Крис Касперски

СМЕРТЕЛЬНАЯ СХВАТКА: АССЕМБЛЕР vs. КОМПИЛЯТОР

kk@sendmail.ru

Крис Касперски
kk@sendmail.ru

Смертельная схватка: Ассемблер vs. Компилятор

"*Сохраняй за собой право думать, даже неправильно – это лучше, чем не думать совсем*".

приписывается Гиппатии

В статье приводится сравнительный анализ качества машинного кода и ручной ассемблерной оптимизации на примере широко распространенных компиляторов Microsoft Visual C++ 6.0, Borland C++ 5.5, WATCOM C++ 10.0

Введение

"Священные войны" вокруг компиляторов бушуют уже очень давно. Одни обожествляют машинную кодогенерацию, другие же стремятся все делать своими руками, порой реализуя целиком программы на чистейшем ассемблере. Конечно, всякий имеет право на выбор, но этот выбор должен делаться осмысленно, а не вслепую. Между тем, каждая из сторон, распускает совершенно неправдоподобные слухи. Приверженцы компиляторов убеждают окружающих в том, что человек физически не способен учесть все архитектурные особенности современных процессоров и, якобы, одному лишь оптимизатору эта работа по плечу. Их противники в качестве контраргумента обычно приводят ассемблерную реализацию канонической программы "Hello, World!", – по объему раз в ***триста*** меньше "самого оптимального кода", сгенерированного компиляторами.

Вот и разберись: кому же верить, а кому нет? Такая неопределенность часто нервирует начинающих программистов, пытающихся разобраться: стоит ли изучать ассемблер или же это пустая трата времени? Чтобы там ни говорили сторонники

компиляторов, машинная оптимизация ***всегда*** будет проигрывать человеку, поскольку, действует по жесткому, заранее заложенному в нее шаблону, а человек же способен на ***качественно новые*** решения. Касательно же невозможности учета архитектурных способностей современных процессоров, тем кто это утверждает, легко возразить – не стоит, право же, судить все человечество по себе. Оптимальное планирование потока команд вполне по силам программисту средней руки, конечно, при условии наличия соответствующей подготовки. К тому же, техника оптимизации процессоров последнего поколения стала значительно ***проще***, чем была лет пять тому назад. С другой стороны, ассемблер – это не волшебная лампа Аладдина, он не способен творить чудеса. Во всяком случае, реализация полиномиального алгоритма на ассемблере еще никогда не превращала его в логарифмический. За исключением особо оговариваемых случаев, речь может идти лишь о количественном, но отнюдь не качественном выигрыше (пример с "Hello, World!" – как раз и есть один из таких редких случаев, и ниже он будет рассмотрен во всех подробностях). К тому же, при неумелом обращении (или применении знаний, почерпнутых из книжек по оптимизации десятилетней давности) ручная оптимизация становится просто посмешищем компиляторов!

В общем, здесь есть, с чем разбираться...

Краткий экскурс в историю или ассемблер – это всегда весна

До середины девяностых борьба ассемблера с компиляторами шла с переменным успехом. Верх одерживала то одна, то другая сторона. После выхода новых, более быстродействующих микропроцессоров, интерес к ассемблеру на какое-то время угасал – всем казалось, что наконец-то настали те золотые времена, когда для решения подавляющего большинства задач достаточно одних лишь языков высокого уровня. Впрочем, эйфория никогда не длилась долго, вслед за возросшими процессорными мощностями усложнялись и возлагаемые на них задачи. Возможностей языков высокого уровня вновь катастрофически не хватало, и интерес к ассемблеру незамедлительно вспыхивал с новой силой.

Практически ни один, сколь ни будь заметный проект тех лет, не обошелся без ассемблера. Ассемблерный код в изобилии присутствовал и в MS-DOS, и в Windows, и в Quake, и в Microsoft Office, и... во многих других продуктах. Короче, слона шапками не закидать – плохой компилятор остается плохим и на быстрых процессорах, а потому рынок сбыта такого компилятора будет сильно ограничен.

Жесточайшая конкуренция разработчиков компиляторов, в конечном счете, обернулась благом (и в первую очередь – для прикладных программистов), ибо привела к интенсивному совершенствованию алгоритмов машинной оптимизации. Уже к концу девяностых качество оптимизирующих компиляторов практически достигло теоретического идеала, сравнявшись в решении ***штатных задач*** с программистами средней квалификации.

Сегодня, когда редкий программист обходится без визуальных средств разработки и даже "чистые" высокоуровневые языки выходят из моды, ассемблер и вовсе выглядит архаичным рудиментом, задвинутым на одну полку с перфоратором и ламповой ЭВМ. Между тем, слухи о его скорой смерти сильно преувеличены. Ассемблер жив! И лучшее подтверждение этому – тот факт, что основным языком разработки драйверов Microsoft объявляет именно его, ассемблер, а вовсе не Си\Си++ или, скажем, Паскаль. Без ассемблера (или специализированных компиляторов) невозможно использовать преимущества новых мультимедийных команд параллельной обработки данных. Наконец, ассемблер по-прежнему незаменим в создании высокопроизводительных математических и графических библиотек. В общем, ситуация с вживлением ассемблера стабилизировалась и на оставшуюся у него вотчину языки высокого уровня более не претендуют.

Критерии оценки качества машинной оптимизации

Основные критерии качества кода это: ***быстродействие***, ***компактность*** и ***время***, затраченное на его разработку. Причем, оптимизация программы по всем трем критериям невозможна в принципе. В частности, при выравнивании кода и структур данных по кратным адресам производительность программы возрастает, но вместе с нею увеличивается и ее размер.

Считается, что скорость – более приоритетная характеристика, нежели объем. Сегодня, когда количество оперативной памяти измеряется сотнями мегабайт, а емкость диска – сотнями гигабайт, компактность программного кода, действительно, уже не столь критична, однако, конечному пользователю отнюдь не все равно: сколько мегабайт занимает программа – один или миллион. Поэтому, практически все современные компиляторы поддерживают как минимум два режима оптимизации: **maximum speed** и **minimum space**.

Однако скрупулезное тестирование не входит в наши планы. Оставим это на откуп читателям, а сами ограничимся компромиссным режимом **максимальной оптимизации**, пытающийся достичь наивысшей скорости при наименьшем объеме кода. Именно такой режим и используется подавляющим большинством программистов, а потому он наиболее интересен.

Отметим также, что манипулирование ключами "тонкой настройки" оптимизатора может значительно изменить результаты тестирования, причем, как в худшую так и в лучшую стороны. Но в этом случае будут уже сравниваться не сами оптимизаторы, искусство их настройки, а это – тема совершенно другого разговора.

Важно понять: *точно оценить **общее** качество оптимизации на примере **частных** случаев невозможно*. Вот, например, Microsoft Visual C++ умеет подменять деление переменной на константу умножением, (что осуществляется в десятки раз быстрее!) а Borland C++ – нет. В зависимости от того, встречается ли константное деление в оптимизируемой программе или нет, соответствующим образом будет варьироваться и разница в быстродействии кода, сгенерированного обоими компиляторами.

Поэтому, в отсутствии конкретного примера, можно говорить лишь о приблизительной, прикидочной оценке качества компиляторов. Позднее (см. статью "*Сравнительный анализ оптимизирующих компиляторов языка Си\Си++*", опубликованное в N... журнала "Программист") мы рассмотрим этот вопрос во всех подробностях, а сейчас же нас в первую очередь будет интересовать усредненное качество машинной оптимизации на примере типовых алгоритмов.

Конечно, понятие "типовой алгоритм" очень относительное и субъективное. Для одних программистов, например, показательно Фурье-преобразование, другие же в своей практике могут и вовсе не сталкиваться с вещественной арифметикой. Нижеследующий выбор заведомо нерепрезентативен, но... определенную пищу для размышлений он все-таки дает.

Методики оценки качества машинной оптимизации

Задача оценки качества кодогенерации намного сложнее, чем может показаться на первый взгляд. Прежде всего, следует разделять, собственно, сам компилятор и его окружение (среду, библиотеки и т. д.). В частности совершенно некорректно сравнивать размер откомпилированного примера "Hello, World" с его ассемблерной реализацией. Вызов `'printf("xxx");'` компилятор транслирует приблизительно в следующий код: `"push offset xxx\call printf\pop eax"` – круче уже не оптимизируешь!

Обратите внимание на размер объектного файла, сформированного компилятором. Не правда ли, он мало в чем не уступает объектному файлу ассемблерной реализации? Конечно, после подключения всех необходимых библиотек, размер откомпилированного файла увеличивается в десятки раз, в то время как объем ассемблерного модуля практически не изменяется. Да, это так, но при чем здесь компилятор?! Ему встретился вызов printf, – он его послушно включил в объектный файл. Если бы программист захотел вывести строку напрямую через соответствующую API функцию операционной системы (как он поступил в ассемблерной реализации), исполняемый файл сразу бы похудел на десяток-другой килобайт. Что еще остается? Ах да, ***среда***, называемая так же RTL (***R**un **T**ime **L**ibrary* – библиотека времени исполнения) – служебные функции, вызываемые самим компилятором. Несмотря на то, что библиотеки времени исполнения являются неотъемлемым компонентом компилятора, к качеству ***кодогенерации*** они не имеют никакого отношения, т. к. с точки зрения компилятора функции RTL ничем не отличаются от обычных библиотечных функций.

Избыточность штатных библиотечных функций и библиотеки времени исполнения на крохотных проектах очевидна, – вывод строки "Hello, World" не использует и сотой доли возможностей функции printf, но в программе, состоящей из нескольких тысяч строк, соотношение между полезными и служебными функциями нормализуется и коэффициент полезного действия библиотек практически вплотную приближается к единице.

Таким образом, сравнивать эффективность компилятора и ассемблера на примере библиотечных функций – это вопиющая некорректность. Следует рассматривать лишь чистые реализации, не обращающиеся к внешнему коду. В противном случае, будет сравниваться не качество машинной и ручной оптимизации, а совершенство библиотек (написанных, кстати, в большинстве своем на ассемблере) с ручной оптимизацией. Совершенно бесполезно сравнивать и размеры объективных файлов, – помимо кода они содержат массу посторонней информации, причем в рассматриваемых ниже примерах ее объем превышает размер машинного кода в десятки раз!

Истинную картину вещей дает лишь дизассемблер, – загружаем в него объективный или исполняемый файл, без разницы, и от адреса конца функции вычитаем адрес ее начала. Это и будет подлинный размер тела функции.

Измерять же производительность еще проще – достаточно засечь время выполнения функции и... Правда тут есть одно "но". Если уж мы взялись оценивать именно качество кодогенерации, а не быстродействие компьютера следует учесть и по возможности свести к нулю все посторонние эффекты. Во-первых, к моменту вызова функции все, обрабатываемые ей данные, должны находиться в кэше первого уровня, иначе неповоротливость памяти сотрет все различия в производительности тестируемого кода. Во-вторых, размер обрабатываемых данных должен быть достаточно велик для того, чтобы компенсировать накладные расходы на вызов функции, передачу аргументов, снятие показаний со счетчика производительности и т. д. Все нижеследующие примеры обрабатывают 4.000 элементов типа int, – это дает стабильный и воспроизводимый результат, т. к. "насыщение" наступает уже на 1.000 элементах, после чего накладные расходы уже не играют сколь ни будь заметной роли.

Сравнительный анализ основных компиляторов

Итак, давайте рассмотрим три довольно популярных алгоритма: ***копирование*** блока памяти, ***поиск*** минимума среди множества чисел и пузырьковую ***сортировку***. Такой выбор не случаен. *Операции копирования* (равно как сравнения и поиска в памяти) программисты всегда предпочитали реализовывать ассемблере, поскольку микропроцессоры Intel 80x86 поддерживают специальные машинные инструкции, изначально ориентированные на эти цели. В частности, копирование памяти осуществляется командой REP MOVS – своеобразным аналогом функции memcpy. К сожалению, эквивалентные ей (команде REP MOVS) конструкции в языках Си/Си++ отсутствуют. Можно, конечно, вызвать саму memcpy, но это будет нечестно – мы же договорились библиотечные функции не использовать! (тем более что memcpy за редкими исключениями пишется отнюдь не на Си, а на ассемблере). На уровне чистого языка существует лишь один путь решения данной задачи – поэлементное копирование массива в цикле. Проблема в том, что компиляторы пока еще не научились понимать физический смысл компилируемой программы, и даже такой очевидный алгоритм копирования ни один из известных мне оптимизаторов ни в жизнь не распознает. Компилятор дословно переведет программу на язык

машинного кода, сохранив при этом и сам цикл, и все временные переменные, используемые для пересылки данных. Как следствие – полученный код будет далеко не самым оптимальным, проигрывая ассемблеру и по скорости, и по размеру, да и по времени, затраченному на его создание, кстати. Поэтому, очень интересно знать: насколько эффективной окажется компиляция заведомо неоптимального кода. Данный пример позволяет оценить целесообразность использования ассемблера для решения задач, имеющих аппаратную поддержку со стороны процессора, в отсутствии эквивалентных конструкций в языке высокого уровня.

Поиск минимума – достаточно тривиальный алгоритм, элементарно укладывающийся в несколько строк и на ассемблере, и на языке высокого уровня. Столь малое пространство не дает оптимизатору развернуться и показать себя во все красе. Да это нам, собственно, и не нужно! Напротив, представляет интерес установить: какое количество избыточного кода воткнул сюда компилятор! Благодаря тому, что количество полезного кода в нем очень невелико, такой вырожденный пример становится весьма чувствительным даже к микроскопическим порциям "мусора".

Наконец, *сортировка* представляет собой довольно типичный пример программистского кода, и по ней вполне можно судить о "средневзвешенном" качестве оптимизации конкретных компиляторов.

Сравнивать различные компиляторы между собой – проще всего. Сравнение же компилятора с человеком – значительно сложнее, поскольку возникает неопределенность: с каким именно человеком его сравнивать. С профессионалом экстра класса? Но будет ли показательным такое сравнение? Мы же говорим не о теоретическом превосходстве человеческого интеллекта над машинным, а о практических путях решения задач, стоящих перед рядовыми программистами. Может ли рядовой программист рассчитывать на помощь экстра профессионала? Вряд ли! Скорее всего, засучив рукава, ему придется взяться за кодирование самому.

Поэтому, приведенные ниже ассемблерные примеры умышленно составлены как средний по качеству, но далеко не идеальный код. Причем, поскольку целевой процессор заранее не оговорен, при их подготовке использовались лишь самые общие приемы оптимизации, без учета оптимального планирования потока команд. Тем не менее, это действительно оптимизированный ассемблерный код, приблизительно соответствующий уровню программиста средней руки. (Кстати, интересно: кто из читателей сможет улучить качество кода хотя бы на процент?).

Хорошо, с человеком мы разобрались. Теперь дело – за компиляторами. Какие же из них выбрать? Давайте остановимся на следующем "джентльменском наборе": Microsoft Visual C++ 6.0, Borland C++ 5.5, WATCOM C++ 10.0.

Обсуждение результатов тестирования

Итак, тестирование началось... Прогон "подопытных" примеров на процессорах Intel Pentium-III 733 и AMD Athlon 1.400 (см. рис. 1, рис. 2) говорит о достаточно высоком качестве кодогенерации современных компиляторов. В среднем (за вычетом особо оговариваемых исключений) производительность

откомпилированных программ лишь на 20%-30% уступает вручную оптимизированному ассемблеру. Конечно, это весьма внушительная величина (особенно, если вспомнить, что эталонная ассемблерная программа достаточно далека от идеала). Так, кто там говорил, что машинная оптимизация уступает человеку не более одного-двух процентов? А ну, подать сюда этого человека!

С другой стороны, разрыв производительности (за редкими исключениями) все же не столь велик, чтобы перенос программы на ассемблер привел к качественным изменениям.

А теперь обо всем этом подробнее. Как и следовало ожидать, наибольший разрыв в производительности наблюдается на *копировании памяти*, впрочем, этот разрыв значительно сокращается с ростом тактовой частоты процессора. Если на P-III 733 наименьшее отставание составило целых 25%, то на Athlon 1.400 – всего 9%! Едва ли последняя цифра нуждается в комментариях – Microsoft рулит и жизнь прекрасна. Быстрота современных процессоров, помноженная на мощь современных компиляторов – и никаких ассемблерных вставок. Конечно, не все компиляторы одинаково хороши. Так, WATCOM – вообще в осадке; Borland уверенно держит позиции на Intel, но генерирует несколько неоптимальный код с точки зрения AMD.

С *поиском минимума* все компиляторы справились одинаково хорошо, а Microsoft Visual C++ вообще построил идеальный по своей красоте код, лишь из-за досадной случайности не дотянувшийся до 100% результата, – начало цикла пришлось на наихудший с точки зрения микропроцессора адрес: 0x4013FF. "Благодаря" этому каждая итерация облагается несколькими штрафными тактами, что, в конечном счете, выливается во вполне весомые потери. Чаще всего, впрочем, судьба оказывается не столь жестока и код, сгенерированный компилятором, исполняется максимально эффективно. Однако, нет никаких гарантий, что даже малейшее изменение программы, (да, да, – даже выкидывание лишнего кода!), не ухудшит ее производительности (под час весьма значительно). Увы, в этом смысле компиляторы пока еще тупые до безобразия. Они либо вовсе не выравнивают переходы, либо выравнивают ***все*** переходы, что неоправданно увеличивает размер программы и нередко дает обратный эффект, многократно снижая ее производительность (если программа в результате такого распухания не поместится в кэш). Между тем, правильное решение – выравнивать лишь часто выполняемые переходы, в частности циклы, но, увы, ни в одном известном мне компиляторе это не реализовано.

Заметно лучше сложилась ситуация с *сортировкой*. Компилятор Microsoft Visual C++ отстает от ассемблерного кода всего лишь на 13%-14%. За ним с минимальным отрывом идет Borland C++ со своими 15% и 24% для Athlon 1.400 и P-III 733 соответственно. Последнее место занимает WATCOM, ни в чем не уступающий Borland'у на Pentium'е, но безапелляционно сдающий свои позиции на Athlon'е. Ну, не виноват же он, что создавался в ту далекую эпоху, когда и процессоры, и техника оптимизации были совсем другими? В общем, WATCOM неплохой, но сильно устаревший оптимизатор, и любовь к нему (у тех, у кого она имеется) не должна слепить глаза, – WATCOM'ом сегодня не самый лучший выбор.

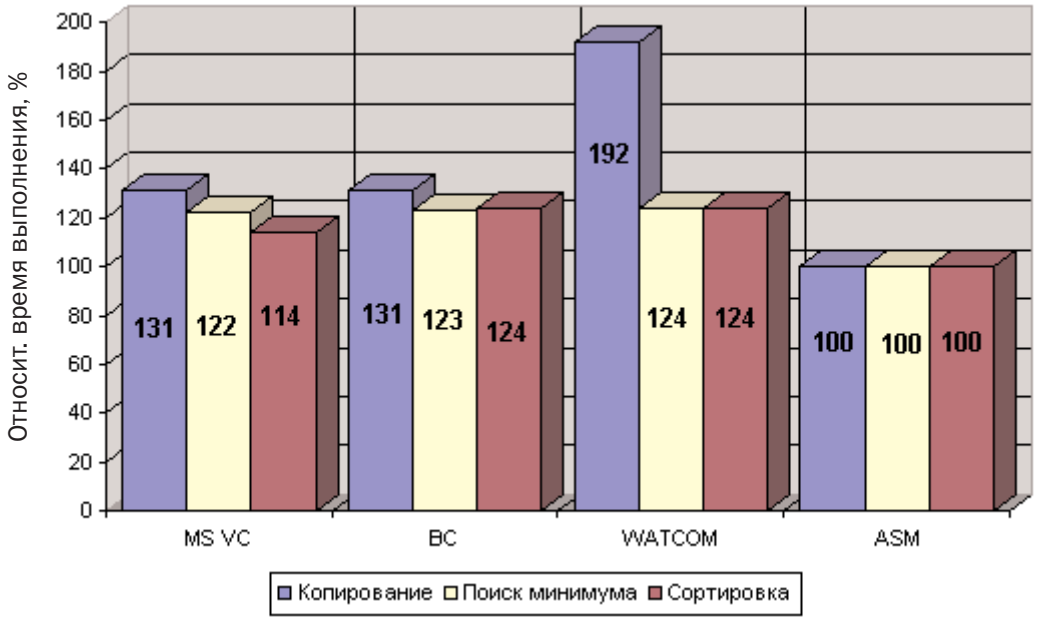

Рисунок 1. *Сравнение качества машинной кодогенерации по скорости на Intel P-III 733*

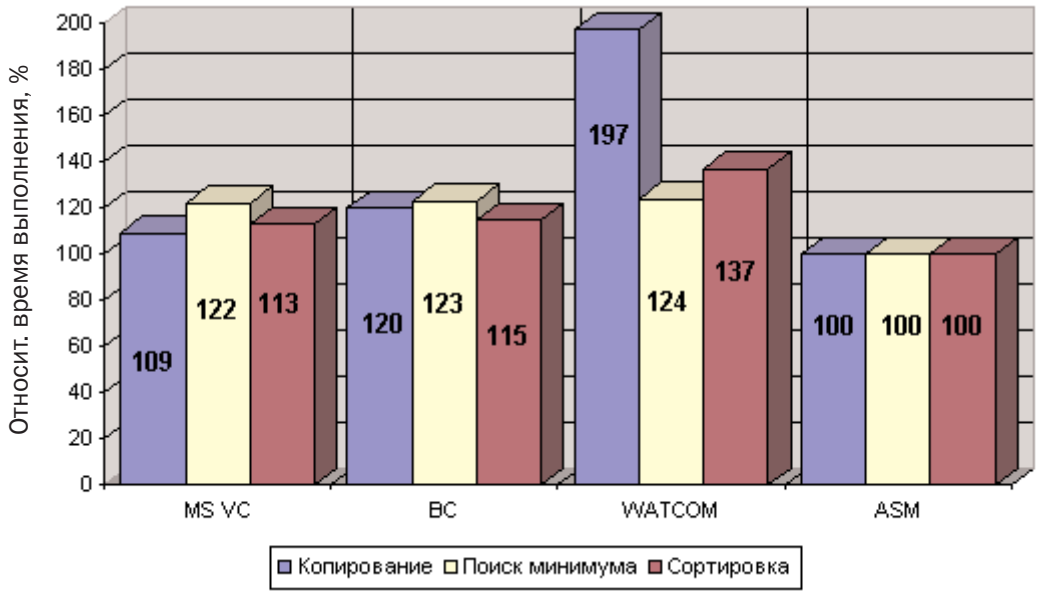

Рисунок 2. *Сравнение качества машинной кодогенрации по скорости на AMD Athlon-1.400*

Перейдем теперь к сравнению размера откомпилированного и ассемблерного кода. Первое, что бросается в глаза, – весьма внушительный отрыв Microsoft Visual C++ от своих конкурентов. Однако, и он отстает от "ручного" кода, не дотягивая по меньшей мере 23%, причем по мере упрощения задач этот разрыв резко увеличивается, достигая в случае примера с копированием памяти целых 76%! Ух, ты! Ассемблерная реализация оказалась практически вдвое короче!

У конкурентов же ситуация еще хуже. Значительно хуже. Грубо говоря, можно утверждать, что перенос программы на ассемблер как минимум сокращает ее размер раза в полтора-два. Тем не менее, два раза – это не триста и с этим вполне можно жить.

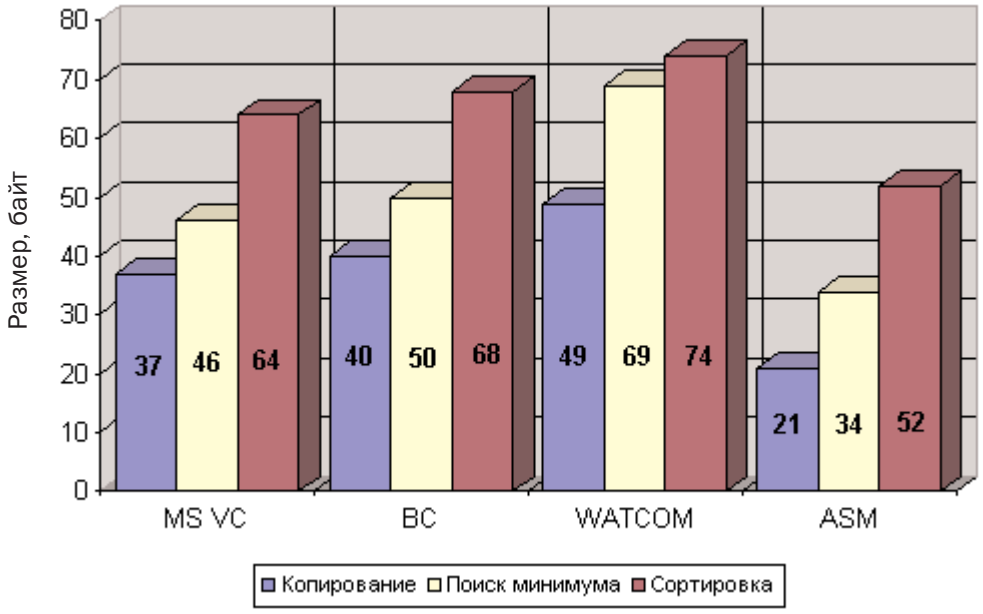

Рисунок 3. *Сравнение качества машинной кодогенерации по размеру*

Наглядная демонстрация качества машинной оптимизации

Разговор о качестве машинной оптимизации был бы неполным без конкретных примеров. Показатели производительности – слишком абстрактные величины. Они дают пищу для размышлений, но не объясняют: почему откомпилированный код оказался хуже. Как говорится, пока не увижу своими глазами, пока не пощупаю своими руками – не поверю!

Что ж, такая возможность у нас есть! Давайте, изучим непосредственно сам ассемблерный код, сгенерированный компилятором. Для экономии бумажного пространства ниже приведен лишь один пример – результат компиляции программы пузырьковой сортировки компилятором Microsoft Visual C++. Пример довольно показательный, ибо ощутимо улучшить его качество, не вывихнув при этом себе мозги, практически невозможно. Да вы смотрите сами. Если не владеете ассемблером – не расстраивайтесь, в листинге присутствуют подробные комментарии (надеюсь, вы понимаете, что их вставил не компилятор?).

```
.text:004013E0 ; ■■■■■■■■■■ S U B R O U T I N E ■■■■■■■■■■
.text:004013E0
.text:004013E0
.text:004013E0    c_sort        proc near             ; CODE XREF: sub_401420+DAp
.text:004013E0
.text:004013E0    arg_0         = dword ptr  8
.text:004013E0    arg_4         = dword ptr  0Ch
.text:004013E0
.text:004013E0        push        ebx
.text:004013E0   ; сохраняем регистр EBX, т.к. функция обязана сохранять
.text:004013E0   ; модифицируемые регистры, иначе программа рухнет
.text:004013E0   ;
.text:004013E1        mov         ebx, [esp+arg_4]
.text:004013E1   ; загружаем в bx самый правый аргумент - количество элементов
.text:004013E1   ; точно так поступил бы и человек
.text:004013E1   ; ага, локальные переменные адресуются непосредственно через ESP
.text:004013E1   ; это экономит один регистр (EBP), высвобождая его для других нужд че-
.text:004013E1   ; ловек так не умеет… вернее, не то, что бы совсем не умеет, но адре-
.text:004013E1   ; сация через ESP заставляет заново вычислять местоположение локальных
.text:004013E1   ; переменных при всяком перемещении верхушки стека (в частности при пе-
.text:004013E1   ; редаче функции аргументов) что ну очень утомительно…
.text:004013E1   ;
.text:004013E5        push        ebp
.text:004013E6        push        esi
.text:004013E6   ; сохраняем еще два регистра
.text:004013E6   ; вообще-то, человек мог воспользоваться и командой PUSHA, сохраняющей
.text:004013E6   ; все регистры общего назначения, что было бы намного короче, но в то
.text:004013E6   ; же время, увеличило бы потребности программы в стековом пространстве
.text:004013E6   ; и несколько снизило бы ее скорость
.text:004013E6   ;
.text:004013E7        cmp         ebx, 2
.text:004013E7   ; сравниваем значение аргумента n с константой 2
.text:004013E7   ;
.text:004013EA        push        edi
.text:004013EB        jl          short loc_40141B
.text:004013EB   ; а вот здесь пошла оптимизация кода под ранние процессоры P-P MMX
.text:004013EB   ; сравнение содержимого EBX и анализ результата сравнения разделен
.text:004013EB   ; командой сохранения регистра EDI. Дело в том, что P-P MMX могли спа-
.text:004013EB   ; ривать команды, если они имели зависимость по данным впрочем, в дан-
.text:004013EB   ; ном конкретном случае такая оптимизация излишня т.к. процессор пыта-
.text:004013EB   ; ется предсказать направление перехода еще до его реального исполнения.
.text:004013EB   ; Впрочем, перестановка команд - карман не тянет и никому не мешает
.text:004013EB   ;
.text:004013ED        mov         ebp, [esp+0Ch+arg_0]
.text:004013ED   ; загружаем в EBP значение кране левого аргумента - указателя на сор-
.text:004013ED   ; тируемый массив. Как уже сказано, так адресовать аргументы умеют только
.text:004013ED   ; компиляторы, человеку это не под силу
.text:004013ED   ;
.text:004013F1
.text:004013F1        loc_4013F1:                       ; CODE XREF: C_Sort+39j
.text:004013F1   ; цикл начинается с нечетного адреса. Плохо! это весьма пагубно сказы-
.text:004013F1   ; вается на производительность
.text:004013F1   ;
.text:004013F1        xor         esi, esi
.text:004013F1   ; очищаем регистр ESI, выполняя логического XOR над ним самим вот на это
.text:004013F1   ; человек - способен!
```

```
.text:004013F1  ;
.text:004013F3      cmp         ebx, 1
.text:004013F6      jle         short loc_40141B
.text:004013F6  ; эти две команды лишние.
.text:004013F6  ; Для человека очевидно, если EBX >= 2, то он всегда больше одного
.text:004013F6  ; А вот для компилятора это - вовсе не факт! (Ну темный он)
.text:004013F6  ; Дело в том, что он превратил возрастающий цикл for в убывающий цикл
.text:004013F6  ; do/while с пост условием /* убывающие циклы с постусловием реализуют-
.text:004013F6  ; ся на x86-процессорах намного более эффективно */ но для этого компи-
.text:004013F6  ; лятор должен быть уверен, что цикл исполняется хотя бы раз - вот он
.text:004013F6  ; и помещает в код дополнительную и абсолютно лишнюю в данном случае
.text:004013F6  ; проверку. Впрочем, она не отнимает много времени
.text:004013F6  ;
.text:004013F8      lea         eax, [ebp+4]
.text:004013F8  ; быстрое сложить EBP с 4 и записать результат в EAX об этом трюке
.text:004013F8  ; знают далеко не все программисты, и обычно выполняют данную задачу
.text:004013F8  ; в два этапа MOV EAX, EBX\ADD EAX, 4
.text:004013F8  ;
.text:004013FB      lea         edi, [ebx-1]
.text:004013FB  ; быстро вычесть из EBX единицу и записать результат в EDI
.text:004013FB  ;
.text:004013FE
.text:004013FE      loc_4013FE:                     ; CODE XREF: C_Sort+35j
.text:004013FE      mov         ecx, [eax-4]
.text:004013FE  ; Оп с! Команда, расположенная в начале цикла пересекает границу 0x10
.text:004013FE  ; байт, что приводит к ощутимым задержка исполнения.Да, забыл сказать,
.text:004013FE  ; эта инструкция загружает ячейку src[a-1] в регистр ECX
.text:004013FE  ;
.text:00401401      mov         edx, [eax]
.text:00401401  ; загружаем ячейку src[a] в регистр EDX
.text:00401401  ;
.text:00401403      cmp         ecx, edx
.text:00401403  ; сравниваем ECX (src[a-1]) и EDX (src[a]). вообще-то, это можно было ре-
.text:00401403  ; ализовать и короче CMP  ECX, [EAX], избавлюсь от команды MOV EDX, [EAX]
.text:00401403  ; однако, это ни к чему, т.к. значение [EAX] нам потребуется ниже при
.text:00401403  ; обмене переменными и эта команда все равно бы вылезла там.
.text:00401403  ;
.text:00401405      jle         short loc_401411
.text:00401405  ; переход на ветку loc_401411, если ECX <= EDX в противном случае -
.text:00401405  ; выполнить обмен ячеек
.text:00401405  ;
.text:00401407      mov         [eax-4],    edx
.text:0040140A      mov         [eax],      ecx
.text:0040140A  ; обмен ячейками. Вообще-то, можно было бы реализовать это и через XCHG,
.text:0040140A  ; что было бы на несколько байт короче, но у этой инструкции свои про-
.text:0040140A  ; блемы - не на всех процессорах она работает быстрее…
.text:0040140A  ;
.text:0040140C      mov         esi, 1
.text:0040140C  ; устанавливаем ESI (флаг f) в
.text:0040140C  ; человек мог бы сократить это на несколько байт, записав это так:
.text:0040140C  ; MOV   ESI,    ECX. Поскольку ECX > EDX, то ECX !=0, при условии, конечно,
.text:0040140C  ; что EDX >= 0. разумеется, компиляторам такое не по зубам, однако, это
.text:0040140C  ; алгоритмическая оптимизация и к качеству кодогенерации она не имеет
.text:0040140C  ; никакого отношения
.text:0040140C  ;
```

```
.text:00401411        loc_401411:                     ; CODE XREF: C_Sort+25j
.text:00401411            add         eax, 4
.text:00401411    ; увеличиваем EAX (a) на 4 (sizeof(int))
.text:00401411    ;
.text:00401414            dec         edi
.text:00401414    ; уменьшаем счетчик цикла на единицу (напоминаю, компилятор превратил наш
.text:00401414    ; for в убывающий цикл)
.text:00401414    ;
.text:00401415            jnz         short loc_4013FE
.text:00401415    ; переход к началу цикла, пока EDI не равен нулю некоторые человеки здесь
.text:00401415    ; лепят LOOP, который хоть и компактнее но исполняется значительно
.text:00401415    ; медленнее
.text:00401415    ;
.text:00401417            test        esi, esi
.text:00401417    ; проверка флага f на равенство нулю
.text:00401417    ;
.text:00401419            jnz         short loc_4013F1
.text:00401419    ; повторять цикл, пока f равен нулю
.text:0040141B
.text:0040141B        loc_40141B:
.text:0040141B            pop         edi
.text:0040141C            pop         esi
.text:0040141D            pop         ebp
.text:0040141E            pop         ebx
.text:0040141E    ; восстанавливаем все измененные регистры
.text:0040141E    ;
.text:0040141F            retn
.text:0040141F    ; выходим из функции
.text:0040141F        C_Sort      endp
.text:0040141F
```

Итак, какие у противников компиляторов есть претензии к качеству кода? Смогли бы они реализовать эту же процедуру хотя бы процентов на десять лучше? Согласитесь, здесь есть чему поучиться даже весьма квалифицированным программистам!

Определение ситуаций предпочтительного использования ассемблера

Итак, мы вплотную подошли к ответу на главный вопрос: в каких именно случаях обращение к ассемблеру целесообразно, а в каких – нет. Часто программист (даже высококвалифицированный!) обнаружив профилировщиком "узкие" места в программе, **автоматически** принимает решение о переносе соответствующих функций на ассемблер. А напрасно! Как мы уже убедились, разница в производительности между ручной и машинной оптимизацией в подавляющем большинстве случаев очень невелика. Очень может статься так, что улучшать уже нечего, за исключением мелких, "косметических" огрехов, результат работы компилятора идеален и никакие старания не увеличат производительность, более чем на 3%–5%. Печально, если это обстоятельство выясняется лишь ***после*** переноса одной или нескольких таких функций на ассемблер. Потрачено время, затрачены силы... и все это впустую. Обидно, да?

Прежде, чем приступать к ручной оптимизации не мешало бы выяснить: насколько не оптимален код, сгенерированный компилятором, и оценить имеющийся резерв производительности. Но не стоит бросаться в другую крайность и полагать, что компилятор ***всегда*** генерирует оптимальный или близкий к тому код. Отнюдь! Все зависит от того, насколько хорошо вычислительный алгоритм ложиться на конструкции выбранного языка высокого уровня. Некоторые задачи решаются одной машинной инструкцией, но целой группой команд языков высокого уровня. Наивно надеяться, что компилятор поймет физический смысл компилируемой программы и догадается заменить эту группу инструкций одной машинной командой. Нет! Он будет тупо транслировать каждую инструкцию в одну или (чаще всего) несколько машинных команд, со всеми вытекающими отсюда последствиями...

Итак, правило номер один. *Прежде, чем оптимизировать код, обязательно следует иметь надежно работающий не оптимизированный вариант*. Создание оптимизированного кода на ходу, по мере написания программы, невозможно! Такова уж специфика планирования команд – внесение даже малейших изменений в алгоритм в подавляющем большинстве случаев требует кардинальных переделок кода. Потому, приступайте к оптимизации только после тренировки на "кошках" – языке высокого уровня. Это поможет пояснить все неясности и темные места алгоритма. К тому же, при появлении ошибок в программе подозрение всегда падает именно на оптимизированные участки кода (оптимизированный код за редкими исключениями крайне ненагляден и чрезвычайно трудно читаем, потому его отладка – дело непростое), вот тут-то и спасает "отлаженная кошка". Если после замены оптимизированного кода не оптимизированные ошибки исчезнут, значит, и в самом деле виноват оптимизированный код. Ну а нет, – ищите их где-нибудь в другом месте.

Правило номер два. *Не путайте оптимизацию кода и ассемблерную реализацию*. Обнаружив профилировщиком узкие места в программе, не торопитесь переписывать их на ассемблер. Сначала убедитесь, что сделано все возможное для увеличения быстродействия кода в рамках языка высокого уровня. В частности, следует избавиться от прожорливых арифметических операций (особенно обращая внимание на целочисленное деление и взятие остатка), свести к минимуму ветвления, развернуть циклы с малым количеством итераций... в крайнем случае, попробуйте сменить компилятор (как было показано выше, качество компиляторов очень разниться друг от друга). Если же все равно останетесь недовольны результатом тогда...

Правило номер три. *Прежде чем переписывать программу на ассемблер изучите ассемблерный листинг компилятора на предмет оценки его совершенства*. Возможно, в неудовлетворительной производительности кода виноват не компилятор, а непосредственно сам процессор или подсистема памяти, например. Особенно это касается наукоемких приложений, жадных до математических расчетов и графических пакетов, нуждающихся в больших объемах памяти. Наивно думать, что перенос программы на ассемблер увеличит пропускную способность памяти или, скажем, заставит процессор вычислять синус угла быстрее. Получив ассемблерный листинг откомпилированной программы (для Microsoft Visual C++ это осуществляется

ключом /FA), бегло просмотрите его глазами на предмет поиска явных ляпов и откровенно глупых конструкций наподобие: "MOV EAX,[EBX]\MOV [EBX],EAX". Обычно гораздо проще не писать ассемблерную реализацию с чистого листа, а вычищать уже сгенерированный компилятором код. Это требует гораздо меньше времени, а результат дает ничуть не худший.

Правило номер четыре. *Если код, ассемблерный листинг, выданный компилятором идеален, а программа без видимых причин все равно исполняется медленно, не отчаивайтесь и загрузите ее в дизассемблер.* Как уже отмечалось выше, оптимизаторы крайне неаккуратно подходят к выравниванию переходов и кладут их куда глюк на душу положит. Наибольшая производительность достигается при выравнивании переходов по адресам, кратным шестнадцати, и будет уж совсем хорошо, если все тело цикла целиком поместиться в одну кэш-линейку (т. е. 32 байта). Впрочем, мы отвлеклись. Техника оптимизации машинного кода – тема совершенно другого разговора. Обратитесь к документации, распространяемой производителями процессоров – Intel и AMD.

Правило номер пять. *Если существующие команды процессора позволяют реализовать ваш алгоритм проще и эффективнее, – вот тогда уж действительно, тяпнув для храбрости пива, забросьте компилятор на полку и приступайте к ассемблерной реализации с чистого листа.* Однако, с такой ситуацией приходится встречаться крайне редко, к тому же не стоит забывать, что вы не на одиноком острове. Вокруг вас огромное количество высокопроизводительных тщательно отлаженных и великолепно оптимизированных библиотек. Так зачем же изобретать велосипед, если можно купить и готовый?

И, наконец, последнее правило номер шесть. *Если уж взялись писать на ассемблере, пишите максимально "красиво" и без излишнего трюкачества.* Да, недокументированные возможности, нетрадиционные стили программирования, "черная магия" – все это безумно интересно и увлекательно, но... плохо переносимо, непонятно окружающим (и даже самому себе после возвращения к исходнику-тексту десятилетней давности) и вообще несет в себе массу проблем. Автор этих строк неоднократно обжигался на своих же собственных трюках, самое обидное, что трюки эти были вызваны отнюдь не "производственной необходимостью", а... ну, скажем так, "любовью к искусству". За любовь же, как известно, приходится всегда платить. Не повторяете чужих ошибок! Не брезгуйте комментариями и непременно помещайте все ассемблерные функции в отдельный модуль. Никаких ассемблерных вставок – они, как и сам ассемблер, непереносимы и создают очень много проблем при портировании приложений на другие платформы.

Единственная предметная область, не только оправдывающая, но, прямо скажем, провоцирующая ассемблерные извращения, это защита программ. О чем мы и поговорим ниже...

Особое замечание о создании защитного кода на ассемблере

Защитные механизмы, бесспорно, предпочтительнее всего реализовывать на ассемблере, используя максимум трюков и извращений. Эффективность ассемблера здесь вторична, главное – максимально запутать взломщика. Компиляторы же генерируют достаточно предсказуемый код и почерк каждого из них профессиональным хакерам хорошо известен. Достоинства ассемблера в том, что он практически не ограничивает полет фантазии и позволяет воплощать в жизнь практически любые идеи. Полиморфный, шифрованный, самомодифицирующийся код, антиотладочные и антидизасемблерные приемы – ниша по праву принадлежащая ассемблеру. Целесообразность использования тех или иных защитных механизмов – тема другого разговора, здесь же мы будем обсуждать лишь пути их реализации.

Ассемблерные трюки – вообще большая тема, однако, следует различать трюк как таковой (оригинальная идея и/или нетрадиционный пример программирования) и недокументированные возможности процессора и операционной системы. Трюки, при грамотном подходе к ним, вполне безобидны и никаких проблем не создают. Классический пример трюка – расшифровка программы одноразовым блокнотом, возвращаемым функцией rand. Поскольку, функция rand всегда возвращает одну и ту же последовательность, она может использоваться для шифровки/расшифровки программы. Если дизассемблер не сумеет распознать rand в откомпилированной программе – хакер ногу сломит, пока догадается: как устроена и работает такая защита. Какие проблемы может создать этот трюк? Совершенно верно, – никаких. А вот пример "грязного хака", основанного на недокументированных возможностях: в Windows 95 регион адресного пространства от 0xC0000000 до 0xF0000000, хранящий низкоуровневые компоненты системы, свободно доступен прикладным приложениям, что очень облегчает борьбу с отладчиками и всякими мониторами. Правда, под Windows NT первая же попытка обращения к этой области приводит к генерации исключения с последующим закрытием приложения – нарушителя. В результате, конечный пользователь теряет возможность запускать защищенную программу под Windows NT. Вот за это многие и не любят ассемблер. Но, позвольте, разве ж ассемблер виноват? Не используйте недокументированных особенностей (а если уж совсем невтерпеж, то используйте их с умом) – и проблем ни у кого не будет!

В последнее время, кстати, наметилась устойчивая тенденция к отказу от ассемблера даже в защитных механизмах. Действительно, многие трюки замечательно реализуются и на языках высокого уровня. В частности, динамическую расшифровку кода (равно как и исполнение кода в стеке) можно реализовать и на чистом Си/Си++, достаточно лишь получить указатель на функцию (Си это позволяет), после чего с ее содержимым можно делать все, что угодно. И вовсе не обязательно для этого спускаться на уровень голого ассемблера. Так же, язык высокого уровня облегчает написание полиморфных генераторов и виртуальных машин (машина Тьюринга, сети Петри, стрелка Пирса и т. д.), единственное, что нельзя на нем реализовать – так это самомодифицирующийся код. Вернее, воз-

можно, но с жесткой привязкой к конкретному компилятору (ибо необходимо знать: как и во что транслируется каждая строка), а подобная практика – дурной тон. Привязываться ни к чему и ни когда не стоит, к тому же трудозатраты при создании самомодифицирующегося кода на языке высокого уровня намного выше, чем на ассемблере.

Тем не менее, возможность создания защит на чистых Си/Си++, многих хакеров старого поколения просто корежит – они и слышать об этом не хотят (автор, кстати, сам такой). Что поделаешь, традиции и привычки – штучки упрямые. Ну, красиво программирование на голом ассемблере, понимаете? А создание защит непосредственно в машинных кодах вызывает ничем не передаваемое удовлетворение по своему эмоциональному накалу сравнимое разве что с оргазмом.

Это – программирование ради программирования, нацеленное не на конечный результат, а на сам процесс его достижения. "*Для некоторых людей программирование является такой же внутренней потребностью, подобно тому, как коровы дают молоко, или писатели стремятся писать*". Николай Безруков.

Программирование на ассемблере как особый род искусства

Компьютер уже давно перестал быть машиной для небольшой горстки Избранных и с каждым днем он все стремительнее и стремительнее превращается в пылесос. Современные программисты, абстрагировавшись от "железа" и даже от самих вычислительных алгоритмов, видят перед собой мышь да визуальную панель с компонентами. Написать программу стало так же легко, как сварить пакетный суп. Конечно, свои положительные моменты в этом есть, но... есть определенная категория людей, для которой жизнь – ни на секунду не прекращающийся поиск и преодоление сложностей (ни слова про гамак и ласты!). Если задуматься: какую практическую ценность несет в себе покорение горных вершин? Гораздо комфортнее и с меньшим риском к ним можно добраться и на вертолете.

Увы! Чем легче достается, тем меньше удовлетворения оно приносит. Визуальное программирование слишком просто, чтобы быть по настоящему интересным. С другой стороны, чем выше уровень языка, тем больше приходится соблюдать предписаний и тем меньше возможность для самовыражения. А Художники отличаются от всех остальных тем, что в каждой работе передают свое видение мира, частицу своего "Я".

Интерес к ассемблеру, часто доходящий до фанатизма, как раз и объясняется тем, что ассемблер – лучшее средство пощупать железо компьютера руками, это превосходная арена для интеллектуальной борьбы, и, наконец, – великолепный способ с пользой и интересом скоротать свободное от работы время.

Существует огромное множество ассемблерных головоломок – от "написать программу на байт короче, чем у соседа", до "создать самообучающуюся шахматную игру, занимающую не более двух килобайт". На ассемблере пишутся многие "демки", на нем же создаются "крякимы" (в дословном переводе "взломай меня")...

Никто не спорит, что все, перечисленное выше, можно реализовать и на языках высокого уровня, причем за несравнимо более короткое время при не сильно худшей эффективности. Да! Можно! Но... неинтересно. Мы, комсомольцы, видите ли, без ласт и гамака любить не можем...

Поэтому (и это очень важно!), если вы встретите человека, беззаветно преданного ассемблеру и презирающего языки высокого уровня, не спешите ломать пальцы о клавиатуру, переубеждая его в обратном. Девять из десяти – ассемблер любит он не за достоинства, а, напротив, за отсутствие таковых (если понимать под "достоинствами" удобства цивилизации). Один из десяти – он просто выпендривается и программирует на ассемблере, чтобы продемонстрировать окружающим свою крутость. Тогда тем более не стоит его переубеждать – с возрастом само рассосется.

Заключение

Вердикт: ассемблер жил, ассемблер живет, ассемблер будет жить. Наблюдаемое засилье высокоуровневых языков и визуальных средств разработки – явление временное. Это затишье перед бурей. А буря грянет, можете не сомневаться. Не сегодня-завтра перед программистами встанут новые задачи, подчистую съедающие все вычислительные мощности и требующие еще. Главное – быть готовым к этому моменту и вовремя предложить свои знания, умения и навыки, дождавшись острой нехватки ассемблерных специалистов. (Мимоходом: программисты, знающие практически забытый ныне Фортран, с руками отрываются на Запад, ибо там половина научных приложений написана на Фортране, но некому их сопровождать – старые кадры уходят на пенсию, а новое поколение выбирает пеп**Си**).

Если же вы органически не приемлите наживу и бизнес – программируйте на ассемблере из спортивного интереса. Последние поколения Pentium'ов в этом отношении – просто клад и там, поверьте, есть чему поучиться!

Программируйте! Удачи вам и... побольше сложностей от жизни!

Приложение. Исходные тексты

```
void __cdecl c_cpy(int *src, int *dst, int n)
{
        int a; int t;

        if (n<1) return;        // нечего копировать!

                                // поэлементное копирование массива
        for (a=0;a<n;a++) dst[a]=src[a];

        return;
}


_asm_cpy proc

        PUSH ESI                        ; / *
        PUSH EDI                        ; сохраняем регистры
        PUSH ECX                        ; */

        MOV     ESI,[ESP+4+3*4]         ; src
        MOV     EDI,[ESP+8+3*4]         ; dst
        MOV     ECX,[ESP+8+4+3*4]       ; n

        REP MVSD                        ; копируем одной командой!

        POP ECX                         ; /*
        POP EDI                         ; восстанавливаем регистры
        POP ESI                         ; */

        ret                             ; выходим
_asm_cpy endp


int __cdecl c_min(int *src, int n)
{
        int a; int t;

        if (n<2) return -1;     // Не среди чего искать минимум!

                                // Присваиваем первому элементу массива статус
                                // "условно наименьшего"
        t=src[0];

                                // Если такой, кто будет меньше нашего "меньшего"?
                                // есть да, то предать статус ему.
        for(a=1;a<n;a++) if (t>src[a]) t=src[a];

        return t;
}
```

```
_asm_min    proc

        PUSH    ESI                     ; сохраняем
        PUSH    EDI                     ;              регистры

        MOV     ESI,[ESP+8+4]   ; src
        MOV     EDX,[ESP+8+8]   ; n

        CMP     EDX,2                   ; есть среди чего искать?
        JB      @exit                   ; нет элементов для поиска

        MOV     EAX, [ESI]              ; присваиваем первому элементу
                                        ; статус "условно наименьшего"
@for:                                   ; начало цикла
        MOV     EDI, [ESI]              ; в EDI - очередной элемент
        CMP     EAX, EDI                ; если кто еще меньше?
        JB      @next                   ; если нет, - следующий элемент
        MOV     EAX, EDI                ; передать статус

@next:
        ADD     ESI, 4                  ; перейти к следующему элементу
        DEC     EDX                     ; уменьшить счетчик цикла на 1
        JNZ     @for                    ; повторять цикл пока EDX > 0

@exit:
        POP     EDI                     ; восстанавливаем
        POP     ESI                     ;              регистры
        ret
_asm_min endp



void __cdecl c_sort(int *src, int n)
{
        int a; int t; int f;
        if (n<2)
        return;                         // Меньше двух элементов сортировать нельзя!

        do{
                f=0;                    // устанавливаем флаг сортировки в нуль

                                        // Перебираем все элементы один за другим
                for (a=1; a<n; a++)
                                        // если следующий элемент меньше предыдущего
                                        // меняем их местами и устанавливаем флаг
                                        // сортировки в единицу
                        if (src[a-1]>src[a])
                                {
                                        t=src[a-1];
                                        src[a-1]=src[a];
                                        src[a]=t;
                                        f=1;
                                }
                                // повторять сортировку до тех пор, пока не дождемся
                                // первого "чистого" прохода, т.е. прохода без изменений
        } while(f);
}
```

```
_asm_sort   proc
         MOV     EDX,[ESP+8]           ; n
         CMP     EDX,2
         JB      @exit

         PUSH    ESI
         PUSH    EBP
         PUSH    EBX

@while:
         MOV     ESI,[ESP+4+4*3]       ; src
         MOV     EDX,[ESP+8+4*3]       ; n
         XOR     EBP,EBP               ; EBP - flag
         NOP
         NOP
@for:
         MOV     EAX, [ESI]
         MOV     EBX, [ESI+4]

         CMP     EAX, EBX
         JAE     @next_for
         MOV     EBP, EBX
         MOV     [ESI+4], EAX
         MOV     [ESI],EBX

@next_for:
         ADD     ESI, 4
         DEC     EDX
         JNZ     @for

         OR      EBP,EBP
         JNZ     @while

         POP     EBX
         POP     EBP
         POP     ESI
@exit:

         ret
_asm_sort endp
```