

Естественный первый вопрос, который наверняка возникает у любого из вас – зачем изучать низкоуровневое программирование и всё, что с ним связано. Ведь в настоящее время существует большое множество распространенных языков программирования высокого уровня, готовых библиотек и прочее, прочее, которое избавляет от необходимости много думать. Чтобы в будущем не возвращаться к этому вопросу, я попробую ответить на него сейчас.

В первую очередь, конечно, не факт, что всё изученное в этом курсе достаточно отложится, ибо как бы мы ни разбираемся в вещах, без практики и повторения знания притупляются. Тем не менее, остается открытым вопрос, зачем во все это вникать.

Вообще-то, совсем не факт, что абсолютно всем программистам нужно знать ассемблер. Уверен, что веб-разработчиком он не нужен. Но, возможно, что некоторым программистам будет полезно знать хотя бы принципы работы процессора и внутреннего устройства современных ОС.

Подобные аргументы легко подкрепляются довольно известными личностями в мире программирования, причем как в России, так и за рубежом. Александр Степанов, русско-американский учёный в области информатики и вычислительной техники, который был топ-менеджером одних из крупнейших ИТ-компаний, как, например, AT&T (у них существует синтаксис ассемблера, который мы, возможно, затронем) и создатель STL, ставшей частью стандартной библиотеки языка C++ в предисловии ко 2-му изданию книги C++ и STL — справочное руководство (не его авторства) пишет следующее:

Для того чтобы быть хорошим программистом, важно понимать, что в действительности происходит «за кулисами» высокоуровневого языка программирования. Следует знать по крайней мере пару разных архитектур.

Джоэл Спольски, программист и писатель, известный на весь мир своим блогом, в котором обсуждает вопросы программирования, прежде всего под Windows, сооснователь крайне популярного и наверняка знакомого вам Stack Overflow, работавший в Microsoft как руководитель команды программистов, работавших над Excel пишет в своей книге Джоэл о программировании (Joel on Software):

Я думаю, что некоторые крупнейшие ошибки — даже на самых верхних уровнях архитектуры — происходят из-за слабого или неверного понимания некоторых простых вещей на самых нижних уровнях.

А теперь давайте перейдем непосредственно к аргументам. Первым, и наиболее важным аргументом, на мой взгляд, является следующий: вы понимаете, что вообще происходит. Невозможно объяснить рядовому пользователю Windows, что значит «убить процесс», потому что он просто не знает о существовании процессов. Рядовой юзер мыслит в контексте окон и ярлыков на рабочем столе.

Возникает аргумент даже в плане пользования операционными системами – человек, чаще всего пользующийся Windows, попав в мир Unix-подобных ОС и не зная, что находится у них «под капотом», будет пытаться решать любую проблему, как истинный пользователь Windows – путем поиска того пункта меню и той галочки, что решат проблему – и в этом мире такие способы решения не работают. Естественно, всё это сложнее, но открывает намного больше возможностей. А именно: поскольку UNIX разрабатывалась инженерами и для инженеров, в ее основу была положена концепция toolbox (ящик с инструментами). Что это значит? Это значит, что при создании софта и встроенных утилит для UNIX не делали универсальные программы, каждая из которых выполняла бы внутри себя все, необходимые пользователю действия, а для

каждой небольшой задачи создавалась своя утилита, которая выполняла свою задачу, только одну, но делала это хорошо. Дело пользователя было при помощи набора этих утилит выполнить операции, которые ему нужно сделать.

При этом из этого набора утилит можно составлять цепочки и последовательности действий, что позволяет легко автоматизировать рутинные, часто повторяющиеся операции.

Для того, чтобы утилиты могли обмениваться между собой результатами своей работы, в качестве носителя информации был выбран текстовый файл. Для обмена информацией между утилитами были изобретены "pipes" (трубы). При помощи "труб" информация с выхода одной команды может быть передана на вход второй, та ее обрабатывает, выдает свою информацию на выход, которая может быть передана на вход третьей и так далее.

В общем, в результате UNIX позволяет пользователю легко создавать простые программные комплексы, выполняющие повторяющиеся действия как по команде пользователя, так и в автономном режиме.

Такой подход имеет как плюсы, так и недостатки. С одной стороны он дает больший контроль над системой, гибкость в настройке, но при этом повышается порог вхождения в систему, или говоря простыми словами, прежде, чем что-нибудь сделать, как правило, нужно изучить основы.

С другой стороны, в Windows доминирует другая концепция. Эта концепция — максимально облегчить вхождение пользователя в задачу. Программы в Windows как правило большие, на каждое действие есть пункт в меню или иконка.

Ухудшает ситуацию о построении комплексов на базе Windows то, что большинство программ — коммерческие и используют свои, бинарные и как правило закрытые форматы данных и файлов. Такой подход превращает компьютер в устройство, которое может выполнять ограниченный изготовителем ПО набор функций, в пределе в этакый своеобразный "тостер", который выполняет только то, что задумал его изготовитель.

Конечно, в обеих системах не доминирует свой подход на 100 процентов. Как в Windows есть возможность пользоваться текстовой консолью и создавать .bat файлы, так и в UNIX есть большой набор программ, со свойствами присущими скорее "тостерному" подходу. И все таки описанная разница в подходах есть и она достаточно ярко выражена.

Понимание важно не только в отношении операционных систем — без него в том числе невозможно осознать, какие алгоритмы и методы в написании программ более производительны и эффективны других, причем речь идет в том числе и о крупных вещах вроде баз данных. Также, как следствие из предыдущего пункта, понимая, что происходит, вы совершаете меньше ошибок. Невозможно избежать ошибок переполнения буфера, целочисленного переполнения, не понимая происходящего. Не зная мат части, невозможно обнаружить ошибку в следующем коде:

```
void doSomethingUseful(float a, float b) {  
    if(a == b) {  
        // сделать что-то ...  
    }  
    // сделать что-то ещё ...  
}
```

Из-за ошибок округления большинство чисел с плавающей запятой оказываются неточными. Пока эта неточность остается небольшой, ее обычно можно игнорировать.

Однако это также означает, что числа, которые, как мы ожидаем, должны быть равными (например, при вычислении одного и того же результата разными правильными методами), часто немного отличаются, и простой тест на равенство не проходит. Например:

```
float a = 0.15 + 0.15
float b = 0.1 + 0.2
if(a == b) // может быть false
if(a >= b) // тоже может быть false
```

Решение состоит не в том, чтобы проверить, совпадают ли числа в точности, а в том, является ли их разница допустимо малой. Погрешность, с которой сравнивается разница, часто называется эпсилон. Самый простой вариант:

```
if( Math.abs(a-b) < 0.00001) // так делать не стоит
```

Это плохой вариант, потому что фиксированный эпсилон, выбранный из-за того, что он «выглядит маленьким», на самом деле может быть слишком большим, когда сравниваемые числа очень малы. Сравнение вернет «истину» для совершенно разных чисел. А когда числа очень большие, эпсилон может оказаться меньше наименьшей ошибки округления, так что сравнение всегда возвращает «ложь». Следовательно, необходимо посмотреть, меньше ли относительная погрешность, чем эпсилон:

```
if( Math.abs((a-b)/b) < 0.00001 ) // всё ещё неправильно!
```

Есть несколько особых случаев, когда такой вариант не работает:

Когда и *a*, и *b* равны нулю. 0,0 / 0,0 - это «not a number», что вызывает исключение на некоторых платформах или возвращает false для всех сравнений.

Когда только *b* равно нулю, деление дает «бесконечность», которая также может вызвать исключение, или быть больше эпсилона при сравнении.

Также такой вариант возвращает false, если и *a*, и *b* очень малы, но находятся на противоположных сторонах от нуля, даже если они являются наименьшими возможными ненулевыми числами.

Кроме того, результат не коммутативен (`nearlyEquals(a, b)` не всегда равен `nearlyEquals(b, a)`). Чтобы решить эти проблемы, код должен быть намного сложнее.

Знание различных низкоуровневых вещей существенно упрощают отладку приложений. Два программиста отлаживают одну программу. Один из них знаком с ассемблером, потому что он загоняет прогу в OllyDbg (под UNIX есть хороший аналог — kgdb), ставит точки останова, смотрит, что собственно происходит и быстро находит ошибку. Второй с ассемблером не знаком, потому что лучшее, что он может сделать — это посмотреть значения переменных во время выполнения программы.

Рано или поздно оба программиста понимают, что что-то не так в библиотеке стороннего производителя, предназначенной для работы с протоколом FTP. Один из программистов знает этот протокол и умеет пользоваться `tcpdump`, а второй нет...

Следующий аргумент — оптимизация программ. Да, переписывание программы с C++ на ассемблер в наши дни, скорее всего, не даст ощутимого прироста производительности, если только речь не идет о каком-нибудь сжатии или шифровании данных. Зато переписывание с «высокоуровневого» Python на «низкоуровневый» C++ может весьма существенно ускорить программу. Не говоря уже о том, что переписанная программа будет кушать на порядок меньше памяти.

Помимо возможности переписать программу, знание основ, как уже было сказано, поможет вам выбрать более эффективный алгоритм. Например, в криптографии эллиптические кривые намного быстрее алгоритма RSA. Вы можете сколько угодно

пытаться оптимизировать последний, но он все равно не догонит ЭК в плане производительности. Это как пытаться разогнать пузырьковую сортировку до уровня quicksort.

Изучая основы, вы открываете для себя множество новых возможностей, помимо уже названных. Возьмем к примеру перехват Windows API. Перехват системной функции API заключается в изменении некоторого адреса в памяти процесса или некоторого кода в теле функции таким образом, чтобы при вызове этой самой API-функции управление передавалось не ей, а вашей функции, подменяющей системную. Эта функция, работая вместо системной, выполняет какие-то запланированные вами действия, и затем, в зависимости от вашего желания, либо вызывает оригинальную функцию, либо не вызывает ее вообще. Казалось бы, нигде, кроме как в вирусах, это не нужно. Перехват функций является очень полезным средством в том случае, если вы хотите отследить, изменить или заблокировать некоторые конкретные действия приложения. И продолжает так казаться до тех пор, пока вам не захочется написать свой отладчик. Или антивирус. Или программу, создающую виртуальный жесткий диск. Или сделать portable версию какого-нибудь приложения не своего производства. Сколь бы высокоуровневой ни была Java, тут она вам не поможет.

Но труднее всего, как мне кажется, переоценить наличие опыта разработки собственного транслятора или интерпретатора. Тут открывается большое количество новых возможностей. Мы можем реализовать язык сценариев для своего приложения, что, скажем, в игровых движках — must have. Можем написать препроцессор для C++, добавляющий в язык синтаксический сахар (не замедляющие программу компоненты, которые более удобны для пользователя) для работы с юникодом, умными указателями и регулярными выражениями. Можем написать свой транслятор языка Си в хитрый байт-код для виртуальной машины, с целью усложнить реверсинг.

Смысл всего вышесказанного в том, что никакие высокоуровневые языки и готовые библиотеки не освобождают вас от необходимости знания мат части. Или, по крайней мере, ее основ. Уже упомянутый Джоэл Спольски называет это «законом дырявых абстракций». Мы можем сколько угодно абстрагироваться от машинного кода, программируя на C#. Или от битиков на жестком диске при работе с файловой системой. Но этот закон все равно тянет нас вниз, ибо в идеале абстракция полностью защищает вышележащий уровень от деталей реализации нижележащего, но идеала редко можно добиться.

Первый вопрос, который задает себе человек, впервые услышавший об ассемблере, - а зачем он, собственно, нужен? Особенно теперь, когда все пишут на различных языках высокого уровня? Действительно очень многое можно создать на том же Питоне, но ни один язык, даже такой популярный, не может претендовать на то, чтобы на нем можно было написать абсолютно все.

Итак, на ассемблере пишут: все, что требует максимальной скорости выполнения: ядра операционных систем реального времени и просто критические участки программ; все, что взаимодействует с внешними устройствами: драйверы, программы, работающие напрямую с портами, звуковыми и видеоплатами; все, что использует полностью возможности процессора: ядра многозадачных операционных систем и вообще любые программы, переводящие процессор в защищенный режим;

все, что полностью использует возможности операционной системы: вирусы и антивирусы, защиты от несанкционированного доступа, программы, обходящие эти защиты, и программы, защищающиеся от данных программ; и многое другое.

Часто, разумеется, выдвигаются мотивы того, что ассемблер на самом деле не очень-то и нужен.

Говорят, что программы на ассемблере трудно понять. Разумеется, на ассемблере легко написать неудобочитаемую программу... точно так же, как и на любом другом языке! Если вы знаете язык и если автор программы не старался ее запутать, то понять программу будет не сложнее, чем если бы она была написана на любом другом языке.

Говорят, что программы на ассемблере трудно отлаживать. Программы на ассемблере легко отлаживать - опять же при условии, что вы знаете язык. Более того, знание ассемблера часто помогает отлаживать программы на других языках, потому что оно дает представление о том, как на самом деле функционирует компьютер и что происходит при выполнении команд языка высокого уровня.

Говорят, что писать на ассемблере сложно. В этом есть доля правды. Очень часто авторы программ на ассемблере «изобретают велосипеды», программируя заново элементарные процедуры типа форматированного вывода на экран или генератора случайных чисел, в то время как программисты на С просто вызывают стандартные функции. Библиотеки таких функций существуют и для ассемблера, но они не стандартизированы и не распространяются вместе с компиляторами.

В заключение. Большинство популярных языков программирования — это языки высокого уровня. Например, Java, Python или С#. Конечно, программировать на них можно и слабо представляя, как на самом деле работают различные уровни абстракции. Но для хорошего IT-специалиста важно понимать и то, как устроена платформа, с которой он взаимодействует, как функционирует среда разработки, компилятор, отладчик. Это приводит нас к разговору о низкоуровневом программировании.

LLP (low-level programming) — это удобный способ разобраться с тем, как писать надёжные, быстрые, компактные и эффективные программы. Изучение ассемблера и Си полезно прежде всего, в том числе прежде более высокоуровневых языков, ради того, чтобы лучше почувствовать, как работает ПО.

IT-индустрия быстро развивается и подвержена моде на технологии и языки. Фундаментальные концепции — например, модели вычислений, виртуальная память, компиляция и сборка программ — изменяются гораздо реже. Понимание того, как базовые принципы таких концепций проявляются в одном языке программирования, поможет быстро освоить похожий язык, если это потребуется для работы. LLP в этом случае дает понять, как программа из исходного текста становится набором машинных инструкций, и как они выполняются на компьютере. В свою очередь знакомство с механизмами управления памятью и принципами работы компилятора поможет писать менее требовательный к вычислительным ресурсам код.

Уточнение: Нам достаточно часто приходится встречаться с тезисом о том, что LLP — это только программирование железа и тому подобные задачи. Это не так.

Где можно встретить LLP на практике:

Трейдинг. Торги на бирже совершают программы с невероятно высокими требованиями к времени реакции на колебания рынка. За счёт «удалённости» языков высокого уровня от «железа», очень сложно предсказать время отклика программы на внешнее событие.

Поэтому здесь задействуется LLP, чтобы обеспечить контроль над скоростью транзакций.

Робототехника. Вычислительные ресурсы здесь ограничены, поэтому высоки требования к качеству ПО и компактности кода. Общепринятые подходы создания ПО тут не всегда работают и зачастую разработка может идти на Си или другом системном языке.

Системное ПО. Это — ОС, компиляторы (JIT, AOT), браузеры (сегодня они стали платформой для создания сложных приложений). Его работа не всегда заметна обывателю, но создаёт инфраструктуру для прикладного ПО.

Для чего ещё полезно низкоуровневое программирование?

Обратная разработка (обратное проектирование, обратный инжиниринг, реверс-инжиниринг; англ. reverse engineering) — исследование некоторого готового устройства или программы, а также документации на него с целью понять принцип его работы; например, чтобы обнаружить недокументированные возможности (в том числе программные закладки), сделать изменение или воспроизвести устройство, программу или иной объект с аналогичными функциями, но без прямого копирования.

Применяется обычно в том случае, если создатель оригинального объекта не предоставил информации о структуре и способе создания (производства) объекта.

Исследование и обратная разработка программ обычно осуществляются с целью дальнейшей модификации, копирования, или, например, написания генераторов ключей, алгоритм работы которых получен на основе анализа алгоритма их проверки. Также исследование программ применяется с целью получения некоторых закрытых сведений о внутреннем устройстве программы — о протоколе сетевого обмена с сервером, аппаратным средством, ключом защиты или о взаимодействии с другой программой. Ещё одна область применения — получение информации о способах экспортирования данных из многочисленных проприетарных форматов файлов.

С развитием Интернета популярные операционные системы и программы всё интенсивнее исследуются на предмет обнаружения в них уязвимостей или т. н. «дыр». В дальнейшем найденные дыры могут использоваться для получения несанкционированного доступа к удалённому компьютеру или компьютерной сети. С другой стороны, обратная разработка применяется при исследовании антивирусными компаниями вредоносного ПО с целью добавления его сигнатур в базы своих продуктов.

В настоящее время под словами «reverse engineering» чаще всего понимается т. н. clean room reverse engineering, то есть процесс, при котором одна группа разработчиков анализирует машинный код программы, составляет алгоритм данной программы на псевдокоде либо, если программа является драйвером какого-либо устройства, составляет исчерпывающие спецификации интересующего устройства. После получения спецификаций другая группа разработчиков пишет собственный драйвер на основе полученных спецификаций или алгоритмов. Такой подход позволяет избежать обвинений в нарушении авторских прав на исходную программу, так как по законам, к примеру в США, попадает под понятие «fair use», то есть добросовестного использования оригинальной программы. Результат обратной разработки редко идентичен оригиналу, что и позволяет избежать ответственности перед законом, особенно при условии контроля отсутствия этой идентичности первой группой разработчиков и отсутствия нарушений торговых марок и патентов.

И, наконец, хоть это и было уже сказано, LLP используется для понимания сути работы всего, для того чтобы оптимизировать программы на скорость/размер/использование памяти, а также для поиска, эксплуатации и устранения уязвимостей в ПО.

История

Центральный процессор представляет из себя сложную интегральную схему, которая является одним из ключевых составляющих элементов современного ПК. Первые компьютеры появились примерно в 40-х годах прошлого века, работая на электромеханических реле и вакуумных лампах. Они обеспечивали функционирование первых вычислительных машин. В 60-х годах появились первые интегральные микросхемы которые на долгое время стали неотъемлемой частью любого вычислительного устройства. Началом эпохи современных CPU можно смело назвать 1971-й год.

Первый микропроцессор (процессор, реализованный в виде одной интегральной схеме) появился в 1971 г. Его создала фирма Intel, которая с тех пор и остаётся лидером в области разработки микропроцессоров. Этот процессор, работавший с 4-разрядными данными, фактически представлял собой микрокалькулятор. Первым однокристальным микропроцессором считается 4-битный Intel 4004, вышедший 15 ноября 1971 года. Компания Intel только начинала свой путь становления и ее создатели, Роберт Нойс, Гордон Мур и Эндрю Гроув, потратили немало сил на процесс развития. Благодаря вкладу итальянского физика Федерико Фаджина, инженерам компании удалось разместить ключевые компоненты на один чип и создать микропроцессор 4004.

Intel 4004 работал на частоте 108 кГц (проводил 92 600 операций в секунду). Объем памяти доходил до 4 Кб, разрядность шины — 4 бита.

В начале 70-х компания выпустила первый 8-битный центральный процессор Intel 8008. Он разрабатывался одновременно с 4004 под заказ для Computer Terminal Corporation (в последствии Datapoint). Но компания отказалась от CPU (как и от сотрудничества с Intel) из-за того, что процесс создания микросхемы вышел за пределы установленных сроков, а его характеристики не соответствовали ожиданиям. По техническим характеристикам микропроцессор 8008 во многом соответствовал предыдущей версии. Объем памяти был 16 Кб. Тактовая частота оказалась ниже, чем у 4004, она составляла 500 кГц. По скорости 8-битный процессор Intel отставал от 4-битного. Разрядность шины была 8-бит. Процессор мог обратиться к 8 портам ввода и 24 портам вывода. Одной из первых компьютерных систем на основе микропроцессора стал проект Sac State 8008 (1972 год). Это был полноценный микрокомпьютер с дисковой ОС, цветным дисплеем, ОЗУ 8 Кб и диском 3+2 Мб, клавиатурой, модемом, принтером. Он предназначался для обработки и хранения медицинских записей.

Весной 1974 г. фирма создала усовершенствованный микропроцессор 8080, работающий с 8-разрядными машинными словами и памятью до 64 Кб; это уже был настоящий центральный процессор универсальной ЭВМ, хотя и очень простой. Тактовая частота составляла 2 МГц (со временем 2,5 МГц, 3,1 МГц и 4 МГц). Разрядность шины данных составляла 8-бит, а шины адреса — 16-бит. У 8080 была весьма развита система команд: 16 команд передачи данных, 31 команда для их обработки, 28 команд для перехода (с прямой адресацией), 5 команд управления. В 1976 г. Появилась первая персональная ЭВМ (процессор плюс память и устройства ввода-вывода), разработанная фирмой Apple. За счет высокой производительности процессора пользовался успехом. На базе Intel 8080 компания MITS выпустила микрокомпьютер

Altair-8800. Несмотря на скромные характеристики (256 байт оперативной памяти, отсутствие монитора и клавиатуры) он обрел небывалую популярность и раскупался очень быстро.

В 1976 году компания Intel поставила перед своими инженерами серьезную задачу – создать первый в мире микропроцессор, поддерживающий работу в формате многозадачности, а также обладающий встроенным в чип контроллером памяти. Сейчас эти технологические особенности можно без труда обнаружить даже в самых доступных процессорах на рынке, но 45 лет назад подобные технологические новшества обещали перегнуть целую эпоху – Intel планировала перейти на 32-битные вычисления в тот момент, когда господствовали 8-битные системы, и даже 16-бит были очень далеко. К сожалению, или к счастью, амбиции руководителей Intel столкнулись с суровой реальностью в виде нескольких переносов сроков, технологическими проблемами и осознанием того, что технологии 1976 года еще не шагнули так далеко, чтобы воплотить в жизнь такие смелые задумки. А главное – Intel настолько увлеклась созданием, как сказали бы на западе, *over-engineered* архитектуры, что упустила из виду практичность с точки зрения программного обеспечения. Именно непрактичность и нарочитую усложненность системы раскритиковал на одном из совещаний приглашенный эксперт по имени Стивен Морс – 36-летний инженер микроэлектроники, специализирующийся тогда на программном обеспечении. Тем не менее, Intel не торопилась учитывать критические замечания, поэтому заметки Морса отправились в долгий ящик.

Но как оказалось позже, они были крайне полезными – уже в июле 1976 года небольшая компания Zilog, основанная изобретателем Intel 4004 и Intel 8008 Федерико Фаджином, а также менеджером Intel Ральфом Унгерманном и еще одним разработчиком 4004, японцем Масатоси Сима, представила на рынке свой процессор Z-80, ставший фактической работой над ошибками на базе Intel 8080.

Усовершенствовав архитектуру оригинального процессора Intel, команда Zilog предложила недорогой и производительный процессор, сразу же полюбившийся многим производителям техники и ведущих платформ того времени. Именно Z-80 лег в основу легендарного ZX Spectrum, а также был установлен в не менее известный Commodore 128 в качестве сопроцессора. Z-80 стал невероятно успешным во многих уголках мира, и этот успех не мог остаться незамеченным – в Intel срочно решили, что Z-80 нужен достойный конкурент.

Именно здесь руководители компании вспомнили о замечаниях Стивена Морса, и предложили тому возглавить создание принципиально нового процессора, призванного составить конкуренцию новинке от Zilog. Intel не видели особенных причин задавать рамки в этом проекте – тогда всем казалось, что новый процессор будет быстрым ответом на Z-80, и забудется в течение следующих лет, поэтому Морс получил зеленый свет на любые эксперименты. Именно навязчивая мысль о том, что процессор должен строиться вокруг эффективности работы с ПО, как выяснилось позже, стала ключевой для развития всей индустрии.

В мае 1976 года Стив Морс приступил к работе над архитектурой нового процессора. В сущности, задача, поставленная перед Морсом, была проста. Если новый 16-битный чип должен давать значительное увеличение скорости по сравнению с 8-битным 8080-м, он должен отличаться по ряду параметров. Но Intel хотела добиться, чтобы потребители обращались к ней повторно. Как один из способов добиться этого рассматривалась возможность перевода на более высокий уровень системы,

разработанной для менее мощного процессора, при замене которого новым она будет работать. Для этого, в идеале, новый процессор должен быть совместим с любой программой, написанной для 8080.

Морсу приходилось отталкиваться от проекта 8080, в соответствии с которым процессор назначал «адрес» каждому месту, где хранились числа, подобно ярлыкам классификатора. Адреса представляли собой 16-битные двоичные числа, что позволяло обозначить 65536 различных адресов. Этот потолок был приемлем, когда разработчикам требовалось экономно использовать память. Однако теперь потребителям понадобился больший объем, они настаивали на преодолении барьера в 64 Кбайт.

В июле 1978 года новый процессор, получивший название Intel 8086, появился на рынке. Его разработка велась более двух лет. Объем памяти достигал 1 Мб. Тактовая частота составляла 4 МГц — 10 МГц, разрядность регистров и шины данных была 16 бит, а разрядность шины адреса — 20 бит. Intel 8086 отличался скоростью работы. В процессоре 8086 насчитывалось четырнадцать 16-разрядных регистров: 4 общего назначения, 2 индексных регистра, 2 указательных, 4 сегментных регистра, программный счётчик или указатель команды и регистр флагов. Его выход не стал фурором или невероятным успехом. Впервые процессор попал на прилавки в составе нескольких бюджетных компьютеров, не пользующихся популярностью, а также использовался в различных терминалах. Чуть позже он лег в основу микроконтроллера NASA, где использовался для контроля над диагностическими системами ракетного пуска вплоть до начала 2000-х годов.

Морс покинул Intel в 1979 году, как раз перед тем, как компания представила Intel 8088, — практически идентичный 8086 микропроцессор, обеспечивавший совместимость с 8-битными системами через деление 16-битной шины на два цикла. Сам Морс назвал этот процессор «кастрированной» версией 8086.

Легендарный статус 8088 получил позже, когда в 1980 году IBM впервые задумалась о покорении рынка персональных компьютеров и создании компьютера, который был бы достаточно недорогим, и включал в себя комплектующие среднего класса. Именно IBM 5150, более известный под брендом IBM PC, и получил в основу процессор 8088 (по сути, все тот же 8086), благодаря чему Intel стала широко известной даже в кругах рядовых пользователей. А ведь на место 8088 претендовала и Motorola 68000 (основа первого Apple Macintosh), но руководство IBM отдала предпочтение Intel. IBM PC быстро превратился в главную силу на рынке компьютерных систем, и Intel, следуя логике «далее-лучше», продолжила выпускать процессоры — 80186, 80286, 80386, 80486, Pentium и так далее — на базе все той же основы Стивена Морса, заложенной им еще в 8086. Именно благодаря двум последним цифрам архитектура стала известна как «x86», а невероятная популярность компьютеров IBM обеспечила Intel огромные прибыли и узнаваемость в качестве бренда.

В плане архитектурных особенностей Intel 8086 во многом опирался на опыт разработки процессора 8080. Несмотря на некоторые параллели, 8086 стал первым 16-битным процессором компании, располагавшим 16 каналами данных и 20 адресными каналами, способными обрабатывать до 1 Мб данных, а также имел широкий набор инструкций, позволявших, среди всего прочего, проводить операции деления/умножения. Особенностью работы 8086 было наличие двух режимов — Минимального и Максимального, последний из которых предполагал использование процессора в

системе с несколькими процессорами, а первый – в классических системах с одним процессором.

В Intel 8086 впервые появилась очередь инструкций, позволяющая хранить до шести байт инструкций напрямую из памяти, значительно сокращая время на их обработку. 16-битная природа процессора не была основана лишь на нескольких компонентах, ведь 8086 составляли 16-битный ALU, 16-битные регистры, а также внутренняя и внешняя шина данных, обрабатывающие данные по 16-битным инструкциям, благодаря чему система работала значительно быстрее, чем с более ранними процессорами Intel.

Конечно, из-за такого масштабного набора инноваций 8086 был значительно дороже предшественника, но и в подобном ключе у потребителя был выбор — Intel предлагала купить новинку в нескольких вариантах, зависевших от частот процессора – они варьировались от 5 до 10 МГц.

С точки зрения архитектуры микропроцессор Intel 8086 состоял из двух аппаратных модулей – модуля выполнения и модуля интерфейса шины. Модуль выполнения указывал модулю интерфейса шины, откуда получать данные инструкций, а после этого приступал к их подготовке и выполнению. Его суть сводилась к управлению данными с помощью декодера инструкций и блока ALU, при этом сам модуль не имел прямого соединения с шинами данных, и работал исключительно через модуль интерфейса шины.

Модуль выполнения содержал блок АЛУ, предназначенный для выполнения логических и арифметических операций, таких как умножение, деление, сложение, вычитание или операции по типу OR, AND и NOT. Также здесь был 16-битный регистр флагов, хранивший различные состояния операций в аккумуляторе – всего их было 9. Помимо флагов модуль выполнения операций содержал 8 регистров общего назначения, которые использовались для передачи данных через шину в 16 бит. При этом сохранялась совместимость с предыдущим поколением программного обеспечения для 8-битных систем, потому что регистры общего назначения (AX, BX, CX, DX) могли работать как в режиме 16-битной шины, так и в режиме считывания данных с младших (AL, BL, CL, DL) и старших (AH, BH, CH, DH) регистров одновременно, обеспечивая двухканальную работу в формате 8-битной шины. Именно благодаря акценту на совместимость с предшествующими платформами с точки зрения программного обеспечения архитектура x86 стала ключевой и послужила основой для большинства последующих процессоров.

Наконец, последним из регистров в модуле стал 16-битный указательный регистр, который сохранял адрес сегмента данных в буфере памяти, необходимый для выполнения операции. Остальные функциональные части относились к соседнему модулю интерфейса шины.

Модуль интерфейса шины содержал в себе значительно больше функциональных компонентов – он отвечал на обработку всех данных и отправку инструкций в модуль выполнения, считывание адресов из памяти компьютера и информации со всех доступных портов ввода-вывода, а также за запись данных в доступную память и через вышеуказанные порты. Из-за того, что модуль выполнения не имел прямого соединения с модулем интерфейса шины, взаимодействие блоков происходило посредством внутренней шины данных.

В данном модуле содержится одна из ключевых архитектурных особенностей процессора 8086 – очередь инструкций. Модуль интерфейса шины включает очередь инструкций, способную хранить до 6 байт инструкций в буфере, отсылая новые

инструкции по конвейеру после того, как от модуля выполнения поступит соответствующий запрос. Термин *pipelining* появился именно с выходом на рынок процессора 8086, так как он означает подготовку следующей инструкции в момент, когда предыдущая находится в процессе выполнения.

Здесь же располагается 4 сегментных регистра, отвечающих за буферизацию адресов инструкций и сопутствующих им данных в памяти компьютера, и тем самым обеспечивающих доступ к нужным сегментам центральному процессору.

Наконец, последним из регистров является 16-битный указатель команды, содержащий адрес следующей для выполнения инструкции.

Стивен Морс, создавая концепцию небольшого «дочернего» процессора в стенах Intel, едва ли мог предположить, что находится на пороге создания исторического микропроцессора. Выход Intel 8086 был скромным и неоднозначным, однако его младший брат 8088 обрел славу в составе IBM PC/XT, позволив компании Intel обрести известность и получить колоссальную прибыль.

Архитектура x86 легла в основу всех дальнейших процессоров Intel, осознавшей удобство и универсальность концепции Морса «сначала ПО — потом начинка». Каждый следующий процессор строился на фундаменте предыдущего, обрастая новыми технологиями, инструкциями и блоками, но по своей сути немногим отличался от 8086.

И даже сегодня, глядя на i7 8086K, нужно понимать, что где-то глубоко внутри него еще находятся корни того самого процессора, увидевшего свет 40 лет назад, ознаменовавшего открытие эпохи x86.

В 1983 г. Фирма Intel разработала микропроцессор 80186. Объем памяти составлял 1 Мб, разрядность шины данных была 16-бит, а шины адреса — 20-бит. Тактовая частота достигала 6 МГц — 25 МГц.

Он практически не использовался, т.к. в том же году появился более совершенный микропроцессор 80286. На его основе IBM в 1984 г. построила свой очередной ПК — IBM AT (*advanced technology*). В процессоре 80286 предусмотрены аппаратные средства для реализации многозадачного режима работы ЭВМ: Объем оперативной памяти составлял 16 Мб, а в защищенном режиме можно было использовать до 1 Гб виртуальной памяти. Разрядность регистров и шины данных составляла 16-бит. В зависимости от модели, тактовая частота могла быть 6 МГц, 8 МГц, 10 МГц или 12,5 МГц (при 12,5 МГц процессор выполнял не менее 2,66 млн операций в секунду). Однако в целом возможности этого процессора оказались недостаточными для реального использования такого режима, поэтому процессор 80286 фактически представляет собой просто более быстрый вариант процессора 8086. В процессоре 80286 не была предусмотрена страничная организация памяти, которая используется современными ОС, в нём была поддержка сегментной модели памяти.

Реально защищённый режим начал использоваться только с появлением нового поколения микропроцессоров — 32-разрядных. В 1987 г. фирмой Intel был создан процессор i386 (сохранил обратную совместимость с 8086 и 80286, через страничное преобразование процессор мог адресовать до 4 Гб физической памяти и до 64 Гб виртуальной памяти, тактовая частота составляла 12 МГц — 40 МГц.), а в 1990 г. — процессор i486 (объем памяти составлял 4 Гб, тактовая частота была 25 МГц — 50 МГц.). Они могут работать в двух режимах — в реальном режиме, в котором они

фактически представляют собой очень быстрые варианты процессора 8086, и в защищённом режиме, позволяющем реализовать многозадачность.

В 1993 г. Фирма Intel разработала 64-разрядный микропроцессор, получивший собственное имя Pentium.

Все указанные процессоры объединяют в семейство 80x86, поскольку в них соблюдается преемственность: программа, написанная для младшей модели, может быть без каких-либо изменений выполнена на любой более старшей модели. Обеспечивается это тем, что в основе всех этих процессоров лежит система команд процессора 8086, в старшие же модели лишь добавляются новые команды (главным образом, необходимые для реализации многозадачного режима). Таким образом, процессор 8086 – это база, основа для изучения всех остальных моделей данного семейства.

Особенности работы ПК

Для того чтобы освоить программирование на ассемблере, следует познакомиться с двоичными и шестнадцатеричными числами. Иногда в тексте программы можно обойтись и обычными десятичными числами, но без понимания того, как на самом деле хранятся данные в памяти компьютера, очень трудно использовать логические и битовые операции, упакованные форматы данных и многое другое. Практически все существующие сейчас компьютерные системы, включая Intel, используют для вычислений двоичную систему счисления. В их электрических цепях напряжение может принимать два значения, и эти значения называли нулем и единицей. Двоичная система счисления как раз и использует только эти две цифры, а вместо степеней десяти, как в обычной десятичной системе, здесь применяют степени двойки. Чтобы перевести двоичное число в десятичное, надо сложить двойки в степенях, соответствующих позициям, где в двоичном стоят единицы. Для перевода десятичного числа в двоичное можно, например, разделить его на 2, записывая остаток справа налево. Чтобы отличать двоичные числа от десятичных, в ассемблерных программах в конце каждого двоичного числа ставят букву b. (binary)

Минимальная единица информации называется битом. Бит принимает только два значения - обычно 0 и 1. На самом деле они совершенно необязательны - один бит может принимать значения «да» и «нет», показывать присутствие и отсутствие жесткого диска, - важно лишь то, что бит имеет только два значения. Но многие величины принимают большее число значений, следовательно, для их описания нельзя обойтись одним битом.

Единица информации размером 8 бит называется байтом. Байт - это минимальный объем данных, который реально может использовать компьютерная программа. Даже для изменения значения одного бита в памяти надо сначала считать байт, содержащий его. Биты в байте нумеруют справа налево, от нуля до семи, нулевой бит часто называют младшим битом, а седьмой – старшим. Именно на байты делится ОЗУ компьютера.

Так как всего в байте восемь бит, он может принимать до $2^8 = 256$ разных значений. Байт используют для представления целых чисел от 0 до 255 (тип unsigned char в C), целых чисел со знаком от -128 до +127 (тип signed char в C), набора символов ASCII (тип char в C) или переменных, принимающих менее 256 значений, например для представления десятичных чисел от 0 до 99.

Следующий по размеру базовый тип данных - слово. Размер одного слова в процессорах Intel - два байта. Биты с 0 по 7 составляют младший байт слова, а биты с 8 по 15 - старший. В слове содержится 16 бит, а значит, оно может принимать до $2^{16} = 65\,536$ разных значений. Слова используют для представления целых чисел без знака со значениями 0-65 535 (тип unsigned short в C), целых чисел со знаком от -32 768 до +32 767 (тип short int в C), адресов сегментов и смещений при 16-битной адресации. Два слова подряд образуют двойное слово, состоящее из 32 бит, а два двойных слова - одно учетверенное слово (64 бита). Байты, слова и двойные слова - основные типы данных, с которыми мы будем работать.

Еще одно важное замечание: в компьютерах с процессорами Intel все данные хранятся так, что младший байт находится по младшему адресу, поэтому слова записываются задом наперед, то есть сначала (по младшему адресу) - последний (младший) байт, а потом (по старшему адресу) - первый (старший) байт. Если из программы всегда обращаться к слову как к слову, а к двойному слову как к двойному слову, это не оказывает никакого влияния. Но если вы хотите прочитать первый (старший) байт из слова в памяти, то придется увеличить адрес на 1. Двойные и учетверенные слова записываются так же - от младшего байта к старшему. Эта запись называется little-endian нотацией, которая противоположна big-endian - когда сначала записываются старшие разряды, а затем младшие, как это происходит в десятичных числах. Вопрос: зачем?

Главное неудобство двоичной системы счисления - это размеры чисел, с которыми приходится обращаться. На практике с двоичными числами работают, только если необходимо следить за значениями отдельных битов, а когда размеры переменных превышают хотя бы четыре бита, используется шестнадцатеричная система. Она хороша тем, что компактнее десятичной, и тем, что перевод в двоичную систему и обратно происходит очень легко. В шестнадцатеричной системе используется 16 «цифр» (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F), и номер позиции цифры в числе соответствует степени, в которую надо возвести число 16, следовательно:

$$96h = 9 \times 16 + 6 = 150$$

Перевод в двоичную систему и обратно осуществляется крайне просто - вместо каждой шестнадцатеричной цифры подставляют соответствующее четырехзначное двоичное число:

$$9h = 1001b, 6h = 0110b, 96h = 10010110b$$

В ассемблерных программах при записи чисел, начинающихся с A, B, C, D, E, F, в начале приписывается цифра 0, чтобы не перепутать такое число с названием переменной или другим идентификатором. После шестнадцатеричных чисел, по аналогии с двоичными, ставится буква h (hexadecimal).

Легко использовать байты, слова или двойные слова для представления целых положительных чисел - от 0 до 255, 65 535 или 4 294 967 295 соответственно. Чтобы применять те же самые байты или слова для представления отрицательных чисел, существует специальная операция, известная как дополнение до двух. Для изменения знака числа выполняют инверсию, то есть заменяют в двоичном представлении числа все единицы нулями и нули единицами, а затем прибавляют 1. Например, пусть используются переменные типа слова:

$$150 = 0096h = 0000\,0000\,1001\,0110b$$

инверсия дает: $1111\ 1111\ 0110\ 1001b + 1 = 1111\ 1111\ 0110\ 1001b = \text{OFF6Ah}$

Проверим, что число на самом деле -150: сумма с +150 должна быть равна нулю:

$+150 + (-150) = 0096h + \text{OFF6Ah} = 10000h$

Единица в 16-м разряде не помещается в слово, следовательно, мы действительно получили 0. В данном формате старший (7-й, 15-й, 31-й для байта, слова, двойного слова) бит всегда соответствует знаку числа: 0 - для положительных и 1 - для отрицательных. Таким образом, схема с использованием дополнения до двух выделяет для положительных и отрицательных чисел равные диапазоны: -128...+127 - для байта, -32 768...+32 767 - для слов, -2 147 483 648...+2 147 483 647 - для двойных слов.

Самые распространенные варианты значений, которые может принимать один бит, - это значения «правда» и «ложь», используемые в логике, откуда происходят так называемые «логические операции» над битами. Так, если объединить «правду» и «правду» — получится «правда», а если объединить «правду» и «ложь» - «правды» не получится. В ассемблере нам встретятся четыре основные операции - И (AND), ИЛИ (OR), «исключающее ИЛИ» (XOR) и отрицание (NOT). Все перечисленные операции являются побитовыми, поэтому для выполнения логического действия над числом надо перевести его в двоичный формат и произвести операцию над каждым битом, например:


$96h \text{ AND } 0Fh = 10010110b \text{ AND } 00001111b = 00000110b = 06h$

Для представления всех букв, цифр и знаков, появляющихся на экране компьютера, обычно используется всего один байт. Символы, соответствующие значениям от 0 до 127, то есть первой половине всех возможных значений байта, были стандартизированы и названы символами ASCII (хотя часто кодами ASCII именуют всю таблицу из 256 символов). Сюда входят некоторые управляющие коды (символ с кодом 0Dh - конец строки), знаки препинания, цифры (символы с кодами 30h - 39h), большие (41h - 5Ah) и маленькие (61h - 7 Ah) латинские буквы. Вторая половина символьных кодов используется для алфавитов других языков и псевдографики, набор и порядок символов в ней отличаются в разных странах и даже в пределах одной страны. Например, для букв одного только русского языка существует пять вариантов размещения во второй половине таблицы символов ASCII (см. приложение 1). Существует также стандарт, использующий слова для хранения кодов символов, известный как UNICODE или UCS-2, и даже двойные слова (UCS-4), но мы пока не будем на нем останавливаться.

Порядковый номер байта в памяти называется его адресом; память с точки зрения процессора есть последовательность байтов, каждому из которых присвоен адрес как его порядковый номер. Чтобы адресовать 1 МВ памяти (2^{20}), требуется диапазон адресов 00000: FFFFF. Байт – наименьшая адресуемая ячейка памяти. Есть и другие ячейки памяти – Слово (Word) – два байта и двойное слово DWORD - два слова. Для работы с данными в памяти недостаточно иметь только адрес участка памяти, так как с него может начитаться как байт, так и слово или двойное слово. Поэтому нужно учитывать тип, то есть размер операнда.

Помимо ячеек памяти для хранения данных (правда для кратковременного хранения) можно использовать и регистры – ячейки, расположенные прямо в центральном процессоре и доступные из машинных программ. Доступ к регистрам осуществляется

намного быстрее, чем к ячейкам памяти, поэтому использование регистров заметно уменьшает время выполнения программ.

 Все регистры имеют размер слова (8086), за каждым из них закреплено определённое имя (AX, Sp и тд). По назначению и способу использования регистры можно разбить на следующие группы:

- Регистры общего назначения (ax,dx,cx,bx,si,di,bp,sp)
- Сегментные регистры (cs,ds,ss,es)
- Указатель команд (IP)
- Регистр флагов (Flags)

Регистры общего назначения

Регистры AX (аккумулятор), BX (база), CX (счетчик), DX (регистр данных) могут использоваться без ограничений для любых целей - временного хранения данных, аргументов или результатов различных операций. Названия регистров происходят от того, что некоторые команды применяют их специальным образом: так, аккумулятор часто необходим для хранения результата действий, выполняемых над двумя операндами, регистр данных в этих случаях получает старшую часть результата, если он не умещается в аккумулятор, регистр-счетчик работает как счетчик в циклах и строковых операциях, а регистр-база - при так называемой адресации по базе. Отдельные байты в 16-битных регистрах AX - DX тоже могут использоваться как 8-битные регистры и иметь свои имена. Старшие байты этих регистров называются AH, BH, CH, DH, а младшие - AL, BL, CL, DL. Остальные четыре регистра - SI (индекс источника), DI (индекс приемника), BP (указатель базы), SP (указатель стека) - имеют более конкретное назначение и применяются для хранения всевозможных временных переменных. Регистры ESI и EDI необходимы в строковых операциях, EBP и ESP - при работе со стеком.

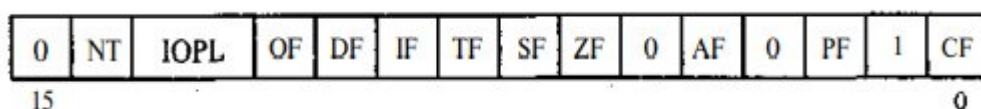
Сегментные регистры

При использовании сегментированных моделей памяти для формирования любого адреса нужны два числа - адрес начала сегмента и смещение искомого байта относительно этого начала (в бессегментной модели памяти flat адреса начал всех сегментов равны). Операционные системы (кроме DOS) могут размещать сегменты, с которыми работает программа пользователя, в разных местах памяти и даже временно записывать их на диск, если памяти не хватает. Так как сегменты способны оказаться где угодно, программа обращается к ним, применяя вместо настоящего адреса начала сегмента 16-битное число, называемое селектором. В процессорах Intel предусмотрено 4 16-битных регистров - CS, DS, ES, SS, где хранятся селекторы. Это означает, что в любой момент можно изменить параметры, записанные в этих регистрах. Ни в каких логических и арифметических операциях эти регистры не могут участвовать. В них можно только писать и читать. Да и то, с ограничениями. Эти регистры используются для сегментирования адресов. Суть тут в следующем: в ПК, чтобы использовать полностью память требуется использовать длинные адреса. Но при использовании длинных адресов команды становятся тоже длинными. Из-за этого увеличивается размер программы. Поэтому поступают следующим образом: заносят в сегментные регистры базовый адрес сегмента, и относительно него рассчитывают относительный адрес команды или данных. Такой подход удобен при использовании нескольких сегментов. В Cs находится базовый адрес сегмента кода, в Ds - данных, в Ss - стека и в

ES – дополнительного сегмента. Особенно стоит отметить CS и SS: отвечают за сегменты двух особенных типов - сегмент кода и сегмент стека. Первый содержит программу, исполняющуюся в данный момент, следовательно, запись нового селектора в этот регистр приводит к тому, что далее будет исполнена не следующая по тексту программы команда, а команда из кода, находящегося в другом сегменте, с тем же смещением. Смещение очередной выполняемой команды всегда хранится в специальном регистре IP (указатель инструкции), запись в который также приведет к тому, что далее будет исполнена какая-нибудь другая команда. На самом деле все команды передачи управления - перехода, условного перехода, цикла, вызова подпрограммы и т. п. - и осуществляют эту самую запись в CS и IP.

Стек - организованный специальным образом участок памяти, который используется для временного хранения переменных, передачи параметров вызываемым подпрограммам и сохранения адреса возврата при вызове процедур и прерываний. Легче всего представить стек в виде стопки листов бумаги (это одно из значений слова «stack» в английском языке) — вы можете класть и забирать листы только с вершины стопки. Поэтому, если записать в стек числа 1, 2, 3, то при чтении они окажутся в обратном порядке - 3, 2, 1. Стек располагается в сегменте памяти, описываемом регистром SS, и текущее смещение вершины стека отражено в регистре SP, причем во время записи значение этого смещения уменьшается, то есть он «растет вниз» от максимально возможного адреса. Такое расположение стека «вверх ногами» может быть необходимо, к примеру, в бессегментной модели памяти, когда все сегменты, включая сегменты стека и кода, занимают одну и ту же область - память целиком. Тогда программа исполняется в нижней области памяти, в области малых адресов, и IP растет, а стек располагается в верхней области памяти, и SP уменьшается. При вызове подпрограммы параметры в большинстве случаев помещают в стек, а в BP записывают текущее значение SP. Если подпрограмма использует стек для хранения локальных переменных, SP изменится, но BP можно будет использовать для того, чтобы считывать значения параметров напрямую из стека (их смещения запишутся как BP + номер параметра).

Еще один важный регистр, использующийся при выполнении большинства команд, - регистр флагов FLAGS. В FLAGS каждый бит является флагом, то есть устанавливается в 1 при определенных условиях или установка его в 1 изменяет поведение процессора.



CF - флаг переноса. Устанавливается в 1, если результат предыдущей операции не уместился в приемнике и произошел перенос из старшего бита и если требуется заем (при вычитании), в противном случае - в 0. Например, после сложения слова 0FFFFh и 1, если регистр, в который надо поместить результат, - слово, в него будет записано 0000h и флаг CF = 1.

PF - флаг четности. Устанавливается в 1, если младший байт результата предыдущей команды содержит четное число битов, равных 1, и в 0, если нечетное. Это не то же самое, что делимость на два. Число делится на два без остатка, если его самый младший бит равен нулю, и не делится, когда он равен 1.

AF - флаг полупереноса или вспомогательного переноса. Устанавливается в 1, если в результате предыдущей операции произошел перенос (или заем) из третьего бита в четвертый. Этот флаг используется автоматически командами двоично-десятичной коррекции.

ZF - флаг нуля. Устанавливается в 1, если результат предыдущей команды – ноль

SF - флаг знака. Он всегда равен старшему биту результата.

TF - флаг ловушки. Он был предусмотрен для работы отладчиков, не использующих защищенный режим. Установка его в 1 приводит к тому, что после выполнения каждой программной команды управление временно передается отладчику (прерывание 1)

IF - флаг прерываний. Сброс этого флага в 0 приводит к тому, что процессор перестает обрабатывать прерывания от внешних устройств (см. описание команды INT). Обычно его сбрасывают на короткое время для выполнения критических участков кода.

DF — флаг направления. Он контролирует поведение команд обработки строк: когда он установлен в 1, строки обрабатываются в сторону уменьшения адресов, когда DF = 0 - наоборот.

OF - флаг переполнения. Он устанавливается в 1, если результат предыдущей арифметической операции над числами со знаком выходит за допустимые для них пределы. Например, если при сложении двух положительных чисел получается число со старшим битом, равным единице, то есть отрицательное, и наоборот.

Режимы работы процессора

В настоящее время микропроцессор с 32-битной Intel-архитектурой может работать в одном из четырех режимов: реальный режим, защищённый режим, системный режим и режим виртуального 8086.

Реальный режим (Real Mode)

После инициализации (системного сброса) центральный процессор находится в реальном режиме. В реальном режиме центральный процессор работает как очень быстрый i8086 с возможностью использования 32-битных расширений. Механизм адресации, размеры памяти и обработка прерываний микропроцессора 8086 полностью совпадают с аналогичными функциями других микропроцессоров с 32-битной Intel архитектурой в реальном режиме.

Реальный режим не поддерживает никакой многозадачности и способен выполнять только одну программу за раз. Это возвращение к тому самому вопросу о том, можно ли прервать выполнение программы в DOS – в реальном DOS так делать было нельзя. Во многих современных операционных системах отдельные программы можно закрывать по желанию, потому что каждая задача является своим отдельным процессом, запечатанным в своей области памяти со своими ресурсами, инкапсулировано. В DOS такой инкапсуляции нет – каждая программа является единственной выполняющейся и имеет полную власть над системой, поэтому хоть программу и можно завершить посреди её выполнения, это оставит систему в очень неопределённом состоянии, в котором следующая программа наиболее вероятно вылетит с ошибкой/крашнется/всё равно заставит перезапускать систему. Именно поэтому и нужно прерывание на завершение программы, потому что это выполняет все необходимые действия перед её завершением и позволяет запускать следующую, в то время как убийство процесса не приведет ни к чему хорошему, разве что краш DOSBox или даже самой системы DOS.

Режим системного управления (System Management Mode).

В новых поколениях микропроцессоров Intel, начиная с 80386 появился режим системного управления. Он предназначен для выполнения некоторых действий с возможностью их полной изоляции от прикладного программного обеспечения и даже от операционной системы: приостанавливается выполнение любого другого кода и запускается специальная программа-обработчик. В 80386 режим работал только на специальных процессорах, но позже был полностью внедрен в 80486 и Pentium, и все более современные процессоры его поддерживают. Микропроцессор переходит в этот режим только аппаратно с помощью прерываний SMI, которые не могут быть вызваны программно; к примеру, по сигналу от чипсета или периферии, Микропроцессор возвращается из режима системного управления в тот режим, при работе в котором был получен соответствующий сигнал по команде RSM. Эта команда работает только в режиме системного управления и в других режимах не распознается, генерируя исключение #6 (недействительный код операции).

Среди возможных применений SMM:

Обработка системных ошибок, таких как ошибки памяти и чипсета;

Функции защиты, например выключение процессоров при сильном перегреве;

Глубокие уровни энергосбережения;

Управление питанием — например, схемами изменения напряжения;

Эмуляция периферии, которая не была реализована на материнской плате или реализация которой содержит ошибки;

Эмуляция мыши и клавиатуры PS/2 при использовании таких же устройств с интерфейсом USB;

Централизованная конфигурация системы, например на ноутбуках Toshiba и IBM;

Обход систем защиты, встроенных в ОС;

Запуск высокопривилегированных руткитов, как было предложено на Black Hat 2008[2][3];

Защищенный режим (Protected Mode)

Защищенный режим является основным режимом работы микропроцессора. Ключевые особенности защищенного режима: виртуальное адресное пространство, защита и многозадачность. В защищенном режиме программа оперирует с адресами, которые могут относиться к физически отсутствующим ячейкам памяти, поэтому такое адресное пространство называется виртуальным. Размер виртуального адресного пространства программы может превышать емкость физической памяти и достигать 64Тбайт.

Виртуальный режим i8086 (V86)

В режим V86 процессор может перейти из защищённого режима, если установить в регистре флагов бит виртуального режима. Когда процессор находится в виртуальном режиме, его поведение во многом напоминает поведение процессора i8086. В частности, для адресации памяти используется схема <сегмент:смещение>, размер сегмента составляет 64 килобайта, а размер адресуемой в этом режиме памяти - 1 мегабайт. Виртуальный режим предназначен для работы программ, ориентированных на процессор i8086 (или i8088). Но виртуальный режим – это не реальный режим процессора i8086, имеются существенные отличия. Процессор фактически продолжает использовать схему преобразования адресов памяти и средства мультизадачности защищённого режима.

В виртуальном режиме используется трансляция страниц памяти. Это позволяет в мультизадачной операционной системе создавать несколько задач, работающих в

виртуальном режиме. Каждая из этих задач может иметь собственное адресное пространство, каждое размером в 1 мегабайт.

Все задачи виртуального режима обычно выполняются в третьем, наименее привилегированном кольце защиты. Когда в такой задаче возникает прерывание, процессор автоматически переключается из виртуального режима в защищённый. Поэтому все прерывания отображаются в операционную систему, работающую в защищённом режиме.

Обработчики прерываний защищённого режима могут моделировать функции соответствующих прерываний реального режима, что необходимо для правильной работы программ, ориентированных на реальный режим операционной системы MS-DOS.

Реальный режим

Изначально реальный режим не имел названия и стал называться таким только тогда, когда в 80286 появился новый режим, называемый защищенным, ибо создавался для защиты процессов друг от друга, чтобы не позволять процессам иметь доступ к областям памяти друг друга.

Любой, даже более современный процессор Intel, непосредственно после запуска находится именно в реальном режиме, все необходимые действия для перехода в защищенный производит непосредственно загрузчик BIOS/OS. Но об этом будет отдельный цикл лекций.

Вначале подается сигнал сброса - он удерживается до тех пор, пока не будет стабилизировано напряжение, и не дает выполнять никаких действий. Также его задача состоит в том, чтобы выполнить некоторую предустановку всех устройств, включая сам процессор (задание стартовых значений регистров, например), чтобы их работа всегда начиналась одинаково.

Выше было сказано, что процессор в реальном режиме эмулирует работу процессора 8086, он является 16-разрядным и работает с памятью до 1 Мб. Поэтому рассмотрим общую схему базирования адресов на данном процессоре. Программируя под данный процессор, обычно исходят из того, что он 16-разрядный, но каким тогда образом 16-разрядным адресом можно однозначно адресовать память, размером в 1 Мб, то есть в 2^{20} ? Так как, если в оперативной памяти имеется 2^k ячеек, то для ссылок на эти ячейки нужны k-разрядные адреса. При большом объёме памяти, большим будет и размер адресов, а это ведёт к увеличению длины команд, что в свою очередь увеличивает длину программы в целом. Это не есть хорошо. Чтобы сократить размер команд, поступают следующим образом.

Память делят на условные участки, которые принято называть сегментами. Начальные адреса сегментов (их называют базами) могут быть любыми, а вот длины сегментов не должны превосходить 2^m ячеек, где $m < k$.

Тогда, абсолютный адрес любой ячейки памяти A можно представить в виде суммы $A = B + \text{offs}$, где B – адрес базы сегмента, а offs – смещение относительно начала сегмента. Ограничение размера сегмента означает, что $0 \leq \text{offs} \leq 2^m - 1$, и поэтому, для записи смещений достаточно m разрядов. Следовательно, в сумме $A = B + \text{offs}$, большая часть адреса A приходится на базу B , а offs – это лишь небольшой добавок. Учитывая это, можно поступить следующим образом. Если в команде надо указать абсолютный адрес A , то большее слагаемое – базу B – «упрятываем» в некоторый регистр R , а в команде указываем лишь этот регистр и метшее слагаемое offs , т. Е. получится так:

КОП ... R offs ...

Это уменьшает размер команд, что мы и хотели добиться.

Повторюсь, размер памяти равен 1 Мб = 2^{20} байт ($k=20$), поэтому абсолютные адреса здесь 20-разрядные, а размеры сегментов не должны превышать величину 64Кб, т.е. 2^{16} байтов ($m=16$), поэтому-то смещения здесь и являются 16-разрядными. Общая схема базирования адресов предполагает, что размеры базовых (сегментных) регистров достаточно большие, ведь мы предполагаем, что начать сегмент можно в любом месте памяти, даже в самом конце (пусть, например, мы хотим разместить сегмент по адресу 0x77777, что вполне можем себе позволить, ведь этот адрес не превышает 1 Мб), но это противоречит тому факту, что процессор 8086 является 16-разрядным, а, значит, и разрядность его регистров тоже будет равна 16, каким же образом мы всё-таки можем тогда работать с памятью, размером до 1 Мб?

А решается это следующим образом: мы до сих пор предполагали, что сегмент можно начинать с любого адреса, но, если начинать сегмент не просто с любого адреса, а с любого адреса, кратного 16, то получится следующая картина: крайние левые 4 бита адреса начала сегмента будут нулевыми, или, последняя цифра адреса начала сегмента в hex виде будет всегда 0! То есть адреса будут иметь вид: XXXX0. А раз так, то этот крайний правый нулик можно не писать, а просто его подразумевать. Это сэкономит нам целый разряд (в hex виде) и адрес примет вид XXXX, что как раз и уместится в 16-разрядное число. В компьютере так и делается. В сегментных регистрах указывается не весь адрес, а только первые 16 битов начального адреса сегмента, т.е. первые четыре шестнадцатеричные цифры. Например, если началом сегмента будет адрес 0x12340, то в сегментном регистре будет храниться число 0x1234. Начальный адрес сегмента (без последнего 0x0) называется номером сегмента и обозначается как seg. Тогда вычисление абсолютного (физического) адреса памяти А по значению пары SR:OFF, где SR – сегментный регистр, а OFF – смещение, будет иметь следующий вид:

$$A = SR \times 16 + OFF.$$

При такой адресации адреса 0400h:0001h и 0000h:4001h будут ссылаться на одну и ту же ячейку памяти, так как $400h \times 16 + 1 = 0 \times 16 + 4001h$.

Минимальный адрес: 0000h:0000h (0 байт).

Максимальный адрес: FFFFh:FFFFh ($FFFFh \times 16 + FFFFh$) = 10FFEFh = 1 114 095 Б = 100000h + 10000h - 11h = 1 МиБ + 64 КиБ - 17 Б).

Описанный способ вычисления физического адреса позволяет адресовать (1 МиБ + 64 КиБ - 17 Б + 1 Б) памяти (диапазон адресов 0000h...10FFEFh); 1 байт добавляется для учёта байта с нулевым адресом. Однако в процессорах 8086/8088 имеется всего 20 адресных линий (металлических дорожек), поэтому размер доступной памяти составляет 2^{20} Б = 1 МиБ (диапазон адресов 0000h...FFFFFh), а при адресации выше (в диапазоне 100000h...10FFEFh) происходит переполнение («заворот») — старший единичный бит адреса игнорируется, и происходит обращение к 64 КиБ в начальных адресах (0000h...FFEFh).

Процессоры 80286 имеют 24-битовую адресную шину (возможна адресация 224 Б = 16 МиБ памяти), поэтому в них переполнения («заворот») не происходит.

Очевидно, использование реального режима накладывает значительные ограничения: адресация не больше 1 Мб памяти, сегменты размером не больше 64 Кб. Для доступа в любые другие сегменты необходимо использовать инструкции перехода, которые подразумевают смену не только регистра указателя инструкций (IP), но и сегментного регистра кода CS.

Использование

После включения питания компьютера или после подачи на процессор сигнала сброса процессор архитектуры x86 начинает работу в реальном режиме. В этом режиме начинается исполнение кода BIOS IBM-PC-совместимого компьютера. В реальном режиме может быть выполнена инициализация некоторой аппаратуры (например, инициализация контроллера ОЗУ чипсета), необходимая для работы программ после переключения процессора в защищённый режим. Если размеры кода и данных программы, выполняющей инициализацию аппаратуры, невелики, эта программа может выполняться в реальном режиме. Так, например, операционная система (ОС) DOS работает в реальном режиме и не пытается перевести процессор в защищённый режим. Ранние версии ОС Microsoft Windows могли работать только в реальном режиме. Даже ОС Windows 3.0 среди трёх режимов работы предусматривала запуск в реальном режиме и могла выполняться на процессоре 8086. Тем не менее, очевидно, что реальный режим не предназначен для многозадачных приложений, только лишь для однозадачных/однопроцессных операционных систем типа DOS.

Процессоры 80286 и более новые модели, работая в реальном режиме, в основном, имеют такие же ограничения на размер адресного пространства, как и процессоры 8086. Для использования памяти большего размера программам, разработанным для реального режима, необходимы специальные программные средства. В процессорах 80386 и более новых моделях появилась возможность, не документированная фирмой Intel и позволяющая перевести процессор в режим, неофициально названный режимом «unreal». Работая в режиме unreal, программа может, хоть и с некоторыми ограничениями, использовать 32-битовое физическое адресное пространство (2^{32} Б = 4 Гб).

Несмотря на то, что фирма Intel не предусмотрела возможность перехода процессора 80286 из защищённого режима в реальный режим, компьютер IBM PC/AT имеет такую возможность благодаря аппаратным особенностям и поддержке со стороны BIOS. Компьютер IBM PC/AT позволяет программам подавать сигнал сброса на процессор. Код BIOS может различать причины/режимы перезапуска, анализируя содержимое ячейки энергонезависимой памяти. Любая программа, записав в названные ячейки памяти подходящие значения и подав сигнал сброса на процессор, может заставить процессор перезагрузиться; после перезагрузки процессор начнёт исполнять код BIOS; код BIOS прочтёт значения из вышеуказанных ячеек памяти, не станет выполнять начальную загрузку, не станет изменять содержимое ОЗУ (в ОЗУ останутся те же данные, что и до перезагрузки процессора), и передаст управление коду, расположенному в ОЗУ по адресу, записанному в ячейку ОЗУ с определённым адресом. Таким образом программа может многократно переключаться между защищённым и реальным режимами, хотя этот способ требует сравнительно большого расхода времени на каждое переключение.

Возможность «нормального» (без ухищрений) программного перехода из защищённого режима в реальный режим была предусмотрена фирмой Intel только в процессорах 80386 и в более новых моделях. Однако в 80386 большую ценность имеет другая новая возможность — режим виртуального 8086 (V86, VM86). В режиме V86 программы могут использовать как бы прежний (сегментный) способ адресации памяти процессора 8086; при этом процессор будет находиться в защищённом режиме, а физический (линейный) адрес, вычисленный по правилам 8086, будет подвергаться страничной трансляции. Благодаря режиму V86 появилась возможность создания виртуальных

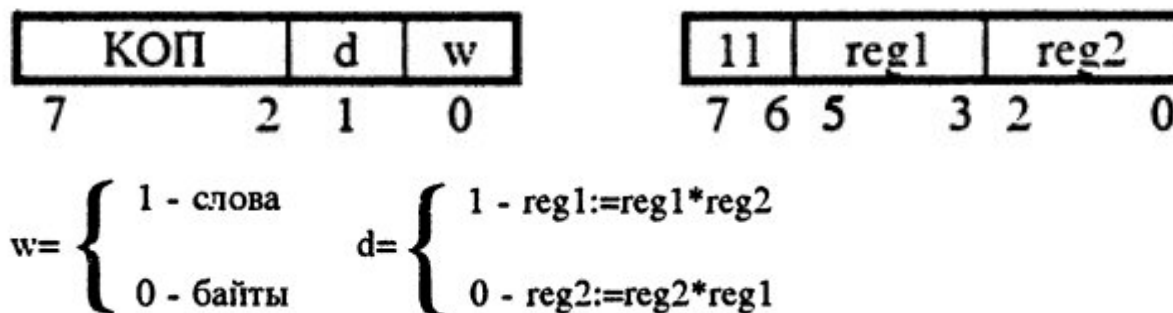
машин. Операционная система может ограничить доступ к той или иной области памяти (см. защита памяти) для каждой виртуальной машины, может выделять для них виртуальную память вместо реальной (физической) и может контролировать обращения к портам ввода-вывода. Перечисленное используется для организации работы ОС DOS под управлением многозадачных ОС вроде Microsoft Windows. При этом каждой виртуальной машине DOS доступен только 1 МиБ адресного пространства, и одновременно могут быть запущены несколько виртуальных машин DOS.

Представление команд

Машинные команды ПК занимают от 1 до 6 байтов. Код операции (КОП) занимает один или два первых байта команды. В ПК очень много машинных команд, что не хватает 256 различных кодов, чтобы их запоминать, поэтому некоторые операции объединяют в группу и им даётся один и тот же КОП, во втором же байте, это КОП уточняется. Также, во втором байте хватает места, чтобы уточнить тип операндов и их адресацию. В остальных байтах команды указываются сами операнды.

Команды ПК могут иметь от 0 до 2 операндов. Размер операндов – байт, слово или двойное слово. Операнд может быть указан в самой команде (непосредственный операнд), либо может находиться в одном из регистров ПК и тогда в команде указываются этот регистр, либо может находиться в ячейке памяти, тогда в команде указывается адрес этой ячейки памяти. Некоторые команды требуют фиксированного нахождения операндов (деление, умножение AX), тогда это операнд явно не указывается в команде.

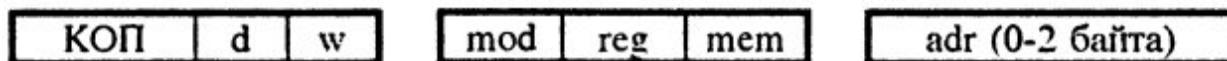
Регистр – регистр (2 байта)



Команды этого типа описывают действие $\text{Reg1} = \text{Reg1} * \text{Reg2}$ или $\text{Reg2} = \text{Reg2} * \text{Reg1}$. Задействованы регистры общего назначения. Старшие биты первого байта команды определяют какую именно команду нужно выполнить, первый бит *d* определяет куда сохранять данные, а нулевой бит *w* определяет размер операндов: слово или байт. 6-7 биты второго байта всегда равны 1, они зарезервированы. После чего идут сами операнды – регистры. Коды регистров ОН представлены 3 разрядами:

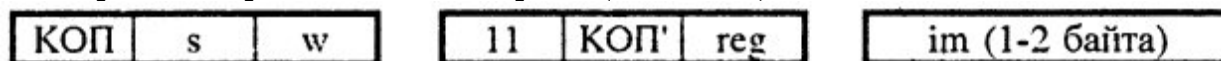
<i>reg</i>	<i>w=1</i>	<i>w=0</i>	<i>reg</i>	<i>w=1</i>	<i>w=0</i>
000	AX	AL	100	SP	AH
001	CX	CL	101	BP	CH
010	DX	DL	110	SI	DH
011	BX	BL	111	DI	BH

Регистр – память (2-4 байта)



Эти команды описывают операции $reg = reg * addr$ или $addr = addr * reg$, где reg – регистр, а $addr$ – адрес ячейки памяти. Бит w первого байта определяет размер операндов, а бит d указывает куда записывается результат. Трёхбитовое поле второго байта reg указывает операнд-регистр, двухбитовое поле mod определяет, сколько байтов в команде занимает операнд адрес (00 – 0 байтов, 01 – 1 байт, 10 – 2 байта). Трёхбитовое поле mem указывает способ модификации этого адреса.

Регистр – непосредственный операнд (3-4 байта)



$Reg = Reg \times Im$

Бит w указывает на размер операндов, а поле reg – на регистр операнд. Поле КОП в первом байте определяет лишь группу операций, в которую входит операция данной команды, уточняет операцию поле КОП' второго байта. Сам непосредственный операнд может занимать 1 или 2 байта (в зависимости от бита w), при этом операнд размером в слово записывается в команде в перевернутом виде. Ради экономии места, если операнд слово, но может быть задан в виде байта, то в первом байте бит s устанавливается в 1, бит w соответственно тоже установлен. Тогда перед выполнением операции бит автоматически расширится до слова.

Давайте рассмотрим команду «загрузить число 0x1234 в регистр AX». На языке ассемблера она записывается очень просто — `MOV AX, 0x1234`. К настоящему моменту вы уже знаете, что каждая команда представляется в виде двоичного числа. Ее числовое представление называется машинным кодом. Команда `MOV AX, 0x1234` на машинном языке может быть записана так:

0x11xx: предыдущая команда

0x1111: 0xB8, 0x34, 0x12

0x1114: следующие команды

Мы поместили команду по адресу 0x1111. Следующая команда начинается тремя байтами дальше, значит, под команду с операндами отведено 3 байта. Второй и третий байты содержат операнды команды `MOV`. А что такое 0xB8? После преобразования 0xB8 в двоичную систему мы получим значение 10111000b.

Первая часть — 1011 — и есть код команды `MOV`. Встретив код 1011, контроллер «понимает», что перед ним — именно `MOV`. Следующий разряд (1) означает, что операнды будут 16-разрядными. Три последние цифры определяют регистр назначения. Три нуля соответствуют регистру `AX` (или `AL`, если предыдущий бит был равен 0, указывая таким образом, что операнды будут 8-разрядными).

Чтобы декодировать команды, контроллер должен сначала прочитать их из памяти. Предположим, что процессор только что закончил выполнять предшествующую команду, и `IP` (указатель команд) содержит значение 0x1111. Прежде чем приступить к обработке следующей команды, процессор «посмотрит» на шину управления, чтобы проверить, требуются ли аппаратные прерывания.

Если запроса на прерывание не поступало, то процессор загружает значение, сохраненное по адресу 0x1111 (в нашем случае — это 0xB8), в свой внутренний (командный) регистр. Он декодирует это значение так, как показано выше, и «понимает», что нужно загрузить в регистр `AX` 16-разрядное число — два следующих байта, находящиеся по адресам 0x1112 и 0x1113 (они содержат наше число, 0x1234). Теперь процессор должен получить из памяти эти два байта. Для этого процессор

посылает соответствующие команды в шину и ожидает возвращения по шине данных значения из памяти.

Получив эти два байта, процессор запишет их в регистр AX. Затем процессор увеличит значение в регистре IP на 3 (наша команда занимает 3 байта), снова проверит наличие запросов на прерывание и, если таких нет, загрузит один байт по адресу 0x1114 и продолжит выполнять программу.

Если запрос на прерывание поступил, процессор проверит его тип, а также значение флага IF. Если флаг сброшен (0), процессор проигнорирует прерывание; если же флаг установлен (1), то процессор сохранит текущий контекст и начнет выполнять первую инструкцию обработчика прерывания, загрузив ее из таблицы векторов прерываний.

К счастью, нам не придется записывать команды в машинном коде, поскольку ассемблер разрешает использовать их символические имена.