерялся в байтах, а не К байтах, и процессоры были менее мощными и сложными, программисты так и делали фактически, они используют аналогичные приемы, чтобы добиться максимальной производительности от своего кода. Однако при программировании ПК пожиратель циклов очереди предварительной выборки проявил бы такую осторожность синхронизация кода - действительно сложная задача, и любое незначительное улучшение производительности этот результат никогда не сможет оправдать увеличение сложности программирования и ограничений для творческого программирования, которое повлечет за собой такой подход. Кроме того, все эти усилия тратятся впустую на более быстрых 8088-х, 286-х и других компьютерах с другим исполнением скорости и характеристики обновления. С этим ничего не поделаешь: полезный код обращается к памяти часто и с нерегулярными интервалами, а при длительном обновлении DRAM это всегда требует своей цены.

Если вы все еще лелеете мысли о сокращении накладных расходов на обновление DRAM, подумайте вот что. Инструкции, которые, как правило, не сильно страдают от обновления DRAM, - это те, которые имеют высокое соотношение времени выполнения ко времени выборки инструкций, и это не самые быстрые инструкции ПК. Конечно, не имело бы смысла использовать более медленные инструкции только для того, чтобы уменьшить накладные расходы на обновление DRAM, поскольку это общее время выполнения—обновление DRAM, инструкция извлечение и все остальное — это имеет значение.

Важно понимать, что обновление DRAM обычно замедляет работу снижение производительности может значительно варьироваться непредсказуемо, в зависимости от того, как обновления DRAM взаимодействуют с шаблоном вашего кода обращений к памяти. Когда вы используете таймер Zen и получаете дробное количество циклов для времени выполнения инструкции, это часто приводит к пожирателю циклов обновления DRAM. (Цикл адаптера дисплея является еще одним возможным виновником, и на процессорах 386s и более поздних версий промахи кэша и угрозы выполнения конвейера также приводят к подобному эффекту.) Всякий раз, когда вы получаете два результата синхронизации, которые отличаются меньше или больше, чем кажутся на первый взгляд обычно это тоже обновление DRAM. Благодаря DRAM refresh возможны вариации до 8,33 процента производительности кода ПК - это норма для курса.

Состояния ожидания

Состояния ожидания - это циклы, в течение которых центральный процессор осуществляет доступ по шине к устройству на ПК шина временно останавливается этим устройством, пока устройство готовится завершить чтение или запись. Состояния ожидания - это действительно самый низкий уровень производительности кода. Все мы обсуждали (и будем обсуждать), что даже на доступы DMA могут влиять состояния ожидания.

Состояния ожидания существуют, потому что центральный процессор должен иметь возможность сосуществовать с любым адаптером, независимо от того, независимо от того, насколько медленно (в разумных пределах). Ожидается, что 8088 сможет завершить каждую шину доступ —чтение или запись памяти или ввода—вывода -за 4 цикла, но адаптеры не всегда могут реагировать так быстро по ряду причин. Например, адаптеры отображения должны разделять доступ к памяти дисплея между центральным процессором и схемой, генерирующей видеосигнал на основе содержимого памяти дисплея, поэтому они часто не могут немедленно выполнить запрос центральный процессор на чтение или запись в память дисплея. Чтобы разрешить этот конфликт, адаптеры дисплея можно указать процессору ждать во время обращений к шине, вставив одно или несколько состояний ожидания, как показано на рисунке 4.6. Процессор просто сидит и работает вхолостую, пока вставлены состояния ожидания, затем завершает доступ, как только адаптер дисплея указывает на свою готовность по более длительная вставка состояний ожидания. То же самое было бы верно для любого адаптера, который не смог поддерживать работу с процессором.

Имейте в виду, все это прозрачно для исполняемого кода. Команда, которая сталкивается с состояниями ожидания выполняется точно так же, как если бы состояний ожидания не было, только медленнее. Состояния ожидания - это ничто больше или меньше, чем потерянное время с точки зрения процессора и вашей программы.

Понимая обстоятельства, при которых могут возникать состояния ожидания, вы можете избежать их по возможности. Даже если невозможно обойти состояния ожидания, вам все равно полезно понимать, как они могут замедлять выполнение вашего кода.

Во-первых, давайте узнаем немного больше о состояниях ожидания в отличие от обновления DRAM. В отличие от Обновление DRAM, состояния ожидания не возникают на какой-либо регулярной основе и не имеют определенной продолжительности. Состояния ожидания могут возникать только тогда, когда команда выполняет операции с памятью или чтение или запись ввода-вывода. Как наличие состояний ожидания, так и количество состояний ожидания подключение к любому заданному доступу по шине полностью контролируется устройством, к которому осуществляется доступ. Когда дело доходит до состояний ожидания, центральный процессор пассивен, просто принимая любое ожидание состояния, которое устройство, к которому осуществляется доступ, выбирает для вставки в ходе доступа. Все это имеет смысл, учитывая, что весь смысл механизма состояния ожидания заключается в том, чтобы позволить устройству растягивать любой доступ к себе на столько времени, сколько ему нужно для выполнения доступа.

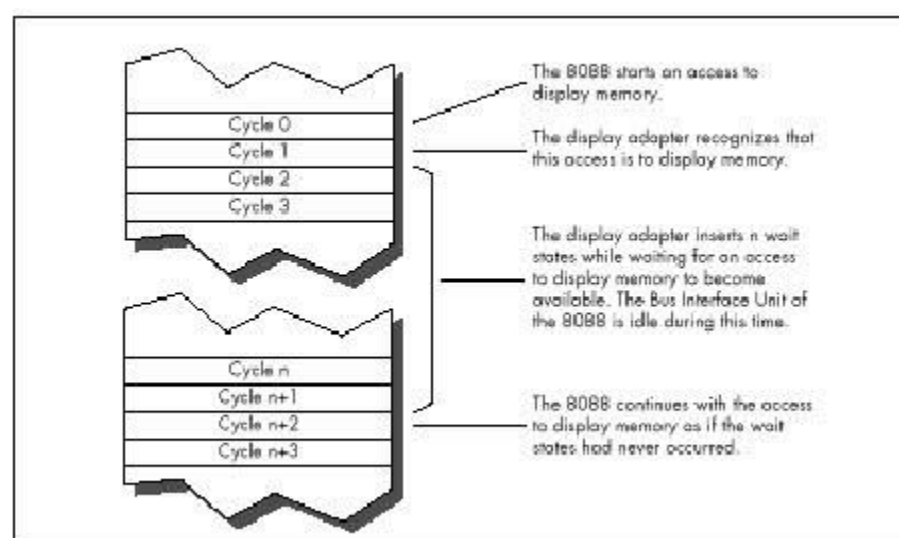


Рис. 4.6 Состояния ожидания видео, добавленные адаптером дисплея.

Как и в случае с обновлением DRAM, состояния ожидания не останавливают 8088 полностью. Исполнительный блок может продолжать обработку, пока вставлены состояния ожидания, до тех пор, пока ЕС не потребуется для выполнения доступа к шине. Однако на ПК состояния ожидания чаще всего возникают, когда команда обращается к операнду памяти, поэтому фактически модуль выполнения обычно останавливается из-за состояний ожидания. (Выборки команд редко ожидают на ПК на базе 8088, потому что система памяти находится в состоянии нулевого ожидания. Системы памяти класса AT обычно вставляют 1 или более состояний ожидания однако.)

Как выясняется, состояния ожидания создают серьезную проблему только в одной области ПК. В то время как любой адаптер может устанавливать состояния ожидания, на ПК это делают только адаптеры отображения в той степени, в какой это серьезно влияет на производительность .

Потребляющий цикл адаптер дисплея

Адаптеры дисплея должны обслуживать два основных устройства, и это создает фундаментальную проблему производительности . Основное устройство № 1 - это схема, которая управляет экраном дисплея. Эта схема должна постоянно считывать информацию из памяти дисплея, чтобы получить информацию, используемую для рисования символы или точки, отображаемые на экране. Поскольку экран необходимо перерисовывать между 50 и 70 раз в секунду, и поскольку для каждого перерисовывания экрана может потребоваться столько же с объемом памяти дисплея 36 000 считываний (больше в режимах Super VGA), master # 1 является действительно требовательным мастером. Однако, каким бы требовательным ни был master # 1, его потребности всегда должны удовлетворяться — иначе качество изображения на экране пострадает.

Мастер № 2 - это центральный процессор, который считывает данные из памяти дисплея и выполняет запись в нее для манипулировать байт, видео, схемы читает, чтобы сформировать изображение на экране.

Мастер #2-менее важно, чем учитель #1, так как процессор влияет только на качество дисплея

косвенно. Другими словами, если видеосхеме приходится ждать обращения к памяти дисплея, в изображении появятся дыры, выпадет снег и тому подобное, но если процессору приходится ждать отображения доступ к памяти, программа просто будет работать немного медленнее — ничего страшного.

Очень важно, какой мастер важнее, поскольку и ЦП, и видеосхема должна получить доступ к памяти дисплея, только один из двух основных устройств может одновременно считывать или записывать информацию из памяти дисплея. Потенциальные конфликты разрешаются путем отказа гарантируя видеосхеме любое количество обращений к памяти дисплея, в котором она нуждается, при этом центральный процессор ожидает, пока останутся все обращения к памяти дисплея.

Оказывается, процессору 8088 приходится долго ждать по трем причинам. Во-первых, видеосхема может занимать до 90% доступной памяти дисплея
время доступа, как показано на рисунке 4.7, занимает всего около 10% всего дисплея доступ к памяти для 8088. (Эти проценты значительно различаются среди многих Клоны EGA и VGA.)

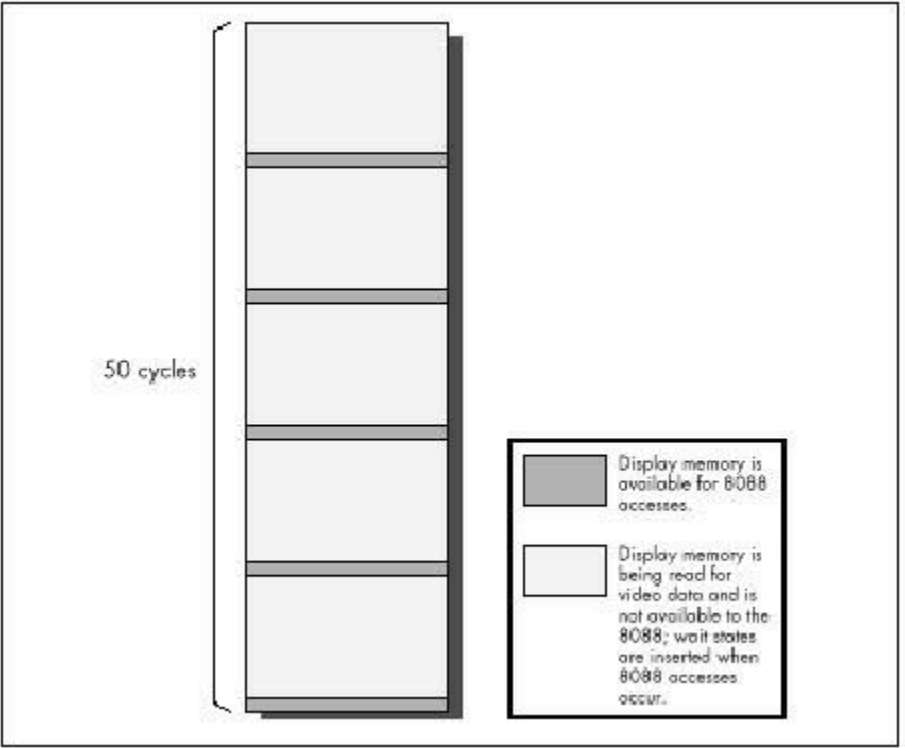


Рисунок 4.7 Распределение доступа к памяти дисплея.

Во-вторых, поскольку отображаемые точки (или пиксели, сокращение от “элементы изображения”) должны быть нарисованы на экране с постоянной скоростью, многие адаптеры отображения предоставляют память выполняется доступ только с фиксированными интервалами. В результате может быть потеряно время, пока 8088 синхронизируется с началом следующего доступа к памяти адаптера дисплея, даже если видео в это время схема не обращается к памяти дисплея, как показано на рисунке 4.8.

Наконец, время, необходимое адаптеру дисплея для завершения доступа к памяти, связано с скоростью часов, которые генерируют пиксели на экране, а не в памяти скорость доступа 8088. Следовательно, время, необходимое для завершения работы памяти дисплея доступ 8088 на чтение или запись часто занимает больше времени, чем системная память завершение доступа, даже если 8088 повезло получить свободный доступ к памяти дисплея как только он становится доступным, снова как показано на рисунке 4.8. Любой или все три фактора Я описал, что может привести к состояниям ожидания, замедлению работы 8088 и появлению проблем с отображением цикл работы адаптера.

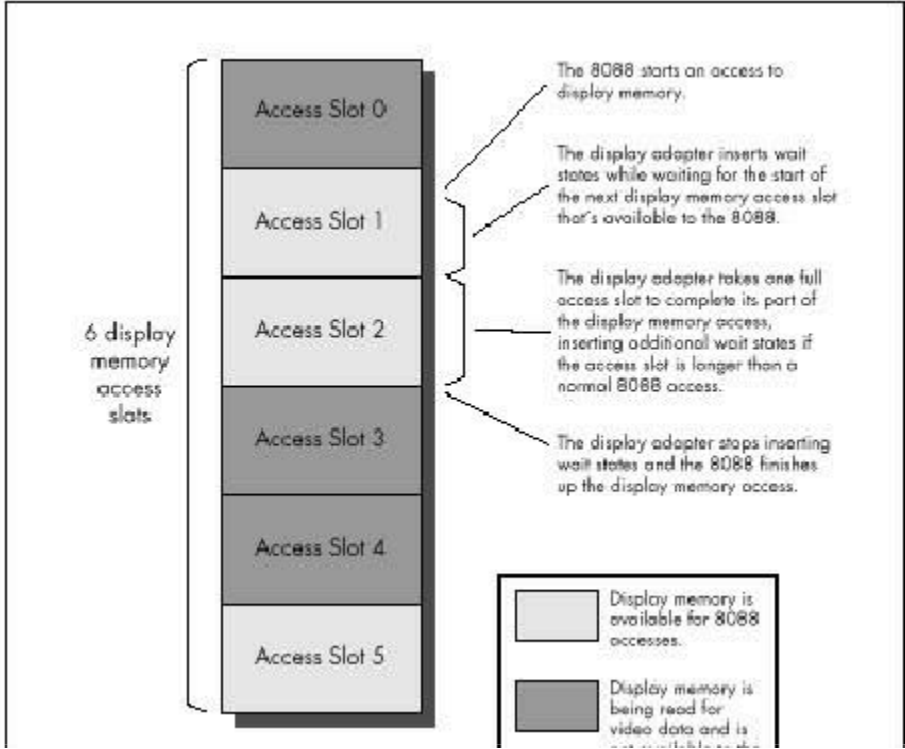




Рисунок 4.8 Слоты доступа к памяти дисплея.

Если что-то из этого вам непонятно, не волнуйтесь. Важным моментом является то, что память дисплея не очень быстрая по сравнению с обычной системной памятью. Насколько он медленный? Невероятно медленный. Помните, каким медленным был злополучный PCjr от IBM? На случай, если вы забыли, я обновлю ваш Память: PCjr был в лучшем случае вдвое быстрее ПК. У PCjr был 8088 работает на частоте 4,77 МГц, как и ПК — как вы думаете, почему он был намного медленнее? Я скажу вам почему: вся память в PCjr была памятью дисплея.

Достаточно сказано. Однако вся память ПК не является памятью дисплея, и если если вы достаточно тупы, чтобы поместить код в память дисплея, компьютер не будет работать так медленно, как PCjr. (Размещение кода или других не видеоданных в неиспользуемых областях дисплея память кажется отличной идеей — пока вы не учтете влияние на предварительную выборку инструкций о сокращении вдвое и без того низкой производительности доступа к памяти 8088. Запуск вашего код из памяти дисплея работает вроде как на гипотетических 8084—8086 с 4-разрядной шиной. Не рекомендуется!) Учитывая, что ваш код и данные хранятся в обычной системе объем памяти ниже отметки в 640 КБ, насколько сильно влияет цикличность адаптера дисплея eater на производительность?

Ответ значительно варьируется в зависимости от того, какой адаптер дисплея и о каком дисплее режим, о котором мы говорим. Потребляющий много времени адаптер дисплея хуже всего работает с улучшенным Графическим адаптером (EGA) и оригинальным видеографическим массивом (VGA). (Многие видеокарты, особенно новые, они содержат гораздо меньше состояний ожидания, чем оригинальные VGA от IBM. С с другой стороны, у Super VGA больше байт памяти дисплея, к которым можно получить доступ в режиме high- режим разрешения.) В то время как цветной / графический адаптер (CGA), монохромный дисплей Адаптер (MDA) и видеокарта Hercules (HGC) зависят от адаптера дисплея кроме того, они также потребляют много времени, но страдают в меньшей степени. Поскольку VGA представляет собой базовый стандарт для компьютерной графики сейчас и в обозримом будущем, и поскольку он самый сложный графический адаптер для повышения производительности, мы ограничим наше обсуждение VGA (и его близким родственником, EGA) до конца этой главы.

Влияние потребления циклов адаптером дисплея

Даже на EGA и VGA влияние потребления циклов адаптером дисплея зависит от выбранного режима отображения. В текстовом режиме пожиратель циклов адаптера дисплея редко является важным фактор. Дело не в том, что пожиратель циклов отсутствует; однако всего 4000 байт управляют отображение всего текста в текстовом режиме, и даже с пожирателем циклов адаптера дисплея это просто не обработка 4000 байт занимает так много времени. Даже если бы пожиратель циклов адаптера дисплея должен был из—за того, что 8088 занимает до 5 мкс на каждый доступ к памяти дисплея - более пяти в обычном режиме — все равно потребуется всего 4000 x 2x5 мкс, или 40 мс, для чтения и записи каждого байта памяти дисплея. Это много времени, измеряемого в 8088 циклах, но это меньше, чем мгновение ока по человеческому времени, а производительность видео важна только по человеческому времени. В конце концов, весь смысл рисования графики заключается в передаче визуальной информации, и если эта информация может быть представлена быстрее, чем видит глаз, то это по определению достаточно быстро .

Это не означает, что потребляющий цикл адаптер дисплея не может иметь значения в текстовом режиме. В В главе 3 я пересказал историю дебатов среди авторов писем в журнал о том, насколько быстро можно писать символы, чтобы отобразить память, не вызывая снега. Разработчики тщательно подсчитали время цикла команд Intel, чтобы увидеть, сколько операций записи в память дисплея они могли бы втиснуть в один горизонтальный интервал повторной трассировки. (На CGA только в течение короткого интервала повторной трассировки по горизонтали и более длительного интервала повторной трассировки по вертикали доступ к памяти дисплея в текстовом режиме из 80 столбцов возможен без вызывая снегопад.) Конечно, теперь мы знаем, что их главным грехом было игнорирование очереди предварительной выборки; даже если бы не было состояний ожидания, их расчеты были бы

чрезмерно оптимистичными. Однако существуют также состояния ожидания памяти дисплея, поэтому расчеты были не просто оптимистичными, а безумно оптимистичными.

Несмотря на ситуации в текстовом режиме, подобные описанным выше, когда адаптер дисплея по-настоящему увлекательный цикл происходит в графическом режиме, и особенно в высоком разрешении графические режимы EGA и VGA. Проблема здесь не в том, что обязательно существует больше состояний ожидания для каждого доступа в режимах highgraphics (которые варьируются от адаптера к адаптеру и от режима к режиму). Скорее всего, проблема просто в том, что на экран приходится намного больше байт памяти на экран в этих режимах, чем в графических режимах с более низким разрешением и в текстовых режимах, поэтому требуется гораздо больше обращений к памяти дисплея — для каждого требуется своя доля отображения состояния ожидания памяти — для отрисовки изображения заданного размера. Когда получая доступ ко многим тысячам байт, используемых в графических режимах с высоким разрешением, совокупные эффекты состояний ожидания отображения памяти могут серьезно повлиять на код производительность, даже измеряемая во времени человека.

Например, если мы предположим, что для доступа к памяти дисплея EGA требуется столько же 5 мкс графический режим высокого разрешения, который мы предположили для текстового режима, займет $26,000 \times 2 \times 5$ секунд, или 260 мс, для прокрутки экрана один раз в графическом режиме высокого разрешения EGA, режим 10 часов. Это более четверти секунды - ощутимо по человеческим меркам, вечность по компьютерным стандартам.

Звучит довольно серьезно, но мы сделали необоснованное предположение о памяти скорость доступа. Давайте получим точные цифры. В листинге 4.11 доступ к памяти дисплея осуществляется на максимальная скорость 8088 достигается за счет REP MOVSW с памятью дисплея в качестве источника и назначения. Код из листинга 4.11 выполняется за 3,18 мкс на каждый доступ к дисплею память — не так долго, как мы предполагали, но, тем не менее, достаточно долго.

ЛИСТИНГ 4.11 LST4-11.ASM

```
>; Times speed of memory access to Enhanced Graphics
; Adapter graphics mode display memory at A000:0000.
    mov ax,0010h
    int 10h;select hi-res EGA graphics
                ; mode 10 hex (AH=0 selects
                ; BIOS set mode function,
                ; with AL=mode to select)

    mov ax,0a000h
    mov ds,ax
    mov es,ax;move to & from same segment
    sub si,si;move to & from same offset
    mov di,si
    movcx, 800h;move 2K words
    cld
    call ZTimerOn
    rep movsw;simply read each of the first
                ; 2K words of the destination segment,
                ; writing each byte immediately back
                ; to the same address. No memory
                ; locations are actually altered; this
                ; is just to measure memory access
                ; times

    call ZTimerOff
    mov ax,0003h
    int 10h;return to text mode
```

<Для сравнения давайте посмотрим, сколько времени занимает тот же код при доступе к обычной системе ОПЕРАТИВНАЯ память вместо памяти дисплея. Код из листинга 4.12, который выполняет повторное перемещение от сегмента кода к сегменту кода выполняется за 1,39 доллара США на память дисплея доступ. Это означает, что в среднем 1,79 доллара США (более 8 циклов!) теряются из-за пожирателя циклов адаптера дисплея при каждом доступе. Другими словами, цикл адаптера дисплея- eater может более чем вдвое увеличить время выполнения кода 8088!

ЛИСТИНГ 4.12 LST4-12.ASM

```
>; Times speed of memory access
to normal system ; memory.
```

```
<>movax,ds
mov es,ax;перемещение в тот же
сегмент и из него sub si, si; перемещение
в то же смещение и из него перемещение
di, si перемещениеcx,800 часов; переместить
2 тысячи слов cld вызвать
ZTimerOn представитель movsw; просто
прочитайте каждое из первых
; 2K слов сегмента назначения,
; немедленная запись каждого байта обратно
; по тому же адресу. Нет памяти
; местоположения фактически изменены; это
; просто для измерения доступа к памяти
; время
вызовите ZTimerOff
```

<Имейте в виду, что здесь мы говорим о наихудшем случае; влияние дисплея потребление цикла адаптером пропорционально проценту времени, затрачиваемого данной кодовой последовательностью доступ к памяти дисплея.



Подпрограмма рисования линий, которая выполняет, возможно, дюжину инструкций

для каждого доступа к памяти дисплея, обычно теряет меньшую производительность для потребляет больше циклов, чем программа копирования блоков или прокрутки использующая инструкции по воспроизведению. Масштабируемая трехмерная графика, которая тратите много времени на выполнение вычислений (часто используя очень медленную арифметику с плавающей запятой), как правило, меньше страдает.

Кроме того, код, который обращается к памяти дисплея нечасто, как правило, страдает только примерно половина состояний ожидания максимальной памяти дисплея, потому что в среднем такой код будет доступ к памяти дисплея на полпути между одним доступным слотом доступа к памяти дисплея и следующим. В результате код, который обращается к памяти дисплея, работает менее интенсивно, чем код в листинге 4.11 дисплей в среднем потеряет 4 или 5, а не 8 с лишним циклов. потребляющий цикл адаптер при каждом обращении к памяти.

Тем не менее, потребление цикла адаптером дисплея всегда сказывается на графическом коде. Интересно, что на машинах ATs и 80386 это сказывается намного выше, потому что, хотя эти компьютеры могут выполнять гораздо больше инструкций в микросекунду, чем ПК на базе 8088, доступ к памяти дисплея на этих компьютерах занимает столько же времени, сколько на ПК на базе 8088. Помните, что ограниченная скорость доступа к графическому адаптеру является неотъемлемой характеристикой адаптера, поэтому самый быстрый компьютер в мире не сможет получить доступ память дисплея на йоту быстрее, чем позволяет адаптер.

Что делать с потребителем циклов адаптера дисплея?

Что мы можем сделать с потребителем циклов адаптера дисплея? Что ж, мы можем минимизировать отображение обращения к памяти, когда это возможно. В частности, мы можем попытаться избежать чтения / изменения / записи операции с отображаемой памятью, подобные тем, которые используются для маскирования отдельных пикселей и обрезки изображений. Почему? Потому что операции чтения / изменения / записи требуют двух обращений к отображаемой памяти (одного чтение и одна запись) при каждом манипулировании памятью дисплея. Вместо этого мы должны попытаться используйте операции записи такого типа, которые устанавливают все пиксели в заданном байте памяти дисплея одновременно, поскольку для таких операций записи не требуются сопутствующие права доступа на чтение. Ключевым моментом здесь является то, что только для записи байта в память дисплея требуется вдвое меньше обращений к памяти дисплея чем для чтения байта из памяти дисплея, маскирования его части и изменения остальной части, и запишите байт обратно в память дисплея. Вдвое меньшее количество обращений к памяти дисплея означает вдвое меньшее количество состояний ожидания памяти дисплея.



Более того, 486-е и Pentium, а также последние версии Super VGA используют схемы кэширования записи, которые значительно ускоряют запись в память дисплея быстрее, чем чтение в память дисплея.

Аналогично, пожиратель циклов адаптера дисплея делает популярный эксклюзивный или анимационная техника, которая требует парные чтения и записи памяти дисплея, меньше-

чем-подходит для ПК. Исключающее или анимации следует избегать в пользу простого запись изображений для отображения памяти, когда это возможно.

Другим принципом программирования адаптера дисплея на 8088 является многократное очень быстрый доступ к памяти дисплея, чтобы использовать как можно больше дефицитных максимально использует память дисплея. Это особенно важно, когда требуется быстро отрисовывать большое количество изображений, поскольку можно использовать практически все доступные дисплеи доступ к памяти за короткий промежуток времени в память дисплея может быть записано много байтов. Повторяющиеся строковые инструкции идеально подходят для максимального использования памяти дисплея обращается; конечно, повторяющиеся строковые инструкции могут использоваться только для целых байтов, так что это еще один аргумент в пользу изменения памяти дисплея по одному байту за раз. (На более быстром процессоры, однако, используют память дисплея настолько медленно, что часто приходится выполнять несколько чтобы воспользоваться преимуществами, требующими выполнения инструкций между обращениями к памяти дисплея, циклы, которые в противном случае были бы потрачены впустую на состояния ожидания.)

Было бы полезно подробно изучить проблему пожирателя циклов адаптера дисплея с большим количеством пример кода и тайминги выполнения, но, увы, у меня сейчас нет для этого места. На данный момент все, что вам действительно нужно знать о потребителе циклов адаптера дисплея, это то, что на 8088 вы можете потерять более 8 циклов времени выполнения при каждом доступе к памяти дисплея. При интенсивном доступе к памяти дисплея потери действительно могут быть такими же высокими как 8 циклов (и до 50, 100 или даже больше на 486-х и Pentium в паре с медленным VGA), в то время как для среднего графического кода потери приближаются к 4 циклам; в любом случае, влияние на производительность значительно. Есть только один способ выяснить, насколько потребляющий цикл адаптер дисплея оказывает существенное влияние на любую конкретную графику код, и это, конечно же, для измерения производительности этого кода.

Потребляющие цикл: краткое описание

В этой главе мы рассмотрели много сложного материала, так что не расстраивайтесь, если вы поняли не все из прочитанного; все станет ясно из дальнейшего чтение, особенно после того, как вы изучите, рассчитаете время и настроите код, который написали сами. Что действительно важно, так это то, что вы закончите эту главу, понимая, что на 8088:

- 8-разрядный пожиратель циклов шины приводит к тому, что каждый доступ к операнду размером со слово занимает 4 цикла дольше, чем эквивалентный доступ к операнду размером с байт.
- Пожиратель циклов очереди предварительной выборки может привести к увеличению времени выполнения команд в четыре раза по сравнению с официально задокументированным временем цикла.
- Пожиратель циклов обновления DRAM замедляет работу большей части кода ПК, снижая производительность в диапазоне до 8,33 процента.
- Время использования адаптера дисплея обычно удваивает, а может и более чем утроить продолжительность стандартного 4-циклового доступа к памяти дисплея, при этом интенсивный доступ к дисплею доступ к памяти страдает больше всего.

Эти базовые знания о велосипедистах помогут вам лучше понять результаты, представленные Zen timer, а это значит, что вы уже на пути к написанию статьи высокопроизводительный код на ассемблере.

Что все это значит?

Вот и все: жизнь в программном интерфейсе. Это не особенно приятно представьте себе обитателей этого странного царства, где встречаются аппаратное и программное обеспечение, малоизвестные пожиратели циклов, которые высасывают скорость из вашего ничего не подозревающего кода. Тем не менее, некоторые из эти пожиратели циклов могут быть сведены к минимуму за счет сокращения инструкций, использования регистров, использования операндов памяти размером в байт и как можно меньшего доступа к памяти дисплея. Ни один из пожирателей циклов не может быть устранен, а динамическое обновление оперативной памяти вряд ли может быть устранено вообще; тем не менее, не лучше ли вам знать, насколько быстро на самом деле выполняется ваш код — и

почему—чем вы читали официальное время выполнения и гадали? И хотя конкретные пожиратели циклов различаются по важности на более поздних процессорах семейства x86, с некоторыми задержками цикла пожиратели полностью исчезают и появляются новые, концепция, согласно которой понимание этих малоизвестных гремлинов является ключом к успеху, остается неизменной, как мы увидим снова и снова в последующих главах.

Глава 5 – Пересечение границы

Поиск файлов с перезапускаемыми блоками

Мы только что переехали. Эти три маленьких слова должны вселить ужас в сердце любого у кого есть что-то большее, чем спальный мешок и зубная щетка. Нашим последним переездом был обычный зоопарк — и еще кое-что. Потому что расстояние от старого дома до нового было всего пять миль, мы использовали машины, чтобы перевозить все, что меньше стиральной машины. У нас есть большое домашнее хозяйство — кошки, собаки, дети, друзья, называйте как хотите — так что процесс переезда занял несколько поездок на машине. Большое количество — 33, если быть точным. Я лично потратил около 15 часов просто ездил туда-сюда между двумя домами. Переезд занял несколько дней.

Больше никогда.

Вы, вероятно, задаетесь вопросом о двух вещах: какое отношение это имеет к высокопроизводительному программированию, и почему, черт возьми, я не арендовал грузовик и не переехал за один или два дня поездки, экономия часов за рулем? Так получилось, что второй вопрос отвечает на первый. Я не брал напрокат грузовик, потому что казалось проще и дешевле использовать легковые автомобили — большой грузовик не нужен ездить, без аренды, распределять работу более управляемо и так далее.

Это было не проще и даже ненамного дешевле. (Водить машину стоит совсем немного - 330 миль, не говоря уже о стоимости 15 часов моего времени.) Но в то время казалось, что хотя мой подход был бы проще и дешевле. На самом деле, я не осознавал, сколько времени я потратил впустую, разъезжая туда-сюда, пока не сел писать эту главу.

В главе 1, Я кратко обсудил использование перезапускаемых блоков. Это, как вы, возможно, помните, процесс обработки порциями наборов данных, слишком больших, чтобы поместиться в памяти, чтобы они могли обрабатываться примерно так же быстро, как если бы они помещались в памяти. Перезапускаемый блок подход очень быстрый, но относительно сложный в программировании.

На противоположном конце спектра находится побайтовая обработка, при которой DOS (или, в менее экстремальные случаи, группа библиотечных функций) разрешается выполнять всю тяжелую работу, так что вам приходится иметь дело только с одним байтом за раз. Побайтовая обработка проста в программировании, но может быть чрезвычайно медленной из-за огромных накладных расходов, возникающих в результате вызова DOS каждый раз, когда необходимо обработать байт.

Звучит знакомо? Так и должно быть. Я использовал побайтовый подход, и накладные расходы, связанные с ездой туда-сюда, привели к снижению производительности. Аренда грузовика (перезапускаемый блочный подход) потребовала бы больше усилий и предусмотрительности, но окупилась бы сторицей.



Простой, знакомый подход часто не имеет ничего в свою пользу, за исключением того, что он требует меньше размышлений; не самое большое преимущество при написании высокопроизводительного кода — или при перемещении.

И с учетом этого давайте рассмотрим довольно сложное приложение с перезапускаемыми блоками.

Поиск текста

Приложение, которое мы собираемся изучить, выполняет поиск в файле указанной строки. Мы разработайте программу, которая будет искать строку в файле, указанном в командной строке, (также указанную в comline), а затем сообщит, была ли строка найдена или нет. (Поскольку искомая строка получается через argv, она не может содержать никаких пробелов символов.)

Это очень ограниченное подмножество того, что могут делать поисковые утилиты, такие как grep, и на самом деле это не так задумано как общепользовательное приложение; цель - дать представление о перезапускаемых блоках в частности и оптимизация в целом в ходе разработки поисковая система. Однако эту поисковую систему будет легко подключить к любой программе, и ничто не мешает вам использовать его в более плодотворном контексте, например для поиска по набору файлов, выбираемому пользователем.

Первое, на что следует обратить внимание при разработке нашей программы, - это соответствующий текстовый поиск подход к использованию. Существуют буквально десятки работоспособных способов поиска файла. Мы можем немедленно отбросить все подходы, которые предполагают чтение любого байта файла более один раз, потому что время доступа к диску на порядки медленнее, чем любая обработка данных выполняется нашим собственным кодом. Основываясь на нашем опыте, описанном в главе 1, мы также можем отказаться от все подходы, которые получают байты либо по одному, либо небольшими наборами из DOS. Мы хотим считывать большие “заполненные буферами” байты одновременно из искомого файла, и чем больше буфер тем лучше — чтобы минимизировать накладные расходы DOS. Хороший примерный размер - это буфер, который будет составлять от 16 до 64 КБ, в зависимости от точного подхода к поиску, 64 КБ - это максимальный размер, поскольку указатели рядом обеспечивают превосходную производительность.

Итак, мы знаем, что хотим работать с большим буфером, заполняя его как можно реже. Теперь нам нужно выяснить, как выполнять поиск по файлу, загружая его в этот большой буфер порциями. Чтобы достичь этого, мы должны знать, как мы хотим осуществлять поиск, и это не сразу очевидно. С чего нам начать?

Что ж, было бы полезно подумать, как бы мы выполняли поиск, если бы наш поиск включал только один буфер, уже находящийся в памяти. Другими словами, предположим, нам не нужно беспокоиться с обработкой файлов вообще, и далее предположим, что нам не нужно иметь дело с поиском через несколько блоков. В конце концов, это хорошее описание важнейшего внутреннего цикла нашей программы поиска, на которую программа будет тратить практически все свое время (помимо неизбежных накладных расходов на доступ к диску).

Избегая ловушки строк

Самым простым подходом было бы использовать библиотечную функцию C / C ++. Наиболее близкое соответствие что нам нужно, так это strstr() , который ищет в одной строке первое вхождение а вторая строка. Однако, хотя strstr() и работала бы, она не идеальна для наших целей.

Проблема заключается в следующем: где мы хотим выполнить поиск в буфере фиксированной длины для первого появления строки, strstr() выполняет поиск в строке первого вхождения другой строки.

Мы могли бы поместить нулевой байт в конец нашего буфера, чтобы позволить strstr() работать, но зачем беспокоит? Функция strstr() должна тратить время либо на проверку конца строки выполняется поиск, либо на определение длины этой строки — напрасные усилия, учитывая, что мы мы уже точно знаем, какой длины наш буфер поиска. Даже если данная strstr () реализация написана хорошо, ее производительность пострадает, по крайней мере, для нашего приложения, от ненужных накладных расходов.



Это иллюстрирует, почему вы не должны думать о библиотечных функциях C / C ++ как о черных ящиках; поймите, что они делают, и попытайтесь выяснить, как они это делают это, и соотнесите это с их производительностью в интересующем вас контексте.

Методы грубой силы

Учитывая, что ни одна библиотечная функция C / C ++ не соответствует нашим потребностям в точности, очевидная альтернатива этот подход основан на методе грубой силы, который использует `memcmp()` для сравнения всех потенциальных объектов соответствие местоположения в буфере строке, которую мы ищем, как показано на рисунке 5.1.

Кстати, мы могли бы, конечно, использовать наш собственный код, работающий с указателями в цикле, чтобы выполнить сравнение вместо `memcmp()`. Но `memcmp()` почти наверняка будет использовать очень быструю инструкцию `REPZ CMPS`. Однако никогда не предполагайте! Не помешало бы использовать отладчик для проверки реальной реализации `memcmp()` в машинном коде из вашего компилятора. При необходимости вы всегда можете написать свой собственный язык ассемблера реализация `memcmp()`.

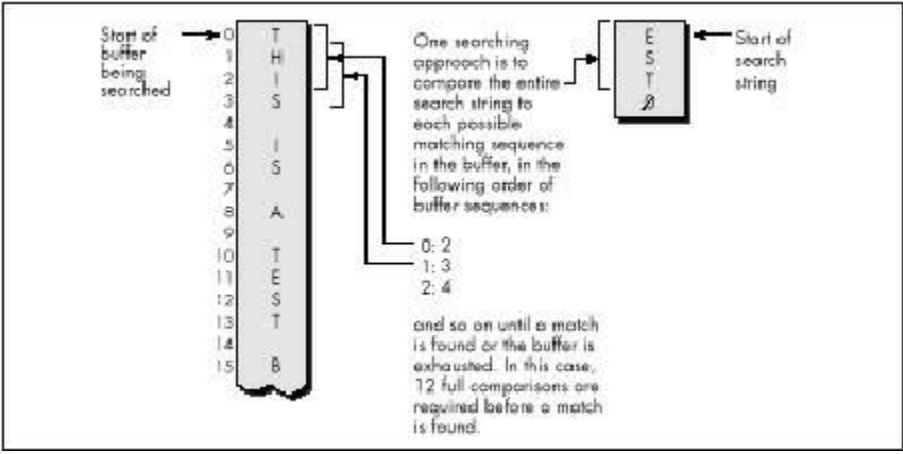


Рисунок 5.1 Техника поиска методом перебора.

Вызов `memcmp()` для каждого потенциального местоположения соответствия работает, но влечет за собой значительные накладные расходы. Для каждого сравнения требуется передача параметров и выполнение вызова и возврат из `memcmp()` вместе с прохождением цикла сравнения. Несомненно, есть способ получше!

Действительно, есть. Мы можем исключить большинство вызовов `memcmp()`, выполнив простой тест для каждого потенциального местоположения, которое сразу отклонит большинство таких местоположений. Мы просто проверьте, совпадает ли первый символ потенциально подходящего местоположения буфера первый символ строки, которую мы ищем. Мы могли бы выполнить эту проверку с помощью указатель в цикле для сканирования буфера в поисках следующего соответствия первому символу, останавливаясь на проверяет соответствие с остальной частью строки только тогда, когда совпадает первый символ, поскольку показано на рисунке 5.2.

Использование `memchr()`

Однако есть еще лучший способ реализовать этот подход. Используйте `memchr()` функция, которая делает не что иное, как находит следующее вхождение указанного символа в буфере фиксированной длины (предположительно, с помощью чрезвычайно эффективного Повторите инструкцию `SCASB`, хотя, опять же, не мешало бы проверить). Используя `memchr()` для сканируя потенциальные соответствия, которые затем можно полностью протестировать с помощью `memcmp()`, мы можем создать высокоэффективную поисковую систему, которая использует имеющуюся у нас информацию с максимальной пользой о буфере, в котором выполняется поиск, и строке, которую мы ищем. Наш движок также в значительной степени полагается на повторяющиеся строковые инструкции, предполагая, что `memchr()` и `memcmp()` библиотечные функции закодированы должным образом.

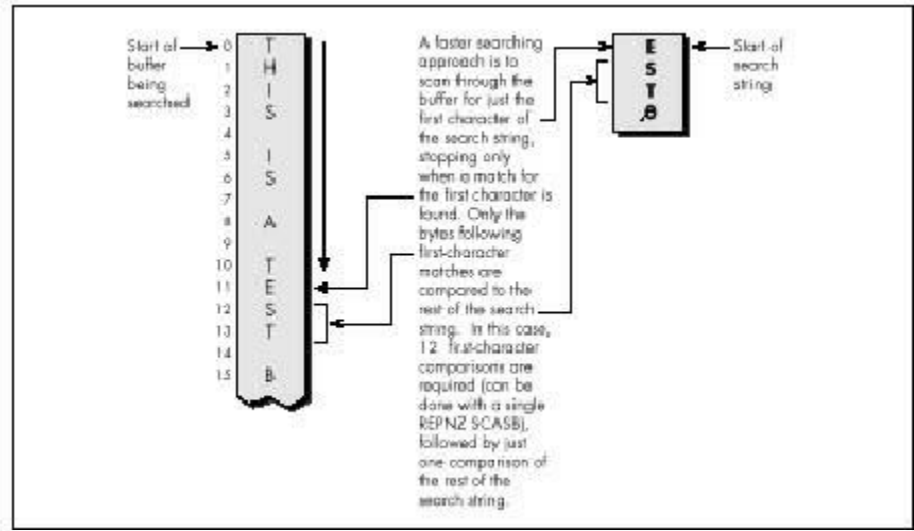


Рисунок 5.2 - Более быстрый метод поиска строк.

Мы собираемся использовать этот подход в нашей программе поиска файлов; единственный трюк заключается в при принятии решения о том, как интегрировать этот подход с перезапускаемыми блоками для поиска по файлам размером больше нашего буфера. Это, конечно, не самый быстрый поиск из возможных алгоритм; в качестве одного из примеров, алгоритм Бойера-Мура, который умело устраняет многие местоположения буфера как потенциальные совпадения в процессе проверки предыдущих местоположений, может быть значительно быстрее. Однако алгоритм Бойера-Мура довольно сложен для понимания и реализации, и это отвлекло бы нас от нашей основной задачи - перезапуска блоки, поэтому мы сохраним это для следующей главы (если быть точным, главы 14). Кроме того, я подозреваю, вы обнаружите, что подход, который мы будем использовать, достаточно быстр для большинства целей.

Теперь, когда мы выбрали подход поиска, давайте интегрируем его с обработкой файлов и поиском по нескольким блокам. Другими словами, давайте сделаем его перезапускаемым.

Создание возможности перезапуска поиска

Так получилось, что собрать части этой поисковой программы воедино не составляет особого труда . В принципе, мы будем считывать данные из буфера (мы будем работать с 16 КБ за раз, чтобы избежать проблемы с переполнением знака с целыми числами), искать в нем совпадение с описан механизм memchr() / memstr () и завершите работу с ответом “строка найдена”, если желаемая строка найдена.

В противном случае мы загрузим в другой буфер данные из файла, проведем поиск в нем и так далее. единственный трюк заключается в обработке потенциально совпадающих последовательностей в файле, которые начинаются с одной буферизуют и заканчиваются в следующем — то есть последовательностях, охватывающих буферы. Мы справимся с этим, скопировав непроверенные байты в конце одного буфера в начало следующего и прочитав намного меньше байтов при следующем заполнении буфера.

Точное количество байт, которые должны быть скопированы из конца одного буфера в начало буфера далее следует длина искомой строки минус 1, поскольку именно столько байтов в конец буфера не может быть проверен на возможные совпадения (потому что проверка будет выполняться в конце буфера).

На самом деле это все, что нужно. В листинге 5.1 показана программа поиска файлов. Как вы можете видите, это не особенно сложно, хотя требуется несколько довольно непрозрачных строк кода для обработки слияния конца одного блока с началом следующего. Код, который выполняет поиск по одному блоку — функции SearchForString() — просто и компактно (как и должно быть , учитывая, что это, безусловно, наиболее интенсивно выполняемый код в списке).

Листинг 5.1 прекрасно иллюстрирует основную концепцию перезапускаемых блоков: Организуйте свою программу так, чтобы вы могли выполнять обработку в каждом блоке так быстро, как только сможете, если был только один блок — то есть на максимальной скорости — и делайте свои блоки как можно более большими, чтобы минимизировать накладные расходы, связанные с переходом от одного блока к следующему.

ЛИСТИНГ 5.1 SEARCH.C

```
/* Программа для поиска файла, указанного в первой командной строке
   * аргумент для строки, указанной во второй командной строке
   * аргумент. Выполняет поиск путем чтения и поиска блоков
   * размером BLOCK_SIZE. */

#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <alloc.h> /* alloc.h для компиляторов Borland,
                    malloc.h для компиляторов Microsoft */

#define РАЗМЕР блока 0x4000 /* мы будем обрабатывать файл блоками 16К */

/* Выполняет поиск указанного количества последовательностей в указанном
   буфере на предмет совпадений со строкой поиска SearchStringLength . Примечание
   что вызывающий код уже должен иметь сокращенную длину поиска
   при необходимости компенсировать расстояние от конца
   буфер до последнего возможного начала
   совпадающей последовательности в
   buffer.
   */
```

```
int SearchForString(беззнаковый символ * Buffer, длина поиска int,
unsigned char * searchString, int SearchStringLength)
```

```
unsigned char * Потенциальное совпадение;
```

```
/* Выполнять поиск до тех пор, пока есть местоположения, которые
потенциально совпадают оставшиеся */
```

```
while ( длина поиска ) {
```

```
/* Посмотреть, можно ли найти первый символ строки
поиска */ if ( (Потенциальное соответствие =
```

```
>memchr(Buffer, *SearchString, SearchLength)) == NULL ) {
```

```
break; /* No matches in this buffer */
```

```
/* The first character matches; see if the rest of the string
also matches */
```

```
if ( SearchStringLength == 1 ) {
```

```
return(1); /* That one matching character was the whole
```

```
search string, so we've got a match */
```

```
else {
```

```
/* Check whether the remaining characters match */
```

```
if ( !memcmp(PotentialMatch + 1, SearchString + 1,
```

```
SearchStringLength - 1) ) {
```

```
return(1); /* We've got a match */
```

```
/* The string doesn't match; продолжайте, указывая мимо
```

```
местоположение потенциального соответствия, которое мы только что отклонили * /
```

```
Длина поиска -= буфер потенциального соответствия + 1;
```

```
Буфер = Потенциальное соответствие + 1;
```

```
возврат (0); /* Соответствие не найдено */
```

```
main(int argc, char *argv[]) {
```

```
int Done; /* Указывает, выполнен ли поиск */
```

```
дескриптор int; /* Дескриптор файла, в котором выполняется поиск */
```

```
int WorkingLength; /* Длина текущего блока */
```

```
int SearchStringLength; /* Длина строки для поиска */
```

```
int BlockSearchLength; /* Длина для поиска в текущем блоке */
```

```
найден int; /* Указывает на окончательное завершение поиска
```

```
статус */
```

```
int NextLoadCount; /* количество байтов для чтения в следующий блок,
```

```
учет байтов, скопированных из
```

```
последнего блока */
```

```
unsigned char *Рабочий блок; /* Буфер хранения блоков */
```

```
символ без знака *Строка поиска; /* Указатель на строку для поиска */
```

```
unsigned char *NextLoadPtr; /* Смещение, с которого следует начать загрузку
```

```
следующий блок, учитывающий
```

```
байты, скопированные из последнего блока */
```

```
/* Проверьте правильное количество аргументов */
```

```
if ( argc!= 3 ) {
```

```
printf("использование: поиск по имени файла search-string\n");
```

```
выход(1);
```

```
/* Попробуйте открыть файл для поиска */
```

```
if ( (Handle = открыть(argv[1], O_RDONLY | O_BINARY)) == -1) {
```

```
printf("Не удастся открыть файл: %s\n", argv[1]);
```

```
выход(1);
```

```
/* Вычислить длину текста для поиска */
```

```
Строка поиска = argv[2];
```

```
SearchStringLength = strlen(строка поиска);
```

```
/* Попытаться получить память для буферизации данных */
```

```
if ( (WorkingBlock = malloc(BLOCK_SIZE)) == NULL ) {
```

```
printf("Не удастся получить достаточно памяти");
```

```
выход(1);
```

```
/* Загрузите первый блок в начале буфера и попробуйте
```

```
заполнить весь буфер */
```

```
NextLoadPtr = рабочий блок;
```

```
NextLoadCount = РАЗМЕР БЛОКА;
```

```
Готово = 0; /* Поиск еще не завершен * /
```

```
Найдено = 0; /* Предположим, мы не найдем совпадения */
```

```
/* Ищите файл в блоках размером с блок */
```

```
выполняйте {
```

```
/* Считывайте в том количестве байтов, которое необходимо для заполнения блока
```

```
(с учетом байтов, скопированных из последнего блока), или
```

```
остальные байты в файле, в зависимости от того, что меньше */
```

```
if ( (Рабочая длина = чтение(дескриптор, NextLoadPtr,
```

```
NextLoadCount)) == -1) {
```

```
printf("Ошибка чтения файла %s\n", argv[1]);
```

```
выход(1);
```

```
/* Если мы прочитали не все запрошенные байты, мы закончили
```

```
после
```

```
этого блока, независимо от того, найдем мы совпадение или нет */
```

```
<>if ( WorkingLength != NextLoadCount ) {

    Done = 1;

    /* Account for any bytes we copied from the end of the last
       block in the total length of this block */

    WorkingLength += NextLoadPtr - WorkingBlock;

    /* Calculate the number of bytes in this block that could
       possibly be the start of a matching sequence that lies
       entirely in this block (sequences that run off the end of
       the block will be transferred to the next block and found
       when that block is searched)

    */

    if ( (BlockSearchLength =

        WorkingLength - SearchStringLength + 1) <= 0 ) {

        Done = 1; /* В этом блоке слишком мало символов для
                   совпадений быть не может, так что это
                   последний блок, без которого мы закончили
                   нахожу совпадение

        */

    else {

        /* Выполнить поиск в этом блоке */

        if ( строка поиска(WorkingBlock, BlockSearchLength,
            Строка поиска, SearchStringLength)) {

            Найдено = 1; /* Мы нашли совпадение */

            Готово = 1;

        else {

            /* Скопируйте любые байты из конца блока, которые начинаются
               потенциально совпадающие последовательности, которые могли бы заканчиваться
               от конца блока к следующему блоку */

            if ( SearchStringLength > 1 ) {

                метсру (рабочий блок,

                    Рабочий блок+РАЗМЕР БЛОКА - длина строки поиска + 1,
                    SearchStringLength - 1);

                /* Настроен для загрузки следующих байтов из файла после
                   байты, скопированные с конца текущего блока */

                NextLoadPtr = WorkingBlock + SearchStringLength - 1;

                NextLoadCount = BLOCK_SIZE - длина строки поиска + 1;

            } пока ( !Готово);

            /* Сообщить о результатах */

            if ( Найдено) {

                printf("Строка
                найдена\n"); } else {

                printf("Строка не найдена\n");

            выход (Найден); /* Возвращает статус найдено/не
                               найдено в качестве

                               Уровень ошибки DOS */

            }

        }

    }

}
```

<Интерпретация того, куда идут циклы

Чтобы повысить общую производительность листинга 5.1, я бы обычно преобразовал SearchForString() на этом этапе в язык ассемблера. Однако я не собираюсь этого делать и причина тому - такой же важный урок, каким, вероятно, будет любое обсуждение оптимизированной сборки кода. Уделите минутку рассмотрению некоторых интересных аспектов производительности реализация на C, и все должно стать намного понятнее.

Как вы помните из главы 1, одно из важных правил оптимизации включает знание того, когда оптимизация вообще стоит того, чтобы ею заниматься. Другое правило включает в себя понимание того, на что уходит большая часть времени выполнения программы. Это более верно для Листинг 5.1 лучше, чем вы могли подумать.

Когда листинг 5.1 выполняется в исходном файле сборки размером 1 МБ, требуется около трех секунд, чтобы найдите строку “xxxend” (которая находится в конце файла) на компьютере с частотой 20 МГц 386, с весь файл находится в дисковом кэше. Если уменьшить размер БЛОКА с 16K до 4K, время выполнения заметно не увеличивается! При 2K программа немного замедляется; это происходит только после того, как блок размер уменьшается до 64 байт, а время выполнения становится примерно вдвое больше, чем у Буфер размером 16 КБ.

Итак, первое, что мы обнаружили, это то, что, хотя большие блоки действительно делают лучше производительности, прирост производительности может быть не очень большим и не оправдывать дополнительная память, необходимая для этих блоков большего размера. Наше следующее открытие заключается в том, что даже

хотя мы считываем файл большими фрагментами, большая часть времени выполнения листинга 5.1 тем не менее, тратится на выполнение функции read().

Когда я заменил вызов функции read() в листинге 5.1 кодом, который просто вводит в заблуждение программа думает, что считывается файл размером 1 МБ, программа работала почти мгновенно—менее чем 1/2 секунды, даже когда искомая строка не везде можно найти. Напротив, в листинге 5. Для выполнения функции 1 требуется три секунды, даже если выполняется поиск отдельного символа, который нигде не найден в файле, случай, когда один вызов memchr () (и, следовательно, один REPNZ SCASB) может устранить весь блок одновременно .

В целом, время, необходимое для вызовов доступа к диску DOS, занимает не менее 80 процентов время выполнения, а время поиска составляет менее 20 процентов от общего времени выполнения. Фактически, время поиска, вероятно, намного меньше 20 процентов от общего количества, учитывая, что накладные расходы на загрузку программы, выполнение кода запуска C, открытие файла, выполнение printf(), выход из программы и возврат в оболочку DOS также являются включено в мои тайминги. Учитывая это, должно быть очевидно, почему преобразование в ассемблер язык не стоит таких хлопот — лучшее, что мы могли бы сделать, ускорив поиск, - это улучшение примерно на 10 процентов, и для этого потребуется более чем удвоение производительности кода, который уже использует повторяющиеся строковые инструкции для выполнения большей части работы.

Маловероятно.

Знать, когда сборка бессмысленна

Вот почему мы не собираемся переходить на язык ассемблера в этом примере — а это не так сказать, что никогда не стоило бы переводить поисковую систему из листинга 5.1 в ассемблер.

Если, например, ваше приложение обычно выполняет поиск в буферах, в которых первый символ строки поиска встречается часто, как это может быть при поиске в текстовом буфере для строки, начинающейся с пробела, реализация на ассемблере может быть в несколько раз быстрее. Почему? Поскольку ассемблерный код может переключаться с REPNZ SCASB на сопоставьте первый символ с REPZ CMPS, чтобы проверить остальные символы всего за несколько Инструкции.

Напротив, листинг 5.1 должен возвращать данные из memchr (), настраивать параметры и вызывать memstr() чтобы сделать то же самое. Аналогично, assembly может переключиться обратно на REPNZ SCASB после несоответствие намного быстрее, чем в листинге 5.1. Накладные расходы на переключение высоки; когда поиск строки “zy” в файле, полностью заполненном символом z, в листинге 5.1 занимает почти 1/2 минуты, или почти на порядок дольше, чем при поиске в файле заполняется обычным текстом.

Возможно, также стоило бы преобразовать поисковую систему в ассемблер для поиска, выполняемого полностью в памяти; устранены накладные расходы на доступ к файлам, улучшен поиск- производительность движка напрямую привела бы к значительному повышению общей производительности. Одним из таких приложений, структура которого была бы во многом такой же, как в листинге 5.1, было бы поиск через расширенные буферы памяти, а другой будет выполнять поиск через огромные (охватывающие сегменты) буферы.

И вот мы обнаруживаем, как это часто бывает, что оптимизация определенно не является "разовым" вопросом, и что не существует такого понятия, как единый “наилучший” подход.



Вы должны знать, что обычно будет делать ваше приложение, и вы должны знать, какой вариант вас больше интересует - средний или наихудший производительность, прежде чем вы сможете решить, как наилучшим образом ускорить вашу программу — и, действительно, стоит ли вообще ускорять это.

Кстати, не думайте, что только потому, что очень большие размеры блоков не сильно улучшают ситуацию производительность, не стоило использовать перезапускаемые блоки из листинга 5.1. Листинг 5.1 выполняется более чем в три раза медленнее при размере блока 32 байта, чем при размере блока 4K, и любой побайтовый подход, несомненно, был бы еще медленнее из-за накладных расходов на повторяющиеся вызовы DOS и / или библиотеки ввода-вывода C stream.

Перезапускаемые блоки действительно минимизируют накладные расходы на вызовы доступа к файлам DOS в листинге 5.1; это просто нет способа уменьшить эти накладные расходы до такой степени, чтобы они стали стоящими попытка еще больше повысить производительность нашей относительно эффективной поисковой системы. Хотя поисковая система ни в коем случае не оптимизирована полностью, она, тем не менее, работает так же быстро, как для этого есть все основания, учитывая баланс производительности между компонентами этой программы.

Всегда смотрите, куда идет выполнение.

Я объяснил два важных урока: знайте, когда стоит продолжить оптимизацию, и используйте перезапускаемые блоки для обработки больших наборов данных в виде серии блоков, причем каждый блок обрабатывается на высокой скорости. Первый урок менее очевиден, чем кажется.

Когда я приступал к написанию этой главы, я полностью намеревался написать язык ассемблера версии листинга 5.1, и я ожидал, что версия на ассемблере будет намного быстрее. Когда я на самом деле посмотрел, куда уходит время выполнения (что я и сделал, изменив программу чтобы удалить вызовы функции read(), но для выполнения можно было бы использовать профилировщик кода то же самое гораздо проще), я обнаружил, что лучший код в мире не имел бы большого значения большая разница.



Когда вы пытаетесь ускорить код, найдите время, чтобы определить горячие точки в вашей программе, чтобы вы знали, где требуется оптимизация и будет ли это иметь существенное значение до того, как вы потратите свое время.

Что касается перезапускаемых блоков: здесь мы рассмотрели значительно более сложное применение перезапускаемые блоки, чем мы делали в главе 1, что, в конце концов, оказалось не таким сложным. В задачах программирования, которые вы решаете, не допускайте нарушений, таких как строки, которые охватывают блоки, побуждают вас соглашаться на простые, общие — и медленные-решения. Сосредоточьтесь на затем сделайте внутренний цикл — код, который обрабатывает каждый блок, — максимально эффективным структурируйте остальной код для поддержки внутреннего цикла.

Программирование с перезапускаемыми блоками непросто, но когда возникает проблема со скоростью, используя перезапускаемые блоки в нужных местах с лихвой окупают себя благодаря значительно улучшенной Производительность. И когда скорость, конечно, не является проблемой или в коде на это нет времени - важно то, что вам и в голову не придет тратить свое время на оптимизацию.

Не так ли?

Глава 6 – Взгляд за пределы номинальной стоимости

Как машинные инструкции могут делать больше, чем вы думаете

Впервые я встретил Джеффа Дантеманна на авторском ужине, организованном PC Tech Journal осенью Comdex, в далеком 1985 году. Джефф уже был достаточно известен как компьютерный редактор и писатель, хотя и не так известен, как Complete Turbo Pascal, выпуски с 1 по 672 (или около того), TURBO TECHNIX и компьютерные ТЕХНОЛОГИИ вскоре сделали бы его. Я мне посчастливилось сидеть рядом с Джеффом за обеденным столом, и, что неудивительно, наши оживленные разговоры часто вращались вокруг компьютеров, компьютерного письма и больше компьютеров (не обязательно в таком порядке).

Хотя в то время я зарабатывал на жизнь компьютерной работой и получал от нее удовольствие, я тем не менее, когда я вырос, у меня были смутные амбиции стать писателем-фантастом. (С тех пор я понял, что это вряд ли помещает меня в элитную компанию, особенно в компьютерный мир, где, кажется, каждый второй человек говорил мне, что планирует писать “когда-нибудь” научную фантастику. Учитывая, что, вероятно, меньше 500 — я предполагаю здесь — оригинальные научно-фантастические рассказы и фэнтези и, возможно, еще несколько романов, чем ежегодно в этой стране публикуются, я предвижу несколько кризисов среднего возраста.)

Во всяком случае, у меня накопилась небольшая коллекция отказных листков, и я воображал себя кем-то вроде опытного специалиста в этой области. В конце ужина, когда другие авторы полусерьезно жаловались на то, как мало им платят за работу в Tech Journal, я наклонился к Джеффу и прошептал: “Знаешь, платят здесь не так уж плохо”. Ты должен посмотреть, сколько они платят за научную фантастику — даже ребятам, которые получают награды!”

На что Джефф ответил: “Я знаю. Я был номинирован на два ”Хьюго”.

О.

Если бы я знал, что сижу рядом с настоящим, живым писателем-фантастом, номинированным на премию писатель, ей-богу!—Я бы выкачал из него все, что было в моих силах, но такая возможность мне и в голову не приходила. Я был на ужине, организованном компьютерным журналом, сидел рядом с редактор, который только что закончил книгу о Turbo Pascal, и, черт возьми, было очевидно что подходящей темой были компьютеры.

На этот раз мораль не “не суди о книге по обложке”. Джефф на самом деле тот, кем он казался на первый взгляд: компьютерный писатель и редактор. Однако он тоже нечто большее; номинальная стоимость не была полной. Вы также обнаружите, что номинальная стоимость не всегда является полной стоимостью в компьютерное программирование, особенно при работе на языке ассемблера, где многие инструкции обладают талантами, превосходящими их очевидные способности.

С другой стороны, существует также ряд инструкций, таких как LOOP, которые предназначены для выполнения определенных функций, но не всегда являются лучшими инструкциями для этих функций. Так что не судите о книге и по ее обложке.

Язык ассемблера для семейства x86 не похож ни на один другой язык (за что мы должны, без колебаний выразить нашу огромную благодарность). Язык ассемблера отражает дизайн процессора, а не то, как мы думаем, поэтому он полон множества инструкций, которые выполнять похожие функции, инструкции со странными и часто сбивающими с толку побочными эффектами и бесконечные способы объединения различных инструкций для выполнения практически одних и тех же действий, часто с кажущимися незначительными различиями, которые могут оказаться на удивление важными.

Для создания наилучшего кода, вы должны точно решить, что вам нужно выполнить, затем составьте последовательность инструкций, которая позволяет достичь этой цели наиболее эффективно, независимо от того, для чего обычно используются инструкции. Вот почему оптимизация для ПК - это искусство, и именно поэтому лучшим языком ассемблера для семейства x86 будет почти всегда превосходящий скомпилированный код. Имея это в виду, давайте не будем придавать этому значения — и раз уж мы об этом заговорили, я приведу несколько примеров того, как не судить о книге по ее обложке.

Суть всего этого: вы должны относиться к инструкциям семейства x86 с пониманием того, что они делают, а не того, что вы привыкли думать, что они делают. Да, SHL сдвигает шаблон влево — но поисковая система таблица может делать то же самое, и часто быстрее. ADD действительно может добавить два операнды, но он не может поместить результат в третий регистр; LEA может. Набор команд - это ваш исходный материал для написания высокопроизводительного кода. Ограничивая себя только размышлениями о различных инструкциях определенными, устоявшимися способами, вы ставите себя в существенное неудобство каждый раз, когда садитесь программировать.

Короче говоря, семейство x86 может сделать гораздо больше, чем вы думаете, — если вы будете использовать все, что оно может предложить. Попробуйте!

Много лет назад я увидел клип "Шоу Дэвида Леттермана", в котором Леттерман зашел в магазин “Just Lamps” и спросил: “Так что вы здесь продаете?”

“Лампы”, - сказали ему. “Просто лампы”. “Ты что, читать не умеешь”?

“Лампы”, - сказал он. “Понятно. А что еще?”

Из этого фрагмента возвышенного идиотизма мы можем многое узнать о том, как определить полную ценность инструкции. А именно:

Вкратце, что делают режимы адресации памяти x86?

“Вычислить адреса памяти”, - без сомнения, ответили вы. И вы, конечно, правы. Но что еще они делают?

Они выполняют арифметические действия, вот что они делают, и это совершенно другой и часто ошибочный подход полезный взгляд на вычисления адресов памяти.

Например, предположим, что у вас есть базовый адрес массива в BX и индекс в массив в SI. Вы могли бы сложить два регистра вместе для адресации памяти, вот так:

```
>add bx, si
mov al,[bx]
```

<Или вы могли бы позволить процессору выполнить арифметические действия за вас с помощью одной инструкции:

```
>mov al,[bx+si]
```

<Эти два подхода функционально взаимозаменяемы, но не эквивалентны с точки зрения производительности, и какой из них лучше, зависит от конкретного контекста. Если это одноразовый доступ к памяти, лучше всего позволить процессору выполнить сложение; это обычно это выполняется быстрее, чем отдельная команда ADD . Если это память доступ в цикле, однако, процессору 8088 выгодно выполнять добавление вне цикла, по возможности, сокращающее эффективное время вычисления адреса внутри цикл, как показано ниже:

```
Добавить      bx, si
LoopTop:
mov          al, [bx]
inc          bx
цикл          LoopTop
```

Здесь перемещение AL, [BX] на два цикла быстрее, чем перемещение AL, [BX + SI].

Однако на 286 или 386 баланс смещается. ДВИЖЕНИЕ AL,[BX + SI] занимает не больше времени, чем ПЕРЕМЕЩЕНИЕ AL, [BX] на этих процессорах, потому что эффективные вычисления адресов обычно не занимают вообще никакого дополнительного времени. (Согласно руководству MASM, требуется один дополнительный тактовый сигнал, если используются три компонента адресации памяти, как в MOV AL, [BX + SI + 1]. Я не был смог подтвердить это из публикаций Intel, но тогда я не особо усердствовал.) Если если вы оптимизируете для 286 или 386, то можете воспользоваться преимуществами процессора способностью выполнять арифметические вычисления как часть вычислений адресов памяти без ущерба для производительности.

486 - странный случай, в котором использование индексного регистра или базового регистр, являющийся местом назначения предыдущей инструкции, может замедлить работу, так что это так обычно, но не всегда, лучше выполнять сложение вне цикла на 486. Все однако вычисления адресации памяти бесплатны на Pentium. Я расскажу о 486 проблемы с производительностью в главах 12 и 13 и Pentium в главах с 19 по 21.

Вероятно, вас не особо удивило известие о том, что вы можете использовать режимы адресации для выполнения арифметики адресации памяти, которую в противном случае пришлось бы выполнять с помощью отдельных арифметических инструкций. Однако вам, возможно, будет немного интереснее услышать что вы также можете использовать режимы адресации для выполнения арифметических действий, которые не имеют ничего общего с адресацией памяти и с парой преимуществ перед арифметическими инструкциями, при этом.

Как?

С LEA единственная инструкция, которая выполняет вычисления адресации памяти, но не фактически адресует память. LEA принимает стандартный операнд адресации памяти, но не делает ничего, кроме сохранения вычисленного смещения памяти в указанном регистре, который может быть любым регистром общего назначения. Работа LEA проиллюстрирована на рисунке 6.1, который также показывает работу ADD от регистра к регистру, для сравнения.

Что это нам дает? Две вещи, которые ADD не предоставляет: возможность выполнять сложение с двумя или тремя операндами и возможностью сохранения результата в любом регистре, а не только в одном из исходных операндов.

Представьте, что мы хотим добавить BX к DI, добавить два к результату и сохранить результат в AX. Очевидное решение таково:

```
>mov ax,bx
add ax,di
add ax,2
```

<(Было бы более компактно увеличить AX в два раза, чем добавлять к нему два, и это вероятно, это будет быстрее на 8088, но это не то, что нам нужно в данный момент.)
элегантное альтернативное решение просто:

```
>lea ax,[bx+di+2]
```

<Аналогично, любое из следующих действий скопировало бы SI плюс два в DI

```
>mov di,si
add di,2
```

<или:

```
>lea di,[si+2]
```

<Имейте в виду, что единственными компонентами, которые LEA может добавить, являются BX или BP, SI или DI, и константа смещение, поэтому в большинстве случаев оно не заменит ADD. Кроме того, LEA работает значительно медленнее, чем ADD на 8088, хотя это так же быстро, как ДОБАВИТЬ на 286 или 386, когда используется менее трех компонентов адресации памяти. LEA на 1 цикл медленнее, чем ПРИБАВЬТЕ к 486, если сумма двух регистров используется для указания памяти, но не медленнее, чем ПРИБАВЬТЕ к Pentium. Как на 486, так и на Pentium LEA также может быть замедлен за счет устранения блокировок.

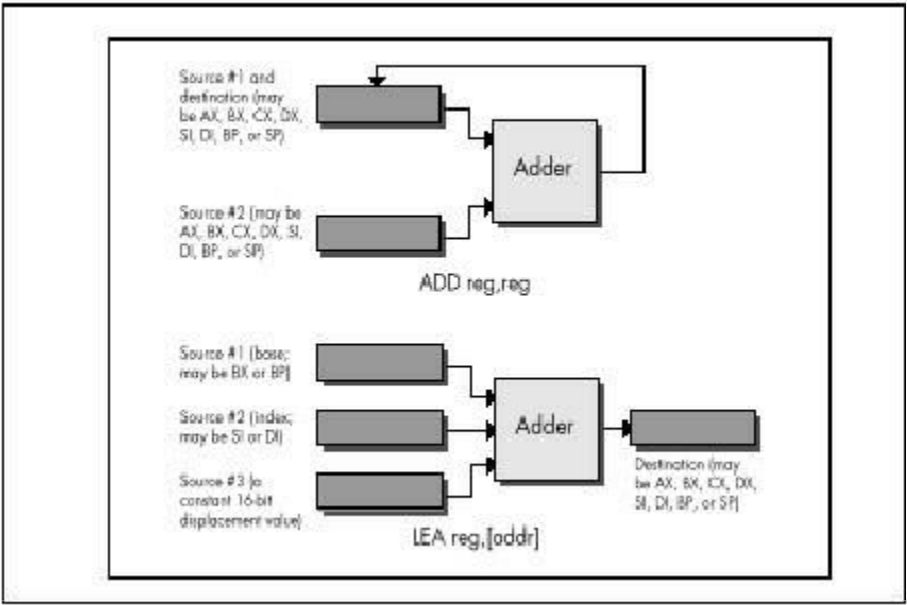


Рисунок 6.1 Работа ADD Reg, Reg по сравнению с LEA Reg, {Addr}.

LEA действительно является самостоятельной инструкцией “надстройки” на 386-м, 486-м и Pentium, где он может воспользоваться улучшенными режимами адресации памяти, присущими этим процессорам. (486 и Pentium предлагают те же режимы, что и 386, поэтому я буду ссылаться с этого момента только на 386.) 386 может делать две очень интересные вещи.: Он может использовать любой 32-разрядный регистр (EAX, EBX и т.д.) В качестве базового регистра адресации памяти и / или регистр индекса адресации памяти, и он может умножать любой 32-разрядный регистр, используемый в качестве индекса на два, четыре или восемь в процессе вычисления адреса памяти, как показано в Рисунок 6.2. Давайте посмотрим, для чего это нужно.

Что ж, очевидным преимуществом является то, что любые два 32-разрядных регистра или любой 32-разрядный регистр и любая константа или любые два 32-разрядных регистра и любая константа могут быть сложены вместе, при этом результат сохраняется в любом регистре. Это делает 32-разрядный LEA гораздо более полезным в общем плане чем стандартный 16-разрядный LEA в роли дополнения с независимым назначением.

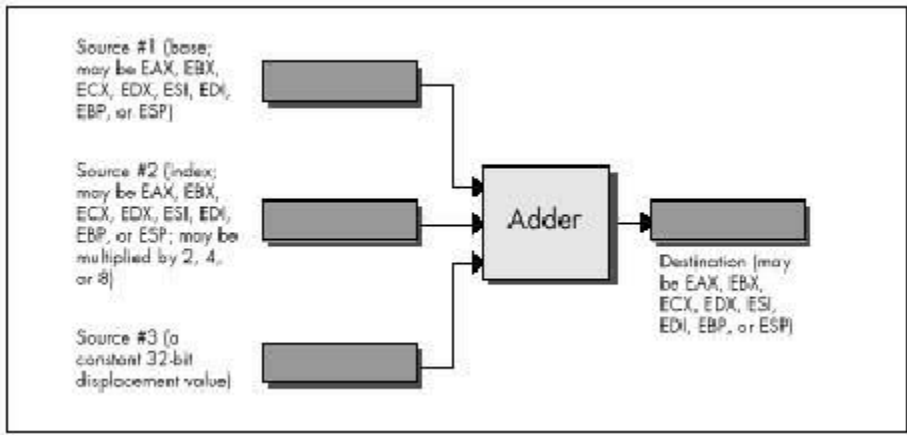


Рисунок 6.2 Работа 32-разрядного LEA reg, [Addr].

Но что еще LEA может делать на 386, кроме add?

Он может умножать любой регистр, используемый в качестве индекса. LEA может умножаться только на степень двойки значения 2, 4 или 8, но это полезно чаще, чем вы можете себе представить, особенно когда работа с указателями на таблицы. Кроме того, умножение на 2, 4 или 8 равнозначно сдвигу влево из 1, 2 или 3 бит, так что теперь мы можем добавить до двух 32-битных регистров и константу, а также сдвиг (или умножьте) один из регистров до некоторой степени — и все это с помощью одной инструкции. Для пример,

```
>lea edi,TableBase[ecx+edx*4]
```

<заменяет все это

```
>mov edi,edx
shl edi,2
add edi,ecx
add edi,offset TableBase
```

<при указании на запись в таблице с двойной индексацией.

Умножение с помощью LEA с использованием степеней, отличных от двух

Вы все еще впечатлены всем, что LEA может сделать на 386? Хотите верьте, хотите нет, но еще одна функция все еще ждет нас. LEA действительно может выполнять быстрое умножение 32-разрядного регистра на некоторые значения, отличные от степеней двойки. Видите ли, один и тот же 32-разрядный регистр может быть и тем, и другим база и индекс на 386, и их можно масштабировать как индекс во время использования без изменений в качестве базы. Это означает, что вы можете, например, умножить EBX на 5 с помощью:

```
>lea ebx,[ebx+ebx*4]
```

<Без LEA и масштабирования умножение EBX на 5 потребовало бы либо относительно медленного MUL вместе с инструкцией настройки, либо двух или трех отдельных инструкций в том же духе из следующих

```
>movedx, ebx
shl ebx,2
add ebx,edx
```

<и в любом случае потребовало бы уничтожения содержимого другого реестра.

Умножение 32-битного значения на множитель, не равный степени двух, всего за 2 цикла - довольно изящный трюк, хотя он работает только на 386 или 486.



Полный список значений, на которые LEA может умножить регистр на 386 или 486 является: 2, 3, 4, 5, 8, и 9. Этот список не включает все множители, которые вы могли бы захотеть, но он охватывает некоторые часто используемые, и производительность трудно превзойти.

Я хотел бы выразить свою благодарность Дуэйн Стронгу из Metagraphics за его помощь в мозговой штурм используется для 386-й версии LEA и для указания сложностей с 486-ю таймингами команд.

Глава 7 – Локальная оптимизация

Оптимизация на полпути между алгоритмами и подсчетом циклов

Возможно, вы так не думаете, но о программировании производительности можно многому научиться из the Great Buffalo Sauna Fiasco. А именно:

Место действия - Буффало, штат Нью-Йорк, в разгар зимы, когда снега навалило на несколько футов глубиной. Четверо студентов колледжа, живущих в типичном студенческом жилье, промерзли до костей. На третьем этаже их дома, неизолированном и настолько холодном, что он непригоден для жилья, есть старинная ванная. В один сказочно холодный день приходит вдохновение.:

“Эй, мы могли бы превратить эту ванную в сауну”!

Начинается столпотворение. Кто-то спешит и покупает газовый обогреватель, и с значительным риском для жизни и конечностей подключает его к заброшенной, но все еще работающей газовой трубе, которая однажды топили плиту на третьем этаже. Кто-то еще берет листы пластика и застилает их стены ванной, чтобы сохранить влагу внутри, а еще один студент набирает полное ведро из камней. Оставшийся парень приносит несколько старых деревянных стульев и расставляет их сделайте скамейки по бокам ванной. Вуаля — мгновенная сауна!

Они включают газовый обогреватель, ставят перед ним ведро с камнями, закрывают дверь, снимают одежду и садятся попариться. Имейте в виду, в этой комнате еще нет 50 градусов По Фаренгейту, но газовый обогреватель гремит вовсю. Несомненно, впереди более теплые времена.

Действительно, они наступают. Температура поднимается до 55 градусов, затем 60, затем 63, затем 65 и наконец ползет до 68 градусов.

И на этом все заканчивается.

68 градусов - это тепло для неизолированного третьего этажа в Буффало в разгар зимы. Черт возьми, тепло. Однако для сауны не особенно тепло. В конце концов, кто-то признает очевидное и допускает, что, в конце концов, это могла быть глупая идея, и все соглашаются, выключают обогреватель и уходят, каждый, без сомнения, предлагая молчание спасибо, что они выбрались из этого без каких-либо инцидентов, потребовавших серьезной операции.

Итак, мы видим, что лучшая идея в мире может потерпеть неудачу либо из-за отсутствия надлежащего дизайна, либо достаточной мощности. Основной причиной фиаско Great Buffalo Sauna была нехватка мощности; газовый обогреватель был слишком маленького размера. Это аналогично попытке пишите программы, которые включают такие функции, как растровый текст и поиск по многосегментным буферам, без использования высокопроизводительного языка ассемблера. Любой КОМПЬЮТЕР язык может выполнять практически любую функцию, о которой вы только можете подумать — в конце концов. Этот обогреватель в конечном итоге помещение тоже нагрелось бы до 110 градусов — примерно к первому числу Июня или около того.

Фиаско Great Buffalo Sauna также пострадало от фундаментальных недостатков дизайна. Подробнее мощный обогреватель действительно сделал бы комнату более горячей — и вполне мог сжечь в процессе дом дотла. Аналогичным образом, правильный выбор алгоритма и хороший дизайн имеют фундаментальное значение для производительности. Дополнительная мощность и превосходный язык ассемблера реализация дает программу, с которой стоит возиться только в контексте хорошего Дизайна.



Оптимизация языка ассемблера - небольшой, но важный аспект работы с ПК мир программирования. Используйте его экономно и только в рамках хороший дизайн — но проигнорируйте его, и вы можете обнаружить, что различные части вашего тела анатомия находится на холоде.

Итак, черпая силу духа в знании того, что наши поиски чисты и достойны, давайте продолжим наше исследование инструкций на языке ассемблера со скрытыми талантами и инструкции с хорошо известными талантами, которых меньше, чем кажется. В процессе, мы увидим, что существует еще один, очень важный уровень оптимизации между уровень алгоритма / проектирования и уровень подсчета циклов / индивидуальных инструкций. Я буду называть это локальная оптимизация среднего уровня; она предполагает сосредоточение внимания на оптимизации последовательностей инструкции, а не отдельные инструкции, все с прицелом на реализацию проектов максимально эффективно, учитывая возможности набора инструкций семейства x86.

И да, на случай, если вам интересно, вышеприведенная история действительно правдива. Был ли я там? Позвольте мне сформулирую это так: Если бы был, я бы никогда этого не признал!

Когда ЦИКЛ - плохая идея

Давайте сначала рассмотрим инструкцию, которая меньше, чем кажется: ЦИКЛ. Там нет загадка в том, что делает ЦИКЛ; он уменьшает CX и разветвляется, если CX не уменьшается до нуля. Это так прекрасно подходит для задачи обратного отсчета циклов, что любой опытный программист x86 инстинктивно вводит количество циклов в CX и обращается к LOOP при настройке цикла. Это прекрасно — ЦИКЛ, конечно, работает так, как рекламируется, — но есть одна проблема:



На половине процессоров семейства x86 LOOP работает медленнее, чем DEC CX за ним следует JNZ. (Конечно, DEC CX / JNZ не совсем эквивалентен LOOP, потому что DEC изменяет флаги, а LOOP - нет, но в большинстве ситуаций они сопоставимы.)

Как это может быть? Не спрашивайте меня, спросите Intel. На 8088 и 80286 цикл действительно быстрее чем DEC CX / JNZ на цикл, а цикл, как правило, еще немного быстрее, потому что это байт короче и поэтому может быть извлечен быстрее. Однако на 386-м все меняется; ЦИКЛ состоит из двух циклов медленнее, чем DEC / JNZ, и время выборки для одного дополнительного байта даже для некешированного 386 обычно не имеет значения. (Помните, что 386 извлекает четыре байта инструкции со скоростью рор.) ЦИКЛ на три цикла медленнее, чем DEC / JNZ на 486, и 486 выполняет инструкции за такое небольшое количество циклов, что эти три цикла означают, что DEC / JNZ почти в два раза быстрее, чем цикл. быстрее, чем цикл. Кроме того, в отличие от ЦИКЛА, DEC не требует использования CX, поэтому Решение DEC / JNZ работает быстрее и гибче на моделях 386 и 486, а также на Pentium в том числе. (Кстати, все это не просто теория; я рассчитал относительную производительность LOOP и DEC CX / JNZ на кэшированном 386, и LOOP действительно работает медленнее.)



Ситуация еще более странная для относительного JCXZ цикла, который разветвляется тогда и только если CX равен нулю. JCXZ быстрее, чем J CX, CX / JZ на 8088 и 80286 и эквивалентен на 80386, но примерно в два раза медленнее на

Кстати, не поддавайтесь на уловки JCXZ и сделайте что-нибудь вроде этого:

```
>and cx, 0fh; Isolate the desired field
jcxz SkipLoop; If field is 0, don't bother
```

<В инструкции AND уже установлен нулевой флаг, так что это

```
>and cx, 0fh; Isolate the desired field
jz...SkipLoop; If field is 0, don't bother
```

<будет работать просто отлично и быстрее на всех процессорах. Используйте JCXZ только тогда, когда нулевой флаг не установлен он уже установлен для отображения состояния CX.

Уроки LOOP и JCXZ

Чему мы можем научиться у LOOP и JCXZ? Во-первых, что одна инструкция, предназначенная для выполнения сложной задачи, не обязательно быстрее, чем несколько инструкций, которые вместе выполняют одно и то же. Во-вторых, относительные достоинства инструкций и правил оптимизации различаются в удивительно большой степени в семействе x86.

В частности, если вы собираетесь написать код защищенного режима 386, который будет выполняться только на 386, 486 и Pentium, вам было бы разумно пересмотреть свое использование более эзотерических членов набора инструкций x86. LOOP, JCXZ, различные инструкции для конкретных аккумуляторов и даже строковые инструкции во многих случаях больше не предлагают тех преимуществ, которые были у 8088. Иногда они просто ничуть не быстрее, чем другие общие инструкции, поэтому не стоит из кожи вон лезть, чтобы ими воспользоваться; иногда, как с LOOP они на самом деле медленнее, и вам было бы лучше вообще избегать их в мире 386/486. Для начала рекомендуется просмотреть продолжительность цикла команд в руководствах MASM или TASM или ознакомиться с продолжительностью цикла в литературе Intel; опубликованный цикл время выполнения ближе к фактическому времени выполнения на 386 и 486, чем на 8088, и является достаточно надежными показателями относительного уровня производительности инструкций x86.

Избегание ЦИКЛОВ любого уровня.

Подсчет циклов и прямая замена инструкций (например, DEC CX / JNZ для ЦИКЛА) это методы, которые относятся к самому низкому уровню оптимизации. Это важный уровень, но он довольно механический; как только вы изучите возможности и относительную производительность несмотря на различные уровни инструкций, вы должны быть в состоянии довольно легко выбрать лучшие. Более того, с этой задачей компиляторы справляются превосходно. Я хочу сказать, что вам не следует слишком увлекаться подсчетом циклов, потому что это небольшой (хотя и важный) это часть общей картины оптимизации, а не область, в которой заключается ваше наибольшее преимущество.

Локальная оптимизация

Одним из уровней, на котором программирование на языке ассемблера окупается сторицей, является уровень локальная оптимизация; то есть выбор наилучшей последовательности инструкций для задачи. Ключом к локальной оптимизации является просмотр набора инструкций 80x86 как набора строительных блоков, каждый из них обладает уникальными характеристиками. Ваша задача - расположить эти блоки так, чтобы они работали хорошо. Не имеет значения, для чего предназначены инструкции или каковы их имена есть; важно только то, что они делают.

Наше обсуждение ЦИКЛА в сравнении с DEC / JNZ - отличный пример оптимизации по циклам подсчет. Это стоит знать, но как только вы это усвоите, вы просто будете регулярно использовать DEC / JNZ в нижней части циклов в коде, специфичном для 386/486, и все. Кроме того, вы будете экономить по максимуму на нескольких циклах каждый раз, и, хотя это немного помогает, особой разницы это не принесет.

Теперь давайте на мгновение отступим назад и без предубеждений рассмотрим, что может сделать для нас x86 набор инструкций. Основная часть времени как с циклом, так и с DEC / JNZ отнимается за счет ветвления, которое, как оказалось, является одним из самых медленных аспектов каждого

процессор семейства x86, а остальное берется за счет уменьшения регистра count и проверки, равен ли он нулю. Могут быть способы выполнять эти задачи немного быстрее выбирая разные инструкции, но они могут выполняться только так быстро, а ветвление не может даже все это происходит так быстро.



Таким образом, хитрость заключается не в том, чтобы найти самый быстрый способ уменьшить количество и условно разветвляться, а скорее в том, чтобы выяснить, как добиться того же результат без частого уменьшения или ветвления. Помните

Проблему Кобияши Мару в "Звездном пути"? Здесь применим тот же принцип: переопределите переводы проблему в ту, которая предлагает лучшие решения.

Рассмотрим листинг 7.1, который выполняет поиск в буфере до тех пор, пока не будет найден указанный байт, найден нулевой байт или проверено указанное количество символов. Такая функция была бы полезна для сканирования максимального количества символов в буфере с нулевым завершением . Листинг 7.1, в котором используется цикл в основном цикле, выполняет поиск пример строки за период (") в 170 мкс на кэшированном 386-м канале 20 МГц.

Когда цикл в листинге 7.1 заменяется на DEC CX / JNZ, производительность повышается до 168 us, что менее чем на 2 процента быстрее, чем в листинге 7.1. Фактически, выборка инструкций, инструкция выравнивание, характеристики кэша или что-то подобное влияют на эти результаты; Я бы ожидал немного большего улучшения — примерно на 7 процентов — но это максимум, что подсчет циклов мог бы принести нам пользу в этом случае. (Уже все в порядке; LOOPNZ можно было бы использовать на нижняя часть цикла и другие оптимизации, безусловно, возможны, но все это не приведет к даже близко к тем преимуществам, которые мы собираемся увидеть от локальной оптимизации, и в этом весь смысл.)

ЛИСТИНГ 7.1 L7-1.ASM

```
>; Program to illustrate searching through a buffer of a specified
; length until either a specified byte or a zero byte is
; encountered.
; A standard loop terminated with LOOP is used.
.model small
.stack 100h
.data

; Sample string to search through.
SampleStringLabel byte
    db...'This is a sample string of a long enough length '
    db 'so that raw searching speed can outweigh any
    db 'extra set-up time that may be required.',0
SAMPLE_STRING_LENGTH equ $-SampleString
; User prompt. Prompt db...'Enter
character to search for:$'
; Result status messages.
ByteFoundMsg db...'0dh,0ah
                db...'Specified byte found.',0dh,0ah,'$'
                ZeroByteFoundMsg db 0dh, 0ah
                db...'Zero byte encountered.',0dh,0ah,'$'
NoByteFoundMsg db 0dh,0ah
                db...'Buffer exhausted with no match.', 0dh, 0ah, '$'

.code Start
proc near
    mov ax,@data;point to standard data segment
    mov ds,ax
    mov dx,offset Prompt
    mov ah,9;DOS print string function
    int 21h;prompt the user
    mov ah,1;DOS get key function
    int 21h;get the key to search for
    mov ah,al;put character to search for in AH
    mov cx, SAMPLE_STRING_LENGTH;# of bytes to search
    mov si,offset SampleString;point to buffer to search
    call SearchMaxLength;search the buffer
    mov dx,offset ByteFoundMsg;assume we found the byte
    jc...PrintStatus;we did find the byte
                                ;we didn't find the byte, figure out
                                ;whether we found a zero byte or
                                ;ran out of buffer

    mov dx,offset NoByteFoundMsg
                                ;assume we didn't find a zero byte
    jexz PrintStatus;we didn't find a zero byte
```

```
<>mov dx,offset ZeroByteFoundMsg;we found a zero byte
PrintStatus:
mov ah,9;DOS print string function
int 21h;report status
mov ah,4ch;return to DOS
int 21h
Start endp
; Function to search a buffer of a specified length until either a
; specified byte or a zero byte is encountered.
; Input:
; AH = character to search for
; CX = maximum length to be searched (must be > 0)
; DS:SI = pointer to buffer to be searched
; Output:
; CX = 0 if and only if we ran out of bytes without finding
; either the desired byte or a zero byte
; DS:SI = pointer to searched-for byte if found, otherwise byte
; after zero byte if found, otherwise byte after last
; byte checked if neither searched-for byte nor zero
; byte is found
; Carry Flag = set if searched-for byte found, reset otherwise
SearchMaxLength proc near
cld
SearchMaxLengthLoop:
lodsb;get the next byte
cmp al,ah;is this the byte we want?
jz ByteFound;yes, we're done with success
and al,al;is this the terminating 0 byte?
jz ByteNotFound;yes, мы закончили с ошибкой
цикл SearchMaxLengthLoop; это ни то, ни другое, поэтому проверьте следующий
;байт, если есть

ByteNotFound:
clc; возвращает статус "не найдено"
ret
ByteFound:
декабрьsi _; возвращаем к местоположению, в котором
;мы нашли искомый байт

stc;возвращаем "найдено" состояние
ret
Максимальная
длина поиска endp
конецНачало
```

<Разворачивание петель

Листинг 7.2 использует другой подход, разворачивая цикл таким образом, что проверяется наличие четырех байт каждый выполняемый ЦИКЛ. Внутри цикла в каждом листинге используются одни и те же инструкции, но Листинг 7.2 составлен таким образом, что три четверти ЦИКЛОВ исключены. Листинги 7.1 и 7.2 выполняют точно такую же задачу и используют одни и те же инструкции в цикле — алгоритм поиска никоим образом не изменился, но мы упорядочили инструкции в листинге 7.2 по-другому, и в этом вся разница.

ЛИСТИНГ 7.2 L7-2.ASM

```
>; Program to illustrate searching through a buffer of a specified
; length until a specified zero byte is encountered.
; A loop unrolled four times and terminated with LOOP is used.
.model small
.stack 100h
.data
; Sample string to search through.
SampleString label byte
db...'This is a sample string of a long enough length '
db...'so that raw searching speed can outweigh any '
db...'extra set-up time that may be required.',0
SAMPLE_STRING_LENGTH equ $-SampleString
; User prompt.
Prompt db'Enter character to search for:$'
; Result status messages.
ByteFoundMsgdb...0dh,0ah
db...'Specified byte found.',0dh,0ah,'$'
ZeroByteFoundMsg db 0dh,0ah
db...'Zero byte encountered.', 0dh, 0ah, '$'
NoByteFoundMsgdb...0dh,0ah
db...'Buffer exhausted with no match.', 0dh, 0ah, '$'
; Table of initial, possibly partial loop entry points for
; SearchMaxLength.
SearchMaxLengthEntryTable label word
dw _SearchMaxLengthEntry4
dw _SearchMaxLengthEntry1
```

```
SearchMaxLengthEntry2
дг _ SearchMaxLengthEntry3

.код
Начать прок рядом
movу ax,@data ; указывает на стандартный сегмент данных
movу ds, ax
movу запрос dx, смещение
movу ah, 9 ; Функция строки печати DOS
инт 21h ; запрашивает пользователя
movу >ah,1 <>;DOS get key function

<>int21h;get the key to search for
movah,al;put character to search for in AH
mov cx, SAMPLE_STRING_LENGTH;# of bytes to search
movsi,offset SampleString;point to buffer to search
call SearchMaxLength;search the buffer
movdx,offset ByteFoundMsg;assume we found the byte
jc...PrintStatus;we did find the byte

;we didn't find the byte, figure out
;whether we found a zero byte or
;ran out of buffer
mov dx,offset NoByteFoundMsg
;assume we didn't find a zero byte

jexz PrintStatus;мы не нашли нулевого байта
mov dx,смещение ZeroByteFoundMsg ; мы нашли нулевой байт

PrintStatus:
movah, 9; Строковая функция печати DOS
int21h; состояние отчета
перемещениеah, 4ch; возврат в DOS
int21ч

Запустите endp

; Функция для поиска в буфере заданной длины до тех пор, пока не будет найден
; указанный байт или нулевой байт.
; Входные данные:
; AH = символ для поиска
; CX = максимальная длина для поиска (должна быть > 0)
; DS: SI = указатель на буфер для поиска
; Вывод:
; CX = 0 тогда и только тогда, когда у нас закончились байты без поиска
; либо желаемый байт, либо нулевой
; DS: SI = указатель на искомый байт, если он найден, в противном случае байт
; после нулевого байта, если найден, в противном случае байт после последнего
; байт проверяется, если ни искомый байт, ни ноль
; байт не найден
; флаг переноса = устанавливается, если искомый байт найден, сбрасывается в противном случае

Процесс поиска максимальной длины рядом
cld
movbx, cx
Добавитьпример, 3; рассчитайте максимальное количество проходов
shrex, 1; через петлю, которая равна
shrex, 1; развернутую 4 раза
ibx,3; вычислите индекс для записи
; таблица точек для первого,
; возможно, частичный цикл
shlбх, 1;подготовьтесь к поиску размером со слово
jmp SearchMaxLengthEntryTable[bx]
; переход в развернутый цикл для выполнения
; первый, возможно, частичный цикл

SearchMaxLengthLoop:
SearchMaxLengthEntry4:
lodsб; получить следующий байт
стрэл, а; это тот байт, который нам нужен?
jz _ByteFound; да, мы успешно завершили
иал, ал; это завершающий байт 0?
jz ByteNotFound; да, мы закончили с ошибкой

SearchMaxLengthEntry3:
lodsб; получить следующий байт
стрэл, а; это тот байт, который нам нужен?
jz...ByteFound; да, мы успешно завершили
и все, все; это завершающий байт 0?
jz ByteNotFound; да, мы закончили с ошибкой

SearchMaxLengthEntry2:
lodsб; получаем следующий байт
сmp ал, ah; это тот байт, который нам нужен?
jz ByteFound;да, мы успешно завершили
и все, все; это завершающий байт 0?
jz ByteNotFound; да, мы закончили с ошибкой

SearchMaxLengthEntry1:
lodsб; получаем следующий байт
страл, ах; это тот байт, который нам нужен?
jz _ByteFoundда, мы завершили с успехом.,.,.,.,.
иал, ал; это завершающий 0-байт?
jz...ByteNotFoundда, мы закончили с ошибкой.....
цикл SearchMaxLengthLoop; это ни то, ни другое, поэтому проверьте следующее
; четыре байта, если есть

ByteNotFound:
clс; возвращает
статус "не найдено"
ret
```

```
<>ByteFound:
    decsi _;point back to the location at which
                ; we found the searched-for byte
    stc;return "found" status
    ret
SearchMaxLength endp
endStart
```

<Насколько велика разница? Листинг 7.2 выполняется за 121 мкс — на 40 процентов быстрее, чем листинг 7.1, несмотря на то, что листинг 7.2 по-прежнему использует цикл, а не DEC CX / JNZ. (Цикл в листинге 7.2 можно было бы развернуть и дальше; это просто вопрос того, сколько еще памяти вы хотите использовать обменять на постоянно снижающуюся производительность.) Это типично для локальной оптимизации; это не часто приводит к улучшениям на порядок, которые алгоритмические улучшения может производить, но может обеспечить вам критическое улучшение на 50 или 100 процентов, когда вы исчерпали все другие возможности.



Суть проста: вы можете получить гораздо больше, отступив немного назад и думая о том, как процессор быстрее всего выполнит задачу, чем вы можно, сохраняя цикл здесь или там, используя разные инструкции. Постарайтесь мыслить на уровне последовательностей инструкций, а не отдельных инструкций, и научитесь относиться к инструкциям x86 как к строительным блокам с уникальными характеристиками а не как к инструкциям, предназначенным для конкретных задач.

Вращение и перестановка с помощью таблиц

В качестве другого примера локальной оптимизации рассмотрим поворот или смещение маски в нужное положение. Сначала давайте рассмотрим простую задачу установки бита N в AX равным 1.

Очевидный способ сделать это - поместить N в CL, повернуть долото в нужное положение и / или его с помощью AX, следующим образом:

```
ПЕРЕМЕЩЕНИЕ BX, 1
SHL BX, CL
ИЛИ_ AX, BX
```

Это решение очевидно, потому что оно использует преимущества специальных возможностей семейства x86 для сдвига или поворота на переменное количество битов, указанное CL. Однако на 8088 это занимает в среднем около 45 циклов. На самом деле предварительный расчет выполняется намного быстрее для получения результатов передайте номер бита в BX и просмотрите сдвинутый бит вверх, как показано в листинге 7.3.

ЛИСТИНГ 7.3 L7-3.ASM

```
>SHL BX,1;prepare for word sized look up
OR AX,ShiftTable[BX] ;Look up the bit and OR it in
ShiftTableLABEL WORD
BIT_PATTERN=0001H
    REPT 16
    DW BIT_PATTERN
    BIT_PATTERN=BIT_PATTERN SHL 1
    ENDM
```

<Несмотря на доступ к памяти, этот подход занимает всего 20 циклов — более чем в два раза быстрее, чем переменная shift. И снова мы смогли значительно повысить производительность — не за счет знания самых быстрых инструкций, а за счет выбора самой быстрой последовательности инструкций.

В конкретном примере выше мы снова сталкиваемся с трудностями оптимизации в семействе x86. Поиск по таблице выполняется быстрее на 8088 и 286, но это немного медленнее на 386 и не быстрее на 486. Однако код, специфичный для 386/486, может использовать улучшенная адресация для выполнения всей работы всего за одну инструкцию, в соответствии со строками фрагмента кода в листинге 7.4.

ЛИСТИНГ 7.4 L7-4.ASM

```
>OR EAX, ShiftTable [EBX*4] ; look up the bit and OR it in
```

```
<Таблица сдвига      МЕТКА      DWORD
    BIT_PATTERN=0001H
    REPT  32
        DD BIT_PATTERN
    BIT_PATTERN=BIT_PATTERN SHL 1
ENDM
```



Этот пример не только иллюстрирует преимущества локальной оптимизации, но и показывает, что предварительный расчет результатов обычно выгоден; это часто делается во время или до начала сборки, но предварительно рассчитанные таблицы также могут быть созданы во время выполнения во время выполнения. Это всего лишь один аспект фундаментального правила оптимизации: перемещайте как можно больше работайте над своим критически важным кодом, насколько это возможно, любыми необходимыми средствами.

NOT Переворачивает биты—флаги Not

Инструкция NOT переворачивает все биты в операнде с 0 на 1 или с 1 на 0. Это настолько просто, насколько могло бы быть, но, тем не менее, у NOT есть небольшой, но интересный талант: это не влияет на флаги. Это может раздражать; Однажды я потратил добрый час, отслеживая ошибку вызванную моим бессознательным предположением, что NOT устанавливает флаги. В конце концов, любой другой арифметическая и логическая инструкция устанавливает флаги; почему бы и нет? Вероятно, потому, что NOT вообще не является считается арифметической или логической инструкцией; скорее, это манипуляция данными инструкции, такие как mov и различные повороты. (Это RCR, RCL, ROR и ROL, которые влияют только на флаги переноса и переполнения.) НЕ часто используется для задач, таких как переключение масок, где нет причин проверять состояние результата, и в этом контексте это может будет удобно сохранить флаги неизмененными для последующего тестирования.



Кроме того, если вы хотите НЕ использовать операнд и установить флаги в процессе, вы можете просто XOR его с помощью -1. Другими словами, единственное функциональное различие между NOT AX и XOR AX, 0FFFFH заключается в том, что XOR изменяет флаги, а NOT

- нет.

Набор команд x86 предлагает множество способов выполнения практически любой задачи. Понимание тонких различий между инструкциями — установлены ли флаги для пример —может иметь решающее значение, когда вы пытаетесь оптимизировать последовательность кода и у вас заканчиваются регистры или когда вы пытаетесь минимизировать ветвление.

Увеличение с переносом и без переноса

Другой случай, когда существуют два немного отличающихся способа выполнения задачи, включает в себя добавление 1 к операнду. Вы можете сделать это с помощью INC, как в INC AX , или вы можете сделать это с помощью ADD, как в ADD AX, 1. В чем разница? Очевидное различие заключается в том, что INC обычно на байт или два короче (исключением является ADD AL,1 , который в двух байтах имеет одинаковую длину как INC и др.) и работает быстрее на некоторых процессорах. Менее очевидно, но не менее важно то, что ADD устанавливает флаг переноса, в то время как INC оставляет флаг переноса нетронутым.

Почему это важно? Потому что это позволяет INC функционировать как манипулятор указателями данных инструкция для арифметики из нескольких слов. Вы можете использовать INC для продвижения указателей в коде как показано в листинге 7.5, без необходимости выполнять какие-либо действия для сохранения статуса переноса от одного добавления к следующему.

ЛИСТИНГ 7.5 L7-5.ASM

```
>CLC;clear the Carry for the initial addition
LOOP_TOP:
MOVAX,[SI];get next source operand word
ADC[DI],AX;add with Carry to dest operand word
INCSI _;point to next source operand word
INCSI _
INCDI _;point to next dest operand word
INCDI _
LOOPLOOP_TOP
```

<Если бы использовался ADD, флаг переноса должен был бы сохраняться между добавлениями с кодом в соответствии со строками, показанными в листинге 7.6.

ЛИСТИНГ 7.6 L7-6.ASM

```
>CLC;clear the carry for the initial addition
LOOP_TOP:
MOV AX,[SI] ;get next source operand word
ADC [DI],AX ;add with carry to dest operand word
LAHF;set aside the carry flag
ADD SI,2 ;point to next source operand word
ADD DI,2;point to next dest operand word
SAHF;restore the carry flag
LOOP LOOP_TOP
```

<Не то чтобы подход, основанный на листинге 7.6, обязательно был лучше или хуже; это зависит от процессора и ситуации. Подход, приведенный в листинге 7.6, отличается, и если вы понимаете различия, вы сможете выбрать наилучший подход для любого кода, который вам доведется написать. (Кстати, DEC обладает тем же свойством сохранения флага переноса.)

В последнем примере есть пара интересных аспектов. Во-первых, обратите внимание, что ЦИКЛ вообще не влияет ни на какие флаги; это позволяет флагу переноса оставаться неизменным от одного добавление к следующему. Неизменность арифметических флагов является общей характеристикой инструкции программного управления (в отличие от арифметических и логических инструкций, таких как sub и AND, которые изменяют флаги).



Правило не в том, что арифметические флаги меняются всякий раз, когда процессор выполняет вычисление; скорее, флаги меняются всякий раз, когда вы выполняете арифметический, логический элемент управления или флажок (например, CLC для снятия флага переноса) инструкция.

Мало того, что LOOP и JCXZ не изменяют флаги, но и REP MOVS, который отсчитывает CX до 0, также не влияет на флаги.

Другим интересным моментом в последнем примере является использование LAHF и SAHF, которые передают младший байт регистра FLAGS в AH и из него соответственно. Эти инструкции были созданы для того, чтобы помочь обеспечить совместимость с 8080-ми (именно с 8080-ми, а не с 8088) Инструкции PUSH PSW и POP PSW, но они получаются компактными (один байт) инструкции по сохранению и восстановлению арифметических флагов. Однако хочу предостеречь: SAHF восстанавливает флаги переноса, нуля, знака, вспомогательного переноса и четности, но не Флаг переполнения, который находится в старшем байте регистра FLAGS. Также имейте в виду, что LAHF и SAHF обеспечивают быстрый способ сохранения флагов на 8088, но являются относительно медленными инструкциями на 486 и Pentium.

Бывают случаи, когда отсутствие флага переноса является очевидной ответственностью INC. Например,

```
INC AX
ADC DX, 0
```

не увеличивает 32-битное значение в DX: AX. Для этого вам потребуется следующее:

```
ДОБАВИТЬ AX, 1
АЦП DX, 0
```

Как всегда, обратите внимание!

Глава 8 – Ускорение C с помощью Языка Ассемблера

Переключение языков, когда вы знаете, что это поможет

Когда я был старшекласником, появилась поп-песня “Времена года на солнце” в исполнении некий Терри Джекс взлетел вверх по поп-чартам и провел, насколько я помню, две недели подряд на вершине американского топ-40 Кейси Касема. “Seasons in the Sun” не была особенно хорошей песней, в первую очередь потому, что текст был глупым. Я никогда не понимал, почему песня стала хитом, но, как это часто бывает с ничем не примечательной, но популярной музыкой забытые одно- или двухсерийные группы (“Не вымещай на мне свою любовь”, “Билли не Будь героем” и др.), я слышал это повсюду в течение месяца или около того, а потом больше об этом не думал думал 15 лет.

Недавно, однако, я наткнулся на обзор малоизвестной коллекции Rhino Records поп-хиты 1970-х годов. Зная, что Джефф Дантеманн является поклонником подобной эзотерики (который вы знаете, кому принадлежит альбом Peppermint Trolley Company?), я отправил ему рецензию. Его это позабавило, и, пока мы перебирали названия старых песен, всплыла “Seasons in the Sun”. Я выразил свое удивление, что песня, которая действительно "было не очень хорошо" имело такой большой успех.

“Ну”, - сказал Джефф, “Я думаю, что это пострадало при переводе с французского”.

Ах-ха! Тайна раскрыта. Очевидно, все, кроме меня, знали, что это был перевод с Французского, и эта новизна, несомненно, сделала песню большим хитом. Перевод также был несомненно, это ответственно за сочные тексты; спорю с dollars to donuts, что оригинальные французские тексты были сильнее.

Что подводит нас к теме этой главы, ускоряя C с помощью язык ассемблера. Если вы хотите ускорить работу программы на C путем преобразования выбранных ее частей (обычно не более нескольких функций) на язык ассемблера, убедитесь, что в итоге вы получаете высокопроизводительный код на языке ассемблера, а не отлаженный код на C. Компиляторы, такие как Microsoft C / C ++ и Watcom C, к настоящему времени довольно хороши в тонкой настройке C код, и вы вряд ли добьетесь большего успеха, взяв сборку компилятора языковой вывод и его настройка.



Чтобы процесс перевода кода C на язык ассемблера стоил того проблемы, вы должны игнорировать то, что делает компилятор, и разрабатывать свой код на языке ассемблера с точки зрения чистого языка ассемблера. C просто адекватным переводом вы рискуете изрядно потрудиться за небольшое вознаграждение или вообще без него.

Кстати, когда ты в последний раз слышал о Терри Джексе?

Билли, не будь компилятором.

Ключом к оптимизации программ на C с использованием языка ассемблера, как всегда, является хорошее написание код на языке ассемблера, но с добавлением изюминки. Правило 1 при преобразовании кода C в ассемблер таково: не думайте как компилятор. Это легче сказать, чем сделать, особенно, когда код C, который вы конвертируете, легко доступен в виде модели, и также доступен ассемблерный код, который генерирует компилятор. Тем не менее, принцип не думать как компилятор важен и в той или иной форме является основой для всего, что я обсужу ниже.

Прежде чем я продолжу обсуждение правила 1, позвольте мне упомянуть правило № 0: оптимизируйте только там, где это важно. Основная часть времени выполнения в любой программе тратится на очень небольшую часть код, и большая часть кода за пределами этой небольшой части не оказывает какого-либо заметного влияния на Производительность. Если только вас в высшей степени не беспокоит размер кода (область, в которой программы, работающие только на ассемблере, могут работать в Excel), Я бы посоветовал вам написать большую часть вашего кода на C и зарезервировать ассемблер для действительно важных разделов вашего кода; это формула, которую я поиск дает наибольшую отдачу.

Это не означает, что полные программы не следует разрабатывать с учетом оптимизации

языка ассемблера. Как вы вскоре увидите, ориентируя свои структуры данных на язык ассемблера действительно может быть полезным занятием, даже если большая часть вашего кода написана на С. Однако, когда дело доходит до реальной оптимизации кода и / или преобразования его в ассемблер, делайте это только там, где это важно. Приобретите профилировщик — и используйте его!

Также возьмите за правило сосредоточиться на совершенствовании дизайна вашей программы и алгоритмики подходите к работе на концептуальном уровне и / или С, прежде чем приступать к работе на любом языке ассемблера оптимизация.



Оптимизация на языке ассемблера - последний и далеко не единственный шаг в цепочка оптимизации, и как таковая, должна выполняться последней; преобразование

слишком быстрое преобразование в ассемблер может привести к блокировке вашего кода до того, как дизайн станет оптимальным. В по крайней мере, преобразование в ассемблер имеет тенденцию усложнять будущие изменения и отладку это замедляет работу и ограничивает ваши возможности.

Не вызывай у меня свои функции, детка

Чтобы мыслить иначе, чем компилятор, вы должны понимать, что обычно делают компиляторы и программисты на С, и чем это отличается от того, что делает язык ассемблера хорошо. В этом процессе может быть полезно изучить код, генерируемый вашим компилятором, либо просматривая код в отладчике, либо заставляя компилятор генерировать ассемблер выходной файл на языке. (Последнее выполняется с помощью /Fa или /Fc в Microsoft C / C ++ и -S в Borland C ++.)

Программисты на С склонны модулировать свой код множеством вызовов функций. Это хорошо для читаемый, надежный, повторно используемый код, и это позволяет компилятору оптимизировать лучше, потому что это может работать с меньшим количеством переменных и операторов в каждой области оптимизации — но это не так хорошо, если смотреть с уровня языка ассемблера. Вызовы и возвраты выполняются медленно, особенно в модели с большим кодом, и операции, необходимые для размещения параметров в стеке, также дороги.

Это означает, что когда вы хотите ускорить часть программы на С, вы должны определить всю критическую часть и переместить всю эту критическую часть в функцию языка ассемблера. Вы же не хотите перемещать часть внутреннего цикла на язык ассемблера, а затем вызывать его из С каждый раз в цикле; вызов функции и возврат накладных расходов были бы неприемлемы. Массово выделите критичный код и перенесите его в ассемблер и постарайтесь избегать вызовов и возвратов даже в вашем ассемблерном коде. Правда, в ассемблере вы можете передавать параметры в регистрах, но вызовы и возвраты сами по себе по-прежнему выполняются медленно; если дополнительные циклы, которые они выполняют, не влияют на производительность, то код, в котором они находятся, вероятно, не критичен, и, возможно, вы решили преобразовать слишком много код в ассемблер, не так ли?

Фреймы стека сильно замедляют работу

Компиляторы С работают в рамках модели фрейма стека, посредством чего переменные находятся в блоке стековая память и доступ к ним осуществляется через смещения из BP. Компиляторы могут хранить несколько переменные в регистрах и могут ненадолго сохранять другие переменные в регистрах, когда они используются неоднократно, но фрейм стека является базовой архитектурой. Это хороший архитектура; она гибкая, удобная, ее легко программировать, и она довольно компактна код. Однако фреймы стека имеют несколько недостатков. Они должны быть сконструированы и уничтожены, что требует как времени, так и кода. Они настолько просты в использовании, что имеют тенденцию смещать программиста на ассемблере в пользу доступа к переменным памяти чаще, чем может быть необходимо. Наконец, вы не можете использовать BP в качестве регистра общего назначения если вы собираетесь получить доступ к фрейму стека, и наличие этого седьмого доступного регистра иногда действительно полезно.

Это не значит, что вы не должны использовать фреймы стека, которые полезны и часто

Разрываясь между двумя сегментами

Компиляторы С не очень хороши в обработке сегментов. Некоторые компиляторы могут эффективно обрабатывать один дальний указатель, используемый в цикле, оставляя ES установленным на время цикла. Но два указатели far, используемые в одном и том же цикле, сбивают с толку каждый компилятор, который я видел, в результате чего полный адрес segment:offset перезагружается каждый раз, когда используется любой указатель.



Это особенно влияет на производительность в защищенном режиме 286 (например, в OS / 2 1.X или расширителе Rational DOS), поскольку сегмент загружается в защищенный режим занимает минимум 17 циклов по сравнению с всего лишь 2 циклами в реальном режиме.

На языке ассемблера у вас есть полный контроль над сегментами. Используйте его и, при необходимости, реорганизуйте свой код, чтобы минимизировать загрузку сегмента.

Почему ускорить работу сложно.

Вы можете подумать, что наиболее очевидным преимуществом языка ассемблера перед С является то, что он позволяет использовать все формы инструкций и все регистры всеми способами, в то время как С компиляторы, как правило, используют подмножество регистров и инструкций ограниченным числом способов. Да и нет. Это правда, что компиляторы С обычно не генерируют такие инструкции, как XLAT, rotates или строковые инструкции. С другой стороны, XLAT и rotates полезны в ограниченном наборе обстоятельств, а строковые инструкции используются в функциях библиотеки С. Фактически, код библиотеки С, вероятно, будет тщательно оптимизирован экспертами и может быть намного лучше, чем эквивалентный код, который вы создали бы сами.

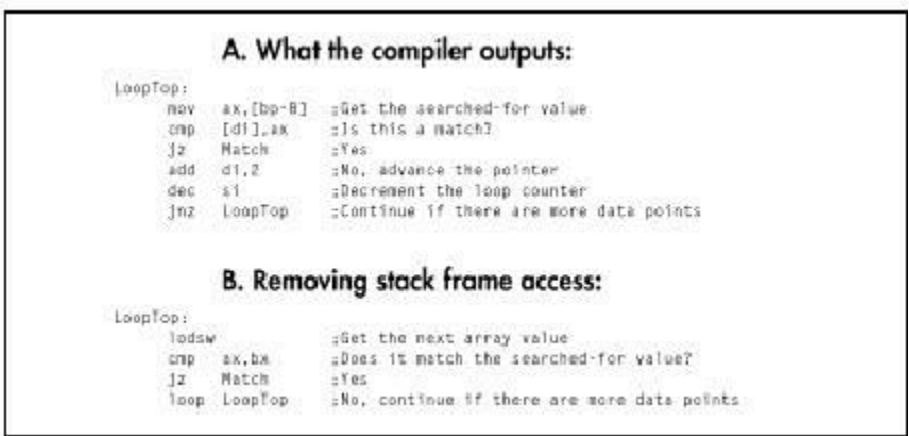
Я хочу сказать, что компиляторы С создают лучший код, чем вы? Нет, я говорю, что они могут, если вы не используете язык ассемблера должным образом. Написание кода на языке ассемблера вместо С ничего не гарантирует.



Вы можете написать хорошую сборку, плохую сборку или сборку, которая практически неотличима от скомпилированного кода; скорее всего, вы этого не сделаете напишите последнее, если вы считаете, что оптимизация заключается в настройке скомпилированный код на С.

Конечно, вы, вероятно, сможете использовать регистры более эффективно и воспользоваться пара инструкций, которые пропустил компилятор, но код не станет намного лучше таким образом, он станет быстрее.

Настоящая оптимизация требует переосмысления вашего кода, чтобы использовать преимущества ассемблера язык. Цикл С, который выполняет поиск совпадений в целочисленном массиве, может скомпилироваться.



На рисунке 8.1 Изменен вывод компилятора для цикла.

примерно до рисунка 8.1A. Вы можете посмотреть на это и настроить его в соответствии с кодом, показанным на Рисунок 8.1B.

Поздравляем! Вы успешно удалили весь доступ к фрейму стека, который вы использовали

ЦИКЛ (хотя DEC SI / JNZ на самом деле работает быстрее на машинах 386 и более поздних версий, как я объяснял в предыдущей главе), и вы использовали строковую инструкцию. К сожалению, новый код не будет выполняться намного быстрее. Может быть, на 25 процентов быстрее, может быть, чуть больше. Большое дело. Вы устранили атрибуты компилятора — фрейм стека и ограничение использование регистров — но вы по-прежнему думаете как компилятор. Попробуйте это:

```
>repnz scasw
jz...Match
```

<Это простой пример, но, я надеюсь, убедительный. Напрягайте свой мозг, когда вы оптимизируете.

Доводите его до предела

Максимальная оптимизация языка ассемблера достигается при изменении правил; это то есть когда вы реорганизуете всю программу, чтобы обеспечить использование более совершенного языка ассемблера код в небольшой части кода, которая больше всего влияет на общую производительность. Например, учтите, что данные, которые были найдены в последнем примере, хранятся в массиве структур, с каждой структурой в массиве, содержащей также другую информацию. В этой ситуации, REP SCASW не может быть использован, потому что данные, по которым выполняется поиск, не будут непрерывными.

Однако, если потребность в производительности при поиске в массиве достаточно срочна, нет нет причин, по которым вы не можете реорганизовать данные. Это может означать удаление массива элементы структуры и их хранение на собственных массива так, что рэп SCASW может использоваться.



Организация данных программы, так что производительности критически важных возможность оптимизации разделов - ключевая часть дизайна, и ее легко изменить если на этапе проектирования вы не разберетесь в этом досконально и работайте над объединением ваших потребностей в данных, критических разделов вашей программы и потенциальных возможностей оптимизации языка ассемблера.

Подробнее об этом чуть позже.

Подводя итог, вот на что следует обратить внимание при попытке преобразовать код C в оптимизированный язык ассемблера:

- Перевести всю критически важную для производительности секцию на один язык ассемблера функция.
- По возможности не используйте вызовы или обращения к фреймам стека внутри критического кода и избегайте любых ненужных обращений к памяти.
- Меняйте сегменты как можно реже.
- Оптимизируйте с точки зрения того, что ассемблер делает хорошо, а не с точки зрения точной настройки скомпилированного C-код.
- При необходимости измените правила в интересах ассемблера; например, реорганизуите структура данных, обеспечивающая эффективную обработку на языке ассемблера.

Тем не менее, позвольте мне показать некоторые из этих принципов в действии.

Примерное исследование C-to-Assembly

Листинг 8.1 - пример приложения на C, которое я собираюсь использовать для изучения оптимизации в действии. Листинг 8.1 на самом деле неполный — в нем плохо рассматривается “отсутствие совпадений”, и это предполагается, что сумма всех совпадений поместится в int-но это прекрасно подойдет в качестве пример оптимизации.

ЛИСТИНГ 8.1 L8-1.C

```
>/* Program to search an array spanning a linked list of variable-
sized blocks, for all entries with a specified ID number,
```

```
        <=>and return the average of the values of all such entries. Each of
        the variable-sized blocks may contain any number of data entries,
        stored as an array of structures within the block. */

#include <stdio.h>

#ifdef _TURBOC_

#include <alloc.h>

#else

#include <malloc.h>

#endif

void main(void);

void exit(int);

unsigned int FindIDAverage(unsigned int, struct BlockHeader *);

/* Structure that starts each variable-sized block */

struct BlockHeader {

    struct BlockHeader *NextBlock; /* Pointer to next block, or NULL
                                     if this is the last block in the
                                     linked list */

    unsigned int BlockCount; /* The number of DataElement entries
                               in this variable-sized block */

}; _ /* Structure that contains one
element of the array we'll search */

struct DataElement {

    unsigned int ID; /* ID # for array entry */

    unsigned int Value; /* Value of array entry */

}; _

void main(void) {

    int i,j;

    unsigned int IDToFind;

    struct BlockHeader *BaseArrayBlockPointer,*WorkingBlockPointer;

    struct DataElement *WorkingDataPointer;

    struct BlockHeader **LastBlockPointer;

    printf("ID # for which to find average: ");

    scanf("%d",&IDToFind);

    /* Build an array across 5 blocks, for testing */

    /* Anchor the linked list to BaseArrayBlockPointer */

    LastBlockPointer = &BaseArrayBlockPointer;

    /* Create 5 blocks of varying sizes */

    for (i = 1; i < 6; i++) {

        /* Try to get memory for the next block */

        if ((WorkingBlockPointer =
            (struct BlockHeader *) malloc(sizeof(struct BlockHeader) +
            sizeof(struct DataElement) * i * 10)) == NULL) {

            exit(1);

            /* Set the # of data elements in this block */

            WorkingBlockPointer->BlockCount = i * 10;

            /* Link the new block into the chain */

            *LastBlockPointer = WorkingBlockPointer;

            /* Point to the first data field */

            WorkingDataPointer =
                (struct DataElement *) ((char *)WorkingBlockPointer +
                sizeof(struct BlockHeader));

            /* Fill the data fields with ID numbers and values */

            for j=0 j < (i * 10); j++, WorkingDataPointer++) {

                WorkingDataPointer->ID = j;

                WorkingDataPointer->Value = i * 1000 + j;

                /* Remember where to set link from this block to the next */

                LastBlockPointer = &WorkingBlockPointer->NextBlock;

                /* Установите указатель "next block" для последнего блока равным NULL, чтобы указать
                что блоков больше нет */

                WorkingBlockPointer->NextBlock = NULL;

                printf("Среднее значение всех элементов с идентификатором %d: %u\n",
                    IDToFind, FindIDAverage(IDToFind, BaseArrayBlockPointer));

                выход(0);

            /* Выполняет поиск по массиву записей DataElement, охватывающему
            связанный список блоков переменного размера, начиная с блока
            указывает BlockPointer для всех записей с совпадающими идентификаторами
            Выполняется поиск по идентификатору и возвращает среднее значение этих записей. Если
            совпадений не найдено, возвращается ноль */

            unsigned int FindIDAverage(unsigned int SearchedForID,

                <=>struct BlockHeader *BlockPointer)
```

```
<struct DataElement *указатель
данных; unsigned int IDMatchSum;
unsigned int IDMatchCount;
unsigned int WorkingBlockCount;
```

```
IDMatchCount = IDMatchSum = 0;
```

```
/* Выполнять поиск по всем связанным блокам, пока не будет найден последний блок
(помеченный нулевым указателем на
следующий блок)
выполнен поиск */
```

```
>do {
    /* Point to the first DataElement entry within this block */
    DataPointer =
        (struct DataElement *) ((char *)BlockPointer +
        sizeof(struct BlockHeader));
    /* Search all the DataElement entries within this block
    and accumulate data from all that match the desired ID */
    for (WorkingBlockCount=0;
        WorkingBlockCount<BlockPointer->BlockCount;
        WorkingBlockCount++, DataPointer++) {
        /* If the ID matches, add in the value and increment the
        match counter */
        if (DataPointer->ID == SearchedForID) {
            IDMatchCount++;
            IDMatchSum += DataPointer->Value; /* Point to the
next block, and continue as long as that pointer
isn't NULL */
        } while ((BlockPointer = BlockPointer->NextBlock) != NULL);
        /* Calculate the average of all matches */
        if (IDMatchCount == 0)
            return(0); /* Avoid division by 0 */
        else
            return(IDMatchSum / IDMatchCount);
    }
```

<Основная часть листинга 8.1 создает связанный список блоков памяти различного размера и сохраняет массив структур в этих блоках, как показано на рисунке 8.2. функция FindIDAverage из листинга 8.1 выполняет поиск в этом массиве всех совпадений с а указанный идентификационный номер и возвращает среднее значение всех таких совпадений. FindIDAverage содержит два вложенных цикла, внешний из которых повторяется один раз для каждого связанного блока и Внутренний, повторяющийся один раз для каждого элемента массива в каждом блоке. Внутренний цикл - critical one— компактен, содержит всего четыре оператора и должен довольно хорошо поддаваться оптимизации компилятора.

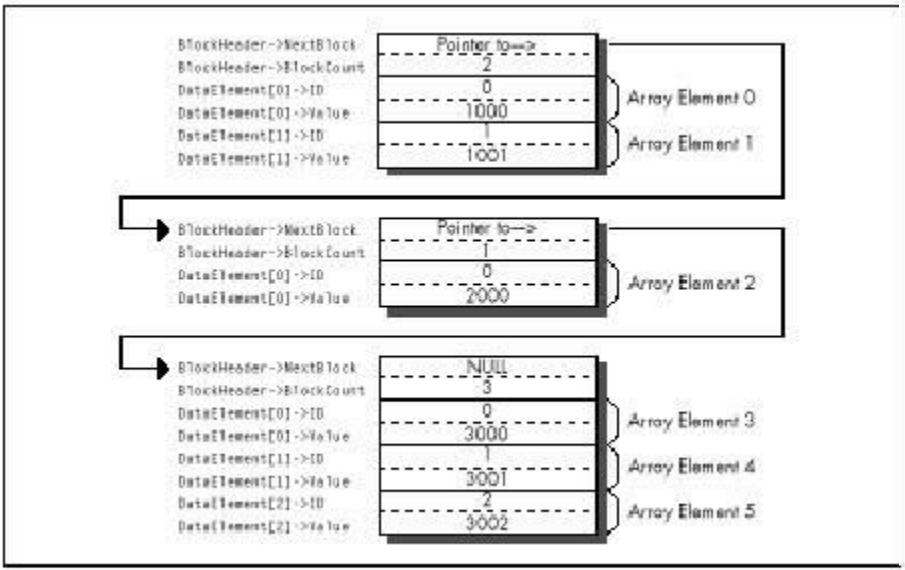


Рисунок 8.2 Формат хранения связанных массивов (версия 1).

Так получилось, что Microsoft C / C ++ действительно хорошо оптимизирует внутренний цикл FindIDAverage. В листинге 8.2 показан код, который Microsoft C / C ++ генерирует для внутреннего цикла, состоящий из всего семи инструкций на языке ассемблера внутри цикла. Компилятор умен достаточно для преобразования переменной индекса цикла, которая подсчитывает, но используется только для подсчета циклов, в переменную обратного отсчета, чтобы можно было использовать инструкцию цикла.

ЛИСТИНГ 8.2 L8-2.COD

```
>; Code generated by Microsoft C for inner loop of FindIDAverage.
;|*** for (WorkingBlockCount=0;
; / * * *WorkingBlockCount<BlockPointer->BlockCount;
;|***WorkingBlockCount++, DataPointer++) {
    mov WORD PTR [bp-6],0;WorkingBlockCount
    mov bx,WORD PTR [bp+6];BlockPointer
    cmp WORD PTR [bx+2],0
    je _$FB264
    mov cx,WORD PTR [bx+2]
    add WORD PTR [bp-6],cx;WorkingBlockCount
    mov di,WORD PTR [bp-2];IDMatchSum
    mov dx,WORD PTR [bp-4];IDMatchCount

$L20004:
;|*** if (DataPointer->ID == SearchedForID) {
    mov ax,WORD PTR [si]
    cmp WORD PTR [bp+4],ax;SearchedForID
    jne $1265
        IDMatchCount++;
        inc dx_
    }
```

```
<>;/**IDMatchSum += DataPointer->Value;
    add di,WORD PTR [si+2]
;/**
;/**f
$I265:
    add si,4
    loop $L20004
    mov WORD PTR [bp-2],di;IDMatchSum
    mov WORD PTR [bp-4],dx;IDMatchCount
$FB264:
```

<Сложно добиться большей производительности от этого кода, изменив его, поскольку примером может служить листинг 8.3, доработанная ассемблерная версия FindIDAverage, которая была создана путем просмотра выходных данных ассемблера MS C / C ++ и их корректировки. Листинг 8.3 устраняет весь доступ к фрейму стека во внутреннем цикле, но это, пожалуй, все ограничения что еще нужно сделать. Результат, как показано в таблице 8.1, заключается в том, что листинг 8.3 выполняется на скромных 11 процентов быстрее, чем листинг 8.1 на 386. Результаты могут значительно отличаться в зависимости от от характера просматриваемого набора данных (средний размер блока и частота совпадений). Но, в таком случае, понимание типичных и наихудших условий является частью оптимизация, не так ли?

ЛИСТИНГ 8.3 L8-3.ASM

```
>; Typically optimized assembly language
version of FindIDAverage. SearchedForID
equ 4;Passed parameter offsets in the
BlockPointer equ 6; stack frame (skip over
pushed BP ; and the return address)
NextBlockequ 0;Field offsets in struct BlockHeader
BlockCountequ 2 BLOCK_HEADER_SIZE equ 4;Number of
bytes in struct BlockHeader ID _equ 0;struct DataElement
field offsets Valueequ 2 DATA_ELEMENT_SIZE
equ 4;Number of bytes in struct DataElement
.model small
.code
public _FindIDAverage
```

<Таблица 8.1 Время выполнения FindIDAverage.

	На 20 МГц 386	На 10 МГц 286
Листинг 8.1 (MSC с максимальной оптимизацией)	294	768
	микросекундымикросекунды	
Листинг 8.3 (сборка)	265	644
Листинг 8.4 (Оптимизированная сборка)	212	486
Листинг 8.6 (Оптимизированная сборка с реорганизованными данными)	100	207

```
>_FindIDAverage procnear
    pushbp;Save caller's stack frame
    movbp, sp;Point to our stack frame
    pushdi _;Сохранение регистровых переменных C
    толкатьsi _
    SUBdx, dx;IDMatchSum = 0
    movbx, dx;IDMatchCount = 0
    movsi, [bp + BlockPointer] ; Указатель на первый блок
    movax, [bp+SearchedForID] ; Идентификатор, который мы ищем
; Поиск по всем связанным блокам до последнего блока
; (отмечен нулевым указателем на следующий блок) выполнен поиск.
BlockLoop:
; Указывает на первую запись DataElement в этом блоке.
    leadi, [si+BLOCK_HEADER_SIZE]
; Выполните поиск по всем элементам данных внутри этого блока
; и накапливать данные от всех, которые соответствуют желаемому идентификатору.
    movcx, [si+blockCount]
    jcxz DoNextBlock;В этом блоке нет данных
Внутриблочный цикл:
    стр[di+ID],ax; Совпадают ли наши идентификационные данные?
    jnzНет совпадения; Нет совпадения
    incbx _; У нас есть совпадение; IDMatchCount ++;
    Добавитьdx,[di+Значение] ;IDMatchSum += Указатель данных-> Значение;
NoMatch:
    Добавитьdi, DATA_ELEMENT_SIZE ; указывает на следующий элемент
    цикл IntraBlockLoop
; Укажите на следующий блок и продолжите, если
этот указатель не равен нулю. DoNextBlock:
    movsi,[si + NextBlock] ; Получить указатель на следующий блок
```

	<и si,si	;Это указатель null?
	jnz BlockLoop	Нет, продолжения
; Вычислить	в среднее	Матчи.
	sub ax, ax	
	и bx, bx	Предположим, что мы не нашли совпадений
	jz _Выполнено	Мы не нашли никаких совпадений, возвращаем 0
	xchgax,dx	Подготовиться к разделению
	div bx _	Вернуть IDMatchSum / IDMatchCount
Выполнено:	хлопси _	Восстановить регистровые переменные C
	хлопди _	
	pop	
	ret bp.	Восстановить фрейм стека вызывающего объекта
_FindIDAverage	end ENDP	

В листинге 8.4 представлены некоторые сложные методы оптимизации. Цикл разворачивается восемь раз, что исключает значительное разветвление, и вместо используется scasw CMP [DI], AX. (Обратите внимание, однако, что scasw на самом деле медленнее, чем CMP [DI], AX на 386 и 486, и иногда быстрее на 286 и 8088 только потому, что он короче и следовательно, может выполняться предварительная выборка быстрее.) Эта расширенная настройка дает 39-процентный результат улучшение по сравнению с исходным кодом C — существенная, но не огромная отдача от вложенные усилия по оптимизации.

ЛИСТИНГ 8.4 L8-4.ASM

```
>; Heavily optimized assembly language version of FindIDAverage.
<>; Features an unrolled loop and more efficient pointer use.
SearchedForID equ4;Passed parameter offsets in the
BlockPointerequ6; stack frame (skip over pushed BP
; and the return address)
NextBlockequ0;Field offsets in struct BlockHeader
BlockCountequ2
BLOCK_HEADER_SIZE equ 4;Number of bytes in struct BlockHeader
ID _equ0;struct DataElement field offsets
Valueequ2
DATA_ELEMENT_SIZE equ 4;Number of bytes in struct DataElement
.model small
.code
public _FindIDAverage
_FindIDAverage procnear
pushbp.;Save caller's stack frame
movbp,sp;Укажите на наш фрейм стека
толкатьди _;Сохранить регистровые переменные C
толкатьси _
движениеди, ds;Подготовиться к SCASW
перемещениеes, ди
cld
SUBdx, dx;IDMatchSum = 0
movbx, dx; IDMatchCount = 0
movsi, [bp + BlockPointer] ; Указатель на первый блок
movax, [bp+SearchedForID] ; Идентификатор, который мы ищем
; Поиск по всем связанным блокам до последнего блока
; (отмечен нулевым указателем на следующий блок) выполнен поиск.
BlockLoop:
; Указывает на первую запись DataElement в этом блоке.
lea di,[si+BLOCK_HEADER_SIZE]
; Выполнить поиск по всем элементам данных в этом блоке
; и накапливать данные от всех, которые соответствуют желаемому идентификатору.
movcx,[si+blockCount] ; Количество элементов в этом блоке
jcxz DoNextBlock; Пропустить этот блок, если он пустой
перемещениеbp, cx; *** фрейм стека больше недоступен ***
Добавитьпример, 7
схрсх, 1;Количество повторений развернутого
shrпример, 1; цикл = (количество блоков + 7) / 8
shrex, 1
ибр,7; Создайте точку входа для
shlbp, 1; первый, возможно, частичный проход через
jmpcs: [LoopEntryTable+ bp] ; развернутый цикл и
; вектор к этой точке входа
выровнять2
Слово-метка для записи циклов
dw _LoopEntry8, LoopEntry1, LoopEntry2, LoopEntry3
dw _LoopEntry4, LoopEntry5, LoopEntry6, LoopEntry7
Макрос M_IBL P1
локальное соответствие
LoopEntry&P1&:
scasw; Есть ли у нас совпадение идентификаторов?
jnzНет совпаденияНет совпадения
;У нас есть совпадение
incbx _; IDMatchCount ++;
Добавитьdx, [di];IDMatchSum += Указатель данных-> Значение;
NoMatch:
Добавитьdi, DATA_ELEMENT_SIZE-2 ; указывает на следующий элемент
; (SCASW продвинулся
уже на 2 байта)
endm
```

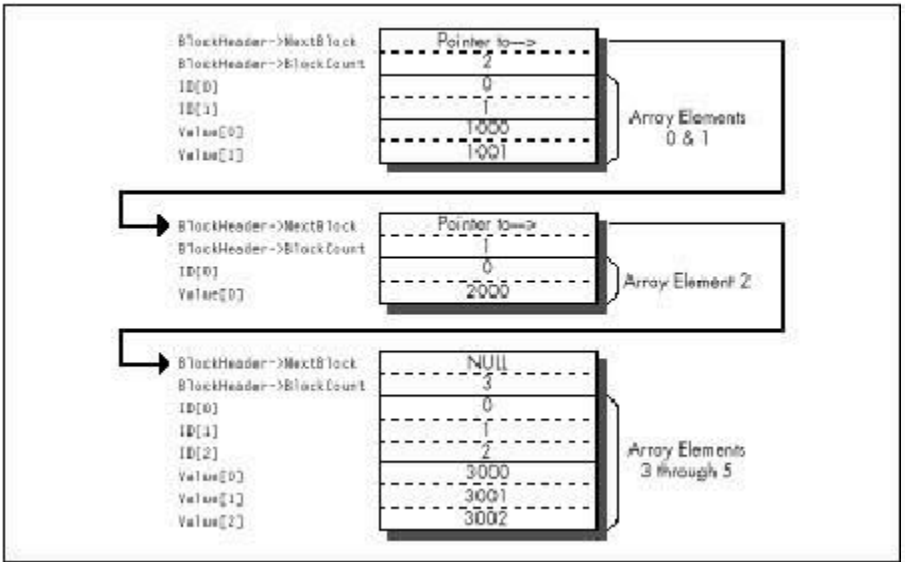
```
<>align2
IntraBlockLoop:
M_IBL8
M_IBL7
M_IBL6
M_IBL5
M_IBL4
M_IBL3
M_IBL2
M_IBL1
loop IntraBlockLoop
; Point to the next block and continue if that pointer isn't NULL.
DoNextBlock:
movsi,[si+NextBlock] ;Get pointer to the next block
andsi,si;Is it a NULL pointer?
jnzBlockLoop;No, continue
; Calculate the average of all matches.
subax,ax;Assume we found no matches
andbx, bx
jz _Done;We didn't find any matches, return 0
xchgax, dx;Prepare for division
divbx _;Return IDMatchSum / IDMatchCount
Done:popsi _;Restore C register variables
popdi _
popbp.;Restore caller's stack frame
ret
_FindIDAverage ENDP
end
```

<Списки 8.5 и 8.6 вместе делают заключительный шаг и изменяют правила в пользу assembly язык. В листинге 8.5 создается тот же список связанных блоков, что и в листинге 8.1. Однако вместо хранения массива структур внутри каждого блока, он хранит два массива в каждом блок, один состоит из идентификационных номеров, а другой - из соответствующих значений, как показано на рисунке 8.3. Никакая информация не теряется; данные просто переставляются.

ЛИСТИНГ 8.5 L8-5.C

```
/* Программа для поиска в массиве, охватывающем связанный список блоков переменного размера
, для всех записей с указанным идентификационным номером,
и возвращает среднее значение всех таких записей. Каждый из
блоки переменного размера могут содержать любое количество записей данных,
хранится в виде двух отдельных массивов, одного для
идентификационных номеров и
одного для значений. */
```

```
>#include <stdio.h>
#ifdef _TURBOC_
#include <alloc.h>
#else
#include <malloc.h> #endif
void main(void); void
exit(int); extern unsigned
int FindIDAverage2(unsigned
int,
struct
BlockHeader *);
```



<Рисунок 8.3 Формат хранения связанных массивов (версия 2).

```
/* Structure that starts each variable-sized
block */ struct BlockHeader {
    struct BlockHeader *NextBlock; /* Pointer to next block, or NULL
                                     if this is the last block in the
                                     linked list */
    unsigned int BlockCount; /* The number of DataElement entries
                               in this variable-sized block */
};

Hi _
void main(void) {
    int i, j;
    unsigned
    int IDToFind;
```

```
<>struct BlockHeader *BaseArrayBlockPointer,*WorkingBlockPointer;
int *WorkingDataPointer;
struct BlockHeader **LastBlockPointer;
printf("ID # for which to find average: ");
scanf("%d",&IDToFind); /* Build
an array across 5 blocks, for testing */ /* Anchor
the linked list to BaseArrayBlockPointer */
LastBlockPointer = &BaseArrayBlockPointer;
/* Create 5 blocks of varying sizes */

<>for (i = 1; i < 6; i++) {
    /* Try to get memory for the next block */
    if ((WorkingBlockPointer =
        (struct BlockHeader *) malloc(sizeof(struct BlockHeader) +
        sizeof(int) * 2 * i * 10)) == NULL) {
        exit(1);
    /* Set the number of data elements in this block */
    WorkingBlockPointer->BlockCount = i * 10;
    /* Link the new block into the chain */
    *LastBlockPointer = WorkingBlockPointer;
    /* Point to the first data field */
    WorkingDataPointer = (int *) ((char *)WorkingBlockPointer +
        sizeof(struct BlockHeader));
    /* Fill the data fields with ID numbers and values */
    for j=0 j < (i * 10); j++, WorkingDataPointer++) {
        *WorkingDataPointer = j;
        *(WorkingDataPointer + i * 10) = i * 1000 + j;
    }
    /* Remember where to set link from this block to the next */
    LastBlockPointer = &WorkingBlockPointer->NextBlock;
    /* Установите указатель "next block" для последнего блока равным NULL, чтобы указать
    что блоков больше нет */
    WorkingBlockPointer->NextBlock = NULL;
    printf("Среднее значение всех элементов с идентификатором %d: %u\n",
        IDToFind, FindIDAverage2(IDToFind, BaseArrayBlockPointer));
    выход(0);
}
```

ЛИСТИНГ 8.6 L8-6.ASM Выполнен

```
поиск idequ4; Переданы смещения параметров в
BlockPointerequ6; фрейм стека (пропустить нажатый BP
<>; Alternative optimized assembly language version of
FindIDAverage ; requires data organized as two arrays with
each BlockHeader as an array of two-value element
structures. This allows the ; use of REP SCASW for ID searching.
BlockCountequ2
block_header_sizeequ4;Количество байтов в struct BlockHeader
    .модель маленькая
    .код
    общедоступный _FindIDAverage2
процедура _FindIDAverage2рядом
    толкатьдавление;Сохраняем фрейм стека вызывающего абонента.,,,
    movbp, sp,, Указываем на наш фрейм стека
    толкатьdi _;Сохранить регистровые переменные C
    толкатьsi _
    движениеді, ds; Подготовиться к SCASW
    движениеes, di
    cld
    перемещениеsi, [bp + BlockPointer] ; Указатель на первый блок
    movax, [bp + SearchedForID] ; Идентификатор, который мы ищем
    SUBdx, dx;IDMatchSum = 0
    перемещениеbp, dx;IDMatchCount = 0
    ;*** фрейм стека больше недоступен ***

; Выполнить поиск по всем связанным блокам до последнего блока
; (отмечен нулевым указателем на следующий блок) выполнен поиск.
BlockLoop:
; Поиск по всем элементам DataElement в этом блоке
; и накапливать данные от всех, которые соответствуют желаемому идентификатору.
    movcx, [si +количество блоков]
    jcxz DoNextBlock; Пропустите этот блок, если нет данных
    ; для поиска по
    movbx, cx;Мы будем использовать BX, чтобы указать на
    shlbx, 1; соответствующая запись значения в
; случай совпадения идентификатора (BX - это
; длина массива идентификаторов в байтах)
; Указывает на первую запись DataElement в этом блоке.
    leadi, [si+BLOCK_HEADER_SIZE]
Внутриблочный цикл:
    repnzscasw; Поиск идентификатора
    jnzDoNextBlock;Нет совпадения, блокировка завершена
    incbp.; У нас есть совпадение; IDMatchCount ++;
    Добавитьdx,[di+bx-2];IDMatchSum += Указатель данных->Значение;
```

```
<>; (SCASW has advanced DI 2 bytes)
    and ex, ex; Is there more data to search through?
    jnz IntraBlockLoop; yes
; Point to the next block and continue if that pointer isn't NULL.
DoNextBlock:
    mov si, [si+NextBlock] ; Get pointer to the next block
    and si, si; Is it a NULL pointer?
    jnz BlockLoop; No, continue
; Calculate the average of all matches.
    sub ax, ax; Assume we found no matches
    and bp, bp
    jz _Done; We didn't find any matches, return 0
    xchg ax, dx; Prepare for division
    div bp; Return IDMatchSum / IDMatchCount
Done: pop si _; Restore C register variables
    pop di _
    pop bp; Restore caller's stack frame
    ret
_FindIDAverage2 ENDP
end
```

<Весь смысл этой перестановки в том, чтобы позволить нам использовать REP SCASW для поиска по каждому блоку, и это именно то, что делает FindIDAverage2 в листинге 8.6. Результат: В листинге 8.6 среднее значение вычисляется примерно в три раза быстрее, чем исходное C реализация и более чем в два раза быстрее, чем в листинге 8.4, при этом сильно оптимизирован, как и последний код.

Я надеюсь, вы поняли картину. Пошаговая оптимизация, которой так много многие из нас любят заниматься в виде головоломок, - это весело, но компиляторы могут делать это почти так же хорошо, как и вы может, и в будущем наверняка сделает это лучше. Чего не может сделать компилятор, так это связать воедино потребности спецификации программы на высоком уровне и процессора на низком уровне, в результате получается критически важный код, который выполняется примерно с той скоростью, с какой позволяет аппаратное обеспечение. Единственный программное обеспечение, которое может это делать, расположено к северу от вашей грудины и немного позади носа. Смахните с него пыль и запустите в работу — и ваш код больше никогда ни с чем не спутают от Hamilton, Joe, Frank, eynolds или Bo Donaldson and the Heywoods.

Глава 9 – Советы, которые дали мне мои читатели

Всевозможные способы оптимизации из этой области

Еще в старших классах я посещал занятия по математике у мистера Бурджиса, чьи самые известные характерными чертами были постоянная ходьба и поистине огромные ступни. Мой друг Барри, который сидел в заднем ряду, прямо за мной, утверждал, что именно из-за его больших ступней Мистер Бурджис был таким беспокойным. Барри предположил, что эти ноги были такими тяжелыми, что если мистер Бурж слишком долго оставался на одном месте, пол мог провалиться под напряжением, погружая несчастного учителя глубоко под мантию Земли и, возможно, вплоть до Китая. На этот счет было нарисовано много забавных карикатур.

К сожалению, Барри был слишком занят рисованием карикатур или, наоборот, сном, чтобы на самом деле выучить какую-либо математику. В долгосрочной перспективе это не стало препятствием для Барри, который впоследствии стал вице-президентом по продажам компании по упаковке ветчины, где предположительно, к нему редко обращались с просьбой вывести квадратное уравнение. Отсутствие у Барри однако в то время эрудиция вызывала некоторые проблемы. В одном памятном случае, Барри был в полусне, с открытыми, но расфокусированными глазами, подперев подбородок рукой рука в классической позе “если я засну, моя голова отвалится от руки, и я проснусь”, когда мистер Бурджис раскрыл убийственную проблему:

“Барри, реши это для ”Икс", пожалуйста". На доске было выведено уравнение:


$$x - 1 = 0$$



“Минус 1”, - быстро сказал Барри.

Мистер Бурджис скорбно покачал головой. “Попробуй еще”. Барри напряженно думал. Он знал фундаментальное правило, согласно которому ответом на большинство математических вопросов является либо 0, 1, бесконечность, -1, или минус бесконечность (однако не применяйте это правило к балансу вашей чековой книжки); к сожалению, это давало ему только 25-процентный шанс угадать правильно.

“Один”, - украдкой прошептала я.

“Ноль”, - объявил Барри. Мистер Бурджис еще печальнее покачал головой.

“Один”, - прошептала я громче. Барри выглядел еще более задумчивым — плохой знак, — поэтому я прошептала “один” еще раз, еще громче. Барри выглядел таким задумчивым, что его глаза чуть не закатились и я поняла, что он просто делал все возможное, чтобы убедить Мистера Бурджису, что Барри решил эту проблему сам.

Когда Барри приблизился к кульминации своего волнующего выступления и открыл рот, чтобы заговорить, Мистер Бурджис посмотрел на него с большим беспокойством. “Барри, ты меня хорошо слышишь?”

“Да, сэр”, - ответил Барри. “Почему?”

“Ну, я мог слышать ответ всю дорогу сюда. Вы, конечно, могли слышать это всего в одном ряду отсюда? ”

Класс пришел в неистовство. С таким же успехом они могли отправить нас домой пораньше за все, что мы сделали за остаток дня.

Мне нравится думать, что я знаю о программировании производительности больше, чем Барри знал о математике. Тем не менее, я всегда приветствую хорошие идеи и комментарии, которые присылают многие читатели за эти годы у меня их было множество. Поэтому в этой главе, я думаю, я отплачу вам тем же посвятив главу отзывам читателей.

Еще один взгляд на LEA

Несколько человек отметили, что, хотя LEA отлично подходит для выполнения определенных дополнений (см. Главу 6), это не идеальная замена ADD . В чем разница? LEA, инструкция адресации по обмену, не влияет на флаги, в то время как арифметическая команда ADD инструкция, безусловно, влияет. Это не проблема при выполнении дополнений, которые используйте только те величины, которые умещаются в одном машинном слове (32 бита в защищенном режиме 386, 16 в противном случае), но это делает LEA бесполезным для операций с несколькими словами, в которых используется перенос отметьте, чтобы связать воедино частичные результаты. Например, в этих инструкциях

```
ДОБАВЬТЕ EAX, EBX
ADC EDX, ECX
```

заменить невозможно

```
>LEA EAX, [EAX+EBX]
ADC EDX, ECX
```

<поскольку LEA не влияет на флаг переноса.

Характеристика отсутствия переноса для LEA становится явным преимуществом при выполнении однако арифметика указателей. Например, следующий код использует LEA для улучшения указатели при добавлении одной 128-битной переменной памяти к другой такой переменной:

```
>MOV ECX,4 ;# of 32-bit words to add
CLC
;no carry into the initial ADC
ADDLOOP:
MOV EAX, [ESI] ;get the next element of one array
ADC [EDI],EAX ; add it to the other array, with carry
LEA ESI,[ESI+4] ;advance one array's pointer
LEA EDI, [EDI+4] ;advance the other array's pointer
LOOP ADDLOOP
```

<(Да, я мог бы использовать LODSD вместо MOV / LEA; я просто иллюстрирую здесь точку зрения. Кроме того,

LOD всего на 1 цикл быстрее, чем MOV / LEA на 386, и на самом деле более чем в два раза медленнее на 486.) Если бы мы использовали ADD, а не LEA для продвижения указателей, перенос из переход от одного АЦП к следующему должен быть сохранен либо с помощью PUSHF / POPF, либо с помощью LAHF / SAHF. (В качестве альтернативы, мы могли бы использовать несколько INC, поскольку INC не влияет на флаг переноса.)

Короче говоря, LEA действительно отличается от ADD . Иногда это лучше. Иногда нет; это характер различных замен инструкций и оптимизаций, которые будут происходить с вами со временем. “Лучших” инструкций на x86 не существует; все зависит от то, что ты пытаешься сделать.

Но наверняка есть много интересных вариантов, не так ли?

Портфолио Кеннеди

Читатель Джон Кеннеди регулярно делится интригующими приемами программирования на ассемблере, многие из которых я никогда больше нигде не видел упомянутыми. Джону нравится оптимизировать размер, в то время как я больше склоняюсь к скорости, но многие из его оптимизаций хороши для обеих целей. Вот несколько моих любимых:

Код Джона для установки AX в абсолютное значение -:

```
CWD
XOR AX, DX
SUB AX, DX
```

Это ничего не делает, когда бит 15 AX равен 0 (то есть, если AX положительное значение). Когда AX отрицательное значение, код “не учитывает” это и добавляет 1, именно так вы выполняете дополнение двойки отрицание. Для случая, когда AX не является отрицательным, этот трюк обычно превосходит начинку из стандартный код абсолютного значения:

```
и          AX, AX      ; отрицательный?
JNS        Положительный; нет
ОТРИЦАНИЕ  AX          ; да, отрицание это
Положительно:
```

Однако код Джона работает медленнее на 486; как вы, без сомнения, начинаете понимать (и как Я объясню в главах 12 и 13), 486 -й - это мир оптимизации сам по себе.

Вот как Джон копирует блок байтов из DS: SI в ES: DI, перемещая столько данных, сколько возможно по слову за раз:

```
>SHR CX,1;word count REP MOVSW ;copy
as many words as possible ADC
CX, CX ;CX=1 if copy length was odd,
;0 else
REP MOVSB;copy any odd byte
```

<(АЦП CX, CX можно заменить на RCL CX, 1; что быстрее, зависит от процессора тип.) Может быть трудно поверить, что вышеописанное выполняется быстрее, чем это:

```
SHR      CX, 1      ; количество слов
REP      MOVSW      ; скопируйте как можно больше слов
; возможно
JNC      CopyDone; выполняется при четной длине копии
MOVSB    ; скопируйте нечетный байт
CopyDone:
```

Однако, как правило, это так. Конечно, если длина нечетная, подход Джона влечет за собой штраф примерно равный времени начала повторения для MOVSB. Однако, если длина четная, Подход Джона не ветвится, сохраняя циклы и не опустошая очередь предварительной выборки. Если длины копий равномерно распределены между четными и нечетными, подход Джона быстрее в большинство систем x86. (Но не на 486-й.)

Джон также указывает, что на 386-й несколько LEA могут быть объединены для выполнения умножения, которые не могут быть обработаны одним LEA, так же как множественные сдвиги и добавления можно использовать для умножения, только быстрее. LEA можно использовать для умножения в одной инструкции на 386, но только на значения 2, 3, 4, 5, 8, и 9; несколько последовательных LEA

вместе они могут обрабатывать гораздо более широкий диапазон значений. Например, видеопрограммисты несомненно, знакомы со следующим кодом для умножения AX на 80 (ширина в байтах растровое изображение в большинстве режимов отображения на ПК):

```
SHL      AX, 1 ; *2
SHL      AX, 1 ; *4
ШЛ      AX, 1 ; *8
SHL      AX, 1 ; *16
ПЕРЕМЕЩЕНИЕ  BX, AX
ШЛ      AX, 1 ; *32
SHL      AX, 1 ; *64
Добавить    AX, BX; *80
```

Используя LEA на 386, вышеуказанное можно было бы сократить до

```
LEA      EAX, [EAX*2]
LEA      EAX, [EAX*8] ; *16
LEA      EAX, [EAX+EAX*4] ; *80
```

что все еще не так быстро, как использование таблицы подстановки типа

```
>MOV EAX, MultiplesOf80Table [EAX*4]
```

<, но близко и занимает намного меньше места.

Конечно, на 386 версии shift и add также могут быть сокращены до этого значительно более эффективный код:

```
SHL      AX, 4 ; *16
ПЕРЕМЕЩЕНИЕ  BX, AX
SHL      AX, 2 ; *64
Добавить    AX, BX
```

Ускорение умножения

Это подводит нас к умножению, одной из самых медленных операций x86, которая позволяет для значительной оптимизации. Один из способов ускорить умножение - использовать shift и add , LEA или таблица подстановки для жесткого кодирования операции умножения для фиксированного множителя, как показано выше. Другой способ заключается в том, чтобы воспользоваться функцией раннего выпуска модели 386 (и 486, но в интересах краткости с этого момента я буду говорить просто “386”), расставив ваши операнды так, чтобы множитель (всегда крайний правый операнд, следующий за MUL или IMUL) был не больше другого операнда.



Почему? Потому что 386 обрабатывает один бит множителя за цикл и немедленно завершает умножение, когда все значащие биты множителя были обработаны, поэтому для умножения а требуется меньше циклов большой множитель, умноженный на малый множитель, чем маленький множитель, умноженный на большой множитель, примерно в 1 цикл для каждого исключенного значащего бита множителя.

(Для этого трюка предусмотрено минимальное время выполнения; ниже 3 значащих битов множителя дополнительные циклы не сохраняются.) Например, умножение 32 767 на 1 составляет 12 циклов быстрее, чем умножение 1 на 32 727.

Правильный выбор операнда в качестве множителя может творить чудеса. Согласно опубликованным спецификациям, для умножения 386-го требуется 38 циклов на множитель с 32 значащими битами, но только 9 циклов умножения на множитель 2, повышение производительности более чем в четыре раза! (Мои тесты регулярно показывают, что умножение занимает на 3-4 цикла больше времени, чем спецификации указывают, но преимущество меньших множителей в расчете на цикл на бит остается в силе тем не менее.)

Это подчеркивает еще один интересный момент: MUL и IMUL на 386 настолько быстры, что альтернативные подходы к умножению, хотя в целом они еще быстрее, имеют смысл только в действительно критичный по времени код.



На 386SXs и некэшированных 386s, где размер кода может существенно влиять



производительность благодаря предварительной выборке команд, компактным MUL и IMUL инструкции могут приближаться к "оптимизированным" альтернативам, а в некоторых случаях даже превосходить их .

В целом, MUL и IMUL являются разумными исполнителями на 386-м, которых больше не следует избегать в большинстве случаев — и вы можете помочь этому, упорядочив свой код таким образом, чтобы он был меньше операнд присваивает множитель всякий раз, когда вы знаете, какой операнд меньше.

Это не означает, что ваш код должен тестировать и менять местами операнды, чтобы убедиться, что меньший из них является множителем; это редко окупается. Я больше говорю о случае, когда вы масштабируете массив на значение, которое всегда находится в диапазоне, скажем, от 2 до 10; поскольку значение масштаба всегда будет небольшим, а элементы массива могут иметь любое значение, значение масштаба является логичным выбором для множителя.

Оптимизация оптимизированного поиска

Роб Уильямс пишет о замечательной оптимизации для REPNZ SCASB, оптимизированной на основе процедура поиска, которую я обсуждал в главе 5. В качестве краткого обновления я описал поиск в буферизуйте текстовую строку следующим образом: Сканируйте первый байт текстовой строки с помощью REPNZ SCASB, затем используйте REPZ CMPS для проверки полного соответствия всякий раз, когда REPNZ SCASB находит совпадение с первым символом, как показано на рисунке 9.1. Принцип заключается в том, что большая часть буфера символы не будут соответствовать первому символу какой-либо заданной строки, поэтому REPNZ SCASB, безусловно, является самый быстрый способ поиска на ПК, который можно использовать для исключения большинства потенциальных совпадений; каждое оставшееся потенциальное совпадение затем может быть полностью проверено с помощью REPZ CMPS.

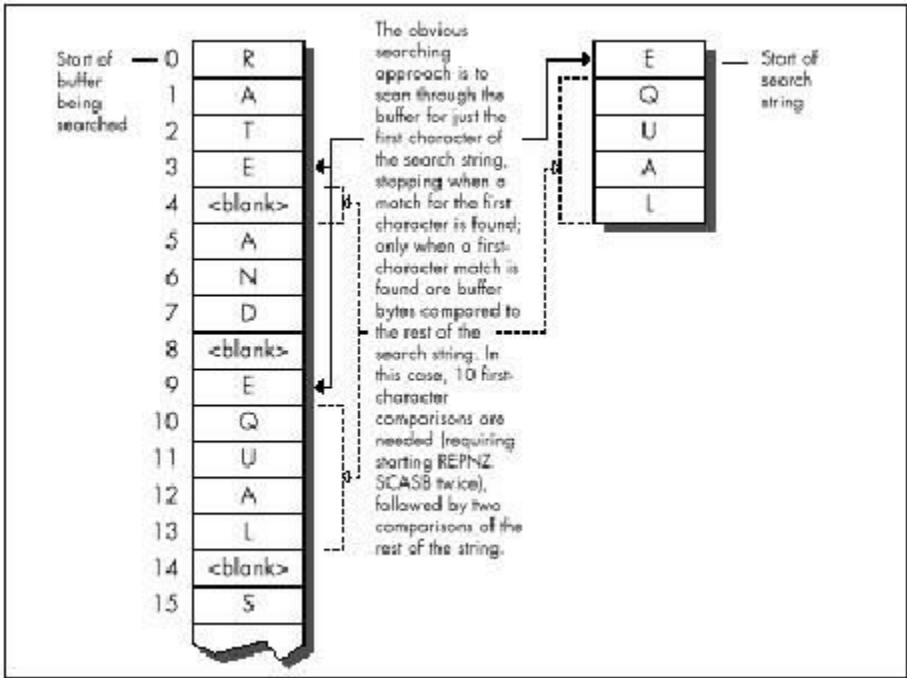


Рисунок 9.1 Простой метод поиска текстовой строки.

Откровение Роба, которое он без объяснения причин приписывает Эдгару Аллену По (поиск никогда больше?), было ли это, безусловно, самой медленной частью всей сделки - обработка REPNZ SCASB совпадения, для которых требуется проверить оставшуюся часть строки с помощью REPZ CMPS и перезапустить REPNZ SCASB, если совпадение не найдено.



Роб указывает, что количество совпадений REPNZ SCASB может быть легко уменьшено простым сканированием символа в искомой строке это реже всего встречается в буфере, в котором выполняется поиск.

Представьте, если хотите, что вы ищете строку “PABHO”. Согласно моему подходу, вы бы используйте REPNZ SCASB для сканирования каждого вхождения “Е”, которое довольно часто встречается в обычный текст. Роб указывает, что было бы разумнее отсканировать “Q”, а затем создать резервную копию один символ и проверьте всю строку, когда будет найдена “Q”, как показано на рисунке 9.2. “Q”, вероятно, встречается гораздо реже, что приводит к гораздо меньшему количеству проверок всей строки и гораздо более быстрой обработке.

двух подходов. Насколько сильно отличается откровение Роба? Много. Даже когда весь вызов функции C для Findstring выполняется по времени—вызовы strlen, нажатие параметра, вызов, настройка и все остальное — версия Findstring из листинга 9.2, которая управляется Листинг 9.3 для поиска редко встречающегося “Q” выполняется примерно на 40 процентов быстрее на 20 MHz кэшировал 386 для тестового поиска из листинга 9.3, чем версия Findstring в Листинг 9.1, который всегда сканирует первый символ, в данном случае “E.” Однако, когда только циклы поиска (код, который фактически выполняет поиск) в двух версиях При сравнении строк поиска, листинг 9.2 выполняется более чем в два раза быстрее, чем листинг 9.1 — значительное улучшение по сравнению с кодом, который уже использует REP NZ SCASB и REPZ CMPS.

Что мне так нравится в подходе Роба, так это то, что он демонстрирует, что оптимизация включает в себя гораздо больше, чем выбор команд и подсчет циклов. Списки 9.1 и 9.2 используйте практически те же инструкции и даже используйте тот же подход к сканированию C помощью REP NZ SCASB и с помощью REPZ CMPS для проверки совпадений при сканировании.



Разница между списками 9.1 и 9.2 (которая дает вам больше, чем удвоение производительности) полностью обусловлена пониманием природы обрабатываемые данные и изменение кода для отражения этих знаний.

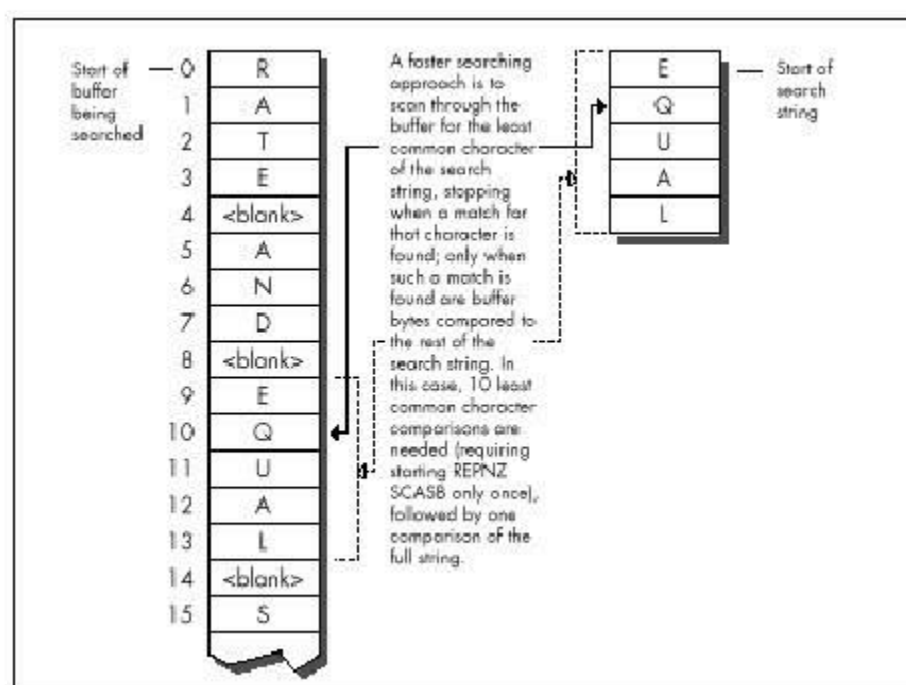


Рисунок 9.2 Более быстрый метод поиска текстовой строки.

ЛИСТИНГ 9.1 L9-1.ASM

```
>; Searches a text buffer for a text string. Uses REP NZ SCASB to sca"n
; the buffer for locations that match the first character of the
; searched-for string, then uses REPZ CMPS to check fully only those
; locations that REP NZ SCASB has identified as potential matches.
;
; Adapted from Zen of Assembly Language, by Michael Abrash
;
; C small model-callable as:
; unsigned char * FindString(unsigned char * Buffer,
; unsigned int BufferLength, unsigned char * SearchString,
; unsigned int SearchStringLength);
;
; Returns a pointer to the first match for SearchString in Buffer, or
; a NULL pointer if no match is found. Buffer should not start at
; offset 0 in the data segment to avoid confusing a match at 0 with
; no match found.
```

Parms struc

dw 2 dup(?) ;pushed BP/return address

Buffer;pointer to buffer to search

BufferLength dw _?;length of buffer to search

SearchString dw _?;pointer to string for which to search

SearchStringLength?;length of string for which to search

Parms ends

small

.code

public _FindString

_FindString proc near

pushbp.;preserve caller's stack frame

movbp,sp ;point to our stack frame

pushsi.;preserve caller's register variables

pushdi _

cld;make string instructions increment pointers

movsi,[bp+SearchString];pointer to string to search for

movbx,[bp+SearchStringLength] ;length of string

and bx,bx

jz...FindStringNotFound;no match if string is 0 length


```

        cld                                ;заставить строковые инструкции увеличивать указатели
>movsi, [bp+SearchString];pointer to string to search for
movcx,[bp+SearchStringLength] ;length of string
jcxzFindStringNotFound;no match if string is 0 length
movdx, [bp+BufferLength];length of buffer
subdx,cx;difference between buffer and search
                                ; lengths
                                jc...FindStringNotFound ;no match if search string is
                                ; longer than buffer
incdx ; difference between buffer and search string
                                ; lengths, plus 1 (# of possible string start
                                ; locations to check in the buffer)

movdi, ds
moves, di
movdi, [bp+Buffer];point ES:DI to buffer to search thru
movbx, [bp+ScanCharOffset] ;offset in string of character
                                ; on which to scan

adddi,bx;point ES:DI to first buffer byte to scan
movax,[si+bx] ;put the scan character in AL
incbx _;set BX to the offset back to the start of the
                                ; potential full match after a scan match,
                                ; accounting for the 1-byte overrun of
                                ; REPNZ SCASB

FindStringLoop:
    movcx, dx;put remaining buffer search length in CX
    repnzscasb;scan for the scan byte
    jnzFindStringNotFound ;not found, so there's no match
                                ;found, so we have a potential match-check the
                                ; rest of this candidate location

    pushdi _;remember the address of the next byte to scan
    movdx,cx;set aside the remaining length to search in
                                ; the buffer

    subdi,bx;point back to the potential start of the
                                ; match in the buffer

    movsi, [bp+SearchString];point to the start of the string
    movcx,[bp+SearchStringLength] ; string length
    shrex, 1;convert to word for faster search
    jncFindStringWord;do word search if no odd byte
    cmpsb;compare the odd byte
    jnzFindStringNoMatch;odd byte doesn't match, so we
                                ; haven't found the search

string here FindStringWord:
    jcxzFindStringFound;if the string is only 1 byte long,
                                ; we've found a match

    repzcmpsw;check the rest of the string a word at a time
    jz...FindStringFound;it's a match

FindStringNoMatch:
    popdi _;get back pointer to the next byte to scan
    anddx, dx;is there anything left to check?
    jnzFindStringLoop;yes-check next byte

FindStringNotFound:
    subax,ax;return a NULL pointer indicating that the
    jmpFindStringDone; string was not found

FindStringFound:
    popax _;point to the buffer location at which the
    subax,bx; string was found (earlier we pushed the
                                ; address of the byte after the scan match)

FindStringDone:
    popdi _;restore caller's register variables
    popsi _
    popbp.;restore caller's stack frame
    ret

_FindString
endp

end

```

<ЛИСТИНГ 9.3 L9-3.C

```

/* Program to exercise buffer-search routines
in Listings 9.1 & 9.2 */ #включить <stdio.h>
#включить <string.h> #определить длину ОТОБРАЖЕНИЯ
40 extern unsigned char * FindString(беззнаковый
символ *, беззнаковый int,
    unsigned char *, unsigned int, беззнаковый int);
void main(недействительный);
static unsigned char TestBuffer[] = "Когда в ходе
человеческих \ событий одному человеку становится
необходимым распустить \ политические группы, которые
соединили их с другими, и \ занять среди держав земли
отдельное и равноправное положение \ на что законы природы
и Бога природы дают им право..."; void main() {
    статический символ без знака testString[] = "равно";
    временный буфер символа без знака[DISPLAY_LENGTH+1];
    символ без знака *MatchPtr;
    /* Найдите testString и сообщите о результатах */
    if ((MatchPtr = FindString(TestBuffer,
        (unsigned int) strlen(TestBuffer), тестовая строка,
        (unsigned int) strlen(testString), 1)) == NULL) {
        /* Тестовая строка не найдена */
        printf("\'%s \'не
найдено \ n", тестовая строка);
    }
}

```

```
<>} else {
    /* TestString was found. Zero-terminate TempBuffer; strncpy
       won't do it if DISPLAY_LENGTH characters are copied */
    TempBuffer[DISPLAY_LENGTH] = 0;
    printf("\n%s\n" found. Next %d characters at match:\n\n%s\n\n",
        TestString, DISPLAY_LENGTH,
        strncpy(TempBuffer, MatchPtr, DISPLAY_LENGTH));
}
```

<Вы заметите, что в листинге 9.2 я не использовал таблицу частот символов в английском тексте чтобы определить символ, который следует сканировать, а не позволить вызывающему пользователю сделать этот выбор. Каждый буфер байтов имеет уникальные характеристики, и частота использования английских букв вполне может быть неподходящей. Что, если буфер заполнен текстом на французском? Кириллица? Что, если это не текст который ищется? Возможно, приложению стоило бы создать динамическую таблицу частот для каждого буфера, чтобы можно было выбрать наилучший символ сканирования для каждого поиска. Или, возможно, нет, если поиск не критичен по времени или буфер невелик.

Дело в том, что вы можете значительно повысить производительность, понимая природу данных, с которыми вы работаете. (Кстати, это в равной степени верно и для языков высокого уровня программирование.) Листинг 9.2 очень похож на и лишь немного сложнее. чем в листинге 9.1; разница заключается не в смазке для локтей или подсчете циклов, а в органическая интегрирующая оптимизирующая технология, которую мы все носим в своих головах.

Короткие сорта

Дэвид Стаффорд (недавно из Borland и Borland Japan), который является одним из лучших программистов на ассемблере, которых я когда-либо встречал, написал процедуру с возможностью вызова на С это сортирует массив целых чисел в порядке возрастания. Это было бы не особенно важно примечательно, за исключением того, что процедура Дэвида, показанная в листинге 9.4, имеет длину ровно 25 байт. Посмотрите на код; вы будете продолжать говорить себе: “Но это не работает ... о, да, я думаю" это работает”. Как говорится в рекламе соуса для спагетти Prego, он там — и какая это работа упаковка. В любом случае, Дэвид говорит, что 24-байтовая процедура сортировки ускользает от него, и он хотел бы знать, может ли кто-нибудь придумать что-нибудь подобное.

ЛИСТИНГ 9.4 L9-4.ASM

```

    перемещение dx, [bx]      ;поменять местами два соседних целых числа
    xchg dx, [bx +2]
    xchg dx, [bx]             ; мы расположили их в правильном порядке?
Наверх:   стр dx, [bx]
    j1 Наверх                ; нет, поменяй их местами обратно
    inc bx _                 ;перейти к следующему целому числу
    inc bx _
    цикл top _
    dop dx _                 ; получить обратный адрес (точку входа)
    pop cx _                 ; получить счетчик
    pop bx _                 ; получить указатель
    толкать bx _             ; восстановить указатель
    dec cx _                 ; уменьшить количество
_sort:   толкать cx _        ; количество сохранений
    толкать dx _             ; восстановить обратный адрес
    jg... Наверх             ;если cx > 0

    ret
конец
```

Полное 32-разрядное разделение

Одним из наиболее раздражающих ограничений x86 является то, что, хотя операнд dividend для Команда DIV может иметь размер 32 бита, как делитель, так и результат должны быть 16 бит.

Это особенно раздражает в отношении результата, потому что иногда вы просто не знаете, больше ли отношение делимого к делителю 64К-1 или нет — и если если вы угадали неправильно, вы получите это ужасное прерывание деления на ноль. Итак, что же делать когда результат может не поместиться в 16 бит или когда делимое больше 32 бит? Отступить вернуться к подходу подразделения программного обеспечения? Это сработает, но так медленно.

Есть другой метод, который намного быстрее, чем чисто программный подход, хотя и не такой гибкий. Этот метод допускает сколь угодно большие дивиденды и результаты, но делитель по-прежнему ограничен 16 битами. Это не идеально, но это решает ряд проблем, в частности, исключается возможность прерывания деления на ноль из-за слишком большого результата.

Этот метод не включает в себя ничего более сложного, чем разбиение деления на фрагменты размером со слово, начинающиеся с самого значимого слова в делимом. Самое значимое слово делится на делитель (без возможности переполнения, потому что есть только 16 бит в каждом); затем остаток добавляется к следующим 16 битам делимого, и процесс повторяется, как показано на рисунке 9.3. Этот процесс эквивалентен делению вручную, за исключением того, что здесь мы прекращаем перенос остатка вручную только после каждое слово в разделителе; аппаратное разделение позаботится обо всем остальном. В листинге 9.5 показано функция для деления произвольно большого дивиденда на 16-разрядный делитель, и в листинге 9.6 показано пример деления большого дивиденда. Обратите внимание, что тот же принцип может быть применен к обработке сколь угодно больших дивидендов в коде машинного режима 386, но в этом случае операция может выполняться по dword, а не по слову, одновременно.

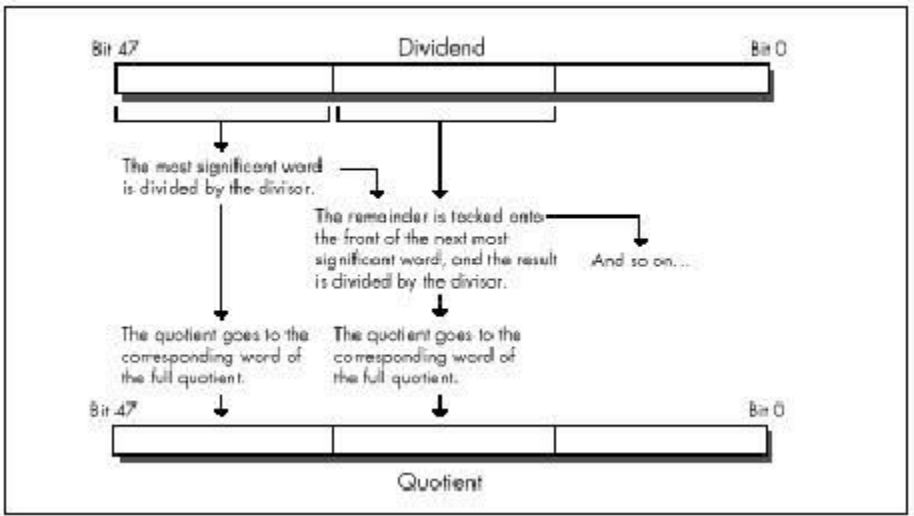


Рисунок 9.3 Быстрое разделение на несколько слов на 386.

Что касается обработки знакового деления со сколь угодно большими дивидендами, это можно сделать легко достаточно запомнить знаки дивиденда и делителя, разделив абсолютное значение значение дивиденда по абсолютному значению делителя и применение сохраненных знаков чтобы установить правильные знаки для частного и остатка. Могут быть более хитроумные способы чтобы получить тот же результат, например, используя IDIV; если вы знаете о таком, напишите мне список книг Coriolis Group.

ЛИСТИНГ 9.5 L9-5.ASM

```
>; Divides an arbitrarily long unsigned dividend by
a 16-bit unsigned ; divisor. C near-callable as:
    unsigned int Div(unsigned int * Dividend,
    int DividendLength, unsigned int Divisor,
    unsigned int * Quotient);
; Returns the remainder of
the division. ;
; Tested with TASM 2.

<>parms struc
    dw _2 dup (?) ;pushed BP & return address
    Dividend dw?;pointer to value to divide, stored in Intel
    ; order, with lsb at lowest address, msb at
    ; highest. Must be composed of an integral
    ; number of words DividendLength dw?;#
of bytes in Dividend. Must be a multiple
    ; of 2
    Divisor dw?;value by which to divide. Must not be zero,
    ; or a Divide By Zero interrupt will occur
    Quotient dw?;pointer to buffer in which to store the
    ; result of the division, in Intel order.
    ; The quotient returned is of the
same
    ; length as the dividend
```

```

        .модель                маленькая
        .код
        общедоступный         _Div
_Div proc рядом

    толкать    bp.                ;сохранить фрейм стека вызывающего объекта
    движение   bp, sp              ; указать на наш фрейм стека
    толкать    si _                ; сохранить регистровые переменные вызывающего абонента
    толкать    di _

    std                    ; мы работаем с msb на lsb
    mov ax, ds
    mov  es, ax              ; для STO
    mov  cx, [bp + делимая длина]
    SUB  cx, 2
    mov  si, [bp + дивиденд]
    добавить si, cx          ; указать на последнее слово дивиденда
                                ; (наиболее значимое слово)
    mov  di, [bp + Частное]
    di, cx                   ; указать на последнее слово частного
                                ; буфер (наиболее значимое слово)

    mov  bx, [bp + делитель]
    shr  пример, 1
    inc  cx _                ; количество обрабатываемых слов
    sub  dx, dx              ; преобразовать начальное слово-делитель
                                ; в 32-разрядное ;значение для DIV

DivLoop:

    lodsw                    ; получить следующее по значению слово-делитель
    div bx
    stosw                    ;сохраните это слово частного
                                ;DX содержит остаток в этой точке,
                                ; готов добавить к следующему слову-делителю

    цикл      DivLoop
    mov  ax, dx              ;верните остаток ; восстановить
    cld                    настройку флага направления по умолчанию ; восстановить
    pop  di _                регистровые переменные вызывающего объекта
    pop  si _
    pop  давление.          ; восстановить фрейм стека вызывающего объекта
    ret

_Div endp

конец
```

ЛИСТИНГ 9.6 L9-6.C

/* Пример использования функции Div для выполнения деления, когда результат не умещается в 16 битах */

```

#include <stdio.h>

extern unsigned int Div(unsigned int * Dividend,
                        int DividendLength, беззнаковый делитель
int,
                        беззнаковый int * Частное);

main() {
    беззнаковый long m, i = 0x20000001;
    беззнаковый int k, j = 0x10;

    k = Div(( unsigned int *)&i, sizeof(i), j, (unsigned int *)&m);
    printf("%lu / %u = %lu r % u \ n", i, j, m, k);
}
```

Пересмотрено "Сладкое местечко"

Еще в томе 1, номер 1 "КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ" (апрель / май 1990 г.) Я написал самый первый номер журнала НАХ (№ 1), в котором превозносились достоинства размещения ваших самых обычно используемые автоматические переменные (на основе стека) в “удобном месте” стека, в области от +127 до -128 байт от BP, указатель фрейма стека. Причина заключалась в том, что 8088 может хранить смещения адресации, которые попадают в этот диапазон, в один байт; для больших перемещений требуется полное слово памяти, что увеличивает размер кода на по байту на команду и, таким образом, снижает производительность из-за увеличения времени выборки команды.

Это приобретает новое значение в родном режиме 386, где отклонение от оптимального варианта стоит не один, а два или три байта. Там, где у 8088 было два возможных смещения размеры, либо в байтах, либо в словах, на 386 есть три возможных размера: byte, word или dword. Однако в собственном режиме (32-разрядный защищенный режим) требуется байт префикса, чтобы использовать смещение размером со слово, поэтому для переменной, расположенной за пределами контрольной точки, требуется либо два дополнительных байта (дополнительный байт смещения плюс байт префикса), либо три дополнительных байта

(смещение dword, а не смещение байта). В любом случае количество инструкций увеличивается вызывает тревогу.

Производительность может пострадать, а может и не пострадать из-за отсутствия необходимого, в зависимости от процессора, архитектуры памяти и сочетания кода. На 486-м префикс байт часто обходится дороже цикл; на 386SX увеличенный размер кода часто снижает производительность, потому что инструкции должен быть извлечен через полупинтовую 16-битную шину; на 386 эффект зависит от набора команд и наличия кэша.



В целом, однако, так же важно сохранить наиболее используемые переменные в самое приятное место стека в 386 native mode, как и в 8088.

В ассемблере легко контролировать организацию вашего фрейма стека. Однако в C, вам нужно будет выяснить схему распределения, которую ваш компилятор использует для автоматического распределения переменные и объявляйте автоматику соответствующим образом для получения желаемого эффекта. Это может быть сделано: я сделал это в Turbo C несколько лет назад и уменьшил размер программы (по общему признанию, большой) на несколько К —неплохо, если учесть, что “сладкий точечная ” оптимизация, по сути, бесплатна, без реорганизации кода, изменения логики или сложного мышления.

Жесткий подсчет циклов

Далее мы переходим к пункту, который понравится счетчикам циклов, особенно потому, что он включает в себя явно некорректную документацию со стороны Intel. Согласно документам Intel, все Инструкции RCR и RCL, которые выполняют вращения с помощью флага переноса, как показано на Рисунок 9.4, выполняют 9 циклов на 386-м регистре при работе с операндом. Мои измерения показывают, что время выполнения в 9 циклов почти справедливо для многобитных сквозной перенос, который я рассчитал по 8 циклов за штуку; например, RCR AX, CL занимает 8 циклов на моем 386, как и RCL DX, 2. Сравните это с ROR и ROL, которые могут вращаться содержимое регистра на любое количество битов всего за 3 цикла.

Однако поворот на один бит через флаг переноса не занимает 9 циклов, вопреки *Справочному руководству программиста Intel 80386, или даже 8 циклов. Фактически, RCR reg, 1 и RCL reg, 1 занимают 3 цикла, точно так же, как ROR, ROL, SHR и SHL. По крайней мере, так быстро они работают на моем 386, и я очень сомневаюсь, что вы найдете разное время выполнения на других 386s. (Однако, пожалуйста, дайте мне знать, если найдете!)*

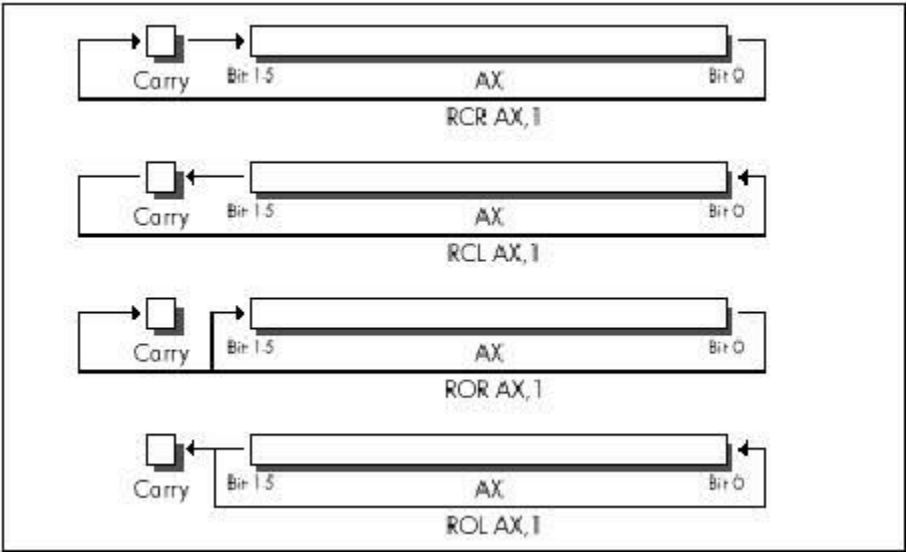


Рисунок 9.4 Выполнение инструкций поворота с использованием флага Переноса.

Интересно, что, согласно справочнику программиста микропроцессора Intel i486 Manual, 486 может выполнять RCR или RCL регистр на один бит за 3 цикла, но для этого требуется от 8 и 30 циклов для выполнения многобитового регистра RCR или RCL!

Особого урока здесь нет, просто предостережение относиться с подозрением к многобитовому RCR и RCL, когда производительность имеет значение, и к документации по времени цикла следует относиться со всей серьезностью.

Встроенные переходы Far

Вы когда-нибудь задумывались, как закодировать переход far к абсолютному адресу в ассемблере язык? Вероятно, нет, но если вы когда-нибудь это сделаете, вы будете рады следующему пункту, потому что очевидное решение не работает. Вы могли бы подумать, что все, что потребуется для перехода, скажем, 1000: 5 было бы JMP FAR PTR 1000: 5, но вы ошибаетесь. Это даже не соберите. Затем вы можете подумать о создании в памяти дальнего указателя, содержащего 1000: 5, как показано ниже:

```
>Ptr dd?  
    mov word ptr [Ptr],5  
    mov word ptr [Ptr+2],1000h  
    jmp [Ptr]
```

<Это сработает, но за счет снижения производительности. На 8088, JMP DWORD PTR [*mem*] (an
непрямой дальний прыжок) занимает не менее 37 циклов; JMP DWORD PTR *label* (прямой дальний прыжок)
занимает всего 15 циклов (плюс, почти наверняка, несколько циклов для выборки инструкций). На
386, задокументировано, что непрямой дальний прыжок занимает не менее 43 циклов в реальном режиме (31
дьюим защищенный режим); задокументировано, что прямой дальний прыжок занимает не менее 12 циклов, что
примерно в три раза быстрее. По правде говоря, разница между этими двумя далеко не так велика;
самый быстрый прыжок, который я измерил для прямого дальнего прыжка, равен 21 циклу, и я измерил непрямой дальний
прыжки совершаются со скоростью 30 циклов, так что прямой все равно быстрее, но не на столько. (Ох уж эти циклические
блужды документации по времени!) Кроме того, задокументировано, что прямой дальний прыжок занимает не
менее 27 циклы в защищенном режиме; почему такая большая разница в защищенном режиме, я понятия не имею.

В любом случае, возвращаясь к нашей первоначальной проблеме перехода к 1000: 5: хотя косвенный
дальний переход сработает, прямой дальний переход по-прежнему предпочтительнее.

В листинге 9.7 показана короткая программа, которая выполняет прямой дальний вызов 1000:5. (Не
запускайте ее, если только вы не хотите привести к сбою свою систему!) Это делается путем создания
фиктивного сегмента в 1000H, так что метка FarLabel может быть создана с желаемым атрибутом far в
правильное расположение. (Сегменты, созданные с помощью “AT”, не приводят к генерации каких-либо
фактических байтов или выделению какой-либо памяти; это просто шаблоны.) Это немного запутанно, но,
по крайней мере, это работает. Возможно, есть решение получше; если оно у вас есть, поделитесь им.

ЛИСТИНГ 9.7 L9-7.ASM

```
>; Program to perform a direct far jump to  
address 1000:5. ; *** Do not run this  
program! It's just an example of how ***  
; *** to build a direct far jump to an absolute  
address*** ; ; Tested with TASM 2 and MASM 5.
```

```
<>FarSegsegment at 01000h  
    org 5  
FarLabel label far  
FarSegends  
    .modelsmall  
    .code  
start:  
    jmpFarLabel  
endstart
```

<Кстати, если вам интересно, как я до этого додумался, я просто применил
своего хорошего друга Давнее правило Дэна Илловски по работе с MASM:

Если очевидное не работает (а обычно так и не работает), просто попробуйте
все, что вам придет в голову, неважно, насколько нелепо, пока не найдете то, что
работает — правило с большим количеством история на своей стороне.

Настройка 32-разрядных регистров: соотношение времени и пространства

Чтобы закончить эту главу, рассмотрим эти два пункта. Сначала в 32-разрядном защищенном режиме,

```
>sub eax,eax  
inc eax
```

<выполнение занимает 4 цикла, но имеет длину всего 3 байта, в то время как

```
>mov eax,1
```

<выполнение занимает всего 2 цикла, но имеет длину 5 байт (поскольку константы собственного режима dwords и инструкция mov не подписывают расширение). Оба фрагмента кода являются способами установите EAX равным 1 (хотя первое влияет на флаги, а второе - нет); это классический компромисс между скоростью и пространством. Второе,

```
>or _ebx,-1
```

<выполнение занимает 2 цикла и имеет длину 3 байта, в то время как

```
>move bx,-1
```

<выполнение занимает 2 цикла и имеет длину 5 байт. Обе инструкции устанавливают EBX равным -1; это классический компромисс —ну и дела, это вообще не компромисс, не так ли? или лучший способ установить 32-битный регистрируйтесь на все 1-битные значения, так же как sub или xor - лучший способ установить регистр на все 0-битные значения. Кто одобрил бы это? Это просто показывает, как 32-битные смещения и константы 386 собственный режим изменяет привычный ландшафт оптимизации 80x86.

Однако имейте в виду, что я обнаружил, что OR, И, ADD, и тому подобное на цикл медленнее, чем mov при работе с непосредственными операндами на 386 при некоторых обстоятельствах, по причинам, которые до сих пор ускользают от меня. Это просто усиливает первое правило оптимизации: Измерьте свой код в действии и не доверяйте документированному времени цикла.

Глава 10 – Кодирование пациента, быстрее Код

Как Быстрая работа может привести к выполнению обхода

Каждое воскресенье мой дедушка разгадывает кроссворд "Нью-Йорк Таймс". Чернилами. С без единого изъяна.

Актуальность которого станет очевидна в мгновение ока.

Кто такой мой дедушка, так это выдающийся специалист по подбору шаблонов. Ты тоже специалист по подбору шаблонов . Я тоже. Мы ничего не можем с этим поделать; это приходит вместе с территорией. Попробуйте сосредоточиться на тексте, а не на читать его. Не получается. Вы слышите голос кого-то знакомого и не узнаете его? Я не могу. И как из Девяти миллиардов Имен Бога получается, что мы способны мгновенно узнать одно лицо из тысяч, которые мы видели за свою жизнь, даже спустя годы, с другого ракурса и в другом свете? Хотя мы принимаем их как должное, наши возможности подбора рисунка, безусловно, являются чудом по сравнению с хлебом и рыбой.

Однако под “сопоставлением с образцом” я подразумеваю нечто большее, чем просто распознавание. Я имею в виду, что мы в целом способны получать сложные и часто кажущиеся прискорбно неадекватными данные, мгновенно сопоставляйте это невероятно гибким способом с нашим прошлым опытом, экстраполируйте, и приходите к потрясающим выводам, которые компьютеры вряд ли могут делать вообще. Отличным примером являются кроссворды; учитывая пару букв и загадочную подсказку, мы каким-то образом можем придумать одно из нескольких сотен тысяч известных нам слов . Попробуйте написать программу для этого! Более того, мы не обрабатываем данные методом последовательного перебора, как это делают компьютеры. Решения, как правило, виртуальны мгновенный или вообще отсутствующий; ни один из этих $N \log N$ или N^2 время выполнения для нас.

Само собой разумеется, что сопоставление с шаблоном - это хорошо; более того, это большая часть кем мы являемся, и, как правило, чем быстрее мы это делаем, тем лучше. Хотя и не всегда. Иногда недостаточной информации действительно недостаточно, и в нашей спешке получить головокружительный порыв поиска решения приводит к неправильным или неоптимальным выводам достигнуто, как подтвердит любой, кто когда-либо разгадывал воскресный кроссворд Times. До сих пор,

мой дедушка каждое воскресенье разгадывает эту головоломку чернилами. В чем его секрет? Терпение и дисциплина. Он никогда не вставляет слово, пока не утвердит его в своей голове с помощью пересекающихся слов, независимо от того, насколько сильным может быть желание записать что-то там, где он может видеть это и ощущение, что он чего-то добивается.

Здесь прослеживается удивительно близкая параллель с программированием. Программирование, безусловно, подобие сопоставления с образцом в том смысле, который я описал выше, и, как в случае с кроссвордами головоломки, слишком быстрое следование своим инстинктам программирования может стать помехой. Для многих программисты, включая меня, испытывают сильное желание найти работоспособный подход к конкретной проблеме и начать кодировать ее прямо сейчас, то, что некоторые люди называют “взломом” программы. Выполнять первое, что приходит в голову вашему программисту по сопоставлению шаблонов программирования, может быть очень весело; вы получаете мгновенное удовлетворение и ощущение безграничного творчества. Лично я всегда стремился получить результаты от своей работы как можно скорее; я тяготел к графике из-за ее мгновенного и очень заметного удовлетворения. Со временем однако я научился терпению.



Я стал тратить все большую часть своего времени на выбор алгоритмы, проектирование и просто на то, чтобы дать своему разуму время успокоиться, чтобы
работайте над проблемами и придумывайте неочевидные подходы перед
программированием; и я обнаружил, что дополнительное время с лихвой окупается
как за счет
сокращения времени кодирования, так и за счет улучшенных программ.

В этой главе я собираюсь познакомить вас с простой, но наглядной историей болезни, которая наглядно демонстрирует мудрость отсрочки получения удовольствия, когда сталкиваешься с программированием проблемы, чтобы у вашего ума было время обдумать проблемы с других точек зрения. В альтернативные решения, которые вы найдете, делая это, могут показаться очевидными, как только вы их придумаете. Они могут даже не сильно отличаться от ваших первоначальных решений. Однако часто они будут намного лучше — и у вас даже не будет возможности решить, лучше они или нет, если вы возьмете первое, что придет вам в голову, и побежите с этим.

Аргументы в пользу отсроченного удовлетворения

Однажды я решил почитать "Алгоритмы" Роберта Седжвика (Эддисон-Уэсли), которая оказалась замечательной, стимулирующей и полезнейшей книгой, которую я настоятельно рекомендую. Моя история, однако, включает только то, что произошло в первых 12 страниц, ибо именно на этих страницах Седжвик обсуждал алгоритм Евклида.

Алгоритм Евклида (открытый Евклидом, известным специалистом по евклидовой геометрии, очень давно назад, когда компьютеры еще использовали основную память) - это простой алгоритм это решает одну из простейших задач, которые только можно вообразить: нахождение наибольшего общего делителя (GCD) двух натуральных чисел. Седжвик указывает, что это полезно для уменьшения дроби до наименьших слагаемых. Я уверен, что это полезно и для других целей, хотя на ум ничего не приходит. (Давным-давно я написал статью о небольшой оптимизации кода, который даже отдаленно не зависел от времени, и был завален письмами, сообщающими мне об этом так. Я знал, что это не критично по времени; это был просто хороший пример. Так что пока закрой глаза и представьте, что поиск НОД не только необходим, но и должен быть выполнен как можно быстрее потому что это идеально подходит для того, что я хочу подчеркнуть здесь и сейчас. Хорошо?)

Тогда задача под рукой проста: найдите наибольшее целое значение, которое равномерно делит два произвольных положительных целых числа. Вот и все, что нужно сделать. Так что разогревай свой шаблон сопоставителя ... и вперед!

Синдром грубой силы

У меня странное чувство, что ты уже выяснил, как найти GCD, еще до того, как я сказал “иди”. Это то, что я делал, когда читал алгоритмы; прежде чем я прочитал еще одно слово, я должен был

разобраться в этом сам. Программисты такие: дайте им задачу, и они поймут немедленно остекленейте, поскольку они пытаются решить проблему еще до того, как вы закроете рот. Это мгновенный отклик, безусловно, может быть впечатляющим, но он также может иметь неприятные последствия, как это было в моем случае.

Видите ли, я стал жертвой распространенной ошибки программирования, синдрома “грубой силы”. основной этого синдрома является то, что существует множество проблем, требующих очевидного применения грубой силы решения — с одним небольшим недостатком. Недостатком является то, что если бы вы попытались применить решение методом перебора вручную, то есть решить одну проблему с помощью карандаша и бумаги или калькулятор — обычно это требует от вас терпения и дисциплины, чтобы работать над проблемой в течение примерно семисот лет, не считая приема пищи и сна, чтобы получить ответ. Поиск всех простых чисел меньше 1 000 000 - это хороший пример; просто разделите каждое число до 1 000 000 на каждое меньшее число и посмотрите, что осталось. На протяжении большей части истории человечества, люди были вынуждены придумать более умные решения, такие как Сито Эратосфена (мы были бы в большом проблема, если бы у древних греков были компьютеры), главным образом потому, что примерно через пять минут работы методом перебора внимание людей отвлекается на другие важные имеет значение, например, как далеко бумажный самолетик улетит из окна второго этажа.

Однако в наши дни это не так. Компьютеры любят скучную работу; они очень терпеливы и дисциплинированный, и, кроме того, один человеческий год = семь собачьих лет = два миллиона компьютерных лет. Итак, когда мы сталкиваемся с проблемой, которая имеет очевидный, но чрезвычайно длительный характер решение, мы склонны говорить: “А, пусть это сделает компьютер, это быстро”, - и возвращаться к изготовлению бумажных самолетиков. К сожалению, решения методом перебора, как правило, работают медленно, даже когда выполняются современными микрокомпьютерами, которые способны выполнять несколько MIPS, за исключением когда я опаздываю на встречу и хочу закончить компиляцию и запустить еще один тест перед уходом, и в этом случае кристалл в моем компьютере, по-видимому, предназначен для автоматически возвращается к частоте 1 Гц.)

Решение, которое я мгновенно придумал для поиска GCD, примерно такое же грубое, как вы можете получить: Разделите как большее целое число (iL), так и меньшее целое число (iS) на каждое целое число, равное или меньше меньшего целого числа, пока не будет найдено число, которое делит оба равномерно, как показано на рисунке 10.1. Это работает, но это паршивое решение, требующее столько сколько есть * 2 деления; очень дорого, особенно при больших значениях iS . Например, найти для GCD 30,001 и 30,002 потребовалось бы 60,002 делений, что само по себе, без учета тестов и ветвей, заняло бы около 2 секунд на 8088 и более более 50 миллисекунд даже на частоте 25 МГц 486 — это очень большой срок для компьютерных лет, и не незначительный и для человеческих лет.

Листинг 10.1 представляет собой реализацию метода грубой силы для вычисления НОД. В таблице 10.1 показано, сколько времени требуется при таком подходе для нахождения НОД для нескольких пар целых чисел. Поскольку ожидалось, что производительность крайне низкая, когда значение iS велико.

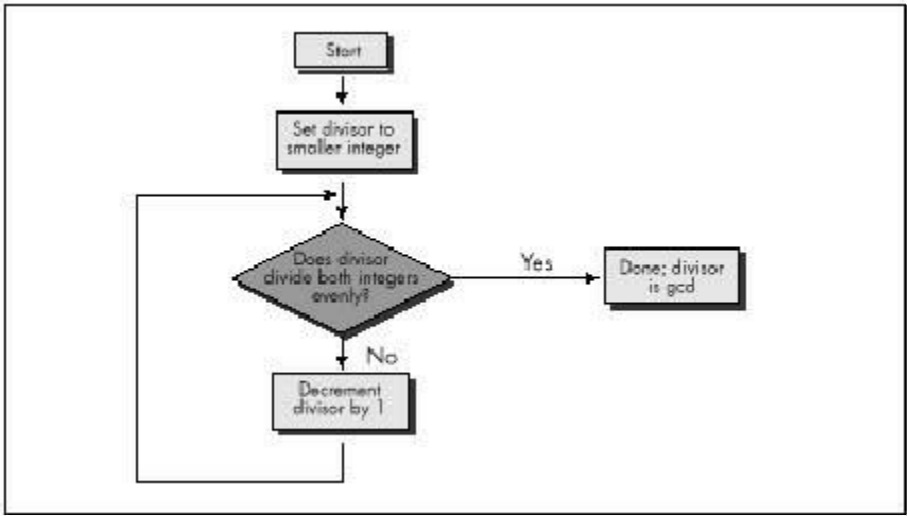


Рисунок 10.1 Использование алгоритма перебора для поиска GCD.

Пары целых чисел, для которых нужно найти GCD

Таблица 10.1 Производительность реализаций алгоритма GCD.

	90 & 27	42 & 998	453 & 121	& 165	27432 & 17550
	60ps	110 с	311 мс	426 с	43580 с
Листинг 10.1 (Перебор)	(100%)	(100%)	(100%)	(100%)	(100%)
Листинг 10.2 (Вычитание)	25	72		67 280 72	
	(42%)	(65%)	(22%)	(66%)	(0.16%)
Листинг 10.3 (Разделение: рекурсивный код Евклида	20	33	48	32	53
алгоритм)	(33%)	(30%)	(15%)	(8%)	(0.12%)
Листинг 10.4 (С-версия рекурсивных данных	12	17	25	16	26
Алгоритм Евклида; нормальная оптимизация)	(20%)	(15%)	(8%)	(4%)	(0.06%)
	12	16	20	15	23
Листинг 10.4 (/Ох = максимальная оптимизация)	(20%)	(15%)	(6%)	(4%)	(0.05%)
Листинг 10.5 (Версия данных для сборки	10	10	15	10	17
рекурсивный алгоритм Евклида)	(17%)	(9%)	(5%)	(2%)	(0.04%)

Примечание: Эффективность выполнения списков с 10.1 по 10.5 при нахождении наибольшего общего делителя различных пар целых чисел. Время указано в микросекундах. Проценты представляют время выполнения в процентах от времени выполнения листинга 10.1 для того же целого числа пара. Списки 10.1-10.4 были скомпилированы с использованием Microsoft C / C ++, за исключением, как уже отмечалось, использовалась оптимизация по умолчанию. Все время измерялось с помощью Zen timer (из главы 3) на кэшированном 386-м частотном диапазоне 20 МГц.

ЛИСТИНГ 10.1 L10-1.C

```
/* Находит и возвращает наибольший общий делитель двух положительных
   целых чисел. Работает путем пробы каждого целого делителя между
   меньшим из двух целых чисел и 1, пока не будет найден делитель, который делит
   оба целых числа поровну. Весь код на C протестирован
   с помощью компиляторов Microsoft
   и Borland.*/

>unsigned int gcd(unsigned int int1, unsigned int int2) {
    unsigned int temp, trial_divisor;
    /* Swap if necessary to make sure that int1 >= int2 */
    if (int1 < int2) {
        temp = int1;
        int1 = int2;
        int2 = temp;
        /* Now just try every divisor from int2 on down, until a common
           divisor is found. This can never be an infinite loop because
           1 divides everything evenly */
        for (trial_divisor = int2; ((int1 % trial_divisor)!= 0)
            ((int2 % trial_divisor)!= 0); trial_divisor-)
            return(trial_divisor);
    }
```

<Напрасные прорывы

Первое решение проблемы GCD, предложенное Седжвиком, было в значительной степени тем, которое я придумал. Затем он указал, что GCD для iL и iS совпадает с GCD для iL-iS и iS. Это было очевидно (как только Седжвик указал на это); по самой природе разделения любой число, которое делит iL равномерно nL раз и равномерно nS раз, должно делить iL-iS равномерно nL-nS раз. Учитывая это, я немедленно разработал новый, более быстрый подход, показанный в листинге 10.2.

ЛИСТИНГ 10.2 L10-2.C

```
/* Находит и возвращает наибольший общий делитель двух положительных
   целых чисел. Работает путем вычитания меньшего целого числа из
   большего целое число до тех пор, пока либо значения не совпадут (в этом случае
   это gcd), либо большее целое число не станет меньшим из
   два, в этом случае два целых числа меняются
   ролями, и
   процесс вычитания продолжается. */

>unsigned int gcd(unsigned int int1, unsigned int int2) {
    unsigned int temp;
    /* If the two integers are the same, that's
       the gcd and we're
       done */
    if (int1 == int2)
        return(int1);
    if (int1 < int2)
        temp = int2 - int1;
    else
        temp = int1 - int2;
    return(gcd(int1, temp));
}
```

```
<>if (int1 == int2) {
    return(int1); /* Swap if necessary
to make sure that int1 >= int2 */
if (int1 < int2) {
    temp = int1;
    int1 = int2;
    int2 = temp;
    /* Subtract int2 from int1 until int1 is no longer the larger of
the two */
do {
    int1 -= int2; } while (int1 > int2);
/* Now recursively call this
function to continue the process */
return(gcd(int1, int2));
```

<В листинге 10.2 iS многократно вычитается из iL до тех пор, пока iL не станет меньше или равен iS . Если iL становится равным iS, то это GCD; в качестве альтернативы, если iL становится меньше, чем iS, iL и iS переключают значения, и процесс повторяется, как показано на рисунке 10.2. количество итераций, требуемых этим подходом относительно листинга 10.1, сильно зависит от значений iL и iS, поэтому это не всегда быстрее, но, как показано в таблице 10.1, листинг 10.2, как правило, намного лучший код.

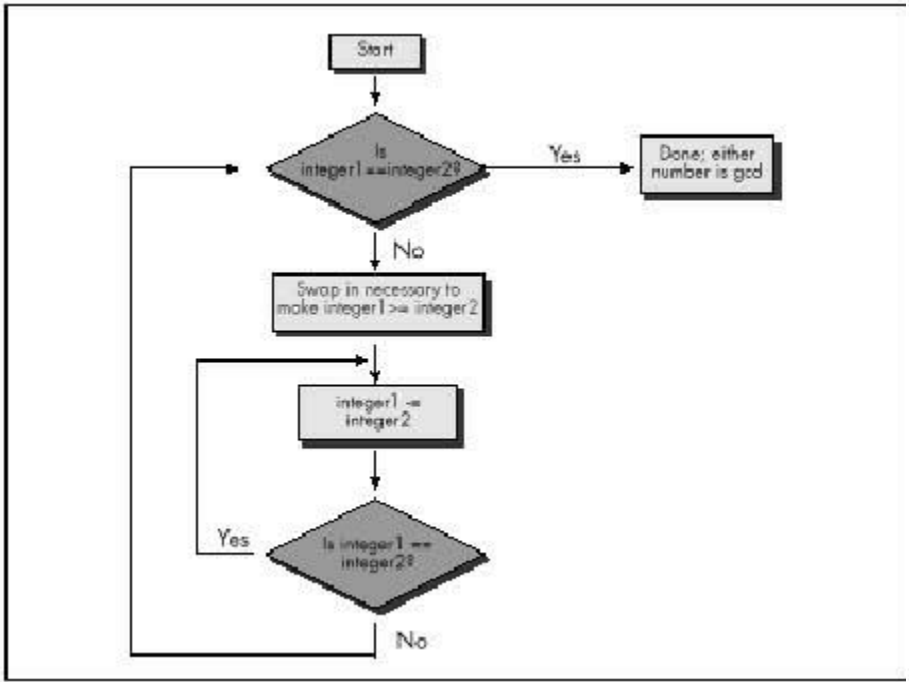


Рисунок 10.2 Использование алгоритма повторного вычитания для поиска GCD.

Листинг 10.2 - гораздо более серьезная ошибка, чем листинг 10.1, несмотря на то, что он быстрее. Листинг 10.1 - это очевидно, что это упрощенный подход с применением грубой силы; никто не сможет спутать его ни с чем другим. Немного подумав, это можно ускорить любым из нескольких способов. (Просто пропуская тестирование всех делителей между iS и iS /2, не включительно, сократило бы время в наихудшем случае например, вдвое; это не особенно хорошая оптимизация, но она иллюстрирует, как Листинг 10.1 можно легко улучшить.) Листинг 10.1 - халтурная работа, требующая вдохновения.

Листинг 10.2, с другой стороны, вдохновил меня — и в значительной степени растратил его впустую из-за спешки. Если бы Седжвик не сказал мне обратное, я вполне мог бы предположить, что Листинг 10.2 был оптимизирован, ошибка, которую я никогда бы не допустил с листингом 10.1. Я пережил концептуальный прорыв, когда понял точку зрения Седжвика: меньший число может быть вычтено из большего числа, не влияя на их НОД, тем самым недорого уменьшая масштаб проблемы. И я спешу сделать это прорывная реальность, я пропустил весь ее размах. Как говорит Седжвик на следующей странице, число, которое получается путем вычитания iS из iL до тех пор, пока iL не станет меньше, чем iS, в точности равно то же самое, что и остаток, который получается путем деления iL на iS — опять же, это присуще природа деления — и это основа алгоритма Евклида, показанного на рисунке 10.3. Листинг 10.3 представляет собой реализацию алгоритма Евклида.

ЛИСТИНГ 10.3 L10-3.C

```
/* Находит и возвращает наибольший общий делитель двух целых чисел.
Использует алгоритм Евклида: делит большее целое число на
меньший; если остаток равен 0, меньшее целое число является GCD,
в противном случае меньшее целое число становится большим целым числом,
остаток становится меньшим целым
числом, и процесс
повторяется. */
```

```
>static unsigned int gcd_rekurs(unsigned int, unsigned int);
unsigned int gcd(unsigned int int1, unsigned int int2){
    unsigned int temp;
    /* If the two integers are the same, that's the GCD and we're
       done */
    if (int1 == int2) {
        return(int1);
    }
    /* Swap if necessary to make sure that int1 >= int2 */
    if (int1 < int2) {
        temp = int1;
        int1 = int2;
        int2 = temp;
    }
    /* Now call the recursive form of the function, which assumes
       that the first parameter is the larger of the two */
    return(gcd_rekurs(int1, int2)); static unsigned
int gcd_rekurs(unsigned int larger_int,
    unsigned int smaller_int)
int temp;
    /* If the remainder of larger_int divided by smaller_int is 0,
       тогда smaller_int - это gcd */
    if ((temp = larger_int % smaller_int) == 0) {
        return(smaller_int);
    }
    /* Сделайте smaller_int большим целым числом, а остаток -
       меньшим целым числом и вызовите эту функцию рекурсивно для
       продолжить процесс */
    вернуть (gcd_rekurs(smaller_int, temp));
```

<Как вы можете видеть из таблицы 10.1, алгоритм Евклида превосходит, особенно для больших чисел (а представьте, если бы мы работали с большими длинами!).

 Если бы я реализовывал определение НОД без помощи Седжвика, Я бы, конечно, не остановился на листинге 10.1, но вполне мог бы это сделать закончил перечислением 10.2 в моем энтузиазме по поводу “блестящего” открытия вычитания меньшего с использованием алгоритма Евклида для нахождения числа GCD из большего. В коммерческом продукте недостаток моего терпения и дисциплины мог действительно дорого обойтись.

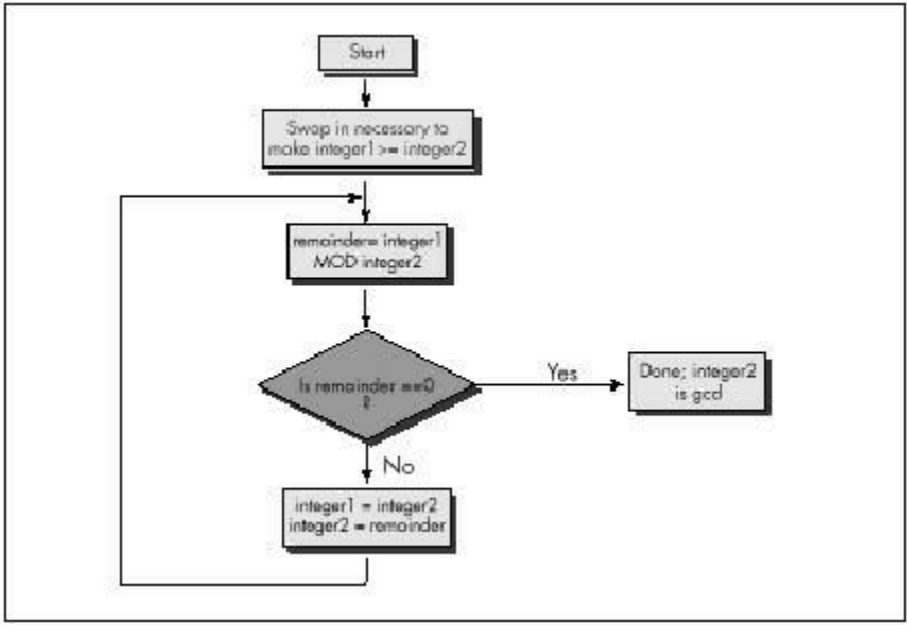


Рисунок 10.3 Использование алгоритма Евклида для нахождения НОД.

Дайте своему разуму время и пространство, чтобы побродить по краям важного программирования проблемы, прежде чем вы остановитесь на каком-либо одном подходе. Первую главу этой книги я озаглавил “ Лучший оптимизатор у вас между ушей”, и это по-прежнему верно; что еще более верно, так это то, что оптимизатор у вас между ушей лучше всего работает не на этапе внедрения, а в самом начале, когда вы пытаетесь представить, как лучше всего объединить то, что вы хотите сделать, и то, на что способен компьютер.

Рекурсия

Алгоритм Евклида прекрасно поддается рекурсии, настолько, что реализация подобная листингу 10.3, возникает практически без раздумий. Однако, опять же, остановитесь на минутку

и рассмотрим, что на самом деле происходит на уровне языка ассемблера, в листинге 10.3. Есть рекурсия, а затем есть рекурсия; рекурсия кода и рекурсия данных, если быть точным. В листинге 10.3 приведена рекурсия кода — рекурсия через вызовы — наиболее часто используемый вид поскольку он концептуально самый простой. Однако рекурсия кода имеет тенденцию быть медленной, потому что он вводит параметры и вызывает подпрограмму для каждой итерации. Листинг 10.4, в котором используется рекурсия данных, намного быстрее и не сложнее, чем в листинге 10.3. На самом деле, вы могли бы просто сказать, что в листинге 10.4 используется цикл, и игнорировать любое упоминание рекурсии; концептуально, однако, листинг 10.4 выполняет те же рекурсивные операции, что и листинг 10.3.

ЛИСТИНГ 10.4 L10-4.C

```
> /* Finds and returns the greatest common divisor of two integers.
   Uses Euclid's algorithm: divides the larger integer by the
   smaller; if the remainder is 0, the smaller integer is the GCD,
   otherwise the smaller integer becomes the larger integer, the
   remainder becomes the smaller integer, and the process is
   repeated. Avoids code recursion. */
unsigned int gcd(unsigned int int1, unsigned int int2) {
    unsigned int temp;
    /* Swap if necessary to make sure that int1 >= int2 */
    if (int1 < int2) {
        temp = int1;
        int1 = int2;
        int2 = temp;
    }
    /* Now loop, dividing int1 by int2 and checking the remainder,
       пока остаток не станет равным 0. На каждом шаге, если остаток не равен
       0, присваивайте int2 int1, а остаток int2, затем
       повторите */
    для (;;) {
        /* Если остаток от int1, деленный на int2, равен 0, то int2 равен
           НОД */
        if ((temp = int1 % int2) == 0) {
            return(int2);
        }
        /* Сделайте int2 большим целым числом, а остаток -
           меньшим целым числом и повторите процесс */
        int1 = int2;
        int2 = temp;
    }
}
```

<Оптимизация работы с пациентами

Наконец-то мы готовы оптимизировать определение НОД в классическом смысле. Таблица 10.1 показана производительность листинга 10.4 с максимальной оптимизацией Microsoft C / C ++ и без нее а также показана производительность листинга 10.5, языка ассемблера версия листинга 10.4. Несомненно, оптимизированные версии работают быстрее, чем неоптимизированные версия листинга 10.4 — но выигрыш невелик по сравнению с результатами, полученными от оптимизация более высокого уровня в списках с 10.2 по 10.4.

СПИСОК 10.5 L10-5.ASM

	dw	—	?	; добавленный BP
	dw	—	?	; добавленный обратный адрес
международный	dw		?	; целые числа, для которых нужно найти
int2	dw		?	; НОД
разделяет	заканчивается	маленькая		
	.модель			

```
>public_gcd
align 2
_gcd proc near
push bp;preserve caller's stack frame
movbp,sp;set up our stack frame
push si;preserve caller's register variables
push di
;Swap if necessary to make sure that int1 >= int2
mov ax,int1[bp]
mov bx,int2[bp]
cтpax,bx;is int1 >= int2?
jnb IntsSet;yes, so we're all set
xchg ax,bx;no, so swap int1 and int2
IntsSet:
; Now loop, dividing int1 by int2 and checking the remainder, until
; the remainder is 0. At each step, if the remainder isn't 0, assign
; int2 to int1, and the remainder to int2, then repeat.
GCDLoop:
;if the remainder of int1 divided by
; int2 is 0, then int2 is the gcd
subdx,dx;prepare int1 in DX:AX for division
divbx _;int1/int2; remainder is in DX
anddx,dx;is the remainder zero?
jz...Done;yes, so int2 (BX) is the gcd
;no, so move int2 to int1 and the
; remainder to int2, and repeat the
; process
movax,bx;int1 = int2;
movbx,dx;int2 = remainder from DIV
;-start of loop unrolling; the above is repeated three times-
subdx,dx;prepare int1 in DX:AX for division
divbx _;int1/int2; remainder is in DX
anddx,dx;is the remainder zero?
jz...Done;yes, so int2 (BX) is the gcd
movax,bx;int1 = int2;
movbx,dx;int2 = remainder from DIV
subdx,dx;prepare int1 in DX:AX for division
divbx _;int1/int2; remainder is in DX
anddx,dx;is the remainder zero?
jz...Done;yes, so int2 (BX) is the gcd
movax,bx;int1 = int2;
movbx,dx;int2 = remainder from DIV
subdx,dx;prepare int1 in DX:AX for division
divbx _;int1/int2; остаток указан в DX
idx,dx; остаток равен нулю?
jz _Выполнено; да, значит, int2 (BX) - это gcd
movax,bx;int1 = int2;
movbx,dx;int2 = остаток от DIV
;-конец разворачивания цикла-
jmpGCDLoop
align2
Готово:
перемещениеax,bx; верните GCD
всплывающееedi _; восстановить регистровые переменные вызывающего абонента
всплывающее окносi _
popbp.; восстановить фрейм стека вызывающего устройства
ret
gcd endp
завершить
```

<Оптимизация языка ассемблера заключается в подборе шаблонов в локальном масштабе. Честно говоря, это также такая скучная работа методом перебора, в которой люди никудашные; компиляторы могли бы превзойти вас оптимизировать вас на этом уровне за один проход, если бы они знали столько о коде, который вы пишете так, как делаете вы, чего не делают они.

 Оптимизация дизайна — концептуальные прорывы в понимании взаимосвязи между потребностями приложения и характером данных приложение работает, и то, что может сделать компьютер, носит глобальный характер сопоставление с образцом.

Компьютеры намного хуже справляются с подобными сопоставлениями, чем люди; компьютеры не имеют возможности интегрировать огромное количество разрозненной информации, большая ее часть носит весьма расплывчатый характер определена или может быть изменена. Люди, как ни странно, лучше справляются с глобальной оптимизацией чем при локальной оптимизации. С одной стороны, это интереснее. С другой, это сложно и достаточно неточно, чтобы использовать интуицию и вдохновение, два сильно недооцененных

на первый план выходят инструменты программирования. И, как я указывал ранее, люди склонны к мгновенному решению даже самых сложных проблем, в то время как компьютеры увязать в геометрически или экспоненциально увеличивающемся времени выполнения. О, это может занять дни или недели, пока человек усвоит достаточно информации, чтобы прийти к решению, и решение может быть лишь близким к оптимальному, но само решение (или, по крайней мере, каждая из частей решения) приходит в мгновение ока.

Эти вспышки - это ваш программный модуль сопоставления шаблонов, выполняющий свою работу. Ваша задача - предоставить вашему программисту сопоставления шаблонов возможность познакомиться с каждой проблемой и проработать ее дважды или трижды, с разных ракурсов, чтобы увидеть, какие неожиданные решения он может предложить .

Немного натяните поводья. Не измеряйте прогресс строками кода, написанными сегодня; вместо этого измеряйте его общим прогрессом и качеством. Расслабьтесь и прислушайтесь к тому тихому внутреннему голосу, который обеспечивает настоящие прорывы. Остановитесь, посмотрите, послушайте - и подумайте. Вы не только обнаружите, что это более продуктивный и творческий способ программирования, но и обнаружите, что это веселее.

И подумайте, что вы могли бы сделать со всеми этими дополнительными компьютерными годами!

Глава 11 – Переключение между 286 и 386-м номерами

Новые регистры, новые инструкции, новые тайминги, новые сложности

Эта глава, адаптированная из моей предыдущей книги "Дзен языка ассемблера" (1989; сейчас вышла из печати), содержит обзор процессоров 286 и 386, часто противопоставляя эти процессоры с 8088. На момент написания этой статьи 8088 был королем процессоров., а 286-й и 386-й были новенькими в квартале. Сегодня, конечно, все трое процессоры вышли за пределы своих возможностей, но многие миллионы процессоров по-прежнему используются, и 386-й в частности, его по-прежнему стоит учитывать при оптимизации программного обеспечения.

В этой главе представлен интересный взгляд на эволюцию архитектуры x86 в большей степени, чем вы могли ожидать, поскольку семейство x86 достигло полной зрелости с 386; 486 и Pentium на самом деле не что иное, как более быстрые 386s, с очень небольшим количеством новых функциональных возможностей. В отличие от этого, 286-я добавила ряд инструкций, достойную производительность и защищенный режим к возможностям 8088 и 386-й добавлено больше инструкций и совершенно новый набор режимов адресации, что привело семейство x86 в 32-разрядный мир, который представляет будущее (и, все чаще, настоящее) персональные компьютеры. В этой главе также рассказывается о влиянии на оптимизацию вариаций в архитектуре процессоров и памяти, которые распространены в мире ПК. Итак, хотя 286 и 386-е больше не являются основным направлением вычислительной техники, эта глава представляет собой полезное сочетание урока истории, обзора x86 и подробностей о двух рабочих лошадках процессоры, которые по-прежнему широко используются.

Семейство имеет значение.

Хотя семейство x86 является большим, лишь несколько членов семейства, в том числе в 8088, 8086, 80188, 80186, 286, 386SX, 386DX, многочисленные модификации 486, и теперь Pentium — действительно важно.

8088 сейчас практически исчезли с компьютерной арены. 8086 некоторое время использовался довольно широко некоторое время, но сейчас практически исчез. Модели 80186 и 80188 так и не прижились для использования на ПК и не требуют дальнейшего обсуждения.

Таким образом, у нас остаются чипы высокого класса: 286, 386SX, 386, 486 и Pentium. На момент написания этой статьи 386SX быстро идет по пути 8088; люди

понимают, что его относительно небольшого преимущества в стоимости по сравнению с 386 недостаточно, чтобы компенсировать его относительно большой недостаток производительности. В конце концов, 386SX страдает от того же изнуряющая проблема, которая нависла над 8088 — слишком маленькая шина. Внутренне 386SX - это 32-разрядный процессор, но внешне это 16-разрядный процессор с неоптимальной архитектурой, особенно для 32-разрядного кода.

Я не собираюсь подробно обсуждать 386SX. Если вы обнаружите, что программируете для 386SX, следуйте тем же общим правилам, которым следует следовать для 8088: используйте короткие инструкции, используйте регистры как можно активнее и не разветвляйтесь. Другими словами, избегайте использования памяти, поскольку 386SX по определению лучше справляется с внутренней обработкой данных, чем с доступом к памяти.

486-й - это отдельный мир для целей оптимизации, а Pentium - это отдельная вселенная. Мы рассмотрим их отдельно в последующих главах.

Таким образом, у нас остается всего два процессора: 286 и 386. Каждый из них был ПК стандартным в свое время. Модель 286 больше не используется в новых системах, но их миллионы Системы на базе 286 все еще используются ежедневно. Модель 386 все еще используется в новых системах., хотя срок его службы подходит к концу, и он используется даже шире, чем 286-й. Будущее явно принадлежит 486-му и Pentium, но 286-му и 386-му все еще очень значительная часть современного ландшафта.

Пересечение залива на 286 и 386-м моделях

Помимо значительно улучшенных характеристик, самая большая разница между 8088 и в 286 и 386 (а также более поздних процессорах Intel) в 286 представлен защищенный режим, а в 386 значительно расширены возможности защищенного режима. Однако мы всего лишь в этой книге поговорим о работе 286 и 386 в реальном режиме. Защищенный режим предлагает совершенно новую схему управления памятью, которой нет поддерживается 8088. В нем может выполняться только код, специально написанный для защищенного режима режим; это чужеродная и враждебная среда для программ MS-DOS.

В частности, сегменты - это разные существа в защищенном режиме. Они являются селекторами— индексы в таблицу дескрипторов сегментов — вместо обычных старых регистров, и они не могут быть установлены в произвольные значения. Это означает, что сегменты нельзя использовать для временного хранения или как часть быстрой неделимой 32-битной загрузки из памяти, как в

```
>les ax,dword ptr [LongVar]
mov dx,es
```

<которая загружает Longvar в DX: AX быстрее, чем это:

```
>mov ax,word ptr [LongVar]
mov dx,word ptr [LongVar+2]
```

<Защищенный режим использует эти измененные сегментные регистры для предоставления доступа к значительному объему памяти больше, чем в реальном режиме: 286 поддерживает 16 мегабайт памяти, в то время как 386 поддерживает 4 гигабайта (4К мегабайт) физической памяти и 64 терабайта (64К гигабайт!) виртуальной памяти.

В защищенном режиме ваши программы обычно запускаются под управлением операционной системы (OS / 2, Unix, Windows NT или аналогичной), которая обеспечивает гораздо больший контроль над компьютером, чем MS-DOS. Операционные системы с защищенным режимом обычно могут запускать несколько программ одновременно, и производительность любой одной программы может гораздо меньше зависеть от кода качество зависит не от того, насколько эффективно программа использует службы операционной системы и как часто и при каких обстоятельствах операционная система вытесняет программу. Программы защищенного режима часто представляют собой в основном наборы вызовов операционной системы, и производительность любого кода, не ориентированного на операционную систему, может зависеть в первую очередь от насколько большой временной отрезок операционная система выделяет этому коду для запуска.

Короче говоря, в целом программирование в защищенном режиме - это совсем другое дело в целом из того, что я описывал в этой книге. В оптимизации специально для защищенного режима в данной операционной системе определенно есть сноровка ... но это не так чему мы учились, и сейчас не время углубляться в это. В целом, тем не менее, стратегии оптимизации, обсуждаемые в этой книге, по-прежнему применимы в защищенных режим; это просто проблемы, характерные для защищенного режима или конкретной операционной системы, которые мы не будем обсуждать.

В логове пожирателей циклов, часть II

В разделе интерфейс программирования, модели 286 и 386 значительно отличаются от 8088. Тем не менее, за одним исключением и одним дополнением, пожиратели велосипедов остаются во многом такими же, то же самое на компьютерах, построенных на базе 286 и 386. Далее мы рассмотрим каждый из знакомых пожиратели циклов я рассмотрел в главе 4, поскольку они применимы к 286 и 386, и мы рассмотрим нового члена банды, пожирателя циклов выравнивания данных.

Единственный пожиратель циклов, который исчезает на 286 и 386, - это пожиратель циклов 8-разрядной шины. 286 - это 16-разрядный процессор как внутри, так и снаружи, а 386 - это 32-разрядный процессор как внутри, так и снаружи, таким образом, несоответствие размеров исполнительного устройства / шинного интерфейса устранена проблема 8088. Следовательно, больше нет необходимости использовать переменные памяти размером в байт предпочтительнее переменных размером в слово, по крайней мере, до тех пор, пока переменные размером в слово начинаются с четных адресов, как мы вскоре увидим. С другой стороны, доступ к переменным размером в байт по-прежнему осуществляется ничуть не медленнее, чем доступ к переменным размером в слово, поэтому вы можете использовать тот размер, который лучше всего подходит для данной задачи.

Вы могли бы подумать, что устранение 8-битного пожирателя циклов шины означало бы, что пожиратель циклов очереди предварительной выборки также исчезнет, поскольку на 8088 очередь предварительной выборки пожиратель циклов является побочным эффектом 8-разрядной шины. Это казалось бы тем более вероятным, учитывая что и 286, и 386 имеют большие очереди предварительной выборки, чем 8088 (6 байт для 286, 16 байт для 386) и может выполнять доступ к памяти, включая команды выборка за гораздо меньшее количество циклов, чем у 8088.

Однако пожиратель циклов очереди предварительной выборки не исчезает ни на 286, ни на 386 по нескольким причинам. С одной стороны, инструкции ветвления по-прежнему опустошают очередь предварительной выборки, поэтому выборка инструкций по-прежнему замедляет работу после большинства ветвлений; когда предварительная выборка очередь пуста, не имеет большого значения, насколько она велика. (Даже если не считать опорожнения очередь предварительной выборки, ответвления не особенно быстрые на 286 или 386, как минимум по семь с лишним циклов на штуку. По возможности избегайте ответвлений.)

После отделения действительно имеет значение, насколько быстро может пополниться очередь, и вот тут мы подходим к вторая причина, по которой пожиратель циклов очереди предварительной выборки продолжает жить: 286 и 386 настолько быстры иногда исполняющее устройство может выполнять инструкции быстрее, чем они могут быть извлекается, даже несмотря на то, что выборка инструкций происходит намного быстрее на 286 и 386, чем на 8088.

(При прочих равных условиях слишком медленная выборка инструкций является большей проблемой на 286 больше, чем на 386, поскольку 386 извлекает 4 байта инструкций за раз по сравнению с 2 байты инструкций, извлекаемые на доступ к памяти 286. Однако 386 также обычно работает как минимум в два раза быстрее, чем 286, что означает, что 386 может легко выполняться инструкции быстрее, чем они могут быть извлечены, если только не используется очень высокоскоростная память.)

Наиболее существенная причина, по которой пожиратель циклов очереди предварительной выборки не только выживает, но и однако преимущество 286 и 386 заключается в различных архитектурах памяти, используемых в компьютерах, построенных на основе 286 и 386. Из-за архитектуры памяти 8-разрядная шина пожиратель циклов заменен новой формой пожирателя циклов состояния ожидания: состояния ожидания включены обращается к обычной системной памяти.

286 и 386 были разработаны таким образом, чтобы обеспечить относительно небольшую потерю производительности при предварительной выборке пожиратель циклов очереди...при использовании с памятью с нулевым ожиданием: память, которая может завершать работу доступ к памяти осуществляется так быстро, что состояния ожидания не требуются. Однако истинное нулевое ожидание- память состояния почти никогда не используется в этих процессорах. Почему? Потому что память, которая может работать на уровне 286, довольно дорогая, а память, которая может работать на уровне 386, - это очень дорого. Вместо этого разработчики компьютеров используют альтернативные архитектуры памяти, которые предлагают большую производительность за доллар, но меньшую производительность в целом, чем память с нулевым ожиданием состояние. (Можно построить системы с нулевым состоянием ожидания для 286 и 386; это просто настолько дорого, что это редко делается.)

Совместимые IBM AT и true используют память с одним состоянием ожидания (некоторые клоны AT используют память с нулевым состоянием ожидания, но такие клоны встречаются реже, чем с одним состоянием ожидания В клоны). Системы 386 используют широкий спектр систем памяти, включая высокоскоростную кэши, чередующуюся память и оперативную память со статическими столбцами, которые вставляют значения от 0 до около 5 состояний ожидания (и намного больше, если используются 8 или 16-разрядные карты расширения памяти); точное количество состояний ожидания, вставленных в любой момент времени, зависит от взаимодействия между выполняемым кодом и системой памяти, в которой он запущен.



Производительность большинства систем памяти 386 может сильно различаться в зависимости от одной доступ к памяти другому, в зависимости от таких факторов, как то, какие данные находятся в кэше и к какому чередующемуся банку данных и / или оперативной памяти к столбцу обращались последним.

Множество используемых систем памяти не позволяют нам оптимизировать работу для 286/386 компьютеры с точностью, возможной для 8088. Вместо этого мы должны писать код это работает достаточно хорошо при различных условиях, обнаруженных на арене 286/386.

Состояния ожидания, которые возникают при большинстве обращений к системной памяти в 286 и 386 компьютеры означают, что почти каждый доступ к системной памяти — памяти в DOS обычная область памяти 640 КБ — замедляется. (Доступ осуществляется на компьютерах с высокой скоростью кэши могут быть свободны от состояния ожидания, если нужные данные уже находятся в кэше, но обязательно возникнут состояния ожидания, если данные не кэшированы; это явление приводит к время выполнения команд сильно варьируется.) Хотя это наше первое столкновение с системой состояния ожидания памяти, мы уже сталкивались с пожирателем циклов в состоянии ожидания раньше: дисплей пожиратель циклов адаптера, который мы обсуждали вместе с другими способами пожирателей циклов 8088 еще в главе 4. Системная память обычно имеет меньше состояний ожидания для каждого доступа, чем память дисплея. Однако к системной памяти также обращаются гораздо чаще, чем к дисплею память, поэтому состояния ожидания системной памяти причиняют много вреда — и больше всего они вредят выборке инструкций.

Учтите это: 286 может сохранять немедленное значение в памяти, как в MOV [WordVar], 0, всего за 3 цикла. Однако, длина этой инструкции составляет 6 байт. 286 способен выполнять выборку 1 слово каждые 2 цикла; однако архитектура AT с одним состоянием ожидания растягивает это до 3 циклов. Следовательно, требуется девять циклов для извлечения шести байтов инструкций. Далее кроме того, для записи в память требуется 3 цикла, что обеспечивает общий доступ к памяти время до 12 циклов. В целом, время доступа к памяти — особенно предварительная выборка команд — значительно превышает время выполнения, в той степени, в какой может потребоваться данная конкретная команда выполнение в четыре раза дольше, чем выполнение в исполнительном модуле.

И это, мой друг, безошибочно является пожирателем циклов очереди предварительной выборки. Я мог бы добавить, что пожиратель циклов очереди предварительной выборки находится в редкой хорошей форме в приведенном выше примере: соотношение 4 к 1 время выборки инструкций до времени выполнения находится в классе с лучшим (или худшим!) из всех найденных на 8088.

Давайте проверим пожиратель циклов очереди предварительной выборки в действии. Листинг 11,1
10 раз MOV [WordVar], 0. Таймер Zen сообщает, что в режиме ожидания на частоте 10 МГц 286 на

clone (компьютер, используемый для всех тестов в этой главе), листинг 11.1 выполняется за 1,27 мкс на команда. Это 12,7 циклов на команду, как мы и рассчитывали. (Эти дополнительные семь десятых цикла приходится на обновление DRAM, к которому мы вскоре перейдем.)

ЛИСТИНГ 11.1 L11-1.ASM

```
>; *** Listing 11.1 ***
; Measures the performance of an immediate move to
; memory, in order to demonstrate that the prefetch
; queue cycle-eater is alive and well on the AT.
    jmpSkip
    even;always make sure word-sized memory
        ; variables are word-aligned!

WordVar dw0
;
Skip:
    callZTimerOn
    rept 1000
    mov[WordVar],0
    endm
    callZTimerOff
```

<Что это значит? Это означает, что, практически говоря, 286, используемый в АТ, не имеет 16-разрядной шины. С точки зрения производительности, 286 в АТ имеет двухразрядную трети 16-разрядной шины (10,7-разрядной шины?), поскольку доступ к каждой шине на АТ занимает на 50 процентов больше времени, чем следовало бы. 286, работающий на частоте 10 МГц, должен иметь доступ к памяти с максимальной частотой 1 слово каждые 200 нс; однако в 10 МГц АТ эта частота составляет сокращается до 1 слова каждые 300 нс за счет памяти с одним состоянием ожидания.

Короче говоря, близкий родственник нашего старого друга 8-битного пожирателя циклов шины - системной памяти пожиратель циклов состояния ожидания по-прежнему преследует нас на всех компьютерах 286 и 386 с нулевым состоянием ожидания, кроме, и это означает, что пожиратель циклов очереди предварительной выборки жив и здоров. (Система пожиратель циклов состояния ожидания памяти на самом деле не является новым пожирателем циклов, а скорее вариантом пожиратель циклов общего состояния ожидания, еще одним пожирателем циклов которого является адаптер дисплея вариант.) Хотя 286 в АТ может получать инструкции намного быстрее, чем 8088 в ПК, он может выполнять эти инструкции еще быстрее.

В мире 386 картина менее ясна, поскольку существует так много различных систем памяти архитектуры, но аналогичные проблемы могут возникнуть на любом компьютере, построенном на базе 286 или 386. Потребление цикла очереди предварительной выборки является даже фактором, хотя и меньшим, для конечных автоматов с нулевым ожиданием как потому, что ветвление опустошает очередь, так и потому, что некоторые инструкции могут опережать даже на ноль—на 5 циклов больше официального времени выполнения.)

Подводя итог, можно сказать, что:

- Инструкции с доступом к памяти не выполняются со своей официальной скоростью при ненулевом ожидании состояние 286/386 компьютеров.
- Пожиратель циклов очереди предварительной выборки снижает производительность компьютеров 286/386, особенно когда используется память с ненулевым состоянием ожидания.
- Ветви часто выполняются со скоростью ниже их номинальной на 286 и 386, поскольку очередь предварительной выборки опустошена.
- Степень влияния очереди предварительной выборки и состояний ожидания на производительность варьируется от одного компьютера 286/386 к другому, что делает точную оптимизацию невозможной.

Какой урок можно извлечь из всего этого? Несколько вещей:

- Делайте свои инструкции короткими.
- Сохраняйте их в регистрах; избегайте памяти, поскольку память обычно не успевает за процессором.
- Не прыгайте.

Конечно, именно эти правила применимы и к оптимизации 8088. Разве это не

Согласование данных

Благодаря 16-разрядной шине 286 может обращаться к переменным памяти размером со слово так же быстро, как и к переменным размером в байт. Однако есть загвоздка: это верно только для переменных размером со слово, начинающиеся с четных адресов. Когда 286 запрашивается выполнить команду размером со слово, начинающийся с нечетного адреса, фактически выполняет два отдельных доступа, каждый из которых извлекает 1 байт, точно так же, как 8088 делает для всех обращений размером со слово.

Рисунок 11.1 иллюстрирует это явление. Преобразование обращений размером в слово в нечетные адреса в обращения размером в два байта прозрачны для доступа к памяти инструкции; все, что известно любой инструкции, это то, что к запрошенному слову был получен доступ, независимо от не имеет значения, требовался ли доступ размером в 1 слово или 2 байта для его выполнения.

Штраф за выполнение доступа размером со слово, начинающегося с нечетного адреса, легко подсчитать. Два доступа занимают в два раза больше времени, чем один доступ.



Другими словами, эффективная пропускная способность внешней шины данных 286 уменьшается вдвое, когда выполняется доступ размером со слово к нечетному адресу.

Это, в двух словах, пожиратель циклов выравнивания данных, единственный новый пожиратель циклов из 286 и 386. (Пожиратель циклов выравнивания данных является близким родственником 8-разрядной шины 8088 — пожиратель циклов, но поскольку он ведет себя по-другому — встречается только по нечетным адресам — и его можно избежать с помощью другого обходного пути, мы будем считать его новым пожирателем циклов.)

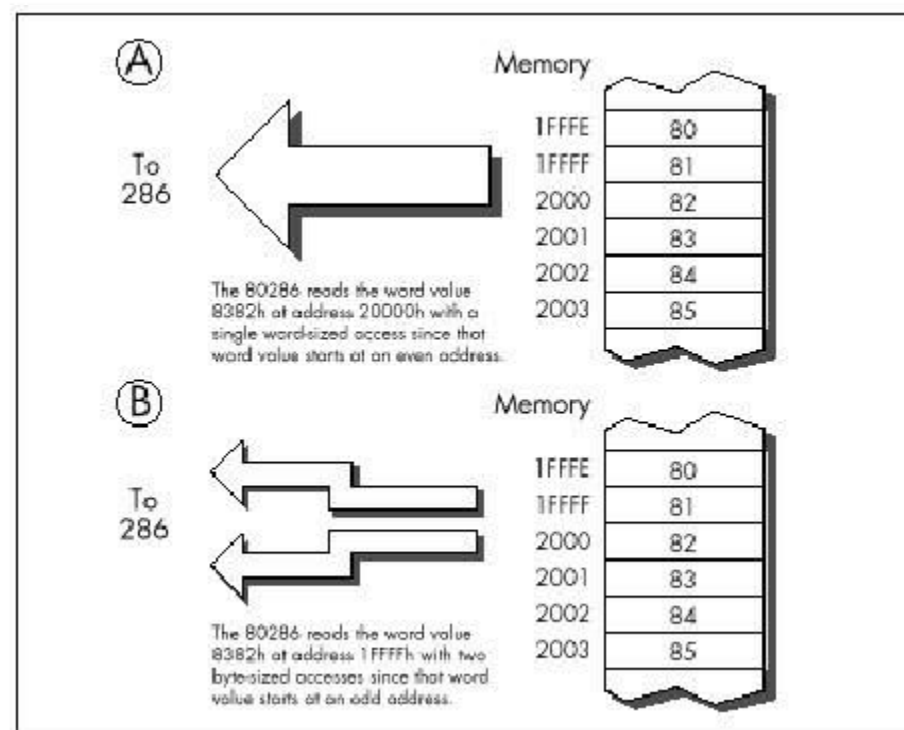


Рисунок 11.1 Пожиратель циклов выравнивания данных.

Способ справиться с пожирателем циклов выравнивания данных прост: не выполнять доступ размером со слово к нечетным адресам на 286, если вы можете с этим справиться. Самый простой способ избежать цикла выравнивания данных - проще всего поместить директиву ДАЖЕ перед каждой из ваших переменных размером со слово. EVEN делает смещение следующего собранного байта четным на вставку NOP, если текущее смещение нечетное; следовательно, вы можете гарантировать, что любая переменная размером со слово может быть эффективно доступна 286, просто указав перед ней ЧЕТНУЮ.

Листинг 11.2, который обращается к памяти по слову за раз, причем каждое слово начинается с нечетной адрес, выполняется на частоте 10 МГц при клонировании за 1,27 мс на повторение movsw, или 0,64 мс на доступ к памяти размером со слово. Это составляет более 6 циклов на доступ размером со слово, который разбивается на два отдельных доступа к памяти — 3 цикла для доступа к старшему байту каждого слова и 3 цикла для доступа к младшему байту каждого слова, неизбежный результат обращения к памяти без слов. Доступ к памяти с выравниванием по размеру слова - плюс немного больше для обновления DRAM.

ЛИСТИНГ 11.2 L11-2.ASM

```
>*** Listing 11.2 ***
; Measures the performance of accesses to word-sized
; variables that start at odd addresses (are not
; word-aligned).
Skip:
    pushds...
    popes _
    movsi,1 ;source and destination are the same
    movdi,si ; and both are not word-aligned
    movex,1000 ;move 1000 words
    cld
    callZTimerOn
    repMOVSW
    callZTimerOff
```

<С другой стороны, листинг 11.3, который в точности совпадает с листингом 11.2, за исключением того, что обращения к памяти выполняются с выравниванием по словам (начинаются с четных адресов), выполняется за 0,64 мс на повторение movsw или 0,32 мкс на обращение к памяти размером со слово. Это составляет 3 цикла на одно слово доступ размером с слово — ровно в два раза быстрее, чем доступ без выравнивания по словам в листинге 11.2, как мы и предсказывали.

ЛИСТИНГ 11.3 L11-3.ASM

```
>; *** Listing 11.3 ***
; Measures the performance of accesses to word-sized
; variables that start at even addresses (are word-aligned).
;
Skip:
    pushds...
    popes _
    subsi,si ;source and destination are the same
    movdi,si ; and both are word-aligned
    movcx,1000 ;move 1000 words
    cld
    call ZTimerOn
    repMOVSW
    callZTimerOff
```

<Пожираатель циклов выравнивания данных имеет интригующие последствия для ускорения 286/386 код. Небольшая осторожность и несколько байтов для обеспечения соответствия размера слова выравнивание переменных и блоков памяти по словам может буквально удвоить производительность определенный код, запущенный на 286. Даже если это не удваивает производительность, выравнивание слов обычно помогает и никогда не вредит.

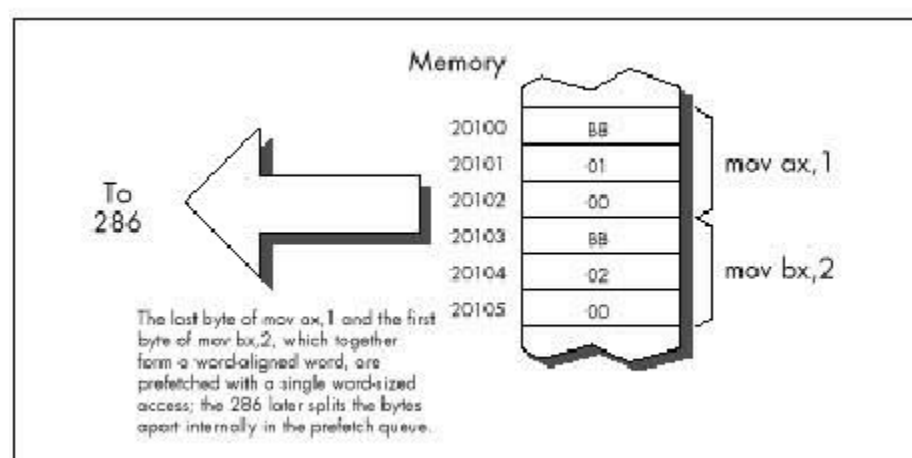
Выравнивание кода

Отсутствие выравнивания слов также может мешать выборке команд на 286, хотя не до такой степени, чтобы это мешало доступу к переменным памяти размером со слово. 286 выполняет предварительную выборку инструкций по слову за раз; даже если данная инструкция не начинается с четный адрес, 286 просто одновременно выбирает первый байт этой инструкции что он извлекает последний байт предыдущей инструкции, как показано на рисунке 11.2, затем разделяет байты внутри. Это означает, что в большинстве случаев инструкции выполняются так же, как быстро, независимо от того, выровнены они по словам или нет.

Однако существует штраф за выравнивание без слов для ответвлений на нечетные адреса. На ответвление на нечетный адрес, 286 способен извлекать только 1 полезный байт с помощью первой команды выборки, следующей за ветвью, как показано на рисунке 11.3. Другими словами, отсутствие выравнивание по словам целевой инструкции для любой ветви эффективно сокращает количество команд мощность выборки 286 вдвое для первой выборки команды после этой ветви. В то время как это может показаться не таким уж большим, вы были бы удивлены тем, что это может сделать с узкими циклами; на самом деле, краткий рассказ уместен.

Когда я разрабатывал Zen timer, я использовал свой надежный AT-клон на базе 286 AT с частотой 10 МГц, чтобы проверить базовую функциональность таймера путем измерения производительности простых командных последовательностей . Я ехал без каких-либо проблем, пока не рассчитал время на следующее код:

```
>movcx,1000
call ZTimerOn
LoopTop:
loop LoopTop
call ZTimerOff
```



<Рисунок 11.2 Предварительная выборка с выравниванием по словам на 286.

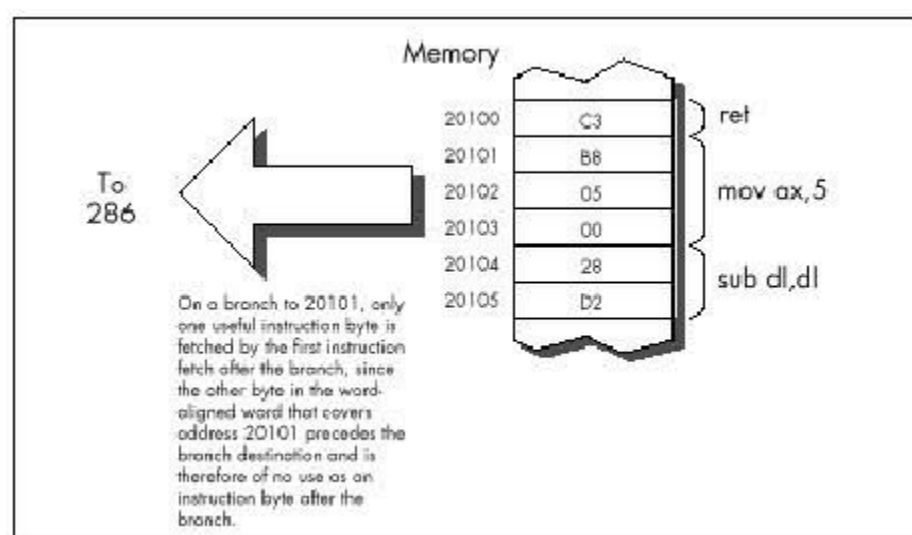


Рисунок 11.3, как байты инструкций извлекаются после перехода.

Теперь этот код должен выполняться, скажем, максимум за 12 циклов на цикл. Вместо этого это заняло больше времени 14 циклов на цикл, время выполнения, которое я никак не могу объяснить. Прокрутив это в голове некоторое время, я взглянул на код в отладчике ... и ответ сам выскочил у меня. Цикл начался с нечетного адреса! Это означало, что каждый раз в цикле требовались две выборки инструкций; одна для получения байта кода операции инструкции ЦИКЛА, которая находилась в конце одного слова, выровненного по слову, и другого чтобы получить байт смещения, который находился в начале следующего слова, выровненного по слову.

Одно простое изменение сократило время выполнения до разумных 12,5 циклов на цикл:

```
>movcx,1000
call ZTimerOn
even
LoopTop:
loop LoopTop
call ZTimerOff
```

<Хотя назначение ветвей с выравниванием по словам может повысить производительность ветвления, это это неудобно и может значительно увеличить размер кода, поэтому в большинстве случаев этого делать не стоит. Кроме того, ЭВЕН при необходимости вставляет NOP-инструкцию, а время, необходимое для выполнения NOP иногда может свести на нет преимущество в производительности от наличия ветви, выровненной по слову, пункт назначения.



Следовательно, лучше всего выравнивать по словам только те пункты назначения ответвлений, до которых можно добраться исключительно путем разветвления.

Я рекомендую вам только выравнивать по слову начальные смещения ваших подпрограмм, как в:

```
even FindChar
proc рядом
```

По моему опыту, эта простая практика является единственной формой последовательного выравнивания кода обеспечивает разумную отдачу за затраченные байты и усилия, хотя иногда это также окупается

Выравнивание и 386-й

До сих пор мы обсуждали выравнивание только в том, что касается 286-го. Что, спросите вы, из 386-го?

386 добавляет проблему выравнивания по двойным словам (то есть выравнивания по адресам, которые кратны четырем). Правило для 386 таково: обращения к памяти размером со слово должны быть выровнены по словам (при обращении к памяти размером со слово невозможно пересекать двойные слова границы), и обращения к памяти размером с двойное слово должны быть выровнены по двум словам. Однако в реальном (в отличие от 32-разрядного защищенного) режиме память размером с двойное слово обращения к ним редки, поэтому простое правило выравнивания слов, которое мы разработали для 286 подходит и для 386 в реальном режиме.

Что касается выравнивания кода ... правило выравнивания слов в начале подпрограммы из 286 служит здесь тоже достаточно хорошо, поскольку оно позволяет избежать наихудшего случая, когда извлекается всего 1 байт вход в подпрограмму. В то время как оптимальная производительность диктует выравнивание по двум словам для подпрограмм требуется 3 байта, что является высокой ценой за оптимизацию, которая улучшает производительность только на процессорах post 286.

Выравнивание и стек

Одним из побочных эффектов пожирателя циклов выравнивания данных в 286 и 386 является то, что вам следует никогда не допускать, чтобы указатель стека становился нечетным. (Вы можете сделать указатель стека нечетным с помощью добавляя к нему нечетное значение или вычитая из него нечетное значение, или загружая в него нечетное значение.) Нечетный указатель стека в 286 или 386 (или стек, не выровненный по двойным словам в 32-разрядный защищенный режим на 386, 486 или Pentium) значительно снизит выполнение PUSH, POP, CALL и RET, а также INT и IRET, которые выполняются для вызывайте функции DOS и BIOS, обрабатывайте нажатия клавиш и входящие серийные символы, а также управляйте мышью. Я знаю программиста Forth, который значительно улучшил производительность из сложного приложения на AT просто заставляя интерпретатор Forth поддерживать четный указатель стека в любое время.

Интересным следствием этого правила является то, что вы не должны увеличивать SP дважды, чтобы добавить 2, даже хотя это занимает меньше байт, чем добавление SP, 2. Указатель стека нечетный между первой и второй инструкциями INC, поэтому любое прерывание, возникающее между двумя инструкциями, будет обслуживаться медленнее, чем обычно. То же самое касается двойного уменьшения; используйте Вместо этого SUB SP, 2 .



Постоянно выравнивайте указатель стека.

Пожиратель циклов обновления DRAM: по-прежнему стихийное бедствие

Пожиратель циклов обновления DRAM - это пожиратель циклов, который меньше всего изменился по сравнению с формой 8088 на 286 и 386. В AT для обновления DRAM используется чуть более пяти процентов всех доступных обращений к памяти, что немного меньше, чем на ПК, но в тех же пределах приблизительный показатель. В то время как ограничение на обновление DRAM несколько различается для разных AT-клонов и 386 компьютеров (фактически, несколько компьютеров построены на статической оперативной памяти, которая не требует обновлять вообще; аналогично, кэши состоят из статической оперативной памяти, поэтому кэшированные системы обычно меньше страдают от обновления DRAM), показатель в 5 процентов является хорошим эмпирическим правилом.

В принципе, эффект пожирателя циклов обновления DRAM практически одинаков во всем мире, совместимом с ПК: он довольно мал, поэтому не сильно влияет на производительность; неизбежный, так что в любом случае нет смысла беспокоиться об этом; и неприятный, поскольку это приводит к дробному подсчету циклов при использовании таймера Zen. Как и в случае с ПК, заданная последовательность кода на AT может выполняться с разной скоростью в разное время в результате

Таким образом, в обновлении DRAM на компьютерах 286/386 нет ничего особенного. Имейте в виду это, но не стоит слишком беспокоиться — обновление DRAM по-прежнему является деянием Божьим, и с этим ничего не поделать с этим ничего нельзя. К счастью, внутренние кэши 486 и Pentium благодаря тому, что обновление DRAM в значительной степени не влияет на производительность этих процессоров.

Потребляющий цикл адаптер дисплея

Наконец, мы подходим к последнему из пожирателей циклов - пожирателю циклов адаптера дисплея. Вот есть два способа взглянуть на этот пожиратель циклов на компьютерах 286/386: (1) Это намного хуже чем это было на ПК, или (2) это примерно то же самое, что и на ПК.

В любом случае, пожиратель циклов адаптера дисплея - крайне плохая новость для 286/386 компьютеры, а также для 486-х и Pentium. Фактически, это потребление циклов в этих системах в значительной степени отвечает за популярность местной шины VESA (VLB).

Два способа взглянуть на потребляющий цикл адаптер дисплея на компьютерах 286/386 фактически одинаковы. Как вы помните из моего предыдущего обсуждения этого вопроса в главе 4, адаптеры отображения предлагают только ограниченное количество обращений к памяти дисплея в течение любого заданного периода времени. 8088 способен использовать большинство, но не все из них слоты с REP MOVSW, поэтому количество обращений к памяти, разрешенное дисплейным адаптером, ограничено например, стандартный VGA достаточно хорошо совместим с доступом к памяти 8088 Скорость. Конечно, доступ к VGA значительно замедляет работу 8088, но, поскольку мы сейчас выяснится, что “значительно” - понятие относительное. То, что VGA делает с производительностью ПК ничто по сравнению с тем, что он делает с более быстрыми компьютерами.

В идеальных условиях 286 может обращаться к памяти намного быстрее, чем 8088. А 286 с частотой 10 МГц способен обращаться к слову системной памяти каждые 0,20 мс с помощью ПРЕДСТАВИТЕЛЬ MOVSW, уменьшающий на 1 байт каждые 1,31 мкс то, чем может управлять 8088 на ПК. Однако доступ к памяти дисплея далеко не идеален для 286. Во-первых, большинство адаптеров дисплея являются 8-разрядными устройствами, хотя более новые адаптеры по своей природе 16-разрядные. Одним следствием этого является то, что только 1 байт может быть прочитан или записан за один доступ к дисплею доступ к памяти размером с слово к 8-разрядным устройствам автоматически разделяется на 2 отдельных доступы размером в байт по шине АТ. Другим следствием является то, что доступы просто медленнее; на автобус вставки дополнительного ожидания для доступа к 8-разрядным устройствам с ней надо полагать, что такие устройства были разработаны, для ПК и не могут надежно работать на скорость.

Однако 8-битный размер большинства дисплейных адаптеров является лишь одним из двух факторов, снижающих скорость, с которой 286 может обращаться к памяти дисплея. Гораздо больше циклов потребляет ограничения доступа к памяти, присущие адаптерам дисплея, то есть ограниченное количество доступов к памяти дисплея, которые адаптеры дисплея предоставляют 286. Посмотрите на это таким образом: если представитель MOVSW на ПК может использовать более половины всех доступных обращений к памяти дисплея, тогда насколько быстрее может выполняться код, запущенный на 286 или 386, при доступе к памяти дисплея?

Все верно — менее чем в два раза быстрее.

Другими словами, инструкции, которые обращаются к памяти дисплея, не будут выполняться намного быстрее на ATS и более быстрых компьютерах, чем на ПК. Это объясняет одну из двух точек зрения, высказанных в начале этого раздела: потребляющий цикл адаптер дисплея составляет примерно то же самое на компьютерах высокого класса, что и на ПК, в том смысле, что это позволяет командам, обращающимся к памяти дисплея, выполняться примерно с одинаковой скоростью на всех компьютерах.

Конечно, картина несколько иная, если сравнить производительность инструкций, обращающихся к памяти дисплея, с максимальной производительностью этих

Инструкции. Инструкции, которые обращаются к памяти дисплея, получают гораздо больше состояний ожидания при запуске на 286, чем на 8088. Почему? Хотя 286 способен на доступ к памяти намного чаще, чем у 8088, мы видели, что частота доступ к памяти дисплея определяется не скоростью процессора, а дисплеем сам адаптер. В результате обоим процессорам фактически разрешено примерно одинаковое максимальное количество обращений к отображаемой памяти в любой момент времени. Тогда, по определению, 286-й должен проводить гораздо больше циклов ожидания, чем 8088-й.

И это объясняет вторую точку зрения, высказанную выше относительно адаптера дисплея потребляющий цикл по сравнению с 286 и 386. Потребляющий цикл адаптер дисплея, измеренный в циклы, потерянные из-за состояний ожидания, действительно намного хуже на компьютерах АТ-класса, чем на РС, и это еще хуже на более мощных компьютерах.



Насколько плох пожиратель циклов адаптера дисплея на АТ? Это настолько плохо: основано исходя из моего (немалого) опыта в выборе времени доступа к адаптеру дисплея, Я обнаружил, что пожиратель циклов адаптера дисплея может снизить скорость компьютера АТ или даже 386 до скорости, близкой к скорости ПК, при доступе к памяти дисплея.

Я знаю, в это трудно поверить, но потребляющий цикл адаптер дисплея выдает ровно столько обращений к памяти дисплея за заданное время, и не более, независимо от того, насколько быстро процессор есть. Фактически, чем быстрее процессор, тем больше потребляет времени адаптер дисплея снижает производительность команд, обращающихся к памяти дисплея. Адаптер дисплея cycle-eater не только по-прежнему присутствует в компьютерах 286/386, но и стал хуже, чем когда-либо.

Что мы можем сделать с этой новой, более опасной формой пожирателя циклов дисплейных адаптеров? Решение такое же, как и на ПК: используйте память дисплея как можно реже. Насколько это возможно.

Новые инструкции и функции: 286

Модели 286 и 386 предлагают ряд новых инструкций. Модель 286 имеет относительно небольшое количество количество инструкций, которых нет в 8088, в то время как у 386 есть эти инструкции и еще немало, наряду с новыми режимами адресации и размерами данных. В связи с этим мы обсудим 286 и 386 отдельно.

В 286-м есть ряд инструкций, разработанных для работы в защищенном режиме. Как я уже сказал, в этой книге мы не собираемся обсуждать защищенный режим; в любом случае, защищенный... инструкции по режиму обычно используются только операционными системами. (Я должен упомянуть, что защищенный режим 286 обеспечивает возможность обращения к 16 МБ памяти, что значительное улучшение по сравнению с 1 МБ у 8088. Однако в реальном режиме программы по-прежнему ограничены 1 МБ адресуемой памяти на 286. В любом режиме каждый сегмент по-прежнему ограничен 64 КБ.)

Существует также несколько инструкций для реального режима, специфичных для 286, и они могут быть весьма полезны. BOUND Проверяет границы массива. ENTER и LEAVE поддерживают компактный и быстрый стек построение и удаление фрейма, идеально подходит для взаимодействия с языками высокого уровня, такими как C и Pascal (хотя на самом деле эти инструкции выполняются относительно медленно на 386 и его преемники, и их следует использовать с осторожностью, когда важна производительность). ВХОДЫ И ВЫХОДЫ это новые строковые инструкции, которые поддерживают эффективную передачу данных между памятью и вводом-выводом порты. Наконец, PUSHA и POPA нажимают и извлекают все восемь регистров общего назначения.

Пара старых инструкций получают новые функции на 286. Во-первых, версия 286 Функция PUSH способна перемещать константу в стек. С другой стороны, 286-й допускает все сдвиги и выполняет вращение не только для 1 бита или количества битов, указанного CL, но для любого постоянного количества битов.

Новые инструкции и функции: 386

386 несколько сложнее, чем 286, в отношении новых функций. Еще раз, мы не будем обсуждать защищенный режим, который на 386 поставляется с возможностью обращения вверх до 4 гигабайт на сегмент и 64 терабайт в целом. В реальном режиме (а в виртуальном - 86, который позволяет 386 выполнять многозадачные приложения MS-DOS и который идентичен реальному режим, насколько это касается программ MS-DOS), программы, работающие на 386, являются по-прежнему ограничен 1 МБ адресуемой памяти и 64 КБ на сегмент.

386 имеет множество новых инструкций, а также новые регистры, режимы адресации и размеры данных, которые уменьшились из защищенного режима. Давайте быстро взглянем на них новые функции реального режима.

Даже в реальном режиме возможен доступ ко многим новым и расширенным регистрам 386. Большинство этих регистров являются просто 32-разрядными расширениями 16-разрядных регистров 8088. Например, EAX - это 32-разрядный регистр, содержащий AX в качестве младших 16 бит, EBX - это 32-разрядный регистр, содержащий BX в качестве младших 16 битов, и так далее. Также есть два новых сегментные регистры: FS и GS.

386 также поставляется с множеством новых инструкций для реального режима, помимо тех, которые поддерживаются 8088 и 286. Эти инструкции могут сканировать данные побитово, устанавливать Флаг Переноса на значение указанного бита, расширять данные со знаком или обнулять их по мере перемещения, установите для регистра или переменной памяти значение 1 или 0 на основе любого из возможных условий можно протестировать с помощью условных переходов и многого другого. (Опять же, будьте осторожны: многие из этих сложных Инструкции для 386 работают медленнее, чем эквивалентные последовательности простых инструкций на 486 и особенно Pentium.) Более того, как старые, так и новые инструкции поддержка 32-разрядных операций на 386. Например, относительно просто копировать данные в фрагменты по 4 байта на 386-м диске, даже в реальном режиме, с помощью MOVSD ("переместить строку команда double") или для отрицания 32-битного значения с помощью NEG eax.

Наконец, в реальном режиме можно использовать новые режимы адресации 386, в которых любой для адресации памяти может использоваться 32-разрядный регистр общего назначения или пара регистров. Более того, умножение регистров адресации памяти на 2, 4 или 8 для поиска в таблицы word, double word или quadword могут быть встроены прямо в адресацию памяти режим. (32-разрядные режимы адресации более подробно обсуждаются в последующих главах.) В защищенный режим, эти новые режимы адресации позволяют адресовать полные 4 гигабайта на сегмент, но в реальном режиме вы по-прежнему ограничены 64 КБ, даже с 32-разрядными регистрами и новые режимы адресации, если только вы не используете какие-то неортодоксальные трюки с сегментом регистры.



Хорошо обратите внимание: эти трюки не обязательно работают с системным программным обеспечением, таким как Windows, поэтому я бы не рекомендовал их использовать. Если вам нужны 4- гигабайтные сегменты, используйте 32-разрядную среду, такую как Win32.

Правила оптимизации: Чем больше все меняется...

Давайте посмотрим, что мы узнали об оптимизации 286/386. В основном мы узнали следующее что наши знакомые пожиратели компьютерных циклов все еще применяются, хотя и в несколько иных формах, и что основные правила оптимизации для ПК справедливы для компьютеров на базе ATs и 386. Вы не ошибетесь ни на одном из этих компьютеров, если будете придерживаться краткости инструкций, использовать интенсивно регистры и экономить память, не переходить на другие разделы и избегать доступа к дисплею память как чума.

Хотя мы их не затрагивали, повторяющиеся строковые инструкции по-прежнему желательны на 286 и 386, поскольку они обеспечивают большую функциональность на байт инструкции и исключите как пожиратель циклов очереди предварительной выборки, так и ветвление. Однако string инструкции на 286 и 386 не так впечатляюще превосходны, как на инструкции доступа к памяти, не являющиеся строками, были ускорены с 8088 года.

Есть один пожиратель циклов с новыми возможностями для моделей 286 и 386, и это данные пожиратель циклов выравнивания. Из программы обработки циклов выравнивания данных мы получаем новое правило: Word- выровняйте переменные размером с слово и запускайте подпрограммы по четным адресам.

Детальная оптимизация.

В то время как основные правила оптимизации 8088 остаются в силе на компьютерах, построенных на базе 286 и 386, многие оптимизации, зависящие от конкретных инструкций, больше не выполняются для выполнения время выполнения большинства инструкций на 286 и 386 сильно отличается от времени выполнения 8088. Мы уже видели один такой пример иногда огромной разницы между 8088 и 286/386 раз выполнения инструкции: MOV [Wordvar], 0, у которого есть блок выполнения время выполнения составляет 20 циклов на 8088, время выполнения в ЕС составляет всего 3 цикла на на 286 и 2 цикла на 386.

Фактически, производительность практически всех инструкций доступа к памяти была значительно улучшена на 286 и 386. Ключом к этому улучшению является почти сокращение времени вычисления эффективного адреса (ЕА). Где 8088 занимает от 5 до 12 циклы для вычисления советника, 286 или 386 обычно не требуют никакого времени для выполнения вычисления. Если используется режим адресации база + индекс + смещение, такой как MOV AX,[wordArray+bx+ si], используется на 286 или 386, для выполнения советника требуется 1 цикл вычисление, но это и наихудший случай, и единственный случай, когда есть какой-либо советник накладные расходы вообще.

Исключение времени вычисления советника означает, что время выполнения ЕС в памяти- время выполнения инструкций по адресации намного ближе к времени выполнения только для регистрации в ЕС Инструкции. Например, в 8088 ADD [Wordvar] 100H - это команда из 31 цикла, В то время как ADD DX, 100H - это команда из 4 циклов - соотношение почти 8 к 1. Напротив, на 286 ДОБАВЬТЕ [WordVar], 100H - это команда из 7 циклов, в то время как ADD DX, 100H - это команда из 3 циклов инструкция — соотношение всего 2,3 к 1.

Таким образом, может показаться, что использовать регистры на 286 менее необходимо, чем это было на 8088, но это просто не так по причинам, которые мы уже видели. Ключ в этом это: 286 может выполнять инструкции по адресации памяти так быстро, что нет запасных время предварительной выборки инструкций во время выполнения этих инструкций, поэтому очередь предварительной выборки иссякает, особенно на АТ с его памятью с одним состоянием ожидания. На АТ, 6-байтовая инструкция ДОБАВЬТЕ [WordVar], 100H - это фактически, по крайней мере, 15-тактная инструкция, потому что 3 цикла - это требуется для извлечения каждого из трех слов инструкции, и требуется еще 6 циклов, чтобы прочитать wordvar и записать результат обратно в память.

Конечно, команда только для регистрации ADD DX, 100H также замедляется - до 6 циклов — из-за предварительной выборки команд, оставляя соотношение 2,5 к 1. Теперь, однако, давайте посмотрим на производительность того же кода на 8088. Код только для регистрации будет выполняться через 16 циклов (4 байта инструкций по 4 цикла на байт), в то время как код доступа к памяти будет выполняться за 40 циклов (6 байт инструкций по 4 цикла на байт плюс 2 слова размером доступ к памяти осуществляется со скоростью 8 циклов на слово). Это соотношение 2,5 к 1, точно такое же, как *на 286*.

Это все теоретически. Мы полагаемся не на теорию, а на реальную производительность, поэтому давайте запустим этот код через Zen timer. На ПК, листинг 11.4, который выполняет только регистрацию добавление выполняется за 3,62 мс, в то время как листинг 11.5, который выполняет добавление к переменной памяти , выполняется за 10,05 мс. На клоне АТ с частотой 10 МГц листинг 11.4 выполняется за 0,64 мс, в то время как Листинг 11.5 выполняется за 1,80 мс. Очевидно, что АТ намного быстрее ... но соотношение листинга 11.5 к листингу 11.4 практически идентичны на обоих компьютерах, на уровне 2.78 для ПК и 2.81 для АТ. Во всяком случае, форма ADD только для регистрации имеет несколько большее преимущество на АТ, чем в данном случае на ПК.

Теория подтверждена.

ЛИСТИНГ 11.4 L11-4.ASM

```
>; *** Listing 11.4 ***
; Measures the performance of adding an immediate value
; to a register, for comparison with Listing 11.5, which
; добавляет немедленное значение к переменной памяти.

    вызовите ZTimerOn
    повторите 1000
    Добавитьdx, 100h
    endm
    позвонитеZTimerOff
```

<ЛИСТИНГ 11.5 L11-5.ASM

```
>; *** Listing 11.5 ***
; Measures the performance of adding an immediate value
; to a memory variable, for comparison with Listing 11.4,
; which adds an immediate value to a register.
,
    jmpSkip
    even;always make sure word-sized memory
        ; variables are word-aligned!

WordVar dw0
;
Skip:
    callZTimerOn
    rept1000
    add[WordVar] 100h
    endm
    callZTimerOff
```

<Что происходит? Проще говоря: выборка команд контролирует общее время выполнения на обоих процессорах. Как 8088 в ПК, так и 286 в АТ могут выполнять байты инструкции в листингах 11.4 и 11.5 выполняются быстрее, чем их можно получить. Поскольку инструкции имеют одинаковую длину на обоих процессорах, само собой разумеется, что соотношение общего времени выполнения инструкций также должно быть одинаковым на обоих процессорах. Длина инструкции определяет время выполнения, а сама инструкция длины одинаковы - следовательно, и соотношение времен выполнения одинаковое. 286 может извлекать и выполнять байты инструкций быстрее, чем 8088, поэтому код выполняется намного быстрее на 286; тем не менее, поскольку 286 также может выполнять эти байты инструкций намного быстрее, чем он может их извлекать, общая производительность по-прежнему в значительной степени ниже определяется размером инструкций.

Всегда ли это так? Нет. Когда очередь предварительной выборки заполнена, происходит доступ к памяти инструкции на 286 и 386 намного быстрее (по сравнению с инструкциями только для регистрации) чем на 8088. Однако, учитывая состояния системного ожидания, преобладающие на компьютерах 286 и 386 очередь предварительной выборки, вероятно, будет довольно часто пустой, особенно когда выполняется код, состоящий из инструкций с коротким временем выполнения EU. Конечно, это как раз тот код, который мы, скорее всего, напишем при оптимизации, поэтому производительность высокоскоростного кода, скорее всего, зависит от размера инструкции, чем от время выполнения EU на большинстве компьютеров 286 и 386, как и на ПК.

Все это лишь способ сказать, что более быстрый доступ к памяти и расчет ЕА несмотря на это, так же желательно, чтобы инструкции были короткими, а доступ к памяти - минимум на 286 и 386, как и на 8088. И способ сделать это - использовать регистрирует как можно больше, использует строковые инструкции, использует краткие формы инструкций, и тому подобное.

Чем больше вещей меняется, тем больше они остаются неизменными...

Нам нужно обсудить последний вопрос, связанный с 286: аппаратный сбой POPF при определенных обстоятельствах на 286.

Проблема заключается в следующем: иногда POPF разрешает прерывания, когда прерывания изначально отключено, и параметр, добавленный во флаг прерывания из стека, сохраняет прерывания ВЫКЛ. Другими словами, прерывание может произойти, даже если флаг прерывания никогда не установлен в 1. Я не хочу раздувать эту конкретную ошибку до предела. Это только вызывает проблемы в коде, который ни при каких обстоятельствах не допускает прерываний, а это редкий вид код, особенно в пользовательских программах. Однако некоторый код действительно должен иметь прерывания полностью отключены, без каких-либо шансов на прерывание. Например, например, критической части BIOS диска может потребоваться извлечение данных с диска контроллер в тот момент, когда он становится доступным; задержка даже в несколько сотен микросекунд может привести к неправильному прочтению данных целого сектора. В этом случае одно неуместное прерывание во время POPF это может привести к повреждению жесткого диска, если это прерывание происходит во время работы диска BIOS считывает сектор таблицы распределения файлов.

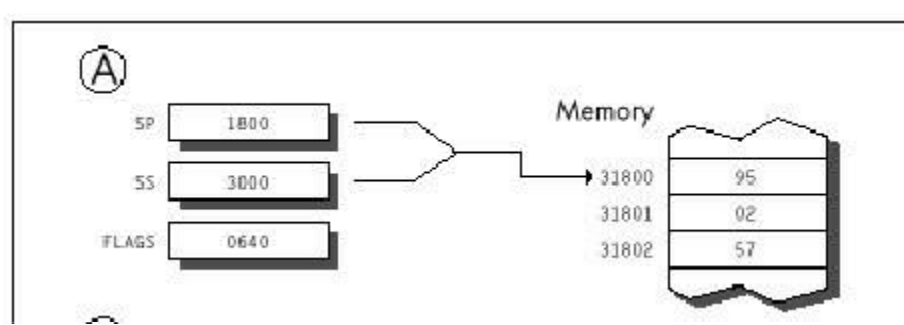
Существует обходной путь для ошибки POPF. Хотя обходной путь прост в использовании, он значительно медленнее, чем POPF, и к тому же стоит несколько байт, поэтому вам не захочется использовать это в коде, который может переносить прерывания. С другой стороны, в коде, который действительно не может быть прерван, вы должны рассматривать эти дополнительные циклы и байты как дешевую страховку от таинственный и неустойчивый сбой программы.

Одна из очевидных причин обсудить обходной путь POPF заключается в том, что он полезен. Другая причина в том, что что обходной путь является отличным примером кодирования ассемблера на уровне Zen, поскольку есть четко определенная цель, которую необходимо достичь, но нет очевидного способа сделать это. Цель состоит в том, чтобы воспроизвести функциональность инструкции POPF без использования POPF, и для этого нужно начать с вопроса, что именно делает POPF.

Все, что делает POPF, - это помещает слово сверху стека в регистр FLAGS, как показано на Рисунок 11.4. Как мы можем это сделать без POPF? Конечно, дизайнеры 286-го предполагали, что Мы будем использовать POPF для этой цели и намеренно не предоставили никакой альтернативы подход, поэтому нам придется разработать собственный альтернативный подход. Для этого нам придется поискать инструкции, которые содержат некоторые из тех же функций, что и POPF, в надежда на то, что одна из этих инструкций может быть каким-либо образом использована для замены POPF.

Ну, есть только одна инструкция, отличная от POPF, которая загружает регистр FLAGS напрямую из стека, и это IRET, который загружает регистр FLAGS из стека по мере того, как он ветви, как показано на рис. 11.5. у iret нет известных ошибок, подобных тем, которые досаждают POPF, так что это, безусловно, кандидат на замену popf в приложениях без прерываний. К сожалению, IRET загружает регистр FLAGS с третьим словом в стек, а не слово в верхней части стека, как в случае с POPF; дальний обратный адрес, который я ПОЛУЧАЮ всплывает в CS: IP находится между вершиной стека и словом, всплывшим во ФЛАГАХ Зарегистрироваться.

Очевидно, что segment:offset, который IRET ожидает найти в стеке над pushed флаги отсутствуют, когда стек настроен для POPF, поэтому нам придется немного скорректировать стек , прежде чем мы сможем заменить IRET на POPF. Что нам нужно будет сделать, так это нажать на сегмент:смещение инструкции после нашего кода обхода в стек прямо над перемещенные флаги. затем IRET перейдет по этому адресу и установит флаги, в конечном итоге по адресу выскочила инструкция после кода обходного решения с флагами. Это всего лишь результат это произошло бы, если бы мы выполнили POPF — С бонусом, что прерываний нет может случайно произойти, когда флаг прерывания равен 0 как до, так и после pop.



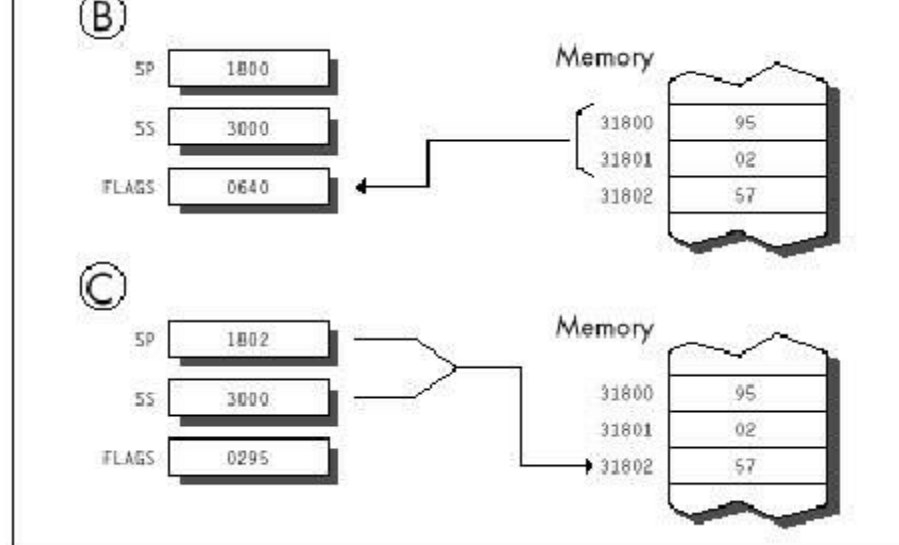
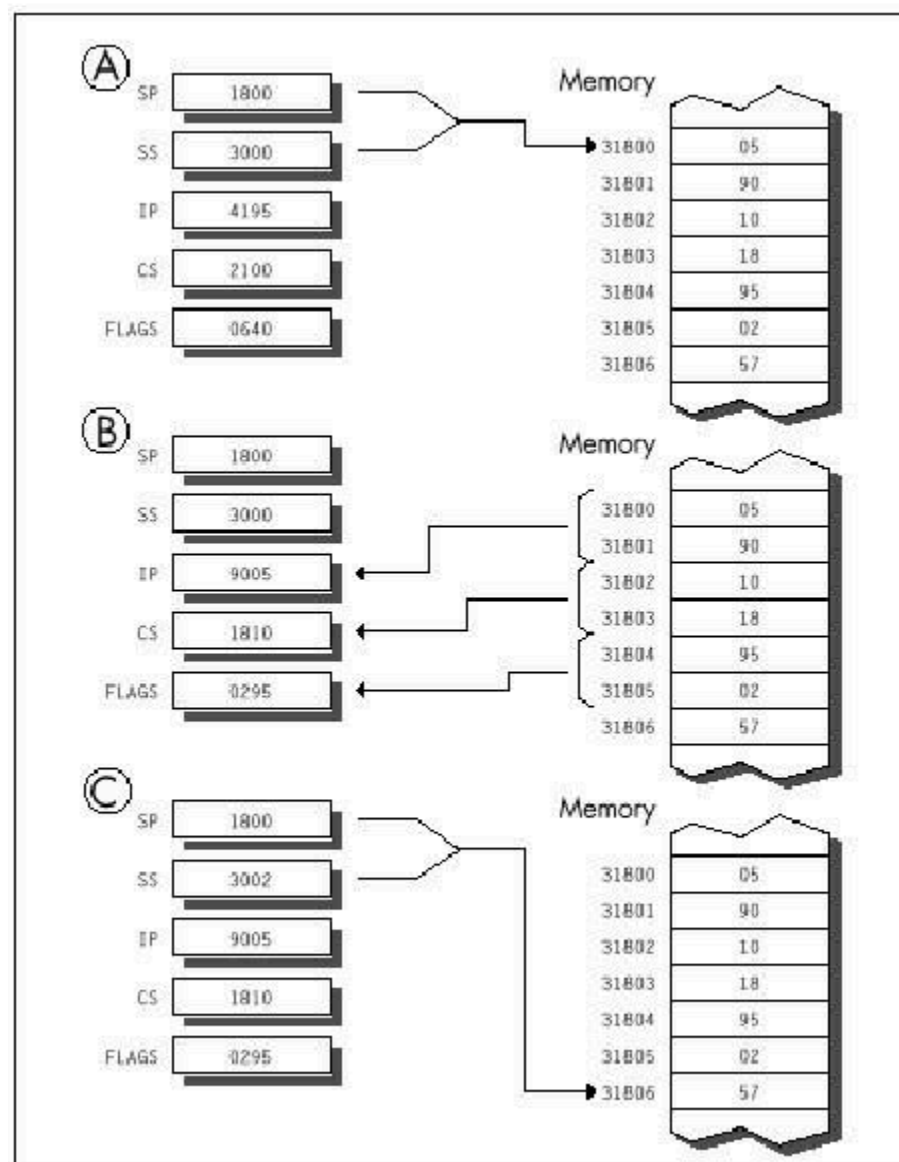


Рисунок 11.4 Работа POPF.

Как мы можем нажать сегмент: смещение следующей инструкции? Что ж, нахождение смещения в следующей инструкции путем выполнения ближайшего вызова этой инструкции - проверенный трюк. Мы можем сделать что-то подобное здесь, но в этом случае нам нужен вызов far, поскольку IRET требуется как сегмент, так и смещение. Мы также разветвляемся в обратном направлении, чтобы адрес нажатие на стек укажет на инструкцию, с которой мы хотим продолжить. Код работает следующим образом:

```
>jmp short popfskip
popfiret:
    iret; branches to the instruction after the
        ; call, popping the word below the address
        ; pushed by CALL into the FLAGS register
popfskip:
    call far ptr popfiret
        ; pushes the segment:offset of the next
        ; instruction on the stack just above
        ; the flags word, setting things up so
        ; that IRET will branch to the next
        ; instruction and pop the flags
; When execution reaches the instruction following this comment,
; the word that was on top of the stack when JMP SHORT POPFSKIP
; was reached has been popped into the FLAGS register, just
as
; if a POPF instruction had been executed.
```



<На рисунке 11.5 Показана работа IRET.

Работа этого кода проиллюстрирована на рисунке 11.6.

Обходной путь POPF лучше всего реализовать в виде макроса; мы также можем эмулировать far вызовите, нажав CS и выполнив ближний вызов, тем самым сократив код обходного пути на 1 байт:

```
>EMULATE_POPFmacro
    local popfskip, popfiret
    jmp short popfskip
```

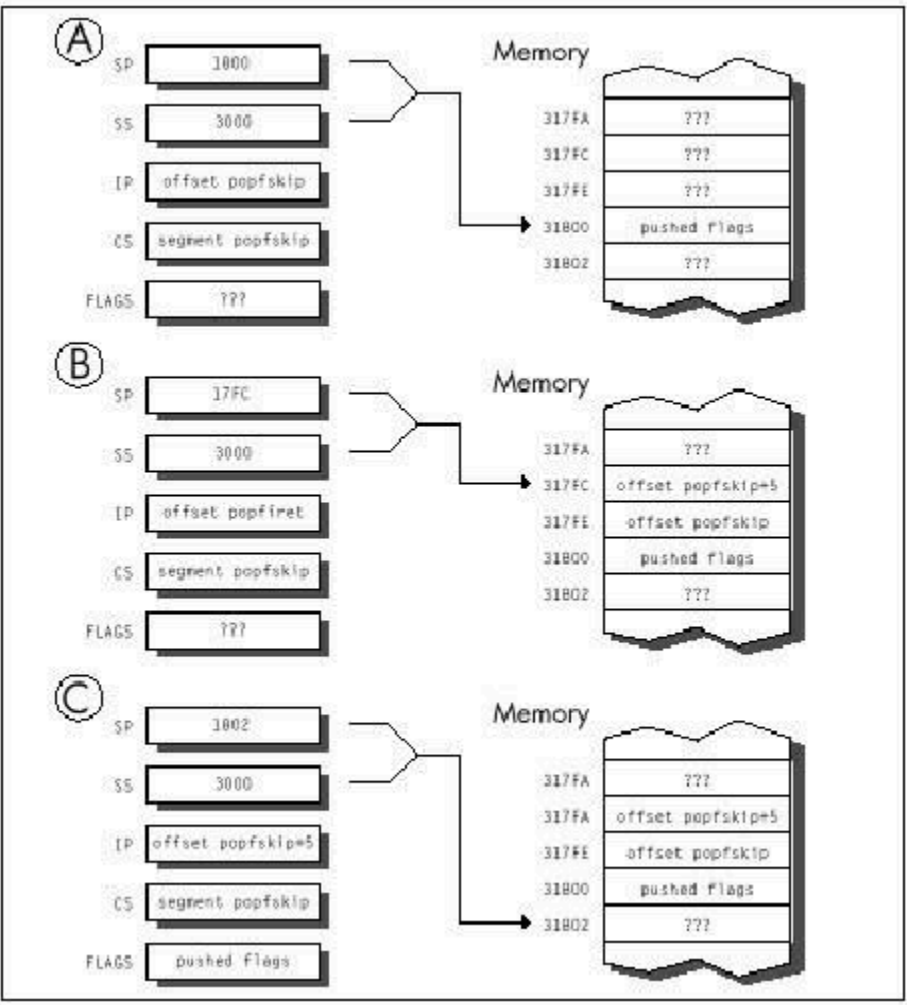
```
<>popfret:
    iret
popfskip:
    push cs
    call popfret
endm
```

<Кстати, флаги могут быть установлены намного быстрее, если вы хотите изменить регистр в процессе. Например, следующий макрос эмулирует POPF только одним переход, но стирает AX:

```
>EMULATE_POPF_TRASH_AX macro
    push cs
    mov ax,offset $+5
    push ax
    iret
endm
```

<Это не идеальная замена POPF, поскольку POPF не изменяет никаких регистров, но это быстрее и короче, чем EMULATE_POPF, когда вы можете сэкономить регистр. Если вы используете 286-конкретные инструкции, которые вы можете использовать, которые еще короче, не изменяют регистры и разветвляются только один раз. (Конечно, эта версия EMULATE_POPF не будет работать на 8088.)

```
>. 286
EMULATE_POPFmacro
    push cs
    push offset
    $+4 iret
endm
```



<Рисунок 11.6 Код обхода ошибки POPF.

Стандартная версия EMULATE_POPF на 6 байт длиннее POPF и намного медленнее, как и следовало ожидать, учитывая, что он включает в себя три ветви. Любой человек в здравом уме предпочел бы POPF более крупному, медленному макросу с тремя ветвями — при наличии выбора. В некое, однако здесь нет выбора; более безопасный — хотя и медленный — подход является лучшим. (Имея люди связывают ваши программы с вышедшими из строя компьютерами - это нежелательная ситуация, независимо от независимо от того, насколько несправедливы обстоятельства, при которых это происходит.)

И теперь вы знаете природу и обходной путь для ошибки POPF. Независимо от того, понадобится ли вам когда-нибудь обходной путь или нет, это аккуратно упакованный пример огромной гибкости набора инструкций x86.

Глава 12 – Переход к 486

Это не просто более крупный 386-й.

Итак, этот коммивояжер идет по дороге и видит группу мужчин, копающих канаву голыми руками. “Ого, вот!” - говорит он. “Что вам, ребята, нужно, так это модель” Канавокопатель 8088!” И он достает совок и продает его им.

Через несколько дней он заезжает снова. Они довольны совком, но он продает их новейшая технология рытья канав - лопата модели 80286. Это позволяет им оставаться довольными пока он снова не зайдет с лопатой модели 80386 (шириной 32 дюйма с узким указывает, имитируя шпатель), и это удерживает их, пока он не вернется с тем, что им действительно нужно: бульдозером модели 80486.

Достигнув вершины очереди, продавец некоторое время не звонит им. Когда он это делает, они не только не слишком дружелюбны, но и копают 80386 лопатой; бульдозер стоит в стороне. “Какого черта вы пользуетесь этой лопатой?” спрашивает продавец. “Почему вы не копаете бульдозером”?

“Видит бог, мы пытались”, - говорит бригадир “но все, что мы могли сделать, это просто поднять” чертову штуковину!”

Замените “процессор” на различные землеройные орудия, и вы получите представление о том, насколько правила оптимизации для 486 отличаются от тех, к которым вы привыкли. Хорошо, это не так уж и плохо — но при встрече с процессором, в котором используются строковые инструкции часто следует избегать, а перемещения из памяти в регистр часто выполняются так же быстро, как и из регистра в - регистрируйте движение! - воскликнула Дороти (прежде чем она скрылась из виду в водовороте безнадежно перепутанные метафоры) “Я не думаю, что мы больше не в Канзасе, Тото”.

Войдите в 486

Ни один чип, являющийся прямым, полностью совместимым потомком 8088, 286 и 386, не смог бы когда-либо меня называли чипом RISC, но 486, безусловно, содержит элементы RISC, и именно они элементы, которые в наибольшей степени отвечают за уникальность оптимизации 486. Простой, распространенный инструкции выполняются за один цикл процессором Core, подобным RISC, но другие инструкции выполняются практически так же, как и на 386, где каждая инструкция занимает не менее 2 циклов. Например, перемещение AL, [TestChar] занимает всего 1 цикл на 486, предполагая, что и инструкции, и данные находятся в кэше — на 3 цикла быстрее, чем у 386—но stosb занимает 5 циклов, что на 1 цикл медленнее, чем у 386. Модуль выполнения с плавающей запятой внутри 486 также намного быстрее, чем математический сопроцессор 387, в основном потому, что, поскольку он выполнен на том же кремнии, что и центральный процессор (в 486 встроен математический сопроцессор), он более тесно связан. Результаты иногда бывают поразительными: FMUL (умножение с плавающей запятой) обычно быстрее на 486, чем IMUL (умножение на целое число)!

Энциклопедический подход к оптимизации 486 сам по себе занял бы целую книгу, поэтому в этой главе я собираюсь затронуть только основные моменты оптимизации 486, затронув несколько правил оптимизации, некоторые задокументированы, некоторые нет. Возможно, вы также захотите ознакомиться с *следующие источники информации о 486: Справочник программиста микропроцессора i486* Руководство от Intel; “Оптимизация 8086: цельтесь в середину и молитесь”, в Марте, Журнал доктора Добба за 1991 год; и “Максимальная производительность: на пути к 486” в ноябре, *Журнал программиста за 1990 год*.

Правила оптимизации с помощью

В Приложении G к справочному руководству программиста микропроцессора i486 Intel перечисляет ряд методов оптимизации для 486. Они не являются исчерпывающими (мы рассмотрим две недокументированные оптимизации в ближайшее время) и не совсем точные (мы исправим два из правил здесь), список Intel, безусловно, является хорошей отправной точкой. В частности, список отражает степень, в которой оптимизация 486 отличается от оптимизации для более ранних процессоров x86. Как правило, я буду обсуждать оптимизацию для реального режима (на данный момент это наиболее широко используемый режим), хотя многие правила должны применяться и к защищенному режиму



оптимизация 486, как правило, более точная и менее разочаровывающая, чем оптимизация для других процессоров x86, потому что каждый 486 имеет идентичный внутренний кэш. Всякий раз, когда и выполняемые инструкции, и данные, к которым обращаются инструкции, находятся в кэше, эти инструкции будут выполняться в постоянное и поддающееся вычислению количество циклов на всех 486-х секундах с небольшой вероятностью помехи из очереди предварительной выборки и без учета скорости внешней памяти.

Другими словами, для кэшированного кода (которым почти всегда является критичный по времени код) производительность предсказуема и может быть рассчитана с хорошей точностью, и эти вычисления будут применить на любом 486. Однако “предсказуемый” не означает “тривиальный”; напечатанное время цикла для различных инструкций это еще не вся история. Вы должны быть осведомлены обо всех правилах, документированных и undocumented, которые используются для расчета фактического времени выполнения — и раскрытие некоторых из этих правил - именно то, чему посвящена эта глава.

Опасности индексированной адресации

Правило № 1: Избегайте индексированной адресации (то есть старайтесь не использовать ни два регистра, ни масштабируемую адресацию для указания на память).

Intel предостерегает от использования индексации для адресации памяти, поскольку существует один цикл штрафа за индексированную адресацию. Достаточно верно — но “индексированная адресация” может означать не то, что вы ожидаете.

Традиционно SI и DI считаются индексными регистрами процессоров x86. Это не однако здесь подразумевается “индексированная адресация”. В реальном режиме индексированная адресация означает, что для указания на используются два регистра, а не один или ни одного, память. (В этом контексте использование одного регистра для адресации памяти является “базовой адресацией”, независимо от того, какой регистр используется.) ПЕРЕМЕЩЕНИЕ AX, [BX+ DI] and MOV CL, [BP + SI + 10] выполняют индексированную адресацию; MOV AX, [BX] и MOV DL, [SI + 1] делают нет.



Следовательно, в реальном режиме правило состоит в том, чтобы избегать использования двух регистров для указания на память, когда это возможно. Часто это просто означает добавление двух регистрируется вместе вне цикла до фактического обращения к памяти.

В качестве примера, вы могли бы придерживаться этого правила, заменив код

```
>LoopTop:
  add ax,[bx+si]
  add si,2
  dec cx
  jnz LoopTop
```

<следующим

```
>add si,bx
LoopTop:
  add ax,[si]
  add si,2
  dec cx
  jnz LoopTop
  sub si,bx
```

<который вычисляет ту же сумму и оставляет регистры в том же состоянии, что и в первом пример, но избегает индексированной адресации.

В защищенном режиме определение индексированной адресации немного сложнее. Использование двух регистров для адресации памяти, как в MOV EAX, [EDX + EDI], по-прежнему влечет за собой штраф в размере одного цикла. Кроме того, использование масштабированной адресации 386/486, как в mov [ECX*2], EAX,

также представляет собой индексированную адресацию, даже если для указания на память используется только один регистр.

Вся эта суета из-за одного цикла! Вы вполне можете задаться вопросом, насколько сильно может измениться один цикл . В конце концов, на 8088 вычисление эффективного адреса занимает минимум 5 циклов. Однако на 486 1 цикл имеет большое значение, потому что многие инструкции, включая большинство инструкции только для регистрации (MOV, ADD, CMP и т.д.) Выполняются всего за 1 цикл. В частности, Перемещения в память и из нее выполняются за 1 цикл — если им не мешает что-то вроде индексированная адресация, в этом случае они замедляются до половины скорости (или хуже, как мы увидим вкратце).

Например, рассмотрим пример суммирования, показанный ранее. Версия, использующая адресацию по базе + индексу ([BX + SI]), выполняется за восемь циклов в цикле. Как и ожидалось, версия, использующая базовую адресацию ([SI]), выполняется на один цикл быстрее, по семь циклов за цикл. Однако код цикла выполняется на 486 так быстро, что за один цикл, сэкономленный с помощью базовая адресация ускоряет весь цикл более чем на 14 процентов.

В ключевом цикле на 486 действительно может иметь значение 1 цикл.

Вычисляйте указатели памяти заранее.

Правило № 2: Не используйте регистр в качестве указателя памяти в течение следующих двух циклов после загрузки его.

Intel заявляет, что если назначение одной инструкции используется в качестве базовой адресации компонент следующей инструкции, то налагается штраф за один цикл. Это правило, в отличие от все, что когда-либо ранее встречалось в семействе x86, отражает сильно конвейерный характер 486. По-видимому, 486 запускает каждое вычисление эффективного адреса перед началом инструкция, для которой это потребуется, как показано на рисунке 12.1; это эффективно уменьшает адрес время вычисления исчезает, потому что это происходит во время выполнения предыдущей команды.

Конечно, 486 не может выполнить эффективное вычисление адреса для целевой команды заранее, если один из компонентов адреса неизвестен до запуска команды, и это именно тот случай, когда предыдущая команда изменяет один из регистров адресации целевой команды . Например, в коде

```
>MOV BX, OFFSET
MemVar MOV AX, [BX]
```

<486 никак не может вычислить адрес, на который ссылается mov Ax, [Bx] до ПЕРЕМЕЩЕНИЕ BX, СМЕЩЕНИЕ MemVar завершается, поэтому конвейерная обработка этого вычисления заранее невозможна возможно. Хорошим обходным путем является изменение вашего кода таким образом, чтобы хотя бы одна инструкция лежит между загрузкой указателя памяти и его использованием. Например, последующее удаление, как в следующем примере

```
LoopTop:
Добавить    ax, [si]
Добавить    si, 2
dec        cx _
jnz         LoopTop
```

выполняется быстрее, чем предварительное включение, как в:

```
LoopTop:
Добавить     si, 2
Добавить    ax, [SI]
декабрь    cx _
jnz         LoopTop
```

Теперь, когда мы понимаем, что Intel подразумевает под этим правилом, позвольте мне сделать очень важный комментарий: Мои наблюдения показывают, что для кода реального режима документация занижает размер штрафа за прерывание конвейера вычисления адреса на загрузка указателя памяти непосредственно перед его использованием.



Суть дела, по-видимому, в том, что если регистр является местом назначения одной инструкции, а затем используется следующей командой для обращения к памяти в реальном режиме, теряется не один, а два цикла!

Однако в 32-разрядном защищенном режиме штраф фактически равен 1 циклу, который Intel.

Учитывая, что перемещение обычно занимает всего один цикл, это довольно большая потеря. Для примера, цикл после уменьшения, показанный выше, на 2 полных цикла быстрее, чем цикл предварительного увеличения, что приводит к 29-процентному улучшению производительности всего цикла. Но подождите, это еще не все. Если регистр загружен за 2 цикла (что обычно означает 2 инструкции, но, поскольку некоторые 486 инструкции занимают более 1 цикла,

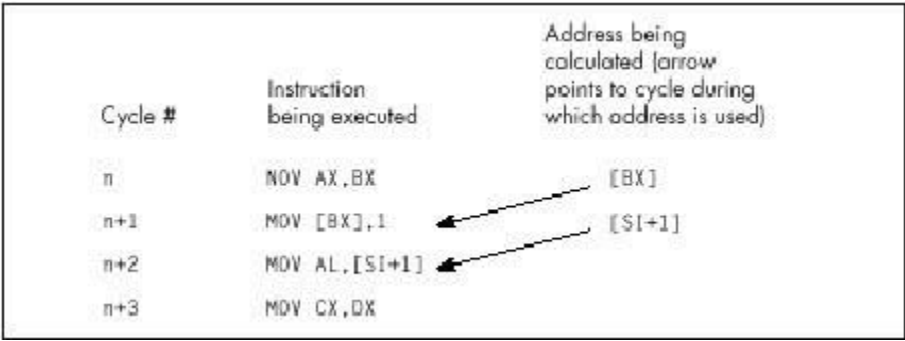


Рисунок 12.1 Конвейерная обработка адресов с опережением на один цикл.

2 не всегда эквивалентны) перед использованием для указания на память теряется 1 цикл. Следовательно, принимая во внимание, что этот код

```
mov bx, смещение MemVar
mov ax, [bx]
inc dx _
dec cx _
jnz LoopTop
```

теряет два цикла из-за прерывания конвейера вычисления адреса, этот код

```
>movbx,offset MemVar
incdx _
movax, [bx]
decCX _
jnzLoopTop
```

<теряет только один цикл, и этот код

```
>movbx,offset MemVar
incdx _
deccx _
movax, [bx]
jnzLoopTop
```

<циклы вообще не теряются. По-видимому, конвейер вычисления адресации 486 на самом деле запускается на 2 цикла вперед, как показано на рисунке 12.2. (По правде говоря, это мое лучшее предположение на данный момент заключается в том, что конвейер адресации действительно запускается только на 1 цикл вперед; дополнительный цикл возникает, когда конвейеру адресации приходится ждать записи регистра в файл регистра, прежде чем он сможет прочитать его для использования в вычислениях адресации. Однако я здесь я предполагаю, и модель с опережением на 2 цикла на рисунке 12.2 отлично подойдет для целей оптимизации.)

Очевидно, что тщательная перестановка 486 обладает значительным потенциалом оптимизации код.

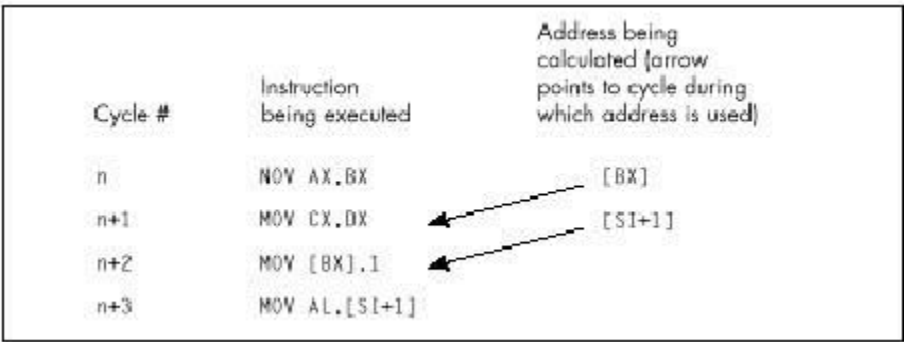


Рисунок 12.2 Конвейерная обработка адресов с опережением на два цикла.

Предупреждение: я совершенно уверен, что устранение прерывания конвейера на 2 цикла вперед описанный мной штраф существует в двух тестируемых мною 486-х устройствах. Однако гарантии нет что Intel не изменит этот аспект 486 в будущем, особенно учитывая, что в документация указывает на обратное. Возможно, ограничение на 2 цикла является результатом ошибки в начальные шаги 486 и когда-нибудь вернутся к документированному штрафу за 1 цикл; аналогично для недокументированных оптимизаций, которые я опишу ниже. Тем не менее, ни одна из оптимизации, которые я предлагаю, повредят производительности, даже если недокументированные рабочие характеристики 486-го устройства исчезнут, и они, безусловно, помогут производительность по крайней мере на некоторых 486-х прямо сейчас, поэтому я считаю, что их стоит использовать.

Конечно, нет гарантии, что я полностью прав в отношении оптимизаций обсуждаемых в этой главе. Не зная внутренностей 486, все, что я могу сделать, это потратить время кодируйте и делайте выводы из результатов; Я приглашаю вас выработать свои собственные правила и сверьте их с моими. Также, скорее всего, есть другие оптимизации, о которых я не знаю. Если у вас есть дополнительная информация об этих или любых других недокументированных для оптимизации, пожалуйста, напишите и дайте мне знать. И, конечно, если кто-нибудь из Intel читает это и хочет рассказать нам евангельскую истину, пожалуйста, сделайте это!

Адресация стека и конвейерная обработка адресов

Правило №2А: Правило № 2 иногда, но не всегда, применяется к указателю стека, когда он неявно используется для указания на память.

Intel заявляет, что указатель стека является подразумеваемым регистром назначения для ВЫЗОВА, ВВОДА, LEAVE, RET, PUSH и POP (которые изменяют (E) SP), и что это подразумеваемая базовая адресация зарегистрируйтесь для PUSH, POP и RET (которые используют (E) SP для адресации памяти). Затем Intel подразумевает что вышеупомянутый штраф за конвейер адресации возникает всякий раз, когда стек указатель используется в качестве пункта назначения одной из первых команд, а затем немедленно используется для обращения к памяти одной из вторых команд. Это вызывает проблему из-за неприятных программных искажений, таких как смешивание нажатий И всплывающих окон с другими инструкциями, позволяющими избежать прерывания конвейера адресации. К счастью, вопросы на самом деле все не так мрачно, как указано в документации Intel; мои тесты показывают, что штраф за конвейер адресации появляется только точно, когда задействован указатель стека.

Например, вы, конечно, ожидаете, что последовательность, такая как

```
pop     ax    _
ret
pop     ax    _
снова
:
```

, будет демонстрировать явление прерывания конвейера адресации (SP является одновременно адресатом и регистр адресации для обеих инструкций, согласно Intel), но этот код выполняется за шесть циклов на пару POP / RET, что в точности соответствует официальному времени выполнения. Аналогично, последовательность, подобная

```
pop     dx    _
pop     cx    _
pop     bx    _
pop     ax    _
```

выполняется за один цикл для каждой инструкции, как и должно быть.

С другой стороны, выполнение арифметики непосредственно в SP в качестве явного назначения — для например, для освобождения локальных переменных — и затем использование PUSH, POP, OF RET, определенно может прерывание конвейера адресации. Например,

```
добавить sp, 10h
ret
```

теряет два цикла, потому что SP является явным назначением одной инструкции, а затем

подразумеваемый регистр адресации для следующей, и последовательность

```
добавить    sp, 10
            часов pop_
```

теряет два цикла по той же причине.

Я, конечно, не перепробовал все возможные комбинации, но результаты пока показывают, что указатель стека подвергается штрафу за адресацию конвейера только в том случае, если (E) SP является явным местом назначения одной инструкции и затем используется одной из двух следующих инструкций для адресации памяти. Так, например, SP не является явным операндом POP AX-AX есть—и нет циклы теряются, если за POP AX следует POP или RET . К счастью, тогда нам не нужно беспокоиться о последовательности, в которой мы используем PUSH и POP. Однако добавление в, перемещение в или в идеале вычитание из указателя стека должно выполняться по крайней мере за два цикла до PUSH, POP, RET или любая другая команда, использующая указатель стека для обращения к памяти.

Проблемы с байтовыми регистрами

Есть два способа потерять циклы с помощью байтовых регистров, и ни один из них не насколько я знаю, Intel задокументировала. Давайте начнем с меньшего и более простого из двух.

Правило № 3: Не загружайте байтовую часть регистра во время одной инструкции, затем используйте этот регистр целиком в качестве исходного регистра во время следующей инструкции.

Так, например, было бы плохой идеей сделать это

```
movax, 0

mov     cx, [MemVar1]
mov     al, [MemVar2]
добавить    cx, ax
```

поскольку AL загружается одной инструкцией, то AX используется в качестве исходного регистра для следующей инструкции. Цикл можно сохранить, просто переставив инструкции таким образом, чтобы за загрузкой байтового регистра не сразу следовало использование словарного регистра, например так:

```
movax, 0

mov     al, [MemVar2]
mov     cx, [MemVar1]
добавить    cx, ax
```

Каким бы странным это ни казалось, это правило не является ни произвольным, ни бессмысленным. В принципе, когда регистр назначения байтов является частью регистра источника слов для следующей инструкции, 486 не может напрямую использовать результат первой инструкции в качестве источника для второй инструкция, потому что только часть регистра, требуемого второй инструкцией, содержится в результате первой инструкции. Необходимо прочитать полное, обновленное значение регистра из файла register, и это значение не может быть считано до тех пор, пока результат первой команда не будет записана в файл register, процесс, который требует дополнительного цикла. I'm не собираюсь объяснять это очень подробно, потому что не важно, понимаете ли вы почему это правило существует (только то, что оно действительно существует), но это интересное окно в как работает 486.

На случай, если вам интересно, для типичной последовательности XLAT такого штрафа нет, например

```
>mov bx,offset MemTable
mov al,[si]
xlat
```

<хотя AL должен быть преобразован в слово с помощью XLAT, прежде чем его можно будет добавить в BX и используется для обращения к памяти. Фактически, ни одно из наказаний, упомянутых в этой главе, не применяется к XLAT, очевидно, потому, что XLAT работает настолько медленно — 4 цикла, — что это дает 486-му времени на выполнение вычислений адресации в ходе выполнения инструкции.



Хотя приятно, что XLAT не страдает от различных штрафов за адресацию 486

 , причина этого в основном в том, что XLAT работает медленно, поэтому по-прежнему нет веская причина использовать XLAT на 486.

Как правило, штрафы за прерывание конвейера 486 применяются в первую очередь к быстродействующему ядру инструкции 486, в первую очередь инструкции только для регистра и mov, хотя арифметические и логические операции, которые обращаются к памяти, также часто затрагиваются. Я не я знаю все зависимости производительности, и я не планирую этого делать; выяснять их все было бы большой, скучной работой, не имеющей особой ценности. В принципе, на 486-м вам следует сосредоточиться на использовании этих быстрых базовых инструкций, когда важна производительность, и все правила, которые я буду обсуждать, действительно применимы к этим инструкциям.

Вам не нужно разбираться в каждом уголке вселенной 486, если только вы не несгибаемый ASMhead, который занимается этим ради удовольствия. Просто изучите достаточно, чтобы иметь возможность ускорить ключевые части ваших программ, а остальное время потратьте на быстрый дизайн и общую реализацию.

Больше удовольствия с байтовыми регистрами

Правило № 4: Не загружайте ни один байтовый регистр ровно за 2 цикла до использования любого регистра в адресуйте память.

Это последнее из правил этой главы, самое странное из всех. Если какой-либо байтовый регистр загружен, а затем двумя циклами позже любой регистр используется для указания на память, один цикл теряется. Так, например, этот код

```
>mov al,bl
mov ex, dx
mov si, [di]
```

<выполнение занимает четыре, а не ожидаемые три цикла. Обратите внимание, что это не обязательно чтобы регистр байтов был частью регистра, используемого для обращения к памяти; любой регистр байтов сделает свое дело.

Что еще хуже, загрузка байтовых регистров выполняется как за один, так и за два цикла до использования регистра для адресная память требует двух циклов, как в

```
>mov bl, al
mov c1,3 movbx,[si]
```

<для выполнения которого требуется пять, а не три цикла. Однако штрафа нет, если байт регистр загружается за один цикл, но не за два, прежде чем регистр используется для обращения к памяти. Следовательно,,

```
>mov cx,3 movdl, al
mov si, [bx]
```

<выполняется в ожидаемые три цикла.

По правде говоря, я не знаю, почему это происходит. Очевидно, что это как-то связано с прерыванием запуска конвейера адресации, и у меня есть свои теории о том, как это работает, но на данный момент это чистые предположения. Какой бы ни была причина для этого правила, незнание его — и его взаимодействия с другими правилами — может привести к значительной потере производительности в, казалось бы, герметичном коде. Например, случайный наблюдатель ожидал бы следующего код будет выполняться в 3 цикла:

```
>movbx,offset MemVar
movcl,al
movax, [bx]
```

<Более опытный программист ожидал бы потерять один цикл, потому что загружен BX два цикла перед использованием для обращения к памяти. На самом деле, однако, этот код занимает 5 циклов — на 2 цикла, или 67 процентов, больше, чем обычно. Почему? Ну, при обычном

условия, загрузка байтового регистра — в данном случае CL — за один цикл до использования регистра обращение к памяти не приводит к штрафным санкциям; загрузка на 2 цикла вперед - единственный случай, когда обычно влечет за собой штраф. Однако представьте, что Правило № 4 означает, что загрузка байта регистр прерывает конвейер адресации памяти при его запуске. Рассматривая это таким образом, мы можно видеть, что MOV BX, OFFSET MemVar прерывает конвейер адресации, заставляя его запуститься снова, и затем, предположительно, MOV CL, AL снова прерывает конвейер, потому что конвейер сейчас находится в своем первом цикле: на который может повлиять загрузка байтового регистра.



Я знаю — это кажется ужасно сложным. На самом деле это не так. Как правило, старайтесь не используйте назначения байтов ровно за два цикла до использования регистра для адреса памяти и старайтесь не загружать регистр ни за один, ни за два цикла до этого используйте его для обращения к памяти, и все будет в порядке.

Создайте свой собственный код 486.

На случай, если вы захотите провести собственный анализ производительности 486, позвольте мне показать вам как я пришел к одному из вышеприведенных выводов; в то же время я могу предупредить вас о временных опасностях кэша. В листингах 12.1 и 12.2 показан код, который я прогнал через Zen таймер, чтобы определить эффекты загрузки байтового регистра перед использованием регистра для адресной памяти. Листинг 12.1 выполнялся за 120 мкс на частоте 33 МГц 486, или по 4 цикла на повторение (120 мкс/ 1000 повторений = 120 нс за повторение; 120 нс за повторение /30 нс за цикл = 4 цикла на повторение); Листинг 12.2 выполнялся за 90 минут, или 3 цикла, из которого следует, что загрузка байтовый регистр требует цикла, только если он выполняется ровно за 2 цикла до обращения память.

ЛИСТИНГ 12.1 LST12-1.ASM

```
>; Measures the effect of loading a byte register 2 cycles before
; using a register to address memory.
    movbp,2 ;run the test code twice to make sure
                ; it's cached

    subbx,bx
CacheFillLoop:
    callZTimerOn ;start timing
    rept1000
    movdl,cl
    nop
    movax, [bx]
    endm
    callZTimerOff ;stop timing
    decbp.
    jz...Done
    jmpCacheFillLoop
Done:
```

<ЛИСТИНГ 12.2 LST12-2.ASM

```
>; Measures the effect of loading a byte register 1 cycle before
; использование регистра для обращения к памяти.
    movbp, 2 ; запустите тестовый код дважды, чтобы убедиться
                ; он кэширован

    SUBbx, bx
Цикл заполнения кэша:
вызываем ZTimerOn; запускаем синхронизацию
повторение 1000
    nop
    movdl, cl
    movax, [bx]
    endm
    вызовите ZTimerOff ; остановите хронометраж
    декабрьbp.
    jz...Выполнено
    jmpЦикл заполнения
кэша    Выполнено:
```

<Обратите внимание, что в листингах 12.1 и 12.2 время выполнения тестируемого кода повторяется с точностью до секунды время, чтобы убедиться, что инструкции находятся в кэше на втором проходе, для которого отображаются результаты. Также обратите внимание, что размер кода составляет менее 8 КБ, так что он может весь поместиться во внутренний кэш 486 размером 8 КБ. Если я удвою значение REPT в листинге 12.2 до 2000, увеличив размер тестового кода более чем на 8 КБ, время выполнения более чем удвоится до 224 единиц, или

3,7 цикла на повторение; дополнительные семь десятых цикла приходится на выборку не- кэшированные байты инструкций.



Всякий раз, когда вы видите нецелые результаты синхронизации такого рода, это хорошая ставка что тестовый код или данные не кэшируются.

История продолжается

Безусловно, есть еще много возможностей для изучения 486, включая уникальную предварительную выборку 486 очередь, дополнительные правила оптимизации, ветвления, влияние на производительность кэш, стоимость пропусков в кэше при чтении и последствия сквозной записи в кэш для операций записи. Тем не менее, в этой главе мы рассмотрели довольно много вопросов, и я надеюсь, что вы получили представление о том, в какой степени оптимизация 486 отличается от к чему вы привыкли. Какой бы странной ни была оптимизация 486, ее стоит освоить, ибо 486 в лучшем случае настолько ошеломляюще быстр, что тщательно проработанный код 486 может сделать больше более чем в два раза больше за цикл, чем у лучшего кода 386, что делает его, возможно, в 50 раз быстрее быстрее, чем оптимизированный код для оригинального ПК.

Иногда трудно поверить, что мы все еще в Канзасе!

Глава 13 – Нацеливание на 486

Трубопроводы и другие опасности высокого класса

Печальный, но верный факт: 84 процента американских школьников не знают 92 процентов американской истории. Но только не моя дочь. Недавно мы посетили исторический Старинный форт Тикондерога времен войны за независимость, и теперь она на 97 процентов знает о ключе элемент нашего национального наследия: что основной формой солдат в те дни было то, что кажется нижним бельем, плюс шляпа, чтобы никто не мог пожаловаться, что они подрывают семейные ценности. На! Шучу! На самом деле, она узнала, что в в те дни было чистым совпадением, если пушечное ядро действительно попадало во что-то, на что оно было нацелено, что неудивительно, учитывая отсутствие нарезов, точных деталей и баллистики. Проводники в форте выстрелили из трех пушек; они были ближе всего к цели, чем 50 футов, и это только потому, что помогал ветер. Я думаю, что идея ранних войн заключалась в просто выпустить в воздух столько свинца, чтобы часть его попала во что-нибудь; предпочтительно, но не обязательно во врага.

В наши дни, конечно, у нас есть автоматическое оружие, которое позволяет подростку в одиночку победить всю армию США, не говоря уже о так называемых “умных” бомбах, которые умны в том смысле, что они могут найти и опустошить кошелек налогоплательщика не будучи обнаруженными радаром. Здесь есть очевидный урок о прогрессе, который Я предоставляю вам самим делать выводы.

Вот тот же урок, в другой форме. Десять лет назад у нас был медленный процессор, 8088, для которого было чертовски сложно оптимизировать, и для которого не было никакой хорошей документация по оптимизации доступна. Теперь у нас есть процессор 486, который работает от 50 до в 100 раз быстрее, чем 8088, и для которого нет хорошей оптимизации документация доступна. Конечно, Intel предоставляет несколько полезных советов по оптимизации в

конце *справочное руководство программиста микропроцессора i486*, но, как я обсуждал в

В главе 12 эта информация является неполной и не совсем корректной. Кроме того, большинство программисты на ассемблере не утруждают себя чтением руководств Intel (которые чрезвычайно информативно и хорошо сделано, но читать лишь немного интереснее, чем с телефона книгу), и сразу приступайте к программированию 486 с использованием устаревшей оптимизации 8088 техники, пребывающие в блаженном неведении о новом и сильно мутировавшем поколении пожирателей циклов

Например, рассмотрим, как Терье Матисен удвоил скорость своего счета слов программируйте на 486, просто перетасовав пару инструкций.

Оптимизация конвейера 486

Я уже упоминал Терье Матисена в своих работах раньше. Терье - язык ассемблера выдающийся программист и автор невероятно быстрого общедоступного word-приложения программа для подсчета WC (поставляется в комплекте с исходным кодом; стоит посмотреть, если вы хотите увидеть, как выглядит действительно быстрый код). Терье - постоянный участник тема ibm.pc/fast.code на Vix. В теме “Оптимизация конвейера 486, или TANSTATFC (“Не существует такого понятия, как самый быстрый код”)”, - подробно описал он следующая оптимизация для WC, возможно, лучший пример оптимизации конвейера 486, который я видел пока не видно.

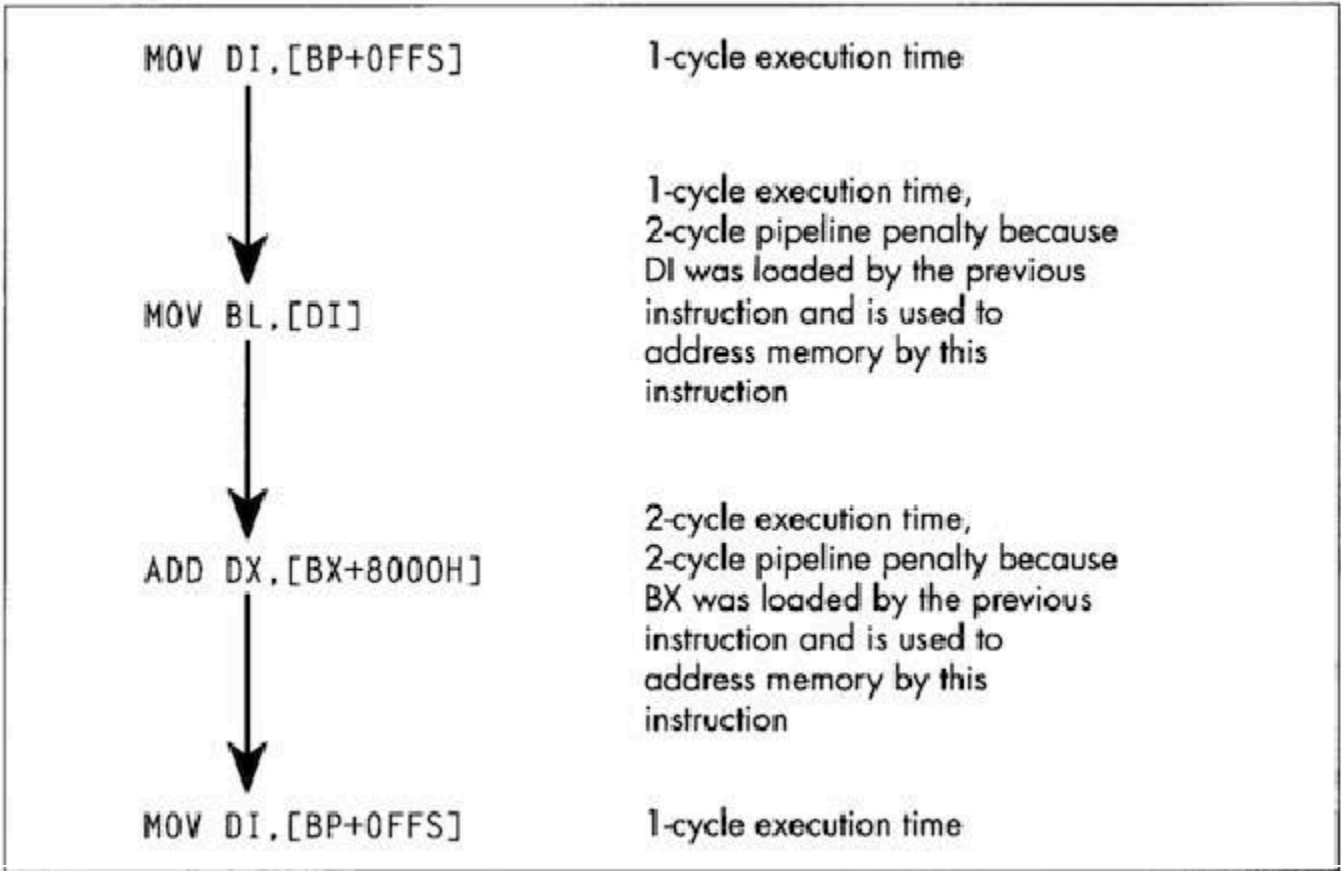
Внутренний цикл Терье изначально выглядел примерно так, как код в листинге 13.1. (Я позволил себе несколько вольностей в иллюстративных целях.) Конечно, Терье разворачивает этот цикл несколько раз (128 раз, если быть точным). Кстати, в листинге 13.1 вы заметите, что Терье считает не только слова, но и строки, из расчета три инструкции на каждые два символа!

ЛИСТИНГ 13.1 L13-1.ASM

```
переместить di, [bp + OFFS]      ; получить следующую пару символов
переместить bl, [di]             ; получить значение состояния для пары
добавить dx, [bx + 8000h]        ; увеличить количество слов и строк
                                ; соответственно для пары
```

Листинг 13.1 выглядит настолько сжатым, насколько это возможно, всего с двумя одноцикловыми инструкциями, одной двухцикловой инструкцией и без ответвлений. Это сложно, но на выполнение этих трех инструкций на самом деле требуется минимум 8 циклов, как показано на рисунке 13.1. Проблема в том, что DI является загружается непосредственно перед использованием для обращения к памяти, и это стоит 2 цикла, потому что это прерывает внутренний конвейер команд 486. Аналогично, BX загружается непосредственно перед тем, как быть используется для обращения к памяти, что требует еще двух циклов. Таким образом, этот цикл занимает в два раза больше времени как, казалось бы, указывает количество циклов, просто потому, что загружаются два регистра непосредственно перед использованием, нарушая конвейер 486.

В листинге 13.2 показан немедленный ответ Терье на эти проблемы конвейерной обработки; он просто поменял местами инструкции, загружающие DI и BL. Одно это изменение сократило время выполнения на символная пара с восьми циклов до пяти циклов! Загрузка BL теперь разделена одной инструкцией при использовании BX для адресации памяти уменьшается штраф за конвейерность от двух циклов к одному циклу. Загрузка DI также отделяется одной инструкцией от использование DI для адресации памяти (помните, цикл развернут, поэтому за последней инструкцией следует первая инструкция), но поскольку промежуточная инструкция занимает два циклов, нет никакого наказания вообще.





Помните, конвейер штрафов уменьшаются с ростом числа циклов, не инструкции, между Дизраптор трубопровода и потенциально пострадавших инструкция.

ЛИСТИНГ 13.2 L13-2.ASM

```
>mov bl,[di];get the state value for the pair
mov di,[bp+OFFS] ;get the next pair of characters
add dx,[bx+8000h] ;increment word and line count
                ; appropriately for the pair
```

<На данный момент Терье почти удвоил производительность этого кода, просто переместив одна инструкция. (Обратите внимание, что замена инструкций также привела к необходимости предварительной загрузки DI в начале цикла; Листинг 13.2 не совсем эквивалентен листингу 13.1.) Я позволю Терье описать свою следующую оптимизацию своими словами:

“Когда я присмотрелся к этому повнимательнее, я понял, что два цикла для окончательного ДОБАВЛЕНИЯ - это всего лишь сумма 1 цикла для загрузки данных из памяти и 1 цикла для добавления их в DX, так что код с таким же успехом можно было написать так, как показано в листинге 13.3. Окончательный прорыв произошел, когда я понял, что, инициализировав AX равным нулю вне цикла, я могу изменить его как показано в листинге 13.4 и выполните окончательное ДОБАВЛЕНИЕ DX, AX после цикла. Таким образом, остаются две одноцикловые инструкции между первой и четвертой строками, избегающие всего конвейера " остановки, общая пропускная способность составляет два цикла / символ ”.

ЛИСТИНГ 13.3 L13-3.ASM

```
перемещение bl, [di]           ; получите значение состояния для пары
перемещение di, [bp + OFFS]    ; получаем следующую пару символов
перемещение ax, [bx + 8000h]    ; увеличиваем количество слов и строк
добавьте dx, ax                ; соответственно для пары
```

ЛИСТИНГ 13.4 L13-4.ASM

```
перемещение bl,[di] перемещение ; получить значение состояния для пары
di, [bp + 0FFS]           ; получить следующую пару символов
добавить dx, ax           ; увеличить количество слов и строк
                        ; соответственно для пары
перемещение ax, [bx + 8000h] ; получите приращения в следующий раз
```

Я хотел бы указать на две довольно примечательные вещи. Во-первых, на одиночный цикл, который Терье сохранил в Листинг 13.4 ускорил весь свой механизм подсчета слов на 25 и более процентов; Листинг 13.4 работает в два раза быстрее, чем листинг 13.1 — и все это в результате не более чем переключения инструкция и разделение другой на две операции. Во-вторых, подсчет слов Терье движок может обрабатывать более 16 миллионов символов в секунду на 486/33.

Продуманная оптимизация 486 может принести большие плоды. QED.

BSWAP: полезнее, чем вы могли подумать

Для 486 есть только 3 несистемных инструкции, уникальных. Ни один из них не является потрясающим, но у них есть свое применение. Рассмотрим BSWAP. BSWAP делает именно то, что подразумевает его название, обмениваясь байты (не биты) 32-разрядного регистра от одного конца регистра к другому, как показано на рисунке 13.2. (BSWAP может работать только с 32-разрядными регистрами; ячейки памяти и 16-разрядные регистры не являются допустимыми операндами.) Очевидное использование BSWAP - преобразование данных из формата Intel (первый младший байт в памяти, также называемый младшим порядковым номером) в Формат Motorola (первый по старшинству байт в памяти или большой порядковый номер), примерно так:

```
lodsd
bswap
stosd
```

BSWAP также может быть полезен для изменения порядка следования битов пикселей в растровом изображении, чтобы их можно было поворачивать на 32 бита за раз с помощью такой инструкции, как ROR EAX, 1. Байт Intel

порядок многословных значений (сначала младший по значению байт) загружает пиксели в неправильном порядке, что касается ротации слов, но BSWAP может позаботиться об этом.

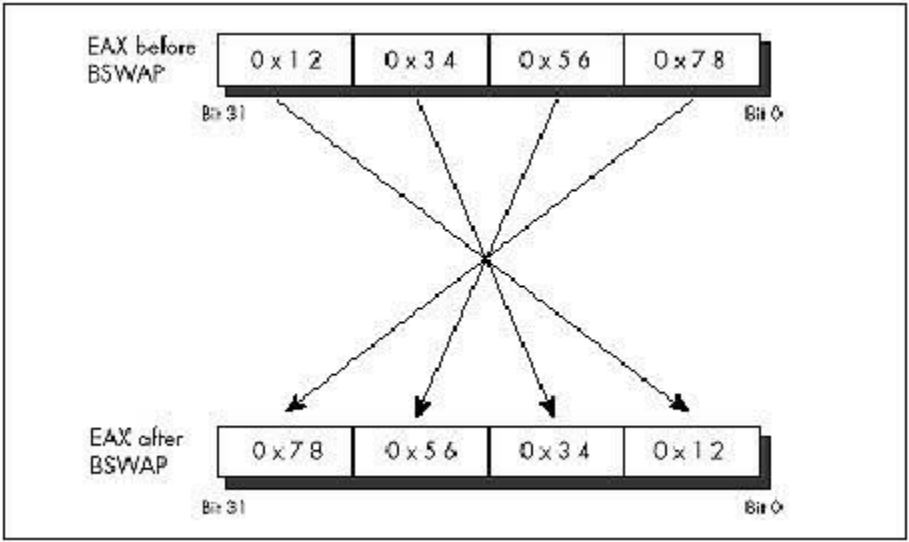


Рисунок 13.2 BSWAP в действии.

Однако, как оказалось, BSWAP полезен и неожиданным образом, поскольку связан с эффективным использование верхней половины 32-разрядных регистров. Как известно любому языку ассемблера программист знает, что набор регистров x86 слишком мал; или, другими словами, это конечно, было бы неплохо, если бы набор регистров был больше. Как известно любому языку ассемблера 386/486 программист знает, что есть много случаев, когда 16 бит достаточно. Например, 16-битный счетчик строк сканирования обычно отлично справляется с этой задачей в видеодрайвере, потому что есть очень мало видеоустройств с более чем 65 535 адресуемыми линиями сканирования. Объединение этих два наблюдения приводят к очевидному выводу, что было бы здорово, если бы было несколько способ использования верхних и нижних 16 бит выбранных 386 регистров как отдельных 16-битных регистры, эффективно увеличивающие доступное регистровое пространство.

К сожалению, набор команд x86 не предоставляет никакого способа прямой работы только с верхняя половина 32-разрядного регистра. Следующее лучшее решение - повернуть регистр, чтобы получить вы получаете доступ в младших 16 битах к половине, которая вам нужна, в любое конкретное время, с помощью кода аналогично приведенному в листинге 13.5. Приходится поворачивать 16-битные поля в нужное положение конечно, это не так хорошо, как иметь прямой доступ к верхней половине, но, безусловно, это лучше чем извлекать значения из памяти, не так ли?

ЛИСТИНГ 13.5 L13-5.ASM

```
>mov cx,[initialskip]
shl ecx,16;put skip value in upper half of ECX
mov cx,100;put loop count in CX
looptop:
    ror ecx,16;make skip value word accessible in CX
    add bx,cx;пропустить BX вперед
    incsx _; установить следующее значение пропуска
    ror ecx,16; перевести количество циклов в CX
    количество циклов в дек; цикл обратного отсчета
    цикл jnz looptop
```

<Не обязательно. Сдвиги и вращения являются одними из наихудших инструкций 486, выполнение которого занимает от 2 до 3 циклов. Таким образом, требуется 2 цикла, чтобы преобразовать значение пропуска в CX в листинге 13.5, и еще 2 цикла, чтобы повернуть его обратно к верхней половине ECX. Я бы сказал, четыре цикла - довольно высокая цена, особенно учитывая, что перемещение в память или из нее память занимает всего один цикл. По сути, использование ROR для доступа к 16-битному значению в верхней части половина 16-битного регистра - довольно маргинальный метод, если только по какой-то причине вы не можете доступ к памяти вообще (например, если вы используете BP в качестве рабочего регистра, временно делая недоступным фрейм стека).

На 386-м ROR БЫЛ единственным способом разделить 32-разрядный регистр на два 16-разрядных регистра. На однако BSWAP 486 может не только выполнять эту работу, но и делать ее лучше, потому что BSWAP выполняется всего за один цикл. Дополнительное преимущество BSWAP заключается в том, что он не влияет ни на какие флаги, в отличие от ROR. С кодом на основе BSWAP, подобным приведенному в листинге 13.6, к старшим 16 битам регистра можно получить доступ всего за 2 цикла служебных данных и без изменения каких-либо флагов, что делает технология упаковки двух 16-разрядных регистров в один 32-разрядный регистр намного полезнее.

ЛИСТИНГ 13.6 L13-6.ASM

```
mov     пример,[initialskip]
bswap   ecx      ;поместить значение пропуска в верхнюю половину ECX

mov
looptop:  cx, 100; ввести количество циклов в CX

bswap   ecx      ; сделать доступным пропущенное значение word в CX
Добавитьbx, cx   ; пропустить BX вперед
inc     cx _     ; установить следующее значение пропуска
bswap   ecx      ; введите количество циклов в CX

dec     cx
jnz     looptop; цикл обратного отсчета
```

Нажатие и выталкивание памяти

Нажатие или выталкивание ячейки памяти, как в PUSH WORD PTR [BX] ИЗ POP [MemVar], является компактный и простой способ ввода значения в стек или из него, особенно при нажатии параметры для вызова С-совместимой функции. Однако на 486 это непривлекательные инструкции с точки зрения производительности. Ввод ячейки памяти занимает четыре цикла; в отличие от этого, загрузка ячейки памяти в регистр занимает только один цикл, и нажатие на регистр занимает всего еще 1 цикл, в общей сложности два цикла. Следовательно,,

```
>movax, [bx]
push ax
```

<в два раза быстрее, чем

нажатие на слово ptr [bx]

и единственная цена заключается в том, что предыдущее содержимое AX уничтожается.

Аналогично, для извлечения ячейки памяти требуется шесть циклов, но для извлечения регистра и записи для этого в память требуется всего два цикла вместе взятых. В справочном руководстве программатора микропроцессора i486 указано время выполнения в 4 цикла для извлечения регистра, но не обращайтесь на это внимания ; для извлечения регистра требуется всего 1 цикл.

Почему такая удобная операция, как нажатие или выталкивание памяти, выполняется так медленно? Правило 486 заключается в том, что простые операции могут быть выполнены за один цикл благодаря RISC-ядру 486 выполняются быстро; в то время как сложные операции, которые должны быть выполнены выполняются в микрокоде так же, как и на 386, почти все относительно медленно. Медленно, сложные операции включают в себя все строковые инструкции, кроме повторов, а также XLAT, цикл и, конечно, PUSH * mem * и POP * mem*.



По возможности старайтесь использовать инструкции 486 по 1 циклу, включая MOV, ДОБАВЬТЕ, ЗАМЕНИТЕ, CMP, ADC, SBB, XOR, И, ИЛИ, ПРОВЕРЬТЕ, LEA, а также НАЖМИТЕ reg и POP reg.

Дополнительным преимуществом этих инструкций является то, что их часто можно изменять для достижения максимальной эффективности конвейера, как в случае с Terje оптимизация, описанная ранее в этой главе.

Оптимальный 1-битный сдвиг и поворот

На 486-м n—битные команды сдвига и поворота формируются - как в ROR AX, 2 и SHL BX,9 — это двухтактные инструкции, но 1-битные формы - как в ROR AX,1 и SHL BX,1— являются трехтактными инструкциями. Поди разберись.

Ассемблеры по умолчанию используют 1-битную инструкцию для 1-битных сдвигов и вращений. Это не неразумно, поскольку 1-битная форма на байт короче и работает так же быстро, как n-битные формы на 386 и быстрее на 286, а n-разрядная форма даже не существует на 8088. В однако по-настоящему критический цикл, возможно, стоило бы вручную собрать n-разрядную форму одноразрядного сдвига или поворота на бит, чтобы сохранить этот цикл. Самый простой способ сделать это - собрать 2-разрядную форму желаемой инструкции, как в SHL AX, 2, затем посмотрите на шестнадцатеричные коды, которые ассемблер генерирует и использует DB для вставки их в ваш программный код, при этом значение two

заменяется значением `one`. Например, вы могли бы определить, что `SHL AX, 2` выполняет сборку к байтам `0C1H 0E0H 002H`, либо просматривая дизассемблирование в отладчике, либо заставляя ассемблер сгенерировать файл списка. Затем вы могли бы вставить `n`-разрядную версию `SHL AX, 1` в свой код следующим образом:

```
>movax,1
db 0c1h, 0e0h, 001h
movdx, ax
```

<В конце этой последовательности `DX` будет содержать `2`, и будет выполнена быстрая `n`-разрядная версия `SHL AX, 1`. Если вы используете этот подход, я бы рекомендовал использовать макрос, а не вставлять `DBS` в середину вашего кода.

Опять же, этот метод выгоден только на `486`. Это также не относится к `RCL` и `RCR`, где вы определенно захотите использовать `1`-разрядные версии, когда сможете, потому что `n`-разрядные версии работают ужасно медленно. Но если вы оптимизируете для `486`-й, эти лакомые кусочки может сэкономить несколько критических циклов — и Господь знает, что если вы оптимизируете для `486` — то есть, если вам нужна еще большая производительность, чем вы получаете от неоптимизированного кода на `486` — вам почти наверняка понадобится вся возможная скорость.

32-разрядные режимы адресации

`386` и `486` оба поддерживают 32-разрядные режимы адресации, в которых любой регистр может служить регистром адресации базовой памяти, и почти любой регистр может служить потенциально масштабируемый индексный регистр. Например,,

```
перемещение, базовая таблица [ecx+ edx *4]
```

использует абсолютно корректный 32-битный адрес, при этом доступ к байту осуществляется со смещением в `DS`, на который указывает сумма `EDX`, умноженная на `4`, плюс смещение `BaseTable` плюс `ECX`. Это очень мощная схема адресации памяти, намного превосходящая 16-разрядную в стиле `8088` адресация, но она не лишена своих особенностей и затрат, поэтому давайте быстро взглянем на 32-разрядную адресация. (Кстати, 32-разрядная адресация не ограничивается защищенным режимом; 32-разрядные инструкции могут использоваться и в реальном режиме, хотя каждая команда, использующая 32-разрядный адресация должна иметь префиксный байт размера адреса, а наличие префиксного байта требует цикла на `486`.)

Любой регистр может служить компонентом базового регистра адреса. Любой регистр кроме `ESP`, также может служить регистром индекса, который может быть масштабирован на `1`, `2`, `4` или `8`. (Масштабирование очень удобно для выполнения поиска в массивах и таблицах.) Тот же регистр может служить как базовым, так и индексным регистром, за исключением `ESP`, который может быть только базовым. Кстати, имеет смысл, что `ESP` нельзя масштабировать; `ESP`, по-видимому, всегда указывает на допустимый стек, и я не могу придумать ни одной причины, по которой вы хотели бы использовать указатель стека, умноженный на `2`, `4` или `8` в адресе. `ESP` по своей природе является базовым, а не индексным указателем.

Это все, что есть в функциональности 32-разрядной адресации; это очень просто, намного проще чем 16-разрядная адресация с ее резко ограниченным регистром адресации в памяти комбинации. Затраты на 32-разрядную адресацию немного ниже. Единственный стоимость производительности (помимо вышеупомянутого штрафа в `1` цикл за использование 32-разрядной адресации в реальном режиме) составляет штраф в `1` цикл, налагаемый за использование индексного регистра. В этом в контексте вы используете индексный регистр, когда используете масштабируемый регистр или когда вы используете сумму двух регистров для указания на память. `MOV BL, [EBX*2]` использует индексный регистр и требует дополнительного цикла, как и `MOV CL,[EAX + EDX]`; `MOV CL, [EAX + 100H]`, однако, не индексируется.

Другая стоимость 32-разрядной адресации связана с размером инструкции. 16-разрядная адресация в старом стиле обычно (за исключением нескольких особых случаев) используется один дополнительный байт, который Intel называет `Mod- R / M` байт, который помещается сразу после кода операции каждой инструкции для описания режим адресации памяти плюс `1` или `2` необязательных байта смещения адресации — это то есть постоянное значение для добавления в адрес. Во многих случаях 32-разрядная адресация продолжается.

использовать Mod-R/M байт, хотя и с другой интерпретацией; в этих случаях 32-битный адресация не превышает 16-битную адресацию, за исключением случаев, когда используется 32-битное смещение . Например, MOV AL, [EBX] - это 2-байтовая инструкция; MOV AL, [EBX+10H] - 3-байтовая инструкция; и MOV AL, [EBX + 10000 H] - это 6-байтовая инструкция.



Обратите внимание, что смещения на 1 и 4 байта, но не на 2 байта, являются поддерживается 32-разрядная адресация. Размер кода может быть значительно увеличен за счет сохранения переменных стекового фрейма в пределах 128 байт EBP, а переменных в структуры с указанием в пределах 127 байт от начала структуры, так что смещения могут составлять 1, а не 4 байта.

Однако, поскольку 32-разрядная адресация поддерживает гораздо больше комбинаций адресации кроме 16-битной адресации, байт Mod-R / M не может описать все комбинации. Следовательно, всякий раз, когда задействован регистр индекса (как описано выше), за байтом Mod-R/M следует второй байт, байт SIB для предоставления дополнительной адресной информации. Следовательно, всякий раз, когда вы используете масштабированный регистр адресации памяти или используете сумму двух регистров для указания на память, вы автоматически добавляете 1 цикл и 1 байт к этой инструкции. Это не означает, что вы не должны использовать индексные регистры, когда они необходимы, но если вы обнаружите, что используете их внутри циклов ввода, вы должны посмотреть, правильно ли это возможно переместить вычисление индекса за пределы цикла, как, например, в цикле, подобном this:

```
>LoopTop:
    add ax, DataTable [ebx*2]
    inc ebx
    dec cx
    jnz LoopTop
```

<Вы могли бы изменить это на следующее для повышения производительности:

```
Добавить      ebx, ebx
               ; ebx*2
Защелкивание:
Добавить      ax, DataTable[ebx]
Добавить      ebxX, 2
dec           cx
jnz           петлевая головка
shr           ebx, 1 ; ebx * 2/2
```

Я закончу эту главу еще двумя особенностями 32-разрядной адресации. Во-первых, как и в случае с 16-разрядной адресация, адресация, использующая EBP в качестве базового регистра, одновременно обращается к SS сегмент по умолчанию и всегда имеет смещение не менее чем на 1 байт. Это отражает обычное использование EBP для обращения к фрейму стека, но стоит иметь в виду, если вам случится использовать EBP для обращения к памяти, отличной от стека.

Наконец, как я упоминал, ESP нельзя масштабировать. Фактически, ESP не может быть индексным регистром; это должен быть базовый регистр. Однако, по иронии судьбы, ESP - это единственный регистр, который не может быть использоваться для адресации памяти без наличия байта SIB, даже если он используется без регистра индекса. Это результат того, каким образом байт SIB расширяет возможности модуля-R/ M байт, и с этим ничего не поделаешь, но это, по крайней мере, стоит отметить, что неиндексированная адресация на основе ESP позволяет выполнять инструкции на байт больше, чем другая неиндексированная адресация (но не медленнее; нет 1 цикла штраф за использование ESP в качестве базового регистра) в 486.

Глава 14 – Строка Бойера-Мура

Поиск

Оптимизация довольно оптимального алгоритма поиска

Когда вы, кажется, зашли в тупик, остановитесь на минуту и подумайте. Вся необходимая вам информация

может быть прямо у вас перед носом, если вы просто посмотрите на вещи немного по-другому. Вот показательный пример:

Когда я учился в колледже, я обычно проводил лето в кампусе. О, я бы взял одно или два блюда, но в основном это был повод потусоваться и повеселиться. В этом духе мой подруга, Адриан (не моя будущая жена, отчасти по причинам, которые скоро станут очевидными), приехал на автобусе, чтобы провести неделю в менее чем эlegantной квартире за 150 долларов в месяц со мной и, по необходимости, моим соседом по комнате.

Наша квартира была в значительной степени стандартной для двух студентов колледжа мужского пола; может быть, даже на голову выше. Посуда обычно была вымыта, в холодильнике, и ничего больше маленькой собачки не поселилось в ванной. Однако был один камень преткновения (в буквальном смысле): кухонный пол. Это пол — стандартная плитка, с красивым рисунком из черных линий на грязновато-белом фоне (или так мы думали) — никогда не чистился. Под этим я подразумеваю, что я точно знаю, что мы никогда не убирали его, но я подозреваю, что на самом деле его не убирали с конца Юрского периода, а возможно, и раньше. Наши ноги, как правило, прилипали к нему; если бы квартира внезапно перевернулась вверх дном, я думаю, мы все висели бы под потолком.

Однажды мы с моей соседкой по комнате возвращались с баскетбольного матча. Адриан, оставшись на пару часов предоставленной самой себе, очевидно, нашла себе занятие. “Ничего не замечаешь?” спросила она, с краю, чтобы ее голос, что предложил, чтобы мы были прокляты ну лучше.

“Э-э, ты приготовила ужин?” Я догадался. “Помыл посуду? Ты сделала прическу?” Мой сосед по комнате тоже был без понятия.

Она топнула ногой (правда; единственный раз, когда я видела, как это происходит) и сказала “Нет, вы придурки! Кухонный пол! Посмотри на пол! Я вымыл его!”

Пол действительно выглядел потрясающе. На самом деле он был весь белый; черные линии были канавками, заполненными грязью. Мы заверили ее, что это выглядит потрясающе, просто это не было настолько очевидно пока вы не научитесь обращать на это внимание; любой скажет вам, что это не та вещь, которая набросилась на тебя, но это действительно было здорово, без шуток. Мы почти уладили ситуацию когда вошел друг, испуганно огляделся и сказал “Привет! Вы, ребята, положили новый пол?”

Как я уже сказал, иногда все, что вам нужно знать, находится прямо перед вашим носом. Который это подводит нас к поиску строк по Бойеру-Муру.

Обновление для поиска по строкам

Я обсуждал поиск по строкам ранее в этой книге, в главах 5 и 9. Возможно, вам захочется вернуться к этим главам, чтобы получить некоторую информацию о поиске строк в целом. I’m также собираюсь использовать часть кода из этой главы как часть набора тестов этой главы. Для получения дополнительной информации вы можете обратиться к обсуждению поиска по строкам в превосходные алгоритмы на C, автор Роберт Седжвик (Эддисон-Уэсли), который послужил основным справочником для этой главы. (Если вы посмотрите на Седжвика, имейте в виду, что в списке Бойера- Мура на странице 288 есть ошибка: “j > 0” в цикле for должно быть 0 = -0 0,” если только я чего-то не упускаю.)